Digital Equipment Corporation
Maynard, Massachusetts

digital

decsystem10

Maintenance Manual

KI10
CENTRAL PROCESSOR

A-MN-KI10-0-MAN1

decsystem10

KI10
CENTRAL PROCESSOR
MAINTENANCE MANUAL

1st Edition, October 1973

CONTENTS

Page

CHAPTER 1 INTRODUCTION

CHAPTER 2 LOGICAL ORGANIZATION

CHAPTER 3 OPERATION

CHAPTER 4 CONVENTIONS AND NOTATION

CHAPTER 5 CONTROL

CONTENTS (Cont)

CONTENTS (Cont)

ILLUSTRATIONS

FOREWORD

The DECsystem-10 is a general purpose, stored program computing system consisting of processors, memories, and input-output devices, each of which has independent internal timing. Every system must have at least one PDP-10 central processor but may contain several such processors, each of which is the control unit for a large-scale subsystem with its own memories and peripheral equipment. A system may also include direct-access processors, which provide for direct communication between memory and peripheral equipment. Individual processors in a system may share memories and in-out equipment, and different processors and memories may have different speeds and operating characteristics.

Extensive information on the overall system is given in Chapter 1 of the *DECsystem-10 System Reference Manual;* the reference manual also describes the operation and machine-level programming for the entire system. Maintenance documentation for the system is provided by a series of manuals. This manual discusses the logic and maintenance of the KI10 central processor and its basic in-out devices (reader, punch and teletypewriter). Other maintenance manuals cover the other types of processors, the several types of memories that may be used in a DECsystem-10, the various interfaces and control units for peripheral equipment, and the numerous in-out devices themselves. Drawings for any unit are available as the Customer Print Set. Information for connecting peripheral equipment of the customers own design is provided in the *DECsystem-10 Interface Manual,* which includes the necessary information on the circuits used in interfacing. Information on all other circuits is contained in the two-volume set of *DECsystem-10 Replacement Schematics.*

PREFACE

This manual is published to aid service personnel in the operation and maintenance of the KI10 central processor and the basic in-out devices associated with it. Maintenance information for the in-out equipment is confined primarily to those portions integrated into the processor logic; separate manuals for the devices themselves are furnished with the system.

The first three chapters present a general description of the system and its operation. Chapter 1 discusses physical and electrical characteristics. Chapter 2 treats the logical organization of the system in terms of registers and data flow; the treatment is at the intermediate block diagram level to serve as a bridge between the basic system information given in Chapter 1 of the reference manual and the detailed treatment of the hardware in later chapters of this manual. There is no information here on programming, the number system, or instruction formats (which are found in the reference manual), but the entire Appendix A of the reference manual, with its tables of instructions and device mnemonics and its complete list of instruction operations in symbolic form, is included in Appendix A of this manual. The appendix also contains diagrams of the DECsystem-10 ASCII code and the formats of the words used as instructions, numbers and pointers. Chapter 3 explains the operation of the processor and the basic IO devices, including all the information given on this subject in the reference manual. Although the chapter does contain some information of a maintenance nature, it is limited to a discussion of the controls and indicators that are visible to the operator. Photographs of the various panels are printed on foldouts in Appendix B.

The next six chapters present a complete, detailed description of the system logic. Chapter 4 describes the elements used in implementing the processor logic and discusses the basic engineering documentation, including the symbols and terminology used in the logic drawings and flow charts. The next three chapters describe the hardware for main control, including control registers, fast memory, processor cycles and the console, for interfacing with memory, and for logical and arithmetic processing. Chapter 8 explains the sequences of events, with reference to the flow charts, through which the processor performs all of the basic instructions. Chapter 9 covers input-output, including IO instructions, the IO bus, priority interrupt, and the interfaces for the basic IO equipment. This chapter also describes the readin function, which makes significant use of elements in all parts of the processor, especially in-out logic and console. The reader is strongly advised not to embark upon any logic chapter in this or any other DECsystem-10 maintenance manual without first gaining a thorough understanding of the material presented in Chapter 4.

Chapter 10 contains information useful in maintaining the system, including a description of the engineering drawings, maintenance operation, maintenance programming, adjustments, and a list of diagnostics.

All engineering drawings referred to in the text may be found in the front two-thirds of the *KI10 Customer Print Set.* Documents of particular use to the reader are the following.

DECsystem-10 System Reference Manual	DEC-10-HGAD-D
DECsystem-10 Site Preparation Guide	DEC-10-SITE-D
DECsystem-10 Layout Kit	DEC-10-KITB-D
DECsystem-10 Interface Manual	DEC-10-HIFC-D
KI10 Customer Print Set	B-DD-KI10-0
DECsystem-10 Replacement Schematics	B-MN-PDP10-0-MOD1 B-MN-PDP10-0-MOD2

CHAPTER 1
INTRODUCTION

Before reading this manual, service personnel should be familiar with the organization and function of the KI10 central processor and the DECsystem-10 as a whole to the extent covered in the System Reference Manual. It is unnecessary at first to be familiar with the details of every instruction, but begin by reading thoroughly the following parts of the reference manual: all of Chapter 1, the text portions (consisting mostly of introductory remarks to the instruction groups) in sections 2.1 to 2.10 and 2.14, and all of sections 2.12, 2.13 and 2.15 treating input-output, priority interrupt, and machine modes including paging. Effective maintenance however requires adeptness at programming, so in the long run one should be thoroughly familiar with the entire contents of the first three chapters in the reference manual (the last two pages of section 2.14 and all of sections 2.16 and 2.17 can be skipped, as they apply only to the KA10).

1.1 PHYSICAL CHARACTERISTICS

Most DECsystem-10 equipment is housed in steel cabinets or bays, each of which has an indicator panel at the top. The KI10 has three such bays numbered from left to right (Figure 1-1). Complete physical dimensions, clearance requirements, and the like are listed in the Site Preparation Guide. At the center of bay 3 (the console bay) are the console operator panel and a small maintenance panel. Behind the door below the console shelf is a vertical panel for connections to the console teletypewriter. A DK10 real time clock is ordinarily mounted in this space as well. Above the maintenance panel are the paper tape reader and punch. The space between these and the indicator panel at the top is often used for a DECtape transport. The vacant panel at the left of the reader is sometimes used for a telephone.

Behind the doors on the front of bays 1 and 2 is the module wiring, which is on mounting panels into which modules are inserted from the rear. Each bay has sixteen horizontal rows lettered A to T from the top (skipping G, I, O and Q), and each row has forty-four module connectors or slots numbered left to right. An individual row is identified by the bay number followed by the row letter, but for checking margins, rows 1A–1T are addressed as 00–17 octal and rows 2A–2T are addressed as 20–37. Both designations are printed at the outer ends of the rows (the left end in bay 1, the right end in bay 2). An individual slot is identified by the row designation followed by the slot number. Each slot has two columns of eighteen wire-wrap pins protruding through the panel from the module connector pins on the rear. The pins are lettered A to V in pairs from the top (skipping G, I, O and Q) and an individual pin is identified by the pin letter and column number (1 left, 2 right) appended to the row designation. At the outside end of each pair of rows is a small panel containing circuit breakers and margin check switches. At the bottom of bay 2 are the sockets for the memory and IO buses and two half rows of logic modules designated 3A and 3B.

Inside the doors at the rear of the bays are inner mounting doors, which are used for mounting the power equipment. At the bottom of the bay 3 mounting door are the 857 and 858 power controls (Figure 1-2), but the main 845 power control is mounted at the bottom on the side of the bay (against the end panel). The three power supplies at the top of the bay door are for margin checking only; dc voltages for the logic are supplied by the remaining power supplies in the middle of the bay 3 door and on the mounting doors on bays 1 and 2.

Looking into bay 1 or bay 2 from the rear (Figure 1-3) one can see yet another door, the cooling assembly door, which completely encloses the modules mounted at the front of the bay. Each cooling assembly contains six thermistors, and at the top and bottom are small fans that cool the logic modules by drawing air down through them.

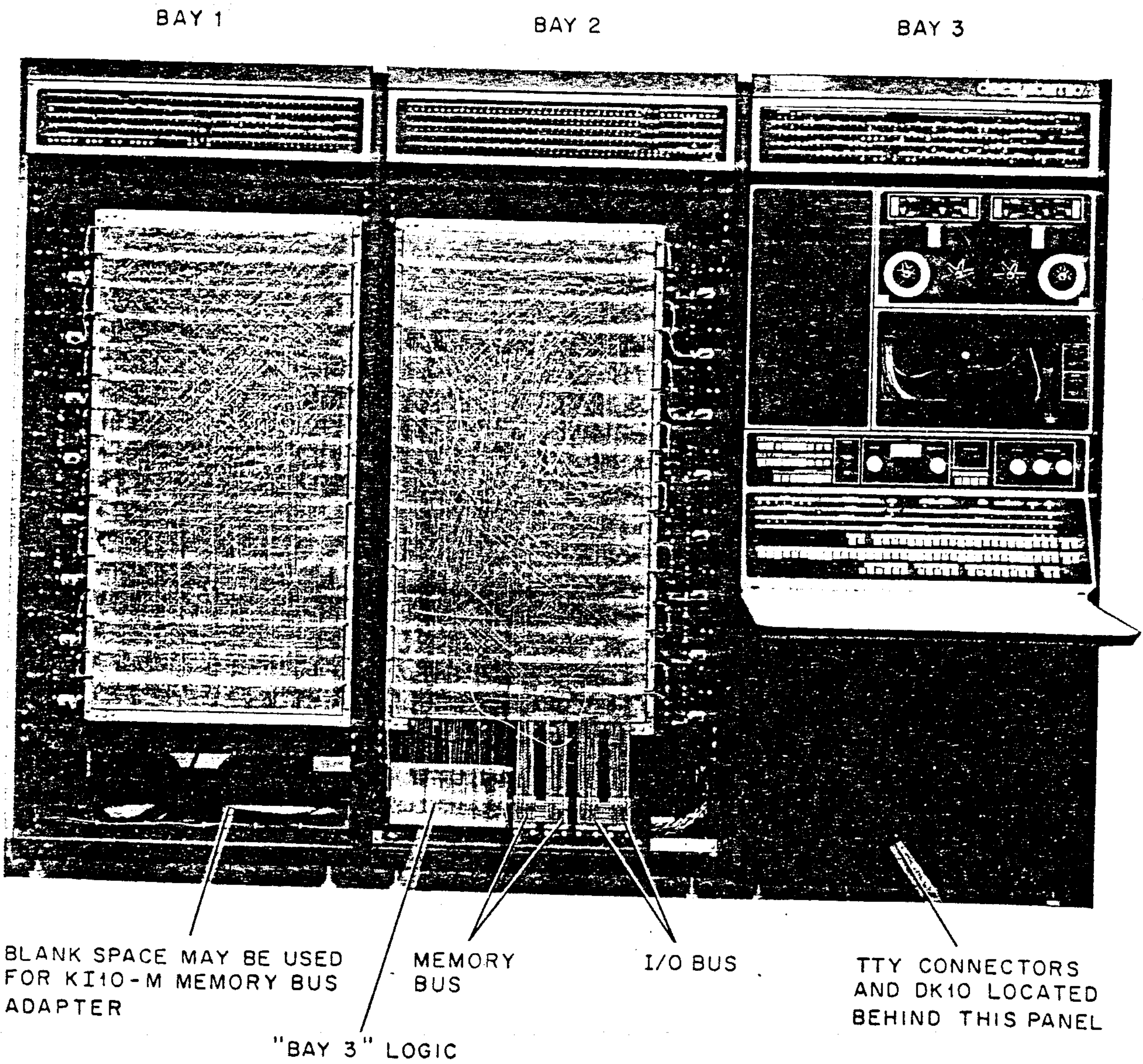


Figure 1-1 KI10 Front View

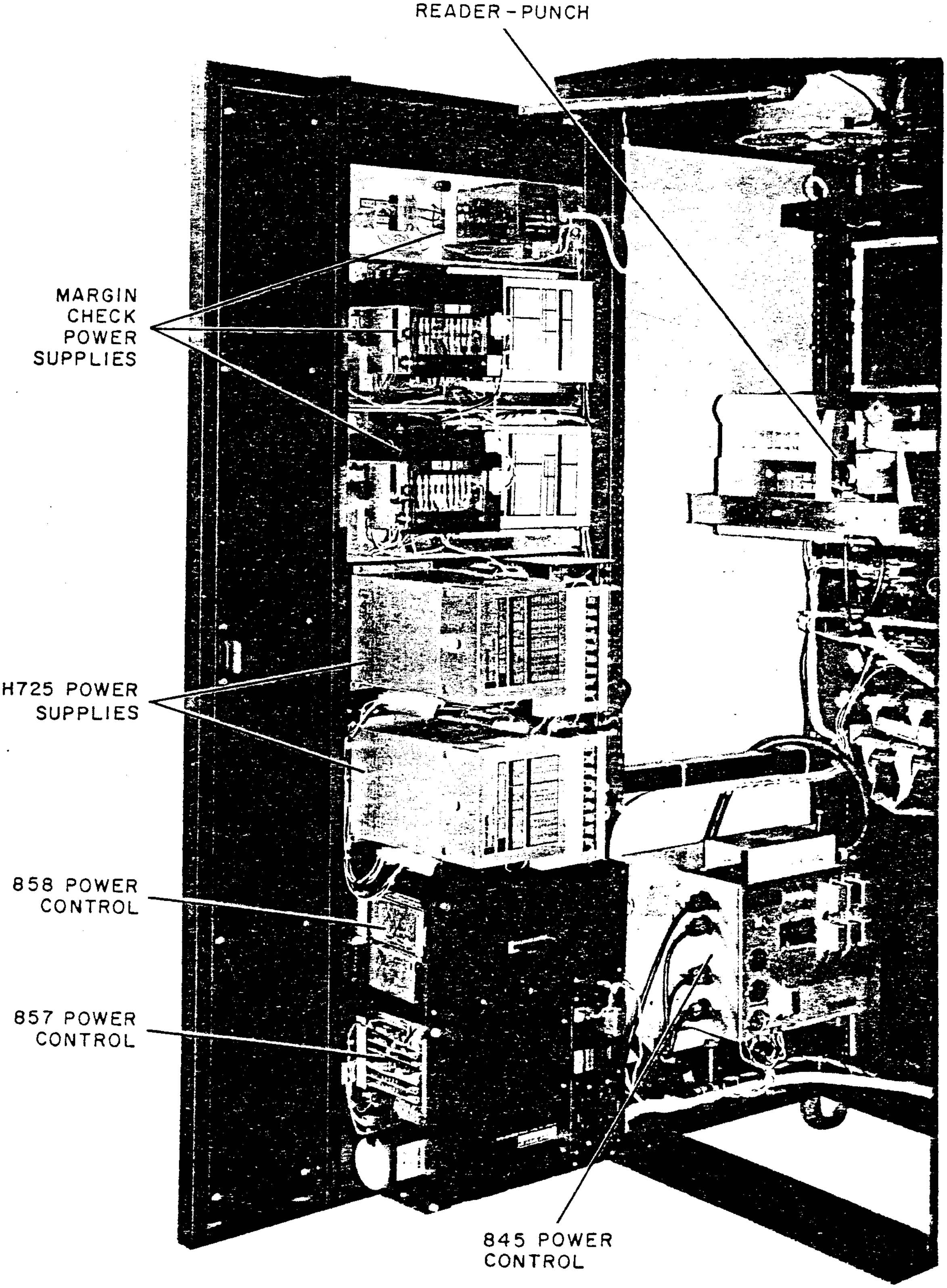


Figure 1-2 KI10 Rear View, Bay 3

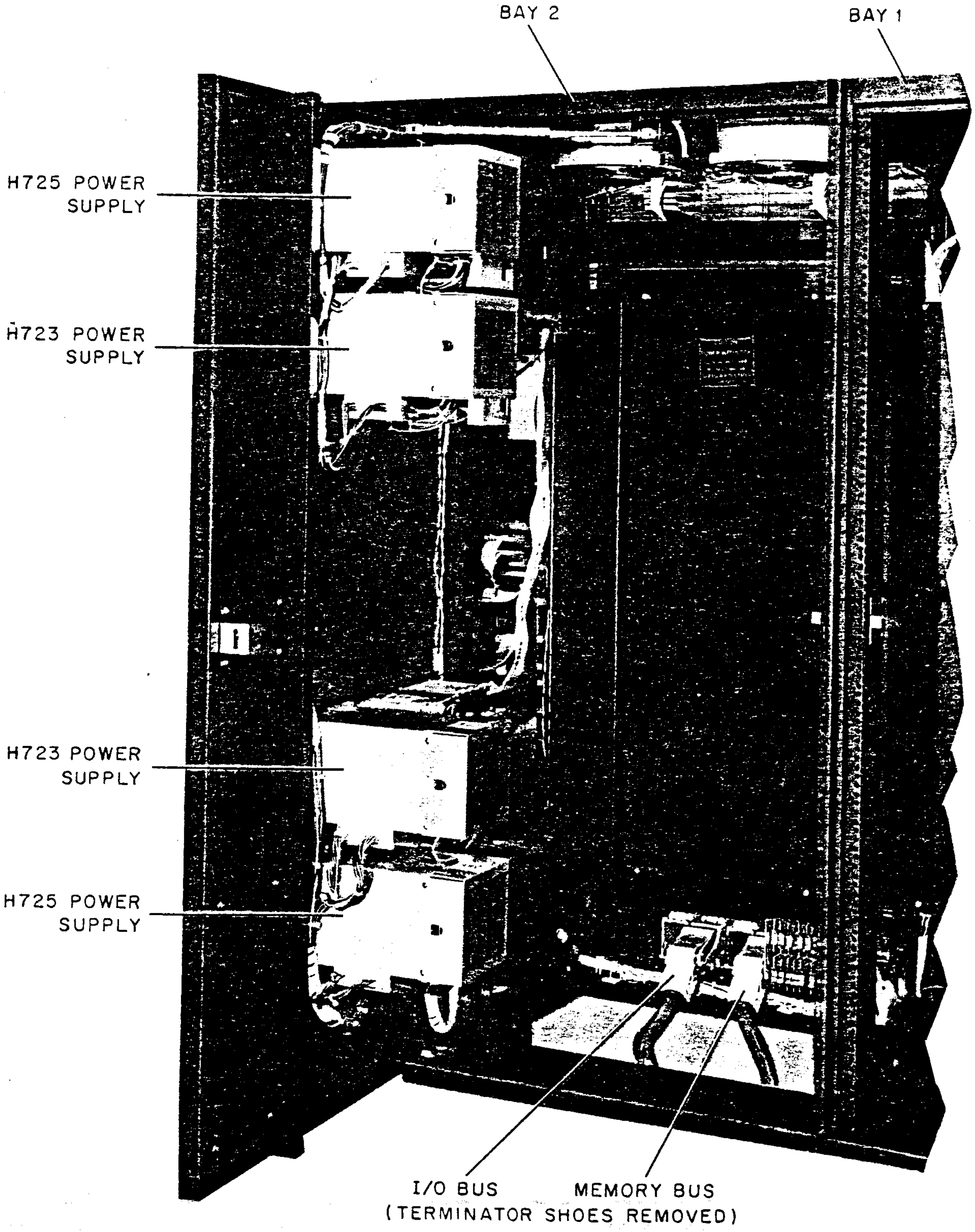


Figure 1-3 KI10 Rear View, Bay 2

The processor is connected to the other units in its subsystem (memories, direct-access processors, and peripheral equipment) by five buses, the IO bus, memory bus, DEC standard power control bus, margin check bus, and +5 margin bus. Each bus actually has two cables, right and left, where the right cable is for equipment at the right of the processor, and the left cable is for equipment at the left of the processor. As mentioned above, the memory and IO buses originate at the bottom of bay 2. (The basic IO devices and the parts of the processor that are treated like IO are connected directly into the processor logic without using the IO bus cables at all.) Both cables in the power control bus originate at the 857 power control; the cables in the +5 margin bus originate at the console maintenance panel, as does the right margin check bus. But for margin checking (other than +5 V dc), the first unit at the left is the processor itself, and the left margin check cable originates at a distribution point at the left end of the processor frame (at the bay 1 end panel).

1.2 ELECTRICAL CHARACTERISTICS

Complete information on line voltage, ac power, and types of external wiring and receptacles is given in the Site Preparation Guide. The ac input to the processor is at the 845 power control at the bottom of the console bay. This control has the main circuit breakers and supplies switched ac through power cables to the several bays; it also has convenience outlets and other switched and unswitched outputs for various uses, including connections to the other power controls and a 6.3 V ac signal to the logic for the line frequency clock. Of the two power controls at the bottom of the bay 3 mounting door, the 857 (the lower one) provides timing for power on and off, controls the failure lights on the maintenance panel, has sensors for the six thermistors in bay 1, and has an override switch; the 858 contains the restart logic, the overvoltage detector, and sensors for the six thermistors in bay 2.

The H725 power supply provides a floating 15 volts; those at the bottom of the mounting doors on bays 1 and 2 are connected for -15 volts, the remaining ones at the tops of those doors and in the middle of the bay 3 door are connected for +15 volts. The H723 units in bays 1 and 2 supply +8 volts. The dc voltages required by the logic are +5 and -15 volts. The +5 volts for each pair of logic rows is provided by a pair of regulators (series pass elements) in the small panel at the outside end of the rows; these regulators use +15, +8 and -15 volts. The +15 volts is also used by the lights and the EIA teletypewriter.

The KI10 logic is special TTL circuitry with high noise immunity. The low and high logic levels are 0 and +3 volts dc with tolerances of 0 to +.4 volt and +2.6 to +5 volts. Voltage levels may go outside these limits during transient conditions, but must be within the limits in the steady state. Any gate is guaranteed to hold the appropriate output when a low gate input goes as high as 1.0 volt or a high input goes as low as 2.0 volts. Pulses from pulse amplifiers in the logic are 70 ns nominal width but some are adjustable; specifications are the same as for levels, *ie* a pulse is simply a very short level. Rising edges are used for all edge-triggered circuits, such as the clock inputs of the D-type flipflops that are used extensively throughout the logic. Levels used on the IO and memory buses and in a very small portion of the punch logic are nominally -3 and 0 volts; memory bus pulses are 70 to 100 ns width. The logic symbology used in the drawings is essentially that of MIL-STD-806B.

The reader and punch are wired into the tap on the primary of the upper H725 power supply in bay 3 so that they operate on 115 volts regardless of the line voltage at the site. Power and signal connections to the console teletypewriter are through the panel under the console shelf (this panel has one switched and two unswitched convenience outlets for terminals, scopes, etc). All external units must have their own line power sources, but all can be controlled from the processor console. Memories and peripheral equipment designed specifically for the KI10 (such as the MF10 memory) are placed in operation by a ground remote turnon signal on the power control bus; this bus also has an emergency shutdown signal that turns off all equipment regardless of the state of the turnon signal and even if the power control in an external unit is in local mode. Older memories designed for use with the PDP-6 or KA10 (161, 163, 164, MA10, MB10, MD10, ME10) are controlled by a -15 volt turnon signal on the margin check bus. Turnon relays in older peripheral equipment may be controlled via the margin check bus or even by switched ac voltage from the 845 power control, although the latter cannot be used as a power source.

CHAPTER 2
LOGICAL ORGANIZATION

Logically the processor can be viewed as comprising four areas: basic control logic, memory logic, arithmetic logic, and input-output, although there is some crossover among them. The detailed hardware description of these areas is presented respectively in Chapters 5, 6, 7 and 9 (there is also some special maintenance logic discussed in Chapter 10). In some cases the material presented in the reference manual serves as an adequate introduction to the detailed treatment of the hardware, especially for an IO interface, where an understanding of the machine-level programming usually requires a greater understanding of the hardware that implements it. The block diagram on page 1-3 of the reference manual does show the registers that are of significance to the programmer, but a very large amount of detail is hidden not only in the box labeled "arithmetic logic," but in the boxes representing the registers as well. A much more detailed diagram of the processor is available in the Customer Print Set as drawing FD-KI10-0-REG. Here we see all of the processor registers, the data paths connecting them, and much of their associated gating. A circle containing a plus sign indicates a mixer through which information can pass from one of a number of different sources; numbers in parentheses by the data paths indicate the number of bits. The left section of the drawing shows the control and memory areas combined, the small section at the lower right shows the fast memory selection, and the rest of the drawing is devoted to the arithmetic logic. The in-out logic appears only to the extent of connections to the IO bus. The fast memory (FM) appears with the arithmetic logic, reflecting its function as a set of accumulators. However it is also a part of memory, and as a set of index registers it is part of basic control, although in the latter case its output goes to the adder – the control logic performs the effective address calculation through the arithmetic logic (the discussion of fast memory control and addressing is in Chapter 5).

2.1 CONTROL

The most fundamental control circuit is a clock that times processor operations. The clock regulates a sequence of time states, which vary in length depending upon the operations that must be set up in them. Each clock pulse triggers the events that result from enabling levels set up during the previous time state and also triggers the events that set up the next time state. These states are grouped into cycles that carry out the different actions necessary for performing instructions and other special operations. The four basic cycles are instruction, fetch, execute and store (Figure 2-1). In the first of these the processor handles the instruction word and calculates the effective address, where each level of indirect addressing requires a repetition of the cycle. In the remaining cycles the processor fetches the operands, executes the instruction, and stores the result. Execution of the simpler instructions requires only the basic execute cycle, but other instructions require special execution sequences of varying complexity between the execute cycle and the store cycle.

To begin each instruction the control logic supplies an address to the memory interface and requests that an instruction be fetched. Initially these events are triggered from the console, but as the program progresses each instruction triggers the fetching of the next. The instruction address is supplied to the address bus AB as shown at C5 in the block diagram. Initially an address comes from the console address switches, and for a jump a new address comes from the adder AD, but for the normal sequence PC supplies an address through a +1 gate to reference the next location in sequence. The address from the bus goes to the memory address register MA with appropriate modification by the paging hardware. The address also goes to PC, which may therefore receive a new address, but usually receives its previous contents incremented by one (a location is skipped by sending a PC address around the loop without referencing memory).

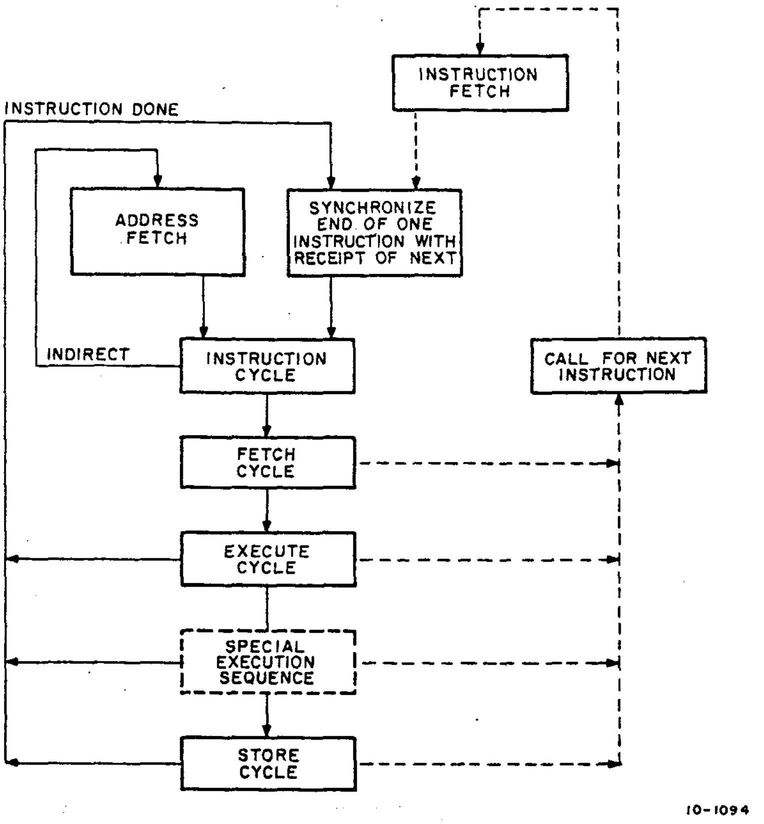


Figure 2-1 Basic Cycles

At the beginning of the instruction cycle, the instruction word from memory is received by the memory buffer MB (at the lower left), from which the instruction code and accumulator fields are latched into IR; this latter register can also be set up from the console for initial read in. A large network associated with IR decodes the instruction code to determine what operations to perform. If indexing is called for, the address part of the instruction word from MB is added in AD to the contents of an index register from FM; and if the address is indirect, the result is sent to AB to retrieve an address word. This also comes to MB but without affecting IR.

The instruction cycle is repeated until the effective address E is in AD, whereupon the fetch cycle sends it to AB for fetching the memory operand, which comes into AR from the memory bus. At the same time the AC address is used to select a fast memory register to supply the AC operand. After the various actions necessary for the execution of an instruction are done, the result is stored in memory, an accumulator, or both, as required. The FM selection (shown at the right) includes two bits supplied by a DATAO PAG, to select the fast memory block (user only) and four bits for the address of the location in the selected block. The 4-bit address is from MB for addressing an index register, from MA when FM is addressed for a memory operand (when fast memory is treated as part of memory), from IR for addressing an accumulator, and from IR through a +1 gate for addressing a second accumulator.

Also covered as part of basic control in Chapter 5 are trapping, machine modes, the response to a page failure, and console operations (except the readin function, which uses the in-out logic and is treated in Chapter 9). The mode logic includes control over switching from one mode to another, determination of the type of paging, detection of an illegal entry into a concealed area, and the base registers that point to the user and executive process tables. When the paging circuits in the memory logic detect a page failure, the processor enters a special page fail cycle in which it executes appropriate recovery procedures, constructs a page fail word by means of the so-called magic numbers, and generally enters a page fail trap.

The memory indicators MI on the console can display information from memory via MB, but the program can also load them with an IO instruction. The operator can load the data and address switches from MI; with IO instructions, the program can load the address switches and read the data switches.

2.2 MEMORY INTERFACE

Most of the interface between the processor and the memory bus comprises control circuits for the memory subroutine; this includes page checking, timing the sequence of events for calling and responding to memory, overlapping memory subroutines, and overlapping calls on the memory bus. However the interface does include the MB and MA registers and the associative memory, which supplies the information for constructing physical addresses from the virtual addresses supplied by the program. For a direct, unpaged reference MA receives the 18-bit address from the address bus, and this is expanded to 22 bits for an absolute address from the console. Otherwise the left nine bits of the virtual address are compared with the entries in the virtual part of the page table for the type of paging in effect; and if there is a match, the physical part of the table supplies the three bits for checking the type of reference and the most significant thirteen bits of the physical address. If there is no match, memory control performs a refill cycle in which a half word from the page map via MB supplies new mapping data to the associative memory at the location specified by the associative memory address counter. This reload counter can be loaded and read by IO instructions, and it increments automatically whenever the location to which it points is used for a mapping.

Associated with MB is a net that checks the parity of each word received from memory and generates the parity bit for each word being sent to memory. Another interface element is the MA special logic, which supplies low order address bits to MA for special references to the process tables (trap, page failure, MUUO, interrupt, auto restart).

2.3 ARITHMETIC LOGIC

The arithmetic logic is in two major parts, the smaller of which contains the shift counter SC. Computations on exponents and computing position in byte manipulation are done in SC and its adder SCAD. SC is also used as a counter to control operations that require a sequence of steps, such as shifting, multiplying and dividing. Special numbers needed for particular manipulations and for counting steps are supplied through the SCAD data net as listed at the right. While SC is controlling a floating point operation, the exponent is saved in the floating exponent register FE.

The major part of the arithmetic logic is based on the full-word adder AD and three full-word registers, the arithmetic register AR, the selected location in fast memory used as a passive register, and a buffer register BR. Almost all of the simpler instructions are performed using only these elements and often only some of them. AD is the receiving point for input from the IO bus and special words are constructed in it from the magic numbers. It can be enabled to produce the equivalence function or the arithmetic sum of its inputs, but otherwise its output is the AND function of the inputs. AR can directly receive information from PC, IR, MA and the flags for the construction of PC words, UUO words and MAP words.

Operations at an intermediate level of complexity require the multiplier-quotient register MQ. This register holds the mask in byte manipulation (received via AD from the mask generator), but in all other operations acts as a right extension of AR even though it holds the multiplier in multiplication and the quotient is built up in it in division. The AR-MQ combination is used in a straightforward manner in double length shift and rotate instructions and in floating point add and subtract. In multiplication the multiplier is shifted out of it at the right as bits of the double length product are shifted in at the left (partial products are added into AR). In division it supplies bits of the low order part of the double length dividend to AR as bits of the quotient are brought in at the right.

Double precision floating point operations require the full capacity of the arithmetic logic with 28-bit left extensions of AD and AR. ADX and AD act together as a double precision adder, ARX and AR hold one double operand with MQ acting as a right extension for triple length manipulation, and FM and BR together hold the other double operand. Such operations even make arithmetic use of MB for temporary storage of the high part of the multiplier while the low part is being used in MQ and the high part of the quotient while the low part is being built up in MQ.

2.4 INPUT-OUTPUT

The basic elements of the processor in-out are in-out transfer control (IOT) and the interface to the IO bus (IOB). Together they control the movement of device codes, control signals and data over the IO bus. The timing of in-out operations is derived from the main clock but uses a special IOT timer to control a sequence of stretched time states, as operations over the bus are much slower than operations in the processor.

The requesting of interrupts by peripheral devices is handled and synchronized by a PI request sequence that uses both the main clock and a special PIR asynchronous clock. Some events in the sequence are timed by the main clock, but it makes no difference which time states the processor happens to be in. In responding to requests over the bus, the PIR sequence determines the channel and then sends out a request grant signal so the nearest device that is requesting an interrupt on that channel can send back an interrupt function word. Upon receiving the word, the PIR logic synchronizes the request to the processor clock in order to wait for the processor to interrupt the program and begin a PI cycle, which is simply a set of basic processor cycles devoted to executing an interrupt function. If an interrupt instruction produces overflow, the processor executes a second PI cycle. Upon completing one or two PI cycles, the processor may return to the interrupted program or may hold an interrupt by beginning an interrupt service routine. The hardware contains four sets of seven flags, two sets of which are used for synchronizing requests and holding interrupts on the seven channels. The other sets are used by the program for turning individual channels on and off and for forcing interrupts on individual channels.

The IO hardware in the processor includes interfaces for the basic IO equipment – reader, punch and teletypewriter. The information provided in section 2.12 and Chapter 3 of the reference manual is quite sufficient as an introduction to the detailed descriptions of the hardware for these interfaces.

CHAPTER 3
OPERATION

This chapter describes the processor controls and indicators that are readily accessible to the operator and discusses the normal operation of the processor, reader, punch, and console teletypewriter. Some maintenance information is included, but descriptions of any controls and indicators mounted behind the doors of the processor bays and the detailed discussion of operation for maintenance purposes are in section 10.2.

3.1 CONTROL PANELS

Photographs of the various processor control panels are printed on foldouts in Appendix B. Most of the controls and indicators used for normal operation of the processor and for program debugging are located on the console operator panel and the small maintenance panel just above it; these are shown together in Figure B-1. The panels at the tops of the bays contain only indicators, most of which are for maintenance (Figures B-2 to B-4).

In the upper half of the operator panel are four rows of indicators, and below them are three rows of two-position keys and switches. Physically both are pushbuttons, but the keys are momentary contact whereas the switches are alternate action. Relative to the internal logic, the switches are actually flipflops that are controlled by the buttons but which in many cases can also be "operated" by the program. A switch is on or represents a 1 when it is illuminated. Buttons that actually trigger operating sequences in the processor are the operating keys, which are located in the right half of the bottom row. Operating switches are those that supply control levels for governing various processor operations; these include the buttons in the left half of the bottom row (except SINGLE PULSER), the paging switches at the left end of the third row, and the buttons at the left in the top two rows at the left end of the maintenance panel. The remaining buttons are sense switches, groups that constitute switch registers, and various other special keys and switches that supply information to the program or to specific hardware functions, or perform special functions of various sorts separate from the normal processor operating sequence.

The thirty-six numbered switches in the second row from the bottom on the operator panel and the twenty-two numbered switches in the row above them are the data and address switches through which the operator can supply words and addresses for the program and for use in conjunction with the operating keys and switches. At the right end of each of these switch registers is a pair of keys that clear or load all the switches in the register together. The load button sets up the switches according to the contents of the corresponding bits of the memory indicators (MI) in the fourth row. At the left end of the maintenance panel are switches to select the device for readin mode and a set of sense switches, which can be interrogated by the program.

The center section of the maintenance panel contains a voltmeter and controls for margin checking, and the right section contains speed controls for slowing down the program. Between these is a counter that registers the total time processor power has been on (the counter reads hours if the line frequency is 50 Hz, but at 60 Hz it counts six for every five hours). Below the counter are four special buttons, two of which are locks that are used to prevent inadvertent manipulation of the keys and switches while the processor is running: the console data lock disables the data and sense switches; the console lock disables all other buttons except those that are mechanical, which group comprises the four under the counter and the readin device switches.

Power is supplied to the system by means of the switch at the right end in the group under the counter. This switch is lit while power is on, but the power light in the upper right corner of the operator panel is lit only when the system is actually in operation or is ready for operation; after power turnon the light does not come on until power is stabilized in the correct range. At the left of the margin check controls are three red lights that indicate an overtemperature condition somewhere in the processor logic, a tripped circuit breaker, or a coding assembly door open. Whenever any of these lights goes on the Power Failure flag sets and power automatically shuts down.

3.1.1 Indicators

When any indicator is lit the associated flipflop is 1 or the associated function is true. Some indicators display useful information while the processor is running, but many change too frequently and can be discussed only in terms of the information they display when the processor is stopped. The program can stop the processor only at the completion of the HALT instruction; the operator can stop it at the end of every instruction, in certain memory references, or following every clock pulse (the last allows extremely slow speed operation with the clock running slowly or each clock pulse triggered individually by the operator).

Of the large groups of lights on the operator panel, the right half of the second row displays the contents of PC, the third row displays the instruction being executed or just completed, and the fourth row is the memory indicators. The left third of the third row displays IR; in an IO instruction the left three instruction lights are on, the remaining instruction lights and the left accumulator light are the device code, and the remaining accumulator lights complete the instruction code. The right half of the row displays the virtual address on the address bus, and the I and index lights reflect the states of the corresponding bits of the memory buffer. Hence the right two thirds of the row changes with every memory reference, and the I and index lights actually display the indirect bit and the index register address only following an instruction fetch or an indirect reference in an effective address calculation.

Above the memory indicators appear two pairs of words, PROGRAM DATA and MEMORY DATA. If the triangular light beside the former pair is on, the indicators display a word supplied by a DATAO PI; if any other data is displayed the light beside MEMORY DATA is on instead. While the processor is running, the addresses used for memory reference are compared with the contents of the address switches in a manner determined by the paging switches and the User Address Compare Enable flag. Whenever the two addresses are equal and the comparison is enabled, the contents of the addressed location are displayed in the memory indicators. However, once the program loads the indicators, they can be changed only by the program until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key (see below).

The four sets of seven lights at the left display the state of the priority interrupt channels. The PI ACTIVE lights indicate which channels are on. The IOB PI REQUEST lights indicate which channels are receiving request signals over the in-out bus; the PI REQUEST lights indicate channels on which the processor has accepted requests. Except in the case of a program-initiated interrupt, a REQUEST light can go on only if the corresponding ACTIVE light is on. The PI IN PROGRESS lights indicate channels on which interrupts are currently being held; the channel that is actually being serviced is the lowest-numbered one whose light is on. When an IN PROGRESS light goes on, the corresponding REQUEST goes off and cannot go on again until IN PROGRESS goes off when the interrupt is dismissed. PI ON indicates the priority interrupt system is active, so interrupts can be started (this corresponds to CONI PI, bit 28). PI OK 8 indicates that there is no interrupt being held and no channel waiting for an interrupt; this signal is used by the real time clock to discount interrupt time while timing user programs.

Note: If a REQUEST light stays on indefinitely with the associated IN PROGRESS light off and PC is static, check the PI CYC light on the indicator panel at the top of the console. If it is on, a faulty program has hung up the processor. Press RESET.

The four lights at the center of the top row indicate the processor mode. One and only one of these lights can be on and they represent the combined states of the User and Public flags. The rest of the top row contains the power light and the following control indicators.

RUN

The processor is in normal operation with one instruction following another (although the light remains on at a stop in a memory reference). When the light goes off, the processor stops.

STOP MAN

The operator has stopped the processor by pressing STOP or RESET.

STOP PROG

The processor has been stopped by a HALT instruction. At the completion of the instruction the address lights display the jump address (the location from which the next instruction will be taken if the operator presses the continue key), and the AR lights at the top of bay 2 display an address one greater than that of the location containing the instruction that caused the halt.

STOP MEM

The processor has stopped at a memory reference. This can be due to satisfaction of an address condition selected at the console, reference to a nonexistent memory location, or detection of a parity error.

KEY MAINT

One of the following switches is on (this light is equivalent to CONI APR, bit 8): FM MANUAL, MEM OVERLAP DIS, SINGLE PULSE, MARGIN ENABLE, SINGLE INST, STOP PAR. Any one of these switches being on implies that the processor is being operated for maintenance purposes, and is not running at maximum speed.

KEY PG FAIL

A key function has caused a page failure. No page fail trap is executed in response to a key-induced failure; if the processor is running, it continues the program.

The remaining processor lights are on the indicator panels at the tops of the bays. No attempt is made here to explain the meanings of these lights, as that is effectively the objective of the next five chapters – the lights reflect the logic of the machine. The large groups of lights on the panel at the top of bay 2 (Figure B-3) display the contents of the adder, the AR, BR and MQ registers, and the selected location in fast memory. At the right of the registers are a number of miscellaneous control signals, primarily enables for the shift counter, but also the enables for the IR latches and signals associated with the fetching and completion of an instruction. At the right end the upper four rows display the SC adder, its data inputs, and the shift counter and floating exponent register. The bottom row displays the AR flags, where FXU is Floating (exponent) Underflow and DCK is No Divide (divide check). FXU HOLD is a nonprogram flag that plays a role in determining underflow conditions. At the end are the flipflops that inhibit the clock and prolong its period.

The right halves of the top two rows of the bay 1 panel (Figure B-2) display the contents of the AD and AR extensions. Below these are three general flags used by the hardware and the enables for AD and ADX. The rest of the lights in the top four rows display all of the time state flipflops, flags and special control levels for the processor cycles, traps, and special sequences except those for in-out, priority interrupt, page fail and key functions. BYF6 in the top row is the First Part Done flag; the TN lights at the right end of the fourth row are the trap flags (TN 0 is Trap 2). The right half of the bottom row displays the physical address for each memory reference and the type of memory request. At the left are the lights for the associative memory. The AB 14–17 lights at the center are always either off or reflect the states of address switches 14–17.

The lights in the top row of the panel on the console bay (bay 3, Figure B-4) display either the contents of the in-out bus, the paper tape reader buffer, MB, or the information supplied by the last DATO PAG, as selected by the 4-position switch in the right section of the maintenance panel. The rest of the panel displays the user and executive base registers, and a multitude of signals for memory control, fast memory control, the key logic, paging, priority interrupt, in-out, the basic in-out devices (reader, punch, teletypewriter) and the processor flags. Note that the TRAP ENABLE light at the center of the second row is the Page Enable flag, which also enables overflow traps (DATAI PAG, bit 22). PAGE LAST MUUO PUB at the very center of the panel is the Disable Bypass flag. The User IOT flag is in the middle of the third row, and COMP ADR BRK INH near the left end of the bottom row is Address Failure Inhibit.

3.1.2 Operating Keys

The operating keys can be used whether RUN is on or off. If the processor is running when a key is pressed, it simply pauses at an appropriate point in the program to perform a key cycle to execute the function. These key functions are effectively of three types. The first three keys on the left are for the initiating functions, read in, start, and continue: these functions place the processor in operation under conditions determined primarily by the function itself. The next two keys are for the terminating functions, stop and reset: if the processor is running, these functions stop it. The last five keys are for the independent functions, execute, examine, examine next, deposit, and deposit next. These functions have no inherent effect on processor operation: if the processor is not running it simply performs a key cycle and stops; if it is running, it pauses to perform a key cycle and continues the program. (However the data deposited or the instruction executed may have an effect.) Moreover the independent functions are affected by the setting of the paging switches, which determine the address space in which the function is performed.

The logic responds to the keys in two stages. When a key is pressed or several are pressed simultaneously, the logic latches them. From among the buttons latched, the processor then accepts the request for the function that has priority; the priority order is the same as the order of the keys from left to right on the panel except that reset has first priority. As soon as a function request is accepted, the corresponding button lights up and remains lit until the function is completed. If the processor is not already in operation, it performs the accepted function immediately; otherwise it saves the function until it can be performed. While any button is lit, however, no function request can be accepted; in other words, although the processor will interrupt the program to perform a key function, it will not interrupt one key function for another. It will however do one key latch while a key is lit and accept the highest priority latched function once the current function is done. Provision is also made in the logic so that the RESET key can be used to stop the processor no matter what.

READ IN

Clear all IO devices and all processor flags. Turn on RUN and EXEC MODE KERNEL (trapping and paging will both be disabled as TRAP ENABLE at the top of the console bay will be off). Execute DATAI *D*, 0 where *D* is the device code specified by the readin device switches at the left end of the maintenance panel (the rightmost device switch is for bit 9 of the instruction and thus selects the least significant octal digit (which is always 0 or 4) in the device code). Then execute a series of BLKI *D*, 0 instructions until the left half of location 0 reaches zero. After storing the last word in the block, fetch that word as an instruction from the location in which it was stored as specified by PC. Since RUN has been set the processor begins normal operation at the location containing the last word. (For information on the data format refer to the System Reference Manual.)

Note that the key function lasts throughout the processing of the entire block. This means that readin cannot be interrupted for another key function. Hence if it must be stopped (*eg* because of a crumpled tape), press RESET.

START

Turn on RUN and EXEC MODE KERNEL, and begin normal operation by fetching the instruction at the location specified by address switches 18–35. The memory subroutine for the instruction fetch loads the address into PC for the program to continue. This function does not disturb the flags or the IO equipment.

CONT

If STOP MEM is on begin normal operation at the point at which the processor is stopped in a memory subroutine. Otherwise turn on RUN and begin normal operation by fetching an instruction from the location specified by PC.

STOP

Turn off RUN so the processor stops with STOP MAN on. At the stop PC points to the location of the instruction that will be fetched if CONT is pressed (this is the instruction that would have been done next had the processor not stopped). The processor may stop in the middle of a two-part instruction, but pressing CONT restarts the instruction without repeating any first-part actions that would adversely affect the result.

RESET

Clear all IO devices, disable auto restart, high speed operation and margin programming, clear the processor flags (lighting EXEC MODE KERNEL), turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA), turn off RUN and stop the processor.

If this function is not performed within 10 ms (*eg* because READ IN is lit), the key triggers a panic reset that produces all of the standard reset actions and also clears all but the mechanical console keys and switches. If STOP ever fails to stop the processor, pressing this key will, but not without destroying information. To save the processor state press SINGLE INST and SINGLE PULSE simultaneously.

XCT

Execute the contents of the data switches as an instruction without incrementing PC, even if a skip condition is satisfied in the instruction. If PAGING USER is on and PAGING EXEC is off, execute the instruction in user virtual address space; otherwise use executive address space. If the instruction is an XCT or LUUO, the instruction called by it is also executed.

Note that an instruction executed from the console can alter the processor state like any instruction in the program: it can halt the processor, can change PC by jumping, alter the flags, or even cause a non-existent memory stop (but not a page fail trap, even if it turns on the KEY PG FAIL light).

> NOTE
>
> The remaining key functions all reference memory. They can therefore light KEY PG FAIL and set such flags as Nonexistent Memory and Parity Error, and they all turn on the triangular light beside MEMORY DATA, turning off the light beside PROGRAM DATA. *Performing one of these functions with the ADDRESS STOP switch on stops the processor in the memory subroutine (with STOP MEM on).* These functions use an address supplied by the address switches, and the way that address is interpreted is determined by the paging switches. If both paging switches are off, the function uses a 22-bit absolute physical address supplied by address switches 14–35, and fast memory references are made to the block selected by the FM block switches at the left end of the maintenance panel. If either paging switch is set, the function uses a virtual address supplied by address switches 18–35 and the FM block switches have no effect (in other words the function has access to one of the virtual address spaces defined for a normal program). If PAGING EXEC is on, the function has access to executive address space; if PAGING EXEC is off and PAGING USER is on, the function has access to user address space.

EXAMINE THIS

Display the contents of the location specified by the paging and address switches in the memory indicators.

EXAMINE NEXT

Add 1 to the address displayed in the address switches, and display the contents of the location then specified by the paging and address switches in the memory indicators.

DEPOSIT

Deposit the contents of the data switches in the location specified by the paging and address switches, and display the word deposited in the memory indicators.

DEPOSIT NEXT

Add 1 to the address displayed in the address switches, deposit the contents of the data switches in the location then specified by the paging and address switches, and display the word deposited in the memory indicators.

3.1.3 Operating Switches

Besides defining the address space for the independent key functions, the paging switches also perform this service for address comparison and for the group of five switches just at the left of the operating keys. Whenever the processor references memory or an accumulator, it may compare the virtual address used with that specified by address switches 18–35 and may take some action if the two are identical. There are a number of conditions that affect the comparison. First, comparison can be made only for memory references and accumulator write references – there is never a comparison for an index register reference or an accumulator read reference. Given the proper type of reference, the comparison must be enabled. If PAGING EXEC is on and PAGING USER is off, the comparison is enabled for executive address space; if PAGING EXEC is off and PAGING USER is on, the comparison is enabled for user address space provided the program has turned on USER ADR COMP (User Address Compare Enable flag) in the upper left corner of the console indicator panel; if both paging switches are on, the comparison is enabled for executive address space, provided USER ADR COMP is on (in other words with both switches on, PAGING USER applies the flag condition to PAGING EXEC). In a reference of the correct type with the comparison enabled, if the virtual address on the address bus is identical to the address in switches 18–35, the processor displays the contents of the addressed location or accumulator in the memory indicators (unless the light beside PROGRAM DATA is on).

Except in an AC reference, the same situation that causes the word display can also be made to stop the processor or produce an address failure, depending upon the purpose for which the reference is made as selected by the three address condition switches. FETCH INST selects the condition that access is for retrieval of an instruction, including an instruction executed by an XCT, a user LUUO or a dispatch interrupt, but not a trap or standard interrupt instruction, nor one executed by an executive LUUO. FETCH DATA selects read-only access for retrieval of an operand or an address word in an effective address calculation, but not the PC word in an MUUO. WRITE selects access for writing, including read-modify-write, but not writing done by an MUUO or executive LUUO. Whenever a memory reference satisfies both the comparison condition and any selected address condition, the processor performs the action selected by the other two switches. ADDRESS STOP halts the processor with STOP MEM on and PC pointing to the instruction that was being performed (running with ADDRESS STOP on slows down the processor). ADDRESS BREAK causes an address failure except in an instruction performed while COMP ADR BRK INH is on. ADDRESS STOP also stops any examine or deposit function in the memory subroutine.

Conditions associated with the comparison are displayed by the COMP lights in the middle of the bay 3 indicator panel. From left to right these indicate an accumulator write reference, a memory read reference, equal addresses in a synchronous reference (an operand reference, but limited to the first in a double operand) and equal addresses in an asynchronous reference (an instruction fetch or the second in a double operand).

The description of each switch relates the action it produces while it is on.

SINGLE INST

Whenever the processor is placed in operation, clear RUN so that it stops at the end of the first instruction. Hence the operator can step through a program one instruction at a time, pressing START for the first one and CONT for subsequent ones. Each time the processor stops, the lights display the same information as when STOP is pressed. Note that read in cannot be done in single instruction mode, as the function extends over many instructions and there is thus no way to continue.

APR CLK FLAG (Clock flag) on the console indicator panel is held off to prevent clock interrupts while SINGLE INST is on. Otherwise interrupts would occur at a faster rate than the instructions.

> CAUTION
>
> It is not generally worthwhile to attempt to use the interrupt system in single instruction mode except with the slowest start-stop devices, such as reader, punch and teletypewriter. In any event an interrupt hangs up the processor, and the operator must dispose of it manually before single instruction operation can continue.

SINGLE INST will not stop the processor if a hangup prevents it from getting to the end of an instruction. Use STOP, RESET, or SINGLE PULSE.

SINGLE PULSE

Inhibit the clock so that a single clock pulse is generated each time SINGLE PULSER is pressed. If the processor is not already in operation, an operating key must be pressed before SINGLE PULSER can be used. If the processor is running, it converts to single pulse operation at the beginning of the instruction cycle; hence the clock will not stop if the processor does not reach the instruction cycle, say because it is hung up in a multiply or divide sequence. To force the processor into single pulse operation regardless of its position in the operating sequence, turn on both SINGLE INST and SINGLE PULSE; this stops the processor without destroying information, as would occur if RESET were pressed.

STOP PAR

Stop with STOP MEM on at the end of any memory reference in which even parity is detected in a word read. A parity stop is indicated by the following: PAR ERR FLAG (Parity Error flag) is on in the bottom row on the bay 3 indicator panel; and among the PAR lights in the third row from the bottom, ERR is on, IGN (ignore parity) is off, and BIT displays the parity bit for the word read.

If IGN is on (it displays a signal from the memory), parity errors are not detected and no stop can occur. Running with STOP PAR on slows down the processor.

STOP NXM

Stop with STOP MEM on if a memory reference is attempted but the memory does not respond within 100 μs. This type of stop is indicated by FLAGS NXM (Nonexistent Memory flag) being on in the bottom row on the bay 3 indicator panel.

REPEAT

If SINGLE PULSE is on and the processor is placed in operation, slow down the clock so that the processor runs at a clock rate determined by the speed controls at the right end of the maintenance panel. If the processor is not already running, it can be placed in single-pulse repeat operation by pressing an operating key and then pressing SINGLE PULSER. If the processor is running and the switches are turned on in the order REPEAT/SINGLE PULSE, then it goes into single pulse operation automatically at the beginning of the instruction cycle. If the processor is running with REPEAT off, it stops at the beginning of the instruction cycle when SINGLE PULSE is turned on; to restart it, turn on REPEAT and then press SINGLE PULSER twice. In any event repetition ceases (and the light in the SINGLE PULSER button goes off) whenever the processor gets to a point where the clock would have stopped anyway had SINGLE PULSE not been on. To restart, simply press SINGLE PULSER. The lamp in the SINGLE PULSER button goes off at each clock pulse and turns back on each time the clock is retriggered; hence the button glows with an intensity that is relative to the clock duty cycle (*eg* for a given speed, the light will be dimmer for a program with many memory references). When either REPEAT or SINGLE PULSE is turned off, operation terminates after one more clock.

If SINGLE PULSE is off and any operating key is pressed, then every time the repeat delay can be retriggered, wait a period of time determined by the setting of the speed control and repeat the given key function. The point at which the processor can restart the repeat delay depends upon the type of key function being repeated as follows.

> For an initiating function the delay starts when the processor stops with RUN off. This is either when the program gives a HALT instruction (STOP PROG) or following the first instruction if SINGLE INST is on.
>
> For an independent function the delay starts every time the function is done whether RUN is on or off.
>
> A terminating function stops the processor and the delay starts every time the function is repeated. Reset is generally used only to provide a chain of reset pulses on the IO bus, and stop is used to troubleshoot the clock.

In any case continue to repeat the function until REPEAT is turned off. (The function is often repeated once more, but this is noticeable only with very long repeat delays.)

The speed control includes a six-position switch that selects the delay range and a potentiometer for fine adjustment within the range. Delay ranges are as follows.

Position	*Range*
1	200 ns to 2 μs
2	2 μs to 20 μs
3	20 μs to 500 μs
4	500 μs to 6 ms
5	6 ms to 160 ms
6	160 ms to 4 seconds

MI PROG DIS

Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA) and inhibit the program from loading any switches or displaying any information in the memory indicators. The indicators will thus continually display the contents of locations selected from the console.

MEM OVERLAP DIS

Prevent memory control from overlapping cycles on the memory bus. (This has no effect on pipelining within memory control, such as overlapping the page checking of consecutive memory subroutines.)

MARGIN ENABLE

Enable maintenance operation, including writing with even parity in memory and checking speed or voltage margins. Maintenance actions attempted by the program are indicated by the last four lights on the left end of the second row from the bottom on the bay 3 indicator panel. With maintenance operation enabled, writing with even parity and checking speed margins are otherwise entirely under program control. Voltage margins may be checked by the program or the operator (for information on the margin select and manual margin address switches, refer to section 10.2).

FM MANUAL

All fast memory references for any purpose (index register, accumulator, memory) and under any conditions are made to the fast memory block selected by the FM block switches. When FM MANUAL is off, the block switches control fast memory references only in examine and deposit type key functions with both paging switches off (*ie* with the function using physical addressing). Turning on FM MANUAL overrides all other conditions so that all fast memory references are controlled by the block switches.

3.1.4 Panel Maintenance

A panel indicator is worthless if the bulb is burned out. Before attempting to use the information presented by the panels, press the LAMP TEST button below the counter on the maintenance panel; this turns on all of the lamps so any that are burned out can easily be detected. To replace a lamp in a button, pull out the button cap. The bulb will come with the cap, so remove it, put a new one in the cap, and push the cap back into the panel.

> CAUTION
>
> The lamp test checks not only the bulbs but also indicator driver transistors and indicator cables. Hence after changing a bulb always repeat the test. If the replacement bulb will not light, the problem is probably more serious and requires more significant corrective maintenance.

Replacing a bulb for an indicator requires removal of the panel. To remove the operator panel on early units, place your hands at the bottom corners and press in on the flush catches that are underneath the ends of the panel. On later units, remove the panel by grasping the small black latches on either side; then remove the inner aluminum lamp shield by lifting it out. The panel will snap free and can be pulled away. Pull out the bad bulb and insert a fresh one, but exercise some care in doing this – the bulb has a pair of pins that must be inserted in the socket, and shorting the terminals will burn out a transistor. After replacing the bad bulbs, snap the panel back in place.

The indicator panels are hinged at the bottom. Grasp the panel at the top and pull down. It is unnecessary to return to the console to find the bad bulb, as each bay has its own lamp test button, located at the left of the lights behind the panel.

On the maintenance panel the only lamps not in buttons are those for the failure indicators. To remove the panel, first remove the four switch knobs, each of which is held to its shaft by a pair of Allen set screws. The panel is held in place by Velcro strips at the ends and can be removed simply by pulling it out. Since handling the maintenance panel is somewhat of a chore, it is probably best to replace all three bulbs whenever one burns out. Press the panel back in place, then put on the knobs oriented so that the set screws are against the flats of the shafts. Tighten the screws lightly, rock each switch to make sure it is oriented properly, and then tighten the screws thoroughly.

3.2 CONSOLE IN-OUT EQUIPMENT

The console teletypewriter is generally on a stand by the console. The reader and punch are located in a drawer above the maintenance panel, but the face of the reader is available on the front of the drawer, and at its right are a slot for removing tape from the punch and a pair of rocker switches for feeding tape through reader or punch. Indicators for all three devices are on the panel at the top of the console.

The ASCII code used with all console teletypewriters and generally used for alphanumeric data on paper tape is shown on page A21. The main part of the diagram shows the configuration of bits 1–5 or channels 1–5 for the various characters in four sets, where a dot represents a 1 or a hole (the orientation of the tape is indicated by the little circles, which represent feed holes). At the left end is the configuration for channels 6 and 7 for the four sets of characters. Channel 8 (which is not shown) is ordinarily used for parity. For the standard even parity of the teletypewriter, the eighth bit is 1 for the characters printed on the dark background.

3.2.1 Reader

The contents of the reader buffer can be displayed in the top row of lights on the indicator panel by setting the IND SELECT switch on the maintenance panel to PTR. The remaining indicators for the reader are the PTR lights at the middle of the bottom row and near the left end of the second row.

Tapes for the reader must be unoiled and opaque. To load the reader, place the fanfold tape stack vertically in the bin at the right, oriented so that the front end of the tape is nearer the read head and the feed holes are away from you. Lift the gate, take three or four folds of tape from the bin, and slip the tape into the reader from the front. Carefully line up the feed holes with the sprocket teeth to avoid damaging the tape, and close the gate. Make sure that the part of the tape in the left bin is placed to correspond to the folds, otherwise it will not stack properly. If the program requires that the Tape flag be set and it is not, briefly press the feed switch at the right; setting the flag also sets Done to signal the program that the tape is loaded. After the program has finished reading the tape, run out the remaining trailer by pressing the feed switch.

3.2.2 Punch

The punch is behind the reader in the console drawer. Fanfold tape is fed from a box at the rear of the drawer. After it is punched, the tape moves into a storage bin from which the operator may remove it through the slot on the front. Pushing the feed switch beside the slot clears the punch buffer and punches blank tape as long as it is held in. Busy being set prevents the switch from clearing the buffer, so pressing it cannot interfere with program punching.

To load tape, first empty the chad box behind the punch. Then tear off the top of a box of fanfold tape (the top has a single flap; the bottom of the box has a small flap in the center as well as the flap that extends the full length of the box). Set the box in the frame at the back and thread the tape through the punch mechanism. The arrows on the tape should be underneath and should point in the direction of tape motion. If they are on top, turn the box around. If they point in the opposite direction, the box was opened at the wrong end; remove the box, seal up the bottom, open the top, and thread the tape correctly.

To facilitate loading, tear or cut the end of the tape diagonally. Thread the tape under the out-of-tape plate, over the rear guide plate, and through the punch die block; open the front guide plate (over the sprocket wheel), push the tape beyond the sprocket wheel, and close the front guide plate. Press the feed switch long enough to punch about a foot and a half of leader. Make sure the tape is feeding and folding properly in the storage bin.

To remove a length of perforated tape from the bin, first press the feed switch long enough to provide an adequate trailer at the end of the tape (and also leader at the beginning of the next length of tape). Remove the tape from the bin and tear it off at a fold within the area in which only feed holes are punched. Make sure that the tape left in the bin is stacked to correspond to the folds; otherwise, it will not stack properly as it is being punched. After removal, turn the tape stack over so the beginning of the tape is on top, and *label it* with *name, date,* and other appropriate **information.**

Indicators for the punch are the PTP lights near the middle of the bottom two rows. The numbered lights display the last line punched.

3.2.3 Teletypewriter

Connections for the console teletypewriter are at the vertical panel behind the door below the console table (Figure 3-1). In the upper half of this panel are two rotary switches that select the input and output speed, a toggle switch that selects the signal type, and a pair of sockets for the signal cable. The console teletypewriter is usually an LT35A Teletype (KSR), for which both rotary switches are set to 110, the toggle is set to CURRENT, and the signal cable is plugged into the upper socket. (For a Model 37 the speed would be 150, the toggle would be set to EIA, and the cable would be plugged into the lower socket.) In the lower half of the panel are switched and unswitched convenience outlets for the teletypewriter, scopes, and other equipment used at the console.

The teletypewriter is actually two independent devices, keyboard and printer, which can be operated simultaneously. Power must be turned on by the operator. The switch is beside the keyboard, and has an unmarked third position (opposite ON), which turns on power but with the machine off line so it can be used like a typewriter. The keyboard resembles that of a standard typewriter. Codes for printable characters on the upper parts of the keytops are transmitted by using the shift key; most control codes require use of the control key. The line feed spaces the paper vertically at six lines to the inch, and must be combined with a return to start a new line. In line with the space bar are four red buttons, the left one of which is not connected. The local line feed and return keys affect the printer directly and do not transmit codes. Pressing REPT and striking any character key (with or without the shift key on) causes repeated transmission of the corresponding code so long as the repeat button is held down.

Indicators for the teletypewriter are the TTI, TTO and TTY lights in the right half of the bottom two rows. The numbered TTI lights display the last character typed in from the keyboard.

Procedures for loading paper and changing the ribbon are given below. For further information refer to Typing Unit section 574-220-100 in Teletype Bulletin 281B, Vol. 1 (*Technical Manual: Model 35 Keyboard Send-Receive (KSR) and Receive-Only (RO) Teletypewriter Sets*), which is supplied with the machine.

Paper. The unit has a sprocket feed and uses 8-1/2 X 11 fanfold form paper. Printed forms can be torn off against the edge of the glass window in front of the platen. To replace the paper, first remove the upper cover by pressing the cover release buttons on the sides. To free the remaining old paper for removal, lift the paper guides by pushing the lever marked PUSH at the right of the platen. To load new paper, insert it through the slot in the cover and over the plate behind the platen, lining up the holes at the edges of the paper with the sprockets, and press the local line feed to draw the paper in under the platen.

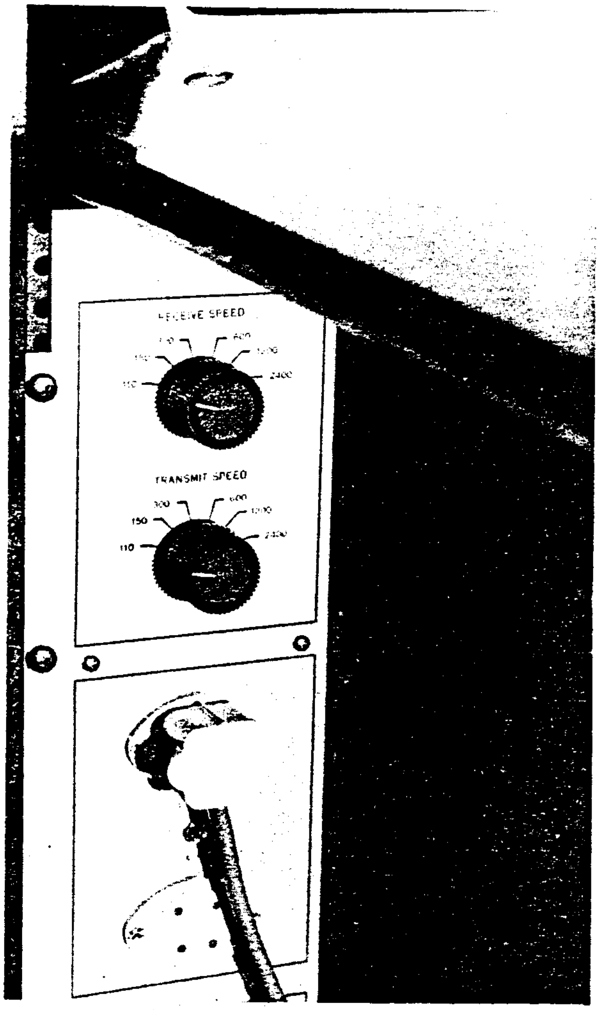


Figure 3-1 Console Teletypewriter Panel

Ribbon. Replace the ribbon whenever it becomes worn or frayed or when the printing becomes too light. Disengage the old ribbon from the ribbon guides on either side of the type block, and remove the reels by lifting the spring clips on the reel spindles and pulling the reels off (the ribbon feed mechanism is called out in Figure 4 in the reference mentioned above). Remove the old tape from one of the reels and replace the empty reel on one side of the machine; install a new reel on the other side. Push down both reel spindle spring clips to secure the reels. Unwind the fresh ribbon from the inside of the supply reel, over the guide roller, through the two guides on either side of the type block, out around the other guide roller, and back onto the inside of the takeup reel. Engage the hook on the end of the ribbon over the point of the arrow in the hub. Wind a few turns of the ribbon and make sure that the reversing eyelet has been wound onto the spool. Make sure the ribbon is seated properly and feeds correctly in operation.

