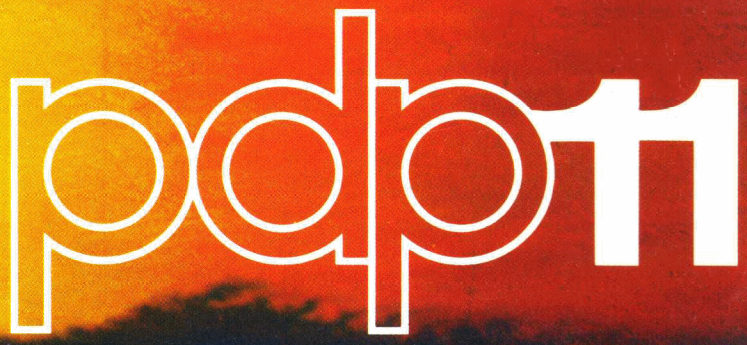
KB11-A
central processor unit
maintenance manual



digital

DEC-11-HKBAA-B-D

KB11-A
central processor unit
maintenance manual

digital equipment corporation · maynard. massachusetts

1st Edition, April 1972
2nd Printing (Rev), July 1972
3rd Printing, October 1972
4th Printing, February 1973
5th Printing, July 1973
6th Printing (Rev), December 1973
7th Printing, May 1974
8th Printing, September 1974

Printed in U.S.A.

CONTENTS

CONTENTS (Cont)

Page

CONTENTS (Cont)

CONTENTS (Cont)

Page

ILLUSTRATIONS

ILLUSTRATIONS (Cont)

TABLES

TABLES (Cont)

INTRODUCTION

This manual describes the KB11-A Central Processor Unit, which is the basic component of the PDP-11/45 Programmed Data Processor System. The purpose of this manual is to:

a. provide an overall understanding of how the KB11-A functions in the PDP-11/45 System.

b. describe how the KB11-A logic works in sufficient detail to enable maintenance personnel to perform on-site troubleshooting and repair.

The principal changes in the printing (5th) of this manual are:

- a discussion of instruction timing has been added (Paragraph 2.3).
- Paragraphs 7.7.2.5 and 7.7.3.2 (Bus Pause), 7.7.7 (Parity Error Logic), and 7.7.13.1 (Power Down Sequence) have been revised to reflect the latest ECOs.

Chapter 1 introduces the purpose and use of the KB11-A and describes how the processor interfaces with the other components and options in the PDP-11/45 System.

Chapter 2 summarizes the KB11-A address modes and instruction set. This summarized information is presented in Instruction Set Processor (ISP) notation. Complete descriptions and examples are provided in the *PDP-11/45 Processor Handbook.*

Chapter 3 describes console control and indicator functions and basic operating procedure. This information is presented to enable maintenance personnel to perform maintenance tests, using the console switches and indicators, and to load and run the PDP-11/45 diagnostic programs.

Chapter 4 introduces the KB11-A principles of operation to familiarize maintenance personnel with the processor features and characteristics.

Chapter 5 provides a block diagram description of the KB11-A architecture, including the data paths and the internal control structure. The notations on the block diagrams included in this chapter provide a key index to the detailed logic schematics in the KB11-A engineering drawing set.

Chapter 6 presents an analysis of the KB11-A flow diagrams. The flow diagrams, which are included in the KB11-A engineering drawing set, provide an essential understanding of how the KB11-A executes instructions and hardware subroutines. Chapter 6 includes an example that traces the execution of a single instruction completely through the flow diagrams.

Chapter 7 contains a detailed logic description of each of the modules in the KB11-A Central Processor Unit (CPU). Simplified diagrams of complex logic and sample timing diagrams are included. However, the descriptions relate directly to the block schematics for each module, which are part of the KB11-A engineering drawing set. These block schematics are referenced throughout the chapter in a short-form notation. For example, the overall

drawing number for the 8-sheet block schematic of the M8100 DAP module is D-CS-M8100-0-01. In short-form notation, these sheets are referenced alphabetically as drawings DAPA through DAPJ. This short-form notation is also used as a prefix for each signal mnemonic, to indicate which block schematic shows the logic that asserts each signal.

This manual is to be used with the following PDP-11/45 System manuals and related publications:

PDP-11/45 System Maintenance Manual	DEC-11-H45B-D
MS11 Semiconductor Memory Systems Maintenance Manual	DEC-11-HMSB-D
FP11 Floating-Point Processor Maintenance Manual	DEC-11-HFPA-D
KT11-C Memory Management Unit Maintenance Manual	DEC-11-HKTB-D
PDP-11/45 Processor Handbook	DEC, 1973
PDP-11 Peripheral Handbook	DEC, 1973–74

CHAPTER 1
GENERAL DESCRIPTION

This chapter describes how the KB11-A Central Processor Unit interfaces with other components and options in the PDP-11/45 System. It also provides a brief functional description of the KB11-A. Additional descriptions of the KB11-A structure and functions are provided in succeeding chapters.

1.1 SYSTEM DESCRIPTION

A PDP-11/45 System block diagram is shown in Figure 1-1. The system includes the following PDP-11/45 components, options, and related peripheral devices:

a. the KB11-A Central Processor Unit
b. Unibus A and B, connected to core memory, input/output (I/O) devices, and mass storage peripherals
c. an MS11 Semiconductor Memory System
d. an FP11 Floating-Point Processor
e. a KT11-C Memory Management Unit option (or the SJB module)

In addition to these components, the system may include additional Unibus-connected components, including other processors.

1.1.1 The Basic System

A basic PDP-11/45 System is composed of the KB11-A Central Processor Unit, core memory, and I/O devices, connected by Unibus A. Such a system can perform virtually all operations that can be performed by any PDP-11/45 System configuration. Information enters and leaves the system through the peripheral I/O devices (and the KB11-A console). As the central processor for the system, the KB11-A fetches instructions from memory and executes the instructions.

Many of the instructions specify operations to be performed on data, which can be data that is stored in the memory, in the processor, or data transferred with the peripheral device.

When the processor executes instructions, both the instructions (including any address constants used by the instructions) and the data are transferred on the Unibus, under the control of the processor. The processor can also respond to special conditions that can occur at any time (i.e., asynchronously). These conditions can be internal conditions, such as power failure, bus errors, or stack overflow; or they can be external conditions, which are indicated to the processor by interrupt operations initiated by the peripheral devices. The processor responds to these asynchronous conditions by performing a series of data transfers which change the processor's operating context. In other words, the processor may execute a different program (or series of instructions) for each type of asynchronous event, and the processor can save the status of one program for later resumption while running another program.

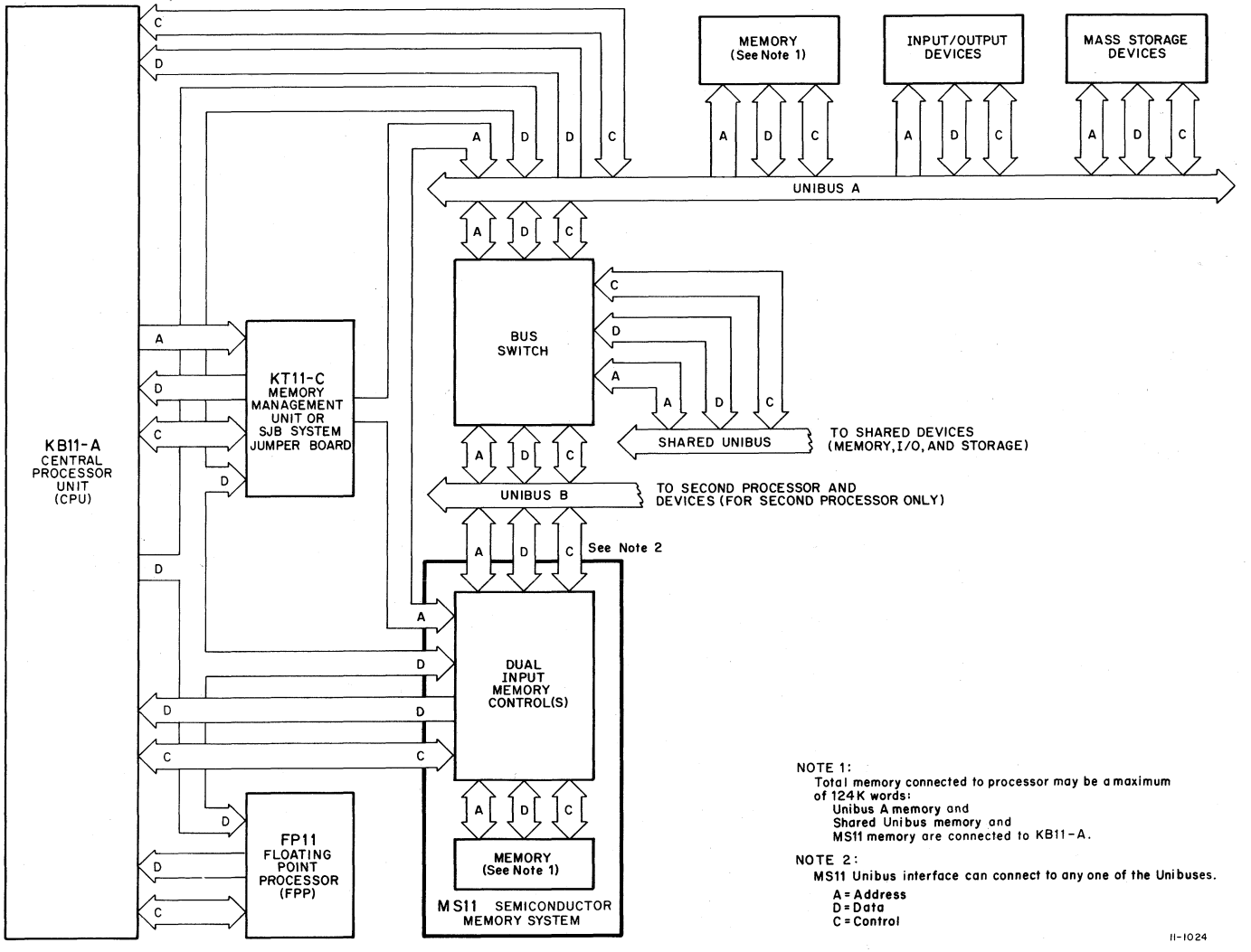


Figure 1-1 System Block Diagram

The system jumper board provides a simple, invariable mapping between the 16-bit addresses used by the KB11-A processor and the 18-bit addresses used on the Unibus. The address mapping is dependent on the three most-significant of the 16 bits in the KB11-A processor address; if these bits are all 1s, the two most-significant bits of the Unibus address are forced to be 1s; otherwise, the two most-significant bits of the 18-bit Unibus address are forced to be 0s.

1.1.2 A Faster Basic System

The data processing capacity of a PDP-11/45 can be increased, for many applications, by increasing either the speed of the memory or the speed at which data operations are performed.

An MS11 Semiconductor Memory System increases the memory speed in two ways:

a. The access time of the memory is much less (typically 300 ns or less) than the access time of the Unibus memories (typically 500 ns or more).

b. The MS11 is connected to the KB11-A processor by a Fastbus, which provides faster transfer times than the Unibus.

The FP11 Floating-Point Processor provides faster data manipulation in two ways:

a. Floating-point arithmetic operations can be performed at hardware speed, without the fetching and interpretation of sequences of instructions (i.e., the execution of a subroutine).

b. Other instructions can be executed in parallel with a floating-point instruction, because the KB11-A processor is free to fetch and execute other instructions while the FP11 processor completes a floating-point instruction.

Figure 1-1 illustrates a PDP-11/45 System that includes an MS11 Semiconductor Memory System, and an FP11 Floating-Point Processor. The KB11-A processor performs all data transfers among parts of the system. All address information for transfers between the processor and memory, or between the processor and the Unibus peripherals, is provided by the SJB system-jumper board. Transfers between the KB11-A processor (CPU) and the FP11 processor (FPP) do not require address information; instead, the processors use control signals that specify the type of information to be transferred, and that also control the timing and direction of the transfer.

1.1.3 A Virtual Machine System

The PDP-11/45 computer system is particularly well-suited to a type of operation in which the computer system provides a "virtual machine" for each user program. In the virtual machine, the user program operates in isolation from all other programs; the computer system provides many high-level services such as device-independent I/O, memory management, program scheduling, and protection of the system from the user program. Many of the high-level functions are provided by system programs; the KB11-A processor can execute a variety of trap instructions used for communication between these system programs and the user program. The processor also has special operating modes for user programs, in which certain processor operations are prohibited to protect the system from improper use of these operations.

One of the major functions of the virtual machine is memory management. This can take two forms:

a. The management of a scarce resource; programs larger than the available memory can be run by loading each part of the program as it is needed.

b. Control of a resource of increased size; although the KB11-A uses only 16 address bits (and can thus address only 2^{16} locations), other system components use 18 address bits (i.e., there can be 2^{18} locations, or 4 times as many as the processor can address directly), so the processor needs some means of specifying how the 16-bit addresses are to be mapped into the 18-bit addresses.

The KT11-C Memory Management Unit is an option that replaces the SJB System Jumper Board to provide both forms of memory management. By allowing a virtual address space to be mapped partly into the physical address space, and partly (through non-resident traps) on secondary storage devices, the KT11-C Memory Management Unit enables the KB11-A processor, with appropriate system software, to simulate a much larger available space. This requires the use of a mass storage device on the Unibus, as shown in Figure 1-1.

The KT11-C can also be used to map the address space of the processor onto the larger Unibus address space by converting the 16-bit addresses to 18-bit addresses. The mapping can be changed dynamically, at any time, so that a program can access the entire Unibus address space a part at a time.

The memory management unit also provides some of the system protection against user programs. The KT11-C can map different programs into different parts of the physical address space, providing a different context for each program; this is done so that both user mode and system programs can be in the memory at the same time, without conflict. In addition, the KT11-C Memory Management Unit provides various types of access protection to prevent a user from inadvertently altering or destroying valuable data.

When the memory management system is used to control scarce resources, large blocks of data must be transferred between a mass storage device and the memory. To prevent these transfers from using too large a part of the processing time, the mass storage device is allowed to conduct the transfers without processor intervention, using the Unibus. The KB11-A processor arbitrates the use of the Unibus, so that the data transfers between the mass storage device and the memory are interleaved with the data transfers between the processor and memory. The mass storage device uses interrupts to inform the processor when the transfer is completed or when an error occurs.

The control section of the MS11 Semiconductor Memory System has two data transfer ports. One is used by the KB11-A processor, and the other is connected to Unibus B, so that a mass storage device can transfer directly to the MS11 memory. The PDP-11/45 System makes use of the two data paths to reduce the interference between the mass storage device and the processor on the Unibus. This is done as follows:

a. When the mass storage device is using the Unibus to transfer to, or from, Unibus memory, the processor can transfer to, or from, MS11 memory without conflict.

b. There can be up to two memory controllers in the MS11 memory system; when the mass storage device is operating with one controller, the processor can operate with the other.

These considerations can greatly reduce the system overhead by allowing both the processor and the mass storage device to operate at maximum speed most of the time. In expanded systems, bus switches can be used to further improve the capacity for simultaneous operation.

1.1.4 Expanded Systems

The elements that distinguish a PDP-11/45 System have all been introduced in the three preceding paragraphs. These elements are the following:

a. the KB11-A Central Processor Unit
b. the MS11 Semiconductor Memory System
c. the FP11 Floating-Point Processor
d. the KT11-C Memory Management Unit

All PDP-11/45 Systems include the KB11-A; however, all other components are optional.

In addition to these components, a PDP-11/45 System can include any Unibus device or memory, and can be structured to have more than one Unibus.

A bus switch can be used to separate the Unibus into two parts. Normally, the system operates with the two parts connected; however, after a data transfer operation has been started between the mass storage device and the MS11 memory, the bus switch is opened so that the KB11-A processor can transfer data to, or from the Unibus memory without conflicting with the transfers of the mass storage device. This improves the parallel operation described in Paragraph 1.1.3.

The system shown in Figure 1-1 uses a more-complex bus switch to connect the KB11-A processor and a second processor to a third shared Unibus. In this multiprocessor configuration, both processors can access the MS11 Semiconductor Memory System, and both processors can control devices on the shared Unibus. One application for this structure is to have the second processor control the main mass storage device in the system, performing optimization programs and error recovery. This relieves the KB11-A processor of much of the burden of the memory management and program swapping functions, thereby allowing the KB11-A to proceed with data processing at maximum efficiency.

1.2 FUNCTIONAL DESCRIPTION

The basic functions performed by the KB11-A processor include the following:

a. manipulating data
b. transferring data among other devices
c. fetching and executing instructions
d. responding to asynchronous conditions

1.2.1 Data Manipulation

Figure 1-2 is a functional block diagram which illustrates the structure of the KB11-A processor data paths. The data manipulation elements can perform arithmetic, logic, and shift operations on data from various sources, and the result of each data manipulation can be distributed to various destinations. The primary area for the storage of data in the processor is the general registers, which are used to store data and address constants. Another register that is connected to the data manipulation elements is the bus register (BR); this register is a central point in the data paths because all data that enters the processor from other devices enters through the BR, and all data that is transmitted from the processor to other devices is transmitted from the BR.

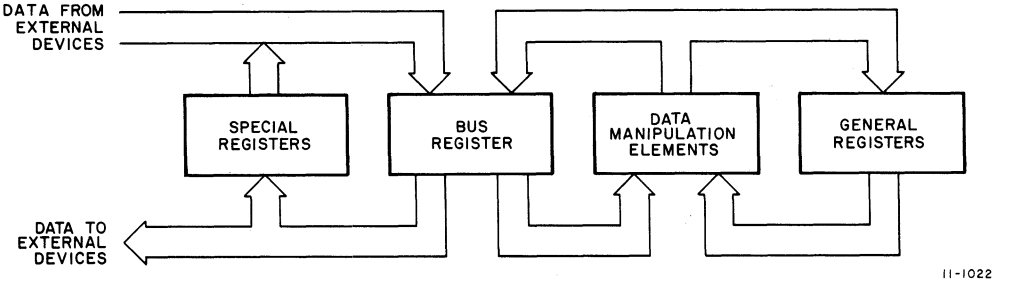


Figure 1-2 Data Paths, Functional Block Diagram

1.2.2 Transferring Data

Data transfers between devices in a PDP-11/45 System take place on the Unibus or for certain devices, connect directly to the processor (the KT11-C, the FP11, and the MS11 system) on an internal bus or Fastbus. These

buses are part of the processor. Some Unibus data transfers occur without processor intervention; they are performed by devices that can become Unibus master and can directly provide address and control information. For most simpler devices (especially memories) and for devices on the Fastbus, the KB11-A processor controls the data transfers and provides address and control information.

The KB11-A processor provides address information from the general registers; the control signals for data transfer are provided by the control section of the processor. All data that enters and leaves the processor does so through the BR register. Data transfers can be combined with data manipulation; most PDP-11 instructions provide the ability to operate directly on data from other devices (such as memory), and to return the data to the devices in the same instruction.

1.2.3 Handling Instructions

The users specify the data manipulation and transfer operations that the KB11-A processor is to perform by a series of instructions. The instructions are stored in the memory of the PDP-11/45 System and can be transferred as data. Each instruction, in turn, must be transferred from the memory to the processor, where it is decoded and used to guide the processor in executing a series of operations.

Figure 1-3 illustrates the control section of the KB11-A processor on a functional block diagram level. The control logic of the processor produces control signals which cause various operations in the data paths of the processor, and are external to the processor on the Unibus and Fastbus. The states of these control signals are selected by various inputs. The inputs that are most important in determining the sequence of operations executed for any instruction are the inputs from the data paths and from the instruction register (IR).

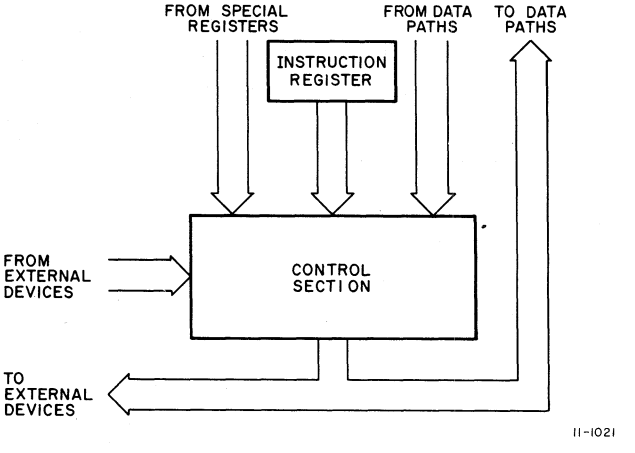


Figure 1-3 Control Section, Functional Block Diagram

The data paths' inputs are selected information about the data that is currently being processed. These inputs are used as conditions to determine which variation of the instruction sequence should be used. The IR register is loaded from the same inputs as the BR; however, the outputs of the IR are used only for instruction decoding, and the IR is loaded only when the data has been fetched specifically as an instruction (i.e., the contents of the IR are seldom changed when the BR is loaded).

1.2.4 Handling Asynchronous Conditions

The KB11-A processor responds to various types of asynchronous conditions. In general, the response of the processor is to store the current operating context (the processor status and the address of the next instruction of the current program, as well as the operating mode and register set selection), load a new context, and then begin executing a service program for the recognized condition. The service program begins at an address specified in the new context information.

This response to asynchronous conditions is controlled by a sequence of signals generated in the control section of the processor. The control section produces this sequence when certain inputs are recognized, provided that the processor is in a state where the response is allowed (many asynchronous conditions are ignored until the processor has completed an instruction). The inputs to the control section that are important for recognizing these conditions are the inputs from devices external to the processor and the inputs from the special registers. The special registers are treated by the KB11-A processor as external to the processor; they are loaded from the BR and read into the BR. These registers include the stack limit and programmed interrupt request registers which contribute signals used by the control section of the processor to determine what asynchronous conditions exist. The processor status register is also included in the special registers; it is used by the control section to determine the asynchronous conditions to which the processor should respond.

CHAPTER 2
ADDRESS MODES AND INSTRUCTION SET

This chapter summarizes the KB11-A Central Processor Unit address modes and instruction set. Its purpose is to define the KB11-A and provide tabular, quick-reference information. A complete description of KB11-A address modes and instructions, with additional details and examples, is provided in the *PDP-11/45 Processor Handbook*.

Instruction Set Processor (ISP) notation is used to define the processor operations for each address mode and instruction. Table 2-1 defines the modified ISP symbology used in this chapter. Appendix A of the *PDP-11/45 Processor Handbook* provides a more detailed description of ISP notation.

Table 2-1
ISP Symbology

Symbol	Definition
⟨ ⟩	Defines the limits of an expression, such as word length ⟨15:0⟩.
[ ]	Defines the limits of a memory declaration; Mw [SP] specifies the address of the stack pointer in memory.
←	The expression to the left of this symbol is replaced by the expression to the right of this symbol, Z ← 1 indicates the Z bit is set, PC ← PC + 2 indicates the program counter register (PC) is incremented by 2.
cat	Indicates concatenation; registers to the left and right of this expression are considered to be 1.
equiv	Designates that expressions to the left and right are equivalent.
&	Logical AND
OR	Logical inclusive-OR
~	Negate
XOR	Logical exclusive-OR
'	Indicates that a reference to the expression with which this symbol is used may cause side effects, e.g., registers may be changed as a result of the operation.
;	Used as a delimiter
;next	A sequential delimiter, the operation to the left must occur before the operation to the right.
m	Designates an address mode; address mode 1 is indicated by m = 1.

(continued on next page)

Table 2-1 (Cont)
ISP Symbology

Symbol	Definition
rg	General register 7 (program counter)
ai	Auto-increment; by 2 for word instructions, and by 1 for byte instructions.
r	Indicates a result; used many times with limit symbols as an intermediate register (r ⟨15:0⟩).
+	Addition; expression to the left is added to expression to the right.
-	Subtraction; expression to the right is subtracted from expression to the left.
X	Multiply; expression to the left is multiplied by expression to the right.
/	Divide; expression to the left is divided by the expression to the right.
sign-extend	The sign bit of a byte, bit 7, is extended through bits 8 to 15.
Mw	Memory word declaration; the address in brackets points to the memory location.
nw'	Indicates next word, as pointed to by the PC with side effects ('). The word is at the next sequential PC address, or the word pointed to by the next word (deferred addressing).
R [dr]	Indicates that a register (R) address as a memory declaration is that of a device register.
D	Destination
Db	Byte destination
S	Source
Sb	Byte source

2.1 ADDRESS MODES

The instruction set of the PDP-11/45 implements the flexibility of the general purpose registers through the address modes. Table 2-2 lists all the address modes, including the program counter (PC) register address modes. These address modes, along with the general purpose register designation, determine the instructions' operands (source and/or destination) and form part of the 16-bit instruction format (Figure 2-1).

Table 2-2
Address Modes

Mode	Designation	Symbolic	ISP	Description
			General Purpose Register Addressing	
0	register	R	if (m=0) then Rr ⟨w1:0⟩;	The register (R, Rr) is the operand.
1	register deferred	@R or (R)	if (m=1) then M[Rr];	Defer to operand through register (R, Rr) as address.
2	auto-increment	(R)+	if (m=2) and (rg≠7) then (M[Rr]; next Rr ← Rr + ai);	Defer to operand through register (R, Rr) as address, then increment.
3	auto-increment deferred	@(R)+	if (m=3) and (rg≠7) then (M[Mw[Rr]]; next Rr ← Rr + 2;	Defer to operand through (R), Mw [Rr] as address, then increment register (R, Rr).
4	auto-decrement	-(R)	if (m=4) then (Rr ← Rr - ai); next M[Rr];	Decrement register (R, Rr), then defer to operand through register (R, Rr) as address.
5	auto-decrement deferred	@-(R)	if (m=5) then (Rr ← Rr - ai; next M[Mw[Rr]]);	Defer to operand through (R), Mw Rr after decrement of register (R, Rr).
6	indexed	±X(R)	if (m=6) and (rg≠7) then M[nw' + Rr];	Index via register = (R, Rr) by the amount specified in next PC word (X).
7	indexed deferred	@±X(R) or @(R)	if (m=7) and (rg≠7) then M[Mw[nw' + Rr]];	Defer to operand through index of register (R, Rr) specified in next PC word (X) as address.
			PC Register Addressing	
2	immediate	#n	if (m=2) and (rg=7) then nw' ⟨w1:0⟩	Defer to operand through PC value (next word); next word is immediate operand.
3	absolute	@#A	if (m=3) and (rg=7) then M[nw']	Defer via next word (PC address) as address to operand; absolute addressing.
6	relative	A	if (m=6) and (rg=7) then M[nw' + PC];	Relative to PC; uses next word as deferred address of operand.
7	relative deferred	@A	if (m=7) and (rg=7) then M[Mw[nw' + PC]];	Defer relative to PC; uses next word as address of deferred address of the operand.

NOTE: The following symbols are used in this table:

R = Register
X, n, A = next program counter (PC) word (constant)

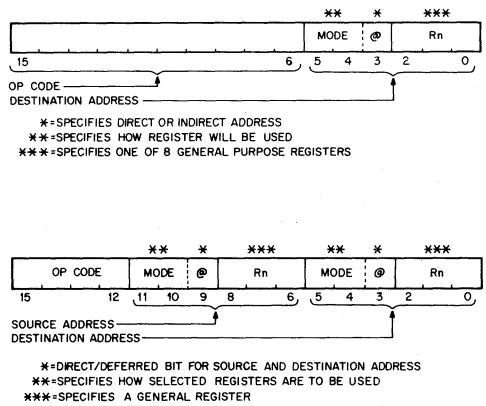


Figure 2-1 Single and Double Operand Address Modes

2.2 KB11-A INSTRUCTIONS

The KB11-A instruction set is divided into the following six groups of instructions:

a. **Double Operand** – Arithmetic, logical, and move instructions are included in this group (Table 2-3).

b. **Register and Operand** – Multiply, divide, and arithmetic shifts that specify a register, and an operand, are included in this group (Table 2-4).

c. **Single Operand** – Shifts, multiple precision instructions, and rotates are in this group (Table 2-5).

d. **Program Control** – This group includes all the instructions that explicitly change the PC and processor status word (PS), such as branches, subroutines, and traps (Table 2-6).

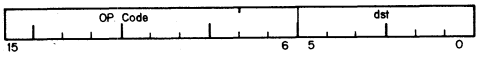
e. **Operate Group** – The processor control instructions such as Halt and Wait are included in this group (Table 2-7).

f. **Condition Code Operators** – This group includes the instructions that clear and set the PSW condition codes (Table 2-8).

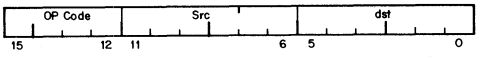
The format of each group of instructions is illustrated in Figure 2-2.

In addition to these instructions, the KB11-A decodes all floating-point instructions that are executed by the FP11-B Floating-Point Processor. The floating-point instruction set is described in the *FP11 Floating-Point Processor Maintenance Manual*, DEC-11-HFPA-D.

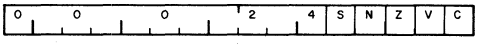
Single Operand Group



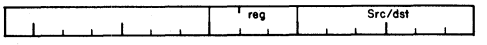
Double Operand Group



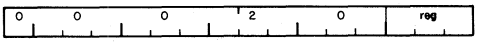
Condition Code Operators



Register Source or Destination



Subroutine Return



Branch

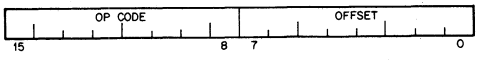


Figure 2-2 Instruction Formats

Table 2-3
Double Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
MOV Move (Src to Dst) 01SSDD	r ← S'; next N ← r ⟨15⟩ if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0), V ← 0; D' ← r	Move source to intermediate register, r. Set N if negative. Set Z if 0. Clear V. Transmit result to destination.
MOVB Move Byte (Src to Dst) 11SSDD	r ← Sb'; next N ← r ⟨7⟩; if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; Db' ← r	Move source to intermediate register, r. Set N if negative. Set Z if 0. Clear V. Transmit result to destination.

CMP
Compare
(Src to Dst)
02SSDD
	r ⟨16:0⟩ ← S' - D'; next
	
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (S ⟨15⟩ = ~ D ⟨15⟩) & (S ⟨15⟩ XOR r ⟨15⟩) then
	(V ← 1 else V ← 0);
	C ← r ⟨16⟩
		Source and destination operands are compared, but unaffected.
		Only condition codes are affected, as follows:
		Set N if r is negative.
		Set Z if r is 0.
		Set V if operands have opposite signs and the sign of the source
		is the same as the result, r.
		Set C if 17th bit is carry.

CMPB
Compare Byte
12SSDD
	r ⟨8:0⟩ ← Sb' - Db'; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (Sb ⟨7⟩ = ~ Db ⟨7⟩ & (Sb ⟨7⟩ XOR r ⟨7⟩) then
	(V ← 1 else V ← 0);
	C ← r ⟨8⟩
		Same as CMP, except operands are bytes.

BIT Bit Test 03SSDD	r ← D' & S'; next N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0	Logical AND of source and destination operands. Set N if negative. Set Z if 0. No overflow.
BITB Bit Test, Byte 13SSDD	r ← Db' & Sb'; next N ← r ⟨7⟩; if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0	Same as BIT, except byte
BIC Bit Clear 04SSDD	r ← D' & ~ S'; next N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; D ← r	AND destination operand with complemented source operand. Set N if negative. Set Z if 0. Clear V and put result in destination address.
BICB Bit Clear, Byte 14SSDD	r ← Db' & ~ Sb'; next N ← r ⟨7⟩; if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0); V ←0; Db ← r	Same as BIC, except byte.

(continued on next page)

Table 2-3 (Cont)
Double Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
BIS Bit Set 05SSDD	r ← D' OR S'; next N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; D ← r	Inclusive OR of source operand and destination operand. Set N if negative. Set Z if 0. Clear V. Put result in destination.
BISB Bit Set, Byte 15SSDD	r ← Db' OR Sb'; next N ← r ⟨7⟩; if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; Db ← r	Same as BIS, except byte.

ADD
Add
06SSDD
	r ⟨16:0⟩ ← S' + D'; next
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (S ⟨15⟩ equiv D ⟨15⟩) & (S ⟨15⟩ XOR r ⟨15⟩) then (V ← 1 else V ← 0);
	C ← r ⟨16⟩;
	D ← r ⟨15:0⟩
		Add source and destination to provide 17-bit sum.
		Set N if negative result.
		Set Z if 0.
		Set V if both operands were same sign and the result is of opposite sign.
		Set C if carry.
		Put result in destination.

SUB
Subtract
16SSDD
	r ⟨16:0⟩ ← D' - S'; next
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (D ⟨15⟩ XOR S ⟨15⟩) & (D ⟨15⟩ XOR r ⟨15⟩) then (V ← 1 else V ← 0);
	C ← r ⟨16⟩;
	D ← r ⟨15:0⟩
		Subtract source operand from destination operand.
		Set N if negative results.
		Set Z if 0.
		Set V if operands had different signs and result is opposite sign from destination.
		Set C if a carry.
		Put result in destination.

Table 2-4
Register and Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description

MUL
Multiply
070RSS
	r ⟨31:0⟩ ← D' × R[sr] ; next
	
	if (r ⟨31:0⟩ = 0) then (Z ← 1 else Z ← 0);
	N ← r ⟨31⟩;
	if (r ⟨31:0⟩ < -2^{15}) OR (r ⟨31:0⟩ ≥ 2^{15}) then
	(C ← 1 else C ← 0);
	V ← 0;
	R[sr] ⟨15:0⟩ ← r ⟨31:16⟩; next
	R[sr OR 1] ⟨15:0⟩ ← r ⟨15:0⟩;
		Multiply contents of source register and destination to form 32-bit product.
		Set Z if product is 0.
		Set N if product is negative.
		Set C if product is more than 16-bit result.
		
		No overflow possible; clear V.
		Store the high-order result in R.
		Store the low-order result in succeeding register if R is even number. Otherwise, store in R.

DIV
Divide
071RSS
	r1 ⟨31:0⟩ ← R[sr] cat R[sr OR 1]/D'; next
	
	r2 ⟨15:0⟩ ← R[sr] cat R[sr OR 1] -(r1 × D);
	next N ← r1 ⟨15⟩;
	if (r1 ⟨31:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (D = 0) then (C ← 1 else C ← 0);
	if (r1 ⟨15⟩ = 0) & (r1 ⟨31:16⟩ ≠ 0)
	OR
	if (r1 ⟨15⟩ = 1) & (r1 ⟨31:16⟩ ≠ -1)
	OR
	if (D = 0) then (V ← 1 else V ← 0);
	R[sr] ← r1 ⟨15:0⟩
	R[sr OR 1] ← r2
		The 32-bit dividend, R, R OR 1, is divided by source operand D. R must be even number.
		Determine the remainder.
		Set N if quotient is negative.
		Set Z if quotient is 0.
		Set C if divide by 0 attempted.
		Set V if divisor is 0, or if the result is too large to be stored as a 16-bit number.
		
		Store quotient in R.
		Store remainder in R OR 1.

ASH
Arithmetic Shift
072RDD
	r ⟨79:0⟩ ← sign-extend (R[sr] ⟨15:0⟩ × 2 ↑
	(D' ⟨5:0⟩ + 32) mod 64); next
	
	R[sr] ⟨15:0⟩ ← r ⟨47:32⟩; next
	if (R[sr] = 0) then (Z ← 1 else Z ← 0);
	if ((R[sr] ⟨15⟩ = 0) & (r ⟨79:48⟩≠ 0) OR
	(R[sr] ⟨15⟩ = 1) & (r ⟨79:48⟩ ≠ -1)) then
	(V ← 1 else V ← 0);
	N ← R[sr] ⟨15⟩;
	if (D ⟨5⟩ = 1) then C ← r ⟨31⟩;
	if (D ⟨5⟩ = 0) & (D ⟨5:0⟩ ≠ 0) then
	C ← r ⟨48⟩;
	if (D ⟨5:0⟩ = 0) then C ← 0
		Contents of R are shifted NN places right or left, where NN equals the six low-order bits of DD.
		NN = -32 to +31.
		Store result in R.
		Set Z if result is 0.
		Set V if sign of register changed during shift.
		
		Set N if result is negative.
		Load C from last bit shifted out of register.

(continued on next page)

Table 2-4 (Cont)
Register and Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
ASHC Arithmetic Shift Combined 073RDD	r ⟨95:0⟩ ← sign-extend (R[sr] cat R[sr OR 1] × 2 ↑ (D' ⟨5:0⟩ + 32) mod 64); next	Contents of R, and R ORed with 1, form a 32-bit word (R = 31:16, ROR 1 = 15:0) that is shifted right or left NN places, specified by six low-order bits of destination operand, DD.
	R[sr] ← r ⟨63:48⟩; next R[sr OR 1] ← r ⟨47:32⟩; next	Store results in R and R OR 1.
	if (R[sr] cat R[sr OR 1] = 0) then (Z ← 1 else Z ← 0);	Set Z if result is 0.
	N ← R[sr] ⟨15⟩;	Set N if result is negative.
	if (r ⟨63⟩ = 0) & (r ⟨95:64⟩ ≠ 0) OR if (r ⟨63⟩ ≠ 0) & (r ⟨95:64⟩ ≠ -1) then (V ← 1 else V ← 0);	Set V if sign bit changes during the shift.
	if (D ⟨5⟩ = 1) then C ← r ⟨31⟩;	Load C with high order if left shift.
	if (D ⟨5⟩ = 0) & (D ⟨5:0⟩ ≠ 0) then C ← r ⟨64⟩;	Load C with low order if right shift.
	if (D ⟨5:0⟩ = 0) then C ← 0	Otherwise, clear C.
XOR Exclusive-OR 074RDD	r ← R[sr] XOR D'; next	The exclusive-OR of the register and the destination operand is stored in the destination address.
	if (r = 0) then (Z ← 1 else Z ← 0);	Set Z if result is 0.
	N ← r ⟨15⟩;	Set N if result is negative.
	V ← 0;	Clear V; no overflow possible.
	R[sr] ← r	
SOB Subtract One and Branch 077R offset	r ← R[sr] - 1; next R[sr] ← r;	Decrement register by 1. If result is not equal to 0, branch.
	if (r ≠ 0) then (PC ← PC - 2 × df ⟨5:0⟩)	Subtract 2 × 6-bit offset from PC to get new PC.

Table 2-5
Single Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
CLR Clear dst 0050DD	D' ← 0; N ← 0; Z ← 1; V ← 0; C ← 0	Clear destination, N, V, and C; set Z.
CLRB Clear Byte dst 1050DD	Db' ← 0; N ← 0; Z ← 1; V ← 0; C ← 0	Clear destination byte.
COM Complement dst 0051DD	r ← ~ D'; next N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then Z ← 1 else Z ← 0); V ← 0; C ← 1; D ← r	Complement destination. Set N if negative. Set Z if 0. Clear V. Set C. Put result in destination.
COMB Complement Byte dst 1051DD	r ← ~ Db'; next N ← r ⟨7⟩; if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; C ← 1; Db ← r	Same as COM, except byte.

INC
Increment dst
0052DD
	r ← D' + 1; next
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨15:0⟩ = 100000_8) then (V ← 1 else V ← 0);
	D ← r
		Result is sum of D plus 1.
		Set N if negative.
		Set Z if 0.
		Set V if result equals 100000_8 (dst was 077777_8).
		Put result in destination.

INCB
Increment
Byte dst
1052DD
	r ← Db' + 1; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨7:0⟩ = 200_8) then (V ← 1 else V ← 0);
	Db ← r
		Same as INC, except byte.
		
		Set V if result equals 200_8 (dst byte was 177_8).

DEC
Decrement
dst
0053DD
	r ← D' − 1; next
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨15:0⟩ = 77777_8) then (V ← 1 else V ← 0);
	D ← r
		Result is destination operand minus 1.
		Set N if negative.
		Set Z if 0.
		Set V if result equals 77777_8 (dst was 100000_8).
		Put result in destination.

DECB
Decrement
Byte dst
1053DD
	r ← Db' − 1; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨7:0⟩ = 177_8) then (V ← 1 else V ← 0);
	Db ← r
		Same as DEC, except byte.
		
		Set V if result is 177_8 (dst byte was 000_8).

(continued on next page)

Table 2-5 (Cont)
Single Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description

NEG
Negate dst
0054DD
	r ← -D'; next
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨15:0⟩ = 100000_8) then (V ← 1 else V ← 0);
	if (r ⟨15:0⟩ = 0) then (C ← 0 else C ← 1);
	D ← r
		Negate D by 2's complement.
		Set N if negative result.
		Set Z if 0.
		Set V if destination operand was 100000_8.
		Clear C if result is 0, otherwise set C.
		Put result in destination.

NEGB
Negate Byte
1054DD
	r ← -Db'; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨7:0⟩ = 200_8) then (V ← 1 else V ← 0);
	if (r ⟨7:0⟩ = 0) then (C ← 0 else C ← 1);
	Db ← r
		Same as NEG, except byte.

ADC
Add Carry
0055DD
	r ← D' + C; next
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨15:0⟩ = 100000_8) & (C = 1) then (V ← 1 else V ← 0); next
	if (r ⟨15:0⟩ = 0) & (C = 1) then (C ← 1 else C ← 0);
	D ← r
		Add the C bit to the destination.
		Set N if negative.
		Set Z if 0.
		Set V if destination was 077777_8 and C was 1.
		Set C if destination was 177777_8 and C was 1.

ADCB
Add Carry Byte
1055DD
	r ← Db' + C; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨7:0⟩ = 200_8) & (C = 1) then (V ← 1 else V ← 0); next
	if (r ⟨7:0⟩ = 0) & (C = 1) then (C ← 1 else C ← 0);
	Db ← r
		Same as ADC, except byte.

SBC
Subtract Carry
0056DD
	r ← D' - C; next
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨15:0⟩ = 100000_8) then (V ← 1 else V ← 0);
	if (r ⟨15:0⟩ = 0) & (C = 1) then (C ← 0 else C ← 1);
	D ← r
		Subtract C bit from contents of destination.
		Set N if negative.
		Set Z if 0.
		Set V if result is 100000_8.
		Clear C if result is 0 and C = 1.
		Put result in destination.

SBCB
Subtract Carry Byte
1056DD
	r ← Db' - C; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0);
	if (r ⟨7:0⟩ = 200_8) then (V ← 1 else V ← 0);
	if (r ⟨7:0⟩ = 0) & (C = 1) then (C ← 0 else C ← 1);
	Db ← r
		Same as SBC, except byte.

(continued on next page)

Table 2-5 (Cont)
Single Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
TST Test 0057DD	r ← D' - 0; next N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; C ← 0	Sets N and Z condition codes according to contents of destination address.
TSTB Test Byte 1057DD	r ← Db' - 0; next N ← r ⟨7⟩; if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; C ← 0	Same as TST, except byte.

ROR
Rotate Right
0060DD
	r ⟨16:0⟩ ← D' ⟨0⟩ cat C cat D' ⟨15:1⟩; next
	
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	C cat D ⟨15:0⟩ ← r ⟨16:0⟩; next
	if (N XOR C) then (V ← 1 else V ← 0)
		17-bit intermediate result is C and contents of destination rotated right one place.
		Set N if high order bit is set.
		Set Z if result is 0.
		Put 17-bit result into C bit and destination.
		Load V with exclusive-OR of N and C (after rotation is complete).

RORB
Rotate Right
Byte
1060DD
	r ⟨8:0⟩ ← Db' ⟨0⟩ cat C cat Db' ⟨7:1⟩; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0);
	C cat Db ← r ⟨8:0⟩; next
	if (N XOR C) then (V ← 1 else V ← 0)
		Same as ROR, except byte.

ROL
Rotate Left
0061DD
	r ⟨16:0⟩ ← D' ⟨15:0⟩ cat C; next
	
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	C cat D ← r ⟨16:0⟩; next
	
	if (N XOR C) then (V ← 1 else V ← 0)
		17-bit result is C and contents of destination rotated left one bit.
		Set N if result is negative.
		Set Z if result is 0.
		Put result into C and D. Bit 15 into C bit and previous C bit into bit 0.
		Load V with exclusive-OR of N and C after rotation is complete.

ROLB
Rotate Left
Byte
1061DD
	r ⟨8:0⟩ ← Db' ⟨7:0⟩ cat C; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0);
	C cat Db ← r ⟨8:0⟩; next
	if (N XOR C) then (V ← 1 else V ← 0)
		Same as ROL, except byte.

