EK-EBOX-UD-003

# EBOX
# INSTRUCTION EXECUTION UNIT
# UNIT DESCRIPTION

digital equipment corporation • marlborough, massachusetts

1st Edition, May 1976
2nd Edition, January 1976
3rd Edition (Rev), December 1976



Printed in U.S.A.

**This document was set on DIGITAL's DECset-8000 computerized typesetting system.**

## CONTENTS

CONTENTS (Cont)

## ILLUSTRATIONS

## ILLUSTRATIONS (Cont)

## ILLUSTRATIONS (Cont)

## ILLUSTRATIONS (Cont)



## TABLES

# PREFACE

This manual contains three levels of EBox theory descriptions. The three levels are:

1. *Overview* – The overview identifies and introduces, in a simplified fashion, the basic hardware and firmware organization of the EBox. The major elements are presented without many details to provide a capsule view of the EBox structure.

2. *Functional Description* – This section describes the primary EBox function, which is to execute the KL10 instruction set and thus provide the specified functions, which generally include the following:

   Memory Reads and Writes
   Internal Operations
   EBus Operations

   The functional description is the most comprehensive part of the EBox Theory. Here the basic elements of the EBox are described in the context of how they implement the primary EBox function.

3. *Logic Description* – This section provides a detailed logic description of each of the board types that comprise the EBox. These descriptions are written to support the functional description. The logic description section is the most detailed part of the EBox. This material is presented to expand the functional description so that the information provided in the functional description can be directly related to the engineering logic diagrams.

# SECTION 1
# OVERVIEW

## 1.1 INTRODUCTION

The EBox is the instruction execution unit in the KL10 system. A central processor is formed when a memory interface unit (MBox), 10-11 interface unit (DTE), and PDP-11/40 processor are interfaced with the EBox. The MBox is the memory interface unit in the KL10 system to which the EBox directs its core memory requests. The PDP-11/40 is the front end processor that provides console functions and bootstrapping facilities and drives the standard PDP-11 peripherals. The DTE is the interface between the EBox and the PDP-11/40 console processor. The EBox communicates with the DTE, and hence the console processor, over a 36-bit data bus called the EBus, and uses three function lines (F00–F02), seven controller select lines (CS00–06), and two additional signal lines (Demand and Transfer) for arbitration and control of data transfers between the EBox and its internal and external devices. A pseudo-interface, which consists of a 23-bit address, 36-bit data, a number of request type qualifiers, and additional signals (including request and response), provides for arbitration and control of data transfers between the EBox and MBox.

The EBox contains the following (Figure 1-1):

1. A data path that consists of an Arithmetic Register (AR), Arithmetic Register Extension (ARX), Adder (AD), Adder Extension (ADX), various other registers, and a shift matrix.

2. An address path that consists of a 23-bit Program Counter (PC) and 23-bit Virtual Memory Address register (VMA).

3. Eight fast register blocks, each containing 16 × 36-bit words; each block of 16 registers is program-assignable.

4. A 13-bit Instruction Register (IR), which accepts the 9-bit operation code and 4-bit accumulator address.

5. Two somewhat autonomous control elements to provide control between the MBox and EBox, as well as the EBus and EBox. These are the MBox control and EBus control, respectively (Figure 1-1).

6. A control section storing and aiding the implementation of KL10 instructions.

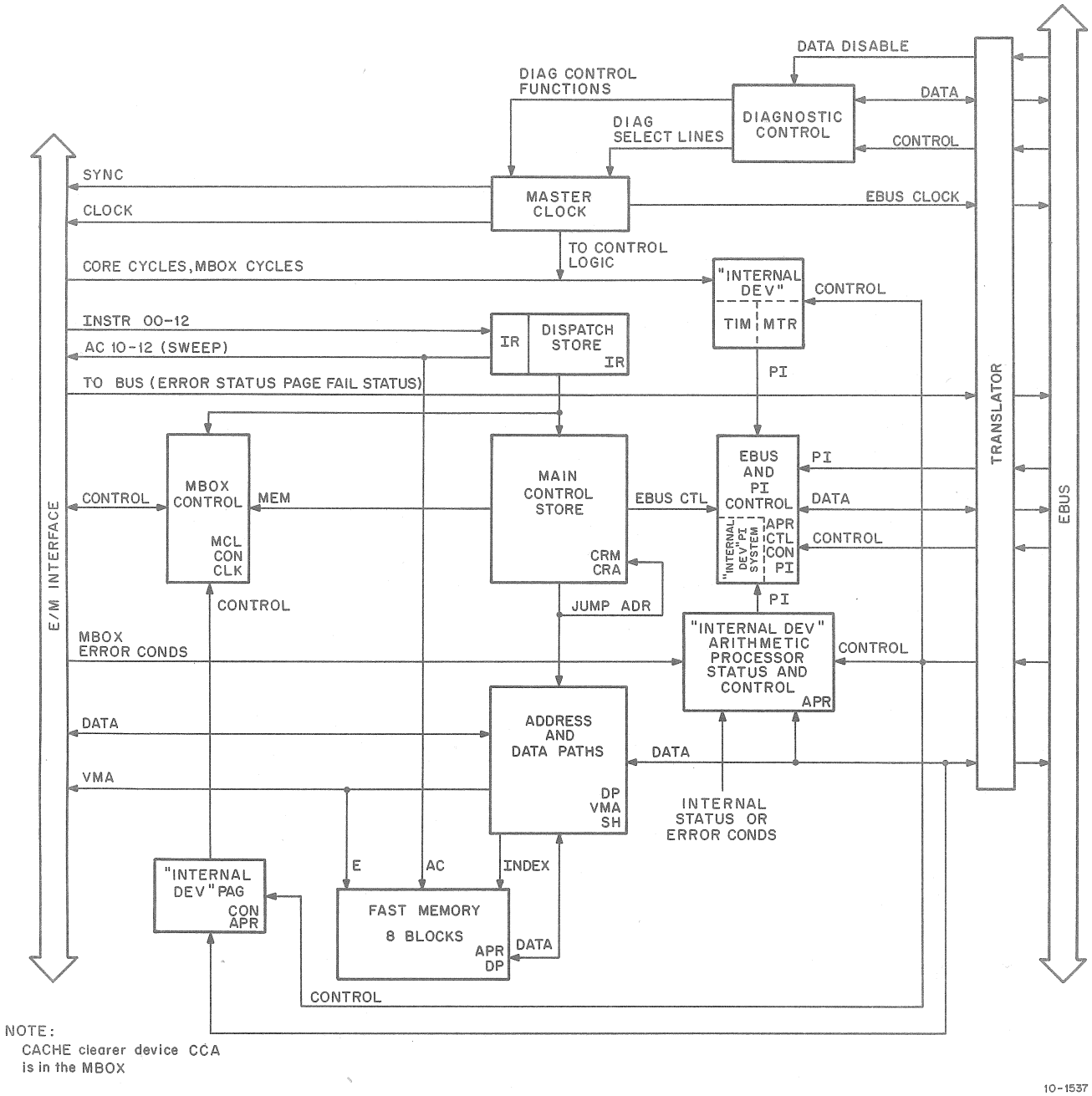



Figure 1-1 EBox Simplified Block Diagram

The control portion of the EBox comprises two Random Access Memories (RAMs). The first is called the Dispatch RAM (DRAM); it consists of storage for 512 decimal words, one word for each KL10 instruction. During instruction execution, the content of the DRAM word provides information about the type of memory references required by the executing instruction. It also provides an index into the main control programs contained in a second control memory called the Control RAM (CRAM). The CRAM consists of storage for 1280 microinstruction words that are structured into a sophisticated control program. The main program consists of a main loop and a number of subroutines or handlers. The structure provides for the implementation of a wide variety of internal register transfers, arithmetic and logical control, memory interface, and EBus control functions. The control program is generally referred to as the "microcode." Associated with the microcode and CRAM is a hardware pushdown stack, which enables the control program to make subroutine calls up to four levels deep, while performing various KL10 instructions. The basic machine control flow may be viewed as a pyramid, as shown in Figure 1-2. The instruction initially enters the IR consisting of two sections. One section, bits 0–8, holds the op code of the instruction, and the other, bits 9–12, holds the Accumulator (AC) address. During the instruction fetch cycle, the IR is unlatched via Load IR. During this time, it sets up with the op code. When the fetch cycle terminates, Load IR is removed and the IR latches.

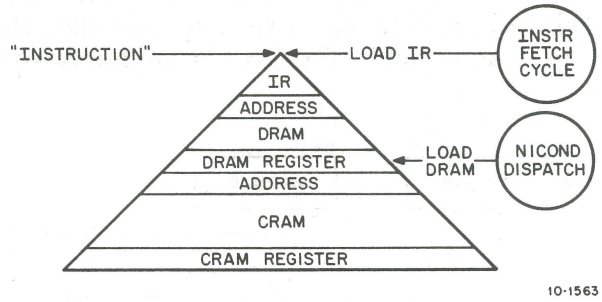



Figure 1-2 Control Pyramid

Because of the provision for prefetching, instructions may enter IR during the execution of the current instruction. This implies that, for these cases, the information provided by IR for the currently executing instruction must be somehow saved, while allowing IR to set up with the op code of the next instruction. This is accomplished by selecting an appropriate word from the DRAM.

The op code contained in the IR is used to address a corresponding DRAM word, and a Next Instruction Condition (NICOND) unlatches the DRAM register during this time. Encoded in the DRAM register fields (A, B, and J) is all information necessary for operand fetching, storing, and the microprogram executor jump address. Therefore, those instructions that prefetch an instruction do not require the IR to be reliable beyond the point of loading the DRAM register.

Input/output (I/O) instructions never prefetch. The device select code and operation for these instructions are specified directly in the IR. This must be made available to the microcode I/O handler during the instruction's execution cycle.

A special case in DRAM addressing is concerned with the JRST instruction. Because the JRST instruction encodes its JRST type in IR 9–12, these bits can be used directly as part of the DRAM word for this instruction. Normally, the DRAM address is as shown in Figure 1-3.

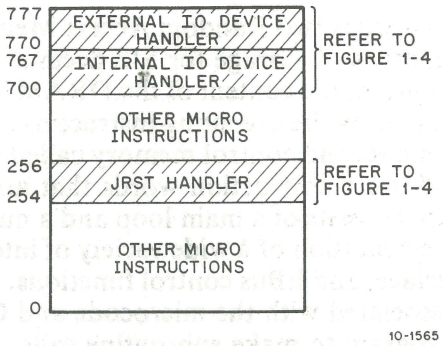



Figure 1-3 DRAM I/O, JRST

Figure 1-4 illustrates the organization of the DRAM. By sharing portions of the DRAM between even/odd instruction, the shared pieces become half the nonshared. Therefore, the A, B, and J7–10 portions consist of 10 × 512 words and the P, J4, J1–3 portions consist of 5 × 256 words. This saves essentially 5 × 256 words of DRAM storage. In addition, for JRST DRAM COMMON, bit 4 is made zero and DRAM J7–10 is replaced by IR 9–12, again yielding a savings. Here the savings is 5 × 16 words of DRAM storage. The areas allocated by the DRAM are indicated in Figure 1-3.

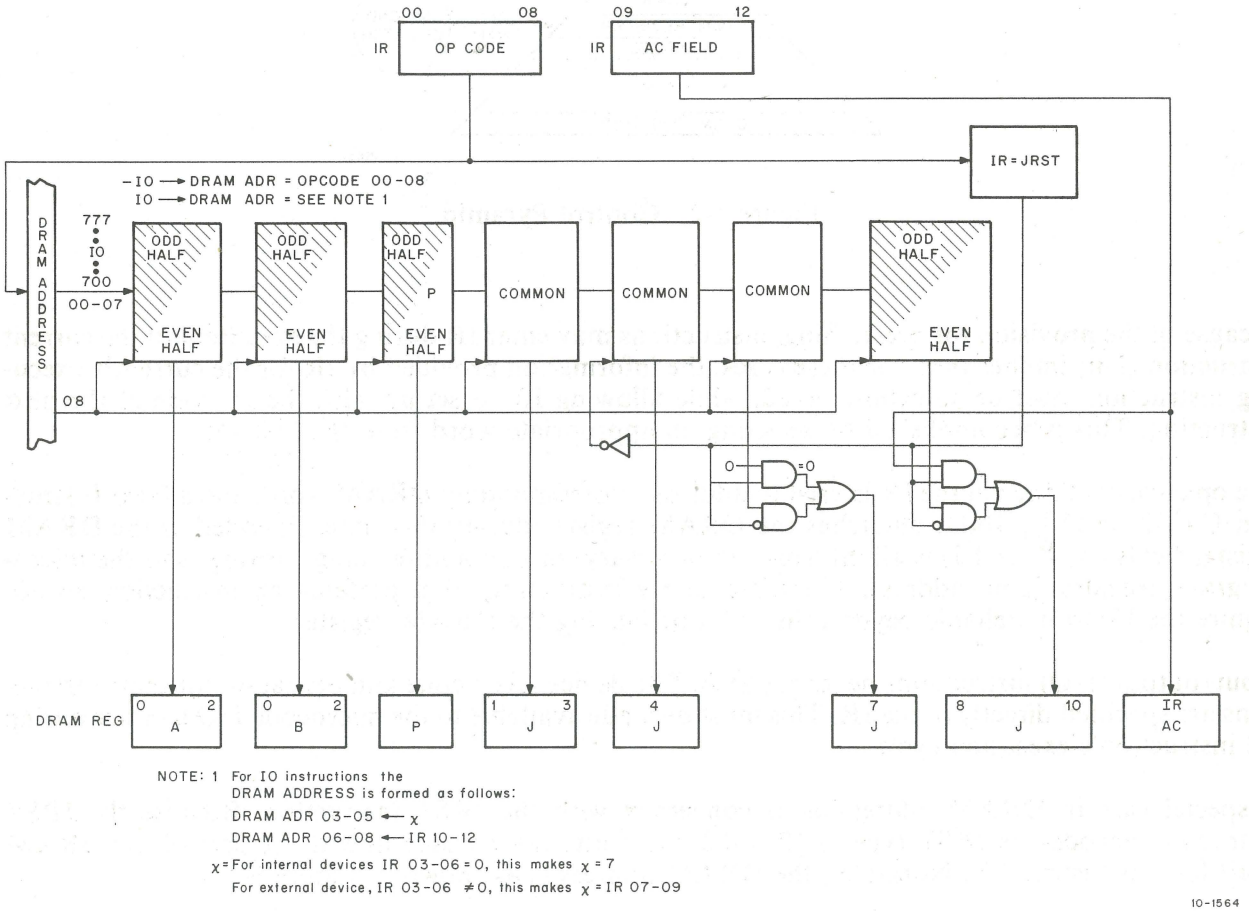



Figure 1-4 DRAM Organization


Included in the EBox is the master clock, which provides a time base for system operation. It distributes clock and sync pulses to the MBox, DTE, internal devices, system buses, and to the EBox itself. All operations in the KL-based system are synchronized to the master clock, which runs at 50 MHz. The master clock can be started, stopped, single stepped, and otherwise controlled by the console processor via the diagnostic control logic. This logic is distributed between the EBox and the DTE. Besides controlling the master clock, the diagnostic control logic provides a means for monitoring processor status and diagnostic registers in both the EBox and the MBox. The master clock is divided to supply a 25 MHz clock to the MBox and a 6.25 MHz clock to the EBus and SBus.

The EBox clock is variable and controlled by the microcode. The EBox and MBox are composed of emitter-coupled logic (ECL), while the DTE and external devices are composed of transistor-transistor logic (TTL). These two forms of logic are not directly compatible so the EBus is interfaced to the DTE, as well as external devices, via a special controllable logic-level shifter called the *Translator*. This is steered by the EBox and provides for both ECL to TTL transfer and TTL to ECL transfer.

The normal program flow may be interrupted through the use of one of eight interrupt control lines (PI0–7). This allows the servicing of peripheral devices and controllers, as well as internal devices, while executing the main program. The central processor contains six internal devices that are program selectable via KL10 I/O instructions. These devices are:

Priority Interrupt (PI)
Arithmetic Processor Status (APR)
Paging (PAG)
Cache Clearer (CCA)
Meter (MTR)
Timer (TIM)

Instructions, comprising a program, are maintained in core and/or fast memory. These instructions are fetched and executed by the EBox. The control program within the EBox evaluates fields of information that are part of the instruction currently being performed. Using various registers, fast memory, and adders, together with the VMA register and associated logic, the control program calculates an effective address; fetches any required operands; performs the instruction-dependent functions (e.g., those functions specified in the op code); stores the generated results; and fetches the next instruction. The logical data path between the instruction itself and the MBox is formed by the AR and ARX, together with various auxiliary registers, and the several adders contained on the Data Path Board (EDP). The IR receives the op code and accumulator address (IRAC) effectively for each instruction, while the ARX receives the entire instruction word consisting of the op code, accumulator address, Indirect bit, and Index register address, as well as the initial address supplied with the instruction referred to as the Y address. The control program contained within the DRAM passes through a well-defined "loop" consisting of microcode handlers, each of which performs a portion of the overall instruction execution. These correspond closely with the traditional processor cycles of Instruction, Address Calculation, Data Fetch, Execution, and Data Store with auxiliary cycles being Interrupt, Page Fault, and Trap.

## 1.2 BASIC FUNCTIONAL BLOCKS

The seven basic EBox functional blocks (Figures 1-5 and 1-6) are:

1. Instruction Register-Dispatch-Main Control Store
2. Fast Memory
3. Address Path
4. Data Path
5. Request and MBox Control
6. EBus and PI Control
7. EBox Control Logic

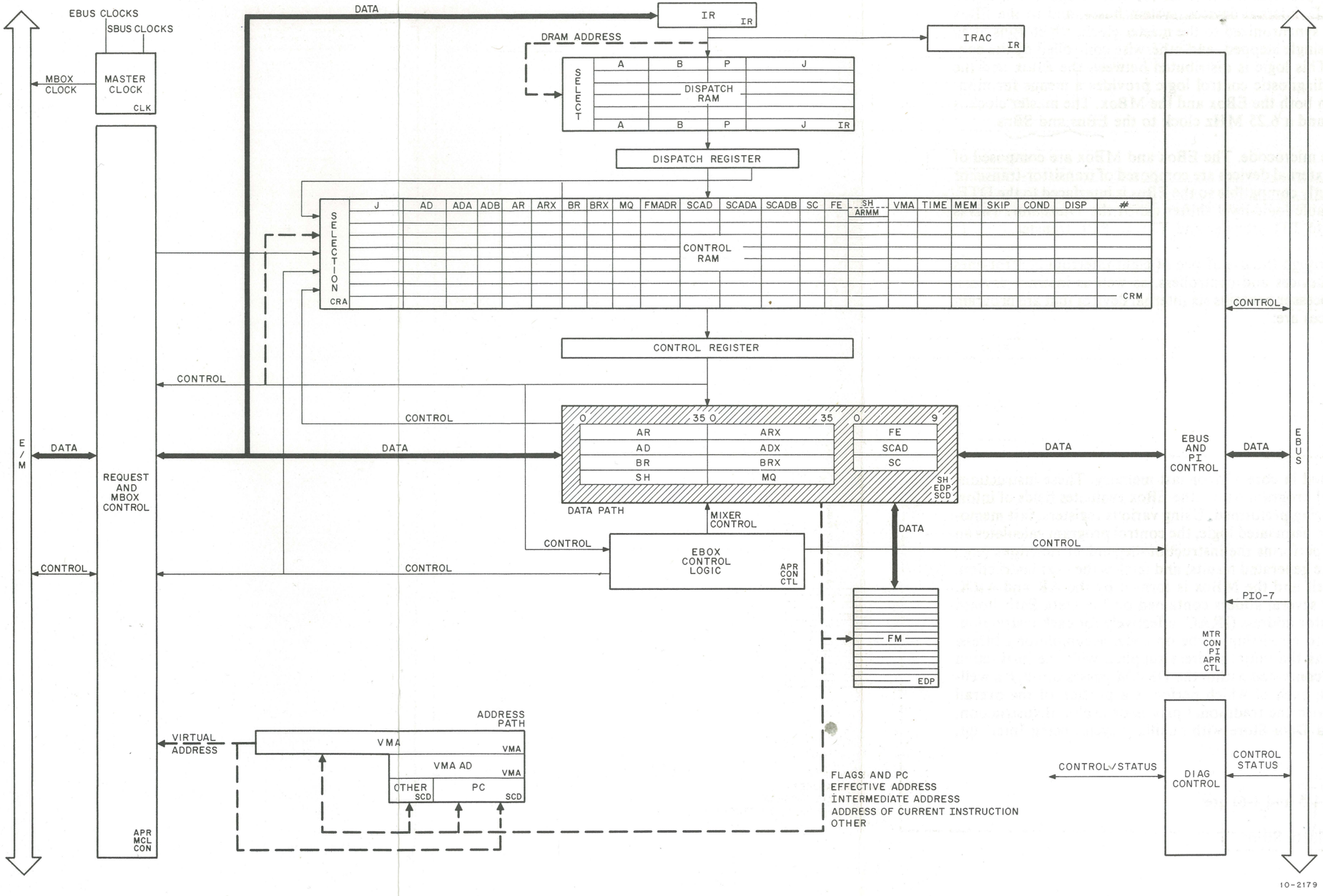
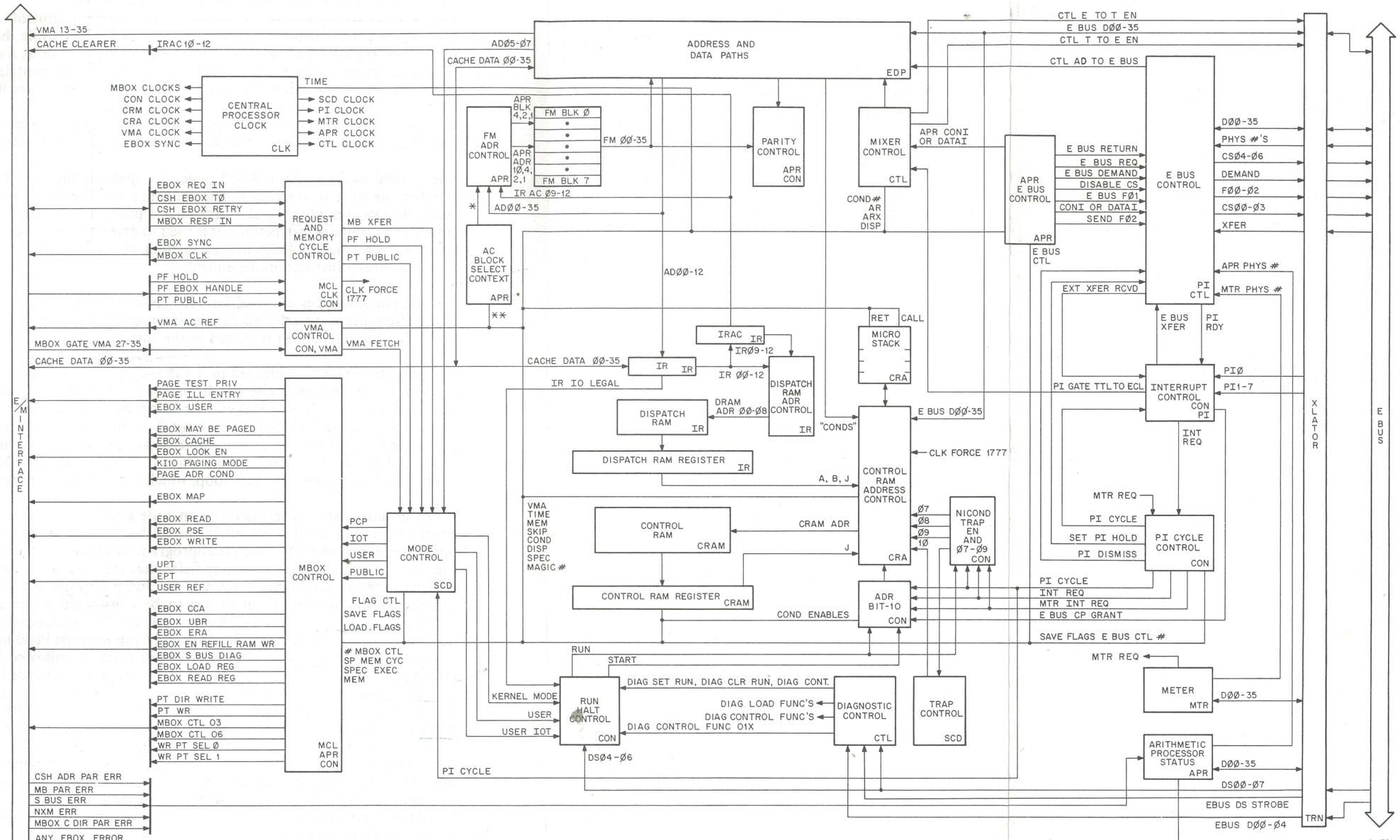



Figure 1-5 EBox RAM Structures, Interfaces, and Controls Block Diagram




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 1-6 EBox Overall Block Diagram

### 1.2.1 Instruction Register-Dispatch-Main Control Store

The Instruction register is the center of all processor control. Instructions are fetched from Main Memory or Fast Memory. The instruction enters ARX while the op code and AC address enter the Instruction register. The op code (bits 00–08) is used to address a word in the DRAM that is unique for each instruction in the KL10 instruction set. This word contains three fields of information and a parity bit. The Instruction, Dispatch, and Control formats are illustrated in Figure 1-7.

Because all instructions do not require the same types of data fetches, execution states, or data storage, they are handled uniquely for each instruction or, in some cases, for each class of instruction.

The A field (0–2) of the DRAM generally specifies the data fetch requirements, if any, as well as whether the next instruction in the sequence may be fetched early (prefetched). The B field (3–5) generally specifies where to store the results produced during execution; but in the case of Test, Skip, Jump, and Compare instructions, it is used to determine whether to skip the next sequential instruction or jump. The J field (14–23) is used to enter at the appropriate point in the Executor Microprogram and is generally instruction-dependent.

Specific microroutines are used for each class of instruction. Associated with the DRAM is a register that buffers the word selected for the instruction currently being performed. This register is loaded soon after the instruction is placed in the Instruction register.

The microprogram is contained in a 1280 × 75-bit RAM called the CRAM. Both the DRAM and CRAM are loaded when the KL10 system is powered up. This is accomplished by the PDP-11/40 processor via the DTE and makes use of diagnostic control logic within the EBox. Associated with the CRAM is a register that buffers each word or microinstruction read from the CRAM. This register is called the Control register and its contents are decoded to provide overall control of the seven major functional blocks described in Subsection 1.2. In addition, the Microprogram is structured into what might be called a *main loop*. This loop, which is passed through regularly, is illustrated in Figure 1-8.

When an instruction is fetched, the op code and accumulator address are placed in the IR and the entire instruction word is placed in one of the Data Path registers called the ARX. Movement from one routine (or *handler*) in the microprogram to another is made via a microcode Dispatch function. The Control register contains many fields that are used for different types of control. Two such fields that are used to control this movement are Jump Address and Dispatch Field. The Dispatch function enables various hardware conditions to be considered when an instruction has been fetched and enables the most important condition to prevail. Two such conditions that are illustrated in Figure 1-8 are Priority Interrupt Request Pending and Trap Request Pending. The hardware is arranged in such a fashion that priority interrupts have highest priority, followed by traps; the current instruction has lowest priority. The dispatch that takes the microprogram to the Process Instruction Block is called the NICOND and is given after a Fetch request for the next instruction. If no priority interrupts or traps are pending, the microprogram enters the next block to calculate the effective address. Here the dispatch is called Effective Address Modification (EAMOD) and enables the hardware to sample indirect field bit 13 of ARX together with indexing field bits 14–17. The KL10 instruction specification allows multilevel indirect addressing with indexing at each level where indexing, if specified, is performed first. The microprogram evaluates bits 14–17; if nonzero, the contents of bits 14–17 are used to access the specified 36-bit Index register. The right-most half of the Index register (bits 18–35) is added to the Y field of the instruction word (bits 18–35); the right-most 18 bits of this result are used in the next step of the effective address calculation. Simultaneously, the state of ARX bit 13 is tested and, if equal to a 1, a memory request is generated to the MBox control portion of the EBox. Each time a word is fetched in this fashion and has bit 13 equal to 1, the same sequence occurs until finally a word is fetched with bit 13 equal to 0. Then, one more level of indexing may be specified and the result is the effective address. At this time, the A READ dispatch is given and the A field of the DRAM is evaluated to enable a required operand to be fetched; if specified, a prefetch is also set up at this time. Table 1-1 lists the A field codes and the specific function required.

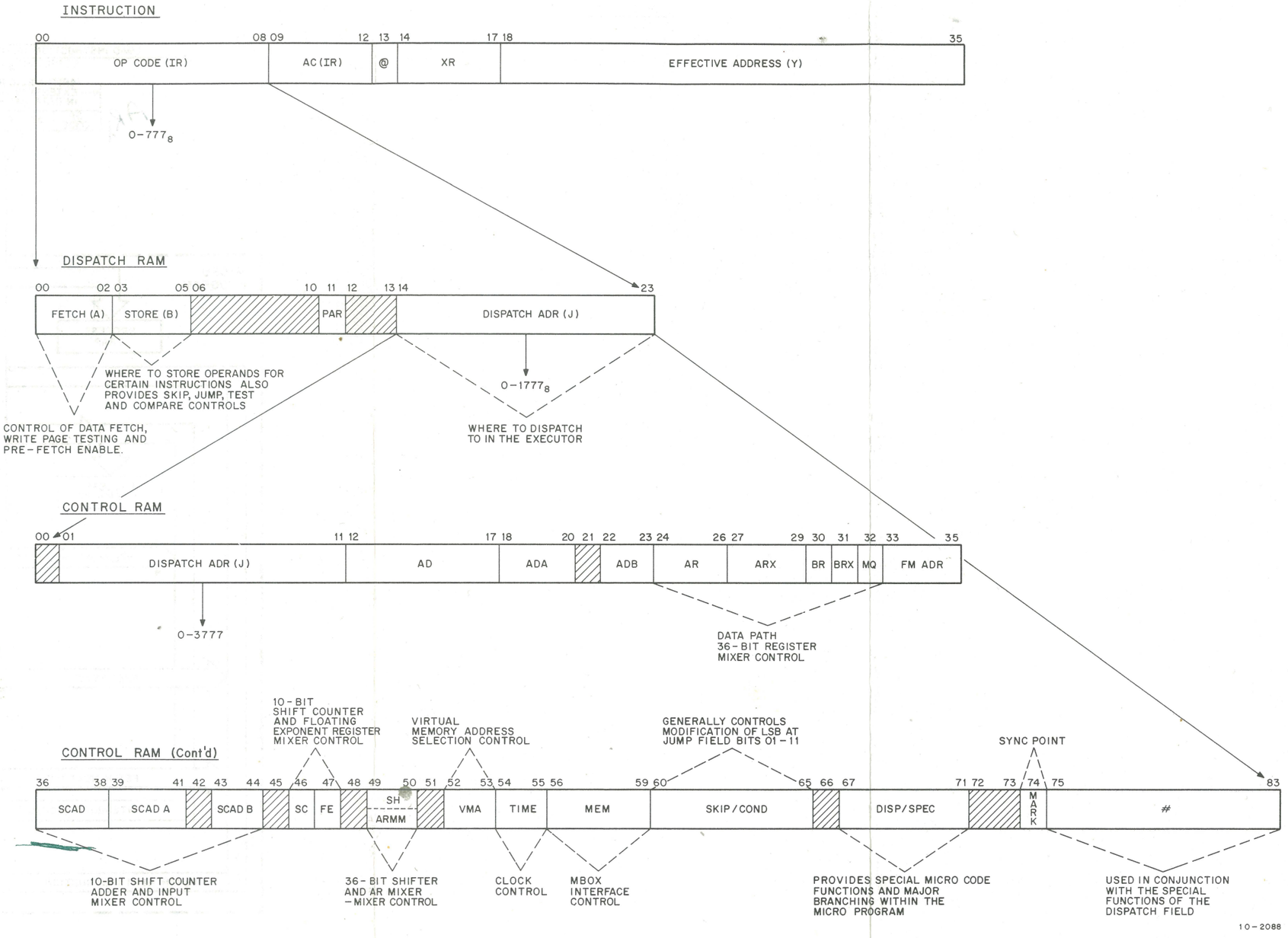



Figure 1-7 Instruction, Dispatch, and Control Formats

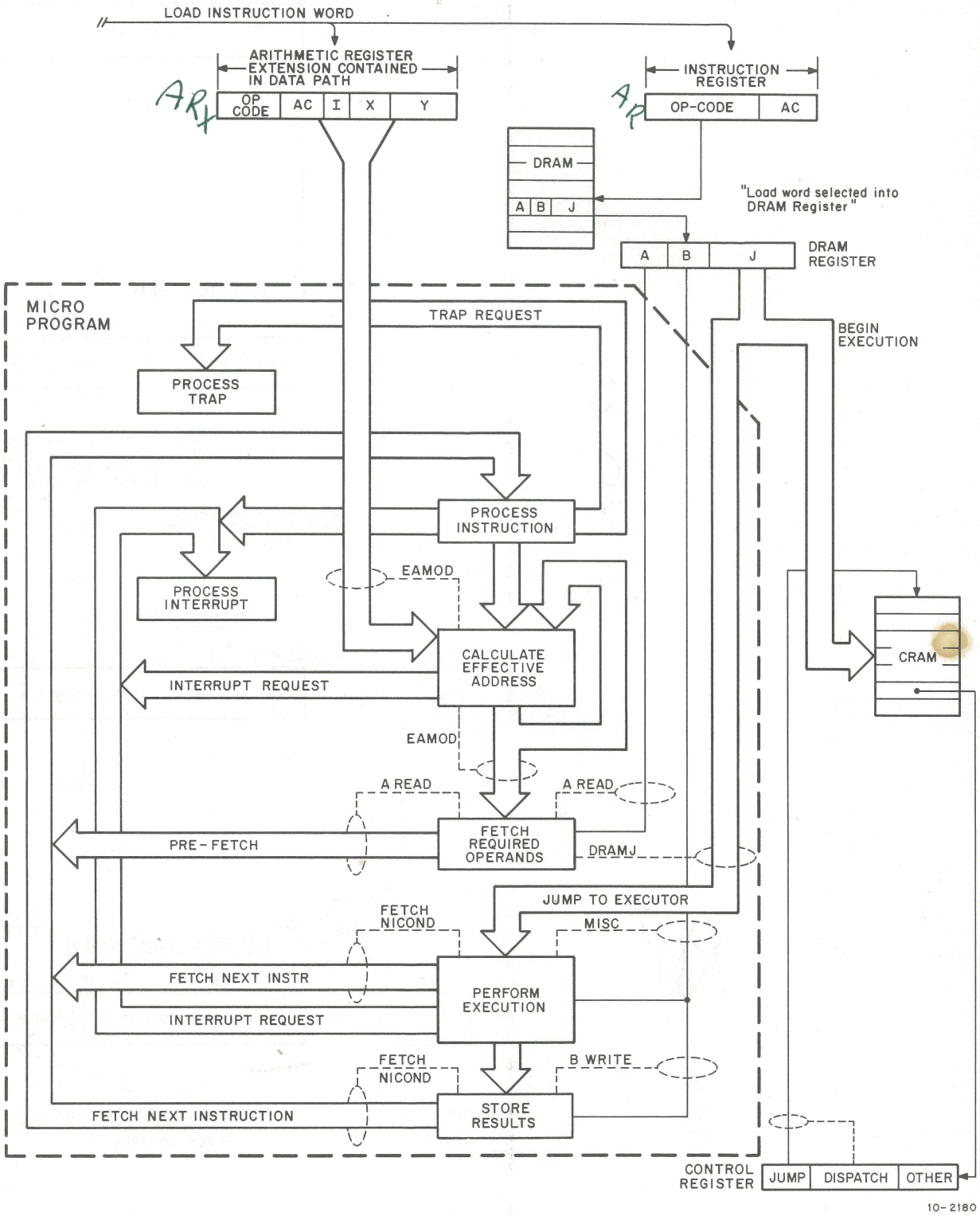



Figure 1-8 Microprogram Main Loop

**Table 1-1 AREAD**

| DRAMA 3-Bit Code | MEM/AREAD | DISP/AREAD |
|---|---|---|
| 0 | Immediate class instruction; prefetch disabled. | DRAM J DISP |
| 1 | Immediate class instruction; prefetch enabled. | DRAM J DISP |
| 2 | Not used | 42 |
| 3 | Write-check the paging; prefetch disabled. | 43 |
| 4 | Data read required; prefetch disabled.* | 44 |
| 5 | Data read required; prefetch enabled.* | 45 |
| 6 | Data read required as separate cycle; also write-check the paging; prefetch disabled. | 46 |
| 7 | Data read modify write required; prefetch disabled. | 47 |

*These two cases are distinguished only by dispatching to different microcode locations. The microcode entered at location 45 prefetches, that at 44 does not.

The next block is entered to perform the specific execution function or functions for the particular instruction by the microprogram giving a DRAM J dispatch. Remember that each instruction has its own DRAM word with a unique Jump field specifying where to go for that instruction's execution. The execution is very complex and is described in detail elsewhere in this manual. Basically, it performs all required arithmetic, logical, or other types of functions required, and may also, in some cases, fetch additional operands as required. Upon completion of this portion of the microprogram, the next instruction may be started, provided that no data storage is required. If storage is required, two basic cases must be considered. Those instructions that do not know where to store their data utilize the B field of the DRAM as an index into the final block to store results. After storing results, the next instruction is fetched and a NICOND dispatch is issued. Instructions that know where to go specifically in order to store their data do so by jumping to a specific location in the microprogram, but may use the B field of the DRAM to decode additional types of memory requests as required. This completes the basic loop.

### 1.2.2 Fast Memory

An instruction word has only one 18-bit address field for addressing any location throughout all of memory. Most instructions, however, have two 4-bit fields for addressing the first 16 locations of memory. These 16 locations consist of a set of 16 general-purpose, high-speed integrated circuit registers grouped locally into eight physical blocks, which are software-assignable by block. Non-I/O instructions have an accumulator address field that can address one of these 16 locations as an accumulator. Every instruction has a 4-bit Index register address field that can address 15 of these locations for use as Index registers in modifying the 18-bit memory address. (A zero Index register address specifies no indexing.) The factor that determines whether one of the first 16 locations in memory is an accumulator or an Index register is not the information it contains, nor how its contents are used, but rather how the location is addressed. The eight blocks of fast memory are contained physically on the data path board within the EBox. This allows much quicker access to these locations whether they are addressed as accumulators, index registers, or ordinary memory locations. They can even be addressed from the program counter, gaining faster execution for a short but often repeated subroutine. Of the eight blocks contained within the EBox, block 7 is permanently assigned to the microcode. Referring to Figure 1-9, the monitor uses an assigned AC block in the same way that a user program described in the following paragraphs would. The microcode uses the assigned AC block when executing complex instruction algorithms. From the remaining blocks (0–6), two can be assigned under program control (DATAO PAG) as the current and previous context AC blocks. The current context AC block is used by the user program for indexing in effective address calculation and for general storage as specified by the AC field of the instruction and/or by the effective virtual address (location 0–17).

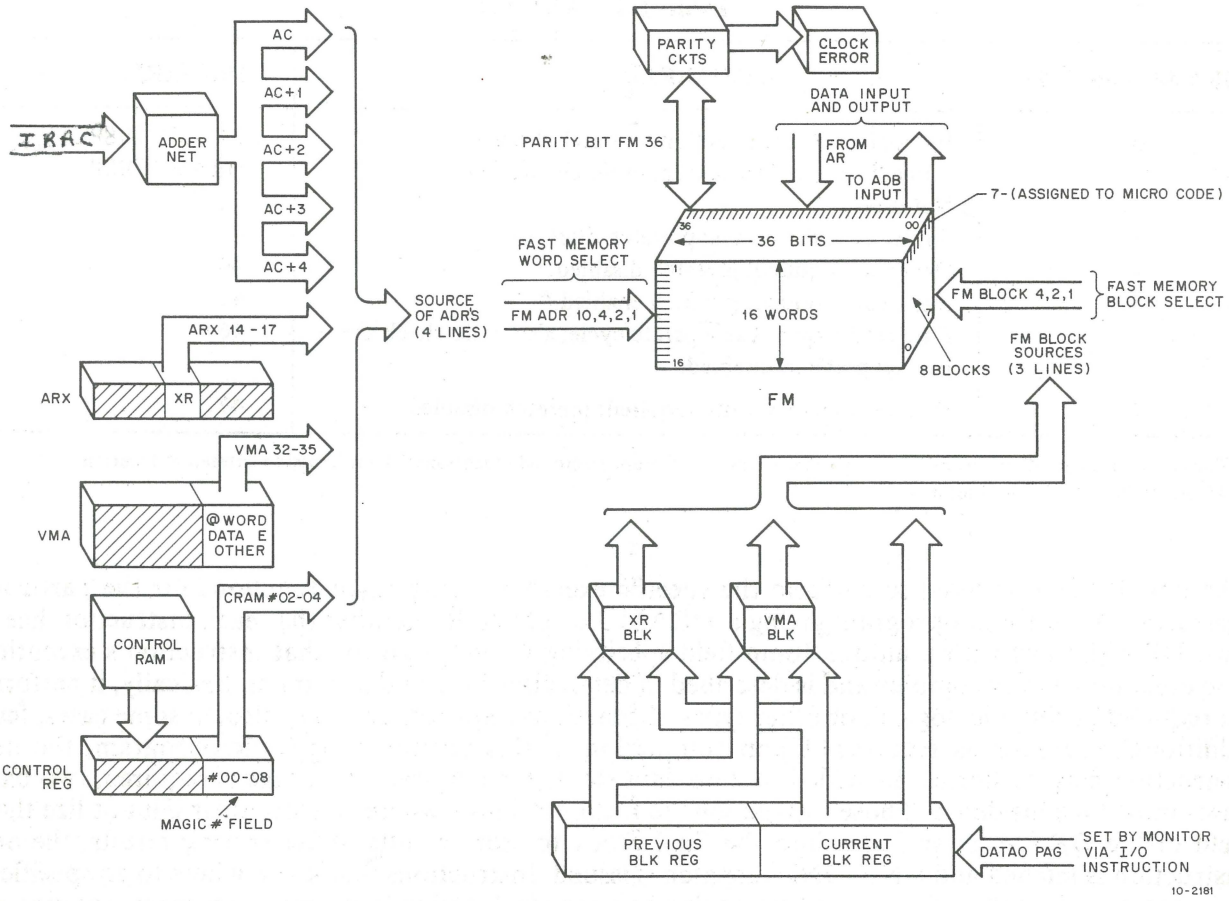



Figure 1-9 Basic Fast Memory Structure

The previous context AC block is used by the monitor to allow the monitor to reference the previous user's address space to pass arguments, data, or status information between the user program and the monitor. This is normally done when the user program executes a monitor call for some type of service.

The microprogram running within the CRAM may select eight possible sources to be the word address for fast memory; these sources are indicated on the figure as follows:

AC
AC+1
AC+2
AC+3
AC+4
ARX 14 - 17
VMA 32 - 35
CRAM 05 - 08

The selection of the appropriate source is a function of the 3-bit microinstruction FM ADR FIELD. The block to be used is selected by the same FM ADR FIELD and corresponds to three block sources as indicated in Table 1-2.

**Table 1-2 FM Selection**

| FM ADR Field | FM ADR 10, 4, 2, 1 Source | FM ADR BLK 4, 2, 1 Source |
|---|---|---|
| 0 | AC | Current Block |
| 1 | AC+1 | Current Block |
| 2 | ARX 14–17 | XR Block* |
| 3 | VMA 32–35 | VMA Block* |
| 4 | AC+2 | Current Block |
| 5 | AC+3 | Current Block |
| 6 | AC+4 | Current Block |
| 7 | CRAM #05–08 | CRAM #02–04 |

*These may select either the current or previous AC block address.

The selection of AC, AC+1, AC+2, and AC+3 is a function of the class of KL10 instruction being performed. All non I/O instructions specify an accumulator address in the instruction word, bits 9–12.

The logical instructions – Logical Shift Combined (LSHC) and Rotate Combined (ROTC) – specify the use of both AC and AC+1. Similarly, the fixed-point arithmetic instructions Multiply (MUL), Divide (DIV), and Arithmetic Shift Combined (ASHC) specify use of AC and AC+1. The double integer arithmetic instructions Double Add (DADD), Double Subtract (DSUB), Double Multiply (DMUL), and Double Divide (DDIV) specify use of AC, AC+1, AC+2, and AC+3. As pointed out previously, the microprogram is permanently assigned AC block 7 for its own use. During extended instruction processing, the microprogram addresses words in AC block 7 by using magic number field bits 05–08, while selecting AC block 7 with magic number field bits 02–04. These ACs provide temporary working storage for the microprogram. Similarly, the microprogram addresses AC+4 by combining the AC address taken from IR AC9–2 with bits of the magic number field in an adder network to produce AC+4

For selection of AC, AC+1, AC+2, AC+3, or AC+4, the current block is always used. Whenever a main memory reference is made, the microcode references the fast memory location given by VMA 32–35, enabling the hardware to switch the reference to fast memory, if necessary. When the instruction's effective address is calculated, the microprogram allows the specified Index register to be addressed in fast memory by enabling ARX 14–17 to address the word. For both cases, i.e., VMA 32–35 or ARX 14–17 addressing fast memory, the AC block may be either the current block or the previous block, but is a function of the context of the instruction.

If an executive XCT is performed in response to a user's call (MUUO), then the previous physical block and current physical block will be made to be different unless the operating system saves the user's current AC block and then wishes to use the same block once again, which is unlikely. As an example, assume the user is assigned AC block 1; his previous AC block would initially be 1 also. If the user then performs an MUUO, the executive subroutine entered may safely load the current AC block with some other block number and the previous user block will remain unchanged. The operating system may perform an executive XCT utilizing the user's previous block and an AC within that block. The hardware enables the selection at the time of the previous block for indexing. In addition, the operating system may also reference the user's AC block (previous context block 1 in the example) from the VMA. In this case (referring to Figure 1-9), mixer selection 3 is enabled and the microword FM ADR field specifies VMA.

During normal instruction processing, if VMA bits 13–31 are equal to 0, the address in bits 32–35 is an FM address.

Some examples using the current AC block in various selections are given below. Assume the following is performed by the operating system:

```
EXAC = 1                    ;This will default to Exec block
                            ;#0, AC#1
HRLEI EXAC, 102200          ;Load bit, current Blk#2
                            ;Previous Blk#2.
      .
      .
DATAO PAG, EXAC             ;Load the current Blk# = 2, and the
                            ;Previous Blk# = 2.
      .
      .
JRST 2, @ USRPCWD           ;Pick up user mode, flags, and
                            ;turn on user.
      .
      .
```

The following codes are for the user:

```
AC1 =1                      ;This will be in Blk#2
AC2 =2                      ;This will be in Blk#2
MOVEI AC1, 777777           ;The word 0,777777 to AC1
HRLEM AC1, AC2              ;The word 777777,777777 to AC2
SETCMM, AC1                 ;The one's comp of the word in AC1 to AC2
                            ;which is equal to 777777,0
PUSH AC1, 3(AC2)            ;This instruction attempts to
                            ;push the contents of AC2 into
                            ;location AC1. It will cause PDOVL
                            ;and this generates TRAP#2.
```

In the example, the symbol EXAC is defined as the number 1. Assume, for this example, that EXAC is referenced as an AC accumulator in executive block 0. The first use of EXAC is in the instruction HRLEI EXAC, 102200. This instruction takes the number in the Y field of the instruction, which, in this example, is the effective address, and places it in the left half of EXAC (which is executive AC1), with the sign of the right half of the word 0,102200 extended in the right half of EXAC. In this instruction, the current AC is referenced in bits 9–12 of the instruction, and the mixer selection is 0. To load the user AC blocks, both current and previous, it is necessary now for the executive to perform the indicated DATAO PAG instruction.

The left half-word in EXAC contains the necessary bits to enable the loading of the current and previous blocks (EBus bits 6, 7, and 8 for the current block and bits 9, 10, and 11 for the previous block). Next, we assume location USRPCWD contains the appropriate bit configuration to start the user for whom we loaded the AC block numbers. The instruction JRST 2, @ USRPCWD makes an indirect reference to location USRPCWD. The resulting word will then contain the user mode bit (bit 5), possibly the public mode bit (bit 7), any other relevant flags in the remaining left half-word, and the user virtual address in the right half-word. The user has defined the symbols AC1 and AC2 as having the values 1 and 2, respectively. As indicated in this example, these correspond to AC1 and AC2 in block number 2. The first instruction performed by the user is MOVEI AC1, 777777, which places the number 0,777777 in accumulator 1. On the next instruction, the word in AC1 as addressed by instruction field bits 9–12 is read out. Remember that during the effective address calculation, the AC number is loaded from ARX 9–12 into register AC in the EBox.

The FM ADR field of the microword that is performing the fast memory reference will specify a field function of 0, which will select the current block as well as register AC which, as pointed out, contains the value of AC 1 (1). The operation, specified by the instruction, is to take the right half of AC1 and store it into the left half of AC2 with its sign extended into the other half-word. Because the sign of the right half-word in AC1 is negative, the result is the word 777777,777777. Notice that we must now reference AC block 2, location 2, by using VMA bits 32–35. This operation is specified with a different microcontrol word and at a different time than the fetch of the word from AC1. Actually, the content of AC1 is obtained by performing a READ; the word 777777,777777 is stored into AC2 on B WRITE. The next instruction, SETCMM, reads the word from AC1 as addressed by VMA, takes the 1's complement of it, and stores the result (777777,0) back into AC1 again as addressed using VMA. Thus, the same address is used for read as well as write. Finally, the PUSH instruction performs an indexing function using the current AC block. The number 3, which is the Y field in the instruction, is added to the number contained in AC2, as addressed in the example, using the mixer selection of 2 (XR).

Thus, the address is taken from ARX 14–17 during the effective address calculation. The number 3 is added to the number 777777,777777 and the right half of the result (2) is used as the effective address. Then the instruction attempts to push the number 777777,777777 onto the stack as addressed by the updated right half of the word in AC1. The updating takes place first. The word is fetched from AC1 using the current block and the address in the EBox register AC. Then, this word has +1 added to both halves and, if the left word is such that the addition causes a carry from bit 0, a pushdown list overflow trap occurs.

### 1.2.3 Address Path

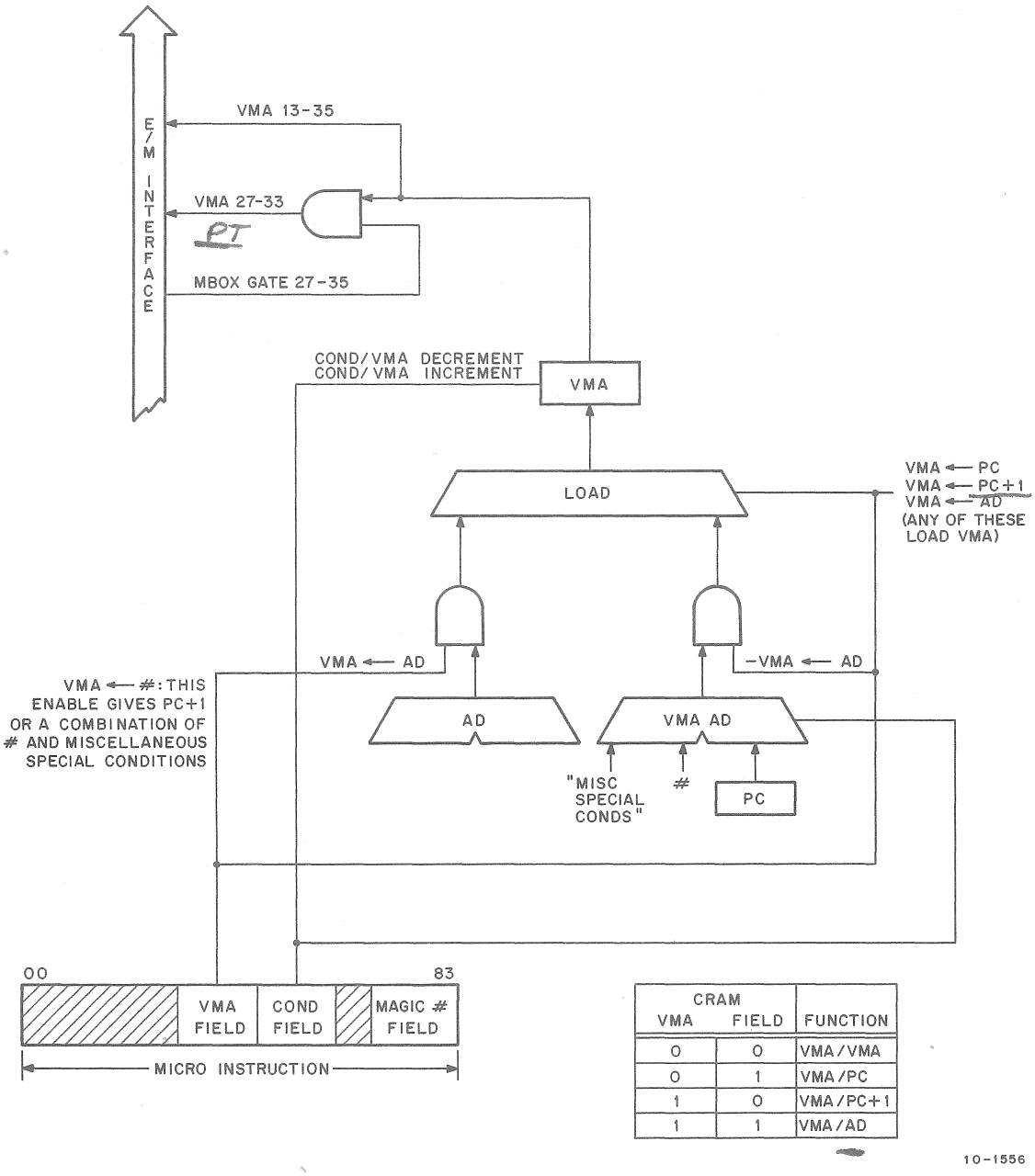
The EBox performs a program by executing instructions retrieved from locations addressed by the PC, a 23-bit register contained in the EBox data path. At the beginning of each instruction, PC is incremented by one so that it normally contains an address one greater than the current instruction. Sequential program flow is altered by changing the contents of PC, either by incrementing it an extra time as in a Skip instruction, or by replacing its contents with the value specified by a Jump instruction. Instructions may be fetched either from core memory, which is external to the EBox, or from fast memory, which is internal to the EBox.

Generally, instructions provide at least two operand addresses to the EBox. One address is that of an internal accumulator, and is addressed by bits 9–12 of the instruction. The other address, also supplied by the instruction, may be used to address either core or fast memory and is contained in bits 13–35 of the instruction word. This is a composite address, such that bit 13 specifies the type of addressing, i.e., direct or indirect; bits 14–17 specify an index register for use in address modification; and bits 18–35 address a virtual memory location.

Because the PC is used to keep track of where in the program the EBox is executing instructions, an additional register is provided to handle addresses that can be generated during effective address calculations, during operand reads and/or writes, and at other times. This 23-bit register, also contained in the EBox data path, is called the VMA register.

Figure 1-10 illustrates the basic path connections from the PC and AD. A control field consisting of two bits in the microinstruction is provided to select the source of input to VMA. This field is called the "VMA field." In addition, two other fields are used to provide alternate input to the VMA as well as provide the ability to increment or decrement the VMA directly. These fields, also a part of the microinstruction word, are called the "condition field" and "magic number field."

Referring to Figure 1-10, to load the VMA from AD, the microinstruction VMA field is coded symbolically as "VMA/AD." The field format is indicated at the lower right of the figure. The AD is enabled into the input of the VMA register by the function VMA ← AD, and the input to VMA is enabled for any of the following functions: VMA ← PC, VMA ← PC+1, or VMA ← AD.




| CRAM VMA | CRAM FIELD | FUNCTION |
|---|---|---|
| 0 | 0 | VMA/VMA |
| 0 | 1 | VMA/PC |
| 1 | 0 | VMA/PC+1 |
| 1 | 1 | VMA/AD |

Figure 1-10 VMA Structure Simplified

Similarly, to update the PC (Figure 1-11), the microinstruction VMA field is coded to specify the function "VMA/PC+1." This disables VMA ← AD, and so the VMA defaults to VMA ← VMA AD as input. At this time, the COND field must not be VMA ← # if it is desired to enable the VMA AD to implement the function A+B. The A input to VMA AD is from PC bits 13–35. The B input is forced to +1 if – PC+1 INH is true, and if the VMA field specifies the function "VMA/PC+1." The input to VMA is enabled for PC+1 as well. Certain instructions such as JUMPXX, AOJXX, or SOJXX conditionally load VMA with either E or PC+1. Instructions such as SKIPXX, TEST, CAIXX, and CAMXX conditionally skip an instruction, so VMA may be loaded with either PC+1 or PC+2. In general, the VMA is loaded with PC+1 for most instructions by the microinstruction following the effective address calculation (assuming no special instructions and not loading VMA from AD). Those instructions that perform an instruction prefetch will enable the VMA from PC+1 on the A READ dispatch function. This function is used to trigger the Fetch cycle and, conditionally, the microprogram enters the wait state until the operand arrives when the data is fetched from the MBox. If this is the case, and the prefetch condition is true, the VMA input will be PC+1; when the MBox responds, restarting the EBox clock, the VMA loads with PC+1.

Instructions such as MOVEI, ADDI, SUBI, and HXXXI fetch no operands during A READ; instead, they use the effective address as data. These instructions prefetch the next instruction and the microprogram does not enter the wait state at all. Thus, the VMA is loaded with PC+1 as the microprogram passes through A READ dispatch.

The function VMA+1 is used by such instructions as double MOVE, JSA, and JSR. Here, the microinstruction VMA field is not used, but the function VMA+1 is enabled by the condition field coded as COND/VMA INC. The VMA register itself contains logic for the incrementation. Similarly, the function VMA – 1 is used by byte and ADJBP instructions in cases where a word must be fetched from E – 1. Once again, the VMA field is not used; instead, the condition field is coded COND/VMA DEC. This is also a VMA built-in function.

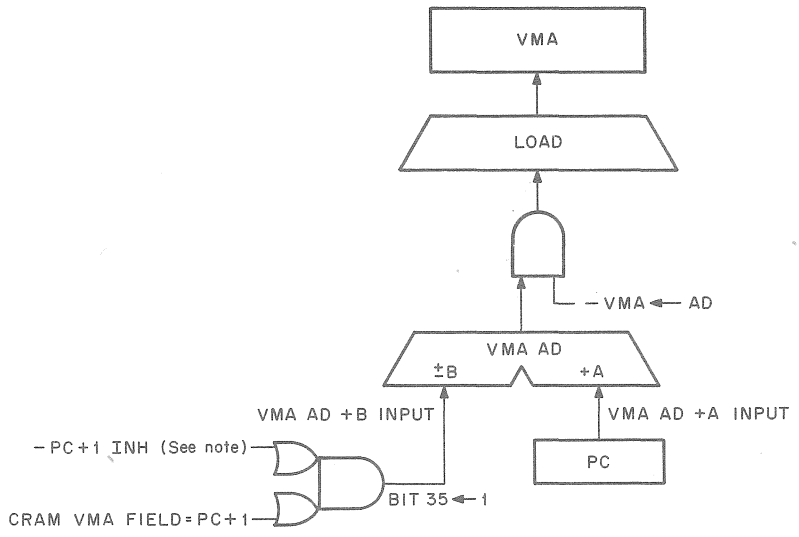



Figure 1-11 PC + 1 Function

The special number, magic number, and miscellaneous conditions shown on VMA AD in Figure 1-10 are used during LUUO, MUUO, and PI handling to generate a range of special addresses to reference the user or executive process tables in memory. During these types of functions, the VMA AD is controlled by VMA #, which enables the Boolean function "B." MVA AD B input bits 27–35 are manipulated, while bits 18–26 are cleared; this allows for the generation of process table word addresses in the range of 000–777. Note, however, that addresses in the range of 40–510 only are currently generated by hardware.

### 1.2.4 Request and MBox Control

In general, most of the EBox memory request type operations are controlled by the 4-bit MEM field in the microinstruction (Figure 1-12). This may be used alone or with the DRAM A or B field values for most reads and writes. In addition, the 5-bit special microinstruction field (SPEC) can specify a function SP MEM CYCLE, which is sometimes used with the magic number field (a 9-bit microinstruction field) to modify MBox read and write operations, e.g., for MUUO or LUUO. Note that the basic MBox activity involves a request, a virtual address, and MBox qualifiers consisting of a multitude of control signals that qualify the type of request being made. This is followed by:

1. A response from the MBox with the data when the request is successful,
2. PF HOLD followed by MBox response IN and no data on a page fault, or
3. MBox response IN with data followed by MB PAR ERR, for an MB parity error condition.

Additional conditions are covered elsewhere in this manual.

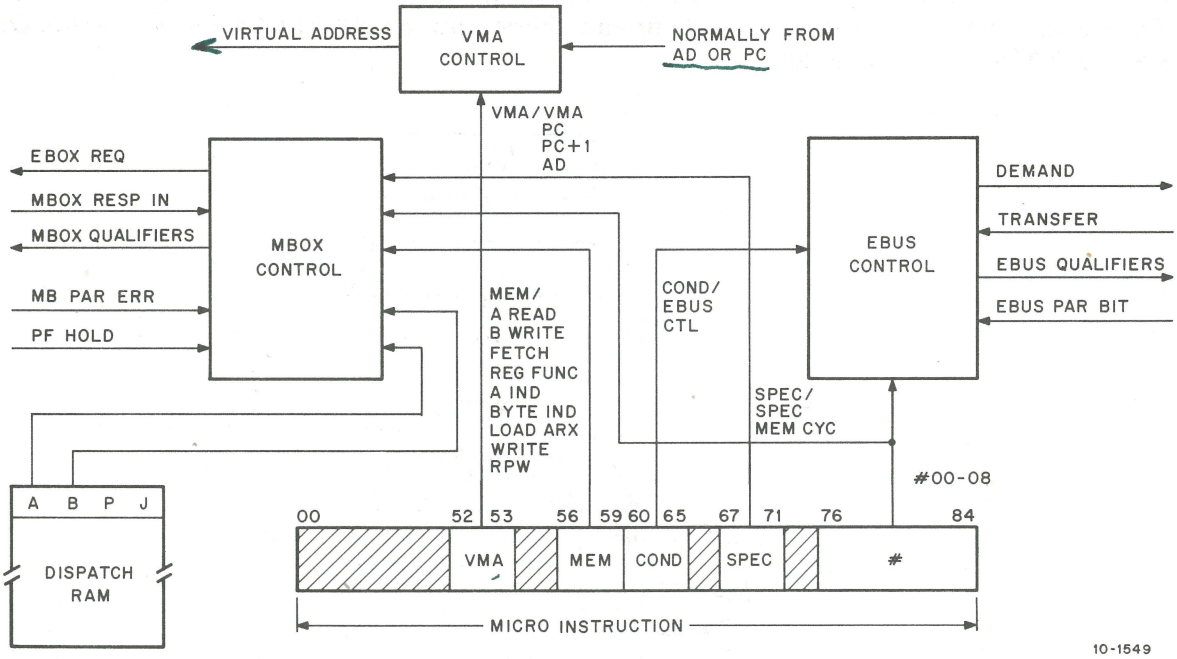



Figure 1-12 MBox-VMA-EBUS Control Simplified

**1.2.4.1 KI Style Paging** – For each MBox request involving a virtual address translation, the MBox must verify that the virtual address is legal. In general, the physical page must be in core for a read and be writable for a write. In addition, the address space to which it belongs must correspond to that being referenced, i.e., a public program cannot read or write into a private address space.

Two styles of paging are implemented; the first is patterned after the KI10 processor's memory management scheme; the second after the KL10 style.

The MBox contains two base registers that can be loaded via the EBox. These registers are used as the base address of core page tables during virtual memory address translation. The base registers are 13 bits wide. The User Base Register (UBR) is loaded by performing a privileged I/O instruction (DATAO PAG); similarly, the Executive Base Register (EBR) is loaded by performing another privileged I/O Instruction (CONO PAG). These registers are normally loaded by the operating system at predetermined times. For example, the EBR is normally loaded once when the operating system is bootstrapped. Also, each time a user is started in a normal multiprogramming environment, i.e., more than one user program resident in core memory, the UBR is reloaded to point at the User Page Table.

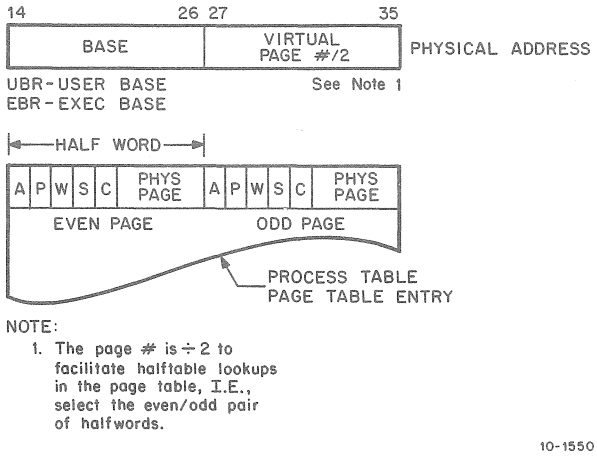



Figure 1-13 Page Table Access

Each time the EBox makes a memory reference to the MBox (Figure 1-13), the MBox evaluates the virtual address. The details of this operation can be found in the MBox chapter of the *KL10 Theory of Operation Manual.* Basically, the page number supplied in VMA 18–26 is used as an index into a hardware page table within the MBox. The MBox looks for the referenced page in this table. If it is not found, the MBox uses the appropriate base register (UBR or EBR) with the virtual page number supplied in VMA to form a 22-bit physical memory address, as indicated.

The appropriate entry is obtained and then written by the MBox into a hardware page table within the MBox. (Actually, eight half-word entries are fetched at a time, but for this level of explanation, only one is considered.)

The five bits A, P, W, S, C (generally called use bits or page descriptor bits) are tested against the qualifiers sent by the EBox during the reference. Then the MBox, using the physical address, looks in the cache for the word requested. If it does not find the word, it concatenates the physical page address (Figure 1-14) with the virtual word address provided in VMA bits 27–35 and makes a second physical memory reference. This address is indicated in Figure 1-15.

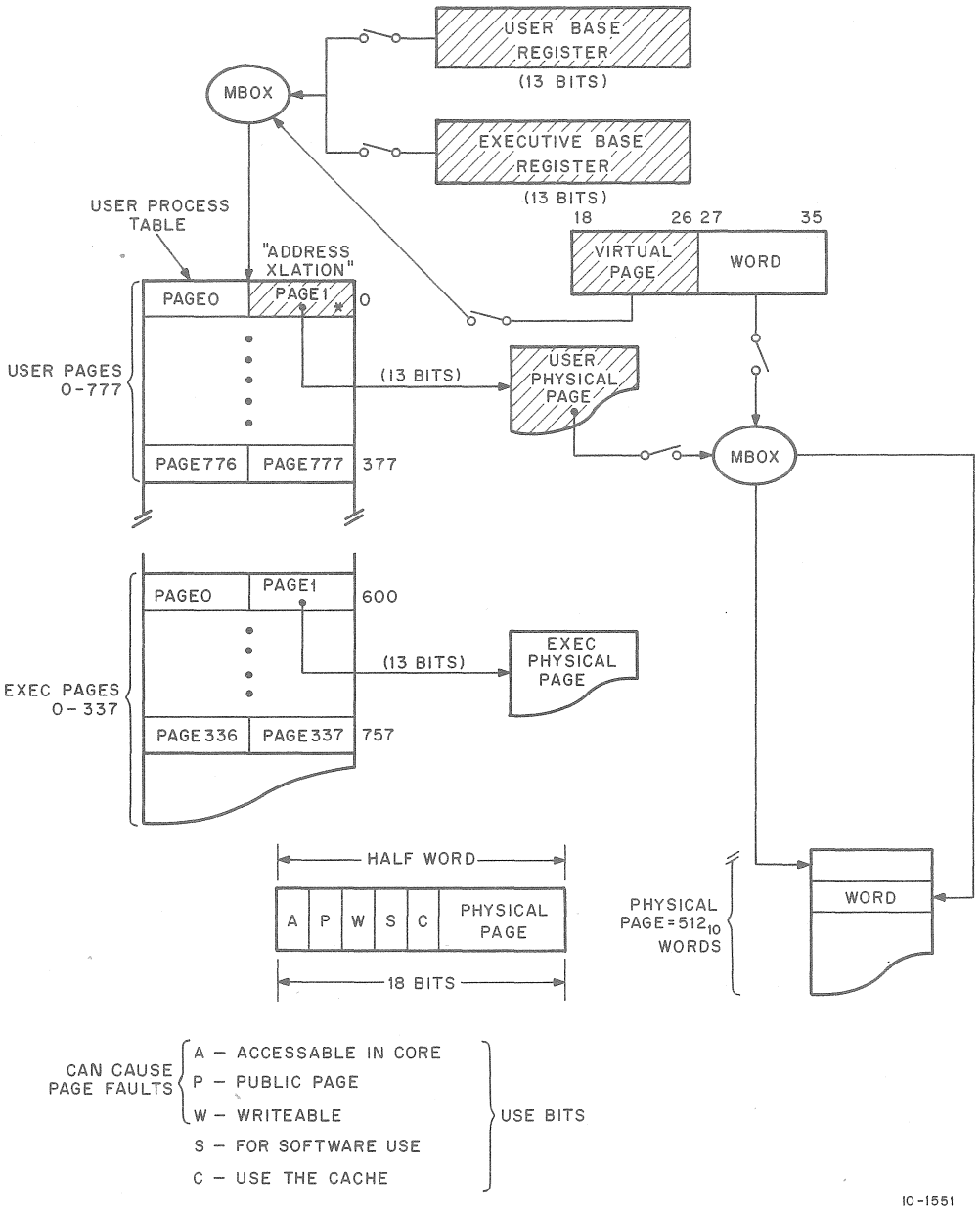



Figure 1-14 KI Style Paging

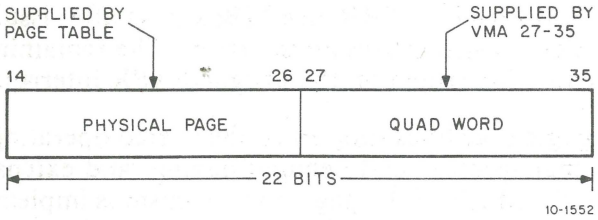



Figure 1-15 Physical Memory Address Format

**NOTE**

**A quadword is a block of four contiguous words whose address differs only in the two least significant bits.**

In practice, address bits 14–33 specify a 4-word block called a *quadword*; bits 34 and 35 specify which word within that quadword is required by the EBox, or is being written by the EBox. Once the address translation process has been successfully completed for a virtual page, subsequent references to that same page cause the MBox to fill in the corresponding words in the cache within the MBox. Each time a reference finds a valid word in the cache during a read, it is placed on the EBox cache data lines and MBox response is issued. Page faults occur as follows: For the initial reference, the MBox looks in the hardware page table in the MBox, does not find the physical page address, and performs the subsequent process table reference (refill cycle) for the half-word containing the use bits and physical page address. Then, upon receiving the eight half-word entries from core memory, the MBox finds the access bit turned off, i.e., 0; then a page fault is generated. The eight half-words are always written in the MBox hardware page table (directory) whether or not the access bit in the associated word is on. However, when the access bit for the associated word is off, the MBox asserts PAGE FAIL HOLD. The MBox loads an internal register (EBus register) with a page fail status word that describes the type of fault and also contains information about the user's virtual address. Referring to Figure 1-16, the EBox detects the PAGE FAIL HOLD level from the MBox, and forces the CRAM address logic to CRAM location 1777. Here the page fault handler is entered. It performs the indicated functions (Figure 1-16), and enters an Executive routine to handle the fault.

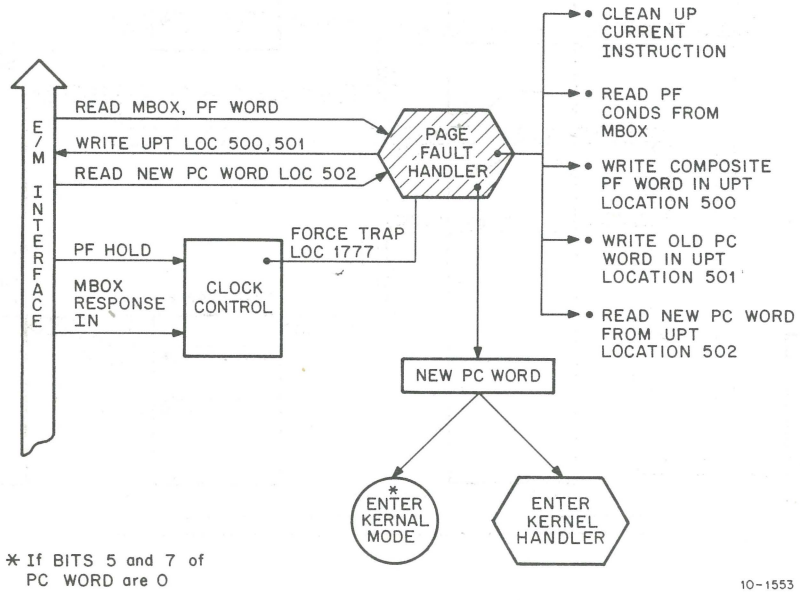



Figure 1-16 Page Fault Overview

In addition, the MBox asserts MB PARITY ERR five MBox ticks after issuing MBOX RESPONSE IN. This sets APR MB PAR ERR, which causes an interrupt. The remaining errors set appropriate APR error flags and likewise cause interrupts on the assigned APR interrupt channel.

**1.2.4.2 KL Paging** – The KL paging facilities support sophisticated operating system features such as efficient program working set management and demand paging, and extensive sharing of data and programs on a page-by-page basis. Much of the paging mechanism is implemented by the KL microcode, rather than just specific hardware. This combination of microcode and hardware is referred to as the KL10 pager of TOPS-20 paging.

Refer to Figure 1-17. Each user's virtual address space comprises 32 equal sections of 256K words per section (512 pages of 512 words per page). A section is represented by one of 32 section pointers located in the User Process Table (UPT). For EXEC sections, the 32 section pointers are in the EXEC Process Table (EPT). The monitor can divide the EXEC address space into "per-process" and "per-job" areas through the use of indirect pointers; no such division is built into the Pager.

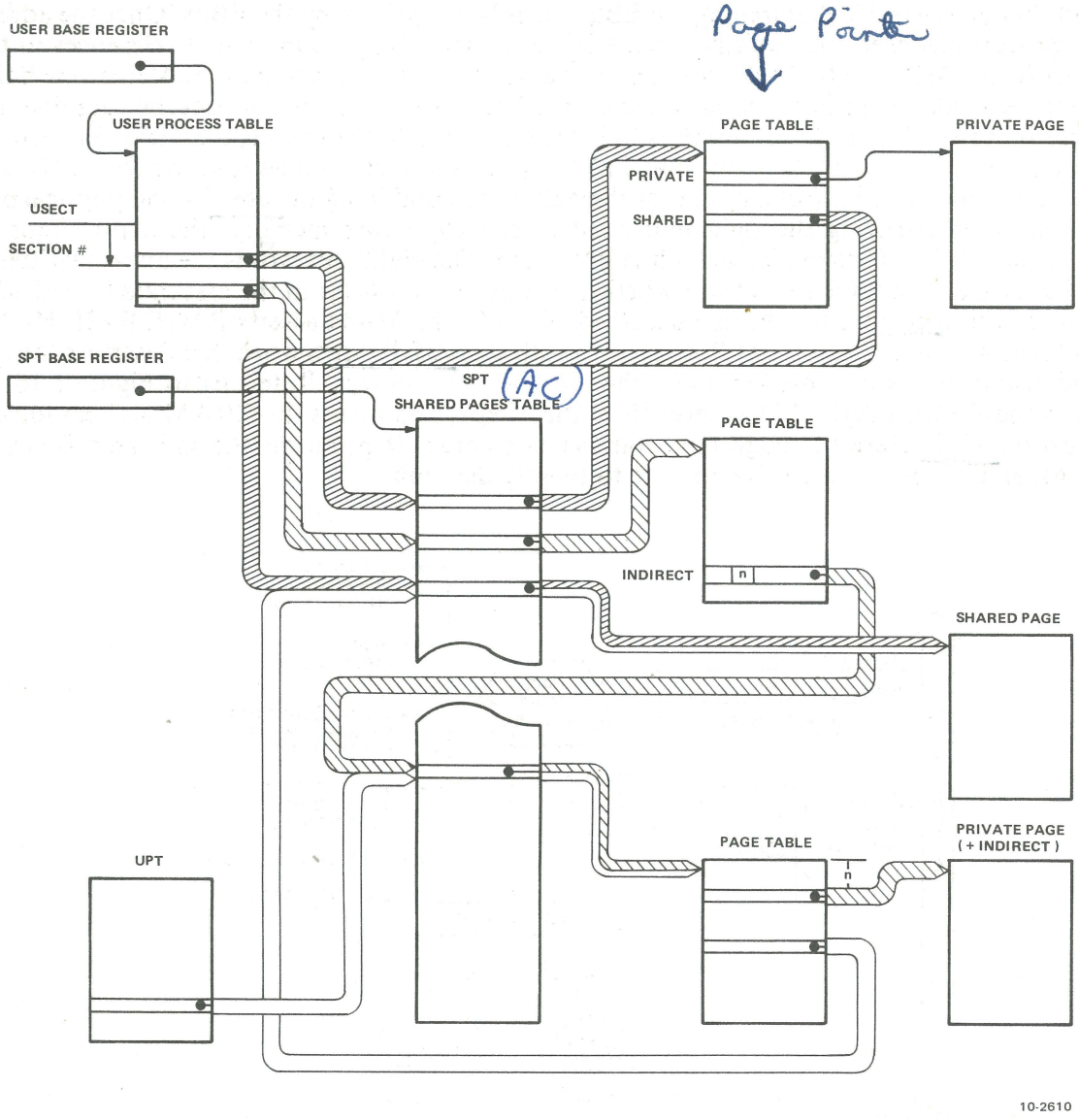



Figure 1-17 KL Paging Layout

A section pointer eventually addresses a page table that represents all pages in a 256K virtual address space. The section pointer may be Immediate, Shared, or Indirect, but must yield a physical address of a page table that represents all pages of the section.

The page pointer is divided into three sections: Type Code, Access Bits, and Storage. Figure 1-18 illustrates the basic page pointer format and Figure 1-19 shows the sequence of steps in its interpretation:

1. A virtual memory reference addresses a section pointer in the UPT or EPT for EXEC operation.

2. The section pointer is used to fetch an entry from the SPT (this is a pointer to a page table).

3. The SPT entry points to a location within a page table representing 512 pages by one page pointer for each page.

4. The page table holds the physical page number required to complete the virtual to physical address mapping.

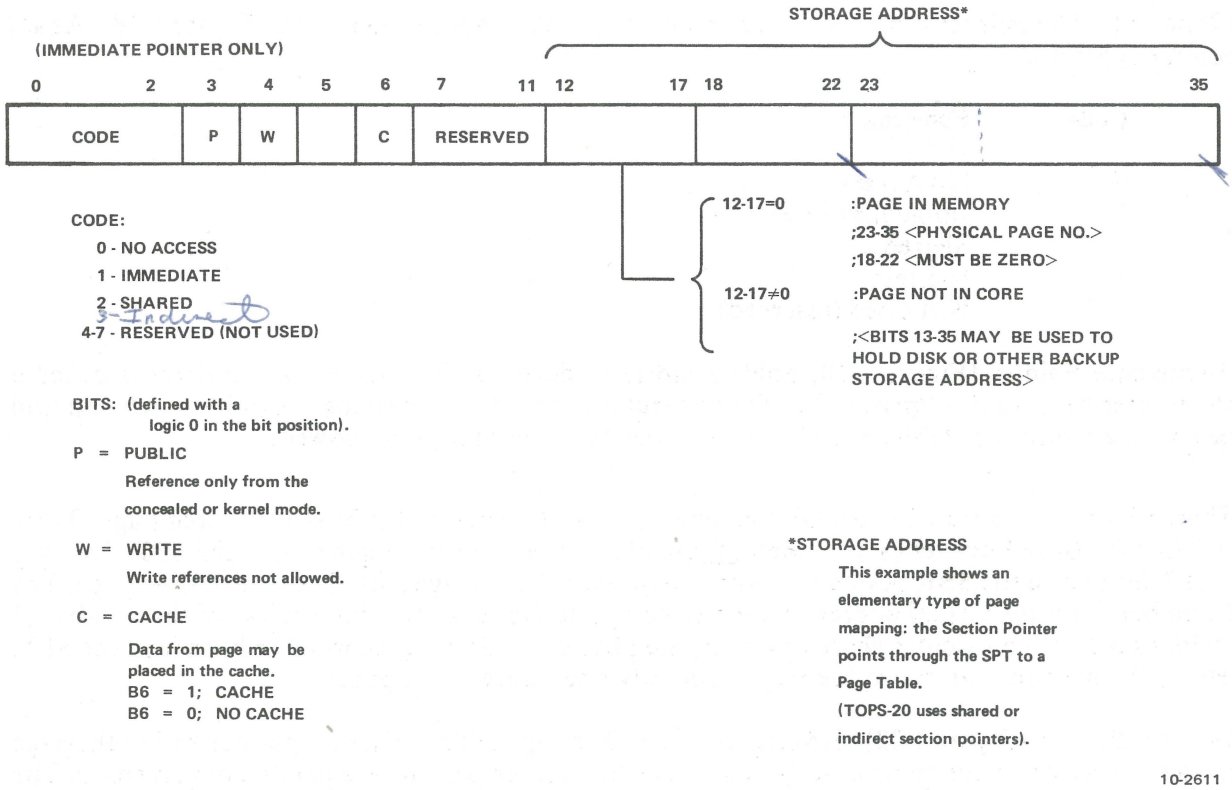



Figure 1-18 Page Mapping (Virtual to Physical)

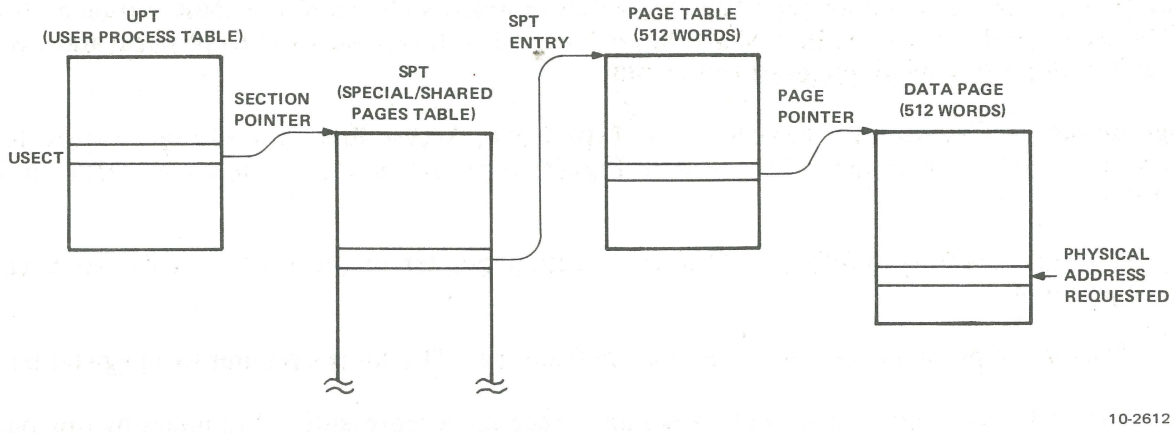



Figure 1-19 Typical Paging Path

These steps describe the most elementary and immediate reference type. The complexity of other reference types requires a discussion of pointer types.

**Page Pointers** – The pointer type is encoded in bits 0–2 of the page pointer word (Figure 1-18). Again the pointer types are:

| Code | Function |
|---|---|
| 0 | No Access |
| 1 | Immediate or Private |
| 2 | Shared |
| 3 | Indirect |
| 4–7 | Not Used (reserved) |

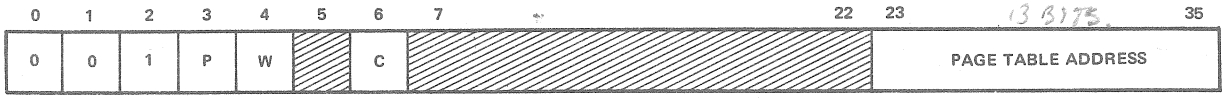
The Immediate Pointer (Figure 1-20) holds a storage address in bits 12–35. The pointer is called a private pointer because it is "private" to the particular page table containing the pointer. This should not be confused with the Public bit, which describes the type of access allowed.

The Shared Pointer (Figure 1-21) contains an index that addresses into the Special/Shared Pages Table (SPT). The SPT Base Register (SBR; reserved AC block) points to the beginning of the SPT. The sum of the SPT index and the SBR points to a word containing the storage address of the desired page. The word number from the virtual address is used to complete the reference. Regardless of the number of page tables holding a particular shared pointer, the physical address is recorded only once in the SPT. Therefore, the monitor can move the page with only one address to update.

The Indirect Pointer (Figure 1-22) identifies both another page table and a new pointer within the page table. This allows one page to be exactly equivalent to another page in a separate address space. The object page is located by using the SPT index.

Like a Shared Pointer, the SPT index in the Indirect Pointer allows the physical address of the page table to be stored in just one place. If the associated page is in memory, the page number field of the Indirect Pointer is used to select a new pointer word from the page table. This pointer can be any one of three types previously described, or no access and the access bits are ANDed with the access bits of the Indirect Pointer.

IMMEDIATE SECTION POINTER ($1_8$)



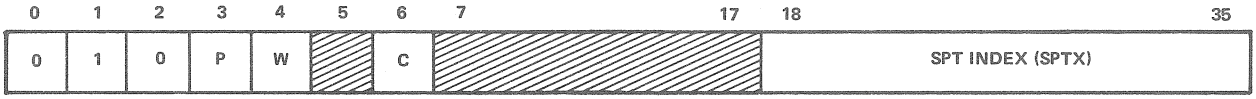

| BIT | DEFINITION | DESCRIPTION |
|---|---|---|
| 00-02 | Pointer Type | A $1_8$ in this field defines the Immediate Section Pointer. |
| 03 | Public Bit | If this bit is off (0), the page may only be referenced by programs running in Concealed or Kernel Mode. |
| 04 | Write Bit | When set, allows write references to be executed to the page. |
| 05 | Not Used | |
| 06 | Cache Bit | When set, allows page data to be entered into the Cache. |
| 07-22 | Not Used | |
| 23-35 | Address Bits | Defines the Page Table Address. |

NOTE: BITS 12-35 CONTAIN A "STORAGE ADDRESS". IF 12-27 ≠ 0, A TRAP IS CAUSED.

10-2613

Figure 1-20 Immediate Section Pointer

SHARED SECTION POINTER ($2_8$)

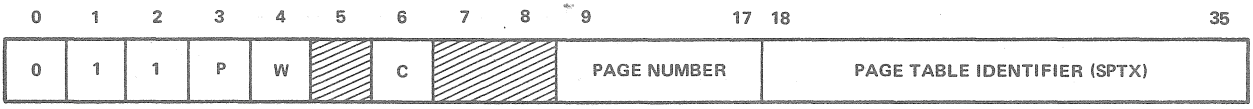



| BIT | DEFINITION | DESCRIPTION |
|---|---|---|
| 00-02 | Pointer Type | A $2_8$ in this field defines the Shared Section Pointer. |
| 03 | Public Bit | If this bit is off (0), the page may only be referenced by programs running in Concealed or Kernel Mode. |
| 04 | Write Bit | When set, allows write references to be executed to the page. |
| 05 | Not Used | |
| 06 | Cache Bit | When set, allows page data to be entered into the Cache. |
| 07-17 | Not Used | |
| 18-35 | SPT Index | The SPT entry is found at the physical core address given by the sum of the SPT base register and the SPT Index. |

10-2614

Figure 1-21 Shared Section Pointer

INDIRECT SECTION POINTER ($3_8$)




| BIT | DEFINITION | DESCRIPTION |
|---|---|---|
| 00-02 | Pointer Type | A $3_8$ in this field defines the Indirect Section Pointer. |
| 03 | Public Bit | If this bit is off (0), the page may only be referenced by programs running in Concealed or Kernel Mode. |
| 04 | Write Bit | When set, allows write references to be executed to the page. |
| 05 | Not Used | |
| 06 | Cache Bit | When set, allows page data to be entered into the Cache. |
| 07-08 | Not Used | |
| 09-17 | Section Table Index (Page Number) | Indicates the location within the Page Table of the new pointer (indirect reference). |
| 18-35 | SPT Index | The SPT entry is found at the physical core address given by the sum of the SPT base register and the SPT Index. |

10-2615

Figure 1-22 Indirect Section Pointer

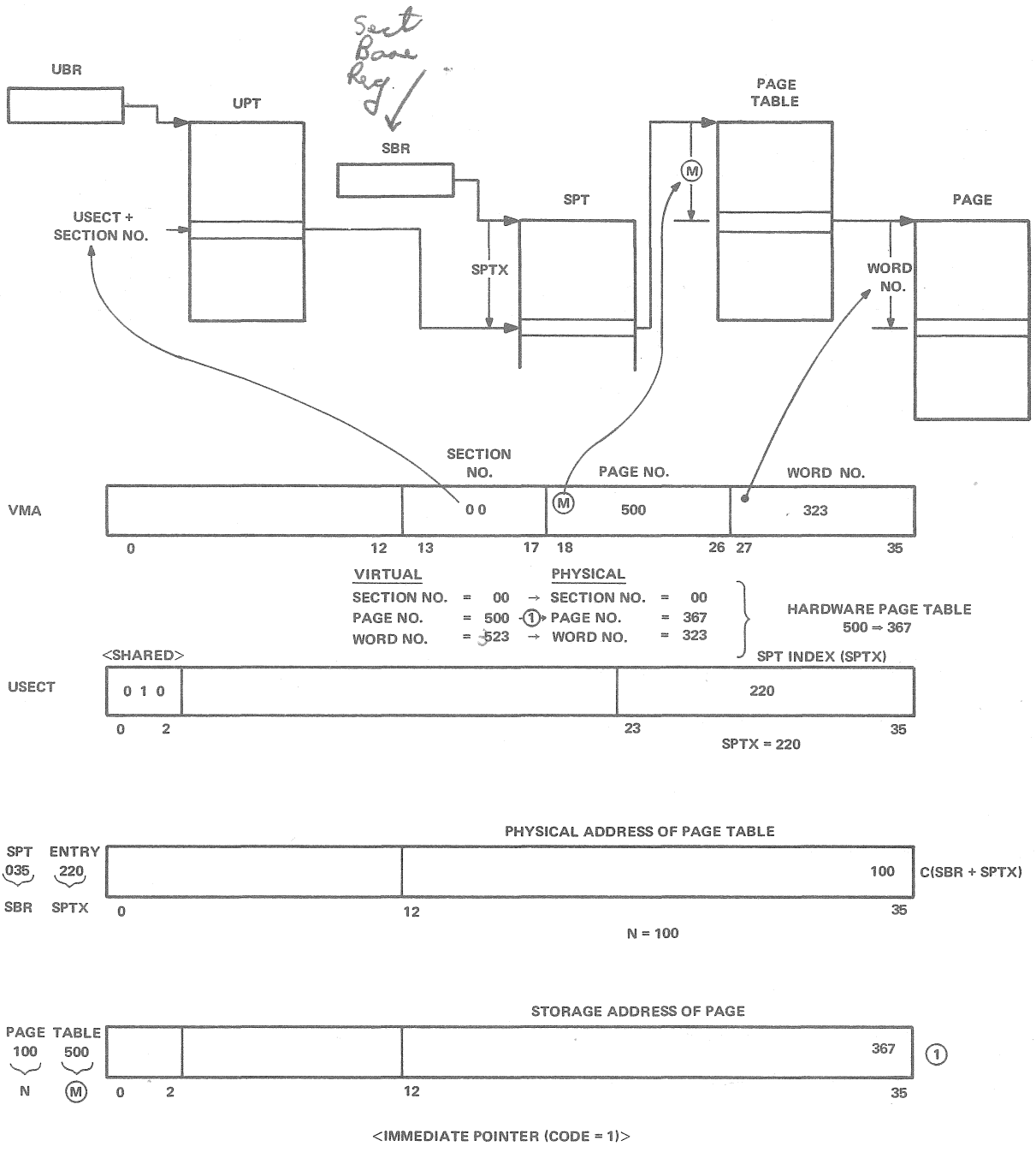
The Indirect chaining may be arbitrary in depth, but the PI will break out of indirect chain and restart after the PI to service a priority interrupt in the case of long direct chains or indirect loops.

Some examples (Figures 1-23 through 1-25) of pointer interpretation follow: a flow chart (Figure 1-26) is provided to aid in working through the examples.

**Special/Shared Pages Table** (SPT) – The Special/Shared Pages Table (SPT) contains the physical addresses of pages that are shared by many page tables, or of pages used in a special way, i.e., as page tables. They are stored in one common location to allow modification to the pages by changing a single entry. The SPT Index is added to the STP base address to form a physical address of the associated entry.

**Core Status Table** (CST) – Virtual memory management requires information about memory references generated by each user's processes. Adding the Core Status Table (CST) base register to the physical page number from a storage address permits the monitor to address and update information regarding the page reference. Figure 1-27 shows the flow of updating using a CST entry. This enables pages to be ordered by "age" (time of last reference) and classified by the type of process referencing the page.

The reference indication is carried by assigning one bit to each active process. By placing a 1 in that bit positon in the pager data word, then, when a reference is made, the 1 is placed in the CST word in the bit position assigned to the process making reference. The modified bit (35) is set if the page is modified, permitting the monitor to avoid swapping out of pages to which only read references are made.

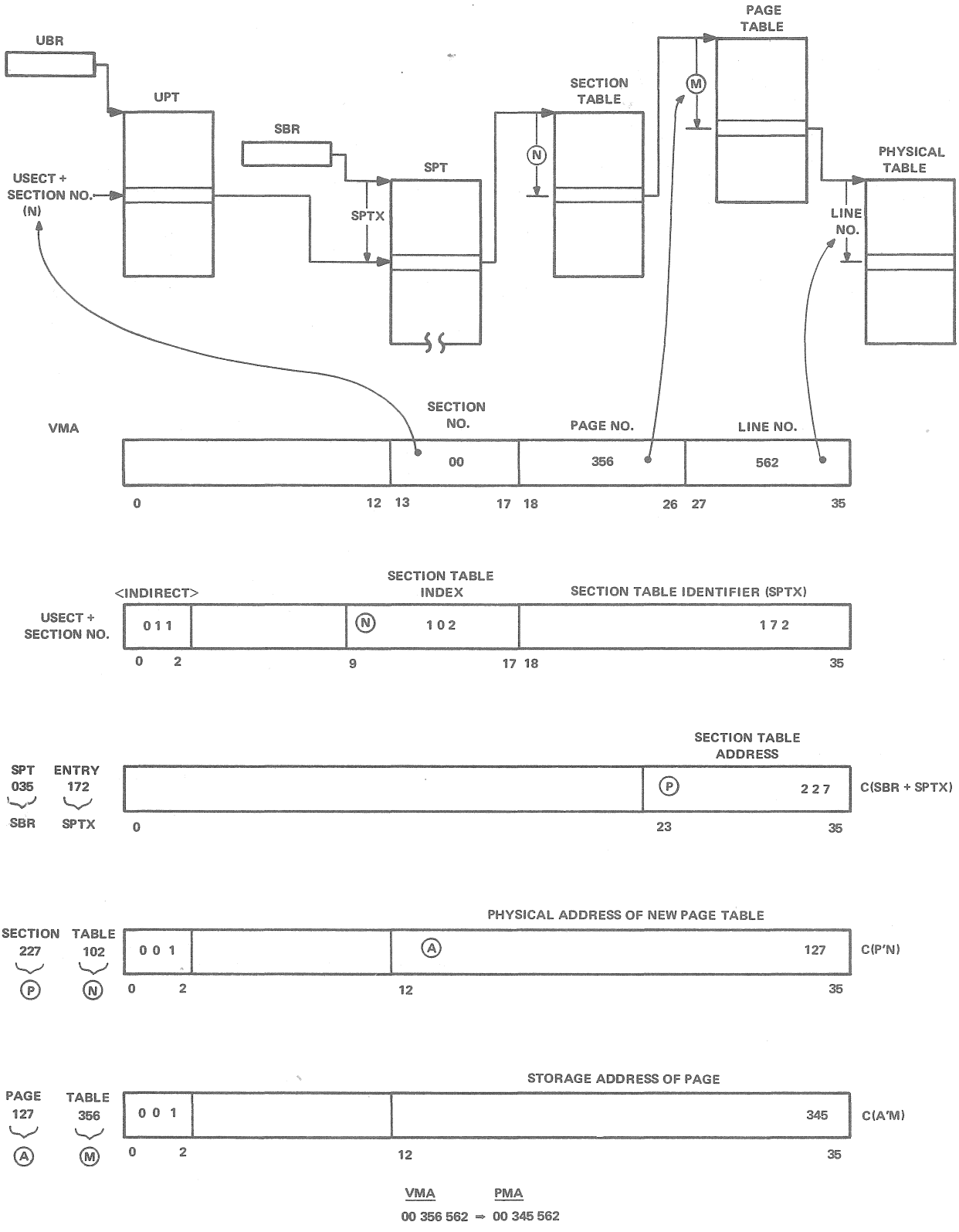



C (UBR' USECT + SECTION NO.) CONTAINS SPTX
C (SBR + SPTX) CONTAINS PAGE TABLE PAGE NO. N
C (N'M) CONTAINS STORAGE ADDRESS OF DESIRED PAGE

NOTES: A'B :: = A CONCATENATED WITH B
Assume page is in core.

10-2616

Figure 1-23 Pointer Interpretation (Normal Section Pointer; Shared)




NOTE: Assume page is in core.

