AN EMULATION ORIENTED, DYNAMIC MICROPROGRAMMABLE PROCESSOR (VERSION 3)

by

Charles Neuhauser

25 October 1975

Technical Note No. 65

Digital Systems Laboratory

Stanford Electronics Laboratories

Stanford University

Stanford, California 94305

The work described herein was partially supported by the U.S. Energy Research and Development Administration under contract #AT(04-3) 326PA39.

Digital Systems Laboratory

Stanford Electronics Laboratories

Technical Note No. 65

25 October 1975

AN EMULATION ORIENTED, DYNAMIC MICROPROGRAMMABLE PROCESSOR (VERSION 3)

by

Charles Neuhauser

ABSTRACT

This report describes the CPU of the Stanford Emulation Laboratory, known as the EMMY system. The EMMY CPU is a 32 bit microprogrammable processor designed specifically for the task of emulation research. The control store is dynamic, that is, it is writable by the CPU and thus serves for data storage as well as for microinstruction storage.

This report is a reissue of two previous reports, of the same title, issued at Johns Hopkins University as Hopkins Computer Research Reports #28 and #28.1. However, the material in this report differs somewhat from the previous reports in that the previous reports provided a design specification, and this report describes the system as it is now implemented Specifically, this report provides an EMMY system user with the basic information necessary to microprogram the EMMY CPU and to design hardware and software interfaces to the system bus.

PREFACE

This report describes the Stanford Emulation Laboratory. This laboratory is the result of several years of development first begun at the Johns Hopkins University. Because of its scope, the EMMY project includes the contributions of several individuals. Specifically, the author wishes to acknowledge the follow people:

System Design	Affiliation
Joe Davison	(1)
Lee Hoevel	(1+2)
Dr. Robert McClure	(3)
Dr. Michael Flynn	(1+2)

System Implementation	
Bob Domenico	(3)
Mike Fung	(3)
Dan Davies	(2+3)
Stan Levy	(3)

Affiliation:

(1) The Johns Hopkins University	Baltimore, Maryland
(2) Stanford University	Stanford, California
(3) Palyn Associates Inc.	4100 Moorpark Ave. San Jose, California 95117

--- TABLE OF CONTENTS ---

EMMY PROCESSOR -- PRINCIPLES OF OPERATION

EMMY PROCESSOR -- PRINCIPLES OF OPERATION

1. General Introduction

The EMMY processor is a dynamically microprogrammable machine specifically designed for emulation oriented tasks in research, education and production environments. By making use of high speed RAM technology in the processor control store, this system allows for convenient user microprogramming. In fact, the EMMY system is designed to allow the end user to become directly involved with the manipulation of the processor's primitive computational and storage resources.

This report provides the user with the information necessary to design microprograms for the EMMY processor and to design hardware and software interfaces to the bus system. Principles of operation for the various system bus devices currently available in the laboratory will be the subject of a future report.

1.1 Principal Features

One of the principal design objectives of EMMY has been to give the user direct access to the primitive resources such as adders, shifters and storage. This is necessary if the user is to emulate conventional processor structures efficiently. Primitive resources in the processor are directed in their operation by a microprogram stored in a 4096 word control store, whose locations may be written in a time comparable to the read cycle. This dynamic accessing capability of control store allows the user to quickly load and modify control microprograms for the purpose of debugging and experimentation. Second, because the control store may be accessed under the direction of the current microinstruction word, control store may serve as the primary fast storage resource in a target machine emulation. Thus, control store locations may be used, for example, to hold data emulating the registers of a target machine. Finally, the two level storage hierarchy consisting of main memory and control store allows the user to establish an explicit caching situation in which low usage microinstruction and data sequences may be held in main store and subsequently moved into control store on a demand basis.

In choosing the microinstruction set of EMMY the primary objective was to give the user explicit access to primitive resources in a way which reflects the implicit usage of primitive resources in conventional processors. For purposes of discussion the primitive resources of EMMY may be divided into three classes:

1) Functional -- adder, shifter, etc.,
2) Memory, and
3) Procedural -- testing, branching, etc..

To control these resources efficiently the 32 bit microinstruction word has been divided into two halves. In essence, one half controls functional resources and the other half controls memory resources, with both halves having the capability of controlling procedural resources. In conventional terms one may think of the microinstruction as being a hybrid of 'horizontal' and 'vertical' control organization in that half of the microinstruction appears to be 'vertically' microprogramming a subset of the available resources. This hybrid approach allows the user to capture in the emulator the implicit parallelism available in conventionally structured target machines.

Primitive resources in the EMMY have been designed to be minimally structured and easily accessible, in order to allow the user to structure them as required. All internal data paths are 32 bits in width, the same as the microinstruction word width. The principal functional resources available include a fast arithmetic/logical unit and a fast single and double word shifter. Memory resources consist of an eight register file, control store and the processor bus system where main memory resides. For purposes of specifying the sequencing of microinstructions a full range of condition codes are generated and stored by the processor. These codes maybe tested flexibly by the microprogrammer and used to influence the sequencing of microinstructions.

Since it is intended as an emulator host for a wide class of machines, the EMMY processor is capable of handling a variety of resource requirements with respect to word size. This is accomplished by using the fast shifter resource in conjunction with immediate mask data from the current microinstruction word to allow the microprogrammer to manipulate directly bits and fields within data words. Thus, the EMMY processor provides the user with a great deal of freedom when matching the resources in the processor to particular target machine requirements. While the general design philosophy has been to provide generality in the EMMY resources and their access, several microinstruction classes have been specified which give the microprogrammer the capability of build specific high level operations such as multiply and divide efficiently.

1.2 Processor Specifications and Implementation

1.2.1 General Specifications

All data paths in the EMMY are 32 bits in width, which is the same as the microinstruction word width. Within the processor are eight programmer accessible registers of which seven are general purpose and one is reserved for machine state information. Control store consists of 4K words, which may be used for both microinstruction and dynamic data storage. All EMMY arithmetic operations including microinstruction address formation are two's

complement arithmetic.

The processor host bus system has a 32 bit data word capability and is based on a 24 bit addressing scheme, thus allowing direct access to 16M discrete locations. Logically, the host bus system uses an asynchronous intercommunication scheme to allow devices of various speeds to cooperate efficiently. In the basic system configuration the main memory system consists of 64K bytes of storage with a cycle time of approximately 1 usec.

Microinstruction execution times require varying lengths of time to complete based on multiples of the 35 nsec internal machine cycle. In a simple situation in which control memory is referenced only for the microinstruction a complete cycle consumes 385 nsec or eleven internal cycles, of which 6 cycles are used in the microinstruction fetch and the remainder are spent in actual operand processing. If subsequent data accesses to control store are required the cycle will be extended by 180 nsec.

Host bus interactions (with the processor as either active or passive participant) require varying amounts of time depending on the particular bus units involved. When the EMMY CPU initiates the access, however, it is able to resume processing while awaiting the response.

1.2.2 Implementation

The EMMY system is based on several technologies, specifically:

1) Processor Logic -- Emitter coupled logic (MECL 10K)
2) Control Store -- N-channel MOS (AMS 7001)
3) Bus System -- Open Collector TTL

Figure 1-1 shows the system layout. The majority of the CPU is contained on a single 12" x 15" wirewrap board and consists of approximately 300 IC packages. Below the processor board is a card frame which holds the control store and peripheral bus units. Interconnection on the backplane of this card rack serves as the physical system bus. Control store consists of nine cards each containing a 4 bit by 4K slice of the total system (one card is used to store the parity check bit) and a single additional card containing miscellaneous address circuitry. The CPU communicates with the bus and the control store through another card called the 'I-Board'. This card is used to control micromemory sequencing and some aspects of bus communications. Access to the bus is controlled by an 'Arbiter Card' which performs the access control functions described in section 4.

The remaining card slots are available for bus devices, such as, the Datapoint Interface, Maintenance Console, and Main Memory Controller.

1.3 EMMY System Configuration

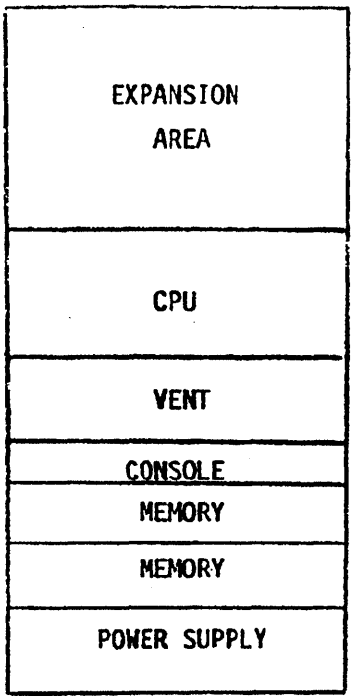
Figure 1-2 illustrates a typical system configuration of the EMMY processor. The particular configuration shown is intended to serve as an emulation research laboratory in which various machine architectures, both 'hard' and 'soft', may be studied and analyzed. Laboratory facilities enable the experimenter to generate emulator microprograms and target machine test programs, load these programs, control their operation during the experiment and gather results for analysis upon termination.

Accessibility and observability of the EMMY and other laboratory resources is the key to success in this environment and is dependent upon efficient inter-unit communications. Primary communications in the laboratory system take place on the host bus system which provides a 32 bit, asynchronously controlled data path between units. In addition to the basic EMMY processor, consisting of the EMMY CPU, control store and an emulation oriented, main memory, the host bus may also include the following:

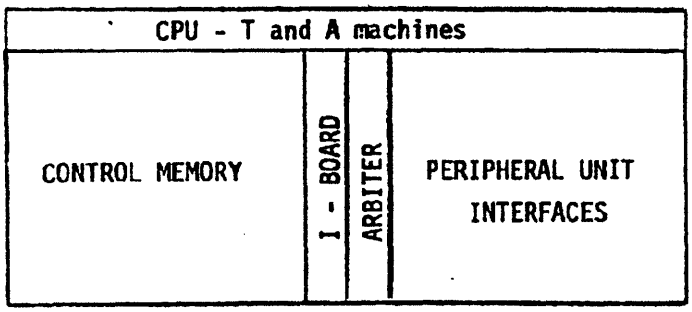
1) Disk controller,
2) Programmer's console,
3) Block access controller,
4) Datapoint 2200 interface, and
5) Auxiliary bus translator.

Host bus structure is such that any two units (with adequate logical capability) may use the bus for communication without the intervention of the EMMY CPU. Furthermore, the EMMY CPU and control store are directly accessible from the bus, thus allowing the experimenter to control the system from an external bus unit such as Datapoint 2200 terminal. The Datapoint 2200 is an 'intelligent' terminal system consisting of a processor, 8K (bytes) of memory, a CRT, keyboard and two cassette tape drives. During laboratory operation the experimenter will use the Datapoint 2200 to initialize the EMMY processor and control its operation. By using the limited, though specialized, processing capabilities of the Datapoint 2200 to handle user/system interaction the EMMY processor system can be devoted to the emulation task.

For special purpose applications auxiliary bus translators may added to match the EMMY host bus electrically and logically to the requirements of a particular manufacturer's peripheral line. Also included on the host bus system is a block access controller designed to move blocks of data between the host bus memory devices in an efficient manner without constant CPU supervision.



EMMY SYSTEM FRAME



CPU and CARD RACK

EMMY SYSTEM - PHYSICAL ARRANGEMENT

FIGURE 1-1

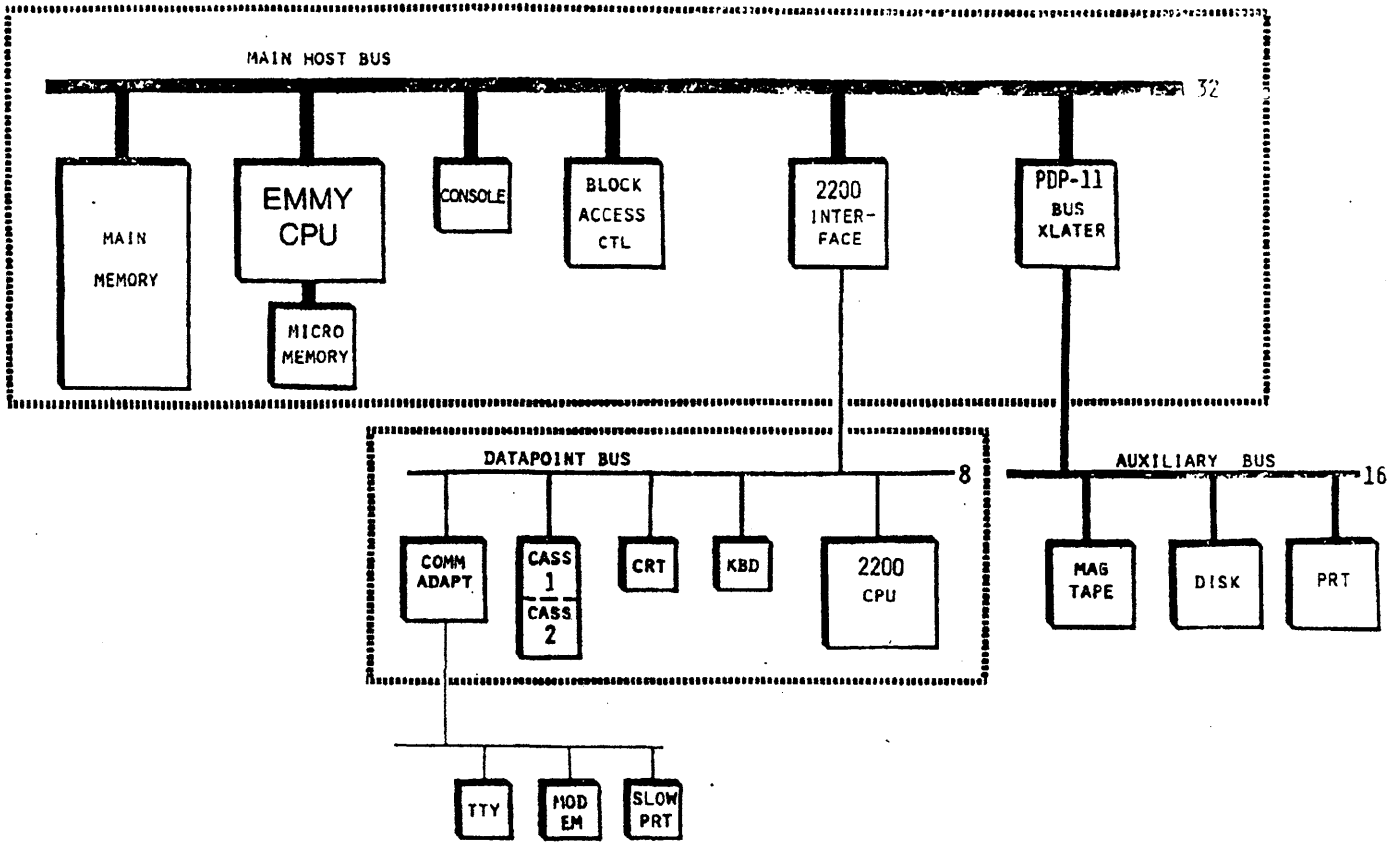


Figure 1.2 Structure of Emulation Laboratory Facility

2. Processor Structural Details

In order to microprogram the EMMY processor effectively the user must be familiar with the general principles of the processor's internal structure. However, unlike many other microprogrammable machines, the user is not required to comprehend minute details of the processor operation. Likewise, due to the 'hybrid' nature of the processor's instruction set, microinstruction specification is as straight forward as in a 'vertical' architecture but retains the resource access characteristic of 'horizontal' organizations. In this section the user will be introduced to the basic structural aspects of EMMY, the general microinstruction formats, and a procedure for estimating the timing of microinstructions. In the following section (3) the syntactic and semantic details of each instruction type are presented.

