EK-EBOX-UD-003

EBOX
INSTRUCTION EXECUTION UNIT
UNIT DESCRIPTION

digital equipment corporation • marlborough, massachusetts

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

The material in this manual is for informational purposes and is subject to change without notice. Digital Equipment Corporation assumes no responsibility for any errors which may appear in this manual.

Printed in U.S.A.

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

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

The material in this manual is for informational purposes and is subject to change without notice. Digital Equipment Corporation assumes no responsibility for any errors which may appear in this manual.

Printed in U.S.A.

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

CONTENTS

CONTENTS

CONTENTS (Cont)

ILLUSTRATIONS

ILLUSTRATIONS (Cont)

ILLUSTRATIONS (Cont)

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.

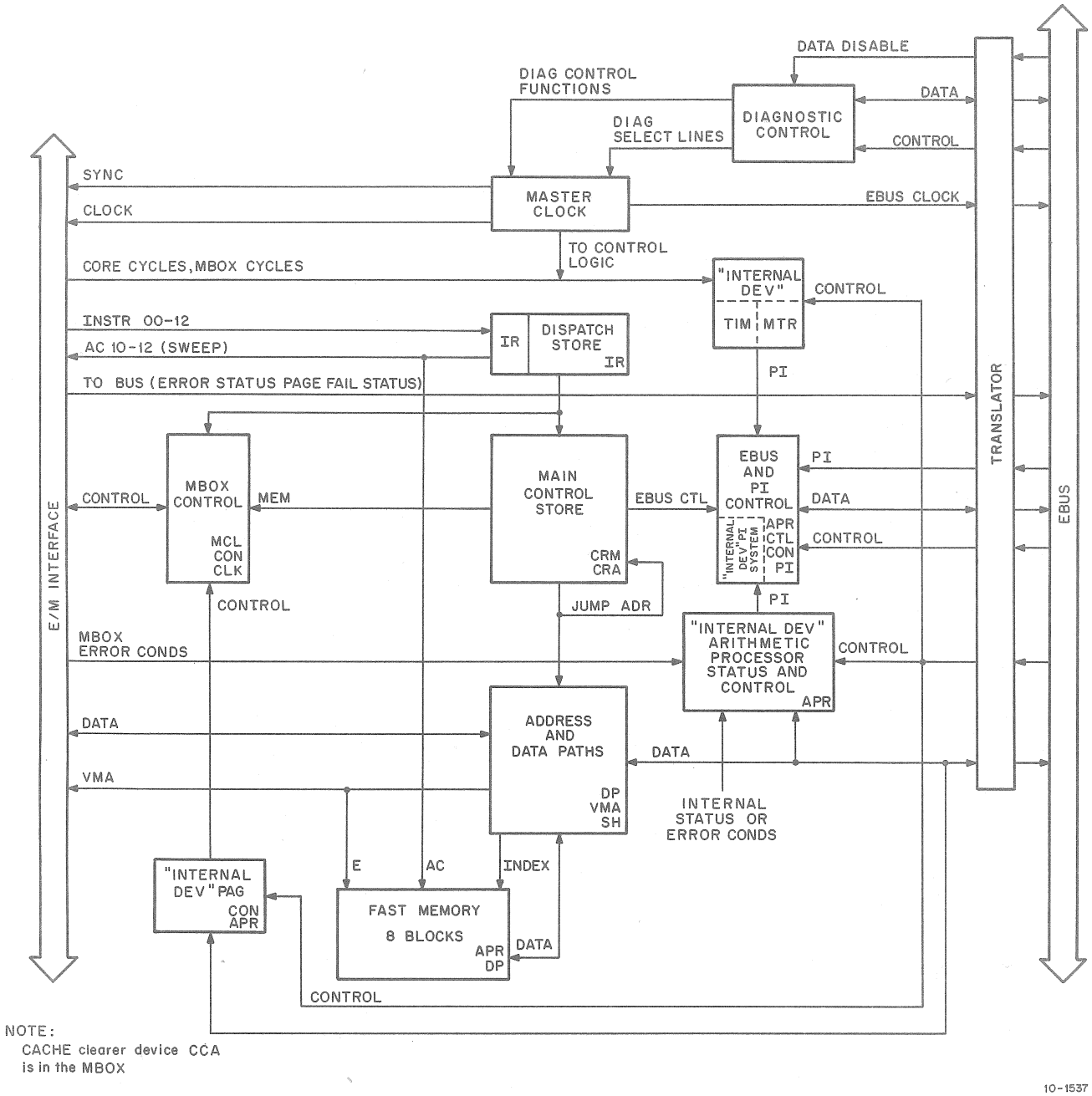


Figure 1-1 EBox Simplified Block Diagram

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.

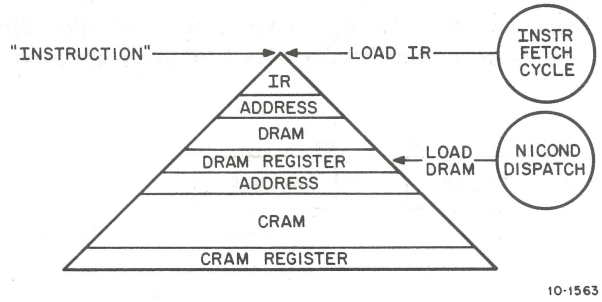


Figure 1-2 Control Pyramid

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.

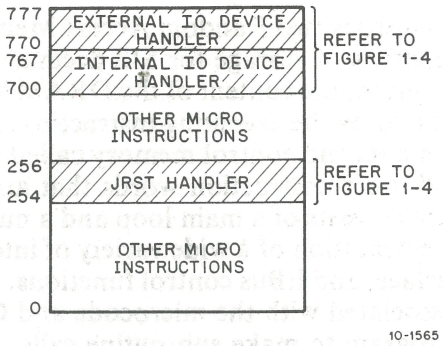


Figure 1-3 DRAM I/O, JRST

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.

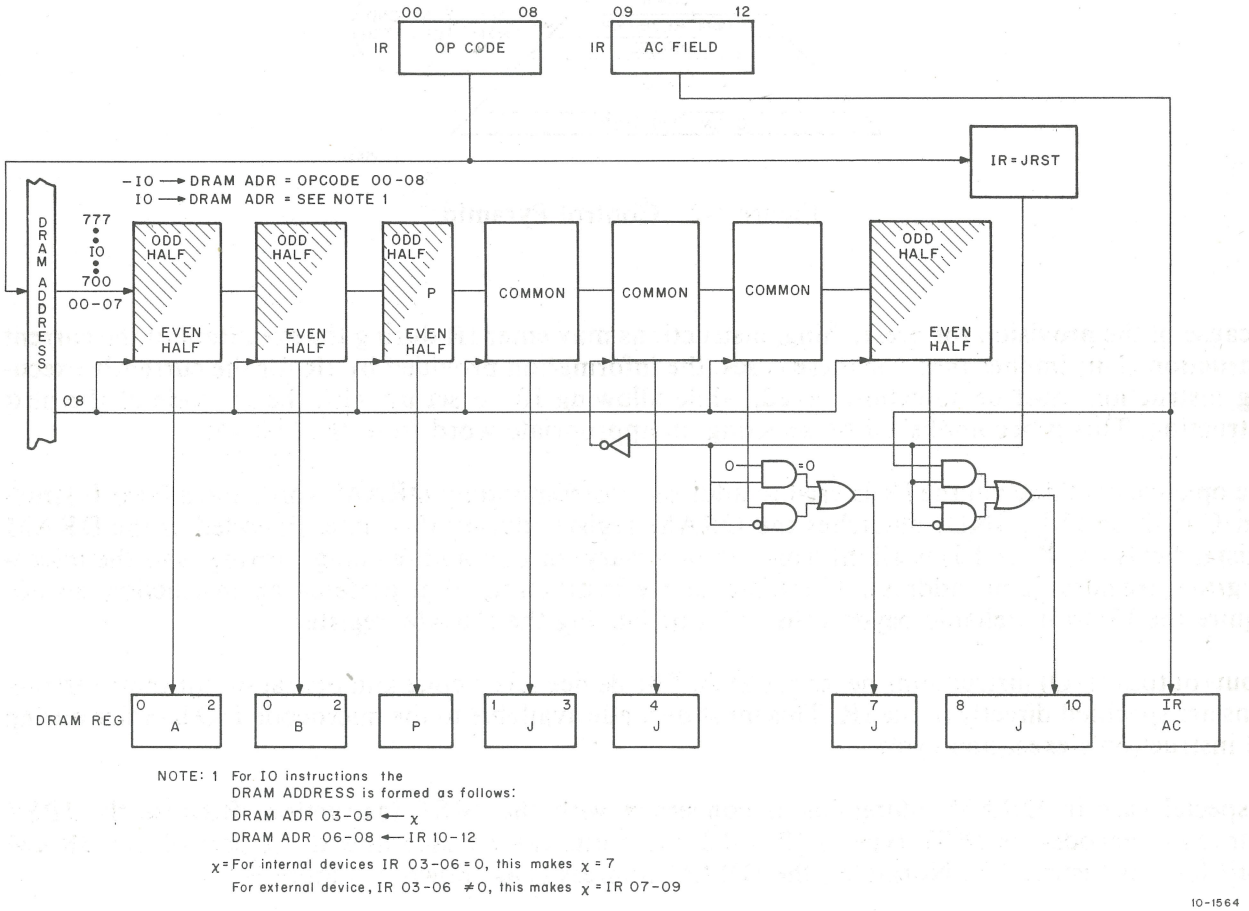


Figure 1-4 DRAM Organization

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

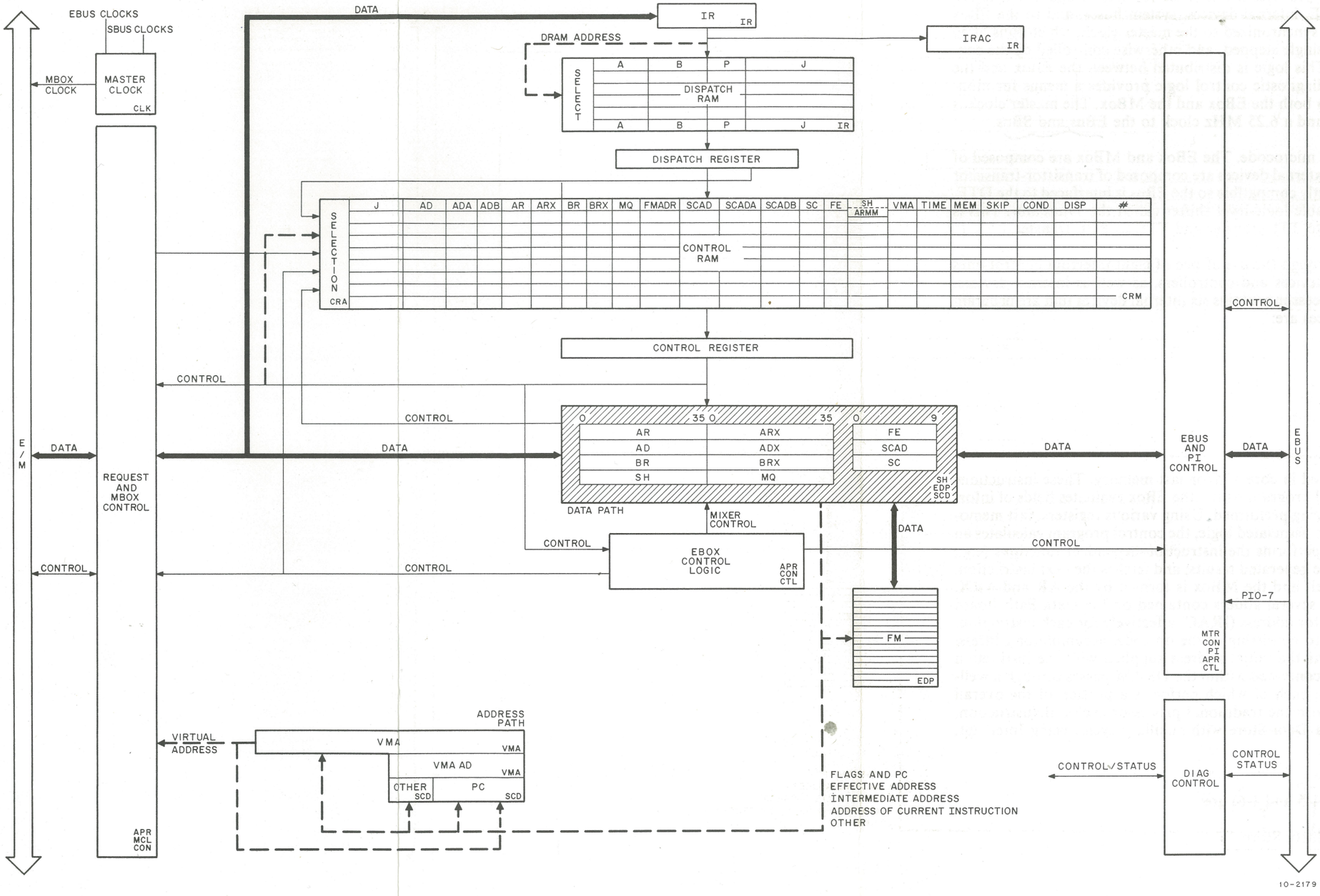
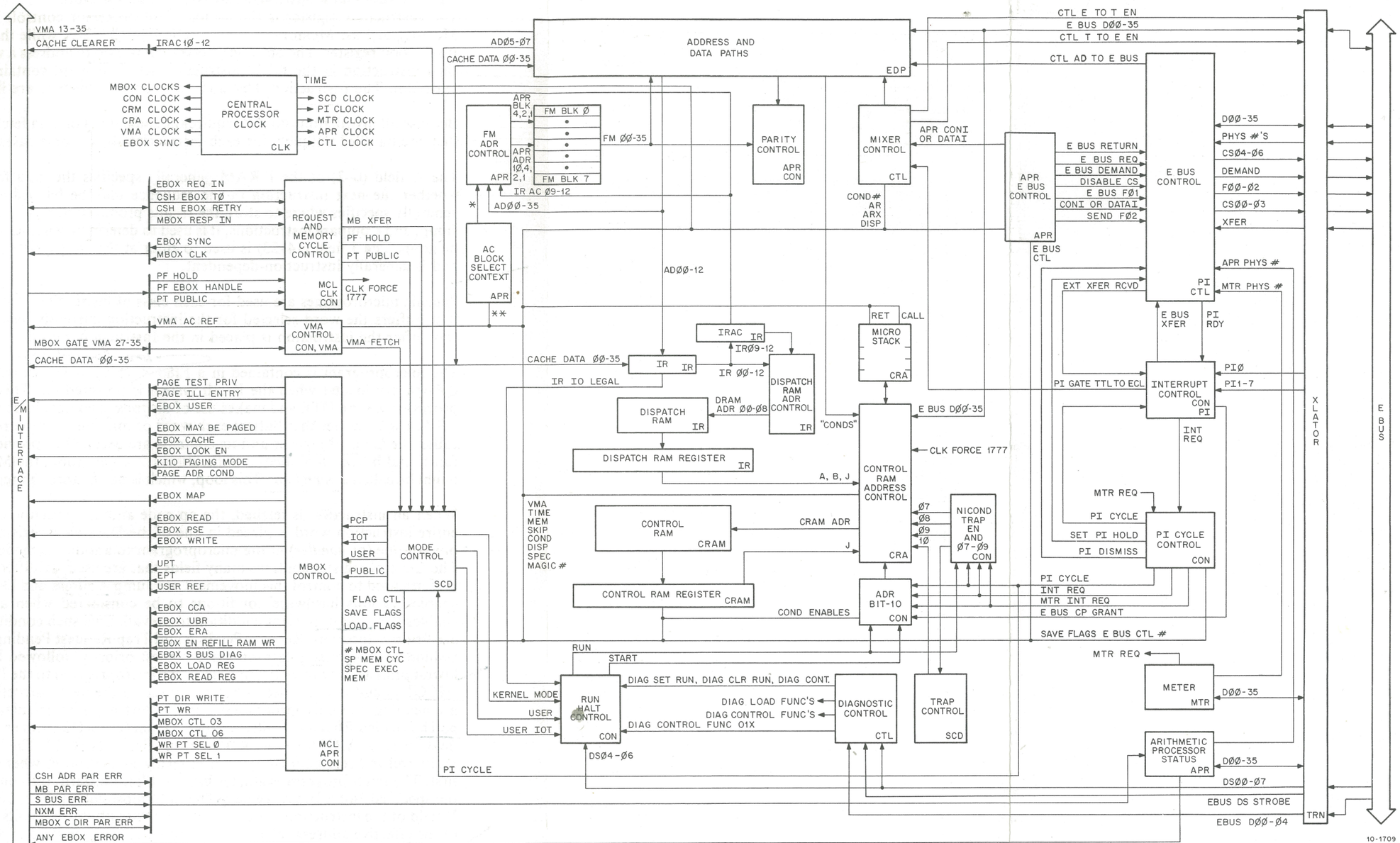


Figure 1-5 EBox RAM Structures, Interfaces, and Controls Block Diagram



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.

