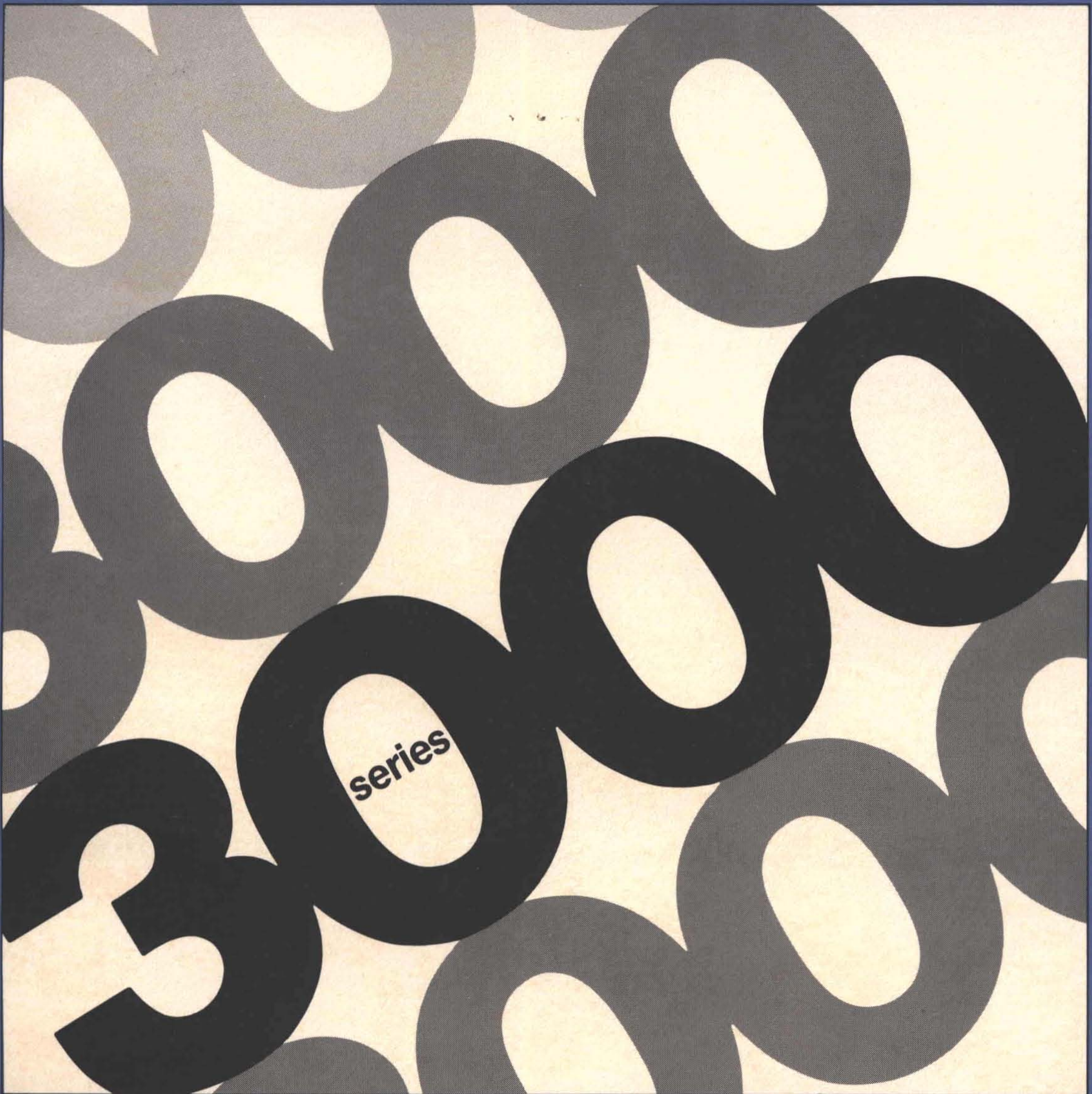
$5.00

intel®

Series 3000 Microprogramming Manual



98-210A

intel®

Series 3000 Family Of Computing Elements — The Total System Solution.

Since its introduction in September, 1974, the Series 3000 family of computing elements has found acceptance in a wide range of high performance applications from disk controllers to airborne CPU's.

The Series 3000 family represents more than a simple collection of bipolar components, it is a complete family of computing elements and hardware/software support that greatly simplifies the task of transforming a design from concept to production.

The Series 3000 Component Family

A complete set of computing elements that are designed as a system requiring a minimum amount of ancillary circuitry.

3001	Microprogram Control Unit.
3002	Central Processing Element.
3003	Look-Ahead Carry Generator.
3212	Multi-Mode Latch Buffer.
3214	Interrupt Control Unit.
3216/26	Parallel Bi-directional Bus Driver.
ROMs/PROMs	A complete set of bipolar ROMs and PROMs.
RAMs	A Complete family of MOS and bipolar RAMs.

The Series 3000 Support

A comprehensive support system that assists the designer in writing microprograms, debugging hardware and microcode, and programming prototype and production PROMs.

CROMIS	Cross microprogram assembler.
MDS-800	Microcomputer development system with TTY/CRT, line printer, diskette, PROM programmer and high speed paper tape reader facilities.
ICE-30	In-circuit emulation for the 3001 MCU.
ROM-SIM	ROM simulation for all of Intel's Bipolar ROMs and PROMs.
Application Notes	Central processor and disk controller designs and system timing considerations.
Customer Course	Comprehensive 3 day course covering the component family, CPU and controller designs, microprogramming and the MDS-800, ICE-30 and ROM-SIM operation.

The Series 3000 family is designed to provide a Total System Solution: high performance, minimum package count and total commitment to support.

MICROPROGRAMMING THE SERIES 3000

REVISION A

Intel® Corporation
3065 Bowers Avenue
Santa Clara, Ca 95051

TABLE OF CONTENTS

TABLE OF CONTENTS
(continued)

SECTION I
INTRODUCTION

For some time, microprogramming has been recognized as a powerful technique for the design of a complex processing system such as the central processing unit of a general purpose computer. Recent advances in high-speed LSI circuit technology have made it possible and economically practical to apply a microprogramming solution to a wide range of design problems.

A microprogramming approach to a design problem requires two distinct steps to achieve the design objective:

1. Design a microprogrammable processor capable of meeting the design objective.
2. Design a microprogram that will direct the activity of the processor to satisfy the design objective.

The first of these steps is a logic design problem; the second step is a programming problem. The Intel® Series 3000 Bipolar Microcomputer Set supports the first aspect. CROMIS, the Cross Microprogramming System, supports the second.

1.1 SERIES 3000 BIPOLAR MICROCOMPUTER SET

The Intel® Series 3000 Bipolar Microcomputer Set is a complete, compatible family of high performance LSI circuit components that serve as the basic building blocks for custom microprogrammed processors and controllers. The set's two major components, the 3001 Microprogram Control Unit (MCU) and 3002 Central Processing Element (CPE), establish the foundation of a flexible microprogrammable architecture. The user can build on this foundation to meet the requirements of a wide variety of special applications.

TYPICAL CONFIGURATION

Although a user's configuration is designed to meet the requirements of his particular application, it is worthwhile to consider how the Series 3000 elements might be applied in a typical microprogrammable processor-controller implementation. The configuration illustrated in Figure 1-1 may differ in some ways from the one a user is microprogramming. However, a discussion of Figure 1-1 identifies the roles played by the Microprogram Memory, the Microprogram Control Unit (MCU) and the Central Processing Element (CPE), which are fundamental to the architecture of any Series 3000 configuration.

MICROPROGRAM MEMORY

MICROINSTRUCTION WORD

The microprogram memory may be viewed (see Figure 1-2) as an array of locations, each providing storage for one unit of control information: a microinstruction word. Each location is identified by a unique address. An address applied to the address inputs of the microprogram memory selects a location. The bit pattern stored in the selected location (i.e., the microinstruction word) appears in the form of a pattern of binary signal levels on the control lines connected to the data outputs of the microprogram memory. Thus, each bit in the selected microinstruction word determines the state of one control line.

FUNCTION BUSSES

FIELDS

A group of related control lines forms a function bus. Each function bus controls the operation of, or supplies data to a functional unit in the system. The grouping of the control lines into function busses imposes natural functional divisions on the microinstruction word. These divisions are called fields. Each field of the microinstruction word drives a single function bus and thereby governs the behavior of one functional unit. Therefore, each field of a microinstruction word can be viewed as providing an instruction to be executed by one functional unit in the system. In other words, the microinstruction word consists of a fixed group of instructions that are executed in parallel by corresponding functional units when the microinstruction word is selected from the microprogram memory.

MICROINSTRUCTION CYCLE

During a basic operating cycle, called a microinstruction cycle, an address is applied to the microprogram memory, selecting one microinstruction word for execution. The fields of the selected microinstruction word dictate the functions to be performed or initiated by the functional units in the configuration during that cycle.

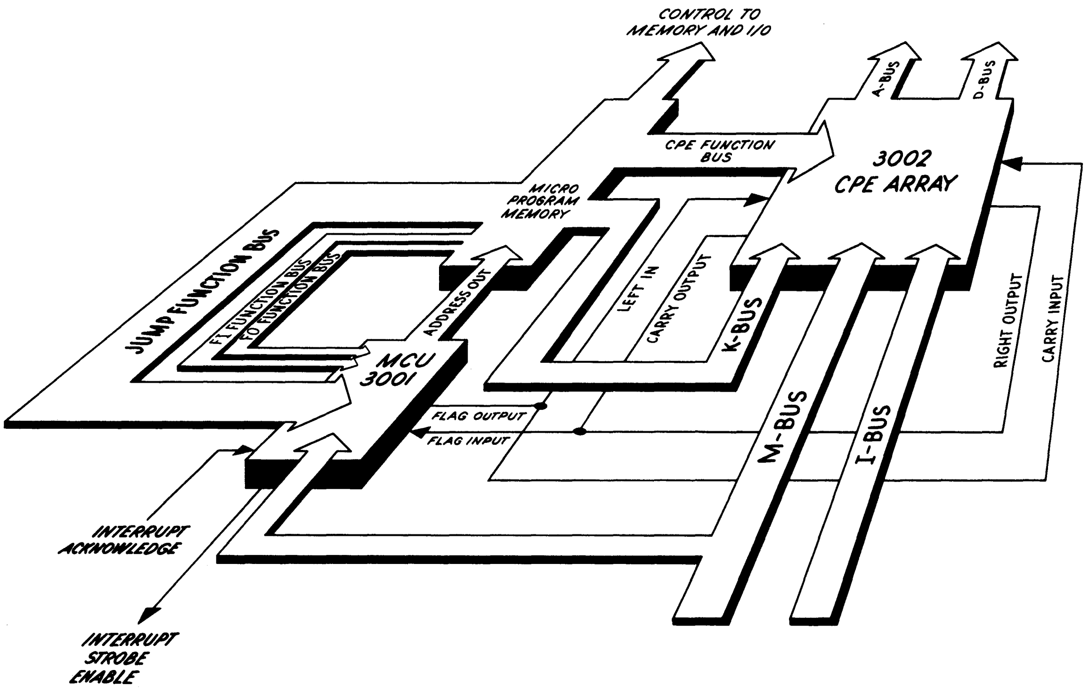


Fig. 1-1. Typical Series 3000 Configuration

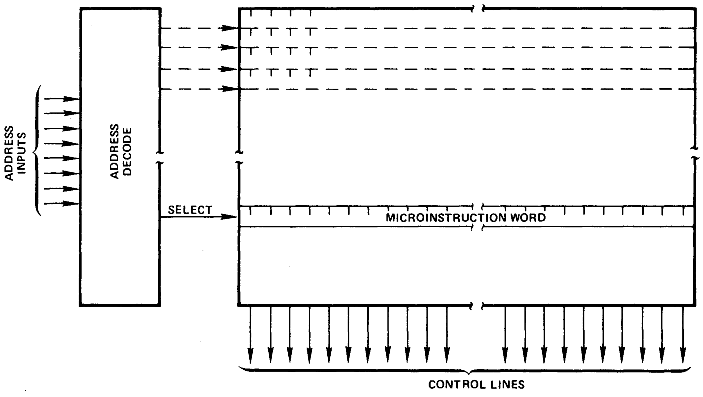


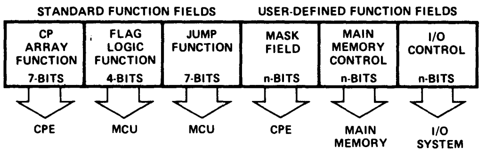
Fig. 1-2. Microprogram Memory

ROLE OF THE MICROINSTRUCTION WORD FIELDS

One of the microinstruction word fields controls the operation that determines which microinstruction word will be executed during the next microinstruction cycle. Another field specifies the operation to be performed by the data processing section (the CPE array). Still other fields may control an external main memory and I/O devices.

MICROINSTRUCTION WORD FORMAT

The design flexibility of the Series 3000 computing elements makes it impossible to describe every format exactly. However, the microinstruction word format below reflects the general requirements of the typical configuration illustrated in Figure 1-1:



3001 CENTRAL PROCESSING ELEMENT (CPE)

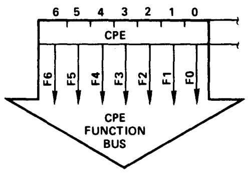
CPE ARRAY

The CPE is a complete 2-bit data processing module. It includes a number of general and special purpose registers, arithmetic and logic circuits, and several busses for data input and data output, as illustrated in Figure 1-3. Virtually any number of CPE's may be connected, as shown in Figure 1-4, to implement a data processing section of any desired word width. For example, 8 CPE's may be arrayed to produce a 16-bit processing section.

As CPE's are wired together, all registers, arithmetic and logic circuits, and data paths expand accordingly. However, the seven function inputs (F0-F6) to each CPE are connected in parallel to form a single 7-bit CPE Function Bus for the entire array. This arrangement allows the microprogrammer to view the CPE array as a single functional unit capable of executing a variety of data processing functions on N-bit operands.

CPE FIELD

In a typical configuration, a 7-bit field, called the CPE field, in the microinstruction word drives the CPE Function Bus:



This field, then, determines the function to be executed by the CPE array during a microinstruction cycle.

CPE FUNCTION BUS

The 7-bit CPE Function Bus controls the internal operation of the CPE array by selecting the operands and operation to be performed during a machine cycle. The Arithmetic and Logic Unit (ALU), under the control of the function bus decoder, performs over 40 Boolean and arithmetic functions including 2's complement arithmetic and logical AND, OR, NOT, and exclusive-NOR.

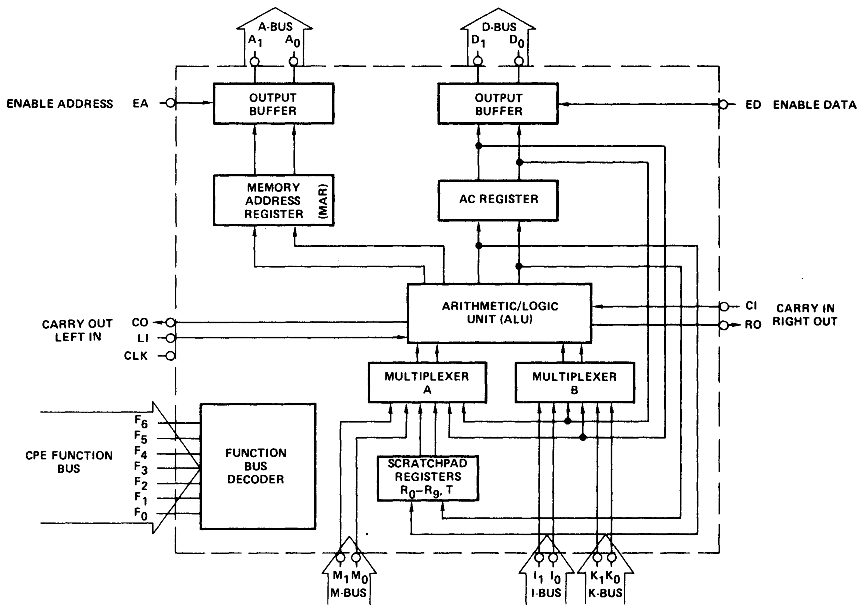


Fig. 1-3. 3002 CPE Organization

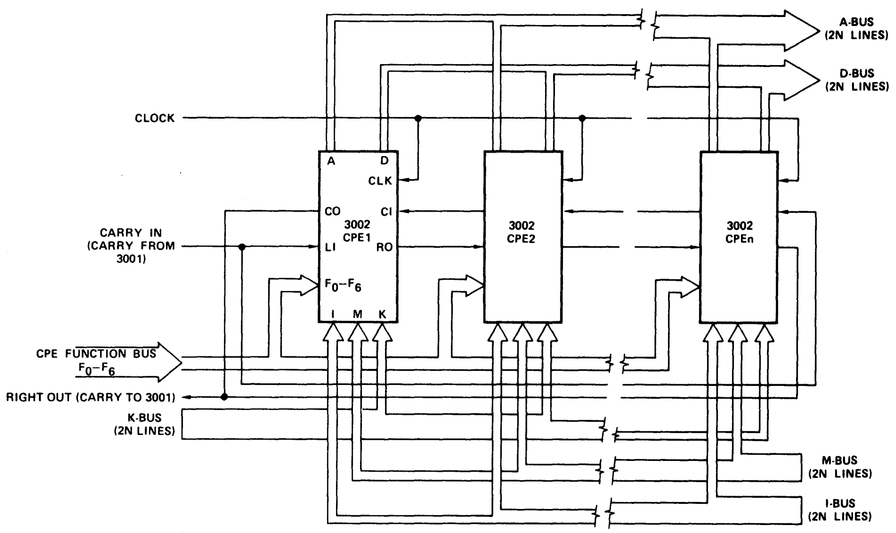


Fig. 1-4. CPE Array Configuration

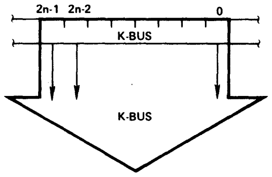
ARRAY FUNCTIONS

Three busses, the M-bus, the I-bus and the K-bus, provide paths by which data enters the CPE array. Within the array, eleven scratch pad registers, an accumulator (AC) register and a Memory Address Register (MAR) provide data storage. Two internal multiplexers select operands for the arithmetic and logic unit (ALU) from data in the registers and on the input busses. Results of ALU operation can be stored in the scratch pad registers or the accumulator. The ALU provides a special data transfer path to the MAR. Data in the MAR and in the accumulator can be gated onto the output busses, the A-bus and the D-bus.

K-BUS

The information on the K-bus is used in every CPE function and has a modifying effect on the result of every function. Consequently, it is typical to put a field in the microinstruction word to provide either direct or encoded drive for the K-bus inputs to the CPE array.

K-BUS FIELD



This drawing illustrates a one-to-one correspondence between the 2n bit positions of the K-bus field in the microinstruction word and the 2n K-bus input lines to the CPE array. For most applications, it is possible to reduce the size of the K-bus field by strapping groups of K-bus lines together. In fact, all the K-bus lines may be strapped together in a single group and driven by a one-bit K-bus field; in this case, only an all-zero or all-one k-bus input is possible, which is sufficient for some applications.

AFFECT OF K-BUS ON CPE ARRAY OPERATIONS

The K-bus participates in every CPE operation. The K-bus inputs are always ANDed with the B-multiplexer outputs into the ALU. Consequently, bit masking can be performed with the mask supplied to the K-bus directly from the microinstruction. Placing the K-bus in the all one or all zero state will, in most cases, select or deselect the accumulator in the operation. This toggling effect on the accumulator nearly doubles the CPE's repetoire of functions. For instance, with the K-bus in the all-zero state the data on the M-bus may be complemented and loaded into the CPE's accumulator. The same function selected with the K-bus in the all-one state will exclusive-NOR the data on the M-bus with the contents of the accumulator.

CARRY AND SHIFT INPUT AND OUTPUT

ZERO DETECTION

The CPE provides independent carry input, carry output, shift input and shift output. The shift input and output are functional only during a shift right operation. The carry input and output participate in arithmetic operations. In non-arithmetic operations the carry output of the CPE array is the logical OR of all the bits of the result of operation; in other words, the carry output serves as a "not zero" status indication for non-arithmetic operations. Zero detection in conjunction with the masking function of the K-bus input provides a complete bit testing facility.

CPE FUNCTIONS

Table 1-1 summarizes some of the important CPE functions. Note that these functions are described for all zero or all one K-bus inputs only. A complete description of all CPE functions appears in Chapter 4.

3001 MICROPROGRAM CONTROL UNIT (MCU)

PROVIDES THREE INDEPENDENT FUNCTIONS

The 3001 Microprogram Control Unit (MCU) provides the microprogrammer with three independently controllable functional facilities:

Jump Function — A mechanism for controlling the sequence in which the microinstructions are accessed from the microprogram memory:

Flag Input Function — A mechanism for saving the state of the Flag Input (which is usually connected to the Carry/Shift Output of the CPE array);

Table 1-1. ALL-ZERO AND ALL-ONE K-BUS CPE FUNCTIONS

MNEMONIC	K-BUS = 00 MICRO-FUNCTION	MNEMONIC	K-BUS = 11 MICRO-FUNCTION
ILR	$R_n + CI \rightarrow R_n, AC$	ALR	$AC + R_n + CI \rightarrow R_n, AC$
ACM	$M + CI \rightarrow AT$	AMA	$M + AC + CI \rightarrow AT$
SRA	$AT_L \rightarrow RO \quad AT_H \rightarrow AT_L \quad LI \rightarrow AT_H$	—	(See Appendix B)
LMI	$R_n \rightarrow MAR \quad R_n + CI \rightarrow R_n$	DSM	$11 \rightarrow MAR \quad R_n - 1 + CI \rightarrow R_n$
LMM	$M \rightarrow MAR \quad M + CI \rightarrow AT$	LDM	$11 \rightarrow MAR \quad M - 1 + CI \rightarrow AT$
CIA	$AT + CI \rightarrow AT$	DCA	$AT - 1 + CI \rightarrow AT$
CSR	$CI - 1 \rightarrow R_n$ See Note 1	SDR	$AC - 1 + CI \rightarrow R_n$ See Note 1
CSA	$CI - 1 \rightarrow AT$	SDA	$AC - 1 + CI \rightarrow AT$
—	(See CSA above)	LDI	$I - 1 + CI \rightarrow AT$
INR	$R_n + CI \rightarrow R_n$	ADR	$AC + R_n + CI \rightarrow R_n$
	(See ACM above)	—	(See AMA above)
INA	$AT + CI \rightarrow AT$	AIA	$I + AT + CI \rightarrow AT$
CLR	$CI \rightarrow CO \quad 0 \rightarrow R_n$	ANR	$CI \vee (R_n \wedge AC) \rightarrow CO \quad R_n \wedge AC \rightarrow Rn$
CLA	$CI \rightarrow CO \quad 0 \rightarrow AT$	ANM	$CI \vee (M \wedge AC) \rightarrow CO \quad M \wedge AC \rightarrow AT$
—	(See CLA above)	ANI	$CI \vee (AT \wedge I) \rightarrow CO \quad AT \wedge I \rightarrow AT$
—	(See CLR above)	TZR	$CI \vee R_n \rightarrow CO \quad R_n \rightarrow R_n$
—	(See CLA above)	LTM	$CI \vee M \rightarrow CO \quad M \rightarrow AT$
—	(See CLA above)	TZA	$CI \vee AT \rightarrow CO \quad AT \rightarrow AT$
NOP	$CI \rightarrow CO \quad R_n \rightarrow R_n$	ORR	$CI \vee AC \rightarrow CO \quad R_n \vee AC \rightarrow R_n$
LMF	$CI \rightarrow CO \quad M \rightarrow AT$	ORM	$CI \vee AC \rightarrow CO \quad M \vee AC \rightarrow AT$
—	(See NOP above)	ORI	$CI \vee I \rightarrow CO \quad I \vee AT \rightarrow AT$
CMR	$CI \rightarrow CO \quad \overline{R_n} \rightarrow R_n$	XNR	$CI \vee (R_n \wedge AC) \rightarrow CO \quad R_n \overline{\oplus} AC \rightarrow R_n$
LCM	$CI \rightarrow CO \quad \overline{M} \rightarrow AT$	XNM	$CI \vee (M \wedge AC) \rightarrow CO \quad M \overline{\oplus} AC \rightarrow AT$
CMA	$CI \rightarrow CO \quad \overline{AT} \rightarrow AT$	XNI	$CI \vee (AT \wedge I) \rightarrow CO \quad I \overline{\oplus} AT \rightarrow AT$

NOTES:

1. 2's complement arithmetic adds 111 . . . 11 to perform subtraction of 000 . . . 01.
2. R_n includes T and AC as source and destination registers in R-group 1 micro-functions.
3. Standard arithmetic carry output values are generated in F-group 0, 1, 2 and 3 instructions.

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

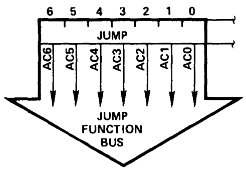
Flag Output Function —A mechanism for controlling the state of the Flag Output (which is typically connected to the Carry/Shift Input to the CPE array).

The MCU has separate function bus inputs for controlling each of these facilities, as illustrated in Figure 1-5.

JUMP FUNCTIONS

The MCU defines a comprehensive and powerful microprogram memory addressing scheme that incorporates a repetoire of eleven conditional and unconditional Jump functions as well as a microprogram interrupt capability. In a typical configuration, a 7-bit field in the microinstruction word, called the JUMP field, drives the Jump Function Bus to the MCU.

JUMP FIELD



Each of the MCU's eleven jump functions is represented by a unique coding of the JUMP field. From two to five bits of the field select the JUMP function while the remaining bits supply part of the destination address. During each microinstruction cycle the MCU executes the Jump function specified by the microinstruction currently being accessed from the microprogram memory. In executing this Jump function, the MCU formulates the address that will be used to access the microprogram memory for the next microinstruction in the microprogram sequence. In this way, the microprogram controls its own sequencing. The conditional Jump functions provide a test facility by allowing selected information maintained by the MCU (e.g., in the C and Z flags, PR-latch, etc.) to influence program sequencing.

CONCEPTUALIZING MICROPROGRAM MEMORY

ROWS AND COLUMNS

It is simpler to understand the MCU Jump functions if the Microprogram Memory is conceptualized as a two-dimensional matrix consisting of 32 rows and 16 columns, providing a total of 512 microinstruction locations. Refer to the illustration in Figure 1-6. The location of microinstruction is identified by its row address and its column address in the matrix. The 9-bit microprogram memory address, which is provided by the MCU, specifies the row address in the high order five bits and the column address in the low order four bits. (It is possible to implement Microprogram Memories larger than 512 locations, as discussed later.) For example, from a particular row and column location, it is possible to jump unconditionally to any other location in that row, using one Jump function, or any other location in that column, using another Jump function.

JUMP SET

For a given location in the matrix and MCU Jump function, there is a fixed subset of microprogram addresses that may be selected as the next address. These addresses are referred to as the jump set for the Jump function; each Jump function has a jump set associated with it.

Table 4-1 provides a brief summary of the MCU's Jump functions. A more detailed description is presented in Section 4.1.

NEXT ADDRESS LOGIC

Under normal operation, the MCU's Microprogram Address Register holds the address of the microinstruction word currently being accessed from the microprogram memory. During a microinstruction cycle, the Next Address Logic, under primary control of the Jump Function Bus, formulates the address that will be clocked into the Microprogram Address Register at the end of the current microinstruction cycle. The encoded information on the Jump Function Bus specifies the mode (i.e., the Jump Function) by which the Next Address Logic will formulate the next address, and, in addition, it supplies part of the next address itself. In performing a conditional Jump function, the Next Address Logic selects information from the Microprogram Address Register and the Jump Function Bus to form what may be termed a "base address."

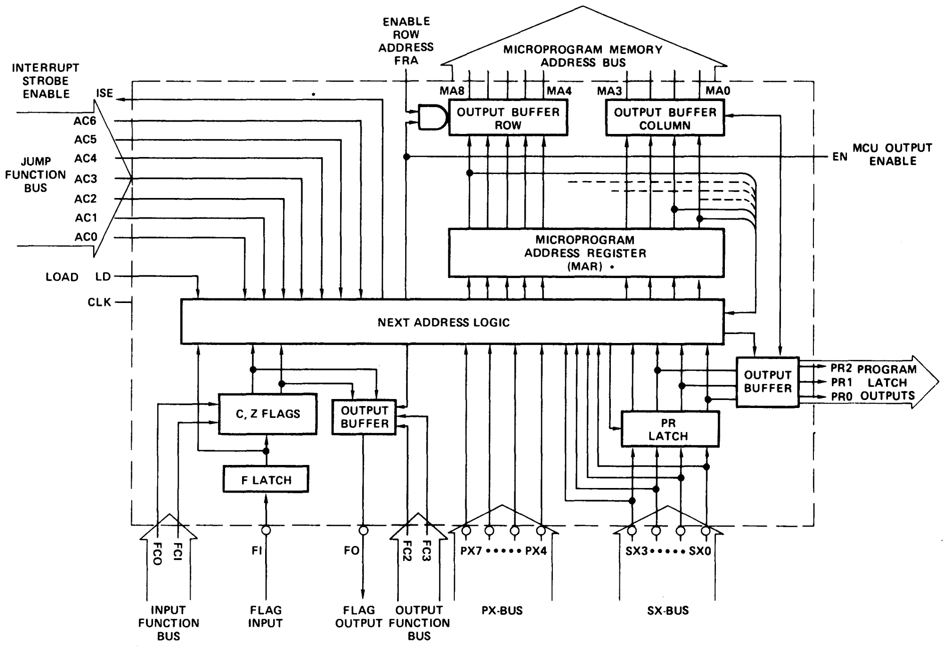
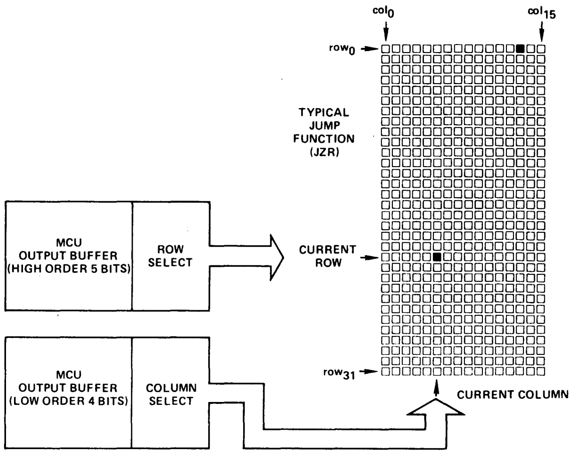


Fig. 1-5. 3001 MCU Organization



col0
col15
row0
TYPICAL JUMP FUNCTION (JZR)
MCU OUTPUT BUFFER (HIGH ORDER 5 BITS)
ROW SELECT
CURRENT ROW
MCU OUTPUT BUFFER (LOW ORDER 4 BITS)
COLUMN SELECT
row31
CURRENT COLUMN

Fig. 1-6. Microprogram Memory Addressing

The Next Address Logic then selects one, two or four bits of information from the latch or bus being tested to supply a "displacement" to the "base address," which serves to form the complete next address. Consequently, there are two possible outcomes of a Jump/test F-latch function (JFL), whereas there are 16 possible outcomes of a Jump/test PX bus function (JPX).

AUXILIARY OPERATIONS CONTROLLED BY JUMP FUNCTIONS

Certain Jump functions also control auxiliary operations in the MCU. The JPX jump function not only specifies a conditional jump based on the information on the PX-bus, it also specifies that information in the PR-latch can be tested during some subsequent cycle via the JPR Jump function. The JCE jump function performs the additional function of enabling the three low order bits of the PR-latch (PR0, PR1, and PR2). These three MCU outputs constitute a special control facility that may be applied in a particular configuration as the design engineer sees fit.

