MICROPROGRAMMING REFERENCE MANUAL

Microdata 3200

3200 MICROPROGRAMMING REFERENCE MANUAL

February, 1976

®

Microdata Corporation
17481 Red Hill Avenue, Irvine, California 92714
Post Office Box 19501, Irvine, California 92713
Telephone: 714/540-6730 · TWX: 910-595-1764

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

1 INTRODUCTION

1.1 THE MICRODATA 3200 COMPUTER

The Microdata 3200 is a high-performance, low-priced computer which employs microprogramming techniques to expand user capabilities. This manual describes the logical and physical structure of the 3200, and presents microprogramming procedures for users wishing to develop a computer that is an extension of the standard Microdata 32/S computer.

The 3200 Computer

The Microdata 3200 is a 16-bit machine with 16K bytes of 350 nanosecond MOS main memory, addressable to the byte level. It is microprogrammed using a bipolar 32-bit Control Memory (expandable to 4K words) that has a 135 nanosecond cycle time.

The 3200 utilizes a common bus, called the MONOBUS, for accessing all main memory modules and I/O device controllers. Memories and controllers of various speeds may be mixed on the asynchronous MONOBUS and uniformly accessed with standard memory reference instructions. Overlapped bus requesting and data transferring permits very high-speed data transfers.

Input/Output (I/O) can be byte or word oriented under program control, or block oriented under either computer control (concurrent I/O) or controller hardware control (Direct Memory Access). Four external interrupt lines establish the relative priorities of groups of I/O device controllers. Relative priority among the controllers on each line is established by their positions along the MONOBUS. Each I/O device controller may be manually assigned to a specific address and interrupt line. A unique interrupt processing procedure and environment may be specified for each I/O device address.

3200 Microprogramming

The standard Microdata 32/S computer is implemented via firmware on the microprogrammable 3200. To develop a computer that is an extension of the standard 32/S, an appropriate set of microprograms (i.e., firmware) must be developed, debugged, and stored in the Control Memory of the 3200. The 3200 then efficiently emulates the computer defined by the special microprogrammed firmware.

Meaningful microprograms may be realized through the use of the CAP32 Microassembly Language. Source routines written in CAP32 are assembled by the CAP32 Microassembler, which is a program written in PL/1 for operation on host machines such as the IBM 360/50, 370/145, or larger machines of that family. A CAP32 Microassembler that will run on the Microdata 32/S is currently being planned; consult a Microdata representative for availability information.

Conventional programming can often be performed with little attention being paid to hardware aspects. This is not true of microprogramming. The 3200 structure makes it mandatory that the programmer have a thorough knowledge of the logical and physical structure of the system. The 3200 is competent over a wide range of problems, but is superlative where the problem matches the optimization features of the machine. Thus, the programmer must know the 3200 hardware from a logic viewpoint, must know the CAP32 language, and must know the effects of one upon the other.

The 3200 machine language microcode that results from the CAP32 micro-assembly is stored in the 3200 Control Memory, defining a user-level machine language. The "3200-x" machine defined by that microcode will then execute user-programs written in user-language "x".

1.2 HOW TO USE THIS MANUAL

This manual is written in modular format with each pair of facing pages presenting a single topic.

The approach taken in this manual differs substantially from the typical reference manual format. Here each pair of facing pages discusses an individual topic. Generally the left-hand page is devoted to text, while the right-hand page presents figures referred to by that text. At the head of each text page are a pair of titles, the first one naming the section and the second one naming the topic. Immediately below these titles is a brief summary of the material covered in the topic.

The advantage of this format will become readily apparent to the reader as he begins to use this manual. First of all, the figures referred to in the text are always conveniently right in front of the reader at the point where the reference is made. Secondly, there is a psychological advantage to the reader in knowing that when he has completed reading a topic and goes to turn the page, he is done with one idea and ready to encounter a new one.

A scan of the Table of Contents provides a quick overview of the objectives of this manual. The topics should normally be studied in the sequence in which they are presented; however, extensive cross-references (and a complete index) permit random access. Cross-references are provided via footnotes which reference the applicable topic(s) by topic number.

A complete set of reference tables are provided in the appendix to this manual. These tables are referenced as needed throughout the topics.

Special symbols are used throughout this manual for purposes of clarity and conciseness; these symbols are defined in Figure A. In presenting general instruction and code formats, certain conventions apply; these conventions are defined in Figure B.

Symbol	Description
.	*The period character (.) is used in signal names to specify the applicable bit(s). For example, "FBUS.3" denotes the third bit of the FBUS, while "Z.2" denotes the second bit of the Z-Register.*
:	*The colon (:) is used as a separator for range limits. For example, "0:9" denotes the numeral 0 through 9, while "FBUS.4:0" denotes bits 4 through 0 of the FBUS.*
:=	*This sumbol is used to denote replacement. For example, "X := 13" means that the current content of the X-Register is replaced by the value 13, while "Y := Y+1" means that the current content of the Y-Register is incremented by 1. This symbol is primarily used in the Reference Tables in the appendix to this manual.*

Figure A. Special Symbols Used in This Manual

Convention	Description
UPPER CASE	*Characters or words printed in upper case are required and must appear exactly as shown.*
lower case	*Characters or words printed in lower case are parameters to be supplied by the user.*
[ ]	*Brackets surrounding a word and/or parameter indicate that the word and/or parameter is optional and may be included or omitted at the user's option.*

Figure B. Conventions Used in General Formats

2 HARDWARE ORGANIZATION

2.1 AN OVERVIEW

The hardware of the 3200 is divided into two main categories: the Central Processing Unit (CPU) and the External Devices. Communication between the CPU and the External Devices is via the high-speed MONOBUS.

The overall organization of the 3200 hardware is illustrated in Figure A. The CPU is divided into six main sections:

- The Micro-Control Section
- The User-Level Instruction Fetch Section
- The Data Interface Section
- The Arithmetic/Logic Section
- The Local Memory and Auxiliaries Section
- The Front Panel Section

These sections of the CPU are synchronized with a 135-nanosecond clock; they are described in detail in the remaining topics of this section.

The External Devices consist of the Main User-Level Memory and a full complement of peripherals (such as tapes, discs, terminals, and communications controllers). All external devices, including the Main User-Level Memory, work asynchronously from the CPU and communicate over a high-speed bus arrangement, called the MONOBUS.

The MONOBUS contains 16 data lines, 18 address lines, and 19 control lines, providing high-speed data transfers in as little as 500 nanoseconds (depending on the speed of the external device). The MONOBUS is controlled by the CPU and the Direct Memory Access (DMA) unit. Allocation of the MONOBUS to one of the controlling units is by priority polling, each unit holding the MONOBUS just long enough to effect the current transfer. The MONOBUS includes 4 priority interrupt lines to signal the CPU for service.

The Main User-Level Memory uses MOS semiconductor technology, and has a write and read access time of 300 nanoseconds. The memory module contains 16K bytes of storage, and is addressed at the byte level. A two-byte word structure is superimposed, with word addresses being even. Word addresses therefore correspond to the byte address of the more significant (left) byte of the word. Data transfer is via 16-bit words over the MONOBUS.

With the MONOBUS addressing capability of 18 bits, there are 256K addresses available. Some of these are reserved for I/O devices, and for addressing the Control Memory for the transfer of microcode. The rest are available for addressing main memory.

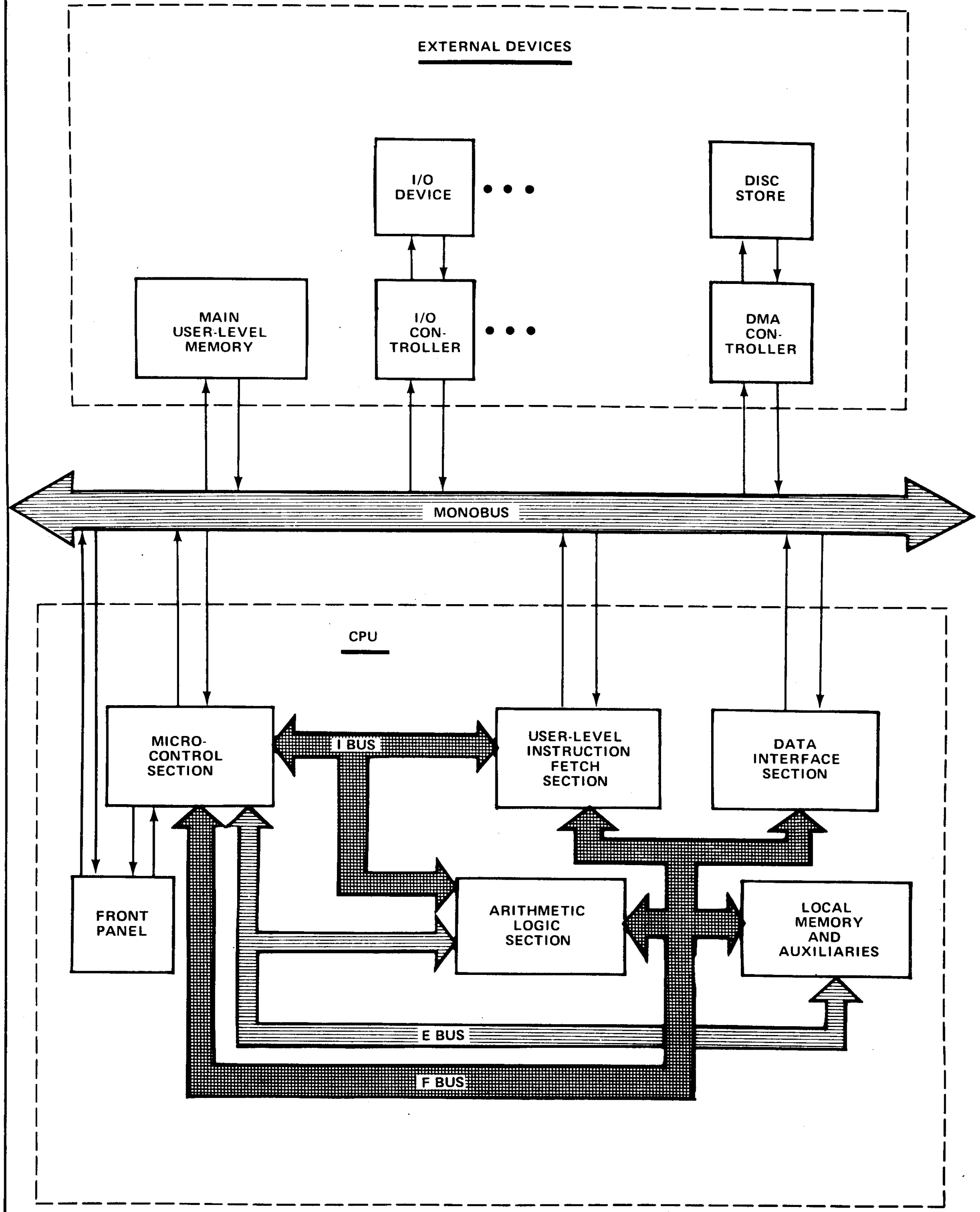


Figure A. Overall Organization of 3200 Hardware

2.2 THE MICRO-CONTROL SECTION

The structure of the Micro-Control section is shown in Figure A. The eight parts of the Micro-Control section have significant roles in microprogramming.

To the programmer, the 32-bit Control (C) Register is probably the most important item in the 3200. The C-Register receives and executes a single 32-bit microcommand from the Control Memory. Control signals diverge from the C-Register and effect gates throughout the CPU and on the MONOBUS, thus controlling all data transfers and transformations. The programmer must know the effect of each C-Register bit and how statements in the CAP32 language effect those bits.[1]

The Control Memory (32-bits/word, 4K words max) stores the microroutines which define the behavior of the user-level machine. The Control Memory may be implemented either in Read-Only Memory or in Writable Control Memory (or a blend of both).

Control Memory Addressing Logic provides decoding for the 8 addressing modes of the Control Memory. Since the Control Word is short (32 bits) and is accessed from the Control Memory with each CPU clock, optimum use must be made of Control Memory addressing bits. The CAP32 Microassembler assists in this regard, as most Control Memory addressing is handled automatically by CAP32. The programmer must know the addressing modes sufficiently to understand CAP32's limitation problems, and modify his microroutine structure so that CAP 32 can cope with his demands.[2]

The 12-bit Last Access (L) Register holds the Control Memory address of the microcommand being executed. In several Control Memory addressing modes, the L-Register contents form part of the address of the next control word accessed.[3]

The 16-bit Save (S) Register is a multipurpose register which receives and saves the contents of the L-Register, and effects a subroutine return. The S-Register can receive data from the CPU on the FBUS. These data are used for Control Memory addressing.[4]

The Plus-Minus Branching Test Logic monitors selected bits throughout the system, and (under microprogram control) modifies the least significant bit of the Control Memory address under designated test conditions.[5]

The Procedure Control Branch Logic responds to machine conditions and external interrupt signals. Normally, the handling of machine faults and interrupts occurs automatically through the efforts of the PC Branch hardware at the time of each first digit branch. However, microroutines must be provided to define desired responses to various conditions arising.

The 8-bit Query (Q) Register contains a mask word controlling the set of interrupts enabling the PC Branching Logic. The Q-Register is set from the FBUS.[6]

References:

[1]Topic 3.2
[2]Topic 6.1
[3]Topic 6.1
[4]Topics 7.4, 7.5, and 8.2
[5]Topics 7.1, 7.2, and 7.3
[6]Topic 8.4

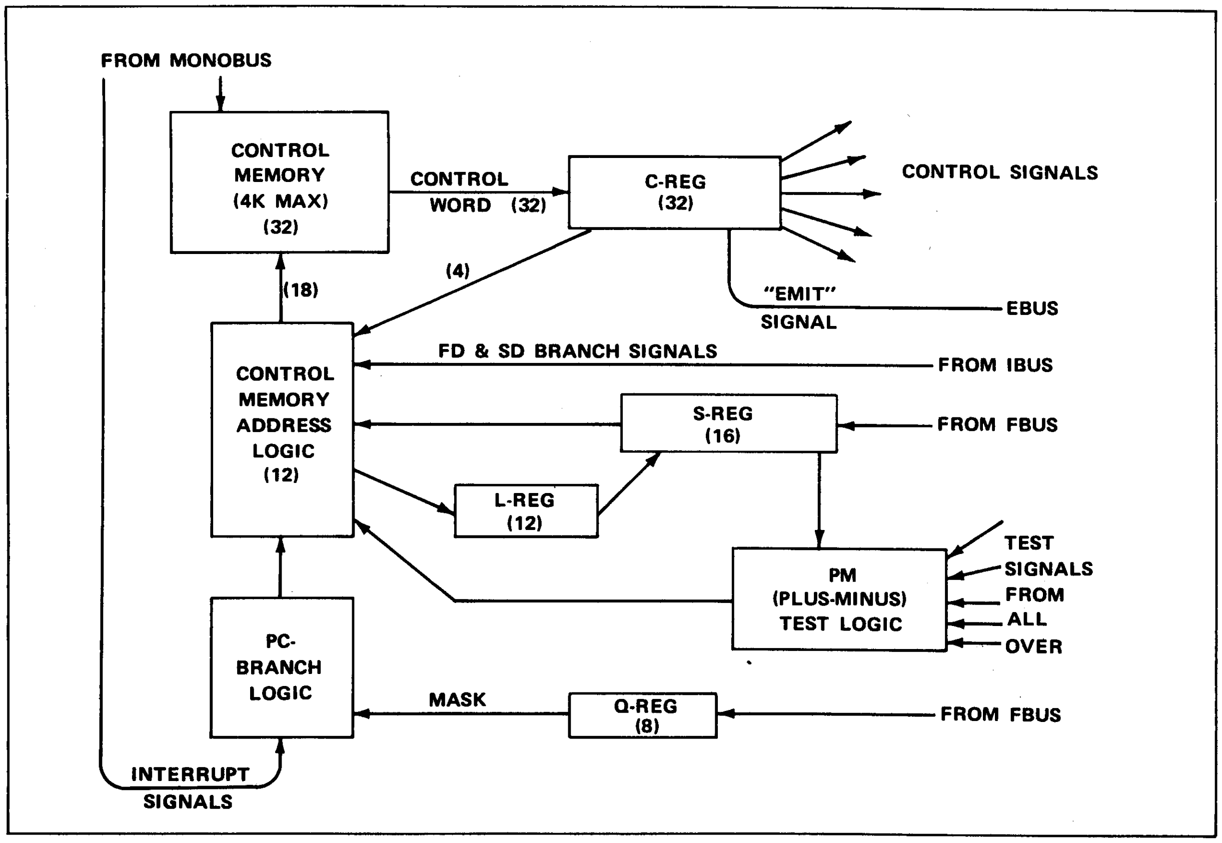


Figure A. The Micro-Control Section

2.3 THE USER-LEVEL INSTRUCTION FETCH SECTION

The User-Level Instruction Fetch section of the Microdata 3200 is the mechanism which implements the user-program at the microlevel.

The microroutine is a sequence of microcommands which carry out a single user-level machine instruction. The User-Level Instruction Fetch section retrieves and interprets user-instructions one at a time, transferring microlevel control to the microroutine which will execute the microcommand sequence for instruction fetched.

Figure A illustrates the User-Level Instruction Fetch section; this section contains two main registers:

- Instruction (I) Register
- Program (P) Register

Associated with the I-Register is an I-Save section including the First Digit (FD) and Second Digit (SD). This portion of the 3200 also includes two miscellaneous items: the General Indicators (GI) and the W-Counter.

The 18-bit P-Register is 1 of 2 CPU registers capable of directly addressing the MONOBUS. The other is the M-Register, which is used to fetch data. (The P-Register has the higher priority.) The P-Register fetches instruction words and any other words in the program stream from the Main User-Level Memory into the I-Register. When the P-Register is incremented to an even value corresponding to a word address in Main Memory, the instruction fetch cycle is automatically initiated and the P-Register addresses the MONOBUS. About 300 nanoseconds later (barring MONOBUS congestion) the fetched instruction or program stream word appears in the I-Register.

The 16-bit I-Register word is subsequently handled byte-by-byte in the CPU, with the Least Significant Bit (LSB) of P-Register determining which I-Byte is to be used. An I-Byte may be sent to the Arithmetic/Logic section via the IBUS and BBUS. Alternatively, the First Digit (upper 4 bits of the current I-Byte) may be used to control a 16- or 32-way branch in Control Memory. The P-Register is incremented automatically by a First Digit Branch (FDB). The programmer can optionally increment the P-Register when an I-Byte is sent to the Arithmetic/Logic section.[1]

The I-Save Register retains the current I-Byte when the FDB is used, freeing the I-Register to receive the next I-Word prior to the time needed.

SD, the second digit of I-Save, is available for a second 16- or 32-way branch, or for further general decoding in the Arithmetic/Logic section. Selected combinations of bits from both digits of I-Save may be the basis for simple 2-way branches as well.[2]

The General Indicators (GI) can be set to any values by a block transfer from the FBUS, then read back into the Arithmetic/Logic section via the IBUS. The GI's can also be used as independent bits to control 2-way branching.[3]

The W-Counter is a 4-bit down-counter that can be initialized from the FBUS, then read back into the Arithmetic/Logic section, or tested against zero and decremented.[4]

References:

[1]Topics 8.3 and 8.4
[2]Topic 8.5
[3]Topic 7.2
[4]Topic 7.2

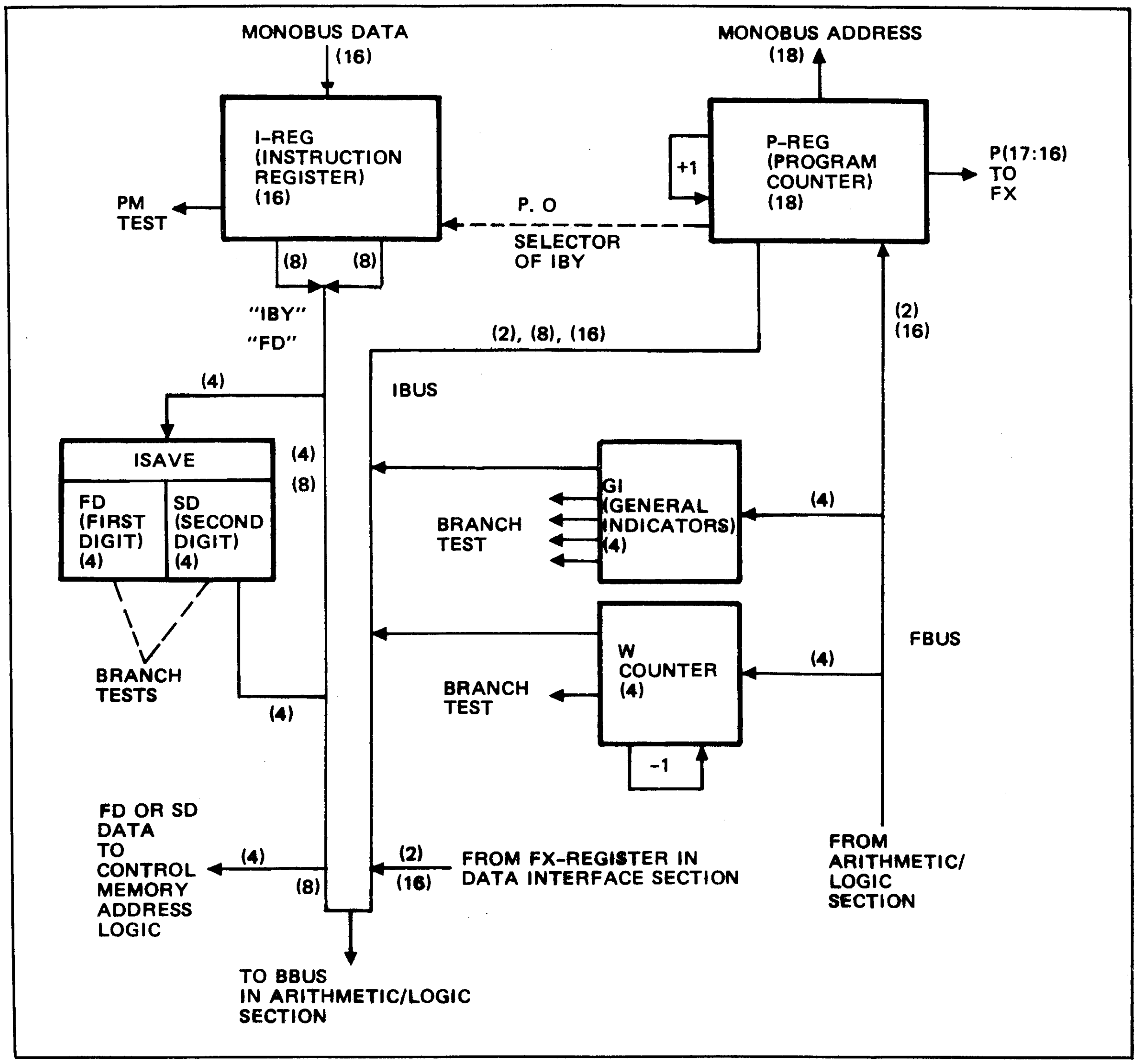


Figure A. The User-Level Instruction Fetch Section

2.4 THE DATA INTERFACE SECTION

The Data Interface section is the second path of the communication between the MONOBUS and the CPU; this communication path is of lower priority than the User-Level Instruction Fetch interface. Whereas the Instruction Fetch section is used only to direct words from the program stream into the CPU, the Data Interface is bidirectional and is used for moving to and from the CPU.

The Data Interface section is illustrated in Figure A. This logic section consists of 2 main registers (the M-Register and the D-Register), a set of input D-Gates, 2 auxiliary registers, and 2 1-bit MONOBUS Write Flags. The CPU side of the Data Interface is almost entirely the FBUS (the CPU's 16-bit central data bus). There is a connection from the upper 2 bits of the P-Register (P.17:16) into the FX-Register to aid in 18-bit addressing within a 16-bit system. P.17:16 is also referred to as PX.

When loaded with an 18-bit address, the M-Register initiates a request for MONOBUS control. The address may point to a word or byte in the Main User-Level Memory, to a word in the Control Memory, or to a peripheral controller. When the MONOBUS responds, a read or write action occurs that depends on the loading of the D-Register with a word or byte to be transmitted. If the full D-Register (or either byte of the D-Register) is loaded before the conclusion of the current MONOBUS access, one or both of the MONOBUS Write Flags are turned on. The contents of the D-Register are then read into the CPU from the MONOBUS. Without Write Flags on, data are read into the CPU from the MONOBUS through the D-Gates, onto the FBUS.

Though the D-Register, D-Gates, and FBUS are all 16-bits wide, adequate provision is made for handling 8-bit data. Provision is also made to control the release of MONOBUS, so that a fast Read/Write cycle can be executed without losing MONOBUS Control.

Auxiliary registers FX and SX assist in extending the normal 16-bit data orientation of the system to permit 18-bit addressing of the MONOBUS. Connections from the FBUS, PX, SX, and Local Memory addressing logic provide data and control for establishing the upper 2-bits of the MONOBUS address. This is accomplished by explicit microinstructions, or as a natural byproduct of other actvity.[1]

Reference:

[1]Topic 4.5

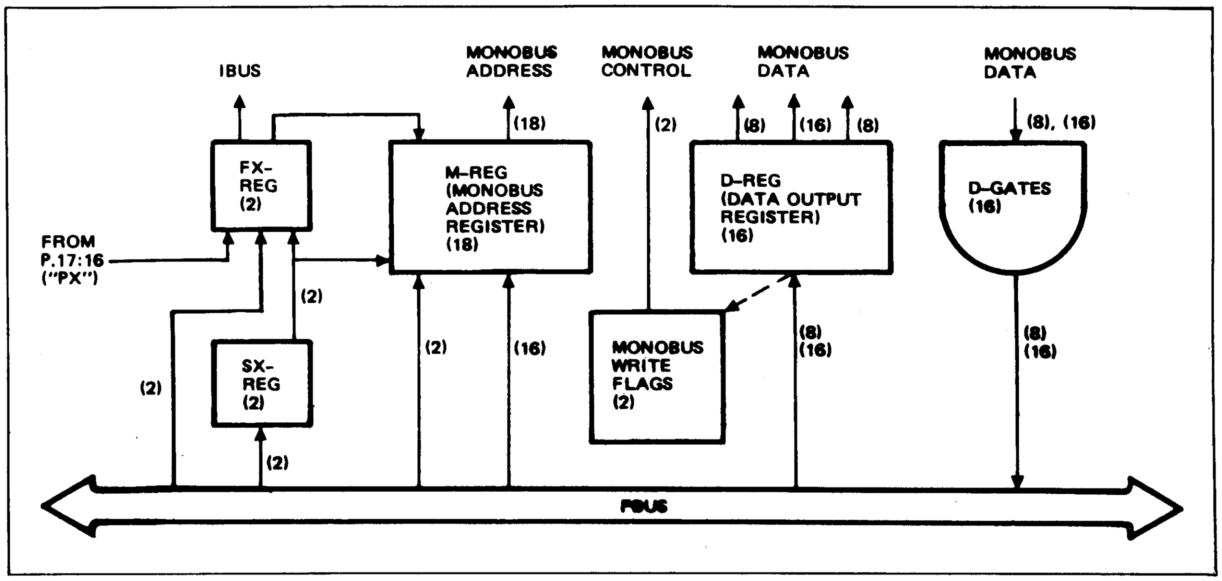


Figure A. The Data Interface Section

2.5 THE ARITHMETIC/LOGIC SECTION

The Arithmetic/Logic section of the 3200 CPU consists of three 16-bit Working Registers (X, Y, and Z), the ALU proper, Arithmetic Flags, and the Target Flags (TFG) Register. All of these entities are interconnected through the ABUS, the BBUS, and the FBUS.

The Arithmetic/Logic section is illustrated in Figure A. The BBUS receives outside inputs via the IBUS (from the User-Level Instruction Fetch section) and from the C-Register (in the Micro-Control section) via the EBUS. Selected bits from the ALU are transmitted to the Branching Test Logic in the Micro-Control section, and to the Local Memory Addressing Logic.

Registers X, Y, and Z provide a variety of single and combinatorial shift capabilities, with bit fill selectable by the programmer. Shifting is carried out as a separate working register operation, not as part of an ALU operation.[1]

The ALU provides addition and subtraction (with and without use of an input carry or borrow), data transfer, complementing, AND, OR, and EXCLUSIVE-OR operations. Data widths are 8 or 16-bits and are determined by the output destination. ALU output is always via the FBUS to the selected destination(s).[2]

Input to the ALU is via the ABUS and/or BBUS. The ABUS is fed by the Y- or Z-Register. The BBUS is fed by the X- or Y-Register, and the EBUS or IBUS. Restrictions which apply to the combination of sources, data widths, etc., are discussed in the referenced topics.[3]

ALU number representation and arithmetic are in 2's complement form. The Arithmetic Flags, when specifically enabled, are set after each ALU operation and reflect the Carry/Borrow, Overflow, Negative, and Zero conditions of the ALU output. The left-to-right order of the target flags (C, O, N, Z) are worth remembering.

Target Flags (TFG) is a 4-bit register than can be set to the same four condition values as the Arithmetic Flags. Alternately, TFG can be set from the lowest 4 bits of the Z-Register (Z.3:0 = Z4 = ZA).[4]

References:

[1]Topic 5.6
[2]Topics 4.1 and 4.2
[3]Topics 5.1, 5.2, and 5.3
[4]Topic 5.5

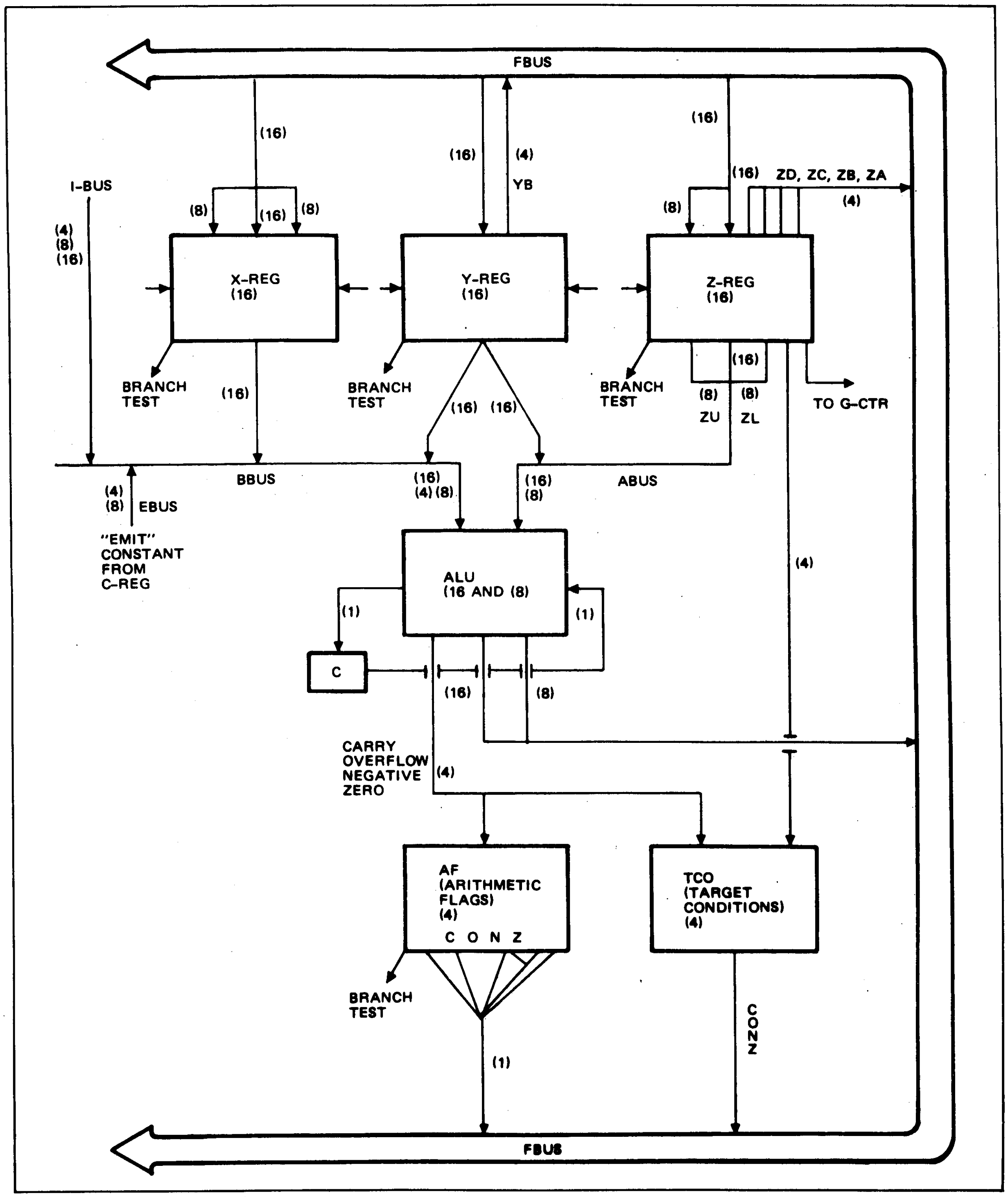


Figure A. The Arithmetic/Logic Section

2.6 THE LOCAL MEMORY AND AUXILIARIES SECTION

This section of the 3200 CPU contains the Local (Fast) Memory and several auxiliary sub-sections. The Local Memory consists of 32 high-speed, 16-bit registers.

