AN EMULATION ORIENTED, DYNAMIC MICROPROGRAMMABLE PROCESSOR*

(VERSION 2)

by Charles Neuhuaser

ABSTRACT

This report is an update of an earlier report and specifies the design of a low cost, dynamic microprogrammable processor intended primarily as a tool for research and instruction. Changes to the earlier report include:

1) Specification of extended arithmetic operations (e.g. multiply)
2) Changes to method of accessing memory
3) Specification of additional A-machine instructions
4) Simplification of I/O structure
5) Specification of host machine bus structure
6) ISP-like description of the three internal sub-machines

* This work was supported by the U.S. Atomic Energy Commission under contract AT (11-1 3288)

DISCLAIMER

CONTENTS

SYSTEM DESCRIPTION OF THE JHU EMULATION LABORATORY AND HOST MACHINE

Charles Neuhauser

Electrical Engineering
Johns Hopkins University
Baltimore 21218 Maryland

1. Introduction

The emulation laboratory described in this paper is being implemented at Johns Hopkins University as a research and educational facility. Currently, we are planning to use this laboratory to support research in the following areas:

1) Examination of and experimentation with novel system architectures,

2) Evaluation of various directly executed language (DEL) structures and the effective structure of their associated base machines [4], and

3) Dynamic analysis of the performance of current machine organizations by a combination of emulation and embedded data collection routines.

The design of the emulation laboratory and, particularly, the host machine have evolved from several seminars given at Johns Hopkins [1], [2], [3]. From these discussions we felt that there were several important features that our laboratory should included.

1) The host machine should be a microprogrammable processor with a dynamic control memory, that is, control memory should be writable by the micromachine in a time comparable to the read cycle. This allows the control memory to emulate the register and control resources of the target machine.

2) Structurally, the host machine should make available low level parallelism in the use of microresources, similar to that exhibited in target machines [5].

3) Laboratory units should be connected by a flexible bus structure.

4) Control of the laboratory should be exercised by a processor independent of the micromachine to allow convenient real time control and examination of emulated target machines.

2. System Structure

2.1 System Hardware Structure

Physical facilities in the emulation laboratory are organized around two bus systems; the bus system associated with the host machine and the bus system associated with the control processor. Functionally, laboratory facilites may be divided into four subsystems as shown in fig. 1:

1) Control,
2) Communication,
3) Host machine processor, and
4) Host machine external memory.

Logically, the center of the laboratory is the host machine processor which is a 32 bit, vertically organized, dynamically microprogrammable processor. Details of this facility are given later. Within the host machine is a writable control memory which represents the virtual control and data resources of the emulated target machine. Associated with the host machine is the 'host' bus by which the host machine communicates with the external environment. One part of this environment is the external memory subsystem which will ultimately contain core memory, disc memory and I/O devices and perhaps another host machine. During emulation the external memory subsystem will be used to emulate the primary and secondary memory facilities of the target machine. Also attached to the host bus is a block access controller (BAC) which, once initialized, is able to move a block of data between host bus units without direct host machine supervision. We are planning to provide a translation device to interface the 32 bit wide host bus to a PDP-11 compatable I/O device bus.

Control and observation of an emulation run are effected by the use of a Datapoint 2200 processor, a character oriented 'intelligent' terminal. Associated with this processor is a bus which we designate as the 'control' bus. Integral to the processor physically and attached to the control bus are a CRT display, a keyboard, and two cassette tape drives. A communication adaptor attached to the control bus is provided to generate serial ASCII code for the communication subsystem consisting of a model 38 TTY, a phone line modem and direct links to local computational facilities. Together with the 2200 processor, this subsystem provides a means of creating, editing and assembling emulation microprograms.

Between the host and control busses is an interface which allows the 2200 processor to communicate with and control the host machine and its associated resources. The host bus is structured so that any attached device may take control of the bus, thus allowing the interface to directly load and examine host processor register memory, control memory or external memory. A maintenance console consisting of basic control and indicator functions is also attached to the host bus. Both the interface and maintenance console also have several direct lines to the host machine processor to effect special manual control functions such as halt, step and run.

2.2 System Software Structure

In preparation for a typical emulation experiment, the microprogrammer will create and edit the microprogram using the Datapoint 2200 and its associated peripherals. Initially we will assemble these microprograms on the 2200 although ultimately we expect to use the host machine for microassembly. In addition to the microprogram the experimenter may also use the control processor to create, edit and store programs for target machines.

Microprograms and target machine programs are loaded into the host subsystem by the control processor via the interface and the host bus. In conventionally structured emulation experiments the host control memory will contain the emulation microprogram and the main memory, attached to the host bus, will contain the programs of the target machines. In some cases the experimenter may elect to store low usage microprograms in host main memory and use the block access controller to 'page' them into control memory.

During an emulation experiment the experimenter will control and observe the operation of the emulator from the control processor. For the purpose of gathering statistics related to the dynamic operation of a target machine, the microprogram will contain code to maintain event counters in control memory. These counters may easily be examined by the control processor during an emulation experiment. Since the control procesor has direct access via the interface to all memory and control locations in the host processor subsystem, we expect that observation will generally require little if any cooperation from the emulation microprogram.

3. Description of the Host Machine

3.1 General Parameters

The central physical facility of the emulation laboratory is the microprogrammable host machine. This machine contains eight general purpose registers and obtains microinstructions directly from a 4K word control memory. Memory cycle time is approximately 200 nsec, but the host machine cycle may be longer if it requires several control memory cycles. Data and microinstruction words within the host machine are all 32 bits in width. One of the general purpose registers is dedicated to maintaining the current state of the host machine (i.e. current microaddress, condition codes, etc.).

3.2 Structure

One of our design objectives was to incorporate low level parallelism into the structure of the host machine, preferably parallelism which reflected the parallelism inherent in the structure of target machines. For example, parallelism exists in the simultaneous operation of functional units, such as ALU's, and the fetching of the next instruction [5]. Also many non-functional operations, address calculation and so forth, take place concurrently

with functional operations. For purposes of analysis and description Flynn has divided target machine instructions into three classes [6]:

1) Functional (arithmetic, logical, Boolean)
2) Memory (including address calculation and modification)
3) Procedural (determination of next instruction address)

In the design of the host machine we have attempted to provide separate resources to perform each class of instruction at the microinstruction level and also at the level of the target machine.

Within the host machine exist three separate, yet interdependent, finite state machines, each receiving control input from the current microinstruction and each controlling a resource associated with one class of instructions (fig. 2). These machines are designated as:

1) T-machine (controls functional resources),
2) A-machine (controls memory resources), and
3) I-machine (controls fetching of the next microinstruction).

Microinstructions in the host machine are formated so that in general one half of the instruction (the TCF field) controls the T-machine and the other half (the ACF field) controls the A-machine. The I-machine may be controlled by either half of the microinstruction.

Both the T- and the A-machines manipulate data residing in the eight general purpose registers. The A-machine also moves data between control memory and the registers and initiates communications with external memory units on the host bus. I-machine operation controls the fetching of the next microinstruction from control memory. Host machine state information necessary to control the I-machine is contained in register 0 of the register file. Since this state register is directly accessable to the microprogrammer, flexible procedure oriented operations are possible.

3.3 Instruction Set Structure

In the following sections we will outline the important features of of the host machine instruction set (fig. 3). Microinstructions are 32 bits in length the high 14 bits, the TCF field, being dedicated to the control of the T-machine and the remaining 18 bits, the ACF field, dedicated the control of the A- and I-machines.

3.3.1 T-machine Instructions

T-machine instructions are designed to provide the basic functional operations that the microprogrammer needs to emulate the functional and control aspects of the target machine. Instructions for the T-machine may be divided into the following classes:

1) Logical,
2) Arithmetic,

3) Shift and Rotate,
4) Extended Arithmetic, and
5) Field Insert and Extract.

Instructions in the first four classes have a standard format which specifies opcode, subopcode, two register operands and indicates the possible use of immediate data. When immediate data is specified the 18 bit field usually used to control the A-machine is expanded into a 32 bit quantity for immediate data usage. The extended arithmetic instruction subopcodes are designed to give the microprogrammer powerful single cycle operations with which to build complex target machine instructions, such as multiply and divide, by repetition.

Field insert and extract instructions are full word instructions which the microprogrammer may use to isolate and move fields of data words residing in the registers. The insert instruction, for example, takes a word from one register, rotates it by a specified amount (0-31 bit positions) and places the result in a designated register under masking specified by the ACF field. We expect this instruction to be very useful in breaking down target machine instructions and in matching host machine resources to target machine requirements when the respective word lengths differ.

3.3.2 A-machine Instructions

A-machine instructions are used by the microprogrammer to access control memory, manipulate address pointers and communicate with external devices on the host bus system. A-machine instructions may be subdivided in to the following classes:

1) Move registers directly to and from control memory,
2) Load a register with immediate data,
3) Access memory resources indirectly,
4) Manipulate pointers, and
5) Maintain stacks in control memory.

Access to external memory is designed so that once the operation is initiated the micromachine instruction counter may continue to advance while awaiting the completion of the operation. This is an important source of parallelism in the emulation of instruction and data fetch in many target machines. A-machine stack operations allow the microprogrammer to access and maintain stacks in control memory and to check for stack boundary errors. Pointer manipulation instructions involve register incrementing, decrementing, addition and conditional branching on results.

3.3.3 I-machine Instructions

Fetching of the next microinstruction is controlled by the I-machine. Microinstructions are fetched sequentially from control store unless the I-machine is specifically directed to fetch from a different location. Since the machine state, which includes the microinstruction address, is contained in one of the general purpose

registers, the programmer may change the usual sequence by using the current microinstruction to to modify the state register.

Within the state register is an eight bit condition code field representing various aspects of the previous T-machine operation (Fig. 4). Instructions are provided to allow the microprogrammer to test these condition codes and control the operation of the A and I-machines. These instructions are classified as:

1) Conditional,
2) Branching, and
3) Looping.

A conditional instruction is one in which the TCF field of the micro instruction specifies the testing of the condition codes and controls the subsequent execution of the A-machine. If the indicated condition is found to hold then the instruction for the A-machine, as specified in the ACF field, is executed, otherwise it is skipped. Using this facility the microprogrammer is able to specify conditional jumps, stacking operations, memory accesses and so forth.

A branch instruction may be specified in the A-machine control field (ACF) and allows the programmer to test the condition codes and perform a short relative jump from the current location based on the results. This instruction is used to provide control of the I-machine concurrently with T-machine operation.

Pointer modification instructions, which control the A-machine, may also provide looping capability. The results of each pointer modification operation may be tested for one of the common arithmetic conditions (e.g. less than zero), and the results of the test may cause a short relative jump. This instruction allows the microprogrammmer to control repetitive operations such as normalize and multiply easily. In fact, the emulation of a target machine multiply instruction requires only one microinstruction since the extended arithmetic instruction 'multiply step' and the looping instruction may be combined.

4. Conclusion

We believe that the laboratory described above will provide flexible facilites for the efficient emulation of a range of target machine architectures. In addition to containing a powerful 32 bit host machine the laboratory is structured to allow the experimenter to control and observe the operation of the host machine from an independent processor.

The author acknowledges the contributions of his adviser, Dr. Mike Flynn, and his colleagues at Johns Hopkins University, Joe Davison, Lee Hoevel and Ken Kapulka (now at IBM, Gaithersburg, Maryland). In addition, the assistance of Dr. Bob McClure of Palyn Associates (San Jose, California) has been necessary in the practical realization of this project.

5. References

[1] L. Yelowitz et al; "A Simulator for a Microprogrammed Computer, Its Microassembler, and an Emulation"; Hopkins Computer Research Report #1; April 1971

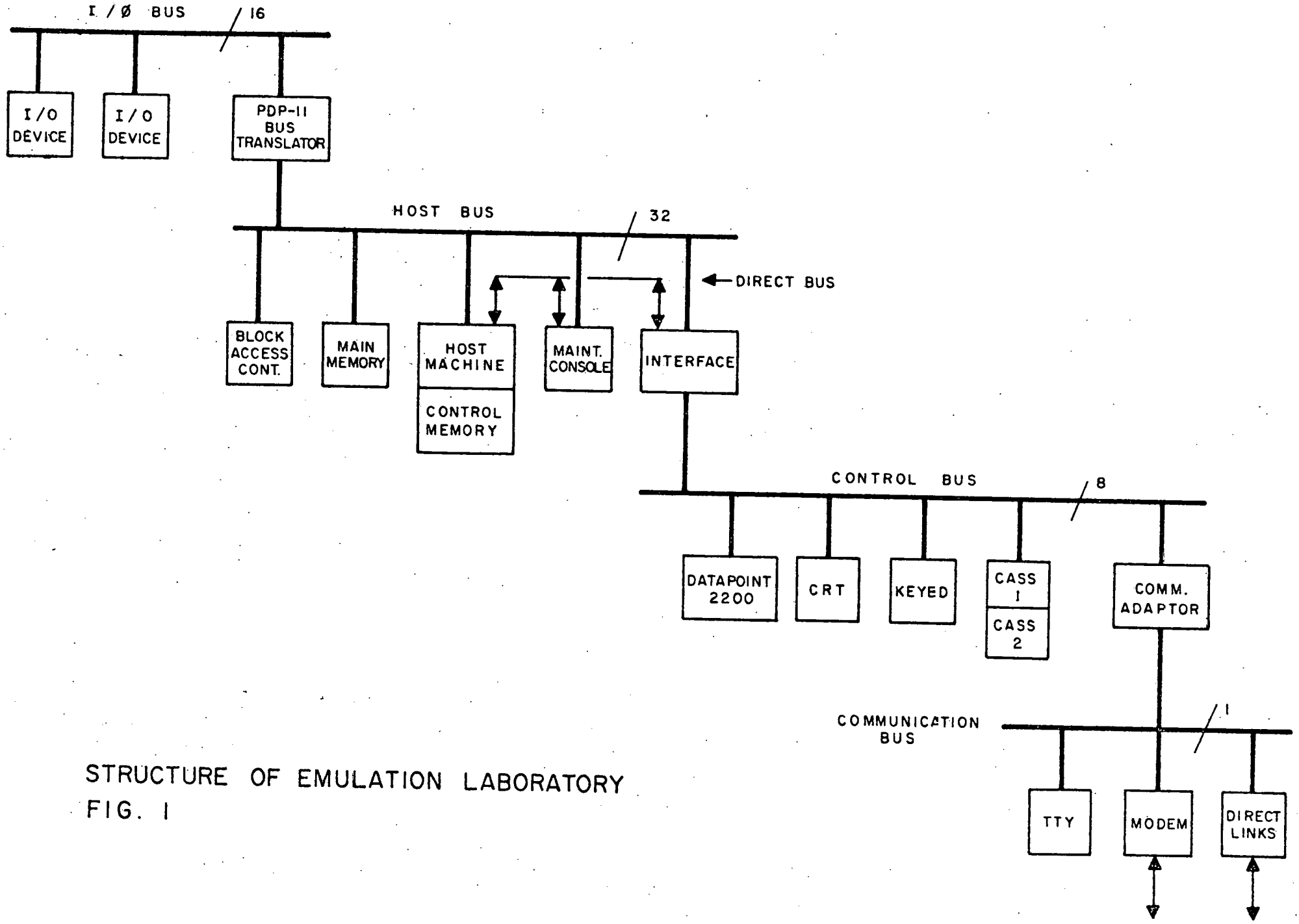
[2] J. Davison; "The JHU Universal Host Machine II - Part I: The Machine"; Hopkins Computer Research Report #31; March 1974