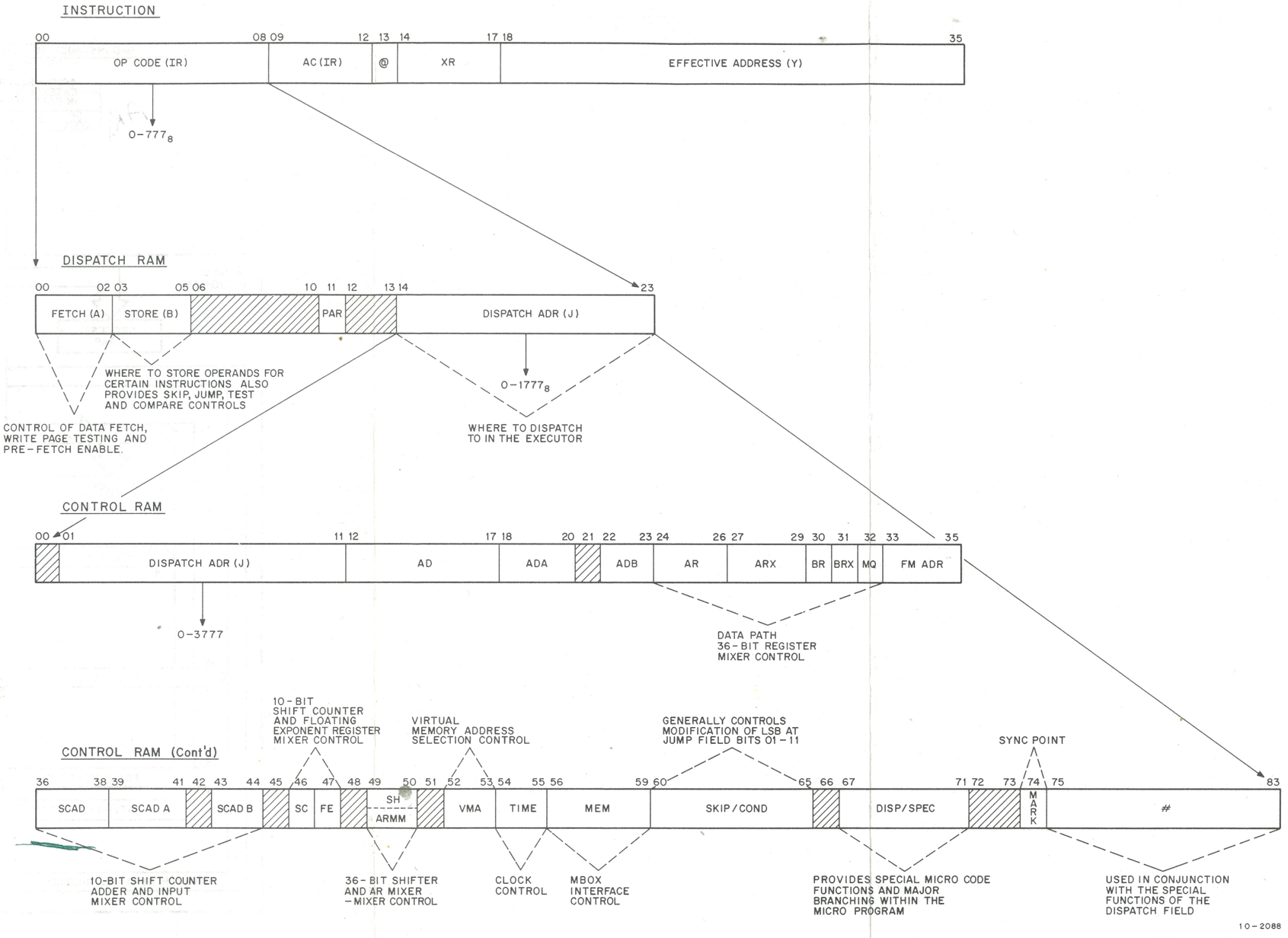


Figure 1-7 Instruction, Dispatch, and Control Formats

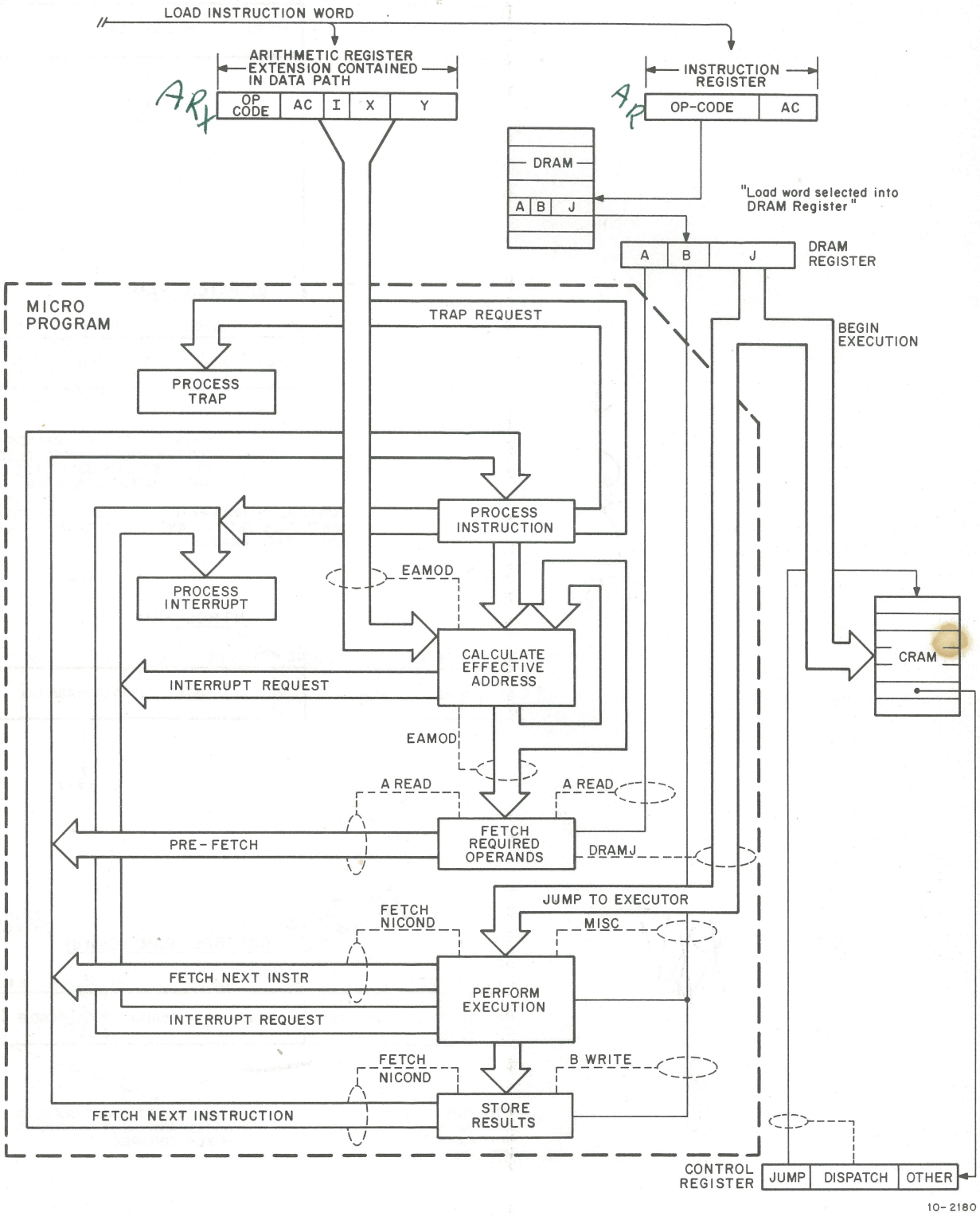


Figure 1-8 Microprogram Main Loop

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).

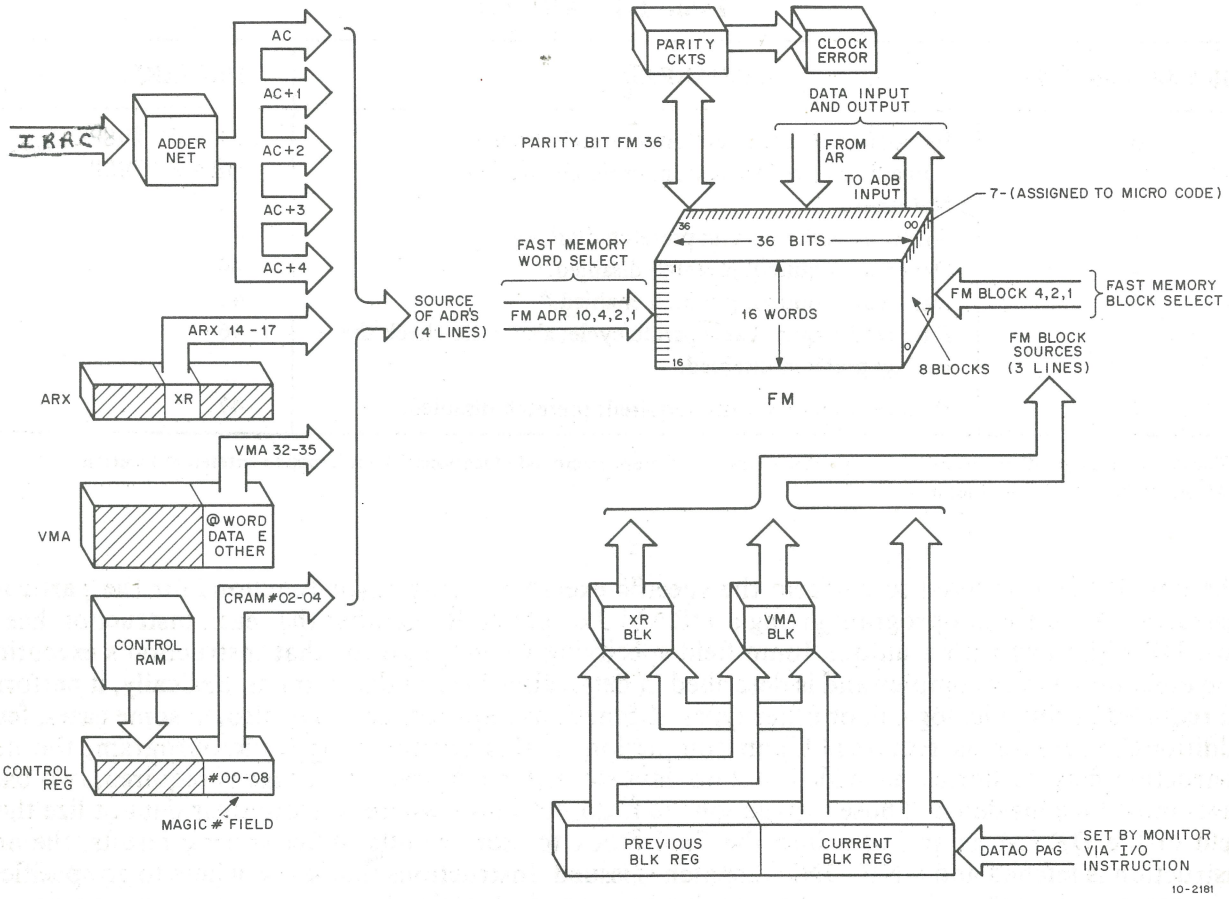


Figure 1-9 Basic Fast Memory Structure

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

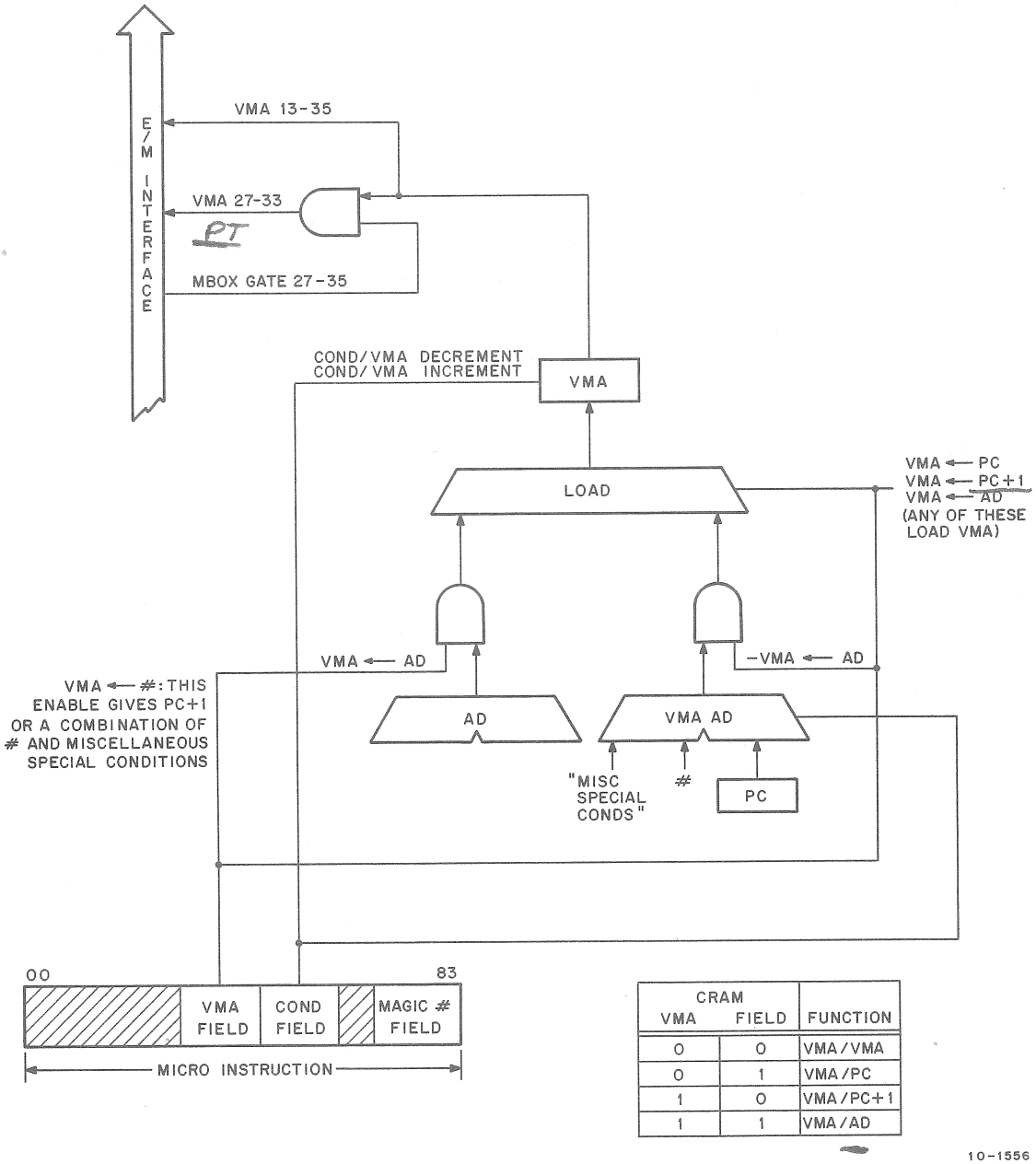
The EBox performs a program by executing instructions retrieved from locations addressed by the PC, a 23-bit register contained in the EBox data path. At the beginning of each instruction, PC is incremented by one so that it normally contains an address one greater than the current instruction. Sequential 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.

