SECTION 2
FUNCTIONAL DESCRIPTION

2.1 INTRODUCTION

Figure 2-1 illustrates the major functional elements of the EBox. The purpose of this drawing is to support the functional descriptions contained in this section. The major data and address paths and the individual controls introduced in the previous section are shown on this diagram with some additional detail. Major interfaces are also shown in some detail.

The interface between the EBox and the MBox is not a bus, but is functionally shown and described as if it were, because its operation is similar to that of a bus.

As described in Section 1, the EBox serves as the Instruction Execution Unit for the KL10 system. Access to main memory is logically controlled by the MBox; therefore, as the EBox requires memory operands or instructions, it performs MBox cycles to obtain these words. These cycles take place over the E/M interface. In a similar fashion, access to I/O devices is via the EBus. Devices may communicate with the EBox over the EBus by utilizing the priority interrupt system. In addition, as the EBox requires status or data from devices connected to the EBus or wishes to transmit data or control information to devices on the EBus, it does so by performing EBus cycles. These cycles take place over the EBus. Figure 2-2 illustrates these primary hardware cycles. The implementation of MBox or EBus cycles is via the microprograms stored in the CRAM.

2.2 MICROPROGRAM STATES AND PROCESSOR CYCLES

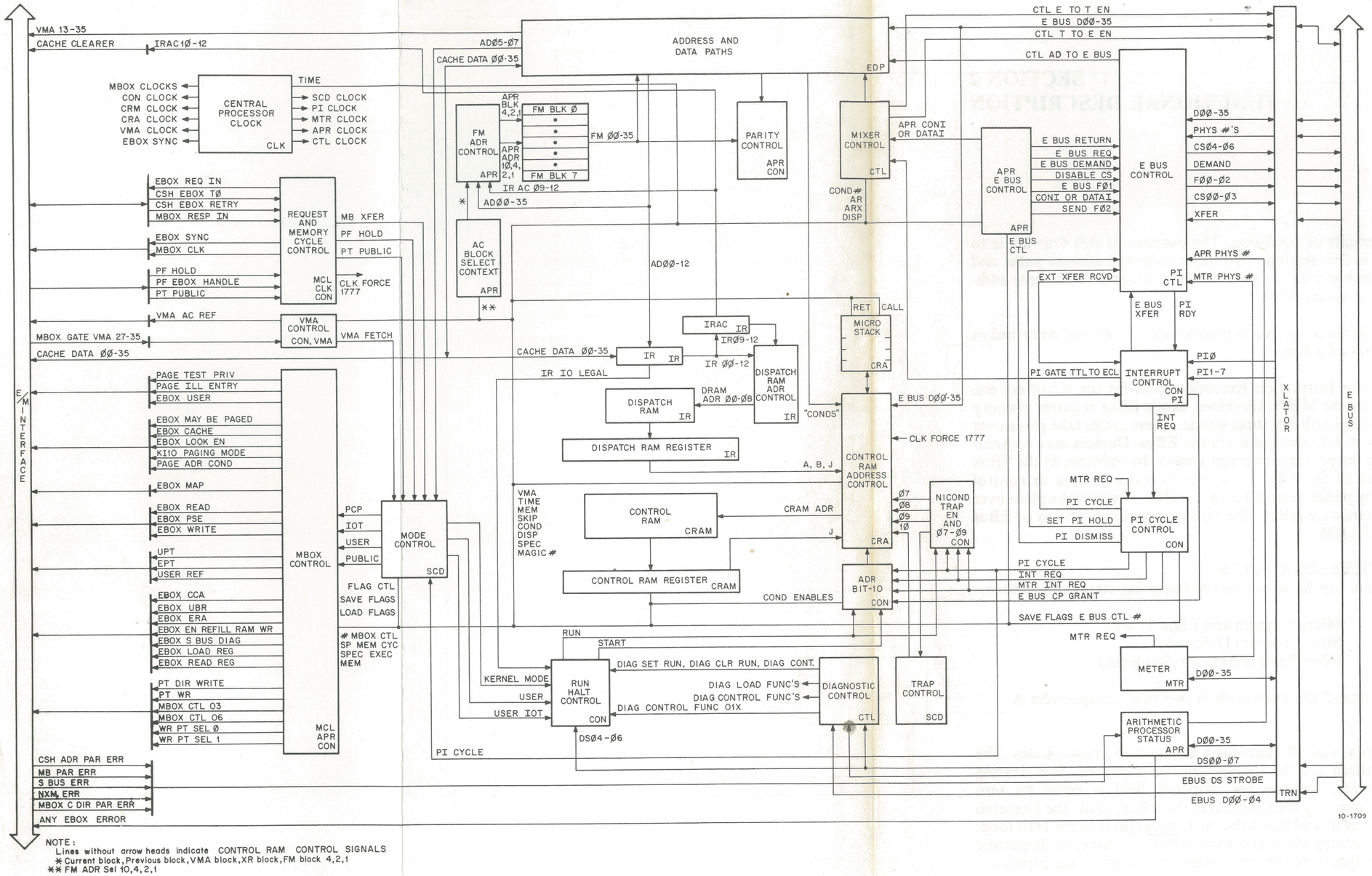
Referring to Figure 2-3, the EBox microprogram can be in one of the following states at any time:

- Microprogram Running
- Microprogram Wait State
- Microprogram Halt Loop
- Microprogram and EBox Frozen
- Microprogram Deferred
- EBox Reset (Power Up Sequence)

A discussion describing how to read and understand the microcode is provided in Appendix A.

2.2.1 EBox Reset

During the power up sequence, the EBox, MBox, and all controllers are reset to known states. The EBox, MBox, EBus, and SBus clocks are initialized and the CRAM register is cleared. This clearing action places the EBox in the diagnostic state, because the dispatch field is equal to zero (DISP/DIAG). A program running in PDP-11 memory then initializes the EBox, loads the Dispatch RAM and verifies it, loads the CRAM and verifies it, and starts the microprogram into the Halt loop. In general, at this time, the system must be bootstrapped; to accomplish this, a number of diagnostic functions are necessary. This is discussed in Section 3 and in the system and interface descriptions.



NOTE:
Lines without arrow heads indicate CONTROL RAM CONTROL SIGNALS
* Current block, Previous block, VMA block, XR block, FM block 4,2,1
** FM ADR Sel 10,4,2,1