2.1 Processor Structure

2.1.1 General Principles

Microprogramming is an activity concerned with the direct control of machine resources. Within the EMMY CPU machine resources fall roughly into three catagories:

1) Functional - concerned with data transformation,
2) Memory - concerned with storage access, and
3) Procedural - concerned with the selection (perhaps conditionally) of the next microinstruction.

The organization of EMMY allows the programmer to access and control these resources independently.

Figure 2-1 illustrates the functional structure of the EMMY resources (the actual stucture is discussed later and is shown in figures 2-2 through 2-5). To control the three resource groups (functional, memory and procedural) the microinstruction word is divided into two halves: the left half (bits 31-18) which controls the functional resources and the right half (bits 17-0) which controls the memory resources. Microinstructions are normally selected sequentially from control store. This order may be changed conditionally or unconditionally by instructions from either half of the current microinstruction word.

Associated with the control of the resource groups in EMMY are three finite state sub-machines designated as follows:

1) T-machine (Transformation) - controls functional resources
2) A-machine (Auxiliary) - controls memory resources, and

3) I-machine (Instruction fetch) - controls procedural resources.

These sub-machines each control their associated resources under the direction of the applicable segment of the microinstruction word. Each machine functions independently of the others except when data dependent conflicts occur. Thus, for example, the I-machine is continually attempting to fetch the next microinstruction except when it finds the control memory busy, possibly answering a request by the A-machine.

In addition to the three sub-machines discussed above the EMMY also has a forth sub-machine, the bus controller. The bus controller is not under the direct control of the microinstruction word but rather answers requests presented to it by the A-machine or the system bus. When requested by the A-machine the bus controller will oversee the movement of data between EMMY internal storage (i.e. registers or control store) and the EMMY bus system. Once initialzied the bus controller will carry out the bus operation requested independently thus allowing the A-machine (and consequently the EMMY CPU) to continue processing microinstructions. THe bus controller also handles bus requests for access to EMMY registers and control store by intervening in the normal sequencing of the I-machine.

2.1.2 Specific Structure

Figure 2-2 illustrates the important data paths and units which comprise the EMMY processor. In general, the processing of an EMMY microinstruction proceeds in three steps each under control of one processor sub-machine. The normal sequencing is: I-machine first, T-machine second and A-machine last. Depending on the particular microinstruction one or more of the sub-machine sequences may be omitted. In addition to these sequences, EMMY may also carry out special purpose sequences associated with bus access and interrupt handling.

2.1.2.1 I-machine Sequence (Figure 2-2)

The address of the next microinstruction and other state information is maintained in register 0 of the register file. At the start of the I-sequence this address is fetched from R0 and placed in uMAR, the micromemory address register. The micromemory is cycled and the results of the read operation are deposited in the MIR (microinstruction register) for decoding. Simultaneously, R0 is incremented, using the ALU, so that it points to the next (assumed) microinstruction.

2.1.2.2 T-machine Sequence (Figure 2-3)

A typical T-machine sequence begins with the fetching of one or two operands from the register file. These operands are placed in the auxiliary registers Ra and Rb. Operands are processed as required by the ALU and the result is returned to the register file. If condition codes are generated by the result then the machine state information contained in R0 of the register file is updated. A functional instruction may obtain one operand from the MIR as immediate data via the data path between the MIR and the second operand input to the ALU.

On microinstructions requiring a shift or rotate operation the auxiliary registers Ra and Rb are used together to form a 64 bit shift and rotate unit. Shifting is controlled by the shift counter and ALU.

2.1.2.3 A-machine Sequence (Figure 2-4)

Generally, A-machine sequences move data between two memory resources (e.g. register to micromemory, micromemory to bus memory). In addition, some A-machine sequences may update registers using the ALU to perform simple operations such as addition. Address input to the micromemory is via the micromemory address register (uMAR). During A-machine sequences the uMAR obtains an address from either the register file, the current microinstruction word residing in the MIR or the EMMY bus system. Data input to micromemory resides in the micromemory data register (uMDR). Inputs to the uMDR originate in the register file or on the system bus. Micromemory outputs are directed to either the bus system via the bus data register (BDR) or the register file via the ALU.

A-machine sequences which involve the EMMY bus system begin by moving an address into the BDR and initializing the bus control unit. On bus write operations the BDR is loaded with data (from the register file or micromemory) after the address is accepted by the bus system. On bus read operations data is returned to the CPU and is deposited in the register file or micromemory as required.

Due to the asynchronous nature of the EMMY bus system, the A-machine only initiates the bus transfer action., while the actual transaction is completed later. In the meantime, microinstruction fetching may continue unles another attempt is made to access the bus system. In this case the A-machine will not proceed until the previous transaction is completed.

2.1.2.4 Special Sequences (Figure 2-5)

Bus units external to the EMMY processor may read or write register file and micromemory locations on a shared basis with the CPU. For such operations address information from the bus is directed to either the uMAR or register file access controller (not shown). Data paths and sequences correspond to the CPU initiated sequences described above for register file, micromemory and bus system transfer.

An interrupt sequence begins with a special command from the bus system. No data is transferred, but instead the address received from the bus system is used to address micromemory via the uMAR. Register 0 from the register 0 from the register file is then placed in this micromemory location. New contents for register 0 are retrieved from the even-odd pair associated with the given location and loaded into R0 of the register file. By this process the old machine state is saved and the new machine state set without any intervening CPU processing.

2.2 Microinstruction Set Structure

2.2.1 General Structure

EMMY microinstructions are designed to allow the microprogrammer to access primitive resources in a direct manner. This accomplished by logically dividing the 32 bit microinstruction into two halves; a left half, 14 bits wide, and a right half, 18 bits wide. The three resource groups discussed earlier are designated functional, memory and procedural (F, M, and P respectively). Each half may designate control for one group of resources and additionally the right half may be used as immediate data in functional resource operations. This then gives rise to five basic microinstruction formats which designate control for particular resources:

LEFT HALF	RIGHT HALF
F-control	F-data
F-control	M-control
F-control	P-control
P-control	M-control
P-control	P-control

For each microinstruction (F, M or P) a 'NOP' code exists thus allowing the microprogrammer to use a specific resource independently.

Unless microinstruction sequencing is explicitly modified the normal sequence of microinstruction fetch is sequential by control store location. Because the current microinstruction address is maintained in the register file the microinstruction fetch sequence may be modified by a functional or memory

microinstruction.

2.2.2 Brief Description of the Microinstruction Set

Because the microinstruction word in EMMY is a hybrid of vertical and horizontal format, the microinstruction set consists of several thousand instructions even where register and memory address designations are excluded. To make discussion easier the microinstructions have been divided into classes and sub-classes. The basic classes of microinstrutions are:

1) Functional - data transformation,
2) Memory - data storage and address calculation, and
3) Procedural - microinstruction flow of control.

In the discussion below the general features and characteristics of each class are examined. Specific details are given in Section 3.

2.2.2.1 Functional Instructions

Functional microinstructions are designed primarily to perform operations which transform data (i.e. arithmetic and logical operations). These microinstructions are performed by the T-machine and are specified by the left half field of the microinstruction word. Figure 2-6 shows bit formats of the six currently implemented functional microinstructions. By sub-class the functional microinstructions are:

1) Logical - performs bitwise Boolean operations,
2) Arithmetic - performs two's complement arithmetic and compare operations,
3) Shift/Rotate - performs single and double length shift and rotate operations,
4) Extended - performs fragments of specialized arithmetic operations,
5) Extract - isolates a specific field within a data word, and
6) Insert - inserts a specific field into a data word.

The left three bits of the left half field of the microinstruction specify the microinstruction sub-class as shown in figure 2-6. For the first four sub-classes (Logical, Arithmetic, Shift/Rotate and Extended) the use of the remaining bits is the same. Four fields are identified:

1) I - use or non-use of immediate data,
2) OP - specific operation,
3) BF or BF/VAL - operand register or small immediate value
4) AF - operand source and sink register.

Generally speaking, these microinstructions process two operands and produce a single result which is returned to the register specified by the AF field. The I field determines the source of one operand; either from the register file (as specified by the BF field) or from the right half of the microinstruction word (expanded as described below).

For the Insert and Extract instructions the fields designated have the following meaning:

1) POS - amount of field rotation,
2) AF - operand source and sink register, and
3) BF - operand source.

The insert and extract instructions always use immediate data from the right half field of the current microinstruction. When used for immediate data the 18 bit quantity in the right half field is expanded to form a 32 bit quantity as shown in figure 2-7. The right 16 bits of the field are data and the left two bits (17 and 16) specify whether the 16 bit data quantity given is to be right or left justified and whether the remaining 16 bits are to be zero or one filled.

2.2.2.2 Memory Microinstructions

Memory microinstructions are used by the microprogrammer to move data between the various memory resources of the EMMY system and to perform simple memory related arithmetic operations, such as address calculation. All memory type microinstructions are specified in the 18 bit right half field of the microinstruction word. Execution of these microinstructions is controlled by the A-machine. Five memory microinstructins are specified:

1) Load Register - load register from control store,
2) Store Register - store register to control store,
3) Load Immediate - load register with immediate data,
4) Indirect Access - memory to memory transfer, and
5) Pointer Modification - register address calculation and test.

The first three bits of the 18 bit field are used to designate the particular memory sub-microinstruction desired (figure 2-8). For the first three sub-classes (Load Register, Store Register and Load Immediate) have two fields specified:

1) CF - designates a register, and
2) ADR - designates a control store address or immediate data.

The Load Immediate instruction is used primarily to load control store addresses into registers. An operation useful in microprogram branching.

The remaining two sub-classes (Indirect Acce and Pointer Modification) have five fields specified:

1) CF - source and/or sink operand register,
2) DF - sink operand or small immediate value,
3) EF - sub-opcode field,
4) XOP - sub-opcode field, and
5) VAL - immediate value (-8 to +7).

The pointer modification instruction, in addition to its address calculation capability, is also used to directly control the sequencing of the microinstruction stream. Its primary usefulness is in the microprogramming of short counting loops.

2.2.2.3 Procedural Microinstructions

Procedural microinstruction are used to control the sequencing of the microinstruction stream. A procedural microinstruction may be specified in either half of the miroinstruction word. This class of microinstructions may be considered to control the I machine, in that procedural instructions may modify the current microinstruction address in register 0 either directly or indirectly. Figure 2-9 shows the formats of the three procedural class microinstructions:

1) Conditional,
2) Branching, and
3) Looping (Pointer Modification).

A Conditional microinstruction appears only in the left half field of the microinstruction word. Two fields are given:

1) CMASK - code test mask, and
2) COP - test type specification.

The Conditional microinstruction performs a test on the condition or indicator codes (see section 2.4.2) maintained in register 0. Depending on the outcome of this test the microinstruction specified in the right half field of the microinstruction word is executed or skipped. Since the microinstruction in the right half field may, among other things, modify the current microinstruction address in register 0, conditional branching may be implicitly specified.

In performing the test specified, the Conditional instruction uses the CMASK field as a mask to identify relevant bits in the condition or indicator codes and uses the COP to specify the test type and the logical sense (i.e. true or false) of the result. The nature of the conditional test is explained more fully in section 2.4.2 and the exact definition is given in section 3.

The Branch microinstruction, which appears only in the right half field of the microinstruction word, performs tests on the condition or indicator codes in the same manner as the Conditional microinstruction. An additonal field, the VAL field, is specified. If the test result is logically 'true' then the sign extended value of the VAL field is added to the next microinstruction address pointer, thus causing a short relative branch. The fields BMASK and XOP correspond functionally to the CMASK and COP fields of the Conditional microinstruction.

The 'Looping' procedural microinstruction is another aspect of the Pointer Modification microinstruction described in the preceeding section. In addition to performing simple arithmetic calculations (addition, subtraction) on two registers, the Pointer Modification microinstruction may test the results of the calculation and based on these results perform a short relative branch. The distance of the relative branch is given by the VAL field (sign extended). The Pointer Modification or 'Looping' microinstruction is intended primarily to allow the microprogrammer to specify short counting loops in microcode, such as might be required in multiply or normalize operations.

2.3 Address Structure

Basic memory resources within the EMMY system consist of registers, control store and the bus memory system. Nearly all memory locations are general purpose in nature. Those which have special segnificance will be discussed below.

2.3.1 Registers

Eight registers are provided in the EMMY CPU. One of these, register 0, is dedicated (in hardware) as the machine state register containing information such as the next microinstruction address pointer and the current condition codes. The remaining seven registers are available for general use by the microprogrammer.

2.3.2 Control Store

Control store consists of 4096 locations. All locations are available for general purpose use by the programmer except locations 044 through 04D which are reserved, by convention, for interrupt information according to the following scheme:

Address	Interrupt Source
044-045	Programmer's Console
046-047	Main Memory System
048-049	Datapoint Interface
04A-04B	Block Access COntroller
04C-04D	Bus Time-out.

When an interrupt occurs the EMMY hardware will store the current contents of register 0 into the odd location of the appropriate interrupt address pair. Then the contents of the even location are used to replace the current contents of register 0 and thus initialize a new machine state. No other registers are changed.

2.3.3 Bus Addresses

Bus addresses are specified by a 24 bit quantity which allows the microprogrammer to directly access 16M locations. The following locations have been assigned specific purposes:

Address	Purpose
FF0000-FF0FFF	Control Store Access (000 to FFF)
FF1000-FF1007	CPU Register Access (0 to 7)
FE0000-FE0003	Programmer's Console (see Appendix)
FD0000	Datapoint Interface (see Appendix)
000000-03FFFF	Main Memory. (see Appendix)

2.4 Machine State Word (Register 0)

Register 0 of the register file contains information about the current state of the EMMY processor. Bit formats of this register are given in figure 2-10. The contents of register 0 may be divided logically into four groups:

1) Microaddress Register (MAR),
2) State,
3) Indicator Codes, and
4) Condition Codes.