[3] C. Neuhauser; "An Emulation Oriented, Dynamic Microprogrammable Processor"; Hopkins Computer Research Report #28; November 1973

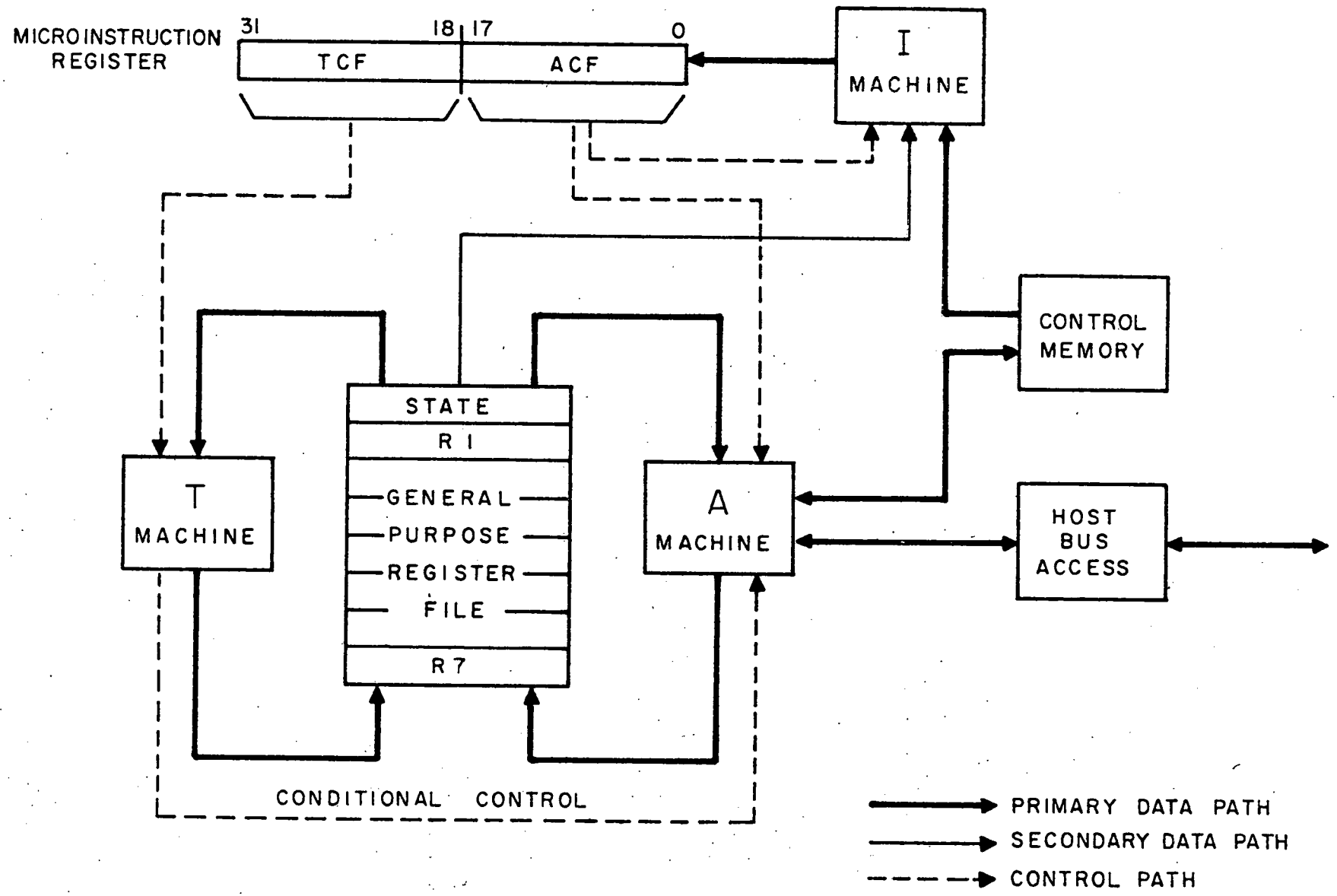
[4] L. Hoevel; "'Ideal' Directly Executed Languages: An Analytical Argument for Emulation"; To be published in IEEE Transactions on Computers; August 1974

[5] R. McClure; "Parallelism in Microprogrammed Controls"; The International Advanced Summer Institute on Microprogramming; 1972

[6] M. Flynn; "Trends and Problems in Computer Organizations"; To be published in Proceedings of IFIPS Congress; September 1974

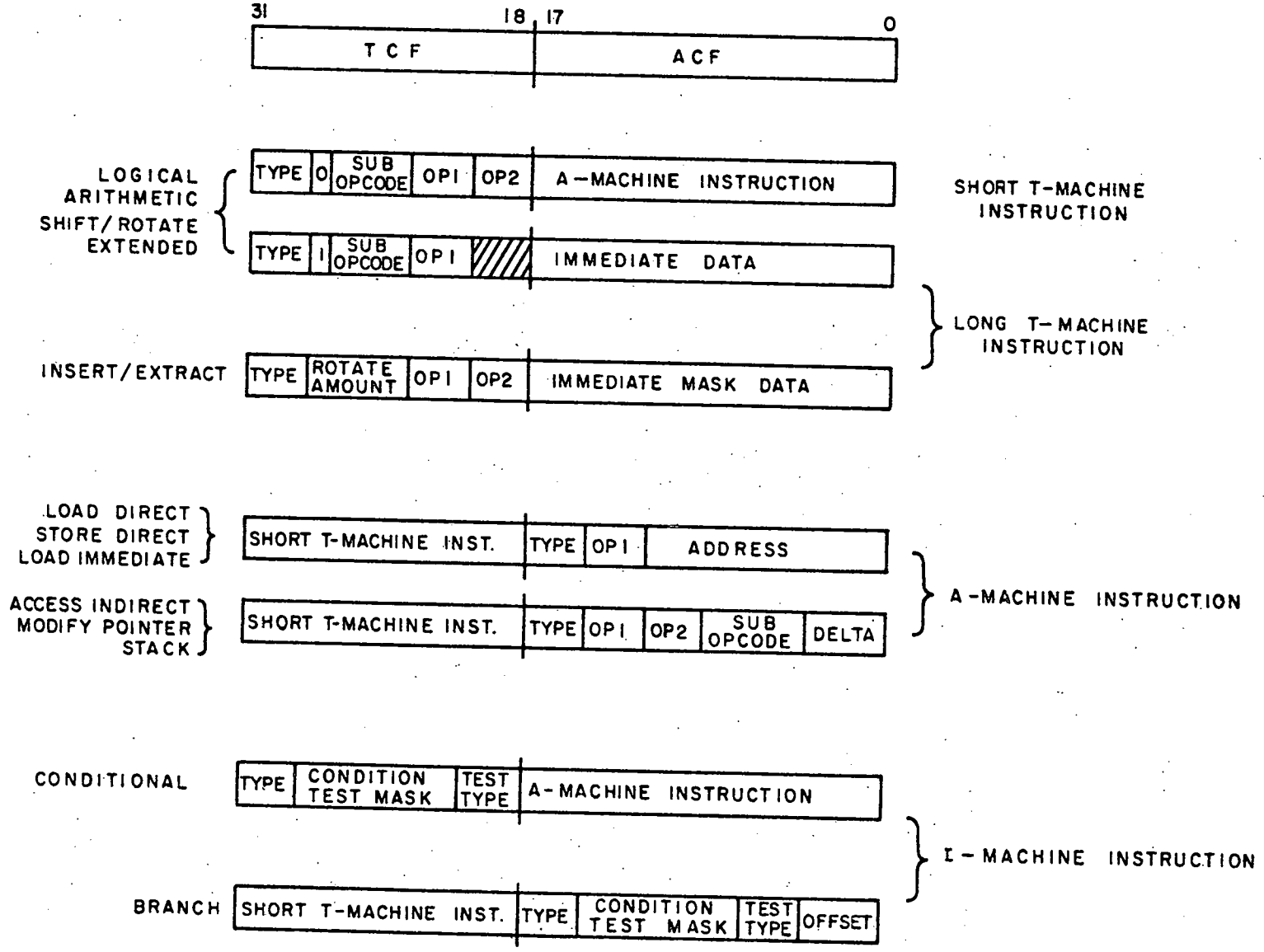


STRUCTURE OF EMULATION LABORATORY
FIG. 1



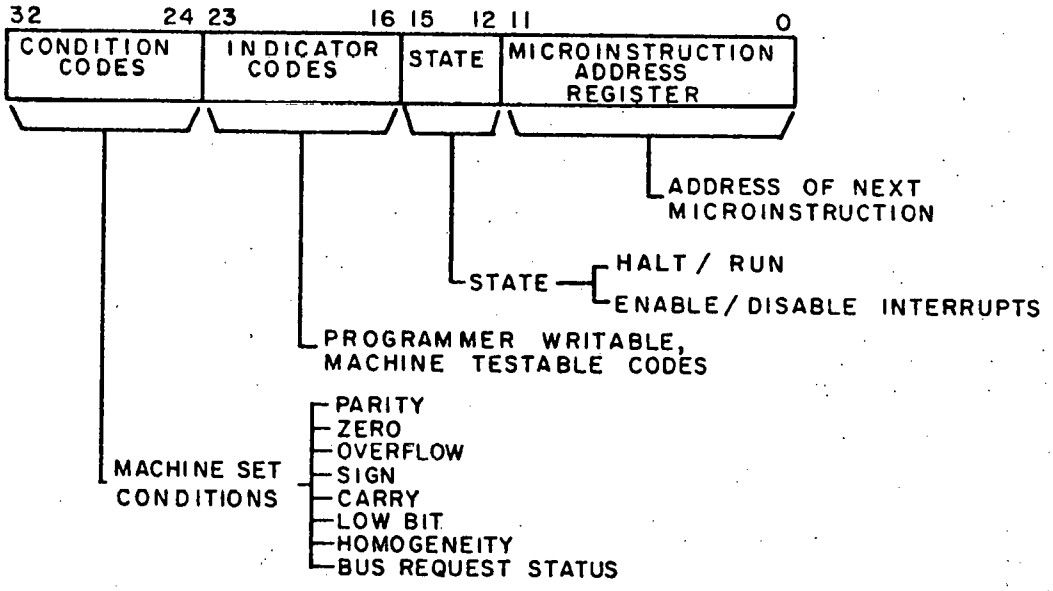
STRUCTURE OF HOST MACHINE

FIG. 2



STRUCTURE OF HOST MACHINE INSTRUCTION SET

FIG. 3



LAYOUT OF HOST MACHINE STATE REGISTER

FIG. 4

DESCRIPTION OF HOST MACHINE BUS STRUCTURE

1. Introduction

The host machine bus system is the means by which the host machine communicates with the external environment. We designate this bus system as the 'host' bus. Figure 1 shows the host bus and some associated units. Four sub-busses exist within the host bus system:

1) Data bus: consisting of lines to pass data, address and command signals,

2) Signal bus: consisting of various signal lines used to effect orderly inter-unit communications,

3) Clock bus: consisting of various timing signals, and

4) Direct bus: consisting of several lines needed to exercise direct control of the host machine.

2. Inter-unit Communication Philosophy

Logically, the host bus is quite similar to the PDP-11 Unibus, although somewhat more flexible. All units on the bus may have the capability of seizing the bus and transmitting bus commands. One may view the bus as being a resource for which the various bus units compete. Competition for the bus is monitored and resolved by a bus priority system based on the relative electrical position of competing units on the bus.

At the start of every bus cycle the unit with the highest priority is given control of the bus. This unit is allowed to execute one complete bus access cycle before the bus requests are rechecked. Communications on the bus are by means of a fully interlocked request/acknowledge discipline. The unit in control of the bus on a particular cycle is designated as the 'master' and the unit it communicates with is called the 'slave'. The objective of a bus operation is the transfer of data between the master and slave devices. Once this transfer has been completed the bus is released for reuse even though the communicating units may not have completed their internal cycles. This approach promotes efficient utilization of the bus.

3. Bus Semantics

Before giving the details of bus operation we will enumerate the individual lines in the sub-busses and give their semantics.

DATA BUS

Address (24) address of the bus resource being accessed by

the current master device.

Data	(32)	data to be communicated between master and slave unit.
Command	(1)	operation to be performed by the slave unit.

SIGNAL BUS

Request	(1)	a 'low' on this line indicates that one or more units are requesting use of the bus.
Available	(1)	This line is looped through each bus unit (see figure 2). When this line is 'low' at the input to a unit it indicates that the bus is available to that unit.
MSIG	(1)	used by the master unit to indicate that information on the data bus (address, command and possibly data) is valid.
SSIG	(1)	used by the slave unit to indicate that it is processing the master unit request. This signal is also used by the slave to indicate that data has been placed on the bus.
Reject	(1)	used by the slave unit to indicate that a request by the master unit cannot be honored during the current cycle.