INTERRUPT

The MCU jump control logic supports an interrupt facility; the JZR Jump function plays a special role in the interrupt scheme. When the MCU executes a JZR Jump function that specifies column address 15 (row address 0 is implied by the JZR Jump function), the MCU activates its Interrupt Strobe Enable (ISE) output. If an interrupt condition is pending, as detected by external interrupt logic such as the Intel® 3214 Interrupt Control Unit, the external interrupt logic responds to ISE by disabling the MCU's row address outputs (via the MCU's Enable Row Address input) and forcing an alternate row address onto the microprogram memory row address lines. The result is that the next microinstruction word is accessed from the alternate row (typically row 31) of column 15 of the microprogram memory rather than from row 0 of column 15. The alternate location is typically the beginning of an interrupt service microroutine.

EXTENDED ADDRESSING

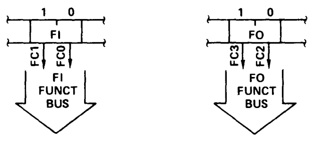
The MCU supports a basic microprogram memory addressing capability of 512 words; a larger addressing capability can be implemented, however. External registers may be added to serve as an extension to the MCU's 9-bit Memory Address Register. These latches are loaded, at the end of each microinstruction cycle, directly from an address extension field in the currently executing microinstruction word. For example, a two-bit address extension field in the microinstruction word (with two external address extension registers) plus the 9-bit MCU address provide a total microprogram memory addressing capability of 2048 words.

ADDRESS EXTENSION FIELD

To understand the function of an address extension field, it is useful to conceptualize the microprogram memory as a three-dimensional matrix of planes, rows and columns, as illustrated in Figure 1-7. The address extension field supplies the plane address, while the MCU (under control of the JUMP field) supplies the row and column addresses.

FI AND FO FIELDS

In a typical configuration, two 2-bit fields in the microinstruction word, called the FI field and FO field, drive the MCU's Flag Input and Flag Output Busses, respectively.



The MCU's Flag Output and Flag Input functions are summarized in Tables 4-2 and 4-3.

FLAG INPUT FUNCTIONS

The MCU's Flag Input functions provide a mechanism for maintaining the status of selected CPE array operations. Some attribute of the result of every CPE array operation is reflected on either the Carry Output or the Shift Output of the array, depending upon the operation. Since only one of these status outputs is active for any given operation, these two lines are typically tied together and connected to the MCU's Flag Input. During every microinstruction cycle, the

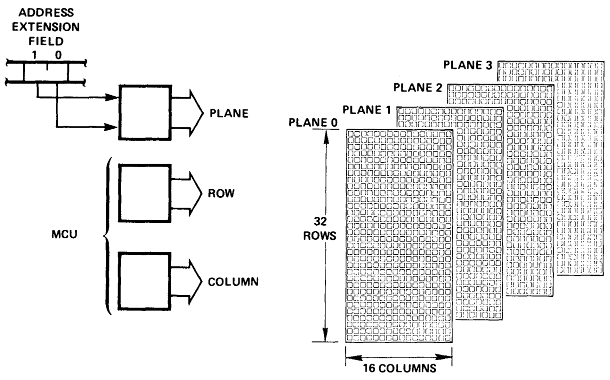


Fig. 1-7. Microprogram Memory Extension

F-LATCH

F-latch in the MCU automatically stores the state of the Flag Input. Consequently, the state of the F-latch reflects the status of the current array operation.

C AND Z FLAGS

It is often desirable to save the status of a particular result over several microinstruction cycles. The C and Z flags provide this capability. The behavior of these flags is controlled by the 2-bit Flag Input Function Bus to the MCU. During a microinstruction cycle, the control information on the Flag Input Function Bus determines whether the state on the Flag Input is to be stored in the C-flag, the Z-flag, both flags or neither flag. (The states of the F-latch, the C-flag and the Z-flag can be tested via the MCU jump functions JFL, JCF and JZF, respectively.)

FLAG OUTPUT FUNCTIONS

The MCU's Flag Output is typically connected to the Carry Input and the Shift Input to the CPE array. Consequently, the state of the Flag Output affects the result of any arithmetic or shift right operation performed by the array. The MCU's 2-bit Flag Output Function Bus controls the state of the Flag Output. During a microinstruction cycle, the control information on the Flag Output Function Bus determines whether the Flag Out is forced to a "0" state, a "1" state, the state of the C-flag or the state of the Z-flag.

SUMMARY

To the microprogrammer, a microprogrammable processor represents a computing machine with special properties that must be directed by a microprogram to perform a particular processing task. A microprogram is composed of a series of individual microinstructions that are selected and executed by the machine in timed sequence. In a basic machine cycle, one microinstruction is selected for execution. That microinstruction dictates the operations to be performed or initiated by each functional unit in the system during that execution cycle. The microinstruction also provides information to determine which microinstruction in the microprogram is to be selected for execution during the next machine cycle.

In order to write an effective and efficient microprogram, the microprogrammer must have a thorough understanding of his microprogrammable processor configuration. He must understand the functions that each group of logic in the configuration is capable of performing and how those functions relate to each other and to the microinstruction word format. Functional descriptions of the CPE and MCU, from the microprogrammer's point of view, are provided in Sections 4 and 5, respectively. A functional description of a user-configured system, however, must be provided by the design engineer who specifies the configuration.

1.2 CROSS MICROPROGRAMMING SYSTEM (CROMIS)

CROMIS

Intel's Cross Microprogramming System supports the Series 3000 user in the development and implementation of microprograms. CROMIS is supplied to the user in the form of two Fortran IV source programs, XMAS (Cross Microassembler) and XMAP (ROM Programming File Generator).

XMAS LANGUAGE

XMAS defines a symbolic, machine-oriented microprogramming language called the XMAS Language. The XMAS Language is specifically suited to the microprogramming characteristics of the Series 3000 computing elements. The important features of the XMAS Language include:

- A complete set of mnemonics to represent the various CPE and MCU functions;
- Symbolic addressing to simplify the representation of microprogram sequencing;
- A mechanism to define mnemonics representing the functions controlled by user-defined fields;
- Special support for the microinstruction word field that drives the CPE's K-bus;
- Special support for addressing a microprogramming memory greater than 512 words.

The XMAS Language simplifies the task of writing microprograms for Series 3000 configurations.

A fundamental function of CROMIS is to convert an XMAS Language microprogram into a format suitable for programming the individual ROM or PROM devices that will serve as the physical microprogram memory. This function involves two phases: the assembly of the XMAS Language program, and the generation of a ROM programming file. This functional division is provided because, during the development process, a microprogram will probably be assembled many times before the microprogram is mapped into the ROMs or PROMS. Consequently, CROMIS is supplied as two separate programs: XMAS and XMAP. The functional relationship between XMAS and XMAP is represented in Figure 1-8.

XMAS

During assembly, XMAS reads records from a source file. These records contain Control Language statements and XMAS Language statements. The Control Language statements specify I/O formats, select files and establish other parameters. The XMAS Language statements constitute the symbolic representation of the microprogram; they define new fields, define special symbols and specify the microinstruction bit patterns.

XMAS generates two files during an assembly: a list file and a microcode file. The list file may contain the source file statements, the bit patterns generated for those statements that specify microinstructions, error messages, a cross reference directory and a graphic representation of the microprogram memory image generated by the assembly. The microcode file is an intermediate binary file of the bit patterns generated by the assembly. XMAS error messages are listed in Appendix G.

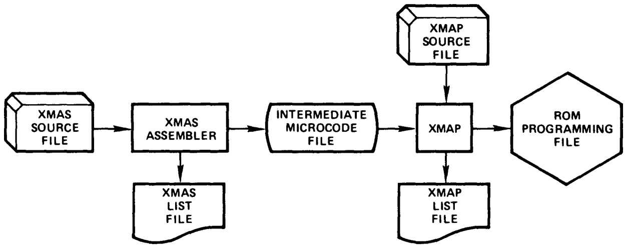


Fig. 1-8. Relationship Between XMAS and XMAP

XMAP

During ROM programming file generation, XMAP reads records from a source file that contains Control Language statements and ROM Mapping language statements. These statements specify the mapping of the microinstruction bit patterns in the microcode file, generated during assembly by XMAS, into the bit locations of the ROM's or PROM's that will be used for the microprogram memory.

In processing an XMAP language program, XMAP is capable of listing the following reports when directed to do so by the proper Control Language statements:

- XMAP language source statements with any error messages, and/or a binary dump of each ROM specified by a ROM mapping statement;
- XMAP program memory.

XMAP error messages are summarized in Appendix H.

SECTION 2
XMAS LANGUAGE

The XMAS language is an extensible microassembly language used to write microprograms for custom Series 3000 configurations. The XMAS language allows the microprogrammer to symbolically represent microinstructions and microprogram sequencing, without compromising his control over the exact coding of microinstruction words or their locations in the microprogram memory.

The XMAS language is specifically designed to support the microprogramming characteristics of the Series 3000 computing elements. The language includes a set of mnemonics for representing the various CPE and MCU functions, and it allows symbolic addressing to simplify the specification of microprogram sequencing.

One of the unique features of the XMAS language is its extensibility. Because of the wide variety of configurations that are possible with the Series 3000 computing elements, the detailed format of the microinstruction word cannot be built into the XMAS language. Consequently, the XMAS language provides a mechanism for describing extensions to the microinstruction word in the form of user-defined fields. The language also provides a mechanism for associating functional mnemonics with the new fields the user defines.

Section 2.1 provides a general introduction to the XMAS language. This introduction relies heavily on examples to illustrate the functions of XMAS language constructs. Sections 2.2 through 2.14 provide a formal description of XMAS language syntax and semantics. These sections are intended to serve as a reference manual for the XMAS language. In the reference sections, syntax is described using a modified BNF notation; this notation is described in Appendix A.

2.1 XMAS LANGUAGE OVERVIEW

LANGUAGE CHARACTERISTICS

The XMAS language is a free-format language. Syntactic entities may appear anywhere in a source record; the order in which entities appear is often, but not always, discretionary. Commas and spaces may be used freely and interchangeably to enhance readability.

RESERVED WORDS

IDENTIFIERS

The reserved words, which are listed in Table 2-2 of Section 2.2, constitute the permanent vocabulary of the XMAS language; the definitions of the reserved words are intrinsic to the XMAS language. In an XMAS language program the microprogrammer will have occasion to define symbolic names (formally called identifiers) to serve as names of new fields, function mnemonics, statement labels (for symbolic addressing), etc. All symbolic names in an XMAS language program, both the reserved words and user-defined identifiers, can have only one definition. In other words, a particular identifier may not serve as, say, both a field name and a statement label.

PROGRAM STRUCTURE

An XMAS language program is expressed as a series of "declaration statements" followed by a series of "specification statements." A semicolon (;) marks the end of each statement, and the reserved word EOF marks the end of the program. Stated symbolically, an XMAS language program has the following general form:

```
D; D;... D; S; S;... S; EOF
```

where the Ds represent declaration statements, and the Ss represent specification statements.

DECLARATION STATEMENTS

In general, the declaration statements in an XMAS language program establish the framework for writing the specification statements. Some of the functions of declaration statements are:

- To declare and describe new fields in the microinstruction word and associate functional mnemonics with the new fields;
- To establish a hierarchy of default bit pattern assignments for fields;

- To designate a K-bus field;
- To designate an address extension field;
- To define symbols that represent character strings or numeric values.

In all, there are six types of declaration statements, which are identified by the reserved words: FIELD, IMPLY, KBUS, ADDRESS, VALUE and STRING. They are all discussed in this subsection.

SPECIFICATION STATEMENTS

Specification statements are the active elements of an XMAS language program. There is a one to one correspondence between a specification statement and a microinstruction word. In the context of a program, each specification statement provides XMAS, the microassembler, with sufficient information to generate the bit patterns for every field in a single microinstruction word and to assign that word to a particular location in the microprogram memory.

MICROINSTRUCTION WORD FORMAT

INTRINSIC FIELDS

The architecture of the Series 3000 configuration prescribes the format of the microinstruction word. Because the Intel 3001 Microprogram Control Unit (MCU) and an array of Intel 3002 Central Processing Elements (CPE array) are assumed to be central to any Series 3000 configuration, certain assumptions about the microinstruction word format can be made: it must have a field to govern the function of the CPE array, and it must have fields to govern the Jump function, the Flag Input function and the Flag Output function of the MCU. Consequently, XMAS initially assumes a basic microinstruction word of the following form:

```
 6 5 4 3 2 1 0 1 0 1 0 6 5 4 3 2 1 0
|     CPE      | FI | FO |   JUMP     |
```

The reserved words CPE, FI, FO and JUMP are called intrinsic field names. They may be used in an XMAS language program to refer to their respective fields, as discussed later.

USER-DEFINED FIELDS

In most cases, the microinstruction word must control other logic in the microprocessor. Consequently, the microinstruction word must be extended to include some number of additional fields. For example, the microinstruction word might be required to supply the K-bus inputs to the CPE array. Additional fields might be required to control an external main memory and other logic functions within the configuration. For a particular configuration, the complete microinstruction word might have the following format:

```
 6 5 4 3 2 1 0 1 0 1 0 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 2 1 0 3 2 1 0
|     CPE      | FI | FO |   JUMP     |       KB       | MEM | CONT |
```

Here, the symbols KB, MEM and CONT are arbitrarily chosen to represent the new fields.

It is important to note that the microinstruction word pictured above is only a logical representation of the microinstruction format. Each field should be viewed as an independent entity. There may, of course, be functional relationships between the fields. For example, the CPE field, the FI field and the K-bus field (KB in this example) all play a role in determining the functions performed by the CPE array. This does not, however, imply any positional relationship between these fields within the microinstruction word.

FIELD DECLARATIONS

As previously mentioned, XMAS initially assumes the existence of only the four intrinsic fields: CPE, FI, FO and JUMP. In the XMAS language, additional fields must be declared via FIELD statements. For example, the FIELD statements:

INTRODUCTION TO THE FIELD STATEMENT

```
KB     FIELD   LENGTH=8;

MEM    FIELD   LENGTH=3;

CONT   FIELD   LENGTH=4;
```

declare the new fields KB, MEM and CONT. The number on the right hand side of the assignment operator (=), which follows the reserved word LENGTH, specifies the number of bit positions to be allocated to the microinstruction word for the new field. The symbol on the left hand side of the reserved word FIELD is the name the microprogrammer has chosen to associate with the new field. Names defined in this context are called user-defined field names. Like the intrinsic field names, user-defined field names are used in other statements to refer to their respective fields.

SPECIFICATION STATEMENT

Once the programmer has described all extensions of the microinstruction word via FIELD statements, he has established the framework required to write complete specification statements. Each specification statement must provide XMAS with sufficient information to assemble a complete microinstruction word (i.e., a bit pattern for every field) and to locate that word in the microprogram memory. A specification could take the following form:

PRIMITIVE FORM OF THE SPECIFICATION STATEMENT

```
70:  CPE=23  FI=3  FO=0  JUMP=37  KB=0  MEM=0  CONT=2;
```

The number preceding the colon (:) specifies the microprogram memory location for the microinstruction word represented by the statement. The values following the assignment operators (=) specify the bit patterns for the fields associated with the field names: CPE, FI, FO, JUMP, KB, MEM and CONT.

REMARKS ABOUT STATEMENT FORMAT

The order in which field bit pattern assignments occur in a specification statement is arbitrary. The preceding statement could just as well have been written as:

```
70:  CPE=23, KB=0, FO=0
     FI=3
     CONT=2, MEM=0
     JUMP=37;
```

Here, the statement has been arranged to emphasize the functions to be performed by the microinstruction word that the statement represents.

This example also illustrates the free format nature of the XMAS language. Commas, which are the equivalent of spaces in the XMAS language, may be used to separate statement entities. Since a semicolon (;) is required to mark the end of a statement, a statement may be continued on any number of lines; also, two or more statements may appear on the same line. These remarks apply to declaration statements as well as specification statements.

ORDER IN WHICH SPECIFICATION STATEMENTS APPEAR IN A PROGRAM

Since every specification statement must include an explicit microprogram memory address assignment, specification statements may appear in any order in an XMAS language program. Typically, the microprogrammer will write specification statements in an order that reflects the execution sequence of the microprogram.

INTEGER REPRESENTATIONS

In the preceding examples, all values have been represented as decimal integers. The XMAS language also supports hexadecimal (H), octal (O or Q) and binary (B) integer representations. The same statement can be written:

```
46H:  JUMP=0100101B  MEM=0  CONT=2  CPE=27Q  FI=3  FO=0  KB=0;
```

A base designator (H, O, Q or B) must follow the digits in all integer representations except decimal (where the base designator D is optional). A hexadecimal integer must begin with a decimal digit so that it may be distinguished from an XMAS identifier; a leading zero is always sufficient.

EXPRESSIONS

The value of the microprogram memory location assignment must be represented by an integer, but the values for field assignments may be represented by expressions. For example, the JUMP field could have been specified by:

```
JUMP=5+(010B SHL 4)
```

where + is the addition operator and SHL is the shift left operator. The parentheses are not actually required in this case because the SHL operator has a higher precedence than the + operator. Other operators that may appear in expressions are: NOT (logical complement), AND (logical AND), OR (logical OR), XOR (logical exclusive OR) and SHR (shift right). (The formal rules that govern expressions are presented in Section 2-5.)

KEYWORD ASSIGNMENT FORM

In the example specification statements above, all field assignments took the form:

```
field identifier = expression
```

This form of field assignment is called a keyword assignment. Although a keyword assignment may always be used, it is often not the most convenient form for specifying a field.

A keyword assignment tends to emphasize the bit pattern itself rather than the function that the bit pattern designates. When writing a specification statement, the microprogrammer is more concerned with specifying the functions to be performed by the microinstruction; he does not really care how those functions may be encoded in the fields of the microinstruction word.

MICROPS

INTRINSIC MICROPS

Another form of field assignment involves a special type of symbol called a "microp" (pronounced mike-ro-op). A microp is similar to an instruction mnemonic in a conventional assembly language. The XMAS language includes a set of microps (listed in Table 2-2) for each of the four intrinsic fields: CPE, FI, FO and JUMP. Each microp is a mnemonic for one of the functions controlled by its respective field.

FI AND FO MICROPS

The four microps associated with the FI field are mnemonics for the four Flag Input functions of the MCU. For example, the FI microp HCZ represents the Flag Input function "Hold C and Z Flags." Likewise, the four microps associated with the FO field are mnemonics for the four Flag Output functions of the MCU. For example, the FO microp FF0 represents the Flag Output function "Force Flag Output to Zero." These microps could appear in a specification statement to specify the FI and FO fields:

```
46H:  JUMP=25H  MEM=0  HCZ  FF0  CONT=2  KB=0  CPE=27Q;
```

The microps HCZ and FF0 make explicit bit pattern assignments for the FI and FO fields in this statement. (In fact, HCZ is equivalent to the keyword assignment FI=3 and FF0, to FO=0.)

CPE MICROPS

The encoding of the CPE field designates both the function to be performed by the CPE array and the internal CPE register that is to participate in that function. The microps associated with the CPE field are mnemonics for the CPE functions. The XMAS language also includes mnemonics for the internal CPE registers; these mnemonics are called register names (listed in Table 2-2). In a specification statement, a CPE microp must always be followed by a register name enclosed in parentheses. For example,

```
LMI (R7)
```

means that the function represented by the CPE microp LMI is to be performed with register R7. In the specification statement:

```
46H:  JUMP=25H  MEM=0  HCZ  FF0  KB=0  CONT=2  LMI (R7);
```

LMI (R7) makes an explicit bit pattern assignment for the CPE field.

JUMP MICROPS

The microps for the JUMP field are mnemonics for the various methods by which the MCU may develop an address to access the microprogram memory for the next microinstruction in a program sequence. In a specification statement, a JUMP microp must always be followed by

TABLE 2-1. XMAS CHARACTER SET SUMMARY

CHARACTER	USED
A through Z (alphabetics) 0 through 9 (numerics)	• To compose reserved words, user-defined identifiers and integers
; (semicolon)	• Separates statements within program
: (colon)	• Delimits the microprogram memory address and statement labels in a specification statement.
= (equals)	• Assignment operator.
+ (plus)	• Arithmetic addition operator.
( ) (parentheses)	• Used to indicate precedence in an expression. • Used to enclose operands of CPE and JUMP microps.
' (single quote)	• To enclose a character string.
$ (dollar sign)	• To introduce a Control Language statement.
* (asterisk)	• Used in certain contexts to inhibit XMAS addressing checking.
/* (slash-asterisk) */ (asterisk-slash)	• To mark the beginning (/*) and end (*/) of a comment.
. (period)	• Interpreted as a null character, only used to punctuate integers and identifiers.
, (comma) (space)	• Used to delimit identifiers and integers where no other delimiter appears. • Used to enhance readibility of statements.

TABLE 2-2. XMAS RESERVED WORDS

	DECLARATIONS			MICROPS						
				FIELD						
PROGRAM TERMINATOR	STATEMENT TYPE	FIELD STATEMENT KEYWORDS	INTRINSIC FIELD NAMES	JUMP	FI	FO	CPE		CPE REGISTER NAMES	OPERATORS
EOF	FIELD	LENGTH	JUMP	JCC	HCZ	FFC	ACM	INR	AC	AND
	IMPLY	DEFAULT	FI	JCE	SCZ	FFZ	ADR	LCM	R0	NOT
	KBUS	MICROPS	FO	JCF	STC	FF0	AIA	LDI	R1	OR
	ADDRESS		CPE	JCR	STZ	FF1	ALR	LDM	R2	XOR
	VALUE			JFL			AMA	LMF	R3	SHL
	STRING			JLL			ANI	LMI	R4	SHR
				JMP			ANM	LMM	R5	
				JPR			ANR	LTM	R6	
				JPX			CIA	NOP	R7	
				JRL			CLA	ORM	R8	
				JZF			CLR	ORR	R9	
				JZR			CMA	SDA	T	
							CMR	SDR		
							CSA	SRA		
							CSR	TZA		
							DCA	TZR		
							DSM	XNI		
							ILR	XNM		
							INA	XNR		

one or more operands enclosed in a single set of parentheses. An operand of a JUMP microp is an expression that represents the microprogram memory address of a microinstruction word that is a target of the JUMP function. The JUMP microp JZR represents an unconditional branch to row zero of the microprogram memory. In the statement:

```
46H: JZR (05H) MEM=0 HCZ FF0 KB=0 CONT=2 LMI (R7);
```

JZR (05H) specifies that the next microinstruction in the microprogram sequence is to be taken from microprogram memory location 05_{16}. JUMP microps that represent conditional Jump functions require two, four or sixteen operands depending upon the number of possible states that the condition being tested can have. The JUMP microp JFL represents a two-way conditional branch determined by the state of the F-latch in the MCU. It could appear in a specification statement as:

```
JFL (12H 13H)
```

where 12_{16} is the branch destination address if the F-latch is reset, and 13_{16} is the branch destination address if the F-latch is set.

SYMBOLIC ADDRESSING

A specification statement may include one or more symbolic statement labels. The specification statement:

```
05H: ACCESS: NEXT: JZR (4) CONT=1 KB=0 MEM=2 HCZ FF0 LMI (R7);
```

ADDRESS IDENTIFIERS

defines the two symbols ACCESS and NEXT, both of which have the value 05_{16} (the microprogram memory location assignment for the microinstruction word specified by this statement). A symbol defined in this context is called an address identifier because it represents a particular location in the microprogram memory. Address identifiers may be used as operands of JUMP microps, allowing the programmer to represent program sequencing symbolically:

```
46H: START:    ...  JZR (ACCESS)          ...;
05H: ACCESS:   ...  JCR (DO)              ...;
01H: DO:       ...  JCC (TEST)            ...;
31H: TEST:     ...  JFL (FRESET FSET)     ...;
12H: FRESET:   ...  JZR (FINISH)          ...;
13H: FSET:     ...  JZR (FINISH)          ...;
06H; FINISH:   ...  JCC (START)           ...;
```

XMAS ADDRESS CHECKING

As XMAS assembles specification statements, it performs Jump function checking. If XMAS determines that a specification statement is referencing another specification statement that is not within the range of the Jump function, XMAS will output an error message to its list file. The following statements illustrate an addressing error:

```
46H: START:    ...  JZR (ACCESS)          ...;
                .
                .
07H: FINISH:   ...  JCC (START)           ...;
```

The microp JCC represents the Jump function "Jump in Current Column." Since the micro-instruction words specified by these two statements have not been assigned to the same column in the microprogram memory, XMAS will report an addressing error when it assembles the second statement.

SPECIAL JUMP MICROP JMP

The JUMP microps JCC, JCR and JZR are mnemonics for the MCU's unconditional Jump functions "Jump in Current Column," "Jump in Current Row" and "Jump in Row Zero," respectively. The special JUMP microp JMP represents all of these unconditional Jump functions; in other words, JMP may be used in place of JCC, JCR or JZR in any specification statement.

```
46H: START:    ...  JMP (ACCESS)          ...;
05H: ACCESS:   ...  JMP (DO)              ...;
01H: DO:       ... .JMP (TEST)            ...;
31H: TEST:     ...  JFL (FRESET FSET)     ...;
12H: FRESET:   ...  JMP (FINISH)          ...;
13H: FSET:     ...  JMP (FINISH)          ...;
06: FINISH:    ...  JMP (START)           ...;
```

For each occurrence of the microp JMP, XMAS will attempt to select the appropriate unconditional Jump function. If this is not possible, XMAS will report an addressing error. (There are no addressing errors in this example.)

DEFAULT ADDRESSING

In cases where specification statements appear in the order in which the microinstructions they represent are to be unconditionally executed, an explicit specification for the JUMP field is not required. Compare the preceding example with the following:

```
46H: START:    ...                        ...;
05H: ACCESS:   ...                        ...;
01H: DO:       ...                        ...;
31H: TEST:     ...  JFL (FRESET FSET)     ...;
12H: FRESET:   ...  JMP (FINISH)          ...;
13H: FSET:     ...                        ...;
06H: FINISH:   ...  JMP (START)           ...;
```

Here, Jump functions have been specified only where they are required. XMAS will attempt to supply the appropriate unconditional Jump function when none is specified.

USER-DEFINED MICROPS

MORE ABOUT THE FIELD STATEMENT

Microps for the CPE, FI, FO and JUMP fields are intrinsic; that is, they are part of the XMAS language. The XMAS language provides a facility that allows the programmer to define microps to be associated with the new fields he creates. Microps for a given field are defined in the FIELD statement that creates that field. The FIELD statement:

```
MEM  FIELD  LENGTH=3  MICROPS (READ=2 WRITE=3);
```

defines the field MEM and allocates three bit positions to it. The statement also defines two microps that are to be associated with the new field MEM: READ and WRITE. READ carries a bit pattern of 10_2 for the MEM field, and WRITE carries a bit pattern of 11_2 for the MEM field. In the specification statement:

```
24H:FETCH:  JMP (ACCESS)  READ  LMI (R7)  HCZ  FF0  KB=0  CONT=2;
```

READ takes the place of the keyword assignment MEM=2.

EXPLICIT BIT PATTERN ASSIGNMENT FORMS

Most of the specification statements shown in the preceding examples include an explicit bit pattern assignment for every field in the example microinstruction word: CPE, FI, FO, JUMP, MEM, CONT, and KB. (The only exceptions presented so far have been the cases where the order of the specification statements implied the coding for the JUMP field.) In a specification statement, an explicit assignment for a field can take one of two forms: a keyword assignment or a microp assignment. But there are a number of other non-explicit mechanisms that the programmer can employ to specify fields.

DEFAULTS

XMAS provides default bit pattern assignments for the FI and FO fields if these fields are not otherwise specified in a specification statement. The default for the FI field is 11_2, which indicates the Flag Input function "Hold C and Z Flags" (HCZ); the default for the FO field is 00_2, which indicates the Flag Output function "Force Flag Output to Zero" (FF0).

```
32H:STEP:  IRL (R7)  FF1  READ  KB=0  JUMP (FETCH);
```

Since FI is not specified in this statement, XMAS assigns FI its default bit pattern (11_2). The FO microp FF1 ("Force Flag Output to One"), however, provides an explicit assignment for FO, which overrides the default.

USER DECLARED FIELD DEFAULT

When the programmer creates a new field via a FIELD statement, he may declare a default bit pattern assignment for that field. The FIELD statement:

```
CONT  FIELD LENGTH=4  DEFAULT=6
```

defines the new field CONT and declares a default bit pattern of 0110_2 for this field. The specification statement:

```
84H:  CLR (R2)  JZR (FETCH)  FF0  HZC  READ  KB=0;
```

does not include a bit pattern assignment for the field CONT. Consequently the assembler assigns CONT its default bit pattern as if CONT = 6 had actually appeared in this specification statement. The default for a given field that is declared in the FIELD statement may be overridden in a specification statement by specifying the field in some other way. For example, the statement:

```
84H:  CLR (R2)  JZR (FETCH)  FF0  HZC  READ  KB=0  CONT=2;
```

includes an explicit assignment for CONT, which overrides this field's default assignment.

CPE MICROP K-BUS DEFAULTS

In Series 3000 configurations, a user-defined field in the microinstruction word provides either direct or encoded drive for the K-bus inputs to the CPE array. Because the K-bus participates in all CPE functions, the field serving as the K-bus field has a modifying effect on all CPE functions. Consequently, the CPE microps, which are mnemonics for CPE functions, are divided into two groups: those that imply an all-zero default for the K-bus field and those that imply an all-one default for the K-bus field. The CPE microp CLR represents a "clear register" function, and the CPE microp ANR represents an "AND register with AC" function, but CLR (R2) and ANR (R2) both represent the identical bit pattern for the CPE field. They differ only in that CLR implies an all-zero bit pattern for the K-bus field.

KBUS STATEMENT

The K-bus field default that each CPE microp implies is built into XMAS. However, if these defaults are to be effective, the programmer must identify which of the fields he has defined is to receive these defaults. The KBUS statement provides this mechanism. The statements:

```
KB  FIELD  LENGTH = 8;
KB  KBUS;
```

serves to define KB and to identify that field as the K-bus field, respectively. Note that the FIELD statement is still necessary to define the field KB.

In the specification statement:

```
84H: CLR (R2)  JZR (FETCH)  FFO  HZC  READ  CONT=2;
```

the bit pattern for the KB field is given by the K-bus default provided by the CPE microp CLR. The defaults provided by the CPE microps can always be overridden by specifying the KB field in another way.

```
84H: CRL (R2)  JZR (FETCH)  FFO  HZC  READ  CONT=2  KB=OFH;
```

Here, the explicit assignment for KB overrides the all-zero default assignment that CLR provides for KB.

IMPLY STATEMENT

A microp carries an explicit bit pattern assignment only for the field for which the microp is defined. However, a microp can also carry default bit pattern assignments for fields other than the one for which the microp is defined. The IMPLY statement is used to declare the defaults that a user-defined microp carries.

```
CONT  FIELD  LENGTH=4  MICROPS  (LOAD=7  GATE=17Q  STOP=2  RUN=3);
RUN  IMPLY  FI=2  MEM=2  KB=OFH
LOAD  IMPLY  MEM=4
```

The FIELD statement defines the field CONT and the microps LOAD, GATE, STOP and RUN to be associated with the field CONT. The first IMPLY statement declares the defaults that the microp RUN carries for the FI, MEM and KB fields. The second IMPLY statement declares the default that the microp LOAD carries for the MEM field.

```
81H:  CLR (R4)  JZR (NEXT)  FFO  RUN  WRITE;
```

Here, the default for the FI field that the CONT microp RUN carries serves to specify the FI field since this field is not otherwise specified in this statement. Both CLR and RUN carry defaults for the K-bus field KB; however, RUN's default for KB overrides CLR's default for KB. RUN also supplies a default for the MEM field; however, the microp WRITE is an explicit assignment for the MEM field and, consequently, overrides RUN's default for the MEM field.

DEFAULT HIERARCHY

As the preceding examples have implied, there is a hierarchy for the various mechanisms by which the bit pattern for a given field may be supplied in a specification statement. The following list identifies the mechanisms presented so far in their order of decreasing effectiveness.

1. An explicit keyword or microp assignment;
2. A default carried by a user-defined microp as declared in an IMPLY statement;
3. In the case of the field declared to be the K-bus field, the default carried by a CPE microp;
4. The default declared for the field in the defining FIELD statement.

An explicit assignment overrides any form of default assignment. A microp carries an explicit assignment only for the field for which it is defined. A user-defined microp carries a default assignment for one or more other fields if that microp has appeared in an IMPLY statement. The default carried by a user-defined microp for a given field is used in the absence of an explicit assignment for that field. When a field is defined in a field statement, a default bit pattern may be declared for that field. This default is overridden by any other form of specification for that field.

