MICRO2 USER'S GUIDE

REFERENCE MANUAL

AA-H531A-TE

June 1979

digital equipment corporation - maynard, massachusetts

First Printing, June 1979

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECsystem-20	TYPESET-11
ASSIST-11	RTS-8	ITPS-10
TMS-11		

CONTENTS

PREFACE

Manual Objectives

This manual provides a tutorial for the MICRO2 assembler. It introduces the MICRO2 language and gives examples of its use. The appendixes contain a reference section for the MICRO2 language, examples of the input and output files, and a listing of the error messages produced by MICRO2.

Intended Audience

This manual is intended for assembly language programmers and hardware engineers. The reader is assumed to be familiar with microprogramming and the characteristics of the architecture for which the microprograms are to be written.

Structure of this Document

This manual introduces the language constructs of MICRO2. The identification constructs are presented first. Then the process of defining fields and macros is discussed. Then microinstructions and the process of allocation is considered. Finally, conditional assembly, output listings, and the use of MICRO2 is described.

CHAPTER 1

INTRODUCTION TO MICRO2

The MICRO2 assembler converts microprograms written in its source language to absolute object code. The source language of MICRO2 allows you to define names for fields and macros and then use these names in specifying the actions to be performed by the microprogram.

The MICRO2 assembler performs two logical functions: translation and address selection. In translating names within a microinstruction to the appropriate set of bits, the assembler performs valuable syntax and validity checking. In assigning addresses, the assembler helps you lay out branches and allocate storage in an effective manner.

1.1 TRANSLATION

To construct an object microprogram, the assembler interprets a source microprogram written in a language that defines and uses names to set the appropriate bits in the microwords of the program. Names, called field-names, are defined to identify a sequence of bits within the microword. For example, bits 22 through 19 are associated with the field-name ALU by the following field-definition:

```
ALU/=<22:19>
```

Names, called value-names, are then defined to represent some or all of the possible values for the field. Value-names are specified following a field definition by a series of name and value pairs, connected by the character "=". For example, the value-names ADD, SUB, AND, OR, and A are associated with the specified values for the field ALU, as follows:

```
ALU/=<22:19>
  ADD=0
  SUB=1
  AND=2
  OR=3
  A=4
```

Then the following field-setting sets bits 22 through 19 to the value 4:

```
ALU/A
```

In addition to this basic ability to refer to fields and their values symbolically, macros can be defined to produce a notation in which the functions of the microword, not the specific field-settings, are given. For example:

```
POP STACK "MEMFNC/STKRD, ALU/A, CLKT1/YES"
```

The above macro-call within a microinstruction is equivalent to setting the three fields shown in the macro definition, but is more convenient and readable.

1.2 ADDRESS SELECTION

MICRO2 performs two types of allocation, either SEQUENTIAL or RANDOM. Further, it lets you select an address for a microinstruction or leave the address selection to the assembler. If you are using a RANDOM allocation algorithm, you can make assertions about the relationships among a set of addresses and MICRO2 will select an appropriate set of addresses.

1.3 PROGRAM STRUCTURE

The MICRO2 assembler is a line-oriented processor, which accepts a sequence of input lines written in MICRO2 source language and produces a listing file and an object module.

MICRO2 accepts continuation lines in three cases:

- Expressions
- Macro-bodies
- Microinstructions

In each case the line to be continued must have a "," as its last nonblank character.

The input to MICRO2 is a source program. A MICRO2 source program can contain one or more memories. The bit-numbering direction and the program radix apply to the entire program. The following sections describe the way in which the direction and radix are extablished. Then the fundamental program unit, the memory, is considered.

1.4 SPECIFYING THE BIT NUMBERING

The .LTOR and .RTOL keywords define the way in which the bits of your microword are numbered, so that when you define a field MICRO2 knows whether to count from the left end or the right end of the word. The form of the bit-numbering keyword line is the keyword itself, namely:

```
.LTOR
.RTOL
```

The .LTOR keyword directs MICRO2 to consider the bits of the microword numbered from left to right. The .RTOL keyword directs MICRO2 to consider the bits numbered from right to left.

For example, suppose you have a 16 bit word and you refer to the bits 12 through 15 of that word. If you specify the .LTOR keyword, then the bits are numbered from left to right as follows:

```
                        1 1 1 1 1 1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+--------------------------------+
|                        X X X X |
+--------------------------------+
```

And MICRO2 considers bits 12 through 15 as the rightmost four bits of the word. However, if you specify the .RTOL keyword, then the bits are numbered from right to left as follows:

```
 1 1 1 1 1 1
 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+--------------------------------+
|X X X X                         |
+--------------------------------+
```

And MICRO2 considers bits 12 through 15 as leftmost four bits of the word.

If no bit-numbering keyword is given, then .LTOR is assumed.

MICRO2 uses the first .LTOR or .RTOL keyword it finds to establish the direction in which the bits are numbered and ignores any subsequent bit-numbering keywords in the program.

1.5 IDENTIFYING THE RADIX

In a source program, MICRO2 interprets some numbers according to the program radix and considers other numbers to always be decimal. MICRO2 considers a number with a decimal point to be a decimal number and a number without a decimal point to be an integer. The way in which an integer is interpreted depends on the context in which it appears. The values that are interpreted according to the program radix are identified in the following chapters.

The program radix is set by either the .OCTAL or .HEXADECIMAL keywords. The form of the .OCTAL and .HEXADECIMAL keyword-line is simply the keyword itself, as follows:

```
.OCTAL
.HEXADECIMAL
```

If you include the .OCTAL keyword, any value in the program that is interpreted according to the program radix is interpreted as octal. If you include the .HEXADECIMAL keyword, such values are interpreted as hexadecimal.

The program radix can be set only once in a program. MICRO2 uses the first program radix line it finds to establish the program radix and ignores any subsequent program radix lines in the program.

1.6 MEMORIES

A program can be divided into as many as seven sub-programs, or memories. Except for the bit-numbering and program radix, which can be specified only once in a program, all other constructs are considered to belong to a memory. Each memory has its own address-space, word-width, field- and macro-definitions and microinstructions.

1.6.1 Single Memory Program

If a program contains only one sub-program (memory) then the notion of program and memory can be indistinguishable.

1.6.2 Multiple-Memories

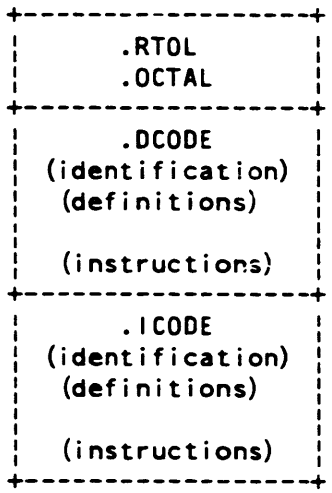
If a program contains more than one sub-program, then each sub-program is associated with a particular memory. You can have as many as seven separate sub-programs.

The beginning of a section of the memory is specified by a memory-indicator keyword. The memory-indicator keywords are as follows:

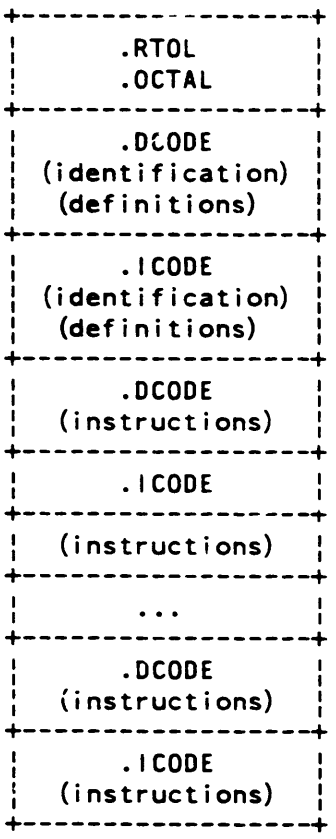
```
.UCODE
.DCODE
.ECODE
.ICODE
.OCODE
.CCODE
.MCODE
```

The identification, definitions, and instructions following that memory-indicator and up to the end of the program or another memory-indicator are associated with the specified memory.

1.6.2.1 Mixing Memories - You can switch back and forth among memories in your program. For example, suppose you have a special instruction decode memory (D) and an instruction interpretation memory (I). You can either write the program in two pieces as shown symbolically in the following diagram:



or you can group the decode and interpretation parts for each instruction as represented in the following diagram:



CHAPTER 2

PROVIDING IDENTIFICATION

MICRO2 includes directives that make the resulting program easy to access and understand. Using these directives, you can provide a title and subtitle for your microprogram, include a version number, create a table of contents, specify the paging of the program, define the number of bits in the microword, and add comments to the program.

2.1 PROVIDING A TITLE FOR YOUR PROGRAM

The .TITLE keyword-line supplies a title for MICRO2 to use as part of the heading of the output listing. MICRO2 reproduces the quoted string following the .TITLE keyword at the top of each page of your listing as part of the first line of the heading. The .TITLE keyword-line has the following form:

```
.TITLE "title-string"
```

For example, supppose you give the following .TITLE line:

```
.TITLE "XYZ Machine"
```

The title that appears as part of the heading on each page of your listing is:

```
XYZ Machine
```

A title can be specified only once in a memory. MICRO2 uses the text given in the first .TITLE keyword-line it finds and discards any subsequent .TITLE keyword-lines in the memory.

2.2 ADDING A VERSION NUMBER

The .VERSION keyword-line supplies a version number for MICRO2 to use as part of the heading of the output listing. The .VERSION keyword-line has the following form:

```
.VERSION/"version-no"
```

MICRO2 prints the version-no as part of the subtitle. The version-no can be any MICRO2 name or number. If version-no is a number, MICRO2 interprets it according to the program radix and prints it on the output listing subtitle line as a decimal number.

For example, suppose you give the following .VERSION line:

```
.VERSION/"10"
```

If the program radix is octal, MICRO2 includes the version number "8" in the subtitle.

2.3 ADDING A TABLE OF CONTENTS

The .TOC keyword-line supplies a subtitle and adds an entry to the table of contents. MICRO2 reproduces the text given in quotes following the .TOC keyword as part of the second line of the page heading of the output listing.

Suppose you add the following .TOC keyword-line to your program:

```
.TITLE "XYZ Machine - Version 1B"
.TOC"Introductory Remarks"
```

Then the subtitle that appears on the next page of your listing is:

```
Introductory Remarks
```

MICRO2 reproduces the text from the .TOC keyword-lines at the front of your listing to provide a table of contents for the listing. You can create an attractive table of contents by indenting the text on the .TOC lines to indicate subordinate topics as shown in the output listing of the sample program given in Appendix B.

You can add blank lines to format your table of contents by including .TOC keywords that have a null string as text. For example, suppose you add the following .TOC keywords to your program:

```
.TOC "Introductory Remarks"
...
.TOC ""
.TOC "Field Definitions"
...
.TOC ""
.TOC "Macro Definitions"
```

The table of contents in your output listing is double spaced by the null strings as follows:

```
;    TABLE OF CONTENTS

;1   Introductory Remarks
;6
;7   Field Definitions
;24
;25  Macro Definitions
```

The number that appears in each table of contents line is the line number assigned by MICRO2 to that line in the output listing. Using this line number, you can quickly locate the place in the listing where the referenced information appears.

2.4 PAGING YOUR PROGRAM

The .PAGE keyword-line indicates a new listing page and, optionally, provides a table of contents entry and a subtitle.

To simply indicate a new page, you include the .PAGE keyword without any text string, as follows:

```
.PAGE
```

MICRO2 starts a new page in the output listing and places the .PAGE keyword on the first line of that page following the heading.

To start a new page, add a subtitle, and make an entry in the table of contents, you add a text string to the .PAGE keyword-line, as follows:

```
.PAGE "text-string"
```

MICRO2 starts a new page, using the text string given as the subtitle and includes the text string in the table of contents.

A .PAGE keyword with a text-string is operationally equivalent to a .TOC keyword with that text string followed by a .PAGE keyword without a text string.

You can use as many .PAGE keywords in the program as you wish. A microprogram that is paged so that each topic appears on a new page is easy to read initially and convenient to reference at a later time.

.5 SPECIFYING THE MICROWORD WIDTH

The .WIDTH keyword-line specifies the number of bits in the microword. A .WIDTH keyword-line consists of the keyword .WIDTH, followed by a slash and the number of bits in the word, as follows:

```
.WIDTH/number-of-bits
```

The number given for number-of-bits is always interpreted by MICRO2 as a decimal number. Thus to define a microword that consists of 64 bits, you include the following line in your microprogram:

```
.WIDTH/64
```

The maximum value that can be given for number-of-bits is 128.

MICRO2 uses the first .WIDTH keyword-line it finds to establish the width of the microword for the memory and it rejects any subsequent .WIDTH keyword-lines in the memory.

If you don't specify the width, MICRO2 deduces the word width from the field definitions. It assumes the width of the word to be the value formed by adding one to the largest bit position specified in the set of field-definitions.

2.6 COMMENTS

Comments can be included anywhere in the program. A comment begins with a ";" character and ends at the end of the line.

You can enter full line comments by starting the line with a ";" character. You can add comments to lines by following the last character on the line by some spaces and the ";" character.

For examples of comments, see the sample program in Chapter 9.

CHAPTER 3

DEFINING FIELDS AND VALUES

The set of bits within the microword that make up a particular field can be associated with that field name. Further, a set of qualifiers can be defined that add semantic meaning to the field and a set of value-names can be defined to set the field.

The simplest form of a field-definition defines a name for a field as follows:

```
ALPHA/=<6:0>
```

The above definition associates the field name ALPHA with bits 6 though 0 of the microword.

The addition of value-definitions permits the field to be set as well as referenced symbolically. For example:

```
ALPHA/=<6:0>
  A1=1
  A2=2
  A3=3
```

The above definitions associate the value-name A1 with the value 1, A2 with 2, and A3 with 3 and make the following field-setting possible:

```
ALPHA/A2
```

The above field-setting sets the field A (bits 6 through 0) of the microword to the value A2 (2).

The addition of qualifiers permit default settings, parity adjustment, and the like. The following sections describe the process of defining field-names and value-names.

3.1 DEFINING NAMES FOR FIELDS

A field-definition consists of the name followed by the separator '/=' followed by the position of the bits within the word to be associated with the field name followed by a list of one or more qualifiers. The form is:

```
field-name /= < left-bit:right-bit > { , qualifier ... }
```

The field-name can be any valid MICRO2 name. The rules for forming MICRO2 names are given in Section 3.2.

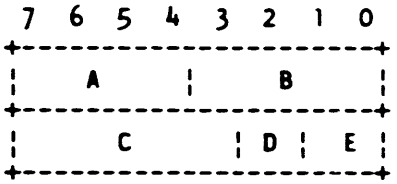
The left-bit and right-bit are decimal numbers that identify the beginning and end bits of the field in the microword. If you specified right-to-left bit numbering, then the left-bit must be greater than or equal to the right-bit. If you specified left-to-right bit numbering, then the left-bit must be less than or equal to the right-bit.