The structure of the Local Memory and Auxiliaries section is shown in Figure A. This logic section contains the following sub-sections:

- Local Memory (LM)
- Single-Bit Generator (SB)
- LM and SB Addressing Logic
- G-Counter
- T-Counter
- N-Stats

Communication with the Local Memory (LM) and Auxiliaries is via the FBUS, with the C- and Z-Register having some direct inputs. The LM and SB Addressing Logic provides control signals effecting the upper 2 bits of M-Register (for MONOBUS addressing).

The Single-Bit Generator (SB) provides a 16-bit signal on the FBUS equal to any one of the 16 possible powers of 2 which are expressable in 16 bits (the selected power of 2 corresponding to the specified LM and SB address in the range 0:15). Each signal consists of one "1", bit, hence the name.

The LM consists of 32 high-speed, 16-bit registers arranged in 2 banks of 16 registers each, for addressing purposes. These are designated as "Primary" and "Secondary" Local Memory. Ten of the 16 Primary LM Registers have special hardwired functions as well as serving as general-purpose high-speed local storages. LM0:LM3 serve as part of the stack head. LM10:LM15, when addressed, effect the 2 most significant bits of the MONOBUS address (M.17:16). See referenced topics for further details.[1]

The LM and SB addressing Logic provides capabilities for addressing any one of the SB Generators, or any one of the 32 Local Memory registers as a source or destination (but not both in the same microcycle). It also provides for a variety of side-effects that can be specified to occur in a Local Memory access microcycle.

The G-Counter is a 4-bit cyclic up-counter that can be initialized from the lowest 4 bits of the Z-Register or Emit Field. The G-Counter is then used for LM and SB addressing and optionally incremented as a side effect.[2]

The T-Counter is a 2-bit cyclic bidirectional counter used by the 32/S emulation in stack head manipulation. The N-Stats is a 5-bit shift register used to represent the current stack head structure. Detailed stack head processing is discussed in the referenced topics.[3] For now, it is enough to recognize that stack head manipulation provided in the 3200 localizes the Top-of-Stack in the Y-Register, with 0 to 4 additional words in LM0:LM3 forming the rest of the stack head. The remainder of the stack is in the Main Memory. The Stack Handling Logic provides the integration of N-Stats, T-Counter, and LM and SB Addressing, that makes efficient stack handling possible.

References:

[1]Topics 4.2 and 4.3
[2]Topic 4.3
[3]Topics 9.1 through 9.5

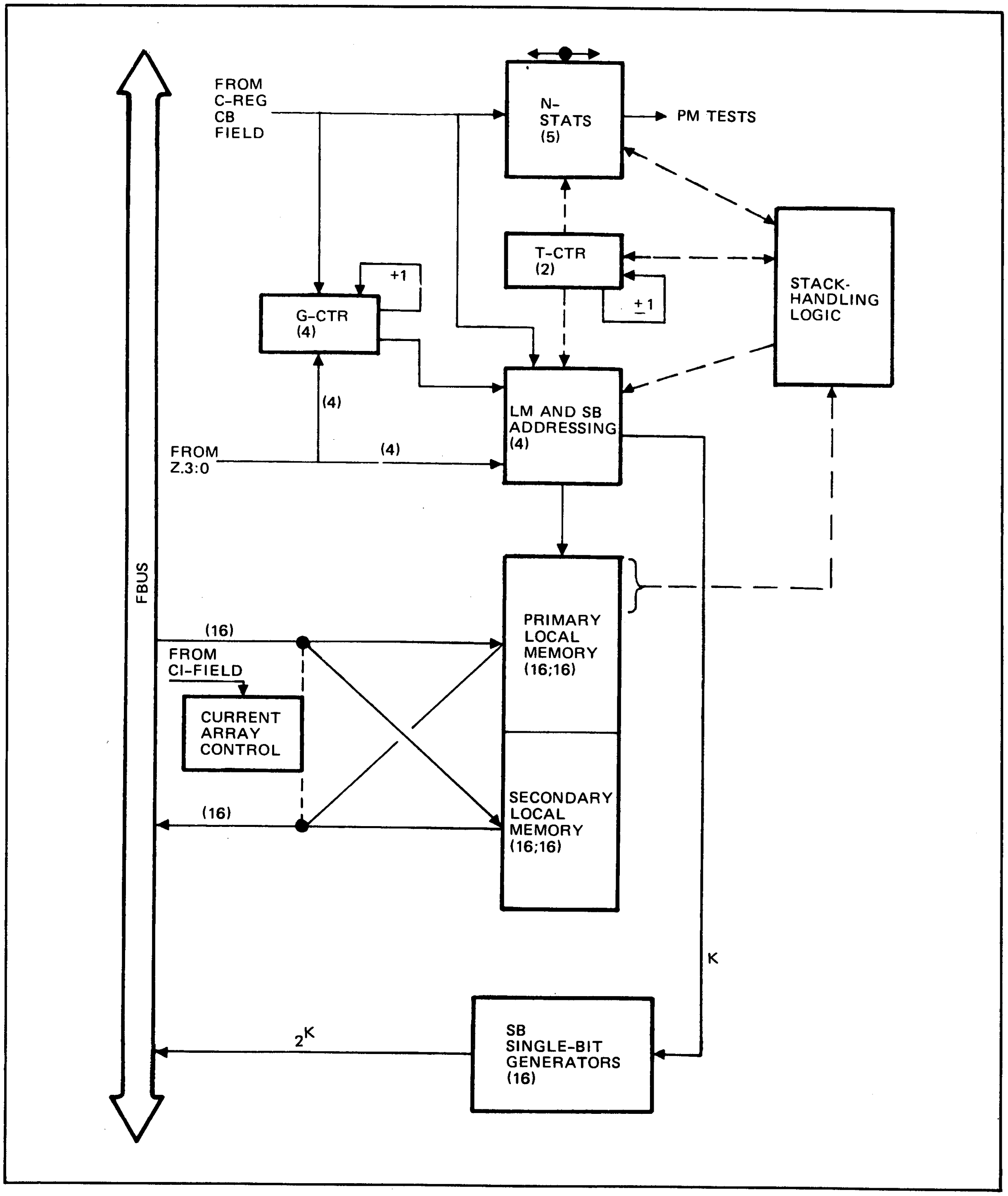


Figure A. The Local Memory and Auxiliaries Section

2.7 THE FRONT PANEL SECTION

The Front Panel section of the CPU is unique in that a large fraction of the available front-panel display and control capabilities is subject to microprogram control. It can be tailored to the particular firmware set (and user-level instruction set) that is being implemented.

The front panel normally supplied for users interested in extensive micro-programming development is adapted for operation with the 32/S firmware set. Every panel contains switches for the following basic functions: OFF, STOP, LOCK, RUN, HOLD, LOAD, and INT. The full maintenance panels used in micro-programming development also contain 7 Status Indicators, 11 Control Switches, 18 Data Bit Entry Switches, 18 Display Selector Switches, an 18-Light Address Display, and a 16-Light Data Display.

Due to flexible firmware control of the Front Panel logic, the displays and controls are not limited in their effect to the CPU, but may also interact with the remainder of the system via the MONOBUS.

The complete maintenance panel is directly addressable on the MONOBUS, to permit the necessary microprogram to read the Data Switches and Panel Status Word, and to output to the Display Registers.

The basic front panel is illustrated in Figure A. Figure B shows the complete maintenance front panel of the 32/S emulation.

Details regarding the Front Panel are covered in a separate section of this manual.[1]

Reference:
[1]Topics 11.1 through 11.6

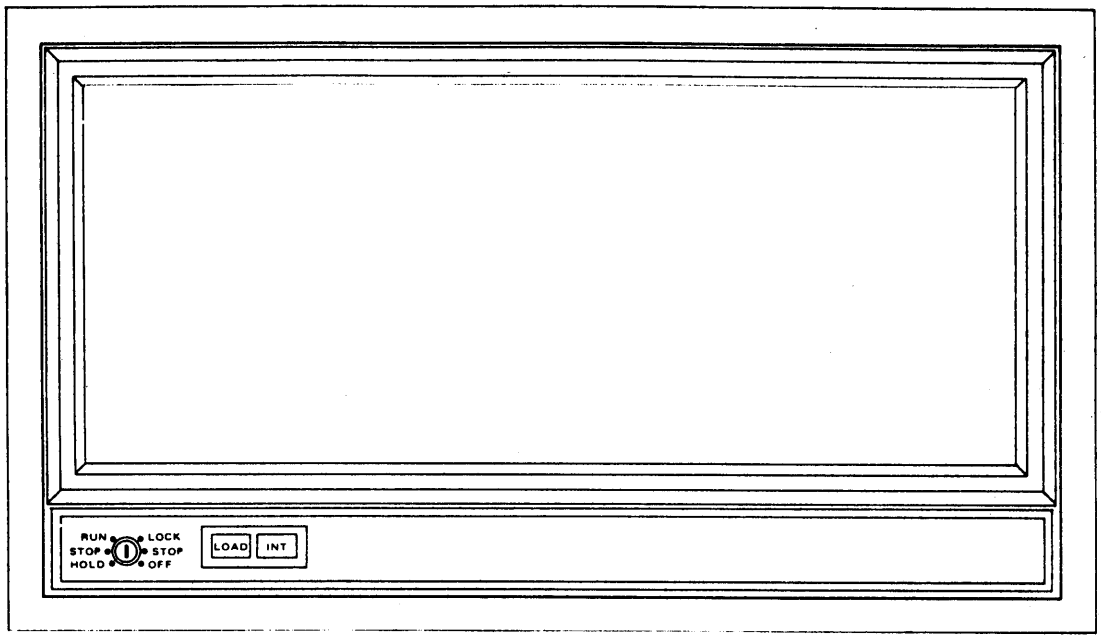


Figure A. Basic Panel

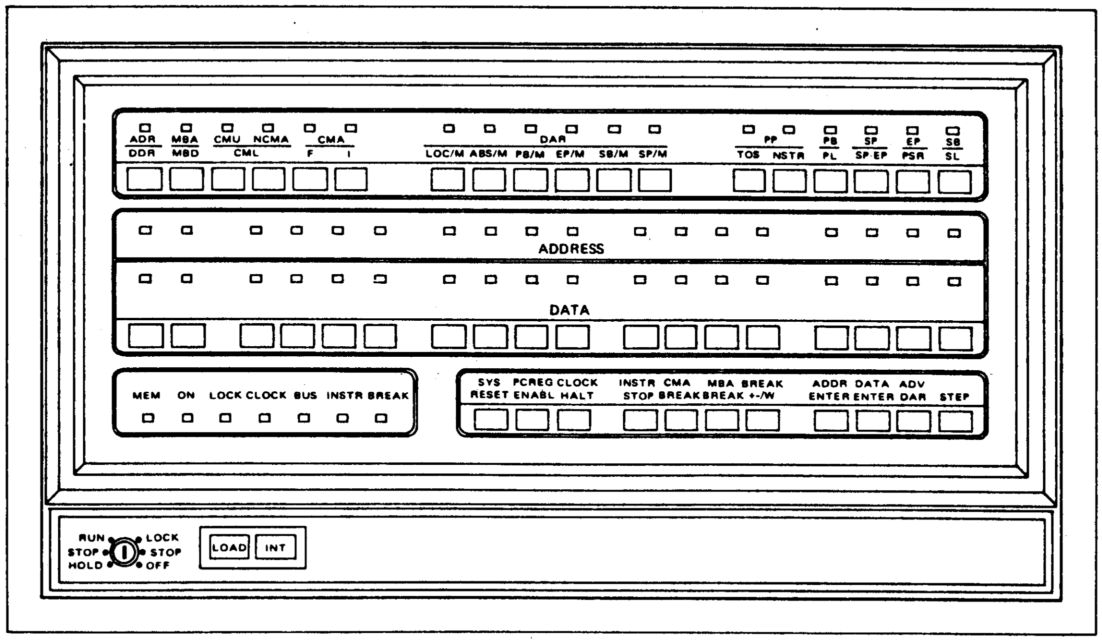


Figure B. Maintenance Panel

3 INTRODUCTION TO CAP32

3.1 MICROINSTRUCTION FORMAT

This topic describes the general format of the CAP32 microinstruction.

CAP32 source microprograms contain one statement per line: an executable microinstruction which will assemble into a line of object microcode, or a microassembler pseudoinstruction directing CAP32 to perform an evolution at microassembly time. Each CAP32 instruction contains up to four specific fields, as shown in Figure A.

The Label Field (LAF), which starts in column 1, is conventional in form. It may be empty, contain an asterisk denoting a comment line, or contain a symbol. (Many of the CAP32 pseudoinstructions require a symbol in LAF.) If the line contains an executable instruction, a symbol in LAF will permit the line to assume the destination of a normal jump or branch instruction.[1] Jumps or branches to unlabeled lines in CAP32 are restricted to $\pm$n from the microprogram counter (denoted by *+n or *-n). Jumps or branches to a line relative to a labeled line (of the form symbol +n) are not supported.

A symbol in CAP32 consists of 1 to 6 alphanumerics, beginning with a letter. Letters include the period symbol (.), and the 26 letters (A through Z). The alphanumerics consist of the letters and the decimal digits (0 through 9). Hence, a symbol in CAP32 may have a period inserted anywhere, and may include digits after the first character. For example: I5D, FD3, .FDX, and .001 are legitimate CAP32 symbols. 4FLUSH, SEGMENT, and $SAFE are illegal.

In a pseudoinstruction, the Operation Field (OPF) consists of a keyword identifying the particular pseudo operation called for. Otherwise, the instruction is executable and the OPF consists of 1 or 2 subfields (OPF1 and OPF2) separated by a comma (with no space). OPF2 may in turn contain 2 subfields separated by a comma.

OPF1 specifies in part the principal operation to take place during the 135-nanosecond microcycle in which the current microinstruction is executed. Other fields of the statement must also be interpreted to complete the specification of the action. The principal operation specified by OPF1 may be:

- A shift in one or more working registers (X, Y, or Z).
- An arithmetic or logic expression evaluation.
- A data transfer from the D-Gates (hence the MONOBUS), the Local Memory, or one of the other sources feeding the FBUS.

When occurring, OPF2 specifies a concurrent secondary action involving the stack handling facilities or the G-Counter.[2]

The Result Field (REF) consists of 1 to 3 subfields (REF1, REF2, and REF3). REF1 completes the specification of the principal operation partially specified by OPF1. The allowed values of REF1 are heavily dependent on OPF1. For shift operations, REF1 specifies the working register(s) to be shifted. For data transfers and arithmetic or logic operations, REF1 specifies the destinations receiving the FBUS signal from the Arithmetic/ Logic section, or the named signal source. Allowable destinations, operations, and sources are discussed in detail in the referenced topics.[3]

When occurring, REF2 specifies the label of the next microinstruction to be executed. When making a conditional branch, REF2 also specifies the condition to be tested and the alternative destinations. If REF2 does not appear, CAP32 selects the next sequential microinstruction.

When occurring, REF3 specifies desired concurrent side effects.[4] REF3 also permits direct encoding of the C-Register bits, free from CAP32 micro-assembler constraint, but without its aid and protection.

References:

[1]Topics 7.1 through 7.6
[2]Topic 4.3 and Topics 9.1 through 9.5
[3]Topics 4.1 through 4.6
[4]Topics 4.3, 5.5, and 8.4

Field	Definition
Label Field (LAF)	*From col 1 to the first blank; empty if col 1 is blank.*
Operation Field (OPF)	*From next non-blank to first subsequent blank*
Result Field (REF)	*From next non-blank to first subsequent blank.*
Comment Field (COF)	*From next non-blank, if any, to end of line.*

Figure A. CAP32 Instruction Fields

3.2 C-REGISTER FIELDS

This topic describes the relationship of the CAP32 microinstruction fields to the related Control (C) Register fields.

The C-Register fields are shown in Figure A by bit position, field name, field function, and the CAP32 microinstruction field effecting them. The C-Register is 32 bits wide, numbered from 31 at the most significant end to 0 at the least significant end. The C-Register contains bits C.31:0, with nine fields identified (CJ, CI, CG, CF, CE, CD, CC, CB, and CA). All of the fields are four bits wide (on Hex digit boundaries), except CE and CD which share a single HEX digit (CE is one bit wide and CD is three bits wide). Since CD is based on a natural Hex digit boundary at its right end, its range is valued at 0:F. CE is to the right of Hex digit boundary, so when interpreting a Hex display of the microcode, CE will add either 0 or 8 to the Hex digit containing CE and CD. Hence the CE field has a value of either 0 or 8.

When a given field has a specified value, such as field "Cx" having value "v", the condition "Cxv" is said to exist. For example, condition CIA occurs when field CI has Hex value A.

Usage of the CB field, thus probably blocking it for other usage during the current microcommand cycle, is referred to as "CB-Constraint".

Knowledge of the C fields is required to understand constraints that the machine architecture places on source language statements, and the resulting performance. Each CAP32 instruction is assembled into one Control Word. A "loading" of the C-Register occurs at the time the Control Word is executed as a microinstruction. The C-Register appears on the microprogram listing as the object microcode in Hexadecimal notation.[1]

Normally, the CAP32 Microassembler determines the value of each C-Register field as the programmer desires (expressed in CAP32), handling many of the details automatically. For example, the choice of Control Memory addressing mode (field CD) is established by CAP32 in most cases, and the address is determined by CAP32 through use of a memory allocation routine. The routine seeks to completely fill up the available memory, while meeting the restrictions inherent in the 3200 branching structure.[2]

If the programmer desires, he may force CAP32 to establish each microinstruction precisely as desired, using one or more expressions of the form "Cfield=value" in subfield REF3 of the CAP32 instruction. For example, "CB=14,CA=2" would force the CB field to value Hex E, and the CA field to value 2. These forced specifications supersede those values the remainder of the CAP32 statement would have produced. This procedure has inherent hazards.

References:

[1]Topic 6.2
[2]Topics 7.1 through 7.6

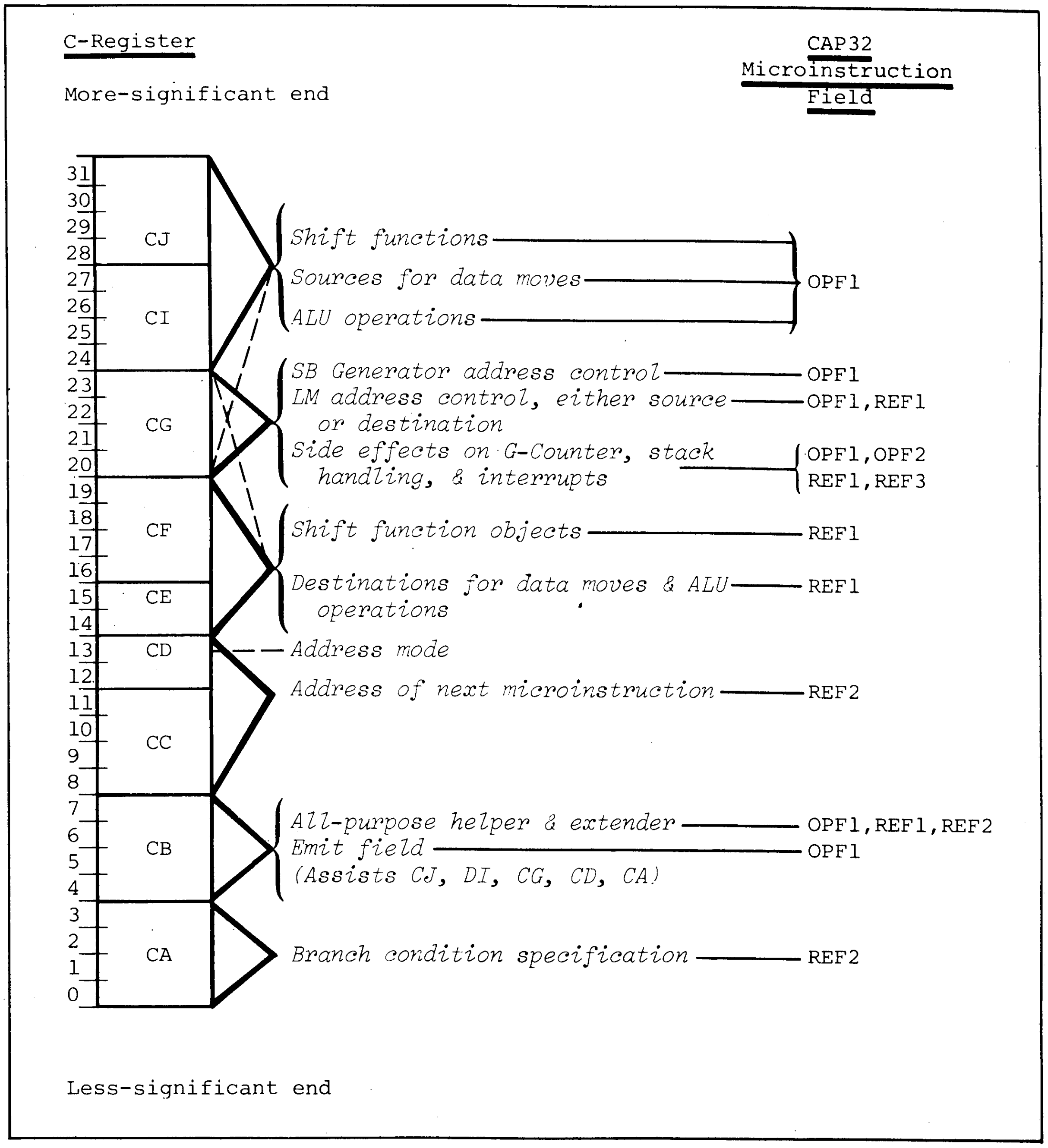

C-Register
More-significant end
CAP32 Microinstruction Field
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
CJ
CI
CG
CF
CE
CD
CC
CB
CA
Shift functions
Sources for data moves
ALU operations
OPF1
SB Generator address control
OPF1
LM address control, either source or destination
OPF1,REF1
Side effects on G-Counter, stack handling, & interrupts
OPF1,OPF2
REF1,REF3
Shift function objects
REF1
Destinations for data moves & ALU operations
REF1
Address mode
Address of next microinstruction
REF2
All-purpose helper & extender
OPF1,REF1,REF2
Emit field
OPF1
(Assists CJ, DI, CG, CD, CA)
Branch condition specification
REF2
Less-significant end

Figure A. Field Structure of the C-Register with Relation to Fields and Subfields of the CAP32 Microinstruction

4 FBUS DESTINATIONS AND SOURCES

4.1 DIRECT FBUS DESTINATIONS

FBUS destinations are specified in the REF1 subfield of the CAP32 microinstruction; they are separated into two categories: direct FBUS destinations and Local Memory (LM) FBUS destinations.

Direct FBUS destinations are specified by using one code from Reference Table A-2 (e.g., X, or Y:Z, or ZU:F). Local Memory (LM) FBUS destinations are specified by using one code from Reference Table A-3 (e.g., LM(G,IG)). Two destination codes may appear in the same microinstruction REF1 subfield if one is from each category (e.g., X:LM(8), or DB:LM(T,DT), or LM(G):Y:Z).

The remainder of this topic is devoted to a discussion of the direct FBUS destination codes. The user should refer to Reference Table A-2 (in Appendix A) and to Figure A (in this topic).

The null code (.) is used in REF1 if no destination is required for the FBUS signal (i.e., the microcommand is a conditional branch, without data movement).

REF1 code F is an input selector for the Target Register, and an enabling specification for the Arithmetic Flags (AF).[1] Here the AF's are to be set to reflect the Carry, Overflow, Negative, and Zero status of the ALU output. (As such, F is not a true FBUS destination code but is listed with them.)

REF1 code SBR specifies selected bits of the S-Register to store the L-Register contents. The L-Register holds the Control Memory address of the current microcommand.[2] SBR is distinguished from REF1 code S which directs setting of the full S-Register from the FBUS.

REF1 codes X, Y, and Z specify the most common FBUS destinations (Working Registers X, Y, and Z), which in turn may drive the ALU, provide for shifts, or be tested.[3]

REF1 codes XU, XL, and ZU specify transfer of half the FBUS to half of a Working Register. Similarly, code DB specifies output of a single byte (half the FBUS) to the MONOBUS Data Register D. Each of these codes directs copying of the ALU output low-order byte (ALU.7:0) into the ALU output high-order byte (ALU.15:8). Thus if the FBUS source is the ALU, a shift of the ALU output lower-byte into both halves of the FBUS results. Note that if the ALU is not the FBUS source, no shift occurs on the FBUS.

REF1 codes D, DB, MR, MW, and P impact the MONOBUS, providing memory and I/O device access.[4] REF1 codes P and PX control the user-level Program Counter. The user is cautioned that code P loads P.15:0 from the FBUS and also initiates MONOBUS action; thus, code PX must be used first if a major jump in memory or I/O address space is planned. This direct control of the user-level Program Counter is only for jumps and unusual situations. Normal advance of the counter to the next sequential instruction is provided automatically.[5]

REF1 codes FXA, and FXB, and SX enable the selection of the memory bank for the next data access to the main memory or I/O devices. Final selection of memory location or I/O address is through REF1 codes MR or MW.[6] REF1 codes

W and GI provide for setting of the W Counter and General Indicator (GI) bits. Finally, code Q provides for setting of the Interrupt Masks.[7]

References:

[1]Topics 5.4 and 5.5
[2]Topic 2.2
[3]Topics 4.3, 5.1, 5.3, and 5.6
[4]Topics 4.5 and 4.6
[5]Topic 8.3
[6]Topic 4.5
[7]Topic 8.4

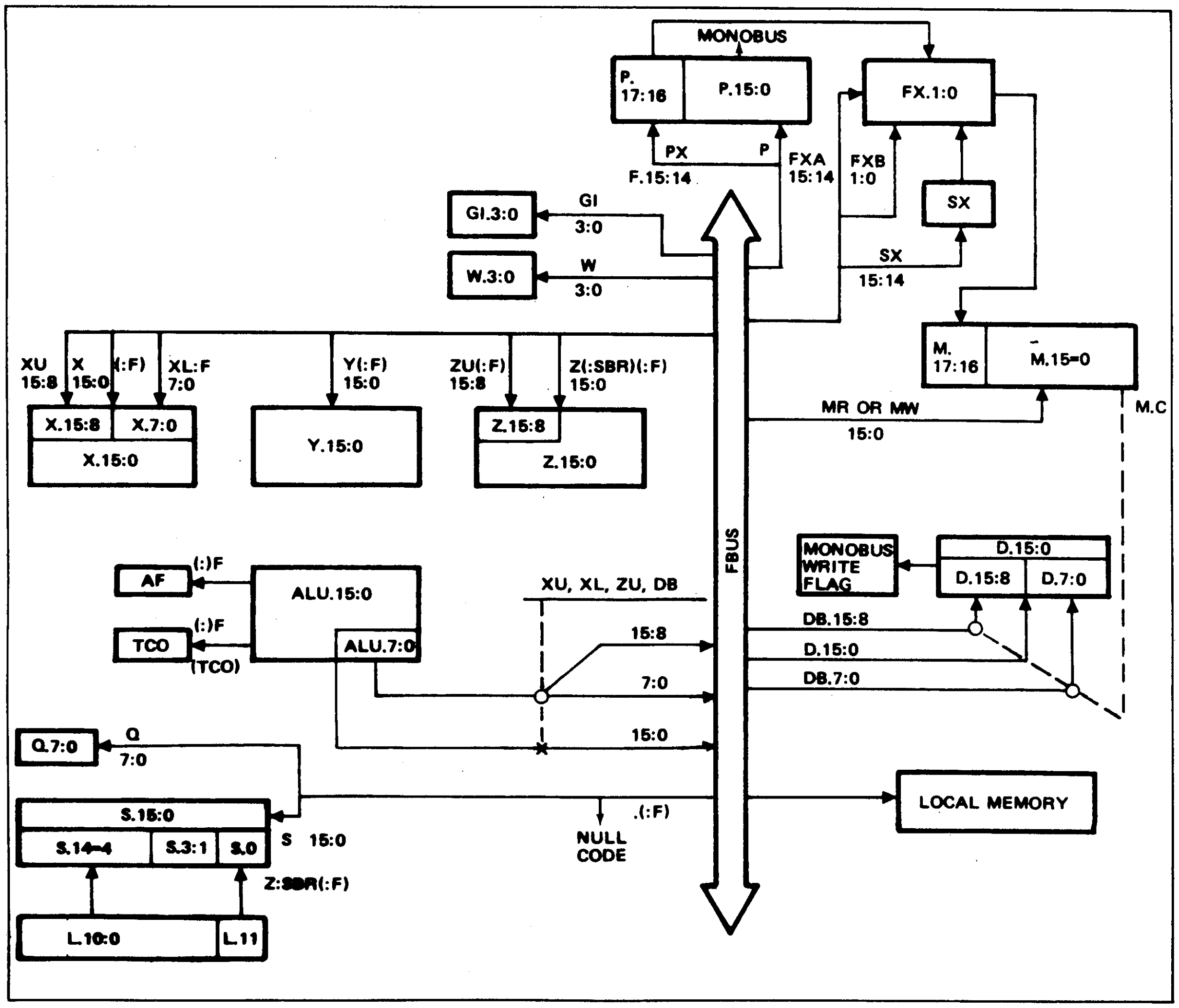


Figure A. Effects of Direct FBUS Destination Codes

4.2 LOCAL MEMORY (LM) FBUS DESTINATIONS

The second category of FBUS destination is Local Memory (LM). LM destinations are specified by using one code from Reference Table A-3.

LM consists of 32 high-speed registers in two arrays, denoted Primary and Secondary.[1] Array selection (Primary and Secondary) is accomplished when LM is used as an FBUS signal source.[2] LM destination codes cannot modify the current array selection mode established by a previous LM source code. Thus an LM destination is always in the "Current" LM array, denoted CLM.

The syntax of an LM destination code used in subfield REF1 is either of the following:

```
     LM(index)

     LM(index,side-effect)
```

where "index" has a value in the range 0:15, and where "side-effect" specifies a unique concurrent side-effect, such as setting or incrementing the G-Counter.

The G-Counter is a 4-bit register specifically connnected an an LM index. When the G-Counter is used as an index, incrementation of the register can be a specified side-effect (i.e., using code IF), thus providing a convenient means of "stepping" through either array of 16 LM registers. Incrementation occurs after use of the G-Counter as an index. The G-Counter is initially set as the side-effect load G (LF) when one of the other indexing methods is used.

Index code Z4 allows direct use of data to index into the LM. The data item is first loaded into Z.3:0, then LM destination code LM(Z4) or LM(Z4,LG) is used, where LG loads the G-Counter with the current value Z4.

LM can be indexed directly through a microcode literal value, denoted by rn in Table A-3. In the CAP32 program form a symbol previously defined by a REGNAM pseudoinstruction may be used for rn. In the final machine Control Word, the value of rn is contained in the CB field. Hence, where possible, G or Z4 LM indexing should be used rather than rn indexing, to minimize the use of the all-purpose helper field CB. This also reduces the probability of conflict between the indexing and another control function of the same microcommand. Often, LM(rn,LG) will be used when an LM register is written (a few microcommands later), thus eliminating the CB-Constraint.

The fourth method of LM indexing is via the T-Counter (codes LM(U,IT) or (LM(T,DT)). This method is totally involved with the specific facilities for stack handling.[3]

An important restriction to keep in mind is that LM may be used as a source or as a destination during one microcycle, but not as *both*. Word transfers from one LM register to another (with or without modification) requires cycling the word through Working Register X, Y, or Z, and the ALU, which takes two microcommands.

Finally, anticipating subsequent topic coverage, the user should note that certain LM registers have special properties. Registers 0:3 of one array are used in the stack head, addressable by the T-Counter.[4] Registers 10:11 have special influence on MONOBUS addressing, making them particularly appropriate as Program Counters. Registers 12:15 have special influence on MONOBUS addressing, making them particularly appropriate as Data Stack Environment Pointers.

References:

[1]Topic 2.6
[2]Topic 4.3
[3]Topics 9.1 through 9.5
[4]Topic 9.2

4.3 LOCAL MEMORY (LM) AND SINGLE-BIT GENERATOR ARRAYS AS FBUS SOURCES

FBUS sources are specified in the OPF1 subfield of the CAP32 microinstruction. The local array FBUS sources consist of the Local Memory (LM) and the Single-Bit Generator arrays.

A Single-Bit Generator produces a full word of 16 bits; this word contains a single 1-bit, with the rest 0's. If the code SB(index) or SB(index,side-effect) is used in the OPF1 subfield of a CAP32 statement, the value 2**index is placed on the FBUS. That is, bit "FBUS.index" is set to 1 and all other FBUS bits are set to 0. The allowable syntax and sematics for "index" are shown in part 2 of Reference Table A-4. For example, if OPF1=SB(5), then at execution time FBUS=0020_{16}=100000_2. Similarly, if G=15 and OPF1=SB(G), then at execution time FBUS=8000_{16}.