10-2617

Figure 1-24 Pointer Interpretation (Indirect Section Pointer)

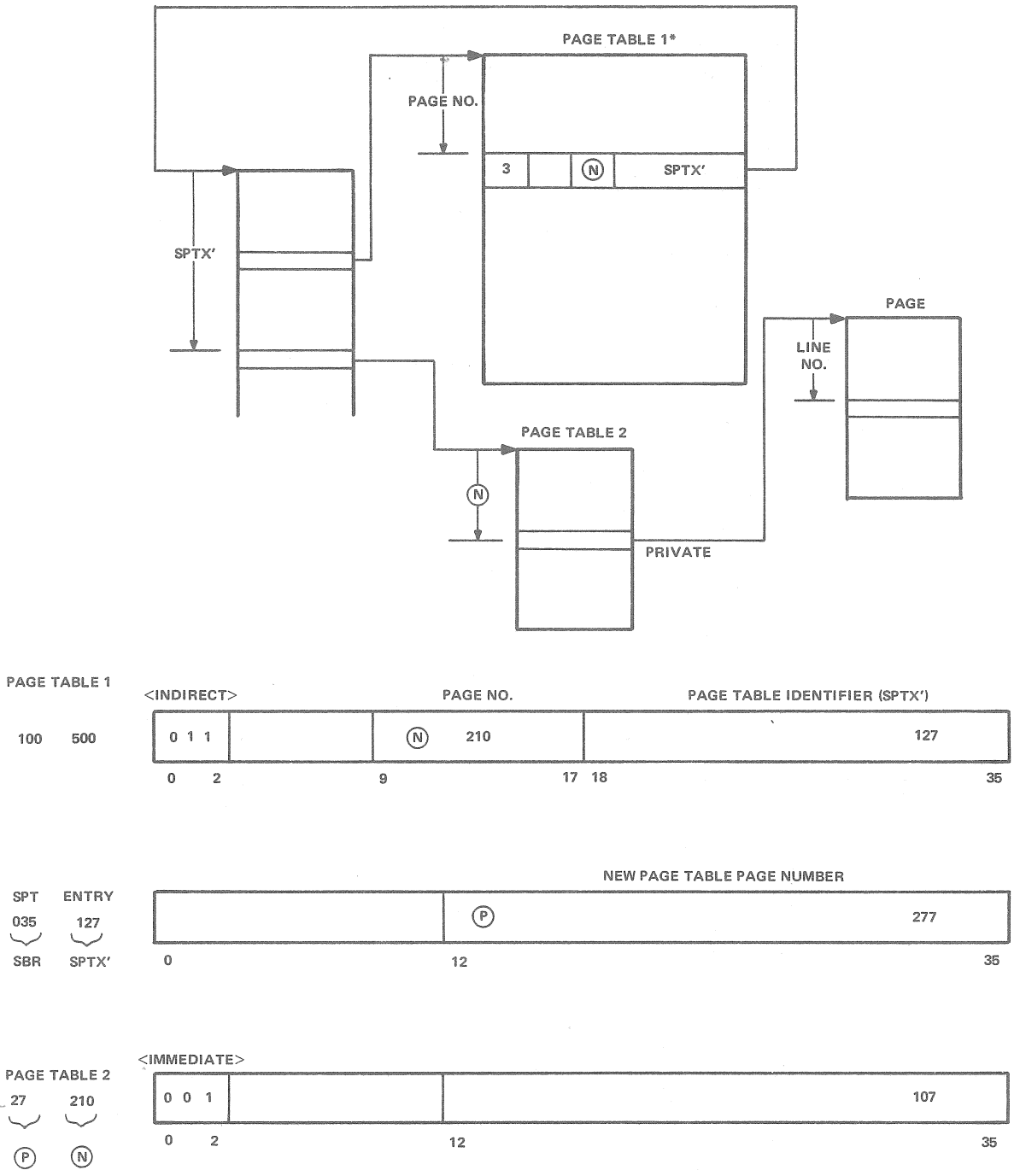



Figure 1-25 Pointer Interpretation (Indirect Page Pointer)

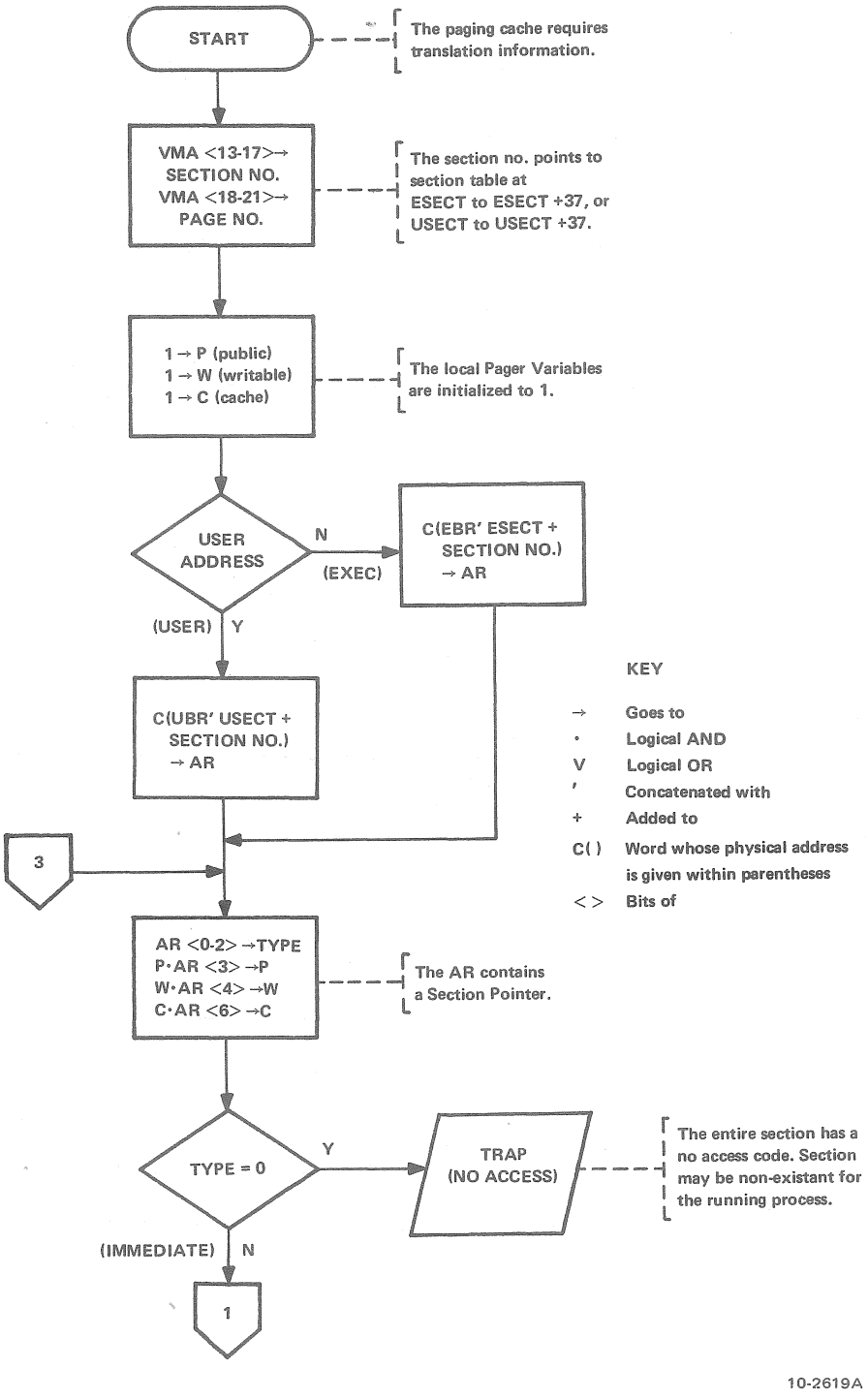



Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 1 of 5)

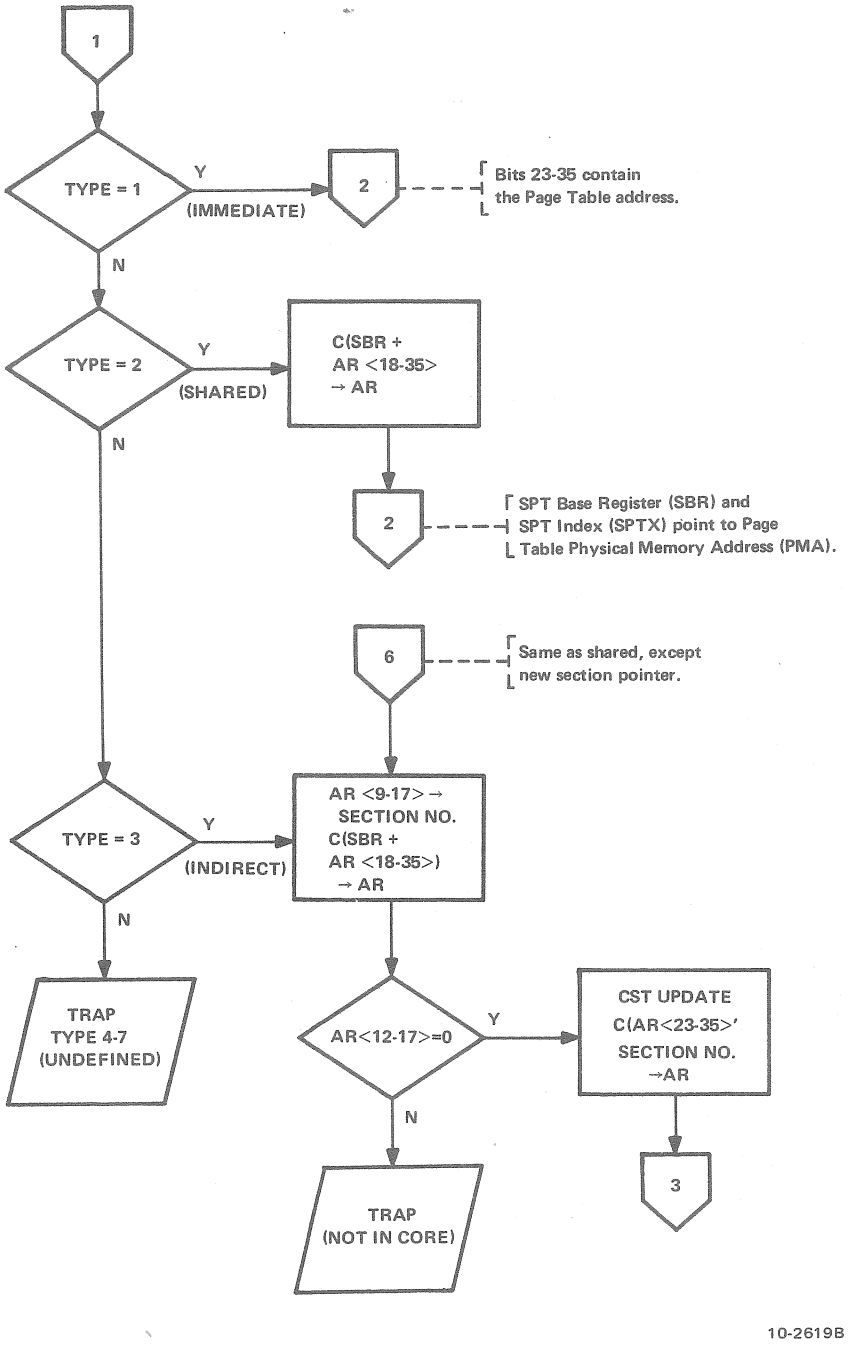



Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 2 of 5)

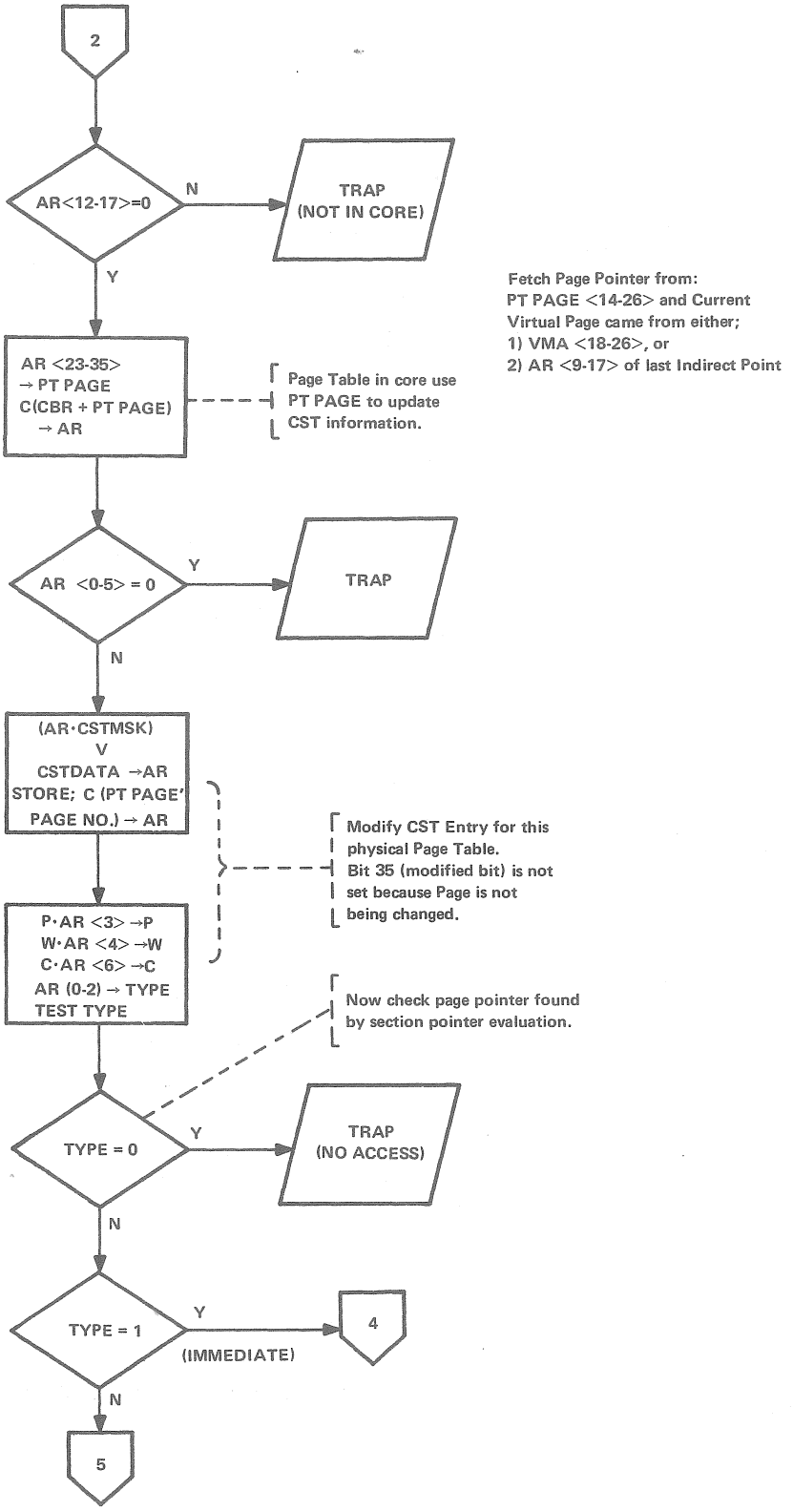



Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 3 of 5)

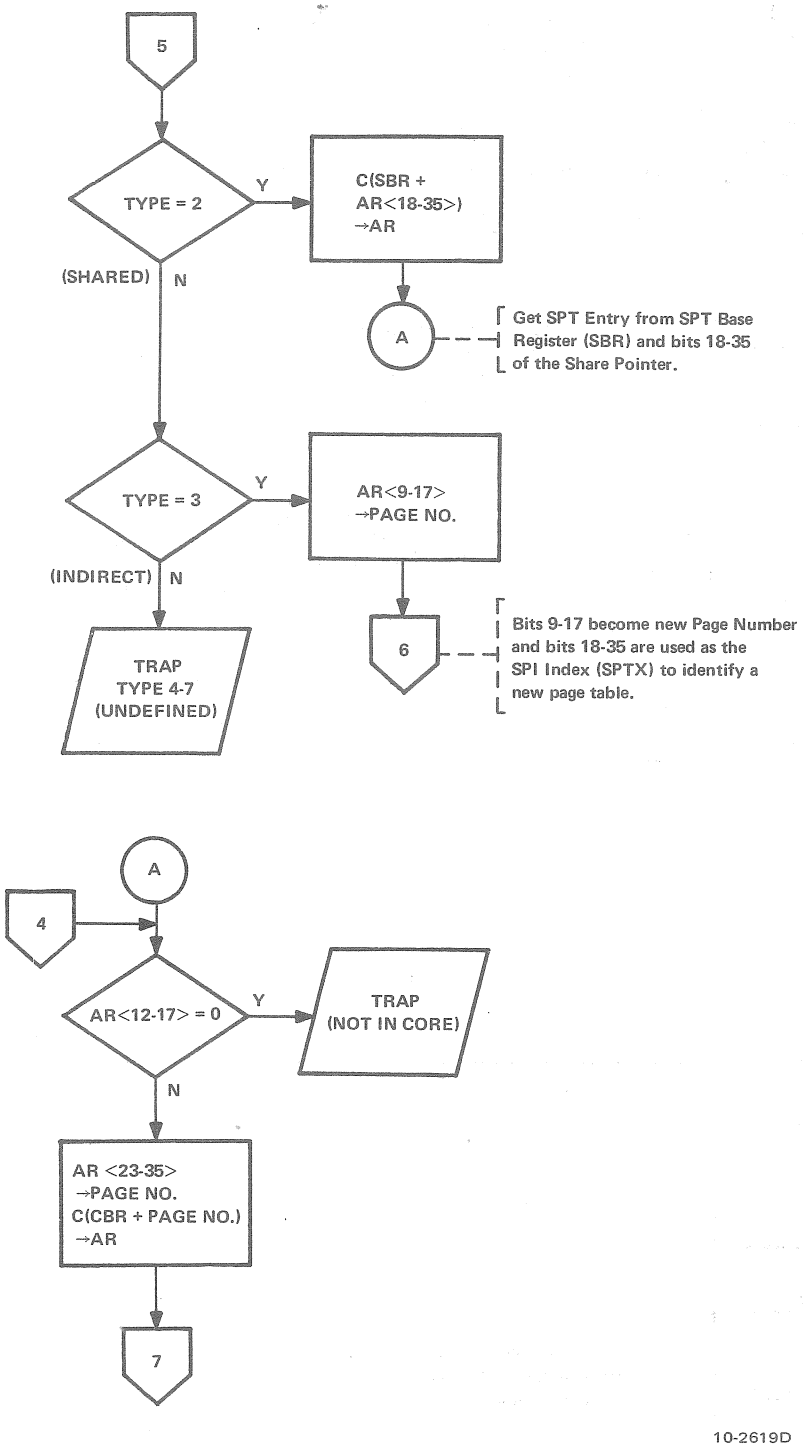



Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 4 of 5)

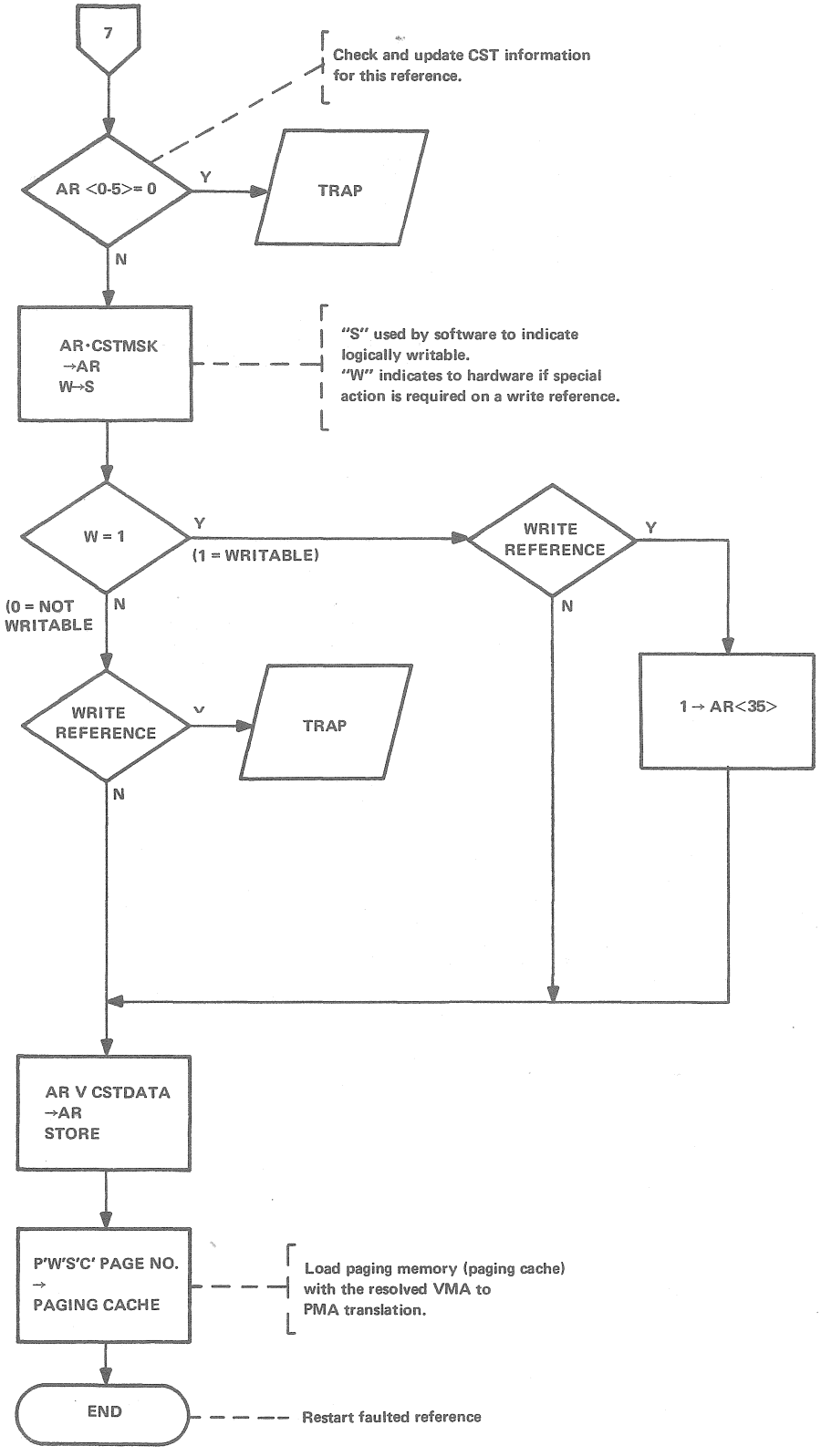



Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 5 of 5)

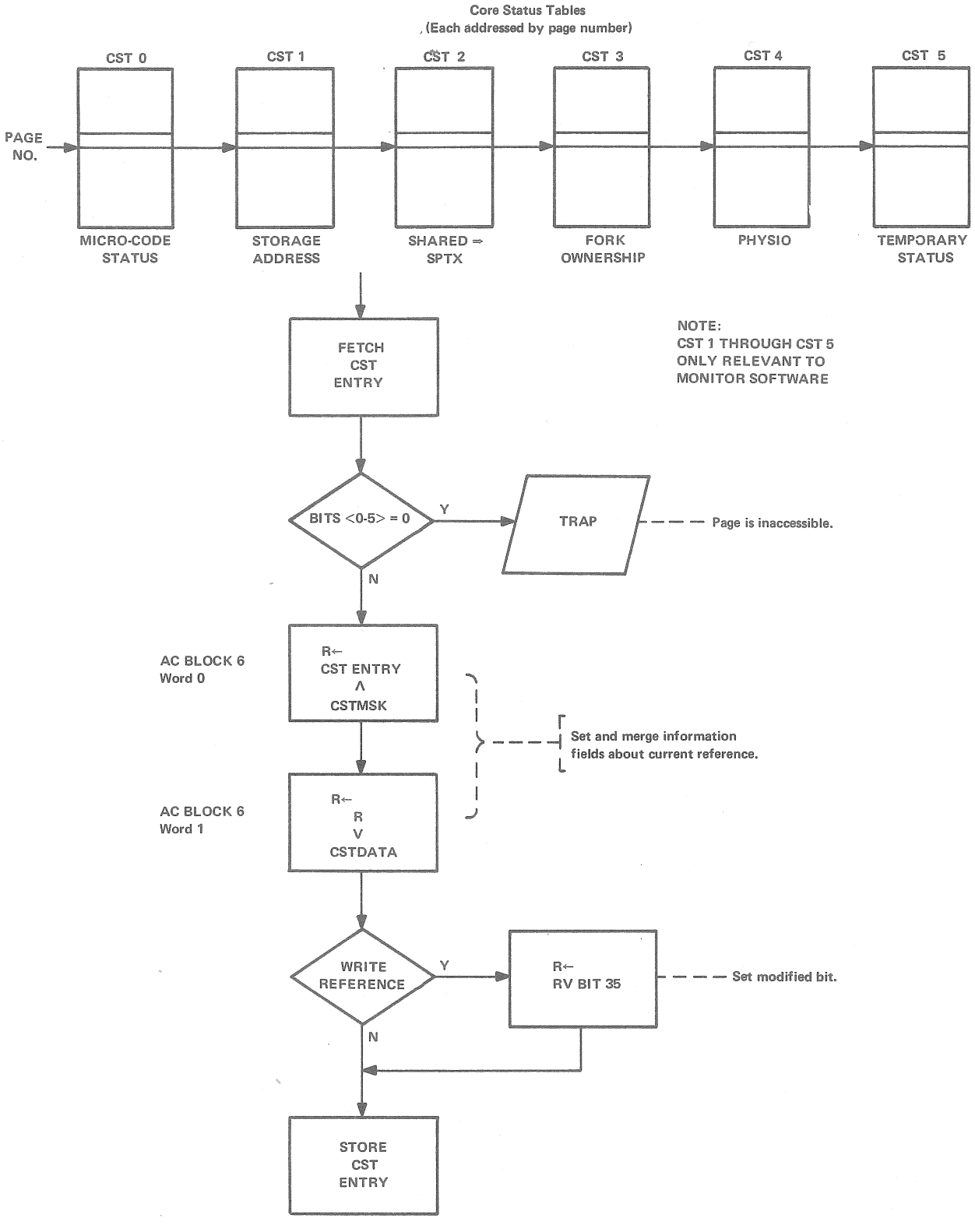



Figure 1-27 KL Core Status Tables Updating Flow Diagram

**Paging Hardware Support** – The paging hardware is transparent to the user. All memory, both virtual and physical in user and monitor space, is divided into pages.

The virtual address comprises 23 bits, five (5) bits for section numbers, nine (9) bits for virtual page numbers, and nine (9) low-order bits (line number), which address the location within the page. The virtual page number is first used as an index into a hardware page table that contains up to 512 direct virtual-to-physical address translations. If the 13-bit physical address is found in the hardware page table, a 22-bit physical address is formed by concatenating the 13-bit physical address with the 9-bit line number. If the entry does not exist in the hardware page table, a sequence of translations is initiated to locate a page table in memory that contains a physical address (if one exists) for the virtual page.

**Cached Paging Data** – The hardware page table referred to at the beginning of this section is effectively a cache of paging data (not to be confused with the memory data cache) that has been accumulated by previously fetching the data from memory, or by previous pointer interpretation. A virtual address is first checked against the current contents of this hardware pager and, if found, immediately returns a physical address. If the physical address is not found, the pointer interpretation (Figure 1-26) fetches information from memory to resolve the virtual address. Upon completion, this translation may be placed in the hardware page table forming the cache of recently used page addresses.

The hardware page table is loaded by the microcode. The paging cache is implemented as 512 entries, one for each page of a user's virtual address space. The EXEC and USER are offset from each other, but they share the same 512 entries. Therefore, at any given time, the paging cache holds translation information about most of the active pages. A guarantee that the 512 most recently used pages will be addressed by the paging cache cannot be made. However, the last page used will *always* be in the paging cache.

When the monitor takes any action that would invalidate information about existing virtual-to-physical address translation, the paging cache must be either partially or completely cleared. Examples of such instances are:

1. Change of user process – clear entire paging memory (entire user address space has changed).

2. One page removed from core – clear the entire paging memory (several Shared and Indirect Pointers may have used the page).

3. Pointer is removed from UPT – clear the entire paging memory (association for many pages through UPT is changed).

4. Monitor mapped page to EXEC space for local use – only one entry cleared (When page is unmapped, only that one pointer must be cleared. Because this facility is provided by the pager, it may be used to reduce reload overhead.)

If the paging data is not found, the flow in Figure 1-26 is followed. A special trap is initiated and the microcode saves vulnerable EBox data before starting on the pointer tracing algorithm. If the algorithm is successful, the resolved pointer and associated information are loaded into the paging memory, the EBox registers are restored, and the memory request is again issued.

The microcode must also handle the first Write Request trap, inhibiting the write until the modified bit can be set. The pager maintains this modified bit. The microcode implements this as follows.

During a paging memory reload, the write access bit (W) is set in the paging memory only if the current memory reference is a write (and a write is legal for the page). Thus, if the first reference to a page is a read, the W bit in the corresponding paging memory entry sets to 0. A subsequent write reference causes another trap to the microcode. On this second trap, the pointer interpretation is repeated and the paging memory is reloaded, this time with the W bit set.

**1.2.4.3 MBox Error Conditions** – In addition to the page fault mechanism, the following five types of errors can be generated by the MBox to the EBox:

1. Cache Address Parity Error
2. MBox Address Parity Error
3. SBus Error
4. Nonexistent Memory
5. MB Parity Error

The MB Parity Error is handled similar to a page fault. The AR Parity Network, upon detecting a parity error in a data fetch or an instruction fetched from the MBox, causes the page fault handler to be called.

**1.2.4.4 VMA Control** – Two basic types of virtual addresses can be passed to the MBox for core memory references. The first type is consistent with KI-style paging; the second is consistent with KL-style paging. In both forms of addressing, note that the VMA lines actually consist of 23 bits. For KI-style paging, bits 13–17 are unused and forced to 0. In the logical sense, the virtual address may be viewed for KI-style paging as consisting of 18 bits of addressing information. The basic address translation mechanism is indicated in Figure 1-28.

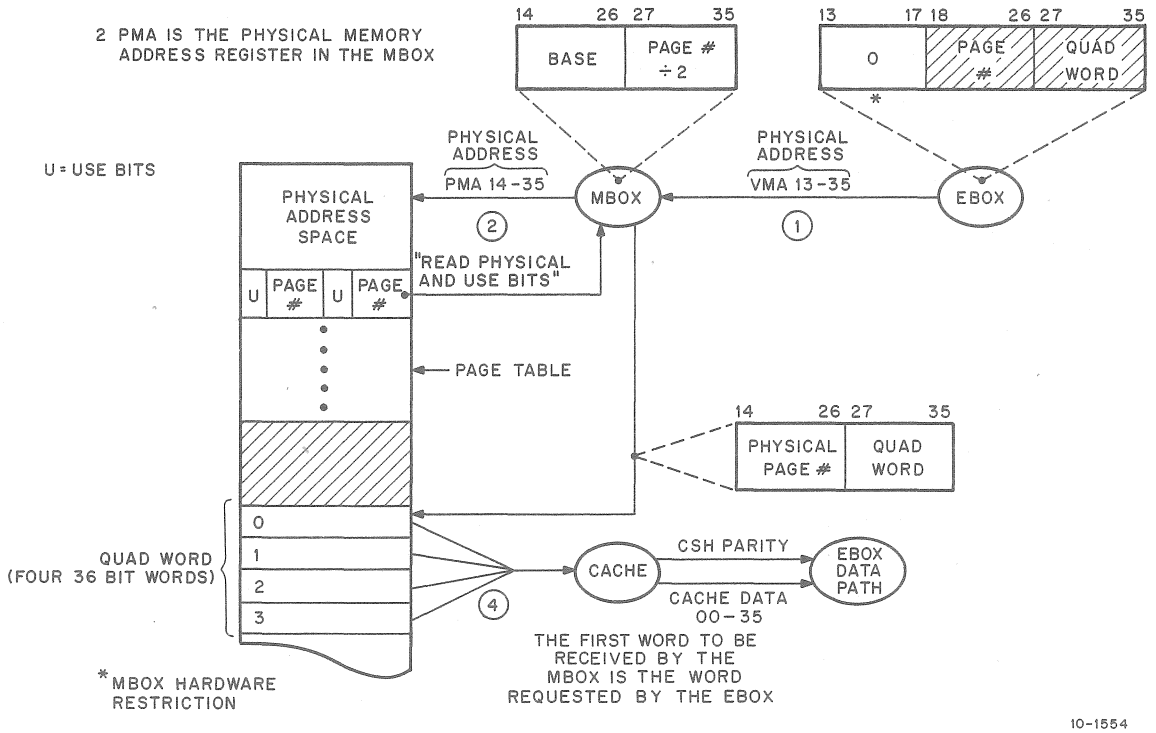



Figure 1-28 Basic Address Translation

Actually, the virtual address in KI10 paging mode is derived from the instruction Y field, which may be modified during the effective address calculation. This consists of 18 bits. The additional five bits (VMA 13–17) are present to facilitate KL paging mode, which can generate a 23-bit virtual address. However, the MBox does utilize the high-order part of the VMA as indicated in Figure 1-29 to generate a Hashed Page Table address for internal use. The hashing technique is basically an associative process, but precludes the necessity for hardware associative memory.

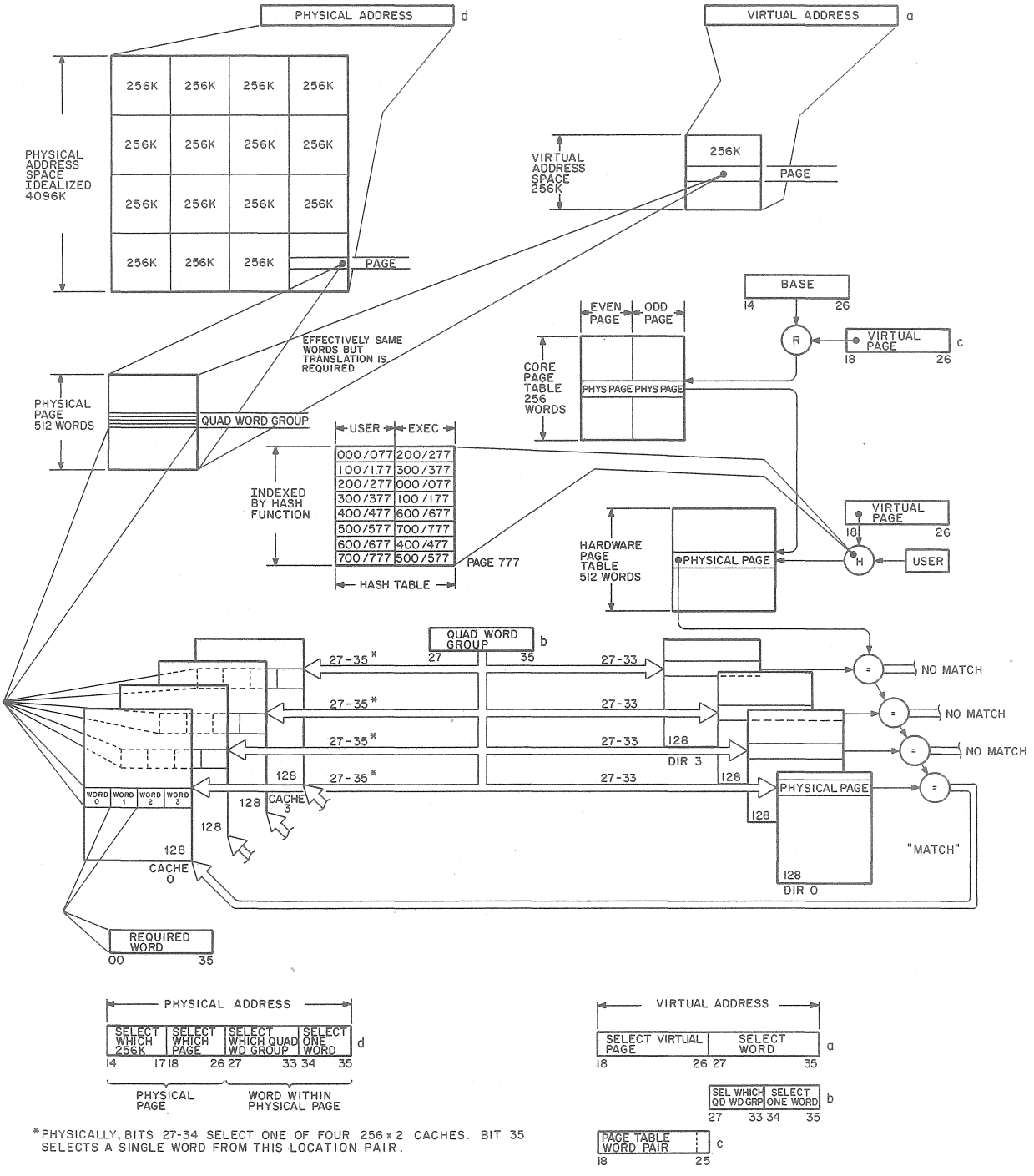



Figure 1-29 Virtual Address Mapping, KI10 Paging Mode

The VMA can be loaded from the ADDER or VMA ADDER. Generally, during calculations for the effective address, it is loaded with the contents of ARX via the ADDER. At this time, ARX contains an intermediate address [Y + C(XR)] or E.

### 1.2.5 EBus Control and PI Control

The EBus control consists primarily of two major sections. One section is used exclusively for priority interrupt handling (PI CONTROL) and the second is used for I/O instruction handling (EBUS CONTROL). Each KL10 controller (except the DIA20 I/O Bus Adapter) is assigned a device code. This code is seven bits wide (IR 3–9). In addition, each device controller is wired to contain a physical device number that relates to a preassigned scheme, and is slot dependent. Thus, Massbus controllers hold physical numbers in the range of 0–7; DTE20 numbers $10–13_8$ and DIA20 number $17_8$. This provides a physical priority scheme that supplements the programmable priority interrupt system.

In the situation illustrated in Figure 1-30, both DSKs are assigned to the same PI level (level 5). This is accomplished by the operating system with a CONO PI to the PI system enabling the processor to accept interrupts on level 5. In addition, the operating system performs a CONO DSK, assigning the DSK to level 5. For the situation where both DSKs interrupt simultaneously, the EBox arbitrates the priority interrupt levels and then physical device numbers are requested from both DSKs. These are arbitrated according to the fixed scheme discussed previously. The DSK with physical No. 0 has highest priority in this situation.

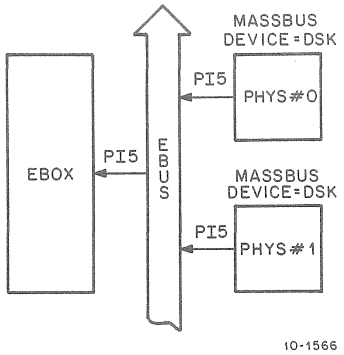



Figure 1-30 Simultaneous Interrupts

The basic dialogue is shown in Figure 1-31. Once the priority interrupt system has been turned on and set up by the operating system to handle interrupts, the EBox control automatically carries out all dialogues necessary to obtain the API function word. When the API function is on the EBus and transfer is received from the device, the EBus control asserts PI READY, signaling the microprocessor to take over. The microprocessor looks at this line, however, only at specific times during normal instructions. One such instance is at NICOND Dispatch, which always occurs at the beginning of each instruction. If at NICOND time, the PI RDY condition is true (INT REQUEST sets), the PI HANDLER is called. To prevent further interruptions until the function can begin, the microprocessor sets the PI CYCLE flag. This causes the EBus Control to defer any further PI READYs. The PI HANDLER evaluates the API function word (Figure 1-32) and performs the indicated service. As long as PI CYCLE is on, other interrupts are not honored by the microprocessor. The time that PI CYCLE is cleared is dependent upon the service performed. If the interrupt is a standard interrupt to 40 + 2n, the instruction in 40 + 2n should save the hardware state of the EBox, i.e., the flags, PC word. Appropriate instructions are JSR and MUUO. Bad choices are JSP and PUSHJ, which use ACs. The choice is particularly bad because at the time of the interrupt nothing is known about their contents.

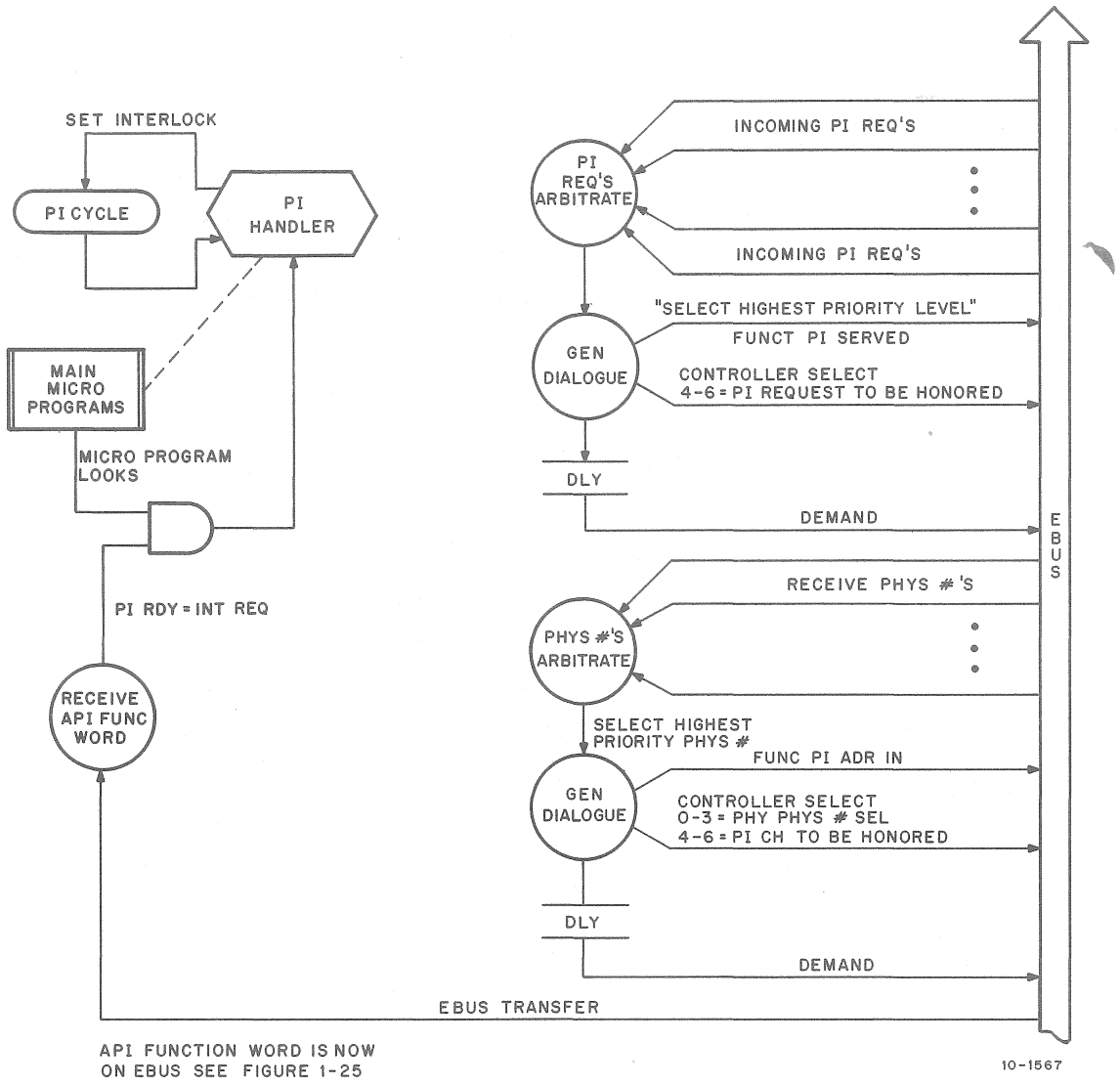



Figure 1-31 PI Dialogue Overview

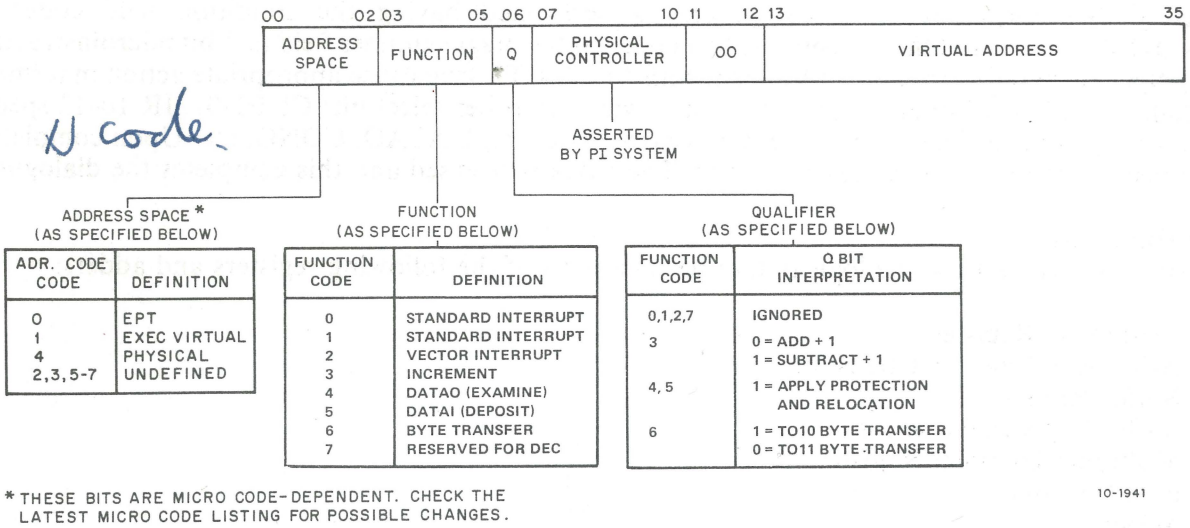



Figure 1-32 API Word Format

Generally, a JSR instruction is placed in 40 + 2n for calls to the operating system PI HANDLER. This instruction causes PI CYCLE to clear. At this time, a pending interrupt may request microprocessor attention and can raise PI READY. In general, for the other cases, the equivalent of one instruction is provided before PI CYCLE is cleared.

**I/O Instruction Dialogue Overview** – For I/O instruction transfers, the basic concept is illustrated in Figure 1-33. The EBus Driver is called from the I/O HANDLER to generate the appropriate EBus dialogue. First, the EBus is requested. This is necessary because the EBus is also used by the PI system. If the EBus is free, the EBus driver sets a CP GRANT flag to hold control of the EBus; if the EBus is in use, the EBox waits.

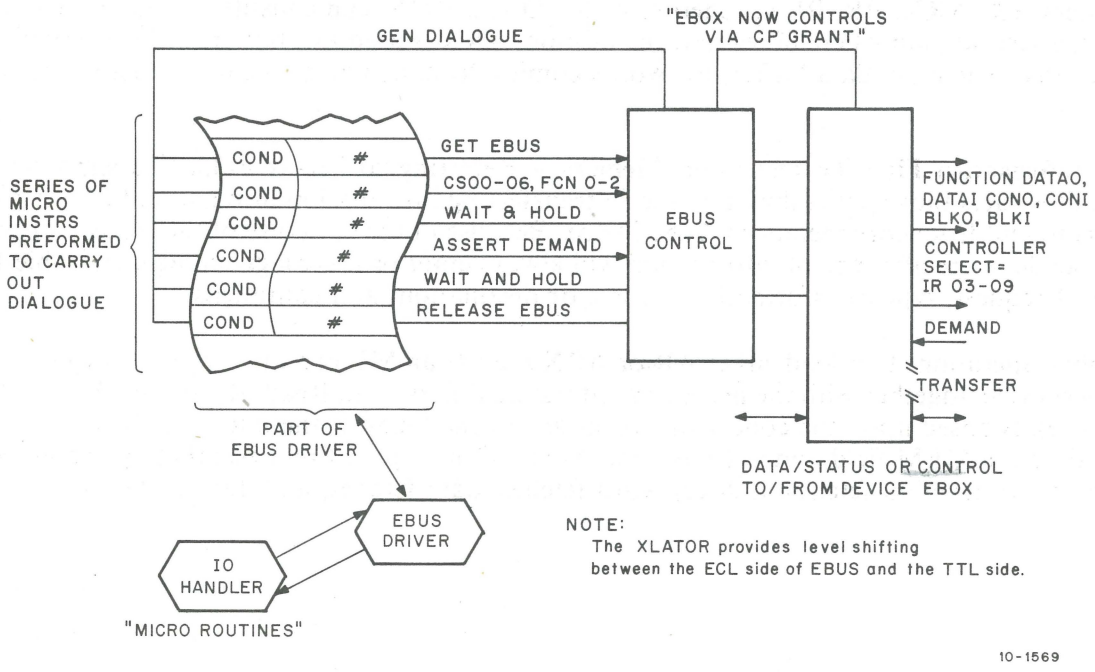



Figure 1-33 I/O Instruction Dialogue Overview

Basically, a sequence of microinstructions is performed having the condition field coded as COND/EBUS CTL and the appropriate bits coded in the magic number field (a 9-bit microinstruction field). Specific patterns in the number field with EBUS CTL true cause appropriate action in terms of the dialogue. IR bits 3–9 are used to develop device controller select bits CS 00–07. IR 10–12 specify the function to be performed by the EBus control logic, i.e., DATAO, CONO, etc. Upon completing the transfer, the device generates a transfer. The EBus is released and this completes the dialogue.

### 1.2.6 Data Path

Referring to Figure 1-34, the logical data path consists of the following registers and adders:

Arithmetic Register
Arithmetic Register Extension
Buffer Register
Buffer Register Extension
Multiplier Quotient Register
Fast Memory
Adder
Adder Extension

Also included is fast memory and a 36-bit shift matrix that can implement various shifting operations on data in AR, ARX, or the combined AR and ARX. The above registers and adders constitute the arithmetic logic in the EBox. This logic is used to handle words in logical operations, data transfers, and fixed-point arithmetic (including effective address calculation). In these operations, fast memory is used as a passive register; its output is the contents of the addressed Index register or Arithmetic register. In association with the full word registers listed above, the shift counter (SC) and shift matrix (SH) provide shifting in shift instructions, byte manipulation and, where required, in various instructions. The SC, with its adder (SCAD), and the floating exponent register (FE) are used for handling floating-point exponents and various other special functions.

Double-precision floating-point and double precision integer operations require use of ARX, ADX, and MQ, where ADX is a 36-bit extension of the main AD and ARX is a 36-bit extension of AR. Thus, the registers AR, ARX, BR, BRX, together with AD and ADX, can constitute a 36-bit, a 72-bit, and with MQ, a 108-bit path where necessary. In addition, ARX is used as a buffer for instructions fetched from memory. The main data buffer, for words coming from or going to core or fast memory, is the AR.

**1.2.6.1 Information Flow To and From Memory** – Referring to Figure 1-35, this simplified block diagram illustrates those paths that are used in transferring information into and out of fast memory, as well as to and from core memory via the MBox. Because of the structure of the EBox and design of the microcode, a specific type of information will always enter or leave a given register. Table 1-3 lists the type of request, type of information, source or destination, and comments.

All memory operations that load either AR or ARX require an MBox request cycle. The generation of this request cycle, together with the necessary request qualifiers (e.g., Read, Read PSE Write, Write, or Read-Write), is based upon the code specified in one of the fields of the microinstruction word. This field is called the MEM field and is 4 bits wide. Some of the types of requests that can be initiated by this field are: instruction fetches, indirect word fetches, data fetches, and data writes.

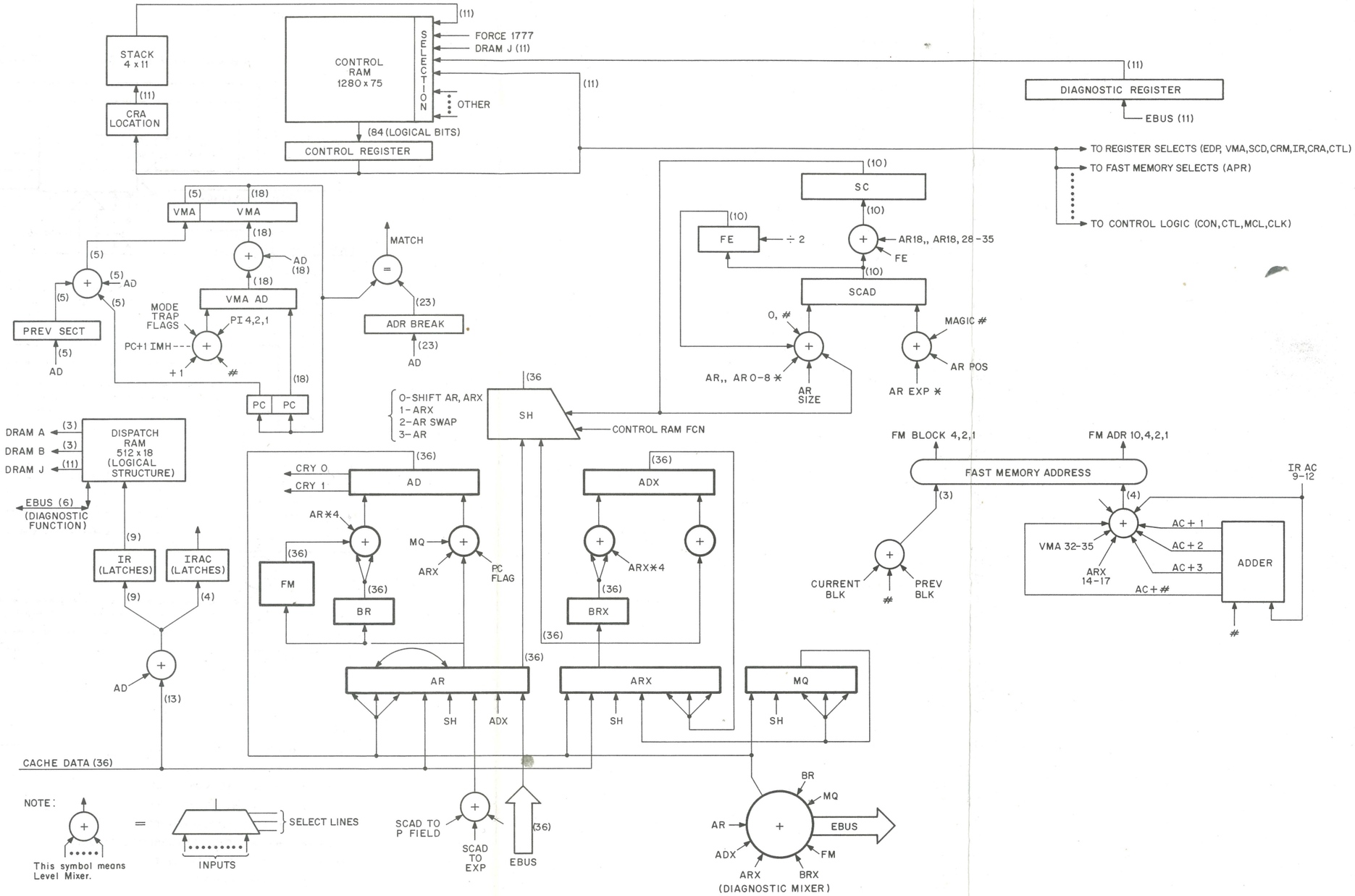
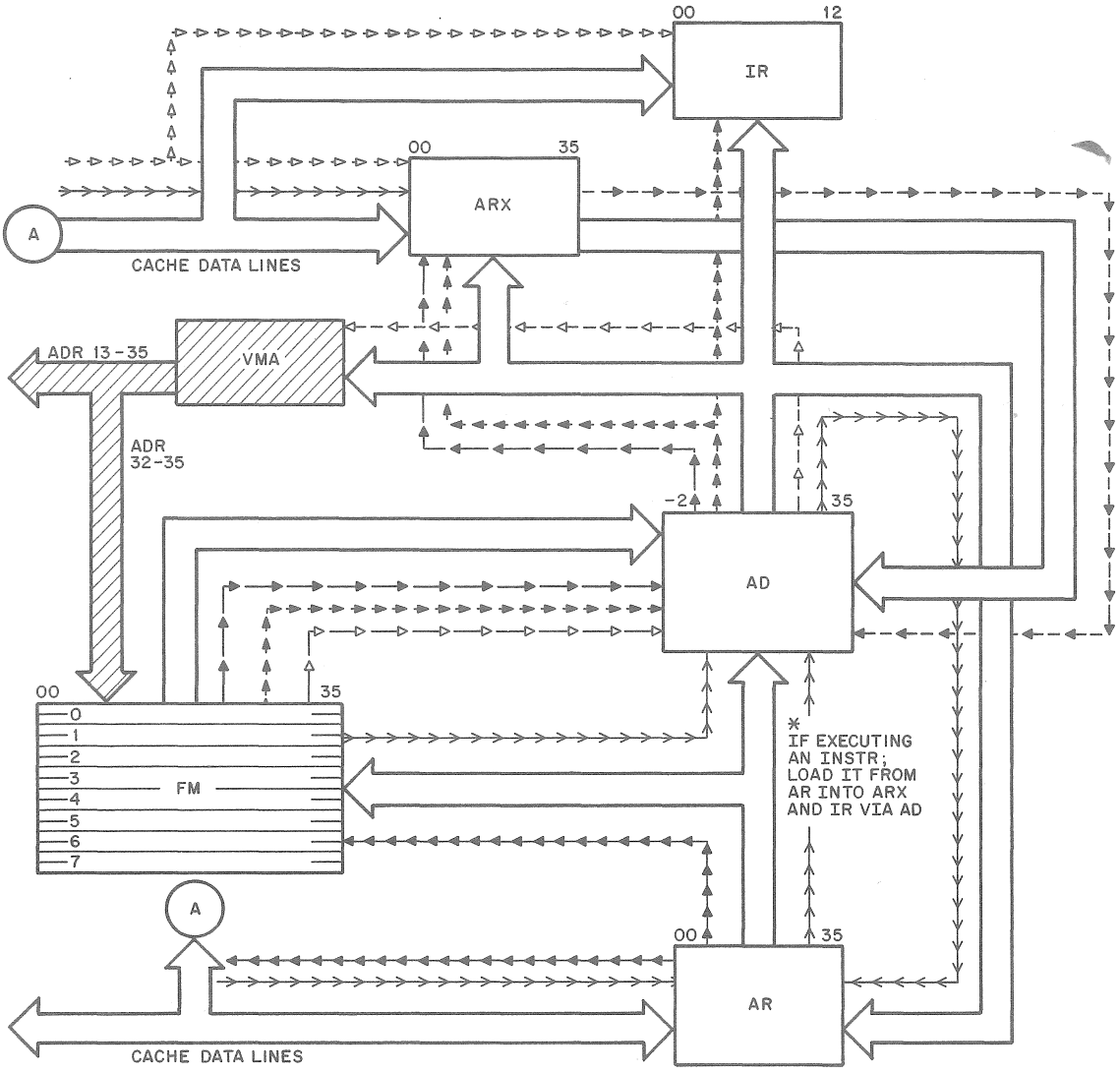



Figure 1-34 KL10 Register Interconnection Diagram




LEGEND:

- INSTR FROM CORE*
- INSTR FROM FM*
- (INDEX REGISTER)
- @ WORD FROM FM
- INDEXED ADDRESS
- Y FIELD OF CURRENT INSTRUCTION
- DATA READ OR E
- DATA WRITE

NOTES:

1. * = Exceptions are: XCT
2. Parity logic is not included on this drawing.

10-2183

Figure 1-35 Core and Fast Memory Information Flow

**Table 1-3 Memory Information Flow**

| Type of Request | Type of Information | Source | Destination | Comments |
|---|---|---|---|---|
| Read | Instruction | Core Memory or Fast Memory | ARX | Loaded via cache data lines if from core memory or via the AD if from fast memory. |
| Read | Data | Core Memory or Fast Memory | AR, ARX, or both | Loaded via cache data lines if from core memory or via AD if from fast memory. |
| Write | Data | AR | Core Memory or Fast Memory | AR goes to the FM and to the cache, regardless of which reads it. |
| Read | Indirect Word | Core Memory or Fast Memory | ARX | Loaded via cache data lines if from core memory or via AD if from fast memory. |
| Read | Index Register | Fast Memory | AR, VMA | The contents of the addressed Index register is read into the ADDER "B" input where it is added to the current value of Y. The sum is loaded into both AR and VMA under microcode control. |

The microinstruction contains a number of separate fields for register selection including a 3-bit AR field and a 3-bit ARX field. In addition, three fields are provided for controlling the adder; two of these, the ADA (3-bit field) and ADB (2-bit field), select various inputs to the adder. The third field, AD (a 6-bit field), controls the adder directly. The actual selection of the source or destination registers depends on the following:

1. The microinstruction register select field function
2. The source or destination memory (e.g., fast memory or core memory).

As an example, consider an instruction fetch (not a prefetch) from fast memory. Refer to Figure 1-36. The MEM field function of the microinstruction desiring the word is coded as FETCH. From this, the term MCL LOAD ARX is produced and routed to EBox Control No. 1, where it partially enables the ARX SELect 1 and ARX SELect 2 Mixer Selection logic. The final selection is a function of the address contained in VMA. If this address is a fast memory address (e.g., VMA 13–31 = 0), then the ARX SELect 2 line is fully enabled and the ARX SELect 1 line is inhibited by VMA AC REF. Similarly, if the address in VMA is a core memory address, VMA AC REF will be false, inhibiting the ARX SELect 2 line and enabling the ARX SELect 1 line.

As indicated in Figure 1-36, there are eight inputs to the ARX. The microinstruction may select any of these eight inputs, if required, simply by coding the ARX field appropriately. The AR and its associated mixer are very similar to the ARX. In the case of reading a word of data into AR from core memory, the MEM field function, LOAD AR, is latched into the request qualifier register in the memory control, partially enabling the AR mixer select 11 and select 2 lines to the AR mixer. Once again, the selection is a function of the address in VMA. If bits 13–31 of the virtual address are equal to zero, the adder is enabled into the AR number 2 input, but if the address in bits 13–31 of VMA is nonzero, the cache data lines are enabled into the AR number 2 input. As with ARX, the microinstruction may select any of the eight inputs on the AR mixer, if required. Figure 1-37 is a simplified version of the EBox data paths. The basic path connections and the direction of transfers are indicated. Along the bottom of the figure is the portion of the microinstruction word format that controls the data path. The simplified path does not show shift left or shift right connections.

**1.2.6.2 Information Flow I/O and Priority Interrupt** – Figure 1-38 is a simplified path diagram used by I/O and priority interrupt operations. The major path is the shaded area, including the AR, adder, EBus, translator external or internal devices, and MQ. The portion that is cross-hatched may be generically called the "inspection and control path" and includes the SH, SC, SCAD, FE, and CRAM address logic. The remaining paths and registers are used as working registers; the usage depends on the specific operation.

Note that internal device information flow (control data) is not translated, but rather utilizes the internal ECL EBus. External device information, however, entering or leaving the EBox, must be translated in the direction TTL to ECL or ECL to TTL. If the operation being performed is a CONI or DATAI, the destination register is AR. If the operation is CONO or DATAO, the source is AD. The processing of interrupts is more complex. The destination for the API function word is initially AR, but the function performed in response to the decoding of this word may involve an instruction fetch, a data read and write, a data out, or a data in operation. The microprogram begins to process the interrupt when the AR contains the API function word transmitted from device and the EBus handshake has been completed.

The microprogram places a copy of this word into MQ for use later and performs a SHIFT Dispatch on the API function code to the appropriate routine in the microprogram. To implement this dispatch, the AR is enabled into the shift matrix; then the output bits (SH 00–03) are sampled in the CRAM Address Control logic. In addition, another type of dispatch can be performed; this is called AR 00–03 Dispatch.

When the API function specifies a standard interrupt (API FCN 0 or 1) an instruction is fetched from 40 + 2n, where n is equal to the interrupting channel 1–7. These interrupt locations generally contain a JSR instruction that must be performed in order to preserve the flags and PC of the interrupted program. In addition, the current ACs must not be disturbed and the interrupt handler (monitor routine) must be entered for polling of devices. In these situations, the microcode forms the correct address in VMA (40 + 2n) and begins an instruction fetch by issuing a microinstruction with MEM equal to FETCH. This fetch is from the Executive Process Table (EPT) and requires that the request qualifier, EBox EPT, be asserted in order that the MBox access the EPT for the instruction.

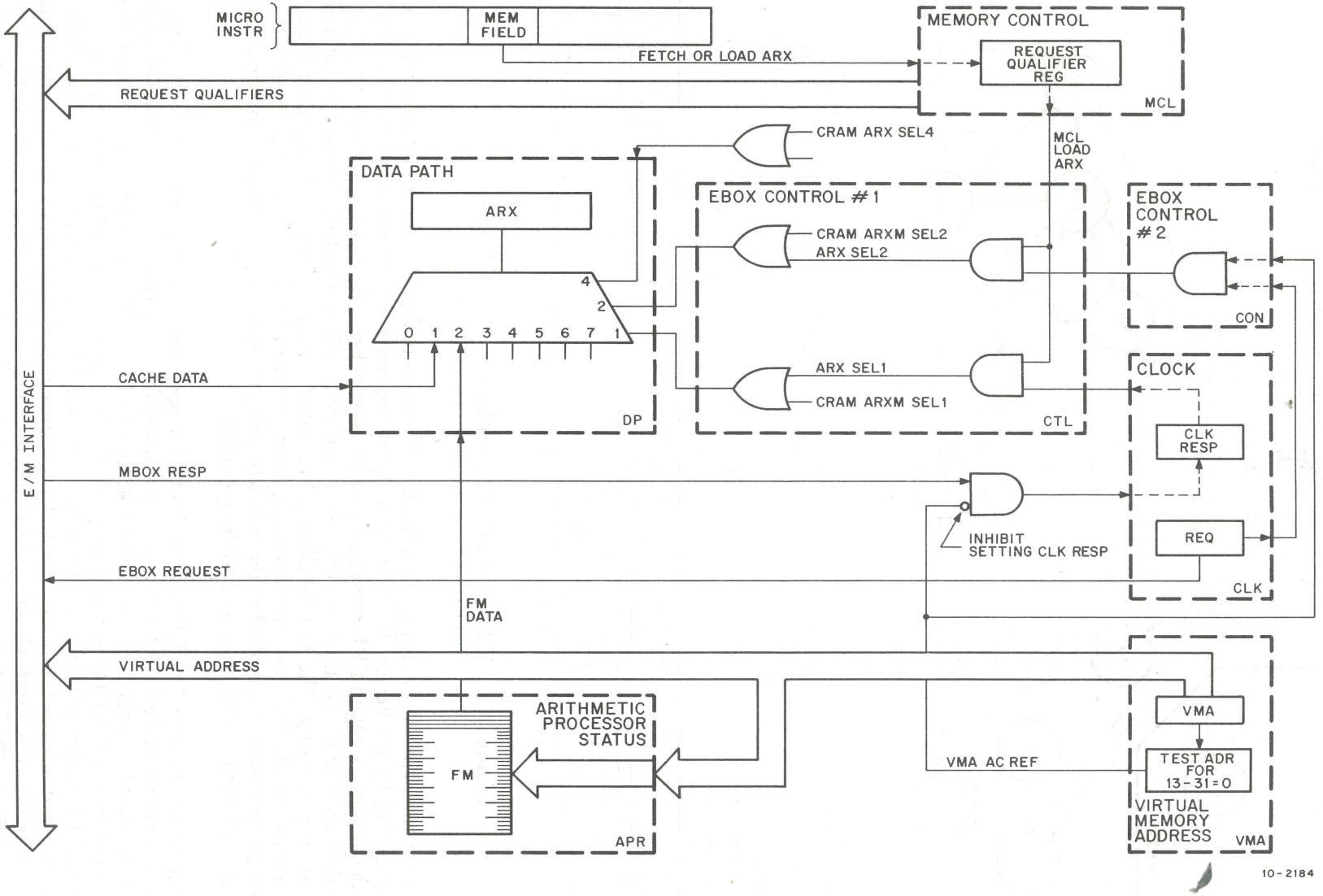



Figure 1-36 Loading ARX

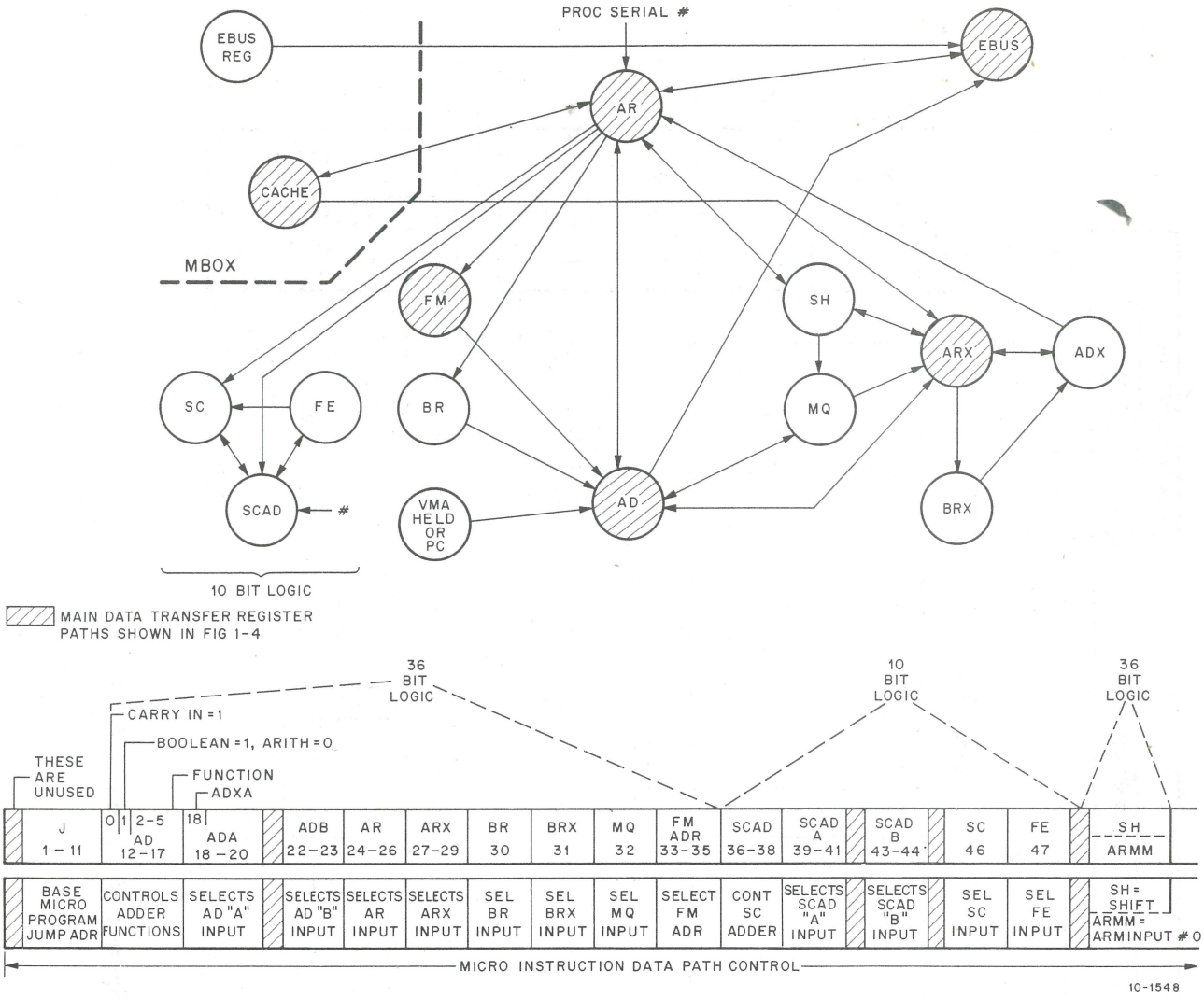



Figure 1-37 EBox Data Paths Simplified Paths Diagram

When the API function specifies a dispatch (API FCN 2), the virtual address of an interrupt instruction (JSR) is provided by the device. In this situation, the request does not assert the qualifier EBox EPT because the address is not an EPT address, but rather somewhere in the virtual address space. For the situations described up to this time, the instruction will enter ARX. Control is passed to the main microcode loop for processing. The API function (PI increment or PI decrement) is slightly different, in that a word must be fetched from the virtual address provided by the device. This word is then incremented or decremented as specified in the API word and the result is written back into memory. Here the AR is used both for the read and write operations.

API functions 4 and 5 require a DATAO and a DATAI, respectively, to be performed to the device. Prior to performing the specified DATAO, a word is fetched from the virtual address provided in the API word and this word is loaded into AR. The path is now from AR to AD and then to the EBus, which is controlled for the DATAO by the microcode. For the specified DATAI, the operation is the reverse. The required word is obtained from the device via the EBus under microcode control (EBus dialogue) and the word is loaded into AR. Next, the contents of AR must be written into the virtual address supplied by the API word. Of the remaining functions, only API FCN6 is used and this is reserved for the DTE20 (10-11 Interface). Examines and deposits, as well as byte transfers, may be requested by the DTE. This subject is covered in Section 2.

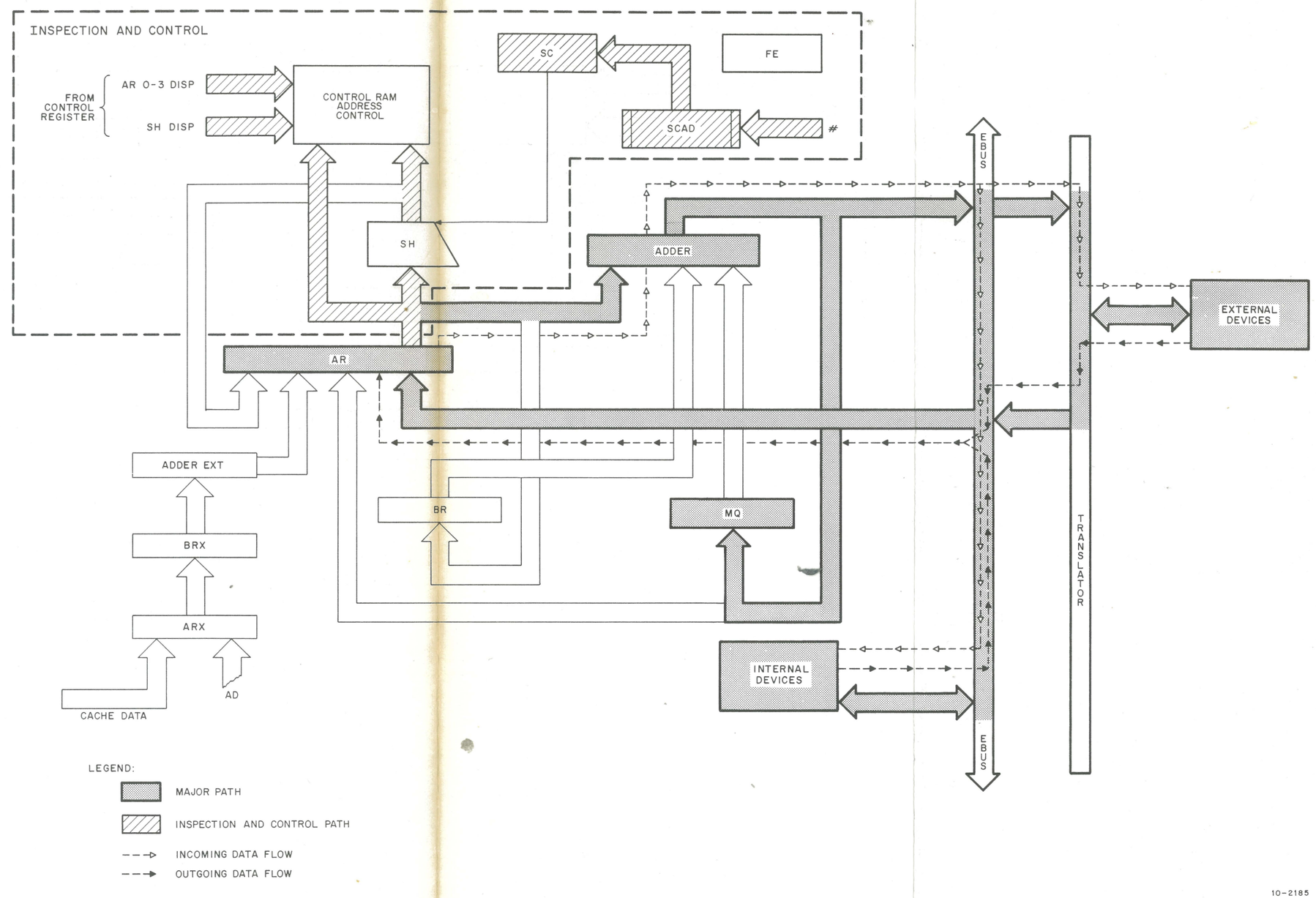



Figure 1-38 Input/Output Priority Interrupt Information Flow

# 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

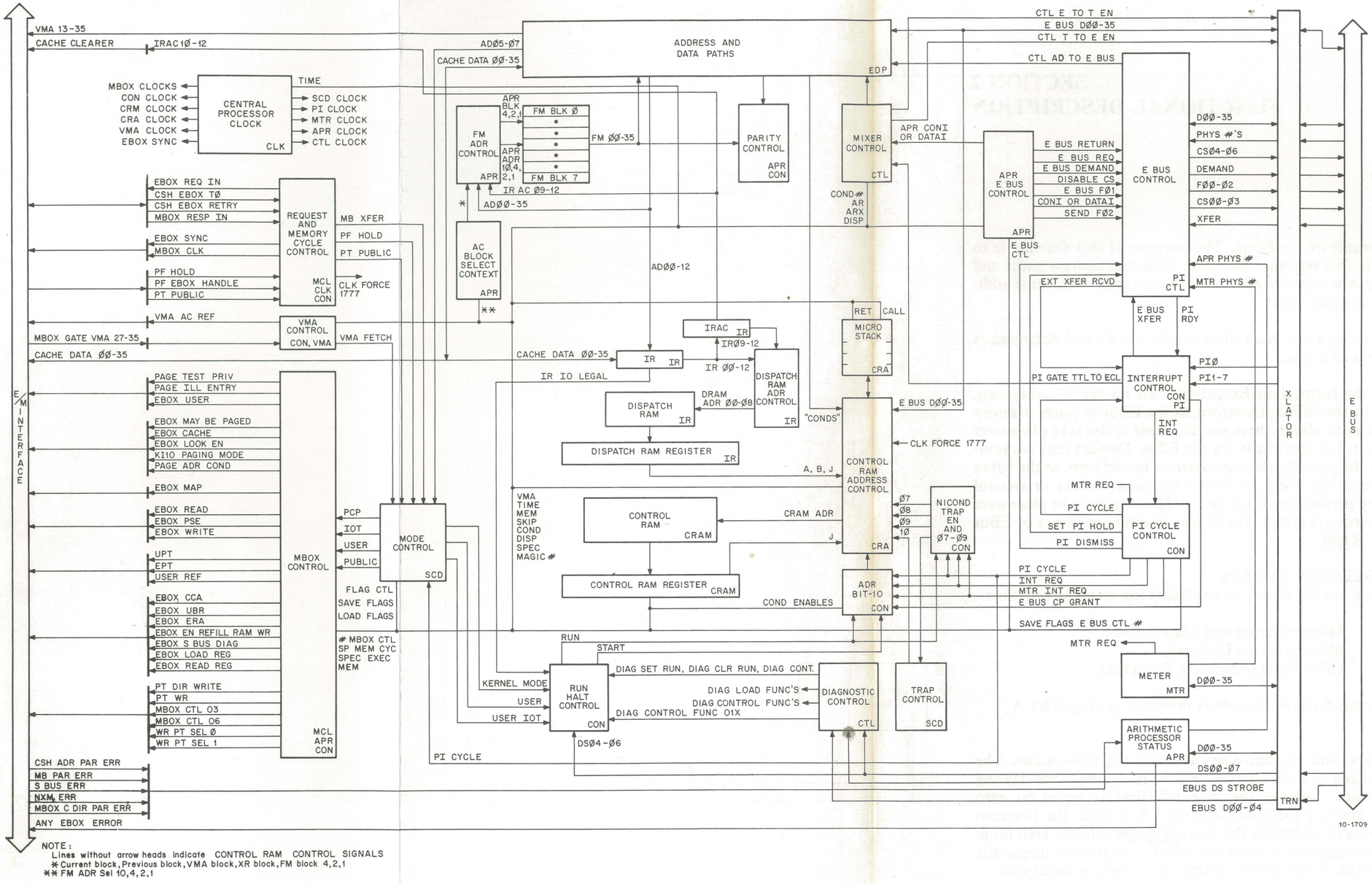
Referring to Figure 2-3, the EBox microprogram can be in one of the following states at any time:

- 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

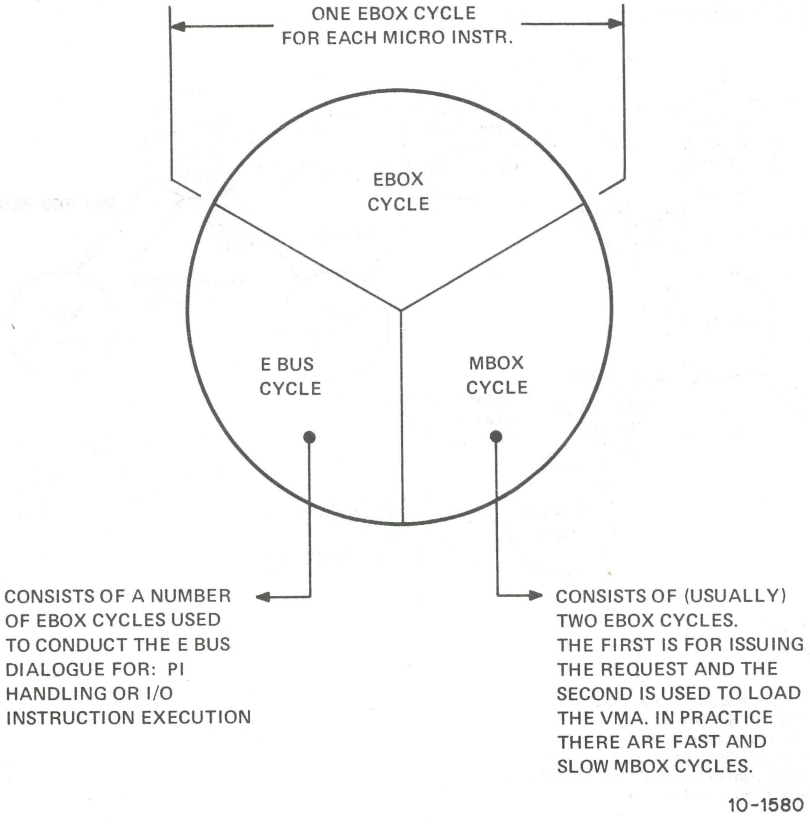



Figure 2-2 Primary Hardware Cycles

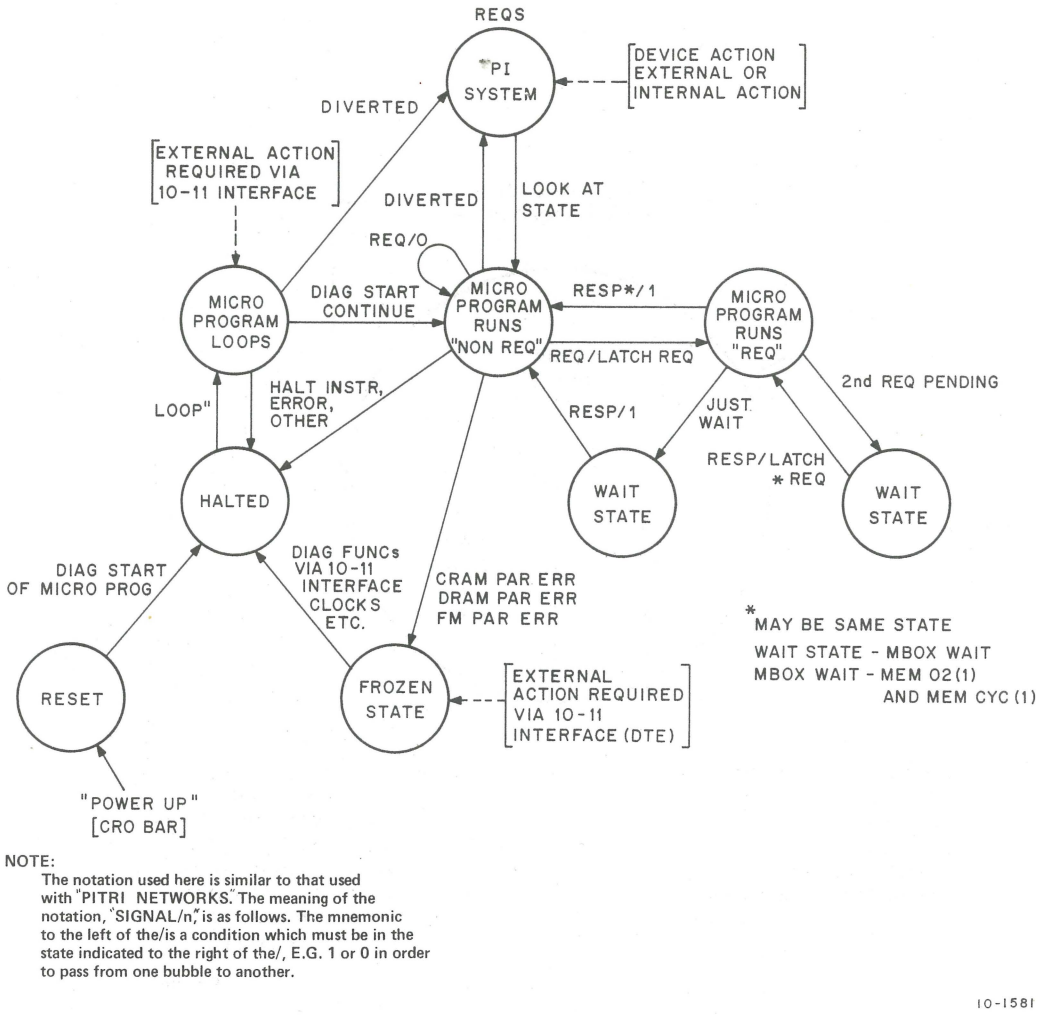



Figure 2-3 Microprogram Static States

### 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.

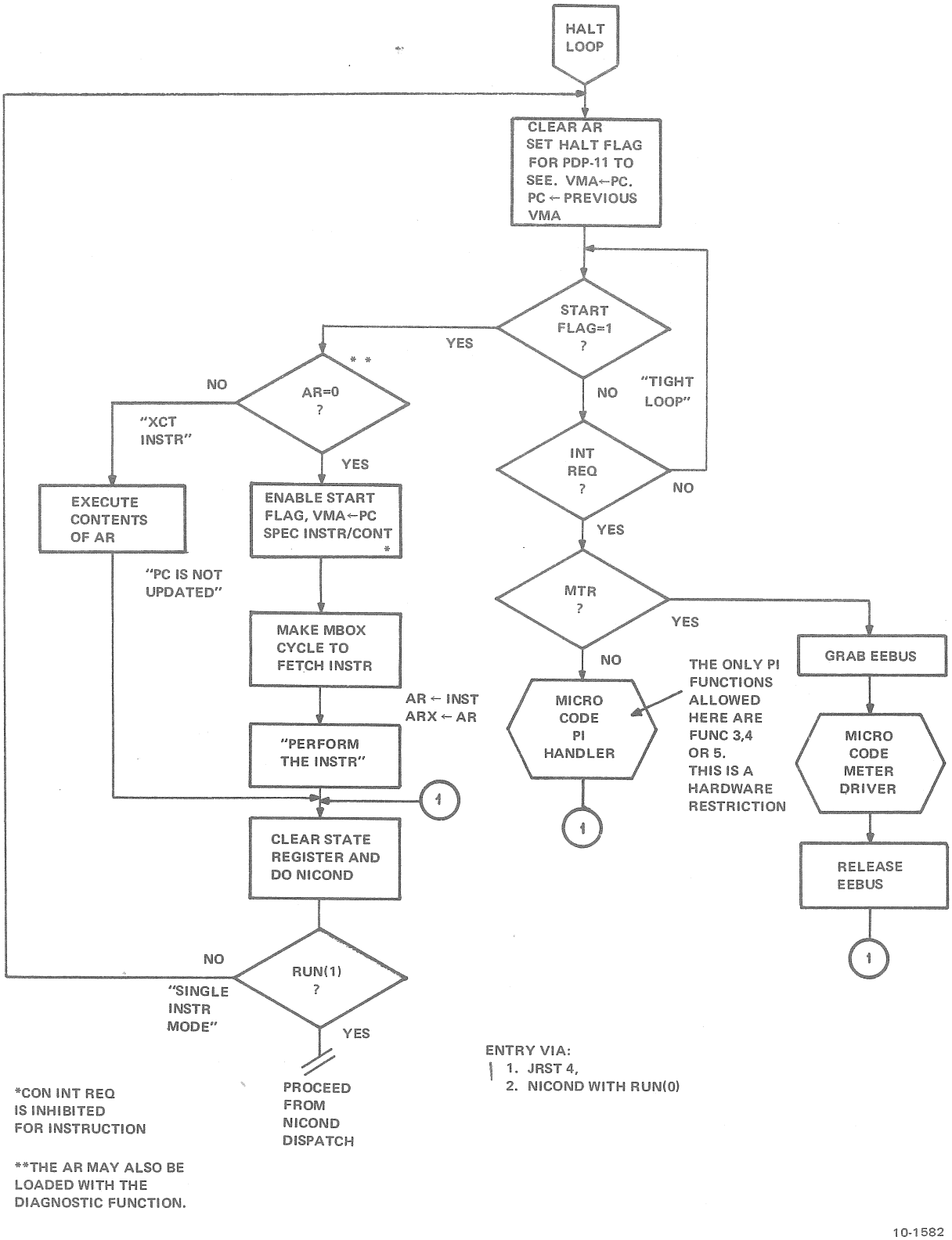



Figure 2-4 Microprogram Halt Loop

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).

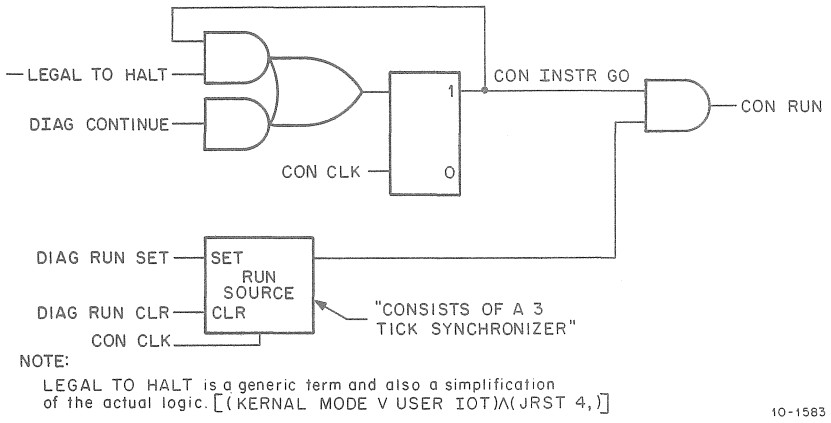



Figure 2-5 Run-Halt-Continue Logic

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.**

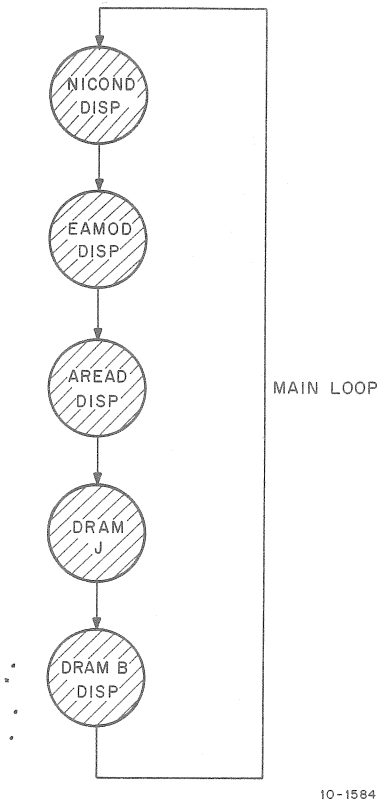



Figure 2-6 Dispatch Path State Diagram

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.

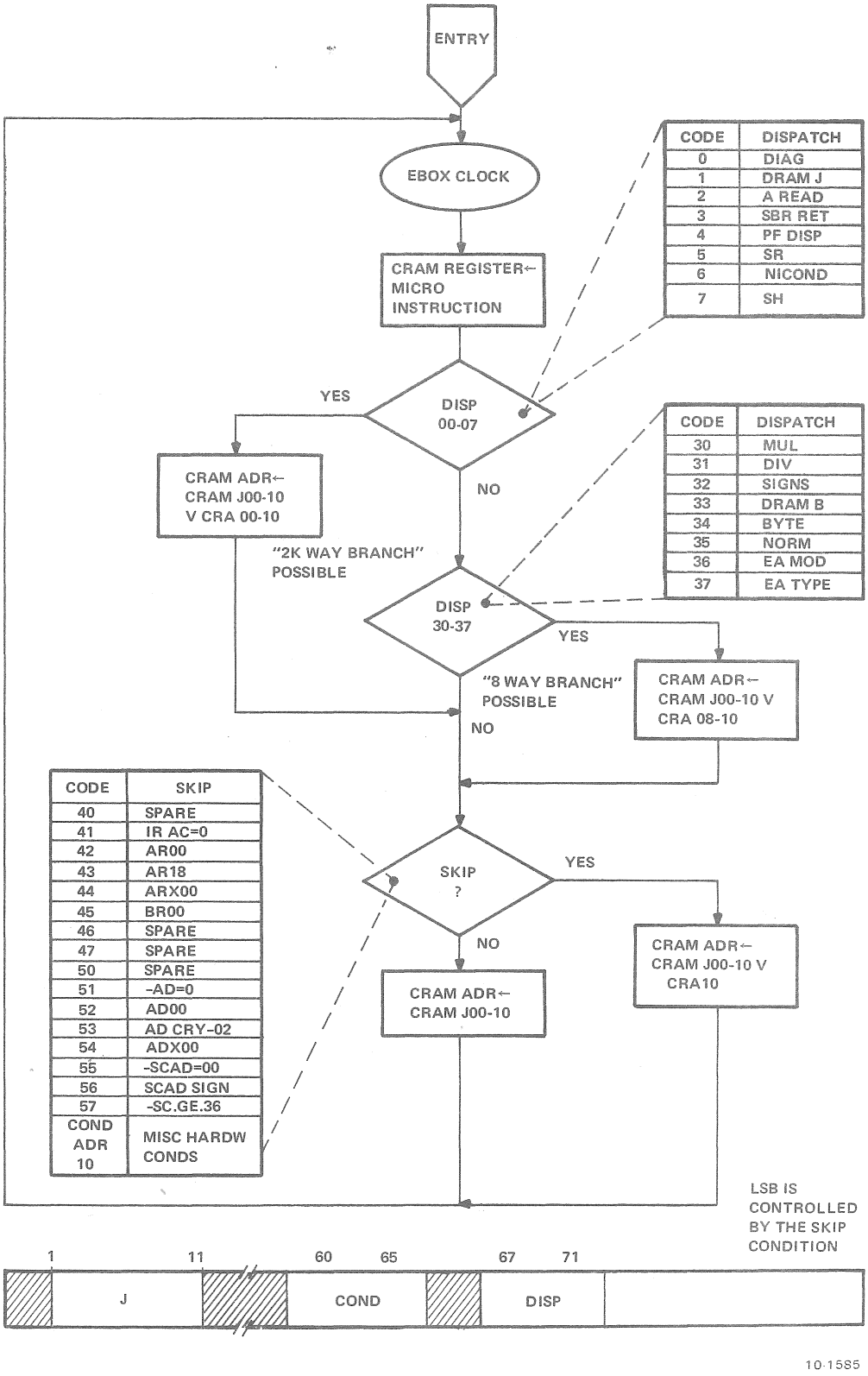



Figure 2-7 Basic Microprogram Address Control

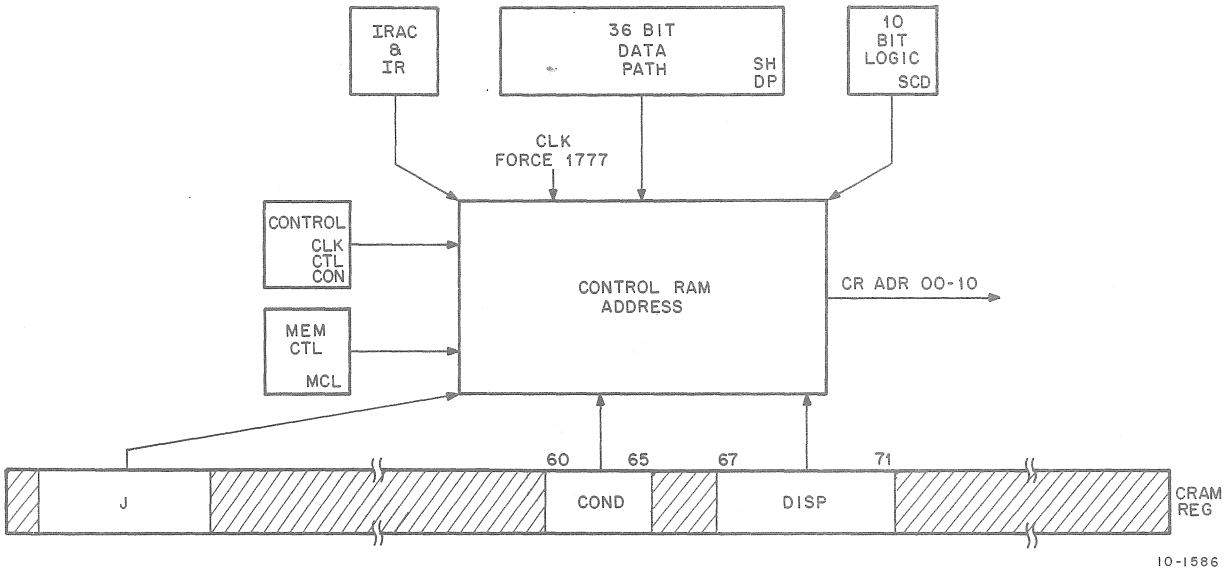



Figure 2-8 CRAM Address Inputs Simplified

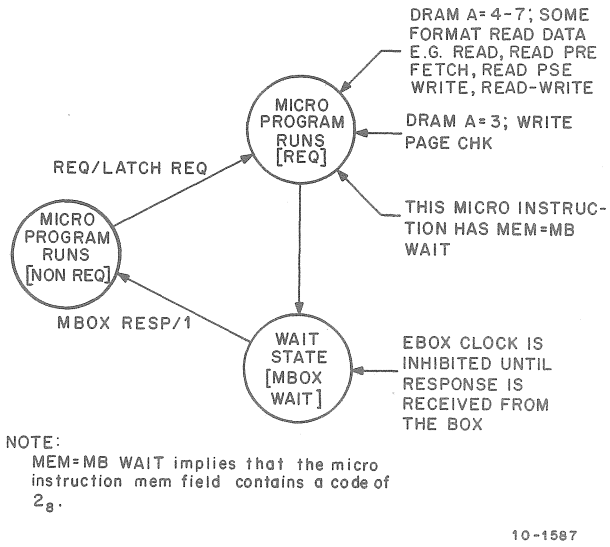



Figure 2-9 Wait State

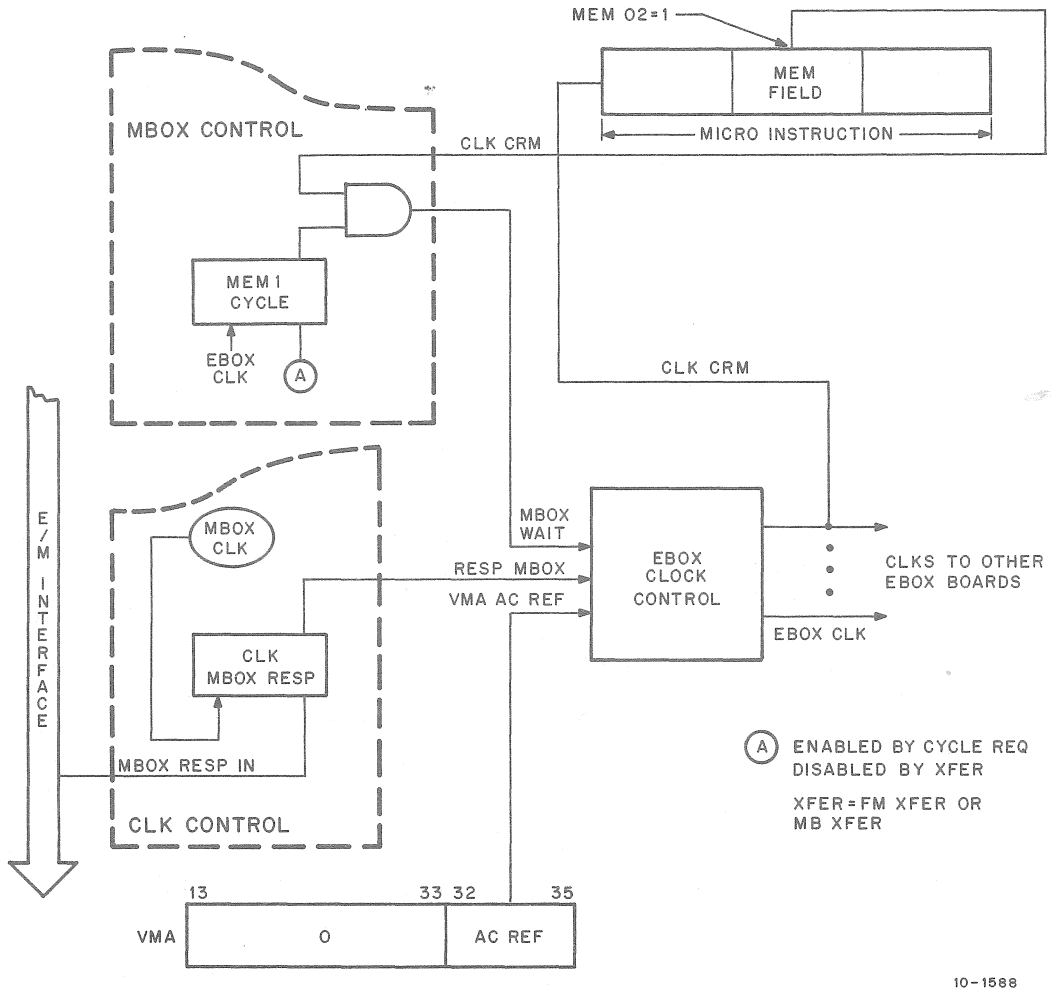
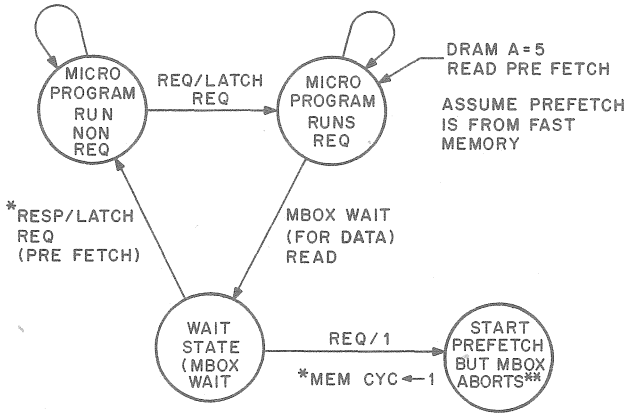



Figure 2-10 MBox Wait and EBox Clock




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: PI CYCLE | RUN | MTR INT REQ | INT REQ | AC REF | TRAP EN | ANY TRAP | Low-Order CRAM ADR Bits as Follows: 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

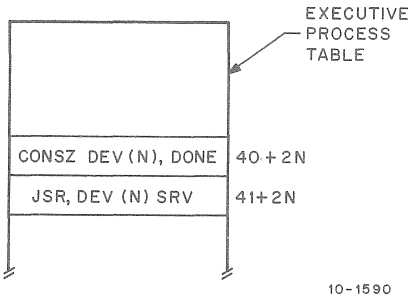



Figure 2-12 PI 40 + 2n Skip

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).

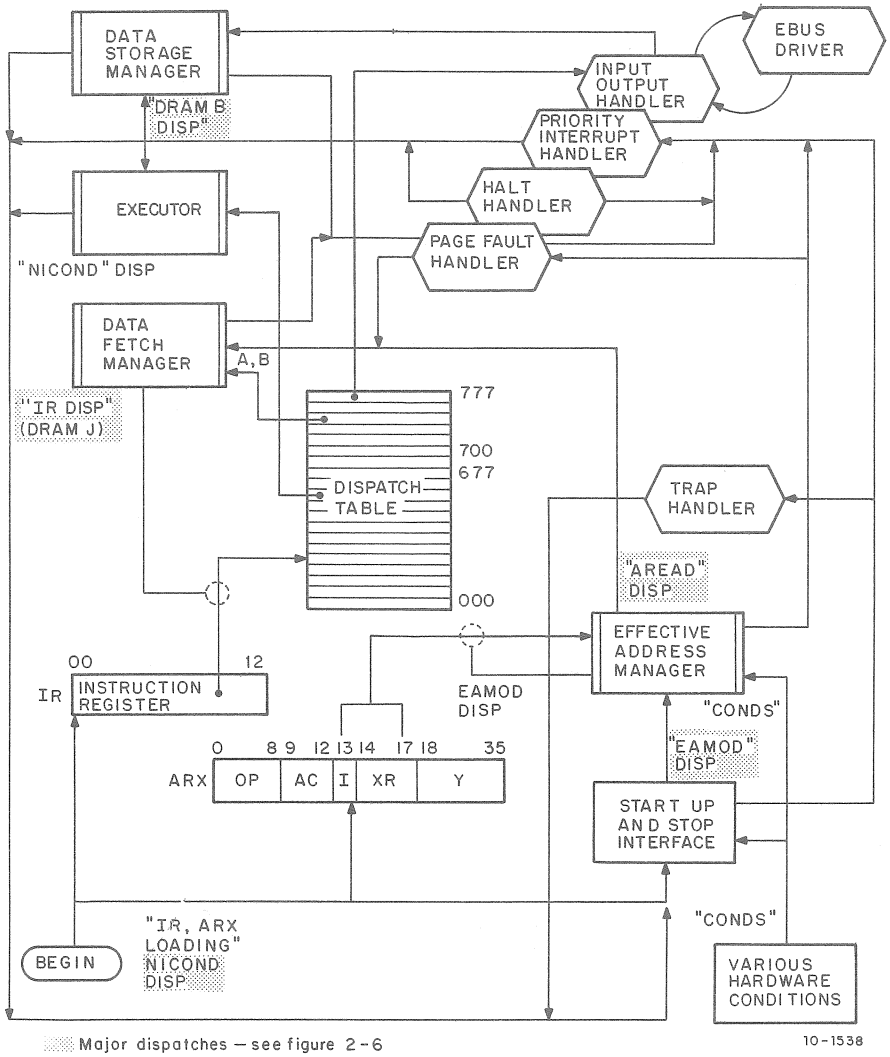



Figure 2-13 M Program Modules

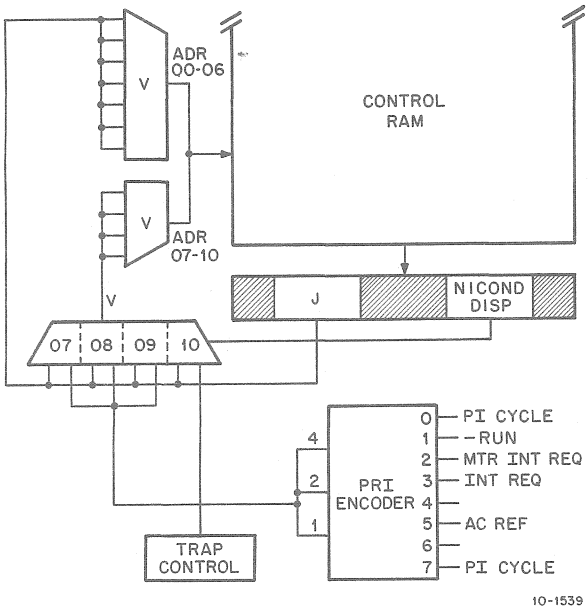



Figure 2-14 Startup and Stop Interface

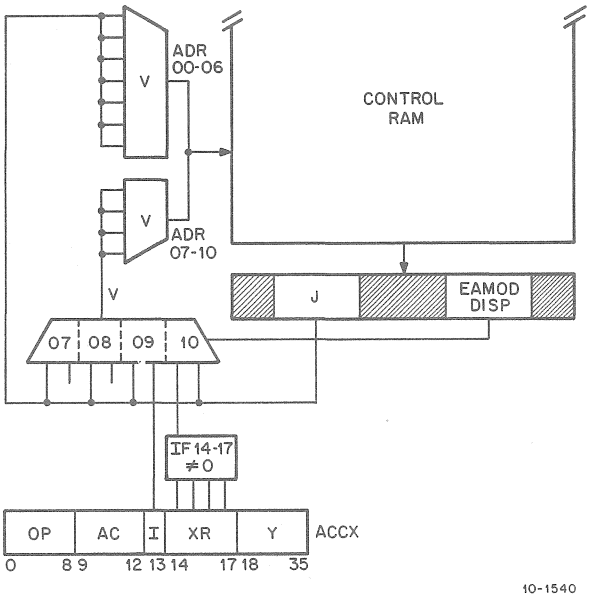



Figure 2-15 Effective Address Manager

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.

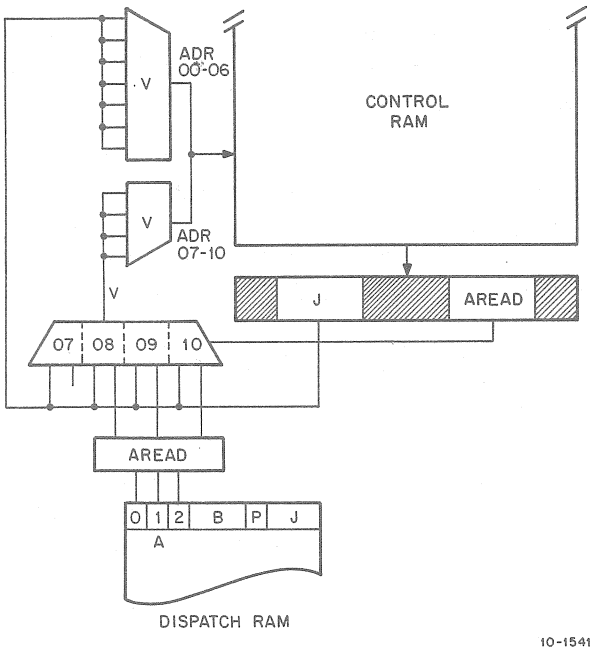



Figure 2-16 Data Fetch Manager

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).

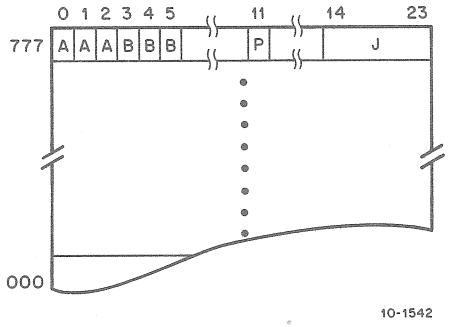



Figure 2-17 Dispatch Table Fields

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.