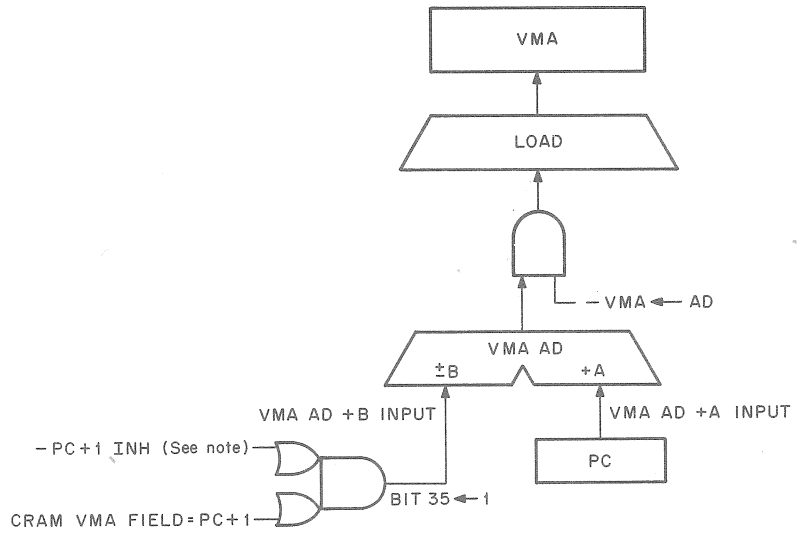


Figure 1-11 PC + 1 Function

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.

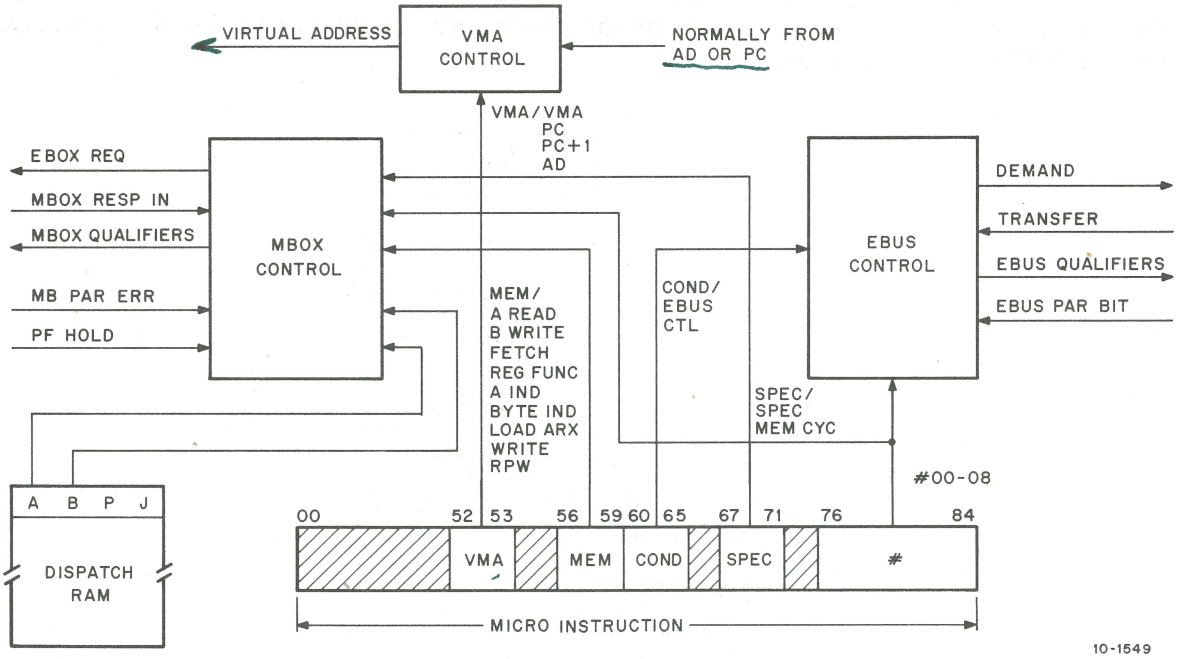


Figure 1-12 MBox-VMA-EBUS Control Simplified

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.

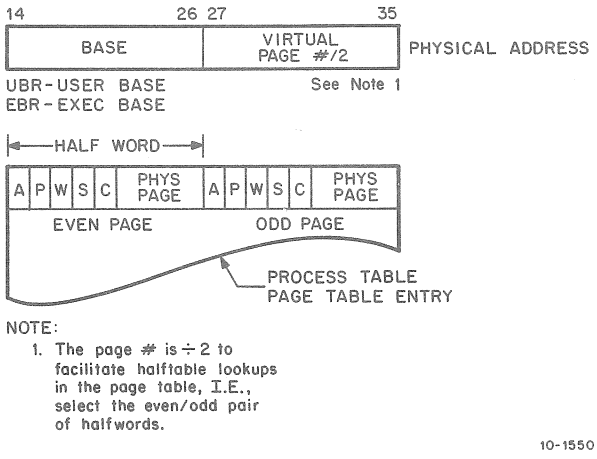


Figure 1-13 Page Table Access

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.

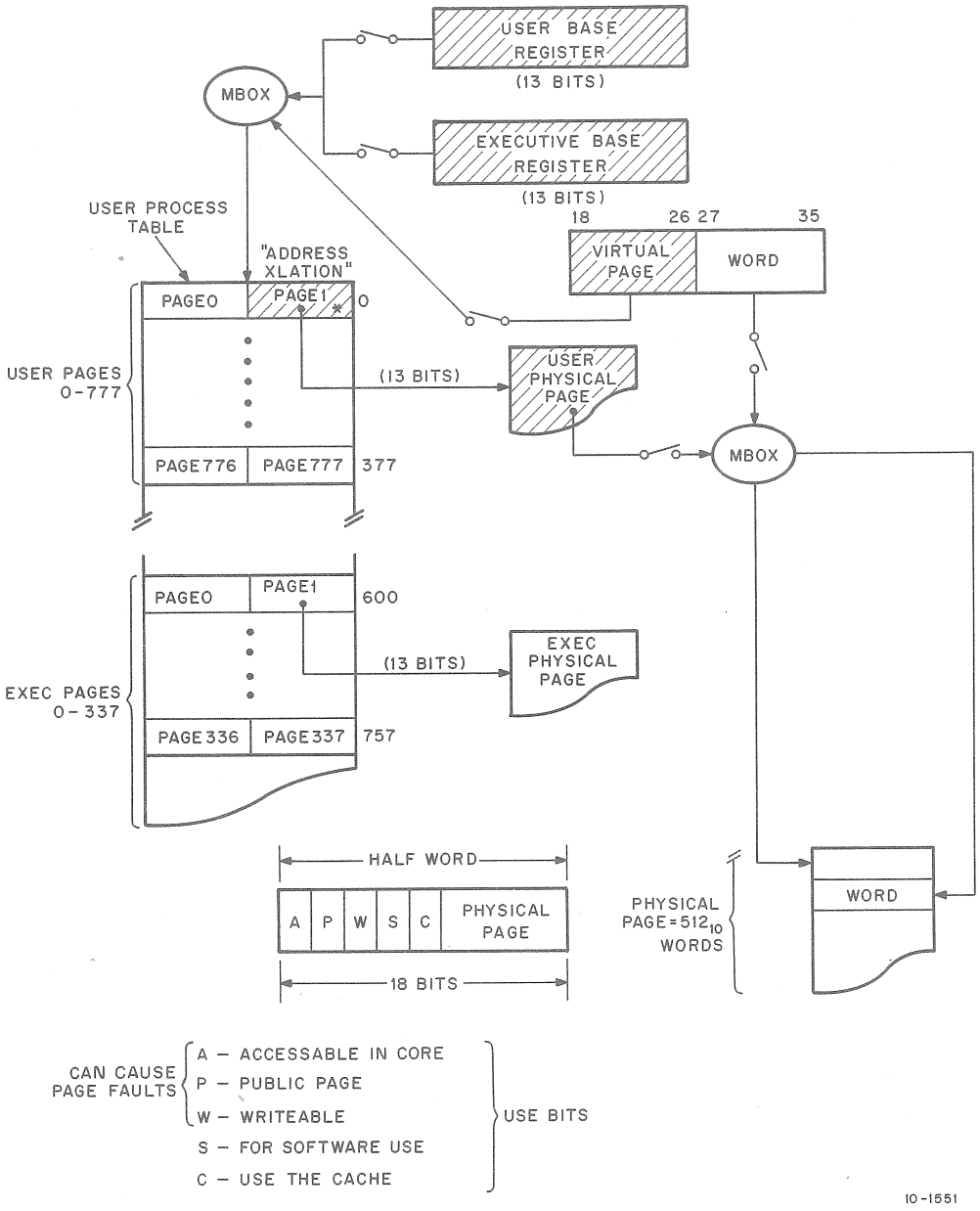


Figure 1-14 KI Style Paging

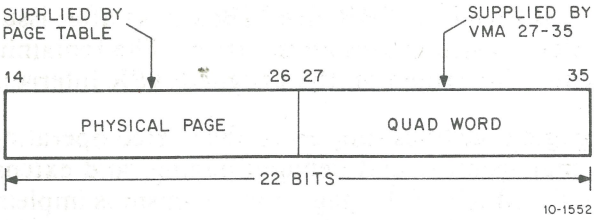


Figure 1-15 Physical Memory Address Format

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.

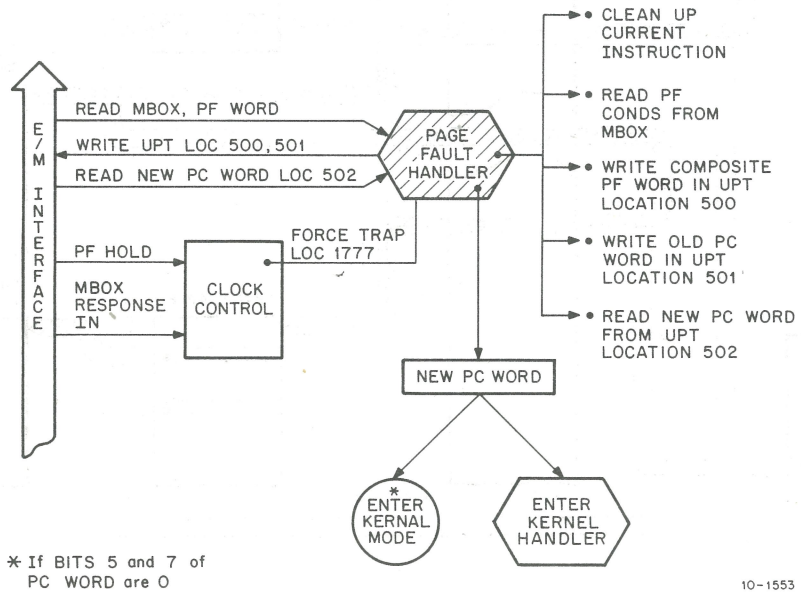


Figure 1-16 Page Fault Overview

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.

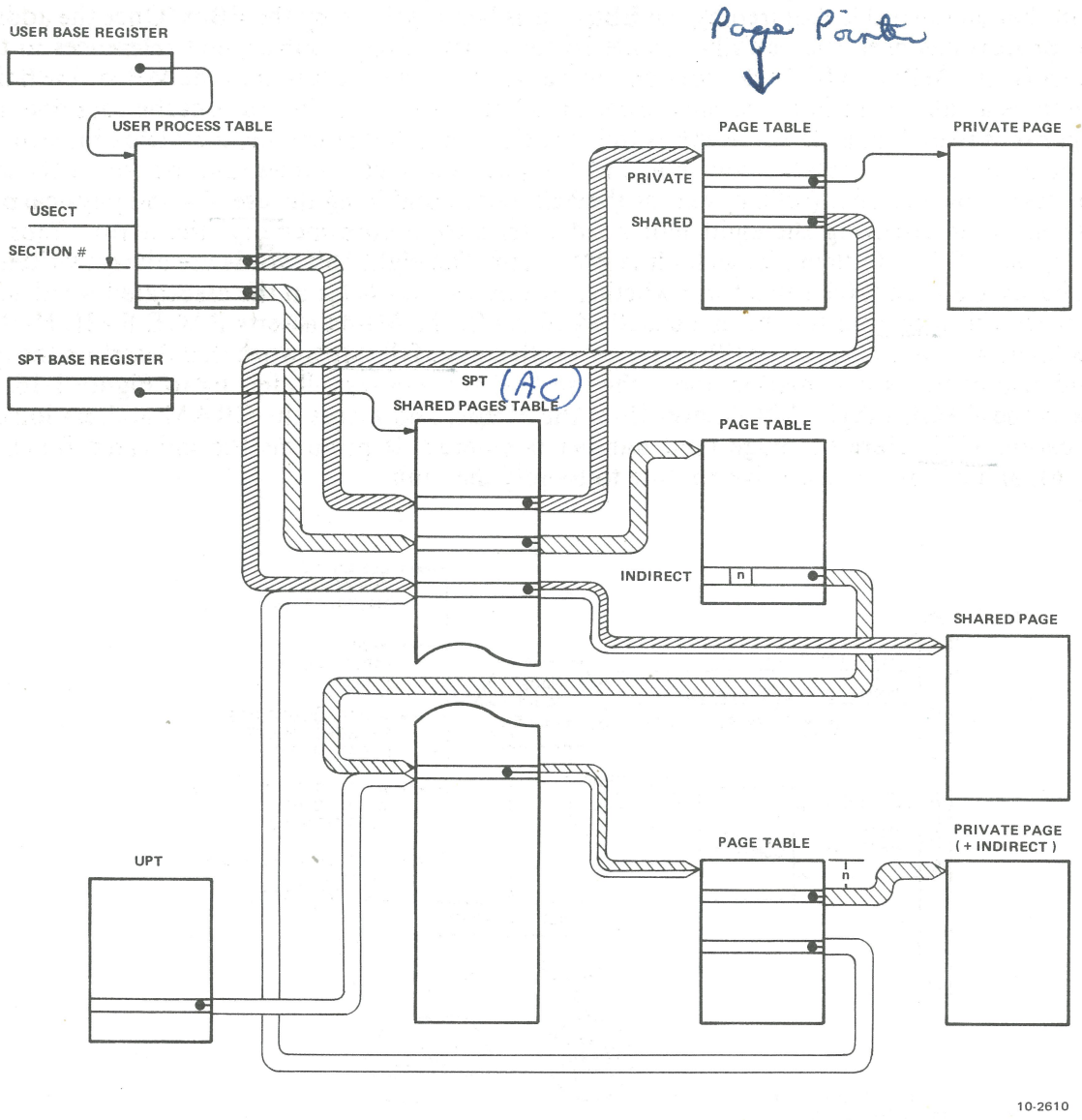


Figure 1-17 KL Paging Layout

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.