Specification of an LM register as a source is complicated by the existence of the two LM arrays (Primary and Secondary) each containing 16 registers. Two of the OPF1 codes, LM and LMS, establish a definite (permanent) LM mode. The LM code establishes the Primary LM array as the Current Local Memory (CLM). LMS establishes the Secondary array as the CLM. The other two OPF1 codes, LMC and LMN, leave the established CLM as is: LMC prescribes use of the CLM, while LMN prescribes use of the Noncurrent Local Memory (NLM).

The codes used to index within the selected LM array are identical to those described for LM FBUS destinations[1], except that they are used in subfield OPF1 instead of REF1. Also, codes U and T may be used with or without side-effects, as desired. Codes U and T are discussed in conjunction with Stack Handling.[2] Note again that LM may be either a source or destination in any microcommand, but not both, since C-Field CG is used for gating both separately.

Working registers X, Y, and Z may be used as FBUS sources, by using codes X, Y, or Z (respectively) in subfield OPF1. The signals from X, Y, or Z come through the ALU, as discussed in the referenced topics.[3]

Codes LG and IG, used for independent of local array source or destination codes, also effect the G-Counter. (Reference Table A-5 provides details.) The G-Counter can be set by using code Z4,LG or code rn,LG as subfield OPF2 of a CAP32 statement. In this case Z4=Z.3:0 and rn denotes a register name defined by REGNAM, or a decimal number in the range 0:15. The G-Counter can be incremented by using code IG as subfield REF3 of a CAP32 statement, inserting a null REF2 subfield if necessary.

Sample usage of the source codes described above are illustrated in Figure A.

References:

[1]Topic 4.2
[2]Topics 9.1 through 9.5
[3]Topics 5.1 through 5.6

CAP32 Microinstruction		Action
SB(0)	Y	*Loads Register Y with value 1.*
SB(1)	Z	*Loads Register Z with value 2.*
LMC(7,LG)	Z	*Loads Register Z with the value now stored in Register 7 of the currently active LM array, and sets the G-Counter to 7.*
LMS(14)	Y:Z	*Selects the secondary LM array to be currently active, and loads registers Y and Z with the contents of secondary LM register 14.*
LMN(G,IG)	D	*With the primary LM array currently active (and keeping it that way), this instruction loads the MONOBUS D-Register with the contents of the secondary LM array indexed by the G-Counter. The G-Counter is then incremented.*

Figure A. Sample Usage of LM and Single-Bit Generator Arrays as FBUS Sources.

4.4 OTHER DIRECT FBUS SOURCES

In addition to the local array FBUS sources[1], seventeen other sources drive the FBUS directly, bypassing the ALU. These sources vary from 16 bits to 1 bit in active width, and provide 0's in unused bit positions. Each is specified by use of the proper code in subfield OPF1.

Reference Table A-6 summarizes these additional direct FBUS sources.

OPF1 codes D or DB cause the current MONOBUS Data Word (MDW), or possibly just one byte of that word, to be placed on the FBUS.[2]

OPF1 code TFG loads the 4 low-order bits of the FBUS from the 4-bit Target Flags Register (TFG). TFG is a register which reflects the conditions of same previous ALU operation.[3] Each bit loaded into the F-BUS reflects a specific target flag: 3, carry; 2, overflow; 1, negative; 0, zero.

OPF1 code SWIT allows bits representing 4 hardware test switches in the 4 low-order bit positions of the FBUS. These internal cabinet switches are for programmed maintenance use and are not operator-available.

Four OPF1 codes (ZD, ZC, ZB, and ZA) load FBUS.3:0, the 4 low-order bits of the FBUS, with the 4 Hex digits stored in the Z-Register. ZD indicates the most significant Hex digit; ZA indicates the least. These source codes are convenient for many decoding operations. OPF1 code YB loads FBUS.3:0 with Hex digit B from Y-Register Y.7:4.

Finally, eight OPF1 codes of the form F(f) bring to the low-order bit of the FBUS (FBUS.0) selected 1-bit flags derived logically from the 4 micro-level hardware Arithmetic Flags (AF).[4] Six of the eight codes provide all of the normal comparisons of the ALU result against zero (Zero, Non-Zero, Negative, Non-Negative, Positive, Non-Positive). The final two codes report Carry and Overflow conditions. A CB-Constraint results.

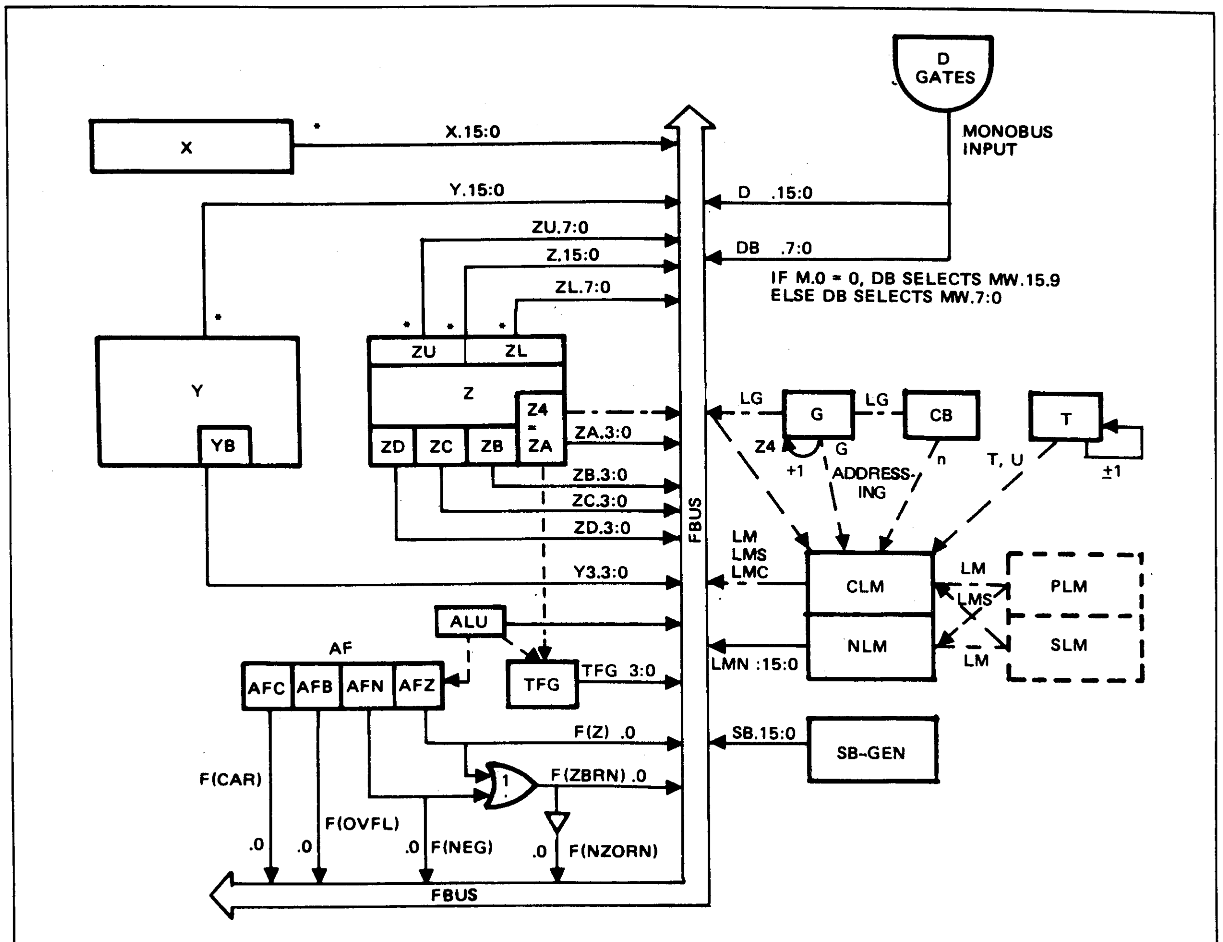
All direct sources listed in Table A-6 drive the FBUS directly (not through the ALU); hence, they do not respond to the Byte-type destination codes[5] so as to copy their bit pattern from the lower byte to the upper byte of the FBUS. Thus, destination codes XU, ZU, or DB will not cause any transfer of these 4-bit or 1-bit source signals into the upper byte of X, Z, or D (respectively).

Source codes D or DB must not be used in the same microcommand as destination codes D or DB. If such use is attempted, the MONOBUS Controller may hang in permanent error.

The entire set of direct FBUS inputs is shown graphically in Figure A.

References:

[1]Topic 4.3	[3]Topic 5.5	[5]Topic 4.2
[2]Topic 4.5	[4]Topic 5.4	



*These FBUS inputs are in fact through the ALU (see Topic 5.1).

Figure A. Direct FBUS Inputs and Selected ALU FBUS Inputs

4.5 MONOBUS DATA ACCESS

The three modes of MONOBUS data access are: Read, Read-Rewrite, and Write. All three involve data transfers between the MONOBUS and the FBUS, using the D-Register for data written to the MONOBUS from the FBUS, and using D-Gates to input (read) a MONOBUS Data Word to the FBUS.

MONOBUS access begins by establishing a MONOBUS address in the 18-bit Monobus Address Register (M). The upper 2-bit segment (M.17:16) should be planned first; these bits are set as a side-effect of loading the lower 16 bits of M from the FBUS. M.17:16 is set from the Local 2-bit FX Register unless the lower 16 bits of M come from the Local Memory (LM) Registers, specified by CB indexing in the range 10:15. If the CB type index specification of LM source indexing is 10 or 11, then M.17:16 is set from the upper 2 Program Counter bits (P.17:16), thus setting FX. In the CB type index is 12,13,14, or 15, then M.17:16 is set from the two-bit 5X Register, thus setting FX. With the setting of the upper 2 MONOBUS address bits planned, a microcommand is executed transferring the lower 16 bits from their source to M, via the FBUS. The microcommand has the basic syntax:

```
     address-source      MR
or
     address-source      MW
```

REF1 code MR is used for Read access; MW is used for either Write access or Read-Rewrite access. MR and MW do not actually establish data flow direction across the MONOBUS/FBUS interface. Rather, they specify conditions under which the MONOBUS will be released after use. Code MR specifies immediate release of the MONOBUS after any access. Code MW specifies the MONOBUS is to be held until after a Write access is completed. Sequence 'source MR, source D' does not work.

The actual data flow direction between FBUS and MONOBUS is set by the pair of MONOBUS Write Flags associated with the two bytes of the D-Register. The data flow direction is MONOBUS to FBUS (i.e., read) unless one or both Write Flags are set. This occurs when destination code D or DB is used in subfield REF1, shown in Reference Table A-2. The Write Flags are cleared at the end of every MONOBUS Write access. Hence, if D or DB is used in REF1, then a Write occurs; otherwise a Read access occurs and D or DB should be used in OPF1 to take the data from the D-Gates and place then on the FBUS.

Summaries of the three different access modes are depicted in Figures A, B, and C.

The user should note that microcommand "data-source D" can precede microcommand "address-source MW" if desired. However, "data-source DB" must follow "address-source MW". The system will suspend operation if D or DB is used in subfield OPF1 without prior use of MR or MW in REF1 since previous access.

1. Use following microcommand:

   address-source MR

2. Allow other microcommands to intervene here; up to 2 cause no further delay.
3. Use one of the following microcommands:

   D data-destination
   DB data-destination

4. MONOBUS is released as soon as data is received on the FBUS.

Figure A. Read Access Procedure

1. Use following microcommand:

   address-source MW

2. Allow other microcommands to intervene here; up to 2 cause no further delay.
3. Use one of the following microcommands:

   D data-destination
   DB data-destination

4. MONOBUS is not released after Read; provide data revision microcode here but do not waste any microcycles as the MONOBUS is tied up.
5. Use one of the following microcommands:

   data-source D
   data-source DB

6. MONOBUS is released after Write.

Figure B. Read-Rewrite Access Procedure

1. Use following microcommand:

   address-source MW

2. Allow other microcommands to intervene here; up to 2 cause no further delay.
3. Use one of the following microcommands:

   data-source D
   data-source DB

4. The MONOBUS is released as soon as the WRITE is complete.

Figure C. Write Access Procedure

4.6 MONOBUS WORD AND BYTE TRANSFERS

The lowest order bit of a MONOBUS address (M.0) denotes the byte address within the current MONOBUS word. The high-order byte is indicated by an M.0 value of 0; a value of 1 indicates the low-order byte.

The incoming signal on the MONOBUS from the I/O or Memory Controller to the D-Gates includes both bytes. The high-order byte is always from the even address (with address-bit.0=0); the low-order byte is always from the odd address (with address-bit.0=1). Therefore, a MONOBUS access is always full-word oriented, up to the D-Gates connected to the FBUS.

If the FUBS source code DB is used in subfield OPF1, then bit M.0 is used to select one byte of the input word at the D-Gates for transmission to the FBUS. The other byte of the input word is lost insofar as that Read cycle is concerned. The MONOBUS is realeased after the selected byte is passed to the FBUS, and that word must be fetched again if the other byte is subsequently needed.

Behavior is slightly different on output. DB as a destination code in REF1 causes a data transfer from the FBUS into one byte of the D-Register as determined by M.0, and sets the corresponding Upper or Lower MONOBUS Write Flag.

CAUTION

Because M.0 selects the byte, the value of M must be set before D is loaded, or the wrong byte may be selected.

Only the selected byte is transmitted from D to the MONOBUS; the Controller receiving the byte also receives a control signal indicating it is just a single byte.

Since M cannot be set without initiating a MONOBUS access cycle, it is not possible to load the two bytes into D separately and then write them both out together as a MONOBUS word. DB is designed for byte-by-byte output only. Composition of separate bytes into an output word should be carried out in the Working Registers, then transmitted to the D-Register by use of output code D in subfield REF1.

Figure A presents a number of CAP32 examples illustrating the concepts introduced in this and the previous topic.

CAP32 Sequence		Action

LM(7)
LMS(5)
	MW
	D
		Writes SLM(5) in memory at FX//PLM(7); i.e., writes secondary LM register 5 to the MONOBUS address given by FX and primary LM register 7.

LM(4) DB	MR Y	*Reads from the MONOBUS address given in PLM(4) and places input data byte in Y.*
LM(5) D Y+1	MW Y D	*Increments the word at MONOBUS address given in PLM(5). Note that "increment" means Read, Add 1, Rewrite.*
X D Y D	MR Y MR Y	*Reads a full word from the address now specified in FX and X (i.e., FX//X). That word is then used as the 16 lower-order bits of a new address, still using FX as the upper two bits of the address. A data word is then read from the new address and placed in Y.*
LM(8) D ZU Y Z	MW Y:Z Z ZU D	*Reads the word addressed by FX//PLM(8) and then interchanges the two bytes of that word, rewriting it at the same address.*

Y
D,8,LG
ZU
ZL
	MR
	Z
	LM(G,IG)
	LM(G)
		Reads a word addressed by FX//Y. Places the upper byte of that word in the lower byte of CLM(8) (i.e., CLM(8).7=0). Places the lower byte of the word in the lower byte of CLM(9).

Y,4,LG
LM(G,IG)
LM(G)
X
	MW
	XU
	XL:F
	D
		Composes a data word from upper bytes of CLM(4) and CLM(5), and writes this word at address FX//Y.

Figure A. Sample MONOBUS Word and Byte Transfers

5 ALU SOURCES AND OPERATIONS

5.1 ALU SOURCES

The ALU is a major channel for signal transfer to the FBUS. The ALU transfers, complements, decrements, and provides two-operand logic and arithmetic functions of up to 14 different sources. These include numeric literals, Program Counter (P), Instruction Register (I), and the Working Registers (X, Y, and Z).

Figure A illustrates all ALU sources. The reader should also carefully study Reference Table A-7 in Appendix A.

If an FBUS destination is specified as byte-type (by use of XU, XL, ZU, or DB in REF1), the lower 8 bits of the ALU output are copied into the upper byte of the FBUS (15:8); the upper byte of the ALU output (ALU.15:8) is lost.

When any signal is transmitted through the ALU, the Arithmetic Flags and the Target Flags may be set at the option of the programmer.[1]

The syntax for transfers from any source via the ALU to the FBUS is to simply name the source in the OPF1 subfield. To transfer the one's complement of the source rather than the source signal itself, place a prime (single quote) after the source name in OPF1. Figure B gives a number of examples.

Any ALU source signal can be transmitted (or complemented and then transmitted) to the FBUS. However, for arithmetic and logical functions (or decrementing) the ALU input bus structure must be considered. Figure A emphasizes bus identification of each source (ABUS or BBUS). Only Working Register Y appears on both the ABUS and the BBUS. All other sources appear on one bus only, not both. An entire family of sources feed into the BBUS through its extension (IBUS). These include the Program Counter (P), its upper byte (PU), two mixed fields (SDX and GIW), and a group of I-type inputs.[2]

References:

[1]Topics 5.4 and 5.5
[2]Topics 8.3, 8.4, and 8.5

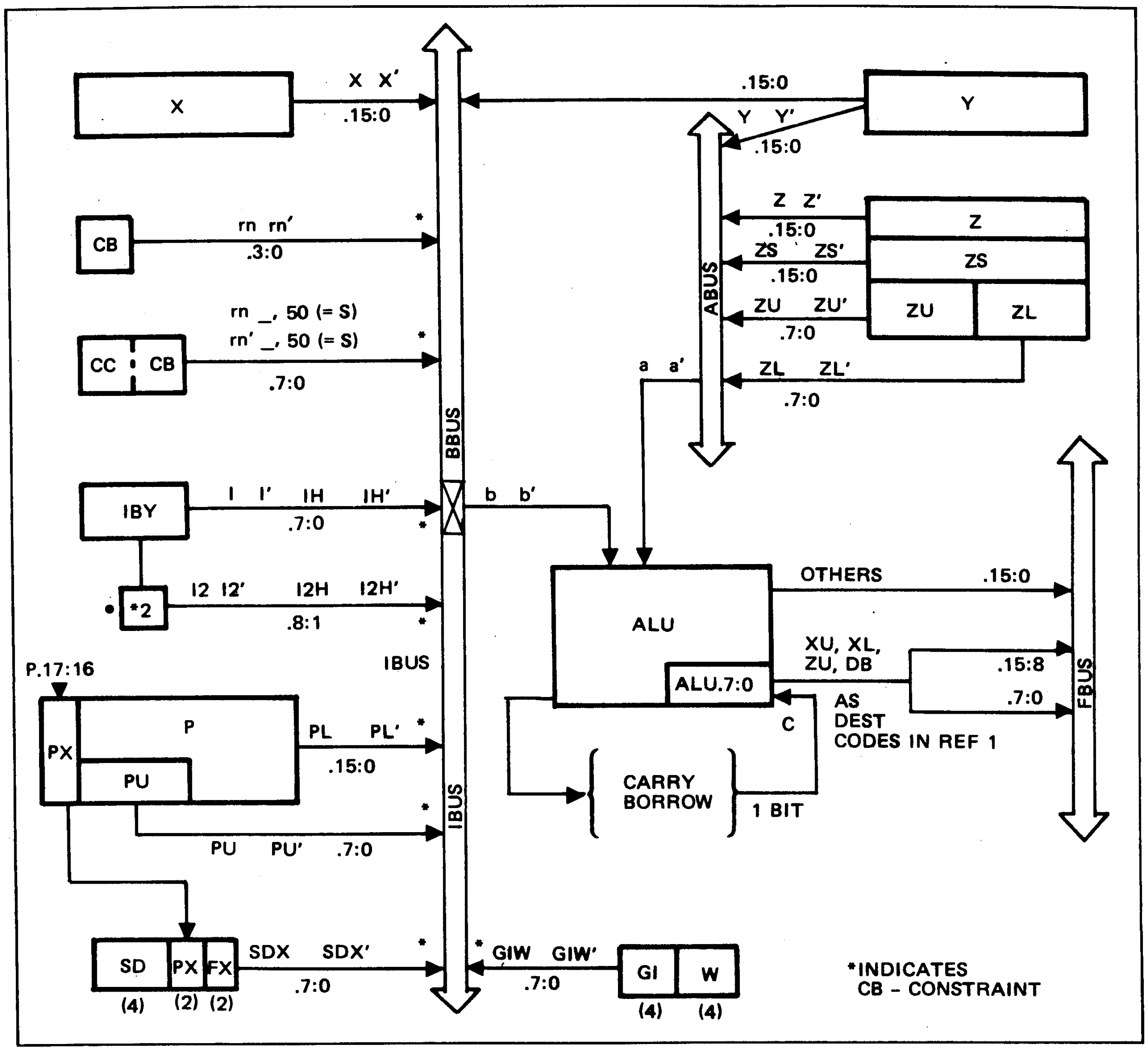


Figure A. ALU Sources and Bus Structure

CAP32 Microinstruction		Action
12'	Y	*Sets Y equal to FFF$_{16}$ (i.e., one's complement of 12).*
15'	LM(5)	*Sets CLM(5) equal to FFF0$_{16}$.*
Y'	Y:LM(6)	*Complements Y and stores result in both Y and in CLM(6).*

Figure B. Complementation Examples

5.2 ALU PAIRED LOGIC OPERATIONS AND ABUS DECREMENTATION

The ALU provides four two-input (paired) logic operations, taking one input from the ABUS and the other from the BBUS. A special decrementation logic operation is also provided.

Paired Logic Operations

For these operations, the syntax of subfield OPF1 taken on one of the following four forms:

```
a!b
a&b
a&b'
a*b
```

where "a" and "b" denote ABUS and BBUS source codes respectively); where "!" denotes a bitwise Inclusive-OR logic operation; where "&" denotes a bitwise AND logic operation; where "*" denotes a bitwise Exclusive-OR logic operation. Note that only in the form a&b' is complementation provided for the operand.

All logical operations are performed on 16 bits; however, a code used for a byte-size destination (XU, XL, ZU, DB) causes loss of the upper 8 bits, with the lower 8 bits of the result being copied into the upper byte (thus the result is present in both bytes of the ALU output).

Reference Table A-7 indicates that literals may be used as masks (i.e., as inputs to AND operations) only on BBUS inputs to the ALU. These are somewhat restricted by the normal limitation of literals to 4 bits. An 8-bit literal requires a subroutine call and return.[1] All literals may be complemented, which aids in forming masks. Working Registers also aid in forming complex masks through their upper byte and Hex digit codes.

If destination code F is used in REF1 (singly or in combination), the Arithmetic Flags (AF) for Zero (AFZ) and Negative (AFN) conditions are set to reflect the result. However, the Carry Flag (AFC) and the Overflow Flag (AFO) settings are always zero after a logic operation.[2] Target Flags may also be influenced.[3]

Sample logic operations are shown in Figure A.

ABUS Decrementation

Decrementation is normally considered arithmetic (subtracting 1 from the operand); however, the 3200 includes a special decrementation instruction which does not effect the Overflow and Carry Flags, but rather acts as a special logic operation. Special decrementation applies to ABUS sources only; it is specified by the following syntax:

```
a-ONE       destination
```

where "a" denotes an ABUS source code. The "a" is followed by a minus sign and "ONE" spelled out. If "1" is used in place of "ONE", decrementing will occur, but through normal channels of arithmetic with Carry (Borrow) and Overflow provided, and with CB-Constraint.[4] The use of "ONE" avoids CB-Constraint. Sample ABUS decrementation operations are illustrated in Figure B.

References:

[1]Topic 7.5 [3]Topic 5.5
[2]Topic 5.4 [4]Topic 5.3

CAP32 Microinstruction(s)		Action
Y*Y	LM(G)	*Clears CLM(G).*
Z!1	Z	*Z.0 is set to 1, while Z.15:1 remain unchanged.*
LMC(G) Z&Y'	Z LM(G)	*Clears the bits of CLM(G) that correspond in position to bits Y of value 1.*

Figure A. Sample Paired Logic Operations

CAP32 Microinstruction(s)		Action
Y-ONE	Y	*Sets Y equal to Y-1 mod 10000_{16}.*
Y-ONE	LM(Z4,LG)	*Sets CLM(Z.3:0) equal to Y-1 mod 10000_{16}; sets G equal to Z.3:0; leaves Y and Z unchanged.*
SB(6) Y-ONE Y&X	Y Y X	*Clears upper 10 bits of X without using Z.*
Z-ONE Z' SB(Z4) Y-ONE Y&X	Z Z Y Y X	*Clears the upper k bits of X, where k is the value stored in Z.3:0. If Z.3:0 = 0, then all of X is cleared.*
Y&X LMN(9) D Y-ONE	X MR Y Y:LM(14)	*Reads a word from the MONOBUS at location FX//SLM(9), then places a value one less than the value of that word into Y and PLM(14); assumes PLM = CLM.*

Figure B. Sample ABUS Decrementation and Paired Logic Operations

5.3 ALU ARITHMETIC OPERATIONS

The ALU provides 4 two's complement arithmetic operations: addition or subtraction with carry or borrow, and addition or subtraction without carry or borrow.

An arithmetic operation is specified by using one of the following syntax forms in the OPF1 subfield:

```
a+b
a+b+C
a-b
a-b-C
```

where "a" and "b" denote the ABUS and BBUS source codes, as for the paired logic operations[1]; and where "C" denotes the use of the carry or borrow bit from the next less significant word (i.e., carry for addition and borrow for subtraction). Note that reverse subtraction (i.e., b-a) is not provided.

A carry output bit is provided from either addition operation, and a borrow output bit is provided from either subtraction operation. The borrow bit is the complement of the carry bit normally provided at a simple binary adder's output; the borrow is 1 if a borrow is required.

If destination code F is used in subfield REF1, then all four AF's are set by the results of the arithmetic operation.[2] Similarly, if code TFG is used in subfield REF3, and code F is used in subfield REF1, then the 4 bits of TFG are also set by the arithmetic result.[3]

Arithmetic is either 16-bits or 8-bits wide, depending on the FBUS destination code. REF1 codes XU, XL, ZU, and DB are 8-bit arithmetic; all other destination codes are 16-bit arithmetic.

Figure A shows sample arithmetic operations.

References:

[1]Topic 5.2
[2]Topic 5.4
[3]Topic 5.5

CAP32 Microinstruction(s)		Action
Y-Y	LM(G)	*Clears CLM(G) to 0.*
Y+X	Y	*Adds X to Y.*
Z+Y	Y	*Adds Z to Y.*
Z	MW	*Adds X to memory word addressed by FX//Z.*
D	Y	
Y+X	D	
LM(G)	Y	
Y+1	LM(G)	*Increments PLM(G).*
Y-ONE	Y	
Y'	Y	*Changes sign of Y.*
Y,6,LG	MR	
D	X	*Adds word from memory at address Y to the two-word integer in PLM(7)//PLM(6).*
LM(G)	Z	
Z+X	LM(G,IG):F	
LM(G)	Z	
Z+O+C	LM(G)	
Y	MR	*Adds 1 to the secondary LM register indexed by the 4 lowest-order bits of the memory word addressed by Y.*
D	Z	
LM(Z4,LG)	Z	
Z+1	LM(G)	
LMC(4,LG)	Y	*Adds X to CLM(4).*
Y+X	LM(G)	
LMC(4)	Y	*Adds CLM(4) to X.*
Y+X	X	
X,4,LG	Y	*Subtracts CLM(4) from X.*
LMC(G)	X	
Y-X	X	
Y,5,LG	MR	*Adds the memory word addressed by Y to the memory word addressed by PLM(5).*
D	X	
LM(G)	MW	
D	Z	
Z+X	D	

Figure A. Sample Arithmetic Operations

5.4 ALU ARITHMETIC FLAGS

Upon request, the Arithmetic Flags (AF's) and the Target Flag Register (TFG) will monitor and report the Carry, Overflow, Negative, or Zero conditions of ALU output. This topic describes the AF's.

AF's remain unchanged by microcommand execution unless destination code F is used in subfield REF1 (alone or in combination). When F is used, the AF's are enabled. (Unless enabled, the AF's remain unchanged.) If the AF's are enabled, their new values depend upon the ALU output as discussed in the following paragraphs.

If the destination code used in REF1 is either XL:F or ZU:F, then the AF settings reflect the lower half condition of the ALU output (ALU.7:0), rather than the full 16-bit ALU output. All other destination codes involving F cause the new AF settings to reflect the full ALU output. (In the following discussions, the term ""AALU" denotes the appropriate ALU output, full or half.)

The new values of the enabled AF depend on whether the ALU operation was arithmetic or not. Only OPF1 codes a+b, a+b+C, a-b, and a-b-C are considered to be arithmetic (OPF1 code a-ONE is not considered arithmetic for this purpose). If the ALU operation is not arithmetic, and if the AF's are enabled, then:

- AFC (Carry Flag) is cleared to 0.
- AFO (Overflow Flag) is cleared to 0.
- AFN (Negative Flag) is set equal to the sign bit of AALU (ALU.7 or ALU.15 as appropriate), with a value of 1 for negative, 0 for positive.
- AFZ (Zero Flag) is set to 1 if all AALU bits are 0, otherwise AFZ is cleared to 0.

If the ALU operation is a+b, and the AF's are enabled, then:

- AFC (Carry Flag) is set to the value of the carry bit out of the sign position of AALU (i.e., 1 if a carry is propagated, 0 otherwise).
- AFO (Overflow Flag) is set to 1 if the addition has overflowed, otherwise AFO is cleared to 0. (Overflow is defined as if ABUS.SIGN and BBUS.SIGN are equal to each other and not equal to the AALU.SIGN an overflow exists.
- AFN (Negative Flag) is set to the true sign of the sum, regardless of overflow. (If there has been overflow, AFN≠AALU.SIGN, otherwise AFS=AALU.SIGN.)
- AFZ (Zero Flag) is set to 1 if all AALU bits are 0, otherwise AFZ is cleared to 0.

If the ALU operation is a+b+C, and the AF's are enabled, then:

- AFC, AFO, and AFN are set as described for a+b above.
- AFZ (Zero Flag) is left a 0 if it was to 0 at the start of the operation; otherwise it is set as described for the a+b operation. The effect is to provide a multiword precision arithmetic zero test. If any word of the result fails to be zero, the overall zero test will fail, and AFZ will be zero.

If the ALU operation is a-b, and the AF's are enabled, then:

- AFC (Carry Flag) is set to the borrow bit (the one's complement of the carry bit) out of the sign position of AALU; it is set to 1 if a borrow of 1 is required.
- AFO (Overflow Flag) is set to 1 if the subtraction has overflowed, otherwise AFO is cleared to 0. (Here overflow is defined as if ABUS.SIGN is not equal to BBUS.SIGN and the ABUS.SIGN is not equal to AALU.SIGN then an overflow exists.
- AFN and AFZ are set as described for the a+b operation.

If the ALU operation is a-b-C, and the AF's are enabled, then:

- AFC, AFO, and AFN are set as described for a-b above.
- AFZ is set as described for the a+b+C operation above.

The utility of the AF bits is primarily for branch testing.[1] If desired, the individual AF bits (or the logical-OR of bits AFZ and AFN) may be placed directly in FBUS.0, as shown in Reference Table A-6. In any case, use of the AF bits causes CB-Constraint. Sample setting of the AF's is illustrated in Figure A.

Reference:

[1]Topic 7.3

CAP32 Microinstruction		Action				
Z+Y	F:Z	*Adds Y to Z; sets AF's:*	*AFC*	*AFO*	*AFN*	*AFZ*
		$E426_{16}$+$C077_{16}$:	*1*	*0*	*1*	*0*
		$52AA_{16}$+$ABCD_{16}$:	*0*	*0*	*1*	*0*
		$9E4F_{16}$+$A017_{16}$:	*1*	*1*	*1*	*0*
Z-X	F:X	*Subtracts X from Z; sets AF's:*				
			AFC	*AFO*	*AFN*	*AFZ*
		$E426_{16}$-$C077_{16}$:	*0*	*0*	*0*	*0*
		$9E4F_{16}$-$A017_{16}$:	*1*	*0*	*1*	*0*
		$57DD_{16}$-$A823_{16}$:	*1*	*1*	*0*	*0*

Figure A. Sample Setting of Arithmetic Flags (AF's)

5.5 ALU TARGET FLAG REGISTER

This topic describes the Target Flag Register (TFG) of the ALU.

The Target Flags are similar to the Arithmetic Flags described in the preceding topic.[1] The differences are as follows:

- The TFG is an ordered-bit register (C,O,N,Z) from TFG.3 to TCO.0. TFG may be loaded from the lowest four bits of Z-Register (Z.3:0). TFG may be sent to the FBUS as a 4-bit Hex digit. This differs from the behavior of the AF's.
- The TFG can be enabled to reflect the ALU output conditions (as the AF's can), but with a subtle difference in the reflection of overflow. The TFG remains unchanged unless it is enabled. Enabling of the TFG occurs if code TFG is used in subfield REF3. If TFG is enabled, and code F is used as part of the destination code in REF1, then the TFG will reflect the C,O,N,Z ALU output conditions.

When both the AF's and the TFG are enabled (by code F in REF1, and code TFG in REF3), then the AF's and the TFG are set identically, except for the following: the overflow-Flag bit (TFG.2) is not cleared by ALU action, but only by moving a zero into that position from Z.2. The AF-Overflow bit (AFO) is cleared by execution of any non-arithmetic ALU operation, or any arithmetic operation that does not overflow. Therefore, if TFG.2 is set to 1 at the beginning of an arithmetic operation, and that operation does not overflow, then TFG.2 will still be set to 1 at the end of the operation, whereas AFO will equal 0.