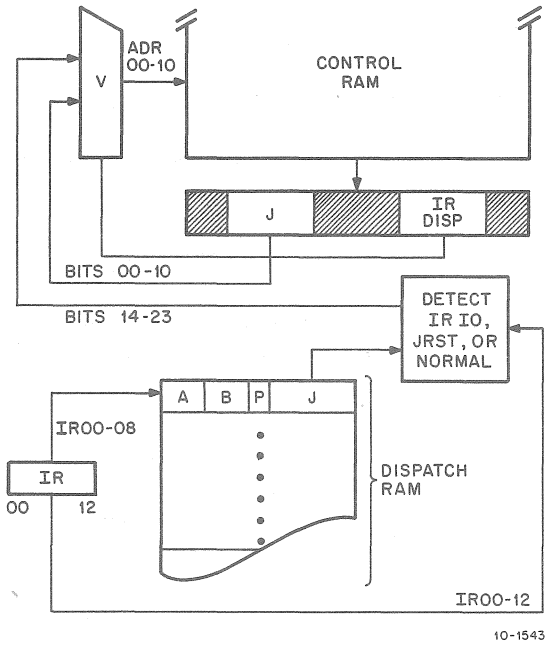



Figure 2-18 Executor

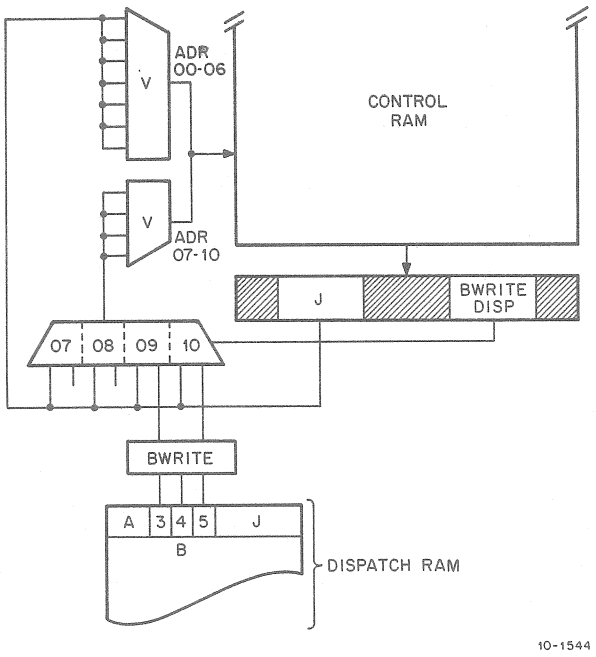



Figure 2-19 Data Store Manager

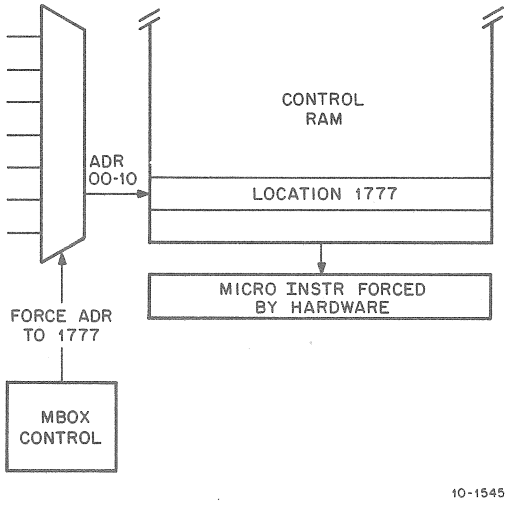



Figure 2-20 Page Fault Handler

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 |

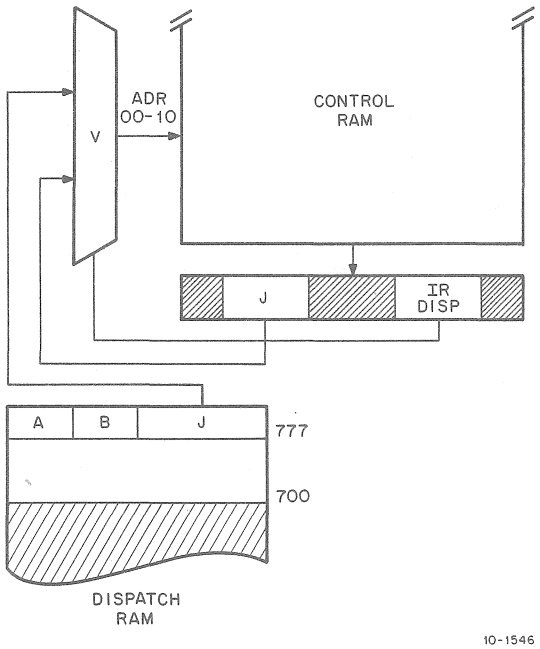



Figure 2-21 Input/Output Handler

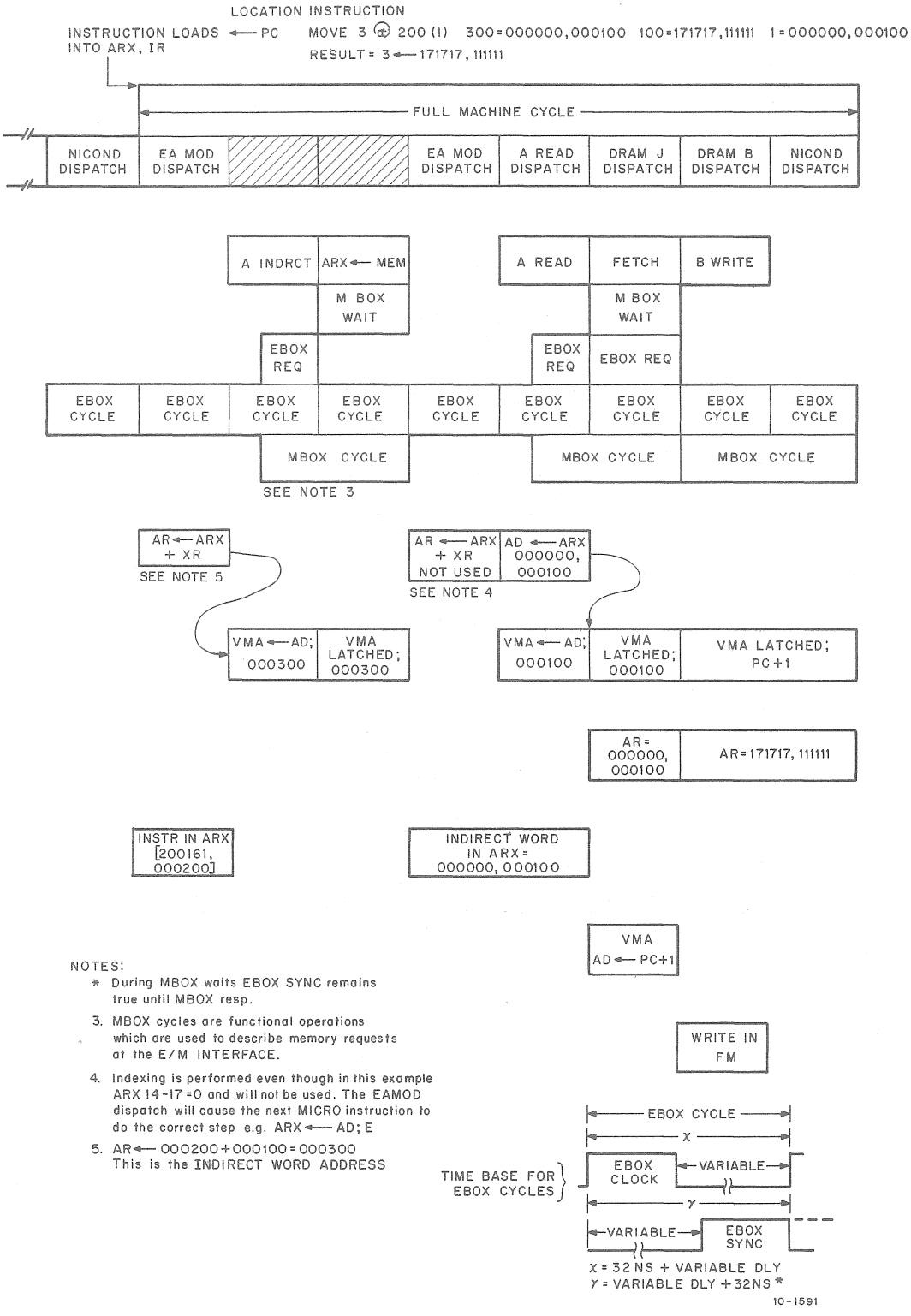



Figure 2-22 Basic Machine Cycle Overview (Sheet 1 of 2)

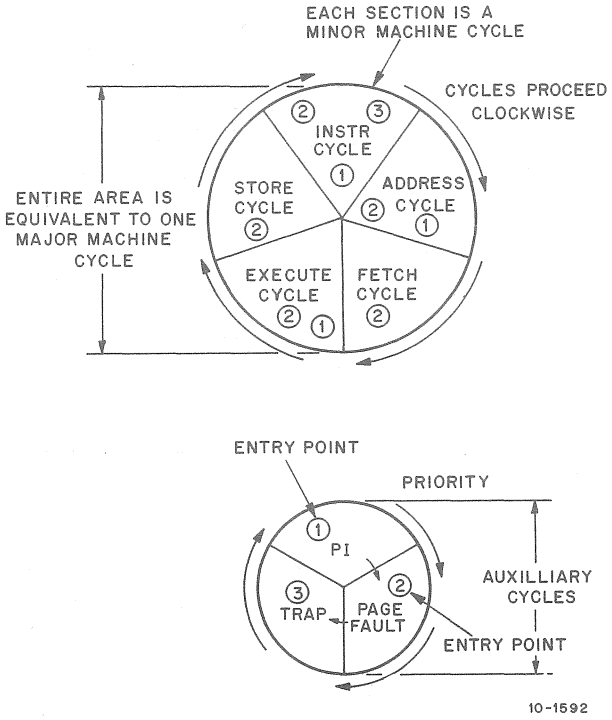



Figure 2-22 Basic Machine Cycle Overview (Sheet 2 of 2)

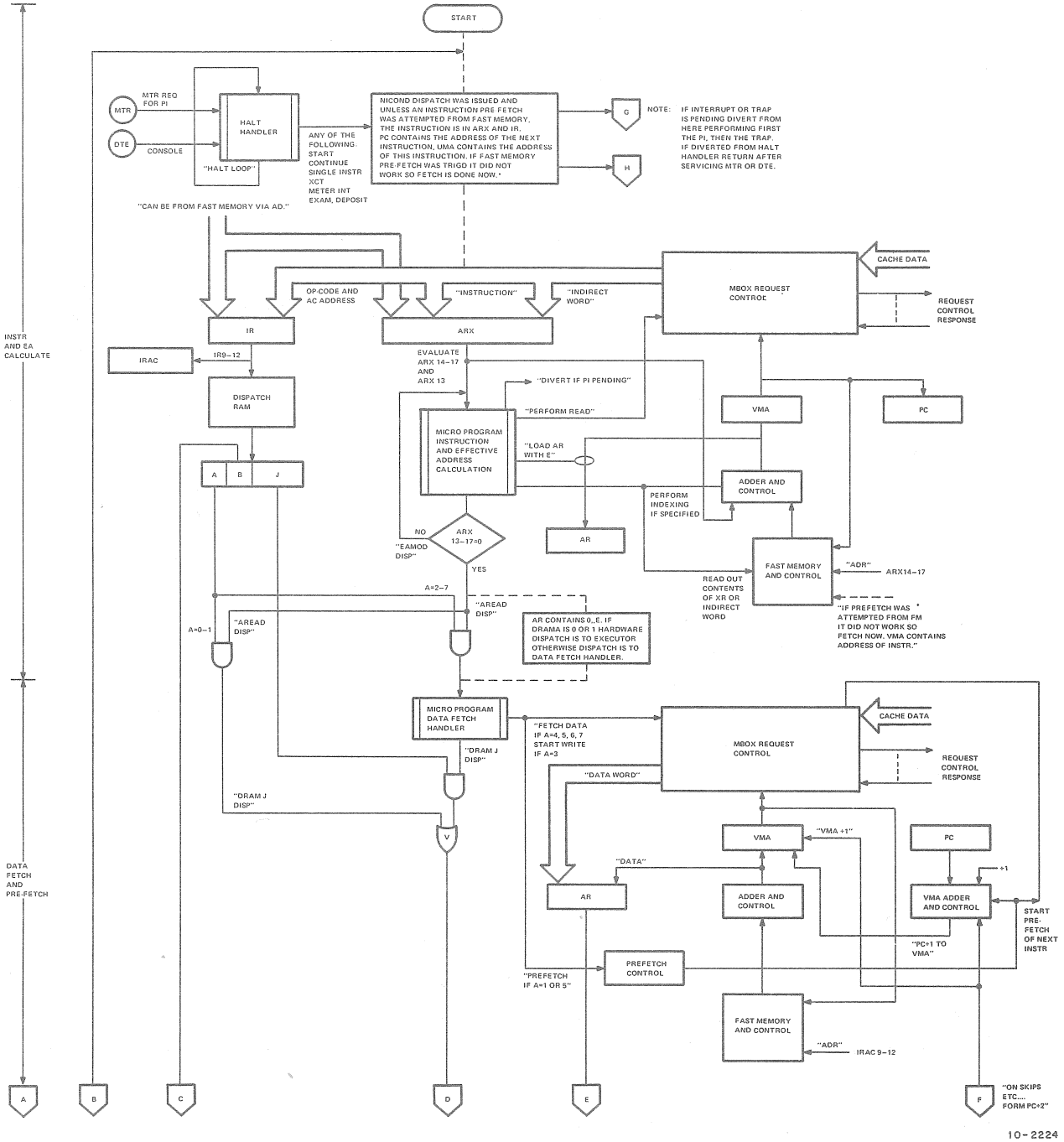



Figure 2-23 KL10 Processor Sequence of Operation (Sheet 1 of 2)

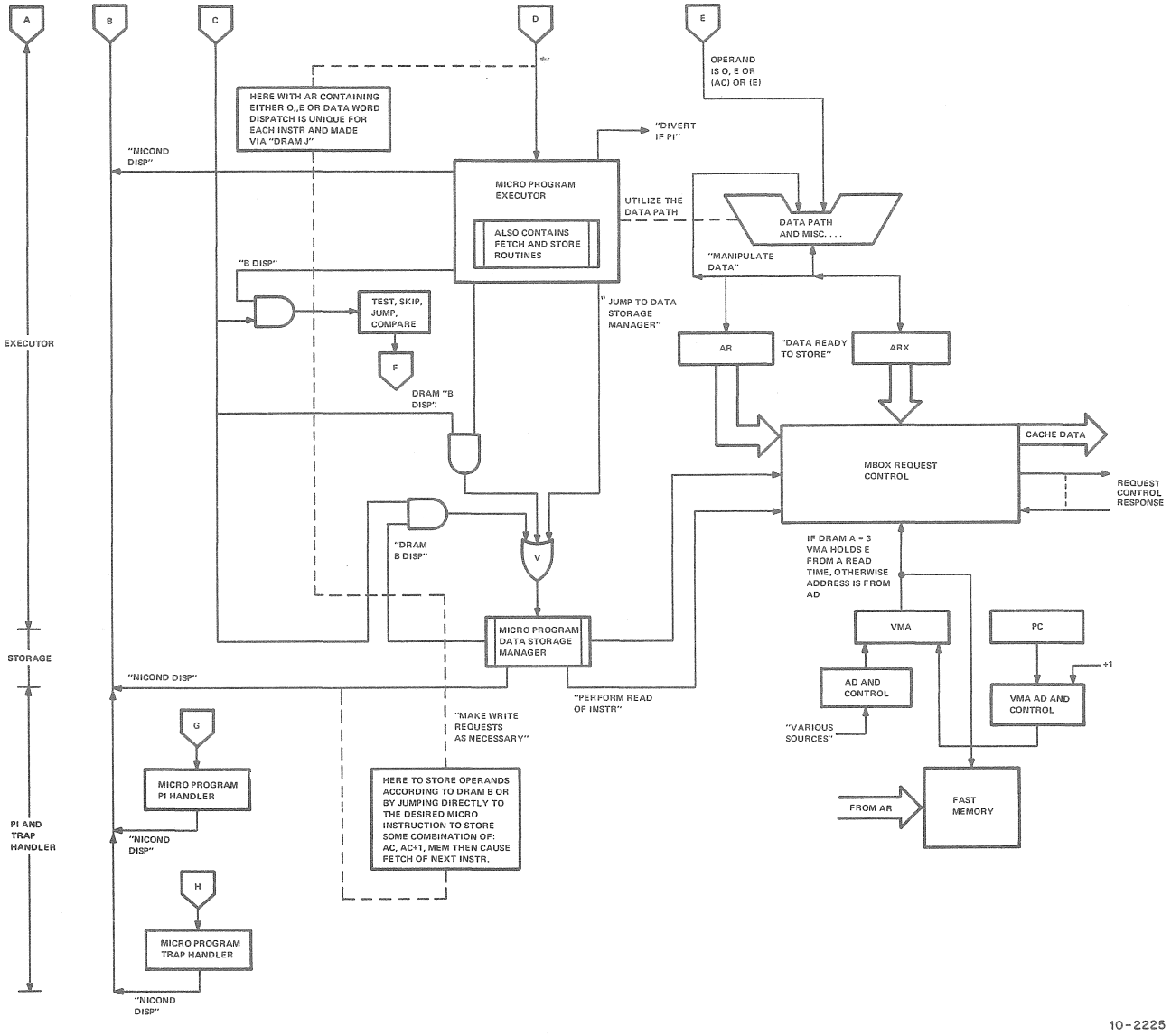
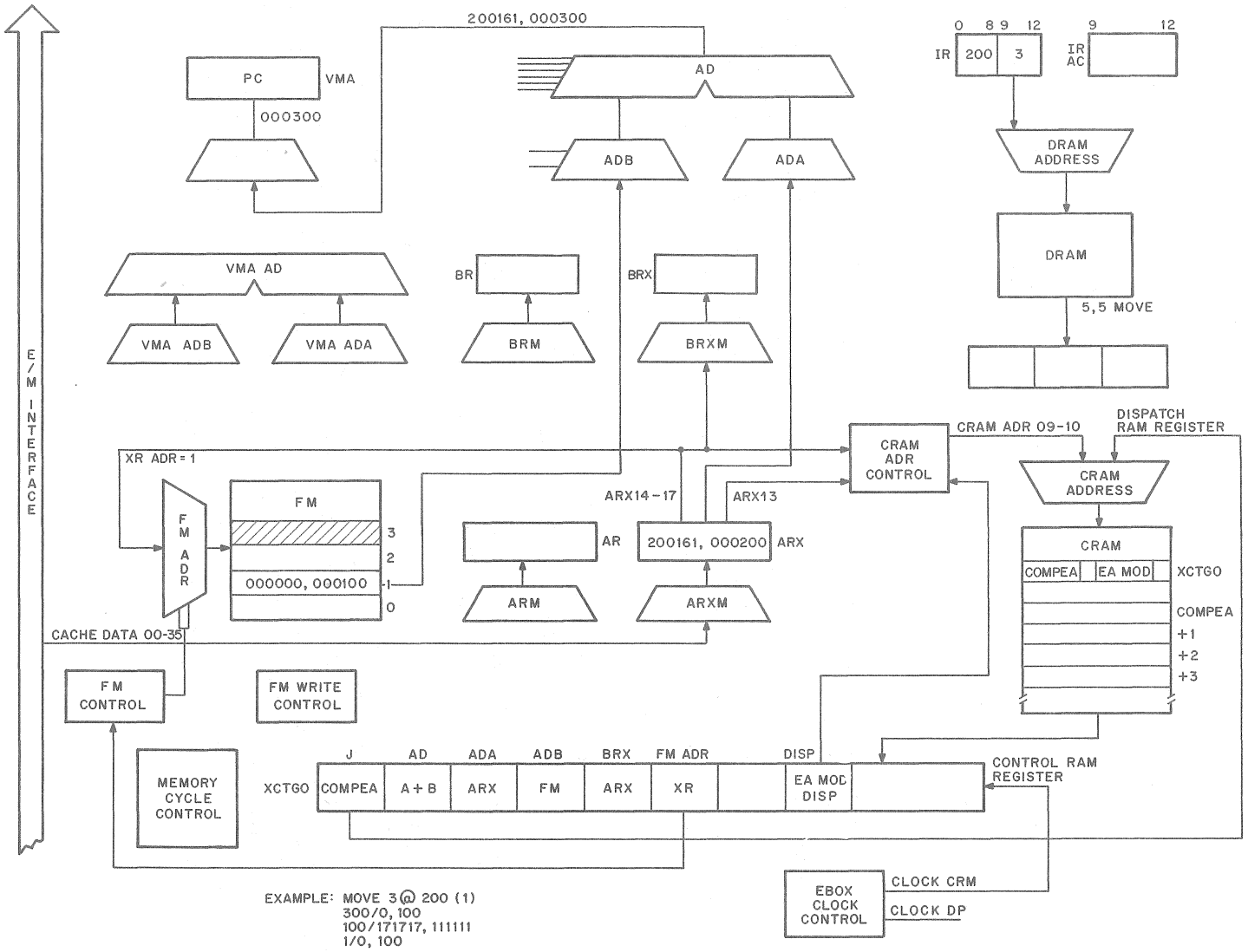



Figure 2-23 KL10 Processor Sequence of Operation (Sheet 2 of 2)

Figures 2-24 through 2-33 illustrate the microprogram steps and basic EBox hardware used to perform the example instruction. Figure 2-22 can be used to follow the various operations at each micro-instruction step.

### 2.3.1 Instruction Cycle – NICOND Dispatch to XCTGO

The instruction enters the ARX through the ARX mixer (ARXM) via the cache data lines. Although not shown, the MBox response enables the mixer selection and the EBox clock (CLK DP) loads the ARX on the Data Path Board with the instruction. The NICOND Dispatch for this example is to symbolic location XCTGO; Figure 2-24 indicates the major microinstruction fields. The Jump address contains the base address of a 4-word block used to calculate the effective address. Each micro-instruction in this block is used for a different form of address calculation, and is selected based upon the state of ARX14–17 and ARX13 when EA MOD DISPATCH is given. The EBox hardware utilizes ARX14–17 and ARX13 to modify bits 09–10 of the CRAM address. This yields the possibilities listed in Table 2-4.




NOTE:
This operation reflects the micro code version 71.

10-1593

Figure 2-24 Instruction Cycle: NICOND Dispatch → XCTGO

Table 2-4 Address Calculation

| CRAM Address | ARX14–17 | ARX13 | Function |
|---|---|---|---|
| COMPEA | 0 | 0 | ARX = E |
| COMPEA+1 | Nonzero | 0 | Perform indexing as specified by ARX14–17. |
| COMPEA+2 | 0 | 1 | Perform indirection VMA ← ARX18–35 |
| COMPEA+3 | Nonzero | 1 | Perform indexing as specified by ARX14–17, then perform indirection VMA ← ARX18–35 + (XR) |

While at XCTGO, to speed things up, the indexing operation is started. The fast memory address field in the microinstruction causes the FM control to address fast memory utilizing ARX14–17, which in the example is 1. The ADA input is enabled to select the ARX as input to the ADDER A input. This is controlled by the microinstruction ADA field. Similarly, the ADB field enables the ADB input to select addressed FM location 1. The microinstruction AD field specifies the ADDER function as A+B. Thus, the ADDER begins to add the contents of location 1 in fast memory to the instruction in ARX. At this time, the Buffer register extension is enabled from ARX by the microinstruction BRX field.

NOTE

**The IR contains the op code of the instruction MOVE, which is 200, and the AC field, which is 3.**

The op code value (200) is used to address the DRAM to obtain the appropriate word for this instruction. This word is indicated on the input to the DRAM register (5,5,MOVE).

### 2.3.2 Indirect Word Request

For an Indirect Word request, the CRAM register contains the microinstruction fetched from COMPEA+3 as indicated in Figure 2-25. The Jump address now specifies a direct jump to symbolic location INDRCT. The AD, ADA, ADB, and FMADR fields are maintaining the indexing calculation and the calculated address 000300 is forming at the input to the VMA. The MEM microinstruction field is coded as A IND. This enables the memory cycle control to set up and generate an MBox cycle (Figure 2-26). This begins with the assertion of EBOX REQUEST IN, together with the qualifier EBOX READ. Table 2-5 lists the MEM field function that generates requests. An IND is a function that may be followed by a microinstruction having the MEM field coded as MB WAIT.

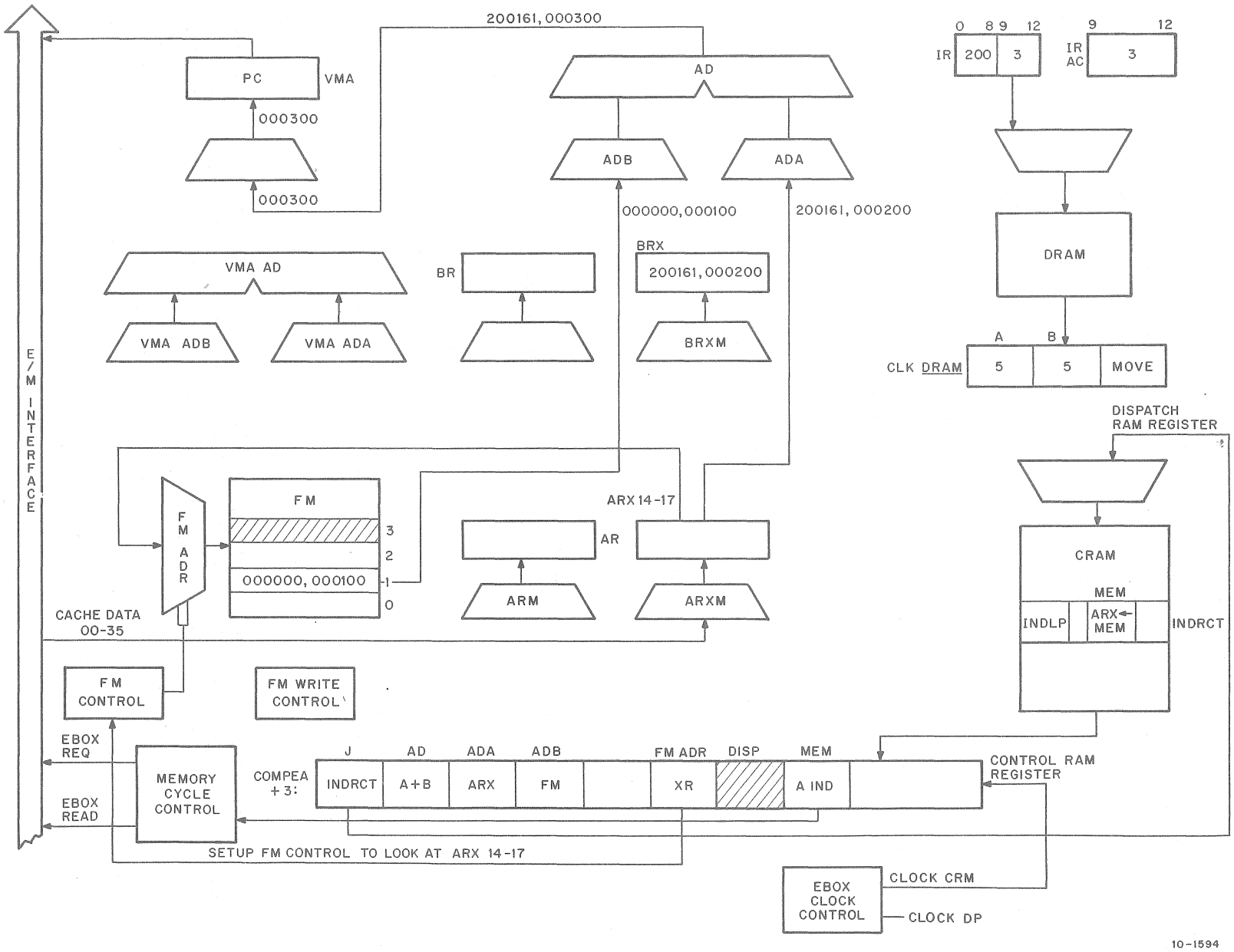



Figure 2-25 Set Up and Make Indirect Work Request

**Table 2-5 MBox Cycle Requests**

| MEM 02 | MEM Field | MEM 00 | Function | Causes | MBox Wait |
|---|---|---|---|---|---|
| 0 | 04 | 0 | A READ | Fetch Cycle | No |
| 0 | 05 | 0 | B WRITE | Store Cycle | No |
| 1 | 06 | 0 | FETCH | Instruction Fetch | Yes |
| 1 | 07 | 0 | REG FUNC | MBox register reference | Yes |
| 0 | 10 | 1 | A IND | Indirect reference during effective address calculation | No |
| 0 | 11 | 1 | BYTE IND | Indirect reference for byte instruction special | No |
| 1 | 12 | 1 | LOAD AR | Data read during execution, loads into AR | Yes |
| 1 | 13 | 1 | LOAD ARX | Data read during execution, loads into ARX | Yes |
| 0 | 14 | 1 | AD FUNC | Not used | No |
| 0 | 15 | 1 | BYTE RD | Data read during byte execution loads into AR and ARX | No |
| 1 | 16 | 1 | WRITE | Store data during execution, writes from AR | Yes |
| 1 | 17 | 1 | RPW | Initiates a read PSE write cycle, data loads into AR | Yes |

The time field for the microinstruction at location COMPEA+3 specifies a period between the EBox clock that loaded the microinstruction from COMPEA+3 and the next EBox clock. It allows sufficient time for the access of fast memory to be completed. Note that EBox request and EBox sync are concurrent (Figure 2-26). The earliest time that the MBox can clear the request is on the MBox clock following EBox sync. In Figure 2-26, EBox sync occurs one MBox clock prior to where the time field indicates EBox clock can occur, but because MBox wait is true and the MBox has not yet responded, the EBox clock is postponed as indicated.

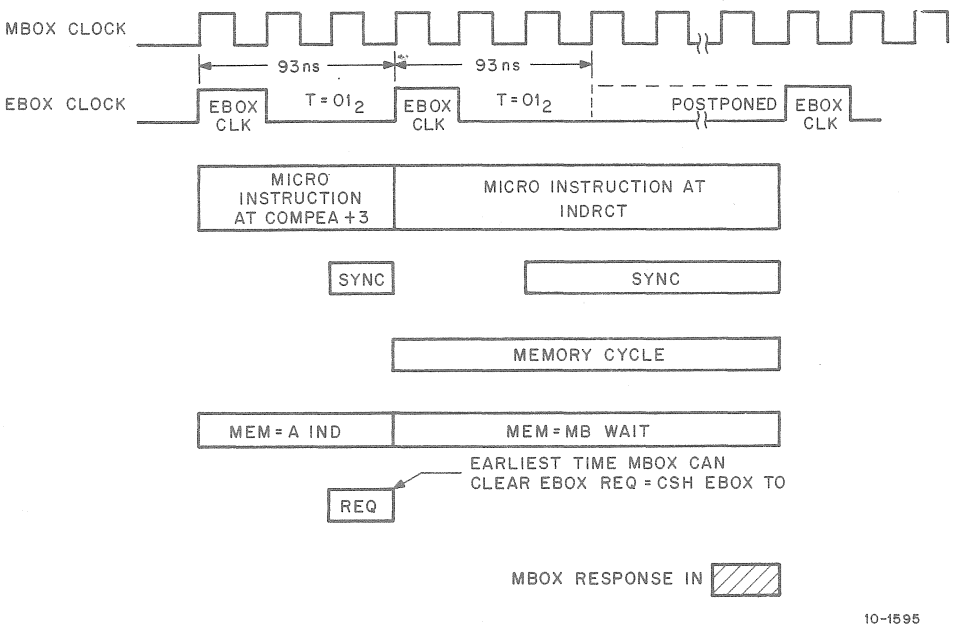



Figure 2-26 MBox Cycle

### 2.3.3 MBox Response to Indirect Word Request

Figure 2-27 illustrates the microinstruction fetched from symbolic location INDRCT. Again, a direct Jump is specified (in this instance, to INDLP). A response from the MBox is anticipated. ARX ← MEM is a MACRO statement. It specifies MEM to be MB WAIT and also selects FM as addressed by VMA 32–35. The ARXM is actually input from both AD on the 2 input and the cache data on the 1 input. The MBox response causes the EBox hardware to generate MB XFER, which selects the correct input. In this example, the cache data lines containing the indirect word 000000,000100 are loaded into ARX.

### 2.3.4 Address Calculation Continues

Referring to Figure 2-28, the CRAM register contains the microinstruction fetched from symbolic location INDLP. This setup is once again to perform indexing as though it were really specified. At this time, ARX contains indirect word 000000,000100; ARX14–17 and ARX13 are zero. Thus, even though the microinstruction specifies the calculation of indexing, the hardware calculates the proper CRAM address based upon ARX14–17 = 0 and ARX13 = 0.

The basic jump address is COMPEA and this is the next CRAM address. The dispatch is EAMOD and, on the next EBox clock, the microinstruction from COMPEA is fetched. Note, too, that the DRAM register is latched and contains the A, B, and Executor Jump address.

### 2.3.5 A READ Dispatch – Set Up Data Fetch and Prefetch

Refer to Figure 2-29. Once the effective address has been calculated, what has been traditionally called the Fetch cycle follows. The CRAM register contains the microinstruction fetched from COMPEA. The J field is zero in this case. The EBox hardware, upon detecting a Read Dispatch, inspects the dispatch A field and forces the CRAM address to 40 + A. Thus, in this example, the address becomes 45. Address 40 + A is defined by hardware. The effective address in ARX18–35 is enabled into the ADDER A input by the AD field coded as A, with ADA selecting ARX. To begin the data fetch, the MEM field is coded as A READ and this, with the A field, generates EBOX REQUEST and EBOX READ. On the next EBox clock, the effective address is loaded into AR.

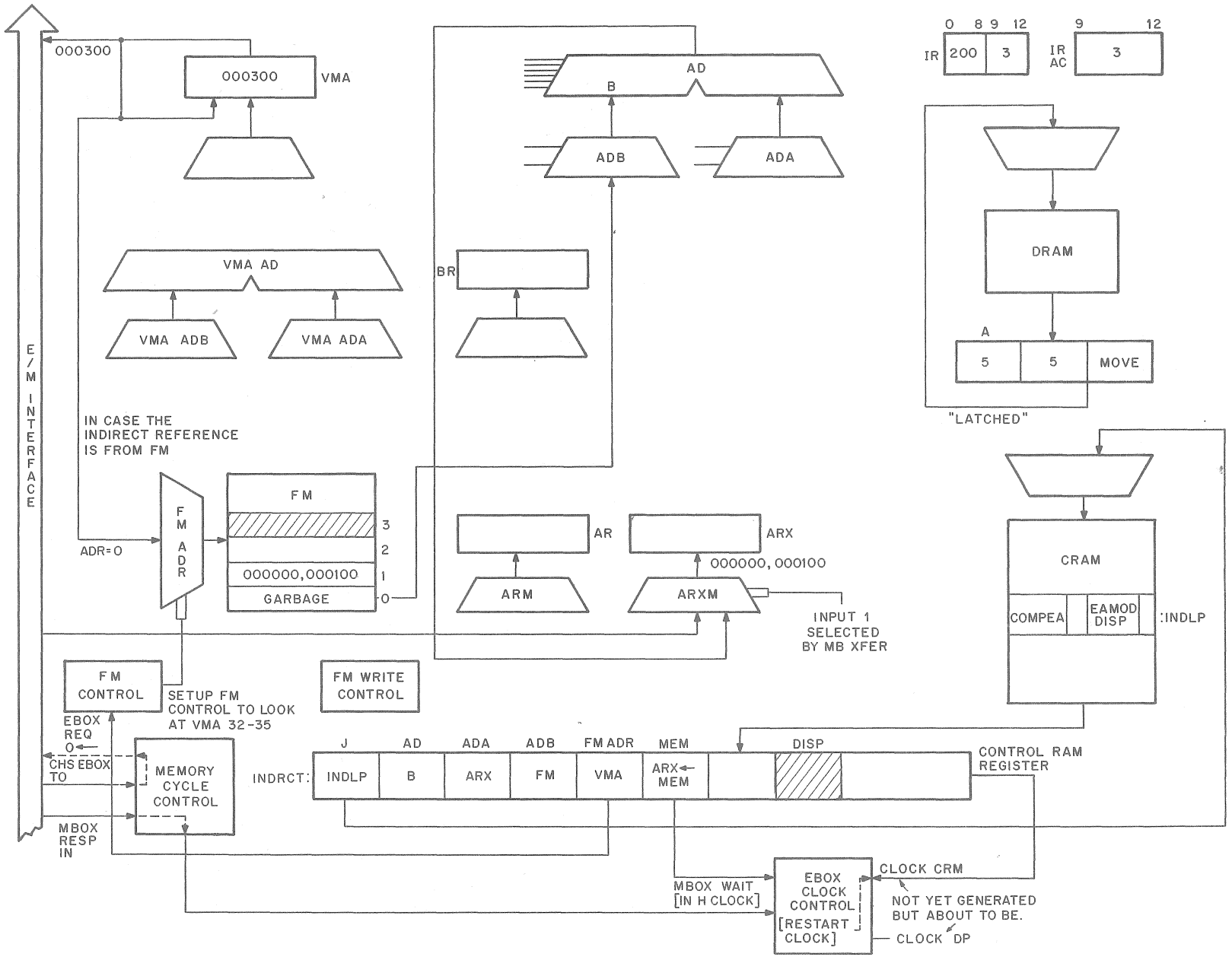



Figure 2-27 MBox Response to Indirect Request

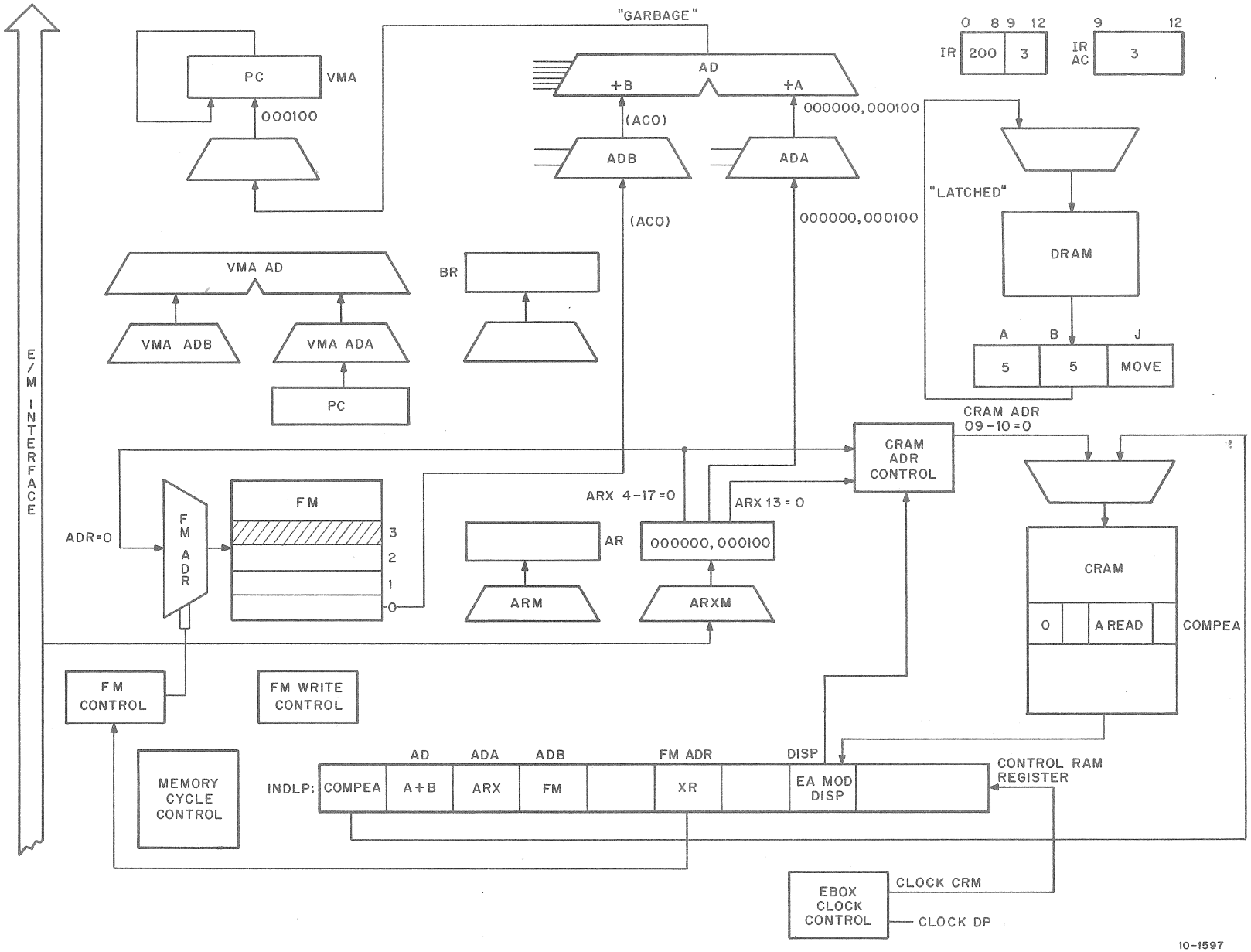



Figure 2-28 Address Calculation Continues

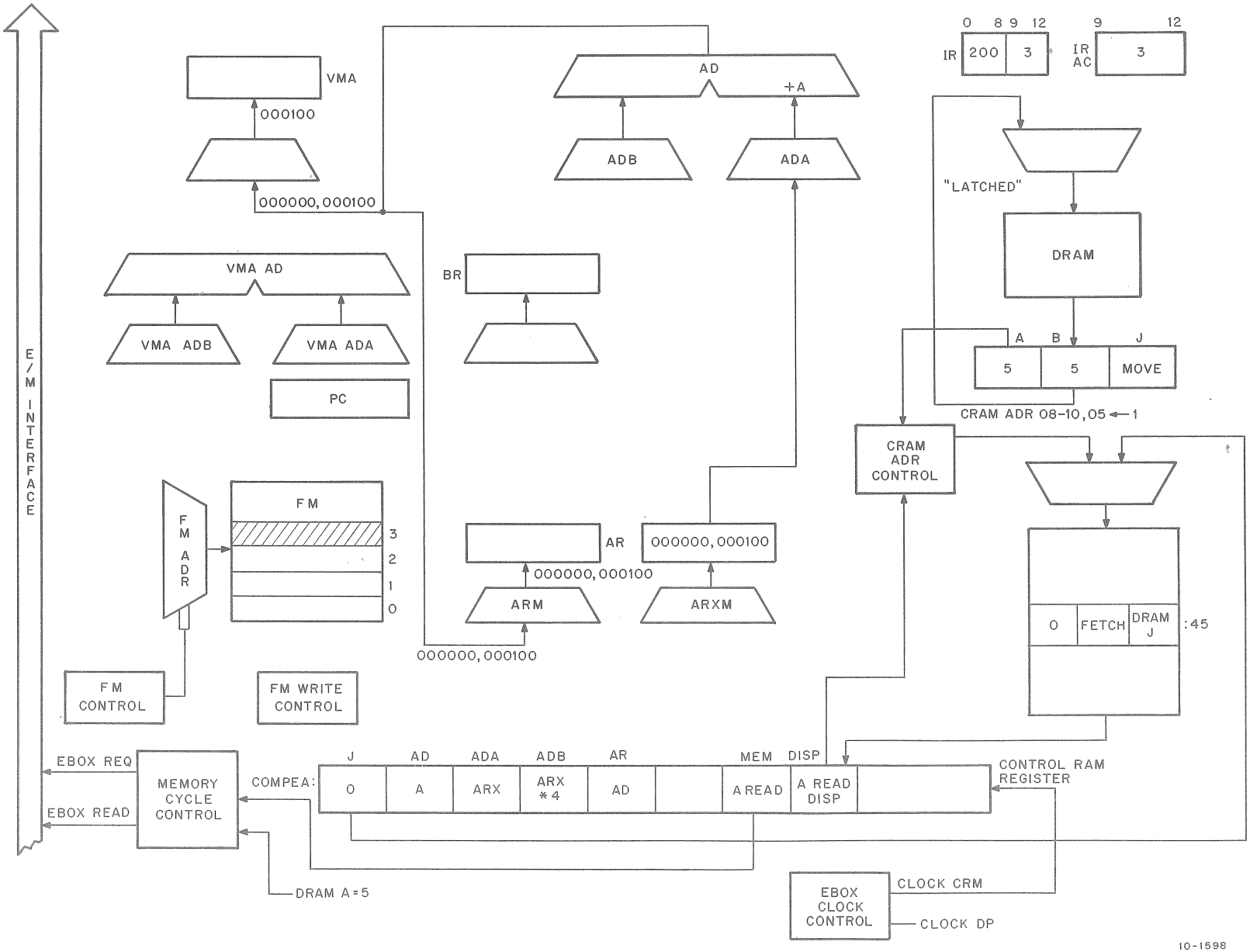



Figure 2-29 AREAD Dispatch Setup Data Fetch

### 2.3.6 MBox Response to Data Read – Prefetch Begins

Figure 2-30 illustrates the CRAM register containing the microinstruction from location 45. The jump address once again is zero, because the actual jump address is provided by the DRAM register jump field. In the case of MOVE, the symbolic address is "MOVE." This location contains the first microinstruction in the executor for the MOVE instruction. Only one microinstruction is required for the execution of the basic MOVE. This dispatch field contains DRAM J, enabling the CRAM address control to utilize the jump address in the dispatch register. Thus, for the basic MOVE, symbolic location "MOVE" contains the desired microinstruction. The MEM field is coded as fetch to enable the memory cycle control to begin the prefetch by asserting EBox request with EBOX READ.

Until the MBox response to the data read is received, the VMA is latched and only the VMA input contains the updated PC value. When the MBox response is received, the VMA is loaded with the updated PC value (PC+1). At the same EBox clock, the data on the cache data lines is clocked into AR (000100). Referring to Figures 2-30 and 2-31, the FMADR field enables FM to be addressed via VMA 32–35, even though in this example VMA address 000100 is not an FM address. FM location 0 is actually accessed and enabled via ADDER B into the AR mixer.

The Memory Cycle Control asserts LOAD AR. The address in VMA is checked in the VMA Control and, because it is not a fast memory address, –VMA AC REF is asserted. This is passed to EBox Control No. 1 logic and inhibits the generation of FM XFER.

MBox RESPONSE IN is passed to the EBox clock control where it becomes (on the next MBox clock) RESPONSE MBox. This, with LOAD AR, enables the selection of ARM SEL 1, which enables the cache data into AR. The EBox clock then strobes the AR register. This clock also clocks the next microinstruction from symbolic location MOVE into the CRAM register.

### 2.3.7 Executor – Set Up for Store Cycle

For the basic MOVE instruction, the data word in AR must be stored in the FM location specified in the AC field of the currently executing instruction. The microinstruction J field contains the base address for the data storage microprogram. This is symbolic location ST0. The Dispatch field is coded as DISP B, which enables the B field of the DRAM register to modify the low-order three CRAM address bits (CRAM 08–10). The B field is 5 for MOVE and this yields symbolic location STAC. If, for example, ST0 was physically 60, the resulting address would be generated by logically ORing 60 with 5 for a result of 65, symbolically STAC.

Referring to Figure 2-32, IRAC contains AC address 3, and is enabled to address FM because the microinstruction FM ADR field is coded as AC0. This is the AC specified by AC 09–12. The MEM field specifies B WRITE, but no request is issued. This is because the memory cycle control samples the DRAM B field and inhibits an EBox request when DRAM B01 is a zero.

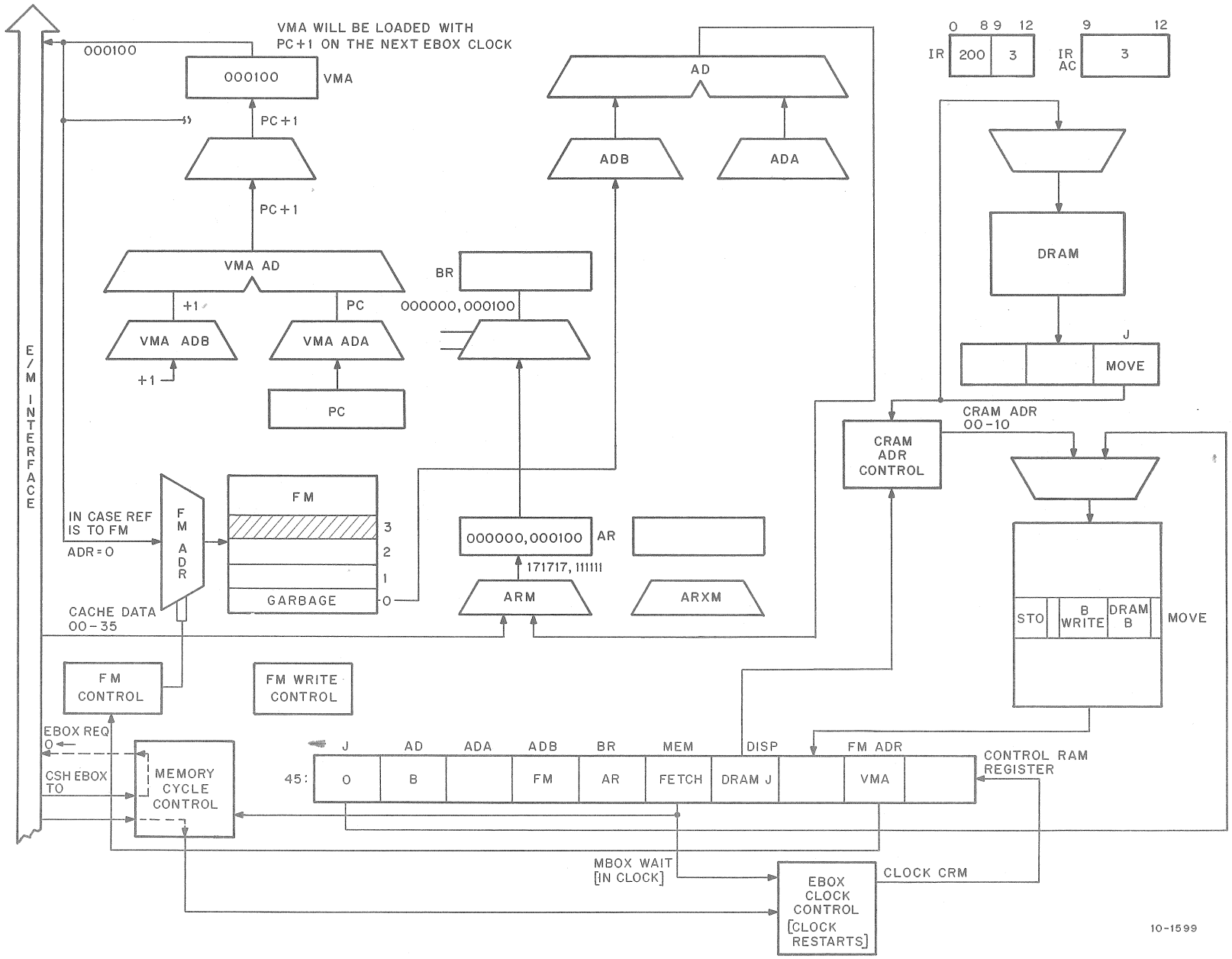



Figure 2-30 MBox Response with Data Word Requested

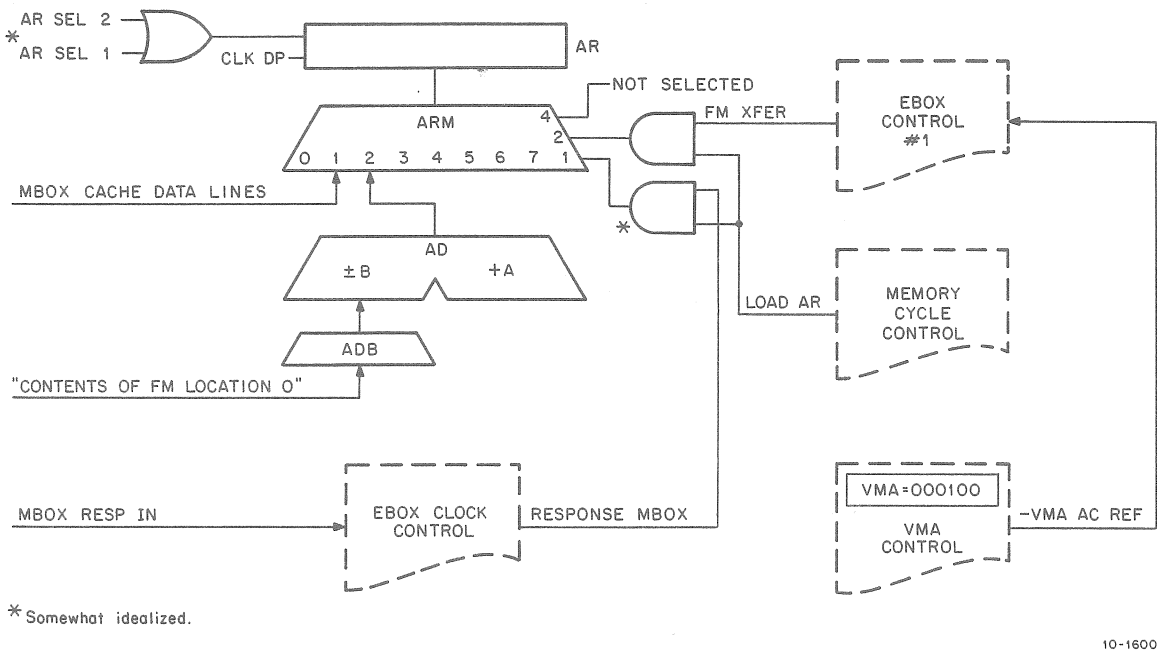



Figure 2-31 Hardware Selection of ARM Data

### 2.3.8 Finish Store Cycle – Perform NICOND Dispatch

The CRAM register now contains the microinstruction from symbolic location STACK (Figure 2-33). The J field specifies the base address NEXT and the Dispatch field contains NICOND Dispatch. This completes the basic machine cycle by reentering the instruction cycle once again.

The FM ADR field maintains the FM address via IRAC and the COND field is coded as FM WRITE to write the contents of AR into FM location 3. The MEM field is coded as MB WAIT for the cases where the next instruction has been prefetched from memory. This forces the EBox to wait until the instruction enters the ARXM and MBOX RESPONSE is received. If the instruction is being fetched from fast memory, MB WAIT has no effect and the microprogram selects the appropriate micro-instruction to load ARX from fast memory as addressed by VMA 32–35.

## 2.4 PAGE FAIL CYCLE INTRODUCTION

Normally, primary memory is the MBox cache memory, secondary memory is core memory, and the auxiliary memory is a disk or drum. Information is moved into the core only on demand (Demand Paging), i.e., no attempt is made to move a page into core memory, and consequently words into the cache, until some program references it. Information is returned to core memory in accordance with a hardware algorithm in the MBox hardware. Information is returned from core memory to auxiliary storage at the discretion of the operating system's paging algorithm. Information movement across the gap bridging the level between auxiliary storage and core memory-cache memory is called page traffic.

The MBox, in a sense, is an interface between the EBox (processor) and the SBus. It provides individual mapping (relocation) of each page (512 words) of both user and monitor address spaces, using separate maps for each. The MBox uses hardware storage to access and load the mapping information.

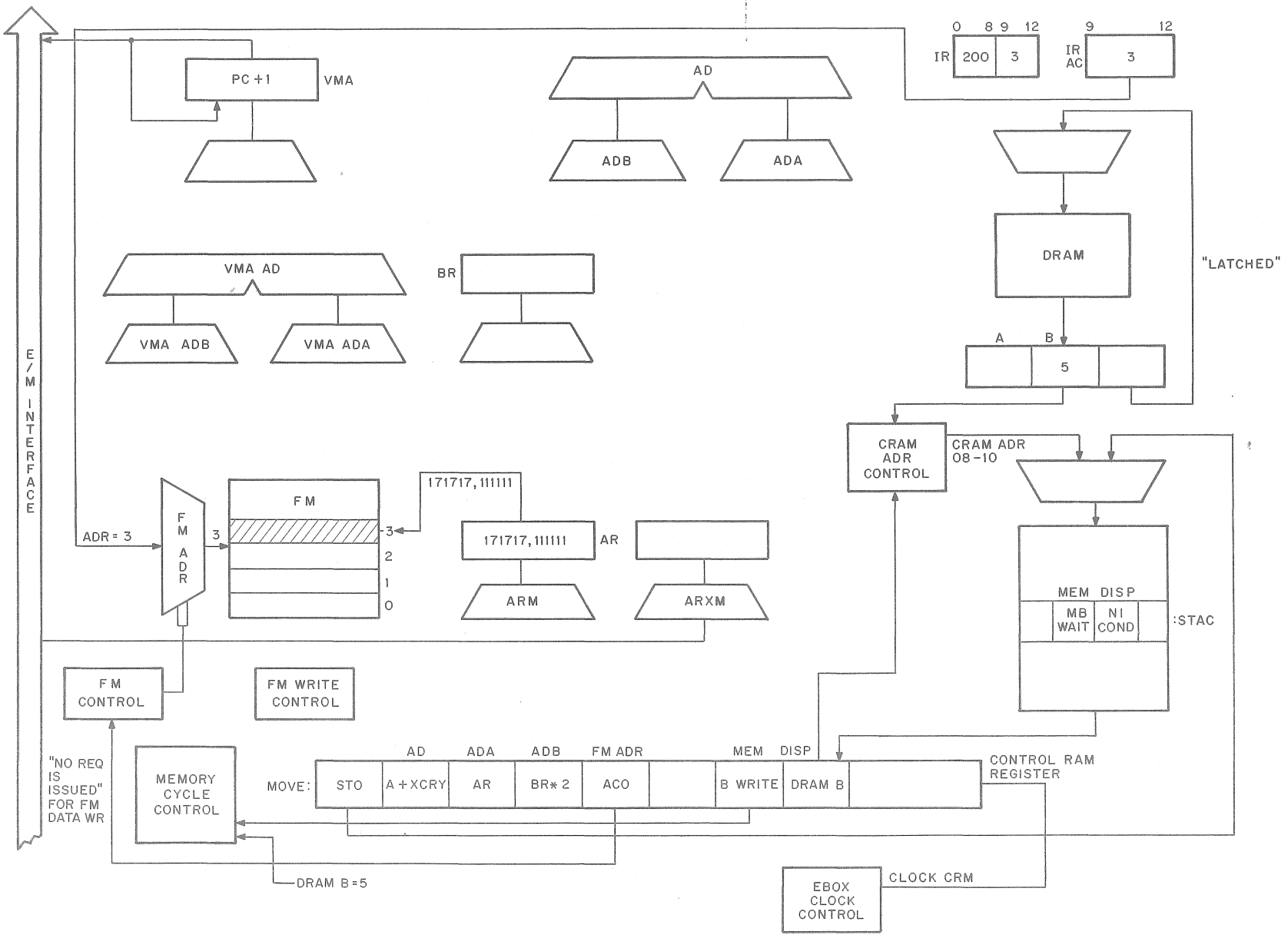



Figure 2-32 Executor Setup for Store Cycle

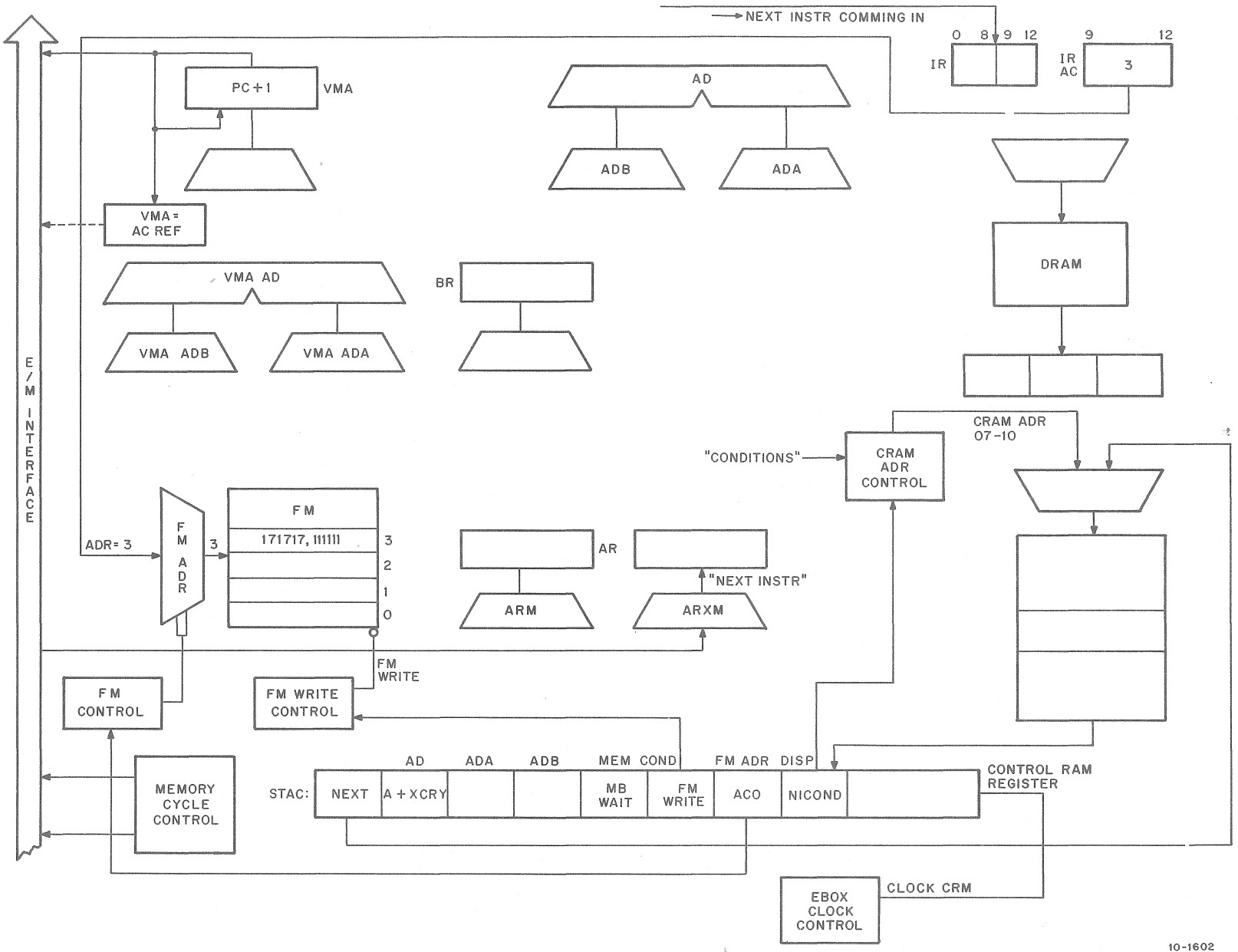



Figure 2-33 Finish Store Cycle, Perform NICOND Dispatch

It also contains a 2048 word cache for holding the data for the mapped references. On each memory request from the EBox, the nine high-order bits of the virtual address and the type of request (read, write) are compared with the contents of the hardware tables in the MBox. If a match is found, the location containing the match also contains 13 high-order address bits to reference the physical page in the cache. If no match is found, a 512-word "Page Table" in physical core memory is referenced. The word selected in this page table is determined by a dispatch based on the original nine high-order address bits. The 13 high-order address bits and use bits found in this word are written into the MBox hardware table; the use bits are checked against the type of EBox reference. Four possible cases exist concerning the disposition of the use bits:

1. The page is not in core.
2. The page is protected from the type of request.
3. The page is nonexistent.
4. The page is in core and is compatible with the type of request.

For the first three cases, a page fault (trap) occurs; for the fourth case, the requested word is fetched from core memory (actually words are fetched four at a time, differing only in the two least significant address bits) and written into the cache. Concern here is with the page fault situations. The MBox constructs a page fault word in one of its internal hardware registers, the EBus register. The word contains information relating to the type of fault that occurred. The EBox is waiting for an MBox response to its request; the MBox, therefore, asserts PF HOLD, and some time later asserts MBOX RESPONSE IN. When the EBox recognizes the PF HOLD signal, it forces the CRAM address to 1777. This is the first microinstruction in the micropage fault handler. The EBox does not issue an EBox clock until the CRAM address has had time to set up. Once the address is stable, a single EBox clock is issued to the CRAM board to access the microinstruction.

### 2.4.1 Page Fail Handling – Functional Flow

Figure 2-34 is a functional flow of the microprogram page fault handler. The EBox contains a 4-bit state register. This register, during certain instructions, holds a number that may be used to modify the state of the CRAM address. For instructions that do not use the State register, it contains zero. Generally, the STRING, EDIT, and BLT instructions require cleanup following a page fault so that they may be properly terminated. For these cases, the State register contains a value in the range of 1–7. The more general case is discussed here; this is where the State register contains zero. For both cases, INSTR ABORT (coded in the condition field of the microinstruction fetched from CRAM address 1777) performs the following functions:

TRAP REQ 1 ← TRAP CYCLE 1
TRAP REQ 2 ₂ TRAP CYCLE 2
ADR BRK INH ← ADR BRK CYCLE

These actions are necessary to assure that the PC flags reflect the state of the EBox when a page fault occurs during the fetch of the trap instruction, during its execution, or during an address break page fault. A State register dispatch is given, but because the State register is clear, the base address is used to obtain the next microinstruction. A priority interrupt has a higher priority than a page fault (Figure 2-35); therefore, a pending interrupt is checked for first. If INT REQUEST is true, the PI Handler is entered to service the interrupt. If no interrupts are pending, the page fault is handled. The third level of priority is given to traps and finally to all other events being processed by the microprogram.

A page fault occuring in response to an API interrupt function is a fatal error. Thus, when the page fault handler finds PI CYCLE set, it sets the I/O Page Failure flag, dismisses the failing interrupt, and then, if possible, restores the EBox to the state it was in prior to the interrupt. The setting of IOPF eventually causes an interrupt on the APPR error channel. The PF Handler now attempts an instruction fetch.

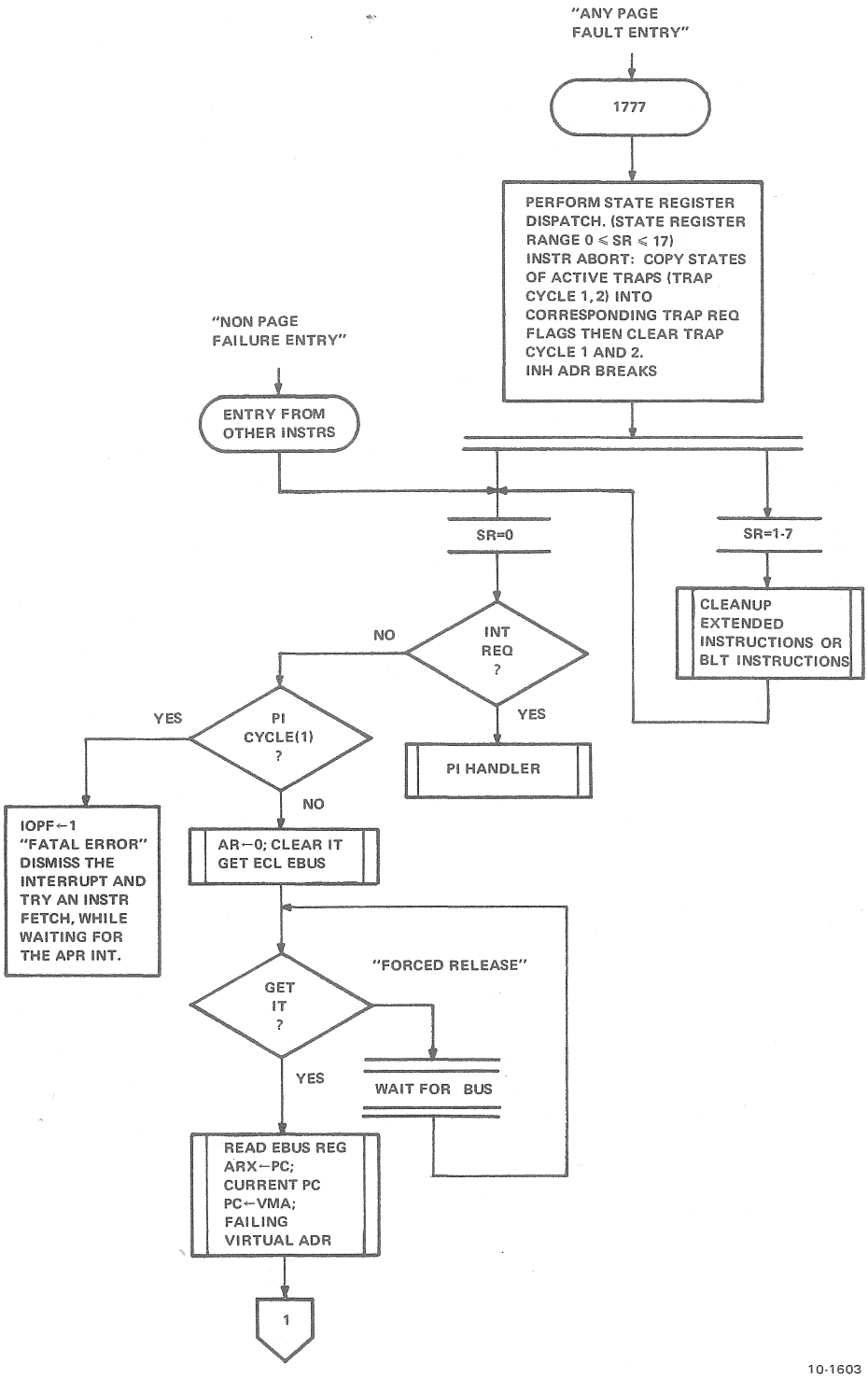



Figure 2-34 Page Fail Handling (Sheet 1 of 2)

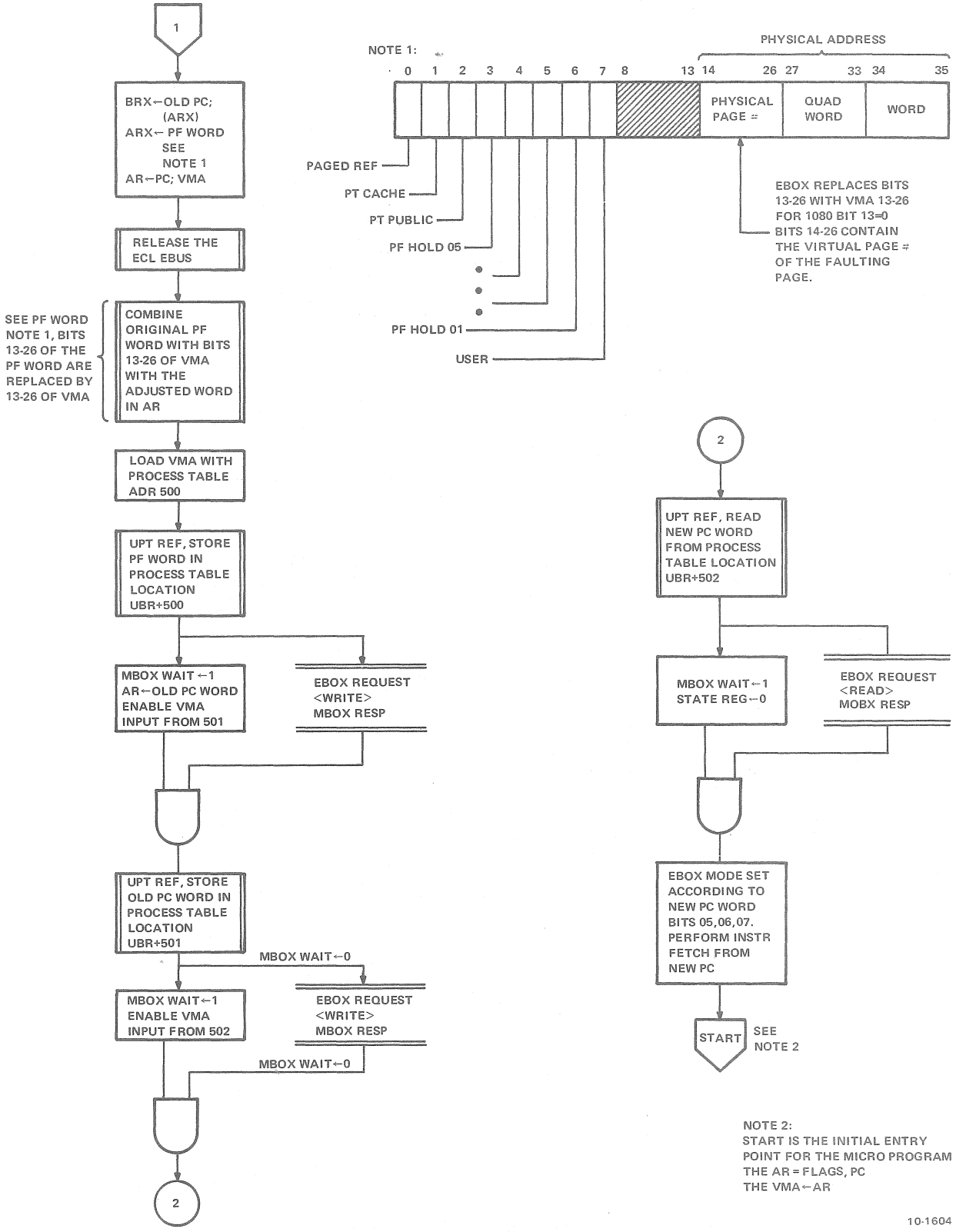



Figure 2-34 Page Fail Handling (Sheet 2 of 2)

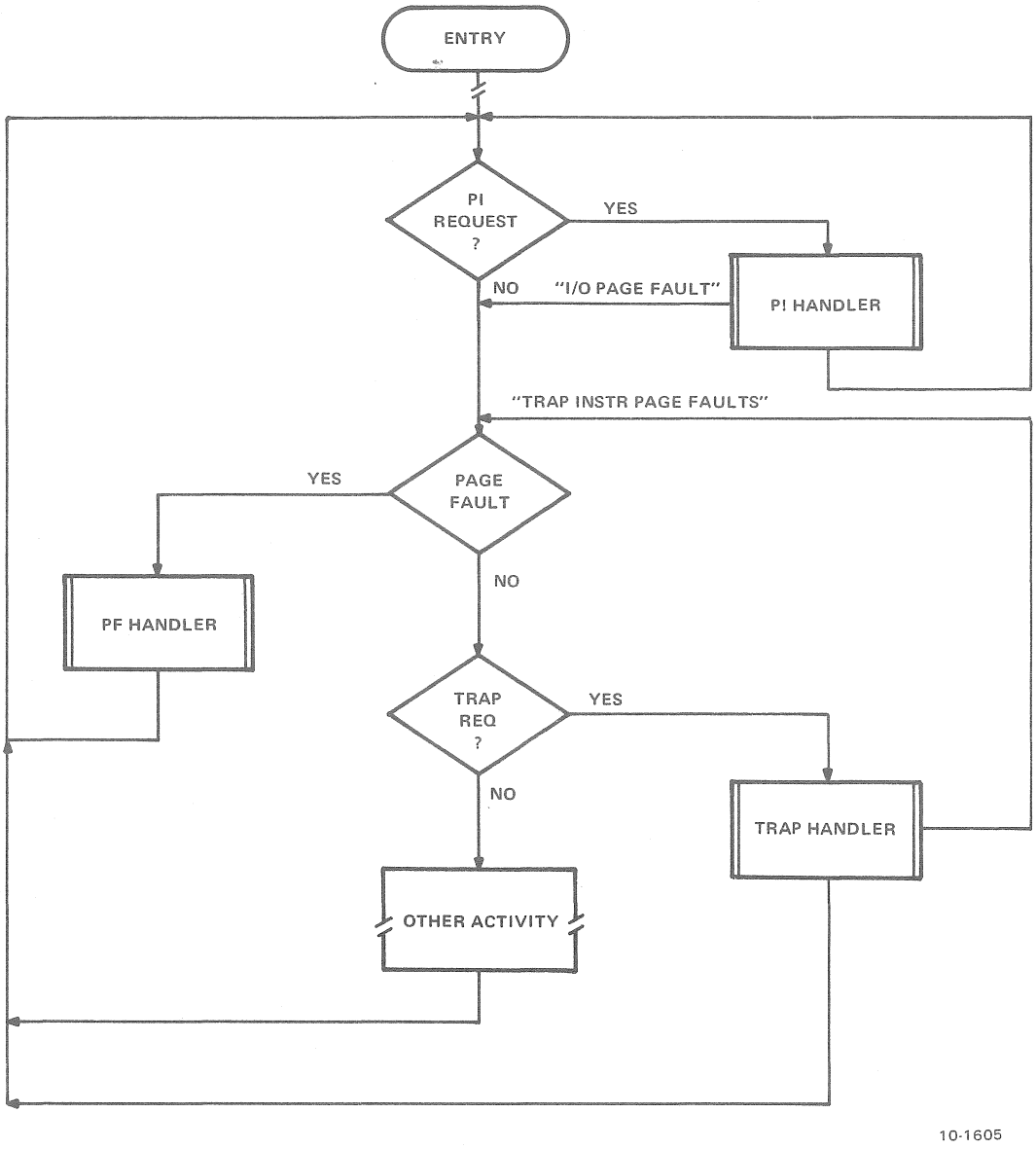



Figure 2-35 EBox Priorities

**Obtaining and Adjusting the PF Word** – Assuming PI CYCLE is clear, the AR is cleared and the ECL EBus is requested. This is to transfer the PF word from the MBox EBus register to the AR register in the EBox via the EBus. Because the PI system and external or internal devices can also use the EBus, the microprogram must force its release. When the ECL side is obtained, the EBox reads the PF word into AR. The PF word, as it is constructed by the MBox, contains the physical page number in bits 14–26. The EBox must replace this with the virtual address and also clear bit 13. The current virtual PC is temporarily placed into ARX; the failing VMA is placed into AR while the old PC is saved in BRX. The ECL EBus is then released. The ARX and AR are shifted to adjust bits 13–26 to be the VMA 13–26.

Figure 2-36 shows the three locations in the user process table dedicated to page fault handling.

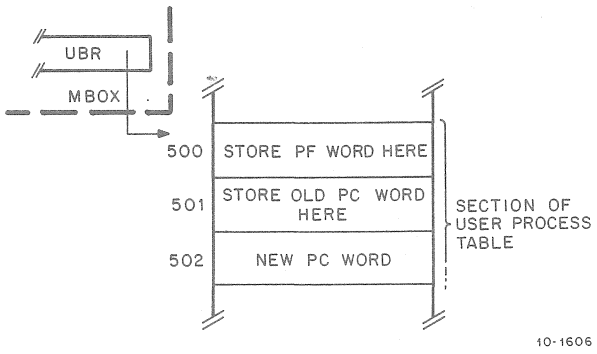



Figure 2-36 Process Table PF Location

### 2.4.2 Process Table References

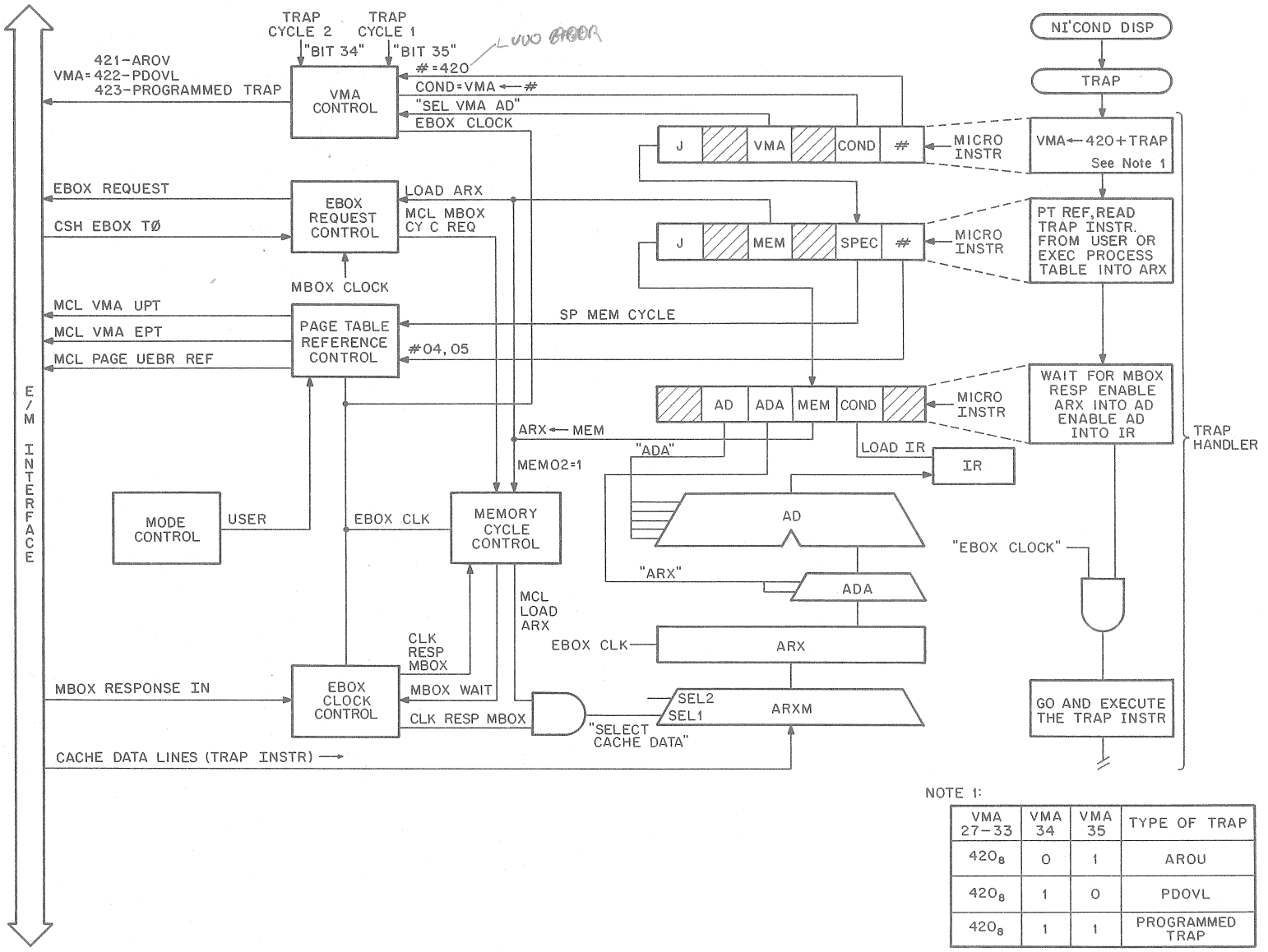
The VMA is loaded with low-order process table location 500 and an EBox request is issued to write the PF word (concurrently in AR) into process table location UBR+500. The next microinstruction is loaded and EBox clock sets MEM CYCLE, causing MBOX WAIT. The AR is enabled from the old PC word; the input to VMA is now 501. As soon as the MBox responds, MBOX WAIT is removed and the cycle is repeated. This time the EBox request is to write the old PC word (now in AR) into process table location UBR+501. Once again, the next microinstruction is loaded and EBox clock sets MEM CYCLE, causing MBOX WAIT. The VMA input is now 502. As soon as the MBox responds, MBOX WAIT is removed and the cycle repeats, in this instance for reading a new PC word from process table location UBR+502. The new PC word places the EBox in a specified mode and the first instruction is fetched from the appropriate handler. This completes the page fault cycle.

## 2.5 TRAP CYCLE – INTRODUCTION

A Trap is produced by setting either of two trap request flags in the EBox (TRAP REQ1 or TRAP REQ2). The programmer knows these flags as TRAP2 and TRAP1. The conditions that set TRAP REQ1 are equivalent to the arithmetic overflow conditions that set SCD OV. TRAP REQ2 is set by the various pushdown overflow conditions: the left half of the pointer is counted down to -1 (no carry out of bit 0) in a POPX, or is counted up to zero in a PUSHX. (The condition for this is the presence of a carry out of bit 0, but the condition is detected by the microprogram and the trap request flag is set.)

### 2.5.1 Trap Handling

The Trap Handler (Figure 2-37) is entered at NICOND Dispatch time providing its priority is highest of the major priority events. The microprocessor NICOND Dispatch, together with four queues arranged in a round robin priority structure, is shown in Figure 2-38. The TRAP request is served only when no priority interrupt requests are pending and no page fault is pending. It does, however, pre-empt the normal instruction cycle. Both the user and exec process tables contain dedicated locations for processing traps. These locations are XXX 421 for arithmetic overflow (TRAP1), XXX 422 for pushdown overflow (TRAP2), and XXX 423 for the programmed trap (TRAP3). XXX is replaced by the appropriate base register (UBR or EBR), which resides in the MBox. The base register used by the MBox is determined by the state of the qualifiers sent during the EBox request. The MBox fetches the appropriate trap instruction and places it on the cache data lines while issuing MBOX RESPONSE IN. The EBox then executes the trap instruction. It is possible for the EBox request for the trap instruction to cause a page fault. If this occurs, the page fault handler is entered at CRAM address 1777 and the trap cycle flags are pushed into the trap request flags so that the trap flags may be saved; the trap cycle properly reenters at a later time.




NOTE 1:

| VMA 27–33 | VMA 34 | VMA 35 | TYPE OF TRAP |
|---|---|---|---|
| $420_8$ | 0 | 1 | AROU |
| $420_8$ | 1 | 0 | PDOVL |
| $420_8$ | 1 | 1 | PROGRAMMED TRAP |

10-1607

Figure 2-37 Trap Cycle

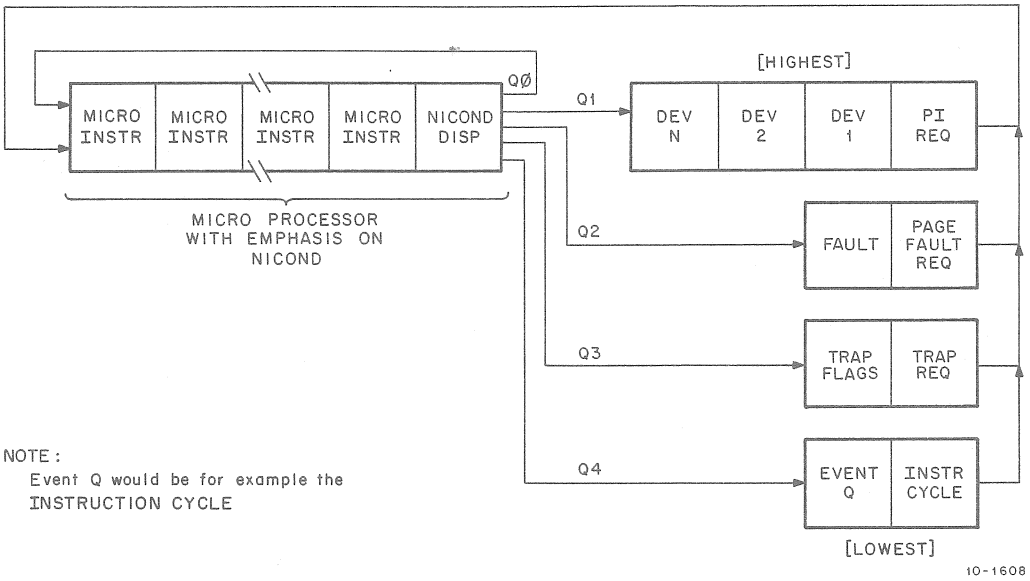



Figure 2-38 Central-Server Model (Round Robin Priorities)

### 2.5.2 Address Generation

Referring to Figure 2-37, the VMA is enabled to be input from the VMA ADDER. The condition field of the current microinstruction enables the number field to generate the process table low-order address 420; the low-order two bits of VMA AD 34 and 35 assume the state of the trap flags.

### 2.5.3 PT Reference for Trap Instruction

The next microinstruction must generate the EBox request and enable the appropriate qualifiers to appear on the E/M Interface lines. The page table reference control samples the state of the USER, together with the special function and number bits and then asserts either MCL VMA UPT and MCL PAGE UEBR REF for a USER trap situation or asserts MCL EPT and MCL PAGE UEBR REF for an EXEC trap situation. The MEM field is coded to load ARX and enable the EBox request.

Assuming no page fault occurs, the MBox fetches the instruction, places it on the cache data lines, and asserts MBOX RESPONSE IN. The MEM cycle control samples the MEM field function LOAD ARX to enable one leg of the ARXM and CLK RESP MBOX enables the other leg. Thus, the instruction enters ARX on the next EBox clock. Next, op code and AC field of the instruction in ARX must be enabled into the ADDER and then latched into IR. The condition field of the current microinstruction COND/LOAD IR unlatches the IR for one EBox cycle, allowing the AD to load into IR. On the next EBox clock, it latches again. The final step is to perform the trap instruction. This completes the trap cycle.

## 2.6 INTERRUPT CYCLE - INTRODUCTION

The system must possess a true priority interrupt system that is flexibly structured and controlled. Its operation in establishing priorities and recording and sequencing interrupt requests is essentially instantaneous and independent of EBox action. Interrupts of high priority must be permitted to interrupt partially completed responses to those of lower priority. To maintain fast response, interrupt requests should require no decoding action on the part of the EBox to determine their source or nature. Capability for dynamically varying the priority structure to meet the demands of a changing environment must be available. In addition, no other system element may be designed such that its proper operation requires inhibition of the priority interrupt system for any period of time.

The basic priority interrupt level has four mutually exclusive states that can be described as Disarmed (-PI ON), Armed (PI ON), Waiting (PI REQ), and Active (PI HOLD). Figure 2-39 shows the basic concept of the interrupt system for two channels. It is arranged in four groups, the interrupt state, the FF configuration for two of the seven possible channels, the level enable, and the source of change signal. In the Disarmed state, the interrupt level rejects all incoming interrupt trigger signals. By performing a CONO PI and specifying the appropriate bits, the priority interrupt system can be armed or disarmed for any or all channels.

In Figure 2-39, the processor (CPU) performs a CONO PI and arms both channels. In the armed state, the interrupt level accepts a trigger signal from an outside source or from an internal source, e.g., the APR, and moves to the waiting state (REQUEST STATE), where it remains until it is acknowledged by the EBox. All waiting and enabled requests are input to a priority network where they are compared with the current state of the priority interrupt system. In this example, both channel 1 and channel 2 are requesting service, and both channels have previously been armed by a CONO PI instruction. In addition, an interrupt is shown holding on channel 2. Thus, until it is dismissed by the processor, the channel 2 request pending is held in abeyance. Furthermore, the channel 1 request causes the device subroutine for channel 2 to be interrupted, diverting the processor to the device subroutine for channel 1. The first instruction that will be executed as a result of an interrupt (subroutine type service) is a JSR instruction. This instruction saves the processor flags, program counter value, and also holds the interrupt.

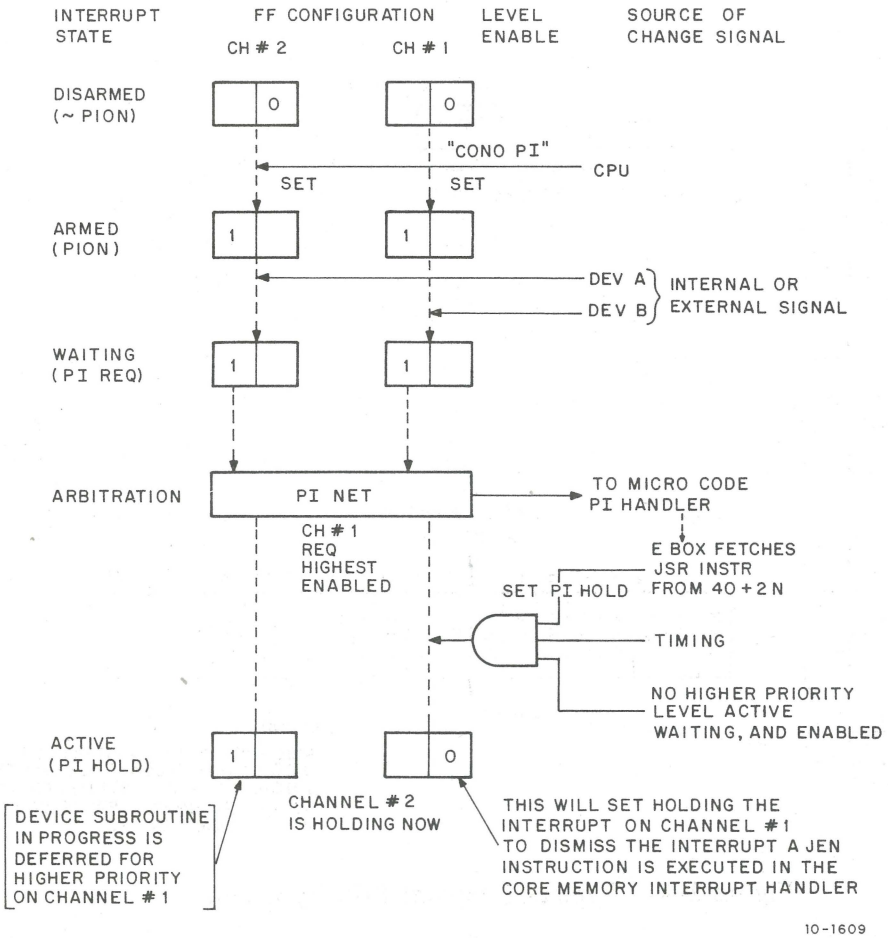



Figure 2-39 Interrupt Level Operations

When service has been completed, the service routine dismisses the interrupt, restores the flags and program counter, and the channel 2 subroutine continues. Interrupt channels are organized into seven basic levels, which are software assignable (armed): the lowest number has the highest priority within the numbered sequence (Figure 2-40). Each channel is subdivided into finer levels or priority by hard-wired physical device numbers. As indicated, the first eight physical numbers (0–7) are assigned to 1–8 Massbus controllers in the system. The next four physical numbers (8–11) are assigned to 1–4 DTE20s (10/11 Interfaces); and numbers 12–14 are reserved for expansion. Finally, physical number $15_{10}$ is assigned to the I/O bus adapter (one exists per system, if needed).

Each interrupt channel has a dedicated pair of unique locations within the EPT. These locations may be indicated as 40 + 2n, and 41 + 2n, where n represents the channel number. When a device initiates an interrupt in the KL10 system and is selected for service, the device places onto the EBus a special function word hereafter labeled API function. This function contains information that specifies the type of service required. Figure 1-32 indicates the format of this word. Note that the format varies from device to device, but the functions that can be specified in bits 3–5 are common to all system devices. Function codes of 0, 1, and 7 cause instruction fetches from 40 + 2n initially and, depending upon the type of instruction in 40 + 2n, may at some point perform an instruction fetch from 41 + 2n. In general, 40 + 2n contains one of the following types of instructions:

JSR
JSP*
PUSHJ*
MUUO

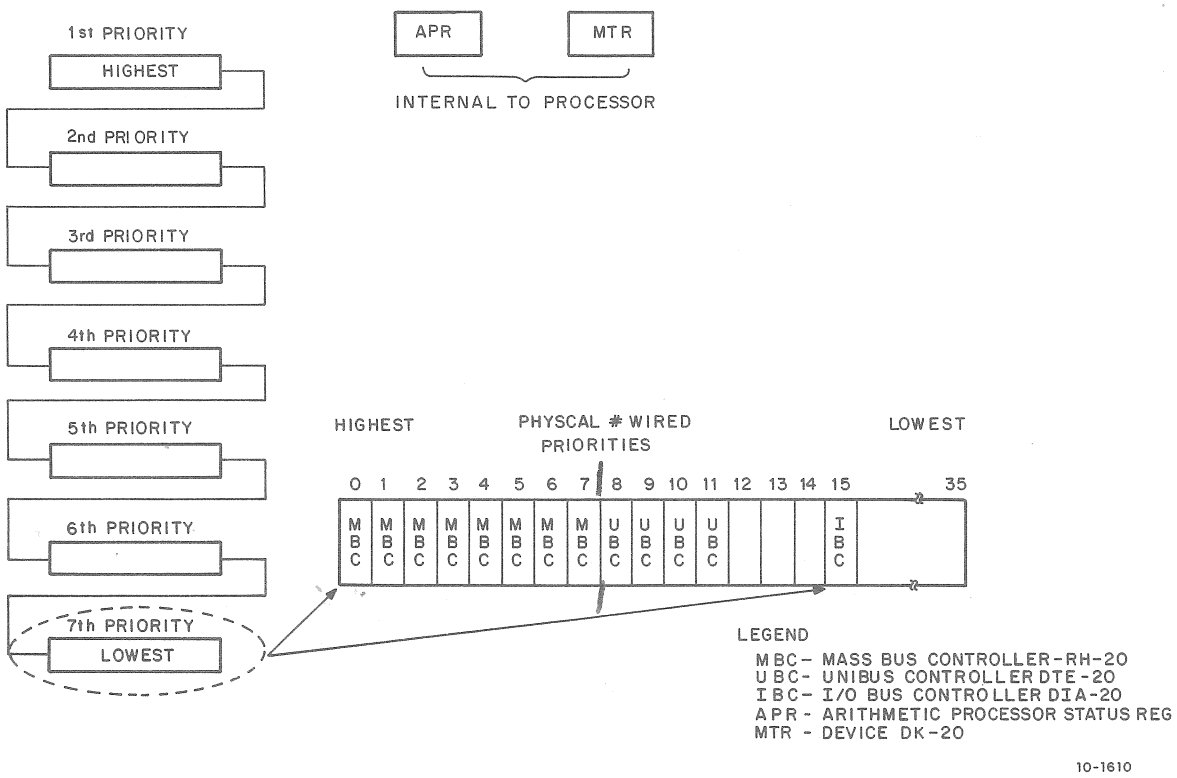



Figure 2-40 Typical Interrupt Priority Chain

*These instructions should not be used because nothing is known about the ACs when the interrupt occurs. JSR or MUUO are better choices.

All of these instructions save the flags and PC, a requirement when entering the device service routine. If the instruction at 40 + 2n is a BLKX instruction, a specified number of transfers are performed, one transfer at a time, each time returning to the interrupted program or to a higher level subroutine. On the last transfer, the return to the interrupted program is "NOT SKIPPED" and an instruction is fetched from 41 + 2n. In a similar fashion, if 40 + 2n contains a SKIP class instruction; when the skip condition is satisfied, a return to the interrupted program takes place. If the skip is not satisfied, the instruction in 41 + 2n is executed instead of the return. The API function generated by the Massbus controller is always a function code of 2 in bits 3–5; this implies a dispatch to the physical address provided in the API function word. The dispatch is into the device handler for the Massbus devices. The type of API function requested varies with the device or controller responding.

It is possible for the processor to generate a program request for an interrupt on any of the seven channels. This permits the processor to carry out the highly time-sensitive portion of the interrupt response, and to then create for itself a low priority interrupt to call for the deferred servicing of the less time-sensitive portion at a less pressing time.

### 2.6.1 Duration of Uninterruptable Intervals

Such an interrupt system is of little value if the CPU can remain in an uninterruptable state for any significant period of time. Under normal operating conditions, the longest uninterruptable interval must be kept short. In addition, no malfunctioning peripheral hardware or software can be allowed to "hang up" the CPU in a noninterruptable state.

### 2.6.2 Interruptable Instructions

To ensure that the longest uninterruptable interval that the EBox may experience in normal operation is short, some long instructions have been designed so that they may be interrupted during execution. First, all instructions are interruptable at indirect references during the effective address calculation. Second, instructions that consist of two parts may be interrupted between the two parts, a PC flag being set to record this for later, when only the second part will be done. Third, iterative instructions, such as BLT, may be interrupted at any point, as an AC pointer defining work still to be done is beng updated continually.

### 2.6.3 General Interrupt Sequencing

The mechanism for handling the various levels of interrupt priority in the hardware, and the relation between this mechanism and the device subroutine call and return sequence as it might occur in practice are shown in Figure 2-41. Three channels are armed by setting their PION flags. Channel 2 has highest priority, followed by channel 3, and finally by channel 4. Note that the execution of a CONO PI instruction caused the PION flags to set. Three separate interrupts occur simultaneously on channels 2, 3, and 4. The priority network is shown arbitrating the three priorities. The lowest channel (highest priority) is serviced, provided it is of higher priority than the current level.

In this example, all three channels are requesting and no channels are currently holding interrupts; thus, the channel with the lowest number is selected. As a result of the arbitration, the selected channel number is combined with the appropriate constant to form the address 44[40+2X (2)]. In Figure 2-41, the device subroutine is entered by fetching and executing the instruction in EPT location 44, which in this instance is a JSR. The request is not cleared until the program issues CONO, DEV. Notice during the entire service routine (in this example), the requests on channels 3 and 4 are waiting for the processor. The last instruction to be executed in the device subroutine is a JEN (JRST 12); this restores the flags saved by the JSR instruction executed in 40 + 2n and dismisses the interrupt on channel 2, which is holding off channels 3 and 4.

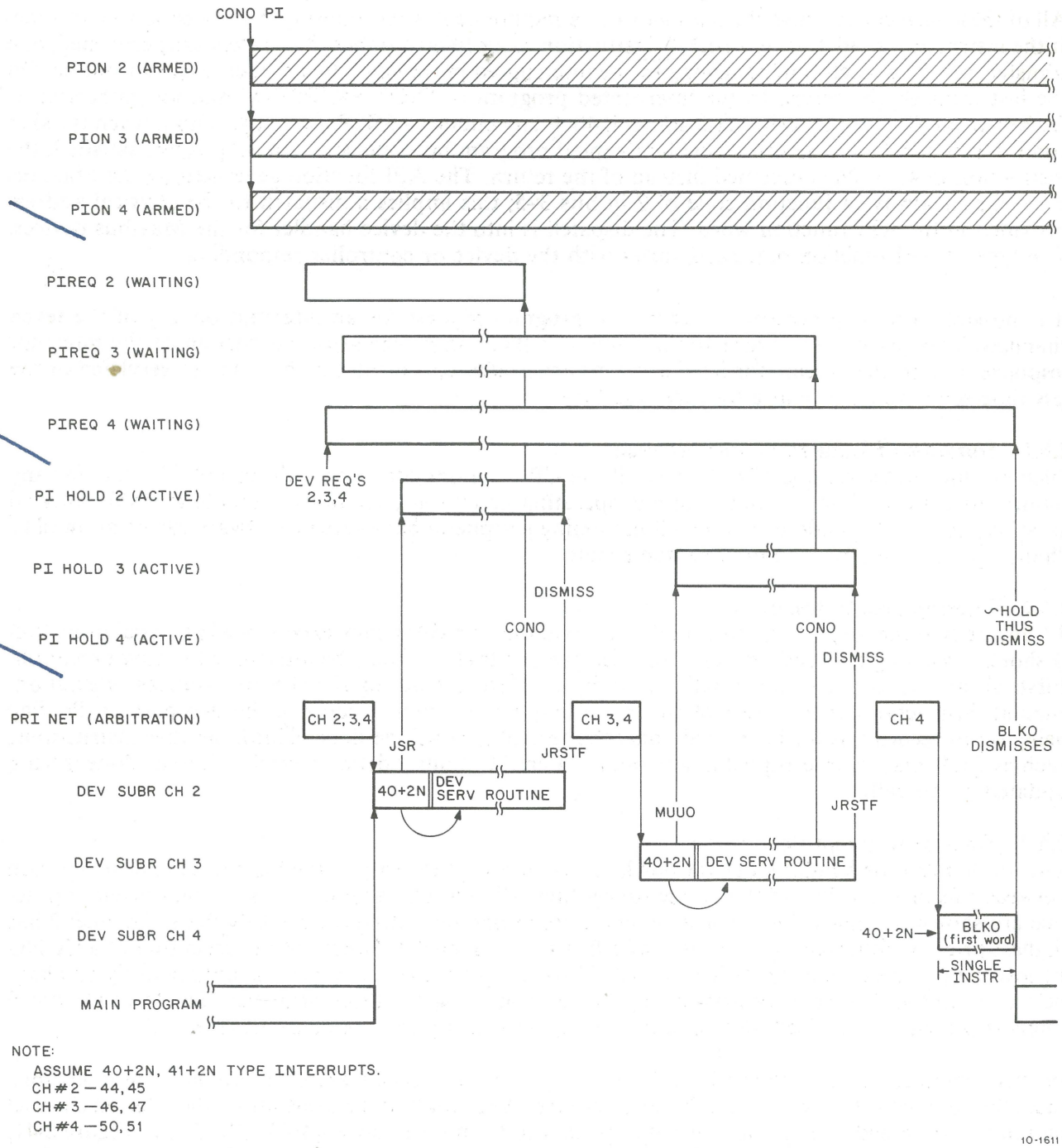



Figure 2-41 Basic Interrupt Sequencing

### 2.6.4 Interrupt Dialogue

The handling of the EBus dialogue and processor bus requests during I/O instruction execution and priority interrupts is provided by the Priority Interrupt Board, which comprises the necessary interfacing logic, control logic, and registers. Initially (Figure 2-42), assume that the appropriate PION flags have been set on the PI Board and it is now capable of accepting interrupts. For this example, the DTE20 will generate an interrupt for a byte of data. The drawing is divided into three sections: EBox, control activity, and DTE20. The control activity consists of control action taken by either the EBox or the DTE20, as appropriate.

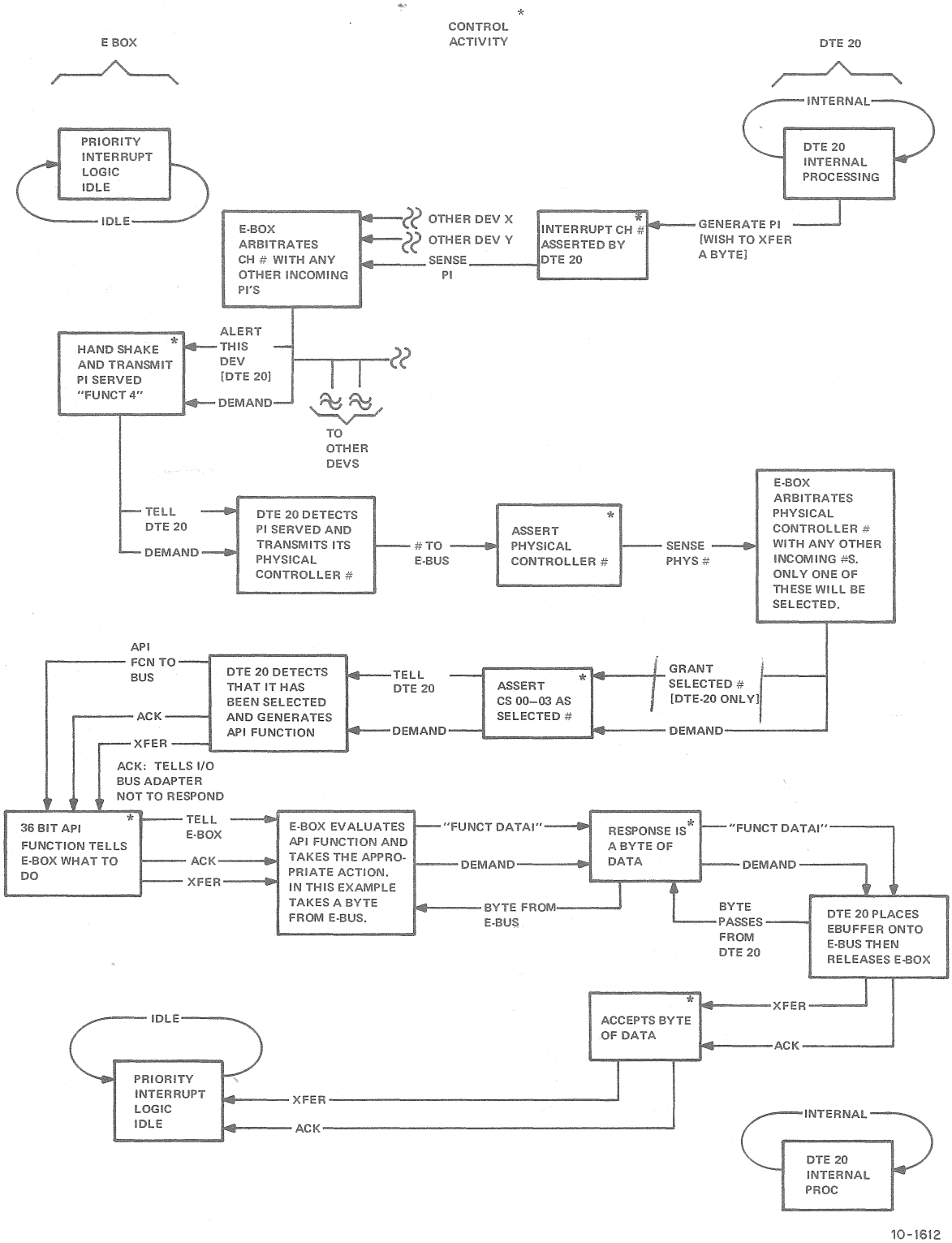



Figure 2-42 Interrupt Dialogue Overview

The DTE20 asserts one of its interrupt lines PI 1–7; this level enters the PI Board where, as indicated, it is arbitrated with any other incoming requests and any holding interrupts. The PI Board now commences a dialogue between all candidates on the selected interrupt channel. The selected channel number is encoded in controller select (CS) lines 04–06. The function "PI SERVED" is encoded in function (F) lines 00–02. These signals are placed on the EBus and 200 ns later the PI Board asserts the signal DEMAND. This signal instructs the device (DTE20) to place its physical controller number on a prespecified bit position of the EBus (bit positions 8–11). Each controller, therefore (including the I/O bus adapter, bit position 15, disks or drums, bit positions 0–7), on the selected channel does the same. Approximately 400 ns later, the EBox drops DEMAND; however, the controller select and function lines do not change for an additional 150 ns after DEMAND is removed. The physical controller numbers received by the EBox over the EBus are arbitrated in much the same way as the channel priorities. An exception is the ARP, which is an internal KL10 device, and does not fall into quite the same type of scheme, i.e., it does not place a physical number on the EBus; obviously this is not necessary because it is already within the EBox. Rather, it provides a physical number directory on the board. This device vies with the device that is selected on the basis of physical number highest priority (Figure 2-40). Basically, the lower the numeric value of the EBus bit position onto which the device is hardwired to place its physical number, the higher the priority of that bit. The highest physical number priority, therefore, is given to bit position 0, and the next to bit 1, and so on. The highest priority physical number (in this example only) is assumed to be that of the DTE20 (one of four such possible Unibus controllers on the EBus).

The PI Board now asserts the encoded physical number of the selected controller (DTE20) in controller select (CS) lines 00–03, the interrupting channel number encoded in CS lines 04–06, and the function "PI ADDRESS IN" is encoded in function lines (F) 00–02. Again, the EBox waits a period of 200 ns and then asserts DEMAND. At this point only, one controller has been selected; it compares its physical number (hardwired on its backplane) to the number received on EBus bits 00–03. Upon determining that it is the selected controller, the DTE20 places the required API interrupt function onto the EBus data lines and asserts ACKNOWLEDGE and TRANSFER to the EBox. The ACKNOWLEDGE signal causes the I/O bus adapter to ignore the function code "PI ADDRESS IN." In the absence of ACKNOWLEDGE, PI ADDRESS IN would enable the I/O bus adapter to send its API function to the EBox, because no decoding and comparison logic exists in the adapter. This logic does exist in the DTE20 and other devices. The TRANSFER signal specifies to the EBox that the appropriate device has responded, and alerts the EBox that an interrupt is set up and pending. If the API function is sent during a DTE20 to 10 byte transfer, this could specify that the EBox perform a DATAI function to the DTE20; in this way, a byte of data is picked up as indicated in Figure 2-40.

The case of DTE20 byte transfer is somewhat unique in that the DTE20 holds onto the EBus until the EBox transmits the appropriate function, in this case DATAI encoded in function select lines 00–02 (at this time CS00–06 = 0). The byte is picked up by the EBox, and the DTE20 generates ACKNOWLEDGE and TRANSFER once again. This completes the operation. Note that ACKNOWLEDGE informs the I/O bus adapter not to respond to the functions being carried out. Because the requests on channels 3 and 4 have been pending during the service routine, when the interrupt that has been holding on channel 2 is dismissed, the priority net arbitrates between channels 3 and 4 and selects 3 for service. This generates the address 46 (40 + 2n), and this time the instruction is an MUUO. As with the JSR during the execution of the MUUO, the request is transferred to the channel 3 hold flag. Note that in the example, the request on channel 4 is still waiting for service. Finally, the JEN instruction at the end of the channel 3 service routine restores the flags and priority interrupt system, dismissing the interrupt on channel 3. In the same fashion as with the other interrupts, the priority net generates the

address 50 (40 + 2n). In this case, however, location 50 contains a BLKO instruction, which cannot save the flags or PC of the interrupted process. This type of instruction behaves in a special manner when used in an interrupt location; the BLKO instruction performs a series of transfers to a specific device; however, after each transfer, return is passed to the current PC value, whatever it is. This continues until the last transfer is completed, when the instruction in EPT location 51 (41 + 2n) is executed. This instruction should be of the type that saves the flags and PC, and will generally enter a subroutine probably to set up a new block pointer, because the current one has been expended. Note that in the beginning some main program, perhaps the monitor, was interrupted, and now control is passed back to it.