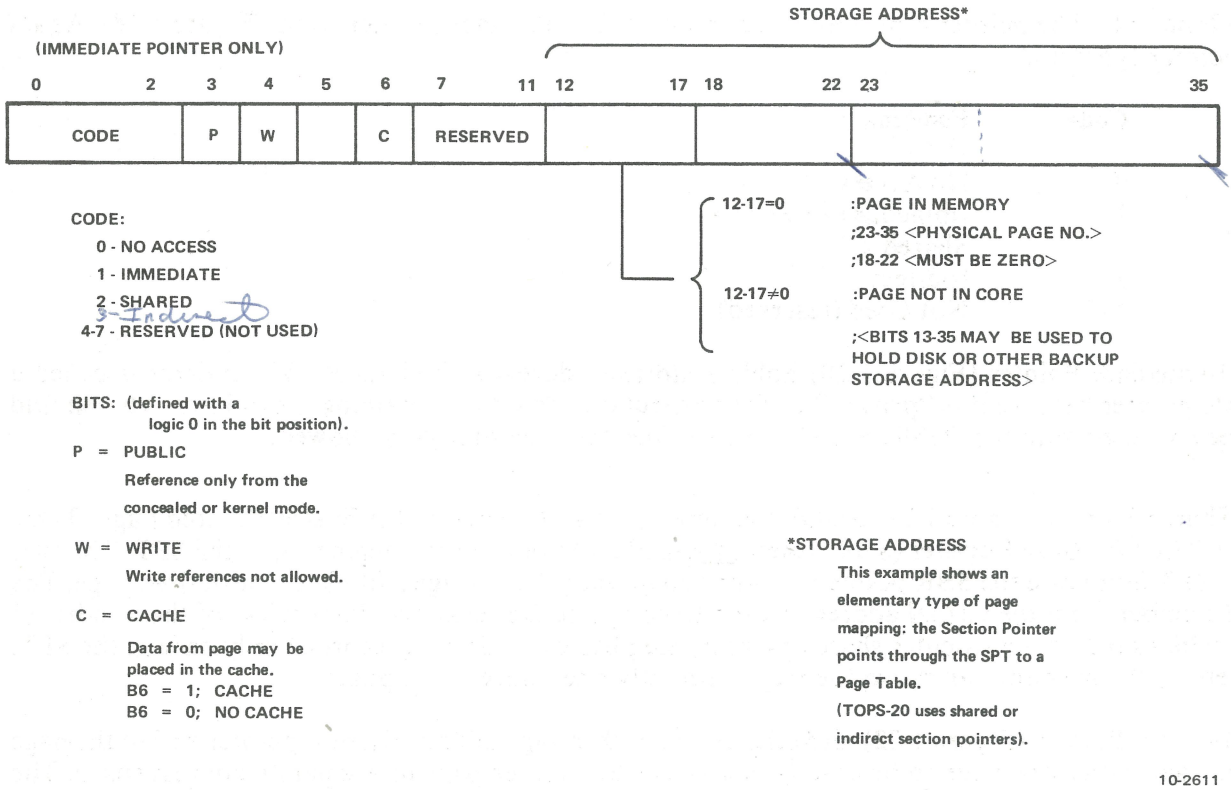


Figure 1-18 Page Mapping (Virtual to Physical)

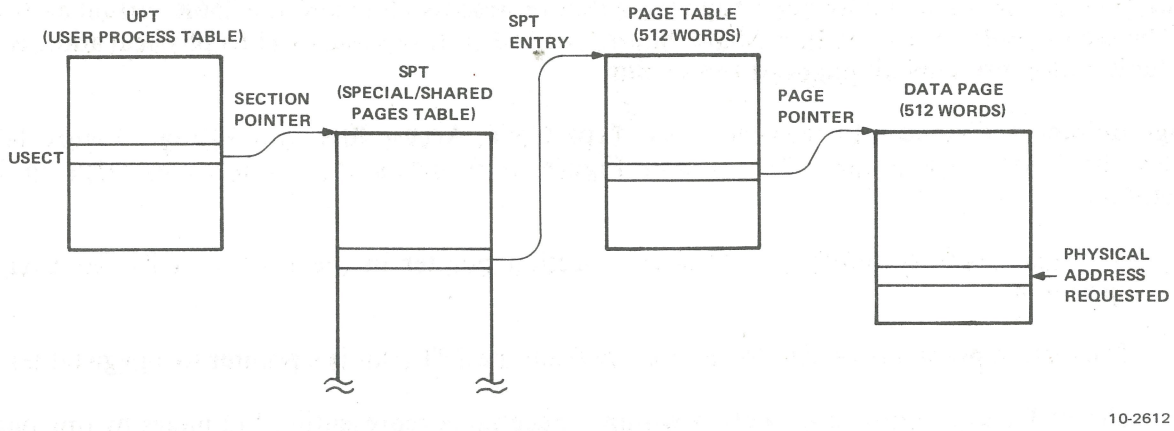


Figure 1-19 Typical Paging Path

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)

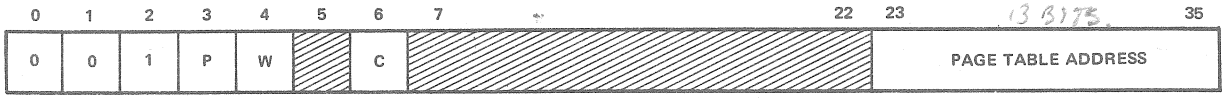
The Immediate Pointer (Figure 1-20) holds a storage address in bits 12–35. The pointer is called a private pointer because it is "private" to the particular page table containing the pointer. This should not be confused with the Public bit, which describes the type of access allowed.

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)



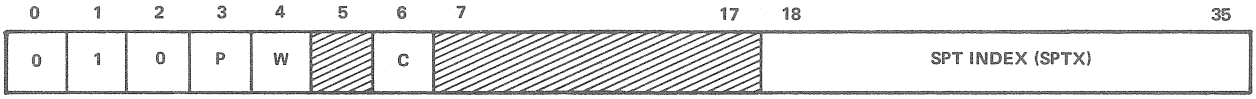
BIT	DEFINITION	DESCRIPTION
00-02	Pointer Type	A 1_8 in this field defines the Immediate Section Pointer.
03	Public Bit	If this bit is off (0), the page may only be referenced by programs running in Concealed or Kernel Mode.
04	Write Bit	When set, allows write references to be executed to the page.
05	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

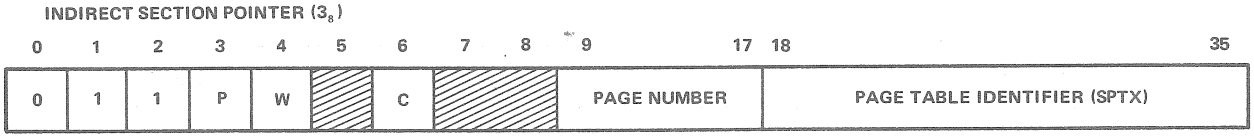
SHARED SECTION POINTER (2_8)



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



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

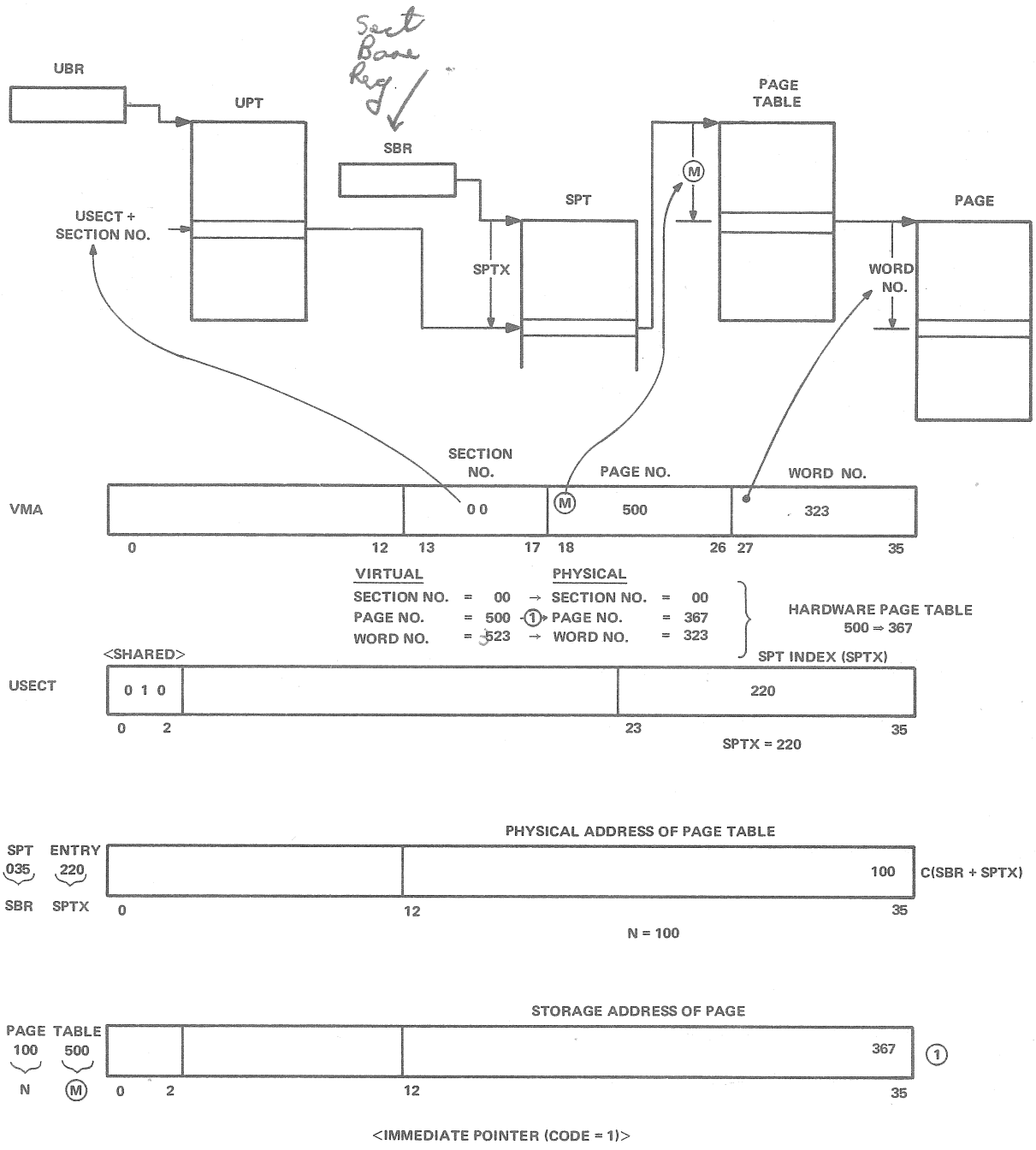
The Indirect chaining may be arbitrary in depth, but the PI will break out of indirect chain and restart after the PI to service a priority interrupt in the case of long direct chains or indirect loops.

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.

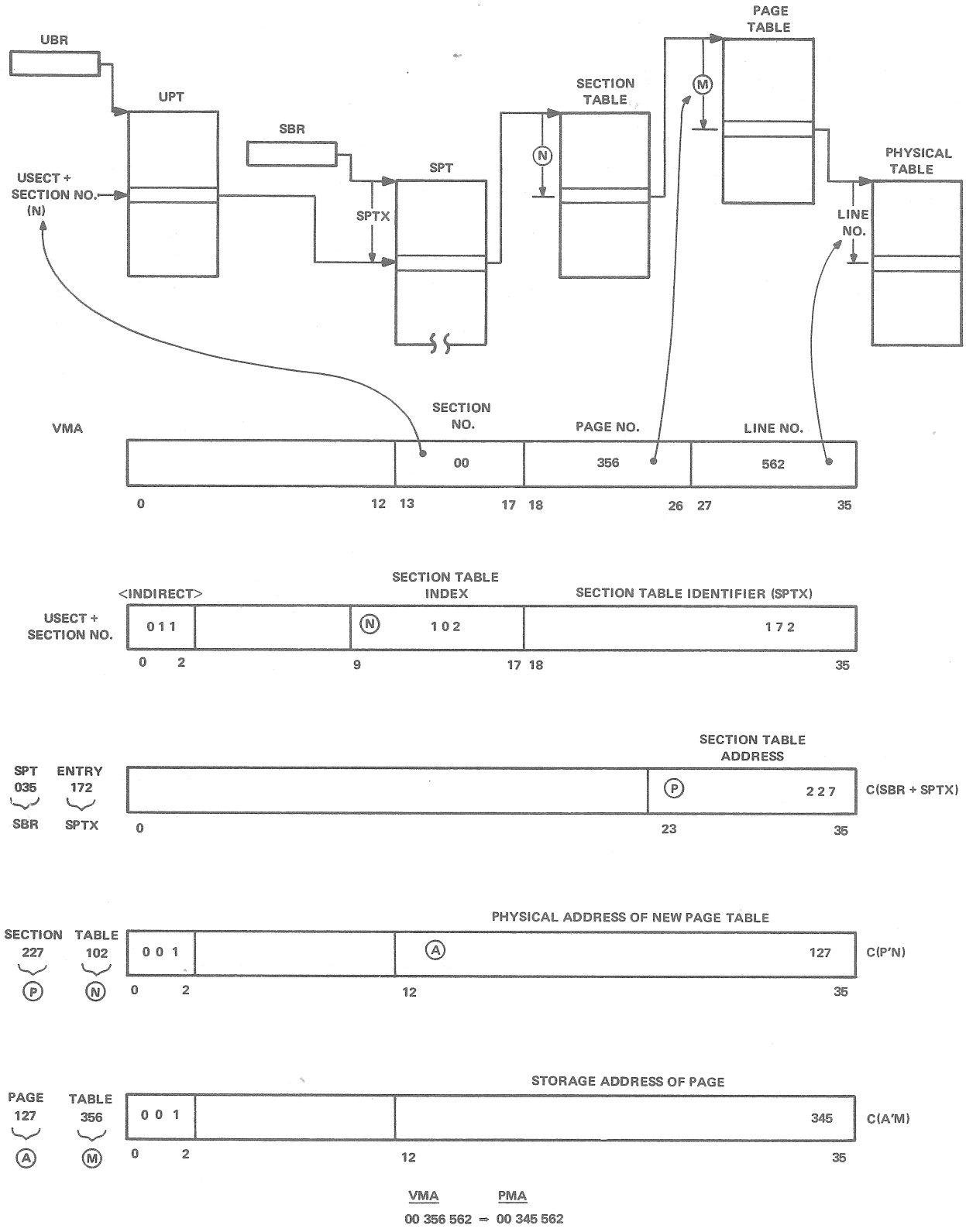


C (UBR' USECT + SECTION NO.) CONTAINS SPTX
C (SBR + SPTX) CONTAINS PAGE TABLE PAGE NO. N
C (N'M) CONTAINS STORAGE ADDRESS OF DESIRED PAGE