The MAR (bits 11-0) contains the pointer to the next microinstruction. By manipulating this pointer, either directly with functional or memory microinstructions or indirectly with procedural microinstructions, the microprogrammer may change the normal sequential fetching of microinstructions.

In the four bit State field only two bits are currently used. One bit (15) designates whether the EMMY processor is halted or running, and the other bit (14) specifies whether interrupts are enabled or disabled.

The high 8 bits (31-24) of register 0 contain the processor set condition codes and the following 8 bits (23-16) contain the programmer set indicator codes. The contents of either code group may be tested using the Conditional or Branch microistruction described in the previous section.

Indicator codes are intended for use by the microprogrammer to maintain temporary information which is used directly in conditional tests. Indicator codes are not disturbed by the processor when it updates other register 0 information, such as

the MAR or the condition codes. The indicator codes usually find application in holding state information about the emulated target machine, such as whether the current emulated instruction starts on a full or half word boundary.

2.4.1 Condition Code Semantics

Eight condition code bits are specified. Condition codes are set according to the results of Logical, Arithmetic and some Extended class microinstructions. Bit semantics, as shown in figure 2-11, are relatively independent thus facilitating complex conditional testing using the Conditional or Branch microinstruction.

The first two bits of the condition codes give direct data relating to arithmetic results. The overflow combination is set if the carry into the sign bit (bit 31) differs from the carry out. Bits 29,28 and 27 of the condition code correspond to the generated carry and the high and low bits of the result. Bit 26 designates whether all bit positions of the result are the same or not, and bit 25 indicates whether the result had even parity (bit 25 = 1) or not (bit 25 = 0). Bit 24 (BUSY) indicates the status of the last bus operation issued by the CPU. If it is uncompleted bit 24 is '1', otherwise bit 24 is '0'.

2.4.2 Condition Code Testing

Testing of the condition and programmer codes is by means of the Conditional or Branch microinstructions. Test information consists of an eight bit mask and a three bit test type. The mask indicates the subset of the condition codes to be tested and, the test type specification indicates how the test is to be carried out. The three bits in the test type are:

1) V - sense (normal or inverted),
2) C - complement codes before masking, and
3) S - code to be tested (condition or indicator).

Generation of the test result proceeds as follows. Depending upon the 'S' bit either the condition or indicator codes are selected for testing. The selected codes are then complemented or not according to the 'C' bit. Results are then product masked (i.e. ANDed) with the eight bit mask given in the test instruction and all bits of the result are ORed together. The resulting bit gives the sense of the test (i.e. valid or invalid) and may be further complemented by the 'V' bit to get the desired sense of the test. This procedure, though complex, allows the microprogrammer a great deal of flexibility in defining conditional statements.

In effect, the programmer isolates a group of bits from the appropriate code field (condition or indicator) using the mask and

tests these bits as follows:

V	C	Test
0	0	Any bit is set
0	1	Any bit is not set
1	0	All bits are not set
1	1	All bits are set

2.5 Determination of Microinstruction Execution Time

Determination of microinstruction execution times in EMMY is complicated by several factors, some of which, in a practical sense, are not under the direct control of the microprogrammer. Because of various indeterminate and uncontrollable factors (such as bus contention) exact timing for a given sequence of microcode may be impossible. However, microinstruction timing may be made with sufficient accuracy to allow the microprogrammer to choose between alternative sequences which perform the same function.

To estimate execution time the microprogrammer must consider the following:

1) Basic microinstruction execution time
2) Possible degradation due to contention for control store
3) Effects of bus accessing

2.5.1 Basic Microinstruction Execution Time

Figure 2-11 illustrates the components of a complete EMMY cycle. The basic cycle (in execution order) consists of a microinstruction fetch (IFETCH), execution of the T-machine operation and execution of the A-machine operation. CPU accesses, by other bus units, may occur prior to the IFETCH or between the T- and A-machine execution phases.

Ignoring external accesses to the CPU, the execution time for a given microinstruction may be determined by adding the time required for the IFETCH, T-operation and A-operation. The times given in figure 2-11 are in terms of the number of minor cycles each stage of execution consumes. In the current wire wrapped implementation a minor cycle is 35 nsec in length.

Some microinstructions, such as Indirect Access, take variable amounts of time depending upon the options specified. Others, such as Pointer Modifiy and Multiply Step take varying lengths of time in a data dependent manner. Further, the A-machine execution stage may be skipped entirely for one of the following reasons:

1) Conditional test fails
2) The ACF field is used as immediate data
3) Bit 28 is set on Extended Arithmetic microinstructions

2.5.2 Control Store Contention

Conceptually, the control store cycle in the current EMMY implementation consists of two phases; an access phase and a recovery phase. Microinstruction processing will continue immediately following the access phase. However, a subsequent control store access may encounter delay if it begins before the recovery phase is complete. There are two sources of control store contention of concern to the microprogrammer: first, contention between the IFETCH and a following control store access and, second, between the bus and the A-machine.

Nine minor cycles must elapse between the start of an IFETCH and the next control store access. Usually there is no conflict, since the IFETCH consumes six minor cycles and most T-machine instructions consume three or more. Currently, only the Extended Arithmetic 'transfer' operation consumes less than three cycles. If this T-instruction is followed by a control store access (either from the bus or the A-machine) then a delay of one minor cycle will result.

If the A-machine instruction requires an immediate access to control store and a bus operation occurs between the T- and A-execution cycles, a delay of two minor cycles will occur. This happens on all bus accesses even those which involve only the register file.

2.5.3 Bus Access Timing

CPU delays due to bus accessing are, in general, to determine exactly. Roughly speaking, there are four sources of delay:

1) Completion of deferred operation
2) Initialization of access
3) Response time of the accessed device
4) Asynchronous slave access

Operations in which the CPU reads slave bus devices (R <- X or M <- X) are termed deferred operations, since the access is only initiated by the CPU. Later, the data requested will be returned at which time the CPU must 'give up' cycles to the slave device. Completion of 'deferred' bus operations requires six or eight minor cycles depending upon where in the microinstruction cycle the completion occurs and the delay due to contention for control store.

Whenever the CPU initiates a bus access it will be delayed until its request for bus access has been answered (see section 4). This delay depends upon bus traffic and the contention for specific bus devices. Once the bus access request has been accepted, CPU processing may proceed. If another bus access is attempted before completion of a pending request the CPU will delay until the first request is completed.

Bus devices have widely varying response times which may cause delays in microinstruction execution. After bus access is obtained there may be a significant delay before the addressed slave device recognizes its address and responds. This delay is device dependent.

The final consideration in CPU timing is the effect of asynchronous slave accesses to the CPU. At a minimum, a read of the CPU requires six minor cycles, and a write requires fourteen cycles. Much longer delays may occur if the accessing slave device is slow in sending data or responding to a read. Interrupts require a minimum of sixteen cycles.

To summarize, the determination of microinstruction timing should proceed as follows:

1) Determine the sum of times consummed in the IFETCH, T- and A- minroinstruction execution phases,
2) Add delays associated with control store contention,
3) Account for delays due to deferred operations,
4) Consider possible effects due to contention for the system bus, and
5) Allow for delays due to asynchronous slave accesses.

2.6 Exceptions

Two error conditions are detected by the EMMY processor during normal operation:

1) Control store parity fault, and
2) Bus time-out.

Each control store word is protected by a single parity bit. Parity is generated on write operations and is checked on read operations. If a parity failure is detected during a read operation, the EMMY CPU will halt and indicate a '1' on the bus PARITY ERROR line. The parity error condition may be reset using the PARITY RESET bus line, however, the EMMY CPU will not resume processing until the RUN line is pulsed.

Bus time-out occurs when a single device holds the system bus for longer than 75 usec. A bus time-out will cause an interrupt to the control store address pair 04C-04D and indicate a '1' on the TIMEOUT bus line. The time-out condition may be cleared by

issuing a MASTER CLEAR signal. In general, bus time-outs occur when an attempt is made to address a non-existant device or use a device improperly in a data transfer operation.

Both the PARITY RESET and MASTER CLEAR lines are available on the system bus (see section 4.) but are not accessible directly by the CPU. Thus, from the viewpoint of the microprogrammer, the CPU is unable to clear the parity or time-out conditions independently.