CHAPTER 4
CONVENTIONS AND NOTATION

Accompanying every DEC system is a set of drawings that defines the system physically, electrically and functionally. Just below the title in the lower right corner of each drawing is the drawing identification written in four boxes. In the left box is a letter indicating the size of the original drawing; in the next is a two-letter code indicating the type of drawing. The third box contains the drawing number in three parts: the left part is the identification number of the equipment, the second is the drawing variation number, and the third is a number that identifies the individual drawing or a mnemonic code that identifies both the individual drawing and the material presented on it. If a drawing is revised after being signed by the project engineer, the revision letter is written in the right box. For a drawing with several sheets, both the sheet number and the number of sheets are written below and to the left of the size letter.

Almost all original drawings are D size but for convenience are reduced to B size. Some typical codes for drawing type are block schematic BS, flow diagram FD, circuit schematic CS, module utilization MU. Most of the drawings with this manual are for the KI10 processor and that designation is therefore used in the first part of most drawing numbers. All drawings are variation 0, corresponding to the standard production machine. If a particular machine has some special feature, the drawings that show the implementation of that feature have a different variation number; and if after a time, a different model of the KI10 becomes the standard production model, the drawings that reflect the differences from the earlier version will also have a different variation number. The description of the equipment is in terms of the drawings that represent the standard production machine at the time the manual was printed; the print set however reflects any later revisions and any special features unique to a given installation.

The drawings associated with the text are entirely block schematics and flow diagrams, which are often referred to respectively as logic drawings and flow charts. These are identified simply by the third part of the drawing number. *Eg* the logic for the page fail cycle is on BS drawing KI10-0-PF and the flow chart for it is FD drawing KI10-0-PF. In the text the former is referred to simply as "drawing PF" or "print PF", and the latter is referred to as "flow chart PF" or simply "chart PF". This chapter is devoted to discussing these drawings and how to use them to understand the machine. Other types of drawings and their use are discussed in section 10.1.

4.1 DESIGN CONVENTIONS

Before investigating the details of the processor logic or even the symbol conventions used in representing it, there are a number of characteristics of the implementation of that logic with which the reader should be familiar. With a thorough understanding of these basic logical characteristics, the task of learning the material presented in the next five chapters will be lessened considerably.

Time States and Clock. The processor performs each instruction by stepping through a sequence of time states that are timed by a clock (Figure 4-1). Each clock terminates a time state and performs some operations or events necessary for the execution of the instruction. The clock also sets a D-type flipflop (one with clock and data inputs) to place the processor in the next time state. The clock logic then waits for some period depending upon those actions taken; the period may be as little as 100 ns or as much as 230 ns. During the waiting period, the time state flipflop causes the generation of the various levels needed to gate the actions that will be taken at the next clock, *ie* at the end of the time state. The waiting period selected is that required for level transitions through the various logic networks. If one of the actions taken by a clock is a call to memory control for a memory subroutine, the clock stops. The processor in the meantime holds the new time state until the clock is restarted by a signal returned from memory control.

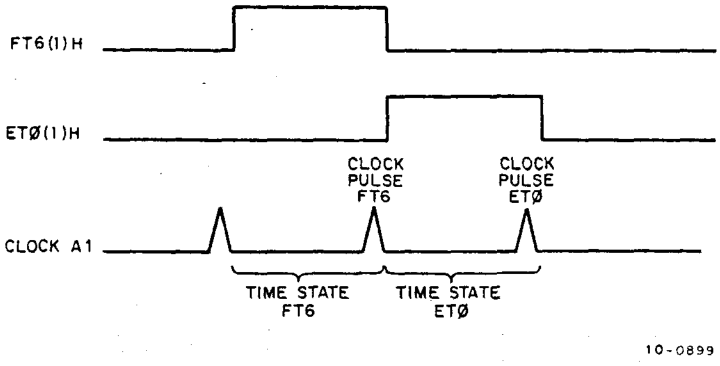


Figure 4-1 Time State/Clock Relationship

There is no difference between the clock that ends one time state and that which ends another. All clocks look alike. To perform the particular actions needed for the time state, the clock drive lines that spread like tentacles throughout the machine are gated by the 1 state of the time state flipflop in conjunction with IR decoder outputs, switches associated with the instruction, and sometimes functions of the data. In effect, the 1 state of the flipflop and the time state are logically equivalent. The events that occur at any clock are those that have been set up during the time state; and as a time state ends, the single time state flipflop that is set by the clock gives rise to the next.

Flipflops and Flags. Among the gating levels generated during a time state are enabling levels for the data inputs to various control flipflops, including the flipflop for the next time state. At the end of the time state, the clock sets any flipflops whose data inputs are enabled. If during the next time state the enabling level goes away (as it always does for a time state flipflop), then the next clock clears the flipflop. A flag is simply a flipflop that has some mechanism for staying on over a number of time states. The flag may be a regular clocked flipflop that is enabled by several consecutive time states; or it may be a regular flipflop that is clocked by a unique signal (not the time state clock). A flag may also be a flipflop constructed of gates; it thus has unique set and clear functions, which are independent of the clock except insofar as the time states play a role in generating those functions. But the most common flag is a regular clocked flipflop whose 1 state, anded with the negation of some time state or the negation of some clear signal, produces the enable level for its own data input. Hence once set, the flag stays on until a clock occurs in the specified time state or when the clear signal is true. Note however that the negation of one enabling level does not clear a flag if some other signal enables the data input at the same time; *ie* any enable signal applied to the data input overrides any signal that is not enabled.

Registers. Most registers are composed of regular clocked flipflops, which the clock loads from the inputs enabled during the time state. Registers AB, MA and IR are composed of latches, which freely follow the inputs enabled by an enabling flag; when such a register is latched, it then holds the information that is available at its inputs at the time the latching occurs. Finally the adder can be regarded as a register since its outputs are available for loading information into other registers. The adder however is loaded only in the sense that information can be enabled into its inputs by flipflops that select the source – in other words the adder freely follows the source and cannot be latched, but the flipflop keeps the same source available. Hence once the level transitions through the adder are finished, the output remains stable so long as the input remains stable (*ie* for the rest of the time state).

The differences in the way these elements operate makes a considerable difference in the way the signals written on the prints and in the flow charts must be interpreted. Suppose that a given time state produces the two gating levels BR AR EN and AD BR+ EN. The first of these means that the output of AR is enabled at the BR input gates (the signal name is read as "BR from AR enable"). But the meaning of the second is very different and is unique to the adders (AD, ADX, SCAD)' it means that the data input of the AD BR+ flipflop is enabled. Hence the clock that terminates the time state will actually load the contents of AR into BR, but will simply set AD BR+, whose 1 state will in turn enable the BR outputs into the adder (the term BR+ is used because the complement of BR, *ie* BR−, can also be enabled). Now the data received by BR is the data that was in AR during the time state, but the data enabled into the adder is not that which was contained in BR during the time state. Suppose AR contains *X* and BR contains *Y*. Then at the clock, *X* is loaded into BR; and during the next time state (*ie* during the waiting period between this clock and the next) the quantity *X* (now in BR) is also enabled into AD. In other words BR receives what AR contained, but the adder receives what BR receives. In terms of these two enabling levels, the quantity *Y* is simply lost. Failure to keep this point forcefully in mind while following through the instruction flow charts will result in total and utter confusion. The basic idea is that at a given clock a register receives the information contained in a source, whereas during the next time state the adder builds up an output from information received by a source. A register changes instantaneously at a clock, but the adder changes gradually between clocks.

Latches combine features of the two procedures. Information is held on the address bus AB by a set of latches. Setting AB AD enables the output of the adder onto the address bus, which changes as AD changes. When AB AB is subsequently set, latching the bus, the information latched is that held by the adder at the time the latching occurs.

Register Clocks. The main clock drives are triggered at every clock pulse. But since a flipflop is automatically cleared unless enabled, each flipflop register has its own clock, which must be enabled in order for the main clock to produce a clock for the register. This way the registers remain stable over a number of time states. The generation of an enabling level for the register input gating must also enable the register clock if the register is to be affected. Enabling the clock without enabling any of the input gating clears the register. To clear a register the time state may simply enable the clock directly or generate a clear level that in turn enables the clock and may cause other actions elsewhere. The only circumstance in which an input enabling level for a register does not enable its clock is when there is a choice between clearing and loading. Suppose at a given time state a register must be loaded if *X* is true, otherwise cleared. To do this the time state enables the clock directly but enables the input gating conditioned on *X*.

The clocks for all of the flipflop registers have true enabling levels. The clock for the adder logic is enabled by the negation of an inhibit – the adder logic is cleared at every clock unless the AD clock is specifically inhibited.

Overlapping, Pipelining, Prefetching. Memory control can overlap two read cycles in different memories. It can supply an address to one memory, and then send an address for a second access to another while waiting for the data from the first. This type of overlapping increases the number of memory cycles that can be handled over the bus; *eg* with memories interleaved, the fetch of a two-word operand is speeded up considerably. Although bus overlapping decreases waiting time in processor operations, it is a function of the interface between memory control and the bus, and hence does not enter into the overall flow of processor events.

Besides bus overlapping, there is also overlapping of the memory subroutines called by the processor time states. This is made possible by a pipelining technique, wherein memory control is divided into a sequence of functional entities that are operationally independent. *Eg* page checking is done with the address on the address bus, and this can be performed even though MA contains an address that is waiting to be sent out on the memory bus. The processor takes advantage of this pipelining by requesting simultaneous synchronous and asynchronous memory subroutines or requesting an instruction fetch before memory control is free. In the first instance the processor requests two subroutines in such a way that as each section of the pipeline becomes free in the first subroutine, memory control automatically makes use of it for the second. The first of these subroutines must be for an operand fetch or store, which is regarded as synchronous because the processor makes the request only when memory control is already free. There are two cases of this type, a double operand fetch, and an operand fetch or store combined with an automatic instruction fetch.

As soon as the page check is complete for the first of two operand fetches, the address is moved from the address bus to MA, and memory control recycles to perform the page check for the second fetch. Thus the second subroutine follows the first right down the pipeline. The double operand fetch is always of this type, because the address for the second operand is known in advance. In the other case, the processor can request an automatic instruction fetch at the same time as a synchronous subroutine request if PC already points to the next instruction (*ie* it cannot still be changed) and the current instruction will not make another request for a memory operand subroutine later on. The degree to which the pipelining is effective depends upon many conditions along the way. There are various synchronizers to prevent the second subroutine from overtaking the first. If the first subroutine is for writing in memory, then there is no overlapping on the bus and the second subroutine must wait for the write access to end. If the paging requires access to the page map, then it cannot be completed until the page refill cycle is done, and that cycle must itself live within the pipelining restrictions.

To prefetch an instruction is simply to request a memory subroutine to fetch the next instruction before the present instruction is done. Hence an automatic instruction fetch is a prefetch although there do exist circumstances in which the asynchronous subroutine may not get beyond the page check until the end of the store cycle. The processor can also prefetch by requesting an instruction fetch when the address bus is free, the address of the next instruction is available, and there are no further memory operand requests to be handled. The fact that a prefetch is requested independently does not mean that it occurs later or takes more time than an automatic request would. A prefetch may be as early as the beginning of the fetch cycle, as in a jump instruction that uses no operands and has the effective (jump) address available from AD immediately (an automatic fetch requires an address from PC). Unless memory control is completely free, how far the prefetch can go depends still on what is going on in the pipeline, but in many cases (*eg* an arithmetic instruction that stores the result only in an accumulator), the next instruction may already be waiting in MB by the time the current instruction is finished.

4.2 LOGIC DRAWINGS

The logic drawings (block schematics) are block diagrams that show the function and location of every logic element used in the processor. The logic modules and the symbols used to represent them are shown in a series of nine block schematics, PDP10-0-LE1 to LE9. These drawings are located at the beginning of Volume 2, DECsystem-10 Replacement Schematics, B-MN-PDP10-0-MODL. Drawing LE1 shows the most basic logic elements: the D-shaped symbol represents an AND gate, and the arrowhead-like symbol represents an OR gate. An input or output shown as a plain line represents a high level, a line with a circle on it represents a low level (invariably +3 V dc and ground in the processor). For those familiar with traditional DEC logic symbology, the circle is equivalent to a solid diamond or arrowhead, the plain line is equivalent to an open diamond or arrowhead. If the input (such as to a pulser) requires a transition, the appropriate waveform is written inside the symbol at the input. On the logic drawings every element is identified by the module type number given in the LE drawings and the location of every element is listed just below the type number. For each module the LE drawings list the power, ground and unused pins as well as the signal pins; only the last are shown on the logic drawings. Occasionally the symbol for a logic element contains a number in a square or circle; this indicates the adjustment procedure that must be performed if the module is replaced (complete information is given in section 10.4.)

Inspection of LE1 might give the impression that all simple gates are AND gates, but every AND gate is also an OR gate. Consider the AND gates shown at the top left of LE1. Since in each of these gates both inputs high produces a low output, it is evident that if either input is low the output is high. Hence an AND gate for high inputs is an OR gate for low inputs. In purely circuit terms one should consider both the function and signal polarity; thus if the input and output levels of an AND or OR gate are of opposite polarity, the gate is regarded as NAND or NOR. But since voltage levels per se are not important to functional understanding, in discussing the logic we shall regard the gates simply as AND or OR.

The simple gates are combined in a variety of ways to provide the OR of a number of AND functions or to provide extensive mixing with common enabling levels for register input gating as shown in LE1 and LE2. The symbol for an exclusive OR gate (XOR) is the OR symbol with an extra line at the back of the arrowhead as shown at the lower right in LE7. Most of the remaining logic elements are represented by boxes that are labeled for the logic function, such as a +1 gate, a decoder, or a parity net. It is recommended that the reader go through the LE drawings and familiarize himself with all of these logic element symbols.

Flipflops are all D-type and are represented by the symbols shown on LE6. A single module may have a number of independent flipflops, or a group that has a common clock input or common clock, clear and set inputs. In the rectangle representing the flipflop, the clock input and the clocked data input are always at the bottom left and right respectively (beneath the "0" and "1"). The unclocked clear and set inputs are always on the 0 and 1 sides. Outputs are always at the top. Note that the output terminals are drawn twice, showing the polarities associated with either state of the flipflop. The polarities of the output terminals for the 0 state are as shown at the left side of the rectangle (over the "0"); for the 1 state the terminals have the polarities shown over the "1". This agrees with the convention that neither voltage level categorically represents 1 or 0, true or false. A given logic function may have different assertion levels in different places depending upon gate input requirements. A signal is always regarded as true when it has the polarity shown for the input or output associated with it. In other words if the signal X appears at an input with a circle, then X is true when low. The very same physical signal line may appear elsewhere without the circle but with the signal designation $\sim X$; this is exactly equivalent, for now a high level on the line indicates that $\sim X$ is true, *ie* X is false.

The advantage in showing the two states of a flipflop at both polarities and in using logic levels at either polarity simply by negating the signal name is that there is never any need to invert the name of a signal as it appears at a logic net; all logical conditions appear in the drawing with correct truth values. When a flipflop output is used in a logic net, the signal name indicates the enabling state of the flipflop. To determine the physical source of the signal (the output terminal to which the signal line is connected), one must know both the name and the polarity. *Eg* the signal FF(1) at high assertion originates at the output terminal that is high when flipflop FF is in the 1 state; at low assertion this signal actually originates at the other output terminal. In any event it is always possible to determine whether any two logic points are connected together electrically, and gates are always represented in the simplest fashion consistent with this requirement "Simplest fashion" means in such a way that the Boolean equations can be read directly from the network without requiring mental gymnastics. A situation that occurs frequently in the block schematics is a signal generated by an AND gate, each of whose inputs is an OR gate (this is done to minimize the delay or the number of gates). Theoretically a four-input AND with two-input ORs would mean an equation with sixteen terms. However redundance and contradiction in the inputs invariably reduces this to only a few sensible cases. As an example consider the gate at the lower right in drawing CMP1. Although at first glance one may be terrified of this net, its equation actually has only three terms:

$$\begin{aligned}\text{COMP EN} \;=\; & (\text{KEY EXEC PAGING(1)} \wedge \text{KEY USER PAGING(0)} \wedge {\sim}\text{USER PAGING}) \;\vee \\ & (\text{KEY EXEC PAGING(0)} \wedge \text{KEY USER PAGING(1)} \wedge \text{USER PAGING} \wedge \text{PAGE ADR COMP(1)}) \;\vee \\ & (\text{KEY EXEC PAGING(1)} \wedge \text{KEY USER PAGING(1)} \wedge {\sim}\text{USER PAGING} \wedge \text{PAGE ADR COMP(1)})\end{aligned}$$

Most of the elements shown in the LE drawings are general in nature, although a few are special circuits for a particular KI10 purpose. Almost all are easily understandable simply by inspecting the block diagrams that represent them, and those that are not are treated in greater detail at appropriate places in the text. In particular the adder at the right on LE3 is very complex and is discussed at length in section 7.1. All elements are represented in the logic drawings as they are shown in the LE drawings with the exception of the latched register driver at the left on LE7. This circuit is used for the many clocks throughout the processor and appears in the logic drawings as the RD symbol shown at the right of the circuit. The flipflops associated with the switches are shown as they appear at the lower right on LE6, but the sketch at the left in the module drawing helps to explain how it works. In use, one of the outputs is connected back to the data input so that the flipflop changes state each time the momentary-contact button is pressed. The output may be fed back directly so that it always acts as a toggle, or it may be fed back through a mixer so that it acts as a switch-controlled toggle but can also be loaded by the processor logic.

4.2.1 Signal Notation

All signal names are mnemonics that indicate both the function of the signal and its source. In almost all cases, signal names are phrases, sometimes lengthy, whose meanings are unmistakable. Typical mnemonic terms used are LT left, RT right, EN enable, INST instruction, CLK clock, SH shift, F fetch, S store, SC shift count, CRY carry, etc. The names of signals associated with the various registers use the mnemonic register designations given in Chapter 2, and signals associated with the execution of individual instructions use the instruction mnemonics defined in the reference manual. Often the letter *X* is used to indicate a variable letter in a mnemonic, *eg* IR HLLXX represents the IR decoder output for all left-to-left half word transfers with any effect on the other half of the destination and any mode. Complete signal names are made up of as many pieces as are necessary for clarity. *Eg* right shifting in AR is produced by two logic signals, AR SH RT A and AR SH RT B. Each of these generates a pair of shift signals for each half of AR: the right shift enabling signals for the left half of AR are AR LT SH RT EN A and the same thing with B substituted for A at the end; the equivalent signals for the right half of AR are the same except that RT is substituted for LT.

Numerals representing register bits are merely appended to the register name: bit 8 in AR is AR 08 and bits 14–17 in MB are MB 14–17. The state of a flipflop is represented by a numeral in parentheses following the name: the 1 state of bit 8 in AR is AR 08(1). A level that enables a transfer between two registers has the name of the receiving register as its first term, followed by the name of the source, followed by EN. Hence the level that enables the next clock to load the flags into AR is AR FLAGS EN. Where a number of nets produce signals that all perform the same function, they all have the standard names but are differentiated by appending A, B, C, etc. A gating level that is true throughout an instruction and is gated by a time state to produce a specific function at that time state has the appropriate name with the name of the time state included. AR AD BR (ET0) transfers AD into AR and AR into BR, both at ET0.

In general the enabling level at the data input to a flipflop is indicated by the final term IN or EN. The former is used for flipflops in registers (*eg* MQ 34 IN); the latter is used for control flipflops such as the time states. There are exceptions to this, particularly in that a level labeled EN, perhaps generated by a number of conditions, may itself be applied to yet another gate whose output is connected to the data input. Where a continuous line connects an output to all points where it is used as input, it usually has only a wiring designation; in other words a signal is named only when it is used on other drawings, is used at disconnected points on the same drawing or goes through a cable.

The flipflops that control the time states have names of the form *XTN* where *X* is the name of the cycle or sequence and *N* is a number. Flags frequently are indicated by the letter *F*. *Eg* the double floating divide sequence has time states DFDT0, DFDT1, etc, and certain operations needed for several time states in the sequence are controlled by a flag DFDF1.

The source of any signal can be determined from its first term or its first letters. Each register with associated logic and each control system, whether it occupies several drawings, one drawing, or only part of a drawing, has a mnemonic designation that appears in the drawing number and at the beginning of the name of any signal originating in that part of the logic. This source code may appear naturally as part of the signal name; if not, it is merely prefixed to the name. Thus all signals derived from AR bits and all gating levels for AR have names that begin with AR. There is some further differentiation in that the special AR inputs have prefix ARI, and the signals associated with the AR flags have prefix ARF (although the flags themselves have names with prefix AR, as it is obvious that they are on the AR flag print). Usually names with identical prefixes are easily distinguishable; *eg* the AR bits themselves originate on the prints showing the AR register, as against AR gating levels, which are on the associated logic prints. In some cases there may be several drawings of similar functions with identical prefixes, and the reader must simply become used enough to the organization of the prints to know where to look for a given signal (in any event the ambiguity is never greater than two or three drawings). Any signal line without a name is labeled with a 3-digit number and is uniquely designated by that number appended to the drawing mnemonic.

In the memory control logic some signals have a three-letter prefix indicating the drawing; where the prefix is simply MC, the first letter of the next term identifies the drawing. Often designations that are seemingly ambiguous turn out to be unique upon closer investigation. There are many signals with prefix AD, but the AD AR+ logic is on print ADAP, and the AD AR- and AD MB nets are together on print ADAM. In cases where two or three different logical entities are on a single print, the reader must learn the combinations; *eg* the BLT logic happens to be with the double moves on print DMBL, and the FIX logic is with the divide subroutine on drawing DS. There is also some graphic ambiguity which disappears as soon as the reader becomes familiar with functional meanings. An obvious case is SC for shift count, wherein there are many signals beginning with SC, SCA and SCAD; at first glance SCE may appear to belong with these, but it is easily recognized as a store signal as soon as the reader realizes that it means "store contents of E".

Lastly the reader must beware of assuming that the prefix necessarily has functional significance, for such an assumption can lead to considerable confusion. The most obvious example is PI CYCLE DIVERTED, which definitely does not mean that the signal has anything to do with diverting a PI cycle. Rather it means that the cycle is diverted and the signal is generated on one of the PI prints (more often than not the diversion is to a PI cycle).

4.3 FLOW CHARTS

The flow charts show every event the processor performs and show the flow of operations in a manner consistent with the actual gating and timing in the hardware. The memory control flow charts are based on a sequence of level changes and timing pulses, whereas the rest of the flow charts for the processor are based on sequences of time states. In general flow is downward. Upward flow occurs only upon returning to an earlier part of a sequence (as in a loop) or when going from one flow line to the next when several are on the same drawing; in any case no conditions or events are ever shown along a rising line. As shown in Figure 4-2, unique time pulses are indicated by ellipses, but the main clock is shown in a five-sided, flag-shaped symbol labeled according to the time state it terminates. Events are shown in boxes that are at one side of a line, unless an event actually is responsible for further movement along the line. If an event occurs only on some condition, that condition is written at the left of a colon and the resulting event is written at the right. Brackets enclosing a condition mean that it is bound to be true by virtue of the flow line in which it appears; brackets enclosing an event indicate that although it does happen, it is irrelevant to the particular sequence. Parentheses, brackets and braces are used for grouping in logic functions; a semicolon means that commentary follows.

Essentially there is no passage of time along a continuous line. A break in a line indicates that movement along the line cannot continue beyond that point unless the condition written in the break is satisfied. This does not indicate any passage of time and the condition must be satisfied then; movement cannot restart should the condition later become true. Actual passage of time is indicated by horizontal lines breaking the flow line. A pair of single horizontal lines indicates a fixed delay, with the delay time written between the lines. Double lines indicate a call for a memory subroutine; between the line pairs there are always two terms, where the upper identifies the particular operation (operand fetch, page check), and the lower names the signal that is returned by the subroutine to restart the clock and thus continue the flow along the line. Two or more flow lines entering the top of a box (usually divided) that has double lines at top and bottom and a single flow line leaving the bottom indicates a synchronizer. This means a point at which parallel paths join, and movement continues out of the box only when all conditions written in it are satisfied.

To see how these symbols are used, refer to the flow chart of the instruction cycle, FD drawing IC. In the upper left corner is the entry into the sequence from a memory subroutine that fetches the instruction (this is the lower half of a subroutine symbol whose upper half appears somewhere else). From this subroutine, MCRST1 sets INST RDY provided the lengthy condition written in the break between the two is true. Now the box containing the event of setting INST RDY appears directly in the vertical flow line because events beyond that point are not produced by MCRST1 – they are produced by the transition of INST RDY to the 1 state. This transition clears MR RESTART B

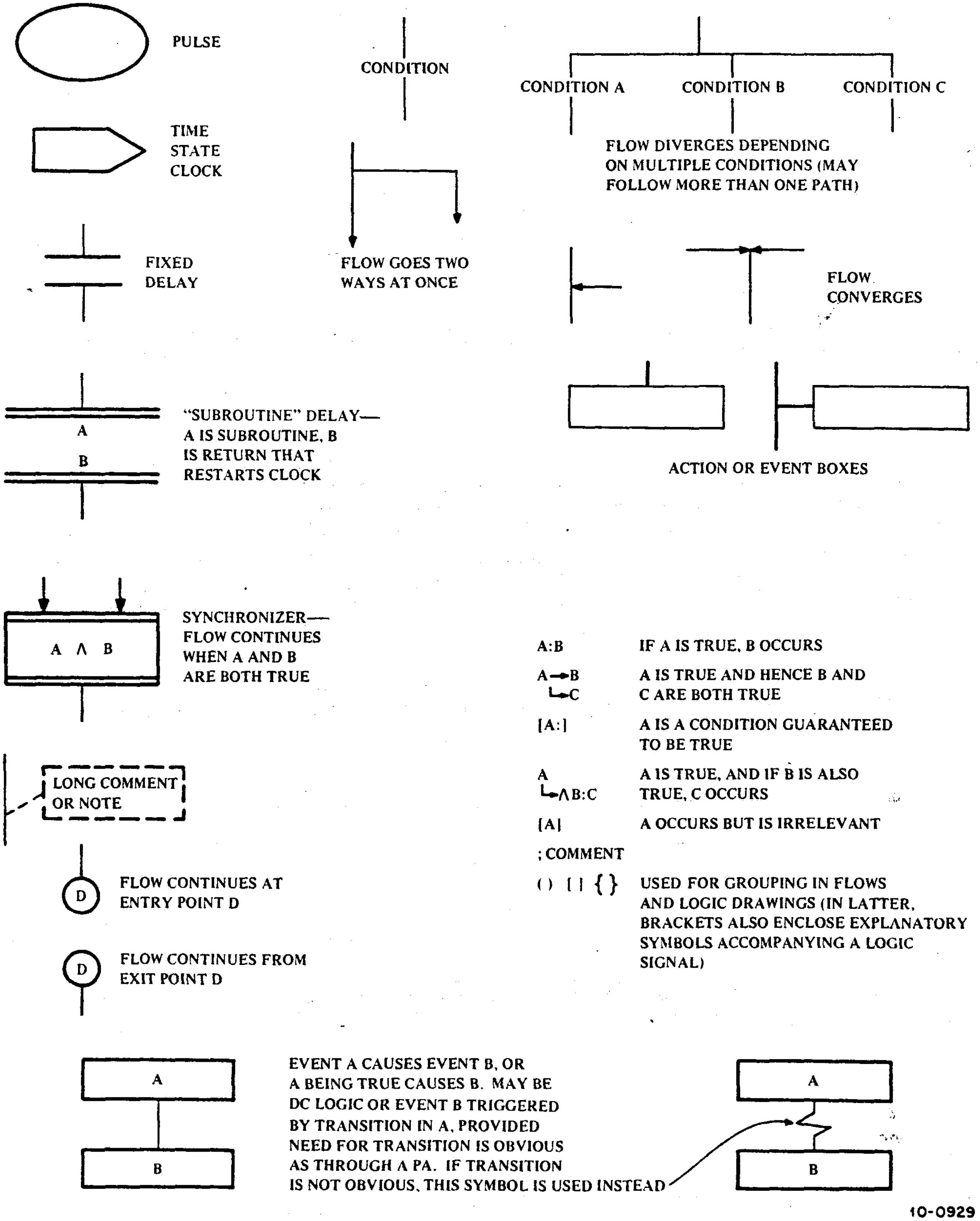


Figure 4-2 Flow Chart Conventions

(which does not enter into the flow path), and following a fixed delay the level INST RDY DLYD is generated provided the flipflop is still set. The double box below that indicates that flow continues only when INST RDY DLYD and INST DONE DLYD both become true, *ie* that this point has been reached along the flow line we have been considering and also along the flow line from the right. The satisfaction of these two conditions starts the clock (indicated by the fact that the line continues to a clock symbol), and if IT1 and IT2 are both 0 it also generates IT0(1). Note that a condition that allows passage along a line is not repeated as a condition in an event box, for until there is a time delay of some kind, it is implied that that condition is necessary for every event shown. The large box at the right of the IT0 clock shows all events that occur at that time, some conditioned and some not, except those that involve redirection of flow. The choice of which branch to follow below the box is determined by the conditions written in the breaks in the branches. And although the events listed in the next box along the lines occur only when the corresponding path is followed, nonetheless they occur at the same time as the events in the large box above; in other words from a given clock all events directly along a flow line are produced by that clock until some delay appears in the line. Note that along each of these branch lines there is an event box at the side, and a second box connected by a vertical line to the first. The events in each of these second boxes are not produced directly by the IT0 clock, but are instead produced by the bottom event listed in the box above. In other words when IT1 IN is true, the IT0 clock sets IT1, and its transition in turn clears INST RDY and disables IT0(1).

The flow continues in the same manner and the events shown among the many branches at the bottom center of the chart are produced by either the IT1 clock or the IT2 clock depending upon which flow line served as entry. Note that near the bottom of the flow line at the left, the condition F CYC START is enclosed in a dashed clock symbol. Although it is a condition that allows the IT1 clock to continue flow along the line, it also has many of the properties of a time state and is essentially a substate of IT1.

Finally there are two significant points that must be discussed concerning most of the items written in the boxes. First, they are not themselves the actions taken, and second, they do not actually happen at the time represented by the box. In reality they are the gating levels for the actions taken, and they are generated in the time state that is terminating, *ie* during the waiting period of the clock. However, the action taken is implied directly by the name of the gate. *Eg* a typical term is AD BR– EN, which indicates that the clock sets a flipflop AD BR–, which in turn gates the complement of BR into the adder. In the usual notation this function may more correctly be written as

AD BR– EN: AD BR–

where some convention (such as brackets) should be used to indicate that the term on the left is not simply a possible condition, but rather is bound to be true because of the current time state. Since each such gate name includes the name of the flipflop being set, it seems reasonable to reduce the tremendous redundance that would otherwise result and simply write the gate name. The situation exactly as stated here holds for the adders and the registers made up of latches (IR, AB, MA). For the other registers, the gate is attached directly to the register input gating; hence the action implied by a term such as BR AR EN is that it causes the transfer BR from AR, *ie* AR is loaded into BR.

In many cases gating levels are generated through several gating stages (indicated by right arrows for implication), rather than being derived immediately from the time state. The left part of Figure 4-3 shows the way the flow chart is drawn for loading the flags and PC into AR, AR into BR, and BR into AD at ET2 in a JSP. Note that AR FLAGS EN is an intermediate level in the generation of AR PC EN as well as being the gate for a final result. The right part of the figure shows the actual meaning of the flow chart in terms of level generation during the time state and specific events at the end. The important element for the reader to realize is that the actual events are implied by the gate names, and the gates or strings of gating levels in the box are a recap of the level transitions during the time state, rather than being generated by the clock.

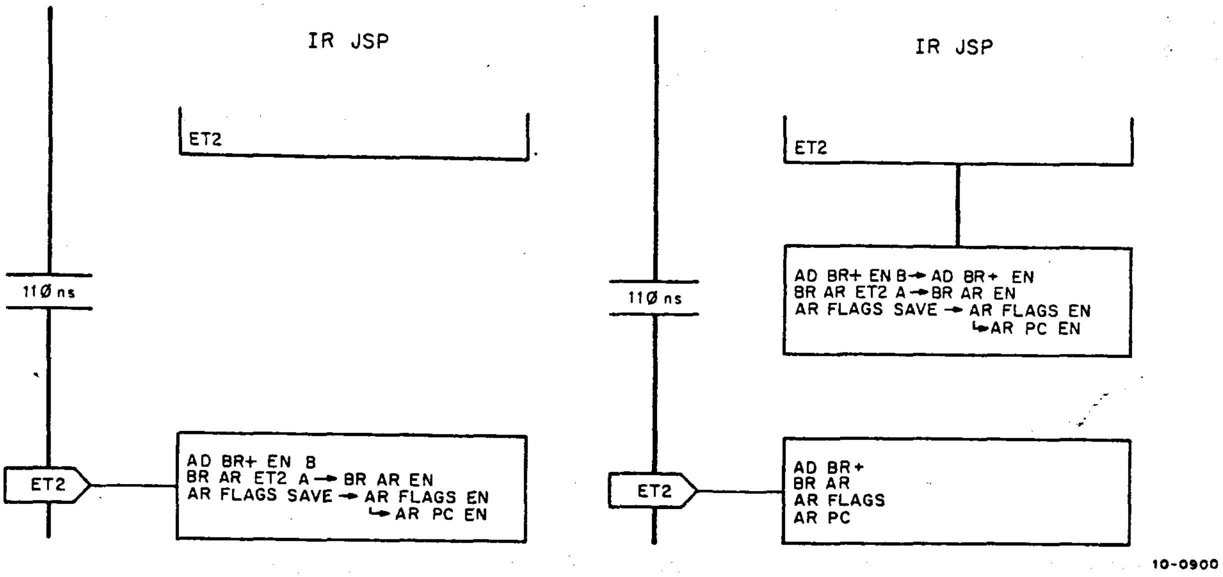


Figure 4-3 Flow Chart Events

4.3.1 Terminology

The terminology used in the flow charts is as consistent as possible with the signal notation as it appears on the logic drawings. When a flipflop *X* is set or cleared by a gate output applied to the direct set or clear terminal, the event is indicated by *X* SET or *X* CLEAR, even though there are not such signals on the prints. If the signal *X* SET is used in the logic drawings and is not exactly equivalent to the event implied by that name, then the name of the flipflop is given alone and shall be taken to mean that the flipflop is set. In a few cases there are signals whose names include the word SET where it does not strictly apply; *eg* the signal PFF1 SET is not applied to the PFF1 set input, but rather enables the data input. On the flow chart PFF1 SET is shown as generating (among other things) PFF1 EN and its true nature is therefore evident. If a flag F is held on by the negation of a clear signal with a name of the form F CLR, the appearance of that signal name means the flag is cleared. But if the data input to a flag is disabled by the occurrence of some time state, and there is no clear signal, then the event is represented by the expression "Clear F". All real signal names and unambiguous pseudo signal names (such as *X* SET) are written on the charts in capital letters. Lower case is used for commentary and descriptive material, and also to represent events for which there are neither valid signal names nor unambiguous imitations.

The final gate for an event *X* is generally regarded as *X* EN even though that term may not appear on the prints (because it is not used elsewhere), and even though it may feed into a group of logic elements to handle different parts of a register (*eg* AD AR+ EN actually enables six AD AR+ flipflops for the adder, and AR AD EN actually drives four lines to the AR gates, two for each half). Very common events are written without the EN (in other words the name of the event itself is used) provided no intervening levels need be indicated. This is true of CLK INH, CLK LONG CYCLE, AD CRY 36, often AD ADD, and the enabling of all time state flipflops (FT6, ET2, ST3).

Where a number of logically different signals with very similar names converge to produce the same event, the exact names are given in the charts so that the reader can follow back through the net to the particular inputs he is interested in. On the other hand where a number of logically identical signals have various letters appended to differentiate them electrically, only the basic signal name is listed, as the reader can easily determine which is the source by checking the inputs at the gate under consideration. The only liberty taken with input signals that are not identical is when a complicated logic function has a very obvious reduction that can easily be matched with the net that produces it. Hence if a logic net generates the function

$$A \wedge [(B(0) \wedge C(0)) \vee (B(1) \wedge C(1))]$$

it is represented in the flow charts simply as $A \wedge (B = C)$.

The final gating level is not even included if it is generated by a prior level that has exactly the same name except for the addition of a letter on the end; hence AR AD EN G shall be taken as equivalent to AR AD EN G → AR AD EN. If one intermediate level generates another whose name differs only in a letter following the EN, then only the final letter is repeated; the expression AD AR+ EN J → A is written in place of AD AR+ EN J → AD AR+ EN A.

The switches are basic gates, such as SCE and FCE, that are available for use throughout an instruction, whereas the terms used to represent actions in the boxes are generally true only in particular time states. Between these are some levels that are true throughout an instruction but for use only at a particular time state. An example is AR AD BR (ET2) B, which is true throughout the jumps and other instructions, and continuously generates AR AD ET2 EN, which at ET2 generates AR AD EN A. These levels are written in the boxes as though generated at the particular time state, but they can be recognized because the time state always appears in the name, usually but not always in parentheses (FETCH or something similar is generally used in signals for the F CYC ACT EN time state).

The main clock is applied through ungated drivers to all of the flipflops for the overall control of the system but is gated to supply separate clocks for the various registers (AR, MQ, etc) so that these will change state only when transfers are made. The AD clock is generated at every main clock except when inhibited, and the flows show only the generation of the inhibit (in other words the nonoccurrence of the event is indicated rather than the event). The other clocks are triggered only to clear a register or when gating levels are generated for the registers; *eg* BR AR EN not only produces the gating levels for the left and right halves of BR but also gates the driver to produce the BR clock. Since a register clock must always be produced for any register action to take place, it is listed in the flows only when it is produced by some gate different from the gating level for the register or is produced separately to clear the register (with or without a conditional gating level that would, if generated, override the clear). The enable for the PC clock is always indicated as it is also the sole input enabling level for the register. The adder produces an equivalence function if the EQV input is asserted alone, and produces a sum if both the EQV and ADD inputs are asserted. For an equivalence the flows show the setting of AD EQV, but for add only AD ADD is shown and it is assumed that AD EQV is always generated with it.

In general all events produced by a signal are listed in detail every time the signal occurs. In a few cases a signal of considerable importance occurs frequently at different places in the charts, often producing a large number of events, and for which nothing is shown. The processor can call for the prefetch of the next instruction either by generating MC INST FETCH EN or by setting MC INST FETCH NEXT; for the events surrounding these actions the reader must refer to the flow chart of the memory subroutine call. Each instruction indicates that it is finished by generating CLK EN INST DONE, which produces the actions necessary to terminate the instruction and set up the conditions for the next; the complete ramifications of this are shown at the entry to the instruction cycle flow.

CHAPTER 5
CONTROL

This chapter describes those parts of the processor logic that exercise fundamental control over the machine mode, the sequencing of the program, the decoding, timing and execution of instructions, and the initiation and termination of processor operations from the console. The basic operating cycle – the "main sequence" – of the processor is the sequence of time states that controls the execution of a single instruction, from receipt of the instruction from memory to the processor indication that the instruction is done. In some cases a part of the sequence may be repeated by a single instruction or may be used by a noninstruction type operation. When this sequence performs a trap instruction, the processor is regarded as being in a trap cycle; similarly an interupt instruction is executed in a PI cycle. Operations associated with the console keys are carried out in a key cycle, which may be just a few key time states or may include an entire main sequence.

The timing of any string of time states is determined by a clock that has four different periods, in order to vary the lengths of the states depending upon the operations that must be performed within them. At various points within a string (depending upon instruction requirements), the processor can call for a memory subroutine; at such times the clock stops and holds the present time state until it is restarted from memory control. The memory call may be to fetch or store an operand, but it may be simply for a page check to determine – before memory, PC, etc. are changed – whether a specified location is alright for subsequent storage of a result. At the completion of each instruction the processor also stops the clock to wait until the next instruction is received from memory.

The time states that make up the main sequence are divided into groups called "cycles", which are not to be confused with the trap, PI and key cycles defined above. Basically the main sequence for performing any instruction comprises four cycles: instruction, fetch, execute and store. In the instruction cycle the processor decodes the instruction and calculates the effective address, repeating the cycle as needed for each level of indirect addressing. The fetch cycle fetches the necessary operands. The processor then proceeds with the execution of the specific arithmetic, logical, transfer or control operations called for by the instruction. For many of the simpler instructions the execute cycle suffices, but in many cases the processor goes from the execute cycle into a special sequence of time states for the given instruction. At the completion of the execute operations, whether in the execute cycle or in a special sequence, the processor goes on to the store cycle if there is a result to be stored.

If a page failure occurs in a memory subroutine there is no return to the time state being held. Instead the present time state is cleared, and the processor performs a page fail cycle, through which it starts a new main sequence to respond to the failure. The term "processor cycles" refers collectively to the page fail cycle and the four cycles that make up the main sequence.

5.1 BASIC CONTROL ELEMENTS

Before investigating the processor cycles it is advisable that the reader understand the fast memory (described in the next section) and the circuits that control certain basic functions, namely processor timing, instruction decoding, program sequencing, and the movement of addresses via the address bus.

5.1.1 Clock

Figure 5-1 is a simplified block diagram of the clock, whose detailed logic and flow are shown in BS and FD drawings CLK. Initially the clock must be started from the console by KEY CLK START. This triggers the main clock, which in turn triggers a number of clock drives. The ungated drivers are shown on the clock drawing; these supply the clocks for the time state flipflops and most other control flipflops and flags. The gated drivers produce clocks for the registers and the adder logic; although included in the clock flow chart, these drivers are shown on the prints for the logic with which they are associated. Note that all of the register drivers produce clocks only when their enabling levels are asserted, whereas the clock for the adder logic is always triggered unless inhibited.

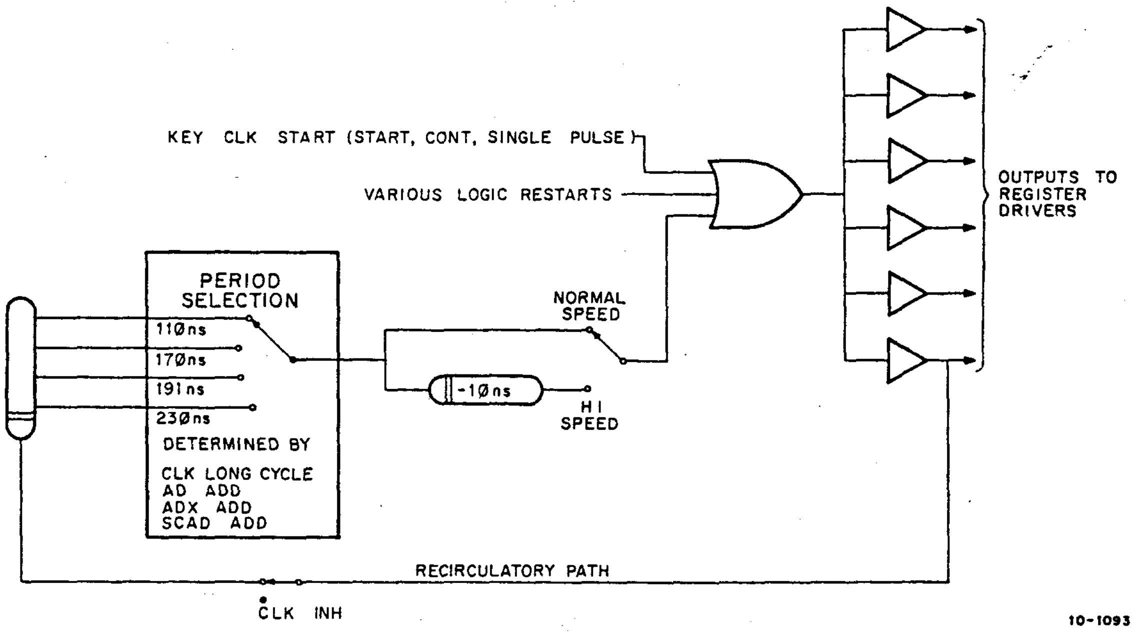


Figure 5-1 Clock

CLK A1 produces a minimum delayed clock through a delay that is adjusted for a 110 ns loop for processor operation at normal speed; for checking speed margins the program can switch to a high speed delay that is adjusted for a 100 ns loop. At this point in the loop, events triggered by the clock drives have settled so that the loop period can be determined. The clock loop may be inhibited altogether because the clock is stopping for a memory operation. If the clock is not inhibited and the time state does not call for an add function or a long cycle, then the minimum loop pulse returns to the beginning of the clock loop. Otherwise, even though the clock is not inhibited, other gates enter into the flow paths to extend the loop to the length necessary for the operations being performed. A time state that requires an extended add function but for which no long cycle has been requested requires a loop of 191 ns. A long cycle or an add function that is not extended requires a loop of 170 ns, but if there is any add function and a long cycle has been requested as well, then the loop is lengthened to 230 ns.

In other flow charts the clock loop is never shown in the detail discussed here. Between clocks the flow charts show only the required period or the possible periods depending upon conditions specific to the given time state, even though the conditions necessary to determine the period are not available until some point within the time state (the period-determining conditions are checked following the minimum clock delay).

The clock loop output retriggers the loop at the beginning unless the processor is in single pulse operation, in which case there is only one clock at a time, each triggered independently from the console. The clock provides synchronization for single pulsing by setting and clearing a synchronizing flipflop for each restart: either return path to the beginning of the loop sets KEY CLK RDY SYNC, and halfway through the minimum loop CLK DRIVE D1 clears it (this logic is shown in the lower left corner of drawing KEY2).

In normal operation the clock loop produces a string of clock drives with variable spacing depending upon the requirements of each time state. This continues until the CLK INH flipflop is set, at which time the clock stops, and it does not start again until the loop is retriggered by one of the conditions shown at the top of the flow chart. The left three entries are for single synchronous operations, *ie* for fetching or storing a one-word operand. When an operand is being fetched, the processor waits until it is available, at which time MCRST1 retriggers the clock on the condition that the operation is a synchronous read and is not a page refill (of course a page failure may prevent the return of MCRST1 at all). If the instruction requires storage of a result without fetching an operand, the processor waits for a page check before taking any irreversible actions; at the end of the delay for page checking, memory control generates a pulse that restarts the clock if the page is alright, the memory request is synchronous (*eg* it is not for fetching an instruction) and the same request is not also for reading. If the clock is restarted by a paging return, then sometime later in the instruction it will again be stopped for writing, following which it is restarted by MCRST0. The next entry from the left is for the double precision floating point instructions (including double moves) that require the fetching of two words from memory. After the high order word is fetched, the processor restarts from the regular operand return to go ahead with various operations, such as adjusting for the signs of the operands and manipulating the exponents (which are determined of course solely by the high order words). Upon reaching the point at which the double length operand is needed, the processor again stops and waits until the low order word is available, at which time the FCE2 synchronization restarts the clock.

The next synchronization entry is for restart following the completion of an instruction when the next instruction has been fetched and is ready for execution. The remaining two entries are for circumstances that take the processor out of the normal flow sequence. These are diversion to a PI cycle when an interrupt is ready following completion of an instruction, and restarting the processor in a page fail cycle. The clock is always at rest when a page failure occurs as this condition is determined within the memory subroutine; diversion to a PI cycle can occur under conditions other than that mentioned above, but at such times the clock is going.

The right and left halves of print CLKC show the logic for stopping the clock and requesting a long cycle (the actual period being determined both by the state of CLK LONG CYCLE and the add functions specified by the arithmetic logic). The conditions that enable setting the long cycle flipflop are all quite straightforward and are listed at the appropriate places in the flow charts. Among the conditions that set CLK INH, note that CLK MISC INH is anded with a condition that contradicts one of the conditions that produces it; in other words DFDT8 requires the other events produced by CLK MISC INH without stopping the clock. The bottom enabling gate for CLK INH involves the two clock levels generated by the nets in the lower part of the drawing. CLK EN INST DONE usually represents the completion of an instruction; it stops the clock unless PI OV is set, indicating that the processor is to enter the second of a pair of PI cycles, or CLK PSEUDO INST DONE is asserted. This last signal is generated in two time states in which some but not all of the events associated with the completion of an instruction must occur. CLK EN INST DONE arises at ST2 only as the result of a page failure and the processor must continue; at PIT3 the pseudo. signal prevents some of the events that would otherwise be produced by CLK EN INST DONE, but the clock is stopped anyway by CLK MISC INH.

5.1.2 Instruction Decoding

Drawing IR shows the latches that make up the instruction register; this register receives the left twelve bits of each instruction word from MB. IR holds the instruction code and AC address or the IO instruction code and device code throughout the entire execution of a given instruction. The register does not change during an instruction except in the transition from a block IO instruction to a data IO instruction. After storing the incremented pointer, a BLKX returns to the fetch cycle, and thus FT3 or FT5 in an IOT sets IR 12 to convert to the instruction code of the corresponding data IOT. The only other manipulation of the latches is in read in, where the register is set up with 1s held continuously in the left three bits. During the first cycle, IR 12 is also held set so that read in begins with a DATAI to bring in the pointer, then continues with a string of BLKIs.

The logic that controls IR is shown in Figure 5-2 and at the top of print IRMB. In program operation the completion of each instruction (or simulation of this event by an initiating key function) enables IR MB, and at IT0 IR is latched. (The inhibition of IR MB EN by BYF5–6 SET holds IR over into the second part of a byte instruction.) Following IT0 the processor may go to either IT1 or IT2 – the latter being for a trap cycle. Thus as soon as the processor enters IT1, the IR MB CLR disables IR MB EN, whose negation in turn keeps IR latched. However should a trap intervene so that the processor instead enters IT2, IR IR is cleared and the latches run free again until the processor restarts at IT0 for the trap cycle. The rest of the gates (for the key logic) are redundant.

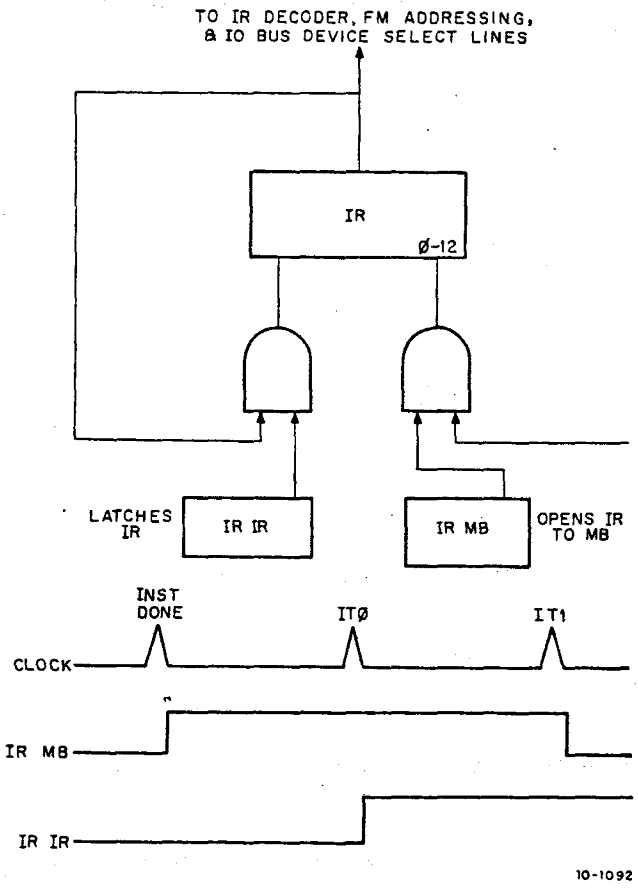


Figure 5-2 IR Control

The remaining IR drawings show the decoding of the IR contents into instruction control levels. The decoding begins at the upper left of IR1, where the left three bits are decoded into eight primary levels provided the processor is not in a deposit cycle or a page fail cycle is not in KT3 (to prevent the possibility of decoding a word examined from the console), and is not performing one of the special PI functions (the instruction executed in a normal or dispatch interrupt is decoded like any other). Where one of these first levels represents a single class of instructions, further decoding is in various arrangements of logic gates, but levels that represent a variety of instructions enable additional decoders for bits 3–5, and some of the outputs at this level enable decoders for bits 6–8. Four gates at the lower right in IR1 decode bits 7 and 8 for the modes, although the levels generated are used for other purposes as well. In the upper left of IR2 is a pair of decoders for bits 3–6, where the octal digits in the output signals correspond to the numbers of the Boolean functions (*ie* the left digit is 0 or 1 for bit 3, the right digit is bits 4–6).

The numbers in brackets show the corresponding instruction codes. Although some other use is made of these levels, they appear primarily in the center of IR2 for specifying groups of Boolean instructions that have common attributes. IOT decoding is mostly at the lower right in IR2. IR3 shows most of the shift and data transmission decoding in its left half, the double precision instructions in the upper center and right, and the decoding for LUUO, MUUO and various illegal and unused codes in the lower right.

These three prints do not of course show all of the control levels generated from IR to control individual instruction situations. The test instructions for example are represented only by the level IR TEST decoded from bits 0–2, and the rest of the decoding is shown on the TEST drawing. The beginning of each instruction flow chart lists all of the decoding and control levels for the instruction, including those on the IR prints and elsewhere.

In order to make it easier to see the IR decoding in terms of logical relationships, flow diagram IRD shows all of the logic of the IR prints in a decoding tree; here individual bits generally appear in order from left to right, and the values of bits or groups of bits are in order from top to bottom. In other words the decoding of bits 0–2 is represented by the eight terms in the left column, and the decoding spreads out to the right from each of these through standard decoders and logic nets in a variety of configurations. In most cases groups of bits are decoded to produce a number of signals, but in some cases sets of signals are anded to produce common functions, as shown respectively by lines diverging and converging toward the right.

Most of the decoding is quite straightforward and in any event is further clarified on the flow charts. Something should be said however about the decoding shown in the lower right of IR3. Codes above 110 that are not used generate the signal IR UNUSED (these appear on the decoding tree between IR 1XX and IR 2XX). The illegal instructions are a JEN or HALT in user or supervisor mode, or an IOT given when the processor is not in a PI cycle and USER IOTS ILLEGAL is set (which indicates the processor is in supervisor mode or is in user mode with USER IOT clear). The actual decoder output IR JRST is generated when IR contains 254 and it is not illegal; similarly IR IOT is produced by anding IR 7XX (representing any IOT code) with ~IR ILLEGAL INST. The signal IR MUUO EN is derived from the standard MUUO codes, and IR MUUO is the combination of this, IR 10X and the illegal and unused codes.

5.1.3 Program Counting

The mechanism for specifying consecutive addresses for the retrieval of instructions in the program is shown in drawing PC. Note that in spite of the name, PC is not a counter – it is simply a register whose outputs are applied to +1 gates, which in turn generate a set of PC+1 outputs. The gate outputs represent a number one greater than PC or simply represent PC depending upon whether or not the +1 enable input is asserted. The enabling input is applied to the low order end of the gate for the low order half of the register; the gate for the high order half also has an enabling input, but it is just the carry out of the low order gate. The signal that enables the gates is the 0 state of an inhibit flipflop, so that the PC+1 outputs supply the incremented address unless the inhibit flipflop is set. Thus the address of either the present instruction or the next is readily available simply by manipulating PC+1 INH.

When the processor is ready to fetch an instruction in sequence, the incremented address is supplied from the PC+1 gates to the address bus, which in turn supplies it back to the PC register. Thus program counting is effected by the loop of PC, the PC+1 gates, the address bus and back to PC (Figure 5-3). When a skip condition is satisfied, this loop is used to advance PC during the instruction. Then when the processor goes for the next instruction PC already points to the next location, and PC+1 therefore points to the location two beyond the current one.

The PC+1 outputs are available both to the address bus and to AR for saving a return address in a jump or MUUO. Generally an address saved should be for a return to the next instruction, *ie* the instruction that would have been performed had the jump not occurred. On the other hand suppose an instruction is terminated because of a page failure or there is an interrupt between the two parts of a byte instruction. Then the current address must be saved for a later return to the beginning of the interrupted instruction. New addresses are always supplied to PC via the address bus regardless of the point of origin.

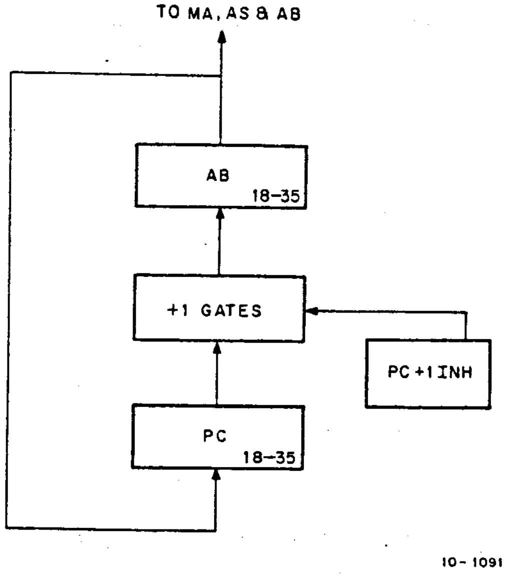


Figure 5-3 Program Count Loop

Drawing PCC shows the PC clock and the inhibit flipflop at the right, and the net for determining the jump condition in a JFCL at the left. The second net from the top in the center of the drawing generates PC CHANGE, which prevents automatic fetching of the next instruction during any instruction in which PC might be altered. The remaining logic is for the enabling of the PC clock. All of this is straightforward except for the complicated network in the very center of the drawing. The two enable outputs of this network are anded below to enable the clock. The output of the large OR gate in the left half of the network asserts PC CLK (ET2) EN C (provided the appropriate input conditions are satisfied) and at ET2 also asserts the B output level. For the remaining inputs to the AND gates at the right, the B level is automatically asserted at ET2 in the skips or CAXX, but the C level is asserted only if the appropriate ADZ condition is satisfied. The condition is P or R in the skips, P or Q in CAXX.

The PC clock is also controlled directly from the memory subroutine through the gating just at the left of the register driver. The clock reloads PC from the address bus at the completion of the page check for any instruction fetch except an instruction being executed by an XCT. After the address is latched into MA, MCTO clears PC+1 INH just in case the address had been taken from PC without incrementing. Note that the address being loaded into PC may be from the counting loop whether incremented or not, or it may be a completely new address originating from AD, as in a jump, or from AS at the initiation of operations from the console.

5.1.4 Address Bus

Addresses are supplied to the address bus through the eighteen latches shown in print AB. This register can receive addresses from the PC+1 gates, the AS+1 gates associated with the address switches, and the right half of the adder.

In the upper left of drawing ABC are nets that decode for 0s in bits 19–21 and bits 18–31 to check respectively for a proper small user address and a fast memory reference (for information on the AS input to the latter net, refer to the discussion of the address switches in section 5.6). The rest of the drawing shows the circuits that control the bus, including four flags; the right three of these enable the various address inputs, and the left one latches the bus. When one input flag is set, the others are cleared through the net at the lower left so that only one source at a time can supply an address. Memory signals that set AB AD and AB PC directly occur at times when it is known that the other flags are clear. All of the enabling levels for input from PC cause AB PC to be set except that AB PC (FETCH) EN does not enable the flag if it is being generated by an IOT during read in. AB AD EN sets AB AD except when it is being generated by AB AS EN; this latter signal enables AB AS but also generates AB AD EN to produce the clear for the other flags.

A selected input flag remains set until some other is selected or the logic at the left sets AB AB to latch the bus. The 1 state of this flag clears the input flags through a delay (D5) that allows enough time for the bus to be properly latched, as shown in the timing diagram in Figure 5-4. Because of the nature of the register clocks, AS and PC remain stable during latching. The adder however changes at every clock unless either the adder enabling levels remain constant or the AD clock is inhibited. To hold the adder steady during latching, any condition that generates AB AB EN also generates AB AD CLK INH when the adder is supplying the address (AB AD(1)). Note that there is one condition, IR 3XX at ET0, that generates the inhibit but has no connection with the AB logic. This holds the adder steady through two time states in the test instructions to allow enough time to set up the test conditions.

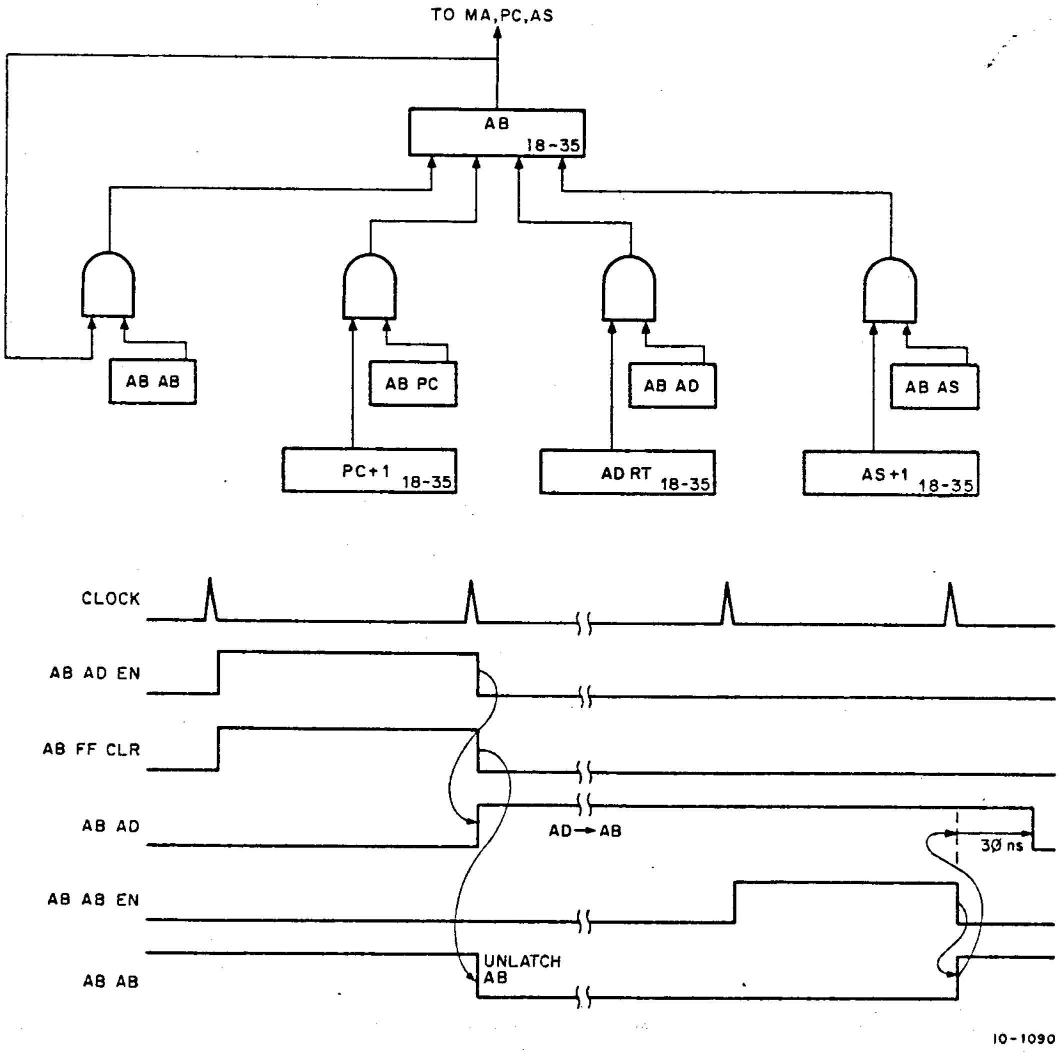


Figure 5-4 Address Bus Timing

5.2 FAST MEMORY

Although fast memory can be called from memory control, *ie* the program can address fast memory for a memory operand, it is easier to view the fast memory as part of the processor control logic rather than part of the memory logic. Drawing FMR shows the four groups of fast memory registers, each group comprising three packages that together provide storage for sixteen 36-bit words. Each group has a selection level and a write level, and all receive the four address lines. These lines select one register in the selected group, and if the write level is off, the contents of the selected register are available at the outputs and are supplied to the processor register gating through the buffers shown in FMB. When the write level is asserted, data supplied through the mixer in drawing FMIN is written into the selected register. The word written may be supplied by AR or BR depending upon the state of the SAC BR flag in the store logic. Figure 5-5 is a simplified diagram of fast memory addressing and control.