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)

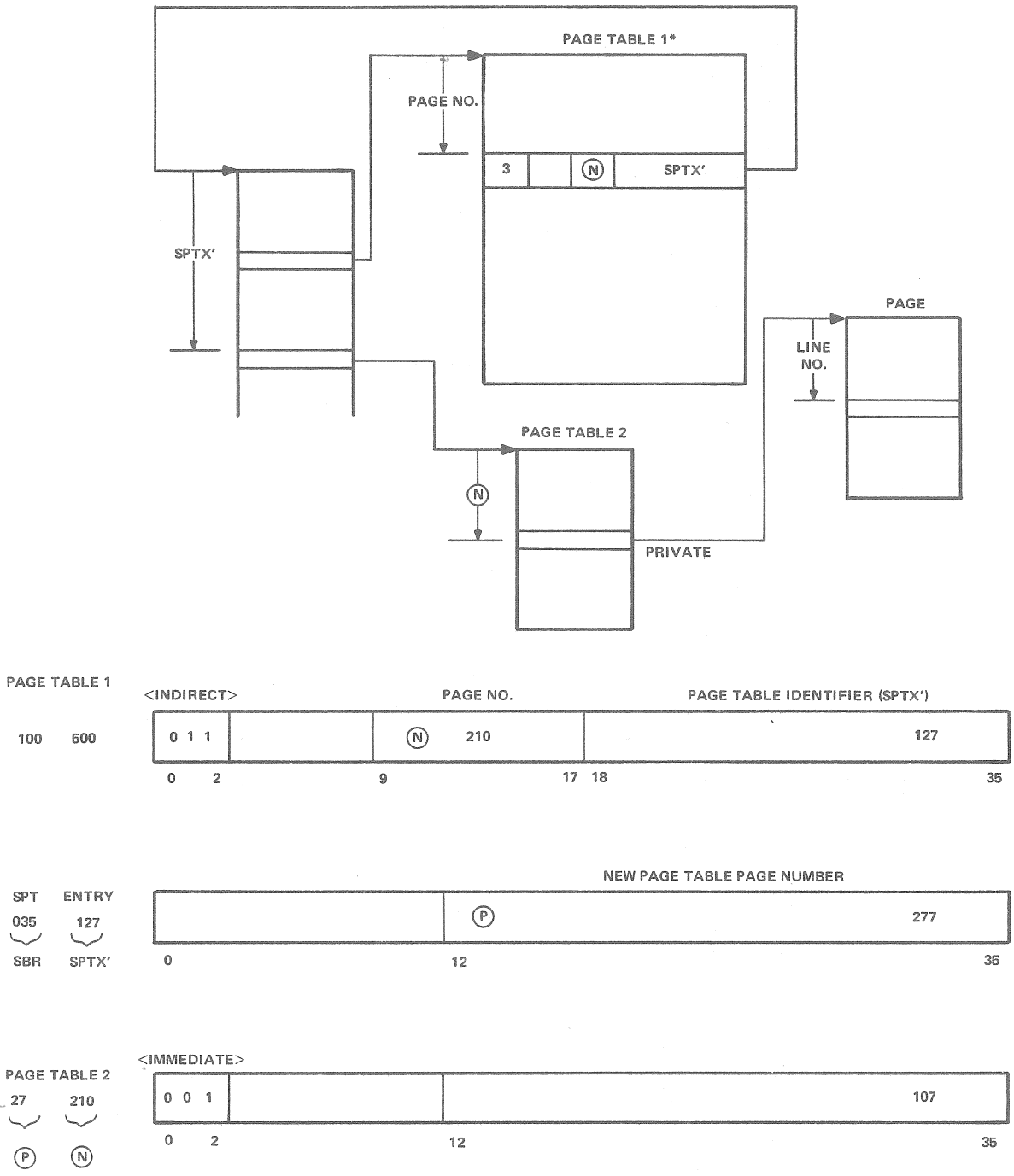


Figure 1-25 Pointer Interpretation (Indirect Page Pointer)

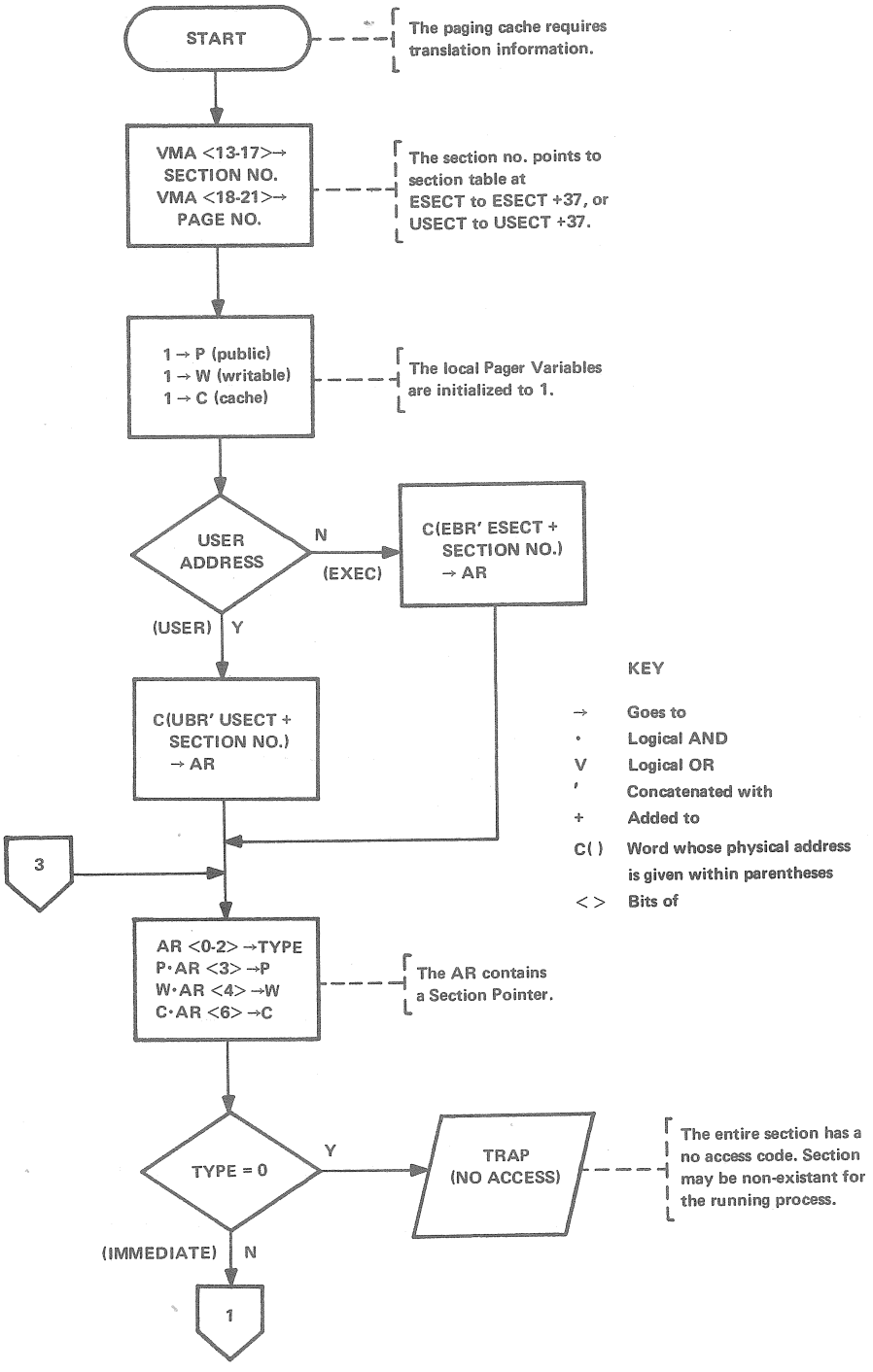


Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 1 of 5)

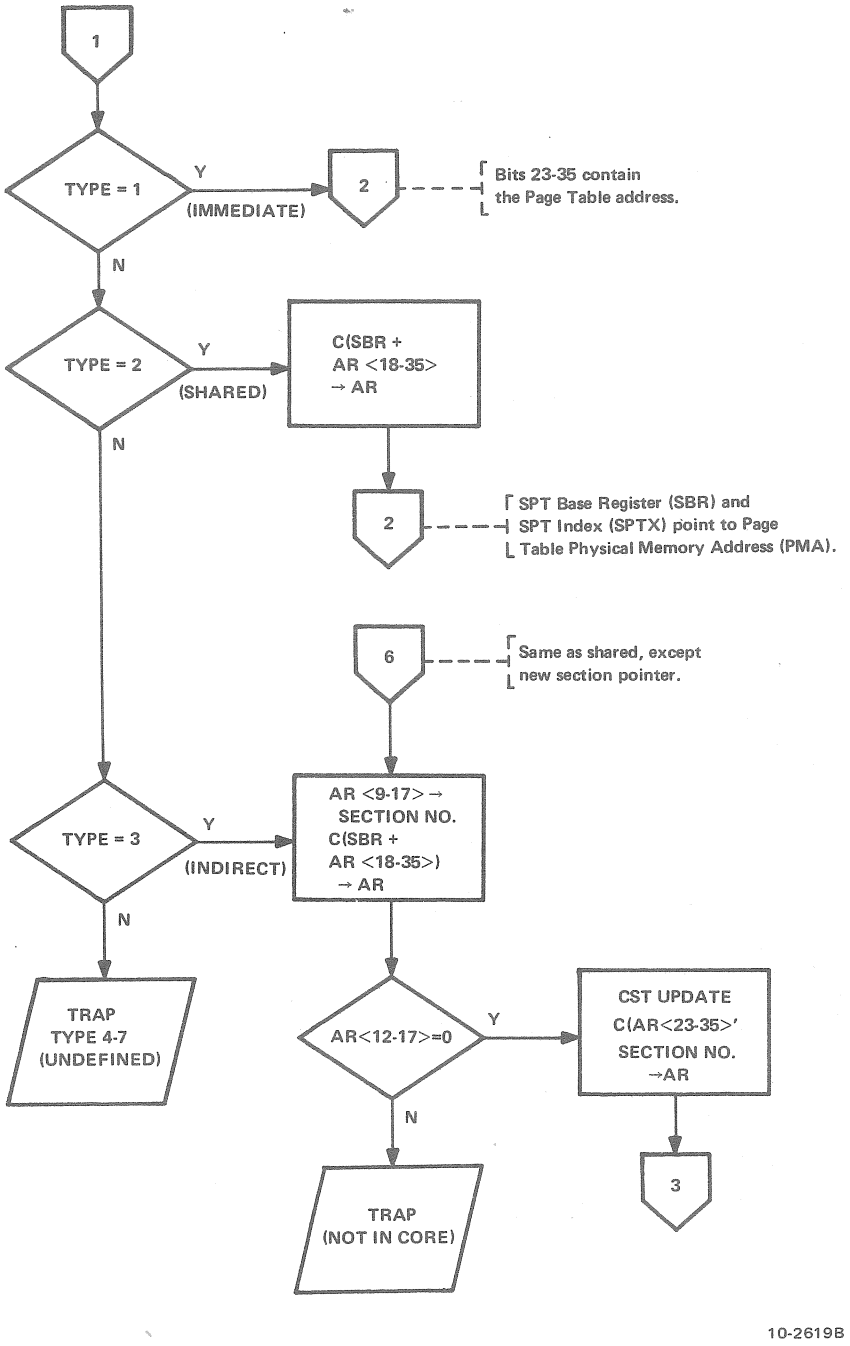


Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 2 of 5)

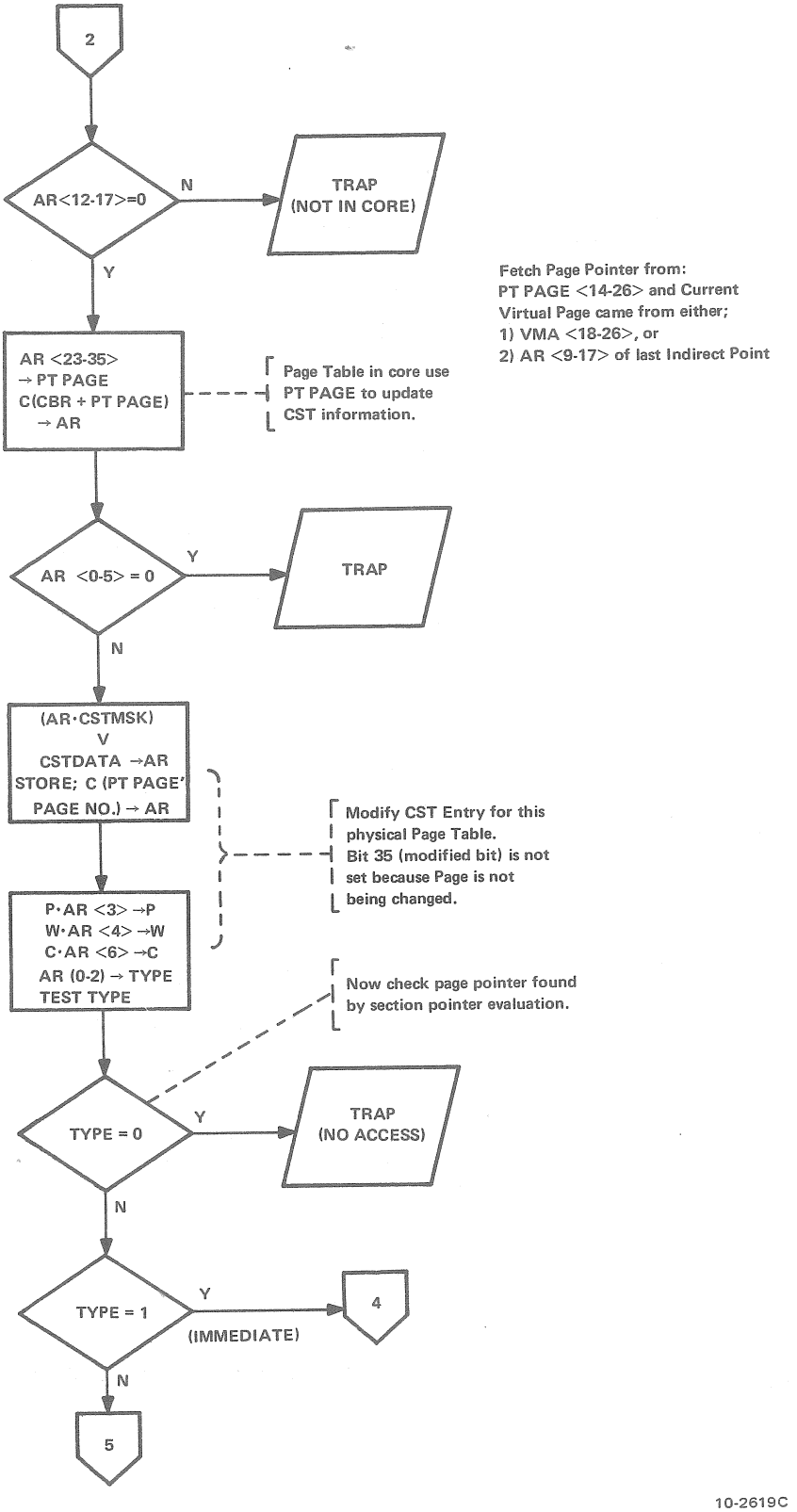


Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 3 of 5)