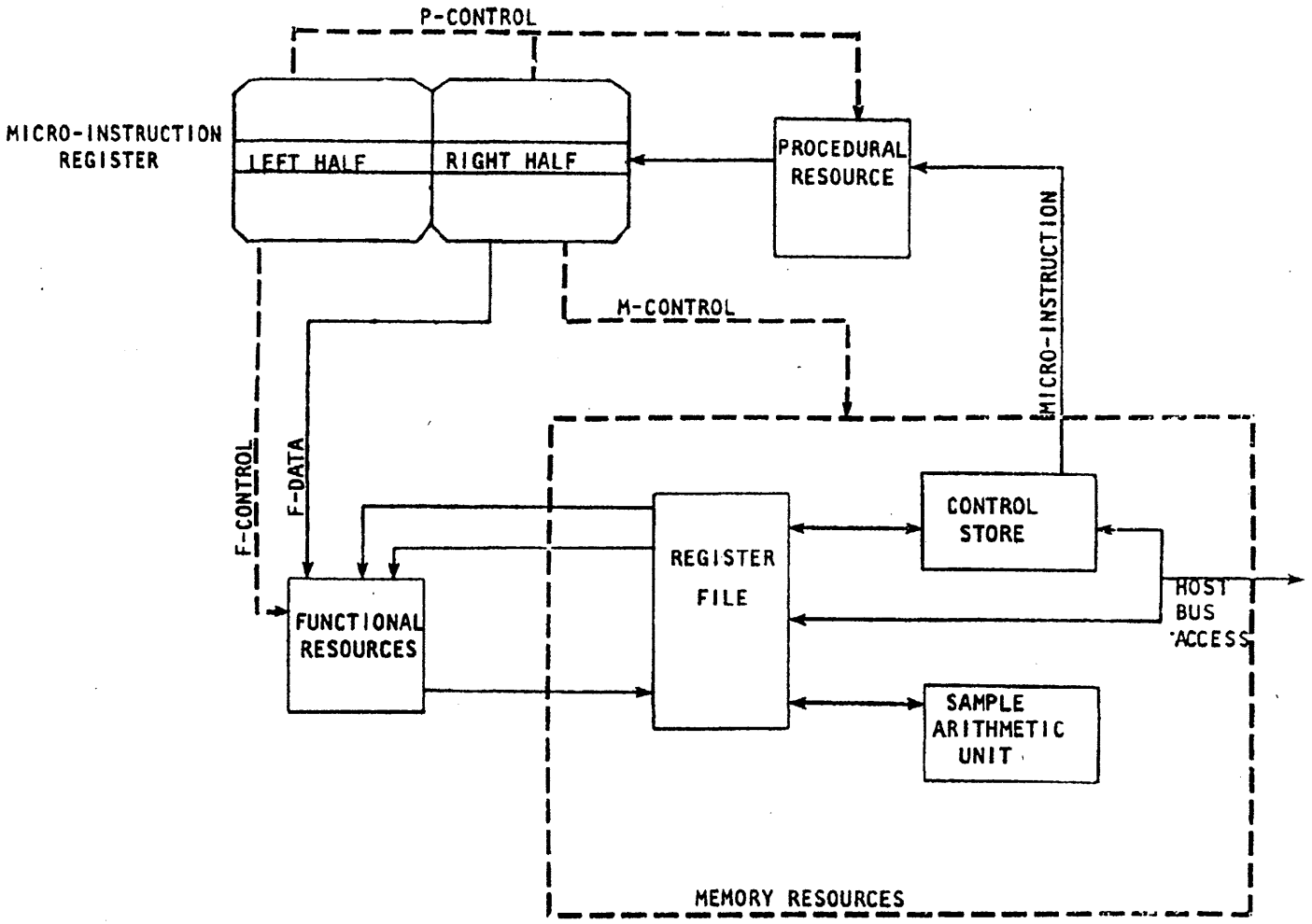


Figure 2.1 Functional Structure of Emmy

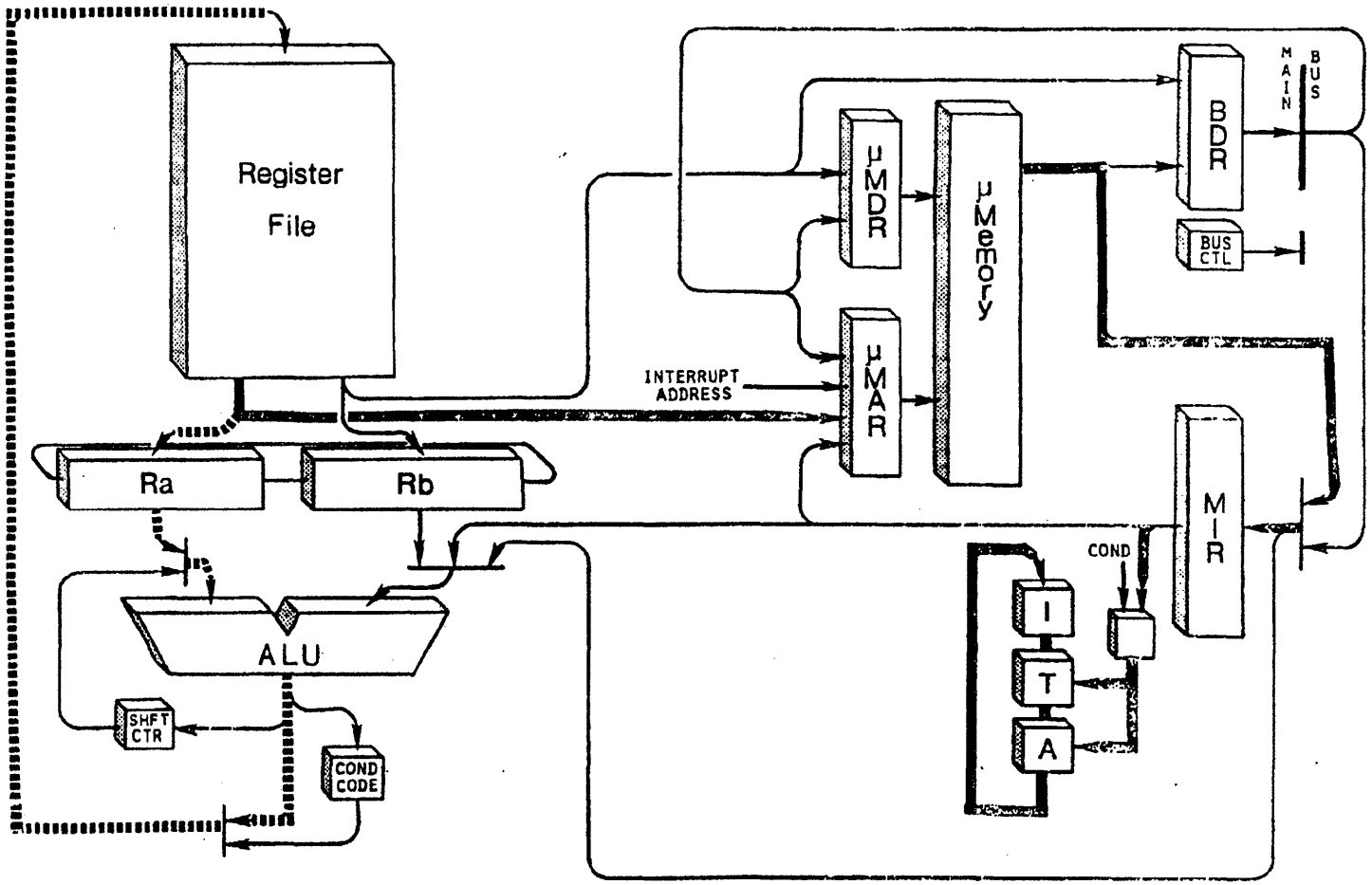


Figure 2.2 I-Machine Sequence

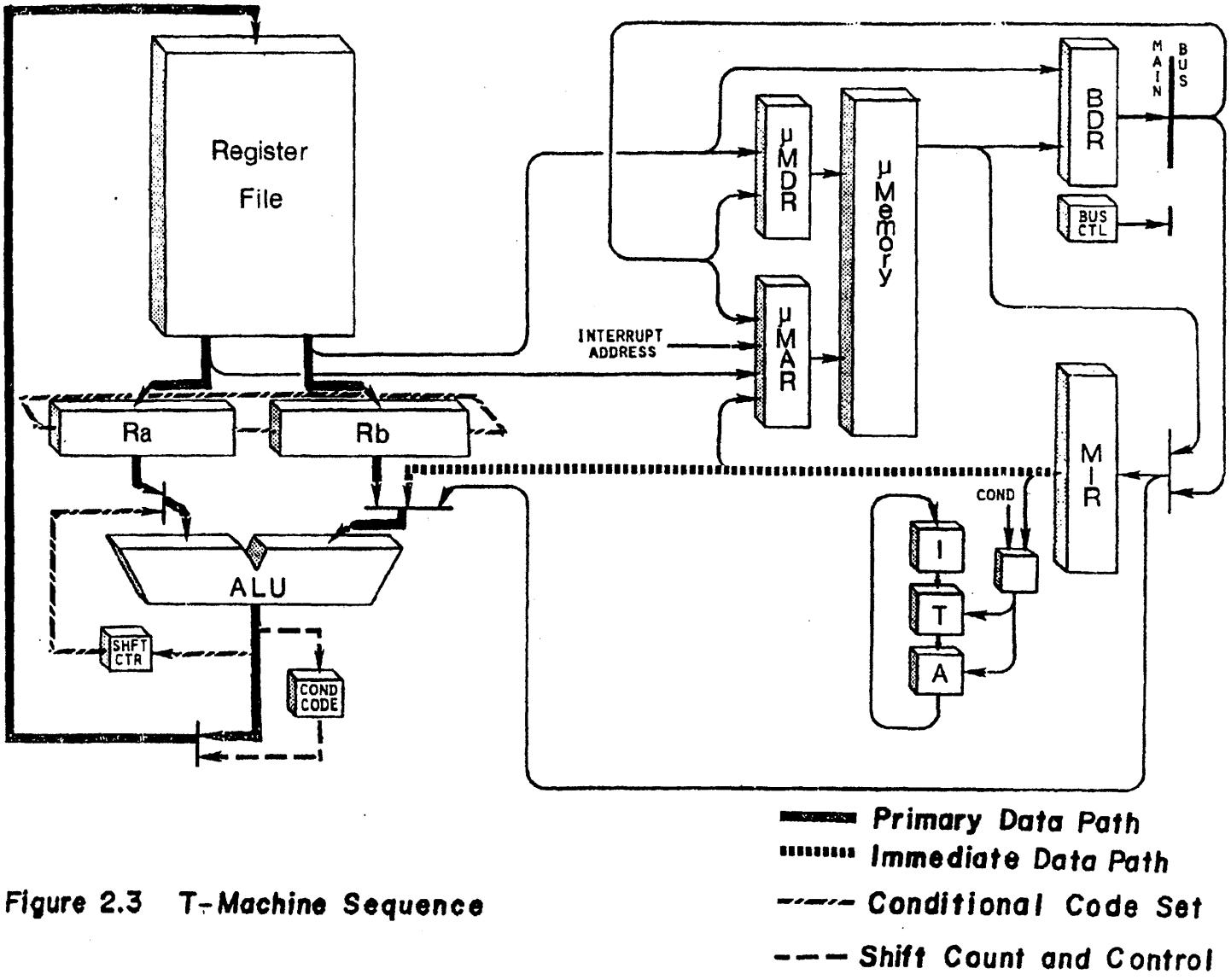


Figure 2.3 T-Machine Sequence

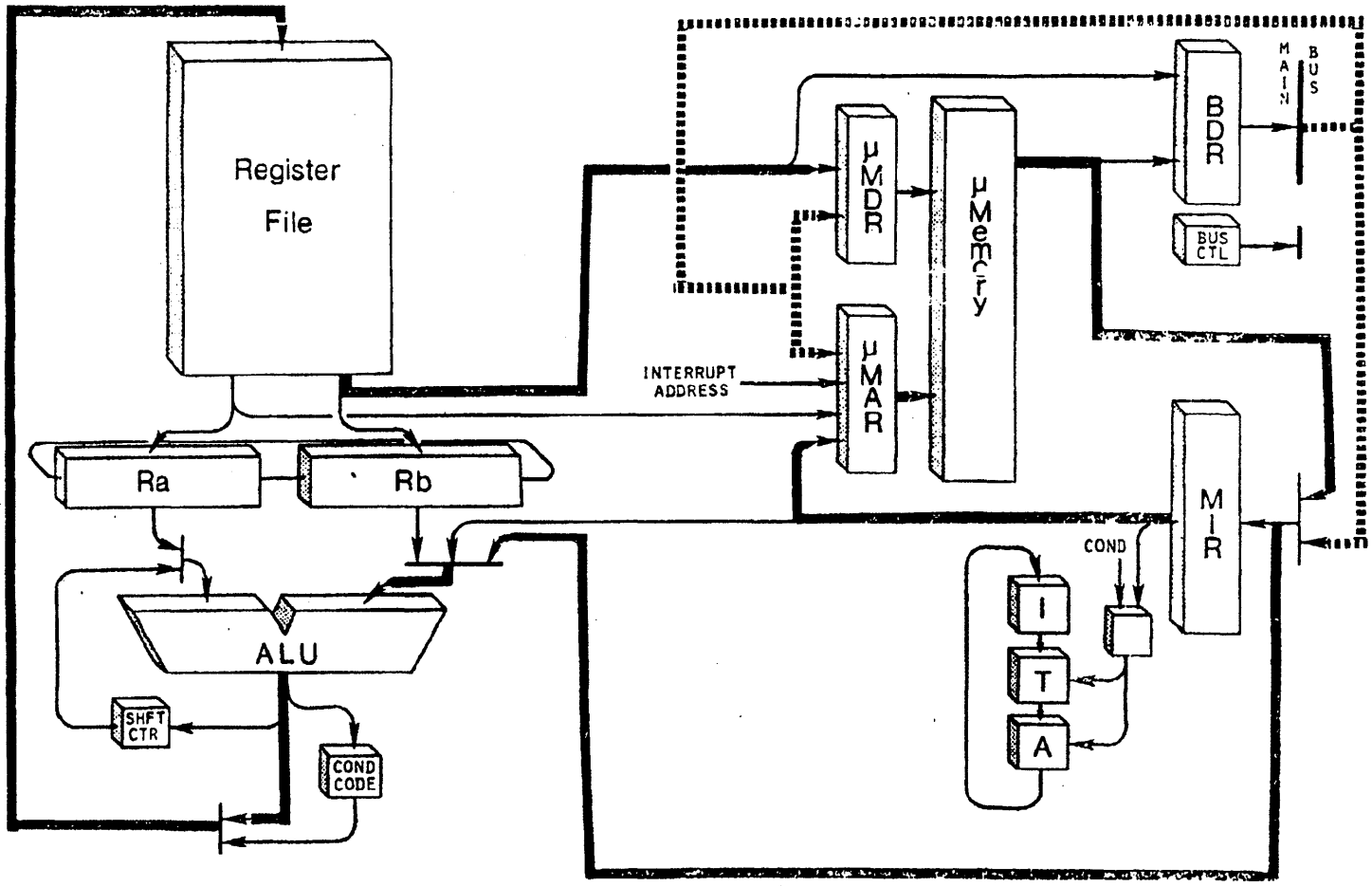


Figure 2.4 A-Machine Sequence

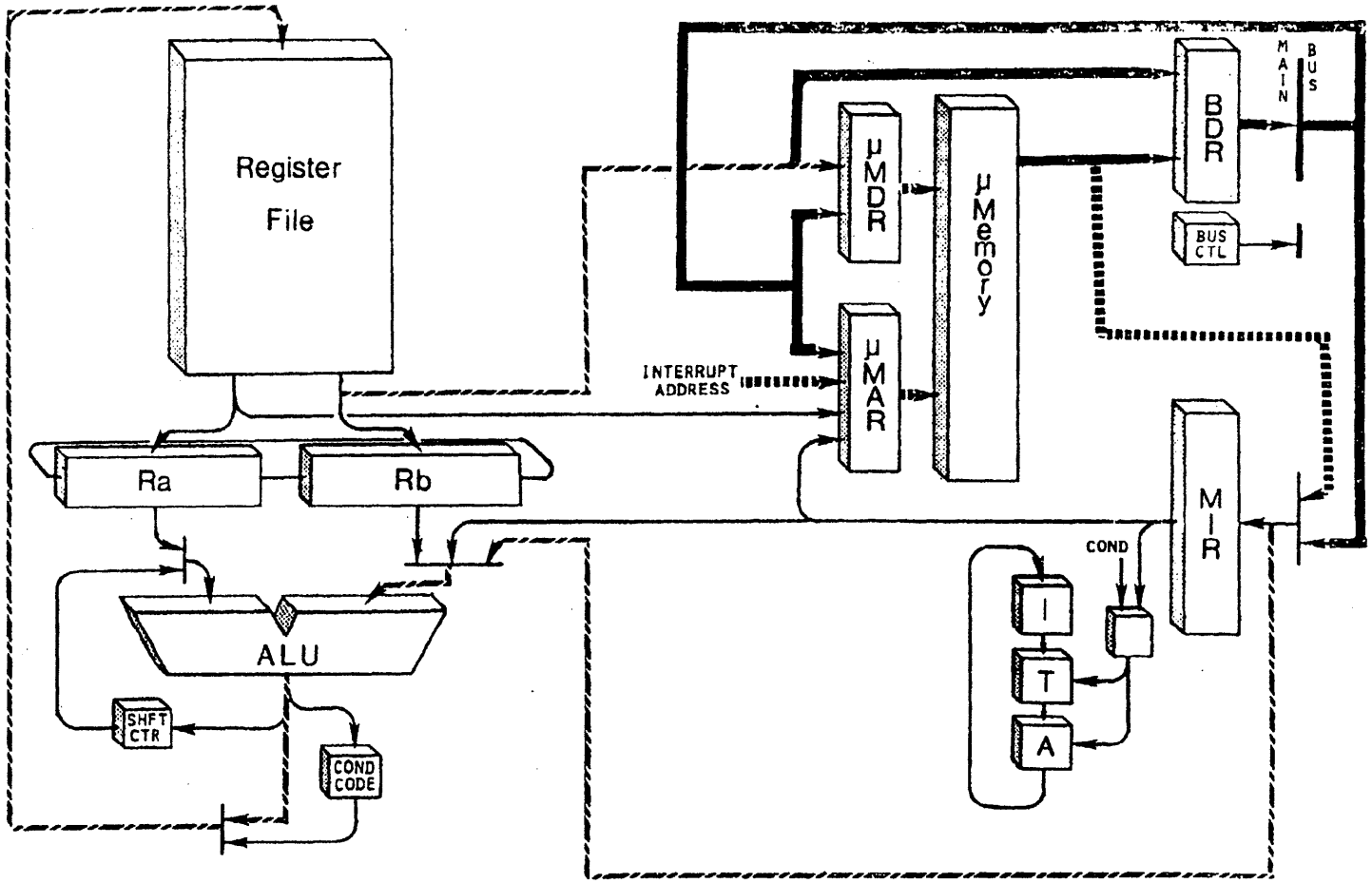
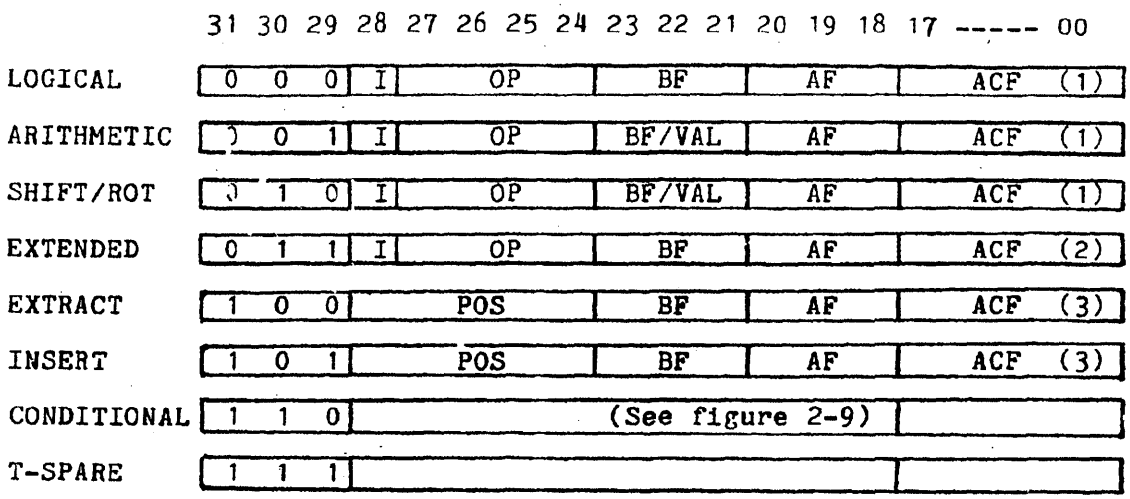


Figure 2.5 Special Sequences



Notes: The ACF field is used as follows:

(1) As immediate data or as an A-machine instruction depending upon the I bit (28):

I=0 => A-machine instruction
I=1 => Immediate data

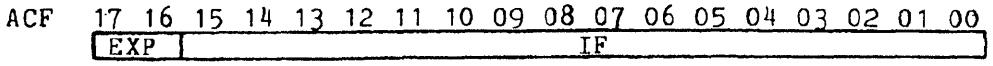
(2) As A-machine instruction or A-machine NOP depending upon the I bit (28):