The TFG may not be tested directly for branching purposes, but must first be transmitted to a Working Register (X, Y, or Z), where it may be tested.[2]

If the TFG is enabled by code TFG in REF3, and code F is not used in REF1, then the TFG will be set equal to Z.3:0. This is the method required for resetting the TFG. If code TFG is used in subfield OPF1, then the TFG will be the source of the FBUS signal. This use of TFG as a source is the primary action of the microcommand in which it appears; TFG as a destination is secondary, since TFG is driven by private "back-door" paths, and not the FBUS.

Figure A exemplifies the setting of the TFG.

Use of AF's Versus Use of TFG

Since the AF's and the TFG are similar in many ways, questions may arise as to when one should be used as opposed to the other. Normally, AF's should be used at the microlevel to monitor internal workings of the microroutines. Many such conditions need not be reported to the user. The TFG, on the contrary, should normally be used to report results of the microroutine. The user may then decide other actions before attending to the particular TFG condition reported. In the interim, the condition codes may stay undistrubed in the TFG, and AF's may be used extensively at microlevel without interference to or from the TFG. Note that the AF's are effected when the TFG manipulated by the ALU output; therefore, conditions cannot be set in the TFG from the ALU, without their existence in the AF's. The AF's, however, can be affected without influencing the TFG.

In the time-sharing context at the user level, the machine state must be savable and restorable. Since the TFG can be reloaded via the Z-Register, the TFG is well adapted to that particular operational requirement. It would be difficult to use AF's in that way, since AF bits are not easily restored. Precautions should be taken when using the TFG for multiword precision arithmetic. Overflow bit TFG.2 must explicitly be cleared via the Z-Register immediately before processing the most significant word (i.e., since TFG.2 may well have been set by earlier words, and clearing of that bit is not automatic).

References:

[1]Topic 5.4
[2]Topic 7.3

CAP32 Sequence		Action
LMS(0)	.	*Saves the TFG in secondary Local*
TFG	LM(O)	*Memory cell 0.*
LMS(0)	Z	*Retrieves and restores the TFG*
Y	.,,TFG	*stored above.*
LMC(5,LG)	Z	*Adds a one-word integer in X to the*
Z+X	F:LM(G)	*two-word integer in LM(4)//LM(5).*
LMC(4,LG)	Z	*Sets AF and TFG.*
Z+O+C	F:Z,,TFG	
Z	LM(G)	
F(Z)	Z	
TFG	X	
Z&X	Z	
Y	.,,TFG	

Figure A. Sample Setting of the Target Flag Register (TFG)

5.6 ELEMENTARY SHIFT OPERATIONS

This topic presents a discussion of elementary shift operations.

Shifts are performed in one or more of the Working Registers (X, Y, and Z). All microlevel shifts are shifted one bit position. For longer shifts, a small loop is established which performs the desired number of elementary shifts. Refer to the referenced topic for examples of multibit shift loops under control of W.[1]

To perform an elementary shift, the user must specify the direction (left or right), the register(s) to be shifted, and the source of the fill bit to be moved into the bit position vacated at one end of each shifted register. The CAP32 syntax for elementary shift specification is:

```
     SL(lsb-fill)        registers

     SR(msb-fill)        registers
```

where "SL" denotes a shift left operation, "SR" denotes a shift right operation, "lsb-fill" denotes the least significant bit fill character, "msb-fill" denotes the most significant bit fill character, and "registers" denotes a suitable destination code from Reference Table A-2, of one of the following types:

Type xy:	X or XL or XU or Y
Type z:	Z or ZU
Type yz:	Y:Z

If a type xy or type z register code is specified, a single register (or half-register) is shifted, as named by the code. If Y:Z is specified, both registers Y and Z are shifted during the same microcycle.

For each shift direction, the allowable fill values depend on the type of register code specified. The possibilities are displayed in Reference Tables A-8 and A-9.

The available elementary shifts may be related to conventional shift concepts. For example, among the available single-word (16-bit) shifts are:

Single Left Circular:	SL(Y)	Y
Single Right Arithmetic:	SR(Y)	Y
Single Left Logical:	SL(O)	X
Single Left Logical:	SL(O)	Y
Single Left Logical:	SL(,O)	Z
Single Right Logical:	SR(O)	X
Single Right Logical:	SR(O)	Y
Single Right Logical:	SR(,O)	Z

Here are two sets of simultaneous logical one-word shifts:

Single Left Logical Pair:	SL(O,O)	Y:Z
Single Right Logical Pair:	SR(O,O)	Y:Z

The available conventional double-word (32-bit) shifts are:

Double Left Circular:	SL(Z,Y)	Y:Z
Double Right Circular:	SR(Z,Y)	Y:Z
Double Right Arithmetic:	SR(Y,Y)	Y:Z
Double Left Logical:	SL(Z,O)	Y:Z
Double Left Logical:	SL(O,Y)	Y:Z
Double Right Logical:	SR(O,Y)	Y:Z

A novel arithmetic right shift is available for X or Y, based on the true sign of the most recent ALU result that occurred with Arithmetic Flags enabled:

Single Right True-sign:	SR(N)	Y
Single Right True-sign:	SR(N)	X
Double Right True-sign:	SR(N,Y)	Y:Z

A novel left shift of Z is available, with fill of 1 if the sign bits of X and Y are the same, and fill of 0 if the sign bits of X and Y are different:

Single Left Sign-equivalent:	SL(,XY)	Z

Triple-word shifts may be compounded by a succession of single and/or double word shifts. Here is one example (others are in the exercises):

Triple Left Logical:	SL(Y)	X
	SL(Z,O)	Y:Z

Figure A illustrates some sample shift operations. For further details regarding shifts, the reader should refer to Reference Tables A-8 and A-9.

Reference:

[1]Topic 7.2

CAP32 Sequence		Action
Y	Z	*Causes a single right*
SR(Z)	Y	*circular shift of Y.*
Y,4,LG	.	
LMC(G,IG)	Y	*Causes a triple-word (48-bit)*
LMC(G,IG)	Z	*right arithmetic shift of*
LMC(G)	X	*CLM(4)//CLM(5)//CLM(6).*
SR(Z)	X	
SR(Y,Y),4,LG	Y:Z	
Y	LM(G,IG)	
Z	LM(G,IG)	
X	LM(G)	

Figure A. Sample Shift Operations

6 CONTROL MEMORY ADDRESSING AND CAP32 ASSEMBLY LISTING

6.1 CONTROL MEMORY ADDRESSING

This section consists of two topics. This initial topic describes Control Memory addressing. The second topic discusses the format of the CAP32 assembly listing. These topics are presented at this point to provide sufficient background information for subsequent sections of this manual.

The Control Memory of the 3200 consists of 4K 32-bit words, organized as shown in Figure A. The address of the current microinstruction control location is in the 11-bit L-Register at the start of its execution. By convention, the least significant bit of L is interpreted as a sign (0 as minus, 1 as plus) and displayed (on the Maintenance Panel) as the right most 4 bits with value 0= - 1=+. That sign if conventionally written after the pair address, in listings of 3200 microprograms.[1] Figure B presents some examples of Control Memory addressing.

The Control Memory is highly structured; the two words of a location are the next alternative instructions in a simple branch.[2] The 16 pairs in a block are the alternatives in a vector branch (computed GO TO).[3] These restrictions on branching yield substantial efficiency and speed at execution time, but cause extra microassembler effort to fit every Control Word into a usable location.

Every 3200 microprogram Control Word contains (or points to) the address of its successor (next microinstruction) or alternative successors. There is no "automatic" control flow to the next higher absolute Control Memory address. However, if not otherwise specified, the CAP32 microassembler assigns the next Control Word in the input sequence. At execution time, the 3200 discovers this by examining the Word, which must point to the next one.

There are 8 addressing modes for the Control Memory. The mode value (0:7), which is used to fetch the next Control Word, is stored in the CD-Field of the current Control Word (C-Register bits C.14.12). The programmer generally has only partial control over the mode used; CAP32 exercises a great deal of judgment in placing the microcommands in control memory. The 8 addressing modes are presented in Reference Table A-10. The modes will be examined herein as needed.

The programmer exercises required control over the Control Word location through use of selected CAP32 pseudoinstructions such as PLACE[4], through entries in subfield REF2 (next-address subfield), and through use of statement labels.

References:

[1]Topic 6.2
[2]Topics 7.1 through 7.6
[3]Topics 8.1 through 8.5
[4]Topic 8.6

Organization		Bits	Location	Value
Control Memory	= 8 pages	xxx	L.10:8	0:7
Each page	= 16 blocks	xxxx	L.7:4	0:F
Each block	= 16 location	xxxx	L.3:0	0:F
Each location	= 2 words	x		-,+

Figure A. Control Memory Organization

Value in L	Meaning
10E1	*10E+ (Page 1, Block 0, Pair E, Word +)*
3C20	*3C2- (Page 3, Block C, Pair 2, Word -)*
07B1	*07B+ (Page 0, Block 7, Pair B, Word +)*
7D00	*7D0- (Page 7, Block D, Pair 0, Word -)*
2CE1	*2CE+ (Page 2, Block C, Pair E, Word +)*

Figure B. Sample Control Memory Addressing

6.2 CAP32 ASSEMBLY LISTING

The format of the assembly listing produced by the CAP32 Microassembler is discussed in this topic.

Figure A presents a sample (partial) CAP32 assembly listing. The page heading of the listing identifies the assembler, the program being assembled, the date and time of assembly, and the page number. A partial list of column headings is given on the second line of the listing. There are 12 columns of interest, as discussed in the following paragraphs.

The extreme left column of Figure A is blank since there were no assembly errors. If the programmer requests an address structure that CAP32 cannot handle, an E appears in the first column. Other error flags may also occur.

The column titled "LOCN" gives the Control Memory assigned by the assembler for every executable statement (and VECTOR pseudoinstruction). As shown, the assembler often takes advantage of addressing properties, and uses successive addresses having a common pair address (7 in this case). Also, note that of the 6 executable statements shown, 4 are paired.

The column titled "INSTR" gives the Control Word value in hexadecimal notation.

Two character positions to the right of the Control Word value, an asterisk (*) is shown if the statement was CB-Constrained by the programmer, calling for specific use of the CB Field. (The assembler may or may not have used CB for its purposes.)

The column titled "+ SUCC -" lists the Instruction Sequence Number (see below) of the successor to the current instruction. If the successor is a + word, it is listed under the "+"; if the successor is a - word, it is listed under the "-". If the current line is a conditional branch instruction,[1] then the two successors will be shown (one for each possible branch). The user should note that the assembler assigns a Control Word to a + word or - word at its pleasure, if not forced to a particular sign via branching or other considerations.

The column titled "ISN" gives the Instruction Sequence Number of the current line, which the assembler uses to refer to the source code line during the assembly process. Note that the source code is in ISN order, but that the actual Control Memory addresses (as given in the LOCN column) are far out of order on the listing.

Five columns are bracketed under the title "SOURCE"; these columns represent the CAP32 source statements as input by the user. The first column shows the Label Field (LAF); an asterisk in character position 1 of this field indicates a comment line. Following the LAF are the Operation Field (OPF), the Result Field (REF), and the Comment Field (COF), as previously discussed.[2] Card numbers occur in the right-most column of the assembly listing.

The assembler also provides a complete cross-reference listing, tables of all vectors, and the final machine language code arranged in Control Memory address order.

References:

[1]Topics 7.1 through 7.6
[2]Topic 3.1

```
CAP3200 MICRO-ASSEMBLER. VERSION 1.   ASSEMBLY LISTING   PROGRAM: F32SA    74/01/17 12:18  PAGE 10
 LOCN    INSTR     +     SUCC    .   ISN ------------------------------ SOURCE ----------------------

                                                                                               471.
                                 * *    FIRST DIGIT 7 - FOUR BIT LITERAL (SECOND DIGIT)        472.
                                 * *                                                           473.
                                 ***                                                           474.
017+   00C1254B    222   223  221  FD7    LM(U)     X,N5(,FD7.3)      SAVE POSS. SPILL, SKIP IF NOT FULL  475.
054+   007300C2 *        450  222         LM(SP,LG) Z,SBR(PUSH)       PUSH SPILL INTO MEM      476.
054-   03031240 *        224  223  FD7.3  SDX       Z                 GET SECOND DIGIT OF OPCODE  477.
024-   81F02CF1    225        224  FD7.4  Y         LM(U,IT)          PUSH TOS                 478.
OCE+   0E02700F          12   225         ZB        Y,N1(NEXT)        4 BIT SEC. DIG. TO TOS, EXIT  479.
017-   03032240 *        224  226  FD7.   SDX       Z,FD7.4           GET SEC. DIG. OF OPCODE  480.
```

Figure A. Sample CAP32 Assembly Listing (Partial)

7 BRANCHING CAPABILITIES

7.1 UNCONDITIONAL AND CONDITIONAL BRANCHING

This topic describes the unconditional branching capability of CAP32, and presents a discussion of the general syntax for the conditional branching capabilities.

Unconditional Branching

Labels used in CAP32 statements must consist of 1 to 6 letters, digits, or period characters (.). Labels must begin with a letter, and must begin in character position 1 on the line that defines them (i.e., the line in which they appear in the LAF field). Use of a defined label in subfield REF2 of a statement causes assembly of an unconditional branch (i.e., a simple jump) to the defining statement of that label. The jump may be forward or backward. Figure A illustrates the general format of a simple jump. An unconditional branch may also be made to a location relative to the current statement, *±n in REF2. Here the asterisk (*) refers to the current instruction. As a specific example, consider Figure B. In the example, note that the "-4" applies to the Instruction Sequence Number (ISM) series, not to the absolute Control Memory addresses.

Conditional Branching Syntax

A conditional branch in a CAP32 microprogram is called for by the use of subfield REF2 in one of the following forms:

```
test-code(tlabel,flabel)
test-code(tlabel)
test-code(,flabel)
test-code(=S)
```

where "test-code" is selected from Reference Table A-11; where "tlabel" names the statement to be branched to if the test succeeds (is true); where "flabel" names the statement to branched to if the test fails (is false); and where "=S" is a keyword indicating that the word to be branched to is one word of the pair addressed by S.15:4. The +word is used if the test succeeds, and the -word is used if the test fails.)[1] In all cases, the two alternative statements to be branched to must be the + and - words of the same pair. The CAP32 assembler always assigns "tlabel" to the + word and "flabel" to the - word of a pair.

If either "tlabel" or "flabel" is missing (with the other present) the missing label is assumed to have the value *+1 (i.e., the next executable statement in the listing having ISN one greater than the ISN of the current statement). That next statement is compelled to be one of the paired alternatives to which control will transfer, and accordingly is paired with the given label. Either "tlabel" or "flabel" (or both) may be of the form *+n rather than a symbol. Here "n" denotes a decimal integer, and "*" denotes the current statement, thus *+n denotes the executable statement with ISN = CISN + n (where CISN is the current instruction sequence number). Some examples of allowed REF2 forms are shown in Figure B.

The test evaluates conditions existing at the start of the current instruction (not at completion). Though the current instruction is "after the fact" of the test, it is executed each time regardless of which alternative branch is taken. The flowchart of Figure C clearly shows this effect; the test is before the main action of the instruction, and the branch is after the action.

The available "test-codes" are summarized in Reference Table A-11, and are also detailed in subsequent topics within this section.[2]

References:

[1]Topic 7.4 [2]Topics 7.2 through 7.5

LAF	OPF	REF	COF
label	opf	ref	label usage in LAF defines label.
	opf	ref,label	label usage in REF2 causes jump.

Figure A. General Form of Unconditional Branch (Simple Jump)

REF2 Subfield	Action
test-code(CRY1)	*Branches to CRY1 if "test" succeeds; otherwise continues to *+1.*
test-code(*)	*Repeats current instruction if "test" succeeds; otherwise continues to *+1.*
test-code(,*)	*Repeats current instruction if "test" fails; otherwise continues to *+1.*
test-code(GULF,*-4)	*Branches to GULF if "test" succeeds; otherwise branches back to *-4.*

Figure B. Sample REF2 Test Forms

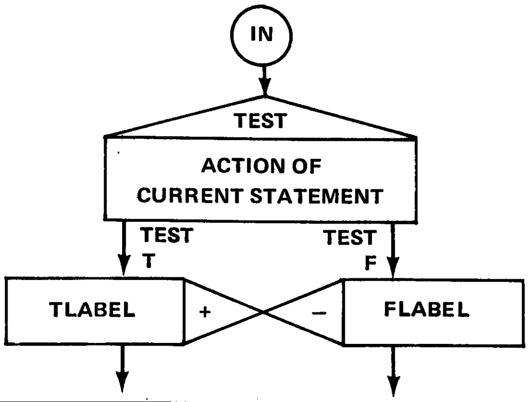


Figure C. General Flowchart of a Conditional Branch (Showing Test Before the Action to Paired Statements, and Branch After the Action).

7.2 CONDITIONAL BRANCHING: GI AND WNZ TESTS

The GI tests are used to test the General Indicator bits. The WNZ test is used to test the W-Counter.

GI Tests

GI tests are specified by using test-codes GI3, GI2, GI1, or GI0; they test General Indicator Bits GI.3, GI.2, GI.1, or GI.0 (respectively). The test succeeds if the specified GI bit is 1.

Any 4-bit quantity may be loaded into the GI-Register (from the FBUS) for testing. Figure A shows an example where bit Z.10 is tested, and a branch to label ZGO occurs if Z.10 has a value of 1. Figure B illustrates the flowchart for this example.

WNZ Test

The "W-Counter Not Zero" test is specified by using test-code WNZ. The W-Counter will be tested; if the W-Counter is not equal to 0, the test succeeds. Additionally, the W-Counter will be decremented by 1.

The W-Counter is a 4-bit counter, which is initialized from the FBUS via REF1 code W. Thus, the W-Counter is an excellent mechanism for all microlevel interactions of 16 cycles or less.

The example in Figure C shifts the Z-Register left logically 5 bits, filling Z.0 with zero each time. A flowchart of this example is shown in Figure D.

```
OPF        REF        COF
---        ---        ---

ZC         GI         LOAD Z.11:8 INTO GI.
Y*         .,GI       IF GI.2=1 GOTO ZGO; ELSE GOTO *+1.
```

Figure A. Sample Usage of GI Test (See Text on Opposite Page)

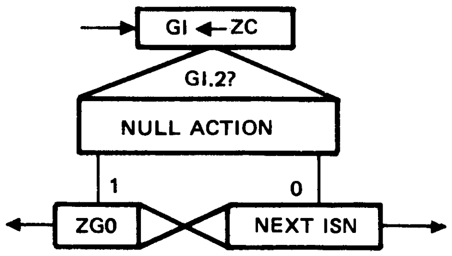


Figure B. Flowchart for Example Shown in Figure A.

```
OPF        REF         COF
---        ---         ---

4          W           INITIALIZES W AT 4.
SL(0)      Z,WNZ(*)    IF W≠0, DECREMENT W AND SET REPEAT;
                       SHIFT Z; REPEAT IF SET.
```

Figure C. Sample Usage of WNZ Test (See Text on Opposite Page)

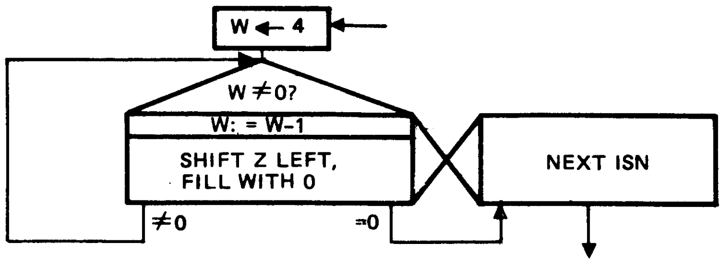


Figure D. Flowchart for Example Shown in Figure C

7.3 CONDITIONAL BRANCHING: ARITHMETIC FLAG AND WORKING REGISTER TESTS

Arithmetic Flag tests are used to test the results of ALU operations. Working Register tests are used to test the "end bits" of the Working Registers.

Arithmetic Flag Tests

Arithmetic Flag Tests are coded similarly to the corresponding AF source codes.[1] They test the result of the most recent ALU operation in which REF1 code F was specified. All AF tests cause CB-Constraint.

The AF test-codes are listed in Figure A. To interpret the AF tests, the detailed behavior of the ALU must be understood.[2] In particular, recall AFC and AFO are cleared by every non-arithmetic operation, that AFN reports the "true-sign" of arithmetic (corrected for overflow) and not just the sign bit, and that AFC on subtraction is the borrow bit (the complement of the binary-adder carry).

The example in Figure C will cause a branch to YBIG if Y>X, otherwise a branch will be made to XBIG. The example in Figure D will write the larger of X and Y to the location addressed by Z.

Working Register Tests

Working Register tests are used to test the "end bits" of the X, Y, and Z Working Registers. Figure B presents selected samples of the test-codes used for these tests. Note these tests reflect the actual bits; if an overflow occurred in Y, for example, then test-code Y15 may give a different answer than test-code NEG. The Working Register tests may often be usefully combines with shifts.

The example in Figure E counts the number of "1" bits in Y, placing the count in CLM(7). The flowchart for this example is shown in Figure F.

References:

[1]Topic 5.4 [2]Topics 5.1 through 5.6

Condition	Test-Code	AF Tested
Zero	Z	AFZ
Negative	NEG	AFN
Positive	NZORN	AFZ' & AFN'
Non-Zero	NZ	AFZ'
Non-Negative	NNEG	AFN'
Non-Positive	ZORN	AFZ ! AFN
Carry	CAR	AFC
Overflow	OVFL	AFO

Figure A. Arithmetic Flag Test-Codes

Condition	Test-Code	Bit Tested
Y shows negative	Y15	Y.15 = 1
Z shows negative	Z15	Z.15 = 1
Z shows odd	Z0	Z.0 = 1
X and Y are of different sign	XY15	X.15 ≠ Y.15
Y and Z are different parity	YZ00	Y.0 ≠ Z.0

Figure B. Working Register Test-Codes (Selected Samples)

	ISN	OPF	REF
	101	Y - X	F
*	102	Y	.,NZORN(YBIG,XBIG)

Figure C. Sample Usage of AF Test (Branches to YBIG if Y > X, Otherwise Branches to XBIG)

	ISN	OPF	REF
	201	Y - X	F
*	202	Z	MW,NNEG(*+2)
	203	X	D,(*+2)
	204	Y	D

Figure D. Sample Usage of AF Test (Writes Larger of X and Y to Location Addressed by Z)

	+ SUCC	-	ISN	LAF	OPF	REF	COF
	302		301		Y*Y	Z	CLEAR Z.
*		303	302		Z',7,LG	W	W = 15, G = 7.
*	304	305	303	SLY	SL(Y)	Y,Y15(,*+2)	SHIFT AND TEST Y.
*		305	304		Z+1	Z	ADD 1 TO COUNT
	303	306	305		Z	LM(G),WNZ(SLY)	STORE COUNT, TEST FOR END.

Figure E. Sample Usage of Working Register Test (Counts "1" Bits in Y, Placing Count in CLM (7)).

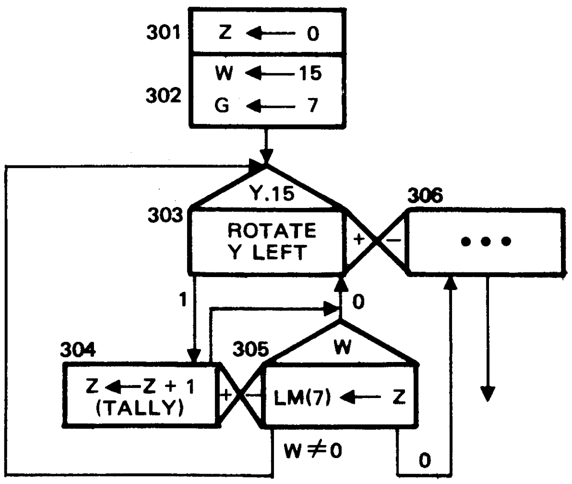


Figure F. Flowchart for Example in Figure E

7.4 SUBROUTINE BRANCHING

A subroutine call saves the "other-word address" of the current pair of words in the S-Register before branching to the subroutine. Hence a subroutine called from the + word of a pair returns to the - word of that same pair, and vice versa.

A subroutine call may be specified by use of the code SBR(label) in subfield REF2 of a CAP32 statement. The current word address (with + or - changed) is saved in S.15:4//S.0, and control then passes to the statement beginning with the label. Consider the example shown in Figure A. Statement ISN 253 (located at 057+) contains subroutine call SBR(PUSH). Hence, return address 057- will be stored in S.15:4//S.0, and there will be a branch to PUSH. The + successor column shows that PUSH is at ISN 450. Subroutine PUSH contains two statements (ISN 450 and 451) located at 037+ and 032+. No successor is shown for ISN 451, because 451 is the subroutine return, and the assembler cannot predict where the subroutine will return (as it cannot know from where the subroutine has been called). The subroutine return code is S0(=S) in REF2. Test S0 tests the value of S.0. Keyword (=S) indicates S.15:4 holds the address of the pair to be branched to. Hence return is to 057-. The flowchart for this example is shown in Figure B.

A second method of calling a subroutine uses destination code Z:SBR or Z:F:SBR in subfield REF1. A "Store L in S" then supplements the normal meaning of Z or Z:F as destination and flag codes. L is saved just as when SBR(label) is used in REF2, except that now REF2 is still free for a branch destination specification. For example, if ISN 500 is paired with ISN 501, and 500 contains:

```
     Z-ONE Z:SBR,Z15(DUN,RIT)
```

then a conditional branch to either DUN or RIT occurs, depending on the value of Z.15. A subroutine return in either DUN or RIT would return control to ISN 501. A flowchart of this example is given in Figure C.

Both methods of calling subroutines normally pair the statement containing the SBR in REF1 or REF2 with the next statement (having ISN one larger). However, if for some other reason the calling statement is paired, that other pairing prevails. Consider the example shown in Figure D. Here statement ISN 107 (PL) is paired with ISN 212 (MI) by the branch test in ISN 100. Control passes either to ISN 107 (PL) or 212 (MI), depending on "testx". If control goes to 107, subroutine RHO is entered as ISN 440, after execution of statement 107. Return after the subroutine is from 441 to 212. The conventional return point (ISN 108) is used for other purposes entirely.

LOCN		+ SUCC	-	ISN	LAF	OPF	REF
057+	*	450		253		LM(SP,LG)	Z,SBR(PUSH
057-	*		255	254		SDX	Z
⋮				⋮			
037+	*	451		450	PUSH	Z+2	MW:LM(G)
037+				451		X	D,S0(=S)

Figure A. Sample Subroutine Branch (See Text)

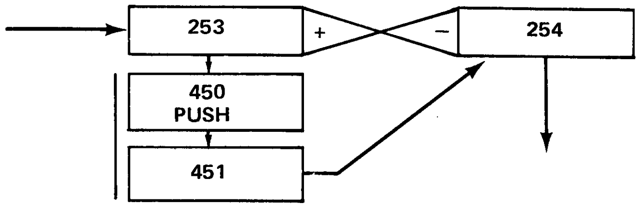


Figure B. Flowchart for Example Shown in Figure A

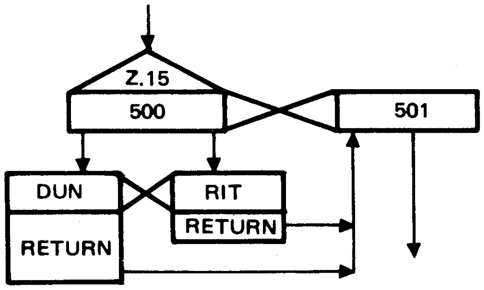


Figure C. Flowchart Showing Conditional Branch to One of Two Subroutines with Common Return (See Text)

+ SUCC	-	ISN	LAF	OPF	REF
107	212	100		opf	ref1,testx(PL,MI)
		⋮			
400		107	PL	opf	ref1,SBR(RHO)
		108			
		⋮			
213		212	MI	opf	ref
		213			
		⋮			
	441	440	RHO	opf	ref
		441		opf	ref1,S0(=S)

Figure D. Sample of Subroutine Call and Return Words Having Non-Successive ISN Values (See Text)

7.5 FULL-BYTE LITERAL SUBROUTINES AND CONTROLLED WORD PAIRING

Two additional branching capabilities are discussed in this topic: full-byte literal subroutines and controlled word pairing.

Full-Byte Literal Subroutines

In subroutine return code S0(=S), test S0 calls for bit S.0 to determine whether the + or - word of the pair addressed by =S will be used. (S.0 = 1 calls for +.) Any use of =S in REF2 calls for mode CD5 Control Memory addressing. As shown in Reference Table A-8, C-Field CC is not used in mode CD5, thus both CC and CB are available to hold an 8-bit literal, provided CB is not constrained some other way.

Since the CB field can only hold a 4-bit literal, in order to use this feature the CC field must be filled in with the high order 4 bits of the literal. This can be accomplished by writing:

```
     Y-4  Z,,cc=2
```

which causes 36 to be subtracted from Y and the result placed in the Z register.

A useful form is the one-line full-byte literal subroutine shown in Figure A. ISN 66 will be executed between ISN 47 and 48, and will set Y = AC_{16} before executing ISN 48. Thus, full-byte literals may be used in one-line subroutines, or in the last line of any subroutine.

Controlled Word Pairing

Two useful test-codes are available: "+" and "-". Code "+" selects the + word of the addressed pair. Code "-" selects the - word unconditionally. Consider the example shown in Figure B. In ISN 120, code "-" in subfield REF2 always transfers control to SIS (ISN 161). The inclusion of BRO as label forces BRO to be paired with SIS. Control does not go to BRO at all up to this point.

For controlled word pairing, the full-byte literal one-line subroutine can be placed in-line (in the used ISN sequence), instead of remotely. An example is shown in Figure C. Here, ISN 101 causes A and C to be paired + and - (respectively). Hence, SBR in REF2 of A establishes its return at C, since A and C are already paired. So, for convenience, B and C can be put in execution order after A. Operationally, this result is no different result than when the one-line subroutine B is remote; however, the program is more readable, and the desired pairing is guaranteed. If the desired result is not feasible, the assembler will inform the programmer, avoiding the obscure run-time error.

```
LOCN      + SUCC -     ISN     LAF     OPF     REF
----      --------     ---     ---     ---     ---

                66     47              opf     refl,SBR(LIT)
xxx+                   48
                       .
                       .
                       .
xxx-  *                66      LIT     12      Y,SO(=S), CC=10
```

Figure A. Sample One-Line Full-Byte Literal Subroutine

```
LOCN      + SUCC -     ISN     LAF     OPF     REF
----      --------     ---     ---     ---     ---

               161     120             opf     refl,-(BRO,SIS)
                       .
                       .
                       .
xxx+                   135     BRO     opf     ref
                       .
                       .
                       .
xxx-      162          161     SIS     opf     ref
                       162
```

Figure B. Sample Controlled Word Pairing

```
LOCN      + SUCC -     ISN     LAF     OPF     REF
----      --------     ---     ---     ---     ---
          102          101             opf     refl,+(A,C)
xxx+            103    102     A       opf     refl,SBR(B)
                104    103     B       lita    dest,SO(=S),cc-litb
xxx-      105          104     C       opf     ref
                       105

where lc is the low order 4 bit of the literal and lc is
the high order 4 bits.
```

Figure C. Sample One-Line Full-Byte Literal Subroutines with Controlled Word Pairing

7.6 ABSOLUTE STATEMENT LOCATION AND CONTROL BRANCHING

If desired, the programmer can exercise absolute control over statement location in Control Memory (not only the pairing). Furthermore, absolute branching may be specified based on data values stored in main memory. These techniques should be used sparingly, with careful consideration of their necessity.

Pseudo-instruction PLACE determines absolutely the location of the executable statement following it. (PLACE is discussed in a subsequent topic.)[1] The PLACE statement itself is not given an ISN, since the next line contains the statement it locates, with its own ISN. PLACE statements are vitally important in setting up vector arrays for controlled or computed jumps, as discussed subsequently.[2]

An absolute branch in Control Memory (based on an address stored in a data word in main memory) is set up by storing the desired target pair address in the upper three Hex digits of the word (bits .15:4). The example shown in Figure A shows a CAP32 sequence which reads such an address word, and jumps to the - word of the pair so addressed. Assume a pointer to the word in main memory in CLM(10), with FX holding the upper two bits of the main memory address. Statement ISN 102 is executed before the jump occurs, and it can involve a full-byte literal, since it uses mode CD5 to establish the next statement to be executed. This example would be extremely difficult to implement in a real system, since there is no check on the validity of the word coming from main memory (considered as a target pair address in the control memory). It is safer to jump to one of a closely controlled set of target Control Memory addresses.[3]

References:

[1]Topic 8.6
[2]Topic 8.1
[3]Topics 8.1 through 8.6

LOCN	+ SUCC -		ISN	OPF	REF	COF
		101	100	LM(10)	MR	ADDRESS MAIN MEMORY.
	102		101	D	S	READ WORD INTO S, VALUE xxx.
			102	opf	refl,-(=S)	JUMP USING MODE CD5.
			⋮			
			159	PLACE	xxx-	DEFINES NEXT LINE AT xxx-.
xxx-	161		160	opf	ref	JUMP DESTINATION.