ASR
Arithmetic
Shift Right
0062DD
	r ← D'/2; next
	C ← D ⟨0⟩;
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0 then (Z ← 1 else Z ← 0); next
	if (N XOR C) then (V ← 1 else V ← 0);
	D ← r
		Contents of destination shifted right one place (÷ 2).
		Least-significant bit loaded into C.
		Set N if result negative.
		Set Z if result 0.
		Load V with exclusive-OR of N and C after shift is complete.
		Put result into destination.

(continued on next page)

Table 2-5 (Cont)
Single Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description

ASRB
Arithmetic
Shift Right
Byte
1062DD
	r ← Db'/2; next
	C ← Db ⟨0⟩;
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0); next
	if (N XOR C) then (V ← 1 else V ← 0);
	Db ← r
		Same as ASR except byte.

ASL
Arithmetic
Shift Left
0063DD
	r ← D' ⟨15⟩ cat D' ⟨13:0⟩ cat 0; next
	
	C ← D ⟨14⟩; next
	N ← r ⟨15⟩;
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); next
	if (N XOR C) then (V ← 1 else V ← 0);
	D ← r
		Shifts contents of destination left one place, but sign bit remains in most significant place.
		Bit 14 loaded into C.
		Set N if result negative.
		Set Z if result 0.
		Load V with exclusive-OR of N and C after shift completed.
		Put result in destination.

ASLB
Arithmetic
Shift Left
Byte
1063DD
	r ← Db' ⟨7⟩ cat Db' ⟨5:0⟩ cat 0; next
	C ← Db ⟨6⟩; next
	N ← r ⟨7⟩;
	if (r ⟨7:0⟩ = 0) then (Z ← 1 else Z ← 0); next
	if (N XOR C) then (V ← 1 else V ← 0);
	Db ← r
		Same as ASL, except byte.

MARK Mark 0064nn	SP ← PC + 2 + (2 X df ⟨5:0⟩); next PC ← R[5] ; next R[5] ← Mw [SP] ; SP ← SP + 2	Adjusts stack pointer by the number of words indicated in the low 6 bits of the instruction (2 X nn locations). Puts old PC (R5) into PC. Contents of old R5 popped into R5.
MFPI Move From Previous Instruction Space 0065DD	r ← D'; next SP ← SP - 2; N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; Mw [SP] ← r	Get destination operand from previous I space. Push stack. Set N if negative. Set Z if 0. Clear V. Put operand into current address space.
MFPD Move From Previous Data Space 1065DD	r ← D'; next SP ← SP - 2; N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; Mw [SP] ← r	Get destination operand from previous D space. Push stack. Set N if negative. Set Z if 0. Clear V. Put operand into current address space.
MTPI Move To Previous Instruction Space 0066DD	r ← Mw [SP] ; SP ← SP + 2; next N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; D' ← r	Get data from current stack. Pop stack. Set N if negative. Set Z if 0. Clear V. Move to previous I space destination.

(continued on next page)

Table 2-5 (Cont)
Single Operand Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
MTPD Move To Previous Data Space 1066DD	r ← Mw [SP] ; SP ← SP + 2; next N ← r ⟨15⟩; if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0); V ← 0; D' ← r	Get data from current stack. Pop stack. Set N if negative. Set Z if 0. Clear V. Move to previous D space destination.

SXT
Sign Extend
destination
0067DD
	if (N = 1) then (r ⟨15:0⟩ ← -1 else r ⟨15:0⟩ ← 0);
	next
	if (r ⟨15:0⟩ = 0) then (Z ← 1 else Z ← 0);
	D' ← r
		If the N bit is set, then -1 is placed in the destination operand.
		Otherwise, 0 is placed in the destination operand.
		Set Z if result is 0.

Table 2-6
Program Control Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
BR Branch Unconditional 0004 loc	PC ← PC + sign-extend (instr ⟨7:0⟩ X 2)	Always branch. PC changed as follows: Eight least-significant bits of instruction are multiplied times 2 and added to PC with sign extended.
BNE Branch Not Equal 0010 loc	if (Z = 0) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if Z is 0.
BEQ Branch on Equal 0014 loc	if (Z = 1) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if Z is 1.
BGE Branch if Greater than or Equal (zero) 0020 loc	if (N equiv V) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if N is equivalent to V.
BLT Branch on Less Than 0024 loc	if (N XOR V) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if exclusive-OR of N and V equal 1.
BGT Branch on Greater Than 0030 loc	if (~Z & (N equiv V)) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if Z not 0 and N equals V.
BLE Branch on Less Than or Equal (zero) 0034 loc	if (Z OR (N XOR V)) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if Z equals 1 or if exclusive-OR of N and V equals 1.
BPL Branch on Plus 1000 loc	if (N = 0) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if N is 0.
BMI Branch on Minus 1004 loc	if (N = 1) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if N is 1.
BHI Branch on Higher 1010 loc	if ~(C OR Z) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if C and Z are 0.

(continued on next page)

Table 2-6 (Cont)
Program Control Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
BLOS Branch on Lower or Same 1014 loc	if (C OR Z) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if C or Z is 1.
BVC Branch on Overflow Clear	if (V = 0) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if V is 0.
BVS Branch on Overflow Set 1024 loc	if (V = 1) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if V is 1.
BHIS Branch on Higher or Same 1030 loc	if (C = 0) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if C is 0.
BLO Branch on Lower 1034 loc	if (C = 1) then (PC ← PC + sign-extend (instr ⟨7:0⟩ X 2))	Branch if C is 1.
JSR Jump to Subroutine 004RDD	SP ← SP - 2; next Mw [SP] ← R[sr]; R[sr] ← PC PC ← D address	Push contents of R onto stack. Store current PC in R. Load subroutine address into PC.
RTS Return from Subroutine 00020R	PC ← R[dr]; R[dr] ← Mw [SP]; SP ← SP + 2	Load contents of R into PC. Pop stack pointer into R.

Table 2-7
Operate Group Instructions

Mnemonic Instruction and Op Code	ISP Notation	Description
HALT Halt 000000	Off ← true	Processor halts with console in control. No activities or instructions can be executed until a console actions restarts the processor.
WAIT Wait 000001	Wait ← true	Processor relinquishes bus and waits for an external interrupt.
IOT I/O Trap 000004	SP ← SP - 2; next Mw [SP] ← PS; SP ← SP - 2; next Mw [SP] ← PC; PC ← Mw [20]; PS ← Mw [22]	Push PS onto Stack. Push PC onto stack. Get new PC from location 20. Get new PS from location 22.
RESET Reset External Bus 000005	Init ← 1; Delay (20 milliseconds); next Init ← 0	Send INIT on Unibus for 20 ms.
SPL Set Priority Level 00023N	PS ⟨7:5⟩ ← df ⟨2:0⟩	Load three least significant bits, N, into PS.
RTI Return from Interrupt 000002	PC ← Mw [SP]; SP ← SP + 2; next PS ← Mw [SP]; SP ← SP + 2	Pop PC off stack. Pop PS off stack. (RTI permits trace trap.)
RTT Return from Interrupt 000006	PC ← Mw [SP]; SP ← SP + 2; next PS ← Mw [SP]; SP ← SP + 2	Pop PC off stack. Pop PS off stack. (RTT inhibits trace trap.)
EMT Emulator Trap 104 Code (104000 – 104377)	SP ← SP - 2; next Mw [SP] ← PS; SP ← SP - 2; next Mw [SP] ← PC; PC ← Mw [30]; PS ← Mw [32]	Push PS onto stack. Push PC onto stack. Get new PC and PS from locations 30 and 32.
TRAP Trap 104 Code (104400 – 104777)	SP ← SP - 2; next Mw [SP] ← PS SP ← SP - 2; next Mw [SP] ← PC; PC ← Mw [34]; PS ← Mw [36]	Push PS onto stack. Push PC onto stack. Get new PC and PS from locations 34 and 36.

Table 2-8
Condition Code Operators

Mnemonic Instruction and Op Code	ISP Notation	Description
CLC Clear C 000241	if (instr ⟨4⟩ = 0 & instr ⟨0⟩ = 1) then C ← 0	When bit 4 of the instruction is 0 bits 3, 2, 1, and 0 clear corresponding bits in PS.
CLV Clear V 000242	if (instr ⟨4⟩ = 0 & instr ⟨1⟩ = 1) then V ← 0	
CLZ Clear Z 000244	if (instr ⟨4⟩ = 0 & instr ⟨2⟩ = 1) then Z ← 0	
CLN Clear N 000250	if (instr ⟨4⟩ = 0 & instr ⟨3⟩ = 1) then N ← 0	
CCC Clear all Condition Codes 000257	if (instr ⟨4⟩ = 0 & instr ⟨3:0⟩ = 17) then (C ← 0; V ← 0; Z ← 0; N ← 0)	
SEC Set C 000261	if (instr ⟨4⟩ = 1 & instr ⟨0⟩ = 1) then C ← 1	When bit 4 of the instruction is 1, bits 3, 2, 1, and 0 set corresponding bits in PS.
SEV Set V 000262	if (instr ⟨4⟩ = 1 & instr ⟨1⟩ = 1) then V ← 1	
SEZ Set Z 000264	if (instr ⟨4⟩ = 1 & instr ⟨2⟩ = 1) then Z ← 1	
SEN Set N 000270	if (instr ⟨4⟩ = 1 & instr ⟨3⟩ = 1) then N ← 1	
SCC Set all Condition Codes 000277	if (instr ⟨4:0⟩ = 37) then (C ← 1; V ← 1; Z ← 1; N ← 1)	

2.3 KB11-A INSTRUCTION TIME

Instruction execution time variation with core memory among a group of processors is due to several factors. The most important of these are:

a. Memory cycle time variation
b. Memory access time variation
c. Unibus propagation time
d. Logic gate delay variations
e. Synchronization uncertainty.

The *PDP-11/45 Processor Handbook* includes a chart of "typical" instruction times in Appendix B, Instruction Timing. An individual system will execute instructions +15 percent, −10 percent of these times, assuming that the core memory is located less than 3 feet from the processor on the Unibus (within the processor box).

For a specific processor/memory system, all of the five major sources of variation are fixed. Hence, it is possible, by taking a few simple measurements, to create a nearly exact set of instruction times for a specific system.

It is the purpose of this section to explain how to generate such a tailored set of instruction times. Paragraph 2.3.7 presents the justification for the method.

Specifically, this section explains how to calculate the time to process a core memory cycle under all possible conditions. To calculate an instruction time, one or more memory cycle times must be added to the additional time required to internally process the instruction. An example of calculating an instruction time is given in this section. Such instruction time calculations require understanding and use of the KB11-A Flow Diagrams (D-FD-KB11-A-03) that are included with the PDP-11/45 System Engineering Drawings (DEC-11-H45A-D).

The flow charts are explained in Chapter 6 (Microprogram Flow Diagrams) of this manual.

2.3.1 Approaches – Typical/Minimum/Maximum/Measured

Typical: The instruction times in the Instruction Timing Chart are typical. A given machine will not vary more than +15 percent, −10 percent from these times. The instruction times represent a machine with the following typical times (MM11-S, MF11-L, or ML11 Memories):

Memory Cycle Time	900 ns
Memory Access Time	
DATI	325 ns
DATO	140 ns
Unibus Propagation Delay	0 ns
MSYN	T_{MS} − 235 ns
MSYN Delayed	T_{MSD} − 355 ns
SSYN Re-sync	T_{SS} − 120 ns

For most purposes, the "typical" times are appropriate. Under special circumstances, it is possible to create minimum, maximum, and tailored instruction times, using the information which follows.

Minimum: This is the fastest that instructions can be processed. Such calculations take into account the least logic delay and assume zero propagation delay between logic elements.

Maximum: This is the slowest that instructions can be processed. It assumes the worst case logic delay under worst case temperature (very cold). It also assumes zero propagation delay between logic elements.

Measured: By making a few measurements on a system, the instruction time for all instructions on a specific system can be calculated. Note that all times will be different for each memory unit on the system since each one will have different cycle and access time.

2.3.2 Steps to Calculate Instruction Times

There are three steps required to calculate instruction times with the KB11-A Processor (PDP-11/45 and PDP-11/50) for other than the typical values listed in the *PDP-11/45 Processor Handbook,* Appendix B.

2.3.2.1 **Step 1: Subcycle Times** – The first step is to specify values for several subcycle times. Paragraph 2.3.7 explains the source of these subcycle times. The times to be determined are:

T_C – Memory Cycle Time
T_A – DATI Access Time
T_A – DATO Access Time
T_{MS} – MSYN Generation Time
T_{MSD} – MSYN Generation Time Delayed
T_P – Unibus Propagation Delay
T_{SS} – SSYN Restart Time
$T_{S/M}$ – SSYN to MSYN Time

Paragraph 2.3.3 gives minimum, typical, and maximum values for these subcycle times. In addition, instructions on how to measure the specific time on a given system are included.

2.3.2.2 **Step 2: Cycle Times** – Next, the subcycle times above must be used to create a set of seven cycle times. The seven cycle times are:

1. DATI or DATIP
2. DATO
3. DATI or DATIP with Previous DATO
4. DATI or DATIP with Previous DATI
5. DATO with Previous DATI
6. DATO with Previous DATO
7. DATO with (T_{MSD}).

Cycle times 1 and 2 are normal cycles. Cycle times 3–6 may be increased due to previous memory activity that leaves the memory busy. Cycle time 7 waits for data from the processor. There are no memory cycles that wait both for busy memory and data from the processor. Paragraph 2.3.4 explains how to calculate these seven cycle times.

2.3.2.3 **Step 3: Instruction Time** – An instruction time is defined as beginning with microstate IRD.00 and ending with the completion of the fetch of the next instruction. The instruction time is then the sum of one or more cycle times from step 2 and, generally, one or more processor microstate times. Each microstate time, that is not part of a memory cycle, takes 150 ns. Paragraph 2.3.5 gives an example of instruction time calculation.

2.3.3 Determining Subcycle Times

Paragraphs 2.3.3.1–2.3.3.6 give values for the subcycle times and instructions for measuring the time on a specific system.

2.3.3.1 MSYN Generation Time (T_{MS}) –

Calculated T_{MS}:

Minimum	195 ns
Typical	235 ns
Maximum	300 ns

Measuring T_{MS}:

Measure from the leading edge of signal TIGD T3 B H going positive (pin E12J2) to the leading edge of signal BUSA MSYN L going negative (pin E12D1).

NOTE
All pin numbers refer to the KB11-A backpanel unless otherwise stated.

Program:

Do BR. in memory of interest.

2.3.3.2 MSYN Generation Time Delayed (T_{MSD}) –

Calculated T_{MSD}:

Minimum	319 ns
Typical	355 ns
Maximum	417 ns

Measuring T_{MSD}:

Same as T_{MS}.

Program:

```
GO:   MOV R0, @ R0
      JMP GO (where R0 is not within the program)
```

Use longest T_{MSD} as value.

NOTE
All values of T_{MS} and T_{MSD} are increased by 90 ns if the memory management option (KT11-C) is operating.

2.3.3.3 MM11-L Access Time (T_A) –

Access Time MM11-L (DATI)

Calculated T_A:

Minimum	300 ns
Typical	325 ns
Maximum	400 ns

Measure on the MM11-L Backpanel:

Measure the time from the negative-going edge of BUS MSYNC L on pin B03V1 to the negative-going edge of BUS SSYN L on pin B03U1.

Access Time MM11-L (DATO)

Calculated T_A:

Minimum	80 ns
Typical	140 ns
Maximum	160 ns

Measuring T_A:

Use the same measuring points as DATI. Use the shorter of the two times displayed (the longer one is a DATI).

Program:

```
GO:     MOV R0, TEMP
        JMP GO
TEMP:   0
```

2.3.3.4 MM11-L Cycle Time (T_C) – The MM11-L/Cycle time is 900 ns ± 25 ns.

To measure, do BR. in the memory of interest. Measure the time from the leading edge of MSYN (positive edge) pin 13 of E23 on the G110 module of the MM11-L to the trailing edge of MSEL (positive edge) pin 11 of E01 on the G110 module.

2.3.3.5 Unibus Propagation Delay (T_P) – The round trip delay of signals on the Unibus is 3.6 ns/ft.

In addition, each device (unit load) located between the KB11-A and the MM11-L of interest adds an equivalent of 1 foot delay (3.6 ns).

2.3.3.6 SSYN Resync Time (T_{SS}) –

Calculated T_{SS}:

Minimum	90 ns
Typical	120 ns
Maximum	170 ns

Measuring T_{SS}:

Measure the time from the leading edge of BUSA SSYN L (negative edge) on pin C12J1 to the leading edge of TIGD T3 B H (positive going) on pin E12J2.

2.3.4 Calculating Cycle Times

Instructions and formulas for calculating cycle times are given in this section. Their derivation is explained in Paragraph 2.3.7.

2.3.4.1 DATI and DATIP –

$$\text{Cycle Time} = 150 + T_{MS} + T_A + T_P + T_{SS}$$

2.3.4.2 DATO –

$$\text{Cycle Time} = 150 + T_{MS} + T_A + T_P + T_{SS}$$

2.3.4.3 DATI or DATIP with Immediately Previous DATO –

2.3.4.3.1 DATO Is Not Part of DATIP-DATO Sequence –

a. Compute $T_{S/M} = T_P + T_{SS} + 150 + T_{MS}$

b. Compute the time memory is busy for a DATO:

$$T_B = T_C - T_A$$

c. Subtract $T_{S/M}$ from T_B. If the difference is positive, the value is added to the time calculated for DATI or DATIP in Paragraph 2.3.4.1. If the difference is negative, the previous DATO has no effect on the DATI or DATIP time.

2.3.4.3.2 DATO Is Part of DATIP-DATO Sequence –

a. Compute $T_{S/M}$ from Paragraph 2.3.4.3.1., a.

b. Compute the time memory is busy for a DATO where:

$$T_B = [T_C - T_A \text{ (DATIP)}] - T_A \text{ (DATO)}$$

c. Subtract $T_{S/M}$ from T_B. If the difference is negative, the DATO will have no effect on the DATI or DATIP time.

2.3.4.4 DATI or DATIP with Immediately Previous DATI –

a. Compute the time memory is busy for a DATI:

$$T_B = T_C - T_A.$$

b. Subtract $T_{S/M}$ as calculated in Paragraph 2.3.4.3 from T_B as just calculated. If the difference is positive, the value is added to the time calculated for a DATI or DATIP in Paragraph 2.3.4.1. If the difference is negative, the DATI has no effect on DATI or DATIP time.

2.3.4.5 DATO with Immediately Previous DATI – Add the difference calculated in Paragraph 2.3.4.4, part b to the DATO time calculated in Paragraph 2.3.4.2. If that difference is negative, the DATI has no effect on the DATO time.

2.3.4.6 DATO with Immediately Previous DATO (where first DATO is not part of DATIP-DATO sequence) – If the difference computed in Paragraph 2.3.4.3.1, part c, is positive, add it to the DATO time. If it is negative, the previous DATO has no effect on the current DATO.

2.3.4.7 DATO (with T_{MSD}) – Recompute a value for Paragraph 2.3.4.2 using T_{MSD} instead of T_{MS}.

2.3.5 Example of Calculating an Instruction Time

This section will illustrate the three steps, for the following instruction, using typical values:

MOV R0, @ – (R3)

2.3.5.1 Step 1 – For a typical system, we have as subcycle values:

900 ns = T_C Memory Cycle
325 ns = T_A DATI Memory Access
140 ns = T_A DATO Memory Access
235 ns = T_{MS} MSYN Generation
0 ns = T_P Unibus Delay
355 ns = T_{MSD} MSYN Generation Delayed
120 ns = T_{SS} SSYN Restart

2.3.5.2 **Step 2 –**

1. DATI or DATIP

   $T = 150 + T_{MS} + T_A + T_P + T_{SS}$

   $T = 150 + 235 + 325 + 0 + 120$

   $T = 830$ ns

2. DATO

   $T = 645$ ns

3. DATI or DATIP with previous DATO
   a. DATO is not part of DATIP-DATO Sequence:

      $T = 1085$ ns

   b. DATO is part of DATIP-DATO Sequence:

      $T = 830$ ns

4. DATI or DATIP with previous DATI

   $T = 900$ ns

5. DATO with previous DATI

   $T = 715$ ns

6. DATO with previous DATO

   $T = 900$ ns

7. DATO (T_{MSD})

   $T = 765$ ns

2.3.5.3 **Step 3** – Go to sheet 1 of KB11-A microstate flow diagrams. First, generate a list of the microstates that the KB11-A will sequence through in processing the instruction.

Microstate Code	Address	Flow Sheet	Comment
IRD.00	343	3	BUST
D45.00	004	6	BEND
D10.00	162	6	BUST
D10.10	231	6	PAUSE
D10.20	233	6	–
D10.50	311	6	BUST (T_{MSD}!)
D10.40	157	6	PAUSE
FET.01	337	1	BUST
FET.11	321	1	PAUSE

Before proceeding with the details, there are several interesting things to observe about the above sequence. First, IRD.00 and D45.00 do *not* constitute a memory cycle. The "BEND" effectively causes the "BUST" to be ignored. Second, the states D10.50 and D10.40 will have a T_{MSD}. Note that in D10.50 the BR (Unibus Data Register) is changed at the end of the microstate. The symbolic representation from the flow charts is:

$$t_6 \ BR \leftarrow SHFR$$

or as shown below:

D10.50 (311)

DATIP, DATI, DATO, DST, SRC OPERAND TO BR
t_1 BA ← DR; BC ← BSOP1
t_2 SHFR ← SR
t_3 BUST; GR [0]
t_6 BR ← SHFR

NOTE
In this "BUST" microstate the BR register is loaded at t_6 necessitating the use of T_{MSD}.

Third, note that the fetch of the next instruction will possibly be delayed by the immediately previous cycle.

The time for IRD,00 and D45.00 is:

IRD.00 = 150 ns
D45.00 = 150 ns

D10.00 and D10.10 constitute a DATI cycle with *no* immediate previous memory cycle, hence (type 1):

D10.00 + D10.10 = 830 ns.
D10.20 has no memory cycle, hence, is D10.20 = 150 ns.
D10.50 and D10.40 constitute a DATO cycle that is delayed by processor data (type 7):
D10.50 + D10.40 = 765 ns.
FET.01 and FET.11 constitute a DATI cycle that is immediately preceded by a DATO cycle (type 3); FET.01 + FET.11 = 1085 ns.

Adding these up:

150 ns	IRD.00
150 ns	D45.00
830 ns	D10.00 and D10.10
150 ns	D10.20
765 ns	D10.50 and D10.40
1085 ns	FET.01 and FET.11
3130 ns	Total time for MOV R0, @ – (R3)

2.3.6 Comments on the Instruction Times Table (PDP-11/45 Processor Handbook)

Since many instructions follow the same microprogram sequence in the KB11-A Processor, it is possible to compute the instruction time for more than one instruction at once. Likewise, the processing of source and destination operands occurs through the same sequences for all instructions.

The table is organized to take advantage of these common paths by grouping sets of instructions together. Also, for convenience, the execution portion of an instruction is combined with the fetch of the next instruction into one number.

2.3.7 KB11-A Cycle Delays and Speed Variation

This section assumes an understanding of Paragraphs 7.7.1–7.7.3 of this manual, which describe Unibus timing and operation. Cycle times for core memory on the Unibus will be derived. The effect of previous bus activity and bus length will be taken into account.

Cycle Time Definition: The *cycle time* is defined from the leading edge of T1 of the BUST microstate to the leading edge of the T1 that follows the PAUSE ROM state.

Method: First, the times for a cycle time unaffected by previous activity will be calculated. Then the effect of previous activity will be added.

2.3.7.1 **Basic Memory Cycle** – The processor starts a cycle by running through two time states (T1 and T2) for a total elapsed time of 60 ns. The logic to generate MSYN is activated at the end of T2. Normally, some 235 ns later, MSYN is actually generated. It propagates down the Unibus to the memory where, if the memory is not busy with a previous cycle, it immediately activates a memory cycle. The memory, after a delay of 140–400 ns generates SSYN. SSYN then propagates back up the Unibus to the processor.

The processor, during the time of MSYN and SSYN generation, has stepped through a series of time states and is waiting to proceed into T3 of the PAUSE cycle. When SSYN reaches the processor, it takes about 140 ns to step the processor to T3. It then runs through T3, T4, and T5 of PAUSE, taking an additional 90 ns, thus completing the cycle time. These "subportions" of the cycle time will not be examined in detail.

T1, T2 of BUST are fixed by the crystal clock to be exactly 60 ns.

T_{MS} is the time from the end of T2 in the BUST microstate to MSYN appearing on the copper wire of the Unibus (output of Unibus driver circuit). Figure 2-3 shows the derivation of this time. Note that the effect of the four logic elements on the left is to cause signal UBCB DESKEW A to become true, either 30 ns after T3 or 60 ns after T3, if the total delay is greater than 30 ns.

T_{MSD}. In several PAUSE microstates, the DESKEW register is reset at T2. This adds either 90 or 120 ns to the time for MSYN to be generated. The additional time is required to allow proper deskew between the Unibus DATA lines and MSYNC when the DATA lines are changed during the BUST microstate. T_{MSD} is used only with cycles that start in the following BUST microstates:

Code	Address	Flows Sheet	Code	Address	Flows Sheet
D12.00	001	5	SHR.10	123	11
D12.01	002	5	EXC.00	031	11
D10.50	311	6	FSV.00	245	12
D10.30	122	6			

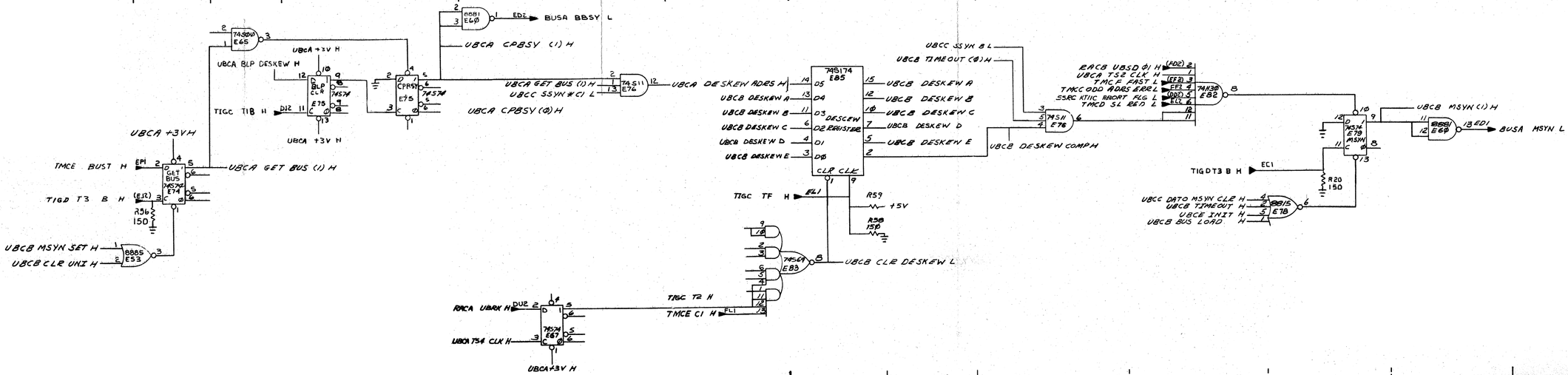
Unibus propagation delay is the time for a signal to propagate down and back one foot of Unibus cable is nominally 3.6 ns. This figure accounts for the time for MSYN to get from the processor to the memory and for SSYN to get from the memory to the processor.

Memory access time is determined, entirely, by the memory being accessed. It is defined, *here* and *only here*, as the time from receiving MSYN at the receiver input to the time of sending SSYN at the driver output. On MM11-L type core memories, on *write* cycle, the time is about 140 ns. For *read* cycles, it is less than 400 ns and typically 325 ns.

LOGIC ELEMENT	74S74	74S00	74S74	74S11	TOTAL	DESKEW	74S174	74S11	74H30	74S74	8881	TOTAL	T_{ms} TOTAL
Minimum Delay	2.25	1.75	1.75	2.25	8	180.	5	2.25	.0	1.75	10.	19.	199 ns
Typical Delay	7.	3.	5.	4.5	19.5	180.	9	4.5	8.9	5.	26.	53.4	235 ns
Maximum Delay	12.	6.	14.	8.5	40.5	210.	27	8.5	12.0	14.	55.	226.5	327 ns

Second TIGC TFH after T3 makes UBCB DESKEW A true (delay > 30 ns)

Next TIGC 7F H after T3 makes UBCB DESKEW A true (delay < 30 ns)



							T_{msd} TOTAL
Min	300	5	2.25	.0	1.75	10	19.	319 ns
Typ	300	9	4.5	8.9	5.0	26	53.4	355 ns
Max	300	27	8.5	12.0	14.0	55	116.5	417 ns

11-2400

Figure 2-3 Derivation of Time from Leading Edge of T3 to BUSA MSYN L (T_{MS})

T_{SS} is the time the processor requires to "restart" the timing after receiving SSYN. Figure 2-4 shows the derivation. Note that the process is largely synchronous with the processor clock.

Note that we have not, thus far, accounted for the effect of the memory being busy from a previous cycle at the time it receives MSYN.

2.3.7.2 Effect of Previous Cycle Memory Busy – If a PAUSE microstate immediately proceeds the BUST microstate that starts our cycle, then the memory may be busy with the previous cycle and, hence, delay its response to MSYN.

Figure 2-5 shows how the current memory cycle is delayed by an immediately previous DATI or DATO cycle. Consider the time a previous cycle needs to complete a memory cycle, once it has issued SSYN. The memory will be busy for a period equal to the difference between the access time and the cycle time for the type of memory cycle. If the memory cycle time (T_C) is 900 ns, the cycle is a DATO with an access time (T_A) of 140 ns, then the memory will continue to be busy for

$$900 \text{ ns } (T_C) - 140 \text{ ns } (T_A) = 760 \text{ ns.}$$

Hence, the memory, under these circumstances, will not process another MSYN for 760 ns after issuing SSYN. Meanwhile, consider what is happening within the processor once SSYN is generated. For the DATO cycle above, the following occurs:

1. SSYN propagates up the Unibus to the processor (½ X T_P).
2. SSYN causes the timing to restart (T_{SS}).
3. The processor runs through T3, T4, and T5 of the PAUSE microstate, then through T1 and T2 of the next BUST microstate (150 ns).
4. MSYN is then generated, T_{MS} or T_{MSD} (= $T_{MS/D}$).
5. MSYN propagates down the Unibus to the memory (½ X T_P).

We can now add these times:

$$T_{S/M} = ½ \times T_P + T_{SS} + 150 + T_{MS/D} + ½ \times T_P$$

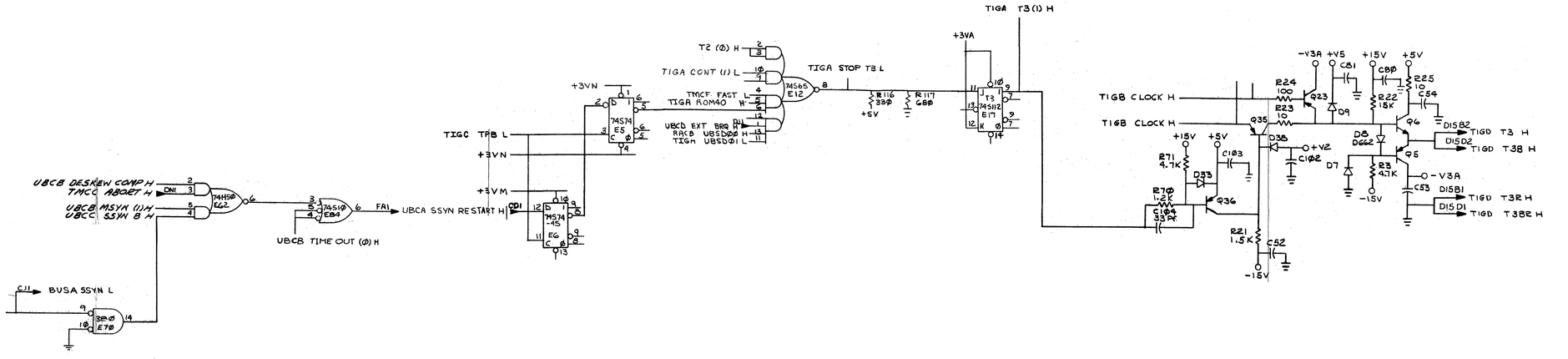
$$T_{S/M} = T_P + T_{SS} + 150 + T_{MS/D}$$

If we assume that $T_P = 0$ and use typical times for T_{SS} and T_{MS}:

$$\begin{array}{rl} & 150 \\ & 120\ T_{SS} \\ & \underline{235\ T_{MS}} \\ T_{MS/D} = & 505 \text{ ns} \end{array}$$

Hence, the processor has a new MSYN ready 505 ns after receiving SSYN, but the memory will not process it for 760 ns.

$$\begin{array}{l} 760 \text{ ns memory busy} \\ \underline{505 \text{ ns next MSYN}} \\ 255 \text{ ns = delay} \end{array}$$



LOGIC ELEMENT	380	74H50	74S10	SYNC WAIT	SYNC	SYNC	SYNC	T_{SS} TOTAL
Minimum Delay	10	–	1.75	0	30	30	15	86.75 ns (Min)
Typical Delay	20	7	3.	15	30	30	15	120. ns (Typ)
Maximum Delay	45	15	6.	30	30	30	15	171. ns (Max)

11-2399

Figure 2-4 Derivation of Time from Leading Edge of SSYN to T3 (T_{SS})

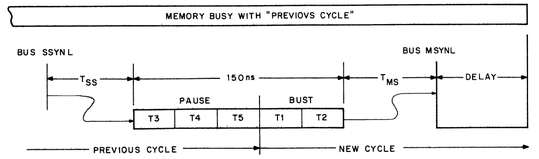


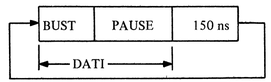
Figure 2-5 Cycle Delay Due To Memory Busy

Hence, for a typical processor/memory combination, a DATI cycle immediately preceded by a straight DATO (not DATIP-DATO) would be extended by 240 ns over a normal DATI. In the case of a DATIP-DATO, the DATO portion takes only one-half a normal memory cycle (450 ns). Hence, such DATOs cannot delay following cycles since the processor cannot issue another MSYNC early enough.

This completes the justification of the cycle times as listed in Paragraph 2.3. Next, the system percentage speed tolerance is derived.

2.3.7.3 Fast Processor – The condition, which would lead to the greatest percentage deviation in a negative direction of instruction time, would be an instruction that was not limited by memory speed. That is an instruction that had no "immediately previous cycles."

Such a loop would be:



Time for the above (using typical DATI):

830 ns + 150 ns = 980 ns

Minimum time for the above:

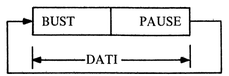
150 ns + 195 ns + 300 ns + 90 ns = 735 ns

Typical time = 830 ns – minimum time = 735 ns = 95 ns faster

$$\frac{\text{95 ns faster}}{\text{980 ns base total}} = 0.097 = 9.7\ \%$$

Minimum = 10 % faster.

2.3.7.4 Slow Processor – Assume that the situation creating the greatest percentage of decrease in system speed is when all microstates are part of memory cycles.



This is a type 4 cycle; hence, first type 1 is calculated:

$$150 \text{ ns} + 300 \text{ ns} + 400 \text{ ns} + 170 \text{ ns} = 1020 \text{ ns}$$

Next, compute:

$$0 \text{ ns} + 170 \text{ ns} + 150 \text{ ns} + 300 \text{ ns} = T_{S/M} = 620 \text{ ns}$$

Taking the longest $T_C = 925$ ns:

$$T_B = 925 \text{ ns} - 400 \text{ ns} = 525 \text{ ns}$$

$$\begin{aligned} T_B &= 525 \text{ ns} \\ -T_{S/M} &= 620 \text{ ns} \\ \hline &-95 \text{ ns} \end{aligned}$$

Hence, the memory cycle is not the limiting factor, and the cycle is 1020 ns.

Next, we compute the increase percentage:

$$\begin{array}{l} 1020 \\ -900 \text{ base} \\ \hline 120 \text{ increase} \end{array} \qquad \frac{120}{900} = 0.135 = 13.5\%. \text{ Rounded off: Max} = 15\%$$

CHAPTER 3
OPERATION

3.1 CONSOLE CONTROLS AND INDICATORS

The operator's control console is shown in Figure 3-1. Control and indicator functions are summarized in Table 3-1.

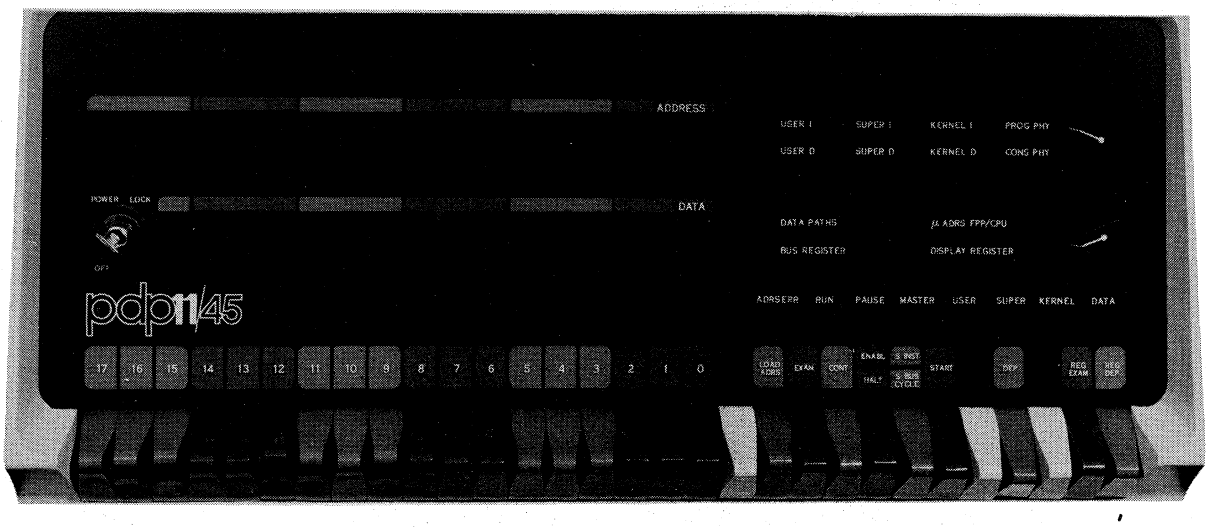


Figure 3-1 KB11-A Control Console

Table 3-1
Control and Indicator Functions

Control or Indicator	Function
Power Control	
OFF	Disconnects power from all units except semiconductor memory system.
POWER	Applies power to all units. All console controls are operable.
LOCK	Disables all console controls except switch register.

(continued on next page)

Table 3-1 (Cont)
Control and Indicator Functions

Control or Indicator	Function
DATA Display Register	16-bit data display, source selected by data display select switch as follows:
DATA PATHS	Displays current data output of ALU shifter.
BUS REGISTER	Displays current contents of bus register A (BRA).
DISPLAY REGISTER	Displays current contents of light register located at physical address 777570.
μADRS FPP/CPU	Displays current floating-point processor ROM address in high byte, and current central processor ROM address in low byte.
ADDRESS Display Register	An 18-bit address display. When the KT11-C Memory Management Unit is implemented and enabled, the displayed address is selected by the adjacent address display select/mode control switch as indicated in Figure 3-2:
PROG PHY	Program Physical Address. Constructed by adding virtual address bits VA ⟨12:06⟩ to the contents of a KT11-C page address register (PAR) to provide physical address bits PA ⟨17:06⟩.
CONS PHY	Console Physical Address. After LOAD ADRS, PA ⟨15:06⟩ equals sum of switches ⟨15:06⟩ and PAR contents; PA ⟨17:16⟩ equals switches ⟨17:16⟩.
USER I, USER D SUPER I, SUPER D KERNEL I, KERNEL D	These six address display select switch positions display a 16-bit virtual address. Address bits 17 and 16 light only if address bits ⟨15:13⟩ are lit. These positions provide console control of the processor mode (kernel, supervisor, user) I or D space. The same address information is displayed for each position.
Switch Register	An 18-bit switch bank used to load addresses or data, depending upon whether LOAD ADRS or DEP switch is operated. The contents of the switch register are accessed by the processor in kernel mode at physical address 777570.
LOAD ADRS	Loads switch register contents into program counter (PCA). If KT11-C is not implemented, switch register bits ⟨17:16⟩ are not used.
EXAM	Displays contents of current ADDRESS display (CONS PHY) in the DATA display. Each consecutive time EXAM is pressed, the ADDRESS display increments by 2 and the contents of the next word location are presented on the DATA display.
CONT	Causes the processor to continue operation from the point at which it stopped. Function depends upon ENABL/HALT and S INST/S BUS CYCLE as follows:
If ENABL:	CONT returns bus control from console to the processor and program operation continues.

(continued on next page)

Table 3-1 (Cont)
Control and Indicator Functions

Control or Indicator	Function
If HALT:	Pressing CONT causes processor to execute a single instruction, if S INST, or continue until a bus cycle has been completed, if S BUS CYCLE.
ENABL/HALT	ENABL allows processor to run in normal operation. The HALT position halts the processor and passes control to the console. HALT affects CONT and START switch functions as described for those switches.
S INST/S BUS CYCLE	Allows the processor to step through program operation either one instruction at a time (S INST), or one bus cycle at a time (S BUS CYCLE).
START	Function depends on ENABL/HALT switch position, as follows:
If ENABL:	Pressing START provides system clear and initiates processor operation at address established by LOAD ADRS function.
If HALT:	Provides system clear only.
DEP	When lifted, deposits current contents of the switch register into the location indicated by the ADDRESS display. Each time DEP is lifted in succession, the location is incremented by 2 and the switch register contents are deposited into the next word location.
REG EXAM	Displays contents of the general register specified by the four low-order bits of the ADDRESS display.
REG DEP	Deposits contents of the switch register into the general register specified by the four low-order bits of the ADDRESS display.
ADRS ERR	Indicates one of the following abort-condition errors has been detected: a. odd address error b. fatal stack violation c. non-existent memory addressed d. parity error e. memory management
RUN	Indicates processor is executing program instructions. Indicator not lit in pause cycle, or while console flag is set.
PAUSE	Indicates processor is in pause cycle, waiting for completion of Unibus or Fastbus transaction.
MASTER	Indicates processor is in control of Unibus as master device or in console mode.
USER	Indicates processor is in user mode. When KT11-C option is enabled, all addresses are in user virtual address space.
SUPER	Indicates processor is in supervisor mode. When KT11-C option is enabled, all addresses are in supervisor virtual address space.

(continued on next page)

Table 3-1 (Cont)
Control and Indicator Functions

Control or Indicator	Function
KERNEL	Indicates processor is in kernel mode. When KT11-C option is enabled, all addresses are in kernel virtual address space.
DATA	Indicates last memory address reference was D space when lit. If not lit, last memory address reference was I space.

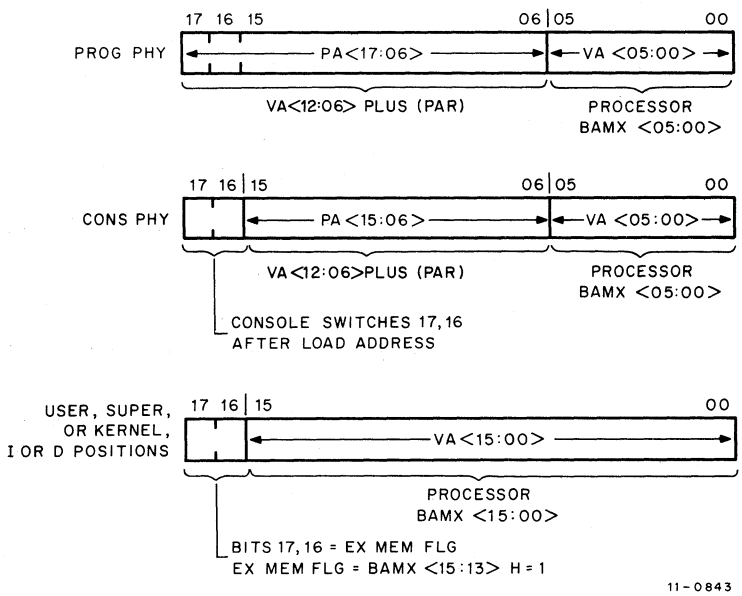


Figure 3-2 Sources of ADDRESS Display with
KT11-C Memory Management Unit

3.2 POWER ON

When the console power control switch is turned from OFF to POWER, all internal registers and buses are initialized. A 70-ms delay allows time for magnetic core memory power to stabilize. Power distribution to the MS11 Semiconductor Memory System is not affected.