CLOCK BUS

To be specified by the designer.

DIRECT BUS

This bus is composed of signal and indicator lines. The signal lines carry control information from the maintenance console to the host machine. These lines are pulsed lines and are not acknowledged. Indicator lines carry information on the state of the host machine to the maintenance console. Direct bus lines are also accessible from the control bus interface.

Signal Lines

Halt	(1)	causes the host machine to halt after the completion of the current cycle.
Run	(1)	causes the host machine to resume execution with the microinstruction designated in the MAR (microaddress register, R0<11:0>).
Step	(1)	If halted, the host machine will execute one microinstruction and halt again.

Clear	(1)	Reset the host machine and initialize bus units.

Indicator Lines

Run/Halt	(1)	current state of the host machine.

4. Explanation of Bus Unit Intercommunication

4.1 Bus Priority

Two signal bus lines are used to resolve multiple bus requests and to grant control of the bus to a single unit: Request and Available. The Request line is common to all bus units and is looped back on itself to provide the Available signal line (see figure 2). Available is passed through each bus unit in series. When a unit desires to obtain a bus cycle it lowers the common request line, causing the Available line to go low. Available is passed from unit to unit until a unit requesting service is encountered, in which case the signal in not propagated further and that unit takes control of the bus. At the end of a bus cycle the unit in control lowers its Request line and allows the Available signal to pass through to succeeding units.

By this scheme units are strictly ordered with respect to priority with the unit closest to the Request loop-around having highest priority. Each priority resolution cycle begins anew when the current unit raises its Request line, thus high priority devices are favored over low priority devices even though a given device of lower priority may have been requesting the bus for a longer period of time.

The suggested priority scheme for the initial system configuration is as follows:

Maintenance Console	(highest)
Control Bus Interface	!
Host Machine	!
Main Memory	!
Block Transfer Unit	!
PDP-11 Bus Translator	(lowest)

4.2 Bus Signaling

The signaling between master and slave bus units is shown in figures 3, 4 and 5. In these figures two logic signals are shown which are internal to bus units: 'Cycle' and 'Device Cycle'. Cycle indicates to the unit's associated bus interface that it desires access. Device Cycle indicates the internal data access or storage cycle that the unit performs.

4.2.1 Master to Slave Transmission

T1	Device raises Cycle requesting bus access.
T2	Request line is lowered requesting priority consideration.
T3	Available IN line becomes low indicating that the unit may now take control of the bus.
T4	Address, Command and Data are placed on the Data bus.
T5	After appropriate delay the MSIG line is raised indicating Address, Command and Data are valid at the slave unit.
T6	MSIG is received by the slave unit.
T7	After checking the address the addressed slave unit begins its Device Cycle and reads the Data lines.
T8	Slave unit raises SSIG indicating that data has been read.
T9	Master unit receives SSIG.
T10	Master unit lowers Address, Command, Data and MSIG lines. Master unit also raises the Request line allowing another unit to gain control of the bus.
T11	The slave unit sees the MSIG line go low.
T12	Slave unit lowers SSIG.

4.2.2 Slave to Master Transmission

Slave to master transmission is basically the same as the master to slave sequence described above. Upon receiving MSIG and checking the Address lines the selected slave unit will begin its read cycle. As soon as data is available the slave unit will place it on the data lines and, after a short delay, assert SSIG. Upon reception of the SSIG signal the master will strobe the data lines and terminate the cycle as in the sequence given above.

4.2.3 Reject Action by a Slave Unit

Since units may be accessed independently and their internal cycles may extend beyond the actual bus cycle, it is necessary to have a means by which a unit may reject a master's request for access if it cannot presently comply. This is the purpose of the Reject signal line. A sequence in which the slave unit rejects a master unit's request is illustrated in figure 5. The basic cycle is as described above except that when the slave unit receives MSIG and recognizes its

address it may, if busy, raise the Reject line instead of SSIG. Upon receiving Reject a master unit will lower the Address, Command, Data and MSIG lines and terminate its bus cycle. The master unit may, at its option, retry the request on a later bus cycle.

4.3 Considerations

1) The initiation of MSIG and SSIG should be delayed by an appropriate amount of time so that when each is received, the Address, Command and Data lines will have stabalized.

2) The bus signalling scheme described above will have to be examined carefully for possible race conditions. Two possibilities come to mind immediately:

    1) When the current bus master unit raises the Request line it also passes the Available line (low) to successive units. Thus if two units are requesting the bus, one of higher priority than the current master and the other of lower priority, it is possible that two units will be enabled simultaneously.

    2) The current master unit terminates its bus cycle with the reception of SSIG from the slave unit. Since the lowering of MSIG causes the slave to lower SSIG it is possible, due to propagation delays, that MSIG and SSIG could be high when the next bus cycle starts. If this proves to be a problem the raising of the Request line could by tied to the lowering of SSIG.

3) If a nonexistent unit is addressed or an existing unit fails it is possible that neither SSIG nor Reject signals may be returned resulting in a bus lock-up. This condition should be indicated on the Maintenance Console. It might be desirable for the Maintenance Console to clear the bus automatically by giving a Reject signal if no slave response is seen after an extended period of time.

5. Bus Address Structure

Addresses on the host bus system are 24 bits in width. For purposes of dividing this address spectrum among the bus devices we will designate the high eight bits of this address to be the unit number and the remaining 16 bits to be the internal unit address. For the initial host bus configuration the unit assignments should be as follows.

Unit 000	Main Memory (0 - 64K)
Unit 001	Main Memory (64K - 128K)
.	
.	
Unit 373	PDP-11 Translator
Unit 374	I/O translator control registers (if used)

Unit 375 Control Bus Interface
Unit 376 Maintenance Console
Unit 377 Host machine and control memory (see below)

To provide all host bus units with access to the host machine for examination and control purposes we have specified the host machine and its control memory to be unit 377. The 16 bit internal address field should be set up as follows.

Bits 15:12	Bits 11:0
0000	Control memory address
0001	General purpose registers (Bits 11:3 ignored)
0010	Microinstruction register (Bits 11:0 ignored)
0100	Interrupt vector register (Bits 11:0 ignored)

Devices on the host bus system interrupt the host machine by writing to the interrupt vector register. If the interrupts are enabled the low 12 bits written in this register will designate the even-odd control memory word pair to be used in interrupt processing. the contents of the current state register (REG[0]) will be stored in the even location and the new state register will be taken from the odd location. A Reject signal will be given if the interrupt register is addressed when host machine interrupts are disabled.

6. Block Access Controller (BAC)

To be defined later.

7. Maintenance Console Functional Description

Figure 6 illustrates the functional aspects of the maintenance console. The maintenance console has access to both the Data/Signal bus and the Direct bus. Direct bus access is used by the maintenance console to control the host machine and to obtain information on its internal state. The Data/Signal bus provides a means by which the maintenance console may examine and change the control memory of the host machine or any other Data bus address.

The Maintenance console may operate in two modes: 'real' and 'virtual'. In the 'real' mode the console is used to issue read and write commands to the host bus system. In the 'virtual' mode the console acts as a virtual I/O device, that is, the switch register and data display may be used by other host bus units for sense and display purposes. In the 'virtual' mode operation of any push button (PB) will cause an interrupt of the host machine.

7.1 Control Functions

Halt	(PB)	send halt signal to host machine.
Run	(PB)	send halt signal to host machine.
Step	(PB)	cause host machine to execute one

		microinstruction.
Master Clear	(PB)	reset host machine and initialize bus devices.
Bus Clear	(PB)	send a Reject signal to clear a bus lock-up.
Switch Reg.	(SW)	manual data input to maintenance console.
Load Address	(PB)	load address register from switch register.
Load Data	(PB)	load data out register from the switch register and display; write contents of data out register to the address given in the address register; increment the address register.
Examine Data	(PB)	read from the address given in the address register and place in data in register; display the contents of the data in register; increment the address register.
Display Data	(PB)	set display to show contents of data out register.
Display Add.	(PB)	set display to show contents of address register.
Mode	(SW)	indicates mode of console (real or virtual).
Power	(SW)	controls power to the host machine.

7.2 Indicator Functions

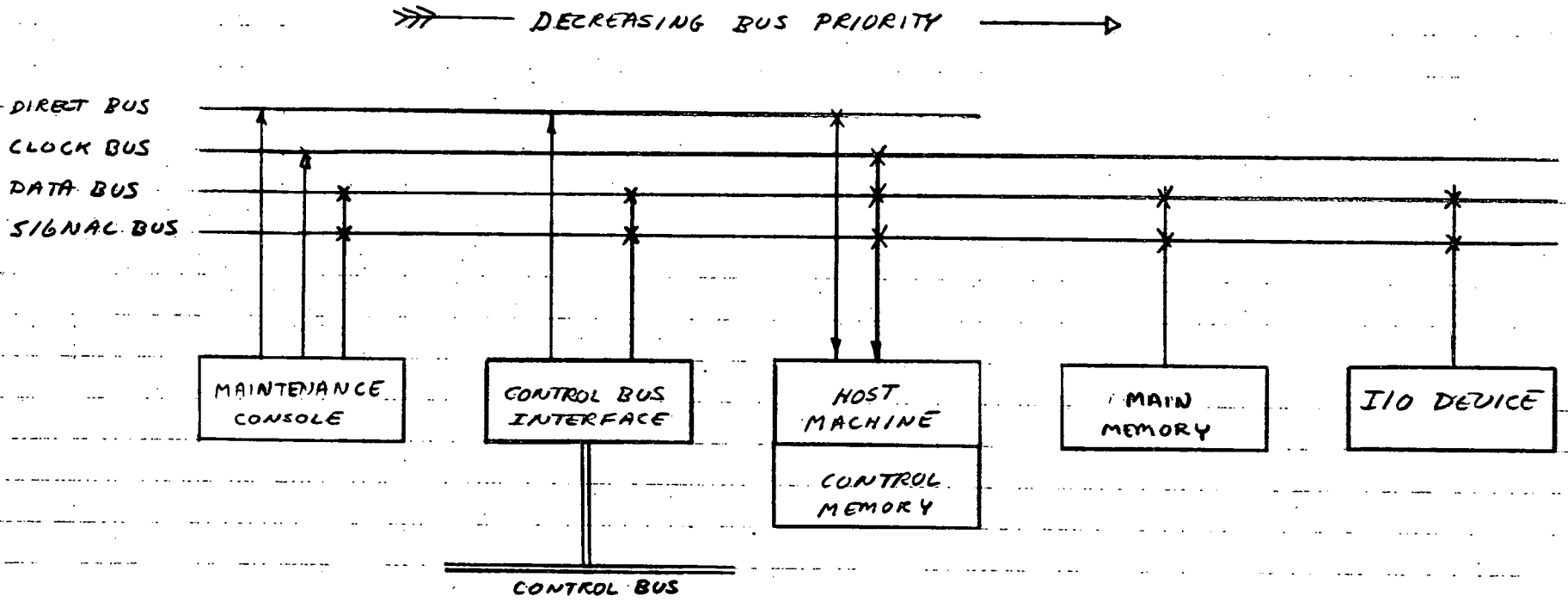
Run	(bit)	on if machine is in run state.
Halt	(bit)	on if machine is in halt state.
Bus Fault	(bit)	on if SSIG or Reject signal does not follow MSIG within 20 usec.
Power	(bit)	on when power is applied to the host machine.
Display	(bits)	Displays the contents of the data in and out registers or the contents of the address register depending upon Mode and the last console operation performed.
Address	(bit)	indicates that display is showing

		contents of address register.
Data	(bit)	indicates that display is showing contents of data in or data out register.

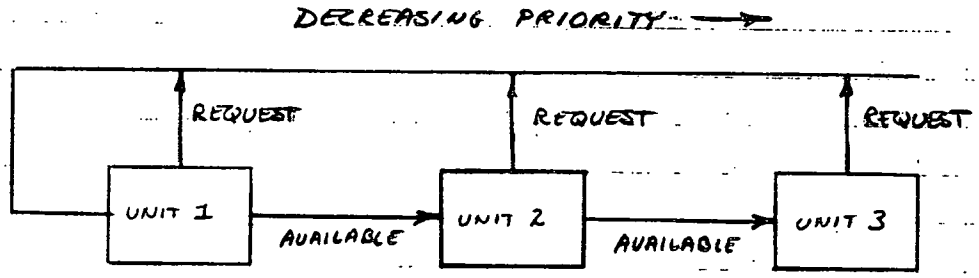
7.3 Considerations

1) For certain frequently used addresses on the Data/Signal bus, such as control memory and hsot machine registers, it might be desirable to have special maintenance console functions to set and send the high bits of the address. For example, the function Load Control Memory Address might be supplied which would load the high bits of the address with the unit number of the host machine and load the low 12 bits from the switch register. Other special address load functions might include Load State Register Address, Load Register Address and Load MIR Address.