Qualifiers are described in Section 3.3.

Consider the following field-definitions:

```
.RTOL
.OCTAL
A/=<7:4>
B/=<3:0>
C/=<7:3>
D/=<2>
E/=<1:0>
```

These definitions identify the fields of the microword as shown in the following diagram:



As you see, you can define overlapping fields, like A and C. MICRO2 knows, for each bit of the microword both its value (0 or 1) and its status (unset or set). MICRO2 warns you if you try to define the state of a bit more than once.

3.2 NAMES

MICRO2 allows a name to be made up of characters from the following set:

```
A B C ...Z Upper case letters
a b c ... z Lower case letters
0 1 2 ... 9 Numbers
!  Exclamation mark
#  Hash mark
&  Ampersand
(  Left parenthesis
)  Right parenthesis
<  Left angle bracket
>  Right angle bracket
*  Asterisk
+  plus sign
-  minus sign
.  period
?  question mark
_  Underscore
   Space and tab
```

MICRO2 considers all the following to be valid names:

```
A12
!!!
2+3*6
B*C?
```

A name can begin with the period character, but such names make the program confusing for another programmer to read. Further, a future version of MICRO2 might use the name you chose as a keyword. Keyword names are reserved. You cannot use a keyword as a name and, therefore, your program would not function correctly under the new version of MICRO2.

A name can begin with a number if it contains characters within it to distinguish it from a number.

For example, the following are all valid names if the program radix is octal:

```
12Q3
4*2
51F
```

However, if the program radix is hexadecimal, the first two are valid names but the third is a hexadecimal number.

3.2.1 Spaces in Names

In addition to the above characters, MICRO2 permits the use of spaces or tabs within a name. MICRO2 considers a space within a microword as a concept rather than as a particular sequence of characters. Thus, MICRO2 considers the following names to be the same although the space in each case is made up of a different character sequence.

```
A B
A  B
A   B
A    B
A  B
A    B
```

However, the symbol shown above is not the same as the symbol AB, which does not have a space within it.

As an example of the effective use of spaces in names, consider the following macro from the sample program given in chapter 9.

```
T1 <-- PC + 1
```

If this name is entered, by mistake, with two spaces between the 1 and the <, MICRO2 still recognizes it as the same macro. That is, the following are the same:

```
T1 < -- PC + 1
T1  < -- PC + 1
```

3.2.2 Upper and Lower Case

MICRO2 interprets upper and lower case letters as the same. Thus, the following symbols are all treated as the same symbol.

```
ABC
ABc
...
abc
```

3.2.3 Length Limitation

MICRO2 imposes a limit of 128 characters on the length of a name. Further, because MICRO2 is a line-oriented processor, a name must fit, with the other necessary information, on a line.

3.2.4 Uniqueness of Names

MICRO2 names are classified as follows:

field-names
value-names
macro-names
expression-names

Field-names, macro-names, and expression-names must be unique with respect to all other names in the same class for a given memory.

Value-names, however, need be unique only with respect to other value names for the same field definition. For example, the following set of value names is valid:

```
A/=<7:4>
  A=0
  B=1
  C=2
B/=<3:0>
  A=2
  B=4
  C=6
```

Since you must always qualify a value name by its associated field-name, MICRO2 can easily distinguish the value-names. That is, MICRO2 interprets A/B as 1 and B/B as 4.

A label is a value-name for the field designated as the address field. Only one field in a memory can be designated as an address field and thus all labels in a given memory must be unique.

Communication between memories can be accomplished by value names. A discussion of field and value names in different memories is given in Section 8.1. The rule is that if the same field name is defined in more than one memory, then the value-names defined for that field in any memory are known in all memories. Thus, if you define the address field to have the same name in several separate memories, then all the labels in those memories must be unique with respect to all other labels in the separate memories.

3.3 QUALIFIERS

Qualifiers are used to establish a default for a field, to identify the field as one that can contain a label, to designate a field to be used for a parity bit, and to associate the setting of a field with the condition of other fields within the microword. The qualifiers are given in the following list:

```
.DEFAULT = default-expression
.ADDRESS
.NEXTADDRESS
.FLOATEPARITY
.FLOATOPARITY
.VALIDITY = validity-expression
```

The following sections consider each of these qualifiers.

3.3.1 The .DEFAULT Qualifier

The .DEFAULT qualifier specifies a value that MICRO2 can use for a field when you do not explicity set the field.

As you will see in Chapter 6, "Microinstructions," MICRO2 forms a microword by starting with a word of bits each of which has a value of zero and a status of unset. Then, it translates field-settings given in the microinstruction to set the bits of the fields. When MICRO2 processes a field-setting, it changes all the bits of the field. It changes the value of each bit to a zero or one and the status of each bit from unset to set. MICRO2 uses the default for a field if, and only if, no bit of the field is explicitly set.

If a bit of a field is set as a result of an overlapping field being set, then the default for that field is not applied.

If you do not set a field and do not indicate a default, then MICRO2 does not set that field and the bits continue to have the value zero and the status unset.

The default-value is an expression. The rules for forming a MICRO2 expression are given in section 4.1. The simplest form of an expression is a number. The following examples use only numbers for default-values. The use of expressions in defaults is illustrated in the sample program in Chapter 9.

3.3.1.1 An Example - To illustrate the use of the .DEFAULT qualifier, suppose we add some defaults to the fields defined in Section 3.1 as follows:

```
.RTOL
.OCTAL
A/=<7:4>,.DEFAULT=1
B/=<3:0>,.DEFAULT=2
C/=<7:3>,.DEFAULT=3
D/=<2>
E/=<1:0>
```

Now consider the cases given in the following sections.

3.3.1.2 Case 1: Simple Default - Suppose we set only the A field in the microinstruction, as follows:

```
A/2
```

MICRO2 first sets the bits in field A to the value 2. Then it considers the defaults in the order specified. It ignores the default for field A, since that field is explicitly set. It uses the default for field B since the status of every bit in that field is unset.

```
 7 6 5 4 3 2 1 0
+---------------+
|0 0 1 0 0 0 1 0|
+---------------+
```

After MICRO2 uses the default to set the bits in field B, all bits are set and MICRO2 does not consider any further defaults.

3.3.1.3 Case 2: Defaults and Overlapping Fields - If more than one default can be applied to a sequence of bits, MICRO2 chooses the default specified first in the microprogram.

In the example given above, the fields A and C overlap and both fields have defaults. Suppose we set only the D field, as follows:

```
D/1
```

MICRO2 sets the bit in field D to 1. MICRO2 then applies the first default in the microprogram, the default value for A, to set bits 7 through 4. MICRO2 does not use the default for B because a bit in that field is set (bit 2) or the default for C because bits 7 through 4 are set by the default value for A.

3.3.2 The .ADDRESS and .NEXTADDRESS Qualifiers

MICRO2 requires that the jump field be identified by either an .ADDRESS or .NEXTADDRESS qualifier. A field defined with the .ADDRESS or .NEXTADDRESS qualifier can be set to the value of any label in the program.

In addition to designating the associated field as a jump field, the .NEXTADDRESS qualifier specifies that the default for the field is the value of the address associated with the next microinstruction given in the program.

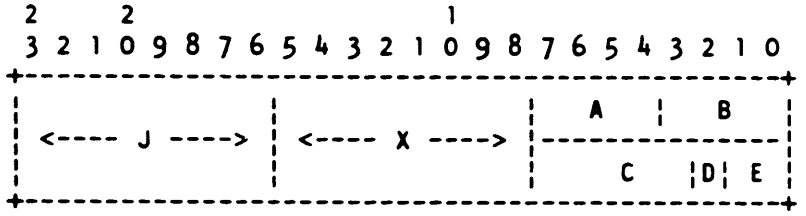
Since the .NEXTADDRESS qualifier serves the dual purpose of identifying a field as an address field and supplying the default for that field, you must not use a .DEFAULT qualifier in combination with a .NEXTADDRESS field.

One field and only one field in a memory can be designated as the jump field by the use of either the .ADDRESS or the .NEXTADDRESS qualifier.

To illustrate the .NEXTADDRESS qualifier, suppose we add definitions for fields X and J to the field definitions given for the discussion of the .DEFAULT qualifier, as follows:

```
.RTOL
.OCTAL
A/=<7:4>,.DEFAULT=1
B/=<3:0>,.DEFAULT=2
C/=<7:3>,.DEFAULT=3
D/=<2>
E/=<1:0>
X/=<15:8>
J/=<23:16>,.NEXTADDRESS
```

The definitions describe the following microword:



Suppose the following sequence of microinstructions are given for the above definitions:

```
0201:
  L1:  A/1,B/1,J/L2
0203:
  L2:  A/2,B/1
0204:
  L3:  A/3,B/3
0206:
  L4:  A/4,B/4
```

The field J, which has the .NEXTADDRESS qualifier, is set explicitly in the first microinstruction (L1) to the label L2, which is the octal value 0203. In the other microinstructions, J is not set and thus MICRO2 provides the default setting of the address of the next microinstruction. In the microinstruction labelled L2, J is set to the octal value 0204. Similarly, in the microinstruction labelled L3, J is set to 0206 (octal).

The fields X and J each contain eight bits. The field J, having the .NEXTADDRESS qualifier, can be set to the value of a label; the X field, however, can only be set to a value. For example, suppose the label L1 is allocated to the value 0212. We can set both X and J to the value 0212, as follows:

```
X/212,J/L1
```

Or we can write:

```
X/212,J/212
```

But we cannot set the field X to the label L1.

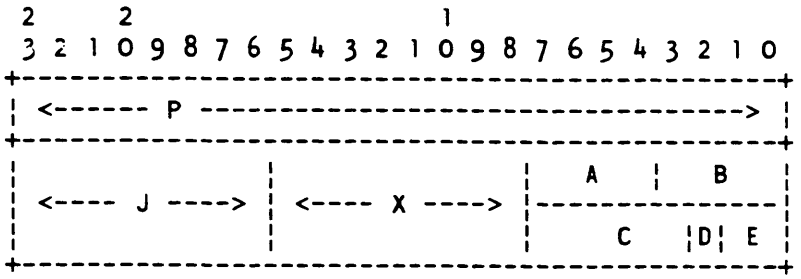
3.3.3 The .FLOATOPARITY and .FLOATEPARITY qualifiers

The parity qualifiers, .FLOATEPARITY and .FLOATOPARITY, designate a field for MICRO2 to search to find an unset bit to use as a parity bit. When you add this qualifier to a field definition, MICRO2 examines the field after all the values are set and the defaults are applied and uses the first unset bit it finds in that field, searching from left to right, to adjust the parity to even or odd. If you specify the .FLOATEPARITY, MICRO2 adjusts the parity to even parity if it is not already even; if you specify .FLOATOPARITY, MICRO2 adjusts to odd parity. Even parity means that the number of 1's in the word is an even number. Odd parity means that the number of 1's is an odd number.

Suppose we add the following field definition to the set we have been developing:

```
P/=<23:0>,.FLOATOPARITY
```

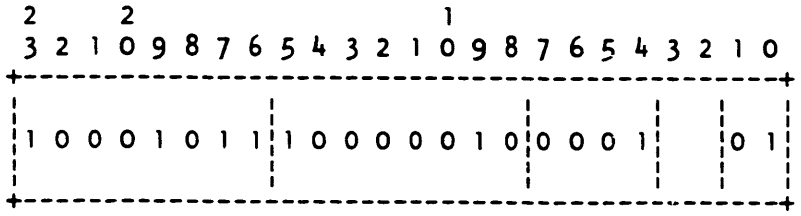
The microword now looks as follows:



Suppose we set field E to 1, X to the octal value 202, and J to the value of LAB3 in a microinstruction as follows:

```
LAB2:  E/1,X/202,J/LAB3
```

Assuming LAB3 is the octal value 213, the microword contains the bits shown in the following diagram before the parity is adjusted:

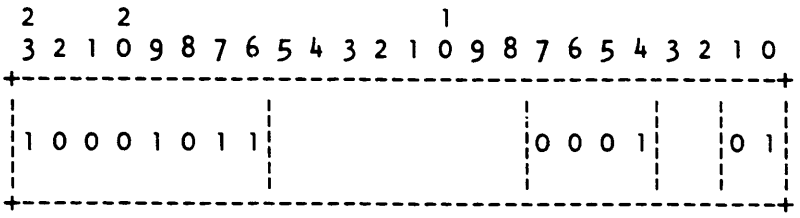


In the above microword, fields J, X, and E are set explicitly and field A is set by default. That is, all bits except bits 3 and 2 are set either explicitly or by default. The .FLOATOPARITY qualifier associated with the field P causes MICRO2 to search from left to right within that field (that is starting from bit 23) to find the first unset bit if the parity of the word is not odd. The first unset bit is bit 3 and MICRO2 sets that bit to 1 to turn the parity of the word to odd parity.

Suppose we do not set X. That is, we write:

```
LAB2:  E/1,J/LAB3
```

The microword before parity is:



When MICRO2 searches the field P from left to right, the first unset bit it finds is bit 15. Therefore, MICRO2 sets bit 15 to 1 to adjust the parity to odd parity.

3.3.4 The .VALIDITY Qualifier

The .VALIDITY qualifier lets you make assertions about the conditions under which a field can be legally set. The .VALIDITY qualifier associates a validity expression with a field. If the validity expression is not satisfied when the field is set in a microword, MICRO2 produces a warning message.

The .VALIDITY qualifier and use of the validity expressions is a deep subject, which is described in detail in Chapter 4.

3.4 VALUE NAMES

A value-name is a name associated with a particular value of a particular field.

A value-name is defined by a value-definition following the field definition. A value-definition consists of the value-name followed by the separator '=' followed by the value to be equated with that name when the name is used to set the field. In addition, a value-definition can specify one or two values, which are added into two separate counters, .TIME1 and .TIME2, when the field value name is used to set the field. The value-definition can also have its own .VALIDITY expression. Thus, the form is:

```
value-name=value,t1-value,t2-value,.VALIDITY=exp
```

The t1-value, t2-value, and .VALIDITY expression are all optional. In the simplest case, a value name is associated with a value for a field as follows:

```
A/=<7:4>,.DEFAULT=1
  A0=0
  A1=1
  A4=4
```

In the above example, three value-names are defined for field A, namely A0, A1, and A4. These names can be used in a microinstruction to set A as follows:

```
LAB:   A/A1
```

Normally field-names and field-value names have more mnemonic content than the ones given above. For example, consider the following excerpt from the sample program given in chapter 9:

```
SETCC/=<15>,.default=0

  NOP  = 0          ;Do nothing with condition codes
  SET  = 1          ;Set condition codes according
                    ;to IR value and status of ALU.
```

3.4.1 Counter Values

If the value-definition includes a t1-value, then that value is added into the counter .TIME1. If the field-value-definition includes a t2-value, then that value is added into .TIME2. The resulting values of .TIME1 and .TIME2 can then be used in validity expressions. Chapter 4 describes the use of these counters in validity expressions.