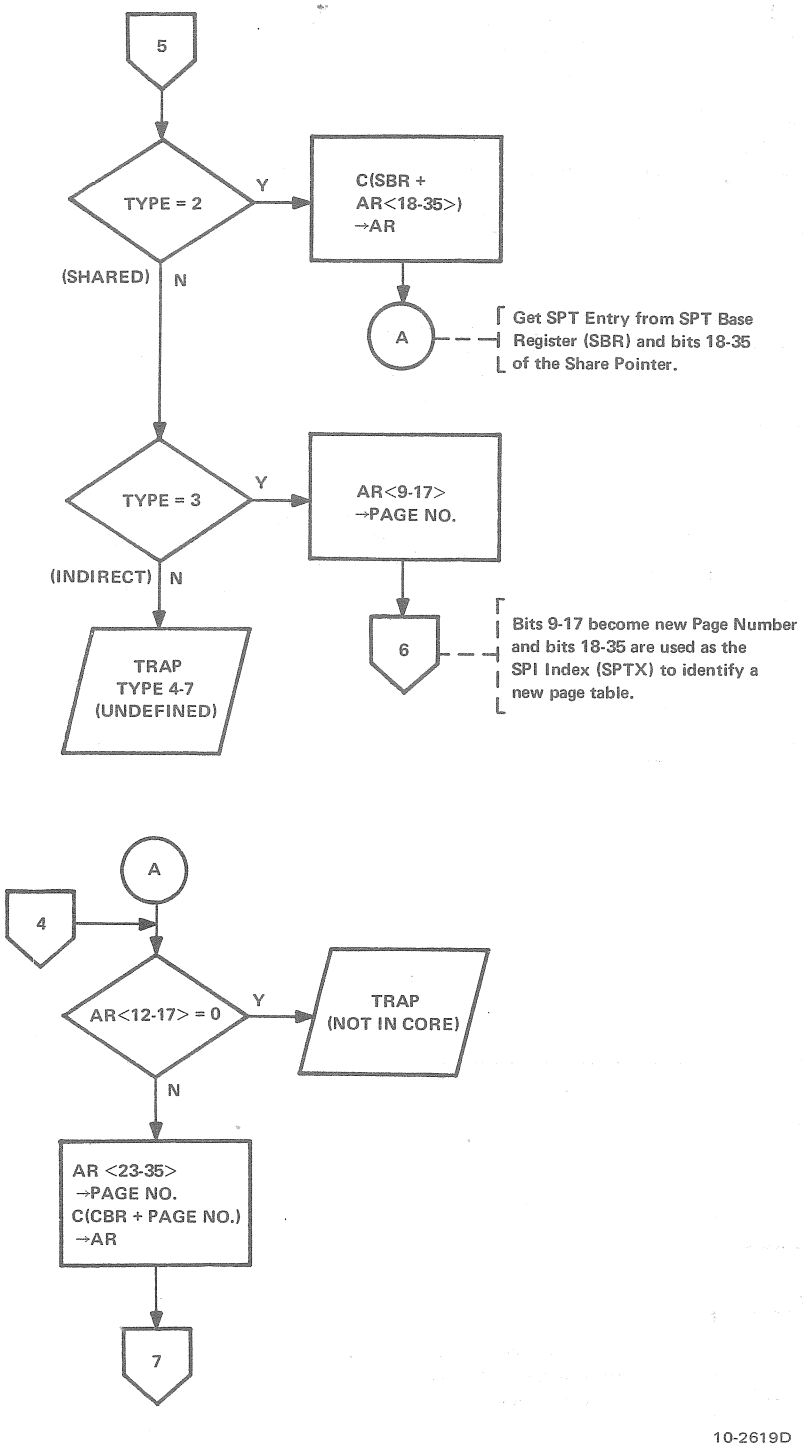


Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 4 of 5)

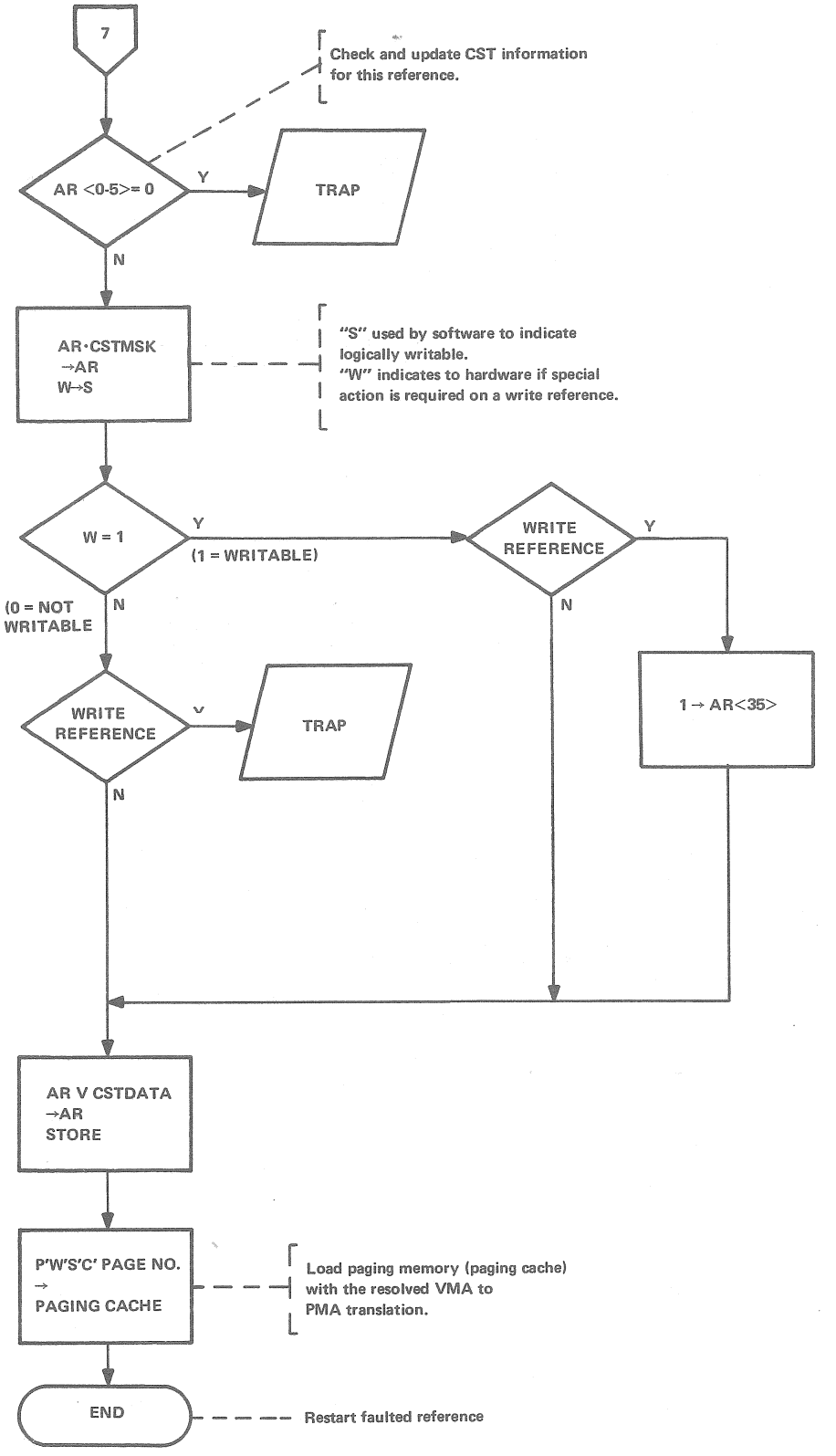


Figure 1-26 Pointer Interpretation Flow Diagram (Sheet 5 of 5)

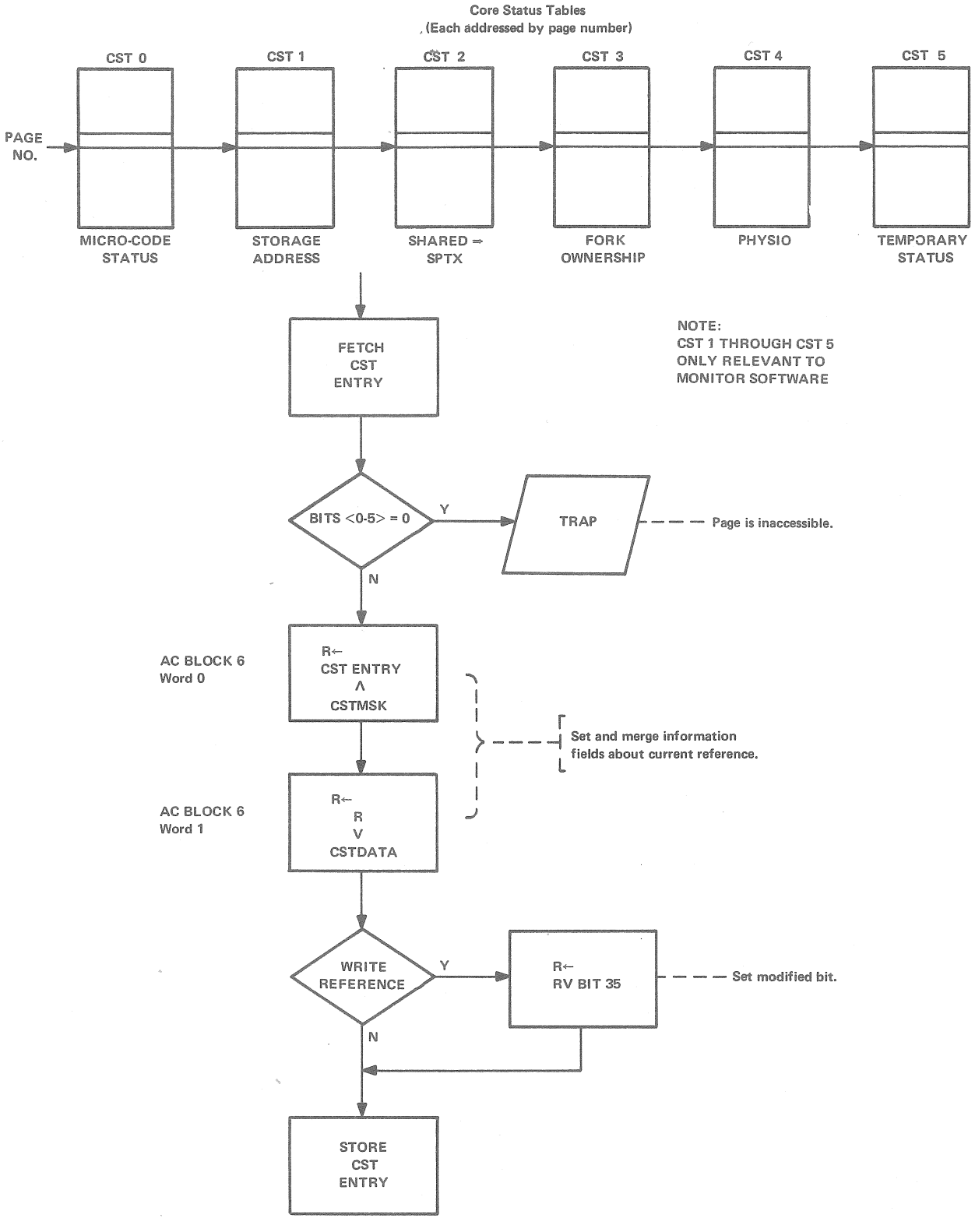


Figure 1-27 KL Core Status Tables Updating Flow Diagram

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.

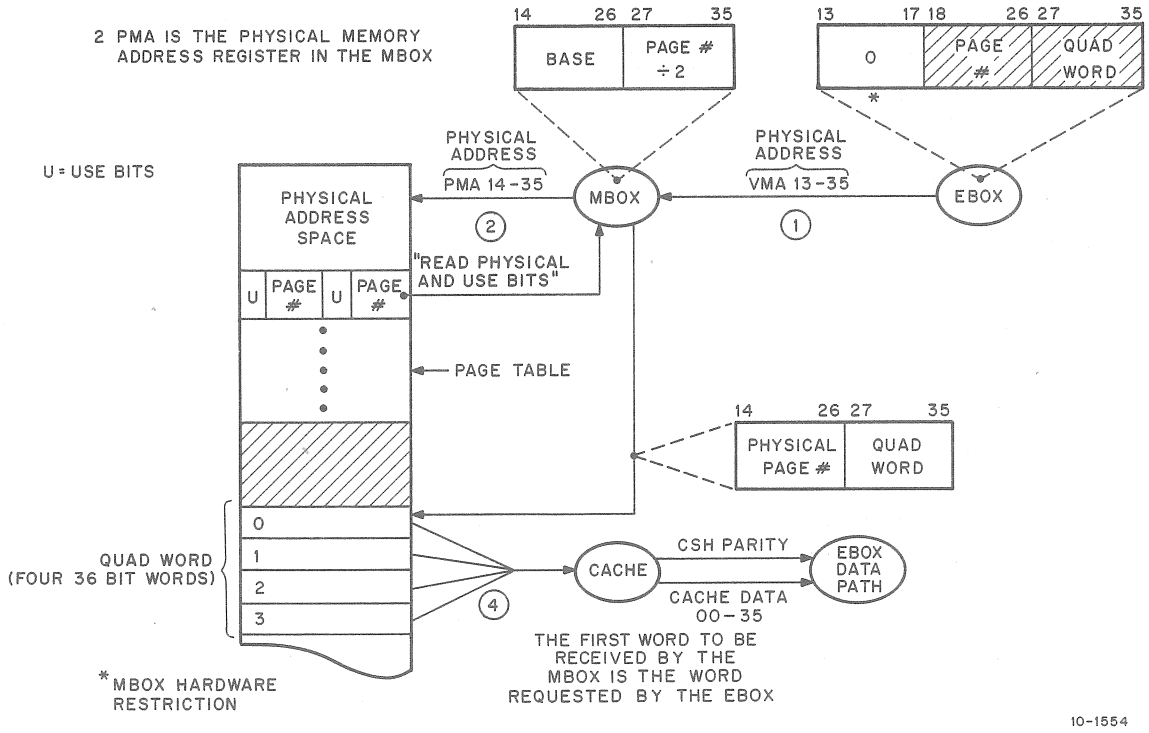
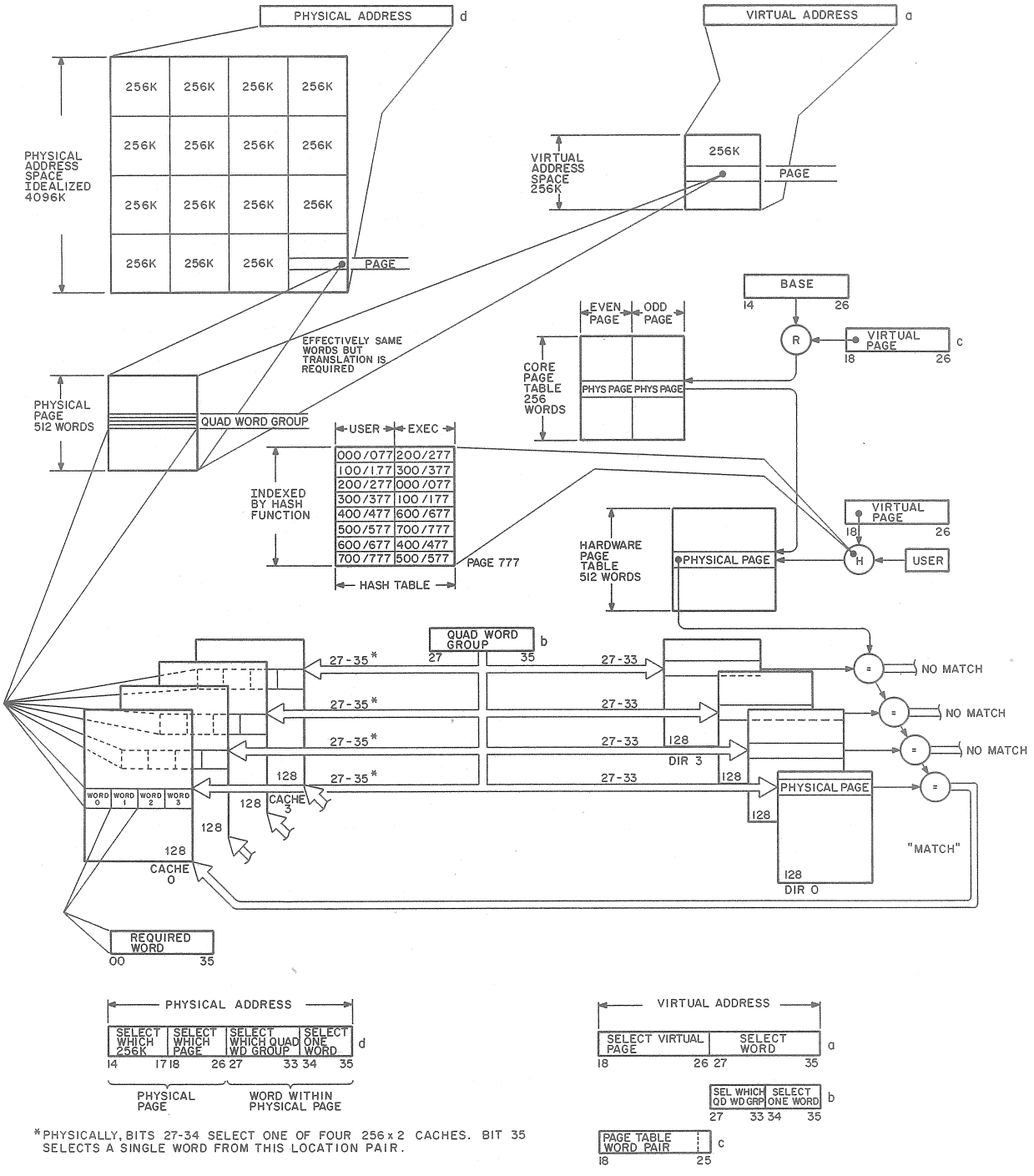