ADDRESS EXTENSION

ADDRESS EXTENSION FIELD

The MCU, under control of the Jump field, supports a basic microprogram memory addressing capability of 512 words (i.e., the MCU developes a 9-bit address). In Series 3000 configurations with a microprogram memory larger than 512 words, extra address information is typically provided by an address extension field in the microinstruction word. For example, in a microinstruction word of the format:

```
6 5 4 3 2 1 0 1 0 1 0 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 2 1 0 3 2 1 0 1 0
|     CPE     | FI | FO |    JUMP     |      KB       | MEM | CONT |XA|
```

the field XA is a 2-bit address extension field. This field may be viewed as providing the information required to select one of four possible planes in the microprogram memory. (Each plane consists of 32 rows and 16 columns, as addressed by the MCU.) Thus, for this example, the location of microinstruction word is identified by an 11-bit address, consisting of a plane address (2-bits), a row address (5-bits) and a column address (4-bits).

ADDRESS STATEMENT

The microprogrammer declares an address extension field in the following way:

```
XA  FIELD  LENGTH = 2;
XA  ADDRESS;
```

The field statement defines a new 2-bit field called XA. The ADDRESS statement identifies the XA field as an address extension field.

Declaring a field to be an address extension field completely relieves the programmer of the responsibility of making a bit pattern assignment for this field in a specification statement, as the following example demonstrates.

```
463H: NEXT   ...                       ...;
223H:        ...  JMP (NEXT)           ...;
```

In processing the second statement, XMAS will use the 2 high-order bits of the jump destination address (463H:) as the bit pattern assignment for the XA field.

$$463H{:}\rightarrow \quad \underbrace{1\ 0}_{\text{plane}}\ \underbrace{0\ 0\ 1\ 1\ 0}_{\text{row}}\ \underbrace{0\ 0\ 1\ 1}_{\text{column}}$$

In this case, the XA field will receive a bit pattern assignment of 10_2, which is the plane address of the target microinstruction represented by the first statement.

VALUE STATEMENT

Programmers often find it desirable to use a symbol to represent a frequently used constant. The VALUE statement allows the programmer to define a symbol and associate a numeric value with it.

For example, the VALUE statements:

```
MASK1  VALUE  0FH;
MASK2  VALUE  NOT MASK1 AND 0FFH;
```

define the symbols MASK1 and MASK2. The expression in the second statement gives MASK2 a value of F0H.

VALUE IDENTIFIER

A symbol defined in a VALUE statement is called a value identifier. A value identifier may be used anywhere an expression is allowed. For example, in the specification statement:

```
84H:  TZR(R2)  KB = MASK2  READ  RUN  JZR(FETCH);
```

MASK2 supplies the bit pattern for the KB field.

STRING STATEMENT

A microprogrammer may find that a particular group of symbols occur frequently in his XMAS language program. For example,

```
READ  RUN  JZR(FETCH)
```

might often appear together in specification statements. The STRING statement allows the programmer to define a symbol and associate a character string with that symbol. For example, the STRING statement:

```
DONE  STRING  'READ  RUN  JZR(FETCH)';
```

associates DONE with the string of characters enclosed in the single quote (') characters. DONE can be used in any specification statement in place of the character string it represents. For example, the specification statements:

```
84H:  TZR(R2)  KB=MASK2  DONE;
```

and

```
84H:  TZR(R2)  KB=MASK2  READ  RUN  JZR(FETCH);
```

are functionally equivalent.

COMMENTS

Comments can make a program more readable and greatly enhance the program's documentation value. In the XMAS language, any sequence of characters enclosed in /* and */ is treated as a comment.

```
/* THIS IS A COMMENT */
```

A comment may appear anywhere a blank character is allowed.

2.2 STATEMENT CHARACTERISTICS, CHARACTER SET AND RESERVED WORDS

SOURCE FILE

XMAS reads source statements from a source file. An XMAS source file consists of a sequence of physical records, each of which may contain up to 120 characters. The XMAS source file is discussed again in Section 3.1.

CONTROL COMMANDS

A dollar sign($) in character position one of input record is used to identify a Control Language statement. Control Language statements are used to select certain assembly option such as list file format. Strictly speaking, Control Language statements are not considered part of the XMAS Language, and their discussion is deferred to Section 9.

FREE FORMAT

XMAS Language statements are free format; physical record boundaries and character position within a record are not significant. There are no restrictions as to where a statement begins or ends or how long it is. A statement may span several records, or more than one statement may be wholly or partially contained within a single record. The semicolon (;), when it appears outside a comment, marks the end of a statement.

Spaces and commas may be used freely between statement elements to enhance readability. A comment may appear anywhere a space is allowed. XMAS treats the period (.) as a null character. The period may be used to punctuate only identifiers and integers (where spaces or commas are not allowed).

CHARACTER SET

The XMAS language character set includes the characters shown in Table 2-1. All characters not shown in the table are treated as spaces by XMAS.

RESERVED WORDS

Symbolic names that are intrinsic to the XMAS language are called reserved words. XMAS reserved words are listed by catagory in Table 2-2.

2.3 IDENTIFIERS

Symbolic names are formally called identifiers. The following rules govern the formation of an XMAS identifier:

SYNTAX

$$\langle\text{identifier}\rangle ::= \langle\text{letter}\rangle \begin{bmatrix}\langle\text{letter}\rangle\\ \langle\text{digit}\rangle\end{bmatrix} \ldots$$

```
<letter> ::= A!B!C!D!E!F!G!H!I!J!K!L!M!
             N!O!P!Q!R!S!T!U!V!W!X!Y!Z!

<digit> ::= 0!1!2!3!4!5!6!7!8!9
```

These rules state that an identifier is composed of a letter optionally followed by any combination of letters and digits.

IDENTIFIER LENGTH

There is no restriction as to the length of an identifier; however, only the first seven characters are significant in establishing the uniqueness of an identifier. Examples of valid identifiers are:

```
X  Y4Y  INPUTSTART  INPUTST
```

NULL CHARACTER

where the last two examples will be treated as the same identifier. The period (.), which is treated as a null character, may be freely inserted between the characters of an identifier to improve its readability. For example, the identifiers LEFTBIT and LEFT.BIT are equivalent, although their visual effects are different.

IDENTIFIER TYPES

The context in which an identifier is defined determines the attributes of that identifier and the way in which that identifier may be employed in other statements. These are five types of identifiers that the programmer may define in an XMAS language program. Table 2-3 summarizes these identifier types. For more detailed information about an identifier type, refer to the subsection that describes the defining statement.

UNIQUE DEFINITION

Every identifier in an XMAS language program must have a unique meaning. In other words, a given identifier can be defined only once in an XMAS language program. For example, the identifier ABC cannot be defined as, say, both an address identifier (i.e., a specification

TABLE 2-3. SUMMARY OF USER-DEFINED IDENTIFIER TYPES

IDENTIFIER TYPE	HOW DEFINED	ATTRIBUTES	VALID USAGE
Value identifier	In a VALUE statement.	Carries a numeric value.	May appear in any expression in any statement following the defining statement.
Address identifier	By appearing as a label on a specification statement.	Carries a numeric value of the microprogram memory location it represents.	May appear in any expression in any specification statement.
Field name (user-defined)	In a FIELD statement.	Symbolizes a field in the microinstruction word.	Used in statements to refer programmatically to the field it represents (e.g., keyword assignment in a specification statement).
Microp (user-defined)	By appearing in the MICROPS portion of a FIELD statement.	Carries an explicit bit pattern assignment for the field for which it is defined. If it has appeared in an IMPLY statement, carries default bit pattern assignments for one or more other fields.	May appear in one IMPLY statement. May appear in any specification statement.
String name	In a STRING statement.	Represents a character string.	May appear anywhere the character string it represents would be valid.

statement label) and a field name, even though the context is different. The XMAS reserved words are intrinsically defined; they may not be redefined by the programmer.

2.4 INTEGERS

The XMAS Language supports decimal, binary, octal and hexadecimal integer representations. The following rules govern the formation of XMAS language integers:

SYNTAX

```
⟨integer⟩ :: = ⟨decimal integer⟩!
               ⟨binary integer⟩!
               ⟨octal integer⟩ !
               ⟨hexadecimal integer⟩
```

$\langle\text{decimal integer}\rangle :: = \{\langle\text{decimal digit}\rangle\} \ldots [\text{D}]$

$\langle\text{binary integer}\rangle :: = \{\langle\text{binary digit}\rangle\} \ldots \text{B}$

$\langle\text{octal integer}\rangle :: = \{\langle\text{octal digit}\rangle\} \ldots \left\{\begin{matrix}\text{Q}\\ \text{O}\end{matrix}\right\}$

$\langle\text{hexadecimal integer}\rangle :: = \langle\text{decimal digit}\rangle\ [\langle\text{hexadecimal digit}\rangle] \ldots \text{H}$

```
⟨decimal digit⟩ :: = 0!1!2!3!4!5!6!7!8!9

⟨binary digit⟩ :: = 0!1

⟨octal digit⟩ :: = 0!1!2!3!4!5!6!7

⟨hexadecimal digit⟩ :: = 0!1!2!3!4!5!6!7!
                         8!9!A!B!C!D!E!F
```

BASE DESIGNATORS

The base designator for a decimal integer (D) is optional; the base designator for a binary integer (B), an octal integer (Q or O) or a hexadecimal integer (H) is required. The leading character of a hexadecimal integer must be a decimal digit; a leading zero is always sufficient.

Examples of valid XMAS integers are:

2 32Q 110B 33FH 55D 55

NULL CHARACTER

The period (.), which is treated as a null character, may be freely inserted between the characters of an integer to improve its readability. For example, the integers 3467 and 3.467 are equivalent, although their visual effects are different.

An integer is a digit string that represents a number. The range of legitimate integers is:

INTEGER RANGE

$$0 \leqslant i \leqslant 2^{64} - 1;$$

that is, integers may have a 64-bit precision. If an integer exceeds this value, an error message will be output to the list file, a one value will be substituted, and assembly will proceed.

2.5 EXPRESSIONS

The following rules govern the formation of an XMAS expression:

$$\langle\text{expression}\rangle ::= \langle\text{term}\rangle \left[\begin{matrix} + \\ \left\{\begin{matrix}\text{OR}\\ \text{XOR}\end{matrix}\right\}\end{matrix}\right] \langle\text{TERM}\rangle \quad \ldots$$

⟨term⟩ :: = ⟨subterm⟩ [AND⟨subterm⟩] . . .

$$\langle\text{subterm}\rangle ::= \langle\text{factor}\rangle \left[\begin{Bmatrix} \text{SHL} \\ \text{SHR} \end{Bmatrix} \langle\text{factor}\rangle \right] \ldots$$

⟨factor⟩ :: = [NOT] ⟨primary⟩

⟨primary⟩ :: = ⟨value identifier⟩!
⟨address identifier⟩!
⟨integer⟩!
(⟨expression⟩)

Examples of valid XMAS expressions are:

26H
FETCH
LOW+2
NOT (X AND Y OR Z)

where each of the identifiers (FETCH, LOW, X, Y, and Z) is assumed to be either a value identifier (defined in a value statement) or an address identifier (defined by appearing as a specification statement label). Other types of identifiers (e.g., microps and register names) are invalid in expressions.

An expression evaluation yields a single numeric result. XMAS obtains a value for an expression by performing the indicated arithmetic and logic operations on the actual numeric values of the primaries. Primaries are considered to have a 64-bit precision, and the result of an expression evaluation has a 64-bit precision. The result of an expression evaluation must be a value in the range:

PRECISION

RANGE

$$0 \leqslant \text{value} \leqslant 2^{64} - 1.$$

In the event of an overflow, only the least significant 64 bits are retained. A constant which exceeds 2^{64} is replaced by 1 and an error message is generated.

TRUNCATION

The context of an expression in an XMAS language program determines how many bits of the result are actually used. For example, if an expression is used to supply a value to be assembled into a four-bit field, the four low-order bits of the result of the expression evaluation are used. If any higher order bit of the result is not zero, XMAS outputs a truncation error warning message to the list file.

OPERATORS

Table 2-4 summarizes the XMAS language operators and identifies their precedence. If several operators of equal precedence occur at the same level of evaluation, then the order of evaluation among those operators is from left to right. Parentheses are used to override the assumed precedence.

2.6 COMMENTS

Comments are explanatory remarks, which may appear in an XMAS language program to improve the program's readability and documentation value. The form of an XMAS comment is:

⟨comment⟩ :: = /*⟨character string excluding * ⟩*/

TABLE 2-4. EXPRESSION OPERATORS

OPERATOR	MEANING	PRECIDENCE
NOT	Logical complement	First
SHL	Logical shift left	Second
SHR	Logical shift right	Second
AND	Logical AND	Third
OR	Logical OR	Fourth
XOR	Logical exclusive OR	Fourth
+	Arithmetic addition	Fourth

The following is an example:

```
/* THIS IS A COMMENT */
```

An XMAS comment may appear anywhere a blank character is permitted.

2.7 XMAS LANGUAGE PROGRAM

The following rules govern the structure of a program:

SYNTAX

```
⟨program⟩ :: = [⟨declaration part⟩]
                ⟨specification part⟩ EOF

⟨declaration part⟩ :: = { ⟨declaration statement⟩; } . . .

⟨declaration statement⟩ :: = ⟨string statement⟩!
                              ⟨value statement⟩!
                              ⟨field statement⟩!
                              ⟨imply statement⟩!
                              ⟨k-bus statement⟩!
                              ⟨address statement⟩

⟨specification part⟩ :: =  ⟨specification statement⟩;  . . .
```

The ⟨declaration part⟩ of a program is syntactically optional, but in nearly all practical programs a ⟨declaration part⟩ will be present. The ⟨specification part⟩ is, however, required; an error message will be issued if the ⟨specification part⟩ is missing. A program is terminated by the reserved word EOF.

SEMICOLON

A semicolon (;) is required to mark the end of each statement in the program. A missing semicolon, if detected, causes an error message to be issued. Also, an error message will be issued if a declaration statement appears in the ⟨specification part⟩, or vice-versa.

Subsequent subsections discuss each of the statement types.

2.8 STRING STATEMENT

The STRING statement defines a string identifier and associates it with a character string.

SYNTAX

```
⟨string statement⟩ :: = ⟨string identifier⟩ STRING
      {⟨character string⟩} . . .
⟨string identifier⟩ :: = ⟨identifier⟩
```

CONCATENATION OF STRINGS

The reserved word STRING is followed by one or more ⟨character string⟩s enclosed in single quote (') characters. When more than one quoted ⟨character string⟩ is present, concatenation of the individual strings is implied. For example the separate strings:

```
'ABC' 'DEF' 'GHI'
```

are equivalent to the single string:

```
'ABCDEFGHI'
```

When XMAS encounters a string identifier in the text of the program outside a comment or a STRING statement, XMAS treats the string identifier as if the ⟨character string⟩ it represents were actually there in its place. In other words, the microassembler substitutes the character string for the string identifier; however, the substitution does not appear in the list file.

STRING LENGTH

The implementation of XMAS on a particular host computer system imposes a restriction on the number of string characters the microassembler can store while assembling an XMAS language program. However, the restriction applies to the total number of characters in all ⟨character strings⟩ in the program rather than to the number of characters in a particular ⟨character string⟩. If the total program string length limit is exceeded, XMAS will output an error message to the list file, and abort the operation (i.e., XMAS will exit).

A string identifier may not appear in the quoted part of a STRING statement. With this exception, a ⟨character string⟩ may be composed of any sequence of characters. To be useful, however, the ⟨character string⟩ must have validity in the context of some other XMAS language statement. A single-quote character (') may be in a ⟨character string⟩ by representing it by two consecutive quotes. For example, the statement:

```
QUOTE STRING  '''';
```

defines the string identifier QUOTE and associates it with a ⟨character string⟩ consisting of one single-quote character.

EXAMPLE

The creation of a string identifier that may serve as a new microp that supplies a non-intrinsic default (not all-zeros or all-ones) to the K-bus field is one example of the use of the STRING statement. The statement:

```
SRAM   STRING 'KB = 101101B   SRA'
```

defines SRAM in such a way that it may be employed like an intrinsic CPE microp in specification statement. When

```
SRAM(AC)
```

appears in a specification statement, XMAS will treat it as if

```
KB = 101101B  SRA(AC)
```

has actually appeared. Note, however, that SRAM carries an explicit assignment for the field KB (assumed to be the user-defined K-bus field) rather than a default assignment.

2.9 VALUE STATEMENT

The VALUE statement defines a value identifier and associates it with a numeric value.

SYNTAX

⟨value statement⟩ :: = ⟨value identifier⟩ VALUE ⟨experssion⟩

⟨value identifier⟩ :: = ⟨identifier⟩

The value to be associated with the value identifier is the result of the evaluation of the expression. The expression may not contain any identifier other than a label that has not been declared in some previous VALUE statement.

Assuming that they appear in a program in the order shown, the following is an example of two valid VALUE statements:

EXAMPLES

```
ONE   VALUE  1;
NOT-TWO   VALUE   NOT (ONE +1) AND 0FFH;
```

The statement:

```
JZR  VALUE  47FH;
```

is invalid because the identifier JZR is reserved word.

A value identifier may be employed as a primary in an expression in any statement subsequent to the one in which that value identifier is declared. In other words, a value identifier must be declared before it is referenced.

Using a value identifier instead of an integer or a complex expression often makes a program easier to modify. For example, if a particular constant is represented as an integer in 20 specification statements in a program, all 20 statements would have to be changed in order to change the constant. If, however, that same constant is represented by a value identifier in every statement, only the declaring VALUE statement need be changed.

2.10 FIELD STATEMENT

The FIELD statement declares and describes a new field. The FIELD statement defines a field name to represent the new field, and it may optionally define microps to be associated with the new field.

SYNTAX

⟨field statement⟩ :: = ⟨field name⟩ FIELD ⟨field spec⟩ . . .

⟨field name⟩ :: = ⟨identifier⟩

⟨field spec⟩ :: = LENGTH = ⟨integer⟩ ! DEFAULT = ⟨expression⟩

MICROPS ({⟨microp⟩ = ⟨expression⟩} . . .)

FIELD NAME

The first rule states that a FIELD statement must consist of a field name followed by the reserved word FIELD; the reserved word FIELD may be followed by one or more ⟨field spec⟩s, although none are required, The field name that appears in the first rule is the symbolic name to be used to represent the field being created. The second rule defines the three forms that a ⟨field spec⟩ may take. Each of these forms may appear once at most in a given FIELD statement. There is no restriction as to the order in which the ⟨field spec⟩s may appear.

LENGTH

The LENGTH ⟨field spec⟩ is used to specify the length attribute of the field being created. The value following the assignment operator (=) is the number of bit positions in the microinstruction word to be allocated to the new field. This value must be expressed as an integer; more complex expressions are not allowed. If the LENGTH ⟨field spec⟩ does not appear in a FIELD statement, the field is allocated one bit position. For example, the statement:

```
K2  FIELD  LENGTH = 3;
```

creates a new field called K2 and allocates three bit positions to it. The statement:

```
K1  FIELD;
```

creates a new field called K1 and allocates one bit position to it.

DEFAULT

The DEFAULT ⟨field spec⟩ is used to declare a default bit pattern for the field being created. The default bit pattern is the truncated value of the expression that follows the assignment operator (=). The default bit pattern declared in the DEFAULT ⟨field spec⟩ is the lowest in the hierarchy of defaults for the field (field default hierarchy is discussed in Section 2.14). If the DEFAULT ⟨field spec⟩ does not appear in the FIELD statement, the field being created has no default bit pattern associated with it; that is, XMAS does not automatically supply a default bit pattern for the field. In the two example statements shown above neither the field K1 nor the field K2 has a default bit pattern associated with the field itself. (However, microps bound to other fields may imply a default bit pattern for these fields, as discussed in Section 2.11.)

MICROPS

The MICROPS ⟨field spec⟩ defines one or more microps and associates them with the field being created. The parentheses following the reserved word MICROPS enclose a list of one or more microp declarations of the form:

```
⟨microp⟩ = ⟨expression⟩
```

The truncated value of the expression is the bit pattern the microp represents for the field. A microp may imply default bit patterns for fields other than the one with which it is associated. However, the default bit patterns that are implied by a particular microp are declared in an IMPLY statement (Section 2.11), not in the FIELD statement. The statement:

```
CLOCK  FIELD  DEFAULT = 1  MICROPS  (NO.CLOCK = 0);
```

creates a one-bit field called CLOCK with a default bit pattern of 1, and it defines the microp NO.CLOCK, which is to be associated with that field and is to represent a bit pattern of 0 for that field. In the statement:

```
LOAD  FIELD  LENGTH = 2  DEFAULT = 0  MICROPS (LOADA = 1  LOADB = 3);
```

two microps are defined for the field LOAD.

MAXIMUM MICROINSTRUCTION SIZE

The concatenation of the intrinsic fields (JUMP, CPE, FI, and FO) constitutes the initial microinstruction word when assembly of an XMAS language program begins. As the micro-assembler encounters each FIELD statement, it concatenates the new field to the current microinstruction word. XMAS is capable of supporting a microinstruction word up to 64 bit positions. Because the intrinsic fields require a total of 18 bit positions, the combined requirement of user-defined fields cannot exceed 46 bit positions.

2.11 IMPLY STATEMENT

A microp may imply default bit patterns for fields other than the field for which it is defined. The IMPLY statement provides the mechanism for associating these default bit patterns with a microp. The following rules define the IMPLY statement:

SYNTAX

```
⟨imply statement⟩ :: = ⟨microp⟩ IMPLY ⟨imply list⟩
⟨imply list⟩ :: = {⟨field name⟩ = ⟨expression⟩} . . .
```

The ⟨microp⟩ may be either a user-defined microp (defined in a FIELD statement) or an intrinsic microp.

The second rule states that the ⟨imply list⟩ consists of one or more items of the form:

```
⟨field name⟩ = ⟨expression⟩
```

Here, the field name identifies either an intrinsic or user-defined field other than the field for which the microp is defined.

For example, the statement:

```
LOADB IMPLY GATE = 0 CCTL = 1100B;
```

associates with the microp LOADB a default of 12 (1100B) for the field CCTL. Whenever the microp LOADB appears in a specification statement to specify the field for which it is defined, these default values will be assembled into the fields GATE and CCTL unless these fields are explicitly specified.

2.12 KBUS STATEMENT

The KBUS statement identifies the user-defined field that is to be treated as the K-bus field.

SYNTAX

```
⟨k-bus statement⟩ :: = ⟨field name⟩ KBUS
```

The field name must be defined in a FIELD statement somewhere else in the program. In other words, a KBUS statement does not create a new field; it merely ascribes a "K-bus attribute" to a field whose existance is established in a FIELD statement. Only one KBUS statement may appear in a program.

K-BUS DEFAULTS

The microps that are associated with the intrinsic CPE field all imply either an all-zero or an all-one default bit pattern for the K-bus field, since the K-bus inputs to the CPE array participate in every function that the CPE array performs. The KBUS statement identifies the user-defined field that is to receive CPE microp K-bus defaults. K-bus defaults may, of course, always be overridden in a specification statement by an explicit specification of the K-bus field or by an implication from another microp.

EXAMPLE

The two statements:

```
KB FIELD LENGTH = 16 MICROPS (HI.BYTE = 0FF00H LO.BYTE = 0FFH);
KB KBUS;
```

define the field KB and identify KB as the K-bus field, respectively. Note that the FIELD statement does not declare a default for the KB field, since that default would always be overridden by a CPE microp K-bus default. The FIELD statement does, however, declare two microps, which may be used in specification statements, when required to override the K-bus defaults that the CPE microps provide.

2.13 ADDRESS STATEMENT

The ADDRESS statement identifies the user-defined field that is to be treated as the microprogram memory address extension field.

SYNTAX

⟨address statement⟩ :: = ⟨field name⟩ ADDRESS

The field name must be defined in a FIELD statement elsewhere in the program.

ADDRESS EXTENSION FIELD

The MCU developes a 9-bit microprogram memory address, which provides a basic addressing capability of 512 words. One method of addressing a microprogram memory larger than 512 words is to allocate an address extension field to the microinstruction representation, which may serve as a memory "plane select" field. The ADDRESS statement identifies such a field to XMAS.

XMAS HANDLES ADDRESS EXTENSION FIELD

XMAS assembles the high-order address bits needed to address the desired microprogram memory word into the field identified in the ADDRESS statement. The microassembler obtains the information to generate the extra address bits from the integer address, which appears in every specification statement. Consequently, the programmer need not explicitly specify the contents of the address extension field.

EXAMPLE

The two statements:

```
XADR  FIELD  LENGTH = 2;
XADR  ADDRESS;
```

define the field XADR and identify it as an address extension field, respectively.

2.14 SPECIFICATION STATEMENT

SYNTAX

A specification statement specifies, either explicitly or implicitly, the bit patterns for each field of a single microinstruction word and assigns that word to a microprogram memory location. Optionally, a specification statement may define one or more address identifiers.

⟨specification statement⟩ :: = ⟨label part⟩ {⟨fields part⟩} . . .

⟨label part⟩ :: = [*] ⟨integer⟩: [⟨address identifier⟩:] . . .

⟨address identifier⟩ :: = ⟨identifier⟩

$$\langle\text{fields part}\rangle ::= \left\{\begin{array}{l}\langle\text{field name}\rangle = \langle\text{expression}\rangle \\ \langle\text{microp}\rangle \\ \langle\text{CPE microp}\rangle \quad (\langle\text{register name}\rangle) \\ \langle\text{JUMP microp}\rangle \; (\underset{*}{\{}\langle\text{expression}\rangle\} \ldots)\end{array}\right\} \ldots$$

The first rule states that a specification statement consists of a ⟨label part⟩ followed by one or more ⟨fields part⟩s.

MICROPROGRAM MEMORY ADDRESS ASSIGNMENT

The ⟨label part⟩ must include a construct of the form:

⟨integer⟩:

The value represented by the integer is the microprogram memory address assignment for the microinstruction word specified by the statement. All integer representations are permitted (Section 2.3), but the integer value must be between 0 and n-1, where n is the number of words in the microprogram memory. Furthermore, a given integer may appear in the ⟨label part⟩ of only one specification statement. The significance of the optional asterisk (*), which may appear before the integer in the ⟨label part⟩, is discussed later in this section.

ADDRESS IDENTIFIER

An address identifier is defined when it appears in the ⟨label part⟩ of a specification statement in the form:

⟨address identifier⟩:

The ⟨label part⟩ may include any number of address identifiers, but none is required. Each address identifier takes on the value of the microprogram memory address assignment for the statement (i.e., the value of the integer in the ⟨label part⟩). An address identifier represents symbolically the microprogram memory location of the microinstruction word specified by the statement; it is, therefore, useful as an operand of a JUMP microp to indicate microprogram sequencing. In general, an address identifier may be employed in any specification statement where an expression is allowed.

FIELD SPECIFICATION FORMS

Each ⟨fields part⟩ of a specification statement may take one of four forms, as shown in the third rule above. In the form:

KEYWORD SPECIFICATION

⟨field name⟩ = ⟨expression⟩

The field name identifies the field that is to receive the bit pattern supplied by the value of the expression truncated to the length of the field. This form of field specification is called a keyword specification. A keyword specification is an explicit specification for one field; it carries no implications for any other fields in the microinstruction word.

FI, FO OR USER-DEFINED MICROP

Another form that a ⟨fields part⟩ may take is simply:

⟨microp⟩

Here, the microp is either an intrinsic or user-defined microp (other than a CPE microp or JUMP microp). The microp carries an explicit bit pattern assignment for the field for which it is defined. If the microp has appeared in an IMPLY statement, the microp carries default bit pattern assignments for one or more other fields. Default hierarchy is discussed later in this section.

CPE MICROP

The ⟨fields part⟩ may take the form:

⟨CPE microp⟩ (⟨register name⟩)

The CPE microps and register names are listed in Table 2-2. This ⟨fields part⟩ form is an explicit assignment for the CPE field. All CPE microps carry a default pattern for the K-bus field.

JUMP MICROP

The final form that the ⟨fields part⟩ may take is:

$\langle\text{JUMP microp}\rangle(\left\{ \begin{matrix} \langle\text{expression}\rangle \\ * \end{matrix} \right\} \ldots)$

The JUMP microps are listed in Table 2-2. The parentheses enclose one or more operands. Each JUMP microp symbolizes one of the MCU's conditional or unconditional Jump functions. Unconditional JUMP microps require one operand; conditional JUMP microps require two, four or sixteen operands depending upon the microp. Each operand represents one of the microprogram memory addresses that may be the target of the Jump function. An operand may appear as any valid expression; typically the expression consists of simply an address identifier, which is defined in the ⟨label part⟩ of another specification statement. An asterisk (*) may appear in place of an expression only in the context of a conditional Jump function; its meaning in this context is discussed below.

The following specifications statement includes examples of each of the four ⟨fields part⟩ forms:

EXAMPLE

```
41H:L1:L2:  CONT=15  READ  ADR(R7)  JZR(FETCH);
```

This specification statement specifies the microinstruction word at location 41_{16} of the microprogram memory. The statement defines the address identifiers L1 and L2, either of which may be used in other specification statements to reference this microinstruction word. CONT=15 is a keyword specification for the user-defined field, CONT, which must have been defined in a FIELD statement. READ is a microp associated with some user-defined field; READ must have been declared in the FIELD statement that defined the field with which READ is associated. READ carries an explicit bit pattern assignment for its field. If READ has appeared in an IMPLY statement, READ carries a default bit pattern assignment for one or more other fields (e.g., FI and/or FO). XMAS will supply a default bit pattern assignment for the FI and/or FO fields if these fields are not otherwise specified. ADR(R7) makes an explicit bit pattern assignment for the CPE field. ADR (as do all CPE microps) carries a default bit pattern assignment for the K-bus field. If a user-defined field name has appeared in a KBUS statement, then this field will receive the K-bus default assignment. If a K-bus field has not been identified, the CPE K-bus default is not used. JZR (FETCH) is an explicit bit pattern assignment for the JUMP field. Here, the identifier FETCH would be normally an address identifier (i.e., it appears in the ⟨label part⟩ of another specification statement), but it could be a value identifier (defined in a VALUE statement).

XMAS ADDRESS CHECKING

XMAS handles the JUMP field in a special way. Each JUMP microp represents an MCU Jump function. When a JUMP microp appears in a specification statement, XMAS determines the subset of microprogram memory locations that the microinstruction word being specified is capable of referencing from its location in the microprogram memory using the indicated JUMP function. The operand or operands of the JUMP microp must represent locations in this subset, or XMAS will output an error message to the list file. The JUMP microp JZR requires its target address to be in row zero of the microprogram memory. Consequently, in the previous example, FETCH must represent a value in the range 0 to 15 if the microinstruction word that FETCH defines is to be reached using the JZR Jump function. For a conditional Jump function such as:

```
JRL (L1 L2 L3 L4)
```

each operand must represent a target address that is reachable, and the operands must be listed in their ascending order of value.

USE OF THE OPTIONAL ASTERISK

If the programmer knows that a conditional jump will never select a particular target, say L3 (i.e., the corresponding condition will never occur), an asterisk (*) may be substituted for the operand:

```
JRL (L1 L2 * L4)
```

The asterisk relieves the programmer of the responsibility of supplying a valid operand. If an asterisk appears in the ⟨label part⟩ of a specification statement, XMAS will not perform address checking for that statement. For example, in the statement:

```
*41:L1:  CCTL=15  LOADA  ADR(R7)  JZR(FETCH);
```

an error will not be reported if FETCH is out of range. Typically address checking is overridden in this fashion when the programmer knows that address bits from an external source will be forced onto the microprogram memory address bus (e.g., an interrupt).

XMAS GENERATED ADDRESSING

If an explicit specification for the JUMP field does not appear in a specification statement, XMAS will attempt to supply an unconditional jump code that will allow the microinstruction currently being specified to access the microinstruction specified by the next specification statement in the source file. If XMAS is unable to supply an appropriate jump code (i.e., the next specification statement is for a microprogram memory location that is out of range), XMAS will output an error message to the list file.