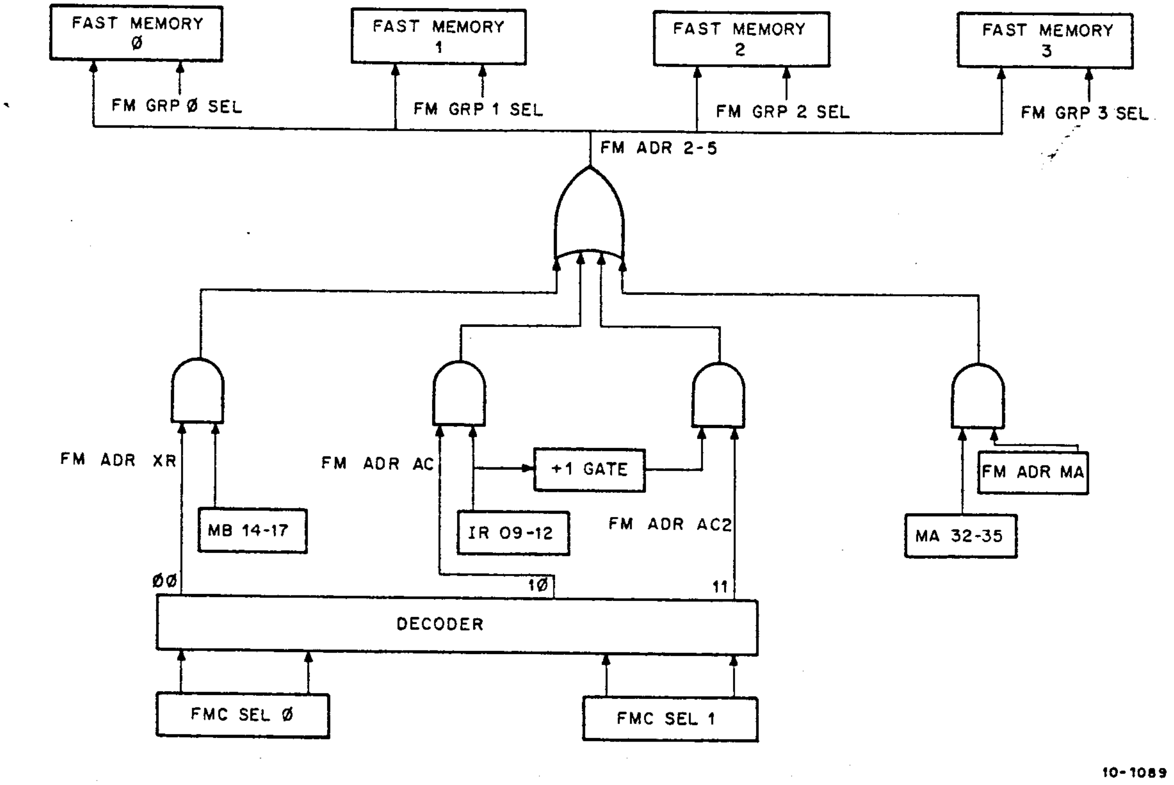


Figure 5-5 Fast Memory Control

Fast memory addressing is determined by the logic shown in drawing FMA. Selection of the group is determined by the nets and flipflops in the top and bottom parts of the left half of the drawing. Essentially there are three ways the group can be selected. If FM ADR EN is asserted, the group is selected according to the FM ADR bits, which are loaded by the same instruction that supplies the base addresses for the process tables. If FM BLOCK EN is asserted, group selection is determined by the settings of the FM block switches on the console. If neither enabling level is true, then group 0 is selected, as is usually the case in executive mode. (Note that what the programmer refers to as an "AC block" is referred to in the hardware as an "FM group", for the term "block" is applied to the flipflops associated with the switches.)

The block switch enabling level is true whenever the FM manual switch is on and also when both paging switches are off during a key function that calls for a memory operation. Selection according to the program-selected group is controlled by the net at bottom center. If the switches are not enabled, then FM ADR EN is true while user paging is in effect except during an executive XCT. However a memory access with user paging sets FMA USER HOLD, so that if FM ADR MA is subsequently set, FM ADR EN is asserted anyway. The latter flipflop is set only for a true fast memory reference for a memory operand, *ie* it is not set when an executive XCT is calling an accumulator or is accessing the shadow area or the AC stack.

The string of gates across the center of the drawing generates the fast memory address bits from various sources depending upon the type of access. The address is supplied from MA when the fast memory is called for a memory operand reference. An index register or AC reference uses an address supplied by MB bits 14–17 or IR bits 9–12 respectively. A reference for a second accumulator uses an address one greater than the IR AC address as supplied through the +1 gate at the lower right.

Fast Memory Control. The net at B7 in drawing FMC generates the address selection levels for index register and accumulator access according to the states of the FMC SEL flags provided FM ADR MA is clear. The respective states of the selection bits for addressing fast memory as XR, AC and AC2 are 00, 10 and 11. At the beginning of an instruction both bits are 0, and a fast memory location can be referenced only as an index register. After completion of the effective address calculation, FMC SEL 0 is set at the earliest possible time without waiting until AC access is needed; this is in the instruction cycle itself if there is no indexing in the last step of the address calculation, at some time in the fetch cycle, or even within the memory operand fetch. The reason for this is that it takes time to set up the address, select the fast memory register, and get its contents onto the output lines; it is best therefore to get the AC selection set up as early as possible even if it turns out no AC reference is necessary.

Some of the signals in the left half of the print indicate an AC reference and others indicate an AC2 reference. All AC2 signals generate FMC AC2 EN, but all signals of either type generate FMC AC   AC2 EN. The latter signal sets FMC SEL 0, which must be 1 for either type of reference, whereas the former sets FMC SEL 1. AC is always fetched before AC2. If later in the instruction the combined signal should be true without the AC2 signal, then bit 1 is cleared; and if FMC AC2 EN comes back on, the right bit is set again. The whole point of the procedure is this: during an effective address calculation, fast memory can be referenced only for an index register, and the next fast memory reference in the instruction (not counting possible memory operand references) must be AC. The selection bits are left in the 10 configuration unless AC2 must be used. If two accumulators are fetched and two are also to be stored, then bit 1 is cleared and later set again. In any event the completion of the instruction clears both bits so they are ready for indexing in the next instruction.

The remaining logic in FMC controls writing and memory operand access. Any condition that requires writing in a fast memory location enables FM WR, which generates a write level for the selected group. The write level must turn off before there is any change in the input, so FM WR(1) clears itself through a delay adjusted for the timing listed on the drawing. The other condition applied to the direct clear input of FM WR indicates that an illegal entry has been made to concealed mode (section 5.5); it thus prevents any change in the accumulators before the next memory access, at which time there will be a page failure.

A memory reference to fast memory is handled by means of FM ADR MA, which is set and cleared from memory control for an instruction or operand fetch (these are treated in the memory description). On the other hand memory operand storage in fast memory is handled directly by the store cycle. The reason for this is that the paging determines the storage conditions before the instruction is executed; hence in the store cycle the processor already knows that the coming reference is to fast memory and it saves considerable time by handling the storage directly instead of restarting the memory subroutine. If MC STORE IN AC is set, the store cycle first sets MA ADR MA to select the fast memory register from MA, then sets FM WR and FMC STORE IN AC. With the latter flipflop set, the next clock clears both it and FM ADR MA.

5.3 PROCESSOR CYCLES

Each instruction is performed by the sequence of cycles: instruction, fetch, execute, and if necessary, store. The instruction, fetch and store cycles are presented here in considerable detail, but the description of execute is limited to an overview, as the detailed discussion of instruction execution, including the many special sequences inserted between execute and store, is left to the discussion of the instruction flow charts in Chapter 8. Also discussed here is the page fail cycle through which the processor returns to the main sequence following a page failure in a memory subroutine.

5.3.1 Instruction Cycle

Drawing INST shows the logic that controls the instruction cycle and flow chart IC shows the entries into and the events that make up this cycle. Entry into the cycle requires two independent events that must be synchronized. These are that the next instruction fetched from memory must be ready and that the current instruction must be completed (refer back to Figure 2-1). In normal operation, instruction fetching is overlapped so that which event occurs first and how long a synchronization wait is required depends entirely upon the present instruction and the circumstances surrounding its execution. Initially both events must be produced by action at the console.

Memory control indicates that it has fetched an instruction by returning MCRST1 to set INST RDY on the condition that an instruction fetch had been requested, and the memory subroutine just performed was not for a synchronous read or a page refill (either of which may be going on when an overlapped instruction fetch is called). The byte entry simulates instruction ready to begin the second part of a byte instruction by performing an effective address calculation on the pointer. The final ready entry is for key functions – not to be confused with a true memory instruction fetch initiated from the console; in read in the key logic sets up one BLKI after another, whereas in execute the main sequence is taken to execute an instruction supplied by the data switches. Setting INST RDY clears MR RESTART B just in case the processor is restarting automatically following a power failure or a maintenance timeout.

The completion of an instruction is indicated by the generation of CLK EN INST DONE without the simultaneous generation of CLK PSEUDO INST DONE. These two signals are generated together following the storage of a page fail word, at which time the processor goes to IT2 to start the page fail trap cycle. CLK EN INST DONE clears various flags that may have controlled the preceding instruction and clears the FMC select bits to prepare for possible use of an index register. (The other clocks in the instruction cycle also clear various flags in preparation for performing a new instruction, but none of these is mentioned in the text unless it is of exceptional significance.) If the processor has just completed the first part of a two-part byte instruction, BYF5 and BYF6 are set; otherwise IR is unlatched and allowed to follow MB. If PI OV is set the processor is just completing an interrupt instruction that overflowed, so it proceeds directly to the PICT0 time state to enter a second PI cycle. But if PI OV is clear, PI CYC is clear and the clock stops. INST DONE sets if the processor is not entering a PI cycle or there is a pseudo instruction fetch. (The latter condition at PIT3 indicates a normal or dispatch interrupt, which is treated like the beginning of an instruction; other PI operations go directly to the fetch cycle. In any event PIT3 stops the clock.)

INST RDY(1) and INST DONE(1) both enter delays that are adjusted for a minimum wait before the next instruction begins. The turnon of INST DONE DELAYED also triggers a pulse that diverts the processor to a PI or key cycle if PI CYC RDY is set (PI CYC RDY being 1 at this time implies that no instruction has been prefetched). If no diversion is ready, the coincidence of the two delayed signals produces the IT0 time state and restarts the clock. (Note that in spite of the way the time state signal is represented, there is no IT0 flipflop. IT0(1) is a gate-generated level, and its negation is represented by IT0(0).) In following through the IT0 and IT1 flow the reader will notice that a few events appear to make no sense in terms of handling an instruction word. The reason for this is that indirect addressing causes the cycle to be repeated to handle additional address words in an effective address calculation. The questionable events involve the left half of the word, and they are meaningful only on the final iteration of the loop when the left half may contain information for restoring the flags.

The clock that terminates IT0 latches the instruction code into IR and enables the address part of the instruction into the adder, and through it onto the address bus. If the address calculation is already complete, FMC SEL 0 is set for possible subsequent accumulator reference. If there is no indexing MB left is also enabled into the adder for possible flag restoration. On the other hand, if instruction bits 14–17 do specify an index register, the right half of the selected fast memory location is added to the address part of the word, and the left half of the index register is added into the adder left half in case it contains wanted information. The apparent incrementing of the left half actually produces a simple load because of the way the adder gating works, as explained in section 7.1. XCTP ACTIVE being set indicates that the previous instruction was executed by an executive (paged) XCT, so on this condition the processor clears the flags that controlled that execution. Finally if the single pulse switch has been turned on while the processor was running, single pulse operation begins with IT0.

Regardless of the path the processor follows from IT0, the entry into the next time state clears INST RDY and disables the IT0 state. If the processor is to start a trap cycle it goes to IT2 for operations described in section 5.4. But the processor continues to IT1 if it is already in a trap cycle or a PI cycle, or if no trap flag is set. At this point the current instruction has been decoded, so instruction conditions appear in some of the gating. The effective address is transferred to AR right from the adder, and in a JRST the left half of the adder is loaded into AR left. IR MB is cleared so that MB can be changed without affecting IR. If the instruction generates SAC BR, the flag that selects the source for AC storage is set for use in the store cycle. A HALT sets KEY PROG STOP and clears KEY RUN. If the instruction is a JRST 1 taken from a concealed location, the last instruction public flag is cleared, but if the instruction (whatever it is) is not taken from a concealed location, that flag is set. MQ is cleared unless it may already contain information that will be needed later. Hence the clear is inhibited by BYF5 being 1, indicating that the processor is starting the second part of a byte instruction with the mask in MQ; and it is also inhibited for a skip or an IOT, in case the instruction is being executed by a dispatch interrupt, where MQ contains an interrupt address that may be needed for a second PI cycle.

If MB 13 is 1, and the indirect is not to be diverted, the processor stops the clock and generates a pseudo instruction fetch to get an address word from memory. Thus for effective address calculation the return to the beginning of the loop is made through the memory subroutine. When INST RDY DLYD comes on for the address word, synchronization is immediate as INST DONE remains set. If an indirect loop is to be diverted, the processor goes to the PI cycle or the key cycle depending on the state of PI RDY SYNC.

If the effective address calculation is complete, as indicated by MB 13 being 0, IT1(1) generates F CYC START so that the processor starts the fetch cycle. Note that this does not mean that the processor proceeds to the fetch cycle with the fetch start events occurring at the next clock. Rather it means that if the present address is direct, the IT1 and F CYC START time states occur simultaneously and the same clock produces the events for both.

5.3.2 Fetch Cycle

The initial switch FCE is generated by all of those instructions that require fetching a single operand from location E, as shown in the upper left of drawing F1. A subset among these conditions also generates FCE PSE to call for a read-modify-write memory subroutine, *ie* one in which memory control fetches an operand and then holds the memory for later storage of a result in the same location. Instructions that read and write in E but are too long to allow the processor to hang onto memory generate separate FCE and SCE switches (for the latter refer to the store cycle below). The signal F MEM REF (at the lower left) represents any instruction that requires a memory reference during the fetch cycle; this includes all instructions that generate FCE or SCE, and also those that require a two-word memory operand or the fetching of a word from a location specified by the left or right half of AC.

Drawing F2 shows the flipflops that specify the fetch time states, but there is not a one-to-one correspondence between these flipflops and the states. The first fetch time state is F CYC START, which is simultaneous with IT1 when the effective address calculation is complete. The large net in the center of F1 generates the final fetch time state F CYC ACT EN, in which the processor actually begins to perform the various actions required for the execution of the instruction. This time state is produced by FT6(1) but is also produced by IT1(1) if there is neither a fetch memory reference not indexing in the final step of the address calculation. In other words it is possible for all three time states, IT1, F CYC START and F CYC ACT EN, to occur simultaneously.

The remaining logic on F1 is actually for the synchronization of the execute cycle with the fetch of the low order word in double precision operations. At ET2 these instructions set FCE2 WAIT, and the return from memory control of the second operand sets FCE2 SYNC. The synchronization of the delayed outputs of these two flipflops restarts the clock.

Besides SCE there is another initial store switch that affects the fetch cycle. This is STORE, which indicates that the instruction will store a word in a location other than E. This switch delays the automatic overlapping of the next instruction fetch, because later in the current instruction there will have to be a page check after the storage address is determined.

The procedure for fetching the operands necessary for an instruction is shown in flow chart F. As the cycle starts, AB is receiving the effective address from AD, and AD left contains either zero or restore bits for the flags. The first clock in the cycle moves E or AR right and for a JRST moves the restore bits to AR left (these are IT1 actions). For F CYC START the clock clears INST DONE. It also sets XCTP ACTIVE to control memory operand references if this instruction is being executed by an executive XCT that called for crossing over between the user and executive virtual address spaces (for a byte instruction this action is held off till the second part, as all pointer operations are in executive space). The flow paths in the left half of the chart are relatively unique sequences required for specific instructions; they are included in the flow chart so that it is a complete representation of the fetch cycle, but the detailed description is left for the treatment of the individual instruction flows in Chapter 8. The IR DPOP line triggers a double operand fetch but goes directly to FT7 to increment E so it will be ready for fetching the low order word. The return from the high order fetch restarts the sequence, which continues to ET2 and then stops to wait for the second word.

The flow path at the left is for instructions that must fetch a word from a memory location specified by AC before beginning operations (if any) with the effective address. There are also entries into various parts of this path for returning to the fetch cycle to repeat part of the main sequence. The processor returns from ST2 to FT3 to do the DATAO for a BLKO or the third part of an MUUO. A similar return for a BLKI is to FT5. To begin the second part of an MUUO or a DMOVXM the processor returns to FT6. The bottom of the chart also shows two entries to FT6 from a memory subroutine, in one case from a page check, in the other following an operand fetch. These two conditions are set up by the entry into the PI cycle for PI functions other than a normal or dispatch interrupt.

The flow paths in the right half of the chart are for the great majority of instructions, which fetch neither a double operand nor an operand specified by AC. Of the two main paths, the left is for instructions that either fetch the contents of E or will subsequently store a result in E and thus require a page check. For these the processor latches the address bus, inhibits the AD clock so the bus input cannot change, and stops the main clock. A MAP, CONI or DATAI prohibits the memory subroutine from actually calling memory: in the first case this is done because there will actually be no memory access, and in the other cases the memory subroutine will be restarted later just in case the processor has to wait for the IO bus before getting the word to be stored. If the instruction generates both FCE and SCE because it takes too long for a read-modify-write cycle, MC SPLIT CYC SYNC is set so that the memory subroutine will disconnect from memory following the fetch and wait to be restarted later by the instruction. Normally an FCE PSE subroutine holds memory, but it is also split in single pulse operation, if there is a possibility of a parity or address stop, or if the drum split signal is asserted in any but a skip instruction. If PC cannot change in the current instruction and there will be no storage in a location other than E, F CYC START sets MC INST FETCH START so that the memory subroutine will go directly to the next instruction fetch as soon as it is able.

The time state that follows depends upon the time available for latching the address bus (remember that if the last step of the effective address calculation requires indexing, that indexing is going on through the adder during F CYC START). If an operand is to be fetched or there is no indexing, the processor proceeds to FT6 and calls for a memory read or write or both depending upon the fetch and store requirements. The memory subroutine switches the fast memory from an index register to AC, and the return from the subroutine (with the operand or simply to indicate completion of a page check if there is no fetch) restarts the clock. With indexing but no fetch, the processor calls for a memory write and enters FT8; upon receipt of the return from the page check, the processor first switches the fast memory to AC before proceeding to FT6. Of course the reader should realize that there may be no return at all to the fetch cycle from a memory subroutine, as there is always the possibility of a page failure, in which case the processor clears the time state and goes into the page fail cycle discussed below. In the MAP instruction, however, no real memory access is attempted; if a page failure occurs, PF CLK START causes the processor to continue where it was holding in the fetch cycle instead of going into the page fail cycle.

The final path at the right is for instructions that make no memory reference at all, *eg* instructions that use only immediate operands or perform entirely control functions. If the instruction is an XCT, a JFCL that will jump, or a JRST that neither halts nor restores the flags, then the processor calls for the next instruction fetch. For any other instruction the fast memory address is switched to AC (note that in the three instructions mentioned, IR 09–12 is not used as an address). If the instruction will neither change PC nor store a result in a location other than E, the clock places PC on the bus and sets MC INST FETCH NEXT, causing memory control to set MC INST FETCH START on the next clock. If there is no indexing the processor simultaneously performs the initial execute actions. Otherwise the processor goes to FT6 to allow time for the fast memory address to switch over. The clock also enables AR into the adder so that E will be available from it, as in the FCE and SCE flow paths where the AD clock is inhibited.

5.3.3 Execute Cycle

The logic and flow for the execute cycle are shown in BS and FD drawings E. After the effective address is computed and the necessary operands are fetched, the processor begins the actual execution of the instruction in the F CYC ACT EN time state. From this point the sequence of states in execute is determined by the ET enabling levels generated by the large nets at the left in the logic drawing. Which time states are used depends entirely upon the instruction requirements: from F CYC ACT EN the cycle may go to any time state. From ET0 the cycle may go to either ET1 or ET2, but the chain can be broken here: the condition for ET1 is that ET1 be enabled and that there be no shift count subroutine. In multiplication and byte manipulation, the processor goes to one of the SC time states and returns to ET1 following completion of the subroutine. Note however that for the path to go directly from ET0 to the next ET clock requires that there be neither an SC subroutine nor a non-E store. For such storage ET0 enters the next time state, but the clock stops for a page check and restarts upon the return from memory control.

Regardless of which time states are used, the execute cycle always ends at ET2 except in an IO instruction. If the IO bus is not available at ET1 of an in-out transfer, the processor goes into a loop; when the bus does become available, the processor goes into a special IOT sequence that returns to ET2 from IOTT9. However, during the loop the sequence may be broken entirely by diversion to a PI cycle or key cycle in the same manner as at the end of the instruction cycle. If the fetch cycle set MC MEM GO INH to prevent memory control from actually calling memory, ET2 clears the flag and restarts the write subroutine. From ET2 the processor either enters a special execution sequence for the instruction, goes directly to the store cycle to store the results, or if neither of these is necessary, generates CLK EN INST DONE to indicate that the instruction has been completed and to return to the instruction cycle. Of course most of what happens in the execute cycle is for specific instructions and is not included here as it fills many sheets of instruction flow charts. The remaining operations shown in the execute flow are for a trap cycle, which is discussed in section 5.4.

5.3.4 Store Cycle

Drawing ST2 shows the generation of the initial switches that control the storage of the results of an instruction. SCE represents all those situations that require storage in location E, whereas STORE indicates memory operand storage in some location other than E. Any instruction automatically stores a result in AC unless it generates SAC INH. AC storage is from AR unless the instruction generates SAC BR, which causes IT1 to set the flag shown in the lower part of print ST1; the outputs of this flag control the source of input to fast memory. The flag is also set independently in two floating point situations where it cannot be left on throughout the instruction. If an instruction produces a two-word result, it generates SAC2 for storage of the low order word in a second accumulator.

The remaining nets in ST2 generate signals to control events surrounding the entry into the store cycle. ST1 COND arises in ET2 if some storage function is necessary and entry is not inhibited by ST INH. In general this latter signal (generated in the upper right) indicates that the instruction must go through a special sequence before getting to the store cycle, but sometimes it just inhibits the standard entry when entry by other means is desirable because of special conditions. *Eg* the special sequence for a JFFO may not be needed, and entry is therefore from ET2 in some cases but from the special sequence in others. ST INH disables the standard inhibit for some instructions that call for the next instruction fetch at ST1, and the entry for these instructions is supplied by ST INST FET ST1 EN regardless of what is needed in the way of operand storage. The remaining conditions that generate ST1 COND represent the completion of various special sequences, some of which do double duty by performing other functions, such as clearing AR or MQ.

The store time states are controlled by the remaining logic in drawing ST1 and the storage procedure is shown in flow chart S. Entry in all ST1 COND cases except page failure is from the final execution time state, whose clock handles storage in AC at the same time that it sets ST1. The only regular execution entry not included in the standard condition is from the completion of the normalize sequence for single precision floating point instructions other than in long mode. Entry from PFT3 of the page fail cycle is through a memory subroutine that does a page check for storage of the page fail word. The left block transfer entry is for the BLT loop, *ie* for completing a transfer with subsequent return to the beginning of the loop from ST2. The other BLT entry is for termination of the transfer either because it is complete or it has been stopped by an interrupt or a page failure (the last case prepares for storing the page fail word). Entry from KT3 is for using the store cycle to deposit a word from the console.

By ST1 there can be no further need for the address bus or changing PC, so any instruction that has not already called for the next instruction fetch does so now unless it is going to return to an earlier point in the sequence or continue execution. Also at this time a BLT enables the PC clock to increment the destination address, in an MUUO flags are adjusted to go on to the next part, and SAC BR FF is cleared so any subsequent fast memory storage is from AR. If there is no further storage the instruction terminates with CLK EN INST DONE. For storage in AC2 the FM address is switched to AC2, and if there is no memory storage, the processor goes on to ST4.

The existence of ST2 COND indicates that memory storage is required so the clock sets ST2, enables the parity bit, clears MC MEM GO INH (in case it has been set since ET2), and loads AR into MB so the result is available to the memory bus. Since the page check has been done, it is already known whether storage is to be in core or fast memory, and memory control makes this knowledge available by means of the state of MC STORE IN AC. If this flag is clear the clock stops, and a signal delayed from the turnon of ST2 waits for memory control to indicate that the write cycle is in progress. This synchronization supplies the write restart, and the clock restarts upon the return from memory control. If the cycle has been split but not already restarted or the system is in single pulse operation, the delayed ST2 signal itself restarts the write part of the memory subroutine and then waits for synchronization from memory control.

If operand storage is in fast memory, the store cycle saves time by handling the storage directly instead of going through memory control. (Since memory control does not function at all in this situation, ST1 must inhibit the clock if a memory stop is called for.) The ST1 clock switches the fast memory address to MA. The next clock does not affect FM ADR MA because FMC STORE IN AC is 0, but it does set that flag and FM WR. With FMC STORE IN AC now set, the next clock (regardless of time state) clears FM ADR MA and FMC STORE IN AC, and the transition to 0 of the latter clears MC MA MA.

At ST2 the only store function left is SAC2. If there is a low order word to be put in AC2, the processor must get to ST4; but if the memory operand storage was in fast memory, it goes by way of ST3 to allow time for completion of that storage and for switching over the FM address circuits from MA to AC2. ST4 moves the low order word from MQ to AR, sets FM WR so fast memory control will write the contents of AR into AC2, and continues the cycle to ST5. There are also five nonstore time states that enter ST5, and three of these are for floating point operations that also write a low order word in AC2. The fourth is the immediate exit from a page fail cycle for a page failure in a PI cycle; there is no page fail trap for this fatal error. The last is to restart following a page failure in a deposit function.

Note the small flow diagram at the lower right. Any time state that sets FM WR for writing in fast memory, whether treated as an accumulator or ordinary memory, also sets COMP AC WRITE if the comparison condition is satisfied for the fast memory address. Setting the flipflop enables the fast memory inputs to the mixer for the memory indicators, and causes the next time state to trigger the MI clock to load the indicators with the information being written into fast memory. For further information on this subject refer to section 5.6.1.

The other paths from ST2 are for special returns to earlier parts of the main sequence and a direct path to ST5 if there is no special exit nor AC2 storage. Since ST5 is the end of the main sequence, it always generates CLK EN INST DONE, clears BYF6 in a DMOVXM, and generates KEY DONE if the store cycle was used for a deposit from the console.

For all the special exits from ST2 (*ie* ST5 inhibit and no AC2 storage), the adder is enabled onto the address bus. For the next iteration in a block transfer the word count and source address in the two halves of BR are incremented and the processor returns to BLTT1. The remaining instruction returns are all to the fetch cycle for the second part in a DMOVXM or a block IOT, or the second or third part of an MUUO. Note that for a BLKI, MC MEM GO INH is set just as it is at the beginning of the fetch cycle in an ordinary DATAI. The final exit is for a page failure, which simulates the end of an instruction; however ST2 generates CLK PSEUDO INST DONE, so the clock does not stop and the return is made directly to IT2 for a page fail trap.

5.3.5 Page Fail Cycle

The flags at the top of drawing PF hold the information for generating the page fail word in the format described in detail in section 2.15 of the System Reference Manual. The signals that load these flags and that synchronize the page fail cycle come from memory control. If a page check fails, the signal MC PF QUICK sets PF SYNC (lower left) and the completion of the page delay loads the hold flags. If a page refill cycle is required, the cycle completion loads the hold flags, and if the information taken from the page map indicates that the page is inaccessible, it also sets PF SYNC. In this case the hold flags are loaded even though there may be no failure, for if there is a failure, memory control dispenses with the second page delay. If the page is accessible there is another page delay, and the flags are reloaded at its completion if there is a failure.

The first flag on the left is for bit 8 and it is set if user paging is in effect. The other flags are for bits 31–35, and these are set up by the gating below them. If there is a hard failure bit 31 is set and PF CODE 2X prevents the loading of the A, W and S data into bits 32–34. An address failure generates the illegal entry signal but with neither PAGE LAST INST PUBLIC nor PAGE PRIVATE INST set; the combination of these conditions thus sets bits 34 and 35 to produce the number 23 as the type of failure. An illegal entry with a 1 in either of those page flags sets only bit 35 for the number 21. Any other type of proprietary violation also sets only bit 35. Similarly a page refill error sets bit 34, and a small user violation sets no bit other than the most significant (22 and 20). If there is no numbered failure, the negation of PF CODE 2X allows bits 32–34 to be loaded from the page map information, and the absence of a small user violation or page refill error allows bit 35 to be set if the memory subroutine would write in the page. Note that the signal that controls bit 32 is not derived from the page information directly, but is instead PAGE REFILL CYCLE(0). If a failure occurs as the refill cycle is completed, it must be because the page is inaccessible; therefore bit 32 receives 0 and bits 33 and 34 receive the data being made available to the associative memory from the map half word for the page. If the page failure occurs in the page delay, the page must be accessible; hence bit 32 receives 1 and bits 33 and 34 are loaded from the data supplied by the associative memory.

The remaining logic in the drawing is for control of the cycle, and the cycle flow is shown in chart PF. Once PF SYNC is set there may be a wait for receipt of a sync point from memory control; *eg* it would be undesirable to wipe out the present time state because of a page failure that occurs in an overlapped instruction fetch before the current instruction is even finished. The synchronization triggers PFT0, which clears the present time state by setting MR STATE CLR FF and substitutes the appropriate page fail time state provided there is no MAP condition. This condition, shown at the bottom of the print, is generated both by the MAP instruction and by a memory operation called from the console. In MAP there is a return to the original time state just as though there were no page failure, and a page failure produced by the operator cannot be allowed to interfere with normal processor operation. Hence for either of these situations there is no state clear. If the failure is associated with an examine or deposit, PF KEY sets (turning on the console key page fail light) and PFT0 triggers MCT RECYCLE to fetch the next instruction, which will be used only if the processor is running. The condition MC ASYNC START(0) prevents the fetch from recycling again, should it fail.

The clearing function clears the flipflops that define the time state and the type of cycle, but does not destroy flags or other information that must be saved (such as the time state set up by PFT0). Following the clear, the clock restarts. If there was no state clear the processor is either in KT3, in which case the key function continues, or in FT6 or FT8, in which case the MAP resumes just as it would following a memory subroutine with no page failure. If the failure occurred in a BLT, the return is made to a special part of the BLT sequence at BLTT5; this special sequence duplicates the operations performed by the regular page fail cycle, but it also performs special functions necessary for premature termination in the same fashion as though the instruction had been terminated by an interrupt. A page failure in a PI cycle is regarded as a fatal error, so for this the processor goes to PFT4 to set the In-out Page Failure flag, stop any writing that may have been called for in the memory subroutine, put PC on the address bus and set MC INST FETCH NEXT, and then goes to ST5 to terminate the sequence.

If none of the above conditions holds, the processor enters a standard page fail cycle at PFT1. This clears various memory control flags to prevent a new instruction fetch or the fetching of a second operand for the terminated instruction. PFF1 SET not only sets the indicated flag but also sets TRAP PAGE FAULT; and if PAGE LAST INST PUBLIC is 1, it sets PAGE CLR PRIVATE, whose transition clears PAGE PRIVATE INST to prevent a further page failure in case the present one was caused by an illegal entry. The clock turns on the AD gate that enables the magic numbers and PFF1(1) makes the information for the page fail word available to the AD gating via these inputs. As shown in drawing MAMS, PF HOLD USER conditions bit 8, AB 18–26 supply the virtual page number to bits 9–17, and the remaining hold flags supply the failure type to bits 31–35.

PFT2 loads a page fail word into AR and enables AD onto the address bus (although this will not actually be used). At the next clock the STORE switch generated by PFF1(1), which disables the IR decoding so there are no other instruction switches except SAC INH, sets up a synchronous memory write subroutine including AB AB EN to start the necessary sequence, but it sets both MA SPECIAL and MA SPECIAL UBR so that memory control accesses the user process table at the location specified by the MA special levels. As shown on drawing MAS, PFF1(1) generates the appropriate address levels to select location 426 for storing an executive page failure word, otherwise 427.

At PFT3 the processor also enters ST1 (where the special BLT page fail sequence rejoins the standard flow path), and following the page check it enters ST2 for the standard store cycle write restart, which in this case writes the page fail word in the user process table. The ST2 clock produces the standard instruction termination events as conditioned by CLK EN INST DONE, but CLK PSEUDO INST DONE prevents the clock from stopping and the processor goes directly to IT2 for a trap cycle.

5.4 TRAP LOGIC

A trap is produced by setting any of the right three flags at the top of drawing TRAP. When a page failure occurs, TRAP PAGE FAULT is set at the same time as PFF1. The TN flags 0 and 1 are what the programmer knows as the Trap 2 and Trap 1 flags. The conditions that set TN 1 are equivalent to the arithmetic overflow conditions that set AR OV (section 7.2). TN 0 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 saved by setting FLAG 2). The entry into a trap cycle is shown in the instruction cycle flow chart IC. To begin a page fault trap the page fail cycle returns directly from ST2 to IT2. To bring about an overflow trap the signal TN=0 must be negated, *ie* one of the trap flags must be set and EBR TRAP EN must be set to enable overflow traps. If TN=0 is false and the processor is not already in a trap cycle or a PI cycle when it starts an instruction at IT0, then rather than going to IT1 to continue the instruction it goes instead to IT2. In this time state the clock sets TRAP CYC and TRAP BEGIN, inhibits PC+1 since the interrupted instruction must be restarted after the trap, and unlatches IR so that it can receive a new instruction code from MB. To supply the address for retrieving the trap instruction, USER MODE(1) causes the clock to set MA SPECIAL UBR to supply the base address of the user process table; otherwise the executive process table is selected. It also sets MA SPECIAL to gate the special address levels into MA to select the location in the table. Following IT2 the processor may be diverted to a PI or key cycle just as in IT1, but if it is not diverted it stops the clock and calls for a pseudo instruction fetch. During the memory subroutine TRAP BEGIN supplies the special address bits by generating TRAP ANY for any trap and TRAP REAL for an overflow trap (refer to print MAS). For any trap, 1s are supplied to address bits 27 and 31, and for an overflow trap 1s are supplied to bits 34 and 35 as determined by the TN flags. Thus for a page fail trap the instruction is taken from location 420 in the process table, and for an overflow trap it is taken from one of the locations in the range 421–423 corresponding to the number contained in TN 0–1.

When the instruction is ready, synchronization is immediate since INST DONE is still set. For this instruction the processor must go from IT0 to IT1 as it is now in a trap cycle (although the cycle can be diverted if the trap instruction uses indirect addressing). Once fetched, the trap instruction is performed like any other in the address space in which the trap occurred, except that TRAP CYC(1) causes MUUOF1(1) to generate MA SPECIAL 35. There are two locations for MUUO PC words for each machine mode, and MA SPECIAL 35 selects the odd one as is required when the MUUO is being executed as a trap instruction.

If an instruction that overflows also causes a page failure, the trap for the latter has precedence by virtue of the fact that TRAP PAGE FAULT being set disables TRAP REAL. Moreover this flag also generates TN CLR INH so that the only reasonable way to generate TRAP SATISFIED is by the condition AR FLAGS EN. In other words a page fail trap instruction should be a jump that saves the flags, thereby satisfying the page failure trap and also saving and clearing the TN flags so that a proper return can later be made to the overflow trap.

For overflow traps, the processor generally produces TRAP SATISFIED in every main sequence because such a trap usually occurs right after the instruction that causes it. However the trap cannot be satisfied until the trap instruction is guaranteed of completion. If TRAP SATISFIED were generated and then the instruction were interrupted, the trap would be lost – with the TN flags clear, there would be no way to return. Ordinarily the trap is satisfied at the beginning of the execute cycle unless there is non-E storage, in which case satisfaction is delayed until ET1, *ie* following the required page check. But the standard generation of TRAP SATISFIED is inhibited by assertion of TN CLR INH. This gate implies an instruction that cannot satisfy a trap at all (as in a page fault trap or a PI cycle) or an instruction that can be interrupted after the execute cycle (or the first execute cycle). In a DFDV or BLT, TRAP SATISFIED must be put off until some time state beyond the last point at which the instruction can be interrupted. In the other cases it is put off until the execute cycle of the final part (*eg* in the non-XCT instruction that is finally executed by a string of XCTs).

5.5 MODE CONTROL

The monitor selects the process tables by loading the user and executive base address registers shown in print UEBR. The base address used in accessing a process table is supplied by UBR or EBR (through the mixer at the top of the print) as selected by gating levels from special memory address control (print MAS). To supply the base addresses and other information, the program gives a DATAO PAG, which generates UBR LOAD if bit 0 of the data word is 1 and generates EBR LOAD if bit 18 is a 1. The latter loads information from the right half of the word into EBR and the Page Enable flag shown at the bottom of UEBR. The UBR LOAD signal loads information from the left half of the word into UBR, into FM ADR 0–1 to select the fast memory group for the user, and into the Small User and User Address Compare Enable flags shown at the left in drawing PAG2.

The logic that controls the machine mode and the type of paging is shown in print USER and in the upper half of print PAG2. Ordinary manipulation of the flags by the program is accomplished by means of the ARF LOAD signal, which is generated by either a JRST 2 or an MUUO. However for some of the flags, further gating produces restrictions, so that in some cases manipulation can be achieved only by an MUUO, and in others only from a particular mode or accompanying a particular mode change. As an example consider USER MODE, which is the User flag. This flag is controlled by bit 5 of the PC word in an MUUO. A JRSTF can set User (*ie* from executive mode) but cannot clear it, as once set it remains set until PAGE LEAVE USER is negated. This signal, which is produced by the net in the center of drawing PAG2, is generated by ARF LOAD only in part three of an MUUO, at which time it also allows the instruction to manipulate the Public flag (shown at D7) according to bit 7 of the PC word.

The flipflops in the lower right of USER supply certain mode information and also save it for one time state beyond which it is specified by the inputs (the mode flags change at ET2 but the instructions that change them end at ST1). The left one indicates that the processor is in user or supervisor mode because either User or Public is set. The right one indicates that user IOTs are illegal (except for device codes 740 and above) because either the processor is in user mode with User IOT clear or is in supervisor mode. In the upper left is User IOT, which controls the availability of in-out instructions to a user program, and also has a use in executive mode wherein it selects the type of memory for which memory operand references can cross over from executive to user space in an executive XCT. A JRSTF can clear this flag at any time but can set it only in executive mode; an MUUO can set or clear it in either mode.

The logic associated with the five flipflops at the top of drawing PAG2 controls switching among the modes (in conjunction with USER MODE) and detects an illegal entry into the concealed area of the address space. For the latter the signal PAGE CHK PRIVATE is generated (lower left corner) while the memory subroutine is actually fetching an instruction under normal circumstances (*ie* the fetch is not from fast memory or a process table and memory control is not performing a page refill in preparation for the actual instruction fetch). The assertion of this signal causes memory control (via timing signal MCT0) to set PAGE PRIVATE INST if the instruction is being taken from a nonpublic paged area or the unpaged executive area. While the processor is in concealed mode the flag is simply held on by every memory subroutine until an instruction is fetched from a public area, indicating a return to public mode (wherein the clearing of PAGE PRIVATE INST causes the next IT1 to set PAGE LAST INST PUBLIC). But when PAGE PRIVATE INST is set initially by retrieval of a private instruction from public mode, the determination of whether the entry is legal must wait until the instruction is decoded in the instruction cycle. This is handled by means of the upper gate that generates PAGE LIP CLR A (at C4) to clear Public at IT1 if the instruction is a JRST 1. Any other instruction allows Public to remain set, and both flags set is one of the conditions that produces PAGE ILL ENTRY at the lower left. This signal has no immediate effect (although the generating of it prevents writing in AC), but it causes a page failure during the next page check. (The other conditions for PAGE ILL ENTRY are for an address failure, which is treated in section 5.6.3).

The flag at D4, PAGE LAST MUUO PUBLIC, is the Disable Bypass flag, which when set prevents the supervisor from using an executive XCT to access the concealed user area. It can be manipulated by either a JRSTF or an MUUO, but it has an effect only in executive mode and is therefore manipulated in a meaningful way only by instructions that are under the control of the executive (*ie* instructions that enter or are in executive mode).

As described above, an MUUO can control both Public and User by generating PAGE LEAVE USER. The other conditions that generate this signal cause both Public and User to be cleared so that the processor enters kernel mode whenever it is placed in normal operation from the console or the flags are saved in an interrupt instruction (an instruction that calls an interrupt routine). A JRSTF can clear Public only by generating PAGE LIP CLR A, which requires a 1 in PC word bit 5 when user paging is not in effect; in other words the program can enter concealed mode simply by clearing Public (*ie* without making a valid entry) only when it is in executive mode and is simultaneously entering user mode (this method is not available for a public program to enter concealed mode or a supervisor program to enter kernel mode).

Manipulation of PAGE PRIVATE INST outside of a memory subroutine is effected through the direct set and clear inputs, but this control is clocked because the inputs are derived from the clocked flipflops at the upper right. A JRSTF or an MUUO with a 1 in bit 7 produces PAGE ENTER PUBLIC to set PAGE CLR PRIVATE. When ARF LOAD is true but bit 7 is 0, PAGE ENTER PUBLIC is negated, allowing PAGE SET PRIVATE to be set provided either the instruction is an MUUO or is an executive instruction that is entering user mode (the same condition that allows a JRSTF to clear Public). Note that PAGE ENTER PUBLIC is also generated when Public is on at the beginning of a page fail cycle to disable PAGE ILL ENTRY in case that was the reason for the failure.

5.5.1 User Paging

The remaining logic on USER determines the type of paging and access to the user AC stack. In the large net at the lower left, the third AND gate specifies that user paging is in effect whenever the processor is in user mode unless such paging is inhibited by the gate at the right; this inhibit applies during a key cycle or a PI cycle until memory control begins to fetch the next instruction, which must come from user space. The gate above the user mode gate selects user paging during a key cycle if the user paging switch is on and the exec paging switch is off, where again the specification is disabled when memory control goes for the next instruction, which must be taken from the space associated with the mode. The remainder of the net is for the selection of "user paging" in an instruction executed by an executive XCT.

The flags that control the selection of the type of paging in an executive XCT are on the right in drawing XCT. At ET2 in an XCT in executive mode, 1s in bits 12 and 11 set XCTP RD and XCTP WR. Then at the beginning of the executed instruction (the second part in a byte instruction) a 1 in either of these flags sets SCTP ACTIVE. That an instruction is being executed by an executive XCT is indicated by XCTP ACTIVE being on. This condition produces XCT PROT BYPASS to allow the supervisor to access the user concealed space if PAGE LAST MUUO PUBLIC is clear; and if user fast memory block 0 is selected, it generates XCT SHADOW REF for each USER PAGING access, so any fast memory operand reference is to the shadow area when USER PAGING is true.

The net that selects which memory operand references shall use "user paging" is at B8 in print USER. XCTP RD(1) enables the net for a double operand fetch and a synchronous read or read-modify-write; XCTP WR(1) enables it for a memory access that is limited to writing. With the output of this net true and XCTP ACTIVE set, USER PAGING is true if USER IOT is set; but if that flag is clear, "user paging" applies only to fast memory references and the user AC stack is therefore enabled if the reference is within locations 0–17. The program selects the stack pointer by giving a CONO PAG, to load the flipflops in the upper right. For a reference to the AC stack, USER AC STACK EN supplies this pointer to MA 27–31 to select a group of sixteen locations in the user process table.

5.6 CONSOLE CONTROL

The basic element in the control of the processor from the console is the key logic, but first let us consider a number of lesser elements that also play a role. At bottom center in print APR2 are the connections to the mechanical readin device switches. Most other switches, such as the sense switches at the top of the drawing, use the switch flipflop circuit described at the end of section 4.2. At the right of the sense switches is the net for the Maintenance Mode flag; this is set by various switches, any one of which being on indicates that the processor is being operated for maintenance purposes, and almost invariably indicates that it is not running at full speed. Most of the operating switches are in the upper half of print KEY3. The two IND drawings show the mixer for the shared indicators, which the operator can use to display data from different sources as selected by the switch at the top of IND1. Drawings DS1 and DS2 show the data switch flipflops, whose data inputs are fed by a mixer so they can receive information from the memory indicators as well as acting like toggles. The control signals for the mixer and flipflops are supplied by the logic associated with the clear and load keys for the switch register at the right in drawing KEY2. The one-shot in the upper right corner provides a good edge to trigger the PAs that supply load pulses for the data and address switches. The MI inputs to the mixer are enabled while the load key is held on, and the toggle inputs are enabled at all other times.

5.6.1 Address Switches

The switches on the AS prints work the same way as the data switches but have a 4-bit input mixer; hence they not only can act as toggles and receive data from MI, but can be loaded by the program from the IO bus and can function as a counter via the address bus just like PC (AS goes to AB through the +1 gates in AS1, so loading from AB can be equivalent to incrementing). At the right in print KEY2 is clear and load logic similar to that for the data switches, but now there are three conditions that trigger the load pulse, and the toggle input is enabled only when all three of the others are not. Three of the enables are generated by the nets shown in print KEY2, but the enable for program loading is generated by the flipflop in the upper right corner of print AS1. Incrementing via AB occurs only in a synchronous memory subroutine called from a key function that enables the +1 gates by setting KEY AS+1 (print KEY1 D3). Associated with the address switches are the paging and address condition switches, which can also be loaded by the program; the switches are in the middle row in print KEY3 and the mixer for them is at the lower right in print AS1.

As mentioned, examine next and deposit next increment AS by means of a loop through the AS+1 gates and AB. However AS has 22 bits whereas AB has only 18. Hence the AB inputs to AS 14–17 are provided from AS+1 14–17 through the pseudo AB latches at the right in drawing AS1, and it is these AS AB signals that provide the needed extra four address bits to MA for physical addressing from the console. The pseudo AB bits are latched simultaneously with the real address bus by setting AB AB. AS+1 is gated into AB by setting AB AS, but note that the signal that gates AS+1 14–17 into the pseudo AB bits is AB AS C, which (upper right corner, drawing ABC) is generated from AB AS(1) only when the executive paging switch is off. So with executive virtual addressing from the console, no inputs to these latches are enabled, and the pseudo AB bits are guaranteed to be 0s so they cannot interfere with page data supplied by the associative memory. (It is not necessary to do anything about these bits in user paging as the failure of the signal MA DIRECT (see drawing MAC and section 6.2) then prevents the enabling of AB into MA 14–17 altogether.) A gate at AS1 C7 detects zero in these extra AB bits for an input to the net that decodes AB 18–31 for zero in the upper left corner of print ABC; this last signal is used to detect an AC reference and it cannot be allowed to be true when a physical reference from the console is being made to one of the first sixteen locations in some other 256K block rather than to the real fast memory. (Note that counting in AS 14–17 in key functions, which must be allowed in physical addressing, also occurs in user addressing although it is inhibited in executive addressing.)

5.6.2 Master Clear and Restart

Of the two MR drawings, the second is devoted almost entirely to tables that show thermistor location and the connections for the console lamp test and the console lock and data lock. The switches are at the lower left in MR1. Mode signals generated by the net at the lower right are used only to drive the console mode indicators – they are not used in the logic.

The master clear logic has several clear functions for different purposes. The state clear clears only the time state logic; the master reset clears almost everything but the time state logic, the key logic and operating switches; the key clear clears only the key logic; and finally the power clear clears everything mostly by generating all of the other clear functions. There is also a panic clear that allows the operator to refrain from turning off power but nonetheless to stop the processor when it is running wild. The one-shot at MR1 C5 enables power clearing for 10 ms at power on, power off, and at the timeout of the timer shown in the upper right corner of print APR1 (discussed at the beginning of section 10-3). The power clear enable generates the other clear functions at the upper left and also enables the power clear clock at C4. The key clear is generated only by a power or panic clear, and except for power and panic clearing, the master reset is generated by the key reset (reset and readin functions) whereas the state clear is generated only by a page failure. The panic clear is produced by the logic at C4–D4. If the reset key is on for 10 ms without managing to set the reset successful flipflop, the reset panic delay times out and generates the clear functions other than power clear.

The automatic restart following a power failure or timeout is handled by the circuit at C3-D3. The restart signal comes from the 857 power control, but the enabling of it is handled by logic shown at the center of print APR2 and described at the end of section 9.4. If either the restart signal is true or the timer flipflop is set 10 ms after power clear, MR RESTART B sets. This in turn clears the timer flipflop, triggers the start key function, and generates address 70 for the executive process table through the MA special logic on print MAS (address bit 30 is produced by setting the PI normal cycle flipflop, which is used for setting up addresses of standard interrupts).

5.6.3 Memory Indicators and Address Comparator

Words are displayed on the console by means of the memory indicators in print MI. These flipflops receive their input from the mixer in print MIMX, which in turn may receive information from MB or from the inputs to fast memory. The indicators are loaded by the MI clock shown in the upper right corner of drawing CMP2, and the enables for the mixer are generated by the nets at the lower left from the state of COMP AC WRITE at C2. This flipflop is 0 when the comparison condition is satisfied for a memory read or core write, or MI is loaded by the program, so MB is the source of displayed data for these situations; but a fast memory write or AC store sets the flipflop, so MI receives the data written into fast memory. Giving a DATAO PI when the MI program disable switch is off triggers the clock to load MI and also sets COMP MI PROG (C3); this flag turns on the console program data indicator and disables the memory display logic. The flag is cleared by the operator turning off the switch or pressing reset, and by an examine or deposit function generating the clear signal for it (CMP1 B3). The clear signal also sets COMP AC WRITE to enable the MI clock and selects the FM inputs as the source of information for the indicators, but this is alright as those inputs come from AR, which always holds the information being examined or deposited.

The rest of the logic in the CMP drawings implements the actions discussed at the beginning of section 3.1.3, *ie* it implements the address comparison for displaying data in MI, stopping, or triggering an address failure. (In order fully to understand the operation of the comparator, the reader should be familiar with the memory logic (Chapter

6).) The nets in the upper part of CMP1 detect equality of an address on the bus with one specified by address switches 18–35 (the comparator operates on virtual addresses only). The net at the lower right determines when a comparison is enabled in terms of the paging currently in effect as against the settings of the paging switches. The signal COMP AB=AS at B7 is true when the addresses are equal and the comparison is enabled, but this signal is also asserted in any examinee or deposit so that these key functions act somewhat as though the comparison condition were satisfied in them. The logic for an address stop or address failure is in the lower right of CMP2. COMP CYC TYPE indicates that the type of memory reference being made is one of those selected by the address condition switches; with COMP AB=AS true and the address stop switch on, this signal sets the memory stop flag (print MCM C2) to stop the processor in the memory subroutine. The same conditions that indicate a real comparison condition satisfied also generate COMP ADR BRK provided the address break switch is on and the program has not inhibited the break; this signal in combination with the type of reference selected by the condition switches (equivalent to COMP CYC TYPE) produces PAGE ILL ENTRY through the large net at the lower left in print PAG2, thus causing the page check to fail.

The display of information can occur not only in the cycle types that can cause a stop or break, but also when writing in an accumulator with the comparison enabled and the address switches selecting the same accumulator as determined by the nets in CMP1 upper right and bottom center. The remaining logic on CMP2 is for sorting out the events in the memory pipeline and triggering the MI clock for the correct reference even with overlapping. When COMP AB=AS holds, completion of the page delay (chart MC2) sets up the flipflops in the upper left to remember that the condition is satisfied so that subsequent action can be taken during the memory cycle even though AB changes. In a synchronous reference the left flipflop is set, whereas in an asynchronous reference the right flipflop is set but the left one is held on if it is already set (it is possible to read or write a given memory location and then fetch the next instruction from it). Then in a true read reference with the synchronous condition satisfied (not a page refill), the read restart or its equivalent in a fast memory reference (charts MC3 and MC6 respectively) sets COMP MEM READ. For a core read the parity check triggers the MI clock (chart MC3). For a fast memory read or core write respectively the gate at C4 generates a separate clock trigger at a later pulse in the MCT chain or at the write restart (chart MC5). A fast memory write reference is handled the same way as writing in an accumulator: even though COMP SYNC AB=AS is set at the completion of the page delay, the comparison is reestablished from the FM address in the store cycle; this in turn sets COMP AC WRITE to enable the clock and select the FM inputs to MI (fast memory writing whether as accumulators or memory is done from AR without using MB).

5.6.4 Key Logic

The logic for the key cycle and the functions that can be performed in it is shown in four block schematics and three flow charts. Flipflops in the upper half of print KEY1 control the synchronization of key manipulation to processor operation, the time states for the key cycle, and starting and stopping the processor (the program stop and manual stop flags (D6 and B1) simply drive lights on the console). Much of the logic at the left is for controlling the clock. The remaining circuits are mostly for indicating the initiation and completion of key functions and for returning control to the program from a key function performed while the processor is running. At C8-D8 in print KEY2 are one-shots that produce the readin pulse on the IO bus. The rest of the logic at the left is for single pulse and repeat operations; at upper center, for restarting following a memory stop; at the right, for the address and data switches. At the upper left in print KEY3 is the repeat delay; the upper two rows of flipflops are for the operating switches; at the bottom are additional control flipflops, mostly for readin but the left one controls the key cycle, and the right one a cycle used for a deposit function. Drawing KEYF shows the logic through which the processor responds to key manipulation: the bottom circuits filter out contact bounce, the priority chain in the middle determines which function to perform, and the flags at the top control the individual key functions.

Flow chart KC shows the key cycle and associated sequences such as key synchronization, repeat, and auto restart. Although the key cycle flow shows all entries, branches and exits, the only events listed for the key time states are overall control events or events that are common to several key functions and are generated directly by the key time states because they need not be inhibited for the other functions. Specific events for individual functions are only in the function flows; nine of these are in chart KF, and the more complicated read in is at the left in chart KRMP.

Clock Control. Clock starting and stopping is handled primarily by the logic at the left center in print KEY1, in particular by the nets that generate KEY CLK START (C6), KEY CLK INH (C7) and KEY CLK STOP (B5). When the clock is off, the operator pressing a key makes KEY ANY NO BOUNCE true; this drops KEY CLK STOP, which drops KEY CLK INH, which in turn generates KEY CLK START through its top gate to start the clock. As events progress through key synchronization and the key cycle, any time one input to the stop gate goes true some other input goes false, so the stop and inhibit both remain off. Action beyond the key function involves the flags KEY RUN and KEY IDLE at the upper right (the KEY RUN SET net is on KEY3 D2). When the processor is not running, RUN is clear and IDLE is set. Examine and deposit functions (and of course terminating functions) do not affect this configuration, and key cycle completion turns on the stop and inhibit to stop the clock. A start, continue, or read in however sets RUN and clears IDLE; so long as IDLE remains clear, the stop and the inhibit remain off, and the processor remains in the run state even though the clock may be inhibited temporarily by various internal operations or even inhibited altogether by a memory stop. But otherwise the processor is stopped by setting IDLE, which enables the stop and inhibit. A stop or reset function clears RUN (B1) and sets IDLE simultaneously; HALT sets KEY PROG STOP and clears RUN, which in turn causes diversion at the beginning of the next instruction cycle, and this diversion sets IDLE to stop the clock. The configuration RUN and IDLE both set never occurs. On the other hand, single instruction mode and the execute key function are implemented by operating the processor with RUN and IDLE both off, so that at the end of each instruction the processor acts as though it had just performed a HALT.

The upper gate for KEY CLK INH is an ordinary inhibit for a memory call by a key function. The other input at the lower gate is operative only when a key function is being repeated: with KEY CLK STOP on, KEY REPT DONE being set at completion of the repeat delay drops the inhibit and produces KEY CLK START through the second gate from the bottom. Note that there are two distinct types of repeat, a function repeat and a single pulse repeat. The former is tied to a particular function and must therefore continue to enable that function, whereas the latter simply keeps retriggering the clock and is divorced from the operations the processor is performing. In the logic the two are distinguished by the state of KEY SINGLE PULSE SYNC.

Single pulse operation makes use of the single pulse switch (KEY2 B5) and either the single pulser or the repeat switch on either side of it. The stop and inhibit are now irrelevant, for so long as the single pulse switch is on, KEY SINGLE PULSE SYNC (D6) disables the clock loop (see clock flow chart CLK and section 5.1.1) and generates KEY CLK START every time the single pulser is pushed or the repeat delay is done. But note that in automatic single pulsing, the clock cannot be triggered unless it is ready as indicated by the 1 state of flipflop KEY CLK RDY SYNC in the lower left corner of KEY2. This flipflop is cleared by each clock and is not set again until the end of the currently selected clock period, following the point at which a clock inhibit would interrupt the loop. Hence even with the repeat switch on, the clock stops at any time state in which it would have been inhibited anyway. This synchronization also prevents a very fast repeat from retriggering the clock before the appropriate time has elapsed.

Anded with ~KEY CLK INH in the top gate of the KEY CLK START net is the 0 state of the KEY CLK RUNNING one-shot at the left. This one-shot is disabled while KEY IDLE is clear, and while the processor is in single pulse mode or in a memory stop; but otherwise it is triggered by every clock and times out if there is no clock for a millisecond. Hence if the clock is started at normal speed by a key function that does not clear KEY IDLE, but the clock is broken, then this one-shot supplies a 1 kHz substitute clock that triggers the main clock one pulse at a time. In other words if the processor is so down that not even the clock is working, the operator can still use stop, reset, examine and deposit for the most basic troubleshooting.

In single instruction operation KEY RUN is clear so the processor stops at the end of every instruction. Pressing the single pulse switch sets KEY SINGLE PULSE SYNC at the beginning of the instruction cycle, but note also (print KEY2 D7) that it is set by any clock when KEY RUN is clear. Now if the processor is caught in some sort of a loop (say the divide subroutine) where it never reaches the end of an instruction, then neither STOP nor SINGLE INST nor SINGLE PULSE alone will stop it. Pushing RESET would stop it but would also generate the panic clear, destroying any information about what is wrong. On the other hand, pressing both SINGLE INST and SINGLE PULSE together stops the clock by placing the processor in single pulse operation immediately.

Key Cycle. The major path down the middle of flow chart KC begins with pressing an operating key to set one of the no-bounce flipflops at the bottom of print KEYF. Ordinarily the flipflops are clear, the NC connection of the key is low, and the NO connection is high. Pressing a key reverses the outputs to set the corresponding no-bounce flipflop; and although there is contact bounce, it is guaranteed to be no worse than both outputs high together, so once the key is pressed the flipflop stays set until the key is actually released. The nets in the center are configured such that any no-bounce flipflop being set generates the signal KEY ANY NO BOUNCE at the far right and applies an input to the key function flag that corresponds to the leftmost no-bounce flipflop that is set. At the far left is an OR gate that produces the signal KEY START OR RESTART whenever either KEY START NO BOUNCE or the auto restart flipflop is set; it is this signal that generates KEY ANY NO BOUNCE and is applied to the priority net, so the start function is used for both manual and automatic starts. Note that pressing the continue key sets its no-bounce flipflop only if the processor is not presently in a memory stop; if it is, the key instead triggers the logic for restarting memory.

If the processor is already running, the key synchronization simply uses the clock without regard to what time state sequence the processor happens to be in, except that it cannot do two synchronizations at once. With the processor not running, generation of KEY ANY NO BOUNCE drops KEY CLK STOP to start the clock as explained above. In any event pressing any key causes the clock to set KEY LATCH on print KEY1 D7. This flag latches the no-bounce flipflops so that they cannot change state regardless of what is done physically to the keys. The next clock then sets KEY SYNC (KEY1 D5) to produce KEY STROBE (KEY1 C2), which sets the function flag selected by the priority net. The same clock clears KEY LATCH, and after that KEY SYNC clears as soon as all keys have been released. Subsequently pressing a key will set KEY LATCH, but the procedure cannot go beyond the latching stage until the previous function is completed.

The setting of a key function flipflop generates KEY ANY ACT (KEY1 B6) which in turn produces KEY FCN RDY (KEY2 C6). If the processor is not running, this last signal directly sets KT1 to start the key time chain. Otherwise entry is made through one of the paths in the upper left of the chart. For these the processor sets KEY RDY SYNC to synchronize the key logic to the time state sequence as shown in drawing PIC1 (the same logic is used to synchronize a PI cycle). At a memory subroutine call for an instruction fetch, a key function being ready sets PI CYC RDY, which in turn aborts the instruction fetch and causes diversion from the instruction done entrance of the instruction cycle (see flow chart IC) to either a PI cycle or the key cycle (PI has priority). If the key synchronization is too late for one instruction fetch it might have to wait until the next, but it may start sooner by causing a diversion at one of the other allowable points in an instruction: namely at an address word fetch in an effective address calculation, at a trap instruction fetch (which aborts the program instruction just fetched), or even while an IO instruction is waiting for access to the bus. Diversion at these other points is implemented by setting PI DIVERT INDIRECT (PIC1 C5), which KEY RDY SYNC can do only when a key function other than execute is ready. This restriction on indirect diversion obviously prevents an execute function from interrupting itself, but note that diversion must be for a key function: diversion to the key logic for any reason other than a function being ready must either occur at the completion of an instruction or be inadvertent (see below).

KT1 clears various control flipflops, puts the data switches via the IO bus into AD for those functions that use them (the extended clock period is also for this), and inhibits the PC+1 gate to prevent continue or execute from incrementing PC. Other events in the key cycle are for particular functions and are discussed with them below. Completion of a function is generally indicated by the signal KEY DONE. Only stop and reset are done at KT1 and they idle the processor (but the clock does not stop until the operator releases the key). For any other function KT1 sets KEY CYCLE (KEY3 B7), allowing extended time for memory paging and disabling KEY FCN RDY to prevent the function from continuing to restart its own cycle. Start, continue and read in end at KT2 from whence they go to the instruction cycle to begin the program, but read in is done only after it has executed a key cycle (containing a processor main sequence) once for every word read. The examine and deposit functions use the whole key time chain, and the deposit functions also use the store cycle to deposit the data. Following an examine or deposit the processor stops if it was not in operation when the function was started, but otherwise it returns to the instruction cycle to continue the program. At completion of a function that is not being repeated, KEY DONE generates KEY CLR (KEY1 A4) to clear the function flag at the top of print KEYF. But note that KEY CLR does not clear KEY XCT, and in fact execute is the only function that never produces KEY DONE. The reason for this is that the execute key cycle includes a main sequence for executing the instruction (KT3 triggers the instruction cycle), and the function relies on the completion of the instruction to terminate the function and continue the program.

Turning back now to KEY RDY SYNC, we see that although it is otherwise disabled during a key cycle, it is held on when KEY RUN is clear. Thus if an execute is done when the processor is not running, the attempt by the executed instruction to trigger the next instruction cycle causes a return to the key logic to stop the processor. This same mechanism ensures that the processor will stop after each instruction in single instruction mode, and it also handles the stop following a HALT. But note that even while KEY RUN is clear, KEY RDY SYNC is held off while XCTF is set. This is to prevent premature termination of an XCT or LUUO done as a single instruction or by the execute function; in other words these instructions include execution of the instruction called by the XCT or LUUO.

Generally an entry to the key logic with KEY RUN clear is made from instruction done, and the PC clock is to load the address of the next instruction (being E for a HALT) for a subsequent continue. The other entries can be used with KEY RUN clear only when there is a priority interrupt while the processor is operating in single instruction mode. And for this the processor stops instead of doing the interrupt or performing the instruction. Hence in single instruction mode the operator should either refrain from using interrupts or be prepared to handle them manually.

Key Functions. The functions in flow chart KF are numbered in the order of their priority (for read in see section 9.3).

Reset, Stop. These two functions stop the processor in an identical fashion except that the former triggers the reset signal on the IO bus and produces the master reset to clear the machine. If the reset function is not performed within 10 ms after the key is pressed, the master clear logic produces the panic clear.

Start. KT1 places the processor in operation in kernel mode and places the contents of the address switches on the address bus. KT2 triggers an instruction fetch and enters the instruction cycle.

Continue. If the processor is in a memory stop, pressing the continue key sets KEY MEM CONTINUE (KEY2 C4), which in turn both prevents the setting of the no-bounce flipflop and triggers KEY MEM CONT. For a stop in a write cycle, this pulse directly gives MCRST0 to restart the memory subroutine (flow chart MC5). For a read cycle, KEY MEM CONT produces KEY RD CONT, whose trailing edge generates KEY MEM GO, which in turn clears MC MEM STOP and produces MCRST1 to continue the subroutine (chart MC4).

If the processor is not in a memory stop, pressing the key sets the no-bounce flipflop and waits through the usual key synchronization. When the function begins, KT1 places the processor in operation in its present mode and puts PC on the address bus. KT2 triggers an instruction fetch and enters the instruction cycle.

Execute. KT1 clears KEY IDLE without affecting KEY RUN, and it puts the data switches into AD from whence KT2 loads them into AR. KT3 then enters the instruction cycle, but it also moves the data switches to MB and simulates instruction ready, so the instruction cycle will immediately begin executing the word from the data switches as though it had arrived from memory. KEY XCT remains on, preventing the setting of PI DIVERT INDIRECT so the function cannot interrupt itself. Completion of the instruction is indicated by TRAP SATISFIED, which clears KEY CYCLE and also clears KEY XCT unless the function is being repeated. But note that should some condition prematurely abort the instruction – such as a priority interrupt or execution of a trap instruction in place of the one given by the data switches – then KEY CYCLE clears and prevents the completion of the substitute instruction from clearing KEY XCT; hence the function will be restarted later. Once the function is complete, the processor continues with the program unless KEY RUN is clear, in which case KEY RDY SYNC being set resynchronizes the processor to the key logic to terminate the function.

Examine, Examine Next. These functions are identical except that in examine next, besides placing the output of the AS+1 gates on the address bus, KT1 enables those gates in order to examine the location one beyond that specified by the address switches. KT2 calls a standard memory read subroutine, triggers a memory display through the comparator logic (section 5.6.3), and sets up an automatic instruction fetch if the processor is running. The completion of the page delay moves the (perhaps incremented) address back to AS and simulates the satisfaction of the comparison condition so that the function stops in the memory subroutine if the address stop switch is on. (Note that setting MC ASYNC START disables the condition so that it cannot affect a subsequent instruction fetch.) After the word is read, KT3 triggers a return to the instruction cycle, but this is effective only if KEY IDLE is clear – in other words if the processor was not already running it simply stops.

Deposit, Deposit Next. These two functions differ from one another in exactly the same way that examine and examine next differ. In other respects the deposit functions are very similar to the examine functions, the differences being as follows: KT1 sets KEY DEP CYCLE to generate the SCE switch, and makes use of the common KT1 gating of the data switches into AD in that KT2 loads them into AR and calls a memory write instead of a read. From KT3, KEY DEP CYCLE causes entry to the store cycle for depositing the word, and the function is completed at ST5. However should a page failure occur, the store cycle is eliminated and KT3 goes directly to ST5.