Figure 1-28 Basic Address Translation

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.



*PHYSICALLY, BITS 27-34 SELECT ONE OF FOUR 256x2 CACHES. BIT 35 SELECTS A SINGLE WORD FROM THIS LOCATION PAIR.

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.

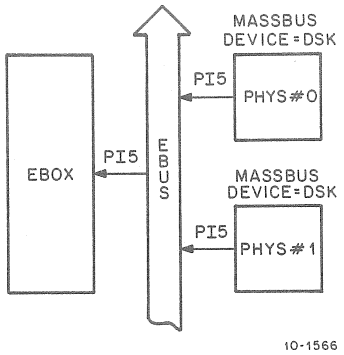


Figure 1-30 Simultaneous Interrupts

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.

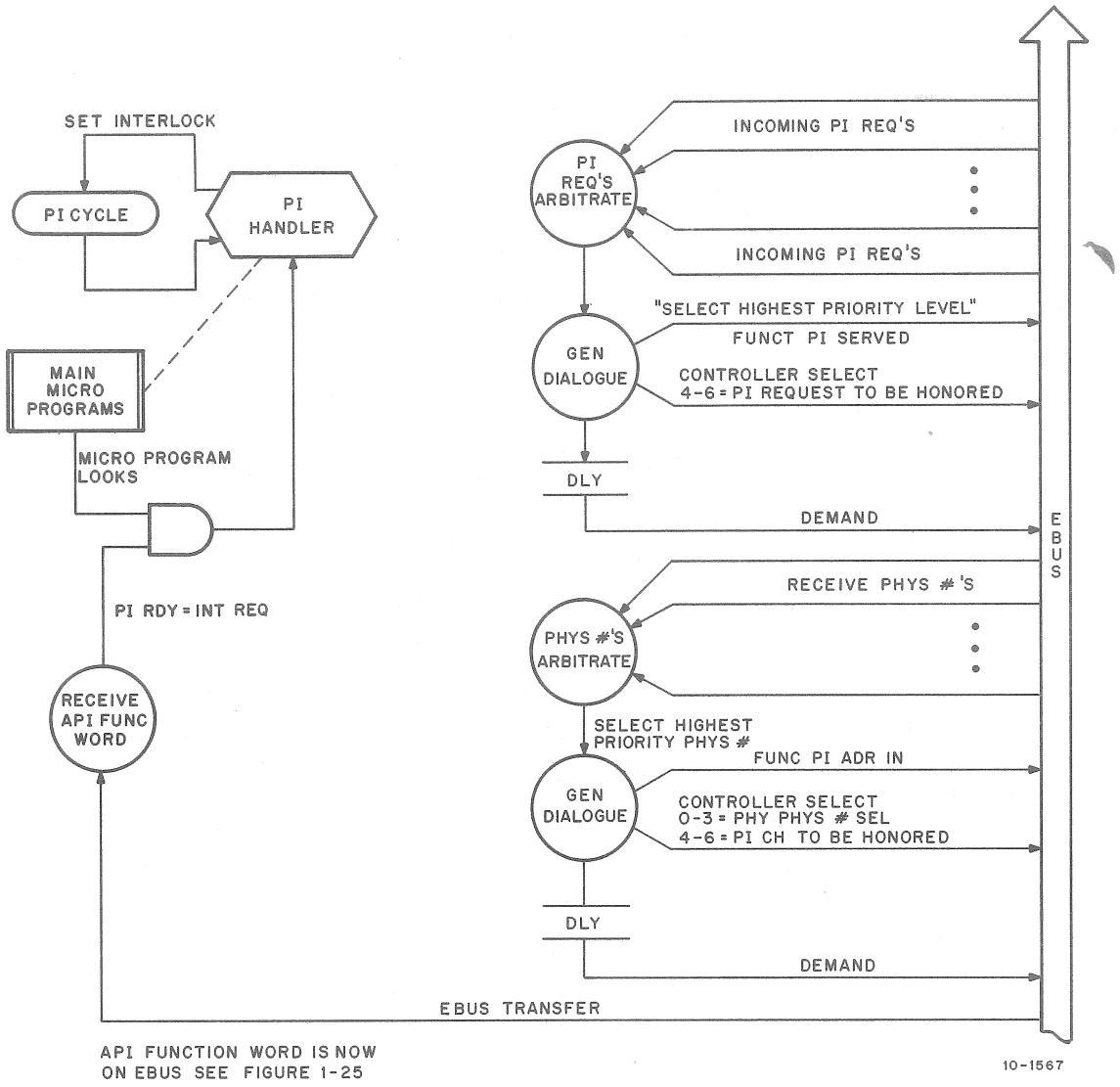


Figure 1-31 PI Dialogue Overview

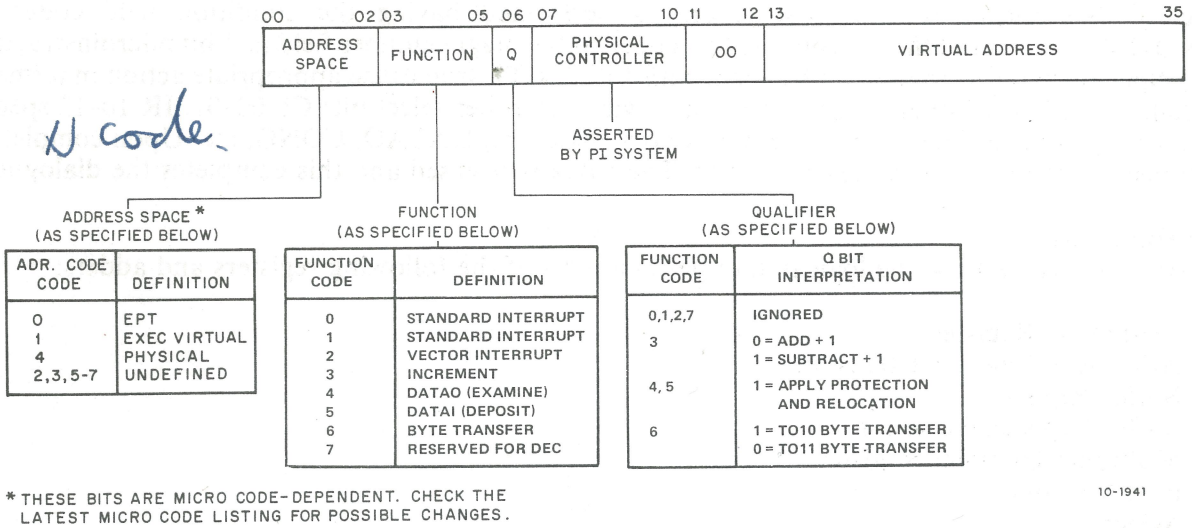


Figure 1-32 API Word Format

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.

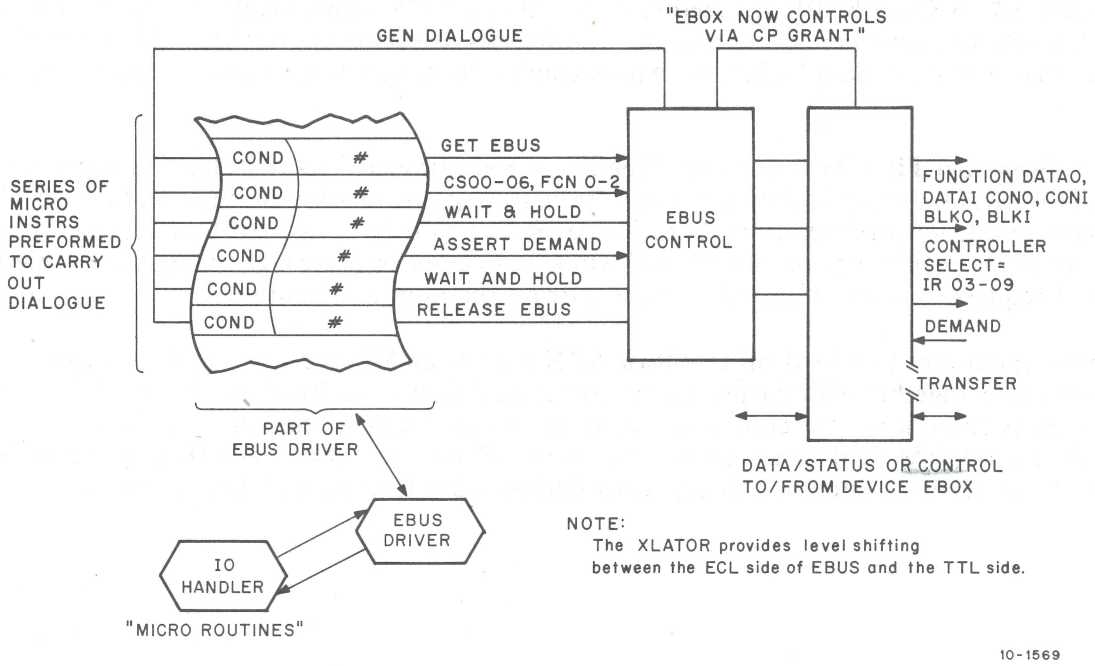


Figure 1-33 I/O Instruction Dialogue Overview

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.