## 2.7 BASIC MACHINE MODES INTRODUCTION

In general, the KL10 permits the operation of a number of different programs, all resident in the machine simultaneously, without interference or undesired interaction among them whether due to an inadvertent program bug or maliciousness. The operation of the machine is divided into two modes, User mode and Exec mode, each with two submodes. User mode consists of Public mode and Concealed mode. Exec mode consists of Supervisor mode and Kernel mode. The machine mode structure and hierarchy are illustrated in Figure 2-43.

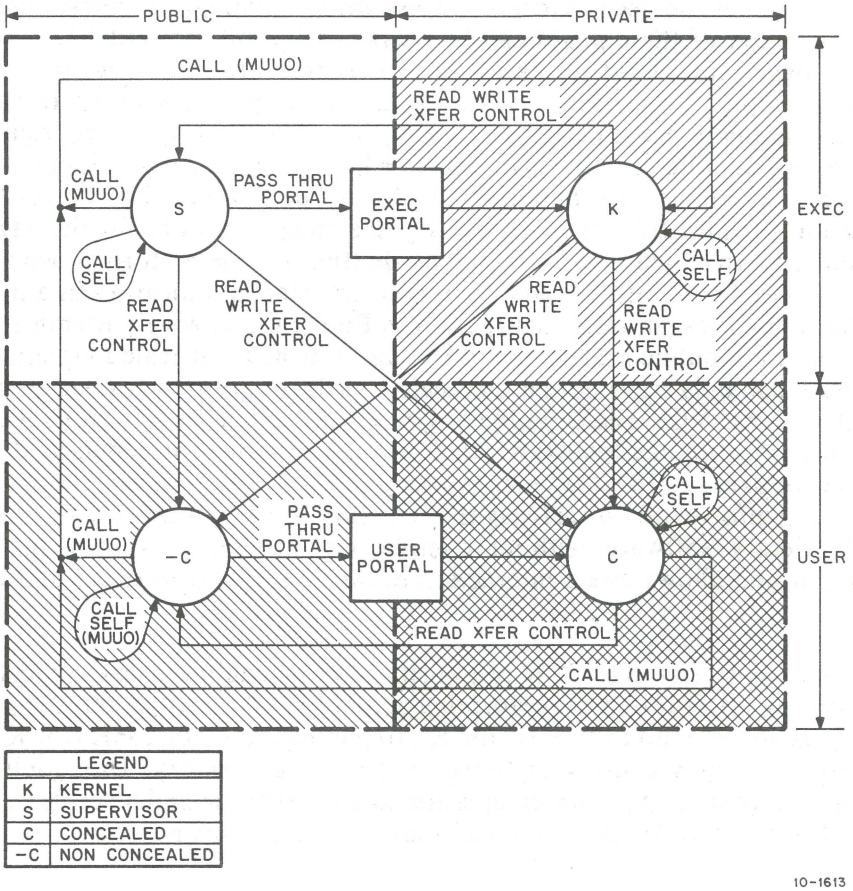



Figure 2-43 Mode Structure and Hierarchy

Basically, the programs of individual users operate in Public User mode, where the program can have access to one of two possible virtual address spaces. If KL10 paging is in effect, the user has access to a virtual address space of 256K words via an 18-bit virtual address, which may not be referred to by any other user (without the cooperation of the monitor). If KI10 paging is turned on, the program has access to a virtual address space of 256K addressed via a 18-bit virtual address, which as previously pointed out cannot be referenced by any other user without the monitor's cooperation. All instructions that do not compromise the integrity of the system are legal; this includes the following:

1. The halt instruction (JRST 4)
2. Any instruction attempting to affect the PI system (JEN)
3. Any I/O instruction directed at devices with device select codes below 740
4. Any reference to the concealed address space except for fetching of a portal instruction
5. All illegal instructions or op codes.

The user's address space (when KL10 paging is in effect) is divided into 32 (decimal) sections; each section contains 512 (decimal) pages and each page consists of 512 (decimal) words. The existence of these pages is nominally invisible to the user program. However, the amount of physical address space available is actually a number of these pages (at least one page), none of which need be contiguous either in physical core or in the user's virtual address space, although it is desirable from a machine standpoint to do so. Each of these pages can be designated public or writable by a 1 in bit 1 or 2, respectively, in the page table word for the page. Pages that are not designated writable cause an instruction, which attempts to write them, to trap to the monitor as a write protection violation page failure. A program running in pages designated public is in Public mode. A program running in pages not designated public is running in Concealed mode. Whether an instruction is performed from Public or Concealed mode is determined by the Last Instruction Public bit of the PC word (bit 7). The Last Instruction Public bit is copied from the Public bit of the page map word for the page from which the instruction was fetched. An instruction in Public mode (that is, one performed with the Last Instruction Public bit a 1 in the PC word), which attempts to transfer to a location in a nonpublic area not containing any Portal instruction, or an instruction in Public mode which attempts to read, write, or execute a location in a nonpublic area, traps to the monitor as a concealed violation page failure. A Public mode program can only transfer to a Concealed mode program by transferring to locations that contain Portal instructions. A Concealed mode program can read, write (if writing is allowed), execute, or transfer to any user location designated public. Concealed mode is provided to allow the loading of a proprietary software package together with a user's program and data while preventing the user's program from copying information discerning the structure of the proprietary software. This provides protection of proprietary software without complicated protective overlay or transfer schemes involving the monitor and allows direct interaction between user and software package with virtually no overhead.

The monitor operates in Exec mode. It is responsible for scheduling users, allocating memory and other facilities, servicing interrupts, and performing actual I/O. At any instant, the monitor has access to an effective address space of up to 8192K (for KL10 paging mode) or 256K (for KI10 paging mode) words and by overt action may address any portion of physical memory. The monitor can be divided into two parts: a normally small part, which operates in Kernel mode and is resident, and a larger part, which operates in User or Supervisor mode and may be swapped as necessary.

The Kernel mode part of the monitor handles the PI system, performs the direct I/O for the system, performs page management, and performs all other functions that affect all users of the system. The Supervisor mode part of the monitor performs the general management of the system (such as MUUO handling and dispatch) functions which affect only one user at a time. The Supervisor mode and Kernel mode of the monitor are analogous to the Public mode and Concealed mode of the user's programs in that the Supervisor runs in that part of the Exec address space designated public and the Kernel runs in that part of the Exec address space which is designated nonpublic; this simplifies illegal reference detection logic. Each address from 20 through 337,777 is broken up into pages, but these addresses can be made to refer to the same addresses in the physical memory space by making the virtual page address equal to the physical address portion in the corresponding page table entry. The entire Exec address space is broken into pages of 512 words which may be designated either accessable or not accessable, public or nonpublic, and writable or nonwritable and can be swapped out. An instruction in Supervisor mode that attempts to write into a page which is not writable will trap as a page failure. An instruction in Kernel mode may write into any location whether or not it is designated public. An instruction in Supervisor mode (that is, one performed with the Last Instruction Public bit a 1 in the PC word) that attempts to transfer to a location in an Exec nonpublic area not containing a Portal instruction traps to the monitor as a page failure. An instruction in Supervisor mode that attempts to read, write, or execute a location in an Exec nonpublic area traps to the monitor. In each instance, the trap is a Kernel violation page failure. A Supervisor mode program can only transfer, i.e., jump to a Kernel mode program, by transferring to locations that contain Portal instructions (JRST 1).

A Supervisor mode program can also reach Kernel mode (or any other mode) by performing an MUUO or other instruction that causes a trap, if the appropriate trap new PC word indicates that the next instruction is in Kernel mode. A Kernel mode program can read, write, execute, or transfer to any location designated public, i.e., in Supervisor mode; all instructions illegal in User mode are also illegal in Supervisor mode.

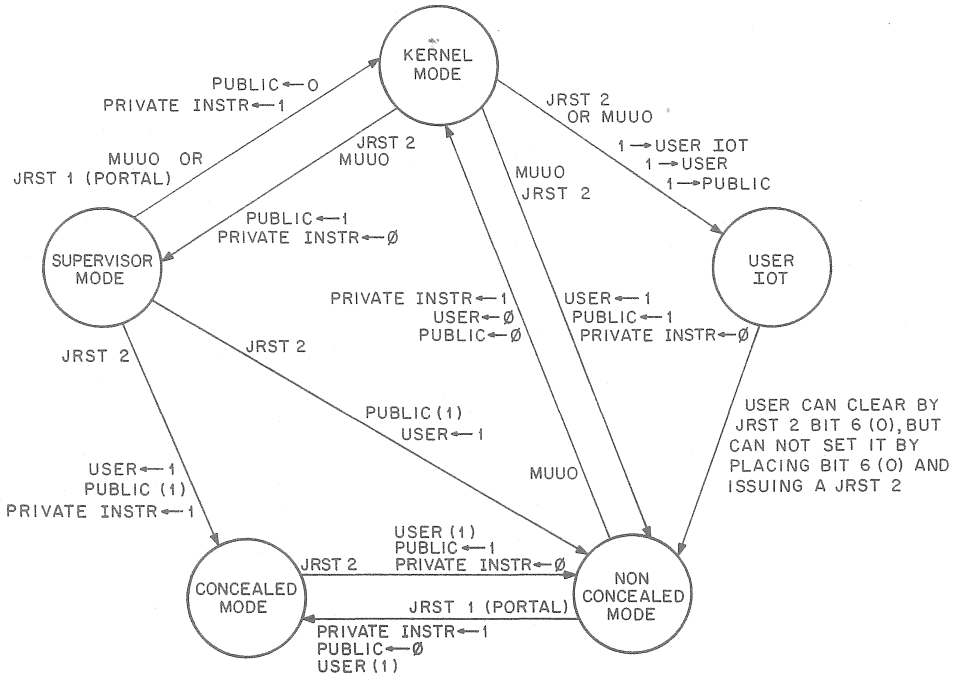
The mode control logic consists of the following:

User Mode
Public Mode
User IOT
Private INSTR
Miscellaneous Combinational Logic

The mode control exerts a powerful influence over the disposition of the processor. It monitors instruction fetches from Public mode to prevent illegal entry to either Concealed mode from User Public mode or Kernel mode from Supervisor. In addition, it detects the fetch of a Portal instruction and adjusts the state of the mode logic accordingly. The relationships between the various modes and their transfer instructions are shown in Figure 2-44. In general, two instructions allow flags that affect processor modes to be manipulated. These instructions are:

MUUO
JRST 2

Of the two, only the MUUO can cause transfers to any mode from any other mode. The JRST 1 (Portal 1) simply allows entry to a Private mode from a Public mode. Each time an instruction fetch is specified and the reference is to a nonpublic page, a test for illegal entry must take place to maintain integrity in the system.




| BIT ASSIGNMENTS | | | | |
|---|---|---|---|---|
| CONTROL OF | BIT05 | BIT06 | BIT07 | NOTES |
| USER MODE | X | | | |
| USER IOT | | X | | IN USER MODE |
| PUBLIC MODE | | | X | |
| PREVIOUS CONTEXT | | X | | IN EXEC MODE |

10-1614

Figure 2-44 Mode Transfer

Referring to Figure 2-44, assume a User Public program has been started by a monitor routine that performed a JRST 2 (a jump and restore flags). To place the processor in User Public mode, bits 7 and 5 of the flag's PC word must be set; this results in the setting of Public mode and user mode, respectively. The processor is now in User Public mode. Assume that the User executes some miscellaneous instructions and then performs an instruction fetch from a nonpublic area. The following test takes place: instruction fetch is decoded from the microinstruction MEM field or specified as a prefetch in the DRAM A field. The E/M Interface asserts EBOX READ and loads the address into VMA. Note that if a reference to a private address for a read or write of data is attempted, it page fails on the attempted reference because PAGE TEST PRIVATE is asserted. However, in this case, the fetch must be allowed from the private address space. Its identity is checked in the EBox and, if it is not a JRST 1 (portal), a page failure occurs on the very next memory reference. This is implemented by delaying generation of the signal that would cause a page failure to be generated by the MBox (PAGE ILLEGAL ENTRY), until the instruction fetch is completed. When the MBox responds with the level – PAGE TABLE PUBLIC (PT PUBLIC), this signal with the MB response sets PRIVATE INSTRUCTION. This causes the generation of PAGE ILLEGAL ENTRY. If the instruction which is decoded by the hardware is not a Portal, Public mode remains set maintaining PAGE ILLEGAL ENTRY, which enables a page fault on the next MBox reference for whatever reason. If the instruction fetched is a portal (JRST 1), then Public is cleared and the processor enters Concealed mode.

All user references and concealed references are paged. The difference between the types of paged references is that user paged references are public while concealed references are nonpublic when referencing the concealed address space and may be public when referencing the users address space. Executive references are paged, this includes both Kernel and Supervisor references. Supervisor mode programs must be capable of reading both User Public and User Concealed address spaces. To bypass the portal mechanism normally necessary for any public program to reference a nonpublic program area, a bypass exists, which is under control of the Kernel; when operational, the Supervisor is allowed to read and possibly write the concealed area as necessary, remembering, of course, that the supervisor is part of the operating system and it is performing job-related tasks within that context.

Normally a public program is only allowed to fetch an instruction from a nonpublic area and this instruction must be a portal (JRST 1) instruction; however, this is necessary for the supervisor to perform its system tasks. Basically, the process for checking a User Public program's reference to a concealed address is as follows. The mode is User Public and an instruction fetch begins. EBOX REQUEST is issued to the MBox, together with the appropriate paging qualifiers and any other appropriate signals. The MBox performs the necessary check of the page descriptor bits; then the state of the Public bit in the page table is asserted over the E/M Interface where, together with signal MB XFER and a signal indicating an instruction fetch is being performed, it is used to enable the setting of Private instruction. If the Page Table Public bit is off, Private instruction is set on the clock occurring concurrently with MBox response. PAGE ILLEGAL ENTRY is not asserted. The response given by the MBox was given at the same time it placed the desired instruction onto the cache data lines; this instruction is now in ARX. If the instruction is indeed a portal instruction (JRST 1), the Public mode will be cleared, removing the PAGE ILLEGAL ENTRY signal. This procedure then has effected the proper entry to Concealed mode. If the instruction was not a Portal, then the PAGE ILLEGAL ENTRY signal will not be removed nor will Public be cleared, which constitutes an illegal state in the EBox. On the very next MBox request by the EBox (providing VMA AC REF is false), a page fault occurs and an appropriate code is placed in the EBus register in the MBox identifying the disposition of this fault. This will shortly be followed by a trap to the operating system as a concealed violation page failure. This same procedure is applied to a Supervisor reference to the Kernel address space, and in this way the integrity of the system is protected from any unwarranted references. Figure 2-45 shows a typical layout of the virtual address space for the various modes. The space shown is for KI10 paging mode (256K, made up of 512 pages numbered 0–777 octal). Any program can address locations 0–17 as these are in a fast memory block and are completely unrestricted (although the same addresses may be in different blocks for different programs). The Public mode user program operates in the public area, part of which may be write protected. The Public program cannot access any locations in the concealed area, except to fetch instructions from prescribed entry points. The Concealed mode user program has access to both the public and concealed areas, but it cannot alter any write protected location whether public or concealed; fetching an instruction from the public area automatically returns the processor to Public mode. The Supervisor mode program is confined within the paged area of the address space. Part of the public area in this space may be write protected, but the program can read information in the concealed area. It cannot, however, alter any location in a concealed area, whether that area is write protected or not. Pages 340–377 constitute the per process area, which contains information specific to individual users and whose mapping accompanies the user page map. In other words, the physical memory corresponding to these virtual pages can be changed simply by switching from one user to another, rather than the operating system changing its own page map. The Kernel mode program can access all of the unpaged area without restriction and can reference all of the accessible paged area both public and concealed, with the usual restriction that it cannot alter a write protected area. As in the case of Concealed mode, fetching an instruction from a public area returns control to Supervisor mode.

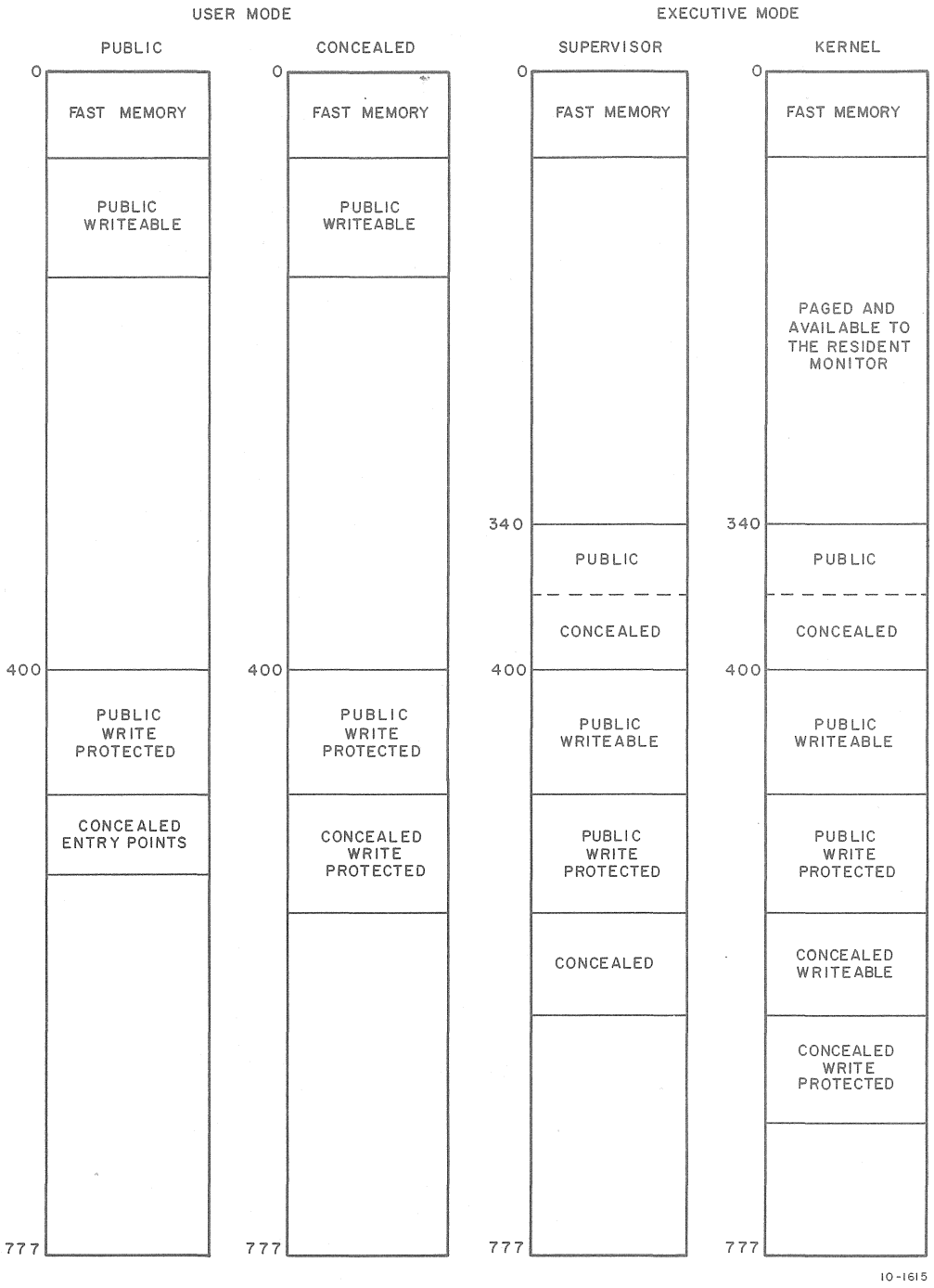



Figure 2-45 Typical Virtual Address Space Configuration

### 2.7.1 Mode Initialization – Private Instruction

When the KL10 system is powered up, the power control issues the signal CROBAR for approximately 5 seconds. This results in the generation of RESET, which causes LEAVE USER to be asserted. LEAVE USER enables the clearing of USER, USER IOT, and PUBLIC and sets PRIVATE INSTRUCTION. This action places the KL10 in Kernel mode. Referring to Figure 2-46, each time an instruction is fetched from either Fast Memory or Core Memory (via MBox), the private instruction recirculation path is broken (Figure 2-47).

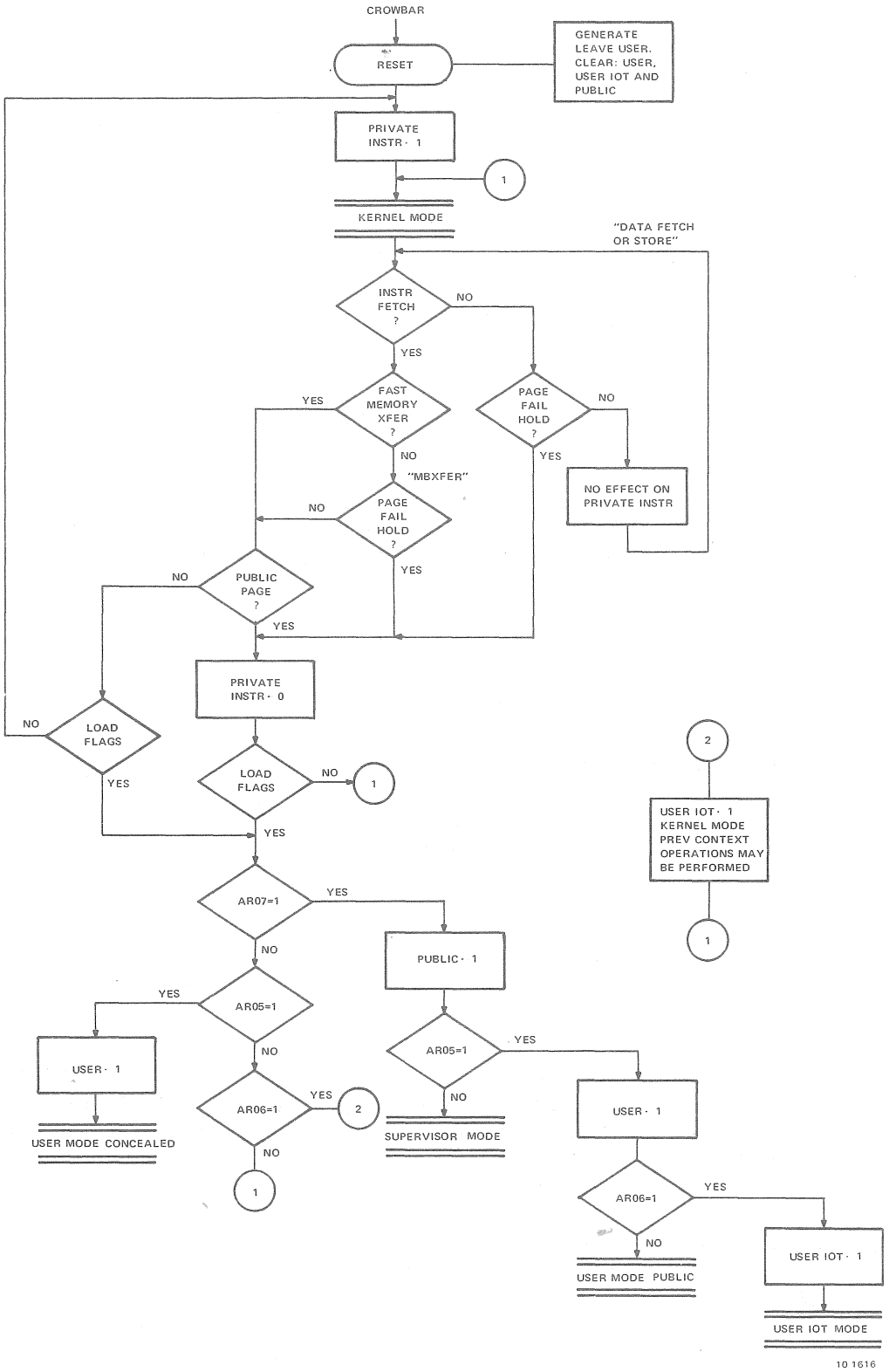



Figure 2-46 Mode Initialization

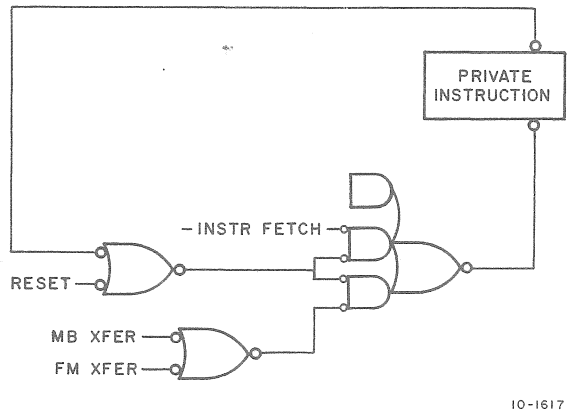



Figure 2-47 Private Instruction Recirculation Path Simplified

If the instruction is fetched from a nonpublic address space (-PUBLIC PAGE), or the mode of the machine is not public (-PUBLIC), then the private instruction is enabled to be set once again (Figure 2-48).

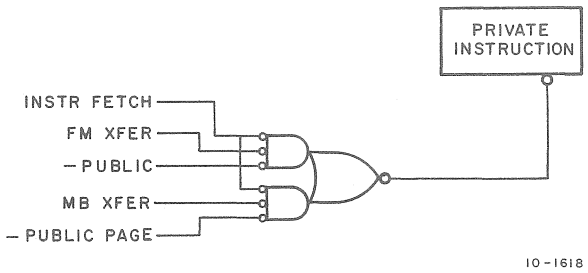



Figure 2-48 Setting Private Instruction

Note that if data is read or written, the upper recirculation leg (Figure 2-48) is not disabled. The Private Instruction flip-flop is used with additional logic that (with the exception of previous context references) detects references to Public mode; together, these elements detect entry to a privileged address space. The Kernel may access any part of the address space regardless of its type. Because the Kernel does not operate in Public mode, illegal entry has no significance.

### 2.7.2 Loading Flags and Changing Mode

Two instructions can change the mode of the machine. These instructions are MUUO and JRST with AC bit 11 set, i.e., JRSTF.

As indicated in Table 2-6, AR bits 05 and 07 are used in various combinations to enter appropriate submodes.

**Table 2-6 Flags Effecting Mode**

| Instruction being performed is MUUO,JRSTF (See Note) | | | | | Major Mode | | | |
|---|---|---|---|---|---|---|---|---|
| Enable PREVCONTXT | User IOT | Flag Bits AR06 | Effecting Modes AR05 | Effecting Modes AR07 | Exec Submodes Kernel | Exec Submodes Super | User Submodes Concealed | User Submodes Public |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| | | N/A | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

NOTE
A JRSTF may not clear user by placing bit 05 (0) but an MUUO may.

In addition, for Direct User I/O, bit 06 (USER IOT) is available to allow the running of privileged user programs with paging in effect. This mode provides some protection against partially debugged monitor routines, and permits running infrequently used device service routines as a user job. Direct control by the user program of special devices is particularly important in real-time applications. A special MUUO is available to enter USER IOT mode, but it is privileged because time-sharing is effectively stopped while in this mode.

### 2.7.3 User Public Mode

Once the processor is in User Public Mode (Figure 2-49), the user program can freely read and write data in the user public address space with the cooperation of the system. When demand paging is in effect, each reference to a previously unreferenced page causes an access page fault. The operating system page manager must assess the fault, obtain the page from mass storage, and build an entry in the user's process table.

Assuming that the current user's process table (PAGE TABLE PART) is initially clear, the first reference causes a NOT IN CORE page fault (Figure 2-50). The EBox, upon detecting the PAGE FAIL HOLD signal from the MBox, enters a microcode page fault handling routine that communicates the failure to the operating system. Next, the page manager or a related routine requests the page from mass storage. When the page is in core, the appropriate process table is constructed and the reference by the user program may be tried once again (Figure 2-51).

The MBox performs the reference to the process table; the use bits now reflect the following:

PAGE IS IN CORE A = 1
PAGE IS WRITABLE W = 1
PAGE IS PUBLIC P = 1
PAGE SHOULD BE CACHED C = 1

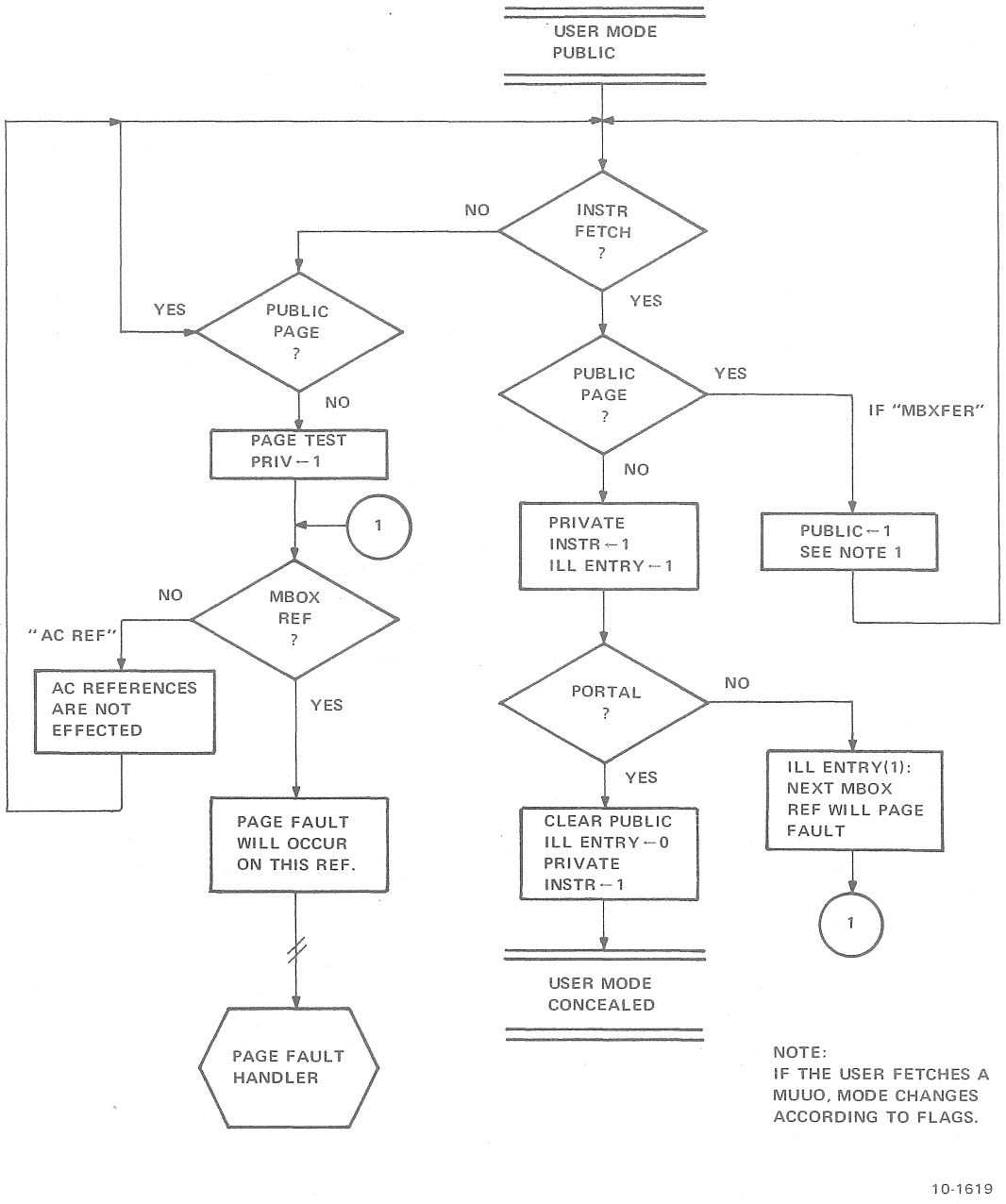



Figure 2-49 User Mode Functional Flow

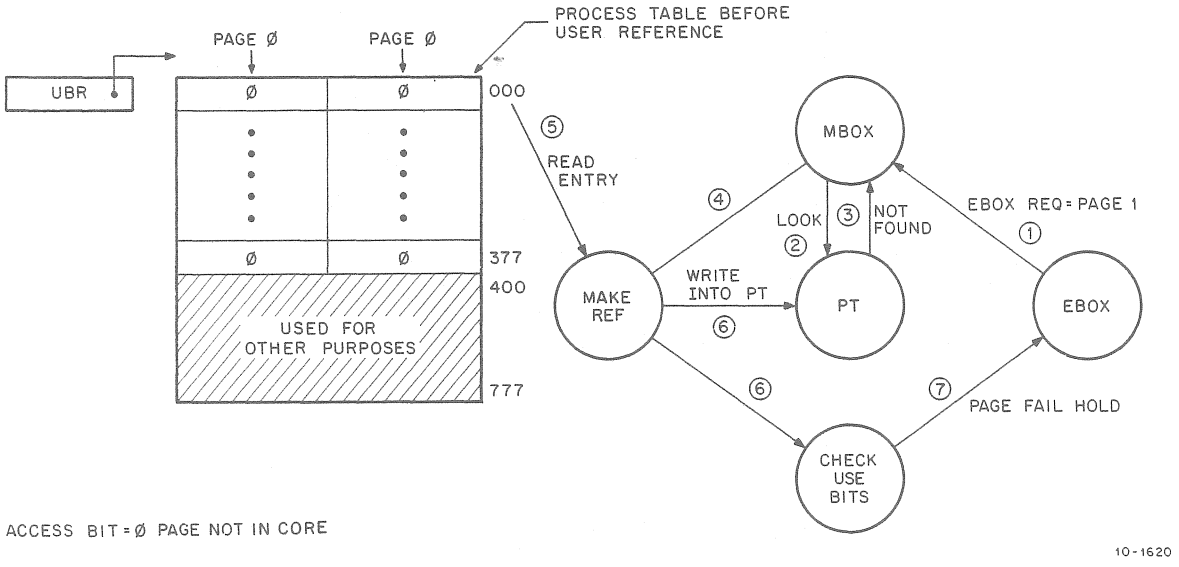



Figure 2-50 User Mode Public Initial Reference

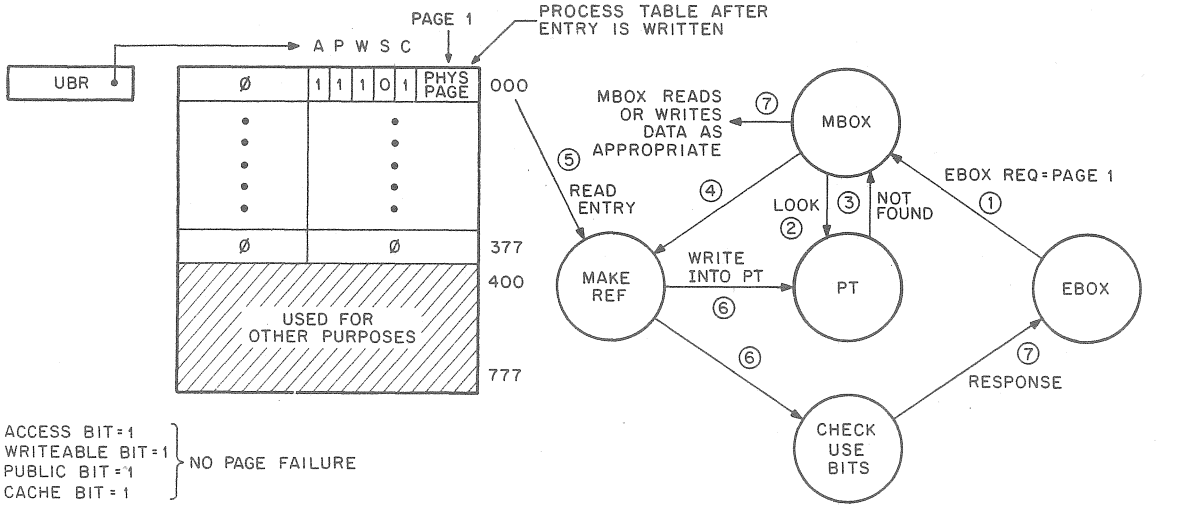



Figure 2-51 User Mode Public Second Reference

The entry (one of eight half-word entries fetched) is written into the page table in the MBox, the MBox then performs the data reference part of the request. This can involve reading or writing and depends upon the type of EBox request. During the reference, PAGE ILLEGAL ENTRY was not asserted because the reference made by the user program was to a public page and it was for an instruction.

**2.7.3.1 Entry from User Public Mode to User Concealed** – To correctly enter User Concealed mode, the User Public program must execute a Portal instruction (Figure 2-49) from the concealed address space. The EBox generates the EBox request and provides the MBox via VMA with the concealed address. The MBox either finds the page entry and use bits in the MBox Page Table (hardware) or performs a refill cycle to obtain it from core memory. Figure 2-52 shows the typical Concealed Page Table format. Presumably, the entry is nonpublic and write protected, and may or may not be cached.

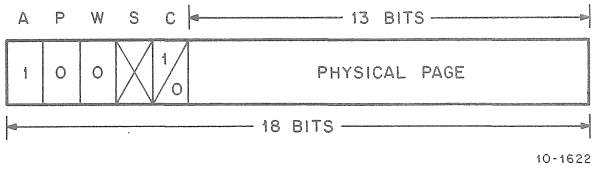



Figure 2-52 Typical Concealed Page Table Format (Half Table Entry)

The MBox asserts PT PUBLIC (0) and MBOX RESPONSE IN to the EBox. Referring to Figure 2-48, MB XFER resulting from MBox response and -PUBLIC PAGE resulting from PT PUBLIC (0) enables the setting of Private instruction. The instruction fetched by the MBox is in ARX at this time. If it is a JRST 1 (Portal), its execution clears Public and the processor enters User Concealed mode. If the instruction is anything else, Public remains set and the next MBox reference occurs with PAGE ILLEGAL ENTRY true, PUBLIC PRIVATE INSTR (1); this causes a page failure.

**2.7.3.2 Concealed Violation Data Reference** – If a User Public program references the concealed address space for read or write, PAGE TEST PRIVATE is asserted during the EBox request and results in an immediate page fault. Page Test Private is a signal composing Public and -INSTR FETCH.

### 2.7.4 Restoration of Programs by the Supervisor

The Supervisor portion of the operating system deals with those tasks which affect one job at a time. It must, therefore, have the ability to restore various programs to an operational status, e.g., by executing a JRST 2 instruction that picks up a PC word consisting of the appropriate flags in the left half and a virtual PC in the right half of the word.

**2.7.4.1 Restoring a Concealed Program** – The Supervisor may restore a concealed program providing it also sets User. Referring to Figure 2-53, while executing a JRST 2 instruction, LOAD FLAGS is derived from the presence in the magic number field of bit 04, and this together with -User (User is off in Supervisor mode), and AD bit 05 (which will set User) generated CLR PUBLIC. Thus, on the next clock pulse, Public clears and User sets, restoring Concealed mode. Figure 2-54 shows the necessary conditions. Note that performing a JRST 2 cannot generate Leave User, unless the processor is in Kernel mode.

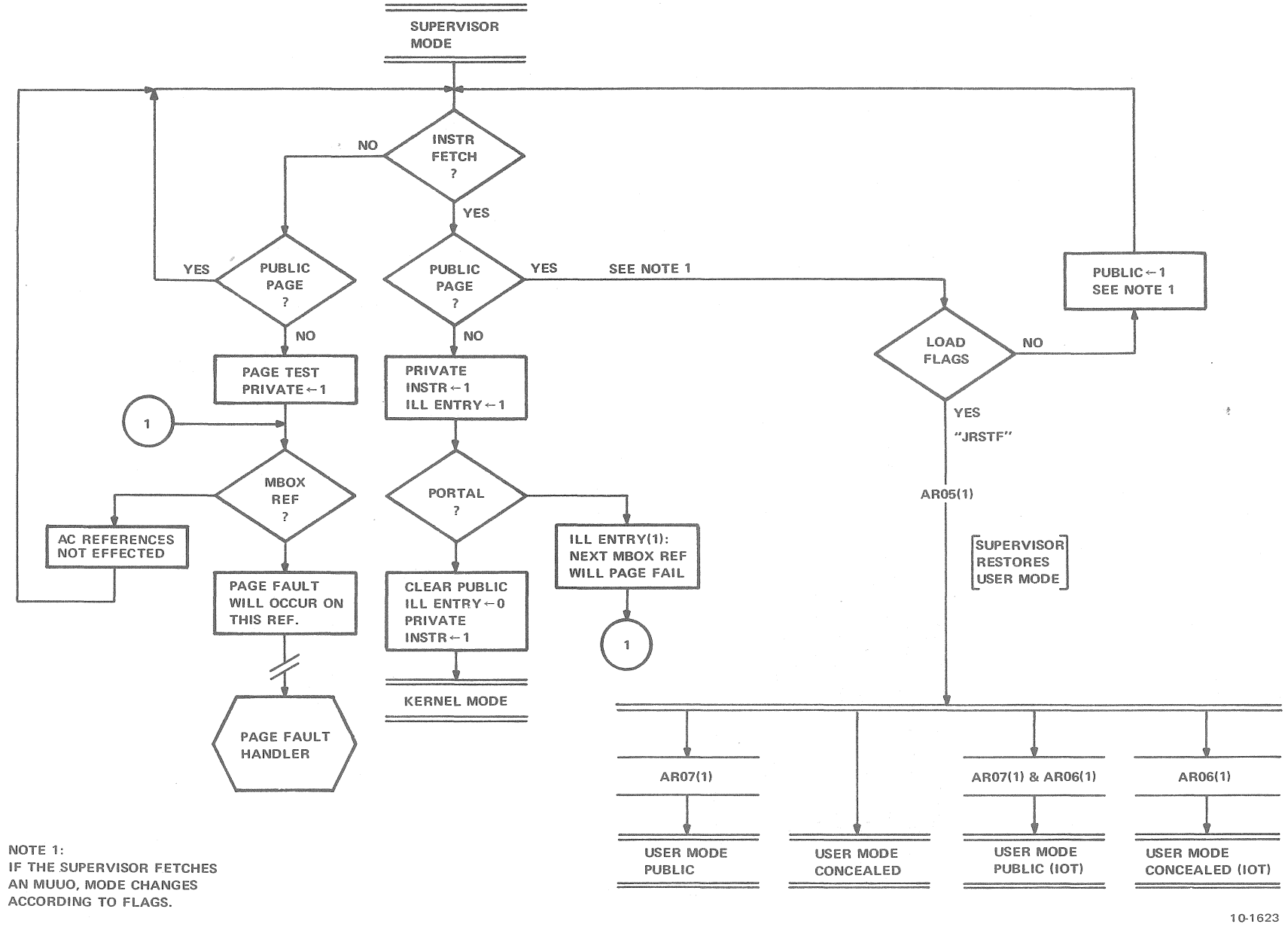



Figure 2-53 Supervisor Mode Functional Flow

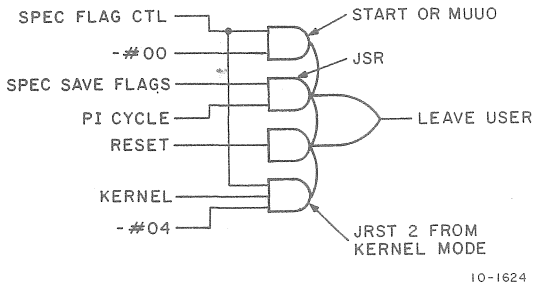



Figure 2-54 Leaving User

**2.7.4.2 Restoring a Kernel Program** – The restoration of a Kernel mode program from Supervisor mode is somewhat different in its mechanics than the restoration of the Concealed program. Basically, the Supervisor must first perform a JRST 2 instruction; this instruction restores all flags except for Public. The JRST must enable the fetching of a Portal instruction that clears Public, placing the machine in Kernel mode. This is a safeguard in the event that the Supervisor may, in error, try to restore some random set of bits and cause the Kernel to be disturbed. In addition, it forces entry to Kernel mode at a known and unique entry point. Figure 2-55 shows that it is not possible for a JRST 2 instruction to clear Public while not setting User as well. Note that a JRST 2 instruction does not generate Leave User unless it is given in Kernel mode. The conditions which enable Leave User are indicated on Figure 2-54.

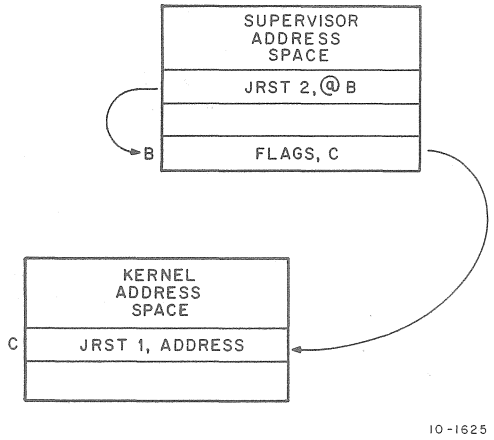



Figure 2-55 Restoring Kernal Program

**2.7.4.3 Restoring a User Public Program** – To restore a User Public program, the Supervisor gives a JRST 2, which sets User. This is the only requirement because both Supervisor and the User Public program run with Public set. The special field function SPEC FLAG CTL, together with magic number 04(1) enables SPEC/LOAD flags which, with AD bit 05, enables User to set on the next clock.

**2.7.4.4 Saving Flags and Leaving User** – It is not generally known at just what moment an interrupt will occur with respect to execution of a given instruction. The microprogram governs the handling of interrupts by looking for interrupts only at certain times. In general, an interrupt is sampled for between each instruction and during certain classes of instructions. The following classes of instructions can be interrupted:

Byte Instructions
Block Transfer Instruction
Input/Output Instructions

In addition, for any instruction, an interrupt is sampled during the portion of the microprogram that performs indirect addressing (INDRCT). An interrupt has higher priority than a Page Fault and thus, upon entry to the Page Failure microroutine, an interrupt condition is tested for; if found, a dispatch to the microroutine for interrupt handling is given.

When an interrupt occurs and the PI logic has completed the handshake, it informs the EBox by asserting a signal PI READY. This results in the microprogram generating a skip to a microinstruction that asserts SPEC/SET PI CYCLE. As a result, Kernel cycle (normally false as long as PI CYCLE is clear) sets, and MCL VMA PUBLIC is disabled. This is necessary to disable the MCL PAGE ILLEGAL ENTRY signal when PI CYCLE sets because the interrupt instruction, which will be fetched from a Kernel address, must not generate a page fault.

When the interrupt instruction is being fetched, User and Public may be set, or Public alone may be set. In the last instance, a page fault would result if some action were not taken to prevent it. This is why MCL PAGE ILLEGAL ENTRY is disabled (by setting PI CYCLE). At the time of the interrupt, the state of the current user ACs is unknown. The instruction in 40 + 2n, therefore, must not disturb the ACs in any way while transferring the flags and PC to the Kernel mode subroutine. Therefore, JSR is a likely instruction for use in 40 + 2n. The JSR instruction causes the flags and current PC to be stored in the effective address of the JSR instruction and then enters the subroutine by performing an instruction fetch from E + 1. After calculating the effective address for the JSR instruction, the microprogram performs a write test which, if successful, is followed by a branch via the DRAM J field to the executor. Now the flags and PC are loaded to be copied into the AR for storage and are then disabled. The microinstruction asserts SPEC FLAG CTL; this with PI CYCLE generates LEAVE USER, which detaches the feedback path for User, User IOT, and Public. In addition, if User were set, User IOT would be set at this time and represent "Previous Context User." This is an indicator to the hardware that previous context references must be in User mode. In any event, the processor enters Kernel mode and begins to handle the interrupt.

**2.7.4.5 User Concealed** – This mode is useful for running certain proprietary programs in User mode without allowing the user to discern the composition of the concealed program. For example, assume a user has developed a program that performs circuit analysis. The user is a time-sharing house and desires that this program be available to users for execution only, that is, the user must not be able to read or write into this program.

In some computer systems, complex overlays in core memory are necessary to assure concealment of the program from its users. In the KL10, this program has been solved by creating two submodes from User mode, each with separate powers and each separate from the other. Both modes, however, run with User on. Figure 2-56 indicates the hierarchial structure present in the KL10 processor. The User Public program can only transfer to a concealed program at a selected entry called a Portal. The instruction fetched must be a Portal instruction (JRST 1). The concealed program can read or write data to the Public area. Figure 2-57 is the Concealed mode functional flow diagram.

| INSTRUCTION | EFFECT ON USER, PUBLIC, AND PRIVATE INSTR FLAGS | PREVIOUS MODE | NEW MODE |
|---|---|---|---|
| MUUO | PUBLIC AND USER CLEAR, PRIVATE INSTR SETS | USER | KERNEL |
| MUUO | PUBLIC CLEARS, PRIVATE INSTR SETS | SUPERVISOR | KERNEL |
| MUUO | USER CLEARS | CONCEALED | KERNEL |
| MUUO | NO EFFECT | KERNEL | KERNEL |
| PORTAL (JRST 1) | PRIVATE INSTR SETS, PUBLIC CLEARS | USER | CONCEALED |
| PORTAL (JRST 1) | PRIVATE INSTR SETS, PUBLIC CLEARS | SUPERVISOR | KERNEL |

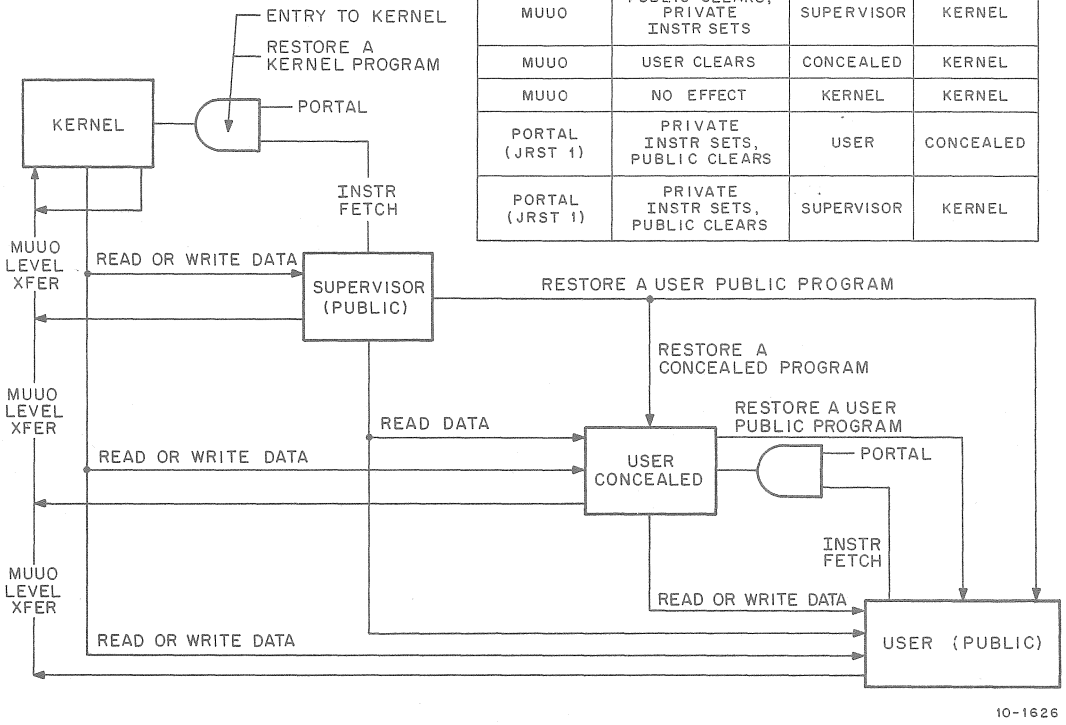



Figure 2-56 Mode Hierarchy

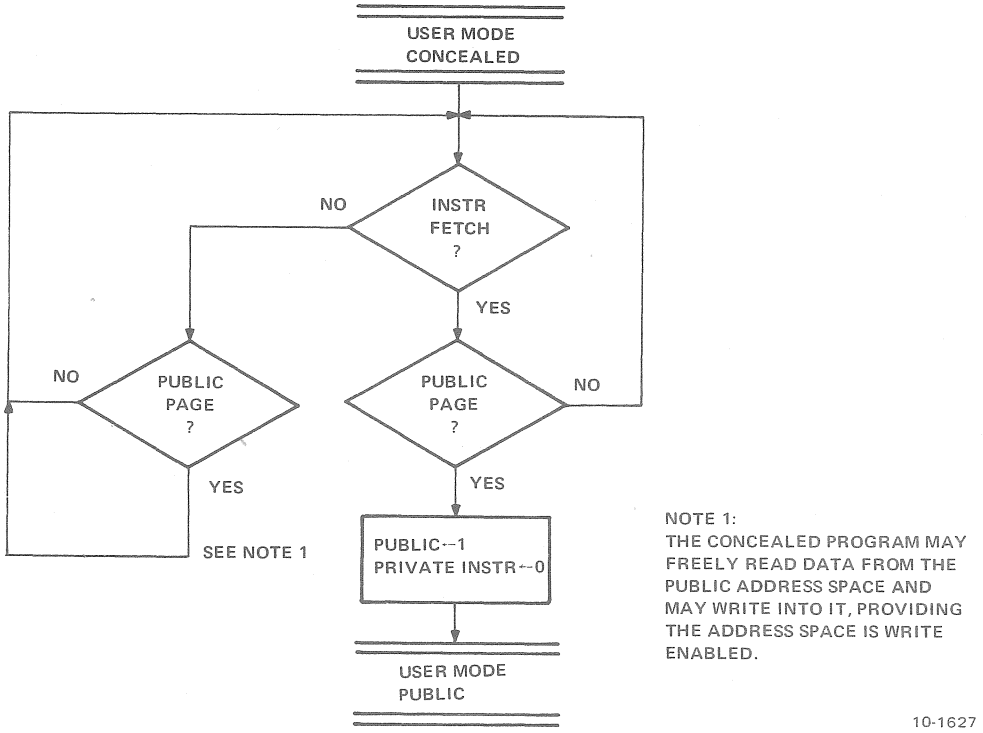



Figure 2-57 Concealed Mode Functional Flow

## 2.8 ADDRESS PATHS

The address paths contained within the EBox are illustrated in Figure 2-58. These paths are implemented to facilitate the formation of the appropriate MBox virtual address. This address is translated by the MBox for KI paging mode and by the microprogram and the MBox for KL paging mode. The MBox can generate the following two basic forms of physical addresses:

1. Refill Address (Relocated)
2. Physical Page Address (Paged)

The VMA serves as a source of data when loading the following MBox registers:

1. User Base Register (UBR)
2. Executive Base Register (EBR)
3. Cache Clearer (CCA)

In addition, it serves as an address and data source when loading the cache refill RAM. As indicated in Figure 2-59, the VMA has the three following basic sources of input:

1. Previous Context Section register (PCS)
2. Virtual Memory Address Adder (VMA AD)
3. Adder (AD)

The following two major addressable areas are addressed by the VMA:

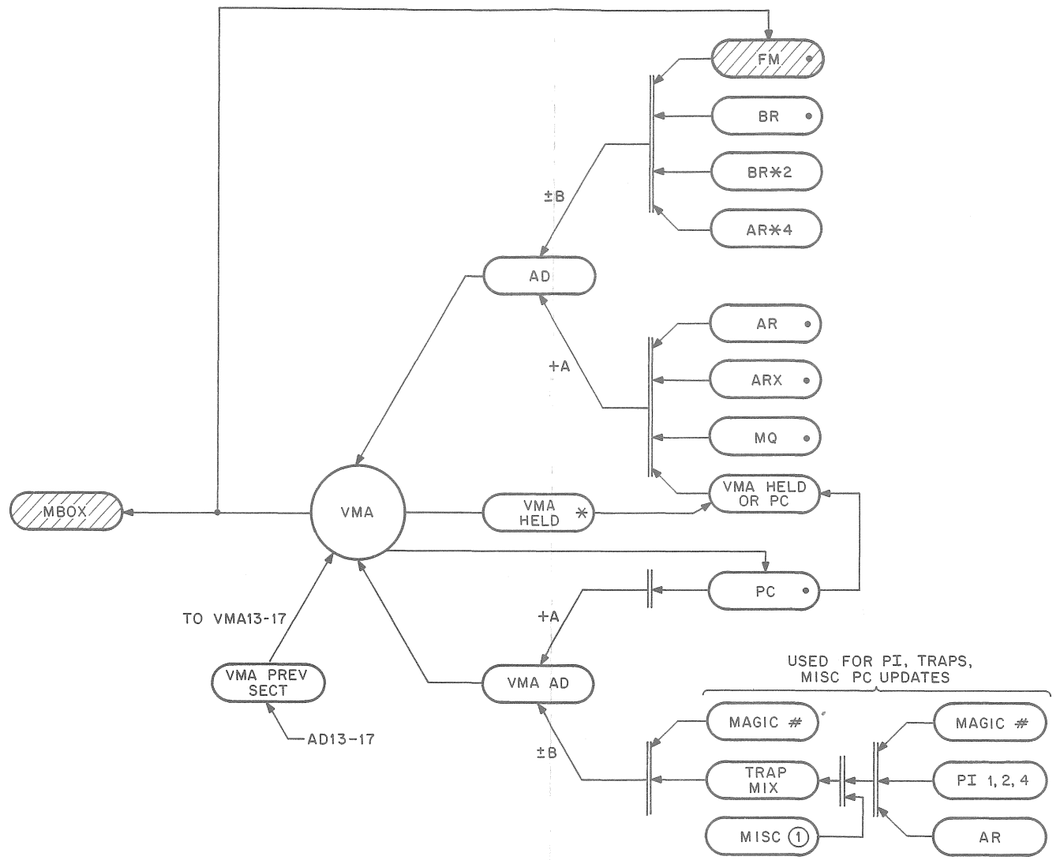
1. MBox
2. Fast Memory (FM)

The MBox may be addressed logically by two types of addresses. Within each type (18-bit and 23-bit addressing) is a class of process table addresses. These addresses are identified to the MBox by the qualifiers asserted during the EBox request (Table 2-7).

**Table 2-7 Virtual Address Classification**

| Type of Address | Class | Addressing Information Supplied |
|---|---|---|
| 18-Bit | KI Paged | VMA 13–17 = 0<br>VMA 18–26 = Virtual Page<br>VMA 27–35 = Quad Word |
| 18-Bit | KI Process Table Reference | VMA 13–17 = MBox Ignores<br>VMA 18–26 = MBox Ignores<br>VMA 27–35 = Process Table Word |
| 23 | KL Paged | VMA 13–17 = Virtual Section<br>VMA 18–26 = Virtual Page<br>VMA 27–35 = Quad Word |
| 23 | KL Process Table Reference | VMA 13–17 = MBox Ignores<br>VMA 18–26 = MBox Ignores<br>VMA 27–35 = Process Table Reference |

**NOTE**
**There are several other special VMA combinations. These will be covered elsewhere.**




• – THESE REGISTERS MAY ACTIVELY BE INVOLVED IN SOME FORM OF ADDRESS. CALCULATION WHICH WILL ULTIMATELY BE PLACED INTO VMA.

* – USED DURING KL10 STYLE PAGING ONLY

(1) – USED TO FORCE PC+1 OR PC+2

| | VMA | | | | |
|---|---|---|---|---|---|
| ADDRESS TYPE | VMA 13–17 | VMA 18–35 | SOURCE OF ADDRESS | BY WAY OF | |
| 18 BIT | PC 13–17 | VMA AD 18–35 | PC | VMA AD | KI PAGING |
| 18 BIT | RECIRCULATED | AD 18–35 | E | AD | |
| 18 BIT | CLEAR | VMA AD 18–26=0, 27–35 | TRAP | VMA AD | |
| 18 BIT | CLEAR | VMA AD 18–26=0, 27–35 | PI OR SPECIAL | VMA AD | |
| 18 BIT | RECIRCULATED | AD 18–35 | @ | AD | |
| 18 BIT | RECIRCULATED | AD 18–35 | MISC | AD | |
| 23 BIT | VMA PREV SECT 13–17 | AD 18–35 | PC 13–17 OR E BUS | VMA PREV SECT AD | KL PAGING |
| 23 BIT | AD 13–17 | AD 18–35 | E (EXTENDED) | AD | |
| 23 BIT | CLEAR | VMA AD 18–26=0, 27–35 | TRAP | VMA AD | |
| 23 BIT | CLEAR | VMA AD 18–26=0, 27–35 | PI OR SPECIAL | VMA AD | |
| 23 BIT | PC 13–17 | VMA AD 18–35 | PC | VMA AD | |

| | | USER | | ~ USER | |
|---|---|---|---|---|---|
| | | KI PAGING MODE | KL PAGING MODE | KI PAGING MODE | KL PAGING MODE |
| PUBLIC | VMA AC REF | VMA 13–33=0<br>VMA 32–35=<br>FM ADDRESS<br>[USER PUBLIC] | VMA 13–33=0<br>VMA 32–35=<br>FM ADDRESS<br>[USER PUBLIC] | VMA 13–33=0<br>VMA 32–35=<br>FM ADDRESS<br>[SUPERVISOR] | VMA 13–33=0<br>VMA 32–35=<br>FM ADDRESS<br>[SUPERVISOR] |
| | ~ VMA AC REF | VMA 13–17=0<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[USER PUBLIC] | VMA 13–17=SECT<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[USER PUBLIC] | VMA 13–17=0<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[SUPERVISOR] | VMA 13–17=SECT<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[SUPERVISOR] |
| ~ PUBLIC | VMA AC REF | VMA 13–33=0<br>VMA 32–35=<br>FM ADDRESS<br>[USER CONCEALED] | VMA 13–33=0<br>VMA 32–35=<br>FM ADDRESS<br>[USER CONCEALED] | VMA 13–17=0<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[KERNEL] | VMA 13–33=0<br>VMA 32–35=<br>FM ADDRESS<br>[KERNEL] |
| | ~ VMA AC REF | VMA 13–17=0<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[USER CONCEALED] | VMA 13–17=SECT<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[USER CONCEALED] | VMA 13–17=0<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[KERNEL] | VMA 13–17=SECT<br>VMA 18–26=<br>VIRTUAL PAGE<br>VMA 27–35=<br>QUAD WORD<br>[KERNEL] |

NOTE: THIS IS THE GENERAL FORMAT ONLY.

10-1628

Figure 2-58 EBox Address Paths
Simplified Path Diagram

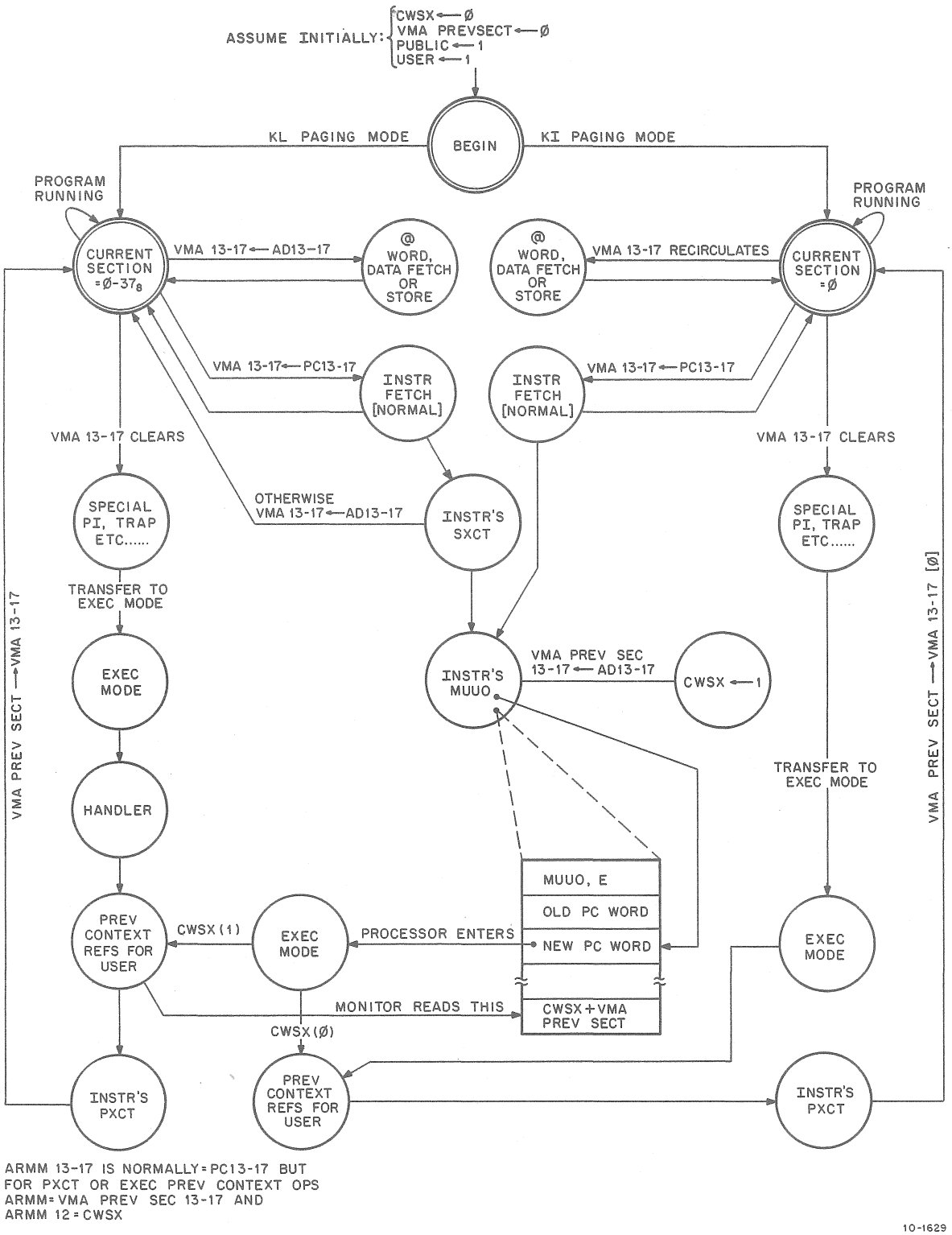



Figure 2-59 Typical VMA 13-17 Manipulations

For these process table references the EBox supplies valid addressing information only on VMA bits 27–35. The MBox replaces VMA 13–26 with the PMA mixer 14–26 to generate a proper physical address.

## 2.9 DATA PATHS

The specific address and data paths in the EBox are illustrated in Figure 2-60.

The functional elements in the address path between the VMA at the MBox/EBox Interface and the primitive address source involved in forming the virtual addresses are:

Virtual Memory Address Register (VMA)
VMA Held or PC Mixer
VMA Held Register
VMA Previous Section
VMA Mixer
VMA Adder (VMA AD)
SCD TRAP Mixer
ADDER (AD)
Arithmetic Register Extension (ARXML)
Arithmetic Register (AR)
Program Counter (PC)
Microinstruction Number Field
Other Miscellaneous EBox Registers

The appropriate virtual address is formed by the VMA under explicit control of the VMA control and the microprogram.

### 2.9.1 Virtual Memory Address Register

The VMA is loaded during an EBox request and remains latched until the MBox responds (Figure 2-61). The VMA is a 23-bit register that accepts input from a double mixer arrangement. Thus, the incrementing or decrementing is performed in the register itself. When both VMA SEL 2 and 1 are clear, the lower mixer is enabled into VMA. The level VMA ← AD selects AD as input. The default is VMA AD as input.

In general, the VMA AD contains one of the following:

PC (18–35)
PC+1 (18–35) + (1)
PC+2 (18–35) + (2)
Process Table Address (27–35)
Fast Memory Address (32–35)

The AD contains one of the following:

Effective Address
@ Word Address
Some Special Address

The VMA Held register is loaded during each MBox memory request [MEM 02 (1)]. The left-most 12 bits of VMA Held are loaded with the request qualifiers, type of paging, context of the reference, and various other signals asserted during the request. The right-most 23 bits of VMA are preserved in VMA Held right. The contents of VMA Held are used during KL Paging mode to buffer the request state while the page fault handler sets up an MBox Page Refill cycle. This operation is generally described in Subsection 1.2.4.2, KL Style Paging and is described later in greater detail.

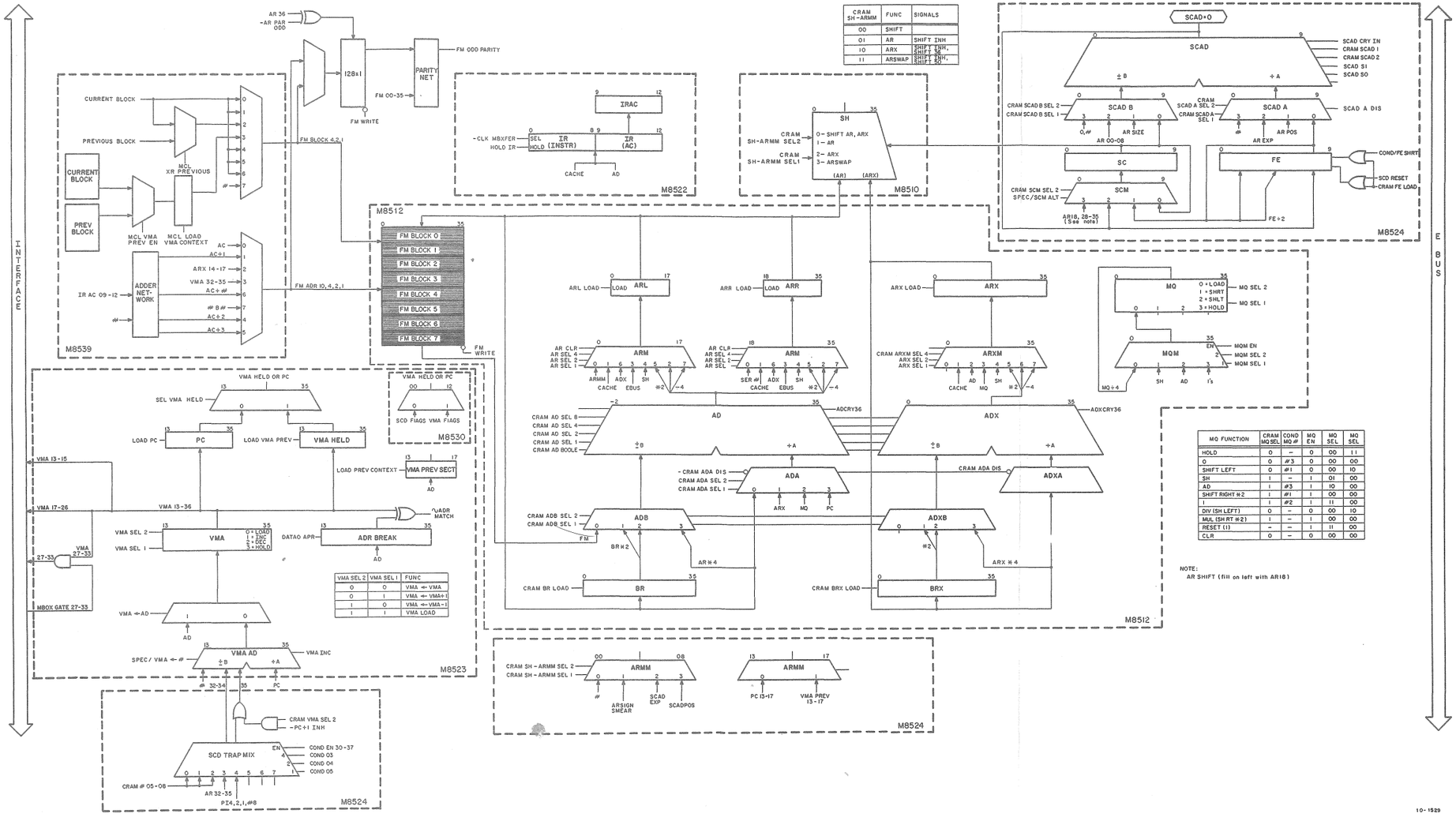



Figure 2-60 EBox Data and Address Paths

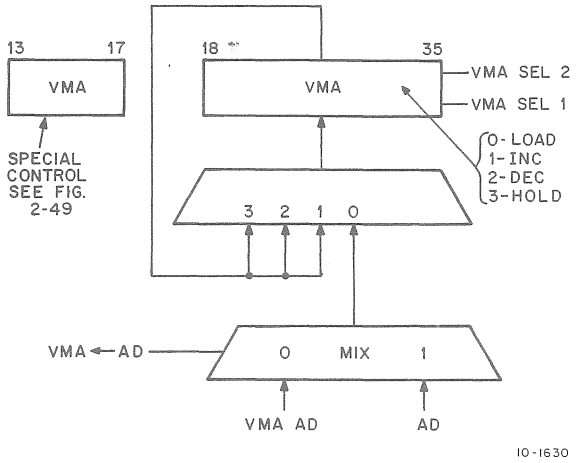



Figure 2-61 VMA Inputs

The first three selections (Subsection 3.2.1) enable the output of VMA into the VMA register for any of the following select codes:

VMA SEL 2 (0) and VMA SEL 1 (1) – Increment
VMA SEL 2 (1) and VMA SEL 1 (0) – Decrement
VMA SEL 2 (1) and VMA SEL 1 (1) – Hold

### 2.9.2 Program Counting

The PC is normally loaded from VMA at NICOND Dispatch, except when PI Cycle is true; this prevents alteration of PC during priority interrupt handling. When the processor is ready to fetch an instruction in sequence, the incremented PC address is supplied to VMA via the VMA AD. The VMA then supplies the address to PC. Thus, program counting is effected by the loop of PC, VMA AD, VMA, and back to the PC (Figure 2-62).

When a skip condition is satisfied, this loop is used to advance the PC during the instruction execution cycle. The PC, therefore, is automatically updated at NICOND time and if the skip is satisfied, it is updated a second time, pointing PC to the location two beyond the current location.

The PC output is available to the AD for saving a return address in a subroutine call JRST, MUUO, or similar instruction. Generally, the address saved should be for a return to the next instruction, i.e., the instruction that would have been performed had the call or jump not occurred. However, if an instruction is terminated because of a page fault or interrupt, the current address must be saved for a later return to the beginning of the interrupted instruction.

### 2.9.3 Loading PC

New addresses are always supplied to PC via the VMA regardless of the point of origin. The update of the PC or its inhibition is controlled by the microprogram. The following conditions cause PC+1 INH to set, inhibiting the update of PC via VMA AD:

Priority Interrupts – Setting PI Cycle
Console Instruction Execution
Halting the Processor – Halted
Performing the Trap instruction in process table location 421, 422, 423

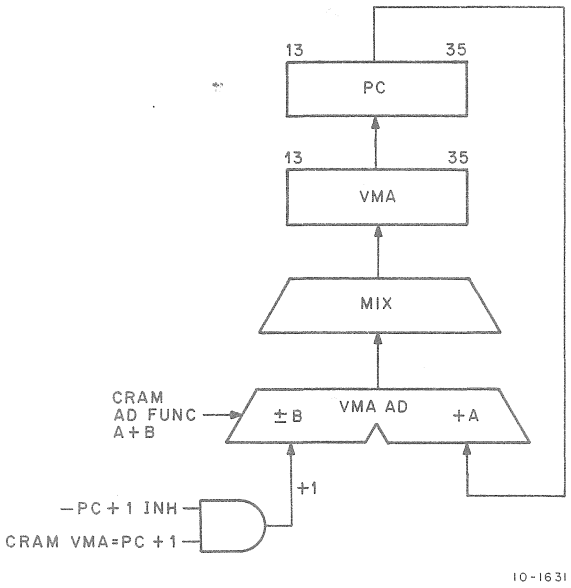



Figure 2-62 Program Count Loop

The PC is loaded at NICOND Dispatch time (Figure 2-63), providing PI CYCLE is clear. In addition, the special field function LOAD PC may also be used to load PC from VMA. During page fault handling, the SPEC/LOAD PC function is used to save the failing virtual address (VMA) in PC while saving the current PC value in ARX. Basically, the MBox builds a page fault status word in its EBus register. The physical page number is stored in bits 14–26 of this word. The EBox page fault handler must replace this address with the virtual page number in VMA 14–26 and then store the updated page fault word in user process table location 500. The operation is as follows:

**Simplified Microprogram Steps Ref PF Handler**

1. ARX ← old PC, PC ← failing VMA
   AR ←EBus Register; PF word

2. BRX←ARX; old PC← ARX AR; PF WORD
   AR← PC; failing VMA

3. At this time, the AR and ARX are Ref PF Handler shifted in such a way as to discard the physical page number and align the proper virtual page number in AR 14–26.

A second case is where SPEC/LOAD PC is used while halting the EBox. In this case, either a Console Halt was issued via the 10-11 interface, or a Halt instruction was performed in either user IOT mode or Kernel mode. The VMA is loaded with the current PC and the PC is loaded with the effective address currently held in VMA. At the time of the halt, the PC value in VMA points to an address one greater than the location containing the Halt and the PC contains E. PC+1 INHIBIT is set to prevent premature incrementation of the jump address now in PC.

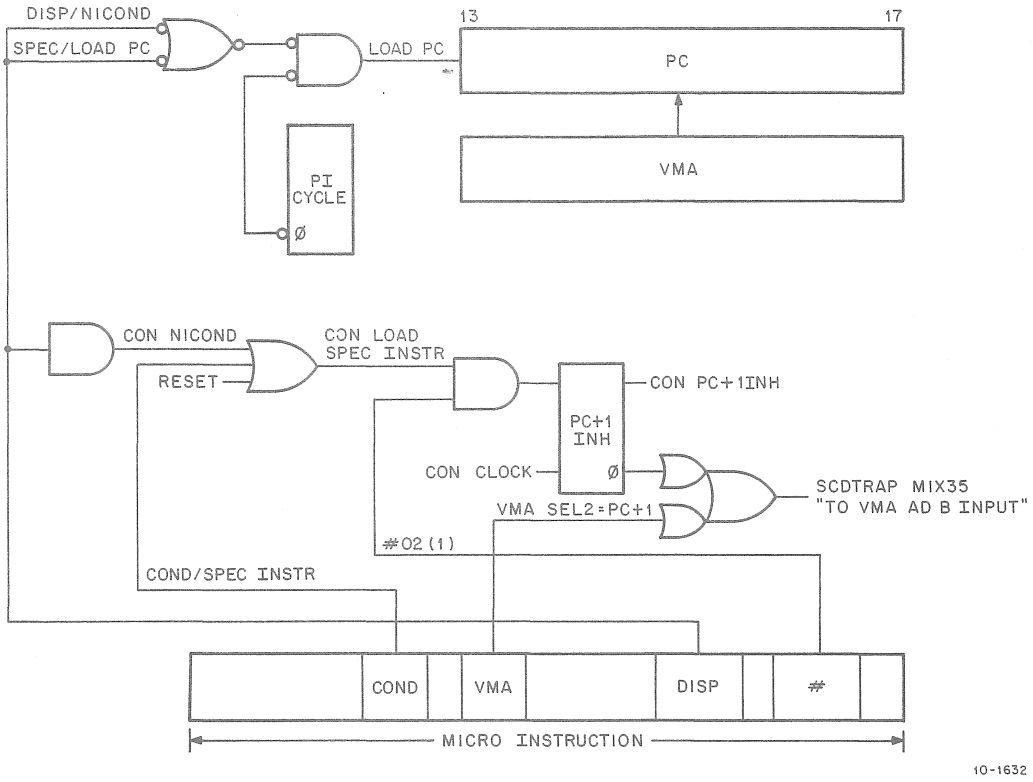



Figure 2-63 PC Loading or Inhibit

### 2.9.4 General Data Path Organization

The data path (Figure 2-60) is divided into four major areas, as listed in Table 2-8.

1. Fast Memory and Fast Memory Address Logic
2. Virtual Memory Address, Program Counter and related logic; 23- and 18-bit logic
3. Arithmetic logic – 36-bit logic
4. Instruction register – 12-bit logic

All of these areas derive control functions from specific fields in the microinstruction.

### 2.9.5 General Data Path Mixer Selection

The microinstruction or microword consists of 75 bits including parity. It is organized into variable length fields that are used to control the data path and control sections of the EBox. In the following pages each field is described functionally in terms of the particular logic with which it is associated.

**2.9.5.1 AD Field** – This field consists of six bits and is used to control the main adder (AD and ADX), that is constructed of type 10181 Arithmetic Logic Units. Table 2-9 lists the ALU functions. The low-order four bits specify one of $16_{10}$ functions. These functions are Boolean or Arithmetic as a function of bit 1 (the mode bit). If bit 1 is a one, the functions are Boolean; if zero, the functions are Arithmetic. Bit 0 is the carry in, when true it adds +1 to any Arithmetic function.

## Table 2-8 Data and Address Path Breakdown

| Major Area | Microfield |
|---|---|
| Fast Memory | FMADR Field<br>COND/FM Write |
| Virtual Memory Addressing | VMA Field<br>COND/VMA ← #<br>+ χ (see Note)<br>COND/VMA DEC<br>COND/VMA INC |
| VMA HELD | COND/LDVMA HELD |
| PC FLAGS (PC LEFT) | COND/AD Flags<br>COND/PCF ← # |
| PC (RIGHT) | SPEC/LOAD PC<br>DISP/NICOND with PI Cycle (0) |
| IR | COND/LOAD IR |
| Shift Count and Auxiliary Arithmetic 10-Bit Logic | SCAD Field<br>SCADA Field<br>SCADB Field<br>SC Field<br>FE Field |
| Arithmetic 36-Bit Logic and 72-Bit Logic<br><br>72-Bit Operations Require SPEC/AD Long | AD Field<br>ADA Field<br>ADB Field<br>AR Field<br>ARX Field<br>BR Field<br>BRX Field<br>MQ Field<br>SH Field<br>ARMM Field |

**NOTE**
**χ is a constant selected by the low-order three bits of the COND code.**

## Table 2-9 ALU Functions

| CIN | M | $S_8$ | $S_4$ | $S_2$ | $S_1$ | BOOLEAN FUNCTION | BOOLEAN CARRIES |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | $\overline{A}$ | A |
| 0 | 1 | 0 | 0 | 0 | 1 | $\overline{A}V\overline{B}$ | A+(A$\overline{B}$) |
| 0 | 1 | 0 | 0 | 1 | 0 | $\overline{A}$VB | A+(AB) |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 2*A |
| 0 | 1 | 0 | 1 | 0 | 0 | $\overline{A}\overline{B}$ | AVB |
| 0 | 1 | 0 | 1 | 0 | 1 | $\overline{B}$ | (A$\overline{B}$) + (AVB) |
| 0 | 1 | 0 | 1 | 1 | 0 | EQV | A + B |
| 0 | 1 | 0 | 1 | 1 | 1 | AV$\overline{B}$ | A+(AVB) |
| 0 | 1 | 1 | 0 | 0 | 0 | $\overline{A}$B | AV$\overline{B}$ |
| 0 | 1 | 1 | 0 | 0 | 1 | XOR | A-B-1 |
| 0 | 1 | 1 | 0 | 1 | 0 | B | (AV$\overline{B}$) + (AB) |
| 0 | 1 | 1 | 0 | 1 | 1 | AVB | A + (AV$\overline{B}$) |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | -1 |
| 0 | 1 | 1 | 1 | 0 | 1 | A$\overline{B}$ | A$\overline{B}$-1 |
| 0 | 1 | 1 | 1 | 1 | 0 | AB | AB-1 |
| 0 | 1 | 1 | 1 | 1 | 1 | A | A-1 |

| CIN | M | $S_8$ | $S_4$ | $S_2$ | $S_1$ | ARITHMETIC FUNCTION | ARITHMETIC CARRIES |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | A | A |
| 0 | 0 | 0 | 0 | 0 | 1 | A + (A$\overline{B}$) | A + (A$\overline{B}$) |
| 0 | 0 | 0 | 0 | 1 | 0 | A + (AB) | A + (AB) |
| 0 | 0 | 0 | 0 | 1 | 1 | 2 * A | 2 * A |
| 0 | 0 | 0 | 1 | 0 | 0 | AVB | AVB |
| 0 | 0 | 0 | 1 | 0 | 1 | (A$\overline{B}$) + (AVB) | (A$\overline{B}$) + (AVB) |
| 0 | 0 | 0 | 1 | 1 | 0 | A + B | A + B |
| 0 | 0 | 0 | 1 | 1 | 1 | A + (AVB) | A + (AVB) |
| 0 | 0 | 1 | 0 | 0 | 0 | AV$\overline{B}$ | AV$\overline{B}$ |
| 0 | 0 | 1 | 0 | 0 | 1 | A-B-1 | A-B-1 |
| 0 | 0 | 1 | 0 | 1 | 0 | (AV$\overline{B}$) + (AB) | (AV$\overline{B}$) + (AB) |
| 0 | 0 | 1 | 0 | 1 | 1 | A + (AV$\overline{B}$) | A + (AV$\overline{B}$) |
| 0 | 0 | 1 | 1 | 0 | 0 | -1 | -1 |
| 0 | 0 | 1 | 1 | 0 | 1 | A$\overline{B}$ -1 | A$\overline{B}$ -1 |
| 0 | 0 | 1 | 1 | 1 | 0 | AB -1 | AB -1 |
| 0 | 0 | 1 | 1 | 1 | 1 | A -1 | A -1 |

NOTE: If CIN is true, add +1 to the given arithmetic function. Carry out is true if the adder, extended left, would need carry in to generate the correct function.
Carry Out is not affected by the mode (i.e., BOOLEAN FUNCTIONS give the same carry as the ARITHMETIC FUNCTIONS).

For Boolean functions, the carry in can cause a carry out if the corresponding Arithmetic function for the same S-bits would have produced a carry given the appropriate inputs. For example, assume the AD function to be performed is A and the A input equals 777777,777777. The Boolean function A performs the 1s complement of the A input, which yields a result of 000000,000000. The corresponding Arithmetic function is A and thus, if carry is true, this yields A + 1. Using the existing A input 777777,777777 +1 gives a sum of 000000,000000 and a carry. If the Boolean function A is given and carry in is true, assuming the same A input as above, the function out is 000000,000000 and a carry is generated.

The 10181 may be thought of as concurrently performing the Arithmetic operation specified and the Boolean operation specified; the sum, however, is not affected when the Boolean functions are implemented, yet the state of Carry Generate and Carry Propagate will reflect the Arithmetic result that would have formed the sum.

**MC10181 Arithmetic Logic Unit Description**
Figure 2-64 is an overview diagram of the ALU logic. Table 2-10 lists the ALU functions, with carry.

$$GEN = \overline{A}(S_4 B + S_8 B)$$

$$PROP = A + S_1 B + S_2 B$$

Signals GEN and PROP are used in each digit to generate the output signal Fn. In the logic mode, carries are inhibited on the output stage, and the logic function F is given by

F GEN V PROP (XOR)

(The output function is the Exclusive-Or of the two internal signals GEN and PROP).

When adding two numbers, in the absence of a CARRY IN, the Exclusive-Or function is the function required. A CARRY IN signal always complements this in this circuitry by controlling the final Exclusive-Or on the output stage.

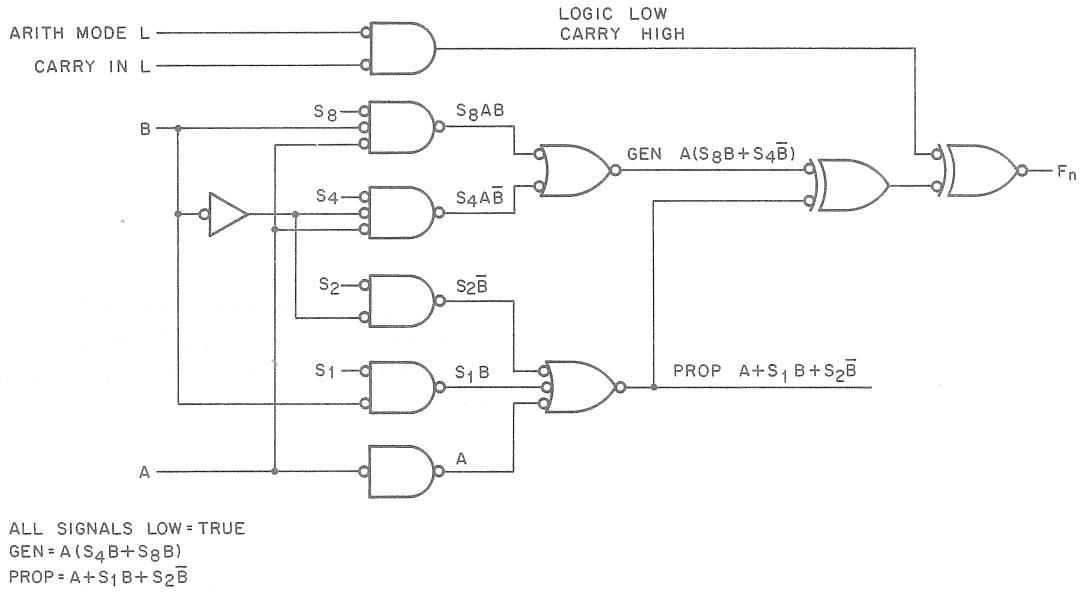



Figure 2-64 ALU Overview

## Table 2-10 ALU Functions With Carry

| Code | | | | GEN | PROP | Logic Fn | Arithmetic | |
|---|---|---|---|---|---|---|---|---|
| $S_8$ | $S_4$ | $S_2$ | $S_1$ | | | | CARRY LOW | CARRY HIGH |
| 0 | 0 | 0 | 0 | A | 0 | $\bar{A}$ | A | A + 1 |
| 0 | 0 | 0 | 1 | A | $A\bar{B}$ | $\bar{A}\vee\bar{B}$ | $A+A\bar{B}$ | $A+A\bar{B}+1$ |
| 0 | 0 | 1 | 0 | A | AB | $\bar{A}\vee B$ | A+AB | A+AB+1 |
| 0 | 0 | 1 | 1 | A | A | 1 | 2*A | 2*A+1 |
| 0 | 1 | 0 | 0 | A∨B | 0 | $\bar{A}\,\bar{B}$ | A∨B | A∨B+1 |
| 0 | 1 | 0 | 1 | A∨B | $A\bar{B}$ | $\bar{B}$ | $A\bar{B}+(A\vee B)$ | $A\bar{B}+(A\vee B)+1$ |
| 0 | 1 | 1 | 0 | A∨B | AB | EQV | A+B | A+B+1 |
| 0 | 1 | 1 | 1 | A∨B | A | $A\vee\bar{B}$ | $A+(A\vee B)$ | $A+(A\vee B)+1$ |
| 1 | 0 | 0 | 0 | $A\vee\bar{B}$ | 0 | $\bar{A}B$ | $A\vee\bar{B}$ | $A\vee\bar{B}+1$ |
| 1 | 0 | 0 | 1 | $A\vee\bar{B}$ | $A\bar{B}$ | A∨B | A B 1 | A B |
| 1 | 0 | 1 | 0 | $A\vee\bar{B}$ | AB | B | $AB+(A\vee\bar{B})$ | $AB+(A\vee\bar{B})+1$ |
| 1 | 0 | 1 | 1 | $A\vee\bar{B}$ | A | A∨B | $A+(A\vee\bar{B})$ | $A+(A\vee\bar{B})+1$ |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | $A\bar{B}$ | $A\bar{B}$ | $A\bar{B}$ 1 | $A\bar{B}$ |
| 1 | 1 | 1 | 0 | 1 | AB | AB | AB 1 | AB |
| 1 | 1 | 1 | 1 | 1 | A | A | A 1 | A |

NOTE
All signals high true except GEN and PROP.

The MC10181 carries out an addition by converting the two numbers at A and B to two alternative signals GEN and PROP, given by

$$\begin{aligned} GEN &= AB \quad (S_8 = 1, S_4 = 0) \\ PROP &= A+B \quad (S_1 = 1, S_2 = 0) \end{aligned}$$

For example:

| | | | | |
|---|---|---|---|---|
| | A = | 0011 | 3 | |
| | B = | 0101 | 5 | |
| then | AB = | 0001 | 1 | (GEN) |
| | A+B = | 0111 | 7 | (PROP) |
| | SUM = | 1000 | 8 | |

Adding any two numbers A and B is equivalent to adding the two functions AB and A+B. However, the advantages of the second part are that one (AB) shows when carries should be generated, while the other (A+B) shows when carries should be propagated. The final sum is the XOR of the two numbers (AB and A+B), complemented by the CARRY IN signal.

$$\begin{aligned} GEN &= A(S_8B + S_4B) \\ PROP &= A + S_1 + S_2B \end{aligned}$$

These two equations show that PROP is generated whenever A is true, which is a requirement for GEN to be true, i.e., GEN implies PROP, and thus whenever GEN is a one, PROP is also a one, and thus GEN plus PROP must generate a carry.

GEN is sufficient indication of carry generation. Similarly, PROP is sufficient indication of carry propagate.

**High Logic**

Actually, the circuit was designed to promote understanding for low logic, and the descriptions and tables given in the literature are far clearer for this case.

Although the circuit does give the correct answers for high logic, the circuit does operate on the low signals. Thus, an addition can be considered as an addition of the zeros, with carry generated from the addition of two zeros, and propagated, as before, by the XOR of the two numbers.

| | | | | |
|---|---|---|---|---|
| A | = | 0 0 1 1 0 | | |
| B | = | 0 1 0 1 0 | | |
| | | 1 0 0 1 1 | | XOR |
| | | 1 0 0 0 1 | | GEN |
| | | 1 1 1 0 1 | | PROP |
| COUT ← | | 1 0 0 0 0 | ← | Cin (low) |
| COUT ← | | 1 0 0 0 1 | ← | Carry (high) |

The correct answer, therefore, occurs when Cin is asserted to the least significant bit. This can be viewed in two ways:

1. Carry is asserted high. In this case, the function considered above is Fn = A plus B and carry input adds a one. This is simple, but GEN and PROP meanings become obscure (especially when passed through the LOOK-AHEAD CARRY block).

   Generate = > (G = High and P = High)
   Propagate = > (G = High)

2. Carry is asserted low. In this case, the above function is Fn - A plus B plus 1, and the carry input subtracts a one, but hardware is simple to follow:

   Generate = > (G = Low)
   Propagate = > (P = Low)

To functionally describe the use of the various Boolean and Arithmetic functions, it is first necessary to define two other microinstruction fields which are used to enable various data to the AD A and B inputs. The first field is ADA, a 3-bit field. ADA can select the inputs shown in Figure 2-65.

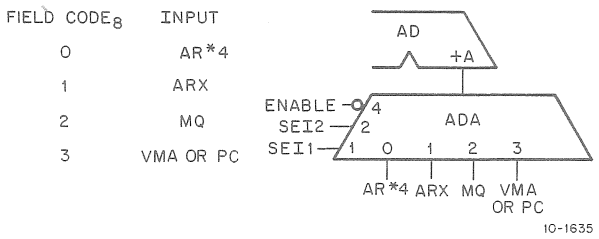
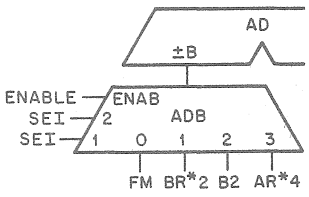



Figure 2-65 ADA Example

The second field is ADB, a 2-bit field. ADB can select the inputs shown in Figure 2-66.

| FIELD CODE$_8$ | INPUT |
|---|---|
| 0 | FAST MEMORY |
| 1 | BR X 2 |
| 2 | BR |
| 3 | AR X 4 |

*ADR also controls ADXB, See ADB Field.




10-1634

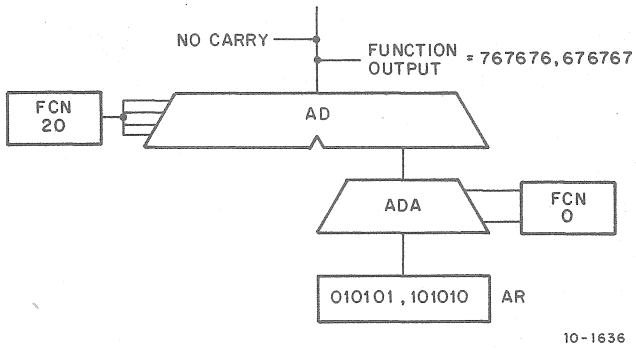
Figure 2-66 ADB Example

The following examples illustrate various operations that might be performed using EBox registers and the ADA or/and ADB input mixers. No guarantee is made that the operations illustrated are used in the microcode.

Example: $\overline{A}$ – Function 20
Initial Conditions: AR = 010101, 101010
ADA Field Function = 0

The function $\overline{A}$ performs the 1s complement of the data in AR (Figure 2-67). The AD function output is 767676,676767. Note that at this time the Carry In is false. No carries are generated in this example because the corresponding carries function is A (Table 2-9).

Example: $\overline{A}\overline{B}$ – Function 24
Initial Conditions: ARX = 777777,777777
FM = 777777,777776
ADA Field = 2
ADB Field = 0




10-1636

Figure 2-67 Function $\overline{A}$

The Boolean function $\overline{A}\overline{B}$ performs the logical AND of the complement of A with the complement of B (Figure 2-68). The value in ARX is selected on the ADA input mixer (777777,777777) and the value in some addressed fast memory location is selected on the ADB input mixer (777777,777776). The result presented to the function output is 000000,000000. Referring to Table 2-9, the corresponding Boolean carries function is A v B; carries are generated for the given values of A and B. For any values of A and B, no carries are generated.

Example: AB – Function 36
Initial Conditions: AR 000000,100001
BR 000765,100070
ADA Field 0
ADB Field 2

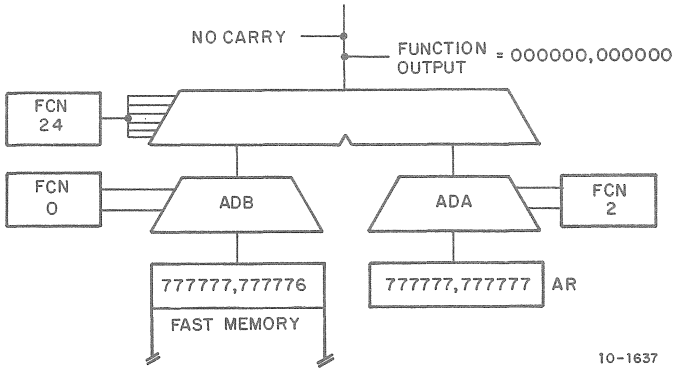



Figure 2-68 Function $\overline{AB}$

The Boolean function AB performs the logical AND of A and B (Figure 2-69). The value in AR (000000,100001) is ANDed with the value in BR (000765,100070) and the result presented to the function output is 000000,100000. Referring to Table 2-9, the corresponding carries function is AB – 1 and, given the existing inputs, it can be demonstrated that a carry from the most significant bit results if the AND of any two values results in a nonzero sum. The following demonstrates this:

```
    000000, 100001
∧   000765, 100070
    ------------------
    000000100000
+   77777 7,777777
1 ← 00000 077777
```

AB Example: A – Function 37
Initial Conditions: ARX = 000000,000100
ADA Field = 2

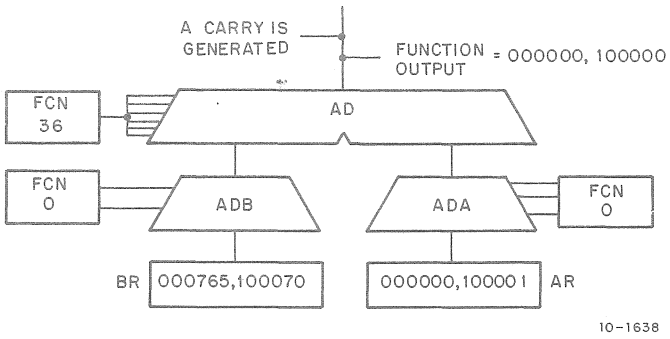



Figure 2-69 Function AB

The Boolean function A produces (at the function output) the value at the ADA input (Figure 2-70). In this example, the result is 000000,000100, but notice that the corresponding carries function is A – 1. Subtracting 1 from 000000,000100 is equivalent to adding –1, which is 777777,777777 in 2's complement notation. The result gives a carry out of the most significant bit of the AD (CRY 0). Thus, although the sum represents the ADA input 000000,000100, a carry is generated.

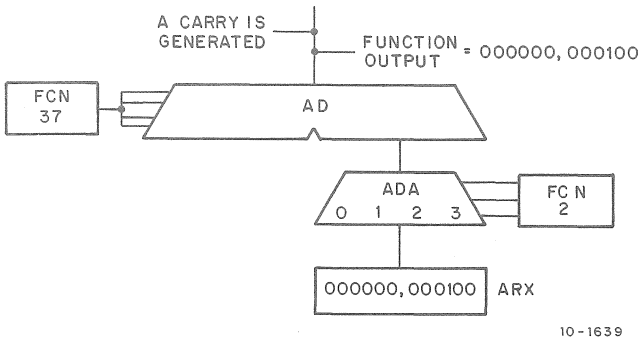



Figure 2-70 Function A

**2.9.5.2 ADA Field** – This field consists of three bits and is used with the main ADDER. Referring to Table 2-11, the low-order two bits select AR(0), ARX(1), MQ(2), and VMA HELD or PC(3). The high-order bit is used as a disable. This bit also controls ADXA. When the high-order bit of the ADA field is zero, ADXA selects ARX and when it is one, it selects zeros.

**2.9.5.3 ADB Field** – This field consists of two bits and is used in a similar fashion to that of ADA in conjunction with the main ADDER. Referring to Table 2-12, the selection is as follows: FM(0), BR*2(1), BR(2), and AR*4(3).

**Table 2-11 ADA, ADXA Selection**

| CRAM | ADA Source | ADXA Source |
|---|---|---|
| 0 | AR | ARX |
| 1 | ARX | ARX |
| 2 | MQ | ARX |
| 3 | PC | ARX |
| 4–7 | 0s | 0s |

**Table 2-12 ADB, ADXB Selection**

| CRAM ADB | ADB Source | ADXB Source |
|---|---|---|
| 0 | FM | (unused) |
| 1 | BR*2 | BRX*2 |
| 2 | BR | BRX/2 |
| 3 | ARX*4 | ARX*4 |

In addition, ADB directly controls ADXB utilizing the same 2-bit field. Here the selection is unused (0), BRX*2(1), BRX/2(2) and ARX*4(3). Although AD and ADX together with ADA, ADXA, ADB, and ADXB normally function concurrently, information in ADX does not affect AD unless so specified. Carries from ADX must be specifically enabled to AD in order to affect its sum.

**2.9.5.4 AR Field** – This field consists of three bits. Figure 2-71 details the breakdown of various combinations of CRAM AR Selection and hardware controlled selection. Generally, the CRAM AR field specifies selection as follows: ARMM(0), CACHE(1), AD(2), EBUS(3), SH(4), ADX*2(5), ADX(6) and ADX/4(7).

AR register loading is controlled by either the hardware or microcode. Normally, the AR register recirculates its contents. Selecting any of the AR select lines CRAM ARM SEL 4, 2, or 1 enables loading AR. The selection of none of the CRAM ARM SEL lines enables the AR mixer to select ARMM. The loading of AR is then a microcode function.

During reads from core, the signal CLK RESPONSE MBOX, selects ARM SEL 1 to enable the cache data lines into AR. Similarly, on reads from fast memory via AD, FM XFER selects ARM SEL 2 to enable the AD into AR. Various combinations of clearing of AR are possible depending on the conditions. This information is given in table form on Figure 2-71.

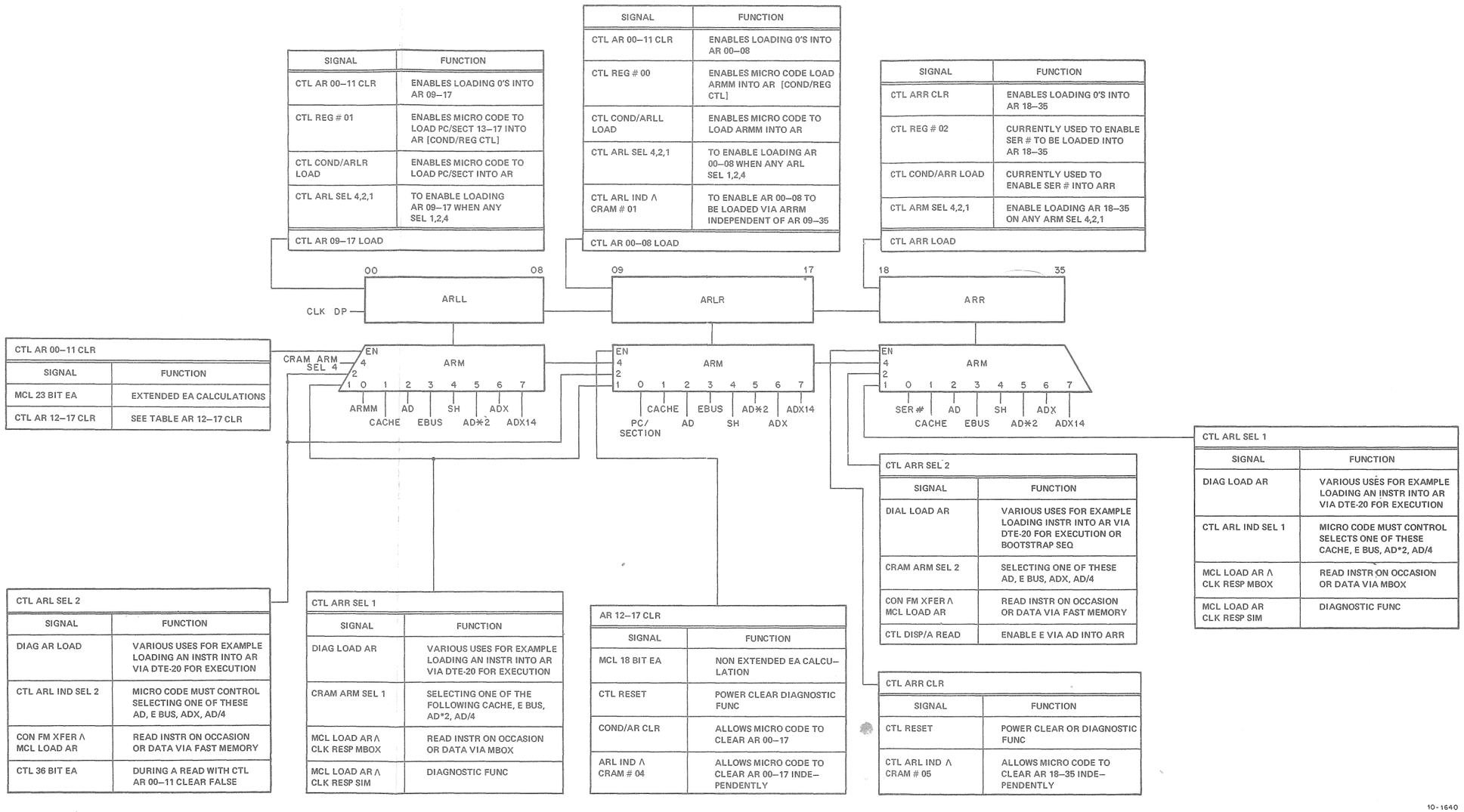
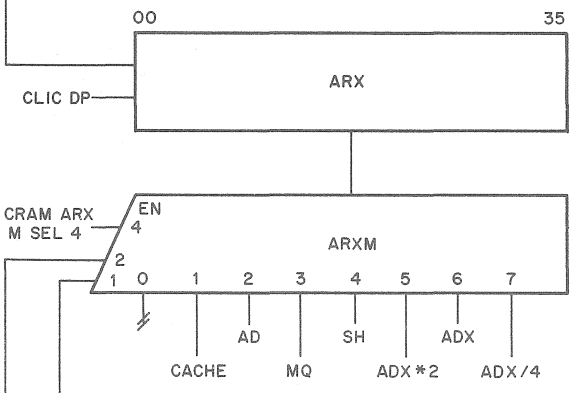



Figure 2-71 AR Selection

**2.9.5.5 ARX Field** – This field consists of three bits. Figure 2-72 details the breakdown of various combinations of CRAM ARX selection and hardware controlled selection. Generally, the CRAM ARX field specifies selection as follows: UNUSED(0), CACHE(1), AD(2), MQ(3), SH(4), AD*2(5), ADX(6), and ADX/4(7). ARX register loading is controlled by either the hardware or microcode. Normally, the ARX register recirculates its contents. Selecting any of the ARX select lines CRAM ARXM SEL 4, 2, or 1 enables loading ARX. The selection of none of these lines currently defaults to an unused input (0). As with AR, during reads from core, CLK RESPONSE MBOX, selects ARXM SEL 1, to enable the cache data lines into ARX. Similarly, on reads from fast memory via AD, FM XFER selects ARXM SEL 2 to enable the AD into ARX. Generally, the ARX is cleared via ARL IND and number 03. The various combinations are shown on Figure 2-72 in table form.

| SIGNAL | FUNCTION |
|---|---|
| CRAM ARXM SEL 4 | SELECTING ONE OF THESE, SH, AD*2, ADX, AD/4 |
| CTL ARX SEL 2 | SEE TABLE BELOW CTL ARXR SEL 2 |
| CTL ARX SEL 1 | SEE TABLE BELOW CTL ARXR SEL 1 |
| CTL RESET | POWER CLEAR OR DIAG FUNC |
| CTL ARX CLR | SEE TABLE CTL ARX CLR |
| CTL ARX LOAD | |

| CTL ARX CLR | |
|---|---|
| SIGNAL | FUNCTION |
| CTL ARL IND ^ CRAM # 03 | TO CLEAR ARX WHILE OPERATING ON AR |




| CTL ARX SEL 2 | |
|---|---|
| SIGNAL | FUNCTION |
| CRAM ARX M SEL 2 | SELECTING ONE OF THESE AD, MQ, ADX, ADX/4 |
| CONFM XFER ^ MCL LOAD ARX | READ OF INSTR, INDIRECT WORD OR DATA VIA FAST MEMORY |

| CTL ARX SEL 1 | |
|---|---|
| SIGNAL | FUNCTION |
| MCL LOAD ARX ^ CLK RESP MBOX | READ OF INSTR, INDIRECT WORD OR DATA VIA MBOX |
| MCL LOAD ARX ^ | DIAGNOSTIC FUNC |
| CRAM ARXM SEL 1 | SELECTING ONE OF THESE, CACHE, MQ, ADX*2, ADX/4 |

10-1641

Figure 2-72 ARX Selection

**2.9.5.6 BR Field** – The BR field consists of one bit and is used to select one of two possible sources as input to the Buffer Register (BR). The following sources may be selected: BR(0), AR(1).