Figure 2-1 EBox Functional Block Diagram

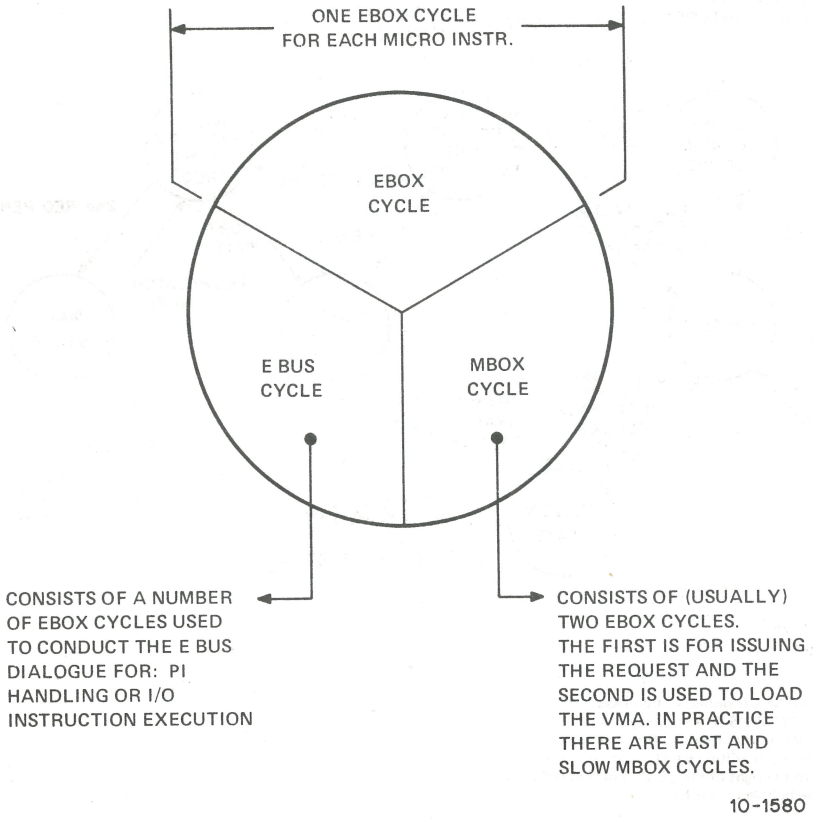


Figure 2-2 Primary Hardware Cycles

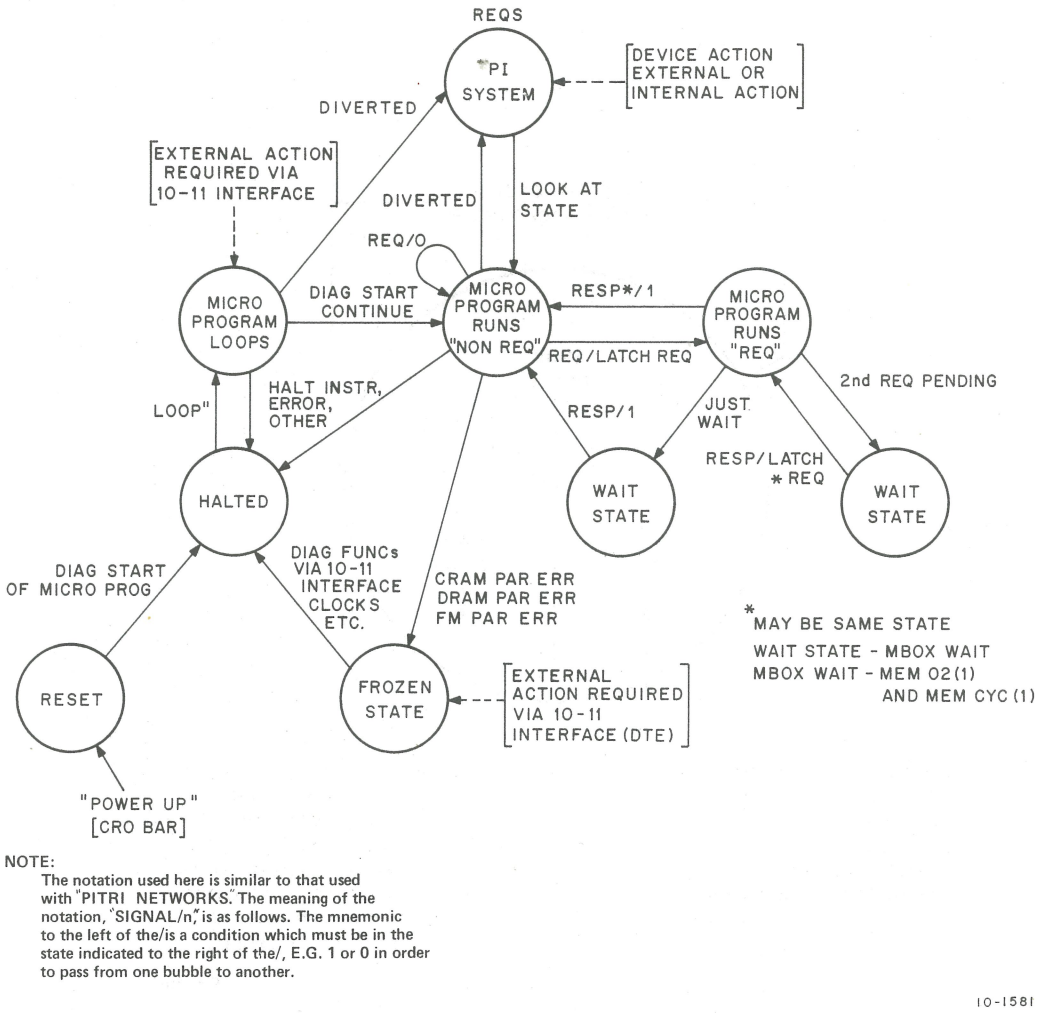


Figure 2-3 Microprogram Static States

2.2.2 Microprogram Halt Loop

The Halt loop is entered following a NICOND Dispatch, when RUN and PI CYCLE are found clear. Figure 2-4 is the flow diagram. Referring to Figure 2-5, the EBox contains a synchronizer (CON START), which is set for three clock periods when CONTINUE is pressed. In addition, it also contains a flag (CON INSTR GO), which is set by CONTINUE and remains set until a HALT instruction is performed. The RUN flag in the EBox consists of a RUN source enabled by DIAG SET RUN and CON INSTR GO true. Referring to Figure 2-4, assuming a HALT instruction has just been performed (JRST 4) and the RUN flag has been found clear at NICOND Dispatch time, the Halt loop is entered. The following occur immediately:

The AR is cleared.
The HALT flag is set.
The current value of PC is loaded into VMA.
The current value of VMA is placed in PC.