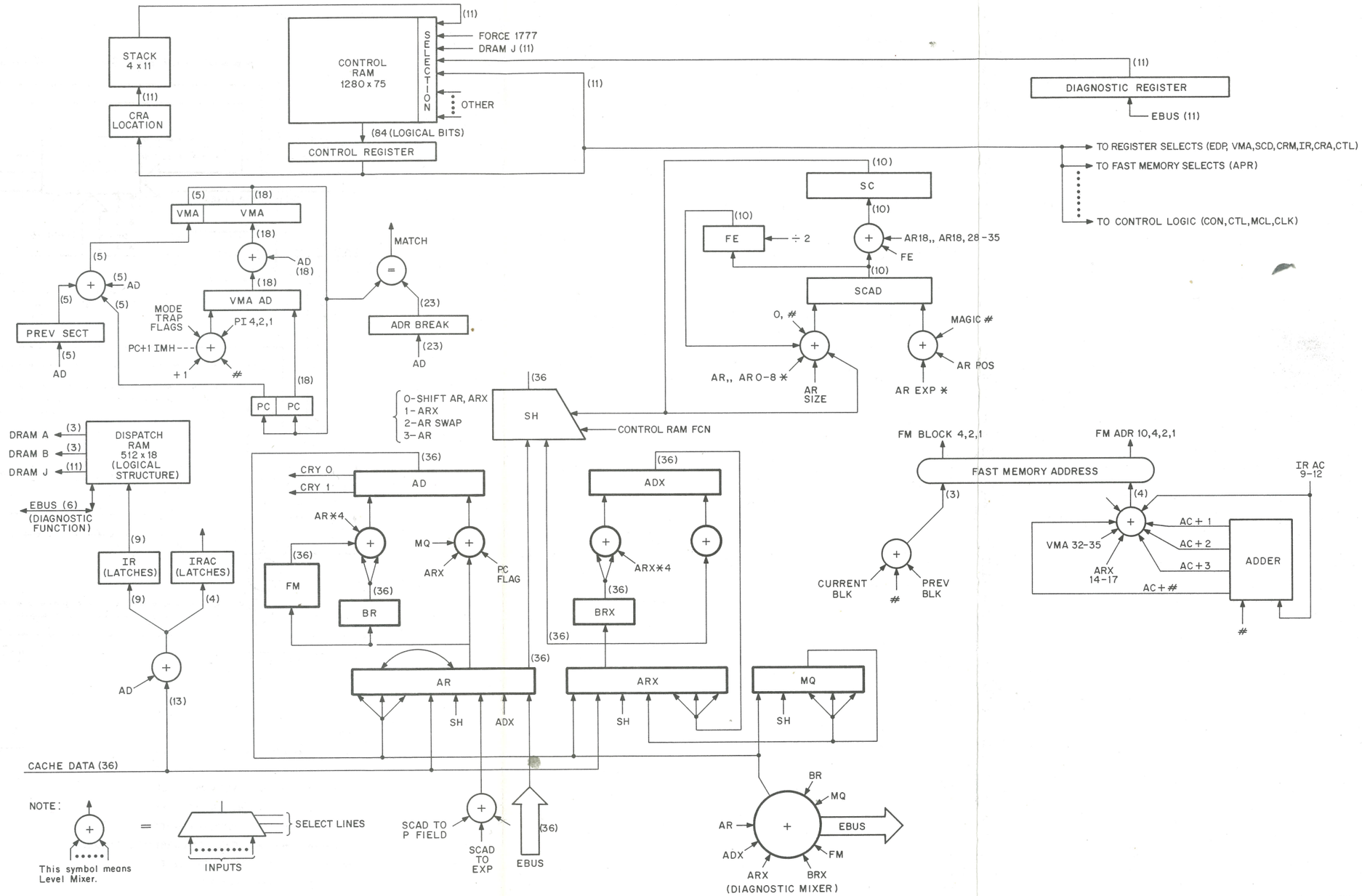
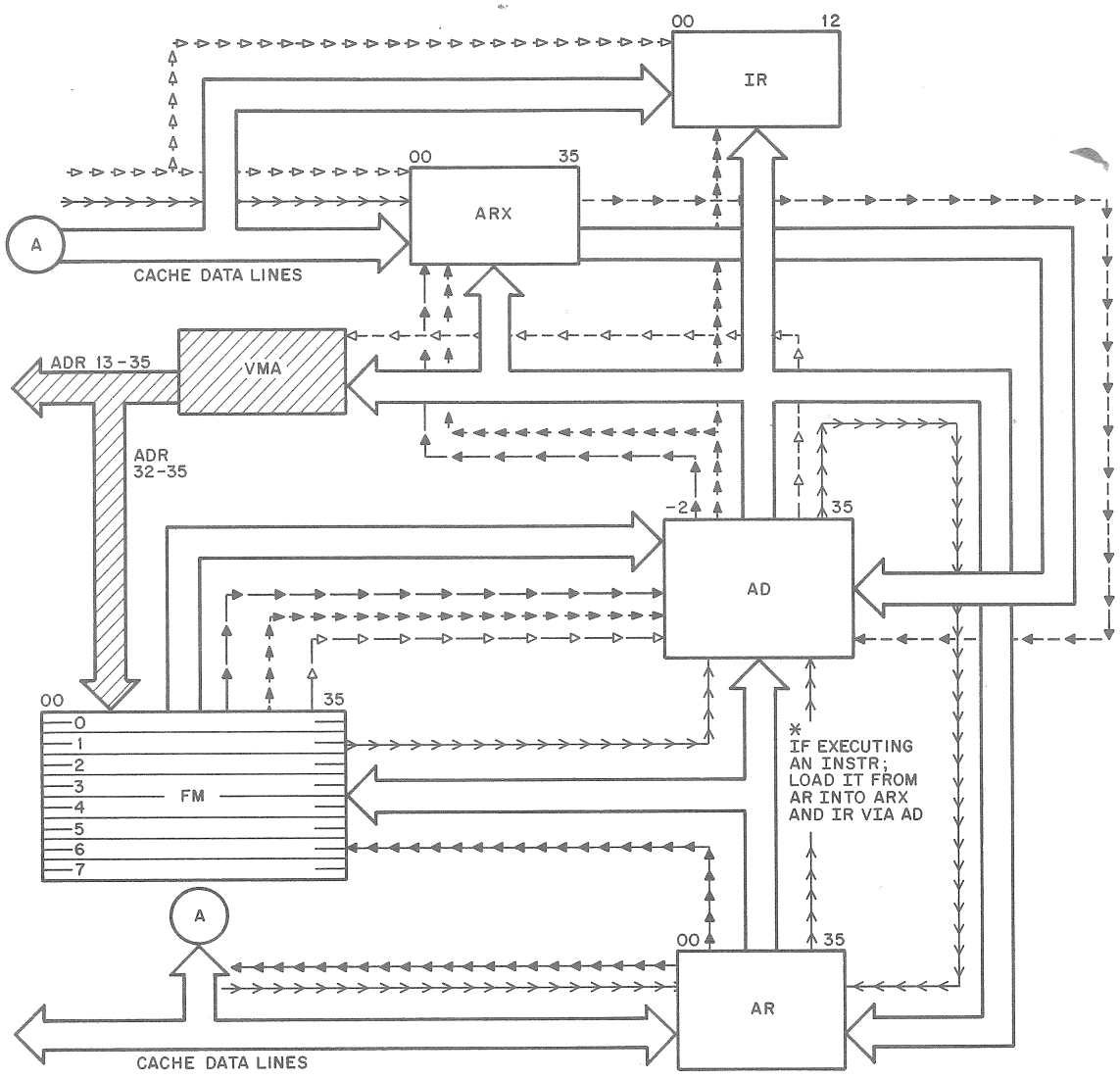


Figure 1-34 KL10 Register Interconnection Diagram



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.

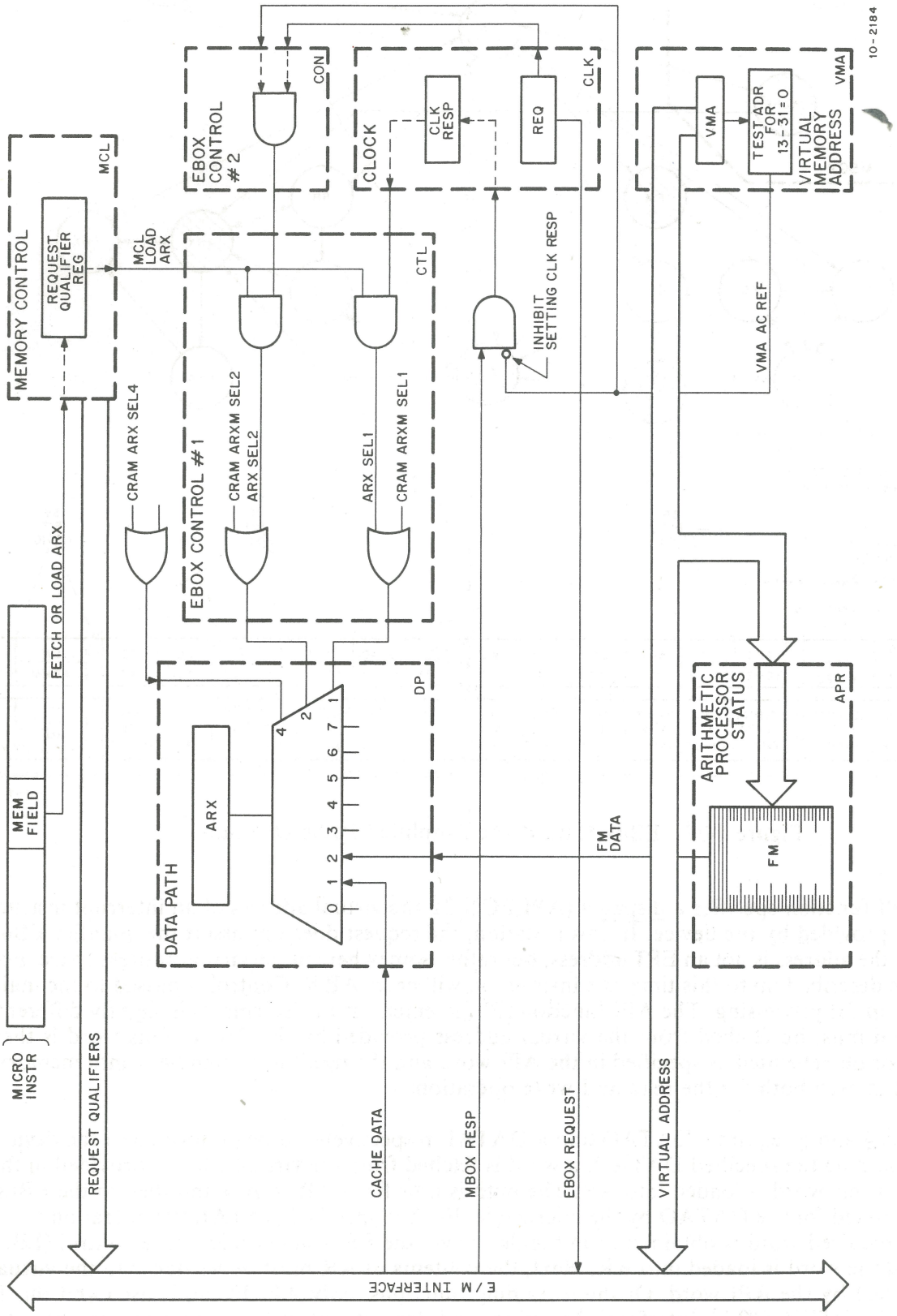


Figure 1-36 Loading ARX

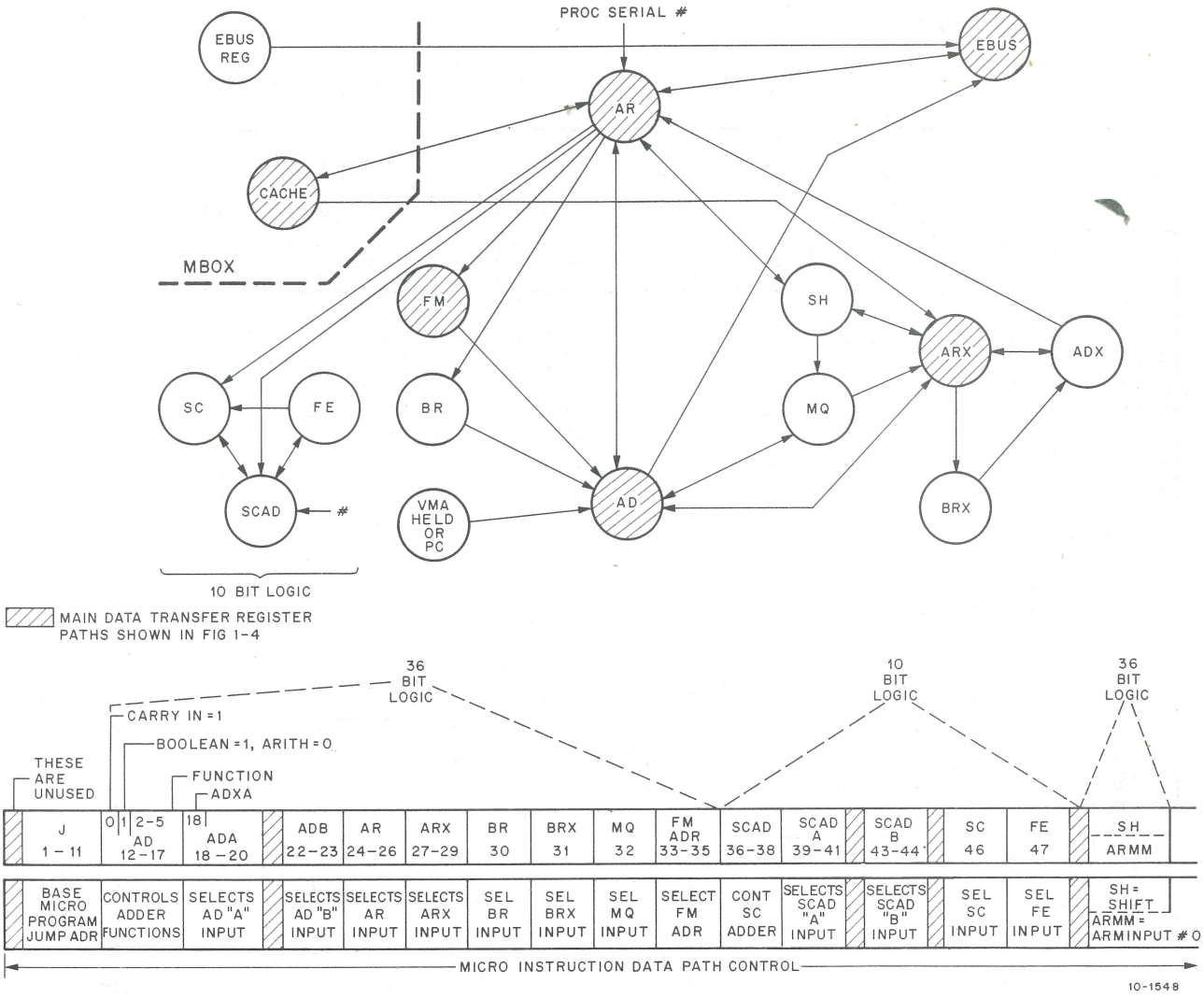


Figure 1-37 EBox Data Paths Simplified Paths Diagram

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.

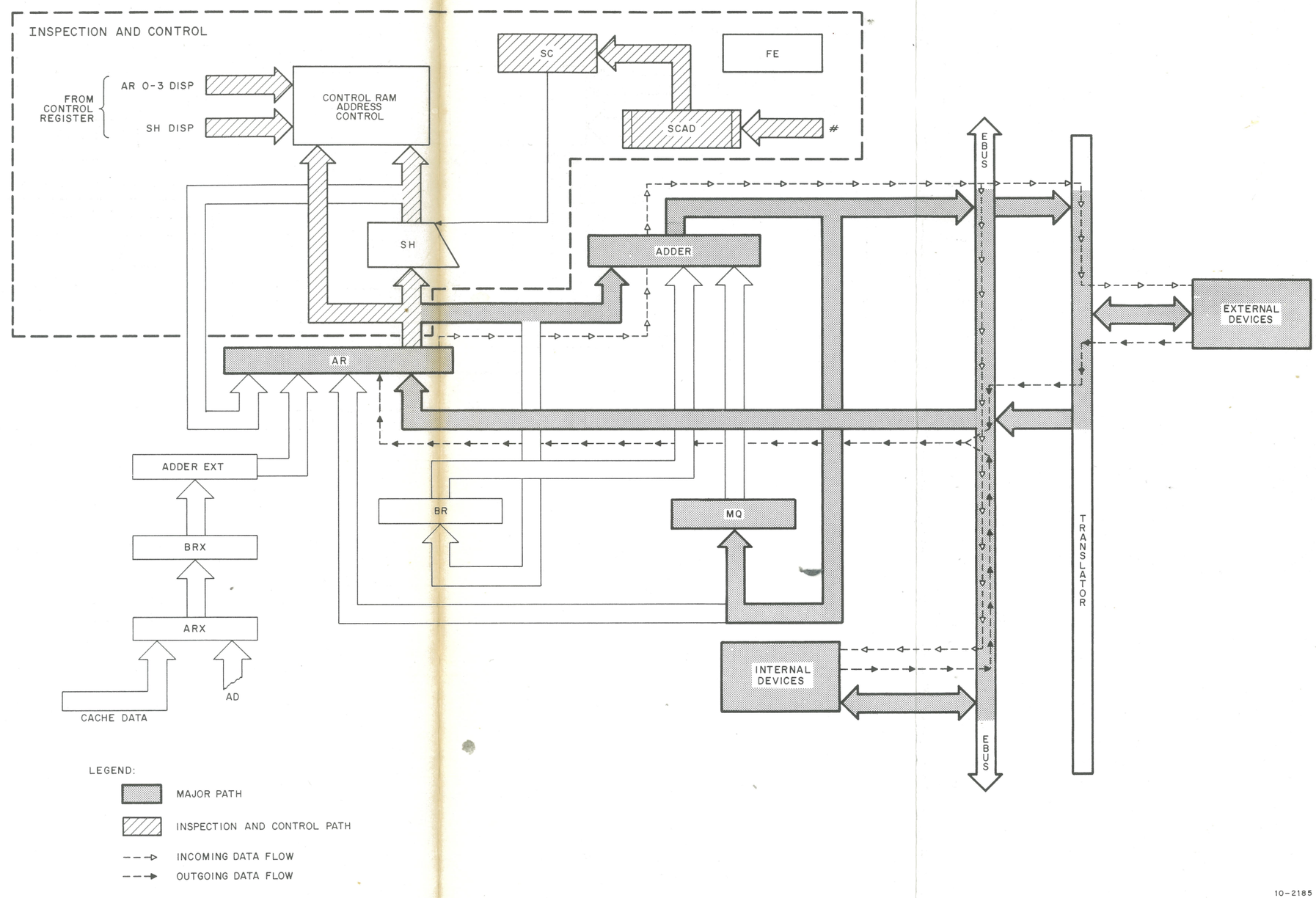


Figure 1-38 Input/Output Priority Interrupt Information Flow