DEFAULTS

Every specification statement must provide either an explicit or implicit assignment for every field in the microinstruction word, except the JUMP field as discussed above. The microinstruction word includes the four intrinsic fields (CPE, JUMP, FI, and FO), as well as all fields the user has established in FIELD statements. The ways in which a given field can be specified, in order of decreasing effectiveness, are:

DEFAULT HIERARCHY

1. An explicit keyword or microp assignment;
2. A default assignment from a microp bound to another field, as declared in an IMPLY statement;
3. In the unique case of the K-bus field, a default assignment from a CPE microp;
4. The field's default assignment, as declared in the FIELD statement or supplied by XMAS for the FI and FO fields.

This means, for example, that an explicit assignment for a given field always overrides any default assignment for that field.

SECTION 3
XMAS LISTING OUTPUT

In assembling an XMAS language program, XMAS is capable of generating the following reports:

- Listing of XMAS language source statements and assembled bit patterns;
- Cross reference directory;
- Graphic representation of the microprogram memory image.

The user selects the information he wants via the control language (Section 9).

LIST FILE

XMAS outputs the selected information to the FORTRAN data file that has been designated as the list file. The list file is page oriented; running error and page counts are given at the top of each page. Via the control language, the user can specify an optional page title, the number of lines per page, the number of characters per line, and the form feed mode to be used between pages. The list file width must be a minimum of seventy-two characters.

Subsequent subsections describe the XMAS reports.

3.1 XMAS SOURCE STATEMENTS AND BIT PATTERNS

The user may choose to have both his XMAS language statements and the bit patterns produced from the assembly of XMAS specification statements included in the list file. A partial listing is shown in Figure 3-1.This sample is intended to illustrate output format only; it does not necessarily represent a meaningful XMAS language program.

SOURCE RECORDS

Each record from the source file is written to the list file left justified in column 8. The records are copied to the list file exactly as they appear in the source file unless the line width of the list file is too small to accommodate the entire record. In this case, the remaining characters of the source record are written in the next line of the list file, again left-justified in column 8.

RECORD NUMBERS

Each record in the source file, beginning with the first XMAS language statement, is assigned a sequential record number. This record number appears as a decimal integer in the list file in columns 1 through 5 of the line in which the corresponding record appears. Record numbers run from 1 through 32767_{10}; if more source records exist, the numbering simply starts again at 1.

BIT PATTERNS

A bit pattern produced by an XMAS specification statement is displayed in the list file on the line immediately following the specification statement. The line is left justified in column 8 and contains the hexadecimal representation of the address of the microinstruction word (in parentheses) followed by the binary bit pattern broken out into the fields of the microinstruction word.

FIELD NAME ORDERING

The CPE field is given first, followed by the FI field, the FO field, the JUMP field, and finally the user-defined fields in the order in which they are defined. The field names appear at the top of each page below the page header. The bit positions are numbered within each field with the rightmost bit of a field numbered zero. If a bit pattern is too long for the line width of the list file, it is broken between fields and continued on the next line. Continuation lines are left justified to begin directly below the beginning of the CPE field. Two blank lines are written between a line containing a bit pattern and the subsequent line containing an XMAS specification statement.

SELECTING OPTIONS

It is possible to select only the source file records for display in the list file. In this case, the XMAS language statements are written to the list file as described above. Field names and bit position numbers do not appear at the top of each page.

It is also possible to select only the bit patterns for display in the list file. In this case, the bit patterns are displayed as described above with the addition that the record number of the beginning of the corresponding XMAS specification statement appears in columns 1 through 5.

```
XMAS VERS 1.Ø THIS IS A SAMPLE OF XMAS OUTPUT      ERRORS=   Ø PAGE  1Ø

RECORD             CPE    FI  FQ    JUMP      MASK     MEM
NUMBER           654321Ø  1Ø  1Ø  654321Ø  876543210  21Ø

  1Ø8  /* DEFINE KBUS FIELD TO HAVE LENGTH 9 */
  1Ø9  MASK  FIELD  LENGTH=9  DEFAULT=Ø;
  11Ø  MASK  KBUS;
  111  MEM   FIELD  LENGTH=3  DEFAULT=Ø;
  112  14Ø: M12:   ILR(R2) FI=ØB FO=ØB MASK=ØØØH JMP(GST);
       (ØØ8CH)  ØØØØØ1Ø  ØØ  ØØ  Ø1ØØ11Ø  ØØØØØØØØØ  ØØØ

  113  141: M13:   ILR(R2) FI=ØB FO=ØB MASK=ØØØH JMP(GST);
       (ØØ8DH)  ØØØØØ1Ø  ØØ  ØØ  Ø1ØØ11Ø  ØØØØØØØØØ  ØØØ

  114  142: M14:   ILR(R2) FI=0B FO=0B MASK=000H JMP(GST);
       (ØØ8EH)  ØØØØØ1Ø  ØØ  ØØ  Ø1ØØ11Ø  ØØØØØØØØØ  ØØØ

  115  6:   GST:   ADR(R9) MASK=1FFH  JLL(WD1,WI1,BD1,BI1);
       (ØØØ6H)  Ø111ØØ1  ØØ  11  11Ø1111  111111111  ØØØ

  116  18:    SDR(RØ) MASK=1FFH;  17:   SDR(R1) MASK=1FFH;
       (ØØ12H)  Ø1ØØØØØ  11  11  Ø11ØØØ1  111111111  ØØØ
       (ØØ11H)  Ø1ØØØØ1  11  11  ØØ11111  111111111  ØØØ

  117  497:   SDR(R2) MASK=1FFH;  5ØØ:  SDR(R3) MASK=1FFH;
       (Ø1F1H)  Ø1ØØØ1Ø  11  11  Ø11Ø1ØØ 111111111  ØØØ
       (Ø1F4H)  Ø1ØØØ11  11  11  Ø11Ø1Ø1 111111111  ØØØ

  118  175:   SDR(R9) MASK=1FFH  JPX(MO,M1,M2,M3,M4,
  119                 M5,M6,M7,M8,M9,M1Ø,M11,M12,M13,
  12Ø                 M14,M15);
       (ØØAFH)  Ø1Ø1ØØ1  11  11  1111ØØØ  111111111  ØØØ

  121  15Ø: L1:  ILR(R2) FI=ØB FO=ØB MASK=ØØØH  JMP(GST); /* ENTRY
       (ØØ96H)  ØØØØØ1Ø  ØØ  ØØ  Ø1ØØ11Ø  ØØØØØØØØØ  ØØØ

  122  FOR GLOBAL VARIABLES */  151:  L2: ILR(R2) FI=ØB FO=ØB
  123  MASK=ØØØH  /* ALTERNATE ENTRY FOR GLOBAL VARIABLES */
  124  JMP(GST);  152:  L3: ILR(R2)  FI=ØB FO=ØB  MASK=ØØØH
       (ØØ97H)  ØØØØØ1Ø  ØØ  ØØ  Ø1ØØ11Ø  ØØØØØØØØØ  ØØØ

  125  JMP(GST);
       (ØØ98H)  ØØØØØ1Ø  ØØ  ØØ  Ø1ØØ11Ø  ØØØØØØØØØ  ØØØ
```

Fig. 3-1. Example of XMAS Source Statements and — Bit Patterns

ERROR MESSAGES

Any error messages will be output to the list file by XMAS. A summary of the XMAS error messages is listed in Appendix G.

3.2 CROSS REFERENCE DIRECTORY

The cross reference directory contains an alphabetical list of all XMAS specification statement labels and, for each label, a list of record numbers in which the label is referenced. The record number corresponding to the record in which the label is defined is enclosed in parentheses. Figure 3-2 illustrates the format of the cross reference directory.

EXAMPLE

```
CROSS REFERENCE DIRECTORY
LABEL                                        REFERENCES
GST                                          112,113,114,(115),121,124,125
L1                                           (121)
L2                                           (122)
L3                                           (124)
M12                                          (112),119
M13                                          (113),119
M14                                          (114),120
```

Figure 3-2. Example of Cross Reference Directory

3.3 MICROPROGRAM MEMORY IMAGE

The microprogram memory image is a graphic representation of the contents of the microprogram memory produced by an XMAS assembly. Jump functions and their targets are emphasized in the image, since the principle purpose of the image is to ease the job of relocating microinstructions. The image consists of an array of cells, each of which represents one microprogram memory location and the microinstruction word it contains. The image has sixteen columns and as many rows as are necessary to encompass the microprogram address space. Figure 3-3 shows a sample of an image produced for a list file with sixty lines per page and a line width of 132 characters.

CELL FORMAT

A typical cell of the array has the form:

```
 - - - - - - -
 *    s1     *
 *    n1     *
 *           *
 *    n2     *
 *    n3     *
 - - -  - - - -
```

Fig. 3-2. Example of Cross Reference Directory

where:

s1	is the JUMP microp for this cell.
n1	is the hexadecimal target address of this cell; if the jump function is conditional, the value is the first closest address in the set of contiguous addresses reachable by the jump function.
n2	is the record number of the specification statement that specifies this cell.
n3	is the number of cells that have this cell as their target.

LIST FILE CONSTRAINS FORMAT

The actual format of the image depends on the line width of the list file. If the line width is 132 characters, sixteen columns of the image will appear across the page. If the line width is less than 132 characters, the image will be printed in two halves with the first eight columns of all rows printed first followed by the last eight columns of all rows. In either case, as many complete rows of image as possible will be printed on each page.

```
XMAS VERS 1.0 THIS IS A SAMPLE OF XMAS OUTPUT     ERRORS=   0 PAGE  10

MICROPROGRAM MEMORY IMAGE

       0H      1H      2H      3H      4H      5H      6H      7H      8H      9H      AH      BH      CH      DH      EH      FH
   =================================================================================================================================
   '   JCR '   JCR *   JCP *   JCC *       '   JPR '   JPX '   JCC =   JCR '   JCR *   JCR *   JCC *   JPR '   JPX '   JCC '   JCC '
   ' 0001H ' 0002H * 0005H * 0013H *       ' 0020H ' 0050H ' 0010H = 0009H ' 000AH * 000BH * 001BH * 0010H ' 0040H ' 007BH ' 005FH '
000H       '       *       *       *       '       '       '       =       '       *       *       *       '       '       '       '
   '   423 '   254 *   524 *   229 *       '   429 '   113 '   119 =   221 '    90 *   421 *    79 *    36 '    84 '   224 '   524 '
   '     5 '     7 *     3 *     8 *       '     9 '     4 '    11 =    10 '     4 *     3 *     2 *     6 '     6 '    13 '     1 '
   ----------------*-------*-------*-------------------------------=---------------*-------*-------*--------------------------------
   '   JCC '   JCC *   JPR *   JPX *   JCC '   JCR '   JCR '   JCC =   JCC '       *   JLL *   JPX *   JCR '   JCR '   JPR '   JCC '
   ' 0040H ' 0051H * 0030H * 0060H * 0034H ' 0016H ' 0017H ' 0077H = 0068H '       * 0014H * 0050H * 001DH ' 001EH ' 0030H ' 007FH '
001H       '       *       *       *       '       '       '       =       '       *       *       *       '       '       '       '
   '   534 '   438 *   238 *   234 *    12 '    56 '   153 '   523 =   642 '       *    73 *   276 *   375 '   224 '   477 '   123 '
   '    11 '     2 *     1 *    12 *     5 '     8 '     1 '     1 =     2 '       *     7 *     5 *     3 '     7 '     4 '     9 '
   ----------------*-------*-------*-------------------------------=---------------*-------*-------*--------------------------------
   '   JCR '   JCR *   JCR *   JZR *       '   JPR '   JPX '   JZR =   JCR '   JCR *   JCR *   JZP *   JPR '   JPX '   JCC '   JCC '
   ' 0021H ' 0022H * 0023H * 0008H *       ' 0020H ' 0050H ' 000AH = 0029H ' 002AH * 002BH * 0008H * 0010H ' 0040H ' 007BH ' 005FH '
002H       '       *       *       *       '       '       '       =       '       *       *       *       '       '       '       '
   '   315 '   364 *   386 *   444 *       '   333 '   299 '   398 =   479 '   400 *    89 *   397 *   285 '   288 '   742 '   712 '
   '     5 '     7 *     3 *     8 *       '     9 '     4 '    11 =    10 '     4 *     3 *     2 *     6 '     6 '    13 '     1 '
   ----------------*-------*-------*-------------------------------=---------------*-------*-------*--------------------------------
   '   JCC '   JCC *   JPB *   JPX *   JCC '   JCR '   JCR '   JCC =   JCC '       *   JLL *   JPX *   JCR '   JCR '   JPR '   JCC '
   ' 0040H ' 0051H * 0030H * 0060H * 0084H ' 0036H ' 003AH ' 0067H = 0068H '       * 0014H * 0050H * 003DH ' 003EH ' 0030H ' 006FH '
003H       '       *       *       *       '       '       '       =       '       *       *       *       '       '       '       '
   '    51 '    72 *    86 *    19 *   634 '   555 '   662 '   853 =   253 '       *   228 *   534 *    49 '   496 '   365 '   387 '
   '    11 '     2 *     1 *    12 *     5 '     8 '     1 '     1 =     2 '       *     7 *     5 *     3 '     7 '     4 '     9 '
   ----------------*-------*-------*-------------------------------=---------------*-------*-------*--------------------------------
   '   JFL '   JCR *   JCC *   JCC *   JPR '       '   JCC '   JCR =   JCC '   JCR *   JCR *   JPX *   JPR '   JCC '   JCC '   JCR '
   ' 0072H ' 0042H * 0062H * 0093H * 0090H '       ' 0066H ' 0048H = 0098H ' 004AH * 004BH * 0090H * 00A0H ' 007DH ' 009EH ' 004CH '
004H       '       *       *       *       '       '       '       =       '       *       *       *       '       '       '       '
   '   385 '   245 *   222 *   154 *    13 '       '    95 '   123 =   352 '   553 *   624 *    35 *   553 '   253 '   354 '   143 '
   '     1 '     6 *    10 *     1 *    42 '       '     1 '     4 =    13 '     5 *     7 *     3 *     3 '     7 '     3 '     5 '
   ----------------*-------*-------*-------------------------------=---------------*-------*-------*--------------------------------
   '   JCR '   JCR *   JCR *   JZR *       '   JPR '   JPX '   JZR =   JCR '   JCR *   JCR *   JZR *   JPR '   JPX '   JCC '   JCC '
   ' 0051H ' 0052H * 0053H * 0008H *       ' 0020H ' 0050H ' 000AH = 0059H ' 005AH * 0058H * 0008H * 0010H ' 0040H ' 007FH ' 006FH '
005H       '       *       *       *       '       '       '       =       '       *       *       *       '       '       '       '
   '    89 '    75 *   286 *   639 *       '   519 '   416 '   429 =   883 '   777 *   558 *   666 *    55 '    88 '   551 '   441 '
   '     5 '     7 *     3 *     8 *       '     9 '     4 '    11 =    10 '     4 *     3 *     2 *     6 '     6 '    13 '     1 '
   ----------------*-------*-------*-------------------------------=---------------*-------*-------*--------------------------------
   '   JCC '   JCC *   JPR *   JPX *   JCC '   JCR '   JCR '   JCC =   JCC '       *   JLL *   JPX *   JCR '   JCR '   JPR '   JCC '
   ' 0040H ' 0051H * 0030H * 0060H * 0084H ' 0066H ' 0067H ' 0087H = 0068H '       * 0014H * 0050H * 006DH ' 006FH ' 0030H ' 008FH '
006H       '       *       *       *       '       '       '       =       '       *       *       *       '       '       '       '
   '    91 '    92 *    93 *    53 *    54 '    62 '    63 '    41 =    39 '       *   912 *   915 *   925 '   936 '   947 '   924 '
   '    11 '     2 *     1 *    12 *     5 '     8 '     1 '     1 =     2 '       *     7 *     5 *     3 '     7 '     4 '     9 '
   ----------------*-------*-------*-------------------------------=---------------*-------*-------*--------------------------------
   '   JFL '   JCR *   JCC *   JCC *   JPR '       '   JCC '   JCR =   JCC '   JCR *   JCR *   JPX *   JPR '   JCC '   JCC '   JCR '
   ' 0072H ' 0072H * 0062H * 0093H * 0090H '       ' 0066H ' 0078H = 0098H ' 007AH * 007BH * 0090H * 00A0H ' 006DH ' 009BH ' 0074H '
007H       '       *       *       *       '       '       '       =       '       *       *       *       '       '       '       '
   '   987 '   976 *   946 *   937 *   926 '       '   527 '   835 =   779 '   998 *   246 *   486 *    36 '   399 '   377 '   244 '
   '     1 '     6 *    10 *     1 *    42 '       '     1 '     4 =    13 '     5 *     7 *     3 *     3 '     7 '     3 '     5 '
   =================================================================================================================================
```

Fig. 3-3. Example Microprogram Memory Image

USE OF SPECIAL CHARACTERS

Groups of four or eight rows, groups of eight columns, and columns two, three, ten and eleven have special significance in the MCU address scheme. For this reason, these rows and columns are set off by the use of special characters. In addition, each page has row and column labels that specify the hexadecimal row and column addresses. The microprogram memory address of any cell in the image is determined by concatenating its row and column address.

SECTION 4
FUNCTIONAL DESCRIPTION OF THE MCU

The 3001 Microprogram Control Unit (MCU) has three functional responsibilities in a typical Series 3000 configuration:

1. Provides sequencing for the microprogram;
2. Supplies the carry/shift input to the CPE array.
3. Handles the carry/shift output from the CPE array.

The organization of the MCU is illustrated in Figure 4-1.

JUMP FUNCTION LOGIC

The MCU's jump function logic determines the sequence in which microinstruction words are accessed from the microprogram memory by supplying a 9-bit microprogram memory address during each microinstruction cycle. The current microprogram memory address is held in the Microprogram Address Register (MAR). The Next Address Logic, under control of the Jump Function Bus, formulates the address that is clocked into the MAR at the end of the current microinstruction cycle.

The encoded information on the Jump Function Bus designates the jump function and provides part of the next address. The jump function determines the manner in which the Next Address Logic formulates the next address. In all, the MCU supports eleven unconditional and conditional jump functions, which are described in Section 4.1.

FLAG OUTPUT LOGIC

In a typical configuration, the MCU's Flag Output (FO) is connected to the Carry Input (CI) and the Left Input (LI) of the CPE array. Thus, the Flag Output furnishes the carry/shift input for CPE array functions. Under control of the Flag Output Function Bus, the MCU can force the Flag Output to zero, one, the current state of the C-Flag or the current state of the Z-Flag. Flag output functions are described in Section 4.2.

FLAG INPUT LOGIC

In a typical configuration, the MCU's Flag Input (FI) is connected to the Carry Output (CO) and the Right Output (RO) of the CPE array. Thus, the Flag Input receives the carry/shift output of the CPE array. During each microinstruction cycle, the state of the Flag Input is automatically saved in the MCU's F-latch. Under control of the Flag Input Function Bus, the MCU can also save the state of the Flag input in the C-flag, the Z-flag, both flags or neither flag. Flag input functions are described in Section 4.3.

MICROINSTRUCTION WORD FIELDS

In a typical configuration, three fields in the microinstruction word control the MCU's functions, as illustrated in Figure 4-2. The JUMP field drives the Jump Function Bus; consequently, the JUMP field designates the jump function to be performed by the MCU. The FO and FI fields drive the Flag Output Function Bus and the Flag Input Function Bus, respectively; consequently the FO field determines the carry/shift input to the CPE array, and the FI field controls the handling of the carry/shift output from the CPE array.

4.1 JUMP FUNCTIONS AND JUMP MICROPS

There are eleven jump functions that the MCU can perform. Four of these jump functions are unconditional and seven are conditional.

UNCONDITIONAL JUMP FUNCTIONS

An unconditional jump function specifies a jump to a single target address. The MCU's jump function logic formulates the target address on the basis of information in the JUMP field (i.e., information on Jump Function Bus) and the location of the current microinstruction (i.e., the data in the MCU's Microprogram Address Register).

CONDITIONAL JUMP FUNCTIONS

A conditional jump function specifies a jump to one of a group of locations depending upon the data in or on the latch or bus being tested by the jump function. The number of possible target addresses for a conditional jump function depends on the number of states that the condition being tested can take. The MCU's jump function logic formulates the target address

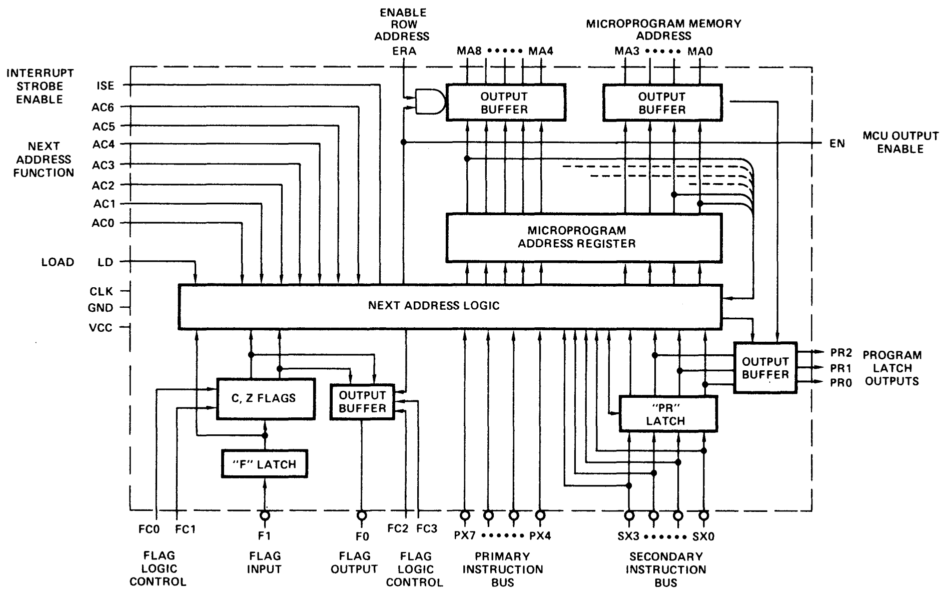


Fig. 4-1. MCU Functional Organization

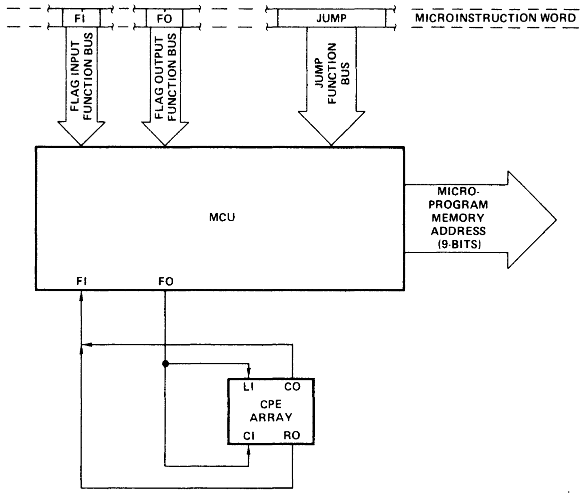


Fig. 4-2. Relationship Between MCU and Microinstruction Word

based on information in the JUMP field, the location of the current microinstruction and the current state of the data being tested.

MICROPROGRAM MEMORY ORGANIZATION

To understand the jump functions, it is helpful to visualize the microprogram memory as a two-dimensional matrix of 32 rows and 16 columns. Thus, a location in the matrix is identified by a row address and a column address. The high order five bits of the MCU's Microprogram Address Register address the row (row_0–row_{31}), and the low order four bits address the column (col_0–col_{15}).

JUMP SET

JUMP SET DIAGRAMS

From a given location in the matrix (i.e., a given row and column) only a subset of other locations in the matrix can be reached using a given jump function. Figure 4-3 illustrates the subset of locations that can be reached by each of the eleven jump functions from row_{21}, col_5 of the matrix. A similar set of diagrams could be produced for any other location in the matrix. The XMAS language includes a set of 12 microps for specifying jump functions in specification statements. These microps are described below and are summarized in Table 4-1.

JZR — **Jump to Row Zero**

FORMAT

JZR (L)

DESCRIPTION

JZR specifies an unconditional jump to one of the sixteen column positions of row zero. L must identify a microinstruction in row zero. Because the current microinstruction's position is not used in formulating the next address for the JZR jump function, may be used to address any location in row zero from anywhere in the microprogram memory. JZR (15) performs the special function of activating the MCU's Interrupt Strobe Enable line (ISE), which may be used to implement a microprogram interrupt facility. NOTE: If extended memory is being used, JZR will cause a jump to row zero in the plane indicated by the address extension field (refer to Section 1.1).

JCR — **Jump in Current Row**

FORMAT

JCR (L)

DESCRIPTION

JCR specifies an unconditional jump to one of the sixteen column positions of the current row. L must identify a microinstruction in the current row.

JCC — **Jump in Current Column**

FORMAT

JCC (L)

DESCRIPTION

JCC specifies an unconditional jump to one of the thirty-two row positions of the current column. L must identify a microinstruction in the current column.

JMP — **General Unconditional Jump**

FORMAT

JMP (L)

DESCRIPTION

JMP may be used in place of JCC, JCR and JZR to specify an unconditional jump either in the current column, the current row or row zero. L must identify a microinstruction in the current column, the current row or row zero. (To activate ISE, JZR (15) should be used instead of JMP (15)).

JCE — **Jump in Current Column/Row Group and Enable PR – Latch Outputs**

FORMAT

JCE (L)

DESCRIPTION

JCE specifies an unconditional jump to one of eight positions in the current row group of the current column. L must identify a location in the current row group: row0-row7, row8-row15, row16-row23, or row24-row31. L must identify a location in the current column. JPX also enables the output buffer for the three lowest order bits of the PR-latch.

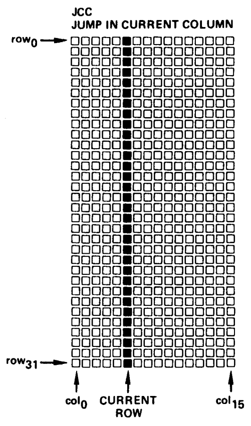


Fig. 4-3a

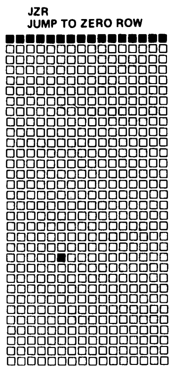


Fig. 4-3b

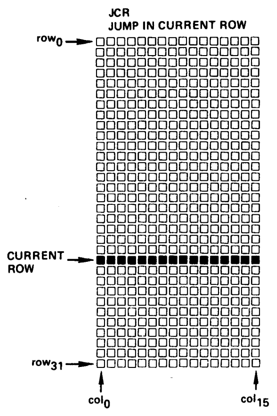


Fig. 4-3c

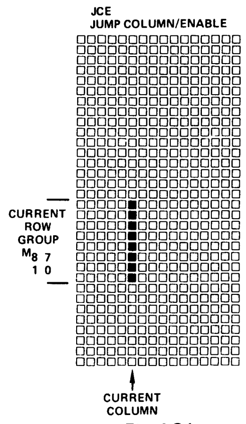


Fig. 4-3d

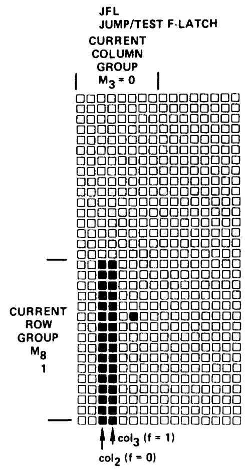


Fig. 4-3e

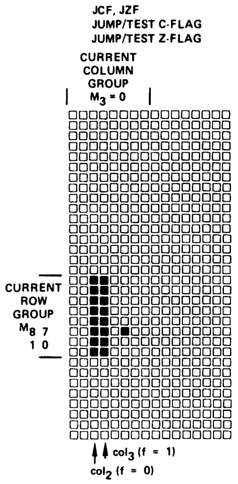


Fig. 4-3f

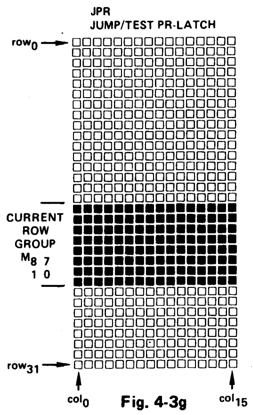


Fig. 4-3g

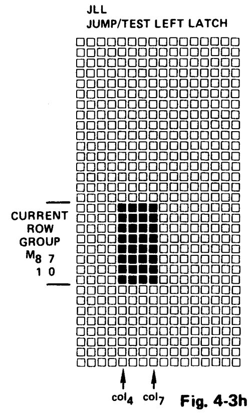


Fig. 4-3h

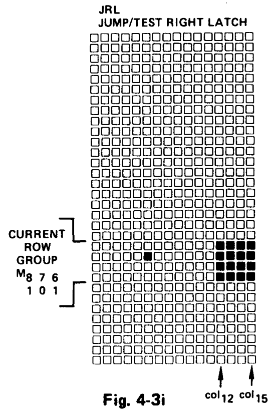


Fig. 4-3i

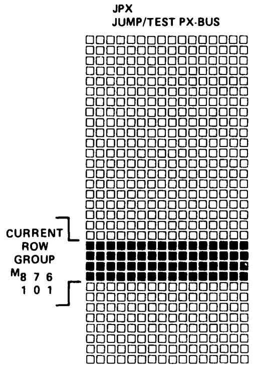


Fig. 4-3j

Fig. 4-3. Jump Set Diagrams

TABLE 4-1. SUMMARY OF MCU JUMP FUNCTIONS

MNEMONIC	JUMP FUNCTION
JZR	Jump to row zero
JCC	Jump in current column
JCR	Jump in current row
JCE	Jump in current column/row group and enable PR-latch outputs
JFL	Jump/test F-latch
JCF	Jump/test C-flag
JZF	Jump/test Z-flag
JPX	Jump/test PX-bus and load PR-latch
JPR	Jump/test PR-latch
JLL	Jump/test two leftmost PR-latch bits
JRL	Jump/test two rightmost PR-latch bits

TABLE 4-2. SUMMARY OF MCU FLAG OUTPUT FUNCTIONS

MNEMONIC	FLAG OUTPUT FUNCTION
FF0	Force FO to 0
FFC	Force FO to C-flag
FFZ	Force FO to Z-flag
FF1	Force FO to 1

TABLE 4-3. SUMMARY OF MCU FLAG INPUT FUNCTIONS

MNEMONIC	FLAG INPUT FUNCTION
SCZ	Set C and Z Flags to F
STZ	Set Z-flag to F
STC	Set C-flag to F
HCZ	Hold C-flag and Z-flag

JFL **Jump/Test F-Latch**

FORMAT JFL (L0 L1)

DESCRIPTION JFL specifies a test on the F-latch. L0 must identify a location in the current row group: row0-row15 or row16-row31. L0 must identify a location in col_2 if the current column is col_0-col_7 or in col_{10} if the current column is col_8-col_{15}. L0 and L1 must identify sequential column locations (i.e., columns 2 and 3 or 10 and 11). L0 is selected if the F-latch contains 0, and L1 is selected if the F-latch contains 1. See Figure 4-3e. NOTE: The test on the F-latch is performed *after* the results of the current microinstruction being executed (on the FI line) are reflected in the state of the F-latch.

JCF **Jump/Test C-Flag**

FORMAT JCF (LO L1)

DESCRIPTION JCF specifies a test on the C-flag. LO must identify a location in the current row group: row_0-row_7, row_8-row_{15}, row_{16}-row_{23} or row_{24}-row_{31}. LO must identify a location in col_2 if the current column is col_0-col_7 or in col_{10} if the current column is col_8-col_{15}. LO and L1 must identify sequential column locations. LO is selected if the C-flag contains 0, and L1 is selected if the C-flag contains 1. See Figure 4-3f. NOTE: The test on the C-Flag is performed *before* a flag input microp in the current microinstruction being executed could change the state of the C-Flag. Consequently JCF tests the results of a previous microinstruction, not the current one.

JZF **Jump/Test Z-Flag**

FORMAT JZF (10 L1)

DESCRIPTION JZF specifies a test on the Z-flag. JZF is identical to JCF except the Z-flag, rather than the C-flag, is used in selecting the target address.