I=0 => A-machine instruction
I=1 => A-machine NOP (skip A-cycle)

(3) As immediate data only

FUNCTIONAL MICROINSTRUCTION FORMATS

FIGURE 2-6

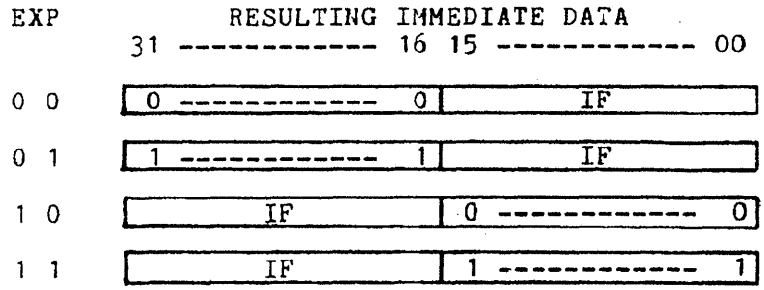


EXP -- Expansion specification
0 Right justify IF field
1 Left justify IF field

0 Zero fill remainder
1 One fill remainder

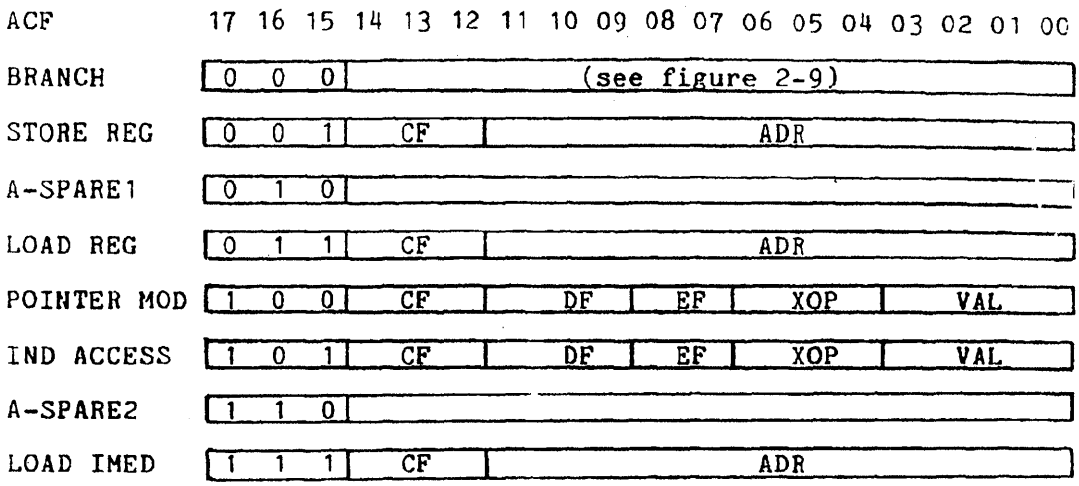
IF -- Sixteen bit immediate data quantity

Notes: Example of expansion



EXPANSION OF ACF FIELD TO FORM IMMEDIATE DATA

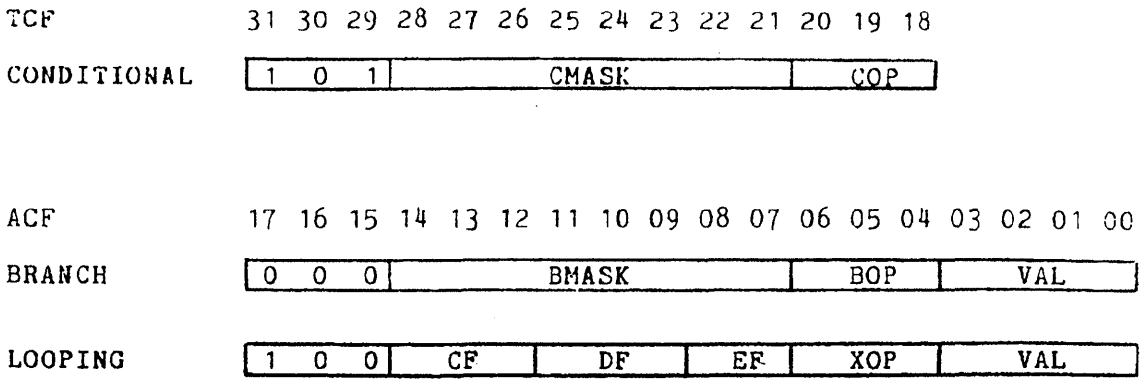
FIGURE 2-7



Notes: The pointer modification instruction also acts as a procedural instruction (see figure 2-9).

MEMORY MICROINSTRUCTION FORMATS

FIGURE 2-8

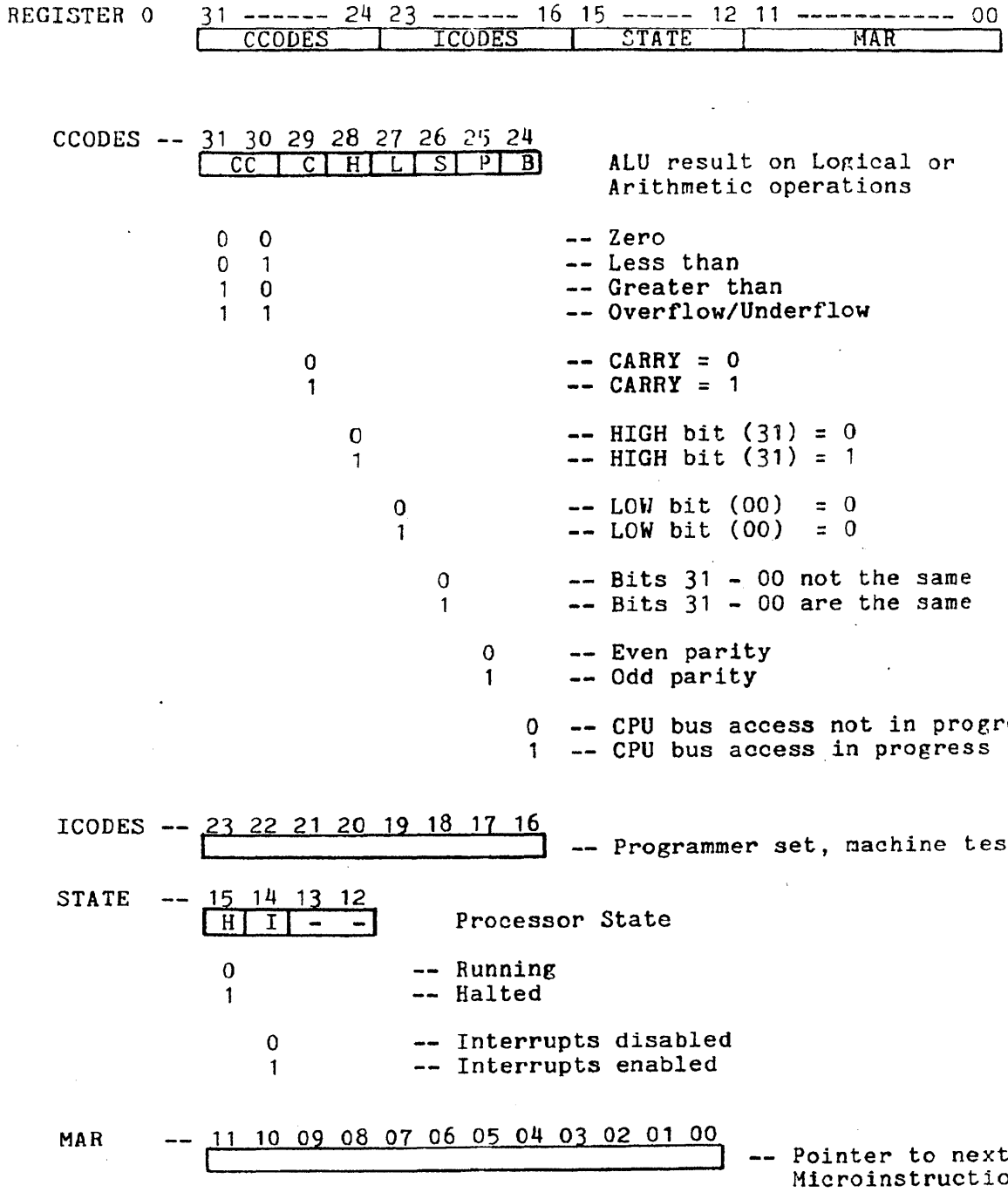


Notes: 1) Conditional microinstruction controls the A-machine execution of the ACF field.

2) 'Looping' is another aspect of the Pointer Modification microinstruction.

PROCEDURAL MICROINSTRUCTION FORMATS

FIGURE 2-9



FORMAT OF REGISTER 0 (PROCESSOR STATE WORD)

FIGURE 2-10

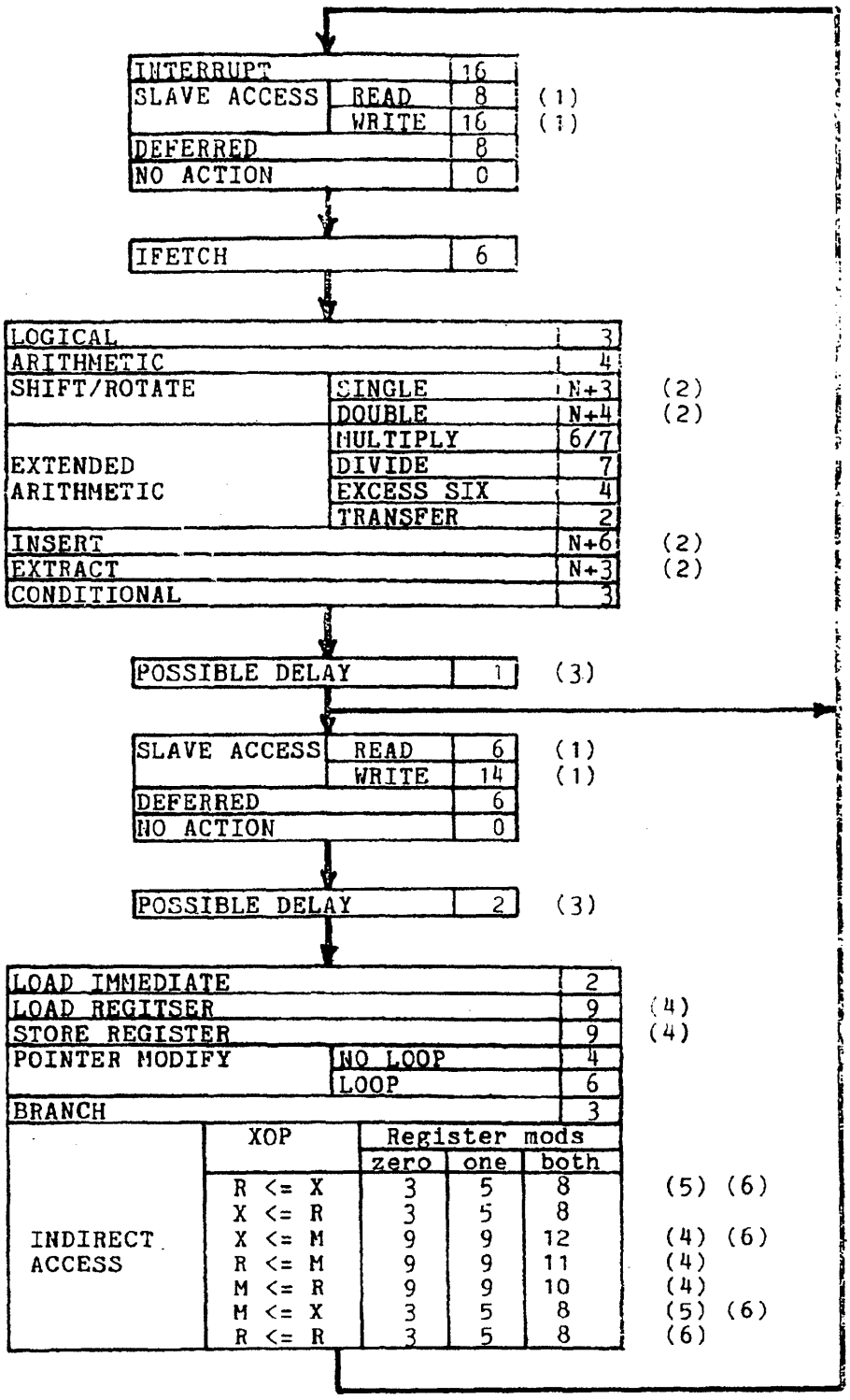


FIG. 2-11 TIMING ESTIMATION
(See notes on following page)

NOTES FOR FIGURE 2-11

NOTE 1: Times for slave access to the EMMY CPU assume that the accessing master device acts instantaneously. Thus the times shown are minimum times and do not take into consideration device characteristics.

NOTE 2: 'N' stands for the number of bit shift steps required to perform the operation.

NOTE 3: A delay is possible here if the control store is still busy serving a previous request and, it is required by one of the A-machine instructions indicated in note (4). Delays are as follows:

Delay of 1 cycle - If the contol store is still busy with the IFETCH because the T-machine operation executed in fewer than 3 cycles. This currently applies only to the extended arithmetic transfer.

Delay of 2 cycles - If a slave access or deferred operation takes place between the T- and A-machine cycles. The delay occurs even if the operation involves only the register files.

NOTE 4: Operations shown require immediate use of the control store and thus may affect the delay mentioned above.

NOTE 5: Operations shown require a deferred access which will occur at some future point in time.

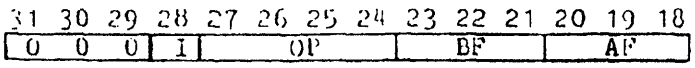
NOTE 6: An indeterminate delay is associated with the bus operations shown since the CPU will not continue processing until bus access is granted and the bus address is accepted as valid. Aside from delay caused by contention on the system bus, a delay may may occur if a previous CPU bus request is still outstanding.

3. MICROINSTRUCTION SYNTAX AND SEMANTICS

In this section the bit syntax and semantics of the EMMY microinstruction set is given. Each microinstruction class (e.g. Logical) is explained on a separate page for convenient reference.