Figure A. Absolute Branching Example

8 VECTORS AND PSEUDOINSTRUCTIONS

8.1 AN OVERVIEW

This section discusses Vectors and CAP32 pseudoinstructions. This initial topic introduces the main concepts involved.

A pseudoinstruction is a directive to the CAP32 microassembler; it does not result in generation of an object microcommand.

A branch set is a named list of labels defined by the BRSET pseudoinstruction, which has the following syntax:

```
     LAF           OPF          REF
     ---           ---          ---
  list-name       BRSET      label-list
```

where "label-list" is an ordered list of from 1 to 16 statement labels, separated by commas. Each label is defined elsewhere in the program by its occurrence in the Label Field (LAF) of an executable statement. The label-list may contain null elements indicated by two adjacent commas, and may be broken after any comma and continued on the next input line. When the Control Memory locations for all labels in the label-list are finally established by CAP32, the labels will be at consecutive word addresses of the same sign. The fact that the labels are at consecutive locations is not a serious restriction, because of the non-consecutive nature of Control Word access in the 3200. A BRSET label-list in CAP32 functionally corresponds to a label array in PL/I, to the list of statement numbers in a FORTRAN computed GO TO, and to the list of procedure names in the COBOL statement "GO TO procedure-list DEPENDING ON ...". A BRSET must appear in the CAP32 source code before any statement referring to it, and before any label in its label-list is defined.

A Vector consists of one or two branch sets associated with a method of determining an index value specifying one of the labels to be branched to. A Vector must be defined by a VECTOR pseudoinstruction, which has the following syntax:

```
     LAF          OPF          REF
     ---          ---          ---
vector-name     VECTOR    type,list-name [,list-name]
                          type-,list-name
                          type+,list-name
```

where each "list-name" has been defined previously by a BRSET pseudo-instruction, and where "type" associates the Vector with one of the index methods shown in Figure A. A VECTOR pseudoinstruction containing a single list-name in REF establishes that list as a 1-to-16-way switch (label array), with all addresses of the some wordsign. If + or - is specified as a suffix to "type" in REF, the wordsign is established as specified. If no sign is specified in REF, the assembler decides on the sign.
A VECTOR pseudoinstruction containing two list-names in REF establishes a 2-to-32-way switch (label array), with the first list-name defining the labels at - words, and the second list-name defining the labels at + words. Labels in corresponding ordinal positions in the two label-lists are necessarily at the - and + word locations of the same pair in Control Memory.

If the programmer wishes to establish the absolute location of a Vector, a PLACE pseudoinstruction[1] may be used immediately before the VECTOR pseudo-instruction. The sign specified in the PLACE argument must be acceptable to the VECTOR statement. Consider the example shown in Figure B. Here Vector MAVEC is a Second-Digit (SDB) Vector with four pairs of labels, the - word labels identified by list "MAV" and the corresponding + words labels identified by list "MAV.". The location of the first word of the first label-list named in the VECTOR statement is given as the location of the Vector (here 020-). Hence MA0, MA1, MA2, and MA3 are at 020-, 021-, 022-, and 023- (respectively), and MA0., MA1., MA2., and MA3. are at 020+, 021+, 022+, and 023+ (respectively). An ISN is given to a VECTOR statement, because the assembler must reference it often. Vector MAVEC is sketched in flowchart form in Figure C.

The user should note that each location of a Vector hold a full CAP32 statement, not just an address. A Vector is neither a data table nor a pure jump table; it is a procedure switch. When reaching an element of a Vector list, it is already in the first statement of an algorithm. Note that a subroutine called in a statement that is a Vector element automatically returns to the corresponding Vector element of opposite wordsign, or its equivalent. A Vector branch is called for in REF2 of a CAP32 statement, by naming the Vector (for single-list Vectors) or by using "test(vector)" (for two-list Vectors). Subsequent topics explore the various Vector branch calls.

Reference: [1]Topic 8.6

Type	Control Memory Address Mode	
DATA	CD4	(Data Branch)
FDB	CD7	(First-Digit Branch)
INT	CD7	(Interrupt or PC Branch)
SDB	CD6	(Second-Digit Branch)

Figure A. Vector Types

LOCN	ISN	LAF	OPF	REF
		MAV	BRSET	MA0,MA1,MA2,MA3
		MAV.	BRSET	MA0.,MA1.,MA2.,MA3.
			PLACE	020-
020-	358	MAVEC	VECTOR	SDB,MAV,MAV.

Figure B. Sample Absolute Vector Location (See Text)

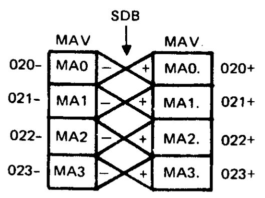


Figure C. Flowchart of Vector MAVEC from Figure B

8.2 DATA VECTORS

The simplest type of 3200 Vector is the DATA Vector.

The index for a DATA Vector is the low-order four bits of the S-Register (S.3:0). Hence, to branch into a one-list DATA-type switch, the general CAP32 sequence shown in Figure A may be used. A specific example of this form is shown in Figure B. Here the DATA Vector is called simply by naming it in REF2. If the DATA Vector were of two-list form, however, the call would require a test of the following form:

```
    opf          refl,test(vec2)
```

For example, Figure C shows an 8-by-2 Vector called with selection from Y.2:0 and Z.15. To show it can be done, a new value for Z has been established at the end of the sequence (in ISN 150). The Z.15 tested is the sign bit of the old value of Z.

A DATA Vector uses Control Memory addressing mode CD4 (the Data Branch) in which the address word has the composition shown in Figure D. The variable data values set the low-order Hex digit of the pair address and the sign. The CC-Field is set by the assembler to correspond to the block of the Vector. It is now seen that a DATA Vector should be placed at the beginning of a block having low-order digit 0. The current page must be used, since L.10:8 is not changed in CD4 mode addressing. Hence a DATA Vector can be called only in the page in which it is defined.

Another important restriction is that a DATA Vector cannot be called from within a conventional microlevel subroutine, since the loading of S.3:0 through destination code S in REF1 destroys the currently saved subroutine return address.

```
LOCN    LAF    OPF       REF
----    ---    ---       ---
        aaa    BRSET     label-list
xxx±    bbb    VECTOR    DATA,aaa
                 .
                 .
                 .
               source    S
               opf       refl,bbb
```

Figure A. General Form for Branching into a One-List DATA Vector

```
LOCN    + SUCC -    ISN    LAF      OPF       REF
----    --------    ---    ---      ---       ---
                           DVSET    BRSET     DVA,DVB,DVC,DVD,DVE,DVF
210                 701    VVD      VECTOR    DATA,DVSET
                     .
                     .
                     .
26D+         742    741             ZB        X
25D-                742             Y         .,VVD
```

Figure B. Specific Example of Branching into a One-List DATA Vector

```
LOCN    ISN    LAF      OPF       REF
----    ---    ---      ---       ---
               NEG      BRSET     NA,NB,NC,ND,NE,NF,NG,NH
               POZS     BRSET     PA,PB,PC,PD,PE,PF,PG,PH
                        PLACE     230-
230-    100    NAV      VECTOR    DATA,NEGS,POZS
         .
         .
         .
        149             Y&7       S
        150             LM(G)     Z,Z15(NAV)
```

Figure C. Specific Example of Branching into a Two-List DATA Vector

New bit	L.10	L.9	L.8	L.7	L.6	L.5	L.4	L.3	L.2	L.1	L.0	L.11
Set from	L.10	L.9	L.8	CC.3	CC.2	CC.1	CC.0	S.3	S.2	S.1	S.0	±test
*example	2 (same)			3 (new)				data (variable)				data (variable)

*See Figure C

Figure D. Address Word Composition Used by DATA Vector

8.3 USER-LEVEL INSTRUCTION FETCH AND DECODE CYCLE

The First-Digit and Second-Digit Branch Vector mechanisms are best understood in the context of the user-level instruction fetch cycle built into the hardware.

The user-level instruction fetch MONOBUS cycle is initiated by the P-Register (the user-level program counter), when either of these two events occurs:

- P is loaded from the FBUS through use of code P in subfield REF1.
- P is incremented from an odd to an even value when either 1) a First-Digit Branch (FDB) Vector name is used in a REF2 subfield with syntax "vector" or "test(vector)" and an FDB occurs, or 2) code "I" or "I2" is used as the b-code in OPF1, to feed part of the I-Register to the BBUS as an ALU input (CB-constrained).

Once the instruction fetch MONOBUS cycle is initiated, any subsequent reference to I-Register automatically causes a delay (for completion of the fetch cycle), so that the new value of the I-Register will be used. At all times the lowest order bit of P (P.0) acts as a byte selector on the I-Register, making only one of the two I-Bytes available to the CPU. If P is even, the high (left) byte of I is selected; if P is odd, the low byte is selected. In the following discussion, the selected byte of I is denoted by "IBY".

When P is incremented from even to odd, IBY switches from high to low byte with no MONOBUS fetch cycle occurring (since the needed byte is already placed in I). However, if P is loaded with an odd value from the FBUS, a MONOBUS fetch cycle occurs, and the low byte is selected as soon as I is loaded. In this case the high byte is not accessible to the CPU at all.

IBY can affect the CPU in eleven different ways, as tabulated in Reference Table A-12. Five of these involve the currently selected IBY; the other six involve ISAVE (the saved value of IBY from a previous selection). Each of the eleven effects are discussed below.

The first two IBY effects involve the current IBY, causing no change in the I-Register or the selection of IBY. Code IH used as the b-code in OPF1 places IBY on the BBUS as the low byte of the b-input to the ALU. Code I2H used similarly in OPF1 places IBY on the BBUS, shifted left one bit from the low byte.

The next two IBY effects are similar except that they also increment P, causing an advance of IBY to the next I-Register byte (with or without a MONOBUS access), as required to get the next IBY. Code I (like IH) places IBY in the low byte position on the BBUS. Code I2 (as I2H) shifts IBY one bit left before placing it on the BBUS, effectively multiplying IBY by 2.

The fifth IBY effect, the Vector First-Digit Branch (FDB), also uses the current IBY. First, all pending interrupts are handled before any other action takes place. Secondly the current IBY is saved as ISAVE, replacing the byte saved by the previous FDB. Third, the four high-order bits of IBY (the "First Digit") are placed in L.3:0, to effect a 16-way branch. Finally, P is incremented, advancing IBY to the next I-Register byte (with MONOBUS access only if required). To cause an FDB, an FDB-type Vector name is used in REF2, as discussed in the next topic. The FDB is particularly important, as it is the sole method of attending to interrupts, and is the only way to load ISAVE.

The last six effects involve ISAVE, the type which was the IBY at the previous FDB time. In the Vector Second-Digit Branch (SDB), the low-order (second) digit of ISAVE is placed in L.3:0 to effect a 16-way branch. To get an SDB, and SDB-type Vector name is used in REF2. Note that SDB is a branch on the lower 4 bits of the same byte the previous FDB used for a branch on the upper 4 bits.

Effect 7 in Reference Table A-12 takes second-digit (SD) on the BBUS as part of a compound ALU input that also includes FX and PX. The OPF1 code for this is SDX. SD appears as BBUS.7:4 and can most readily be separated from SX and FX by routing the signal to Z, then using code ZB as the next OPF1.

Finally, there are four separate $\pm$ branch tests based on selected bits of ISAVE. These tests involve bits of Second-Digit (SD), and also bits of the Previous First-Digit (PFD); they were designed to optimize the decoding of the user-level instruction set. A general test capability on IBY may be implemented by using an I-type OPF1 subfield on the current IBY to get it into Y or Z for masking, shifting, and testing.

As an example, assume that P has just been incremented to point to the even memory address containing the byte named G, and that the next four bytes in memory are named H, I, J, and K. Figure A shows the effects of a sample I-Register action sequence.

I-Register Action	Bits Acted Upon	P Incremented	MONOBUS Access	Resulting IBY	Resulting ISAVE
FDB	G.7:4	YES	NO	H	G
"I2H"	H.15:0	NO	NO	H	G
SDB	G.3:0	NO	NO	H	G
"I"	H.15:0	YES	YES	I	G
"BYTE"	G.5:3	NO	NO	I	G
"DBL"	G.5:3	NO	NO	I	G
FDB	I.7:4	YES	NO	J	I
"I2"	J.15:0	YES	YES	K	I

Figure A. Sample I-Register Action Sequence (See Text Above)

8.4 FIRST-DIGIT BRANCHING VECTORS AND INTERRUPT VECTORS

First-Digit Branching (FDB) Vectors and Interrupt (INT) Vectors are established using type-codes FDB and INT (respectively). Addressing in Control Memory resulting from these Vectors is of mode CD7.

The chosen strategy for decoding a user-level instruction depends on the specific format of the instruction. As described earlier, the 3200 optimizes decoding of instructions having a byte structure. An initial FDB on the 4 left bits of the instruction can be followed (as needed) by an SDB on the 4 right bits of the same instruction, preserved in ISAVE. Subsequent program bytes can be read into the CPU by I-type OPF1 codes (via) the BBUS) and handled as parameters or addresses, as appropriate. When the first byte of the next instruction is in the IBY position and the current instruction is complete, an FDB will start the next decoding cycle.

If the user-level instruction word structure does not adapt to this optimum scheme, the program byte can be read into the CPU by use of I-type OPF1 codes. The byte is then masked and selected using ALU logic functions and shifts, and finally used via a DATA Vector branch or via ordinary two-way ± branches. The FDB or the SDB Vector branch cannot be used on any argument other than IBY (for FDB) or ISAVE (for SDB). Interrupt handling can only be invoked through an FDB call. Hence the 3200 programmer is committed to using the First-Digit Branch mechanism.

The programmer establishes an FDB Vector just as a DATA Vector is established, except that subfield REF1 of the defining VECTOR pseudoinstruction is "FDB". An FDB-type Vector may have one or two label-lists, as desired. Each is defined by a BRSET pseudoinstruction and is subject to the same conventions on wordsign and pairing as other Vectors.[1] A unique FDB Vector restriction is that it must be located either in Block 1 or Block 9 of a Control Memory page, and its associated Interrupt Vector (of type INT) must be right below it, in Block 0 or Block 8. As Reference Table A-8 shows, addressing mode CD7 places bit string 000 in L.6:4 if an interrupt is pending, otherwise it places bit string 001 in L.6:4. These CD7 values define the block in the page. The programmer should place the FDB Vector through use of the PLACE pseudoinstruction, specifying value 010-, 090-, 110-, 190-, ... (i.e., one of the legal locations for an FDB Vector). Then the matching Vector of type INT must be defined, and placed at 000+, 080+, 100+, 180+, ... (correspondingly). The INT Vector is necessarily of single-list type, and must be placed at + address. The FDB Vector could be placed at + addresses if it were of single-list type, but often double-list FDB Vectors prove useful. Figure A provides an example. Note that every possible target location must be named. The naming is arbitrary, but ordinal naming is very helpful.

A 48-way conditional branch has been established in Figure A. It can be entered by a statement having in its REF2 subfield the form test(NEXT). The test result will determine whether to use list FDL. (for + word addresses) or FDL (for - word addresses); test success causes entry to list

"FDL.". IBY.7:4 determines the point in the list to which control will be transferred provided no interrupt is pending. If an interrupt is pending, control will then transfer to the appropriate element of list INTL. When interrupt handling is completed, control shall normally be returned to the FDB Vector. Such a return is not automatic, and must be explicitly programmed into each interrupt routine.

The 16 possible types of interrupts are listed in Reference Table A-13 (from highest to lowest priority). Interrupts are handled in priority order, if more than one are pending. Two forms of interrupts are directly initiatable from the microprogram. First, if an FDB branch is desired without interrupt servicing, code PI in subfield REF3 (of the statement which precedes the statement containing the FDB Vector name in REF2) will prevent interrupts for that one FDB. Second, 8 of the interrupts are under the control of the Q-Register, and will not be accepted unless the corresponding bit of Q is set to 1. Q-Bit assignments to the various interrupts are also given in Reference Table A-13. For TRACE and WAIT, the Q-Bit only enables the interrupt, but actually establishes it as pending. TRACE has the highest priority; WAIT is the lowest.

Reference:

[1]Topics 8.1 and 8.2

```
LOCN   ISN   LAF    OPF      REF

             INTL   BRSET    WAIT,INTO,INT1,INT2,INT3,PARITY,OPRINT,
                             MONTIM,STOP,PANEL,CCIO,RTCLCK,LOAD,
                             RESTRT,PFAIL,TRACE

             FDL    BRSET    FD0,FD1,FD2,FD3,FD4,FD5,FD6,FD7,FD8
                             FD9,FDA,FDB,FDC,FDD,FDE,FDF

             FDL.   BRSET    FD0.,FD1.,FD2.,FD3.,FD4.,FD5.,FD6.,FD7.,
                             FD8.,FD9.,FDA.,FDB.,FDC.,FDD.,FDE.,FDF.

                    PLACE    000+

000+   7     INTV   VECTOR   INT+,INTL

                    PLACE    010-

010-   8     NEXT   VECTOR   FDB,FDL.,FDL
```

Figure A. Sample FDB and INT Vectors with Associated Mechanisms

8.5 SECOND-DIGIT BRANCHING VECTORS

A Second-Digit Branch (SDB) Vector is similar to a DATA Vector, except for type-code SDB. Addressing in Control Memory resulting from an SDB Vector is of mode CD6.

Mode CD6, like mode CD4 of the DATA Vector, allows any address in the current page to be the target. The SDB Vector location should begin at an address ending in digit 0; this CAP32 assembler automatically arranges this, so no PLACE pseudoinstruction is required with SDB Vectors.

SDB Vectors may be one or two label-lists. In either case the index that determines where the target is in the label-list is SD, bits ISAVE.3:0, as listed in Reference Table A-12.

An SDB Vector will normally be used within a routine to which control has been transferred to by a prior FDB Vector, since FDB execution is required to place IBY into ISAVE, thus making an SDB branch possible. An SDB normally provides the second level of jump decoding for a particularly complex set of instruction codes. At SDB time the next instruction byte is already on its way to (or present in) IBY. If that next instruction byte is a parameter, it may be transferred over the IBUS and the BBUS through use of I-type codes. It can be gated, masked, and shifted in the ALU, as needed. On the other hand, if the digits of SD (ISAVE.3:0) are a parameter that should not just be a multiway jump, SD may then be transferred over the IBUS and the BBUS to the ALU through use of OPF1 code SDX.

Eventually, by using I and I2 in most cases, the byte IBY is the first byte of the next instruction. Then FDB normally is used. If, however, FDB is unsuitable for decoding, the first instruction byte must be preserved by using an IH or I2H code in OPF1. Then FDB must be used to service interrupts. If a jump is not desired, the WAIT interrupt may be set by setting Q.0. Since WAIT is of lowest priority, other pending interrupts will be serviced first. A WAIT interrupt routine is now written which serves as the start of your actual decoding. The FDB is never actually executed. Normally, an FDB is a good way to begin the decoding of an instruction.

Figure A shows an example that starts after the FDB of an instruction decode, and then increments the contents of a secondary Local Memory register, based on bits SD.3:0 as a selection index.

Figure B illustrates an example that starts after the FDB of an instruction decode, and interprets the second and third bytes of the instruction as a byte address in the main memory, with the upper two bytes of the address (used as M.17:16) given by ISAVE.1:0. The addressed byte is fetched from the main memory, and placed in the lower half of the Y-Register.

The example in Figure C shows an instruction fetch cycle that sets flag GI.3 based on bit IBY.7 (1 for 1 and 0 for 0), and then branches to one of eight routines based on the value in bits IBY.6:4.

```
OPF                         REF

SDX                         Z
ZB                          Z
LM(Z4)                      Y
Y+1                         LM(Z4)
```

Figure A. Sample Routine Starting After FDB of Instruction Decode (See text)

```
OPF                         REF

I                           XU
SDX                         Z
ZB                          FXB
I                           XL:F
X                           MR
DB                          Y
```

Figure B. Sample Routine Starting After FDB of Instruction Decode (See text)

```
LAF            OPF            REF

FDL            BRSET          FD0,FD1,FD2,FD3,FD4,FD5,FD6,FD7
                              FD8,FD9,FDA,FDB,FDC,FDD,FDE,FDF
               PLACE          010-
FDVEC          VECTOR         FDB,FDL
               .
               .
               .
               IH             Y
               GIW            Z
               ZB             Z
               Z&7            Z,Y15(,*+2)
               Y              .,FDVEC
               .
               .
               .
FD1            opf            ref
               .
               .
               .
FD7            opf            ref
FD8            Y              .,FD1
FD9            Y              .,FD2
               .
               .
               .
FDF            Y              .,FD7
```

Figure C. Sample Instruction Fetch Cycle (See text)

8.6 FURTHER CAP32 PSEUDOINSTRUCTIONS

In addition to those previously discussed in this section, a number of CAP32 pseudoinstructions are available. Pseudoinstructions are directives to the CAP32 microassembler and do not result in the generation of object microcode.

PAGE Pseudoinstruction

The PAGE pseudoinstruction assigns a name to the current block (512 words maximum) of microcode to be assembled, and makes an absolute assignment to a particular page of Control Memory. The syntax of this pseudoinstruction is as follows:

LAF	OPF	REF
name	PAGE	startloc,endloc

where "name" is the name desired for the microcode block; where "startloc" is the absolute address of the beginning of the block in Control Memory; and where "endloc" is the absolute address of the end of block in Control Memory (which must be in the same page as "startloc"). If omitted, "endloc" is assumed to be at the end of the current page. The following illustrates an example:

```
M32D      PAGE        300-,3C7+
```

PLACE Pseudoinstruction

The PLACE pseudoinstruction assigns an absolute location to the executable statement or the VECTOR definition which immediately follows the PLACE statement. The syntax is:

OPF	REF
PLACE	loc

where "loc" is an absolute location in the current page. An example places vector NEXT at location 010- is shown below:

```
          PLACE       010-
NEXT      VECTOR      PDB,FDX.,FDX
```

END Pseudoinstruction

An END placed in field OPF terminates the assembly.

EXTERN Pseudoinstruction

The EXTERN pseudoinstruction provides linkage to other pages of the microcode. CAP32 provides no automatic dynamic linkage capability.

The programmer must know the absolute location of the variable to be referenced in the other page. The syntax is:

LAF	OPF	REF
symbol	EXTERN	LOC

where "symbol" is the name of the variable defined in the other page, and "loc" is its absolute location. For example:

```
     STF        EXTERN       085-
```

When the "symbol" is the name of a vector defined by a VECTOR pseudoinstruction in another page, then the Vector type (FDB or INT only) must be given following "loc" (complete with + or -, if appropriate).

LAF	OPF	REF
vector-name	EXTERN	loc,type

For example:

```
     NEXT       EXTERN       000+,FDB
```

REGNAM Pseudoinstruction

Local symbol definition is provided by the REGNAM pseudoinstruction. The syntax is:

LAF	OPF	REF
symbol	REGNAM	value

where "value" is a decimal number in the range 0:15. "Symbol" is usually a Local Memory array register name when used with REGNAM. Symbol can then be used as rn in any executable instruction where rn is applicable (see Reference Table A-1). For example:

LAF	OPF	REF	COF
DAR	REGNAM	8	DISPLAY ADDRESS REGISTER
SB	REGNAM	13	STACK BASE

Note that REGNAM assigns a numeric value to a symbol used to reference a Local Memory register, but has no control whatsoever over the Local Memory mode (primary or secondary). The programmer must set the mode correctly so that the symbol actually refers to the register it mnemonically suggests. In the example, Stack Base (SB) is singularly primary register 13, yet LMS(SB,LG) refers to secondary register 13.

BRSET and VECTOR Pseudoinstructions

These pseudoinstructions have previously been defined.[1]

Reference:

[1]Topic 8.1

9 STACK PROCESSING

9.1 STACK PROCESSING IN MAIN MEMORY ONLY

Stack processing on the 32/S is supported by special hardware features and the CAP32 microcommand structure. Before discussing the special features, however, it is necessary to discuss the simplest form of stack processing, with the stack entirely in Main Memory.

A Main Memory stack begins at a location called the Stack Base (SB). Stack elements are single words of two bytes each. As elements are pushed onto the head of the stack, they are stored at every increasing addresses. A Stack Pointer (SP) points to the word currently at the top of the stack (TOS). The memory area reserved for the stack is bounded by an upper limit called the Stack Limit (SL). The SL is equal to the largest allowable value for SP. When SP attempts to exceed SL, a stack overflow condition will result.

To push a word onto the main memory stack, SP must first be incremented by 2 (since memory addressing is in bytes); thus the new word is stored at M(SP). The new word is now TOS, and the former TOS word is now TOS1, at M(SP-2). If the new word to be pushed onto the stack is in Y, then the CAP32 sequence shown in Figure A should be used. Note that SP in the OPF field as an argument of LM has a value equal to the index of SP in Local Memory. This sequence assumes the upper two bits of the memory address are properly in FX, or that SP is stored in Local Memory so it will call out the correct upper two bits from SX or P.17:16.[1] The SL is ignored.

Removing a word from the top of the stack is called a pop operation. To pop TOS, read it into Y and then decrement SP by 2. The CAP32 sequence for this operation is shown in Figure B.

As shown, the routines for pushing and popping a word at the top of the memory stack are straightforward, yet always involve a memory reference. Also, there are no provisions for checking for stack overflow. A stack overflow check is complex and time consuming. Popping TOS will never induce overflow (underflow is ignored). A push operation could cause an attempt by SP to exceed SL. Figure C shows a push operation which tests for stack overflow. This test has increased the push routine from 3 words to 6. Also the stack overflow routine (STKOV) is paired with ISN 25, because it was jumped to from a test and branch, rather than from a single jump.

Reference:

[1]Topic 4.6

OPF	REF	COF
LM(SP,LG)	Z	FETCH SP INTO Z.
Z+2	MW:LM(G)	INCREMENT, USE, AND RESTORE SP.
Y	D	WRITE NEW TOS WORD TO MEMORY

Figure A. CAP32 Sequence to Push Word From Y Onto Stack

OPF	REF	COF
LM(SP,LG)	Z:MR	FETCH SP, SET G, REQ MONOBUS.
Z-2	LM(G)	DECREMENT AND RESTORE SP.
D	Y	RECEIVE TOS WORD FROM MEMORY.

Figure B. CAP32 Sequence to Pop TOS

ISN	LAF	OPF	REF	COF
21	PSOV	LM(SP,LG)	Z	FETCH SP, SET G.
22		LM(SL)	X	FETCH SL.
23		Z-X	F	COMPARE SP-SL.
24		Y	.,NNEG(STKOV)	IF SP > SL, GO STKOV.
25		Z+2	MW:Z:LM(G)	NO OVFL: INCRMNT,USE SAVE SP.
26		Y	D	WRITE NEW TOS WORD TO MEMORY.
27	STKOV	opf	ref	STACK OVFL, PAIRED WITH 25.

Figure C. CAP32 Push Sequence With Overflow Test

9.2 STACK ORGANIZATION: THE T-COUNTER AND N-STATS

The 3200 hardware supports a stack organization consisting of a main stack body in Main Memory, with an active Stack Head of from 0 to 5 words in the Y-Register and primary Local Memory registers 0:3.

When the Stack Head is empty, the stack is fully in Main Memory, and can be used as a simple stack (described in the previous topic) with the TOS word pointed to by the Stack Pointer (SP). When the Stack Head is not empty, the TOS word is stored in the Y-Register. There then may be from 0 to 4 additional words stored in Local Memory at locations called LT, LW, LV, and LU (respectively), working down the stack from TOS. It is also convenient to name the stack words TOS, TOS1, TOS2, etc. from the top of stack down. For a full Stack Head, the correspondence of these names is shown in Figure A. If the Stack Head contains only 4 words, then LU is missing. For 3 words, LV is also missing, and so on. For the minimum non-empty Stack Head (1 word) the locations used are Y, M(SP), M(SP-2), etc.

The absolute locations of Stack Head words LT, LW, LV, and LU are generally unimportant to the programmer; what is important is that the T-Counter always points to LT when the stack is in a stable state; i.e., LT = LM(T). Furthermore, (T+1) mod 4 points to LU; i.e., if 1 is added to T (with 4 reset to 0 when it occurs), then LU is located at LM(T+1) mod 4. For example, the correspondence between the virtual names and absolute locations for these registers might be as shown in Figure B.

The programmer can address LM(T) or LM(U) either as a source or as a destination; mod-4 arithmetic on the T-Counter is available. Thus, access to Local Memory Stack Head words is in the cyclic virtual address space T, U, V, or W (not in the normal absolute index address space of Local Memory). That is why the absolute Local Memory address assignment is unimportant.

Since the Stack Head need not always be full, a device is needed to indicate the number of words in the Stack Head, and also to indicate when the T-Counter has been moved away from its stable-state location (pointing to LT). A unique device, the 5-bit N-Stats shift register, is provided for these purposes. When the stack is in a stable state, the left end of N-Stats has a number of 1-bits equal to the number of words in the Stack Head. For example, a full Stack Head has N = 11111; a Stack Head with words in Y and LT has N = 11000; and an empty Stack Head shows N = 00000.

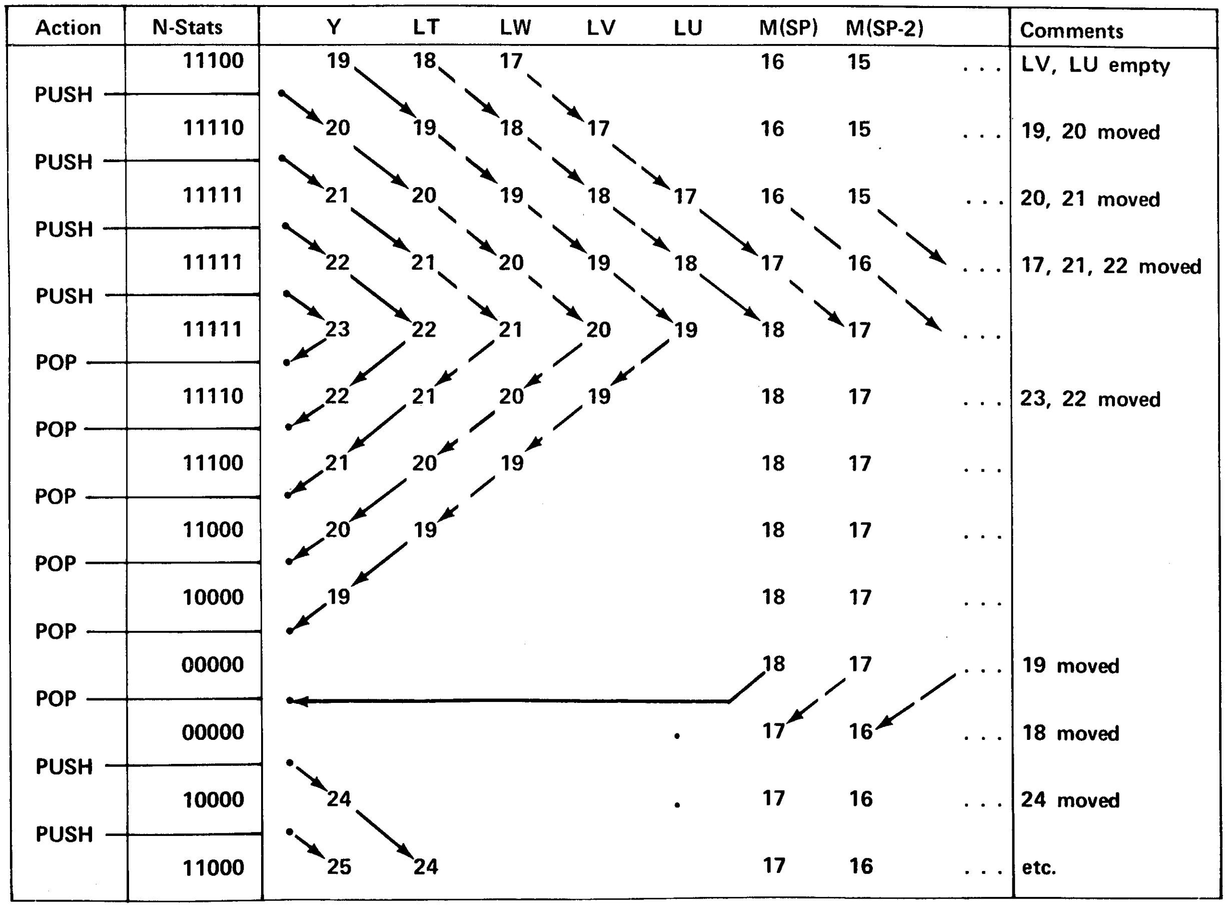
Figure C shows a sequence of push and pop operations, and the effect they produce on the Stack Head and the N-Stats Register. Data moves shown by solid arrows are real; the words are actually moved into and out of Y, between Y and LT, from LU to M(SP), and out of M(SP). The moves shown by dashed arrows are virtual, caused by changing the values of the two pointers (T for Local Memory, and SP for Main Memory). In the first push shown, T is incremented and points to the former LU register, then Y is moved to where T now points. The former LT is now LW, etc., and data words 18 and 17 have not moved, but have been renamed. Similarly, on the fourth push, SP is incremented by 2.

TOS	TOS1	TOS2	TOS3	TOS4	TOS5	TOS6	...
Y	LT	LW	LV	LU	M(SP)	M(SP-2)	...
← Stack Head →					← Memory Stack → ...		

