varian data machines /a varian subsidiary

V70 MICROPROGRAMMING USER'S MANUAL

varian data machines

98 A 9906 073

JUNE 1974

PREFACE

Preface (about the guide itself -- prerequisites, its organization and why).

Microprograms are aptly called **firmware** to place them between the realms of software and hardware. Where those two conventional divisions of a computer overlap is an area which provides many of the best features of both. The use and benefits of microprogramming depend upon the user having an understanding of both and their complex interaction.

The reader of this guide should have some knowledge of the hardware components of a computer system, such as the functions and uses of registers, schemes of handling interrupts etc. Programming techniques which make efficient assembly-language functions like indexing and high-speed algorithms will be useful here too. When a microprogram is executed thousands of times more often than any one application program, its *fine tuning* is also needed that many more times. Also the microprogrammer should know the problem-oriented languages used. To choose which operators to microprogram, the designer must be aware of the eventual applications. Combining operators which are often used in the same sequence could form a single microprogrammed operator with a greater overlapping of actions.

All components of a computer system seem to be increasingly complex yet easier and easier to use. Though microprogramming adds more complexity the result is to make a system easier to use. One goal of this guide is to bring microprogramming into the range of a good programmer. To that end the guide is written in simple language (with a minimum of exotic terms and a glossary to look up any of those) and a gradual progression from the big picture to the details through numerous examples. The examples are annotated and explained with the same tools that will aid in the planning as well as understanding.

This guide is both an introduction and a reference. If microprogramming is new to you, start at the beginning of this guide and use it as a tutorial. Later the book can be used for reference. The charts and examples are built up in a logical development so that the complete examples will be a pattern for your programming.

Varian Data Machines does not assume responsibility for microprograms written and implemented according to the recommendations outlined herein.

To improve the usefulness of this guide please return the reader questionaire in the back after reading and using this volume.

Related Documentation

The Writable Control Store manual (98 A 9906 08x) provides information about the installation, theory of operation, maintenance and test programs for the hardware storage of microprograms.

Information about the Varian 70 series processor is contained in the applicable system handbook and in more detail in the Processor Manual (98 A 9906 02x). (The x at the end of each document part number is the revision number and can be any digit 0 through 9.)

The VORTEX Reference Manual (98 A 9952 10x) describes the use of the VORTEX operating system. The MOS (Master Operating System) Reference Manual (98 A 9952 09x) provides similar information necessary to use microprogramming software with that operating system.

The following Varian manuals provide additional aids to the understanding of Varian Computer Systems.

Title	**Document Number**
72 System Handbook	98 A 9906 20x
73 System Handbook	98 A 9906 01x
74 System Handbook	98 A 9906 21x
Core Memory Manual	98 A 9906 03x
Semiconductor Memory Manual	98 A 9906 04x
Option Board Manual	98 A 9906 05x
Power Supply Manual	98 A 9906 06x

TABLE OF CONTENTS

SECTION 3
TECHNIQUES

SECTION 4
MICROPROGRAM DATA ASSEMBLER, MIDAS

SECTION 5
CODING FROM FLOW DIAGRAMS

SECTION 6
MICROPROGRAM SIMULATOR, MICSIM

SECTION 7
MICROPROGRAM UTILITY PROGRAM, MIUTIL

SECTION 8
DECODER CONTROL STORE, I/O CONTROL AND ADDITIONAL TOPICS

SECTION 9
GLOSSARY

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS *(continued)*

LIST OF TABLES

SECTION 1

INTRODUCTION

Most of this book discusses how to microprogram. As an incentive to read further, here are some general reasons why to microprogram. The advantages of microprogramming are based upon a comparison with a conventional system either completely without microprogramming or where it is not accessible (figure 1-1). After a brief summary of the advantages a comparison with a conventional system gives more details and a specific picture of a microprogrammed operation.

1.1 ADVANTAGES

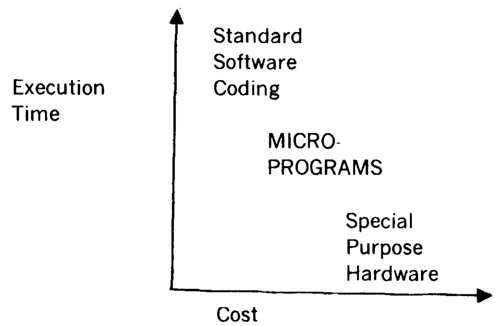
A basic reason to microprogram is the one stated at first. The initial idea was proposed for a "systematic" approach **to the " usual somewhat ad hoc procedure" used to design** the control system of a machine. The narrow view in the design of either software or hardware without an awareness of the other can lead to a less efficent functioning, like a refrigerator converted into a vacuum cleaner -- there may be some common useful parts but we would push around a great deal that did not help the vacuuming. Good basic operators which match the eventual application will improve the entire efficiency.

The usual random logic can be reduced with a more structured organization. A conventional computer system uses a collection of counters, special flip-flops, decoding networks and other components unique to a particular purpose for control logic. In contrast a microprogrammed memory replaces most of this. The microprogram storage is formed of regular and repetitive units. There are fewer components thus increasing the reliability of the system.

The flexibility of the instructions in the control store offers the ability to change the system in ways so basic that they are not at all feasible in a fixed instruction set. Field changes can be made by merely changing the controlling microprograms. Final systems definition can be postponed until a later stage of the design. Performance can be economically expanded at a lower cost.

Emulation of a number of diverse devices, not only processors but peripheral controllers for instance, can be carried out on a single microprogrammed system. Simultaneous emulation of some devices can be made or the target system can be changed depending upon needs. This would save some reprogramming and retraining and yet gain the speed and reliability of a more advanced system. Also the documentation and minor logistic problems of a new machine would be avoided.

For more reliability and the continuous performance necessary in many uses of computers, diagnostics and servicing aids may be implemented in the control store. To pinpoint problems the microprocessor can both test and set states not available to the assembly-language programmer on a conventional machine.



Instructions Tailored To Particular Environments

In general, microprogrammed instructions permit more compact program representation. They use less main memory than the equivalent would in conventional code. Consequently, fewer memory fetches for anything other than data are needed.

As an example of a possible microprogrammed operator which reduces memory fetches, consider a common use of arrays. Higher-level programming languages, such as FORTRAN, BASIC, COBOL -- in fact, nearly all -- have facilities for expressing a repetitive linear data structure, a list or array. Arrays are an integral part of a large class of techniques for diverse problems. Yet good operators for arrays as such are not available as simple, single instructions in a conventional machine.

In usual machine code the function of adding two numerical arrays of the same size and number of elements usually requires a series of actions as follows for each pair of elements:

a. load memory to register

b. add memory to register

c. store register result in memory

d. update indices and close loop

The first two steps would each require a memory fetch and the last step as many as three memory fetches.

A microprogrammed instruction would provide initializing data descriptors and repetitively executing micro-operators over the described arrays of data. To start the program segment would require several steps:

a. load the starting address, increment and extent of each array

b. load the result's starting address, increment and extent

c. define the end and branch condition

This initialization could be followed by one instruction to execute the newly-defined operator equivalent to the series of typical instructions.

An extension of this principle of reducing memory retrieval of instructions occurs in some special cases where data normally resident in the stream of instructions can be removed and instead reside in special-purpose micro-routines. For example, if the array addition algorithm above were limited to fixed-length arrays with fixed-size elements, the increment and extent parameters could be stored as local constants in the microprogram, eliminating the need to transfer this information in the initial sequence.

1.2 GUIDE TO THIS MANUAL

The purpose of this section is to provide the user with a helpful idea of the structure of the remainder of the manual. The order of the following sections is based on the order in which a programmer needs the information to plan, then code, test and run microprograms.

Information in the sections

Introduction (Section 1)
  Advantages of microprogramming
  Guide to the remainder of the manual
  Conventions (defining some words and notation) in the manual
  Components of microprogrammed systems

Capabilities (Section 2)
  Micro operations available in central control store
  Building blocks of microprograms providing data transfer and transformations, conditional tests, and memory access

Techniques (Section 3)
  Explanation of interface with the 620 emulation
  Procedures to use flow diagrams to write microprograms
  Examples of microprograms

Microprogram Assembler (Section 4)
  Directives to code microprograms
  Macros
  Operating instructions

Coding from Flow Diagrams (Section 5)
  Conversion steps and tables
  Examples from section 3

Microprogram Simulator (Section 6)
  Directives
  Operating instructions

Microprogram Utility (Section 7)
  Directives
  Operating instructions

Decoder control store, I/O control and additional topics (section 8)
  Format and use of optional decoder control store
  I/O microprogramming procedures and example

Glossary (Section 9)
  Terminology for microprogramming defined
  Mnemonics defined

1.3 NOTATION IN THIS MANUAL

References to Microinstruction Fields

Within the microinstruction the fields are named with the two-letter references recognized by the micro-assembler. Some of the fields have names which are used in the text, such as the CF field conveniently called the **carry field**.

References Within Fields

The bits within the fields are often discussed one at a time. Several techniques are used to single out bits. A field may be represented with the letter **X** in bit positions not involved in the action being discussed. **1X** for a two-bit field indicates that only the high-order bit is required to be one in this action, i.e., setting the field to 10 or 11. High-order and leftmost are synonymous to select a particular bit or group of bits. Similarly low-order and rightmost select the same bit or a contiguous set of bits. Finally less often a bit is mentioned by number with the convention that bits are numbered from right to left starting with zero.

Syntax of Directives

In the directive formats for the microprogramming software the syntax is given with the following conventions:

Boldface type indicates a required parameter

Italic type indicates an optional parameter

Upper-case type indicates that the item is to be entered exactly as written

Lower-case type indicates a variable and shows where the user enters a value for that variable.

The formation of a list of the same items is indicated by three consecutive periods.

For example, the syntax for the MIDAS FORM statement is as follows

label	**FORM field(1), field(2),...,.** *field(n)*

Where:

label	is a symbol as defined in MIDAS basic elements
each **field**	is a field identifier which is the field length in decimal, followed by an optional hexadecimal constant enclosed in parentheses

Numbers

Microinstruction fields are given in binary notation unless indicated otherwise in the context of the reference.

Definitions

To remove one barrier that often exists to the understanding of microprogramming this section clarifies some terms we use.

In a computer system many different kinds of storage exist for data, instructions or both. Microprograms reside in the system's control store. All control store must be writable in some manner so that the control information can be introduced. The desire for greater speed often leads to the design of storage that can only be loaded once and even then only by mechanical or electromechanical means. These are designated as **read only** or ROM for read-only memory. This differentiates them from the arrays whose contents can be changed by the user. This is called writable control store (WCS).

The microprogram is a series of microinstructions. A microinstruction resides in one fixed-length word in control store. The microword is 64 bits long and selects the operations which occur in one machine cycle (with some exceptions). The individual operations, micro-operations or primitives, are defined by fields within the microword.

In this manual whenever you encounter unfamilar words look for the definition at the first use of the word or consult the glossary in section 9.

1.4 COMPONENTS

1.4.1 Hardware for Microprogrammed Systems

Though the software for microprogramming provides an interface for the user to program the system, to plan a good system one needs to be very aware of the actual functions of the hardware. The tangible parts of the microprogramming system are described below.

Processor

The major functional components of the Varian 70 series processor (figure 1-2) are central control, data loop, memory control, I/O data loop, and I/O control. The processor communicates with the computer control panel via the I/O bus.

The processor speed is 165 nanoseconds for a microinstruction.

Central Control

Central control provides supervision for most of the major components in the processor. Direct control is exercised over the data loop. Requests may be made to other components, such as memory and I/O control.

The key element in central control is a 64-bit control buffer. This buffer, which is simply a microinstruction, completely describes a set of actions for the other processor components. For example, the data loop might be instructed to increment one of the general-purpose registers. The memory control might be requested to begin the fetch of a 16-bit word from main memory. Thus, the control buffer holds the current microinstructions. It is somewhat analogous to the instruction register in assembly-language programming.

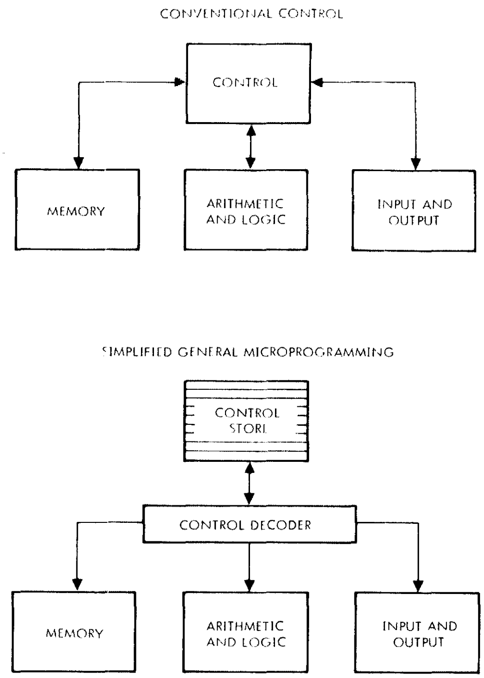
The 64 bits also specify the location of the new contents for the control buffer. The control buffer is always loaded from 64-bit central control store. Thus, execution of a microprogram basically consists of the control buffer being sequentially loaded with the appropriate 64-bit values. Central control store in a Varian 70 series system is divided into pages, each consisting of 512 64-bit words. Page zero of central control store always contains a set of microinstructions which direct the processor components to behave like a 620/f. This set of 512 microwords is thus called the 620/f emulation, and resides in read-only memory (ROM). Other central control store pages may be added with the writable control store (WCS) option, thus allowing the user to specify in detail the actions of the processor components.

The microprograms for the standard instruction set are described in the microinstruction flowcharts in the System Maintenance Manual and in assembly language in an appendix to this guide.

Data Loop

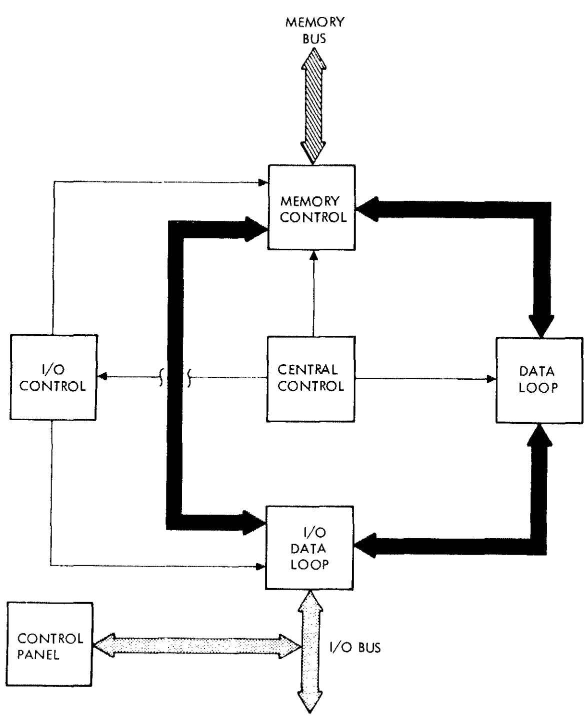
The data loop provides transfer paths, data transformation circuits, storage registers and counters (figure 1-3).

Under control of the central control buffer the arithmetic and logic unit (ALU) performs basic arithmetic functions



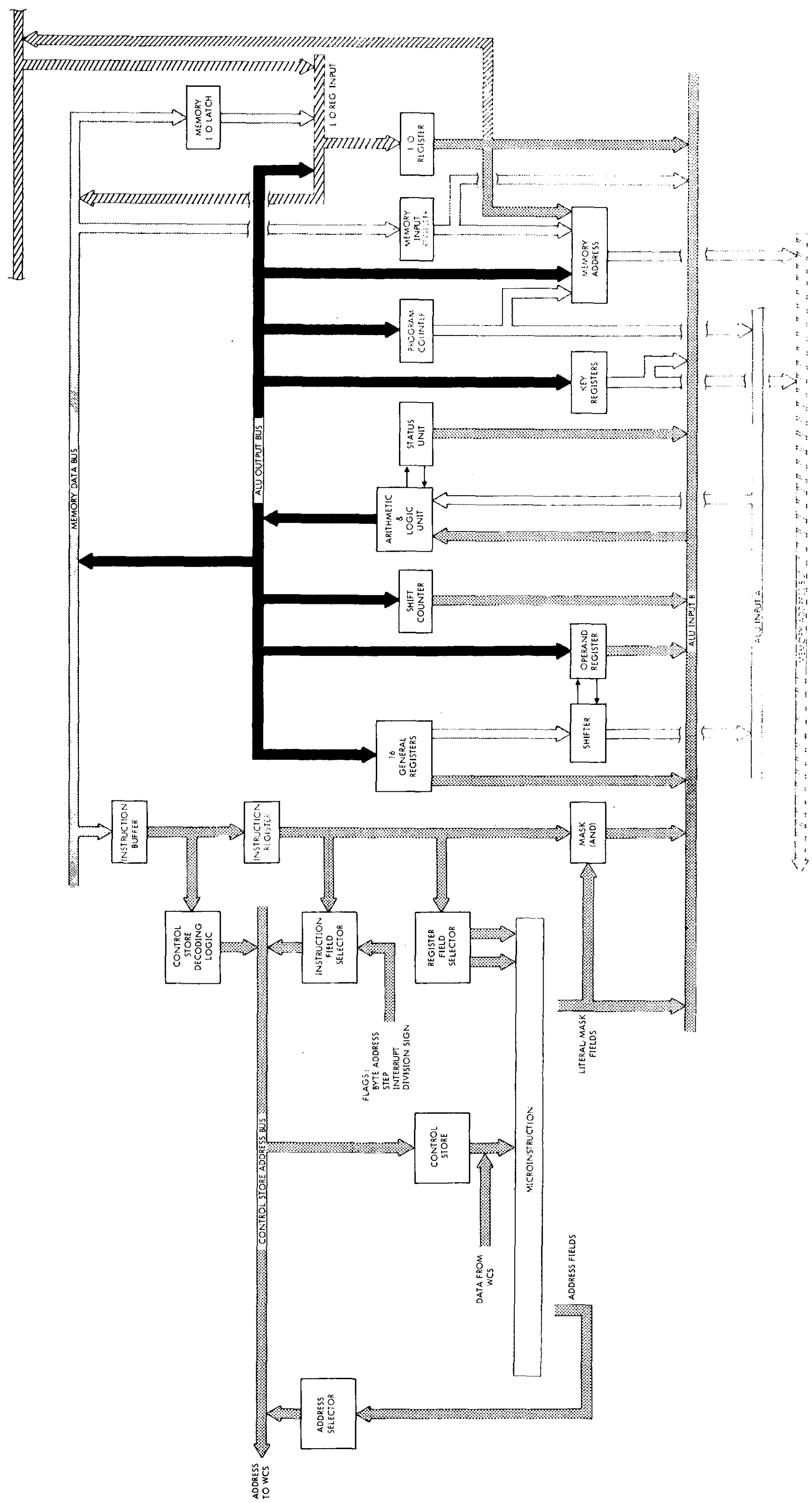
VTII-1800

Figure 1-1. Simplified Comparison of a Microprogrammed and a Conventional Computer



VTII-1500A

Figure 1-2. Varian 73 Processor Block Diagram



VT13-292

Figure 1-3. Varian 73 Processor Data Paths such as addition, subtraction, and the common logical functions including AND and OR. ALU output can be directed to a number of places, including registers and counters in the data loop, registers in the I/O data loop, and to memory control.

Memory Control

The memory control section of the processor performs tasks initiated by the central control, I/O control and options. These tasks consist of reading a 16-bit word from memory or writing a word or byte into memory.

Memory control acknowledges receipt of the signal to the requesting sections and signals when done with the task. When one request is accepted no others are acknowledged until the current one is completed, but central control can override its own prior request.

I/O Data Loop

The I/O data loop contains a multiplexor, I/O data register, and drivers and receivers. Three sources of data are applied to the I/O data loop: data from the I/O bus, data from the arithmetic and logic unit, and data from the memory I/O register (MIOR). The input data is selected by the I/O multiplexor under control of the I/O control signals and transferred on to the bidirectional I/O bus.

In addition to being applied to the I/O drivers, the output of the I/O data register is applied to the data loop and memory control sections.

I/O Control

The I/O control operates under control of an independent read-only memory (ROM). It performs I/O operations initiated either by the central control or I/O device activity. This permits I/O operations to proceed with minimal impact on internal processor functions. The I/O performs programmed I/O initiated by the central control. Both normal and high-speed direct memory access (DMA) are handled by the I/O control. I/O interrupts are processed by I/O control.

1.4.2 Writable Control Store

The Writable Control Store (WCS) extends the processor's read-only control store to permit addition of new instructions, development of microprogrammed diagnostics, and optimal tailoring of the computer system to its applications.

Unlike the read-only control store which contains the Varian 70 series standard instruction set and cannot be altered, the WCS can be loaded from the computer's main memory under control of I/O instructions. This capability of altering the contents of the WCS gives the user complete access to the resources of the computer system.

A test program for the WCS hardware is provided to assist in maintaining the system. Operating the test program is described in the maintenance manual for the WCS.

Configurations

The WCS is available in three configurations:

1. One page (512 words) of control store and a subroutine stack (Model 7X-4001)
2. Half page of control store and a subroutine stack (Model 7X-4000)
3. One page with a subroutine stack, a writable decoder control store and a writable I/O control store (Model 7X-4002)

Model 7X-4002 is shown in the block diagram of figure 1-4. The three control stores shown in this diagram are the writable counterparts for read-only components of the processor.

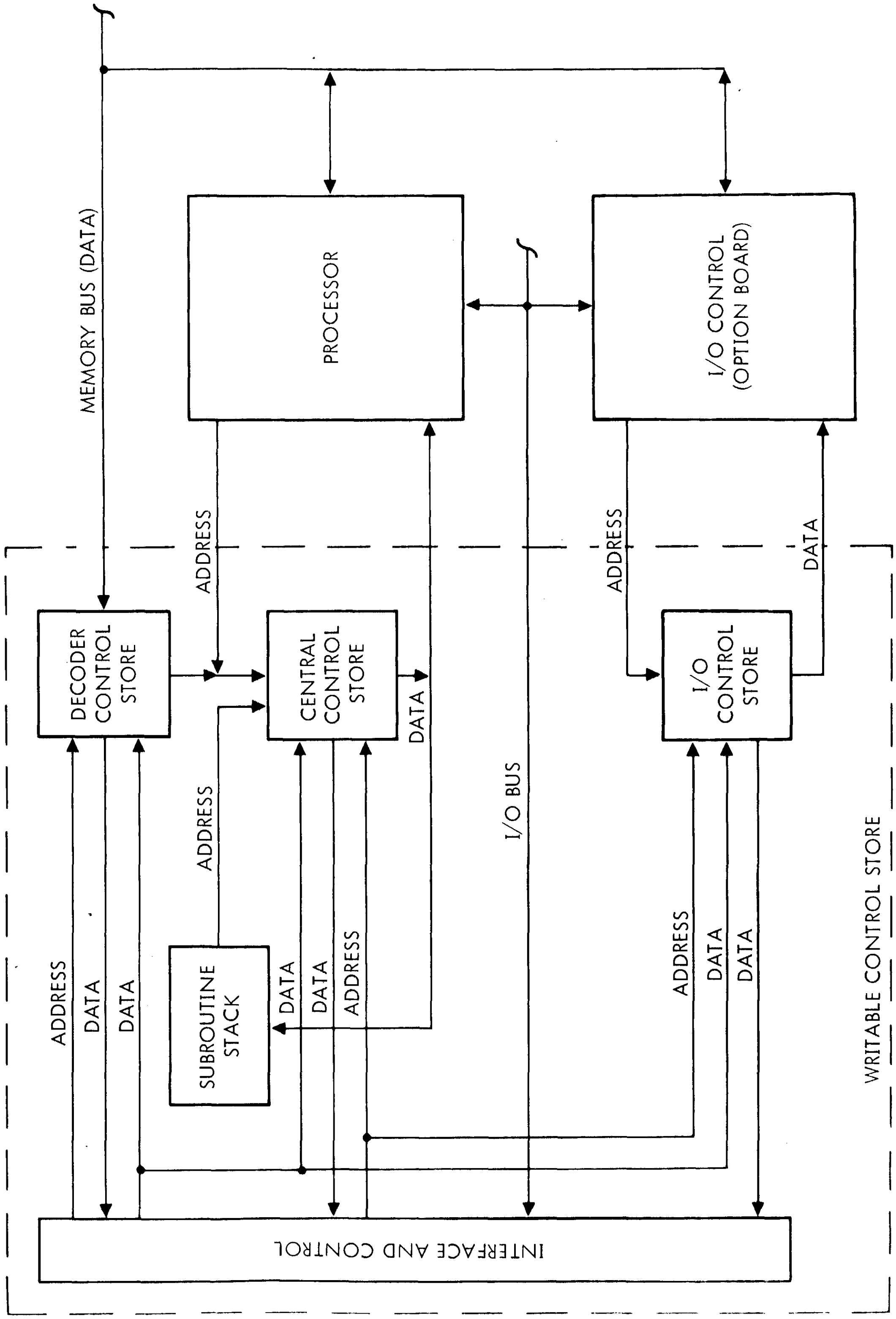
The decoder control store replaces the instruction buffer, decoder, and decoding logic in the processor to improve instruction set changes. It is formed from two 16-word by 16-bit memory arrays with the logic that decodes main memory instructions into an address for the central control store.

The central control store is a counterpart of the page zero of read-only storage. With each processor clock pulse, a 64-bit microinstruction is read from the central control store to specify the actions to occur. A typical microinstruction may define several operations such as selecting the next control store microinstruction to be executed, test conditions for branching, initiating memory operations and selecting ALU functions.

The I/O control store contains a 256-word memory array of 16-bit words.

A standard feature with all WCS models is the subroutine stack that increases storage efficiency by providing a call and return capability for subroutines of microinstructions. Up to 16 addresses for branches can be stored in the stack. Operations are provided for pushing, popping, and deleting an entry.

Up to three writable control store pages (2048 words including the page-zero read-only store) can be installed in a Varian 70 series computer system. Each writable control store page unit is contained on a printed-circuit board that plugs into a Varian 70 series mainframe.



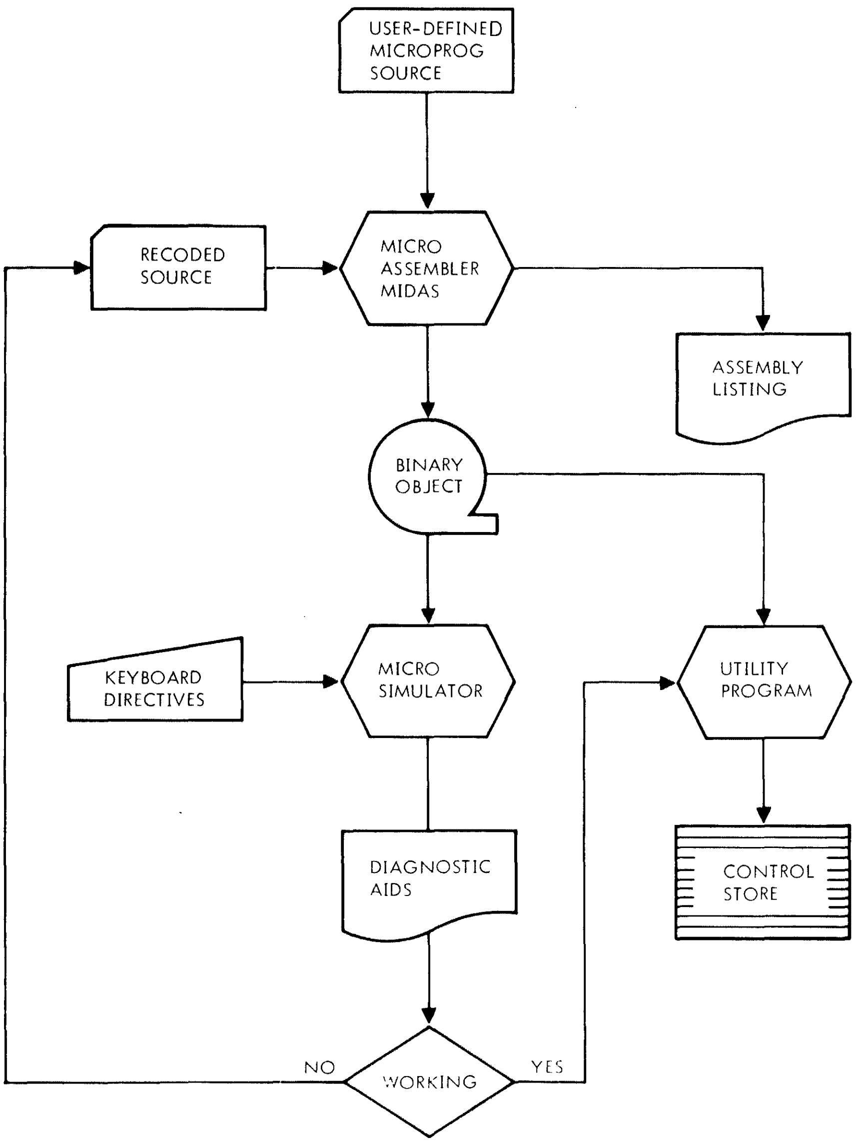
VTII-1816

Figure 1-4. Writable Control Store Block Diagram

1.4.3 Software Modules

Microprogram preparation uses a sequence of software provided with the WCS. First the program is written and assembled with a special assembler called MIDAS. Upon error-free assembly the code is run in a simulated environment which is completely independent of a WCS. The ability to trace and correct the execution is available with the microsimulator. These first two steps can occur without a WCS. Then only when the microprograms are checked completely the code can be loaded in the WCS with the micro-utility program. In addition to loading the utility provides some diagnostics. These steps are depicted in figure 1-5.

All the components of the microprogramming software were designed to operate both under operating systems, MOS and VORTEX, and as stand-alone programs on the Varian 70 or 620 series computers. Operating systems require a minimum configuration (see the manual for the particular



VTII-1799

Figure 1-5. Steps for Realizing Microprograms operating system). Table 1-1 lists the hardware requirements for microprogramming software.

Assembler

An assembler is a computer program which translates symbolic statements into machine instructions. The symbols are more meaningful than the strings of bit settings they represent. In addition to simply translating from symbolic to the executable code, the assembler assigns storage locations to the assembled instructions and produces a form of the instructions for loading into the processor's control store.

The microprogram data assembler (MIDAS) allows the user to prepare microprograms for the WCS. Through the use of operation mnemonics, symbolic addressing, address-field calculation, macro definitions, error detection and auto-

MIDAS is designed to provide the user with a tool for microprogram implementation. While relieving the user of much of the tedious housekeeping associated with generating microinstructions and their data fields, it also allows the user to describe the microinstructions at their most fundamental level.

Simulator

Verifying that the microprogram does indeed solve the problem is the next step. A logical step in implementing a microprogram is to run it with the microsimulator. The effects of executing a faulty microprogram are likely to be worse than those caused by poor assembly-language coding.

The simulator runs the output from the assembler within main-memory storage. At selected times conditions and the contents of data locations can be changed and examined. Projected changes can be simulated to evaluate eventual changes to the microprograms.

After determining that the code is error-free the WCS can be loaded with the utility program, which uses a command set as consistent as possible with the simulator.

Utility

Loading the WCS with the assembled and test microcode is performed by the microprogram utility, MIUTIL. In addition, on-line debugging directives are available through the utility.

Table 1-1. WCS Software Configuration Matrix

Program	Operating System	Memory (K) 8	12	16	20	24	32	TTY Keyboard/ Printer	TTY PT Reader	TTY PT Punch	High-Speed PT Reader
Micro-Assembler	VORTEX			X	R	O	O	X	N	N	O
MIDAS	MOS	X	R	O	O	O	O	X	X	N	O
	SA	X	R	O	O	O	O	X	X	X	O
Micro-Simulator	VORTEX				X	R	O	X	N	N	X
MICSIM	MOS			X	R	O	O	X	X	N	R
	SA			X	R	O	O	X	X	N	R
Micro-Utility	VORTEX			X	O	O	O	X	N	N	X
MIUTIL	MOS	X	R	O	O	O	O	X	X	N	R
	SA	X	R	O	O	O	O	X	X	N	R
WCS Test Program		X	N	N	N	N	N	R	O	N	X

(continued)

Table 1-1. WCS Software Configuration Matrix
(continued)

Program	Operating System	High-Speed PT Punch	Card Reader	Card Punch	Line Printer	Mag Tape	Rotating Memory	WCS Option
Micro-Assembler MIDAS	VORTEX	O	R	O	R	O	X	
	MOS	O	R	R	R	X	O	
	SA	O	R	O	R	O	N	
Micro-Simulator MICSIM	VORTEX	N	R	N	R	O	X	
	MOS	N	R	N	R	X	O	
	SA	N	R	N	R	O	N	
Micro-Utility MIUTIL	VORTEX	N	R	N	R	O	X	X
	MOS	N	R	N	R	X	O	X
	SA	N	R	N	R	O	N	X
WCS Test Program		N	N	N	N	N	N	X

Legend:

X = minimum configuration

R = recommended (recommended in place of its minimum counter part)

O = optional (can be used but program will function completely without it)

N = not used with the program

SECTION 2
CAPABILITIES

This section describes micro-operations available with Varian 70 series systems. The operations are grouped into the following categories:

a. data transfer and transformation

b. addressing and conditional actions

c. memory access

d. other controls

A basic example follows these sections. Some important timing considerations are presented at the conclusion of this section of capabilities.

This section describes only central control store programming.

I/O and decoder control stores are treated in section 8.

2.1 GENERAL MICROINSTRUCTIONS

The 64 bits of the microinstruction are grouped into fields referenced by either an ordinal number or a two-letter name for the microprogram assembler. The full resources of the system can be exploited by the user who is familiar with all the defined microinstruction fields. To start most common operations, a limited set of fields is involved.

Because some of the bit combinations in the microword have no function, the user should be cautious and avoid coding those bit settings not defined. Undefined codes may be assigned new functions in the future.

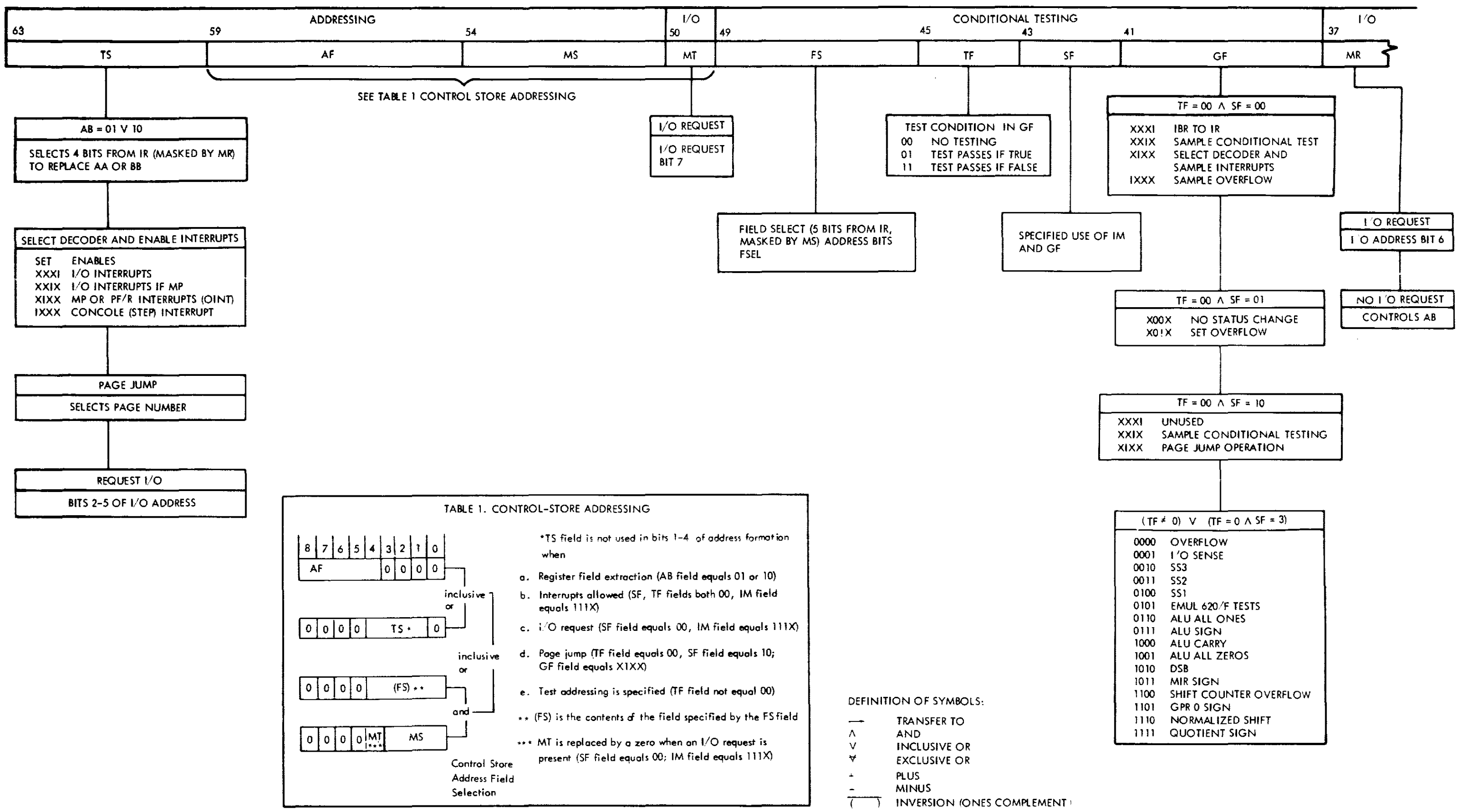
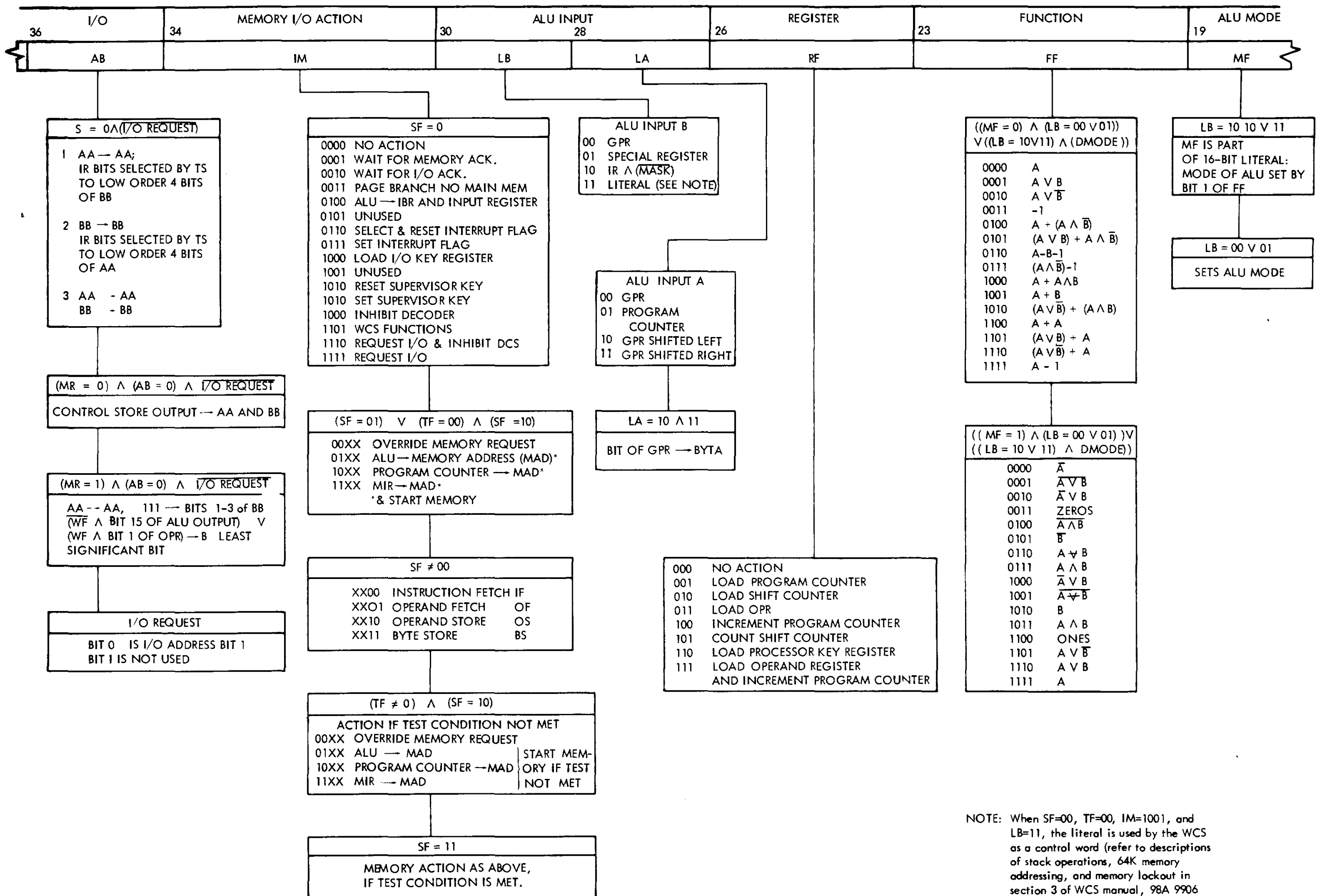


Figure 2-1. Microinstruction Fields (1 of 3)

VTI3-295 A



NOTE: When SF=00, TF=00, IM=1001, and LB=11, the literal is used by the WCS as a control word (refer to descriptions of stack operations, 64K memory addressing, and memory lockout in section 3 of WCS manual, 98A 9906 082).

Figure 2-1. Microinstruction Fields (2 of 3)

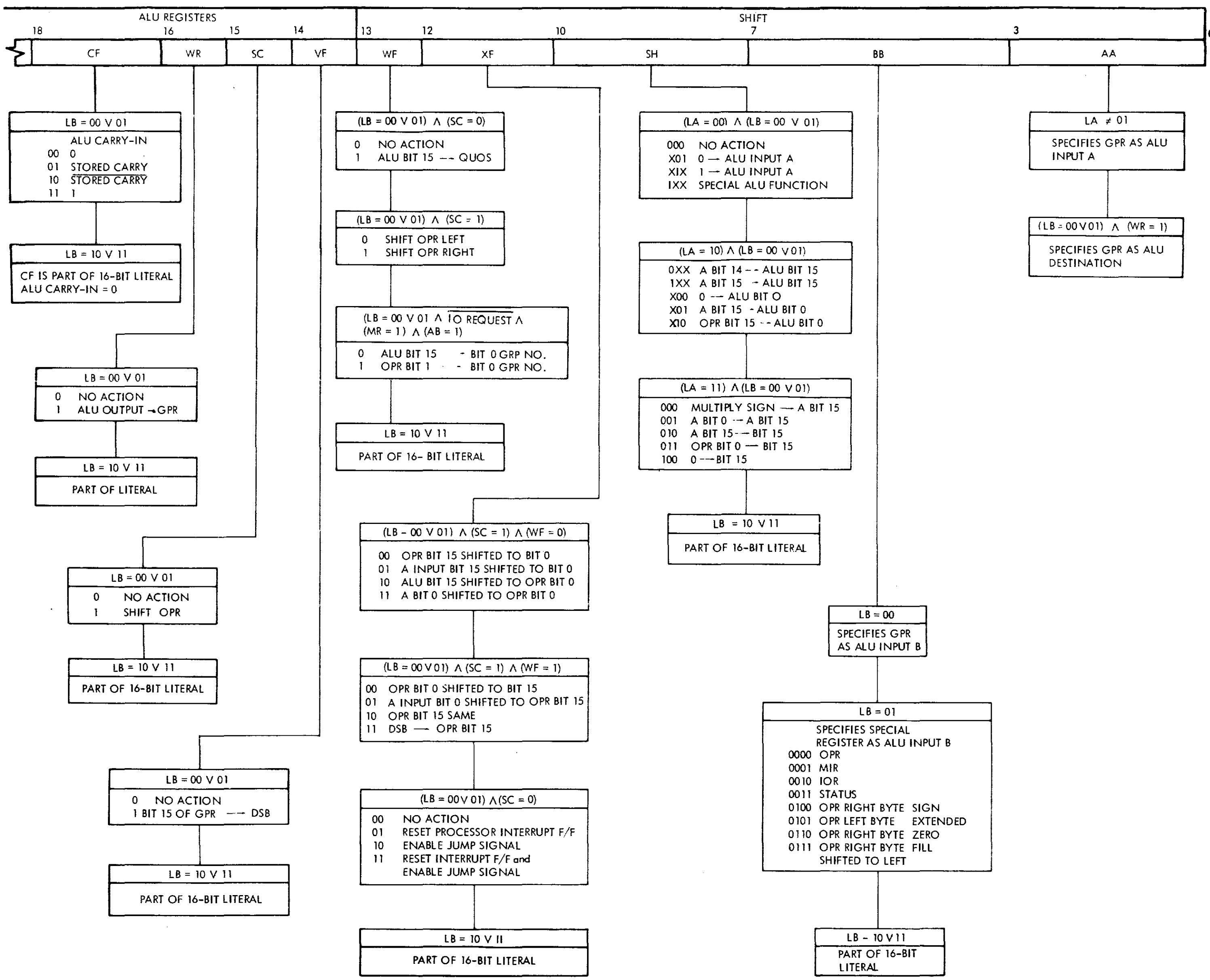


Figure 2-1. Microinstruction Fields (3 of 3)

VT13-297 A

THIS PAGE
INTENTIONALLY LEFT
BLANK

2.2 DATA TRANSFER AND TRANSFORMATION

2.2.1 ALU Input Sources

Input to the arithmetic and logic unit (ALU) is selected by a combination of fields. The ALU receives two inputs, A and B. Two buses can move information to the ALU but the same sources are not available for both buses. Some inputs to the ALU can be sent on either bus and some on both. The general-purpose registers can be selected as input upon either bus and a specific register selected for each bus.