The following symbols have been used to denote commonly encountered operations:

-	Logical NOT
+	Logical OR
⊕	Logical EXCLUSIVE OR
*	Logical AND
¦	Concatenation of bit fields
plus	Two's complement addition
EXT	External memory (i.e. the system bus)
MEM	Control store memory
REG	Register file memory
MAR	Microaddress register (bits 0 to 11 of Register 0)



0 OP2 <= REG[BF]
1 OP2 <= EXPANDED IMMEDIATE FIELD

OP				Semanics	Name
0	0	0	0	-REG[AF]	COMPLEMENT
0	0	0	1	-(REG[AF]*OP2)	NAND
0	0	1	0	-REG[AF]+OP2	
0	0	1	1	LOGICAL 1	ONES
0	1	0	0	-(REG[AF]+OP2)	NOR
0	1	0	1	-OP2	COMPLEMENT NUMBER
0	1	1	0	-(REG[AF]⊕OP2)	XNOR
0	1	1	1	REG[AF]+-OP2	
1	0	0	0	-REG[AF]*OP2	
1	0	0	1	REG[AF]⊕OP2	XOR
1	0	1	0	OP2	TRANSFER
1	0	1	1	REG[AF]+OP2	OR
1	1	0	0	LOGICAL 0	CLEAR
1	1	0	1	REG[AF]*-OP2	
1	1	1	0	REG[AF]*OP2	AND
1	1	1	1	REG[AF]	TEST

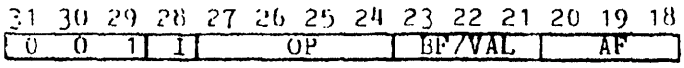
NOTES: 1) Result is returned to REG[AF].

2) All condition codes except OVERFLOW are set as required.

2) OP field is the same as function select on MC10181.

3) Operators used are:

+ stands for 'OR'
⊕ stands for 'EXCLUSIVE OR'
- stands for 'NOT'
* stands for 'AND'



0 IOP <= REG[BF]
1 IOP <= EXPANDED IMMEDIATE FIELD

0 Store result in REG[AF] and set condition codes
1 Set condition codes only

0 OP2 <= IOP
1 OP2 <= VAL

0	0	REG[AF] plus - OP2 plus 1
0	1	REG[AF] plus - OP2 plus C
1	0	REG[AF] plus OP2
1	1	REG[AF] plus OP2 plus C

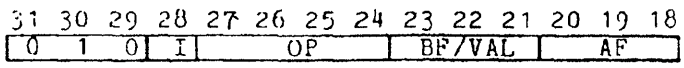
NOTES: 1) Conditions codes are set on result.

2) OVERFLOW results when carry into sign bit is not the same as the carry out of the sign bit.

3) VAL field is not sign extended when used as OP2.

4) The operator 'plus' is two's complement addition.

--- SHIFT/ROTATE ---



0 P <= REG[BF]
1 P <= EXPANDED IMMEDIATE FIELD

0 Single length operation
1 Double length operation

0 Shift amount is specified by P
1 Shift amount is specified by VAL

0 0 LEFT SHIFT LOGICAL
0 1 LEFT ROTATE
1 0 RIGHT SHIFT LOGICAL
1 1 RIGHT SHIFT ARITHMETIC

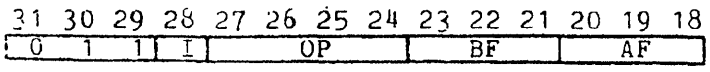
NOTES: 1) Condition codes are not set.

2) Single length shift: REG[AF] is source and destination.

3) Double length shift: REG[AF] is source and destination of high order 32 bits.
REG[AF⊕1] is source and destination of low order 32 bits.

4) Bit 25 is direction: 0 => LEFT
1 => RIGHT

5) On RIGHT SHIFT ARITHMETIC the sign bit is preserved.



0 Execute A-machine instruction
1 Skip A-machine instruction

OP				Name
0	0	0	0	UNASSIGNED
0	0	0	1	--
0	0	1	0	--
0	0	1	1	--
0	1	0	0	--
0	1	0	1	--
0	1	1	0	--
0	1	1	1	--
1	0	0	0	--
1	0	0	1	--
1	0	1	0	--
1	0	1	1	UNASSIGNED
1	1	0	0	DIVIDE STEP
1	1	0	1	TRANSFER
1	1	1	0	EXCESS SIX
1	1	1	1	MULTIPLY STEP

NOTES: 1) DIVIDE STEP

initialize: REG[AF]|REG[AF⊕1] is dividend
REG[BF] is divisor

finalize: REG[AF] is remainder
REG[AF⊕1] is quotient

Sequence: 1) if REG[AF] minus REG[BF] ≥ 0
then REG[AF] <= REG[AF] minus REG[BF]
2) Shift REG[AF]|REG[AF⊕1] LEFT LOGICAL by one bit.
3) If result of step 1 was ≥ 0
then shift 1 into REG[AF⊕1]
4) No condition codes are set.

2) TRANSFER REG[AF] <= REG[BF]
(no condition codes are set)

--- (Continued on next page) ---

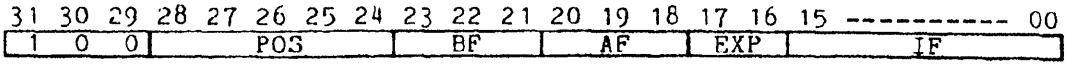
3) EXCESS SIX For each 4 bit digit position of REG[BF] which is greater than 9 (1001) store 6 (0110) in the corresponding digit position of REG[AF]. No condition codes are set

4) MULTIPLY STEP
initialize: REG[BF] is multiplicand.
REG[AF⊕1] is multiplier.

finalize: REG[AF]¦REG[AF⊕1] holds double length result.

Sequence: 1) Shift REG[AF]¦REG[AF⊕1] RIGHT ARITHMETIC by one bit.
2) If OVERFLOW was set on previous MULT STEP then Complement sign bit of REG[AF]
3) If bit 0 of REG[AF⊕1] was a 1
then REG[AF] <= REG[AF]+REG[BF]
Set OVERFLOW if necessary
else clear OVERFLOW bit
4) No condition codes are set.

--- EXTRACT ---



POS -- Amount of LEFT ROTATE

EXP IF
(immediate mask data)

NOTES: 1) Sequence: 1) Left rotate contents of REG[BF] by amount specified by POS field.
2) 'AND' with MASK generated from expanded ACF field.
3) Place result in REG[AF].

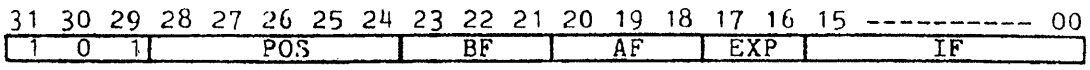
2) Algebraically, the sequence is defined as:

REG[AF] <= (left rotate(REG[BF],POS))*MASK

3) MASK is expanded immediate field.

4) Condition codes are not set.

--- INSERT ---



POS -- Amount of LEFT ROTATE

EXP IF
(immediate mask data)

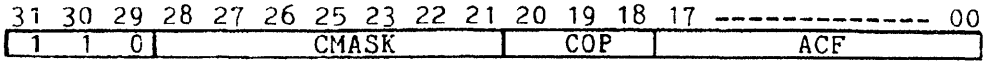
NOTES: 1) Sequence: 1) Left rotate contents of REG[BF] by amount specified by the POS field.
2) Generate a MASK from the expanded ACF field.
3) The result from step (1) will be 'inserted' into REG[AF] bit by bit where ever the the corresponding bit position of the MASK is '1'. Where a bit position of the MASK is '0' then the corresponding bit position of REG[AF] is unchanged.

2) Algebraically, the sequence above may be expressed as:

REG[AF] <= (Left Rotate(REG[BF],POS))*MASK)+(REG[AF]*-MASK)

3) MASK is immediate data from expanded ACF field.

4) No condition codes are set.



CMASK -- Mask for code bit selection

COP			-- Test Specification
V	C	S	
0			Normal sense
1			Inverted sense
	0		Normal codes
	1		Inverted codes
		0	Test condition codes
		1	Test indicator codes

ACF -- (Conditionally executed A-machine instruction)

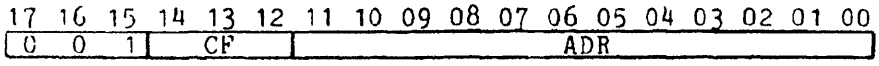
NOTES: 1) Skip A-machine instruction specified by the ACF field if:

$$V \oplus \bigcup_{i=0}^{7} (CMASKi * (C \oplus ((-S*CCODEi) + (S*ICODEi))))$$

2) Semantically, the 'S' bit selects either the condition codes or the indicator codes for testing. The particular bits of the selected codes to be tested are specified by 1's in the CMASK field. Bits 'V' and 'C' specify the test of these bits as follows:

V	C	Skip A-machine instruction if:
0	0	Any bit is set
0	1	Any bit is not set
1	0	All bits are not set
1	1	All bits are set

--- STORE REGISTER ---



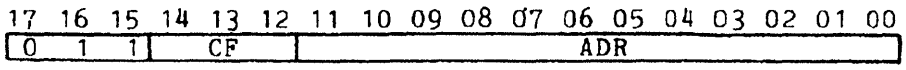
CF -- Source register

ADR -- Destination address in micromemory

NOTES: 1) The register designated by the CF field is stored in the micromemory address given in the ADR field. Algebraically:

MEM[ADR] <= REG[CF]

--- LOAD REGISTER ---



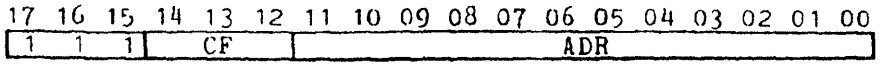
CF -- Destination register

ADR -- Source address in micromemory

NOTES: 1) The contents of the micromemory address specified in the ADR field is loaded into the register specified by CF. Algebraically:

REG[CF] <= MEM[ADR]

--- LOAD IMMEDIATE ---

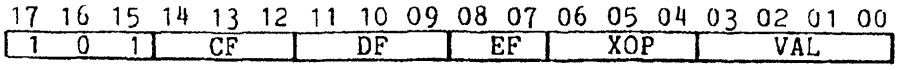


CF -- Destination register

ADR -- Immediate data

NOTES: 1) The register designated by CF is loaded with the immediate data specified by the ADR field. ADR is sign extended to form the 32 bit immediate data quantity.

--- INDIRECT ACCESS ---



CF -- Destination address pointer

DF -- Source address pointer

EF -- Modification specification

EF		
0	0	No modification
0	1	REG[DF] <= REG[DF] plus VAL
1	0	REG[CF] <= REG[CF] plus VAL
1	1	REG[DF] <= REG[DF] plus VAL (and) REG[CF] <= REG[CF] plus VAL

XOP -- Transfer specification

XOP			
0	0	0	MEM[REG[CF]] <= EXT[REG[DF]]
0	0	1	REG[CF] <= EXT[REG[DF]]
0	1	0	EXT[REG[CF]] <= REG[DF]
0	1	1	EXT[REG[CF]] <= MEM[REG[DF]]
1	0	0	REG[CF] <= MEM[REG[DF]]
1	0	1	MEM[REG[CF]] <= REG[DF]
1	1	0	REG[CF] <= REG[DF]
1	1	1	Unassigned

VAL -- Immediate value for pointer modification

NOTES: 1) Abbreviations in transfer specification are:
REG -- Register file address
MEM -- Micromemory address
EXT -- External bus system address

2) VAL is sign extended to form a 32 bit quantity for use in the pointer modification step which follows the transfer step.

3) When external bus operations are specified the address pointer register involved is used as follows:

Bits	Usage
15 - 00	Unit internal address
23 - 16	Unit address
31 - 24	Command

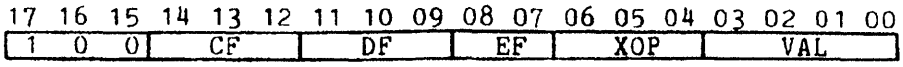
--- (Continued on next page) ---

Bit 24 of the command word is ignored and the corresponding bit on the system bus is set explicitly by the transfer specification as follows:

XOP			Bit 24
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1

4) The access busy bit of register 0 is set whenever an external access is started and cleared when the access is completed. Once the external access has been started the CPU may continue to execute microinstructions provided the external memory is not accessed while the busy bit is '1'. If an external access does occur while the busy bit is set the CPU will cease execution until the busy bit is cleared (i.e. when the previous CPU issued bus operation finishes).

--- POINTER MODIFICATION and LOOP ---



CF -- Source and sink register

DF -- Source register

EF -- Modification specification
(REG[CF] gets result)

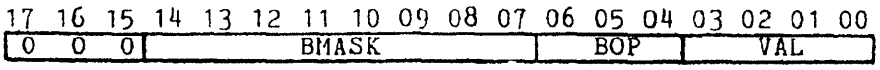
0	0 REG[CF] plus 1
0	1 REG[CF] plus -1
1	0 REG[CF] plus REG[DF]
1	1 REG[CF] plus -REG[DF] plus -1

XOP -- Loop specification

0	
1	Loop if REG[CF] < 0
0	
1	Loop if REG[CF] = 0
0	
1	Loop if REG[CF] > 0

VAL -- Amount of relative branch

NOTES: 1) VAL is sign extended
2) Sequence: 1) Perform operation specified in EF field and place result in REG[CF]
2) Test the resulting contents of register CF as specified by the XOP field.
3) If the loop test is 'true'
then MAR <= MAR plus VAL
else MAR is unchanged



BMASK -- Mask for code bit selection

BOP -- Test specification

V	C	S	
0			Normal sense
1			Inverted sense
	0		Normal codes
	1		Inverted codes
		0	Test condition codes
		1	Test indicator codes

VAL -- Amount of relative branch

NOTES: 1) VAL is sign extended to form amount of relative branch

2) MAR <= MAR plus VAL if:

$$V \oplus \bigcup_{i=0}^{7} (BMASKi * (C \oplus ((-S*CCODEi) + (S*ICODEi))))$$

3) Semantically, the 'S' bit selects either the condition codes or the indicator codes for testing. The particular bits of the selected codes to be tested are specified by '1's in the BMASK field. Bits 'V' and 'C' specify the test of these bis as follows:

V	C	Branch to MAR plus VAL if:
0	0	Any bit is set
0	1	Any bit is not set
1	0	All bits are not set
1	1	All bits are set

4. BUS SYSTEM INTERFACING

This chapter supplies the basic information necessary for an EMMY system user to design and interface units with the processor bus system. Both logical and electrical considerations are discussed. To determine the specific pin assignments on the system backplane and card slot assignments the user should refer to the applicable system schematic drawings and wire list documentation.

4.1 Inter-unit Communication Philosophy

The EMMY processor bus system is a bi-directional, 32 bit wide data bus which makes use of a fully interlocked, asynchronous data transfer scheme. Control is distributed among the bus units, and any bus unit which is electrically and logically capable may gain control of the bus system for the purposes of transfering data. Bus access is based on a 'simple' priority system in which the highest priority device is always granted access.

Conceptually a complete bus operation sequence may be divided into two sub-sequences:

1) Access sequence, and
2) Data Transfer sequence.

During the access sequence, bus devices bid for control of the bus system. A device which gains control of the bus during this sequence is refered to as a 'master' device, and this device controls the subsequent data transfer cycle. During this transfer cycle the device will issue an address on the bus system which will designate another bus unit as a participant in the data transfer. This unit is refered to as a 'slave' unit. All unaddressed units remain inactive with respect to the bus during the data transfer phase of the bus operation. Any unit may be designed to have 'master' and/or 'slave' capability. The CPU, for example, has both accessing capabilities.