For example, suppose we associate a t1-value with the value-name A1, a t2-value with the value-name B1, and both a t1 and t2-value with the value-name A4, as follows:

```
A1 = <7:4>,.DEFAULT=1
     A0 = 0
     A1 = 1,3
     A4 = 4,2,6
B = <3.:0>,.DEFAULT=2
     B0 = 0
     B1 = 1,,8
```

If we use the value-name A1 in a microinstruction, the counter .TIME1 is increased by the associated t1-value, 3. If we use both A4 and B1, then .TIME1 is increased by 2 (for A4) and .TIME2 is increased by 6 (for A4) and 8 (for B1). Chapter 4 describes the use of these counters in validity expressions.

CHAPTER 4

EXPRESSIONS AND VALIDITY CHECKING

Validity expressions allow the addition of semantic meaning to a definition language by indicating the conditions under which it is invalid to use a particular field or value name.

When a field or value name that has an associated validity is used in a microinstruction, MICRO2 evaluates the expression. If the value of the expression is 1 (true), then MICRO2 takes no action. However, if the value of the expression is 0 (false), MICRO2 issues a warning message. In practice, any value that is not 1 is considered to be a false value.

The following sections consider expressions in general and then the ways in which a validity expression can be associated with a field or value name.

4.1 EXPRESSIONS

An expression in MICRO2 is enclosed in angle brackets. An expression can be any of the following:

- A number
- An expression name
- A function call
- A field value name
- A field contents indicator
- A predefined symbol

An expression can occupy more than one line as long as the last character of each line to be continued is a comma character. For an example of a continued expression, see Section 4.1.7.

The following sections consider each of these cases in detail.

4.1.1 Numbers

MICRO2 recognizes integers or decimal numbers. An integer is interpreted according to the program radix. A number with a decimal point is always interpreted as a decimal number. You set the program radix by adding either the .OCTAL or the .HEXADECIMAL keyword to your program. If you do not give a program radix, then an octal radix is assumed.

Some special cases occur in the interpretation of integers, as follows:

- If the program radix is octal and an integer contains an 8 or a 9 digit, then the number is interpreted as a decimal number.
- MICRO2 interprets a character sequence beginning with a letter as a name. Thus, if the program radix is hexadecimal and the first digit in a number is a letter, then you must write the integer with an initial zero to prevent its being interpreted as a name. For example:

```
F12     interpreted as a name
0F12    interpreted as an integer
```

4.1.2 Expression-Names

An expression-name is defined by the .SET keyword as follows:

```
.SET/expression-name = <expression>
```

For example:

```
.SET/SWITCH1 = <1>
```

MICRO2 associates the value 1 with the expression-name SWITCH1. You can use the expression-name in subsequent expressions. For example:

```
.SET/SWITCH2 = <SWITCH1>
```

The .SET keyword-line is described in detail in connection with conditional assembly capability in Section 10.2.

4.1.3 Function Calls

MICRO2 provides functions for comparison, arithmetic, and Boolean operations. Also, MICRO2 provides functions to detect parity, shift, and select a case from a set of choices.

The functions are given in the following table:

Function	Value
Comparison	
.EQL[op1,op2,...]	1 if op1=op2=...
.NEQ[op1,op2,...]	1 if op1<>op2 and op2<>op3 and ...
.GTR[op1,op2,...]	1 if op1>op2>...
.GEQ[op1,op2,...]	1 if op1>=op2>=...
.LSS[op1,op2,...]	1 if op1<op2<...
.LEQ[op1,op2,...]	1 if op1<=op2<=...
Arithmetic	
.MAX[op1,op2,...]	Value of largest operand
.MIN[op1,op2,...]	Value of smallest operand
.SUM[op1,op2,...]	op1+op2+...
.PROD[op1,op2,...]	op1*op2*...
.DIFF[op1,op2]	op1-op2
.QUOT[op1,op2]	op1/op2 (truncated)
.MOD[op1,op2]	remainder of op1/op2
Boolean	
.NOT[op]	Boolean complement of op
.AND[op1,...]	Boolean 'and' of operands
.OR[op1,...]	Boolean 'or' of operands
.XOR[op1,...]	Boolean 'xor' or operands
.NAND[op1,...]	Boolean complement of the 'and'
.NOR[op1,...]	Boolean complement of the 'or'
.EQV[op,...]	Boolean complement of the 'xor'
Miscellaneous	
.PARITY[op1,op2,...]	If operands contain an even number of 1's, then 1 else 0
.SHIFT[op1,op2]	If op2 is positive, then shift op1 left op2 places else shift op1 right op2 places
.CASE[op1]OF[op2,...]	The (op1-th + 1) operand of the list. That is, if op1 is 0, the first op2 is used. Up to 32 choices can be given.
.SELECT[{op1,op2,}...]	The first op2 for which op1 is true

The operands of a function can be expressions. Thus, you can construct very complex expressions. Some examples of expressions are given in Section 4.1.7.

.1.4 Value-names

Value-names are the names that you associate with a value for a given field, as described in Section 3.4.

Since a value-name is only defined for a specific field, it must be qualified by the field-name when you use it in an expression as follows:

```
field-name/value-name
```

For example, suppose we define the following field- and value-names:

```
A/=<7:4>
  A0=0
  A1=1
  A4=4
```

Then, the value-names can be used in an expression as follows:

```
<.EQL[<A/A1>,<X>]>
```

The above expression is equivalent to:

```
<.EQL[1,<X>]>
```

In the above expressions, X is an expression name.

4.1.5 Field Contents Indicators

The contents of a field can be designated in an expression by giving the field-name followed by a slash. For example, to find out if the current contents of field B contains the value 4, you write the following expression:

```
.EQL[<B/>,<4>]
```

For some uses of field contents indicators, see Section 4.2 on "The .VALIDITY qualifier."

4.1.6 Predefined Symbol Names

Three symbols are predefined in MICRO2, as follows:

Symbol	Meaning
.	The address of the current microinstruction
.TIME1	The value of the counter associated with t1-values.
.TIME2	The value of the counter associated with t2-values.

The .TIME1 and .TIME2 predefined symbols are described in section 3.4 in connection with the definition of value-names.

As an example of the use of these symbols, consider the following example:

```
A/=<7:4>,.DEFAULT =2
  A0=0,10,10
  A1=1,10
  A4=4,10
B/=<3:0>,.DEFAULT =2
  B0=0,5,8
  B1=1,2,12
X/=<15:8>.ADDRESS,.VALIDITY=<.LSS[.TIME1,14]>
```

The definition of A0 specifies that its use to set field A in a microinstruction adds 10 to the .TIME1 counter and 10 to the .TIME2 counter; the definition of A1 specifies that its use adds 10 to the .TIME2 counter; and so on.

Now suppose we have the following microinstructions:

```
LAB1:  A/A0,B/B0,X/L2

LAB2:  B/B1,X/L3
```

After all the fields are set in the first microinstruction, labelled LAB1, the counter .TIME1 contains the value 15 and the counter .TIME2 contains the value 18. When MICRO2 evaluates the validity expression for the field X, it finds that the value of .TIME1 is greater than 14 and thus the expression is false. MICRO2, therefore, reports an error.

After all the fields are set in the second microinstruction, .TIME1 contains 2 and .TIME2 contains 12. The validity expression is true and no error is reported.

4.1.7 Examples

Now let's consider some examples of expressions. Suppose we want to produce a true result if the contents of field A and the contents of field B both equal 1. That is, if the following is true:

```
<A/>=<B/>=1
```

We use the .EQL function a follows:

```
.EQL[<A/>,<B/>,1]
```

To find out if the contents of field A and field B equal 1 and the contents of field C and field D do not equal 1, we use the .EQL, .NEQ, and .AND functions as follows:

```
.AND[<.EQL[<A/>,<B/>,1]>,<.NEQ[<C/1>,1,<D/>]>]
```

To perform the computation:

```
<A/>+<B/>+2*<C/>
```

We use the .ADD and .PROD functions as follows:

```
.ADD[<A/>,<B/>,<.PROD[2,<C/>]>]
```

To choose a value based on the value of the predefined name .TIME1, we use the .CASE function as follows:

```
.CASE[<.TIME1>]OF[10,2,6,21]
```

If the value of .TIME1 is 0, then the function value is 10. If the value of .TIME1 is 1, then the function value is 2. And so on. If the value of .TIME1 is greater than 3, an error is reported.

If the length of the expression exceeds a line or if formatting the expression will aid its readability, we can continue the expression on several lines.

For example, we can write:

```
OPCODE/=<15:8>,.VALIDITY=<CASE[VAL]OF[ADD,
                                      SUB,
                                      MULT,
                                      DIV]
```

4.2 VALIDITY EXPRESSIONS

The .VALIDITY qualifier associates an expression with a field- or value-name. The form of the .VALIDITY qualifier is:

```
.VALIDITY = expression
```

MICRO2 evaluates validity expressions after all the fields explicitly given in the microinstruction have been set and after any defaults are applied. If the value of that expression is true (1), then MICRO2 takes no action. However, if the value of that expression is false (0), then MICRO2 prints a warning message.

4.2.1 Associating A Validity Expression With A Field

Suppose we assign a .VALIDITY qualifier to field A, in the following set of field definitions, as follows:

```
.RTOL
.OCTAL
A/=<7:4>,.DEFAULT=1,.VALIDITY=<.EQL[<B/>,3]>
B/=<3:0>,.DEFAULT=2
```

The validity expression in the above example asserts that setting field A to a value is legal only if the value of field B is 3. If field A is set when field B contains any other value, an error message is reported. Let's consider some cases:

Microinstruction	Microword	Comment
A/1,B/3	00010011	B is explicitly set to 3, thus setting A is valid.
A/1	00010010	B is set to 2 by default thus setting A is invalid and a warning message is reported.
A/1,B/2	00010010	B is explicitly set to 2, thus setting A is invalid and a warning message is reported.
B/1	00010001	A is not explicitly set, thus the validity expression is not evaluated.

4.2.2 Associating a Validity expression with a Value-Name

A value-name definition can contain a validity expression. Suppose we add a validity expression to the set of field definitions given above.

```
A/=<7;4>,.DEFAULT =2
  A0=0,10,10
  A1=1,0,10,.VALIDITY=<.EQL[B/,0]>
  A4=4,12
B/=<3:0>,.DEFAULT =2
  B0=0,5,8
  B1=1,2,12
```

```
    X/=<15:8>.ADDRESS,.VALIDITY=<.LSS[.TIME1,14]>
```


Then consider the following cases:

Microinstruction	Result
A/A1,B/B0	Validity expression associated with value-name A1 is true because field B contains 0.
A/A1	Validity expression associated with A1 is false because field B contains the default value 2.
A/A0,X/2	Validity expression associated with field name X is true because .TIME1 contains 10, which is less than 14.
A/A4,B/B0	Validity expression associated with X is false because .TIME1 contains 17.

As another example, suppose we add a validity expression to both a field definition and a value name definition as follows:

```
A/=<7;4>,.DEFAULT =2,.VALIDITY=<.LSS[.TIME1,.TIME2]>
  A0=0,10,10
  A1=1,0,10,.VALIDITY=<.EQL[B/,0]>
  A4=4,10
B/=<3:0>,.DEFAULT =2
  B0=0,5,8
  B1=1,2,12
  B2=2,20
X/=<15:8>.ADDRESS,.VALIDITY=<.LSS[.TIME1,14]>
```

Now consider the following cases:

Microinstruction		Result
LAB5:	A/A1,B/B0	.TIME1 contains 5 and .TIME2 contains 18. Thus both validity checks are satisfied
LAB6:	A/A1,B/B1	The validity expression associated with the field value name A1 is false; therefore, an error is reported
LAB7:	A/A1,B/B2	Neither validity check is satisfied; therefore, two errors are reported

CHAPTER 5

DEFINING MACROS

The macro capability of MICRO2 permits the definition of a representation for a microprogram at a higher level than the basic field-value pairs. Once the fields of your microword are defined, a set of macros that set groups of fields appropriately for certain operations can be specified.

To define a macro, you write the macro-name followed by a quoted string that contains the macro-body. When you use the macro-name in a microinstruction, then MICRO2 replaces the name by the macro-body.

The simplest case of a macro definition is one that does not contain any parameters. For example:

```
M1 "A/A0,B/B0"
```

Writing the name M1 in a microinstruction is equivalent to writing the pair of field settings. That is, the following are equivalent:

```
L1:  M1          L1:  A/A0,B/B0
```

The following sections describe the process of defining and using macros.

5.1 THE MACRO-NAME

Macro-names are formed using the set of characters given in Section 3.2. In addition to these characters, MICRO2 recognizes square bracket pairs and commas in macro-names as indicators of the number and position of the macro parameters.

A macro with one parameter contains an empty square bracket pair in a macro-name. For example, each of the following macro-names requires one parameter.

```
ABC[]
A[]BC
[]ABC
```

A macro with two parameters contains either two square bracket pairs or a comma within a single square bracket pair, as follows:

```
ABC[][]
AB[]C[]
[]ABC[]
ABC[,]
A[,]BC
[,]ABC
```

The only limitation on the number of parameters in a macro-name is the length of the line. The macro-call with all its arguments must fit on one line.

The number and position of parameters in a macro-name are an integral part of the name. Thus the macro-names given above all define different macros. That is, the macro ABC[][] is not the same as ABC[,] and so on.

5.2 THE MACRO-BODY

The macro-body consists of any combination of field-settings and macro-calls separated by commas. When a macro is used in a microinstruction, MICRO2 replaces the macro-name by the macro-body associated with that name. Consider a macro defined in the sample program in chapter 9:

```
Pop Stack "MEMFNC/STKRD, ALU/A, CLKT1/YES"
```

Writing the macro-call Pop Stack in a microword is equivalent to writing the three field settings given in the macro-body. In Chapter 9, we will see how the definition of a macro language facilitates the process of writing and understanding a microprogram.

5.2.1 Continuing a Macro Definition

Macro definitions can be continued to the next line. If the last non-blank character of a line in the macro-body is a "," (comma) character, MICRO2 assumes that the next line is a continuation line. For example, the above macro could have been defined using three lines as follows:

```
Pop Stack "MEMFNC/STKRD,
          ALU/A,
          CLKT1/YES"
```

5.2.2 Nested Macro Definitions

Since a macro-body can contain a macro-call, any number of macro languages can be constructed, one upon another. You can, for example, start with a primitive set of macro definitions. Then using these primitive definitions, add another, more sophisticated, level of macros, and so on.

5.3 THE MACRO-CALL

The macro-call is the macro-name with actuals filling the indicated parameter positions.

The macro-call for a macro without parameters is the macro-name itself. The macro-call for a macro with parameters includes the actuals within the square brackets of the macro name.

MICRO2 first replaces a macro-call by the macro-body. If the macro-body contains macro-calls, MICRO2 replaces those calls. MICRO2 continues in this way until the microinstruction contains only field-settings.