Any of the general-purpose registers can be shifted on its way on the A bus to the ALU. Shifting can be one bit position to the left or right.

Input to the ALU can be from one or two of the general-purpose registers. The use of one of these registers is indicated by setting field LA to zero for input on the A bus, and LB for input on the B bus. The specific register is specified in AA and/or BB.

For example to use registers R2 and R4 as the input to the ALU

field	LB	LA	BB	AA
value (hex.)	0	0	2	4
Mnemonic	B$GPR	A$GPR	R2	R4

LA can also specify that the register indicated by AA is shifted or rotated. Shift left and shift right are indicated in the LA field and the shift field, SH.

Special Registers as ALU Input

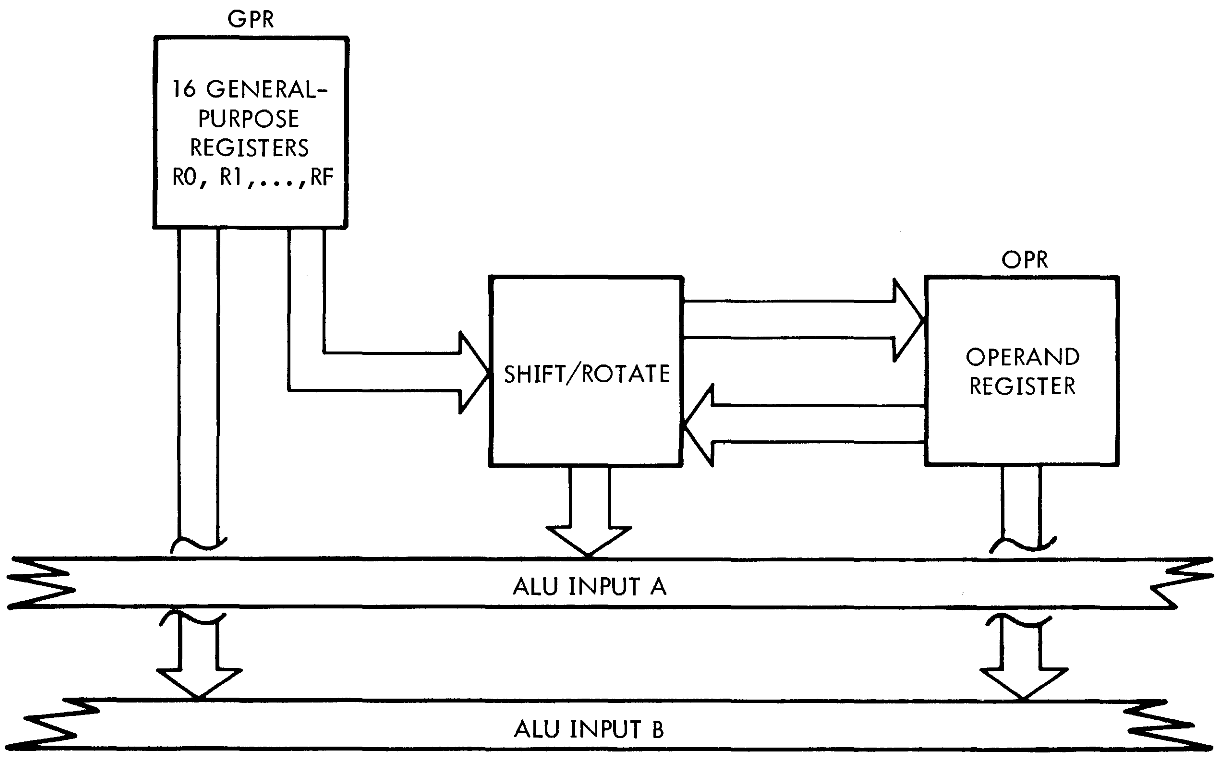
By setting the LB field to one, SREG for special register the value in the BB field takes on a different meaning:

0	OPR	Operand register
1	MIR	Memory input register
2	IOR	I/O register
3	STAT	Processor status word
4	ORSE	Operand right byte sign extended
5	OLSE	Operand left byte sign extended
6	ORZF	Operand right byte zero fill
7	OLZF	Operand right byte in the left byte position zero fill

Table 2-1. ALU Input A Bus Selections

ALU Input A Bus Source	Fields: LA	SH	LB
Program counter	01	XXX	XX
General-purpose register (any one of 16) specified in AA	00	Neither X01 nor X1X	0X
General-purpose register (any one of 16) specified in AA	00	XXX	1X
All zeros input	00	X01	0X
All ones input	00	X1X	0X
General register (in AA) shifted left	10	See below	0X
Bit 15 = register bit 14		0XX	
Bit 15 = register bit 15		1XX	
Bit 00 = zero		X00	
Bit 00 = register bit 15		X01	
Bit 00 = operand register bit 15		X10	
General register (in AA) shifted right	11	See below	0X
Bit 15 = multiply sign flag		000	
Bit 15 = register bit 00		001	
Bit 15 = register bit 15		010	
Bit 15 = operand register bit 00		011	
Bit 15 = zero		100	

X indicates the bit in that position is of no consequence to this action.



VTII-1802

Figure 2-2. General-Purpose Registers, Operand Register and ALU Input

Table 2-2. ALU Input B Bus Selections

ALU Input B Bus Source	LB	BB
General-purpose register (any one of 16)	00	Specifies register
Operand register full word	01	0000
Operand register right byte with sign extended	01	0100
Operand register left byte with sign extended	01	0101
Operand register right byte with zeros in left byte	01	0110
Operand register right byte in left byte position; zeros in right	01	0111
Memory input register (MIR)	01	0001
I/O register (IOR)	01	0010
Processor status word (STAT)	01	0011
Instruction register masked by 16-bit literal constant consisting of fields MF, CF, WR, SC, VF, WF, XF, SH and BB. A one in the mask fields forces the corresponding ALU input bit to a zero.	10	Part of mask
16-bit literal constant consisting of the ones complement of fields MF, CF, WR, SC, VF, WF, XF, SH and BB	11	Part of constant

NOTE: When the 16-bit literal or mask is used, the ALU mode is forced to the arithmetic mode if the FF field bit 1 is a zero and to the logical mode if the FF field bit 1 is a one. A carry of zero is forced. The ALU output may not be written into any general register in this case. The WR field, which would specify such an operation is disabled for use as part of the 16-bit literal or mask.

Processor Status Word

The processor status word may be applied to the ALU input B bus when the LB field equals 01 and the BB field equals 0011. Processor status bits are assigned as follows:

Bit	Function	Name
00	Not used (logic 1)	
01	Supervisor mode flag	SUPR
02	ALU zero flag	ALUZ
03	Shift counter bit 00	
04	Shift counter bit 01	
05	Shift counter bit 02	
06	Shift counter bit 03	
07	Shift counter bit 04	
08	Overflow flag	OVFL
09	ALU all ones flag	ALUO
10	ALU sign flag	ALUS
11	ALU carry flag	ALUC
12	Processor key register bit 0	
13	Processor key register bit 1	
14	Processor key register bit 2	
15	Processor key register bit 3	

2.2.2 ALU Functions

Two sources for data, an action to be performed by the arithmetic and logic unit and a destination for the result are all specified in a single microinstruction.

The ALU function is determined by three fields in microinstruction. These fields, function, mode and carry, are grouped together to give meaningful names to some common operations, like ADD for addition. This entire group of fields can be set to execute combinations which do not have convenient names in the assembler.

One basic ALU action or an operator is chosen. There are three categories of operations. Arithmetic operations available at this level include addition, subtraction, increment etc. Logical operators which have convenient single-word names are AND, OR, exclusive OR, NOT implication and equivalence. Also the ALU can perform more complicated logical functions explained later in this section.

Table 2-3 lists some of the more common arithmetic and logical operations and the corresponding fields.

Table 2-3. ALU Operations

Assembler Mnemonic	ALU Action	FF	MF	CF
ZERO	all zeros	0011	1	00
ØNES	FFFF	1100	1	00
TRNA	A	1111	1	00
TRNB	B	1010	1	00
INCA	A + 1	0000	0	11
DECA	A − 1	1111	0	00
ADD	A + B	1001	0	00
SUB	A − B	0110	0	11
SHFA	A + A	1100	0	00
AND	$A \wedge B$	1011	1	00
OR	$A \vee B$	0001	0	00
EOR	$A \veebar B$	0110	1	00
NOTA	$\bar{A}$	0000	1	00
NOTB	$\bar{B}$	0101	1	00

*cannot be used when input B is mask or literal

ALU Mode

There are two modes available for the ALU, arithmetic and logical. In the arithmetic mode the user selects a type of carry input to the ALU to be used with the arithmetic action. In logical functions the value of the carry field (CF) is ignored. The mode is directly set as either arithmetic or logical by the MF field. Indirectly the mode can be set when the actual mode field is part of a literal or literal mask. If the LB field is 10 or 11 in binary, the MF and CF fields are part of a 16-bit constant. In this case the ALU mode is taken from the bit 1 setting of the FF field (consequently this limits the functions available with a literal or mask).

Carry Flag

The CF field specifies carry input to the ALU as follows:

CF	Value of Carry In
00	Zero
01	Stored carry
10	Stored carry complement
11	One

The carry flag ALUC, bit 11 of STAT, is altered only if SF is set to zero or two, TF to zero and the GF field to XX1X. Under these conditions carry is set or reset to the carry produced by the ALU. The only meaningful conditions for carry are the arithmetic functions such as add, increment, decrement and subtract. For these conditions the carry flag is set as follows. MF is zero for all of the following.

Table 2-4. Carry Flag Settings

FF	Function	If Carry In = 0	If Carry In = 1
0000	A	Reset	Set if result = 0
0001	$A \vee B$	Reset	Set if result = 0
0010	$A \vee \bar{B}$	Reset	Set if result = 0
0011	-1	Reset	Set unconditionally
0100	$A + (A \wedge \bar{B})$	X	X
0101	$(A \vee B) + (A \vee \bar{B})$	X	X
0110	A-B-1	Set if $[(A_{15} = B_{15}) \wedge (A \geq B)] \vee$ $[(A_{15} \neq B_{15}) \wedge (A < O)]$	Set if $[(A_{15} = B_{15}) \wedge (A > B)] \vee$ $[(A_{15} \neq B_{15}) \wedge (A < O)]$
0111	$(A \wedge \bar{B}) -1$	Set if result is $\neq -1$	Set unconditionally
1000	$A + (A \wedge B)$	X	X
1001	$A + B$	Set if $[(A < O) \wedge (B < O)]$ ✓ $[(A_{15} \neq B_{15}) \wedge$ $(A_{15} = O)$ / $(\lvert A \rvert \geq \lvert B \rvert)] \vee$ $[(A_{15} \neq B_{15}) \wedge$ $(B_{15} = O) \wedge$ $\lvert B \rvert \geq \lvert A \rvert)]$	Set if $[(A < O) \wedge (B < O)] \vee$ $[(A_{15} \neq B_{15}) \wedge (A_{15} = O) \wedge$ $(A \geq B)] \vee$ $[(A_{15} \neq B_{15}) \wedge (B_{15} = O) \wedge$ $(B \geq A)] \vee [\text{Result} = O]$
1010	$(A \vee \bar{B}) + (A \wedge B)$	X	X
1011	$(A \wedge B) -1$	Set if result $\neq -1$	Set unconditionally
1100	$A + A$	Set if $A_{15} = 1$	If $A_{15} = 1$
1101	$(A \vee B) + A$	X	X
1110	$(A \vee \bar{B}) + A$	X	X
1111	$A - 1$	Set if result $\neq -1$	Set unconditionally

Arithmetic Operations

The FF field determines an arithmetic operation as indicated below when the MF field is 0. Carry input is set independently. When bit 1 of FF is zero the arithmetic mode is selected when the actual mode field is part of a mask or literal. The expressions in parentheses are evaluated first from left to right. Any further evaluation is performed from left to right.

Logical Operations

When MF is one, the logical operations occur as indicated below by FF field settings. The carry field is ignored. Symbol indicates exclusive OR operation.

Arithmetic Functions

FF Value	ALU Action
0	A
1	$A \vee B$
2	$A \vee \bar{B}$
3	All ones
4	$A + (A \wedge \bar{B})$
5	$(A \vee B) + (A \wedge \bar{B})$
6	$A - B - 1$
7	$A \wedge \bar{B} - 1$
8	$A + (A \wedge B)$
9	$A + B$
A	$(A \vee \bar{B}) + (A \wedge B)$
B	$(A \wedge B) - 1$
C	$A + A$
D	$(A \vee B) + A$
E	$(A \vee \bar{B}) + A$
F	$A - 1$

SYMBOLS

$\vee$	Inclusive OR
$\veebar$	Exclusive OR
+	Addition
−	Subtraction
	logical AND
$\bar{c}$	complement

Logical Functions

FF Value	ALU Action
0	A
1	$\overline{A \vee B}$
2	$\bar{A} \wedge B$
3	All zeros
4	$\overline{A \wedge B}$
5	$\bar{B}$
6	$A \veebar B$
7	$A \wedge \bar{B}$
8	$\bar{A} \vee B$
9	$\overline{A \veebar B}$
A	B
B	$A \wedge \bar{B}$
C	All ones
D	$A \vee \bar{B}$
E	$A \vee B$
F	A

2.2.3 ALU Output Destinations

The ALU output will be determined by the function performed. This data can be directed by the microinstruction to the general-purpose registers, some of the special registers, counters, and indirectly to memory and I/O.

A multiple destination can be one of the general-purpose registers and a special register.

The direct assignments of the ALU result is specified by a combination of fields, WR, LB, AA and RF. The first three are used to specify any one of the 16 general-purpose registers while RF selects sending data to the program counter, operand register, shift counter or key register.

Table 2-5. ALU Output Data Destination

Destination	Control Fields				
	RF	WR	SF	IM	LB
DIRECT CONTROL					
General register (any 1 of 16) (Specified in AA)		1			0X
Program counter	001				
Operand register	011 or 111				
Shift counter	010				
Processor key register	110				
INDIRECT MEMORY CONTROL					
NOTE: Transfer occurs only if cycle is successfully initiated)					
Memory data bus			Not 00	XX1X	
Memory address register			Not 00	01XX	
Memory input register and instruction buffer			00	0100	
INDIRECT I/O CONTROL					
I/O register			00	111X	
NOTE: Transfer is under direct control of I/O control. Operation is specified by TS, AB, MR fields and contents of I/O control store.					

2.2.4 Other Registers

Shift Counter

The shift counter is an 8-bit counter which may be incremented and tested independent of the ALU. It is thus useful in keeping track of iteration in a microprogram. The counter may be tested for overflow using test addressing. The overflow condition occurs when the shift counter is minus one.

An instruction which both increments and tests the shift counter will be testing the old value. If the counter is loaded with negative number and incremented to 0, the one instruction delay is no problem. This is because checking the old value for − 1 produces the same result as checking the new value for zero.

Program Counter

The program counter is a 16-bit register which can be incremented and/or used as a memory address, independent of the ALU. The following are considerations when incrementing the program counter:

a. if the same microinstruction uses the P register for a memory address, the new value of P will be used.

b. if the microinstruction both increments P and uses P as an ALU input, unpredicatable results are obtained. In general, using P as an ALU input and incrementing P should not be done in the same instruction.

Processor Key Register (KEY)

A 4-bit processor key register supplies signals for memory operations initiated by the processor. These four bits in conjunction with the high-order bits of the normal memory address are used by the memory map option determine physical addresses. It should be noted that this key register is different from the map register used under VORTEX II. The latter is loaded over I/O and cannot be conveniently accessed from the micro level.

I/O Key Register

A similar key register for I/O is a 4-bit register which supplies signals to the memory map option during memory operations initiated by the I/O control.

Operand Register

The operand register is a 16-bit register which has special shifting abilities. As previously noted, the ALU input A bus may have any of the 16 general-purpose applied shifted left or right one-bit position. In addition, the operand register may be shifted left or right independently or in conjunction with shifting of any general register. This can occur any time the 16-bit literal or mask is not in use.

When the LB field is equal to 0X (no literal/mask) the SC, WF and XF fields define operand register shifting.

When the SC field equals 0 no shifting takes place. When the SC field equals 1, the operand register is shifted left if the WF field equals 0 and right if the WF field equals 1.

For left shifts the next contents of the operand register bit 00 is specified by the XF field. If XF equals 00 operand register bit 15 is copied to bit 00 to permit independent circular shifting. If XF equals 01 bit 15 of the general register specified by the AA field is copied to bit 00.

This permits double-length circular shifting. If XF = 10 the complement of the ALU output bit 15 is copied to bit 00. If XF = 11 the operand register bit 00 is set to zero.

For right shifts the next contents of the operand register bit 15 is specified by the XF field. If XF equals 00 operand register bit 00 is copied to bit 15 to permit independent circular shifting. If XF equals 01 bit 00 of the general

Table 2-6. Operand Register Shift Operations

Control Field

	LB	SC	WF	XF
No shifting		0		
No shifting	1X			
Shifting of operand register	0x	1		
Left shifting			0	
Bit 00 = operand register bit 15				00
Bit 00 = general register bit 15 (specified in AA)				01
Bit 00 = ALU bit 15 complement				10
Bit 00 = zero				11
Right shifting			1	
Bit 15 = operand register bit 00				00
Bit 15 = general register bit 00 (specified in AA)				01
Bit 15 = operand register bit 15				10
Bit 15 = SHFT (shift flag)				11

register specified by the AA field is copied to bit 15 to permit double-length circular shifting. If XF equals 10 the operand register bit 15 is maintained at its current state to permit independent arithmetic shifting. If XF equals 11 the shift flag (SHFT) is copied to bit 15.

2.3 ADDRESSING

2.3.1 General

Executing instructions in an order other than strictly sequential gives programs flexibility and compactness. The ways in which the order of microinstructions can be varied are similar to those used in assembly-language programs. For the microassembler the usual order of execution takes the next instruction -- the contents of word five after word four and so on -- unless a jump or branch specifies the change in order. In reality each and every microinstruction specifies the next one to be executed, but usually the assembler constructs sequential-execution addressing automatically.

A jump in a microprogram can be a conditional action based on the true or false state of flags or signals in the system. In microinstructions the jump is not a separate instruction but the sampling and/or testing and the branch itself are specified in fields of a microword. In addition to conditional and unconditional branches, the branch may be from one page to another. The page jump is described following a few simpler cases and conditions.

Three basic types of addressing create the address of the next microinstruction to be executed. Normal addressing is the simplest case. The next address is specified by the current microinstruction. Field-selection addressing uses an instruction register field to specify the address for the next microinstruction. In decoding addressing (using the decoder control store) the instruction buffer specifies the next address (section 8 in this manual describes the use of this feature).

Three other types of addressing are similar to the basic types. Conditional addressing uses testing of various conditions to choose one of two addresses. The page jump can specify both the page and word number within the page for the next microinstruction. Interrupt addressing uses both the microinstruction and the system's interrupt logic to determine the next microinstruction.

2.3.2 Normal Addressing

Normal addressing is used to arbitrarily specify the next microinstruction address. No conditional testing is involved, no interrupts are active or they are disabled and decoder addressing is not specified. The FS and TS fields are set equal to 0000 and the MT field equals 0 so the low order address contribution (bits 0-3) is governed entirely by the MS field. The high order bits (4-8) are supplied by the AF field.

8	7	6	5	4	3	2	1	0
AF					MS			

Control Store Address --
Normal Addressing

No reset
No interrupts
No decoding
FS = 0000
MT = 0
TS = 0000 or
TF = 0

Normal Addressing with TS Field

The TS field may be used to form bits 1 through 4 of the control store address when none of the following conditions is true:

a. Register field extraction (AB field equals 01 or 10)

b. Interrupts allowed (SF and TF field both 00; GF field equals X1XX)

c. I/O request (SF field equals 00; IM field equals 111X)

d. Page jump (TF field equals 00; SF field equals 10; GF field equals X1XX)

The address is formed by the inclusive OR of the TS field into bits 1 through 4 of the address obtained with normal addressing (FS field equals to 0000; no decoding; no interrupts, MT field equals 0).

8	7	6	5	4	3	2	1	0
AF					MS			

inclusive or

0	0	0	0	TS				0

Control Store Address
Normal Addressing with
TS Field

2.3.3 Field Selection Addressing

The contents of the instruction register and a number of processor flags may be used to form a control store address. Any 1- to 5-bit contiguous field from the instruction register may also be used in forming the low-order five bits of control store address. Thus, up to a 32-way branch may be performed based on instruction register contents. This permits detailed instruction decoding. In addition, the interrupt flag, byte address flag, shift flag and console step mode may be selected to alter the control store address.

Field selection addressing is used any time the FS field is not equal to 0000. The field selection address contribution for all values of the FS field is shown in the tables below. Any bit of the field selection contribution may be forced to a zero by use of the MS and MT fields. The field masks bits 0-3 of the field select contribution. The MT field masks bit 4. A zero in any bit of the MS and MT fields forces the contribution of the corresponding field selection bit to zero. When an I/O request is issued (SF field equal to 00 and IM field equal to 111X) the MT field is used as part of the I/O operation specification. In this case, the MT field is ignored and bit 4 of the field selection address contribution is masked to zero.

The field selection address contribution is shown below for all values of the FS field.

High-order address bits 4 through 8 are provided by the AF field.

The TS field is logically ORed into the control store address bits 1 through 4 under the same conditions as normal addressing into TS field. Thus, the composite field selection address is formed as follows:

Control Store Address Bit					
4	**3**	**2**	**1**	**0**	**FS Field**
One	One	One	One	One	0
One	One	One	One	INT	1
One	01	One	SHFT	BYTA	2
One	One	One	One	STEP	3
04	03	02	01	00	4
05	04	03	02	01	5
06	05	04	03	02	6
07	06	05	04	03	7
08	07	06	05	04	8
09	08	07	06	05	9
10	09	08	07	06	A
11	10	09	08	07	B
12	11	10	09	08	C
13	12	11	10	09	D
14	13	12	11	10	E
15	14	13	12	11	F

Numbers 00 through 15 refer to instruction register bits

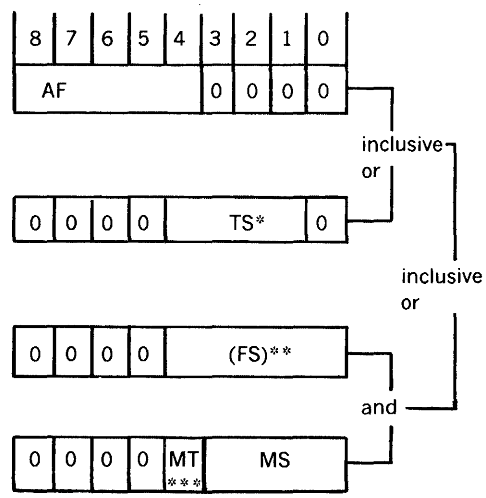
INT is the interrupt flag (complement)

BYTA is the byte address flag

SHFT is the shift flag

STEP is true when the console is in the STEP mode

Figure 2-3. Field Selection Address Contribution



Control Store Address Field Selection

* TS field is not used in bits 1-4 of address formation when:

a. Register field extraction (AB field equals 01 or 10)

b. Interrupts allowed (SF, TF fields both 00, IM field equals 111X)

c. I/O request (SF field equals 00; IM field equals 111X)

d. Page jump (TF field equals 00; SF field equals 10; GF field equals X1XX)

e. Test addressing is specified (TF field not equal 00)

** (FS) is the contents of the field specified by the FS field

*** MT is replaced by a zero when an I/O request is present (SF field equals 00; IM field equals 111X)

Normal addressing and normal addressing with TS field are a subset of the field selection addressing set, i.e., the FS field equals 0000 and the MT field equals 0.

2.3.4 Test Addressing

Two addresses must be specified when test operations are performed -- one for use if the test passes and one for use if it fails. Testing is specified whenever the TF field is not equal to 00. If the test is to pass when the condition tested is true, the TF field must be equal to 10. If the test is to pass when the condition tested is false, the TF field must be equal to 11. The condition to be tested is specified by the GF field.

The address used if the test passes is identical to that formed by field selection addressing. The address used if test fails is made up of the AF and TS fields as shown below.

8	7	6	5	4	3	2	1	0
AF					0	0	0	0

inclusive or

0	0	0	0	TS				0

Control Store Address -- Test Fails

2.3.4.1 Conditions

Whether or not a test is to be done and the way the test passes are indicated in the test field (TF). Testing is specified whenever the TF is not zero. If the test is to pass when the condition is true, the TF is equal to 10. If the test is to pass when the condition is false, the value of the TF should be 11.

The condition to be tested is specified in the GF field.

Summary of Conditions Mnemonics

Value of GF	Mnemonic for Assembler
0	OVFL
1	IOSR
2	SSW3
3	SSW2
4	SSW1
5	TFIR
6	ALUO
7	ALUS
8	ALUC
9	ALUZ
A	SHFT
B	MIRS
C	SFTC
D	GPRS
E	NORM
F	QUOS

Meanings and Use of Conditions

OVFL Overflow may be set and reset unconditionally: It may sample data-loop conditions. Automatically reset by system reset or microinstruction in which the GF value is TFIR and the instruction register bit 0 is set and the test met.

IOSR I/O Sense Response (discussed in I/O section)

SSW3, SSW2 and SSW1 Sense switches are set and reset only by manual manipulation on the control panel.

TFIR Test from instruction register which determines a set of conditions tested simultaneously. Nine bits of the instruction register cause the following tests:

0 Overflow
1 Positive/NOT bit
2 Negative/NOT bit
3 R0 of General-purpose registers
4 R1 of General-purpose registers
5 R2 of General-purpose registers
6 Sense switch 1
7 Sense switch 2
8 Sense switch 3

ALUO ALU all ones

ALUS ALU sign flag

ALUC ALU carry flag

ALUZ ALU all zeroes

SHFT Shift flag copies bit 15 of the general register specified in the AA field whenever the literal or mask is not being used and the VF value is 1. This flag may be shifted into the operand register bit 15. It may be tested by a microinstruction to cause a branch to either of two microinstructions.

MIRS Memory input register sign

SFTC Overflow of the shift counter

GPRS General-purpose register 0 bit 15 (sign)

NORM Normalize flag is set after any microinstruction which the ALU output bus bit 15 is not equal to bit 14. It will be reset after any microinstruction during which the ALU output bus bits 14 and 15 are the same.

QUOS Quotient flag copies bit 15 of the ALU output after a microinstruction in which the literal or mask is not being used and the WF value is right or 1 and SC field is zero.

MULS Multiply sign sets any microinstruction during which any of the following three conditions existed:

1. ALU output bit 15 and ALU input A bit 15 were both equal to 1
2. ALU output bit 15 and ALU input B bit 15 were both equal to 1
3. ALU input A bit 15 and input B bit 15 were both equal to 1.

This flag may be applied to the ALU input A bus during right shift operations

BYTA Byte address flag copies bit 00 of the general register specified by the AA field whenever a general-purpose register is specified as shifted input to the ALU input A bus. This flag may be used to determine the address of the next microinstruction and for memory byte store operations (SF not equal to zero and IM field equal XX11) determines which byte of the addressed memory location is to be altered. If BYTA equals zero, the left byte is selected. BYTA equal to one selects the right byte. BYTA is set or reset during the microinstruction rather than at the end.

A wide variety of flags are available for use in microprogramming. In general, they may be tested no sooner than the microinstruction after which they were set. In other words, a microinstruction which both changes a flag and tests will be testing the old value of the flag.

The conditions that cause a flag to be set depend on the particular flag. In addition some flags require that the microinstruction specify sampling before they will be set. For example, the ALU all zeros (ALUZ) flag will not be set unless the ALU is all zeros and sampling is requested.

The following table lists some of the major flags. ALUZ, ALUC, ALUS, and ALUO are sampled together by any microinstruction in which SF equals X0, TF equals zero, and GF equals XX1X.

Summary of flags requiring sampling for microprogrammed conditions.

Flag	Sampling
NORM	no
MULS	no
SHFT	yes
QUOS	yes
BYTA	no
OVFL	yes
ALUZ	yes
ALUC	yes
ALUO	yes
ALUS	yes

Table 2-7. Overflow Flag Control

OVERFLOW FLAG CONTROL

Operations	Fields				Conditions Bit 15 ALU Input		ALU Output
	TF	SF	GF	FF	AA	BB	
Set overflow	00	01	X01X				
Reset overflow	00	01	X10X				
Sample overflow	00	01	X11X				
(ADD)				1XXX			
SET					0	0	1
					1	1	0
DON'T SET*					1	0	X
					0	1	X
(SUBTRACT)				0XXX			
SET					1	0	0
					0	1	1
DON'T SET*					0	0	X
					1	1	X

Also, reset by system reset or a microinstruction specifying test of the 620/f test condition with the instruction register bit 00 on in which the test passes.

Overflow may be sampled to be set if SF = 00 and GF = 1XXX. It will not be reset even if no overflow exists.

* If set previously, overflow will remain set regardless of sampling conditions.

2.3.4.2 Addresses in Branches

The destination address when the test fails must be an even word address. The destination addresses of both the pass and fail conditions must be within 32 words of each other.

Procedure for Address Assignment

Following completion of a flowchart assignment of control store, address assignment may be performed. A useful procedure is:

1. Assign the microprogram entry addresses consistent with the desired format of the BCS instructions.

2. Assign addresses to microinstructions to be executed upon receipt of an interrupt. These addresses must be X XXXX 0111.

3. Assign addresses to all microinstructions to be executed following those using TEST ADDRESSING where the "test fails" condition prevails.

4. Assign addresses to all microinstructions to be executed by field selection addressing. If field selection specifies test of the interrupt, byte address, shift, or console step flags assign addresses to the microinstructions to be executed in accordance with the following restrictions:

	Flag On	Flag Off
Interrupt	X XXXXXXXX0	XXXXXXXX1
Byte Address	X XXXXXXXX1	XXXXXXXX0
Shift	X XXXXXXX1X	XXXXXXX0X
Console Step	X XXXXXXXX1	XXXXXXXX0

5. Recheck all field select and test addressing microinstructions for addressing consistency. Prepare a list of assigned addresses and corresponding microinstruction numbers labels (keyed to the flowchart) to avoid duplicate assignments.

6. Other microinstructions may have their addresses arbitrarily assigned by the programmer or the assembler.

2.3.5 Page Jump Addressing

The microinstruction specifies a branch to a location in another 512-word page by executing a page jump. In this case, a 13-bit address is generated which sets a new active page number and specifies an address within that page. The page number is specified by the TS field. The word address is specified by field select addressing.

12	11	10	9	8	7	6	5	4	3	2	1	0
TS				Address modified field select addressing								

Control store address
page jump

A Page Jump with memory is specified by the TF field equal to 00; the SF field equal to 10; and the GF field equal to X1XX.

A page jump without initiating a memory cycle is specified by setting the TF and SF fields to zero, and the IM field = 0011.

2.3.6 Interrupt Addressing

When interrupts are allowed and an interrupt is active in a class which is enabled by the TS field, the low-order four bits of the control store address are supplied by the interrupt logic and the high order bits from the AF field.

8	7	6	5	4	3	2	1	0
AF					IIA			

IIA is supplied by interrupt logic.

IIA is 7 for I/O interrupts and 1 for second tests of I/O interrupts after initiation of the I/O interrupt sequence.

The TS field enables interrupts whenever bits are set as follows:

Bit Set	Enables
0	I/O interrupts
1	I/O interrupts only if memory protection is installed
2	Memory protection interrupt
3	STEP, console step mode interrupt

2.4 MAIN MEMORY CONTROL

Memory access may be initiated in a microinstruction which indicates the type of operation and the address source. Main memory access includes the fetching and storing of data to and from the memory through the memory buses. Memory can either be the core or semiconductor variety (as distinct from the disc or drum storage often called rotating memory, which is accessed as a peripheral device through I/O facilities).

When a microinstruction initiates an access, the memory control section handles the complete operation. This permits the microprogram to initiate access to/from memory and perform other functions (ALU etc.) while the access actually occurs the microprogram can detect the completion of the memory access by specifying a wait for memory done.

Two different types of fetches can be requested. The instruction fetch (IF) moves the contents of a 16-bit word from main memory to the memory input register (MIR) and the instruction buffer (IBR). The operand fetch (OF) moves a 16-bit word to the memory input register and does not change the instruction buffer. Instruction fetches are usually used for fetching 16-bit macroinstructions for decoding from the IBR. The operand fetch is used for general data and address fetches. The microword which requests a fetch provides the address in main memory. After the request is made it is handled completely by memory control and requires no further actions in the following microinstructions.

Example of fetch sequence

n	n+1	n+2
request instruction fetch	wait for memory done	(data is ready for use in MIR)

Memory requests to store data are of two types. The first is the operand store (OS), which stores a 16-bit word in main memory. The second is the byte store (BS), which stores only an 8-bit byte. As with the fetch operations, the microinstruction which requests the store must furnish the main-memory address for the operation. Microinstructions following the request for a store must provide the data to be stored on the ALU until the memory operation is complete.

Example of store sequence

n	n+1	n+2
request store using P as address	R0 → ALU wait for memory done	(operation complete)

During operand stores, the memory data are derived from the ALU output. If the ALU input is from any of the 16 general-purpose registers and an arithmetic operation is specified for the ALU, incorrect parity data may be stored in memory. This situation can be avoided by using only logical ALU functions during operand stores; or by addressing the general-purpose register to the proper ALU input during the microinstruction that initiates the memory store cycle. Figure 2-4 is a coding example of an operand-store sequence using an arithmetic operation with a general-purpose register as the data source.

Completion of a memory operation is detected either with the wait-for-memory-done function or by requesting another memory operation. Wait-for-memory-done suspends microinstruction execution until the memory operation is complete. Requesting another memory operation has the same effect because microword cannot complete until its memory request is acknowledged by memory control and requests are not acknowledged until any previous request is complete.

Override

An active memory access may have the type of operation changed by the next microinstruction. By making an immediate change the immediately prior action is overridden. This can be conditional upon the result of the same test available for addressing (GF field).

Example:

Microinstruction Cycle n	Microinstruction Cycle n+1	Microinstruction Cycle n+2
Initiate memory store	memory store starts	memory store continues
	override possible	too late to override

Memory cycles may be initiated by microinstructions either unconditionally or depending on the results of a test.

2.4.1 Unconditional Cycle Initiation

A memory cycle is unconditionally initiated or overridden when the SF field equals 01 or if the SF field equals 10 and the TF field equals 00.

The IM field specifies the type of operation and the address source. Permitted operations are:

IM Value	Action
XX00	Read data from memory into the instruction buffer and memory input register (instruction fetch).
XX01	Read data from memory into the memory input register (operand or address fetch).

DAS CODING FORM

PAGE ____ OF ____

varian data machines
a varian subsidiary

PROGRAMMER | DATE | PROGRAM

LABEL | OPERATION | VARIABLE AND COMMENT FIELD | IDENTIFICATION

```
MICRØ1 GEN     /*,10(ØS$ALU),12(A$GPR),24(R7),14(TRNB),15(LØG),
              C11(B$GPR),23(R6),6(MEMC)
*
*THIS MICRØ INITIATES A STØRE MEMØRY CYCLE USING AN ADDRESS FRØM R6
*IT ALSØ PRE ADDRESSES R7 WHICH WILL BE USED IN THE NEXT MICRØ
*
MICRØ2 GEN     /*,6(SPEC),10(WAITMD),14(ADD),11(B$SPEC),23(MIR),
              C12(A$GPR),24(R7)
*THIS MICRØ PRØVIDES THE DATA TØ BE STØRED BY ADDING THE CØNTENTS
*ØF R7 TØ MIR
```

96A0090-000A

Figure 2-4. Coding Example of an Operand-Store Sequence

IM Value	Action
XX10	Write the full word output of the ALU into memory.
XX11	Write the byte from the ALU specified by the byte address flag (BYTA) into the corresponding memory byte. The other memory byte at the designated word address is unaffected. If BYTA is false, the left byte is written. If BYTA is true, the right byte is written.

BYTA, the byte address flag, copies bit 0 of the general register specified by the AA field whenever a general-purpose register is specified as shifted input to the ALU input A bus.

The operation may be changed by the following microinstruction by specifying the new operation with the IM field equal to 00XX. This permits, for example, conversion of a store cycle into a fetch or an instruction fetch into an operand fetch.

The data to be written to memory must be maintained at the ALU output by the microinstruction(s) following initiation until the cycle is complete.

The source to be used for loading the memory address register is specified as follows:

IM = 01XX	ALU output
IM = 10XX	Program counter
IM = 11XX	Memory input register

2.4.2 Conditional Cycle Initiation

A memory cycle may be initiated (or overridden) or not depending on the results of a test specified by the GF field. Conditions tested were described previously in the section of test addressing.

If the TF field is not equal to 00 and the SF field equals 10, the cycle will be initiated (or overridden) if the tested condition is false.

If the SF field is equal to 11, the cycle will be initiated (or overridden) if the tested condition is true.

In either case, the IM field specifies the operation to be performed and the address source to be used as described in the previous section.

2.4.3 Special Transfer

ALU output data may be transferred to the instruction buffer and memory input register by using the memory data bus. This does not involve activation of any memory module. To initiate this transfer the SF field must be equal to 00 and the IM field equal to 0100. The ALU output data must be set up by the initiating microinstruction and maintained for one more microinstruction.

2.4.4 Wait for Memory Done

The wait-for-memory-done function suspends microinstruction execution until memory control signals completion of central control's prior request. This function is SF = 0 and IM = 0001. If no central control has no prior request active, the wait-for-memory-done has no effect.

Table 2-8. Memory Operations

Function	Control Field SF	TF	IM
UNCONDITIONAL INITIATION	01		
or	10	00	
CONDITIONAL INITIATION			
Condition True	11		
Condition False (Condition Specified in GF)	10	Not 00	
EITHER			
Operation			
Read memory data into instruction buffer and memory input register			XX00
Read memory data into memory input register			XX01
Write ALU word output			XX10
Write ALU byte output			XX11
Address Source or Override			
Override operation			00XX
ALU output			01XX
Program counter			10XX
Memory input register			11XX
SPECIAL TRANSFER (ALU output to instruction buffer and memory input register)	00		0100

2.5 MICROPROGRAMMING EXAMPLE

General

As an example of instruction implementation using Varian microprogramming, the steps of a single-word addressing load accumulator LDA in the direct address mode will be traced.

SS1M

Initially the instruction pipeline is assumed to be empty so a new instruction must be fetched from main memory. The first microinstruction studied will be that obtained from control store location 13E (**all addresses are given in hexadecimal**). This location has the label **SS1M**, which is one of the microprogram's standard states.

The microinstruction fields at 13E are:

TS	AF	MS	MT	FS	TF	SF	GF
0000	01001	0010	0	0000	00	01	0000

MR	AB	IM	LB	LA	RF	FF	MF
0	00	1000	00	00	000	0000	00

CF	WR	SC	VF	WF	XF	SH	BB	AA
0	0	0	0	0	00	000	0000	0000

The function of this microinstruction is to initiate an instruction fetch from the memory address specified by the program counter. Note that the SF field equal to 01 specifies unconditional initiation of the memory cycle. The IM field specifies use of the program counter for an address source and the instruction buffer and memory input register as destinations for data received from memory. The FS, MT, TS and TF fields contain all zeros so normal mode addressing is specified. The next control store address will be 092. No other fields of the microinstruction are pertinent.

SS2M

Location 092 is another microprogram standard state labeled **SS2M**. It continues the process of filling the pipeline by initiating another instruction fetch using the incremented contents of the program counter.

The microinstruction fields at 092 are:

TS	AF	MS	MT	FS	TF	SF	GF
0000	00010	1101	0	0000	00	01	0000

MR	AB	IM	LB	LA	RF	FF	MF
0	00	1000	00	00	100	0000	0

CF	WR	SC	VF	WF	XF	SH	BB	AA
00	0	0	0	0	00	000	0000	0000

Again the SF field is equal to 01 and the IM field is equal to 1000 specifying another instruction fetch using the program counter. In this case, however, the RF field equals 100 specifying that the program counter will be incremented before it is used an address. This microinstruction will not be immediately executed as the previous microinstruction initiated memory activity and the memory interface will remain busy until the first instruction from memory is loaded into the instruction buffer and the memory input register. At the time, the current microinstruction completes and the next microinstruction from location 02D becomes active. Normal addressing occurs again due to FS, TS, MT and TF fields being zero. No other fields of the microinstruction are pertinent.

SS3M

Location 02D is another microprogram standard state labeled "SS3M". It causes decoding of the instruction fetched from memory while checking for interrupts. It also copies the instruction buffer into the instruction register to make room for the next instruction from memory.

The microinstruction fields at 02D are:

TS	AF	MS	MT	FS	TF	SF	GF
1110	01101	0110	0	0000	00	00	0101

MR	AB	IM	LB	LA	RF	FF	MF
0	00	0110	00	00	000	0000	0

CF	WR	SC	VF	WF	XF	SH	BB	AA
00	0	0	0	0	00	000	0000	0000

This microinstruction manipulates no data paths nor does it initiate any memory cycles. Its sole purpose is to check for interrupts and, if there are none, cause a branch to the required microsequence. The TF and SF fields are both equal to 00 and the GF field bit 0 is a one causing transfer of the instruction buffer to the instruction register. The GF field bit 2 is a one, thus enabling interrupts and decoder addressing. The TS field defines the interrupts which are enabled -- all except I/O interrupts unless the memory protect option is installed. The IM field specifies selection of the interrupt flag. If this flag were set, interrupts would be suppressed. The flag is reset by this microinstruction. If an interrupt were active and the interrupt flag had not been set, the next control store address would be 0DX where X designates the four bits supplied by the interrupt logic. This would produce a branch to the interrupt microprogram sequence.

Assuming no interrupts are present, the new control store address will be determined by the decoder logic. The instruction fetched from memory is assumed to be 10F9 (hexadecimal) or 010371 (octal). This is a V73 "LDA" instruction with direct addressing of location 00F9 (hexadecimal). The most significant four bits of the instruction buffer address the first decoder control store at location one. The next four bits address the second decoder control store at location 00. The decoder control store contents are:

1st decoder

Control store location 1	B12 = 1 B8-B0 = 110000010

2nd decoder

Control store location 0	A8-A0 = 010000000

Since B12 equals 1, the B8-B0 and A8-A0 address components are logically ORed to produce an address of 182.

SWA10

Location 182 contains the first microinstruction of the single word addressing sequence (SWA10) for the instruction fetched from memory. It forms the effective address by masking bits 00 through 10 from the instruction register. It also initiates the operand fetch.

The microinstruction fields at 182 are:

TS	AF	MS	MT	FS	TF	SF	GF
0000	10010	1111	0	0000	00	01	0000

MR	AB	IM	LB	LA	RF	FF
0	00	0101	10	00	011	1010

MF	CF	WR	SC	VF	WF	XF	SH	BB	AA
1	11	1	1	0	0	00	000	0000	0000

16-bit mask literal

The LB field equals 10 so the ALU B input bus will have the contents of the instruction register masked by the 16 bits of the MF, CF, WR, SC, VF, WF, XF, SH and BB fields (a zero in the mask enables the corresponding instruction register bit). The mask equals F800 so the low order 11 bits of the instruction are used.