In addition to the data transfer capability outlined above, the bus also contains a group of direct signalling lines which are related to control and status functions of the CPU.

4.2 Bus Line Semantics

4.2.1 Electrical Semantics

Electrically, the processor bus system is based on open collector TTL logic. Thus, a "1" on the bus (assuming positive device logic) corresponds to 0 volts on the actual bus lines, and, conversely, a "0" corresponds to +5 volts. In this section we will refer to the bus signals "1" and "0" as they appear to a

device before transmission and after reception.

4.2.2 Logical Semantics

Before giving the details of inter-unit communications on the EMMY bus system we will briefly outline the semantics of the bus lines from a logical standpoint. These lines may be grouped as follows:

1) Direct Lines -- Used to directly indicate and control the status of the EMMY CPU.

2) Access Control Lines -- Used to resolve multiple requests for bus service.

3) Transfer Control Lines -- Used to synchronize data transfer between a master and slave unit.

4) Data Lines -- Used to transfer address and data information between units.

4.2.2.1 Direct Lines

The eight direct lines are divided into two groups: control and status. These lines may be used by bus units to sense CPU status and to influence the status of the CPU and other bus units.

Direct Status Lines

RUN/HALT - 1 => CPU is halted
0 => CPU is running

PARITY ERROR - Indicates a control memory parity error has been detected.

TIMEOUT - Indicates that a bus operation has not been completed within 75 usec.

INTERRUPT - 1 => Interrupts enabled
0 => Interrupts disabled

Direct Control Lines

RUN - Signals the CPU to enter run mode.

HALT/STEP - Signals the CPU to halt. When the CPU is halted a signal on this control line will cause the CPU to execute one additional microinstruction.

PARITY RESET - Resets the CPU parity error condition.

MASTER CLEAR - Signals all bus units to initialize.

4.2.2.2 Access Control Lines

At the start of a bus cycle the four access control lines are used by all devices which are requesting master status to resolve possible conflicts.

BUSCLK - This line is driven by a 100 nsec clock which operates whenever the bus is idle and seeking requests.

REQUEST/USING - Signals that one or more devices are requesting the bus, or that a single device is currently in master status and is using the bus.

AVAILABLE - This signal is chained from one potential master unit to the next on the bus. When REQUEST/USINGs "1" the semantics of AVAILABLE at the input of a unit are:

1 => Bus is available for access by receiving unit.

0 => Bus is being used by a higher priority unit.

Semantics of AVAILABLE at a unit output are:

1 => AVAILABLE at the unit input is one and the unit does not wish to use the bus.

0 => AVAILABLE at the input is "0" or AVAILABLE at the input is "1" and the unit is using the bus.

CREJ - When "1", this line indicates that an access control cycle has completed.

4.2.2.3 Transfer Control Lines

The six transfer control lines are used by master and slave units to synchronize the transfer of data on the bus data lines.

ASIG - Indicates that the bus data lines contain valid address information.

ISIG - Indicates that the bus data lines contain a valid CPU interrupt vector.

AACK - Indicates that bus data has been recognized as a valid address by a slave unit or as an interrupt vector by the CPU.

REJECT - Indicates that information on the data lines has been recognized as a valid address by a slave unit, but that the slave unit cannot respond at the present time.

DSIG - Indicates that the bus data lines contain valid data.

DACK - Indicates that data on the bus data lines has been accepted by the receiving bus unit.

4.2.2.4 Data Lines

The thirty two data lines are used for the transfer of address, interrupt vector and data information between communicating bus units.

4.3 Sequencing of Bus Operations

A complete bus operation consists of two phases: competition among requesting master units for control of the bus system and an asynchronous transfer of data between a single master and slave unit pair. Figure 4-1 schematically illustrates the sub-structure of a typical 'master' device. Such a device has three internal controllers:

1) Device controller,
2) Access controller, and
3) Transfer controller.

The device controller works directly with the device itself and its associated internal storage mechanism. The access controller handles the first phase of the bus sequence, gaining control of the bus, while the transfer controller oversees the transfer of data to the 'slave' bus unit once bus access has been obtained. The three internal controllers are linked logically by three internal signals:

1) REQ - Signals the access controller that the device controller is ready to initiate a bus data transfer operation.

2) GO - Indicates to the transfer controller that the

access controller has obtained the bus.

3) DONE - Used by the transfer controller to indicate to the device controller that the transfer operation is complete.

In the following sections we will outline the logical structure of the access and transfer controllers. The logical structure of the device controller is, of course, dependent upon the characteristics of the particular device involved.

4.3.1 Logical Structure of the Access Controller

Logically speaking, the access controller responds to requests (via the REQ line) for bus service from the device controller by using control access lines to obtain control of the bus system. After doing this it signals the transfer controller (via GO) to begin the transfer operation. In gaining control of the bus the access controller uses the following lines:

1) BUSCLK
2) REQUEST/USING
3) AVAILABLE (IN and OUT)
4) REJECT (a signal from the transfer controller)
5) MASTER CLEAR

The AVAILABLE line originates at the bus controller and is chained down the bus system from one potential master unit to the next. Priority on the processor bus system is determined by access controller position on the AVAILABLE line with the device electrically closest to the bus system controller having the highest priority and the ellectrically most distant access controller having lowest priority. This chaining is illustrated in figure 4-2. Note that the AVAILABLE(OUT) line of one unit become the AVAILABLE(IN) line of the next. Further AVAILABLE(OUT) from the last unit in the chain is returned to the bus system controller where it is 'OR'ed with the ASIG and ISIG transfer control lines to generate CREJ. Thus, semantically, the CREJ line is used to signal all bus units that an access cycle has been completed.

Figure 4-3 sumarizes the logical requirements placed on access controller operation in the form of a state diagram. A controller begins its request sequence in state 0 (idle). When the bus itself is idle the REQUEST/USING LINE will be "0" and the BUSCLK line will show a 100 nsec clock signal. On the rising edge of this clock all master devices receiving REQ from their associated device controllers will enter state 1 (requesting bus).

Each access controller in state 1 will send "1" on the REQUEST/USING line. REQUEST/USING signals from all access controllers are in the system are 'OR'ed on the bus since this is

an open collector system. The bus system controller upon receiving this signal will stop the BUSCLK line and issue a "1" on the AVAILABLE line. AVAILABLE passes from the highest priority master unit to the lowest. If a master unit is in state 0 (idle) it should allow AVAILABLE to continue to the next unit in the chain. However, if a master unit is in state 1 (requesting bus) it should not pass AVAILABLE and instead transition to state 2 (using bus).

At this point, of those units which initially requested the bus only a single master unit will be in state 2. This unit will continue to send "1" on the REQUEST/USING line and will issue a GO signal to its associated transfer controller. During the operation of the transfer controller an ASIG or ISIG will be issued. Either of these signals will cause the bus controller to send "1" momentarily on the CREJ line. Semantically, this event indicates that an access cycle has ended, and all units in state 1 should return to state 0 to await a new bus access cycle.

The current bus master will remain in state 2 awaiting the completion of the data transfer phase of the bus operation. If the transfer operation completes normally, the transfer controller will signal DONE to the device controller which in turn will set REQ to "0" returning the master to state 0. In state 0 the master will release the REQUEST/USING line which in turn causes the bus controller to set AVAILABLE to "0" and restart the system clock. The access cycle then begins anew.

Alternatively, the operation of the transfer controller may cause a REJECT signal from the selected slave device, indicating that the slave is busy servicing a previous request. In this event the transfer controller will terminate the transfer sequence and the access controller will release REQUEST/USING and enter state 3.

Once a REJECT is received the rejected master unit enters a state sequence in which it must wait for three complete access cycles to pass before being allowed to obtain bus assess again. This is represented in figure 4-3 by the sequence (3-4-5-6-7-8). This state sequence is paired as follows: (3-4), (5-6) and (7-8). Transitions between state pairs occur when the access controller sees the CREJ signal. Transitions within a state pair occur when the bus clock rises to "1". In states 4,6 and 8 the accesss controller must signal "1" on the REQUEST/USING line, however, it must allow AVAILABLE to pass through to lower priority devices. This allows devices other than the rejected unit to gain the bus. If, however, no other devices are requesting the bus, the AVAILABLE(OUT) signal from the last access controller will cause the system bus controller to signal CREJ and thus advance the state of the rejected access controller. In states 3, 5 and 7 the rejected access controller must lower REQUEST/USING (as if it were in state 0) and allow the bus clock to run. This will assure that eventually the rejected access controller will be reset to state 0

within three bus cycles even if it is the only device requesting the bus.

A device controller may elect to take special action when rejected. It may, for example, remove its request to the access controller (i.e. set REQ to "0"). In any event the access controller will allow at least 3 complete access cycles to pass before allowing a new REQ signal to be answered.

A "1" on the MASTER CLEAR line, by definition, should set all access controllers to state 0 and reinitialize their associated device and transfer controllers.

4.3.2 Logical Structure of the Transfer Controller

The transfer controller oversees the transmission of address and data information on the bus data lines. Typically, the transfer control phases consists of two sub-phases in which the data lines are used first for address transmission and then for data transmission. Each transmission is controlled in an asynchronous manner by making use of the six transfer control lines:

1) ASIG
2) ISIG
3) AACK
4) REJECT
5) DSIG
6) DACK

During the address transmission a slave device is identified. During the data transmission data information is passed between the master and the selected slave device.

Currently, three types of bus transmission sequences are defined:

1) READ - A master unit receives data from a slave unit,
2) WRITE - A master unit sends data to a slave unit, and
3) INTERRUPT - A master unit sends an interrupt vector to the CPU

The READ and WRITE operations are essentially similar (except for the direction of data transmission) and both involve address and data transmission cycles. The INTERRUPT operation, however, requires only an address transmission.

4.3.2.1 Address Transmission Sequence

The standard address transmission sequence, as shown in figure 4-4a, involves the use of the data lines and the ASIG and

AACK transfer control lines. The master unit begins by placing address information on the bus data lines. After a delay of at least 60 nsec, to allow for data skewing on the bus, it signals with a "1" on ASIG. Upon receiving ASIG all slave devices on the bus compare the address sent to their own internally set address. If the address is valid, a single slave unit will recognize it. This unit should store any portion of the address it needs for future reference (e.g. the command and internal address) and send a "1" on the on the AACK (address acknowledge) line. The master unit upon seeing AACK will remove the address information from the lines and "lower" ASIG. In response, the slave unit will "lower" AACK completing the address transmission cycle.

According to the transmission scheme discussed above the semantics of the ASIG and AACK lines are as follows:

ASIG	AACK	Semantics
0	0	Idle (transmission complete)
1	0	Valid address on data lines
1	1	Address recognized by a slave unit
0	1	Address removed from data lines

There are two variations on the address scheme described above: the interrupt and reject sequences.

The interrupt sequence (figure 4-4b) is essentially the same as the normal address transmission except that the ISIG line is used instead of the ASIG line to signal address validity. The low 12 bits of the interrupt vector sent on the bus data lines must be even and specifies an even-odd pair of control store locations. Register 0 is stored in the odd control store location, and a new register 0 is fetched from the even location of the pair. Control store addresses are assigned by convention to the various bus units for interrupt purposes as outlined in section 2.3.2.

Although the CPU may be interrupted at any time, bus devices should not attempt to interrupt the CPU when the INTERRRUPT line is "0". Interrupt sequences are not followed by a data transmission sequence.

A slave unit which is addressed by a master may respond with a REJECT signal instead of an AACK. This indicates that the slave, while recognizing its address, is currently unable to serve the master unit. This usually occurs when the slave is still busy completing a previous request. The reject sequence is shown in figure 4-4c. A master unit which is rejected will not begin a data transmission sequence. It may resubmit its request to the slave unit in accordance with the restrictions outlined in section 4.3.1.

During the address transmission sequence the thirty-two data lines are used to send address and command information according to the following convention:

Data Lines	Usage
Bits 15 to 00	Unit Internal Address
Bits 23 to 16	Unit Address
Bits 31 to 24	Command

Currently, slave unit addresses are assigned as follows:

Address	Unit
FF	CPU
FE	Console
FD	Datapoint Interface
00 -- 03	Main Memory

Bit 24, the low bit of the command, determines the direction of the data transmission on the bus:

Bit 24	Semantics
0	Read (master receives from the slave)
1	Write (master sends to the slave)

The other bits of the command may be used as required by the individual bus units.

4.3.2.2 Data Transmission Sequence

A data transmission sequence, as shown in figure 4-4d, follows the address transmission except in the case of an interrupt or reject sequence. Depending upon the command sent during the address transmission, either the master or the slave unit will be designated as the sender. The other unit will be the receiver (e.g. on Writes the master unit is the sender while the slave unit is the receiver). The sending unit begins by placing data on the data lines and after a 60 nsec deskewing delay sends a "1" on DSIG. Upon reception of DSIG the receiving unit stores from the data line and sends a "1" on DACK (data acknowledge). Upon receiving DACK, the sending unit removes data from the data lines and "lowers" DSIG. The response of the receiving unit is to lower DACK completing the data transmission sequence. At this point the the master device should release control of the bus system.

Semantically, the states of the DSIG and DACK lines may be represented as follows:

DSIG	DACK	Semantics
0	0	Idle (transmission complete)
1	0	Valid data on the data lines
1	1	Data latched at the receiving unit
0	1	Data removed from the data lines

4.3.2.3 Bus Error Conditions

A CPU timing circuit monitors the BUSCLK line and if BUSCLK is "0" for longer than 75 usec the TIMEOUT line is "raised" and the CPU is interrupted. A timeout interrupt indicates that a bus master has failed to complete its operation in the allocated time. Normally this is caused by addressing a non-existant slave unit.

4.4 Electrical Requirements of the Bus System

The processor bus system is based on open collector, non-Schottky TTL logic. Bus drivers in the system are 7438 and 8T26 gates capable of sinking 48ma and 40ma respectively. A central pull-up resistor of 220 ohms is provided.

Each bus unit shall interface to a bus line through a single non-Schottky TTL driver and receiver. Further, a bus unit shall not extend the bus electrically from the backplane without providing input/output buffering. The number of bus units is limited electrically (but not logically) to 9.