Figure A. Nomenclature for a Full Stack Head

LT	LW	LV	LU
LM(2)	LM(1)	LM(0)	LM(3)

Figure B. One of Possible Stack Head Local Memory Assignments



Action	N-Stats	Y	LT	LW	LV	LU	M(SP)	M(SP-2)		Comments
	11100	19	18	17			16	15	. . .	LV, LU empty
PUSH										
	11110	20	19	18	17		16	15	. . .	19, 20 moved
PUSH										
	11111	21	20	19	18	17	16	15	. . .	20, 21 moved
PUSH										
	11111	22	21	20	19	18	17	16	. . .	17, 21, 22 moved
PUSH										
	11111	23	22	21	20	19	18	17	. . .	
POP										
	11110	22	21	20	19		18	17	. . .	23, 22 moved
POP										
	11100	21	20	19			18	17	. . .	
POP										
	11000	20	19				18	17	. . .	
POP										
	10000	19					18	17	. . .	
POP										
	00000						18	17	. . .	19 moved
POP										
	00000						17	16	. . .	18 moved
PUSH										
	10000	24					17	16	. . .	24 moved
PUSH										
	11000	25	24				17	16	. . .	etc.

Figure C. Effects of a Sequence of Push and Pop Operations on the Stack Head and N-Stats

9.3 SIMPLE STACK OPERATIONS IN AN ACTIVE STACK HEAD

This topic describes the use of the eight available Stack Head manipulation command codes. These codes are summarized in Reference Table A-14.

A push can involve many different actions. If the Stack Head is empty, the source is moved to Y, and N is shifted right with fill of 1. If the Stack Head is partially full, then T is incremented, N is shifted right with fill of 1, Y is stored at the new LT position (old LU), and finally the source is moved to Y. If the Stack Head is full, the foregoing action must be extended to save the old LU, add 2 to SP, and then store the old LU at the new M(SP).

Figure A shows a subroutine to push X onto the stack. ISN 101 through 103 is a sequence to read LU and push it to memory if the Stack Head is full. If N5 fails, the Stack Head is not full and control goes to ISN 104, with the main action of 101 wasted. If N5 succeeds, the Stack Pointer (SP) is advanced 2 bytes and LU is stored at the new M(SP). No change is made in T or N because of what follows. ISN 104 advances T by 1 mod 4 (thus making old LU new LT), shifts N right 1 with fill of 1, and moves Y to new LT; hence 104 is "push Y to LM". ISN 105 copies X to Y, thus completing the over-all "push X to stack" complies.

A similar subroutine which pops the stack to X is shown in Figure B.

Finally, a subroutine to read TOS to X is shown in Figure C. This routine establishes TOS in Y if it is not already there. This routine establishes TOS in Y if it is not already there. This routine is not a pop since the TOS value and the stack length remain the same.

The reader should note that the Nx tests, as defined in Reference Table A-11, succeed if the named bit of the N-Stats register is equal to 1.

LOCN		+ SUCC	-	ISN	LAF	OPF	REF	COF
	*	104	102	101	PSX	LM(SP,LG)	Z,N5(,PSX4)	GET SP; TEST N FOR FULL HEAD.
xxx+	*		103	102		Z+2	MW:LM(G)	ADVANCE SP; START MONOBUS.
		104		103		LM(U)	D	READ LU, PUSH TO MEMORY.
xxx-			105	104	PSX4	Y	LM(U,IT)	SR(1) N, T+1; PUSH Y TO LT.
				105		X	Y,SO(=S)	COPY X TO Y; RETURN.

Figure A. A Subroutine to Push X Onto the Stack

LOCN		+ SUCC	-	ISN	LAF	OPF	REF	COF
		203	202	201	POX	Y	X,N1(,POX3)	COPY Y TO X; TEST N NOT EMPTY.
xxx+				202		LM(T,DT)	Y,SO(=S)	POP LT; SL(0) N, T-1; RTN.
xxx-	*		204	203	POX3	LM(SP,LG)	Z:MR	GET & SEND SP; START MONOBUS.
	*	205		204		Z-2	LM(G)	RETARD SP, SAVE IT.
				205		D	X,SO(=S)	POP MEMORY TO X; RETURN.

Figure B. A Subroutine to Pop Stack to X

LOCN		+ SUCC	-	ISN	LAF	OPF	REF	COF
		303	302	301	RTXY	Y	.,N1(,RTXY3)	TEST FOR N EMPTY.
xxx+				302	RTXY2	Y	X,SO(=S)	COPY Y TO X; RETURN.
xxx-	*		304	303	RTXY3	LM(SP,LG)	Z:MR	GET & SEND SP; START MONOBUS.
	*	305		304		Z-2	LM(G)	RETARD SP AND SAVE.
			302	305		D	Y,RTXY2,LT	POP MEMORY, PUSH TO Y; SR(1) N.

Figure C. A Subroutine to Read TOS to X and Establish Y as TOS if Necessary

9.4 SPECIAL STACK HEAD PROCESSES

This topic discusses special Stack Head loading and unloading processes.

An active Stack Head involving the Y-Register and up to 4 Local Memory registers serves as a speed and efficiency producer during long algebraic or logical expansions. However, when the CPU is required for some other task (possibly involving some other stack), it is necessary to unload the Stack Head to memory, so that the stack can be saved for future use. The Stack Head may be of any length in range 0:5 and must be unloaded from the bottom, since that is the way it fits on top of stack memory. If the Stack Head tests to be of length 0 or 1, the process is trivial. At length 2, a choice of methods is available. Lengths 3:5 dictate the standard way, which for compact implementation is also used for length 2. This is illustrated in the subroutine in Figure A.

The execution time for the subroutine shown in Figure A is dependent on the Stack Head length:

Stack Head length:	0	1	2	3	4	5
Execution time:	3	4+1MW	10+2MW	12+3MW	14+4MW	16+5MW

For loading, five words are popped off the memory part of the stack to form a full Stack Head. Command form "LM(T,DT)" in REF1 and also form "DT" in REF3 are designed for the loading operation. The question arises of when this should be done. Normally a Stack Head is "grown" item-by-item with push operations. A full stack is rarely as effective as a partially full one.

LOCN		+ SUCC -		ISN	LAF	OPF	REF	COF
	*	504	502	501	STUF	LM(SP,LG)	Z,N2(STUF4)	IF > 1 IN HEAD, GO STUF4.
uuu-		510	503	502		LM(T,DT)	.,N1(STUFA)	IF Y IN HEAD, CLEAR N & GO STUFA.
vvv-				503		Y	.,SQ(=S)	HEAD EMPTY, RETURN.
uuu+		506	505	504	STUF4	LM(U,IT)	X,N5(STUF6)	GET LU; IF HEAD FULL GO STUF6.
www-		506	505	505		LM(U,IT)	X,N5(,*)	GET LU UNTIL ITS LB; SHIFT N RT.
www+	*		507	506	STUF6	Z+2	MW:Z:	ADVANCE.
		508	509	507		X	D,N4(,STUF9)	UNLOAD ITEM; IF TOS, GO STUF9.
xxx+		506		508		LM(U,IT)	X,STUF6	GET LU, SHIFT N RIGHT.
xxx-		510		509	STUF9	LM(U,IT)		CLEAR N BY SR(0) N.
vvv+	*		511	510	STUFA	Z+2	MW:Z:LM(G)	ADVANCE & SAVE SP.
				511		Y	D,SO(=S)	WRITE Y TO MEMORY AT SP.

Figure A. Subroutine to Unload Stack Head of Any Length to Memory

9.5 STATE DIAGRAM

This topic presents a stack processing state diagram.

An over-all perspective of the 3200 stack handling capability is given in the state diagram of Figure B. The states pictured are states of the Stack Head, not of the full stack. Each Stack Head state has a value of N associated with it, and some actions that cause a change of state. (In some cases, the actions cause some change of content, but no change of state.)

It is essential to understand the distinction between stable and transient states. The stable Stack Head states are those with N all zeros or N1=1, in which the Stack Head contents are marked by the number of 1's at the left of N (as to the number of elements in the head), and where the T-Counter points to TOS1 in LM. Transient states occur during the load or unload process, in which the 1's of N are moved to the right (in unload action), or have not arrived at the left (in load action), and where T points to the location currently being unloaded or unloaded at the bottom of the Stack Head.

In Figure A, the states marked "U" and the states marked "S" are transient in nature, as intermediates in the load process (which ends stably at H5), or in the unload process (which ends stably at H0). H0:H5 are the stable states of indefinite duration.

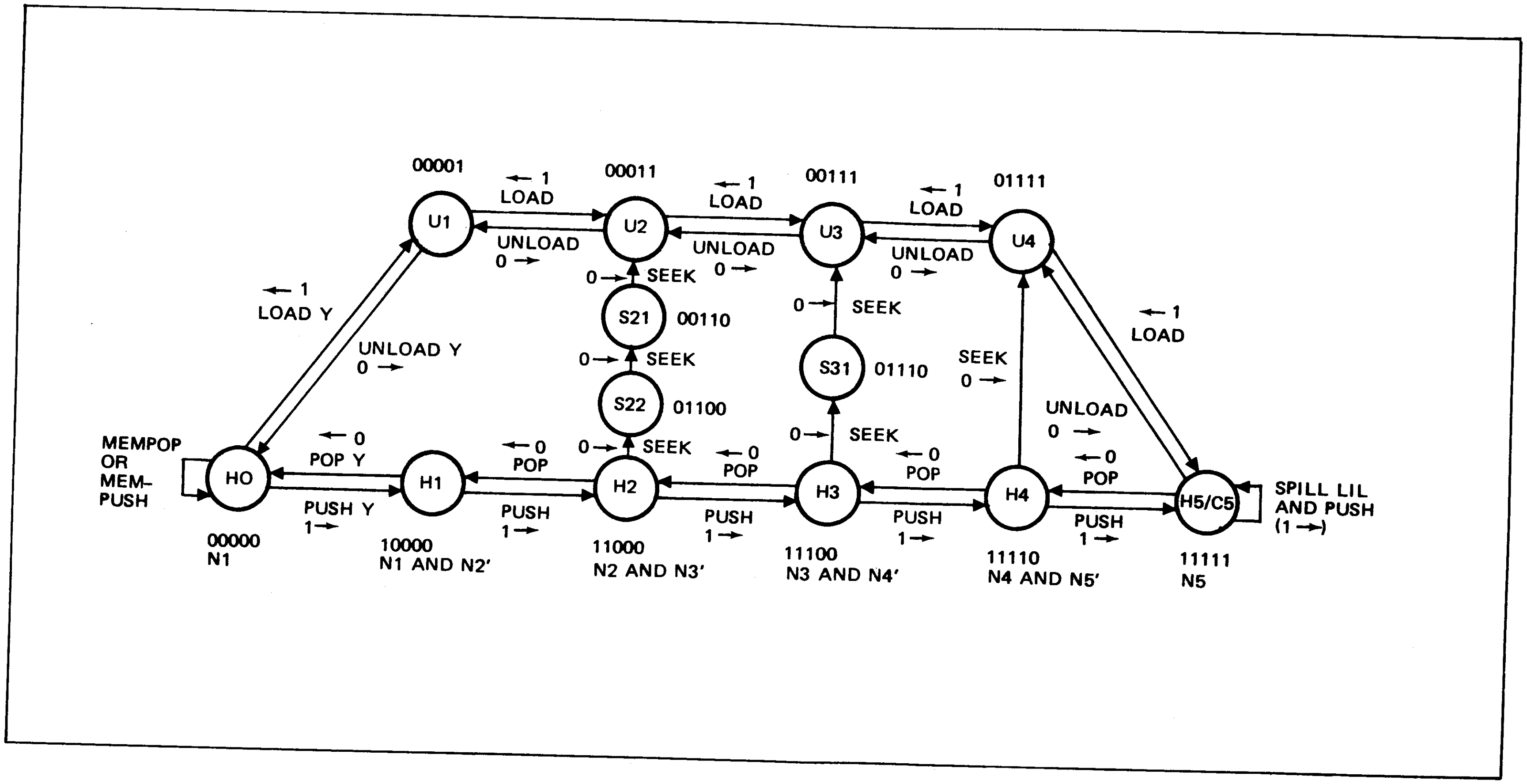


Figure A. State Diagram of the 3200 Stack Head

10 INPUT/OUTPUT MICROPROGRAMMING

10.1 INPUT/OUTPUT CONTROLLERS

Input/Output (I/O) microprogramming depends critically on the design of various I/O device controllers which can be attached to a system. In the 3200, the I/O microprogramming situation is eased by the existence of a general pattern into which all I/O device controllers must fit, and by the use of MONOBUS as the main communication medium to and from the I/O devices.

I/O device controllers fall into two main categories: active and passive. An active controller contains logic and circuitry needed to request MONOBUS control for data or control transfers. A passive controller cannot request the MONOBUS, but can respond to a MONOBUS request signal bearing its MONOBUS address. The principal active controllers are of CPU or Direct Memory Access (DMA) type. Discussion here is limited to a system with one CPU, in which that CPU can request a single transfer to or from a passive I/O controller. The CPU can also request a DMA controller to carry out an entire sequence of memory accesses via the MONOBUS, without further CPU intervention or control until the access sequence is complete and an appropriate interrupt signal is sent.

Each I/O controller is accessed through a Device Register Block (DRB), which is 8 words (16 bytes) long and has a MONOBUS Address (MBA) in the range 3C000:3FFFEF. (MBA range 3FFF0:3FFFF is reserved.) The general format of a DRB is shown in Figure A. Variations from the general format may be expected to satisfy special device needs, but such variations must be within the scope of the available firmware.

User device operations generally consist of four phases:

- Observation of device status by reading word 0 (and possibly word 3) of the DRB, to see if access is possible at the moment.
- Establishment of DRB values that should apply to the access being initiated, ending with establishment of an appropriate I/O order (command in word DRB(1)).
- Determination of device access completion, through the interrupt mechanism.
- Reading of input data (if any) from word DRB(2); also reading of any error data from word DRB(0).

At the simplest firmware design level, all the above sequencing is pushed up into software (at the user level), and the firmware simply reads and writes words from and to the DRB. Firmware for such action may already exist in adequate measure. More elaborate machine design will move some of the above sequencing down to the firmware level, reducing the number of separate software instructions the user must employ to accomplish an access.

To understand the above decisions, three reference tables on I/O programming are included in the Appendix. Reference Table A-15 details the bits of the Status Word (DRB(0)). Reference Table A-16 describes the Order Byte (DRB(1).7:0). Reference Table A-17 lists the mode controls of DRB(1).15:8.

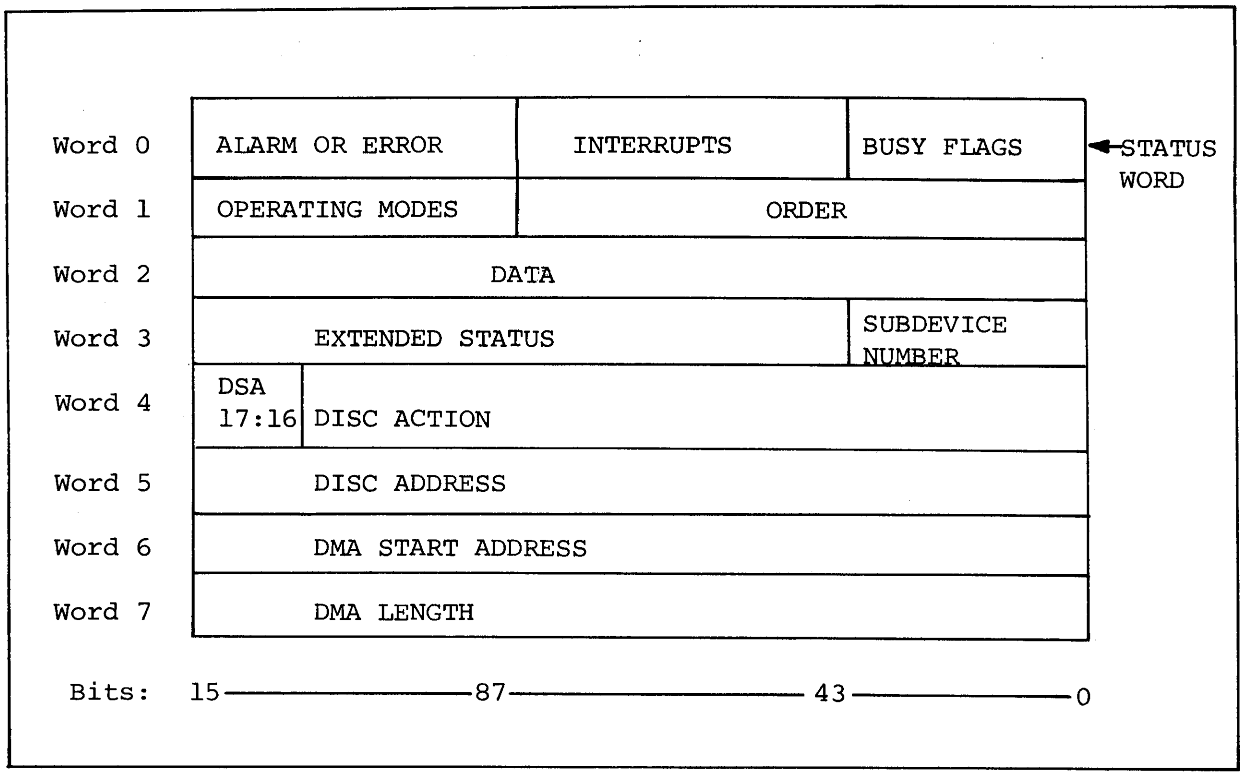


Figure A. General Format of a Device Register Block (DRB)

10.2 INTERRUPT SYSTEM

This topic discusses the 3200 interrupt system.

A device external to the CPU (i.e., an I/O device) may interrupt the CPU by pulling one of the following backplane signals low:

- INR0/ (External Interrupt 0)
- INR1/ (External Interrupt 1)
- INR2/ (External Interrupt 2)
- CIOR/ (Concurrent I/O Interrupt)

This will eventually cause an action request and a procedure code branch in the CPU. The CPU will read the appropriate interrupt Response Word (see Figure A) to determine which device is causing the interrupt. As a side effect of this read, the interrupt select daisychain will be activated. Upon receipt of the Interrupt Select in (ISIX/), the external device checks the four low-order MONOBUS address bits. If these bits correspond to the interrupt (each Interrupt Response Word has unique four low-order bits) the device is causing, then the device gates its device address onto the D-Bus, replies, and ceases driving the backplane interrupt signal. Otherwise, the device must drive Interrupt Select Out (ISOX/) low.

INTERRUPT	BACKPLANE SIGNAL	MONOBUS ADDRESS OF RESPONSE WORD
External Interrupt 0	INR0/	3FFF0
External Interrupt 1	INR1/	3FFF2
External Interrupt 2	INR2/	3FFF4
Concurrent I/O Interrupt	CIOR/	3FFF8

Figure A. Interrupt Response Words

11 CONTROL PANELS

11.1 BASIC PANEL

This topic describes the 3200 Basic Panel.

The 3200 Basic Panel is illustrated in Figure A. This panel has only three controls: an eight position rotary key-switch, and two push-button switches. The key-switch has six labeled positions, as shown below.

Position	Function
OFF	Causes the backplane signal PSC1/ to go low, which (in turn) causes power supply shutoff. (This includes power for the MOS memories.)
HOLD	Causes the backplane signal PSC2/ to go low, which (in turn) causes a Power Fail interrupt in the CPU and causes the power supply to go to standby. This means that +5V, +12V, and -12V are off, but the MOS memory power (+21V and +5.2V) is on and the refresh logic is running. (The unlabeled position between HOLD and OFF is wired to HOLD.)
STOP	Causes the backplane signal STOP/ to go low, which (in turn) causes a Stop interrupt in the CPU and enables the LOAD push-button switch. There are two positions labeled STOP, each performing the same function.
RUN	This position does basically nothing. The power supply is on and the CPU is running the default condition. (The unlabeled position between RUN and LOCK is wired to RUN.)
LOCK	Causes the backplane signal LOCK/ to go low. This has implications only if the Maintenance Panel is installed.

The key-switch positions are summarized in Reference Table A-18.

The two push-button switches are as follows:

Switch	Function
LOAD	If the backplane signal STOP/ is low, pressing this switch causes the backplane signal LOAD/ to go low and creates a Load interrupt in the CPU.
INT	Pressing this switch causes the backplane signal OINT/ to go low and creates an Operator interrupt in the CPU.

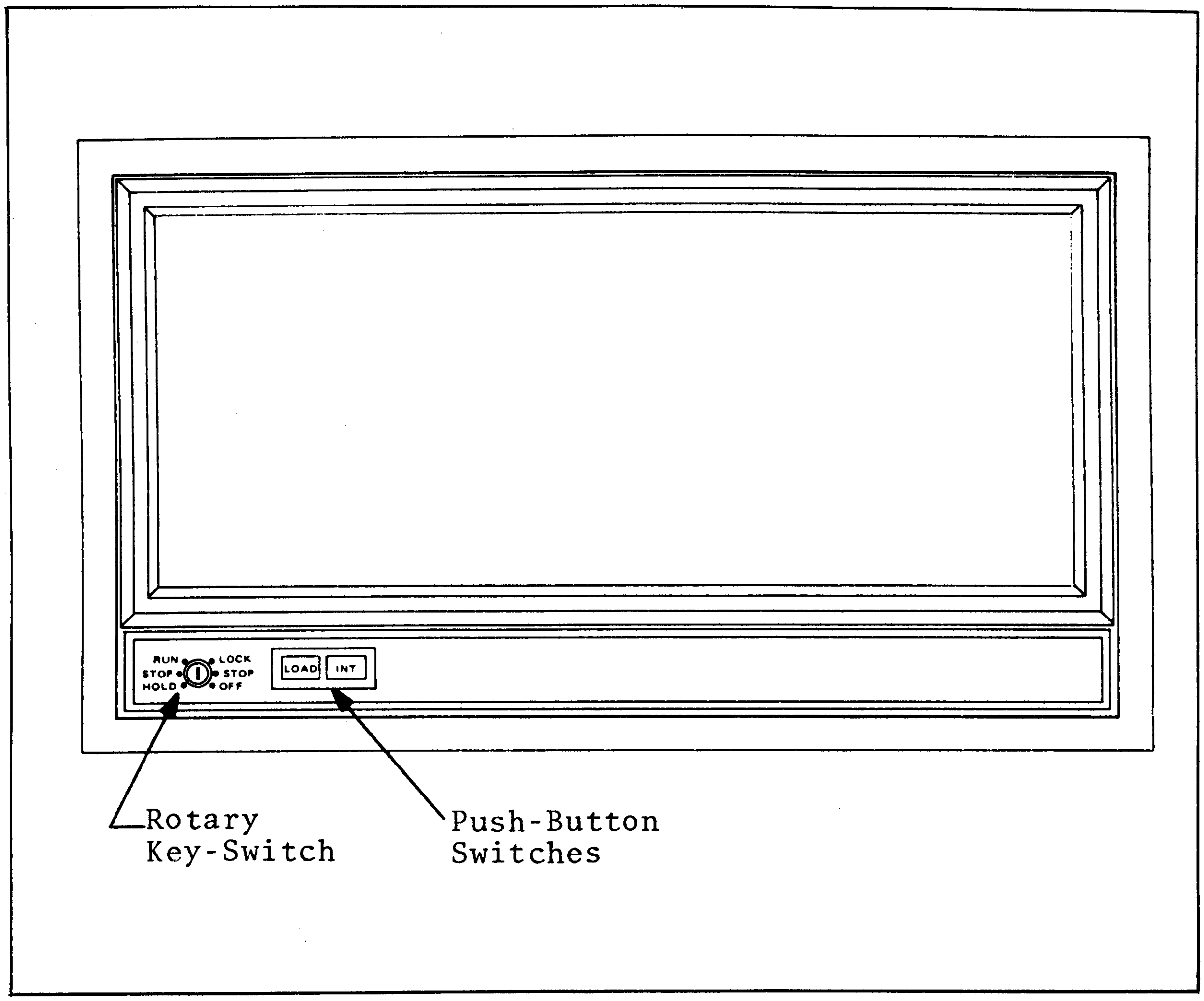


Figure A. Basic Panel

11.2 MAINTENANCE PANEL

The 3200 Maintenance Panel has two main components: the Maintenance Panel Logic Board (MPLB) and the Maintenance Panel Switch Board (MPSB).

The MPLB plugs into any card slot in the backplane. In a multi-chassis system the MPLB must be plugged into the same chassis as the CPU. In a zoned chassis the MPLB must be in the same zone as the CPU.

The MPSB is mounted in a Basic Panel, as shown in Figure A. The MPSB is not directly connected to the Basic Panel electronics. The MPSB is connected to the MPLB by a ribbon cable that runs from the top edge of both boards.

The Maintenance Panel will not function without a free-running CPU clock (backplane signal CLK2/).

There are six functional areas of interest to a user of the Maintenance Panel. They are as follows:

- Hardware Display Selector Switches
- Firmware Display Selector Switches
- Address Display
- Data Display
- Data Switches
- Indicator Display
- Control Switches

These areas are discussed in the remaining topic of this section.

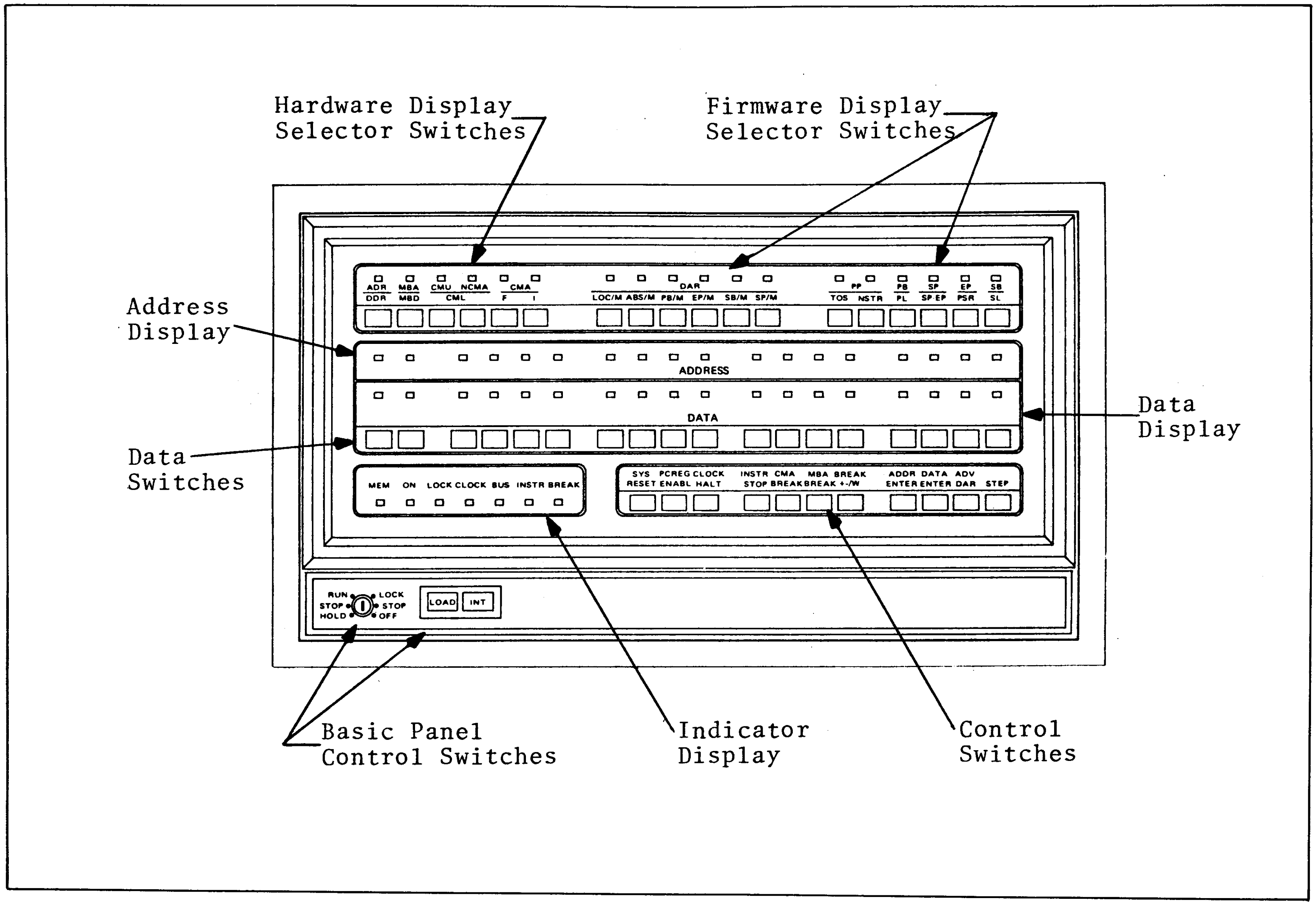


Figure A. Maintenance Panel

11.3 MAINTENANCE PANEL CONTROLS AND INDICATORS

This topic discusses the hardware display selector switches, the indicator switches, and the control switches.

Hardware Display Selector Switches

The displays selected by the six extreme left display selector switches are driven by hardware contained within the MPLB. These displays require no CPU support, and therefore the MPLB generates no panel interrupts. These six switches are summarized in Reference Table A-21, and are also described below.

Switch	Function
ADR/DDR	Displays the contents of the ADR and DDR. This display is used to "freeze" one of the firmware driven displays.
MBA/MBD	Displays the Address, Data, and Write flags of the last selected MONOBUS cycle. If the MBA Break switch is on (down), then only MONOBUS cycles that meet the break condition are selected.[1] The Write Even Byte Flag (WEBF/) is bit 17 of the data display, and the Write Odd Byte Flag (WOBF/) is bit 16. This is the only time these bits of the data display are used. The information for MBA/MBD display is only stored in the ADR and DDR while any hardware display except ADR/DDR is selected.
CMU/CML	Displays the upper 16 bits of the current Control Word and the lower 16 bits of the Control word.
NCMA/CML	Displays the next Control Word address in bit 14 through bit 4 of the address display, the plus/minus address bit in bit 0 of the address display, and the lower 16 bits of the Control Word in the data display.
CMA/F	Displays the address of the current Control Word in the address display; previous to the last CPU clock, this was the data in the NCMA/CML address display. The FBUS (FB15/ to FB00/) is in the data display.
CMA/I	Displays the address of the current Control Word in the address display, and the IBUS (IB15/ to IB00/) in the data display.

Indicator Displays

There are seven indicator displays. These displays are summarized in Reference Table A-19, and are discussed below.

Display	Function
MEM	Indicates the +21 and +5.2 voltages and the RFSH/ signal are being supplied to the MOS memories. As long as this indicator is on, the data stored in the MOS memories are preserved. If this light goes out, it means there has been a loss of power to the MOS memories and the data stored have been lost. If this occurs, the key-switch on the Basic Panel must be turned to the OFF position before the Power Supply will turn on again.

ON	Indicates the +5, +12, and -12 voltages are ON.
LOCK	Indicates the key-switch is in the lock position and the LOCK/ signal on the backplane is low.
CLOCK	Indicates the CPU clock (CLK1) is running.
BUS	Indicates MONOBUS cycles are occurring.
INSTR	Indicates software instructions are being executed.
BREAK	Indicates a break condition has been satisfied.[2]

Control Switches

There are 11 control switches; these switches are described in Reference Table A-20 and below:

Switch	Function
SYS RESET	When ON, causes a CPU master reset (RSET/ goes low). This function is disabled when the key-switch is in the LOCK position.
PCREG ENABL	When ON, enables the C-Register on the MPLB and disables all other C-Registers. This C-Register is loaded from the data switches with the ADD ENTER and DATA ENTER control switches. This function is disabled when the key-switch is in the LOCK position
CLOCK HALT	When ON, causes a timing hold in the CPU. The CPU clock (CLK1) stops and firmware execution ceases. The switch functions are disabled by an unsatisfied break condition or when the key-switch is in the LOCK position.
INST STOP	When ON, causes the backplane signal STOP/ to go low. This generates a Stop interrupt in the CPU, and execution of software instructions ceases. If the CLOCK HALT switch is OFF, then the function of this switch is disabled by an unsatisfied break condition. The switch functions are disabled when the key-switch is in the LOCK position.
CMA BREAK MBA BREAK BREAK + -/W	The action of these three switches is related to the generation of break conditions.[3]
ADDR ENTER	When ON, causes bit 2 of the Panel Status Word[4] to be a 1. If the PCREG ENABL switch is on, this causes the data switches to be copied into the upper 16 bits of the panel C-Register.
DATA ENTER	When ON, causes bit 1 of the Panel Status Word to be a 1. If the PCREG ENABL switch is on, the data switches will be copies into the lower 16 bits of the panel C-Register.
ADV DAR	When ON, causes bit 0 of the Panel Status Word to be a 1.
STEP	When ON, enables the Stop Break condition.

References:

[1]Topic 11.5
[2]Topic 11.5
[3]Topic 11.5
[4]Topic 11.4

11.4 MAINTENANCE PANEL FIRMWARE DISPLAY SELECTOR SWITCHES AND MONOBUS INTERFACE

This topic discusses the firmware display selector switches, and the Maintenance Panel MONOBUS Interface.