The ALU mode is determined by the FF field (1010) in conjunction with the LB field (forces logical mode) resulting in an ALU function of the ALU = B.

The RF field equals 011 so the ALU output is copied into the operand register.

The SF field equals 01 so unconditional memory control is specified by the IM field (0101) to be fetch an operand into the memory input register using the ALU output for an address source. This microinstruction will complete when the memory cycle initiated by the microinstruction at 092 completes.

The FS, TS, TF and MT fields all contain zeros so normal addressing is used and the AF and MS fields specify the next control store address of 12F.

SWA20

Location 12F contains the second microinstruction of the single word addressing sequence (SWA20). It decodes bits 13-15 of the instruction register contents to determine the class of the single word addressing instruction.

The microinstruction fields at 12F are:

TS	AF	MS	MT	FS	TF	SF	GF
0000	11110	1100	1	1111	00	00	0000

MR	AB	IM	LB	LA	RF	FF	MF
0	00	0000	00	00	000	0000	0

CT	WR	SC	VF	WF	XF	SH	BB	AA
00	0	0	0	0	00	000	0000	0000

No data manipulation or memory control operations are performed by this microinstruction. It serves only to branch to the specific microsequence for the class of single-word addressing instruction contained in the instruction register. Field select addressing is used to perform this decoding (FS field is not equal to 0000). The FS field is equal to 1111 so the selected field is bits 11 through 15 of the instruction register. The composite address formation is illustrated:

	8 7 6 5 4 3 2 1 0
AF field contribution:	1 1 1 1 0 0 0 0 0
or =	1 1 1 1 0 0 0 0 0
TS field contribution:	0 0 0 0 0 0 0 0 0
Field selected from instruction register: (I = 10F9)	0 0 0 0 0 0 0 1 0
and =	0 0 0 0 0 0 0 0 0
Mask consisting of MT and MS fields	0 0 0 0 1 1 1 0 0

Final effective address produced by inclusive or

1 1 1 1 0 0 0 0 0

The address of the next microinstruction is then 1EO.

LDA1

Location 1EO is the first microinstruction specific to the **LDA** instruction (LDA1).

This microinstruction increments the program counter and initiates another instruction fetch from main memory.

TS	AF	MS	MT	FS	TF	SF	GF
0000	01011	0101	0	0000	00	01	0000

MR	AB	IM	LB	LA	RF	FF	MF
0	00	1000	00	00	100	0000	0

CF	WR	SC	VF	WF	XF	SH	BB	AA
00	0	0	0	0	00	000	0000	0000

The RF field equals 100 specifying that the program counter will be incremented during this microinstruction.

The SF field equals 01 so unconditional memory control is specified by the IM field (1000) to fetch an instruction into the instruction buffer and memory input register using the program counter for an address source. (Note that the program counter is incremented during the microinstruction so the new value will be used for the memory cycle).

Normal addressing is used to specify the next microinstruction address (TF, TS, FS, MT fields are all zero). The AF and MS fields define the address to be 0B5.

LDA2

Location 0B5 is the second microinstruction specific to the **LDA** instruction (LDA2). This microinstruction transfers the contents of the memory input register to the accumulator, R0; transfers the instruction buffer containing the next instruction to the instruction register to make room for the instruction whose fetch was initiated by the microinstruction 1E0; decodes the instruction buffer to determine the starting address of the next microsequence and checks for interrupts.

The microinstruction fields at 0B5 are:

TS	AF	MS	MT	FS	TF	SF	GF
1111	01101	0110	0	0000	00	00	0101

MR	AB	IM	LB	LA	RF	FF	MF
0	00	0110	01	00	000	1010	1

CF	WR	SC	VF	WF	XF	SH	BB	AA
00	1	0	0	0	00	000	0001	0000

The ALU B input is specified by the LB field (equal to 01) to be one of the special registers. The BB field (equal to 0001) defines the memory input register as the source.

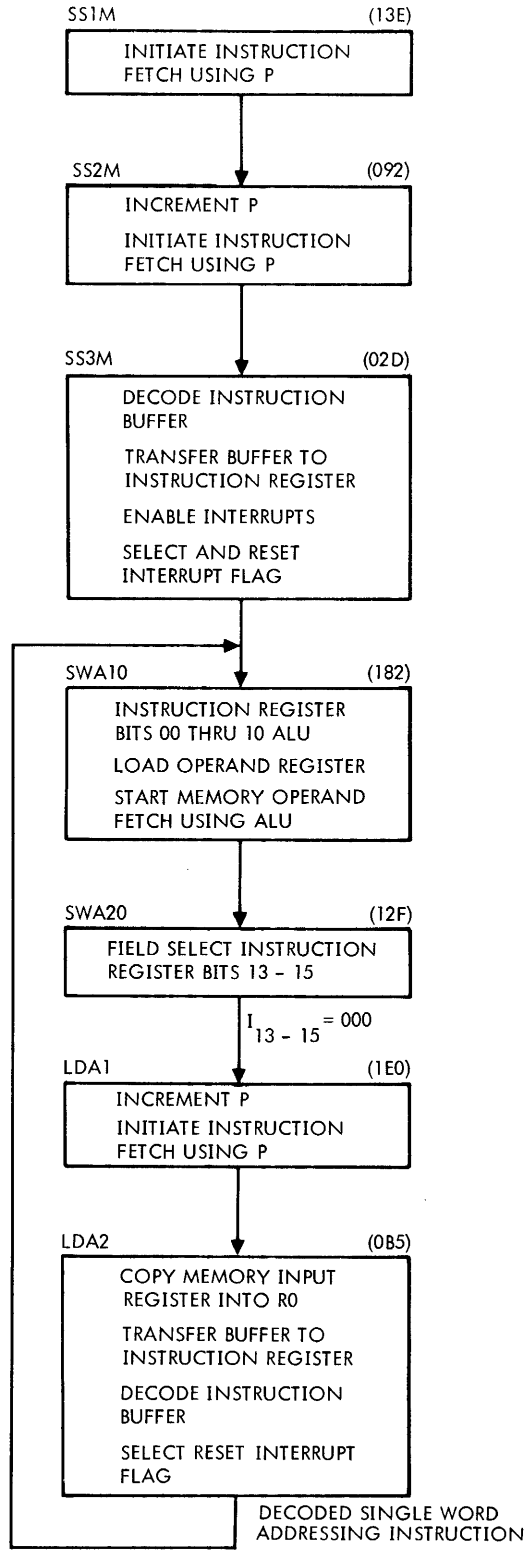
The ALU operation is specified to be in the logical mode (MF = 1) with the ALU output equal to the ALU B input (FF = 1010).

The WR bit equals a one so the ALU output data will be written into the register specified by the AA field (AA = 0000) which is the accumulator (A register). This is the execution phase of the LDA instruction.

The SF and TF fields are both equal to 00 and the GF field bit 0 is a one so the instruction buffer contents are copied into the instruction register. The GF field bit 2 is a one so the instruction decoder is enabled and interrupts are checked.

The IM field equal to 0110 with the SF field equal to 00 selects and resets the interrupt flag. If the flag is set, the decoded address and interrupts are suppressed and the next microinstruction is fetched from location 0D0. All interrupt classes are enabled as the TS field contains all ones. If an interrupt is active and the interrupt flag is off, only the decoded address is suppressed and the next microinstruction is fetched from the address specified by the AF field and the interrupt logic. This address is 0DX where X is the address supplied by the interrupt logic ($X \neq 0$).

If no active enabled interrupts exist, the next microinstruction will be fetched from the address specified by the



VTII-1938

Figure 2-5. Flowchart for LDA Instruction

	IDENT (HEX. ADDR.)	SS1M (13E)	SS2M (92)	SS3M (2D)	SWA10 (182)	SWA20 (12F)	LDA1 (1E0)	LDA2 (0B5)
MEMORY	FUNCTION		FETCH LDA	FETCH NEXT INST.	FETCH NEXT INST.	FETCH OPERAND	FETCH OPERAND	FETCH THIRD INST.
	REQUEST	IF	IF		OF		IF	
	ADDRESS	P	P		ALU		P	
ALU	INPUT A							
	INPUT B				I $\wedge$ 07FF			MIR
	OUTPUT				TRNB			TRNB
	DESTINATION							R0
STATUS	SAMPLE							
	TEST							
ADDRESSING	MODE			DECODE	FIELD SELECTION I13–I15			DECODE
	ADDRESS	SS2M	SS3M	FROM DECODER	SWA20	LDA1+X WHERE X = 0,4,8,...28	LDA2	FROM DECODER
OTHER	SPECIAL ACTIONS		INCP	ENABLE INTERRUPTS IBR→I			INCP	IBR→I ENABLE INTERRUPTS

NOTE:
Timing diagram shows the start-up and execution of a sequence of single-word addressing instructions (330 nanosecond memory cycle time is assumed).

VT11-2084

Figure 2-6. Flow Diagram of LDA Instruction

decoder control store logic. If the instruction buffer contains another single-word addressing instruction, the next address will be 182 (SWA10) and the sequence will be repeated.

Figures 2-5 and 2-6 show a flowchart and flow diagram of the microinstruction sequence described. Note that the pipeline effect of buffering instructions permits efficient use of the memory. (A 330-nanosecond semiconductor memory was assumed).

2.6 TIMING CONSIDERATIONS

Most microinstruction operations take place at the conclusion of the cycle. Certain exceptions do exist. ALU inputs are sampled at the midpoint in time of the cycle. Control-store address information, memory addresses, and most register and flag changes occur at the end of the microinstruction execution. The areas below should be considered while planning microprograms.

Program counter incrementation (RF = 100 or 111)
Incrementation takes place at the midpoint of the microinstruction. Thus the program counter value applied to the ALU input will not be the incremented value. The new value will be used as a memory address, if the program counter is specified as an address source.

Byte address flag
The byte address flag is set or reset at the temporal midpoint of the microinstruction. Thus its new value may be used to determine which byte of the memory location is to be altered.

Memory write operations
ALU inputs, function, mode and carry must be maintained constant throughout any memory write cycle. This is accomplished by specifying another memory cycle immediately following the current cycle thus interlocking execution of the next microinstruction with completion of the memory cycle in progress or by using the wait for memory done function (SF = 00, IM = 0001).

Special transfers
The transfer of ALU data to the instruction buffer and

memory input register requires ALU data to be maintained for two microinstructions.

I/O operations

If the I/O section is not idle when a new I/O operation is specified, microinstruction execution will not occur until the I/O becomes idle. A wait for I/O done function (SF = 00, and IM = 0010) will cause a similar wait condition until the I/O DN bit becomes true.

Use of the I/O register

If direct memory access or similar I/O operations are possible the I/O register may be altered. Care in use of this register is indicated. Control of the I/O register is described in the I/O section of this guide.

2.7 ADDITIONAL CAPABILITIES

2.7.1 Register Field Control

Many types of instruction words contain fields which specify registers which contain operand data. If all combinations of operations on all possible registers had to be specified by individual microinstructions, the control store size would be quite large.

A Varian 70 series system permits three- or four-bit fields to be selected from the instruction register and stored and maintained in the control-buffer-register specification fields. This permits a single microinstruction to handle all combinations of registers for any operation.

This register field extraction is performed independently of the field select addressing function and both may be used simultaneously.

The AA and BB fields of the microinstruction contained in control store are copied into their corresponding positions in the control buffer any time the AB field equals 00 and the MR field equals 0. This is the normal mode of operation.

When the SF field equals 00 and no I/O request is active, the AB field equals 01 or 10; the TS field specifies a four bit field of the instruction register to be loaded into the control buffer's AA or BB field. The field not being loaded will be loaded into the control buffer's AA or BB field. The field not being loaded will be maintained at its last value. A code of AB equals 01 and loads the field selected into the BB field. A code of AB equals 10 and loads the field selected into the AA field.

The MR bit is used to mask the most significant bit of the selected field. If MR equals zero, the most significant bit of the selected field will be treated as a zero. If MR equals one, the most significant bit of the selected field will be loaded into the designated field.

The AA and BB fields can be maintained in their current state by specifying and AB field equal to 11 while the SF field equals 00 and no I/O request is present.

If no I/O request is present, the AB field equals 00 and the MR field equals 1, the control buffer AA field will be maintained at its current value and the BB field will be forced to either of two addresses depending on data loop conditions and the WF field.

WF field equal to 1

Operand register bit 01 = 1; BB = 1111

Operand register bit 01 = 0; BB = 1110

WF field equal to 0

ALU bit 15 = 1; BB = 1111

ALU bit 15 = 0; BB = 1110

This function is used by the Varian 73 standard instructions microprograms for multiply and divide.

Register field control operations are summarized in the tables following.

Table 2-9. Register Field Control

Function	SF	AB	Control Fields MR	TS	WF
Load A and B fields from control store	00	00	0		
Inhibit loading of A field and place selected 4 bit field (masked) from instruction register into B field	00	01	Mask most significant bit of BB field	Selects field	
Inhibit loading of B field and place selected 4 bit field (masked) from instruction register into A field	00	10	Mask most significant bit of AA field	Selects field	
Inhibit loading of A and B fields	00	11			
Inhibit loading of A field and force B field to 1110 if ALU output bit 15 = 0 or to 1111 if ALU bit 15 = 1		00	1		0
Inhibit loading of A field and force B field to 1110 if operand register bit 01 = 0 or to 1111 if operand register bit 01 = 1		00	1		
All functions are inhibited if an I/O request is issued.					

Table 2-10. Register Field Selection

TS Field	Bits Selected From Instruction Register for register file			
000	03	02	01	00
001	04	03	02	01
010	05	04	03	02
011	06	05	04	03
100	07	06	05	04
101	08	07	06	05
110	09	08	07	06
111	10	09	08	07

Other Controls

Transfer instruction buffer to instruction register

The contents of the instruction buffer will be transferred to the instruction register when TF and SF both equal zero, and GF has a low-order bit set to 1.

Enable Jump Signal

A signal is sent to the memory-protection option designating a jump instruction by setting the LB high-order bit to zero and the SC field to zero and the XF field equal to 11 or 10. If the XF field equals 11, the interrupt flag will be reset.

Reset Interrupt Flag

The interrupt flag will be reset if the LB field equals 00 or 01 and the XF field equals 11 or 01.

Enable Special ALU Mode

(This feature is useful for the standard instruction set, but not generally suggested)

The ALU mode, carry input and overflow sampling may be forced according to the contents of the instruction register by setting the LA and LB fields equals to either 00 or 01 (high-order bit equals zero) and the SH high-order bit equal to 1. In this case, the ALU function will be as follows:

Bit	
3	As specified by FF field
2	most significant 2 bits
1	Instruction register bit 7
0	Instruction register bit 7 complemented

2.7.2 Memory Addressing to 64K

The standard instruction set has addressing capability to 32K words with 15-bit addresses. The use of bit 15 to select indirect addressing mode removes it from use as an address bit. The memory modules can recognize a 16-bit address which increases the range of addresses to 64K words.

The most significant bit of the memory address bus is normally grounded to prevent any address generated by the standard instruction set from attempting to access above 32K words. This is necessary since the high-order bit can be set by indirect memory reference in the host instruction set.

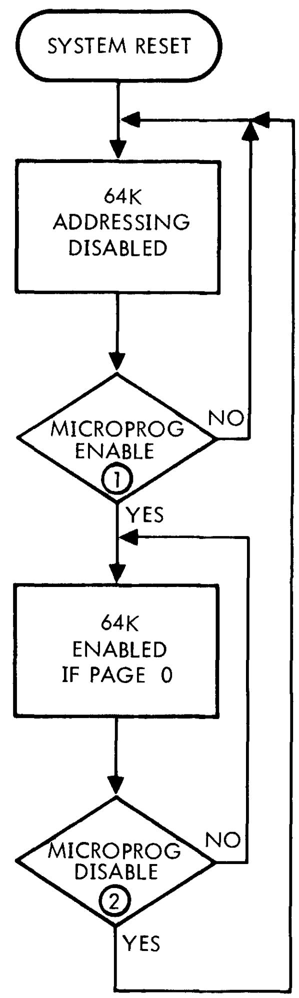
The WCS permits use of the full 16-bit addressing capabilities of a Varian 70 series system. This enabling is automatically inhibited while executing from page zero so standard 620 problems will execute correctly in the lower 32K words of memory.

User-written microprograms in the WCS can generate 16-bit addresses to cause access to the full 64K words. This mode is enabled or disabled with a group of control fields in the microinstruction. Once enabled this mode is retained until explicitly disabled as described below or a system reset occurs. The enabled mode is not effective when page zero is active.

64K Mode of Memory Addressing

Enable	Disable
SF = 0	SF = 0
TF = 0	TF = 0
IM = 1101	IM = 1101
LB = 11	LB = 11
MF = 1	CF = 11 or 10

Changing the memory mode requires all the conditions set as indicated. Figure 2-7 illustrates memory bus control.



(1) ENABLE = IM = 1101 $\wedge$ (T = 0) $\wedge$ (S = 0) $\wedge$ (LB = 11) $\wedge$ (MF = 1)

(2) DISABLE = (IM = 1101) $\wedge$ (T = 0) $\wedge$ (S = 0) $\wedge$ (LB = 11) $\wedge$ (C = 10V11)

VTII-1806

Figure 2-7. Flowchart of Memory Address Control

2.7.3 Memory Bus Lockout Status

Systems in which multiple processors share the use of common memory modules often require the capability of testing the contents of some memory locations and modifying those contents (if the results of the test indicate) without the possibility of another processor gaining access to that location between the test and the change.

WCS Implementation

The WCS permits use of a function allowing the processor it controls to temporarily lockout all memory modules connected to its memory bus. While the memory system is locked out on one port, no accesses are permitted on the other port. To prevent simultaneous lockout from both processors the lockout mode for any memory bus only becomes enabled when the requesting bus actually gains access to the memory (so the other bus cannot establish the lockout mode). The memory lockout mode is set or reset with the following microinstruction fields:

Field	Set LOCKOUT	Reset LOCKOUT
SF	0	0
TF	0	0
IM	1101	1101
LB	11	11
CF	X1	X0
AA	XXX0	XXX1

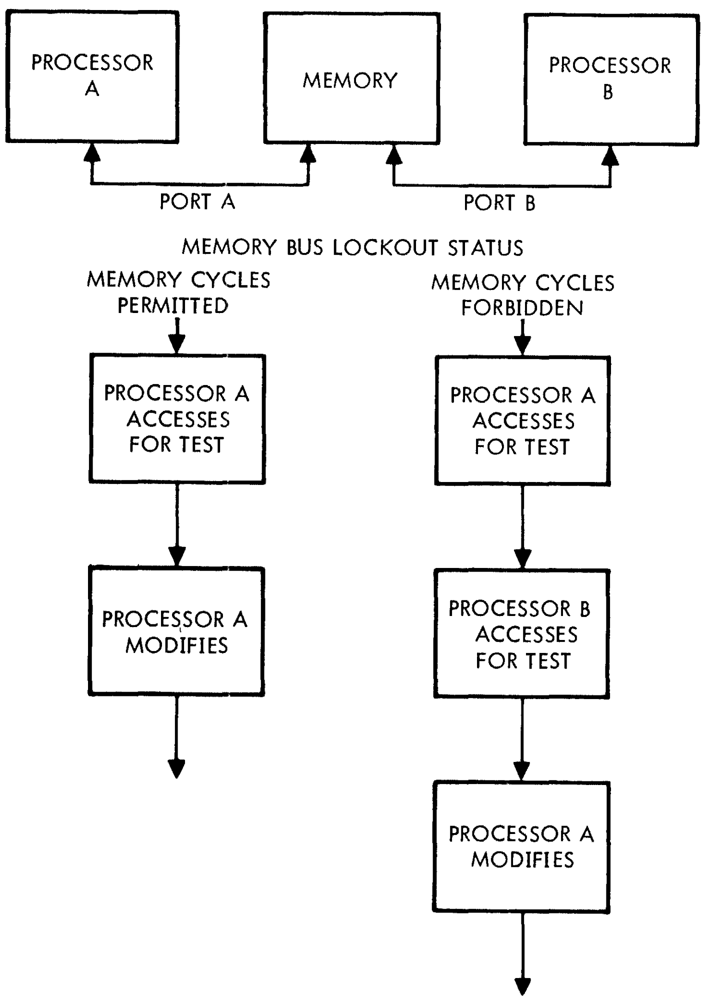
X indicates a bit position not involved in this operation.

If priority memory access (PMA) is present in the system, caution must be exercised to prevent the PMA from establishing its own lockout mode while either processor is in lockout mode. Simultaneous lockout would prevent all further accesses to memory and "lock-up" the system. Figure 2-8 illustrates memory bus lockout.

Lockout is removed by system reset.

2.7.4 Stack Use

Three stack operations, branch/push, branch/pop and branch/delete are used on the microprogram-return stack. All are global and effect a page selection. On the branch/push and branch/delete, the TS field gives the new page number. On the branch/pop, the word at the top of the stack gives the new page number. The return address which is pushed is an independent 13-bit specification



VTII-1808

Figure 2-8. Memory Bus Lockout

provided by mask field of microinstruction from the destination of the branch. The 13-bit specification is made up from the following fields of the microinstruction:

PAGE				Word		
12	11	10	9	8 7	6 5 4	3 2 1 0
WR	SC	VF	WF	XX	SH	BB

All stack operations have a value of zero for the SF and TF fields, IM set to 1110 and LB set to 3. Push requires bit 1 of the AA field set to 1. Pop is designated by bit 2 of the AA field set to 1 and bit 0 of the BB field set to 0. Branch/delete is the same as branch/pop except bit 0 of the BB field is set to 1.

	TF	SF	IM	LB	AA	BB
Branch/push	0	0	D	3	bit 1 = 1	
Branch/pop	0	0	D	3	bit 2 = 1	bit 0 = 0
Branch/delete	0	0	D	3	bit 2 = 1	bit 0 = 1

In initializing the stack an error branch can be pushed into the first location. If a microinstruction tries to "pop" this return, an underflow condition will occur and the error branch will be taken. An attempt to "push" one more level than the sixteen allowed causes a branch to the address at stack location zero.

In addition to pop and push operations on the stack, a stack entry delete operation is provided. This causes a page branch to the address specified by the processor and deletes one entry from the top of the stack.

All stack return addresses including the error return are restricted to the WCS. This avoids conflicts with processor-generated addresses during the pop operation.

Questions and Answers About Microprogramming Stack

Q: The WCS stack push and pop operations do not appear to be mutually exclusive. If both are specified, would the stack first pop the new address then push the return address?

A: Such an operation is undefined and should be avoided.

Q: Do micro stack operations proceed at full speed?

A: The stack operates at the same speed as other writable control store operations -- 190 nanoseconds.

2.7.5 Memory Addressing Using the Optional Memory Map

The memory-map key register (used by VORTEX II) cannot be easily modified from the WCS. As an option, the memory map can be wired to operate with the processor key register. This mode is not supported by standard Varian software. The following paragraphs describe this special mode of operations.

The processor key register is four bits which may be applied to the ALU input bus B as part of the status word. It is loaded from ALU output bus bits 12-15 and applied to the memory address bus as a four-bit extension to the 15-bit memory address register. The key register provides bits 15-18.

18 17 16 15	14 . . . 0
key register	Memory Address Register

memory map input
19 bits

when 64K mode is enabled, bit 15 of the memory address register is also ORed into the effective map input bit 15.

During memory cycles initiated by I/O (DMA), the I/O key register is applied instead.

Care must be taken in using the processor key register as an input to the ALU input bus B. No I/O initiated memory bus activity must take place during application of the status word or the value of the I/O key register may be used instead of the processor key register.

2.7.6 Memory Protection

If the memory protection is enabled, write operations are automatically inhibited. A memory-protection internal interrupt is generated as well as an I/O interrupt request. The memory-protection option may be disabled only by appropriate I/O instructions, not by microinstructions. Care must be taken in using the memory protection if more than 32K words of memory are to be addressed (bit 15 of memory address is enabled). Such use is very specialized and should only be undertaken after consultation with Varian Data Machines.

2.7.7 Address Comparator Logic

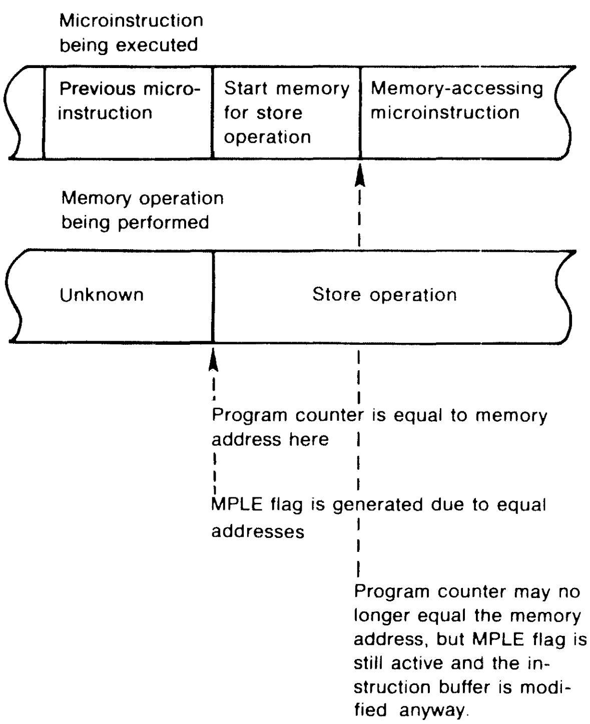
Address comparator logic is provided in Varian 70 series processor to prevent erroneous operation in the event a store instruction stores data into the next memory location in the program (macro). Erroneous operation would occur because the processor fetches the contents of the next memory location (n + 1) before the execution of the current instruction (at location n) is completed. The comparator logic compares the address from the program counter with the address from the memory address lines. If the addresses are equal, the comparator logic generates an equal-address flag (MPLE) which enables the memory contents already fetched into the processor's instruction buffer to be updated to the new contents stored by the store instruction.

A store instruction can thus cause a dynamic alteration to the original program flow. An example where this dynamic alteration would be useful is in forming a BCS macroinstruction in which the address is located in the A register and the operation code is located in a memory location. The A register is combined with the memory location to produce the BCS macroinstruction. By using the STA instruction with direct addressing into location n + 1, the A-register contents are stored in location n + 1 and are processed as the next instruction in the program.

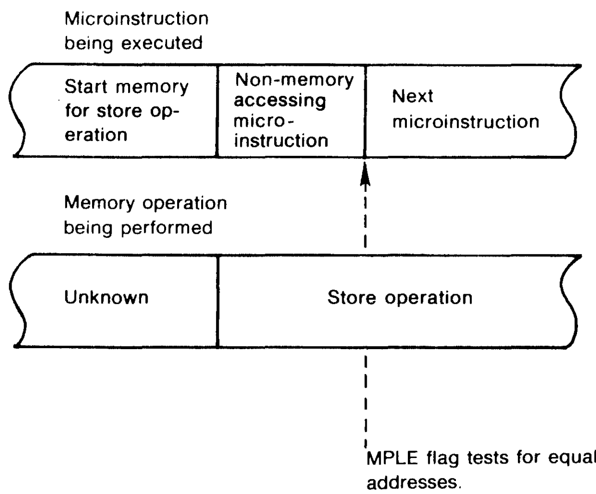
The following items should be considered when microprograms involving a store instruction are written:

a. The instruction buffer is modified if the address in the program counter equals the address on the memory address lines and a non-memory accessing microinstruction is executed during the store operation (no back-to-back memory operations).

b. The instruction buffer is modified if the address in the program counter equals the address on the memory address lines and either a memory accessing microinstruction or a wait-for-memory done condition follows the store operation (back-to-back memory operations). This type of operation is shown in the diagram below:



c. If microprograms are written for a user-defined macroinstruction set and dynamic program alteration occurs, all store operations should be followed by a non-memory accessing microinstruction so that the MPLE flag can test for equal addresses. Any modification to the program counter during execution of the store operation should be avoided. This type of operation is shown in the diagram below:



2.8 QUESTIONS ABOUT MICROPROGRAMMING CAPABILITIES

Q: If a current memory cycle is to alter the memory input register, and the memory input register is specified as the memory address source by the current microinstruction (awaiting memory cycle completion), are the old or new contents of the memory input register used for the next cycle's address? Does the situation change if the memory input register is an ALU input and the ALU is selected as an address source? Does the WCS clock rate affect this?

A: The new value of the memory input register is used when the memory input register is used as an address source. The memory input register should not be used through the ALU to determine the address of the next memory cycle when it can be altered by the current memory cycle. The WCS clock rate does not affect this.

Q: What is the standard entry point to branch to when an interrupt is detected ?

A: Interrupts, when enabled, cause a branch to the address specified by the AF field and interrupt address supplied by the I/O control. Standard I/O interrupts supply an address component of 0111 to the least significant four bits. The most significant five bits are specified by the user (AF field) and may be anywhere in the currently active control store page. At that address, the microprogram should perform the functions of the V73 IWAIT microinstruction (location OD7 on page zero) and then branch to INT1 (OD1 page zero) or perform in the current page the functions of INT1, INT2, INT3 and INT4.

Q: Is data in the memory input register protected against DMA and PMA operations ?

A: Yes, DMA and PMA operations do not alter the memory input register.

Q: When reading data from memory is the data available in the memory input register at a fixed number of microinstructions following memory initiation, or must a wait for memory done be placed before using the data or starting another memory cycle ?

A: Data arrives in the memory input register no sooner than the second microinstruction after its initiation. It may arrive after that. The access time depends upon DMA or PMA or other memory bus cycles, semiconductor memory refresh cycles or core memory rewrite cycles in progress at the time. If a new memory cycle is to be initiated immediately following completion of the current cycle, interlocking is automatic as the execution of microinstructions will cease until the new cycle initiation is accepted by memory control. Otherwise a wait-for-memory-done function must be specified.

SECTION 3

TECHNIQUES

This section describes the use of flow diagrams in writing user microprograms and the interface with the 620 emulation microprogram. Several detailed examples of flow diagrams for sample microprograms are included here. These examples will be continued in later sections, where the flow diagrams will be translated into assembly language.

3.1 INTERFACE WITH 620 EMULATION

3.1.1 Execution of User Microprograms

Branch to Control Store Implementation

The BCS instruction causes a branch to the WCS and always goes to page 1. The control store word in page 1 is specified in bits 0 - 4, allowing a branch to one of the first 32 words, which contain vectors to microprogrammed routines. The BCS instruction is a special coding of an I/O instruction and, as such, is not a generic mnemonic within the DAS assembler language. This instruction for use in symbolic DAS coding must be defined by the user.

The BCS macro is decoded directly on the WCS page during primary decoding time as defined by the processor logic. A BCS is performed only if decoder control store page 0 is currently selected. Any other control store selected causes the macro to be taken as part of a different instruction set. The BCS page branch does not change the decoder control store selection. A local page-branch micro-operation can change the selection of a decoder control store to page 1.

3.1.2 Steps in Instruction Execution

The following are the general stages in the execution of a 16-bit *macro* instruction:

1. A microinstruction initiates an instruction fetch.
2. The instruction is transferred from memory to the instruction buffer.
3. The instruction is copied into the instruction register and a request is made for a decoding of the instruction buffer contents. This decoding simply identifies the instruction to be a member of a certain class of instructions and effectively causes a branch to a microroutine which does any work common to that class; for example, single-word memory-addressing instructions may use the same microroutine for computing the effective memory address.
4. Secondary decoding of the instruction determines its exact identity. This is done by such features as field-selection addressing, which allows using bits from the instruction register to determine a microprogram branch address. Using such methods, the microinstructions which complete the actual execution of the instruction are reached.
5. Microinstructions which form the instruction are executed.

3.1.3 Instruction Pipeline

In our system, the term **instruction pipelining** refers to the technique of fetching the next instruction from memory before the current one has finished executing. This is possible due to the availablility of two 16-bit registers for holding instructions. The first is the instruction buffer (IBR), which receives the instruction being fetched from memory. In IBR the next instruction is held while the current instruction being executed is in the instruction register (I). When ready, the instruction buffer is transferred to the instruction register and the next instruction may be fetched from memory.

The chief advantage of this method lies in the fact that the microinstructions are much faster than the fetches from memory.

Thus, without the pipeline, a one or two microinstruction delay would be added to the execution of each instruction while the processor waited for the instruction from memory.

Interfacing with the Pipeline

The instruction pipeline is crucial to the execution of the standard instruction set. Thus, any new instructions being added through microprogramming must consider and be cautious of the effects and requirements of the pipeline. Because of the pipeline, user's microroutines in WCS can rely on certain things being true when they receive control from page zero. Likewise they must make sure certain techniques are used when they exit to read-only memory.

Upon entry to WCS by a BCS instruction, the following conditions exists:

a. The program counter (P) is pointing to the word following the BCS.

b. The BCS command will be in the instruction register.

c. The word following the BCS will be on its way from memory to the instruction buffer and memory input buffer.

On exit from WCS the microprogram must set conditions for the next command, and maintain the pipeline. In particular the following are required:

a. The next instruction to be executed is in the instruction buffer. This will often be the word after the BCS, which was already on its way there on entry. If the BCS has a parameter, or if the instruction buffer was modified, then the instruction may have to be fetched.

b. The program counter should be incremented to one beyond the location of the next instruction and an instruction fetch initiated. This will not only preserve the pipeline but will also make sure any memory activity necessary to complete setup of condition (a).

c. The instruction buffer should be copied into the instruction register in preparation for its execution.

d. A request for decoding of the instruction buffer contents should be made along with a page branch back to page zero, i.e., ROM. The decoding results in the correct microroutine getting control for execution of the next instruction.

In most cases, the preceding steps can be summarized by the rule:

> The second to last microinstruction should increment P and do an instruction fetch.
>
> The last microinstruction should transfer IBR to I and request decoding addressing.

3.1.4 ROM Standard States

Much of the interfacing with the pipeline can be done by using standard microinstructions (standard states) in page zero. These were developed explicitly for this purpose for use by the 620/f emulation. The most common ones make up the three microword sequence listed below. They may be used simply by doing a page jump directly to whichever microword is appropriate.

Address	Label	Function
13E	SS1M	Restarts the pipeline at P with an instruction fetch by P. It then branches to SS2M.
92	SS2M	Maintains the pipeline by incrementing P and requesting an instruction fetch. It branches to SS3M.
2D	SS3M	This instruction decodes the IBR contents to determine the next microinstruction to execute. It also copies the IBR into I.

3.1.5 Summary of Branches Between WCS and ROM Control Store

From ROM to WCS

BCS Macro (from Decoder Page Zero Only)

This macro ensures the start of a processor fetch during the primary decode of the BCS according to the V73 pipeline rule. The clock change and page selection occur during the primary decoding of the microinstruction.

I/O Branch

Control is transferred to the selected page of central control store during the data phase of the I/O command. I/O branch can go to any central control store page and does not select a decoder.

This mechanism assures that no DMA I/O memory transfers and no processor memory transfers are in process during the clock change.

From WCS to ROM

The I/O branch is not a viable mechanism from WCS to ROM.

A micro level page branch is the standard method for going from WCS to ROM. This operation is the converse of the BCS disscussed above.

Standard state sequences in the ROM provide pipeline start up and various other housekeeping functions for the standard instruction set. These may be of interest for particular microprogramming entrances.

3.1.6 Varian 73 Register Usage

The 620 emulation on Varian 70 series systems uses some general-purpose registers. Using the standard instructions with his own microprograms a user is responsible for preserving the settings and restoring those necessary to their original conditions. The use and requirements for particular registers are described below. All others are only used by user's microprograms.

Registers 0, 1, and 2 are used for the emulation of the A, B, and X registers respectively. These need not be restored by user's microprograms.

Register 3 is forced to all zeros by the halt microprogram and used as a source of zeros by the standard instruction set. Its restoration is required.

Register 4 is also used by the halt program and saves the contents of the instruction register. While the standard microprograms are running it is not used and therefore does not require resetting.

Register 5 is a source of ones for the standard microprograms and must be reestablished as such by a user's microprogram.

Registers E and F (15 and 16) are used as temporary storage for some standard instructions yet their use does not extend beyond the particular single instruction so these two do not need to return to a set value.

Register Usage

Number	Standard Use	Restore
0	A register	no
1	B register	no
2	X register	no
3	All zeros	yes
4	Saves I	no
5	All ones	yes
6-D	None	no
E	Temporary	no
F	Temporary	no

3.2 FLOW DIAGRAM

3.2.1 Rationale

As the reader should now be aware, the 64-bit microword is both extremely powerful and extremely complex. This may result in several problems. A beginning microprogrammer can be completely baffled how to start. Intermediate microprogrammers tend to be confused about how much or how little can be done in single microinstruction.

The microprogram flow diagram is designed to minimize these problems. Making a flow diagram for a microprogram is roughly comparable to the low-level flowcharting of an assembly language program. The flow diagram, however, is designed to provide special assistance to the microprogrammer. It gives the basic capabilities of the standard microword, thus providing reminders of both what can be done and what should be done in each microword.

3.2.2 Format

A sample blank microprogram flow diagram form can be seen in figure 3-1. The vertical columns each represent a single microinstruction.

The horizontal rows are divided into the type of operations that can be performed. A microinstruction is created by going down a column and filling in the appropriate boxes with the specific operations desired in each general category. Many of these operations can be specified using the mnemonics introduced in the previous section. Table 3-1 provides an ordered list of mnemonics.

Specifically, the first row of the flow diagram is used for identifying the particular microword. Labeled **IDENT**, this row is usually left blank unless the microword is referenced elsewhere in the microprogram. Such reference occurs most often when the microword is the target of a jump from another microword. When not empty the box usually contains the label which will be carried through to the actual assembly language version. Depending upon the programmers preference absolute or relative addresses could also be assigned here.

The group of three rows under **MEMORY** specifies both the current state of memory and the requests for memory operations being made in the current microword. The **FUNCTION** row specifies the former. It is useful for charting out memory activity and optimizing the memory usage. In microprograms where memory activity is not critical, this row could be left blank.

The **REQUEST** row indicates the type of memory request being made in the microword. The **ADDRESS** row specifies the source of the memory address for the requested operation. If no request is made, then both these rows can be blank.

The **ALU** section of the flow diagram consists of four rows. These rows specify the two inputs for the ALU, the operation to be performed on them, and the destination of the result.

Two rows are included in the **STATUS** section. The first, **SAMPLE**, specifies which flags and status bits are to be sampled during that microinstruction. Sampling is usually necessary before the flag or status indicators can be tested. The **TEST** row specifies which flag or status bit, if any, is being tested in the current microword. This testing

	IDENT								
MEMORY	FUNCTION								
	REQUEST								
	ADDRESS								
ALU	INPUT A								
	INPUT B								
	OUTPUT								
	DESTINATION								
STATUS	SAMPLE								
	TEST								
ADDRESSING	MODE								
	ADDRESS								
OTHER	SPECIAL ACTIONS								

Figure 3-1. Sample Flow Diagram Form

may be used both for conditional memory requests and conditional addressing.

The two rows of the **ADDRESSING** section specify the addressing method or mode being used and the resulting effective address or addresses. These boxes are often left blank to signify normal addressing with the next column on the right to be executed next. The label contained in the IDENT row can also be used here.

The **SPECIAL ACTIONS** section is provided for the micro-operations which do not fit conveniently into the other sections. Most common among these are the operations on the special registers and counters. These include the operand register, program counter, and shift counter. Such things as register field control or even general comments could also be included here.

3.3 FLOW DIAGRAM MNEMONICS

The following table 3-1 lists the sections of the flow diagram and some applicable mnemonics. These mnemonics represent the most common values and should be sufficient for many microprograms. Other functions without mnemonics can be described in whatever way the user finds clearest. The ways could range from actually writing the field values to putting in verbal commentary.

Table 3-1. Mnemonics for Microprogramming Flow Diagrams

Row	Mnemonic	Comments
IDENT	None	User-supplied labels and addresses
MEMORY FUNCTION	None	User-supplied commentary on memory operations
MEMORY REQUEST	IF	Instruction fetch
	OF	Operand fetch
	OS	Operand store
	BS	Byte store
	TESTF,–	Conditional request (on test condition false)
	TESTT,–	Conditional request (on test condition true)
	WAIT, MEMDN	Wait for memory done (before going to next microword)
MEMORY ADDRESS	ALU	ALU output
	P	Program counter
	MIR	Memory input register
	OVR	Override memory operation of the previous microword using its memory address
ALU INPUT A	Rn (n = 0,1,2,...,F)	General register 'n'
	Rn, SL	General register 'n' shifted left on bit position.
	Rn, SR	General register 'n' shifted right on bit position
	P	Program counter
	ZERO	All zeros (0)
	ONES	All ones (FFFF)
		NOTE: When using a shifted general register, user must specify condition of high and low bits.
ALU INPUT B	Rn (n = 0,1,2,...,F)	General register 'n'
	MIR	Memory input register

(continued)

Table 3-1. Mnemonics for Microprogramming Flow Diagrams *(continued)*

Row	Mnemonic	Comments
	IOR	I/O register
	STAT	Status word
	LIT	The 16-bit value from 0 to FFFF
	MSK	Instruction register masked by 'xxxx
	OPR	Operand register
	ORSE	Operand register right byte, sign extended
	OLSE	Operand register left byte, sign extended
	ORZF	Operand register right byte, zeros in left byte.
	OLZF	Operand register right byte in left byte position, zeros in right byte
		NOTE: When using MSK or LIT, caution should be used to avoid field conflicts with other mnemonics.
ALU OUTPUT	ZERO	All zeros (0)
	ONES	All ones (FFFF)
	TRNA	A (transfer input A)
	TRNB	B (transfer input B)
	INCA	A + 1
	INCB*	$A \vee B + 1$ (B + 1 when A = 0)
	DECA	A − 1
	DECB	A + B (B − 1 when A = FFFF)
	ADD	A + B
	SUB*	A − B
	SHFA	A + A (shift A left one)
	AND	$A \wedge B$
	OR	$A \vee B$
	EOR	$A \veebar B$ (exclusive OR)
	NOTA	$\overline{A}$
	NOTB*	$\overline{B}$
	TCB*	$A \vee \overline{B} + 1$ (two's complement B when A = 0)
	*cannot be used when input B is MSK or LIT.	
ALU DESTINATION	Rn (n = 0,1,2,...,F)	General register 'n'
	Special registers	Refer to special actions row
		NOTES: 1) general register cannot be used here if input B was LIT or MSK. 2) general registers used for both input A and destination must be the same general register.
STATUS, SAMPLE	SHFT	Set shift flag
	OVFL	Set overflow flag
	ALU	Set ALU related flags (i.e., ALUO, ALUS, ALUC, and ALUZ)
STATUS, TEST	OVFL	Overflow flag
	IOSR	I/O sense response

(continued)

Table 3-1. Mnemonics for Microprogramming Flow Diagrams *(continued)*

Row	Mnemonic	Comments
	SSW3	Sense switch three
	SSW2	Sense switch two
	SSW1	Sense switch one
	TFIR	Test from instruction register
	ALUO	ALU ones flag
	ALUS	ALU sign flag
	ALUC	ALU carry flag
	ALUZ	ALU zeros flag
	SHFT	Shift flag
	MIRS	Memory input register sign
	SFTC	Shift counter all ones flag (i.e., overflow)
	GPRS	General register 0 sign
	NORM	Normalize flag
	QUOS	Quotient flag
ADDRESSING, MODE	PJMP to n	Page jump to page 'n'
	FSEL	Field select addressing
	INT	Interrupt addressing
	DECODE	Addressing by decoder control store
	TESTT	test addressing; pass if test condition true
	TESTF	Test addressing: pass if condition false
	POPJMP	Branch/pop to an address specified by stack
		NOTE: these are only a basic set of abbreviations, to be used alone or in combination.
ADDRESSING, ADDRESS	P –	Test pass address
	F –	Test fail address
SPECIAL ACTIONS	POUT	Load program counter with ALU output
	SCOUT	Load shift counter with ALU output
	OPROUT	Load operand register with ALU output
	INCP	Increment the program counter
	INCSC	Increment the shift counter
	INCP, OPROUT	Does both.
	SHFTOP, LFT	Shift operand register left one bit position
	SHFTOP, RGHT	Shift operand register right one bit position
		NOTE: high/low bits must also be specified by user on these two operations
	IBR to I	Transfer instruction buffer to instruction register.
	PUSH,X	Push value x on the stack (requires PJMP addressing mode)
	POPDEL	Delete entry at top of stack (requires PJMP addressing mode)

3.4 FLOW DIAGRAM EXAMPLES

The following examples are included:

1. BCS Entry Point Initialization
2. Memory-to-Memory Block Move
3. Reentrant Subroutine Call
4. Fixed-point ADD to any of 16 general registers with direct addressing to 64K.
5. Cyclic Redundancy Check (CRC) Generation.