**2.9.5.7 BRX Field** – The BRX field consists of one bit and is used to select one of two possible sources as input to the Buffer Register Extension (BRX). The following sources may be selected: BRX(0), ARX(1).

**2.9.5.8 FMADR Field** – The FMADR field consists of three bits and is used in the selection of source addresses for fast memory. Basic selection is as follows:

1. AC0(0), (IRAC 9–12),
2. AC1(1), (IRAC 9–12)+1 Modulo 16,
3. XR(2), (ARX 14–17),
4. VMA(3), VMA 32–35,
5. AC2(4), (IRAC 9–12)+2 Modulo 16,
6. AC3(5), (IRAC 9+2)+3 Modulo 16,
7. CB#(6) current ac block and selection within it is via # field,
8. #B#(7), this is some block selected by # field.

**2.9.5.9 SCAD Field** – The SCAD field consists of three bits and is used to control the Shift Counter Adder (SCAD) during various microinstruction operations. It is wired to implement eight functions as illustrated in Table 2-13. The input mixer structure is similar to that for the AD or ADX in that there are two input mixers labeled SCADA and SCADB. These mixers are selected via two control RAM fields labeled SCADA and SCADB.

**Table 2-13 SCAD Field**

| CRAM SCAD | | | SCAD Function | Function Breakdown | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | | M | S8 | S4 | S2 | S1 | IN |
| 0 | 0 | 0 | A | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | A-B-1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | A+B | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | A-1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | A+1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | A-B | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | A or B | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | A and B | 0 | 1 | 1 | 1 | 0 | 1 |

**2.9.5.10 SCADA Field** – The SCADA field consists of three bits and is used to select various sources as input to the SCADA Input. The following sources may be selected: FE(0), AR POS(1), AR EXP(2), #(3). SCADA selections of 4–7 disable SCADA producing zeros as output.

The floating-point exponent register (FE) is a 10-bit register. The AR position field is used in byte instructions and consists of AR 00–05. The AR exponent field consists of AR bits 00–08 and the magic number field is a 9-bit control RAM field used to implement various operations. The SCADA mixer selection is shown in Table 2-14.

**Table 2-14 SCADA Mixer Selection**

| CRAM SCADA | Source |
|---|---|
| 0 | FE |
| 1 | AR0–5 |
| 2 | AR EXP |
| 3 | # |
| 4–7 | 0s |

**2.9.5.11 SCADB Field** – The SCADB field is a 2-bit field used to select various sources as input to the SCAD ±B input. The following sources may be selected in the SCADB mixer: SC(0), AR SIZE(1), AR00–08(2), and #(3). Selection of 4–7 disables SCADB, producing zeros as output. The SCADB mixer selection is shown in Table 2-15.

**Table 2-15 SCADB Mixer Selection**

| CRAM SCADB | Source |
|---|---|
| 0 | SC |
| 1 | AR 6–11 |
| 2 | AR 00–08 |
| 3 | # |
| 4–7 | 0s |

The shift counter (SC) is a general-purpose 10-bit register used in shift counting operations such as performed in floating-point instruction and shift instruction execution. It also controls the shifter when the SH-ARMM field is zero (SH AR and ARX). The AR SIZE field is used in byte instructions and consists of AR bits 06–11. The AR00–08 is used in string and edit functions. The magic number field is a 9-bit general-purpose CRAM field used for various functions.

**2.9.5.12 SC Field** – The SC field consists of one bit and is used with the special field function SCM alternate. With SC and SCM alternate, four possible sources may be selected as follows:

With the special field function SCM ALT and SC field equal to zero, FE is selected. Similarly, with SCM ALT and SC field equal to one, AR SHIFT is selected. AR SHIFT consists of bits 18 and 28–35 of AR, which are derived from the effective address for shift instructions. If bit 18 is set, the shift specified is a right shift; otherwise, it is a left shift.

**2.9.5.13 SH Field** – The SHIFTER field consists of two bits and is used to select four possible inputs to the shifter. The selection is as follows: the combined AR, ARX(0), AR(1), ARX(2), and AR SWAPPED(3). When shifting AR, ARX left (which is the only way SH shifts physically), SC can specify up to $35_{10}$ shifts. Any number less than 0 or greater than $35_{10}$ selects ARX as output.

**2.9.5.14 The AR Mixer Mixer (ARMM)** – The AR Mixer Mixer (ARMM) field consists of two bits and is used with other control signals and the absence of ARM SEL 4, 2, and 1 to select various sources as input to AR mixer.

The ARMM comprises three parts: bits 00–08, bit 12, and bits 13–17. The same field that controls SH controls ARMM00–08. The following may be selected as input to ARMM00–08: #(0), AR SIGN SMEAR(1), SCAD EXP(2), and SCAD POS(3). AR SIGN SMEAR is AR0–8 from AR0. SCAD EXP is AR0–8 via SCAD, and SCAD POS is AR0–5 via SCAD.

ARMM bit 12 is controlled by CRAM SH-ARMM SEL 1 when transferring the previous section to AR for certain operations. ARMM bits 13–17 are also under control of CRAM SH-ARMM SEL 1 but the signal is actually MCL PREV SECT to ARMM. The default value for ARMM 13–17 is PC 13–17 and the selected value is VMA previous section 13–17.

**2.9.5.15 VMA Field** – The VMA field consists of two bits and is used to select various sources as input to VMA. The following are specified by the CRAM field VMA(0), PC(1), PC+1(2), and AD(3). Address control is presented in Subsection 2.4 and a path diagram is provided to show various combinations in Figure 2-58.

**2.9.5.16 MQ Field** – The MQ field consists of one bit and is used in combination with the following:

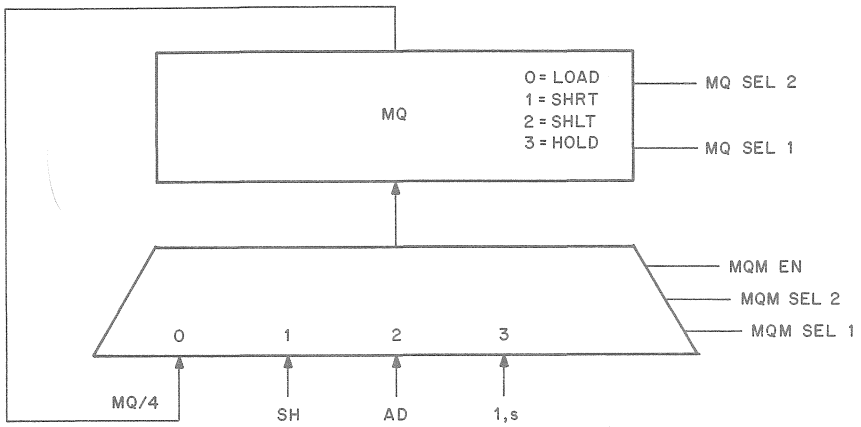
DISP/MUL
DISP/DIV
SPEC/MQ SHIFT
SPEC/REG CONTROL
MAGIC NUMBER FIELD

Refer to Figure 2-73 for various combinations.

## 2.10 EBOX INSTRUCTION SET FUNCTIONAL OVERVIEW

Figure 2-74 breaks down the KL10 instruction set into several functional areas. These areas are related to the minor machine cycles and to the microcode dispatch RAM decoding. The figure shows seven basic areas as follows:

| | | |
|---|---|---|
| 1. | Group | Class of instruction |
| 2. | Address Calculation | xr, @, B, Y |
| 3. | Data Fetch | IMM, Read, Read-Write, Write, Read, Pse Write |
| 4. | Execution | 36-Bit Data Path (DP), 18-Bit Address Path (AP), 23-Bit AP, 10-Bit AP |
| 5. | Special Conditions | Can cause PI, Trap |
| 6. | Store Data | Write |
| 7. | Interruptable | |




| MQM Out | MQM EN | MQM Sel 2 | MQM Sel 1 |
|---|---|---|---|
| MQ/4 | 1 | 0 | 0 |
| SH | 1 | 0 | 1 |
| AD | 1 | 1 | 0 |
| 1's | 1 | 1 | 1 |

| MQ← | MQ Sel 2 | MQ Sel 1 |
|---|---|---|
| MQM | 0 | 0 |
| MQM/2 | 0 | 1 |
| MQM*2 | 1 | 0 |
| Hold | 1 | 1 |

| CRAM MQ Field | SELECTED CONTROL SIGNALS | | | | | CONTROLLING FIELDS | | | COND/REG CTL |
|---|---|---|---|---|---|---|---|---|---|
| CRAM MQ | MQM EN | MQM Sel 2 | MQM Sel 1 | MQ Sel 2 | MQ Sel 1 | SPEC/MQ SHFT | DISP/ DIV | DISP/ MUL | #07–08 |
| 0 | 0 | 0 | 0 | 1* | 1* | 0 | 0 | 0 | 00* |
| 0 | 0 | 0 | 0 | 1* | 0* | 0 | 1* | 0 | 0X* |
| 0 | 0 | 0 | 0 | 1* | 0* | 1* | 0 | 0 | 0X* |
| 1 | 1 | 1* | 0* | 0 | 0 | | | | 11* |
| 1 | 1 | 0* | 1* | 0 | 0 | 0* | 0* | 0* | 00* |
| 1 | 1 | 0 | 0* | 0 | 0 | 0 | 0 | 1* | 00 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 01 |
| 1 | 1 | 1* | 1* | 0 | 0 | 0 | 0 | 0 | 10* |
| 1 | 1 | 0* | 0 | 0 | 0 | 0 | 0 | 0 | 11* |
| Reset | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

10-1642

Figure 2-73 MQ Selection

[1] INTERRUPTABLE *SPECIAL ADDRESS MANIPULATIONS

| | GROUP | ADDRESS CALCULATION | | | | DATA FETCH | | | | | EXECUTION | | | | SPECIAL CONDS | | STORE DATA | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODES | CLASS | XR | @[1] | B | Y | IMM | READ | READ–WRITE | WRITE | READ PSE WRITE | 36 BIT DP | 18 BIT AP | 23 BIT AP | 10 BIT DP | CAN CAUSE PI | TRAP | WRITE | INTERRUPTABLE |
| 200–217 | MOVE GROUP | YES | YES | NO | YES | IMM | BASIC | | MEM | SELF | ALL | | | | NO | NO | IMM TO FM<br>BASIC TO FM<br>MEM TO E<br>SELF TO E AND FM | |
| 500–577 | HALF WORD GROUP | YES | YES | NO | YES | IMM | BASIC | | MEM | SELF | ALL | | | | NO | NO | SAME AS FULL WORD GROUP | |
| 120–125 | DOUBLE WORD FULL WORD GROUP | YES | YES | NO | YES | | BASIC | | MEM | | ALL | | | | NO | NO | BASIC TO FM, FM+1<br>MEM TO E, E+1 | |
| 400–477 | BOOLEAN GROUP | YES | YES | NO | YES | IMM | BASIC | SET MB | MEM | BOTH | ALL | | | | NO | NO | SAME AS FULL WORD GROUP | |
| 260–263 | STACK GROUP | YES | YES | NO | YES | IMM | READ | | | | ALL | ALL | | | NO | YES [PDOVL] | IMM TO FM ALSO CAUSES A FETCH FROM FM | |
| 104–105 | JSYS AND ADJSP | YES | YES | NO | YES | IMM | | | | | ALL | JSYS | | | NO | YES | IMM TO FM | |
| 600–677 | TEST GROUP | YES | YES | NO | YES | IMM | BASIC | | | | ALL | | | | NO | NO | IMM TO FM<br>BASIC TO FM | |
| 330–337<br>350–357<br>370–377 | ARITHMETIC SKIPS | YES | YES | NO | YES | | SKIPXX | | | SO SXX<br>AO SXX | ALL | | | | NO | NO | SKIPXX: IFA ≠ STORE (E); INAC | |
| 300–317 | COMPARES | YES | YES | NO | YES | CAIXX | CAMXX | | | | ALL | CONDITIONAL ALL | | | NO | NO | CAIXX STORES NOTHING<br>CAMXX STORES NOTHING | |
| 320–367<br>252–253 | CONDITIONAL JUMPS | YES | YES | NO | YES | IMM | | | | | ALL | CONDITIONAL ALL | | | NO | AOJX<br>SOJX [AROV] | JUMPX STORES NOTHING<br>AOJX TO FM<br>SOJX TO FM | |
| 252–253 | ARITHMETIC TESTING | YES | YES | NO | YES | IMM | | | | | ALL | CONDITIONAL ALL | | | NO | NO | ALL TO FM | |
| 264–267 | SUBROUTINE CALL | YES | YES | NO | YES | IMM | | | | | ALL | UNCONDITIONAL ALL | | | NO | NO | JSR TO E<br>JSP TO FM<br>JSA TO E AND FM<br>JRA TO AC } ALL CAUSE FETCH | |
| 254–255 | AC DECODED JUMPS | YES | YES | NO | YES | IMM | | | | | ALL | JRSTS ARE UNCONDITIONAL | | | NO | NO | HIGHER LEVEL FUNCTIONS PERFORMED | |
| 256 | XCT* | YES | YES | NO | YES | IMM | | | | | ALL | UNCONDITIONAL | | | NO | NO | FETCH<br>IN KERNAL MODE PXCT | |
| 257 | MAP | YES | YES | NO | YES | IMM | | | | | ALL | UNCONDITIONAL | | | NO | NO | PAGING INFO TO FM | |
| 270–277 | FIXED POINT ARITH | YES | YES | NO | YES | ADDI<br>SUBI | ADD<br>SUB | | ADDM<br>SUBM | ADDB<br>SUBB | ALL | | YES | | NO | ALL [AROV] | SAME AS FULL WORD | |
| 220–227<br>230–237 | FIXED POINT ARITH | YES | YES | NO | YES | IMUL<br>IDIV | | XMULM, XMULB<br>XDIVM, XDIVB | | | ALL | | | ALL | NO | ALL [AROV] | IMULI, IDIVI TO FM<br>IMUL, IDIV TO FM<br>IMULM, IDIVM TO MEM<br>IMULB, IDIVB TO FM, MEM<br>MULI, DIVM TO FM, FM+1<br>MUL, DIV TO FM, FM+1<br>MULM, DIVM TO E, FM+1<br>MULB, DIVB TO E, FM+1 | |
| 114–117 | DOUBLE INTEGER | YES | YES | NO | YES | | BASIC | | | | ALL | | | ALL | NO | ALL [AROV] | D ADD TO FM, FM+1<br>D SUB TO FM, FM+1<br>D MUL TO AC, AC+1, +2, +3<br>D DIV TO E, E+1 | |
| 140–147<br>150–157<br>160–167<br>170–177 | SINGLE PREC FLOATING POINT | YES | YES | NO | YES | IMM | BASIC | MEM<br>BOTH | | | ALL | | | ALL | NO | ALL [AROV] | | |
| 130–132<br>122<br>126–127 | UFA, DFN, FSC<br>FIX, FIXR<br>FLTR | YES | YES | NO | YES | FSC | UFA, FIX<br>FIXR<br>FLT, FLTR | | | DFN | | | | | | ALL [AROV] | | |
| 000–103 | UUO'S | YES | YES | NO | YES | IMM | | | | | ALL | | | ALL | NO | NO | HIGHER LEVEL FUNCTIONS | |
| 134–137 | BYTE GROUP* | YES | YES | NO | YES | | AC≠0<br>ADJBP<br>AC=0<br>IBP | ILDB<br>IDPB<br>[FPD] | | | ALL | | | ALL | NO | NO | IBP, UPDATE POINTER (E)<br>ILDB, UPDATE POINTER (E)<br>BYTE → FM<br>IDPB, UPDATE POINTER (E) BYTE ← AC | YES<br>IN @<br>LOOP |
| 240–247<br>(NOT 243) | SHIFTS AND ROTATES | YES | YES | NO | YES | IMM | | | | | ALL | | | ALL | NO | NO | LSH, ASH: AC → FM<br>LSHC, ASHC: AC+1 → FM+1<br>ROT: AC → FM<br>ROTC: AC+1 → FM+1 | |
| 251 | BLT* | YES | YES | NO | YES | IMM | | | | | YES | YES | NO | YES | NO | NO | MULTIPLE WORDS MOVED SOURCE+N TO DEST+N | YES |
| 700–777 | INPUT OUTPUT | YES | YES | NO | YES | IMM | BASIC | BLKX | MEM | | ALL | | | ALL | YES [CONO PI] | NO | E/E INTERFACE OPERATIONS | |
| 250 | EXCH | YES | YES | NO | YES | | | | | EXCH | ALL | | | NO | NO | NO | E ⇄ FM | |
| 110–113 | DOUBLE PREC FLOATING POINT | YES | YES | NO | YES | | ALL | | | | ALL | | | ALL | NO | ALL [AROV] | DFAD, DFSB: RESULT TO AC, AC+1<br>DFMP, DFDV: RESULT TO AC, AC+1 | |

10-1643

Figure 2-74 Instruction Set Divisions

Once the instruction has been loaded into IR and ARX, the major machine cycle begins; this is shown in Figure 2-75.

Three functional flows and two tables are included to supplement the functional descriptions of the address, fetch, and store cycles that follow.

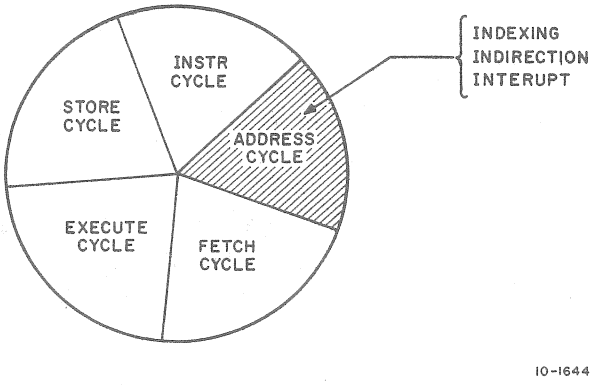



Figure 2-75 Major Machine Cycle

### 2.10.1 Effective Address Calculation

Figures 2-76 and 2-77 illustrate the instruction word formats. Bits 13–35 have the same format in every instruction whether the instruction addresses a memory location or not. Bit 13 is the indirect bit, bits 14–17 are the Index register address and, if the instruction must reference memory, bits 18–35 are the memory address Y. The effective address E of the instruction depends of the values of I, X, and Y.

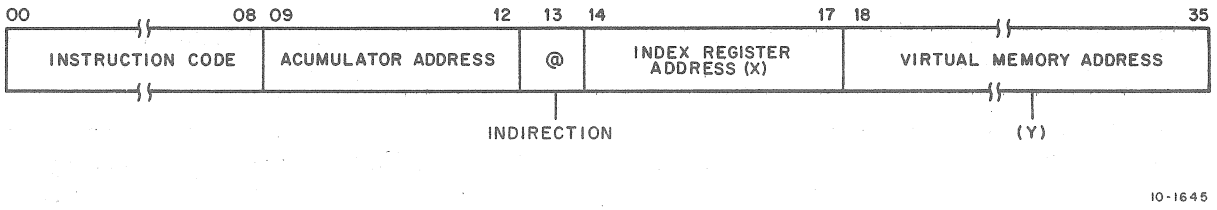



Figure 2-76 Basic Instruction Format

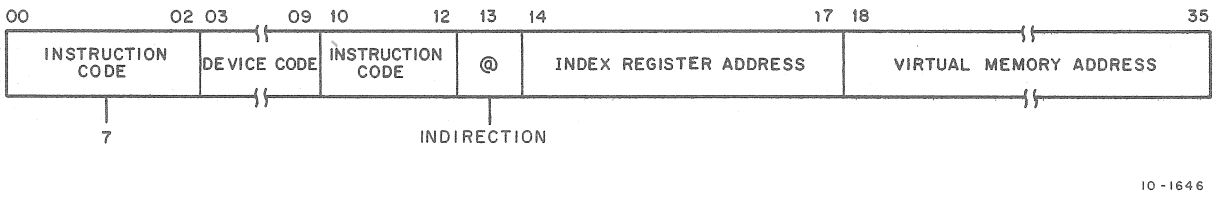



Figure 2-77 In-Out Instruction Format

**2.10.1.1 Indexing** – If the Index register address is nonzero, the contents of the specified Index register are added to the Y address to produce a modified virtual address.

Referring to Figure 2-78, the EBox tests ARX 14–17; if it is nonzero, the contents of the specified Index register are added to ARX 00–35. The result in AD 18– 35 is loaded into AR 18–35 with AR 00–17 cleared, and also loaded into VMA 18–35 while VMA 13–17 is recirculated.

**2.10.1.2 Indirection** – Whether indexing is performed or not, if ARX 13 is equal to 1, indirection will be performed. Two cases are to be considered. The first is where no indexing was performed. Here (indicated on Figure 2–78 as Ⓐ ) VMA 18–35 is loaded via AD with ARX 18–35. In the second case, indexing is performed and the VMA is loaded via AD with AR. Here AR holds the sum of ARX 18–35 and FM 18–35 effectively, with AD bits 00–17 clear.

In either case, VMA 13–17 is recirculated while VMA 18–35 will be loaded via AD. The microinstruction MEM field function for the indirect request is MEM/AIND. This function has MEM 02 = 0, so MBOX WAIT is conditionally a function of the next microinstruction.

**Testing for Interrupts**

The microinstruction causing the EBox request also tests for a pending priority interrupt. If an interrupt is pending, the CRAM address is modified to allow entry to the PI Handler (Figure 2-79).

The request, which is made both to fast memory and core memory via the MBox, is ignored as long as it does not page fault. MBOX WAIT is false, so the EBox clock does not stop at this time. The EBox ignores an indirect reference when an interrupt is pending, but the EBox hardware remembers a page fault (if one occurs) until the page fault handler has been called. After the PF Handler is called, Force 1777 will be cleared.

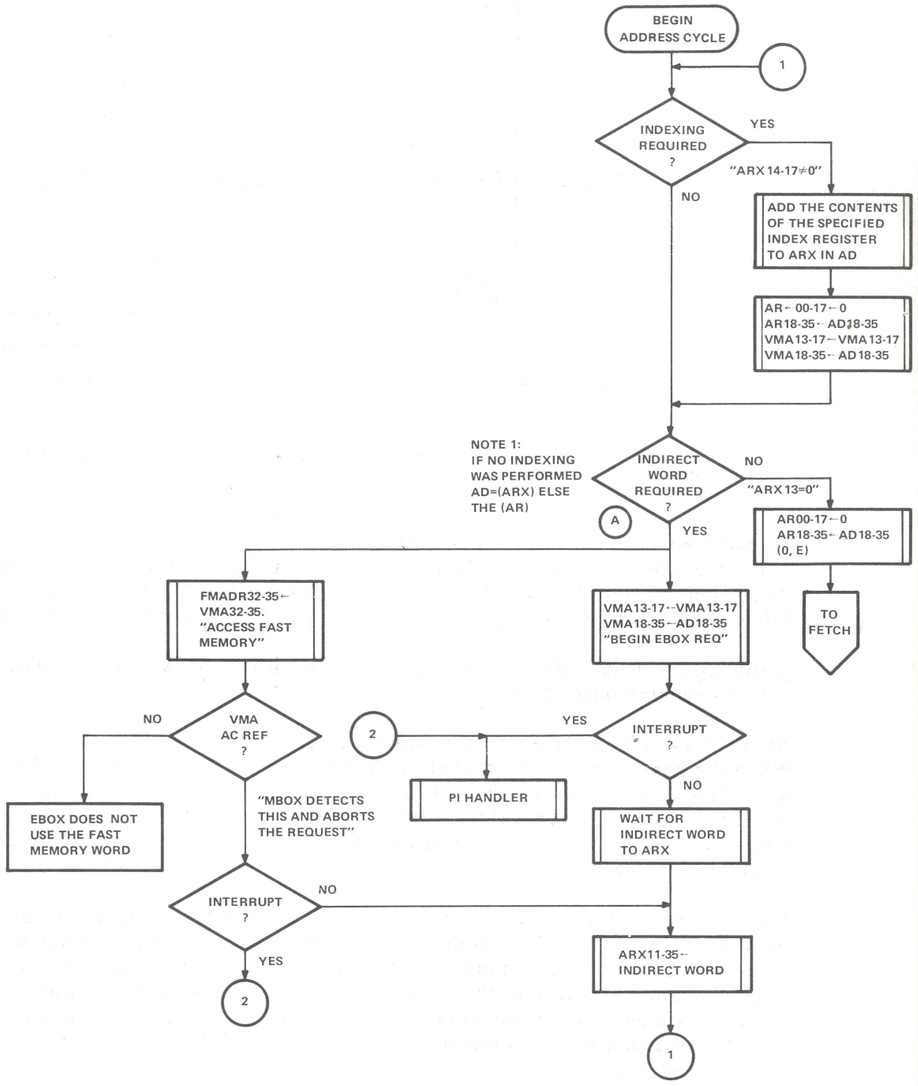
Referring to Figure 2-80, assume the indirect request has been started. Because the indirect reference is always a "READ," the only types of page faults that can occur in KI paging mode are no access (page not in core) or proprietary violation.

The requesting microinstruction detects the interrupt and the microprogram branches (via CRAM Address) to the PI Handler.

If the page fault occurs (for example) because of no access, the MBox must first read from the in core process table to obtain the paging information (use bits A, P, W, S, C and physical page). Reading this can take between 600 and 1000 ns. During this period, the PI Handler is setting up the requested PI service.

Eventually, a read, write or instruction fetch occurs, caused by the handler. When MBOX WAIT becomes true, the clock board (which remembered the Page Fail Hold level) forces the microprogram to the page fault handler.

Now the page fault handler detects the pending interrupt and the microprogram branches back to the PI Handler or to the instruction cycle. Thus, the entry to the page fault handler satisfied the clock board "page fail hold condition" and this condition now clears. Should the EBox make a second MBox reference before the page fault occurs, the EBox waits.




| | | | NUMBER OF EBOX CYCLES REQUIRED | | (A) | | SEE NOTE 2 | |
|---|---|---|---|---|---|---|---|---|
| REF CAN PAGE FAIL? | MBOX COMPLETE CYCLE? | VMA AC REF? | MBOX CYCLE | FAST MEM CYCLE | INDIRECT REF? | INTERRUPT ? | REQUESTING MICRO INSTR | NEXT MICRO INSTR |
| NO | NO MBOX TERMINATES | YES | BEGIN CYCLE, BUT EBOX IGNORES | BEGIN CYCLE, @ WORD TO ARX | YES | NO | MEM/AIND | MEM 02=1 MBOX WAIT |
| NO | NO MBOX TERMINATES | YES | BEGIN CYCLE, BUT EBOX IGNORES | BEGIN CYCLE BUT EBOX IGNORES | YES | YES-DIVERT TO PI HANDLER | MEM/AIND | MEM02=0 |
| YES, IF SO EBOX DIVERTS TO PF HANDLER | YES | NO | BEGIN CYCLE, @ WORD TO ARX | BEGIN CYCLE, BUT EBOX IGNORES | YES | NO | MEM/AIND | MEM02=1 MBOX WAIT |
| YES, BUT NOT ACTED UPON UNTIL THE NEXT MBOX WAIT* | YES | NO | BEGIN CYCLE, BUT EBOX IGNORES | BEGIN CYCLE, BUT EBOX IGNORES | YES | YES-DIVERT TO PI HANDLER | MEM/AIND | MEM02=0 |

*ONCE IN THE PAGE FAULT HANDLER THE INTERRUPT PENDING WILL CAUSE A RETURN TO THE PI HANDLER AND THE PF HOLD EN LEVEL WILL BE CLEARED, REMOVING TEMPORARILY ALL TRACES OF THE FAULT.

NOTE 2:
MEM CYCLE ∧ MEM 02(1) = MBOX WAIT
MBOX RESP OR FM RESP CAUSES MEM CYCLE TO CLEAR MBOX WAIT ∧~VMA AC REF.: EBOX CLOCK STOPS IF:

a. MBOX IS SERVICING THE EBOX REQ
b. WORD IS IN THE CACHE AND TIMEFIELD IS <3 OR . . .
a. MBOX IS SERVICING THE EBOX REQ
b. WORD IS NOT IN THE CACHE OR . . . .
a. MBOX IS SERVICING THE EBOX REQ
b. A PAGE FAULT OCCURS OR . . . .
a. A CONTROL RAM PARITY ERROR IS DETECTED OR . . . .

10-1647

Figure 2-78 Effective Address Calculation

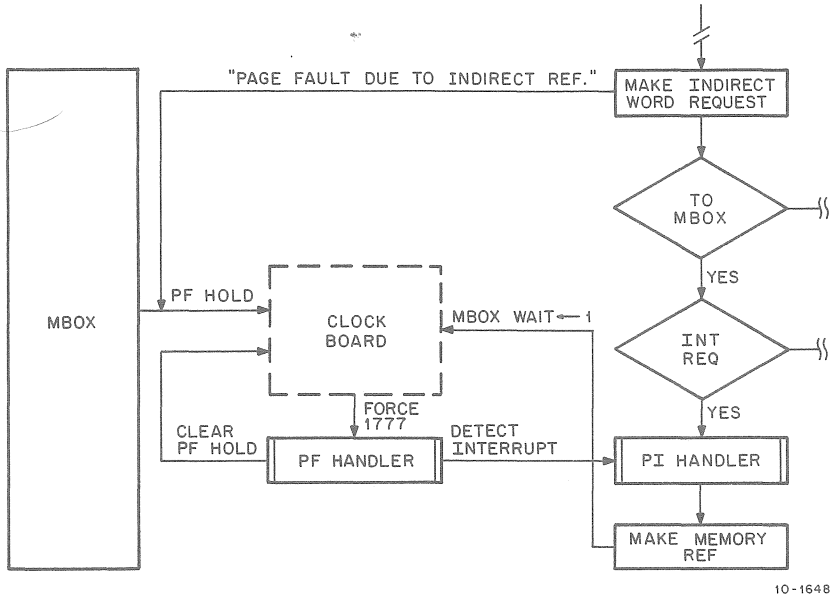



Figure 2-79 Page Fault During Diverted Indirect Reference

**Normal Case – No Interrupts, MBox Request**
When the EBox request is made specifically to the MBox, no interrupts are pending, the microinstruction following that which made the request (MEM/AIND) has its MEM field coded as ARX ← MEM. This function, together with MEM Cycle (1), will generate MBOX WAIT.

Assuming a page fault does not occur, the word loads into ARX. Now as indicated on Figure 2-79, the loop is reentered once again.

**Normal Case – No Interrupts, Fast Memory Request**
When the hardware determines that the VMA contains a fast memory address, it asserts VMA AC REF. This signal is used to inform the MBox that the EBox request is not to be handled by the MBox. Note that the fast memory address control uses VMA 32–35 to access fast memory even though the virtual address may be a core memory address. The hardware directs the use of the information accessed in this manner.

The effective address manager (Figure 2-15) branches within itself using the information provided from ARX 13 and 14–17. In addition, each time it samples this information it should branch to a microinstruction that enables the correct registers to be loaded; it may, however, invoke certain "don't care" operations, providing the next microinstruction executed performs the proper action. For example, assume a microinstruction is to always perform the indexing function in AD, but dispatch to a microinstruction that uses this information only if ARX 14–17 $\neq$ 0. This approach simplifies the design of the logic.

The table at the bottom of Figure 2-78 lists the four possible conditions resulting from indirect references to either MBox or fast memory.

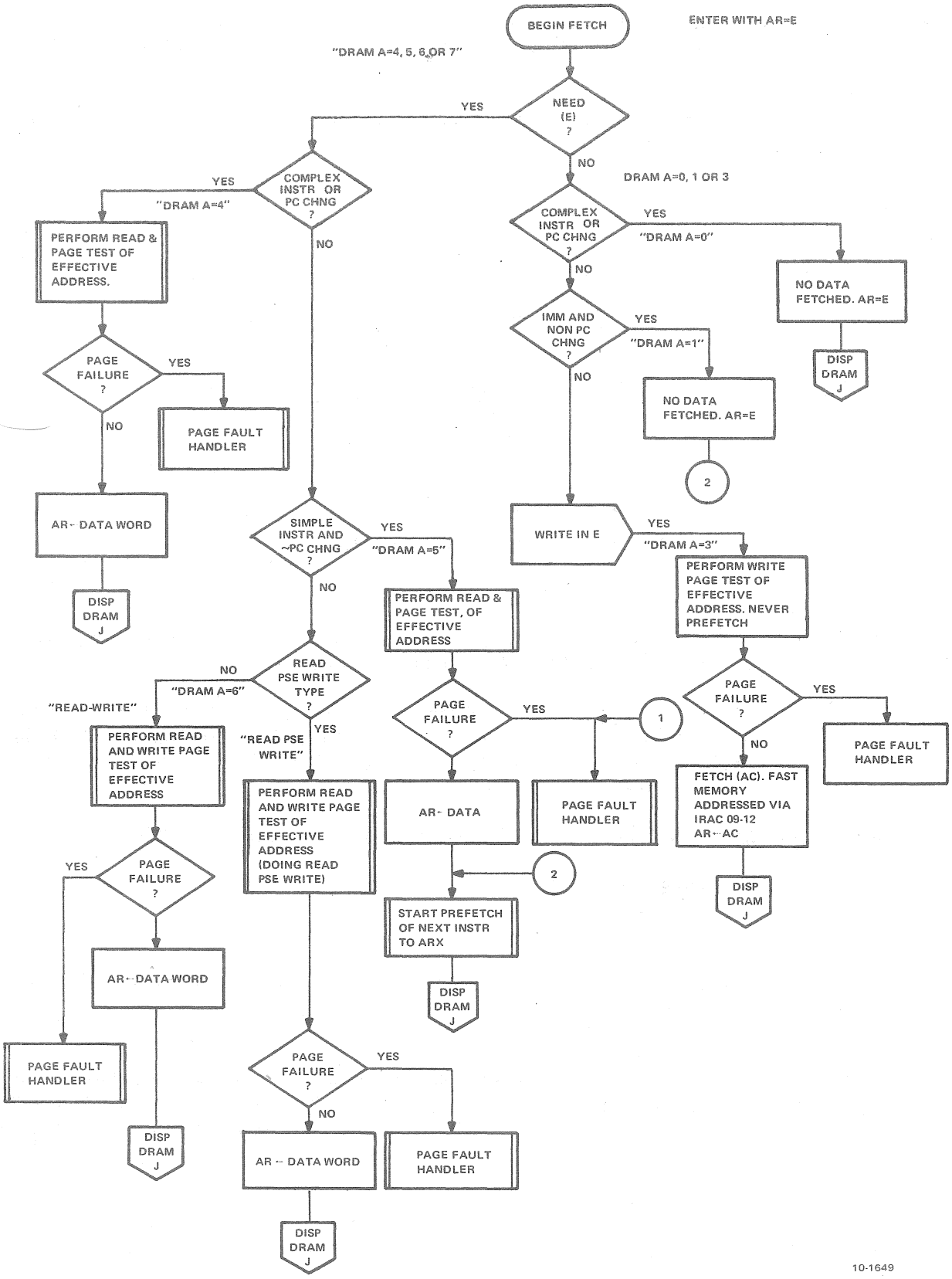



Figure 2-80 EBox Data Fetch

**2.10.1.3 No Indirection or Indexing** – For this case, ARX 18–35 contains the effective address. Here, it remains only to load AR 18–35 via AD with E and clear AR 00–17. The Fetch cycle is now entered.

**2.10.2 Fetch Cycle**
Once the effective address has been calculated, the second minor machine cycle is entered. This is the Fetch cycle and is illustrated in Figure 2-81.

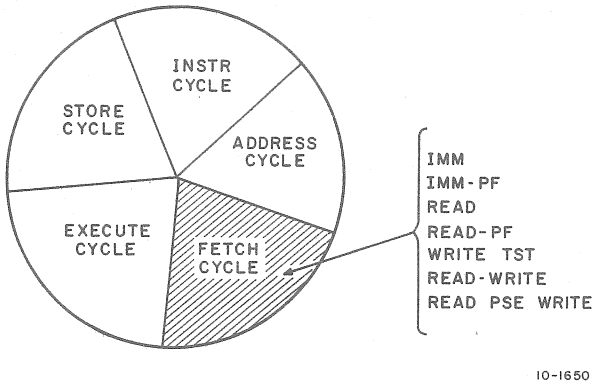



Figure 2-81 Fetch Minor Cycle

After the effective address has been calculated, the microprogram effective address manager gives "A READ DISPATCH" and control is passed to the Data Fetch Manager.

In general, two major classes of instructions exist in terms of the Data Fetch cycle. These two classes are those instructions that require the contents of the effective address and those that do not. Within each of these two categories are a number of divisions. The flow of the Fetch cycle is illustrated in Figure 2-80.

**2.10.2.1 Instructions That Do Not Require (E)** – Three general groups form this category.

1. Complex or PC change instructions
2. Immediate non-PC change instructions
3. Instructions that write in E

For these three groups, the DRAM A field is coded 0, 1, and 3, respectively. The AREAD Dispatch functions are listed in Table 2-16.

**Complex or PC Change Instructions**
The DRAM A field is coded as 0, and no data is requested. In addition, the next instruction is not prefetched. The AREAD/Dispatch dispatches directly to the execute code. This consists of a table lookup, where one discrete entry exists for each instruction. Thus, for example, the move instruction indexes into location "200" in the DRAM. The organization of the DRAM is illustrated in Figure 1-4.

**Immediate and Non-PC Change Instructions**
The DRAM A field is coded as 1, and no data is requested. The next instruction is prefetched and loads into ARX when the instruction becomes available. The AREAD/Dispatch dispatches directly to the execute code.

**Table 2-16 AREAD Dispatch**

| DRAM A | DISP/AREAD | MEM/AREAD | Require (E) |
|---|---|---|---|
| 0 | Executor | No Prefetch | No |
| 1 | Executor | Start Prefetch | No |
| 2 | Not used | | N/A |
| 3 | Symbolic Address 43* | Perform write test. | No |
| 4 | Symbolic Address 44* | "LOAD AR." | Yes |
| 5 | Symbolic Address 45* | A read operation is in progress: "LOAD AR, PREFETCH." | Yes |
| 6 | Symbolic Address 46* | LOAD AR. READ-PAUSE-WRITE | Yes |
| 7 | Symbolic Address 47* | LOAD AR, WRITE TEST | Yes |

*The Data Fetch manager is a combination of hardware mostly on MCL and the microprogram consisting of 43–47.

**Instructions That Write in E**

The DRAM A field is coded as 3 and a write page test is initiated. If the address is not writable, a page failure occurs. This action causes a transfer to the page fault handler as indicated in Figure 2-80.

The appropriate Fetch EBox Qualifiers may be determined by referring to Figure 2-82. For DRAMA = 3 the following qualifiers are specifically asserted:

EBOX REQUEST
EBOX PSE
EBOX WRITE

In addition, the state of the qualifiers is more complex and may depend on the previous history of the EBox. The state is indicated by an asterisk (*). Once again referring to Figure 2-80, if the write page test is successful, the EBox fetches the contents of the addressed fast memory location (via IRAC 09–12) and then dispatches via the DRAM J field to the executor.

| | | | | EBOX REQUEST QUALIFIERS | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CYCLE | MEM FUNC | DRAM A | DRAM B | EBOX REQ | EBOX READ | EBOX PSE | EBOX WRITE | EBOX USER | MAY BE PAGED | KI PAGING MODE | VMA AC REF | PAGE ILLEGAL ENTRY | PAGE TEST PRIVATE | PAGE ADR COND | CACHE LOAD | CACHE LOOK | REMARKS |
| ADDRESS | A IND FOLLOWED BY LOAD ARX | | | X | X | | | * | * | * | * | | | | * | * | INDIRECT WORD READ, MAY BE TO MBOX OR TO FAST MEMORY. VMA AC REF INDICATES WHICH VMA HOLDS ADR. |
| FETCH | FETCH | 1 OR 5 | | X | X | | | * | * | * | | • | | * | * | * | INSTR FETCH. MAY OCCUR FOLLOWING A READ WITH DRAM A=1 OR 5 TOGETHER WITH MEM/FETCH. |
| FETCH | A READ | 0 | | X | X | | | * | * | * | * | • | | * | * | * | INSTR FETCH FOR JRST 0 (IR=JRST0) |
| EXECUTE STORE | FETCH | | | X | X | | | * | * | * | * | • | | * | * | * | PI CYCLE IS CLEAR. USED WHERE NO PREFETCH WAS ISSUED TO CAUSE AN INSTR FETCH. |
| FETCH | A READ | 4–5 | | X | X | | | * | * | * | * | | ① | * | * | * | DATA READ ISSUED BY INSTRUCTIONS REQUIRING THE (E) AS FOLLOWS: COMPLEX OR PC CHANGE INSTRUCTIONS OR SIMPLE NON PC CHANGE INSTRUCTIONS. ① ASSERTED IF ATTEMPTING TO READ DATA FROM A PRIVATE ADDRESS SPACE WITHOUT PROPER PROTOCOL. MBOX READ PAGE TESTS. |
| FETCH | A READ | 6 | | X | X | | X | * | * | * | * | | ② | * | * | * | DATA READ–WRITE ISSUED BY INSTRUCTIONS REQUIRING THE (E) WHICH CONDITIONALLY WRITE INTO E. THESE INSTRUCTIONS ARE AS FOLLOWS: NON READ PSE WRITE TYPE ② ASSERTED IF ATTEMPTING TO READ DATA FROM A PRIVATE ADDRESS SPACE WITHOUT PROPER PROTOCOL. MBOX READ AND WRITE PAGE TESTS. AR LOADS. |
| FETCH | A READ | 7 | | X | X | X | X | * | * | * | * | | ③ | * | * | * | DATA READ PSE WRITE ISSUED BY INSTRUCTIONS REQUIRING THE (E) WHICH WILL UNCONDITIONALLY WRITE INTO E. ③ ASSERTED IF ATTEMPTING TO READ DATA FROM A PRIVATE ADDRESS SPACE WITHOUT THE PROPER PROTOCOL. MBOX READ AND WRITE PAGE TESTS. IF CACHE IS DISABLED FOR THE CYCLE MBOX WAITS FOR WRITE PORTION OF CYCLE. I.E., PT CACHE (0) OR CACHE LOAD (0)∧NOT FOUND. AR LOADS. |
| FETCH | A READ | 3 | | X | | X | X | * | * | * | * | | ④ | * | * | * | DATA WRITE PAGE TEST ONLY. ISSUED BY INSTRUCTIONS NOT REQUIRING (E) BUT WHICH WILL WRITE INTO E. ④ ASSERTED IF ATTEMPTING TO WRITE DATA INTO A PRIVATE ADDRESS SPACE WITHOUT THE PROPER PROTOCOL. MBOX WRITE PAGE TESTS. |
| STORE | B WRITE | 2–3 | | X | | | X | * | * | * | * | | ⑤ | * | * | * | DATA WRITE (WRITE PAGE TEST AND WRITE DATA) USED BY THE GENERAL 4 MODE TYPE INSTRUCTIONS. I.E., IMM, BASIC, MEM, SEL FOR BOTH. SELF MODE STORES CONDITIONALLY IN E WHILE BOTH MODES ALWAYS STORE IN E. IN ADDITION BOTH MODES STORE UNCONDITIONALLY IN AC WHILE SELF E MODE STORES CONDITIONALLY IN AC. STORE VIA AR. ⑤ SAME AS ④. |
| EXECUTE | BYTE IND | | | X | X | | | * | * | * | * | | ⑥ | * | * | * | BYTE POINTER INDIRECT WORD READ. USED AFTER BYTE POINTER HAS BEEN FETCHED WHEN BIT 13 OF THE POINTER IS 1. USED ONLY BY BYTE TYPE INSTRUCTIONS. ACTS LIKE EBOX READ TO MBOX. MBOX READ PAGE TESTS. BOTH AR AND ARX ARE LOADED. ⑥ SAME AS ③. |
| EXECUTE | BYTE RD | | | X | X | | | * | * | * | * | | ⑦ | * | * | * | BYTE DATA READ. USED AFTER BYTE INDIRECT HAS COMPLETED. USED BY BYTE TYPE INSTRUCTIONS. ACTS LIKE EBOX READ TO MBOX. MBOX READ PAGE TESTS. BOTH AR AND ARX ARE LOADED. ⑦ SAME AS ⑥. |
| EXECUTE STORE MISC | WRITE | | | X | | | X | * | * | * | * | | ⑧ | * | * | * | GENERAL PURPOSE WRITE. USED MANY PLACES. SOME EXAMPLES WOULD BE INSTRUCTIONS WHICH STORE MORE THAN ONE OPERAND, SUCH AS DOUBLE TYPE INSTRUCTIONS. INSTRUCTIONS WHICH SKIP, OR MODIFY AND SKIP BUT DID NOT FETCH (E) AND ARE GOING TO WRITE INTO E. MBOX TREATS AS WRITE. WRITE PAGE TESTS. |
| EXECUTE | LOAD AR | | | X | X | | * | * | * | * | * | | | * | * | * | |

• IF AN INSTRUCTION IS FETCHED BY A PUBLIC PROGRAM FROM A PRIVATE ADDRESS SPACE, AND THE INSTRUCTION IS NOT A PORTAL, ILL ENTRY WILL CAUSE THE MBOX TO PAGE FAIL ON THE NEXT MBOX REF.

* THESE QUALIFIERS ARE TRUE OR FALSE DEPENDING ON THE SPECIFIC TYPE OF REQUEST BEING MADE.

10-1651

Figure 2-82 Address-Fetch-Execute-Store General Memory References

**2.10.2.2 Instructions That Require (E)** – Under this category are four general groups. These groups are as follows:

1. Complex or PC change instructions
2. Simple non-PC change instructions
3. Non-(read-PSE-write) type instructions
4. Read PSE write type instructions

For these four groups, the DRAM A field is coded 4, 5, 6, or 7, respectively.

**Complex or PC Change Instructions**
The DRAM A field is coded as 4, causing a dispatch to location 44. A read page test is performed by the MBox. If the address is not accessible (not in core), the MBox performs a refill cycle and then checks the use bits.

If the access bit is clear, a page fault occurs and the EBox transfers to the page fault handler (microcode page fault handler). Otherwise, the requested word is loaded into AR. For the appropriate EBox qualifiers, refer to Figure 2-83. Finally, a DRAM J dispatch is performed to the executor.

**Simple Non-PC Change Instructions**
The DRAM A field is coded as 5, causing a dispatch to location 45. The basic read is the same as for DRAM A = 4. If no page fault occurs, the MBox issues MBox RESPONSE with the data word. Now the VMA loads with the prefetch address and this cycle begins. This MBox cycle will run in parallel with the Execution cycle, which may not use ARX. Finally, a DRAM J dispatch is performed at location 45; the VMA is loaded with PC + 1 and the prefetch begins.

**Non-Read PSE Write-Type Instructions**
A number of instructions are in this category; a few examples follow.

The first example is SETMB. This instruction (Boolean Group), reads a word from memory and unconditionally stores it in memory and AC. Because writing the word back into the same address is redundant, only a write page test is required to assure that the word (if in core memory) is writable. If this page fails, then the operation is aborted anyway. Otherwise, the word read is stored only in fast memory as addressed by IRAC 09–12. The read-write (separate cycles) may be thought of as consisting of a read and conditional write. If the write cycle is really desired, the MEM field function MEM/Write may be used to write (Figure 2–82).

The second example concerns instructions such as IDIVM, IDIVB, DIVM, and DIVB. These instructions reference memory for both read and can generate no divide. This aborts the division operation. If the class of instruction is read PSE write and the cache is disabled for the reference, then the MBox waits for the write portion of the cycle; the EBox performs an unnecessary write operation.

A third cause is for BLKI and BLKO I/O instructions. Here a pointer word is fetched from the effective address. This pointer is normally updated and stored back in the effective address.

One problem is that the legality of performing the I/O instruction is tested after the pointer has been fetched. This is necessary because the pointer is fetched during the Fetch cycle, while legality (IO LEGAL) is tested during execution. Should the BLKX instruction be illegal in the current EBox mode, an unnecessary pointer back off and write would be necessary.

Other cases are concerned with very long instructions, which could hold up the MBox.

| CYCLE | COND FUNC | SPEC FUNC | MEM FUNC | # FIELD 0 – 8 | EBOX REQUEST QUALIFIERS: EBOX REQ | VMA EPT | VMA UPT | PAGE UEBR REF | EBOX MAP | EBOX LOAD REG | EBOX READ REG | EBOX CCA | EBOX UBR | EBOX EBR | EBOX ERA | EN REFILL RAM WR | EBOX SBUS DIAG | WRPT SEL 0 | WRPT SEL 1 | PT WR | PT DIR WR | MBOX CTL 03 | MBOX CTL 06 | REMARKS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EXECUTE MUUO PART 1,2. METER REQUEST, PAGE FAIL PART 1,2. | | SP MEM CYCLE | WRITE SEE FIG 2-54 | #220 | X | | X | X | | | | | | | | | | | | | | | | WRITE INTO USER PROCESS TABLE. MBOX USES VMA 27–35 ONLY. MBOX APPENDS UBR 14–26 TO FORM PHYSICAL REFERENCE UNPAGED. CANNOT PAGE FAIL. |
| EXECUTE MUUO PART 3, PAGE FAIL PART 3 | | SP MEM CYCLE | LOAD AR FOLLOWED BY MB WAIT SEE FIG 2-54 | #220 | X | | X | X | | | | | | | | | | | | | | | | READ NEW PC WORD FROM USER PROCESS TABLE. MBOX USES VMA 27–35 ONLY. MBOX APPENDS UBR 14–26 TO FORM PHYSICAL REFERENCE UNPAGED. CANNOT PAGE FAIL. |
| EXECUTE DEPOSIT | | SP MEM CYCLE #08(1): PHYS REF | WRITE SEE FIG 2-54 | #110 | X | X | | X | | | | | | | | | | | | | | | | FOR DEPOSIT WRITE ACCORDING TO RELOCATED VIRTUAL ADDRESS. THERE WAS NO PROTECTION VIOLATION. |
| EXECUTE EXAMINE, DEPOSIT, METER REQ | | SP MEM CYCLE #08(1): PHYS REF | LOAD AR FOLLOWED BY MB WAIT SEE FIG 2-54 | #110 | X | X | | X | | | | | | | | | | | | | | | | FOR DEPOSIT OR EXAMINE, READ PROTECTION AND RELOCATION INFORMATION FROM EXEC PROCESS TABLE. FOR MTR REQUEST READ DOUBLE PRECISION WORD AS APPROPRIATE. |
| EXECUTE STD 40+2N INT 2ND PART STD INT | | SP MEM CYCLE #00(1): FETCH EN IN | LOAD ARX | #510 | X | X | | X | | | | | | | | | | | | | | | | INSTR FETCH FROM EPT FOR STD INTERRUPT OR 2ND PART OF STD INTERRUPT. MBOX USES VMA 27–35 ONLY. MBOX APPENDS EBR 14–26 TO FORM PHYSICAL REFERENCE UNPAGED. CANNOT PAGE FAIL. |
| EXECUTE NICOND PI VECT, EXAMINE SEE FIG 1-25 | | SP MEM CYCLE #08(1): PHYS REF | LOAD AR FOLLOWED BY MB WAIT SEE FIG 2-54 | #110 IF EPT SPECIFIED SEE FIG 1-25 | X | * | SPEC EXEC KILLS USER EN | * | | | | | | | | | | | | | | | | FETCH WORD FROM ADDRESS SPACE SPECIFIED VIA API WORD BITS 0–2 MAY BE: EPT, EXEC VIRTUAL, UPT, USER VIRTUAL, PHYSICAL. THE STATE OF VMA EPT, VMA UPT, MAY BE PAGED CONTROLS CONTEXT OF REFERENCE. |
| NICOND TRAP FETCH | | SP MEM CYCLE | LOAD ARX FOLLOWED BY LOAD ARX | #30 | X | * | * | * | | | | | | | | | | | | | | | | FETCH INSTR FROM USER PROCESS TABLE IF TRAP OCCURRED IN USER MODE AND EXEC PROCESS TABLE IF TRAP OCCURRED IN EXEC MODE. THE STATE OF VMA UPT AND VMA EPT IS A FUNCTION OF MCL USER EN. |
| EXECUTE BLKO PI | I/O LEGAL | | REG FUNC | #7 | X | | | | | | | | | | | | X | | | | | | | TRANSMITS 36 BITS OF CONTROL INFORMATION FETCHED FROM E TO THE INTERNAL OR EXTERNAL MEMORIES. THIS IS PERFORMED VIA THE MBOX. THE MEMORY CONTROLLER SELECTED RETURNS A WORD WHICH IS STORED IN E+1. |
| EXECUTE DATAI PAG | I/O LEGAL | | REG FUNC | #142 | X | | | | | | X | | X | | | | | | | | | | | READS INFORMATION FROM EBOX AND MBOX. FIRST READS AC BLOCKS, CWSX AND VMA PREV SECT THEN REQUESTS MBOX TO PUT UBR INTO EBUS REG. EBOX THEN READS EBUS REG. |
| EXECUTE CONI PAG | I/O LEGAL | | REG FUNC | #143 | X | | | | | | X | | | X | | | | | | | | | | READS INFORMATION FROM EBOX AND MBOX. FIRST READS LOOK, LOAD, SEC TRAPEN, THEN REQUESTS MBOX TO PUT EBR INTO EBUS REG. EBOX THEN READS EBUS REG AND STORES RESULT IN E. |
| EXECUTE BLKI PI | I/O LEGAL | | REG FUNC | #144 | X | | | | | | X | | | | X | | | | | | | | | READS THE MBOX ERROR ADDRESS REGISTER. THE WORD IS STORED IN THE SPECIFIED AC. |
| EXECUTE BLKO PAG | MBOX CTL | | | STEP 1=#1 STEP 2=#21 STEP 3=#0 | | | | | | | | | | | | | | | X | | X | | | INVALIDATE ENTRIES IN THE PAGE TABLE WITHIN THE MBOX. EBOX USER REFLECTS USER MODE, AND THE PAGE TO INVALIDATE IS IN VMA. |
| EXECUTE DATAO PAG | I/O LEGAL | | REG FUNC | #242 | X | | | | | X | | | X | | | | | | | | | | | LOADS CONDITIONALLY AC BLKS, PREV CONTEXT, AND UBR (IN MBOX) AND CLEARS THE PAGE TABLE IN THE MBOX. VMA 25–35 CONTAIN THE DATA TO BE LOADED INTO UBR. |
| EXECUTE CONO PAG | I/O LEGAL | | REG FUNC | #243 | X | | | | | X | | | | X | | | | | | | | | | LOADS THE CACHE STRATEGY BITS (LOOK, LOAD) SECTION AND TRAP EN FLAGS IN THE EBOX. THE EBR (IN THE MBOX) IS LOADED VIA VMA 24–35. |
| EXECUTE MAP | I/O LEGAL | | REG FUNC | #540 | X | | | | X | | X | | | | | | | | | | | | | READS PAGE FAILWORD FROM MBOX EBUS REGISTER. THE WORD IS STORED IN THE SPECIFIED AC. |
| EXECUTE BLKO APR | I/O LEGAL | | REG FUNC | #145 | X | | | | | X | | | | | | X | | | | | | | | WRITE THE CACHE REFILL ALGORITHM. VMA 18–20 CONTAINS THE ALGORITHM BITS AND VMA 27–33 CONTAIN THE REFILL ALGORITHM ADDRESS. |

10 1652

Figure 2-83 Execute-Register-MBox Control and Miscellaneous General Memory References

The DRAM A field is coded as 6, causing a dispatch to location 46; the MBox performs both a read and write page test. The address must be both accessible and writable, even though this portion of the operation only reads a word. If a page failure occurs, the EBox transfers control to the page fault handler. Otherwise, the word enters AR and then a DRAM J dispatch is issued.

**Read PSE Write Type Instruction**
The DRAM A field is coded as 7 causing a dispatch to location 47; the request qualifiers are shown on Figure 2-82. The MBox performs both a read and write test, and if no page fault occurs, reads a word from the specified (Xlated) address.

If the cache is disabled for the reference and the word requested was not in the cache (a Refill cycle was necessary first), then the MBox is held waiting until the EBox issues the write portion of the cycle. The word requested loads into AR and a DRAM J dispatch is issued to enter the Executor.

### 2.10.3 Execution Cycle

The Executor is entered from the Fetch cycle. While in the Fetch cycle, the (E) or (AC) is fetched in accordance with the DRAM A field. In addition, read and/or write page testing is performed while in the Fetch cycle. The EBox Execution cycle overview is in Figure 2-84.

Early in the Instruction cycle, the DRAM is accessed using one of three basic types of addresses.

Referring to Figure 2-84, if the instruction is JRST 0–17, then the IR address is used to address the DRAM initially as indicated. Thus, the JRSTs handler is entered at location 254 for JRST and 255 for JFCL.

From the initial dispatch into the handler, the IRAC is used to redispatch within the handler for the proper type of JRST. For JFCL, a JUMP is made to a separate handler from the initial dispatch

If the instruction is an I/O type, then the DRAM address is formed by the hardware such that the dispatch is in the range of 700–777. Once the I/O handler has been entered, a determination must be made as to whether the instruction is legal in the current processor mode. If it is determined that the instruction is not legal, the MUUO executor is used to store the illegal instruction and PC word in the user process table. Following this, a new PC word is fetched. This new PC word causes the processor to enter an executive routine in core memory. If the I/O instruction is legal, use of the EBus is obtained and the appropriate EBus dialogue is carried out. The specific actions evoked depend upon the device and the type of I/O instruction being performed.

The remaining instructions index into the DRAM utilizing the op code in IR bits 00–08. Two general categories exist as follows:

1. Simple Type – stores in AC, E, or both

2. Complex Type – may store in AC, AC+1, E via normal store cycle or else store via a special handler, or may do some of each

The complex instructions may nest microcode subroutines up to four levels deep.

Referring to Figure 2-85, the mechanism consists of CRA LOC, a register that is loaded with the "current microinstruction address." This register is loaded at the same time that the CRAM register is loaded with a new microinstruction. In addition, a 4-word stack is provided. The contents of CRA LOC are pushed onto the top of the stack when the call has been asserted by the microinstruction. To return from a subroutine, the returning microinstruction asserts DISP/Return. This pops the top entry off of the stack and onto the CRAM address mixer lines, where it is logically ORed with the J field of the microinstruction, asserting DISP/Return.

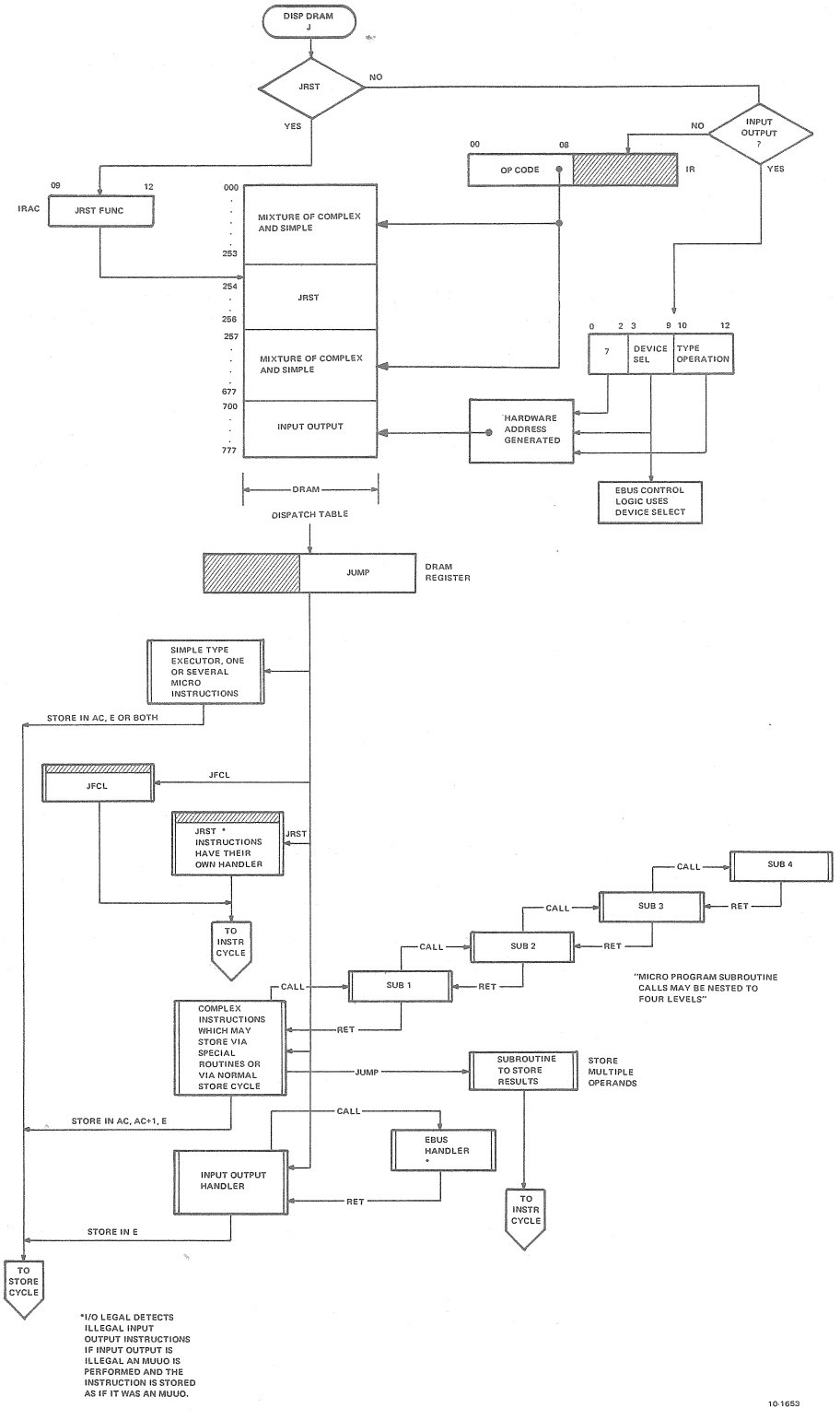



Figure 2-84 EBox Execution Cycle Overview

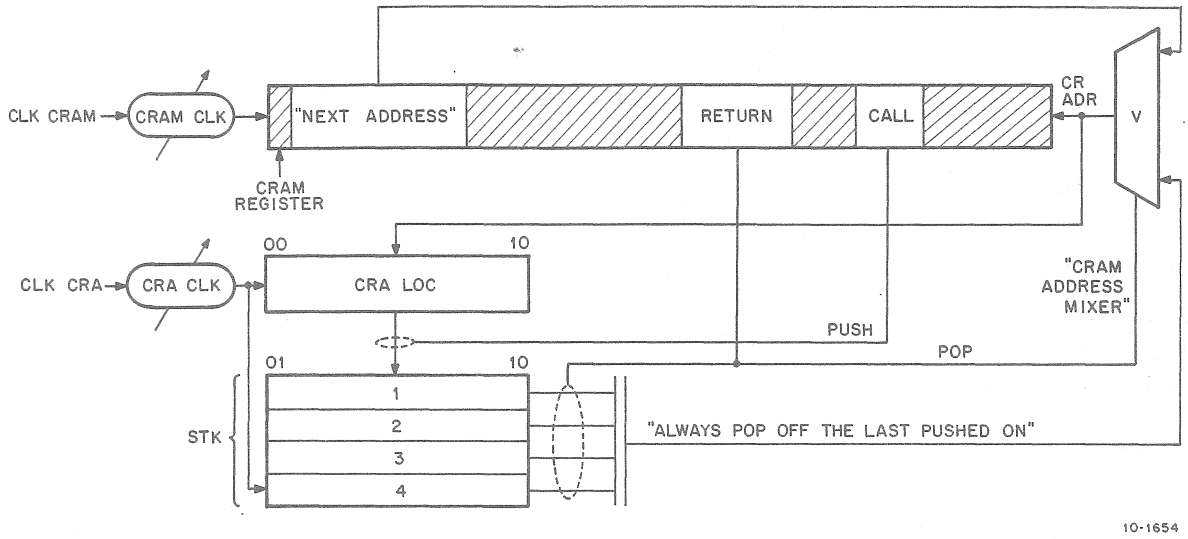



Figure 2-85 Microstack Operation

Some of the complex instructions, such as DMUL, which stores in AC, AC+1, AC+2, and AC+3, use a separate handler for storing multiple operands. This type of instruction does not pass through the normal store cycle. Other complex instructions, such as MULB, which stores in AC, AC+1 and E, store multiple operands via the normal store cycle.

### 2.10.4 EBox Data Store Cycle

The flow for the EBox Store cycle, illustrated in Figure 2-86, is used by most of the instructions executed by the microprogram Executor. Exceptions to this are certain instructions such as DMUL, which stores more than two ACs. For these instructions, a special handler exists that is entered from the executor. This handler stores all the operands and then issues an instruction fetch followed by a NICOND Dispatch. In this text, the more general categories (which do use the normal store cycle) are covered.

**2.10.4.1 Basic Four Mode Type Instructions** – This type of instruction may have one of four basic modes as follows:

1. Immediate or Basic – store in AC only
2. Memory – store in E
3. Both – store both in AC and E
4. SELF – store in E and conditionally store in AC. Note that if writing back in E is redundant, the write cycle is skipped.

Writing for these four mode instructions is controlled by MEM/DRAM B and the DRAM B field code. The store cycle is dispatched with DISP/DRAM B. Thus, the dispatch RAM B field (three bits) is used to form the low-order three bits of the Store cycle address.

**Immediate or Basic Mode**

Referring to Figure 2-87, the DRAM B field is coded as 5. The contents of AR are written into fast memory, which is addressed via IRAC 09–12. Because a large number of these instructions prefetch the next instruction, it is necessary to assert MB WAIT in the event MEM cycle is set waiting for a response from the MBox. This has no affect if MEM cycle is clear. NICOND Dispatch enables entry to the instruction cycle if no priority interrupts, page faults, or traps are pending.

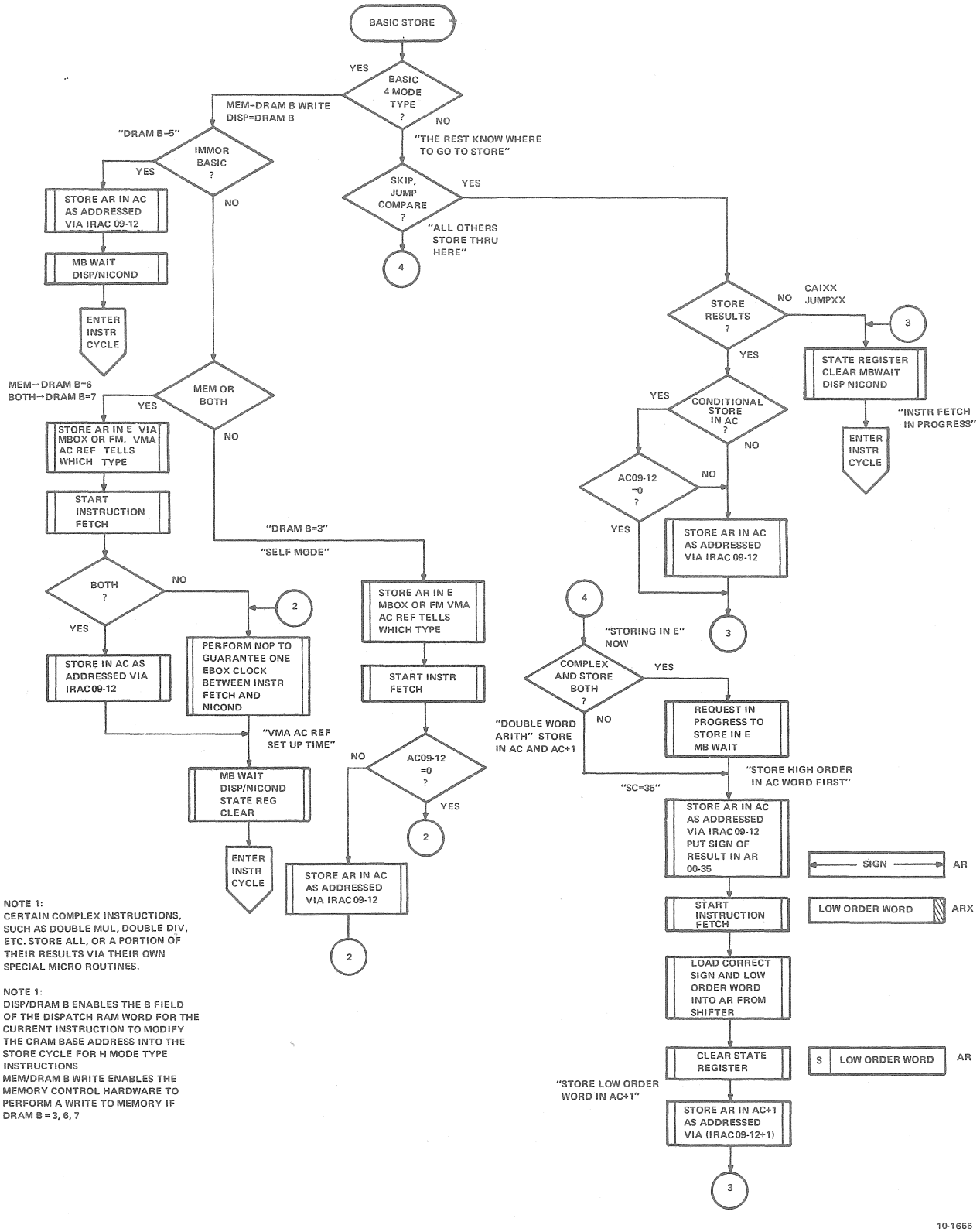



Figure 2-86 EBox Data Store

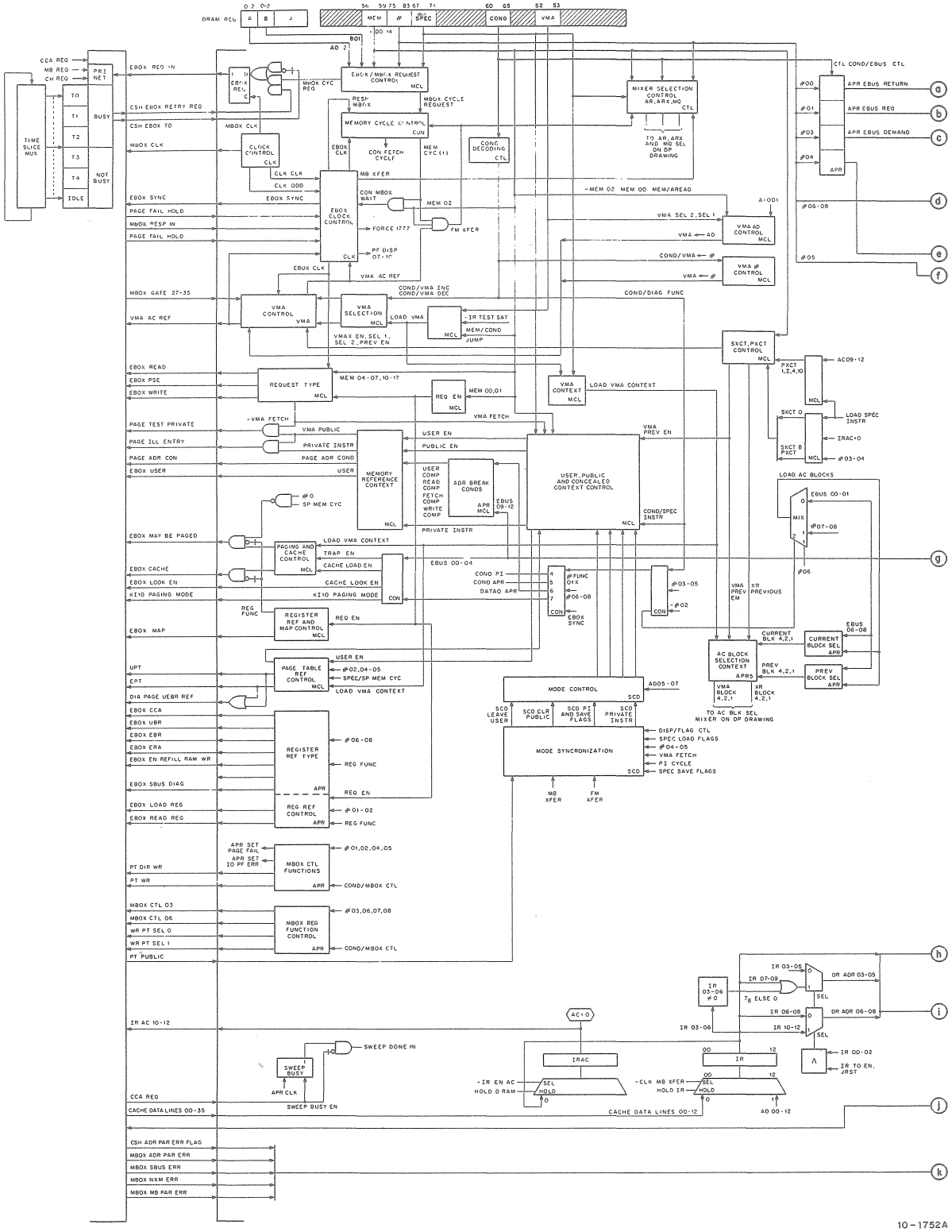



Figure 2-87 MBox-EBox-EBus Control (Sheet 1 of 2)

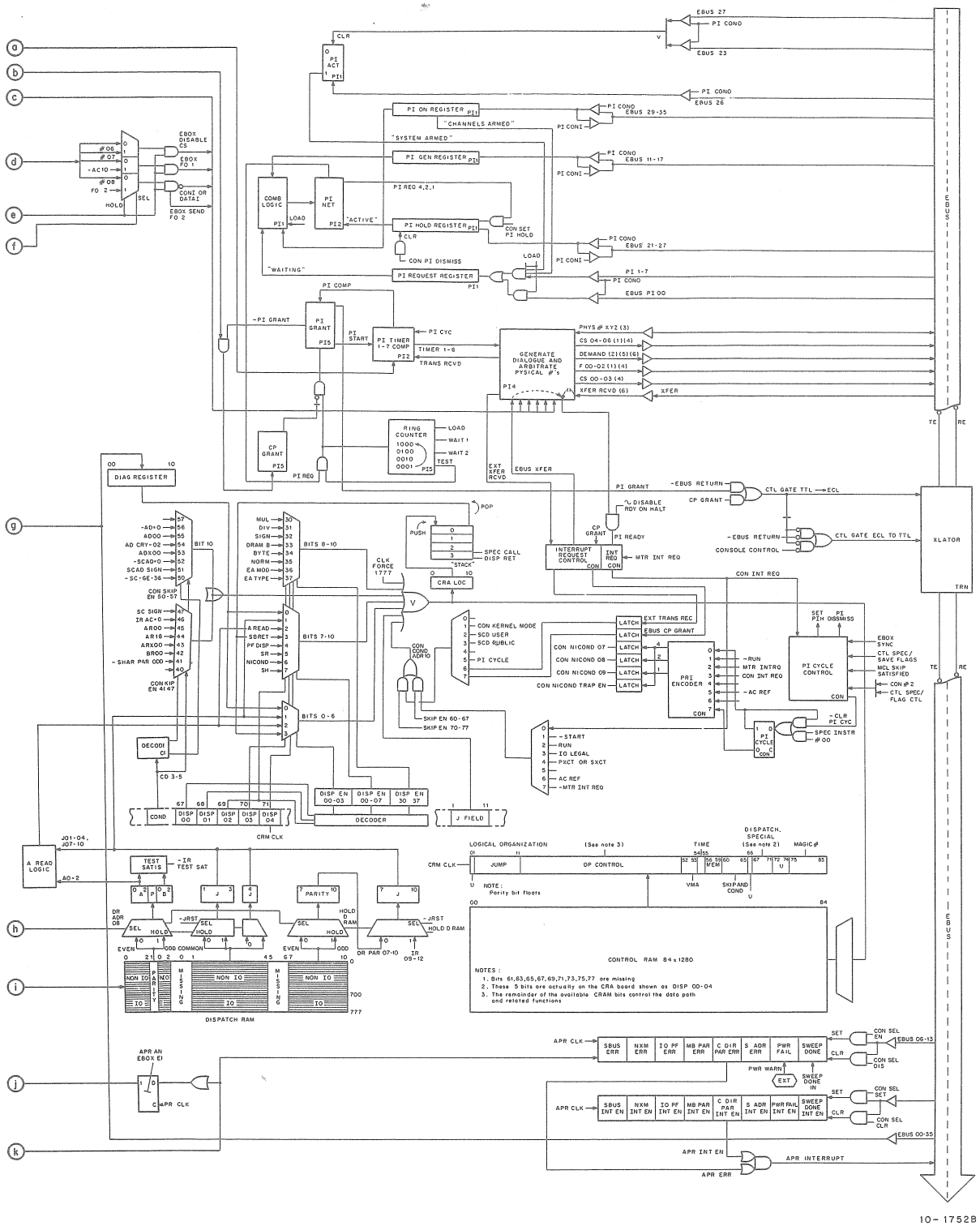



Figure 2-87 MBox-EBox-EBus Control (Sheet 2 of 2)

**Memory or Both Mode**
The DRAM B field is coded as 6 for memory mode instructions. If VMA 13–33 is clear, storing is to fast memory. Otherwise, an MBox request is made to store AR in cache memory. VMA AC REF notifies the MBox to abort the cycle when it is to fast memory. An unconditional instruction fetch is enabled at this time. The VMA input is via VMA AD (PC+1) and, as soon as MBox RESPONSE is received, this is latched into VMA.

To allow VMA addressing to stabilize in case the instruction is being fetched from fast memory, a NOP microinstruction is performed. This is followed by MB WAIT, state register clear (in case the instruction fetch page fails), and finally NICOND Dispatch is issued.

For Both Mode, DRAM B is coded as 7. Here, the departure is made after storing the AR in E. The AR is also stored in fast memory as addressed by IRAC 09–12. Now MB WAIT is asserted while clearing the state register and NICOND Dispatch is issued.

**SELF Mode**
Once again referring to Figure 2-87, the DRAM B field is coded as 3. SELF mode instructions are generally read/write type; this means that the virtual address was read and write page tested during the fetch cycle.

Writing is allowed only if not redundant, or as specified by IRAC being nonzero. AR is stored in E, the instruction fetch is started, and the AC field of the instruction is tested (in IRAC). If IRAC is nonzero, the AR is stored in the addressed fast memory location (as addressed via IRAC). If IRAC is zero, no storing in fast memory is performed. In either case, a microinstruction NOP is performed. This guarantees one EBox clock between the instruction fetch and the NICOND Dispatch to follow, allowing adequate setup time for the NICOND logic to detect a fast memory reference (VMA AC REF) for those cases where the instruction fetch is to fast memory.

**2.10.4.2 SKIP, JUMP Compare Instructions** – The following instructions listed in Table 2-17 fall into this category.

**Table 2-17 Skip, Jump, Compare Instructions**

| Main Group | Instr | Unconditional Store | Conditional Store | Stores Nothing | Op Code |
|---|---|---|---|---|---|
| Arithmetic Skips | SKIPXX | No | Yes if IRAC ≠ 0 | No | 330–337 |
| | AOSXX | Yes | Yes if IRAC ≠ 0 | No | 350–357 |
| | SOSXX | Yes | No | No | 370–377 |
| Conditional Jumps | JUMPXX | No | No | Yes | 320–327 |
| | AOJXX | Yes | No | No | 340–347 |
| | SOJXX | Yes | No | No | 360–367 |
| Arithmetic Testing | AOBJP | Yes | No | No | 252 |
| | AOBJN | Yes | No | No | 253 |
| Compares | CAIXX | No | No | Yes | 300–307 |
| | CAMXX | No | No | Yes | 310–317 |

**No Results Stored – CAIXX, JUMXX**
Referring to Figure 2-87, because CAIXX and JUMPXX store no results, preparations are made for entry to the instruction cycle. The state register is cleared, MB WAIT is asserted, and a NICOND Dispatch is issued. Depending upon the outcome of Test Satisfied, the next instructionfetch is from PC+1, PC+2, or E.

**Conditional Storage in AC – SKIPXX AOSXX, SOSXX**
IRAC is sampled and if nonzero, AR is stored in fast memory as addressed via IRAC 09–12. Depending upon the outcome at Test Satisfied, the next instruction fetch is from PC+1 or PC+2 and this is in progress. The state register is cleared, MB WAIT is asserted, and a NICOND Dispatch is issued.

**Unconditional Storage – SOJXX, AOJXX, AOBJX**
These instructions all store unconditionally, in fast memory from AR, as addressed via IRAC, then prepare to enter the Instruction cycle. The state register is cleared, MB WAIT is asserted, and NICOND Dispatch is issued. Both SOSXX and AOSXX unconditionally store in E and conditionally store in AC.

**2.10.4.3 Store Cycle for Other Instructions** – Generally, the remaining instructions that use the Store cycle fall into two groups. These are instructions that store results in AC, AC+1 and E, and those instructions that store results in AC and AC+1 only. All these are complex instructions.

**Complex and Store Both**
For these instructions, the store flow is entered with a write request already in progress to store the high-order result of some operation and MB WAIT is asserted (MEM/MBWAIT). Also, the shift counter (SC) contains 35, enabling alignment of the low-order word with the sign of the high-order word later in this flow. The AR is now stored in fast memory as addressed via IRAC and the sign is smeared in AR 00–35. At this time, AR contains all sign bits and ARX contains the low-order word left-justified. The instruction fetch begins. The AR and ARX are shifted left 35 places and the result (correctly signed) is loaded into AR via SH. Now the state register is cleared and the low-order word (in AR) is stored in IRAC + 1. The EBox hardware facilitates the incrementation of IRAC by +1. Finally, the appropriate entry to the instruction cycle is made.

**Complex and Store in AC, AC+1**
The basic difference here is that these instructions bypass the storage into E. Otherwise, the operation is identical to that for Complex and Store Both.

## 2.11 INTERFACE CONTROL

### 2.11.1 Introduction
Figure 2-88 illustrates the major functional control elements of the EBox. The purpose of this drawing is to support the functional descriptions contained in this section. In addition, it is provided to support the E/M interface control and E/E interface control functional descriptions to follow.

The EBox is associated with two interfaces, the EBox/MBox Interface and the EBox/EBus Interface. The E/M interface is treated as a pseudo-bus because in many ways it behaves as a bus. In the first portion of the functional description, the basic organization and function of the firmware microprogram was described. In addition, the major machine cycle was defined and described in terms of its functional elements.

Thus, the individual microprogram modules (Figure 2-13), taken collectively, comprise the main microprogram. The blending of this program with certain pieces of EBox hardware constitutes the basic machine cycle (Figure 2-88).

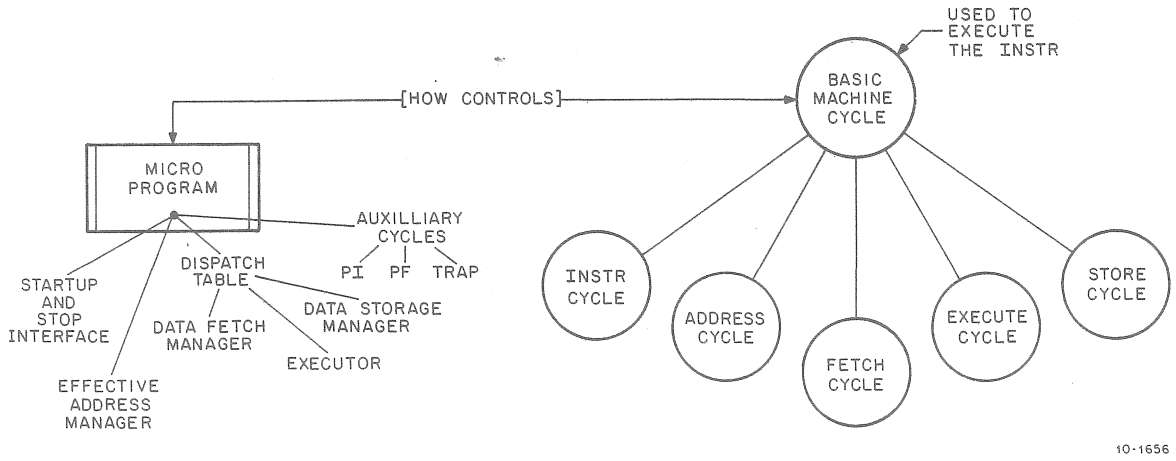



Figure 2-88 Basic Machine Cycle Summary

Figure 2-89 is the subcycle summary and Figure 2-90 is the hardware cycle summary.

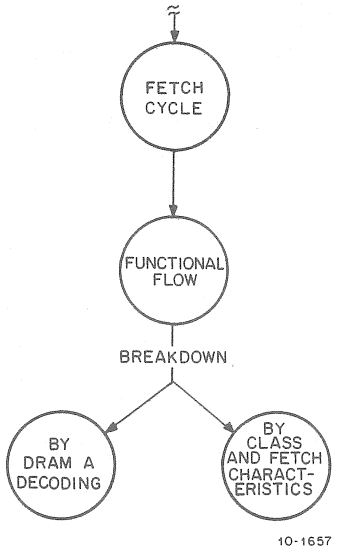



Figure 2-89 Subcycle Summary

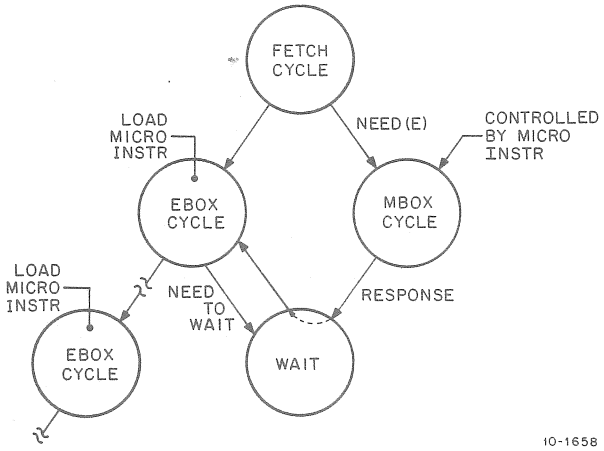



Figure 2-90 Hardware Cycle Summary

Next, the basic subcycle was presented in terms of a functional flow with additional graphics to support the description; in the interface section, the relationship of the hardware to the internal EBox cycles was described. These basic cycles were introduced in Subsection 2.1 as EBox, MBox, and EBus cycles. For example, the fetch cycle can be viewed as composed of a number of EBox and MBox cycles.

### 2.11.2 MBox Control

Referring to Figure 2-91, a number of functional elements work together to implement the basic MBox cycle. The grouping of the interface signals shown is as listed in Table 2-18.

To exercise the functional areas illustrated on Figure 2-91, a basic data fetch is covered in four steps. These steps are related to EBox timing in terms of occurrence.

**Table 2-18 Request Summary**

| Grouping | Signals |
|---|---|
| Basic EBox Request Handshake | EBOX REQUEST<br>CSH EBOX TO<br>CSH EBOX RETRY REQ<br>PF HOLD<br>MBOX RESPONSE IN |
| Address and Address Control | VMA 13–35<br>VMA AC REF |
| Timing | EBOX SYNC<br>MBOX CLOCK |
| Type Request | EBOX USER<br>EBOX READ<br>EBOX PSE<br>EBOX WRITE |
| Address Violation Logic | PAGE TEST PRIVATE<br>PT PUBLIC<br>PAGE ILLEGAL ENTRY<br>PAGE ADDRESS COND |

MBOX

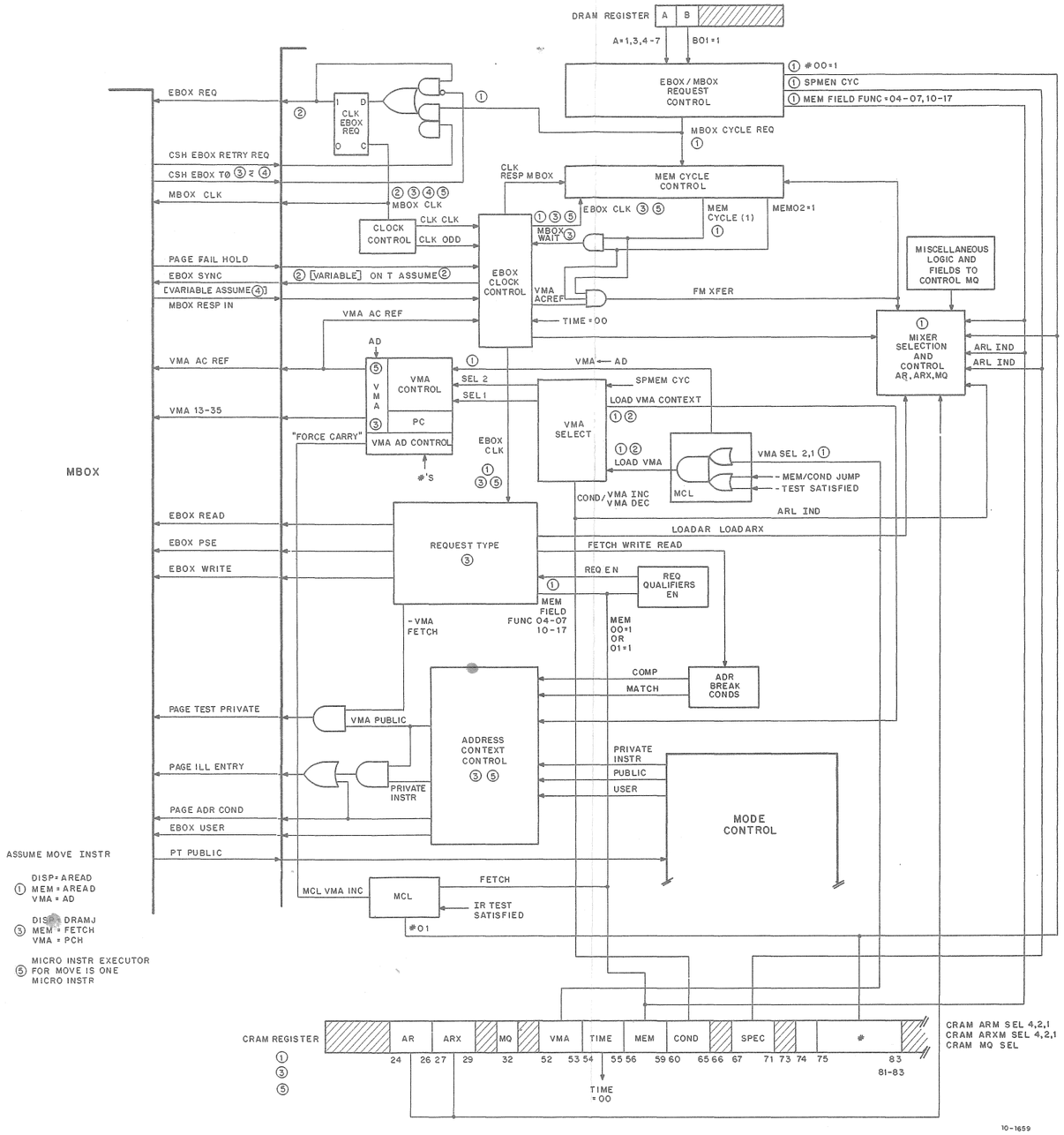



Figure 2-91 General Memory Request Control Simplified

**2.11.2.1 DATA FETCH REQUEST EN – Begin EBox Cycle (Figure 2-92)** – The flow is entered at an EBox clock and the CRAM register loads. The microinstruction begins to be decoded. Note that the MEM field is the major input to the MBox control logic. Assume that the effective address has been calculated, the MEM field is coded as AREAD, and the dispatch RAM A field is 5. In Figure 2-91 at ① , the MEM field function AREAD is a code of 4. This enables MBOX CYCLE REQ. In addition, if MEM 01 = 1, then REQ EN is asserted to enable the request qualifiers to be latched on the next EBox clock. MBOX CYCLE REQ enables the EBox request to be asserted on the next MBox clock. As indicated on the flow, this is a fast cycle. Two basic classes exist: fast and slow. The timing is illustrated in Figure 2-93.

Signal CLK SYNC EN must wait to occur, so that (for a fast cycle) EBOX SYNC sets at the same time as EBox request.

Referring to Figure 2-91, the VMA field, with other signals, enables LOAD VMA. In addition, the effective address must be input to VMA via AD so the VMA code (3) generates VMA ←AD.

The basic period between the leading edge of one EBox clock and the leading edge of the next is controlled by the T field of each microinstruction, along with certain other hardware signals. The basic pulse width of the positive EBox clock is fixed at 32 ns but the time between clocks is variable. EBOX SYNC occurs one MBox clock prior to the MBox clock that causes EBox clock to occur. The basic relationships are indicated in Figure 2-94.

**2.11.2.2 Begin MBox Cycle – End Current EBox Cycle and Start Next (Figure 2-95)** – As soon as SYNC EN is true, EBOX SYNC sets and MBOX CYCLE REQ (FAST CYCLE) enables EBox request to set (refer to ② on Figure 2-91). At this point, MBOX WAIT is tested and found clear. (This function is described in basic terms is Subsection 2.2.4.)

To summarize, the EBox request is then issued, and the VMA input mixer is set up and enabled to load with E via AD. The request type logic is enabled to assert the appropriate combination of EBox Read, PSE, and/or Write (which occur on the EBox clock to come at ③ ). In addition, the Address Context Control is enabling the proper combination of its qualifiers also to be asserted at ③.

Now another MBox clock occurs ③ ; simultaneously, an EBox clock occurs. The following actions result:

EBOX CLOCK ← 1
EBOX REQ ← 1 (REDUNDANT)
MEM CYCLE ← 1; MBOX WAIT ← 1
VMA LOADS AND LATCHES
CRAM ← NEXT MICRO INSTR
EBOX QUALIFIERS LATCHED

Thus, we have passed through one EBox cycle and now reenter the flow to begin a second EBox cycle.

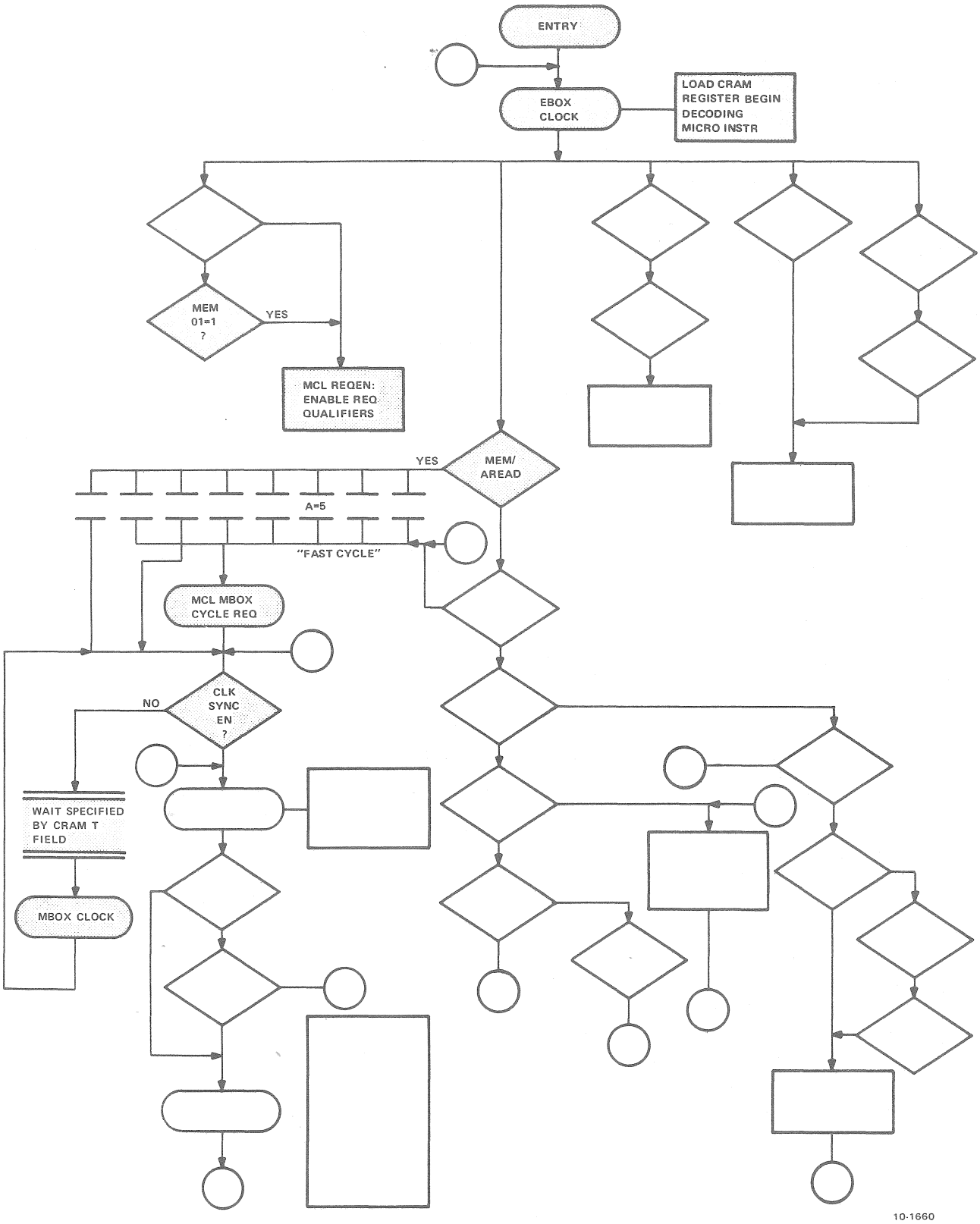



Figure 2-92 Begin EBox Cycle Data Fetch Request

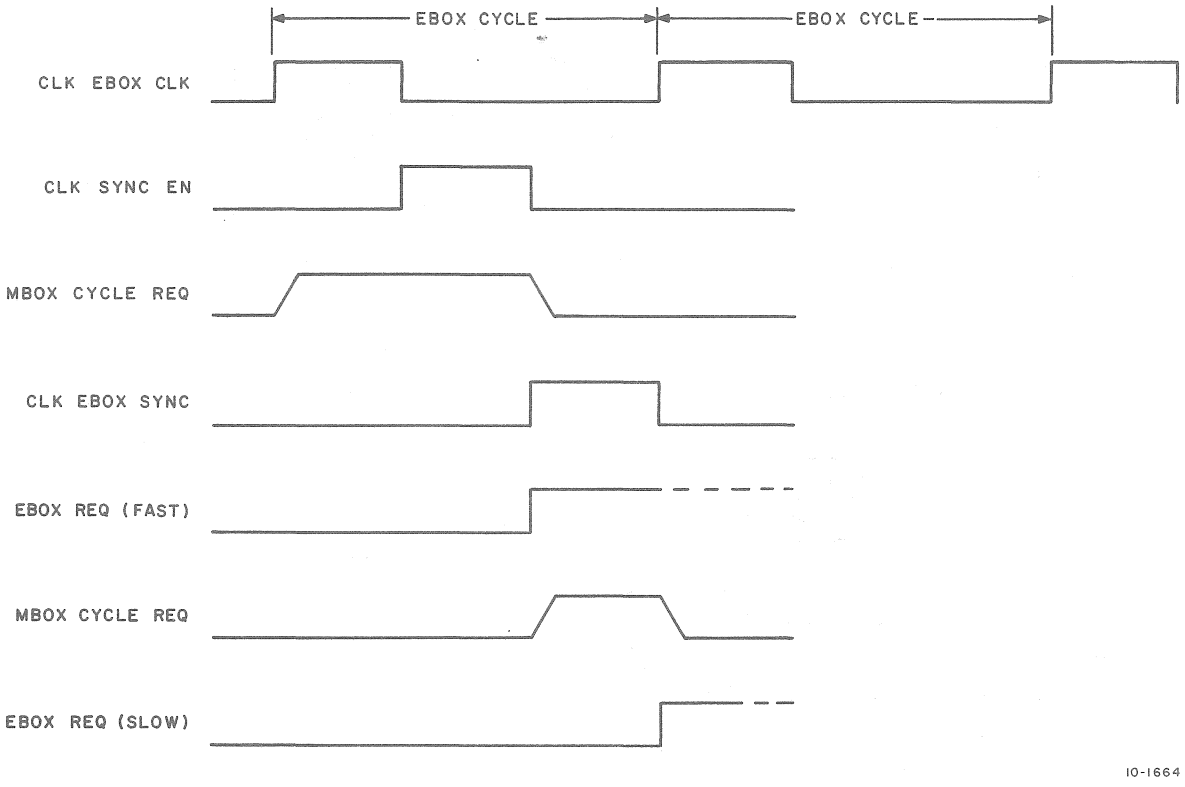



Figure 2-93 EBox Request Fast or Slow

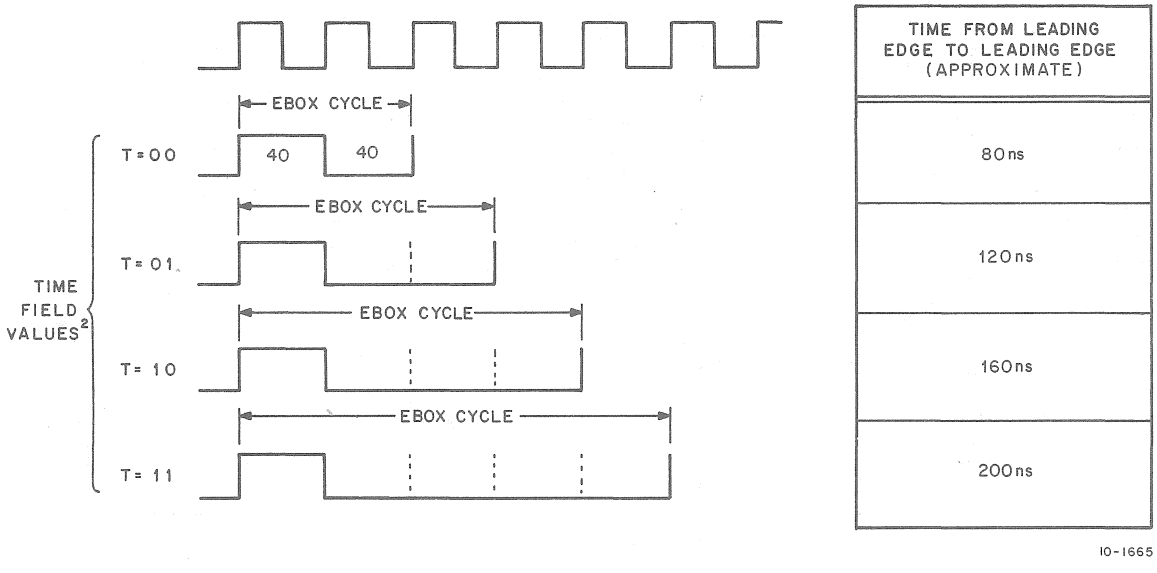



Figure 2-94 Basic EBox Clock Period

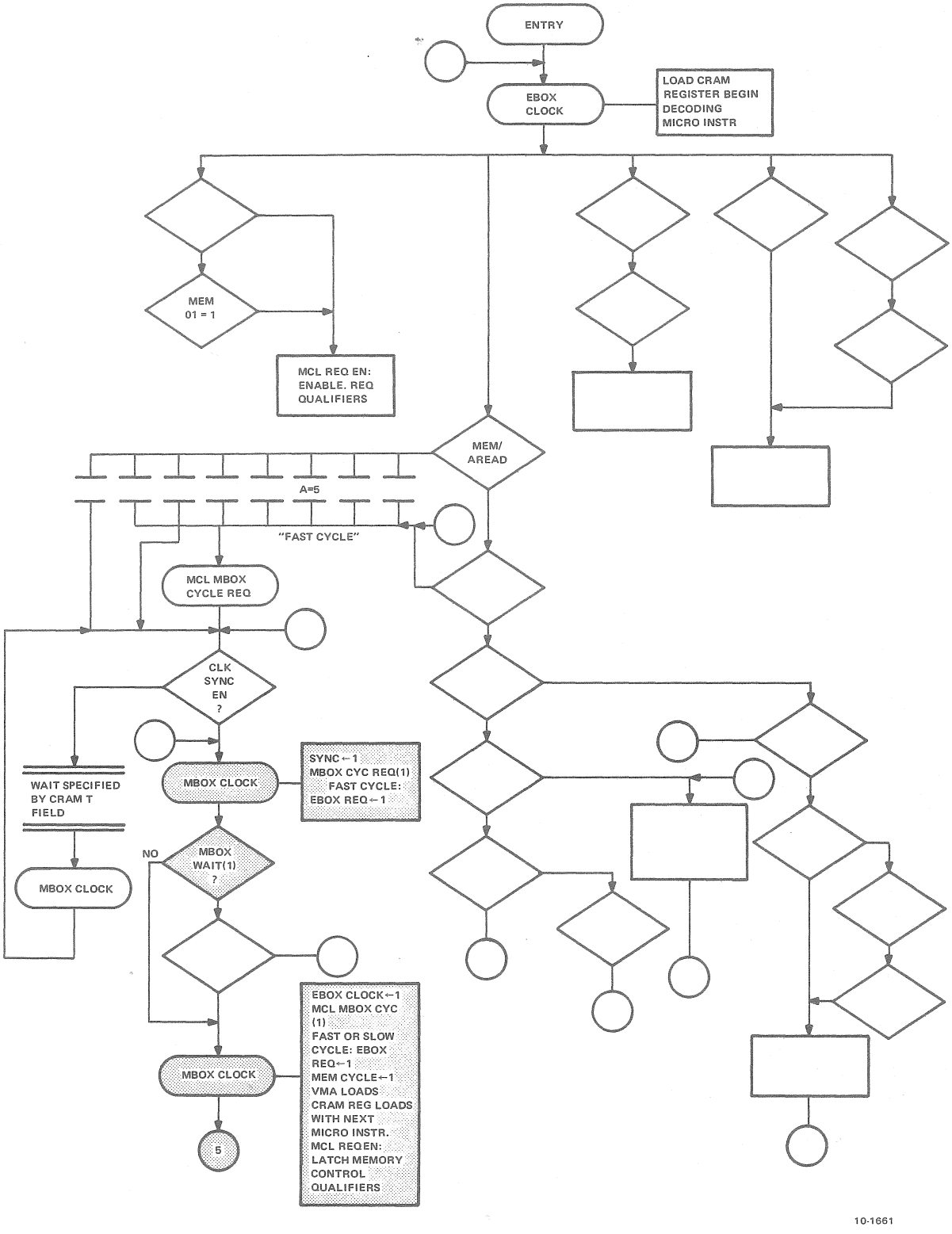



Figure 2-95 Begin MBox Cycle, End Current EBox Cycle, Begin Next EBox Cycle

**2.11.2.3 SETUP PREFETCH – Wait for MBox Response** – Referring to Figure 2-96, the flow is reentered at ⑤ where the EBox clock generated loads the second microinstruction (Figure 2-91 ③ ). Now the MEM field function is FETCH and MEM 02 = 1. If the MBox has not responded with the word requested (E), MEM cycle is still set. The combination of MEM 02 (1) and MEM Cycle (1) generates MBOX WAIT. Providing that the request is not to fast memory, the EBox stops until the MBox response occurs.