The power-up initialization logic forces the central processor ROM address to ZAP.00 (CPU μADRS 200). At that point, the sequence of microprogram-controlled events is determined by the setting of the ENABL/HALT switch on the console.

3.2.1 ENABL Function

When power is turned on at the console with ENABL/HALT set to ENABL, the processor executes the power-up microprogram sequence and halts at the address determined by the start vector. The start vector is determined by jumper connections on DAP Module M8100. On most processors, the jumpers are cut so that the processor is vectored to virtual address 24_8, which is the power-fail trap location. For those processors, the ADDRESS

display will be 30. When the console START switch is pressed, the processor will execute program instructions, starting at that location.

3.2.2 HALT Function

When power is turned on at the console with ENABL/HALT set to HALT, the processor is forced to ZAP.00 and then branches to CON.00 (CPU μADRS 170). The processor remains at rest in the CON.00 microstate until a console control function is initiated. The functions of the console switches are described in the following paragraphs.

NOTE

If the START switch is pressed while ENABL/HALT is in HALT position, a console reset occurs. As a result, the processor is suspended in ZAP.00 and the timing generator is cleared. As soon as the START switch is released, the processor reverts to CON.00.

3.3 CONSOLE OPERATIONS

This paragraph provides additional information on console operations related to processor functions described in Chapter 6.

3.3.1 HALT Switch Functions

If the HALT switch is pressed while the processor is running, the console flag is set, causing the processor to enter a rest state at microstate CON.00, at location 170. This microprogram ROM address is displayed in the low byte of the DATA display when the data display select switch is set to μADRS FPP/CPU. Succeeding operations depend upon console control settings.

3.3.1.1 **HALT/CONT with S INST** – With the S INST/S BUS CYCLE switch set to S INST, the processor will execute a single instruction each time the CONT switch is pressed. At the end-of-instruction microstate associated with each instruction sequence, a strobe occurs to set the console flag and cause the processor to enter the CON.00 microstate. The time state generator continues to run.

3.3.1.2 **HALT/CONT with S BUS CYCLE** – With the S INST/S BUS CYCLE switch set to S BUS CYCLE, the processor will execute the program until the next bus cycle is completed, each time the CONT switch is pressed. The time state generator is suspended in time state T2 of the next bus cycle after PAUSE.

NOTE

Single-step operations are provided for maintenance operations. Detailed descriptions of their use with special maintenance cards are provided in Chapter 4 of the system maintenance manual.

3.3.2 EXAM Switch Functions

Use the following procedures to examine the contents of a memory location or internal register:

Step	Procedure
1	Set up address of location or register on switch register.
2	Set address display select switch to CONS PHYS.
3	Set data display select switch to DISPLAY REGISTER.

(continued on next page)

Step	Procedure
4	Press LOAD ADRS switch and check ADDRESS display for selected address.
5	Press EXAM and observe DATA display.

Each successive time EXAM is pressed, the contents of the next successive word location are displayed. The ADDRESS display will indicate the location. However, the initial address, loaded into the program counter (PCA) will not be incremented. If the START switch is pressed, execution starts from the initial address.

3.3.3 DEP Switch Functions

Use the following procedures to deposit data into a memory location or internal register:

Step	Procedure
1	Set up address of location or register on switch register.
2	Set address display select switch to CONS PHYS.
3	Press LOAD ADRS switch and check ADDRESS display for selected address.
4	Set up data to be deposited on switch register.
5	Lift DEP switch.
6	Set data display select switch to DATA PATHS and check DATA display for correct input.

Each successive time DEP is pressed, the contents of the next successive word location are accessed. There is no need to increment the address manually.

NOTE
The address cannot be incremented beyond the current 32K-word boundary using the step-examine or step-deposit features.

3.3.4 REG EXAM and REG DEP Functions

These switches permit the operator to examine the contents of the general register and to deposit the contents of the switch register into the general registers. Table 3-2 lists the general register addresses.

To examine the contents of the general register and deposit the contents of the switch register into the general register, use the following procedures:

Step	Procedure
1	Set the switch register to the general register address.
2	Press LOAD ADRS. The ADDRESS display will indicate the selected register address.
3	To examine the contents, press REG EXAM. The contents will be displayed by the DATA display.
4	To deposit, set the data into the switch register, then press REG DEP. The DATA display will indicate the deposited data.

NOTE
The REG EXAM and REG DEP switches do not provide automatic address stepping. Each general register must be addressed individually, using the LOAD ADRS switch.

Table 3-2
General Register Addresses

Address (octal)	General Register Name
0	R0 – General Register Set 0
1	R1 – General Register Set 0
2	R2 – General Register Set 0
3	R3 – General Register Set 0
4	R4 – General Register Set 0
5	R5 – General Register Set 0
6	R6 – Kernel Mode Stack Pointer (SP)
7	R7 – Program Counter (PC)
10	R0 – General Register Set 1
11	R1 – General Register Set 1
12	R2 – General Register Set 1
13	R3 – General Register Set 1
14	R4 – General Register Set 1
15	R5 – General Register Set 1
16	R6 – Supervisor Mode Stack Pointer (SP)
17	R7 – User Mode Stack Pointer (SP)

3.4 ADDRESS DISPLAY SELECT

The source of the ADDRESS display is determined by the 8-position address display select switch; it depends on implementation and enabling of the KT11-C Memory Management Unit. For example, the KT11-C option may be available but is not enabled if bit 0 of KT11-C status register SR0 (physical address 777572) is cleared. Figure 3-2 shows the source of the ADDRESS display with memory management implemented and enabled. Virtual address (VA) bits are logically identical to processor bus address multiplexer (BAMX) bits. The six low-order bits, VA ⟨05:00⟩, indicate displacement within a 32-word block and are not affected by relocation or address display select switch positions.

3.4.1 PROG PHY Function

Use this address display select switch position to display the current 18-bit physical address. The physical address is constructed by adding virtual address bits VA ⟨12:06⟩ to the contents of the 12-bit page address register (PAR).

3.4.2 CONS PHY Function

Use this address display select switch position to display results of loading an address from the console switch register. Physical address bits PA ⟨17,16⟩ are generated directly from switch register bits SR ⟨17,16⟩.

3.4.3 USER, SUPER, or KERNEL Functions

The ADDRESS display source for each of the USER, SUPER, and KERNEL switch positions is the 16-bit virtual address VA ⟨15:00⟩. The two most-significant bits are logic 1 if bits 15 through 13 are all logic 1.

The primary purpose of these six mode-related switch positions is to provide direct console-controlled access to the I and D space PAR groups associated with each mode of operation. The following chart lists the PAR group associated with each switch position.

Address Display Select Switch	Page Address Register (PAR) Group	Physical Address Ranges*
USER I	UIPAR0 – UIPAR7	777640 – 777656
USER D	UDPAR0 – UDPAR7	777660 – 777676
SUPER I	SIPAR0 – SIPAR7	772240 – 772256
SUPER D	SDPAR0 – SDPAR7	772260 – 772276
KERNEL I	KIPAR0 – KIPAR7	772340 – 772356
KERNEL D	KDPAR0 – KDPAR7	772360 – 772376

*Virtual address bits VA ⟨15:13⟩ select one of eight specific PAR addresses within each group.

NOTE

If the KT11-C option is not implemented, the 16-bit virtual address, VA ⟨15:00⟩, is always the ADDRESS display source. Bits 15, 14, and 13 are ANDed to provide bits 17 and 16.

3.5 HOW TO LOAD AND RUN PROGRAMS

Figure 3-3 is a flowchart which shows the procedure required to load and run programs. The following paragraphs detail the procedures indicated in the flowchart.

3.5.1 Loading the PDP-11 Bootstrap Loader

Use the following procedures to manually load the PDP-11 Bootstrap Loader program, DEC-11-L1PA-LA (Table 3-3):

Step	Procedure
1	Set ENABL/HALT switch to HALT to give bus control to the console when powering up the processor.
2	Turn OFF/POWER/LOCK switch to POWER. Press START to clear system, including KT11-C option, if implemented.
	NOTE **Because the primary purpose of these procedures is to instruct maintenance personnel in loading and running diagnostic programs, be sure the KT11-C option is initially disabled.**
3	Set starting address of Bootstrap Loader into the switch register. Be certain that the correct value of xx is used (017744 for 4K memory, 037744 for 8K memory, 057744 for 12K memory, etc.) (Table 3-3).
4	Set address display select switch to CONS PHY and press LOAD ADRS. The starting address should be displayed by the ADDRESS indicators.

(continued on next page)

Step	Procedure
5	Set first instruction of Bootstrap Loader program into the switch register (Table 3-3). Lift DEP switch. The switch register contents should be displayed by the DATA indicators, with the data display select switch set to DISPLAY REGISTER.
6	Set contents of the next address of the Bootstrap Loader program into the switch register and lift DEP switch.

NOTE

It is not necessary to load addresses after the starting address has been loaded because the address is incremented by 2 each time the DEP switch is lifted sequentially.

7	Repeat Step 6 to deposit the Bootstrap Loader program. When loading the contents of xx7766, make certain the correct xx value is used. When loading the contents of the last address, make certain the correct device address, yyyyyy, is used, as indicated in Table 3-3.
8	Load the starting address of the Bootstrap Loader and use the EXAM switch to verify that the program has been loaded correctly.

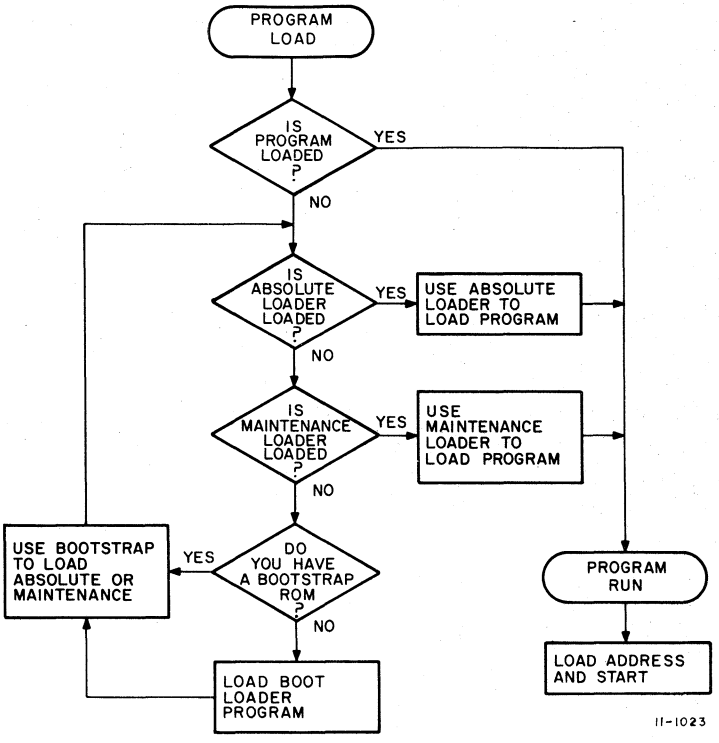


Figure 3-3 Flowchart of Procedure Required to Run a Program

Table 3-3

PDP-11 Bootstrap Loader
DEC-11-L1PA-LA

Address*	Contents	Symbolic		
xx7744	016701	START:	MOV	DEVICE,R1
xx7746	000026			
xx7750	012702	LOOP:	MOV	#.-LOAD+2,R2
xx7752	000352			
xx7754	005211	ENABLE:	INC	@R1
xx7756	105711	WAIT:	TSTB	@R1
xx7760	100376		BPL	WAIT
xx7762	116162		MOVB	2(R1),LOAD(R2)
xx7764	000002			
xx7766	xx7400			
xx7770	005267		INC	LOOP+2
xx7772	xx7756			
xx7774	000765	BRNCH:	BR	LOOP
xx7776	yyyyyy			

NOTES: 1. The highest available 4K page of memory is represented by xx. In a PDP-11/45 with up to 28K of memory, the first address of the Bootstrap Loader is one of the following, depending upon the total memory available; xx will be the same for all subsequent addresses.

Available Memory	Starting Address
4K	017744
8K	037744
12K	057744
16K	077744
20K	117744
24K	137744
28K	157744

2. Location xx7776 contains device address of paper-tape reader.
3. Use address 177560 for teletypewriter paper-tape reader; use address 177550 for high-speed paper-tape reader.

* Starting address, xx, is determined by memory configuration.

3.5.2 Loading the PDP-11 Absolute Binary Loader

Use the following procedures to automatically load the PDP-11 Absolute Binary Loader program, DEC-11-L2PC-LA:

Step	Procedure
1	Set ENABL/HALT switch to HALT and press START to clear the system.
2	Make certain that the PDP-11 Bootstrap Loader has been stored in memory, as described in Paragraph 3.5.1, or the equivalent ROM bootstrap is supplied.
3	Set starting address of Bootstrap Loader into the switch register. Make certain the correct value of xx is used (017744 for 4K memory, 037744 for 8K memory, 057744 for 12K memory, etc.) (Table 3-3).
4	Set address display select switch to CONS PHY and press LOAD ADRS. The starting address should be displayed by the ADDRESS indicators.

(continued on next page)

Step	Procedure
5	Set teletypewriter LINE/OFF/LOCAL switch to LINE. This connects the teletypewriter to the processor.

NOTE

If a high-speed paper-tape reader is used instead of the teletypewriter, make sure that the device address in the Bootstrap Loader program corresponds to the device, as described in Table 3-3.

6	Place the PDP-11 Absolute Binary Loader tape in the paper-tape reader, with the special leader (a sequence of 351 punches) under the reader station. Blank leader does not work.
7	Set ENABL/HALT switch to ENABL and press START switch. The tape will be read into the memory and the processor halts when the entire program is loaded.

3.5.3 Loading the Maintenance Loader

The Maintenance Loader program, MainDEC-11-D9EA, provides an alternate method of loading diagnostic programs that can be used if the Absolute Binary Loader fails to work because of a hardware failure. This loader should only be used to load diagnostic programs if the Absolute Binary Loader malfunctions.

Use the following procedures to automatically load the maintenance loader:

Step	Procedure
1	Set ENABL/HALT switch to HALT and press START to clear the system.
2	Make certain that the PDP-11 Bootstrap Loader has been stored in memory, starting at address 017744.

NOTE

The Maintenance Loader operates in the lowest 4K page of memory. If some other page must be used, several locations must be changed as listed in Table 3-4 after the Maintenance Loader program is loaded.

3	Set starting address of Bootstrap Loader, 017744, into switch register and press LOAD ADRS.
4	Set teletypewriter LINE/OFF/LOCAL switch to LINE.
5	Place the Maintenance Loader tape in the paper-tape reader.
6	Set ENABL/HALT switch to ENABL and press START switch. The tape will be read into memory. The processor halts when the entire program is loaded.

NOTE

If the Maintenance Loader is not loaded into the lowest 4K page of memory, make location changes listed in Table 3-4 at this time.

Table 3-4
Maintenance Loader Changes*

Change Contents Of:	To:
xx7502	xx7470
xx7510	xx7474
xx7542	xx7475
xx7566	xx7475
xx7624	xx7776
xx7674	xx7474

Where xx equals:
03 for 4–8K page
05 for 8–12K page
07 for 12–16K page
11 for 16–20K page
13 for 20–24K page
15 for 24–28K page

*No changes are required when Maintenance Loader program is loaded into the lowest (0–4K) page.

CHAPTER 4
PRINCIPLES OF OPERATION

The purpose of this chapter is to introduce several concepts used in the design of the KB11-A processor and in the PDP-11/45 System. Some of these concepts are used throughout the descriptions of the KB11-A operation and implementation; other concepts are presented because they illustrate why the processor has certain features and is structured the way it is.

The concepts presented in this chapter are general in nature and they apply to many different computer systems. The specific applications of each concept in the KB11-A processor and in the PDP-11/45 System are not all described in this chapter. The reader who is primarily interested in the details of the KB11-A operation may wish to skip this chapter and read just Chapters 6 and 7; the reader who wants an overview of the processor's structure may wish to read just Chapter 5. However, many of the concepts introduced in this chapter are used throughout the succeeding three chapters and are helpful in gaining a complete understanding of the KB11-A processor.

4.1 MICROPROGRAMMING

The KB11-A processor uses a microprogram control section which reduces the amount of combinational logic in the processor. This paragraph introduces the concept of microprogramming by first describing a digital computer, then dividing the computer into various parts, and finally, describing how some of these parts differ for a microprogrammed processor.

4.1.1 Digital Computer Description

Although a computer can effect complicated changes to the data it receives, it must do so by combining a large number of simple changes in different ways. The part of the digital computer that actually operates on the data is the processor. (The KB11-A is the processor of a PDP-11/45 computer.) A processor is made up of logical elements; some of these elements can store data, others can do such simple operations as complementing a data operand, combining two operands by addition or by ANDing, or reading a data operand from some other part of the computer. These simple operations can be combined into functional groups; such a group is called an *instruction,* and it includes operations that read data, operations that combine, change, or simply move the data, and operations that dispose of the data. Instructions can be further combined into *programs,* which use the combined instructions to construct even more complex operations.

The logical elements of a processor can only perform a small number of operations at one time. Therefore, to combine operations into an instruction, the instruction is divided into a series of operations (or groups of operations that can be performed simultaneously). The processor does each part of the series in order. One way to describe how the processor executes an instruction is to call each operation (or group of operations) a *machine state.* An instruction then becomes a sequence of machine states which the processor enters in a specific order.

The processor can be completely described in terms of machine states by listing all the machine states in which the processor can perform (i.e., all the different operations or groups of operations that it can perform) and all the sequences in which these machine states can occur. The sequence of machine states is determined by the current state of the computer; this includes such information as the instruction being executed, the values of the data being operated on, and the results of previous instructions.

In terms of the machine state description, the processor can be divided into two parts. The first part, called the *data* section, includes the logic elements that perform the operations which make up a machine state. The second part, called the *control* section, includes all the logic that determines which operations are to be performed and what the next machine state should be. The data section and control section are discussed in the following paragraphs.

4.1.2 The Data Section

Figure 4-1 is a simplified block diagram that shows the divisions of the processor in a digital computer. During each machine state, the data section performs operations selected by signals from the control section. The data section provides inputs to the control section which help determine the next machine state; the data section also exchanges data with other devices external to the processor.

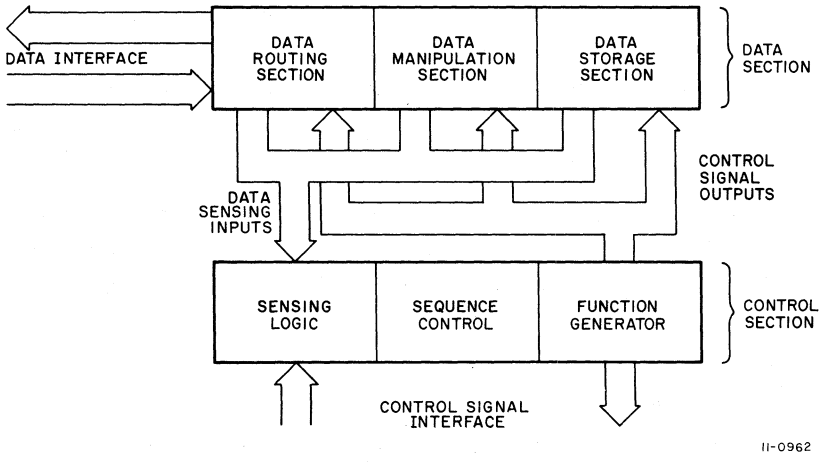


Figure 4-1 Simplified Processor Block Diagram

The data section can be divided into three functional sections; each section is discussed in one of the following paragraphs.

4.1.2.1 The Data Storage Section – For the processor to combine data operands it must be able to store data internally while simultaneously reading additional data. Often, a processor stores information about the instruction being executed, about the program from which the instruction was taken, and about the location of the data being operated on, as well as a number of data operands. When the processor must select some of the internally stored data, or store new data, the control section provides control signals which cause the appropriate actions within the data storage section.

4.1.2.2 The Data Manipulation Section – This section includes the various logic elements that actually change data. Many of these elements are controlled by signals from the control section which select the particular operation to be performed. Data manipulation is performed on data being transferred between the processor and the rest of the system, and on data that remains within the processor. In some cases, the data that remains within the processor is used to control the processor by providing inputs to the sensing section of the processor control.

4.1.2.3 The Data Routing Section – The interconnections between the logic elements in the data storage section and the elements in the data manipulation section are not fixed; they are set up as required in each machine state. The control section generates signals that cause the logic elements in the data routing section to form the appropriate interconnections within the processor, and between the data interface and the data storage and manipulation sections.

4.1.2.4 The Data Section in the KB11-A – Paragraphs 5.2 through 5.6 of this manual describe the data section of the KB11-A on a block diagram level; the paragraph is divided into three subsections which correspond to the storage, manipulation, and routing sections discussed above.

4.1.3 The Control Section

The simplified block diagram in Figure 4-1 shows that the control section of a processor receives inputs, which are used by the sensing logic to help select the next machine state, from all parts of the data section of the processor. The control section also generates control signals to all parts of the data section and communicates with other parts of the computer system through control signals. The following paragraphs describe the three parts of the control section.

4.1.3.1 The Sequence Control Section – The primary control of the processor is the selection of the sequence of machine states to be performed. This is done by the sequence control section which selects the next machine state on the basis of:

a. the current machine state
b. inputs from the data section (such as the instruction type or the data values)
c. information about external events

The sequence control section maintains information about the current machine state, and receives information from the data section and the external environment through the sensing section.

4.1.3.2 The Function Generator – In each machine state, the data section performs operations selected by signals from the control section of the processor. The function generator produces these control signals on the basis of the current machine state (and sometimes, to a very limited extent, on inputs, from the sensing section, of information such as the instruction type).

4.1.3.3 The Sensing Logic – In general, the sequence control section requires inputs that select one of a limited number of machine states to follow the current state. Because the conditions used to distinguish which state should follow the current state may be different for different current states, the sensing section acts as a selector to provide only the currently-needed inputs to the sequence control section.

4.1.3.4 The Control Section in the KB11-A – Paragraphs 5.7 and 5.8 of this manual describe the control section of the KB11-A processor on a block diagram level. The function generator comprises the microprogram ROM, its output buffer, and several logic elements that generate control signals based on sensed inputs (notably through the subsidiary ROMs). The sequence control comprises the microprogram address generation logic. The

sensing section includes the various logical elements that receive inputs from the data section, especially the condition-code generator, the subsidiary ROMs, and the branch logic. The external interface includes parts of the sensing and function generation logic, while the timing module includes part of the lowest level of sequence control.

4.1.4 Microprogramming in the Control Section Implementation

This paragraph describes two methods of implementing the control section of a processor. The first method, which is called the "conventional" method for the purposes of this discussion, uses combinational networks, with many inputs combined in varying ways to produce each output. The second method, which is called "microprogramming", replaces most of the combinational networks with an array structure. The array requires a small number (approximately 10) of inputs to select the output states for a large number (approximately 100) of signals. Because the array is a regular structure, it is simpler to construct and understand, and less expensive.

4.1.4.1 Conventional Implementation – In a conventional processor, each control signal is the output of a combinational network that detects all the machine states (and other conditions) for which the signal should be asserted. The machine state is represented by the contents of a number of storage elements (such as flip-flops), which are loaded from signals that are, in turn, the outputs of combinational networks. The inputs to these networks include:

a. the current machine state
b. sensed conditions within the processor
c. sensed external conditions

The number of logical elements in the processor is often reduced by sharing the outputs of networks which generate intermediate signals needed in the generation of several control signals, or even in the generation of control signals and machine states. Unfortunately, while this reduces the size of the processor, it increases the complexity and difficulty of understanding the device because it is no longer obvious what conditions cause each signal. In addition, the distinction between the sequence control and the function generator is blurred, which makes it more difficult to determine whether improper operation is caused by a bad machine state sequence or, more simply, by the wrong control signals within an otherwise correct machine state.

4.1.4.2 Microprogrammed Implementation – The microprogrammed implementation is based on the following observation: each control signal is completely defined if its value is known for every machine state. The function generator section can therefore be implemented as a storage device: the storage is divided into words, with each word containing a bit for every control signal; there is one word for each machine state. During each machine state, the contents of the corresponding word in the storage element are transmitted on the control lines. For most control signals, the output of the storage unit is the control signal, and no additional logic is required.

The two tasks of the sequence control section are to select the next machine state, and to provide information about the current machine state to the function generator. The only information that the function generator in a microprogrammed processor requires is which word to use as control signals. Therefore, the sequence control simply provides an address that selects the correct word. The sequence control must also select the address of the next word to determine the machine state sequence. Because the next machine state is determined in part by the current machine state, information is stored in the microprogram that helps to select the next state; the microprogram word contains the control signal values and the address and sensing control information required by the microprogram address generation logic (i.e., by the sequence control).

In a microprogrammed control like the one described above, the two major portions of the control section have been simplified to regular logical structures. The function generator is entirely separate from the sequence control, so it is easy to isolate malfunctions to the microprogram storage or to the address generator. In addition,

the sensing logic is simplified, because each sensed condition is reduced to a single signal and the sensing logic selects the appropriate signals for the current machine state based on signals output from the microprogram storage. To summarize this discussion, a microprogrammed processor has a simpler, more regular, more easily repaired control structure, based on the generation of control signals from stored information, and the selection of each machine state based on information stored in the current machine state and on information from a simplified sensing section.

4.2 PARALLEL OPERATION (PIPELINING)

In a digital computer system, the processor is usually the fastest part of the system. In order to achieve the maximum speed of operation, all parts of the processor should be used as much of the time as possible. To prevent the processor from wasting time waiting for other parts of the system, the processor must make use of the external data transfer interface as much as possible. Because any one operation that the processor performs uses only part of the processor's available resources, the two considerations above require the processor to perform several operations in parallel.

In general, the sequence of operations required for each instruction uses various parts of the processor at different times. Some parts of the processor, such as the program counter, are used only during the early parts of the instruction; others, like the shift counter, are used only during later parts of the instruction. The processor can only be fully utilized if different parts of the processor can be used for parts of different instructions during the same machine state.

When the processor works on the early part of an instruction at the same time that it completes the previous instruction, this form of parallel operation is called *pipelining.* The processor attempts to make continuous use of the external data interface by fetching each word addressed by the program counter (PC) in succession (incrementing the PC during each transfer), on the assumption that the next word required will be the one following the current instruction. In the pipelining analogy, the processor attempts to fill a pipe, corresponding to the different parts of the processor used successively by each instruction, with a series of instructions.

The current instruction often requires some other words from the external storage. At times, the next instruction does not follow the current instruction because the PC has been explicitly changed by the current instruction. When either of these two conditions occurs, the processor must stop the data transfer begun after the instruction fetch, and begin a data transfer with a different address. In the pipeline analogy, this is a break in the smooth flow of instructions through the pipe; some time is lost before the pipe drains (the current instruction is completed) and can be refilled (a new instruction fetched and a transfer begun to read the word following that instruction).

A second form of parallel operation occurs in the KB11-A to further improve the utilization of the processor. Because the processor includes several types of data storage and data manipulation elements, with different interconnections, several data transfers can take place within the processor simultaneously. As an example, during the same machine state that completes an external data transfer, the processor can read a general register into a temporary storage register, and perform an addition that adds a constant to the program counter.

The use of parallel operations within an instruction reduces the number of machine states (and therefore the total time) required to execute each instruction; the use of pipelining further reduces the number of machine states required to execute a program by effectively eliminating the elapsed time between many external data transfers.

4.3 VIRTUAL MACHINES

As described in Chapter 1 (and in more detail in the following chapters), the KB11-A processor can perform many functions. The processor executes instructions and operates on data, both of which are stored in memory, and it responds to various asynchronous events.

The response to an interrupt or trap is not entirely designed into the processor. Instead, the response is controlled by a series of instructions (a program) which is selected by a simpler hardware response when the asynchronous event is detected. Often, a number of programs are required to respond to a number of events, and the scheduling, coordination, and interaction of these programs is one of the most important (and difficult) parts of programming a computer system.

In many applications, the user programs that are written for the system are treated as though they are interrupt response programs. This is done to simplify the scheduling, to allow each user program to operate with a terminal (some form of character input/output device), and to allow several user programs to operate at once. By running several programs at once, the processor can be utilized more fully than is generally possible with only one user program, which would often be waiting while devices other than the processor completed data transfer operations. With several programs to be run, the processor can be switched among the programs so that those ready to run have the use of the processor while others are waiting. The use of the processor for several programs at the same time is called *multiprogramming.*

Running programs in a multiprogrammed system presents several difficulties. Each program can be run at arbitrary times, but all the programs must be capable of running together without conflict. A failure in one program must not be allowed to affect other programs. Each program must be able to use all features of the system in a simple, easily-learned manner, preferably in such a way that the program does not need to be modified to run in a different hardware configuration.

These difficulties are overcome by providing each program with a *virtual machine.* The programmer writes his program as though it is to run by itself; the program uses any system resources (such as memory or peripheral devices), and the system provides the services necessary to support the program and coordinate it with other programs in operation. The physical hardware in the system is combined with a control, or *executive* program to simulate a more powerful hardware machine; it is for this more powerful, but abstract, machine that the programs are written.

Based on this discussion, the hardware machine and the executive program must combine to fulfill the following four major objectives of the virtual machine:

a. Mapping – The virtual machine of the program currently in operation must be assigned to some part of the hardware machine.

b. Resource management – The scheduling of programs, and the allocation of parts of the hardware machine, must be performed by the executive program.

c. Communication – The virtual machine must be able to request services from the executive program, and the executive program must be able to transfer data back and forth with the user programs.

d. Protection – The system that supports the virtual machine, and all other virtual machines, must be protected from failures in any one virtual machine.

Each of these subjects is discussed in one of the following paragraphs.

4.3.1 Mapping

Each time a program is run (or, if the multiprogramming system is running several programs in a round-robin manner, each time a program resumes operation), it has some of the system hardware allocated to it. This generally includes some part of the memory to contain the instructions and data required by the program, some of the processor's registers, a hardware stack (which is actually an area in the memory and a pointer to that area in a processor register), possibly some peripheral devices, and perhaps a fixed amount of the processor's time. All of these allocations must be made in such a way that the hardware machine can then execute the user program with a minimum of extra operations; i.e., so that the execution of the user program requires as few additional memory cycles, or additional machine cycles, as possible. Therefore, the allocation is done entirely in the hardware machine; registers in the hardware contain all the allocation (mapping) information, and all references to virtual addresses, virtual stack locations, virtual register contents, or virtual devices converted by hardware to physical references.

In a PDP-11/45 System, the mapping is done by two devices. The mapping of virtual registers into processor registers, of the virtual stack, and of the virtual program counter, is done by loading the appropriate values into the processor registers; one of two sets of general registers can be selected for the user, and the processor has a separate stack pointer for user mode, while the program counter is changed by interrupt and trap operations and by the return from interrupt (RTI) or return from trap (RTT) instructions.

The remaining mapping functions distribute the virtual memory into the physical memory. In the physical memory, many specific addresses are reserved for special functions; the lowest addresses are used for interrupt and trap vectors, while the highest addresses are used for device registers. Because all the functions that require reserved addresses in the physical memory are performed either by the physical machine or by the control program, these addresses need not be reserved in the virtual machine. Therefore, the programs written to be run in the virtual machine can use any addresses; specifically, these programs can start at address 000000 and continue through ascending addresses to the highest address needed.

In discussions of the virtual memory and the physical memory, it is often necessary to describe the addresses used to select data items within the memory. The range of addresses that it is possible to use is called the *address space.* The maximum range of addresses that can be used in the virtual machine (which in the PDP-11/45 is the maximum number that can be contained in a 16-bit word) is called the virtual address space, while the maximum range of physical addresses that can exist in the hardware system is called the physical address space (in the PDP-11/45 this can be all the addresses expressed by an 18-bit number).

If the user program is to use addresses in the virtual address space that are reserved in the physical address space, then the virtual address space must be *relocated* to some other part of the physical address space. In a multiprogramming system, several user programs, each in its own virtual address space, may be sharing the physical address space. Therefore, the relocation of the virtual address space into the physical address space must be variable; each time a program is run, it may be allocated a different part of the physical address space. The KT11-C provides the capability of varying the relocation for each user program by storing a map of the memory allocation in a set of registers.

4.3.2 Resource Management

In a multiprogramming system, each user program operates in a virtual machine that can utilize any of the possible devices or functions of the physical machine, as well as many functions performed by the executive program. The resources that exist in the system must be allocated to each user program as required, but without allowing conflicts to arise where several user programs require the same resources. The physical machine and the executive program must resolve any protective conflicts by scheduling the resources for use by different programs at different times, and must schedule the user programs to operate when the resources are available.

The management of input/output or peripheral devices is beyond the scope of this discussion, which is primarily concerned with the basic PDP-11/45 System. Within the system, the two most important resources, which require the most care and effort to control, are the memory and the processor.

4.3.2.1 **Processor Management** – The processor can only operate on one instruction at a time (this is not strictly true, as discussed in Paragraph 4.2, because of the pipelining of instructions and because of the parallel operation of the FP11 Floating-Point Processor, but these overlapping operations do not affect this discussion). When several programs are sharing the use of the processor, the processor operates on each program in turn; either the processor is shared among the programs by using periodic interrupts to allow the executive program to transfer the processor to another user program, or each user program runs to completion before the next user program begins. To share the processor on a time basis, the executive program must perform the transfer from one virtual machine to another. Each virtual machine is given control of the physical machine by loading the map of that virtual machine into the physical machine. That is, the executive program changes virtual machines by changing the contents of the processor registers used by the virtual machine, and by changing the contents of the registers in the KT11-C which map the virtual address space.

4.3.2.2 **Memory Management** – Memory management is much more complicated than processor management. If a program uses a large proportion of the virtual address space, and only a small amount of memory is physically available in the system, the program may be too large to fit into the memory all at once. Fortunately, in most programs only a small part of the program (or possibly several small parts, one for the instruction stream and one or more for blocks of data) is used at any one time. To take advantage of this fact, the virtual address space is divided into *pages* so that each page can be mapped separately. Only the pages that are in use in the current instruction are required to be in the physical memory during the execution of that instruction.

As described in Chapter 1 of this manual, a system which uses the KT11-C memory management unit to permit each virtual machine to have a larger address space than the available physical memory must also include a mass storage device to hold those parts of each virtual memory that are not in the physical memory. As a program proceeds through a sequence of instructions, it requires different pages of the virtual memory. The memory map in the KT11-C includes relocation information for each page of the virtual address space, and also includes information specifying which pages are currently in the physical memory. If the processor attempts to perform transfers with a virtual address which is on a *non-resident* page, the KT11-C stops the execution of the instruction and, through a trap function, begins the execution of a part of the executive program which transfers the required page into the physical memory and changes the map in the KT11-C to reflect the newly available page.

4.3.2.3 **Memory Use Statistics** – If it is necessary for the executive program to bring a page into the physical memory, but all of the physical memory is already in use, the executive program must remove some other page (from the same virtual machine or, in a multiprogramming system, from some other virtual machine) from the physical memory. When a page is removed from the physical memory, a copy of that page must be stored in the mass storage device; if a copy of the page is already on the mass storage device, and none of the data (or instructions) stored on the page have been changed, the writing of the page onto the mass storage device can be bypassed. Each time a page must be replaced, the executive program attempts to predict which page is least likely to be used in the future, so that it will not soon need to be moved back into the physical memory.

The KT11-C Memory Management Unit includes hardware to permit choosing the page to replace and to determine whether that page must be written onto the mass storage device. Each external data transfer performed by the KB11-A processor requires that the KT11-C Memory Management Unit convert a virtual address into a physical address. At the same time, the KT11-C keeps track of which virtual pages have been accessed and which virtual pages have been written into. The executive program operates on the assumption that pages which have been

recently accessed will also be used soon in the future. To find a page which can be replaced, the executive program looks for a page which has not been used, preferably from the address space of a user other than the current user. If there are no virtual pages currently in the physical memory that have not been accessed, the executive program looks for a page that has not been written into, to avoid having to copy a page to the mass storage device. If all the virtual pages in the physical memory belong to the current user, the executive program looks for a page that has not been used recently, again preferably one that has not been written into. By use of the hardware memory management unit and of a variety of scheduling and allocation algorithms in the executive program, the PDP-11/45 System can provide a number of user programs with virtual machines of great power and flexibility, with a minimum burden on the user program.

4.3.3 Communication

A program running in a virtual machine must be able to communicate with the executive program, to request various services performed by the executive program, or to determine the status of the system. The same type of communication can be used for communication between virtual machines, by providing inter-machine communication as a service through the executive program. The same hardware functions that provide a means for the user program to communicate to the executive program are also used by the executive program to determine the status of the user program when a trap or abort condition occurs.

The user program requests services by executing trap instructions (such as EMT, TRAP, or IOT). Abnormal conditions caused by a program failure, such as an odd address for a word data transfer, or an attempt to execute a reserved instruction, cause internal processor traps. In either case, the trap function performed by the processor serves to notify the executive program that an instruction is required.

4.3.3.1 Context Switching – The executive program must then begin executing instructions to perform the requested service or to correct the failure condition, if possible. However, in order for the hardware machine to operate on any program other than the user program, the mapping information must be changed to reflect the allocations used by the new program.

The trapping function performs the change of most of the mapping information. The contents of the program counter (PC) and the processor status (PS) registers are changed directly; the old contents are stored on a stack in memory pointed to by a stack pointer, and the new contents are supplied from locations called a trap vector. The address of the trap vector is provided by the processor and depends on the type of trap instruction or trap condition, so that for each trap instruction or condition, a different PC and PS can be supplied.

The KT11-C Memory Management Unit stores the maps for both the executive program and one user program, in separate registers. The processor indicates which map should be used to relocate virtual addresses. During the execution of instructions (as opposed to the interrupt and trap service function), the address space map to use is specified by bits 15 and 14 of the PS. These bits also specify which stack pointer register in the processor to use (there is a separate register for each virtual machine). Because the trap and interrupt service function loads the PS register with a new value, this function changes almost the entire virtual machine context directly.

The only remaining parts of the virtual machine context that require changes are the general registers in the processor. These can be changed either by saving the contents of the registers from the previous virtual machine on the hardware stack and loading new contents, or by selecting the alternate set of general registers (the processor has two sets of general registers 0 through 5). Register set selection is controlled by bit 11 of the PS register, so this method can be used in conjunction with the trap service function.

To summarize the change of virtual machines, the mapping in the hardware system includes the selection of a register set, a stack pointer, a program address (in the program counter), an address space, and a processor status. The trap and interrupt service function, which is performed by the processor as an automatic response to trap an instruction or abnormal condition, can change all of these selections as follows:

a. The program counter and processor status are changed directly.
b. Bits 15 and 14 of the PS select the new address space and stack pointer.
c. Bit 11 of the PS selects the new register set.

The mapping and selection information for the previous virtual machine is completely saved, either by remaining in unselected portions of the processor and the memory management unit, or by being stored on the hardware stack. If the selected register set is shared with other virtual machines, the register contents must be changed by an instruction sequence.

4.3.3.2 Inter-Program Data Transfers – When the new virtual machine begins executing a service program for the programmed request (if a trap instruction was executed) or abnormal condition (if a trap condition occurred), the service program must get information from the previous virtual machine. This information may define the status of the previous virtual machine after an abnormal condition occurred so that the service program can correct the condition and restore the correct status before returning control to the previous virtual machine. If the service program is performing a service, the information required from the calling program may define the specific type of service to perform, or provide the addresses of data buffers, or specify device and file names.

Most information required by the service program is stored in the calling program's address space. To get this information, and to return information to the calling program, the service program must be able to operate in the present address space and transfer data in the previous address space, at the same time. The KB11-A processor provides instructions to do this.

The special instructions that transfer data between virtual address space make use of the processor status register to specify which address space is being used by the current virtual machine, and which address space was used by the previous machine (this is identified by bits 13 and 12 of the PS). The data is transferred between the hardware stack of the current address space and arbitrary addresses of the previous address space. The calculations of the virtual address in the previous address space are performed by the processor, in the normal data fetch sequences, using data in the current address space; i.e., any index constants or absolute addresses used to generate the virtual address are taken from the current address space, just as the instructions are.

Each virtual address space is divided into an instruction (I) space and a data (D) space, as described in Paragraph 4.3.4. Each I or D space has a full set of 2^{16} virtual addresses. Therefore, the communication instructions are available in two versions; one to transfer with the previous instruction space, and one to transfer with the previous data space. A different instruction is needed for each transfer direction, as well, so there are four communication instructions: move to previous instruction space (MTPI), move to previous data space (MTPD), move from previous instruction space (MFPI), and move from previous data space (MFPD).

4.3.3.3 Returning to the Previous Context – Because all the mapping and context information for the previous virtual machine is saved when the trap and interrupt service function sets up a new virtual machine, the hardware system can resume the execution of any program at the same point that it was interrupted. This is done with a return from interrupt (RTI) or return from trap (RTT) instruction, which replaces the program counter and processor status values of the current virtual machine with the stored values from the previous virtual machine.

The processor status selects most of the mapping information, as described previously, so the return instructions completely restore the previous context.

4.3.4 Protection

The hardware system and the executive program must be protected from programming failures in each virtual machine. In addition, most systems provide protection so that no program operating in a virtual machine can take control of the system or affect the operation of the system without authorization. A third form of protection that is useful in a large and complex system is the protection of the executive program against itself. The executive program is divided into a basic, carefully written *kernel,* which is allowed to perform any operation, and a broader *supervisor,* which can not perform privileged operations, but which provides various services useful to the executive program and to the user programs.

The forms of protection provided include the different address spaces for different types of programs, a variety of restricted access modes, and restricted processor operations. The address space protection can be used with any type of program, whether operating in user, kernel, or supervisor mode. The restricted processor operations are usable only in kernel mode; supervisor mode has the same restrictions as user mode.

4.3.4.1 Separate Address Spaces – The most basic protection against modification of the executive program by a user program (or of the kernel section by the supervisor section) is the separation of the address spaces. A program operating in user mode operates in the user address space. It can not access any physical addresses that are not in that address space, regardless of their correspondence to addresses in any other virtual address space. The executive program can prevent a user program from accessing other virtual address spaces through the communication instructions (MTPI, MTPD, MFPI, MFPD) by forcing bits 13 and 12 of the stored processor status word to 1s (to reflect user mode) before executing an RTI or RTT instruction to return control to the user program. This forces the previous mode bits in the processor status register to take on user mode, just as the current mode bits are set to user mode, and the communication instructions operate only within the user address space (Paragraph 4.5).

4.3.4.2 Access Modes – Within one address space, it is often useful to be able to protect certain parts of a program from unintentional modification. This can be done by allowing the data in those addresses to be read, but prohibiting transfers into the addresses. This is known as read-only (or write-protected) access. Areas in a virtual address space that contains alterable data must permit read/write access, but areas that contain unmodified instructions may be read-only.

Another useful form of access protection distinguishes between read accesses that fetch instructions (or address constants) and any accesses that transfer data. If instructions can be accessed by the processor only as instructions, they can be executed but can not be read or transferred to any other part of the address space. This prevents the user from determining what the instructions are in order to tamper with the instruction sequence or attempt to modify the program in undesirable ways. This type of access restriction is called execute-only access.

The KT11-C Memory Management Unit provides read/write, read-only, and execute-only access modes in the PDP-11/45 System. The access mode is stored in the mapping registers along with the relocation information; in fact, when a page of the virtual address space is not in memory, a special access code that identifies the page as *non-resident* is used. The execute-only access mode is not a separate access mode, but is provided by separating the address space into two address spaces that are used for the different kinds of transfers. One address space is used for all transfers that fetch instructions, and is called the instruction (I) space, while a second address space is used for all data transfers, and is called the data (D) space. If the two address spaces are mapped separately, attempts to use the same address for an instruction and for data may address different physical locations. If no addresses in the D space correspond to the physical addresses used in the I space, then the instructions can not be accessed as data and an execute-only access mode has been achieved. This mode must be used with caution;

however, tables that are accessed by indexed address modes must be in D space and MARK instructions, which are stored on the hardware stack as data and then executed, require the stack to be in the same virtual addresses in I and D space.