Each of the examples includes a description of the problem, a description of how it was handled, and a flow diagram. Later in this manual, the examples will be continued in the form of assembler listings of the code produced from each of the flow diagrams in section 5.

3.4.1 BCS Entry Point Initialization

This is essentially an example of making a micro subroutine which is simply a NOP. From the standpoint of being an example, it details how to reach WCS and then return to the *macro* level. From a functional standard point, it provides meaningful initialization for the 20 (hex) BCS entry points in WCS. By loading this program before all others, any unused BCS entry points will have meaningful contents (as opposed to possibly fatal random contents).

Referring to the flow diagram, (figure 3-2) the thirty-two entry points are all initialized to the same microinstruction. It is simply a page branch to a standard microword, SS2M, on page zero. This will result in a return to the *macro* level by maintaining the pipeline and returning control to the ROM central control store.

3.4.2 Memory-to-Memory Block Move

This microprogram is designed to move a block of **n** words from one area in memory to another.

For purposes of this example, the microprogram is called by executing a BCS to word zero of WCS page one. It takes its arguments in the following format:

A register (R0):	**to** address
B register (R1):	**from** address
X register (R2):	block length

When called, words are sequentially copied from their old location (**from** address) to their new position (**to** address). The number of words moved is equal to the block length.

The following commentary describes how the microprogram operates. Refer to the flow diagram figure 3-3.

Word zero in page one is the entry point for the BCS instruction. It contains a branch to a microword labeled MBM, which may be on any WCS page. This is the actual beginning of block move and no further decoding of the BCS is done.

The microprogram starts by setting up its parameters. The current program counter value is saved in R7. Next, the **from** address minus one is put in its place. Having it in the program counter will allow easier use of it as an address source for memory requests. The **to** address is also decremented. These addresses are decremented because they are incremented in the instructions which request the memory operations.

After this initialization, a three microinstruction loop is entered which does the actual block move. The first microword, (MBMA), increments the **from** address in the program counter. It then requests that the word at that address be fetched from memory. It also puts the memory input register (MIR) onto the ALU output. Once the looping is begun, the MIR will contain the word just fetched from memory. Placing it on the ALU will cause it to be stored at the **to** address, since the previous micro in the loop requested a write of ALU output into memory.

The second mircoword in the loop decrements the block length in R2. The ALU output (i.e., the new value) is sampled for testing in the next microword.

The next microword, the third and last in the loop, increments the **to** address in R0 and tests the ALU sign flag. If it is off, then the block length has not yet become negative and the necessary number of words has not yet been moved. In this case, an operand store is requested using the **to** address as the destination. The next microword will have to specify the the value to be stored, so a loop is made back to MBMA which will do this. This loop also causes the next word to be fetched and the process continues until the block length goes negative. In that case the loop is exited and the extra memory fetch requested is simply forgotten.

Microword MBMB restores the program counter to the address in R7 and starts a memory cycle to restore the pipeline. A branch is executed to standard state SS2M which increments the program counter and starts a second memory fetch to fill the instruction pipeline. Upon entering standard state SS3M, the macroinstruction is decoded and control is returned to the processor's central control store.

3.4.3 Reentrant Subroutine Call and Return

This example provides call and return microprograms for reentrant subroutines. The subroutine call stores its return address in the X register (R2) and saves the original contents of X on a stack pointed to by the B register (R1).

The subroutine return simply pops the stack back into the X register and branches back to the return address.

	IDENT	(0 thru IF)							
MEMORY	FUNCTION								
	REQUEST								
	ADDRESS								
ALU	INPUT A								
	INPUT B								
	OUTPUT								
	DESTINATION								
STATUS	SAMPLE								
	TEST								
ADDRESSING	MODE	PJMP to 0							
	ADDRESS	SS2M (92)							
OTHER	SPECIAL ACTIONS								

VTII-2028

Figure 3-2. Flow Diagram for BCS Entry Point Initialization

	IDENT	word 0 page 1	MBM			MBMA			MBMB
MEMORY	**FUNCTION**					storing data	fetching data	fetching data	
	REQUEST					ØF		TESTF ØS	IF
	ADDRESS					P		ALU	ALU
ALU	**INPUT A**		P	R0	R1	—	R2	R0	R7
	INPUT B					MIR			
	OUTPUT		TRANA	DECA	DECA	TRANB	DECA	INCA	TRANA
	DESTINATION		R7	R0	see below	—	R2	R0	see below
STATUS	**SAMPLE**						ALU		
	TEST							ALUS	
ADDRESSING	**MODE**	PJMP						TESTT	PJMP to 0
	ADDRESS	MBM						P-MBMB F-MBMA	SS2M (092)
OTHER	**SPECIAL ACTIONS**				PØUT	INCP			PØUT

VTI1-2029

Figure 3-3. Flow Diagram for Memory-to-Memory Block Move

For purposes of this example, the subroutine call is executed by doing a BCS to word 1 of WCS page 1. The word following the BCS is taken as the effective address of the subroutine being called. The subroutine return is made by executing a BCS to word 2 of WCS page 1.

The stack operations are performed in the following way. A *push* causes the B register to be decremented and the X register stored at the resulting address. A *pop* causes the X register to be loaded from the memory location pointed to by the B register followed by the B register being incremented.

The following is a detailed description of the **subroutine call.** Refer to the flow diagram in figure 3-4.

The first microinstruction of the routine is at the BCS entry point. On the memory-to-memory block move, this first microword of the program did nothing but branch to the actual microroutine. The only reason for not combining it with the next microinstruction was to clarify the relationship of the entry point and the rest of the program. In an actual application where execution time is critical, the microwords would have been combined. This is done on the subroutine call example. The first microword decrements the stack pointer (R1) and begins saving the contents of R2 at the resulting address. It then does a page branch to the rest of the microroutine which could be on any WCS page.

The second microword places R2 on the ALU so that it will be stored by the memory request in the first microword. R2 must be on the ALU for the entire duration of the write into memory. Since this could take a variable amount of time, (depending on the type of memory in the system), a request is made to wait for the memory-done signal. This means the next microword will not be executed until the write operation is complete and thus, R2 will stay on the ALU for the necessary time.

The third microword saves the return address in R2. The program counter is currently pointing to the word after th BCS instruction. That word contains the effective address of the subroutine to be called. Thus, the return address is obtained simply by incrementing the program counter and then storing it in R2. This microword also begins the actual transfer to the subroutine to be called. This is done by restarting the pipeline at the address of the subroutine. That address is already in the MIR due to the fact it was the word after the BCS.

The fourth microword sets the program counter to the second word in the subroutine call and requests it be fetched. This completes the restarting of the instruction pipeline and a return can be made to ROM control. This is done with a page jump to SS3M on page 0. Note that the fourth microword has performed all the functions of SS2M.

The following is a detailed description of the **subroutine return.** Refer to the flow diagram in figure 3-5.

The first microword begins restarting the instruction pipeline at the return address. Also, the program counter is restored.

The second microinstruction begins the fetch of the original contents of R2 off the stack.

The third microword increments the stack pointer to finish the pop of the stack. It also finishes the restart of the instruction pipeline by requesting another instruction fetch by the incremented program counter.

The last microword restores the old contents of R2, which by now have been transferred from memory to the memory input register (MIR). Since the pipeline has now been restored, the microword can return to ROM using a page jump and with request for decoding addressing. The decode will generate the next address in page zero based on the next 'macro' instruction to be executed.

Note that the second to last microword performs the functions of SS2M and the last microword performs the functions of SS3M.

3.4.4 64K-Memory ADD to any of the General-Purpose Registers

This example adds the contents of any location in 64K words of memory to the contents of any of the 16 general-purpose registers, R0, R1,...,RF. The sum replaces the previous contents of the specified register. If overflow occurs, the overflow status bit will be set. The addressing mode for this example will be indexing by general register R1.

In execution the contents of LOC bit 8 - 15 specify a branch to control store (BCS) instruction. Bits 0 - 3 define the operation to the performed and the addressing mode to be used. Bits 4 - 7 specify the general register affected.

With indexing the contents of all LOC + 1 are added to the contents of the register (R1), and the 16-bit sum is used as the effective address of the operand. The operand is fetched from memory and is added to the contents of the register specified by the LOC 4 - 7.

A flow diagram follows as figure 3-6.

The strategy used for the operation described above has the following steps:

1. (AD1 or AD1A) enter from decoding of BCS in page zero. Address fetch cycle has been initiated. Initiate next instruction fetch and increment P.
2. Transfer contents of MIR (address value) to OPR to preserve against alteration by previously initiated instruction fetch.
3. Perform indexing by adding contents of R1 to contents of OPR. Initiate operand fetch using ALU output as effective address.

(continued)

	IDENT	word 1 page 1	LAB1						
MEMORY	FUNCTION		store of R2 on stock		fetch of first subr. inst.				
	REQUEST	ØS	WAIT MEMDN	IF	IF				
	ADDRESS	ALU		MIR	ALU				
ALU	INPUT A	R1	R2	P	ZERO				
	INPUT B				MIR				
	OUTPUT	DECA	TRNA	INCA	INCB				
	DESTINATION	R1		R2	see below				
STATUS	SAMPLE								
	TEST								
ADDRESSING	MODE	PJMP			PJMP to 0				
	ADDRESS	LAB1			SS3M (02D)				
OTHER	SPECIAL ACTIONS				PØUT				

Figure 3-4. Flow Diagram for Subroutine Call

VTII-2031

	IDENT	word 2 page 1	LAB2						
MEMORY	**FUNCTION**		fetch of nxt. instr.	fetch of orig. R2	fetching second instruction				
	REQUEST	IF	ØF	IF					
	ADDRESS	ALU	ALU	P					
ALU	**INPUT A**	R2	R1	R1					
	INPUT B				MIR				
	OUTPUT	TRNA	TRNA	INCA	TRNB				
	DESTINATION	see below		R1	R2				
STATUS	**SAMPLE**								
	TEST								
ADDRESSING	**MODE**	PJMP			PJMP to 0; DECØDE				
	ADDRESS	LAB2			from IBR by decode				
OTHER	**SPECIAL ACTIONS**	PØUT		INCP	IBR to I				

Figure 3-5. Flow Diagram for Subroutine Return

	IDENT	ADI/ADIA*	AD2	AD3	AD4		AD5		
MEMORY	**FUNCTION**	ADDRESS AF FETCH	IF →		ØF →		IF		
	REQUEST	IF		ØF	IF				
	ADDRESS	P		ALU	P				
ALU	**INPUT A**			R1			Rx*		
	INPUT B		MIR	ØPR	MIR		MIR*		
	OUTPUT		TRNB	ADD			ADD		
	DESTINATION						Rx		
STATUS	**SAMPLE**						ØVFL, ALU		
	TEST								
ADDRESSING	**MODE**						PJMP to 0 DECØDE		
	ADDRESS	AD2	AD3	AD4	AD5		WØRD 0 PAGE 0		
OTHER	**SPECIAL ACTIONS**	INC P *located at page 1 word 00 and 10	ØPRØUT		INCP register Bits I4-7	field select	IBR to I *from previous micro register field select		

Figure 3-6. ADD from 64K-Memory to General-Purpose Register

4. Wait for completion of operand fetch by specifying next instruction fetch with incremented program counter and field select register specifications from instruction bits 4 - 7 into AA field. Set BB field to select MIR.
5. Add contents of MIR to contents of previously selected register and store sum in selected register. Sample overflow condition. Page jump to V73 page zero with decoding of instruction fetched by step 1.

Execution Time Estimate

Execution time depends upon the memory speed involved. With 330 nanoseconds semiconductor memory the pipeline is kept full. The number of microinstruction times from decoding to decoding is six. All of these are from writable control store. The execution time is therefore six times 190 or 1140 nanoseconds. Since three memory cycles are involved, the effective three cycle time is 1140 divided by 3, or 380 nanoseconds.

3.4.5 Cyclic Redundancy Check (CRC) Generation

INSTRUCTION FORMAT

15 … 9	8	7 … 4	3 … 0	
1 0 5		CRC Vector		LOC
Data Array Word Address				LOC + 1
Byte Count				LOC + 2

DATA FORMAT: Packed 2 bytes in each word as follows:

Byte 1	Byte 2
Byte 3	Byte 4
⋮	⋮
Byte N–1 may be last byte	Byte N

The packed byte array at the specified address is scanned and the 16-bit cyclic redundancy check is performed. The 16-bit CRC is left in the accumulator (A register or R0). If the accumulator is not cleared before entry, the accumulator's contents will be included in the CRC.

The CRC polynomial word is $X^{16} + X^{15} + X^{2} + 1$, which is commonly used in binary synchronous communication.

Since array size can be quite large, the instruction can be interrupted after service of every two bytes. When interrupt service is completed, the process of CRC generation is resumed and runs to completion (except as interrupted). The overflow flag is used to signal an interrupted instruction. If it is set, contents of the B and X registers are taken as data address and byte count respectively.

R0, R1 and R2 (A, B and X) registers are used by this instruction. R0 is the current CRC value. R1 is the current data array address. R2 is the current byte count value. RF contains the CRC polynomial (octal 100005). The overflow flag is used to designate an incomplete instruction.

The calling sequence normally used would be:

TZA	(clear accumulator)
ROF	(reset overflow flag)
BCS	CRC
Address	(data array address)
Byte count	(number of bytes in array)

•
•
•

Detailed Description of Procedure

1. Enter from decoding of BCS in page 1. Address fetch cycle has been initiated. The overflow flag is used to designate an incomplete instruction, i.e., one which was interrupted before the entire byte array was scanned for CRC generation. If such an interrupt had occurred the current data array address and byte count in registers R1 and R2 should be used rather than the corresponding values used when the instruction was initiated. A memory cycle to fetch the byte count is initiated conditionally. The overflow flag is tested for an "off" condition. The 16-bit word representing the CRC polynomial is placed in OPR. If the overflow flag is off, the next step is step 2. If it is on, step 1A is executed.
2. The data array address is copied from MIR into R1.
3. The contents of R1 is used as an address (through the ALU) and the first pair of bytes is fetched. The overflow flag is set to indicate that the instruction is incomplete.
4. The byte count is copied from MIR into R2. ALU status is sampled so that the byte count can be tested for zero in step 5.
5. The shift counter is loaded with – 8 (the number of bits per data byte). The ALU zero status flag is tested to see if the byte count was zero. Execution is suspended (by a "wait for memory done") until the two data bytes are fetched. If the ALU zero flag is off, the next step is 5A; otherwise, step 18 is next.

5A. The CRC polynomial placed in OPR in step 1 is now placed in RF.

6. The data bytes in MIR are copied into OPR.

(continued)

7. Steps 7, 8, 9, 10, 10A, and 11 constitute the iterative loop which accumulates the CRC for the left data byte. In step 7, R0 (the CRC) is shifted one bit left and applied to the ALU input A while the shift counter is incremented. Bit 15 of R0 is copied into the shift flag (DSB). Bit 15 of OPR is applied to ALU input A bit 00. OPR is also shifted one bit left. The CRC polynomial in RF is applied to ALU input B. The exclusive OR is performed by the ALU and the result is placed into R0. The shift counter is tested to see if the eighth bit of the left byte has been processed. If it has, step 10 is executed next; if not, step 8 is next.

8. The DSB flag is tested to see if a correction cycle is needed. (If bit 15 of the old CRC was a zero, the exclusive OR operation of step 7 must be cancelled.) If a correction cycle is necessary, step 9 is executed next; otherwise, the next bit of the data byte is processed by returning to step 7.

9. This correction cycle exclusively ORs the CRC in R0 with the polynomial in RF. The result is placed in R0. When this is done the resulting CRC is that which would have been obtained if step 7 had not performed an exclusive OR. The next bit of the data byte is next processed by returning to step 7.

10. This step is entered from step 7 after the last bit of the left data byte is processed. The DSB flag is tested to determine the need for a correction cycle. The byte count in R2 is decremented. The ALU status is sampled so that it can be tested for zero in step 11. If a correction cycle is necessary, step 10A is executed; otherwise, step 11 is next.

10A. This is a correction cycle identical to step 9.

11. The shift counter is reinitialized to −8 for processing the right data byte. The ALU zero status flag is tested to determine if the right byte should be processed. If ALUZ is not equal to one, the next step is 12; if ALUZ equals one, the next step is 18.

12. This step is identical to step 7. The right data byte which has been shifted left in OPR is now processed.

13. This step is identical to step 8.

14. This step is identical to step 9.

15. The operations of step 10 are performed. The DSB flag is tested as in step 10. If it is set, step 15B is next; otherwise, the correction cycle of step 15A is next.

15A. This step is identical to step 10A.

15B. This step tests for interrupts. If an interrupt is present, step 20 is next; otherwise, step 16.

16. The data array address pointer in R1 is incremented and used as an address for a fetch of the next operand byte pair, if the ALU zero flag is off (indicating the decremented byte count at step 25 was not zero). If the byte count was not zero, step 17 is next; otherwise, step 18 is executed.

17. The shift counter is initialized to −8 and execution is suspended until the next pair of data bytes is fetched from memory. Step 6 is next.

1A. If step 1 determines the overflow flag to be set indicating an incomplete instruction, step 1A initiates the fetch of a data word from memory using the contents of R1 as an address. Step 17 is executed next.

18. If step 16, 11, or 5 determines the byte count to be zero, step 18 resets the overflow flag to indicate completion of the instruction. The program counter is incremented and the net instruction fetch is initiated.

19. A page jump to ROM (page zero) V73 standard state /SS2M, is executed. /SS2M will initiate another instruction fetch to fill the pipeline.

20. If an interrupt was detected at step 15B, the interrupt status must again be tested by step 20. This is because interrupts can be overriden by DMA traps and must be checked twice to ensure that a trap has not occurred which would abort the interrupt. The I/O control is requested to perform an I/O interrupt sequence. Decoding is inhibited since only the interrupt status is to be tested. If an interrupt is found, step 21 is next; otherwise, step 16 is next.

20B. The cycle is performed as in step 10A.

21. If an interrupt was found at step 20, the data array address in R1 is incremented and the ALU zero flag is tested to determine if the byte count at step 15 was zero. If it was not zero, step 22 is next; otherwise, step 24 is executed.

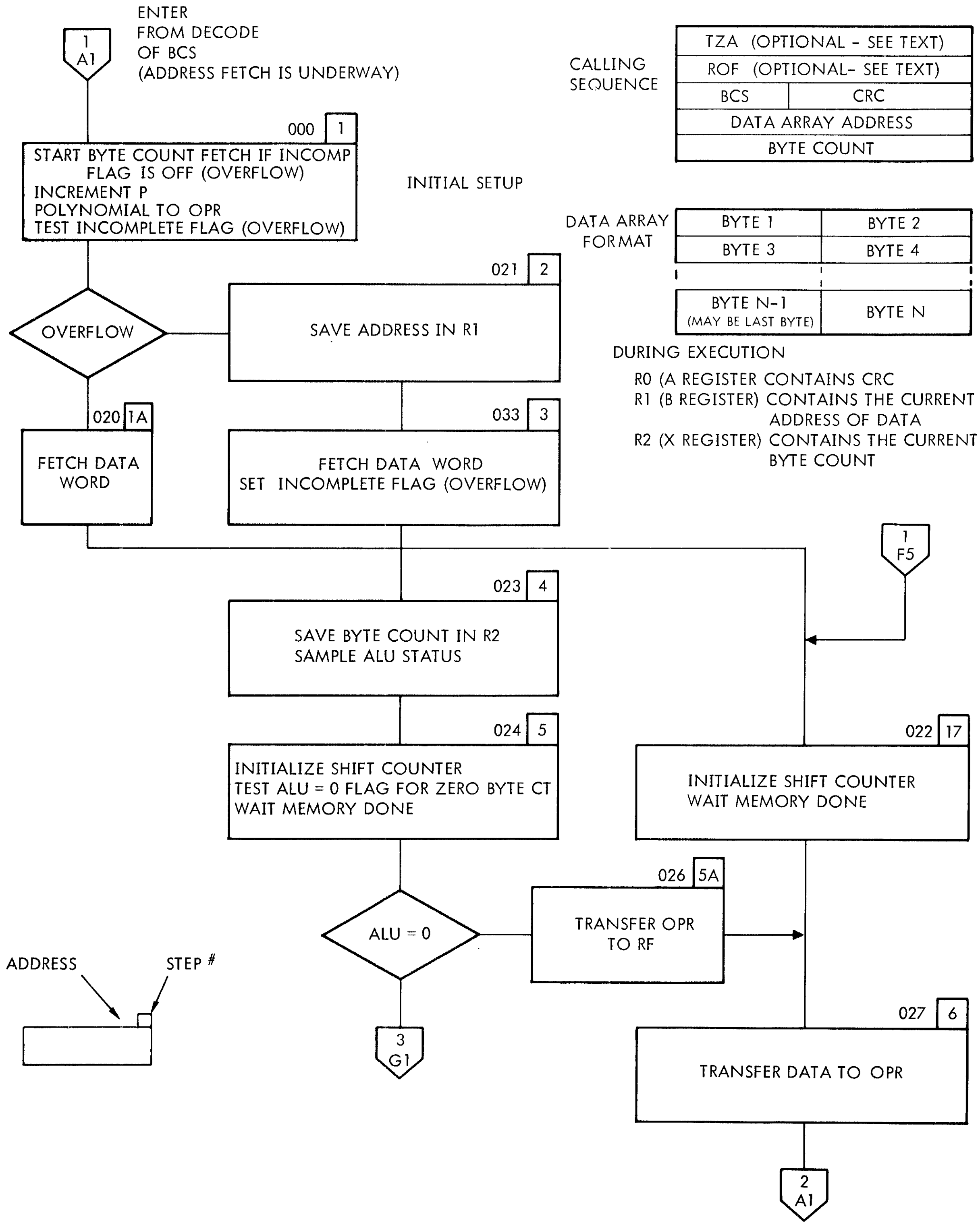
22. The program counter is reduced by 3 to point to the BCS instruction. After completion of the interrupt routine this instruction will be refetched and the overflow flag will be tested in step 1 to determine the need to initialize R1 and R2 from the instruction second and third words.

23. Execution is suspended until the I/O control signals completion of the interrupt sequence, then a page jump to ROM standard interrupt state/INT2 is performed.

24. If the byte count was zero, the overflow flag is reset and an instruction fetch is initiated with the incremented program counter value.

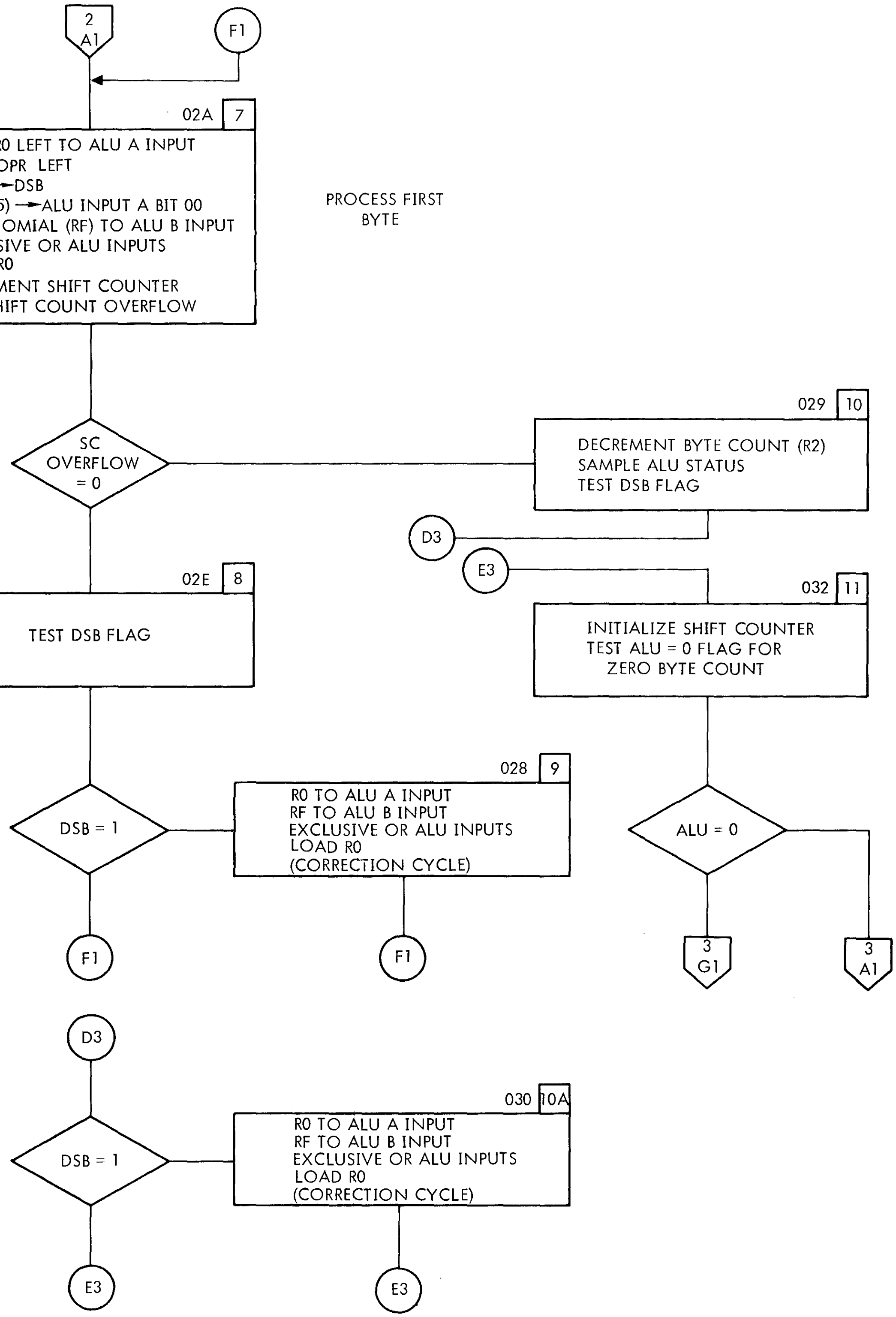
CRC Generation Timing

Execution time depends on memory speed and data array size. If no interrupts occur the timing consists of (a) initialization -- fetch of BCS, address and byte count and first byte pair. This involves one ROM decode cycle and WCS microinstructions 1, 2, 3, 4, 5, 5A, 11, and 6 all at 190 nanoseconds (assuming a 330 nanoseconds main memory cycle). Initialization thus amounts to 1520 nanoseconds. (b) CRC processing -- each byte takes 16 to 24 steps with the average 20 plus steps 10, 11, 15, 15B and 16 all at 190 nanoseconds. Processing takes an average of 8550 nanoseconds for each byte pair. (c) cleanup involves steps 18 and 19 from WCS at 190 nanoseconds, and the memory cycle of SS2M at 330 nanoseconds. Clean up takes a maximum of 710 nanoneconds. Altogether the timing for an array of N bytes averages (2230 + 1/2(N − 2)) times 8550 nanoseconds.



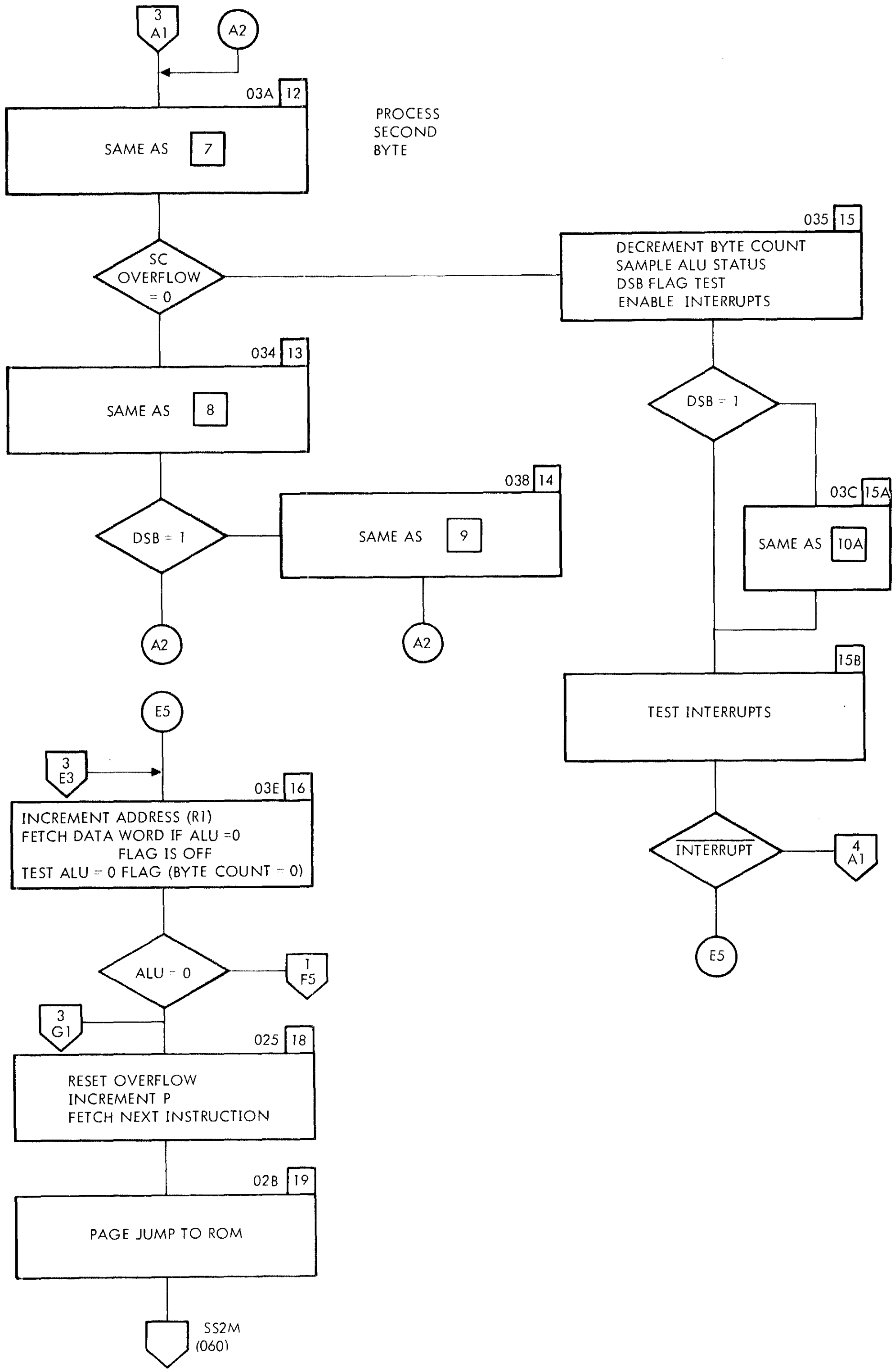
VT12-402

Figure 3-7. Flowchart for Cyclic Redundancy Check Generation Microprogram *(1 of 4)*



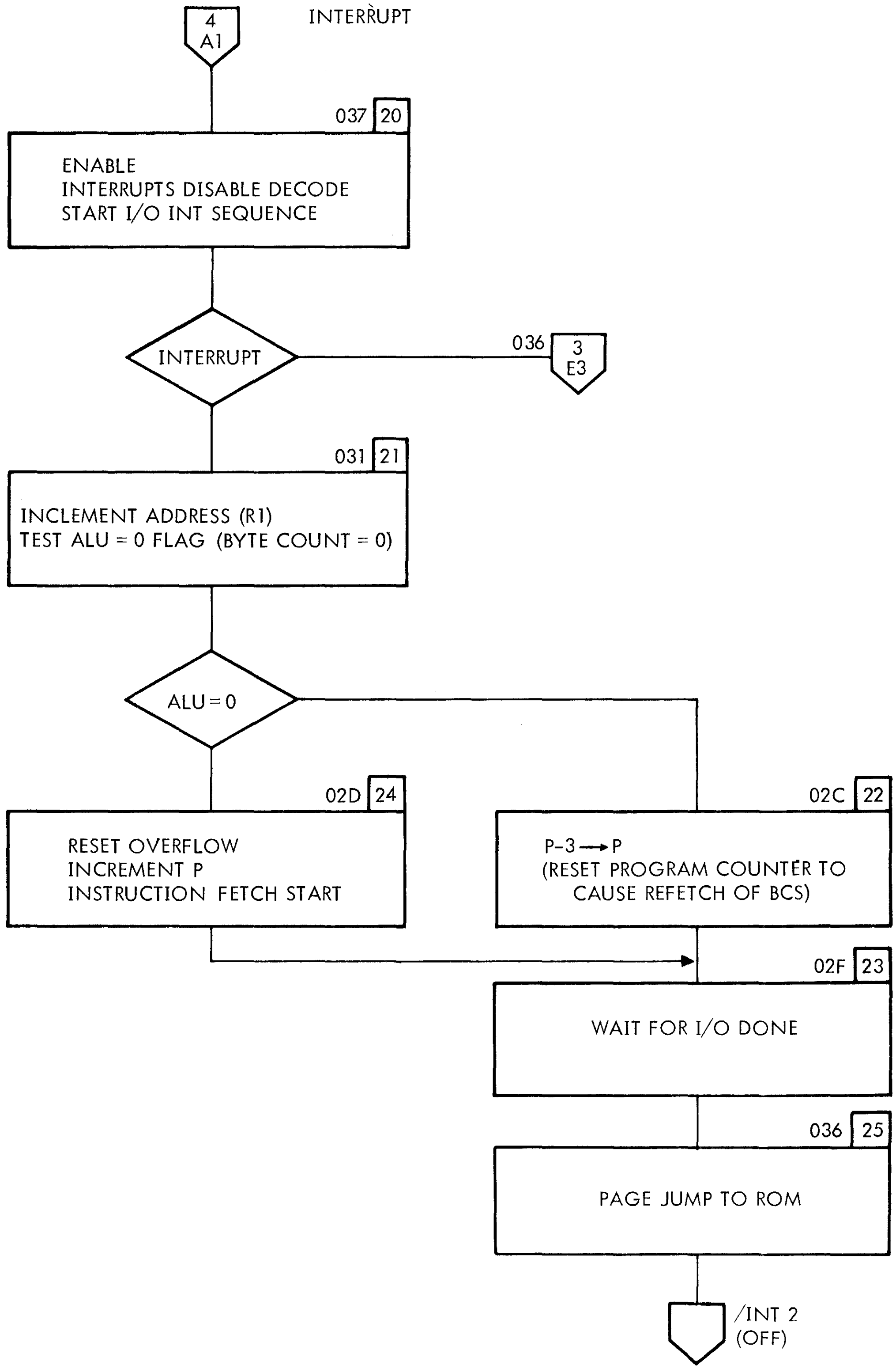
VT12-400

Figure 3-7. Flowchart for Cyclic Redundancy Check Generation Microprogram *(2 of 4)*



VT12-401

Figure 3-7. Flowchart for Cyclic Redundancy Check Generation Microprogram *(3 of 4)*



VTI1-1803

Figure 3-7. Flowchart for Cyclic Redundancy Check Generation Microprogram *(4 of 4)*

		CRC1	CRC2	CRC3	CRC4	CRC5	CRC5A	CRC6	CRC1A
	IDENT	CRC1	CRC2	CRC3	CRC4	CRC5	CRC5A	CRC6	CRC1A
MEMORY	**FUNCTION**	ARRAY ADDRESS FETCH	BYTE COUNT FETCH →	→	DATA FETCH →	→			
MEMORY	**REQUEST**	ØF TESTF		ØF		WAIT			ØF
MEMORY	**ADDRESS**	P		ALU		MEMDN			ALU
ALU	**INPUT A**								
ALU	**INPUT B**		MIR	R1	MIR	MSK, FFF8	ØPR	MIR	R1
ALU	**OUTPUT**	MSK, 8005	TRNB	TRNB	TRNB	TRNB	TRNB	TRNB	TRNB
ALU	**DESTINATION**	TRNB	R1		R2		RF		
STATUS	**SAMPLE**				ALU				
STATUS	**TEST**	ØVFL				ALUZ			
ADDRESSING	**MODE**	TESTF	NØRM	NØRM		TESTT		NØRM	NØRM
ADDRESSING	**ADDRESS**	T-CRCIA F-CRC2	CRC3	CRC4		T-CRCI8 F-CRC5A		CRC7	CRC17
OTHER	**SPECIAL ACTIONS**	INCP, ØPRØUT		SET ØVFL		SCØUT		ØPRØUT	

Figure 3-8. Flow Diagram of CRC Generation (1 of 4)

VT11-2033

VT11-2034

	IDENT	CRC7	CRC8	CRC9	CRC10	CRC10A	CRC11		CRC25
MEMORY	FUNCTION								
	REQUEST								
	ADDRESS								
ALU	INPUT A	R0,SL		R0	R2	R0			
	INPUT B	RF		RF	R3	RF	MSK, FFF8		
	OUTPUT	EØR		EØR	FF6	EØR	TRNB		
	DESTINATION	R0		R0	R2	R0			
STATUS	SAMPLE	SHFT			ALU				
	TEST	SFTC					ALUZ		
ADDRESSING	MODE	TESTT	FSEL MS = 2	NØRM	FSEL MS = 2	NØRM	TESTT		PJMP
	ADDRESS	T-CRC10 F-CRC8	CRC9	CRC7	X'032	CRC11	T-CRC18 F-CRC12		INT2 (page 0, X'FF)
OTHER	SPECIAL ACTIONS	SHFTOP,LFT 0 →O PR00 ØPR15→ALUA 00 INCSC					SCØUT		

Figure 3-8. Flow Diagram of CRC Generation (2 of 4)

	IDENT	CRC12	CRC13	CRC14	CRC15	CRC15A	CRC15B	CRC16	CRC17
MEMORY	FUNCTION								
	REQUEST							ØF TESTF	WAIT
	ADDRESS							ALU	MEMDN
ALU	INPUT A	R0,SL		R0	R2	R0		R1	
	INPUT B	RF		RF	R3	RF			MSK. FFF8
	OUTPUT	EØR		EØR	FF6	EØR		FFO,CF3	
	DESTINATION	R0		R0	R2	R0		R1	
STATUS	SAMPLE	SHFT			ALU				
	TEST	SFTC						ALUZ	
ADDRESSING	MODE	TESTT	FSEL MS = 2	NØRM	FSEL MS = 2	NØRM	NØRM	TESTT	NØRM
	ADDRESS	T-CRC15 F-CRC13	CRC14	CRC12	CRC15B	CRC15B	CRC16	T-CRC18 F-CRC17	CRC6
OTHER	SPECIAL ACTIONS	SHFTØP,LFT 0→ØPR00 ØPR15+ALUA00 INCSC					ENABLE INTERRUPTS SUPPRESS DECØDE		SCØUT

Figure 3-8. Flow Diagram of CRC Generation (3 of 4)

VT11-2035

VTI1-2036

	IDENT	CRC18	CRC19		CRC20	CRC21	CRC24	CRC23	CRC22
MEMORY	FUNCTION		NEXT INSTR. FETCH →					NEXT INSTR. FETCH	
	REQUEST	IF					IF		
	ADDRESS	P					P		
ALU	INPUT A					R1			P
	INPUT B								MSK,FFFC
	OUTPUT					FF0,MF0, CF0			ADD
	DESTINATION					R1			
STATUS	SAMPLE								
	TEST					ALUZ			
ADDRESSING	MODE	NØRM	PJMP		NØRM	TESTT		NØRM	
	ADDRESS	CRC19	SS2M (PAGE 0, X'92)		CRC16	T-CRC24 F-CRC22		CRC25	
OTHER	SPECIAL ACTIONS	RESET ØVFL INCP			ENABLE INTERRUPTS SUPPRESS DECODE START IØ INT. CYCLE		RESET OVFL INCP	WAIT IØ DØNE	PØUT

Figure 3-8. Flow Diagram of CRC Generation (4 of 4)

SECTION 4

MICROPROGRAM DATA ASSEMBLER, MIDAS

For execution the microprograms must be expressed in the internal machine language, yet during their development it is advantageous to express the program in a symbolic language which has more meaning to the person writing the program. This symbolic language is then translated into the executable machine language by the assembler.

In addition MIDAS assembler provides

- symbolic addressing
- macro-definition capability
- user-defined microword formats
- user-defined opcodes
- address field calculations
- error detection
- concordance listing with MOS or VORTEX using the concordance program CONC

4.1 BASIC ELEMENTS

The source language input to the assembler consists of a sequence of records. Each record contains 80 character positions. These characters are represented internally in standard 8-bit ASCII codes. The following paragraphs describe the content and format of the input to MIDAS.

Characters

The characters forming the symbolic source statements are described below. Characters not in this set can appear only in the comment field.

Alphabetic:	A through Z
Numeric:	0 through 9
Special Characters:	/ slash
	* asterisk
	+ plus sign
	− minus sign
	space (blank)
	' apostrophe
	(left parenthesis
	) right parenthesis

MIDAS statements are formed from the character set above. The comment field can contain valid 70/620 ASCII characters in addition to any from the MIDAS character set. Literals may be formed from any ASCII characters.

Symbols

The programmer may create symbols to be used for statement labels or to define numeric values. A symbol may contain one to six characters from the alphabetic or numeric subset. The first character of a symbol must be alphabetic.

Examples of correctly formed symbols

```
ABC4    INPUT1    SAVE4X    P23456
```

Symbols may also use the pound sign (#) or dollar sign ($) character in any character position.

Example

```
A$B#C1   $RUN   A$TOP   #FIVE
```

Constants

A constant is a self-defining term. Four types of constants are available: decimal integer, hexadecimal, octal and binary.

A decimal constant is an unsigned sequence of decimal digits. The value of a decimal constant may not exceed 32767.

A hexadecimal constant is an unsigned sequence of hexadecimal digits, base 16, preceded by the letter X and an apostrophe. The maximum hexadecimal number processed by the assembler is X'7FFF.

An octal constant is an unsigned sequence of octal digits, 0 through 7, preceded by the letter O and an apostrophe. An octal constant can not exceed O'77777.

A binary constant is an unsigned sequence of ones and zeros preceded by the letter B and an apostrophe. Binary constants may be as large as 16 bits.

Expressions

An expression is a single term or a series of terms connected by the following operators. All are integer operators.

+ Addition
− Subtraction
* Multiplication
/ Division

A term is a symbol, constant, or a special symbol, *, which denotes the program location counter. A term is associated with a value inherent to the term in the case of a constant, or assigned by the assembler.

Multi-term expressions are evaluated from left to right. No parentheses are allowed. Expressions are reduced to a single value by the procedure below.

1. Each term is given a value
2. Multiplication and division are performed from left to right
3. Addition and subtraction are performed left to right
4. If an expression has a leading minus sign, the value is computed as though a zero term preceded the minus sign. A leading plus sign is ignored.
5. The value resulting is right-justified in its generated resultant field. Unspecified leading bit positions contain zeros.

Program Location Counter

The assembler maintains a program location counter which is automatically initialized to zero at the start of each assembly. As program statements are processed the assembler assigns consecutive memory (WCS) addresses to the microinstructions generated, unless the program location counter is explicitly modified. The counter may be modified by the ORG and ALOC directives. The asterisk (*) character as a label denotes the current value of the program location counter.

4.2 GENERAL FORM OF STATEMENTS

Input to the assembler is in the form of statements in punched-card images. The statement is contained in a fixed format in character positions 1 through 72. 73 through 80 are reserved for sequencing information and have no effect on the generated microprogram.

A statement is divided into a label, operation, continuation, operand, and comment field. These are discussed in order below.

Label

A source statement can be associated with a symbolic label, which allows the statement to be referenced from other statements in the program. The label, if present, must begin in character position 1 and is terminated by a space. A label may be a one to six character symbol.

Operation

The operation field may consist of the format-defining operator FORM, the label of a predefined or user-defined format statement, a macro name or an assembler directive. The operation field begins in position 8 and is terminated by a space.

Continuation

Continuation lines may be used when additional lines of coding are required to complete a statement originating on one line. There can be up to three continuations per statement. This is designated by the character C in position 15. The actual statement continues in positions 16 through 72. Continuation lines are only valid for the format and program statements.

Operand

The operand field begins in position 16 and is terminated by a space. The operand field may contain subfields separated by commas.