This is true whether a page fault occurs or not, although PF hold is asserted 5 MBox clocks before MBOX RESPONSE is asserted when a page fault has occurred. In this example, assume that the MBox is working on the request, but has not yet responded.

Referring to the flow (Figure 2-96), the current microinstruction MEM field function fetch is a code of 6. Note, however, that because a priority interrupt takes precedence over any other activity, PI CYCLE is checked before enabling the MCL MBOX CYCLE REQ. Here PI CYCLE is clear, so ② points to a "Fast Request." Again, a wait for SYNC EN, as defined by the T field, takes place. The state of the SYNC EN during MBOX WAIT is always true; this keeps EBOX SYNC true until the response is received.

The MBox continues to run during the waiting period. Thus, MBOX CLOCK sets EBOX REQUEST even though the VMA is still latched up with E. During the waiting period, the VMA input receives PC+1 via VMA AD.

The EBox now loops, waiting for MBOX RESPONSE to restart the EBox clock.

**2.11.2.4 MBOX RESPONSE RECEIVED** – Referring to Figure 2-97, MBOX RESPONSE enables the EBox clock. Thus, EBOX CLOCK becomes true and, simultaneously, EBOX SYNC becomes false. The third microinstruction is now loaded into the CRAM register (Figure 2-91 ⑤ ) and is decoded. In addition, the VMA is loaded and latched with PC+1, the request qualifiers are latched and now, with the requested data word in AR, a DRAM J dispatch is issued.

**2.11.2.5 General Memory Cycle Control** – Figure 2-98 contains all combinations of the MEM field that can generate MCL MBOX CYCLE, and hence EBOX REQ. In general, the following functions are of the "Slow Cycle" type:

B WRITE
PI FETCHES
SKIP SATISFIED FETCHES
REG FUNCTIONS
SP MEM CYCLES

A Slow cycle is required during MEM/REG FUNC because the MBox requires additional time to decode the type of request. In all the "slow" cycle types, the EBOX does not necessarily have time to determine whether to make the request (or not) before EBOX SYNC. Thus, the decision, and therefore the request, is delayed purely for hardware timing reasons.

## 2.12 EBUS INTERFACE CONTROL

The I/O system for the KL10 processor includes the EBus, the peripheral equipment with its interfaces to the EBus, and various control logic. The EBus interface may be controlled either by the EBox during input or output instruction execution, or by the PI system during priority interrupt handling. Subsection 2.8.1 gives a basic summary of the EBus signals. This is followed by a functional description of the interface, which is covered at two levels. The first level describes the basic functional organization and operation of the PI board and other related logic. The second description deals with the microprogram to PI board interfacing. This description attempts to give insight into the manner in which the hardware and the microprogram interact to carry out various interface related functions.

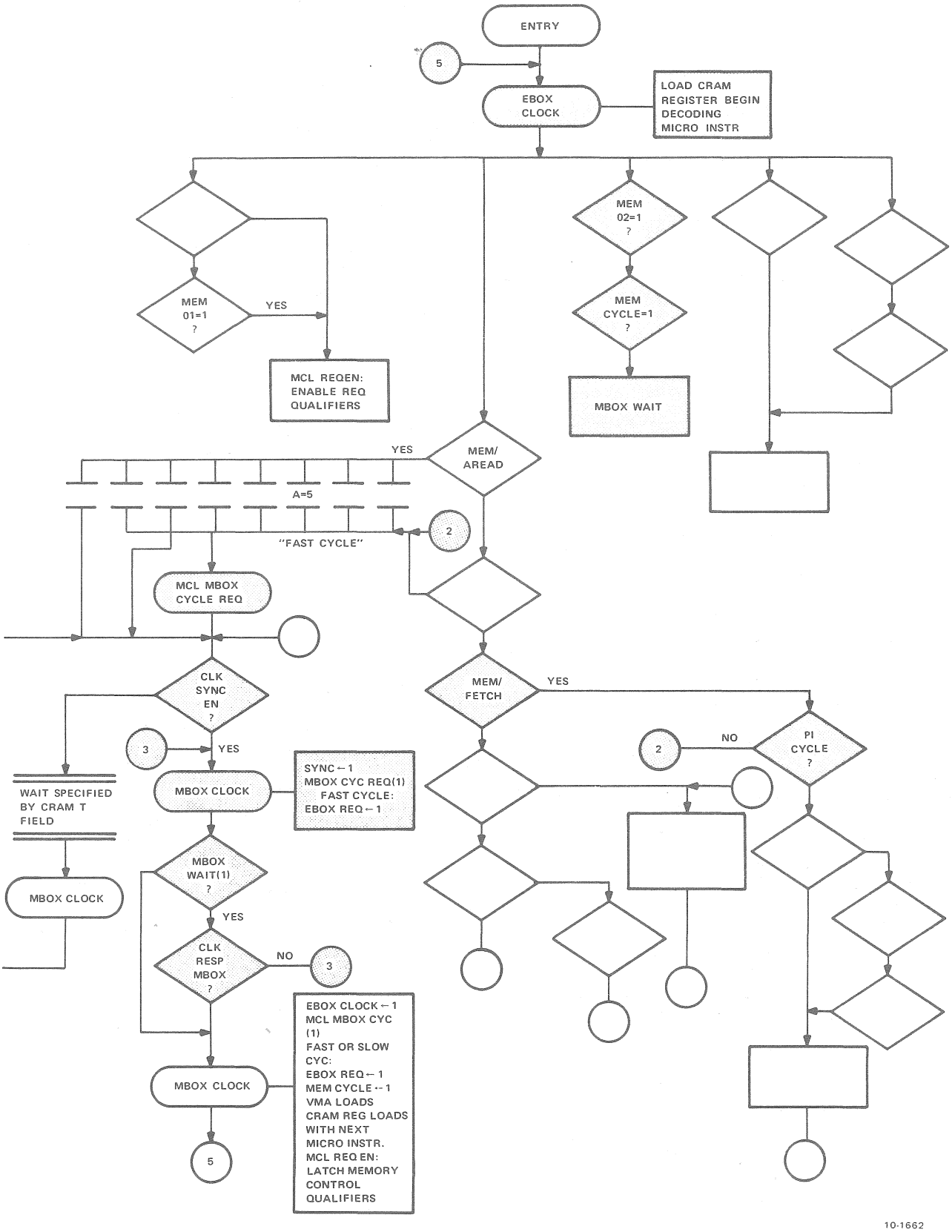



Figure 2-96 Setup Prefetch Waiting for MBox Response

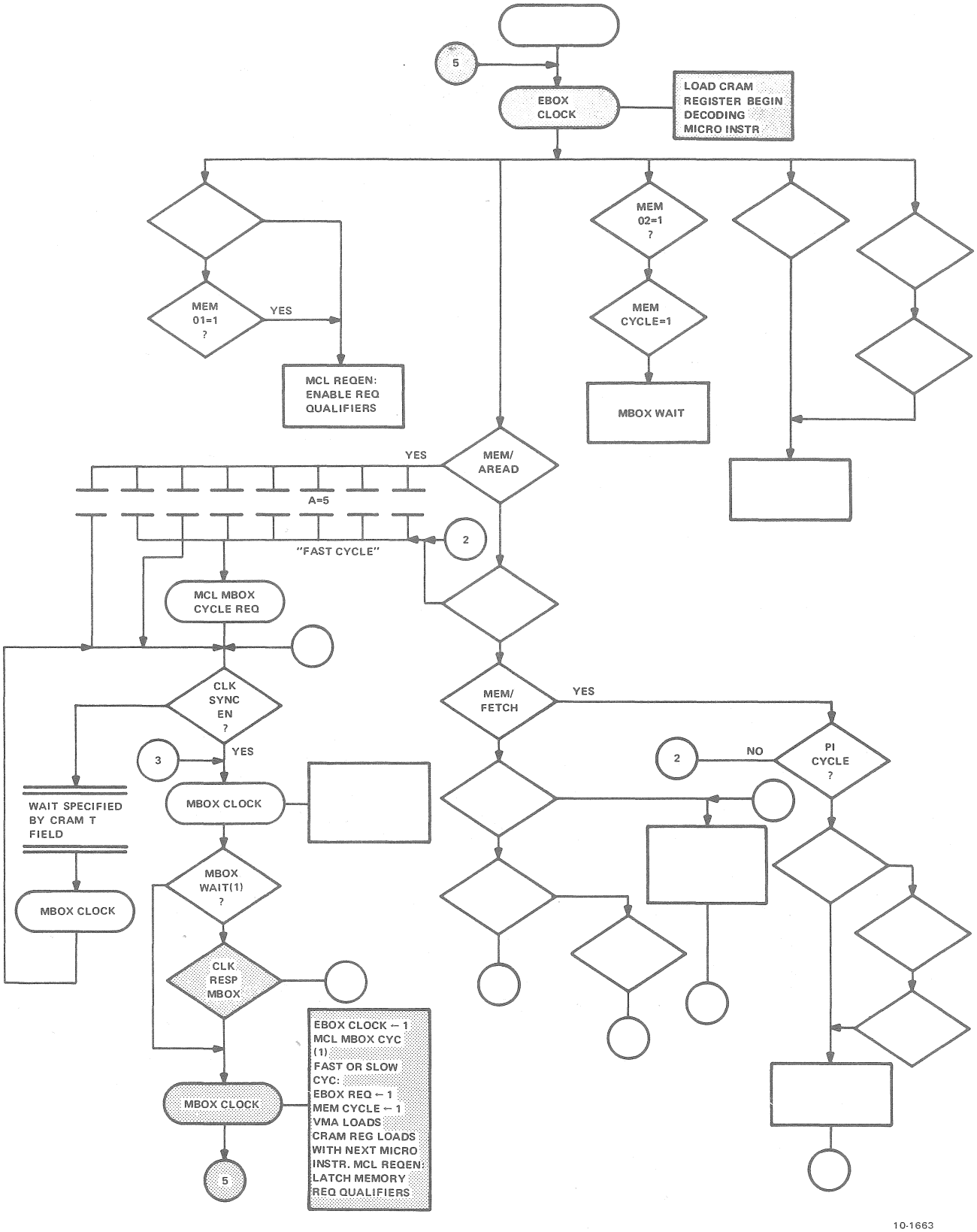



Figure 2-97 Receive MBox Response, End Current MBox Cycle, End Current EBox Cycle, Begin Next EBox Cycle, Begin MBox Cycle

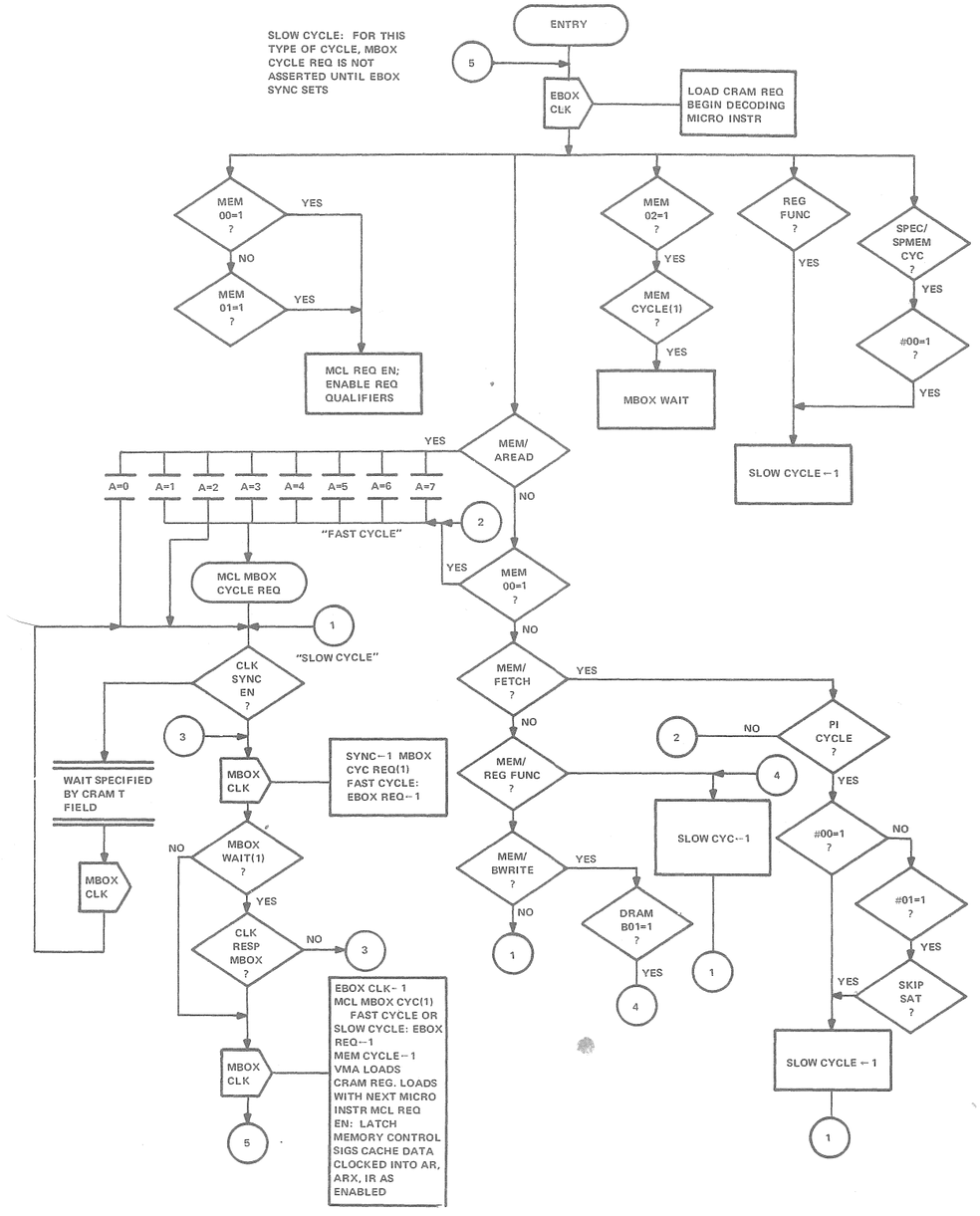




Figure 2-98 General Memory Cycle Control Flow

### 2.12.1 EBus Signal Lines

The EBus consists of 60 signals. All devices, including the KL10, are connected to these lines in parallel. The bidirectional nature of 36 of the signals permits some information to flow in both directions. These lines are the data lines. The remaining 24 signals are used for control functions. Table 2-19 lists the data transfer signals.

**Table 2-19 Data Transfer Signals**

| Name | Mnemonic | Number of Lines |
|---|---|---|
| Data | D(00:35) | 36 |
| Controller Select | CS(00:06) | 7 |
| Function | F(00:02) | 3 |
| Demand | DEM | 1 |
| Acknowledge | ACK | 1 |
| Transfer | XFER | 1 |

**DATA LINES D(00:35)** – The 36-data lines transfer information between the EBox and its devices. The most significant bit is bit 00; the least significant bit is bit 35.

**CONTROLLER SELECT LINES CS(00:06)** – These seven lines select the desired controller for a data transfer. Each controller has a unique select code hardwired on the backplane of the device.

**FUNCTION LINES F(00:02)** – The function lines specify the type of data transfer (or non data transfer) to take place. Table 2-20 lists the functions implemented.

**Table 2-20 Table Data Transfer Commands**

| F00 | F01 | F02 | Operation |
|---|---|---|---|
| 0 | 0 | 0 | CONO |
| 0 | 0 | 1 | CONI |
| 0 | 1 | 0 | DATAO |
| 0 | 1 | 1 | DATAI |

**DEMAND (DEM)** – This line causes the addressed controller to inspect the CS and F lines and decode their meaning. Upon implementing the specified function, Transfer and Acknowledge are asserted in response and data is placed onto or taken from the EBus as specified by the decoded function.

**ACKNOWLEDGE (ACK)** – This signal line notifies the I/O bus adapter not to respond to the current operation. If it does not detect ACKNOWLEDGE within some period following assertion of DEMAND, it attempts to perform the transfer. It does not decode the CS lines as the standard KL10 devices do.

**TRANSFER** – This line is asserted by the selected controller when it is ready to execute the specified function as decoded in F(00:02).

**PRIORITY TRANSFER LINES** – To perform priority interrupts between the KL10 and its devices, the same basic set of signals is used in a slightly modified form. Table 2-21 lists the necessary signals as they are used.

**Table 2-21 Priority Transfer Signals**

| Name | Mnemonic | Number of Lines |
|---|---|---|
| Controller Select | CS(04:06) | 3 |
| Controller Select | CS(00:03) | 4 |
| Function | F(00:02) | 3 |
| Demand | DEM | 1 |
| Acknowledge | ACK | 1 |
| Transfer | XFER | 1 |

**CONTROLLER SEL CS (04:06)** – During interrupt arbitration, these three lines represent the octal encode of the interrupting channel.

**CONTROLLER SEL CS(00:03)** – These four lines specify the controller or device that the EBox is to honor during this interrupt sequence. This is, of course, only a single device or controller, even though several may be interrupting on the same channel. This code also corresponds to the hardwired physical device number of the appropriate controller or device. In CONTROLLER SEL CS(00:03), the range is 0 through 17.

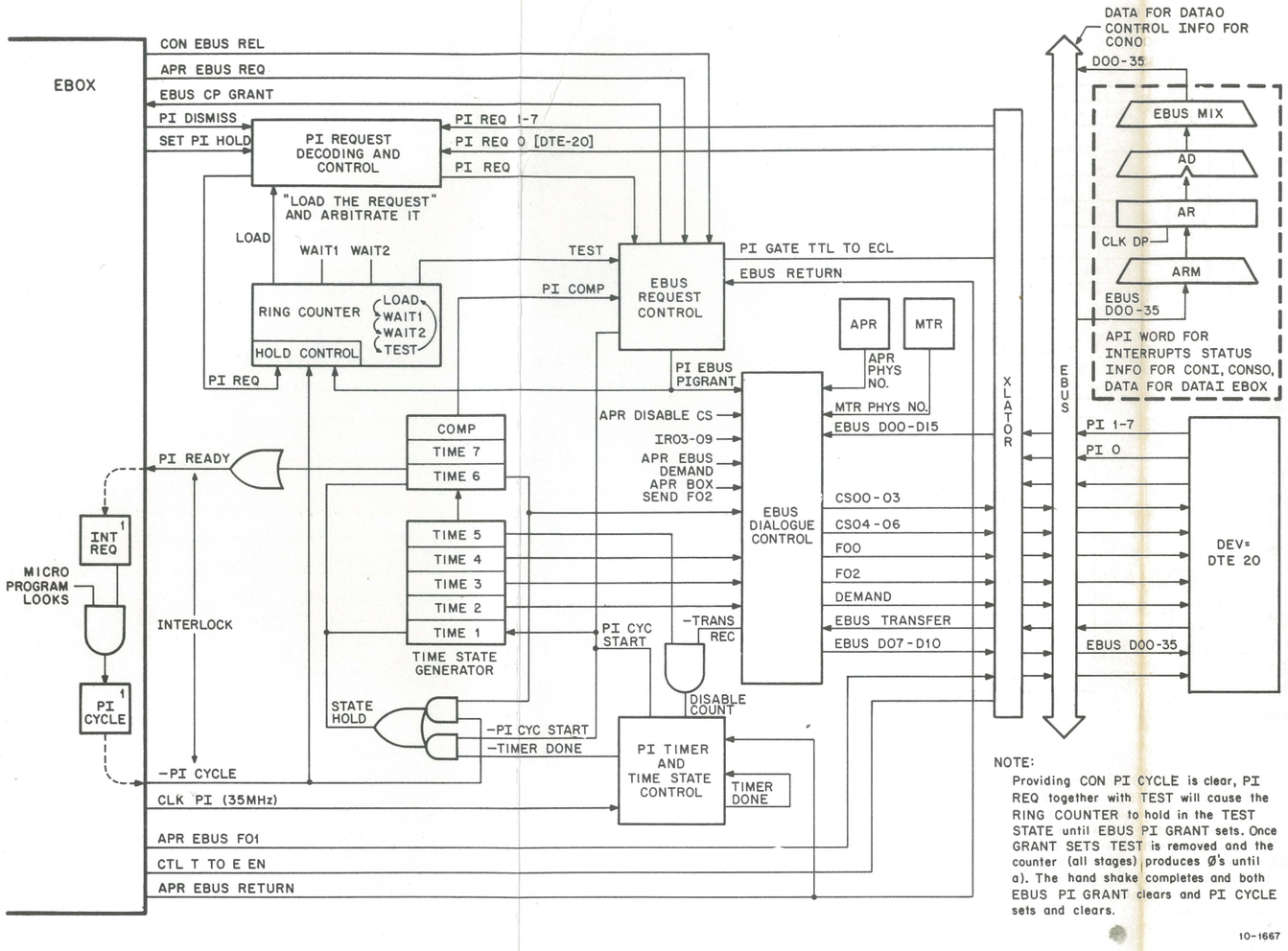
**FUNCTION F(00:02)** – Two functions are generated during the interrupt dialogue. The first is a code of 4 in F(00:02) and specifies to the interrupting controllers that those controllers being addressed by Channel number in CS(04:06) should send their Physical Controller number by placing them onto the EBus upon sensng DEMAND. The second function is a code of 5 in F(00:02) and specifies to the interrupting controllers or devices that one has been selected. The selected controller will see CS(00:03) as the same number as its physical controller number.

**ACKNOWLEDGE (ACK)** – Same as for data transfers.

**TRANSFER (XFER)** – In the case of interrupts, the device selected for service by the EBox places a special function on the EBus data lines D(00:35). Figure 2-99 is the EBus interface functional block diagram. Table 2-22 lists the priority transfer commands.

**Table 2-22 Priority Transfer Commands**

| F00 | F01 | F02 | Operation |
|---|---|---|---|
| 1 | 0 | 0 | PI SERVED |
| 1 | 0 | 1 | PI ADDRESS IN |




NOTE:
Providing CON PI CYCLE is clear, PI REQ together with TEST will cause the RING COUNTER to hold in the TEST STATE until EBUS PI GRANT sets. Once GRANT SETS TEST is removed and the counter (all stages) produces Ø's until a). The hand shake completes and both EBUS PI GRANT clears and PI CYCLE sets and clears.

10-1667

Figure 2-99 EBus Interface Functional Block Diagram

### 2.12.2 EBus Interface Organization

Referring to Figure 2-99, the interface consists basically of six functional elements. These elements are as follows:

1. PI Request Decoding and Control
2. PI Request Counter and Control
3. EBus Request and Control
4. EBus Dialogue Control
5. PI Timer and Time State Control
6. Time State Generator

The EBus request control and EBus dialogue control are used both by the EBox to carry out I/O transfers and by the PI system in response to an interrupt. During priority interrupt handling, the EBus dialogue is carried out in asynchronous fashion. This operation is controlled by the PI timer and time state control, together with the time state generator.

To obtain the use of the EBus dialogue control, the PI request decoding and control logic must compete with the EBox. No priority exists, and control is obtained on a first-come, first-served basis. Once the EBus has been granted to the EBox, the priority interrupt must wait until the EBox releases the bus.

If the PI system obtains the EBus, the EBox may "demand" the EBus if a page fault occurs (EBus Return).

### 2.12.3 Interrupt Handling – Loading the Request

Referring to Figure 2-99, there are two cases. The first is an interrupt request from some device on PI 1–7. This may be from any KL10 device, including the APR. The second case is an interrupt from the DTE20 on channel 0. Only the DTE20 may generate channel 0 interrupt requests.

In either case, the PI request enters the PI request decoding and control logic. Here there is a variation in priority. The PI system must be turned on in order for a request on channel 1–7 to be inspected, while interrupts on channel 0 will always be inspected whether the PI system is on or off. The ring counter controls the sampling of PI requests and also determines when a particular request (the highest) is ready to be serviced. In general, "PI LOAD" enables all active requests 0–7 into a request register, providing corresponding PI ON enables are on for channels 1–7.

A programmer may disable interrupts on selected channels by clearing PI ON for each channel he desires to inhibit (note PION0 is in the DTE20). This is done by performing a CONO PI instruction. While the ring counter advances through "WAIT 1" and "WAIT 2," the priority network arbitrates all incoming priority interrupt levels and selects the one with the highest priority (numerically lowest number).

**2.12.3.1 Testing the Request** – Next, PI TEST is asserted with PI REQ to request the EBus. PI TEST remains true until EBUS PI GRANT sets, giving the EBus to the PI system. Once PI GRANT sets, the PI TEST condition is cleared and the ring counter is disabled until the entire EBus dialogue is carried out and PI CYCLE is "set and cleared" by the microprogram.

**2.12.3.2 Requesting the EBus** – Setting EBUS PI GRANT begins the EBus dialogue by enabling the assertion of CS 04–06 as the selected channel and F00(4) as function PI SERVED, and also causes the PI timer to begin its sequence by setting PI CYC START.

In general, all external devices that connect to the EBus are presumed to be composed of TTL logic. The PI and EBox logic consist of ECL logic. To temporarily connect these two different types of logic requires use of a logic level shifter. This device is called a translator. The translator must be notified of the conversion direction, TTL to ECL or ECL to TTL. Actually, only the data portion of the EBus is switched from one level to the other. The control signals are connected to fixed level shifting logic. For example, EBUS DEMAND is a unidirectional signal and it is connected to a noncontrollable level shifting gate on the translator module (ECL to TTL).

**2.12.3.3 Beginning the Dialogue** – The setting of PI EBUS PI GRANT asserts the level PI GATE TTL TO ECL, which causes translation of incoming data from TTL logic levels to ECL logic levels. The PI timer and time state control manipulates the time state generator such that each time state is held for the appropriate length of time. The following relationships exist between the dialogue signals and the time state logic:

CSH 04–06: EBUS PI GRANT
F00: EBUS PI GRANT
DEMAND: sent at T2, T5, and T6
LATCH INCOMING PHYS numbers: T3
CS00–03: T3
F02: T4
EBUS TRANSFER: WAIT AT T5 FOR TRANSFER
PI CYCLE: WAIT AT T6 FOR PI CYCLE TO SET

**2.12.3.4 Interlocks and Dialogue Completion** – Upon entering T5, the timer is inhibited from incrementing the count until EBUS TRANSFER is received or forced. While waiting, the timer holds the loaded count. As soon as TRANSFER is received and recognized by the PI logic, the timer is once again allowed to count down T5.

Thus, while T5 is counted down, the API word is stabilizing on the input to AR. Next, T6 is entered and here the absence of PI cycle causes STATE HOLD to be asserted. This time the timer may count down and even generate TIMER DONE. If this point is reached and PI CYCLE is still false, the timer loads the count specified by T6 and continues to count while waiting for PI CYCLE to set. The PI board must not begin to service a second interrupt before the microprogram has a chance to look at the first one. Hence, the timer is prevented from entering T7 COMP, until the microprogram has set PI CYCLE. This also enables the ring counter to perform load.

Assuming PI CYCLE sets, the time state generator proceeds through T7 and into complete (COMP). Note that the EBus dialogue control removes DEMAND some time before removing the CS and F lines. This avoids the possibility of misselection of a device. The generation of COMP enables PI EBUS PI GRANT to clear, removing F00 and CS04–06.

### 2.12.4 Basic Input Output Control

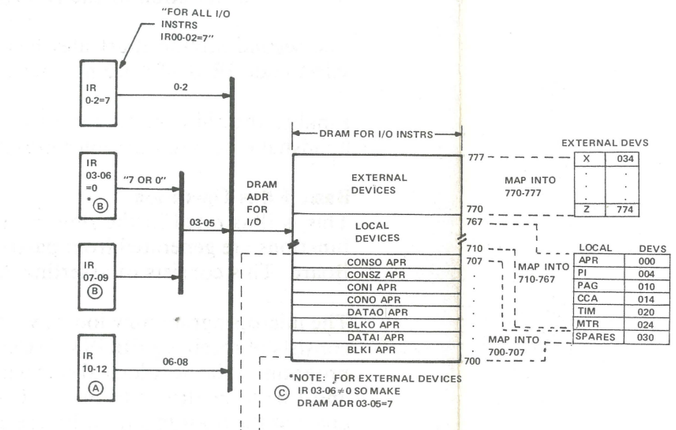
Referring to Figure 2-99, the implementation of I/O operations is similar to interrupt processing, if taken at the point where the EBus is requested. The difference is that instead of a hardware arbitration process taking place, followed by a single request subsequently asking for the EBus, the microprogram I/O handler (part of the executor) requests the EBus. This is accomplished utilizing the condition field function COND/EBUS CTL, together with a particular pattern in the magic number field all in the same microinstruction. Only the resulting signal is indicated on the figure (APR EBUS REQ) but the various other signals are simply formed by combinations of COND/EBUS CTL and an appropriate magic number.

**2.12.4.1 Requesting the EBus** – The EBus request control treats both an EBox-EBus request (APR EBUS REQ) and a PI EBus request equally. Whichever request is seen by the EBus request control first receives the EBus.

The microprogram is waiting for an indication that it has been granted the EBus. The indication of this condition is EBUS CP GRANT. The microprogram loops, waiting for this signal to become true. Once this occurs, the next step in the operation may be performed.

**2.12.4.2 Dialogue Overview** – Basically, the EBox decodes bits 10–12 of the instruction to determine which type of I/O operation is to be performed. Eight possible combinations exist; these are indicated in Figure 2-100 at the bottom left. The logical mapping of I/O op code into appropriate DRAM addresses is also illustrated in Figure 2-100.

| | OPERATION | COND/ EBUS CTL | MAGIC # FIELD 0 1 2 3 4 5 6 7 8 | FUNCTION | #0 APR EBUS RETURN | #1 APR EBUS REQ | #2 CON EBUS REL | #3 APR EBUS DEMAND | #4 HOLD STATE | #5 (0) #5 (1) SELECTION CONTROL | #6 #6 APR EBOX DISABLE CS | – APR AC10 (1) #7 APR EBUS F01 | APR F02 EN (1) #8 APR EBOX SEND F02 | CS 00–06 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | BASIC EBUS OPERATION USED BY ALL I/O INSTR | YES | 0 1 0 0 0 0 0 0 0 | REQUEST EBUS | | X | | | | | | | | |
| | | YES | 0 0 0 0 1 1 0 0 0 | I/O INIT | | | | | X | X | | X | X | IR 03–09 |
| | | YES | 0 0 0 1 1 0 0 0 0 | SET EBUS DEMAND | | | | X | X | | | Holding | Holding | Holding |
| | | YES | 0 0 0 0 1 0 0 0 0 | CLEAR EBUS DEMAND | | | | | X | | | Holding | Holding | Holding |
| | | YES | 0 0 1 0 0 0 0 0 0 | RELEASE EBUS | | | X | | | | | | | |
| MTR INT, INTERNAL DEVICE CONTROL, PAGE FAIL HANDLER, READ EBUS REG | USED TO OBTAIN ONLY THE ECL SIDE OF EBUS. THEN GIVE BACK LATER | YES | 1 0 0 0 0 0 0 0 0 | GRAB ECL EBUS | X | | | | | | | | | |
| | | YES | 0 0 0 0 0 0 0 0 0 | RELEASE ECL EBUS | | | | | | | | | | |
| | DEPOSIT, BYTE XFER OR PI DATAO, FOLLOWED BY BASIC EBUS OPERATION | YES | 0 0 0 0 1 0 1 1 0 | SET DATAO | | | | | X | | X | X | | 0'S |
| | EXAMINE, BYTE XFER OR PI DATAI, FOLLOWED BY BASIC EBUS OPERATION | YES | 0 0 0 0 1 0 1 0 0 | SET DATAI | | | | | X | | X | | | 0'S |




| OPERATION | AC10 | AC11 | AC12 | F00 | F01 | F02 |
|---|---|---|---|---|---|---|
| CONO | 1 | 0 | 0 | 0 | 0 | 0 |
| CONI | 1 | 0 | 1 | 0 | 0 | 1 |
| DATAO | 0* | 1 | 1 | 0 | 1* | 0 |
| DATAI | 0* | 0 | 1 | 0 | 1* | 1 |
| BLKO | 0* | 1 | 0 | 0 | 1* | 0 |
| BLKI | 0* | 0 | 0 | 0 | 1* | 1 |
| CONSO | 1 | 1 | 1 | 0 | 0 | 1 |
| CONSZ | 1 | 1 | 0 | 0 | 0 | 1 |

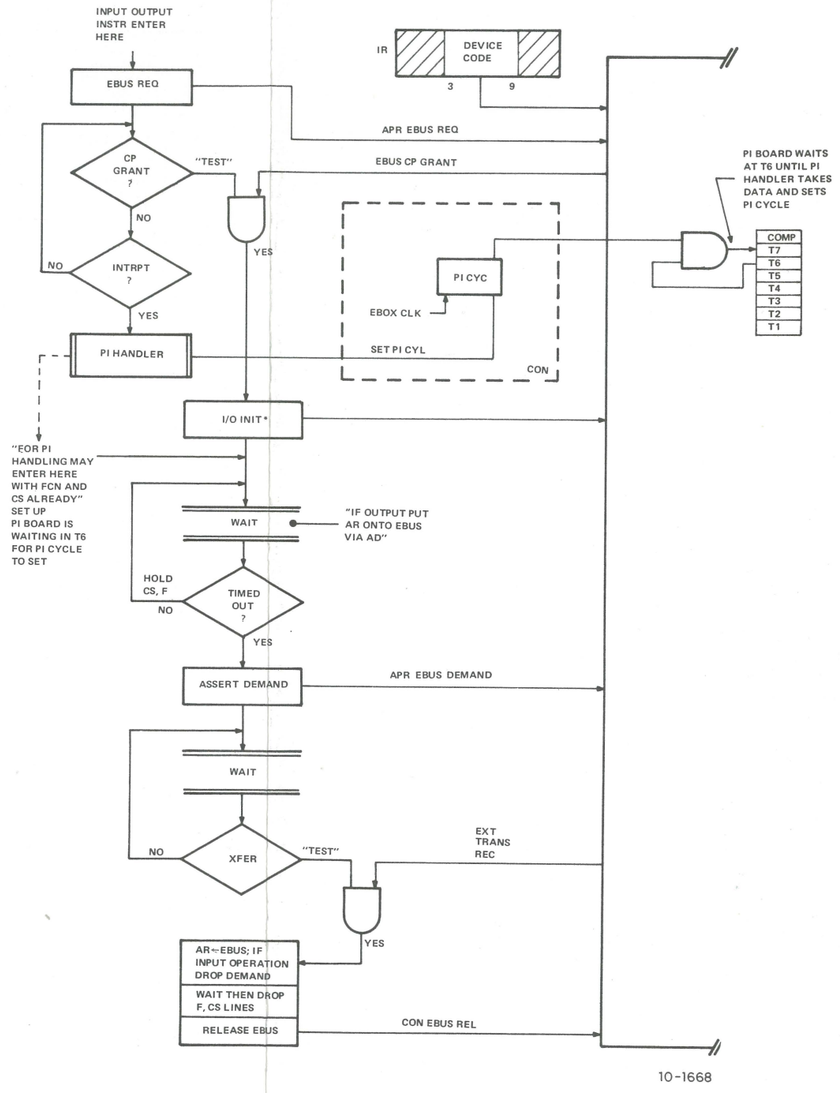



Figure 2-100 EBus Control Functions

The dispatch to the proper operation is obtained by mapping bits 10–12 into DRAM ADR 06–08, while the device address 3–6 is mapped into DRAM ADR bits 03-05. Thus, for example, a DATAI APR with op code 701 is mapped into DRAM address 701. Similarly, BLKO PAG, with op code 722 is mapped into DRAM address 722. This is device $010_8$; therefore, the type of operation performed is determined in advance and the DRAM jump address is coded to cause a jump to the appropriate group of microinstructions. The device select code is in bits 3–9 of IR and must be used to address the device. This addressing is accomplished by converting 3–9 to CS00–06 in the proper form. The function is controlled by the combination of two EBox control signals, APR EBOX SEND F02 and APR EBUS F01. With these two signals, all combinations of input and output operations may be performed as indicated on Figure 2-100. Notice that EBus F00 is not used for any of the operations. This signal is generated during priority interrupt dialogue for the function PI SERVED (Function 4) and for PI ADDRESS IN (Function 5).

**2.12.4.3 Functional Breakdown** – Figure 2-100 is essentially composed of three sections. The first is a breakdown of the EBus microcode operations into four basic suboperations as follows:

1. Basic EBus operation as used by all I/O instructions.
2. ECL EBus acquisition and subsequent release
3. Generation of the DATAO function followed by the basic EBUS operation
4. Generation of the DATAI function followed by the basic EBus operation

The second section illustrates how the operation specified in IR 10–12 and a portion of the device select code IR 03–05 are mapped into the DRAM words that pertain to I/O operations.

Finally, the third section consists of a simplified flow of the basic EBus operation, including the handshake between the microprogram EBus driver and the PI Board.

**Basic EBus Operation**
This is illustrated in the flow on the bottom right of Figure 2-100. Five basic COND/EBUS CTL functions are generated from particular magic number bits. The first is to request the EBus from the PI Board. This consists of asserting APR EBUS REQ.

The microprogram now loops, waiting for an indication that it has obtained the EBus. The indication consists of receiving EBUS CP (Central Processor) GRANT from the PI Board. This moves the microprogram to the next logical step which is IO INIT. Here magic number 5 enables the function lines F01 and F02 to be driven from -APR AC10 and APR F02 EN, respectively. The table of I/O operations given at the bottom left on Figure 2-100 shows that F01 is true whenever AC10 is false. This is true for DATAO, DATAI, BLKO, and BLKI. Conversely, F02 is true whenever AC10 is true, or both AC10 and AC11 are false.

Magic number 4 is used to latch the particular function (HOLD IT). Note that during the IO INIT period, IR 03–09 is passed to the PI Board to become CS00–06. A fixed delay is generated by the microcode at this time to allow the controller select lines to set up at the device.

Next, SET EBUS DEMAND is issued, while holding the previous function lines F01 and F02 as previously set up. Once again, the microprogram waits a predetermined period. The waiting is controlled by the time field and the number of successive microinstructions issued. Thus, two successive microinstructions with T = 5 is approximately 300 ns.

Now the microprogram loops, waiting for TRANSFER from the device. This signal indicates that the device has completed the specified transaction and has either taken or transmitted status, data, or control over the EBus. At this time, if the operation was CONSO, CONSZ, CONI, BLKI or DATAI, the EBus is loaded into AR. If the operation was CONO, BLKO or DATAO, during IO INIT the AD is enabled to the EBus. The AD contains the contents of AR.

Finally, DEMAND is removed by issuing the function CLR EBUS DEMAND. Notice that number 4 holds the function lines up. It is necessary to remove DEMAND first while still maintaining the function and CS lines in order to prevent a spurious misselection. Now the function and CS lines are dropped and the EBus is relinquished by issuing RELEASE EBUS. This action causes EBUS CP GRANT to clear.

**PI Handler and EBus Operation**
Once again referring to the flow on Figure 2-100, note that after issuing EBUS REQUEST and while testing for CP GRANT, an interrupt is tested for. If an interrupt is pending, the PI Handler is entered. This means that EBUS PI GRANT was set when EBUS REQUEST was issued and EBUS CP GRANT could not set anyway.

The PI Board has negotiated with the device for the API function word that is now on the input to AR. The PI Board is holding in T6, waiting for PI cycle to be set.

Examine, Deposit, or Byte transfers requested by the 10-11 interface require separate control of the controller select and function lines. For these cases, SET DATAO or SET DATAI is issued independently. Then the EBus routine is entered at the point where the CS and F lines are setting up. If the operation is DATAO of T011 transfer, the AR is placed onto the EBus via AD. The remainder of the EBus operation is identical to that for basic EBus operation.

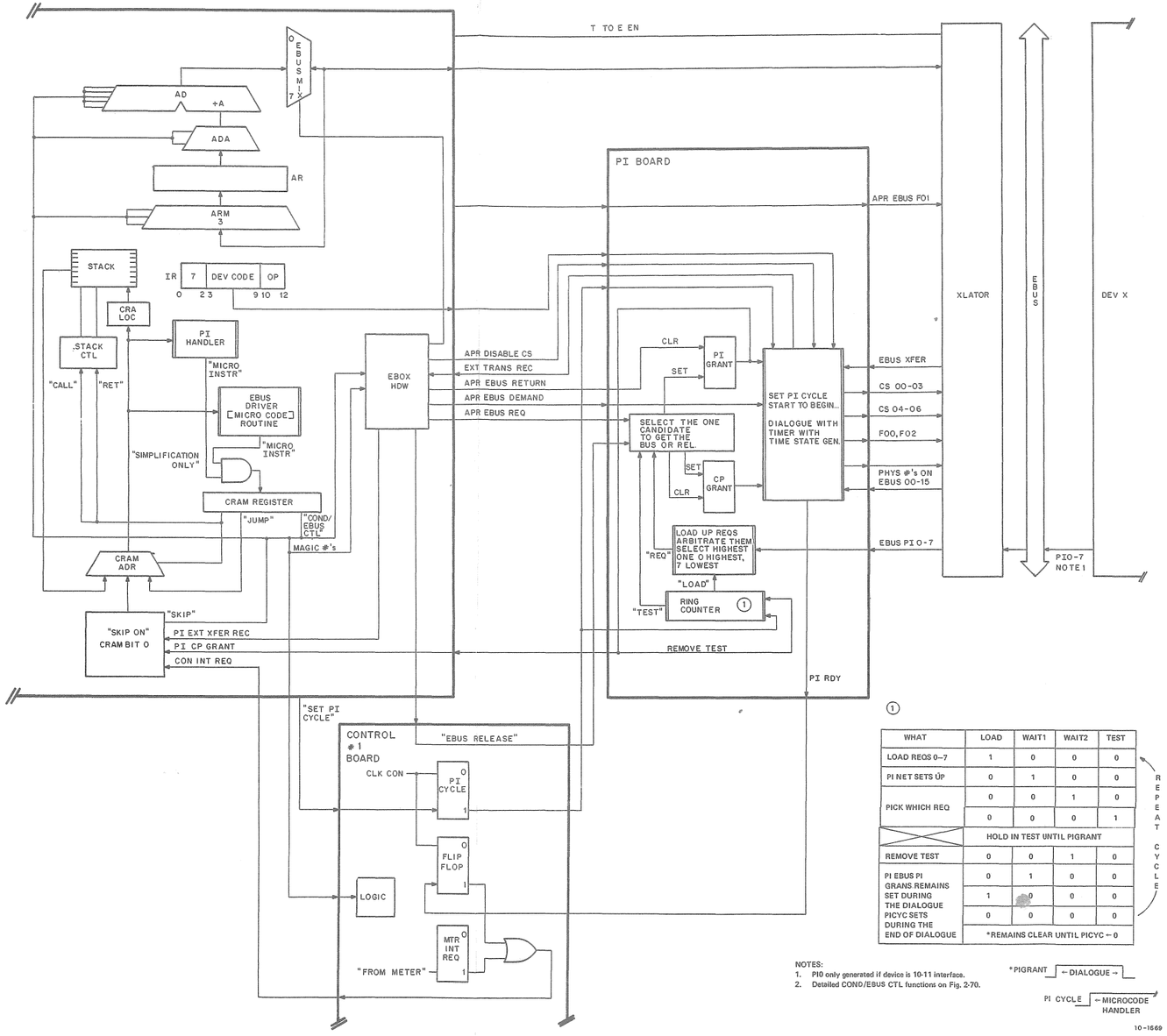
**ECL EBus Acquisition** – At various times, the ECL portion of the EBus is required for some form of transfer. Some examples of this requirement would be processing interrupts for internal devices such as APR, PI SYSTEM, or TIM. Also, performing I/O instructions involving these devices would require the use of the ECL EBus. A second example is the case of page fault handling in the microcode. At some time, the MBox-EBus register must be read over the EBus into AR. Thus, the ECL EBus is necessary for this operation. The function necessary to acquire the ECL EBus is COND/EBUS CTL with magic number bit 0 set. This actually takes the EBus away from the PI system. It does not abort the PI operation (if any) but merely causes it to be dealyed. The signal APR EBUS RETURN causes the PI timer and time state generator to HOLD and it clears EBUS PI GRANT. The ECL EBus is relinquished by issuing RELEASE ECL EBUS, which takes away APR EBUS RETURN. Now the PI may continue from the point at which it was held.

### 2.12.5 PI and EBus to Microcode Interface

Figures 2-101, and 2-102 are concerned with the interaction of the PI Board and certain other EBox related hardware with the PI Handler and EBus Driver. Both of these handlers are microprograms. Figure 2-101, illustrates the basic signal interfacing between fucntional elements of the PI Board, Control Number 1 Board, and various EBox hardware used during EBus transactions with the Microcode PI Handler and EBus Driver. Figure 2-102 generally relates the PI Handler and EBus Driver functions to the PI Board hardware for given operations. Figure 2-103 is supplied to support functional descriptions to follow.

**2.12.5.1 Sensing the Interrupt** – Initially, assume that the PI Board is enabled and idle. Two devices (DSK) assert interrupts on the same priority interrupt channel; DSKA on channel 5 and DSKB on channel 5. Thus, based on the fixed physical number scheme, the range of physical numbers is 0–7. Further, assume that DSKA is wired to be physical number 1 and that DSKB is wired to be physical number 7, and that DSKA is the device selected.

Referring to Figure 2-101, PI Level 5 is received from both devices and is loaded into PI Request register 5 for arbitration. Because both DSKs are interrupting on the same channel, the PI Network need only check those channels holding interrupts. If none is holding on 5 through 1 (0 is DTE20 and never holds), then channel 5 is selected. The next phase begins by asserting REQ to obtain use of the EBus.




①

| WHAT | LOAD | WAIT1 | WAIT2 | TEST |
|---|---|---|---|---|
| LOAD REQS 0–7 | 1 | 0 | 0 | 0 |
| PI NET SETS UP | 0 | 1 | 0 | 0 |
| PICK WHICH REQ | 0 | 0 | 1 | 0 |
| | 0 | 0 | 0 | 1 |
| | HOLD IN TEST UNTIL PIGRANT | | | |
| REMOVE TEST | 0 | 0 | 1 | 0 |
| PI EBUS PI GRANS REMAINS SET DURING THE DIALOGUE | 0 | 1 | 0 | 0 |
| | 1 | 0 | 0 | 0 |
| PICYC SETS DURING THE END OF DIALOGUE | 0 | 0 | 0 | 0 |
| | *REMAINS CLEAR UNTIL PICYC ← 0 | | | |

REPEAT CYCLE

NOTES:
1. PIO only generated if device is 10-11 interface.
2. Detailed COND/EBUS CTL functions on Fig. 2-70.

*PIGRANT ← DIALOGUE →
PI CYCLE ← MICROCODE HANDLER

10-1669

Figure 2-101 EBox PI Board to Microcode Interface

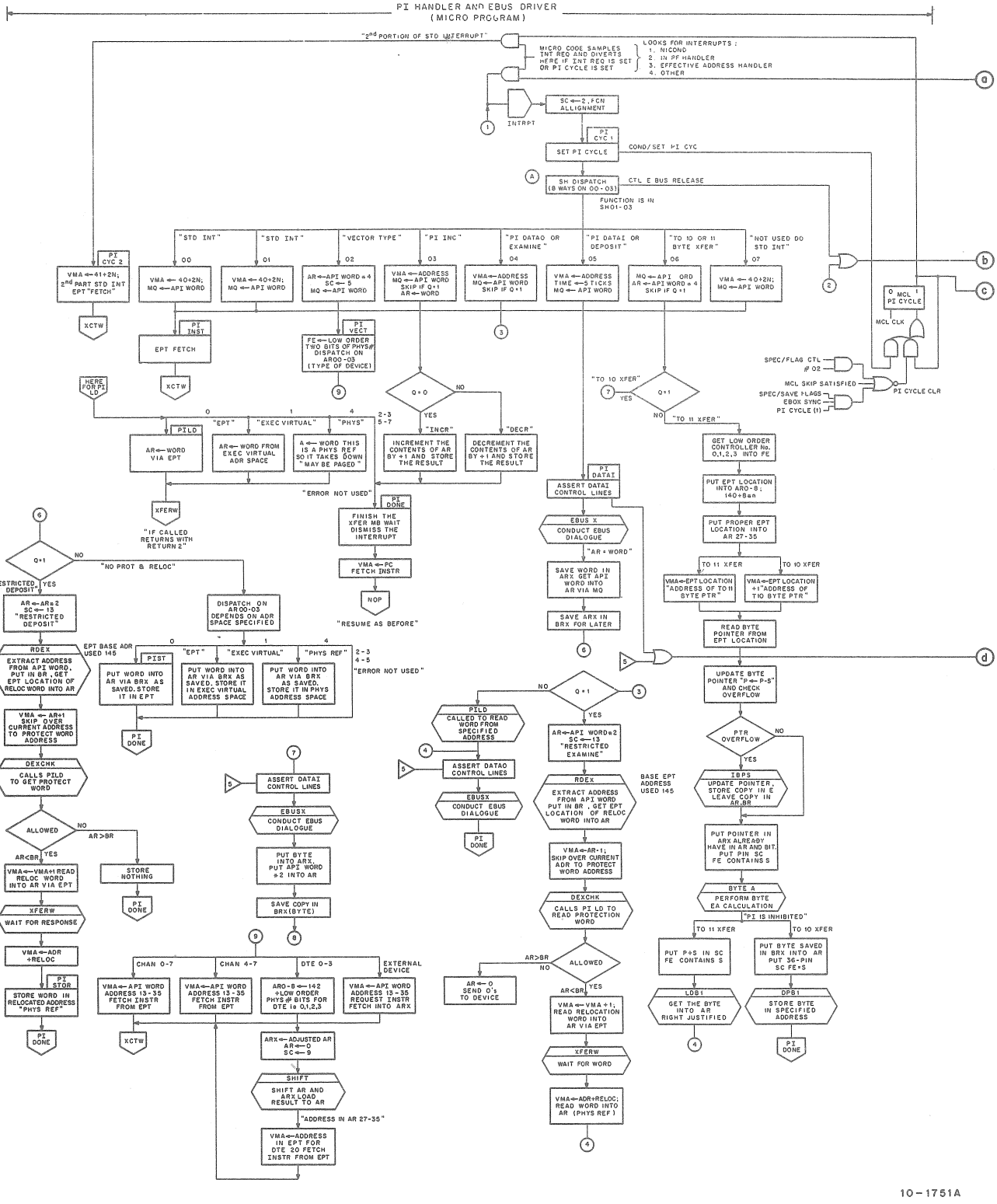



Figure 2-102 EBus Control Hybrid Flow (Sheet 1 of 2)

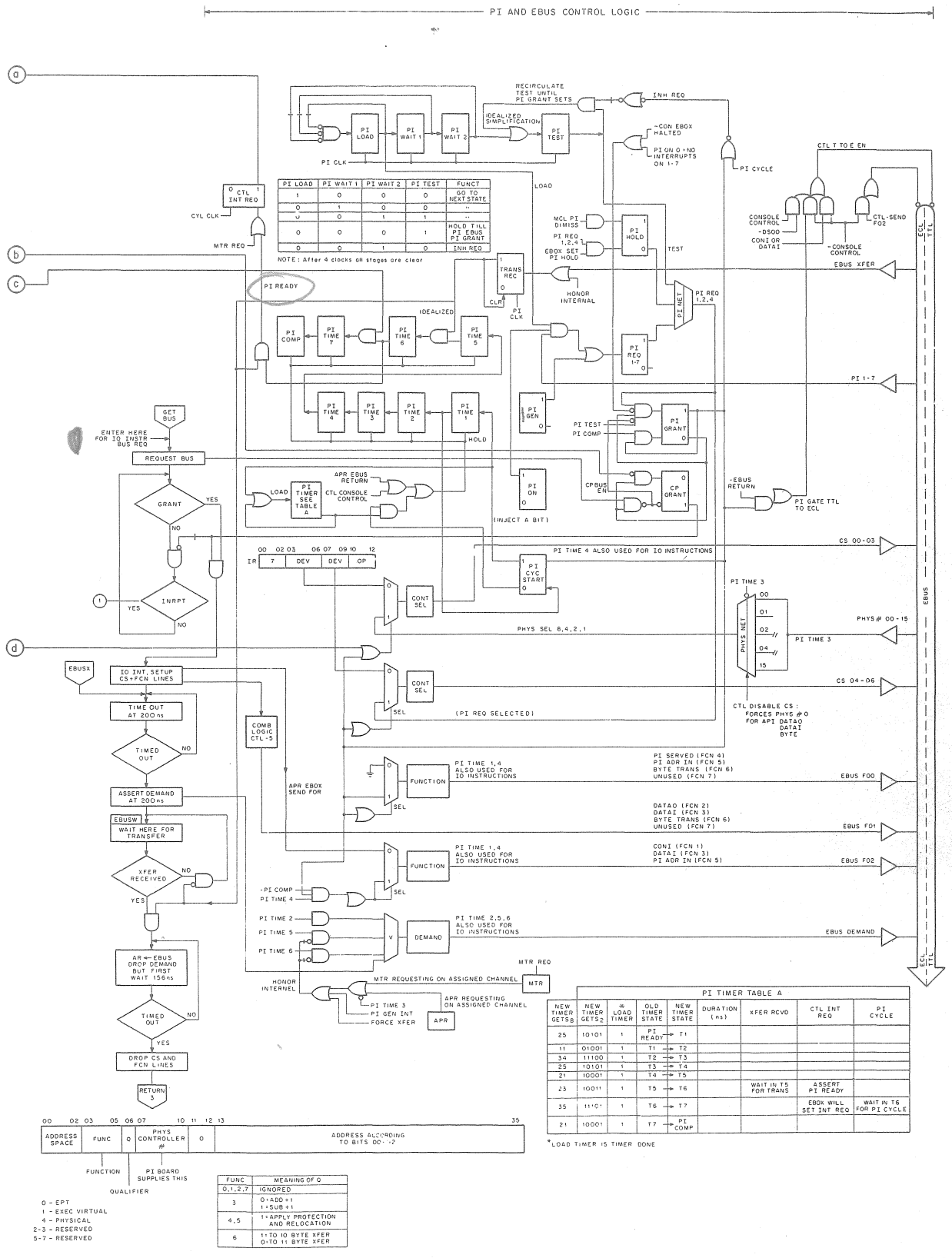



Figure 2-102 EBus Control Hybrid Flow (Sheet 2 of 2)

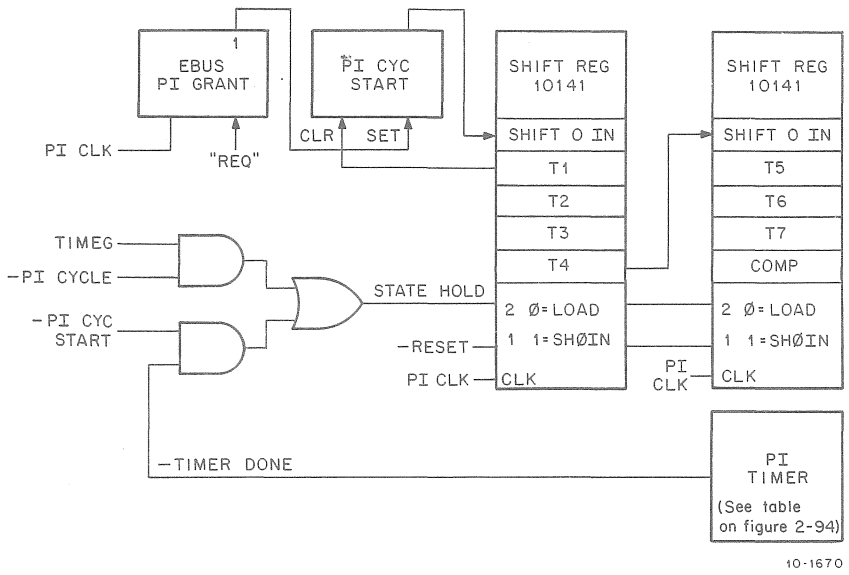


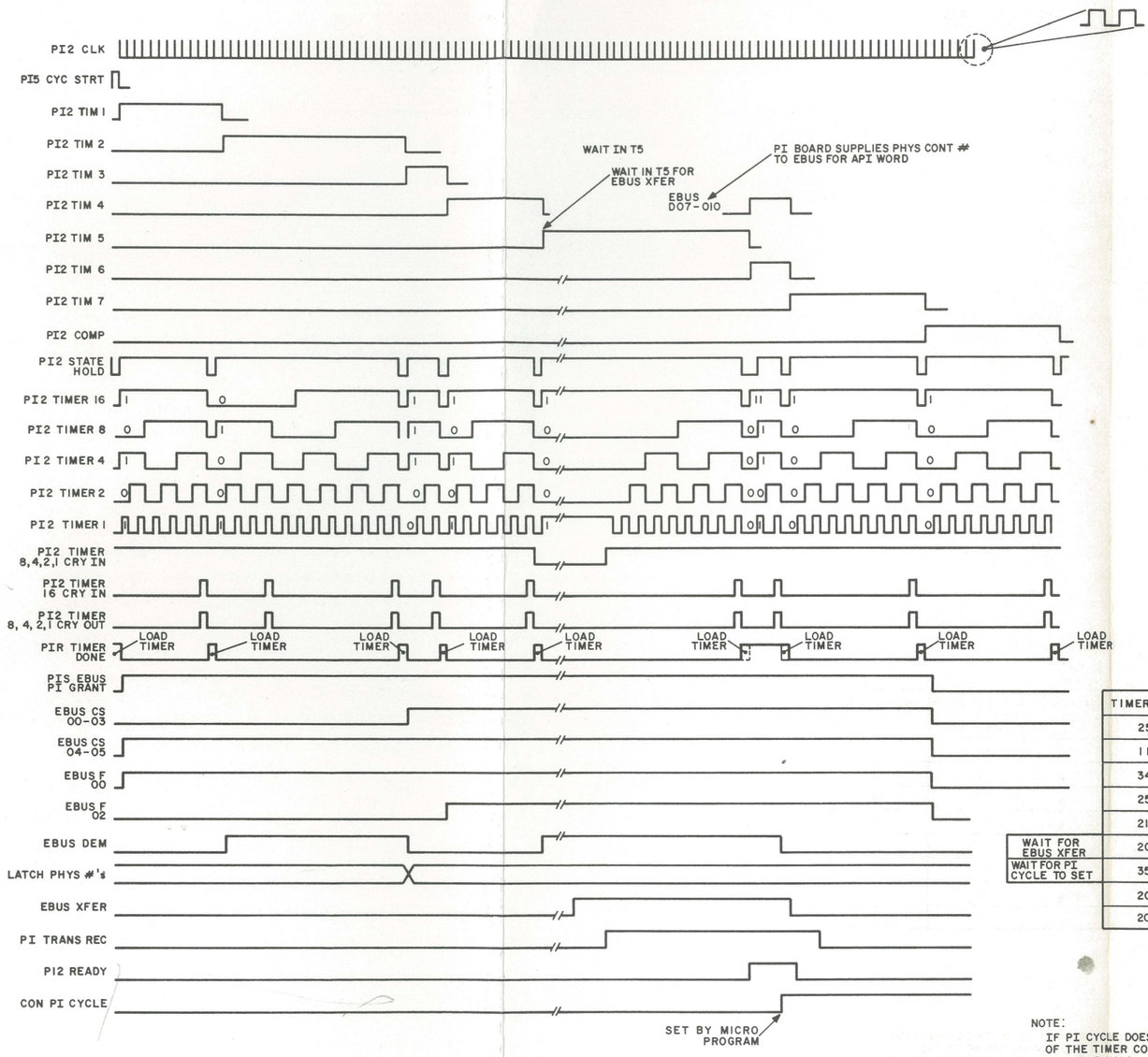

Figure 2-103 Time State Generator Control

**2.12.5.2 Requesting the EBus** – To obtain use of the EBus, the PI logic must set EBUS PI GRANT. This is illustrated on Figure 2-102. Note that the following requirements must be fulfilled to set EBUS PI GRANT:

1. PI test must come up.
2. REQ must be true (PI 4, 2, 1 = some selection).
3. The Ebox may not be halted or there are no interrupts selected on 1–7.
4. EBUS PI GRANT is currently clear.
5. The PI Board is not trying to set CP Grant.

If all five conditions are satisfied, EBUS PI GRANT sets. If the conditions are not currently satisfied, the interrupt waits.

**2.12.5.3 Beginning the Dialogue** – At this time, several events take place. The setting of EBUS PI GRANT enables setting of cycle state, which begins the dialogue. In addition, the PI Timer (see the table on Figure 2-104) is loaded with $25_8$, which defines the duration of the time state entered, in this case time 1. The time states are used to direct the EBus dialogue from beginning to completion. EBUS PI GRANT forces F00 to a 1. This function (4) is PI served and is issued together with CS 04–06, which are encoded to be the selected channel (5). The interrupting devices (in this example two DSKs) decode the function lines F00–02, together with the controller select lines CS 04–06. The PI timer counts from $25_8$ to $37_8$ then generates TIMER DONE. The devices have now had sufficient time to decode the CS and F lines so the next phase of the dialogue begins. The timer is now loaded with $11_8$, Time T1 is removed and T2 is entered.




| TIMER GETS | CURRENT STATE | NEW STATE |
|---|---|---|
| $25_8$ | PI CYC START | T1 |
| $11_8$ | T1 | T2 |
| $34_8$ | T2 | T3 |
| $25_8$ | T3 | T4 |
| $21_8$ | T4 | T5 |
| $20_8$ | T5 | T6 |
| $35_8$ | T6 | T6 |
| $20_8$ | T6 | T7 |
| $20_8$ | T7 | COMP |

WAIT FOR EBUS XFER

WAIT FOR PI CYCLE TO SET

NOTE:
IF PI CYCLE DOES NOT SET BY THE COMPLETION OF THE TIMER COUNTDOWN DURING T6 THE PI BOARD HOLDS AT T6 UNTIL PI CYCLE SETS

10-1754

Figure 2-104 PI Timing

Time 2 enables EBUS DEMAND. Note that the function PI served and controller select lines are maintained. The DSKs are commanded to place their "hardwired" physical numbers onto the EBus, bit 1 for physical number 1 and bit 7 for number 7. Referring to Figure 2-103, DEMAND is held up through Time 2 and then removed while the F and CS lines are maintained. It is good procedure to remove the DEMAND signal before attemping to change the function lines; this avoids any spurious misselection. The timer is next loaded with $25_8$ and T3 (a brief time state) is entered. Here, two functions are performed:

1. The physical numbers, by now on the inputs to a register on the PI Board, are clocked into that register for arbitration.

2. The PI Board is timing out a period of time until it is safe to change the function lines.

The next part of the dialogue is begun when Time 4 is entered.

Here, F00 and F02(5) are asserted; CS00–03 reflect the encoded physical number that has highest priority (#01) and CS04–06 still reflect the PI channel being served. When Time 4 is removed and T5 sets, DEMAND is asserted once again. This time DSKA is selected as the DSK to be serviced. DEMAND commands DSKA to place its API word on the EBus and to assert EBUS TRANSFER to the EBox. The PI Board waits in Time 5 until TRANSFER is received, or forced. If, for example, the interrupting device (DSKA) can respond to most of the dialogue but cannot send EBUS TRANSFER, the PI Board waits. If TRANSFER is not forthcoming, TRANSFER is forced and the EBus (which contains zeros) is treated as an API function of 0. This ultimately causes a 40 + 2n interrupt on the interrupting channel. The DSKs service routine must then decide what went wrong. Assume that the DSKs succeed in placing the appropriate API function word on the EBus and generate TRANSFER. The timer is loaded with $35_8$ and Time 6 is entered where PI READY is asserted. At this point, the PI Board is notifying the EBox microprogram that the API word is currently on the AR mixer inputs.

**2.12.5.4 Terminating the Dialogue** – With the assertion of PI READY, the PI Board waits in Time 6 until the PI Handler (microcode handler) looks at the interrupt. PI READY enables INT REQ to set in the EBox and when the PI Handler detects this, it sets PI CYCLE. Now the timer continues by entering Time 7, drops DEMAND and finally enters COMP, where the CS and FUNC lines, together with EBUS PI GRANT, are removed. This completes the PI Boards dialogue.

**2.12.5.5 Entry to the PI Handler** – Referring to Figure 2-102, the handler is entered at symbolic location INTRPT, with the API word loading into AR, and PI CYCLE not yet set. Thus, the PI Board is at this time in Time 6, waiting for PI CYCLE to be set. The shift counter is loaded with 2, in order to enable the API word in AR to be shifted left two positions, bringing the function code in bits 03–05 into bits 01–03. PI CYCLE is set and then a shift dispatch is given; depending upon the function 0–7, the dispatch is to one of eight routines within the main handler.

**Function 00 – STD INTERRUPT NO TRANSFER**
The word is buffered in MQ. The VMA is loaded with the appropriate 40 + 2n address. This address is implemented via the SCD TRAP mixer (refer to Figure 2-60) and derived from number with PI 4, 2, 1. PI 4, 2, 1 is simply the octal equivalent of the channel on which the interrupt was taken. Thus, the instruction is fetched from 40 + (2 × 5) in the example cited in Subsection 2.8.5.3. This yields an address in VMA of 0000050.

The program branches to Execute Wait (XCTW) where the microprogram waits for the instruction fetched to load into AR. This instruction should be a "JSR," which saves the flags and PC and then enters a subroutine in main memory to deal with the situation. The performing of a JSR causes SPEC/SAVE flags, which clear PI cycle and set PI HOLD, to hold the interrupt.

**Function 01 – STD INTERRUPT KI10, KA10 Device via I/O Bus Adapter or KL10 Device via EBus**
The implementation of this function is identical to that for Function 00. The difference between the function codes is that Function 01 is a premeditated request for a "STD INTERRUPT," where Function 00 is a bus failure condition.

**Function 02 – VECTOR INTERRUPT**
The word is buffered in MQ. The API word contains an address in bits 13–35 and an address space qualifier in bits 0–2. The address is loaded into VMA. Now a dispatch is given on AR00–03. The API word format is presented on Figure 2-102. Note that only three address spaces may currently be specified:

0 – EXEC PROCESS TABLE (EPT)
1 – EXEC VIRTUAL ADDRESS SPACE
4 – PHYSICAL ADDRESS

A routine is called for the storage operation PILD (illustrated in Figure 2-102).

**Fetching from EPT – T**
VMA bits 27–35 receive the AR bits 27–35 via AD. The EBox makes an EPT reference. Referring to Figure 2-83, the qualifiers asserted to the MBox are as follows:

EBOX REQUEST
VMA EPT
PAGE UEBR REF

The hardware normally looks at a combination of SPEC/SP MEM cycle with magic number and user enable to select either VMA EPT or UPT, depending on the state of user. In this case, however, user must be disabled to enable a direct reference to EPT. The AR is loaded with the instruction fetched from CPT. This instruction is either the first of a series of instructions in a service routine or an instruction directing entry to a service routine. As with 40 + 2n interrupt instructions, the instruction should be a JSR to save the flags and PC. By performing a JSR, SPEC/SAVE flags clear PI CYCLE and set PI HOLD on the PI Board. This holds the interrupt.

**Fetching from EXEC Virtual Address Space**
The API word is buffered in the MQ. For this case, the address in bits 13–35 of the API word is a complete virtual address. In fetching from EPT, only bits 27–35 of the address in bits 13–35 contain address information. The MBox appended a base address (EBR) to this 9-bit address. Here the request qualifiers are as follows:

EBOX REQUEST
EBOX READ

The MBox translates the address and supplies the instruction that loads into AR. Once again, transfer is to XCTW, to wait until the instruction actually loads into AR. Then the instruction is performed as with the previous EPT reference.

**Fetching from Physical Memory**
Here, the address contained in the API word bits 13–35, contains a physical address in bits 22–35 while bits 13–17 are clear. To cause a physical reference to occur, the magic number field is coded with number 08 set and this, together with SPEC/SP MEM cycle, inhibits the qualifier MAY BE PAGED. If this signal is not present during EBus request, the MBox does not page the address. The instruction loads into AR as before and then performs. Once again, SPEC/SAVE flags clears PI CYCLE and sets PI HOLD.

**Function 03 – PI INCREMENT**

This function causes a word in the specified address (API word bits 13–35) to be incremented or decremented as a function of the Q BIT in the API word. If Q = 1, the function is decremented; otherwise, it specifies increment. Referring to Figure 2-102, the API word is buffered in MQ and Q is tested. If Q = 0, the contents of the address specified in the API word 13–35 are fetched and incremented. The incremented word is then stored back in the same address and an instruction fetch is performed from PC. This contains the interrupted program. Note that the microcode must set PI HOLD in order to hold an interrupt on the PI Board. This is done when the 40 + 2n or vector function fetches and performs a JSR or similar instruction. Here, after completion of the storage operation, the interrupt is dismissed and PI CYCLE is cleared. PI CYCLE is cleared with SPEC/FLG CTL and number 02.

**Function 04 – PI DATAO or EXAMINE**

The 10-11 interface may perform an Examine function to either core memory or fast memory. In addition, the address supplied in the API word may be a relocated address or not depending on the Q BIT in the API word. Associated with the Examine operation are two words of information for each 10-11 interface in the system. These word pairs are in predefined areas in the EPT. One word of the pair is a protection constant, which limits the address of the virtual address sent in the API word. The number of pages specified in bits 13–26 may be less than or equal to the value of the protection constant, but not greater than that value. The microprogram utilizes the low-order 2 bits of the physical number supplied to the API word (bits 7–10) and forms an address 140 + 8n, where n is the low-order 2 bits of the physical number for the interrupting 10-11 interface. The physical numbers are hardwired as $10_8$–$13_8$. This gives low-order 0, 1, 2, or 3. The FPT location thus obtained is accessed for the protection constant and the comparison is made. If a violation occurs (protection violation), a word of zeros is transmitted to the 10-11 interface via the EBus. If no violation occurs, the relocation word is fetched from EPT and added to the address supplied in 13–26 of the API word. This address is now treated as a physical reference and it is not paged. The word is obtained and transmitted via DATAO function to the 10-11 interface. Upon completion of the EBus dialogue, the PI CYCLE is cleared. Note that for the 10-11 interface Examine function, the interrupt occurs on channel 0.

This channel is implemented solely to enable the 10-11 interface to utilize the PI facility at any time, whether it is on or off for DMA type transfers. No HOLD flip-flop exists for PI0, so clearing PI CYCLE effectively releases the PI0 interrupt. Devices other than the 10-11 interface may utilize this operation under the classification PI DATAO. Two differences in its implementation from that of Examine exist. First, no protection or relocation is applied and hence no violation can occur. A page fault, however, can occur. If this occurs, the PF Handler sets IOPF and transfers control to the operating system. The second difference is that other devices interrupt on channels in the range of 1–7. Once again, holding the interrupt for this one time transfer is unnecessary and only clearing PI CYCLE is necessary to release the PI Board. Other than these differences, the operation is identical to Examine.

**Function 05 – PI DATAO or DEPOSIT**

In terms of the 10-11 interface, this operation is the reverse of Examine, except that after the 10-11 interface sends the API function (which contains the address), the EBox must perform a DATAI function to obtain the 36-bit word to deposit in the specified address. A second difference is that if a violation occurs, after performing the protection check a violation occurs, no word is stored in the specified address. With these exceptions, the operation is basically the same from the point where the 36-bit word is obtained from the 10-11 interface to the completion of the operation.

**Function 06 – PI BYTE TRANSFER**
This function can only be carried out between a 10-11 interface and the EBox. This function is initiated on PI channel 0 as are Examine and Deposit. The transfer is part of either a T011 or T010 byte transfer occurring in the 10-11 interface. The information being transferred is either a byte right-justified in EBus bits 28–35, or a word right-justified in EBus bits 20–35. The API word specifies whether the transfer is T010 or T011 by the state of the Q BIT. If Q = 1, the transfer is T010; otherwise, it is a T011 transfer. In addition, the PI Board is supplying the physical number in bits 07–10 of the EBus while the API word is present. The other portions of the word 0–2, 11–35 are ignored.

**T010 Byte Pointer Fetch, Byte Read, and XFER**
The low-order two bits of the physical controller number 0, 1, 2, or 3 are obtained and combined with EPT base location 14X to form the EPT location of the T011 byte pointer. Next, the byte pointer is obtained from the EPT and updated. The pointer is a standard KL10 byte pointer. The microcode for load byte instructions is used for the pointer update. Note that the byte pointer may specify indirection and/or indexing. Once the effective address has been calculated, the updated byte pointer is stored back in its slot in EPT and the byte is obtained by performing an EBox request. Finally, the byte now in AR is transferred via the EBus (DATAO) to the 10-11 interface and PI CYCLE is cleared.

**T010 Byte Pointer Fetch, Byte Transfer and Storage**
The byte is initially requested by issuing a DATAI to the 10-11 interface. The byte is then picked up via EBus 28–35 and loaded into ARX and into BRX. Next, the low-order two bits of the physical controller number 0, 1, 2, or 3 are obtained and combined with EPT base location 14X to form the EPT location of the T010 byte pointer. The byte pointer is obtained from the EPT and updated. The pointer is a standard KL10 byte pointer. For the T011 XFER, the microcode for deposit byte is used for the pointer update and, as with the byte pointer for T011 XFER, may specify indirection and/or indexing. Once the effective address has been calculated, the updated byte pointer is stored back in its slot in the EPT and the byte is stored in the pointer's effective address. Finally, PI CYCLE is cleared and this terminates the operation.

**Function 07 – UNASSIGNED**
This function is unassigned and currently behaves the same as function 00.

# SECTION 3
# LOGIC DESCRIPTIONS

In this section, a selection of the twelve board types comprising the EBox are described in detail. Wherever possible, a functional perspective is given to highlight the particular functions a board or portion of a board implements, and multiple boards are shown interconnected to aid in tracing various control signals from one functional area to another.

## PHYSICAL CONFIGURATION

The EBox consists of a total of 23 modules, configured as indicated in Figure 3-1. A brief description of each module is contained in the following paragraphs.

*Module M8532, Priority Interrupt Control (PIC)* – One board, illustrated on customer prints PIC 1–6, contains PI ON register 1–7, PI GEN register 1–7, PI REQUEST Register 0–7, PI HOLD register 1–7, and the PI ACTIVE flip-flop. In addition, it contains the priority interrupt networks for arbitration of priority interrupt requests, EBus dialogue logic, control and internal timing, and the assignment registers for the ABR: PIA APR 1,2,4 and Meter PIA 1,2,4.

*Module 8526, Clock (CLK)* – One board, illustrated on customer prints CLK 1–6, contains the crystal-controlled master clock oscillator and crystal-controlled margin clock oscillator, as well as Source and Rate Selection registers and their associated logic. It contains logic and counters to produce the EBus clock, SBus clock, MBox clocks, and EBox clocks. In addition, it contains single step, burst, normal, and diagnostic mode logic and registers. It also contains MR reset, EBus reset, crobar logic, error detection logic, page fail, and MBox request logic.

*Module 8539, Arithmetic Processor Status (APR)* – One board, illustrated on customer prints APR–7, contains an 8-bit APR Status register, 8-bit Interrupt Enable register, and associated interrupt request detection logic. It contains the EBus dialogue control logic used while performing I/O instructions. In addition, it contains the address break compare enable bits, fetch comp, read comp, write comp, and user comp. It contains a 5-bit section register, fast memory bit 36, RAM storage, and parity network. It also contains the fast memory block and word addressing logic, mixers, adder network and current, previous XR, and VMA Block Selection registers. It also contains MBox control and MBox register function decoding logic.

*Module 8525, EBox Control No. 2 (CON)* – One board, illustrated on customer prints CON 1–6, contains CRAM condition field decoding; COND and SKIP enables; and VMA select lines CON VMA SEL 1 and 2. It contains meter, interrupt request and interrupt request detection logic, run and continue logic, IR strobe, DRAM strobe, start logic, various flip-flops, and associated sychronizer logic. It also contains the NICOND decoding and COND ADR bit 10 logic. It contains a 4-bit State register, diagnostic function decoding logic, Parity Enable register, Cache Strategy register, paging enable, trap-enable bits, and I/O control signals for CONO APR, CONO PI, CONO PAG, and DATAO APR. It contains the Load AC blocks and Load Previous Context signals, 4-bit Microcode State register, AR and ARX bit 36 with associated logic, fast memory, write logic, various PI control signals, and associated logic.

Figure 3-1 EBox Module Utilization

*Module 8527, EBox Control No.1 (CTL)* – One board, illustrated on customer prints CTL 1–4, contains CRAM dispatch, field decoding, some adder carry control logic, and register mixer selection control logic for AR, ARX, MQ, and PC. It also contains the majority of the diagnostic decoding logic and the translator enables T to E enable and E to T enable.

*Module 8523, Virtual Memory Address (VMA)* – One board, illustrated on customer prints VMA 1–6, contains an 18-bit VMA adder, VMA AC reference detection logic, a 23-bit VMA register, and associated input mixing logic. It also contains a 23-bit Address Break register, associated match logic, 23-bit Program Counter register, 23-bit VMA Held register, and AR Mixer Mixer (ARMM) logic bits 13–17.

*Module 8528, Data Path (DP)* – Six boards, illustrated on customer prints DP 1–5, each contain six bits of a full 36-bit data path. Each board contains the following mixers: AR Mixer (ARM), ARX Mixer (ARXM), MQ Mixer (MQM), ADA Input Mixer, ADB Input Mixer, ADXA Input Mixer, and ADXB Input Mixer. In addition, each board contains the following registers: Arithmetic Register (AR), Arithmetic Register extension (ARX), Buffer Register (BR), Buffer Register extension (BRX), and Multiplier Quotient register (MQ). It also contans fast memory, the adder (AD), and adder extension (ADX). In addition, it contains the fast memory, write pulse generation logic, and fast memory, write pulse generation logic, and fast memory parity network.

*Module 8512, Control RAM (CR)* – Five boards, illustrated on customer prints CR 1–7, each contain 14 bits of the control word (microinstruction) stored in RAMs containing 1280 words. In addition, each board contains CRAM address gating and 14 bits of the CRAM output register (CRAM register).

*Module 8511, Control Ram Address (CRA)* – One board, which is illustrated on customer prints CRA 1–6. This board contains the circuitry to generate the address of the next CRAM word. This includes the microcode push-down stack, plus the Dispatch and Skip logic.

*Module 8510, Shift Matrix (SH)* – One board, illustrated on customer prints SHM 1–5, contains shift counter decoding logic, shift matrix, and AR and ARX parity networks.

*Module 8530, Memory Control (MCL)* – One board, illustrated on customer prints MCL 1–7, contains CRAM MEM field decoding; memory request enable logic; request type decoding, e.g., MCL VMA Read, MCL VMA Pause, MCL VMA Write. It also contains User and Public Enable logic, as well as all the request-type qualifiers. It contains bits 1–12 of the VMA Held or PC Mixers, together with various VMA Control and Selection logic.

*Module 8522, IR, DRAM, and Carry (IRD)* – One board, illustrated on customer prints IRD 1–5, contains the 13-bit Instruction register (IR), 4-bit IRAC register, DRAM address mixers, DRAM, and 15-bit DRAM Output register. In addition, it contains the IR Test Satisfied logic and normalization CRAM address bits (IR NORM 08–10). It also contains the AD and ADX carry anticipation networks (CARRY SKIPPER).

*Module 8524, Shift Counter Adder (SCAD)* – One board, illustrated on customer prints SCD 1–6, contains the 10-bit Shift Counter register and associated input mixer, 10-bit Floating Exponent register, and associated input mixer, AR Mixer Mixer (ARMM) bits 0–8, and SCD TRAP Mixer (32–35). It also contains the 10-bit Shift Counter Adder (SCAD) as well as the Program Counter Flags register and mode control logic.

## 3.1 INSTRUCTION REGISTER LOADING AND CONTROL

Refer to Figures 3-2 and 3-3. The IR is composed of 13 mixer latches as illustrated. The default selection is AD selected by -CLK MB XFER. The alternate selection is the cache data lines selected by CLK MB XFER. Because the IR consists of latches (DC devices), the clock is used indirectly to synchronize unlatching and latching of IR. This is done by ORing the EBox clock with the control signal on the IR Board. Unlatching the IR may be accomplished in one of three ways.

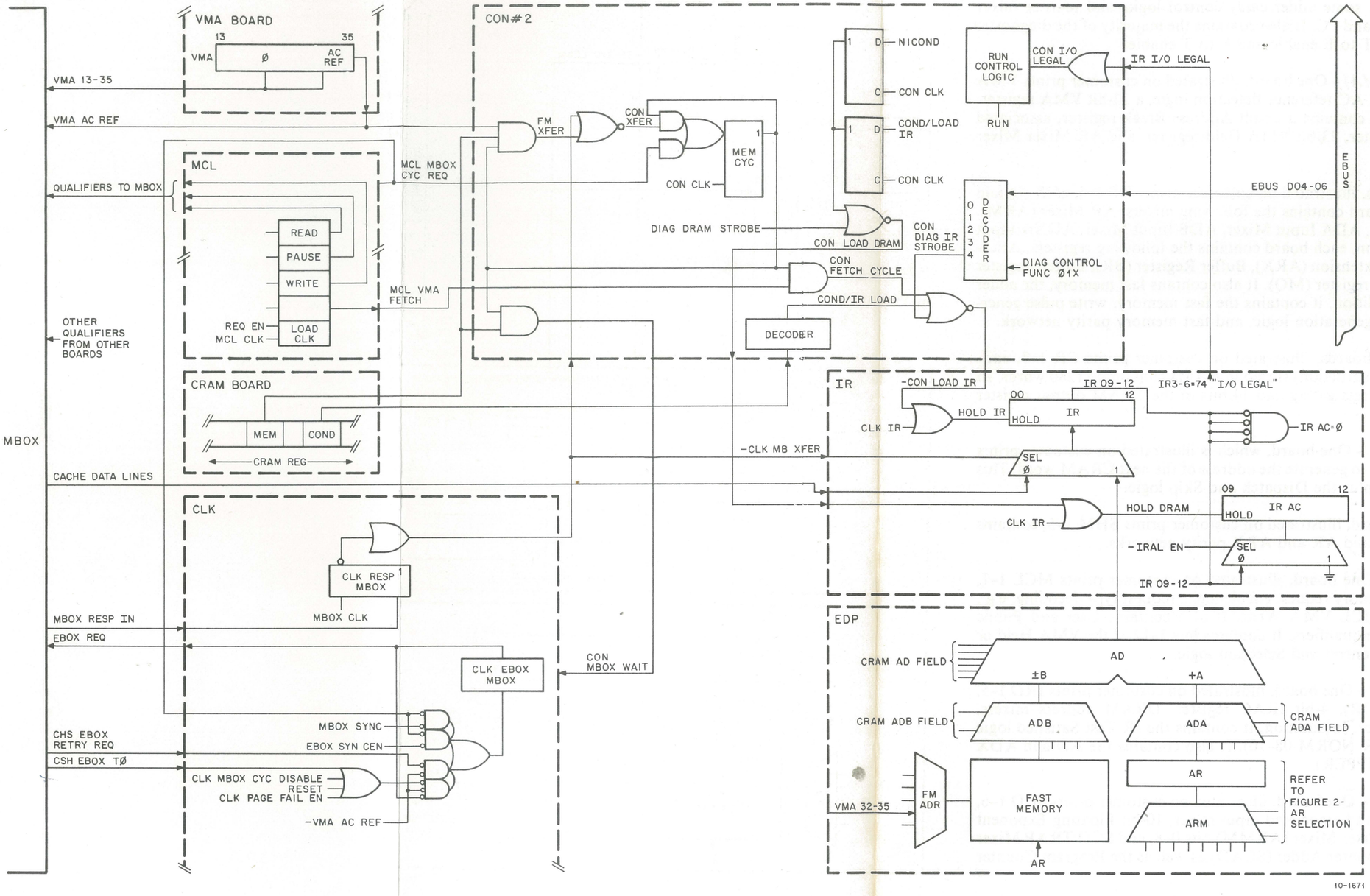



Figure 3-2 IR DRAM Control (Part 1)

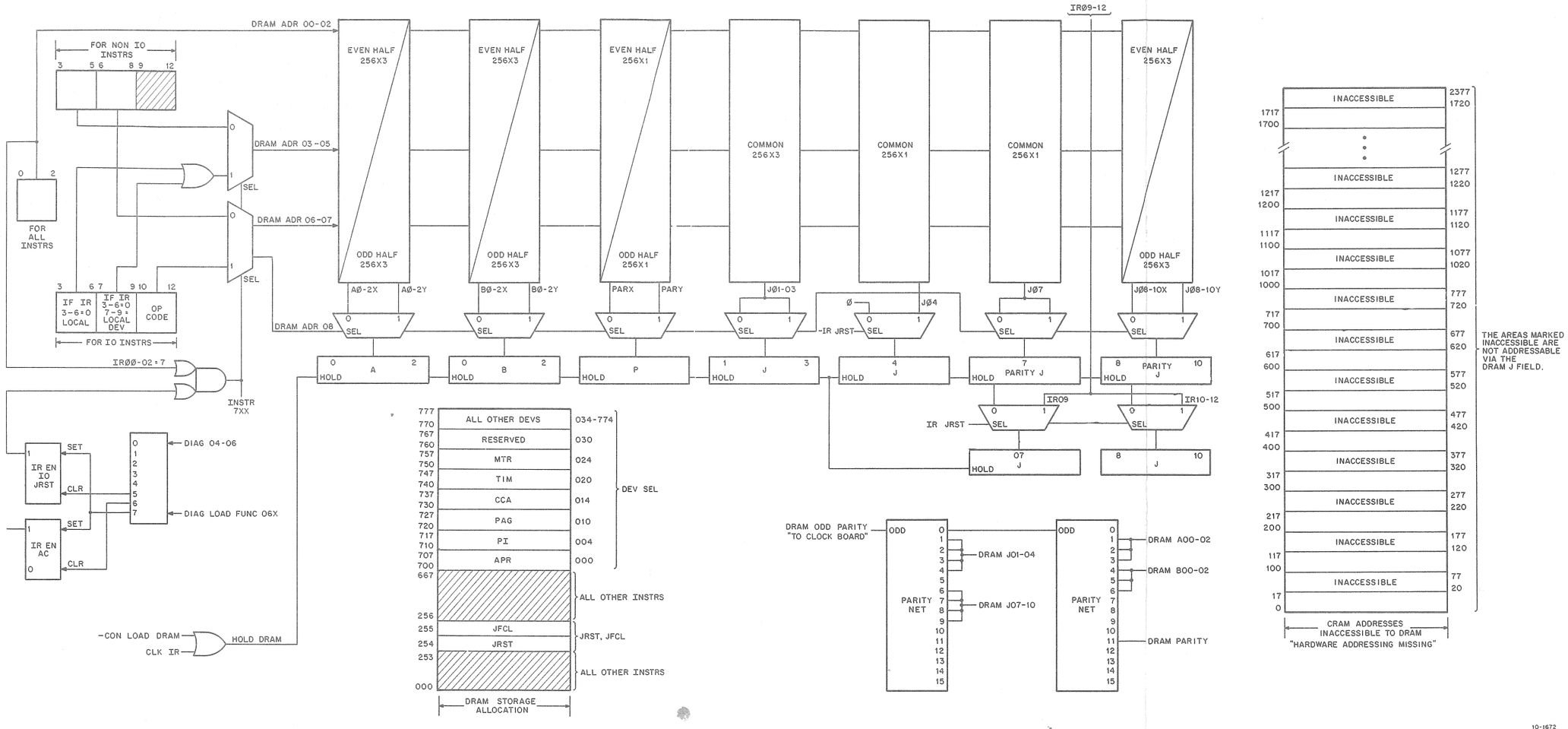



Figure 3-3 IR DRAM Control (Part 2)

During an instruction fetch, a logic level MCL FETCH is developed together with EBox Read. These qualifiers are latched at the same time that the VMA is latched during the EBox request. They are latched until the next EBox request. Each time a memory cycle is begun for any reason, MEM CYCLE sets. It remains set until one of two events occurs. Either MBXFER occurs in response to an MBox cycle, or FM XFER occurs in response to an internal fast memory cycle. Either of these decouples the feedback path for the MEM CYCLE flip-flop. Note that while MEM CYCLE and MCL VMA FETCH are true, the IR is unlatched because -CON LOAD IR becomes false removing HOLD IR.

A second method for unlatching the IR is via the microinstruction COND field function COND/LOAD IR. This may be used in cases where an instruction is loaded into AR to be executed. The microinstruction selects the AD function as "A" while selecting the AR on the ADA input. Because the default selection for IR is the AD, the instruction in AR would appear on the IR input mixer.

The operation of unlatching and loading in this manner takes one microinstruction as indicated in Figure 3-4. Note that CLK IR is logically ORed with -CON LOAD IR on the IR Board.

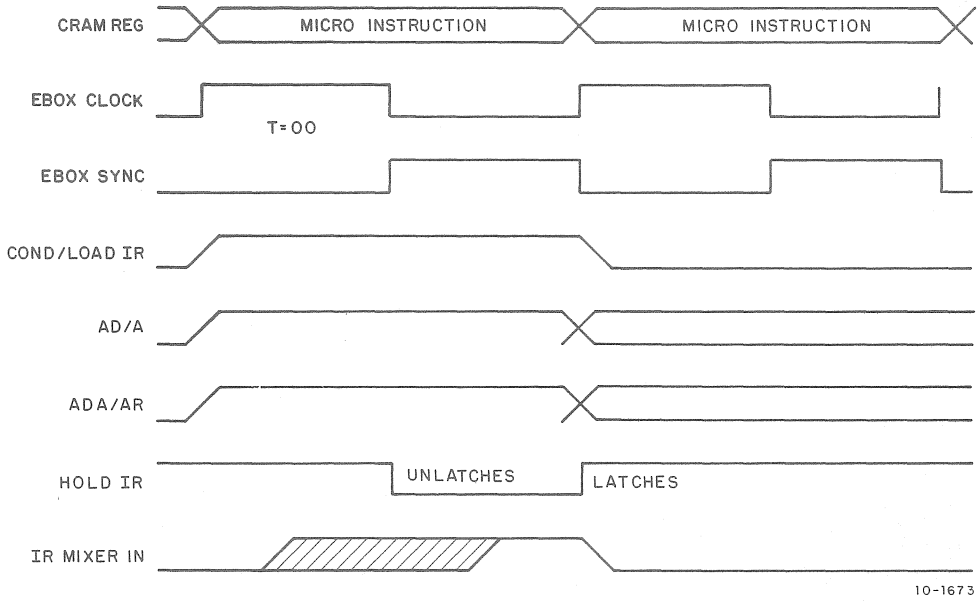



Figure 3-4 IR Loading Via AR (COND/LOAD IR)

By using diagnostic console function 014 (STROBE IR), information previously loaded into AR or ARX may be loaded into IR. This provides a powerful diagnostic tool. In addition, this function is used to address the DRAM while loading it.

When fetching instructions from fast memory via AD, it is sometimes necessary to use the COND/IR LOAD function to enable AD to IR. Referring to Figures 3-2 and 3-5, VMA bits 32–35 address fast memory as specified by the microinstruction FM ADR field. At the same time (for example), the ARX field selects AD while the AD field selects "B". The ADB field function is FM and once again the COND field is LOAD IR.

Once again, note that the unlatching and latching of IR is in step with the EBox clock (CLK IR).

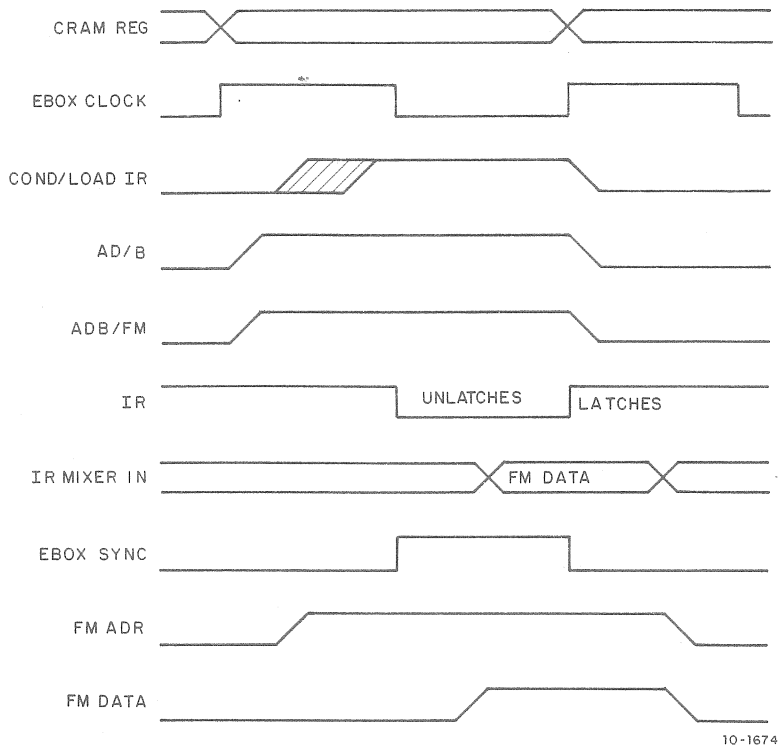



Figure 3-5 Loading IR Via FM (COND/LOAD IR)

### 3.1.1 DRAM and IRAC Control

The DRAM register is controlled in a manner similar to that of IR. The DRAM register consists of 19 mixer latches. Refer to Figure 3-3; unlatching the DRAM register may be accomplished in one of three ways. As with IR, note unlatching and latching of the DRAM register is synchronized by ORing the EBox clock with the control signal on the IR Board.

Each time that the COND/LOAD IR function is used to unlatch the IR, it also enables the generation of CON LOAD DRAM on the next EBox clock. Thus, the IR unlatches beginning with the trailing edge of one EBox clock and latches on the leading edge of the next. Similarly, the DRAM register unlatches beginning with the trailing edge of the EBox clock that latched IR, and latches once again on the leading edge of the following EBox clock. The timing is illustrated in Figure 3-6.

A similar operation takes place following NICOND Dispatch. Referring to Figures 3-2 and 3-7, NICOND is latched into a flip-flop on the control board at the same time that the microinstruction selected by the NICOND Dispatch loads into the CRAM register.

Here we assume the case where some instruction has completed its store cycle. An earlier micro-instruction generated MEM/FETCH which started the EBox Request.

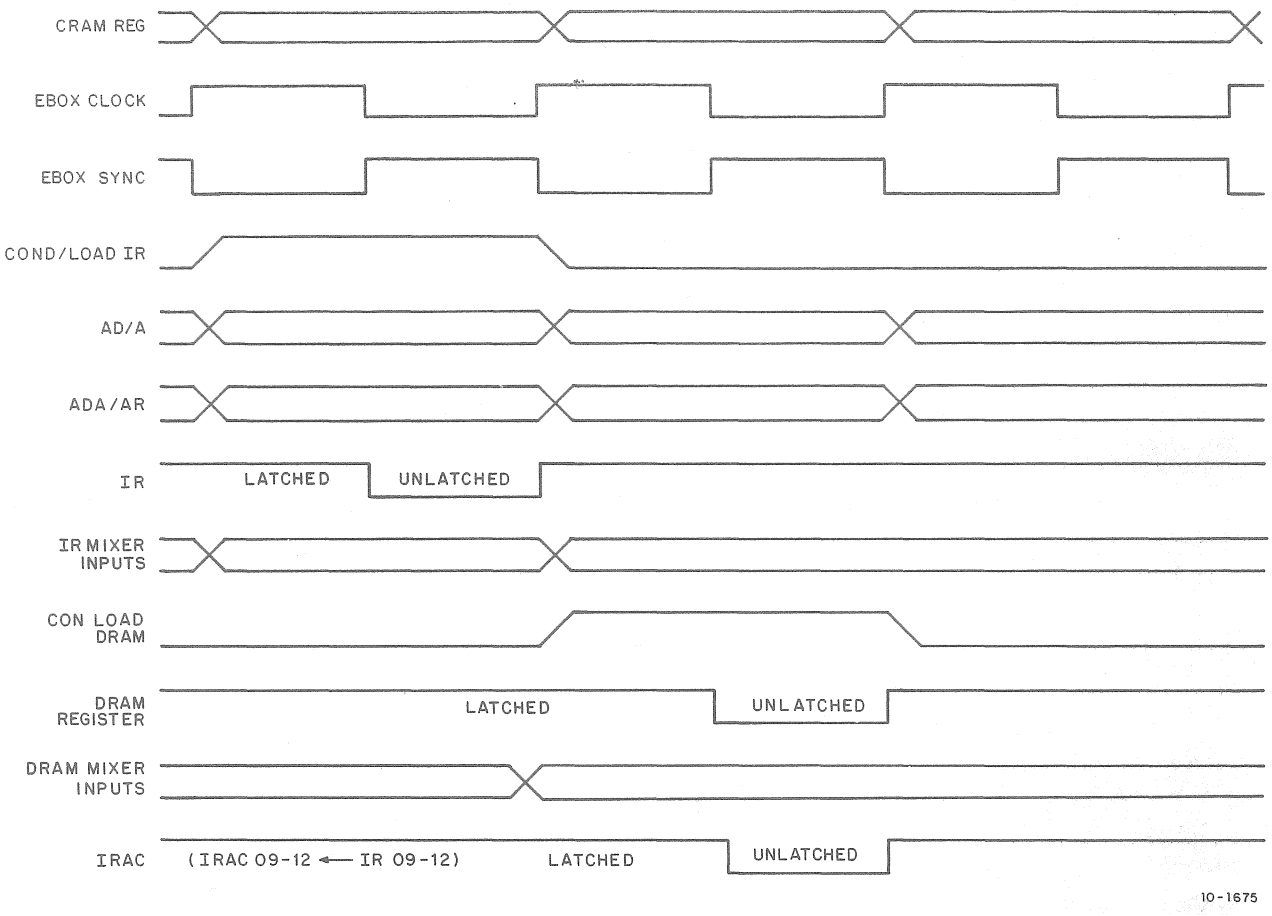



Figure 3-6 DRAM Loading Following COND/LOAD IR

### 3.1.2 DRAM Addressing and Selection

Assume IR EN IO, JRST, and IR EN AC are set. The DRAM addressing logic maps the incoming instruction code into the DRAM register as indicated in Figure 3-3. Note that I/O instructions address the DRAM in a slightly different fashion than non-I/O instructions. I/O instructions have bits 0–2 of IR equal to 7; this is detected on the IR Board as IR INSTR 7XX and enables the DRAM ADR to be formed as follows:

DRAM ADR 00–02 ←IR 00–02
DRAM ADR 03–05 [IR 7–9 v111]
DRAM ADR 06–08 ←IR 10–12

As indicated on the figure, for I/O instructions, IR 3–9 is the device select code. If bits 3–6 are equal to zero, the device is local to the processor, i.e., in the EBox. Currently, there are six local devices:

APR: DEV 000
PI: DEV 004
PAG: DEV 010
CCA: DEV 014
TIM: DEV 020
MTR: DEV 024
(UNUSED: DEV 030)

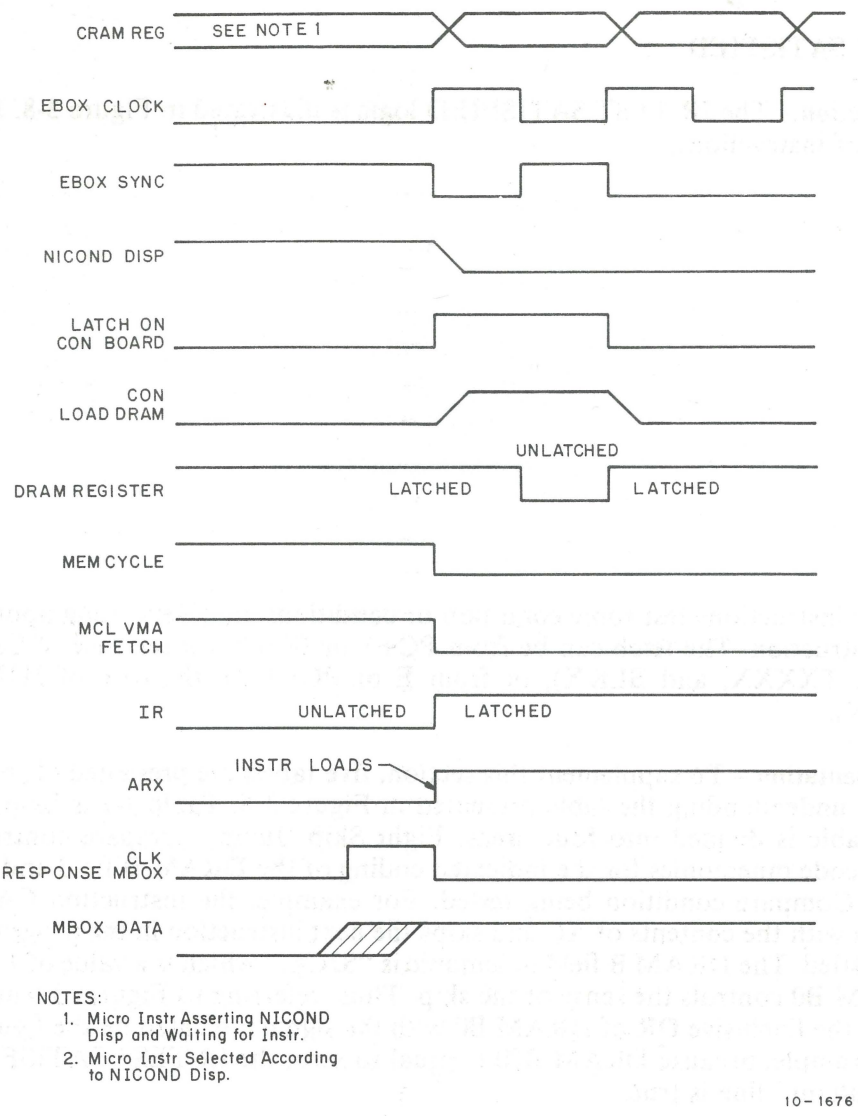



Figure 3-7 NICOND Dispatch and Waiting

If IR bits 3–6 are nonzero, the device is external to the processor. This includes device select codes 034 to 774.

All other op codes in the range of 000–677 address locations in the DRAM that correspond to locations 000–677. This is illustrated in Figure 3-2. DRAM address 00–02 is formed from IR 00–02, while DRAM address 03–08 is formed from IR 03–08.

AC decoded jumps JRST and JFCL reference locations in the DRAM that correspond to their numerical op codes (254 and 255, respectively). The DRAM register is loaded specially for JRST. Note that IR JRST (Figure 3-3) forces DRAM register J4 to zero while enabling DRAM J07–10 to be input from IR 09–12. This enables the microcode for JRST to be entered at the appropriate location relative to the type of code in IR 09–12.

DRAM register bits 00, 05, and 06 are missing in the hardware (Figure 3-3). This prevents DRAM J Dispatch from accessing certain CRAM locations.

### 3.1.3 IR TEST SATISFIED

**3.1.3.1 Introduction** – The IR TEST SATISFIED logic is illustrated in Figure 3-8. It is used with the following types of instructions:

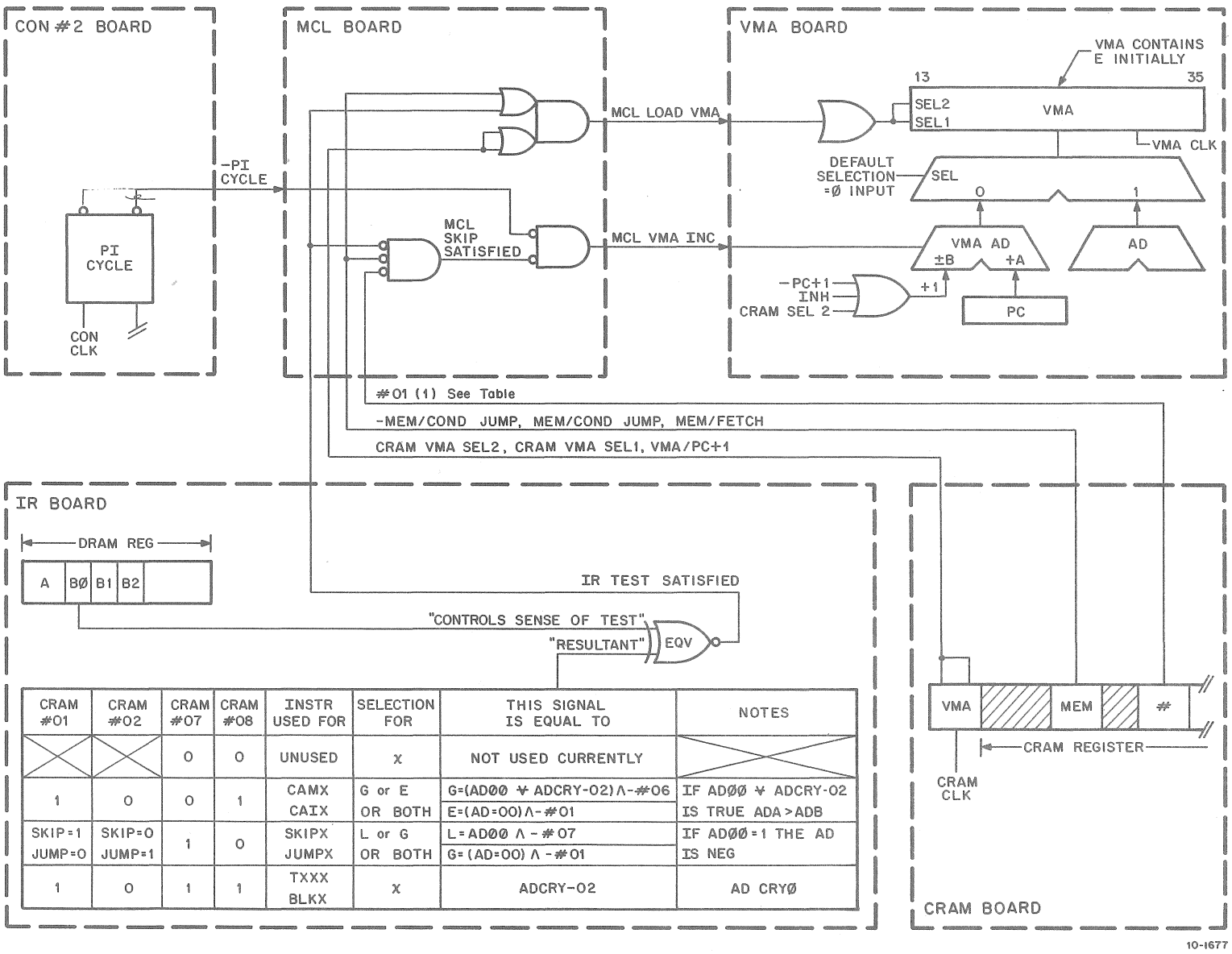
CAMXX
CAIXX
SKIPXX
JUMPXX
TXXXX
BLKX
AOSXX
SOSXX
AOJXX
SOJXX
AOBJX
JFCL

In general, these instructions test some condition or conditions and, depending upon the result of the test, fetch an instruction. The fetch can be from PC+1 or PC+2, (in the case of CAIXX, CAMXX, SKIPXX, AOS, TXXXX, and BLKX), or from E or PC+1 (in the case of JUMPXX, AOJXX, SOJXX, AOBJX).