Firmware Display Selector Switches

The displays selected by the 12 extreme right display selector switches are driven by the CPU in response to panel interrupts. When one of these displays is selected, the MPLB displays the ADR and the DDR, and generates a Panel action/request (PANEL/) approximately every nine msec. In response to the panel action request, the CPU (controlled by internal firmware) reads the panel status and data switches, and writes the appropriate information back into the ADR and the DDR.

Maintenance Panel MONOBUS Interface

The Maintenance Panel has an interface to the MONOBUS that allows firmware control of some displays. The interface is nonstandard because it treats the MONOBUS address as a word address. (It is not possible to do byte reads or writes.) The interface responds to MONOBUS addresses in the range of $3FFFC_{16}$ to $3FFFF_{16}$. The assignments of these addresses are as follows:

Address	Assignment
3FFFC	Panel Status Word
3FFFD	Extension Word
3FFFE	Address Word
3FFFF	Data Word

These assignments are discussed further in Reference Table A-22. The format of the Panel Status Word is shown in Figure A. The bits of this word have the following significance:

Bit(s)	Description
Advance	This bit is set when the ADV DAR control switch is depressed. It is reset when the Panel Status Word is read or the switch is released.
Data Enter	This bit is set when the DATA ENTER switch is depressed. It is reset when the Panel Status Word is read or the switch is released.
Address Enter	This bit is set when the ADDR ENTER control switch is depressed. It is reset when the Panel Status Word is read or the switch is released.
Lock	When this bit is ON, the Basic Panel key-switch is in the LOCK position.
Display Code	This 4-bit code shows which display has been selected. Referring to the Display Selector Switches, the 6 on the left have a code of zero and the 12 on the right have (left to right) codes 1 to 12. Codes 13 to 15 are reserved for expansion.

Data Switch Extension	These 2 bits are (left to right) Data Switch 17 and Data Switch 16.

Reading the Panel Status Word resets the Panel Action Request. For further information regarding the Panel Status Word, see Reference Table A-23.

The format of the Extension Word is illustrated in Figure B. "Writes" to this word are used to specify the 2 high-order bits of the Address Display. Note that "writes" to this word cause the upper 2 bits of the display to be reset.

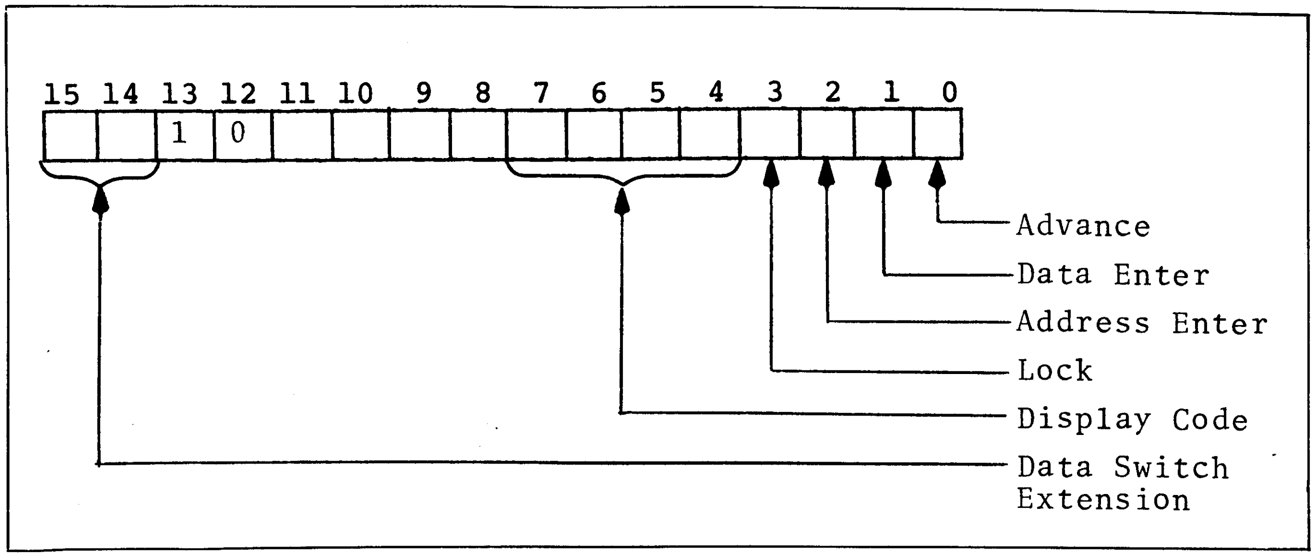


Figure A. Panel Status Word

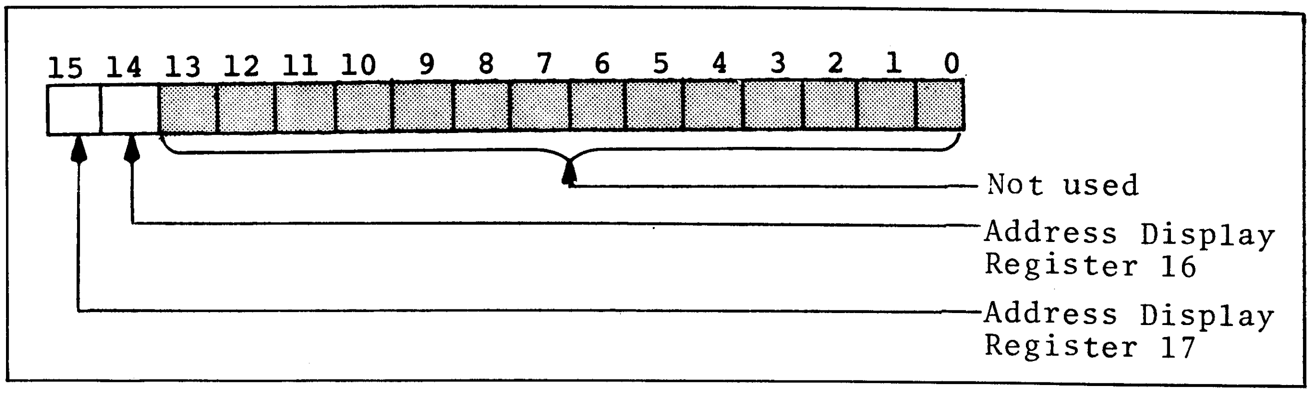


Figure B. Extension Word (Write Only)

11.5 MAINTENANCE PANEL BREAK CONDITION

Break Conditions are used to debug hardware, firmware, and software.

A Break Condition isolates a particular event occurring within the CPU and allows it to be used to control some MPLB action, resulting in some useful diagnostic information.

A Break Condition is enabled as a result of user Maintenance Panel action (e.g., turning on the MBA BREAK switch). At this time, an unsatisfied Break Condition is said to exist. When the selected event occurs, the specified diagnostic action takes place and a satisfied Break Condition is said to exist. To repeat this cycle, it is necessary to disable the Break Condition.

There are three types of Break Conditions:

- The Step Break Condition is enabled by depressing the STEP control switch. This condition is satisfied by the occurrence of a First Digit Branch.[1]

- The Control Memory Address (CMA) Break Condition is enabled when the CMA BREAK control switch is ON and the MBA BREAK control switch is OFF. It is satisfied by the occurrence of a particular Control Word address, selected by the Data Switches. The address is entered into the Data Switches in the same format as the CMA display. The location address goes in bit positions 14 to 4 of the Data Switches and the plus/minus address bit (if used) goes in bit 0. These Break conditions are reset by disabling the CMA Break or by the occurrence of a Step Break condition.

- The MONOBUS Address (MBA) Break Condition is enabled when the MBA BREAK switch is ON. It is satisfied by the occurrence of a particular MONOBUS address as selected by the Data Switches. The desired 18-bit MONOBUS address is entered on the Data Switches. These Break conditions are disabled by turning off the MBA BREAK switch, or by the occurrence of a Step Break condition.

References:

[1]Topic 8.4

CMA BREAK SWITCH	MBA BREAK SWITCH	+-/W SWITCH	BREAK FUNCTION
ON	OFF	OFF	Break on the selected C.M. address.
ON	OFF	ON	Break on the selected C.M. address, but disregard the $\pm$ address bit.
OFF	ON	OFF	Break on the selected monobus address.
OFF	ON	ON	Break on the selected monobus address, but only on writes.
ON	ON	OFF	Break on the selected monobus address, but disregard the low order address bit.
ON	ON	OFF	Break on the selected monobus address, but disregard the low order address bit and only on writes.

Figure A. Enabling Various Types of Breaks

11.6 MAINTENANCE PANEL INTERNAL LOGIC

This topic discusses the internal logic of the Maintenance Panel Logic Board (MPLB) and the Maintenance Panel Switch Board (MPSB).

The MPLB and MPSB are connected by a full duplex serial channel, which is controlled by hardware known as the Shift Control Logic. This logic is driven by the 135-nanosecond free-running CPU clock (CLK2/). The clock is divided down to provide a 60.44 usec shift clock (CKIN) and a 8.888 msec clock (CKPANEL), which start shift cycles and initiate Panel Action Requests, if necessary. 48 Data bits are shifted each way every shift cycle.

On the MPSB, the Control and Data Switches (and an encoded version of the Display Selector Switches) are clocked into the Input Shift Register (32 data bits and 16 pad bits) and are then shifted to the MPLB. Exceptions are the Clock Halt and Sys Reset Control Switches which are debounced and sent to the MPLB on dedicated lines. There the lower 16 Data Switches are clocked into the Data Switch Register; the rest of the data are clocked into the Control Register. The Control Register Output controls the different functional capabilities of the MPLB such as Panel C-Register and Break-Point Logic. The Control Register output also provides data for the Panel Status Word.

Concurrent with the action described in the preceding paragraph, the Display Code (a subset of the Control Register) feeds into the Display Code (a subset of the Control Register) feeds into the Display Control Logic which controls the Display Multiplexer. The dual 4-way by 16-bit Display Multiplexer is fed by all possible address and data display sources, and selects one of each source. These sources are clocked into the Output Shift Register with the data for the Indicator Display and the Display Code. These data are then shifted to the MPSB where they are clocked into the Holding Register.

The Holding Register output drives the Address Display, Data Display, Indicator Display, and Display Code Decoder. The Decoder drives the appropriate indicator above its corresponding Display Selector Switch.

Block diagrams of the MPSB and MPLB are presented in Figures A and B, respectively.

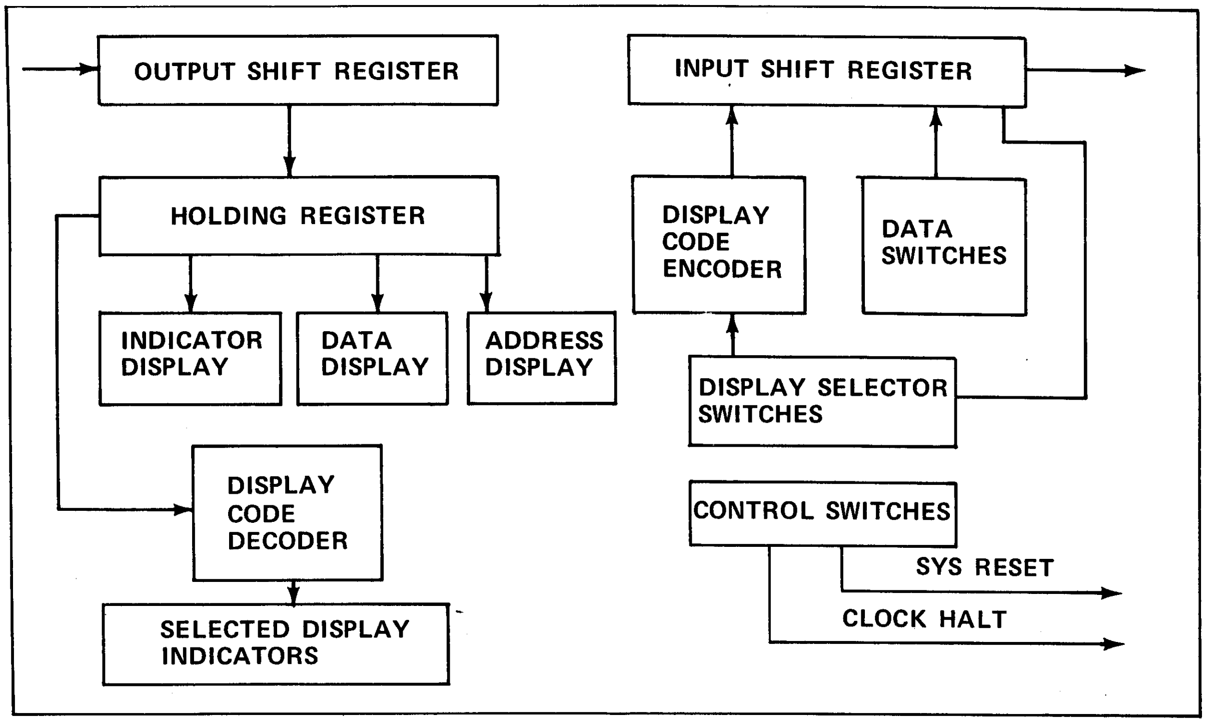


Figure A. Maintenance Panel Switchboard (MPSB)

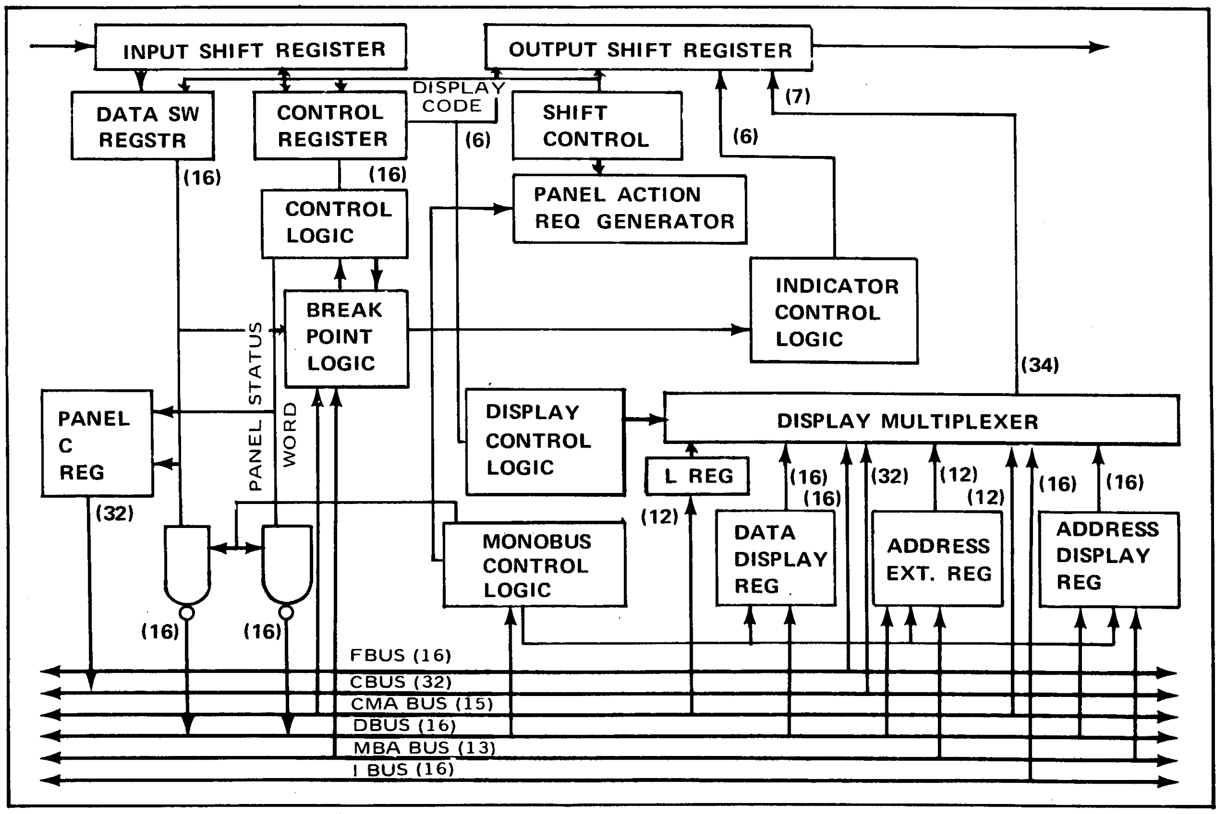


Figure B. Maintenance Panel Logic Board (MPLB)

APPENDIX A

REFERENCE TABLES

This appendix presents 23 reference tables, labeled Reference Table A-1 through A-23.

The following "symbols" are used in Reference Table A-1.

Symbol	Columns	Meaning
____		Underlining shows CB-Constaint.
#	1,2	C-Field value.
.	19	Use "." so REF1 will not be null. See Topic 4.2.
a b	9,13,14	Items "a" and "b" are substituted as needed in column 9 expressions. A value for "a" may not be used for "b", and vice versa; items may be used in pairs only as shown. See notes i & n, and Topics 5.1:5.3.
e	15:18	C-Field CG has default value 0 if none of the forms shown appears. Then local memory is undisturbed. See Topics 4.2 and 4.3.
fg	18:20	REF1 may include a Local Memory destination from column 18 (CG), or a destination from column 19 or 20 (CF), or both, separated by a ":". See Topic 4.1; also see column 5, for REF1 (CB).
i	8,9,14	Any column 8 form may be used for "i" of column 14 in arithmetic/logic expressions of column 9. See Topic 8.3.
js	2	Subscript "(js)" denotes a "jump-spec" or branch specification of form (vector) or (label,label) or (=S) or (label) or (,label). See Topics 7.1 and 8.1.
n	5,9,14	"n" is a decimal (not hex) number in the range 0:15, used as the b-input to arithmetic/logical expressions of column 9. A symbol defined by REGNAM may be used, but rarely makes sense; see note rn. When the subroutine return statement is used, numeric literal n is extended to 8 bits, using field CC, and hence has range 0:255. See Topic 7.5.
p	10,15,18	Column 15 shows the subscript or suffix that must be used with LMx() or SB() of column 10 to form a full OPF1 field specifying Local Memory or a

Symbol	Columns	Meaning
p (cont)		Single-Bit Generator as the data source. Column 18 shows the full REF1 forms for Local Memory as a destination; note that "LM" is used without a third letter. See Topics 4.2 and 4.3.
rn	6,15,16, 18	Parameter "rn" is either a decimal numeric constant like "n", or a register name (symbol) that has been set to a value in the range 0:15 by a REGNAM pseudo-instruction (see Topic 8.6). In either case, "rn" affects CG whether in OPF2 or REF1.
s	2	Field CA, values 0:3, is used to designate explicitly the + or - word of a Control Memory pair, possibly with the subroutine feature (saving L in S). Assignment of these CA-values is indirect (by CAP32) after its memory allocation analysis.
t	17	Additional syntax for subfield REF3 is C-Field direct encoding, of form "Cx=n", where "x" is a C-Field designation (A:G,I,J), and "n" is a decimal number (0:15), giving the desired C-Field value. See Topic 3.2.
vw	11,12	Pairs "v" and pairs "w" are fill specifications for the end bits emptied in the one-bit shifts of column 9. The forms shown are for two staging registers (Y:Z) shifted simultaneously. If only one register is shifted, the form is shortened; if the full form is "vw1,vw2", then for x or y shifted the form becomes "vw1", and for Z shifted the form becomes ",vw2". Note the comma in the Z-shift fill-code. See Topic 5.6.

REFERENCE TABLE A-1.
Syntax for CAP 32 OPF and REF Fields Related to C-Field Values

CAP-32 field		REF2			OPF1 ! OPF2 ! REF1		OPF1 and its parameters									OPF2	REF3	REF1 : REF1			
C-field		CA	CA						CJ	CI					CG				CF		#
Conditions			CA6	CA7	See Col 13	CG5 ⋮ CG7	CJO & CIA	See Col 13		CJO	CJ1	CJ2	CJ3 ⋮ CJF		CJO & CI(O:4)	CJ(1:F) ! CI(6:F)			CEO	CE8	
Subst:			q	k	n	rn	f	i		m	v	w	a	b	p						
In col			2	2	13	15,16,18	10	13		9	9	9	9	9	10						
Mnemonics	0	s	GIO	Z	O	sym	Z	IH	m	LM(p)	0,0	0,0	Y	n	e	e	e	e	.	F	0
	1	s	GI1	NZ	1	sym	NZ	I	SL(v)	LMN(p)	0,Y	0,Y	Y	Y		Z4,LG			X	X:F	1
	2	s	GI2	NEG	2	sym	NEG	I2H	SR(w)	LMS(p)	0,XY	0,X	Y	X	Z4			LM(Z4)	Y	Y:F	2
	3	s	GI3	NNEG	3	sym	NNEG	I2	a-ONE	LMC(p)	0,1	0,1	Y	i	Z4,LG			LM(Z4,LG)	Z	Z:F	3
	4	SO(js)		ZORN	4	sym	ZORN	SDX	a-b	SB(p)	Y,0	Y,0	ZU	n			TFG		Y:Z	Y:Z:F	4
	5		SD3	NZORN	5	sym	NZORN	GIW	a-b-C		Y,Y	Y,Y	ZU	Y		rn,LG			Z:SBR	Z:SBR:F	5
	6	q(js)	DBL	CAR	6	sym	CAR	PU	a+b	D	Y,XY	Y,X	ZU	X	rn			LM(rn)	XU	XL:F	6
	7	k(js)	CIO	OVFL	7	sym	OVFL	P	a+b+C	DB	Y,1	Y,1	ZU	i	rn,LG			LM(rn,LG)	ZU	ZU:F	7
	8	BYTE(js)		Z15	8	sym			a	TFG	Z,0	N,0	Z	n			PI		D	DB	8
	9	FETCH(js)			9	sym			a ! b	YB	Z,Y	N,Y	Z	Y			IG		MR:Z	MW:Z	9
	A	WNZ(js)			10	sym			b′	F(f)	Z,XY	N,X	Z	X	G			LM(G)	MR	MW	A
	B	N5(js)		REM	11	sym			a′	SWIT	Z,1	n,1	Z	i	G,IG			LM(G,IG)	P	PX	B
	C	N4(js)		XY15	12	sym			a&b	ZD	1,0	Z,0	ZL	n	U		IT		FXA	FXB	C
	D	N3(js)		Y15	13	sym			b	ZC	1,Y	Z,Y	ZL	Y	T		DT		SX	W	D
	E	N2(js)		Z0	14	sym			a&b	ZB	1,XY	Z,X	ZS	X	U,IT			LM(U,IT)	S	Q	E
	F	N1(js)		YZ00	15	sym			a*b	ZA	1,1	Z,1	ZS	i	T,DT			LM(T,DT)	GI		F
Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

REFERENCE TABLE A-2. DIRECT FBUS DESTINATION CODE SETS USED IN CAP32 SUBFIELD REF1

REF1	Action	CFE
.	null	00
F	Enable AF, Select ALU for TFG. *1	08
X	X:=FBUS.	10
X:F	X:=FBUS, Enable AF, Select ALU for TFG. *1	18
Y	Y:=FBUS.	20
Y:F	Y:=FBUS, Enable AF, Select ALU for TFG. *1	28
Z	Z:=FBUS	30
Z:F	Z:=FBUS, Enable AF, Select ALU for TFG. *1	38
Y:Z	Y:=FBUS, Z:=FBUS.	40
Y:Z:F	Y:=FBUS, Z:=FBUS, Enable AF, Select ALU for TFG. *	48
Z:SBR	Z:=FBUS, S.14:4:=L.10:0, S.0:=(L.11)'.	50
Z:SBR:F	Z:=FBUS, S.14:4:=L.10:0, S.0:=(L.11)', Enable AF, Select ALU for TFG.	58
XU	ALUU:=ALUL,*2 XU:=FBUSU.	60
XL:F	ALUU:=ALUL,*2 XL:=FBUSL, Enable AF, Select ALUL for TFG. *1	68
ZU	ALUU:=ALUL,*2 ZU:=FBUSU.	70
ZU:F	ALUU:=ALUL,*2 ZU:=FBUSU, Enable AF, Select ALUL for TFG. *1	78
D	D:=FBUS, Set Monobus Write Flags.	80

DB	ALUU:=ALUL,*2 next
	IF M.0=0 THEN DU:=FBUSU, Set Monobus Upper-Write Flag;
	ELSE DL:=FBUSL, Set Monobus Lower-Write Flag.
		88

MR:Z	Z:=FBUS, Set M.17:16,*3 M.15:0:=FBUS, Start Monobus Read-release Acc.	90
MW:Z	Z:=FBUS, Set M.17:16,*3 M.15:0:=FBUS, Start Monobus Write-release Acc.	98
MR	Set M.17:16,*3 M.15:0:=FBUS, Start Monobus Read-release Access	A0
MW	Set M.17:16,*3 M.15:0:=FBUS, Start Monobus Write-release Access	A8
P	P.15:0:=FBUS, Start Monobus Read into I-Register	B0
PX	P.17:16:=FBUS.15:14	B8
FXA	FX:=FBUS.15:14	C0
FXB	FX:=FBUS.1:0	C8
SX:=	SX:=FBUS.15:14	D0
W	W:=FBUS.3:0	D8
S	S:=FBUS	E0
Q	Q:=FBUS	E8
GI	GI:FBUS.3:0	F0

NOTES: *1 For TFG-register to be enabled, "TFG" must be used in subfield REF3.
*2 ALUU:=ALUL has no effect unless ALU is source of FBUS signal.
*3 Set M.17:16 is:

$$\text{IF OPF1} = \left\{\begin{array}{l}\text{LM}\\ \text{LMN}\\ \text{LMS}\\ \text{LMC}\end{array}\right\}\left\{\begin{array}{l}\text{(rn)}\\ \text{(rn,LG)}\end{array}\right\} \quad \text{AND} \quad \left\{\begin{array}{l}\text{rn:=10:11 THEN FX,M.17:16:=P.17:16}\\ \text{rn:=12:15 THEN FX,M.17:16:=SX}\end{array}\right\}$$

ELSE M.17:16:=FX.

REFERENCE TABLE A-3

LOCAL MEMORY FBUS DESTINATION CODES USED IN CAP32 SUBFIELD REF1

REF1 code	Action (Note CLM) (Note SD)	CG
LM(G)	CLM(G) := FBUS	A
LM(G,IG)	CLM(G) := FBUS; next G := (G+1) mod 16	B
LM(Z4)	CLM(Z.3:0) := FBUS	2
LM(Z4,LG)	CLM(Z.3:0) := FBUS; next G := Z.3:0	3
LM(rn)	(where rn = register-name defined by "rn REGNAM n:	
(Note	or rn = n, a decimal number in range 0:15)	
CB)	CB := rn; next CLM(CB) := FBUS	6
LM(rn,LG)	CB := rn; next CLM(CB) := FBUS, G := CB.	7
LM(U,IT)	T :=(T+1) mod 4; SR(1) N; next CLM(T) := FBUS	C
	(Used exclusively in stack handling, see Table A-14)	
LM(T,DT)	CLM(T) := FBUS; next SL(1) N; T := (T-1) mod 4	D
	(Used exclusively in stack handling, see Table A-14)	

NOTES: CB: Field CB is used directly as the value of rn, hence CB-constraint applies to use of index "rn" or "rn,LG".

CLM: "CLM" denotes currently active local memory array, as set in some prior microcommand which used a local memory source code.

SD: Local memory may be used in a single microcommand either as a source or as a destination, but not both. Fields CI and CJ reflect which it is; see Table A-1. See Table A-4 for Local Memory Source Codes.

REFERENCE TABLE A-4

LOCAL ARRAY FBUS SOURCE CODES USED IN CAP32 SUBFIELD OPF1

Syntax: OPF1 ::= array-code(index) ! array-code(index,side-effect)

Part 1. Array Selection:

Array -code	Semantics / Action	CI
LM	Establish primary local memory as currently active; select from it; perform side-effect if specified./ CLM:=PLM, next FBUS:=CLM(index), [next side-effect-action]	0
LMN	Select from local memory array not currently active; perform side-effect if specified / FBUS:=NLM(index), [next side-effect-action]	1
LMS	Establish secondary local memory array as currently active; select from it; perform side-effect if specified / CLM:=SLM, next FBUS:=CLM(index), [next side-effect-action]	2
LMC	Select from the currently active local memory array; perform side-effect if specified / FBUS:=CLM(index), [next side-effect-action]	3
SB	Select the single-bit generator SB(index); perform side-effect if specified / FBUS:=2**index [next side-effect-action]	4

Part 2. Index Control:

OPF1	Action		CG
array-code(G)	FBUS:array(G)		A
array-code(G,IG)	FBUS:=array(G), next G:=G+1 mod 16		B
array-code(Z4)	FBUS:=array(Z.3:0)		2
array-code(Z4,LG)	FBUS:=array(Z.3:0), next G:=Z.3:0)		3
array-code(rn)	FBUS:=array(CB)	(See note rn)	6
array-code(rn,LG)	FBUS:=array(CB), next G:=CB	(See note rn)	7
array-code(U)	FBUS:=array(T+1 mod 4)	(See Table A-14)	C
array-code(U,IT)	T:=T+1 mod 4, next FBUS:=array(T)	(See Table A-14)	E
array-code(T)	FBUS:=array(T)	(See Table A-14)	D
array-code(T,DT)	FBUS:=array(T), next T:=T-1 mod 4	(See Table A-14)	F

NOTE rn: "rn" denotes a register name defined by a REGNAM pseudo-op, or a decimal number in the range 0:15. The value of rn is stored directly in the microcommand, in field CB, hence use of "rn" involves CB-constraint.

REFERENCE TABLE A-5

CONTROL AND USE OF THE G-COUNTER

CAP-32 Subfield	Subfield Code	Action	Comment
OPF2	Z4,LG	G:=Z.3:0	Local memory not used here
OPF1	array-code(Z4,LG)	FBUS:=array(Z.3:0), next G:=Z.3:0	Local array as source. Note AS.
REF1	LM(Z4,LG)	CLM(Z.3:0):=FBUS, next G:=Z.3:0	Local memory as destination
OPF2	rn,LG	G:=CB (Note CB)	Local memory not used here
OPF1	array-code(rn,LG)	FBUS:=array(CB), next G:=CB	Local array source. Note AS Note CB.
REF1	LM(rn,LG)	CLM(CB):=FBUS, next G:=CB	Local memory destination Note CB.
REF3	IG	G:=G+1 mod 16	Local memory not used
OPF1	array-code(G,IG)	FBUS:array(G) next G:=G+1 mod 16	Local array source, Note AS.
REF1	LM(G,IG)	CLM(G):=FBUS, next G:=G+1 mod 16	Local memory destination

NOTES: AS. Array source denotes Local Memory or Single-bit Generator.

CB. rn denotes symbol defined by REGNAM pseudo-op, or decimal number, range 0:15. Value is stored in C-field CB, hence CB-constraint results.