JPX **Jump/Test PX-Bus and Enable PR-Latch**

FORMAT JPX (LO L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15)

DESCRIPTION JPX specifies a test on the data on the PX-bus. LO must identify a location in the current row group: row_0-row_3, row_4-row_7, row_8-row_{11}, row_{12}-row_{15}, row_{16}-row_{19}, row_{20}-row_{23}, row_{24}-row_{27}, row_{28}-row_{31}. LO must identify a location in col0. LO through L15 must identify sequential column locations. LO is selected if the data on the PX-bus is 0000_2, L1 is selected if the data on the PX-bus is 0001_2 . . . and L15 is selected if the data on the PX-bus is 1111_2. JPX also saves the data on the SX-bus in the PR-latch. See Figure 4-3j.

JPR **Jump/Test PR-Latch**

FORMAT JPR (LO . . . L15)

DESCRIPTION JPR specifies a test on the PR-latch. LO must identify a location in the current row group: row -row , row -row , row_{16}-row_{23} or row_{24}-row_{31}. LO must identify a location in col0. LO through L15 must identify sequential column locations. LO is selected if the PR-latch contains 0000_2, L1 is selected if the PR-latch contains 0001_2, . . . and L15 is selected if the PR-latch contains 1111_2. See Figure 4-3g.

JRL **Jump/Test Rightmost PR-Latch Bits**

FORMAT JRL (LO L1 L2 L3)

DESCRIPTION JRL specifies a test on the right two bits of the PR-latch. LO must identify a location in row_4-row_7 if the current row is row_0-row_7, in row_{12}-row_{15} if the current row is row_8-row_{15}, in row_{20}-row_{23} if the current row is row_{16}-row_{23} or in row_{28}-row_{31} if the current row is row_{24}-row_{31}. LO must identify a location in col 12. LO through L3 must identify sequential column locations. LO is selected if the PR-latch contains $xx00_2$, L1 is selected if the PR-latch contains $xx01_2$, . . . and L3 is selected if the PR-latch contains $xx11_2$. See Figure 4-3i.

JLL **Jump/Test Leftmost PR-Latch Bits**

FORMAT JLL (LO L1 L2 L3)

DESCRIPTION JLL specifies a test on the left two bits of the PR-latch. LO must identify a location in the current row group: row_0-row_7, row_8-row_{15}, row_{16}-row_{23} or row_{24}-row_{31}. LO must identify a location in col4. LO through L3 must identify consecutive column locations. LO is selected if the PR-latch contains $00xx_2$, L1 is selected if the PR-latch contains $01xx_2$, . . . and L3 is selected if the PR-latch contains $11xx_2$. See Figure 4-3h.

4.2 FLAG OUTPUT FUNCTIONS AND FO MICROPS

There are four flag output functions. The XMAS language includes a set of four FO microps for designating the flag output functions in specification statements. The FO microps are described below and summarized in Table 4-1.

FF0	**Force FO to Zero**
FORMAT	FF0
DESCRIPTION	The state of the Flag Output is forced to zero.
FFC	**Force FO to C Flag**
FORMAT	FFC
DESCRIPTION	The state of the Flag Output is forced to the current state of the C-flag.
FFZ	**Force FO to Z-Flag**
FORMAT	FFZ
DESCRIPTION	The state of the Flag Output is forced to the current state of the Z-flag.
FF1	**Force FO to One**
FORMAT	FF1
DESCRIPTION	The state of the Flag Output is forced to one.

4.3 FLAG INPUT FUNCTIONS AND FI MICROPS

There are four flag input functions. The XMAS language includes a set of four FI microps for designating the flag input functions in specification statements. The FI microps are described below and summarized in Table 4-1.

SCZ	**Set C-Flag and Z-Flag to FO**
FORMAT	SCZ
DESCRIPTION	The C-Flag and the Z-Flag are both set to the state of the Flag Input.
STZ	**Set Z-Flag to FI**
FORMAT	STZ
DESCRIPTION	The Z-Flag is set to the state of the Flag Input.
STC	**Set C-Flag to FI**
FORMAT	STC
DESCRIPTION	The C-Flag is set to the state of the Flag Input.
HCZ	**Hold C-Flag and Z-Flag**
FORMAT	HCZ
DESCRIPTION	The C-Flag and the Z-Flag are unaffected.

SECTION 5
FUNCTIONAL DESCRIPTION OF THE CPE ARRAY

The 3002 Central Processing Element (CPE) is a complete, 2-bit data processing module.

CPE ARRAY

Virtually any number of CPEs may be connected together to form a data processing section (called a CPE array) of any desired word width. To the microprogrammer, the CPE array constitutes a single functional unit (rather than a number of individual CPEs) with the N-bit registers, data paths and function circuits, where N is the number of CPEs in the array times two. Figure 5-1 illustrates the effective functional organization of a CPE.

During each microinstruction cycle, an encoded function is applied to the CPE Function Bus inputs. The Function Decoder, in decoding inputs, selects the arithmetic/logic unit function, generates the scratchpad address and controls the multiplexers. The A-multiplexer can select the data on the M-bus or the data in the addressed scratchpad register (R0 through R9 or T) or the accumulator (AC). The B-multiplexer can select the data on the I-bus or the data in the accumulator. The data selected by the B-Multiplexer is ANDed with the data on the K-bus. The result of the operation (i.e., the output of the arithmetic/logic unit) is deposited in the addressed scratchpad register or the accumulator. Certain operations also deposit data in the Memory Address Register.

CARRY INPUT AND CARRY OUTPUT

There is a single carry input (CI) to the low order bit position of the array. In an arithmetic operation, the carry input is included in the sum, and the carry output is the arithmetic carry for the sum. In a Boolean operation, where an arithmetic carry has no meaning, the carry output serves as a "not zero" indication.

LEFT INPUT AND RIGHT OUTPUT

There is a single left input (LI) to the high order bit position of the CPE array and a single right output from the low order bit position of the array. The left input and the right output are active only during right shift operations (when the carry input and carry output are inactive).

MICROINSTRUCTION WORD FIELDS

In a typical configuration, four fields in the microinstruction word are related to the CPE functions, as illustrated in Figure 5-2. The CPE field drives the CPE Function Bus; consequently, the CPE field designates the function to be performed by the CPE array. The K-bus field provides either direct or encoded drive for the K-bus inputs to the CPE array; the K-bus inputs have an effect on every operation performed by the CPE array. The carry input (CI) and left input (LI) to the array are normally connected to the Flag Output of the MCU; the Flag Output is, in turn, controlled by the FO field. The carry output (CO) and the right output (RO) are normally connected to the Flag Input of the MCU; the state of the Flag Input may be saved in the MCU's C or Z flags under control of the FI field.

5.1 CPE FUNCTIONS AND CPE MICROPS

CPE FUNCTIONS

The functions that the CPE is capable of performing are summarized in Table 5-1. A function is designated by a function group (F-Group) and a register group (R-Group). The F-Group is specified by the three high order bits of the CPE field. The R-Group is implied by the low order four bits of the CPE field. R-Group I includes R0 through R9, T and AC and is denoted by the symbol Rn. R-Group II and R-Group III include only T and AC and are denoted by the symbol AT.

F-Group 0 through F-Group 3, are arithmetic functions, with the exception of F-Group 0, R-Group III, which is a right shift function. The carry output (CO) reflects the arithmetic carry of the result of an arithmetic operation. F-Group 4 through F-Group 7 are Boolean functions. The carry output serves as a "not zero" indication in Boolean functions.

CPE MICROPS

The CPE microps (summarized in Table 5-2) are mnemonics for the functions performed by the CPE. Since the CPE microps are intrinsic to the XMAS language, they may be used in specification statements to specify the bit pattern for the CPE field. All CPE microps carry a default bit pattern assignment of all-zero or all-ones for the K-bus field.

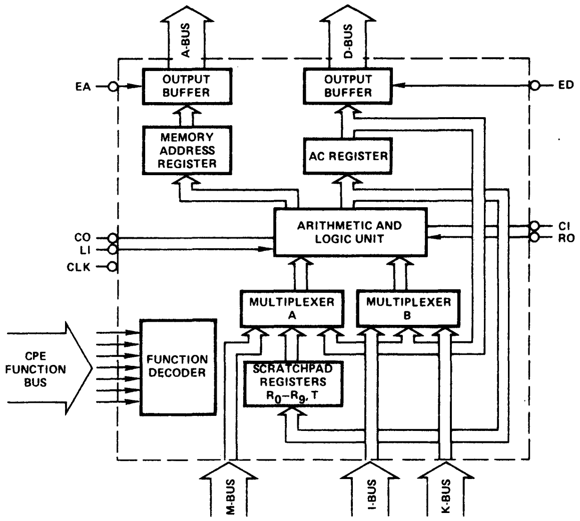


Fig. 5-1. CPE Array Block Diagram

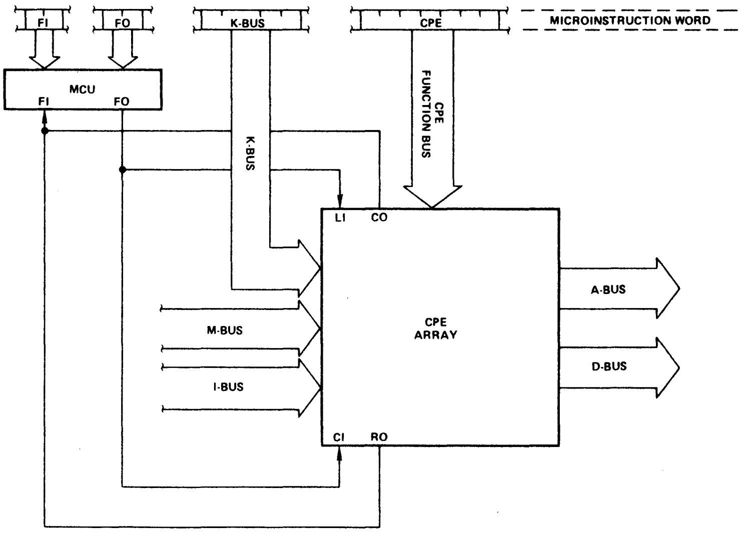


Fig. 5-2. Relationship Between CPE Array and Microinstruction Word

TABLE 5-1. CPE FUNCTION SUMMARY

F-GROUP	R-GROUP	MICRO-FUNCTION	
0	I	$R_n + (AC \wedge K) + CI \rightarrow R_n, AC$	
	II	$M + (AC \wedge K) + CI \rightarrow AT$	
	III	$AT_L \wedge (\overline{I_L \wedge K_L}) \rightarrow RO \quad LI \vee [(I_H \wedge K_H) \wedge AT_H] \rightarrow AT_H$ $[AT_L \wedge (I_L \wedge K_L)] \vee [AT_H \wedge (I_H \wedge K_H)] \rightarrow AT_L$	
1	I	$K \vee R_n \rightarrow MAR$	$R_n + CL + K \rightarrow R_n$
	II	$K \vee M \rightarrow MAR$	$M + CI + K \rightarrow AT$
	III	$(\overline{AT} \vee K) + (AT \quad K) + CI \rightarrow AT$	
2	I	$(AC \wedge K) - 1 + CI \rightarrow R_n$ (see Note 1)	
	II	$(AC \wedge K) - 1 + CI \rightarrow AT$	
	III	$(I \wedge K) - 1 + CI \rightarrow AT$	
3	I	$R_n + (AC \wedge K) + CI \rightarrow R_n$	
	II	$M + (AC \wedge K) + CI \rightarrow AT$	
	III	$AT + (I \wedge K) + CI \rightarrow AT$	
4	I	$CI \vee (R_n \wedge AC \wedge K) \rightarrow CO$	$R_n \wedge (AC \wedge K) \rightarrow R_n$
	II	$CI \vee (M \wedge AC \wedge K) \rightarrow CO$	$M \wedge (AC \wedge K) \rightarrow AT$
	III	$CI \vee (AT \wedge I \wedge K) \rightarrow CO$	$AT \wedge (I \wedge K) \rightarrow AT$
5	I	$CI \vee (R_n \wedge K) \rightarrow CO$	$K \wedge R_n \rightarrow R_n$
	II	$CI \vee (M \wedge K) \rightarrow CO$	$K \wedge M \rightarrow AT$
	III	$CI \vee (AT \wedge K) \rightarrow CO$	$K \wedge AT \rightarrow AT$
6	I	$CI \vee (AC \wedge K) \rightarrow CO$	$R_n \vee (AC \wedge K) \rightarrow R_n$
	II	$CI \vee (AC \wedge K) \rightarrow CO$	$M \vee (AC \wedge K) \rightarrow AT$
	III	$CI \vee (I \wedge K) \rightarrow CO$	$AT \vee (I \wedge K) \rightarrow AT$
7	I	$CI \vee (R_n \wedge AC \wedge K) \rightarrow CO$	$R_n \overline{\oplus} (AC \wedge K) \rightarrow R_n$
	II	$CI \vee (M \wedge AC \wedge K) \rightarrow CO$	$M \overline{\oplus} (AC \wedge K) \rightarrow AT$
	III	$CI \vee (AT \wedge I \wedge K) \rightarrow CO$	$AT \overline{\oplus} (I \wedge K) \rightarrow AT$

NOTE:

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

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

TABLE 5-2 . ALL-ZERO AND ALL-ONE K-BUS CPE FUNCTIONS

MNEMONIC	K-BUS = 00 MICRO-FUNCTION	MNEMONIC	K-BUS = 11 MICRO-FUNCTION
ILR	$R_n + CI \rightarrow R_n, AC$	ALR	$AC + R_n + CI \rightarrow R_n, AC$
ACM	$M + CI \rightarrow AT$	AMA	$M + AC + CI \rightarrow AT$
SRA	$AT_L \rightarrow RO \quad AT_H \rightarrow AT_L \quad LI \rightarrow AT_H$	—	(See Appendix B)
LMI	$R_n \rightarrow MAR \quad R_n + CI \rightarrow R_n$	DSM	$11 \rightarrow MAR \quad R_n - 1 + CI \rightarrow R_n$
LMM	$M \rightarrow MAR \quad M + CI \rightarrow AT$	LDM	$11 \rightarrow MAR \quad M - 1 + CI \rightarrow AT$
CIA	$AT + CI \rightarrow AT$	DCA	$AT - 1 + CI \rightarrow AT$
CSR	$CI - 1 \rightarrow R_n$ See Note 1	SDR	$AC - 1 + CI \rightarrow R_n$ See Note 1
CSA	$CI - 1 \rightarrow AT$	SDA	$AC - 1 + CI \rightarrow AT$
—	(See CSA above)	LDI	$I - 1 + CI \rightarrow AT$
INR	$R_n + CI \rightarrow R_n$	ADR	$AC + R_n + CI \rightarrow R_n$
	(See ACM above)	—	(See AMA above)
INA	$AT + CI \rightarrow AT$	AIA	$I + AT + CI \rightarrow AT$
CLR	$CI \rightarrow CO \quad 0 \rightarrow R_n$	ANR	$CI \vee (R_n \wedge AC) \rightarrow CO \quad R_n \wedge AC \rightarrow Rn$
CLA	$CI \rightarrow CO \quad 0 \rightarrow AT$	ANM	$CI \vee (M \wedge AC) \rightarrow CO \quad M \wedge AC \rightarrow AT$
—	(See CLA above)	ANI	$CI \vee (AT \wedge I) \rightarrow CO \quad AT \wedge I \rightarrow AT$
—	(See CLR above)	TZR	$CI \vee R_n \rightarrow CO \quad R_n \rightarrow R_n$
—	(See CLA above)	LTM	$CI \vee M \rightarrow CO \quad M \rightarrow AT$
—	(See CLA above)	TZA	$CI \vee AT \rightarrow CO \quad AT \rightarrow AT$
NOP	$CI \rightarrow CO \quad R_n \rightarrow R_n$	ORR	$CI \vee AC \rightarrow CO \quad R_n \vee AC \rightarrow R_n$
LMF	$CI \rightarrow CO \quad M \rightarrow AT$	ORM	$CI \vee AC \rightarrow CO \quad M \vee AC \rightarrow AT$
—	(See NOP above)	ORI	$CI \vee I \rightarrow CO \quad I \vee AT \rightarrow AT$
CMR	$CI \rightarrow CO \quad \overline{R}_n \rightarrow R_n$	XNR	$CI \vee (R_n \wedge AC) \rightarrow CO \quad R_n \overline{\oplus} AC \rightarrow R_n$
LCM	$CI \rightarrow CO \quad \overline{M} \rightarrow AT$	XNM	$CI \vee (M \wedge AC) \rightarrow CO \quad M \overline{\oplus} AC \rightarrow AT$
CMA	$CI \rightarrow CO \quad \overline{AT} \rightarrow AT$	XNI	$CI \vee (AT \wedge I) \rightarrow CO \quad I \overline{\oplus} AT \rightarrow AT$

NOTES:

1. 2's complement arithmetic adds 111 . . . 11 to perform subtraction of 000 . . . 01.
2. R_n includes T and AC as source and destination registers in R-group 1 micro-functions.
3. Standard arithmetic carry output values are generated in F-group 0, 1, 2 and 3 instructions.

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

5.2 STRINGOPS FOR CPE FUNCTIONS

VARIATIONS OF CPE FUNCTIONS

By assigning specific values to the data on the K-bus inputs (K) and the carry shift input (CI or LI), many useful operations can be derived from the CPE functions, which are summarized in Table 5-1. For example, consider the following function (F-Group 0, R-Group I):

$Rn + (AC \wedge K) + CI \rightarrow Rn, AC$

If K and CI are zero, the effective function is:

$Rn \rightarrow AC, Rn$

which transfers the data from the specified register (Rn) to the accumulator (AC). If K is all ones and CI is zero, the effective function is:

$Rn + AC \rightarrow Rn, AC$

which adds the data in the specified register (Rn) and the accumulator (AC) and deposits the sum in both registers. If K is zero and CI is one, the effective function is

$Rn + 1 \rightarrow Rn, AC$

which increments the data in specified register (Rn) and deposits the result in both the specified register and the accumulator. An effective function may be indicated in a specification statement with a CPE microp and an FO microp (when the latter is necessary). The increment function above could be designated in a specification statement by:

```
ILR (R2)  FF1
```

The CPE microp ILR carries a default bit pattern assignment of zero for the K-bus field.

STRING STATEMENT

The XMAS language STRING statement provides a mechanism for defining new mnemonics that may be used to represent CPE functions in specification statements. For example, the STRING statement:

```
INCE  STRING  'FF1  ILR'
```

defines the mnemonic INCE. It can be used in a specification statement as

```
INCE (R2)
```

to represent the increment function described previously.

STRINGOPS

CPE function mnemonics that are defined in STRING statements are called "stringops" to differentiate them from microps. Using a stringop in a specification statement is often a more convenient and meaningful way of specifying a CPE function than using the corresponding CPE microp or a combination of a CPE microp and an FO microp.

A SET OF STRINGOPS FOR THE CPE

A set of stringops that cover most of the useful CPE operations is described in Appendix J. Unlike the microps, stringops are not intrinsic to the XMAS language. All stringops used in an XMAS language program must be defined in STRING statements in the declaration part of the program. The defining STRING statement for each stringop is given in Appendix J. For the programmer's convenience, a complete list of these STRING statements is given in Table J-1.

SECTION 6
MICROPROGRAMMING TECHNIQUES AND EXAMPLES

Section 6.1 discusses the proper use of XMAS declaration statements in defining a framework that effectively represents the characteristics of a given Series 3000 configuration. Section 6.2 describes how to efficiently assign each microinstruction to a microprogram memory location. Section 6.3 identifies the programming differences that must be considered when writing a microprogram that is to be executed by a system with a "pipelined" architecture. Finally, Section 6.4 provides examples of how to write and assign to memory microprograms for both non-pipelined and pipelined systems.

6.1 USE OF XMAS DECLARATION STATEMENTS

The declaration statements in a XMAS language program establish the framework for writing and specification statements, as described in Section 2.

FIELD STATEMENT

ADDRESS STATEMENT

The FIELD and ADDRESS statements are used to describe attributes of the particular Series 3000 hardware configuration. If the hardware designer implements a functional block of logic (e.g., memory or I/O logic) requiring a control field in the microinstruction, the microprogrammer must declare and describe that control field via a FIELD statement. Similarly, if the configuration includes more than 512 words of microprogram memory, the microprogrammer must identify a memory address extension field with an ADDRESS statement. Remember that the name of the memory address extension field must be defined in a FIELD statement. For example, the two statements:

```
XADR   FIELD        LENGTH = 2;
XADR   ADDRESS;
```

define the two-bit field XADR, and identify it as a memory address extension field, respectively.

The usefulness of other declaration statements may not be quite as obvious as the FIELD and ADDRESS statements, because these other declaration statements make the microprogram easier to write rather than to describe the hardware configuration. Proper use of the STRING, VALUE IMPLY and KBUS declaration statements can significantly increase the microprogrammer's efficiency in successfully completing his task.

STRING STATEMENT

STRING statements can be particularly helpful. A microprogrammer may know that a particular group of symbols will be needed frequently in his XMAS language program. The STRING statement allows the programmer to define a symbol and associate a character string with that symbol. For example, the STRING statement:

```
NEXT  STRING  'LMI(R7)  READ  FFI  KB=Q';
```

associates NEXT with the string of characters enclosed between the single quote (') characters. NEXT can be used in any specification statement in place of the character string it represents. The CPE stringops, described in Section 5.2, are an excellent example of character strings which have been assigned mnemonics by STRING statements. A stringop groups a specific CPE function, flag control function and a K-bus function together into a single functional mnemonic.

VALUE STATEMENT

The VALUE statement allows the microprogrammer to define a symbol and associate a numeric value with it. The value symbol can then be used in specification statements wherever the numeric equivalent could be used. The most obvious advantage of using a symbol in place of the numeric value is that should the programmer decide to change the numeric value, only one VALUE statement need be changed, instead of many specification statements.

IMPLY STATEMENT

An IMPLY statement declares the default bit pattern(s) that a microp carries for one or more fields. By having certain microps carry default bit patterns for appropriate fields, the programmer can free himself from having to assign explicitly a bit pattern to every field in every specification statement. Recall, however, that an explicit keyword or microp assignment for a given field overrides a default pattern that a microp may carry for that field.

KBUS STATEMENT

Recall that each CPE microp implies an all-zero or all-one default pattern for the K-bus field. If these defaults are to be effective, however, the microprogrammer must define (via a field statement) and identify (via a KBUS statement) the K-bus field that is to receive these defaults. For example, the statements:

```
KB   FIELD  LENGTH=16 MICROPS   (K8000 = 8000H):
     KB   KBUS;
```

define the 16-bit field KB, define the microp K8000 and identify KB as the K-bus default provided by a CPE microp by a keyword assignment (e.g., KB-8000H) or a user-defined microp assignment).

6.2 MICROPROGRAM MEMORY ASSIGNMENT

Ultimately, the microprogrammer must assign each specification statement in the microprogram to a particular location in the microprogram memory. For example, the following specification statement:

```
7BH: LAB:    ILR(R3)    FFO    STZ    JFL(NC, TC);
```

ADDRESS IDENTIFIER

includes an assignment to memory location $7B_{16}$ (i.e., row 7 column 11). In addition, this statement specifies that the address identifier LAB can be used to reference this memory location (refer to Section 2.14).

UNCONDITIONAL JUMPS

The microprogrammer should initially write the microprogram in the logical order in which it will be executed, giving only minimal regard to memory assignment. When a sequence of microinstructions is to be executed in the same order in which they will appear in the source file, an explicit JUMP field specification can be omitted; XMAS will attempt to supply an unconditional jump code that will allow the microinstruction currently being specified to access the microinstruction specified by the next specification statement in the source file.

When a program branch is to be executed unconditionally, the general non-committal JMP microp should be used rather than selecting JCC, JZR or JCR, unless the Interrupt Strobe Enable (ISE) line is to be enabled (use JZR) or the PR-latch outputs are to be enabled (use JCE), as described in Section 4.1.

CONDITIONAL JUMPS

When a conditional branch in the program sequence is required, the programmer will use one of the conditional jump microps in the jump field (JFL, JCF, JZF, JPR, JLL, JRL or JPX). When writing the microprogram, it will be helpful to note the number of possible destinations for each conditional jump. It is also advantageous to assign an address identifier to each possible destination, and to reference each destination in the expression portion of the JMP microp with the address identifier, instead of the integer that represents the actual memory location assigned. For example:

```
ADR(R5)     FFO    JFL(NCY, CY)
.
.
.
CY:     INR(R5)  SCZ
NCY: . . .
```

In this way, the microprogram can be written with all jump targets specified before the specification statements are assigned to memory. The programmer, using the control language, can cause XMAS to output a cross reference directory containing an alphabetical list of all XMAS specification statement labels and, for each label, a list of source file record numbers in which the label is referenced.

MICROINSTRUCTION FLOWCHART

Having written the microprogram with all sequencing represented symbolically or implied by statement order, the actual assignment to microprogram memory locations must be indicated. To assist in this task, a complete microinstruction state sequence chart should be prepared. In such a chart, each microinstruction is represented by a node in the diagram as shown in Figure 6-4 (in section 6.4). Conditional jumps should be labeled as to type and condition corresponding to each possible destination. It is also helpful to show any address labels (identifiers) that may be associated with a microinstruction.

GRID DIAGRAM

The process of assignment can be assisted by using a grid diagram of the microprogram memory showing the 32 rows and 16 columns. As each microinstruction is assigned, the microprogram memory grid diagram is marked to show occupancy of that word and the flowchart is marked to show assignment of the microinstruction.

Before assignment begins, however, one should count all conditionals of each type to assure there are enough targets available to place them.

Using the flowchart as a guide, memory assignment can be easily accomplished if the following sequence is followed.

HARDWARE CONSIDERATIONS

1. Assign those microinstructions whose memory locations are dictated by hardware considerations. For example, the first instruction in a system initialization routine might be required by hardware to be assigned to memory location 00. The location of the first instruction in an interrupt routine might also be defined by hardware constraints.

ASSIGNING CONDITIONAL JUMPS

2. To do the best assignment the most restricted microinstructions should be assigned first. In general, clusters of conditional jump targets which must be located within a limited range, constitute the most restricted set of microinstructions. Assign all conditional jump targets before assigning the microinstructions that reference these targets (i.e., have a conditional JUMP microp). Recall that in 512-word microprogram memory (or in a 512 word plane of an extended memory), there are only 64 possible destination pairs for the JCF, JZF and JFL jump microps, since all three use columns 2 and 3 or columns 10 and 11 as their jump target. It is important, therefore, to insure that enough destination pairs are available for the conditional jumps used in a microprogram. Also remember that the JPX and JPR conditional microps can require one entire row each time that they are used.

ASSIGNING UNCONDITIONAL JUMPS

3. Leave long chains of unconditional jump sequences until last because they have the greatest range of possible destinations. Remember that when the general JMP microp is used or when no explicit JUMP field is specified, the next microinstruction to be executed must be in the current row, current column or row zero. Row zero locations should be used judiciously because only they can be reached from anywhere else in the program using a single JZR jump function.

REASSIGNMENT

4. When reassignment becomes necessary, sequences of unconditional microinstructions should be considered first since they are the easiest to move.

The programmer can, using the control language, cause XMAS to output the source statements and or bit patterns for the microprogram.

If microprogram memory has been incorrectly assigned, XMAS will output the appropriate error message to the list file (see Appendix G). The programmer can also have XMAS produce a graphic representation of the microprogram memory image (refer to Section 3.3) which can be extremely helpful in reassigning memory locations if required.

Figure 6-5 shows the memory assignments that have been made for the two example microprograms in Section 6.4, using the microinstruction flow chart in Figure 6-4.

For a further discussion of microinstruction mapping refer to Intel's AP-13, "Designing Central Processors Using Intel's Series 3000 Computing Elements."

6.3 PIPELINED VS NON-PIPELINED ARCHITECTURE: PROGRAMMING CONSIDERATIONS

A "pipelined" architecture can be implemented by placing a register of edge-triggered D-type flip-flops between the microprogram memory outputs and the circuitry controlled by those outputs. This register allows the executing of the current microinstruction to overlap the fetching of the next microinstruction. The address control lines from the microprogram memory which provide the 3001 MCU with microprogram sequence information (i.e., the jump code) are not routed through the pipeline register, however. Instead, they are applied directly to the AC0-AC7 inputs of the MCU. Figure 6-1 illustrates both a pipelined and non-pipelined architecture.

RESULT DELAY IN PIPELINED SYSTEM

The major difference between microprograms written for pipelined and non-pipelined architectures are associated with conditional jumps which test the results of arithmetic or logical operations executed by the CPE array, that is, conditional jumps caused by the JFL, JCF or JZF microps. In a pipelined architecture, the results of the arithmetic or logical operations are delayed by one microinstruction. That is, the 3001 MCU will receive the jump code when the microinstruction is sent to the pipelined register; however the flag logic input (FI), indicating the results of the microinstruction execution, will not be received by the MCU for another microinstruction cycle. Consequently, the conditional jumps caused by the JFL, JCF and JZF microps must be delayed by at least one microinstruction after the execution of the operation for which the result is to be tested.

JFL MICROP

Remember that the flag input, FI, (usually tied to the CPE array's carry out, CO, and shift right out, RO, lines) is always reflected in the MCU's F-latch before the jump test is made by the MCU. Thus in non-pipelined systems, the JFL microp should appear in the same microinstruction that specifies the operation whose result will test; and in pipelined systems, the JFL microp should appear in the next microinstruction after the one which specifies the operation whose result JFL will test.

JCF AND JZF MICROPS

On the other hand, the setting or resetting of the C and Z flags by the MCU to reflect the level on the flag logic input (FI) line, as the result of a flag control microp (SCZ, STC or STZ), will not occur until after the jump test is made by the MCU. Consequently, even in non-pipelined systems the JCF or JZF microp should appear in the next microinstruction after the one which specifies the operation whose result JCF or JZF will test. In pipelined systems, the JCF or JZF jump microp must be delayed by two or more microinstructions.

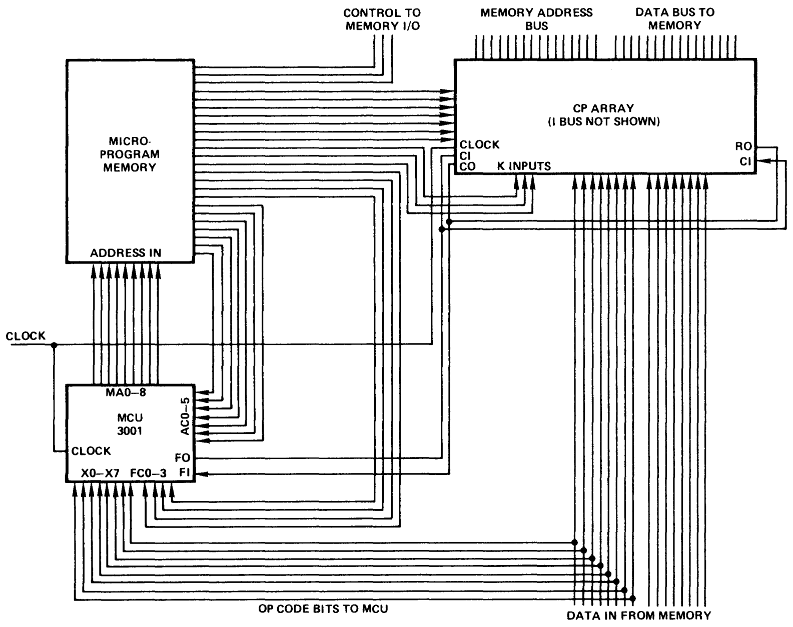
6.4 MICROPROGRAMMING EXAMPLES

Example 1, shown in Figure 6-2, illustrates a 16-bit multiplication routine written for execution on a non-pipelined system. Example 2, shown in Figure 6-3, illustrates the same type of routine, except it was written for execution on a pipelined system. (Example 2 does not show a declaration part because it is assumed to have the same one listed in Example 1.) The primary difference between the two examples is the choice of conditional jump microps in the LOOP+1 and STAY specification statements. In Example 1, JCF and JZF microps are used; in Example 2, two microps are used.