4.3.4.3 **Privileged Instructions** – Certain PDP-11 instructions that affect the operation of the hardware machine must be prohibited in the virtual machine. These include the HALT instruction, which stops the physical machine and thus prevents any virtual machines from operating, the RESET instruction, which stops all input/output devices, regardless of which virtual machine they are allocated to, and various processor status change instructions. These instructions are allowed only in kernel mode so that the executive program can control the entire hardware system; they are ineffective in the supervisor or user mode. The RESET and set priority level (SPL) instructions are allowed to execute in these modes, but have no effect; the HALT instruction activates a trap function so that the executive program may stop all action for the virtual machine that executed the HALT, but continue other virtual machines.

4.4 RE-ENTRANT AND RECURSIVE PROGRAMMING

A program can generally be divided into routines, each of which performs a function that is built up from a sequence of instructions. Often the function performed by a routine is needed in several other routines, so it is desirable to be able to call the routine from many other routines in the program; i.e., the program should be able to transfer the processor to the instructions that execute the function, and then have the processor resume the execution of the instructions following the calling instruction. A routine which is called from other routines is said to be subordinate to those routines and is called a *subroutine;* the special instructions that transfer the processor to the beginning of a subroutine and that return the processor to the calling routine are called *subroutine linkage instructions.*

4.4.1 Recursive Functions

There are some procedures that are most easily implemented as a subroutine that either performs a part of the procedure and then calls itself to perform the rest of the procedure, or completes a computation and returns a partial (and finally, a complete) result. This is called *recursive* operation. The common example of a recursive procedure is one that calculates the factorial of a number (the factorial is the product resulting from the multiplication of a number, n, by all smaller numbers). The recursive procedure to calculate a factorial is as follows:

NOTE
This procedure works only if the original number is a positive integer.

a. If n is 1 or 0, return 1 as the value of factorial n.

b. If n is greater than 1, compute the factorial of n minus 1, multiply that number times n, and return that value.

For example, to compute the value of factorial 3, the procedure is to compute the value of factorial 2 and multiply by 3. However, the value of factorial 2 is the value of factorial 1 times 2. The value of factorial 1 is found by Step a to be 1, so the final result is 1 times 2 multiplied by 3, or 6. The same recursion computes the factorial of any positive integer, in n recursions for a number n.

4.4.2 Use of a Stack in Recursive Routines

When a subroutine is called recursively, the linkage information for each call (the information required to return to the calling program) must be saved during subsequent calls. Since a recursive subroutine can be called again

before it returns from the first call, the linkage information should not be stored in a fixed location; instead, it is stored in an area, with each linkage in a different location and a pointer that identifies the specific location for each linkage.

Because a subroutine must return control to the routine that called it before that routine can return control to any routine that called the latter routine, the last linkage which has not been used for a return must be the first one used; i.e., the linkages must be used in a last-in, first-out sequence. A storage area whose locations are used for last-in, first-out storage is called a *stack*; a pointer is used to point to the last entry placed on the stack, and the subroutine linkage instructions that put information on the stack (a *push* operation), or remove information from the stack (a *pop* operation), change the contents of the pointer so that it always points to the correct word for the next linkage operation.

In the PDP-11/45 System, one of the KB11-A processor's general registers is used by the subroutine linkage instructions as a stack pointer. This register is called the hardware stack pointer (SP) and it must be initialized to point to the first word in a stack area. The same stack is also used for storage of context or linkage information by the trap and interrupt service function, which is described in Paragraph 4.3.3. The traps, interrupts, and subroutine calls are all handled in the same last-in, first-out manner.

A subroutine that can be called recursively should not move data into fixed locations, because later executions of the same subroutine (before the current execution is finished) may also execute the same data transfer instructions. The best way to keep the data storage for each execution of a subroutine separate is to store the data on the stack in the same manner as the linkage information.

4.4.3 Re-Entrant Functions

Keeping the data storage separate from the program is particularly important for programs and subroutines that can be called from more than one virtual machine. If several virtual machines are executing the same program, it is desirable to have only one copy of the program in the physical memory, and to map each virtual address space into the same physical address space. However, in a multiprogramming system, one virtual machine may begin execution of a program and then be interrupted; a second virtual machine may begin execution of the same virtual program and then run out of time; the original virtual machine may resume execution and complete the program; and the second virtual machine may resume execution. The programmer can not make any assumptions about where each virtual machine stops, so the program must be capable of being *re-entered* at any time, regardless of what other virtual machines have done with the program.

Programs designed to store all their data on a stack, so that each virtual machine that uses the program simply uses a different stack, are called re-entrant programs. A different stack pointer is selected each time a different virtual machine is selected (if the executive program changes the context of the user virtual machine, to run a different user, it changes the address mapping of the stack area and the contents of the stack pointer), so each activation of a program executes the program in complete isolation from other activations by other virtual machines.

4.4.4 Indexed Addressing of Parameters

When a program or routine calls a subroutine, the calling routine may send data to the subroutine. The amount of the data to be "passed" to the subroutine may vary, as may the amount of data returned by the subroutine. By placing all the data on the stack, the amount of data becomes unimportant. The subroutine may read different data items on the stack by using the indexed addressing modes with the stack pointer as the base register. Complex subroutines may require that the last word placed on the stack (the word with the lowest virtual address, because the stack expands towards low addresses) contain the number of parameters passed so that the program does not use other data also on the stack but not intended as parameters.

4.4.5 Separate Stack and Index Pointers

Using the stack pointer as the base address for indexed addressing presents problems if the subroutine must, in turn, pass data to another subroutine. Each time the first subroutine calculates a parameter for the second subroutine, it pushes the parameter onto the stack. The address in the stack pointer changes to reflect the new data on the stack. As a result, all instructions in the first subroutine which contain index constants are invalid, because the base value that the index constants are supposed to modify has changed. It would be very difficult, if not impossible, to write a subroutine that could use different index constants as the stack pointer changes (because to remain re-entrant, the program cannot change any part of the instruction code). A much simpler solution is to separate the base register from the stack pointer by copying the stack pointer value into another general register before using the stack for any other data. This is still re-entrant because any change of virtual machine also changes the contents of (or the selection of) all the general registers.

The register commonly used as a separate index pointer is register 5. The standard method of calling subroutines in re-entrant programs uses register 5 as the index pointer, register 6 as the stack pointer, and a word on the stack (at the address contained in the index pointer) that indicates the number of parameters on the stack. In addition to providing a straightforward and completely re-entrant structure, this method is completely compatible with a similar form of non re-entrant subroutine call. The same subroutine can be called both by re-entrant programs and by simpler programs that are not re-entrant.

4.4.6 Subroutine Call Compatibility

In a non re-entrant program, the parameters passed to a subroutine are placed in-line; i.e., they are in the addresses immediately following the address of the calling instruction. The subroutine call and return instructions use a register to store the program counter value for the calling program; the value in the program counter at the time the subroutine call (jump to subroutine or JSR) instruction is executed is the address of the word following the JSR instruction. The standard register specified in JSR instructions is register 5; register 5 can be used as an index pointer while the stack is used for data storage during the execution of the subroutine. The JSR instruction does not destroy the previous contents of register 5 when it stores the return address in that register; the previous contents are pushed on the stack, and are automatically restored by a return from subroutine (RTS) instruction.

When the RTS instruction restores the program counter (PC) value stored by the JSR instruction, the calling program must have some means of bypassing the stored data to get to the next instruction. The word immediately following the calling instruction must contain the number of words occupied by the parameters. Both of these requirements can be fulfilled by placing a branch instruction in the return location; the branch instruction advances the PC so that the first word after the line parameters, and the offset in the eight least-significant bits of the branch instruction, contain the number of *words* (the offset is multiplied by 2 before use to generate a byte address) used for the parameters.

The calling sequence and in-line parameter structure used by non re-entrant routines permits the subroutine to return control to the calling routine with an RTS R5 instruction. For compatibility, the re-entrant subroutine call must also permit the same RTS R5 instruction to perform the return. However, when a subroutine has been called in a re-entrant manner, R5 points to a location on the hardware stack, not to the calling program. In addition, the space in the stack area used by the subroutine call must be released (the stack pointer must be adjusted to point to the first location after the parameter area) so that any additional information on the stack (such as a return linkage to a routine that called the routine that called the current subroutine) is accessible. Thus, the word pointed to by R5 should contain an instruction, whose least-significant bits are the number of parameters passed to the subroutine, which can adjust the stack pointer and also complete the subroutine return sequence.

4.4.7 The MARK Instruction

The PDP-11/45 System uses the MARK instruction to perform this function. The MARK instruction is dependent on the correct setup of registers 5 and 6 (the index pointer and the stack pointer) for its correct operation. It is executed after an RTS R5 instruction that loads the PC from the index pointer, and loads R5 with the old PC of the calling routine from the stack. The MARK instruction then adjusts the stack pointer and effectively performs another RTS R5 to finally return control to the calling routine.

Figure 4-2 illustrates examples of the two types of subroutine calls for a call with three parameters and the features that make them compatible to the subroutine. Figure 4-2A shows the standard non re-entrant call; after the JSR instruction has been executed, R5 and the SP (R6) point to the location shown. Figure 4-2B shows the corresponding situation after the JSR instruction for the standard re-entrant subroutine call. Note the following similarities between the two types of calls:

a. In either case, R5 points to a word that contains the number of parameters in the least-significant bits.

b. The words following the word pointed to by R5 contain the parameters in ascending order (in the illustrations, the addresses increase going from the top of the illustration to the bottom).

c. The stack pointer (SP or R6) points to the last word on the stack used in the call.

d. The first word of the stack area used in the call contains the original contents of R5.

When the subroutine executes an RTS R5 as its last instruction, the RTS and the following instruction (either a BR or a MARK instruction) return control to the address containing the next instruction of the calling routine, and restore the SP to point to the previous contents of the stack. In the re-entrant case, the RTS instruction does not restore the PC directly from R5; instead the old PC is moved from the stack to R5, to be moved from R5 to the PC by the MARK instruction.

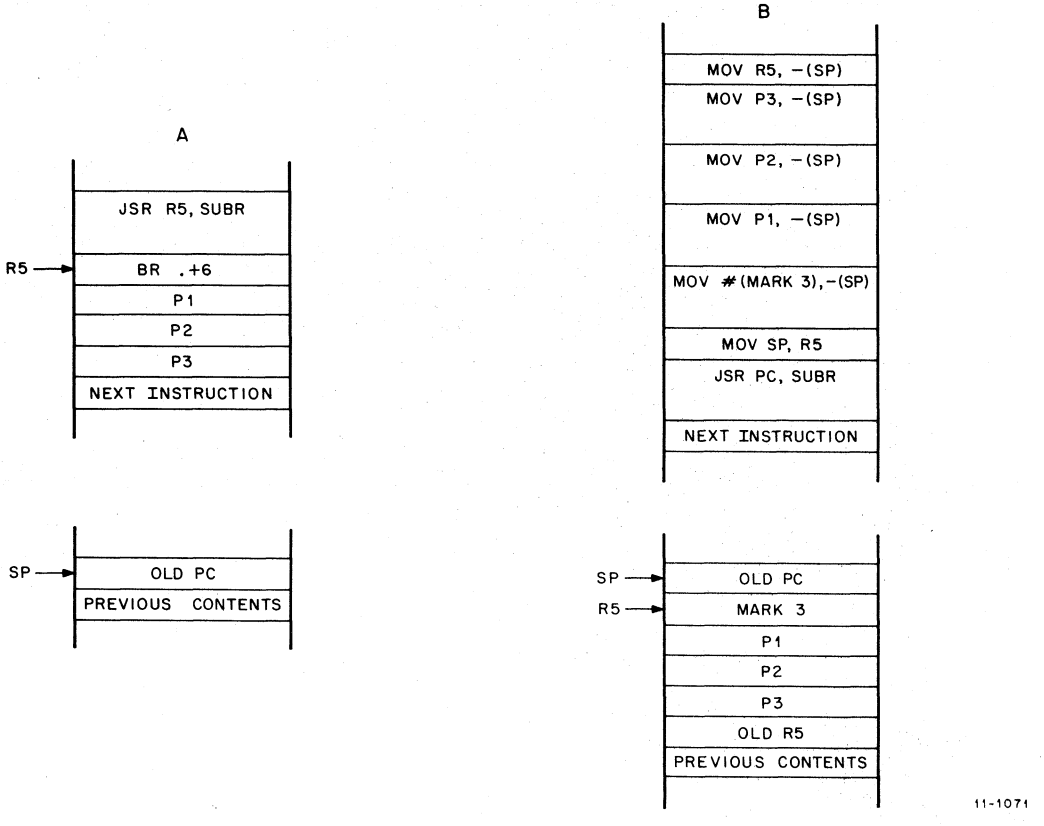


Figure 4-2 Non Re-Entrant and Re-Entrant Subroutine Calls

4.5 PROCESSOR STATUS OPERATIONS

The processor status (PS) word contains several types of information that control the operation of the processor, and of the PDP-11/45 System. Table 4-1 lists the fields within the PS, and the paragraphs of this chapter that discuss the effect of each field.

This paragraph discusses the interaction of the PS fields with asynchronous events in the PDP-11/45 System and the changes that occur to these fields as a result of those interactions. The following discussion is divided into paragraphs according to the fields of the PS word.

4.5.1 Current Processor Mode

The current processor mode selects most of the mapping for the virtual machine and determines whether certain instructions are effective or prohibited. The processor mode can be set by moving a data word to the PS at its Unibus address, or through a trap or interrupt service function (which loads a new PS value from the trap or interrupt vector), or through an RTI or RTT instruction (which restores an old PS from the hardware stack).

Programs running in virtual machines should not be allowed to change the contents of this field. If the current processor mode is changed, the mapping registers in the KT11-C Memory Management Unit that are selected are replaced by the set for the new mode. The result of attempting to continue with the same PC value in the new virtual address space is unpredictable.

The entire PS word is protected from direct transfers by being mapped only into the kernel address space. No other virtual machine has any virtual address that corresponds to the physical address of the PS register, so there is no way to transfer data to the register through instructions. The new value of the PS used during the trap or interrupt service function is taken from a vector (whose location is specified by a vector address supplied by the interrupting device or by the trap recognition logic) that is located in the kernel address space; again, other programs can not access the vector storage, and thus, can not modify the vector contents to affect the PS value. The RTI and RTT instruction can only set, and not clear, these bits, so user programs are prevented from entering other modes while kernel programs can return control to any mode.

4.5.2 Previous Processor Mode

The previous processor mode is used primarily by the communication instructions to define which address space to communicate with. During user mode operation, these bits are set to reflect user mode, so that the user program can not move data into or out of any other address space. These bits are set to reflect the value contained in the current mode bits prior to an interrupt or trap operation. A special kernel mode data transfer is used to fetch the new PS value from the vector address; however, bits 13 and 12 of the PS are not loaded from the data read but from the old value of bits 15 and 14.

During the return from a trap or interrupt service program (via an RTI or RTT instruction), the old PS value is restored from the stacked value. The previous mode bits are protected in a way that prevents user mode programs from altering the bits to allow access to other address spaces. This is done by permitting the bits to be set, but not cleared; since user mode is represented by all 1s, user mode programs can not alter these bits, but other types of programs can gain access to user address space.

Table 4-1
Processor Status Fields

Bits	Function	Description	Refer to Paragraph
15–14	current mode	select the processor operating mode: 00 = kernel 01 = supervisor 10 = not used 11 = user	4.3.3, 4.3.4, 4.5.1
13–12	previous mode	holds the processor mode that was in effect prior to the last trap or interrupt, for use in communication (MT/FP) instructions	4.3.3, 4.5.2
11	register set	selects one of two register sets for general registers 0 through 5	4.3.1, 4.3.2, 4.5.3
10–8	not used		
7–5	priority	selects one of eight processor priority levels that control scheduling of interrupt service routines	4.5.4
4	trace bit	controls operation of a trap function used in program debugging	4.5.5
3–0	condition codes	used to store information about the value of the result of the last data operation	4.5.6, 7.2

4.5.3 Register Set Selection

The register set selection field controls which of two sets of general registers is used. In general, a user program should use only the register set assigned to it by the executive program; the protection of this field is similar to that for the mode fields, so user programs should run with register set 1 selected to prevent the user from changing the selection.

4.5.4 Processor Priority

In a PDP-11/45 System, the processor spends most of its time executing instructions in programs that are running in virtual machines. However, a certain part of the processor time is spent servicing *interrupts* from other devices.

The interrupts indicate that the processor must execute an interrupt service routine to control the operation of the device; for different devices, the interrupts indicate different conditions that have occurred. Different devices can tolerate different amounts of delay before the execution of their service programs; the system uses a scheduling system to determine which interrupt service programs should be honored first.

4.5.4.1 Device Priorities – The scheduling system is based on a structure of priorities. Each device that causes interrupts is assigned to a priority level. When the processor is executing a service routine, the processor priority is set to the same level as the interrupt that started the service routine; this blocks all interrupts on the same (or any lower) priority level. Higher priority interrupts are still honored by stacking the context of the current interrupt service routine and loading a new context from an interrupt vector. The use of a hardware stack to store the context information for interrupted routines permits any number of routines to be *nested,* because each higher level routine must execute to completion and exit (through an RTI instruction) before the lower level routine resumes operation. This last-in, first-out discipline corresponds to the operation of the stack.

4.5.4.2 Program Priorities – In some cases, it is desirable to be able to reschedule part of an interrupt service routine at a different priority. This can occur, for example, when a service routine that normally executes quickly detects an error that requires a long procedure to correct; the error routine should run at a much lower priority. It is preferable to schedule the lower priority section separately, and return control to the interrupted program, so that other high-priority interrupts can be serviced without tying up stack space and other resources with the current interrupt routine.

4.5.4.3 Programmed Interrupt Requests – The same type of program scheduling is useful to the executive program for scheduling different user programs at different priority levels or for scheduling periodic supervisor functions. The KB11-A processor provides a mechanism for scheduling different priority requests, in the form of a programmed interrupt request (PIRQ) structure. This structure consists of a processor register in which bits can be set to represent interrupt requests at different priority levels, and an interrupt vector generator that supplies a fixed vector address whenever the processor honors an interrupt request from the PIRQ register. The PIRQ register is intended to be accessed only in kernel mode so that it is protected from alteration by programs operating in virtual machine; because there is only one request bit for each priority level, there must be a control program for each level that determines what other programs must be run when the request at that level is honored.

The kernel program can also vary the processor priority level directly, either by moving data containing a desired priority to the PS address, or by means of the set priority level (SPL) instruction. The SPL instruction has the advantage that it modifies only the priority level and that it can be executed with only one memory cycle, while a data transfer to the PS address requires many more memory cycles and requires additional processing to avoid changing other parts of the PS word.

4.5.5 The Trace Bit

In some forms of debugging operations, it is useful to be able to trap to a debugging program after the execution of each instruction in the program being checked. The trace trap is provided to perform this function. The trace (T) bit in the PS word generates a trace trap, through a fixed vector, whenever it is set to a 1. This trap occurs after the execution of each instruction while the T bit is set.

The T bit is protected against unintentional modification. It can only be set or cleared during the interrupt or trap response function, from a vector containing a new PS value; or during the execution of an RTI or RTT instruction, from an old PS value on the stack. When data is transferred to the PS address by any other instruction, the value of the T bit is unaffected despite any value in the transmitted data.

4.5.6 The Condition Codes

The four least-significant bits of the PS word contain the processor condition codes. These bits store information about the value resulting from any data manipulation during an instruction. The condition codes are not altered to reflect the results of address calculations, but are changed only when an instruction explicitly operates on an explicit unit of data.

The condition codes can also be set to any specific value by transferring a word containing that value to the PS address. The value of the condition codes are altered by every interrupt or trap response function, and by every RTI or RTT instruction. In addition, individual condition-code bits may be manipulated directly, with the condition-code operate instructions. These instructions provide a means to set any one or more of the condition codes with a single instruction that requires only one memory reference; a similar set of instructions can clear any one or more bits. The condition codes are used in conditional branch instructions, so the various means of manipulating the condition codes are useful because they permit setting up the PS word to respond in a particular way to various branch instructions.

4.6 STACK LIMIT PROTECTION

Each virtual machine, and the kernel mode program, has a separate stack area which is used by the hardware stack pointer (SP) for that machine. The stack pointer contains the virtual address of the last word of the stack area used to store data. As more data is stored on the stack, the value in the stack pointer changes to lower addresses.

The area available for stacked data is not unlimited. If the program continues to add data to the stack, or if an unexpectedly large number of traps and interrupts should occur, the hardware stack mechanism may attempt to store data in locations which have been reserved for other uses; this occurs if the stack pointer *overflows* beyond the boundary of the stack storage area.

In each of the virtual machines, stack overflow protection can be provided through the memory management unit. The stack area is placed in a virtual page that is not used for any other data and is isolated from other virtual addresses used by the program. The isolation required consists of an area of non-resident virtual addresses immediately below the stack area. If the stack pointer moves below the stack area, any memory references using the contents of the stack pointer as an address will be aborted and trapped to the executive program which can take corrective action.

This technique can not be used for the kernel mode stack, however, because the response to a stack overflow in kernel mode is to trap to kernel mode; the trap service operation attempts to push two additional words onto the stack. Therefore, the processor provides a warning trap when the kernel stack first overflows, and provides an emergency recovery sequence that is executed whenever the stack overflow becomes severe.

The kernel mode stack overflow detection is based on the stack limit (SL) register. The register permits the stack overflow address to be adjusted to reflect the position of the stack in the kernel address space. Whenever the processor initiates a data transfer to store data, based on the stack pointer as an address, the address that is transmitted is compared to the contents of the stack limit register. If the transmitted virtual address is higher than the contents of the SL, the stack is still within the stack storage area, and the stacking operation is permitted to proceed. If the transmitted virtual address is less than or equal to the value in the SL, a trap occurs, and the stacking operation is aborted.

The type of trap that occurs depends on the amount by which the transmitted address is less than the contents of the SL. The first 16 words directly below the stack area are reserved for stack overflow. If the stack expands into these words, a special stack overflow trap occurs. This trap uses two of the 16 words for storage, and uses a vector that initiates a special service program to recover from the stack overflow.

If, however, the stack continues to expand beyond the 16 words reserved for stack recovery operations, an emergency stack trap occurs. This trap ignores the current location of the stack and stores the current program context at addresses 0 and 2. The stack overflow program is then initiated. The 2-word emergency stack is provided to prevent the stack from continuing to advance into the prohibited area; if the stack is not adjusted to remain

within the stack storage area before expanding through the 16 reserved words, some failure of the recovery program must be suspected and the emergency measures are taken.

The 16 words reserved for the recovery program are called the yellow zone, and the stack overflow trap that occurs whenever the stack expands into these words is called a yellow zone trap. If the stack expands below the yellow zone, it enters the red zone, and the emergency red zone trap occurs. If any type of bus error or memory management error occurs while the processor is responding to a yellow zone trap, or while the processor is attempting to use the stack pointer as an address (in kernel mode), that error is treated as a red zone error because the processor may not otherwise be able to recover the correct stack information.

4.7 THE MULTIPLY AND DIVIDE INSTRUCTIONS

Two of the instructions performed by the KB11-A processor are sufficiently complex to require treatment on a conceptual level as well as on the more detailed level of the implementation used to perform them. These two instructions are the multiply (MUL) and divide (DIV) instructions.

4.7.1 Number Representation

Before describing the algorithms used for the MUL and DIV instructions, it is helpful to review some aspects of number representation that are important in the following discussions. Numbers are a means of describing *quantities.* In a number system (such as the decimal system that we normally use, or the binary system that is used in digital computers), each number has a unique *representation.* It is important to distinguish between the *quantity* indicated by a number and the *representation* of that quantity.

For example, the number system used in the PDP-11 computer systems is called the 2's complement number system. The phrase "2's complement representation" describes the use of this system. The 2's complement *representation* of the *quantity* 1 is a string of 0s followed by a single 1 in the least-significant position (for a 16-bit representation, this string is 0 000 000 000 000 001). Similarly, the *representation* of the *quantity* minus 1 is a string of all 1s (for a 16-bit representation, this string is 1 111 111 111 111 111). There is also a 2's complement *operation.* When the 2's complement *operation* is performed on the *representation* of the *quantity* 1, the result is the *representation* of the *quantity* minus 1. That is, the *2's complement* (not the representation) of 1 is -1.

Number systems like the decimal and binary systems are called *positional* representations. The same symbol, used as a different digit, has a different meaning because of the difference in position within the number. For example, the 1 in the binary number 10 has the value 2 in decimal representation, but the same symbol 1 in the binary number 100 has the value 4 in decimal representation. The value of the position, which modifies the value of the digit, is linked to the value of the *base* of the number system. Each more-significant position has a value that is equivalent to the value of the position immediately preceding it, multiplied by the base of the system. If the value of the number system base is represented by b, the values of the three least-significant digits of integer numbers, in ascending order, are 1 (actually b^0), b (that is, b^1), and b times b (that is b^2). Representing the digits of a number by the symbols a_n through a_0, the complete representation of a number is: $a_n b^n + a_{n-1} b^{n-1} + \ldots + a_2 b^2 + a_1 b^1 + a_0 b^0$. The representation consists of n+1 digits, and can express a total of b^{n+1} numbers.

If a positional representation is used only for positive numbers, it can express numbers up to $b^{n+1}-1$. However, if the representation uses a complement system to represent negative numbers, the range of numbers that can be expressed is from $-b^{n+1}/2$ to $+(b^{n+1}-1)/2$. For binary numbers, b is 2, so the range of numbers that can be represented is from -2^n to $+2^n-1$. As a result, the 2's complement operation can be expressed as finding 2^n-A, where A is the original number.

4.7.2 The Multiply Algorithm

The process of multiplication is, effectively, one of repeated addition. One number, called the *multiplicand,* is added together a number of times to form a *product*; the number of times the multiplicand is added to the product is determined by the value of the other number. That is, the multiplicand is added as many times as the value of the *multiplier.*

Using 16-bit numbers, the largest number that can be represented in the multiplier is 0 111 111 111 111 111, which can also be represented as $2^{16}-1$. To multiply a number by this quantity would require $2^{16}-1$ additions, which is too much processing to be practical. Fortunately, there is a much more efficient method that is based on the principles of positional notation, as discussed in Paragraph 4.7.1.

The multiplier can be represented as the sum of the values of the individual numbers that form the digits of the number. The multiplicand can then be multiplied by each of the digit values of the multiplier; the resulting partial products are then summed to develop the final product. Each of the partial products has the form $a_n 2^n$. For 16-bit numbers, only 16 partial products are formed, which takes much less time than 2^{16} operations. The generation of each partial product is divided into two parts: first, the multiplicand is multiplied by 2^n, and second, the resulting number is multiplied by the value of the digit, a_n.

When the digits are treated in sequence, starting with the least-significant digit and working up to the most-significant digit, the first factor used to form each partial product differs by 2 for successive bits; that is, the multiplicand times 2^4 is equivalent to (multiplicand * 2^3) * 2. Therefore, the multiplicand is multiplied by 2 before each partial product (except the first) is formed. Multiplication by 2 is the same as shifting one place to the left in binary number systems.

Each a_n can only have the value 1 or the value 0. If the value is 0, the value of the entire partial product is 0; if the value is 1, the shifted multiplicand is added to the sum that becomes the final product. Because the multiplicand is shifted for each digit of the multiplier, and the shifted multiplicand is added to the product if the corresponding bit of the multiplier is a 1, this algorithm is called the "add and shift" multiplication algorithm.

4.7.3 Sign Correction During Multiplication

The 2's complement representation permits the simplest implementation of logical circuits for addition and subtraction, but it requires corrections during multiplication and division operations. As an example of the requirement for corrections, the representation of -A is 2^n-A; when -A is multiplied by B, the actual multiplication is $(2^n-A)*B$ and the result is the representation of 2^nB-AB, instead of the representation of -AB. Therefore, a correction factor of -2^nB must be added to the result to generate the correct representation. Table 4-2 illustrates the corrections required for each combination of signs for the multiplier and multiplicand.

In the KB11-A processor, most of the correction operations are avoided by using a modified representation for the multiplier. Normally, the multiplier would be considered a 16-bit number, and the 2's complement representation of negative numbers would be 2^{16} minus the corresponding positive number. However, for use in the multiplication, a different 2's complement representation is available in which negative numbers are represented by 2^{32} minus the corresponding positive number. The advantages of this representation are illustrated by repeating the example shown in the previous paragraph: the representation of -A is now $2^{32}-A$, so -A times B is equivalent to $(2^{32}-A)*B$, or $2^{32}B$; the factor $2^{32}B$ is shifted beyond the 32-bit product, and does not appear in the final result, which is just the representation of -AB.

Figure 4-3 illustrates the conceptual hardware structure needed for this multiplication algorithm, using the special 2's complement representation for the multiplier, and illustrates the algorithm in a flowchart fashion. The

Table 4-2
Sign Corrections for Add and Shift Multiplication

	Multiplication	Representation of A	Representation of B	Product as Generated	Product Should Be	Correction
Normal 2's Complement Representation	A*B	A	B	AB	AB	none
	-A*B	2^n-A	B	2^nB-AB	-AB	-2^nB
	A*-B	A	2^n-B	2^nA-AB	-AB	-2^nA
	-A*-B	2^n-A	2^n-B	$2^{2n}-2^n$A-2^nB+AB	AB	$+2^n$A+2^nB
Special 2's Complement Representation (see Paragraph 4.7.3)	A*B	A	B	AB	AB	none
	-A*B	2^{2n}-A	B	2^{2n}B-AB	-AB	none
	A*-B	A	2^n-B	2^nA-AB	-AB	-2^nA
	-A*-B	2^{2n}-A	2^n-B	$2^{3n}-2^{2n}$B-2^nA+AB	AB	$+2^n$A

NOTES: 1. Subtracting negative numbers is the same as adding positive numbers, so the correction factors can always be generated by subtracting the appropriate variables.

2. The product is expressed in 2n bits, which can contain numbers up to 2^n-1. Any factor which is greater than or equal to 2^n can be ignored.

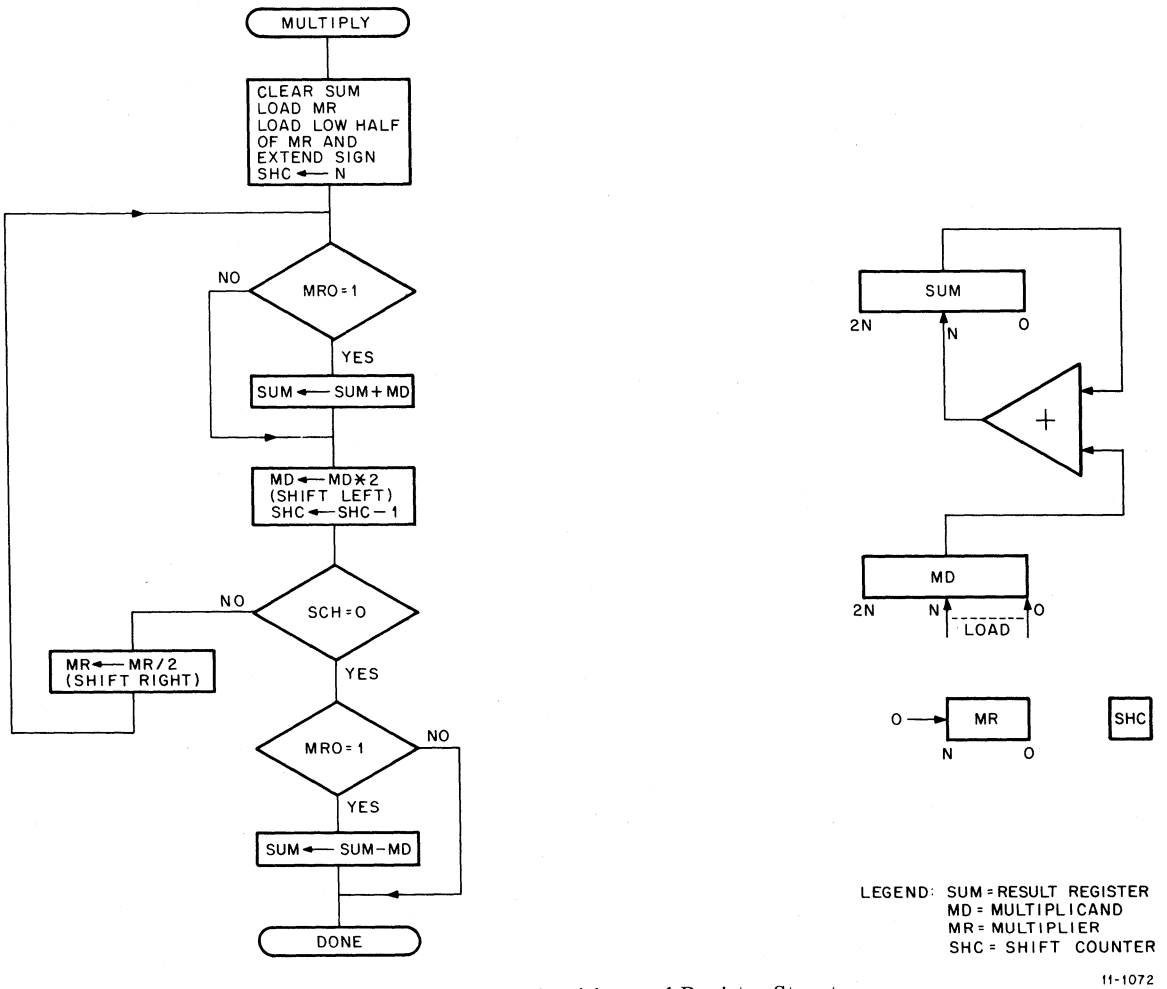


Figure 4-3 Multiply Algorithm and Register Structure

hardware structure represented in the illustration is not the structure used in the KB11-A processor; that structure is illustrated in Chapter 6. See the discussion of the MUL instruction in that chapter for more information on the implementation of the algorithm.

4.7.4 The Divide Instruction

Division is the process of counting the number of times one number (the dividend) can be reduced by another number (the divisor). The count of the number of reductions is called the quotient and the part of the dividend that can not be reduced by the divisor is called the remainder. Division is more complicated than multiplication, for several reasons:

a. Division produces two results, not one.

b. During multiplication, the maximum result occurs when the maximum number is multiplied by itself, and this result fits into two words; during division, the maximum result occurs when the largest possible number is divided by a very small number and the result does not fit into any reasonable number of words; therefore, the division algorithm must recognize the *overflow* condition when the quotient is too large.

c. During the division process, it is necessary to recognize when the partial remainder is smaller than the divisor; usually this is done by recognizing when the last reduction passed through 0 and changed the sign of the remainder. This condition is called *underflow* and requires that the results of the last reduction be restored in some way.

The simplest division algorithm is to subtract the divisor from the dividend until underflow occurs, restore the remainder, and keep a count of all but the last subtraction for the quotient (this algorithm assumes all positive numbers). This procedure is very tedious, particularly if an overflow condition exists, so a shorter algorithm is used that is based on the positional representation of numbers.

The result of the division is a quotient that can be multiplied by the divisor to regenerate the dividend (with a difference equal to the remainder). If, during the multiplication, each bit of the quotient can generate a partial product that becomes part of the total sum, then during the division, each bit of the quotient can be generated individually while reducing the partial remainder by an appropriate amount. To determine what the most-significant bit of the quotient should be, the number that is subtracted from the dividend is equal to the divisor multiplied by the positional value of the most-significant digit.

Figure 4-4 illustrates the division algorithm. At the beginning of the division, the dividend occupies all of a 2-word register. The divisor has been multiplied by 2^n, so that the number which is first subtracted from the dividend is actually the divisor, times the positional value of the most-significant bit. Before each step of the division the divisor is divided by 2, so that the correct number for generating the next bit of the quotient is formed; the division by 2 is done by shifting the 2-word divisor 1 bit to the right. In order for the division algorithm to operate with negative numbers, the reduction that is performed at each step of the division must be the correct operation to reduce the remainder; if the divisor and the partial remainder (that is, the dividend) have the same sign, the divisor is subtracted from the remainder, but if their signs differ, the divisor is added to the remainder to reduce its magnitude.

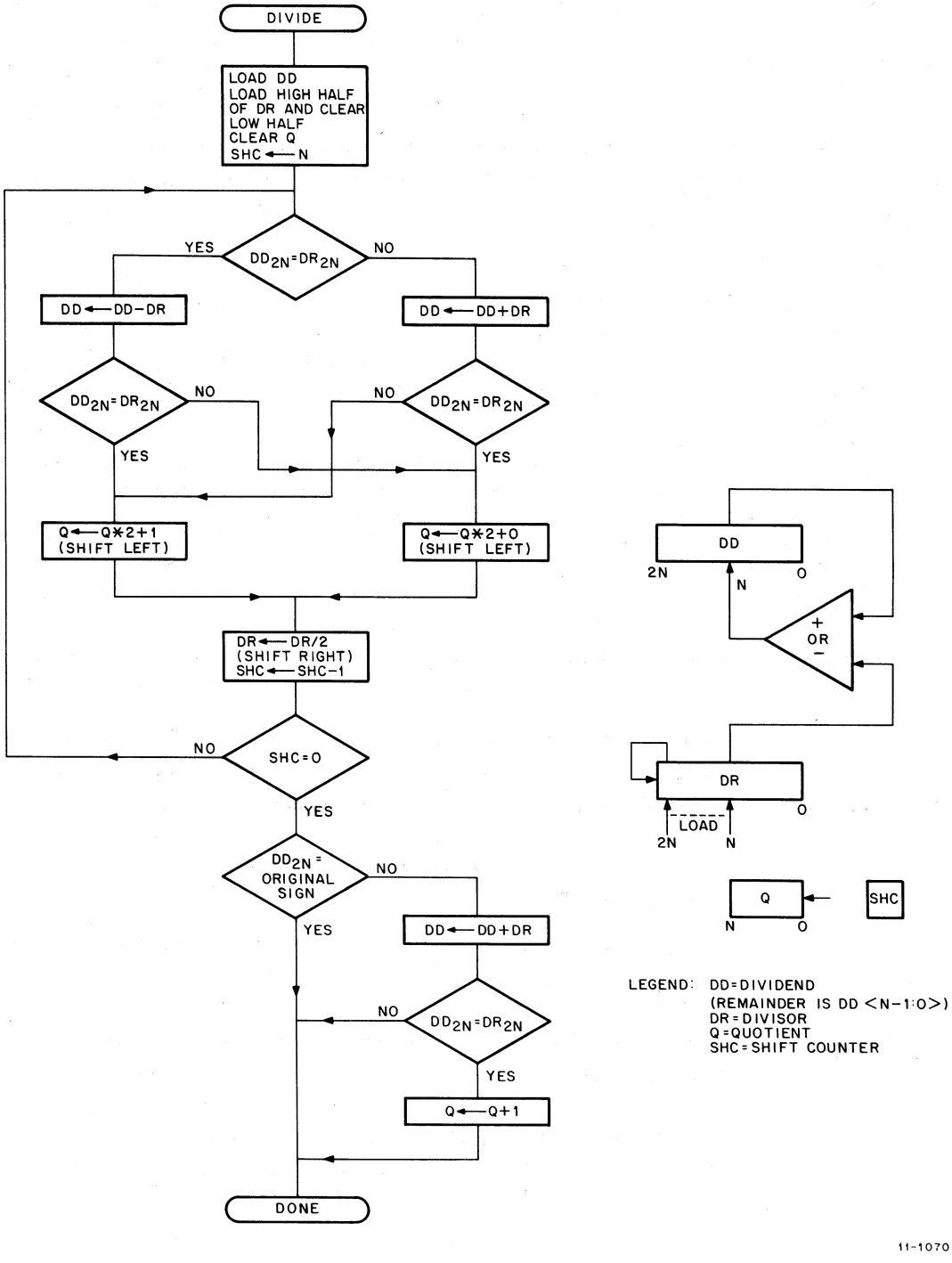


Figure 4-4 Divide Algorithm and Register Structure

The algorithm that is illustrated does not perform a restoration if an underflow condition occurs. Instead, while underflow exists, succeeding operations are performed in the opposite manner to complete the restoration; while an underflow condition exists, the bits of the quotient are set only when the underflow is corrected and are cleared if the operation does not complete the restoration. If the original divisor and dividend are of opposite sign, the quotient should be negative, so bits of the quotient are set only if underflow does occur. As a result of these considerations, the value generated for each bit of the quotient depends on the operation performed and its results, as follows:

a. If the operation was a subtraction (the signs of the divisor and the partial remainder were the same), the quotient bit is set if there was no underflow, and is cleared if there was underflow.

b. If the operation was an addition (the signs of the divisor and the partial remainder were different), the quotient bit is cleared if there was no underflow, and is set if there was underflow.

The non-restoring division algorithm works because an underflow at any step can be corrected to within one multiple of the divisor by the succeeding steps. This is true because a binary number that is represented by all 1s changes to a number that is represented by a 1 followed by all 0s when the number 1 is added to it. Therefore, the multiple of the divisor that is subtracted from the partial remainder at any step is only one more multiple of the divisor than can be expressed by all the less-significant bits of the quotient. The remaining single multiple of the divisor can be restored by a single operation (which is always an addition, because underflow exists and the divisor and partial remainder have different signs) following the steps that generate the quotient bits; this step is also used to correct the remainder.

CHAPTER 5
BLOCK DIAGRAM DESCRIPTION

This chapter introduces the KB11-A Central Processor Unit architecture by describing the block diagrams, which show all major logic elements and interconnections in the processor. The description of the processor is divided into two major sections: data paths, and control. The data paths section includes all logic elements that operate on data that is used external to the processors. The data paths block diagram is shown in Figure 5-1. The control section, which includes all logic elements that operate on data used entirely within the processor (control information), is shown on the control section block diagram, Figure 5-3. A drawing prefix, which indicates where each element is shown in the block schematic, is included within each block on the diagrams.

5.1 DATA PATHS BLOCK DIAGRAM

The data paths block diagram (Figure 5-1) includes data storage elements, data manipulation elements, and data routing elements.

The data storage elements are divided into three groups:

a. general storage registers (Paragraph 5.2)
b. temporary storage registers (Paragraph 5.3)
c. special purpose registers (Paragraph 5.4)

The data manipulation logic elements include:

a. the ALU (Paragraph 5.5.1)
b. shifter logic (Paragraph 5.5.2)
c. constant multiplexers (Paragraph 5.5.3)
d. destination register (Paragraph 5.5.4)
e. shift counter (Paragraph 5.5.5)

The data routing logic elements consist of:

a. ALU interface multiplexers (Paragraph 5.6.1)
b. temporary storage register input multiplexers (Paragraph 5.6.2)
c. external interface multiplexers (Paragraph 5.6.3)

5.2 GENERAL STORAGE REGISTERS

This group of registers includes the program counter (PC), three stack pointer registers (SP), and two sets of general registers (R0 through R5) (Figure 5-2).