**3.1.3.2 Implementation** – To supplement this section, five tables are presented (Tables 3-1 through 3-5), which aid in understanding the table presented in Figure 3-8. Table 3-1 is Skip, Jump, Compare controls. This table is divided into four areas. Eight Skip, Jump, Compare controls are indicated. These are microcode mnemonics for the indicated coding of the DRAM B field and imply the type of Skip, Jump, or Compare condition being tested. For example, the instruction CAIE compares the effective address with the contents of AC and skips the next instruction in the program sequence if the condition is satisfied. The DRAM B field mnemonic is "SJCE," which is a value of 1 in DRAM B. The coding of DRAM B0 controls the sense of the skip. Thus, referring to Figures 3-9 and 3-10, IR TEST SATISIFIED is the Exclusive OR of DRAM B0 with the signal indicated on the figure as "resultant." In the current example, because DRAM B00 is equal to zero, the IR TEST SATISFIED signal is true only if the "resultant" line is true.

As indicated in Figure 3-9, the combination of AD = 0 with DRAM B 01 (0) and CRAM #07(1) enables "resultant" to be true. This yields IR TEST SATISFIED. Referring to Figure 3-8, the VMA contains E, which it received at AREAD time. The VMA field function is PC+1 [CRAM VMA SEL 1 (0) ∧CRAM VMA SEL 2 (1)]. Because PC+1 INHIBIT is false at this time, the "B" input to VMA AD is equivalent to +1, while the VMA AD function is "A+B." The MEM field function is "FETCH," and the magic number field function is "COMP FETCH," which is coded as #201. Thus, #01 (1) with "FETCH" and IR TEST SATISFIED gives MCL SKIP SATISFIED. Providing PI CYCLE is clear, MCL VMA INC increments the VMA AD SUM, which is now PC+1, to a value of PC+2.

Note that either bit of the CRAM VMA field enables one side of the MCL VMA load gate and that IR TEST SATISFIED or -MEM/COND JUMP enables the other side. This is necessary to allow IR TEST SATISFIED to inhibit loading the VMA during Jump-type instructions. VMA contained the jump address prior to the test. Note that the magic number field function and MEM field function for Jump-type instructions is different than that for Skips and Compares. It is necessary to prevent PC+2 from occurring and this is accomplished by blocking the term MCL SKIP SATISFIED. Because the magic number field function for jumps, which is "JUMP FETCH," has #01 (0), the gate is inhibited. If the test is not satisfied, VMA loads with PC+1 and program operation continues.




| CRAM #01 | CRAM #02 | CRAM #07 | CRAM #08 | INSTR USED FOR | SELECTION FOR | THIS SIGNAL IS EQUAL TO | NOTES |
|---|---|---|---|---|---|---|---|
| | | 0 | 0 | UNUSED | X | NOT USED CURRENTLY | |
| 1 | 0 | 0 | 1 | CAMX<br>CAIX | G or E<br>OR BOTH | G=(ADØØ ⩛ ADCRY-02)∧-#06<br>E=(AD=00)∧-#01 | IF ADØØ ⩛ ADCRY-02<br>IS TRUE ADA>ADB |
| SKIP=1<br>JUMP=0 | SKIP=0<br>JUMP=1 | 1 | 0 | SKIPX<br>JUMPX | L or G<br>OR BOTH | L=ADØØ ∧ -#07<br>G=(AD=00) ∧ -#01 | IF ADØØ=1 THE AD<br>IS NEG |
| 1 | 0 | 1 | 1 | TXXX<br>BLKX | X | ADCRY-02 | AD CRYØ |

Figure 3-8 IR Test Satisfied

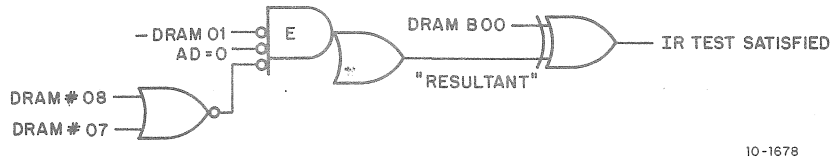



Figure 3-9 IR Test Equal

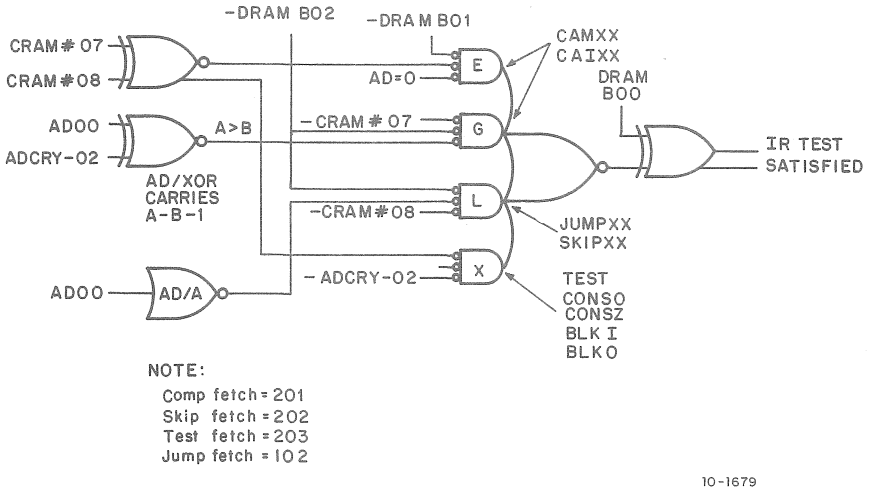



Figure 3-10 IR Test Satisfied Logic

**Table 3-1 Skip, Jump, Compare Controls**

| DRAM B Field | Skip, Jump, Compare Controls | Controls Sense of Skips, Jumps, and Compares DRAM B00 |
|---|---|---|
| 3 | SJC- | 0 |
| 2 | SJCL | 0 |
| 1 | SJCE | 0 |
| 0 | SJCLE | 0 |
| 7 | SJCA | 1 |
| 6 | SJCGE | 1 |
| 5 | SJCN | 1 |
| 4 | SJCG | 1 |

NOTE
See Table 3-4; uses Skip or Jump fetch with various AD functions.

## Table 3-2 Test Controls

| DRAM B Field | Test Controls | Controls Sense of Test DRAM B00 |
|---|---|---|
| 4 | TN- | 1 |
| 0 | TNE | 0 |
| 0 | TNA | 0 |
| 4 | TNN | 1 |
| 5 | TZ- | 1 |
| 1 | TZE | 0 |
| 1 | TZA | 0 |
| 5 | TZN | 1 |
| 6 | TC- | 1 |
| 2 | TCE | 0 |
| 2 | TCA | 0 |
| 6 | TCN | 1 |
| 7 | TO- | 1 |
| 3 | TOE | 0 |
| 3 | TOA | 0 |
| 7 | TON | 1 |

**NOTE**
See Table 3-4; uses TEST fetch with various AD functions.

## Table 3-3 CONSX and BLKX Controls

| DRAM B Field | CONSX, BLKX Controls | Controls Sense of CONSX, BLKX, Skip DRAM B00 | COND Causing Skip |
|---|---|---|---|
| 2 | BLKI | 0 | TEST FETCH<br>TEST BRL |
| 0 | BLKO | 0 | TEST FETCH<br>TEST BRL |
| 5 | CONSO | 1 | TEST FETCH<br>TEST AR BR |
| 1 | CONSZ | 0 | TEST FETCH<br>TEST AR BR |

**Table 3-4 Fetch Control Modifiers**

| Actual Instruction Using | Microinstruction Function | MEM Field | Magic No. Field | 01 | 02 | 07 | 08 |
|---|---|---|---|---|---|---|---|
| CAMXX, CAIXX | COMP FETCH | FETCH | 201 | 1 | 0 | 0 | 1 |
| SKIPXX | SKIP FETCH | FETCH | 202 | 1 | 0 | 1 | 0 |
| BLKO, BLKJ, CONSO, CONSZ, TXXXX | TEST FETCH | FETCH | 203 | 1 | 0 | 1 | 1 |
| JUMPXX | JUMP FETCH | FETCH | 102 | 0 | 1 | 1 | 0 |

**Table 3-5 CRY0 Generation (MACRO)**

| Instruction That Uses | CRY0 Generators Used | AD Field Function | Additional Signal |
|---|---|---|---|
| BLKI, BLKO | TEST BRL | ORCB+1 | GEN CRY 18 |
| CONSO, CONSZ | TEST AR·BR | CRY A·B#0 | |
| TEST | TEST AR·ACO | CRY A·B#0 | |
| TEST | NO CRY | SETCA | |

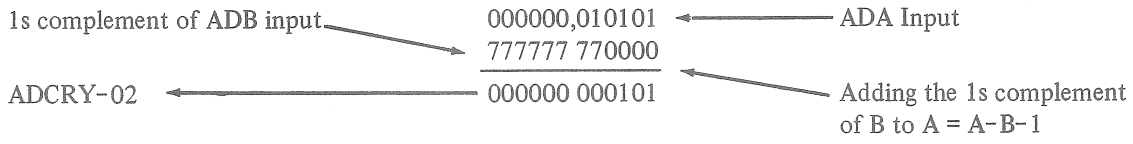
Figure 3-10 illustrates the actual logic that develops IR TEST SATISFIED. The use of the E, G, L and X portions is indicated. The result of the test in the AD determines one of the conditions on each gate. For Equal (E), the term is straightforward AD = 0. In the case of Greater (G), the Exclusive OR of the sign of AD (AD00) with a carry out of the AD sign (AD CRY –02) produces the A >B output when AD is performing the Exclusive OR function. For example, assume CAIG AC, 010101.

```
AR = 000000, 010101   ;O,E
AC = 000000, 007777   ;(AC)
```

The function performed in AD is:

```
ADB←FM; (AC)
ADA←AR; O, E
AD = XOR
```

Note that while the AD performs the logical function XOR, the carry function is A-B-1 (Table 2-8, ALU Functions). Therefore, the ADB input is 000000,007777 and the ADA input is 000000,010101. The operation is as follows:



Note that the following relation is true:

-B = $\overline{B}$+1
-B-1 = $\overline{B}$+1 -1
-B-1 = $\overline{B}$, which is the 1s complement of B.

XORing AD CRY -02 with AD00, which is 0, should indicate A >B.

For less than (L), the term is AD00, and this indicates the AD result as a negative value. Skips utilize the Boolean AD function A. Here, the carries function is really A-1. Thus, if the instruction is SKIP L 0, E, the contents of E are compared with zero and a SKIP occurs if (E) is any negative value. The implementation follows:

```
X: SKIPL 0, E
(E) = 777777, 777774 ; -4
AR = (E)
```

The function performed in AD is ADA ←AR, AD = A and effectively the (AR) is compared to zero because any negative value in AR satisfies the SKIP until a value of zero is placed in AR. This turns off AD00.

The remaining term (X) is used during TEST, BLKI, BLKO, CONSO, and CONSZ instructions. The AD carries function is AB-1. For example, assume the instruction is CONSO DEV, 1. At the time of the test, BR contains 000000,000001, the effective address, and AR contains the bits (if any) from the device. The implementation follows:

```
BR = 000000,000001      ;O,E
AR = 000000, 000001     ;assume the bit was set in the device
```

| | |
|---|---|
| | 000000,000001 |
| "AND" | 000000 000001 |
| | 000000,000001 |
| For the carries function add -1 | 777777,777777 |
| AD CRY -02 ← | 000000,000000 |

Here ADCRY–02 inhibits the (X) function but DRAM B0 is coded to enable the IR TEST SATISFIED condition. The PC is updated by +2 and loaded into VMA (Figure 3-9). If the instruction were CONSZ DEV, 1 and the device flag was not set, the AD function [000000,000000-1] yields -1 and -AD CRY–02. This satisfies the (X) function and DRAM B0 is clear. Once again, the IR TEST SATISFIED condition is satisfied and the SKIP occurs.

## 3.2 PROCESSOR TIMING

The KL10 is a synchronous machine. Figure 3-10 illustrates the basic clock layout and distribution.

### 3.2.1 Clock Overview

The clock resides in the EBox and contains a selectable source (Figure 3-12). This source can be a crystal controlled 50 MHz oscillator, for normal processor operations, but may be an external source for special applications or a 56 MHz crystal-controlled oscillator for speed margining.

Basically, the clock consists of three other rather distinct sections: the clock control, the EBox clock control, and the clock diagnostic control labeled ① , ② , ③ , respectively, in Figure 3-13.

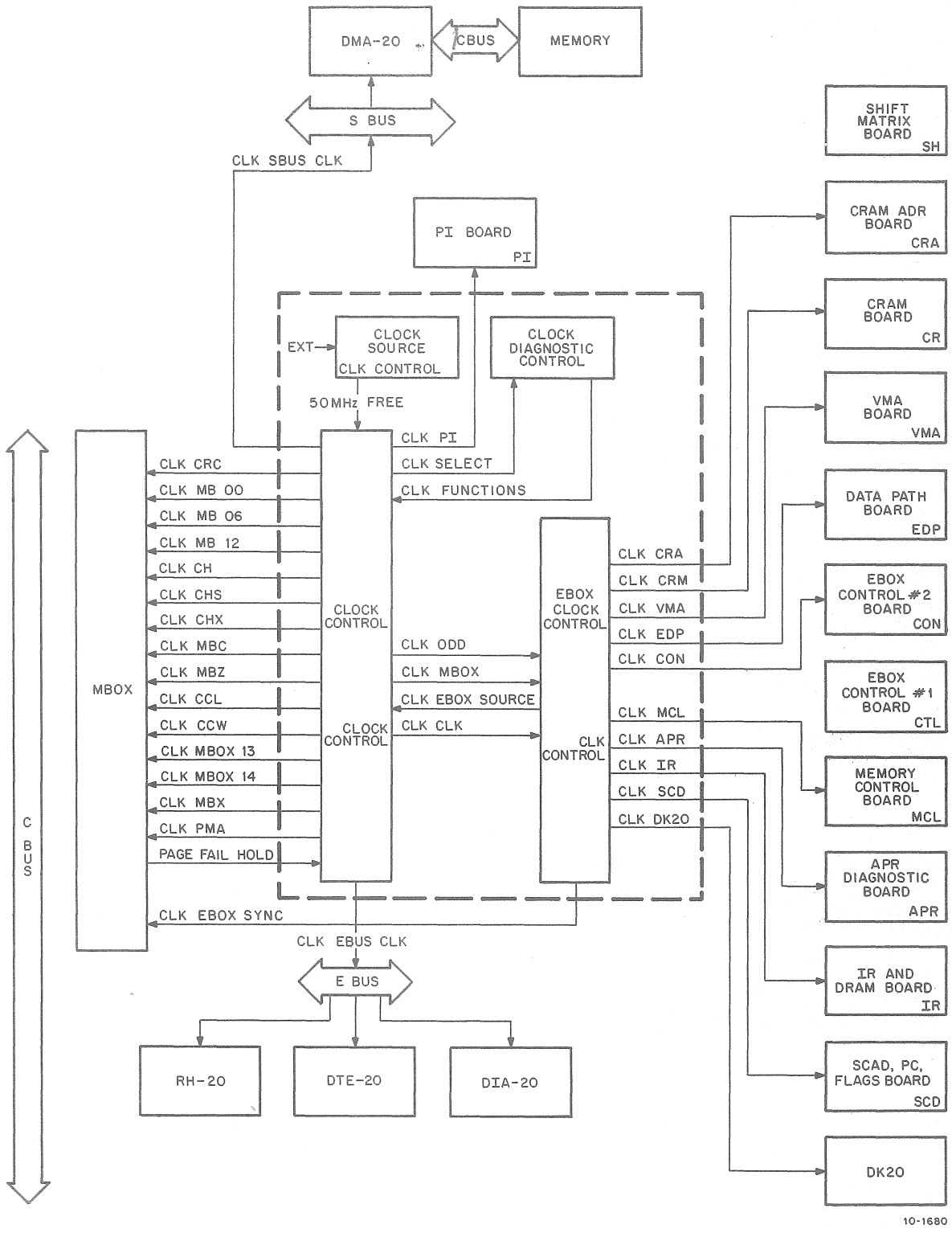



Figure 3-11 Clock Basic Block Diagram

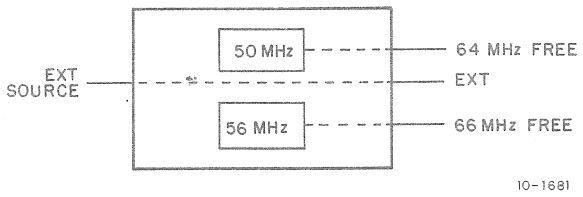



Figure 3-12 Clock Source Simplified

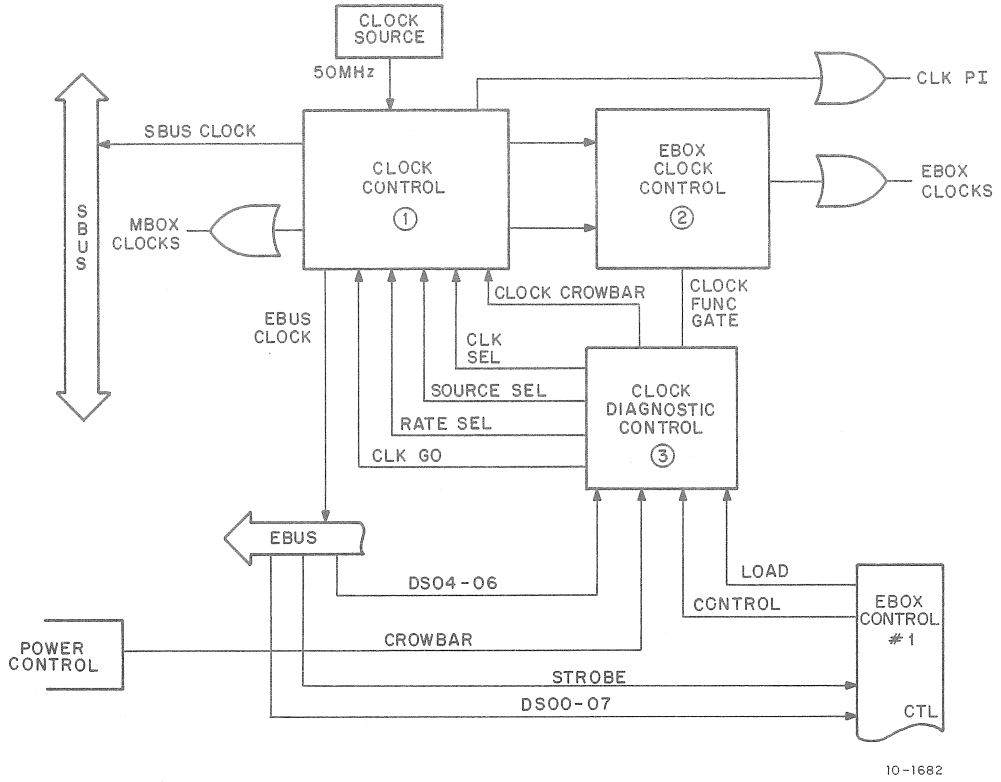



Figure 3-13 Basic Clock Block Diagram

### 3.2.2 Crobar and Clock Initialization

When the KL10 system is powered up, the EBox clock board must be initialized to a known state. In addition, the device controllers on the EBus must be initialized and a series of MBox, EBox, SBus, and EBus clocks must be generated for various initializtion purposes. First, the power controller asserts CROBAR for approximately 5 seconds. This signal is passed to the clock diagnostic control logic, where it enables the initialization process. The clock diagnostic logic contains a 2-bit source selection register, a 2-bit rate selection register, and various other registers and logic. During power up, the state of these registers is undefined. To avoid an improper source selection, the clock CROBAR signal is used directly to select the 50-MHz oscillator as the clock source to be used during the power up initialization phase (Figure 3-14).

The selected 50-MHz source is now divided down as indicated in Figure 3-15 to provide 25-MHz, 12.5-MHz, and 6.25-MHz free-running clocks.

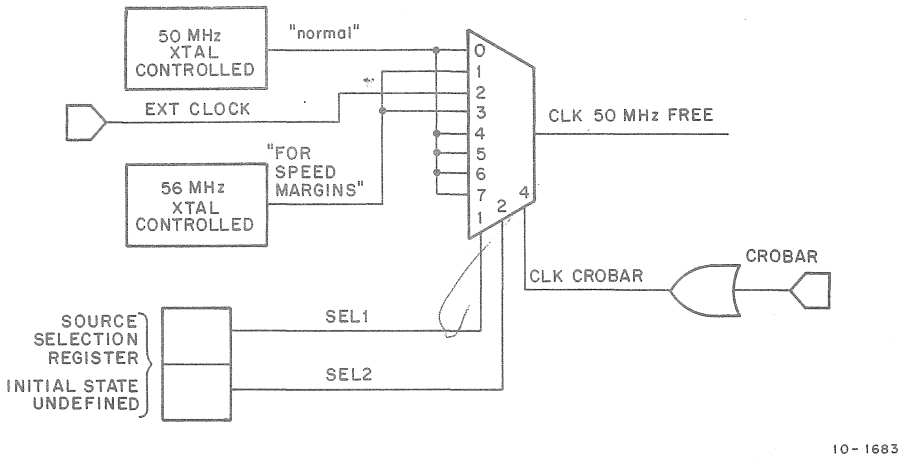



Figure 3-14 Basic Source Selection

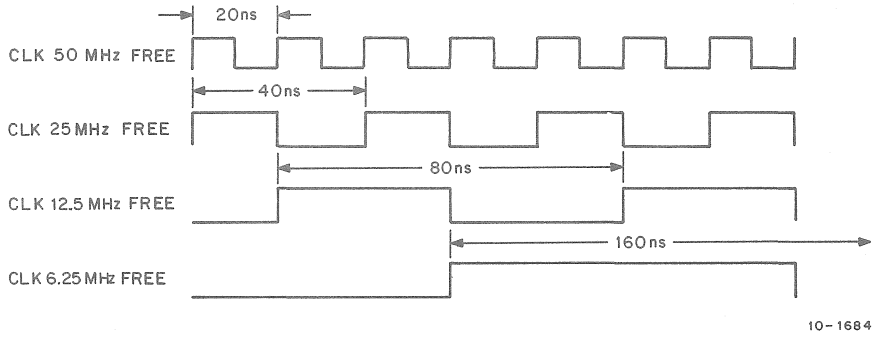
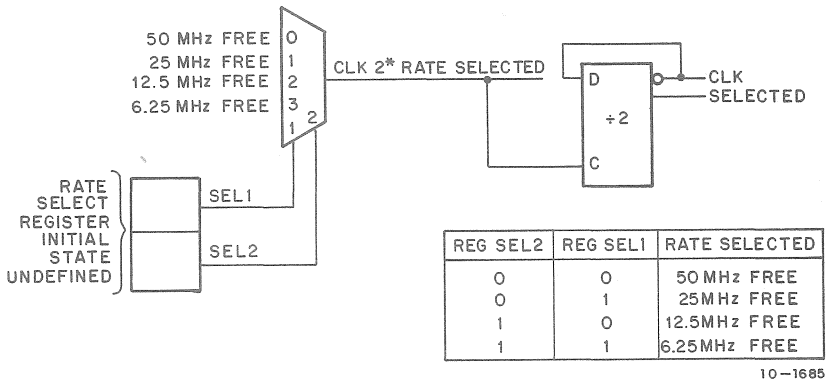



Figure 3-15 Free-Running Clocks

The 50 MHz FREE clock source is next passed to a rate-selectable mixer. However, because the Rate register may initially be in an undefined state, the selected rate is apt not to be the 50 MHz source. This presents no problem because the inputs to the mixer (50 MHz FREE, 25 MHz FREE, 12.5 MHz FREE, or 6.25 MHz FREE) are all even multiples; the rate is not critical during the power up phase of operation. The mixer is shown in Figure 3-16. Its output is labeled 2*Rate Selected, and this output is twice the clock selected frequency.




| REG SEL2 | REG SEL1 | RATE SELECTED |
|---|---|---|
| 0 | 0 | 50 MHz FREE |
| 0 | 1 | 25MHz FREE |
| 1 | 0 | 12.5MHz FREE |
| 1 | 1 | 6.25MHz FREE |

Figure 3-16 Basic Rate Selection

### 3.2.3 EBus Reset

Referring to Figure 3-18, the CLK CROBAR signal enables the counter to subtract one on each 12.5 MHz clock pulse. Once again, the initial state of the counter is undefined. During the crobar period (approximately 5 seconds), the counter is decremented toward zero. When zero is reached, a carry is generated and if CROBAR is false at this time, the -1 function is disabled and the counter is loaded with zeros. This removes ~~EBUS RESET~~. In practice, the counter passes through zero many times until finally CROBAR is removed by the Power Controller logic. Signal EBUS RESET is a 1280 ns square wave.

**3.2.3.1 Initialization Clock Pulse Generation** – As shown in Figure 3-18, CROBAR is shifted four places into the shift register, activating the CLK SS stage. This, with the Clock Selected flip-flop, enables the gated clock. It is this signal (GATED CLK) that becomes the source of the clocks generated via the clock control and EBox Clock Control. When CROBAR is removed, 4 CLK selected pulses later, CLK SS is also removed. The approximate sequence is indicated in Figure 3-17. Figure 3-19 shows the power up timing. Note that this shift register also serves to synchronize CROBAR.

### 3.2.4 EBox Clock Control

The EBox Clock Control provides a source of clocks for the EBox boards together with an MBOX Sync Point (EBOX SYNC), which is always asserted one MBOX Clock prior to the generation of the EBox clock (Figure 3-20).

Depending upon the nature of the EBox cycle (a period extending from the rising edge of one EBox clock to the rising edge of the next), the period between EBOX CLOCK pulses may be extended by some multiple of 40 ns, i.e., 80, 120, 160, 200, etc.

Refer to Figure 3-22; this drawing illustrates the functional structure of the EBOX CLOCK Control. It consists of an MBOX CLOCK counter/marker generator, a clock phase sync detector, an EBox sync source, and an EBox clock source. The CRAM time field (T00, T01) specifies the duration of the EBox cycle (Figure 3-21).

The marker generator consists of a shift register that may be loaded with zeros when EBOX CLK EN is true or have ones shifted in (beginning with the 40-ns stage) for each MBOX CLK generated, as long as EBOX CLK is false. Table 3-6 describes the marker generator.

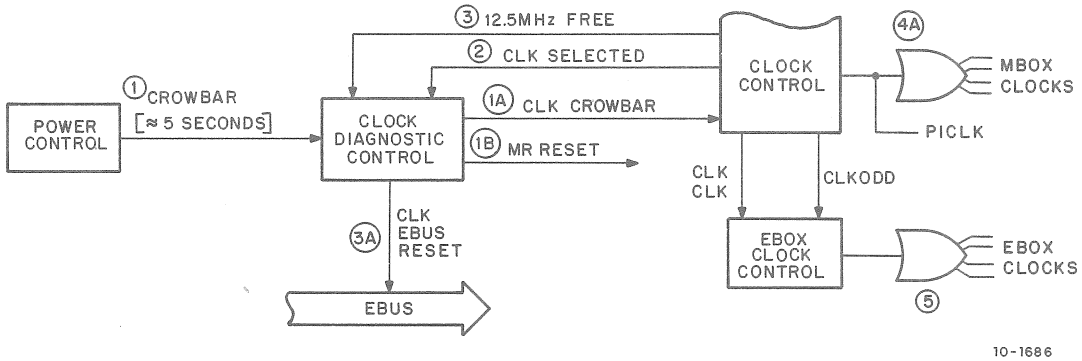



Figure 3-17 Clock Initialization

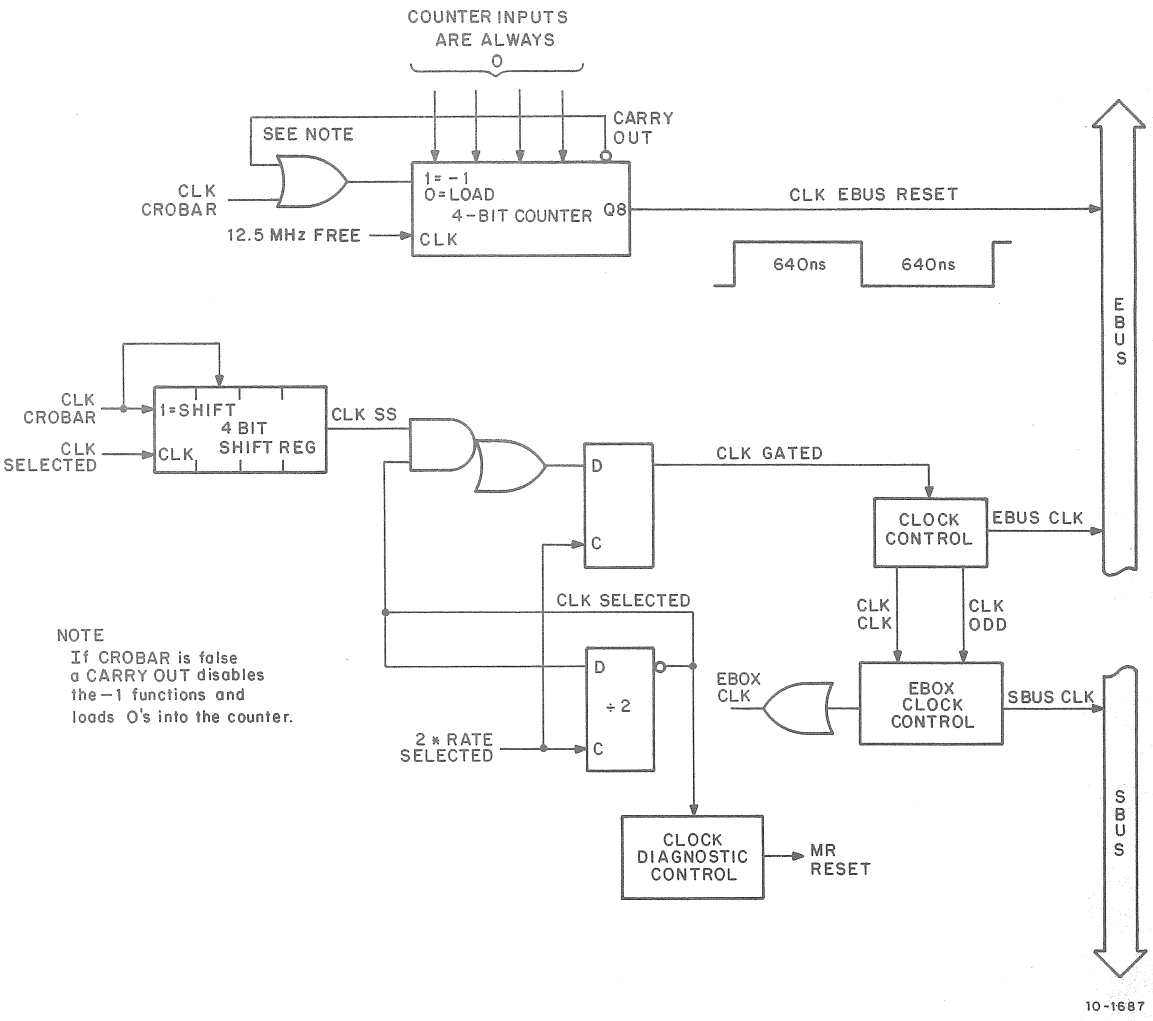



Figure 3-18 EBus Reset and Clock Initialization

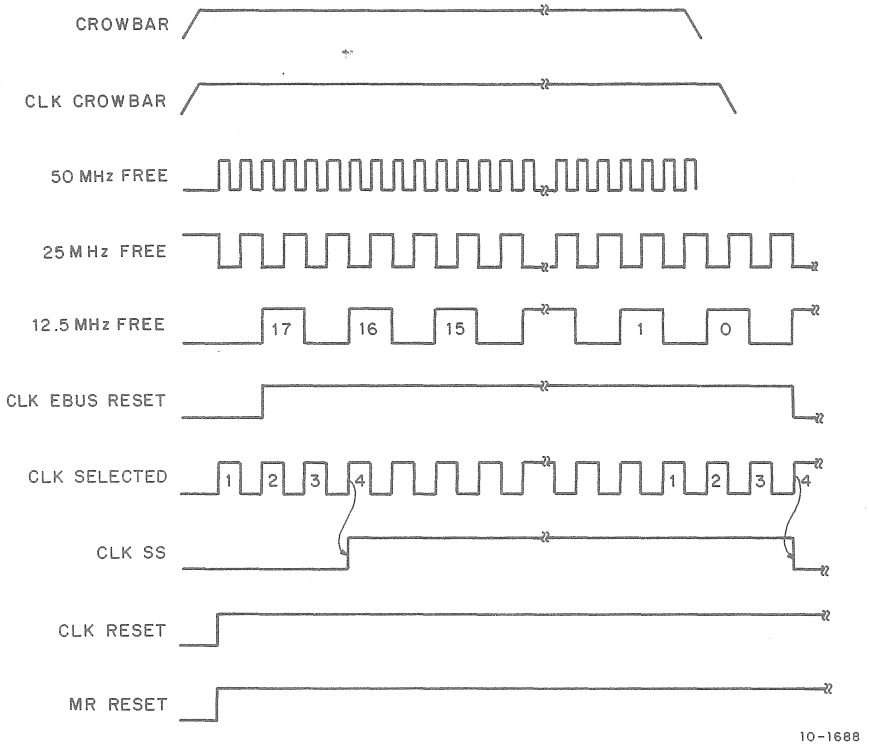



Figure 3-19 Power Up Timing

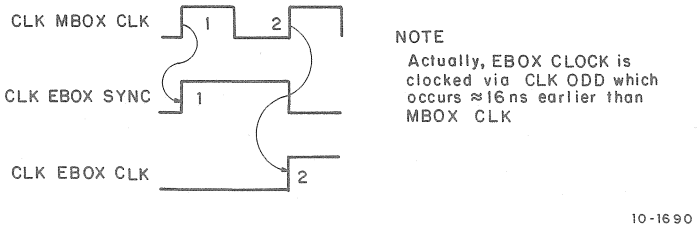



Figure 3-20 Simplified Diagram, MBox Clock, Sync, EBox Clock

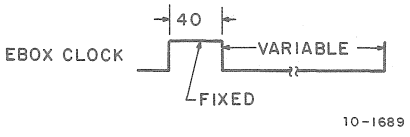



Figure 3-21 EBox Cycle

**Table 3-6 Marker Generator Function**

| T00 | T01 | Duration | MBOX CLK | Marker Generator | | | EBOX CLK EN | EBOX CLK | EBOX SYNC |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 40 ns | 80 ns | 120 ns | | | |
| 0 | 0 | | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 80 | 2 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 120 | 3 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | | 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 160 | 4 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | | 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | | 4 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 200 | 5 | 1 | 1 | 1 | 1 | 0 | 1 |
| x | x | | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

The clock phase sync detector compares the marker generator content with the CRAM time field (loaded at EBOX CLOCK TIME) whenever EBOX CLOCK EN is false. If the marker count compares with the bit combination in the time field, SYNC EN is asserted and the next MBox clock sets EBOX SYNC. EBOX SYNC then enables EBOX CLOCK EN and similarly disables the detector. This completes one cycle.

Note that with MBOX WAIT true, -EBOX CLK EN is also true and EBOX CLK EN is false (Figure 3-22). This enables the MBox clock counter/marker generator to keep shifting 1s from the 40-ns stage toward the 120-ns stage. Similarly, the detector is enabled and when the marker compares with the bit combination in the time field of the CRAM word, SYNC EN will be asserted and remain so until the MBox responds or aborts the cycle. Thus, one MBOX CLK after SYNC EN is asserted, EBOX SYNC will set. In other words, EBOX SYNC is asserted one MBOX CLOCK prior to where EBOX CLOCK would have been asserted.

With SYNC EN true when MBox response is received (Figure 3-22) EBOX CLOCK EN becomes true allowing the marker to reset to 000, and SYNC EN is removed allowing EBOX SYNC to clear on the next MBOX CLOCK. At the same time, EBOX CLK EN becomes true and EBOX SOURCE EN is also true; thus, when EBOX SYNC is cleared, EBOX CLOCK sets (Figure 3-23).

### 3.2.5 Error Detection

Figure 3-24 illustrates the logic that stops all clocks in the event of any of the following:

1. A DRAM parity error occurs.
2. A CRAM parity error occurs.
3. A fast memory parity error occurs.

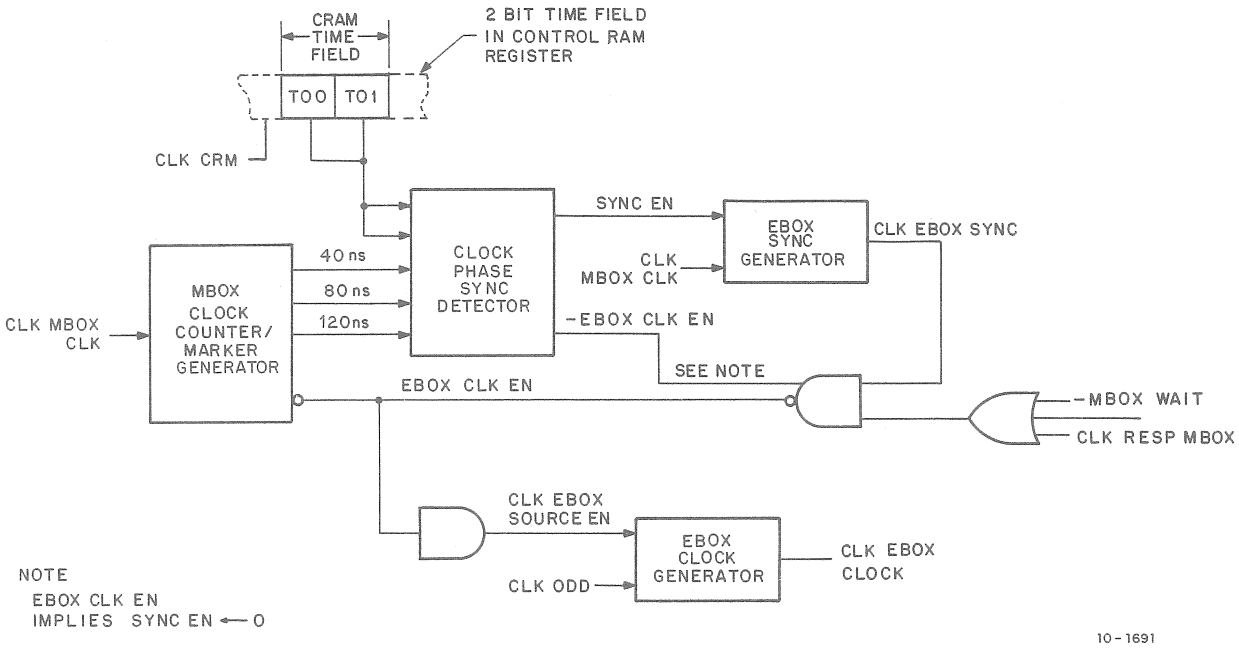



Figure 3-22 EBox Clock Control Block Diagram

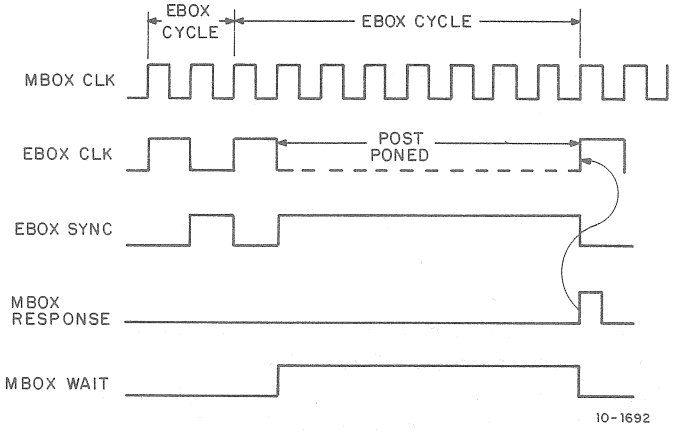



Figure 3-23 Basic MBox Cycle Timing

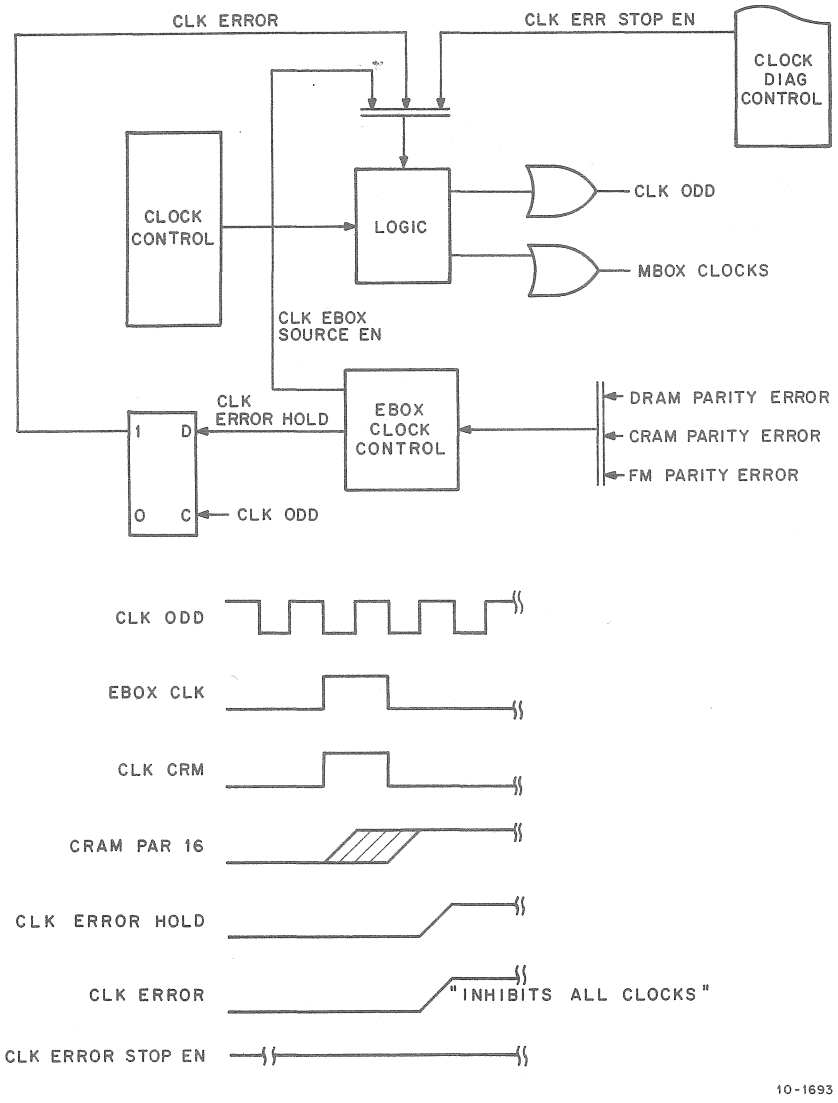



Figure 3-24 Clock Error Stop

The timing shown is for a CRAM parity error. The CRAM register is clocked by CLK CRM; sometime later, the parity network settles and asserts -CRAM PAR 16. This indicates that the CRAM word has dropped or picked up bits and is not correct. The signal -CRAM PAR 16, together with an enable previously set by a diagnostic cycle (CLK CRAM PAR CHECK), enables the generation of CLK ERROR HOLD.

If it is desired to stop on parity errors, CLK ERROR STOP EN must have been set by the console. In this case, on the next occurrence of CLK EBOX SOURCE EN, the CLK ODD gate will be latched false, inhibiting all clocks and freezing the system.

### 3.2.6 Clock Control Logical and Skew Delays

Figure 3-25, illustrates the delays necessary to assure that the proper timing relationship exists between the actual MBOX CLOCKS, EBOX CLOCKS, and the sampling of the CRAM time field. The lumped delay of ≈128 ns consists of fixed logic delays, gate and wire delays. The output is CLOCK ODD and is used to clock a 10141 Shift register, which has a propagation delay of ≈2.65 ns.

**NOTE**
**The times given here are approximate times only.**

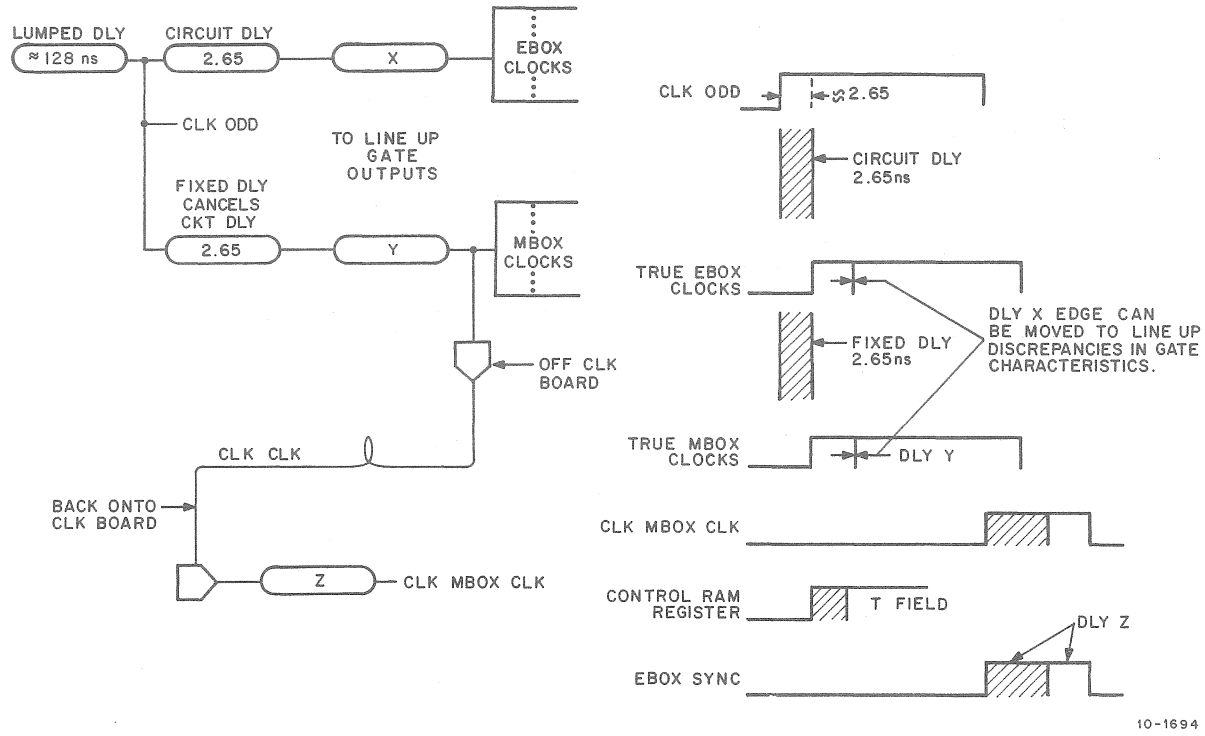



Figure 3-25 Logical Delays and Skew

The output of the Shift register feeds various gates and the various EBox boards receive their clocks from these gates. Delay X allows for lining up the outputs of the gates, "deskewing" the EBox clocks.

The delays are actually etch paths near the fingers on the board and once the delay has been ascertained, a permanent connection is made at the proper point. Figure 3-26 shows the EBox clock fanout; Figure 3-27 shows the MBox clock fanout.

To cancel the effect of the 10141 circuit propagation delay, a fixed 2.65 ns have been inserted in the path between the lumped DLY and the MBOX CLOCKS. Connected in this path also is DLY Y, which performs the same function as DLY X does for the EBOX CLOCKS.

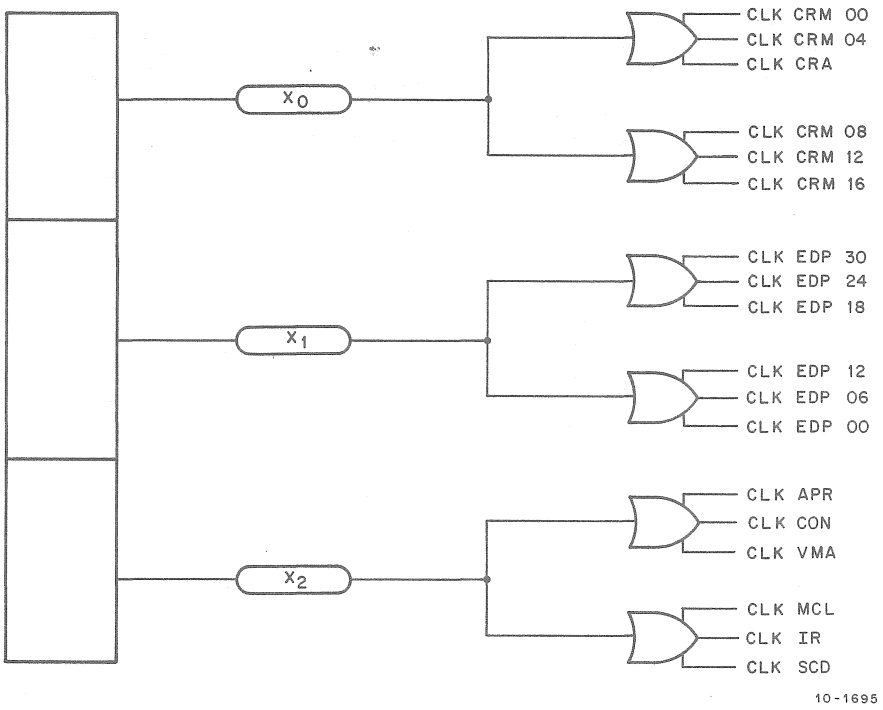



Figure 3-26 EBox Clock Fanout

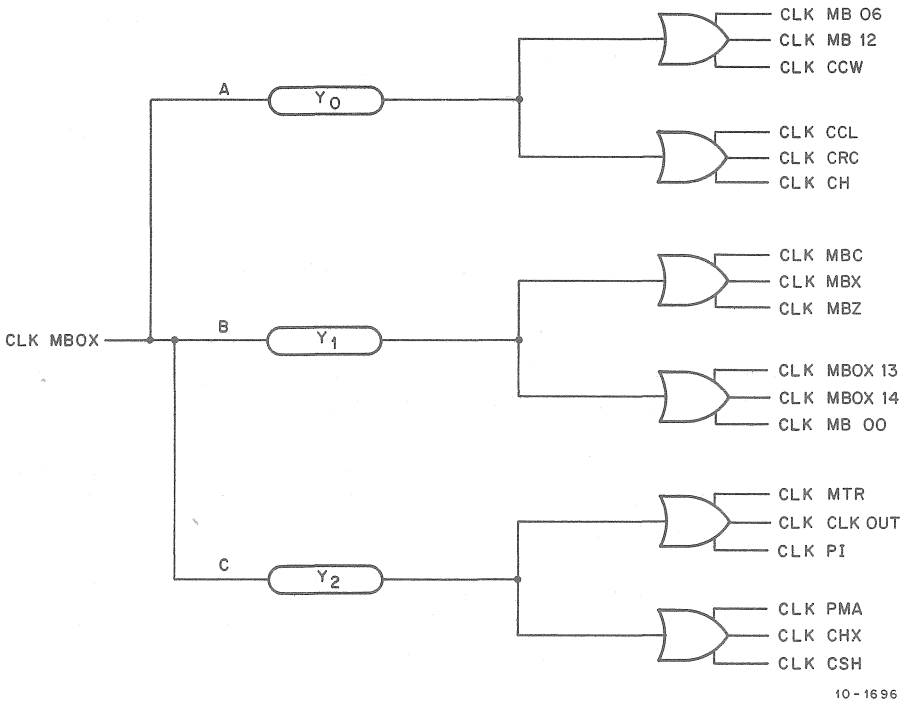



Figure 3-27 MBox Clock Fanout

All the EBOX CLOCKS and the MBOX CLOCKS are lined up leaving the clock board. In order to synchronize the CLK BOARD with the other boards, CLK CLK is passed out through the etch connection on the board. It then reenters the board at DLY 2 where it is deskewed via a coaxial cable, as are all the other CLK signals.

Figure 3-28 illustrates the basic timing for the clock board. Six basic cycles are presented: clock start-up, EBox cycle T = $01_2$, EBox cycle T = $10_2$, EBox cycle including a memory cycle T = $00_2$, EBox cycle T = $00_2$ and finally EBox cycle including a memory cycle and a page fault.

## 3.3 ARITHMETIC PROCESSOR FACILITY

### 3.3.1 Introduction

This facility controls and contains logic relating to the following hardware in the EBox.

Address Break Facility
Arithmetic Processor Status
Processor Identification
Cache Refill RAM Facility
MBox Error Address Register
Fast Memory Addressing and Control

These areas are set up via four KL10 instructions as follows:

DATAO APR – Sets up address break facility.

CONO APR – Sets selected flags in the APR STATUS REG, and/or enables interrupts to occur on selected APR priority interrupt channel.

APRID – Reads the following information from the EBox:

Microcode options
Microcode version number
Hardware options
Processor serial number

RDERA – Reads the ERA register located in the MBox

### 3.3.2 Address Break

One possible use of this hardware in the EBox is associated with the SET BREAK command, which may be issued to the monitor by a user (e.g., during the debugging process). This is primarily useful when the program that is being debugged:

1. Will not fail when DDT has been loaded

2. Destroys DDT when DDT is loaded

3. Destroys the contents of a memory location at an unpredictable point during program execution.

It is possible to break when the specified location is read from, written into, and/or fetched. It is also possible to break on monitor references to items in the user's address space.

Figure 3-29 contains the address break logic. A break may occur at three places in an instruction:

On Instruction FETCH
On DATA FETCH
On DATA WRITE

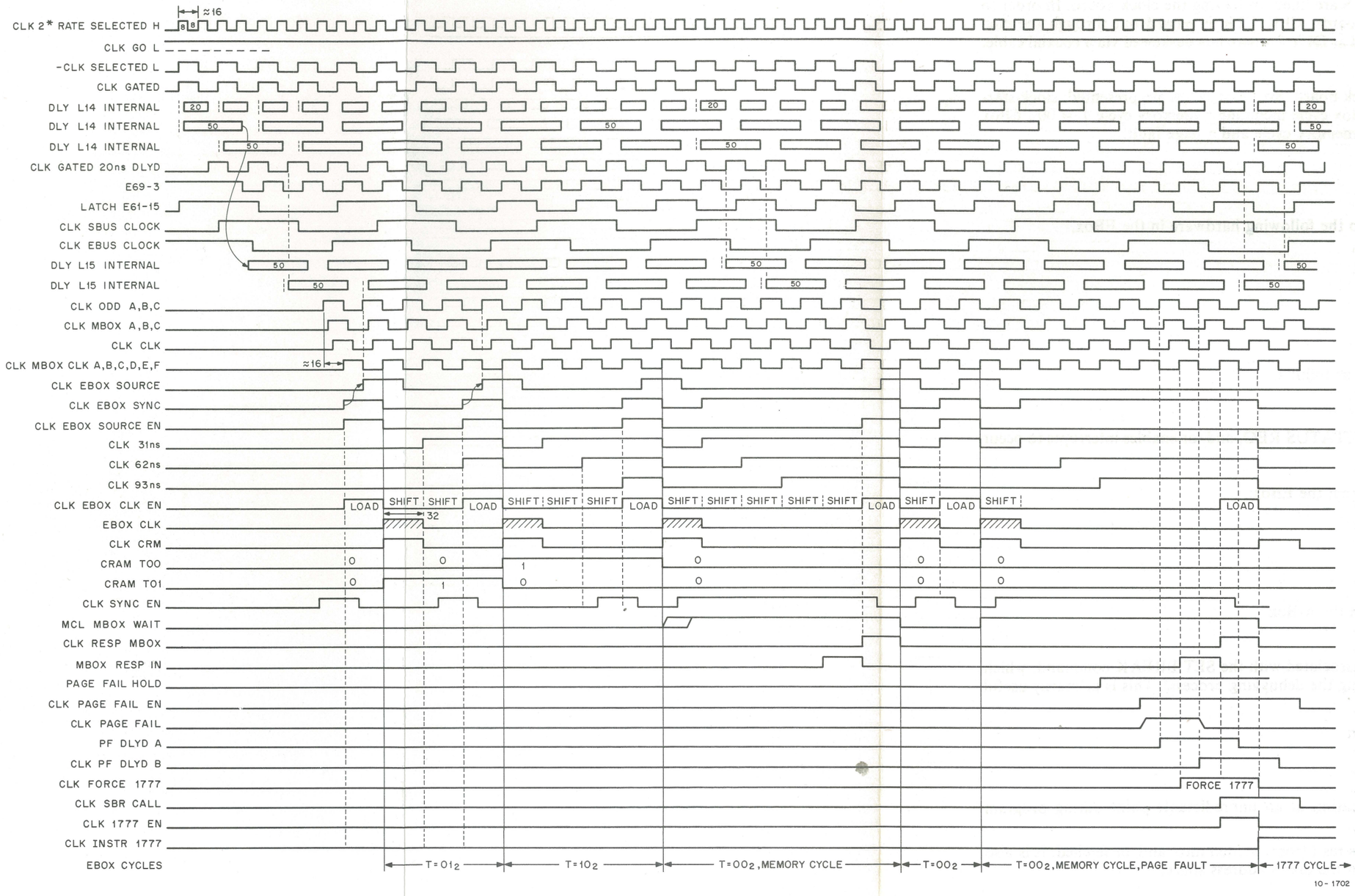



Figure 3-28 Clock Control,
EBox Clock Control Timing

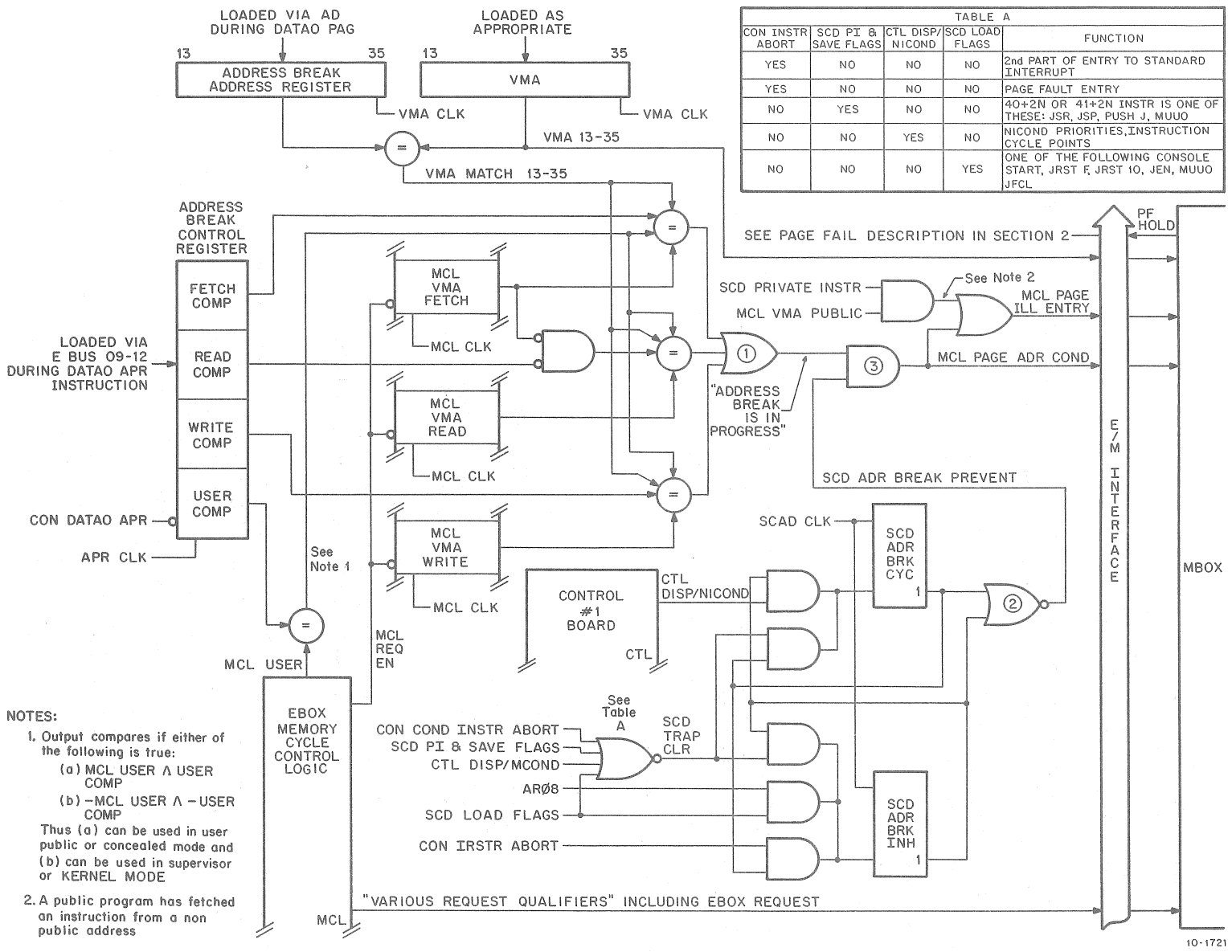



TABLE A

| CON INSTR ABORT | SCD PI & SAVE FLAGS | CTL DISP/ NICOND | SCD LOAD FLAGS | FUNCTION |
|---|---|---|---|---|
| YES | NO | NO | NO | 2nd PART OF ENTRY TO STANDARD INTERRUPT |
| YES | NO | NO | NO | PAGE FAULT ENTRY |
| NO | YES | NO | NO | 40+2N OR 41+2N INSTR IS ONE OF THESE: JSR, JSP, PUSH J, MUUO |
| NO | NO | YES | NO | NICOND PRIORITIES, INSTRUCTION CYCLE POINTS |
| NO | NO | NO | YES | ONE OF THE FOLLOWING CONSOLE START, JRST F, JRST 10, JEN, MUUO JFCL |

NOTES:

1. Output compares if either of the following is true:
   (a) MCL USER ∧ USER COMP
   (b) −MCL USER ∧ −USER COMP
   Thus (a) can be used in user public or concealed mode and (b) can be used in supervisor or KERNEL MODE
2. A public program has fetched an instruction from a non public address

Figure 3-29 Address Break Facility

In addition, the reference may be further qualified to a user or executive reference. The address break conditions are loaded into the EBox hardware by performing a DATAO APR instruction. The left half of (E) specifies the following:

Bit 09: Address Break on FETCH
Bit 10: Address Break on DATA READ
Bit 11: Address Break on DATA WRITE
Bit 12: Address Break on USER REF

The right half of (E) specifies the break address in bits 13–35, where 13–17 represents the virtual section number and 18–35 the virtual page number, line number.

The Address Break Inhibit logic, illustrated in Figure 3-29, may be set up to inhibit an address break by performing any of the following instructions:

JRSTF – JRST2
JEN – JRST 12
JRST 10
MUUO

The PC word provided by these instructions must have bit 8 = 1 to set SCD ADR BRK INH. If a JRSTF is given setting SCD ADR BRK INH, the NICOND Dispatch occurring during the JRSTF transfers the set state of SCD ADR BRK INH into SCD ADR BRK CYC, while clearing ACD ADR BRK INH. Therefore, for the duration of the next instruction, address breaks cannot occur. This is useful, for example, when continuing from an address which subsequently caused an address break. Consider the following example:

```
677/    SETO 3,          ;PUT -1 IN AC3
700/    ADDM 3,300       ;ADD TO TABLE
701/    AOS 700          ;ADD 1 TO TABLE ADR
702/    HRRZ 4,700       ;PUT CURRENT TABLE
703/    CAIE 4,1000      ;ADR IN AC4
704/    JRST 700         ;WHEN IT IS 1000 ALL DONE
```

NOTE

**This sample program illustrates the use of ADR BRK INH and is not meant to be a well-structured program.**

The sample program adds -1 to a table beginning at location $300_8$ and ending at location $1000_8$. A bug exists, however, in this program. Note that the AOS instruction in location 701 is incrementing the table address in the right half of location 700. The problem occurs when the right half of the instruction in 700 becomes 700. At this time, the instruction becomes ADDM 3,700 and this wipes out the instruction in location 700. Several references to location 700 are in the program. First the monitor is requested from a terminal to set ADR break on data write for address 700 to assure that the AOS instruction is working correctly, i.e., attempting a write into 700. The monitor performs a DATAO APR, which sets USER COMP, WRITE COMP, and loads the address break register with 700. At this time, ADR BRK INH is clear and when the EBox performs the write request, the comparator will satisfy the OR gate labeled ① because the following conditions are true:

1. VMA 13–35 = ADR BRK register 13–35
2. MCL VMA WRITE = WRITE COMP
3. MCL VMA USER = USER COMP

At this time, both SCD ADR BRK INH and SCD ADR BRK CYC are clear; therefore, the signals MCL PAGE ADR COND and MCL PAGE ILL ENTRY are asserted together with all other necessary request qualifiers. The MBox detects this condition and places a page fail word in its EBus register (indicating an address break page failure) and asserts PF HOLD to the EBox. The EBox senses this, and enters the microcode page fault handler. Now the EBox flags must be gathered for storage in user process table location 501. Because SCD ADR BRK INH is one of the processor flags, it must be made available; however, at this time it is clear. Regardless of this, the process of obtaining this flag will be discussed. Upon entry to the microcode, CON INSTR ABORT is generated to cause proper termination of the faulting instruction. Referring to Figure 3-29, CON INSTR ABORT enables SCD TRAP CIR, which breaks the recirculation paths for both SCD ADR BRK INH and SCD ADR BRK CYCLE; it also transfers the state of SCD ADR BRK CYC into SCD ADR BRK INH. This makes the flag available for storage in 501. The page fault handler reads the MBox EBus register and stores a page fail word in user process table location 500, stores the flags PC word (PC is now 701) in 501 and then fetches a new PC word from user process table location 502. The processor now enters Execute mode and handles the page failure appropriately.

Eventually, after evaluating the page fault word in 500 and other data, the monitor informs the user at his terminal that a write was attempted to location 700. If after giving the problem some thought, the user requests a break on the same address for write but now suspects that somehow the instruction in 700 is being overwritten by itself, the break can be inhibited. Now the monitor wishes to continue the program by performing the entire AOS instruction to ascertain that it works but also must avoid thewrite page fault associated with this instruction.

The monitor can perform a JRSTF instruction that sets ADR BRK INH and restores the old PC of 701 for the AOS instruction via user process table location 501. Referring to Figure 3-29, during the execution portion of JRSTF, SCD LOAD flag sets SCD ADR BRK INH. During the JRSTF instruction NICOND Dispatch occurs and transfers the set state of SCD ADR BRK INH into the BRK CYCLE flip-flop while clearing SCD ADR BRK INH. The AOS instruction is successfully fetched from 701 and the "AOS write reference" to 700 is prevented from causing MCL PAGE ADR COND because this is blocked by SCD ADR BREAK COND (L). The next NICOND Dispatch clears SCD ADR BRK CYCLE, enabling the ADR BREAK to occur if a write is performed to 700. Eventually, through many tries, the overwrite of the instruction in 700 will be detected by this method. Note this is only a simple example and is not necessarily a practical one.

**3.3.2.1 Address Break INH and Saving Flags** – The signal CON COND INSTR ABORT is generated by the microcode whenever external conditions require the microcode to abort a partially completed instruction. If this occurs during an address break cycle, this signal copies the state of SCD ADR BRK CYC back into SCD ADR BRK INH, thus making it available to save as a bit in the flag's PC word.

**3.3.2.2 Address Break INH and Loading Flags** – SCD LOAD FLAGS can be generated in a number of ways: JRSTF, JRST10, JEN, JRST, and MUUO can set SCD ADR BRK INH. The 10-11 interface can place the flags PC word in AR and perform a console start. This causes the microcode to generate SCD LOAD FLAGS. During a JFCL instruction, the flags are read and the specified flags cleared. Then the microcode reloads the flags using the signal SCD LOAD FLAGS.

### 3.3.3 Arithmetic Processor Status Register

This facility enables special internal conditions to signal the monitor on a priority interrupt channel assigned to the processor. Condition I/O instructions are used to control the appropriate flags and to inspect the conditions of interest.

The arithmetic processor status register consists of two 8-bit registers and associated control logic. One register receives the error or status signals and the other register enables or inhibits the generation of an interrupt when one or more of these error or status flags sets.

Figure 3-30 provides the basic format for the CONO APR word, the basic organization of the error or status flag and the interrupt enable or inhibit for the two registers. In addition, the bit assignments are provided in two tables, as well as the source of the error or status signals available to set the appropriate flags in the APR register.

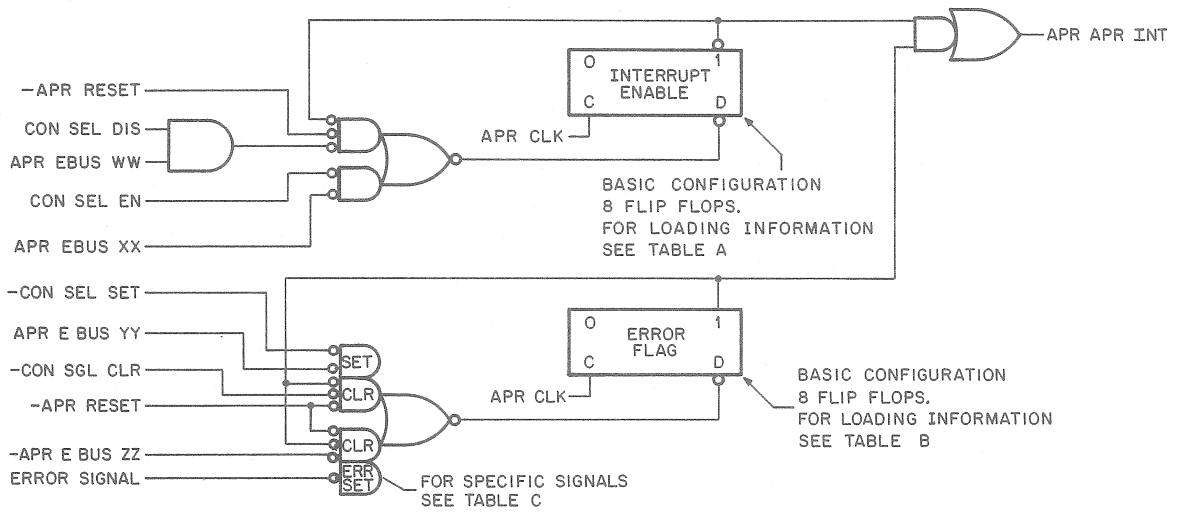



TABLE A

| E BUS BIT WW | CON SEL DI'S | CON SEL EN | E BUS BIT XX | INTERRUPT EN SETS | INTERRUPT EN SETS |
|---|---|---|---|---|---|
| 02 | | YES | 06 | S BUS ERR | |
| 03 | YES | | 06 | | S BUS ERR |
| 02 | | YES | 07 | NXM ERR | |
| 03 | YES | | 07 | | NXM ERR |
| 02 | | YES | 08 | I/O PF ERR | |
| 03 | YES | | 08 | | I/O PF ERR |
| 02 | | YES | 09 | MB PAR ERR | |
| 03 | YES | | 09 | | MB PAR ERR |
| 02 | | YES | 10 | C DIR P ERR | |
| 03 | YES | | 10 | | C DIR P ERR |
| 02 | | YES | 11 | S ADR P ERR | |
| 03 | YES | | 11 | | S ADR P ERR |
| 02 | | YES | 12 | PWR FAIL | |
| 03 | YES | | 12 | | PWR FAIL |
| 02 | | YES | 13 | SWEEP DONE | |
| 03 | YES | | 13 | | SWEEP DONE |

TABLE B

| E BUS BIT YY | CON SEL SET | CON SEL CLR | E BUS BIT ZZ | ERROR FLAG CLRS | ERROR FLAG SETS |
|---|---|---|---|---|---|
| 04 | | YES | 06 | S BUS ERR | |
| 05 | YES | | 06 | | S BUS ERR |
| 04 | | YES | 07 | NXM ERR | |
| 05 | YES | | 07 | | NXM ERR |
| 04 | | YES | 08 | I/O PF ERR | |
| 05 | YES | | 08 | | I/O PF ERR |
| 04 | | YES | 09 | MB PAR ERR | |
| 05 | YES | | 09 | | MB PAR ERR |
| 04 | | YES | 10 | C DIR P ERR | |
| 05 | YES | | 10 | | C DIR P ERR |
| 04 | | YES | 11 | S ADR P ERR | |
| 05 | YES | | 11 | | S ADR P ERR |
| 04 | | YES | 12 | PWR FAIL | |
| 05 | YES | | 12 | | PWR FAIL |
| 04 | | YES | 13 | SWEEP DONE | |
| 05 | YES | | 13 | | SWEEP DONE |

TABLE C

| ERROR FLAG | ERROR SIGNAL |
|---|---|
| S BUS ERR | MBOX S BUS ERR |
| NXM ERR | MBOX NXM ERR |
| I/O PF ERR | APR SET I/O PF ERR |
| MB PAR ERR | MBOX MB PAR ERR |
| C DIR P ERR | CSH ADR PAR ERR |
| S BUS ADR P ERR | MBOX ADR PAR ERR |
| PWR FAIL | PWR WARN |
| SWEEP DONE | APR SWEEP BUSY ∧ -APR SWEEP BUSY EN |

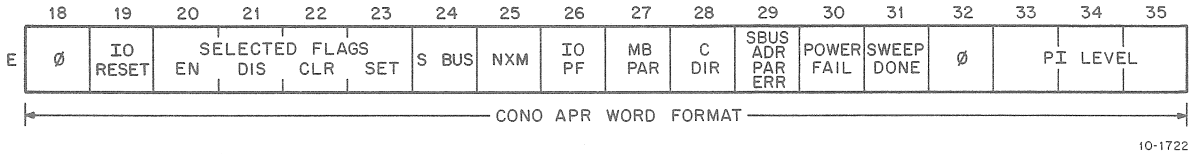



Figure 3-30 APR Register and Interrupt Enables

The basic organization of the APR is illustrated in Figure 3-31. The register is broken down into four sections based on the origin of the error. The first five flags set as a result of an error condition involving some memory activity. Three of the flags: [SBus Error, Nonexistent Memory (NXM) Error, and S ADR Parity Error] originate in the memory adapter (DMA). The remaining two originate in the MBox. The flag IN-OUT PAGE FAIL (IOPF) sets because of an external stimulus, but the actual setting takes place by the microprogram, in response to a page failure that occurred during a priority interrupt. The power failure flag sets when the power controller detects a low voltage condition. The sweep done flag signals the completion of a cache sweep operation. This operation is the result of performing a sweep instruction.

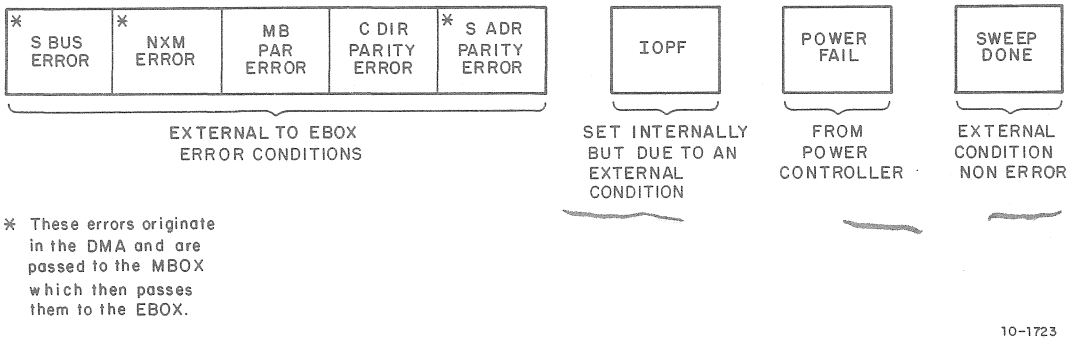



Figure 3-31 APR Register Breakdown

Once again referring to Figure 3-30, to enable interrupts for any or all of the eight conditions, a CONO APR is performed with bit 20 equal to 1 and ones in bits 24 through 31 for the desired flags. Similarly, to disable interrupts for any of the eight flags, which have previously been enabled, place bit 21 equal to 1 and ones in bits 24 through 31 for the flags to be disabled. This means that once the processor has been powered up, and providing a power failure condition has not occurred, that once an interrupt enable has been set, it must be specifically cleared as indicated above.

Any of the eight flags can be selectively set or cleared by placing bit 23 or 22 on, respectively, together with those bits in 24–31 to be changed.

**3.3.3.1 SBus Errors** – Two error lines are available from the DMA to the MBox. These are SBUS ADR PAR ERR and SBUS ERR. If the DMA starts a memory cycle and also detects bad address parity, it sends SBus Acknowledge (SBUS ACKN) to the MBox, acknowledging receipt of the address and within 125 ns transmits SBUS ADDRESS PAR ERR. The MBox now latches the error address register (ERA), which contains the address in question and additional bits which specify information associated with "data parity error conditions." These two bits specify which of the four memory buffers (MBs) the parity error is associated with. The address used to address memory specifies which word is to be transmitted (for a write) or received (for a read) first. This information is contained in bits 34 and 35 of the address. If, for example, the address in the ERA is 101 [bit 34(0) and bit 35(1)] and the address in the PMA used to address memory is 100, the indication is that the word requested by the EBox, for example, was not the word actually causing the data parity error. Thus, in this example, the EBox requested the contents of location 100, received it, and how, while fetching a word from 101 (of a quadword group), an error occurred associated with that word.

In addition, a 3-bit code identifies the origin of the data in the memory buffer register and indicates the type of reference, i.e., read, write, etc. As the MBox latches the ERA, it transmits MBOX RESPONSE IN and MBOX S ADR PARITY ERROR to the EBox. MBOX S ADR PARITY ERROR occurs concurrently, with an MBox clock and, therefore, on the next MBox clock (that will be also an EBox clock) APR S ADR PARITY ERROR sets. Providing the SBUS ADR PARITY ERROR INTERRUPT enable is set, an interrupt will be requested on the APR channel. In addition, to prevent the MBox error condition from being changed, the APR error flag which sets is sent over the E/M interface to recirculate the MBOX SBUS ADR PARITY ERR COND; also, APR ANY EBOX ERR sets and is passed to the MBox to hold the ERA. As a result of the interrupt, the monitor determines that the APR was the source of the interrupt via a condition I/O instruction (CONSO, CONSZ, CONI, APR), make a determination, and finally clear the error flag, releasing the MBox ERA and associated error logic.

**3.3.3.2 Nonexistent Memory** – Each time the EBox makes a memory reference, the MBox interprets the request qualifiers and performs all the steps necessary to satisfy the request. A core memory reference must be issued by the MBox in order for NXM to occur. When the MBox issues a memory request to read or write a word to core memory via the memory adapter (DMA), it starts a timeout (32 $\mu$s) and waits for SBUS ACKN from the DMA indicating acceptance of the request and address. If 32 $\mu$s elapse and SBUS ACKN is not forthcoming, the MBox sets MEM ERR (Figure 3-32).. An additional 32 $\mu$s elapses and if SBUS ACKN has not been received by the MBox, MBox NXM error is asserted together with MBOX RESP IN.

Referring to Figure 3-33, MBOX NXM ERROR is loaded into the APR register with APR CLK. If the NXM ERR interrupt enable is set, APR INTERRUPT is asserted to the PI Board. To preserve the ERA and NXM ERROR in the MBox, the APR NXM flag is recirculated back to the MBox. In addition, PAR ANY EBOX ERR sets, holding the ERA information in the ERA register.

**3.3.3.3 Other External Errors** – Referring to Figure 3-34, all five external error conditions set the appropriate APR ERROR flag and request interrupts (if enabled) on the error channel assigned. Also, all the indicated error flags recirculate to the MBox and all cause APR ANY EBOX ERROR to set, preserving the contents of ERA. Of the five errors, one, MB PAR ERROR, is handled as if it were a page fault. That is, it causes control to be passed to the microcode page fault handler, where it is evaluated. The status word is obtained from the ERA in the MBox. The format for this word is initially as indicated in Figure 3-35.

The page fault microcode places a code in bits 0–5 of $26_8$ and places the virtual address for the reference in bits 13–35 where bits 13–17 are 0 for KI paging mode; this word is stored in user process table location 500. The remainder of the operation is identical with that for a page failure and is covered in Section 2.

**3.3.3.4 Input/Output Page Failure Error** – During a priority interrupt [PI CYCLE (1)], page failures are not expected to occur for interrupt instruction fetches or PI dispatches. This is regarded as a fatal error, and it causes an interrupt on the assigned APR error channel. The page fault handler sets IOPF in the APR register and then dismisses the interrupt. The PC is placed in VMA and an instruction fetch begins while waiting for the PI system to honor the interrupt for the APR.

**3.3.3.5 Power Fail** – The power controller asserts the signal POWER WARN whenever the power supplies reach a marginal value. This results in the setting of the APR POWER FAIL flag and requests an interrupt on the APR error channel.

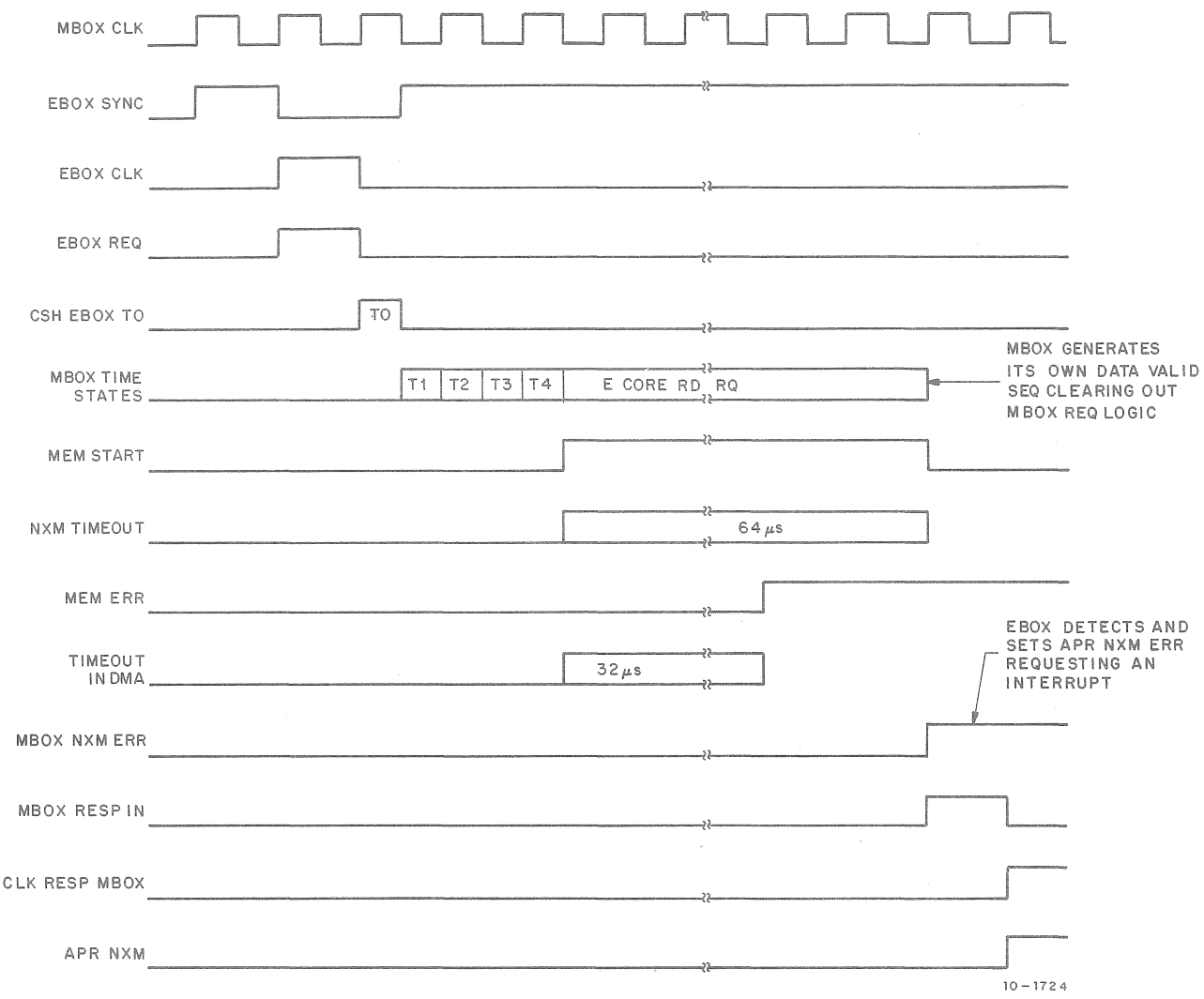



Figure 3-32 NXM Timing Overview

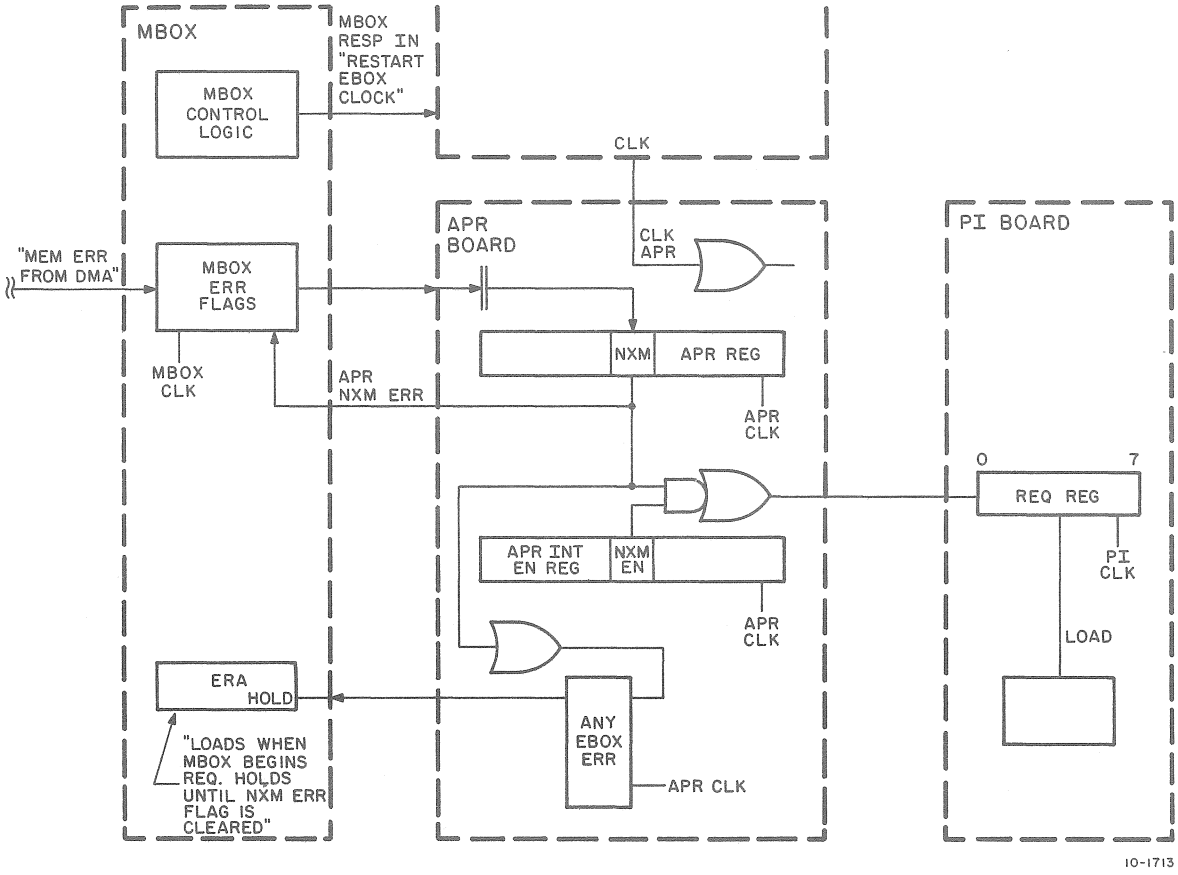



Figure 3-33 NXM Error Overview

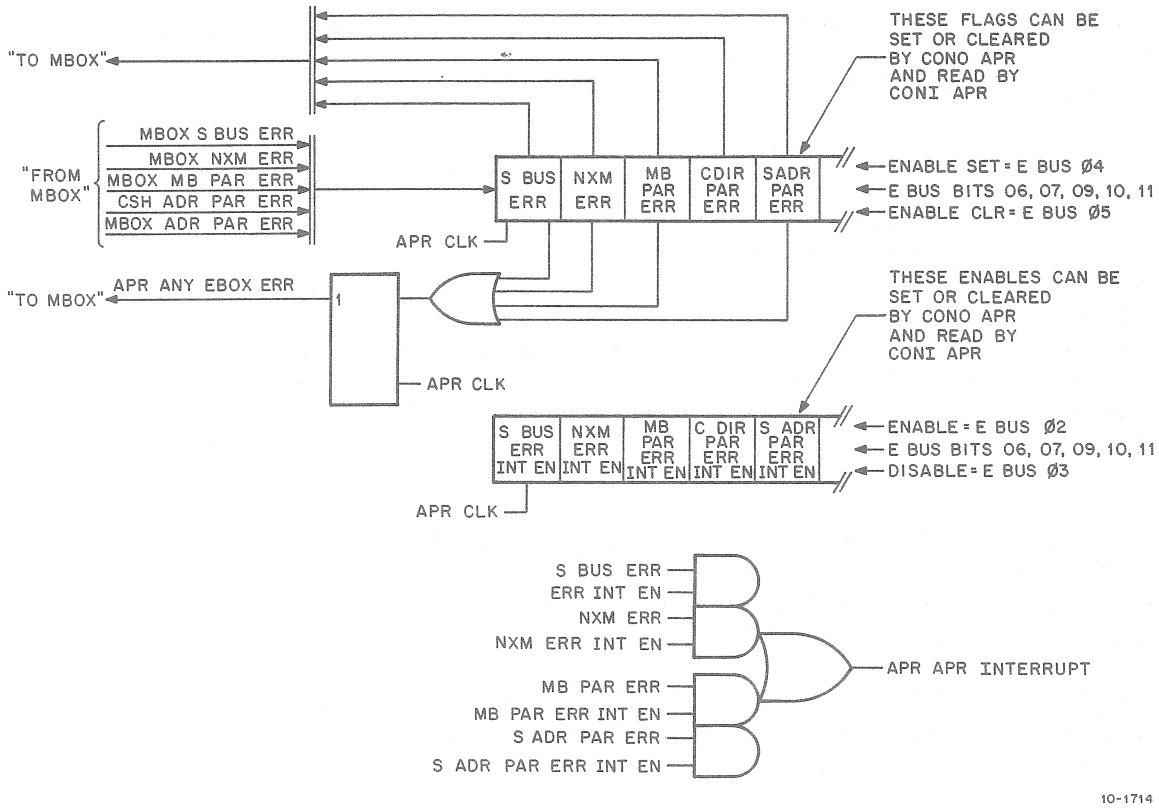



Figure 3-34 External Error Conditions (MBox, SBus)

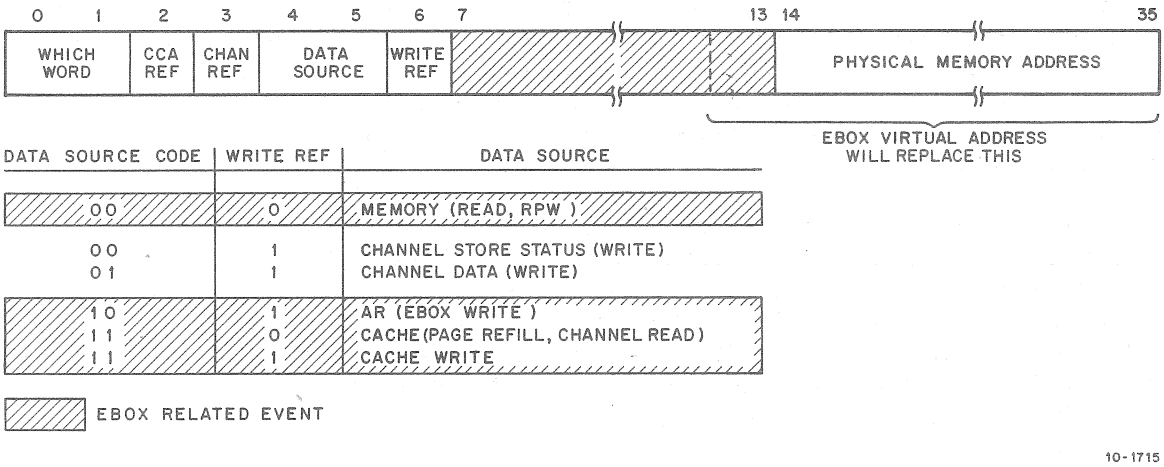



Figure 3-35 ERA Word

**3.3.3.6 SWEEP and SWEEP DONE** – The MBox contains a section of logic called the Cache Clearer (CCA). This is addressed as if it were a device (014), using I/O instructions. Six operations may be initiated. These are listed in Table 3-7.

**Table 3-7 CCA Summary**

| New Mnemonic | Old Mnemonic | Function |
|---|---|---|
| SWPIA | DATAI CCA | Invalidate all cache data; do not update core. |
| SWPVA | BLKO CCA | Sweep cache, validate core, leave cache valid. |
| SWPUA | DATAO CCA | Unload all pages updating core; validate the cache. |
| SWPIO | CONI CCA | Invalidate one page of the cache; do not validate core. |
| SWPVO | CONSZ CCA | Sweep cache, validate one page of core, leave cache valid. |
| SWPUO | CONSO CCA | Unload one page, update core, invalidate the cache. |

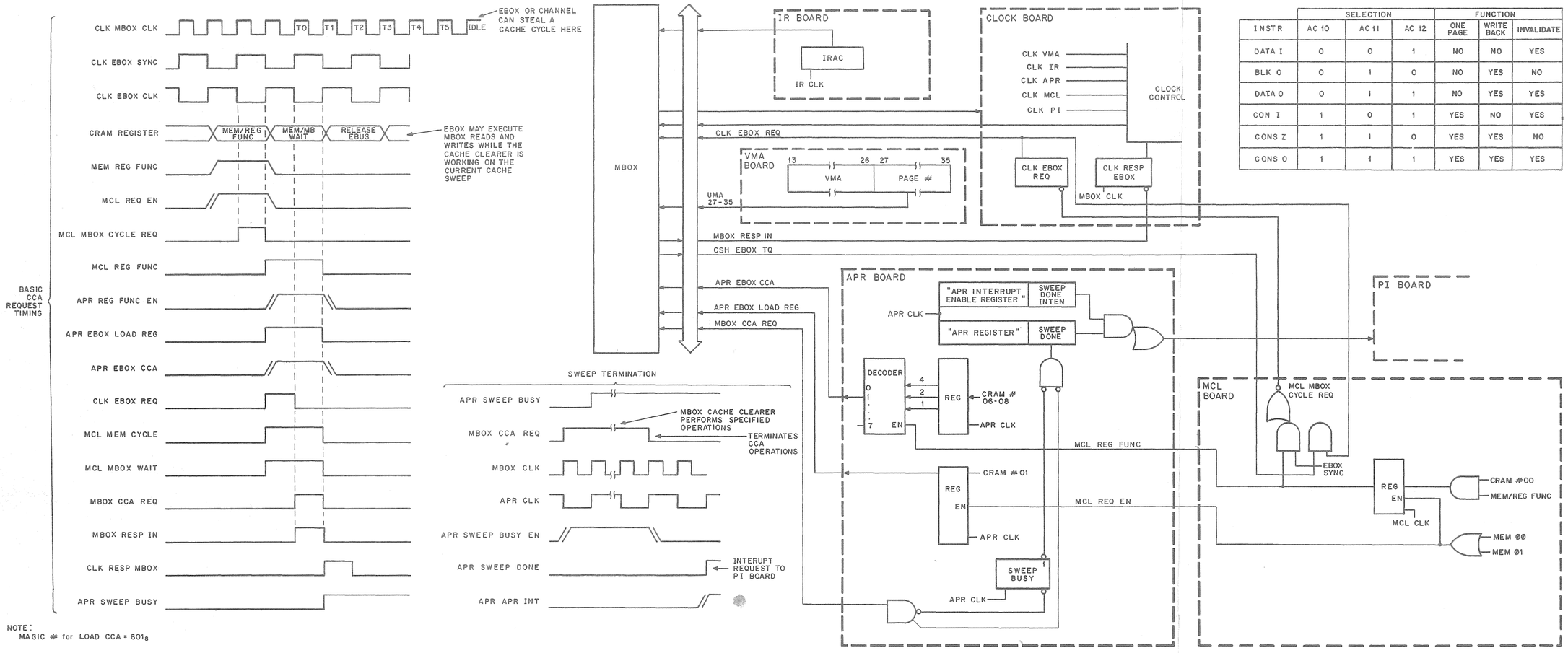
To request CCA cycles from the MBox as a function of one of the six instructions in Table 3-7, the EBox places the virtual page number into VMA 27–35, verifies that the performance of the Sweep instruction (which is privileged) is legal in the current mode of the processor and then either begins the operation or, if illegal, performs an MUUO.

Figure 3-36 illustrates the various logic associated with the sweep operation. Three basic operations can be specified in various combinations by the six types of Sweep instructions. These are illustrated in Figure 3-36 in the table at the upper left.

In the cache, associated with each word of a four word block (quadword), are two bits labeled valid and written. If the valid bit is off for any of the four words, these words are considered to contain incorrect data and, if referenced (for example by the EBox), the words must be fetched from main memory. Similarly, if the written bit is on for any of the valid words, these words contain different data than the copy in main memory and the cache copy is correct. At some point, the written words must be flushed from the cache into core memory. On power up, the cache must be invalidated, clearing all the entries. For this case, the DATAI instruction is performed to device CCA. Because AC bit 10 is 0, the MBox, upon receiving the EBox request and appropriate qualifiers (APR EBOX CCA and APR EBOX LOAD register), will invalidate the entire cache. Similarly, because AC bit 11 is 0, the MBox disregards the written words and no writebacks are performed to core memory. Finally, AC bit 12 is 1, which specifies invalidation.

Referring to Figure 3-36, IRAC contains the AC field 9–12 of the instruction. The microcode executor sets up the request utilizing the MEM field function MEM/REG FUNC together with the magic number field coded as LOAD CCA ($601_8$). To follow the memory request, it is best to refer to Figure 2-98 which can be found in Subsection 2.7.2.5. Note that on Figure 3-36 MEM/REG FUNC (07) has bit 01 equal to 1 and this generates MCL REQ EN. This signal is used to enable the various registers involved in the EBox request to load with the appropriate information prior to latching the VMA. The following conditions set up for the CCA request.

| Controlling Signal(s) | Signal Generated |
|---|---|
| MEM/REG FUNC | MCL REQ EN |
| MCL REQ EN∧ MEM/REG FUNC ∧ CRAM#00 | MCL REG FUNC |
| MCL REQ EN∧ MCL REG FUNC ∧CRAM#01 | APR EBOX LOAD REG |
| APR REG FUNC EN∧ CRAM#06–08 = 1 | APR EBOX CCA |
| MCL REG FUNC∧CLK EBOX SYNC | MCL MBOX CYCLE REQ |




| INSTR | SELECTION AC 10 | AC 11 | AC 12 | FUNCTION ONE PAGE | WRITE BACK | INVALIDATE |
|---|---|---|---|---|---|---|
| DATA I | 0 | 0 | 1 | NO | NO | YES |
| BLK O | 0 | 1 | 0 | NO | YES | NO |
| DATA O | 0 | 1 | 1 | NO | YES | YES |
| CON I | 1 | 0 | 1 | YES | NO | YES |
| CONS Z | 1 | 1 | 0 | YES | YES | NO |
| CONS O | 1 | 1 | 1 | YES | YES | YES |

Figure 3-36 Sweep Logic

The basic timing for the CCA request as well as CCA termination is illustrated in Figure 3-36. The VMA must contain the virtual page number in VMA 27–35 for CONI, CONSZ, or CONSO CCA operations. In the current example (DATAI CCA), the MBox cache clearer does not use this information because the entire cache is to be invalidated. However, the cache clearer has an associated register that is loaded by the MBox with VMA 27–35. IRAC bits 10–12 are similarly loaded into the MBox control logic that directs the type of operation carried out. Each time a CCA cycle is completed in the MBox, an idle period occurs where the channels or EBox can obtain an MBox cycle. The EBox can continue to execute instructions but must guard against defeating the purpose of the Sweep operation, i.e., write new data into already swept words in the cache. Summarizing, three of the six instructions operate on one page of the cache (512 words). For these three instructions a different set of sweep functions is available; these are: invalidate, writeback all written words in the specified page, or perform both. Similarly, three instructions operate on the entire cache (2048 $_{10}$ words) but the operations are the same as with the other three. In all cases, the EBox performs an EBox Request providing the appropriate qualifiers and the VMA contains (in bits 27–35) the page number. The MBox loads its CCA register and then asserts MBox CCA Request together with MBOX RESPONSE IN. Now the EBox is free to perform operations while waiting for SWEEP DONE to generate an APR interrupt. If a second sweep instruction is started by the EBox before the first is completed, the MBox begins the second sweep just as it would another instruction; however, it reloads the CCA register with the new information supplied by the second sweep instruction and does not complete the first.

### 3.3.4 Processor Identification

The processor identification consists of four parts:

Microcode options
Microcode version number
Hardware options
Processor serial number

This information is obtained by performing what was traditionally a BLKI APR, now called APRID. The format is illustrated in Figure 3-37.

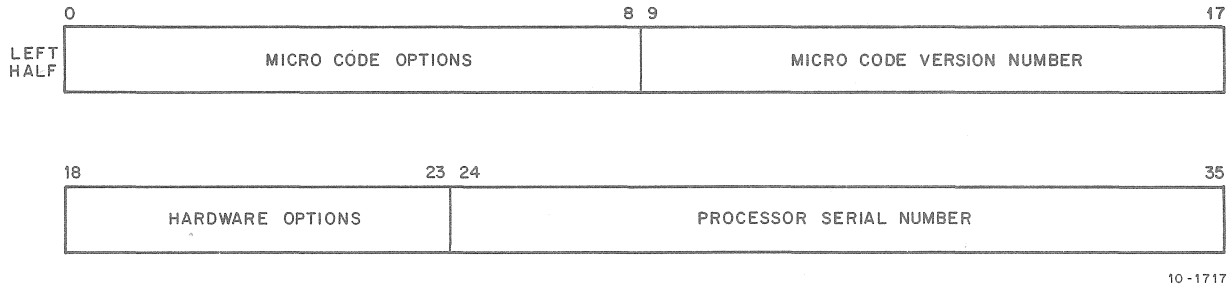



Figure 3-37 APRID Format

This is not strictly a visible hardware function, but rather a combination of microcode and hardware. The microcode for a given version is coded in such a fashion that the version number is obtained utilizing the magic number field and the function AR00–08← number. The microcode obtains the processor serial number that is hardwired to the 0 input of the ADXB mixer and places it in AR. Next, the microcode version number is obtained and adjusted as follows. The serial number in AR is copied to BR and the version number is loaded into AR00–08; next, the ARX. At this time the BR, AR, and ARX are as indicated in Figure 3-38.

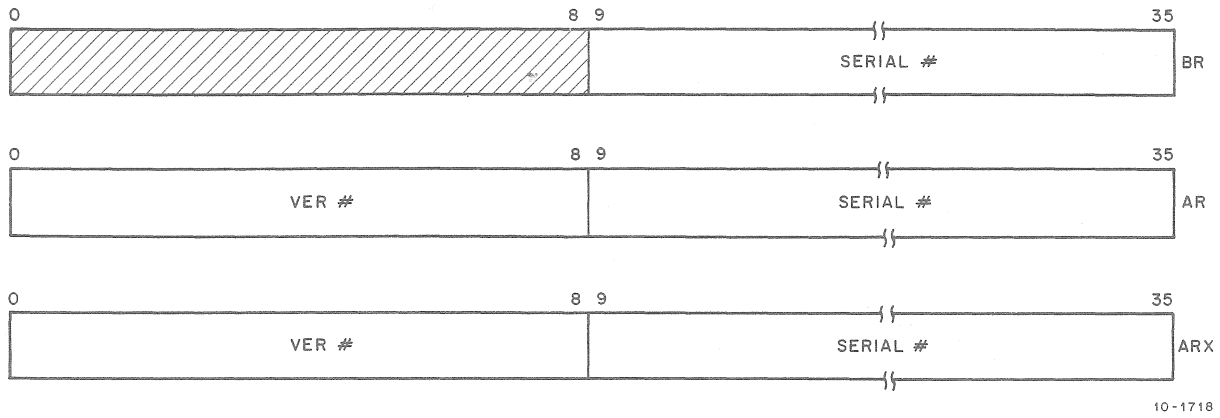



Figure 3-38 Alignment Step 1

The shift counter is loaded with $9_{10}$ and now the combined AR and ARX are shifted left 9 places with the result placed in AR as indicated in Figure 3-39.

The version number is placed in AR 9–17, the serial number in AR 24–35, and the resulting word is stored in location E.

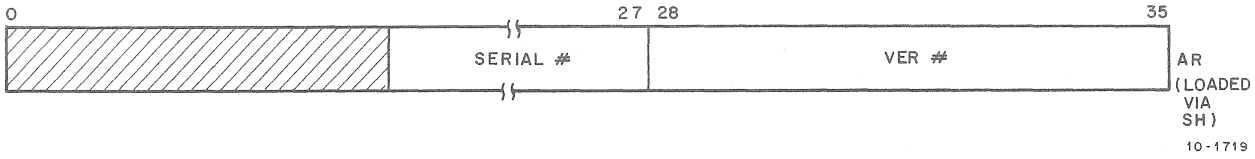



Figure 3-39 Alignment Step 2

### 3.3.5 Cache Refill RAM Facility

The cache refill RAM in the MBox must be loaded with a set of bit patterns called the refill algorithm. This RAM is used by the MBox with a use table and other associated logic to manage the cache refill operation. Generally speaking, when the cache fills up with words, it becomes necessary to displace old words for new ones. It is desirable to displace the words used most infrequently. To do this, an algorithm was developed that specifies which word is to be displaced each time a refill cycle must write into the cache. Figure 3-40 illustrates the basic structure of the MBox Refill RAM and also indicates the format of the effective address provided by the BLKO APR instruction (new mnemonic WRFIL). The microcode executor is entered with the effective address (E) in AR. Because the instruction is privileged, legality is checked first. If the instruction is legal for the current mode of the processor (Kernel or User with IOT set), the instruction is performed; otherwise, an MUUO is effectively performed with the illegal instruction stored in the user process table location 424 in the place where the MUUO is normally stored.