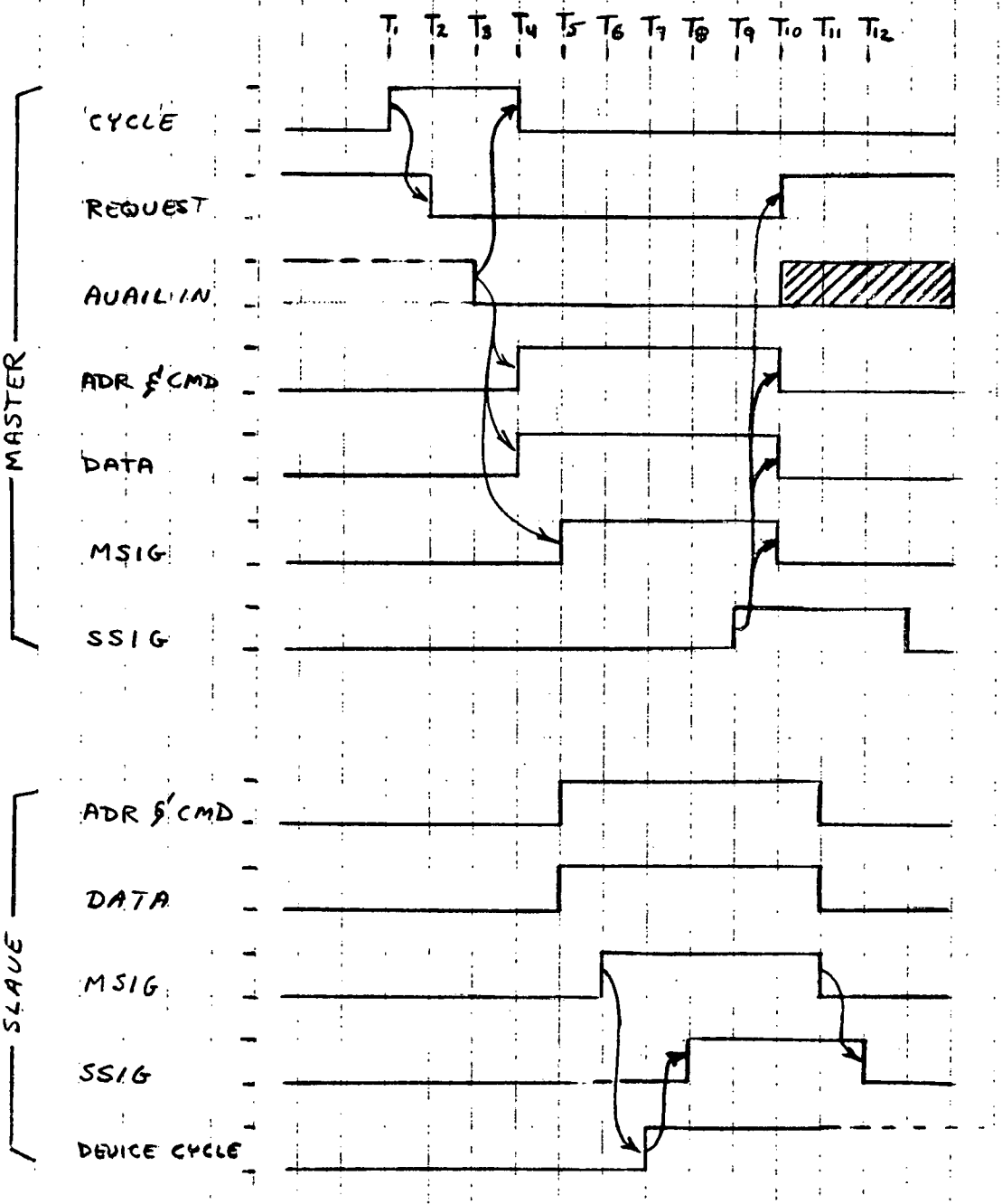
2) A halt instruction should cause the host machine to halt at the completion of the current microinstruction but allow any bus operation already initiated by the host machine to complete.



HOST BUS STRUCTURE FIG. 1



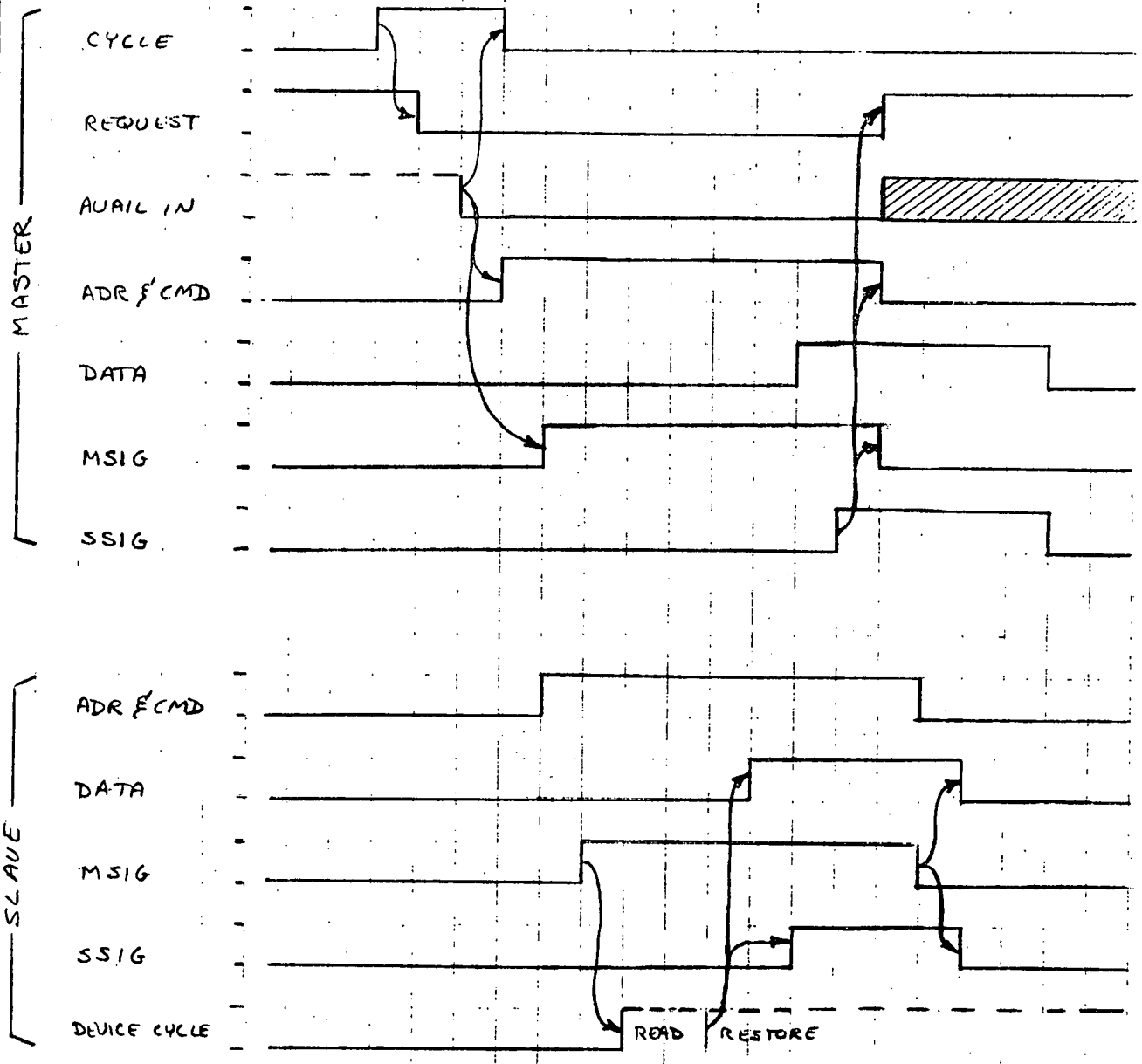
BUS PRIORITY SIGNALLING SCHEME
FIG. 2



SEQUENCING FOR MASTER TO SLAVE TRANSMISSION

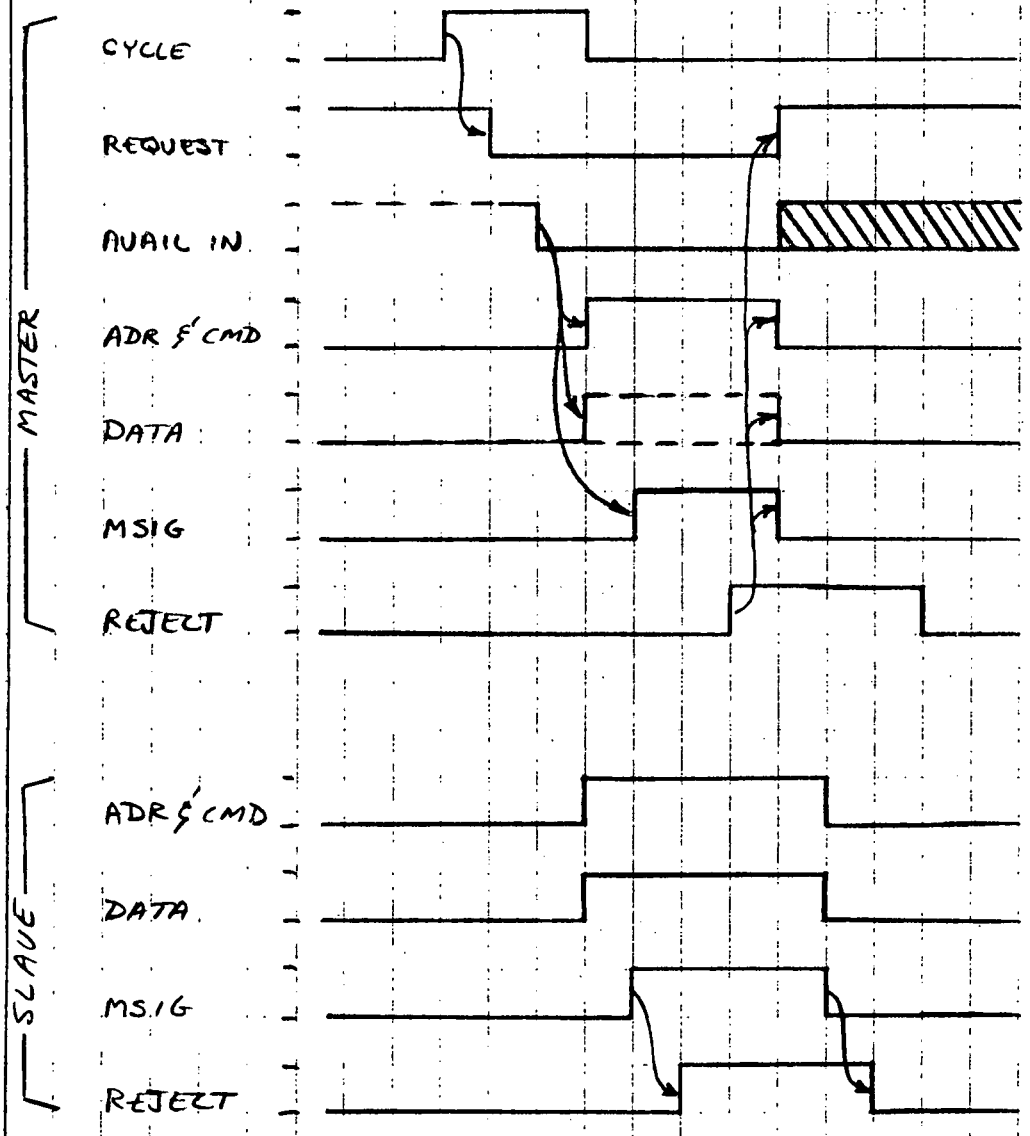
FIGURE 3

NOTE: TO PREVENT RACE CONDITIONS IT MAY BE NECESSARY TO HOLD REQUEST LOW UNTIL THE MASTER RECEIVES SSIG = LOW.



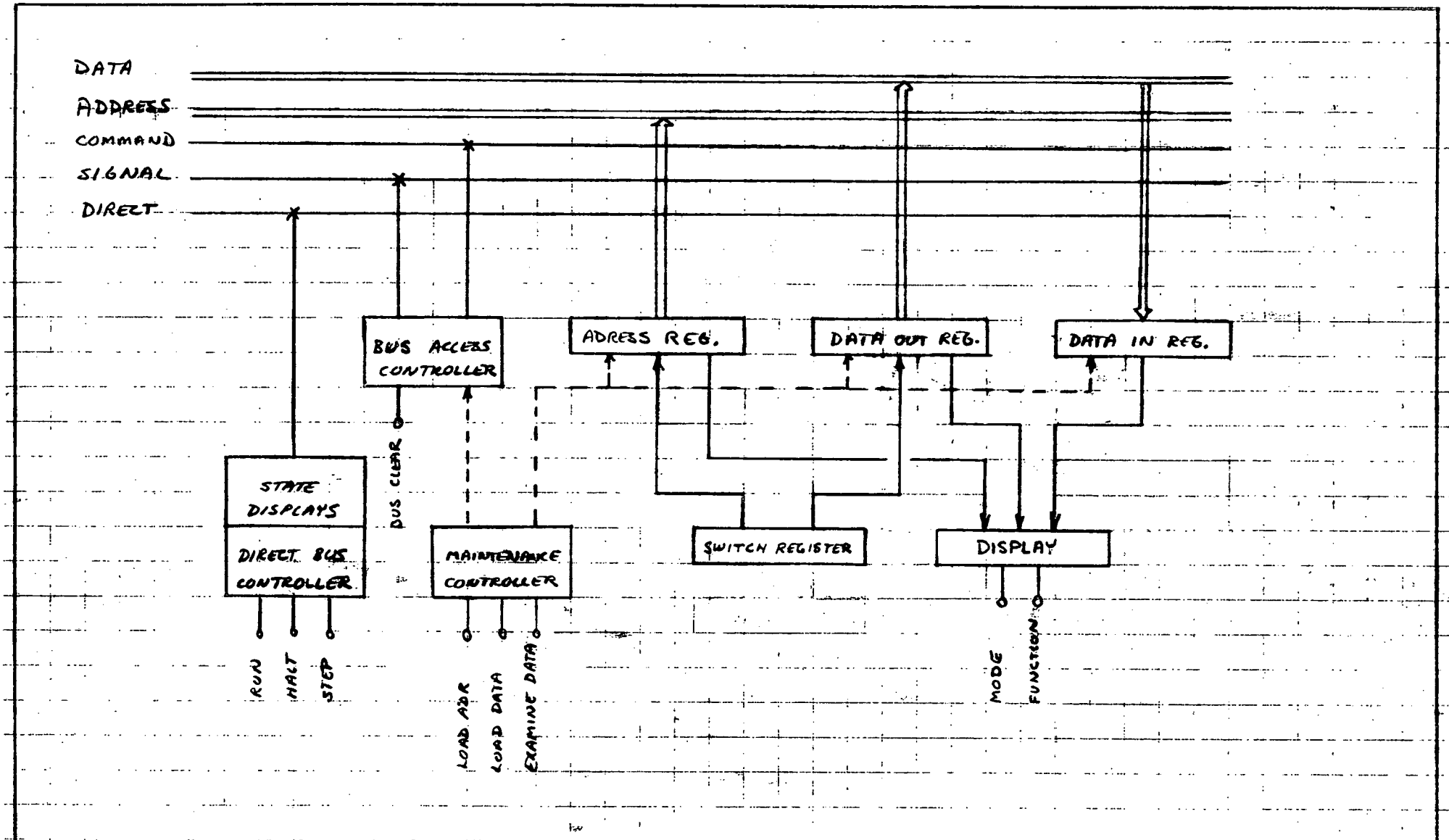
SEQUENCING SLAVE TO MASTER TRANSMISSION

FIGURE 4



SEQUENCING FOR REJECT ACTION BY SLAVE UNIT

FIGURE 5



FUNCTIONAL SCHEMATIC OF MAINTENANCE CONSOLE

FIGURE 6

Short Description of the Notation

In order to give a precise definition of the micromachine the ISP descriptive language has been used. This language is defined by Bell and Newell in 'Computer Structures: Readings and Examples' (pages 628-637) and in the 'PDP-11 Processor Handbook' (pages 207-219). In this section a brief description of the language is given to clarify differences between the standard ISP and the special form used here.