5.2.1 Program Counter (PC)

The PC provides the address of the next instruction to be fetched. For some address modes, instructions that transfer data can consist of more than one word. In these cases, the PC points to each word of the instruction in

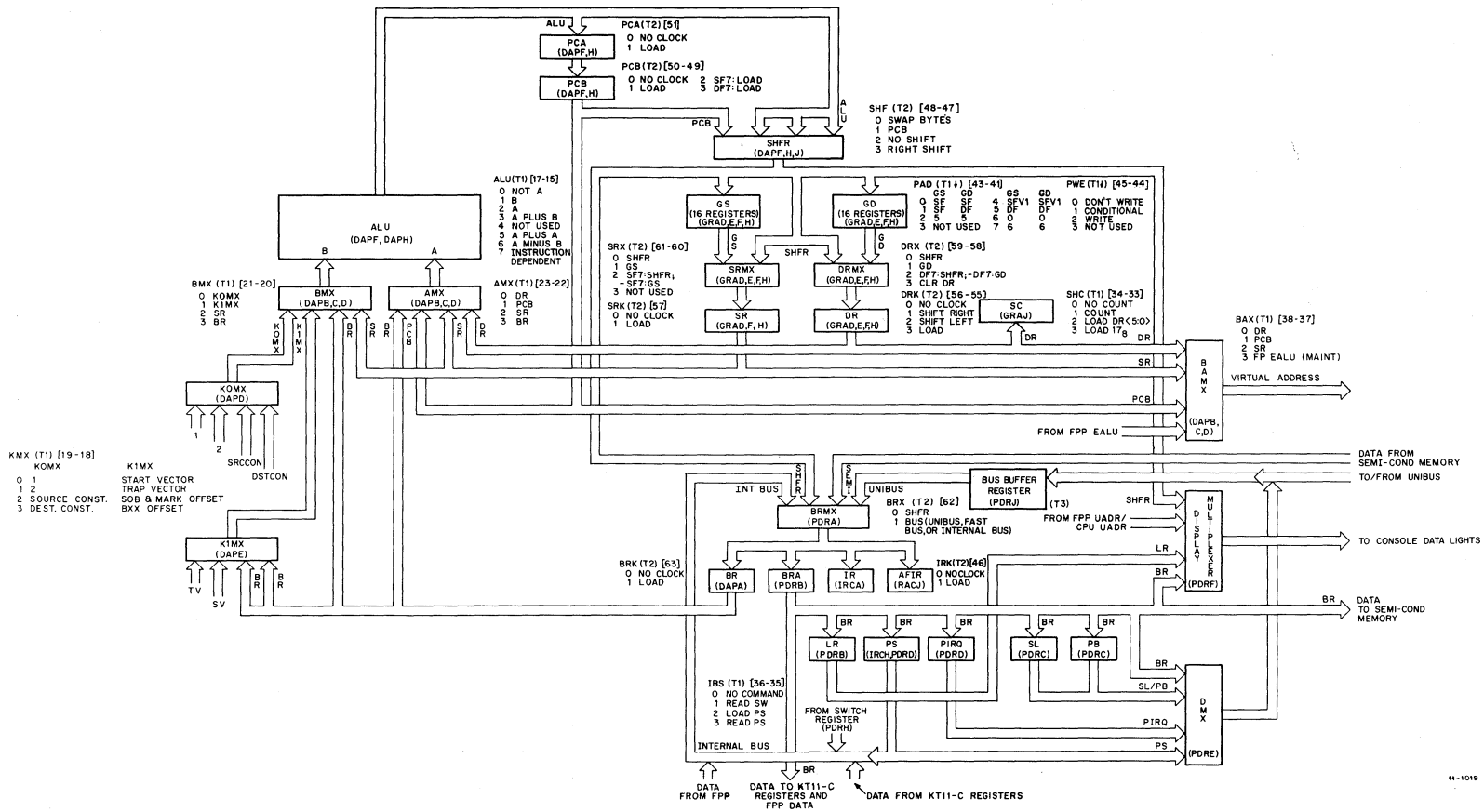


Figure 5-1 KB11-A Central Processor Data Paths, Block Diagram

the order that the words are needed. When the PC is used as an address source, the contents of the PC are gated to the virtual address lines by the bus address multiplexer (BAMX). The PC can be updated while it is being used as an address source. To accomplish this, the PC is implemented by a buffered pair of registers, so that the PCA can be loaded with a new value, while the PCB maintains the old value; the PCB can then be loaded from the PCA when the old value is no longer needed.

The processor can transfer data to the PC from any source that can supply data to the other general registers, and can transfer data from the PC to the same destinations that can be loaded from the other general registers. Specifically, all data loaded into a general register must come through the ALU, which also supplies the inputs to the PCA. The only exception to this rule is for right shifts and byte swaps. If the processor attempts to right-shift the contents of the PC, the PCA is loaded from the ALU outputs, not the shifter (SHFR) outputs, so the data in the PC is unchanged.

During the interrupt and trap service sequences, when new PC and PS values are read from locations specified by a vector address, the old PC and PS are temporarily stored in the PCB and PCA (during internal machine cycles). This is the only time that any data, other than the contents of general register 7, is stored in the PCB. However, the PCA is often loaded in parallel with the general registers so that the PCB can be loaded if the specific register used is number 7.

5.2.2 Stack Pointers (SP)

The KB11-A has three stack pointer (SP) registers. Each SP is used as the hardware stack pointer during one of the processor operating modes. The kernel, supervisor, or user mode is selected by two bits in the processor status (PS) register. All the SP registers are also addressed as general register 6. The selection of a particular SP register is performed by the general register control logic (Paragraph 5.8.3), depending on the current or previous processor state. The previous state, which is used during certain cycles of MTPI, MTPD, MFPI, and MFPD (move to/from previous instruction/data space) instructions, is determined from bits 13 and 12 of the PS, through logic in the general register control.

The SP registers are implemented in the two general register storage elements. These storage elements are the general source (GS) registers and the general destination (GD) registers. The two sets of storage elements contain duplicate copies of all the general registers except the PC. The use of the duplicate copies, and the specific addresses of the different SP registers within the storage elements, are described in the following paragraph.

5.2.3 General Register Sets

In all instructions that transfer data, each address reference specifies one of eight general registers. The specific register (of the 16 in the KB11-A processor) used for each reference depends both on the value of the 3-bit register specification and on the processor state, as represented by the contents of the processor status (PS) word.

Two of the eight general registers that can be specified in the instruction code are also used by the KB11-A as special purpose registers. If the register specification has a value of 7, it specifies the program counter (PC) register. This always refers to the hardware PC register described in Paragraph 5.2.1. If the specification has the value 6, it specifies the hardware stack pointer (SP) register. One of three hardware registers, within the general register data storage elements, is selected depending on the processor mode (Paragraph 5.2.2). If the register specification has the value 0 through 5, one of two registers is selected, depending on the register set selection bit (bit 11 in the PS word). Figure 5-2 illustrates the general register selection in the KB11-A processor.

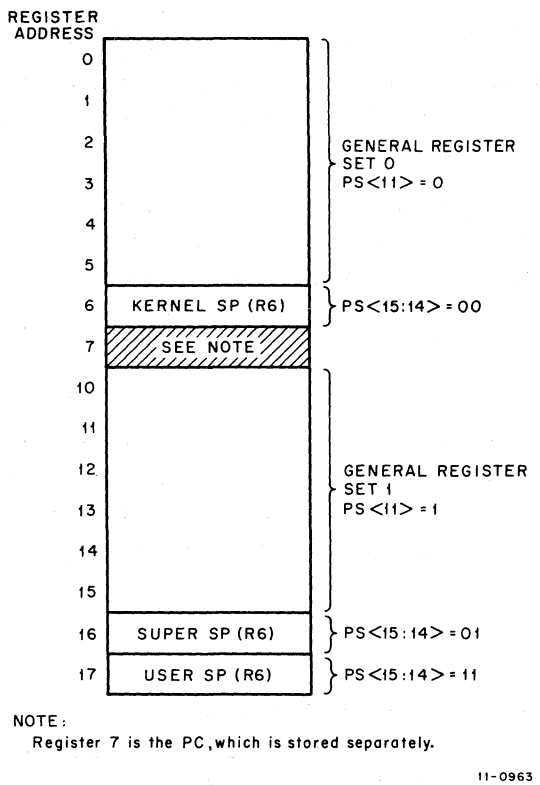


Figure 5-2 General Register Storage in GS and GD Storage Elements

Each of the 16 general registers is duplicated. The duplication allows the processor to access more than one register at a time. Each general register, with the exception of register 7, is implemented by two copies in the two general register storage elements. The general source (GS) registers include 16 registers allocated as shown in Figure 5-2. The general destination (GD) registers contain 16 registers used in an identical manner. When data must be written into a general register, it is written into both copies to ensure that all attempts to read the data will read the same value. However, by specifying different register addresses to the GS and GD storage elements, it is possible to read the contents of a different register from each. This feature is used primarily in reading the contents of the two registers specified by double operand instructions.

Whenever the general registers, as a group, serve as a data source, the PC (register 7) can be selected as one of the general registers. This is accomplished by selecting the PCB input to the SHFR, and allowing the source or destination multiplexer to select the SHFR input, if register 7 is selected, and the GS or GD input if any other register is selected.

5.3 TEMPORARY STORAGE REGISTERS

Temporary storage registers include the source register, the destination register, and the bus register. The source and destination registers are used primarily with the general register sets. The bus register is used primarily to communicate with external data handlers.

5.3.1 Source Register (SR)

The source register (SR) performs two major functions. It is the output buffer for all the general registers when addressed as the source register in an instruction, and it provides temporary storage during the source data-fetch operations.

All output from the GS registers must be transferred through the SR. When the PC is selected as a source register, the data from the PCB is routed through the SHFR to the SR. From the SR, data can be routed anywhere in the processor through the ALU inputs, or the contents of the SR can be used as an address for external data transfers through the BAMX. The SR is also used as a temporary storage register during transfers of data within the processor; e.g., when the old PC and PS are being stacked during an interrupt or trap service sequence, the SR holds the vector address.

The SR is used as a data storage element for intermediate results during instruction execution. The register and operand group instructions, such as multiply, divide, and the arithmetic shifts, use the SR to hold both operands and results.

5.3.2 Destination Register (DR)

In addition to performing two functions similar to the major functions of the SR, the destination register (DR) also operates as a data manipulation element; specifically, the DR is used as a shift register during register and operand instructions such as ASH, ASHC, MUL, and DIV.

All output from the GD registers (and from the PC, when it is selected as a destination register) must be through the DR. Data from the DR can be routed anywhere in the processor through the arithmetic and logic unit (ALU), or used as an address in external data transfers through the BAMX. To transfer the contents of either the SR or the DR to an external data storage location, the data must first be transferred from the SR or DR through the ALU to the BR, and then from the BR to the Fastbus or the Unibus.

The DR differs from the SR in its ability to act as a 16-bit, left or right shift register. This is shown in Figure 5-1 by the values of the DRK microprogram field. The DR is used as a control register and to accumulate the less-significant part of the result during register and operand instructions such as multiply, divide, or the arithmetic shifts. The DR is also the source for data to be loaded into the shift counter (SC) register.

5.3.3 Bus Register (BR and BRA)

The bus register is the data interface between the KB11-A processor and all external devices. All data entering the processor data paths, and almost all data transmitted from the processor, is transferred through the BR. The BR provides many of the inputs to the ALU and is the source of data input to all the special processor control registers.

Because of the wide utilization of the BR outputs, the BR is duplicated to reduce the electrical loading on the register outputs. The second copy of the BR is called BRA. In addition, two registers (IR and AFIR), which share the same inputs as the BR (but are clocked separately), serve to hold instructions and provide inputs to the instruction decoding circuits.

Data inputs to the processor enter the processor on one of three data buses:

a. The Unibus, which connects the processor to a variety of Unibus devices, including memories, mass storage devices, and input/output peripherals.

b. The Fastbus, which connects the processor to the high-speed, semiconductor memories.

c. The internal bus, which connects the processor to the FP11 Floating-Point Processor, the KT11-C Memory Management Unit, and some of the special purpose registers.

Any of these buses can be selected as the input to the BR by the bus register multiplexer (BRMX). The bus selected is dependent only on the physical address used in the external data transfer.

The BR can also be loaded from the processor data paths. In data transfers from the processor to an external device, or to any of the processor control registers, the data is loaded into the BR from the SHFR after passing through the ALU. The BR is used as a temporary register in the same way as the SR or DR during the execution of instructions. In particular, the BR accumulates the more-significant half of the result during multiply and arithmetic shift instructions.

The BR can provide outputs to any of the devices on any of the three data buses. Devices on the Unibus use bidirectional data lines. There are separate data lines on the Fastbus for each direction of transfer. The internal bus, which is used only for transfers into the BR, is paralleled by data lines for transfers out of the BR.

5.4 SPECIAL PURPOSE REGISTERS

The data section includes a number of special purpose registers that provide control information for use by the control section, or provide communication between the console and the processor. The majority of these registers are loaded from, or in conjunction with, the bus register (BR), and can be read into the BR via the internal bus. These registers include the instruction register, the shift counter, the processor status register, the programmed interrupt request register, the stack limit register, and the microprogram break register.

5.4.1 Instruction Register (IR)

When an instruction is fetched from an external data storage location, the data word enters the processor through the bus register multiplexer (BRMX), and is loaded into the BR. To retain the instruction word for decoding during the execution of the instruction, while releasing the BR for other data transfers that may be required during the execution of the instruction, the outputs of the BRMX are simultaneously loaded into the instruction register (IR). The IR is clocked only during data transfers that fetch instructions. The BR is clocked during every external data transfer that brings data into the processor.

To reduce the electrical loading on the outputs of each register, the IR is duplicated. The second copy of the IR is used only by the fork A logic, which has particularly stringent timing requirements, and is therefore called the A fork instruction register (AFIR). The primary instruction register (IR) is used with decoding circuits which operate the subsidiary ROMs, the B and C forks, and a variety of instruction class selectors. All the instruction decoding logic is shown on the control section block diagram, Figure 5-3, and is described in Paragraph 5.7.

5.4.2 Shift Counter (SC)

The shift counter (SC) is a register that performs a data manipulation function. However, the data loaded into the SC is used only for processor control information, and can not be transferred out of the SC.

The SC function is to count towards 0. The direction of counting depends upon the current sign of the SC contents. The control data loaded into the SC is considered a repetition count, which indicates the number of cycles required to execute a complex data manipulation, such as an arithmetic shift or a multiplication. The only indication that the processor receives of the contents of the SC is an indication that the SC does, or does not, contain 0; the counting function is completely defined once the initial count has been loaded.

5.4.3 Processor Status Register (PS)

The processor status (PS) register contains a number of individual bits. Some of these bits control the operation of the processor, while others indicate the value of the result of the last data manipulation operation.

In addition to accepting inputs from the BR (Figure 5-1) the PS receives inputs from the condition-code generation logic. In certain circumstances (the current mode field replaces the previous mode field), some bits of the PS also receive inputs from other bits of the PS. The outputs from the PS during data transfers can be directed to the processor data paths through the BR (by selecting the PS inputs to the internal bus (IBS) and the IBS inputs to the BRMX), or directed to the Unibus through the PS inputs to the Unibus A data multiplexer (DMX). The IBS path is used only for data transfers that implicitly select the PS, such as the stacking operations during interrupt and trap service sequences. When the PS is addressed specifically, the data is transferred on the Unibus, even if the transfer is to the processor data paths (through the BR).

The specific bit utilization in the PS is detailed in Table 5-1. See Chapter 7 for a detailed description of the control functions performed by the PS, and the loading and reading control logic that supports the register.

Table 5-1
Processor Status Word Bit Assignments

Bit	Name	Utilization
15–14	Current Mode	Specifies the current processor mode as follows: a. When PS ⟨15:14⟩ = 00, the processor is in kernel mode; all operations are legal. b. When PS ⟨15:14⟩ = 01, the processor is in supervisor mode; HALT, RESET, and SPL instructions are illegal, and the SUPER address space is used. c. When PS ⟨15:14⟩ = 11, the processor is in user mode; HALT, RESET, and SPL instructions are illegal and the USER address space is used.
13–12	Previous Mode	Specifies the processor mode prior to the last trap, interrupt, or loading of the PS; the values are the same as for the current mode.
11	Register Set	Specifies which general register set is used; if PS11=0, register set 0 is selected; if PS11=1, register set 1 is used.
10–08	Unused	Unused
07–05	Priority	Set the processor priority; this priority determines which levels of programmed and external device interrupt requests are honored.
04	Trace	When PS04=1, the processor traps the trace trap vector address; after each instruction fetch; this facility is used for debugging programs.
03	N	This bit is set whenever the result of the last data manipulation is negative.
02	Z	This bit is set whenever the result of the last data manipulation is 0.
01	V	This bit is set whenever the result of the last data manipulation is incorrect because of an arithmetic overflow.
00	C	This bit is set whenever a carry (generally out of the most-significant bit) occurs during a data manipulation.

5.4.4 Programmed Interrupt Request Register (PIRQ)

The programmed interrupt request register (PIRQ) allows a program to schedule the execution of various subprograms according to a priority scheme, at the same time allowing various levels of hardware interrupt priority to interact with the software priority levels. The register stores interrupt requests set by transferring request data to the PIRQ, and provides information about the requests through encoded data transferred from the PIRQ.

Data is transferred to the PIRQ through the BR whenever the processor recognizes that the physical address is the address assigned to the PIRQ (address 777772). The transfer is entirely internal to the KB11-A processor. The contents of the PIRQ are then output into the priority arbitration logic of the processor, which uses the information from the PIRQ with information from the Unibus and the PS priority level to determine when requests should be honored.

The data in the PIRQ can be transferred to other devices or to other registers in the processor by generating the physical address of the PIRQ during an external data transfer. Because the only outputs from the PIRQ are to the DMX (Unibus A data multiplexer), all transfers which read the PIRQ must be Unibus A data transfers.

5.4.5 Stack Limit Register (SL)

The KB11-A processor performs hardware stack operations, as described in Chapter 4. Because the number of locations occupied by a stack is unpredictable, some form of protection against the stack expanding into locations containing other information must be provided. The basic form of protection is the address relocation provided by the KT11-C Memory Management Unit; however, if the processor is operating in kernel mode with the address relocation inhibited, the processor provides for stack overflow detection through the use of the stack limit register (SL).

The SL is an 8-bit register that is loaded from the eight most-significant bits of the BR whenever the SL is selected by the physical address generated in an external data transfer. This requires the bus address 777775_8 during a byte transfer, or the address 777774_8 during a word transfer. The data is transferred directly from the BR to the SL; no bus operations are required. To read the contents of the SL, however, the SL must be selected by the DMX and the data transferred from the Unibus to the BR. This requires a Unibus data transfer operation. Although the SL and the PB registers share a common DMX input, each register uses a different set of eight data lines, and only one set is selected at a time. Therefore, when the SL is transmitted on the eight most-significant data lines, all 0s are transmitted on the eight least-significant data lines.

5.4.6 Microprogram Break Register (PB)

The microprogram break register (PB) is intended for use as a maintenance tool. When the processor is being operated under the control of the maintenance card, the processor can be halted during any specific microprogram state by setting the address of that state in the PB and setting the switches on the card to the proper positions. During normal operation of the processor, any value can be loaded into the PB without affect on the operation of the processor. The specific procedures are detailed in Chapter 4 of the *PDP-11/45 System Maintenance Manual.*

The PB is loaded directly from the BR whenever the PB address is generated during an external data transfer. The PB is an 8-bit register that is loaded from the eight least-significant bits of the BR. When the PB is read, the data must be transferred through the DMX to the BR by a Unibus A data transfer operation. Refer to Paragraph 5.4.5 for a description of how the DMX inputs are shared by the SL and the PB. The PB is selected by physical address 777770.

5.4.7 Console Switches (SW) and Light Register (LR)

The light register (LR) and the console switches (SW) are not, strictly speaking, data storage elements, but are included in this paragraph because they act as a data sink and a data source, respectively.

The console switches are a form of input to the processor. When an external data transfer with the physical address 777570 attempts to transfer data into the processor, the value set in the SW is transferred to the BR on the internal bus. The LR is a form of output from the processor. Any attempt to output to the same physical address transfers the contents of the BR to the LR, which can be displayed in the console lights. There are no connections between the LR and the SW, so data stored in one can not be retrieved from the other. Although both input and output to the physical address is successful, there is no correspondence between the values output and the subsequent input data.

5.5 DATA MANIPULATION

The major data manipulation elements in the KB11-A processor are the arithmetic and logic unit, with the accompanying constant multiplexers, and the shifter. In addition, two registers perform specific data manipulation operations.

5.5.1 Arithmetic and Logic Unit (ALU)

The primary data processing element in the KB11-A (in fact, the only element that can combine two operands to form a result) is the arithmetic and logic unit (ALU). The ALU can perform a variety of arithmetic operations on two variables, such as addition or subtraction, and can perform a variety of logical operations on one or two variables, such as complementing or ANDing. The specific operation performed at any time is selected by the processor control on the basis of the microprogram word and the current instruction. The manipulated operands are selected by two multiplexers, one for each of the ALU inputs. The operands can be the contents of the SR, the DR, the BR, the PCB, or one of a variety of numbers generated by the constant multiplexers.

The output of the ALU passes through the shifter, and can then be routed to any of the general registers, or to the SR, the DR, or the BR (and the IR, although this is not used). All of these destinations for manipulated data are internal to the processor; when data is transferred out of the processor, it must go through the BRA. Note that when the ALU outputs are routed to the program counter (PC), the signal paths do not pass through the shifter; this means that when certain shift or byte-swap operations are attempted with register 7 as the destination, the data that enters the PCA is unchanged. For example, an ASR PC instruction is executed as a TST PC instruction.

5.5.2 Shifter (SHFR)

In general, the data operand formed by the ALU is routed through the shifter (SHFR) to its ultimate destination. The SHFR can perform right-shift or byte-swap operations on the data, or substitute the contents of the PC for the ALU outputs. In many cases, where an instruction is performed for an odd-byte destination operand, the data manipulation required by the instruction is completed in the ALU and the transfer of the result to the odd-byte data lines is performed in the SHFR, all during one machine cycle.

In addition to its data manipulation (shifting and byte swapping) activity, the SHFR is used as a routing element. When a general register is transferred to the SR or DR, if that register is register 7 (the PC), the PCB is routed through the SHFR to the SRMX and DRMX.

5.5.3 Constant Multiplexers (K0MX, K1MX)

The constant multiplexers (K0MX, K1MX) are primarily routing elements, but they can perform certain limited data manipulation operations. The source and destination constants which can be selected by the K0MX are numbers generated by the processor on the basis of the instruction type. These numbers are used to add or subtract from addresses during the data fetch sequences. The offsets generated by the K1MX are formed from the contents of the BR by shifting and sign-extending the least-significant bits of the data word.

5.5.4 Destination Register (DR)

The destination register (DR) is primarily a temporary storage register (Paragraph 5.3.2); however, it is also used to manipulate the less-significant half of a 2-word operand by performing shifts on the operand. A word of data that is stored in the register can be shifted one bit to the left or right. The bit that is shifted into the register to fill the vacated bit position is generated by special logic in the processor, based on the data in the more-significant word being manipulated and on the instruction type.

5.5.5 Shift Counter (SC)

The shift counter (SC) performs incrementing and decrementing operations on data loaded into it during the execution of certain instructions. This register is primarily a processor loop counter register; its data manipulation capability is a function of its utilization and can not be used for data operands because the SC can not be read.

5.6 DATA ROUTING ELEMENTS

When the processor performs an operation on data operands, the operation is defined by the selection of the data operands, the storing of the result, and the manipulation of the operands. While the last function is performed by the data manipulation elements, the first two functions are performed by the data routing elements.

Data routing is performed in two ways. First, the selection of inputs to storage and manipulation elements is performed by a variety of multiplexers. Second, the loading of data storage elements is controlled to select which elements are loaded at any time. Therefore, all operand selection is performed by multiplexers, and all result storage is performed by generating load signals only for the desired storage elements.

This paragraph describes the multiplexers, which are the data routing elements in the KB11-A processor. The loading of data storage elements is described in Paragraphs 5.2 through 5.4 The multiplexers are organized in the following three groups:

a. ALU interface multiplexers (Paragraph 5.6.1)
b. temporary register input multiplexers (Paragraph 5.6.2)
c. external interface multiplexers (Paragraph 5.6.3)

5.6.1 ALU Interface Multiplexers

The ALU has two sets of inputs and one set of outputs. Each input is connected to a number of data storage (or manipulation) elements by a multiplexer, and the output is passed through a data manipulation element that acts as a multiplexer. One of the input multiplexers can select inputs from two other multiplexers. Table 5-2 lists the inputs and outputs for each of the five multiplexing elements that control the flow of data through the ALU.

5.6.2 Temporary Storage Register Input Multiplexers

Each of the three temporary storage registers (SR, DR, and BR) receives inputs through a multiplexer which selects one of two or four inputs. Table 5-3 lists the inputs and outputs for each multiplexer.

Table 5-2
ALU Interface Multiplexers

Multiplexer	Output To	Input From	Type of Input
AMX	A input of ALU	source register	variable operand
		destination register	variable operand
		bus register	variable operand
		program counter	variable operand
BMX	B input of ALU	source register	variable operand
		bus register	variable operand
		K0MX	constants
		K1MX	constants and sign-extended operands
K0MX	BMX	1	fixed constant
		2	fixed constant
		source constant	generated constant
		destination constant	generated constant
K1MX	BMX	trap vector	generated constant
		start vector	fixed constant
		BR (SOB & MARK)	shifted and sign-extended operand
		BR (branch)	shifted and sign-extended operand
SHFR	general registers, SR, DR, BR, Disp.	ALU	variable operand (can swap bytes or perform right shift)
		PC	variable operand

Table 5-3
Temporary Storage Register Input Multiplexers

Multiplexer	Output To	Input From
SRMX	source register	general source (GS) registers
		shifter (SHFR)
DRMX	destination register	general destination (GD) registers
		shifter (SHFR)
BRMX	bus register	shifter (SHFR)
		Unibus (via PDRJ Bus Buffer register, clocked at each T3)
		Fastbus (SEMI)
		internal bus (IBS)

5.6.3 External Interface Multiplexers

The KB11-A Central Processor Unit external interface is divided into three parts:

a. the explicitly addressed interface
b. the implicitly addressed interface
c. the display interface

The explicitly addressed interface is used in all data transfers where the address is specified by the processor. The address is supplied to the interface through the bus address multiplexer (BAMX), from one of three sources. These sources are the PC and the two temporary registers, SR and DR, that are used as buffers for the general

registers. In addition to these inputs, the BAMX can select an input from the exponent arithmetic and logic unit (EALU) of the FP11. This input is used only to allow this data to be displayed in the console lights during specific machine states when executing floating-point instructions. Data is supplied to the interface through the data multiplexer (DMX) for Unibus transfers, and directly from the BR for Fastbus transfers or internal bus transfers. On data transfers into the processor, one of the three buses is selected by the BRMX.

Implicitly addressed transfers (e.g., to the FP11) do not require sending an address. The data is transmitted by sending a load signal to the appropriate device, or to a register in the processor if the transfer is into the processor. Data is transferred on the internal bus. The internal bus is, therefore, a form of data routing element; selection is accomplished by gating specific data onto the bus from a device, and by loading only specific registers. The display interface selects the data that is to be displayed in the console lights. There are two sets of lights which display program-dependent data; the DATA lights and the ADDRESS lights. The DATA lights display one of four data words selected by the display multiplexer; the ADDRESS lights display addresses based on the outputs of the BAMX. Both displays are controlled by switches on the console; note that the ADDRESS display is also affected by the KT11-C option, if it is implemented.

5.7 CONTROL SECTION

The control section of the KB11-A processor determines the sequence of operations performed by the processor, and controls the interaction of the processor with other devices in the PDP-11 System. Control of the processor is based on the signals generated by the microprogram read-only memory (ROM), while the control of processor-system interaction is performed primarily by asynchronous circuits on three control modules.

The control elements shown on the control section block diagram, Figure 5-3, are divided into three groups:

a. the microprogram ROM, together with the ROM address generation logic and the ROM output buffer logic

b. the external interface, which comprises the UBC, TMC, and TIG modules shown at the bottom of the drawing

c. the combinational logic circuits that interact with the microprogram outputs and with data from the data paths section of the processor to generate some of the processor control signals

Each of these three groups is described in the following paragraphs.

5.7.1 ROM Microprogram Control

The microprogram ROM, shown in Figure 5-3, contains 256 stored processor control words. For each processor machine cycle, one of these stored words is output to the data paths section and to the other processor control circuits. The ROM word is divided into fields, and each field controls a different (but always the same for a given field) part of the processor. A review of the concept of microprogramming is provided in Paragraph 4.1. In Figure 5-3, each control field is listed by a mnemonic name and by bits of the microprogram word occupied by the control field. The control selection that is made, or the action that takes place for each value that can be stored in the field, is listed under the field name. Where possible, the field name and description are placed next to the logical element controlled by that field. For example, Figure 5-4 illustrates the B multiplexer as shown on the block diagram, with the BMX control field description to its left.

The microprogram ROM outputs that control other parts of the processor must be stored in a buffer register, so that the next microprogram word can be selected while the current word is being used. Therefore, a ROM buffer register (RBR) is provided for these outputs. The three output fields that are used to select the next microprogram word (FEN, BEF, and ADR) are not buffered because they are used immediately and the resulting address is buffered.

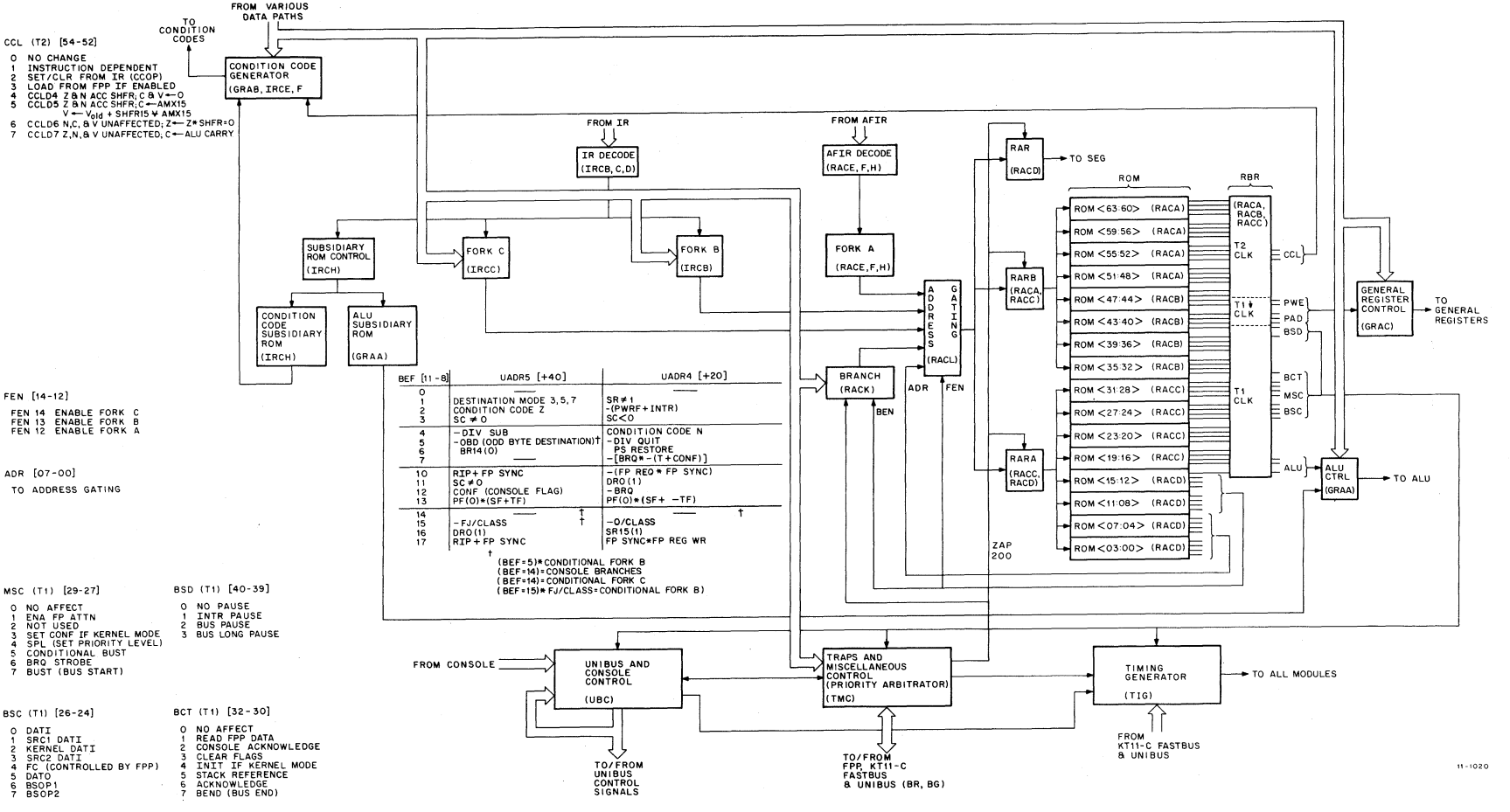


Figure 5-3 KB11-A Central Processor Control Section, Block Diagram

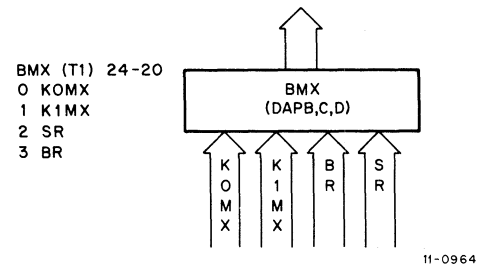


Figure 5-4 Control Field Description Example

Immediately after the beginning of a machine cycle, when a new microprogram word is available, the ROM-address generation circuits begin the calculation of the next ROM address to be selected. This corresponds to selecting the next machine state. The generated address is assembled by the address gating logic and loaded into the ROM address register (RAR). There are three copies of the RAR to accommodate the output loading required for 16 ROM elements, and to transmit the ROM address to the KT11-C paging unit (when it is implemented).

The address gating logic assembles the address from five sets of inputs. The basic input, which is always present, is the address (ADR) field of the current microprogram word. The ADR is ORed with the outputs of the branch logic, which is controlled by the BEF field of the microprogram word. The branch control logic selects a set of condition inputs from signals received from the processor data paths, the condition codes, and from the processor interface modules (specifically, the TMC module). Depending on the state of the selected inputs, the branch control generates one or two signals that are used to modify the ADR.

The three other inputs to the address gating circuits are from the fork logic. The three forks are similar in implementation and purpose. Each fork uses combinational logic to decode the instruction type and a variety of processor conditions, and generates one of a large number of addresses that is combined with the ADR input by masking. Each fork can be enabled by one bit in the fork-enable (FEN) microprogram field; normally all forks are disabled. No more than one fork is ever enabled at a time.

The fork A logic, used to select the machine state to follow an instruction fetch, requires a separate instruction register (AFIR) because this fork must operate rapidly and therefore puts a heavy load on the IR outputs. The B and C forks decode inputs from the primary IR and use the outputs of a subsidiary ROM, which decodes some classes of instructions. These forks are used after a destination operand fetch and a source operand fetch, respectively.

To summarize the operation of the microprogram ROM control logic, during each machine cycle, an address is assembled from any enabled fork combined with the address field of the microprogram word and any enabled branches. This address is loaded into the ROM address register to select a new microprogram word. At the beginning of the next machine cycle, the new microprogram word is loaded into the ROM buffer register and the sequence is continued.

5.7.2 External Interface Control

The interaction between the KB11-A processor and the other parts of the PDP-11/45 System is controlled by three modules in the processor. These modules include the asynchronous circuits that perform timing adjustments, the circuits that generate and receive interlocking bus control signals, and the basic processor timing circuits. The functions of each module are discussed in one of the following paragraphs.

5.7.2.1 Unibus and Console Control (UBC) Module – The Unibus and console control (UBC) module includes the circuits that control transfers with external devices (and some processor special registers), and the circuits that allow the processor to be controlled by the console. The data transfer control circuits perform the necessary operations to gain control of the required data buses, select the address that is to participate in the transfer, and complete the transfer. The console control circuits provide information to the branch control circuits so that the microprogram control can be used to execute various console operations.

5.7.2.2 Traps and Miscellaneous Control (TMC) Module – This module is used to recognize a variety of asynchronous conditions and change the sequence of processor operations in response to these conditions. The TMC module detects various abnormal conditions within the processor, such as power failure, odd address on word transfers, stack overflow, or reserved instructions. When any of these conditions occur, the processor enters a trap service sequence of microprogram states, and the TMC module generates a trap vector that is used to transfer system control to a specific trap service program. The TMC module can also handle a variety of trap-type instructions, which are legal in programs that use them in a defined manner and that have set up the trap vectors for those instructions.

The TMC module also performs priority arbitration for Unibus A, which is controlled by the KB11-A processor. The priority arbitration determines which device shall be bus master, based on the priority level of the bus or non-processor request, and the priority level of the processor. The processor normally assumes the role of bus master when no other device is requesting the bus; the processor must be bus master in order to perform any data transfer on Unibus A. Fastbus transfers can be performed even though the processor is not Unibus A bus master. One of the devices that can request bus mastership, but only to perform an interrupt operation, is the processor's programmed interrupt request (PIRQ) register.

5.7.2.3 The Timing Generator (TIG) Module – The timing generator (TIG) module controls all timing of operations within the processor. All register loading, all data path transfers, and all microprogram word selection is controlled by timing signals from the TIG module which gate the control signals to the respective processor elements. The TIG module contains the processor clock, the time pulse generators that produce timing signals from the basic clock output, and a variety of control circuits that can stop and restart the clock based on asynchronous conditions detected by the UBC and TMC modules. The timing of the processor operations thus interacts with the timing of data transfers in the PDP-11/45 System, and with the console control operations.

5.8 SPECIAL CONTROL LOGIC

There are three special control circuits in the processor which use combinational logic to increase the flexibility of the processor control. Two of these circuits use subsidiary ROMs to define specific operations for individual instructions, and the third performs the additional decoding necessary to control the general register sets. Each of these circuits is described in one of the following paragraphs.

5.8.1 Arithmetic and Logic Unit (ALU) Control

The arithmetic and logic unit (ALU) used in the KB11-A processor can perform 16 different arithmetic operations and 16 different logical operations. Only a subset of these operations are used in the KB11-A. The ALU control circuit transforms the ALU microprogram field (which is compressed into three bits, and can only express eight different operations) into the six control signals necessary to select the appropriate ALU operations. The ALU control circuit can also substitute control signals derived from a subsidiary ROM (whose output is selected by the individual instruction being executed) for the signals derived from the ALU field. This allows the same microprogram word to be used for the execution machine state of a large number (32) of instructions.

The subsidiary ROM is one of two used for a group of data manipulation instructions. When these instructions are being executed, the subsidiary ROM control converts the instruction type to a 5-bit address that selects one word in each of the subsidiary ROMs. This word contains the control signals that correspond to the value required for that instruction. Through the use of the control signals in the subsidiary ROM, any ALU function can be performed.

NOTE
The SHFR operation is also affected when the output of the ALU subsidiary ROM is used. See Chapter 7 of this manual for details of the effects of the subsidiary ROM on the SHFR.

5.8.2 Condition Code Control

The KB11-A processor condition codes are used to store information about the results of each instruction, so that this information can be used in following instructions. The conditions recorded in the condition-code bits differ for each instruction type, and often for the part of the instruction being executed. In addition, the sources of the information to be recorded in the condition codes can vary for different types of instructions.

The condition-code control circuit uses the CCL microprogram field and a subsidiary ROM to determine what data shall cause each condition-code bit to be set or cleared. For most machine cycles, only the CCL field is required to determine what function the condition-code control logic performs. When the CCL field contains a value that specifies that the operation is instruction-dependent, the outputs of the subsidiary ROM (which depend on the current instruction, as described in Paragraph 5.8.1) determine the exact operation.

5.8.3 General Register Control

The KB11-A general registers include two register sets that are duplicated for extra speed in reading the registers. The selection of registers within each implementation is controlled by the PAD microprogram field, and all input in the registers is controlled by the PWE microprogram field.

Because the specific register to be selected can depend on the contents of the instruction register (IR), the contents of the switch register (during a console operation), or directly on the PAD field, combinational logic is used to combine all the different sources according to the requirements of the current machine state. The combinational logic also determines, for conditional write operations, whether the register that is selected is in the general register set or is register 7 (the program counter). In the latter case, no write operation is done within the general register storage area.

CHAPTER 6
MICROPROGRAM FLOW DIAGRAMS

This chapter describes and explains the microprogram flow diagrams that are included in the KB11-A print set. These flow diagrams illustrate the operation of the KB11-A processor on a machine state level; each operation shown on the flowchart corresponds to one processor time cycle, which, in turn, corresponds to one cycle of the microprogram ROM. The information presented on the flowcharts for each microprogram cycle is described in Paragraph 6.1.

6.1 HOW TO READ THE FLOWCHARTS

The succeeding paragraphs describe the flowcharts and the ROM map according to the following three categories:

a. the operations performed by each machine state

b. the microprogram words that are associated with the machine state (i.e., the addresses of the ROM words containing information about the machine state)

c. the flow of control from each machine state to the possible successor states

The symbols used in the flowcharts can be divided in a similar manner. Each machine state is represented by a box on a flowchart. The box contains information about the operations that take place during the machine time cycle for the microprogram word represented by the box. The microprogram word is identified by a symbolic tag and an octal address directly above the box. In many cases, several microprogram words are used for the same microprogram state; the name and address of each word is shown above the box, and the contents of each word are identical (all 64 bits of each word that represents the same state must be identical). During the time cycle for each microprogram state, the KB11-A processor constructs the address of the next machine state; i.e., the processor determines the sequence of machine states. The sequence determination is represented on the flowcharts by the lines of flow; these lines show the variations in machine state sequence that can be affected by branches and forks.

6.1.1 Machine State Description

On the flowcharts, each machine state is represented by a rectangular box like the one shown in Figure 6-1. A symbolic notation is used within the box to describe the major operations that occur during the machine state, while other symbols outside the box are used to describe the machine state sequence and the correspondence of machine states and microprogram words. This paragraph describes the notation used inside the box; the remainder of the flowchart information is described in the following paragraphs.

Each box includes a comment that describes the purpose of the machine state. The comment is at the top of the box and is separated from the remaining information by a line across the width of the box. In Figure 6-1, the comment reads SUCCESSFUL BRANCH, FIX PC, which indicates that this machine state updates the program counter as the result of a decoded branch instruction with the branch conditions met.

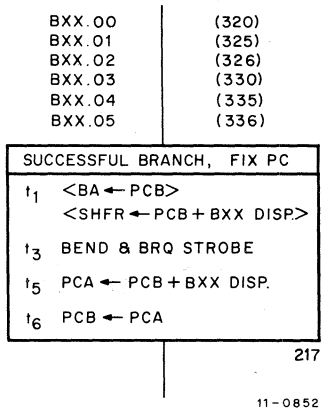


Figure 6-1 A Typical Machine State

The remainder of the symbols in the box describe data transfers, data manipulation, and timing operations. Data transfers move information between data storage and data selection elements in the processor that are represented by mnemonic names. For example, PCB is an acronym for program counter B, which is the name of a data storage element that usually contains the address of the next instruction to be executed. In a similar manner, BA is an acronym for the bus address multiplexer, which is a data selection element that selects one of four data items for use as an address during the current machine state.

Data transfers are indicated by a left arrow (this is the replacement operator of the ISP notation described in the *PDP-11/45 Processor Handbook,* DEC 1972). Data transfers to a data storage element update the contents of the element. The symbols PCB ← PCA indicate that the contents of the PCB are replaced by the contents of the PCA. Transfers to a data selection element indicate that the element selects the contents of the data source, and that the data is passed to other data elements. The symbols BA ← PCB signify that the BA selects an input which is provided from the PCB; the output of the BA is used as an address during data transfers with external devices.

Data manipulation operations change a unit of information or combine units of information. These operations are represented by arithmetic and logical symbols that correspond to the operations performed by the arithmetic and logic unit (ALU), the shifter (SHFR), and various counting and shifting registers. For example, the symbols PCB + BXX DISP. indicate the binary sum of two data words, one the contents of the PCB, the other formed by a sign-extension operation on the displacement that is part of the branch instruction word. The source data for a data transfer operation can be the information resulting from a series of data operations, as in the line PCA ← PCB + BXX DISP..

In several machine states on the flowcharts, the data manipulation operator that is shown is a dollar sign. This indicates that the specific operation performed is instruction-dependent; a subsidiary ROM provides control signals to the ALU and SHFR from a word selected by an address that corresponds to the instruction. The dollar sign is also used to indicate instruction-dependent condition-code load operations. A similar subsidiary ROM provides control signals that alter the sensing used for the condition codes.

In addition to operations on data, the processor must provide timing control for synchronization with external events. Some timing operations are used to sense external conditions which are then decoded as inputs to the machine state sequence control. The symbol BRQ STROBE indicates an operation of this type; the results of this operation affect the machine state sequence through branches on the BRQ condition. Other timing operations start or stop asynchronous cycles within the processor; BUST (bus start) and BEND (bus end) control the

external data transfer cycle, while BUS PAUSE and BUS LONG PAUSE control the processor clock that is used during machine state execution. While these two cycles must be interlocked in a specific sequence, each cycle operates asynchronously except at the occurrence of the timing operations that control the interlocking.

The machine state is the basic unit of time in processor operation because each machine state must be executed completely; i.e., sequence changes can only affect the order of execution of machine states, not the time periods within the state. However, the machine state is divided into five smaller intervals based on the processor clock. These smaller intervals provide clock signals that indicate when data may be loaded into data storage elements, and they are used in synchronization operations.