Repeat. The repeat procedures for functions and single pulsing are in the upper and lower right of chart KC. Repeating is enabled by the logic at the left in print KEY2; turning on the repeat switch sets KEY REPT, which in turn allows the next clock to set KEY REPT SYNC.

Single pulse repeat operation requires that the operator turn on the single pulse switch so that IT0 or a clock triggered by pressing an operating key will set KEY SINGLE PULSE SYNC. Then pressing the single pulser begins each string wherein any clock starts the repeat delay (KEY3 A5,D7), whose timeout sets KEY REPT DONE to retrigger the clock provided it has not been inhibited.

To repeat a function simply turn on the repeat switch and press an operating key. KEY REPT SYNC being 1 enables the start net for the repeat delay and disables KEY CLR to keep the function flag set until the repeat switch is turned off. Each time the function starts, KT1 triggers the repeat one-shot, but it is continuously retriggered as long as the key cycle lasts, and for any initiating function it is further held on until the processor stops. While the one-shot is on it disables KEY FCN RDY to prevent the function flag from restarting the function. Once the one-shot is no longer retriggered nor held on, it begins its delay period; and when it times out it reenables KEY FCN RDY and sets KEY REPT DONE provided the switch is still on. If the processor is stopped, setting KEY REPT DONE restarts the clock to begin the function; otherwise the function waits for synchronization of the processor to the key logic.

CHAPTER 6
MEMORY LOGIC

In a DECsystem-10 each PDP-10 arithmetic processor contains its own 16-word fast semiconductor memory, but the core memories are separate units connected to the processor by a memory bus. The internal operation of these memories, their control functions, their timing, and the way they respond to processor requests are described in separate manuals. This chapter describes only the hardware at the processor end of the memory bus: the logic that requests access to memory, determines the physical address, supplies that address to memory, and controls the transmission and receipt of data.

A clock in a processor cycle or an execution sequence may request access to memory by triggering appropriate events in the memory control section of the processor. Depending upon the type of access, the clock may stop and wait for some response from memory control, or the clock may continue so that ordinary processor operations and the memory subroutine go on in parallel.

To access memory, memory control must supply an address to the memory bus from MA and must place various request signals on the bus. Memory control then waits for a response from the memory addressed by the high-order address bits. Once the memory is free and available to the processor, the time required to transmit or receive data depends upon the characteristics of the particular memory, although this time is always shorter than the memory cycle. For reading, memory control must wait until the data is available and the memory rewrites the word automatically (unless the same location is to be modified after a pause); for writing, memory control need wait only until the memory acknowledges the request, at which time the memory takes the data into its own buffer and continues with the clear and write cycle.

6.1 MEMORY DATA

The two MB drawings show the 36-bit memory buffer through which all data is transmitted between processor and memory. The data inputs to the MB flipflops allow for clocked transfers from AR and FM: from the former principally for sending data to memory, and from the latter when the memory subroutine is fetching data from fast memory. For receipt of information over the memory bus, MB is first cleared and data pulses from the bus are fed directly into the set inputs of the flipflops. The connections to the bus are shown in drawing MBD. At the bottom are the gates through which data pulses are placed on the bus by the write restart signal for the transmission of data to memory according to the contents of MB. Above the cable are the gates for the receipt of data pulses from the bus. These input signals are always fed to the set inputs of the MB flipflops, but they are also fed to the set inputs of the flipflops in AR if the word being received is an operand, as indicated by the gating level from synchronous memory control (in a two-word operand only the first is sent directly to AR).

The control signals for MB are shown in the lower part of drawing IRMB. Of the conditions that enable the MB clock (for the main clock), the readin condition and BLTT4(1) are for clearing MB. The remaining enables are for loading MB as determined by the nets at the right. Loading MB for storage of a result is always at ST1 going to ST2. The conditions that generate MB AR EN B are effectively for simulating the result of an instruction fetch: the word being put in MB is either an instruction being executed from the console or a pointer for the address calculation in the second part of a byte. The various double floating point conditions that generate MB AR EN A and C simply use MB as an arithmetic buffer register like BR.

Among the conditions that trigger the MB clock directly, MCT3 loads MB from FM during a memory reference to fast memory. The conditions at the set of four AND gates just at the left all trigger the clock simply to clear MB. The second gate clears MB following the transmission of data out over the bus to be written. The first and third gates generate the asynchronous MB clock at the end of the page check to make MB ready to receive data in a synchronous fetch or store or in an asynchronous (*ie* instruction) fetch, except in the case where the asynchronous fetch (instruction or second operand) is automatically following a synchronous fetch – in other words in the case where MC ASYNC START is set and one or the other of the hold flipflops is also set. The reason for this is that at the end of the asynchronous page check, MB is in use for the synchronous fetch. To prepare it for the asynchronous fetch is following automatically (as indicated by MCR MB POST CLR being set). In several specific instances, however, the clear function is inhibited. In the first part of a byte instruction, the incremented pointer written back into memory is also saved for use in the second part; MB is cleared by BLTT4, so there is no clear following the page check of the destination in a BLT; and if the processor is stopping because of bad parity, the post clear for a subsequent asynchronous fetch is inhibited in order to save the word containing the error.

Parity. The parity logic (drawing PAR) receives two signals from the memory bus: MAI PARITY which sets PAR BIT, and MAI IGN PAR which sets PAR IGNORE. The former signal accompanies data received from memory and is present if the parity bit of the word read is 1; the latter signal accompanies the acknowledgement from memory and is present if the ignore parity circuit in the memory is enabled. The parity nets at the top of the drawing generate an even parity signal from the data held in MB; this signal is inverted and supplied as a odd parity bit with data being sent out on the bus to be written.

When a word is read from memory, PAR BIT reflects the state of the parity signal received, and the nets determine the parity of the 37-bit word. Unless parity is being ignored (a condition that holds PAR ERR clear) the parity check signal from memory control sets PAR ERR if the output of the nets is even; and the setting of this flag in turn sets the Parity Error flag, which is one of the processor conditions (and can be cleared by a CONO PI, with a 1 in bit 19).

All writing in memory is done at ST2 and at this time the same clock that loads MB also sets PAR BIT so that the nets will generate the correct parity signal to accompany the data to memory. (Note that if the program has specified writing even parity for maintenance purposes, the level PAR WRITE EVEN inverts the output of the nets.)

6.2 MEMORY ADDRESSING

The address to be used in accessing memory is supplied to the bus through the memory address interface as shown in drawing MAI. This interface also handles all of the bus control signals, such as the request, parity and write restart signals to memory and the acknowledgement, data warning, read restart and ignore parity signals from memory. Drawings MAI and MBD together show the complete bus.

The address signals are supplied to the interface from the memory address latches shown in drawing MA1 and the right half of MA2. Although MA always supplies a 22-bit physical address to the bus, addresses supplied to MA are constructed out of physical and virtual pieces. Hence MA is effectively divided into three parts: bits 14–17 select the 256K portion of memory, bits 18–26 select the page, and bits 27–35 select the location in the page. Figure 6-1 shows the sources of the different parts of an address. The source of the information for bits 14–17 and 18–26 is the associative memory scratch pad for normal paged access, the address bus for unpaged access, and one of the base registers for access to a process table. Note however that the AB bits for MA 14–17 do not actually come from the address bus logic – instead they are derived from the address switches (section 5.6.1). In a programmed unpaged reference (to fast memory or the executive unpaged area) these bits are 0s; the only time they are not necessarily 0s is in an access called from the console, in which case all of physical memory is available and the bits are supplied by the address switches (as are all the other bits in the address via AB).

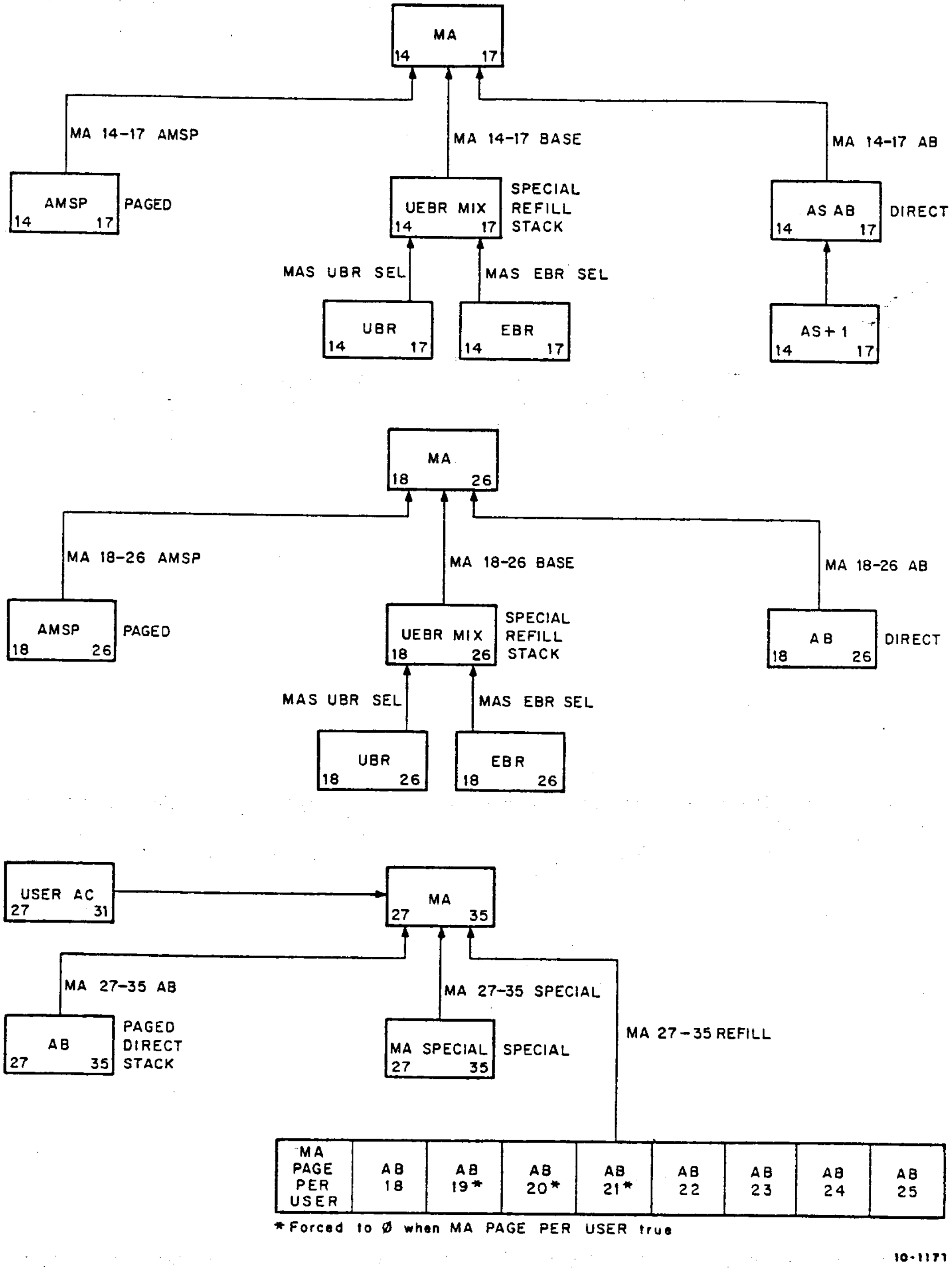


Figure 6-1 Memory Address Sources

For all ordinary memory access MA 27–35 receives address information from AB, and the other two enabling levels, refill and special, are exclusively for accessing the process tables: the former for retrieving page map data for the associative memory, and the latter for all other hardware-defined access. Note that the user AC stack pointer bits are ored directly into MA 27–31 without benefit of an enabling level applied to MA; these inputs are enabled directly by user paging control (section 5.5) and provide an address in the user process table in combination with AB. There is no actual overlap of the user AC pointer with AB, as a stack reference is *ipso facto* a reference within locations 0–17 and hence AB bits 27–31 are all 0. The individual address bits for hardware-defined access to a process table other than for page mapping are the MA special levels shown in drawing MAS. These levels are used to construct the addresses of the table locations used for page failure, MUUO, executive LUUO, trap, interrupt, and auto restart. The flow charts for each of these operations list the necessary special levels, but in any event the levels produced by any particular condition are simply those needed to generate the address bits for the table location appropriate to the operation.

The final group of inputs for MA 27–35 are those that specify the address of a page map word for a page refill cycle. For all user access and ordinary executive paged access the virtual page number from AB 18–26 is shifted right one place and used as the table address – in other words AB 18–25 is supplied to MA 28–35; this is equivalent to dividing the page number by two and is done because each word in the page map contains the map data for two pages. To get a mapping for a page in the per-process area, access must be made to user process table locations 400–417 for pages 340–377; this requires that MA 27 receive 1 and MA 29–31 receive 0s while the remaining flipflops receive the same AB bits as for accessing a location in the first half of the table. To generate these two configurations MA 28 receives AB 18 and MA 32–35 receive AB 22–25, but MA 29–31 receive auxiliary signals and MA 27 receives the signal MA PAGE PER USER. These signals are generated by the logic at the center in drawing MAC. When access is not to the per-process area, MA PAGE PER USER is false, placing a 0 in MA 27 and gating AB 19–21 into MA 29–31. But if the executive addresses a page in the range 340–377 (*ie* address bit 18 is 0 and bits 19–21 are all 1s) MA PAGE PER USER supplies a 1 to MA 27 and disables the auxiliary signals so that MA 29–31 receives 0s. (Note that MA AB 18 and MA AB 19–21=7 include another condition besides that the AB bits be 1s, but the extra condition is not relevant to this particular situation.)

The nets at the right determine whether access is direct, *ie* whether the address supplied is to be taken as a physical address to reference a location without paging. The signal MA DIRECT is true for a fast memory reference or when MA EXEC UNPAGED is asserted. A true fast memory reference is determined by the net at B7 in print MA2: MA AC REF is true when the address is in the range 0–17 and it is not used for an AC stack or shadow reference nor is there a special reference instead. (If an illegal entry has occurred, MA AC REF is inhibited in the memory subroutine that recognizes that page failure.) The signal MA EXEC UNPAGED (on MAC) is true when user paging is not in effect and the address specified is in the executive unpaged area (pages 0–337). However the signals for the AB bits reflect more than the states of those bits. MA AB 18 is false if AB 18 is 0 and MA AB 19–21=7 is false if any among AB bits 19–21 is 0, but both are false when MA KEY DIRECT is true. The net at the top asserts MA KEY DIRECT for references in a key cycle in which the exec paging switch is off, for all references while EBR TRAP EN is clear, and for any special reference. In a key cycle, user paging is in effect if the exec paging switch is off and the user paging switch is on; hence both paging switches off makes MA EXEC UNPAGED true regardless of the address used, making all key memory references direct. EBR TRAP EN is what the programmer knows as the Page Enable flag. In all ordinary computer operation this flag is set. When left clear, it not only disables traps, but also causes the entire executive virtual address space to be unpaged. Hence the executive runs only in kernal mode and directly addresses the first 256K locations in memory. MA SPECIAL (1) gives MA KEY DIRECT simply to prevent the arbitrary contents of AB from forcing selection of the user process table by inadvertently generating MA PAGE PER USER.

The actual selection of the source of an address for any reference is made by the net at the left, and the table lists the sources for the various types of reference. All enabling levels require that MC MA MA be 0, as setting this flipflop latches the register and holds off any enabling levels until the address is taken by the memory. If a reference is neither special nor a refill, the source for MA 27–35 is AB. Hence for a direct reference AB supplies the entire address, where bits 14–17 are 0s or are supplied by the address switches. For an ordinary paged reference the physical page is supplied to MA 14–26 from the scratch pad while AB supplies bits 27–35. For a reference to the user AC stack MA 14–26 receives the user base address and MA 27–35 receives AB with the pointer ored into MA 27–31. For a refill cycle or a special reference MA 14–26 receive the appropriate base address and MA 27–35 receive respectively the refill bits or the special bits.

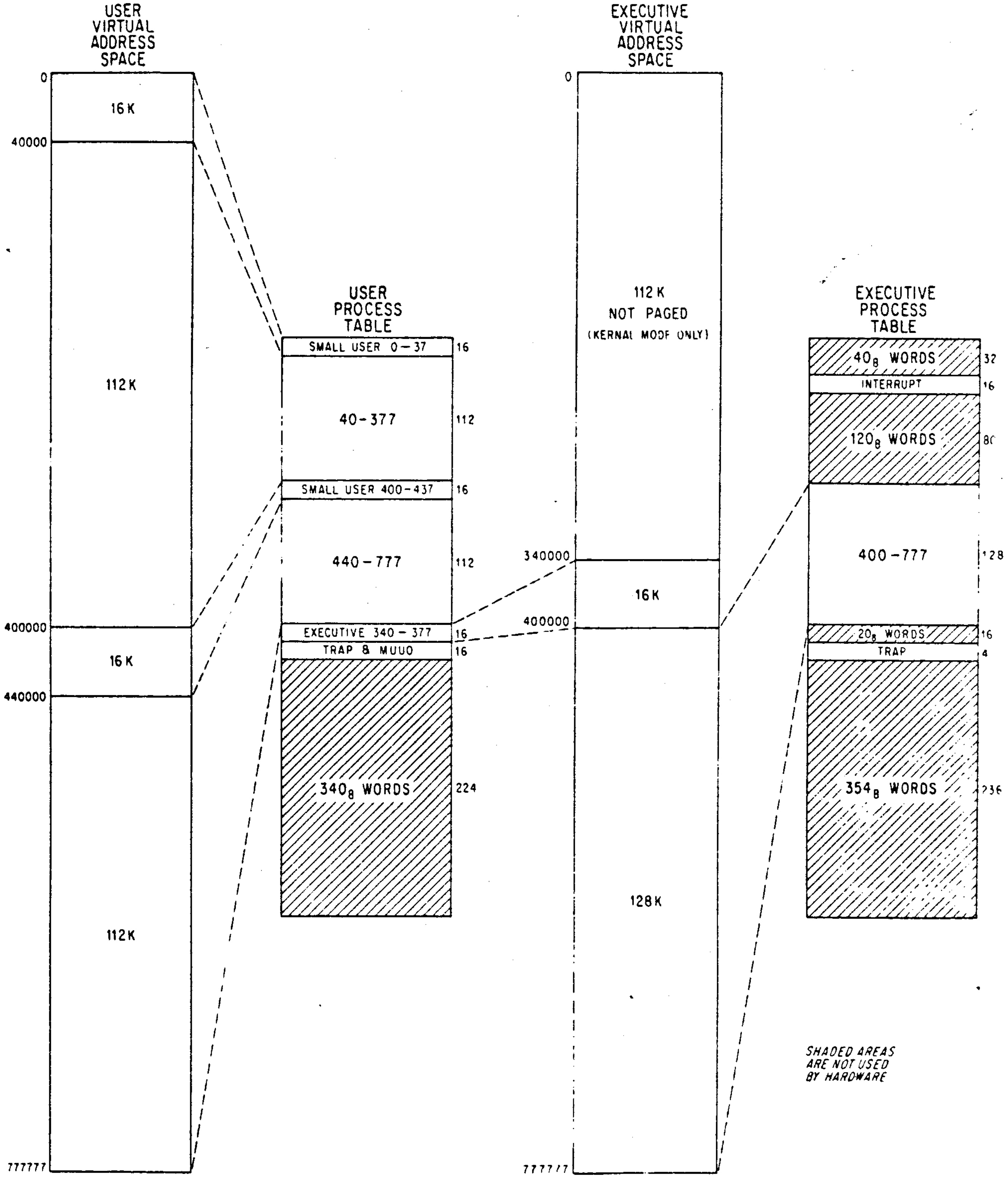
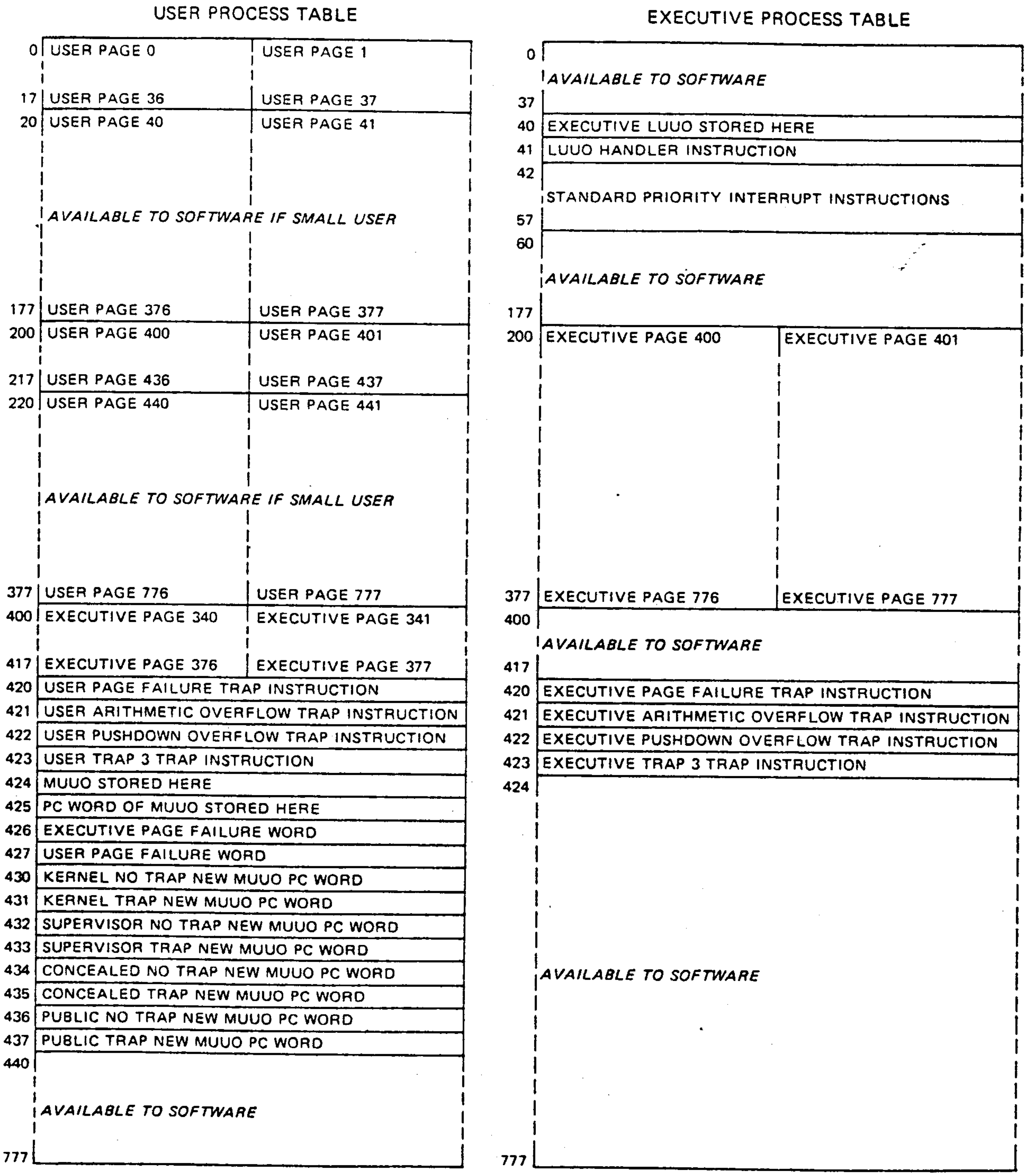


Figure 6-2 Virtual Address Space and Page Map Layout



USER PROCESS TABLE

0	USER PAGE 0	USER PAGE 1
17	USER PAGE 36	USER PAGE 37
20	USER PAGE 40	USER PAGE 41
	AVAILABLE TO SOFTWARE IF SMALL USER	
177	USER PAGE 376	USER PAGE 377
200	USER PAGE 400	USER PAGE 401
217	USER PAGE 436	USER PAGE 437
220	USER PAGE 440	USER PAGE 441
	AVAILABLE TO SOFTWARE IF SMALL USER	
377	USER PAGE 776	USER PAGE 777
400	EXECUTIVE PAGE 340	EXECUTIVE PAGE 341
417	EXECUTIVE PAGE 376	EXECUTIVE PAGE 377
420	USER PAGE FAILURE TRAP INSTRUCTION	
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER PUSHDOWN OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO STORED HERE	
425	PC WORD OF MUUO STORED HERE	
426	EXECUTIVE PAGE FAILURE WORD	
427	USER PAGE FAILURE WORD	
430	KERNEL NO TRAP NEW MUUO PC WORD	
431	KERNEL TRAP NEW MUUO PC WORD	
432	SUPERVISOR NO TRAP NEW MUUO PC WORD	
433	SUPERVISOR TRAP NEW MUUO PC WORD	
434	CONCEALED NO TRAP NEW MUUO PC WORD	
435	CONCEALED TRAP NEW MUUO PC WORD	
436	PUBLIC NO TRAP NEW MUUO PC WORD	
437	PUBLIC TRAP NEW MUUO PC WORD	
440		
	AVAILABLE TO SOFTWARE	
777		

EXECUTIVE PROCESS TABLE

0		
	AVAILABLE TO SOFTWARE	
37		
40	EXECUTIVE LUUO STORED HERE	
41	LUUO HANDLER INSTRUCTION	
42		
	STANDARD PRIORITY INTERRUPT INSTRUCTIONS	
57		
60		
	AVAILABLE TO SOFTWARE	
177		
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400		
	AVAILABLE TO SOFTWARE	
417		
420	EXECUTIVE PAGE FAILURE TRAP INSTRUCTION	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	EXECUTIVE PUSHDOWN OVERFLOW TRAP INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424		
	AVAILABLE TO SOFTWARE	
777		

Figure 6-3 Process Table Configuration

The logic that determines which base address shall be used in a stack, refill or special reference is shown at the upper left in drawing MAS. Any operation that requires reference to a process table other than a page refill or an AC stack reference sets MA SPECIAL, and if the reference is to the user process table it also sets MA SPECIAL UBR. The first of these flipflops selects a base address (*ie* the output of the base address mixer) as the source for MA 14–26; the same selection is also made for a page refill or reference to the AC stack. The nets at the center of print MAS select the base register that is gated into the mixer regardless of whether the base address is actually used. The user base register is selected for all user paging, for references to the per-process area or the user AC stack, and in any operation in which MA SPECIAL UBR is set. Otherwise the executive base register is selected.

6.3 ASSOCIATIVE MEMORY

An associative memory is one whose locations can be selected by their contents (content addressable) as well as by addresses assigned to them. When data is presented to the memory on its data input lines, the memory compares the input data to the words stored within itself. If an input word matches a stored word, the memory asserts a "match" signal for the location containing that stored word. The associative memories used in the KI10 also have linear-select read and write addressing identical to that of a conventional linear-select memory. Figures 6-4 and 6-5 respectively show the detailed structure of a single bit cell and the interconnection pattern among the bit cells. Figure 6-6 shows a 4-word by 5-bit associative memory with the word 10110 presented at the data inputs. An equivalence circuit E is activated when an input bit matches the corresponding bit of a stored word; in the figure, shading indicates the activated gates. When all bits of a word match, the wired-ANDs along the corresponding match line allow that line to be asserted, indicating that the particular stored word matches the input data. In the KI10, use of the Word Empty flag and careful control of the data written in the associative memory guarantee that multiple matches will not occur.

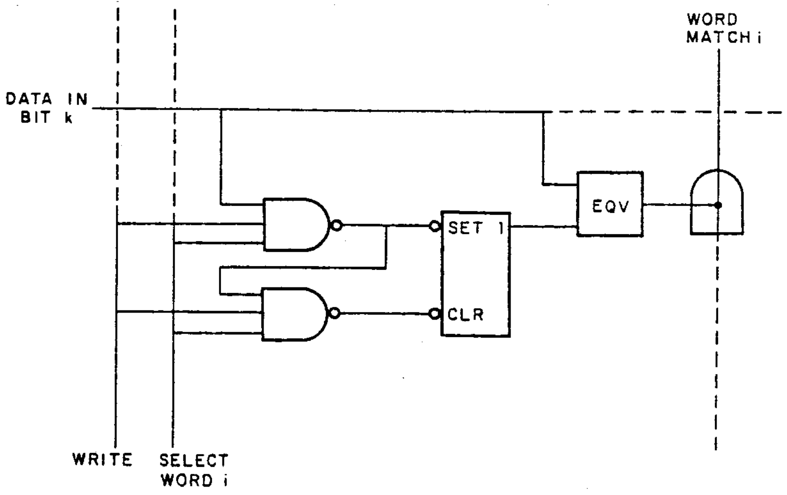


Figure 6-4 Typical Associative Memory Cell

Although the associative memory includes the entire page table and associated logic, the term "associative memory" used alone in the prints refers only to that part of the page table that contains the list of virtual pages. The eight associative memory modules shown in drawings AM1 and AM2 each contain four complete virtual entries. Every module constantly compares its data inputs against the entries it contains, and whenever the input data is equivalent to an entry the module asserts a match output for the corresponding location. The data inputs are the virtual page number on the address bus (AB 18–26), the USER PAGING signal to select the address space, and the Word Empty flag, which is always 0 when the processor is actually checking for a match. (The table entry that matches the inputs is available at the data outputs, but these signals are not used, nor are the bit-by-bit "force match" inputs.) When the AMAC WR pulse is generated, the input data is written into the location selected by the address decoders below the table modules. In this case Word Empty may have either state – 0 when data is actually being entered, 1 when a location is being invalidated.

The inputs to the decoders are produced by the nets at the top of drawing AMA. These signals reflect the state of the page table reload counter AM ADR when an entry is being made in the table or a single location is being invalidated because of a page failure. But a DATAO PAG, invalidates the entire table through the control logic at the lower right in print AMAC. When the paging hardware is selected, the DATAO clear and set signals generate equivalent signals for AMAC. AMAC DATAO CLR sets AMAC CLR ALL, triggers the AMAC write signal, and sets PF WORD EMPTY (which is in the upper right corner of drawing PF). The 1 state of AMAC CLR ALL asserts all of the address inputs to the decoders so the write signal invalidates all locations. The subsequent set signal clears both flipflops.

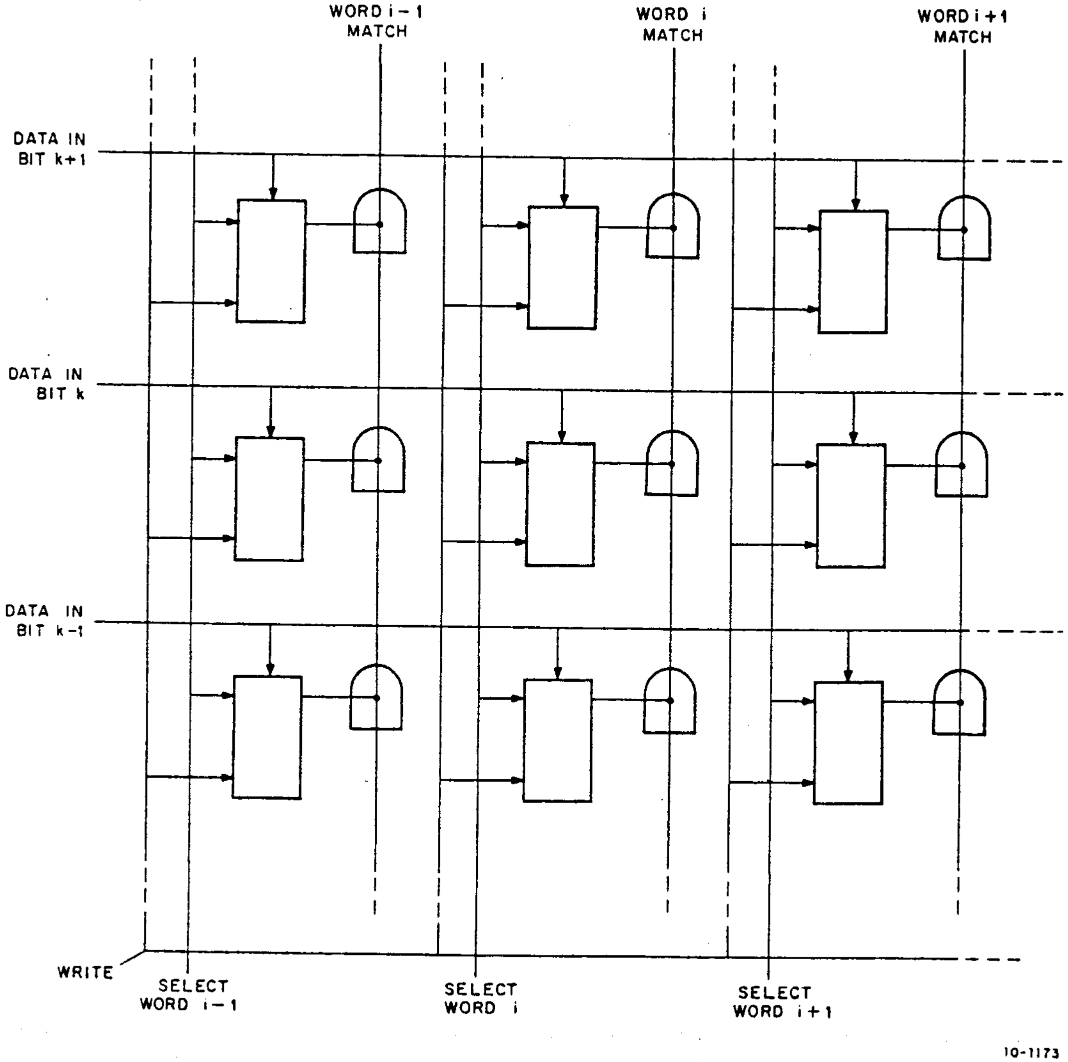


Figure 6-5 Associative Memory Cell Interconnections

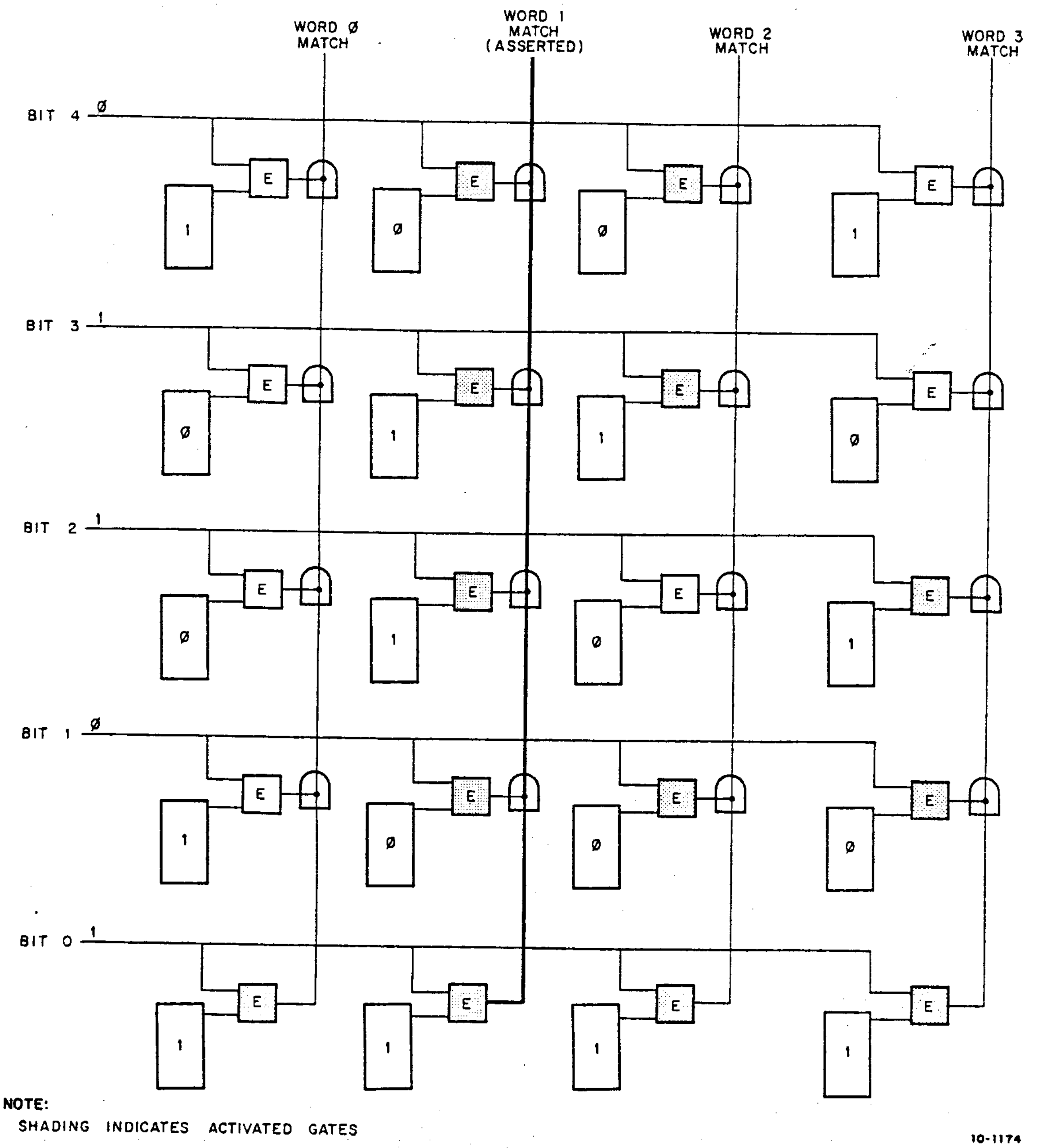


Figure 6-6 Example of Match

The part of the page table that lists the physical map data is the associative memory scratch pad. The two sets of four modules at the top and bottom of drawing AMSP each supply sixteen words of physical map data (four bits per word per module) for the sixteen virtual-page AM entries as selected by the match signals. When a refill writes in AM, the data being written produces a match, which in turn selects the corresponding AMSP location so that a second write pulse loads the physical map data into AMSP. The outputs for page checking and supplying physical address bits to MA are at the tops of the AMSP modules. The inputs at the bottoms of the modules come from the mixer at the top of drawing AMB. Following a refill cycle this mixer receives a half word of page map data from the left or right half of MB according to the state of AB 26, which is the least significant bit of the virtual page number. The scratch pad entries are only sixteen bits, as one bit of the page map data is not used, and the access bit is not kept in the scratch pad for no entry is made at all if the page is inaccessible.

The circuits at the bottom of AMB supply buffered match signals to the gates in the lower half of AMA. These gates determine if there is any match and also encode the number of the match location in binary to place it in the reload counter if there is a page failure. The large net at the left in drawing AMAC determines when a matched entry is in the location specified by the reload counter.

The remaining circuits in AMAC control the operation of the associative memory and in most cases the details of this operation are treated in the discussion of the memory flow. The chain of write pulses at the bottom controls the entry of new data into the table following a page refill cycle and also clears the match location when there is a soft failure. For a hard failure the Monitor will give a DATAO PAG, which clears the associative memory. Loading and counting of the reload counter is controlled by AMAC CLK, which is generated by the net at B5. Following a soft page failure, the clock loads the location of the matched entry (or clears AM ADR if there is no match). Should the counter point to the matched location in a paged reference that is alright, AMAC EQUAL REF is set allowing MCT0 to set AMAC AM+1. With this last flipflop set, the next time the address bus is latched (for a new memory access) the AMAC clock increments the reload counter by loading the address supplied through the +1 gate at the left in AMA. And finally a CONO PAG, sets AMAC IOB so that the CONO set pulse generates the clock to load the counter from bits 31–35 of the output conditions.

6.4 PAGE CHECKING

The circuits that determine whether a reference is paged and, if so, whether or not there is a failure are shown in drawing PAG1 and the lower half of PAG2. Almost all of this hardware effectively forms a large gating network through which level transitions perform the page checking during the page delay in memory control. At the completion of the delay, the signal MC PAGE DLY OVER determines what memory control shall do next depending upon whether the result is alright to make a reference, there is a failure, or a page refill cycle is necessary (see chart MC2).

The large net in the center of PAG1 generates the PAGE FAIL signal. Note that the top input to this net is PAGE ILL ENTRY. The negation of this signal appears in many other gates for determining whether a reference is paged, whether there should be a refill, and whether the result of the page check is alright. The reason for this is that an illegal entry is determined in the instruction cycle on the basis of the instruction fetched, and it produces a page failure in the next memory subroutine even if that subroutine taken by itself is perfectly alright (*eg* even if it references fast memory). In particular the bottom gate for PAGE OK indicates that with no illegal entry waiting, any special reference is alright, and a reference to locations 0–17 is alright if it is to fast memory or if there is a match when it is a shadow reference.

The net in the upper right in PAG1 determines various characteristics of an executive reference. A reference is executive if USER PAGING is false, the location referenced is not in fast memory, and it is not a special reference to the process table. An executive reference is paged or unpaged as determined by memory address control through the signal MA EXEC UNPAGED. If the reference is paged and there is a match, then the page is regarded as found. A similar net for user paging is at the lower right in PAG2. Here there is no unpaged area (except fast memory) and the reference is regarded as paged provided there is no small user violation.

The net at the lower right in PAG1 determines that a reference (regardless of mode) is paged if it is either a user paged reference, an executive paged reference, or a reference to the shadow area. If the reference is paged and there is no match, a page refill is called for. This signal allows the termination of the page delay to set up the flipflops in the lower part of PAG2 to indicate that the memory cycle is for a page refill, to supply a refill address to MA, and to indicate that it is a refill for synchronous access if MC ASYNC START is 0. The signal PAGE REFILL RESTART allows the completion of the cycle to trigger the operations that adjust the contents of the associative memory and begin a new page check. If upon completion of the write sequence in associative memory there is still no match, PAGE REFILL ERROR sets (PAG1 C2) causing a page failure and preventing a second page refill in the same memory subroutine.

Illegal Entry and Address Failure. The large net at the lower left in PAG2 generates PAGE ILL ENTRY for either an illegal entry to a concealed area or an address failure produced by conditions selected at the console. Only the single-stage part of the net (the second gate from the top at the right) is relevant to an illegal entry; as discussed in section 5.5, it is determined in the instruction cycle and causes a page failure in the next page check. The rest of the net is part of the detection of an address failure as discussed in section 5.6.1, and it produces a failure in the memory subroutine in which the failure conditions are satisfied.

Small User Violation. The net at PAG1 C7 generates PAGE SMALL VIOL if a small user program supplies an address in which bits 19–21 are not all 0. This generates a page failure if it is a user reference.

Private Test. The signal PAGE TEST PRIVATE (center PAG2) indicates the circumstances in which a reference to a concealed area can be determined to be a violation: a concealed reference is alright if the processor is in concealed or kernal mode or is executing an interrupt or key function. The test is not made in an instruction fetch, as an illegal entry cannot be determined until after the instruction is decoded. An executive unpaged reference generates PAGE OK if PAGE TEST PRIVATE is false, but produces a page failure if that signal is true. If PAGE TEST PRIVATE is true for an executive paged reference with a match to a nonpublic page, the reference is alright if the program is not attempting to write (the supervisor can read concealed executive information), but otherwise the reference fails. A user nonpublic page found with PAGE TEST PRIVATE true causes a failure if the Disable Bypass flag is set, as indicated by XCT PROT BYPASS being false.

Write Test. The net at the lower left in PAG1 specifies the various conditions in which a write test must be made. To test writing at all, there must of course be a match. In a public page there is a test for either a user or executive paged reference. The write test is also made in a user or executive paged reference if there is no private test or in a user paged reference where the protect bypass is in effect. The bottom gate of the PAGE FAIL net determines what references are for writing: any synchronous subroutine in which writing is called or the cycle is split. PAGE TEST WRITE produces PAGE FAIL if the reference is for writing and the page is not writeable, but otherwise it generates PAGE OK.

6.5 MEMORY CONTROL

To access a memory the processor must place an address and a request on the memory bus. A given memory responds only when it is addressed, but beyond that it responds only to the correct type of request. The memories in a system are of three types for which there are three types of request; these are categorized as slow, fast, and immediate. In general these catagories correspond to memory speed, but this need not be true in the case of an immediate memory. In strict hardware terms, the type classification of a memory is determined by the type of request to which it is set up to respond, and its timing characteristics must be adjusted to fill the requirements for that type.

Memory type is relevant only to the overlapping of memory read cycles on the bus. The overlapping of the other two segments in the memory control pipeline (page checking, MA loading) depends neither upon memory type nor upon the type of access, and there is no bus overlapping for write cycles. (There is no provision in the logic for overlapping a read cycle on a preceding write cycle, and since it makes no sense for an instruction to try to store a result before the operands have been received, the situation of overlapping a write cycle on a preceding read never occurs.) The relevance of memory type to overlapping read cycles hinges on the fact that cycles cannot be nested – data from two different memories must be received in the same order in which the requests were made (the limit of overlapping is two cycles). The basic criterion is that once a request is acknowledged by a memory that takes a given amount of time to send back data, a second request can be made immediately to another memory that takes the same amount of time, but a request to a faster memory must be delayed until it is certain that the data from the second memory will not arrive until after the data from the first. Timing for cycle overlapping is shown in Figure 6-7.

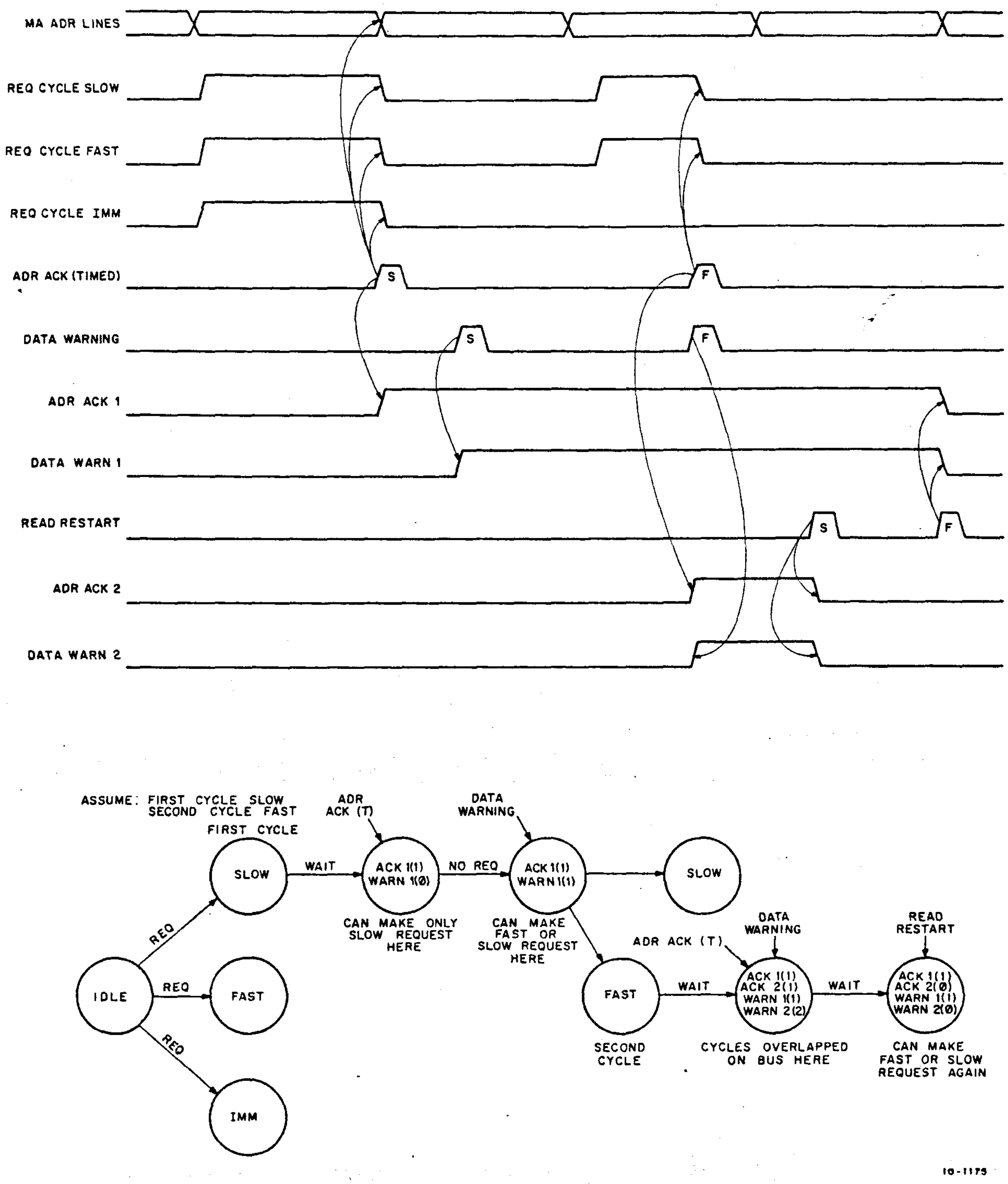


Figure 6-7 Memory Cycle Overlapping

Consider first the slow and fast memories, which are respectively memories with cycle times of about 1.8–2.5 μs and 1 μs. A slow or fast memory responds to the appropriate type of request (a request on the bus line to which the memory is connected) by returning a timed address acknowledgement and subsequently a data warning before sending the read restart. The timing for these signals is not fixed universally but it is fixed within a given system (*ie* for the memories connected to a single processor). The time from address acknowledge (timed) to read restart must be the same for all slow memories, the time from data warning to read restart must be the same for all timed memories fast and slow, and no fast memory can have a time from address acknowledge to read restart that is longer than that for the slow memories. (Memory timing specifications for a processor are posted on the outside of the bay 3 mounting door.) The meaning of the address acknowledgement is that the addressed memory has recognized that it is wanted and has taken the address from the bus; hence memory control can load a new address into MA and can request a second overlapped read cycle from a slow memory. The data warning means that the data will arrive soon enough so that a request can be made for an overlapped cycle to a fast memory. The read restart indicates that the data has been sent to the processor and the access is finished. The read restart does not mean that the memory cycle is finished, as the memory must write the word back into the addressed location. (In a read-modify-write cycle the memory waits instead of writing the same word, but this type of cycle is not under consideration, as writing is not overlapped and memory control also waits.)

An immediate memory may be one that sends data as soon as the request is made (within 200 ns). But as far as the hardware is concerned the definition of an immediate memory is simply one whose cycle cannot be overlapped with any other. The relationship between this and immediate data response is obvious: if the data will be returned immediately, no request can be made until the previous access is complete, and nothing is gained by trying to overlap a subsequent cycle. But this definition of immediate allows the mixing of compatible non-overlapped memories on the bus with fast and slow memories. Any memory designed to operate on the KA10 memory bus can be accommodated by connecting it to the immediate request line and using an adapter (section 6.7). An immediate memory does not send any data warning, and the time from its address acknowledge (not timed) to read restart is indeterminate.

When the processor makes a request, it has no way of knowing what type of memory is being addressed. So to make a request when there is no cycle currently in progress, the processor sends the request signal out on the fast, slow and immediate request lines simultaneously. If an address acknowledge (not timed) is received back, then the memory that has responded is either a real immediate memory or something masquerading in that guise, and the processor completes the access without overlapping. If address acknowledge (timed) is received back either a fast or slow memory has responded, and the processor can therefore attempt to overlap the next cycle. For this next cycle, memory control first sends out a slow request. If no memory has responded by the time the data warning is received, memory control also places a request on the fast line. If no memory has responded by the time read restart is received, then there is no overlapping and a request is placed on the line for immediate memories. If no memory responds within 100 μs, it is assumed that the addressed memory does not exist.

Besides specifying the request type in terms of memory type, memory control must also specify the request in terms of the type of access: read, write, or both. Yet another request signal, which is not used with any present memory, is for sequential access. This request signal is true when it is quite likely that the next rerequest to the same memory will be made to the next consecutive address. A memory that would use this line might have a core stack four words wide. For a random access the memory would retrieve a set of four words which it would save in a buffer while sending one to the processor. Thus for the first access in a set, the memory would be fast or slow, but would be immediate for access to locations whose contents are already held in the buffer.

Do not confuse fast memory, *ie* "the" fast memory, with "a" fast memory, which is an external memory that responds to a fast request. "The" fast memory has the accumulators and index registers and can be referenced as ordinary memory for an instruction or an operand, but in terms of request type it is effectively an immediate memory. However since it is in the processor, memory control handles it directly without using the bus and it returns no address acknowledge. Note that throughout the reference manual and elsewhere in this manual the term "AC reference" refers to reference to fast memory for an accumulator as specified by bits 9–12 of an instruction word. But in the memory drawings, and hence in the discussion here, "AC reference" refers to referencing "the" fast memory from memory control, *ie* as ordinary memory.

6.5.1 Memory Subroutine Logic

The control and timing circuits for the memory subroutine are shown in seven MC logic drawings. Before discussing the flow of events that make up the subroutine, which requires six flow charts, let us go through the block schematics to see where everything is and how the more complex circuits function individually. Figure 6-8 is a simplified block diagram of memory control.

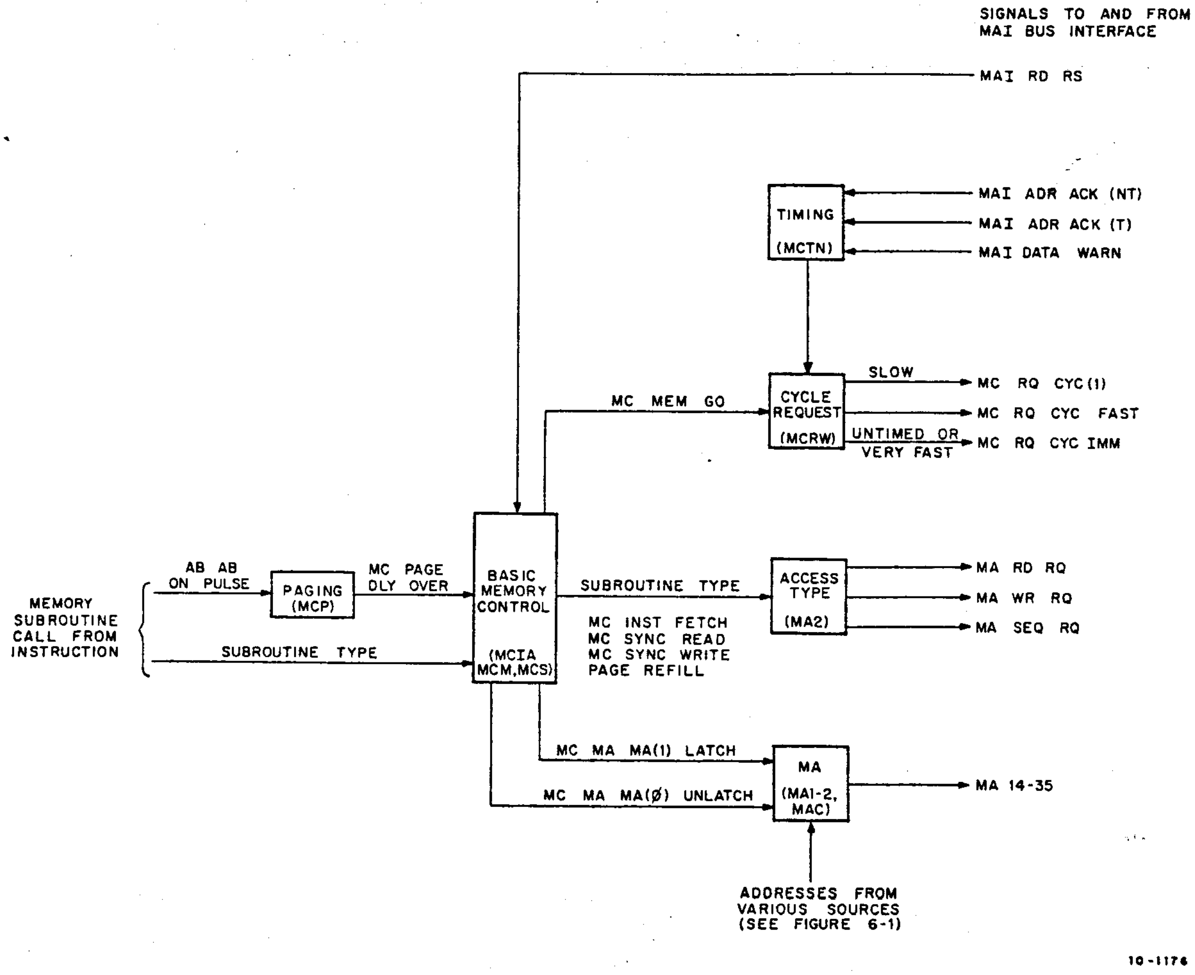


Figure 6-8 Memory Subroutine and Bus Control

The output of the net that takes up the entire lower half of drawing MCIA starts a memory subroutine to fetch a word that is handled by the instruction cycle, *ie* a program instruction, an address word, or an interrupt or trap instruction. Note that the net includes a flipflop that asserts the output during the time state following that in which the flipflop is set. MC INST FETCH EN starts the memory subroutine by setting both MC INST FETCH START and MC ASYNC START. For an automatic fetch MC INST FETCH START is set alone through the net just above the center of the drawing, and the (overlapped) subroutine starts automatically when the signal MCT AB AB SET from the previous subroutine sets MC ASYNC START. This same signal starts the asynchronous fetch of a second operand in a double operand fetch. Note that even if the clock is running, both start flipflops remain set until the first clock after the processor reaches a sync point (which stops the clock). In an instruction fetch, synchronization occurs when the preceding instruction is done; in a double operand fetch, synchronization for the second operand follows **ET2**.

The flipflop in the upper right is set for the fetch of a word that is to be handled by the instruction cycle but is not a regular program instruction. The conditions that set this flipflop also generate MC INST FETCH EN to start the subroutine, but the 1 state of MCI PSEUDO INST FET prevents the two start flipflops from generating MC INST FETCH at the right. This last signal indicates that the fetch is for a true program instruction.

The two flipflops at the upper right in drawing MCS indicate when a memory subroutine is for fetching or storing an operand. For a double operand fetch, the flipflop at top center causes the subroutine for the first operand to start a second subroutine automatically. The hold flipflops at bottom center indicate when a synchronous cycle is actually to be held over the bus – in other words the cycle will not be voided because of a page failure or skipped because it is an AC reference. Whenever a synchronous read cycle is held, the gates at C7 produce a signal that loads the word fetched into AR as well as MB. MC STORE IN AC at the lower left is for use by the store cycle in determining whether to restart the memory subroutine or handle AC storage itself. The remaining logic in the upper left is for calling a split cycle, *ie* separate read and write cycles to the same location in a single instruction (with only one page check). A long instruction that calls for both FCE and SCE always produces a split cycle, but a read-modify-write cycle is also split if there is any possibility of the processor stopping between the read and write parts.

The logic at the top and center of drawing MCP handles the timing for parity checking and controls the parity stop on error. The rest of the logic controls the delay for page checking (just above the center is the synchronizer for advancing from the page check to loading MA). The logic in the left two thirds of print MCM handles the loading of MA and the setting of MC MEM GO to make the memory subroutine actually call for a memory cycle. The inhibit flipflop at the bottom is set in various instruction situations to prevent the memory call for writing if there is likely to be a considerable wait before the store cycle gives the write restart (supplying the data to be written). Most of these situations involve input instructions that may have to wait for the IO bus (MAP acts like a write instruction but no memory call is made at all). When the call is delayed by inhibiting MEM GO, the write part of the subroutine must later be started by setting the flipflop at bottom center in print MCRW. This is done prior to the write restart so that the memory is already waiting when the processor has the data ready. Note that single pulse operation enters into the delay of the memory call as well as into the split cycle already discussed. The reason for this is that the processor has no way of knowing the time between clocks (nor indeed if the next clock will even occur) in single pulse operation, and therefore cannot afford to hang onto a memory while waiting for the write restart. Hence for writing, single pulse operation usually inhibits MEM GO (the situation for read-modify-write or both FCE and SCE is handled by split cycle) and also delays starting the write subroutine until the same clock that generates the write restart.

The rest of the left two thirds of print MCRW has the various restarts: to restart the processor after data has been fetched, to indicate the completion of a refill cycle, to restart the write cycle when data is available for writing, to restart the processor after writing or a memory stop, and to simulate the processor restart when a write cycle is killed as in MAP or a division that cannot be performed. At the right are the cycle requests for the various types of memories; MC REQ CYC(1) always produces a slow request and produces a fast or immediate request when appropriate conditions occur. The requests for access type are placed on the bus at the same time MA is loaded as shown at the left in drawing MA2. Here are the latches for read and write requests, a sequential request, and an AC reference. Note that in a split cycle the gate at the bottom controls a signal that inhibits the write request during the separate read cycle and then switches over the latches from read to write.

The memory stop logic is at the upper right in drawing MCM. The upper flipflop synchronizes the parity stop and address stop switches to the memory subroutine; the lower flipflop actually stops the processor in the memory subroutine when a situation specified from the console actually occurs.

Timing. Drawing MCTF shows the separate timing chain for an AC read reference and the flipflops that control the destination of the data read. At right center in drawing MCTN is the logic for determining that the addressed memory does not exist, at bottom right is the timing chain through which the memory subroutine triggers an automatic second subroutine that overlaps the first, and at C5 is a flipflop that indicates an untimed (nonoverlapped) cycle.

The state of memory control is its position in the memory subroutine as shown in the flow charts, and with overlapping memory control can be in two positions simultaneously. The state vis-a-vis memory cycles over the bus is determined by the four flipflops at the lower left in MCTN. The left pair keeps track of the number of timed address acknowledgements that have been received without receipt of the associated restart (in other words the flipflops count the number of cycles in which the position of memory control is between address acknowledgement and restart). The right pair performs the same function for data warnings. If both left flipflops are clear, there is no fast or slow memory cycle currently in progress, although a request may have been made. The receipt of address acknowledge (timed) sets MCT ADR ACK 1. If a restart arrives before a second acknowledgement, the flipflop clears and no overlapping occurs. But if a second acknowledgement arrives first, the second flipflop is set. The next restart then clears the second flipflop leaving the first alone, and finally the second restart clears the first flipflop to again indicate that no cycle is in progress. As shown at the top of the drawing, an immediate request can be made when MCT ADR ACK 1 is clear, and an overlapped fast request can be made after MCT DATA WARN 1 is set. Delay lines provide for establishing a minimum time between the condition that allows the request and the address acknowledge from memory.

6.6 MEMORY SUBROUTINE

The events in the memory subroutine are shown in a set of six MC flow charts. The first three show the main part of the subroutine, from the call to the response by memory, which is applicable to all types of requests. MC4 shows the read return for subroutines that fetch a word (including a refill for the page table), MC5 shows the write restart in the store cycle, and MC6 shows the special sequence for an AC read reference (AC writing is handled directly by the store cycle and is shown in that flow chart). Figure 6-9 shows the memory pipelining and bus overlapping wherein an automatic instruction fetch follows an operand fetch.

6.6.1 Subroutine Call and Page Delay

Entries to the memory subroutine from various instruction and other special situations are in the upper half of chart MC1. The top row is for operand entries including key and PI functions. The second row is for instruction type entries and is divided into four groups. From left to right these are an instruction fetch that will start automatically following the page check in a subroutine already in progress or just beginning, an instruction fetch that will begin in the next time state, the standard instruction fetch or prefetch, and a pseudo instruction fetch for an address word or a trap or interrupt instruction. The actual initiation of the subroutine for the first of these four is the recycle entry (at the very center of the drawing), which comes from a later point in the subroutine. The other three entry groups trigger the subroutine by generating MC INST FETCH EN, which sets a pair of start flipflops. The actual beginning of the subroutine is AB AB EN, which is produced both by MC INST FETCH EN and by all of the operand entries in the top row. At this time the parity and address stop switches are synchronized to the subroutine. Setting AB AB generates a pulse that clears the AB flags after the bus is latched and triggers a delay for paging. The delay is also triggered by the completion of a page refill cycle to reenter the subroutine for the actual memory reference. Following the page delay the entire subroutine can be aborted if it is for an instruction fetch and an interrupt or key function has been synchronized; in other words memory control does not bother to fetch the next instruction if it is already known that it will not be performed anyway. If the subroutine is to continue, MC PAGING RDY is set, but this is done after additional paging time if MC PAGE SLOW EN has been set indicating an operation (a key or PI cycle or an instruction executed by an executive XCT) in which there may be a switch from one address space to another. At this point flow may continue to the AC reference section, which is described separately below.