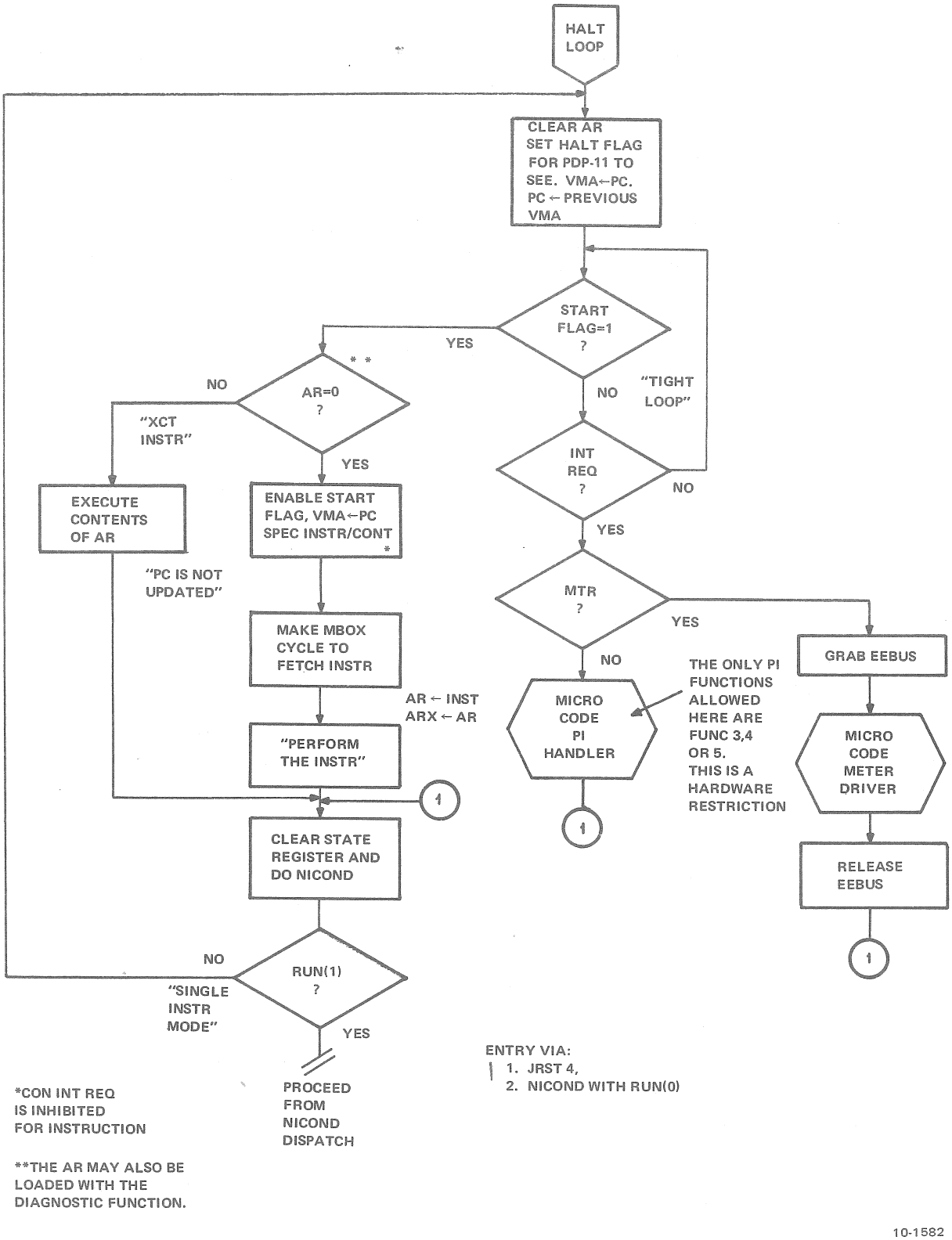


Figure 2-4 Microprogram Halt Loop

Thus, if the HALT instruction was fetched from location 600, and the effective address supplied in the HALT instruction was 100, PC would become 100 and VMA would become 601 (the updated PC value). The START flag is tested to determine if CONTINUE was pressed. In this case, START will be clear. If an interrupt is pending, the PI Handler is entered to service this interrupt.

When this is done the next instruction is requested. This is followed by a NOOP microinstruction. Finally, the State register (a hardware register in the EBox) is initialized clear. Then NICOND Dispatch is issued and the Halt loop is entered again.

If no interrupts are pending, the "Tight loop" is entered, continually checking the START flag and interrupt requests. Note that HALT INSTR does not clear the RUN source, but merely clears INSTR GO, which removes the CON RUN signal (Figure 2-5).

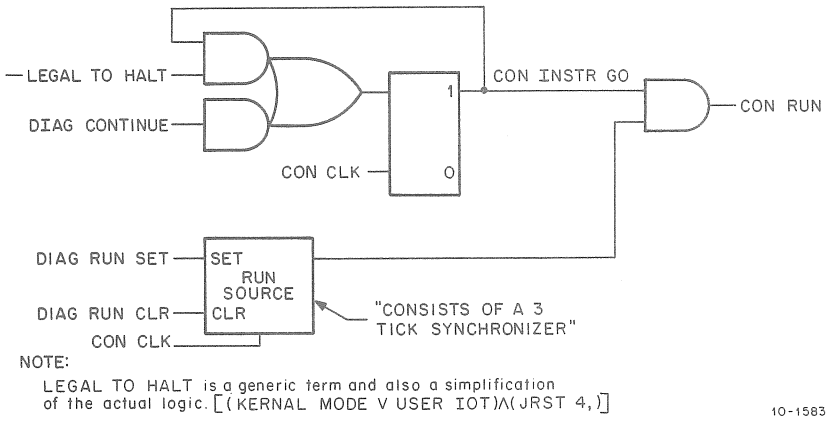


Figure 2-5 Run-Halt-Continue Logic

The HALT instruction is a "privileged instruction"; therefore, the EBox must be in either diagnostic, USER IOT, or KERNEL mode to clear CON INSTR GO. The PDP-11 may clear the RUN source at any time by issuing (via the 10-11 Interface) DIAG RUN CLR. This causes the Tight loop to be entered at the next NICOND Dispatch (assuming no interrupts are pending).

If it is desired to execute a single instruction, the AR may be loaded with the desired instruction by use of the prescribed DIAG function, issued via the 10-11 Interface. After the AR has been loaded, the START flag is enabled by issuing DIAG CONTINUE. The AR is tested for a nonzero value. If it is nonzero, the contents of AR are executed; upon its completion, the Halt loop is once again entered.

It should be noted that PC+1 INHIBIT is true during the Execute function, to prevent the PC from being updated. Similarly, by clearing AR and pressing CONTINUE while CON RUN is disabled, one instruction may be fetched at a time and executed, or the program may be resumed if CON RUN is true after performing the instruction in AR. For this function, the microcode, at XCTW, is used to fetch the instruction and wait for it. This instruction is performed, and the PC is allowed to be updated by +1. At the end of the instruction, NICOND Dispatch is issued and the state of CON RUN is tested together with other hardware conditions, to determine what to do next.

2.2.3 Microprogram Running

Once the microprogram is running, it may enter any of the other states (Subsection 2.2). Normally, the microprogram passes through a regularly defined sequence consisting of at least the five main dispatches (Main loop) shown in Figure 2-6. Between each dispatch, some number of microinstructions is performed. A rough equivalence exists between the traditional computer machine cycles and those of the EBox. In general, the relationship is as shown in Table 2-1.

Table 2-1 EBox Main Loop/Traditional Machine Cycle Comparison

EBox Dispatch Main Loop	Traditional Machine Cycles
NICOND Dispatch	Instruction
EAMOD Dispatch	Address
A READ Dispatch	Fetch
DRAM J (See Note)	Execute
B WRITE Dispatch	Store

NOTE
This dispatch is referred to in the Microcode as IR Dispatch.

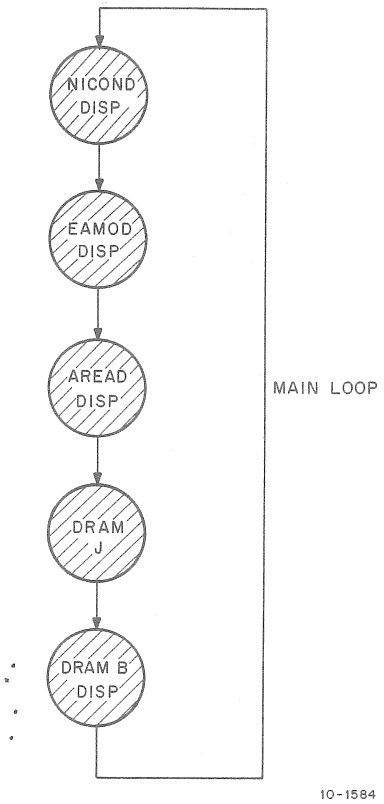


Figure 2-6 Dispatch Path State Diagram

Altogether, there are 16 dispatches. The five basic dispatches constitute the main loop; an additional eleven are, in general, instruction dependent and usually, if issued, follow an IR Dispatch (DRAM J DISP). Each time an EBox clock tick occurs, the CRAM register is loaded with a microinstruction. This microinstruction then contols formation of the next microinstruction address. This is accomplished by the particular coding of the appropriate microinstruction fields. In general, there are four types of CRAM address modifications (Figure 2-7):

Branch On Condition
Branch On Condition With Skip
Skip
Jump

The CRAM address logic samples conditions (Figure 2-8) supplied by various portions of EBox logic, together with the current microinstruction J, COND, and Dispatch fields, and then generates the next CRAM address (CR ADR 00–10).