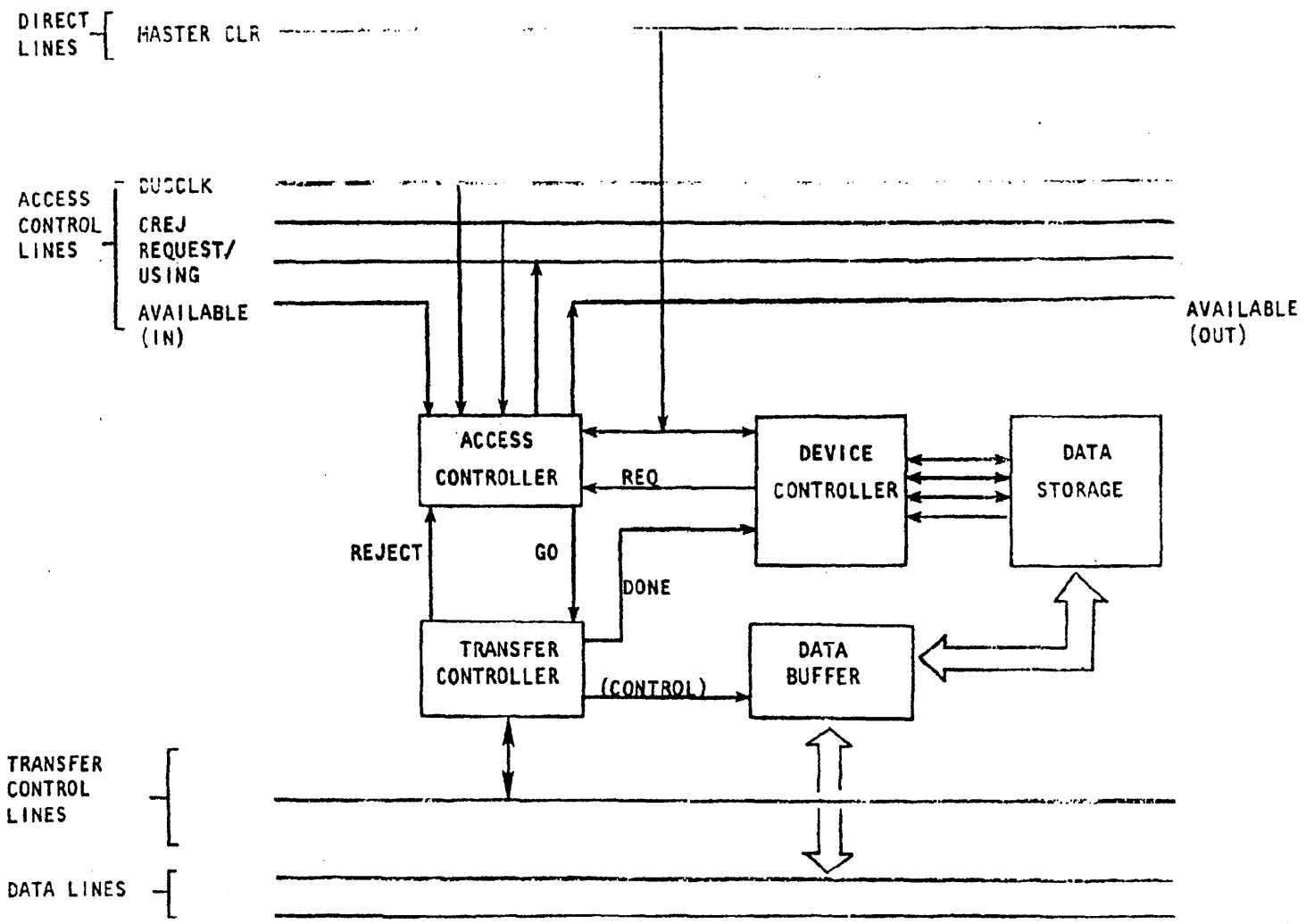


Figure 4.1 Schematic of a Bus Master Unit

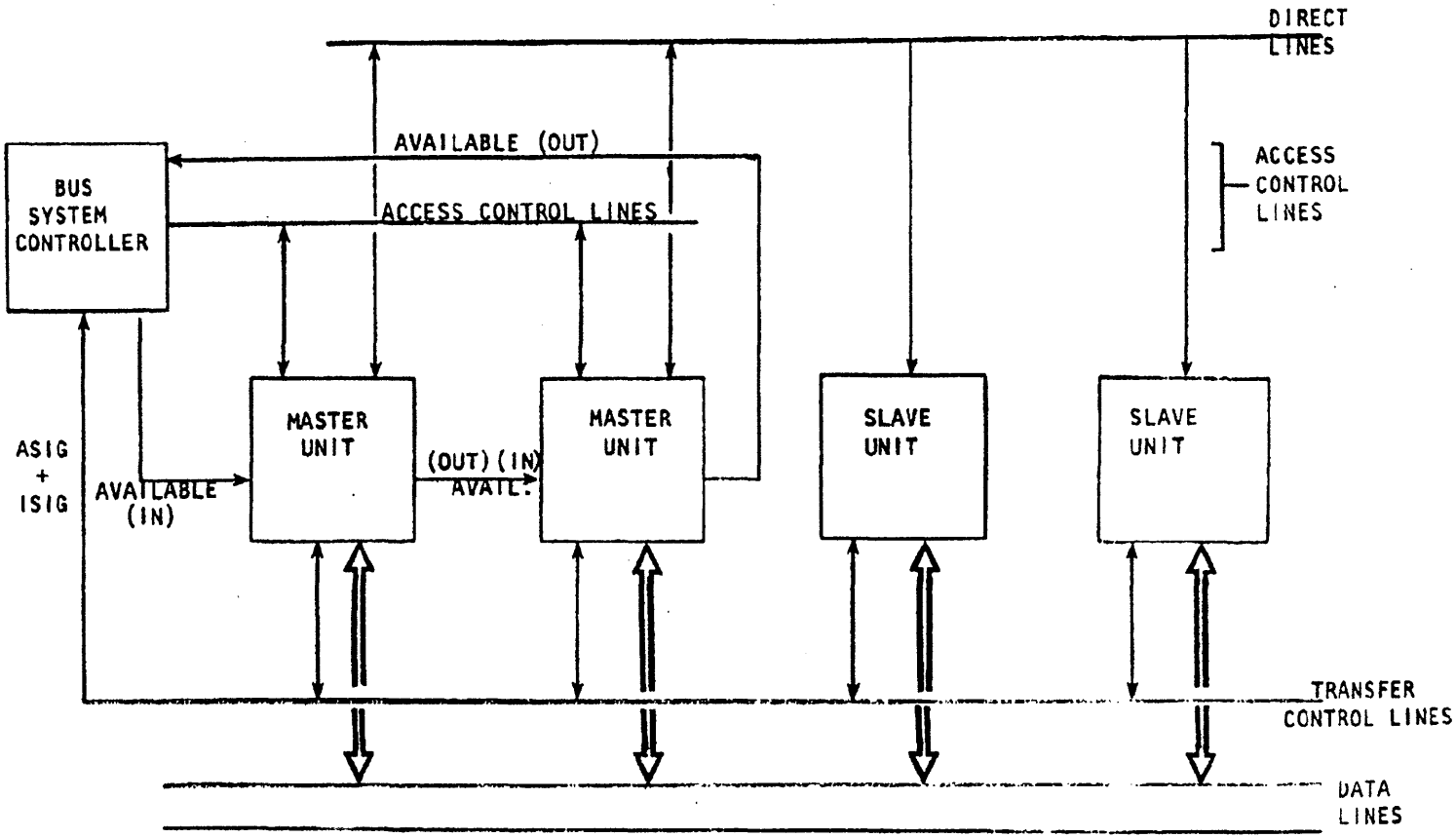


Figure 4.2 Bus System Structure

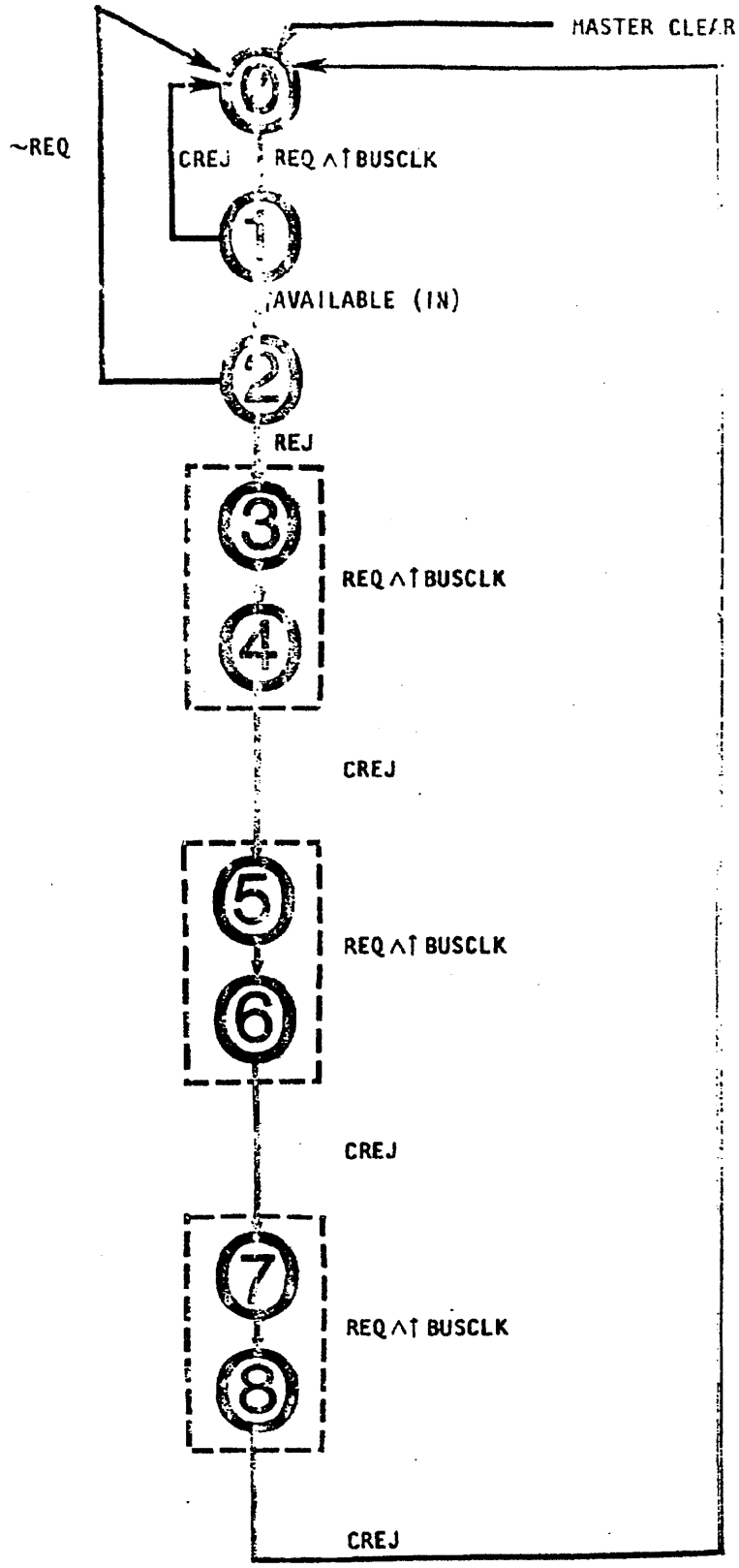


Figure 4.3 Access Controller Logic

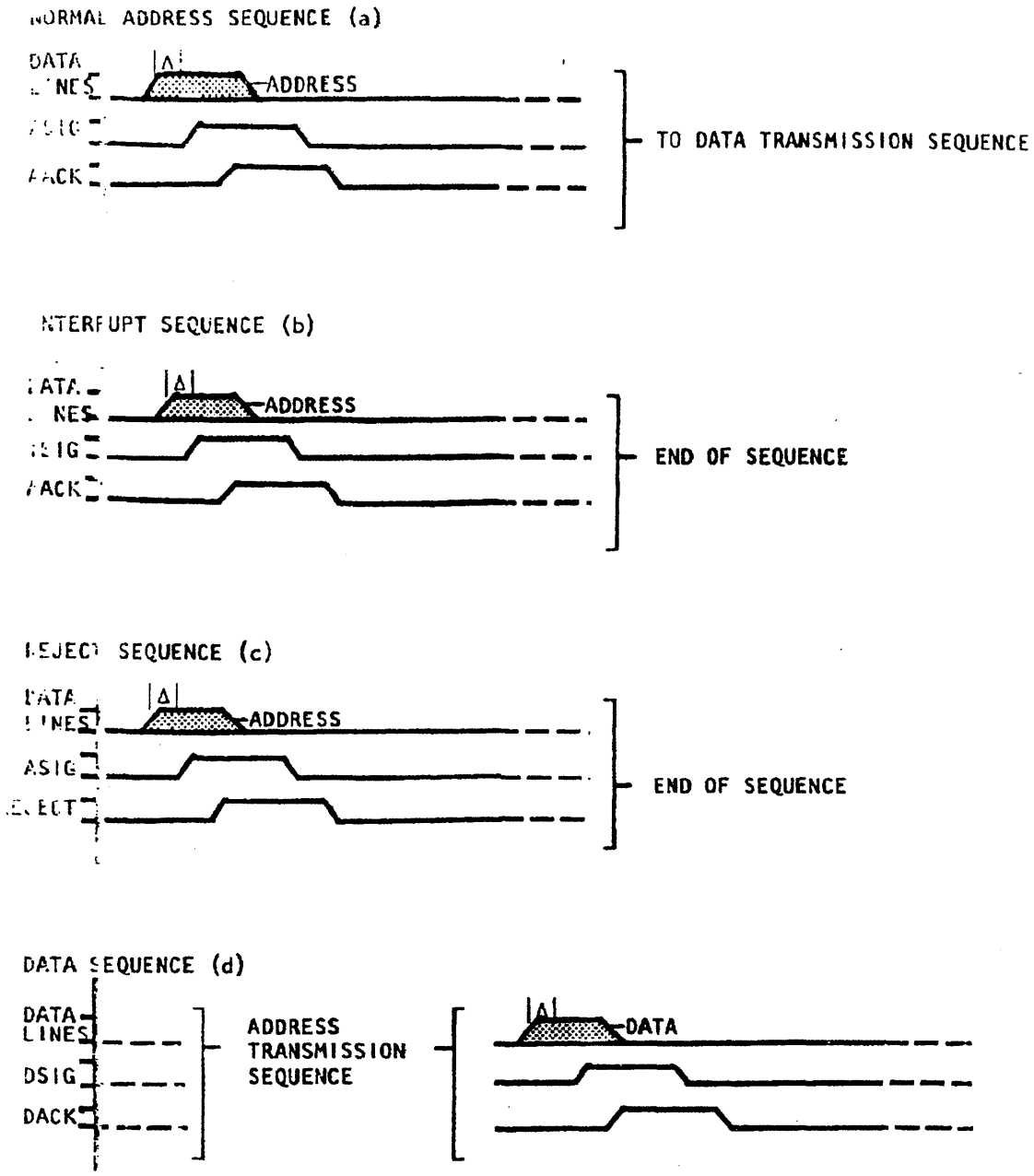


Figure 4.4 *Transfer Sequences*