MC PAGING RDY indicates that paging is complete, and it is one element in a synchronizer which may wait to continue to the loading of MA. The middle condition in the synchronizer is generally not relevant as it applies only to single pulse operation, where the sequence does not continue with a synchronous subroutine until the clock stops or with an overlapped asynchronous subroutine until the preceding read is finished. The left term is simply that MA has been unlatched, indicating that the address it held is no longer needed. This is usually the case once the address acknowledge is received, but if there is a possibility of a memory stop, the address is held longer for display to the operator should the stop actually occur. Hence MA is unlatched through the top pair of entries if the cycle is simply

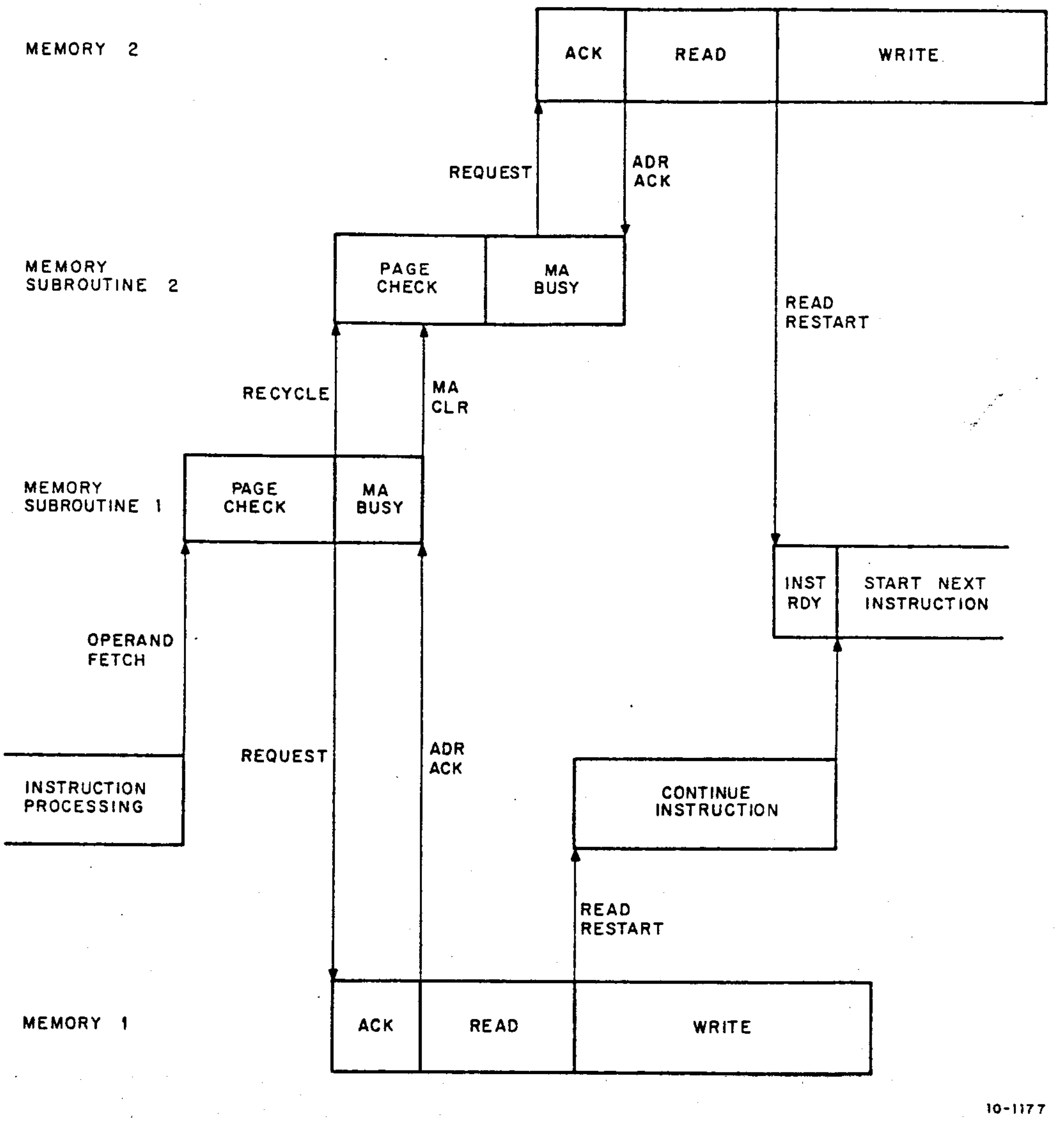


Figure 6-9 Pipelining, Overlapping, Prefetching

for a page refill, or if there can be no stop and either there is only one access to memory, or if the cycle is split then the current access is for write. The next two entries are for unlatching at the appropriate restart after the stop has occurred or would have occurred; the bottom two are for a fast memory access where no cycle actually takes place over the bus; and the final entry is for an access that is voided after it has been set up. Once the synchronizer is satisfied, there is a delay to make sure MA is unlatched, and the pulse MC PAGE DLY OVER continues the subroutine to the page check section.

6.6.2 Page Check, Memory Go, and Recycle

At the top of chart MC2, the termination of the page delay switches the fast memory to accumulator access if the processor is in FT6, holds the condition that the subroutine is neither for an MAP nor a key memory reference, allows for a possible parity stop if the switch has been synchronized and the subroutine cannot hold memory, and if the addresses compare, sets up flags in the comparison logic for the type of reference. From MC PAGE DLY OVER, flow continues along one and only one of the three main branches for page refill, page OK and page failure, and continues on one or more of the minor branches as well. Consider the minor branches first. Events along the leftmost branch simply reload PC for any instruction except one being executed by an XCT; this generally increments PC so it points to the current instruction, but in special circumstances PC+1 INH(1) may keep it constant. The branch at the middle of the drawing clears MB in preparation for receiving data except in a BLT or in an instruction or second operand fetch where a previous synchronous access is not yet complete. The next branch clears AR for an operand fetch, sets up the synchronous hold flags if a synchronous memory cycle will actually be called, and indicates whether there may be storage of a memory result in fast memory. In a memory subroutine for a key function, the address (perhaps incremented) is loaded into the address switches. The final branch at the right is for an AC reference (see below).

The major branches all depend on the result of the page check, and the paging equations written at the upper left, center, and bottom of the chart reflect the page checking procedure described in section 6.4. For a soft page failure, the memory subroutine produces a clock that loads the address of the failing associative memory word into the reload counter and a clock that loads the statistics on the failure into the page fail register (the equations at the lower right are described in section 5.3.5). At the same time MC PF QUICK sets PF SYNC to start the page fail cycle, and if the failure is neither hard nor for an MAP condition, the AMAC write sequence is used to invalidate the page table location containing the word responsible for the failure.

The two remaining major branches, for page refill and page OK, begin with different actions but then join to continue with the memory access that is necessary in either case. If there has been no match, the subroutine enters a page refill cycle, sets up the page table to receive a new entry, and waits while a page map address is substituted for the given virtual address. If the page is alright, the subroutine restarts the clock if the page check was made only for subsequent writing, sets a flag in the AMAC logic to increment the reload counter later if it now points to the matched location, and sets MC MEM STOP if an address stop condition has been satisfied. For either branch the address is loaded into MA. Then MCT0 clears PC+1 INH on a true instruction fetch, increments the reload counter on an equal reference, and adjusts PAGE PRIVATE INST depending upon whether the cycle to be requested is to a concealed area provided it is for fetching an instruction (the flag is left alone for any other type of reference).

Flow continues along this line only for a synchronous operand access when the subroutine is to restart automatically; there is also an entry at this point to restart the program following a page failure in a key memory function. Recycling is only for an automatic instruction fetch or the second of a double operand fetch, in which case MCT RECYCLE gates an address from the appropriate source into AB; subsequent pulses in the line set MC ASYNC START to indicate the type of subroutine and prevent a second recycle, and latch the address bus to begin an overlapped subroutine as shown at the center of chart MC1. The extra time following MCT RECYCLE for a second operand fetch is to make sure that the subroutine does not restart before the incremented address is available.

The remaining branch for page refill and page OK is to begin actual memory access. For a page refill, setting MC MEM GO is unconditioned; but for the real memory reference, the sequence continues only if MC MEM GO INH is clear. If the subroutine is stopped at this point, the write part is restarted later by one of the conditions that sets MC WR SUBR START FF shown entering from the right of the flow line. With the setting of MC MEM GO, the subroutine enters the synchronizer for requesting a memory cycle unless flow continues instead to the AC reference section (see below).

6.6.3 Request Cycle and Read Restart

Initial synchronization for requesting a memory cycle is at the upper left in chart MC3. For an actual memory cycle, MC REQ CYC is set immediately if there is no cycle already in progress or a cycle in progress is for reading with a timed memory. If the flag cannot be set immediately, it is set as soon as the current cycle is completed. Setting MC REQ CYC produces all three types of request if there is no overlap; otherwise it makes a slow request now, makes a fast request if no memory has responded by the time the data warning is received, and makes an immediate request if there has been no response by the time the previous cycle is finished. The response by the memory is the address acknowledge, either timed or not timed. Either of these clears MA to ready the synchronizer at the bottom of chart MC1 if access is for a page refill or there is no possibility of a memory stop and either there is no writing or the cycle is not split. A common acknowledge signal sets MCT UNTIMED CYCLE if the acknowledgement was not timed or overlapping has been disabled from the console, and for a write request it sets a flag indicating the write cycle is in progress in preparation for the write restart in the store cycle. If the ignore parity switch at the memory is set, a pulse indicating that fact arrives with address acknowledge for a timed memory but may arrive at any time up to the restart for an untimed memory. For a timed memory the data warning follows the address acknowledge to allow a fast overlapped request.

For a read reference with either type of memory the data is accompanied by the read restart, which triggers the parity check, performs certain functions having to do with an address comparison (the details of this are discussed in section 5.6.3), adjusts the acknowledge and warning flags, and sets a flag to indicate that MB must be cleared after the parity check if access is for an operand read only that will be followed automatically by an instruction or a second operand fetch. If the parity of a word read is even and parity is not being ignored, the parity check pulse sets the Parity Error flag, and it also sets MC MEM STOP if in addition a parity stop has been specified from the console. The parity check pulse also loads the word into the memory indicators if the address compare condition has been satisfied, and it clears MB if the read restart had set MCR MB POST CLR provided the instruction is not a BLT and there is no parity stop (for the latter case the bad word is saved). The subroutine ends at the bottom of chart MC3 if there is a memory stop, but if MC MEM STOP is clear, MCRST1 continues the subroutine to the read return section. MCRST1 is triggered directly by the read restart if there can be no parity stop, but it is otherwise triggered by MC PAR CHK (provided of course that either the parity is correct or errors are being ignored).

Nonexistent Memory. The left center of chart MC3 shows the events for determining that no memory has responded to a request. If the request stays on the bus for 100 μs, an integrating one-shot times out, clearing the request flipflop. If there is still no response within an additional 2 μs, the first NXM pulse sets the Nonexistent Memory flag and MC MEM STOP if stopping is requested from the console. For a write request, MCT ADR ACK ALL sets a flag to indicate the write cycle is in progress; for a read request, a second NXM pulse triggers MCRST1 unless the memory is stopping.

6.6.4 Read Return and Refill Reentry

Among the entries that trigger MCRST1 at the top of chart MC4 are three from the read restart section just discussed. However MCRST1 can also be triggered by completion of an AC reference, again provided there is no memory stop, or from the console following a memory stop in a read subroutine. MCRST1 switches the request from read to write for a split cycle, clears MCT UNTIMED CYCLE to allow further requests, and clears MA to allow the next page check if a stop was possible and there is no split cycle. If access was for reading a single or first operand, MCRST1 restarts the clock; but if access was for fetching an instruction or a second operand, it sets the appropriate flag to synchronize the instruction cycle or the continuation of the double floating point instruction.

The remaining logic is to reenter the subroutine if the access just completed was for getting a new entry for the page table. For this MCRST1 triggers MC REFILL DONE, which goes directly to the page fail cycle if the map data read indicates the page is not accessible. Note that the pulse always triggers the clock that loads possible failure statistics into the page fail register just in case there is an access failure (if there is not, the statistics will be changed if a failure of another sort does occur). For an accessible page, operations continue with the AMAC write sequence to put the new data in the page table (this sequence is also used for a page failure to invalidate the entry that caused the failure). The first write pulse not only writes in the associative memory but also sets PAGE MATCH LATCH if there is no match – a refill cycle is executed only if there is no match, whereas when the sequence is executed for a page failure there must have been a match that caused the failure. The second write pulse writes the map data in the scratch pad, and in the meantime the new information written in the associative memory should have given rise to a match that clears PAGE MATCH LATCH. It is instructive at this point to look at the latch, which is below the page refill error flipflop at the right on drawing PAG1. The first write pulse sets the latch if there is no match, but the subsequent occurrence of a match clears it. Thus the latch being set at AMAC WRITE DONE indicates that there was no match at the first write pulse and there is still no match, and this condition causes the final pulse to set PAGE REFILL ERROR. If the sequence is completing a page refill (rather than being used for a page failure), the done pulse reenters the subroutine for the actual memory access at B5 in chart MC1.

6.6.5 Write Restart

As shown in chart MC5, the write restart is triggered only from the store cycle, and it may have to wait until the write cycle is actually in progress. Usually the write part of the subroutine is started prior to ST2 DLYD, but in some cases, particularly for single pulse operation, it is not started until the clock stops. The write restart sends the data out on the bus and loads the memory indicators if the appropriate address condition is satisfied. The clock is restarted by MCRST0, which results directly from the write restart if there is no memory stop, but otherwise must be triggered from the console. Completion of the write is indicated by MC WR RS DLYD, and this completion is simulated by conditions that kill a write previously requested; these are the MAP instruction, a divide instruction that would store the result but cannot be performed, and a PI cycle just in case the interruption occurred at a time when a write had already been requested. Any of these conditions or a page failure produces MC WRITE KILL to clear MA for the next subroutine.

The completion of the write cycle produces the combination read-write restart to adjust the acknowledge and warning flags. MC WR RS DLYD also clears MB to get rid of the word written unless it is a pointer to be used in the second part of a byte instruction, clears MC WR CYC IN PROG to allow a new request, clears MA if a stop was possible and the subroutine was not a synchronous AC reference, and clears miscellaneous flipflops.

6.6.6 AC Reference

The separate sequence for a read reference to fast memory is shown in chart MC6. Entry is directly from MC PAGE DLY OVER (upper left) to read a single operand. For the first access with a double operand the sequence begins when both MC MEM GO ONCE and MC PAGING RDY are 1. The first of these is set by MC MEM GO for an AC reference, but at that time the second one is clear and does not get set again to start the sequence until the page check is complete for the second operand (this delay is necessary to prevent the fetch of the first operand from wiping out the address for the second before it is latched into AB). Entry for a second operand or an instruction fetch occurs after MC MEM GO is set, but not until ET2 in the double precision instruction or until the previous instruction is done so as not to interfere with the use of the arithmetic registers.

For all entries the sequence sets flipflops that cause the fast memory address to be taken from MA and cause the fast memory output to be loaded into MB. A single or first operand read also triggers MCF FM+ SET to clear the adder logic but set AD FM+ and MCF AR AD FF to gate the FM output into AD and thence into AR. MCT3 then clocks the data into MB and, if indicated, also into AR. MCT4 clears FM ADR MA to reestablish the previous FM address, and clears MA if there can be no memory stop and there is neither a write request nor a split cycle. At MCT4 DLYD there is either a memory stop or MCRST1 is triggered for the standard read return to the top of chart MC4. The remaining MCT3 and MCT4 DLYD events duplicate the address comparison actions of a standard memory access (see section 5.6.3).

6.7 KI10-M MEMORY BUS ADAPTER

The signals on the memory bus are different from those on the memory bus of a KA10 processor. It may however be desirable to connect a KI10 to some of the older memories used with a KA10. Up to 256K of KA10 type memory can be grouped at the far end of each memory bus cable provided a KI10-M memory bus adapter is inserted between it and the KI10 type memories closer in to the processor. The KA10 memories on a cable must be set up as a contiguous block starting at a multiple of 256K. (An adapter is also required for a DL10 that has old memories on its bus but drives a KI10-like memory bus, presumably because it needs to address over 256K.) The adapter can be mounted in some convenient cabinet along the cable or can be mounted at the bottom of processor bay 1 if there are no KI10 memories on the cable. In terms of loading, an adapter is equivalent to two memories, so the maximum number of memories must be reduced by two for each KI10-M.

The print set for the adapter includes a single block schematic KI10-M-MBA (two sheets). The left part of sheet 1 shows the logic for the bus signals that actually change, with the KI10 signals on the left and the KA10 signals on the right. Note that all KA10 memories are regarded as immediate, and the fast and slow memory requests are therefore terminated. Note also the delay at the lower part of D6. This delays the request signal to compensate for any skew introduced into the other signals by the adapter logic. The delay means that the adapter is itself equivalent to 11 feet (3.3 meters) of cable and the maximum length of the bus must therefore be reduced by that amount for each KI10-M. The rest of sheet 1 and the left half of sheet 2 show signals that go right through. The tables on sheet 2 list the corresponding pin connections for the two bus types.

CHAPTER 7
ARITHMETIC LOGIC

This chapter describes the several adders and the various registers that are used in computations. The basic arithmetic components that handle words in logical operations, data transfers and fixed point arithmetic (including effective address calculation) are the adder AD and full-word registers AR, BR and MQ. In these operations however the fast memory itself is used as a passive register: its outputs (being the contents of the addressed index register or accumulator) can be available directly to AD throughout an operation rather than being transferred first to an arithmetic register. In association with the full word registers, the shift counter SC controls shifting in shift instructions, byte manipulation, and where required in arithmetic instructions; SC, with its adder SCAD and the floating exponent register FE, are used for handling floating point exponents. Double precision floating point requires the use of ARX and ADX, which are left extensions of AR and AD (MB is also used as a temporary holding register for one part of a double precision number while the other part is being used or being constructed one bit at a time).

Although this chapter describes the registers, their mixers and the gating associated with them, the details of individual events and the particular configurations of individual shifts and data movements are included in Chapter 8 as part of the discussion of the individual instruction flows.

7.1 ADDER AD

The main adder for 36-bit words is the set of ten M142 adder modules shown in drawings AD1 and AD2. Each module is a 4-bit parallel binary adder, and the ten operate together in parallel through the use of extensive carry skipping circuitry. Each module produces carry generate and carry propagate functions that are dependent only upon the data inputs without making use of individual carries. The carry into the least significant bit (LSB) of any module, although equivalent to the carry out of the preceding module (for the next four less significant bits at the right), is not that carry physically; rather it is a function of the carry propagate and carry generate levels from all preceding modules. Hence the carry into any module is dependent only upon information available immediately without waiting for the generation of any other carries.

Within an M142 each of the four individual bit adders has two data inputs, A and B (supplied by mixers so that the data can come from any two of a number of sources), and a sum output. There is also a carry into each bit. The carry into the LSB is supplied externally, but the carry into any other bit *N* is a function of the LSB carry in and the data inputs to the bits at the right of *N* in the module. Each module also has two enable inputs, ADD and EQV, and the above mentioned carry generate and carry propagate outputs CG* and CP*, which are used to develop the carries into the LSBs of succeeding modules in place of individual carries out of the MSBs. When neither of the control inputs is enabled (both high) the sum output of each stage is the AND function of the data inputs (the sum is 1 if both inputs are 1, otherwise 0). If only the EQV input is enabled (low) the sum is the equivalence function of the data inputs (1 if the inputs are identical, otherwise 0). With both the ADD and EQV inputs enabled, the adder produces a standard sum output from the data and carry inputs. The sum is 1 if just one or all three of the inputs are 1. The carry in for a bit, although generated independently, is equivalent to the carry out of the preceding bit; the carry out of a bit is 1 if any two or all three of the inputs to that bit are 1.

Before discussing the internal logic of the M142 and the details of the carry functions, let us consider the overall structure of the adder. The least significant end is at the lower left in drawing AD2. Note that this module contains only three adder bits, AD 33–35. The carry into the module LSB is always 1, and the data inputs to this bit are identical, both being the 1 state of the AD CRY 36 flipflop. Hence setting AD CRY 36 produces a carry into AD 35; this carry is used to satisfy the requirements of twos complement arithmetic, or simply to increment a number supplied to the data inputs of the adder bits corresponding to the actual words processed.

Continuing from bit 35, the adder configuration is standard until we reach bit 18 at the lower right in print AD1. An extra stage, AD 17.5, is inserted between AD 18 and AD 17. This is done to allow independent operations on the left and right halves of a word. The data inputs to AD 17.5 are the 1 state of AD+1 LT and the 0 state of AD-1 LT. Ordinarily both of these flipflops are clear so the A and B inputs to bit 17.5 together provide a single 1; hence a carry out of bit 18 produces a carry out of bit 17.5 to carry into bit 17 – for normal full word operations the adder acts as though bit 17.5 were not even there. To handle the two halves of the adder independently, the processor adjusts the data inputs to bit 17.5. Setting AD-1 LT prevents a carry out of bit 18 from affecting bit 17, but at the same time that carry is indicated by a sum output of 1 for bit 17.5. On the other hand setting AD+1 LT with AD-1 LT left clear means that both data inputs are 1; hence there is automatically a carry out of bit 17.5 while a carry out of bit 18 is still indicated by AD 17.5 being 1. The AD 17.5 carry out may be used to increment a half word in the adder left half, or it may be used with one set of AD 00–17 inputs being all 1s to produce a simple transfer of the other set while an addition is taking place in the adder right half.

Typical situations in which these several configurations occur are as follows.

AD 17.5 Inputs			
AD+1 LT(1)	*AD-1 LT(0)*	*Situation*	*Effect*
False	True	Ordinary full word operations	Any carry out of bit 18 is passed into bit 17
True	True	Indexing	Generates a carry into bit 17 to suppress all 1s (-1) from idle left half of input mixer
True	True	Compute BLT word count at ETO	Same as above to load S, but also a bit 17.5 sum of 1 indicates a carry out of bit 18 – *ie* E < D
False	False	Increment byte pointer	Prevents pointer overflow from carrying out of bit 18 into bit 17 (the AC address)

The remaining stages at the left are connected in the normal fashion including the sign bit as required for twos complement arithmetic. However at the left of the sign are two additional adder stages, AD -1 and AD -2. These extra stages are configured to supply the correct bits for right shifting in certain operations (as determined by the input gating to AR bits 0 and 1). The A inputs to the extra bits are identical to the A input to AD 00. The B inputs are generated only when the extra bits are relevant to the operation going on in the rest of the adder, in which case they are identical to the AD 00 input except in the operation that doubles BR, in which case they are both equivalent to BR 00 (in other words they receive the sign in what is effectively a left shift of BR). Of course the AD -1 and AD -2 outputs depend not only on these extra inputs but also on the carry out of bit 0.

Although the carry functions of the adder bits are available at the module pins, the only ones that are used elsewhere in the logic are those that represent the carries out of bits 0 and 1. Note however that the carry outputs from the M142 in the upper left of print AD1 are not the signals actually used to represent the carries in the processor logic nets. The adder outputs are ored with FDT2(1) by the nets at AD2 B3 to generate the carry signals that are used. In other words AD CRY 0 and AD CRY 1 do represent the carries out of the adder, but these carries are also simulated by a floating divide time state for the initial subtraction in the divide sequence.

Adder Functions. The detailed organization and internal structure of the adder module are shown in the two sheets of drawing D-CS-M142-0-1. This drawing follows the block schematics in the Customer Print Set and is also in the replacement schematics. At the top of sheet 1 is the module configuration for use with low inputs and low sum outputs. Any low signal into the OR gate at the right indicates a carry into the module LSB, and the outputs at the left are the carry propagate and carry generate functions. In the upper right corner are the equations for the various carry functions. The carry into the LSB is simply the externally supplied carry input. For any other bit the carry in is 1 if both data inputs to the preceding bit are 1s, or at least one of those data inputs is 1 and the data inputs to the bit preceding that are both 1s, and so on through the module. The propagate function is true if there is at least one 1 among the data inputs to every stage in the module; in other words if there is a carry into the module, it is propagated through so that there is a carry out of the module. The generate function represents a carry out of the module that stems from a carry produced within the module; this means that either the data inputs to the MSB produce a carry (are both 1s), or the data inputs to some other bit produce a carry and that carry is propagated to the left through the module. Examples of input bit configurations that produce the various combinations of these functions are given in drawing B-SP-KI10-0-CPCG, which is near the back of the Customer Print Set. Note that the table at the left of the equations indicates that the sum outputs provide an inclusive or exclusive OR function when the add is not enabled.

Sheet 1 also shows an adder configuration for high inputs, where the other functions for the sum outputs are AND and equivalence. The carry equations for the high input case are listed at the right, but now the propagate and generate functions, although correct and productive of the desired results, no longer have any intuitive meaning. Sheet 2 shows the internal gates of the M142 configured for the low input case. It is relatively easy to see that the non-add functions change from OR to AND and exclusive OR to equivalence when the inputs are changed from low to high. But for the add function the equations are cumbersome and working with them is exceedingly tedious.

Now the adder modules as shown in the block schematic are used with high inputs and high sum outputs, but it should be noticed immediately that there is a subtle difference: the module carry functions are shown in the polarities for the low input case, where CP* and CG* are the carry propagate and carry generate functions given in the upper right of the M142 drawing. For non-add functions the reader should regard the modules as being used in the high input configuration so that the functions produced are AND and equivalence. But for addition the reader should regard the modules as being used in the low input configuration with the polarities of the inputs and sum outputs reversed. This point of view makes it quite easy to see how the carry skipping logic works. Since a carry out of a module must either be generated within the module or be propagated through it, the carry out (which is the carry in for the next module) is represented by this equation,

$$\text{CRY OUT} = \text{CG*} \vee (\text{CP*} \wedge \text{CRY IN})$$

where CRY IN is the carry in for this module. But the carry in is simply the carry out from the preceding module, so that term can be replaced by an equivalent function for the preceding module, and the same term in that function can also be replaced, and so on to the right through the length of the adder. Hence for any of the ten M142s, the carry in is a function only of the data inputs at the right, and through the use of ever larger logic nets, all of these functions can be determined with minimum gate delay. Figure 7-1 shows a simplified version of this arrangement for an adder of four bits.

Consider the right end adder module at the lower left in print AD2. Since the data inputs to bit 36 are both the same signal, the permanently wired carry into bit 36 does not appear in the equations, and the artificial carry into bit 35 for incrementing appears in the carry generate function. Hence a carry into the second module from the right is simply the CG* function from the first module. There is a carry into the third module if either a carry is generated in the second module or the first module generates a carry that is propagated through the second module. And the carry functions are extended similarly right through the entire adder. The logic nets at the right in print AD2 and scattered all over AD1 produce the necessary functions entirely in parallel so that there is at most two gate delays even for the carry in at the left end of the adder.

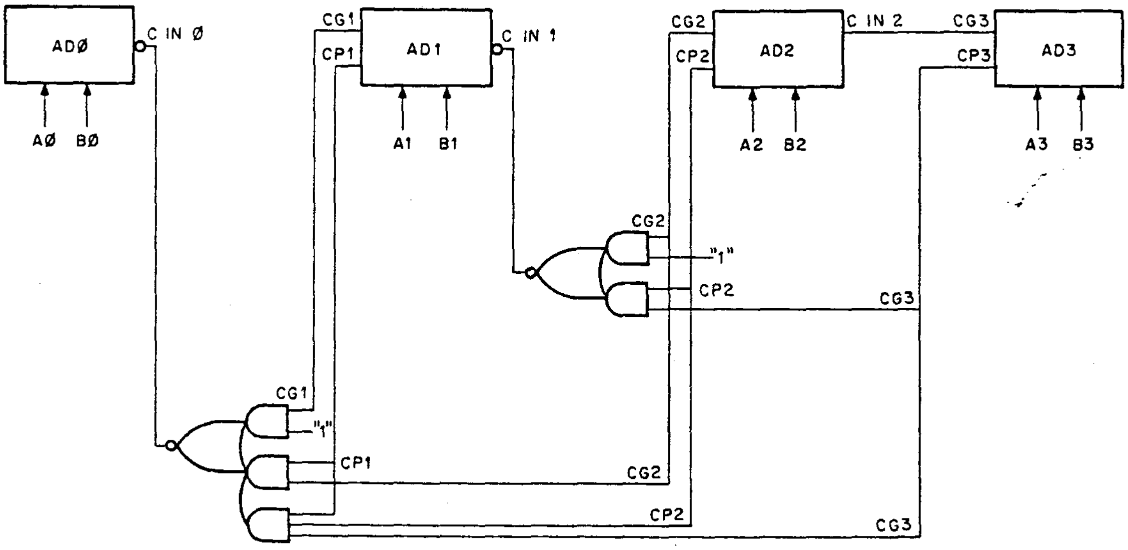


Figure 7-1 Carry Skipping Adder, Simplified

Now that we have seen how the adder works there remains the question: assuming the adder always produces a correct sum in its standard configuration, does it still do so when the inputs are of the "wrong" polarity? Consider the entire adder as a single 40-bit unit. Without a carry into the LSB, if we supply two summands, A and B, at low assertion levels, then the adder produces an output S, also at low assertion levels, which we claim is the sum $A + B$. Since we are using high assertion levels, the real inputs and outputs are actually the complements of the expected inputs and outputs, so we can represent the action of the adder by the equation

$$\sim S' = \sim A + \sim B + C$$

where C is the carry into the LSB and S' is the adder output at the correct assertion levels. The question of course is: what is the relation of S' to S? In twos complement arithmetic the negative of a number is the complement plus 1, so

$$\sim X = -X - 1$$

and we can rewrite the original equation this way:

$$-S' - 1 = -A - 1 - B - 1 + C$$

Since a carry into the LSB is equivalent to adding 1, we get

$$-S' = -A - B$$

or by changing signs,

$$S' = A + B$$

Hence $S' = S$, and the adder gives the desired result. Note that with this adder configuration, the addends and sum are complements only in terms of the internal adder circuitry; in terms of the overall logic in which the adder is imbedded, they are just binary words using appropriate assertion levels.

Addition Algorithms. So far we have shown that the adder adds single bits correctly, that the carry lookahead logic works, and that if the entire adder produces a correct sum in the normal configuration for the circuit, it also does so in the configuration we are using. Now let us consider the result of adding any pair of signed numbers. Calculations are performed as though the words represented 36-bit unsigned numbers, *ie* the signs are treated just like magnitude bits. In the absence of a carry into the sign stage, adding two numbers with the same sign produces a plus sign in the result. The presence of a carry gives a positive answer when the summands have different signs. The result has a minus sign when there is a carry into the sign bit and the summands have the same sign, or the summands have different signs and there is no carry.

Thus the program can interpret the numbers processed in fixed point arithmetic as signed numbers with 35 magnitude bits or as unsigned 36-bit numbers. A computation on signed numbers produces a result which is correct as an unsigned 36-bit number even if overflow occurs, but the hardware interprets the result as a signed number to detect overflow. Adding two positive numbers whose sum is greater than or equal to 2^{35} gives a negative result, indicating overflow; but that result, which has a 1 in the sign bit, is the correct answer interpreted as a 36-bit unsigned number in positive form. Similarly adding two negatives gives a result which is always correct as an unsigned number in negative form.

For convenience let us take the computer representation of the positive number x as $+[x]$ where the brackets enclose the number in bits 1–35. Similarly the representation of $-x$ is $-[2^{35} - x]$ or $-[1 - x]$ depending on whether we are regarding numbers as integers or as proper fractions. The most negative number, -2^{35}, has the form $-[0]$, which is equivalent to the unsigned integer 2^{35}.

There are four cases of addition of two positive 35-bit numbers, x and y.

I. $x + y$
II. $(-x) + (-y)$
III. $x + (-y), \quad x \geqslant y$
IV. $x + (-y), \quad x < y$

When the signs of the operands differ, one must diminish the other; hence in III and IV the magnitude of the result must be less than the larger operand magnitude. This means that overflow can occur only in I and II. As is shown in the proof below, overflow occurs when the carries out of bits 0 and 1 differ, and it is this condition the processor uses to detect the overflow. For convenience in the exposition we shall regard the numbers as proper fractions; to view them as integers, simply substitute "2^{35}" for each occurrence of "1". Since the twos complement format allows a representation for -1 but not $+1$, either x or y may be 1 in II, and y may be 1 in IV. But note that even though the y operand is negative in III, y cannot be 1 as its magnitude must be less than or equal to that of x. Subtraction is equivalent to addition, as the processor simply supplies the complement of the subtrahend to the adder with AD CRY 36 set. Let x be the absolute value of the minuend and y b4i absolute value of the subtrahend. There are again four cases, which correspond respectively to the four cases of addition given above.

I. $x - (-y)$
II. $(-x) - y$
III. $x - y, \ x \geqslant y; \ (-x) - (-y), \ x \geqslant y$
IV. $x - y, \ x < y; \ (-x) - (-y), \ x > y$

Again any negative operand may be -1 provided it does not violate the given magnitude restrictions.

I. If $x + y < 1$ the adder output is $+[x + y]$. If $x + y \geqslant 1$ the carry out of stage 1 changes the sign. Consequently if the addition of two positive numbers gives a negative result, it is apparent that the sum exceeds the capacity of the adder. The processor detects the overflow by checking the sign carries: there is a carry into the sign stage but none out of it. AD then contains

$$-[x + y - 1]$$

II. Ignoring the carry into the sign bit in the addition of two negatives would give

$$\begin{array}{l} -[1 - x] \\ -[1 - y] \\ \hline +[1 + 1 - x - y] \end{array}$$

If $x + y \geqslant 1$ the carry changes the sign and the result is

$$-[1 - x - y]$$

which is the representation of $-(x + y)$. If $x + y > 1$ there is no carry into the sign, and its absence in the presence of a carry out indicates overflow. AD contains

$$+[1 - (x + y - 1)]$$

III. Ignoring the carry into the sign in an addition where the signs are different would give

$$\begin{array}{l} +[x] \\ -[1 - y] \\ \hline -[1 + x - y] \end{array}$$

Since $x \geqslant y$, it follows that $1 + x - y \geqslant 1$. Hence the carry changes the sign and the result is

$$+[x - y]$$

When the operand signs are different, the magnitude of the result cannot exceed the larger operand magnitude and there can be no overflow. Since in this case the positive number is at least as large in magnitude as the negative, there is always a carry into the sign, and this added to the operand minus sign produces a carry out.

IV. The addition of numbers of differing signs where the negative has the larger magnitude gives

$$\begin{array}{l} +[x] \\ -[1 - y] \\ \hline -[1 + x - y] \end{array}$$

Since $x < y$, then $1 + x - y < 1$. Hence there are no carries associated with the sign and no overflow. The above result is the twos complement representation of $x - y$, *ie* $-(y - x)$.

7.1.1 Adder Mixers

The A and B input mixers for the adder are shown in drawings ADA1, 2 and ADB1, 2. Each bit of either mixer is an OR gate with AND gates for inputs. A pair of high inputs at any AND gate produces a low output from the OR gate. Since this low output represents a 0, a high output representing a 1 is produced when no input AND gate to the bit is satisfied. Hence when no enabling level is asserted for a mixer, the mixer output is all 1s; and when a single enabling level and all of the bit inputs associated with it are true, the mixer produces all 0s. The reader should keep these two facts in mind when investigating adder operations in the flow charts. The adder AND function is equivalent to a simple transfer for the output from one mixer if the other mixer supplies all 1s. Similarly an addition is equivalent to a simple transfer when one mixer supplies all 0s. This structure of the mixers also requires that individual bit inputs have the state opposite that called for by the enable. In other words to enable a register through the mixer requires that the bit inputs be of the state opposite that being enabled. For a direct transfer from a register, the set of AND gates (one per mixer bit) for the enabling level receives the 0-state signals from the register bits, so that when an individual bit signal is true the adder output is low representing 0. Conversely a false 0-state signal at an enabled AND gate produces a 1 output (high). The enable for the complement of a register is combined with the 1-state signals from the register.

Unlike a mixer for input to a register, an enabling level for adder input is a flipflop; the level is therefore true during the time state following the state in which the enable condition appears in the flow charts. Hence setting an adder enabling flipflop at a particular time pulse gates the mixer output into the adder during the next time state, so the appropriate adder output is available at the time pulse that terminates that state.

Consider the adder A input mixer shown in drawings ADA1 and ADA2. Here the 0 states of MB and AR bits are combined with the enables for MB and AR+, and the 1 states of AR are combined with the level for AR-. The last set of inputs combines the ~MSK GEN bits with AD MSK GEN(1) to generate a mask when one is wanted, otherwise to supply a zero word. Enabling AR+ and AR- simultaneously also supplies a zero word. At the top of ADA1 are duplicates of mixer bits 0 and 1.

Drawings ADB1 and 2 show the larger B mixer, which supplies inputs to the adder from fast memory (FM) and its complement, BR and its complement, and the in-out bus. The input generated by AD BR+2 is actually twice the contents of BR, *ie* BR shifted left one place with a 0 supplied to AD 35. The last enable is for the magic numbers. This level gates in the signals identified by the label MAGIC (which are often disabled and therefore zero) but at DFDT9 supplies a word made up of a 1 in bit 0 followed by all 0s, and in an LUUO supplies a 1 in bit 30 for address 40 or 41. Like the A mixer, the B mixer also has two extra bits shown at the left in drawing ADAM. These are inputs for the -1 and -2 adder stages, which are used only in operations involving BR (although the two are held at 0 for the magic numbers). When BR+ or BR- is enabled, the extra bits duplicate mixer bit 0; for the doubling of BR the extra bits are equal to BR 00 (at this time mixer bit 0 receives BR 01).

7.1.2 Adder Gating

Although various instruction conditions combined with time states do generate enabling levels in the adder logic, these are the enabling inputs for control flipflops whose outputs enable the mixers in the next time state. Since the adder is used so frequently (it is changed in practically every time state) the clock for the adder control logic is triggered from the main clock (center, drawing ADC1) unless it is specifically inhibited to hold the adder stable through a second time state.

Besides the enables for the data inputs to the flipflops, there is also a signal from memory that is applied to the direct set or clear inputs of all these flipflops. This is the signal MCF AD FM+ SET, which is generated for a memory operand read that is made to fast memory (limited to the first reference in a double operand fetch). Since the clock is stopped during a synchronous memory reference, this signal directly sets the AD FM+ flipflops at the upper right in drawing ADFP and clears all other AD control flipflops. Then at the first clock after the memory subroutine, the operand is loaded into AR via the adder.

AD can receive either AR, or its complement, or both at once for zero. The enabling conditions for AD AR+ are on drawing ADAP. All but two of the AR+ nets are for a full word. The nets at C2 and D6 respectively are for the right and left adder halves separately, but the conditions in these nets are identical except for half word transfers. The conditions that enable the complement of AR into the adder are at the top of drawing ADAM. As shown in the lower right of the same drawing, MB is also available to the adder but only for double precision operations and certain operations in the instruction cycle. For the former the entire word is gated in; at IT0 the right half is always gated in for the effective address calculation, but the left half (possible containing restore bits for the flags in a JRST) is gated in only if there is no indexing (otherwise the restore bits are taken from the index register).

The enabling of either the register or its complement also occurs in the case of FM and BR. The complement of FM is always a full word (top, print ADFM), and as shown in drawing ADFP, the AD FM+ flipflops handle a full word except in a half-word transfer to AC when the other half of the destination is to be saved. The arrangement for BR or its complement is similar as shown in print ADBR. The AD BR– flipflops are always enabled together, but there are separate enabling gates for the left and right triplets of AD BR+ flipflops. Close inspection shows that the inputs to these gates are identical except for the second set from the bottom, which is again a half-word transfer where half of the destination is to be saved.

The remaining register transfers into the adder are from the IO bus (top, ADCR) and from BR shifted left (top, ADC1) which is used only in multiplication. Just below AD BR+2 are the flipflops that gate a mask into the mixer; the enables for AD MSK GEN are produced by the nets at the left of the flipflops and by the net that generates AD MSK EN B at the lower right corner. The signals that these flipflops gate into the mixer are the outputs of the mask generator at the top of drawing MAMS. This generator decodes the section of a byte pointer that specifies the size *S* of the byte to place a 1 in the (*S*+1)st bit from the right to generate a mask for taking the byte out of a word or inserting a byte into a word. Hence among the conditions that set AD MSK GEN only BYTE FIRST PART and BYTE PTR ~INC at B3 in drawing ADC1 actually generate a mask; in all other cases the enable to the mixer is for supplying a zero word.

The rest of the logic on drawing ADC1 is for the flipflops that enable the add input to the adder. Note that the two flipflops that actually enable the left and right adder halves are controlled by the AD clock like the rest of the AD logic; but AD ADD, whose function is limited to extending the clock period and driving the light on the indicator panel, is controlled directly by the main clock. Thus when the AD clock is inhibited to hold the adder state, AD ADD itself clears so as not to delay the next clock – the extended clock period is necessary only for setting up the adder and is not needed for an already stable adder.

At the far right in drawing ADCR is the AD EQV flipflop which is always enabled whenever AD ADD is enabled, but is also enabled alone when the adder is used to generate an equivalence function. The remaining logic in drawing ADCR is AD CRY 36 for producing a carry into bit 35 of the adder, and AD+1 LT and AD–1 LT for controlling the connection between the adder right and left halves. Note that AD+1 LT is set when an index register is added in the effective address calculation; this is done to nullify the 1s produced by the adder input mixer that is not enabled, so that the left half of the index register can be loaded using the add function.

The last set of adder flipflops enable the magic numbers into mixer B. The flipflops and their enabling nets are in the lower half of print ADFM, and the nets that generate the magic numbers are in the lower half of drawing MAMS. The flipflop MAGIC #+1 supplies a 1 to MAGIC #35 in an LUUO (to change the address from 40 to 41) and in integer division. The remaining magic nets on MAMS are for generating the page fail word and supplying a 1 to AD 08 at DNT1. The many other situations that set the AD MAGIC flipflops do so to gate into mixer B some quantity produced directly by nonmagic inputs to the individual mixer bits or more frequently to supply a zero word through mixer B.

7.1.3 Adder Zero Logic

The nets at the top and left of drawing ADZ decode the whole adder and various sections of it for zero or nonzero output. The rest of the logic supplies the conditions used for the arithmetic test instructions. In a comparison of one number against zero, as in the skips and jumps, only the adder output need be considered; but a comparison of one number against another, as in CAM and CAI, involves a subtraction, so the determination of the relative values must in some cases take into account the signs of the operands as well. The two types are the same in terms of testing for the equal or not-equal condition, for with two operands all that matters is whether or not they differ and not how they differ. Hence condition P, which indicates an absolute skip or a skip on an equal or not-equal condition, is used in both types of comparison. For the remaining situations, condition R tests only the adder outputs for the comparison of one word with zero, whereas condition Q takes the signs into account for the same tests in the comparison of one word against another.

7.2 ARITHMETIC REGISTER AR

AR always receives the memory operand and it is used in all arithmetic operations. In double precision floating point it holds the low order word of an operand with the high order part in the AR extension. In some cases, which operand AR holds depends upon their relative values (in floating point addition and subtraction it holds the operand that must be shifted to the right), and it also depends upon the mode, *ie* whether the non-AC operand is the contents of location E or is E itself. In multiplication there is no operand in AR at the beginning as the partial products are added up in AR. In division AR initially holds the dividend, which is diminished and shifted left while the quotient is being built up in MQ (although fixed point division is single precision, it uses a double length dividend wherein AR holds the high order half).

The arithmetic register and its input mixer are shown in drawings AR1–3. The mixer receives high inputs to generate high outputs for 1s, so the individual bit inputs matched with the enabling levels represent 1s for direct input. Most of the enables are applied to the entire mixer, but in some cases the gates are divided into groups; *eg* bits 1–8 can be handled separately from the rest of the register for a floating point exponent. The input bits for a particular enable are generally consecutive from one end to the other, but there are variations at the extremities because of shift requirements. The mixer stages all look alike for bits 3–35 but those for the first three bits on the left appear quite different because there are so many extra conditions for special circumstances; *eg* bit 0 is included in a logical shift but not in an arithmetic one, and the data places in a bit in a particular type of shift often depends on the operation in which the shift occurs.

The enabling levels allow an interchange of the left and right halves of AR and transfers from AD and MQ into AR, but of these only the AD transfer is seen consistently throughout. All shifting is done through the adder so that an arithmetic function and a shift can be combined in a single event; in other words the contents of AR can be combined arithmetically with a word from another register through the adder, and the result shifted left or right for the next step in an arithmetic procedure, all in a single time state. Shifts are one place left, one place right, and two places right.

The top two mixer enabling lines differ from one part of the register to another. In the right half these two lines are used for loading auxiliary inputs and loading PC for a PC word. The individual auxiliary inputs come from the small mixer in drawing ARMD; this mixer provides for reloading the right half of AR into itself when only the left half changes and for loading the mapping data in the MAP instruction. (Note that in the lower right of this drawing are duplicates of some of the AR bits for extra drive capability.) The second line in the left third of the mixer (bits 0–11) carries the signal AR EXP SCAD EN, which enables the loading of an exponent into AR 01–08 from the SC adder and simultaneously reloads bits 0 and 9–11 into themselves. Whenever an exponent is loaded into AR the rest of AR must be saved; this is done by means of the auxiliary inputs in the right half of the register, and the enabling level AR 12–17 SELF B for the bits at the top of AR2.

The top enabling line for the mixer left half is always held high so that the gates to which it is connected can be enabled by individual signals generated in any special configuration that is needed. Associated with this high line at bits 0–12 are a set of ARI data inputs, one for each bit (at the first gate in each mixer stage except at the fourth gate for bit 2). At bits 1 and 2 the high line also enables the second gate for an ARI auxiliary data input. At bits 13–17 the high line is connected to gates that are permanently disabled, so that whenever the register is clocked with no enable generated for this section, bits 13–17 are cleared. This is done to supply 0s in these bits in the assembly of a UUO or PC word.

The inputs associated with the permanent high line for bits 0–12 as well as other special inputs for AR 00 and AR 01 are produced by the logic shown on drawing ARI. In the upper part of this drawing is a secondary input mixer that supplies the ARI data inputs for bits 1–12 (note that mixer stage 12 is in the position where one would ordinarily expect stage 0). This secondary mixer supplies IR to AR left for a UUO, supplies the flags for a PC word, handles the smearing of the sign through the exponent part of AR (while the rest of the register is reloaded into itself), and handles the transfer of position information from the SC adder in byte instructions. Inputs for AR 00 equivalent to those of the secondary mixer are supplied by parts of the two mixer nets in the lower left, ARI AR 00 DATA and ARI AR 00 SELF. The former acts through the same mechanism as the secondary mixer, *ie* it goes to an

AR 00 mixer gate whose other input is held high (note that the top gate in the data net supplies the flag, which is Overflow in user mode, but Public in executive mode). The self net acts as an enabling level whose data is supplied by the ARI AR 00 SELF DATA net at B3; as the name implies, the data is AR 00 itself except in a DFN where the bit is complemented. AR 00 receives itself in a sign smear, the loading of an exponent, and in any arithmetic shift, left or right, one place or two.

At the lower right in print ARI are the auxiliary data nets for bits 1 and 2 of the AR mixer (these supply the missing inputs from MQ and AR right) and a second AR 01 auxiliary data net that displaces the IR input to bit 1 of the ARI data mixer in order to supply both the missing IR bit and additional information for a rotate or arithmetic shift. The remaining ARI logic supplies the rest of the special inputs to bits 0 and 1 of the ARI mixer and various inputs for the ARI mixer nets. Although the way in which a particular function is effected at the left end of the register is sometimes circuitous, the signals that identify the particular logical path used are listed at the appropriate places in the flow charts. As an example consider a two-place right shift. At AR 02 the enable is connected to mixer gates 5 and 6; at each of these gates an extra condition provides for input from ADX 35 in double precision operations, otherwise from AD 00. At AR 01 the enable is also connected to two gates, 4 and 6, and consistent with bit 2, the input is ADX 34 for double precision. But here input is from AD -1 only when ARI SR2 COND is true, and investigation of the net at ARI B5 informs us that this particular data connection is made for a two-place right shift only in fixed point multiplication and single precision floating point. The remaining needed inputs are supplied through the ARI data net from the second auxiliary data net at ARI B1. This latter net supplies AR 00 in any arithmetic shift and in a rotation supplies data from the gate at A6, which as expected shows that input is from the opposite end of the rotating word, being AR 35 in a single rotation, MQ 35 in a combined rotation. We have already mentioned that AR 00 receives itself in any arithmetic shift, and the enable is again connected to two gates, 4 and 8. The input is AD -2 when ARI SR2 COND is true; otherwise input is supplied by ARI SR2 DATA, which as seen at ARI B7, is equivalent to the rotate data gate with the substitution of bit 34 of AR or MQ. A similar analysis would clarify the situation for any other AR operation. The only variation at the right end of the register is the left shift input to AR 35, which is supplied by the auxiliary mixer net at the lower right in drawing ARMC.

AR Gating. The generation of the enables for the AR mixer is shown in three drawings. At the right in ARMC are the AR clock and the nets that generate the clear and double right shift enables. At the left are the enables for single right shifting, transfers from SCAD to AR (for exponents and otherwise) and for smearing the sign. The gate at top center generates a self-transfer for AR 12–35 on a sign smear or a transfer from SCAD.

Drawing ARMA is devoted exclusively to the enabling of AD into AR, in some cases combined with enabling AR to BR. The net at the lower right enables the right half of the mixer independently, and the nets in D3–5 provide a separate enable for the left half. As usual, these are mostly for half word transfers. Print ARMB shows the enables for left shifting, loading MQ, PC or the flags into AR, and moving one half of the register into the other.

7.2.1 AR Flags

From left to right at the top of drawing ARF are the two carry flags, Floating Overflow, Floating Underflow, and No Divide. Below are Overflow and a nonprogram flag that determines the applicability of certain floating point tests to underflow. AR FXU HOLD is cleared at the beginning of the fetch cycle, and so long as it remains clear certain of the conditions that set AR FOV also set AR FXU; should the hold flag be set however, those same conditions indicate only high overflow.

At some point in every possible condition that can set any AR flag, the term PI CYC(0) appears; hence the flags are never affected by an interrupt instruction. As is indicated in the reference manual, the conditions for some flags are subsets of the conditions for others. AR FOV is set by any condition that sets AR FXU and by any floating point condition that sets AR DCK; similarly AR OV is set by all overflow conditions. The carry flags reflect simply the carries out of AD bits 0 and 1 but only at the ARF strobe, which is limited to the basic fixed point addition and subtraction operations (C4).

The enabling net for each flag has a gate for loading the flag from a PC word via AR, and another gate for holding the flag set unless it is to be cleared or the gate is simply disabled to allow the load gate to function. For the four flags that can be tested by a JFCL, the clear signals are supplied by the gates at the lower right, each of which reflects the individual JFCL clear and the general load (C6) for a JRSTF or MUUO. For the other two flags the hold gate is disabled only by the load level.

7.3 ADDER AND AR EXTENSIONS

AD and AR each have a 28-bit left extension, including bits 0 and 9–35, for handling the high order parts of double precision floating point numbers. No exponent part is needed as the exponents are handled entirely by SC and FE.

Drawings ADX1 and 2 show the adder extension ADX. The least significant end is at the lower left in ADX2, where an extra stage ADX 01 duplicates bit 1 of the main adder to provide the proper carry into ADX 35. This arrangement bypasses AD 00 so as not to include the sign bit of the low order part in arithmetic operations. The module carry in is identical to that into the leftmost AD module. One data input is the duplicate A mixer bit for AD 01 and the other is a corresponding B mixer bit whose generation is shown at the top of drawing ADX2. The adder extension operates in exactly the same way as the main adder with the carry functions simply extending from right to left to produce a double length adder. This larger network requires only slightly more time than the main adder, the ratio of the time states for setting up the double as opposed to the single adder being only 191 *vs* 170 ns.

Of the two sets of inputs for the adder, one is just the contents of ARX. The other input comes from the mixer in drawings AXB1 and 2. This mixer operates in exactly the same way as the AD mixer and its enabling levels are supplied by control flipflops. Inputs are the complement of BR, FM or its complement, and twice the contents of FM. In print AXB2 is an extra adder bit, ADX -1 IN B, which in conjunction with ARX 00 (through the signal ADX -1 IN A at ADX1 D5) supplies the inputs to ADX bits -1 and -2 for right shifting. The only ADX carry output is from bit 0; but like the AD carries, the signal used in the logic is not the direct adder output. The logic uses ADX CRY 0, which as shown at ADX2 A4, is the true carry except when it is forced by a double floating divide time state for the initial subtraction in the divide sequence.

The enables for the mixer and the add and equivalence inputs for the adder are in drawing ADXC. ADX EQV is always set with ADX ADD, but is sometimes used alone. The enables are generated only for full words, but note that the nets at the upper left generate signals that enable both FM+ and FM-. This combination produces a mixer output that is all 0s, so that the adder output is derived solely from ARX. State changes in the ADX control logic are timed by the AD clock, because ADX is used only in conjunction with AD.

The ARX register and its input mixer are in drawings ARX1 and 2. Except at bit 35, all of the bit inputs to the mixer are from ADX for loading, left shifting, and right shifting one place or two. Note that for the single right shift, ARX 00 receives ADX -1 except in DFDV, where it is always cleared. The ARX 35 mixer has the same inputs as the other ARX bits for right shifting or a transfer from AD, but in a left shift the input comes from AD 01 except in DFDV, where it is the output of ADX bit 36 (ADX 01).

The ARX clock and the enabling levels for the ARX input mixer are in drawing ARXC. At the upper right are nets that decode ARX for zero or nonzero contents and test the equivalence of ARX 00 with ARX 09 and ARX 10 for use in normalization.

7.4 BUFFER REGISTER BR

BR is used primarily as a temporary holding register and also to supply one of the operands to AD in various arithmetic operations. It holds the multiplicand while the product is being constructed in AR and holds the divisor for subtraction out of the dividend in AR. In all double precision operations it holds the low half of an operand for AD while the high half is supplied to ADX from FM (the other operand being supplied to ADX and AD by ARX and AR).

BR receives full words from ARX or AR. For the former, BR 01–08 (the exponent part) all receive the ARX sign, and the BR sign can be smeared through these bits. When BR is loaded from AR, BR 00 receives AR 00 except at DNT2, when it receives AR 01.

Drawing BRC shows the BR clock, a duplicate flipflop for BR 00, and the enables for the two BR transfers. The sign smear is accomplished by setting the flipflop in the upper right, which in turn sets or clears the appropriate BR flipflops depending upon the state of BR 00.

7.5 MULTIPLIER-QUOTIENT REGISTER MQ

This register holds the multiplier, supplying each bit from MQ 35 as the product is being shifted in at the left from AR. In double precision multiplication it first supplies the low part of the multiplier while the high part is saved in MB; at the completion of the first part of the multiplication sequence, the lowest order part of the product then in MQ is thrown away, and MQ receives the high multiplier for the second part of the sequence. In division the quotient is built up one bit at a time from the right end of MQ (in fixed point division MQ supplies the low part of the dividend to AR as the quotient is being shifted in). Following the first part of a double precision division sequence, the high quotient is moved from MQ to MB and the second part of the sequence builds up the low quotient in MQ.

The MQ register and its input mixer are in drawings MQ 1–3. For most bits the input mixer has four gates for a transfer from AD and MQ inputs for a left shift and a shift right one place or two. There are however special considerations at both ends of the register for shifting and at bits 7, 8 and 9 for single precision floating point operations (to bypass the exponent part in the low order half of a software double precision floating point number). Note that the individual bit inputs for MQ 31 and MQ 34 on a left shift are either the bits that supply the shift information or the function MQ GETS 22. This signal loads 22 octal into MQ through the use of the left shift enable when MQ is clear.

The nets in drawing MQZ decode MQ for zero or nonzero contents. Most of this decoding is done from the outputs of the MQ mixer (the inputs to the MQ flipflops), so that at the same time MQ is loaded, the flipflops at the upper right are adjusted for the the state MQ receives. Note that the signals MQ 34 IN and MQ 35 IN on drawing MQZ are of the opposite polarity of the equivalently-named signals produced by the MQ input mixer; furthermore these signals are not logically equivalent, but include only the conditions needed for the zero detection logic.

Drawing MQC shows the MQ clock, the enables for the MQ mixer, and the special functions that control the shift actions at the register extremities and at bits 7–9.

7.6 SHIFT-COUNT LOGIC

This logic is based on a 9-bit shift counter SC and an associated adder SCAD. SC is used for controlling all shift operations, including the shift procedures in arithmetic and byte instructions; it is also used as an arithmetic register with SCAD for all computations on exponents and the computation of byte position. In floating point instructions the exponent computation is done first, and the result is saved in a holding register FE while SC is being used for the shift procedure.

Print SCFE shows two 9-bit registers SC and FE (in the lower right of SCDR are duplicates of the first three SC bits for extra drive capability). SC can receive the appropriate bits of E directly from AR in a shift-rotate instruction, but all other data is routed through the SC adder, which also supplies the results of arithmetic operations. For use as a counter to control shifting, the SC outputs are applied to the +1 gate at the upper right in drawing SCSR, and the gate outputs are applied to the SC input mixer. Note that SC receives not only the +1 outputs for the corresponding bits for counting, but also the +1 outputs displaced one bit to the right for a right shift that divides the count in half. This works because it is done only when SC contains a negative odd number and thus halves the next greater even number, *ie* the number one less in magnitude. The nets at the lower right in print SCSR decode SC for all 1s, to indicate that the next clock, which will complete the count, must also return to a time state in the sequence from which the shift-count was called. Drawing SCC shows the SC and FE clocks, the enabling nets for the SC input mixer, and other decoding of the SC contents.

At the top of drawing SCAD are the three M142 modules of the SC adder, which functions in exactly the same way as the main adder AD. It has enabling inputs for add and equivalence, +1 inputs at the LSB for incrementing and negating, and inputs to the rest of its stages from the pair of mixers below. Note that at the left end is SCAD - 1, which receives the same mixer inputs as bit 0, but due to the configuration of the unused bit between it and bit 0, its carry in is the carry out of the sign; this extra bit is used in testing for overflow. The upper mixer supplies input from FE, SC or its complement, and the complement of the size part of a byte instruction from AR. This last is for subtracting from the position supplied from AR through the lower mixer. In floating point operations the exponent is supplied to SCAD from AR through the lower mixer either for actual computations or simply for transfer to SC. Note that the exponent gates provide the complement of the exponent MSB (AR 01) to bits 0 and 1 of the adder. This simple inversion converts the exponent from excess-128 code to ordinary twos complement representation, thus greatly simplifying exponent computations. The conversion back to the standard form takes place when the result from SCAD is inserted back into the floating point number in AR (refer to bit 1 of the secondary AR input mixer on drawing ARI).

The final set of gates in the lower mixer is entirely for supplying fixed constants to SCAD from a separate data register that is loaded at the same time it is enabled into the mixer (failing to load it of course supplies zero through the lower mixer, so that addition produces a simple transfer of the information supplied by the upper mixer). These constants include the shift counts for the various multiplication and division operations, and also various constants used in calculations to determine initial byte position for a new word, to eliminate redundant movement in a shift-rotate, and so forth. The data register is at the top of print SCDR, and below it are the gates for producing the appropriate input configurations for the constants indicated by the enabling levels at the left. Two of these are a direct IR decoder output and a time state, but the rest are composite signals generated by the nets in the lower part of drawing SCA2. The rest of the three SCA drawings show the flipflops that provide the enabling levels for the adder and its mixers.

7.6.1 Shift-Count Subroutine

The shift-count subroutine has two time states defined by the SCT flipflops in print SCSR. (The SRT1 flipflop at the left defines a time state in shift-rotate instructions and has nothing to do with the subroutine, although it does serve as the entry to that subroutine.) The table in flow chart SCON lists the special connections that are made at the extremities of ARX, AR and MQ to produce the types of shift movement required in the various instructions. Below the table is a flow chart of the basic shift-count subroutine. The subroutine actually has far more events than are shown here, but the ones not shown are all for particular instruction situations – indeed almost all of them are the special events required for exiting when the subroutine is finished (in each case the complete details of the subroutine for a given instruction are shown in the instruction flow chart).

Although there are two shift-count time states, the actual shift-count loop uses only SCT2; in any given subroutine SCT1 occurs at most once if at all. The reason for this is that right shifting, which is used in all multiplication and division, is done two places at a time. Thus the entry to the subroutine is through SCT1 only for a right shift that is either odd or of an unknown number of places, while entry is directly to SCT2 for left shifting or for right shifting that is known to require an even number of shifts. Hence SCT1 shifts right one if SC 8 is 1 (indicating an odd number) and also shifts SC right to divide in half the even number of next lesser magnitude. SCT2 on the other hand shifts left one or right two and adds one to SC. Until the count reaches - 1, each time state enables AR+ into AD as AR shifting is done through the adder (remember that MQ shifting is done through its own input gating and ARX is always enabled into ADX). Although AR+ is enabled into AD during the final subroutine time state, that state does not generate the enable for AD AR+, as the final clock, although it shifts, must set up other conditions for the return to the calling sequence rather than for further shifting.

Any shift of whatever type always involves AR so AR shifting is always enabled at every time state in the subroutine. Note however that shifting is often enabled for the other registers even when they are not used, as it takes less logic to let them shift irrelevantly than it does to cut out shifting unnecessarily. For left shifting, AR and MQ are always enabled but ARX is enabled only during double floating divide. For the single right shift at SCT1, all three registers are enabled all the time with the single exception that MQ shifting is inhibited during a load byte to prevent messing up the mask. Two-place right shifting is the same except that ARX is enabled only in double precision operations.

The condition for setting SCT2 and thus continuing the shift count requires not only that SC not have reached −1, but also that the function SCT PROCEED be true. Although this function looks somewhat complicated at first glance, it actually allows early termination of a count only for a priority interrupt during the first part of a double floating division. In other words the count proceeds automatically in any other instruction, and even in a DFDV the setting of PI RDY SYNC has no effect after FLAG 2 has been cleared at the completion of the calculation of the high order part of the quotient.

CHAPTER 8
INSTRUCTION FLOW

The hardware described in the preceding three chapters implements the processor execution of the basic instructions, and this chapter is devoted to a discussion of the sequences of events that make up these instructions. (The IO instructions and other special IO sequences are treated in Chapter 9 with the discussion of the IO logic.)

To begin an instruction, the processor performs the instruction cycle in response to the synchronization of a pair of circumstances that are shown in the upper part of flow chart IC. These circumstances are that the previous instruction is done and the instruction about to be performed is ready in MB. For any instruction to be ready for execution, it must first be retrieved from memory and placed in MB; hence before discussing the execution of instructions, let us first consider the various ways in which instructions are fetched.

8.1 INSTRUCTION FETCHING

The upper half of the first memory control flow chart MC1 shows the numerous entries into the memory subroutine, including those for fetching an instruction. Of all the entries shown, only some of those in the right half of the upper group are solely for fetching or storing operands; the lower group is exclusively for fetching instructions, and many of the entries in the upper group are operand entries that may also trigger instruction fetches.

To start normal program operation of the processor, the operator must trigger the initial instruction fetch from the console by using start, continue, or read in (shown in the key function flow charts KF1 and KF2). These functions simulate instruction done by means of a key done condition, and the instruction fetch is triggered by the KT2 entry at D4 in MC1. This entry is to MC INST FETCH EN, which is the primary instruction fetch signal because it directly sets both MC INST FETCH START and MC ASYNC START, and thus calls the subroutine immediately. Once the instruction has been brought out to MB, the return from the memory subroutine generates instruction ready, and the processor responds by beginning the instruction cycle. (Note that continue may not fetch an instruction. If the processor has been stopped within the memory subroutine, pressing the continue key simply restarts the subroutine.)

Once the processor is running and is performing instructions in the normal manner, there are three ways in which new instructions are fetched. When the processor reaches a point in an instruction where it knows that no further events in this instruction can interfere with fetching the next one, then it simply generates MC INST FETCH EN; the entries for this are at C3 and D3 in MC1. A similar way is when the processor knows that all will be well for instruction fetching at the next time state and therefore sets MC INST FETCH NEXT. The entries for this are in the center section of the lower row in MC1, and it can easily be seen that the 1 state of this flipflop causes the next time state to generate MC INST FETCH EN. Generally speaking there are two circumstances in which this type of entry is used: where the next time state may be repeated later in the same instruction, or where it is not unique, *ie* the next state may be any one of several. An example of the latter case is shown at the upper right in fetch cycle flow chart F. Often where there is no operand fetch, it is known that the instruction fetch can be triggered at the next state; but which state that is depends both on whether there was indexing in the final instruction cycle, as well as what execute time states are used by the current instruction. In other words the next state may be F CYC ACT EN, but it is also possible for this time state to coincide with F CYC START, in which case the next time state may be ET0, ET1 or ET2. Now it is obvious that the logic could be implemented for generating MC INST FETCH EN directly in the appropriate time state, but it is also obvious that this would require a great deal more hardware than a simple flipflop.

Both of the above methods trigger the next instruction fetch at a known point in the current instruction. The third method is the automatic instruction fetch, wherein the processor makes two memory subroutine calls simultaneously, and the second is to follow automatically as each step of the first is completed through the subroutine pipeline. This type of access is used at an operand call (fetch or store) when it is known that nothing further in the present instruction can interfere with the fetch of the next. To do this the processor makes the operand call, which directly triggers the subroutine, and also sets MC INST FETCH START; the entries for this are at the left in MC1. (Several of these entries are seen by the flow lines to be paired with operand entries, but the FT5 and ET0 entries also have corresponding operand entries located in D2.) The actual triggering of an instruction fetch requires that both MC INST FETCH START and MC ASYNC START be set. In an automatic fetch only the first of these is set from the instruction sequence. Then when memory control completes the first stage of the synchronous subroutine, it produces the recycle sequence at the far left in MC2; this sequence in turn sets MC ASYNC START and automatically triggers another memory subroutine through the recycle entry at MC1 B4. (Note that the automatic procedure is also used for the second fetch with a double operand, where MCS FCE2 START causes the recycle.)

The direct initiation of an instruction fetch by generation of MC INST FETCH EN at a specific point in the flow for an instruction is shown both by the event and by an arrow indicating the beginning of the instruction fetch flow. The other two types are indicated simply by listing MC INST FETCH NEXT EN or MC INST FETCH START EN, and the reader must realize that such an event sets up the circumstances for the subsequent initiation of the next instruction fetch, either at the next time state or automatically following the operand page check.

The three instruction fetch methods just discussed are all applicable to the normal progression of the processor from one instruction to the next in the program. For each, the true presence of the next instruction in MB produces instruction ready, and the same instruction sequence that produces the instruction fetch also eventually generates instruction done for the synchronization to trigger the next instruction cycle. This cycle decodes the instruction code and performs the first step of an effective address calculation. If the address is indirect, then the instruction cycle itself fetches an address word by calling another memory subroutine as a pseudo instruction fetch, initiated by means of the IT1 entry at MC1 D1. For synchronizing additional instruction cycles in the calculation, the instruction done condition simply remains in effect until the entire calculation is complete, and memory control announces the availability of each address word by generating instruction ready. Note that there is one case in which both instruction ready and instruction done are simulated in the middle of an instruction with no return from memory at all; this is following the first part of a two-part byte instruction where the processor responds with an instruction cycle to begin the effective address calculation on the byte pointer.

So far we have been discussing the normal sequence where each instruction calls the next to continue the program. There are however four circumstances in which this normal flow is interrupted and special conditions are used to restart it. The simplest of these is an overflow trap, where the instruction cycle throws away the instruction just loaded into IR and diverts to IT2 for a pseudo instruction fetch (MC1 D1) to get the special trap instruction out of sequence. The trap instruction then returns the processor to the normal instruction fetching procedure.

Somewhat more complex are diversions to a PI cycle or a key cycle. If either of these diverting circumstances is known far enough ahead, it actually prevents the current instruction from fetching the next one. Otherwise it may interrupt following any instruction cycle in an address calculation and may even interrupt in the middle of an execute cycle if an IOT is waiting for the IO bus; in either of these cases the instruction being performed is thrown away. For a priority interrupt the PI cycle may do a pseudo instruction fetch (PIT2 at MC1 D1) or may simply commandeer part of the main sequence to execute an interrupt function. In the first case there may be a second PI cycle with a second pseudo instruction fetch (the cycle flow is shown in chart PI and its details are discussed in section 9.2). In any event the final PI cycle relies on the instruction or function executed to return to normal instruction fetching.