MICROINSTRUCTION FLOWCHART

Figure 6-4 illustrates the microinstruction flowchart for examples 1 and 2. The conditional jumps at LOOP+2 and STAY are visually apparent, as are their possible targets.

Figure 6-5 shows where each microinstruction in the examples has been assigned within microprogram memory. The two possible targets of each conditional jump (EXIT, STAY and ZERO, ONE) were assigned first to columns 2 and 3, rows 5 and 6 respectively. Then the unconditional jump targets were assigned. Notice that LOOP, which is the unconditional target for START + 2 and ZERO, was assigned to the same column as START + 2 and the same row as ZERO so that it could be accessed by both.



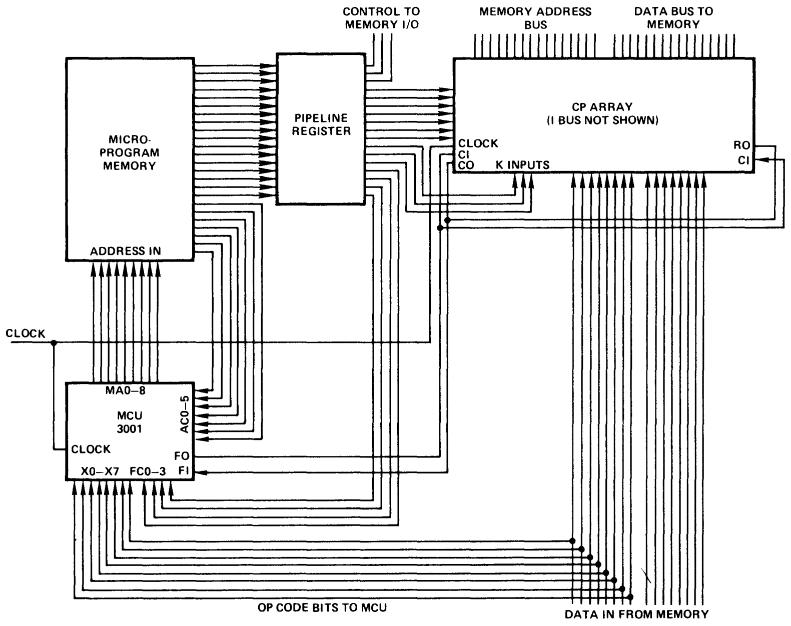


Fig. 6-1. Pipelined vs. Non-pipelined Architecture

```
        /* UNSIGNED 16-BIT MULTIPLY */
        /* ASSUME THAT MULTIPLICAND IS BUFFERED AND AVAILABLE ON M-BUS */
        /* DECLARATION PART APPLIES TO EXAMPLES 1 AND 2 */
        /* K-BUS DEFINITION */

           KB FIELD LENGTH = 16
                    MICROPS (KZERO = Ø, KONES=FFFFH, KFFFØ=FFFØH);
           KB KBUS;

        /* STRINGOP DEFINITIONS (SEE SECTION 5.2) */

           ADDM  STRING  'FFØ AMA';
           INCR  STRING  'KZERO FF1 INR';
           MSKR  STRING  'FFØ TZR';
           TSTR  STRING  'KONES FFØ TZR';
           CLRR  STRING  'KZERO CLR';
           SETR  STRING  'KZERO FFØ CSR';
           SHRT  STRING  'KZERO SRA';

        /* OTHER CONVENIENT CHARACTER STRINGS */

           COUNT  STRING  'R8';  /* LOOP COUNT IN R8 */
           P.P    STRING  'AC';  /* PARTIAL PRODUCT IN AC */
           MULTR  STRING  'T';   /* MULTIPLIER IN T */

        /* SPECIFICATION PART: EXAMPLE 1 FOR NON-PIPELINED SYSTEMS ONLY */

               /* INITIALIZE LOOP COUNTER */

Ø5ØH: START: SETR (COUNT);        /* SET COUNT TO ALL ONES */
Ø51H:        MSKR (COUNT) KFFØ;   /* SET COUNT TO -16 */
Ø54H:        CLRR (P.P) STZ;      /* CLEAR AC AND Z-FLAG */

             /* MAIN MULTIPLICATION LOOP */

Ø64H: LOOP:  TSTR (COUNT) STC;  /*TEST COUNT FOR ALL ZEROS */

Ø61H:        SHRT (MULTR) FFZ STZ JCF (EXIT, STAY); /* SHIFT LSB OF
                 MULTIPLIER INTO Z, SHIFT LSB OF PARTIAL PRODUCT
                 INTO MSB OF T, IF COUNT=ØØØØ JUMP TO EXIT */

Ø53H: STAY:  INRC (COUNT) JZF (ZERO, ONE); /* INCREMENT COUNT AND JUMP
                 AFTER TESTING LSB OF MULTIPLIER */

Ø63H: ONE:   ADDM (P.P); /* ADD MULTIPLICAND TO PARTIAL PRODUCT */

Ø62H: ZERO:  SHRT (P.P) STZ JMP (LOOP); /* SHIFT LSB OF PARTIAL PRODUCT
                 INTO Z-FLAG AND JUMP TO LOOP */

Ø52H: EXIT:  . . .       /* FINAL PRODUCT IN T */
```

Fig. 6-2. Example 1: 16-Bit Multiply Routine for Non-Pipelined Systems

```
        /* UNSIGNED 16-BIT MULTIPLY */
        /* ASSUME THAT MULTIPLICAND IS BUFFERED AND AVAILABLE
           ON THE M-BUS */
        /* DECLARATION PART WILL BE THE SAME AS THAT SHOWN FOR
           EXAMPLE 1 */
        /* SPECIFICATION PART: EXAMPLE 2 FOR PIPELINED SYSTEMS ONLY */
           /* INITIALIZE LOOP COUNTER */
Ø5ØH: START: SETR (COUNT); /* SET COUNT TO ALL ONES */
Ø51H:        MSKR (COUNT) KFFFØ;  /* SET COUNT TO -16 */
Ø54H:        CLRR (P.P) STZ;  /* CLEAR AC AND Z-FLAG */
             /* MAIN MULTIPLICATION LOOP */
Ø64H: LOOP:  TSTR (COUNT);  /* TEST COUNT FOR ALL ZEROS */
Ø61H:        SHRT (MULTR) FFZ JFL (EXIT, STAY);  /* SHIFT LSB OF
                 MULTIPLIER ONTO FI, SHIFT LSB OF PARTIAL PRODUCT
                 INTO MSB OF T, IF COUNT=ØØØØ JUMP TO EXIT */
Ø53H: STAY:  INCR (COUNT) JFL (ZERO, ONE); /* INCREMENT COUNT
                 AND JUMP AFTER TESTING LSB OF MULTIPLIER */
Ø63H: ONE:   ADDM (P.P);  /* ADD MULTIPLICAND TO PARTIAL PRODUCT */
Ø62H: ZERO:  SHRT (P.P) STZ JMP (LOOP);  /* SHIFT LSB OF PARTIAL
                 PRODUCT INTO Z-FLAG AND JUMP TO LOOP */
052H: EXIT:  . . .      /* FINAL PRODUCT IN T */
```

Fig. 6-3. Example 2: 16-Bit Multiply Routine for Pipelined Systems

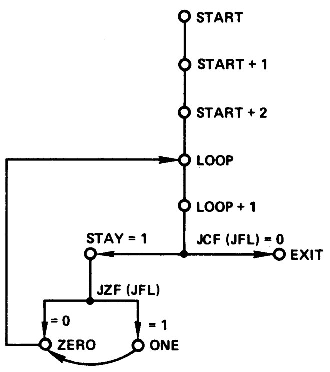


Fig. 6-4. Microinstruction Flowchart

ROW	0	1	2	3	4	5	6	7	8	9	A	B
05	START	START +1	EXIT	STAY	START +2							
06		LOOP +1	ZERO	ONE	LOOP							

Fig. 6-5. Microprogram Memory Grid Diagram

SECTION 7
XMAP LANGUAGE

When XMAS assembles a program it produces a microcode file. XMAP, directed by XMAP language statements, operates on the contents of the microcode file and produces a ROM programming file, which is suitable for programming the actual ROM and/or PROM devices that constitute the physical microprogram memory.

MICROCODE FILE

For each specification statement in an XMAS language program, XMAS outputs to the microcode file a complete description of the specified microinstruction word. This description includes the address of microinstruction word (as specified in the label part of the specification statement) and the logical bit pattern for every field in the microinstruction word. Thus, the microcode file contains a complete logical description of a microprogram.

MICROPROGRAM MEMORY IMPLEMENTATION

ROM and PROM devices are available in a number of organizations including: 512 words by 8 bits, 512 words by 4 bits and 256 words by 4 bits. Individual ROMs or PROMs are configured to form a physical microprogram memory of the required organization. For example, to construct a microprogram memory of 512 words by 24 bits, three 512 word by 8 bit ROM's could be used, where each ROM would carry an 8-bit slice of the microinstruction word for each microinstruction address.

XMAP'S ROLE

The role of XMAP is to map the bit patterns in the microcode file into the desired ROM or PROM bit locations. The XMAP language is used to describe the organization of the microprogram memory components and the detailed mapping procedure.

Section 7.1 provides an introduction to the XMAP language by way of an example mapping problem. Sections 7.2 through 7.4 provide a formal description of the XMAP language. In the formal descriptions, syntax is presented using a modified BNF notation; this notation is described in Appendix A.

7.1 XMAP LANGUAGE OVERVIEW

FREE-FORMAT

The XMAP language is a free format language. Syntactic entities may appear anywhere in a source record. Commas and spaces may be used freely and interchangeably to enhance readability. A comment, which is delimited by /* and */, may appear anywhere a blank character would be allowed.

An XMAP program consists of a series of XMAP statements. A semi-colon (;) marks the end of each statement, and the reserved word EOF marks the end of the XMAP program.

XMAP STATEMENT TYPES

There are only two types of statements in the XMAP language: ROM specifications and mapping specifications. A ROM specification describes the organization of a ROM or PROM. A mapping specification describes the relationship between microprogram addresses and physical ROM addresses for a single ROM or PROM. In addition, a mapping specification dictates which bits of which fields of the microinstruction word are to be mapped into each bit position of a single ROM or PROM.

EXAMPLE

The important characteristics of the XMAP language can be illustrated with a simple example. Figure 7-1 is a pictorial representation of a simple mapping problem.

The microinstruction word has a total of 19 bits and is composed of the following five fields: CPE (7 bits), FI (2 bits), FO (2 bits), JUMP (7 bits) and KB (1 bit). The bars (—) over bit positions in the microinstruction word pictured in Figure 7-1 mean that the corresponding bits must be inverted (i.e., in microcode file are to be programmed as 0s in the physical ROM and vice versa).

MEMORY IMPLEMENTATION

The physical microprogram memory is to be implemented using two 512 word by 8 bit ROMs and two 256 word by 4 bit ROMs, as illustrated in Figure 7-1. ROM #2 will carry program

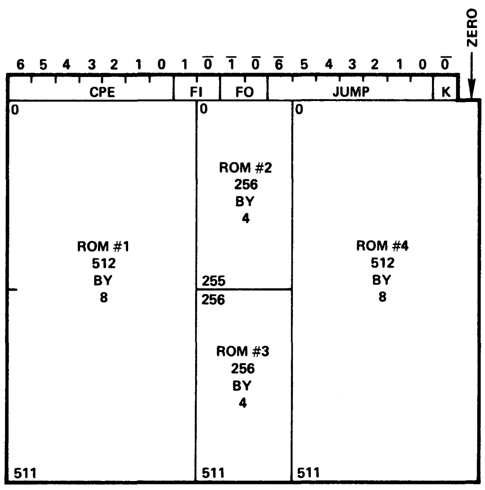


Fig. 7-1. Example Microprogram Memory Configuration

addresses 0 through 255, and ROM #3 will carry program addresses 256 through 511. Since only 19 bits are required for each microinstruction word, the low order bit position of ROM #4 will be programmed as zero (an arbitrary choice).

EXAMPLE PROGRAM

An XMAP language program that describes these mapping requirements is shown in Figure 7-2.

Statement (1) is a ROM specification. It states that the ROM or PROM under consideration has an organization of 512 words by 8 bits.

Statement (2) is a mapping statement. The first part of this statement:

```
WORDS 0 TO 511
```

means that microprogram addresses 0 through 511 are to be mapped into consecutive ROM locations beginning at ROM-address 0. (ROM addresses always begin at 0; the numbers that appear after the reserved word WORDS [i.e., 0 and 511] are always microprogram addresses.) The second part of this statement is the reserved word BITS followed by a list of items enclosed in parentheses. Each item corresponds to a bit position in the ROM beginning with the most significant position and ending with the least significant bit position in the bit list. Each item identifies the field and bit positions within the field that the corresponding ROM bit position is to receive. Since the first (most significant) item is CPE (6), the most significant bit position of each word of ROM1 is to receive the most significant bit, number 6, of the CPE field; the least significant bit position of each word of ROM1 is to receive bit number 1 of the FI field.

Statement (3) describes the organization of the next ROM to be programmed. The first part of statement (4) specifies that microprogram address 0 through 255 are to be mapped into consecutive ROM locations beginning at ROM address 0 (as always). The reserved word INVERT means that bits programmed in this ROM are to be inverted. The BITS list has four items since this ROM has four bit positions. The most significant bit position of the ROM will receive the inverted state of bit 6 of the JUMP field for each microprogram address.

Statement (5) is not really necessary since statement (3) is still in effect. The first part of statement (6) specifies that microprogram addresses 256 through 511 are to be mapped into consecutive ROM locations beginning at ROM address 0 (as always).

Statement (6) specifies that the next ROM to be programmed has an organization of 512 words by 8 bits. The last (least significant) item in the BITS list of statement (8) means that the least significant bit position of the ROM will receive a zero for all ROM addresses. The second least significant item in the BITS list is:

```
K(0)'
```

The single quote (') indicates that the designated bit (bit 0 of field KB) is to be programmed in the inverted state. Note that the other bits in this ROM (i.e., the JUMP field bits) are not to be inverted.

Statement (9), the reserved word EOF, terminates the XMAP language program.

INVERTED ADDRESS INPUT

In configurations where an inverted address will be applied to the address inputs of a ROM or PROM, it is necessary to map microprogram addresses into ascending ROM locations in decending order. For example, if the address applied to ROM #4 is to be inverted, the first part of statement (8) would be written:

```
WORDS 511 TO 0
```

meaning that microprogram address 511 is to be mapped into ROM address 0, microprogram address 510 is to be mapped into ROM address 1, . . . and microprogram address 0 is to be mapped into ROM address 511.

```
(1) ROM  512 BY 8                           /* ROM #1 ORGANIZATION */;
(2) WORDS  Ø TO 511                         /* ROM #1 MAPPING */
                BITS(CPE(6), CPE(5)
                     CPE(4), CPE(3)
                     CPE(2), CPE(1)
                     CPE(Ø), FI(1));



(3) ROM  256 BY 4                           /* ROM #2 ORGANIZATION */;
(4) WORDS  0 TO 255  INVERT                 /* ROM #2 MAPPING */
                BITS(FI(Ø), FO(1)
                     FO(Ø), JUMP(6);


(5) ROM  256 BY 4                           /* ROM #3 ORGANIZATION */;
(6) WORDS  256 TO 511  INVERT               /* ROM #3 MAPPING */
                BITS(FI(Ø), FO(1)
                     FO(Ø), JUMP(6);


(7) ROM  512 BY 8                           /* ROM #4 ORGANIZATION */;
(8) WORDS   Ø  TO 511                       /* ROM #4 MAPPING */
                BITS(JUMP(5), JUMP(4)
                     JUMP(3), JUMP(2)
                     JUMP(1), JUMP(Ø)
                     K(Ø); Ø);



(9) EOF
```

Fig. 7-2. Example XMAP Language Program

7.2 XMAP LANGUAGE PROGRAM

SYNTAX

The following rule governs the structure of an XMAP language program:

$$\langle\text{program}\rangle ::= \begin{bmatrix}\text{ROM spec;}\\ \text{mapping spec;}\end{bmatrix} \ldots \text{EOF}$$

There are two statement types: a ⟨ROM spec⟩ and a ⟨mapping spec⟩. A semi-colon marks the end of each statement, and the reserved word EOF marks the end of the program.

FREE FORMAT

XMAP language statements are free format; physical record boundaries and character positions within a record are not significant. There are no restrictions as to where a statement begins or ends or how long it is. A statement may span several records or may be wholly contained within a single record.

SPACES, COMMAS, COMMENTS AND PERIODS

Spaces and commas may be used freely between statement elements to enhance readability. A comment (see Section 2.6 for syntax) may appear anywhere a space is allowed. XMAP treats the period (.) as a null character.

7.3 ROM SPECIFICATION

SYNTAX

A ROM specification describes the organization of a component (e.g., ROM or PROM) of the microprogram memory that is to be programmed.

⟨ROM spec⟩ :: = ROM ⟨integer1⟩ BY ⟨integer2⟩

⟨integer1⟩ :: = ⟨integer⟩

⟨integer2⟩ :: = ⟨integer⟩

Any valid ⟨integer⟩ representation (see Section 2.4) may be used for ⟨integer1⟩ and ⟨integer2⟩.

The number of words (i.e., addressable locations) in the ROM or PROM being described is specified by ⟨integer1⟩. The number of bits in each word is specified by ⟨integer2⟩. A ⟨ROM spec⟩ remains in effect until another ⟨ROM spec⟩ appears.

EXAMPLE

```
ROM 512 BY 8;

ROM 256 BY 4;
```

The first ⟨ROM spec⟩ describes a memory component with 512 words of 8 bits each. The second statement describes a memory component with 256 words of 4 bits each.

7.4 MAPPING SPECIFICATION

SYNTAX

A mapping specification controls the way in which specified bits from the microinstruction words from specified microprogram addresses are to be mapped into a component (e.g., ROM or PROM) of the microprogram memory.

⟨mapping spec⟩ :: = WORDS ⟨range pair⟩
    [INVERT] BITS (⟨bit spec list⟩)

⟨range pair⟩ :: = ⟨integer3⟩ TO ⟨integer4⟩

$$\langle\text{bit spec list}\rangle ::= \begin{Bmatrix}\langle\text{field name}\rangle\ (\langle\text{integer5}\rangle)\ [\,'\,]\\ 0\\ 1\end{Bmatrix} \ldots$$

```
⟨integer3⟩ :: = ⟨integer⟩

⟨integer4⟩ :: = ⟨integer⟩

⟨integer5⟩ :: = ⟨integer⟩
```

Any valid ⟨integer⟩ representation (see Section 2.4) may be used for ⟨integer3⟩, ⟨integer4⟩ and ⟨integer5⟩.

The ⟨range pair⟩ designates the microprogram addresses that are to be mapped into consecutive ROM locations beginning at ROM address 0. ⟨integer3⟩ designates the microprogram address that is to be mapped into ROM address 0. If ⟨integer4⟩ is greater than ⟨integer3⟩, microprogram address ⟨integer3⟩ + 1 is mapped into ROM address 1, etc. If ⟨integer4⟩ is less than ⟨integer3⟩, microprogram address ⟨integer3⟩ − 1 is mapped into ROM address 1, etc. In all cases, the absolute value of ⟨integer3⟩ − ⟨integer4⟩ must be one less than the number of locations in the ROM being programmed (i.e., ⟨integer1⟩ of the previous ⟨ROM spec⟩). The optional reserved word INVERT, when present, means that the bits listed in the ⟨bit spec list⟩ are to be inverted. That is, zeros input from the microcode file are to be written in the ROM programming file as ones, and vice versa.

The ⟨bit spec list⟩ designates the contents of each bit position of the ROM word. The ⟨bit spec list⟩ must have exactly as many items as the number of bits in the ROM word (i.e., ⟨integer2⟩ in the previous ⟨ROM spec⟩).

An item in the ⟨bit spec list⟩ can take one of three forms. In the form:

```
⟨field name⟩ (⟨integer5⟩) [']
```

⟨field name⟩ must be either a field name that is intrinsic to the XMAS language or a field name that was defined in the XMAS language program whose microcode file is being mapped. ⟨integer5⟩ is a bit number within the field (the rightmost bit of a field is bit 0). The optional apostrophe ('), when present, indicates that the bit is to be inverted. The other two forms that an item in the ⟨bit spec list⟩ can take are 0 and 1. A 0 indicates that the corresponding bit position of the ROM is to receive a 0, and a 1 indicates that the corresponding bit position is to receive a 1.

EXAMPLE

```
WORDS  256  TO  511
    BITS (0, 1, JUMP(4),  KB(2)').
```

This statement specifies that microprogram addresses 256 through 511 are to be mapped into consecutive ROM locations beginning at ROM Address 0. The four bit positions of each ROM word beginning with the most significant bit position are to receive 0, 1, bit 4 of the JUMP field and the complement of bit 2 of the KB field, respectively. KB is assumed to be a user-defined field.

EXAMPLE

```
WORDS  511  TO  256
    INVERT BITS (0, 1, JUMP(4),  KB(2)');
```

This statement specifies that microprogram address 511 through 256 (taken in decending order) are to be mapped into consecutive ROM locations beginning at ROM address 0. All items in the BITS list are to be complemented. The four bit positions of each ROM word are to receive 1, 0, the complement of bit 4 of the JUMP field and bit 2 of the KB field. The following statement:

EXAMPLE

```
WORDS  511  TO  256
    BITS (1, 0, JUMP(4)', KB(2));
```

This statement is identical to the previous statement.

SECTION 8
XMAP LISTING OUTPUT

In processing an XMAP language program, XMAP is capable of generating the following reports:

- Listing of XMAP language source statements and a binary dump of each ROM specified by a ROM mapping statement;
- XMAP program summary

The user selects the information he wants via the control language (Section 9).

LIST FILE

XMAP outputs the selected information to the FORTRAN data file that has been designated as the list file. The list file is page oriented; running error and page counts are given at the top of each page. Via the control language, the user can specify an optional page title, the number of lines per page, the number of characters per line, and the form feed mode to be used between pages. The list file line width must be a minimum of seventy-two characters.

Subsequent subsections describe the XMAP reports.

8.1 XMAP SOURCE STATEMENTS AND ROM DUMPS

The user may choose to have both the XMAP language statements and the ROM dumps included in the list file. A partial listing is shown in Figure 8-1. The three periods in a vertical line are not part of the actual output of but simply indicate that the entire dump is not shown.

```
XMAP VERS 1.0 XMAP SAMPLE                       ERRORS =  0 PAGE 2

RECORD
NUMBER

   25      ROM 512 BY 8;
   26   /* SPECIFY ROM 1 */  WORDS Ø to 511
   27      BITS  (CPE(6), CPE(5), CPE(4), CPE(3), CPE(2), CPE(1),
   28              CPE(0), KB(Ø)');
ROM     1
          (ØØØH)  Ø1ØØ ØØØ1   1ØØ1 11Ø1   1ØØØ Ø1Ø1   1ØØ1 Ø1Ø1
          (ØØ4H)  Ø1ØØ ØØØØ   ØØØØ Ø1ØØ   1ØØØ Ø1Ø1   Ø1ØØ ØØØ1
                                       .
                                       .
                                       .
          (1FCH)  1ØØ1 Ø1Ø1   11ØØ/Ø1ØØ   Ø1ØØ ØØØØ   1ØØ1 11ØØ
   29   /* SPECIFY ROM 2 */  WORDS Ø TO 511
   30      BITS (JUMP(6), JUMP(5), JUMP(4), JUMP(3), JUMP(2), JUMP(1),
                 JUMP(0), FO(Ø), FO(Ø));
ROM     2
          (ØØØH)  ØØ1Ø Ø1ØØ   Ø11Ø Ø1ØØ   Ø11Ø Ø1ØØ   1111 Ø111
          (ØØ4H)  1111 1ØØ1   1Ø1Ø 11Ø1   ØØØ1 1111   Ø11ØØØ1ØØ
                                       .
                                       .
                                       .
```

Fig. 8-1. Example of XMAP Source Statements and ROM Dumps

XMAP SOURCE RECORDS

Each record from the XMAP source file, is written in the list file left justified in column 8. The records are copies to the list file exactly as they appear in the source file unless the line width of the list file is too small to accommodate the entire record. In this case, the remaining characters of the source record are written in the next line of the list file, again left justified in column 8.

RECORD NUMBERS

Each record in the source file, beginning with the first XMAP language statement, is assigned a sequential record number. This record number appears as a decimal integer in the list file in columns 1 through 5 of the line in which the corresponding record appears. Record numbers run from 1 through 32767_{10}; if more source records exist, the numbering simply starts again at 1.

ROM NUMBERS

Each ROM specified by an XMAP ROM mapping statement is assigned a unique ROM number for identification purposes. This ROM number is written in the list file in columns 1 through 8 of the first available line following the line containing the end of the corresponding XMAP ROM mapping statement. The ROM number is used in the XMAP program summary, which is discussed in the next subsection.

ROM DUMP

The binary dump for each ROM begins on the line following the line containing the ROM number. Each line of the dump starts in column 12 with a three digit hexadecimal address in parentheses. This address is followed by the contents of a set of contiguous locations beginning with the location given by the address. Each line of the dump displays the contents of a multiple of four locations, but as many locations are written as will fit within the line width of the list file without violating this constraint. The contents of each location are represented as a binary string separated into four bit fields (one field for a four-bit ROM and two fields for an eight-bit ROM).

SELECTION OF OPTIONS

Via the control language, the user may specify that he wants only the XMAP source statements or only the ROM dumps in the list file. The format of the output under either of these options is the same as described in the appropriate paragraph above. ROM numbers are included in the output in both cases.

ERROR MESSAGES

Any error messages will be output to the list file by XMAP. A summary of XMAP error messages is listed in Appendix H.

8.2 XMAP PROGRAM SUMMARY

The XMAP program summary lists for each bit in the microinstruction word the number of the ROM and the bit position within the ROM into which the bit in question has been mapped by the XMAP language program. An example of the format of an XMAP program summary is given in Figure 8-2.

BIT NAME

The format of a bit name is a field name followed by a bit number enclosed in parentheses. Within each field, the high order bit is given first. The fields are presented in the order in which they are output by XMAS: the CPE field is given first, followed by the FI field, the FO field, the JUMP field and finally the user-defined fields in the order in which they are defined in the XMAS program.

Each bit name is followed by a list of one or more items. Each item is a ROM number followed by a bit number enclosed in parentheses. Items within each list occur in the order of increasing ROM number. Bit number 0 refers to the least significant bit position of each ROM word.

```
XMAP VERS 1.0 XMAP SAMPLE OUTPUT

SUMMARY OF XMAP PROGRAM

BIT NAME        ROM(BIT POSITION)

CPE( 6)           1(1)
CPE( 5)           1(2)
CPE( 4)           1(3)
CPE( 3)           1(4)
CPE( 2)           1(5)
CPE( 1)           1(6)
CPE( 0)           1(7)
FI  ( 1)          3(1),4(1)
FI  ( 0)          3(2),4(2)
FO  ( 1)          3(3),4(3)
FO  ( 0)          2(0)
```

Fig. 8-2. Example of XMAP Program Summary

SECTION 9
CONTROL LANGUAGE

Both XMAS and XMAP include a control language interpreter. Like XMAS and XMAP language statements, control language records are input from the source file. All control language records must, however, precede the first XMAS or XMAP language statement.

USE OF THE CONTROL LANGUAGE

The control language is used to specify various operating parameters for XMAS and XMAP. The control language provides the mechanism for designating files, specifying I/O data record formats and selecting listing options.

Section 9.1 provides an introduction to the control language and describes variables used in the control language. Section 9.2 gives a formal description of control language syntax. The formal description uses the modified BNF notation explained in Appendix A.

9.1 CONTROL LANGUAGE OVERVIEW

RECORD FORMAT

Every control language record must have a dollar character ($) in the first active character position (the left margin of the source file is determined by the value of the control variable LEFT, which is discussed below). Otherwise, control language records are free-format.

CONTROL VARIABLES

The control language deals with a set of variables called "control variables." The control variables, their meanings and their Intel-released default initial values are summarized in Table 9-1. The control variables are discussed in more detail below. Some control variables apply to both XMAS and XMAP, others apply to XMAS only, and others apply to XMAP only (those that begin with ROM).

Via the control language, two kinds of functions can be performed on a control variable: its current value can be displayed, or it can be assigned a new value.

DISPLAY FUNCTION

A display list is used to designate the control variables to be displayed. A display list begins with the special word DISPLAY followed by a list of control variables enclosed in a single set of parentheses. For example, the control record:

EXAMPLE

```
$DISPLAY (LEFT RIGHT)
```

instructs XMAS or XMAP to output the current values of the control variables LEFT and RIGHT to the list file. The display list may also take the form:

```
$ DISPLAY (ALL)
```

which instructs XMAS or XMAP to output the values of all the applicable control variables to the list file. (XMAS will not output the value of, say, ROMFILE since this control variable does not apply to XMAS.)

Typically, the display function is used in interactive mode (i.e., when both the source file and the list file are assigned to a terminal device). The display function provides a convenient means of allowing the user to determine which control variables he may wish to change.

SET FUNCTION

A control variable is assigned a new value when the control variable name appears in control language record followed by an assignment operator (=) and an integer. For example, the control language record:

EXAMPLE

```
$ LEFT = 4  RIGHT = 72
```

assigns the value of 4 to LEFT and 72 to RIGHT. A subset of the control variables, called the binary control variables, have only two significant values: zero or non-zero. The binary control variables all have an initial value of zero. An assignment operator and integer are not required to assign a binary control variable a non-zero value. For example, the control record:

```
$PRINT  BITS
```

TABLE 9-1. SUMMARY OF CONTROL VARIABLES

VARIABLE	MEANING	INITIAL*
BITS	If non-zero, XMAS displays bit patterns in the list file.	0
CODE	If non-zero, XMAP outputs hexadecimal code to the ROM programming file. If zero, XMAP outputs BNPF code.	0
CROSSREF	If non-zero, XMAS outputs a cross reference directory.	0
FORMS	If non-zero, page ejects are issued between pages in the list file. If zero, 6 line feeds are issued between pages.	0
IMAGE	If non-zero, XMAS outputs a microprogram memory image to the list file.	0
LEFT	Value is the columnar location of the left margin in the source file. All characters to the left of LEFT are ignored.	1
LINES	Value is the number of lines per page in the list file.	60
LISTFILE	Values designates the FORTRAN data file which is currently the list file.	
MICROMEMORY	Value specifies the microprogram memory size.	512
PRINT	If non-zero, source file records are echoed on the list file.	0
RIGHT	Value is the columnar location of the right margin of the source file. All characters to the right of RIGHT are ignored.	72
ROMDUMP	If non-zero, XMAP will output ROM dumps to the list file.	0
ROMFILE	Value designates the FORTRAN data file which is currently the ROM mapping file.	
ROMSUMMARY	If non-zero, XMAP will output an XMAP program summary to the list file.	0
SOURCEFILE	Value designates the FORTRAN data file currently the source file.	
TITLE	The character string assigned to TITLE is put in the page header of the list file.	null
WIDTH	Value gives the maximum number of characters in a list file record.	132

*These represent the default values that have been assigned by Intel. These values can be changed, however, at installation time.

assigns a non-zero (i.e., true) value to the control variable PRINT and BITS. The value of the control variable TITLE is a character string (initially null). The control language record:

```
$ TITLE = 'ANY CHARACTERS'
```

associates the character string ANY CHARACTERS with the control variable TITLE.

CONTROL LANGUAGE ERRORS

Exceptional conditions that cause an error message to be issued include finding an unknown control variable name, use of the binary control variable set feature on a control variable that is not two – valued (i.e., not "on" or "off"), and illegal syntax. After issuing an error message to the list file, the values of all control language variables are output to the list file and processing will terminate unless the source file is currently assigned to an interactive device (i.e., a terminal device).

In both XMAS and XMAP, all control variables that require changing must be assigned their desired value before the first XMAS or XMAP language statements. The control variables are described in the following paragraphs.