To illustrate the way in which MICRO2 replaces macro-calls, suppose we define the following three macros:

```
M1  "A/A0,M2,B/B0"
M2  "C/C0,M3"
M3  "D/D0"
```

Then we use the macro-name M1 in a microinstruction:

```
L1:  X/L2,M1
```

MICRO2 first replaces the macro-name M1 by its associated macro-body, as follows:

```
L1:  X/L2,A/A0,M2,B/B0
```

Then MICRO2 replaces M2 by its associated macro-body:

```
L1:  X/L2,A/A0,C/C0,M3,B/B0
```

Then MICRO2 replaces M3:

```
L1:  X/L2,A/A0,C/C0,D/D0,B/B0
```

The above microinstruction consists of only field settings and so no further replacement can occur.

5.3.1 Parameters

Square brackets and commas indicate parameters in the macro-name. The character "@" followed by a decimal integer in the macro-body indicates the position of the parameter in the macro-body. This character pair is called a parameter-designator.

The decimal integer in the parameter-designator refers to the position, numbering from left to right, of the parameter in the name.

If you use only one parameter, then you designate it by '@1' in the macro-body.

Suppose we define the following macro:

```
M4[] "A/A0,B/@1"
```

Then we use it in the following microinstructions:

```
L1:  C/C0,M4[B0]

L2:  M4[B1]
```

MICRO2 expands the macro call as follows:

```
L1:  C/C0,A/A0,B/B0

L2:  A/A0,B/B1
```

If you use several parameters, then you designate the leftmost parameter in the macro-name as '@1', the one to the right of that as '@2', and so on.

Suppose we define a macro with four parameters:

```
M5[,]AB[]C[] "A/@1,B/@2,C/@3,D/@4
```

Then we use it as follows:

```
L1:  M5[A1,B1]AB[C2]C[D3]
```

MICRO2 replaces the macro by its associated macro-body:

```
L1:  A/A1,B/B1,C/C2,D/D3
```

5.3.2 Multiple Use Of Parameter Designator

You can use a parameter designator more than once in a macro body. For example, to set the fields A, B, and C to the same value, you can specify a macro that has that value as a parameter:

```
SET ABC[] "A/@1,B/@1,C/@1"
```

Then to set all the fields to zero, you write:

```
SET ABC[0]
```

5.3.3 Parameter Designators as Field-Values or Macro-Calls

Parameter-designators can be used within the macro-body as either field-values or macro-calls. If the parameter-designator is used as a field value, then the corresponding parameter must be either a value or a field-value-name. Similarly, a parameter-designator used as a macro call must be filled by a parameter that is a macro-call.

For example, consider the following macro-definition:

```
M5[,] "A/@1,@2"
```

When the above macro is called, the first parameter must be a value or field-value-name and the second parameter a macro-call. Suppose we have the following additional macro-definitions:

```
M6 "B/B1"
M7[]+[] "C/@1,D/@2"
```

The following calls on M5 produce the indicated results:

Call	Expansion
M5[1,M6]	A/A1,B/B1
M5[1,M7[C1]+[D2]]	A/A1,C/C1,D/D1

5.3.4 Too Many or Too Few Parameters

As described in Section 5.1, MICRO2 considers the position and the number of the parameters as part of the name. If you supply too many or too few arguments in a macro-call, MICRO2 detects the error, unless you have defined another macro that, by coincidence, the macro-call satisfies.

Suppose you define the following macro:

```
M2[,] "A/@1,B/@2,D/1"
```

Then, if you call it with the proper number of arguments (2), you get the valid replacement, as follows:

```
M2[A1,B1]  is replaced by  A/A1,B/B1,D/1
```

However, suppose you mistakenly call the macro with three parameters, as follows:

```
M2[A1,B1,1]
```

MICRO2 does not recognize the above as a valid call and produces an error message.

However, suppose you define another macro with three parameters, so that you have:

```
M2[,]    "A/@1,B/@2,D/1"
M2[,,]   "A/@1,D/@2,E/@3"
```

Then, the following calls are valid:

Call	Expansion
M2[A1,B1]	A/A1,B/B1,D/1
M2[A1,1,2]	A/A1,D/1,E/2

When you use names that are the same except for the number of macros, you lose the error checking performed by MICRO2. Such naming is, therefore, not recommended.

CHAPTER 6

MICROINSTRUCTIONS

The microinstructions describe the processing to be performed by the microprogram. These microinstructions are expressed in terms of the field and macro-names defined.

For each microinstruction, MICRO2 translates names into the appropriate sequence of bits and creates the associated microword.

The following sections consider the microinstruction and the formation of the microword.

6.1 THE MICROINSTRUCTION

The microinstruction contains the information MICRO2 needs to set the bits of the microword.

Both the address and label can be omitted.

A microinstruction begins with an absolute address assignment, one or more labels, or both. Following this optional information, a sequence of field-settings and/or macro-calls is given separated by commas.

That is, the form of the microinstruction is:

```
address:
{ label: }
    ...
          { field-setting }  ,...
          { macro-call    }
```

A microinstruction is different from all other MICRO2 language constructs because it can occupy several lines.

6.1.1 Continuing a Microinstruction

If a microinstruction occupies more than one line, the separator character ',' must be as the last non-blank character of all lines except the last line. For purposes of this discussion, the end of the line is assumed to be either the ';' character, which begins a comment, or the actual end of line. Thus the last non-blank character of a line means the last non-blank before the ';' or end of line

When MICRO2 finds a comma as the last non-blank character of a line, it continues the microinstruction using the information on the next line. When MICRO2 finds a line that does not end with a comma, it concludes the microinstruction and produces the microword.

Suppose we write the following portion of a microprogram:

```
A/A0,B/B1,
C/C2,
J/L1
```

MICRO2 interprets the above as a single microinstruction that sets fields A, B, C, and J.

However, suppose we omit the terminating commas:

```
A/A0,B/B1
C/C2
J/L1
```

MICRO2 interprets the above as three microinstructions and produces a microword for each.

6.1.2 Allocating a Microinstruction

The following microinstruction contains an address (612) and two labels (R and Q):

```
0612:
R:
Q:
   A/A1,B/B1
```

MICRO2 produces the microword and, allocates it to word 612, and associates the specified absolute octal value 612 with the labels R and Q.

Both the address and label are optional. Suppose you write:

```
R:
     A/A1,B/B1
```

For this case, MICRO2 produces the same microword but chooses the allocation for the word and associates that address with the label R.

When MICRO2 processes the microinstruction, it creates a microword and assigns that microword to an address. The following sections consider these two activities in detail.

6.2 THE MICROWORD

MICRO2 creates a microword in the following way:

1. MICRO2 clears the counters .TIME1 and .TIME2 and begins with a word of the specified length in which each bit has a value of zero and a status of unset.

2. MICRO2 then fills in all the fields that are explicitly set in the microinstruction.

3. Then, MICRO2 sets any fields that have an associated default and that contain only unset bits.

4. Then, MICRO2 evaluates any VALIDITY expressions.

5. Finally, MICRO2 performs any parity adjustment indicated.

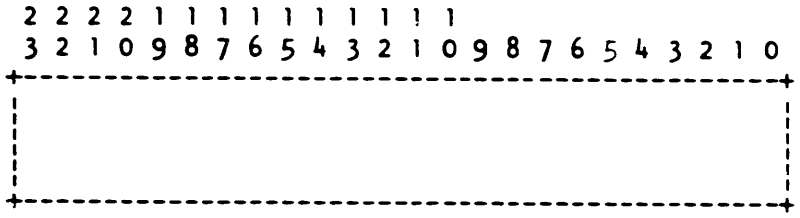
Let's look at the creation of a microword. First, consider the source program:

```
.TITLE "TEST"
.OCTAL
.RTOL
.WIDTH/24
.REGION/0200,0277

A/=<7:4>,.DEFAULT =2
  A0=0,10,10
  A1=1,10,.VALIDITY=<.EQL[<.TIME1>,<.TIME2>]>
  A4=4,10
B/=<3:0>,.DEFAULT =2
  B0=0,5,8
  B1=1,2,12
X/=<15:8>.ADDRESS,.VALIDITY=<.LSS[<.TIME1>,14]>
```

```
J/=<23:16>,.NEXTADDRESS
P/=<23:0>,.FLOATEPARITY

M1[] "A1/@1"
M2[]+[] "A/@1,B/@2"

.TOC "MICROCODE"

A: M1[A1]
```

Now, consider the creation of the microword for the microinstruction labelled A.

1. MICRO2 initializes the counters .TIME1 and .TIME2 and begins with a word consisting of 24 unset (zero) bits.

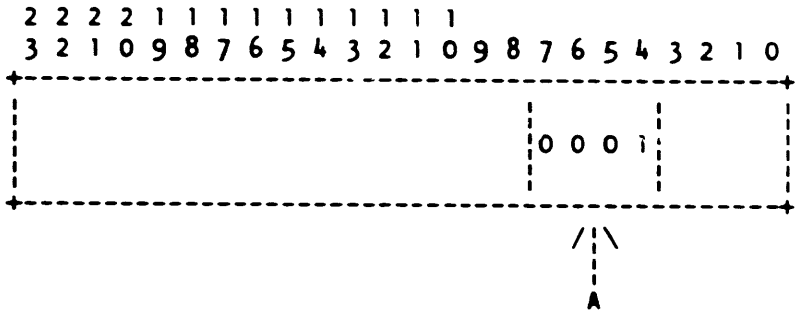


2. MICRO2 expands all macros until the source consists of a set of field settings and then sets those fields.

   Expanding the microinstruction labelled A above produces the following field setting:

```
A:  A/A1
```

   MICRO2 then sets the field A (bits 7 through 4) to the value A1(1). The microword now contains the following values:

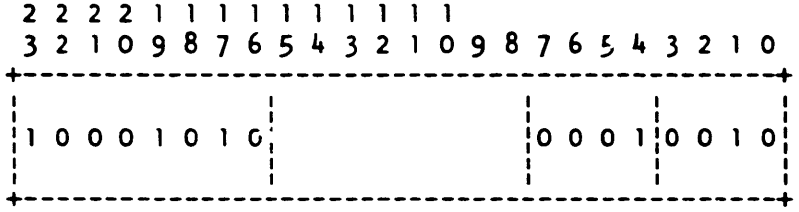


   In setting field A to the value designated by A1, MICRO2 adds the counter values associated with A1 into the counters .TIME1 and .TIME2. After setting field A1, the counter .TIME1 contains the value 10 and the counter .TIME2 still contains 0.

3. Next MICRO2 applies defaults. Fields A and B have defaults. Field A is explicitly set, but no bit of field B is set, so MICRO2 sets field B to its default value (2).

   The field, J, designated as the next address is not explicitly set, so MICRO2 sets it to the value of the next microinstruction. Suppose the next microinstruction address is 0212 (octal).

   The microword now contains the following bits:



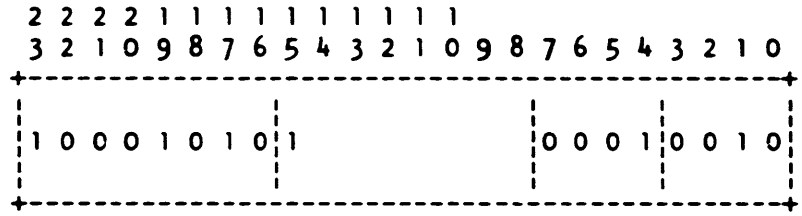
4. Next, MICRO2 evaluates validity expressions. The definitions associate validity expressions with value-name A1 and field X. Field X is not set in this microinstruction, so the validity expression associated with X is not evaluated.

   A warning message is produced because the validity expressions for the value-name A1 is not satisfied since the value of .TIME1 is 10 and the value of .TIME2 is 0.

5. Finally, MICRO2 adjusts the parity in any fields designated as FLOATOPARITY or FLOATEPARITY. Field P is designated as a field to be adjusted for even parity.

   The word at this point has odd parity, so MICRO2 searches from left to right in the field P to find the first unset bit, which is bit 15. MICRO2 sets that bit to 1 to adjust to even parity.

   The microword now contains the following bits:



Bits 14 through 8 contain the value 0 and have the status unset.

CHAPTER 7

ALLOCATION

The address space for the microprogram is established by the .REGION keyword and the method of allocation by the .RA' DOM and .SEQUENTIAL keywords.

7.1 DEFINING THE ADDRESS SPACE

The .REGION keyword determines the address-space. The .REGION keyword is followed by one or more pairs of address limits, as follows:

```
.REGION/low-bound,high-bound...
```

Low-bound and high-bound are expressions whose values are interpreted according to the program radix.

For example, if you want your microprogram to be allocated in the address space that begins with the address 0200 and ends with the address 0277, you specify the following region.

```
.REGION/0200,0277
```

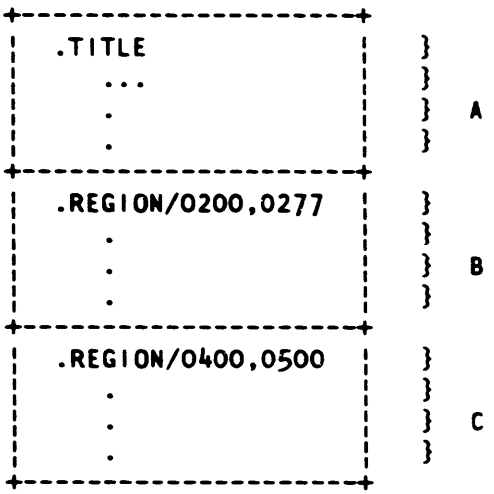
If you want your microprogram to be allocated in an address space that consists of a set of disjoint address ranges, you specify the address limits for each range. For example:

```
.REGION/0302,0320/0400,0461/0200,0207
```

You can specify any number of .REGION keywords. MICRO2 allocates the microinstructions following a .REGION up to the next .REGION keyword (or the end of memory) in the specified address space.

If you do not give a .REGION at the beginning of your memory, MICRO2 assumes that the address space begins at 0 and ends at MAXPC, the highest available address for the given architecture.

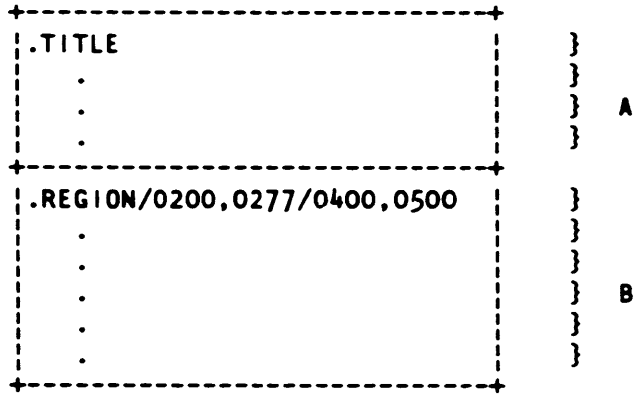
Consider the following program fragment:



The microprogram above has three logical parts from the allocation point-of-view: Part A resides in the address range 0000-MAXPC; part B resides in the range 0200-0277; part C in the range 0400-0500.