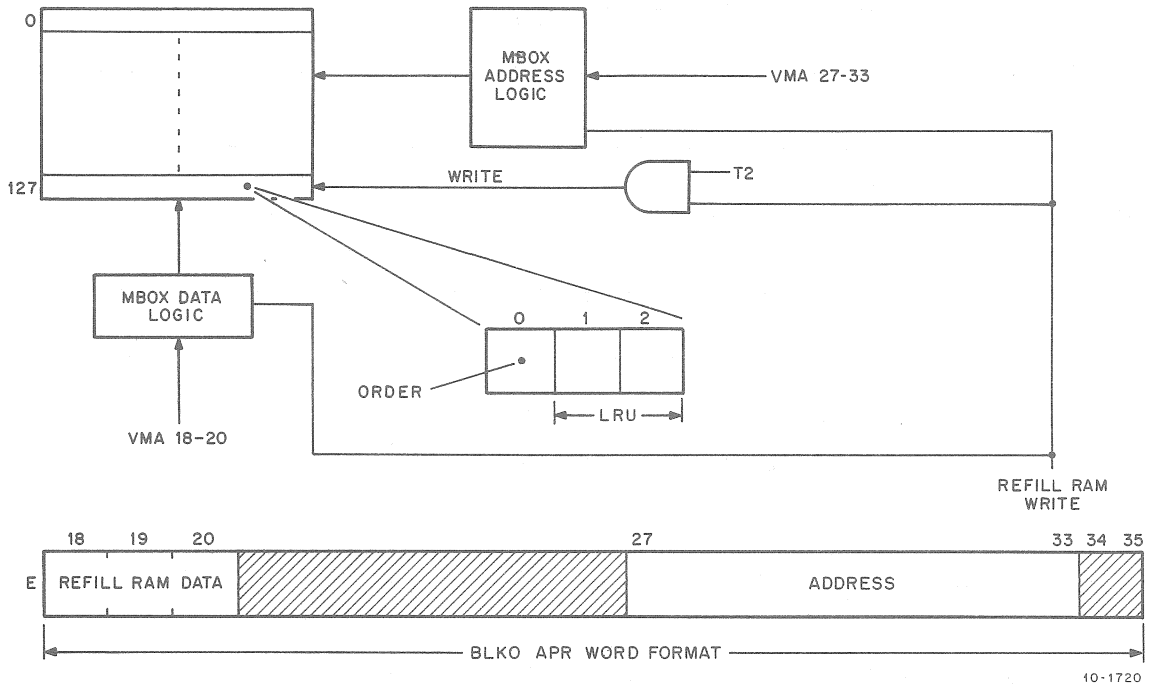



Figure 3-40 Refill RAM Overview

When the instruction is legal, the microcode performs a MEM/REG FUNC with the magic number field coded as WR REFILL RAM. The APR logic decodes the REG FUN during the EBox Request:

```
APR EBOX READ REG
APR EN REFILL RAM WR
```

The MBox writes the three high-order bits (18–20 of VMA) into the refill RAM at the location addressed by bits 27–33 of VMA. Writing the entire algorithm requires a loop using the basic instruction BLKO APR as a focal point. The following is an example:

```
        SETZB Z,AC              ;CLEAR REGISTERS
RAM1:
        MOVE AC,TABLE(Z)        ;PICK UP A WORD
        BLKO APR,0(AC)          ;WRITE THE FILL RAM
        CAIN Z,127              ;DONE ALL 12810 WORDS?
        JSR DONE                ;YES
        AOS Z                   ;NO, UPDATE Z FOR NEXT
        JRST RAM1               ;PICK UP NEXT WORD FROM
        THE TABLE
```

In the sample program, table through table+127 contain the appropriate entries to be written into the MBox Refill RAM. These words are in the format indicated on Figure 3-40. The refill algorithm may be adjusted by changing the sequence of the bit patterns. By doing this, portions of the cache may be bypassed as appropriate. Normally, all four cache quarters would be used equally. Table 3-8 is reproduced as extracted from the MBox theory section simply as an example.

Table 3-8 Sample Algorithm

| Refill RAM Locations | Refill RAM Contents | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0–7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8–15 | 3 | 1 | 2 | 3 | 2 | 1 | 2 | 3 |
| 16–23 | 7 | 1 | 2 | 7 | 1 | 1 | 2 | 7 |
| 24–31 | 6 | 5 | 6 | 7 | 5 | 5 | 6 | 7 |
| 32–39 | 0 | 3 | 2 | 3 | 0 | 2 | 2 | 3 |
| 40–47 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 48–55 | 0 | 7 | 7 | 7 | 0 | 0 | 0 | 7 |
| 56–63 | 4 | 6 | 6 | 6 | 4 | 4 | 6 | 4 |
| 64–71 | 3 | 1 | 3 | 3 | 1 | 1 | 1 | 3 |
| 72–79 | 0 | 7 | 7 | 7 | 0 | 0 | 0 | 7 |
| 80–87 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 88–95 | 4 | 5 | 5 | 7 | 4 | 5 | 4 | 7 |
| 96–103 | 0 | 1 | 2 | 2 | 0 | 1 | 2 | 1 |
| 104–111 | 0 | 5 | 6 | 6 | 0 | 5 | 6 | 0 |
| 112–119 | 4 | 5 | 6 | 5 | 4 | 5 | 6 | 4 |
| 120–127 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

### 3.3.6 MBox Error Address Register

The MBox contains a number of registers that can be loaded and read by the EBox. These registers are address registers for storing the address in the event of an error and for modifying the physical memory address in response to certain request qualifiers. The registers are:

a. User Base Register – UBR
b. Executive Base Register – EBR
c. Cache Clearer Address – CCA
d. Error Address – ERA

The ERA register can only be read by the EBox. In addition, the EBox can also read the contents of the page table to transform (map) the virtual address to the physical address and load the cache refill RAM with the cache refill algorithm.

A status word is formed and stored by the MBox in the event that an error is discovered. The error address is basically a status word that is formed and stored by the MBox when an error is sensed. In the case of a parity, time-out, or an NXM error, the corresponding error flags are set and the error address and associated status bits are loaded into the ERA register. The format of this word was shown in Figure 3-35. This register is read by the EBox when an RDERA (BLKI, PI) instruction is executed.

## 3.4 CONTROL RAM ADDRESSING

Figure 3-41 contains an overview of the CR addressing logic, while Figure 3-47 contains a more detailed version. The CR addressing logic consists of the following general parts:

Pushdown Stack, 4 words × 11 bits
Current Location register (CRA LOC)
CRAM dispatch field for holding the dispatch bits
Miscellaneous CR address gates
Diagnostic register
Dispatch decoding register 0–3 EN, 0–7 EN, 30–37 EN
CRAM loading logic
CRAM address output gates.

The type of function being performed on the CRA board determines the portions of the above-mentioned logic that are used. These functions are broadly classified as:

1. Loading into the CRAM dispatch

   Diagnostic register
   Control RAM dispatch field
   Write logic

2. Decoding the Jump, Dispatch, and Cond (Skip) fields of a microinstruction

   Mixers
   Optionally the Stack
   Optionally the A READ Logic
   Dispatch decoding register

3. Forcing a special CR address during a page fault

   CR address output gates.

In addition to these three classes, diagnostic logic is present on the CRA board for reading various registers, mixers, and signals onto the EBus. This logic is described in a separate section on EBox diagnostic logic.

### 3.4.1 Pushdown Stack

The pushdown stack, consists of eleven clocked shift registers configured as an 11-bit SILO. Two control signals, CTL SPEC/CALL and CTL DISP/RET, control the stack. Figure 3-42 illustrates the basic operation for a sequence of two subroutine calls followed by two subroutine returns. The example presented on the figure is not a practical example of subroutine calling and return, but an example of how the stack behaves in response to the call and return control signals. In practice, each subroutine consists of a number of microinstructions. For convenience, these additional instructions have been omitted. In the example the first microinstruction (J = A) asserts the first call. Note that during the first microinstruction, the CR address is "A", which is the address of the next microinstruction. When CRA CLK occurs, three significant events occur.

1. The CR address "A" is clocked into the current address buffer (CRC LOC).

2. The second microinstruction at location "A" is clocked into the CRAM register.

3. The decoding of this microinstruction begins and, in particular, enables the stack to push CRA LOC on the next CRA CLOCK.

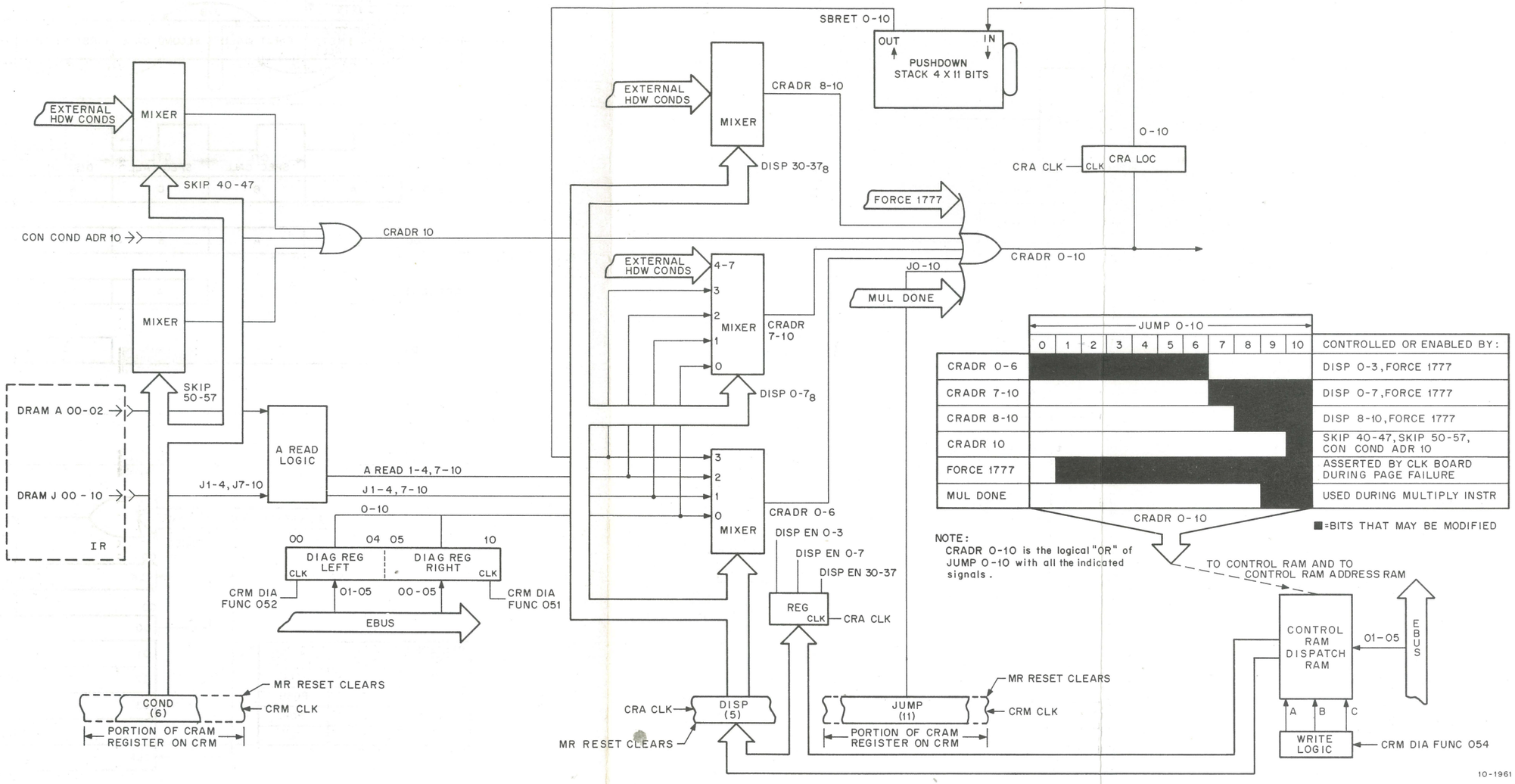



Figure 3-41 CR Addressing Overview

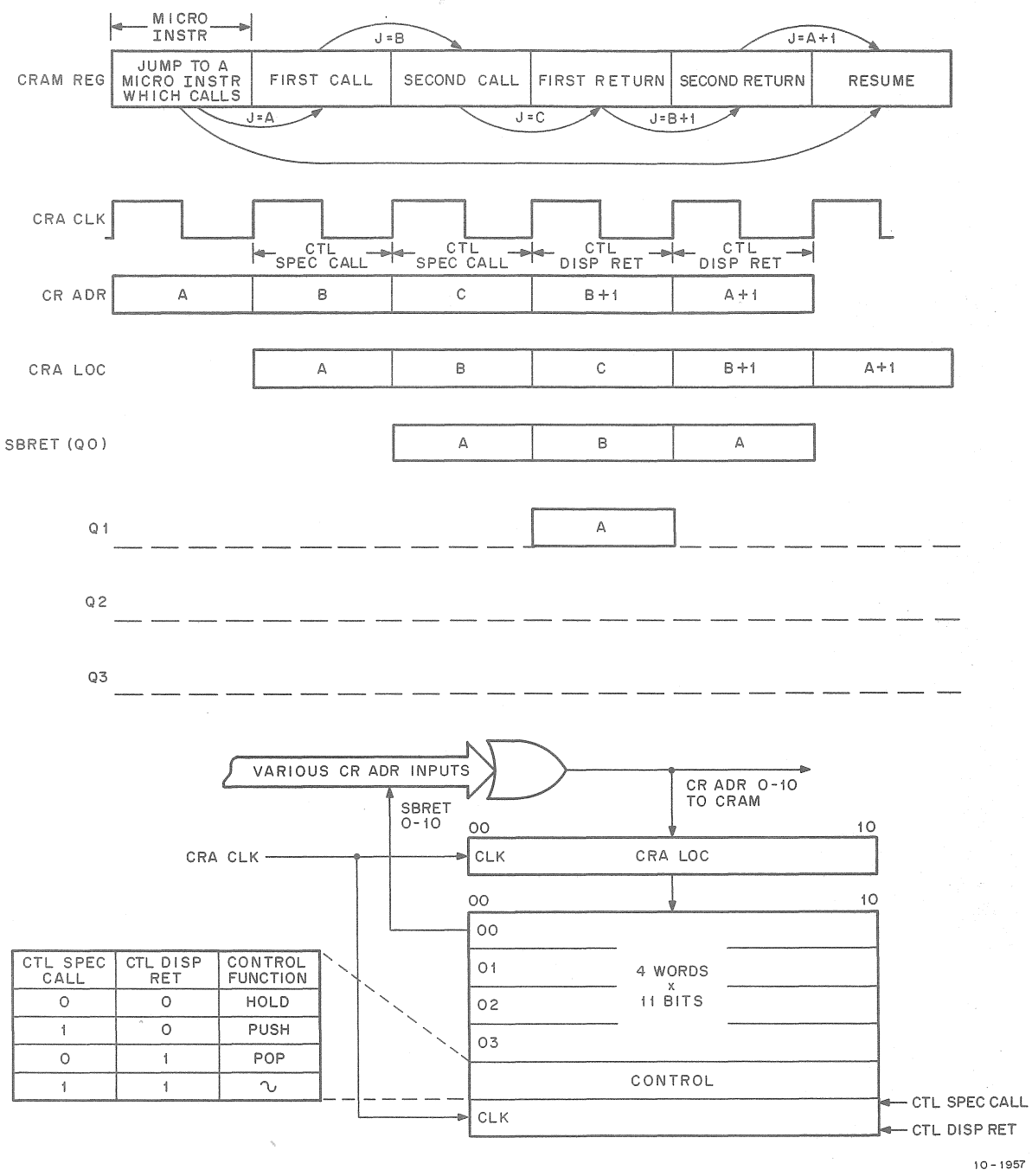



Figure 3-42 Stack Operation Example

Now the CR address becomes "B" as specified by the second microinstruction. Normally, this is the address of the first microinstruction in the subroutine. In the example, it contains a second call (J = B). The next CRA CLOCK again enables the three events indicated above, with the difference being that the CR address is now "B." CRA LOC contains "A." At the next CRA CLOCK, a second push occurs; CRA LOC "B" is pushed onto the stack (Q0) while the previous contents of Q0, which is "A," are pushed one level deeper into Q1 as indicated on the figure. Also, on this clock, the address "C" is clocked into CRA LOC. This time the microinstruction specifies the return function and the Jump address is coded so as to modify the address that will be popped off the stack on the next CRA CLOCK. For example, if the return is to be to the microinstruction following the one that made the call and the top address on the stack is "B" then the least significant bit of the "modifier," which is simply the Jump field of a returning microinstruction, is 1. Thus, the CR address is the logical OR of the address popped off the stack, "B," with the modifier 1, producing the return address B+1. Continuing the example, CRA CLK pops "B" from the stack, clocks the previous CR address (modifier 1) into CRA LOC, and returns to the microinstruction at B+1, which is a second return. Once again, the return is decoded and will enable the address "A" now at the top of the stack to be popped off and logically ORed with the modifier (once again +1) producing a CR address of A+1. This completes the example.

**NOTE**

**In this example, A and B are assumed to be even numbers.**

### 3.4.2 Current Location Register (CRA LOC)

This register consists of 11 clocked D-type flip-flops. Its two main functions are:

1. To provide the current address for the pushdown stack
2. To provide the current address for diagnostic purposes.

### 3.4.3 Control RAM Dispatch Field

The majority of the control storage for the microprogram is on the CRM board. However, the dispatch field, 1280 words of 5 bits, is contained on the CRA board. The Diagnostic register on the CRA board is used to address the entire CRAM, and this includes the portion on the CRAM board as well. Diagnostic functions are used to enable loading data placed on the EBus into the appropriate portion of the CRAM. Refer to Figure 3-41. The Diagnostic register is selected as input to the CRADR 0–6 and 7–10 mixers following power-up. This is true because the entire CRAM register is reset to zero during MR RESET, and this provides a dispatch field of zero. Using diagnostic functions 052 and 051, the Diagnostic register may be loaded from the EBus. This address now selects a word in the CRAM for loading or reading.

### 3.4.4 Miscellaneous CR Address Gates

Refer to Figure 3-43. Functionally, there are four sections of gating:

- CR Address 00–06
- CR Address 07–10
- CR Address 08–10
- CR Address 10

This grouping corresponds to the way in which portions of the CR address lines may be controlled. The CRAM, of course, sees only an address 0–10.

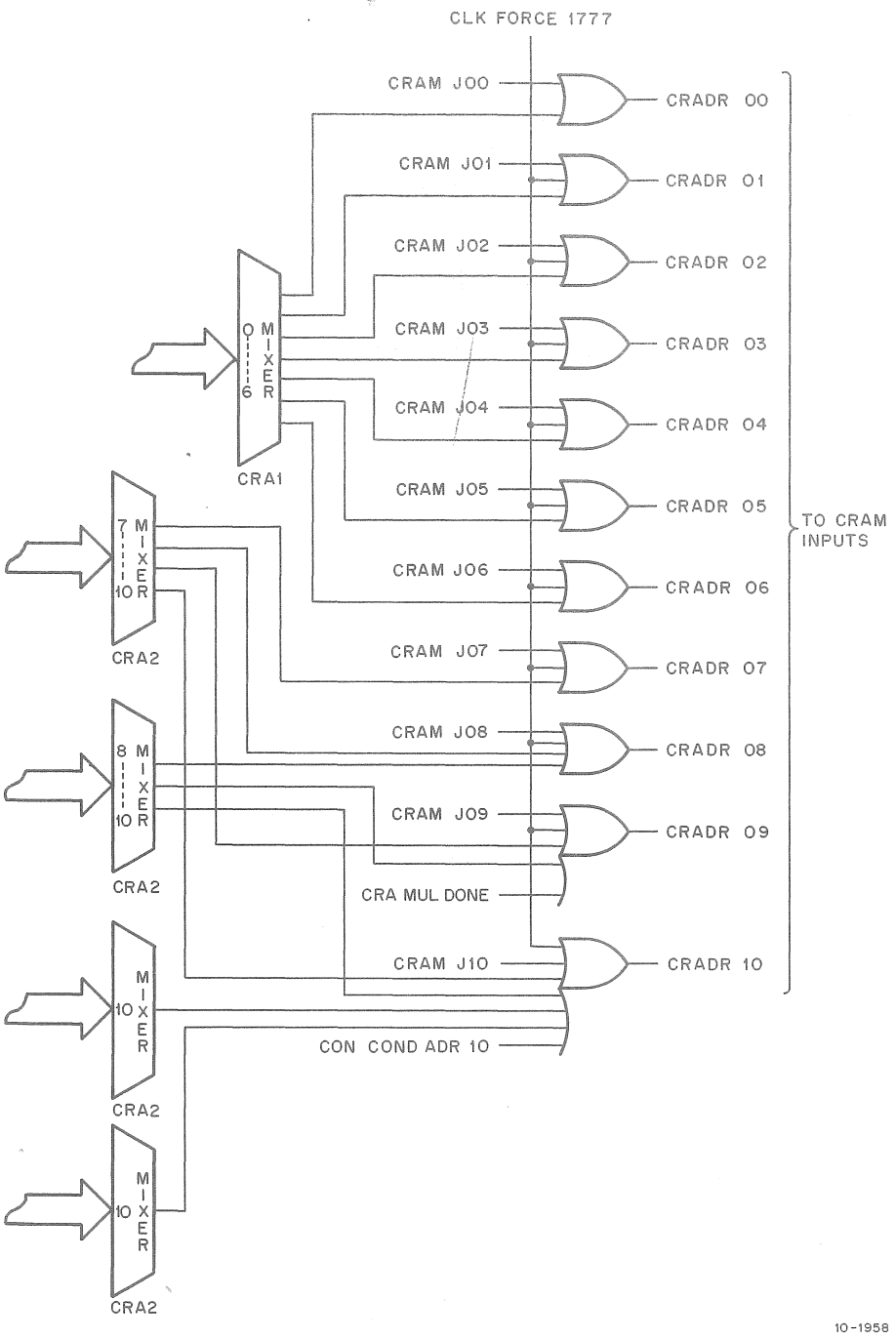
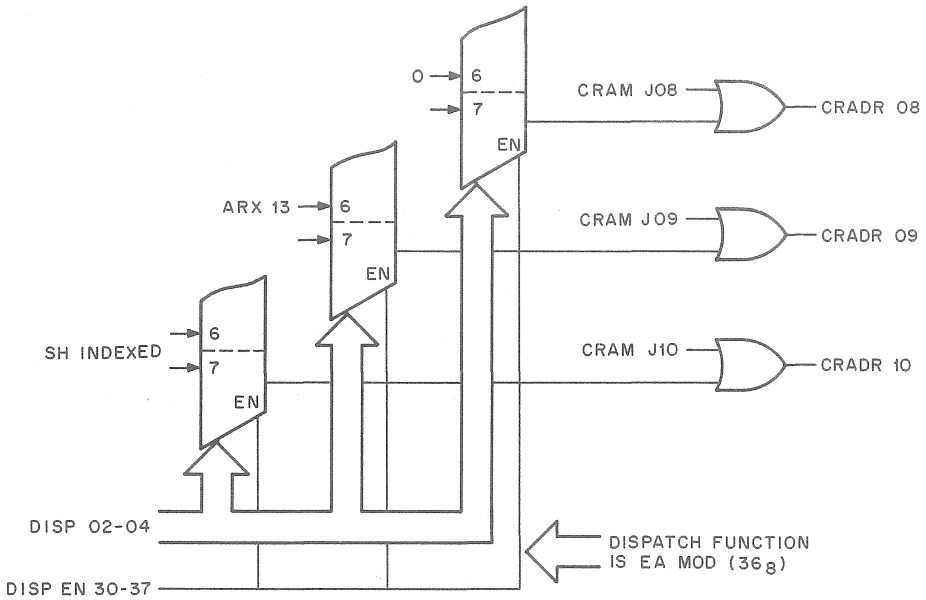



Figure 3-43 CRADR Gates

The fact that the CR address gates are OR gates should be kept in mind when trying to determine an CR output address from a particular input condition or set of conditions. To enable a particular CR address line only requires one of its input lines to be true. For example, consider the example presented in Figure 3-44, which shows the mixers that are used to select conditions to modify CR address bits 08–10. In the example, the dispatch function is effective address modification (EA MOD), which is encoded in the dispatch field as $36_8$. Note that in the example the J field (CRAM J 08–10) is 4 in bits 08–10. The four possible combinations of ARX 13 and SH indexed allow any of the following:

1. No modification to CR ADR 09 and 10
2. Modification to only CRADR 10
3. Modification to only CRADR 09
4. Modification to both CRADR 09 and 10.

Because CRAM J 08 is a 1, the respective output gate, CRADR 08, will be a 1 even though the open pin on that mixer (input 6) is effectively a 0.




| CONTROL | | INPUTS | | | OUTPUT |
|---|---|---|---|---|---|
| DISPEN 30–37 | DISP 02–04 | ARX 13 | SH INDEXED | CRAM J 08–10 | CRADR 08–10 |
| YES | 6 | 0 | 0 | 4 | 4 |
| YES | 6 | 0 | 1 | 4 | 5 |
| YES | 6 | 1 | 0 | 4 | 6 |
| YES | 6 | 1 | 1 | 4 | 7 |

10-1959

Figure 3-44 Example CRADR 08–10

### 3.4.5 Special CR Address Modification Considerations

Three special CR address modification considerations are:

1. CLK FORCE 1777
2. CRA MUL DONE
3. CON COND ADR 10.

**3.4.5.1 CLK FORCE 1777** – This signal originates on the clock board and is used to force the output gates CR address 01–10 to the address $1777_8$. This event occurs during a page fault. The page failure microcode handler begins at CRAM location 1777. Thus, the EBox, as controlled by the clock, enters a prearranged page fail sequence. Loading the first microinstruction from the page fault handler, CLK FORCE 1777 forces the CRAM address lines, as indicated, and then issues a single CRM CLK, which loads the microinstruction into the CRAM register. At this point, EBox's normal operation continues. Note that CLK FORCE 1777 does not affect CR ADR 00, and thus may force the microcode to either 1777 or 3777. The first step of the page fault handler is duplicated in these two locations.

Note, also, that at the same time as the CLK board is forcing CLK FORCE 1777, the CTL board is forcing CTL SPEC CALL in order to place the return address on the pushdown stack.

**3.4.5.2 CON COND ADR 10** – This external signal is formed on the CON board and routed to CRA 2 as CON COND ADR 10. Refer to Figure 3-45, which shows the boards involved in decoding the Cond and Dispatch fields. Note that each board contains tables indicating those functions that are decoded on that board. The signal CON COND ADR 10 is formed when Skip 60–67 or Skip 70–77 are decoded. The various hardware conditions involved are indicated on the tables.

**3.4.5.3 MUL DONE** – During the Dispatch function, MUL, the state of the sign of FE, as well as MQ34 and MQ35, are used to modify the CRAM address in the multiply loop. When the sign of FE becomes false an exit is made from the multiply loop. This is done via CR ADR 08. Simultaneously, MUL DONE (Figure 3-46) is generated to force address bits 09 and 10. This is done merely to save microcode words. Without this logic, MUL DISP would be an 8-way branch; with this logic, it is a 5-way branch.

### 3.4.6 AREAD Logic

Refer to Figure 3-47. The AREAD logic is shown on the lower right-hand side. It consists of a mixer and various gating elements. Basically, this logic is controlled by bits of the DRAM A field. Specifically, when the DRAM A field bits 00 and 01 are 0s; then the AREAD logic AREAD 01–04 and AREAD 07–10 become equivalent (bit for bit) to DRAM J01–04 and DRAM J07–10. When DRAM A00 or 01 is a 1, then AREAD 01–04 and 07–10 generate $40_8$ +A, dispatching to location 42 through 47 in the microcode.

The outputs of the AREAD logic (to be able to modify the CR address lines) must be selected in the appropriate mixers. Once again referring to Figure 3-47, the mixers involved are those controlling CRADR bits 00–06 and 07–10. These mixers will select the AREAD function when the dispatch field is coded as "2."

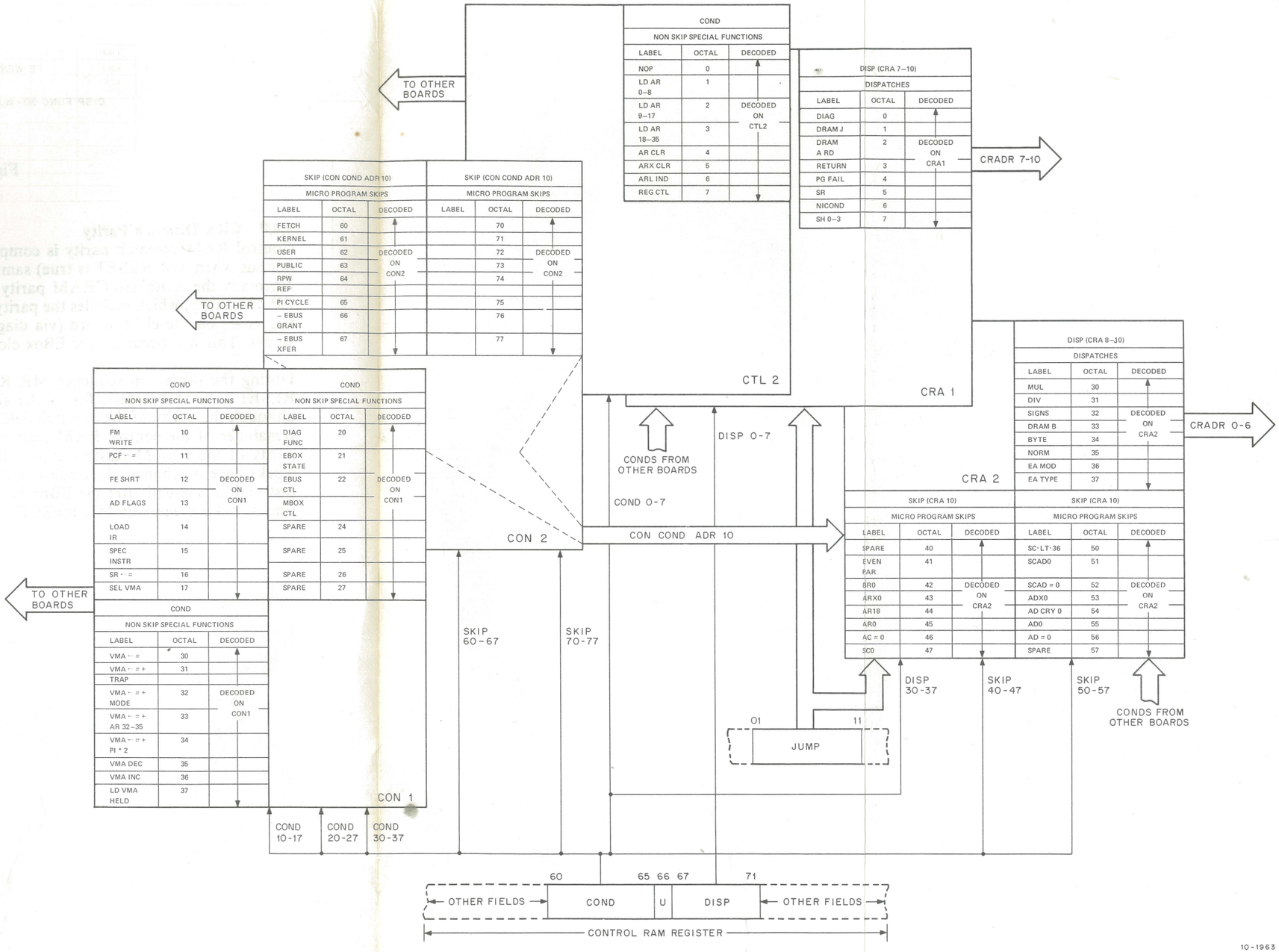



Figure 3-45 COND and Dispatch Layout and Control

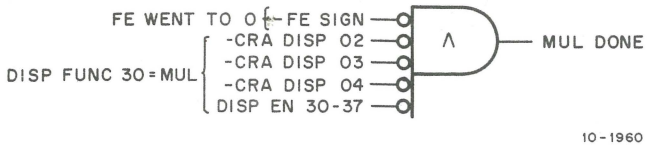
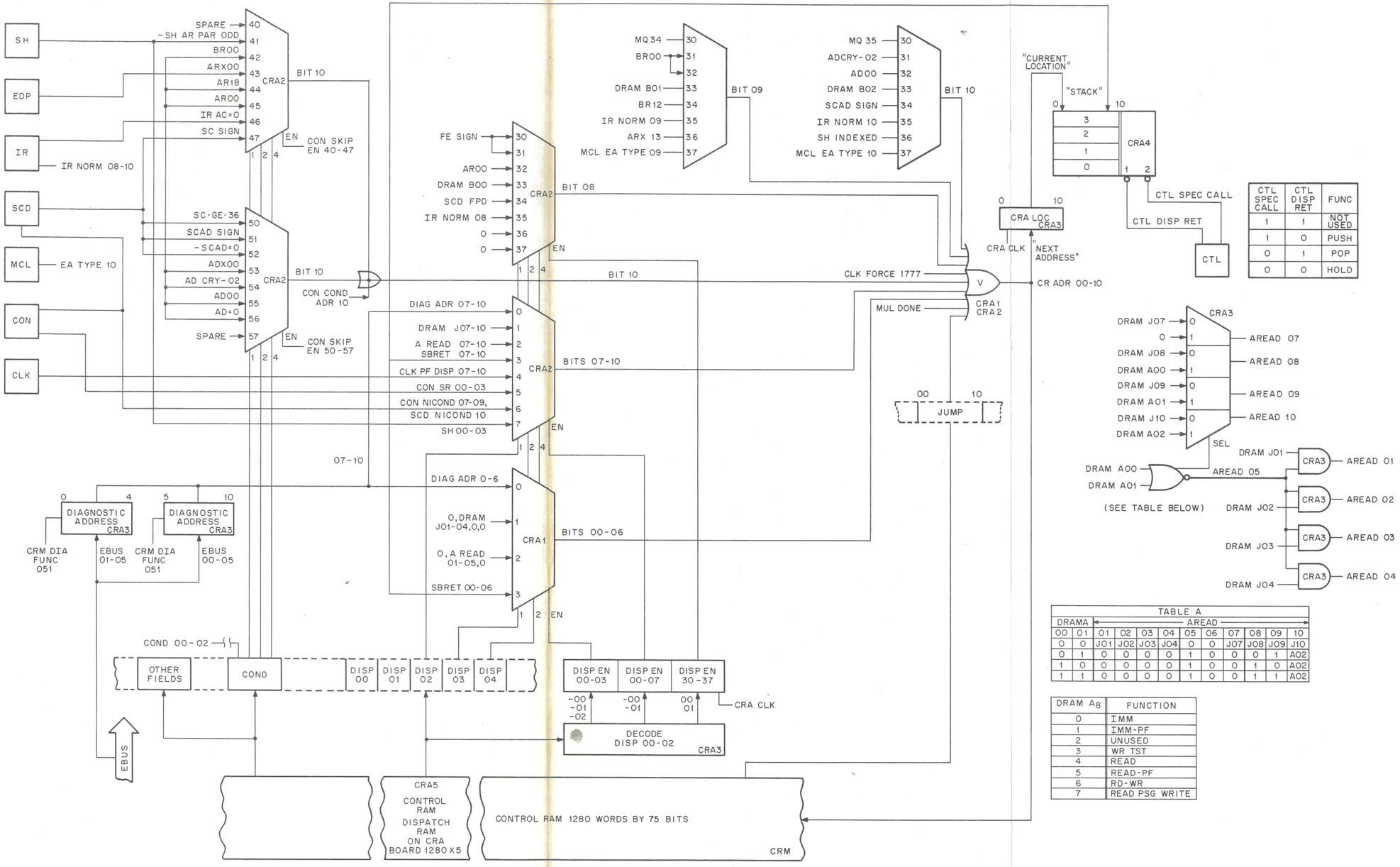



Figure 3-46 MUL Done

### 3.4.7 CRA Dispatch Parity

Control RAM dispatch parity is computed using a 10160 parity circuit. This circuit (except during periods when MR RESET is true) samples CRA DISP bits 00–04 and computes CRA DISP parity. Normally the combined CRAM parity is odd, when correct. The clock board monitors the state of CRAM parity, which includes the parity for the dispatch field. If the CLK CRAM PARITY CHECK flag is set on the clock board (via diagnostic function 044), then any CRAM parity error stops all clocks. This will occur on the EBox clock following the CRAM parity error.

During the power up sequence MR RESET sets and remains set. This generates the signal DISP RESET PARITY, which forces the state of the dispatch parity network to indicate odd parity, although the parity of the dispatch field (which now contains all zeros) is even. This, together with the remainder of the control RAM register which is clear, yields odd parity. The effect is to make the parity of the CRAM register appear to be odd following MR RESET. This logic assures that the clocks have no chance of stopping in the event that CLK CRAM PAR check is true when a CONO instruction is issued after the EBox has been powered up and this instruction causes MR RESET or similarly if a diagnostic MR RESET is issued.




| CTL SPEC CALL | CTL DISP RET | FUNC |
|---|---|---|
| 1 | 1 | NOT USED |
| 1 | 0 | PUSH |
| 0 | 1 | POP |
| 0 | 0 | HOLD |

TABLE A

| DRAMA | | AREAD | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 |
| 0 | 0 | J01 | J02 | J03 | J04 | 0 | 0 | J07 | J08 | J09 | J10 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | A02 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | A02 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | A02 |

| DRAM $A_8$ | FUNCTION |
|---|---|
| 0 | IMM |
| 1 | IMM-PF |
| 2 | UNUSED |
| 3 | WR TST |
| 4 | READ |
| 5 | READ-PF |
| 6 | RD-WR |
| 7 | READ PSG WRITE |

10-1962

Figure 3-47 Control RAM Addressing

# APPENDIX A
# UNDERSTANDING THE MICROCODE

The control portion of the EBox comprises the DRAM and the CRAM. The DRAM has storage for 512 decimal words, one for each KL instruction. During instruction execution, the DRAM word provides information about the type of memory references by the executing instruction. It also provides an index into the main control programs contained in the CRAM.

The CRAM provides storage for 1280 microinstruction words that are structured into a complicated control program called the "microcode." This section defines and explains the microcode. Although many figures of sample listings from the microcode listing are used throughout the discussion, an assumption is made that the reader has an up-to-date copy of the microcode listing (either hard copy or microfiche). The examples shown here refer to specific sections of the listing; the reader may wish to follow the examples through the actual listing while reading this section.

The discussion begins by introducing the microcode and describing field, value, label, and microinstruction definitions. This leads into defining macros, pseduo-operators, and location control. Then, two instructions (MOVE and ADD) are illustrated, leading the reader through the microcode listing. Figures A-17 through A-23 (located at the end of this appendix) complement the discussion and define all the CRAM and DRAM fields. Refer to these figures whenever necessary.

The microcode is presented in groups, with each group (designated a through g) representing four octal digits as they appear in the listing. Each group represents one or more fields. For example, the listing for microcode address 0724 is shown in Figure A-1.

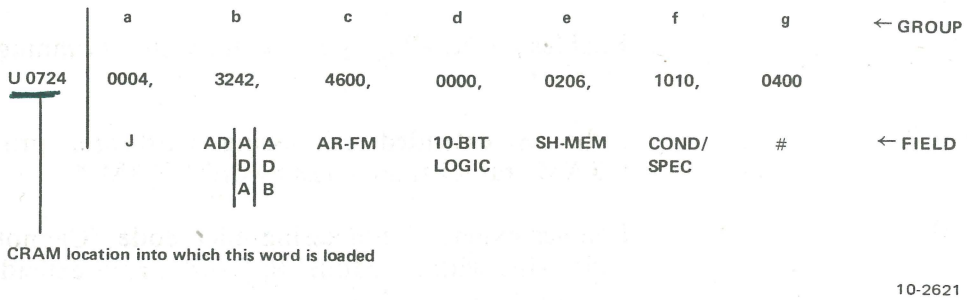



Figure A-1 Sample Microcode Listing

Each of the group's coding is defined in the respective figures listed below:

| Group | Figure |
|---|---|
| a | A-17 |
| b | A-18 |
| c | A-19 |
| d | A-20 |
| e | A-21 |
| f | A-22 |
| g | A-23–25 |

The DRAM contains storage for each instruction. During instruction execution, the DRAM word (Figure A-4) provides information about the type of memory references required by the executing instruction and also provides an index into the main control program located in the CRAM.

**Conditional Assembly Variable Definitions**
The Conditional Assembly variables observed in the microcode listing are listed and defined below. (The definitions are presented for the variable set to 1. The values shown are the normal settings.)

| Variable | Definition |
|---|---|
| TRACKS = 0 | Enables storing the PC after every instruction and creates DATAI/O PI, to read/setup the PC Buffer address.* |
| OP.CNT = 0 | Enables code to build a histogram in core to count the usage of each op code in both USER and EXEC mode.* |
| OP.TIME = 0 | Enables code to accumulate time spent by each op code.* |
| FPLONG = 1 | Enables KA style double precision floating-point instructions (e.g., FADL, FSBL). This feature is not supported in systems running the TOPS-20 monitor. |
| MULTI = 0 | If operating a multiprocessor system, this suppresses cache on unpaged references; paged references are left up to EXEC.* |
| KLPAGE = 0 | Enables the KL-Paging mode, for systems running the TOPS-20 monitor. |
| MODEL.B = 0 | Indicates extended addressing hardware, primarily a 2K CRAM, rather than a 1280 word CRAM.* |
| XADDR = 0 | Enables extended addressing microcode. (Cannot do extended addressing without Model B; Cannot have extended addressing without KL page).* |
| IMULI.OPT = 0 | Enables optimization of IMULI to take only nine multiply steps. |

*This feature is not supported.

| Variable | Definition |
|---|---|
| SXCT = 1 | Enables special XCT instruction, which allows diagnostics to generate large addresses. (Do not need SXCT with extended addressing. Cannot do it in Model B hardware.) |
| EXTEND = 1 | Enables the extended instruction set. |
| DBL.INT = 1 | Enables double integer instructions. |
| ADJBP = 1 | Enables adjust byte pointer. |
| RPW = 1 | Enables Read-Pause-Write cycles for non-cached references by some instructions. |
| WRTST = 0 | Enables Write-Test cycles at AREAD time for instructions such as MOVEM and SETZM.* |
| BACK.BLT = 0 | Enables BLT to decrement addresses on each step if E < RH (AC); breaks many programs.* |
| .SET/INSTR .STAT = 0 | Enable instruction statistics code.* |

**Field Definitions**

The actual (physical) CRAM bits are derived from the CRAM Board logic. However, no logical relationship exists between the physical bits and the respective microword bit names. Figures A-2 and A-3 are located at the end of the introductory discussion, just before the two examples. Figure A-2 shows how the physical CRAM bits are derived. Figure A-3 shows the physical bits and the corresponding microword bit position (and name). The microcode listing is ordered with respect to the microword bit positions, not the actual CRAM order.

Microcode field definitions have the form SYMBOL/ = J, K, L, M. The J parameter is only meaningful when "D" is specified as the default mechanism. The K parameter defines the field size in the number of bits (in decimal). The L parameter defines the field position (in decimal) as the bit number of the right-most bit of the field; bits are numbered from 0 on the left. Note that the position of bits in the microcode word (Figure A-3) bears no relation to the ordering of bits in the hardware microword, where fields are often broken up and scattered. The M parameter is optional; it selects a default mechanism for the field. The legal values of this parameter are the characters D, T, P, or +, where:

D — Indicates that J is the default value of the field if no explicit value is specified.

T — Is used on the time field to specify that the value of the field depends on the time parameters selected for other fields. Within the microcode, T1 parameters are used to specify functions that depend on the adder setup time; T2 parameters are used for functions that require additional time for correct selection of the next microinstruction address.

P — Is used on the parity field to specify that the value of the field should default, such that parity of the entire word is odd. If this option is selected on a field where the size (K) is zero, the microassembler attempts to find a bit somewhere in the word for which no value is either specified or defaulted.

*This feature is not supported.

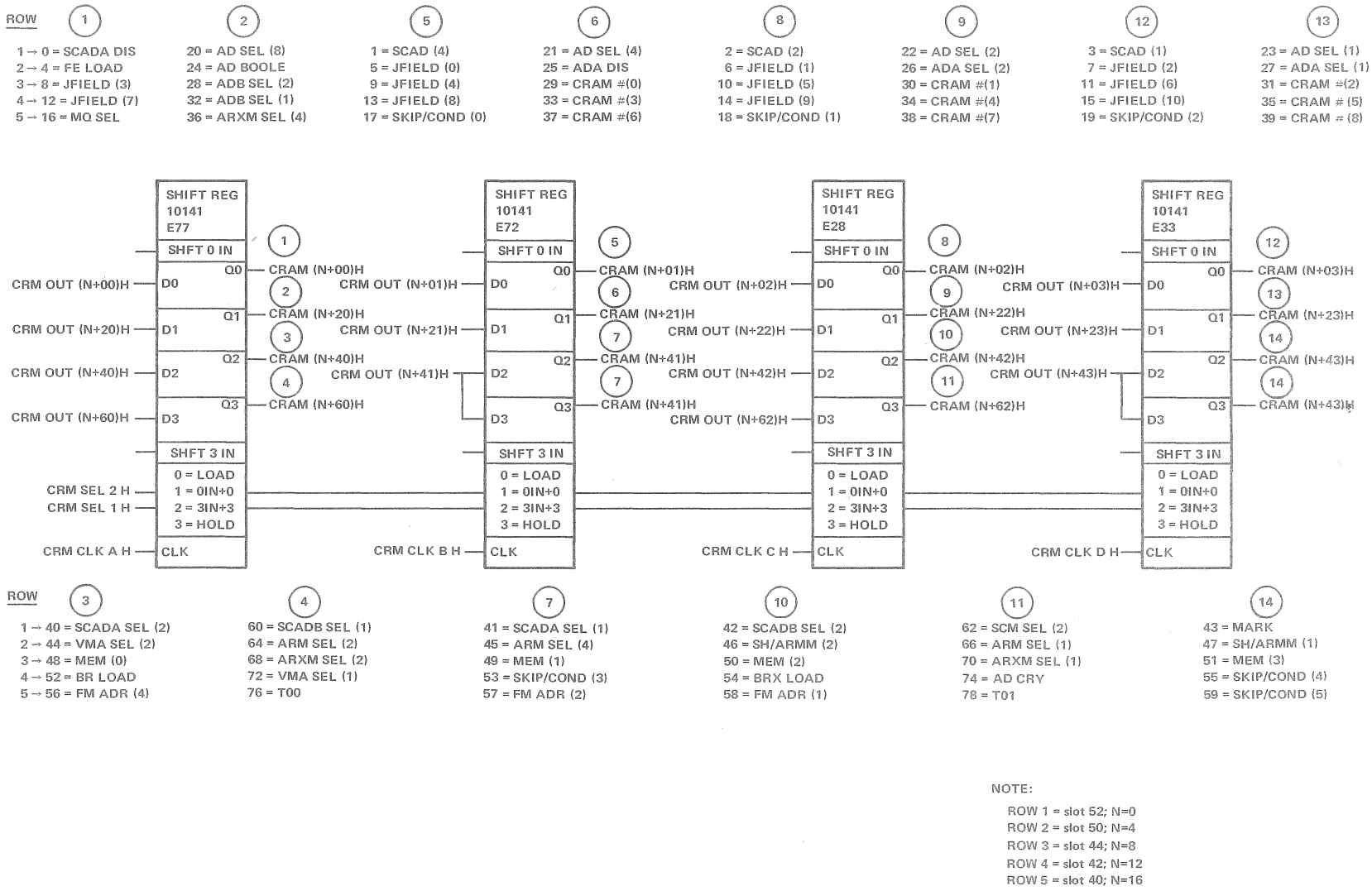



Figure A-2 CRAM Board Logic Physical Bit Position Derivation

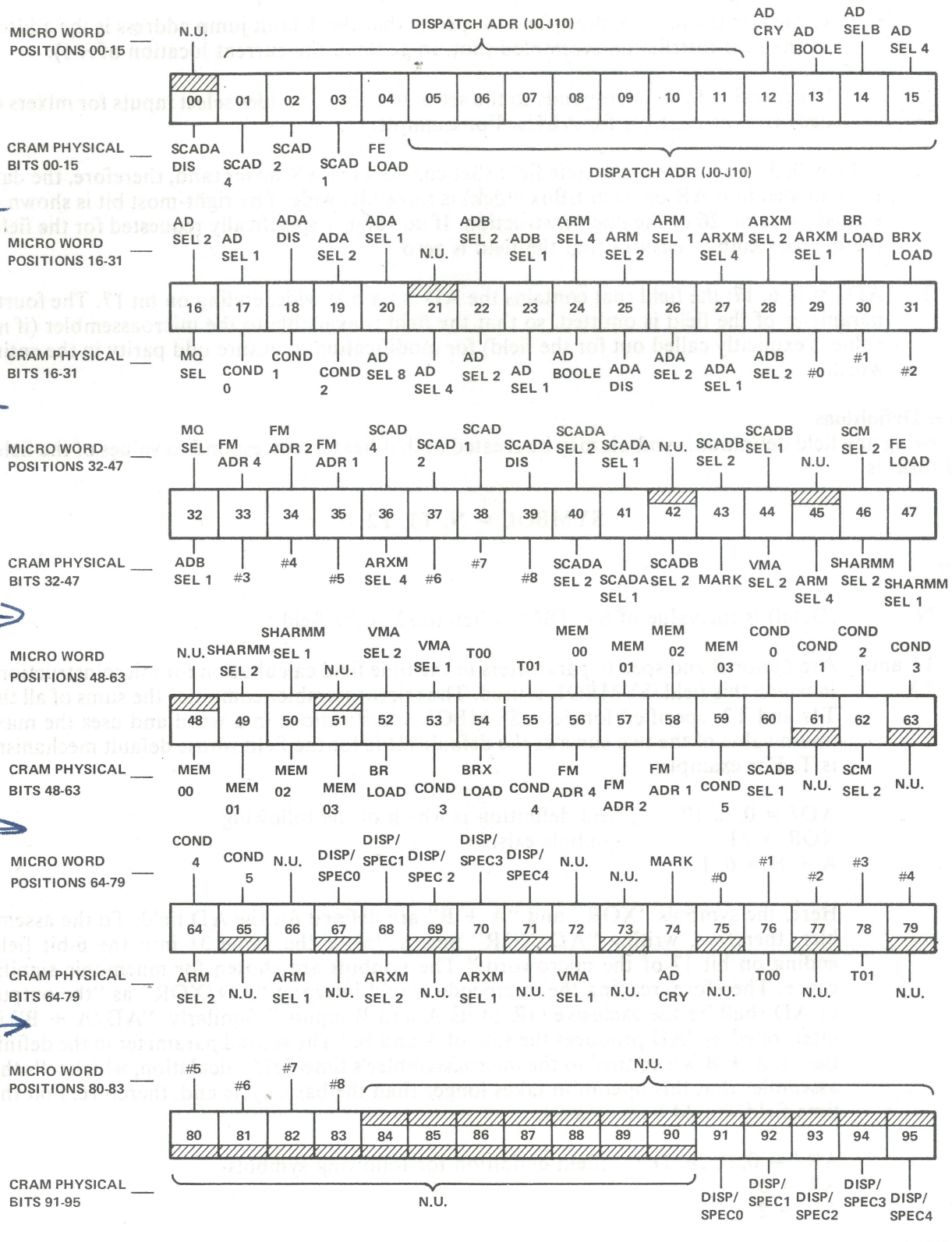



Figure A-3 Actual CRAM Physical Bit Position to Microword Bit Position Correlation

+ Is used on the jump address field to specify that the default jump address is the address of the next instruction assembled (not, in general, the current location of +1).

In general, a field corresponds to the set of bits that provides select inputs for mixers or decoders, or controls for ALUs. For example:

1. AR/ = 0, 3, 26, D; the microcode field that controls the AR mixer (and, therefore, the data to be loaded into AR on each EBox clock) is three bits wide. The right-most bit is shown in the listing as bit 26 of the microinstruction. If no value is specifically requested for the field, the microassembler ensures that the field is zero.

2. AD/ = 0, 6, 17; the field that contains the AD is six bits wide, ending on bit 17. The fourth parameter of the field is omitted, so that the field is available to the microassembler (if no value is explicitly called out for the field) for modification to ensure odd parity in the entire word.

**Value Definitions**

Following any field definition, symbols may be created in that field to correspond to values of the field. The form is

SYMBOL = N, T1, T2

where:

N (Octal) is the value of SYMBOL when used in the field;

T1 and T2 Are optional and specify parameters in the time field calcultation for microinstructions in which this field/SYMBOL is used. The microassembler computes the sums of all the T1s and T2s specified for field/SYMBOL specifications in a word and uses the maximum value of the two sums as the default value for the field whose default mechanism is T. For example:

```
AD/ = 0, 6, 17      ; field definition is which of the following
XOR = 31            ; symbols exist.
A + B = 6, 1
```

Here, the symbols "XOR" and "A + B" are defined for the AD field. To the assembler, therefore, writing "AD/XOR" means "place the value 31 into the 6-bit field ending on bit 17 of the microword." The symbols are chosen for mnemonic significance. Therefore, reading the microcode would interpret "AD/XOR" as "the output of AD shall be the exclusive OR of its A and B inputs." Similarly, "AD/A + B" is interpreted as "AD produces the sum of A and B." The second parameter in the definition of A + B is a control to the microassembler's time-field calculation, which tells the assembler that this operation takes longer than the basic cycle and, therefore, that the time field should be increased.

```
AR/ = 0, 3, 26, D     ;field definition for following symbols
AR = 0
AD = 2
```

Here, the symbols "AR" and "AD" are defined for the field named "AR," which controls the AR mixer. Because only the default case is used, the AR does not change unless a specific request to do so is made. Therefore, the field definition specifies zero as the default value of the field. If the AR is loaded from the AD output, AR/AD is written to set the mixer selects to pass the AD output into the AR.

**Label Definitions**
A microinstruction may be labeled by a symbol followed by a colon preceding the microinstruction definition. The address of the microinstruction becomes the value of the symbol in the field titled "J." For example:

```
TOP:  J/TOP
```

This is a microinstruction whose J field (Jump Address) contains the value "TOP." It also defines the symbol "TOP" to be the address of itself. Therefore, if executed by the microprocessor, the microinstruction would loop on itself.

**Comments**
A semicolon anywhere on a line causes the remainder of the line to be ignored by the assembler; it is purely information to the reader. For example:

```
AD/0, 6, 17          ;field definition in which following symbols
                     ;exist.
```

Only AD/0, 6, 17 is relevant to the assembler; that data following the semicolon is useful information to the reader.

**Microinstruction Definition**
A word of microcode is defined by specifying a field name, followed by a slash (/), followed by a value. The value may be a symbol defined for that field, an octal digit string, or a decimal digit string (distinguished by the fact that it contains "8" and/or "9" and/or is terminated by a period). Several fields may be specified in one microinstruction, by separating field/value specifications with commas. For example:

```
ADB/BR, ADA/AR, AD/A + B, AR/AD
```

In this example, the field named "ADB" is given the value named "BR" (to cause the mixer on the B side of AD to select BR); field "ADA" has the value "AR;" field has the value "A + B," and field "AR" has the value "AD."

**Continuation**
The definition of a microinstruction may be continued onto two or more lines by breaking the definition after any comma. That is, if the last nonblank, noncomment character on a line is a comma, the instruction specification is continued on the following line. For example:

```
ADB/BR, ADA/AR,          ;select AR and BR as AD inputs
    AD/A + B, AR/AD      ;take the sum into AR
```

By convention, continuation lines are indented on extra tab.

**Macros**
A macro is a symbol, the value of which is one or more field/value specifications and/or macros. A macro definition is a line containing the macro name followed by a quoted string that is the value of the macro. For example:

```
AR   AR + BR        "ADB/BR, ADA/AR, AD/A + B, AR/AF"
```

The appearance of a macro in a microinstruction definition is equivalent to the appearance of its value.

**Pseudo-Operators**
The microassembler contains ten pseudo-operators:

| | | |
|---|---|---|
| 1-2. | .DCODE and .UCODE | Select the RAM into which subsequent microcode is loaded and, therefore, the field definitions and macros that are meaningful in subsequent microcode. |
| 3. | .TITLE | Defines a string of text to appear in the page header. |
| 4. | .TOC | Defines an entry for the Table of Contents at the beginning. |
| 5. | .SET | Defines the value of a conditional assembly parameter. |
| 6. | .CHANGE | Redefines a conditional assembly parameter. |
| 7. | .DEFAULT | Assigns a value to an undefined value. |
| 8. | .IF | Enables assembly if the value of the parameter is not zero. |
| 9. | .IFNOT | Enables assembly if the parameter value is zero. |
| 10. | .ENDIF | Re-enables assembly if suppressed by the parameter named. |

**Location Control**
A microinstruction labeled with a number is assigned to that address. The character "=" at the beginning of a line, followed by a string of 0s, 1s, and/or *s, specifies a constraint on the address of the following microinstructions. The number of characters in the constraint string (excluding the "=") is the number of low-order bits contained in the address. The microassembler attempts to find an unused location whose address has zero bits in the positions corresponding to 0s in the constraint string and one bits where the constraint has 1s. Asterisks denote "don't care" bit positions.

If any zeros are in the constraint string, the constraint implies a block of (2 * * N) microwords, where N is the number of 0s in the string. All locations in the block have 1s in the address bits corresponding to 1s in the string. Bit positions denoted by *s are the same in all block locations.

In such a constraint block, the default address progression is counting in the "0" positions of the constraint string, but a new constraint string occurring within a block may force skipping over some locations of the block. Within a block, a new constraint string does not change the pattern of default address progression, it merely advances the location counter over those locations. The microassembler fills them in later.

A NULL constraint string ("=" followed by anything except 0, 1, or *) serves to terminate a constraint block. For example:

a. = 0

This specifies that the low-order address bit must be zero. The microassembler finds an even-odd pair of locations and places the next two microinstructions into them.

b. = 11

This specifies that the two low-order bits of the address must both be ones. Because there are no zeros in this constraint, the assembler finds only one location meeting this constraint.

c. = 0*****

This specifies an address in which the $40_8$ bit is zero. Due to the implementation of this feature in the assembler, the default address progression applies only to the low-order five bits. Therefore, this constraint finds one word in which the $40_8$ bit is 0 and does not attempt to find one where that bit is a 1.

**Microcode Examples**

The following paragraphs lead the reader through the microcode, while defining two instructions: MOVE and ADD. The requirements that the microcode is loaded and running (i.e., in the HALT loop) are assumed. A dispatch (test for an interrupt) occurs during a HALT loop. Once an interrupt is present, the microcode leaves the HALT loop and goes to the first microinstruction.

**MOVE Instruction**

Refer to Figure A-4. The initial dispatch is a NICOND Dispatch. It is a decision starting at microcode address 140 that is used to decide which condition (e.g., TRAP, NICOND) is satisfied. Looking up Next Instruction Dispatch in the microcode listing Table of Content refers the reader to line 2549 in the listing. The decision begins at line 2549. Notice that the actual decisions and respective implementations begin at microcode address 140 (NEXT), and assuming a NICOND Dispatch is present, the listing refers the reader to NEXT + 12 (microcode address 152).

The NICOND Dispatch is the normal case; the instruction is in the ARX and begins execution. Location NEXT + 12 leads (jump to the correct decision) the reader to microcode address 152 (XCTGO), line 2606. Notice in the listing (and Figure A-5):

At XCTGO, on line 2606, the comments state "save the instruction, sign extend Y and calculate the effective address (EA)." The macros define all the things that happen here. Initially, one should consider where to go next. That information is contained in:

1. The J-field, which typically contains the "suggested" next address. In this example, it is 160. Whether that is used or not depends on item 2.

2. The Dispatch (or SPEC) field.

The SPEC field follows the "f" column in the microcode listing (Figure A-22). Specifically, the field observes the last two digits of the "f" column. In this example, those digits are "36." Going to Figure A-22, notice that a decoded 36 in the SPEC field is an EA MODE Dispatch.

An EA calculation is called for, which indicates that under certain conditions the J-field (160) may not be the actual next address. These conditions are Indexing (bits 14–17 of the instruction), Indirection (bit 13), both conditions, or neither condition. In this case, EA MODE dispatch looks at those bits of information in the instruction and then ORs them with 160 (the J-field). Because this simple MOVE instruction uses neither indexing nor indirection, go directly to 160. This appears on line 2647 (if you cannot easily locate this, go to the index at the rear of the listing, look up address 160 and find that line 2647 is where microcode address 160 appears). Refer to Figure A-6.

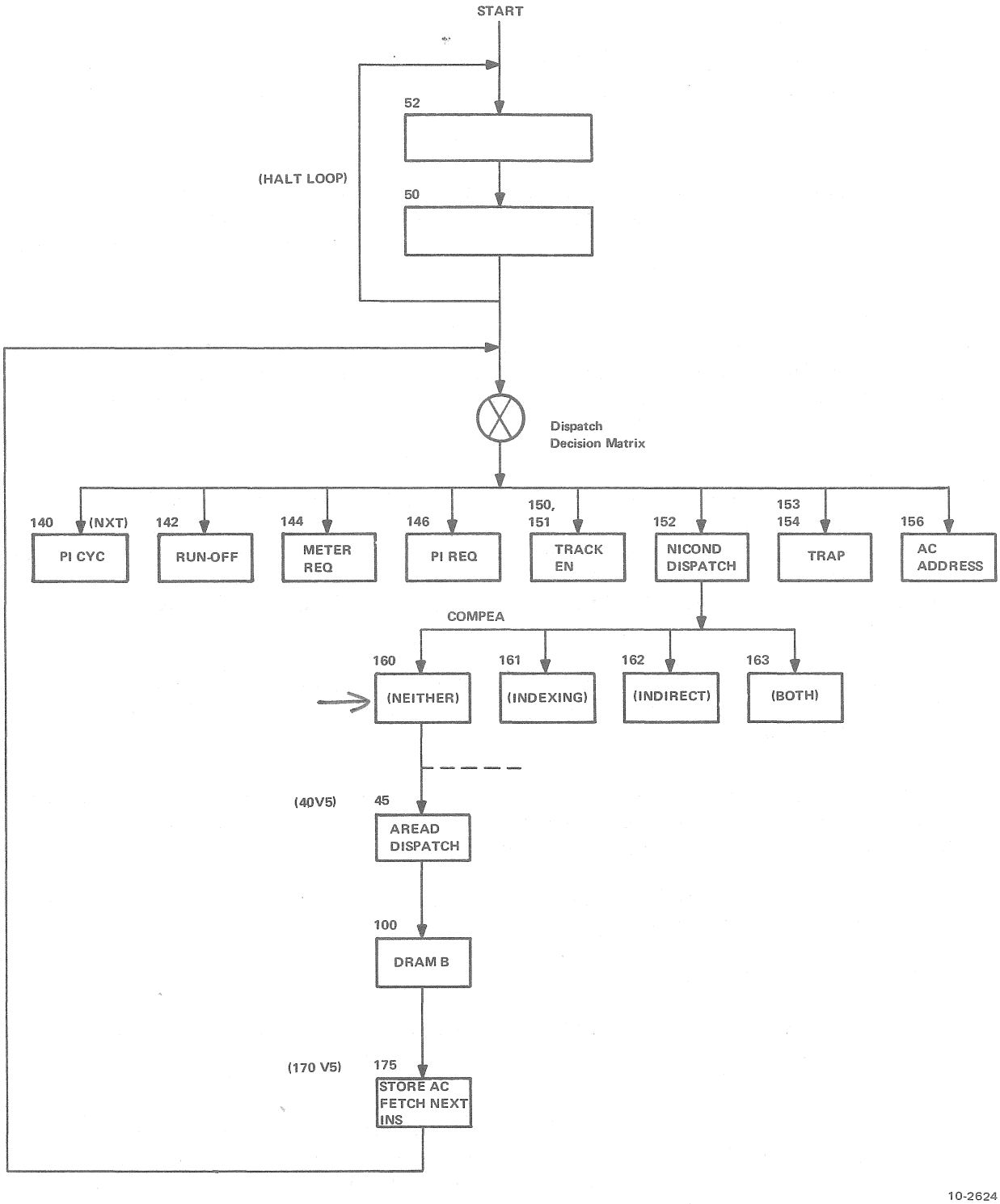



Figure A-4 MOVE Instruction Flow Diagram

```
                                                ;2606 XCTGO: BRX/ARX,AR  ARX,SET ACCOUNT EN,  ;SAVE INSTR, SIGN EXTEND Y,
U0152,0160,0001,4022,2000,2136,0105 ;2607                    XR,EA MOD DISP, J/COMPEA  ;GO CALCULATE EA
```

Figure A-5 Microcode Address 152

```
U0160,0000,3701,0000,0000,0204,0002,0000  ;2647  COMPEA: GEN AR, A READ          ;LOCAL
```

Figure A-6 Microcode Address 160

Again looking at the "f" column, observe the SPEC field is "02." Checking Figure A-22, SPEC code 02 indicates doing an A READ Dispatch by stating DRAM A RD. Go to the microcode listing index for DRAM words (it appears just before the microcode address index). The MOVE instruction is op code 200. Find 200 and notice it refers you to line 2782. Refer to Figure A-7.

```
D0200,5500,0100      ;2782  200:  R-PF,  AC,  J/MOVE     ;BASIC MOVE
```

Figure A-7 DRAM Word 200

This is the DRAM word for the basic MOVE instruction. The A-field is a "5" (Figure A-26). This "5" is ORed with 40 (a constant used whenever an A RD DISPATCH is performed) and the J-field (0000) of microinstruction 160. This results in a "45." Turning again to the index, look up microcode address 45. The index indicates line 2711; see Figure A-8.

```
                                                ;2711 BR/AR,FIN XFER, I FETCH,  ;GET OPERAND, PREFETCH,
U0045,0000,3240,0043,0000,0226,0001,0400  ;2712                TIME 3T, IR DISP, J/O  ;& START EXECUTE
```

Figure A-8 Microcode Address 45

This part of the microcode states: get the operand (from the MBox), begin a prefetch of the next instruction, and begin instruction execution. Notice also in the macros, that an IR Dispatch is called. Looking now at the SPEC field, it is "01;" looking this up in Figure A-22 states DRAM J DISPATCH. A DRAM J DISPATCH dictates calculating where to go by taking *only* the J-field of the DRAM word as the address. In the case of the simple MOVE instruction (look back at Figure A-7), notice the A-field is "5," the B-field is "5," and the J-field is "100."

Looking up microcode address 100 in the index leads the reader to line 2819 (Figure A-9).

```
U0100,0170,0001,0000,0000,0005,0033,0000  ;2819  MOVE:  EXIT     ;STORE AS IS FROM AR
```

Figure A-9 Microcode Address 100

The SPEC field is "33" and, again referring to Figure A-22, now a DRAM B is called. DRAM B is the actual "store the operand." The MOVE began by fetching the operand and placing it in the AR. Finally, it is placed in a particular AC. The DRAM B Dispatch takes the B-field of the DRAM word (5) and ORs it with the J-field (170) of the current microinstruction (address 100). This results in: 170 V 5 = 175. The index takes the reader to line 2749; see Figure A-10.

```
U0175,0140,4001,0000,0403,0002,1006,0000  ;2749  STAC: AC0_AR,NXT INSTR  ;NORMAL AND IMMEDIATE MODES
```

Figure A-10 Microcode Address 175

Observing the SPEC field indicates "06;" this is a NICOND Dispatch. Also, the J-field is 140, taking the reader to the original decision matrix. Again, all the possibilities are considered when the next instruction arrives and the process continues.

Not all fields were discussed here, only the major fields. All fields are illustrated and defined in Figures A-17 through A-26. It is left to the reader to check the unmentioned code fields with the respective defining figures.

**ADD Instruction**

Many of the assumptions used in the MOVE example are used again here (refer to Figure A-11). Assume that the last instruction was a NICOND Dispatch; go through the decision matrix to microcode address 152. Assume Indexing this time, this leads the reader to address 161. Locate address 161 on line 2648 of the listing (see also Figure A-12).

Indexing is handled at this time. The AR is added to the contents of the XR (Index register) to generate the EA. Also, an A READ Dispatch is called out. The A READ leads to the next microcode instruction, which is where the operand is located. Assume AC3 is being used (for example) and its content is "50;" assume the Y-field contains "100." This results in EA = 150.

Again, because of a COMP EA (EA calculation), a "40" is forced into the J-field by the hardware during the A READ Dispatch. Figure A-13 shows the DRAM word for the ADD (270) instruction. Use the DRAM index to locate line 4091.

The A-field of the DRAM word is "5." This, ORed with the forced "40," results in "45." This is microcode address 45, just as in the MOVE example. Locate address 45 on line 2712; this is where the operands are fetched (see Figure A-14).

A "01" is in the "f" column of the SPEC field, a DRAM J Dispatch. Looking back at Figure A-13, notice that the J-field of the DRAM word is "504." Go to the microcode address index and locate address 504 at line 4098 (see Figure A-15).

This is where the ADD takes place. The macros state "A plus B (the two operands) into the AD." The SPEC field (Figure A-13) is a "5." The J-field of the current microinstruction is 170. These two are ORed, resulting in 175. Using the index again, locate address 175 on line 2749 (see Figure A-16).

The operand is stored in AC0 and the J-field leads the reader back to location 140 again, the NICOND Dispatch. The microcode is now ready for the next instruction.

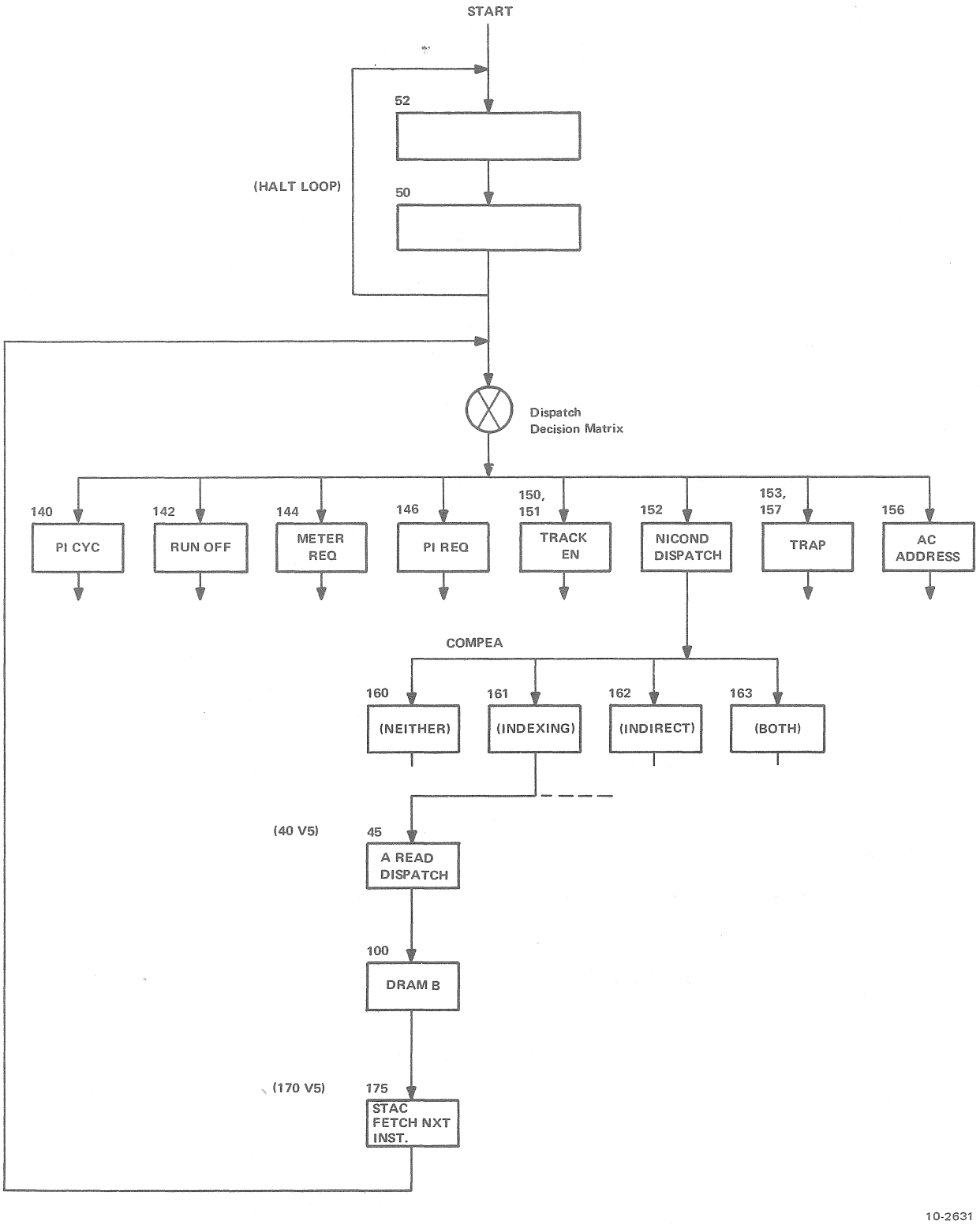



Figure A-11 ADD Instruction Flow Diagram

```
U0160,0000,3701,0000,0000,0204,0002,0000 ;2647  COMPEA: GEN AR, A READ              ;LOCAL
U0161,0000,0600,0002,4000,2224,0002,0000 ;2648 "  GEN AR + XR, INDEXED, A READ  ;LOCAL UNLESS XR> 0
```

Figure A-12 Microcode Address 160, 161

```
D 0270,5500,0504          ;4091  270:  R-PF,  AC,    J/ADD
```

Figure A-13 DRAM Word 270

```
                                  ;2711  BR/AR, FIN XFER, I FETCH  ;GET OPERAND, PREFETCH,
U0045,0000,3240,0043,0000,0226,0001,0400  ;2712  TIME 3T, I/R DISP, J/O  ;& START EXECUTE
```

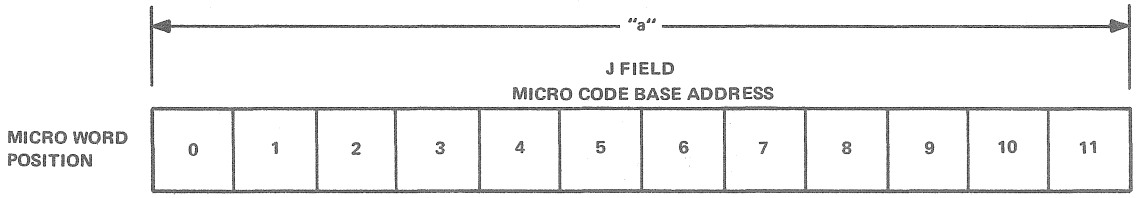
Figure A-14 Microcode Address 45

```
U0504,0170,0600,2000,0000,0025,1333,0000  ;4098  ADD:  AR  AR*AC0, AD/A+B, AD FLAGS, EXIT
```

Figure A-15 Microcode Address 504

```
U0175,0140,4001,0000,0403,0002,1006,0000  ;2749  STAC:  AC0_AR,NXT INSTR  ;NORMAL AND IMMEDIATE MODES
```

Figure A-16 Microcode Address 175




NOTES:

1. The J FIELD defines the base address to which this microinstruction jumps.

10-2637

Figure A-17 Microword "a" Field

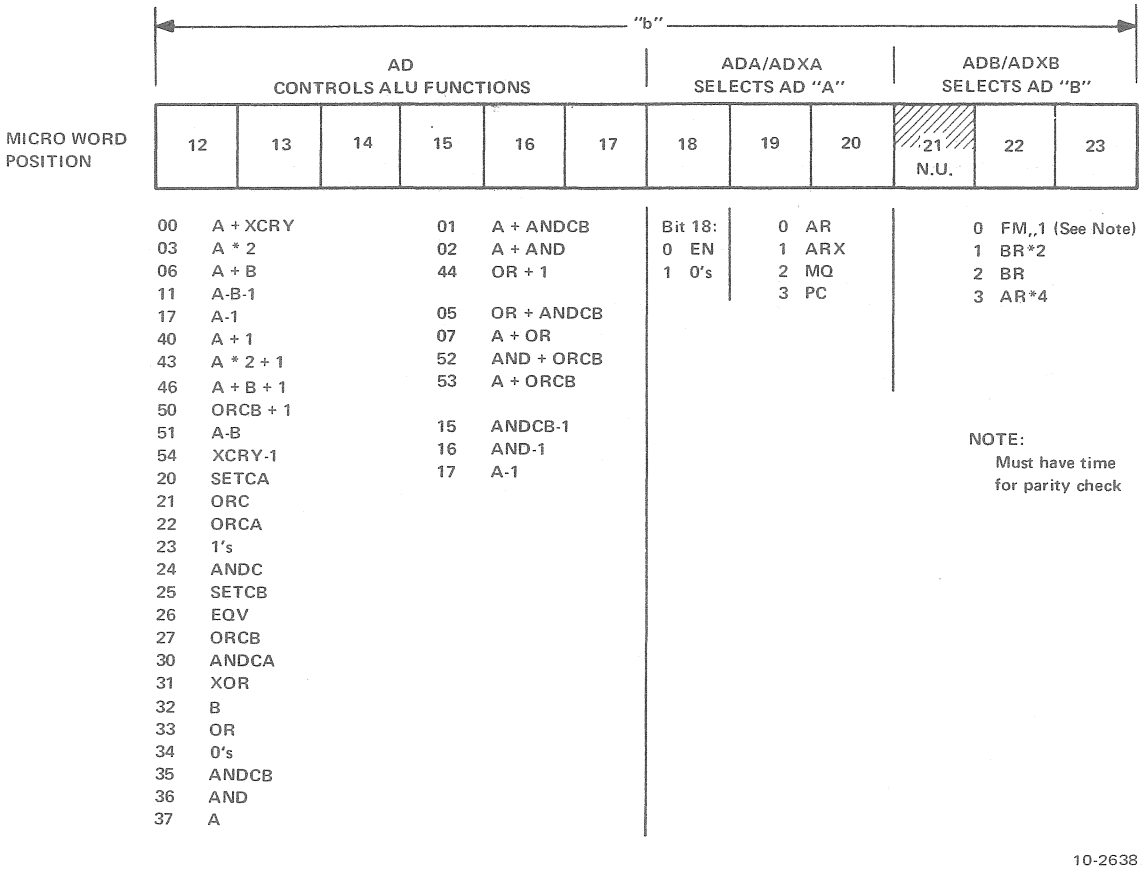



Figure A-18 Microword "b" Field

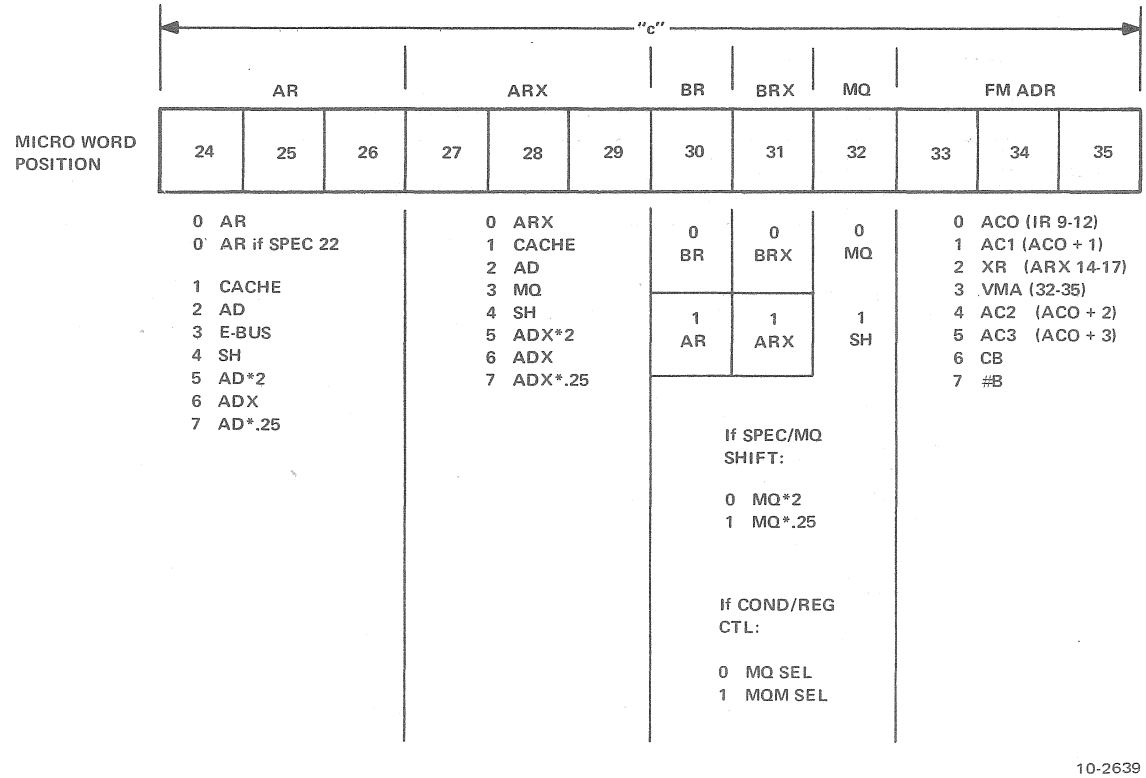
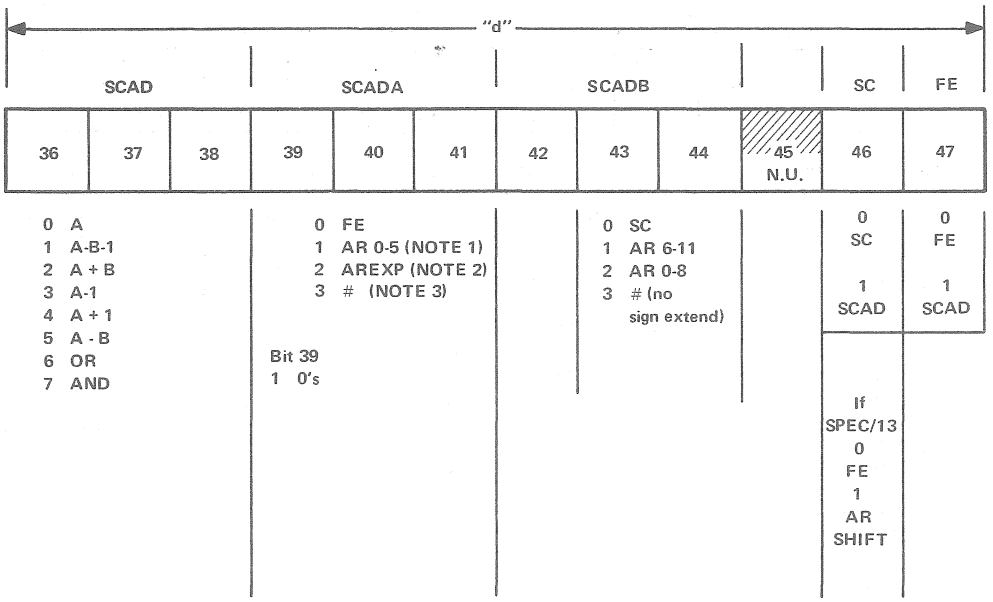


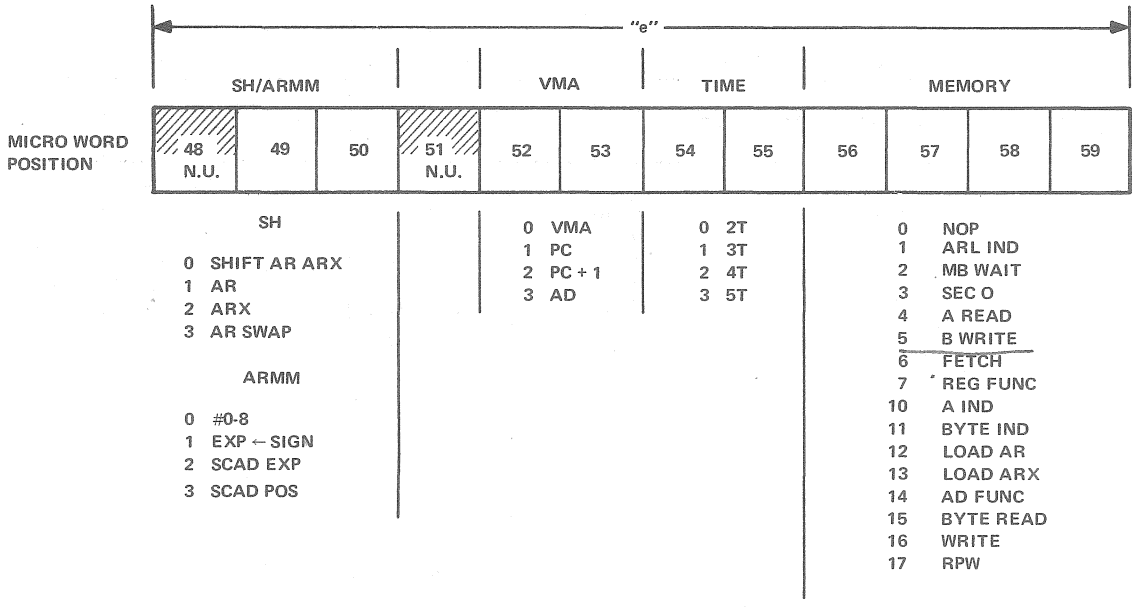

Figure A-19 Microword "c" Field




NOTES:
1. Byte Pointer Position Field
2. |AR (01-08)| XOR |AR00|
3. Sign extended with #00

10-2640

Figure A-20 Microword "d" Field




10-2641

Figure A-21 Microword "e" Field

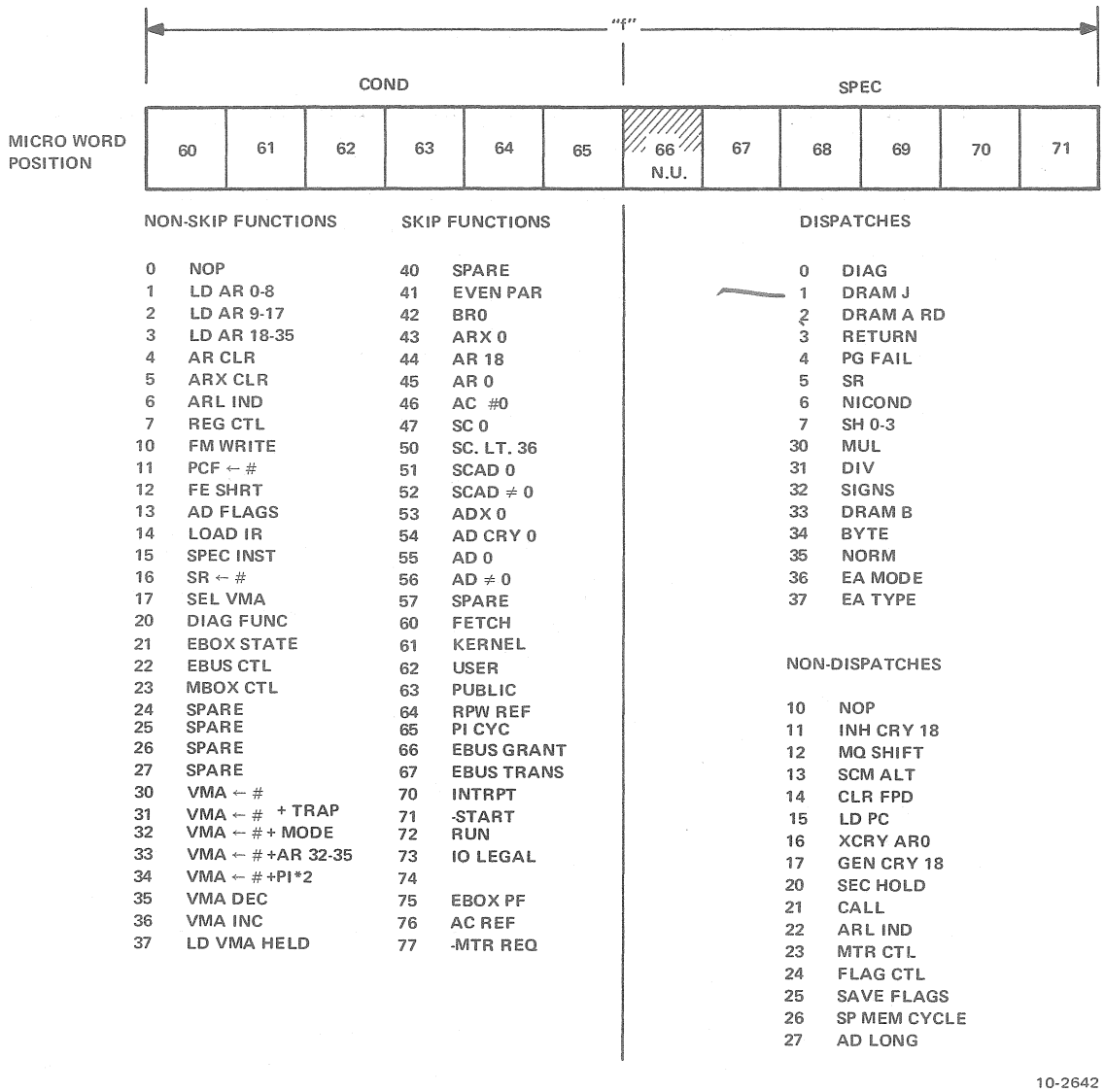



Figure A-22 Microword "f" Field

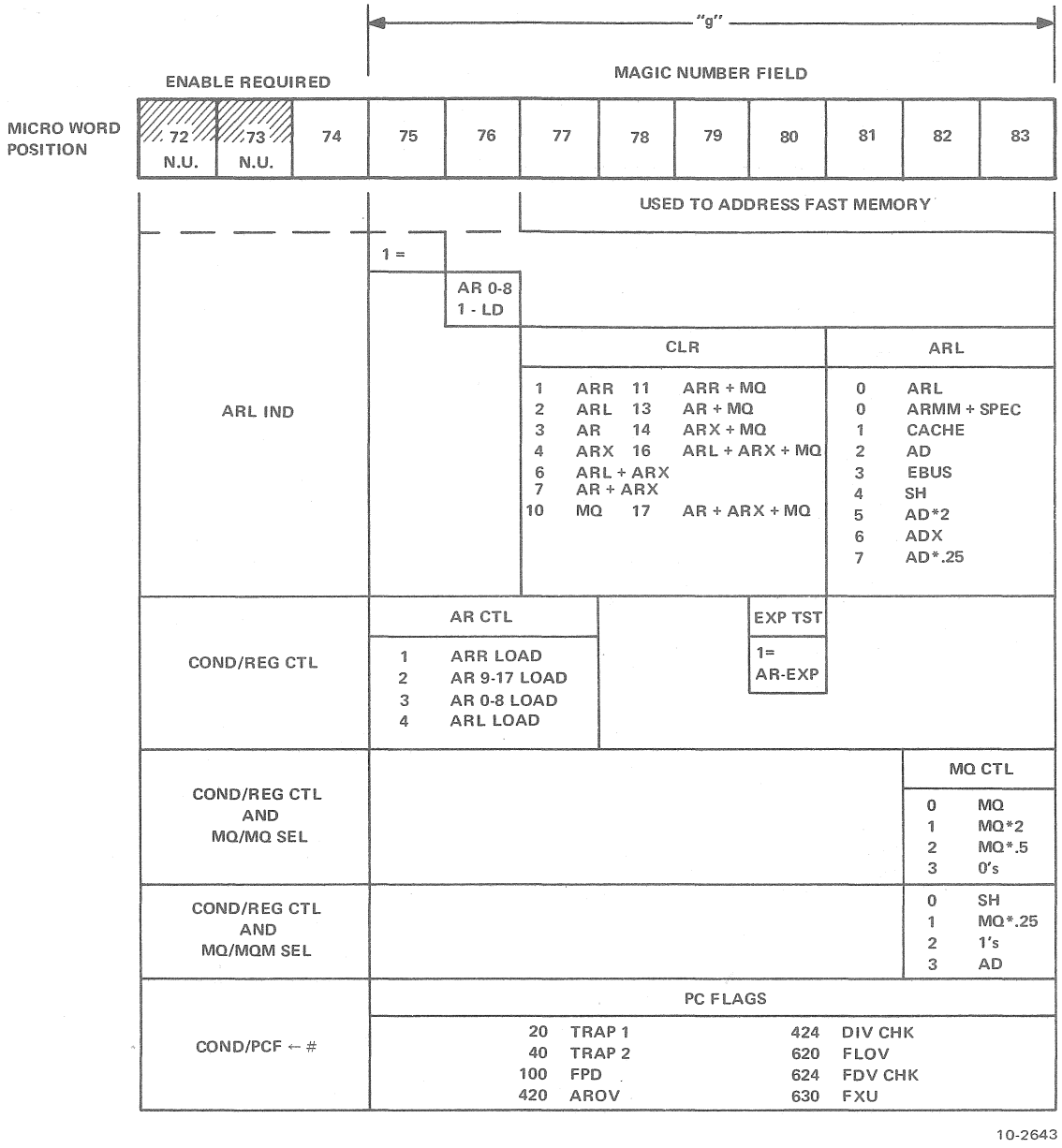
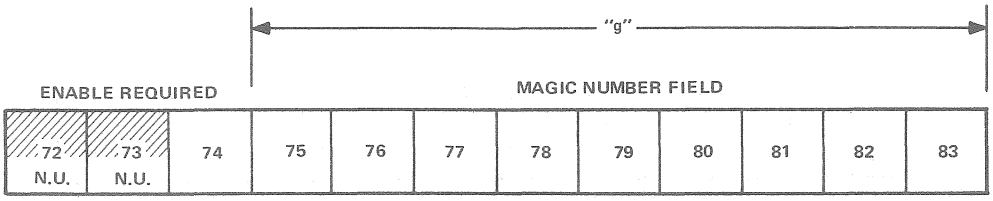



Figure A-23 Microword "g" Field (Magic Numbers)(Sheet 1 of 3)




| | | | | |
|---|---|---|---|---|
| SPEC/FLG CTL | FLAG CTL | | | |
| | 20 | SET FLAGS | 502 | DISMISS |
| | 412 | PORTAL | 602 | JFCL |
| | 420 | RSTR FLAGS | 622 | JFCL + LD |
| | 442 | HALT | | |
| COND/SPEC INST | SPEC INSTR | | | |
| | 4 | INSTR ABORT | 100 | IN H PC + 1 |
| | 10 | INTRPT IN H | 200 | KERNEL CYCLE |
| | 10 | CONT | 302 | HALTED |
| | 20 | PXCT | 310 | CONS XCT |
| | 40 | SXCT | 704 | SET PI CYCLE |
| MEM/FETCH | FETCH | | | |
| | 201 | COMP | 400 | UNCOND |
| | 202 | SKIP | 502 | JUMP |
| | 203 | TEST | 503 | JFCL |
| SPEC/SP MEM CYCLE | | | | |
| | 2 | CACHE IN H | 111 | EPT |
| | 10 | EPT EN | 200 | USER |
| | 20 | UPT EN | 221 | UPT |
| | 31 | PT | 400 | FETCH |
| | 40 | SEC 0 | 431 | PT FETCH |
| | 100 | EXEC | 511 | EPT FETCH |
| | 101 | UNPAGED | 621 | UPT FETCH |
| MEM/REG FUNC | MREG FUNC | | | |
| | 7 | SBUS DIAG | 505 | WR REFILL RAM |
| | 140 | MPA | 601 | LOAD CCA |
| | 502 | READ UBR | 602 | LOAD UBR |
| | 503 | READ EBR | 603 | LOAD EBR |
| | 504 | READ ERA | | |
| COND/MBOX CTL | MBOX CTL | | | |
| | 00 | NORMAL | 21 | CLR PT LINE |
| | 01 | PT DIR CLR | 100 | SET I/O PF ERR |
| | 10 | PT WR | 200 | SET PAGE FAIL |
| | 20 | PT DIR WR | | |

10-2644

Figure A-23 Microword "g" Field (Magic Numbers)(Sheet 2 of 3)

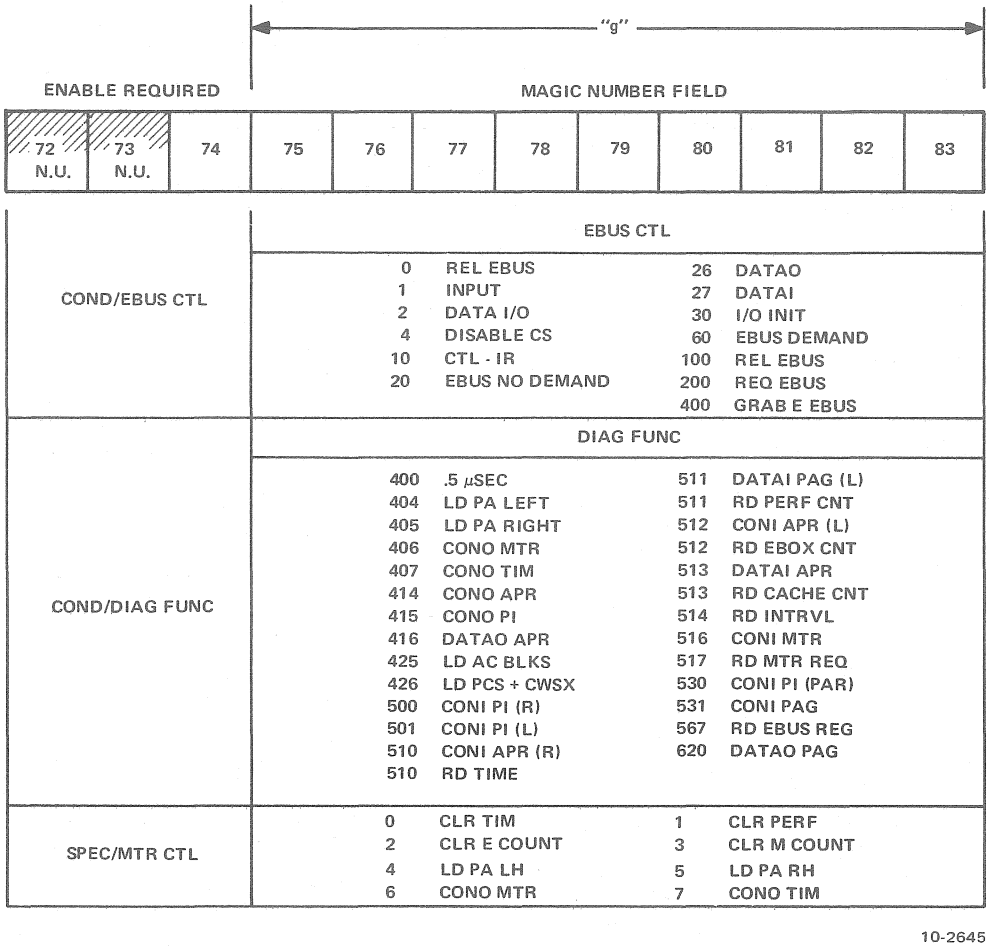



Figure A-23 Microword "g" Field (Magic Numbers)(Sheet 3 of 3)

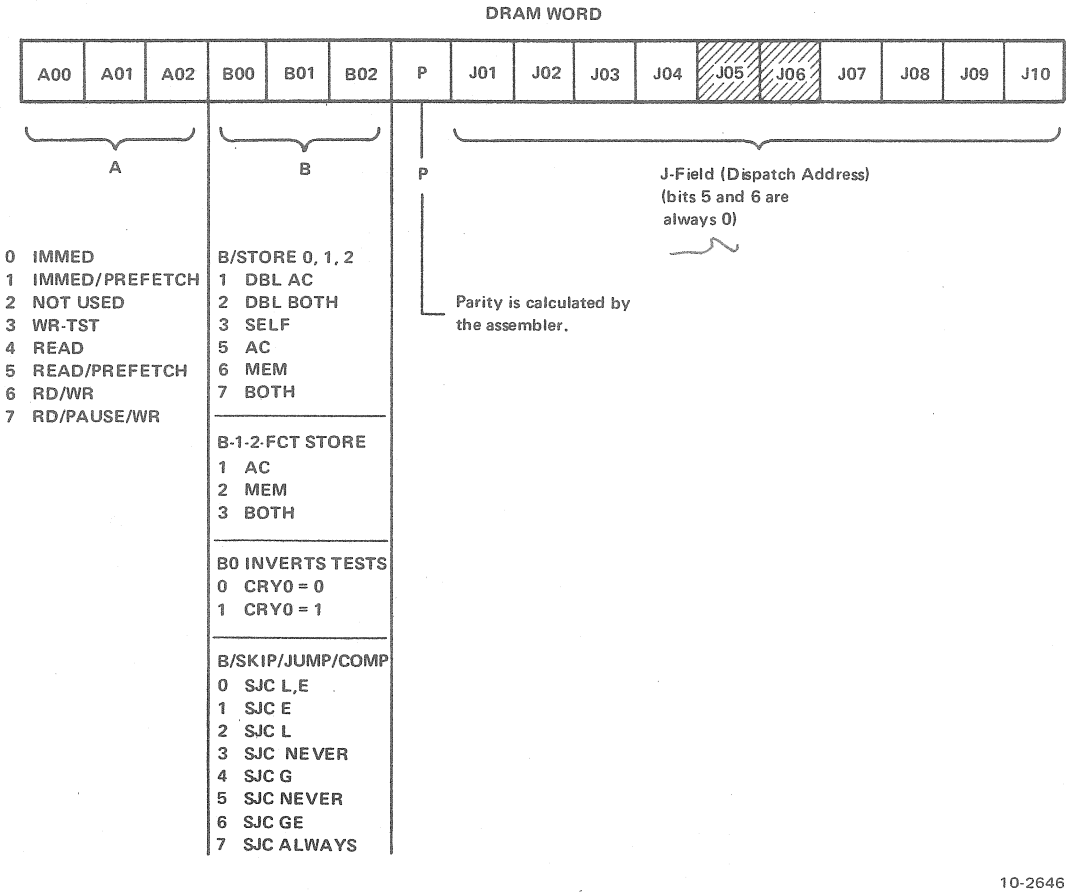



Figure A-24 DRAM Word Format

# APPENDIX B
# ABBREVIATIONS AND MNEMONICS

| | A |
|---|---|
| AC | Accumulator |
| ACKN | Acknowledge |
| ACT | Action |
| AD | Adder |
| ADA | Adder A |
| ADB | Adder B |
| ADR | Address |
| ADX | Adder Extension |
| AF | Action Flag |
| ALT | Alternate |
| ALU | Arithmetic Logic Unit |
| APR | Arithmetic Processor Register |
| AR | Arithmetic Register |
| ARL | Arithmetic Register Left |
| ARM | Arithmetic Register Mixer |
| ARMM | Arithmetic Register Mixer Mixer |
| ARR | Arithmetic Register Right |
| ARX | Arithmetic Register Extension |
| ARXL | Arithmetic Register Extension Left |
| ARXM | Arithmetic Register Extension Mixer |
| ARXR | Arithmetic Register Extension Right |

| | B |
|---|---|
| BOOLE | Boolean |
| BR | Buffer Register |
| BRK | Break |
| BRX | Buffer Register Extension |
| BUF | Buffer |

| | C |
|---|---|
| CRAM | Control RAM |
| CRY | Carry |
| CS | Controller Select |
| CSH | Cache |
| CTL | Control |
| CTOM | Controller-to-Memory or Cache-to-Memory |
| CTR | Counter |
| CWSX | Called With Special Execute |
| CYC | Cycle |

| | D |
|---|---|
| D | Data |
| DIAG | Diagnostic |
| DIR | Directory |
| DIS | Disable |
| DISP | Dispatch |
| DIV | Divide |
| DRAM | Dispatch RAM |

| | E |
|---|---|
| E | Effective Address |
| E to T | ECL to TTL |
| EBR | Executive Base Register |
| EBUS | Execution Bus |
| ECL | Emitter-Coupled Logic |
| EDP | EBox Data Path |
| EN | Enable |
| ENA | Enable |
| ERR | Error |
| ERA | Error Address |
| EPT | Executive Process Table |
| EX | Extension |
| EXP | Exponent |
| EXT | External |
| EXT TRA | External |
| REC | Transfer Receiver |

| | |
|---|---|
| | **F** |
| F | Function |
| FE | Floating Exponent |
| FE | Front End |
| FLG | Flag |
| FM | Fast Memory |
| FOV | Floating Overflow |
| FPD | First Part Done |
| FPD | Floating Point Divide |
| FUNC | Function |
| FXU | Floating Exponent Underflow |
| | **G, H** |
| G | Gated |
| GE | Greater or Equal |
| GEN | Generate |
| H | High |
| | **I** |
| INC | Increment |
| INH | Inhibit |
| INSTR | Instruction |
| INT | Internal |
| INTR | Interrupt |
| INVAL | Invalid |
| IOT | Input/Output Transfer |
| IR | Instruction Register |
| | **J, K, L** |
| J | Jump |
| L | Low |
| LRU | Least Recently Used |
| | **M** |
| MB | Memory Buffer |
| MBC | MBox Control |
| MBX | MBox Control |
| MBZ | MBox Control |
| MCL | Memory Control |
| MEM | Memory |
| MHz | Mega Hertz |
| MIX | Mixer |
| MQ | Multiplier Quotient |
| MQM | Multiplier Quotient Mixer |
| MR | Master |
| MTR | Meter |
| | **N, O** |
| NICOND | Next Instruction Condition |
| NXM | Non-Existent Memory |
| NXT | Next |
| OP | Operation (code) |
| OVN | Overrun |
| | **P, Q** |
| PA | Physical Address |
| PAG | Pager |
| PAR | Parity |
| PC | Program Counter |
| PCF_# | Previous Context Flags from Number |
| PCP | Previous Context Public |
| PC | Program Counter |
| PERF | Performance |
| PF | Page Fault |
| PGRF | Page Refill |
| PI | Priority Interrupt |
| PIA | Priority Interrupt Assignment |
| PIH | Priority Interrupt Hold |
| PMA | Physical Memory Address |
| PREV | Previous |
| PT | Page Table |
| PWR | Power |
| | **R** |
| RAM | Random Access Memory |
| RD | Read |
| RE | Receive ECL |
| REC | Receive |
| REF | Reference |
| REG | Register |
| REL | Release |
| REQ | Request |
| RESP | Response |
| RET | Return |
| RIP | Request in Progress |
| RQ | Request |

| S | |
|---|---|
| S ADR P | Storage Address Parity |
| SBR | Subroutine |
| SBUS | Storage Bus |
| SC | Shift Count |
| SCAD | Shift Count Adder |
| SCADA | Shift Count Adder A |
| SCADB | Shift Count Adder B |
| SCD | Shift Count Adder |
| SCM | Shift Count Mixer |
| SEL | Select |
| SH | Shifter |
| SHRT | Shift Right |
| SIM | Simulate |
| SP | Special |
| SPEC | Special |
| SR | State Register |
| SYNC | Synchronize |

| T | |
|---|---|
| T to E | TTL to ECL |
| TE | Transmit ECL |
| T | Time |
| TRA | Transfer |
| TTL | Transistor-Transistor Logic |

| U, V | |
|---|---|
| UBR | User Base Register |
| UCODE | Micro Code |
| VAL | Valid |
| VMA | Virtual Memory Address |
| XFER | Transfer |
| XR | Index Register |

| W, X, Y, Z | |
|---|---|
| WARN | Warning |
| WC | Word Count |
| WD | Word |
| WR | Write |

# Index

# Reader's Comments

EBOX INSTRUCTION EXECUTION UNIT
UNIT DESCRIPTION
EK-EBOX-UD-003

**Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our publications.**

What is your general reaction to this manual? In your judgment is it complete, accurate, well organized, well written, etc.? Is it easy to use? _____

_____

_____

_____

What features are most useful? _____

_____

_____

_____

What faults do you find with the manual? _____

_____

_____

_____

Does this manual satisfy the need you think it was intended to satisfy? _____

Does it satisfy *your* needs? _____ Why? _____

_____

_____

_____

Would you please indicate any factual errors you have found. _____

_____

_____

_____

Please describe your position. _____

Name _____ Organization _____

Street _____ Department _____

City _____ State _____ Zip or Country _____

CUT OUT ON DOTTED LINE

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

Digital Equipment Corporation
Technical Documentation Department
146 Main Street
Maynard, Massachusetts 01754