FILE DESIGNATION

The control variables SOURCEFILE, LISTFILE and ROMFILE designate the file numbers of the FORTRAN data files currently serving as the source file, list file and ROM programming file (XMAP only), respectively. File numbers are related to physical files via tables in XMAS and XMAP. These file tables are configured by the user when he installs XMAS and XMAP on his system. He also assigns appropriate initial values (i.e., file numbers) to SOURCEFILE, LISTFILE and ROMFILE. By convention, file number 1 is assumed to be a terminal file (e.g., a teletype or CRT); all other file numbers are assumed to be batch files (e.g., a card reader, card image disk file, etc.)

```
$ LISTFILE = 3   SOURCEFILE = 2
```

List file output is to be directed to file 3. Subsequent control language records and XMAS or XMAP language statements are to be input from file 2.

SOURCE FILE FORMAT CONTROL

The control variables LEFT and RIGHT specify the left and right margins of source file record. Characters to the left of LEFT or to the right of RIGHT are ignored. LEFT and RIGHT have initial values of 1 and 72 respectively.

```
$ LEFT = 5   RIGHT = 71
```

Subsequent source file records will be scanned beginning with character position 5 and ending with character position 71. Subsequent control language records, if any, must have a $ in character position 5.

LIST FILE FORMAT CONTROL

WIDTH, LINES AND FORMS CONTROL VARIABLES

The control variables WIDTH, LINES and FORMS control list file format. The value of width gives the maximum number of characters that may appear in a list file record, not including FORTRAN carriage control characters. WIDTH has an initial value of 132 and may not be assigned a value greater than 132 nor less than 72. The value of LINES is the number of lines per page in the list file; the initial value for LINES is 60. If FORMS equals zero (its initial value) six line feeds are issued when the line count reaches LINES. If FORMS is not equal to 0, a page eject is issued when the line count reaches LINES.

```
$ WIDTH = 80   FORMS   LINES = 40
```

List file records are to contain a maximum of 80 characters. A form feed is to be issued after every 40 lines are output to the list file.

TITLE CONTROL VARIABLE

The control variable TITLE is used to designate a list file page title. The character string assigned to TITLE is printed as the first line of every page output to the list file by XMAS or XMAP,

```
$ TITLE 'BIPOLAR MICROPROGRAM NOV. 15'
```

The text string BIPOLAR MICROPROGRAM NOV. 15 is printed at the top of every page in the list file.

XMAS LISTING OPTIONS

The binary control variables PRINT, BITS, CROSSREF and IMAGE are used to select XMAS listing options (see Section 3). These variables have initial values of zero meaning that the corresponding listing options are initially inhibited. If PRINT is non-zero, source file records are echoed on the list file with sequential record numbers. Control language records will not be numbered. (PRINT applies to both XMAS and XMAP.) If BITS is non-zero, the assembled bit patterns are displayed in the list file. If CROSSREF is non-zero, XMAS will output a cross reference directory to the list file. If IMAGE is non-zero XMAS will output a graphic microprogram memory image to the list file.

```
$PRINT  BITS  IMAGE
```

Source statement and bit patterns are to be output to the list file during the subsequent assembly. Also, a microprogram memory image is to be output to the list file following the subsequent assembly.

XMAP LISTING OPTIONS

The binary control variables ROMDUMP and ROMSUMMARY are used to select XMAP listing options (see Section 8). These variables have an initial value of zero meaning that the corresponding listing option is initially inhibited. If ROMDUMP is non-zero, XMAP outputs a hexadecimal dump for each XMAP mapping statement. If ROMSUMMARY is non-zero, XMAP outputs to the list file an XMAP program summary.

```
$ ROMDUMP
```

A ROM dump is to be output to the list file.

ROM PROGRAMMING FILE FORMAT

CODE, CONTROL VARIABLE

The binary control variable CODE is used to designate the data format of the ROM programming file. If CODE is zero, XMAP outputs BNPF code (refer to Appendix E) to the ROM programming file. If CODE is non-zero, XMAP outputs hexadecimal code to the ROM programming file.

MICROPROGRAM ADDRESS SPACE

The control variable MICROMEMORY allows XMAS and XMAP to perform microprogram address checking. MICROMEMORY has an initial value of 512. If XMAS or XMAP detects a microprogram address greater than or equal to MICROMEMORY, XMAS or XMAP outputs an error message to the list file.

```
$ MICROMEMORY = 2048
```

XMAS or XMAP will allow microprogram addresses up to 2047.

9.2 CONTROL LANGUAGE SYNTAX

SYNTAX

The following rules govern control language syntax:

$$\langle\text{control record}\rangle ::= \$ \left\{ \begin{matrix} \langle\text{display list}\rangle \\ \langle\text{set list}\rangle \end{matrix} \right\} \ldots$$

$$\langle\text{display list}\rangle ::= \text{DISPLAY} \ (\left\{ \begin{matrix} \text{ALL} \\ \langle\text{control variable}\rangle \end{matrix} \right\} \ldots)$$

$$\langle\text{set list}\rangle :: \left\{ \langle\text{control variable}\rangle \left[\begin{matrix} \langle\text{integer}\rangle \\ \langle\text{character string}\rangle \end{matrix} \right] \right\} \ .$$

```
⟨control variable⟩:: BITS!
                      CODE!
                      CROSSREF!
                      FORMS!
                      IMAGE!
                      LEFT!
                      LINES!
                      LISTFILE!
```

```
MICROMEMORY !
PRINT !
RIGHT !
ROMDUMP !
ROMFILE !
ROMSUMMARY !
SOURCEFILE !
TITLE !
WIDTH !
```

Control language records are free-format except that each must begin with a $ character, which must appear in the column specified by the control variable LEFT

EXAMPLES

```
$ LINES = 59  DISPLAY  (FORMS  IMAGE)
$ PRINT  DISPLAY  (ALL)
$ DISPLAY (LEFT, RIGHT)  DISPLAY (PRINT,
$ TITLE, WIDTH)
$ TITLE = 'BIPOLAR MICROPROGRAM, NOV. 15'
```

All control language records must precede the first XMAS or XMAP language statement. This means that all control variables must be set to their desired value before the first XMAS statement is encountered. The control variables are summarized in Table 9-1.

APPENDIX A
MODIFIED BNF SYNTAX DESCRIPTION NOTATION

The syntax of a language is its structure, or the form and order in which its elements may appear. The BNF syntax description notation, without modification, is essentially a set of rules, each of which defines how a particular syntactic entity is to be constructed. There are two different classes of syntactic entities in BNF; one class is referred to as terminal symbols, and the other as nonterminal symbols. Terminal symbols are those symbols which may appear in a program written in the language that the BNF describes. Examples of these are reserved words and identifiers in ALGOL, the keywords of FORTRAN (DO, GO TO, etc.) and statement labels and numbers in most languages. In general, symbols which may appear in a user program written in a particular programming language are the set of terminal symbols for that language, and the grammer describing it. Nonterminal symbols are symbols which may not appear in a program, and are used in the BNF to build more complex structures in an understandable manner. Terminal symbols are represented in BNF by the symbols themselves, while nonterminal symbols are bracketed by ⟨⟩.

For example, the syntax for an expression in the XMAS language (in a considerably simplified form) is:

```
⟨expression⟩ :: = ⟨term⟩ ! ⟨expression⟩ + ⟨term⟩
⟨term⟩ :: = ⟨factor⟩ ! ⟨term⟩ * ⟨factor⟩
⟨factor⟩ :: = ⟨identifier⟩ ! ⟨integer⟩ ! ⟨expression⟩
⟨integer⟩ :: = 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9
⟨identifier⟩ :: = ⟨letter⟩ ! ⟨identifier⟩ ⟨letter⟩
⟨letter⟩ :: = A ! B ! C ! D ! E !
```

In BNF notation, the vertical bar is used to separate alternatives. Logically, the vertical bar is an exclusive -OR operator. For ::, the words "is defined as" or "has an instance as" or "may be written as" should be substituted. Thus

```
⟨letter⟩ :: = A ! B ! C ! D ! E
```

would be read as

"the nonterminal symbol ⟨letter⟩ may be written as any of the terminal symbols: A or B or C or D or E"

Hence whenever ⟨letter⟩ is found on the right side of another BNF rule, it really represents A or B or C or D or E. It is, as one may observe, much more convenient to refer to ⟨letter⟩ than to say or write the terminals corresponding to it.

To show how these rules are used, take the rule:

```
⟨identifier⟩ :: = ⟨letter⟩ ! ⟨identifier⟩ ⟨letter⟩
```

A valid identifier generated from this rule is:

```
A
```

since any ⟨letter⟩ is a valid ⟨identifier⟩. In addition, now that we have an ⟨identifier⟩ like A, the second alternative in the BNF rule shows that AA is a valid ⟨identifier⟩. In fact, one may recursively expand the rule, using the second alternative:

```
⟨identifier⟩ :: =  ⟨identifier⟩ ⟨letter⟩
```

and since the ⟨identifier⟩ on the right hand side of the rule may be replaced by an ⟨identifier⟩ ⟨letter⟩, one can generate

```
⟨identifier⟩   = ⟨identifier⟩ ⟨letter⟩ ⟨letter⟩
```

This recursive process may be carried out to generate arbitrary long ⟨identifier⟩s. It is terminated by using the first alternative to the rule to replace ⟨identifier⟩:

```
⟨identifier⟩ :: = ⟨identifier⟩ ⟨letter⟩ ⟨letter⟩
                          •
                          •
                          •
                  ⟨identifier⟩ ⟨letter⟩ . . . ⟨letter⟩
```

where the . . . represents an arbitrary number of ⟨letter⟩s. Finally by substitution of ⟨letter⟩ for ⟨identifier⟩, we get

```
⟨identifier⟩ :: = ⟨letter⟩ ⟨letter⟩ . . . ⟨letter⟩
```

Some valid identifiers are:

```
A
AA
ABC ,
CBAD
```

As noted, these are not all the valid identifier, since any arbitrary combination of ⟨letter⟩s generates a valid ⟨identifier⟩. An invalid ⟨identifier⟩ (can't be generated from the BNF rule describing ⟨identifier⟩) is:

```
3ABC
```

since an ⟨identifier⟩ cannot contain an ⟨integer⟩.

We have in fact generated the first modification to BNF notation; the rule describing ⟨identifier⟩s may be written

```
⟨identifier⟩ :: ⟨letter⟩ . . .
```

where the . . . represents an arbitrary number (greater than zero) of occurrences of the entity preceeding it. This is much easier to write and to understand than the recursive rule it replaces.

Another useful addition to BNF notation is a bracketing tool, {} may be used to bracket syntactic entities and constructs to form a single entity. The curly brackets {} are used to indicate that what is enclosed in them is both a requirement when using the rule and a single compound entity. For example, the rules:

```
⟨a⟩ :: = {<b><c>} . . .
⟨b⟩ :: = B
⟨c⟩ :: = C
```

says that an ⟨a⟩ can be an arbitrary number of repititions of BC, such as

```
BC
BCBC
BCBCBC
```

ad infinitum. In other words, the ellipses, . . ., indicating arbitrary repetitions are now associated with ⟨b⟩ ⟨c⟩ as an entity, rather than ⟨c⟩. In addition, if entities are stacked vertically within the brackets, this indicates that one and only one of these stacked alternatives should be chosen. For example, the rule:

$$\langle a\rangle :: = A \begin{Bmatrix} B \\ C \end{Bmatrix}$$

may be written:

```
⟨a⟩ :: = AB ! AC
```

and says, an A must be followed by either a B or a C. The rule

$$\langle a \rangle ::= A \begin{Bmatrix} B \\ C \end{Bmatrix} \ldots$$

generates:

AB
AC
ABC

and in general, an A followed by an arbitrary number of Bs and Cs. The unmodified BNF equivalent of this is considerably more obscure:

⟨a⟩ :: = A ⟨b or c⟩

⟨b or c⟩ :: = B ! C ! ⟨b or c⟩ B ! ⟨b or c⟩ C

This necessarily recursive form is considerably more difficult to conceptualize than the modified BNF form of the rule given above.

The final addition to BNF allowing more abbreviated and less obscure notation is the brackets which indicate an optional entity. A syntactic entity or construct which is enclosed in square [ ] may optionally be chosen when the rule is used. For example, the rule:

⟨a⟩ :: = A [B]

is the same as the rule:

⟨a⟩ :: = A ! AB

and indicates that the A may or may not be followed by a B. It has the similar property to {} in that items enclosed in [ ] become a single entity. The rule

⟨a⟩ :: = A [B C] . . .

is equivalent to

⟨a⟩ :: = A ! A ⟨bc list⟩

⟨bc list⟩ :: = B C ! ⟨bc list⟩ B C

and is much easier to understand and use. This rule says that an ⟨a⟩ may be written as an A, followed by zero or more BCs. When items are stacked vertically between [ ], this indicates that one and only one of the items may be chosen. The rule

$$\langle a \rangle ::= A \begin{bmatrix} B \\ C \end{bmatrix}$$

is equivalent to

⟨a⟩ :: = A ! AB ! AC

But, again, is considerably more compact.

The final caution concerns the interaction between syntax and semantics. The modified BNF rule:

$$\langle\text{field spec}\rangle ::= \text{FIELD} \left\{ \begin{Bmatrix}\text{LENGTH}\\ \text{DEFAULT}\end{Bmatrix} = \langle\text{expression}\rangle \right\} \ldots$$

indicates that

```
FIELD LENGTH = 1   LENGTH = 2   DEFAULT = 0   LENGTH = 1
```

is valid. While it is valid syntactically, its meaning, or semantics, is questionable. In such a case, the description of the semantics for this construct would probably say that the parameters LENGTH and DEFAULT may be specified only once each.

APPENDIX B
SYNTAX SUMMARY FOR THE XMAS LANGUAGE

The purpose of this appendix is to provide a quick reference to the syntax of the XMAS language. The descriptive meta-notational is explained in Appendix A.

```
⟨program⟩ :: = [ ⟨declaration⟩ ]
                ⟨specification part⟩ EOF

⟨declaration part⟩ :: =  {⟨declaration statement⟩ } ; . . .

⟨declaration statement⟩ :: = ⟨string statement⟩ !
                              ⟨value statement⟩ !
                              ⟨field statement⟩ !
                              ⟨imply statement⟩ !
                              ⟨k-bus statement⟩ !
                              ⟨address statement⟩ !

⟨specification part⟩ :: =  {⟨specification statement⟩}  ; . . .

⟨string statement⟩ :: = ⟨string identifier⟩ STRING
            {⟨character string⟩} . . .

⟨string identifier⟩ :: = ⟨identifier⟩

⟨value statement⟩ :: = ⟨value identifier⟩ VALUE ⟨expression⟩

⟨value identifier⟩ :: = ⟨identifier⟩

⟨field statement⟩ :: = ⟨field name⟩ FIELD ⟨field spec⟩ . . .

⟨field name⟩ :: = ⟨identifier⟩

⟨field spec⟩ :: = LENGTH = ⟨integer⟩ ! DEFAULT = ⟨expression⟩

MICROPS (  {⟨microp⟩ = ⟨expression⟩} . . . )

⟨imply statement⟩ :: = ⟨microp⟩ IMPLY ⟨imply list⟩

⟨imply list⟩ :: =  {⟨field name⟩ = ⟨expression⟩ } . . .

⟨k-bus statement⟩ :: = ⟨field name⟩ KBUS

⟨address statement⟩ :: = ⟨field name⟩ ADDRESS

⟨specification statement⟩ :: = ⟨label part⟩ {⟨fields part⟩ }  . . .

⟨label part⟩ :: = [*] ⟨integer⟩: [⟨address identifier⟩:] . . .

⟨address identifier⟩ :: = ⟨identifier⟩

⟨fields part⟩ :: = ⎧ ⟨field name⟩ = ⟨expression⟩
                   ⎪ ⟨microp⟩
                   ⎪ ⟨CPE microp⟩     ⎧ ⟨register name⟩    ⎫
                   ⎩ ⟨JUMP microp⟩    ⎩ {⟨expression⟩} . . . ⎭ . . .

⟨comment⟩ :: = /* ⟨character string excluding */⟩*/

⟨expression⟩ :: =<term>  ⎡⎧ +  ⎫          ⎤
                         ⎢⎨ OR ⎬  ⟨TERM⟩  ⎥ . . .
                         ⎣⎩ XOR⎭          ⎦

⟨term⟩ :: = ⟨subterm⟩ [AND ⟨subterm⟩] . . .

⟨subterm⟩ :: = ⟨factor⟩  ⎡⎧ SHL ⎫ ⟨Factor⟩⎤ . . .
                         ⎣⎩ SHR ⎭         ⎦

 factor    = [NOT] primary
```

```
⟨primary⟩ :: = {⟨value identifier⟩
                ⟨address identifier⟩
                ⟨integer⟩!
               (⟨expression⟩)

⟨integer⟩ :: =  ⟨decimal integer⟩ !
                ⟨binary integer⟩ !
                ⟨octal integer⟩ !
                ⟨hexadecimal integer⟩

⟨decimal integer⟩ :: = ⟨decimal digit⟩} . . . [D]

⟨binary integer⟩ :: =  {⟨binary digit⟩} . . . B

⟨octal integer⟩ :: =  {⟨octal digit⟩} . . . Q
                                            Q

⟨hexadecimal integer⟩ :: = ⟨decimal digit⟩ [⟨hexadecimal digit⟩] . . . H

⟨decimal digit⟩ :: = 0!1!2!3!4!5!6!7!8!9
                     0

⟨binary digit⟩ :: = 0!1

⟨octal digit⟩ :: = 0! 1!2!3!4!5!6!7

⟨hexadecimal digit⟩ :: = 0!1!2!3!4!5!6!7!
                         8!9!A!B!C!D!E!F

⟨identifier⟩ :: = ⟨letter⟩ [⟨letter⟩] . . .
                           [⟨digit⟩ ]

⟨letter⟩ :: = A!B!C!D!E!F!G!H!I!J!K!L!M!
              N!O!P!Q!Q!R!S!T!U!V!W!X!Y!Z

⟨digit⟩ :: = 0!1!2!3!4!5!6!7!8!9
```

APPENDIX C
SYNTAX SUMMARY FOR THE XMAP LANGUAGE

The purpose of this appendix is to provide a quick reference to the syntax of the XMAP language. The descriptive meta-notation is explained in Appendix A.

```
⟨program⟩ :: = ⎧⟨ROM spec⟩;    ⎫ . . . EOF
               ⎩⟨mapping spec⟩;⎭

⟨ROM spec⟩ :: = ROM ⟨integer1⟩ BY ⟨integer2⟩

⟨integer1⟩ :: = ⟨integer⟩

⟨integer2⟩ :: = ⟨integer⟩

⟨mapping spec⟩ :: = WORDS  ⟨range pair⟩
        [INVERT] BITS    (⟨bit spec list⟩)

⟨range pair⟩ :: = ⟨integer3⟩ TO ⟨integer4⟩

⟨bit spec list⟩ :: = ⎧⟨field name⟩ (⟨integer5⟩) [']⎫
                     ⎨0                            ⎬ . . .
                     ⎩1                            ⎭

⟨integer3⟩ :: = ⟨integer⟩

⟨integer4⟩ :: = ⟨integer⟩

⟨integer5⟩ :: = ⟨integer⟩
```

APPENDIX D
SYNTAX SUMMARY FOR THE CONTROL LANGUAGE

```
                      $ (⟨display list⟩)
⟨control record⟩ :: =  {              } . . .
                        (⟨set list⟩    )

⟨display list⟩ :: = DISPLAY  / { ALL                 } \
                            (  {                     }  ) . .
                             \ {⟨control variables⟩  } /

⟨set list⟩ :: =  {⟨control variables⟩  [ = ⟨integer⟩           ]}    . . .
                 {                      [ = ⟨character string⟩  ]}

⟨control variable⟩ :: = BITS !
                        CODE !
                        CROSSREF !
                        FORMS !
                        IMAGE !
                        LEFT !
                        LINES !
                        LISTFILE !
                        MICROMEMORY !
                        PRINT !
                        RIGHT !
                        ROMDUMP !
                        ROMFILE !
                        ROMSUMMARY !
                        SOURCEFILE !
                        TITLE !
                        WIDTH !
```

APPENDIX E
RESERVED WORD SUMMARY

	DECLARATIONS			MICROPS							
PROGRAM TERMINATOR	STATEMENT TYPE	FIELD STATEMENT KEYWORDS	INTRINSIC FIELD NAMES	FIELD						CPE REGISTER NAMES	OPERATORS
				JUMP	FI	FO	CPE				
EOF	FIELD	LENGTH	JUMP	JCC	HCZ	FFC	ACM	INR		AC	AND
		DEFAULT	FI	JCE	SCZ	FFZ	ADR	LCM		RØ	NOT
		MICROPS	FO	JCF	STC	FFØ	AIA	LDI		R1	OR
	IMPLY		CPE	JCR	STZ	FF1	ALR	LDM		R2	XOR
	KBUS			JFL			AMA	LMF		R3	SHL
	ADDRESS			JLL			ANI	LMI		R4	SHR
	VALUE			JMP			ANM	LMM		R5	
	STRING			JPR			ANR	LTM		R6	
				JPX			CIA	ORI		R7	
				JRL			CLA	ORM		R8	
				JZF			CLR	ORR		R9	
				JZR			CMA	SDA		T	
							CMR	SDR			
							CSA	SRA			
							CSR	TZA			
							DCA	TZR			
							DSM	XNI			
							ILR	XNM			
							INA	XNR			

APPENDIX F
BNPF AND HEXADECIMAL CODE

3NPF format is an ASCII representation of a byte in pure binary form. A "B" is punched to indicate the beginning of a byte. ollowing the B, exactly four or eight P's and N's must be punched, depending on the ROM word width and the proramming device. The fifth or ninth character following the B must be an F, indicating the end of a byte. All characters ollowing the F are ignored until another B is encountered. This allows comments (not containing the letter B) to appear etween bytes of data in BNPF format.

3its of data in a BNPF byte appear in left-to-right order from most significant to least significant.

Example: The two bytes '3AF0' would be represented in eight bit BNPF format as

BNNPPPNPNF BPPPPNNNNF

A hexadecimal object file is an ASCII representation of program memory, expressed as a series of hexadecimal digits. These are blocked into records, each of which contains the record length, type, memory load address, and checksum, in addition to data. The description below applies to paper tape on a frame-by-frame basis.

Frame 0 = Record Mark
The ASCII representation of a colon (3A in base 16) is used to signal the start of a record.

Frames 1, 2 = Record length in hexadecimal
This is the count of the actual data bytes in the record. A record length of zero indicates end of file.

Frames 3-6 = Load Address
The four-character starting address at which the following data will be loaded. The first data byte is stored in the location ndicated by the load address. Successive data bytes are stored in successive memory locations.

rames 7, 8 = Record type
urrently, all records are type 0. This field is reserved for future expansion.

rames 9 to 9+2* (record length) = 1 = Data
ch 8-bit memory word is represented by two frames containing ASCII characters 0-9, A-F, which represent a hexadecimal lue between 0 and FF (0 and 255 decimal).

mes 9+2* (record length) to 9+2* (record length)+1 = Checksum
e Checksum is the negative of the sum of all 8-bit bytes in the record, evaluated modulo 256. The sum of all bytes in the ord (including the checksum) should be zero.

e following is an example of hexadecimal object format:

```
:10000000F0D3921COCF1A3FB1216B2COD1F19A1CD2
:0800100016D0F1831A0AFACOBO
:00
```

seful reference on the subject of PROM/ROM programming is:

Intel® Data Catalog
October 1973
pps. 3-31 to 3-34

APPENDIX G
SUMMARY OF XMAS ERROR MESSAGES

1:	PARSE STACK OVERFLOW
2:	SORT STACK OVERFLOW
3:	NAME/STRING STORAGE OVERFLOW
4:	HASH TABLE OVERFLOW
5:	TOO MANY LABELS FOR CROSS REFERENCE DIRECTORY
6:	VALUE SPACE OVERFLOW
7:	SYMBOL TABLE OVERFLOW
8:	FIELD TABLE OVERFLOW
9:	PARSE STACK UNDERFLOW
101:	INVALID CONTROL VARIABLE
102:	IDENTIFIER IS NOT A BINARY CONTROL VARIABLE
103:	MISSING VALUE FOR CONTROL VARIABLE
104:	ATTEMPT TO ASSIGN INVALID VALUE TO CONTROL VARIABLE
105:	ATTEMPT TO ASSIGN OUT OF BOUNDS VALUE TO CONTROL VARIABLE
106:	MISSING) TO TERMINATE DISPLAY LIST
107:	ATTEMPT TO ASSIGN VALUE OF IMPROPER TYPE TO CONTROL VARIABLE
108:	EMPTY DISPLAY LIST
109:	MISSING ' TO TERMINATE TITLE
110:	INVALID CHARACTER
111:	INVALID INTEGER
112:	CANNOT ENTER XMAS STATEMENTS FROM TERMINAL
200:	ILLEGAL NUMBER, 1 ASSUMED
201:	ILLEGAL USE OF /
202:	UNEXPECTED EOF
203:	ILLEGAL USE OF $ OR *
300:	FIELD IS MULTIPLY IMPLIED
301:	FIELD NOT SPECIFIED
302:	NO JUMP SPECIFIED IN FINAL STATEMENT
303:	WRONG NUMBER OF ITEMS IN JUMP LIST
304:	VALUE FOR FIELD WAS TRUNCATED
305:	MICROP ALREADY HAS IMPLY LIST
306:	IDENTIFIER IS NOT A MICROP
307:	UNDECLARED IDENTIFIER
308:	IDENTIFIER IS NOT A FIELD NAME
309:	INTRINSIC FIELD MAY NOT BE USED AS KBUS FIELD
310:	MULTIPLE OCCURRENCE OF KBUS STATEMENT
311:	INTRINSIC FIELD MAY NOT BE USED AS ADDRESS FIELD
312:	MULTIPLE OCCURRENCE OF ADDRESS STATEMENT
313:	MULTIPLY DEFINED IDENTIFIER
314:	MICROP VALUE TRUNCATED TO FIELD LENGTH
315:	MULTIPLE MICROP LISTS FOR FIELD
316:	ZERO FIELD LENGTH IS ILLEGAL
317:	MULTIPLE LENGTHS FOR FIELD
318:	MULTIPLE DEFAULTS FOR FIELD
319:	DEFAULT VALUE WAS TRUNCATED TO FIELD LENGTH
320:	JUMP IS IMPOSSIBLE
321:	ADDRESS IS MULTIPLY DEFINED
322:	ADDRESS OF STATEMENT IS OUT OF RANGE
323:	JUMP FROM PREVIOUS STATEMENT IS IMPOSSIBLE
324:	IDENTIFIER IS NOT A LABEL
325:	FIELD IS MULTIPLY SPECIFIED

326: MICROP IGNORED FOR INVALID FIELD
327: INVALID USE OF IDENTIFIER AS MICROP
328: ATTEMPT TO SPECIFY INVALID FIELD
329: INVALID USE OF IDENTIFIER AS FIELD NAME
330: JUMP LIST IS EMPTY
331: INVALID USE OF IDENTIFIER AS JUMP MICROP
332: ILLEGAL REGISTER FOR MICROP, AC ASSUMED
333: INVALID USE OF IDENTIFIER AS CPE MICROP
334: ADDRESS IN JUMP IS OUT OF RANGE
335: ADDRESS OUT OF SEQUENCE IN JUMP
336: VALUE IDENTIFIER USED BEFORE DEFINITION
337: INVALID USE OF IDENTIFIER IN EXPRESSION, 0 ASSUMED
338: MULTIPLE USE OF FIELD IN IMPLY LIST
339: STRING NAME ENCOUNTERED IN STRING EXPANSION, NULL ID INSERTED
340: FIELD CAUSES MAXIMUM INSTRUCTION LENGTH TO BE EXCEEDED
341: VALUE OF LABEL IN EXPRESSION IS UNDEFINED
500: SYNTAX ERROR

APPENDIX H
SUMMARY OF XMAP ERROR MESSAGES

1: PARSE STACK UNDERFLOW
2: PARSE STACK OVERFLOW
3: MULTIPLY DEFINED SYMBOL
4: SYMBOL TABLE OVERFLOW
5: NAME SPACE OVERFLOW
6: ROM SUMMARY TABLE OVERFLOW
101: INVALID CONTROL VARIABLE
102: IDENTIFIER IS NOT BINARY CONTROL VARIABLE
103: MISSING VALUE FOR CONTROL VARIABLE
104: ATTEMPT TO ASSIGN INVALID VALUE TO CONTROL VARIABLE
105: ATTEMPT TO ASSIGN OUT OF BOUNDS VALUE TO CONTROL VARIABLE
106: MISSING) TO TERMINATE DISPLAY LIST
107: ATTEMPT TO ASSIGN VALUE OF IMPROPER TYPE TO CONTROL VARIABLE
108: EMPTY DISPLAY LIST
109: MISSING ' TO TERMINATE TITLE
110: INVALID CHARACTER
111: INVALID INTEGER
112: CANNOT ENTER XMAP STATEMENTS FROM A TERMINAL
113: SYMBOL TABLE OVERFLOW
200: INVALID USE OF /
201: UNEXPECTED EOF
202: INVALID USE OF *
203: INVALID CHARACTER
[illegible]00: INVALID INTEGER
[illegible]01: INVALID ROM LENGTH
[illegible]02: INVALID ROM WIDTH
[illegible]03: BIT SPECIFIED OUTSIDE OF FIELD
[illegible]04: INVALID INTEGER BIT
[illegible]05: NUMBER OF BITS PROGRAMMED EXCEEDS ROM WIDTH
[illegible]06: INVALID MICROPROGRAM MEMORY ADDRESS
[illegible]07: ROM LENGTH AND RANGE PAIR MISMATCH
[illegible]08: HEX ROM CODE REQUIRES AN 8-BIT ROM
[illegible]09: NUMBER OF BITS PROGRAMMED IS LESS THAN ROM WIDTH
[illegible]10: ROM SIZE SPECIFICATION MUST PRECEDE MAPPING SPECIFICATION
[illegible]00: SYNTAX ERROR

APPENDIX I
SIMPLIFIED LANGUAGE DESCRIPTION

XMAS Definition

Notation

Upper Case Word	Specific keyword, must be used as shown.
Lower Case Word	General class of elements.
Brackets	Optional element, if used select one of the items stacked vertically.
Braces	Require element, select one of the items stacked vertically.
Three dots . . .	Immediately preceeding item may occur one or more times.

Examples of Notation in Use

The notation:

```
  Identifier VALUE expression
```

means the word VALUE must be written as shown. The name of any legal identifier may be substituted where the word "identifier" appears as may any legal expression appear where the word "expression" appears. Such as:

```
  MSBMASK VALUE 32768
  BIT4MSK VALUE 2x6
```

The notation:

$$\begin{Bmatrix}\text{BIT}\\ \text{BYTE}\end{Bmatrix}\begin{bmatrix}\text{VALUE}\\ \text{STRING}\end{bmatrix}$$

fines any of the following forms:

```
 BIT
 BYTE
 BIT VALUE
 BIT STRING
 BYTE VALUE
 BYTE STRING
```

he notation:

```
 identifier   identifier . . .
```

fines the (infinite) sequences:

```
 identifier
 identifier, identifier
 identifier, identifier, identifier
 etc.
```

Declaration Statements

Declaration statements must precede all other statements

```
identifier KBUS
identifier ADDRESS;
identifier VALUE expression;
identifier FIELD   LENGTH integer    DEFAULT expression   MICROPS (  identifier=expression  . . .);
identifier IMPLY   identifier=expression   . . .;
identifier STRING  'character-string'  . . .;
```

Specification Statement

```
*[integer:]

                                         (   register-id       )
                                             expression . . .
  [label-identifier] . . .   identifier      *                      . . .

                                         =   expression
```

XMAP Statements

```
                                         field-name (integer) [']
WORDS integer TO integer BITS (          0                          . . .
                                         1

ROM integer BY integer;
```

Identifier

Letter [up-to-6-letters-or-digits]

Character String

any-character-except-quote

APPENDIX J
STRINGOP EXAMPLES

K-BUS FIELD DEFINITION

For these STRING statements to be valid, it is necessary that the programmer define a new field and identify that field as the K-bus field (via a FIELD statement and KBUS statement, respectively). It is also necessary that the programmer declare two microps for that field (in the defining FIELD statement), KZERO and KONES, to represent the all zero and all one bit patterns for the field. For example, the statements:

```
KB  FIELD  LENGTH = 8
    MICROPS (KZERO = 0   KONES = 0FFH
             KLOW = 0FH  KHI = 0F0H
             KVAL = 10);
KB  KBUS;
```

define the 8-bit field called KB and identify that field as the KBUS field. The FIELD statement also declares microps for the KB field. KZERO and KONES are required for the STRING statements given below. The other microps are arbitrarily chosen and are used in examples below.