Each data transfer or manipulation operation, and each timing control operation, occurs at a particular time within the machine state. These times are indicated by lower case "t"s on the left side of the box. Only the time intervals in which significant operations occur are shown, in order from top to bottom; each interval is identified by a subscript number. In Figure 6-1, the timing operations BEND & BRQ STROBE occur in the third of five time intervals, as indicated by the t_3 at the beginning of the same line in the box; the data transfer operation PCB ← PCA occurs on the clock cycle following the last of the five time intervals, as indicated by the t_6 on the same line (t_6 corresponds to the t_1 of the next machine state unless the machine state sequence pauses for an external data transfer; in this case, the t_6 operations are not delayed).

In principle, all operations performed by the processor affect the stored information. However, some operations do not produce lasting effects; these operations are performed only to provide indications of the internal processor data during maintenance operations. In other words, these operations allow internal data to be displayed when the processor is being manually clocked using a maintenance module. These operations are distinguished by angle brackets to indicate that they do not affect any stored information. In Figure 6-1, the operations at t_1 are for maintenance purposes only.

6.1.2 Machine State Information in the ROM Map

The symbolic representation used in the flowcharts does not indicate all the operations that occur during each machine state, nor does it indicate the actual control signals generated to control the operations that are represented. A more detailed representation of the activity in each machine state is provided by the microprogram ROM map, which is reproduced in drawing K-CS-M8103-0-1 in the engineering drawing set.

The microprogram ROM contains 256 control words of 64 bits each. The ROM map lists the values of every bit in every word of the microprogram. The bits are grouped into control fields; each field controls the operation of one part of the processor as shown on the data paths block diagram, drawing D-BD-KB11-A-02. The value of the bits in each field is represented by an octal number.

Each microprogram word is represented by one line of numbers in the ROM map. Each word corresponds to a unique box on the flowcharts; however, each box on the flowcharts may correspond to several words in the ROM map. The correspondence is indicated in two ways: first, each microprogram word is assigned a symbolic name, with the names of all the words corresponding to a particular machine state differing only in the last character; second, the ROM address of each microprogram word is listed above the corresponding box on the flowcharts, along with the symbolic name. The symbolic names are also shown on the ROM map at the left end of the line representing the microprogram word.

The box illustrated in Figure 6-1 corresponds to six microprogram words. The symbolic name of the machine state is BXX.0; the symbolic names of the corresponding microprogram words add a second digit after the period that serves to differentiate among the actual ROM words. These names, ranging from BXX.00 to BXX.05, are listed to the left of the sequence flow line above the box. The ROM address of each word is shown in parentheses to the right of the flow line.

Each microprogram word contains several fields that are used to calculate the address of the next microprogram word. One of these fields contains a microprogram address. This is not the address shown on the flowchart; it is important to distinguish between the address of the microprogram word and the address in the word. The several words corresponding to a specific machine state are located at different addresses, but they must all contain the same address. The address contained in a microprogram word is shown by a number below the lower right corner of the box, or the address can be found by reading the ROM map line for any of the microprogram words corresponding to the machine state.

When the flowcharts and the ROM map are used to determine the sequence of processor operations, the flowcharts provide a summary of the important operations and a visual representation of the sequence. After gaining an overall picture from the flowcharts, examine the ROM map to learn the details of the control signals generated during each machine state and the exact addresses of the microprogram words accessed.

6.1.3 Machine State Sequence Information

Most machine states specify a unique successor state through a microprogram address in the microprogram word. However, the sequence of machine states can be varied between machine states; this allows a particular state, or sequence of states, to be used for several procedures that follow different sequences after that state. For instance, all instruction fetching is done by one sequence of machine states, followed by the sequence that is applicable for the instruction fetched.

There are two basic ways of determining the next machine state when the sequence is variable. When only a small number (four or less) of successor states is required, a microprogram branch is performed. A branch modifies the microprogram address contained in the microprogram word on the basis of a pair of sensed conditions, to create the actual address of the next microprogram word. Figure 6-2 illustrates a branch of this kind. During the BRK.0 machine state, the address 130 contained in the microprogram word is modified if the console flag (CONF) is set. The modification affects bit 5 of the address to create the address 170.

When a large number of successor states is required, the address is calculated by combinational logic in the processor and replaces a non-significant address contained in the microprogram word. This type of generalized sequence selection is called a fork, and is illustrated in Figure 6-3. While machine state FOP.3 is executed, the fork C logic calculates an address based on the destination mode and instruction type. The current microprogram word contains the microprogram address 377, which allows the fork address to take on any value. (See Chapter 7 for a description of the address calculation logic.) The forks are also called decision points; fork C is decision point 1, and is sometimes represented by the tag DPT.01 in the flowcharts.

The operation of the fork logic and the branch logic is normally mutually exclusive. Sometimes, however, it is necessary to conditionally enable the fork logic; in the example illustrated by Figure 6-4, an extra operation is sometimes necessary to move an odd byte into the even byte position before selecting a machine state sequence to operate on the data. The final selection is done by the fork B (DPT.02) logic. The fork B logic is conditionally enabled during the machine state preceding the D12.3 state; if the destination address is even (DR0 is a 0), the next microprogram state is selected from the address calculated by the fork B logic. However, if the address is odd (DR0 is a 1), the fork B logic is disabled and the address contained in the previous machine state selects microprogram word D12.30. In this word, the fork B logic is unconditionally enabled. The conditional enabling of the fork logic therefore saves one machine state cycle time when the extra state is not needed. The address contained in the machine state preceding D12.3 is 137; the condition that enables the fork also modifies this address to 177, which allows the fork B logic to generate addresses ranging from 0 to 177.

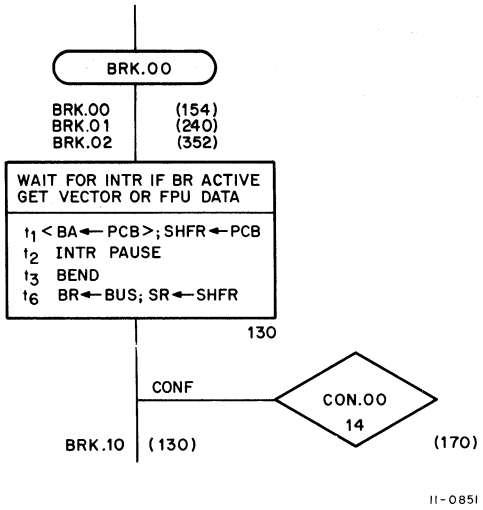


Figure 6-2 An Example of a Microprogram Branch

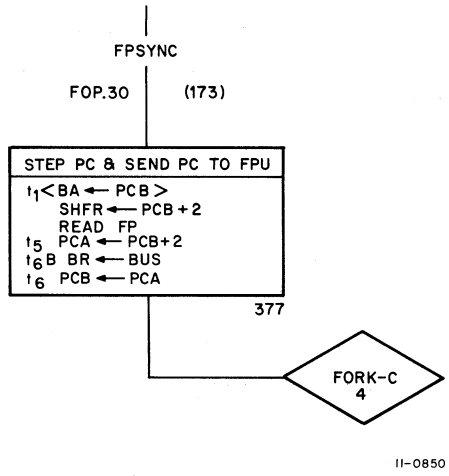


Figure 6-3 An Example of a Microprogram Fork

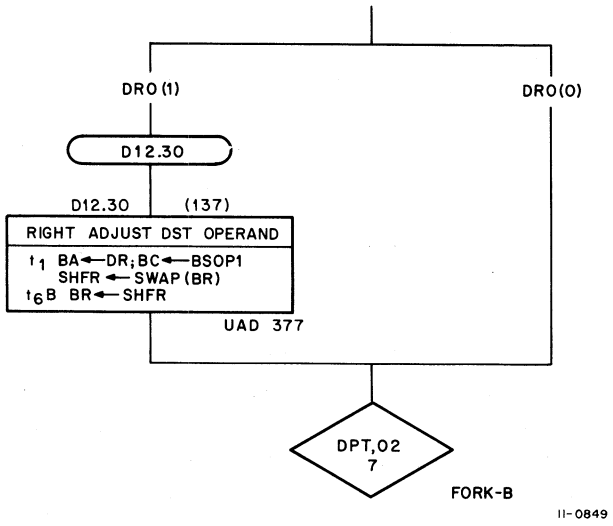


Figure 6-4 Conditional Enabling of Fork Logic

6.1.4 Sequence Symbols in the Flowcharts

The flowcharts illustrate the sequence of machine states with three types of symbols:

a. flow lines
b. connectors, including entry points and off-page connectors
c. branch conditions

Each of these symbol types is described in one of the following paragraphs.

6.1.4.1 Flow Lines – Figure 6-5 illustrates the types of flow lines used in the KB11-A processor flowcharts. The normal sequence of flow is vertically, from top to bottom, on a flowchart. The flow lines connect the boxes that represent the machine states; a vertical line descends from each box to the successor box, as shown in Figure 6-5(a).

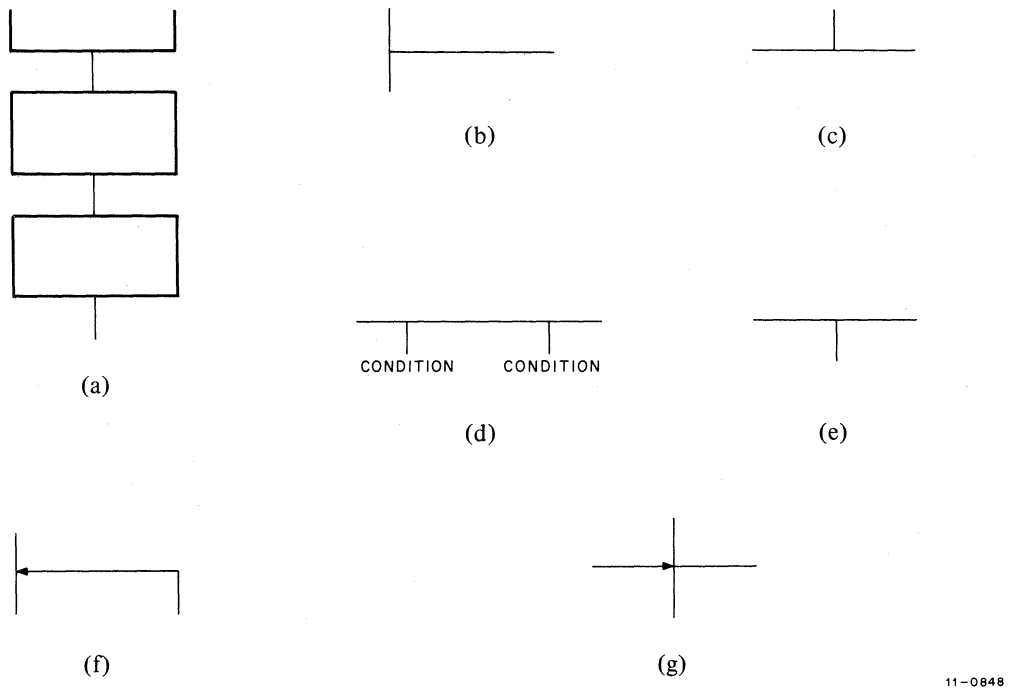


Figure 6-5 Types of Flow Lines Used in Flowcharts

When the sequence is variable, the flow must show branches or forks. Figures 6-5(b) and 6-5(c) illustrate branches. In Figure 6-5(b), the flow follows either the vertical or the horizontal path, depending on the matching of conditions specified as shown in Figure 6-6. In Figure 6-5(c), the flow follows one of the two horizontal paths, depending on conditions. Figure 6-5(d) illustrates the symbol for a fork; one of many paths is selected by uniquely matching the conditions specified on that path. The horizontal line connecting all the possible paths may continue over several pages of the flowcharts, through the use of connectors as illustrated in Figures 6-7(c) and 6-7(g).

In some cases, several sequences combine to use the same machine states for common operations. When the sequences that combine are close together, on the same page, the combination is shown as flow lines like those in Figure 6-5(e). The differing sequences provide descending flow lines that join in a horizontal line from which the common flow line descends. Another method of illustrating combining flows is shown in Figure 6-5(f).

When several flows all branch to a common set of variable sequences, a flow symbol like that shown in Figure 6-5(g) can be used to illustrate the combination and re-division of the sequences.

6.1.4.2 Connector Symbols – Two types of connector symbols are used in the KB11-A processor flowcharts. Entry symbols are used to indicate that a particular sequence of machine states may be executed following other machine states which are not connected to the sequence by flow lines. Figure 6-7(a) through 6-7(c) illustrate entry symbols for entry points at the beginning of an illustrated sequence to a specific machine state in a

sequence, and to the horizontal line for a decision point or fork flow, respectively. The entry point symbol in Figure 6-7(a) or 6-7(b) contains the name of the first succeeding machine state, or in Figure 6-7(c), the name of the fork or decision point.

The second type of connector symbol is the off-page connector. This symbol is used to terminate a connected sequence on a flowchart and to tell what machine state is entered next. When a unique state follows the connector, the symbol is used in the manner illustrated by Figure 6-7(d). When a branch condition determines the next state, the connector symbol can be used in this manner or as shown in Figure 6-7(e). When the next state is determined by a fork, the symbol is used in the manner shown in Figure 6-7(f). If a fork extends over several pages, the connection from each page to the next page on which the fork appears is shown by a symbol like that in Figure 6-7(g).

An off-page connector contains two pieces of information: first, the connector lists the machine state or fork to which the sequence goes; second, the connector indicates the page of the flowcharts on which the succeeding state or sequence appears. In some cases, the off-page connector is accompanied by a microprogram address in parentheses. This is the address of the microprogram word for the next machine state.

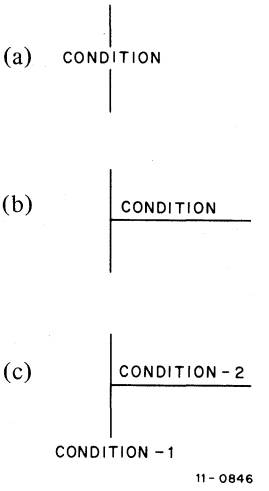


Figure 6-6
Branch Condition Symbols

6.1.4.3 Branch Condition Symbols – The sequence symbols discussed in the preceding paragraphs illustrate the variations in flow, but do not indicate what selects a specific sequence. This is done by listing the conditions for a specific sequence in the flow line that leads to that sequence.

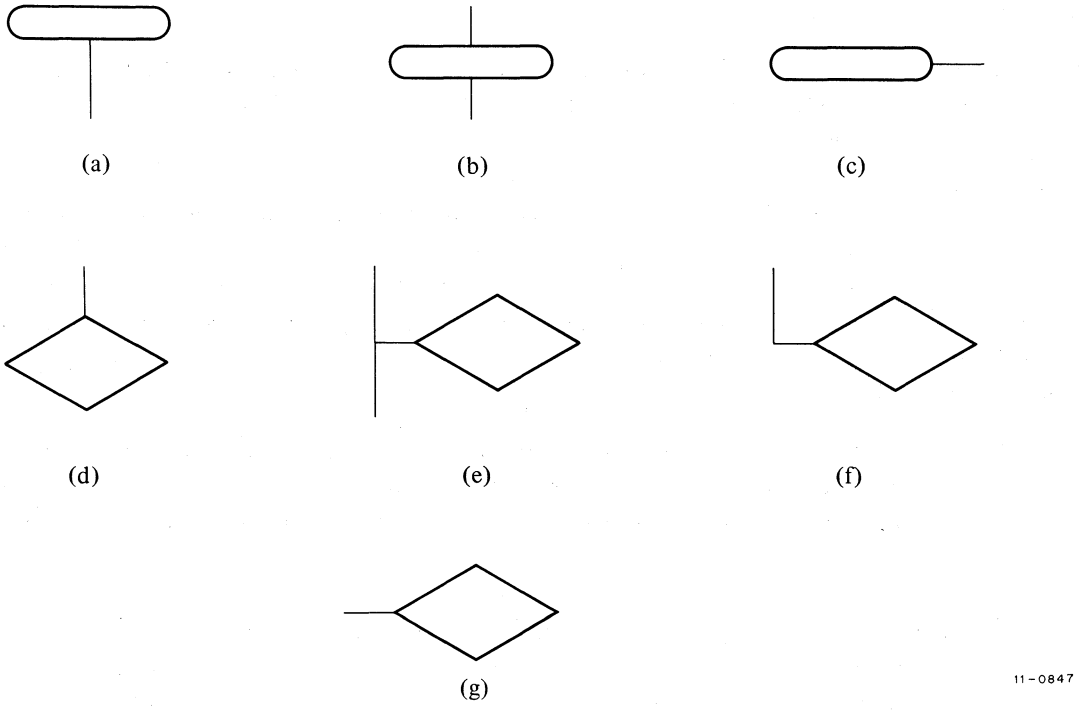


Figure 6-7 Use of Connector Symbols

Whenever possible, the condition that selects a sequence is shown by superimposing the condition on a vertical flow line, as shown in Figure 6-6(a). This indicates that the condition must be met in order for the sequence to follow that path. When a branch is shown by a horizontal line, the condition is listed directly above the horizontal line, close to the branch point, as shown in Figure 6-6(b). In some cases, the branch-enable value corresponding to the conditions at a branch point is listed to the left of the flow line, as shown in Figure 6-6(c); the number in the branch-enable value is the contents of the branch-enable value of the microprogram word corresponding to the machine state preceding the branch.

6.1.5 Locating a Machine State in the Flowcharts

Several tables are provided in this chapter to assist the reader in finding the paragraphs in the chapter that discuss any machine state. Table 6-1 lists the machine states in the numerical order of the ROM word addresses; for each machine state the table lists the mnemonic name of the state, the page of the flowcharts on which it is illustrated, and the number of the paragraph that describes the state. Table 6-2 contains the same information, but it is organized differently; the machine states are listed in alphabetical order according to the mnemonic names.

Tables 6-4 through 6-7 provide information on the machine states entered when the fork logic is enabled. See Paragraph 6.3 for a description of these tables and an explanation of their use.

6.2 FLOWCHART ORGANIZATION

The KB11-A processor flowcharts are divided into 14 drawings that illustrate portions of the flow. Where possible, a continuous sequence of machine states is shown on a single drawing. The succeeding paragraphs describe the machine operations illustrated on each drawing. The description does not attempt to give detailed information about each machine state shown on the drawing; this information can be derived directly from the flowcharts and the ROM map (Paragraph 6.1).

6.2.1 Instruction Fetch

Drawing D-FD-KB11-A-03 (Flows 1) illustrates the instruction fetch sequence, the address calculation sequence for five of the source modes, a special sequence for the MTPI and MTPD instructions, and the execution of the branch type instructions.

6.2.1.1 The Fetch States – The basic instruction fetch sequence requires two machine states: FET.1 (fetch) and IRD.0 (IR decode). The FET.1 state completes a data transfer operation, begun during the last cycle of the previous instruction, which moves the instruction word from an external storage location to the instruction register (IR) and bus register (BR), and increments the program counter by 2. If the data transfer is not overlapped (i.e., if the transfer was not begun before the end of the previous instruction), an additional state is required to begin the data transfer.

The additional state, FET.0, also checks for asynchronous operations (such as bus requests) that must be performed before beginning a new instruction, and branches to the BRK.0 (break) machine state if necessary. When the instruction fetch is overlapped, the machine state that begins the data transfer must also perform the same check.

6.2.1.2 Instruction Decoding – The second state in the basic instruction fetch sequence begins a new data transfer that fetches the word following the instruction word. This data transfer is used for address modes 6 or 7, and for instructions that do not require other data transfers. In all other cases, this data transfer operation is aborted by a bus end (BEND) operation in the machine state following the IRD.0 state. During this machine state, the processor also loads the source and destination registers (SR and DR) with the contents of the general registers

(continued on page 6-15)

Table 6-1

Machine States According to ROM Addresses

Mnemonic	Page	ROM Address	Paragraph
RSD,00	12	000	6,2,11,9
D12,00	5	001	6,2,5,1
D12,01	5	002	6,2,5,1
D30,00	5	003	6,2,5,1
D45,00	6	004	6,2,6,2
D45,01	6	005	6,2,6,2
D67,00	6	006	6,2,6,3
D67,01	6	007	6,2,6,3
HLT,00	3	010	6,2,3,4
WAT,00	3	011	6,2,3,5
RTI,00	2	012	6,2,2,3
TRP,01	12	013	6,2,11,9
TRP,02	12	014	6,2,11,9
RES,00	3	015	6,2,3,3
RTI,01	2	016	6,2,2,3
RSD,01	12	017	6,2,11,9
EXC,80	3	020	6,2,3,2
S13,00	1	021	6,2,1,3
S13,01	1	022	6,2,1,3
S45,10	1	023	6,2,1,3
S45,00	1	024	6,2,1,3
SVC,60	13	025	6,2,12,4
S67,00	2	026	6,2,2,1
S13,10	1	027	6,2,1,3
EXC,90	3	030	6,2,3,2
EXC,00	11	031	6,2,10,1
TST,00	11	032	6,2,10,4
TST-10	11	033	6,2,10,4
JSR,00	11	034	6,2,10,6
JMP,00	11	035	6,2,10,5
FOP,40	7	036	6,2,7,5
SVC,50	13	037	6,2,12,3
RTS,00	2	040	6,2,2,4
SVC,70	13	041	6,2,12,4
RSD,02	12	042	6,2,11,9
SPL,00	3	043	6,2,3,6
CCP,00	3	044	6,2,3,6
MTP,00	1	045	6,2,1,4
MFP,80	11	046	6,2,10,7
MRK,00	2	047	6,2,2,6
MUL,80	3	050	6,2,3,1
DVS,00	3	051	6,2,3,1
ASH,10	7	052	6,2,7,1
ASC,10	7	053	6,2,7,2
S67,10	2	054	6,2,2,1
DVN,40	9	055	6,2,9,2
DVN,50	9	056	6,2,9,2
SOB,00	2	057	6,2,2,5
MUL,00	8	060	6,2,8,1
DIV,00	9	061	6,2,9,1
ASH,00	7	062	6,2,7,1
ASC,00	7	063	6,2,7,2
SHR,00	11	064	6,2,10,3
SVC,10	13	065	6,2,12,2
MFP,00	11	066	6,2,10,7
NEG,00	11	067	6,2,10,2
EXM,10	14	070	6,2,13,4
EXM,00	14	071	6,2,13,4
RAD,00	14	072	6,2,13,3
DEP,10	14	073	6,2,13,4
DEP,00	14	074	6,2,13,4
RDP,00	14	075	6,2,13,3
KST,00	14	076	6,2,13,5
CON,10	14	077	6,2,13,6
PUP,00	12	100	6,2,11,3
FOP,00	2	101	6,2,2,2
MUL,10	8	102	6,2,8,1
DVP,10	9	103	6,2,9,1
DVN,10	9	104	6,2,9,2
DVN,20	9	105	6,2,9,2
DVN,30	9	106	6,2,9,2
DVN,70	9	107	6,2,9,2
D12,90	5	110	6,2,5,1
D12,80	5	111	6,2,5,1
D30,90	5	112	6,2,5,1
D30,80	5	113	6,2,5,1
D45,90	6	114	6,2,6,1
D45,80	6	115	6,2,6,1
D67,90	6	116	6,2,6,3
D67,80	6	117	6,2,6,3
BRK,20	12	120	6,2,11,7
D40,20	6	121	6,2,6,1
D10,30	6	122	6,2,6,4
SHR,10	11	123	6,2,10,2
TRP-02	12	124	6,2,11,9
DVC,00	10	125	6,2,9,5
FET,05	1	126	6,2,1,1
WAT,30	3	127	6,2,3,5
BRK,10	12	130	6,2,11,2
D40,30	6	131	6,2,6,1
EXC,10	11	132	6,2,10,1
FOP,10	2	133	6,2,2,2
EXM,30	14	134	6,2,13,4
D12,70	5	135	6,2,5,2
ASC,61	7	136	6,2,7,4
D12,30	5	137	6,2,5,2
BRK,80	12	140	6,2,11,6
S67,20	2	141	6,2,2,1
S67,30	2	142	6,2,2,1
S13,30	2	143	6,2,2,1
DVD-00	10	144	6,2,9,7
DVE,20	9	145	6,2,9,3
S13,40	2	146	6,2,2,1
DIV,30	9	147	6,2,9,3
FSV,10	12	150	6,2,11,8
MTP,10	1	151	6,2,1,4
BRK,30	12	152	6,2,11,6
EXM,20	14	153	6,2,13,4
BRK,00	12	154	6,2,11,2
D12,60	5	155	6,2,5,2
RTI,10	2	156	6,2,2,3
D10,40	6	157	6,2,6,4
SER,00	12	160	6,2,11,5
D50,20	6	161	6,2,6,1

(continued on next page)

Table 6-1 (Cont)

Machine States According to ROM Addresses

Mnemonic	Page	ROM Address	Paragraph
D10,00	6	162	6,2,6,4
NEG,20	11	163	6,2,10,2
FET,04	1	164	6,2,1,1
DIV,70	9	165	6,2,9,4
ASH,41	7	166	6,2,7,1
CON,01	14	167	6,2,13,1
CON,00	14	170	6,2,13,1
D50,30	6	171	6,2,6,1
RTI,60	2	172	6,2,2,3
FOP,30	2	173	6,2,2,2
FOP,20	2	174	6,2,2,2
D12,10	5	175	6,2,5,2
ASC,31	7	176	6,2,7,4
D10,60	6	177	6,2,6,4
ZAP,00	12	200	6,2,11,1
JSR,10	11	201	6,2,10,6
D07,00	4	202	6,2,4,2
D07,10	4	203	6,2,4,2
D00,80	4	204	6,2,4,1
D00,90	4	205	6,2,4,1
MUL,40	8	206	6,2,8,3
ASC,80	7	207	6,2,7,3
NEG,90	3	210	6,2,3,2
FOP,50	4	211	6,2,4,3
RTI,20	2	212	6,2,2,3
RTI,30	2	213	6,2,2,3
RTI,40	2	214	6,2,2,3
RTI,50	2	215	6,2,2,3
DVC,10	10	216	6,2,9,6
FET,00	1	217	6,2,1,1
DVC,90	10	220	6,2,9,7
D30,10	5	221	6,2,5,3
SVC,80	13	222	6,2,12,4
RTS,10	2	223	6,2,2,4
RTS,20	2	224	6,2,2,4
FSV,20	12	225	6,2,11,8
MUL,50	8	226	6,2,8,3
ASC,60	7	227	6,2,7,4
CON,20	14	230	6,2,13,1
D10,10	6	231	6,2,6,4
DIV,20	9	232	6,2,9,1
D10,20	6	233	6,2,6,4
MRK,30	2	234	6,2,2,6
MRK,20	2	235	6,2,2,6
DVC,20	10	236	6,2,9,6
FET,07	1	237	6,2,1,1
BRK,90	12	240	6,2,11,2
DVP,00	9	241	6,2,9,1
SOB,10	2	242	6,2,2,5
DVN,60	9	243	6,2,9,2
WAT,20	3	244	6,2,3,5
FSV,00	12	245	6,2,11,8
MUL,30	8	246	6,2,8,2
ASC,30	7	247	6,2,7,4
MFP,10	11	250	6,2,10,7
D67,10	6	251	6,2,6,3
MRK,10	2	252	6,2,2,6
BRK,11	12	253	6,2,11,2
DIV,10	9	254	6,2,9,1
RES,10	3	255	6,2,3,3
DVC,30	10	256	6,2,9,6
ASH,40	7	257	6,2,7,1
FET,10	1	260	6,2,1,1
DVN,00	9	261	6,2,9,2
SOB,20	2	262	6,2,2,5
DVE,10	9	263	6,2,9,2
WAT,10	3	264	6,2,3,5
FET,09	1	265	6,2,1,1
MUL,20	8	266	6,2,8,2
ASC,40	7	267	6,2,7,4
ADR,00	14	270	6,2,13,2
NEG,10	11	271	6,2,10,2
DVE,00	9	272	6,2,9,1
WAT,11	3	273	6,2,3,5
JSR,20	11	274	6,2,10,6
JSR,30	11	275	6,2,10,6
DVC,40	10	276	6,2,9,6
ASH,30	7	277	6,2,7,1
SVC,90	13	300	6,2,12,4
NEG,70	3	301	6,2,3,2
ZAP,30	12	302	6,2,11,1
DEP,20	14	303	6,2,13,4
MFP,90	11	304	6,2,10,7
ASH,20	7	305	6,2,7,1
ASC,20	7	306	6,2,7,3
RDP,10	14	307	6,2,13,3
MUL,60	8	310	6,2,8,3
D10,50	6	311	6,2,6,4
D12,20	5	312	6,2,5,2
DIV,60	9	313	6,2,9,3
RAD,10	14	314	6,2,13,3
ZAP,20	12	315	6,2,11,1
FOP,70	4	316	6,2,4,3
S13,20	2	317	6,2,2,1
BXX,00	1	320	6,2,1,5
FET,11	1	321	6,2,1,1
FET,12	1	322	6,2,1,1
DIV,80	9	323	6,2,9,4
FET,13	1	324	6,2,1,1
BXX,01	1	325	6,2,1,5
BXX,02	1	326	6,2,1,5
HLT,10	3	327	6,2,3,4
BXX,03	1	330	6,2,1,5
FET,01	1	331	6,2,1,1
FET,02	1	332	6,2,1,1
PUP,20	12	333	6,2,11,3
FET,03	1	334	6,2,1,1
BXX,04	1	335	6,2,1,5
BXX,05	1	336	6,2,1,5
XXX,XX	13	337	0
PUP,30	12	340	6,2,11,3
PUP,40	12	341	6,2,11,3
RTS,30	2	342	6,2,2,4
IRD,00	1	343	6,2,1,2

(continued on next page)

Table 6-1 (Cont)
Machine States According to ROM Addresses

Mnemonic	Page	ROM Address	Paragraph	Mnemonic	Page	ROM Address	Paragraph
SER,10	12	344	6,2,11,5	FOP,60	4	362	6,2,4,3
RSD,10	12	345	6,2.11,9	DIV,90	9	363	6,2,9,4
DIV,40	9	346	6,2,9,3	XXX,XX	13	364	Ø
PUP,10	12	347	6,2,11,3	XXX,XX	13	365	Ø
DVC,70	10	350	6,2,9,7	DIV,50	9	366	6,2,9,3
DVC,80	10	351	6,2,9,7	SVC,40	13	367	6,2,12,3
BRK.01	12	352	6,2,11,2	DVC,50	10	370	6,2,9,7
DVD.10	10	353	6,2,9,7	DVC,60	10	371	6,2,9,7
TRP,10	12	354	6,2,11,9	ZAP,10	12	372	6,2,11,1
SVC,00	13	355	6,2,12,1	FET,06	1	373	6,2,1,1
FET,08	1	356	6.2.1,1	RES,20	3	374	6,2,3,3
SVC,20	13	357	6,2,12,2	FOP,90	4	375	6,2,4,3
SVC,30	13	360	6,2,12,2	FOP,80	4	376	6,2,4,3
SPL,10	3	361	6,2,3,6	XXX,XX	13	377	Ø

Table 6-2
Machine States According to Mnemonic Names

Mnemonic	Page	ROM Address	Paragraph
ADR,00	14	270	6,2,13,2
ASC,00	7	063	6,2,7,2
ASC,10	7	053	6,2,7,2
ASC,20	7	306	6,2,7,3
ASC,30	7	247	6,2,7,4
ASC,31	7	176	6,2,7,4
ASC,40	7	267	6,2,7,4
ASC,60	7	227	6,2,7,4
ASC,61	7	136	6,2,7,4
ASC,80	7	207	6,2,7,3
ASH,00	7	062	6,2,7,1
ASH,10	7	052	6,2,7,1
ASH,20	7	305	6,2,7,1
ASH,30	7	277	6,2,7,1
ASH,40	7	257	6,2,7,1
ASH,41	7	166	6,2,7,1
BRK,00	12	154	6,2,11,2
BRK.90	12	240	6,2,11,2
BRK,02	12	352	6,2,11,2
BRK,10	12	130	6,2,11,2
BRK,11	12	253	6,2,11,2
BRK,20	12	120	6,2,11,7
BRK,30	12	152	6,2,11,6
BRK,80	12	140	6,2,11,6
BXX,00	1	320	6,2,1,5
BXX,01	1	325	6,2,1,5
BXX,02	1	326	6,2,1,5
BXX,03	1	330	6,2,1,5
BXX,04	1	335	6,2,1,5
BXX,05	1	336	6,2,1,5
CCP,00	3	044	6,2,3,6
CON,00	14	170	6,2,13,1
CON,01	14	167	6,2,13,1
CON,10	14	077	6,2,13,6
CON,20	14	230	6,2,13,1
D00,80	4	204	6,2,4,1
D00,90	4	205	6,2,4,1
D07,00	4	202	6,2,4,2
D07,10	4	203	6,2,4,2
D10,00	6	162	6,2,6,4
D10,10	6	231	6,2,6,4
D10,20	6	233	6,2,6,4
D10,30	6	122	6,2,6,4
D10,40	6	157	6,2,6,4
D10,50	6	311	6,2,6,4
D10,60	6	177	6,2,6,4
D12,00	5	001	6,2,5,1
D12,01	5	002	6,2,5,1
D12,10	5	175	6,2,5,2
D12,20	5	312	6,2,5,2
D12,30	5	137	6,2,5,2
D12,60	5	155	6,2,5,2
D12,70	5	135	6,2,5,2
D12,80	5	111	6,2,5,1
D12,90	5	110	6,2,5,1
D30,00	5	003	6,2,5,1
D30,10	5	221	6,2,5,3
D30,80	5	113	6,2,5,1
D30,90	5	112	6,2,5,1
D40,20	6	121	6,2,6,1
D40,30	6	131	6,2,6,1
D45,00	6	004	6,2,6,2
D45,01	6	005	6,2,6,2
D45,80	6	115	6,2,6,1
D45,90	6	114	6,2,6,1
D50,20	6	161	6,2,6,1
D50,30	6	171	6,2,6,1
D67,00	6	006	6,2,6,3
D67,01	6	007	6,2,6,3
D67,10	6	251	6,2,6,3
D67,80	6	117	6,2,6,3
D67,90	6	116	6,2,6,3
DEP,00	14	074	6,2,13,4
DEP,10	14	073	6,2,13,4
DEP,20	14	303	6,2,13,4
DIV,00	9	061	6,2,9,1
DIV,10	9	254	6,2,9,1
DIV,20	9	232	6,2,9,1
DIV,30	9	147	6,2,9,3
DIV,40	9	346	6,2,9,3
DIV,50	9	366	6,2,9,3
DIV,60	9	313	6,2,9,3
DIV,70	9	165	6,2,9,4
DIV,80	9	323	6,2,9,4
DIV,90	9	363	6,2,9,4
DVC,00	10	125	6,2,9,5
DVC,10	10	216	6,2,9,6
DVC,20	10	236	6,2,9,6
DVC,30	10	256	6,2,9,6
DVD.10	12	144	6,2,9,7
DVC.40	10	276	6,2,9,6
DVD.10	10	353	6,2,9,7
DVC,50	10	370	6,2,9,7
DVC,60	10	371	6,2,9,7
DVC,70	10	350	6,2,9,7
DVC,80	10	351	6,2,9,7
DVC,90	10	220	6,2,9,7
DVE,00	9	272	6,2,9,1
DVE,10	9	263	6,2,9,2
DVE,20	9	145	6,2,9,3
DVN,00	9	261	6,2,9,2
DVN,10	9	104	6,2,9,2
DVN,20	9	105	6,2,9,2
DVN,30	9	106	6,2,9,2
DVN,40	9	055	6,2,9,2
DVN,50	9	056	6,2,9,2
DVN,60	9	243	6,2,9,2
DVN,70	9	107	6,2,9,2
DVP,00	9	241	6,2,9,1
DVP,10	9	103	6,2,9,1
DVS,00	3	051	6,2,3,1
EXC,00	11	031	6,2,10,1
EXC,10	11	132	6,2,10,1
EXC,80	3	020	6,2,3,2

(continued on next page)

Table 6-2 (Cont)

Machine States According to Mnemonic Names

Mnemonic	Page	ROM Address	Paragraph
EXC,90	3	030	6,2,3,2
EXM,00	14	071	6,2,13,4
EXM,10	14	073	6,2,13,4
EXM,20	14	153	6,2,13,4
EXM,30	14	134	6,2,13,4
FET,00	1	217	6,2,1,1
FET,01	1	331	6,2,1,1
FET,02	1	332	6,2,1,1
FET,03	1	334	6,2,1,1
FET,04	1	164	6,2,1,1
FET,05	1	126	6,2,1,1
FET,06	1	373	6,2,1,1
FET,07	1	237	6,2,1,1
FET,08	1	356	6,2,1,1
FET,09	1	265	6,2,1,1
FET,10	1	260	6,2,1,1
FET,11	1	321	6,2,1,1
FET,12	1	322	6,2,1,1
FET,13	1	324	6,2,1,1
FOP,00	2	101	6,2,2,2
FOP,10	2	133	6,2,2,2
FOP,20	2	174	6,2,2,2
FOP,30	2	173	6,2,2,2
FOP,40	7	036	6,2,7,5
FOP,50	4	211	6,2,4,3
FOP,60	4	362	6,2,4,3
FOP,70	4	316	6,2,4,3
FOP,80	4	376	6,2,4,3
FOP,90	4	375	6,2,4,3
FSV,00	12	245	6,2,11,8
FSV,10	12	150	6,2,11,8
FSV,20	12	225	6,2,11,8
HLT,00	3	010	6,2,3,4
HLT,10	3	327	6,2,3,4
IRD,00	1	343	6,2,1,2
JMP,00	11	035	6,2,10,5
JSR,00	11	034	6,2,10,6
JSR,10	11	201	6,2,10,6
JSR,20	11	274	6,2,10,6
JSR,30	11	275	6,2,10,6
KST,00	14	076	6,2,13,5
MFP,00	11	366	6,2,10,7
MFP,10	11	250	6,2,10,7
MFP,80	11	046	6,2,10,7
MFP,90	11	304	6,2,10,7
MRK,00	2	047	6,2,2,6
MRK,10	2	252	6,2,2,6
MRK,20	2	235	6,2,2,6
MRK,30	2	234	6,2,2,6
MTP,00	1	245	6,2,1,4
MTP,10	1	151	6,2,1,4
MUL,00	8	060	6,2,8,1
MUL,10	8	102	6,2,8,1
MUL,20	8	266	6,2,8,2
MUL,30	8	246	6,2,8,2
MUL,40	8	206	6,2,8,3
MUL,50	8	226	6,2,8,3
MUL,60	8	310	6,2,8,3
MUL,80	3	057	6,2,3,1
NEG,00	11	267	6,2,10,2
NEG,10	11	271	6,2,10,2
NEG,20	11	163	6,2,10,2
NEG,70	3	301	6,2,3,2
NEG,90	3	217	6,2,3,2
PUP,00	12	100	6,2,11,3
PUP,10	12	347	6,2,11,3
PUP,20	12	333	6,2,11,3
PUP,30	12	340	6,2,11,3
PUP,40	12	341	6,2,11,3
RAD,00	14	072	6,2,13,3
RAD,10	14	314	6,2,13,3
RDP,00	14	075	6,2,13,3
RDP,10	14	307	6,2,13,3
RES,00	3	015	6,2,3,3
RES,10	3	255	6,2,3,3
RES,20	3	374	6,2,3,3
RSD,00	12	000	6,2,11,9
RSD,01	12	017	6,2,11,9
RSD,02	12	042	6,2,11,9
RSD,10	12	345	6,2,11,9
RTI,00	2	012	6,2,2,3
RTI,01	2	016	6,2,2,3
RTI,10	2	156	6,2,2,3
RTI,20	2	212	6,2,2,3
RTI,30	2	213	6,2,2,3
RTI,40	2	214	6,2,2,3
RTI,50	2	215	6,2,2,3
RTI,60	2	172	6,2,2,3
RTS,00	2	040	6,2,2,4
RTS,10	2	223	6,2,2,4
RTS,20	2	224	6,2,2,4
RTS,30	2	342	6,2,2,4
S13,00	1	021	6,2,1,3
S13,01	1	022	6,2,1,3
S13,10	1	027	6,2,1,3
S13,20	2	317	6,2,2,1
S13,30	2	143	6,2,2,1
S13,40	2	146	6,2,2,1
S45,00	1	024	6,2,1,3
S45,10	1	023	6,2,1,3
S67,00	2	026	6,2,2,1
S67,10	2	054	6,2,2,1
S67,20	2	141	6,2,2,1
S67,30	2	142	6,2,2,1
SER,00	12	160	6,2,11,5
SER,10	12	344	6,2,11,5
SHR,00	11	064	6,2,10,3
SHR,10	11	123	6,2,10,2
SOB,00	2	057	6,2,2,5
SOB,10	2	242	6,2,2,5
SOB,20	2	262	6,2,2,5
SPL,00	3	043	6,2,3,6
SPL,10	3	361	6,2,3,6
SVC,00	13	355	6,2,12,1

(continued on next page)

Table 6-2 (Cont)

Machine States According to Mnemonic Names

Mnemonic	Page	ROM Address	Paragraph	Mnemonic	Page	ROM Address	Paragraph
SVC,1Ø	13	365	6,2,12,2	TST.1Ø	11	333	6,2,1Ø,4
SVC,2Ø	13	357	6.2.12,2	WAT,ØØ	3	011	6,2,3,5
SVC,3Ø	13	360	6,2,12,2	WAT,1Ø	3	264	6,2,3,5
SVC,4Ø	13	367	6,2.12,3	WAT,11	3	273	6,2,3,5
SVC,5Ø	13	037	6,2,12,3	WAT,2Ø	3	244	6,2,3,5
SVC,6Ø	13	325	6,2,12,4	WAT,3Ø	3	127	6,2,3,5
SVC,7Ø	13	341	6,2,12,4	XXX,XX	13	337	Ø
SVC,8Ø	13	222	6,2,12,4	XXX,XX	13	364	Ø
SVC,9Ø	13	300	6,2,12,4	XXX,XX	13	365	Ø
TRP,Ø1	12	013	6,2,11,9	XXX,XX	13	377	Ø
TRP,Ø2	12	Ø14	6,2,11,9	ZAP,ØØ	12	2ØØ	6,2,11,1
TRP.Ø2	12	124	6,2,11,9	ZAP,1Ø	12	372	6,2,11,1
TRP,1Ø	12	354	6,2,11,9	ZAP,2Ø	12	315	6,2,11,1
TST,ØØ	11	Ø32	6,2,1Ø,4	ZAP,3Ø	12	3Ø2	6,2,11,1

specified in the source and destination fields of the instruction; this operation is also done in anticipation of the use of this data, and in many cases the data loaded into the SR and DR is ignored. However, when the data is needed, the anticipatory transfers allow the processor to operate at maximum speed.

6.2.1.3 Source Modes 1 Through 5 – The fork A logic is enabled during IRD.0, so the machine state that follows IRD.0 is determined by decoding the instruction and certain other conditions. Six of the possible sequences that follow IRD.0 are shown on Flows 1. These include the beginning of the data fetch sequence for all binary instructions that have a source mode of 1 through 5. If the source mode is 1, 2, or 3, the external data transfer is restarted with a new address and the incrementation of the source register is started for modes 2 or 3. If the source mode is 4 or 5, the external data transfer can not be continued until the address has been decremented, so the S45.0 state performs a BEND. After performing the data transfer to fetch the word addressed by the source register, the sequence conditionally enables the fork C logic. If the source mode is odd, another data transfer is required to fetch the data addressed by the word just fetched; otherwise the fork determines the next state.

6.2.1.4 Move to Previous Space Instructions – For an MTPI or MTPD instruction, the MTP.0 (move to previous) and MTP.1 states read an address from the stack pointer and begin a data transfer operation to fetch a data word that will be transferred to the destination address. The flow then transfers to the last state of the source-data-fetch sequence, because this state is common to both the MTP sequence and the normal source data sequence.

6.2.1.5 Branch Instructions – For branch instructions, the fork A logic determines whether the branch is successful, and if not, whether a bus request has been sensed. If the branch is successful, the PC must be changed before the next instruction is fetched; this is performed by the BXX.0 (branch) machine state which aborts the previous data transfer. This state also strobes any new bus requests. The BRQ STROBE must be performed in the state preceding the state that starts the instruction fetch; this includes FET.1 (in case the fork A logic returns control directly to FET.0), the next-to-last state of instructions that overlap the instruction fetch, and the last state of instructions that do not provide overlap. The machine state following BXX.0 is FET.0.