Comment

The comment field is optional for documenting programs. The content of this field is output on the assembly listings but in no way has an effect upon the assembly process. The comment field begins with the first non-blank character following the operand field.

4.3 STATEMENT DEFINITIONS

MIDAS processes four types of statements: format, program, assembler-directive and comment.

4.3.1 Format Statement

The format statement labels and describes a structure for the microinstruction generated by the program statement. Each program statement specifies a format in which the user has grouped and broken up fields within the microword to set values. Two predefined formats are GEN and GMSK, "standard" formats shown in figure 4-1. The user may define additional formats through the use of the format statement.

The general form of the format statement begins with a required label followed by the word **FORM** followed by the field identifiers separated by commas. A field identifier consists of a field length in decimal, which may be followed by a hexadecimal constant enclosed in parentheses.

label FORM **field**(1) , *field(2), . . ., field(n)*

Where:

label	is a symbol formed according to the basic elements
each **field**	is a field identifier which is the field length in decimal, followed by an optional hexadecimal constant enclosed in parentheses **length***(constant)*

ordinal field number	name	field size in bits	
1	TS	4	GEN
2	AF/MS	9	
3	MT	1	
4	FS	4	
5	TF	2	
6	SF	2	
7	GF	4	
8	MR	1	
9	AB	2	
10	IM	4	
11	LB	2	
12	LA	2	
13	RF	3	
14	FF	4	
15	MF	1	
16	CF	2	
17	WR	1	
18	SC	1	
19	VF	1	
20	WF	1	
21	XF	2	
22	SH	3	
23	BB	4	
24	AA	4	

ordinal field number	name	field size in bits	
1	TS	4	GMSK
2	AF/MS	9	
3	MT	1	
4	FS	4	
5	TF	2	
6	SF	2	
7	GF	4	
8	MR	1	
9	AB	2	
10	IM	4	
11	LB	2	
12	LA	2	
13	RF	3	
14	FF	4	
15	MK	16	
16	AK	4	

Figure 4-1. Predefined Formats Recognized by MIDAS

Field length can not exceed 16 bits. Fields are specified from left to right. Each field identifier has an implicit ordinal field number associated with it for reference. All 64 bits of the microinstruction word must be allocated. Fields to which constant values have not been assigned are initialized to zero.

Possible errors in the format statement include allocating more than or less than 64 bits and using a constant value exceeding the size of the field. If an attempt is made to redefine a format, an error is indicated and the format is ignored.

Continuation lines can be used on the format statement but a field identifier may not be carried across lines. A comma must complete the field identifier before continuing the statement on the next line. If the last non-blank character of the operation field is a comma, it implies the next record will be a continuation.

Example:

```
LIST    FORM      14,4,2(X'3),2,4,1,2,
                 C4,2,2,7,16(X'1FFF),4
```

4.3.2 Program Statement

The program statement represents the encoding of the microinstructions in symbolic notation. Each program statement references a format statement to be used in building the microinstruction. The format of the program statement is an optional label followed by a format label followed by a program field.

label **format program-field**

Where:

the **program-field** consists of one or more of the following separated by commas.

One address expression
Predefined opcode
User-defined opcode
Field constant

The single address expression specifies the mode of addressing to be used in fetching the next microsinstruction. The address expression, if present, must be the first item in the program field. The format of an address expression is:

/**mode** (*expression*, *fail address*)

Where **mode** is a key denoting the following possible address modes:

N Normal addressing
T Test
F Field Select
S Test and field select
P Page jump
* Implicit

The value of the first expression in parentheses is the an address of the next instruction under non-test conditions, or if the test passes. The value of the second expression is the address of the next instruction if the test fails.

Modes N, F and P require only the first expression. T and S must use both expressions. None is given for the implicit mode.

Address evaluation is performed with the following considerations:

When the address mode uses field selection (modes F and S), the value of the expression must refer to the lower address selected in that field. This address must be an even numbered address.

The contents of the mask field (MS) and the mask extension field (MT), which provide the mask for the field address, must be defined by the user.

In the test or the test-and-field-select modes of addressing, the fail address must be an even numbered word and must be greater than pass address taken modulo 16.

For **example**, if the pass address is X'16, the range of the fail address must be from X'10 to X'1E and an even word. If the pass address is X'26, the fail address may range (on even words only) from X'20 to X'3E.

The value is 13 bits with the high-order four bits specifing a page number and the low-order 9 a word within the page.
The implicit mode generates normal addressing to the program location counter plus one.
In a page jump the expression specified must produce a value which contains both the page and word addressing information. This destination can be defined through use of the EQU directive.
If the test field (TS) is being used to select interrupts or to specify AA or BB field definition, its value must be defined by the user.

Predefined Opcodes

When a predefined opcode is used in the program field, it specifies that a particular value be placed in a field of the microinstruction as defined by the format statement.

Predefined opcodes are symbols consisting of three to six characters. The first two characters identify a field of the defined formats and the following characters specify the value in hexadecimal notation to be placed in the field. These field names must not be used as labels in user-defined opcodes. The two-character names for fields and the permissible range for each is given below.

Predefined opcodes may be used with user-defined formats since each of these opcodes has an ordinal field number associated with it. There is no predefined opcode for the combined address field AF/MS.

Fields of the Microinstruction

Name	Ordinal Number	Range
TS	1	0 - F
MT	3	0 - 1
FS	4	0 - F
TF	5	0 - 3
SF	6	0 - 3
GF	7	0 - F
MR	8	0 - 1
AB	9	0 - 3
IM	10	0 - F
LB	11	0 - 3
LA	12	0 - 3
RF	13	0 - 7
FF	14	0 - F
MF	15	0 - 1
MK	15	0 - FFFF
CF	16	0 - 3
AK	16	0 - F
WR	17	0 - 1
SC	18	0 - 1
VF	19	0 - 1
WF	20	0 - 1
XF	21	0 - 3
SH	22	0 - 7
BB	23	0 - F
AA	24	0 - F

User-Defined Opcodes

Users can assign values to symbols through the EQU directive. The opcode is placed in parentheses and preceded by the decimal ordinal field number designating the format field for the value.

Statement labels and user-defined opcodes must avoid naming conflicts.

Field Constant

A field constant denotes a value to be placed in a microinstruction field. Either decimal, hexadecimal, octal or binary constant is placed in parentheses and preceded by a decimal ordinal field number.

Error Conditions

The effect of error conditions upon the continuing assembly depends upon the type of error. The errors listed below are indicated on the listing. The action shown in parentheses is taken in the program statement.

a. Reference to a non-existent format (program statement is ignored)

b. Value exceeds the size of field (value truncated)

(continued)

c. Both operand in the program field and a format constant are specified for the same field (inclusive OR of the values inserted)

d. Multiple values generated for a field (first used)

e. Inconsistency between the address mode specified and the values of the address control fields e.g., normal addressing and test field (TF) non-zero. (Mode is used to generate address)

Additional Considerations

The assembler evaluates each operand in the program field, and then uses the format indicated to form a microinstruction. Operand values and format field constants are placed in the appropriate fields.

Values computed for a field are inserted in the field right-justified. Fields whose values are not explicitly defined in either the format or program statement are set to zero.

A program statement may have continuation lines, but an operand may not be carried across lines. A comma must complete the operand before continuing the statement on the next line. If the last non-blank character of the operation field is a comma, it implies the next record will be a continuation line.

Example:

```
EXC1    GMSK      /N(EXC2),LB3,RF3,FFA,
                  CMKF7FF
```

4.3.3 Assembler Directives

Instructions to the assembler are known as directives. These statements have label, operation, operand and comment fields. The operation field contains the name of the directive, such as EQU, ORG, ALOC, SPAC, EJEC, MAC and EMAC.

The directives for macro definition MAC and EMAC are described in a later section. Other assembler directives are discussed in order below.

EQU -- Equate

The EQU directive is used to assign symbols to a given value or the value of another symbol. The symbol in the label field is required in this directive. It is defined to have the value of the expression in the operand field.

The format of the EQU directive requires both a symbol in the label field and expression in the operand field. If the expression in the operand field contains a symbol, it must have been previously defined.

If the symbol in the label field has been previously defined or if there is no label, an error is indicated and the statement is ignored.

Examples:

```
THREE   EQU     3
SCZ     EQU     X'FE
V       EQU     THREE+2
```

ORG -- Origin

The ORG directive sets the program location counter to the value of the expression in the operand field.

A symbol in the label field is optional in the ORG directive. The expression to which the program location counter is set must be in the operand field.

If an expression in the operand field contains a symbol, it must have been previously defined. A value of zero or a negative value in the operand field causes an error to be indicated and the statement is ignored. If the expression exceeds the page size, it is an error and causes the assembly to be terminated.

At the beginning of each assembly pass the assembler initializes the program location counter to zero.

Examples:

```
        ORG     X'1E0
        ORG     BEGIN
```

ALOC -- Allocate

The ALOC directive informs the assembler that it is to skip over previously allocated locations as it is assigning sequential addresses to the generated microinstructions.

From the beginning of an assembly pass until the occurrence of the ALOC directive the assembler will keep a list of all assigned locations. After the ALOC directive is processed the assembler will test each new program location counter setting against the list of allocated locations. If a new value is in allocated space, the assembler will increment the counter and again make the test. The sequence will continue until unallocated space is found.

The format of the ALOC directive requires an expression in the operand field, but the symbol in the label field is optional.

An error is indicated and the statement ignored, if the operand field contains a negative value, zero or exceeds the page size.

In the implicit addressing mode the address of the next instruction is the next allocatable location.

Examples:

```
ALOC    FIELD*4
ALOC    ZERO'20
```

SPAC -- Space

The SPAC directive provides a blank line on an assembly listing to improve readability.

Both the label and operand fields of the SPAC directive are ignored. A symbolic source listing shows a blank line in place of SPAC directives.

Examples:

```
SPAC
SPAC    ADD HERE LATER
```

EJEC -- Eject

The EJEC directive causes the assembly listing device to advance to the first print location of the next output page.

Both the label and operand fields are ignored. EJEC is listed.

END -- End

The END directive causes an assembly to be terminated. An END directive is required as the terminal source statement for each assembly.

A symbol in the label field is optional and assigned the value of the program location counter. The operand field is ignored.

4.3.4 Comment

A statement with an asterisk in the first character position is entirely commentary. Its contents have no effect upon the assembly process, however the statement is output to the listing.

4.4 ASSEMBLY-LANGUAGE EXAMPLES

These examples of microinstruction implementation use MIDAS. The following examples show how representative microinstructions in the WCS could be coded as source statements for MIDAS.

Example 1:

```
EXC1    GMSK    /N(EXC2),LB3,RF3,FFA,MKF7FF
```

This example uses the normal mode of addressing.

Example 2:

```
LASL1   GEN     /T(LASL2,LASL1),TF2,GFC,LA2,
               CRF5,WR1,SC1,XF3,SH6
```

This example shows the use of the test mode of addressing, and the use of a cntinuation record.

Example 3:

```
BT10    GEN     /F(BT20),2(X'F),FS4,RF4,XF1
```

This example shows the use of the field select mode of addressing. The field address mask is provided by the hexadecimal field constant.

Example 4:

```
SWA22   GEN     /S(LDA2,SWA26),2(X'C),MT1,FSF,
               CTF3,GFB,LB1,RF3,FFA,MF1,BB1
```

This example shows the use of the test and field select mode of addressing. The field address mask is provided by the hexadecimal field constant and the predefined opcode MT.

Example 5:

```
SEN2    GEN     /*,1(B'1),IMF,LB1,FFA,MF1,WR1,
               CXF1,AAE
```

This example shows the use of the implicit mode of addressing. The instruction initiates I/O activity and the binary field constant provides part of the I/O control store starting address.

Example 6:

```
P       EQU     X'200     PAGE ADDRESS (PAGE 1)
         .
         .
         .
        GMSK    /P(DIV+P),IMD,LB3,
               C15(*+1+P),AK2
```

This example shows the use of the branch/push operation. The operation effects a page selection and the destination and return addresses are global. The destination address is generated by the address expression. Note the page address contribution of P. The expression for field 15 generates the global address which is pushed on the microprogram return stack. P contributes to this again.

Control returns to the instruction immediately following the branch/push instruction in this example.

Example 7:

```
          GEN      IMD,LB3,AA4
```

This example shows the use of the branch/pop operation. The global return address used is the last item pushed on the stack.

Example 8:

```
SS1M      EQU      X'13E
           .
           .
          GEN      P(SS1M),SF2,GF
```

This example shows the use of the page jump mode of addressing. In page selection the value in the address expression must contain both the page and word contribution to the global address. In this example the page jump is to a standard state in the central control store (page 0) from some other page.

Example 9:

```
SS3M      GMSK     /N(SS2MI),1(X'E),GF5,IM6
```

This example uses the normal mode of addressing but selects the decode-ROM and samples interrupts (GF field bit 2 is true). The hexadecimal constant defines the interrupts which are enabled.

The following examples show the use of page branch, branch/push, and branch/pop operations.

Example 10:

```
SS2M      EQU      X'092
           .
           .
           .
MW1       GEN      /P(SS2M),IM3,SF0,TF0
```

This example of a microword, labeled MW1, does a page jump to one of the standard states in read-only memory.

Example 11:

```
PAGE      EQU      X'200       PAGE ONE SPECIFICATION
           .
           .
           .
MW2       GMSK     /P(SUBR+PAGE),TF0,SF0,
                  CIMD,LB3,AK2,15(MW2+1+PAGE)
           .
           .
           .
SUBR      GEN      ....
           .
           .
           .
EXIT      GEN      TF0,SF0,IMD,LB3,AA4,BB0
```

This example calls a micro subroutine and uses the stack to save the return address. The subroutine call is labeled MW2. It forms the return address by adding the word and page numbers, and then pushes the address on the stack. Likewise, the address of the subroutine is formed by adding page and word numbers. The subroutine returns by a microinstruction labeled EXIT which does a pop jump.

4.5 MACRO CAPABILITY

A macro provides a convenient way to generate a sequence of assembler source statements many times in one or more programs. The macro definition is written only once, and a single statement, the macro reference, is written each time the user wishes to generate the desired sequence of statements. These statements are then processed like any other assembler statements. Macro definition uses the MAC and EMAC directives.

MAC -- Macro

The MACRO directive introduces a macro definition. This definition is terminated by the EMAC DIRECTIVE. The name of the macro is the symbol which appears in the label field of the MAC directive. Operand field parameters may be passed from the macro-reference source statement to the macro through use of the special parameter symbols P(1) through P(n).

A macro is invoked by the appearance of the macro name in the operation field of a statement.

The label field must contain a symbol.

In the macro-reference statement the operand field may contain a list of parameters. At the time the macro-reference is encountered, each parameter is evaluated and stored into a table within the assembler. The parameters may be labels, constants, or user-defined opcodes. Predefined opcodes are not permitted. The macro definition is then processed with the values in the table being substituted for the special symbols P(1) through P(n). For example, if the operand field of a macro-reference statement appears as:

```
     2,ABC,X'E0
```

then within the generated macro the value of P(1) is 2, P(2) is the value of the symbol ABC, and the value P(3) is 224.

All arguments in the macro-reference parameter list are evaluated prior to invoking the macro.

An error is indicated and the MAC direction ignored, if the label field does not contain a symbol. Also an error is indicated and the reference is ignored if the macro has not been defined prior to its being referenced.

If a symbol is present in the label field of a macro-reference statement, it is assigned the value of the program location counter at the time the macro is invoked.

A macro definition may contain a reference to another macro definition, nesting definitions. However, a macro may not be called recursively.

EMAC -- End Macro

The EMAC directive terminates a macro definition. The contents of both the label and operand fields are ignored.

Example:

The following example shows the use of macro definition and reference.

```
ONE      EQU     1
TWO      EQU     2
THREE    EQU     3
FOUR     EQU     4
          .
          .
          .
SHFT   MAC
       GEN       /T(*,SS3M1),TF3,SF3,
                 CGFC,IM8,12(P(1)),RF5,
                 CWR1,22(P(2)),AA1
       EMAC
        .
        .
        .
ASLB   SHFT      TWO,FOUR
        .
        .
        .
LRLB   SHFT      TWO,ONE
        .
        .
        .
ASRB   SHFT      THREE,TWO
```

4.6 OPERATING INSTRUCTIONS

This section describes the operating procedure for MIDAS in each of its three environments: VORTEX, MOS and as a standalone program.

MIDAS runs under VORTEX as a level 0 background task and may be cataloged into the background library using the procedures described in the VORTEX Reference Manual (Varian document 98 A 9952 10x).

MIDAS under MOS must be added to the system file using the system preparation Program as described in the Varian Master Operating System Reference Manual (Varian document 98 A 9952 09x).

MIDAS in the standalone environment makes use of the Standalone FORTRAN IV loader, runtime I/O and runtime utility. Use of the components are describe in the Varian 620 FORTRAN IV Reference Manual (Varian document 98 A 9902 03x).

4.6.1 VORTEX Environment

MIDAS is scheduled from the background library at level 0 by the /LOAD,MIDAS directive. MIDAS terminates when the END statement is encountered, and exits to the executive. Only one source program can be assembled for each load of MIDAS.

MIDAS inputs symbolic source statements from the processor Input device (PI) and outputs these statements on the processor output device (PO). When the END statement is encountered, MIDAS rewinds the PO file and commences pass two. During pass two, it inputs source statements from the system scratch device (SS) and produces an assembly listing on the list output device (LO), and object records on the Binary Output device (BO).

PO and SS devices not only must be the same physical device, but the same magnetic tape, drum or disc unit. If PI is assigned to a Rotating Memory Device (RMD) partition, MIDAS assumes the source records are blocked three 40-word records per RMD 120-word physical record. However, if PI is the same logical unit as the System Input Device (SI), and it is assigned to a RMD partition, MIDAS assumes the source records are not blocked but consist of one source record per RMD 120-word physical record. If BO is assigned to a RMD partition, the output is blocked two 60-word object records per RMD 120-word physical reocrd. The assembler's table space may be expanded and consequently larger source programs assembled by use of the VORTEX /MEM directive.

4.6.2 MOS Environment

MIDAS is loaded from the system file by the system loader by means of the /ULOAD,MIDAS directive.

It reads the source records from PI and outputs them to PO. Pass two source input is from SS. When the END statement is encountered on pass one, the SS file is repositioned and reread. During pass two, the output can be directed to BO for the object module and to LO for the assembly listing. When an END statement is encountered on pass two, control is returned to MOS. Therefore, it is necessary to reload MIDAS with another /ULOAD directive if multiple assemblies are desired.

4.6.3 Stand-Alone Environment

MIDAS is loaded by the 620 stand-alone FORTRAN IV loader, along with the runtime I/O and runtime utility. The description of this loading procedure and subsequent execution is described in the Varian 620 FORTRAN IV Reference Manual, where MIDAS is substituted for the DAS MR Assembler. Upon execution, MIDAS will input source records from logical unit 3 (PI), output source records for pass two to logical unit 9 (PO), input pass two source records from logical unit 8 (SS), output binary object records to logical unit 2 (BO), and output assembly listing to logical unit 4 (LO). When the first assembly is completed, subsequent assemblies may be performed by restarting MIDAS at location 0541.

4.7 ASSEMBLER INPUT AND OUTPUT

The following section contains a description of the source records required for assembler input and the object records and listing produced by the assembler.

Source Records

The assembler input consists of a sequence of logical records containing 80 character positions. If a logical record contains more than 80 positions, only the first 80 are input and the remainder are ignored. If a record contains less than 80 positions, blank characters are supplied by the assembler to fill 80 character positions.

Only the first 72 are considered in the assembly process. Character positions 73 through 80 may be used as desired.

Listing Format

An assembly-listing page consists of 44 lines per page with each line containing no more than 120 characters. The lines per page count may be changed when running under an operating system. Each page contains the following:

- Page number and title line followed by a blank line
- Program listing containing two less than the current lines/page count

At the end of an assembly a symbol table will be printed followed by a line containing the message "mmmm ERRORS ASSEMBLY COMPLETE" where mmmm is the accumulated error count expressed as a decimal number.

The line format for the title line is a function of the environment in which MIDAS runs. The following description pertains to the standalone and MOS versions, with the comments in parentheses referring to VORTEX. Beginning with the first character position the format is illustrated below.

Object Code Records

MIDAS produces object code which is input for the microsimulator and the microutility programs. Logical records of the object code are a fixed length of 60 words. Word 1 is the record control word. Word 2 contains an exclusive OR checksum of word 1 and the remaining words of the record. Word 3 through 11 optionally contain a program identification block. Words 12 through the end of the record (or 3 through end of record if there is no program identification block) contain data fields.

Record Control Word Format

The format of the record control word is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
a	1	1	b	c	1	0	0	d	d	d	d	d	d	d	d

where a is 1 if the checksum is suppressed, b is 1 if not starting record, c is 1 when it is not the last record, and d is binary record number modulo 256.

Program Identification Block

This block appears in words 3 through 11 of the starting record of each program. Word 3 contains the highest value of the program counter during the assembly, words 4 through 7 contain an eight-character ASCII program identification, and words 8 through 11 contain an eight-character ASCII program creation date.

Data Field Format

Data fields contain either one- or four-word entries. One-word entries are loader control words, and four-word entries consist of data words.

The format of the loader control word is code in bits 13-15 and an address/count in the low-order 13 bits. A code of zero instructs the loader to ignore this entry. One is the code for setting the loading location counter to the value contained in bits 0 through 12. A value of two indicates the following microinstructions should be loaded. The number of microinstructions minus one is specified in bits 1 through 12.

Data Words

Data words contain microinstructions. Each microinstruction is comprised of four 16-bit words. Word 1 contains bits 63 through 48 of the microinstruction while word 4 contains bits 15 through 0 of the microinstruction. A microinstruction will not be carried across a logical record boundary. If insufficient space remains on a logical record for the four-word microinstruction, the remaining space will be ignored and the microinstruction started on the next logical record.

4.8 ADDING MIDAS TO VORTEX

The micro assembler resides on the background library under VORTEX. After system generation, the user must catalog it in the background library. The following procedure is used to do this.

1. Position the BI device to the microassembler object material.
2. Issue the following directives:

```
/LMGEN
TIDB,MIDAS,ONE,ZERO
LD,BI
LIB
END,BL,E
```

Detailed descriptions of these directives are in the VORTEX Reference Manual.

4.9 ASSEMBLY ERROR MESSAGES

During assembly the symbolic statements are checked for syntactic errors. In addition, a condition may occur where the assembler is unable to determine the correct meaning of the symbolic source statements.

Either case is indicated as an error and up to eight error codes will be output beneath the source statement incorrectly constructed.

NR, LC and IO errors terminate the assembly.

Each error code with the exception of IO is followed by a space and two decimal digits indicating the character position the assembler was scanning when the error was detected.

The error codes and their meanings are listed below.

Error Code	Meaning
AD	Address expression or associated fields in error (see below)
CC	Continuation not expected
CE	Numeric conversion error
DD	Illegal redefinition of a symbol
ER	Syntax error
EX	An expression contained an illegal construction
FN	Field number inconsistent with format
IO	I/O error
LC	Program location counter setting exceeds the maximum WCS page size (512 words)
MF	Duplicate field reference
NR	No memory available for addition of an entry to assembler's tables
NS	No symbol in the label field where required
OP	Operation field undefined
SE	Symbol in label field has a value during pass 2 that is different from the value determined in pass 1
SY	Undefined symbol. A value of zero is assumed
SZ	A value too large for the size of a field, or the fields defined in a format statement do not equal 64 bits

The AD error may occur under these circumstances:

a. With the character pointer in, or at the end of, an address expression:
   1. A test fail address is not on an even numbered word.
   2. A field select origin address is not on an even boundary.
   3. The displacement between the test pass and the test fail addresses is too great.

b. With the character pointer at the end of the operand field:
   1. Normal addressing mode and the FS or MT or TF field is not equal to zero.
   2. Test addressing mode is used and the TF field is equal to zero.
   3. Field selection addressing is the mode and the FS field is equal to zero.
   4. Test and field selection addressing mode and either the FS or TF field equals to zero.
   5. Page-jump addressing mode and either the FS or TF field is not equal to zero.

SECTION 5

CODING FROM FLOW DIAGRAMS

5.1 GENERAL

This section details the conversion of flow diagrams, (as developed in section 3), into code which MIDAS accepts. As examples actual assembler listings of the sample microprograms discussed in section 3 are included.

Flow diagram conversion is basically a matter of table-lookup. Tables are included in this section which list the standard mnemonics and the corresponding assembler code.

Assembler code produced is given in two different formats. The first format produces code using only the predefined assembler opcodes for the GEN or GMSK statements. The second format is based around user-defined opcodes which follow the mnemonics developed thus far as closely as possible. As these are not predefined, some burden is placed on the user to include the necessary EQU directives (these EQUs are available from Varian as a software part). However, the resulting code is considerably more readable than that produced in the first format.

Each column in the flow diagram will produce a single assembler program statement. This transformation can be performed as follows:

1. Fill in the label field if necessary, this will often be from the IDENT section.
2. Choose either GEN or GMSK as format label. The latter, GMSK, is used only when the 16-bit literal/mask is needed.
3. Derive the appropriate address expression
4. For each of the standard mnemonics in the column, add the statements shown in the conversion tables.
5. Replace any nonstandard mnemonics with appropriate field value assignments.

In addition to this translation, other assembler directives must be included to set the location of the program in WCS. In doing this, addressing considerations must be taken into account. For example, in test addressing the failure branch must always be to an even location.

The following table (5-1) lists the standard mnemonics and the assembler code they produce. Following the table, the EQU statements which define the format II opcodes are listed in table 5-2.

Table 5-1. Conversion Table

Row	Mnemonic	Format I	Format II
IDENT	None		
MEMORY FUNCTION	None		
MEMORY: REQUEST, ADDRESS	IF,OVR	IM0	10(IF$OVR)
	IF,ALU	IM4	10(IF$ALU)
	IF,P	IM8	10(IF$P)
	IF,MIR	IMC	10(IF$MIR)
	OF,OVR	IM1	10(OF$OVR)
	OF,ALU	IM5	10(OF$ALU)
	OF,P	IM9	10(OF$P)
	OF,MIR	IMD	10(OF$MIR)
	OS,OVR	IM2	10(OS$OVR)
	OS,ALU	IM6	10(OS$ALU)
	OS,P	IMA	10(OS$P)
	OS,MIR	IME	10(OS$MIR)
	BS,OVR	IM3	10(BS$OVR)
	BS,ALU	IM7	10(BS$ALU)
	BS,P	IMB	10(BS$P)
	BS,MIR	IMF	10(BS$MIR)
	Unconditional	SF1 (or SF2,TF0)	6(MEMC)[or 6(MEMC$),5(0)]

(continued)

Table 5-1. Conversion Table *(continued)*

Row	Mnemonic	Format I	Format II
	TESTT	SF3	6(TESTT)
	TESTF	SF2 (and not TF0)	6(TESTF)
	WAIT,MEMDN	SF0,IM1	6(SPEC),10(WAITMD)
ALU INPUT A	Rn	LA0,AAn	12(A$GPR),24(Rn)
	Rn,SL	LA2,AAn	12(A$GPRL),24(Rn)
	Rn,SR	LA3,AAn	12(A$GPRR),24(Rn)
	P	LA1	12(A$P)
	ZERO	LA0,SH1	12(A$SPEC),22(AZERO)
	ONES	LA0,SH2	12(A$SPEC),22(AONES)
			Note: 1) when using shifted general register user must specify high-low bits through SH field. 2) when using the GMSK format, use 16(Rn) instead of 24(Rn) and AKn instead of AAn.
ALU INPUT B	Rn	LB0,BBn	11(B$GPR),23(Rn)
	MIR	LB1,BB1	11(B$SPEC),23(MIR)
	IOR	LB1,BB2	11(B$SPEC),23(IOR)
	STAT	LB1,BB3	11(B$SPEC),23(STAT)
	LIT,x	LB3,MKy	11(LIT),15(y)
	MSK,x	LB2,MKy	11(MSK),15(y)
			Note: y is the one's complement of x
	OPR	LB1,BB0	11(B$SPEC),23(OPR)
	ORSE	LB1,BB4	11(B$SPEC),23(ORSE)
	OLSE	LB1,BB5	11(B$SPEC),23(OLSE)
	ORZF	LB1,BB6	11(B$SPEC),23(ORZF)
	OLZF	LB1,BB7	11(B$SPEC),23(OLZF)
ALU OUTPUT	ZERO	FF3,MF1	14(ZERO),15(LOG)
	ONES	FF3	14(ONES)
	TRNA	FFF,MF1	14(TRNA),15(LOG)
	TRNB	FFA,MF1	14(TRNB),15(LOG)
	INCA	CF3	14(INCA),16(CRY1)
	INCB	FF1,CF3	14(INCB),16(CRY1)
	DECA	FFF	14(DECA)
	DECB	FF9	14(DECB)
	ADD	FF9	14(ADD)
	SUB	FF6,CF3	14(SUB),16(CRY1)
	SHFA	FFC	14(SHFA)
	AND	FFB,MF1	14(AND),15(LOG)
	OR	FF1	14(OR)
	EOR	FF6,MF1	14(EOR),15(LOG)
	NOTA	FF0,MF1	14(NOTA),15(LOG)
	NOTB	FF5,MF1	14(NOTB),15(LOG)
	TCB	FF2,CF3	14(TCB),16(CRY1)
			Note: The mnemonics INCB and TCB require input A to be ZERO. Mnemonic DECB require input A to be ONES.

(continued)

Table 5-1. Conversion Table *(continued)*

Row	Mnemonic	Format I	Format II
ALU DESTINATION	Rn	WR1,AAn	17(GPROUT),24(Rn)
STATUS SAMPLE	SHFT	VF1	19(S$SHFT)
	OVFL	Refer to Table 2-7	
	ALU	TF0,SF0,GF2	TF0,SF0,7(S$ALU)
STATUS TEST	OVFL	GF0	7(OVFL)
	IOSR	GF1	7(IOSR)
	SSW3	GF2	7(SSW3)
	SSW2	GF3	7(SSW2)
	SSW1	GF4	7(SSW1)
	TFIR	GF5	7(TFIR)
	ALUO	GF6	7(ALUO)
	ALU5	GF7	7(ALUS)
	ALUC	GF8	7(ALUC)
	ALUZ	GF9	7(ALUZ)
	SHFT	GFA	7(SHFT)
	MIRS	GFB	7(MIRS)
	SFTC	GFC	7(SFTC)
	GPRS	GFD	7(GPRS)
	NORM	GFE	7(NORM)
	QUOS	GFF	7(QUOS)
			Note: TF field must also be set in test addressing.
ADDRESSING: MODE, ADDRESS	blank	/*	/*
	FSEL	/F(base),FSx	/F(base),FSx
	INT	user supplied	user supplied
	PJMP to n:		
	1) using stack	/N(word),TSn	/P(word + page)
	2) without memory	/N(word),TSn, SF0,TF0,IM3	/P(word + page), 10(PJMP),SF0,TF0
	3) with memory	/N(word),GF4, SF2,TF0	/P(word + page), 7(PJMP$),6(MEMC$),TF0
	POPJMP	TF0,SF0,IMD, LB3,AA4,BB0	10(STACK),24(POPJMP), LB3,TF0,SF0,BB0
	DECODE		
	1) with IBR to I	TF0,SF0,GF5	5(0),6(0),7(DECOD$)
	2) without IBR to I	TF0,SF0,GF4	5(0),6(0),7(DECODE)
	TESTT	/T(pass,fail), TF2	/T(pass,fail),5(TT)
	TESTF	/T(pass,fail), TF3	/T(pass,fail),5(FT)
SPECIAL ACTIONS	POUT	RF1	13(POUT)
	SCOUT	RF2	13(SCOUT)
	OPROUT	RF3	13(OPROUT)
	INCP	RF4	13(INCP)
	INCSC	RF5	13(INCSC)
	INCP,OPROUT	RF7	RF7

(continued)

Table 5-1. Conversion Table *(continued)*

Row	Mnemonic	Format I	Format II
	SHFTOP,LFT	SC1,WF0	18(SHFTOP),20(LFT)
	SHFTOP,RGHT	SC1,WF1	18(SHFTOP),21(RGHT)
			Note: on shifting OPR XF and AA fields used to determine high/low bits.
	IBR to I		
	with decode	TF0,SF0,GF5	TF0,SF0,7(DECOD$)
	without decode	TF0,SF0,GF1	TF0,SF0,7(IBR$I)
	PUSH,x	TF0,SF0,IMD, LB3,AK2,MKx	10(STACK),16(PUSH), 15(x),LB3,TF0,SF0
	POPDEL	TF0,SF0,IMD, BB1,AA4,LB3	10(STACK),23(POPDEL), LB3,TF0,SF0,AA4

Table 5-2 is the assembler directives needed for the user defined opcodes of format II. These are available on request as released software parts.

Table 5-2. User-Defined Opcodes

```
ADD      EQU   9
ALUC     EQU   8
ALUO     EQU   6
ALUS     EQU   7
ALUZ     EQU   9
AND      EQU   X'B
AONES    EQU   2
AZERO    EQU   1
A$GPR    EQU   0
A$GPRL   EQU   2
A$GPRR   EQU   3
A$P      EQU   1
A$SPEC   EQU   0

BS$ALU   EQU   7
BS$MIR   EQU   X'F
BS$OVR   EQU   3
BS$P     EQU   X'B
B$GPR    EQU   0
B$SPEC   EQU   1

CRY1     EQU   3

DECA     EQU   X'F
DECB     EQU   9
DECODE   EQU   4
DECOD$   EQU   5

EOR      EQU   6

FT       EQU   3

GPROUT   EQU   1
GPRS     EQU   X'D

IBR$I    EQU   1
IF$ALU   EQU   4
IF$MIR   EQU   X'C
IF$OVR   EQU   0
IF$P     EQU   8
INCA     EQU   0
INCB     EQU   1
INCP     EQU   4
INCSC    EQU   5
IOR      EQU   2
IOSR     EQU   1

LFT      EQU   0
LIT      EQU   3
LOG      EQU   1

MEMC$    EQU   2
MEMC     EQU   1
MIR      EQU   1
MIRS     EQU   X'B
MSK      EQU   2

NORM     EQU   X'E
NOTA     EQU   0
NOTB     EQU   5

OF$ALU   EQU   5
OF$MIR   EQU   X'D
```

(continued)

Table 5-2. User-Defined Opcodes *(continued)*

```
OF$OVR   EQU    1
OF$P     EQU    9
OLZF     EQU    7
OLSE     EQU    5
ONES     EQU    3
OPR      EQU    0
OPROUT   EQU    3
OR       EQU    1
ORSE     EQU    4
ORZF     EQU    6
OS$ALU   EQU    6
OS$MIR   EQU    X'E
OS$OVR   EQU    2
OS$P     EQU    X'A
OVFL     EQU    0

PJMPS    EQU    4
PJMP     EQU    3
POPDEL   EQU    1
POPJMP   EQU    4
POUT     EQU    1
PUSH     EQU    2

QUOS     EQU    X'F

R0       EQU    0
R1       EQU    1
R2       EQU    2
R3       EQU    3
R4       EQU    4
R5       EQU    5
R6       EQU    6
R7       EQU    7
R8       EQU    8
R9       EQU    9
RA       EQU    X'A
RB       EQU    X'B
RC       EQU    X'C
RD       EQU    X'D
RE       EQU    X'E
```

(continued)

```
RF       EQU    X'F
RGHT     EQU    1

SCOUT    EQU    2
SFTC     EQU    X'C
SHFA     EQU    X'C
SHFT     EQU    X'A
SHFTOP   EQU    1
SPEC     EQU    0
SSW1     EQU    4
SSW2     EQU    3
SSW3     EQU    2
STACK    EQU    X'D
STAT     EQU    3
SUB      EQU    6
S$ALU    EQU    2
S$SHFT   EQU    1

TCB      EQU    2
TESTT    EQU    3
TESTF    EQU    2
TFIR     EQU    5
TRNA     EQU    X'F
TRNB     EQU    X'A
TT       EQU    2

WAITMD   EQU    1

ZERO     EQU    3
```

5.2 EXAMPLES OF MICROPROGRAMS IN ASSEMBLY LANGUAGE

The five examples of section 3 were coded using the techniques outlined in this section. Comments on the encoding and actual assembler listings follow.

The first three examples use the equates in table 5-2.

5.2.1 BCS Entry Point Initialization

Since physical addresses were assigned at the flow diagram level, the transformation was quite straightforward. Note that a standard deck of all the EQU statements was used though not all were needed.

```
      *
      *
      *       THIS IS INITIALIZATION FOR BCS ENTRY POINTS
      *
      *

      *
      *       THE FOLLOWING ARE SUPPLEMENTAL OPCODES
      *       FOR USE WITH THE MICRO ASSEMBLER
      *
      *
0009  ADD     EQU    9
0008  ALUC    EQU    8
0006  ALUO    EQU    6
0007  ALUS    EQU    7
0009  ALUZ    EQU    9
000B  AND     EQU    X'B
0002  AONE    EQU    2
0001  AZERO   EQU    1
0000  A$GPR   EQU    0
0002  A$GPRL  EQU    2
0003  A$GPRR  EQU    3
0001  A$P     EQU    1
0000  A$SPEC  EQU    0
0007  BS$ALU  EQU    7
000F  BS$MIR  EQU    X'F
0003  BS$OVR  EQU    3
000B  BS$P    EQU    X'B
0000  B$GPR   EQU    0
0001  B$SPEC  EQU    1
0003  CRY1    EQU    3
000F  DECA    EQU    X'F
0009  DECB    EQU    9
0004  DECODE  EQU    4
0005  DECOD$  EQU    5
0006  EOR     EQU    6
0003  FT      EQU    3
0001  GPROUT  EQU    1
000D  GPRS    EQU    X'D
0001  IBR$I   EQU    1
0004  IF$ALU  EQU    4
000C  IF$MIR  EQU    X'C
0000  IF$OVR  EQU    0
0008  IF$P    EQU    8
0000  INCA    EQU    0
0001  INCB    EQU    1
0004  INCP    EQU    4

0005  INCSC   EQU    5
0002  IOR     EQU    2
0001  IOSR    EQU    1
0006  KOUT    EQU    6
0000  LFT     EQU    0
0003  LIT     EQU    3
0001  LOG     EQU    1
0001  MEMC    EQU    1
0002  MEMC$   EQU    2
0001  MIR     EQU    1
000B  MIRS    EQU    X'B
0002  MSK     EQU    2
000E  NORM    EQU    X'E
0000  NOTA    EQU    0
0005  NOTB    EQU    5
0005  OF$ALU  EQU    5
000D  OF$MIR  EQU    X'D
0001  OF$OVR  EQU    1
0009  OF$P    EQU    9
0007  OLZF    EQU    7
0005  OLSE    EQU    5
0003  ONES    EQU    3
0000  OPR     EQU    0
0003  OPROUT  EQU    3
0001  OR      EQU    1
0004  ORSE    EQU    4
0006  ORZF    EQU    6
0006  OS$ALU  EQU    6
000E  OS$MIR  EQU    X'E
0002  OS$OVR  EQU    2
```

(continued)

```
                000A   OS$P    EQU     X'A
                0000   OVFL    EQU     0
                0003   PJMP    EQU     3
                0004   PJMP$   EQU     4
                0001   POUT    EQU     1
                000F   QUOS    EQU     X'F
                0000   R0      EQU     0
                0001   R1      EQU     1
                0002   R2      EQU     2
                0003   R3      EQU     3
                0004   R4      EQU     4
                0005   R5      EQU     5

                0006   R6      EQU     6
                0007   R7      EQU     7
                0008   R8      EQU     8
                0009   R9      EQU     9
                000A   RA      EQU     X'A
                000B   RB      EQU     X'B
                000C   RC      EQU     X'C
                000D   RD      EQU     X'D
                000E   RE      EQU     X'E
                000F   RF      EQU     X'F
                0001   RGHT    EQU     1
                0002   SCOUT   EQU     2
                000C   SFTC    EQU     X'C
                000C   SHFA    EQU     X'C
                000A   SHFT    EQU     X'A
                0001   SHFTOP  EQU     1
                0000   SPEC    EQU     0
                0004   SSW1    EQU     4
                0003   SSW2    EQU     3
                0002   SSW3    EQU     2
                0003   STAT    EQU     3
                0006   SUB     EQU     6
                0002   S$ALU   EQU     2
                0006   S$OVFL  EQU     6
                0001   S$SHFT  EQU     1
                0002   TCB     EQU     2
                0003   TESTT   EQU     3
                0002   TESTF   EQU     2
                0005   TFIR    EQU     5
                000F   TRNA    EQU     X'F
                000A   TRNB    EQU     X'A
                0002   TT      EQU     2
                0001   WAITMD  EQU     1
                0003   ZERO    EQU     3


                       *
                       *       FOLLOWING ARE ROM STANDARD STATE ADDRESSES
                       *
                013E   SS1M    EQU     X'13E        RESTART PIPELINE @ P
                0092   SS2M    EQU     X'092        MAINTAIN PIPELINE
                002D   SS3M    EQU     X'02D        DECODE NEXT INSTRUCTION (IN IBR)

0000                           ORG     0

0000 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0001 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0002 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0003 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0004 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0005 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0006 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0007 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0008 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0009 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
000A 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
000B 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
000C 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
000D 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
000E 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
000F 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0010 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0011 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0012 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0013 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0014 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0015 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0016 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0017 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0018 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
0019 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
001A 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
001B 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
001C 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
001D 0490000180000000          GEN     /N(SS2M),10(PJMP),1(0)     RETURN TO ROM
```

(continued)

```
001E 0490000180000000 164         GEN      /N(SS2M),10(PJMP),1(0)       RETURN TO ROM
001F 0490000180000000 165         GEN      /N(SS2M),10(PJMP),1(0)       RETURN TO ROM

                      167         END
SYMBOLS
0000 A$GPR   0002 A$GPRL 0003 A$GPRR 0001 A$P    0000 A$SPEC
0009 ADD     0008 ALUC   0006 ALUO   0007 ALUS   0009 ALUZ
000B AND     0002 AONE   0001 AZERO  0000 B$GPR  0001 B$SPEC
0007 BS$ALU  000F BS$MIR 0003 BS$OVR 000B BS$P   0003 CRY1
000F DECA    0009 DECB   0005 DECOD$ 0004 DECODE 0006 EOR
0003 FT      0001 GPROUT 000D GPRS   0001 IBR$I  0004 IF$ALU
000C IF$MIR  0000 IF$OVR 0008 IF$P   0000 INCA   0001 INCB
0004 INCP    0005 INCSC  0002 IOR    0001 IOSR   0006 KOUT
0000 LFT     0003 LIT    0001 LOG    0001 MEMC   0002 MEMC$
0001 MIR     000B MIRS   0002 MSK    000E NORM   0000 NOTA
0005 NOTB    0005 OF$ALU 000D OF$MIR 0001 OF$OVR 0009 OF$P
0005 OLSE    0007 OLZF   0003 ONES   0000 OPR    0003 OPROUT
0001 OR      0004 ORSE   0006 ORZF   0006 OS$ALU 000E OS$MIR
0002 OS$OVR  000A OS$P   0000 OVFL   0003 PJMP   0004 PJMP$
0001 POUT    000F QUOS   0000 R0     0001 R1     0002 R2
0003 R3      0004 R4     0005 R5     0006 R6     0007 R7
0008 R8      0009 R9     000A RA     000B RB     000C RC
000D RD      000E RE     000F RF     0001 RGHT   0002 S$ALU
0006 S$OVFL  0001 S$SHFT 0002 SCOUT  000C SFTC   000C SHFA
000A SHFT    0001 SHFTOP 0000 SPEC   013E SS1M   0092 SS2M
002D SS3M    0004 SSW1   0003 SSW2   0002 SSW3   0003 STAT
0006 SUB     0002 TCB    0002 TESTF  0003 TESTT  0005 TFIR
000F TRNA    000A TRNB   0002 TT     0001 WAITMD 0003 ZERO
  0 ERRORS ASSEMBLY COMPLETE
```