2.2.4 Microprogram Wait State

As indicated in Figure 2-3, the Wait state (MBOX WAIT) occurs during memory requests involving the MBox. In general (Figure 2-9), three main uses of the Wait state exist. The first is to assure that the microprogram waits for an MBox response after having started an MBox cycle. The second use is to hold off a second MBox cycle when the MBox has not yet responded to the first MBox cycle.

As shown in Figure 2-10, the EBox clock control samples the following signals:

MBOX WAIT
VMA AC REF
RESP MBOX

If an MBox cycle is started, MEM CYCLE sets, as enabled by the request. It remains set until XFER is generated. When the request is to the MBox, and VMA 13–33 is nonzero, the XFER is generated as a direct result of MBOX RESPONSE IN. If, however, VMA 13–33 is zero, VMA 32–35 is a fast memory address and the EBox aborts the cycle. The XFER is a result of FM XFER, a signal generated from within the EBox itself. If VMA AC REF is true, the EBox clock ignores MBOX WAIT. However, when VMA AC REF is false and MBOX WAIT is true, the EBox clock may be inhibited.

The third case involves instruction prefetches from fast memory (Figure 2-11). For this situation, the microinstruction generating NICOND Dispatch also asserts MB WAIT. This is necessary because the EBox hardware requested the next instruction from the MBox rather than from fast memory. The MBox detects that the VMA address contained a fast memory address and aborts the cycle. The EBox hardware switches the ARX input to the AD output, thus reading from fast memory.

NOTE

XFER = MB XFER ∨ FM XFER

2.2.5 Microprogram and EBox Frozen

The microprogram and EBox frozen state occur in practice when any of the following events occur:

1. DRAM Parity Error while the EBox clock is running.
2. CRAM Parity Error while the EBox clock is running.
3. Fast Memory Parity Error while the EBox clock is running.