If the operator interrupts program flow with start, continue, or read in, then the processor simply starts over as already described. In an examine or deposit function, the same condition that calls the memory subroutine to examine or deposit a word (upper left corner of MC1) also sets up memory control for an automatic instruction fetch to follow the console memory access; synchronization for the instruction cycle is on instruction done simulated by key done. Execute uses the standard instruction procedure for executing an instruction supplied from the console. To trigger the instruction cycle, key done produces instruction done, but the execute function moves the instruction word from the data switches to MB and then simulates instruction ready. The function relies on the instruction executed for the next instruction fetch.

Finally normal flow may be interrupted in a memory subroutine because of a page failure. If the failure is in an ordinary program instruction, the special page fail cycle (chart PF) performs the necessary actions, using part of the BLT sequence if the failure is in a block transfer, then goes to the store cycle to write the page fail word, and finally utilizes the trap cycle to fetch the page fail instruction. Other types of page failures are not regarded as normal and the processor does not respond with a page fail trap. In an MAP the processor simply returns to the instruction. A failure in an interrupt is regarded as catastrophic – the page fail cycle sets an appropriate flag and triggers the fetch of the next instruction (which would ordinarily be aborted by a processor interrupt). Although console functions can be executed while the processor is running, no page failures produced by them are allowed to interfere with the program. Thus in a key function, the page fail cycle directly triggers the next instruction fetch by entering the recycle sequence at MC2 C8.

8.2 DATA TRANSMISSION

This section treats all of the data transfer instructions except PUSH and POP, which are included in the push-pop group in the next section. The data transmission instructions that make use of neither an extended execution sequence nor the shift count subroutine are shown together in flow chart FHWT. These include the half word transfers, the move instructions (which are referred to in the logic as full word transfers), and EXCH, whose flow line at the right is self-explanatory.

The full and half word transfers have common modes and therefore have many common switches generated by IR HWTFWT. Although there are some move signals decoded from IR, the full word group as a whole is represented by IR FWT. With the decoding at the upper left is a table that shows the composition of the word moved and its destination for all sixteen instructions. At the end of the fetch cycle the operand is in AR except for memory mode, which uses ET0 to move it there. Regardless of mode, for an MOVN or an MOVM the negative is formed in AD during the ET2 time state. Then a swap is made if the instruction is an MOVS, whereas if the negative is required, it is loaded from AD into AR and the flags are set up. Of course the negative is always loaded in an MOVN, but the loading takes place in an MOVM only if the operand was originally negative, *ie* if taking the magnitude requires something different than the word already is.

The half word transfers are represented by IR HWT and a number of other decoding levels that represent transfer type, mode, and action taken on the other half. The table at the lower left lists the composition of the resulting word and its destination for all 64 instructions. The most important thing to remember in following the flow is that if there is to be an action taken on the other half of the destination, then data need be fetched only from the source (which supplies the half word to be moved). But if the other half of the destination is to be saved, then both source and destination must be fetched and the result produced by combining half words from both. As with the full word transfers, the word supplying the data to be moved is in AR at the end of the fetch cycle except for memory mode, which uses ET0. For constructing the result, the time state preceding ET2 performs two operations: if there is no action on the other half, then the destination half word to be saved is set up in the correct position in AD; and if the half word being moved is not being swapped, then it is also placed in the correct position in AD. At ET2 the result is constructed by means of at most two half word operations. The half being moved is loaded into AR either directly from AD or, for swapping, from the other side of AR. If the specified action requires that the other half of the word be zero, then the other operation is inhibited so that the AR clock simply clears the other half or AR. But for no action or an action that requires all 1s, the other half is loaded from AD, which contains either the saved half of the destination or all 1s (the latter because no AD input mixer is enabled).

Block Transfer. The logic for the special BLT execution sequence is shown in drawing DMBL and chart BLT shows the flow. The first action of the instruction is to compute the negative word count from the initial and final destination addresses. A carry out of AD right at ET0 indicates that the negative count comes out positive (the final address is less than the initial one), and in this case ET1 loads AR left with −1 instead of the computation result. The rest of the events in the first column save the current program address in MQ, move a number one less than the initial destination address into PC, and construct a word made up of the negative count in the left half and the initial source address in the right. Then in the iterative sequence that performs the transfer (the second column), the word count and source address are incremented together in AD with the address on the right for supplying it to AB to fetch each word; and the destination address is counted in PC, from which it is also supplied to AB for storing each word. Following the BLT sequence on each iteration, the processor uses the store cycle to store the word, update the various parameters, and supply the next source address to AB.

At each BLTT1 the test is made to determine whether to repeat the standard sequence; the processor processes the next word only if there is neither overflow nor an interrupt. If either of these conditions occurs the processor leaves the standard iterative sequence and goes into the flow shown at the right. An overflow of course means that the transfer is finished, and the terminating sequence allows trap satisfaction (which has been held off till now), restores PC, starts the next instruction fetch, and returns to the instruction cycle. If the transfer is not finished but an interrupt has been synchronized, the sequence beginning at BLTT7 handles premature termination, which requires all the normal terminating actions but also saves the next source and destination addresses in AC so that the processor can resume the transfer after the interrupt.

For a page failure in a BLT, control goes directly from one of the memory subroutines to the page fail cycle. But instead of executing as it does for other instructions, the page fail cycle transfers control back to the BLT sequence at the upper right, so that events necessary for premature termination can be done at the same time as the construction and storage of the page fail word. PFF1 being set puts the fail word into AD via the magic numbers at BLTT7, produces the MA special levels to select the appropriate location in the user process table, and generates the store switch to call the memory subroutine from BLTT9 and the write restart from ST1. The return to the instruction cycle is then made from ST2 directly to IT2 for a page fault trap.

Byte Manipulation. Drawing BYTE shows the special logic for the byte instructions. Note in particular the two flags at the upper right, both of which are set at the end of the first part of any byte instruction that has two parts. BYF5 simply distinguishes between the first and second parts. BYF6 is the First Part Done flag, which stays on if the second part is interrupted to prevent a subsequent restart from incrementing the pointer a second time for the same byte.

The first and second parts of the byte instructions are in the left and right halves of chart BYTE. The first part has two entries both of which begin by retrieving the pointer. If the pointer is not to be incremented, the position is loaded into SC and a mask for manipulating the byte is generated and saved in MQ. The mask generator in the upper half of drawing MAMS produces a word with a single 1 such that the number of 0s at the right of it is equal to the size of the byte. Adding this word to -1 (*ie* with the other AD input mixer not enabled) produces a mask with exactly S 1s at the right. If the pointer is to be incremented, the first part subtracts the size from the position to produce a new position and increments the address. Besides generating the mask, ET0 subtracts again, but this time the result is $P - S$ only if there is enough room for another byte in the word. If the previous subtraction produced a positive result, then the incremented address replaces the previous address and 36 is substituted for the position in the subtraction so that the new position points to the first byte in the next word.

ET2 inserts the (perhaps new) position into the pointer, and if the instruction has two parts, it also moves the pointer to MB and sets INST RDY so the instruction cycle will think that the pointer has just come from memory. If the pointer has not been incremented, there is immediate synchronization for entry into the instruction cycle (with BYF5 and BYF6 both set) to begin the effective address calculation indicated by the pointer. An incrementing instruction on the other hand uses the store cycle to put the new pointer back in memory. It then synchronizes to the instruction cycle with the byte flags set if there is a second part, but for an IBP there is instead the standard enabling of MB into IR for the next instruction (note that for an IBP, INST RDY is set by the automatic fetch of the next instruction rather than by ET2).

In the instruction cycle the processor calculates the effective address for the byte, and BYF5(1) prevents MQ from being cleared so that the mask is not disturbed. The second part also has two flow paths beginning at fetch, but in this case the division is between depositing and loading the byte. Either path retrieves the byte word since it must either supply or receive a byte, but the deposit requires both fetch and store (a split cycle is used because of the time taken by the SC subroutine). Following the fetch *P* is negated through the SC adder to control the shift. *P* is actually the right six bits of SC (this being all that is taken from or inserted back into the pointer), so SC can be expected to be positive except where the first part does a byte increment that overflows and *S* is greater than 36. In this case negating *P* would produce a small positive number and hence an unacceptably long shift count, so on the condition SC 0(1) the left three bits of SC are set to guarantee its being negative (this is equivalent to subtracting 64 so the result is $-(100 - S)$). A carry out of bit 0 indicates the byte is at the right end of the word (the zero position), so no shifting is necessary and flow continues on in the execute cycle; otherwise the sequence goes to the shift-count to shift either the byte and mask left to the correct position for depositing or the byte to the right end for loading.

Following the shift in a deposit (or at ET0 if no shift is necessary), the byte part of the word that is to receive the byte is masked out by anding the word with the complement of the mask. (The write part of the split cycle is also started at this time.) Then ET1 masks out all of the word from AC except the byte by anding the AC word with the mask. Finally ET2 ors the byte into the vacated part of the memory word by loading both modified words into AR simultaneously.

For loading, the SC subroutine right-justifies the byte in the memory word, but does not move the mask as it has been generated with the 1s at the right. ET1 ands the mask with the word to get rid of everything but the byte, and finally the byte is loaded into AC.

8.3 CONTROL AND TEST INSTRUCTIONS

All of the jumps, skips and other control instructions are executed in the standard time state sequence except JFFO, which has one special execution state. MUUO uses only the standard main sequence but requires three of them and is effectively a triple instruction.

8.3.1 Jump Instructions

Flow charts JMP1 and 2 show XCT and all of the jump instructions except those that are part of the arithmetic test group. All are self-explanatory except JRST, JRA, JFFO and XCT.

JRST *(center column, JMP1).* JRST involves a number of special events as early as IT1. At the same time that the calculated effective address is loaded into AR right, whatever information is contained in the left half of the last index register or address word used is loaded into AR left in case it contains bits for restoring the flags. HALT sets KEY PROG STOP and clears KEY RUN so that the next time state will synchronize as though a key function were about to be performed. If the instruction is a PORTAL taken from a concealed area, Public is cleared (this is the only legitimate way to enter a concealed area). HALT and JRSTF are regarded as JRST instructions with flags; without flags, the next instruction fetch occurs at the beginning of the fetch cycle.

At ET2 a JEN dismisses the channel on which the current interrupt is being held. A JRSTF loads the flags from AR left, subject of course to the mode in which the instruction is given. A JEN or PORTAL is over at ET2, but for the instructions with flags it places E on the address bus and loads PC into AR. ST1 then triggers the next instruction fetch, which has been delayed until this time for either of the following reasons: if the flags are being restored, the new configuration may well affect the way the instruction is fetched; if the processor is halting, it is necessary to have the key function synchronization (which was not available earlier) set PI CYC RDY. A JRSTF is finished at this point and returns to the instruction cycle. However any JRST that calls for halting also goes back to the instruction cycle, but PI CYC RDY being set aborts the next instruction fetch (chart MC1) and causes instruction done to set up a special time state in which KEY IDLE is set and the jump address is loaded into PC from AB. KEY IDLE then defines a time state of its own in which the processor clock stops and cannot be started until some action is taken at the console.

JRA *(right column, JMP1).* This instruction makes use of an extended fetch sequence to fetch the contents of the location specified by AC left, which it is assumed contains the effective address of some prior JSA. Said address is moved from AC to AR and thence from left to right for transmission via AD to AB for an operand fetch. E is then brought back for the next instruction fetch.

JFFO *(second column, JMP2).* AC is available through AD for preliminary inspection at ET0. If AC is nonzero, FLAG 1 is set to indicate that we must look for a 1 and that the jump will take place; if AC left is zero, FLAG 2 is set to avoid the 18 unnecessary shifts. If we must look for a 1 but there are none in AC left, ET1 places an initial count of 18 in MQ and puts the nonzero half of AC in AR left. Meanwhile if we must jump, E is loaded into PC, and in any event PC is placed on the address bus for the next instruction fetch. At ET2 AC is moved to MQ for shifting and the initial count (if any) is moved to AR to be continued through AD.

If FLAG 1 was not set or if the leftmost bit of the word to be shifted is already 1 (as detected from AR before it is obliterated by loading the count), then we load the count of 0 or 18 into AC2 and the instruction is over. Otherwise we go to the repetition of the special JFFOT1 time state to shift left and count until it is seen that the next shift will move a 1 into MQ 00.

XCT *(right column, JMP2).* XCT immediately starts an instruction fetch from E but XCTF(1) prevents the completion of the page delay from incrementing PC. The special logic for this instruction is in drawing XCT. If the processor is in executive mode, ET2 sets up the special XCT paging flags according to bits 11 and 12 of the instruction word. Then the sequence returns to the instruction cycle to execute the instruction that was fetched out of sequence but without affecting PC.

At F CYC START of the executed instruction (final part in a byte), XCTP RD or XCTP WR being set causes the setting of XCTP ACTIVE to control the paging crossover that can be done by an executive XCT. In particular as seen from the circuits at the lower left, XCTP ACTIVE(1) generates the XCT protection bypass in concealed or kernel mode, and indicates that an AC reference is to the shadow area if user paging is in effect. Other logic of relevance to the execution of an instruction by an executive XCT is shown in drawings USER, PAG1 and PAG2, and is discussed in section 5.5.

8.3.2 Push-Pop Instructions

Chart PP shows the four push-pop instructions. Two of these are for data transmission and two are for control (jumping), but they align as push instructions and pop instructions in terms of structure. Both push instructions increment the pointer and add a word to the stack, but PUSH simply moves a word from E whereas PUSHJ constructs the data (a PC word) and jumps to E. Similarly both pop instructions decrement the pointer and get a word from the stack, but POP simply moves the word to E whereas POPJ uses the word to jump. PUSH and POP also have a combined IR decoding level for some minor common characteristics, but in each case the common characteristic is shared with one of the jump instructions as well.

PUSHJ begins by setting SAC BR FF so that the modified pointer will be put back in AC from BR. After the pointer is incremented in the fetch cycle, its right half is put in AB for a page check of the next location in the stack. In execute, pointer overflow sets Trap 2, the PC word is constructed in AR, the updated pointer is returned to AC, and E is moved via AB to PC and is made available from there with PC+1 inhibited for the next instruction fetch at ST1. Store writes the PC word in the stack. PUSH is essentially the same except that the fetch cycle reads an operand from E, the next instruction fetch does not depend on any events in the instruction and is therefore set up to follow automatically after the stack page check, there is no need to construct data or play with E, and store writes the operand in the stack.

One of the pop instructions is practically a subset of the other: almost every event in POPJ happens in POP, which is mostly just longer. POPJ uses an extended fetch to read a word from the stack according to the pointer, which it then decrements. The single execute time state checks overflow, puts the decremented pointer back in AC, and puts the right half of the stack word on the address bus for the next instruction fetch at ST1. POP begins by setting SAC BR FF and then duplicates the POPJ events, but extends the fetch cycle still further for a page check of E (which will receive the stack word) with the next instruction fetch following automatically. Execute actions are the same except that no instruction address need be supplied, and two time states are required instead of one because POP is handling two words: the decremented pointer which is written back in AC from BR, and the stack word which must be gotten out of the way and then brought back to AR for storing in E.

8.3.3 Logical and Arithmetic Testing

Flow chart TEST shows both of these 64-instruction groups. The arithmetic test instructions at the right are represented by IR 3XX, and there is further decoding for the eight primary instructions, each with eight modes. A single flow chart suffices, but at two of the time states the events are grouped for the three compare, skip and jump instruction types. The reference manual treats the logical test instructions as sixteen primary instructions each with four modes, but there is no comparable decoding in the hardware. Instead the entire group is treated simply as the test instructions under IR decoder output IR TEST, with additional decoding for various characteristics shown at the left in drawing MPYT. A table at the left in the flow chart lists the equations for this decoding and the meaning in terms of mask selected, modification, and mode.

Logical Test. There is considerable variation in the time state sequence for the test instructions. ET0 is called only if the mask must be swapped, which in terms of the instruction mnemonics is either swapped or left; ET1 occurs only if the masked bits are being changed to 1s or there may be a skip. Hence the sequence may include either, both or neither of these time states; *eg* a nonskip instruction that uses a direct mask and changes the bits to 0s goes directly from the fetch cycle to ET2. Any clock may initiate the modification of the masked bits, but the setup for that modification always occurs during ET2 so that the ET2 clock loads the result from AD into AR.

On the assumption that the mask is ready, F CYC ACT EN gates the test word into AD, and it sets up the complement or zeros modification if either of these is called for and there cannot be a skip. For either modification the test word is combined with the complement of the mask, where the AND function clears the masked bits for zeros, otherwise the equivalence function complements them. If there may be a skip, PC is put on the address bus for possible incrementing back into PC; and if there is to be a real test, the mask is gated into AD for anding with the test word (the other conditions for this event are irrelevant). A real test requires extra time, so for equal or not equal with flow going immediately to ET1 because the mask is already available (direct or right); a long cycle is called. If it turns out that the mask is not ready, the processor goes to ET0, which swaps the mask in AR and repeats all of the fetch cycle actions that set up the modification or test.

ET1 then clocks PC if there is a skip, and duplicates the mask in BR for use with ones modification. Complement or zeros modification is set up as above, but ones modification uses the equivalence function with the complements of mask and test word ored in one input mixer, and the other generating all 0s because it receives both the mask and its complement from AR. As shown in the table at the left, the OR function is 1 only where both mask and test word are 0, and the equivalence of this function with 0s produces a 1 in every bit where either mask or test word is 1. ET2 loads the (perhaps modified) test word into AR and calls for the next instruction fetch if the possibility of a skip prevented it from being called in the fetch cycle.

Arithmetic Test. In the arithmetic test instructions the fetch cycle actions include whatever arithmetic operations are necessary and the placing of PC in AB for possible incrementing in those instructions that can skip (the compare instructions as well as the so-called skips). In CAXX either E or the memory word is subtracted from AC; in the skips and jumps the word to be compared against zero (memory or AC respectively) is gated into AD with incrementing or decrementing by one if called for. ET0 then inhibits the AD clock to allow time for the test network to function on the result. In the skips and jumps ET2 places the result in AR and generates a flag strobe if there actually was an addition or subtraction.

Upon satisfaction of the appropriate test conditions, a compare or skip clocks PC. But when a skip is done in a PI cycle PC+1 is inhibited; and going on to the second interrupt instruction depends on the condition not being satisfied, which is indicated by the PC clock not being enabled. When a jump condition is not satisfied, PC is gated into AB in place of AD, which is otherwise already enabled; E is moved to BR and enabled into AD so that it provides the jump address to AB unless it has been displaced by PC. The result (which may not have changed) is written back into AC in the jump and may be in a skip. ST1 calls the next instruction fetch except on overflow in a PI skip, in which case the next instruction is handled by the second PI cycle. Storage in memory occurs only in a skip that adds or subtracts.

The conditions used in the arithmetic test are listed at the lower right and the logic for them is in drawing ADZ. The P condition is used in all of the instructions as it checks for an always skip and for equal or not equal, which can be done by a simple zero-nonzero check whether a single word is being compared with zero or two words are being compared by subtracting one from the other. Inequalities for the skips and jumps are determined by condition R which depends on the sign of the word as supplied through AD. For compare, condition Q tests for inequality – but note that the logic net for this is exactly the same as that for condition R except for the substitution of the function ADZ SIGN for the AD sign bit among the inputs. It is therefore evident that where a sign bit of 1 indicates the test word is less than zero in a skip or jump, the assertion of the sign function in condition Q must indicate that AC is less than E or the memory word.

Inspection of the net for the sign function at the left shows that its inputs are the AC sign through the B mixer, the complement of the AR sign through the A mixer, and the complement of the carry into the sign bit of the adder; the output is the OR of these functions anded in pairs. The table at the bottom left of the flow chart shows that this sign function for comparing two words is equivalent to using the sign for comparing a single word against zero. If AC is positive and AR is negative, then AC is obviously greater than AR; in this case both mixer inputs are 0 and the sign function is false, which is equivalent to the sign bit being 0 as required. Similarly AC negative and AR positive means AC is less, and both mixer inputs being 1 makes the output true. When the signs of AC and AR are the same, the mixer inputs to the net differ and we must therefore consider the carry. Both signs positive and AC the greater is case III of the addition algorithm described in section 7.1, so the carry is 1 and the sign function is false; similarly AC the lesser is case IV, which gives a carry of 0 making the function true. With both numbers negative, the same situations are again cases III and IV, but now the correspondence of AC and AR to *x* and *y* in the algorithm proof is reversed: for negative numbers, the lesser in magnitude is the greater algebraically.

8.3.4 UUOs and MAP

Flow for monitor and local UUOs is in chart UUO. LUUO at the right generates the needed addresses (40 and 41) through both the magic numbers and the MA special levels, where the set that is actually used depends on whether the instruction is done in user or executive mode. The instruction also uses XCTF so that an ordinary instruction fetch can be used to get the instruction from the second location without affecting PC. Access to the first location is handled by the store switch, with the IR-E word made up at ET1. The only unusual characteristic of this instruction is that the MA special levels for it are conditioned upon the negation of certain other MA special levels associated with a trap fetch. The reason for this is that entry into an overflow trap occurs after the instruction following the overflow is both fetched and decoded. Should that next instruction happen to be an LUUO, its MA special levels would interfere with those used to fetch the trap instruction were they not specifically disabled by the trap cycle itself.

MUUO, although lengthy, is quite straightforward except that the relationship between the three parts of the instruction and the three main sequences used by it are somewhat confused. In any other multipart instruction (byte, double move) each part is a complete main sequence. But in MUUO the conditions that define the next part come into being well before completion of the previous sequence (the logic is at the right in print MPYT). All three parts use the MA special levels for access to the user process table, and the first two parts simply construct the IR-E and PC words and store them in locations 424 and 425 of the table. In the third part the logic that generates the MA special levels effectively decodes the User and Public flags to select the pair of locations set aside in the table for each of the four modes; moreover the even location in the pair is used unless the MUUO is performed in a trap cycle, which enables MA SPECIAL 35. At ET2 the same function loads the flags as in a JRSTF, but here there are far fewer restrictions.

As shown at the right in chart KRMP, MAP begins like an ordinary write instruction, calling for a page check, but it inhibits the memory cycle. If all is well there is a normal return from the page check to the fetch cycle; but if there is a page failure of any kind, control goes to the page fail cycle, which rather than trapping simply returns to the fetch cycle in MAP. With memory inhibited, setting ET2 would normally start the write subroutine, but this is also inhibited by the instruction. ET2 loads the map data into AR via the auxiliary mixer in print ARMD, writes the data in AC, and kills the anticipated memory write to open up memory control for the next instruction fetch.

8.4 LOGIC AND FIXED POINT ARITHMETIC

The most basic arithmetic and logical operations are performed by the add-subtract and boolean instructions shown at the left in flow chart ASBM. The ADSUB sequence is entirely self-explanatory; the only possible difficulty in the BOOLE sequence is in determining the actions required for the sixteen boolean operations, and these are listed in a table at the bottom. The remaining instructions discussed here are the shift-rotate group, which includes both logical and arithmetic shifts, and the more complicated fixed point multiply and divide.

8.4.1 Shift-Rotate

The six instructions in this group (chart SR) differ only in the number of words shifted and in the shift connections at their extremities. Needing no memory operands, the sequence begins by switching AB over to PC and calling for the next instruction fetch. The execute cycle puts AC in AR, and for a double length shift it also puts AC2 in MQ and switches the fast memory address back to AC to be ready for the store cycle. All the rest of the execute events are for determining the correct number of places to shift as specified by E but without redundant shifting: in an arithmetic or logical shift information can be lost, so regardless of the value of E there is no need to shift more than 72 places; in a rotate information cannot be lost, so any set of 72 shifts is dispensable. The procedure used is to divide E by 72, generating two quotient bits by successively subtracting out first 144 and then 72. These quotient bits, which are represented in the computations by the signs of the differences, supply the desired information.

ET0 moves E to SC, saves a copy of it in BR as well, and sets up SCAD for the negative of the absolute value of E less 144. At ET1 the objective is to find out if more than 72 shifts are called for. If SC and SCAD are both positive, then E is in the range 0–144 and we simply set up SCAD for the negative of $|E|$ less 72. If the result of the ET0 computation is negative, then the instruction requested more than 144 shifts, so we set FLAG 1 to remember this fact and replace the original E in SC with the computation result; hence the number put in SC is negative and has a magnitude in the range 0–112. Finally if either E is negative (in SC) or the result of the ET0 computation is negative, we use SCAD SC+ EN to set up the negative of $|SC|$ less 72, knowing that SC contains a negative number with magnitude 0–143 either because it contains the original E in that range or contains the result of the ET0 computation.

At ET2 both signs still positive indicates an E in the range 0–72, so for controlling the shift we set up the negative of it in SCAD. Now if the result of the ET1 computation is positive but SC holds a negative number and FLAG 1 has not been set, then SC still contains the original E and it is a negative number with magnitude less than 72, so we simply gate it into SCAD. The next line in the ET2 box has the condition SCAD 0(1), meaning that the ET1 result is negative; this implies that whatever the number is in SC, its magnitude is greater than 72 so we replace it with the negative of $|SC|$ diminished by 72. The same condition appears in the next line because for an arithmetic or logical shift we call for a shift of 72 if either SC now contains a number with magnitude greater than 72, or regardless of what SC now contains, FLAG 1 tells us that the original E specified more than 144 places. Finally for rotate: if either SC or SCAD is negative, implying that more than 72 shifts were called for, we gate SC into SCAD knowing that after ET2 it will contain a negative number whose absolute value is equal to the magnitude of the original E diminished by some multiple of 72 such that it is now in the range 0–72 – in other words, the remainder. Note again that the quantity set up in SCAD from SC during the time state following ET2 may not be the same number that was tested in SC at ET2.

Following ET2 is a special time state SRT1, which loads the final computation result from SCAD into SC, this result being the negative of the number of places that will actually be shifted by the processor in performing the instruction. Now at this point there is again a test of the sign of the computation result, for if the negative in SCAD is really positive, then it must be zero and we must not shift at all. Note that a nonshift ASHC does not make the AC2 sign equal to the AC sign. For a nonzero shift AR is gated into AD (MQ shifting is done through its own gates), and we enter the standard shift-count subroutine at SCT1 for right shifting or directly to SCT2 for left shifting. The subroutine is as described in section 7.6.1 except that loss of a significant bit in a left arithmetic shift sets the Overflow and Trap 1 flags. The diagrams at the right show the configurations of the various shift types, and the equations above them indicate the connections at the register extremities for producing these configurations. Shift connections are also listed in the table on chart SCON.

8.4.2 Multiply

The fixed multiply instructions, shown in the right half of chart ASBM, use no special sequence – execution is entirely within the execute cycle, which calls the shift-count subroutine directly. MUL and IMUL generate the ordinary switches for the standard modes, some of which are in common with the fixed divide instructions. IR XMULX also generates IR MPY, which controls the actual multiplication in all multiply instructions, fixed, floating, and double floating. The first time states following the fetch set up the registers and the shift count: the memory or immediate operand, the multiplicand, ends up in BR; AC supplies the multiplier to MQ; AR is clear; and SC receives a count for 18 double right shifts. ET0 sets FLAG 3 if both operands are negative, the only situation in which a multiplication can overflow the double length product. ET0(1) initially enables the multiply function, which is carried on by SCTC to set up the adder for each multiply step throughout the shift count (print MPYT).

On paper, multiplication is performed by using digits of a multiplier to determine partial products of the multiplicand, where each is shifted left one place from the previous one, and all are added together at the end. The computer algorithm uses two bits at a time of the multiplier and keeps a running sum of the partial products, shifting each sum right two places before adding in the next partial product. Hence ET0 looks at the low order two bits of the multiplier as it is being loaded into MQ and sets up the adder for the first partial product. Each SCT2 clock shifts the double length AD-MQ right two places into AR-MQ and sets up the next sum from the next pair of multiplier bits being shifted into MQ 34–35. Thus as the multiplier is being shifted out of MQ, the low order word of the product is being shifted in. The two extra bits at the left end of the adder pick up any overflow that might occur at any stage of the process and otherwise supply null and sign bits for the right shift.

The various parts of the multiply function listed in the box between ET0 and SCT2 appear quite complicated, but in terms of each pair of multiplier bits, they result in the operations listed in the table at the bottom. With FLAG 1 initially 0, the two multiplier bits being 00 implies that the multiplicand should not be added at all, and there is only the shift; 01 causes addition of the multiplicand once, whereas 10 causes addition of twice the multiplicand, which is accomplished by adding it in shifted one place to the left. Now 11 in the multiplier bits implies that we must add thrice the multiplicand, but there is no way for the logic to do this. Therefore we instead subtract it and then add it four times. The latter operation is done by setting the flag, which causes the multiplier to be added an extra time at the next position. Hence with FLAG 1 set, 00 implies add once, 01 says add twice, and 10 means subtract but leave the flag set for the equivalent of adding 2+1 times. Finally 11 means add four times instead of three, so we simply shift and leave the flag set.

That this procedure produces a correct product is obvious for a positive multiplier, as the use of multiplier bits for adding partial products at different positions is simply equivalent to adding the multiplicand times the various powers of 2 corresponding to the 1s in the multiplier. To see that it works also for a negative multiplier, we need only remind ourselves that the twos complement of a 35-bit positive integer x is $2^{35} - x$. Hence if we use the same procedure and at the end subtract 2^{35} times the multiplicand, we have effectively used a multiplier of $2^{35} - x - 2^{35}$, or simply $-x$. Now the procedure is in fact the same and we need consider only what happens for the final multiplier bit pair with a sign bit of 1; there are four cases depending upon the values of the most significant magnitude bit and the flag. In magnitude terms these two bits correspond to the 34th and 35th powers of 2, where the latter is the sign.

For the 10 case with the flag off, the final shift adds twice, *ie* 2^{35}, making the result $2^{36} - x$. To correct for this, a final double add with the multiplier sign in MQ 34 being 1 causes the final SCT2 to set up AD for a subtraction at the next order of magnitude beyond the whole multiplier, *ie* to subtract 2^{36}. In the other three cases the algorithm itself fortuitously produces the proper subtraction with no correction needed. For 11 with the flag off, the final shift subtracts 2^{34} instead of adding the 2^{34} implied by the 1 in the MSB, and these two events in combination are equivalent to subtracting 2^{35}. The flag being on implies that the processor must add 2^{34} besides doing whatever is called for by the configuration of the final bit pair. The 10 case is therefore equivalent to the 11 case with the flag off, namely the processor subtracts 2^{34} instead of adding it. A 11 pair implies that the negative multiplier ends in a string of 1s, at the beginning of which the processor subtracts some power 2^n. Actions beyond that are limited to shifting, which means that the processor fails to add each power of 2 corresponding to the 1s in the string. The failure to add 2^{34}, 2^{33}, . . . , and 2^n combined with subtracting 2^n is equivalent to subtracting 2^{35}.

After a shift of 36 places, the low order word of the double length product is in the correct position in MQ, but the high order word overlaps into MQ 00, so ET1 shifts AR left one. For IMUL the high order word in positive form is gated into AD, so ET2 can check for the presence of significant bits, which would indicate overflow, while it brings the low product into AR but keeps the sign of the double length product. Overflow is also indicated by a negative product when FLAG 3 is on, meaning both operands are negative – in other words -2^{35} squared.

Storage of the AR word is in AC, memory, or both in the usual fashion. But for Basic, Immediate or Both mode of MUL, the store cycle goes to ST4. This clock transfers the low product from MQ to AR, keeping the sign of the high part, and writes it in AC2.

8.4.3 Divide

Chart DIVS shows the fixed divide instructions and the divide subroutine used by both fixed divide and single floating divide. The time states and other control nets for the subroutine are in print DS. Both DIV and IDIV have the standard modes and share some switches in common with fixed multiply.

The first two clocks after the fetch place the memory or immediate operand, the divisor, in BR and set up SC for 34 left shifts. The rest of the setup prepares the dividend for the divide algorithm, which operates only with a dividend in positive form. AC supplies the high part of the dividend for DIV, but the whole dividend for IDIV. In either event the complement of AC is initially brought to AR, and ET0 sets up the adder to convert it into a positive number. If AC is positive, AD recomplements the complement; conversely if AC is negative, AD converts the complement into the negative, and ET0 sets FLAG 3 to indicate that AC originally supplied a negative dividend.

For IDIV the number in AD is most likely the correct form of the dividend for use by the algorithm, and ET1 moves it to MQ and clears AR; it thus becomes the low order part of a double length dividend whose high part is null. However ET1 also sets up the number 1 in AD, so that if the supposedly positive dividend in MQ turns out still to be negative (-2^{35}), ET2 moves the 1 into AR 35 to form a double length dividend of 2^{35}.

For DIV the negative of the high dividend is probably wrong, because it should be a complement unless the low part is null. But note that if AC is positive, the high part is correct in AD, and in any event the complement of AC is saved in AR. ET0 switches the fast memory over to AC2 for the low dividend. ET1 moves the AD form of the high dividend to MQ and gets the low dividend in positive form. ET2 then moves the positive low dividend to MQ, and if either the original dividend is positive or the negation of the low dividend overflowed, it moves the high dividend (in its original form or negated respectively) from MQ to AR. If the original dividend is negative and its negation produces no overflow, then there is no movement of the MQ form of the high dividend, for AR already correctly contains the complement of AC.

After the dividend is set up, the divisor is compared with it to determine whether the division can be performed. ET2 subtracts the absolute value of the divisor from the high half of the dividend (if the divisor is positive, subtract it; if negative, add it). Since the dividend is positive, the result is also positive if the magnitude of the divisor is less than or equal to the number in AR. For a fixed fraction, the divisor is subtracted from the actual dividend and no overflow is allowed. For a fixed integer, AR is usually clear and the result is positive only for a zero divisor; the worst case is the division of $2^{35} - 1$ by 1, whose integral result can be accommodated. (Placing the one word dividend in MQ effectively multiplies it by 2^{-35}, making it the fractional part of a two word dividend with the binary point in the middle. The quotient is then a proper fraction, which is multiplied by 2^{35} simply by interpreting it as an integer.) However should the original dividend be -2^{35}, AR is not clear, and the result is positive when the divisor is ±1.

If the result of this initial subtraction is positive, DST2 sets Overflow and No Divide, and terminates the procedure so the processor goes on to the next instruction. Dividing by zero is of course meaningless. Placing a 1 in AR 35 allows use of -2^{35} as a dividend, but it cannot be paired with a divisor of -1 because the result of 2^{35} would be too large to accommodate in a single register. (The prohibition on dividing -2^{35} by $+1$ is just a happenstance of the logic.) The reason for prohibiting a fractional division where the result would be greater than 1 is that it is impossible to determine the position of the binary point in the quotient. If the result of the initial subtraction is negative, the division can be performed and the processor goes into the division shift-count at SCT2.

In division on paper, one subtracts out the divisor the number of times it goes into the dividend, then shifts the dividend one place to the left (or the divisor to the right) and again subtracts out. In binary computations the divisor goes into the dividend either once or not at all. Each subtraction of the divisor thus generates a single bit of the quotient. If the subtraction leaves a positive difference, *ie* if the dividend is larger than the divisor, a 1 is entered into the quotient. If the difference is negative, a 0 is entered. To compensate for subtracting too much, the hardware could add the divisor back into the dividend before going to the next subtraction step. But the algorithm instead shifts first and adds the divisor back in at the new position. It then continues to shift and add putting 0s into the quotient until the result again becomes positive. This procedure generates the same quotient without ever going back a step.

The hardware procedure is as follows. As each addition or subtraction is formed in the adder, the result is put in AR shifted one place to the left with AR 35 receiving a new bit of the dividend from MQ 01, and MQ is shifted left bringing in a bit of the quotient at MQ 35. The bit brought in is equal to the carry out of AD 00, which is equivalent to the complement of the adder sign: if the divisor does not go into the dividend, the resulting minus sign (1) produces a 0 quotient bit; if the divisor does go in, the plus sign gives a 1. Each step loads one bit of the quotient into MQ 35, and the low half of the dividend is shifted out of MQ as the quotient is shifted in.

The first step is the test subtraction. Each subsequent step subtracts the absolute value of the divisor if the quotient bit generated in the previous step is 1, but adds it back in if the quotient bit is 0. Since the divisor may have either sign, it is subtracted algebraically if its sign differs from the quotient bit or added if its sign is the same.

The hardware executes 36 steps to generate 35 magnitude bits. The initial DST1 test step must give a 0, which serves as the sign since we are producing a positive quotient. The final DST3 step puts the result of the addition or subtraction directly in AR without shifting so the remainder is in the correct position, but it shifts MQ left putting the sign from the first step in MQ 00 and bringing in the last quotient bit.

To complete the division we must make sure the remainder is correct and determine the correct signs of the results. Since the operations are performed on positive operands, the remainder should also be positive; a negative remainder means that too much has been subtracted. To correct this, DST4 adds the absolute value of the divisor back in if the final quotient bit is 0. If the negative dividend flag is set, DST4 sets up the adder to negate the remainder, so it has the sign of the original dividend; otherwise DST4 simply gates AR into the adder. DST5 moves the quotient to AR and the corrected remainder to MQ. If the negative dividend flag and the divisor sign are the same, the quotient should be positive so the correct quotient and remainder are now in AR and MQ ready for storage. But if the signs of the divisor and the original dividend differ, DST5 sets up the adder to negate AR, and the processor enters the store cycle via DST6, which puts the quotient with correct sign back in AR. Storage is typical, with AC2 receiving the remainder except in memory mode.

As an example of the way this algorithm operates, consider a division of 3-bit fixed fractions with a dividend of +.100100 and a divisor of +.101. By paper computation we obtain the quotient this way.

```
          .111
   101 |100.100
        10 1
        10 00
         1 01
           110
           101
             1
```

Taking the processor registers to be four bits in length, AR contains 0.100, MQ has 0.100, and BR has 0.101. The sequence has four steps.

```
         0.100      .100
        -0.101
        ------
         1.111
1 ←      1.111      .000
        +0.101
        ------
         0.100
2 ←      1.000      .001
        -0.101
        ------
         0.011
3 ←      0.110      .011
        -0.101
        ------
         0.001
4        0.001    ←0.111
```

The quotient is in MQ at the right, the remainder in AR at the left.

8.5 SINGLE PRECISION FLOATING POINT ARITHMETIC

The floating point repertoire includes the four basic operations of addition, subtraction, multiplication and division, each in eight forms: four with rounding including an immediate mode, and four without rounding including a long mode. The execute cycle of any instruction includes ET0 only in immediate mode to swap E to the left. The general procedure for all instructions is to compute the exponent of the result first while setting up the operands, and then to perform the indicated operation on fixed point fractions with the signs smeared – *ie* with all bits of the exponent part made equal to the sign. Except in division, the result is double length with the sign in AR 00 and the fraction in AR 09–35 concatenated with MQ 08–35. Note the use of bits 8–35 of MQ even though the fraction in a floating point word is 27 bits in bits 9–35; in other words the computations keep an extra bit beyond double length. In division the quotient is single length and MQ holds the remainder. For shifting, the hardware treats all of AR and MQ 08–35 as a single double length unit; this includes the exponent part of AR with its bits nullified, but the connection between the registers is directly between AR 35 and MQ 08.

The execution of the four basic operations is shown in three flow charts, as subtraction is done by negating the subtrahend and then adding. None of these flow charts is complete, however, as every instruction execution sequence ends by going to the floating normalize and round routine, FNR (section 8.5.5). This routine is shown in a separate chart that includes storage, which is a special sequence for long mode but is simply the store cycle for the others. At the beginning of the normalize routine, SC always receives the negative of the exponent of the result so the routine can use the SC+1 gate to count down the exponent while shifting the fraction left. Of the miscellaneous floating point instructions, FSC and FLTR both go to FNR. UFA is included in the add-subtract chart; it ends by going to FNR, but not for normalization – only to adjust for possible fraction overflow and to store the result. The flow charts for FIX and DFN are complete.

In exponent computations the true exponent is the actual power of 2, *ie* a number in the range −128 to +127. Do not confuse the sign of the exponent with the sign of the number, which is the sign of the fraction. In a positive number the exponent part is the excess-128 code of the true exponent; in a negative number the exponent part is the complement of the excess-128 code of the true exponent. The transfer of the exponent part of a number from AR to the SC adder automatically converts it to the true exponent or its complement. This is done by gating the complement of AR 01 into both SCAD 0 and SCAD 1. Note that SC and SCAD have nine bits, numbered 0–8 where bit 0 is the sign and bits 2–8 receive the exponent. Hence bit 1 is actually an extra most significant magnitude bit, allowing computations in the range −256 to +255. For any result in the proper range, the MSB must be null, *ie* equal to the sign. Overflow is therefore indicated by the sign and the MSB differing.

Underflow is specifically a result with an exponent that is too small, *ie* less than −128. Should extreme underflow carry the exponent below −256, the sign changes and the result looks positive. Now the preliminary exponent computation cannot possibly overflow that much – the worst case is 2^{-128} squared, resulting in 2^{-256}, and only normalization can make it more negative. It is also obvious that only a negative exponent can underflow, as normalization can never shift 256 places. Therefore, before the processor enters the normalize routine, it sets a hold flag if the preliminary exponent result is positive. Then after normalization, overflow with the hold flag off is taken as an indication of underflow.

8.5.1 Floating Add-Subtract

Chart FASU shows the fetch and execution sequence for UFA as well as for all forms of the floating add and subtract instructions. The fetch cycle for UFA simply gets the operands, puts the memory exponent in SCAD, and goes to ET2. The setup sequence for add and subtract is basically the same, but it includes ET0 if done in immediate mode, and in subtract it includes ET1 to get the negative of the AR operand through the adder.

In floating point addition the exponent of the result is simply the exponent of the larger summand, but the fractions must be aligned so that corresponding bits in the addition are of the same order of magnitude. The addition actually occurs at the end of the sequence with the result going to AR at FAT4. The chain of events from ET2 to FAT4 first determines which exponent is larger and gets the difference between them. It then uses this information to put the fraction of the smaller number in AR and shift it right to the correct magnitude position with respect to the larger number in BR.

ET2 places the memory or immediate operand in BR with its sign smeared and places AC in AR. It then manipulates the exponents in such a way that SCAD contains the difference between them, and the signs of AR and SCAD give the relationship between the exponents as indicated by the tables accompanying the ET2 box. FAT1 sets FLAG 2 if the smaller number is already in AR and makes sure that the difference is available in negative form for controlling the shift. FAT2 puts the negative difference in SC if it is not already there and puts the complement of the larger exponent in SCAD. Note that if the exponents are equal it matters not which one is used, and it may in fact come from either operand depending on the circumstances.

FAT3 juggles the operands so that the larger one always ends up in BR. The smaller one ends up in AR unless the difference between the exponents is greater than 64, in which case AR is cleared and the smaller operand is thrown away since it can have no effect on the result. Now the larger operand may already be in BR, but if it is not, FAT3 moves it from AR to BR with the sign smeared. Events involving the smaller operand take place only if that operand is of significance: if it is already in AR its sign is smeared, but if in BR it is moved to AR. FAT3 also saves the complement of the larger exponent in FE, and if either the exponents are the same or the smaller operand has been thrown away, it sets up the addition of the fractions in AD. If the difference is nonzero but does not exceed 64, we enter the shift-count to move AR-MQ right the number of places specified by the exponent difference, and the final SCT sets up the addition. FAT4 returns the exponent complement to SCAD and puts the sum of the fractions into AR-MQ. Overflow of the fractional addition is indicated by bit 8 being of significance, *ie* being different from the sign. If there is no overflow, the sum is moved directly into AR from AD and SCAD is incremented to change it from the complement of the exponent to the negative. If there is overflow, the sum is loaded into AR-MQ shifted one place to the right, and there is no need to adjust the exponent. The sequence then goes to the normalize and round routine.

8.5.2 Floating Multiply

The execute cycle of FMP, at the left in chart FMSC, puts the multiplicand in BR and the multiplier in AR with their signs smeared, and FMT1 moves the multiplier to MQ where it is used to control the multiplication procedure. In the meantime ET1 produces the intermediate function of the exponents given in the table at the right of the ET1 box, for saving in FE at ET2. FMT1 begins the multiplication, which is done in exactly the same manner as fixed multiply (section 8.4.2) except there are 14 double steps instead of 18. Since the AR-MQ shift connection is directly between magnitude bits, FMT2 shifts the entire product left one to compensate for the right shift associated with the multiplier sign. The final SCT2 retrieves the intermediate exponent function from FE, and FMT2 converts it to the negative of the sum of the exponents of the operands. The sequence then goes to the normalize and round routine.

8.5.3 Floating Divide

Chart FD shows the floating divide instructions, which use the same divide subroutine as DIV and IDIV with a number of extra events, mostly to handle the exponents. The algorithm requires the dividend in positive form, and this is single length except in long mode. ET1 and ET2 put the divisor in BR with its sign smeared. The first of these clocks computes the intermediate exponent function given in the table inside the ET1 box, ET2 converts it to the exponent of the result, *ie* the dividend exponent minus the divisor exponent, and FDT0 saves it in FE. At ET1, AR receives the dividend, which is the high dividend for long mode. If the dividend is negative, ET1 sets FLAG 3 and negates AR in AD, so that at ET2 the positive form is substituted provided the fraction part is not zero. A zero fraction in a negative floating point number is here taken to represent the fraction -1, so FDT0 smears the exponent part of AR if it is now positive, but otherwise substitutes +1 by setting bit 8 and clearing the rest of AR. For long mode, ET2 gates the low dividend in positive form from AC2 into AD, and FDT0 puts it in MQ. Moreover if the low dividend had to be negated and did not overflow, FDT0 converts the high negative to the complement by adding it to all 1s supplied by the ungated adder input mixer. FDT1 puts the correct high dividend back in AR and shifts the low dividend into MQ 08–34.

FDT2 begins the division procedure by generating AD CRY 0 to force the initial subtraction, but this subtraction is done from a dividend that is shifted right one place in AR and MQ. The shift divides the dividend in half so that if normalized operands are used, the division can certainly be performed except in the case of a zero divisor. FDT2 also sets up the shift count for 26 steps in an FDV, or 27 steps if an extra quotient bit is needed for rounding. From this point on the division is performed in the same way as for fixed divide (section 8.4.3), with an initial divide check and a sequence of subtractions and additions to build up the quotient. However an additional test is made at the computation of the first magnitude bit. If the result of the second step at FDT4 is positive, it is evident that it was necessary to halve the dividend initially, and the FDT4 step is therefore counted as part of the shift count with FLAG 1 being set to remember this fact. In other words, without rounding the procedure requires 28 steps: the initial test at FDT3, 26 steps at SCT2, and the final step at DST3. If the prior dividend shift is not necessary, the FDT4 step amounts to no more than a second sign test and is simply thrown away. However if it turns out that the shift is necessary, the FDT4 step is counted in SC and the number at SCT2 is reduced to 25 so the total is still 28.

The final time states in the divide subroutine make sure that the remainder is correct, adjust the signs, and put quotient and remainder in AR and MQ respectively as in the fixed sequence, but here they also handle the exponent and take care of rounding. DST3 negates the exponent, but eliminates the addition of 1 in the negation if FLAG 1 is set; dropping the 1 forms the negative of the exponent increased by 1 to compensate for dividing the dividend by 2 initially. In an FDVX with a positive quotient, the sequence goes directly from DST5 to the normalize routine. If the signs of dividend and divisor differ, DST5 negates the quotient. In an FDVRX the calculated quotient actually has 28 bits; in this case DST5 adds 1 to the quotient if it is positive, and DST6 throws away the remainder and drops the extra quotient bit by shifting right one place. Note that the addition of 1 to the quotient actually changes the true 27-bit quotient only if the extra 28th bit is 1. A negative quotient is rounded simply by dropping the extra bit.

8.5.4 Miscellaneous Floating Point Instructions

The right part of chart FMSC shows FSC, which simply adds E as an exponent to the AC exponent and then enters the normalize routine with the negative of the exponent result in SCAD and AC in AR with its sign smeared.

The remaining instructions, FIX, FIXR, FLTR and DFN are together in chart FFDN. At the end of the fetch cycle, DFN sets up the complement of the exponent part of the low word from memory, and ET0 gates the complement of the high word from AC into AD. The course of action from this point depends upon the low word fraction. If the low fraction is zero, ET0 converts the high complement to a high negative, and the low word is returned to memory unchanged. But for a nonzero low fraction, the complement is the correct form for the high part, and ET0 complements the sign and exponent parts of the low word so they end up unchanged when the low word is negated at ET1. FLTR simply shifts the operand right eight places to the fraction position in AR-MQ, sets up SCAD with the negative of the exponent appropriate to interpreting the number as a fraction rather than an integer, and goes to the normalize routine. The exponent is of course 35, as an integer is changed into a fraction by moving the binary point 35 places from the left end to the right end.

Instructions that fix a floating point number are a little more complicated than the others discussed here and have two special execution time states that are shown on print DS. ET0 subtracts 27 from the true exponent of the floating point operand, so that the revised exponent represents the order of magnitude of the fraction interpreted as an integer. The objective then is to shift the integer to a position such that its exponent vanishes, with left shifting required to get rid of a positive exponent, right shifting for a negative. For shifting to be worthwhile and provide a valid result, the exponent must lie in the range -27 to +8. More than eight shifts left would lose the most significant bits; thus for this case the instruction terminates with Overflow set and the original floating point operand left unchanged in AC. An exponent less than -27 means the integer would be shifted away, so for this ET2 clears AR and storage subsequently clears AC. For other cases the instruction shifts the word in AR with its exponent part nullified; the shift direction is determined by the exponent sign from FE. For a negative exponent, we go directly to the shift-count subroutine to shift right until the exponent is counted up to zero. A positive exponent is first negated, and we then use it to shift left, unless the exponent is 0, in which case we store the result as it is. Following the shift we go to FIXT2 for rounding in the positive direction or adjusting a negative for standard Fortran truncation. FIXR adds 1 to AR if the high order bit of the part shifted out at the right is 1. FIX adds 1 to a negative number if any bit of significance is shifted out to the right.

8.5.5 Floating Normalize and Round

Chart FNR shows the routine that normalizes and stores the result of a floating point operation. The routine also handles rounding except for division, which does its own rounding from an extra quotient bit rather than a low order word in MQ. As we enter the routine at the upper left, SCAD contains the negative of the exponent, both AD and AR hold the high fraction, and MQ holds the low fraction except in division, where it either has the remainder or is clear. If the result is zero we go directly from NRT1 to the store cycle.

In all other cases the objective is to get to NRT4 with the result normalized and ready to store. NRT1 sets AR FXU HOLD if the exponent is positive, performs an initial test to determine whether the result is normalized, and can handle rounding for a result already normalized. NRT2 is the actual normalization loop and it can also handle rounding. Each normalization step shifts the result left one place and decreases the exponent by 1; bits shifted into AR 35 come from MQ 08 except in divide, where 0s are shifted in. NRT3 fixes overflow caused by rounding and guarantees the correct form for a negative. Hence we may go directly to NRT4 from NRT1, or to NRT4 via NRT2, or to NRT3 either directly from NRT1 or via NRT2 and thence to NRT4.

There are actually two normalization tests: NRT1 checks for AR 09 different from the sign to see if a number is already normalized, whereas NRT2 checks for AR 10 different from the sign to determine whether the number will be normalized after the present NRT2 normalization shift. Similarly NRT1 checks MQ 08 for rounding, whereas NRT2 checks MQ 09 for rounding along with a normalization shift. An instruction that calls for rounding, including FLTR, is indicated by a 1 in IR 06. The rounding action is the usual one, *ie* that 1 is added to the high part if the MSB of the low part is of significance in a positive number or is of no significance in a negative number. The actual route through the time states to NRT4 is determined by the three levels NR RND, NR POS and NR NEG, which may be generated in various configurations by either NRT1 or NRT2. The exact meanings of these conditions are given at the center of the chart. The reader should note that for a UFA with a positive result, we always go directly to NRT4 because overflow of a positive fraction is handled by the add-subtract sequence. But any instruction with a negative result must go to NRT3 in case the fraction is -1, which fulfills the normalization condition that the true fraction MSB differs from the sign; it is not, however, regarded as normalized and is changed to -1/2 with appropriate adjustment of the exponent. NRT3 also does a right shift with an increase in the exponent if rounding action at NRT1 or NRT2 changes the fraction to +1. In any case the time state preceding NRT4 sets up the true exponent or its complement depending upon whether the result is positive or negative.

NRT4 inserts the exponent in the high word, takes care of the overflow flags, and for any but a long mode instruction, calls the store cycle, writing the high result in FM except in memory mode. In most cases the fast memory location is AC, but in UFA the result goes to AC2 due to the FMC action taken by UFA at ET2. For long mode other than divide, NRT4 writes the high result in AC, moves the low fraction to the proper position for a floating point word, and goes on to two more NR time states. The first of these switches fast memory over to AC2 and subtracts 27 from the exponent. The second inserts the low exponent in the low word in AR, but clears AR if either the low fraction is zero or the low exponent overflows.

For FDVL the normalize routine enters a sequence of three additional floating divide time states. For this NRT4 sets up the constant -27 in SCAD and gets the positive form of the exponent from AC, which still contains the original dividend. FDT5 puts the quotient in AC and computes the remainder exponent by subtracting either 26 or 27 from the dividend exponent, depending upon the actual number of steps in the division. FDT6 puts the remainder exponent in correct form. FDT7 stores the remainder in AC2 with the exponent inserted, but clears AC2 if the remainder is zero or its exponent is less than -128.

8.6 DOUBLE PRECISION FLOATING POINT ARITHMETIC

Four of these eight instructions simply move a two-word operand or its negative in hardware double precision format between accumulators and memory. The remaining instructions handle the four arithmetic operations, and all use a pair of two-word operands, have no modes, and store a two-word result in AC and AC2. The result is rounded except in divide. In execution, these instructions are very similar to single precision floating point, but some extra complication results from having to juggle more words among the same number of registers. On the other hand there is only one exponent to compute, and there is generally no need for the sign smear, as ARX and ADX have only sign and fraction, and the low word in an operand has a 35-bit fraction. Moreover the logic is constructed assuming normalized operands: the double normalize routine allows only one shift for multiply, and divide computes a quotient that is assumed normalized, which is the case provided normalized operands are used.

The result is formed in ARX-AR-MQ, used as a triple length shift register bypassing both AR 00 and MQ 00. Because of the large number of words that must be handled, MB is used for temporary storage in the multiplication and division procedures, which are done in two parts. In all instructions one operand is held in the register combination FM-BR. No particular distinction is necessary between the operands in multiply – which one is used as the multiplier is of no import – so FM-BR holds the AC-AC2 operand. In add-subtract it is the smaller operand that must be shifted, so the high memory operand may have to be loaded into fast memory; and in divide the memory operand is the divisor, which must be in FM-BR. Furthermore do not assume that the FM location used is necessarily AC; that is the case in add-subtract, but divide uses the AC2 location to hold the high divisor. For all operations involving the extended adder, the reader must keep in mind that ADX has only one input mixer, and ARX is always applied to one set of ADX inputs. Often the transfer of a word *X* is made into ADX by performing the equivalence of ~*X* with zero. And there is actually one instance, in add-subtract, where a word is gated into ADX mixed up with the contents of ARX, and it must subsequently be unmixed.

As with single precision, add and subtract use the same execution sequence, so there are three charts for the four operations, but only divide is complete; add-subtract and multiply end by calling the double normalize routine, which is on chart DN (section 8.6.3). The four double moves are together on one chart.

Except for the two instructions that move a double operand to memory, all require the fetch of a double operand from memory. The details of this double operand fetch are shown at the left in chart DFFM. The fetch is controlled by the level IR DPOP, which also gates many of the basic execute cycle operations required by all of the arithmetic instructions. These general events do not affect the double moves to AC, as they are either irrelevant or the time states in which they occur are skipped. At the beginning of the fetch cycle, IR DPOP calls an ordinary synchronous read for the high memory operand, but it also sets MCS FCE2 START so that memory control will recycle automatically to fetch the low operand. While the first memory subroutine begins, the processor goes on to an extra time state, FT7, which sets up the address E+1 in AD for use by the second subroutine when it begins. At the end of the execute cycle ET2 stops the clock, sets FCE2 WAIT for synchronization with receipt of the low operand, and gates MB into AD so that the low operand will be available at AD following its arrival in MB. The remaining fetch cycle, ET0 and ET1 events are for getting the accumulators and manipulating the exponents. They are listed in the DPOP sequence so the reader will realize just what is supplied by the general sequence as against an individual instruction sequence, but all events are listed in the instruction flows as well.

8.6.1 Double Floating Add-Subtract

These instructions, shown in chart DFAS, begin with exactly the same exponent manipulations as are carried on by single floating add-subtract (section 8.5.1), *ie* they get the complement of the larger exponent for the result and the negative form of the difference between the exponents for shifting the smaller operand. There is however one refinement in the procedure: if FLAG 2 is set at ET1, it can remain on at ET2 only if the negative form of the difference is in fact negative. In other words if the flag happens to be set even though the exponents are equal, it is then cleared to avoid unnecessary operand juggling.

While these exponent manipulations are going on, the execute cycle gets AC2 into BR, high memory into ARX, and for DFSB, the complement of ARX in ADX. We also have AC in AR, but it is dropped because we will get it again from FM anyway. If the memory operand is insignificant, DFAT1 throws it away and ARX-AR is clear. But otherwise DFAT1 loads the memory operand into ARX-AR, and note that for DFSB, ARX contains the complement of the high memory word. Now the objective of all the hair below the DFAT1 box in the right half of the chart is to arrive, with the sum of the fractions and the negative of the exponent, at the DNT1 condition at the lower right corner for exit to the double normalize routine. Immediately preceding the exit are two paths. The one at the far right is solely for the situation where the AC operand is of no significance; in this case ARX-AR already contains the memory operand for DFAD or its negative for DFSB, and we simply get the exponent complement and convert it to the negative. The other path is through DFAT5, for which the preceding time state, whatever it is, sets up the addition of FM-BR with ARX-AR in ADX-AD. DFAT5 puts the sum in ARX-AR-MQ shifted one place to the right to void any fractional overflow, and the shift means that the number supplied by FE is the exponent negative.

Now let us consider the various paths from DFAT1. The path-determining conditions are functions of the relationship between the exponents of the operands and the size of the difference between them. FLAG 2 being on means that the memory operand is larger, and SC 0 being 0 means that the exponents are equal. Since SC holds the difference in negative form, SC bits 1 and 2 must both be 1 for a nonzero difference to be less than or equal to 64, which must be the case for the smaller operand to be of any significance. For a subtract in which the AC exponent is more than 64 greater than the memory exponent, FM-BR contains the AC operand and we go directly to DFAT5 to add it to zero. For any other subtract, DFAT1 sets up the negation of the memory operand and DFAT2 loads it back into ARX-AR. Note that negation requires use of the complement with the adder and we cannot gate the complement of ARX into ADX, but for subtract ARX already contains the complement of high memory.

The main multibranch point occurs at the pseudo-time state DFAT ARGS RDY, which is true during DFAT2 in subtract or DFAT1 in add. From this point we go directly to the far right path if the AC operand is of no significance, and we go directly to DFAT5 if either the exponents are equal (meaning no shifting required) or the memory operand is of no significance. Note however that this last situation is for DFAD only; as mentioned above, the equivalent situation for DFSB bypasses DFAT2 to go directly to DFAT5. Finally for the case where both operands are significant and have unequal exponents, we go down the main flow line in the middle of the chart. If the AC operand is the larger, we skip directly to the shift-count to move the memory operand to the right. But when the memory operand is the larger, we must switch the operand positions; so the DFAT3 condition gates AC-AC2 from FM-BR into ADX-AD, but the high part of this is done in a very round-about fashion because ARX happens to hold the high memory operand. Hence we actually form the equivalence of the AC complement with ARX. Then DFAT3 moves ARX to BR for writing in fast memory, and it also forms the equivalence of the complement of the previous ARX from BR with the previous mixed up result now in ARX. This double equivalence gives us AC in ADX, and DFAT4 moves it to ARX while moving AC2 to AR and low memory to BR. We then enter the shift-count whose completion takes us to DFAT5.

8.6.2 Double Floating Multiply

The double multiply sequence in chart DFFN is so similar to single multiply as almost to require no comment. One significant difference is that the condition for SCAD +1 at ET0 is that AC be positive instead of negative. This happens because the same gate is used for both double divide and double multiply. The result is that the function of the sum of the exponents stored in FE (as given by the table above the ET0 box) is 1 less than the function used in the single case. The negative of the result exponent is then arrived at by twice adding 1, at the final SCT2 and at DFMT3.

Since the multiplier must be held in MQ for the multiply procedure, ET0 smears the sign of its high part in order to provide the sign at bit 8. The multiplicand is the AC operand, and it is held in FM-BR while being used to add partial products in ARX-AR-MQ. During the first part of the multiplication, the high multiplier is saved in MB while the low multiplier is used from MQ. The first part has one single and 17 double steps for the 35 magnitude bits in the low multiplier. At the completion of the first part, DFMT2 throws away the lowest order word of the product (which would contain four words altogether) by loading the high multiplier into MQ. The second part has 14 double steps, and from DFMT3 we go to the double normalize routine with a triple length product and the negative of the exponent.

8.6.3 Double Normalize

Chart DN shows the routine that normalizes and rounds the result of a double floating add-subtract or multiply. The final DFA or DFM time state gives the DNT1 condition, which gates ARX-AR into ADX-AD with a 1 added into AD bit 8 by way of the magic numbers. This rather mysterious operation is done to avoid the extremely long normalization shift that would otherwise occur when a fraction has a very large number of leading zeros. A positive number with 35 high order null bits has 0s in all of ARX and in AR bits 1–8, and a test for this condition is available in the logic. Adding 1 into the 35th magnitude bit of a number that begins with a string of 36 1s produces a carry out of ADX 00, giving us a way to check for 35 high order null bits in a negative number.

DNT1 puts the negative of the exponent into SC from SCAD and decides which of the many paths to take into the routine. If the high order 35 bits of the result are of no significance, we follow the leftmost path to DNT2-DNT3, which moves the low order 27 bits of AR to ARX via BR and ADX, moves the lowest order word from MQ to AR, adds 35 to the exponent, and sets FLAG 1. DNT3 also takes us back to DNT1, setting up the same negative null test as the initial entry to DNT1, but without another instruction fetch. If the next 35 bits are also null, we can repeat the DNT2-DNT3 path, but only for a negative number because FLAG 1 is now set. For a positive number with the next 35 bits null, we go to the rightmost branch to clear AC and AC2. Note the implications of these actions. If the high order 70 bits of a positive number are null, we take the answer to be zero. On the other hand, for 70 high order null bits in a negative number, we go through the leftmost path twice and then normalize the leftovers such as they be.

The actual normalization is accomplished by a loop at DNT4. If the high order word of the fraction does contain significant bits and ARX 09 is the same as the sign, DNT1 takes us to DNT4, which shifts the whole business left 1 and decreases the exponent by 1. The test made at DNT4 is whether the number will be normalized following the current shift; in other words we now check ARX 10 against the sign. However the loop is used only for add-subtract. We return to DNT1 once the number is normalized, but we return immediately following the first DNT4 in DFMP. Multiplying two normalized operands generates a product that can be unnormalized by at most one place, and DNT4 sets FLAG 2 to prevent DNT1 from taking us into the loop a second time should a single shift fail to normalize a product.

Once the result is normalized or FLAG 2 has been set, DNT1 takes us to one of the three remaining branches. If the highest order magnitude bit in MQ is of significance in a positive number or is of no significance in a negative number, we follow the third branch from the left. Here the DNT5 condition adds 1 to the double word in ARX-AR, gets rid of the lowest order word, and sets up SCAD with the negative of one greater than the exponent. The actions at DNT5 depend entirely upon the sign of the fraction. For a positive we shift the fraction right one and take the adjusted exponent in case the rounding may have overflowed; and we then return to DNT1, perhaps to go again to DNT4 to do one more normalization step for rounding that did not overflow. For a negative we take the rounded fraction as it is and go to DNT7.

Except for a zero result, we must eventually get to DNT7 once the fraction satisfies the normalization test condition or at least FLAG 2 has been set (which we will take to mean a normalized result). For a normalized positive fraction wherein the rounding has been done or is not needed, we go directly to DNT7 from DNT1. For a normalized negative, we go to DNT7 from DNT5 after the rounding has been done, or from DNT1 via DNT6 if no rounding is needed. DNT6 is solely for the situation where the fraction is −1 and hence looks normalized but must be changed to −1/2. The DNT6 condition sets up the negative of one greater than the exponent, and if the double length negative fraction is all 0s, DNT6 takes the adjusted exponent and shifts the high order word right one.

No matter how we get to DNT7, the preceding time state produces the DNT7 condition to switch over to the true exponent. The final three DNT clocks put the high and low order words in AR and MQ, handle the overflow flags exactly as in single precision instructions except using the left two bits of the exponent in SC rather than in SCAD, and insert the exponent into the high order word in the form appropriate to the sign of the fraction. Storage is in AC and AC2, with the transfer of the low word from MQ to AR forcing the sign to 0.

8.6.4 Double Floating Divide

Overall, the double floating divide sequence in chart DFD is very similar to single floating divide described in section 8.5.3, but differs in many details. The execute cycle handles the exponents in exactly the same way as FDV. ET0 puts AC in ARX and makes the complement of it available in ADX. ET1 puts AC2 in both AR and MQ. If the dividend is negative, ET1 puts the AC complement in ARX and uses it to negate the dividend in ADX-AD. By ET2 the high and low parts of the dividend in positive form are in ARX and MQ, the original AC2 is in AR, the high divisor is in BR, and we are ready to receive the low divisor via AD from the second operand fetch. In the first few DFD time states, additional juggling results in AC2 saved in MB, the divisor in FM-BR where FM is the AC2 location, and half the positive dividend in ARX-AR. The halving of the dividend is done at DFDT1, with the shift taking place in ARX and MQ. The shift drops out the dividend LSB, but it is retrieved from MB 35 in the left shift at the initial divide test step at DFDT3. Would you believe this is the reason we use the AC2 location in fast memory – so that it is AC2 that is saved in MB.