5.2.2 Memory-to-Memory Block Move

The subroutine was assigned physical location 101, page 1 as its first address. This places word MBMA on an even word, as it must be. Since the microroutine is on page 1, the need for the page jump from the BCS entry point no longer exists. It was included never the less.

```
      *
      *
      *       MEMORY-TO-MEMORY BLOCK MOVE
      *
      *       CALL: BCS TO WORD 0
      *
      *       PARAMETERS:  A REG - 'TO' ADDRESS
      *                    B REG - 'FROM' ADDRESS
      *                    X REG - BLOCK LENGTH
      *
      *

0001  R1      EQU      1
      *
      *       THE FOLLOWING ARE SUPPLEMENTAL OPCODES
      *       FOR USE WITH THE MICRO ASSEMBLER
      *
      *
0009  ADD     EQU      9
0008  ALUC    EQU      8
0006  ALUO    EQU      6
0007  ALUS    EQU      7
0009  ALUZ    EQU      9
000B  AND     EQU      X'B
0002  AONE    EQU      2
0001  AZERO   EQU      1
0000  A$GPR   EQU      0
0002  A$GPRL  EQU      2
0003  A$GPRR  EQU      3
0001  A$P     EQU      1
0000  A$SPEC  EQU      0
0007  BS$ALU  EQU      7
000F  BS$MIR  EQU      X'F
0003  BS$OVR  EQU      3
000B  BS$P    EQU      X'B
0000  B$GPR   EQU      0
0001  B$SPEC  EQU      1
0003  CRY1    EQU      3
000F  DECA    EQU      X'F
0009  DECB    EQU      9
0004  DECODE  EQU      4
0005  DECOD$  EQU      5
0006  EOR     EQU      6
0003  FT      EQU      3
0001  GPROUT  EQU      1
000D  GPRS    EQU      X'D
0001  IBR$I   EQU      1
0004  IF$ALU  EQU      4
000C  IF$MIR  EQU      X'C
0000  IF$OVR  EQU      0
0008  IF$P    EQU      8
0000  INCA    EQU      0
0001  INCB    EQU      1

0004  INCP    EQU      4
0005  INCSC   EQU      5
0002  IOR     EQU      2
0001  IOSR    EQU      1
0006  KOUT    EQU      6
0000  LFT     EQU      0
0003  LIT     EQU      3
0001  LOG     EQU      1
0001  MEMC    EQU      1
0002  MEMC$   EQU      2
0001  MIR     EQU      1
000B  MIRS    EQU      X'B
0002  MSK     EQU      2
000E  NORM    EQU      X'E
0000  NOTA    EQU      0
0005  NOTB    EQU      5
0005  OF$ALU  EQU      5
000D  OF$MIR  EQU      X'D
0001  OF$OVR  EQU      1
0009  OF$P    EQU      9
0007  OLZF    EQU      7
```

(continued)

```
             0005  OLSE    EQU     5
             0003  ONES    EQU     3
             0000  OPR     EQU     0
             0003  OPROUT  EQU     3
             0001  OR      EQU     1
             0004  ORSE    EQU     4
             0006  ORZF    EQU     6
             0006  OS$ALU  EQU     6
             000E  OS$MIR  EQU     X'E
             0002  OS$OVR  EQU     2
             000A  OS$P    EQU     X'A
             0000  OVFL    EQU     0
             0003  PJMP    EQU     3
             0004  PJMP$   EQU     4
             0001  POUT    EQU     1
             000F  QUOS    EQU     X'F
             0000  R0      EQU     0
             0002  R2      EQU     2
             0003  R3      EQU     3
             0004  R4      EQU     4
             0005  R5      EQU     5
             0006  R6      EQU     6
             0007  R7      EQU     7
             0008  R8      EQU     8
             0009  R9      EQU     9
             000A  RA      EQU     X'A
             000B  RB      EQU     X'B
             000C  RC      EQU     X'C
             000D  RD      EQU     X'D
             000E  RE      EQU     X'E
             000F  RF      EQU     X'F
             0001  RGHT    EQU     1
             0002  SCOUT   EQU     2
             000C  SFTC    EQU     X'C
             000C  SHFA    EQU     X'C
             000A  SHFT    EQU     X'A
             0001  SHFTOP  EQU     1
             0000  SPEC    EQU     0
             0004  SSW1    EQU     4
             0003  SSW2    EQU     3
             0002  SSW3    EQU     2
             0003  STAT    EQU     3
             0006  SUB     EQU     6
             0002  S$ALU   EQU     2
             0006  S$OVFL  EQU     6
             0001  S$SHFT  EQU     1
             0002  TCB     EQU     2
             0003  TESTT   EQU     3
             0002  TESTF   EQU     2
             0005  TFIR    EQU     5
             000F  TRNA    EQU     X'F
             000A  TRNB    EQU     X'A
             0002  TT      EQU     2
             0001  WAITMD  EQU     1
             0003  ZERO    EQU     3

                   *
                   *       FOLLOWING ARE ROM STANDARD STATE ADDRESSES
                   *
             013E  SS1M    EQU     X'13E        RESTART PIPELINE @ P
             0092  SS2M    EQU     X'092        MAINTAIN PIPELINE
             002D  SS3M    EQU     X'02D        DECODE NEXT INSTRUCTION (IN IBR)

0000                               ORG     0

                   *
                   *       FOLLOWING IS BCS ENTRY POINT

0000 1808000180000000              GEN     /N(MBM),10(PJMP),1(1)    BRANCH TO BLOCK MOVE ROUTINE

                   *
                   *       FOLLOWING IS ACTUAL BLOCK MOVE ROUTINE
                   *

0101                               ORG     X'101

                   *
                   *       SAVE P IN R7

0101 0810000008F90007      MBM     GEN     /*,12(A$P),14(TRNA),15(LOG),17(GPROUT),24(R7)

                   *
                   *       DECR 'TO' ADDR

0102 0818000000F10000              GEN     /*,12(A$GPR),24(R0),14(DECA),17(GPROUT)

                   *
                   *       DECR 'FROM' ADDR ; PUT IT IN P
```

(continued)

```
0103 0820000001F00001 164         GEN     /*,12(A$GPR),24(R1),14(DECA),13(POUT)

                      166  *
                      167  *      FIRST LOOP MICROWORD; STORE AT 'TO'; REQUEST FETCH OF INCR 'FROM'

                      169  MBMA   GEN     /*,10(OF$P),6(MEMC),11(B$SPEC),23(MIR),14(TRNB),15(LOG),
0104 08280404A4A80010 170                 C13(INCP)

                      172  *
                      173  *      SECOND LOOP MICROWORD; DECR BLOCK LENGTH; SAMPLE RESULT FOR TEST

0105 0830008000F10002 175         GEN     /*,12(A$GPR),24(R2),14(DECA),17(GPROUT),7(S$ALU)

                      177  *
                      178  *      FINAL LOOP MICROWORD; EXIT OR CONTINUE THE LOOP WITH REQUEST

                      179  *      FOR A STORE AT INCREMENTED 'TO' ADDR

                      181         GEN     /T(MBMB,MBMA),5(TT),10(OS$ALU),6(TESTF),
0106 283829C300070000 182                 C12(A$GPR),24(R0),14(INCA),16(CRY1),17(GPROUT),7(ALUS)

                      184  *
                      185  *      EXIT MICROWORD ; RESTORE P AND THE PIPELINE

                      187  MBMB   GEN     /N(SS2M),7(PJMP$),1(0),10(IF$ALU),6(MEMC$),5(0),
0107 0168090201F80007 188                C12(A$GPR),24(R7),14(TRANA),16(CRY1),13(POUT)

                      190         END
```

```
SYMBOLS
0000 A$GPR  0002 A$GPRL 0003 A$GPRR 0001 A$P    0000 A$SPEC
0009 ADD    0008 ALUC   0006 ALUO   0007 ALUS   0009 ALUZ
000B AND    0002 AONE   0001 AZERO  0000 B$GPR  0001 B$SPEC
0007 BS$ALU 000F BS$MIR 0003 BS$OVR 000B BS$P   0003 CRY1
000F DECA   0009 DECB   0005 DECOD$ 0004 DECODE 0006 EOR
0003 FT     0001 GPROUT 000D GPRS   0001 IBR$I  0004 IF$ALU
000C IF$MIR 0000 IF$OVR 0008 IF$P   0000 INCA   0001 INCB
0004 INCP   0005 INCSC  0002 IOR    0001 IOSR   0006 KOUT
0000 LFT    0003 LIT    0001 LOG    0101 MBM    0104 MBMA
0107 MBMB   0001 MEMC   0002 MEMC$  0001 MIR    000B MIRS
0002 MSK    000E NORM   0000 NOTA   0005 NOTB   0005 OF$ALU
000D OF$MIR 0001 OF$OVR 0009 OF$P   0005 OLSE   0007 OLZF
0003 ONES   0000 OPR    0003 OPROUT 0001 OR     0004 ORSE
0006 ORZF   0006 OS$ALU 000E OS$MIR 0002 OS$OVR 000A OS$P
0000 OVFL   0003 PJMP   0004 PJMP$  0001 POUT   000F QUOS
0000 R0     0001 R1     0002 R2     0003 R3     0004 R4
0005 R5     0006 R6     0007 R7     0008 R8     0009 R9
000A RA     000B RB     000C RC     000D RD     000E RE
000F RF     0001 RGHT   0002 S$ALU  0006 S$OVFL 0001 S$SHFT
0002 SCOUT  000C SFTC   000C SHFA   000A SHFT   0001 SHFTOP
0000 SPEC   013E SS1M   0092 SS2M   002D SS3M   0004 SSW1
0003 SSW2   0002 SSW3   0003 STAT   0006 SUB    0002 TCB
0002 TESTF  0003 TESTT  0005 TFIR   000F TRNA   000A TRNB
0002 TT     0001 WAITMD 0003 ZERO
  0 ERRORS ASSEMBLY COMPLETE
```

5.2.3 Reentrant Subroutine Call and Return

These routines were assigned locations beginning at word 110, page 1. As with the previous example, the page jumps are no longer necessary since the routines are on the same page as their BCS entry points. In this case they were simply coded using normal addressing.

```
           *
           *
           *       REENTRANT SUBROUTINE CALL AND RETURN
           *
           *       CALL: FOR SUBROUTINE CALL : BCS TO WORD 1
           *             FOR SUBROUTINE RETURN: BCS TO WORD 2
           *
           *       PARAMETERS:  B REGISTER - STACK POINTER
           *
           *

           *
           *       THE FOLLOWING ARE SUPPLEMENTAL OPCODES
           *       FOR USE WITH THE MICRO ASSEMBLER
           *
           *
0009       ADD     EQU     9
0008       ALUC    EQU     8
0006       ALUO    EQU     6
0007       ALUS    EQU     7
0009       ALUZ    EQU     9
000B       AND     EQU     X'B
0002       AONE    EQU     2
0001       AZERO   EQU     1
0000       A$GPR   EQU     0
0002       A$GPRL  EQU     2
0003       A$GPRR  EQU     3
0001       A$P     EQU     1
0000       A$SPEC  EQU     0
0007       BS$ALU  EQU     7
000F       BS$MIR  EQU     X'F
0003       BS$OVR  EQU     3
000B       BS$P    EQU     X'B
0000       B$GPR   EQU     0
0001       B$SPEC  EQU     1
0003       CRY1    EQU     3
000F       DECA    EQU     X'F
0009       DECB    EQU     9
0004       DECODE  EQU     4
0005       DECOD$  EQU     5
0006       EOR     EQU     6
0003       FT      EQU     3
0001       GPROUT  EQU     1
000D       GPRS    EQU     X'D
0001       IBR$I   EQU     1
0004       IF$ALU  EQU     4
000C       IF$MIR  EQU     X'C
0000       IF$OVR  EQU     0
0008       IF$P    EQU     8
0000       INCA    EQU     0
0001       INCB    EQU     1
0004       INCP    EQU     4

0005       INCSC   EQU     5
0002       IOR     EQU     2
0001       IOSR    EQU     1
0006       KOUT    EQU     6
0000       LFT     EQU     0
0003       LIT     EQU     3
0001       LOG     EQU     1
0001       MEMC    EQU     1
0002       MEMC$   EQU     2
0001       MIR     EQU     1
000B       MIRS    EQU     X'B
0002       MSK     EQU     2
000E       NORM    EQU     X'E
0000       NOTA    EQU     0
0005       NOTB    EQU     5
0005       OF$ALU  EQU     5
000D       OF$MIR  EQU     X'D
0001       OF$OVR  EQU     1
0009       OF$P    EQU     9
0007       OLZF    EQU     7
0005       OLSE    EQU     5
0003       ONES    EQU     3
```

(continued)

```
0000   OPR    EQU    0
0003   OPROUT EQU    3
0001   OR     EQU    1
0004   ORSE   EQU    4
0006   ORZF   EQU    6
0006   OS$ALU EQU    6
000E   OS$MIR EQU    X'E
0002   OS$OVR EQU    2
000A   OS$P   EQU    X'A
0000   OVFL   EQU    0
0003   PJMP   EQU    3
0004   PJMP$  EQU    4
0001   POUT   EQU    1
000F   QUOS   EQU    X'F
0000   R0     EQU    0
0001   R1     EQU    1
0002   R2     EQU    2
0003   R3     EQU    3
0004   R4     EQU    4
0005   R5     EQU    5

0006   R6     EQU    6
0007   R7     EQU    7
0008   R8     EQU    8
0009   R9     EQU    9
000A   RA     EQU    X'A
000B   RB     EQU    X'B
000C   RC     EQU    X'C
000D   RD     EQU    X'D
000E   RE     EQU    X'E
000F   RF     EQU    X'F
0001   RGHT   EQU    1
0002   SCOUT  EQU    2
000C   SFTC   EQU    X'C
000C   SHFA   EQU    X'C
000A   SHFT   EQU    X'A
0001   SHFTOP EQU    1
0000   SPEC   EQU    0
0004   SSW1   EQU    4
0003   SSW2   EQU    3
0002   SSW3   EQU    2
0003   STAT   EQU    3
0006   SUB    EQU    6
0002   S$ALU  EQU    2
0006   S$OVFL EQU    6
0001   S$SHFT EQU    1
0002   TCB    EQU    2
0003   TESTT  EQU    3
0002   TESTF  EQU    2
0005   TFIR   EQU    5
000F   TRNA   EQU    X'F
000A   TRNB   EQU    X'A
0002   TT     EQU    2
0001   WAITMD EQU    1
0003   ZERO   EQU    3

       *
       *      FOLLOWING ARE ROM STANDARD STATE ADDRESSES
       *
013E   SS1M   EQU    X'13E        RESTART PIPELINE @ P
0092   SS2M   EQU    X'092        MAINTAIN PIPELINE
002D   SS3M   EQU    X'02D        DECODE NEXT INSTRUCTION (IN IBR)

       *
       *      FOLLOWING IS CODE FOR SUBROUTINE CALL
       *

0001                  ORG    1

       *
       *      BCS ENTRY POINT PUSHES OLD R2 ON STACK

                      GEN    /N(LAB1),10(OS$ALU),6(MEMC),12(A$GPR),24(R1),14(DECA),
0001 0880040300F10001        C17(GPROUT)

       *
       *      REST OF ROUTINE
       *

0110                  ORG    X'110

       *
       *      WAIT ON STORE OF R2

0110 0888000080F80002 LAB1   GEN    /*,6(SPEC),10(WAITMD),12(A$GPR),24(R2),14(TRNA),15(LOG)
```

(continued)

```
                               *
                               *      FETCH FIRST INSTRUCTION OF SUBR ; STORE INCR P IN R2

                                      GEN     /*,10(IF$MIR),6(MEMC),12(A$P),14(INCA),16(CRY1),
0111 0890040608070002                        C17(GPROUT),24(R2)

                               *
                               *      FETCH SECOND INST OF SUBR; SET NEW P; BACK TO ROM

                                      GEN     /N(SS3M),7(PJMP$),1(0),10(IF$ALU),6(MEMC$),5(0),
                                             C12(A$SPEC),22(AZERO),
0112 0168090221160110                        C11(B$SPEC),23(MIR),14(INCB),16(CRY1),13(POUT)


                               *
                               *      FOLLOWING IS CODE FOR SUBROUTINE RETURN
                               *

0002                                  ORG     2

                               *
                               *      BCS ENTRY POINT - BEGINS FETCH OF INST AT RETURN ADDRESS

                                      GEN     /N(LAB2),10(IF$ALU),6(MEMC),12(A$GPR),24(R2),
0002 08A8040201F80002                        C14(TRNA),15(LOG),13(POUT)

                               *
                               *      REST OF THE ROUTINE
                               *

0115                                  ORG     X'115

                               *
                               *      FETCH OLD R2 VALUE FROM STACK

0115 08B0040280F80001          LAB2   GEN     /*,10(OF$ALU),6(MEMC),12(A$GPR),24(R1),14(TRNA),15(LOG)

                               *
                               *      FETCH SECOND INSTRUCTION AT RETURN ADDRESS ; INCR STK PTR

                                      GEN     /*,10(IF$P),6(MEMC),12(A$GPR),24(R1),14(INCA),16(CRY1),
0116 08B8040404070001                        C17(GPROUT),13(INCP)

                               *
                               *      RESTORE R2 ; BACK TO ROM

                                      GEN      10(PJMP),1(0),7(DECOD$),11(B$SPEC),23(MIR),
0117 00000141A0A90012                        C14(TRNB),15(LOG),17(GPROUT),24(R2)

                                      END
SYMBOLS
0000 A$GPR  0002 A$GPRL 0003 A$GPRR 0001 A$P    0000 A$SPEC
0009 ADD    0008 ALUC   0006 ALUO   0007 ALUS   0009 ALUZ
000B AND    0002 AONE   0001 AZERO  0000 B$GPR  0001 B$SPEC
0007 BS$ALU 000F BS$MIR 0003 BS$OVR 000B BS$P   0003 CRY1
000F DECA   0009 DECB   0005 DECOD$ 0004 DECODE 0006 EOR
0003 FT     0001 GPROUT 000D GPRS   0001 IBR$I  0004 IF$ALU
000C IF$MIR 0000 IF$OVR 0008 IF$P   0000 INCA   0001 INCB
0004 INCP   0005 INCSC  0002 IOR    0001 IOSR   0006 KOUT
0110 LAB1   0115 LAB2   0000 LFT    0003 LIT    0001 LOG
0001 MEMC   0002 MEMC$  0001 MIR    000B MIRS   0002 MSK
000E NORM   0000 NOTA   0005 NOTB   0005 OF$ALU 000D OF$MIR
0001 OF$OVR 0009 OF$P   0005 OLSE   0007 OLZF   0003 ONES
0000 OPR    0003 OPROUT 0001 OR     0004 ORSE   0006 ORZF
0006 OS$ALU 000E OS$MIR 0002 OS$OVR 000A OS$P   0000 OVFL
0003 PJMP   0004 PJMP$  0001 POUT   000F QUOS   0000 R0
0001 R1     0002 R2     0003 R3     0004 R4     0005 R5
0006 R6     0007 R7     0008 R8     0009 R9     000A RA
000B RB     000C RC     000D RD     000E RE     000F RF
0001 RGHT   0002 S$ALU  0006 S$OVFL 0001 S$SHFT 0002 SCOUT
000C SFTC   000C SHFA   000A SHFT   0001 SHFTOP 0000 SPEC
013E SS1M   0092 SS2M   002D SS3M   0004 SSW1   0003 SSW2
0002 SSW3   0003 STAT   0006 SUB    0002 TCB    0002 TESTF
0003 TESTT  0005 TFIR   000F TRNA   000A TRNB   0002 TT
0001 WAITMD 0003 ZERO
  0 ERRORS ASSEMBLY COMPLETE
```

5.2.4 64K Add to General-Purpose Register

```
                           *ADD TO ANY REGISTER FROM 64K MEMORY INDEX BY R1
                           *
0000                              ORG      0
                           *
0000 0100040404000000      AD1    GEN      /N(AD2),SF1,IM8,RF4
                           *
                           *THIS ENTRY USED FOR EVEN REGISTER ADDRESSES.
                           *INITIATE ANOTHER INSTRUCTION FETCH USING INCREMENTED PROGRAM COUNTER.
                           *
0010                              ORG      X'010
                           *
0010 0100040404000000      AD1A   GEN      /N(AD2),SF1,IM8,RF4
                           *
                           *THIS ENTRY USED FOR ODD REGISTER ADDRESSES.
                           *INITIATE ANOTHER INSTRUCTION FETCH USING INCREMENTED PROGRAM COUNTER.
                           *
0020                              ORG      X'020
                           *
0020 0108000023A80010      AD2    GEN      /*,LB1,RF3,FFA,MF1,BB1
                           *
                           *TRANSFER MEMORY INPUT REGISTER TO OPERAND REGISTER TO PREVENT LOSS
                           *DUE TO PREVIOUSLY INITIATED FETCH.  THIS IS THE BASE ADDRESS.
                           *
0021 01100402A0900001      AD3    GEN      /*,SF1,IM5,LB1,LA0,FF9,AA1
                           *
                           *PERFORM INDEXING BY ADDING R1 TO OPERAND REGISTER.  INITIATE OPERAND
                           *FETCH USING ALU OUTPUT.
                           *
0022 4118043404000010      AD4    GEN      /*,TS4,MR1,AB2,BB1,SF1,IM8,RF4
                           *
                           *FIELD SELECT REGISTER SPECIFICATION FROM INSTRUCTION BITS 4-7 TO
                           *A FIELD OF MICROINSTRUCTION.  SET B FIELD TO SELECT MEMORY INPUT
                           *REGISTER.  INITIATE ANOTHER INSTRUCTION FETCH USING INCREMENTED
                           *PROGRAM COUNTER.
                           *
0023 000003C1A0910000      AD5    GEN      /P(X'0000),LB1,LA0,FF9,GFF,WR1,IM3
                           *
                           *ADD CONTENTS OF MEMORY INPUT REGISTER TO THAT OF PREVIOUSLY SELECTED
                           *REGISTER AND STORE BACK THE SUM.  PAGE BRANCH TO ZERO AND DECODE
                           *INSTRUCTION PREVIOUSLY FETCHED.  OVERFLOW AND CONDITION CODES ARE
                           *SAMPLED.  TRANSFER INSTRUCTION BUFFER TO INSTRUCTION REGISTER.
                           *

                                  END
SYMBOLS
0000 AD1     0010 AD1A    0020 AD2     0021 AD3     0022 AD4
0023 AD5
  0 ERRORS ASSEMBLY COMPLETE
```

5.2.5 Cyclic Redundancy Check Generation

```
                        *THIS MICROPROGRAM COMPUTES THE CYCLIC REDUNDANCY CHECK WORD ON A
                        *PACKED BYTE ARRAY USING THE POLYNOMIAL:
                        *          X**16+X**15+X**2+1
                        *ENTRY IS VIA A BCS TO LOCATION 0 OF PAGE 1
                        *THE WORD FOLLOWING THE BCS IS THE DATA ARRAY ADDRESS
                        *THE WORD FOLLOWING THE DATA ARRAY ADDRESS IS THE BYTE COUNT
                        *
                        *THE 16 BIT CRC IS LEFT IN RO
                        *R0,R1,AND R2 ARE ALL USED BY THIS INSTRUCTION (A,B,X). RF IS ALSO USED.
                        *R0 IS THE CURRENT CRC
                        *R1 IS THE CURRENT WORD ADDRESS OF THE DATA
                        *R2 IS THE CURRENT BYTE COUNT
                        *RF CONTAINS THE CRC POLYNOMIAL B'1000000000000101
                        *THE MICROPROGRAM MAY BE INTERRUPTED AFTER EVERY TWO BYTES ARE PROCESSED
                        *IF THE OVERFLOW FLAG IS SET UPON ENTRY THE CURRENT VALUES OF R1 AND
                        *R2 ARE USED INSTEAD OF THOSE SPECIFIED BY MEMORY CONTENTS.
                        *THE ACCUMULATOR (R0) SHOULD BE CLEARED PRIOR TO ENTRY UNLESS CRC IS TO
                        *BE ACCUMULATED WITH A PRIOR CRC VALUE.
                        *
                        *
                        *TYPICAL ENTRY SEQUENCE IS:
                        *       TZA
                        *       ROF
                        *       DATA     0105000
                        *       DATA     ADDR
                        *       DATA     COUNT
                        *
                        *
                        *CRC GENERATION
                        *
0000                            ORG      X'0
0000 01083804E7A7FFAF   CRC1    GMSK     /T(CRC2,CRC1A),TF3,SF2,IM9,LB3,RF7,FFA,MK7FFA,AKF
                        *
                        *ENTRY IS FROM DECODE OF THE BCS.  THE ADDRESS FETCH HAS BEEN INITIATED.
                        *OVERFLOW FLAG IS TESTED TO DETERMINE IF INSTRUCTION WAS INTERRUPTED
                        *FETCH OF BYTE COUNT IS INITIATED USING INCREMENTED PROGRAM COUNTER
                        *THE POLYNOMIAL IS PLACED IN OPR
                        *IF OVERFLOW IS ON GO TO CRC1A OTHERWISE CRC2
                        *
0020                            ORG      X'020
0020 0110040280A80010   CRC1A   GEN      /N(CRC17),SF1,IM5,FFA,BB1,MF1
                        *

                        *COME HERE IF OVERFLOW FLAG WAS ON WHEN INSTRUCTION WAS FETCHED
                        *FETCH DATA BYTE PAIR
                        *
0021 0198000020A90011   CRC2    GEN      /N(CRC3),LB1,FFA,WR1,BB1,AA1,MF1
                        *
                        *SAVE DATA ARRAY ADDRESS IN R1 (FROM MIR)
                        *
0022 01380000E2A00070   CRC17   GMSK     /N(CRC6),IM1,LB3,RF2,FFA,MK0007
                        *
                        *SET SHIFT COUNTER TO -8
                        *WAIT FOR MEMORY DONE FROM DATA FETCH
                        *
0023 0120008020A90012   CRC4    GEN      /*,GF2,LB1,FFA,BB1,MF1,AA2,WR1
                        *
                        *SAVE BYTE COUNT IN R2
                        *SAMPLE ALU STATUS TO CHECK FOR ZERO BYTE COUNT
                        *
0024 31282240E2A00070   CRC5    GMSK     /T(CRC18,CRC5A),TF2,GF9,IM1,LB3,RF2,FFA,MK0007
                        *
                        *PUT -8 IN SHIFT COUNTER (8 BITS PER BYTE)
                        *TEST ALU ZERO STATUS FLAG TO SEE IF BYTE COUNT WAS ZERO
                        *WAIT FOR MEMORY DONE FROM DATA FETCH
                        *IF BYTE COUNT WAS ZERO GO TO CRC18 OTHERWISE CRC5A
                        *
0025 0158050404000000   CRC18   GEN      /N(CRC19),SF1,GF4,IM8,RF4
                        *
                        *WHEN BYTE COUNT WENT TO ZERO RESET OVERFLOW TO INDICATE COMPLETION
                        *START NEXT INSTRUCTION FETCH USING INCREMENTED PROGRAM COUNTER
                        *
0026 0138000020A9000F   CRC5A   GEN      /*,FFA,MF1,AAF,WR1,LB1
                        *
                        *MOVE POLYNOMIAL (IN OPR) TO RF
                        *
0027 0150000023A80010   CRC6    GEN      /N(CRC7),LB1,RF3,FFA,BB1,MF1
                        *
                        *TRANSFER DATA BYTES FROM MIR TO OPR
                        *
0028 01500000006900F0   CRC9    GEN      /N(CRC7),FF6,MF1,WR1,BBF
                        *
                        *THIS IS A CORRECTION CYCLE
                        *R0 TO ALU INPUT A
```

(continued)

```
                      *RF TO ALU INPUT B

                      *EXCLUSIVE OR ALU INPUTS TO R0
                      *
0029 0190808000610032 CRC10  GEN     2(X'032),MT0,FS2,GF2,FF6,MF0,AA2,BB3,WR1
                      *
                      *AFTER LAST BIT IS PROCESSED TEST DSB FLAG FOR A CORRECTION CYCLE
                      *DECREMENT BYTE COUNT
                      *SAMPLE ALU STATUS TO ALLOW CHECK FOR BYTE COUNT ZERO
                      *IF CORRECTION CYCLE NECESSARY GO TO CRC10A OTHERWISE CRC11
                      *
                      CRC7   GEN     /T(CRC10,CRC8),TF2,GFC,LA2,RF5,FF6,MF1,WR1,SC1,VF1,
002A 714823001569DAF0                CXF3,SH2,BBF
                      *
                      *SHIFT R0 LEFT TO ALU INPUT A
                      *SHIFT OPR LEFT
                      *R0(15) TO SHIFT FLAG (DSB)
                      *OPR(15) TO ALU INPUT A BIT 00
                      *POLYNOMIAL (RF) TO ALU INPUT B
                      *EXCLUSIVE OR ALU INPUTS TO R0
                      *INCREMENT SHIFT COUNTER
                      *TEST FOR SHIFT COUNTER OVERFLOW, IF OVERFLOW GO TO CRC8 OTHERWISE CRC10
                      *
002B 0490090000000000 CRC19  GEN     /P(X'0092),SF2,GF4
                      *
                      *PAGE JUMP TO PAGE 0 LOC 060 (SS2M)
                      *
002C 0178000069900030 CRC22  GMSK    /N(CRC23),LB3,LA1,RF1,FF9,MK0003
                      *
                      *SUBTRACT 4 FROM PROGRAM COUNTER TO CAUSE REFETCH OF THE BCS INSTRUCTION
                      *AFTER INTERRUPT PROCESSING
                      *
002D 0178050404000000 CRC24  GEN     /N(CRC23),SF1,GF4,IM8,RF4
002E 4110800000000000 CRC8   GEN     /F(CRC9),FS2,2(X'022),TS4
                      *
                      *TEST SHIFT (DSB) FLAG TO SEE IF CORRECTION CYCLE IS NEEDED.  IF BIT 15
                      *OF THE OLD CRC WAS A ZERO THE EXCLUSIVE OR PERFORMED AT CRC7 MUST
                      *BE CANCELLED.  IF DSB WAS 1 GO TO CRC7 OTHERWISE CRC10
                      *
002F 01B0000100000000 CRC23  GEN     /N(CRC25),IM2
                      *
                      *WAIT FOR IO DONE
                      *
0030 01900000006900F0 CRC10A GEN     /N(CRC11),FF6,MF1,WR1,BBF

                      *
                      *THIS IS CORRECTION CYCLE SIMILAR TO CRC8
                      *
0031 6168224000070001 CRC21  GEN     /T(CRC24,CRC22),TF2,GF9,FF0,MF0,CF3,WR1,AA1
                      *
                      *INCREMENT DATA ARRAY ADDRESS (R1)
                      *TEST ALU ZERO FLAG FOR ZERO BYTE COUNT IF ALU ZERO IS ON GO TO CRC24
                      *OTHERWISE CRC22
                      *
0032 D128224062A00070 CRC11  GMSK    /T(CRC18,CRC12),TF2,GF9,LB3,RF2,FFA,MK0007
                      *
                      *PUT -8 INTO SHIFT COUNTER
                      *TEST ALU ZERO STATUS FLAG TO SEE IF RIGHT BYTE SHOULD BE PROCESSED
                      *IF SO GO TO CRC12 OTHERWISE CRC18
                      *
0033 0118048280A80010 CRC3   GEN     /N(CRC4),SF1,GF2,IM5,FFA,BB1,MF1
                      *
                      *USING R1 AS ADDRESS INITIATE FETCH OF TWO BYTES.
                      *SET OVERFLOW FLAG TO INDICATE INCOMPLETE INSTRUCTION
                      *
0034 4190800000000000 CRC13  GEN     /F(CRC14),FS2,2(X'032),TS4
                      *
                      *IDENTICAL TO CRC8
                      *
0035 41F0808000610032 CRC15  GEN     1(X'4),2(X'03E),MT0,FS2,GF2,FF6,MF0,AA2,BB3,WR1
                      *
                      *PERFORM OPERATIONS OF CRC10.  IF DSB IS SET GO TO CRC15B OTHERWISE
                      *CRC15A
                      *
0036                         ORG     X'036
0036 07F8000180000000 CRC25  GEN     /P(X'00FF),IM3
                      *PAGE JUMP TO PAGE 0 LOC 0FF (INT2)
                      *
0037                         ORG     X'037
0037 71FC012700000000 CRC20  GEN     2(CRC16),1(X'7),MT1,GF4,MR1,IME
                      *
                      *WHEN CRC15 DETECTS AN INTERRUPT CHECK IT AGAIN TO SEE IF IT WAS
                      *OVERRIDEN BY A DMA TRAP.
                      *START IO INTERRUPT SEQUENCE
                      *IF INTERRUPT GO TO CRC21 OTHERWISE CRC16
                      *
0038 01D00000006900F0 CRC14  GEN     /N(CRC12),FF6,MF1,WR1,BBF

                      *
```

(continued)

```
                          *IDENTICAL TO CRC9
                          *
003E                             ORG     X'03E
                          *
003E 71F8010600000000     CRC15B GEN     1(X'7),2(X'03F),GF4,IMC
                          *
                          *LOOK FOR INTERRUPT
                          *
                          *
003F 11282A4280070001     CRC16  GEN     /T(CRC18,CRC17),TF2,SF2,GF9,IM5,FF0,CF3,AA1,WR1
                          *
                          *INCREMENT ARRAY ADDRESS (R1)
                          *FETCH NEXT BYTE PAIR IF ALU ZERO FLAG IS OFF (BYTE COUNT NOT ZERO)
                          *IF BYTE COUNT WAS ZERO GO TO CRC18 OTHERWISE CRC17
                          *
003A                             ORG     X'03A
                          CRC12  GEN     /T(CRC15,CRC13),TF2,GFC,LA2,RF5,FF6,MF1,WR1,SC1,XF3,
003A 21A823001569DAF0                   CSH2,BBF,VF1
                          *
                          *IDENTICAL TO CRC7.  THIS PROCESSES THE RIGHT BYTE OF DATA WHICH HAS
                          *BEEN SHIFTED LEFT IN OPR
                          *
003C                             ORG     X'03C
003C 01F00000006900F0     CRC15A GEN     /N(CRC15B),FF6,MF1,WR1,BBF
                          *
                          *IDENTICAL TO CRC10A
                          *
                          *
                                 END
SYMBOLS
0000 CRC1    0029 CRC10   0030 CRC10A 0032 CRC11   003A CRC12
0034 CRC13   0038 CRC14   0035 CRC15  003C CRC15A 003E CRC15B
003F CRC16   0022 CRC17   0025 CRC18  002B CRC19   0020 CRC1A
0021 CRC2    0037 CRC20   0031 CRC21  002C CRC22   002F CRC23
002D CRC24   0036 CRC25   0033 CRC3   0023 CRC4    0024 CRC5
0026 CRC5A   0027 CRC6    002A CRC7   002E CRC8    0028 CRC9
  0 ERRORS ASSEMBLY COMPLETE
```

SECTION 6
MICROPROGRAM SIMULATOR, MICSIM

The Microprogram Simulator (MICSIM) helps the user find and correct microprogram bugs. Any program development includes some time to verify that the program solves the problem. Testing may find that it does not. Running the microprogram simulator aids in both the discovery and correction of microprogram errors.

When the microprogram is free of errors, the simulator can be used to determine the performance before the design is final, measure the efficiency of the technique and evaluate changes and extensions.

MICSIM runs on all V70 series systems. Microprograms can also be simulated on 620 systems without WCS. The hardware requirements depend upon the operating system used.

6.1 BASIC ELEMENTS

In general this simulator provides the basic facilities for inputting, modifying and outputting the contents of the simulated control store, tracing, and address halt of the microinstructions.

The fundamental program blocks of the simulator are:

a. Simulation control, inputs the simulator commands and directs their execution.

b. Simulator command execution represents the actual execution of the simulator commands.

c. Microinstruction execution, executes a microinstruction by simulating its effect.

d. Simulation information accumulator and list output.

The relationships of the basic program blocks are illustrated in figure 6-1.

Note: The I/O functions of the computer are not simulated.

6.2 GENERAL FORM OF STATEMENTS

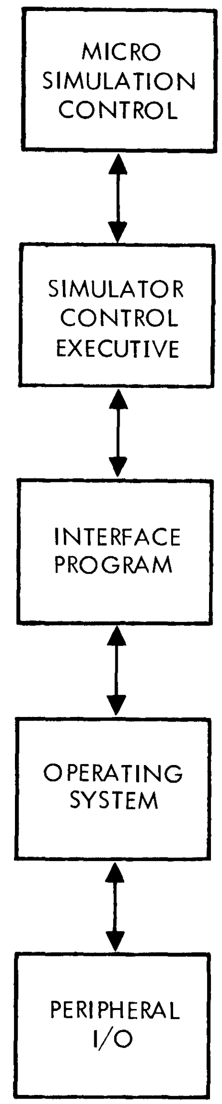
The simulator processes three types of directives. All directives begin with a single letter indicating the type. The following types of actions are handled by the simulator:

a. initialize simulator and storage

b. change and examine storage

c. trace, dump and control execution

Table 6-1 summarizes the directives for quick reference; section 6.7 provides detailed description and examples.



VTII-1810

Figure 6-1. Microsimulator Control Flow

Table 6-1. Summary of Microprogram Simulator Directives

A. Initialize Simulator and Storage

I	Initialize simulator
Pn	Select page n (o through 4)
LC	Load central control store (CCS)
LDA	Load decoder control store (DCS) A

LDB Load decoder control store (DCS) B

MS Select PI as input device

MR Select SI as input device

B. Change and Examine Storage

Ar Alter/Display register r, where r is

- A ALU output
- C Shift counter
- I Instruction register
- K Key register in data loop
- M Memory input register
- O Operand register
- P Program counter
- S Status register

ARn Alter/Display general register n (0 through F hexadecimal)

AJn Alter/Display stack position n (0 through F hexadecimal)

Cm Change/Display main memory word m

ECn Change/Display CCS word n

EDdn Change/Display DCS d (A or B) word n

C. Trace, Dump and Control Execution

D Dump complete CCS

Dm Dump contents of CCS starting at CCS word m

Dm,n Dump contents of CCS from word m to n

D,n Dump from word zero to n

TS Trace set

TR Trace reset

TSn,m Trace from CCS word n to word m

Bn Begin simulated execution at CCS word n

Hn,n Halt at CCS address(es) n

SS Single step set

SR Single step reset

R Return to MOS or VORTEX; Halt in standalone

Two methods of correcting typographical errors are available to the operator. An entire line can be deleted by typing the backslash character (shift/L). The backslash is output as a visual aid. A line feed and a carriage return are output to indicate that the line has been deleted. A character just entered can be deleted by typing the backarrow character. The backarrow character is printed on the Teletype page printer as a visual indicator of the deletion. As many backarrows as necessary can be entered; each deletes one character (but not beyond the beginning of the line).

Each simulator directive is checked for syntax errors as the input is interpreted. When an error is detected by the simulator an error message is output to the Teletype page printer. The simulator then is ready to receive the corrected directive.

The simulator will operate under VDM MOS or VORTEX. For the MOS or stand-alone versions the hardware is described in VDM document number 98 A 9952 09x, VDM 620 Master Operating System. For the VORTEX version the hardware is described in VDM document number 98 A 9952 10x, VORTEX Reference Manual. In addition, the computer must have the arithmetic option, at least 16K (20K for VORTEX) of memory and for two control store pages another 4K of memory is needed. The input/output interface for the MOS and stand-alone versions is described in the document 98 A 9952 09x and VDM document number 89A0023, VDM 620 MOS Input/Output Control System.

The input/output interface for the VORTEX version is described in the above document number 98 A 9952 10R and VDM document number 89A0202, system external Specification for the VORTEX Operating System.

6.3 STATEMENT DEFINITIONS

In the following discussion of simulator dialog, simulator input will be in bold type. This will not appear during actual runs.

All numeric values denoted in the following discussion of the simulator directives are hexadecimal (0-F). Numeric values which are entered on SI are right justified with unspecified leading bit positions containing zeros.

6.3.1 Select Input Media (M)

The select input media directive is used to select the device from which simulator directives will be entered. Normal operation uses the SI device assigned at load time. Using this directive, the PI device assigned at load time can be used as an alternate input device.

The two formats of the directive are:

MS Select PI as input device
MR Select SI as input device

6.3.2 Initialize Simulator (I)

The initialize directive is used to initialize to zero the contents of the simulator registers, the test condition flags, CCS control buffer and the CCS word execution count table. Also, the single step option is reset, the trace option is set and the CCS address halt is set to 200 hex. This directive is normally used at the beginning of each simulation run. The simulator CCS's are not initialized.

6.3.3 Page Select (P)

This directive is used to select the control store page upon which the simulator directive will be executed. Initialization selects page 0. Once a page is selected, all directives will refer to that page until it is change by a new P command or until the system is reinitialized. The format for this command is:

Pn where n = 0, 1, 2, or 3.

6.3.4 Load Control Store (L)

This command is used to read the micro assembler output, assemble the data into usable 64-bit (CCS) words or 16-bit (DCS) words and store the words into the simulator control store.

The format for this command is:

LC -- Load Central Control Store (CCS)
LDA -- Load Decoder A Control Store (DCS)
LDB -- Load Decoder B Control Store (DCS)

The statement **LOAD COMPLETE** will be output to the Teletype following successful loading of the control store.

6.3.5 Alter/Display Simulator Registers (A)

This directive is used to display and change, or display only, the contents of general registers, stack positions and any of the following simulator registers:

Program Counter	(P)
Instruction Register	(I)
Status Register	(S)
Operand Register	(O)
Shift Counter	(C)
Memory Latch	(M)
Processor Key Register	(K)
ALU Output	(A)

a. The format for display or change of the registers above in this directive is:

Ar
mmmm Where c = { nnnn(c/r), nnnn, , , (c/r) }
c

Where **r** is one of the register letters above and **c** is a comma, carriage return, a value followed by a comma or a value. *mmmm* is the contents of that register (output by the simulator) and nnnn is the desired contents. If the command is terminated with a comma (,), the simulator will output the letter A (signifying you are still in this routine) and wait for another register designator. If the directive is terminated with a carriage return (c/r), the simulator returns to the executive. If no change value is input, the contents remain the same.

For the file registers and jump stack, the specific file register or stack position must also be designated upon initial entry.

b. For general-purpose registers

ARi
mmmm
c

Where **i** is a hexadecimal number 0 through F designating the specific register and c is a comma, carriage return, a value or a value followed by a comma.

c. For stack positions

AJn
mmmm
c

Where **n** is a stack position and c is a comma, carriage return, a value or a value followed by a comma.

The rest of the format is identical to that for the other registers except that the comma terminator causes the display of the number and contents of the next sequential file register or stack position. A comma terminator to register or stack position F effects a return to the simulator executive.

Example 1:

AP	Display Program Counter
0776	
	No change, stay in command
A M	Display Memory Latch
14FC	
(c/r)	No change, return

Example 2:

AS	Display Status Word
0000	
FFFF	Change Status to All Ones

Example 3:

ARA	Display General register 10
FFFF	
0000,	Change to all zeros

(continued)

B	Display general register 11
1234	
(c/r)	No change, return

6.3.6 Change/Display Memory (C)

This directive is used to display or display and change a memory location. Both the location and its contents are in hexadecimal notation.

The format of the command is:

```
Cmmmm
  hhhh
c
```

Where **c** is as defined above and **mmmm** is the hexadecimal address of the memory location, hhhh is the contents of that word output by the simulator. If the simulator directive is terminated with a comma, the simulator will display the contents of the next memory location. If the simulator directive is terminated with a carriage return, the change/display memory directive is terminated. If no change value is input, the contents remain the same.

6.3.7 Change/Display CCS Word (EC)

The change/display CCS word simulator directive is used to display and/or change the contents of a CCS word.

The format for the change/display CCS word simulator directive is:

```
ECmmmm                                  nnnnnnnnnnnnnnnn
  hhhhhhhhhhhhhhhh  Where b  =  {       nnnnnnnnnnnnnnnn,
b                                       ,
                                        (c/r)
```

Where mmmm is the (hexadecimal) address of a CCS word, hhhhhhhhhhhhhhhh is the contents of that CCS word (output by the simulator) and nnnnnnnnnnnnnnnn is the desired contents of that CCS word. If the simulator directive is terminated with a comma, the simulator will display the contents of the next CCS word. If the simulator directive is terminated with a carriage return (c/r), the change/display CCS word simulator directive is terminated. If no change value is input, the contents remain the same.

If less than 16 digits are input for a change, the digits are right justified and zeros will appear in the most significant bits not specified.

Example 1

```
EC8A
  0123456789ABCDEF
FEDCBA9876543210
```

Example 2:

```
ECDC
  FFFFFFFFFFFFFFFF
;
  DD
  AAAAAAAAAAAAAAAA
0
```

6.3.8 Change/Display DCS Word (ED)

This directive is used to display and change, or display only, the contents of a DCS A or DCS B word.