Statements in ISP define the transformation and movement of data between storage locations. These storage locations may be registers or memory cells and hold data or control information. Some registers, although defined, may not exist in the actual representation of the micromachine but are used in the definition only to preserve temporary results for future usage. Some examples of storage location definitions:

```
SC<5:0>                    ...a six bit register
REG[7:0]<31:0>             ...eight registers of 32 bits each
MEM[0:7777]<31:0>          ...4K memory cells of 32 bits each
```

The brackets '<' and '>' enclose the register width definition given in decimal notation. SC<5:0> defines a six bit register with the bit positions numbered 5 through 0. When registers are represented in drawings the 0 bit position will be on the right by convention. The square brackets, '[' and ']', enclose the definition of array size as in the second example above where REG[7:0]<31:0> denotes an array of eight registers numbered 7 through 0. Within the square brackets the numbers will be represented in octal.

STATEMENTS

There are three basic ISP statements:

1) Definition statements,
2) Action statements, and
3) Conditional statements.

A definition statement usually takes the form:

```
name := ( other ISP statemnets )
```

For example:

```
TCF<13:0> := MIR<31:18>

STK_error := (MEM[7776] <= REG[0];
              REG[0] <= MEM[REG[DR]<11:0>])
```

The first definition statement above simply defines a new name, TCF, to be a 14 bit field equivalent to the 14 bit field of register MIR bits 31 through 18. The second definition statement defines the name 'STK_error' to stand for the sequence of action statements shown on the right, namely saving and changing REG[0]. Wherever the name

'STK_error' is used in the ISP definition the sequence of statements to the right may be substituted. This substitution may be recursive as in the definition of 'Rotate'.

```
Rotate := (SC≠0 => (OP1 <= OP1<30:1>|OP1<31>;
                    SC <= SC+77 ; #
                    Rotate)
```

Action statements usually have the form:

storage location <= storage location

For example:

```
REG[AF] <= RH

REG[AF] <= REG[AF] + REG[BF]
```

Action statements may be grouped in a sequence with each statement separated by a semicolon. In such a sequence it is assumed that all statements are executed in parallel without regard to their ordering. Each statement uses the value of the storage locations as they existed before the sequence was executed. For example the statement sequence below swaps two registers.

```
REG[0] <= REG[1];
REG[1] <= REG[0];
```

In the event that it is necessary to indicate a definite ordering of two action statements (or groups of action statements) a hash mark, #, following a statement indicates that that statement and those above it must be performed before any following action statements are performed. For example,

```
OP1 <= REG[BF];
SC  <= POS; #
Rotate;
OP2 <= EXPAND; #
.....
.....
```

In the above sequence the first two statements are performed together or in any order, and then the next two statements are performed. This ordering is necessary in this case since 'Rotate' is defined to depend upon the value of OP1 and SC.

Conditional statements have the following form:

condition => (statement or statement sequence)

The condition is a Boolean type statement (e.g. TOP=3) which if evaluated as true will imply than the micromachine performs the action specifed. If the condition evaluates to false then no action is taken. Examples of conditional statements are:

```
OP<2>=1 => (SC <= 1);

OP<1:0>=3 => (CIRH <= REG[AF]+¬OP2+C);
```

In the first example the storage location SC will be assigned a value 1 if bit 2 of the storage location OP is equal to 1. In the second example if the value of the storage location OP<1:0> is 3 then the action statement given will be performed.

OPERATORS

Several operators are defined which transform the data stored in storage locations. These operators are listed below.

∨	bit by bit logical OR
∧	bit by bit logical AND
¬	bit by bit logical NOT
⊕	bit by bit logical EXCLUSIVE OR
<=	move one storage location to another
I	concatenation of two storage locations
+	two's complement addition
=	relational operation 'equals'
≠	relational operation 'not equal'
⊲	relational operation 'less than'
⊳	relational operation 'greater than'
/	this operator is used in conjunction with another bit by bit operator, such as +. The form is operator/ as in the example below:

⊕/CCODE

The / indicates that all bits of the storage location are to be combined using the designated operation, in this case exclusive or. The result is the same as if we had written:

CCODE<1> ⊕ CCODE<2> ⊕ ... ⊕ CCODE<n>

```
/FIELD DEFINITIONS

/Definition of Microinstruction register layout. (see figure 1)

   MIR<31:0>                              /Microinstruction Register

   TCF<13:0> :=  MIR<31:18>               /T-machine Control Field

       TOP<2:0> := MIR<31:29>             /T-machine operation
       I<0>     := MIR<28>                /Defines input to ALU
       OP<3:0>  := MIR<27:24>             /Sub-op-code for T-machine
       POS<4:0> := MIR<28:24>             /Left rotated position of mask
       BF<2:0>  := MIR<23:21>             /B-register designator (source)
       AF<2:0>  := MIR<20:18>             /A-register designator (source & sink)

       CMASK<7:0> := MIR<28:21>           /Test mask for conditional instruction
       COP<2:0>   := MIR<20:18>           /Defines type of conditional test

       T_SPARE<10:0> := MIR<28:18>        /Spare field

   ACF<17:0> := MIR<17:0>                 /A-machine control field

       AOP<2:0>   := MIR<17:15>           /A-machine operation code
       CF<2:0>    := MIR<14:12>           /C-register designator (source or sink)
       ADR<11:0>  := MIR<11:0>            /Micromemory address   (source or sink)

       BMASK<7:0> := MIR<14:7>            /Test mask for branch operation

       DF<2:0>    := MIR<11:8>            /D-register designator (source or cont)
       EF<1:0>    := MIR<7:6>             /A-machine sub-sub-op-code
       XOP<2:0>   := MIR<6:4>             /A-machine sub-op-code
       VAL<3:0>   := MIR<3:0>             /A-machine immediate value

       EXP<1:0> := MIR<17:16>             /Defines expansion of IF field
       IF<15:0> := MIR<15:0>              /Immediate data field

       ASPARE<14:0> := MIR<14:0>          /A-machine spare field

/Definition of memory resources.

   REG[0:7]<31:0>                         /Register storage     (8, 32 bit words)
   MEM[0:7777]<31:0>                      /Micromemory storage  (4K, 32 bit words)
   EXT[0:77777777]<31:0>                  /External storage     (4M, 32 bit words)

       MEM[7776]                          /Storage for REG[0] on STACK overflow
       MEM[7777]                          /Unassigned

/Definition of REG[0], machine state register. (see figure 1 and 18)

   CCODE<7:0> := REG[0]<31:24>            /Condition codes

       CC<1:0> := REG[0]<31:30>           /Arithmetic condition codes
       C<0>    := REG[0]<29>              /Carry
       H<0>    := REG[0]<28>              /High bit of result
       L<0>    := REG[0]<27>              /Low bit of result
       D<0>    := REG[0]<26>              /All bits of result are the same
       P<0>    := REG[0]<25>              /Parity of result
       B<0>    := REG[0]<24>              /External memory busy
```

```
    ICODE<7:0> := REG[0]<23:16>          /Indicator codes

    STATE<3:0> := REG[0]<15:12>          /Machine state

        R<0>   := REG[0]<15>             /Run-Halt bit
        DI<0>  := REG[0]<14>             /Disable-Enable interrupt bit

    MAR<11:0>  := REG[0]<11:0>           /Microaddress register

/Definition of miscellaneous machine registers.

    OP1<31:0>                            /Temporary operand register
    OP2<31:0>                            /Temporary operand register
    RES<63:0>                            /Temporary result register

        RH<31:0> := RES<63:32>           /High result register
        RL<31:0> := RES<31:0>            /Low result register

    SC<5:0>                              /Shift count register
    IVR<11:0>                            /Interrupt vector register
    Interrupt<0>                         /Interrupt pending indicator
    SKIP<0>                              /AOP_execute skip indicator
    OVF<0>                               /Multiply step 'overflow'
    T<0>                                 /Temporary storage for sign
    K<0>                                 /Character operation 'carry'
    CI<0>                                /Character operation 'carry in'
    CO<0>                                /Character operation 'carry out'
```

```
/  DESCRIPTION OF I-MACHINE OPERATION

I_execute := (
   R=0 => ();
   R=1 => (Interrupt∧¬DI => (MEM[IVR<11:0>] <= REG[0];          /Interrupt action
                             REG[0] <= MEM[IVR<11:0>⊕1]);
          (¬Interrupt∧DI => (MIR <= MEM[MAR];                  /Fetch inst.
                             MAR <= MAR+1; #                   /Increment MIR
                             TOP_execute; #
                             (MIR<31>=0∧I=0)          => AOP_execute;
                             (MIR<31:29>=6∧SKIP=0)    => AOP_execute;
                             MIR<31:29>=7             => AOP_execute;
                             SKIP <= 0; #
   I_execute;
            )
```

```
/  DESCRIPTION OF T-MACHINE OPERATION

TOP_execute := (                                    /See figure 2
    TOP=0 => (LOGICAL);
    TOP=1 => (ARITHMETIC);
    TOP=2 => (SHIFT);
    TOP=3 => (EXTENDED);
    TOP=4 => (INSERT);
    TOP=5 => (EXTRACT);
    TOP=6 => (CONDITIONAL);
    TOP=7 => (TOP_SPARE)
                )


EXPAND<31:0> := (                                   /Defines expansion of ACF field
    EXP=0 => (000000₈!IF);                          /for use as immediate data
    EXP=1 => (177777₈!IF);
    EXP=2 => (IF!000000₈);
    EXP=3 => (IF!177777₈);
                )

-------------------------------------------------------------------------------

LOGICAL := (                                                /See figure 3
     I=0 => (OP1 <= REG[BF]);
     I=1 => (OP1 <= EXPAND); #

     OP=0   => (RH <= REG[AF]);                             /NOP
     OP=1   => (RH <= REG[AF]∧OP1);                         /AND
     OP=2   => (RH <= REG[AF]∧¬OP1);
     OP=3   => (RH <= 00000000000);                         /CLEAR
     OP=4   => (RH <= REG[AF]∨OP1);                         /OR
     OP=5   => (RH <= OP1);                                 /XFR
     OP=6   => (RH <= REG[AF]⊕OP1);                         /XOR
     OP=7   => (RH <= ¬REG[AF]∧OP1);
     OP=8   => (RH <= REG[AF]∨¬OP1);
     OP=9   => (RH <= ¬(REG[AF]⊕OP1));                      /XNOR
     OP=10  => (RH <= ¬OP1);                                /CXFR
     OP=11  => (RH <= ¬(REG[AF]∨OP1));                      /NOR
     OP=12  => (RH <= 37777777777₈);
     OP=13  => (RH <= ¬REG[AF]∨OP1);
     OP=14  => (RH <= ¬(REG[AF]∧OP1));                      /NAND
     OP=15  => (RH <= ¬REG[AF]);                            /COM


     RH=0                       =>(CC <= 0);        /Result equals zero
     RH<31>=1                   =>(CC <= 1);        /Result less than zero
     (RH≠0)∧(RH<31>=0)          =>(CC <= 2);        /Result greater than zero
     C <= 0;                                        /Carry set to zero
     H <= RH<31>;                                   /High bit of result
     L <= RH<0>;                                    /Low bit of result
     D <= (∧/RH<31:0>)∨(∧/¬RH<31:0>);               /Bits 31 to 0 are the same
     P <= ⊕/RH<31:0>                                /Parity of result

     REG[AF] <= RH                                  /Store result in register
            )
```

```
ARITHMETIC := (                                          /See figure 4
    (OP<2>=0)∧(I=0) => (OP2 <= REG[BF]);
    (OP<2>=0)∧(I=1) => (OP2 <= EXPAND);
     OP<2>=1        => (OP2 <= 000000000₈|BF); #         /For INC and DEC BF field
                                                         /is used as OP2

     OP<1:0>=0 => (C|RH <= REG[AF]+OP2);                 /C is carry bit
     OP<1:0>=1 => (C|RH <= REG[AF]+OP2+C);
     OP<1:0>=2 => (C|RH <= REG[AF]+¬OP2+1);
     OP<1:0>=3 => (C|RH <= REG[AF]+¬OP2+C); #


   RH=0                          => (CC <= 0);          /Result equals zero
   RH<31>=1                      => (CC <= 1);          /Result less than zero
   (RH≠0)∧(RH<31>=0)             => (CC <= 2);          /Result greater than zero
   (REG[AF]<31>=OP2<31>)∧
        (OP2<31>≠RH<31>)         => (CC <= 3);          /Overflow

   C <= C;                                               /Carry (set on ALU op)
   H <= RH<31>;                                          /High bit of result
   L <= RH<0>;                                           /Low bit of result
   D <= (∧/RH<31:0>)∨(∧/¬RH<31:0>);                      /Bits 31-0 are the same
   P <= ⊕/RH<31:0>;                                      /Parity of result

   OP<3>=0 => (REG[AF] <= RH);                           /Conditional store of
                                                         /arithmetic result
             )
```

```
SHIFT := (                                               /See figure 5
    (OP<2>=0)∧(I=0) => (SC <= REG[BF]<5:0>);
    (OP<2>=0)∧(I=1) => (SC <= EXPAND<5:0>);
    (OP<2>=1)       => (SC <= 0000000000|BF);

     RH <= REG[AF];                          /Load high order part
     OP<3>=1 => (RL <= REG[AF⊕1]); #         /Load low order part on double shift
     Shift_step; #                           /Shift by amount in SC

     REG[AF] <= RH;                          /Store high order part
     OP<3>=1 => (REG[AF⊕1] <= RL)            /Store low order part on double shift
         )

   Shift_step := (
       SC≠0 => (OP<3>=0 => (Single_shift_step);
                OP<3>=1 => (Double_shift_step);
                SC <= SC+77₈; #
                Shift_step
                 )

       Single_shift_step := (
           OP<1:0>=0 => (RH <= RH<30:0>|0);          /left single logical
           OP<1:0>=1 => (RH <= RH<30:0>|RH<31>);     /left single rotate
           OP<1:0>=2 => (RH <= 0|RH<31:1>);          /right single logical
           OP<1:0>=3 => (RH <= RH<31>|RH<31:1>)      /right single arithmetic
                              )
```

```
        Double_shift_step := (
            OP<1:0>=0 => (RES <= RES<62:0>|0);        /left double shift
            OP<1:0>=1 => (RES <= RES<62:0>|RES<63>); /left double rotate
            OP<1:0>=2 => (RES <= 0|RES<63:1>);        /right double logical
            OP<1:0>=3 => (RES <= RES<63>|RES<63:1>)  /right double arithmetic
                                 )
```

```
EXTENDED := (                                          /See figure 6
   I=0 => (OP2 <= REG[BF]);
   I=1 => (OP2 <= EXPAND); #

        OP=0 => (Multiply_step);
        OP=1 => (Divide_step);
        OP=2 => (Form_excess_six);
        OP=3 => (Delay_cycle);
        OP=4 => (Decimal_add);
        OP=5 => (Decimal _subtract);
        OP=6 => (Binary_to_decimal_step);
        OP=7 => (Decimal_to binary_step);
        OP>7 => (Delay_cycle);
             )

Multiply_step := (
   RES <= REG[AF]|REG[AF⊕1]; #                         /Load operands
   RES <= RES<63>|RES<63:1>; #                         /Right shift arithmetic
   RES <= RES<63>⊕OVF;                                 /Overflow from last mult step
   T   <= RES<63>;                                     /Save sign of result
   REG[AF⊕1]<0>=1 => (RH <= RH+OP2); #                 /Conditional add of m'cand
   OVF <= (T=OP2<31>)∧(OP2<31>≠RES<63>); #             /Set overflow
   REG[AF]   <= RES<63:32>;
   REG[AF⊕1] <= RES<31:0>;                             /Store results in registers
                )

Divide_step := (
   RES <= REG[AF]|REG[AF⊕1]; #                         /Load operands
   RH  <= RH+¬OP2+1; #                                 /Subtract divisor
   T   <= REG[AF]<31>≠RH<31>; #                        /T=1 if sign changed
   RES <= RES<62:1>|T; #                               /Shift in 1 if sign changed
   REG[AF]   <= RES<63:32>;
   REG[AF⊕1] <= RES<31:0>;                             /Store results in registers
               )

Form_excess_six := (

   XS6<x:y> := (OP2<x:y>⊗9 => (REG[AF]<x:y> <= 6)

   XS6<3:0>;       /Form excess six for first character
   XS6<7:4>;       /Form excess six for second character
      .
      .
   XS6<31:28>;     /Form excess six for eighth character
                    )

Decimal_add := (

   ADD<x:y> := (                                       /Character add
       K|RH<x:y> <= REG[AF]<x:y>+OP2<x:y>+K; #         /K is carry
```

```
        K|RH<x:y>⊗9 => (K <= 1;
                        RH<x:y> <= RH<x:y>+6)
                )

    K <= 0; #           /Initial carry in is zero
    ADD<3:0>; #         /First character addition
    ADD<7:4>; #         /Second character addition
        .
        .
    ADD<31:28>; #       /Eighth character addition
    REG[AF] <= RH       /Store result in register
                )

Decimal_subtract := (

    COM<x:y> := (OP2<x:y> <= ¬OP2<x:y>+10₈)      /Ten's complement of character

    K <= 1; #                           /Carry in is set to 1
    COM<3:0>; #     ADD<3:0>; #         /First character subtraction
    COM<7:0>; #     ADD<7:4>; #         /Second character subtraction
        .
        .
    COM<31:28>; #   ADD<31:28>; #       /Eighth character subtraction
    REG[AF] <= RH                       /Store result in register
                    )

Binary_to_decimal_step := (

    DM2<x:y> := (                                   /Character multiply by two
        RES<x:y> <= RES<x-1:y>|K; #                 /Shift and enter carry in
        RES<x:y>⊗9 => (K <= 1;                      /Set carry out
                       RES<x:y> <= RES<x:y>+6)      /Correct result
                )

    RES <= REG[AF]|REG[AF⊕1];           /Load operands
    K <= REG[BF]<31>;                   /Carry in is high bit of REG[BF]
    REG[BF] <= REG[BF]<30:0>|0; #       /Left shift REG[BF]
    DM2<3:0>; #                         /First character multiply
    DM2<7:4>; #                         /Second character multiply
        .
        .
    DM2<63:60>; #                       /Sixteenth character multiply
    REG[AF]   <= RES<63:32>;
    REG[AF⊕1] <= RES<31:0>              /Store results in registers
                            )

Decimal_to_binary_step := (

    DD2<x:y> := (                                   /Character divide by two
        CI <= CO;                                   /Get previous carry out
        CO <= RES<y>;                               /Low bit is new carry out
        RES<x:y> <= 0|RES<x:y+1>; #                 /Right shift by one
        CI=1 => (RES<x:y> <= RES<x:y>+5)            /Correction for carry in
                )

    RES <= REG[AF]|REG[AF⊕1];           /Load operands
    CI <= 0; #                          /Carry in is initially zero
    DD2<63:60>; #                       /Sixteenth character divide
    DD2<59:56>; #                       /Fifteenth character divide
```