If MICRO2 uses all the available addresses in 0400-0500, it can not in this case use any addresses left in the range 0200-0277.

Now consider a microprogram that has a single REGION keyword:



This microprogram has two logical parts. Part A resides in address range 0000-MAXPC and Part B resides in address range 0200-0277 and 0400-0500. MICRO2 does not begin allocating in the address range 0400-0500 until it has used all the available addresses in the range 0200-0277. In the previous example, MICRO2 starts allocating in the range 0400-0500 when it finds the REGION keyword that begins logical part C.

MICRO2 considers each range specified individually, even though two ranges may be adjacent or overlapping. For example, suppose you define the following address space:

```
.REGION/0100,0177/0200,0277
```

MICRO2 treats the above as specifying an address space that consists of two separate ranges.

7.2 SPECIFYING THE METHOD OF ALLOCATION

Within the specified address space, you can select either sequential or random allocation.

7.2.1 Sequential Allocation

In sequential mode, MICRO2 allocates a microinstruction by taking the address of the previous microinstruction and adding 1.

MICRO2 begins allocating with the first address in the address space defined by the .REGION keyword and continues incrementing until it reaches either an absolute address assignment or the end of the address range.

When it reaches an absolute address, MICRO2 uses that address for the associated microinstruction and as the new base for incrementation.

When MICRO2 uses the last instruction in an address-range, it chooses the first address in the next address-range for the next microinstruction. After MICRO2 uses the last address in the last range, it uses the address 0000 and issues an error message for each word allocated following the last legal allocation.

Suppose we use sequential allocation and specify the address space with a single absolute assignment as follows:

```
.OCTAL
.REGION/0202,0207/0312,0316
.SEQUENTIAL
L1:  M1[A1]
L2:  M1[B2]
0206:
L3:  M1[3]
L4:  M1[4]
L5:  M1[5]
```

The addresses are allocated starting at the beginning of the region (0202) and continuing sequentially until the absolute assignment, as follows:

```
0202    L1
0203    L2
0204    (unused)
0205    (unused)
0206    L3
0207    L4

0312    L5
0313    (unused)
0314    (unused)
0315    (unused)
0316    (unused)
```

Observe that Addresses 0204 and 0205 will not be allocated unless we force the allocator back with an absolute assignment.

7.2.2 Random Allocation

In random mode, you can specify absolute address assignments and constraints. A constraint selects a set of addresses based on the low order bit configuration. Constraints are described in detail in the next section.

MICRO2 first allocates all absolute assignments and constraints and then allocates the remaining microinstructions starting at the first unallocated address in the first address-range and continuing sequentially through the unallocated addresses of the address space.

Suppose we use random allocation instead of sequential for the preceding example as follows:

```
.OCTAL
.REGION/0202,0207/0312,0316
.RANDOM
L1:  M1[A1]
L2:  M1[B2]
0206:
L3:  M1[3]
L4:  M1[4]
L5:  M1[5]
```

MICRO2 allocates the absolute address assignment first and then begins at the first address in the address space and increments through the space to produce the following allocation:

```
0202    L1
0203    L2
0204    L4
0205    L5
0206    L3
0207    (unused)

0312    (unused)
0313    (unused)
0314    (unused)
0315    (unused)
0316    (unused)
```

7.2.2.1 Constraints - Many microprogrammable microprocessors perform conditional branching by ORing some logic function into the low order bit position of the next microinstruction address. MICRO2 provides a constraint capability for generating a set of addresses for conditional branching.

A constraint consists of an "=" character followed by a constraint string composed of a sequence of 0 and 1 characters.

A constraint specifies a set of addresses. In response to a constraint string, MICRO2 chooses a base address that satisfies the low order bit configuration specified by the constraint. The bits of an address are always ordered from right to left. So the low order bit is the right-most bit.

MICRO2 then assigns the next n microinstructions to the addresses formed by systematically increasing the base address counting only in those bits designated as 0's in the constraint string.

For example, suppose the constraint string is:

```
=0101
```

And suppose MICRO2 chooses the base address 0225 (octal). The binary representation of this address is:

```
1 0 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 1 0 1 0 1
```

The low order bit configuration of the chosen address (0225) satisfies the constraint (0101).

MICRO2 systematically increases the base address by counting in bits 1 and 3, to produce a set of addresses ending in the following bits:

```
0101      (base address)
0111
1101
1111
```

Since we are assuming that MICRO2 chose 0225 for the base address, then the following addresses are assigned:

```
0225
0227
0235
0237
```

MICRO2 can choose any available address in the address space that has the low order bit specified by the constraint. MICRO2 uses the first available address that satisfies the constraint completely. That is, MICRO2 uses the first address for which the base address and the addresses formed by oring the constraint possibilities with the base address are all available.

7.2.2.2 Indicating a bit that can be 0 or 1 - In addition to 0's and 1's, you can use the character '*' in a constraint string. This character informs MICRO2 that it can select an address that has either a 0 or a 1 in that position for the base address.

For example, the constraint 0101 specifies that the low order bit configuration must be 0 or 1, whereas the constraint string 0*01 specifies that the low order bit configuration can be either 0101 or 0001.

7.2.2.3 The size of the address set - The number of microinstructions in the set, n, is determined by the number of zeroes in the constraint string, as follows:

```
n=2**X
```

Where X is the number of 0's in the constraint string.

For example, the constraint 0101 has two zeroes and thus determines a set of four microinstructions.

7.2.2.4 Constraints Within Constraints - If MICRO2 encounters a constraint string within the set of instructions it is allocating to the block of addresses associated with an outer constraint string, it skips to the next address satisfying the inner constraint and then proceeds according to the algorithm specified by the outer constraint. The purpose of the nested constraint is to skip over some addresses that would otherwise be allocated by the outermost constraint.

For example, suppose we have a constraint and a sequence of eight microinstructions labelled L0, L1, and so on, as follows:

```
=000
L0:
L1:
L2:
L3:
L4:
L5:
L6:
L7:
```

If MICRO2 chooses the base address 0200, then it allocates the microinstruction labelled L1 to 0200, L2 to 0201, L3 to 0202 and so on, as follows:

Microinstructions		Allocation
=000		
L0	--->	0200
L1	--->	0201
L2	--->	0202
L3	--->	0203
L4	--->	0204
L5	--->	0205
L6	--->	0206
L7	--->	0207

Now suppose we insert an inner constraint as follows:

Microinstructions		Allocation
=000		
L0: M1[0]	--->	0200
L1: M1[1]	--->	0201
=011		
L2: M1[2]	--->	0203
L3: M1[3]	--->	0204
L4: M1[4]	--->	0205
L5: M1[5]	--->	0206
L6: M1[6]	--->	0207
L7: M1[7]		

The inner constraint directs MICRO2 to skip the 0202 assignment. Thus the total number of addresses within the constraint is seven and the instruction labelled by L7 is outside the constraint. Since MICRO2 can choose any address for L7, it allocates it after it has satisfied all the constrained addresses.

Now suppose we insert another inner constraint:

```
Microinstructions          Allocation

   =000
   L0:  M[0]        --->      0200
   L1:  M[1]        --->      0201
   =011
   L2:  M[2]        --->      0203
   L3:  M[3]        --->      0204
   =110
   L4:  M[4]        --->      0206
   L5:  M[5]        --->      0207
   L6:  M[6]
   L7:  M[7]
```

The microinstructions labelled by L6 and L7 are now outside the constraint.

7.2.2.5 Terminating a Constraint - A null constraint within the scope of the constraint terminates the constraint. A null constraint is the "=" character. A constraint can also be terminated by an absolute address assignment; however, in this case, MICRO2 issues a warning message.

Suppose we want to constrain only five addresses.

```
Microinstructions          Allocation

   =000
   L0:  M[0]        --->      0200
   L1:  M[1]        --->      0201
   L2:  M[2]        --->      0202
   L3:  M[3]        --->      0203
   L4:  M[4]        --->      0204
   =
   L5:  M[5]
   L6:  M[6]
   L7:  M[7]
```

The null constraint terminates the constraint after the microinstruction labelled by L4.

7.2.2.6 Address Space Boundaries - The set of constrained addresses allocated by MICRO2 must lie within an address-range. Suppose you define the following address space:

```
.REGION/0100,0177
```

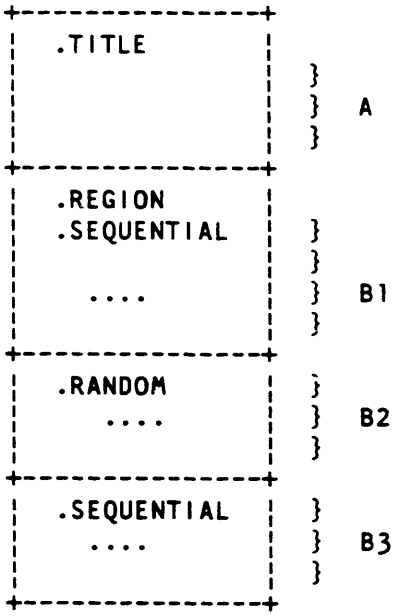
And suppose further, that MICRO2 uses the addresses 0166, 0167, 0176, 0177 to satisfy a constraint. If you define your space as:

```
.REGION/0100,0167/0170,0177
```

Then that set of addresses can no longer be used as a set because addresses 0166 and 0167 belong to one range whereas addresses 0176 and 0177 belong to another.

7.3 MIXING ALLOCATION MODES

You can switch between the two allocation modes in your program, as shown in the following program fragment:



In the above example, the .RANDOM and .SEQUENTIAL keywords divide the address space into 4 sub-spaces, according to the method of allocation.

If you do not give an allocation mode, MICRO2 assumes .RANDOM. Thus in the above example, part A is allocated in the random sequential.

CHAPTER 8

COMMUNICATION

This chapter considers two forms of communication: communication among memories in the same program and communication among separate programs.

8.1 MEMORY COMMUNICATION

Each memory has its own definitions, identification, and address-space. However, if the same field-name is defined in more than one memory, then the value-names defined for that field in any memory are known in all other memories that define the field. This feature lets you communicate between memories.

For example, suppose you define two memories and designate the same field-name as the address field. You can then access the labels of one memory in the other as shown in the following example:

```
        .OCTAL
        .RTOL
        .UCODE
           A/=<12:10>
           J/=<9:0>,.ADDRESS
L1:     A/1,J/L2
L2:     A/2,J/L3
L3:     A/3,J/L4
        .DCODE
           B/=<9:0>
           J/=<16:10>,.ADDRESS
L4:     B/0,J/L1
```

The label name L1 is a value name for the field J, which is defined in memory U. You can jump from the DCODE memory to the UCODE memory by specifying the label L1.

8.2 PROGRAM COMMUNICATION

You can assemble separate programs and load them together in a control store by handling address space assignment and communication. If, for example, you wish to have n separate programs, you divide the control store into n+1 logical spaces, namely:

- Communication Space
- Space for Program 1
- Space for Program 2
  ...
- Space for program n

Suppose you want to assemble two separate programs for a control store of 2000 words. You might reserve the first 10 words for communication and then specify the address space for program 1 as:

```
.REGION/11,1200
```

After you successfully assemble that program, you will know exactly what space it needs and you can specify the space for program 2. Suppose the last word program 1 uses is 1053. You can specify the address space for program 2 as:

```
.REGION/1054,1777
```

Or you can reserve a little extra space for later expansion of program 1 and specify the address space for prgram 2 as

```
.REGION/1100,1777
```

When both your programs are successfully assembled, you assemble the communication region:

```
    .REGION/0,7
     J.=<9:C-,.ADDRESS
0:  J/1400              ;Transfer to program 2
1:  J/762               ;Transfer to program 1 (1)
2:  J/21                ;Transfer to program 1 (2)
```

You can, of course, dispense with the communication space and perform the transfers directly by fixing certain addresses in each program.

CHAPTER 9

A SAMPLE MICROPROGRAM

In this chapter we define a sample machine and then give a microprogram that performs two subroutines, one to add the top two values of a stack and one to use the stack to determine the program control.

We begin by defining the data path of the model mach ne and specifying the microinstruction fields. Then, we discuss the parts of the microprogram. First, we consider the identification part, then the field definitions, then the macro language, and finally the two subroutines.

The input and output listings for the microprogram are given in Appendix B.

9.1 THE DATA PATH

The data path of the sample machine is shown in Figure 9-1.

This architecture, while very simple, has the essential features of a machine, namely:

- o ALU - An arithmetic logic unit
- o PC - A program counter, which points to next instruction to be execu :d
- o IR - An instruction register, which contains the current instruction
- c MA - A memory address register
- o An internal stack for data
- o Input and Output Buses

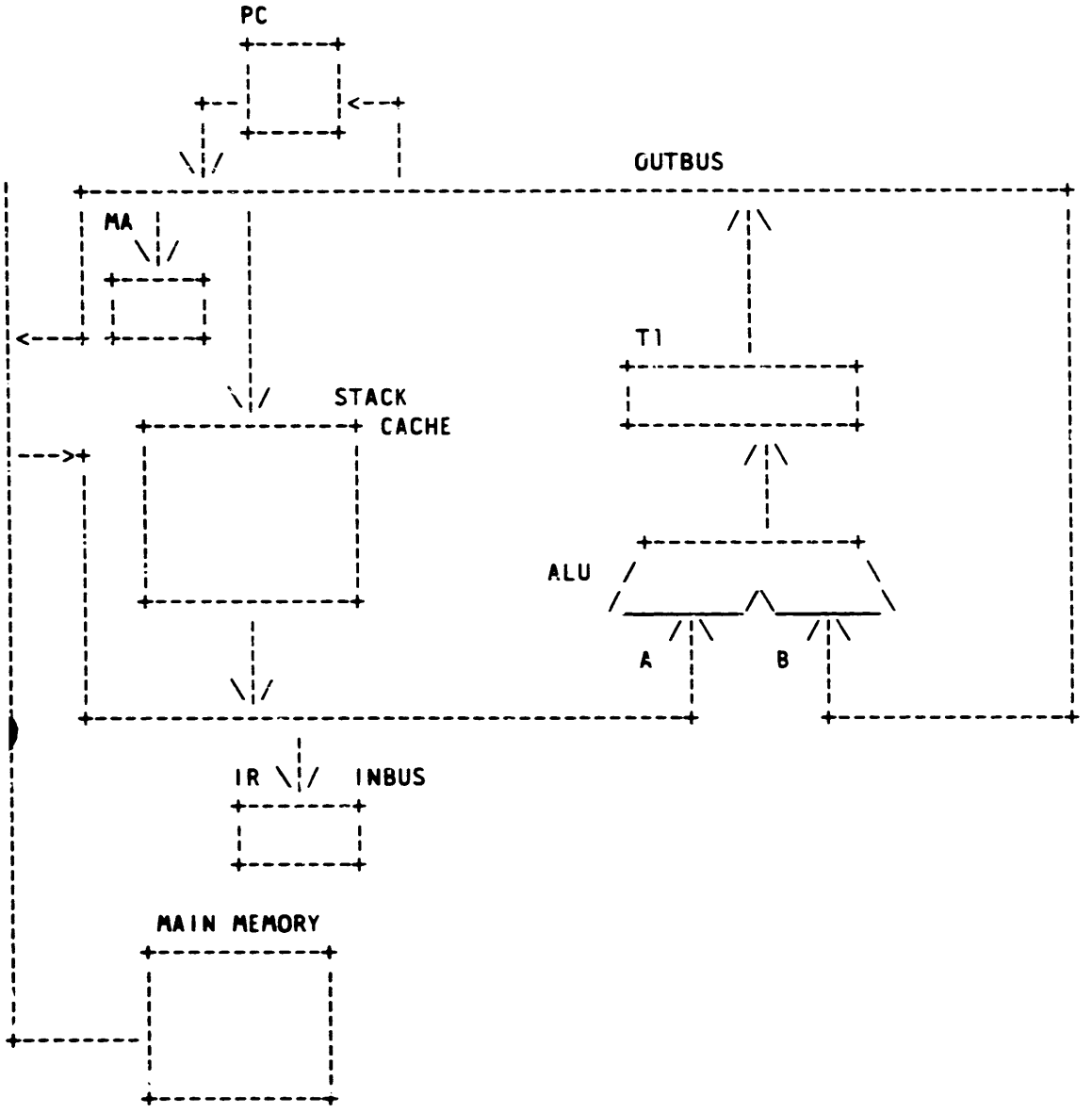


Figure 9-1. Model Machine Data Path

The following sections specify the identification, define the fields of the microword for this data path, develop a macro language in terms of these fields, and use this language. The input and output listings for this program are given in Appendix B.