If the test fails we go to the short sequence beginning with DFDT10 to handle the overflow flags and restore the original contents of AC2. If the test makes it, we go to DFDT4 and subsequently to an SCT2 loop to generate the high quotient fraction in MQ. As with FDV, the step at DFDT4 may or may not be counted depending upon the validity of the quotient bit generated. However the procedure used for this is exactly the opposite of that used in FDV. In the single precision case, FLAG 1 is set and the initial magnitude step is counted as the first step in the shift-count if the right shift is necessary. Conversely, here FLAG 1 is set and we inhibit the count at the first SCT2 if the halving of the dividend is *unnecessary*. The inhibit means that the DFDT4 step is discarded, and we repeat SCT2 27 times in spite of having loaded a count of 26 at DFDT2.

Double floating division takes longer than the maximum time allowable for delaying the start of an interrupt. Hence for each iteration of SCT2 in the first part, we require not only that the count be incomplete but also that there be no interrupt waiting. If an interrupt has been synchronized, we terminate the loop, go to DFDT11 to restore AC2, and fetch the DFDV over again knowing full well it will be aborted by the interrupt.

Once the high part of the quotient is completed, the time left is short enough so we need not bother with interrupts, and we can therefore dispense with AC2, enable the PC+1 gates, and put up TRAP SATISFIED. DFDT5 and DFDT6 do some more word juggling to save the high quotient in MB, and we enter a second part of 35 steps to generate the low quotient fraction. The final SCT2 gets the quotient dividend from FE and increases it by 1 if necessary to compensate for right shifting the dividend. DFDT7 brings in the last quotient bit, checks the divisor sign to adjust FLAG 3 so it indicates that the quotient should be negative rather than the dividend was negative, and brings the adjusted quotient exponent into SC. It also forms the complement of the FE exponent in SCAD so DFDT8 can set the hold flag if the exponent is positive. Of course there can never be enough underflow to require use of the hold flag because we assume the result is already normalized, but we must set it anyway or any overflow will be interpreted as underflow.

If the entire quotient is zero we take the rightmost branch from DFDT8 and clear both accumulators. If the quotient should be negative and the low fraction is nonzero, we go to DFDT9 to negate it by negating the low part and complementing the high. Other cases are handled directly by DFDT8, which simply gates the high quotient into

AD if it is alright as it is (greater than zero), or negates the high part if the quotient is nonzero and should be negative, but has a null low part. Except for a zero quotient, the sequence uses the final two double normalize time states to handle the overflow flags and insert the exponent in the high quotient in the form appropriate to the sign of the fraction. The store cycle puts the two-word quotient in the accumulators, with the low word transfer to AR at ST4 forcing the sign to 0 – which means that the extra logic put in to do just that at the DFDT9 condition is unnecessary.

8.6.5 Double Moves

Chart DM shows the four double move instructions. The short sequence at the left handles the moves to AC. These use the standard double fetch, and they store the double memory operand or its negative in AC and AC2.

The longer sequence at the right handles the double moves to memory, which require two main sequences for the separate storage of the high and low words in E and E+1. The first execute cycle handles the high word, but for move negative it must inspect the low word to determine whether to complement or negate the high part. The second execute cycle then negates the low part, forcing its sign to 0.

CHAPTER 9
INPUT-OUTPUT

The input-output system for a KI10 processor includes the in-out bus, the peripheral equipment with its interfaces to the bus, and several sections of the processor logic. The processor elements are in-out transfer control (including timing, the bus control, and basic IO logic), the priority interrupt, and an interface for the processor that allows IO instructions to control the processor itself as a device. Also discussed here is the key function read in, which makes use of elements of both the key logic (section 5.6) and in-out control. In addition to the above processor equipment, this chapter describes the interfaces for the basic in-out equipment, namely reader, punch and teletypewriter. Note that the basic IO logic in the processor hardware is not the same thing as the basic in-out equipment described in Chapter 3 of the reference manual: the basic IO logic comprises the device selection and IO instructions for priority interrupt, processor device logic, and paging hardware as well as for the basic IO devices.

9.1 IN-OUT CONTROL

The control and timing for transfers over the IO bus are shown in the IOB and IOT drawings. Access to the bus, which is used by IO instructions and by PI requests, is governed by the three flip-flops at the top of print IOBC. The events associated with a PI request take place in parallel with and generally independent of the execution of instructions by the processor. A conflict occurs however when a PI request is made at the same time the processor performs an IO instruction. Each time a bus transfer is completed, the bus must be discharged, and during the discharge period IOB RESETTING SYNC remains set to prevent any further access to the bus. Once the bus is discharged, it is available to either a PI request or an IOT, and access for these is gained respectively by setting IOB PI or IOB IOT.

When an IO instruction gets the bus, IOB IOT(1) allows ET1 to trigger the special execution sequence of IOT time states defined by the flipflops at the top of print IOT1. Unlike other time states, these are not equivalent to the periods between clock pulses, but each instead comprises a number of clock periods as determined by the IOT timer, which is a standard counter utilizing a +1 gate. At the beginning of each IOT time state, the counter is loaded with the number of clock periods the state is to be held, and the transition to the next state occurs when the counter overflows. The number loaded depends on the time state, the transfer direction, and whether the bus is set up for fast or slow operation. The bus can operate at the fast speed if all the devices connected to it are designed for operation with the KI10; but connecting devices designed for the KA10 necessitates operating the bus at the slow speed. Timing diagrams for the two speeds for both directions are in the left half of FD drawing IOBT.

Other control circuits for IO instructions are on drawing IOT2. Within the special IOT sequence the flipflops at top center define the periods during which the device code from IR is placed on the device select lines and the data (from processor or device) is placed on the data lines. Of the flipflops at the right, the lower controls skipping in a noninterrupt BLKX, and the two together control the same function in read in. At the left is IOT INST, which controls the common events for all IOT sequences; this is asserted not only for the actual IO instructions but also for the DATAI and DATAO interrupt function. Similarly, the general control functions for data in and data out are produced by both the corresponding IR decoder outputs and the interrupt functions.

The net in the lower part of the drawing produces the levels that gate output data onto the bus; these are generated during data time but only for a function that calls for outgoing data. A similar net in the same position on drawing IOBC generates the levels that gate information in from the bus at the processor end, but these read levels are enabled throughout the data time regardless of direction; hence outgoing information can also be read at the processor end for use in the basic IO logic, which bypasses the bus cable. The gates are also enabled during PIT2 for reading the interrupt function word. The rest of drawing IOBC shows the remaining logic for the timing of bus signals. The one-shots in the upper right define the bus discharge period and the width of the IOB reset pulse that can be generated from the console or by the program (CONO APR, bit 19). Below these are the gates for broadcasting the number of the channel for which a PI request has been granted, and gates that define the bus clear and set pulses at time states 5 and 7 (the pulse limiter limits the width of the signal produce; in single pulse operation). At the lower right are the jumper connections for selecting between fast and slow bus operation (selection between 50 and 60 Hz line power is also shown here). In the left side of the drawing are the gates that generate the IO instruction pulses and levels for the bus from the basic clear and set pulses and the data time.

Drawings IOBL and IOBR show the processor IO control connections to the bus data lines. The enable for outgoing data places the complement of AR on the lines although physically the connections are made from the input mixer to fast memory. When the read enable is true, data on the bus is made available to AD and other processor logic through the gates shown above the bus cable. The input signals may reflect either the data from the bus or information from either of the basic IO mixers discussed below. Dashed lines stemming from each discharge input gate indicate the bus lines the gate controls. At the left in drawing IOB1 are the gates that supply the device code to the IOB select signals, which are used by the basic IO logic and also provide input to the drivers for the bus IOS lines; the device code is taken from the readin device switches during read in, but at all other times from IR. In the right half of the drawing are the bus connections for the various control signals and the inputs from the PI request lines. At the right in drawing BIO, the signals from the bus request lines are mixed with the PI request signals from the basic IO logic to produce the request signals used in the PI logic.

9.1.1 IO Instruction Flow

Flow chart IOT shows the events for the IO instructions and equivalent interrupt functions. These sequences share many common events, which are controlled by IOT INST rather than by the IR decoder outputs for the instructions. No IO instruction stores an accumulator, and in many PC can change, so PC CHANGE is generated to prevent an automatic instruction fetch. The flow path is controlled primarily by IOT INST, but the effect of this level on BLKX is limited by the fact that the main branch point in the execute cycle occurs in a time state that is not used by BLKX, which to avoid confusion is shown separately at the left. BLKX uses read-modify-write to handle the pointer, and the fetch cycle actions increment the pointer if the instrcution is in an interrupt cycle, or if BYF6 (First Part Done) is clear indicating the instruction has not previously been started and interrupted in the middle. IOT INST puts PC on the address bus for possible skipping and takes the IO bus if it is available. At ET2 the incremented pointer is returned to AR for later storage and the processor determines whether the block is complete. In a PI cycle, PI OV is set if the left half of the pointer has been counted down to zero. Otherwise IOT BLKX SKIP is set if the count is not zero, but this action is inhibited if a PI request is using the bus, as in that case the instruction will not go to completion. Except in a PI cycle, ET2 sets BYF6 to indicate that the first part of the instruction has been done. In the store cycle the incremented pointer is written back in memory. ST2 enables the pointer into AD for use as the address of the data word, and returns the processor to the fetch cycle to switch from the BLKX to the appropriate data instruction (the code in IR is converted by setting IR 12). For a BLKO, the return is to FT3 to read the output word; for a BLKI, MC MEM GO INH is set to hold up the memory cycle in case the processor must wait for the bus, and the return is to FT5 for a page check of the storage location. In either case the processor continues with the events for a data IO instruction.

The center section of the chart shows the actual sequences that use the bus. Most events for these are typical: input instructions do a page check of a storage location and inhibit the memory cycle; a data output instruction fetches the data word. At the right end are the special entries for data operations from a block IO or from the PI cycle for an interrupt function; the various paths all join for the fetch cycle actions, which are called by IOT INST and are the same as those mentioned above for a BLKX. At ET1, if the instruction does not already have the bus, it may go into a loop to wait until it is available. If a PI request already has the bus or is waiting when the bus is discharged, the processor goes to a PI cycle, although it may wait in the loop until the PI cycle is ready. If there is no PI diversion, the instruction eventually gets the bus, at which time it performs its ET1 actions and enters the special IOT sequence of time states. ET1 enables the complement of AR into AD (the bus gates complement the outgoing data), increments PC for a BLKX that requires a skip, duplicates the output conditions in AR left (the sign smear simply prevents AR right from being lost), and moves the mask for a CONSX to BR.

At IOTT0 the device code is placed on the select lines provided the sequence is for an actual programmed instruction decoded from IR, and for output the complement of the operand is placed in AR. The timer is loaded so that IOTT1 lasts for three clocks, and the completion of each subsequent time state loads the timer with the appropriate number for the next time state. IOTT1 begins the data time, and IOTT2 enables the bus data into AD. An input operation bypasses the next three time states, but they are used for output to generate the clear and set pulses for a CONO or DATAO. At IOTT8, the data time ends, the input data (if any) is moved from AD to AR, and if PI OV is clear in any instruction other than a CONSX, the processor begins the prefetch of the next instruction. IOTT9 disables the device code and starts the bus discharge. Clocks 8 and 9 both enable BR and AR into AD so that if the instruction is a CONSX, the expected conditions from BR are anded with the real conditions brought in from the bus to AR. The sequence returns to ET2 to begin the storage operations for the input instructions and to check the result in a condition skip: if the named condition is satisfied, PC is incremented, but in a PI cycle PI OV is set if the named condition is not satisfied.

9.1.2 Basic IO Logic

IO transfers for the basic in-out equipment and the processor elements controlled by IO instructions are handled by internal logic instead of using signals over the IO bus cable. The left two thirds of drawing BIO shows the device code selection for the processor device logic, priority interrupt, paging, reader, punch and teletypewriter. At the lower left is the jumper card for selecting the processor serial number. Input from the above named devices is handled through ordinary logic mixers, whose outputs are ored with the outputs of the bus receivers to produce the real IOB inputs. Drawings BIX1 and 2 show the A mixer that handles data input from the reader and paging hardware, and both data and condition input from the processor device logic. The B mixer on BIX3 handles conditions for priority interrupt and paging, and has larger mixer gates for bits 30–35 to handle conditions from the reader and punch, and both data and conditions from the teletypewriter.

9.2 PRIORITY INTERRUPT

The priority interrupt system is shown in those block schematics and flow charts whose codes begin with the letters PI. Print PIC1 shows the logic that controls the diversion to a PI cycle, the special time states for the cycle, and the various events in the cycle. On PIC2 are the gates through which IO instructions control the interrupt system, the logic that receives and decodes the interrupt function supplied by a device, and a flipflop that causes the MA special logic to generate the address of the interrupt location for a standard interrupt (which is done in a "normal" cycle). At the lower left is the flag through which the program turns the entire system on and off.

Drawings PIHR and PIOG show the flags that control the individual channels. By means of the upper set on PIOG, the program turns individual channels on and off; through the lower set the program can generate interrupt requests on individual channels (note that regardless of the way the PI system responds, a program-generated request remains until the program drops it). PIHR shows the flags that synchronize requests to the channels and hold interrupts on them. The PIR flags remain stable while a request is being processed, as indicated by the level PI RQ being true. When this level is false, the PIR flag for a channel is set if the program generates a request for the channel or a request comes in over the bus PI request line for the channel and the channel is on. Although the program and external devices can make requests on a number of channels simultaneously, and hence set a number of PIR flags together, the system processes a request for only one channel at a time. The selected request from among the PIR flags set is indicated by the single PI REQ level that is asserted from the priority net at the left in print PIN. It is through this net that the processor selects a channel for starting an interrupt. No selection can be made unless the interrupt system is active, as indicated by PI ACT being set. With the system active, the request level for channel 1 is asserted if the request flag for that channel has been set and no interrupt is currently being held on that channel; in other words PIR 1 is set and PIH 1 is clear. If PI REQ 1 is not asserted, then the second request level is asserted if the request flag for that channel is set and no interrupt is currently being held on it. The same conditions extend through the priority net: if any PIR flags are set and the system is active, the net generates a PI REQ level for the lowest numbered channel whose PIR flag is set provided the processor is not already holding an interrupt on the same channel or a channel with higher priority. Through the nets in the upper right the generation of any PI REQ level produces PI RQ and the number of the channel encoded in binary. When no request is being made and no interrupt is being held, the net at the lower right generates the bus signal PI OK 8, which is used by the real time clock to discount interrupt time while timing a user program.

The level PI RQ prevents the processor from accepting any further requests. It also takes the IO bus for the PI request sequence through which the processor determines which device to service and the type of interrupt to perform. This sequence and the PI cycle that results from it are discussed in detail below. In many cases the PI cycle may complete the interrupt and the processor returns to the interrupted program. However if the interrupt instruction is one that saves the flags (JSR, JSP, PUSHJ, MUUO), the net at B7 in print PIC2 generates PI RQ SETS PIH. This signal holds an interrupt by setting the PIH flag for the channel whose PI REQ level is asserted. Below the PIH flags are two sets of gates. Through the lower set, flags once on are held on until the interrupt is dismissed. The upper set receives input from the lower set and also receives the signals for setting a flag to hold an interrupt. Hence several PIH flags may be on simultaneously and the interrupt actually being held is that one corresponding to the lowest numbered flag that is on. Of course an interrupt routine can prevent further interruptions on higher priority channels by turning off the interrupt system; clearing PI SYS ON at the lower left in PIC2 deactivates the request logic by clearing PI ACT at the lower right in PIC1. When a routine is finished it must dismiss the interrupt so as to return to a routine on a lower priority channel or to the main program. Since the interrupt being held corresponds to the lowest numbered PIH flag that is on, dismissal is accomplished by clearing that flag through the lower gate. Besides feedback from the flag, this gate receives two inputs, the negation of PI DISMISS and the negation of a clear signal for the flag. To dismiss, the program must give a JRST with a 1 in bit 9, which generates PI DISMISS through the net at B4 in PIC2, and the priority logic must generate the clear signals from the left up to and including the first PIH flag that is set. PI ACT(1) is the clear signal for PIH 1 and is also the input to the priority logic in the center of drawing PIN. Hence if an interrupt routine deaxtivates the interrupt system, it must reactivate before it can dismiss the interrupt, and the dismissing instruction automatically clears PIH 1. If PIH 1 is clear, the priority logic on PIN generates PI 2 CLR, and if PIH 2 is clear it generates the clear for PIH 3, and so on up to and including the leftmost PIH flag that is set.

9.2.1 PI Request Sequence

The setting of a PIR flag for a channel that has priority results in the generation of PI RQ. This signal gains access to the IO bus by setting IOB PI at the upper left in drawing IOBC once the bus is free. The request logic must wait for the bus if it is already in use by an IOT or is discharging from some previous use. If both the interrupt and the IO logic request the bus simultaneously, the interrupt has priority. Setting IOB PI triggers the bus PIR sequence through which the processor determines which device requested the interrupt and what type of interrupt to perform. This sequence is shown in flow chart PIR, the logic for it is in block schematic PIR, and the timing of the signals over the bus is shown at the right in FD drawing IOBT. The enable input that holds IOB PI set is also applied to the gate at the center of print PIR to generate a signal that broadcasts the number of the channel on which the request has been accepted; this signal sends the number out to the devices over bus lines 0–2 through the gates at C3 in print IOBC. Figure 9-1 shows the various PIR signals on the bus and shows the split routing of the request grant. Figure 9-2 is a block diagram of the typical device logic that interacts with the PI request sequence; the order of events in the sequence is indicated by the circled numbers in the figure.

IOB PI triggers the sequence by allowing the timeout of the one-shot at the right in print PIR. Each timeout of this one-shot retriggers the circuit to produce an independent clock that times the bus events through the row of flipflops at the top of the drawing. The logic keeps track of the number of pulses on the ring counter made up of the three count flipflops at the right. The first clock sets the leftmost flipflop in the row to provide the request sync pulse over the bus; any devices that are requesting an interrupt on the channel whose number is being broadcast synchronize on this pulse. The next clock sets the second flipflop to send out the request grant signal, which goes from one device to the next along the bus and is stopped by the nearest device that has synchronized for the interrupt. Note that in terms of priority, all devices connected to the left bus cable are nearer to the processor than those connected to the right cable; as shown in print IOB1 the grant signal is sent out on the left cable, and the return from the left cable provides the signal sent out on the right cable. The device that has gotten the grant sends back a function word on the data lines, and the arrival of the function as indicated by a 1 on any of lines 3–5 synchronizes the return by causing the next clock to set the third flipflop (the clock also loads the function register at top left in print PIC2). If no function word comes in, PIR RETURN SYNC is set anyway when the grant return comes back from the right end of the bus or by the eighth clock if there is no return. In any event setting PIR RETURN SYNC clears PI REQ GRANT and sets PI READY. This last flipflop turns off the PIR clock and sets up PI control for a PI cycle by setting PI RDY SYNC at the upper right in drawing PIC1.

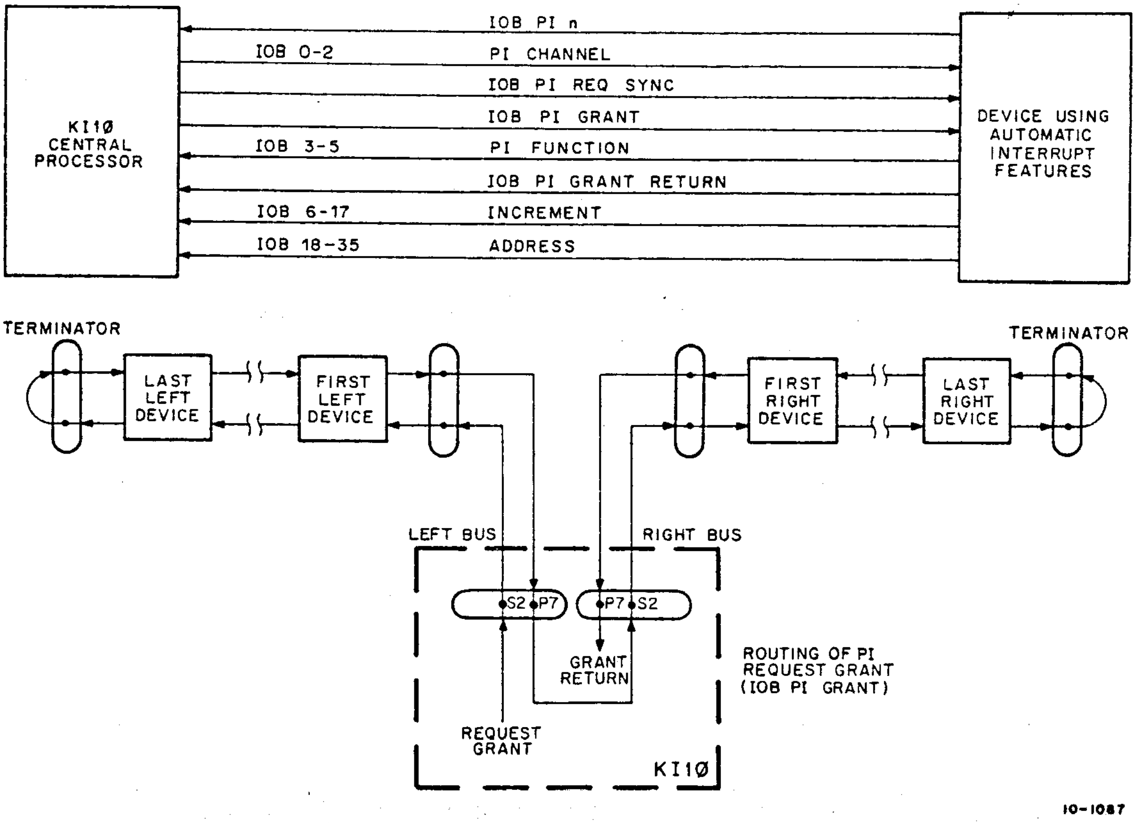


Figure 9-1 PIR Bus Signals

There is no further action in the PIR logic until the processor sets PI CYC, which causes the processor clock to set PIR CYC STARTED. This in turn enables PIR DONE but not until a minimum time has elapsed since PI READY was set as determined by the one-shot below the flipflop. PI REQ GRANT clears when PI READY sets, and the minimum time guarantees that the request grant signal on the bus dies out before another request cycle is started. PIR DONE clears all of the flipflops at the top, holds off the PIR clock, and as indicated on print IOBC, releases the bus by clearing IOB PI and starts the bus discharge. Note that during the PI cycle, IOB PI cannot again be set even though PI RQ may still be true after the bus is discharged.

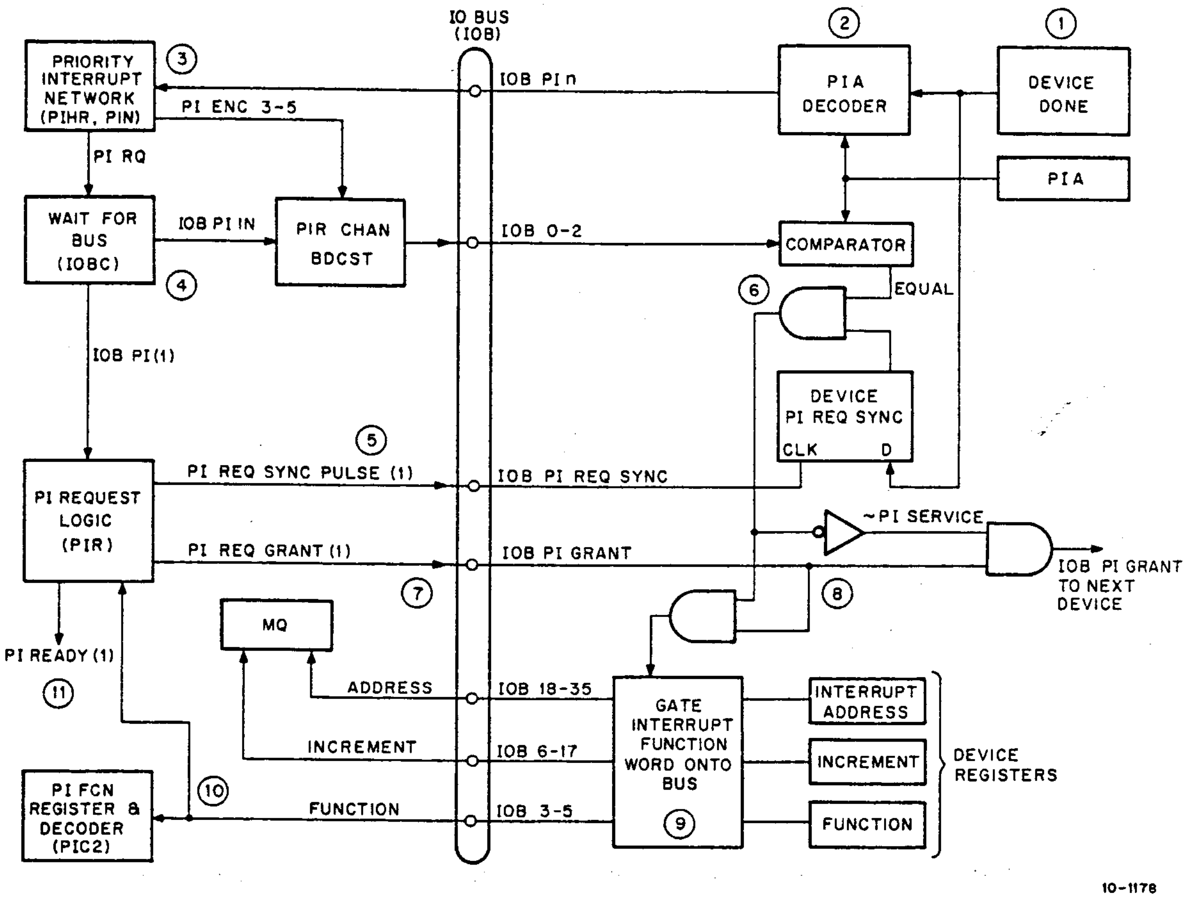


Figure 9-2 Basic PI Routing and Control

A very fast interrupt executed under optimum conditions can go to completion and allow the processor to recognize a new request before the present request cycle is terminated. When this occurs the second time state at the upper left in chart PIR sets IOB PI but fails to trigger the PIR clock because either PI READY or PIR DONE is 1. In this event the condition that inhibits the clock also satisfies the loop, so that repetitions of the second time state coincide with the third state; in other words the condition that enables IOB PI initially continues to be satisfied along with the condition that holds that flag on once it is set. PIR CYC STARTED being 0 prevents the third time state from setting PI RDY SYNC, and the termination of the cycle finally cancels out the loop at the third time state and clears IOB PI. With the bus discharging, the loop at the second time state continues until IOB PI can again be set.

9.2.2 PI Cycle

The two PIC drawings show the logic for the PI cycle and the sequence of events appears in chart PI. The sequence begins with the setting of PI RDY SYNC at the first main clock following the setting of PI READY. There are two methods of diversion of the normal processor sequence to a PI cycle. Following synchronization, PI DIVERT INDIRECT sets and this produces PI CYCLE DIVERTED if the processor must get another address word at IT1 in an effective address calculation, enters a trap cycle at IT2, or must wait for the bus in an IO instruction. In any of these cases, the processor simply drops the current instruction and goes directly to PIT1. However if an instruction fetch is called (automatic or otherwise) while PI RDY SYNC is 1, PI CYC RDY sets to prevent memory control from actually fetching the instruction. (The flag in the lower left corner simply holds the ready condition an extra time state for effecting the inhibit in memory control.) Then the completion of the current instruction generates PI CYCLE DIVERTED, and the clock that sets PIT1 clears the flipflops in memory control and advances PC to point to the next instruction, unless the instruction that would have been fetched was being executed by an XCT or LUUO. Note that all of this diversion logic is used also for entering a key cycle as explained in section 5.6.4.

PIT1 enables the function word into AD and the address part of it onto the address bus, kills any memory write subroutine that may be waiting from the interrupted instruction, clears PI RDY SYNC, and sets PI CYC which enables slow paging because the paging hardware may have to switch address spaces. PI CYC also enables the interrupt function decoder at the left in PIC2. The gate at right center generates a pseudo instruction fetch signal for a dispatch interrupt or any standard interrupt; the standard interrupts include the usual function 1, but also the unused functions 6 and 7 and function 0, which is the failure of any device to supply a function word (as would happen were the interrupt requested by a KA10 type device).

PIT2 sets PIR CYC STARTED, saves the function word in MQ, latches the address bus, and sets up memory control for the type of access needed. A standard interrupt sets PI NORM CYC to generate the address in the process table through the MA special logic. All other functions use the address supplied by the function word. Even though PIT2 calls the memory subroutine, an extra clock is produced to set PIR DONE, although this is not done if the ungrant timer has not timed out (see chart PIR). PIT3 therefore stops the clock and also reestablishes all of the necessary information for memory control.

For the special interrupt functions, the memory subroutine does an operand fetch or page check and the return is to FT6. DATAI and DATAO functions are almost identical to the equivalent IO instructions and are included in flow chart IOT. The increment function, shown at the right in chart PI, starts by moving the memory word to BR and the function word to AR. If the increment is positive, SCAD DATA is set to make the SC adder outputs all 0s (otherwise all are 1s). ET1 then uses the byte position logic to extend the sign to the left from AR 06 through the ARI special inputs. ET2 moves the half word increment from AR left to AR right, extends the sign through AR left by clearing unless the increment is negative in which case AR left is loaded from AD (whose outputs are all 1s when neither AD nor its input mixers are enabled), and enables AD to add the increment to the memory word. From ET2 the sequence goes to DMOVNT3 to put the result in AR and then continues to the store cycle.

For a standard or dispatch interrupt the memory subroutine does an instruction fetch with return to IT0 and the PI cycle executes the instruction. PI CYC being set prevents overflow trapping (by preventing the setting of any overflow flags) and also prevents the instruction from satisfying a trap. Instructions that can produce an interrupt skip generate the signal PI MAY OVF. If the instruction is one of these and does not skip, ET2 sets PI OV, which at the completion of the instruction causes the processor to go to PICT0 to start another PI cycle, instead of clearing PI CYC and returning control to PC. For a standard interrupt PI OV(1) simply increases the MA special address by 1, but for a dispatch interrupt PICT0 brings the function word from MQ to AR, increments the address through AD and enables it onto AB. PIT2 performs the same functions as in the first time around except that it does not set PIR CYC STARTED, and the only memory setup it can do is for a standard or dispatch interrupt. No extra clock is needed but PIT3 occurs anyway and stops the clock. The memory subroutine does an instruction fetch from the second interrupt location and the PI cycle executes that instruction.

At some point in a PI cycle, after it has been determined whether or not an interrupt will be held, PI ACT must be cleared temporarily to turn off PI RQ, and thus clear the PIR flag for the channel (unless of course there is already another request for the same channel). The temporary inactivation is handled by the active inhibit nets at the lower right and bottom center in PIC1. All inhibit conditions include PI OV(0) because the inhibit is in the first PI cycle unless there is overflow, in which case it must be put off until the second. In an IOT the inhibit is at the data time except in a CONSX, which tests for overflow following the transfer. In an instruction that can overflow but has not, the inhibit always occurs at ST1. In an instruction that cannot overflow, the inhibit is at ET2; but again PI OV(0) is a condition, because a DATAX can be produced by an overflowing BLKX, and can therefore produce a second cycle even though the DATAX itself cannot overflow.

9.3 READ IN

This function straddles two areas of the processor logic: part of it is included in the key logic discussed in section 5.6 and part in the IOB and IOT logic of section 9.1. The function itself is made up of several parts as shown at the left in flow chart KRMP.

Part 1 acts in some ways like both reset and start as well as setting up the various readin operations. KT1 generates the master reset to clear the machine and place it in kernel mode with paging disabled (there can be no panic clear however), sets KEY RUN so that the processor will go into normal program operation when the function is finished, and clears MB to produce a zero effective address and prevent IR from being affected by the MB contents. KT1 also sets a setup flag that places the contents of the readin device switches on the IOS lines of the bus, generates a readin pulse on the bus to place that device in readin operation, and sets up IR for the code of a DATAI instruction. KT2 sets a wait flag so that the key cycle loops at KT3 until the device indicates that the first word is ready. KT3 then generates both instruction ready and instruction done for entry to the instruction cycle to execute the DATAI that has been set up in IR. With MB clear, the DATAI brings in the word from the device and stores it in location 0. Within the instruction KEY RDI FF0 being set (from KT3) inhibits the usual fetch cycle transfer of PC to AB; and TRAP SATISFIED clears KEY CYCLE, enabling KEY FCN RDY so that the next clock sets KEY RDY SYNC. Hence at the completion of the instruction the processor returns to the key logic to continue the function.

In the second part (which is repeated for each data word) KT1 again clears MB for a zero effective address to reference the pointer stored in part 1 (and prevent its affecting IR). The setup flag remains on throughout the function, so the device code stays on the bus and the left three IR bits stay set. However KT2 sets KEY RDI FF1 to disable the first cycle level; hence IR now contains the code of a BLKI instead of a DATAI. A loop at KT3 waits till the next word is ready, at which time the processor goes to the instruction cycle to perform a BLKI to store the data word at the location specified by the right half of the incremented pointer. If the block is not complete, the instruction sets IOT BLKX SKIP (print IOT2, right), which in turn sets the hold flag above it. At the end of each BLKI the processor returns to the key logic, and the hold flag allows it to determine whether the block is complete even though the skip flag is always cleared at the end of the instruction.

Once the entire block has been transferred (indicated by failure to set the skip flag), the standard PC clock at the entry to the final key cycle loads the last data address into PC and KT1 moves it to AB. (As in any key cycle PC+1 INH is set, so AB gets the address of the last word, not the one after it.) KT2 produces instruction done in the usual manner, but instead of simulating instruction ready, it actually triggers an instruction fetch; the processor therefore fetches the last data word from the location in which it was stored, and then continues to the instruction cycle to perform it.

Although the flow chart separates the function into three parts, the logic is not actually so nicely ordered. In many cases an event may occur at every instance of a particular time state, but be relevant only the first time, the last time, or some other set of times; *eg* all of the finagling with PC at the instruction done entry to the key cycle and at KT1 is needed only to supply the final address for the instruction fetch (the PC skip clock in the DATAI is also redundant). Note that the condition for KEY DONE (which distinguishes the final part from repetitions of the second) requires not only that the skip flag has not been set but that KEY RDI FF1 has been; this prevents premature termination in the first iteration of part 2 when the hold flag is bound to be clear since no BLKI has yet been performed.

9.4 PROCESSOR DEVICE LOGIC

Drawings APR1 and 2 show the logic through which in-out instructions control the processor as though it were a device. Some of the logic on APR2 has already been discussed as part of console control (§5.6), namely the sense switches and Maintenance Mode flag at the top and the readin device switches at bottom center. The logic for the timer at the right in APR1 and the logic for speed margins and even parity writing at the right in APR2 are for maintenance programming and are discussed in §10.3; suffice it to say that in ordinary circumstances APR2 NORMAL SPEED is set and all of the other flipflops are clear. The IO instruction inputs for the processor device logic are at the left in APR2. Note that in the bottom net the DATAI enable levels are generated not only by DATAI APR, but also by the first two key time states; this is to make the data switches available to AR via the IO bus for the execute and deposit functions.

In addition to the timer, drawing APR1 shows the clock and error flag systems: at the top are the flags, in the middle are the PI assignments and enabling flags, and at the bottom are the decoders for requesting interrupts. The circuit at the lower right uses a signal from the 845 power control to generate the line frequency clock, which sets the Clock flag provided the processor is not in single instruction operation. Each time the flag is set, it enables the decoder at the lower left to request an interrupt on the channel specified by the clock PI assignment provided Clock Enable is set. The other three flags are for error conditions: In-out Page Failure is set by a signal from the page fail cycle, Nonexistent Memory by a signal from memory control, and Power Failure by a low voltage signal from the 858 power control (which signal is available to the program as Power Low, CONI APR, bit 9). The setting of any error flag enables the decoder at the right to request an interrupt on the channel specified by the error PI assignment. This request is also made by setting Parity Error (§6.1) if the program has set the enabling flag, which is in the upper right corner in APR2.

At the center of APR2 is the logic that controls the auto restart feature. The program enables and disables the restart by setting respectively the restart set and clear one-shots, which provide pulses long enough to pick and drop the mechanical restart relay in the 857 power control. Setting Power Failure disables the restart, and to enable it the program must clear the flag and trigger the restart set one-shot. If both one-shots are on together, the period of the set does not begin until the clear times out.

9.5 BASIC IO EQUIPMENT

The interfaces for reader, punch and teletypewriter are included right in the processor logic, in the lower part of bay 2. A good part of the logic for each interface is simply the various elements discussed in the reference manual, namely buffers, the various flags, and the PIA bits with their decoder. Remember that these interfaces are not actually connected to the bus cable: control signals and data are handled directly by the processor IOB logic (section 9.1), and device selection and interrupt signals are handled by the BIO logic (section 9.1.2).

9.5.1 Paper Tape Reader

The reader logic is shown in three PTR block schematics and in flow chart PTR. The actual control logic is almost entirely in print PTR2, including a ring counter (upper left) for counting characters and a Gray-code counter (D3-D4) to distinguish the data phase from the space between characters (the clock runs at twice the character rate). Print PTR1 shows the flags (except Tape, which is on PTR2), PIA logic, IO signals, and a circuit at C6 that produces an end pulse when the last character is strobed – the last character being the first in alphanumeric mode, otherwise the sixth. The 36-bit buffer (PTR3) is a six-stage shift register whose bottom row receives the 6-bit characters from tape. But note that the direct shift connections are broken at the inputs to bits 28 and 29; for these the nets at the left supply the shift for binary, but signals from holes 7 and 8 for alphanumeric.

Setting BUSY or pressing the feed switch produces RUN, which in turn sets ENABLE to enable the clock. Although the clock runs at 600 Hz to produce the motor shift, the first period after a layoff is extended to allow for tape start time. The first motor shift sets POWER to turn on the reader and places the logic in its first data phase. The subsequent appearance of a feed hole produces a pulse that triggers the strobe to read the character into the buffer if either hole 8 is punched or reading is in alphanumeric. Each subsequent pair of motor shifts handles one frame. The strobe for the final character produces the end condition, which sets DONE and clears BUSY. The latter action drops RUN, so that if the program does not set BUSY again (by retrieving the data) within 1.6 ms, the next clock clears ENABLE to disable the clock and start the 40 ms shutdown delay, which holds start-stop operation down to 21 alphanumeric characters or 15-1/2 binary reads per second. If the reader has not been called again before the delay times out, a shutdown pulse turns off reader power. Note that setting TAPE by pushing the feed switch (*ie* while BUSY is clear) sets DONE for an interrupt.

9.5.2 Paper Tape Punch

BS and FD drawings PTP show the punch logic. The punch motor is controlled by the SCR delay (A5 and A7). While the motor is on the punch can be used immediately, but if it is not called for five seconds, the SCR delay times out; subsequently calling the punch starts the motor, but the speed delay prevents any action by the logic for one second while the motor gets up to speed. At power turnon the SCR delay comes on in the 1 state, but the 0 state of MOTOR ALLOW (A1) prevents it from turning on the motor; the first timeout of the delay sets the flipflop, so subsequent control of the motor is exercised solely by the delay.

In operation a sync signal from the punch triggers the solenoid driver delay if BUSY is set or the operator is pressing the tape feed switch. Punching the buffer contents into the tape takes 10 ms, following which the timeout of the delay signals that punching is complete. The program then has 10 ms within which to respond before the next sync starts the next cycle; after this time punching is delayed until yet another sync arrives. Note that holding down the tape feed switch while the punch is not busy holds on the buffer clear signal, allowing the operator to punch blank tape.

9.5.3 Teletypewriter

Among the several TTY prints, TTY3 contains only the switches that select between EIA and dc operation, and select the operating frequency for input and output by choosing among the clocks supplied by the logic in print TTY2. The output clock is twice the bit rate, but the input clock is eight times the bit rate to allow sampling centered within the input bit time. The TTY flipflops at the upper left in TTY2 provide straightforward frequency division by 2, 4 and 8 from the basic 19.2 KHz clock. The other two sets of four flipflops provide further division, where the TTO set runs on a clock with one-fourth the frequency of that used by the TTI set. In each set the flipflops from the right provide division in standard counter fashion by 2, 4 and 8. Bit 0 is set up in such a way that in the standard 8-count it supplies a clock that is equivalent to bit 1 only in the ratio six on/two off, whereas for 110 baud operation it provides a clock that is six on/five off in an eleven count.

The only remaining logic common to both input and output are the PIA bits and the Test flag at the upper right in print TTY1. The data input is represented by the signal ~TTI MARK, which comes from the input line but which can be driven from the output line by setting TTY TEST.

Input. Print TTY1 shows the input logic and chart TTIF shows the sequence of events associated with the reception of a character TTI FLAG is Input Done and TTI ACTIVE is Input Busy. The basic elements of the logic are an 8-bit shift register for receiving the serial character and a 4-bit ring counter for counting of the 8X input clock. Appearance of a space on an idle line sets ACTIVE, which enables the counter and generates an active pulse that sets the spike detector, clears the control flipflops at the upper left, and loads all 1s into the shift register. At the count of 4 (*ie* halfway through the unit period) the setting of the leftmost counter bit produces a sample clock that clears ACTIVE if the line is marking again; in other words a space that lasts less than half a unit is regarded as a transient and is not allowed to place the receiver in operation. The sample clock also shifts the data register right and loads the input into TTI 8, which therefore receives a 0 if the input is still a space, indicating the true beginning of a character. The first sample clock also clears the spike detector so that subsequent clocks, centered in the unit times, keep ACTIVE on while shifting the character into the buffer from the left. When the 0 from the initial space reaches TTI 1, the next sample clock completes the 8-bit data character and sets both FLAG and LAST UNIT. In the next unit time the third clock (from C setting) sets STOP, so that the sample clock has no effect on the shift register but clears ACTIVE.

Output. The output logic is in print TTO but BUSY and FLAG (Done) are at top center in print TTY1; chart TTOF shows the transmission sequence. Although the input logic can effectively forget about a given character once the data bits are received, the output logic must construct the entire character of eleven units for 110 baud operation, otherwise ten units. DATAO TTY, clears FLAG, sets BUSY, and loads the character into the output shift register, always setting STOP 1 but also setting STOP 2 if 110 baud is selected. With STOP 1 set the next 2X output clock sets DATAO SYNC, which allows the second clock to set ACTIVE; this action in turn generates the signal ACTIVE PULSE, which clears LINE to put the initial space on the output line. ACTIVE(1) reverses the input gating to the remaining two control flipflops so that in the next pair of pulses, the first clears SHIFT and sets ACTIVE LATE to disable ACTIVE PULSE, and the second sets SHIFT to shift the character right, placing the first data bit on the line. Each pair of subsequent clocks alternately clears and sets SHIFT to put another unit on the line. As soon as STOP 1 clears, the DATAO SYNC input to ACTIVE is disabled, and the shift that puts the last stop unit into TTO 1 gives rise to EMPTY. Since the first clock in the next pair again clears SHIFT, the second clears ACTIVE at the same time that it loads the last unit into LINE. However clearing ACTIVE holds LINE set to keep the line marking and also generates a done pulse to clear BUSY and set FLAG. To keep the transmitter going at maximum rate the program must give the DATAO within half a unit time so that the same pair of clocks that corresponds to the last stop unit on the line also synchronizes the instruction and sets ACTIVE for the next character.

CHAPTER 10
MAINTENANCE

This chapter provides some maintenance information for the processor and the interfaces for the basic IO equipment, principally at the system logic level. Since neither the processor nor its memories can operate in isolation, the information herein must be used in conjunction with the information in the maintenance manuals for the memories. Lubrication, adjustments and corrective maintenance for the basic IO devices are treated in the following:

Maintenance Manual DEC-00-PC0A-D(1): PC04/PC05 Paper-Tape Reader/Punch (Feed Hole Strobed Models)

NOTE
The information given in the PC04 manual is applicable to the punch without reservation and is applicable to the reader provided ECO KI10-0033 has been installed. For use in a KI10 that lacks this ECO, the PC04 reader must be adjusted so that it stops between frames. (The stop position of the tape must *not* be as indicated in the PC04 manual in this case.)

Teletype Bulletin 281B: Technical Manual, Model 35 Keyboard Send-Receive (KSR) and Receive-Only (RO) Teletypewriter Sets, Vols. 1 and 2 (Vol. 2 is just adjustments)

Teletype Bulletin 1201B: 35 Send-Receive and Receive-Only Sets (KSR and RO) – Parts

Maintenance procedures require the typical tools and test equipment that maintenance personnel are always expected to have, as well as some special items that are supplied with the machine. In particular, adjusting the delays in the processor requires the use of an oscilloscope with the following characteristics:

Vertical amplifier rise time t_r	$\leqslant 5$ ns
Horizontal time base	
Resolution (scale)	$\leqslant 2$ ns/division
Accuracy (tolerance)	$\pm \leqslant .1$ ns/division, $\pm \leqslant 1$ ns full-scale

Oscilloscopes that meet this requirement include the Tektronix portable types 454, 454A, 465A and 475A, and laboratory types in the 7700 series (some scopes have a magnifier feature that can be used to obtain the required performance).

CAUTION
Make sure to put the scope ground clips on ground pins only! Doing otherwise will generally cause the failure of one or more integrated circuits.

10.1 CUSTOMER PRINT SET

Although this manual is hopefully of some use, the real bible for maintenance personnel is the *KI10 Customer Print Set*. Before attempting to deal with the innards of the equipment the reader should be familiar with the contents of this set. The first drawing of twenty sheets is a directory of all KI10 drawings. Sheets 1 and 2 are indexes of the drawings in the Customer Print Set and the Manufacturing Print Set respectively. Sheets 3–6 together are a directory of all KI10 subassemblies, which are represented by boxes with associated find numbers. Each find number refers to a section of the detailed index comprising sheets 7–20; each section lists the individual drawings for the corresponding subassembly, although sometimes reference is made to additional drawing lists. For each drawing the index provides a description, the current revision, and an indication of whether the drawing can be found in the customer or manufacturing print set.

Following the directory are the flow diagrams and block schematics each in alphanumeric order, followed by the M142 circuit schematic (CS). Then come the module utilization drawings (MU), which show the layout of the logic mounting panels, identifying the module type in each slot and indicating the role the module plays in the processor logic. The next two pages are a parts list (PL) for the logic modules giving the quantity of each used and the recommended number of spares the customer should have at the site.

Next are the interconnection drawings (IC) showing the power wiring and panel connections. Bays 1 and 2 each have a PS and an SPE drawing: the PS drawing lists the power supply terminals in the bay, giving their functions and destinations, *ie* the drawing indicates the terminal to which the other end of each wire is connected; the SPE drawing gives the same information for the series pass elements. For bay 3 there are two PS drawings giving the same type of information for the power supplies, power controls, maintenance panel, and also for the thermistors and other power type connections in the logic in bays 1 and 2. Print AC identifies the power equipment and shows the ac wiring to the controls, supplies, panels and fans. Drawing DET lists the various dc connectors and has a schematic showing the left margin check cable and the connections to the 857 power control from various trouble circuits – circuit breakers, air flow switches, cooling assembly switches. The remaining IC drawings show the connections to the switches and indicators on the console and the bay indicator panels. At this point we should note the existence of certain other drawings that present information related to that shown in these IC drawings. Block schematic DCP presents a cross section between the power supplies and a pair of series pass elements, with their associated G830 regulating amplifiers in the adjacent slots, showing both physical connections and logical function. Thermistor locations are given in a table on BS drawing MR2. Also related are the drawings for the various power controls included in the set of DECsystem-10 Replacement Schematics. Note that all of the IC drawings are for control panels or power equipment; a complete listing of all logic connections between bays 1 and 2 is given in BS drawings IBC1 and 2.

The unit assembly drawings (UA) show the top assembly of the processor, *ie* these drawings identify all parts of the equipment (including cables) that can be seen simply by opening the doors. Also included are a number of structural sketches showing the way in which various items are mounted. The callout numbers reference the items listed in an associated parts list that follows (for cables the numbers also refer to items in a wire table on UA sheet 5). After the parts list is a component drawing (CP) that identifies the external components, *ie* the terminators (resistors, capacitors) that are mounted directly on the logic wiring on the front of bays 1 and 2.

The next page is a WL drawing but is really only a face sheet for the wire list, indicating its current state and incorporating it in the print set by reference. The actual wire list is the output listing of the WRP NAMESORT and PINSORT programs.

Then comes the complete print set for the PC04 reader-punch, generally including all drawings for the equipment as used with various DEC computers. The most important vis-a-vis the PDP-10 are the block schematics PC04-CL-PNCH and -RD.

Next are two parts lists. The first specifies the various accessories that come with a KI10, including manuals, print sets, and various mechanical items. The second lists all components other than logic modules, giving the quantity of each used and the recommended number of spares in two categories: the column labeled "site spares" indicates what should be kept on hand in ordinary circumstances, where garden-variety components are readily available from normal sources; the "customer spares" are the numbers that should be kept at a site in a remote area or in circumstances where ordinary components are not readily available.

The remaining drawings are all specifications (SP), of which the first three (each with a multitude of sheets) provide information for adjusting the delays in the processor logic and are discussed in setion 10.4. Drawing CPCG gives examples of the various configurations in which carry propagate and carry generate functions are produced in an adder module. The 22 sheets of drawing SPC list the logical configurations that should be encountered when the various machine operations are single pulsed. The final SP drawing gives complete instructions for installing engineering changes and doing any other required wiring or rework in the computer logic.

10.2 MAINTENANCE OPERATION

Line power enters the system at the 845 power control on the side of the console bay; this unit has the main circuit breakers (the lights are on the line side of the breakers) and a REMOTE/LOCAL switch that must always be in LOCAL. Although ac power is distributed throughout the processor directly from the 845, the console power switch (as well as many of the trouble detectors) is connected to the 857 power control, which in turn switches the 845. Hence actual control over system power is exercised by the 857 and 858, whose switches and indicators are available on the outside at the bottom of the bay 3 mounting door.

The only switch on the 858 (the upper unit) is a processor bypass that allows the operator to leave power up in the rest of the system – specifically those parts of the system connected to the DEC standard (857) power control bus – even when processor power is off. The 857 has a circuit breaker (with a light on the 845 side) and two toggle switches. (Both the 857 and 858 have screw selectors for the site line voltage, but these are preset and should never be touched.) The REMOTE/OFF/LOCAL switch can be used to turn off power during servicing, but is otherwise left in LOCAL except in the unusual circumstances that power for the processor is to be controlled from some remote point. The four lights at the left duplicate the light in the console power switch and the three trouble indicators on the maintenance panel. The detectors for an overtemperature condition are thermistors mounted in the logic panels and air flow switches mounted at the top and bottom of the regulator stacks; the opening of a cooling assembly door is detected by a microswitch in the door; the circuit breakers monitored by the third trouble indicator are those on the series pass elements at the ends of the logic mounting panels (see below). Whenever any trouble light goes on, power automatically shuts down, but the operator can use the upper toggle at the right to override the trouble condition and run the processor even with a trouble light on, in which case the power-switch-on light blinks.

CAUTION

Never leave the processor unattended while it is running with the override switch on. With the 857 in the override condition the KI10 cannot shut itself down in the event of a power short, excessive heat, or other trouble. Note however that the override switch does not override a low voltage shutdown.

Also accessible from the outside is the wiring for the modules inside the 857 and 858. The modules themselves can be reached by opening the mounting door and opening the face plates on the bay side of the controls. The plates are held closed by screw latches.

WARNING

The exposed terminals on the rear of the K614 module and the other internal wiring of the 857 and 858 are at line voltage and are therefore dangerous even when the KI10 is off. To deenergize all circuits in the 857 and 858, turn off the 845 circuit breaker.

On the bay side of the 857 are four convenience outlets of which the unswitched ones are really switched by the 845 and 857 circuit breakers, and the switched ones are dead.

At the outside end of the wiring side of each pair of logic rows is a small panel containing a series pass element and a circuit breaker for the +5 volts for each of the rows (a light at the other end of each row indicates when +5 is on). On the panel is a toggle switch for selecting between fixed and margin -15 volts: on some processors the switch has three positions for selecting among margin voltage for the upper row only or the lower row only, or fixed voltages for both; on other processors the toggle has only two positions for selecting between fixed and margin voltage for both rows at once. (There may also be another switch for selecting margin voltage for the +5, but this is operative only when the panel is used in peripheral equipment – switches at the logic are never used for margining +5 in the processor.) The panel for rows 2P and 2R is also for the two half rows of logic beside the bus sockets at the bottom, *ie* insofar as margins and voltage regulation are concerned, rows 3A and 3B are extensions of rows 2P and 2R respectively.

10.2.1 Margin Check System

Two buses carry margin voltages. The +5 margin bus, both of whose cables originate at the maintenance panel, carries margin voltage only for the +5 in the peripheral equipment; +5 margining within the processor is done by the program or from the console separately from the bus. The margin check bus carries both the +10 and -15 margin voltages and only the right cable originates at the maintenance panel; for margining these voltages, the first device at the left is the processor, and the margin check cable to left peripherals originates at socket J1 at the bottom inside of the left end panel (on the right looking into bay 1 from the rear; the bus signals are supplied via socket J2 from the inside cable).

The margin check supplies are the three at the top of the bay 3 mounting door; from the top these are an H704, a 783C and an H732. The last supplies a variable voltage in the range 0–20 volts, controlled by a variable transformer, for manual margins. Whenever a variable voltage is placed on one of the lines in the margin check bus the other margin line is held fixed; the 783C supplies the fixed +10 and -15 for this purpose. The H704 supplies a very accurate +15 and -15 volts (± .1%) to the D-A converter in the programmable margin system. This supply has an ac fuse for the margin system, and screwdriver adjustments for its two voltage outputs are located on the chassis that protrudes from the panel.

Basic operator control over the margin system is exercised from the maintenance panel. Turning on the margin enable switch allows the program to write even parity in memory and check speed margins (section 10.3), and enables the running of voltage margins in the manner determined by the margin selector at the right of the meter (the center section of the maintenance panel is sometimes referred to in the prints as the margin panel). The voltage knob at the left of the meter is connected to a variable transformer that controls the output of the H732 variable supply. The meter measures the actual voltage selected for manual margins (when the selector is off, it measures the output of the H732 directly). With MARGIN ENABLE on and the selector set to any position except PROC MAN, the program can run +5 margins on the processor logic (program margin levels never appear on the meter). With the selector set to a +10 or -15 position, the H732 output at the appropriate polarity is applied to the corresponding line on the margin check bus (with fixed voltage on the other line), and the meter (with range 0–20 volts) measures the margin voltage on the selected bus cable. The actual application of margin voltage to logic in any piece of equipment depends upon switch settings at the logic; in particular with the selector at -15L, the meter measures the voltage on the left -15 margin line, and the operator can apply the margin voltage to processor logic panels as selected by the toggles on the series pass element panels. A setting of +5L or +5R applies positive H732 output (attenuated) to the indicated side of the +5 margin bus and connects the meter (with range 4.5–5.5 volts) to that side. Finally the PROC MAN position supplies positive H732 output to the single logic row selected by the manual margin address switches at the left and connects the meter (4.5–5.5 volt range) to the monitor return from the same row. Margin addresses 00–17 select rows 1A–1T and addresses 20–37 select rows 2A–2T.

10.3 MAINTENANCE LOGIC

The processor contains logic that allows the program to execute various maintenance procedures as well as logic that implements the typical maintenance actions taken at the console. Many minor elements of the latter have been discussed in preceding chapters, such as the logic associated with switches that disable overlapping of memory cycles or force selection of an FM block specified from the console. Here however we are concerned with such logic on a larger scale, in particular the margin system and certain elements of the processor device logic.

The timer shown at the right in print APR1 enables the program to ensure that the machine will not stay down for a significant period should it go down because of hardware or software malfunction, such as logical errors that may result by running on margins. The program enables and disables the timer by means of a CONO APR,. With the timer enabled, the program must give a CONO APR, with a 1 in bit 18 at least every 1.2 seconds; failure to do so allows a time out, setting APR1 TIMER FF which produces an auto restart by setting MR RESTART B.

At the right in print APR2 are the flipflops that control the speed margins and even parity writing. With the margin enable switch on, the program can give a DATAO APR, that sets APR2 WRITE EVEN PAR; this inverts the output of the parity net when parity is being generated for a word to be written. The program can also set APR2 SPEED MARGIN, which with the margin enable switch on causes the clock to set APR2 HI SPEED. This flag in turn clears APR2 NORMAL SPEED and reduces every clock period 10 ns by substituting a 100 ns delay for the standard base clock delay of 110 ns. If either the operator turns off the switch or the program clears the speed margin flag, the speed flags reverse state and the clock returns to normal operation.

The logic that controls the margin system is shown in the three MAR drawings. MAR1 shows the inputs from the margin switches and the logic that allows the program (with a DATAO APR,) to control MAR PROG EN, supply a margin value to the D-A converter, and substitute a margin address for that specified by the switches. Note that although MARGIN ENABLE appears at first glance to be connected in the same fashion as the margin address flipflops, it is actually quite different and cannot be affected by the program – it is controlled only by the switch. The conjunction (C8) of the margin enable switch being on and either margins enabled by the program or manual +5 processor margins enabled by the margin selector generates MARGIN EN. This signal enables a slow-down capacitor in the D-A converter, making its output suitable for running margins, and switches off the D-A output from the comparator at the top of the print.

MARGIN EN enables the multiplexer in print MAR3 so that the margin address, whether supplied by the program or the operator, enables a single switch in the upper half of the multiplexer to supply the D-A or H732 output (as MAR VOLTAGE) to the specified logic row. It also enables the corresponding switch in the lower half to apply the monitor return from the same row to MONITOR VOLTAGE. This signal is compared with a +5 volt reference at the top in drawing MAR1 to provide a signal whose inverse can be tested as Monitor Low (CONI APR, bit 10). Hence the program can verify that the +5V regulators are varying as commanded by the D-A converter.

Print MAR2 shows the two margin buses and the margin selector. The margin check bus is readily recognizable in section C of the drawing, but a closer look reveals that the +5 margin bus is represented just below it in C4 and C5. At the upper left is the H732 power supply and the variable transformer controlled by the panel voltage knob. The rest of the drawing shows how the many decks of the selector set up the appropriate signals for the different margin situations.

Note that if MARGIN EN is false, the slow-down capacitor is cut out but the program can still load the D-A converter, whose output is switched into the comparator in place of the monitor voltage, so the program can calibrate it. Moreover the D-A output is available at pin 2S02V2 for external use.

10.4 ADJUSTMENTS

In both the block schematics and the MUs, various modules are tagged with numbers in circles or squares. These numbers refer to the delay adjustment procedures given in the delay chart, SP drawing DLY. Whenever a tagged module is replaced, the correspondingly numbered delay procedure or procedures in drawing DLY must be performed. A circle indicates that the replacement module itself must be adjusted, whereas a square indicates that when the tagged module is replaced the procedure must be carried out on some other module. The numbers actually refer to items in all three of the SP drawings related to the delays. Print DLY gives the procedure and indicates the logic signal being adjusted, the block schematic on which it appears, the module type and location, and the delay period. Drawing DPRO specifies the setup for performing the adjustment, being often a short program loop or switch repetition that will produce the necessary scope traces for measuring the delay time. Finally for many of the procedures, print DPIX shows what the scope trace should look like.

CAUTION

All probes used in making delay adjustments should have ground clips on them and should be grounded! Make sure to put the ground clips on ground pins only! Doing otherwise would generally cause the failure of one or more integrated circuits.

For making delay adjustments, probes must be identical, *ie* they must be of the same type and have the same length cables. Check the probes by selecting a convenient pulse and placing both probes on the same point. The two scope traces must appear in exactly the same position in terms of time. If they do not, get better probes or adjust the scope to compensate for any time difference, if the scope is equipped for such adjustment.

Remaining adjustments are of power supply voltages, which should be measured with a digital voltmeter. Voltage requirements, both fixed and adjustable, are as follows.

Supply	*Tolerance*	*Test Points*	*Block Schematic*	*Remarks*
Fixed -15 V	+.5–2.5 V	Bay 1 all rows, 01B2 Bay 2 all rows, 44B2	DCP D1	
Fixed +15 V	+2.5–.5 V	1D05H2, 2D43H2 2F31N2	MAR3 TTY3 B4	Margin multiplexer – print does not actually show test points
Regulated +5 V (SPE and G830)	±50 mV	Bay 1 all rows, 15A2 Bay 2 all rows, 30A2	DCP C1	Test at 1/3 of way across row from G830
H704 D-A Converter supplies				
-15 V	±.1%	2S02U1	MAR1 C3	Pot positions vary – just try one and see
+15 V	±.1%	2S02S1	MAR1 D3	
G831 reference supplies (2HJ43)				
+5.06 V		2H43U1	MAR1 D6	
-10.10 V		2J43U1	MAR1 C6	

10.5 DIAGNOSTIC PROGRAMS

DECsystem-10 diagnostic programs are identified by MAINDEC numbers of the form

MAINDEC-10-D*abcd*

where *abcd* are letters that represent the following.

a Indicates the type of processor on which the program will run.

A	KA10
B	KI10
C	KI10 or KA10
Y	PDP-6
X	Some other processor (not KI10, KA10 or PDP-6)

bc Indicates the type of equipment tested, such as

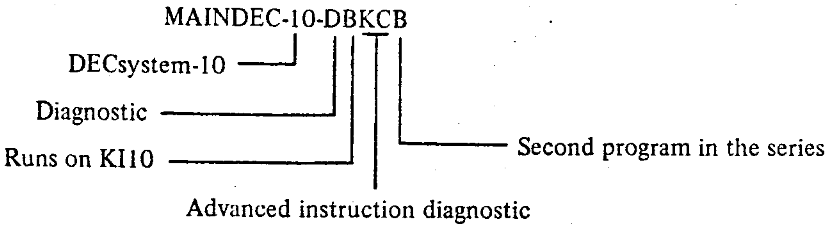
K*x*	Central processor
TM	Magnetic tape
TD	DECtape
MM	Memory
CR	Card reader
Z*x*	Diagnostic CUSP

Central processor diagnostics are further divided by test type.

KA	Basic instruction diagnostic
KB	Basic instruction reliability test
KC	Advanced instruction diagnostic
KD	Advanced instruction reliability test
KE	Memory control – relocation, paging, user mode, etc
KF	Special – console, timing, power supplies, etc

d Simply distinguishes one program in a given series from another (A, B, . . .).

Example:



When a single program requires more than one file, the individual files are specified by adding another letter to the MAINDEC number.

NOTE

Many KA10 programs have been revised for the KI10 and assigned appropriate numbers as per above. But old KA10 IO and memory tests, such as D2CC, D50B, D1KB, etc, can *not* be expected to run on the KI10.

The rest of this section lists the KI10 diagnostic programs, including maintenance diagnostics and reliability tests. All diagnostics are available on DECtape CUSP tapes customer utility service program tapes). Unless otherwise indicated, programs will run in any of these modes of operation:

E	executive
U	user
DM	diagnostic monitor
SU	special user (DKISPL)
SX	system exerciser

MAINDEC-10-	Program	Items Tested

KI10 Basic Instruction Diagnostics, DECtape CUSP DBZAA

DBKAA	Basic instruction diagnostic 1	Moves, Skips, Compares, Boole
DBKAB	Basic instruction diagnostic 2	Add/Sub, Compares, Moves, Boole
DBKAC	Basic instruction diagnostic 3	Logical test, Half word, Adder
DBKAD	Basic instruction diagnostic 4	ACs and Index, PC Change, Flags, Modes, XCT, Indirect
DBKAE	Basic instruction diagnostic 5	PC change, PWT, Add/Sub, Compares
DBKAF	Basic instruction diagnostic 6	Boole, Half word, Test,
DBKAG	Basic instruction diagnostic 7	PUSHX, POPX, XCT, Basic Shift/Rotate
DBKAH	Basic instruction diagnostic 8 (E or DM)	PI, Interrupts, LUUO, IO
DBKAI	Basic instruction diagnostic 9	Shift/Rotate, part 1
DBKAJ	Basic instruction diagnostic 10	Shift/Rotate, part 2
DBKAK	Basic instruction diagnostic 11	Fixed-point Mul/Div, part 1
DBKAL	Basic instruction diagnostic 12	Fixed-point Mul/Div, part 2
DBKAM	Basic instruction diagnostic 13	BLT, Byte, JFFO, etc

KI10 Basic Instruction Reliability, DECtape CUSP DBZBA

DBKBA	Basic instruction reliability test 1	Compares, Skips, EXCH, Boole, Rotate
DBKBB	Basic instruction reliability test 2	Test, Skips, Jumps, Compares
DBKBC	Basic instruction reliability test 3	Test, Half word, Add/Sub, JFFO, etc
DBKBD	Basic instruction reliability test 4	Memory and both modes, PC-sensitive instructions

KI10 Advanced Instruction Diagnostics, DECtape CUSP DBZCA

DBKCA	Advanced instruction diagnostic 1	Floating-point FSC and Add/Sub
DBKCB	Advanced instruction diagnostic 2	Floating-point Mul/Div, Negate
DBKCC	Advanced instruction diagnostic 3	FIX, FIXR, FLTR, Double Moves
DBKCD	Advanced instruction diagnostic 4	Double floating-point hardware
DBKCE	Advanced instruction diagnostic 5	Double floating-point DFAD, DFSB, DFMP, DFDV

KI10 Advanced Instruction Reliability DECtape CUSP DBZDA

DBKDA	Arithmetic reliability test	
DBKDB	Random instruction reliability test	
DBKDC	Interrupt reliability test (E or DM)	

MAINDEC-10-	Program	Items Tested
	KI10 Memory Control, DECtape CUSP DBZEA	
DBKEA	Paging hardware diagnostic (E or DM)	
DBKEB	Monitor UUO and user mode diagnostic (E or DM)	
DBKEC	Paged execute diagnostic (E or DM)	
	KI10 Special, DECtape CUSP DBZFA	
DBKFA	Console function diagnostic (E or DM)	
DBKFB	Instruction timing test (E or DM)	
DBKFC	Power supply calibration test (E or DM)	

APPENDIX A
INSTRUCTION AND DEVICE MNEMONICS

The illustration on the next page shows the derivation of the instruction mnemonics. The two tables following it list all instruction mnemonics and their octal codes both numerically and alphabetically. When two mnemonics are given for the same octal code, the first is the preferred form, but the assembler does recognize the second. For completeness, the table includes the MUUOs (indicated by an asterisk) that are recognized by Macro for communication with the DECsystem-10 Time Sharing Monitor. A double dagger (‡) indicates a KI10 instruction code that is unassigned in the KA10.