If the branch is not successful and no bus requests are sensed, the instruction fetch continues the data transfer begun in IRD.0; if a bus request is sensed, the sequence returns to FET.0 which, in turn, transfers the sequence to BRK.0. Table 6-3 illustrates the exact ROM words used by each branch instruction for the four possible sequences.

6.2.2 Indexed Source Modes and Operate Instructions

Drawing D-FD-KB11-A-03 (Flows 2) illustrates the sequence of machine states for the data fetch for source modes 6 or 7, for the transfer of floating-point instructions to the FPP, and for the execution of five operate instructions.

6.2.2.1 Indexed Source Modes – For the indexed source modes, the transfer begun in machine state IRD.0 is completed and an increment from the source register is added to the data word; the resulting data word is used for a second data transfer. When this transfer is complete, a conditional fork is used to transfer to the sequence required for the current instruction, unless an indirect indexed address requires a third data transfer. In the latter case, the sequence continues through three machine states that are common to the sequences of all indirect source modes (i.e., modes 3, 5, and 7), and in part to the MTPI or MTPD instruction.

6.2.2.2 Floating-Point Instructions – When a floating-point instruction is recognized by the fork A logic, the sequence is transferred to the FOP.0 (floating-point operation) state. In this state, the processor restores the PC

to the value used to fetch the instruction, so that this value can be transmitted to the FPP (which stores the value for use in reporting abnormal conditions during the execution of that instruction, and for restarting the instruction if interrupted), and notifies the FPP that a floating-point instruction is ready to be processed. The processor then enters a wait loop, consisting of two machine states, until the FPP acknowledges the FPATTN (FPP attention) signal and reads the contents of the IR. (The data is actually read from the BR, which at this time contains the same information.) If the FPP is busy with a previous floating-point instruction, the processor may have to wait for several microseconds; during the wait period, the processor looks for other external requests and releases control if any occur. If an interrupt must be processed, the stored PC value allows the floating-point instruction to be re-fetched after the interrupt service is completed. After the IR and PC have been transferred to the FPP, the sequence is determined by the fork C logic to perform the address calculation for the floating-point data.

Table 6-3
Branch Sequences

Conditions:	Successful		Unsuccessful	
Bus Request:	**Present**	**Not Present**	**Present**	**Not Present**
Instruction				
BCC	BXX.03	BXX.00	FET.01	FET.11
BCS	BXX.04	BXX.01	FET.03	FET.13
BEQ	BXX.05	BXX.02	FET.03	FET.13
BGE	BXX.03	BXX.00	FET.01	FET.11
BGT	BXX.03	BXX.00	FET.01	FET.11
BHI	BXX.03	BXX.00	FET.02	FET.12
BHIS	BXX.03	BXX.00	FET.02	FET.12
BLE	BXX.05	BXX.02	FET.03	FET.13
BLO	BXX.04	BXX.01	FET.03	FET.13
BLOS	BXX.04	BXX.01	FET.03	FET.13
BLT	BXX.05	BXX.02	FET.03	FET.13
BMI	BXX.04	BXX.01	FET.03	FET.13
BNE	BXX.03	BXX.00	FET.02	FET.12
BPL	BXX.03	BXX.00	FET.01	FET.11
BR	BXX.05	BXX.02	(always successful)	
BVC	BXX.03	BXX.00	FET.01	FET.11
BVS	BXX.04	BXX.01	FET.03	FET.13

6.2.2.3 RTI and RTT Instructions – The RTI and RTT instructions differ only in the clocking of T bit traps after the data transfers, so the sequence of machine states is identical. This sequence performs two data transfers to restore the previous PC and PS words from the hardware stack, and performs two increment operations on the stack pointer. The sequence then continues with an instruction fetch.

6.2.2.4 RTS Instruction – The RTS sequence performs one register-to-register transfer and one external data transfer to restore the PC and the specified register, and updates the stack pointer (SP) after the transfer. The sequence then returns to the instruction fetch machine states.

6.2.2.5 SOB Instruction – The sequence of machine states for the SOB instruction first generates a new PC value based on the offset in the instruction, and then restores the old PC value if the value in the specified register will be 0 after decrementing. This is done because the test on the value of the register requires one machine state in every case, which can be combined with the calculation of the new PC value, and because the

branch is successful most of the time; thus, the extra machine state to perform the restoration of the old PC value is executed less often than if an extra state were required when the branch is successful. The SOB sequence initiates the fetch of the next instruction during the last machine state which also performs the decrement on the specified register.

6.2.2.6 MARK Instruction – The machine state sequence for the MARK instruction transfers the contents of general register 5 to the PC, transfers the top word on the hardware stack to register 5, then begins fetching the next instruction. The operation of the MARK instruction assumes that the instruction has been fetched from the top of the hardware stack; for a discussion of the purpose and effects of the MARK instruction, see Chapter 4.

6.2.3 No Memory Reference Execution

Drawing D-FD-KB11-A-03 (Flows 3) illustrates the machine state sequences for a variety of instructions that do not require memory references other than the instruction fetch. A number of sequences are shown that transfer immediately to machine states on other pages; they are shown only to illustrate the routing from fork A to these states. These sequences include the breakpoint trap (OP3), IOT trap, the EMT and TRAP traps, and several groups of reserved op codes, including OP7, OP22, and RSVD. The illegal instructions JMP or JSR, with destination mode 0, also transfer directly to a point in the trap machine state sequence. The four instructions ASH, ASHC, MFPI, and MFPD are shown on other pages which do not show the fork A flow line; therefore, off-page connectors are shown on this drawing for these instructions with destination mode 0 (for other destination modes of these instructions, the sequence transfers to the destination address calculation sequences shown on flowcharts 5 and 6).

6.2.3.1 Multiply and Divide with Destination Mode 0 – For the multiply and divide instructions, a special sequence is used when the destination mode is 0. In either case, this sequence precedes the normal sequence for that instruction. The MUL.8 (multiply) machine state sets up the step counter and transfers to the MUL.1 machine state, because the MUL.0 state is used to complete the data transfer begun in the destination data fetch sequence. In the DVS.0 (divide start) state, the contents of the register specified for the destination operand are transferred to the BR, which corresponds to the result of the data fetch sequence for other destination modes.

6.2.3.2 E CLASS and Negate Instructions – For the majority of the instructions that operate on data, one machine state is required to perform the data manipulation. If both the source (if any) and destination modes are 0, the data is already in the SR and DR registers as a result of the IRD.0 state. The data manipulation (selected by the subsidiary ROM for all except the NEG.B instruction) is performed, the data is stored in the general register specified by the destination field, and the sequence returns to the instruction fetch. The NEG and NEG.B instructions require two machine states because the complement and increment operations can not be performed on the data during the same state; therefore the external data transfer operation started in the IRD.0 state is aborted (a bus operation can not be carried across more than two machine states) and the sequence returns to the FET.0 state. The other instructions complete the data operation and return to FET.1 unless a bus request has been sensed; because the transfer to the BRQ service sequence is performed by the FET.0 machine state, the bus operation must be aborted.

6.2.3.3 RESET Instruction – Three processor control instructions, RESET, HALT, and WAIT, are executed by sequences shown on this drawing. The RESET instruction transfers general register 0 to the DR so that the contents of R0 can be displayed in the DATA lights of the console during the reset operation, and then triggers the initialization pulse. The initialization is inhibited if the processor is not operating in the kernel mode; in this case, the instruction is, in effect, a NOP. The machine state that triggers the pulse recycles to itself until the pulse (which lasts for 10 ms) is completed, and then returns the sequence to the instruction fetch sequence.

6.2.3.4 HALT Instruction – The HALT instruction does not actually stop the processor; instead, control is transferred to the console service sequence, which waits for manual intervention to determine further operations. This is performed by setting the console flag and then returning to the instruction fetch sequence where the console flag generates a BRQ, which, in turn, transfers to the break service sequence. The console flag is set only if the processor is in kernel mode; a branch after the HLT.1 (HALT) machine state transfers control to the trap service sequence if the processor is not in kernel mode.

6.2.3.5 WAIT Instruction – The WAIT instruction is used to wait for an asynchronous condition that either initiates the execution of a service program or enters the console service sequence. The basic wait loop consists of two machine states, so that the BRQ STROBE in one state is available for the branch in the other state. When any BRQ is sensed, the sequence goes to the first of two states that test for console requests and then for interrupts or traps (other than T bit traps) that supply vectors. If neither is found, the sequence returns to the wait loop; otherwise, control is transferred to the appropriate sequence.

6.2.3.6 Processor Status Change Instructions – Two instructions that transfer data from the instruction word to the PS word are the CCOP instruction and the SPL instruction. The former affects only the condition code bits (PS⟨03:00⟩), and the latter affects only the priority bits (PS⟨07:05⟩). In the CCOP instruction, the external data transfer begun by the IRD.0 state is aborted because the processor must maintain the data in the BR register until the PS word is reloaded. In the SPL instruction, the first state does the actual transfer to the priority. The second state also begins a new instruction fetch and control transfers to the FET.1 state. SPL is a no-op (no change to the PS) unless the processor is in kernel mode.

6.2.4 Destination Mode 0 Sequences

Drawing D-FD-KB11-A-03 (Flows 4) illustrates the five sequences used when the destination mode is 0. These sequences are entered through the fork C microprogram address calculation; this fork is used to determine the next machine state after a source operand has been fetched. For all instructions except floating-point instructions, these sequences correspond to, or join, the sequences used when both the source and the destination modes are 0.

6.2.4.1 Not Register 7 – When the destination specification in an instruction refers to any general register other than register 7 (the PC), and the other conditions for the sequences shown on this drawing are met, the instruction is executed by the machine state D00.9 (destination mode 0). If the source address is odd, a byte-swap operation must be performed on the contents of the BR before the instruction-dependent data manipulation operation. If the source mode is also 0, no byte swap can be required, and the execution is performed by the EXC.8 (execute) machine state (Paragraph 6.2.3.2).

6.2.4.2 Register 7 – When the destination register is 7, the PC is modified. Because the PC is stored as a separate register (not in the general register set), the execution is accomplished by the EXC.9 machine state, which requires the source data to be in the SR register. A machine state is therefore required to transfer the source data from the BR to the SR. A byte swap can be combined with this transfer if necessary.

6.2.4.3 Floating-Point Instructions – For most floating-point instructions, the destination specification refers to a floating-point accumulator if the destination mode is 0. However, this sequence is also entered for the CFCC instruction and for the load and store status instructions, for which the destination specification refers to the general registers if the destination mode is 0. Therefore, if the instruction is a CFCC instruction, the first machine state transfers the floating-point condition codes from the internal bus to the PS word. The contents of the DR, which contains the data read from the destination register during the IRD.0 machine state, is transferred

to the BR so that the FPP can read the destination if necessary, and an FPATTN signal is sent. The processor then waits in a one-machine-state loop which tests for the FP SYNC signal; if the FPP sends a data word to be stored in the destination register, the FOP.8 (floating-point operation) machine state is entered, otherwise the sequence returns to the instruction fetch sequence. After receiving data from the FPP, the processor again sends the FPATTN signal and enters the wait loop; if the FPP is operating with double precision integers, the data receiving sequence is entered twice and the second word (which is the lower half of the 2-word variable) is stored in the same destination register, overlaying the first word. When the FPP has no more data to send, the processor returns to the instruction fetch sequence.

6.2.5 Destination Modes 1 Through 3

Drawing D-FD-KB11-A-03 (Flows 5) illustrates the machine state sequences used to fetch data specified by destination modes 1, 2, or 3. These sequences are entered from one of the two forks; some are entered from the fork A decision point, for instructions which either do not require a source operand or have a source mode of 0, while others are entered from the fork C decision point after the source operand has been fetched and placed in the SR.

6.2.5.1 Sequence Entry – For all six of the sequences shown on this chart, the external data transfer begun during the IRD.0 machine state is continued, but a BUST is issued to inform the data transfer logic that the address has changed and that all the deskewing delays must be restarted. The four sequences entered from the fork C decision point also start by transferring the contents of the BR to the SR, so that the source data is available in both registers; the opposite transfer is performed for the fork A entry if the destination mode is 1 or 2. If the destination mode is 3, there is no point in loading the BR from the SR because the address fetched by the first external data transfer is stored in the BR for use in the next data transfer.

6.2.5.2 Destination Modes 1 and 2 – There are two entries from the fork C decision point for address modes 1 or 2 because the source data may be an odd byte which must be swapped. This is the only difference between machine states D12.0 (destination modes 1 or 2) and D12.9. After one of these states or D12.0 has been completed, the processor performs a three-way branch, to separate JMP, JSR, and floating-point instructions, and instructions that transfer the source operand to the destination unchanged (specifically, the MOV, MTPI, and MTPD instructions) from all others. For floating-point instructions, the external data transfer is aborted, and the sequence continues through the fork B decision point to the FOP.4 machine state. For JMP instructions, the sequence is directed to the JMP.0 state; for JSR instructions, to the JSR.0 state. For the three direct-transfer (O Class) instructions, the external transfer is forced to be a DATO instead of a DATIP or a DATI, and the transfer is completed before an instruction-dependent condition-code load operation is performed. The last machine state in the sequence for O Class instructions also begins the instruction fetch for the next instruction and checks for asynchronous conditions requiring service.

For all other instructions, the DATI or DATIP transfer is completed, and the fork B logic is conditionally enabled in machine state D12.1. If a byte swap is needed because the destination address is to an odd byte, the extra machine state D12.3 is entered, and then the fork B decision point. Note that in all three of the sequences shown (in machine states D12.6, D12.1, and D12.7) the destination register is incremented by a constant which can be either 0, 1, or 2, depending on the address mode and whether a word or a byte operand is being fetched.

6.2.5.3 Destination Mode 3 – The three sequences for destination mode 3 all enter the D30.1 (destination mode 3) machine state, which completes the data transfer, increments the destination register by the necessary amount, and transfers to the D10.2 machine state, which begins the fetch of the operand addressed by the word just transferred. Because the first transfer during a destination mode 3 sequence can only be a full word, the increment used in the register update is always 2, not 1.

6.2.6 Destination Modes 4 Through 7

Drawing D-FD-KB11-A-03 (Flows 6) illustrates six machine state sequences that are used to fetch the destination operand when the destination address mode is 4, 5, 6, or 7. These six sequences correspond to the six sequences described in Paragraph 6.2.5 for address modes 1, 2, and 3.

The four destination modes are divided into two pairs: modes 4 and 5, which require that the contents of the destination register be decremented before the value is used in the external data transfer, are treated by one of three sequences; modes 6 and 7, which use general register 7 (the PC) first and then use the destination register, are treated by one of three sequences. In either case, two of the three sequences are entered from the fork C decision point, and one from the fork A decision point. The two fork C entries differentiate between source operands that require byte swapping and source operands that do not. There can be no requirement for a byte swap on the fork A entry, because the source operand, if any, must be address mode 0 and the high byte of a register can not be specified.

6.2.6.1 Fork C Entries for Modes 4 and 5 – Machine states D45.8 (destination mode 4 or 5) and D45.9 differ mainly in the microprogram addresses contained in the microprogram word. Each state decrements the DR by the value of the destination constant, which is 1 for a byte operation in mode 4, and 2 for a word operation. Byte operations in mode 5 use a constant of 2 because the data fetched from the address taken from the DR is, in turn, used as an address and must be a full word. The state following D45.8 or D45.9 begins the external data transfer, which may be a DATI, DATIP, or a DATO, depending on the specific instruction. Machine states D40.3 and D50.3, which follow D45.9, also perform the byte-swap operation on the source operand. In each of the two sequences, a different path is taken for destination mode 4 where only one data transfer is needed, than for destination mode 5 where a second transfer is needed. The second transfer is performed by a sequence that is common for address modes 3, 5, and 7; this sequence transfers the first word that is fetched from the BR to the DR and then uses the DR as the address for a second transfer.

6.2.6.2 Fork A Entry for Modes 4 and 5 – Machine state D45.0, which is entered from the fork A decision point, is similar to machine states D45.8 and D45.9, except that a BEND is performed to abort the transfer begun during the IRD.0 machine state. The sequences that follow D45.0 are similar to the sequences that follow D45.8 or D45.9, except that the source operand, if any, is already in the SR.

6.2.6.3 Destination Modes 6 and 7 Entry – For address modes 6 and 7, the first machine state entered from the fork C decision point begins an external data transfer, using the contents of the PC as an address, and performs an increment operation on the PC. The entry from the fork A decision point continues the transfer begun by the IRD.0 machine state, so this entry is to the D67.0 (destination mode 6 or 7) state that follows the first state for the other entries. The D67.1 machine state adds the contents of the DR to the data read into the BR, thus performing the indexing operation, and then transfers to a machine state in the flow sequence for destination modes 4 or 5. The transfer is to D10.3 (a state also used for mode 4) if the mode is 6, or to D10.1 (a state also used for mode 5) if the mode is 7. The shared sequences perform the remaining one or two data transfers to fetch or store the actual data word.

6.2.6.4 Ending Sequence – When the last data transfer has been started, all six sequences enter a combined conditional fork and three-way branch that selects the next machine state. For MOV, MTPI, and MTPD instructions, the last data transfer is a DATO operation, which is completed by machine state D10.4; this state also loads the condition codes. The processor then returns to the instruction fetch sequence. For all other instructions, the DATI or DATIP transfer is completed in machine state D10.6, leaving the destination data in the BR and the

source data in the SR, and the fork B logic is conditionally enabled. If a byte-swap operation is required for the destination data, the D12.3 machine state, which performs this operation for all destination modes 1 through 7, is entered.

6.2.7 ASH, ASHC, and Floating-Point Instructions

Drawing D-FD-KB11-A-03 (Flows 7) illustrates the machine state sequences for the arithmetic shift (ASH) and arithmetic shift combined (ASHC) instructions, and the first machine state of the floating-point instruction service after the destination address calculation.

6.2.7.1 ASH Instruction – When the machine state sequence for the ASH instruction is entered from the fork B decision point, the destination data is in the BR register. The six least-significant bits of the destination word are used as a 2's complement number which is the shift count for the instruction. This data is loaded into the shift counter (SC) from the DR, so the DR is loaded from the BR in the first machine state (ASH.0). This state also maintains the address and transfer (bus conditions) information used during the preceding data transfer, so that all deskewing delays can be completed. The following state performs the loading of the SC and stops all external data transfer activity; in a third machine state, the condition codes are loaded based on the value of the word in the source register, and the shift counter is tested for a 0 shift count. If the shift count is 0, the instruction is completed, and the processor returns to the instruction fetch sequence; otherwise, one of two states is entered, depending on the sign of the shift count. Machine states ASH.3 (arithmetic shift) and ASH.4 perform the actual shift one bit at a time, and increment or decrement, respectively, the shift counter. These states also load the KB11-A condition codes with the results of each shift, so that after the last shift the codes are correct, and test during each cycle to determine whether any further cycles are required. Note that the first change to the SC is performed in the ASH.3 machine state; all tests are done on the value before any changes are performed, so the last cycle in ASH.3 or ASH.4 is performed with the SC equal to 0, and the final value in the SC is -0 (all 1s).

6.2.7.2 ASHC Instruction – The ASHC instruction operates in a manner similar to the ASH instruction, with differences to account for the fact that two words of data are shifted. The first two machine states for the ASHC instruction perform the same functions as the ASH.0 and ASH.1 machine states, and in addition, load the DR (after the SC has been loaded from the previous value in the DR) with the contents of a general register which is selected by ORing the destination register specification with 1. When the destination register specified by the instruction is an even-numbered register, the OR produces the number of the next higher numbered register.

6.2.7.3 Condition Code Loading – The third machine state for ASHC instructions performs the first change of the SC, moves the first data word to the BR, loads the condition codes, and tests for a 0 SC, just as machine state ASH.3 does. However, if the SC is 0, the sequence continues with the ASC.8 (arithmetic shift combined) machine state, instead of returning immediately to the instruction fetch sequence. This state is required to test the second data word, so that the Z condition code can be set on the contents of both words. The ASC.8 machine state also starts the next instruction fetch, so the processor transfers to either FET.1 or BRK.0 rather than FET.0.

6.2.7.4 ASHC Processing – If the SC is not 0, the ASC.2 machine state is followed by the ASC.3 or ASC.4 state. These states perform the same operations as the corresponding states for the ASH instruction, and also cause shifting of the DR (which can be shifted internally, without passing the data through the ALU or SHFR). The bit shifted into the DR is selected by processor hardware. When the SC does reach 0, the next machine state is ASC.6, which performs the same operations as ASC.8, but also stores the second word from the DR into the appropriate general register.

6.2.7.5 **Floating-Point Instructions** – When the fork B logic decodes a floating-point instruction, the FOP.4 (floating-point operation) machine state is entered. This state aborts the last external data transfer started by the destination-data-fetch sequence, and sends the destination address, not the destination data, to the FPP. The sequence then continues with the floating-point service machine states to perform whatever operations are required by the FPP.

6.2.8 Multiply Instruction

The sequence of machine states shown on drawing D-FD-KB11-A-03 (Flows 8) performs a multiplication operation on two words of data, one from a general register and the other in a word specified by a destination field and fetched into the BR. The results of the multiplication are stored in two general registers; one is the register specified in the instruction, and the other is a register whose number is formed by ORing 1 with the number of the specified register (Figure 6-8). (If the specified register has an odd number, only one register is used.)

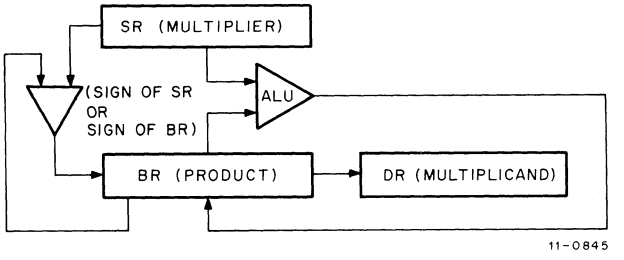


Figure 6-8 Multiply Instruction

The multiplication is performed by an add and shift algorithm as described in Chapter 4. However, in the implementation used, several features of the algorithm are obscured by stratagems used to reduce the amount of logic necessary for the operation. The two most important stratagems are the use of a 16-bit multiplier instead of a 32-bit multiplier, and the accumulation of the product in a register that expands from 16 bits to 32 bits as the multiplication proceeds.

The multiplication algorithm described in Chapter 4 uses a 32-bit multiplier because the double-length multiplier reduces the need for corrections for negative numbers. However, the 32-bit multiplier differs from the original 16-bit multiplier only in having the sign extended into 16 additional bits of greater significance. The effect of this difference on the partial products generated during the multiplication is simply to make the partial product take on the same sign as the multiplier. During the multiplication, therefore, whenever a new partial product is formed by adding the multiplier to the current partial product, the sign of the multiplier is shifted into the partial product after the addition.

As shown in Chapter 4, the maximum number of significant bits in the multiplier (not counting the extended sign) is 16; therefore, the multiplier is loaded into the source register (SR), and all sign extension is done during the shifting of the product. The number of significant bits in the product varies during the multiplication, increasing by one for each cycle; the number of bits in the multiplicand, which must be saved for future cycles, is reduced by one each cycle. The DR performs two functions: at the beginning of the multiplication, it holds the 16-bit multiplicand, but as the multiplicand is shifted out of the DR, each vacated bit is used for the expanding product. Because the bits available are the most-significant bits of the DR, while the product expands by adding bits to the most-significant end, the product is stored by shifting it, as it expands, from the BR into the DR. Instead of shifting the SR with respect to the BR (as the multiplier shifted with respect to the product in Chapter 4), the BR shifts with respect to the SR.

6.2.8.1 Multiplication Setup – The multiplication sequence begins with two machine states that set up the four registers (BR, SR, DR, and SC) used in the sequence, and perform the first test and shift on the DR. Note that all branches refer to the state of the DR and the SC at the beginning of the machine state preceding the branch, not the values in the registers at the end of that state. The operand supplied by the destination-data-fetch sequence is loaded into the DR. and the SC is loaded with the octal value 17 in machine state MUL.0 (multiply). In machine state MUL.1, the BR is cleared; the other operand is assumed to be in the SR as the result of the IRD.0 machine state.

6.2.8.2 Multiplication Process – The multiplication proceeds through 15 cycles of shifting performed by machine states MUL.2 and MUL.3. The branch conditions that select which of these two states is entered are the appropriate conditions to determine whether an addition and shift, or just a shift, is performed.

6.2.8.3 Multiplication Correction – Following the 15 cycles of shifting, either state MUL.2 or state MUL.4 is entered, depending on the need for a correction. The total number of shifts performed during a multiplication is 16 for the BR and 17 for the DR; however, only the last 16 shifts transfer significant bits to the DR.

The MUL.2 or MUL.4 machine state stores the more-significant half of the result into the register specified by the source field, and sets the condition codes on the value of this word. The MUL.5 machine state stores the less-significant half of the result in the register whose number if formed by ORing the source field with 1; if an odd register is specified, this value replaces the more-significant half of the result, which is lost. This is done because many multiplications produce a result which can be contained in only one word, and this result is preserved by this action. The condition codes are altered to represent the value of the entire result; if all 32 bits are 0, the Z bit is set, and if the result can not be contained in one word, the C bit is set.

The final state of the multiply sequence returns either to the instruction fetch sequence, or, if an asynchronous condition needing service was sensed by the BRQ STROBE in machine state MUL.2 or MUL.5, to the break service sequence.

6.2.9 Divide Instruction Sequence

The divide (DIV) instruction is executed by the longest and most complex sequence of machine states used in the KB11-A processor. This sequence is illustrated on two drawings. Drawing D-FD-KB11-A-03 (Flows 9) shows the register setup, the first two overflow tests, and the cycle of states that performs the actual division. Drawing D-FD-KB11-A-03 (Flows 10) shows the quotient and remainder sign corrections and the final overflow test.

The division is performed by a non-restoring divide algorithm that is described in Chapter 4. The hardware implementation (Figure 6-9) uses the SR to hold the divisor and begins with the dividend in the BR and DR registers. The BR contains the more-significant half of the dividend, while the less-significant half is in the DR. Each cycle of the division shifts the dividend one bit to the left and shifts the next bit of the quotient into the least-significant bit of the DR. When the division terminates, the quotient is in the DR and the remainder is in the BR.

The non-restoring divide algorithm can operate with positive or negative operands; however, the KB11-A always operates on a positive dividend to simplify the detection of underflow. The divisor may have either sign. The first two machine states of the division sequence test for a 0 divisor or a negative dividend, and set up the SR and DR registers. If a 0 divisor is sensed, the division is aborted and the C, V, and Z condition codes are set to indicate that an error has occurred.

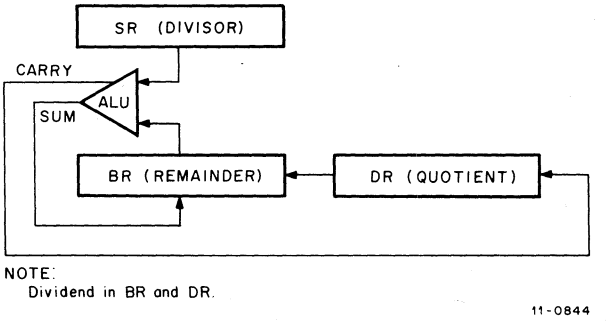


Figure 6-9 Divide Instruction

6.2.9.1 Initial Setup – If the dividend is negative, a sequence is entered to complement the dividend. Note that the branch on the N condition code occurs after machine state DIV.2, although the condition code is loaded in state DIV.1 (divide), because the branch condition must be available at the beginning of the machine state in which the branch is used. Similarly, the branch on the Z condition code after state DIV.1 uses the condition code value set by state DIV.0, not the new value set by the DIV.1 state.

6.2.9.2 Negative Dividend Processing – The sequence beginning with machine state DVN.0 (divide negation) generates the 2's complement of the 2-word dividend as follows:

a. The 2's complement of the less-significant word is formed by first clearing the DR, then subtracting the SR, which contains the low order word, from the 0 in the DR. The DR is cleared so that a subtract from 0, which requires only one machine state, can be used; normally a 2's complement is generated by forming the 1's complement and then incrementing, as shown for the remainder correction steps. The 2's complement of the less-significant word is stored in the register which originally held the less-significant word.

b. Machine state DVN.2 generates a carry from the less-significant word to the more-significant word. That is, if a carry-out of the most-significant bit of the ALU occurs during the operations (which is repeated in machine state DVN.2), a 1 is shifted into the DR.

c. A 1 is subtracted from the DR. If a carry occurred in Step b, the DR contains 0 and the 2's complement of the more-significant word is formed; if no carry occurred, the DR now contains a -1, which cancels the carry insert during the subtraction in machine state DVN.4, and the 1's complement of the SR is formed. This is the correct result if there is no carry.

After the 2's complement of the dividend is formed, the DVN.5 machine state begins the restoration of the divisor to the SR and the dividend to the BR and DR. However, if the dividend is still negative, which occurs if the dividend was the maximum negative number (because the 2's complement notation can express one more negative number than positive number, the largest negative number complements to itself), the division can not be performed and the sequence is aborted.

6.2.9.3 Overflow Test and First Cycle – After the setup is completed, the processor enters the DIV.3 machine state with a positive dividend in the BR and DR, 17_8 in the SC, and the divisor in the SR. The next portion of the sequence performs the first cycle of the division and performs a test for overflow. This test is based on the fact that if underflow does not occur during the first cycle, the quotient is too large to be expressed in 16 bits. If the instruction is not aborted because of overflow, the processor enters the DIV.7 machine state to begin the main divide cycle.

6.2.9.4 Division Process – The test for underflow that determines whether machine state DIV.8 or state DIV.9 is entered is based on the following considerations:

a. If the divisor is negative, adding the divisor to the dividend should produce a result closer to 0 than the original dividend. If the result is negative, underflow has occurred and a 0 is shifted into the DR.

b. If the divisor is negative and the dividend is also negative, an underflow condition already exists. The divisor is subtracted from the dividend to return the dividend to a positive number. If the result is still negative, a 0 is shifted into the DR; if the result is positive, the underflow has been corrected and a 1 is shifted in.

c. For a positive divisor and dividend, a subtraction is performed. If the result is positive, a 1 is shifted into the DR, but if the result is negative, underflow has occurred and a 0 is shifted in.

d. If the divisor is positive and the dividend is negative, an addition is performed to correct an existing underflow. If the result is positive, the underflow has been corrected and a 1 is shifted into the DR, otherwise a 0 is shifted in.

As a result of these considerations, the processor enters machine state DIV.8 if the divisor is positive and there is no underflow (DR0 is a 1), or if the divisor is negative and there is underflow (DR0 is a 0). State DIV.8 performs a subtract operation and shifts the carry-out of the ALU into the DR. (A carry-out of the most-significant bit of the ALU indicates that underflow has occurred; if an uncorrected underflow existed, the carry indicates that it has been corrected.)

If the opposite conditions exist (SR is positive and DR0 is 0, or SR is negative and DR0 is 1), state DIV.9 is entered and an addition is performed, followed by a shift of the DR. Note that the cases for which a carry-out of the most-significant bit of the ALU exists are equivalent to the cases described above for which the least-significant bit of the DR is set.

6.2.9.5 Remainder Storage and Sign Check – After the divide cycle has been performed 15 times (the first division cycle, and the first decrement of the SC, is performed in machine states DIV.3 through DIV.6), the DVC.0 (divide correction) machine state writes the remainder from the BR into the appropriate general register, and transfers control to one of four machine states, depending on whether a remainder correction is required and whether the quotient has the correct sign.

6.2.9.6 Remainder Correction – If, after the last division cycle, the least-significant bit of the quotient is a 0, an underflow condition still exists. This condition can be corrected (unless an overflow condition also exists) by adding a positive divisor or subtracting a negative divisor to correct the remainder. This is done by machine state DVC.1 or DVC.2. If no remainder correction is needed, or following the remainder correction, the DVC.3 or DVC.4 machine state begins the complementing of the remainder in case the remainder has the wrong sign. The current value of the remainder is not disturbed until a determination is made of the appropriate sign.

6.2.9.7 Quotient Sign Change – If the N condition code is set, the original dividend was negative. The complemented remainder, which is negative because the corrected remainder is positive (if all underflow conditions are corrected), is stored as the final value of the remainder. If both the dividend and the divisor were positive, the quotient, which is also positive (the most-significant bit of the quotient must be positive or an immediate overflow condition aborts the division), is written into the appropriate general register. Similarly, if both dividend and divisor are negative, the quotient should be positive and is written in its present form.

If the original signs of the dividend and divisor were different, the quotient should be negative. The quotient is complemented by the machine states DVC.8 and DVC.9; one special case in which the quotient is the most negative number is considered an error.

6.2.10 Memory Reference Execution Sequences

Drawing D-FD-KB11-A-03 (Flows 11) illustrates nine sequences that execute the data manipulation stages of a variety of PDP-11 instructions, when those instructions require external data transfers to complete the instruction execution. These sequences are entered from the fork B decision point.

6.2.10.1 Standard Execution – The majority of the PDP-11 instructions are executed by the EXC.0 (execute) machine state. When this state is entered, the source operand, if any, is in the SR, and the destination operand is in the DR. The EXC.0 state performs one data manipulation operation and loads the condition codes; both the operation performed and the condition-code loading are controlled by subsidiary ROMs (i.e., they are instruction-dependent). For any instruction that is operating on an odd-byte destination operand, the EXC.0 state performs the byte-swap operation in the SHFR automatically.

The EXC.0 machine state also begins an external data transfer operation that is completed in the EXC.1 machine state; this operation transfers the result data to the destination address, which is taken from the DR.

6.2.10.2 Negate Instructions – Several instructions, which are otherwise treated in the same manner as those executed by the EXC.0 state, must be executed separately. The negate and negate byte (NEG.B) instructions require two machine states for execution because the 2's complement of a number is formed by first generating the 1's complement and then incrementing that value. After the negation is performed and the condition codes loaded, the processor performs a byte swap if the destination operand is an odd byte, and starts an external data transfer that is completed in machine state EXC.1.

6.2.10.3 Shifter Instructions – Two instructions, which are normally executed by the EXC.0 machine state, use the SHFR to perform a right shift. These are the ASR and ROR instructions. When these instructions operate on a destination operand taken from an odd-byte location, a second machine state is required to perform the byte swap, which also requires the SHFR. Therefore, the SHR.0 (shift right) machine state performs the same actions as the EXC.0 state, except that no external data transfer is begun and no byte swap is performed. These functions are performed by the SHR.1 machine state. No conflict occurs for the ASL and ROL instructions because left shifts are performed by the ALU, not by the SHFR.

6.2.10.4 Test Instructions – The three instructions that set the condition codes without modifying any stored data, TST, CMP, and BIT, are executed by machine states that do not start an external data transfer for the data operand. To further speed up the operation of the processor, a test is made to determine if the previous data transfer used only the Fastbus. If this is the case, the bus address and control lines do not need to be maintained in their previous conditions, and the fetch of the next instruction is started. When the data transfer uses the Unibus, the bus address and control lines must not be modified until all deskewing is completed, so the sequence transfers to the FET.0 machine state after the TST.0 (test) state.

6.2.10.5 Jump Instruction – The jump (JMP) instruction performs only one operation; it sets a new value in the program counter (PC). The value loaded into the PC is the destination address, not the destination data word. The last external data transfer to fetch the data word is aborted, the PC is loaded, and a transfer to the instruction fetch sequence is performed by the machine state JMP.0 (jump).

6.2.10.6 Jump to Subroutine Instruction – The jump to subroutine (JSR) instruction performs two data transfers in addition to loading the PC. The contents of a register specified by the instruction are saved on the hardware stack, and the previous value in the PC is saved in the specified register. The JSR.0 (jump to subroutine) machine state aborts the last external data transfer, loads the destination address into the PCA (but does not

load the PCB from the PCA, so that the PCB can be stored in the general register), and loads the SR with the contents of the specified register. The JSR.1 machine state transfers the SR to the BR, which is the register that holds data to be transmitted during external data transfers, and loads the DR with the contents of general register 6, the stack pointer (SP). JSR.2 decrements the SP by 2 (to allocate a word at the top of the stack for the data to be stored); the new value is stored in the SP and in the DR for use in the external data transfer begun in JSR.3. The JSR.3 state also transfers the contents of the PCB to the specified general register and loads the PCB from the PCA. The data transfer begun in the JSR.3 state is completed by the SVC.0 (service) machine state, which performs the same function for interrupt and trap sequences.

6.2.10.7 Move From Previous Space Instructions – The MFPI or MFPD instruction transfers data from the destination address to the hardware stack; it acts like a "push" instruction. If the KT11 Memory Management Unit is operating, the address space from which the destination data is taken may differ from the address space that the data is pushed into, but this does not affect the operations within the processor. The MFP.0 state is entered with the data to be transferred in the BR; this state loads the condition codes and loads the SR from the hardware stack pointer. The MFP.8 machine state is entered if the destination mode is 0; this implies that the data is in a general register. This data is loaded into the DR while the bus operation started by the IRD.0 machine state is aborted. The MFP.9 machine state transfers the DR to the BR and loads the SR from the stack pointer. The sequence for destination mode 0 then joins the sequence for the other address modes.

6.2.11 Break Conditions Sequences

Drawing D-FD-KB11-A-03 (Flows 12) illustrates three sequences that are used to set up the new processor state in response to various asynchronous requirements and to execute services for some of these requirements.

6.2.11.1 Abort Sequence – The major machine state sequence illustrated is entered after an operation is aborted (at machine state ZAP.0) or in response to an external bus request or internal console request (at machine state BRK.0). If an operation is aborted, the processor stops any external data transfers in machine state ZAP.0 and control transfers either to the ZAP.1 machine state or to the BRK.0 (break) machine state. If the aborted operation was a data transfer attempting to store the processor status word (this occurs if a trap or interrupt service sequence caused a stack limit, bus error, or relocation address error trap), the old processor status word is rewritten from the BR to the PS by a sequence of three machine states. The machine state sequence then rejoins the main sequence through state BRK.0.

6.2.11.2 Break Sequence – In the BRK.0 state, the processor transfers the PC value to the SR in case the PC value must be stacked, and strobes the bus data into the BR in case an interrupt operation has supplied an interrupt vector. The BRK.0 state also determines whether the console flag is set. This flag can be set by a HALT instruction, by the HALT switch on the console, or by a parity error in the solid state memory. Once the console flag is set, it remains set until cleared by a START or CONT switch operation. The processor cycles in the CON.0 machine state when the flag is set.

If the BRK.0 state has been entered with the console flag not set, the sequence continues with the BRK.1 machine state, which loads the DR from the BR in case the BR contained an interrupt vector, and performs a 4-way branch that depends on the type of condition that caused the sequence to be entered.

6.2.11.3 Power-Up Sequence – If the break condition is a power-up request, the power-up sequence is entered starting at machine state PUP.0 (power up). This sequence loads the DR from the hard-wired start vector and then loads the PC and the PS from the addresses pointed to by the start vector. The previous contents of the PC and PS are not stacked because they are not significant, and because the stack pointer is also not reliable;

a power-up sequence must assume that all registers have been altered when the power to the processor has been off. The RTI.5 major state is entered because it performs the operations required to complete the sequence; it loads the PS with the data loaded into the BR during state PUP.4 and the sequence then continues until a new instruction is fetched.

6.2.11.4 Internal Traps – Internal traps are those conditions that, when recognized by logic in the processor, cause the abortion of a sequence or the abortion of the next instruction fetch. Trap-type instructions are handled by the sequence beginning with the TRP.0 (trap) machine state. The internal traps are divided into two groups; one group includes all errors that cause a failure of a stacking operation and require the use of an emergency stack, while the other group handles all trap conditions that use the normal hardware stack.

6.2.11.5 Stack Errors – When a stack limit error occurs, or a bus error or memory management error on a stacking operation occurs, this error must be serviced without the use of the normal hardware stack. Therefore, an emergency stack is set up, using only the two words at addresses 0 and 2. The stack pointer is loaded with the value 4 because the stacking operation decrements the stack pointer by 2 before each stacking transfer. This value is generated by loading 2 into the SR, then adding 2 again and storing the sum in the stack pointer. The stack error sequence then joins the internal error sequence at machine state BRK.8.

6.2.11.6 Internal Vector Generation – For all internal traps, the BRK.8 machine state sets a trap vector in the DR, and loads the BR with the old PS word. The BRK.3 machine state then begins an external data transfer to load the PC with the value stored at the trap vector address, and loads the PCA from the BR so that the BR can be used to receive the new PC word. The processor now has the old PC in the PCB register and the old PS in the PCA register. The external data transfer is completed and the remainder of the trap or interrupt service sequence is performed by the SVC machine states (Flows 13).

6.2.11.7 Interrupts – All external break conditions, other than the ones discussed above, are assumed to be interrupts from other devices. These conditions supply an interrupt vector on the bus data lines; this vector has been loaded into the BR, and then into the DR, by the BRK.0 and BRK.1 states. The BRK.2 machine state loads the BR from the old PS word, in the same way as the BRK.8 state, and tests for a valid break condition. If no condition exists, the processor returns to the instruction fetch sequence through the RTI.6 machine state, which clears the various flags that might have caused a break condition to be sensed. If the break condition is valid, the BRK.5 machine state begins the interrupt service sequence.

6.2.11.8 Floating-Point Instructions – When the execution of a floating-point instruction by the FPP has been initiated, the processor enters a floating-point service sequence, beginning with machine state FSV.2 (floating-point service). When this state is entered, the DR contains a destination address and the floating-point instruction has been transferred to the FPP. The FSV.2 machine state performs no operations; the BRQ STROBE is required only for the last state preceding the instruction fetch sequence. The processor waits, repeating the FSV.2 state, until the FPP sends a synchronization signal, and then performs a bus transfer if the FPP requests one, or returns to the instruction fetch sequence if no operation is required.

If an external data transfer is required, the FPP sends a request at the same time as the FP SYNC. The transfer may be in either direction; from the FPP to the external storage locations, or from storage to the FPP. In the FSV.0 machine state, the FPP supplies the bus control signals and a bus operation is started using the address in the DR. The BR is loaded from the internal bus, in case the FPP is supplying a word of data for transmission to a storage location; if this occurs, the bus control signals supplied by the FPP also gate the contents of the BR to the external bus.

The FSV.1 machine state completes the bus operation and loads the BR from the external bus, in case the operation is a transfer to the FPP. If the transfer is an FPP, the data is gated from the BR onto the internal bus for use by the FPP, and the FPP can read the data when the FPATTN signal is transmitted. The DR is updated in case the FPP requires additional words of data. The general register specified in the instruction, from which the DR was loaded, is not accessed because the general register was updated by the total amount necessary during the destination address calculation states. After each transfer, the processor waits for the FP SYNC signal before proceeding.

6.2.11.9 Trap Instructions – For trap instructions, the sequence used to initiate the trap service sequence differs from the sequence for internal trap conditions in two respects. First, this sequence must abort any bus operations that have been started; second, the sequence does not generate an acknowledge signal to clear all internal trap conditions. Therefore, two machine states that are otherwise similar to BRK.8 and BRK.3 are used for trap instructions; these are TRP.0 and TRP.1, respectively. The same service sequence is used following TRP.1 as following BRK.3.

6.2.12 The Service Sequence

The machine states illustrated on drawing D-FD-KB11-A-03 (Flows 13) perform the stacking of the old processor state and the loading of a new processor state for various trap and interrupt conditions. The sequence performs four external data transfers, in the following order:

a. The word addressed by a vector in the DR is read and transferred to the PC.

b. The word following the first word in the address space is read and transferred to the PS.

c. The old PS, which has been stored first in the PC, then in the BR, is written in the top word of the hardware stack.

d. The old PC, which has been stored first in the DR, then in the BR, is written in the new top of the hardware stack.

The processor stores the old values until the new values are loaded into the correct registers so that the new PS value can specify the processor operating mode, and therefore, the processor stack pointer (SP) to be used in storing the old values.

6.2.12.1 PC Fetch – The first machine state of the sequence, SVC.0, completes the first data transfer, reading the new PC into the BR from the external bus. The vector address used in the transfer is in the DR; the next address is calculated by adding 2 to the value in the DR and this address is stored in the SR.

6.2.12.2 PS Fetch – The next two states of the sequence, SVC.2 and SVC.3, perform a second external data transfer using the SR as an address source. The new PS value is read into the BR; the previous contents of the BR have already been loaded into the PC, while the old PC has been transferred to the DR. The double buffering of the PC allows the PCB register to hold the old PS word, while the PCA holds the new PC.