The format for the directive is:

```
EDdi                            nnnn
  mmmm     Where c  =  {        nnnn,
c                               ,
                                (c/r)
```

Where **d** is the letter A or B designating DCS A or B, **i** is the DCS address (0-F), mmmm is the contents of the location and nnnn is the desired contents. A comma terminator causes the display of the next sequential address and its contents. A comma terminator to address F effects a return to the simulator executive as does the carriage return terminator. If no change value is input the contents remain the same.

6.3.9 Begin Simulated Execution (B)

The begin-simulated-execution simulator directive is used to start the simulated execution of the CCS microinstructions.

The format for the begin-simulated-execution directive is:

Bmmm

Where **mmm** is the control store memory address for the start of the simulated execution. If no CCS address is given, then the starting address is the CCS address generated as the next CCS address from the last microsimulation. However, if the simulator is initialized in the meantime, the address will be word zero.

Examples:

B0	Begin at word 0 of current page
B7F	
B	Begin from last calculated address

6.3.10 CCS Address Halt (H)

The CCS address halt simulator directive is used to set an address into the simulator such that whenever that CCS address is accessed by the simulator, the simulation process will stop. Since control store addresses are between

0 and 1FF (hexadecimal), specifying an address outside this range effectively "turns off" the address halt. Up to five halt addresses may be set per page. The default value is 200 (CCS word 512).

The format for the CCS address halt simulator directive is:

Hnnn *,nnn,...*

Where *nnn* is the (hexadecimal) halt address.

NOTE: To set multiple halts all addresses must be entered under the same H command.

The halt addresses are set in the page currently selected. To set halt addresses in another page that page must be selected with the "P" command.

Example:

```
H3A9
H100,10A,IFF,0
```

When the halt address is reached, the location and control buffer fields are listed on the line printer if the trace option is ON. Also, the message "CCS HALT" is output to the TTY and line printer. Then the simulator returns to the executive.

6.3.11 Single Microinstruction Step (S)

The single microinstruction step simulator directive is used to set or reset the single step option in the simulator. When the single step option is on, instruction simulation is ceased after the execution of each microinstruction.

The formats for the single microinstruction simulator directive are:

SS Single step ON
SR Single step OFF

The first control store word to be executed must be specified via the begin (B) command. To continue with the next microword enter the B command without an address.

A special form of the SR directive (set single step OFF) can be used to set a limit on the number of microinstructions to be executed before returning to the simulator executive.

The format of this directive is:

SR*nnnn*

Where *nnnn* is 1-4 hex digits specifying the execution limit. When this limit is reached, control is returned to simulator executive. Omission of *nnnn* results in an unlimited run count.

6.3.12 Trace (T)

The trace directive controls output to the line printer. The trace option is normally ON and pertinent data and execution results are listed on the line printer after the simulated execution of each control store instruction.

The format for the directive is:

TS	Set trace ON
TR	Set trace OFF
TS*nnn,mmm*	Set trace ON from word *nnn* to word *mmm*

If *nnn* is missing, its value is defaulted to zero. If *mmm* is missing, its value is defaulted to 200 hex (word 512). If TS is specified with bounds, the current and next CCS addresses are output to LO regardless of whether or not the address is within the bounds; however, the remainder of the trace is suppressed.

The following information is listed on the line printer (LO) for each control store word executed:

1. CCS word address
2. List of CCS word fields and their values
   NOTE: Fields AA, BB, and FF are dynamically altered and need not be equal to the value of the CCS word.
3. Next CCS word
4. Current top of stack
5. Number of items on stack
6. ALU A input
7. ALU B input
8. ALU output
9. Carry in status (CF)
10. Carry out status (ALUC)
11. Contents of the 16 general-purpose registers (R0-RF). (4 per line by 4 lines)
12. Contents of the following registers and flip-flops:

P	Program counter
SC	Shift counter
OPR	Operand register
KREG	Key register processor
IOKR	I/O key register
IBR	Instruction buffer
I	Instruction register
STAT	Status register
IOR	I/O data register
SHFT	Sign store of register A bit 15
QUOS	Storage of sign bit (DAL 15) of ALU output

13. Memory Operations Data
    The values listed are the values at the end of the memory operations for that CCS word. The memory

operations performed are a function of conditions/codes upon entry (values from the last CCS word executed).
When MCCO = 2 the following memory operations data will appear twice per microword trace. The first set is an intermediate value while the second set represents the values at the end of the memory operation.

Memory Condition Code

MCCO = 0	Idle
MCCO = 1	Active but not done
MCCO = 2	Active and done

Memory Operation Code

MOPC = 0	Transfer ALU output to MIL and IBR
MOPC = 1	Read from main memory to MIL and IBR
MOPC = 2	Read from main memory to MIL
MOPC = 3	Write 16-bit ALU output to main memory
MOPC = 4	Write a byte of ALU output to main memory (byte is specified by MBYC)

Main Memory Address Source

MADS = 0	Address is ALU output
MADS = 1	Address is program counter
MADS = 2	Address is memory input register (MIR)
MADS = X	Invalid address source

Byte Designator for Write Operations

MBYC = 1	Right byte
MBYC = 0	Left byte

NOTE: The byte (of the memory word) not designated is not altered.

Memory Interface Registers

The contents of registers MIL and IBR are listed.

Main Memory Address (MMAD)

The main memory address (as specified by MADS) is listed. It is listed for every CCS word executed regardless of the actual memory operation as specified by MCCO and MOPC.

Status of test conditions (test inputs). Each status bit stored in a separate word of memory and the 16-bit word is listed (XXXX). The 16 test conditions are listed on 2 lines, 8 per line. Each test bit is listed as 0000 = false condition; or 0001 = true condition.

Test Bits

0	ALU overflow
1	I/O sense
2	SSW3
3	SSW2
4	SSW1
5	620/f test (for JMP, JMPM, XEC groups of instructions)
6	ALU equals
7	ALU sign
8	ALU carry
9	ALU zero
10	Shift flag
11	MIL 15 (sign bit of memory input register)
12	Shift count = -1
13	A15 – sign of A register for multiply operations
14	DAL 15/DAL 14 (ALU output bits 15 and 14)
15	QS bit

6.3.13 Dump Contents of CCS (D)

The dump CCS directive is used to list on the line printer selected contents of the simulator control CCS and the count of the number of times each word was executed.

The formats for the directive are:

```
Dmmm,nnn
Dmmm
D,nnn
D
```

Where mmm and nnn are the beginning and ending hexadecimal CCS address to dump. If mmm is omitted, dump begins at CCS word 0. If nnn is omitted, the complete contents of the simulated CCS table is dumped starting at mmm. If both m and n are omitted, the complete simulated CCS table, starting at location zero is dumped.

The line printer list format is:

ADDR	HEXADECIMAL		BINARY		EXECUTED
aaaa	hhhhhhhh	hhhhhhhh	bbbbbbbbbbbbbbbb	bbbbbbbbbbbbbbbb	xxxx
			bbbbbbbbbbbbbbbb	bbbbbbbbbbbbbbbb	
aaaa	hhhhhhhh	hhhhhhhh	bbbbbbbbbbbbbbbb	bbbbbbbbbbbbbbbb	
			bbbbbbbbbbbbbbbb	bbbbbbbbbbbbbbbb	

Where (aaaa) is the address of the CCS word in hexadecimal, (hhhhhhhh hhhhhhhh) is the contents of the CCS word in hexadecimal, (bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb) is the contents of the CCS word in binary and xxxx is the execution count in hexadecimal.

The field identifier words and the contents and count of up to 14 locations are listed on each page.

6.3.14 Exit to MOS or VORTEX (R)

The exit to MOS or VORTEX simulator directive is used to effect a transfer of control from the simulator to MOS or VORTEX. NOTE: The use of this directive with the stand-alone version produces a halt.

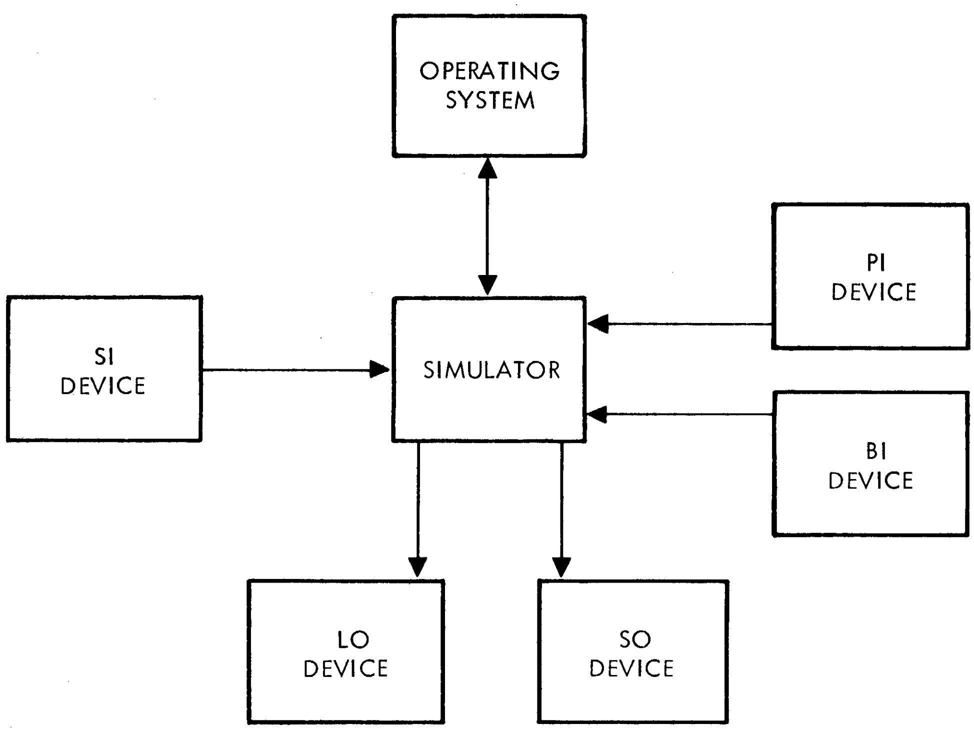
6.4 OPERATING INSTRUCTIONS

The simulator program operates under either MOS, VORTEX, or stand-alone environments. The simulator executive communicates with the software environment in which it is running by means of the appropriate interface program, INTR, provided with the simulator. The user communicates to the program via the system Teletype. The BLD II loader is required when loading of MIDAS object programs for execution under the simulator (MOS or stand-alone only).

When operating under VORTEX, the five background global control blocks (FCB's) are used when the logical unit is an RMD thus permitting the stacking of jobs. The following restraints are made on the use of RMD logical units:

1. SI, PI, and LO are to be in unblocked format.
2. BI must be blocked.

The simulator data flow is shown in figure 6-2.



VTII-1809

Figure 6-2. Microsimulator Data Flow

6.4.1 Program Loading

Under VORTEX, MICSIM can be scheduled from the background library at level zero by the /LOAD,MICSIM directive. Before scheduling, the number of WCS pages in addition to page zero which will be needed should be determined and a /MEM,X directive given. In the /MEM directive, X should be the number of additional WCS pages (beyond page zero) times 4.

Under MOS, each time the simulator is to be executed its relocatable binary object deck should be positioned on the BI device and the /LOAD directive given.

In the stand-alone environment, MICSIM is loaded by the 620 stand-alone FORTRAN IV loader, along with the runtime I/O and runtime utility. (Refer to VDM document numbr 89A0226, Overview and External Specification for information on the Varian 620 stand-alone FORTRAN IV loader.) The simulator uses logical unit numbers 2, 3, 4, 5, and 6 for SI, SO, PI, LO, and BI. The stand-alone loader should be instructed to assign these units to meaningful devices.

Examples:

Sample Loading Procedures

1. VORTEX
   ```
   /JOB,SIM
   /LMGEN
    TIDB,SIM,1,0
    LD,6
   ```
   Test Program (optional)
   Simulator
   EOF (2-7-8-9 multi-punch)
   ```
   LIB
    END,BL,E
    /MEM,x
    /LOAD,SIM
   ```

   x value = 0, only 1 WCS page; = 4, 2 WCS pages; = 8, 3 WCS pages; = 12, 4 WCS pages.

2. MOS
   ```
    /JOB,SIM
   /LOAD
   ```
   Test Program (optional)
   Simulator
   EOF (2-7-8-9 multi-punch)

3. STAND-ALONE
   Load stand-alone loader
   With AID II, change absolute location 7 ($PED) to the desired start load address
   Return to the loader
   Enter the following:
   `200300402504602 (c/r)`
   (to set SI = TY, SO = TY, PI = PT, LO = −77, BI = PT)
   Mount simulator tape in reader
   Enter the following:
   `PM`
   Load Runtime I/O
   Load Runtime Utility

6.4.2 Initial Condition Selection

After loading, the simulator program is automatically entered and outputs the following to SO:

```
VARIAN 73 MICROSIMULATOR
INPUT HIGHEST NUMBER WCS PAGE DESIRED
```

The user then inputs on SI one of the following:

0	(for ROM page only)
1	(for ROM and WCS page 1)
2	(for ROM and WCS pages 1 and 2)
3	(for ROM and WCS pages 1, 2, and 3)

Any other input is an error and the request will be repeated. Following a correct input, the following is output to SO:

```
SI**
```

An SI** indicates that the program is in the simulator executive awaiting a user command. Control is returned to the executive following execution of each command.

All simulator dialog is entered through the SI device and echoed on the SO and LO devices. Dialog may be either conversational or batch depending on the SI device assignment. All of the simulator directives must be terminated with a carriage return; the simulator will output a line feed.

6.4.3 Loading Simulator Central Control Store (CCS) and Decoder Control Store (DCS)

Use the P directive to select the WCS page in which simulation is to take place.

Use the L directive to load the micro assembler output into the specified simulator control store (central or decoder).

Use the M directive to select the input device; either SI or PI.

Use I directive to initialize to zero all the simulator registers, test conditions, control store buffer, status registers and execution count table.

Use the A directive to initialize the program counter, file registers, and instruction register as required.

Position the 620/70 sense switches as required. The simulator program monitors the 620/70 sense switches similar to the Varian computer sensing of its control-panel sense switches.

6.4.4 Other Control (As Required)

Use the E directives to make any patch corrections to the CCS or DCS.

Use H directives to set simulation halts when the specified control store address is reached. The initialized address is 200 hex. and will remain such until specified otherwise.

Use S directives to specify single step operation as required. The initialized condition is run (not step).

Use T directives to specify operation with or without trace listing as required. The initialized condition is with trace.

6.5 PROGRAM EXECUTION

After all initialization and start-up conditions are specified, use the B directive to begin execution at the specified control store address.

6.6 AFTER SIMULATION

6.6.1 Control Store Dump

Use the D directive to dump the control store words and the execution counts for each control store.

6.6.2 Initialization

Use I directive to initialize registers, tables, etc. prior to making another run.

6.6.3 Return to MOS, VORTEX

Use the R directive to return to MOS or VORTEX as required. (NOTE: In the stand-alone version this command effects a halt).

6.7 620 EMULATION

To run programs using the 620/f emulation ROM, the following sequence of events must be done:

1. Load CCS page 0 and DCS page 0 with the 620/f emulation microinstructions.
2. Set CCS halt to 080 (hex) via H command.
3. Set R5 to FFFF (– 1) via AR5 command.
4. Set other registers and sense switches as needed.
5. Set pseudo P register to location (hex) of first macro to be executed via AP command.
6. Set trace and step/run mode as needed.
7. Begin at 13E via B command.

The sequence of events 1 through 6 may be in any order but must be done before event 7. Event 7 begins simulation at standard state 1.

6.8 ADDING SIMULATOR TO VORTEX

The microsimulator resides on the background library under VORTEX. After system generation, however, the user is responsible for cataloging it into the background library. The following procedure may be used to do this. First, position the BI device to the simulator object material. Then, issue the following directives:

```
ILMGEN
TIDB,MICSIM,1,0
LD,BI
LIB
END,BL,E
```

(For detailed descriptions of these directives, refer to the VORTEX Reference Manual.)

6.9 MAIN MEMORY SIMULATION

Simulation of main memory operations is restricted so that a simulation run does not destroy the simulator or related programs. This is accomplished by not simulating writes to memory addresses below 1000 octal or above the start of the simulator. Any attempt to do this will be flagged as an error and the write not be performed; simulation will continue however. A read may be made anywhere in available memory. Memory addressing above 32K will effect **wraparound** if available on the computer.

Creation of a Main Memory Block

VORTEX:

Since VORTEX does not allow a start load address (it is always 1000 octal) for background tasks, the user must create a load module with an empty block at the beginning of the module. A possible way to do this is to set up an object stream as below:

Macro Test Program
BSS Block
DATA 0
Simulator
EOF

Using the BSS block effectively moves the simulator higher in core and thus leaves the memory between 1000 (octal) and the start of the simulator available for main memory. The size of the BSS block depends on the amount of memory available for background and the needs of the user. Too large of a BSS block will cause the load module to abort loading.

MOS:

The same method can be used for MOS as was used for VORTEX or at load time. The start load address may be set to some value larger than the default value (500 octal). For example, to get a main memory block of 1024 words, the load directive might be /L,PR=2500.

6.10 SIMULATOR ERROR MESSAGES

MESSAGE **REASON**

General

MS01 Input could not be interpreted as a valid command.

MS02 A non-hex character was encountered when hex expected.

Initialization

MS03 Insufficient common area to contain specified number of pages.

MS04 The selected page number was not valid.

CS Addressing

MS05 An attempt was made to jump to an unavailable WCS page.

MS06 A BCS instruction was encountered when WCS page 1 is unavailable.

CS Loading

MS07 Read error on BI device.

MS08 EOF encountered before load complete.

MS09 EOD/BEOD encountered before load complete.

MS10 Sequence error on BI.

MS11 Invalid loader code.

MS12 Checksum error.

Memory

MS13 Undefined macro opcode.

MS14 Attempted to write to memory outside defined main memory.

6.11 EXAMPLE OF SIMULATOR OUTPUT

Figure 6-3 shows the simulation listing of the LDA example developed in section 2.

```
PAGE  0000  09/07/73                  VORTEX   MICSIM

VARIAN 73 MICRO SIMULATOR
INPUT HIGHEST NUMBER WCS PAGE DESIRED
0
MS**
P0                       SELECT PAGE ZERO
MS**
LC                       LOAD CENTRAL CONTROL STORE, 620 EMULATION
LOAD COMPLETE
MS**
LDA                      LOAD DECODER A, 620 EMULATION
LOAD COMPLETE
MS**
LDB                      LOAD DECODER B, 620 EMULATION
LOAD COMPLETE
MS**
C400                     PUT AN 'LDA' INSTRUCTION IN MEMORY FOR SIMULATION
0000
10F9                     LDA FROM MEM LOC 'F9'
MS**
CF9                      CHECK WHATS TO BE LOADED
0036

MS**
AP                       SET PROGRAM COUNTER TO THE 'LDA'
0000
400
MS**
SR7                      EXECUTE SEVEN MICRO'S
MS**
B13E                     START EXECUTION AT STANDARD STATE ONE, SS1M
```

Figure 6-3. Simulator Output Format

```
PAGE   0001   09/07/73                 VORTEX   MICSIM

CCS LOC    013E    PAGE  0

   TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
   00   09   02   00   00   00   01   00   00   00   08   00   00

   RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
   00   00   00   00   00   00   00   00   00   00   00   00

NEXT CCS ADDRESS   0092    PAGE  0

CURRENT TOP OF STACK   0000
NUMBER OF ITEMS ON STACK    0

   ALU INPUT A 0000
   ALU INPUT B 0000

   ALU OUTPUT   0000

   CIN   0
   COUT  0

   R0   0000   R1   0000   R2   0000   R3   0000
   R4   0000   R5   0000   R6   0000   R7   0000
   R8   0000   R9   0000   RA   0000   RB   0000
   RC   0000   RD   0000   RE   0000   RF   0000

P      SC     OPR    KREG   IOKR   IBR    I      STAT   IOR    SHFT   QUOS
0400   0000   0000   0000   0000   0000   0000   0000   0000   0000   0000

MCCO 1
MOPC 1
MADS 1
MBYC 0
MIR    0000
IBR    0000
MMAD   0400

                  TEST CONDITION STATES
UVFL   SENS   SSW3   SSW2   SSW1   EMUL   ALUO   ALUS
0000   0000   0000   0000   0000   0000   0000   0000

ALUC   ALUZ   SHFT   MIRS   SFTC   ROAO   NORM   QUOS
0000   0000   0000   0000   0000   0000   0000   0000
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0002  09/07/73              VORTEX   MICSIM

CCS LOC    0092   PAGE  0

   TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
   00   02   0D   00   00   00   01   00   00   00   08   00   00

   RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
   04   00   00   00   00   00   00   00   00   00   00   00

NEXT CCS ADDRESS  002D   PAGE  0

CURRENT TOP OF STACK  0000
NUMBER OF ITEMS ON STACK   0

   ALU INPUT A 0000
   ALU INPUT B 0000

   ALU OUTPUT  0000

   CIN  0
   COUT 0

   R0   0000  R1   0000  R2   0000  R3   0000
   R4   0000  R5   0000  R6   0000  R7   0000
   R8   0000  R9   0000  RA   0000  RB   0000
   RC   0000  RD   0000  RE   0000  RF   0000

P     SC    OPR   KREG  IOKR  IBR   I     STAT  IOR   SHFT  QUOS
0401  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000

MCCD 2
MOPC 1
MADS 1
MBYC 0
MIR   0000
IBR   0000
MMAD  0400
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0003  09/07/73                VORTEX   MICSIM


MCCO 1
MOPC 1
MADS 1
MBYC 0
MIR   10F9
IBR   10F9
MMAD  0401

                TEST CONDITION STATES
OVFL  SENS  SSW3  SSW2  SSW1  EMUL  ALUO  ALUS
0000  0000  0000  0000  0000  0000  0000  0000

ALUC  ALUZ  SHFT  MIRS  SFTC  ROAO  NORM  QUOS
0000  0000  0000  0000  0000  0000  0000  0000
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0004  09/07/73                    VORTEX   MICSIM

CCS LOC    002D    PAGE  0

  TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
  0E   0D   06   00   00   00   00   05   00   00   06   00   00

  RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
  00   00   00   00   00   00   00   00   00   00   00   00

NEXT CCS ADDRESS  0182   PAGE  0

CURRENT TOP OF STACK  0000
NUMBER OF ITEMS ON STACK   0

  ALU INPUT A 0000
  ALU INPUT B 0000

  ALU OUTPUT  0000

  CIN  0
  COUT 0

  R0   0000  R1   0000  R2   0000  R3   0000
  R4   0000  R5   0000  R6   0000  R7   0000
  R8   0000  R9   0000  RA   0000  RB   0000
  RC   0000  RD   0000  RE   0000  RF   0000

P     SC    OPR   KREG  IOKR  IBR   I     STAT  IOR   SHFT  QUOS
0401  0000  0000  0000  0000  10F9  10F9  0000  0000  0000  0000

MCCO 2
MOPC 1
MADS 1
MBYC 0
MIR   10F9
IBR   10F9
MMAD  0401
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0005  09/07/73                VORTEX   MICSIM


MCCO 0
MOPC 1
MADS 1
MBYC 0
MIR   0000
IBR   0000
MMAD  0401

                TEST CONDITION STATES
OVFL  SENS  SSW3  SSW2  SSW1  EMUL  ALUO  ALUS
0000  0000  0000  0000  0000  0000  0000  0000

ALUC  ALUZ  SHFT  MIRS  SFTC  ROAD  NORM  QUOS
0000  0000  0000  0000  0000  0000  0000  0000
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE   0006   09/07/73                 VORTEX   MICSIM

CCS LOC    0182    PAGE   0

   TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
   00   12   0F   00   00   00   01   00   00   00   05   02   00

   RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
   03   0A   01   03   01   01   00   00   00   00   00   00

NEXT CCS ADDRESS  012F    PAGE   0

CURRENT TOP OF STACK  0000
NUMBER OF ITEMS ON STACK    0

   ALU INPUT A 0000
   ALU INPUT B 00F9

   ALU OUTPUT   00F9

   CIN   0
   COUT  0

   R0   0000   R1   0000   R2   0000   R3   0000
   R4   0000   R5   0000   R6   0000   R7   0000
   R8   0000   R9   0000   RA   0000   RB   0000
   RC   0000   RD   0000   RE   0000   RF   0000

P      SC     OPR    KREG   IOKR   IBR    I      STAT   IOR    SHFT   QUOS
0401   0000   00F9   0000   0000   0000   10F9   0000   0000   0000   0000

MCCO 1
MOPC 2
MADS 0
MBYC 0
MIR    0000
IBR    0000
MMAD   00F9

              TEST CONDITION STATES
OVFL   SENS   SSW3   SSW2   SSW1   EMUL   ALUO   ALUS
0000   0000   0000   0000   0000   0000   0000   0000

ALUC   ALUZ   SHFT   MIRS   SFTC   ROAD   NORM   QUOS
0000   0000   0000   0000   0000   0000   0000   0000
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0007  09/07/73                VORTEX   MICSIM

CCS LOC    012F    PAGE  0

   TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
   00   1E   0C   01   0F   00   00   00   00   00   00   00   00

   RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
   00   00   00   00   00   00   00   00   00   00   00   00

NEXT CCS ADDRESS   01E0    PAGE  0

CURRENT TOP OF STACK  0000
NUMBER OF ITEMS ON STACK   0

   ALU INPUT A 0000
   ALU INPUT B 0000

   ALU OUTPUT  0000

   CIN  0
   COUT 0

   R0   0000  R1   0000  R2   0000  R3   0000
   R4   0000  R5   0000  R6   0000  R7   0000
   R8   0000  R9   0000  RA   0000  RB   0000
   RC   0000  RD   0000  RE   0000  RF   0000

P     SC    OPR   KREG  IOKR  IBR   I     STAT  IOR   SHFT  QUOS
0401  0000  00F9  0000  0000  0000  10F9  0000  0000  0000  0000

MCCO 2
MOPC 2
MADS 0
MBYC 0
MIR   0000
IBR   0000
MMAD  00F9
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0008  09/07/73                VORTEX   MICSIM

MCCO 0
MOPC 2
MADS 0
MBYC 0
MIR   0036
IBR   0000
MMAD  00F9

                  TEST CONDITION STATES
OVFL  SENS  SSW3  SSW2  SSW1  EMUL  ALUO  ALUS
0000  0000  0000  0000  0000  0000  0000  0000

ALUC  ALUZ  SHFT  MIRS  SFTC  ROAO  NORM  QUOS
0000  0000  0000  0000  0000  0000  0000  0000
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0009  09/07/73                VORTEX   MICSIM

CCS LOC   01E0   PAGE  0

  TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
  00   0B   05   00   00   00   01   00   00   00   08   00   00

  RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
  04   00   00   00   00   00   00   00   00   00   00   00

NEXT CCS ADDRESS  00B5   PAGE  0

CURRENT TOP OF STACK  0000
NUMBER OF ITEMS ON STACK   0

  ALU INPUT A 0000
  ALU INPUT B 0000

  ALU OUTPUT  0000

  CIN  0
  COUT 0

  R0   0000   R1   0000   R2   0000   R3   0000
  R4   0000   R5   0000   R6   0000   R7   0000
  R8   0000   R9   0000   RA   0000   RB   0000
  RC   0000   RD   0000   RE   0000   RF   0000

P      SC     OPR    KREG   IOKR   IBR    I      STAT   IOR    SHFT   QUOS
0402   0000   00F9   0000   0000   0000   10F9   0000   0000   0000   0000

MCCO 1
MOPC 1
MADS 1
MBYC 0
MIR   0036
IBR   0000
MMAD  0402

                  TEST CONDITION STATES
UVFL   SENS   SSW3   SSW2   SSW1   EMUL   ALUO   ALUS
0000   0000   0000   0000   0000   0000   0000   0000

ALUC   ALUZ   SHFT   MIRS   SFTC   ROAD   NORM   QUOS
0000   0000   0000   0000   0000   0000   0000   0000
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0010  09/07/73                VORTEX   MICSIM

CCS LOC    0085    PAGE  0

   TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
   0F   0D   06   00   00   00   00   05   00   00   06   01   00

   RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
   00   0A   01   00   01   00   00   00   00   00   01   00

NEXT CCS ADDRESS  0080   PAGE  0

CURRENT TOP OF STACK  0000
NUMBER OF ITEMS ON STACK   0

   ALU INPUT A 0000
   ALU INPUT B 0036

   ALU OUTPUT  0036

   CIN  0
   COUT 0

   R0   0036   R1   0000   R2   0000   R3   0000
   R4   0000   R5   0000   R6   0000   R7   0000
   R8   0000   R9   0000   RA   0000   RB   0000
   RC   0000   RD   0000   RE   0000   RF   0000

P      SC     OPR    KREG   IOKR   IBR    I      STAT   IOR    SHFT   QUOS
0402   0000   00F9   0000   0000   0000   0000   0000   0000   0000   0000

MCCO 2
MOPC 1
MADS 1
MBYC 0
MIR   0036
IBR   0000
MMAD  0402
```

Figure 6-3. Simulator Output Format *(continued)*

```
PAGE  0011  09/07/73                 VORTEX   MICSIM


MCCO 0
MOPC 1
MADS 1
MBYC 0
MIR   0000
IBR   0000
MMAD  0402

                TEST CONDITION STATES
OVFL  SENS  SSW3  SSW2  SSW1  EMUL  ALUD  ALUS
0000  0000  0000  0000  0000  0000  0000  0000

ALUC  ALUZ  SHFT  MIRS  SFTC  ROAD  NORM  QUOS
0000  0000  0000  0000  0000  0000  0000  0000
EXECUTION LIMIT SATISFIED
MS**
R
```

Figure 6-3. Simulator Output Format *(continued)*

SECTION 7

MICROPROGRAM UTILITY PROGRAM, MIUTIL

The microprogram utility (MIUTIL) loads information into WCS and provides an interface with hardware features of the WCS.

Two sets of directives are provided. The basic set will allow the user to load the WCS with microassembler output, examine single WCS words and list WCS contents. The second group of directives gives the user access to the debugging features of the control store. With these directives single microstep execution can be done.

The utility operates in three environments, under the VORTEX operating system, MOS operating system and as a stand-alone program. A standard interface program provides compatibility.

7.1 BASIC ELEMENTS

The microprogram utility accepts directives as similar as possible to those of the microprogram simulator.

7.2 GENERAL FORM OF DIRECTIVE

In general a utility directives consists of a unique first character, followed by a string of parameters, terminated by a carriage return. The following sections describe the meaning of each of these first characters and permissible parameters. Table 7-1 summarizes the utility directives.

The following are the utility directives available to the user:

Table 7-1. Summary of Utility Directives

A. Basic Command Set

Pn	Page select
LC	Load central control store (CCS)
LDA	Load decoder control store (DCS) A
LDB	Load decoder control store (DCS) B
MS	Media set, selects PI for input
MR	Media reset, selects SI for input
Exm	Examine/change control store x word m
Dxm,n	Dump control store x word m through n
R	Return the operating system or exit from utility in stand-alone environment

B. Debugging Directives

Nx	Enables control store x
TS	Trace set
TR	Trace reset
Gn	Set microprogram execution address to CCS word n

(continued)

Xn	Execute n microinstructions
I	Initialize WCS
Bn	Branch to CCS word n
Hn	Halt execution at word n

7.3 DIRECTIVE DEFINITIONS

In the following discussion of utility directives, the characters the user inputs are in bold-face type and explanation of the action in regular type.

All numeric values are hexadecimal.

7.3.1 Select Page (P)

This directive selects a particular WCS page for the commands which follow. The directives for loading, and dumping do not accept a page number and thus rely on the previous P command for page selection.

Before the first P command is given by the user, a default page value of 1 is assumed.

The letter P is followed by a hexadecimal digit for the page number. For example **P3** would select page 3.

7.3.2 Load Control Store (L)

This directive loads microassembler output into the writable control store. The user specifies which page is to be loaded by the prior P command. The user specifies which control store should be loaded by the one parameter following the letter L. C indicates central control store, DA or DB for decode control store A or B, and I for I/O control store.

For example, after P2 a directive **LC** would load page two of the central writable control store.

7.3.3 Examine/Change Control Store (E)

Through this directive a single word of WCS may be either examined or changed. The user specifies which control store and the word number. The page is obtained through the previous P directive.

The form of the E directive is **Exmmm** where **x** is either C, DA, DB or I for central, decoder, and I/O control stores respectively, and **mmm** is the address of the control store word in hexadecimal notation.

The utility will type out the contents of the location followed by a carriage return. The user must then do one of the following:

1. Change the contents of the location by typing a new hexadecimal value followed by a carriage return
2. Change the contents of the location and then examine the next location by typing a new hexadecimal value, followed by a comma, followed by a carriage return
3. Examine the next location by typing a comma followed by a carriage return
4. Type a carriage return

For example

	Action Caused
MU**	
P1	Selects page 1
MU**	
EI29	Examine I/O control store location 29
12A3	Computer types contents
0,	User changes contents to zero
002A	
1233	Computer types location 2A
0	User changes its contents to zero
MU**	
ECF	Utility accepts another directive

7.3.4 Dump Control Store (D)

The dump directive provides a listing of the control store contents. The page is determined by the prior P directive. The user specifies the locations and control store type in the parameters.

The general format for the dump command is:

Dx*mmm,nnn*

where **x** is C, DA, DB or I for the specific control store (as above), *mmm* is the hexadecimal location where the dump is to start, and *nnn* is last location to be dumped. If the final location is missing, the last location of the page is assumed. If the first address is omitted, it is assumed to be zero.

Dump directive example:

MU**	
P2	
MU**	
DC	Provides listing of central control store page 2
MU**	
DI30,5A	Provides listing of the I/O control store, locations 30 through 5A
MU**	
DI,5A	List from location zero through 5A
MU**	

Section 7.8 shows a sample printout of the microprogram utility directive D.

7.3.5 Return to Operating System (R)

This directive causes exit from the utility. If running under MOS or VORTEX, control is returned to the operating system. If the utility is running in a stand-alone environment, the R directive causes a halt. There are no parameters, merely the letter R.

7.3.6 Media Set and Reset (M)

This directive allows the selection of an alternate device for input of utility directives. 'MS' selects the 'PI' unit for input. 'MR' returns the utility to the SI unit for input.

Note that receiving an illegal command will cause the media to be automatically reset to SI.

The following directives are designed to operate in the special hardware configuration described in section 7.5.

7.3.7 Enable Control Store (N)

This directive allows the user to enable the specified control stores. The page number used in the one specified by the last P directive.

The general form of the N directive is:

Nx

where **x** is D or I, which specifies enabling of the decoder or I/O control store, respectively.

For example:

MU**	
P1	
MU**	
ND	Enables decoder control store, WCS page 1
MU**	

7.3.8 Trace Execution (T)

The purpose of this directive is to provide the user with a means of following micro execution while it is in progress. To accomplish this, the address of each microinstruction is typed before it is executed.

The general form of the T directive is:

Ta

where **a** is one of the following: S for setting or enabling trace mode, or R for resetting or disabling trace mode.

Before the first T directive is given, the trace mode is reset, i.e., turned off.

The general form of the trace output is:

p-nnn

where **p** is the page number and **nnn** is the word number of the next instruction to be executed.

7.3.9 Set Micro Execution Address (G)

This directive allows the user to choose a location for starting microprogram execution.

This routine will do the following:

1. Step the WCS to stop any execution that might be in progress.
2. Load the micro address register with the specified address.
3. Step the WCS to load the first microword into the control buffer.
4. If trace mode, the next control store address to be executed will be read from the WCS and output to the user.

This directive does not begin execution. It serves only as the setup for an X directive.

The format of the G directive is as follows:

Gn

where **n** is from one to three hex digits specifying a word number in central control store.

The page is obtained from the last P directive.

7.3.10 Execute Microinstruction (X)

This directive is used after the G directive to begin actual micro execution. It can be used to specify free-running execution or execution of a fixed number of micro's followed by a halt. By requesting execution of a single micro, followed by a halt, it can be used to stop free-running execution.

If free-running execution without trace is requested, the fine clock will simply be enabled to run free. There are two ways of interrupting this. An X directive specifying execution of one microinstruction will step the WCS. It can then be restarted by another X directive. The G directive will also stop free-running execution. It sets a starting address, however, and thus it should not be used if the interrupted execution is to be restarted where it left off.

If free-running execution is requested in trace mode, then the WCS is simply single stepped an indefinite number of times. This allows reading of the WCS address before each single step.

If execution of a fixed number of microinstructions is requested, the WCS will simply be stepped the appropriate number of times. If trace mode, then the address will be accessed from the WCS and returned to the user before each micro is executed.

The following is the format of the X directive:

X*n*

Where *n* is zero for free-running execution or non-zero to request execution of *n* microinstructions.

The default value for **n** is 1.

For example:

MU**	
X7	Execute seven microinstructions
MU**	
X0	Enable free-running execution
MU**	
X	Execute one microinstruction (note: this would halt the previous free run)
MU**	

7.3.11 Initialize WCS (I)

The purpose of this directive is to execute an EXC 07X command. This will deselect all WCS control stores, terminate any DMA operations in progress and enable free run of the fine clock. The result is that control will return to the ROM with all WCS activity suspended.

This command should only be used when a meaningful ROM location will receive control. Thus, it should not be used for such things as halting a free-running microprogram.

7.3.12 Branch to CCS (B)

This directive simply executes an I/O branch to the specified address in central control store. Such a branch causes free run execution to begin at that location. The B command thus produces a similar effect to a Gn, X0 directive sequence. The B directive never steps the WCS though, and thus cannot respond to the trace flag.

The general form of the B directive is:

Bn

Where **n** is from one to three hex digits specifying a word number in central control store.

The page number is obtained from the last P directive.

7.3.13 Set Halt Address (H)

This directive may be used with the X directive to single-step microprogram execution to a certain address in WCS.

The format of the H directive is:

Hn

where n is from one to three hexadecimal digits specifying a word in control store. The page number is specified in the last P directive.

Single stepping as a result of an X directive will be terminated when the specified location is the next one to be executed. A message in the trace format will be output to signal this.

The halt can be removed by entering H0. Only one halt address may be set at a time.

7.4 OPERATING INSTRUCTIONS

7.4.1 Program Loading

Under VORTEX, load VORTEX as described in the VORTEX Reference Manual, 98 A 9952 10x. The utility should be in the foreground library. It can be put there at system generation time or added later using the load module generator.

To load the utility and begin execution, an OPCOM schedule directive is necessary. For example:

```
;SCHED,MIUTIL,3,FL,F
```

schedules the utility at priority three.

Under MOS, load MOS as described in the MOS Reference Manual, 98 A 9952 09x. Then, the MOS loader may be used to load the utility program. Execution will begin on successful completion of the load.

For example:

/JOB, UTIL
/LOAD
Utility program binary object
EOF (2-7-8-9 multi-punch)

In a stand-alone environment, load the Varian 620 stand-alone FORTRAN IV system loader as described in VDM document number 89A0226. Instruct the loader to change its logical unit numbers by entering appropriate values. Next, load the utility binary object, followed by the FORTRAN IV stand-alone system runtime I/O tape, followed by the runtime utility tape. On completion of loading, the machine will go into step. Press RUN to start execution.

7.4.2 Program Execution

After successful loading, the utility program is entered automatically. The program will first type VARIAN 73 MICRO UTILITY to identify itself. Next, the configuration will be determined by the following request:

```
DEBUG CONFIG?  (Y or N)
```

The user should then type Y followed by a carriage return, if his system is in the special two-processor debugging configuration described in section 7.5. Otherwise, if his system is simply the standard configuration, the user should type N, followed by a carriage return.

The micro utility will then type

```
EVEN WCS DEV ADDR?
```

The user should then type either 70, 72, or 74, depending on the hardware configuration, followed by a carriage return.

The utility will then type:

```
MU**
```

to indicate that it is ready to accept a directive. Whenever an illegal directive is given, an error message is typed. Description of the various messages can be found in section 7.7. Note that a directive may be in error either due to bad syntax or due to context. An example of the latter case is giving a debugging directive in a non-debugging configuration.

During execution of the D and X directives, SENSE switch 3 may be set to terminate their execution prematurely.

SENSE switch 1 may be set during tracing to suppress listing of page zero addresses.

7.5 DEBUGGING CONFIGURATION

The additional debugging directives of the utility cannot operate on the WCS of the processor on which the utility itself is running. For this reason, a special hardware configuration is needed to use these directives.

The special configuration must have two computer systems: one with a WCS and the other actually operating the utility.

The system which runs the utility program must have the hardware appropriate for the type of operating system or for stand-alone operations. The processor need not have any WCS and the processor itself can be either a 70-series, 620/f, or 620/L. Operating system requirements prevail, since VORTEX does not run on a 620/L.

The Writable Control Store Reference Manual (Varian document number 98 A 9906 08x) describes the physical properties of this two-processor system for debugging.

7.6 ADDING UTILITY TO VORTEX

The microutility resides on the foreground library under VORTEX. After system generation, however, the user is responsible for cataloging it there. The following procedure may be used to do this. First, position the BI device to the microutility object material. Then, issue the following directives:

```
/LMGEN
 TIDB,MIUTIL,2,0
 LD,BI
 LIB
 END,FL,F
```

(For detailed descriptions of these directives, refer to the VORTEX Reference Manual.)

7.7 UTILITY ERROR MESSAGES

Message **Reason**

General

MU01 Input could not be interpreted as a valid command.

MU02 A non-hex character was encountered when hex expected.

Message **Reason**

MU03 EOF detected on SI. Return mode to operating system.

MU04 The selected page number was not valid.

WCS Access

MU05 Unable to access WCS: WCS is busy.

MU06 Unable to access WCS: BIC load in progress.

CS Loading

MU07 Read error on BI device.

MU08 EOF encountered before load complete.

MU09 EOD/BOD encountered before load complete.

MU10 Sequence error on BI.

MU11 Invalid loader code.

MU12 Checksum error.

7.8 EXAMPLES

The following is a sample of microutility output:

```
PAGE   0000   09/07/73                       VORTEX    MIUTIL

VARIAN 73 MICRO UTILITY

DEBUG CONFIG ? (Y OR N)
N

EVEN WCS DEV ADDR ?
/2
MU**
EC25
0000000000000000
,
0026
0000000000000000
,
0027
0000000000000000
8A,
0028
0000000000000000

MU**
DDA8,B
```

```
PAGE   0001   09/07/73                       VORTEX    MIUTIL

DCS A , PAGE  01

0008  0000  0000  0000  0000
MU**
DDB
```

```
PAGE   0002   09/07/73                       VORTEX    MIUTIL

DCS B , PAGE  01

0000  0000  0000  0000  0000  0000  0000  0000  0000
0008  0000  0000  0000  0000  0000  0000  0000  0000
MU**
DC5,7
```

```
PAGE  0003  09/07/73                    VORTEX   MIUTIL

CCS LOC    0005    PAGE 01

   TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
   00   00   00   00   00   00   00   00   00   00   00   00   00

   RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
   00   00   00   00   00   00   00   00   00   00   00   00

CCS LOC    0006    PAGE 01

   TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
   00   00   00   00   00   00   00   00   00   00   00   00   00

   RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
   00   00   00   00   00   00   00   00   00   00   00   00

CCS LOC    0007    PAGE 01

   TS   AF   MS   MT   FS   TF   SF   GF   MR   AB   IM   LB   LA
   00   00   00   00   00   00   00   00   00   00   00   00   00

   RF   FF   MF   CF   WR   SC   VF   WF   XF   SH   BB   AA
   00   00   00   00   00   00   00   00   00   00   00   00

MU**
LC
LOAD COMPLETE
MU**
LI
LOAD COMPLETE
MU**
K
```

SECTION 8

DECODER CONTROL STORE, I/O CONTROL AND ADDITIONAL TOPICS

These topics are not of interest to all microprogrammers. Both decoder and I/O control stores are options and also less trivial to program. Not all applications require an understanding of the item treated as additional topic which is multiple environment applications.

8.1 DECODER CONTROL STORE

Preliminary decoding of instructions in the instruction buffer is performed by the instruction decoder control store and the instruction decoding logic. These elements translate the 16-bit instruction into a 9-bit control-store address according to the contents of the instruction decoder control store.

The instruction decoder control store consists of two 16-word by 16-bit memory arrays. The processor implements this with programmable read-only memory (PROMS). An option of the WCS permits selection of read/write arrays to permit alternate decoding strategies.