In-out device codes are included only in the alphabetic listing and are indicated by a dagger (†). Following the tables is a chart that lists the devices with their mnemonic and octal codes and DEC option numbers for both PDP-10 and PDP-6. A device mnemonic ending in the numeral 2 is the recommended form for the second of a given device, but such codes are not recognized by Macro – they must be defined by the user. Beginning on Page A11 is a list of all instructions showing their actions in symbolic form. Concluding the appendix are charts showing the ASCII code and the formats of the various types of words used in the processor.

MOV
E
e Negative
e Magnitude
e Swapped

Half word
Right
Left
to
Right
Left

no effect
Ones
Zeros
Extend sign

to AC
Immediate to AC
to Memory
to Self

BLock Transfer

EXCHange AC and memory

use present pointer
Increment pointer
and
LoaD Byte into AC
DePosit Byte in memory

Increment Byte Pointer

PUSH down
POP up
~
and Jump

SET to
Zeros
Ones
Ac
Memory
Complement of Ac
Complement of Memory

AND
inclusive OR
~
with Complement of Ac
with Complement of Memory
Complements of Both

Inclusive OR
eXclusive OR
EQuiValence

to
AC
AC Immediate
Memory
Both

SKIP if memory
JUMP if AC

Add One to
Subtract One from
memory and Skip
AC and Jump
if

Compare Ac
Immediate
with Memory
and skip if AC

never
Less
Equal
Less or Equal
Always
Greater
Greater or Equal
Not equal

Add One to Both halves of AC and Jump if
Positive
Negative

Arithmetic SHift
Logical SHift
ROTate
~
Combined

Test AC
with Direct mask
with Swapped mask
Right with E
Left with E

No modification
set masked bits to Zeros
set masked bits to Ones
Complement masked bits

and skip
never
if all masked bits Equal 0
if Not all masked bits equal 0
Always

ADD
SUBtract
MULtiply
Integer MULtiply
DIVide
Integer DIVide

~
Immediate
to Memory
to Both

and Round

Floating AdD
Floating SuBtract
Floating MultiPly
Floating DiVide

~
Long
to Memory
to Both

Floating SCale

Double Floating Negate

Unnormalized Floating Add

FIX

FIX and Round

FLoaT and Round

Double Floating AdD
Double Floating SuBtract
Double Floating MultiPly
Double Floating DiVide

Double MOV
E
e Negative
~
to Memory

Jump
to SubRoutine
and Save Pc
and Save Ac
and Restore Ac
if Find First One
on Flag and CLear it
on OVerflow (JFCL 10,)
on CaRrY 0 (JFCL 4,)
on CaRrY 1 (JFCL 2,)
on CaRrY (JFCL 6,)
on Floating OVerflow (JFCL 1,)
and ReSTore
and ReSTore Flags (JRST 2,)
and ENable PI channel (JRST 12,)

HALT (JRST 4,)

PORTAL (JRST 1,)

eXeCuTe

DATA
BLocK
In
Out

CONditions
in and Skip if
all masked bits Zero
some masked bit One

INSTRUCTION MNEMONICS

NUMERIC LISTING

000	ILLEGAL
001	LUUO'S
:	
:	
037	
040	*CALL
041	*INIT
042	RESERVED FOR SPECIAL MONITORS
043	
044	
045	
046	
047	*CALLI
050	*OPEN
051	*TTCALL
052	RESERVED FOR DEC
053	
054	
055	*RENAME
056	*IN
057	*OUT
060	*SETSTS
061	*STATO
062	*STATUS
062	*GETSTS
063	*STATZ
064	*INBUF
065	*OUTBUF
066	*INPUT
067	*OUTPUT
070	*CLOSE
071	*RELEAS
072	*MTAPE
073	*UGETF
074	*USETI
075	*USETO
076	*LOOKUP
077	*ENTER
100	*UJEN
101	
102	
103	
104	
105	
106	
107	
110	‡DFAD
111	‡DFSB
112	‡DFMP
113	‡DFDV
114	
115	
116	
117	
120	‡DMOVE
121	‡DMOVN
122	‡FIX
123	
124	‡DMOVEM
125	‡DMOVNM
126	‡FIXR
127	‡FLTR
130	UFA
131	DFN
132	FSC
133	IBP
134	ILDB
135	LDB
136	IDPB
137	DPB
140	FAD
141	FADL
142	FADM
143	FADB
144	FADR
145	FADRI
146	FADRM
147	FADRB
150	FSB
151	FSBL
152	FSBM
153	FSBB
154	FSBR
155	FSBRI
156	FSBRM
157	FSBRB
160	FMP
161	FMPL
162	FMPM
163	FMPB
164	FMPR
165	FMPRI
166	FMPRM
167	FMPRB
170	FDV
171	FDVL
172	FDVM
173	FDVB
174	FDVR
175	FDVRI
176	FDVRM
177	FDVRB
200	MOVE
201	MOVEI
202	MOVEM
203	MOVES
204	MOVS
205	MOVSI
206	MOVSM
207	MOVSS
210	MOVN
211	MOVNI
212	MOVNM
213	MOVNS
214	MOVM
215	MOVMI
216	MOVMM
217	MOVMS
220	IMUL
221	IMULI
222	IMULM
223	IMULB
224	MUL
225	MULI
226	MULM
227	MULB
230	IDIV
231	IDIVI
232	IDIVM
233	IDIVB
234	DIV
235	DIVI

236	DIVM
237	DIVB
240	ASH
241	ROT
242	LSH
243	JFFO
244	ASHC
245	ROTC
246	LSHC
247	
250	EXCH
251	BLT
252	AOBJP
253	AOBJN
254	JRST
25404	PORTAL
25410	JRSTF
25420	HALT
25450	JEN
255	JFCL
25504	JFOV
25510	JCRY1
25520	JCRY0
25530	JCRY
25540	JOV
256	XCT
257	‡MAP
260	PUSHJ
261	PUSH
262	POP
263	POPJ
264	JSR
265	JSP
266	JSA
267	JRA
270	ADD
271	ADDI
272	ADDM
273	ADDB
274	SUB
275	SUBI
276	SUBM
277	SUBB
300	CAI
301	CAIL
302	CAIE
303	CAILE
304	CAIA
305	CAIGE
306	CAIN
307	CAIG
310	CAM
311	CAML
312	CAME
313	CAMLE
314	CAMA
315	CAMGE
316	CAMN
317	CAMG
320	JUMP
321	JUMPL
322	JUMPE
323	JUMPLE
324	JUMPA
325	JUMPGE
326	JUMPN
327	JUMPG
330	SKIP
331	SKIPL
332	SKIPE
333	SKIPLE
334	SKIPA
335	SKIPGE
336	SKIPN
337	SKIPG
340	AOJ
341	AOJL
342	AOJE
343	AOJLE
344	AOJA
345	AOJGE
346	AOJN
347	AOJG
350	AOS
351	AOSL
352	AOSE
353	AOSLE
354	AOSA
355	AOSGE
356	AOSN
357	AOSG
360	SOJ
361	SOJL
362	SOJE
363	SOJLE
364	SOJA
365	SOJGE
366	SOJN
367	SOJG
370	SOS
371	SOSL
372	SOSE
373	SOSLE
374	SOSA
375	SOSGE
376	SOSN
377	SOSG
400	SETZ
400	CLEAR
401	SETZI
401	CLEARI
402	SETZM
402	CLEARM
403	SETZB
403	CLEARB
404	AND
405	ANDI
406	ANDM
407	ANDB
410	ANDCA
411	ANDCAI
412	ANDCAM
413	ANDCAB
414	SETM
415	SETMI
416	SETMM
417	SETMB
420	ANDCM
421	ANDCMI
422	ANDCMM
423	ANDCMB
424	SETA
425	SETAI
426	SETAM
427	SETAB
430	XOR
431	XORI
432	XORM
433	XORB
434	IOR
434	OR
435	IORI
435	ORI
436	IORM
436	ORM
437	IORB
437	ORB

440	ANDCB
441	ANDCBI
442	ANDCBM
443	ANDCBB
444	EQV
445	EQVI
446	EQVM
447	EQVB
450	SETCA
451	SETCAI
452	SETCAM
453	SETCAB
454	ORCA
455	ORCAI
456	ORCAM
457	ORCAB
460	SETCM
461	SETCMI
462	SETCMM
463	SETCMB
464	ORCM
465	ORCMI
466	ORCMM
467	ORCMB
470	ORCB
471	ORCBI
472	ORCBM
473	ORCBB
474	SETO
475	SETOI
476	SETOM
477	SETOB
500	HLL
501	HLLI
502	HLLM
503	HLLS
504	HRL
505	HRLI
506	HRLM
507	HRLS
510	HLLZ
511	HLLZI
512	HLLZM
513	HLLZS
514	HRLZ
515	HRLZI
516	HRLZM
517	HRLZS
520	HLLO
521	HLLOI
522	HLLOM
523	HLLOS
524	HRLO
525	HRLOI
526	HRLOM
527	HRLOS
530	HLLE
531	HLLEI
532	HLLEM
533	HLLES
534	HRLE
535	HRLEI
536	HRLEM
537	HRLES
540	HRR
541	HRRI
542	HRRM
543	HRRS
544	HLR
545	HLRI
546	HLRM
547	HLRS
550	HRRZ
551	HRRZI
552	HRRZM
553	HRRZS
554	HLRZ
555	HLRZI
556	HLRZM
557	HLRZS
560	HRRO
561	HRROI
562	HRROM
563	HRROS
564	HLRO
565	HLROI
566	HLROM
567	HLROS
570	HRRE
571	HRREI
572	HRREM
573	HRRES
574	HLRE
575	HLREI
576	HLREM
577	HLRES
600	TRN
601	TLN
602	TRNE
603	TLNE
604	TRNA
605	TLNA
606	TRNN
607	TLNN
610	TDN
611	TSN
612	TDNE
613	TSNE
614	TDNA
615	TSNA
616	TDNN
617	TSNN
620	TRZ
621	TLZ
622	TRZE
623	TLZE
624	TRZA
625	TLZA
626	TRZN
627	TLZN
630	TDZ
631	TSZ
632	TDZE
633	TSZE
634	TDZA
635	TSZA
636	TDZN
637	TSZN
640	TRC
641	TLC
642	TRCE
643	TLCE
644	TRCA
645	TLCA
646	TRCN
647	TLCN
650	TDC
651	TSC
652	TDCE
653	TSCE
654	TDCA
655	TSCA
656	TDCN
657	TSCN
660	TRO
661	TLO
662	TROE

663	TLOE
664	TROA
665	TLOA
666	TRON
667	TLON
670	TDO
671	TSO
672	TDOE
673	TSOE
674	TDOA
675	TSOA
676	TDON
677	TSON
70000	BLKI
70004	DATAI
70004	RSW
70010	BLKO
70014	DATAO
70020	CONO
70024	CONI
70030	CONSZ
70034	CONSO

INSTRUCTION MNEMONICS

ALPHABETIC LISTING

†ADC	024
ADD	270
ADDB	273
ADDI	271
ADDM	272
AND	404
ANDB	407
ANDCA	410
ANDCAB	413
ANDCAI	411
ANDCAM	412
ANDCB	440
ANDCBB	443
ANDCBI	441
ANDCBM	442
ANDCM	420
ANDCMB	423
ANDCMI	421
ANDCMM	422
ANDI	405
ANDM	406
AOBJN	253
AOBJP	252
AOJ	340
AOJA	344
AOJE	342
AOJG	347
AOJGE	345
AOJL	341
AOJLE	343
AOJN	346
AOS	350
AOSA	354
AOSE	352
AOSG	357
AOSGE	355
AOSL	351
AOSLE	353
AOSN	356
†APR	000
ASH	240
ASHC	244
BLKI	70000
BLKO	70010
BLT	251
CAI	300
CAIA	304
CAIE	302
CAIG	307
CAIGE	305
CAIL	301
CAILE	303
CAIN	306
*CALL	040
*CALLI	047
CAM	310
CAMA	314
CAME	312
CAMG	317
CAMGE	315
CAML	311
CAMLE	313
CAMN	316
†CCI	014
†CDP	110
†CDR	114
CLEAR	400
CLEARB	403
CLEARI	401
CLEARM	402
†CLK	070
*CLOSE	070
CONI	70024
CONO	70020
CONSO	70034
CONSZ	70030
†CPA	000
†CR	150
DATAI	70004
DATAO	70014
†DC	200
†DCSA	300
†DCSB	304
‡DFAD	110
‡DFDV	113
‡DFMP	112
DFN	131
‡DFSB	111
†DIS	130
DIV	234
DIVB	237
DIVI	235
DIVM	236
†DLB	060
†DLC	064
†DLS	240

‡DMOVE	120
‡DMOVEM	124
‡DMOVN	121
‡DMOVNM	125
DPB	137
†DPC	250
†DSI	464
†DSK	170
†DSS	460
†DTC	320
†DTS	324
*ENTER	077
EQV	444
EQVB	447
EQVI	445
EQVM	446
EXCH	250
FAD	140
FADB	143
FADL	141
FADM	142
FADR	144
FADRB	147
FADRI	145
FADRM	146
FDV	170
FDVB	173
FDVL	171
FDVM	172
FDVR	174
FDVRB	177
FDVRI	175
FDVRM	176
‡FIX	122
‡FIXR	126
‡FLTR	127
FMP	160
FMPB	163
FMPL	161
FMPM	162
FMPR	164
FMPRB	167
FMPRI	165
FMPRM	166
FSB	150
FSBB	153
FSBL	151
FSBM	152
FSBR	154
FSBRB	157
FSBRI	155
FSBRM	156
FSC	132
*GETSTS	062
HALT	25420
HLL	500
HLLE	530
HLLEI	531
HLLEM	532
HLLES	533
HLLI	501
HLLM	502
HLLO	520
HLLOI	521
HLLOM	522
HLLOS	523
HLLS	503
HLLZ	510
HLLZI	511
HLLZM	512
HLLZS	513
HLR	544
HLRE	574
HLREI	575
HLREM	576
HLRES	577
HLRI	545
HLRM	546
HLRO	564
HLROI	565
HLROM	566
HLROS	567
HLRS	547
HLRZ	554
HLRZI	555
HLRZM	556
HLRZS	557
HRL	504
HRLE	534
HRLEI	535
HRLEM	536
HRLES	537
HRLI	505
HRLM	506
HRLO	524
HRLOI	525
HRLOM	526
HRLOS	527
HRLS	507
HRLZ	514
HRLZI	515
HRLZM	516
HRLZS	517
HRR	540
HRRE	570
HRREI	571
HRREM	572
HRRES	573
HRRI	541
HRRM	542
HRRO	560
HRROI	561
HRROM	562
HRROS	563
HRRS	543
HRRZ	550
HRRZI	551
HRRZM	552
HRRZS	553
IBP	133
IDIV	230
IDIVB	233
IDIVI	231
IDIVM	232
IDPB	136
ILDB	134
IMUL	220
IMULB	223
IMULI	221
IMULM	222
*IN	056
*INBUF	064
*INIT	041
*INPUT	066
IOR	434
IORB	437
IORI	435
IORM	436
JCRY	25530
JCRY0	25520
JCRY1	25510
JEN	25460
JFCL	255
JFFO	243
JFOV	25504
JOV	25540
JRA	267

JRST	254
JRSTF	25410
JSA	266
JSP	265
JSR	264
JUMP	320
JUMPA	324
JUMPE	322
JUMPG	327
JUMPGE	325
JUMPL	321
JUMPLE	323
JUMPN	326
LDB	135
*LOOKUP	076
†LPT	124
LSH	242
LSHC	246
‡MAP	257
†MDF	260
MOVE	200
MOVEI	201
MOVEM	202
MOVES	203
MOVM	214
MOVMI	215
MOVMM	216
MOVMS	217
MOVN	210
MOVNI	211
MOVNM	212
MOVNS	213
MOVS	204
MOVSI	205
MOVSM	206
MOVSS	207
*MTAPE	072
†MTC	220
†MTM	230
†MTS	224
MUL	224
MULB	227
MULI	225
MULM	226
*OPEN	050
OR	434
ORB	437
ORCA	454
ORCAB	457
ORCAI	455
ORCAM	456
ORCB	470
ORCBB	473
ORCBI	471
ORCBM	472
ORCM	464
ORCMB	467
ORCMI	465
ORCMM	466
ORI	435
ORM	436
*OUT	057
*OUTBUF	065
*OUTPUT	067
†PAG	010
†PI	004
†PLT	140
POP	262
POPJ	263
PORTAL	25404
†PTP	100
†PTR	104
PUSH	261
PUSHJ	260
*RELEAS	071
*RENAME	055
ROT	241
ROTC	245
RSW	70004
SETA	424
SETAB	427
SETAI	425
SETAM	426
SETCA	450
SETCAB	453
SETCAI	451
SETCAM	452
SETCM	460
SETCMB	463
SETCMI	461
SETCMM	462
SETM	414
SETMB	417
SETMI	415
SETMM	416
SETO	474
SETOB	477
SETOI	475
SETOM	476
*SETSTS	060
SETZ	400
SETZB	403
SETZI	401
SETZM	402
SKIP	330
SKIPA	334
SKIPE	332
SKIPG	337
SKIPGE	335
SKIPL	331
SKIPLE	333
SKIPN	336
SOJ	360
SOJA	364
SOJE	362
SOJG	367
SOJGE	365
SOJL	361
SOJLE	363
SOJN	366
SOS	370
SOSA	374
SOSE	372
SOSG	377
SOSGE	375
SOSL	371
SOSLE	373
SOSN	376
*STATO	061
*STATUS	062
*STATZ	063
SUB	274
SUBB	277
SUBI	275
SUBM	276
TDC	650
TDCA	654
TDCE	652
TDCN	656
TDN	610
TDNA	614
TDNE	612
TDNN	616
TDO	670
TDOA	674
TDOE	672
TDON	676

TDZ	630
TDZA	634
TDZE	632
TDZN	636
TLC	641
TLCA	645
TLCE	643
TLCN	647
TLN	601
TLNA	605
TLNE	603
TLNN	607
TLO	661
TLOA	665
TLOE	663
TLON	667
TLZ	621
TLZA	625
TLZE	623
TLZN	627
†TMC	340
†TMS	344
TRC	640
TRCA	644
TRCE	642
TRCN	646
TRN	600
TRNA	604
TRNE	602
TRNN	606
TRO	660
TROA	664
TROE	662
TRON	666
TRZ	620
TRZA	624
TRZE	622
TRZN	626
TSC	651
TSCA	655
TSCE	653
TSCN	657
TSN	611
TSNA	615
TSNE	613
TSNN	617
TSO	671
TSOA	675
TSOE	673
TSON	677
TSZ	631
TSZA	635
TSZE	633
TSZN	637
*TTCALL	051
UFA	130
*UGETF	073
*UJEN	100
*USETI	074
*USETO	075
†UTC	210
†UTS	214
XCT	256
XOR	430
XORB	433
XORI	431
XORM	432

FIRST OCTAL DIGIT \ SECOND AND THIRD OCTAL DIGITS	00	04	10	14	20	24	30	34	40	44	50	54	60	64	70	74
0	6,10 APR CPA CENTRAL PROCESSOR	6,10 PI PRIORITY INTERRUPT	10 PAG* KI10 PAGING	10 DA10 CCI PDP-8,9 INTERFACE	10 DA10 CCI2 PDP-8,9 INTERFACE	10 AD10 ADC ANALOG-DIGITAL CONVERTER	10 AD10 ADC2 ANALOG-DIGITAL CONVERTER						10 DLB PDP-11 DATA LINK	DL10 DLC	10 DK10 CLK REAL TIME CLOCK	10 DK10 CLK2 REAL TIME CLOCK
1	6 761 10 PTP PAPER TAPE PUNCH	6 760 10 PTR PAPER TAPE READER	10 CP10 CDP CARD PUNCH	6 461 CDR CARD READER	6 626 10 TTY CONSOLE TELETYPE	6 646 10 LP10 LPT LINE PRINTER	6,10 340 10 VP10 DIS DISPLAY	6,10 340 10 VP10 DIS2 DISPLAY	10 XY10 PLT PLOTTER	10 XY10 PLT2 PLOTTER	10 CR10 CR CARD READER	10 CR10 CR2 CARD READER	10 DLB2† PDP-11 DATA LINK	DL10 DLC2	10 RC10 DSK SMALL DISK	10 RC10 DSK2 SMALL DISK
2	6 136 DC DATA CONTROL	6 136 DC2 DATA CONTROL	6 UTC DECTAPE	551 UTS	6 MTC MAGNETIC TAPE	MTS	516 MTM	6 646 10 LP10 LPT2 LINE PRINTER	10 DC10 DLS DATA LINE SCANNER	10 DC10 DLS2 DATA LINE SCANNER	10 RP10 DPC DISK PACK SYSTEM	10 RP10 DPC2 DISK PACK SYSTEM	10 RP10 DPC3 DISK PACK SYSTEM	10 RP10 DPC4 DISK PACK SYSTEM	6 270 DF DISK FILE	
3	6 DCSA DATA COMMUNICATION	630 DCSB			10 DTC DECTAPE	TD10 DTS	10 DTC2 DECTAPE	TD10 DTS2	10 TMC MAGNETIC TAPE	TM10 TMS	10 TMC2 MAGNETIC TAPE	TM10 TMS2				
4													10 DSS SINGLE SYNCHRONOUS LINE UNIT	DS10 DSI	10 DSS2 SINGLE SYNCHRONOUS LINE UNIT	DS10 DSI2
5																
6																
7																

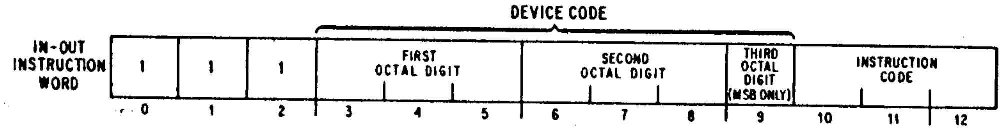
CODES IN THIS SECTION RESERVED FOR USER SPECIAL DEVICES

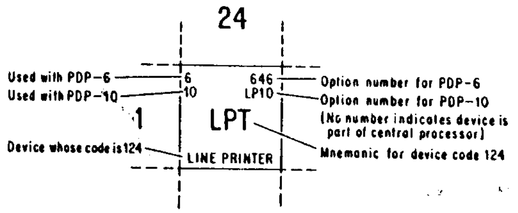
KI10 UNRESTRICTED CODES RESERVED FOR USERS

KI10 UNRESTRICTED CODES RESERVED FOR DEC

*DRUM PROCESSOR IN PDP-6

†PDP-7,8 INTERFACE IN PDP-6





DEVICE MNEMONICS

ALGEBRAIC REPRESENTATION

Pages A13–A20 of this Appendix list, in symbolic form, the actual operations performed by the instructions. They are grouped in this way.

Boolean	A13	In-out	A17
Byte manipulation	A14	Program control	A17
Fixed point arithmetic	A14	Pushdown list	A17
Floating point arithmetic	A14	Shift and rotate	A17
Full word data transmission	A15	Test, arithmetic	A18
Half word data transmission	A16	Test, logical	A19

The terminology and notation used vary somewhat from that in the System Reference Manual, as follows.

AC	The accumulator address in bits 9–12 of the instruction word (represented by *A* in the instruction descriptions).
AC+1	The address one greater than AC, except that AC+1 is 0 if AC is 17.
E	The result of the effective address calculation. E is eighteen bits when used as an address, half word operand, mask or output conditions, but is a signed 9-bit quantity when used as a scale factor or a shift number.
E+1	The address one greater than E, except that E+1 is 0 if E is 777777.
PC	The 18-bit program counter.
(X)	The word contained in register *X*.
$(X)_L$	The left half of (X).
$(X)_R$	The right half of (X).
$(X)_S$	The word contained in *X* with its left and right halves swapped.
A_n	The value of bit *n* of the quantity *A*.
A,B	A 36-bit word with the 18-bit quantity *A* in its left half and the 18-bit quantity *B* in its right half (either *A* or *B* may be 0).
(X,Y)	The contents of registers *X* and *Y* concatenated into a double word operand.
$((X))$	The word contained in the register addressed by (X), *ie* addressed by the word in register *X*.
$A \to B$	The quantity *A* replaces the quantity *B* (*A* and *B* may be half words, full words or double words). *Eg* $(\mathrm{AC}) + (\mathrm{E}) \to (\mathrm{AC})$ means the word in accumulator AC plus the word in memory location E replaces the word in AC.

(AC) (E)	The word in AC and the word in E.
∧ ∨ ∀ ~	The Boolean operators AND, inclusive OR, exclusive OR, and complement (logical negation).
+ − × ÷ | |	The arithmetic operators for addition, negation or subtraction, multiplication, division, and absolute value (magnitude).

Square brackets are used occasionally for grouping. With respect to the values of their terms, the equations for a given instruction are in chronological order; *eg* in the pair of equations

$$(AC) + 1 \rightarrow (AC)$$
$$\mathit{If}\ (AC) = 0\text{:}\quad E \rightarrow (PC)$$

the quantity tested in the second equation is the word in AC after it has been incremented by one. Page references in brackets are for the detailed explanations in Chapter 2 of the System Reference Manual.

Boolean

SETZ	400	0 → (AC)	SETO	474	777777777777 → (AC)
SETZI	401	0 → (AC)	SETOI	475	777777777777 → (AC)
SETZM	402	0 → (E)	SETOM	476	777777777777 → (E)
SETZB	403	0 → (AC) (E)	SETOB	477	777777777777 → (AC) (E)
SETA	424	(AC) → (AC) [*no-op*]	SETCA	450	~ (AC) → (AC)
SETAI	425	(AC) → (AC) [*no-op*]	SETCAI	451	~ (AC) → (AC)
SETAM	426	(AC) → (E)	SETCAM	452	~ (AC) → (E)
SETAB	427	(AC) → (E)	SETCAB	453	~ (AC) → (AC) (E)
SETM	414	(E) → (AC)	SETCM	460	~ (E) → (AC)
SETMI	415	0,E → (AC)	SETCMI	461	~ [0,E] → (AC)
SETMM	416	(E) → (E) [*no-op*]	SETCMM	462	~ (E) → (E)
SETMB	417	(E) → (AC) (E)	SETCMB	463	~ (E) → (AC) (E)
AND	404	(AC) ∧ (E) → (AC)	ANDCA	410	~ (AC) ∧ (E) → (AC)
ANDI	405	(AC) ∧ 0,E → (AC)	ANDCAI	411	~ (AC) ∧ 0,E → (AC)
ANDM	406	(AC) ∧ (E) → (E)	ANDCAM	412	~ (AC) ∧ (E) → (E)
ANDB	407	(AC) ∧ (E) → (AC) (E)	ANDCAB	413	~ (AC) ∧ (E) → (AC) (E)
ANDCM	420	(AC) ∧ ~ (E) → (AC)	ANDCB	440	~ (AC) ∧ ~ (E) → (AC)
ANDCMI	421	(AC) ∧ ~ [0,E] → (AC)	ANDCBI	441	~ (AC) ∧ ~ [0,E] → (AC)
ANDCMM	422	(AC) ∧ ~ (E) → (E)	ANDCBM	442	~ (AC) ∧ ~ (E) → (E)
ANDCMB	423	(AC) ∧ ~ (E) → (AC) (E)	ANDCBB	443	~ (AC) ∧ ~ (E) → (AC) (E)
IOR	434	(AC) ∨ (E) → (AC)	ORCA	454	~ (AC) ∨ (E) → (AC)
IORI	435	(AC) ∨ 0,E → (AC)	ORCAI	455	~ (AC) ∨ 0,E → (AC)
IORM	436	(AC) ∨ (E) → (E)	ORCAM	456	~ (AC) ∨ (E) → (E)
IORB	437	(AC) ∨ (E) → (AC) (E)	ORCAB	457	~ (AC) ∨ (E) → (AC) (E)
ORCM	464	(AC) ∨ ~ (E) → (AC)	ORCB	470	~ (AC) ∨ ~ (E) → (AC)
ORCMI	465	(AC) ∨ ~ [0,E] → (AC)	ORCBI	471	~ (AC) ∨ ~ [0,E] → (AC)
ORCMM	466	(AC) ∨ ~ (E) → (E)	ORCBM	472	~ (AC) ∨ ~ (E) → (E)
ORCMB	467	(AC) ∨ ~ (E) → (AC) (E)	ORCBB	473	~ (AC) ∨ ~ (E) → (AC) (E)
XOR	430	(AC) ⊻ (E) → (AC)	EQV	444	~ [(AC) ⊻ (E)] → (AC)
XORI	431	(AC) ⊻ 0,E → (AC)	EQVI	445	~ [(AC) ⊻ 0,E] → (AC)
XORM	432	(AC) ⊻ (E) → (E)	EQVM	446	~ [(AC) ⊻ (E)] → (E)
XORB	433	(AC) ⊻ (E) → (AC) (E)	EQVB	447	~ [(AC) ⊻ (E)] → (AC) (E)

Byte Manipulation

IBP	133	*Operations on* (E) [*see page 2-16*] *If* P − S ≥ 0: P − S → P *If* P − S < 0: Y + 1 → Y 36 − S → P
LDB	135	BYTE IN ((E)) → (AC) [*see page 2-16*]
DPB	137	BYTE IN (AC) → BYTE IN ((E)) [*see page 2-16*]
ILDB	134	IBP *and* LDB
IDPB	136	IBP *and* DPB

Fixed Point Arithmetic

ADD	270	(AC) + (E) → (AC)	SUB	274	(AC) − (E) → (AC)
ADDI	271	(AC) + 0,E → (AC)	SUBI	275	(AC) − 0,E → (AC)
ADDM	272	(AC) + (E) → (E)	SUBM	276	(AC) − (E) → (E)
ADDB	273	(AC) + (E) → (AC) (E)	SUBB	277	(AC) − (E) → (AC) (E)
IMUL	220	(AC) × (E) → (AC)*	MUL	224	(AC) × (E) → (AC,AC+1)
IMULI	221	(AC) × 0,E → (AC)*	MULI	225	(AC) × 0,E → (AC,AC+1)
IMULM	222	(AC) × (E) → (E)*	MULM	226	(AC) × (E) → (E)†
IMULB	223	(AC) × (E) → (AC) (E)*	MULB	227	(AC) × (E) → (AC,AC+1) (E)
IDIV	230	(AC) ÷ (E) → (AC) REMAINDER → (AC+1)	DIV	234	(AC,AC+1) ÷ (E) → (AC) REMAINDER → (AC+1)
IDIVI	231	(AC) ÷ 0,E → (AC) REMAINDER → (AC+1)	DIVI	235	(AC,AC+1) ÷ 0,E → (AC) REMAINDER → (AC+1)
IDIVM	232	(AC) ÷ (E) → (E)	DIVM	236	(AC,AC+1) ÷ (E) → (E)
IDIVB	233	(AC) ÷ (E) → (AC) (E) REMAINDER → (AC+1)	DIVB	237	(AC,AC+1) ÷ (E) → (AC) (E) REMAINDER → (AC+1)

*The high order word of the product is discarded.
†The low order word of the product is discarded.

Floating Point Arithmetic

FAD	140	(AC) + (E) → (AC)	FADR	144	(AC) + (E) → (AC)
FADL	141	(AC) + (E) → (AC,AC+1)	FADRI	145	(AC) + E,0 → (AC)
FADM	142	(AC) + (E) → (E)	FADRM	146	(AC) + (E) → (E)
FADB	143	(AC) + (E) → (AC) (E)	FADRB	147	(AC) + (E) → (AC) (E)
FSB	150	(AC) − (E) → (AC)	FSBR	154	(AC) − (E) → (AC)
FSBL	151	(AC) − (E) → (AC,AC+1)	FSBRI	155	(AC) − E,0 → (AC)
FSBM	152	(AC) − (E) → (E)	FSBRM	156	(AC) − (E) → (E)
FSBB	153	(AC) − (E) → (AC) (E)	FSBRB	157	(AC) − (E) → (AC) (E)

FMP	160	(AC) × (E) → (AC)	FMPR	164	(AC) × (E) → (AC)
FMPL	161	(AC) × (E) → (AC,AC+1)	FMPRI	165	(AC) × E,0 → (AC)
FMPM	162	(AC) × (E) → (E)	FMPRM	166	(AC) × (E) → (E)
FMPB	163	(AC) × (E) → (AC) (E)	FMPRB	167	(AC) × (E) → (AC) (E)
FDV	170	(AC) ÷ (E) → (AC)	FDVR	174	(AC) ÷ (E) → (AC)
FDVL	171	(AC) ÷ (E) → (AC) REMAINDER → (AC+1)	FDVRI	175	(AC) ÷ E,0 → (AC)
FDVM	172	(AC) ÷ (E) → (E)	FDVRM	176	(AC) ÷ (E) → (E)
FDVB	173	(AC) ÷ (E) → (AC) (E)	FDVRB	177	(AC) ÷ (E) → (AC) (E)

UFA	130	(AC) + (E) → (AC+1) *without normalization*
DFN	131	− (AC,E) → (AC,E)
FSC	132	(AC) × 2^E → (AC)
FLTR	127	(E) *floated, rounded* → (AC)

FIX	122	(E) *fixed* → (AC)	FIXR	126	(E) *fixed, rounded* → (AC)

DFAD	110	(AC,AC+1) + (E,E+1) → (AC,AC+1)
DFSB	111	(AC,AC+1) − (E,E+1) → (AC,AC+1)
DFMP	112	(AC,AC+1) × (E,E+1) → (AC,AC+1)
DFDV	113	(AC,AC+1) ÷ (E,E+1) → (AC,AC+1)

DMOVE	120	(E,E+1) → (AC,AC+1)	DMOVEM	124	(AC,AC+1) → (E,E+1)
DMOVN	121	− (E,E+1) → (AC,AC+1)	DMOVNM	125	− (AC,AC+1) → (E,E+1)

Full Word Data Transmission

EXCH	250	(AC) ↔ (E)
BLT	251	*Move* E − $(AC)_R$ + 1 *words starting with* $((AC)_L)$ → $((AC)_R)$ [*see page 2-10*]

MOVE	200	(E) → (AC)	MOVS	204	$(E)_S$ → (AC)
MOVEI	201	0,E → (AC)	MOVSI	205	E,0 → (AC)
MOVEM	202	(AC) → (E)	MOVSM	206	$(AC)_S$ → (E)
MOVES	203	*If* AC ≠ 0: (E) → (AC)	MOVSS	207	$(E)_S$ → (E) *If* AC ≠ 0: (E) → (AC)
MOVN	210	− (E) → (AC)	MOVM	214	|(E)| → (AC)
MOVNI	211	− [0,E] → (AC)	MOVMI	215	0,E → (AC)
MOVNM	212	− (AC) → (E)	MOVMM	216	|(AC)| → (E)
MOVNS	213	− (E) → (E) *If* AC ≠ 0: (E) → (AC)	MOVMS	217	|(E)| → (E) *If* AC ≠ 0: (E) → (AC)

Half Word Data Transmission

HLL	500	$(E)_L \rightarrow (AC)_L$	HLLZ	510	$(E)_L,0 \rightarrow (AC)$
HLLI	501	$0 \rightarrow (AC)_L$	HLLZI	511	$0 \rightarrow (AC)$
HLLM	502	$(AC)_L \rightarrow (E)_L$	HLLZM	512	$(AC)_L,0 \rightarrow (E)$
HLLS	503	*If* $AC \neq 0$: $(E) \rightarrow (AC)$	HLLZS	513	$0 \rightarrow (E)_R$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HLLO	520	$(E)_L,777777 \rightarrow (AC)$	HLLE	530	$(E)_L,[(E)_0 \times 777777] \rightarrow (AC)$
HLLOI	521	$0,777777 \rightarrow (AC)$	HLLEI	531	$0 \rightarrow (AC)$
HLLOM	522	$(AC)_L,777777 \rightarrow (E)$	HLLEM	532	$(AC)_L,[(AC)_0 \times 777777] \rightarrow (E)$
HLLOS	523	$777777 \rightarrow (E)_R$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HLLES	533	$(E)_0 \times 777777 \rightarrow (E)_R$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HLR	544	$(E)_L \rightarrow (AC)_R$	HLRZ	554	$0,(E)_L \rightarrow (AC)$
HLRI	545	$0 \rightarrow (AC)_R$	HLRZI	555	$0 \rightarrow (AC)$
HLRM	546	$(AC)_L \rightarrow (E)_R$	HLRZM	556	$0,(AC)_L \rightarrow (E)$
HLRS	547	$(E)_L \rightarrow (E)_R$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HLRZS	557	$0,(E)_L \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HLRO	564	$777777,(E)_L \rightarrow (AC)$	HLRE	574	$[(E)_0 \times 777777],(E)_L \rightarrow (AC)$
HLROI	565	$777777,0 \rightarrow (AC)$	HLREI	575	$0 \rightarrow (AC)$
HLROM	566	$777777,(AC)_L \rightarrow (E)$	HLREM	576	$[(AC)_0 \times 777777],(AC)_L \rightarrow (E)$
HLROS	567	$777777,(E)_L \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HLRES	577	$[(E)_0 \times 777777],(E)_L \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HRR	540	$(E)_R \rightarrow (AC)_R$	HRRZ	550	$0,(E)_R \rightarrow (AC)$
HRRI	541	$E \rightarrow (AC)_R$	HRRZI	551	$0,E \rightarrow (AC)$
HRRM	542	$(AC)_R \rightarrow (E)_R$	HRRZM	552	$0,(AC)_R \rightarrow (E)$
HRRS	543	*If* $AC \neq 0$: $(E) \rightarrow (AC)$	HRRZS	553	$0 \rightarrow (E)_L$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HRRO	560	$777777,(E)_R \rightarrow (AC)$	HRRE	570	$[(E)_{18} \times 777777],(E)_R \rightarrow (AC)$
HRROI	561	$777777,E \rightarrow (AC)$	HRREI	571	$[E_{18} \times 777777],E \rightarrow (AC)$
HRROM	562	$777777,(AC)_R \rightarrow (E)$	HRREM	572	$[(AC)_{18} \times 777777],(AC)_R \rightarrow (E)$
HRROS	563	$777777 \rightarrow (E)_L$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HRRES	573	$(E)_{18} \times 777777 \rightarrow (E)_L$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HRL	504	$(E)_R \rightarrow (AC)_L$	HRLZ	514	$(E)_R,0 \rightarrow (AC)$
HRLI	505	$E \rightarrow (AC)_L$	HRLZI	515	$E,0 \rightarrow (AC)$
HRLM	506	$(AC)_R \rightarrow (E)_L$	HRLZM	516	$(AC)_R,0 \rightarrow (E)$
HRLS	507	$(E)_R \rightarrow (E)_L$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HRLZS	517	$(E)_R,0 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$

HRLO	524	$(E)_R,777777 \rightarrow (AC)$	HRLE	534	$(E)_R,[(E)_{18} \times 777777] \rightarrow (AC)$
HRLOI	525	$E,777777 \rightarrow (AC)$	HRLEI	535	$E,[E_{18} \times 777777] \rightarrow (AC)$
HRLOM	526	$(AC)_R,777777 \rightarrow (E)$	HRLEM	536	$(AC)_R,[(AC)_{18} \times 777777] \rightarrow (E)$
HRLOS	527	$(E)_R,777777 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HRLES	537	$(E)_R,[(E)_{18} \times 777777] \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$

In-out

CONO	70020	$E \rightarrow$ COMMAND	CONSZ	70030	*If* $\text{STATUS}_R \wedge E = 0$: *skip*
CONI	70024	STATUS $\rightarrow (E)$	CONSO	70034	*If* $\text{STATUS}_R \wedge E \neq 0$: *skip*
DATAO	70014	$(E) \rightarrow$ DATA	DATAI	70004	DATA $\rightarrow (E)$

BLKO	70010	$(E) + 1000001 \rightarrow (E)$* $((E)_R) \rightarrow$ DATA [*see page 2-77*]
BLKI	70000	$(E) + 1000001 \rightarrow (E)$* DATA $\rightarrow ((E)_R)$ [*see page 2-77*]

Program Control

JSR	264	FLAGS,$(PC) \rightarrow (E)$ $E + 1 \rightarrow (PC)$
JSP	265	FLAGS,$(PC) \rightarrow (AC)$ $E \rightarrow (PC)$
JRST	254	$E \rightarrow (PC)$ [*If* $AC \neq 0$, *see page 2-63*]
JSA	266	$(AC) \rightarrow (E)$ $E,(PC) \rightarrow (AC)$ $E + 1 \rightarrow (PC)$
JRA	267	$E \rightarrow (PC)$ $((AC)_L) \rightarrow (AC)$
JFCL	255	*If* $AC \wedge$ FLAGS $\neq 0$: $E \rightarrow (PC)$ $\sim AC \wedge$ FLAGS $\rightarrow$ FLAGS
XCT	256	*Execute* (E)
JFFO	243	*If* $(AC) = 0$: $0 \rightarrow (AC + 1)$ *If* $(AC) \neq 0$: $E \rightarrow (PC)$ [*see page 2-61*]
MAP	257	PHYSICAL MAP DATA $\rightarrow (AC)$

Pushdown List

PUSH	261	$(AC) + 1000001 \rightarrow (AC)$* $(E) \rightarrow ((AC)_R)$
POP	262	$((AC)_R) \rightarrow (E)$ $(AC) - 1000001 \rightarrow (AC)$*
PUSHJ	260	$(AC) + 1000001 \rightarrow (AC)$* FLAGS,$(PC) \rightarrow ((AC)_R)$ $E \rightarrow (PC)$
POPJ	263	$((AC)_R)_R \rightarrow (PC)$ $(AC) - 1000001 \rightarrow (AC)$*

Shift and Rotate

ASH	240	$(AC) \times 2^E \rightarrow (AC)$	ASHC	245	$(AC,AC+1) \times 2^E \rightarrow (AC,AC+1)$
ROT	241	*Rotate* (AC) E *places*	ROTC	246	*Rotate* (AC,AC+1) E *places*
LSH	242	*Shift* (AC) E *places*	LSHC	247	*Shift* (AC,AC+1) E *places*

*In the KI10, 1 is added to or subtracted from each half separately.

Arithmetic Testing

AOBJP	252	(AC) + 1000001 → (AC)*	*If* (AC) ⩾ 0: E → (PC)
AOBJN	253	(AC) + 1000001 → (AC)*	*If* (AC) < 0: E → (PC)

CAI	300	*No-op*	CAM	310	*No-op*
CAIL	301	*If* (AC) < E: *skip*	CAML	311	*If* (AC) < (E): *skip*
CAIE	302	*If* (AC) = E: *skip*	CAME	312	*If* (AC) = (E): *skip*
CAILE	303	*If* (AC) ⩽ E: *skip*	CAMLE	313	*If* (AC) ⩽ (E): *skip*
CAIA	304	*Skip*	CAMA	314	*Skip*
CAIGE	305	*If* (AC) ⩾ E: *skip*	CAMGE	315	*If* (AC) ⩾ (E): *skip*
CAIN	306	*If* (AC) ≠ E: *skip*	CAMN	316	*If* (AC) ≠ (E): *skip*
CAIG	307	*If* (AC) > E: *skip*	CAMG	317	*If* (AC) > (E): *skip*
JUMP	320	*No-op*	SKIP	330	*If* AC ≠ 0: (E) → (AC)
JUMPL	321	*If* (AC) < 0: E → (PC)	SKIPL	331	*If* AC ≠ 0: (E) → (AC) *If* (E) < 0: *skip*
JUMPE	322	*If* (AC) = 0: E → (PC)	SKIPE	332	*If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*
JUMPLE	323	*If* (AC) ⩽ 0: E → (PC)	SKIPLE	333	*If* AC ≠ 0: (E) → (AC) *If* (E) ⩽ 0: *skip*
JUMPA	324	E → (PC)	SKIPA	334	*If* AC ≠ 0: (E) → (AC) *Skip*
JUMPGE	325	*If* (AC) ⩾ 0: E → (PC)	SKIPGE	335	*If* AC ≠ 0: (E) → (AC) *If* (E) ⩾ 0: *skip*
JUMPN	326	*If* (AC) ≠ 0: E → (PC)	SKIPN	336	*If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*
JUMPG	327	*If* (AC) > 0: E → (PC)	SKIPG	337	*If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*
AOJ	340	(AC) + 1 → (AC)	SOJ	360	(AC) − 1 → (AC)
AOJL	341	(AC) + 1 → (AC) *If* (AC) < 0: E → (PC)	SOJL	361	(AC) − 1 → (AC) *If* (AC) < 0: E → (PC)
AOJE	342	(AC) + 1 → (AC) *If* (AC) = 0: E → (PC)	SOJE	362	(AC) − 1 → (AC) *If* (AC) = 0: E → (PC)
AOJLE	343	(AC) + 1 → (AC) *If* (AC) ⩽ 0: E → (PC)	SOJLE	363	(AC) − 1 → (AC) *If* (AC) ⩽ 0: E → (PC)
AOJA	344	(AC) + 1 → (AC) E → (PC)	SOJA	364	(AC) − 1 → (AC) E → (PC)
AOJGE	345	(AC) + 1 → (AC) *If* (AC) ⩾ 0: E → (PC)	SOJGE	365	(AC) − 1 → (AC) *If* (AC) ⩾ 0: E → (PC)

---

*In the KI10, 1 is added to or subtracted from each half separately.

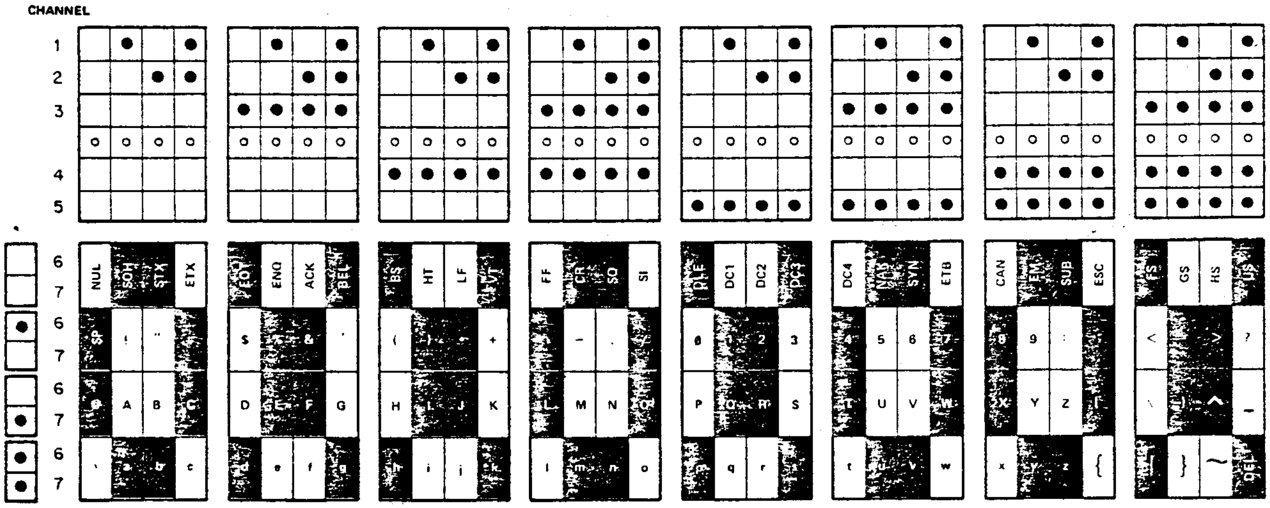
AOJN	346	$(AC) + 1 \rightarrow (AC)$ *If* $(AC) \neq 0$: $E \rightarrow (PC)$	SOJN	366	$(AC) - 1 \rightarrow (AC)$ *If* $(AC) \neq 0$: $E \rightarrow (PC)$
AOJG	347	$(AC) + 1 \rightarrow (AC)$ *If* $(AC) > 0$: $E \rightarrow (PC)$	SOJG	367	$(AC) - 1 \rightarrow (AC)$ *If* $(AC) > 0$: $E \rightarrow (PC)$
AOS	350	$(E) + 1 \rightarrow (E)$ *If* $(AC) \neq 0$: $(E) \rightarrow (AC)$	SOS	370	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
AOSL	351	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) < 0$: *skip*	SOSL	371	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) < 0$: *skip*
AOSE	352	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) = 0$: *skip*	SOSE	372	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) = 0$: *skip*
AOSLE	353	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \leqslant 0$: *skip*	SOSLE	373	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \leqslant 0$: *skip*
AOSA	354	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *Skip*	SOSA	374	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *Skip*
AOSGE	355	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \geqslant 0$: *skip*	SOSGE	375	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \geqslant 0$: *skip*
AOSN	356	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \neq 0$: *skip*	SOSN	376	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \neq 0$: *skip*
AOSG	357	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) > 0$: *skip*	SOSG	377	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) > 0$: *skip*

Logical Testing and Modification

TLN	601	*No-op*	TRN	600	*No-op*
TLNE	603	*If* $(AC)_L \wedge E = 0$: *skip*	TRNE	602	*If* $(AC)_R \wedge E = 0$: *skip*
TLNA	605	*Skip*	TRNA	604	*Skip*
TLNN	607	*If* $(AC)_L \wedge E \neq 0$: *skip*	TRNN	606	*If* $(AC)_R \wedge E \neq 0$: *skip*
TLZ	621	$(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZ	620	$(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZE	623	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZE	622	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZA	625	$(AC)_L \wedge \sim E \rightarrow (AC)_L$ *skip*	TRZA	624	$(AC)_R \wedge \sim E \rightarrow (AC)_R$ *skip*
TLZN	627	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZN	626	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$

TLC	641	$(AC)_L \veebar E \rightarrow (AC)_L$	TRC	640	$(AC)_R \veebar E \rightarrow (AC)_R$
TLCE	643	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \veebar E \rightarrow (AC)_L$	TRCE	642	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \veebar E \rightarrow (AC)_R$
TLCA	645	$(AC)_L \veebar E \rightarrow (AC)_L$ *skip*	TRCA	644	$(AC)_R \veebar E \rightarrow (AC)_R$ *skip*
TLCN	647	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \veebar E \rightarrow (AC)_L$	TRCN	646	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \veebar E \rightarrow (AC)_R$
TLO	661	$(AC)_L \vee E \rightarrow (AC)_L$	TRO	660	$(AC)_R \vee E \rightarrow (AC)_R$
TLOE	663	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TROE	662	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TLOA	665	$(AC)_L \vee E \rightarrow (AC)_L$ *skip*	TROA	664	$(AC)_R \vee E \rightarrow (AC)_R$ *skip*
TLON	667	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TRON	666	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TDN	610	*No-op*	TSN	611	*No-op*
TDNE	612	*If* $(AC) \wedge (E) = 0$: *skip*	TSNE	613	*If* $(AC) \wedge (E)_S = 0$: *skip*
TDNA	614	*Skip*	TSNA	615	*Skip*
TDNN	616	*If* $(AC) \wedge (E) \neq 0$: *skip*	TSNN	617	*If* $(AC) \wedge (E)_S \neq 0$: *skip*
TDZ	630	$(AC) \wedge \sim (E) \rightarrow (AC)$	TSZ	631	$(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDZE	632	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \wedge \sim (E) \rightarrow (AC)$	TSZE	633	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDZA	634	$(AC) \wedge \sim (E) \rightarrow (AC)$ *skip*	TSZA	635	$(AC) \wedge \sim (E)_S \rightarrow (AC)$ *skip*
TDZN	636	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \wedge \sim (E) \rightarrow (AC)$	TSZN	637	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDC	650	$(AC) \veebar (E) \rightarrow (AC)$	TSC	651	$(AC) \veebar (E)_S \rightarrow (AC)$
TDCE	652	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \veebar (E) \rightarrow (AC)$	TSCE	653	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \veebar (E)_S \rightarrow (AC)$
TDCA	654	$(AC) \veebar (E) \rightarrow (AC)$ *skip*	TSCA	655	$(AC) \veebar (E)_S \rightarrow (AC)$ *skip*
TDCN	656	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \veebar (E) \rightarrow (AC)$	TSCN	657	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \veebar (E)_S \rightarrow (AC)$
TDO	670	$(AC) \vee (E) \rightarrow (AC)$	TSO	671	$(AC) \vee (E)_S \rightarrow (AC)$
TDOE	672	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \vee (E) \rightarrow (AC)$	TSOE	673	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \vee (E)_S \rightarrow (AC)$
TDOA	674	$(AC) \vee (E) \rightarrow (AC)$ *skip*	TSOA	675	$(AC) \vee (E)_S \rightarrow (AC)$ *skip*
TDON	676	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \vee (E) \rightarrow (AC)$	TSON	677	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \vee (E)_S \rightarrow (AC)$

ASCII CODE



NOTE: Dark background indicates a character for which a 1 must be punched in channel 8 for even parity.

10-0931

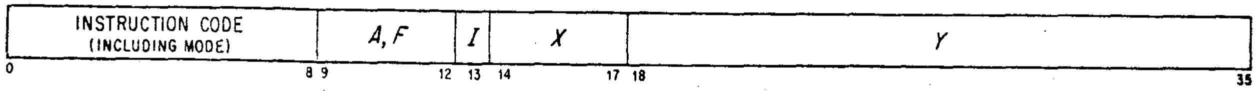
Alternate Graphics

043	£ instead of # on some (non-DEC) units
047	' instead of ′ on early units
136	↑ instead of ^ on early units
137	← instead of _ on early units
174	Nonprinting instead of | on early units
175	Nonprinting ALT MODE instead of } on early units
176	Nonprinting ESC instead of ~ on early units

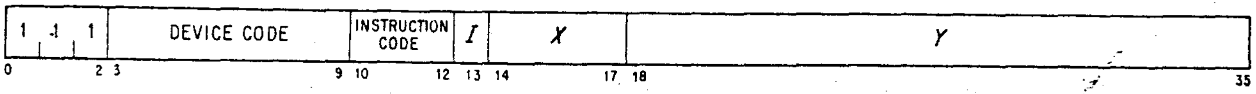
Usually a printer that lacks the lower case character set (bottom line in the chart) prints the corresponding upper case character upon receipt of a lower case code.

PDP-10 WORD FORMATS

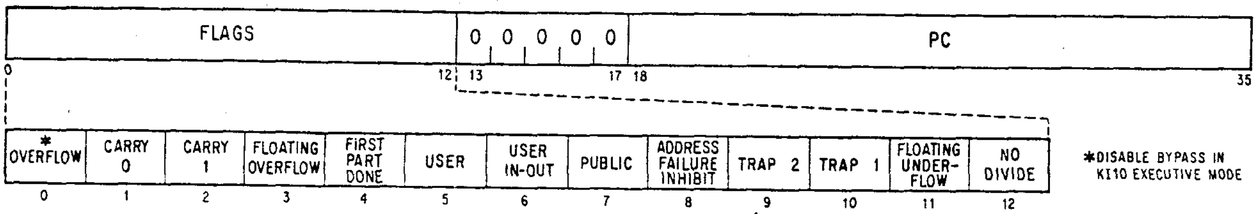
BASIC INSTRUCTIONS



IN-OUT INSTRUCTIONS

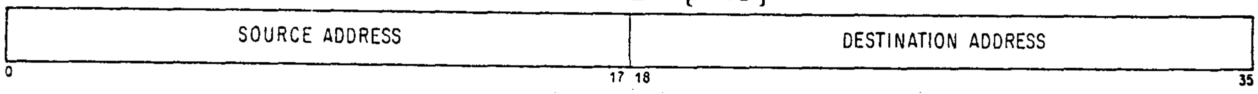


PC WORD

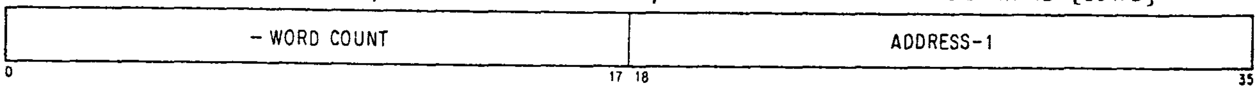


*DISABLE BYPASS IN KI10 EXECUTIVE MODE

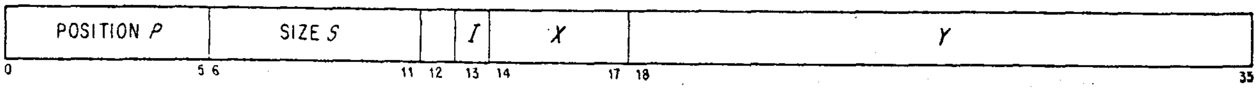
BLT POINTER {XWD}



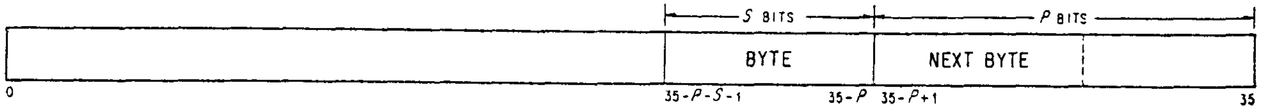
BLKI / BLKO POINTER, PUSHDOWN POINTER, DATA CHANNEL CONTROL WORD {IOWD}



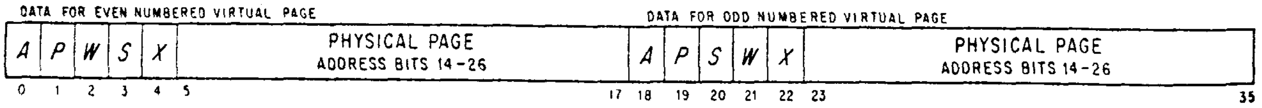
BYTE POINTER



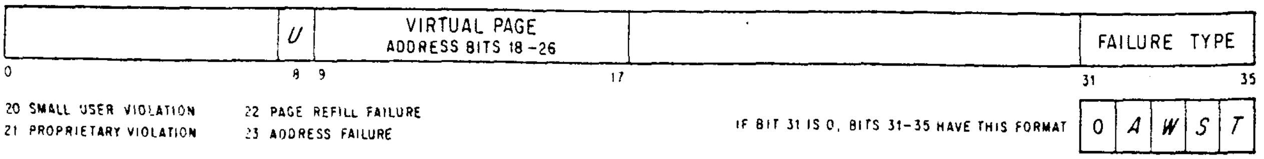
BYTE STORAGE



PAGE MAP WORD



PAGE FAIL WORD



FIXED POINT OPERANDS

SIGN 0+ 1-	BINARY NUMBER (TWOS COMPLEMENT)
0	1 35

LOW ORDER WORD IN DOUBLE LENGTH FIXED POINT OPERANDS

0	LOW ORDER HALF OF BINARY NUMBER (TWOS COMPLEMENT)
0	1 35

FLOATING POINT OPERANDS

SIGN 0+ 1-	EXCESS 128 EXPONENT (ONES COMPLEMENT)	FRACTION (TWOS COMPLEMENT)
0	1 8	9 35

LOW ORDER WORD IN SOFTWARE DOUBLE LENGTH FLOATING POINT OPERANDS

0	EXCESS 128 EXPONENT-27 IN POSITIVE FORM	LOW ORDER HALF OF FRACTION (TWOS COMPLEMENT)
0	1 8	9 35

LOW ORDER WORD IN HARDWARE DOUBLE LENGTH FLOATING POINT OPERANDS

0	LOW ORDER EXTENSION OF FRACTION (TWOS COMPLEMENT)
0	1 35

APPENDIX B
PROCESSOR CONTROL PANELS

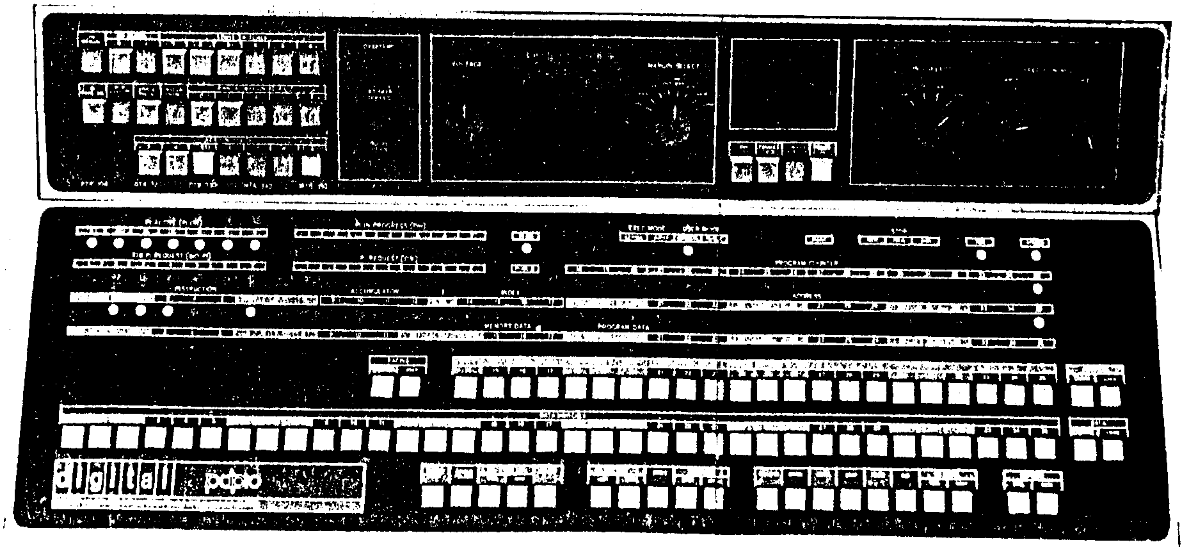


Figure B-1 Console Operator and Maintenance Panels

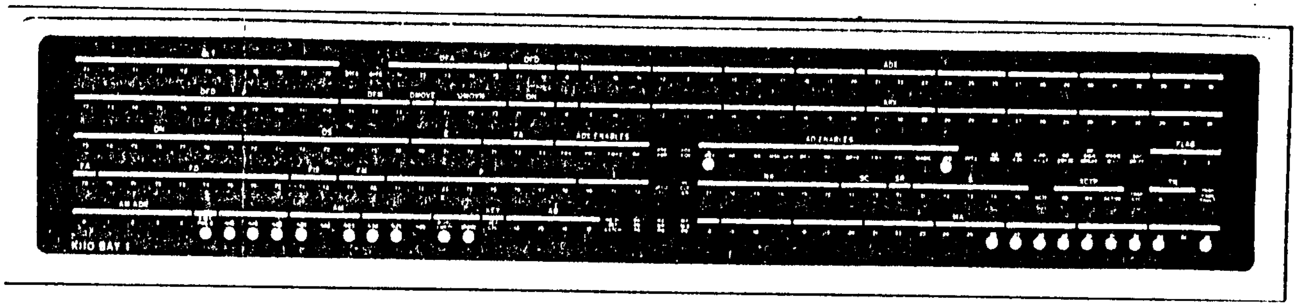


Figure B-2 Bay 1 Indicator Panel

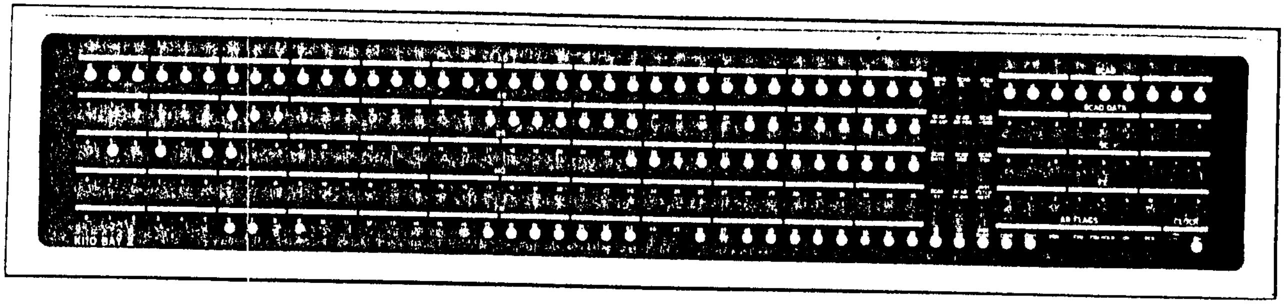


Figure B-3 Bay 2 Indicator Panel

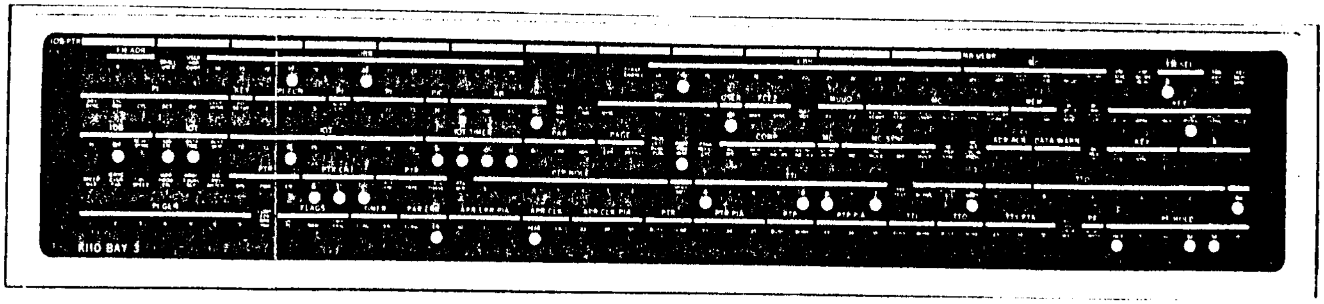


Figure B-4 Bay 3 Indicator Panel

READER'S COMMENTS

KI10 MAINTENANCE MANUAL
A-MN-KI10-0-MAN1

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 ______

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