9.2 IDENTIFICATION

The identification part of the program for the sample machine defines the bit-numbering as right-to-left, the base as octal, the allocation mode as random, and the word width as 23.

9.2.1 Program Excerpt

Here is the identification part of the program:

```
        .title "machine model"
        .rtol
        .octal
        .random
        .width/23
```

9.3 FIELD DEFINITIONS

The sample machine has the following microword:

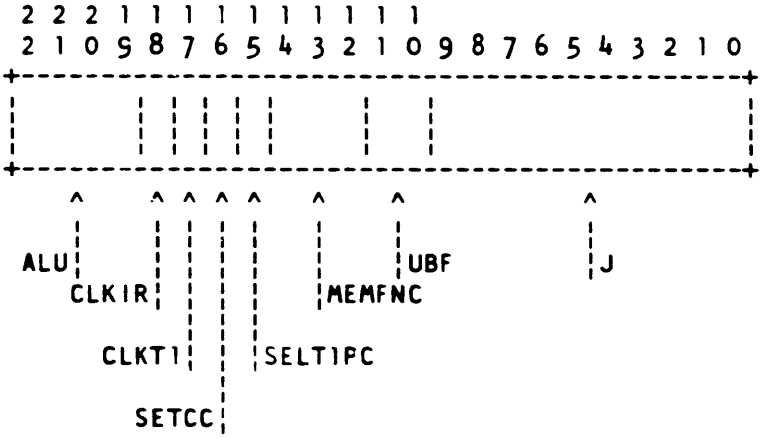


Figure 9-2. Model Machine Microword

9.3.1 Program Excerpt

Here are the field definitions that define this word:

```
;
;  The following expression is used
;  to make sure that T1 and PC are
;  not both clocked at the same time.
;
.SET/T1.PC=<.CASE[<CLKT1/>]OF[1,0]

.toc "Field Definitions"


.toc "   ALU function for this microcycle"

ALU/=<22:19>,.default=<ALU/Zero>

  ADD = 0               ;Addition  A + B
  SUB = 1               ;Subtraction  B - A
  AND = 2               ;Logical A AND B
  OR  = 3               ;Logical  A OR B
  A   = 4               ;Pass A through ALU
  B   = 5               ;Pass B through ALU
  A+1 = 6               ;Increment A  A + 1
  Zero= 7               ;Zero value


.toc  "   IR Control"

CLKIR/=<18>,.default=<CLKIR/NO>

  NO   = 0              ;Do not clock IR
  YES  = 1              ;Load IR with data bus


.toc  "   T1 Control"

CLKT1/=<17>,.default=<CLKT1/NO>

  NO   = 0              ;Do not change T1
  YES  = 1              ;Load T1 with ALU output


.toc  "   OUTBUS Control"

SELT1PC/=<16>,.default=<SELT1PC/T1>

  T1  = 0               ;Tri-state T1 onto OUTBUS
  PC  = 1               ;Tri-state PC onto OUTBUS
```

```
.toc  "   Condition Code Control"

SETCC/=<15>,.default=<SETCC/NOP>

  NOP  = 0            ;Do nothing with condition codes
  SET  = 1            ;Set condition codes according
                      ;to IR value and status of ALU.


.toc  "   Memory Request Control"

MEMFNC/=<14:12>,.default=<MEMFNC/NOP>

  NOP     = 0         ;No operation.
  STKRD   = 1         ;Read top of stack onto INBUS.
  STKWR   = 2         ;Write data on OUTBUS to stack.
  LOADMA  = 3         ;Load Memory Address with OUTBUS.
  READ    = 4         ;Read onto INBUS using MA.
  WRITE   = 5         ;Write OUTBUS value using MA.
  READPC  = 6         ;INBUS gets PC.
  WRITEPC = 7,.VALIDITY=<T1.PC> ;PC gets OUTBUS.


.toc  "   Branch Control"

UBF/=<11:10>,.default=<UBF/JUMP>

  JUMP  = 0           ;Use next address field unchanged.
  ALU   = 1           ;Four way branch
                      ;  00 - Less than zero.
                      ;  01 - Largest negative number.
                      ;  10 - Greater than zero.
                      ;  11 - Zero
  INSTR = 2           ;Instruction decode.
  THREE = 3           ;Reserved.


.toc  "   Next Address Field"

J/=<9:0>,.nextaddress
```

9.4 MACRO DEFINITIONS

The macros that define the language for the machine are divided into five categories; namely:

- Stack interactions
- Memory interactions
- Arithmetic actions
- Branch tests
- Miscellaneous functions

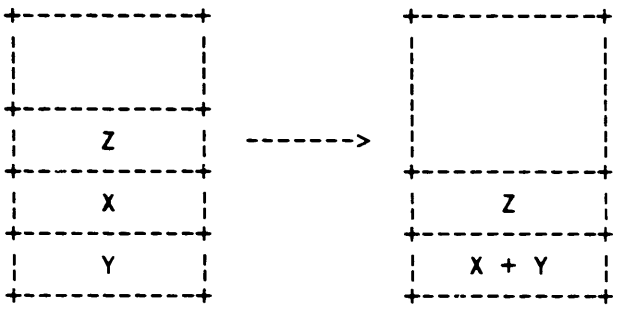
9.4.1 Program Excerpt

Here are the MICRO2 macro definitions for the machine model:

```
.toc  ""
 "Macro Definitions"

.toc  "   Stack Interactions"

Pop Stack              "MEMFNC/STKRD,ALU/A,CLKT1/YES"
Push Stack             "MEMFNC/STKWR,SELT1PC/T1"
Stack Plus T1          "MEMFNC/STKRD,ALU/ADD,CLKT1/YES"

.toc  "   Memory Interactions"

MA <-- PC              "SELT1PC/PC,MEMFNC/LOADMA"
T1 <-- Memory Data     "MEMFNC/READ,ALU/A,CLKT1/YES"
IR <-- Memory Data     "MEMFNC/READ,CLKIR/YES"

.toc  "   Arithmetic Actions"

T1 <-- PC + 1          "MEMFNC/READPC,ALU/A+1,CLKT1/YES"
PC <-- T1              "SELT1PC/T1,MEMFNC/WRITEPC"


.toc  "   Branch Tests"

 ALUCC?                "UBF/ALU"
Instruction Decode     "UBF/INST"


.toc  "   Miscellaneous Functions"

Set Condition Codes    "SETCC/SET"
```

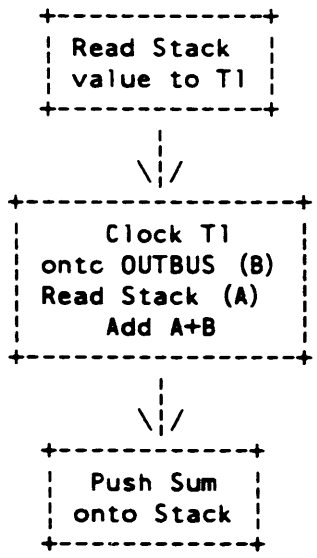
9.5 A SUBROUTINE

The subroutine given below adds the top two values on the stack and replaces these values with the value of their sum, as represented below:



9.5.1 Program Diagram

The program to accomplish this function can be diagrammed as follows:



9.5.2 Program Excerpt

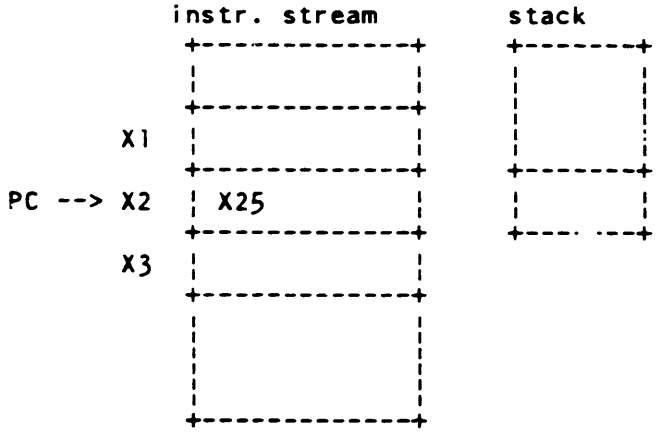
Here is the microprogram that accomplishes this function:

```
.toc ""
.toc "Functions"
 "    Add Top Two Values on Stack"

;
;     ADD (SP) to (SP-1)
;
;  This basic arithmetic instruction adds the top two values
;  on the stack and replaces the result onto the stack.  The
;  condition codes are set according to the result of the
   addition.


ADD01:
  Pop Stack,J/ADD02             ;Read stack value to T1.

ADD02:
  Stack plus T1,                ;Load T1 with (sp) + (sp-1) and
  Set condition codes,J/ADD03   ;save condition codes.

ADD03:
  Push stack,J/INSTFETCH        ;Store sum on stack
```

9.6 ANOTHER SUBROUTINE

This subroutine uses the sign of the value on the top of the stack to determine where to send control.

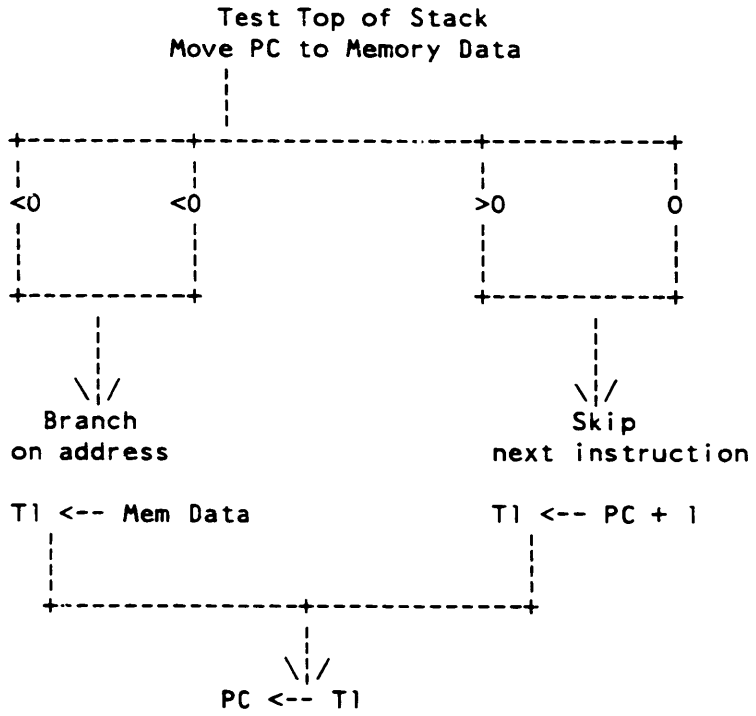
Suppose we have the following:



When the instruction X1 is being executed, the program counter points to the next instruction X2, which contains the address of instruction X25. If the sign of the top of the stack is negative, control is sent to instruction X25. If not, control continues with instruction X3.

9.6.1 Program Diagram

The program to accomplish this function can be diagrammed as follows:



9.6.2 Program Excerpt

```
"   Branch on Sign of Top of Stack"

;
;  TEST (SP) and Branch if Negative
;
;  This control instruction examines the top item on the stack
;  If the value is less than zero then the next word in the
;  instruction address stream contains the absolute address of
;  the next instruction to be executed.  Any other value
;  causes the absolute address in the instruction stream to be
;  skipped and execution continues normally.
;
;  On entry to this routine, the PC points to the absolute
```

```
;
;  On exit, the PC contains the address of the next instruction
;  to be executed.  The top item on the stack has been removed.

TSTBN01:
  Pop Stack,J/TSTBN02          ;Test SP

TSTBN02:
  MA <-- PC,                   ;Memory address of absolute
                               ;address.
  ALUCC?,J/TSTBN03             ;Branch on conditior     SP.

=00

;
;  Negative Number
;
TSTBN03:
  T1 <-- Memory Data,          ;T1 <-- Absolute address.
  J/TSTBN08
;
;  Largest Negative Number
;
mTSTBN04:
  T1 <-- Memory Data,          ;T1 <-- Absolute Address.
  J/TSTBN08

;
;  Greater than zero
;
TSTBN05:
  T1 <-- PC + 1,J/TSTBN08      ;Skip Absolute Address.

;
;  Zero
;
TSTBN06:
  T1 <-- PC + 1,J/TSTBN08      ;Skip Absolute Address.

TSTBN08:
  PC <-- T1,J/INSTFETCH        ;Load PC with next instruction
                               ;address.


INSTFETCH:
  J/INSTFETCH
```

CHAPTER 10

CONDITIONAL ASSEMBLY

The conditional assembly capability lets you suppress the assembly of parts of your program. You select the parts of your program to be suppressed by the appropriate setting of expression-names

Suppose, for example, you have a list processing package that consists of a common structure and a set of subroutines. Each subroutine is an entry point for the package. Sometimes you want to assemble this package with one set of entry points (or functions) and other times with another set. You can accomplish this by using the conditional assembly keywords.

10.1 THE CONDITIONAL ASSEMBLY KEYWORDS

Three keywords are provided for conditional assembly as follows:

```
.IF/expression-name
.IFNOT/expression-name
.ENDIF
```

These keywords divide your program into blocks. The .IF and .IFNOT keywords begin a block. They include an expression-name that is associated with either a true (1) or false (0) value. These keywords have the following meaning.