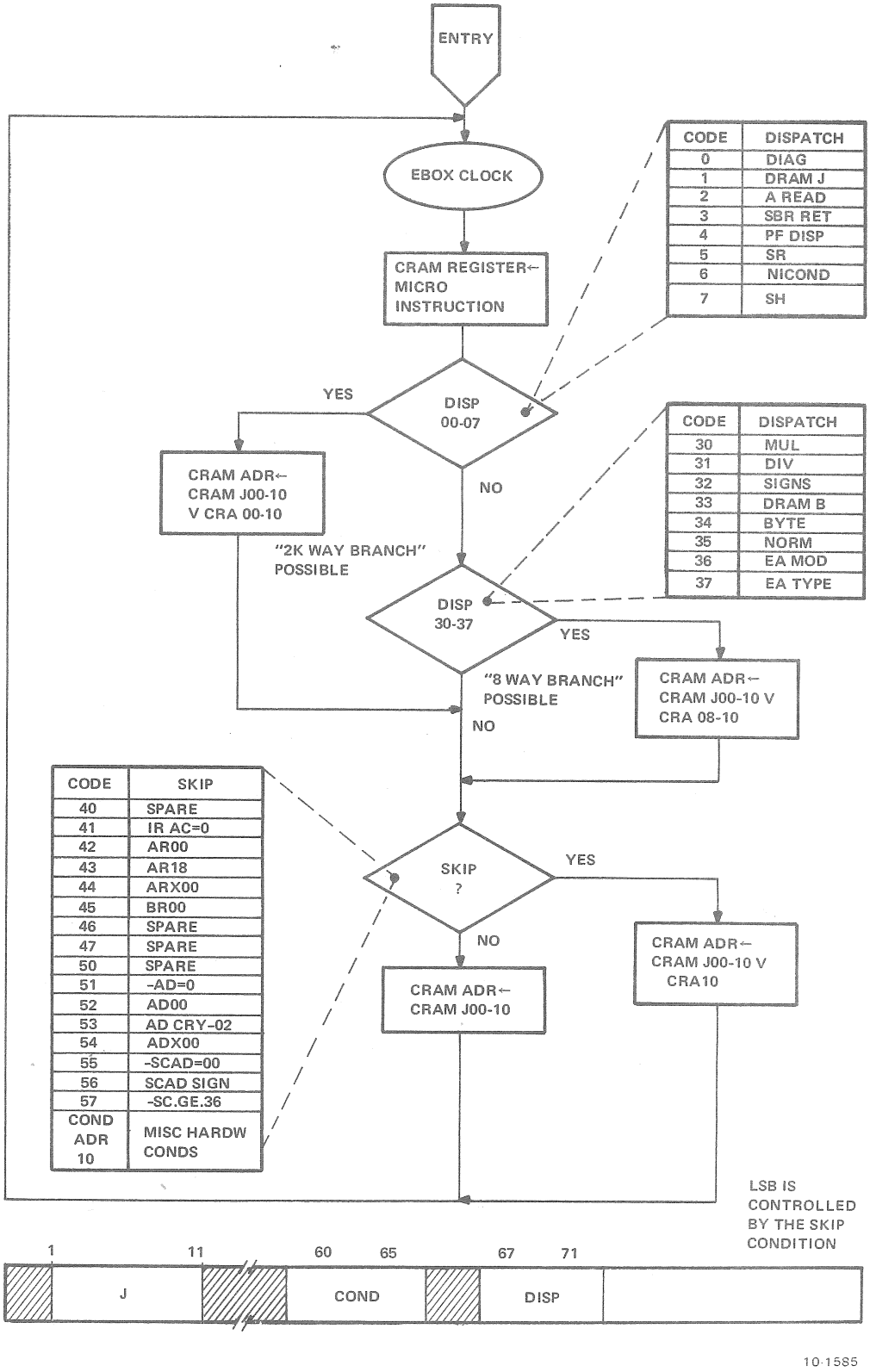


Figure 2-7 Basic Microprogram Address Control

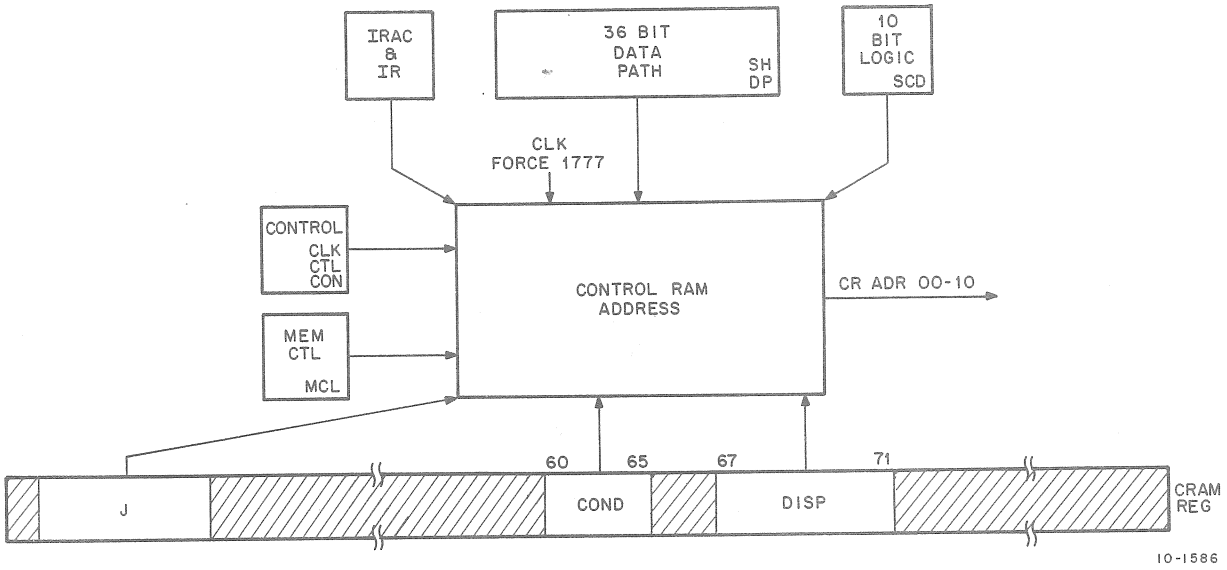


Figure 2-8 CRAM Address Inputs Simplified

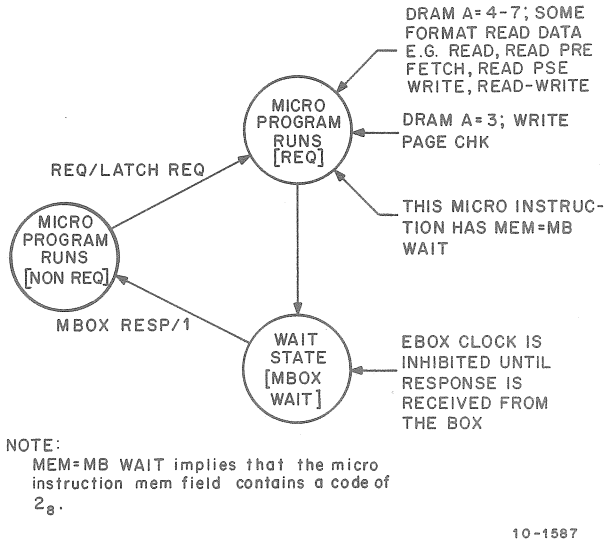


Figure 2-9 Wait State

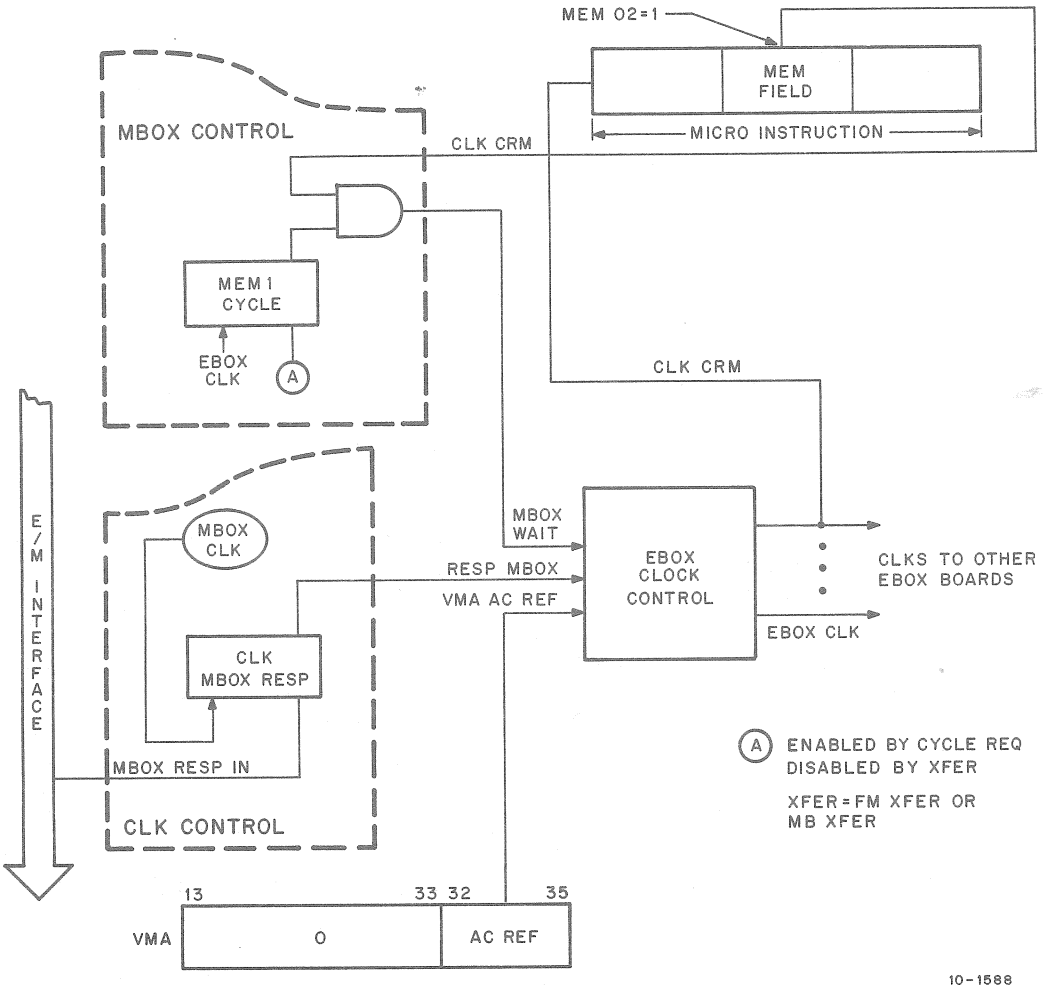
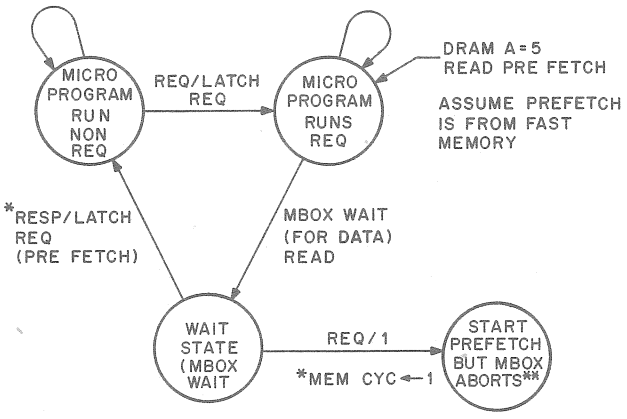


Figure 2-10 MBox Wait and EBox Clock



NOTE:
REQ/1 implies that the request sets up but VMA remains latched from the previous REQ until MBOX response. At that time RESP/LATCH REQ implies that the VMA can LATCH for the 2nd REQ.

* Same EBOX CLOCK

** MICRO program must reinitiate the FETCH later (NICOND)

10-1589

Figure 2-11 MBox Wait on Prefetch from Fast Memory

Associated with each of these error conditions is an enable that must be activated prior to the occurrence of the error to be detected. The three enables are listed in Table 2-2.

Table 2-2 Error Stop Enables

Enable	EBus Bit	Function
CLK FM PAR CHECK	32	DIAG FUNC 046
CLK CRAM PAR CHECK	33	DIAG FUNC 046
CLK DRAM PAR CHECK	34	DIAG FUNC 046

The DRAM words are coded in a specific fashion for each instruction. If a DRAM parity error occurs undetected, it implies that the DRAM word has picked up or dropped an even number of bits. Suppose, for example, that the DRAM J field picked up a bit, which changed the Jump address from 200 to 500. The microprogram would perform properly up to the point where it dispatched to the executor. Here, instead of jumping to the MOVE microprogram, it jumps to the half-word microprogram with erroneous results stored in the specified AC. In a similar fashion, a bit could be picked up or dropped in the DRAM A or B fields with equally disastrous results. The microprogram is a structured entity; an erroneous variation of any of its bits in the CRAM register causes errors in the execution of instructions and could cause the microprogram to lose control of the EBox. As an example, assume a microinstruction is loaded into the CRAM register. The Dispatch field, originally coded as DISP/DRAM B, because of a dropped bit, becomes instead DISP/SIGNS. Thus, the next CRAM address will be computed based on the signs of AR, BR, and AD instead of using the B field of the DRAM word; and this would create the wrong CRAM addresses.