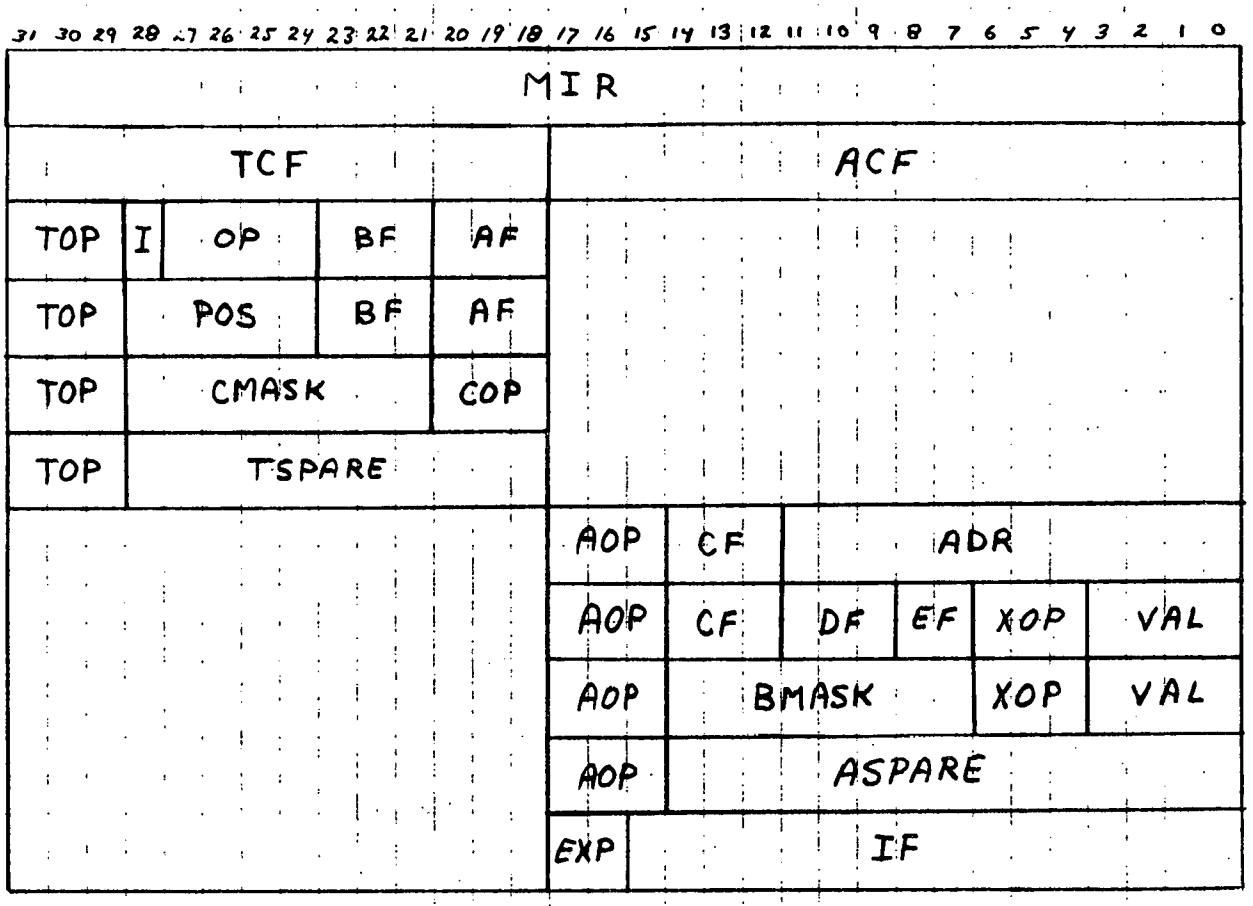
```
        .
        .
    DD2<3:0>; #                          /First character divide
    REG[BF]    <= CO|REG[BF]<31:1>;      /Enter last remainder into REG[BF]
    REG[AF]    <= RES<63:32>;
    REG[AF⊕1] <= RES<31:0>               /Store results in registers
                          )

Delay_cycle := ()                        /NOP for T-machine

-------------------------------------------------------------------------

INSERT := (                              /See figure 7
   OP1 <= REG[BF];
   OP2 <= Expand;
   SC  <= POS; #
   Rotate; #
   REG[AF] <= (OP1∧OP2)∨(REG[AF]∧¬OP2);
         )

   Rotate := (                                          /Left single rotate
       SC≠0 => (OP1<31:0> <= OP1<30:1>|OP1<31>);
                SC <= SC+77₈; #
                Rotate)
             )

-------------------------------------------------------------------------

EXTRACT := (                                            /See figure 8
   OP1 <= REG[BF];
   OP2 <= Expand;
   SC <= POS; #
   Rotate; #
   REG[AF] <= OP1∧OP2;
         )

-------------------------------------------------------------------------

CONDITIONAL := (                                        /See figure 16
   COP<2>⊕∨/(CMASK∧(COP<1>⊕((¬COP<0>∧CCODE)∨(COP<0>∧ICODE))))
       => (SKIP <= 1)
             )

-------------------------------------------------------------------------

TOP_SPARE := ()
```

```
AOP_execute := (                                        /See figure 9
   AOP=0 => (LOAD_DIRECT);
   AOP=1 => (STORE_DIRECT);
   AOP=2 => (LOAD_IMMEDIATE);
   AOP=3 => (BRANCH);
   AOP=4 => (INDIRECT_ACCESS);
   AOP=5 => (POINTER_MODIFICATION);
   AOP=6 => (STACK);
   AOP=7 => (AOP_SPARE)
            )

   DELTA := (VAL<3>=0 => (0000000000₈!VAL);
             VAL<3>=1 => (1777777777₈!VAL))

-------------------------------------------------------------------------

LOAD_DIRECT := (                                        /See figure 10
   REG[CF] <= MEM[ADR]
              )

-------------------------------------------------------------------------

STORE_DIRECT := (                                       /See figure 11
   MEM[ADR] <= REG[CF]
               )

-------------------------------------------------------------------------

LOAD_IMMEDIATE := (                                     /See figure 12
   REG[CF]<11:0> <= ADR                                 /Only low 12 bits are loaded
                  )

-------------------------------------------------------------------------

BRANCH := (                                             /See figure 17
   XOP<2>⊕V/(BMASK∧(XOP<1>⊕((¬XOP<0>∧CCODE)∨(XOP<0>∧ICODE)))
            => (MAR <= MAR+DELTA))
         )


-------------------------------------------------------------------------

INDIRECT_ACCESS := (                                    /See figure 13
   (XOP=0 => (REG[CF]       => REG[DF]);
    XOP=1 => (REG[CF]       => MEM[REG[DF]]);
    XOP=2 => (REG[CF]       => EXT[REG[DF]]);
    XOP=3 => (REG[CF]       <= MEM[REG[DF]]);
    XOP=4 => (MEM[REG[CF]]  <= MEM[REG[DF]]);
    XOP=5 => (MEM[REG[CF]]  => EXT[REG[DF]]);
    XOP=6 => (REG[CF]       <= EXT[REG[DF]]);
    XOP=7 => (MEM[REG[CF]]  <= EXT[REG[DF]]); #

    EF<0>=1  => (REG[DF] <= REG[DF]+DELTA); #
    EF<1>=1  => (REG[CF] <= REG[CF]+DELTA);
                )
```