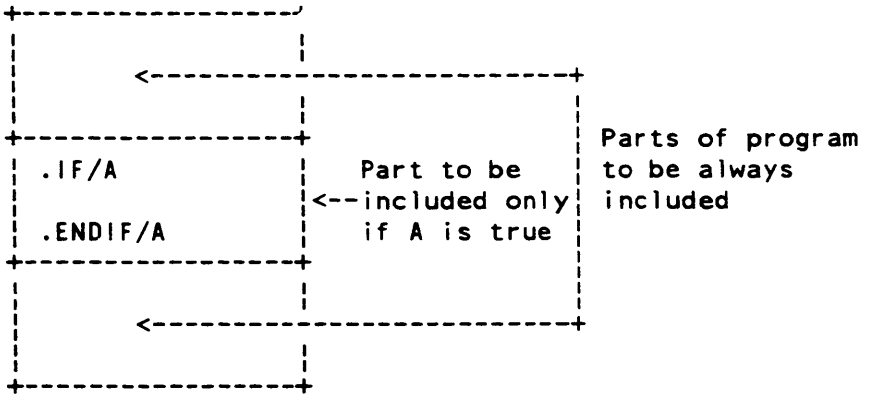
Keyword	Meaning
.IF/expression-name	If expression-name is associated with a true value (1), assemble the following block.
.IFNOT/expression-name	If expression-name is associated with a false value (0), assemble the following block.

In practice, a value is any value that is not 1. For example, if the expression-name has the value 2, MICRO2 considers it to represent a false value.

10.2 CONDITIONAL ASSEMBLY BLOCKS

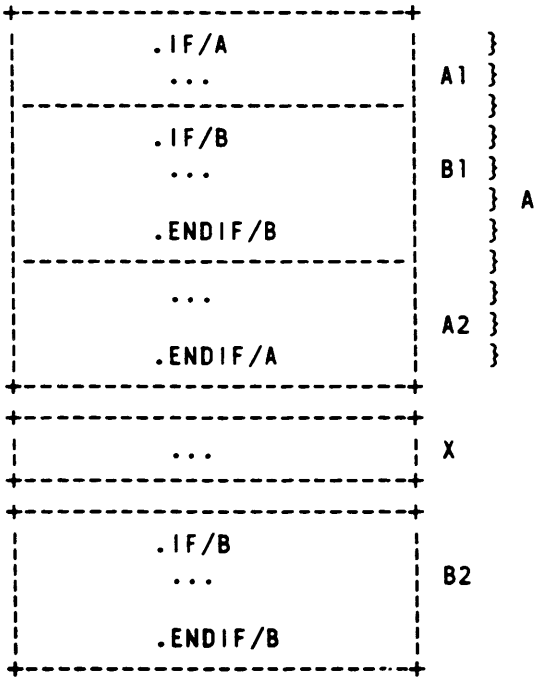
A conditional assembly block begins with either an .IF or .IFNOT and ends with either an .ENDIF for the same expression or another .IF or .IFNOT for the same expression.

For example, suppose you have a portion of your program to be included only under some special condition. You can enclose that portion of the program in a conditional assembly block by branching it with .IF and .ENDIF keywords as follows:



10.3 AN EXAMPLE

Suppose you have a sequence of conditional assembly directives, which divide your program into the following blocks:



The block A is included if the value of A is true (1). However, within block A, the block B1 is included only if the value of B is also true.

The portions of the program that are assembled for the different values of A and B are shown in the following table.

Case	A Value	B Value	Program Portions Assembled
1	T	T	A1,B1,A2,X,B2 (The entire program)
2	T	F	A1,A2,X
3	F	T	X,B2
4	F	F	X

Block X is always assembled because it is outside all conditional assembly blocks. In Case 3, block B1 is not assembled because it is nested within the conditional assembly block controlled by A.

10.4 ANOTHER EXAMPLE

Suppose you have a case in which you want to select some general code and then a set of one out of two pairs of options. You can use one expression-name to accomplish this binary decision, as follows:

```
+------------------------+
|                        |
|     general code       | X
|                        |
+------------------------+
|        .IF/A           | K1
|        option 1        |
+------------------------+
|        .IFNOT/A        | K2
|        option 2        |
+------------------------+
|        .IF/A           | K3
|        option 1        |
+------------------------+
|        .IFNOT/A        | K4
|        option 2        |
+------------------------+
|        .IF/A           | K5
|        option 1        |
+------------------------+
|        .IFNOT/A        | K6
|        option 2        |
+------------------------+
         .ENDIF/A
```

The portions of the program assembled are shown in the following table:

Case	A Value	Program Portions Assembled
1	T	X,K1,K3,K5
2	F	X,K2,K4,K6

10.5 SETTING AND CHANGING EXPRESSION-NAMES

You define and set an expression-name with the .SET keyword as follows:

```
.SET/expression-name=expression
```

For example, to set the expression-name A to a true value for conditional assembly purposes, you use the following .SET directive:

```
.SET/A=1
```

Once you have defined and set an expression-name, you must use the .CHANGE keyword to change its value. Thus to change A to false, after defining it as true above, you must write:

```
.CHANGE/A=0
```

CHAPTER 11

MICRO2 LISTING AND LIST CONTROLS

This chapter describes the input and output listings of MICRO2 and the listing controls you can use to modify the format and content of the output listings.

11.1 ASSEMBLER INPUT

The input to the assembler is a microprogram. The microprogram consists of a sequence of lines, written in MICRO2 source and conforming to the syntactic rules of that language. The input can be prepared using any available editor.

This section discusses an example of assembler input.

11.1.1 Preparing The Input

The first step in preparing the input is writing the microprogram. To write a microprogram, you must be familiar with the internal details of the processor on which the program will run.

You enter the microprogram source using any available editor. From the assembler's point of view, the microprogram consists of a sequence of lines beginning at the start of the first input file and continuing until the end of the last input file is encountered. Within those limits, the microprogram must have the expected structure.

The assembler detects and reports errors, as described in Appendix D. In response to these errors, you edit the input to obtain a valid microprogram. This process continues until either no errors are present or until you are convinced that the messages produced do not affect the validity of your microprogram.

11.1.2 Formatting The Microprogram

Using a standard formatting scheme increases the readability of the microprogram. A standard format for the microprograms has been developed at DIGITAL and is given here for your information.

11.1.2.1 The General Format - If a memory has any definitions, it begins with the identification keywords and continues with field definitions, followed by macro-definitions.

Then the action-part of a microprogram is given. It consists of a sequence of microinstructions.

11.1.2.2 Microinstruction Format -

The rules for formatting a microinstruction are summarized as follows:

1. Precede the microinstruction by any general comments.

2. If the microinstruction has an explicit address, give that address at the left-margin and do not include any other information on that line.

3. If the microinstruction has a label, give that label at the left margin and do not include any other information on that line.

4. Include as many instruction-parts, separated by commas, as will fit in the columns starting at the second tab (column 17) and continuing to column 38.

5. Place any line-specific comments at the fifth tab (column 41).

6. If the allocation mode is random conclude the instruction-part with a branch to the next-address.

7. Separate each microinstruction from the remainder of the microprogram by one or more blank lines.

11.2 THE OUTPUT LISTING

The output listing of a microprogram corresponds to the input listing, except that the assembler prints some additional information, namely:

- A table of contents, formed by listing each .TOC line and .PAGE line with its assigned line number at the beginning of the output listing.
- A line number at the beginning of each line.
- Page headings at the top of each page.
- Microword information, giving the memory, address, and bits for each microinstruction in the microprogram.
- Error messages, if any errors are detected.
- A cross reference listing
- A map
- An error summary and statistics report.

A brief description of each of the above items is given in the following sections.

11.2.1 The Table Of Contents

The table of contents is constructed by collecting the text on the .TOC and .PAGE lines to the beginning of the listing. Judicial placement of the .TOC and .PAGE lines within the listing results in a useful table of contents, by which you can quickly reference any logical section of the microprogram. As the size of a microprogram increases, the value of the table of contents increases.

.TOC lines and the construction of the table of contents are described in detail in Chapter 2. A good example of the use of .TOC lines to produce a comprehensive table of contents can be found in the sample program in Chapter 9.

11.2.2 Line Numbers

MICRO2 numbers each input line. The line number is a decimal number, which starts at 1 and continues, in increments of 1, through 99999.

Since blank lines and comments are assigned line numbers, it is not unusual for a small microprogram to occupy several thousand lines. However, the line limit of 99999 is seldom exceeded.

11.2.3 Page Headings

The assembler divides the output listing into pages. Each page contains a heading line, a sub-heading line, and 54 lines of the microprogram. The page heading gives the following items of information:

- o The output listing file specification
- o The name and version number of the MICRO2 assembler used in assembling the microprogram.
- o The time and date of the assembly.
- o The program title, as derived from the .TITLE line.
- o The page number.

The sub-heading contains the following:

- o The input file specification
- o The subtitle derived from the last .TOC or keyword line.

If a .TITLE line is not given in the microprogram, then the title part of the heading is left blank. Similarly, if no .TOC or lines are given, the subtitle part is left blank.

Page headings are illustrated in the sample output listing given in Appendix B.

11.2.4 The Microword information

The main-listing portion of the microprogram file is divided on the page into two fields. The left field contains the object part of the program, the octal or hexadecimal representation of the microinstruction. The right field contains the source part of the microprogram, as prepared by the user as input to MICRO2.

The width of the left field is determined by the width of the widest microword in the program.

You can give the listing control .NOBIN to suppress the object part of the microprogram. Listing controls are described in Section 11.3. In such a case, the main-listing reproduces the input listing, except for any error messages placed there by MICRO2.

The microword information contains the memory, address, and bits of the microword object, in the following format:

```
memory   address   digit ...
```

For example, consider a microword line from the output listing in Appendix B.

```
U 13, 1041,0014
```

This line indicates that the octal word '10410014' is allocated to location 13 in memory U.

11.2.5 Error Messages

If an error is detected in a microprogram line, then an error message is printed by the assembler preceding that line. Error messages are easy to find within the listing because, instead of a line number, error messages begin with the string '????', followed by the error message.

Appendix D lists all of the assembler error messages.

11.2.6 The Cross Reference Listing

The cross reference listing lists each name in the program and gives information about the places in which it appears.

The cross reference listing consists of up to three parts, one for each of the following classes:

- o field and value names
- o expression names
- o macro names

With the exception of field-names, the cross reference gives the name followed by an ordered list of line numbers. The line number on which the name is defined is followed by the '#' character. The other line numbers indicate the lines on which the name is referenced.

The cross-reference listing for a field-name contains only the line number of its definition and any lines on which a field was referenced with a numeric rather than symbolic value. The set of value-names following define all the references to the field.

As an example of a cross reference listing, consider the following excerpt from the cross reference given in Appendix B.

```
ALU             14#
   A             20# 167 196 207
   ...
```

This cross reference listing shows that the field name ALU is defined on line 14. The value name A is defined on line 20 and used on lines 167, 196, and 207.

11.2.7 The Map Listing

MICRO2 produces a map listing for each memory used in the microprogram. The map listing shows the addresses used by the microprogram and, for each address, the line number of the microword allocated to it.

A map listing line contains the memory-indicator, starting address, and a set of line indicators. A line indicator consists of the line number followed optionally by an "=" or ":" character. If the line number is followed by a "=", the location was allocated by a constraint. If the line number is followed by a ":", the location was allocated by an absolute address assignment. If neither of these characters is present, MICRO2 chose the location according to its allocation algorithm for unconstrained addresses.

The first line indicator identifies the line on which the starting address is allocated. The next line indicator identifies the line on which the starting address plus 1 is allocated, and so on.

Consider the following line from the map listing in Appendix B.

```
U 00 208= 215= 221= 144 147 150 154
```

This line shows that location 00 in memory U is allocated according to a constraint on line 208. Location 01 is allocated according to a constraint on line 215, and so on.

If an address within a map listing is not allocated, MICRO2 leaves the corresponding space blank. If a range of addresses is blank, then those lines that have no allocated addresses are indicated by a message, as follows:

```
lowbound-highbound Unused
```

11.2.8 The Summary

The summary contains a list of all the errors MICRO2 detected, a memory usage breakdown, and a count of the warning and total error messages.

For each error detected, MICRO2 gives the line number and the text of the message.

For each memory, MICRO2 gives the total number of microwords used and the highest address allocated.

The error message count indicates how many warnings were issued and how many error messages were issued. Finally, the warnings and error messages are broken down into the number related to lexical processing and the number related to allocation and symbol resolution.

11.3 THE ULD FILE

The ULD file is the object file produced by MICRO2. It consists of an optional header followed by a code section, followed by the set of field and address definitions used in the program.

11.3.1 The Header

The header, if present, contains information about the program radix and the direction in which the bits are numbered.

11.3.2 The Code Section

The code section contains the address and content of each microword to be loaded into the control store in the following format:

```
[address] memory-id = contents
```

For example, consider the following code-section line:

```
[11]U = 16070012
```

This line indicates that word 11 of memory has the contents 16070012.

If the memory-indicator is not given, it is assumed to be U.

11.3.3 Field and Address Definitions

In this section of the ULD file, all the field and field-value names defined in the U memory are given.

A special control .ALLMEMFIELDS is provided for those who want to see the field and address names from other memories as well as the U memory.

For example, consider the following portion of a ULD file produced under the .ALLMEMFIELDS control:

```
FIELD D_FLD/=<0:2>                ;MEMORY/D
   DVAL1=1
   DVAL2=2
   DVAL3=3
FIELD UCODE_DATA_FLD/=<0:5>       ;MEMORY/U
   UVAL1=1
   UVAL2=2
   UVAL3=3

ADDRESS U_JMP/=<6:12>             ;MEMORY/U
   UADDR1=122
   UADDR2=222
ADDRESS VING/=<3:19>              ;MEMORY/D
   D_ADDR=11
   D_ADDR=12
   D_ADDR=13
```

The above portion contains field-names from both the U and D memory. If, however, you do not give the .ALLMEMFIELDS control, then only the field and address names from the U memory are given, as follows:

```
FIELD UCODE_DATA_FLD/=<0:5>
   UVAL1=1
   UVAL2=2
   UVAL3=3
ADDRESS U_JMP/=<6:12>
   UADDR1=122
   UADDR2=222
```

11.4 LIST CONTROLS

The list controls let you specify which portions of the output listing you want to see reflected in the output file.

For example, suppose you have a long set of definitions. These definitions must be present for the successful assembly of our microprogram. But after the first few assemblies, you don't change these definitions and, therefore, want to suppress their listing. You can accomplish this listing suppression by inserting a .NOLIST before the definitions and a .LIST after the definitions.

MICRO2 determines whether or not to make a contribution to a file by looking at a counter. If the counter contains a positive number, MICRO2 contributes to the associated file. If the counter is a negative number, MICRO2 does not contribute.