In general, all instructions in the KL10 Instruction Set utilize fast memory in some way. In addition, the microprogram always uses fast memory to set up the indexing function. If fast memory parity errors were not detected, bad data could be generated and possibly erroneous instructions fetched from fast memory.

2.2.6 Microprogram Deferred

The microprogram samples the EBox hardware only at specific times for pending priority interrupts or pending traps. One such time is at NICOND Dispatch. Currently, eight possible conditions can occur (Table 2-3). Three of these are related to interrupts, two are related to traps, one is for a halted condition, and the remaining two are the more general cases. Here, the deferred condition is taken to mean that upon finding an interrupt or a trap pending, the microprogram defers the pending instruction and instead handles the interrupt or trap first. In terms of interrupts, the highest priority condition is with PI CYCLE (1). This implies that on the previous NICOND Dispatch INT REQ was true and the microprogram diverted to the PI Handler to perform the first part of a standard (40 + 2n) interrupt. For example, assuming device (n) interrupts, the PI system carries out the necessary dialogue and asserts PI READY. This results in the assertion of INT REQ, which is sampled at NICOND Dispatch time. Now assuming PI CYCLE (0) and RUN (1), the PI Handler is entered. The handler reads the API function word on the EBus into AR and processes it. Here we will assume it specifies a standard interrupt (40 + 2n). Assume the conditions shown in Figure 2-12.

Table 2-3 NICOND Priorities

Why	Where to Go	Conditions to Consider							Low-Order CRAM ADR Bits as Follows				
		PI CYCLE	RUN	MTR INT REQ	INT REQ	AC REF	TRAP EN	ANY TRAP	NICOND TRAP EN	NICOND 07	NICOND 08	NICOND 09	NICOND 10
Second part PI Cycle	BASE ADR	1	0/1	0/1	0/1	0/1	0/1	0/1	0	0	0	0	0
Halt Instruction or 11 caused	BASE ADR+2	0	0	0/1	0/1	0/1	0/1	0/1	0	0	0	1	0
MTR INT Request	BASE ADR+4	0	1	1	0/1	0/1	0/1	0/1	0	0	1	0	0
PI Request but not MTR	BASE ADR+6	0	1	0	1	0/1	0/1	0/1	0	0	1	1	0
Instruction fetched from memory and no traps pending	BASE ADR+12	0	1	0	0	0	1	0	1	1	0	1	0
Instruction fetched from memory and a trap is pending	BASE ADR+13	0	1	0	0	0	1	1	1	1	0	1	1
Instruction must be fetched from FM and no traps pending	BASE ADR+16	0	1	0	0	1	1	0	1	1	1	1	0
Instruction must be fetched from FM and a trap is pending	BASE ADR+17	0	1	0	0	1	1	1	1	1	1	1	1

/// – Overriding condition

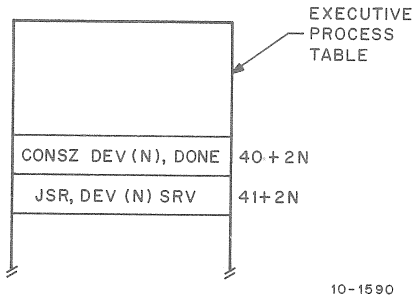


Figure 2-12 PI 40 + 2n Skip

The PI Handler sets PI CYCLE, interlocking the microprogram and PI Board, and temporarily, at least, preventing any further INT REQs from being sampled by the microprogram. The PI Handler forces an instruction fetch from 40 + 2n; note that NICOND is not now generated. The SKIP instruction in 40 + 2n is performed and one of two possible actions results (in this case) from the state of the DONE flag:

DONE (1) – Perform the instruction in 41 + 2n; this instruction must be of such a nature that PI CYCLE is cleared (JSR is such an instruction.)

DONE (0) – Dismiss the interrupt and clear PI CYCLE.

For this example, assume the instruction should be fetched from 41 + 2n [DONE (1)]. The dispatch, therefore, is back to the PI Handler for the second part of the interrupt.

When the PI Handler releases the PI system, NICOND Dispatch finds PI CYCLE still set. Because this is the highest priority condition at NICOND time, the dispatch is back to the PI Handler for the second part of the interrupt. The PI Handler generates the appropriate 41 + 2n address and causes the instruction to be performed, once again omitting a NICOND Dispatch. The instruction fetched must be one of the following:

JSR	
JSP	Changes the ACs; use
PUSH J	not recommended
MUUO	
SKIP (will be satisfied)	

All of these instructions cause PI CYCLE to be cleared.

2.2.7 Microprogram Organization

The basic control program modules are illustrated in Figure 2-13. The symbol containing the Data Storage Manager illustrated in Figure 2-13 represents a predefined process. Examples of such predefined processes include software and hardware subroutines, the Unibus dialogue, and even functions of an alarm clock.

In the microprogram context, the predefined processes represent functional areas of the microcode. Figures 2-14 through 2-21 represent the hardware that controls branching to each of the handlers illustrated on Figure 2-13.

These may be grouped as follows:

The *Startup and Stop Interface* (Figure 2-14) evaluates initial hardware conditions and dispatches to the appropriate handler. The nature of the condition could be a pending priority interrupt, halt condition, etc. Upon completion, all instructions must pass through this process. The mnemonic for the dispatch to this process is DISP/NICOND (Next Instruction Condition).

The *Effective Address Manager* (Figure 2-15) evaluates indirect address flag bit 13, index field bits 14–17 in the ARX (which contains the current instruction), and certain hardware conditions such as PIs or page failures. It either dispatches to the appropriate handler or calculates the effective address by requesting the necessary fast memory (Index) cycles or MBox Indirect (I) cycles. The mnemonic for the dispatch to this process is DISP/EAMOD (Effective Address Mode).