STRINGOP NAMING CONVENTIONS

A functional summary of the stringops described below is given in Table 5-3. The first three characters of the stringop identify the CPE operation, as follows:

Letters	Operation
MOV	Source 1 → Destination
MAR	Source 1 → MAR
ADD	Source 1 + Source 2 → Destination
KAD	K + Source 1 → Destination
INC	Source 1 + 1 → Destination
DEC	Source 1 — 1 → Destination (see notational convention S, below)
AND	Source 1 ∧ Source 2 → Destination
IOR	Source 1 ∨ Source 2 → Destination
XNR	Source 1 $\overline{\oplus}$ Source 2 → Destination
MSK	K ∧ Source 1 → Destination
TST	K ∧ Source 1 → Carry Output (CO)
NOT	$\overline{\text{Source 1}}$ → Destination
NEG	$\overline{\text{Source 1}}$ + 1 → Destination
CLR	0 → Destination
SET	—1 → Destination (see notational convention 5, below)
SHR	Source 1 "right shifted" → Destination

The fourth character of the stringop identifies the source and destination as follows:

Letter	Source 1	Source 2	Destination
E	Rn	AC∧K	Rn, AC
R	Rn	AC∧K	Rn
A	AC∧K		Rn
T	AT		AT
M	M	AC∧K	AT
I	I∧K	AT	AT

The fifth character, when present, is used to identify a carry input (FO field) variation.

NOTATIONAL CONVENTIONS

The following notational conventions are used in the discussions below:

1. KZERO and KONES are assumed to be microps for the user defined K-bus field.

TABLE J-1. CPE STRINGOPS

STRINGOP		FUNCTION	CARRY OUT	K-BUS	CARRY I
MOVE (Rn)		Rn → AC	0 → CO	K = 0	CI = 0
MOVA (Rn)	[K]	AC [∧ K] → Rn	1 → CO	(K = –1)	CI = 1
MOVM (AT)	[K]	M [∧ K] → AT	1 → CO	(K = –1)	CI = 0
MOVI (AT)	[K]	I [∧ K] → AT	1 → CO	(K = –1)	CI = 1
MARR (Rn)		Rn → MAR	0 → CO	K = 0	CI = 0
MARR1 (Rn)		Rn → MAR Rn + 1 → Rn	a.c. → CO	K = 0	CI = 1
MARM (AT)		M → MAR M → AT	0 → CO	K = 0	CI = 0
ADDE (Rn)	[K]	(AC [∧ K]) + Rn → Rn, AC	a.c. → CO	(K = –1)	CI = 0
ADDEC (Rn)	[K]	(AC [∧ K] + Rn + C → Rn, AC	a.c. → CO	(K = –1)	FFC
ADDR (Rn)	[K]	(AC [∧ K]) + Rn → Rn	a.c. → CO	(K = –1)	CI = 0
ADDRC (Rn)	[K]	(AC [∧ K]) + Rn + C → Rn	a.c. → CO	(K = –1)	FFC
ADDM (AT)	[K]	(AC [∧ K]) + M → AT	a.c. → CO	(K = –1)	CI = 0
ADDI (AT)	[K]	I [∧ K] + AT → AT	a.c. → CO	(K = –1)	CI = 0
KADR (Rn)	[K]	Rn [∨ K] → MAR Rn [+K] → Rn	a.c. → CO	(K = 0)	CI = 0
KADM (AT)	[K]	M [∨ K] → MAR M [+K] → AT	a.c. → CO	(K = 0)	CI = 0
INCE (Rn)		Rn + 1 → Rn, AC	a.c. → CO	K = 0	CI = 1
INCR (Rn)		Rn + 1 → Rn	a.c. → CO	K = 0	CI = 1
INCM (AT)		M + 1 → AT	a.c. → CO	K = 0	CI = 1
DECA (Rn)		AC – 1 → Rn	a.c. → CO	K = –1	CI = 0
DECT (AT)		AT – 1 → AT	a.c. → CO	K = –1	CI = 0
DECI (AT)		I – 1 → AT	a.c. → CO	K = –1	CI = 0
DECR (Rn)		Rn – 1 → Rn –1 → MAR	a.c. → CO	K = –1	CI = 0
ANDR (Rn)	[K]	AC [∧ K] ∧ Rn → Rn	l.r. → CO	(K = –1)	CI = 0
ANDM (AT)	[K]	AC [∧ K] ∧ M → AT	l.r. → CO	(K = –1)	CI = 0
ANDI (AT)	[K]	I [∧ K] ∧ AT → AT	l.r. → CO	(K = –1)	CI = 0
IORR (Rn)	[K]	(AC [∧ K]) ∨ Rn → Rn	AC [∧ K] → CO	(K = –1)	CI = 0
IORM (AT)	[K]	(AC [∧ K]) ∨ M → AT	AC [∧ K] → CO	(K = –1)	CI = 0
IORI (AT)	[K]	(I [∧ K]) ∨ AT → AT	I [∧ K] → CO	(K = –1)	CI = 0
XNRR (Rn)	[K]	(AC [∧ K]) $\overline{\oplus}$ Rn → Rn	AC [∧ K] → CO	(K = –1)	CI = 0
XNRM (AT)	[K]	(AC [∧ K]) $\overline{\oplus}$ M → AT	AC [∧ K] → CO	(K = –1)	CI = 0
XNRI (AT)	[K]	(I [∧ K] $\overline{\oplus}$ AT → AT	I [∧ K] → CO	(K = –1)	CI = 0
MSKR (Rn)	[K]	Rn [∧ K] → Rn	l.r. → CO	(K = –1)	CI = 0
MSKM (AT)	[K]	M [∧ K] → AT	l.r. → CO	(K = –1)	CI = 0
TSTR (Rn)	[K]	Rn [∧ K] → Rn	l.r. → CO	(K = –1)	CI = 0
TSTM (AT)	[K]	M [∧ K] → AT	l.r. → CO	(K = –1)	CI = 0
TSTI (AT)	[K]	I [∧ K] ∨ AT → AT	I [∧ K] → CO	(K = –1)	CI = 0
NOTR (Rn)		Rn → Rn	0 → CO	K = 0	CI = 0
NOTM (AT)		M → AT	0 → CO	K = 0	CI = 0
NEGT (AT)		AT + 1 → AT	a.c. → CO	K = 0	CI = 1
CLRR (Rn)	[CI]	0 → Rn	CI → CO	K = 0	(CI = 0)
SETR (Rn)		–1 → Rn	0 → CO	K = 0	CI = 1
SHRT (AT)	[LI]	LI → AT_H AT_H → AT_L AT_L → RO	AT_L → RO	K = 0	(LI = 0)

2. The "format" part of the stringop description gives the general form in which the stringop may appear in a specification statement.
3. Optional items are enclosed in square brackets (e.g., [K] means that an explicit K-bus field assignment is optional and [ΛK] means that, if an explicit assignment is given, ΛK is included in the function.
4. The abbreviation a.c. means arithmetic carry, and l.r. means logical result.
5. The expression −1 refers to the two's complement addition of 111. . .11 (all ones) to perform a subtraction of 000. . .01.

The MOV stringops are used to specify the transfer of data from a source register or bus to a destination register. An explicit K-bus field assignment may accompany a MOVA, MOVM or MOVI stringop if it is desired to mask the information to be transferred.

Notice that the MOVE stringop (as well as many other stringops defined in this section) includes an explicit k-bus specification, even though that specification is the same as the default pattern implied by the ILR microp. Because the character string 'FF0 ILR' only specifies a moving function when the K-bus bit pattern is all zeros (see Table 5-1, F-group 0, R-group 1), it is important that the K-bus pattern be specified as all zeros to ensure that the MOVE stringop will always actually specify a moving function. While it is true that the CPE microp ILR implies an all zero K-bus default, this default would be overridden if an explicit Non-zero K-bus pattern were inadvertently specified; as a result a MOVE stringop with KZERO would specify a masked addition instead of a moving function. By including KZERO in the MOVE stringop, however, any inadvertent use of MOVE with an explicit non-zero k-bus specification will cause an error message to be output to the list file (see Appendix G), because only one assignment is allowed for each field in a specification statement. Thus, while KZERO may not be absolutely necessary, it does provide an extra safeguard when using the MOVE stringop. This same rational applies to the inclusion of explicit K-bus field assignments in many of the stringops defined here, even though these assignments are redundant.

MOVE DEFINITION	MOVE STRING 'KZERO FF0 ILR'
FORMAT	MOVE (Rn)
FUNCTION	Rn → AC 0 → CO
DESCRIPTION	The data in the specified register (Rn) is deposited in the accumulator (AC). The data in the specified register is not changed.
MOVA DEFINITION	MOVA STRING 'FF1 SDR'
FORMAT	MOVA (Rn) [K]
FUNCTION	AC [ΛK] → Rn 1 → CO
DESCRIPTION	If K is not specified, the data in the accumulator (AC) is deposited in the specified register (Rn). If K is specified, the data in the accumulator is ANDed with the data on the K-bus, and the result is deposited in the specified register.
MOVM DEFINITION	MOVM STRING 'FF0 LTM'
FORMAT	MOVM (AT) [K]
FUNCTION	M [ΛK] → AT 1 → CO
DESCRIPTION	If K is not specified, the data on the M-bus is deposited in the specified register (AC or T). If K is specified, the data on the M-bus is ANDed with the data on the K-bus, and the result is deposited in the specified register.

MOVI DEFINITION	MOVI STRING 'FFI LDI'
FORMAT	MOVI (AT) [K]
FUNCTION	I [∧K] → AT 1 → CO
DESCRIPTION	If K is not specified, the data on the I-bus is deposited in the specified register (AC or T). If K is specified, the data on the I-bus is ANDed with the data on the K-bus, and the result is deposited in the specified register.

EXAMPLES

```
MOVE  (R3)              /* MOVE R3 TO AC */
MOVA  (R3)              /* MOVE AC TO R3 */
MOVA  (R3)  KLOW        /* MOVE AC ANDED WITH KLOW TO R3 */
MOVM  (T)               /* MOVE M TO T */
MOVI  (AC)              /* MOVE I TO AC */
```

The MAR stringops are used to load the Memory Address Register (MAR) from a source register or bus. The MARR1 stringop increments the contents of the specified register after the initial contents have been transferred to the Memory Address Register. The MARM stringop transfers the data on the M-bus to both the Memory Address Register and the specified register.

MARR DEFINITION	MARR STRING 'KZERO FFO LMI'
FORMAT	MARR (Rn)
FUNCTION	Rn → MAR 0 → CO
DESCRIPTION	The data in the specified register (Rn) is deposited in the Memory Address Register (MAR).
MARR1 DEFINITION	MARR1 STRING 'KZERO FFI LMI'
FORMAT	MARR1 (Rn)
FUNCTION	Rn → MAR Rn + 1 → Rn a.c. → CO
DESCRIPTION	The data in the specified register (Rn) is deposited in the Memory Address Register (MAR); then, the data in the specified register is incremented.
MARM DEFINITION	MARM STRING 'KZERO FFO LMM'
FORMAT	MARM (AT)
FUNCTION	M → MAR M → AT 0 → CO
DESCRIPTION	The data on the M-bus is deposited in both the Memory Address Register (MAR) and the specified register (AC or T).

EXAMPLES

```
MARR (R7)               /* R7 GOES TO MAR */
MARRI (R7)              /* R7 GOES TO MAR AND R7 IS INCREMENTED */
MARM (AC)               /* M GOES TO MAR AND TO AC */
```

The ADD stringops are used to specify that data from two sources (either two registers or a register and a bus) is to be added arithmetically and the sum deposited in a destination register. An explicit K-bus field assignment may accompany an ADD stringop if it is desired to mask one of the source operands before the addition is performed.

ADDE DEFINITION	ADDE STRING 'FFO ALR'
FORMAT	ADDE (Rn) [K]

FUNCTION	(AC [∧K] + Rn → Rn, AC a.c. → CO
DESCRIPTION	If K is not specified, the data in the accumulator (AC) is added to the data in the specified register (Rn), and the sum is deposited in both the accumulator and the specified register. If K is specified, the data in the accumulator is ANDed with the data on the K-bus, the result is added to the data in the specified register, and the sum is deposited in both the accumulator and the specified register.
ADDEC DEFINITION	ADDEC STRING 'FFC ALR'
FORMAT	ADDEC (Rn) [K]
FUNCTION	(AC [∧K]) + Rn + C → Rn, AC a.c. → CO
DEFINITION	ADDEC is identical to ADDE except that the content of the C-flag in the MCU is included in the sum.
ADDR DEFINITION	ADDR STRING 'FF0 ADR'
FORMAT	ADDR (Rn) [K]
FUNCTION	(AC [∧K] + Rn → Rn a.c. → CO
DESCRIPTION	ADDR is identical to ADDE except that the sum is deposited only in the specified register (Rn). The data in the accumulator is not changed.
ADDRC DEFINITION	ADDRC STRING 'FFC ADR'
FORMAT	ADDRC (Rn) [K]
FUNCTION	(AC [∧K]) + Rn + C → Rn a.c. → CO
DESCRIPTION	ADDRC is identical to ADDR except that the content of the C-flag in the MCU is included in the sum.
ADDM DEFINITION	ADDM STRING 'FF0 AMA'
FORMAT	ADDM (AT) [K]
FUNCTION	(AC [∧K]) + M → AT a.c. → CO
DESCRIPTION	If K is not specified, the data in the accumulator (AC) is added to the data on the M-bus, and the sum is deposited in the specified register (AC or T). If K is specified, the data in the accumulator is ANDed with the data on the K-bus, the result is added to the data on the M-bus, and the sum is deposited in the specified register.
ADDI DEFINITION	ADDI STRING 'FF0 AIA'
FORMAT	ADDI (AT) [K]
FUNCTION	I [∧K] + AT → AT a.c. → CO
DESCRIPTION	If K is not specified, the data on the I-bus is added to the data in the specified register (AC or T), and the sum is deposited in the specified register. If K is specified, the data on the I-bus is ANDed with the data on the K-bus, the result is added to the data in the specified register, and the result is deposited in the specified register.

EXAMPLES

```
ADDE  (R7)                 /* AC PLUS R7 GOES TO AC AND R7 */
ADDR  (R7)                 /* AC PLUS R7 GOES TO R7 */
ADDR  (AC)             /*AC PLUS AC GOES TO AC, WHICH IS
                        EQUIVALENT TO SHIFT LEFT */
ADDRC  (R4)            /* AC PLUS R4 PLUS C-FLAG GOES TO R4 */
ADDM  (T)              /* AC PLUS M GOES TO T */
ADDM  (T)  KLOW        /* AC ANDED WITH KLOW PLUS M GOES TO T */
ADDI  (T)              /* I PLUS T GOES TO T */
```

The KAD stringops are used to add the data on the K-bus to the data in a register or the data on the M-bus and deposit the result in a register. Also, the data on the K-bus is ORed with the data in the register or on the M-bus, and the result is deposited in the Memory Address Register (MAR).

KADR DEFINITION	KADR STRING 'FFO LMI'
FORMAT	KADR (Rn) [K]
FUNCTION	Rn [∨K] → MAR Rn [+K] → Rn a.c. → CO
DESCRIPTION	The data on the K-bus is inclusive ORed with the data in the specified register (Rn), and this result is deposited in the Memory Address Register (MAR). Also, the data on the K-bus is added to the data in the specified register, and the sum is deposited in the specified register. If K is not specified, KADR is identical to MARR.
KADM DEFINITION	KADM STRING 'FFO LMM'
FORMAT	KADM (AT) [K]
FUNCTION	M [∨K] → MAR M [+K] → AT a.c. → CO
DESCRIPTION	The data on the M-bus is inclusive ORed with the data on the K-bus, and this result is deposited in the Memory Address Register (MAR). Also, the data on the M-bus is added to the data on the K-bus, and the result is deposited in the specified register (AC or T). If K is not specified, KADM is identical to MARM.

EXAMPLES

```
KADR (R6) KVAL     /*R6 PLUS K GOES TO R6, R6 ORED WITH
                    KVAL GOES TO MAR */
KADM (T) KVAL      /* M PLUS K GOES TO T, M ORED WITH KVAL
                    GOES TO MAR */
```

The INC stringops are used to increment data in a register or on the M-bus and deposit the result in a register.

INCE DEFINITION	INCE STRING 'KZERO FFI ILR'
FORMAT	INCE (Rn)
FUNCTION	Rn + 1 → Rn, AC a.c. → CO
DESCRIPTION	The data in the specified register (Rn) is incremented, and the result is deposited in both the specified register and the accumulator (AC).
INCR DEFINITION	INCR STRING 'KZERO FFI INR'
FORMAT	INCR (Rn)
FUNCTION	Rn + 1 → Rn a.c. → CO
DESCRIPTION	The data in the specified register is incremented, and the result is deposited in the specified register.

INCM DEFINITION	INCM STRING 'KZERO FFI ACM'
FORMAT	INCM (AT)
FUNCTION	M + 1 → AT a.c. → CO
DESCRIPTION	The data on the M-bus is incremented, and the result is deposited in the specified register (AC or T).

EXAMPLES

```
INCE (R4)    /* R4 PLUS 1 GOES TO R4 AND TO AC */
INCR (R4)    /* R4 PLUS 1 GOES TO R4 */
INCM (T)     /* M PLUS 1 GOES TO T */
```

The DEC stringops are used to decrement data in a register or on the I-bus and deposit the result in a register.

DECA DEFINITION	DECA STRING 'KONES FF0 SDR'
FORMAT	DECA (Rn)
FUNCTION	AC – 1 → Rn a.c. → CO
DESCRIPTION	The data in the accumulator (AC) is decremented, and the result is deposited in the specified register (Rn). The data in the accumulator is not affected.
DECT DEFINITION	DECT STRING 'KONES FF0 DCA'
FORMAT	DECT (AT)
FUNCTION	AT – 1 → AT
DESCRIPTION	The data in the specified register (AC or T) is decremented, and the result is deposited in the specified register.
DECI DEFINITION	DECI STRING 'KONES FF0 LDI'
FORMAT	DECI (AT)
FUNCTION	I – 1 → AT a.c. → CO
DESCRIPTION	The data on the I-bus is decremented, and the result is deposited in the specified register (AC or T).
DECR DEFINITION	DECR STRING 'KONES FF0 DSM'
FORMAT	DECR (Rn)
FUNCTION	Rn – 1 → Rn –1 → MAR a.c. → CO
DESCRIPTION	The data in the specified register (Rn) is decremented, and the result is deposited in the specified register. Also the Memory Address Register is set to all ones.

EXAMPLES

```
DECA (R3)       /* AC MINUS 1 GOES TO R3 */
DECT (T)        /* T MINUS 1 GOES TO T */
DECI (AC)       /* I MINUS 1 GOES TO AC */
DECR (R3)       /* R3 MINUS 1 GOES TO R3, ALL ONES GO TO MAR */
```

The AND stringops are used to specify that data from two sources (either two registers or a register and a bus) is to be ANDed and the result deposited in a destination register. An explicit

K-bus field assignment may accompany an AND stringop if it is desired to mask the result.

ANDR DEFINITION	ANDR STRING 'FFO ANR'
FORMAT	ANDR (Rn) [K]
FUNCTION	AC [ΛK] Λ Rn → Rn l.r. → CO
DESCRIPTION	If K is not specified, the data in the accumulator (AC) is ANDed with the data in the specified register (Rn), and the result is deposited in the specified register. If K is specified, the data in the accumulator ANDed with the data on the K-bus, this result is ANDed with the final result is deposited in the specified register.
ANDM DEFINITION	ANDM STRING 'FFO ANM'
FORMAT	ANDM (AT) [K]
FUNCTION	AC [ΛK] Λ M → AT l.r. → CO
DESCRIPTION	If K is not specified, the data in the accumulator (AC) is ANDed with the data on the M-bus, and the result is deposited in the specified register (AC or T). If K is specified, the data in the accumulator is ANDed with the data on the K-bus, this result is ANDed with the data on the M-bus, and the final result is deposited in the specified register.
ANDI DEFINITION	ANDI STRING 'FFO ANI'
FORMAT	ANDI (AT) [K]
FUNCTION	I [ΛK] Λ AT → AT l.r. → CO
DESCRIPTION	If K is not specified, the data on the I-bus is ANDed with the data in the specified register (AC or T), and the result is deposited in the specified register. If K is specified, the data on the I-bus is ANDed with the data on the K-bus, this result is ANDed with the data in the specified register, and the result deposited in the specified register.

EXAMPLES

```
ANDR (R6)              /* AC ANDED WITH R6 GOES TO R6 */
ANDM (T)               /* AC ANDED WITH M GOES TO T */
ANDM (T) KLOW          /* AC ANDED WITH KLOW ANDED WITH M
                       GOES TO T */
ANDI (T)               /* I ANDED WITH T GOES TO T */
```

The IOR stringops are used to specify that data from two sources (either two registers or a register and a bus) is to inclusive ORed and the result deposited in a destination register. An explicit K-bus field assignment may accompany IOR stringops if it is desired to mask one of the sources before the inclusive OR is performed.

IORR DEFINITION	IORR STRING 'FFO ORR'
FORMAT	IORR (Rn) [K]
FUNCTION	(AC [ΛK]) ∨Rn → Rn AC [ΛK] → CO
DESCRIPTION	If K is not specified, the data in the accumulator (AC) is inclusive ORed with the data in the specified register (Rn), and the result is deposited in the specified register. If K is specified, the data in the accumulator is ANDed with the data on the K-bus, this result is inclusive ORed with the data in the specified register, and the final result is deposited in the specified register.

IORM DEFINITION IORM STRING 'FF0 ORM'

FORMAT IORM (AT) [K]

FUNCTION $(AC\ [\wedge K])\vee M \rightarrow AT \quad AC\ [\wedge K] \rightarrow CO$

DESCRIPTION If K is not specified, the data in the accumulator (AC) is inclusive ORed with the data in the M-bus, and the result is deposited in the specified register (AC or T). If K is specified, the data in the accumulator is ANDed with the data on the K-bus, this result is inclusive ORed with the data on the M-bus, and the final result is deposited in the specified register.

IORI DEFINITION IORI STRING 'FF0 ORI'

FORMAT IORI (AT) [K]

FUNCTION $(I\ [\wedge K])\wedge AT \rightarrow AT \quad I\ [\wedge K] \rightarrow CO$

DESCRIPTION If K is not specified, the data on the I-bus is inclusive ORed with the data in the specified register (AC or T), and the result is deposited in the specified register. If K is specified, the data on the I-bus is ANDed with the data on the K-bus, this result is inclusive ORed with the data in the specified register, and the final result is deposited in the specified register.

EXAMPLES

```
IORR (R0)          /* AC ORED WITH R0 GOES TO R0 */
IORM (T)           /* AC ORED WITH M GOES TO T */
IORI (T)           /* I ORED WITH T GOES TO T */
IORI (T) KLOW      /* I ANDED WITH KLOW ORED WITH T GOES TO T */
```

The XNR stringops are used to specify that data from two sources (either two registers or a register and a bus) are to be exclusive NORed and the result deposited in a destination register. An explicit K-bus field assignment may accompany an XNR stringop if it is desired to mask the accumulator before the exclusive NOR operation is performed.

XNRR DEFINITION XNRR STRING 'FF0 XNR'

FORMAT XNRR (Rn) [K]

FUNCTION $(AC\ [\wedge K])\,\overline{\oplus}\,Rn \rightarrow Rn \quad Rn \wedge AC\ [\wedge K] \rightarrow CO$

DESCRIPTION If K is not specified, the data in the accumulator is exclusive NORed with the data in the specified register (Rn), and the result is deposited in the specified register. If K is specified, the data in the accumulator is ANDed with the data on the K-bus, this result is exclusive NORed with the data in the specified register, and the result is deposited in the specified register.

XNRM DEFINITION XNRM STRING 'FF0 XNM'

FORMAT XNRM (AT) [K]

FUNCTION $(AC\ [\wedge K])\,\overline{\oplus}\,M \rightarrow AT \quad AC\ [\wedge K] \wedge M \rightarrow CO$

DESCRIPTION If K is not specified, the data in the accumulator (AC) is exclusive NORed with the data on the M-bus, and the result is deposited in the specified register (AC or T). If K is specified, the data in the accumulator is ANDed with the data on the K-bus, this result is exclusive NORed with the data on the M-bus, and the final result is deposited in the specified register.

XNRI DEFINITION XNRI STRING 'FF0 XNI'

FORMAT XNRI (AT) [K]

FUNCTION $(I[\wedge K])\,\overline{\oplus}\,AT \rightarrow AT \quad AT[\wedge K] \wedge I \rightarrow CO$

DESCRIPTION

If K is not specified, the data on the I-bus is exclusive NORed with the data in the specified register (AC or T), and the result is deposited in the specified register. If K is specified, the data on the I-bus is ANDed with the data on the K-bus, this result is exclusive NORed with the data in the specified register, and the final result is deposited in the specified register.

EXAMPLES

```
XNRR  (R5)                /* AC XNORED WITH R5 GOES TO R */
XNRM  (T)                 /* AC XNORED WITH M GOES TO T */
XNRI  (T)                 /* I XNORED WITH T GOES TO T */
```

The MSK stringops are used to mask the data in a register or on the M-bus with the data on the K-bus and deposit the result in a register. An explicit K-bus field assignment should be used with a MSK stringop.

MSKR DEFINITION

```
MSKR  STRING  'FF0  TZR'
```

FORMAT

```
MSKR  (Rn)  [K]
```

FUNCTION

Rn [∧K] → Rn I.r. → CO

DESCRIPTION

The data in the specified register (Rn) is ANDed with the data on the K-bus, and the result is deposited in the specified register. If K is not specified, the data in the specified register is not affected.

MSKM DEFINITION

```
MSKM  STRING  'FF0  LTM'
```

FORMAT

```
MSKM  (AT)  [K]
```

FUNCTION

M [∧K] → AT I.r. → CO

DESCRIPTION

The data on the M-bus is ANDed with the data on the K-bus, and the result is deposited in the specified register (AC or T); MSKM is identical to MOVM.

EXAMPLES

```
MSKR (R5) KVAL    /* R5 ANDED WITH KVAL GOES TO R5 */
MSKM (T) KVAL     /* M ANDED WITH KVAL GOES TO T */
```

The TST stringops are used primarily to test data in a register or on the M-bus or I-bus for zero. An explicit K-bus field assignment may accompany a TST stringop in order to test selected bits of the source operand.

TSTR DEFINITION

```
TSTR  STRING  'FF0  TZR'
```

FORMAT

```
TSTR  (Rn)  [K]
```

FUNCTION

Rn [∧K] → Rn Rn [∧K] → CO

DESCRIPTION

If K is specified, the data in the specified register (Rn) is tested for zero. If the specified register contains zero; the carry output (CO) is forced to zero; if the specified register does not contain zero, the carry output is forced to one. If K is specified, the contents of the specified register are ANDed with the data on the K-bus, and the result is deposited in the specified register and tested for zero, as described above. TSTR is identical to MSKR.

TSTM DEFINITION

```
TSTM  STRING  'FF0  LTM'
```

FORMAT

```
TSTM  (AT)  [K]
```

FUNCTION

M [∧K] → AT M [∧K] → CO

DESCRIPTION If K is not specified, the data on the M-bus is deposited in the specified register (AC or T) and tested for zero. If the data on the M-bus is zero, the carry output (CO) is forced to zero; if the data on the M-bus is not zero, the carry output is forced to one. If K is specified, the data of the M-bus is ANDed with the data on the K-bus, and the result is deposited in the specified register and tested for zero, as described above. TSTM is identical to MSKM.

TSTI DEFINITION TSTI STRING 'FF0 ORI'

FORMAT TSTI (AT) [K]

FUNCTION $I\ [\wedge K] \vee AT \rightarrow AT \quad I\ [\wedge K] \rightarrow CO$

DESCRIPTION If K is not specified, the data on the I-bus is inclusive ORed with the data in the specified register (AC or T), and the result is deposited in the specified register. The data on the I-bus is tested for zero; if the data on the I-bus is zero, the carry output (CO) is forced to zero; if the data on the I-bus is not zero, the carry output is forced to one. If K is specified, the data on the I-bus is ANDed with the data on the K-bus. This result is tested for zero, as described above. This result is, also, inclusive ORed with the data in the specified register, and the result is deposited in the specified register. TSTI is identical to IORI.

EXAMPLES

```
TSTR (R1)           /* ONE GOES TO CO IF R1 NOT ZERO */
TSTR (R1) KLOW      /* R1 ANDED WITH KLOW GOES TO R1, ONE GOES
                     TO CO IF RESULT NOT ZERO */
TSTM (T)            /* ONE GOES TO CO IF M NOT ZERO, M GOES TO T */
TSTI (T)            /* ONE GOES TO CO IF I NOT ZERO, I ORED
                     WITH T GOES TO T */
```

The NOT stringops are used to complement the data in a register or on the M-bus and deposit the result in a register. The NEGT stringop is used to take the two's complement of the data in the accumulator or a T register.

NOTR DEFINITION NOTR STRING 'KZERO FF0 CMR'

FORMAT NOTR (Rn)

FUNCTION $\overline{Rn} \rightarrow Rn \quad 0 \rightarrow CO$

DESCRIPTION The data in the specified register (Rn) is complemented, and the result is deposited in the specified register.

NOTM DEFINITION NOTM STRING 'KZERO FF0 LCM'

FORMAT NOTM (AT)

FUNCTION $\overline{M} \rightarrow AT \quad 0 \rightarrow CO$

DESCRIPTION The data on the M-bus is complemented, and the result is deposited in the specified register (AC or T).

NEGT DEFINITION NEGT STRING 'FFI KZERO CIA'

FORMAT NEGT (AT)

FUNCTION $\overline{AT} + 1 \rightarrow AT \quad A.C. \rightarrow CO$

DESCRIPTION The data in the specified register (AC or T) is complemented and incremented (i.e., twos complement), and the result is deposited in the specified register.

The CLRR and SETR stringops are used to transfer zero or all ones, respectively, to a register.

CLRR DEFINITION

```
CLRR  STRING  'KZERO CLR'
```

FORMAT

```
CLRR  (Rn)  [CI]
```

FUNCTION

$0 \rightarrow Rn \quad CI \rightarrow CO$

DESCRIPTION

Zero is deposited in the specified register (Rn). The carry output (CO) is forced to the state of the carry input (CI). If the carry input is not specified, the carry output is forced to zero.

SETR DEFINITION

```
SETR  STRING  'KZERO  FF0  CSR'
```

FORMAT

```
SETR  (Rn)
```

FUNCTION

$-1 \rightarrow Rn \quad 0 \rightarrow CO$

DESCRIPTION

The specified register (Rn) is set to all ones.

EXAMPLES

```
CLRR (AC)                     /* CLEAR AC */
CLRR (R3)                     /* CLEAR R3 */
SETR (R3)                     /* SET R3 TO ALL ONES */
```

The SHRT stringop is used to specify that the data in the accumulator or the T register is to be shifted right one bit position.

SHRT DEFINITION

```
SHRT  STRING  'KZERO  SRA'
```

FORMAT

```
SHRT  (AT)  [LI]
```

FUNCTION

$LI \rightarrow AT_H \quad AT_H \rightarrow AT_L \quad AT_L \rightarrow RO$

DESCRIPTION

The data in the specified register (AC or T) is shifted right one bit position. The high order bit of the specified register is set to the state of the left input (LI). If the left input is not specified (i.e., if the Flag Output function is not explicitly specified), the high order bit of the specified register receives a zero. The right output (RO) is forced to the state of the low order bit of the specified register.

EXAMPLES

```
SHRT  (T)                     /* T RIGHT SHIFTED GOES TO T, HIGH
                              ORDER BIT OF T SET TO ZERO */
SHRT  (AC)  FFI               /* AC RIGHT SHIFTED GOES TO AC, HIGH
                              ORDER BIT OF AC SET TO ONE */
```

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 246-7501

Printed in U.S.A. MCS 049-0276/10K