The list controls are as follows:

Keyword	Meaning
.LIST	Increment the listing counter
.NOLIST	Decrement the listing counter
.CREF	Increment the cross reference counter
.NOCREF	Decrement the cross reference counter
.BIN	Increment the object counter
.NOBIN	Decrement the object counter
.EXPAND	List the field/value pairs produced by expanding macros after the last line of the instruction.
.NOEXPAND	Do not expand macros in the listing.

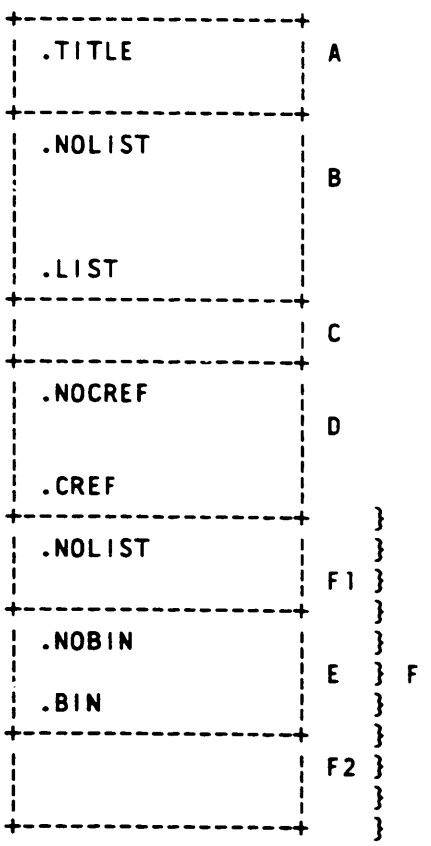
At the beginning of an assembly, each counter has the value 0.

11.4.1 The List Control Counters

If a list control counter is positive, then MICRO2 creates the specified part of listing. If a list control is negative, MICRO2 suppresses the specified part of the listing.

The counter associated with the .LIST and .NOLIST control determines whether or not an output listing is produced. The counter associated with the .BIN and .NOBIN controls determines whether or not the object part (left field) of the listing is produced. The counter associated with the .CREF and .NOCREF controls whether or not names will be added to the cross reference map.

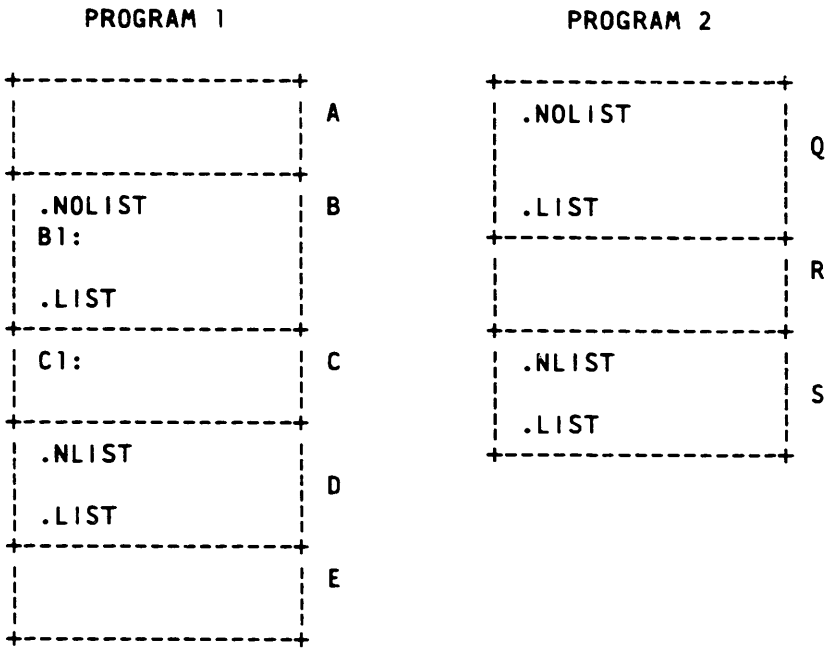
Suppose we add some list controls to a microprogram as follows:



These list controls partition the program for the purpose of output file creation. The listing file contains segments A, C, and D. The object file contains segments A, B, C, D, F1, and F2. The cross reference file contains segments A, B, C, and F.

The fact that the list controls operate with a counter allows you to combine programs and still retain the same list control structure.

Consider the following example:



If we assemble these programs separately, we get listing files as follows:

Program	Listing File Segments
1	A, C, and E
2	R

We can insert program 2 in program 1 at any place and get the correct logical result. If we insert program 2 at B1 (that is, within a nonlisting segment), the listing of program 2 is suppressed. If we insert program 2 at C1 (that is, within a listing segment), then the listing controls within program 2 are interpreted just as if they were when program 2 was a separate segment.

CHAPTER 12

USING MICRO2

To use MICRO2, you invoke it at command level and then give a MICRO2 command line that specifies the output and input files. This chapter describes the user interface in the VAX, DEC 10, and DEC 20 environments.

12.1 VAX/VMS INTERFACE

In the VAX/VMS environment, you call MICRO2 at command level as shown below:

Format

```
+-------------------------------------------------+
|                                                 |
|     MICRO2 input-file-spec                      |
|                                                 |
|     File Qualifiers                             |
|     ---------------                             |
|          /LIST[=file-spec]                      |
|          /NOLIST                                |
|          /ULD[=file-spec]                       |
|          /NOULD                                 |
+-------------------------------------------------+
```

Prompts

```
_File:  input-file-spec
```

File-Parameters

Input-file-spec
: Specifies the names of one or more files to be assembled. If you specify more than one input file, you can use the character '+' to separate file-specs.

Description

MICRO2 assembles the programs contained in the input-file-spec and produces a listing file and an object file.

File Qualifiers

/LIST [=file-spec]

Specifies that you want a listing file. If you include a file-spec, MICRO2 produces the listing in that file. If you do not include a file-spec, MICRO2 uses the name of the input-file, or the name of the first input file in the case of multiple input files, with the default extension .MCR for the listing file. The listing file is described in Section 11.2.

/NOLIST

Specifies that you do not want a listing file.

/ULD [=file-spec]

Specifies that you want an object-file. If you include a file-spec, MICRO2 produces the object file in that file. If you do not include a file-spec, MICRO2 uses the name of the input, or the name of the first input file in the multiple input file cape, with the default extension .ULD for the object file. The object file is described in Appendix C.

/NOULD

Specifies that you do not want an object-file.

Examples

1. MICRO2 ALPHA

   MICRO2 assembles the program in the file ALPHA.MIC and produces a listing file ALPHA.MCR and the object file ALPHA.ULD.

2. MICRO2/LIST=BETA ALPHA

   MICRO2 assembles the program in the file ALPHA.MIC and produces the listing file BETA.MCR and the object file ALPHA.ULD.

3. MICRO2/NOULD ALPHA+GAMMA

   MICRO2 assembles the program formed by the concatenation of ALPHA.MIC and GAMMA.MIC and produces the listing file ALPHA.MCR.

4. MICRO2/LIST=BETA/NOULD ALPHA

   MICRO2 assembles the program in the file ALPHA.MIC and produces the listing file BETA.MCR. MICRO2 does not produce an object file because the qualifier /NOULD is given.

File-Specifications

MICRO2 accepts and processes any legal VAX filename. For purposes of error reporting, MICRO2 abbreviates long filenames by truncating the name to the first six characters and the extension to the first three characters.

12.2 DEC 10 COMMAND LINE INTERFACE

To invoke MICRO2 at command level on the DEC 10, you type one of the following :

```
R MICRO2 command-information

R MICRO2
```

If the command information is not given in the invocation, MICRO2 prompts for the command information with an asterisk (*) character.

The command information consists of the listing-file followed by a comma followed by the file-specification for the object file followed by an "=" character followed by the input file-specifications. That is, the form of the command information is:

```
listing-file, object-file = {input-file},...
```

If you give more than one input file, the assembler reads the microprogram from the specified input files in the order given in the assembly-command-line. The first input file is read, then that file is closed and the second file is opened. Processing continues with the second input file, again until an end-of-file, indicating the end of input on that file, is ready. Processing continues in this way, moving from file to file, until all files are processed. The input files must not contain any line numbers. MICRO2 rejects such files.

The assembler produces the output listing on the listing-file. If a listing-file is not specified, then MICRO2 uses the name of the first input file and the extension .MCR for the name of the listing file. The output listing is described in Section 11.2.

The assembler produces the load module on the object-file and that file can be subsequently given as an input file for the microprogram loader. If an object-file is not specified, no load module is produced. The format of the load module is described in Appendix C.

You get an object file, if you specify an object file name or if you include the "," character preceding it. consider the following MICRO2 calls:

Call	Result
R MICRO2 A,B=C	MICRO2 assembles the program in the file C.MIC and produces the listing file A.MCR and the object file B.ULD
R MICRO2 A,=C	MICRO2 assembles C.MIC and produces the listing file A.MCR and the object file C.ULD
R MICRO2 ,B=C	MICRO2 assembles C.MIC and produces the object file B.ULD

12.2.1 File Specifications

A file specification has the following form:

```
diskname:filename.ext[PPN]
```

Diskname, ext, and PPN are all optional. If diskname or PPN are omitted, the user's default disk or PPN is assumed.

12.3 DEC 20 INTERFACE

You invoke MICRO2 on the DEC-20 system just as on the 10 system, the only exception is that instead of typing R followed by MICRO2 at command level, you simply type MICRO2, as follows:

```
MICRO2
```

APPENDIX A

MICRO2 LANGUAGE SYNTACTIC SUMMARY

This appendix provides a quick reference to the MICRO2 language. First, the syntax of the language is given. Then a summary is given of the elements used to build a MICRO2 source program.

A.1 MICRO2 SYNTACTIC SUMMARY

The MICRO2 syntactic summary gives the syntax, restrictions, and defaults for the MICRO2 language. Following this information, examples of the syntax are given.

The syntax notation used to express the MICRO2 language is the same as the syntax notation used in the BLISS Language Guide (AA-H275A-RK). A detailed description of the notation is contained in Chapter 3 of that manual.

Briefly, a syntactic rule is given in a box. The left part of the box contains the syntactic name being defined. The right side of the box contains the definition.

Concatenation is expressed in the definition by writing the terms one after another. An example of a concatenation is:

```
field-contents-indicator field-name /
```

The above rule defines a field-contents-indicator to be a field-name followed by the character '/'.

Disjunction is expressed by enclosing the possible terms in braces and distinguishing the possibilities either by separating them by the character '!' or by giving them on separate lines. An example o a disjunction is:

```
octal-digit          {0¦1¦2¦3¦4¦5¦6¦7}
```

The above rule defines an octal digit to be a '0' or a '1' or a '2' and so on.

Another example of a disjunctin is:

```
                     { octal-number       }
number               { decimal-number     }
                     { hexadecimal-number }
```

Omission is indicated by including the term 'nothing' as a possibility in a disjunction. An example of omission is:

```
                                { : right-bit }
field-spec           < left-bit { nothing     } >
```

The above rule defines a field-spec to be either a '<' followed by left-bit followed by ':' followed by right-bit followed by '>' or a '<' followed by left-bit followed by '>'.

Finally, replication is indicated by the character sequence '...'. If the replication contains a separator, the separator appears as the first character. An example of replication is:

```
arg-part             [ argument ,... ]
```

The above rule defines 'arg-part' to be a '[' followed by one or more arguments separated by ',' followed by ']'. That is, the following are all valid definitions of 'arg-part':

```
[ argument ]
[ argument, argument ]
[ argument, argument, argument ]
  ...
```

A.1.1 The Program

program	{ .LTOR ¦ .RTOL ¦ nothing }
	{ .OCTAL ¦ .HEXADECIMAL ¦ nothing }
	memory ...
memory	identification-part
	definition-part
	microinstruction-part
identification-part	{ memory-indicator ¦ nothing }
	{ .WIDTH/word-width ¦ nothing }
	{ .TITLE/" " ¦ nothing }
	{ .VERSION/" " ¦ nothing }
	{ .REGION {/low-address, high-address} ... }
	{ ¦ nothing }
	{ .RANDOM ¦ .SEQUENTIAL ¦ nothing }
memory-indicator	{ .UCODE ¦ .DCODE ¦ .ECODE ¦ .ICODE }
	{ .OCODE ¦ .CCODE ¦ .MCODE }
word-width	decimal-integer
low-address high-address	expression-subject-to-program-radix

A.1.1.1 Defaults -

memory-indicator	.UCODE
program-radix	.OCTAL
bit-order	.LTOR
allocation mode	.RANDOM
word-width	highest bit specified in a field-definition plus 1

A.1.1.2 Restrictions -

1. 0 < word-width <= 128
2. low-address < high-address

A.1.1.3 Examples -

```
.RTOL
.OCTAL
.DCODE
.WIDTH/64
.REGION/0100,0177
.RANDOM
```

A.1.2 The Definition Part

```
definition-part       { field-definition }
                      { macro-definition } ...

                                          { , qualifier... }
field-definition      field-name/=field-spec { nothing       }

                        { value-name = value-spec }...

                                 { : right-bit }
field-spec            < left-bit { nothing     }  >

                      { .DEFAULT = expression  }
                      { .ADDRESS               }
qualifier             { .NEXTADDRESS           }
                      { .VALIDITY = expression }
                      { .FLOATEPARITY          }
                      { .FLOATOPARITY          }

                           {   {t1-value} {,t2-value}}
                           { , {nothing } {nothing  }} {.VALIDITY}
value-spec            value { nothing                } {nothing  }

field-name }
value-name }          name

name                  name-char { name-char { space   } ... }
                                {           { nothing }     }

name-char             { letter ¦ number ¦ special-char }
```

letter	{ A ¦ a ¦ B ¦ b ¦ ... ¦ Z ¦ z }
number	{ 0 ¦ 1 ¦ 2 ¦ ... ¦ 9 }
special-character	{ ! ¦ # ¦ & ¦ (¦) ¦ * ¦ + ¦ - ¦ . ¦ ? ¦ _ } { < ¦ > }
space	{ space-character ... }
space-character	{ blank ¦ tab }
left-bit } right-bit }	decimal-integer
value	integer-subject-to-program-radix

A.1.2.1 Restrictions -

1. If .LTOR specified, then in a field-spec left-bit >= right-bit

   If .RTOL specified, then left bit <= right-bit

2. Either .NEXTADDRESS or .ADDRESS must be specified once and only once per memory.

A.1.2.2 Examples -

field-definition

```
CLKT1/= <17>,.default = <CLKT1/NO>
  NO = 0       ;Do not clock T1
  YES = 1      ;Load T1 with ALU output
```

A.1.3 Expressions

expression	< { number } { expression-name } { field-conten s-indicator } { field-value-identifier } { function-call } >
number	{ octal-number } { decimal-number } { hexadecimal-number }
octal-number	octal-digit ...
decimal-number	decimal-digit ... decimal-point
hexadecimal-number	hex-digit ...
octal-digit	{ 0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 }