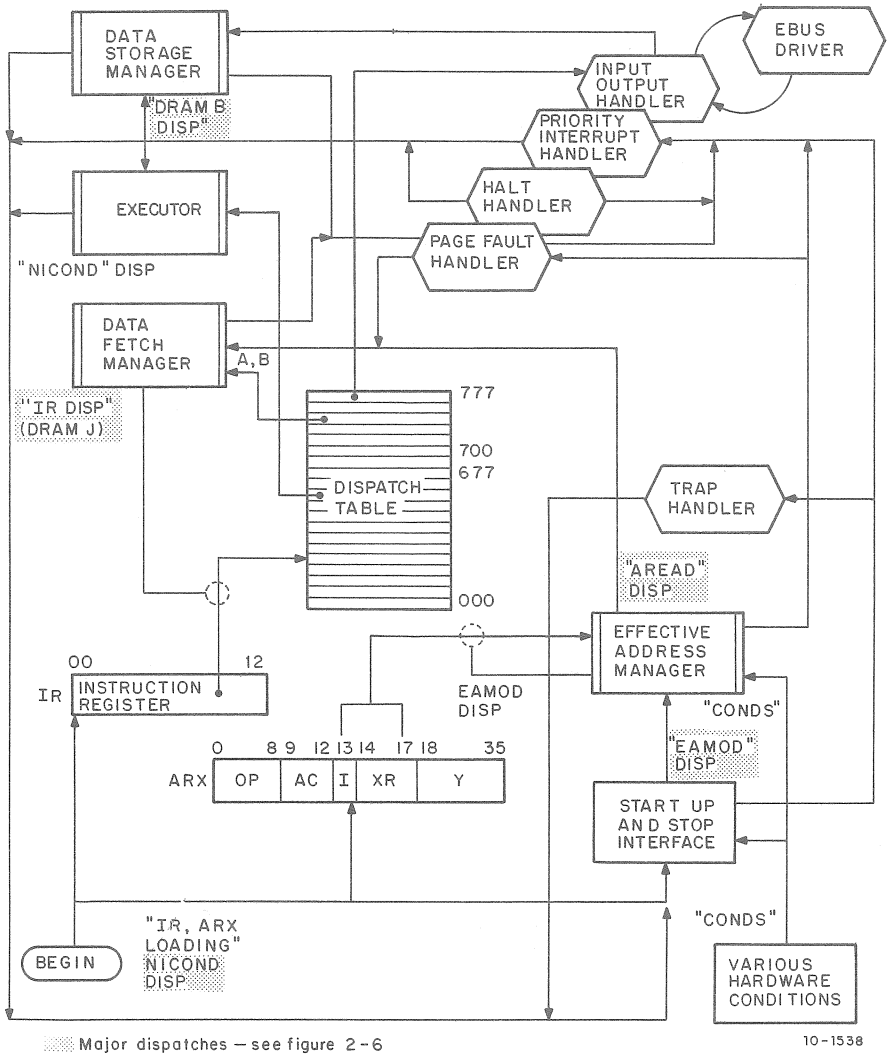


Figure 2-13 M Program Modules

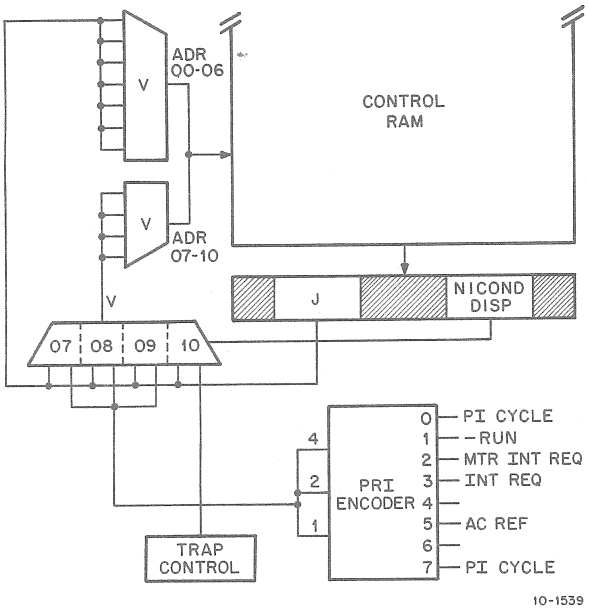


Figure 2-14 Startup and Stop Interface

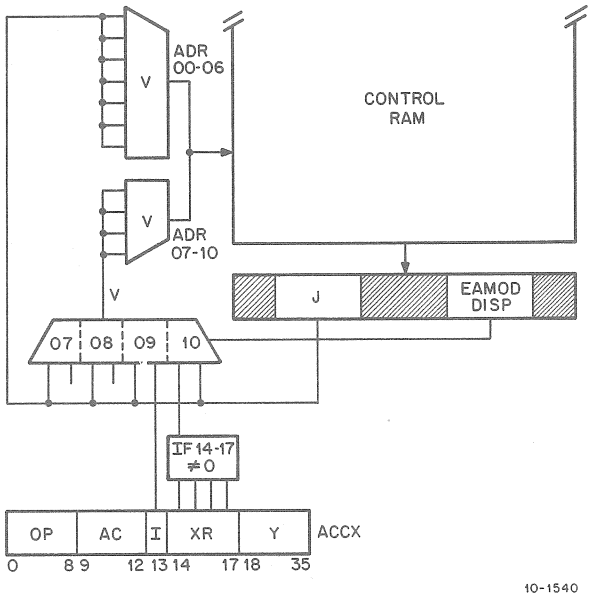


Figure 2-15 Effective Address Manager

The *Data Fetch Manager* (Figure 2-16) evaluates the 3-bit A (FETCH) field (for the current instruction), which is in the Dispatch Table. The code in the 3-bit field defines the type of data fetch or write or combination operation (if any) required. The Data Fetch Manager takes the proper action, i.e., enabling the EBox clock to stop as appropriate, dispatching directly to the executor, or initiating an instruction prefetch. Note the Instruction register is used to address the proper location in the Dispatch Table (DRAM) based upon the op code for the instruction.

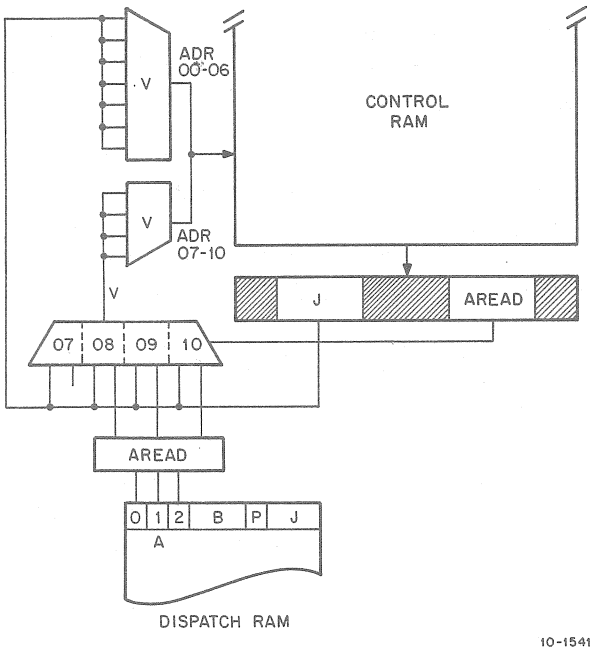


Figure 2-16 Data Fetch Manager

The *Dispatch Table* (Figure 2-17) consists of four fields:

1. DRAM A – Bits 0–2; defines the type of operand fetch cycle.

2. DRAM B – Bits 3–5; defines Jump, Skip, and Compare conditions for certain instructions, or result store mode, etc.

3. DRAM P – Bit 11; parity bit (parity is normally odd).

4. DRAM J – Bits 14–13; jump address. This is the entry address of the executor routine. The mnemonic for the dispatch to the executor is IR DISP (DRAM J) (Instruction Register Dispatch).

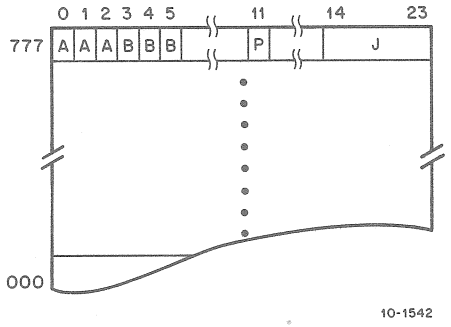


Figure 2-17 Dispatch Table Fields

The *Executor* routine (Figure 2-18) is the bulk of the microprogram. It contains a number of somewhat autonomous routines used to execute the instruction specific functions, e.g., move a half-word from one register to another or push a word onto a subroutine stack.