REFERENCE TABLE A-6

DIRECT FBUS SOURCES OTHER THAN LOCAL ARRAYS

OPF1 Code	Action	CJI	CB
D	FBUS:=MDW / Monobus Data Word in to FBUS	06	-
DB	FBUS.15:8:=0; IF M.0=0 THEN FBUS.7:0:=MDW.15:8 ELSE FBUS.7:0:=MDW.7:0. / Monobus Data Byte selected by lowest-order bit of Monobus address register M input to FBUS	07	-
TFG	FBUS.3:0:=TFG; FBUS.15:4:=0. / Load FBUS from Target Conditions Register	08	-
SWIT	FBUS.3:0:=test-switch-settings, FBUS.15:4:=0.	OB	-
ZD	FBUS.3:0:=Z.15:12, FBUS.15:4:=0 / Load 1st Hex from Z.	OC	-
ZC	FBUS.3:0:=Z.11:8, FBUS.15:4:=0. / Load 2nd Hex from Z.	OD	-
ZB	FBUS.3:0:=Z.7:4, FBUS.15:4:=0. / Load 3rd Hex from Z.	OE	-
ZA	FBUS.3:0:=Z.3:0, FBUS.15:4:=0. / Load 4th Hex from Z.	OF	-
YB	FBUS.3:0:=Y.7:4, FBUS.15:4:=0. / Load 3rd Hex from Y.	09	-
F(Z)	FBUS.0:=AFZ, FBUS.15:1:=0. / Load the Zero-flag.	OA	0
F(NZ)	FBUS.0:-AFZ', FBUS.15:1:=0. / Load the virtual Non-zero flag.	OA	1
F(NEG)	FBUS.0:=AFN, FBUS.15:1:=0. / Load the Negative-flag.	OA	2
F(NNEG)	FBUS.0:=AFN', FBUS.15:1:=0. / Load virtual Non-negative flag.	OA	3
F(ZORN)	FBUS.0:=(AFZ!AFN), FBUS.15:1:=0. / Load virtual flag indicating either Zero or Negative, i.e., Non-positive.	OA	4
F(NZORN)	FBUS.0:=(AFZ'&AFN'), FBUS.15:1:=0. / Load virtual flag indicating neither Zero nor Negative, i.e., Positive	OA	5
F(CAR)	FBUS.0:=AFC. / Load the Carry-Borrow flag.	OA	6
F(OVFL)	FBUS.0:=AFO. / Load the Overflow flag.	OA	7

REFERENCE TABLE A-7

ALU SOURCES USED IN CAP32 SUBFIELD OPF1

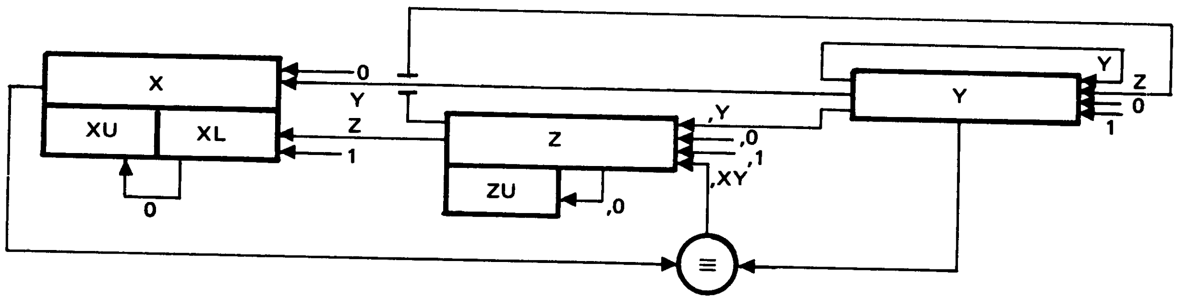
OPF1 COde	Bus	ALU Source Signal (FBUS is same, or complement, or function)	CI	CB
Z	A	ABUS:=Z	8:B	-
ZU	A	ABUS.15:8:=0, ABUS.7:0:=Z.15:8	4:7	-
ZL	A	ABUS.15:8:=0, ABUS.7:0:=Z.7:0	C,D	-
ZS	A	ABUS.15:8:=(8)Z.7, ABUS.7:0:=Z.7:0 (Sign extend Z.7)	E,F	-
Y	A	ABUS:=Y	0:3	-
Y	B	BBUS:=Y (Y appears on BBUS in functional combinations)	1,5,9,D	-
rn	B	BBUS.15:4:=0, BBUS.3:0:=CB (rn is a symbol defined by REGNAM, or a decimal number 0:15, or a decimal number 0:255 in a subroutine return. CB-constraint.)	0,4,8,C	rn
X	B	BBUS:=X	2,6,A,E	-
P	B	BBUS:=P (Program Counter; CB-constraint)	3,7,B,F	7
PU	B	BBUS.15:8:=0, BBUS.7:0:=P.15:8 (CB-constraint)	3,7,B,F	6
I	B	BBUS.15:8:=0, IF P.0=0, THEN BBUS.7:0:=I.15:8, ELSE BBUS.7:0:=I.7:0; next P:=P+1 (CB-constraint)	3,7,B,F	0
IH	B	Like "I", except do not increment P -- "Hold P" (CB...)	3,7,B,F	1
I2	B	Like "I", except shift signal left 1 bit, i.e., times 2. BBUS.15:9:=0, BBUS.0:0=, IF P.0=0 THEN BBUS.8:1:=I.15:8, ELSE BBUS.8:1:=I.7:0; next P:=P+1 (CB-constraint)	3,7,B,F	2
I2H	B	Like "I2", except do not increment P. (CB-constraint)	3,7,B,F	3
SDX	B	BBUS.15:8:=0, BBUS.7:4:=SD, BBUS.3:2:-P.17:16, BBUS.1:0:=FX (CB-constraint)	3,7,B,F	4
GIW	B	BBUS.15:8:=0, BBUS.7:4:=GI, BBUS.3:0:=W (CB-constraint)	3,7,B,F	5

REFERENCE TABLE A-8

ELEMENTARY LEFT SHIFT CODES

Left Shift Syntax: "SL(lsb-fill) register"

Register Codes	Lsb-fill Codes	Lsb-fill Values	Comments
X, XL, Y	0	0	Zero fill
	Y	Y.15	X from Y; Y circular left shift
	Z	Z.15	X or Y from Z
	1	1	One fill
XU	0	X.7	XU fill is independent of lsb-fill code
Z	,0	0	Zero fill
	,Y	Y.15	Z from Y
	,XY	X.15=Y.15	One fill if X.15=Y.15, else zero fill
	,1	1	One fill
ZU	,0	Z.7	ZU fill is independent of lsb-fill code
Y:Z	0,0	0,0	Two simultaneous logical shifts, zero fill
	0,Y	0,Y.15	Z//Y double logical left shift
	0,XY	0,X.15=Y.15	Z//Y double, with Y inversion controlled by X.15
	0,1	0,1	Two simultaneous shifts, 0 fill Y, 1 fill Z
Y:Z	Y,0	Y.15,0	Two simultaneous shifts, Y circular, Z logical
	Y,Y	Y.15,Y.15	Two simultaneous shifts, Y circular, Z from Y
	Y,XY	Y.15,X.15=Y.15	Two simultaneous shifts, Y circular, Z from XY test
	Y,1	Y.15,1	Two simultaneous shifts, Y circular, Z with 1fill
Y:Z	Z,0	Z.15,0	Y//Z double logical left shift
	Z,Y	Z.15,Y.15	Y//Z double circular left shift
	Z,XY	Z.15,X.15=Y.15	Y//Z double circular, with Y inversion controlled by X.15 as bits shift into Z.0
	Z,1	Z.15,1	Y//Z double, with one fill of Z
Y:Z	1,0	1,0	Two simultaneous shifts, 1 fill Y, 0 fill Z
	1,Y	1,Y.15	Z//Y double, with one fill of Y
	1,XY	1,X.15=Y.15	Z//Y double, with Y inversion controlled by X.15 and one fill of Y
	1,1	1,1	Two simultaneous shifts, both one fill

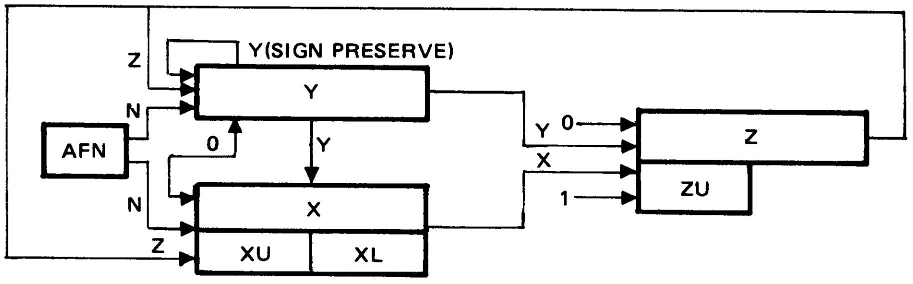


REFERENCE TABLE A-9

ELEMENTARY RIGHT SHIFT CODES

Right Shift Syntax: "SR(msb-fill) register"

Register Codes	Msb-fill Codes	Msb-fill Values	Comments
X	O	0	Zero fill
XU	Y	Y.15	Y sign preservation; X fill from Y sign bit
Y	N	AFN	Fill from Negative Arithmetic Flag
	Z	Z.0	X or Y from Z
XL	O	X.8	XL fill is independent of msb-fill code
	,0	0	Zero fill
Z	,Y	Y.0	Z from Y
ZU	,X	X.0	Z from X
	,1	1	One fill
	0,0	0,0	Two simultaneous logical shifts, zero fill
Y:Z	0,Y	0,Y.0	Y//Z double logical right shift
	0,X	0,X.0	Two simultaneous shifts, 0 fill Y, Z fill from X
	0,1	0,1	Two simultaneous shifts, 0 fill Y, 1 fill Z
	Y,0	Y.15,0	Two simult. shifts, Y sign preserve, Z zero fill
Y:Z	Y,Y	Y.15,Y.0	Y//Z double arithmetic right shift
	Y,X	Y.15,X.0	Two simult. shifts, Y sign preserve, Z fill from X
	Y,1	Y.15,1	Two simult. shifts, Y sign preserve, Z one fill
	N,0	AFN,0	Two simult. shifts, Y true-sign, Z zero fill
Y:Z	N,Y	AFN,Y.0	Y//Z double true-sign right shift
	N,X	AFN,X.0	Two simult. shifts, Y true-sign, Z fill from X
	N,1	AFN,1	Two simult. shifts, Y true-sign, Z one fill
	Z,0	Z.0,0	Z//Y double logical right shift
Y:Z	Z,Y	Z.0,Y.0	Y//Z or Z//Y double circular right shift
	Z,X	Z.0,X.0	Z//Y double right shift with Z fill from X
	Z,1	Z.0,1	Z//Y double right shift with Z one fill

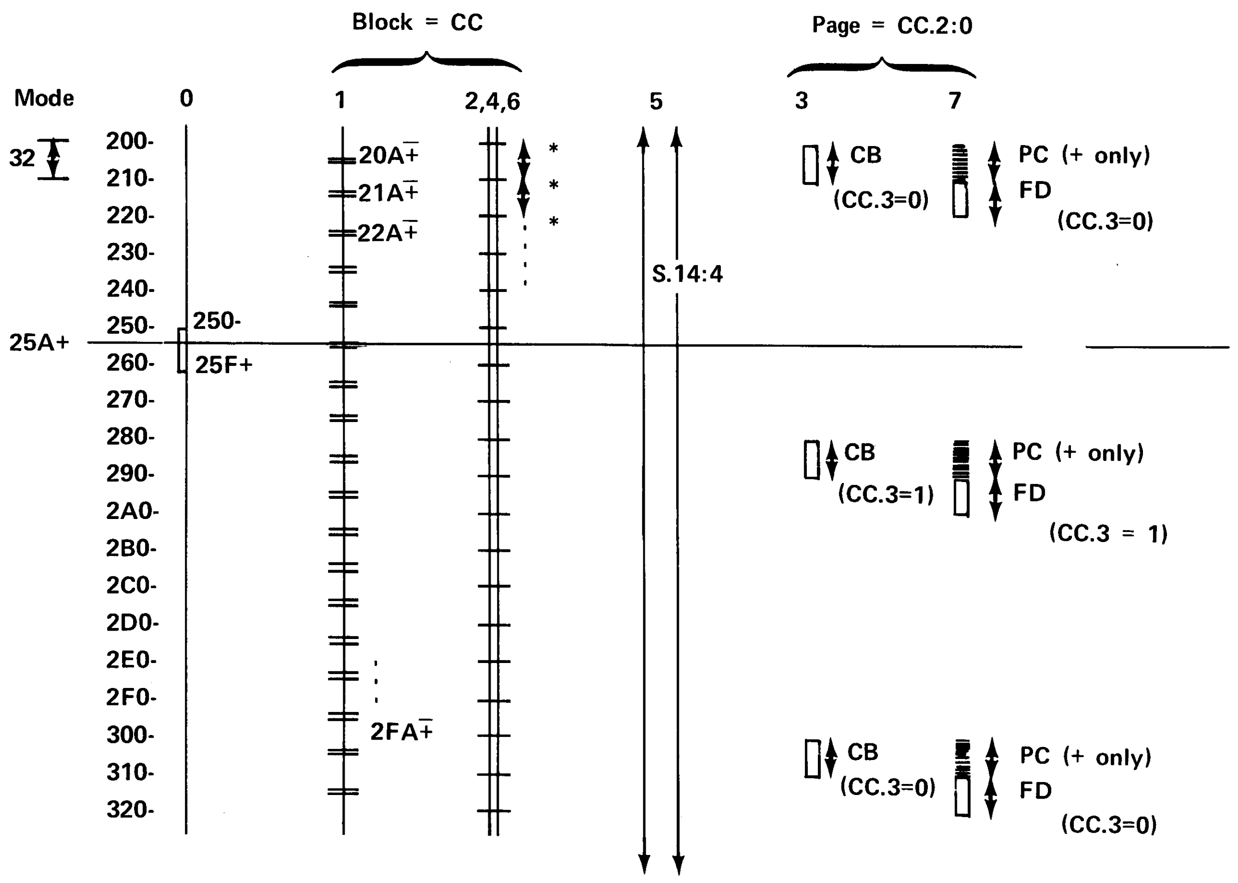


REFERENCE TABLE A-10

CONTROL MEMORY ADDRESSING MODE

Address Mode	Page	Block	Pair	Word	Comment
CD0 (Block)	L.10:8	L.7:4	CC	f(CA)	Any word in current block
CD1 (Mod 16)	L.10:8	CC	1,3:0	f(CA)	Either word in current pair of any block in current page
CD2 (Page)	L.10:8	CC	CB	f(CA)	Any word in current page
CD3 (Newpage)	CC.2:0	CC.3//000	CB	f(CA)	Any word in Block 0 or 8 of any page
CD4 (Data)	L.10:8	CC	S.3:0	f(CA)	Any word in current page
CD5 (Full S-Reg)	S.14:12	S.11:8	S.7:4	f(CA)	Any word in control memory
CD6 (2nd Digit)	L.10:8	CC	SD	f(CA)	Any word in current page
CD7 (1st Digit)	CC.2:0	CC.3//001	FD	f(CA)	Any word in Block 1 or 9 of any page, IF no interrupt!
CD7 (Interrupt)	CC.2:0	CC.3//000	PC	+	If an interrupt is pending, any + word in Block 0 or 8 of any page, as specified by PC, the procedure (interrupt) code

The following diagram shows the range of locations accessible from location 25A+ for the next instruction (for each addressing mode). The asterisk (*) is CB for CD2, S.3:0 for CD4, and SD for CD6.



REFERENCE TABLE A-11

AVAILABLE CONDITIONAL BRANCH TESTS

REF2	Success value / Comment	CA	CB
SO(js)	S.0 / S-register bit.0, the stored sign for a subroutine return; SO normally used with js = "=S", as SO(=S).	4	-
WNZ(js)	W≠0 / W-counter not zero gives success value 1, causing W:=W-1.	A	-
N1(js)	N.1 / N is a shift register with five bits numbered 1:5. Each	F	-
N2(js)	N.2 of these tests succeeds if the corresponding bit of N	E	-
N3(js)	N.3 is a 1. These tests are used in handling the stack	D	-
N4(js)	N.4 head of the 3200 in Y and LM(0:3).	C	-
N5(js)	N.5	B	
GI0(js)	GI.0 / The General Indicators, GI, are four bits designed to	6	0
GI1(js)	GI.1 to be tested. Each test succeeds if the corresponding bit	6	1
GI2(js)	GI.2 is a 1. CB-constrained.	6	2
GI3(js)	GI.3	6	3
Z(js)	AFZ / These tests of the Arithmetic Flags AF succeed if	7	0
NZ(js)	AFZ' the logical truth value of the expression given	7	1
NEG(js)	AFN is 1. CB-constrained.	7	2
NNEG(js)	AFN' NNEG = positive or zero.	7	3
ZORN(js)	AFZ!AFN ZORN = zero or negative.	7	4
NZORN(js)	(AFZ!AFN)' NZORN = definitely positive, not zero.	7	5
CAR(js)	AFC	7	6
OVFL(js)	AFO	7	7
Y15(js)	Y.15 / These tests of the end bits of selected staging	7	D
Z15(js)	Z.15 registers succeed if the bit selected, or the	7	8
Z0(js)	Z.0 logical truth value of the expression given, is 1.	7	E
XY15(js)	X.15≠Y.15 CB-constrained.	7	C
YZ00(js)	Y.0≠Z.0	7	F
REM(js)	(AFZ' & Z.15) ! (AFZ & X.15) / Remainder test; indeed it is a more general switch test, selecting either Z.15 or X.15 depending on the value of AFZ.	7	B
CIO(js)	Succeeds if Concurrent input-output request is pending; used only with i-o controllers having the concurrent i-o feature.	6	7
FETCH(js)	FD.2 / Special tests to speed the decoding of Microdata M32 machine instruc.	9	-
BYTE(js)	FD.1 & FD.0 & SD.3	8	-
DBL(js)	FD.1' & FD.0' & SD.3'	6	6
SD3(js)	SD.3	6	5
+(js)	Transfers control unconditionally to the + or - word, as specified;	1	
-(js)	used to force the pairing of the two labels in js.	0	

NOTE: "js" in Table A-11 is a "jump specification" of one of the forms:

```
(tlabel,flabel) \
(tlabel)         }    See Topic 7.1 for details.
(,flabel)       /
(=S)
```

REFERENCE TABLE A-12

I-REGISTER BYTE EFFECTS ON THE CPU

Part 1. Effects of the currently selected IBY, which depends on P.0 thus:

IF P.0 = 0 THEN IBY := I.15:8; ELSE IBY := I.7:0.

#	Use of IBY Bits	Code Used	Action

1	BBUS: 00000000xxxxxxxx	"IH" in OPF1 b-code (CB-constraint)	BBUS.15.8 := 0
			BBUS.7.0 := IBY

2	BBUS: 0000000xxxxxxxx0	"I2H" in OPF1 b-code (CB-constraint)	BBUS.15:9 := 0, BBUS.8:1 := IBY, BBUS.0 := 0.
3	BBUS: 00000000xxxxxxxx	"I" in OPF1 b-code (CB-constraint)	Same as for "IH", next P := P+1, advance IBY, next IF P.0 = 0, start word fetch
4	BBUS: 0000000xxxxxxxx0	"I2" in OPF1 b-code (CB-constraint)	Same as for "I2H", next P := P+1, advance IBY, next IF P.0 = 0, start word fetch.

5	-
	ISAVE: xxxxxxxx
	L.3:0: xxxx....
		FDB-type vector name in REF2 as <u>vector</u> or <u>test</u>(vector)	1st, handle pending interrupts;
			2nd, save IBY, ISAVE := IBY;
			3rd, branch on FD (First Digit), L.3:0 := IBY.7:4;
			4th, P =: P+1, advance IBY;
			5th, IF P.0 = 0, start word fetch.

Part 2. Effects of ISAVE (a former IBY) on the central processor.

#	Use if ISAVE BITS	Code Used	Action
6	L.3:0: xxxx	SDB-type vector name in REF2 as <u>vector</u> or <u>test(vector)</u>	Branch on SD (Second Digit), L.30:0 := ISAVE.3:0 (See Topic 10.5)
7	BBUS: 00000000xxxxffss	"SDX" in OPF1 b-code (CB-constraint)	BBUS.15:8 := 0, BBUS.7:4 := ISAVE.3:0, BBUS.3:2 := FX, BBUS.1:0 := SX.
8	Test: ..111...	"BYTE(js)" in REF2	Test and branch on ISAVE.5:3 = B'111'.
9	Test: ..000...	"DBL(js)" in REF2 (CB-constraint)	Test and branch on ISAVE.5:3 = B'000'.
10	Test: .1......	"FETCH(js)" in REF2	Test and branch on ISAVE.6 = B'1'.
11.	Test: ....1...	"SD3(js)" in REF2 (CB-constraint)	Test and branch on ISAVE.3 = B'1.

REFERENCE TABLE A-13

INTERRUPT TYPES IN PRIORITY ORDER (FROM HIGHEST TO LOWEST)

Position in INT vector	Q-bit	Name & description
F+	.1	Trace, a firmware interrupt, set by Q.1.
E+		Power Fail.
D+		Restart; recover from Power Fail. Occurs when the +5v. power supply resumes, unless the panel RESET switch is held down.
C+		Load.
B+	.3	Real Time Clock, 120 Hertz.
A+		Concurrent I/O
9+		Panel.
8+		Stop.
7+		Monobus Time Out; expected release did not occur.
6+	.2	Operator interrupt.
5+		Parity error.
4+	.7	I/O Interrupt Line 3.
3+	.6	I/O Interrupt Line 2.
2+	.5	I/O Interrupt Line 1.
1+	.4	I/O Interrupt Line 0.
0+	.0	Wait, a firmware interrupt, set by Q.0.

NOTE: Where no Q-bit is given, the interrupt is not under Q-control. All interrupts may be postponed by use of "PI" in REF3 of the statement preceding the FDB statement.

REFERENCE TABLE A-14

STACK HEAD CONTROL: MANIPULATION OF T-COUNTER AND N-STATS

OPF1	REF	Action / Usage
source	dest,,IT	SR(1) N; T:=(T+1) mod 4. Independently, dest:=source. / Establish dest (usually Y) as TOS when stack head is empty; incrementing of T is usually meaningless in this context.
source	LM(U,IT)	SR(1) N; T:=(T+1) mod 4; next CLM(T):=source. / PUSH source (usually Y) into new LT; may be used to N-mark establishment of TOS in Y (say) by prior statement.
LM(U)	dest	dest:=PLM(U). / Spill LU when it is bottom of a full stack head, in preparation for storing it in memory part of stack.
LM(T,DT)	dest	dest:=PLM(T); next SL(0) N; T:=(T-1) mod 4. / POP LT; may be used to mark POP of TOS from Y (say to empty the stack head.
LM(T)	dest	dest:=PLM(T). / Read TOS1.
LM(U,IT)	dest	SR(0) N; T:=(T+1) mod 4; next dest:=PLM(T) / Unload step, or seek step, or marks unloading of TOS from Y (say) in a subsequent step; in latter 2 uses, dest may be ".".
source	dest,,DT	SL(1) N; T:=(T-1) mod 4. Independently, dest:=source. / Load dest (usually Y) as TOS when stack head is empty, as first step of a complete load-stack-head process.
source	LM(T,DT)	CLM(T):=source; next SL(1) N; T:=T-1) mod 4. / Load step, the second thru fifth steps of the complete load-stack-head process.

REFERENCE TABLE A-15

DEVICE REGISTER BLOCK STATUS WORD (DRB(0))

DRB(0) bits 15:0, from left. DRB(0) location at 0 mod 16 in range 3C000:3FFEF.

Bit	Semantics
0	Controller Busy
1	Device Ready
2	
3	Device Writeable
4	Data Service Interrupt Pending (Controller expects data transfer, and should be serviced, then Acknowledged.)
5	Terminate Interrupt Pending (Bit 0 has just gone from 1 to 0.)
6	Ready Change Interrupt Pending (Bit 1 has just changed.)
7	Special Interrupt Pending
8	Error Bit(s) Set (Any one or more of bits 13:9 has gone to 1.)
9	Data Overrun Detected (Data arrived too fast to be handled; data lost.)
10	Device Parity Error Detected
11	Bus Parity Error Detected (Memory Bus)
12	Error 1 Detected (Varies with device.)
13	Error 2 Detected (Varies with device.)
14	Alarm (Exception condition such as "End of Tape")
15	Operation Aborted

NOTE: Additional Status information may be available in DRB(3), for complex devices. In such cases, DRB(0).8 may be wired to reflect presence of error bits in DRB(4).

REFERENCE TABLE A-16

DEVICE REGISTER BLOCK ORDER BYTE (DRB(1).7:0)

Order Bits		
7 6 5 4	3 2 1 0	Semantics
0 0 0 0	0 0 0 0	No Operation
x x x 0	0 0 1 0	Start Device in Programmed I/O Mode
x x x 1	0 0 1 0	Start Device in Concurrent I/O (CIO) Mode
x x 0 x	0 0 1 0	Start Device in Read Mode
x x 1 x	0 0 1 0	Start Device in Write Mode
x 0 x x	0 0 1 0	Start Device in Run on Error Mode
x 1 x x	0 0 1 0	Start Device in Stop on Error Mode
0 x x x	0 0 1 0	Start Device in Forward Mode
1 x x x	0 0 1 0	Start Device in Reverse Mode
0 0 0 0	0 1 0 0	Stop Device
x x x x	0 1 1 0	Device Control
0 0 0 0	1 0 0 0	Set Special Interrupt
1 x x x	1 0 1 0	Acknowledge (and Clear) Special Interrupt
x 1 x x	1 0 1 0	" " " Ready Change Interrupt
x x 1 x	1 0 1 0	" " " Terminate Interrupt
x x x 1	1 0 1 0	" " " Data Service Interrupt
0 0 0 0	1 1 0 0	Start Device in Initial Program Load Mode
x x x x	1 1 1 0	Spare

x indicates that this bit may be defined as shown on other lines of this table, to effect compound device orders, e.g.,

1 0 1 0 0 0 1 0 is Start Device in Reverse, Run on Error, Write, Programmed I/O Mode

REFERENCE TABLE A-17

DEVICE REGISTER BLOCK MODE BYTE (DRB(1).15:8)

Mode Bits	Value	Semantics
9:8	0 0	Keep Interrupt Mode Unchanged
	0 1	Enable Device Interrupts
	1 0	Disable Device Interrupts
	1 1	Undefined
11:10	0 0	Keep Parity Mode Unchanged
	0 1	No Parity Check or Generation; Data at full width
	1 0	Check for or Generate Even Parity; One Data-bus bit used for parity
	1 1	Check for or Generate Odd Parity; One Data-bus bit used for parity
13:12	0 0	Keep Mode Control #1 Unchanged
	0 1	Undefined
	1 0	Clear Mode Control #1 to 0
	1 1	Set Mode Control #1 to 1
15:14	0 0	Keep Mode Control #2 Unchanged
	0 1	Undefined
	1 0	Clear Mode Control #2 to 0
	1 1	Set Mode Control #2 to 1

NOTE: Meaning of Mode Control #1 and #2 varies with device.

REFERENCE TABLE A-18

BASIC PANEL KEY-SWITCH FUNCTIONS

Label	Function
OFF	All power removed from the system, and battery option (if present) disabled. In all other KEY SWITCH positions, the battery option preserves main memory data in the event of power failure.
STOP	All power on, but with a continuous STOP interrupt present at the processor (position 8+ in the interrupt vector; see Table A-13).
LOCK	Machine running, but with LOAD, INT, and HS1:HS4 disabled, and PSW.4 set. (HS1:HS4 are SYST RESET, PCREG ENABL, CLOCK HALT, and INSTR STOP.) With firmware properly responsive to PSW.4, the Firmware Selector switches (FS1:FS12) will also be disabled.
RUN	Machine running, with all controls operative.
STOP	As in the other STOP position.
HOLD	Main memory power on, to preserve main memory data, but all other power off.

REFERENCE TABLE A-19

MAINTENANCE PANEL STATUS INDICATORS

Label	Indication
MEM	Main memory power on, and memory contents preserved.
ON	Processor power all on.
LOCK	Panel locked, i.e., KEY SWITCH in LOCK position; see Table A-18.
CLOCK	System clock running (with 135 ns period).
BUS	Monobus access in progress.
INSTR	User Level instruction fetches via the Monobus in progress.
BREAK	Breakpoint condition true; see Table A-20, items HC5:HC7. The flash indicating a momentary condition of the selected breakpoint condition true is electronically brightened, so it is visible.

REFERENCE TABLE A-20

MAINTENANCE PANEL HARDWARE CONTROL SWITCHES (HC)

HC#	Label	Function
HC1	SYS RESET	If LOCK state not in effect, initialize processor and Monobus.
HC2	PCREG ENABL	If LOCK state not in effect, inhibit all control memories; next CBUS := PCR (Panel C-register to processor C-bus). Normally, actuation of HC11 (STEP) will follow, to execute the panel microcommand. See also HC8:HC9, to establish microcommand.
HC3	CLOCK HALT	If LOCK state not in effect, halt processor clock; the currently executing microinstruction completes normally.
HC4	INSTR STOP	If LOCK state is not in effect, then cause STOP interrupt (position 8+ in the interrupt vector). Just like STOP position of the KEY SWITCH; see Table A-18.
HC5	CMA BREAK	If HC7 = 0 (off) then BREAKPOINT := (ESR.11:0 = CMA.11:0) else BREAKPOINT := (ESR.11.1 = CMA.11.1). (Breakpoint condition occurs if the 12 lower switches of the Enter Switch Register (except for the least significant switch, if HC7 is set) are equal to the Control Memory Address.) See entry HC11, this table, for BREAKPOINT condition effect.
HC6	MBA BREAK	BREAKPOINT := (ESR = MBA) & (Not HC7 OR Monobus-Write). (BREAKPOINT condition occurs if the entire Enter Switch Register (18 bits) is equal to the Monobus Address and either HC7 is off or Monobus Write is in progress. In effect, if HC7 is on, only a Write access to the ESR address will cause BREAKPOINT; if HC7 is off, any access to the ESR address will cause BREAKPOINT.) See entry HC11, this table, for BREAKPOINT condition effect.
HC7	BREAK -+/W	Flag that affects HC5 and HC6 BREAKPOINT tests. See HC5:HC6.

HC8	ADDR ENTER	DAR.17:0 := ESR.17:0; PSW.2 := 1;
		if HS3 = 1 (on) then PCR.31:16 := ESR.15:0
		(Enter all Enter Switch settings into Display Address Register, and sets Enter Address Bit in Panel Status Word; in a separate action, if HS3 is on, the lower 16 bits of ESR are transferred to the upper half of the Panel C-register, thus forming half of a panel microcommand.) For HS3 detail, see Table A-21.

HC9	DATA ENTER	DDR.15:0 := ESR.15:0; PSW.1 := 1;
		if HS3 = 1 (on) then PCR.15:0 := ESR.15:0
		(Enters lower 16 Enter Switch settings into Display Data Register, and sets Enter Data Bit in Panel Status Word; in a separate action, forms the lower half of a panel microcommand; see HC8, above.

HC10	ADV DAR	DAR := DAR + 1; PSW.0 := 1. (Advances Display Address Register by 1, and sets Advance Bit in Panel Status Word.) Note that any actual incrementation of an address depends on the firmware.
HC11	STEP	If HC3 & NOT HC5 & NOT HC6 then execute one microinstruction. (CLOCK HALT on, with both BREAK controls off.) If HC3 & (HC5 or HC6) then execute microinstructions until BREAKPOINT. (CLOCK HALT on, one or both BREAK controls on; see HC5:HC7.) IF NOT HC3 & HC4 & NOT HC5 & NOT HC6, execute one user-level instruction. (CLOCK HALT off, INSTR HALT on, both BREAK controls off.) If NOT HC3 & HC4 & (HC5 or HC6) then execute user-level instruction. until end of instruction during which BREAKPOINT occurs. (CLOCK HALT off, INSTR HALT on, one or both BREAK controls on.) Else execute microcommands, i.e., RUN.

REFERENCE TABLE A-21

MAINTENANCE PANEL HARDWARE DISPLAY SELECTOR SWITCHES (HS)

HS#	Label	Display Pair in DAR/DDR, and Comments
HS1	DAR / DDR	Display Address Register / Display Data Register (Static) HS1 captures statically the dynamic values displayed in DAR and DDR at the instant HS1 is actuated. DAR and DDR will have been displaying whatever the most recently actuated HS2:HS6 or FS1:FS12 has prescribed.
HS2	MBA / MBD	Monobus Address / Monobus Data (Dynamic)
HS3	CMU / CML	Control Memory Upper Halfword / Control Memory Lower Halfword (Dynamic) HS3 also enables PCR loading from ESR, i.e., establishment of the panel microcommand from two successive settings of the Enter Switch Register; see Table A-20, items HC8:HC9.
HS4	NCMA / CML	Next Control Memory Address / Control Memory Lower Halfword (Dynamic) Since CML contains all of the "next statement" bits, this is a natural pairing that allows evaluation of test results, when stepping.
HS5	CMA / F	Control Memory Address (current) / F-bus Data (Dynamic)
HS6	CMA / I	Control Memory Address (current) / I-bus Data (Dynamic)

NOTE: Each HS, when actuated, turns on the lamp of the indicator right above it, and turns off the effect of the previously actuated HS or FS (Firmware Selector switch).

REFERENCE TABLE A-22

MAINTENANCE PANEL ACCESS THROUGH THE MONOBUS

MBA	REF	Action
3FFFC	MR	Read PSW (Panel Status Word). For contents of PSW, see Table A-23.
3FFFD	MW	Write DAR.17:16 (Upper two bits of Display Address Register) from F.17:16.
3FFFE	MW	Write DAR.15:0 (Lower 16 bits of Display Address Register).
3FFFF	MW	Write DDR (Display Data Register).
3FFFF	MR	Read ESR.15:0 (Lower 16 bits of the Enter Switch Register); the upper two bits of ESR are PSW.15:14; see Table A-21.

NOTE: These front panel MONOBUS addresses behave differently than normal Monobus addresses; each of them causes access to a unique word, even though the odd ones, 3FFFD and 3FFFF, would normally cause access to precisely the same word as the next lower even address, 3FFFC and 3FFFE, if the main memory were being accessed.

REFERENCE TABLE A-23

STRUCTURE OF THE PANEL STATUS WORD (PSW)

Bits	Contents
PSW.15:14	ESR.17:16 (Highest order two bits of the Enter Switch Register)
PSW.13	1
PSW.12:8	0000
PSW.7:4	Identification of the FS (Firmware Selector switch) that has most recently been actuated, range 1:12. If PSW.7:4 = 0000, then an HS (Hardware Selector switch) has been actuated most recently.
PSW.3	LOCK Bit, equal to 1 if and only if the Panel KEY SWITCH is in the LOCK position.
PSW.2	Address Enter Bit, set to 1 when HC8 (ADDR ENTER) is actuated, and restored to 0 when PSW is read via the Monobus, or when HC8 is returned to off position.
PSW.1	Data Enter Bit, set to 1 when HC9 (DATA ENTER) is actuated, and restored to 0 when PSW is read via the Monobus, or when HC8 is returned to off position.
PSW.0	Advance Bit, set to 1 when HC10 (ADV DAR) is actuated, and restored to 0 when PSW is read via the Monobus, or HC10 is returned to off position.

NOTE 1: The PSW may be Read via the Monobus through address MBA = 3FFFC.

NOTE 2: PSW.3:0 are the flags that inform the processor how to respond to an Operator Interrupt ("INT"); see Table A-20, HC8:HC10.