The decoder B control store array uses instruction buffer bits 12 through 15 as an address. The decoder A control store array uses instruction bits 08 through 11 as an address. The formats for these two control store arrays are in figure 8-1.

The decoders are identified as A and B. Bits within them numbered right to left starting with zero, so that bit 10 of decoder B is identifed as B10. A and B designations are accepted by microprogram simulator and utility programs.

The decoder address is enabled by the TF and SF fields both equal to 00 and the GF field equal to X1XX. If an interrupt is present, decoding is inhibited and interrupt addressing is used.

Decoder addressing will be inhibited if the IM field equals 11X0. If decoder addressing is so inhibited and no interrupts are present, field-selection addressing is used.

The possible components of a decoded address are shown in figure 8-1 and 8-2. The nine low-order bits obtained from the decoder B are always used in decoder addressing.

The five most significant bits (4-8) in decoder A are included in the control store address bits 4 through 8 by an inclusive OR, if either of the following bit combinations exist in the first decoder output:

B12 equals zero

or

B15 equals zero

The four least significant bits of decoder A are included in the control store address bits 0 through 3 by an inclusive OR if either of the following bit combinations exist in the first decoder output.

B12 equals zero and B10 equals one

or

B15 equals zero and B10 equals one

The contents of instruction buffer bits 04 through 07 are included in the control store address bits 0 through 3 by an inclusive OR, if either of the following bit combinations exist:

B14 equals zero

or

B15 equals zero and A13 equals one

The contents of instruction buffer bits 00 through 03 are included in the control store address bits 0 through 3 by an inclusive OR, if either of the following bit combinations exist:

B13 equals zero

or

B15 equals zero and A13 equals one

One exception to this is the contribution of instruction buffer bits 04 through 07. The contribution to control store address bit 2 will be the contents of instruction buffer bit 03 instead of bit 06, if the decoder B bit 00 equals one and the decoder A9 equals one.

Decoder addressing is used to perform a preliminary instruction decoding function. It permits instruction classes to be discriminated with the detailed decoding performed later by field-selection addressing after the instruction buffer is transferred to the instruction register.

The meaning of other bits in the two decoder control store words is shown in figures 8-1 and 8-2. These signals are available at a processor connector and are used by Varian 70 series options to detect certain instruction classes.

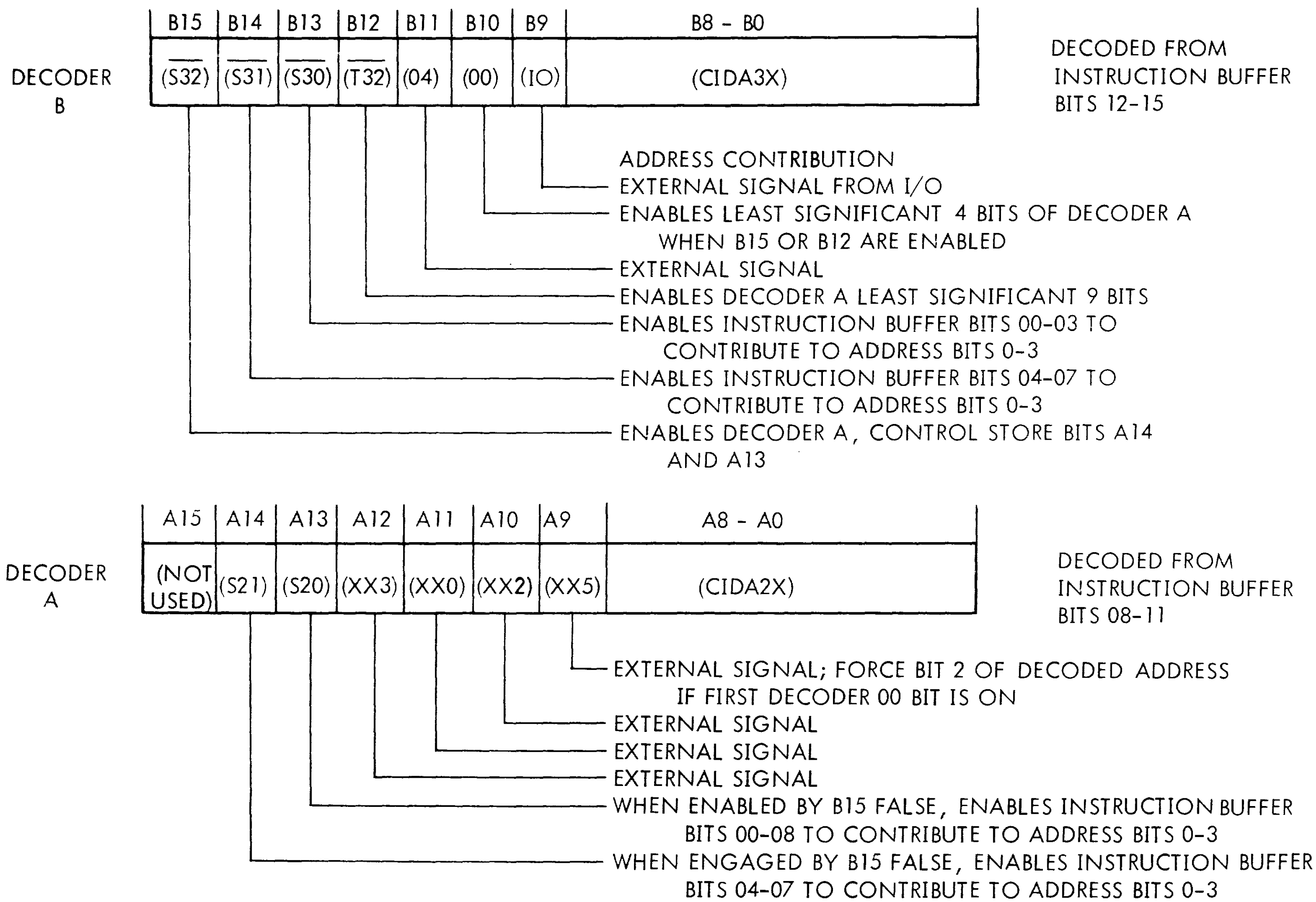
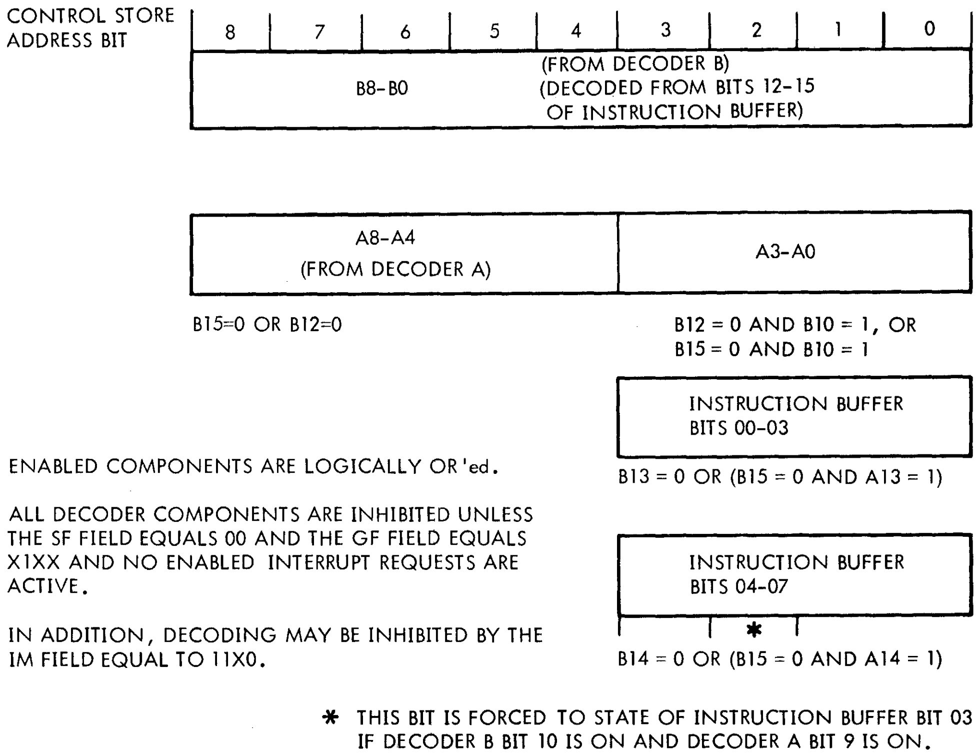


Figure 8-1. Decoder Control Store Format



VTII-1937 A

Figure 8-2. Decoder Address Components

8.2 I/O CONTROL STORE

8.2.1 Microprogram Initiation

The microinstruction can initiate I/O activity by signaling an I/O request while forming a starting address for the independent I/O control store. An I/O request is made by setting the SF field equal to 00 and the IM field equal to 111X. (If the IM field equals 1110, decode addressing is inhibited).

The I/O control-store starting address is specfied by the MT, MR and TS fields.

7	6	5 4 3	2	1	0
MT	MR	TS		AB1*	0

I/O request
SF = 00
IM = 111X

I/O Control Store Starting Address

*AB1 is most significant bit of the AB field

The microinstruction can wait for completion of I/O activity by specifying a wait for I/O done. This is coded by setting the SF field equal to 00 and the IM field equal to 0010. Execution of this and subsequent microinstruction will be inhibited until the I/O sequence is completed. If the I/O is busy performing a sequence and an I/O request is issued execution of the microinstruction specifying new I/O activity will be inhibited until the I/O completes its current sequence.

Standard I/O page zero starting addresses for processor-initiated I/O are:

Hexadecimal Address	Action
04	Sense, EXC or EXCA I/O sequences
0C	Data Input
1C	Data Output

I/O operations can be initiated by external events such as DMA traps. Standard I/O page zero addresses are:

Hexadecimal Address	Action
40	DMA trap out
50	DMA trap in
70	High-speed DMA trap out
80	High-speed DMA trap in
DC	Interrupt

8.2.2 I/O Microprogramming

The I/O control section performs I/O sequences initiated from either the Varian processor microprograms or external DMA trap requests or interrupts.

I/O microprogramming must be undertaken only with a full knowledge of the hardware function of the processor's I/O control section and the WCS's I/O control store. This is described in the Varian 73 Processor and WCS maintenance manuals (document numbers 98 A 9906 02x and 98 A 9906 08x).

No simulator program exists to aid in debugging I/O microprograms.

All special I/O microprogramming must be considered an engineering design more than a programming task.

I/O control performs the following functions in accordance with the sequence I/O microinstructions stored in the I/O control store:

- Control the source of data applied to the I/O register input bus.
- I/O register input bus.
- Control loading on byte shifting of the I/O register.
- Initiate memory cycle requests to the Varian 73 memory control section.
- Initiate I/O bus control signals.
- Wait for completion of external events such as memory cycles, new processor microprogrammed requests, external control signals, etc.
- Signal completion of I/O activity to the processor's central control section.

I/O control store formats are shown in figure 8-3.

The I/O address counter is automatically incremented at completion of each microinstruction unless a "WAIT" or "IDLE" state is entered. This counter is cleared to zero by system reset.

I/O microinstructions are executed from sequential addresses until the end of the sequence whereupon the I/O becomes idle and ready to accept new requests.

As the address counter is loaded with its starting address, the I/O control buffer is loaded with the contents of I/O control store location corresponding to the last contents of the address register. Following a system reset this will be the contents of I/O control store address zero. At all other times it will be the ending address of the previous I/O sequence. In either case, the standard data will cause bits IDLE and DN to become true.

IDLE true indicates the I/O control is not idle and further requests are to be ignored as long as IDLE is true, the I/O address counter and I/O control buffer are enabled.

At each succeeding microinstruction time the address counter is incremented and the I/O control buffer is loaded with the contents of the address designated by the address counter. The 16 bits of the I/O control buffer control all I/O functions. Their use is described below:

CD0 Control the processor's
CD1 I/O data loop multiplexor (IOMXX+)

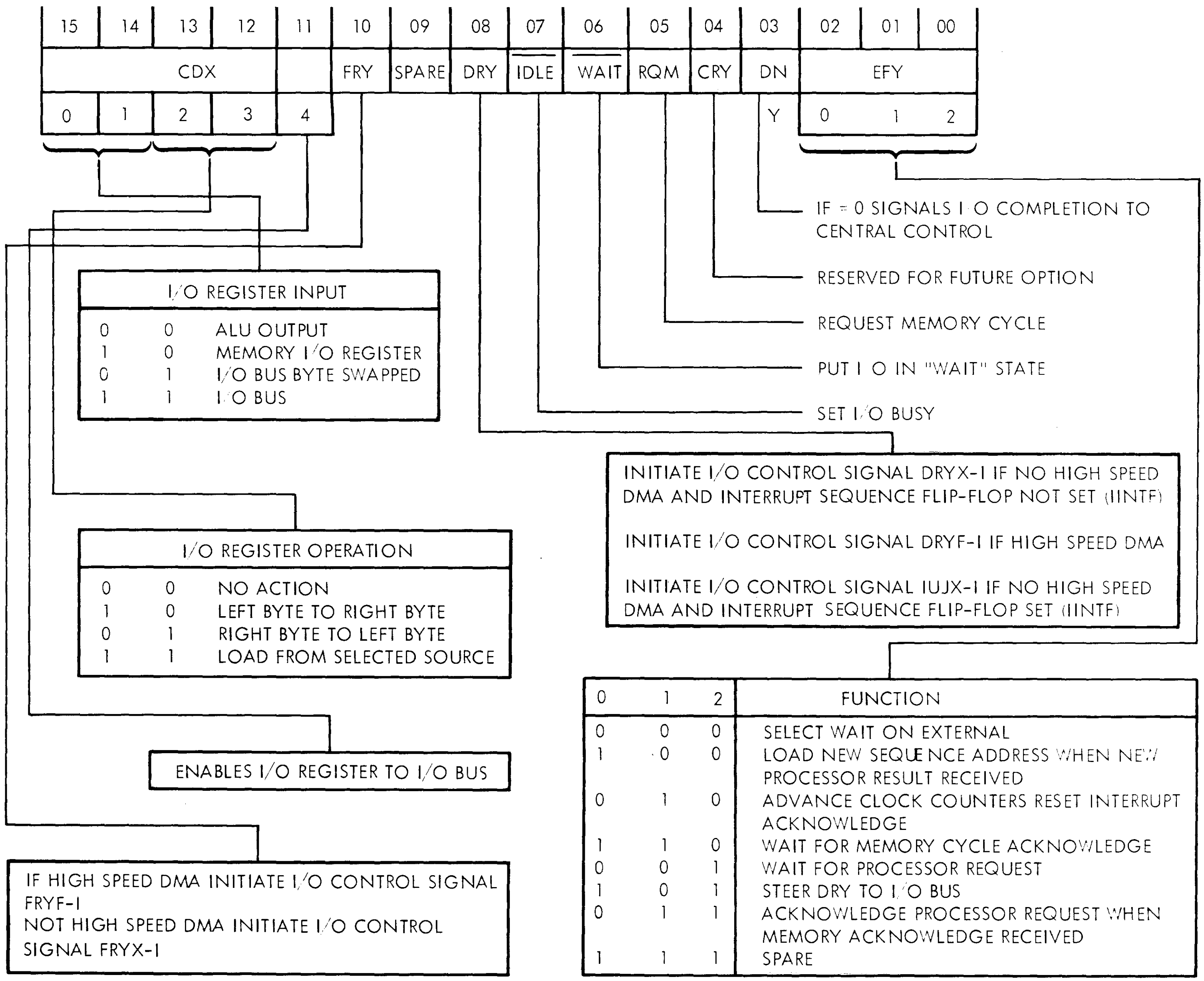
CD 1 0	I/O Register Input
0 0	ALU
0 1	Memory I/O register
1 0	I/O bus byte swapped
1 1	I/O bus

CD2 Control the processor's
CD3 I/O register

CD 3 2	
0 0	No action
0 1	Shift right (left byte to right byte)
1 0	Shift left (right byte to left byte)
1 1	Load from ALU

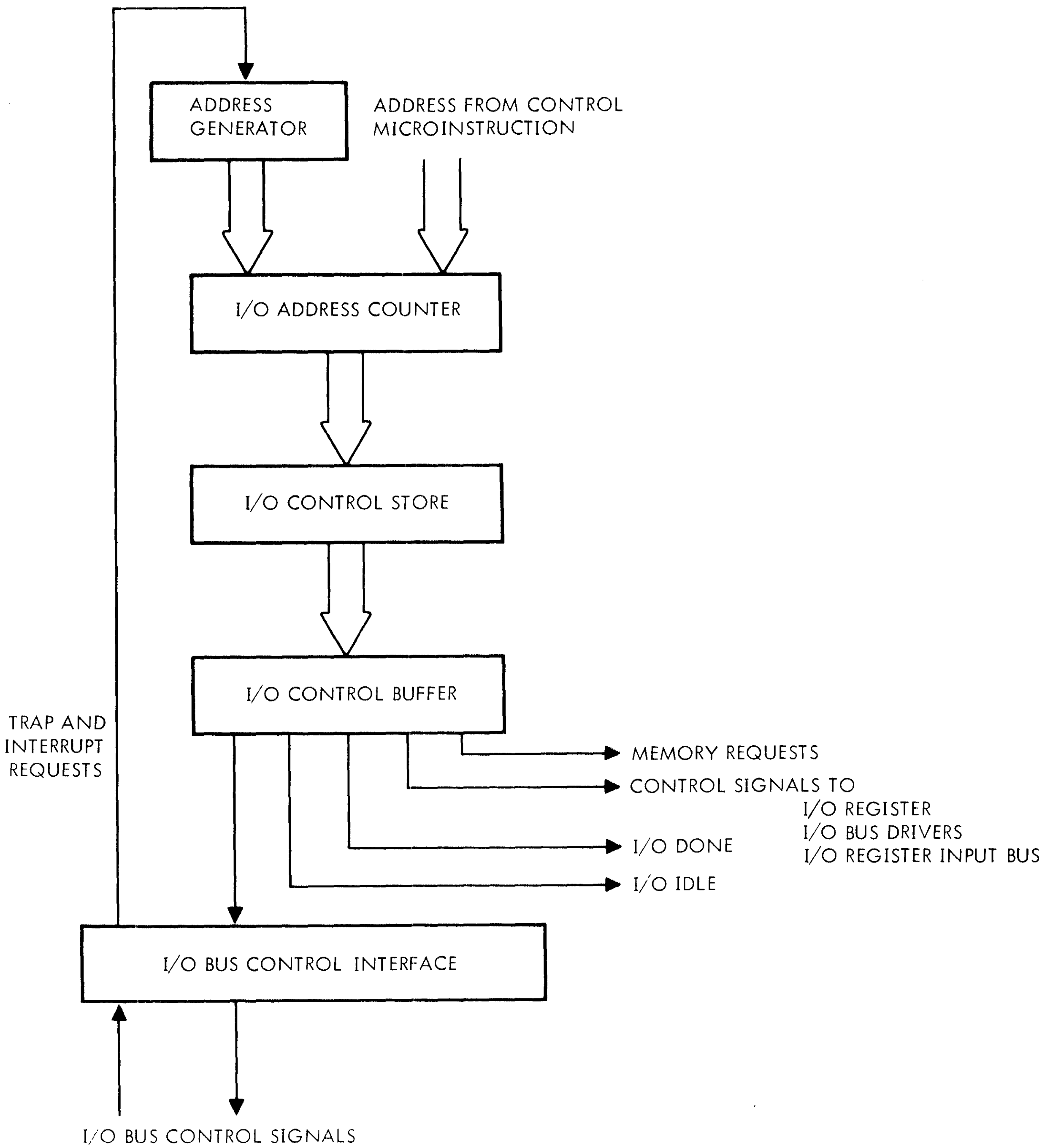
These bits do not directly control the I/O register. The I/O register may also be controlled by IDLE (when the I/O is idle, the register is continously loaded from the ALU).

CD4	Enables the processor's I/O register onto the E-bus.
FRY	Initiates an I/O function ready (FRYX-I) signal. RYX-I is terminated 247.5 nano-seconds later by signal IIIT–.
Spare	Not used.
DRY	Initiates an I/O bus data ready (DRYX-I) signal. DRYX-I is terminated 247,5 nano-seconds later by signal IEDRYN+ derived from IIIT–.
IDLE	Determines idle/busy status of I/O control. While busy the I/O can accept no new requests.



VTII-1811

Figure 8-3. I/O Microinstruction Format



VTII-1934

Figure 8-4. I/O Control Simplified Block Diagram

WAIT Places the I/O control in a "wait" state by inhibiting address counter and ROM buffer clocks until receipt of a designated signal. The I/O may wait for any of the following:

- new processor request
- processor interrupt flag reset
- data memory cycle complete
- external wait signal

Selection of the specific condition is determined by the function bits EF2, EF1 and EF0 of the I/O control buffer.

RQM Requests a DMA memory cycle from the processor's memory control.

CRY Channel request. Reserved for future option.

DN Results in an I/O done signal (IDNC– low) to signal the processor of completion of the I/O sequence.

F2 Function bits which control:

- selection of "wait" condition
- advance of interrupt clock counters
- steering of DRY
- acknowledge interrupt requests
- loading of new sequence addresses

EF 2	1	0	
0	0	0	Select wait on external signal IEXW+
0	0	1	Load new sequence address from CPU if CRQIO+
0	1	0	Advance IUCX and IUCF clock counters
0	1	1	Select wait for memory cycle complete
1	0	0	Select wait on CPU request
1	0	1	Steer DRY to DRYX-I
1	1	0	Acknowledge interrupt sequence request from CPU
1	1	1	Not used

Any I/O sequence continues through successive ROM addresses until address counter and ROM buffer clocks are inhibited by either of two conditions:

IDLE becomes false signifying end of sequence or
WAIT becomes true signaling that the current sequence must stop to wait for some external event such as:

- memory cycle
- new processor request
- new processor request
- interrupt flag set
- external wait line active

For programmed I/O sequences signal DN will become active and at the next microinstruction time IDLE will become active also. IDLE causes I/O sequencing to stop.

The I/O sequence is thus completed leaving the address counter loaded with an address whose contents IDLE and DN. This will be the first data loaded into the ROM buffer when clocks are reenabled.

8.2.3 Example of I/O Microprogram: Clear and Input to A

The flowchart and code sheet following describe the standard programmed I/O sequence for V73 input data transfers. The corresponding flowchart for the processor microprogram to initiate the I/O transfer may be found in the second volume of the System Maintenance Manual.

Referring to the processor microprogram flowchart for the sequence required to start the I/O operation, the first central control address is 1A0. This was obtained with decode addressing. The entire sequence will now be traced.

IABM1 (1A0)

This microinstruction causes the operand register to be loaded with a mask word containing only bit 13 true. Normal addressing specifies the next address.

IABM2 (1C3)

This microinstruction specifies an I/O request with an I/O starting address of OC. If the I/O was idle (the I/O control store buffer IDLE bit was a zero) the I/O control accepts the starting address and simultaneously loads its control buffer with a standard code of 0088. This places the I/O in its "busy" state and signals the processor that the I/O operation was accepted.

During this microinstruction the processor transfers the operand register to register E (this register has been designated S1).

IABM3 (1F3)

This microinstruction logically OR's the contents of register E with the masked (bits 0-8) contents of the **instruction** register. This places the device address, function code and bit 13 (specifying an input transfer) at the ALU output.

In the I/O control the I/O microprogram is executing the microinstruction at location OC which loads the I/O register with ALU output data.

The processor microprogram specifies a "Wait for I/O Done" which causes further processor operations to be suspended until the I/O control signals completion. The remainder of the I/O sequence will now be traced. Addresses are sequential.

I/O address OC is "NOP". It performs no function.

Table 8-1. I/O Microprogram Example Code

I/O address of continues to enable the I/O register to the I/O bus and generates the IFRYX-I control signal to signal I/O devices that a new address and function code may be sampled.

I/O address 10 performs the same function as OF. This allows for I/O bus settling time.

I/O address 11 selects the I/O bus as an input to the I/O register. The selected I/O device may place its data on the I/O bus.

I/O address 12 continues to select the I/O bus as an input to the I/O register and generates control signal IDRYX-I.

I/O address 13 continues to select the I/O bus as an input to the I/O register, continues to generate IDRYX-I and causes the I/O register to be loaded with the data placed on the I/O bus. I/O control buffer bit "DN" becomes false permitting microinstruction execution to proceed.

I/O address 14 returns the I/O control to an idle condition. Simultaneously the next central control microinstruction is executed.

CIA (09D)

This microinstruction transfers the I/O register contents to register 0 (the A register). The program counter is incremented and a new instruction fetch is initiated. The microprogram branches to SS3M (02D) where the instruction buffer is decoded to branch to the start of the next instruction.

Note that I/O address 15 will be executed when the next I/O operation is started. This microinstruction contains the standard code of 0088 which will place the I/O in its "busy" state.

8.3 MULTIPLE ENVIRONMENT APPLICATIONS

This section describes using the Varian 70 series WCS for extended instruction execution and dual/multi environment applications.

This section discusses the application of WCS to extend the standard V70 series emulation of a Varian 620/f to perform additional instructions and functions. It also discussed a dual environment implementation, which can be extended to multi-environment machine.

Application of the WCS to Extend Execution Capabilities

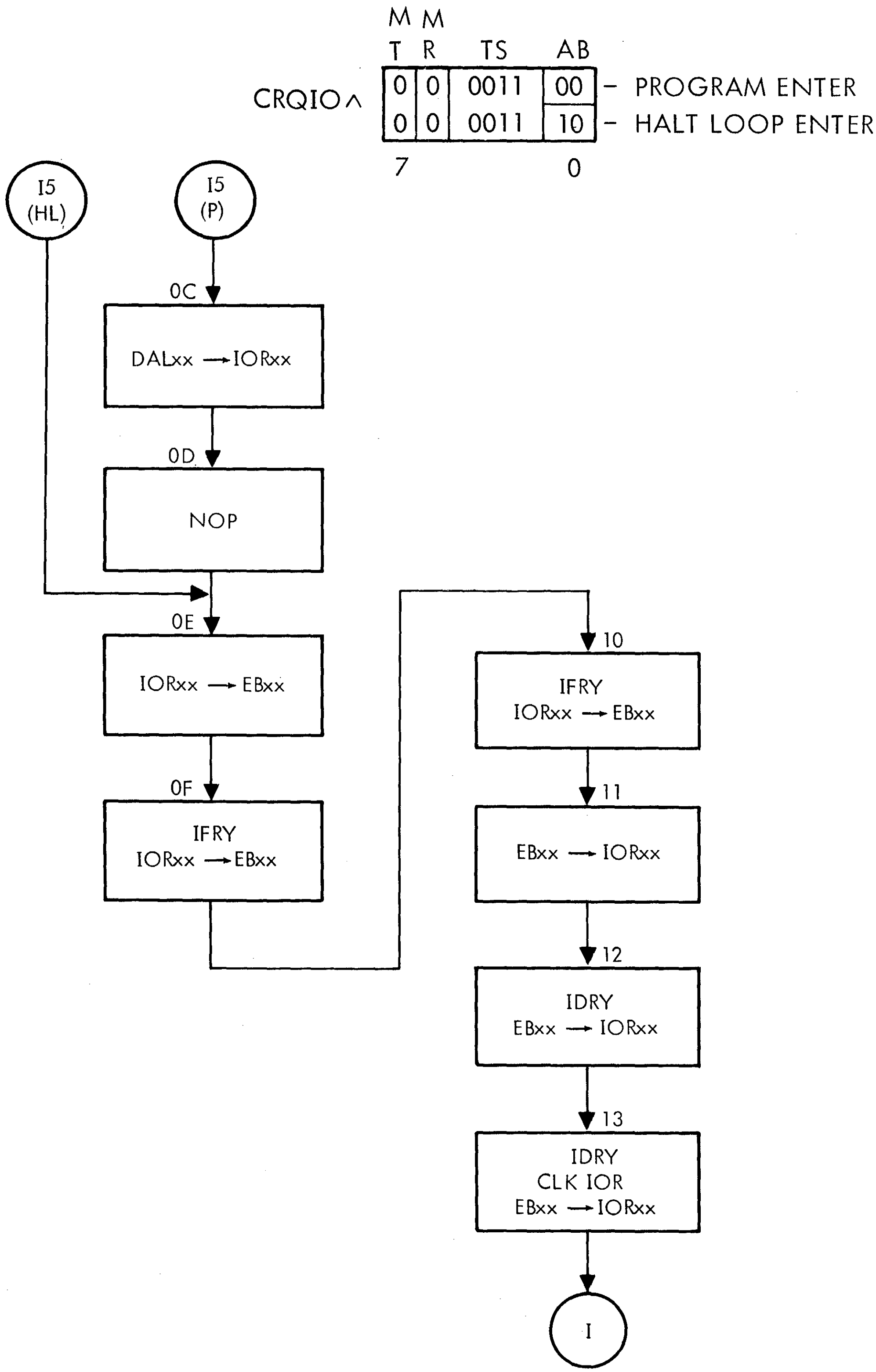
Using the macro BCS, it is possible to define entry points in extended micro store for a large number of special functions. These extended functions can be defined to use V70 series hardware not explicit in the 620/f emulation such as 16 general purpose accumulator registers and more explicit status testing. Examples of application of this capability would be implementation of floating point arithmetic, stack organizations and so on. Characteristic of extended operations is that no primary decodes would occur during the operation (exceptions are possible of course). It is possible to enable interrupts while disabling primary decode so it would be possible to allow interrupts during very long microsequences. However, the point of interruptability and its ramifications would have to be carefully considered.

Application of the WCS to Dual/Multi Environment Operation

Emulation of instruction architectures other than that of the host machine is achieved by performing primary control store address decoding in the WCS extended control store. It is possible to have unique architecture in each 512 word block of control store. Some possible examples of this would be:

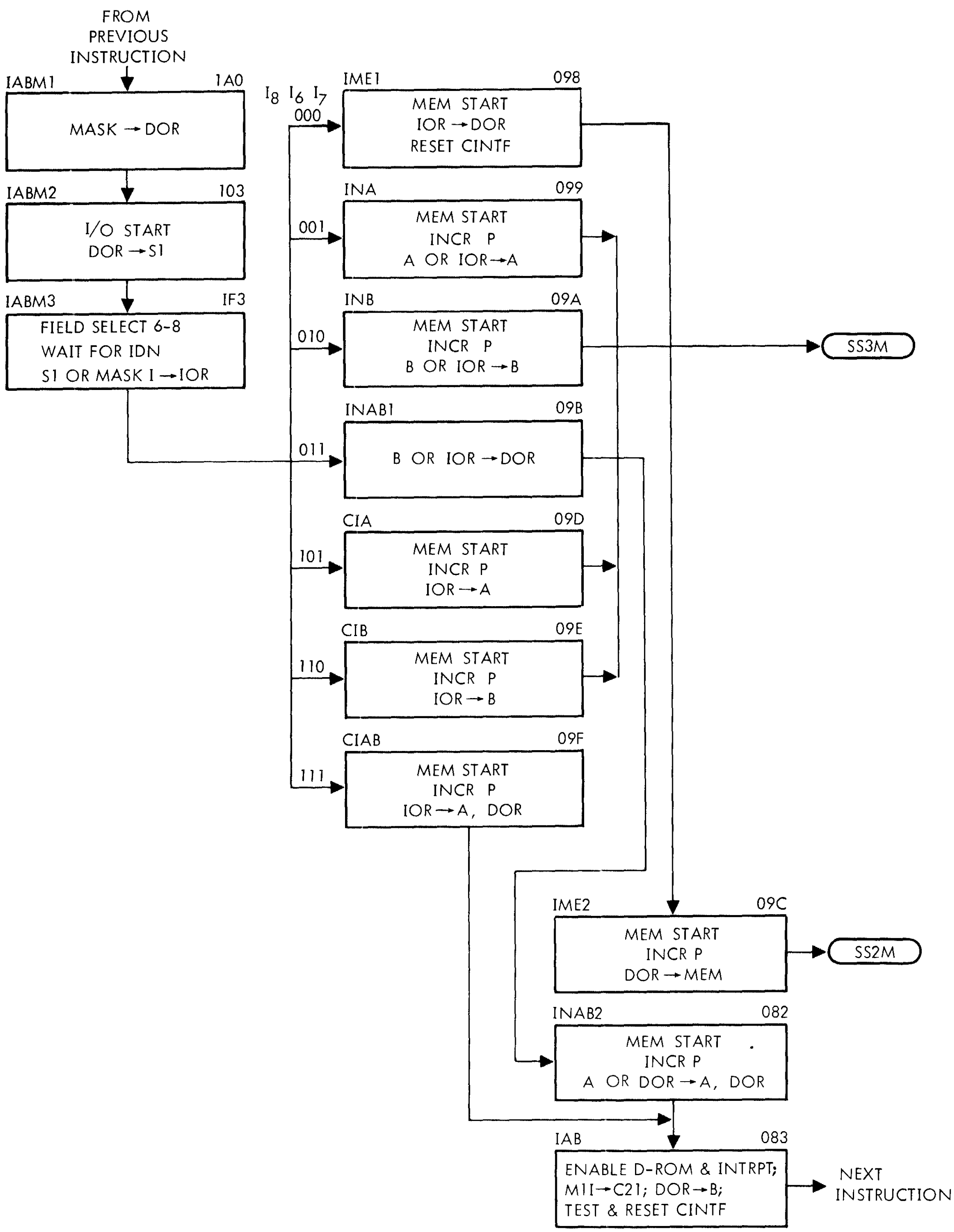
1. Hardware emulation of a VXX machine under control of WCS in the V70 series systems.

2. Implementation of a higher level language processor operating under control in the V70 series systems.



VTII-1812

Figure 8-5. Flowchart of I/O Microprogramming Example



VT11-1815

Figure 8-5. Flowchart of I/O Microprogramming Example *(continued)*

An Example of a Second Environment Architecture and Call Sequence

For our example, we will define a second environment E2 (as distinguished from the V70 series system environment E1) which can use general registers of the V70 series systems as stack pointers, general purpose accumulators and so forth. The question arises as to interruptability of this second environment and what registers are available to E2.

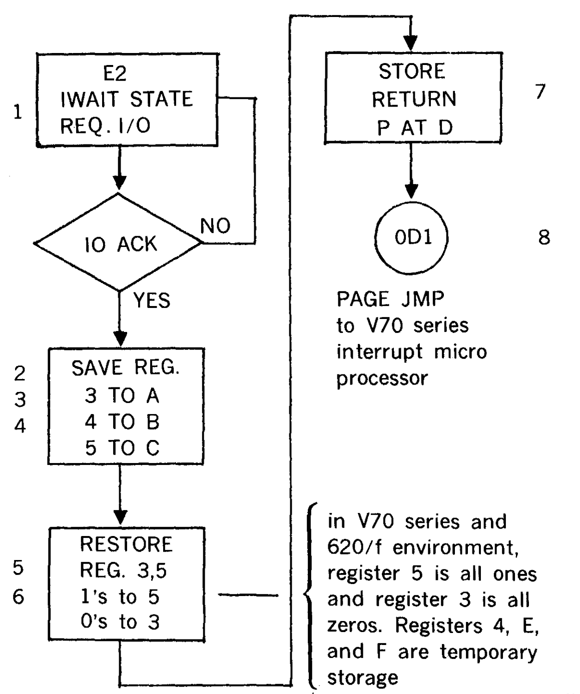
A macro sequence to call E2 from the V70 series systems could be:

P	BCS (105000) page jump to E2 entrance micro
(P) + 1	xxxxx LOC of first instruction of E2 in main memory
(P) + 2	BCS (105001) page jump to E2 interrupt return entrance

E2 Entrance and Interrupt Micro Code

The normal entrance micro code saves (P) + 2 at register E for reference in case of an interrupt. Also, it can be used to return jump to the next V70 series system instruction when environment 2 is completed.

Upon receiving an E1 interrupt while in E2, the microsequence (simplified) is as follows:



(continued)

The content of E is the return instruction location as required by control word 0D1. Only registers 3,4,5, E and F may be subsequently modified by 620 code and it is only necessary to save 3,4,5 as the return path will supply restoration of E.

The interrupt return is implemented via the BCS at the V70 series interrupt return reference. The interrupt return entry code restores registers 3, 4, 5 from A, B, and C respectively and stores the location of the interrupt return BCS in E. The code then restarts the instruction pipeline at the reference stored in D. Note that the 70 series interrupt routine is responsible for maintaining A, B, and X registers (0,1,2).

E2 Register File Usage

We can now see that the second environment has 10 registers (0-9) available for general purpose use, while E is allocated for the interrupt return page jump instruction address. Registers A, B, C, D and F are also available for intermediate usage between interruptable states.

Considerations of Saving and Storing Status

The above example does not define how status is to be saved and restored. This should be considered when defining the interruptability of the second environment. In any event, register and overflow status will be maintained by the V70 series environment interrupt routines but the equal, less than and greater than status is more difficult. This may involve saving the status in the interrupt return micro code.

Further Discussion of Multi-Environment Systems

The above example of interrupt handling in multi-environment machine is presented as an exploration of a mechanism which solves the problem given a particular set of system restraints (interrupt service routines are in the host V70 series environment and do not use other than normal 620/f instructions, i.e., instructions only use registers 0, 1, 2, 3, 4, 5, E, F).

Each different set of environments may require different mechanisms of interrupt handling. Some will require saving registers in main memory, possibly at locations relative to the location of the interrupt return page jump. An alternate environment might utilize its own I/O drivers, which would involve locking out interrupts and swapping out interrupt entrance code and possibly also the interrupt processing routines. In this situation the second environment might offer system executive control as well as its optimized functions. When environment, register save/restore will probably have to be comprehensive and in main memory.

Other Multi-Environment Considerations for

the V70 Series System Reset

The system reset function will normally be wired to return control to the host module (normally zero).

Power Fail/Restart

The system executive is expected to contain the necessary job restart information in case of a power fail. Therefore, the host environment is not required to save facilities of an alternate environment (some of which are unknown to the host machine). The E2 environment could be saved if desired by using a special instruction such as a 620/f extension macro which saves and restores the file.

Step Mode

If it is desirable to single step computer operation in alternate environments, it is necessary to micro code a halt loop in that environment. The alternate environment has the option of enabling or disabling the step function in its micro code.

Conclusion

This section described two basic applications for the Varian 70 series WCS. Its use for extending the instruction set of the standard 620 emulator is quite straight forward. Its application to produce a dual or multi environment machine was also seen to be practical and feasible with the system problem of interrupt handling examined in some detail. In fact, a second environment which offered 10 general purpose registers and 5 scratch registers for implementing stack/queue pointers, floating point registers or whatever, was demonstrated.

Because of the ability to add new instructions to the 620 emulation in the V70 series and the flexibility of micro coding, the example is really only one of many possibilities. The mechanism generally will be designed to meet requirements of the system definition.

SECTION 9
GLOSSARY

Entries are brief descriptions of terms appearing in the text. These definitions reflect the usage preferred for consistency and a minimum of terms. Whenever two words have been used previously for the same item a choice was made in favor of the most meaningful and unambiguous.

AA	microinstruction field of bits 0 - 3 to select an ALU source on bus A and/or destination
AB	microinstruction bit 35, which is used in field-selection addressing and I/O requests
addressing	determination of next instruction to be executed
AF	microinstruction field which contributes to address generation
ALU	**Arithmetic and Logical Unit**, the logical and storage providing data transfer and basic arithmetic and logical operations in the processor
ALUC	flag for ALU carry, bit 11 of processor status word
ALUO	flag for ALU output all ones, bit 9 of processor status word
ALUS	flag for ALU sign, bit 10 of processor status word
ALUZ	flag for ALU output all zeros, bit 2 of processor status word
application software	program oriented to solving problems rather than managing systems resources
ASCII	**American Standard Code for Information Interchange** codes for character representation
assembler	computer program which translates symbolic statements into machine executable instructions, see MIDAS
BB	microinstruction field of bits 4 through 7, which specify the ALU *(continued)* source on the B bus or a part of mask literal
BCS	mnemonic for **Branch to Control Store**, a 16-bit **MACRO** instruction which initiates execution of microprograms in WCS
BIC	Buffer Interlace Controller
binary	numbering system in which only two states are represented, one and zero
BYTA	flag which indicates left or right byte of word
byte	8-bit unit
CF	microinstruction field which varies the type of carry action on ALU actions
control buffer	contains current microinstruction being executed; separate for central control logic (64 bits) and I/O control logic (16-bits)
control store	memory in which microinstructions are stored
cycle	time required to execute one microinstruction
cycle, memory	time required to access and restore storage in main memory
cyclic redundancy check	technique for validating storage or transmission reliability
data path	transfer media for data within processor
DCS	Decoder Control Store, optional programmable control store for instruction decoding

DMA	Direct Memory Access
direct addressing	instructions contain actual effective memory address to be used, in contrast with relative or indirect addressing
emulation, 620	standard microprogram that resides in control store page 0, and directs execution of Varian 620 instructions
FF	microinstruction field which specifies ALU action
field select	technique of addressing which uses the bits of the instruction register to determine a microprogram branch address
GF	microinstruction field, which specifies condition to be tested
GPR	general-purpose register, one of 16 16-bit registers
GPRS	general-purpose register 0 bit 15 (sign)
hexadecimal or **hex**	numbering system using base 16, representing numbers with digits and letters A through F
IF	Instruction Fetch
IIA	interrupt address supplied by option board to indicate type of interrupt
IM	microinstruction field designating type of memory control
instruction buffer	storage for instruction immediately after fetched from memory
instruction register	storage for instruction for an instruction to be executed
IOCS	**I/O Control Store**, optional unit of programmable storage for varying I/O rates and disciplines
IOR	I/O Register
key register	four-bit register which supplies signals for memory operations used by memory-map option
LA	microinstruction field which in conjunction with AA specifies the ALU input on bus A
LB	microinstruction field which in conjunction with BB specifies the ALU input on bus B
MAD	Memory Address Register
mask	literal constant ANDed with instruction register
memory map	hardware option to allow addressing memory to 256K
microinstruction	64-bit word from WCS specifying the actions to occur during one cycle
microprogram	vehicle for implementing control function of a computer
MIR	Memory Input Register
MIRS	flag for memory input register sign
MK	16-bit mask field (assembler mnemonic)
MR	microinstruction bit 37 used to specify I/O address bit 6 or to control AB field use
MS	microinstruction addressing field
MT	bit 50 of microinstruction which specifies bit 7 of an I/O address
MULS	Multiply Sign flag
NORM	Normalize flag
OF	Operand fetch
OP	microinstruction fields combined to specify ALU action (bits 23 - 17)
OPR	operand register
overflow	ALU action indicated by OVFL flag; condition caused by attempt to push too many addresses into microprogram stack
P	program counter
page	unit of writable control store of 512 words, 64 bits each

page jump	a branch with a microprogram beyond the extent of the page currently being executed
pop	to remove an address from top of microprogram stack
processor	unit that performs and controls execution of instruction
program counter	register for memory address; usually used for keeping track of **MACRO** level execution
push	to add an address to top of stack
pipelining	technique which allows next instruction to be fetched during an otherwise unused memory cycle
QUOS	flag for quotient
RF	microinstruction field of bits 24 through 26 used to specify transfer and increment of some special registers
ROM	Read Only Memory: such as page 0 of V70 series control stores; contains the microinstructions to emulate Varian 620 system
SC	bit 15 of microinstruction; specifies shift of operand register or is part of mask literal
SF	bits 42 and 43 of microinstruction; specify interpretation of the IM field
SH	microinstruction field which specifies some special ALU actions or shift operations
SHFT	flag for shift
SHTC	flag for overflow of the shift counter
stack, microprogram	linked storage locations (16) used in microprogram subroutine call and return
STAT	processor status word
STEP	mode of computer execution one instruction at a time
SSW	SENSE switch 1 - 3 on control panel
SUPR	supervisor mode flag, bit 1 of processor status word
TF	microinstruction field of bits 45 and 46 which specify whether testing occurs and whether it is for true or false condition
TS	microinstruction field of bits 60 through 63, which selects a field from the instruction register, specifies a page number for a page jump, contributes a portion of an I/O address, or enables selected interrupts
underflow	condition upon attempting to remove or pop more addresses than are in a microprogram stack
VF	microinstruction bit 14, which specifies moving bit 15 of R0 to divide-sign bit (DSB), or a part of mask
WCS	Writable Control Store; which is read and loaded over the I/O bus
WR	microinstruction field of bit 16 that specifies whether or not the general-purpose registers are being loaded
WF	single bit (13) in microinstruction to designate transfer of the ALU

varian data machines

varian data machines /a varian subsidiary