The *Data Store Manager* (Figure 2-19) dispatches on the DRAM B field. In addition, when called from the executor as a subroutine only, e.g., MEM/WRITE, it defines the appropriate MBox control signals and dialogue and initiates the write operation. When the Data Store Manager is entered in the context of a store cycle, control generally passes to that process from the Executor. Finally, a NICOND Dispatch is generated and control passes to the Startup and Stop Interface.

The *Priority Interrupt Handler* is dispatched to or from discrete points in the microprogram. Interrupts are scanned during NICOND Dispatch, while computing the effective address, and during certain longer instructions, such as BLT.

Control is passed to the Page Fault Handler (Figure 2-20) routine from the Effective Address Manager or Data Store Manager when the MBox asserts PF HOLD prior to an MBox response during a memory request. The implication is that a memory address violation occurred, i.e., an access failure. write protection violation, or similar violation. In addition, when implementing KL10-style paging, PF HOLD with EBOX HANDLE may be asserted to the EBox from the MBox. The implication here is that the paging address translation should be accomplished via microprogram rather than in the MBox itself. The Page Fault Handler is also used for certain error conditions.

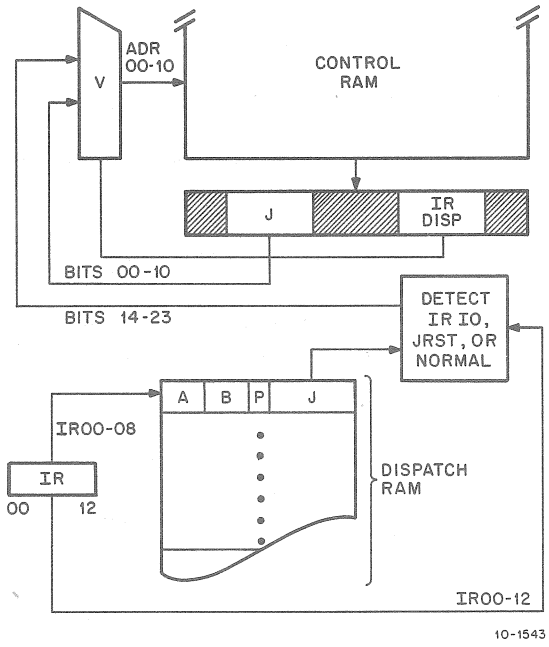


Figure 2-18 Executor

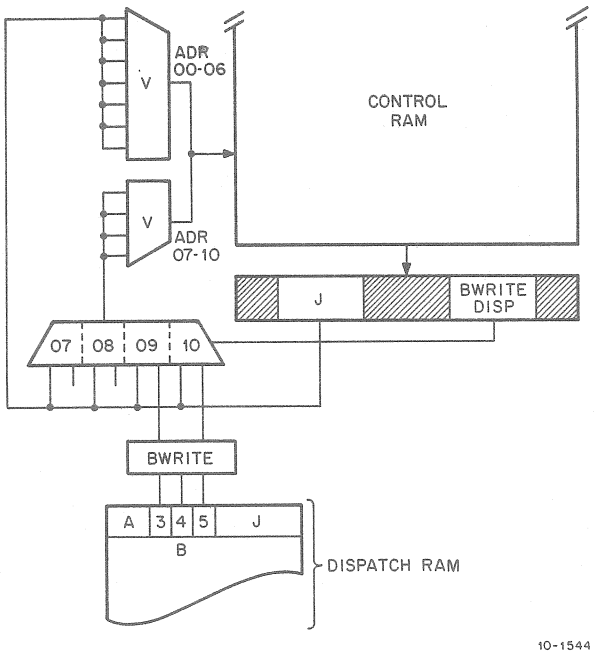


Figure 2-19 Data Store Manager

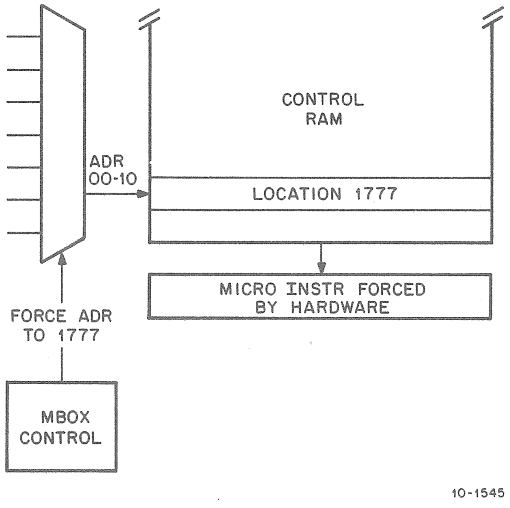


Figure 2-20 Page Fault Handler

The *Halt Handler* routine is entered from the Startup and Stop Interface when the RUN flip-flop is clear at NICOND Dispatch time. The RUN flip-flop can be cleared by various mechanisms. For example, when a HALT instruction is executed, RUN is disabled. On power up, RUN must be set by a diagnostic function initiated from the DTE20.

The *I/O Handler* (Figure 2-21) is dispatched via IR Dispatch from the Dispatch Table on DATAO, CONO after the data or status has already been fetched, or directly on DATAI, CONI, CONSO, or CONSZ. The handler calls the EBus driver, which generates the necessary EBus dialogue with the device. On BLKI or BLKO, the pointer has been fetched but must be updated, stored back at E, and the first word fetched. This is performed in the I/O Handler first. When the data has been fetched, the EBus driver is called. On DATAI or CONI, the EBus driver is called to negotiate the transfer from the selected device over the EBus to the EBox. The I/O Handler then passes control to the Data Store Manager where the data is stored.

2.3 BASIC MACHINE CYCLE

The basic machine cycle for a typical instruction is illustrated in Figures 2-22 and 2-23. The cycle begins at the EBox clock following NICOND Dispatch and terminates at the trailing edge of the next NICOND Dispatch. In this example, assume that the instruction MOVE 3 @ 200 (1) has been fetched from core memory symbolic location PC. The following information relates to the example:

PC/	MOVE 3 @ 200 (1)	Current Instruction
PC+1/	NEXT INSTRUCTION	
300/	000000, 000 100	Indirect Address = 300
100/	171717, 111111	Effective Address = 100
1/	000000, 000100	Index Register = 1

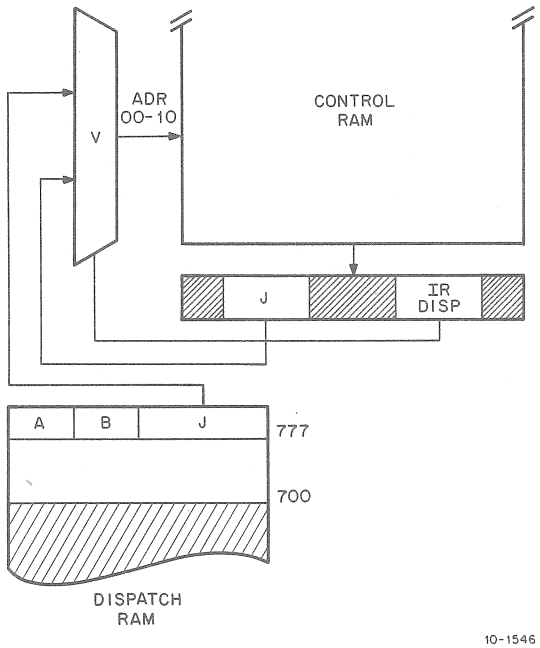


Figure 2-21 Input/Output Handler