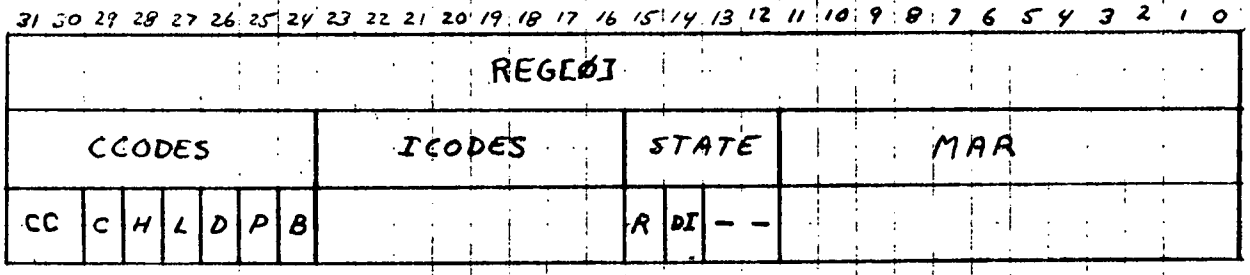
```
POINTER_MODIFICATION := (                                   /See figure 14

   (EF=0 => (REG[CF] <= REG[CF] + 00000000001DF);                    /INC
    EF=1 => (REG[CF] <= REG[CF]+¬(00000000001DF)+1);                 /DEC
    EF=2 => (REG[CF] <= REG[CF]+REG[DF]);                            /ADD
    EF=3 => (REG[CF] <= REG[CF]+¬REG[DF]+1)); #                      /SUB

   ((XOP<0>∧REG[CF]<31>)∨                          /Test for less than zero
    (XOP<1>∧(REG[CF]=0))∨                          /Test for equal to zero
    (XOP<2>∧(REG[CF]<31>=0∧REG[CF]≠0)))            /Test for greater than zero
        => (MAR <= MAR+DELTA);                     /If true then loop
                        )
```

```
STACK := (                                                  /See figure 15

   Limit := (REG[DF]<7:0>=REG[DF]<31:24>∨          /Limit=1 if pointer is at
             REG[DF]<7:0>=REG[DF]<23:16>)          /high or low stack limit

   STK_error := (MEM[7776] <= REG[0];              /Save old state register
                 REG[0] <= MEM[REG[DF]<11:0>])     /Fetch new state register

   Increment := (REG[DF]<11:0> <= REG[DF]<11:0>+DELTA)


   XOP<2>=0 => (Increment;                         /No transfer specified
                Limit => (STK_error));

   XOP<2>=1 => (XOP<0>=0 => (Increment;                                    /PUSH
                             Limit  => (Increment; #
                             ¬Limit => (MEM[REG[DF]<11:0>] <= REG[CF]);

                XOP<0>=1 => (REG[CF] <= MEM[REG[DF]<11:0>];                /POP
                             Increment;
                             Limit => (STK_error))
        )
```

```
AOP_SPARE := ()
```

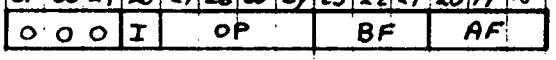
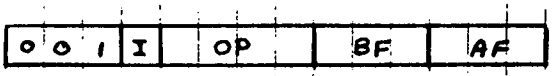
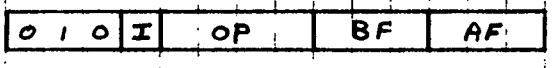
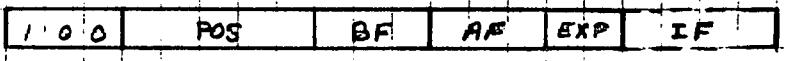
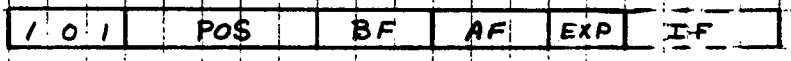
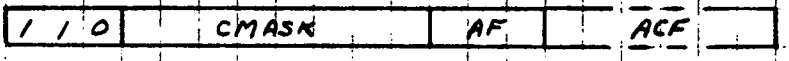
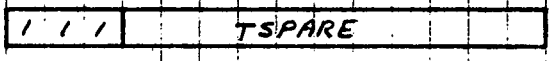


FIELD DEFINITIONS FOR MICROINSTRUCTION REGISTER



FIELD DEFINITION FOR STATE REGISTER

FIELD DEFINITIONS
FIGURE 1

INSTRUCTION TYPE	INSTRUCTION FORMAT	ACF USAGE	FIGURE NUMBER
LOGICAL		①	FIG. 3
ARITHMETIC		①	FIG 4
SHIFT/ROTATE		①	FIG. 5
EXTENDED		①	FIG. 6
INSERT		②	FIG. 7
EXTRACT		②	FIG. 8
CONDITIONAL		③	FIG. 16
SPARE		④	

NOTES: USAGE OF THE ACF FIELD (BITS 0-17) IS AS FOLLOWS

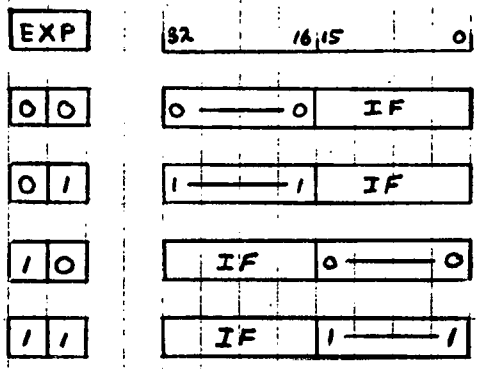
① ACF IS EITHER IMMEDIATE DATA OR A-MACHINE INSTRUCTION AS DETERMINED BY THE I BIT.

② ACF IS IMMEDIATE MASK DATA. (SEE BELOW)

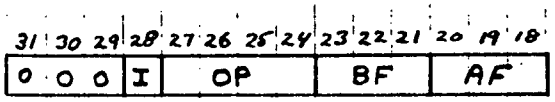
③ ACF IS CONDITIONALLY EXECUTED A-MACHINE INSTRUCTION

④ ACF IS UNCONDITIONALLY EXECUTED A-MACHINE INSTRUCTION

EXPANSION OF ACF FIELD FOR USE AS IMMEDIATE DATA



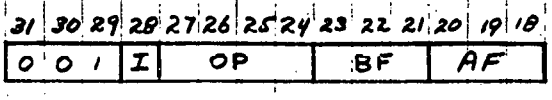
T-MACHINE INSTRUCTION FORMATS
FIGURE 2



0	OP1 INPUT IS REG[B]
1	OP1 INPUT IS EXPANDED ACF FIELD

0	0	0	0	REG[AF]	⇐ REG[AF]	NOP
0	0	0	1	REG[AF]	⇐ REG[AF] ∧ OP1	AND
0	0	1	0	REG[AF]	⇐ REG[AF] ∧ ¬ OP1	
0	0	1	1	REG[AF]	⇐ Ø	CLEAR
0	1	0	0	REG[AF]	⇐ REG[AF] ∨ OP1	OR
0	1	0	1	REG[AF]	⇐ OP1	XFR
0	1	1	0	REG[AF]	⇐ REG[AF] ⊕ OP1	XOR
0	1	1	1	REG[AF]	⇐ ¬ REG[AF] ∧ OP1	
1	0	0	0	REG[AF]	⇐ REG[AF] ∨ ¬ OP1	
1	0	0	1	REG[AF]	⇐ ¬ (REG[AF] ⊕ OP1)	XNOR
1	0	1	0	REG[AF]	⇐ ¬ OP1	CXFR
1	0	1	1	REG[AF]	⇐ ¬ (REG[AF] ∨ OP1)	NOR
1	1	0	0	REG[AF]	⇐ 37777777777_8	
1	1	0	1	REG[AF]	⇐ ¬ REG[AF] ∨ OP1	
1	1	1	0	REG[AF]	⇐ ¬ (REG[AF] ∧ OP1)	NAND
1	1	1	1	REG[AF]	⇐ ¬ REG[AF]	COM

LOGICAL INSTRUCTION FORMAT
FIGURE 3



0	OP2 is REG[BF]
1	OP2 is EXPANDED ACF FIELD

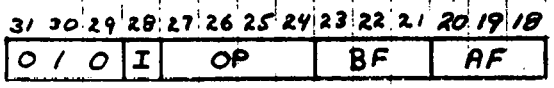
0	0	0	0	REG[AF]	⇐	REG[AF] + OP2		ADD
0	0	0	1	REG[AF]	⇐	REG[AF] + OP2 + C		ADC
0	0	1	0	REG[AF]	⇐	REG[AF] + ¬OP2 + 1		SUB
0	0	1	1	REG[AF]	⇐	REG[AF] + ¬OP2 + C		SBB
0	1	0	0	REG[AF]	⇐	REG[AF] + DEL	①	INC
0	1	0	1	REG[AF]	⇐	REG[AF] + DEL + C		ICC
0	1	1	0	REG[AF]	⇐	REG[AF] + ¬DEL + 1		DEC
0	1	1	1	REG[AF]	⇐	REG[AF] + ¬DEL + C		DCC
1	0	0	0					
1	0	0	1					
1	0	1	0					
1	0	1	1					
1	1	0	0					
1	1	0	1					
1	1	1	0					
1	1	1	1					

COMPUTATION IS THE SAME AS ABOVE EXCEPT THAT REG[AF] REMAINS UNCHANGED; CONDITION CODES ARE SET

NOTES:

① DEL IS BF FIELD EXTENDED WITH ZEROS

ARITHMETIC INSTRUCTION FORMAT
FIGURE 4



0	SHIFT AMOUNT IS REG[BF]
1	SHIFT AMOUNT IS EXPANDED ACF FIELD

0	- SINGLE LENGTH SHIFT ①
1	- DOUBLE LENGTH SHIFT ②

0	- SHIFT AMOUNT INDICATED BY I BIT
1	- SHIFT AMOUNT IS DEL ③

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

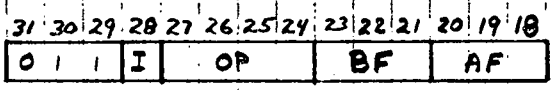
NOTES:

① REG[AF] IS SOURCE AND DESTINATION ON SINGLE SHIFTS

② ON DOUBLE SHIFT:
HIGH ORDER PART IS REG[AF]
LOW ORDER PART IS REG[AF ⊕ 1]

③ DEL IS BF FIELD EXTENDED WITH ZEROS

SHIFT / ROTATE INSTRUCTION FORMAT
FIGURE 5



0	SECOND OPERAND IS REG[BF]
1	SECOND OPERAND IS EXPANDED ACF FIELD

0	0	0	0	MULTIPLY STEP ①
0	0	0	1	DIVIDE STEP ①
0	0	1	0	FORM EXCESS SIX
0	0	1	1	
0	1	0	0	DECIMAL ADDITION
0	1	0	1	DECIMAL SUBTRACTION
0	1	1	0	BINARY TO DECIMAL CONVERSION STEP ①
0	1	1	1	DECIMAL TO BINARY CONVERSION STEP ①
1	0	0	0	DELAY CYCLE ②
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

BF - OPERAND REGISTER

AF OPERAND/DESTINATION REGISTER

NOTES:

① THESE OPERATIONS USE DOUBLE LENGTH OPERANDS REG[AF] AND REG[AF⊕1]

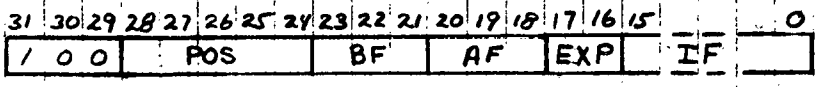
② DELAY CYCLE SETS NO CONDITION CODES

	INPUTS			OUTPUTS		
	REG[AF]	REG[AF⊕1]	OP2	REG[AF]	REG[AF⊕1]	REG[BF]
MULTIPLY STEP	∅	MULTIPLIER	MULTIPLICAND	HI RESULT	LO RESULT	X
DIVIDE STEP	DIVISOR	—	DIVIDEND	REMAINDER	QUOTIENT	X
EXCESS SIX		—	DECIMAL NUMBER	EXCESS SIX	X	X
DECIMAL ADD	DECIMAL OPERAND	—	DECIMAL OPERAND	DECIMAL RESULT	X	X
DECIMAL SUB	DECIMAL OPERAND	—	DECIMAL OPERAND	DECIMAL RESULT	X	X
BIN TO DEC	∅	∅	BINARY NUMBER	DECIMAL HIGH PART	DECIMAL LOW PART	X
DEC TO BIN	DECIMAL HIGH PART	DECIMAL LOW PART	—	X	X	BINARY NUMBER

SEMANTIC OF EXTENDED OPERATIONS

EXTENDED INSTRUCTION FORMAT

FIGURE 6



POS - AMOUNT OF LEFT ROTATE

BF - SOURCE REGISTER

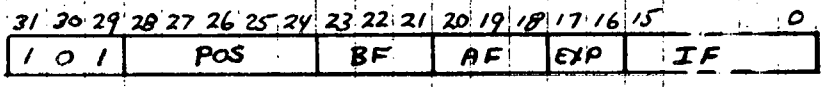
AF - DESTINATION REGISTER

EXP | IF - IMMEDIATE MASK DATA

NOTES:

① OPERATION:
REG[AF] ⇐ (LEFT_ROTATE(REG[BF], POS) ∧ MASK) ∨ (REG[AF] ∧ ¬MASK)
WHERE MASK IS EXPANDED ACF FIELD

INSERT INSTRUCTION FORMAT
FIGURE 7



POS - AMOUNT OF LEFT ROTATE

BF - SOURCE REGISTER

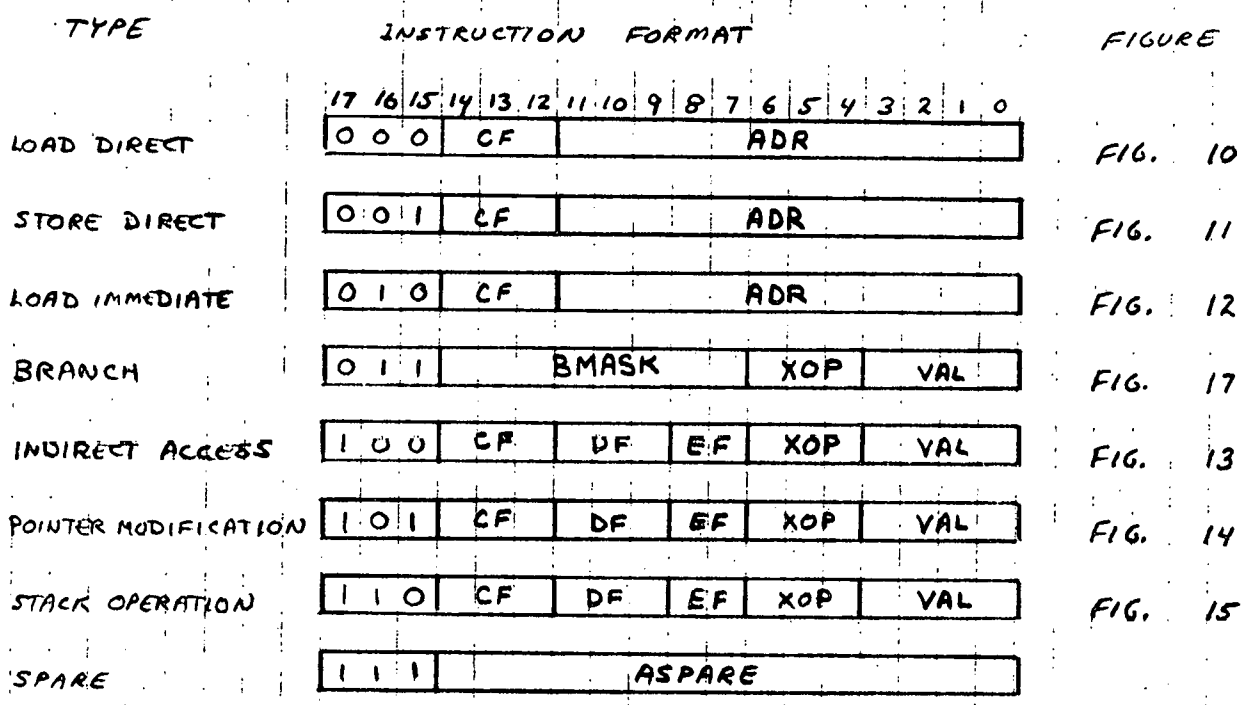
AF - DESTINATION REGISTER

EXP | IF - IMMEDIATE MASK DATA

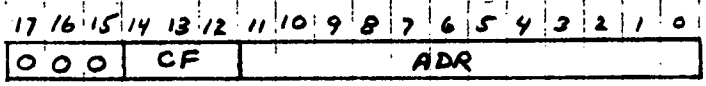
NOTES:

① OPERATION:
REG[AF] ⇐ LEFT_ROTATE(REG[BF], POS) ∧ MASK
WHERE MASK IS EXPANDED ACF FIELD

EXTRACT INSTRUCTION FORMAT
FIGURE 8



A-MACHINE INSTRUCTION FORMAT
FIGURE 9

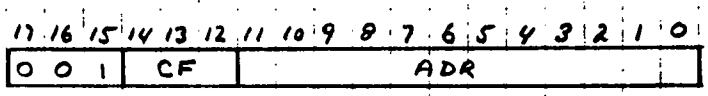


CF - DESTINATION REGISTER

ADR - SOURCE ADDRESS IN CONTROL MEMORY

REG[CF] ⇐ MEM[ADR]

LOAD REGISTER INSTRUCTION FORMAT
FIGURE 10

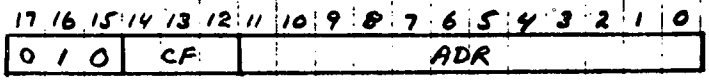


CF - SOURCE REGISTER

ADR - DESTINATION ADDRESS IN CONTROL MEMORY

MEM[ADR] ⇐ REG[CF]

STORE REGISTER INSTRUCTION FORMAT
FIGURE 11

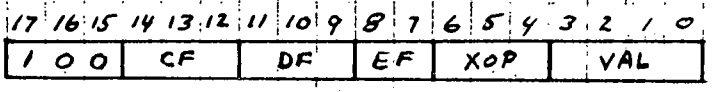


CF - DESTINATION REGISTER

ADR - IMMEDIATE DATA

REG[CF]<11:0> ⇐ ADR

REGISTER IMMEDIATE INSTRUCTION FORMAT
FIGURE 12



CF - REGISTER DESIGNATOR

DF - REGISTER DESIGNATOR

EF	
0 0	- NO MODIFICATION
0 1	- REG[DF] ⇐ REG[DF] + VAL
1 0	- REG[CF] ⇐ REG[DF] + VAL
1 1	- REG[CF] ⇐ REG[CF] + VAL; REG[DF] ⇐ REG[DF] + VAL

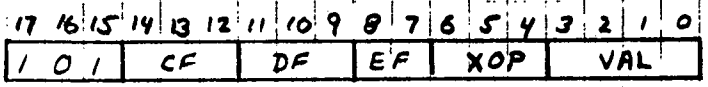
- POINTER MODIFICATION

XOP			
0 0 0	REG[CF]	⇒	REG[DF]
0 0 1	REG[CF]	⇒	MEM[REG[DF]]
0 1 0	REG[CF]	⇒	EXT[REG[DF]]
0 1 1	REG[CF]	⇐	MEM[REG[DF]]
1 0 0	MEM[REG[CF]]	⇐	MEM[REG[DF]]
1 0 1	MEM[REG[CF]]	⇒	EXT[REG[DF]]
1 1 0	REG[CF]	⇐	EXT[REG[DF]]
1 1 1	MEM[REG[CF]]	⇐	EXT[REG[DF]]

- MEMORY OPERATION CODE

VAL - IMMEDIATE VALUE FOR POINTER MODIFICATION, BIT 3 IS EXTENDED.

INDIRECT ACCESS INSTRUCTION FORMAT
FIGURE 13



CF - DESTINATION REGISTER

DF - SOURCE REGISTER

EF - REGISTER MODIFICATION OPERATION

EF	
0 0	REG[CF] ← REG[CF] + (0000000000 1 DF) ①
0 1	REG[CF] ← REG[CF] + ¬(0000000000 1 DF) ①
1 0	REG[CF] ← REG[CF] + REG[DF]
1 1	REG[CF] ← REG[CF] + ¬ REG[DF]

XOP - TEST ON RESULT OF POINTER MODIFICATION

XOP	
0	
1	- LOOP IF GREATER THAN ZERO
0	
1	- LOOP IF EQUAL TO ZERO
0	
1	- LOOP IF LESS THAN ZERO

VAL - AMOUNT OF RELATIVE JUMP IF SPECIFIED CONDITION IS TRUE. BIT 3 IS EXTENDED.

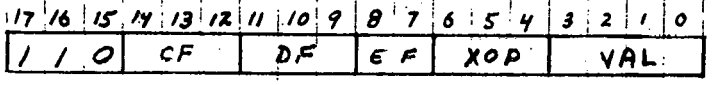
NOTES:

① DF FIELD IS EXTENDED WITH ZEROS

② A "1" IN THE APPROPRIATE XOP FIELD BIT INDICATES THAT THE RELATIVE JUMP WILL OCCUR IF THE INDICATED CONDITION RESULTS FROM THE POINTER MODIFICATION

POINTER MODIFICATION INSTRUCTION FORMAT

FIGURE 14



CF - SOURCE OR DESTINATION FOR STACK OPERATION

DF - CONTROL REGISTER

EF - SPARE

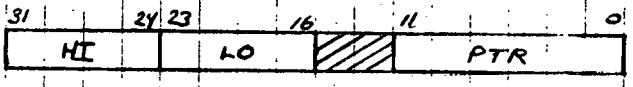
XOP

0 - NO TRANSFER, COUNT ONLY
1 - TRANSFER
0 - NO TEST AGAINST BOUNDS
1 - TEST AGAINST BOUNDS
0 - PUSH
1 - POP

VAL - INCREMENT AMOUNT FOR POINTER. BIT 3 IS EXTENDED.

NOTES:

① LAYOUT OF CONTROL REGISTER



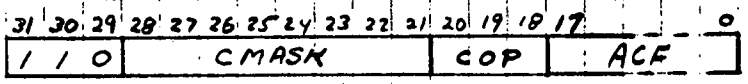
HI - LOW EIGHT BITS OF HIGH LIMIT

LO - LOW EIGHT BITS OF LOW LIMIT

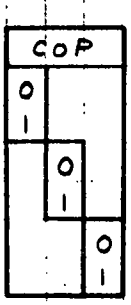
PTR - POINTER TO TOP OF STACK

STACK INSTRUCTION FORMAT

FIGURE 15



CMASK - MASK FOR CODE TEST



- TEST SPECIFICATION
- NORMAL SENSE
- COMPLEMENTED SENSE
- USE NORMAL CODES
- USE COMPLEMENTED CODES
- USE CONDITION CODES
- USE INDICATOR CODES

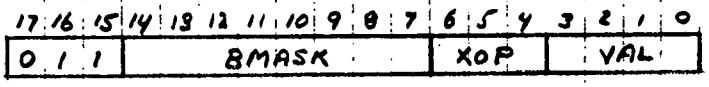
ACE - CONDITIONALLY CONTROLLED A-MACHINE OPERATION

NOTES:

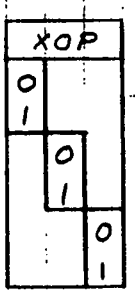
① A-MACHINE OPERATION SPECIFIED IS SKIPPED IF:

$$COP\langle 2\rangle \oplus \bigvee_i (CMASK_i \wedge (COP\langle 1\rangle \oplus ((\neg COP\langle 0\rangle \wedge CCODE_i) \vee (COP\langle 0\rangle \wedge ICODE_i))))$$

CONDITIONAL INSTRUCTION FORMAT FIGURE 16



BMASK - MASK FOR CODE TEST



- TEST SPECIFICATION
- NORMAL SENSE
- USE NORMAL CODES
- USE COMPLEMENTED CODES
- USE CONDITION CODES
- USE INDICATOR CODES

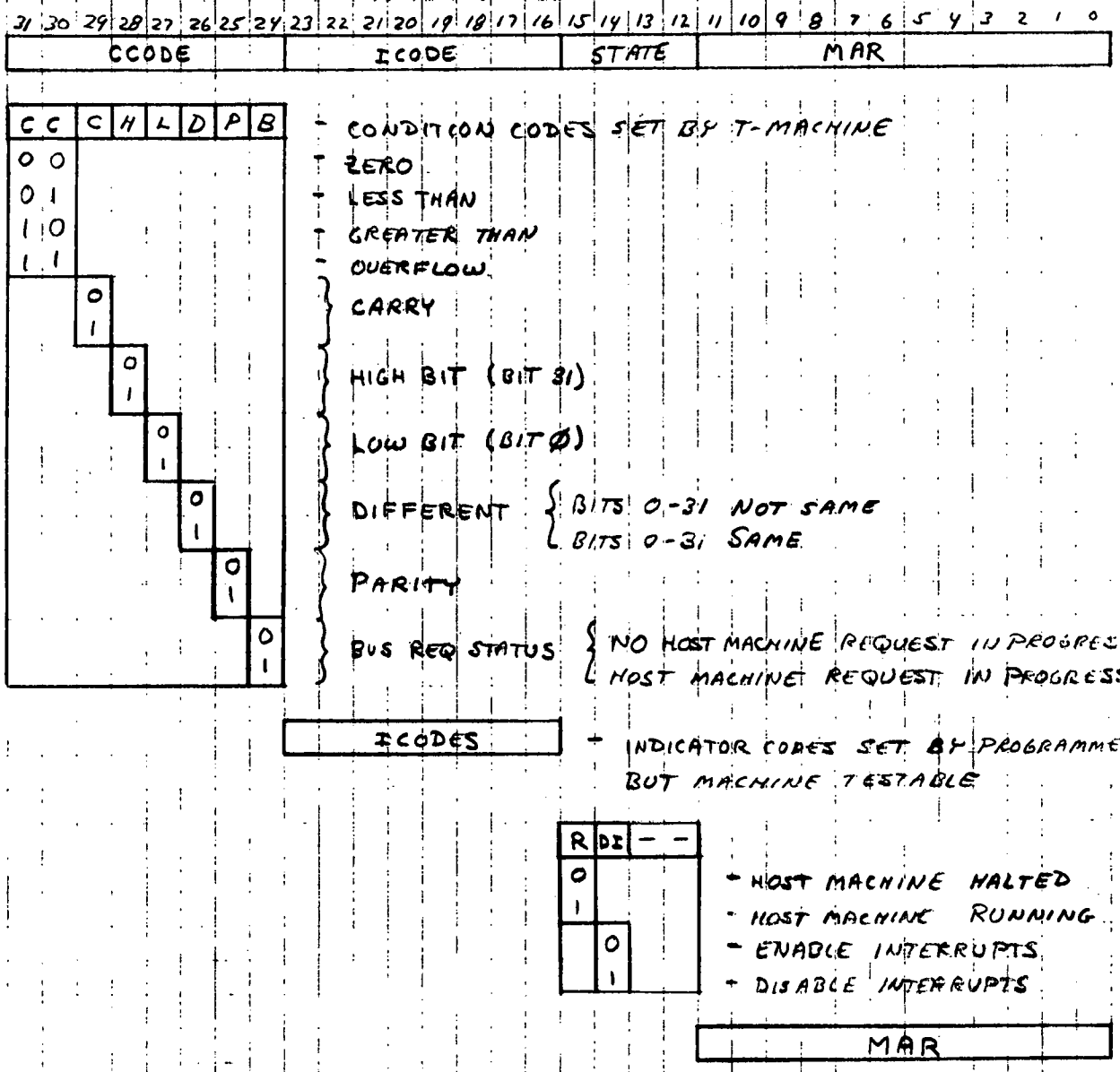
VAL - RELATIVE BRANCH AMOUNT BIT 3 IS EXTENDED

NOTES:

① BRANCH TO MAR + VAL IF

$$XOP\langle 2\rangle \oplus \bigvee_i (CMASK_i \wedge (XOP\langle 1\rangle \oplus ((\neg XOP\langle 0\rangle \wedge CCODE_i) \vee (XOP\langle 0\rangle \wedge ICODE_i))))$$

BRANCH INSTRUCTION FORMAT FIGURE 17



REGISTER Ø FORMAT

FIGURE 18