6.2.12.3 Stacking Setup – The SVC.3 machine state loads the PS from the BR and saves the previous processor mode in the PS. The SVC.4 state prepares the processor for the third external data transfer by moving the old PS value from the PCB to the BR and reading the SP specified by the new PS word into the SR for use as an address. In the SVC.5 state, the SP is decremented by 2 to point to the new top of the stack, and the new PC value is loaded into the second PC register.

6.2.12.4 Stacking the Old Values – The SVC.6 and SVC.7 states perform the third external data transfer, writing the old PS from the BR to the top of the hardware stack. The SP is again decremented by 2, to point to the next top of the stack for the old PC word; this value does not disturb the SR which is being used as the address source during the transfer. As the transfer is completed, the old PC is transferred from the DR to the BR, and the DR is loaded with the final value of the SP, so that the SVC.8 and SVC.9 machine states can perform the final data transfer to push the old PC onto the top of the hardware stack. Finally, the processor returns to the instruction fetch sequence to fetch the instruction pointed to by the new PC.

6.2.13 Console Operation Sequences

Normally, the KB11-A processor performs operations specified by stored programs and operates in a continuous sequence without manual intervention. However, the processor can be controlled manually by the console switches; these switches signal the processor to perform certain operations that are executed by the machine state sequences illustrated on drawing D-FD-KB11-A-03 (Flows 14).

6.2.13.1 Processor Rest State – In order for the processor to perform any console switch operations, the processor must be in the console state CON.0 (console) to recognize which operation is required and to enter the appropriate machine state sequence. The CON.0 state is entered whenever a halt instruction is executed (in kernel mode), the HALT switch on the console is set (and the PANEL LOCK switch is off), or a memory parity error occurs (and parity halt is enabled). In any of these cases, the sequences that transfer to the CON.0 state load the current PC into the SR and the contents of general register 0 into the DR.

A special 10-way branch is used to select the machine state sequence following the CON.0 state. The branch decodes the console switches and the previous console operation to select a sequence; if no switch is activated, the processor loops in the CON.0 state. Each switch activation causes one execution of the corresponding sequence; no further sequences are executed until the switch is released and any switch is again activated.

Many of the sequences shown on this drawing require data supplied from the console. This data is supplied by a set of switches on the console, through the BR register; the BR is loaded from the switches during the CON.0 (console) state.

6.2.13.2 Load Address Function – The console operations that transfer data to the processor or to storage locations require an address to specify the location where the data is stored. This address is taken from the SR register. The SR can be loaded with a new value by the load address (LOAD ADRS) function, which is executed by the ADR.0 (address) machine state. This state transfers the contents of the BR (which is equivalent to the value in the switch register) to the SR; it also loads the PCA register so that any START sequence following the load address sequence uses the address that was loaded for the first instruction fetch. Following the ADR.0 state, the processor enters the CON.2 state to clear the switch activation and return to the CON.0 state. The new address is in the SR and the DR is unchanged.

6.2.13.3 Register Examine and Deposit – Console operations can also be used to read the contents of, or load a new value into, either the external storage locations or the processor general registers. The general register operations utilize the address stored in the SR by transferring the contents of the SR to the IR, which permits decoding the least-significant bits of the address as a register number. The specific register addressed can be loaded from the BR, or the contents of the specific register can be transferred to the DR and displayed during the CON.0 state. If the PC (general register number 7) is addressed, both the PCA and the PCB are loaded, which destroys the old PC value at the time of the halt. Following any operation on a general register, the SR contains the address which specified the register and the DR contains the current contents of the register.

6.2.13.4 Memory Examine and Deposit – The external storage locations can be examined and loaded in the same manner as the general registers. In addition, the storage examine (EXAM) and deposit (DEP) functions can be repeated, with automatic adjusting of the address in the SR, to access successive words of the storage address space. The address incrementation is controlled as follows: whenever the EXAM or DEP switch is activated, the processor determines whether the previous console operation was the same operation currently being requested; if it was, the address is incremented before the external data transfer occurs (the function is examine step or deposit step); otherwise, no incrementation takes place (the function is examine or deposit).

The deposit function, which begins with the DEP.1 machine state, performs an external data transfer, specifying a DATO bus operation, and uses the SR as the address and the BR as the data. The deposit step function uses the same machine states, following the DEP.0 (deposit) state, which increments the SR. Following either function, the SR contains the address into which the data was deposited and the DR contains the data that was transferred.

The examine function, which begins with the EXM.1 (examine) machine state, performs one external data transfer, specifying a DATI bus operation; the address is taken from the SR and the data is loaded into the BR. An additional state, EXM.3, is required to load the DR from the BR. The examine step function uses the same machine states preceded by the EXM.0 state to increment the SR. Following either sequence, the SR contains the address which was read and the DR contains the contents of that address.

6.2.13.5 Start Operation – The KB11-A processor can re-enter the normal instruction interpretation mode as the result of either of two console switch operations. The START switch operation forces the next instruction to be taken from the address specified by a previous LOAD ADRS or REG DEP switch to register 7 operation; if no LOAD ADRS or REG DEP operation was done, the processor continues with the next instruction following the instruction executed before the processor entered the console mode. The continue operation always continues with the instruction following the one executed immediately before entering console mode, unless general register 7 has been explicitly modified.

The KST.0 (key start) machine state executes the start function by loading the PCB from the PCA register and transferring to the RES.0 (reset) machine state; the latter state begins a sequence that initializes the state of the processor and the system before fetching the next instruction from the address loaded in KST.0

6.2.13.6 Continue Functions – The CON.1 (continue) machine state returns the processor to the instruction fetch sequence by a sequence that consists of the BRK.1, BRK.2, and RTI.6 machine states. If the console ENABL/HALT switch is in the ENABL position, the processor continues with the normal operation sequences; if the switch is in the HALT position, and the console S INST/S BUS CYCLE switch is in the S INST position, the processor returns to the CON.0 state after the execution of one instruction.

If the console switches are in the HALT and S BUS CYCLE positions, the KB11-A processor does not enter the console states. Instead, the processor clock actually stops in the machine state following the completion of an external data transfer. No console operations other than a CONT (which simply restarts the processor clock) can be performed.

6.3 FOLLOWING AN INSTRUCTION THROUGH THE FLOWCHARTS

The preceding section described the flowcharts in detail, in the order of the appearance of the machine state sequences on the drawings. To follow a particular instruction through the flowcharts, the reader must know which machine state sequences apply to that instruction in the particular state of the processor; specifically, the reader must know which machine state will be entered from various fork decision points.

The tables and diagrams in this paragraph are designed to help the reader determine the exact sequence of machine states for a particular instruction. Starting with either the binary code of an instruction or the symbolic name of the instruction, the reader can determine what machine state is entered from each decision point, and what branches are taken at some of the primary branch points within the sequences shown.

Figure 6-10 illustrates the correspondence between the binary value of a 16-bit word and the instruction that is represented by that value. If the reader is starting from a binary code (e.g., if the reader is stepping through the machine states using the maintenance module manual clock, and has an instruction in the IR (and the BR) at the end of the FET.1 machine state), the procedure to determine what instruction the code represents is as follows:

a. Starting with the most-significant bit of the instruction code, look down the corresponding column of Figure 6-10 to find the number that matches the value of that bit in the instruction.

b. The horizontal line to the right of that number leads to another vertical column, for the next most-significant group of bits in the binary code. Look down that line to find the number that matches the value of the corresponding bit or bits in the instruction.

c. Repeat Step b for each portion of the binary code until the last number is followed by the symbolic name and structure of an instruction instead of a horizontal line. That instruction corresponds to the given binary code.

When the symbolic code for an instruction is known, the reader can find that instruction in Table 6-4, which specifies the machine state sequences used to execute that instruction. The table is in alphabetical order according to the mnemonic codes used for the instructions, and lists both the instruction classes, if any, and the machine states entered from various decision points, when used. The instruction classes are groupings of the instructions according to properties of the execution sequences (e.g., I, P, and O Class instructions perform a DATI, DATIP, or DATO bus transfer as the last transfer of the destination data fetch sequence). While the fork A decision point is used by all instructions (the fork A decision point follows the instruction fetch sequence and is, in effect, the instruction decoding system), not all instructions use the fork B or fork C decision points; those which do not are indicated by the entry "not used" in the appropriate column.

Whenever possible, the entry for each active decision point specifies a machine state by its symbolic name, with the number of the flowchart where that state is illustrated in parentheses. If a particular machine state depends on additional conditions, those conditions are shown preceding the corresponding machine state and are separated from the state by a colon. There are four groups of conditions where the number of machine states that can be entered is too large to be shown directly in the table; these conditions are indicated as follows:

a. In the fork A column, the entry "see Table 6-5" refers to Table 6-5, which illustrates the machine states entered for various source modes.

b. In the fork A column and in Table 6-5, the entry "see Table 6-6" refers to Table 6-6, which illustrates the machine states entered for various destination modes.

c. In the fork A column, the entry "fetch" in the entries for branch instructions indicates a transfer to FET.0 if asynchronous requests require service, and to FET.1 if no asynchronous requests are sensed.

d. In the fork C column, the entry "see Table 6-7" refers to Table 6-7, which illustrates the machine states entered for various destination modes.

To follow an instruction through the flowcharts, first find the instruction in Table 6-4, using Figure 6-10 if necessary. Then, using Tables 6-5 and 6-6 if necessary, follow the instruction through the instruction fetch sequence, the fork A decision point, and the machine state sequence following that decision point. The conditions at any branches in that sequence can be determined from the current processor state; if the branch conditions depend on the instruction class, see Table 6-4. As the machine state sequence passes through other decision points, use Table 6-4 and Table 6-7 as necessary to determine the next state at each decision point.

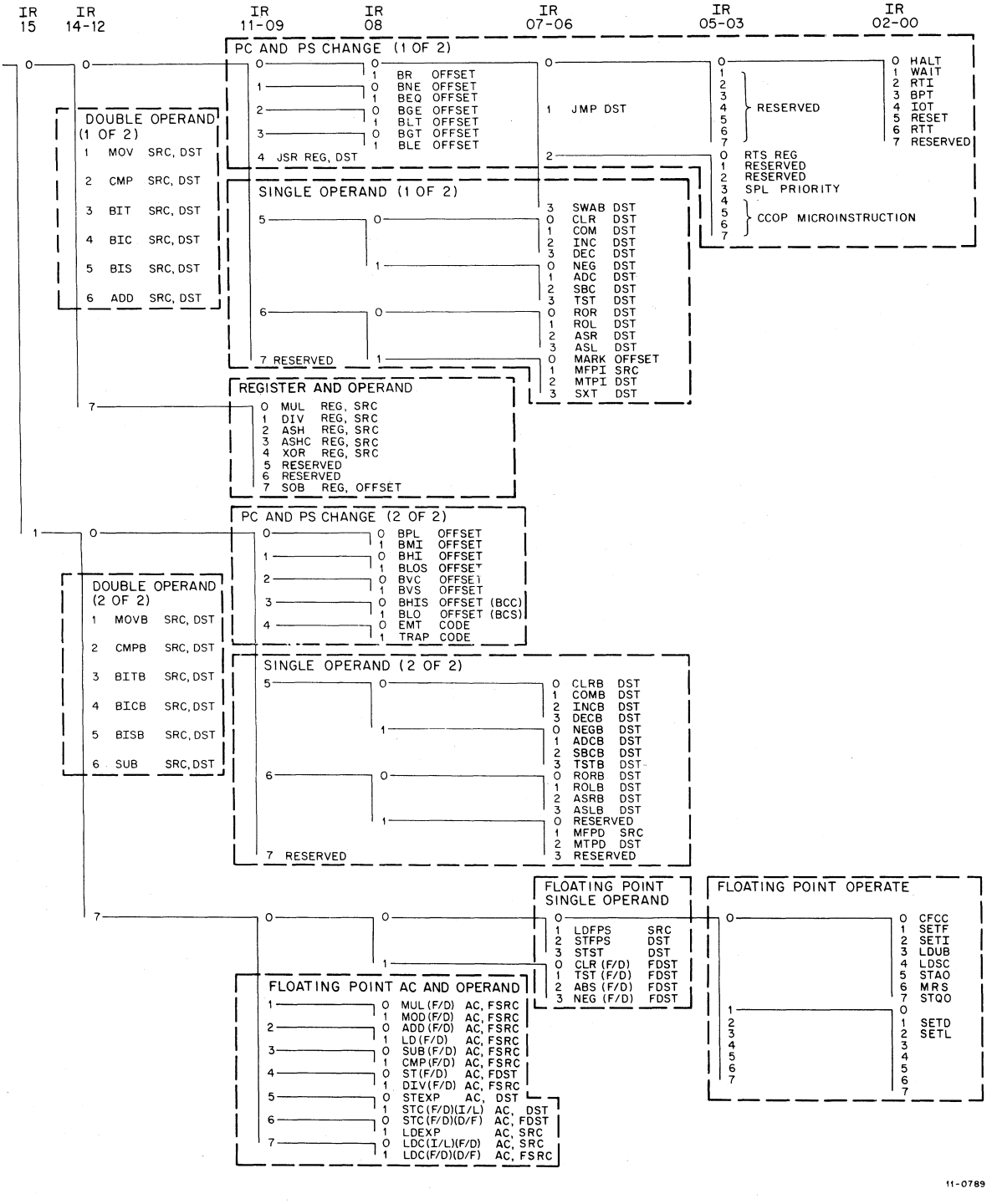


Figure 6-10 Determination of an Instruction from the Binary Code

Table 6-4
Microprogram Instruction Properties

Instruction	Class	Fork A	Fork B	Fork C
ADC or ADCB	P, E	see Table 6-6	EXC.0(11)	not used
ADD	P, E	see Table 6-5	EXC.0(11)	see Table 6-7
ASH	none	see Table 6-6	ASH.0(7)	not used
ASHC	none	see Table 6-6	ASC.0(7)	not used
ASL or ASLB	P, E	see Table 6-6	EXC.0(11)	not used
ASR or ASRB	P, E	see Table 6-6	DR0(1):SHR.0(11), DR0(0):EXC.0(11)	not used
BCC	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BCS	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BEQ	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BGE	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BGT	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BHI	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BHIS	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BIC or BICB	P, E	see Table 6-5	EXC.0(11)	see Table 6-7
BIS or BISB	P, E	see Table 6-5	EXC.0(11)	see Table 6-7
BIT or BITB	I, E	see Table 6-5	FAST:TST.1(11), -FAST:TST.0(11)	see Table 6-7
BLE	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BLO	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BLOS	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BLT	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BMI	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BNE	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used

(continued on next page)

Table 6-4 (Cont)
Microprogram Instruction Properties

Instruction	Class	Fork A	Fork B	Fork C
BPL	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BPT	none	TRP.0(12)	not used	not used
BR	none	BXX.0(1)	not used	not used
BVC	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
BVS	none	BCOK:BXX.0(1), -BCOK:fetch	not used	not used
CCOP	none	CC0.0(3)	not used	not used
CLR or CLRB	P, E	see Table 6-6	EXC.0(11)	not used
CMP or CMPB	I, E	see Table 6-5	FAST:TST.1(11), -FAST:TST.0(11)	see Table 6-7
COM or COMB	P, E	see Table 6-6	EXC.0(11)	not used
DEC or DECB	P, E	see Table 6-6	EXC.0(11)	not used
DIV	I	DM0:DVS.0(3), -DM0; see Table 6-6	DIV.0(9)	not used
EMT	none	TRP.0(12)	not used	not used
FLOATING POINT	F	FOP.0(2)	FOP.4(7)	DM0:FOP.5(5),-DM0; see Table 6-7
HALT	none	HLT.0(3)	not used	not used
INC or INCB	P, E	see Table 6-6	EXC.0(11)	not used
IOT	none	TRP.0(12)	not used	not used
JMP	J	DM0:TRP.0(12), -DM0; see Table 6-6	JMP.0(11)	not used
JSR	J	DM0:TRP.0(12) -DM0; see Table 6-6	JSR.0(11)	not used
MARK	none	MRK.0(2)	not used	not used
MFU	I	DM0:MFU.8(11), -DM0; see Table 6-6	MFU.0(11)	not used
MOV	O, E	see Table 6-5	not used	see Table 6-7
MOVB	E, P	see Table 6-5	not used	see Table 6-7
MTU	O	MTU.0(1)	not used	see Table 6-7

(continued on next page)

Table 6-4 (Cont)
Microprogram Instruction Properties

Instruction	Class	Fork A	Fork B	Fork C
MUL	I	DM0:MUL.8(3), -DM0; see Table 6-6	MUL.0(8)	not used
NEG or NEGB	P	DM0:NEG.7(3), -DM0; see Table 6-6	NEG.0(11)	not used
RESET	none	RES.0(3)	not used	not used
ROL or ROLB	P, E	see Table 6-6	EXC.0(11)	not used
ROR or RORB	P, E	see Table 6-6	DR0(1):SHR.0(11), DR0(0):EXC.0(11)	not used
RTI	none	RTI.0(2)	not used	not used
RTS	none	RTS.0(2)	not used	not used
RTT	none	RTI.0(2)	not used	not used
SBC or SBCB	P, E	see Table 6-6	EXC.0(11)	not used
SOB	none	SOB.0(2)	not used	not used
SPL	none	SPL.0(3)	not used	not used
SUB	P, E	see Table 6-5	EXC.0(11)	see Table 6-7
SWAB	P, E	see Table 6-6	EXC.0(11)	not used
SXT	P, E	see Table 6-6	EXC.0(11)	not used
TRAP	none	TRP.0(12)	not used	not used
TST or TSB	I, E	see Table 6-6	FAST:TST.1(11), -FAST:TST.0(11)	not used
WAIT	none	WAT.0(3)	not used	not used
XOR	P, E	see Table 6-6	EXC.0(11)	not used

Table 6-5
Fork A Binary

Source Mode	Machine State
0	see Table 6-6
1	S13.0 (1)
2	S13.0 (1)
3	S13.0 (1)
4	S45.0 (1)
5	S45.0 (1)
6	S67.0 (2)
7	S67.0 (2)

Table 6-6
Fork A Unary

Destination Mode	Machine State
0	(DF7 + BRQ):EXC.9(3), -(DF7 + BRQ):EXC.8(3)
1	D12.0 (5)
2	D12.0 (5)
3	D30.0 (5)
4	D45.0 (6)
5	D45.0 (6)
6	D67.0 (6)
7	D67.0 (6)

Table 6-7
Fork C Binary

Destination Mode	SR0	Machine State
0	0	DF7:D07.1(4), -DF7:D00.9(4)
	1	DF7:D07.0(4), -DF7:D00.8(4)
1	0	D12.8 (5)
	1	D12.9 (5)
2	0	D12.8 (5)
	1	D12.9 (5)
3	0	D30.8 (5)
	1	D30.9 (5)
4	0	D45.8 (6)
	1	D45.9 (6)
5	0	D45.8 (6)
	1	D45.9 (6)
6	0	D67.8 (6)
	1	D67.9 (6)
7	0	D67.8 (6)
	1	D67.9 (6)

6.4 AN INSTRUCTION EXAMPLE

This paragraph traces one instruction through a sequence of machine states to illustrate the process of finding each machine state and using the flowchart and ROM map information to understand the operations performed by the processor. The example instruction and the environment in which it is executed is shown in Figure 6-11.

```
001000 022767 000015 000100        CMP        #15,        CHAR
                              .
                              .
                              .
                              .
001106 000000                 CHAR: .WORD      0
```

Figure 6-11 Instruction Execution Example

Immediately before the processor begins the machine state sequence for this instruction, the program counter (PC) contains the value 1000_8, the processor status word contains the value 000340, there are no bus requests or other asynchronous conditions, and the processor is about to enter the FET.0 machine state. In this state, a DATI bus operation is begun, using the contents of the PC as the address.

The next machine state entered is FET.1. In this state, the PC is updated (the new value is loaded into the PCA and does not disturb the PCB, which is still being used for the address in the data transfer) and the word that is read is loaded into the IR and BR. The PCA now contains 1002, the IR and BR contain 022767, and the PCB still contains 1000; finally, after the bus operation is completed, the PCB is updated to 1002.

The third machine state entered is IRD.0. In this state, the fork A logic is enabled. According to Figure 6-10, the binary number in the IR represents a CMP instruction; the entry for this instruction in Table 6-4 refers to Table 6-5, which indicates that for a source mode of 2 (as specified by the third octal digit of the instruction), the next machine state is S13.0. The IRD.0 machine state also loads the SR and DR with the updated PC value (1002), and begins a new DATI operation with the updated PC (1002) as the address.

In the S13.0 machine state, the second DATI is restarted, using the contents of the SR as the address (in this case the address does not change from the IRD.0 state to the S13.0 state, but when the source register is not register 7, the address generally changes). The contents of the SR (1002) are incremented by 2, and this value is written back into the PCA and PCB, which now contain 1004.

The fifth machine state entered for this instruction is the S13.1 state. In this state, the second DATI is completed, with the data that has been read loaded into the BR register. The new contents of the BR are 15 (the contents of the word following the instruction). The DR is loaded with the updated contents of the register specified by the destination field of the instruction (because this is register 7, the DR is loaded from the PCB); the new contents of the DR is 1004.

For a source mode of 2, the branch condition in S13.1 enables the fork C logic. The entry for the CMP instruction in Table 6-4 refers to Table 6-7, which indicates that, for a destination mode of 6 (as specified by the fifth octal digit of the instruction), the next machine state is the D67.8 machine state, which is shown on drawing D-FD-KB11-A-03 (Flows 6). This machine state transfers the BR to the SR, and begins the third DATI bus operation, using the contents of the PCB as the address.

The next machine state is the D67.0 state, which completes the third DATI and increments the PCA by 2. Because the DR is intended to reflect the current contents of the specified register, the DR must also be updated to reflect the new value in the PC, which is 1006. The data read into the BR is 100.

Following the D67.0 state, the processor enters the D67.1 state, where the PCB is loaded from the PCA and the contents of the BR is added to the contents of the DR. The result (1106) is loaded into the DR. The branch condition in this machine state selects the D10.3 state to follow the D67.1 state.

In the D10.3 machine state, the processor begins a fourth bus operation, using the contents of the DR (1106) as the address. The type of bus operation performed depends on the instruction class; according to Table 6-4. a CMP instruction is an I Class instruction, so a DATI operation is begun. This machine state also loads the BR from the SR, so that both registers contain a 15.

Following the D10.3 state, the next state entered depends on the instruction class. A CMP instruction is not F, J, or O Class, so the D10.6 state is entered. This state completes the fourth DATI operation, loading the contents of the location addressed by the DR (location 1106) into the BR (which now contains all 0s). The old contents of the BR are selected by the data paths and can be displayed in the console data lights by selecting the data paths input to the lights.

The D10.6 machine state branch condition enables the fork B logic. The entry for a CMP instruction in Table 6-1 indicates that the next machine state is TST.0 (assuming that the KB11-A processor is operating with core, rather than solid state memory). This machine state is illustrated on drawing D-FD-KB11-A-03 (Flows 11).

The CMP instruction does not alter any data words, so no further bus operations are required. The TST.0 machine state maintains the state of the bus address and control lines, to prevent deskewing problems, and performs instruction-dependent data operations and condition-code loading. Specifically, the BR is subtracted from the SR and the condition codes are loaded as in a MOV instruction on the output of the SHFR. A BRQ STROBE is performed so that the branch in the next machine state (which is FET.0) can be performed. The final values of the condition codes are as follows: the N bit is cleared, the Z bit is cleared, the V bit is cleared, and the C bit is cleared.

CHAPTER 7
LOGIC DESCRIPTION

This chapter describes the KB11-A logic in sufficient detail to allow maintenance personnel to review the purpose and function of the logic shown on each sheet of the block schematics. The text makes maximum reference to the block schematics and is organized on a sheet-for-sheet basis with those drawings wherever possible. Short-form drawing references are used throughout the text. For example, "drawing DAPA" refers to the first, or A, sheet of the DAP module block schematic. This is the same convention used in the drawings to indicate the source of a signal as part of its mnemonic. Thus, the signal RACA UBRK H is generated as shown on sheet A of the RAC module block schematic.

7.1 DAP MODULE M8100

The Data Paths (DAP) Module M8100 contains most of the data paths logic elements, including the bus register; A, B, and bus address multiplexers; the ALU; and the shifter.

7.1.1 Bus Register

Drawing DAPA illustrates the bus register (BR). The BR is implemented by three 6-bit data latches that receive data from the bus register multiplexer (BRMX). Only 16 bits of data are stored. The complement of bit 14 is also stored in the BR for use on the RAC module as a microbranch condition. DAPA BR ⟨03:00⟩ H are also brought to module pins; these signals are used to directly load the processor condition codes (on drawing IRCH) from bus data.

The register is loaded on the low-to-high transition of the clock input. The signal RACA UBRK H enables the clock input. The register is then clocked by either clock pulse TIGC T1 L for data path control, or by UBCB BUS LOAD L for a DATI bus long pause operation.

7.1.2 A, B, and Bus Address Multiplexers

Refer to drawings DAPB, DAPC, and DAPD. Each multiplexer selects one of four inputs, depending on the state of a pair of microprogram bits. The six least-significant bits of each multiplexer are shown on DAPB; the next six most-significant bits are shown on DAPC, and the four most-significant bits are shown on DAPD. The relationship between the AMX, BMX, and BAX microprogram bit values and the input selected is shown on DAPB for each multiplexer.

The A multiplexer (AMX) selects the A inputs to the arithmetic and logic unit (ALU). The source register (SR) and destination register (DR) are used to buffer the outputs of the processor general register and for temporary storage of operands fetched from bus locations.

The B multiplexer (BMX) selects the B inputs to the ALU. The constant multiplexers (K1MX and K0MX) supply small address increments, vector addresses for internal traps, and calculated offsets for instructions that make relative changes to the PC. The K0MX generates only positive values up to 10. Negative values are generated by using the ALU in subtract mode, so only the four least-significant bits are implemented; the eight most-significant bits are generated by sign-extension logic for sign extension when moving a byte (MOVB) to a general register. The K1MX generates only even numbers, so K1MX00 is always 0.

Several bits from the AMX and several bits from the BMX are brought to module pins. These signals are connected to the IRC module for use in generating processor condition codes.

The bus address multiplexer (BAMX) selects the source of the 16-bit virtual address transmitted by the processor during data transfers.

Refer to drawing DAPC. The BMX has separate selection signals for the high byte and the low byte. The purpose of this division is discussed in the description accompanying drawing DAPD.

The K1MX input to BMX11 and BMX08 is inhibited when the start vector input to the K1MX is selected. The start vector may take on either values of less than 200_8 or values between 773200 and 773374 inclusive; the higher set of values is generated by providing sign extension for the start vector and blocking bit 11 and bit 08 to generate 3200 instead of 7600.

Drawing DAPD illustrates the four most-significant bits of each of three multiplexers: the A multiplexer (AMX), the B multiplexer (BMX), and the bus address multiplexer (BAMX).

The operation of the BMX is complicated by the requirement for separate control of each byte. The microprogram bits that control the BMX are passed directly to the low-byte multiplexers, but are inhibited from the high-byte multiplexers by GRAA SGNEX MOVB L. This signal indicates that a single byte of data is being transferred to a processor general register; the sign is extended using the high byte of the K0MX input, which is selected when DAPD BMXS1 HIB L and DAPD BMXS0 HIB L are not asserted.

Drawing DAPD also shows the logic used in the constants multiplexer K0MX. The relationship between the KMX microprogram control bits and the input selected is shown on the drawing. The source and destination constants are generated on the IRC module. The remaining constants (values of 1 or 2) are generated by directly wiring a logic 1 to the appropriate bit positions. The value 2 constant can be inhibited when doing floating-point bus operations and the FRMJ FP ADDR INC L signal is not asserted. The K0MX outputs are brought to module pins because the value output by the K0MX during register modification (in address calculations) is sent to the memory management status register 1 (drawing SSRF in the KT11-C engineering drawing set).

DAPD K0EX H acts as a sign-extend signal for the K0MX input to the BMX. This sign extension is not used with the K0MX itself, but with the bus register or source register input when a byte is being sign-extended during a MOVB to a register.

If the memory management unit is not enabled, DAPD EX MEM FLAG H is used to extend the 16-bit virtual address to an 18-bit physical address. If DAPD BAMX ⟨15:13⟩ H are all 1s, the two most-significant bits of the 18-bit physical address SAPJ PA ⟨17:16⟩ H are also set to 1. Otherwise, the two bits are set to 0. When the memory management unit is active, only the 16-bit virtual address is used.

7.1.3 Constant Multiplexer 1 (K1MX)

Refer to drawing DAPE. The constant multiplexer 1 (K1MX) generates vector addresses and calculates program counter offsets. The multiplexer is controlled by the two KMX microprogram bits as shown on the drawing.

The start vector is used to fetch a new program counter (PC) and processor status (PS) during the power-up sequence. The value of the start vector (SV) is selected by jumpers on the module, as shown in the upper-left section of drawing DAPE. A jumper in place generates a 1.

Start vectors (and trap vectors) always begin on even word boundaries (that is, with address bits 01 and 00 both 0). DAPE SV ⟨06:02⟩ select a vector address within a range, while SV07 is used to select either the range from 0 to 174_8 or the range from 773200 to 773374. The range selection is accomplished by sign-extending the start vector to all bits except bit 11 and bit 08 (drawing DAPC).

The trap vector (TV) is used to select a new PC and PS following a trap operation. The trap vectors for a variety of internal conditions are defined by the logic in the lower-left corner of the drawing. The chart on DAPE defines the specific vector for each condition. If none of these conditions is present, but the processor is doing a trap operation, the trap vector is set to 4. This occurs for non-existent memory references, memory parity errors, odd address errors, fatal stack violation errors, and executing the Halt instruction in user or supervisor modes of operation. The K1MX constants for EMT and TRAP instructions are one-half their assigned values. This is because they are executed by the same machine states (Flows 12) that cause reserved instructions to be left shifted (so that vector 4 forms vector 10).

The third input to K1MX, BR ⟨07:00⟩ H, is used for the offset in subtract 1 and branch (SOB), and MARK instructions. This offset is always in full words and is always a positive quantity that is subtracted from the PC in the ALU. Because all PDP-11 Systems use byte addresses, the offset, as it appears in the instruction, must be multiplied by 2 to generate the proper value to be subtracted from the PC. This is done by shifting the 6-bit offset 1 bit to the left. For example, bit BR00 is the input to the multiplexer for bit 01. The BR is used because it contains the same value as the instruction register (IR) at the time of the PC modification, and is directly accessible to the data path logic.

The fourth input to K1MX is used for the offset in successful branch instructions. The branch offset can be either positive or negative; the value taken from the instruction is first multiplied by 2 (shifted left) and then sign-extended, and the resulting 16-bit number is added to the PC. The branch offset can have values from $+127_{10}$ to -128_{10} words; BR ⟨06:00⟩ provides 2^7 or 128 numbers, and the left shift provides word (rather than byte) addresses.

7.1.4 Arithmetic Logic Unit, Shifter, and Program Counter

7.1.4.1 Arithmetic Logic Unit (ALU) – Refer to drawings DAPF and DAPH. The ALU does most of the data manipulation in the processor. It operates on two 16-bit words of data and a carry input to produce one 16-bit word of data and a carry output. The carry signals are not active when the ALU is operating in the logical mode. Drawing DAPF shows the low byte and DAPH shows the high byte.

The 16-bit ALU is implemented with four 74S181 4-bit Arithmetic Logic Units. Each 74S181 includes look-ahead carry generation for the four bits. A second level of look-ahead carry generation is provided by the 74182-1 carry generator. The carry-propagate (P) and carry-generate (G) outputs of each 74S181 (except the most-significant four bits) are connected to the corresponding inputs of the 74182-1, and the carry outputs of the 74182-1 are connected to the appropriate carry inputs of the ALUs. The least-significant bit carry input is controlled by GRAA ALUC H, based on the output of the subsidiary instruction-dependent ALU control ROM.

The ALU can perform any one of 16 logical functions (each output bit is dependent only on the corresponding input bits) or any one of 16 arithmetic functions (each output is dependent on the corresponding input bits and on a carry propagated from less-significant bits). The selection of a particular function is controlled by five signals from the GRA module which select the mode (arithmetic or logical) and the function. The functions used in the KB11-A are charted on drawing DAPF. The complete 74S181 truth table is listed in Appendix A of the *PDP-11/45 System Maintenance Manual.*

The function select inputs, S⟨3:0⟩, of the ALUs require three unit loads each. Therefore, the function and mode select signals from the GRA module drive two sets of inverters. One set supplies control signals to the low-byte ALUs, while the second set supplies control signals to the high-byte ALUs. This reduces the fanout requirements on each inverter to an acceptable level; e.g., DAPF LS3 H and DAPH HS3 H are logically identical.

In addition to the data and carry outputs, each ALU element has a comparator output which indicates (if the ALU is in subtract mode) that the two inputs are equal. These outputs, which are open-collector driven, are wire-ANDed for each data byte to generate equality signals that are used in forming the condition codes.

DAPF A = B ⟨7:0⟩ H indicates that the inputs to the low data byte are equal.

DAPF A = B ⟨15:0⟩ L indicates that the inputs to the entire word are equal. DAPH BUS A = B ⟨15:8⟩ H is the wired-AND of the A = B outputs for the high-byte ALUs on drawing DAPH.

7.1.4.2 Shifters and Program Counter – The output of the ALUs are routed to the shifters (SHFR) and to the program counter (PC). The program counter is implemented as a double-buffered register, to permit the contents of the PC to be changed while the previous contents are being used as the address in a data transfer. The double buffering requires two registers, PCA and PCB. The PCA register is loaded directly from the ALU under control of a clocking signal that is enabled by a microprogram bit. The outputs of the PCA go only to the PCB, which has a separate clocking signal under separate microprogram control.

The shifter (SHFR) is a 4-input multiplexer that provides unshifted, right-shifted, and byte-swapped inputs from the ALU, and can also accept the contents of the PCB as an input. The output of the SHFR is directed to the general registers, the source and destination registers, and the bus address of bus data lines. The shifter output provides the console data display referred to as DATA PATHS. Left shift operations are performed in the ALU by using the A plus A mode. The sum of A added to A is equivalent to the product 2A, which in turn is equivalent to shifting A (as a binary number) one bit to the left.

Special operations are required in the shifter for the most-significant bit of each byte. The shifter logic for data bits 7 and 15 are shown separately on drawing DAPJ.

7.1.4.3 Shifter Logic – Refer to drawing DAPJ. The most significant bit of the shifter is SHFR 15. The shifter inputs are similar to the inputs for other shifter bits when the byte-swap or unshifted ALU inputs are selected. However, the input used when the right shift mode is selected is dependent on the instruction being executed. Normally, on a right shift operation, the sign of the data word is extended. This is done by routing ALU15 (the most-significant, or sign, bit) to the right shift inputs of both DAPJ SHFR 15 and DAPH SHFR 14. For right rotate (ROR and RORB) instructions and multiply instructions, this procedure is modified by forcing a second level 2-input 74S157 multiplexer to select GRAJ SHFR DATA H instead of DAPH PCB15 H. The signal GRAJ SHFR DATA consists of the carry (C) bit for the rotate instruction; for the multiply instruction, the input is used to extend the sign of the result during the calculation and to correct the sign on the cycle if necessary.

The shifter logic for data bit 7 must operate the same as the normal bits for word data, and as the most-significant bit for byte data. The right shift input must be able to receive one of three values; ALU08 for word data; ALU07 for byte shifts, if not a rotate instruction; or the carry (C) bit for an RORB instruction. This is accomplished by multiplexing the C bit with the PCB input and forcing the SHFR to accept input B for an RORB instruction; for any other byte shift, the SHFR is forced to accept input C, the no shift input, so the SHFR07 and SHFRA07 both receive ALU07. SHFRA15 and SHFR15 signals and SHFRA07 and SHFR07 signals are logically identical and appear only for additional loading capacity.

7.1.4.4 Program Counter Clocks – Refer to drawing DAPJ. The two PC registers are clocked separately. The PCA register is clocked when pulse TIGD T5 H, enabled by the microprogram bit RACA UPCA H, produces DAPJ CLKPCA H.

DAPJ CLKPCB H is controlled by two bits of the microprogram word. In addition to directly enabling the clock signal on the pulse TIGC T1 H, the control bits can selectively enable clocking of the PCB, but only if the register selected by an instruction is register 7, the PC. To determine which register is selected by the instruction, IRCB SRCF7 H and IRCC DSTF7 H are generated if the corresponding register-select bits in the instruction register are set. When the processor updates the contents of a register during an address calculation, the updated contents are clocked into the PCA register. If the selected register is register 7, the updated contents can be clocked into PCB on the next cycle. The PCA register is clocked on time pulse T5 and the PCB register is clocked on the following time pulse T1.

7.1.4.5 Control Signals – Refer to drawing DAPJ. The remaining logic on this drawing generates four signals that are used in generating the processor condition codes and one signal that is used in 2-word left shifts:

a. DAPJ AMX SIGN H is the sign of the A input to the ALU. This signal corresponds to AMX15 if the processor is operating on word data, or to AMX07 if the processor is operating on byte data.

b. DAPJ ALU SIGN H is the sign of the ALU output; it is taken from ALU15 for word data or from ALU07 for byte data.

c. DAPJ A=B ⟨15:8⟩ + BYTE H indicates either that the high data byte is all 0s or that the processor is operating on byte data. This signal is used in determining whether all the active data is 0s for the Z condition code.

d. DAPJ ALUCN L is the carry output of the active portion of the ALU; it takes the carry output from the high byte for word data or the carry output from the low byte for byte data. This signal is used to generate the carry (C) condition code.

e. DAPJ LEFT DATA H is the input to the least-significant bit of the destination register (DR) during left shifts that shift both the ALU data and the destination register. Normally, this input is 0 for left shifts, but during the execution of the divide instruction (DIV), the input receives the carry output of the ALU.

7.2 GRA MODULE M8101

The General Registers and ALU Control (GRA) Module M8101 contains the general register and ALU control logic. The SR, DR, and SC registers are also included on this module.

7.2.1 Arithmetic and Logic Unit Control

Refer to drawing GRAA. Data manipulation is performed in the KB11-A processor by an arithmetic and logic unit (ALU) that can combine two operands in various ways or perform operations on either operand singly. During each machine cycle, the operation performed by the ALU (and the operation performed on the ALU output by the shifter) is selected by a set of control signals from the GRA module. The logic that generates these control signals is shown on this drawing.

During most machine cycles, the ALU and the shifter (SHFR) are controlled directly by the bits in the ALU and SHF fields of the main ROM microprogram word. However, if the value of the ALU control field is 7, the control signals are generated from the outputs of the subsidiary ALU control ROM, located on the GRA module. This feature is called the instruction-dependent control of the ALU.

There are eight control signals generated on this drawing (as well as several signals used to generate other data path control signals). These eight signals include:

a. GRAA ALU ⟨S3:S0⟩ L
b. GRAA ALUC H
c. GRAA ALUM L
d. GRAA SHFRS ⟨S1:S0⟩ L

GRAA ALU INS DEP L controls two 74S158 multiplexers that select the source of these ALU control signals. When the signal is high, the main ROM ALU and SHF fields are the source. If the ALU field is 7, GRAA ALU INS DEP L selects the subsidiary ROM on the GRA module.

7.2.1.1 **Non-Instruction-Dependent Control** – The ALU control field in the main microprogram ROM is a 3-bit field that controls the values of six control signals. There is not a one-to-one relationship between the ROM bits and the control signals, and not all possible combinations of control signals can be generated. Each control signal is, in general, the result of decoding the ROM bits and sensing selected inputs from the condition codes and the instruction decoding. Table 7-1 shows, for each value of the ALU control field, the operation performed by the ALU and the states of the ALU control signals necessary to select that operation. The numbers at the bottom of the columns indicate which ALU control field values generate each signal. The logic connected to the non-instruction-dependent inputs of the multiplexer generates the signals for the values shown.

Table 7-1
Non-Instruction-Dependent ALU Control Signals

UALU	Operation	S3	S2	S1	S0	M	Cin
0	not A	L	L	L	L	H	
1	B	H	L	H	L	H	
2	A (plus carry)	L	L	L	L	L	L
3	A plus B (plus carry)	H	L	L	H	L	L
4	not used						
5	A plus A (plus carry)	H	H	L	L	L	L
6	A - B	L	H	H	L	L	H
7	instruction-dependent						

7.2.1.2 **Instruction-Dependent Control** – When the ALU control signals are instruction-dependent, each of the six signals is controlled by a separate output signal from the subsidiary ALU control ROM shown on drawing GRAA. The two signals, ALUS0 and ALUS1, unconditionally take on the value of the ROM outputs. The other two select signals, ALUS2 and ALUS3, are blocked when the SWAB instruction is being executed. The SWAB instruction does not have a unique ROM word, and uses the same word as the ASL instruction with some of the control signals modified in this manner. The ALU control ROM map is shown on drawing GRAK.

The ALUM (mode control) signal is taken directly from the ROM except when the SXT instruction is executed with the sign bit set. The value stored in the subsidiary ROM for the SXT instruction causes the ALU to generate a logic 0. When the mode bit is forced off, the ALU generates an arithmetic minus 1. The mode bit is also forced off (arithmetic mode) for the ROL, ADC, and SBC instructions, to force the ALU into the A plus A mode. The combination of both ROMM and ROMC asserted is used to indicate that special treatment of the carry bit is necessary.

The generation of the ALU C (carry-in) signal is modified for two classes of instructions. The DIV and ASHC instructions operate on 2-word operands, and the instruction-dependent state is one that shifts the two words left. The carry-in must take on the state of the most-significant bit of the less-significant word. For the ADC and ROL instructions, a carry insert signal is generated if the C bit is set; for the SBC instruction, the signal is generated if the C bit is cleared. This data-dependent carry generation is controlled by the assertion of both ROMM and ROMC, as described in the previous paragraph.

GRAA SGNEX MOVB is generated when a MOVB instruction is being executed. This instruction is used to extend the sign of the byte into the high byte when the destination is a general register.

GRAA WORD + OB SWAP L and H indicate that the significant SHFR outputs include the high byte, and the sign of the output is bit 15 (rather than bit 7).

The SHFR control signals, GRAA SHFRS0 and SHFRS1, are normally derived directly from the main microprogram ROM. When the ALU control signals are instruction-dependent, these signals are also instruction-dependent and are taken directly from the subsidiary ALU control ROM. For the SWAB instruction, and for certain instructions that require a byte swap during execution, these signals are forced low to generate the swap-byte control of the SHFR.

7.2.2 Shifter Zero Detection

Refer to drawing GRAB. The GRAB Z DATA2 L logic on this drawing detects all 0s data at the shifter (SHFR) outputs. Depending on the operation being performed, either the entire word of data or only one byte of data may be significant. For word data, the two wire-ANDed circuits must both detect all 0s. For normal byte operations, only the low byte (SHFR07 through SHFR00) must be all 0s. During operations on odd bytes or during a SWAB instruction, only the high byte is tested. A fourth input, enabled by IRCF CHECKZ H, is used when the final result is two words to clear the 0 (Z) bit if the second word does not contain all 0s. If the second word is all 0s, the Z bit retains the previous value. Thus, only if both words are all 0s will the Z bit be set.

7.2.2.1 **Left Save** – GRAB LEFT SAVE (1) H and its complement are used during the divide instruction to determine, after each subtraction cycle, whether the next cycle should also subtract. The signal DAPJ LEFT DATA H is the carry-out of the ALU.

7.2.2.2 **Odd Byte Destination** – GRAB OBD (1) H and its complement are used to indicate that the destination address in the DR register points to an odd byte. This flip-flop is clocked at the same time as the DR register and receives the same input when the DR is in the load mode.

7.2.3 General Register Address Logic

Refer to drawing GRAC. The logic shown on drawing GRAC generates signals that control the selection of one of 16 general registers in each of two scratchpad memories in the KB11-A processor. The processor has two sets of eight registers, from a programmer's point of view, but each set is duplicated so that two registers can be read at one time. When data is written into a register, both sets must access the same register; however, there is no logical protection against addressing different registers during writing. The microprogram is responsible for selecting an input to the register address generators that generates two identical addresses.

7.2.3.1 **Source and Destination Address Multiplexers** – The microprogram selects the sources of the scratchpad addresses. The microprogram includes a 3-bit PAD field that selects one of seven sets of sources; the value of 3 in the PAD field is not used. Some of the sources are constants, and are generated by +3V and 0V inputs to the GDAM and GSAM multiplexers; others are taken from the IR source and destination register specifications of the instruction. The chart on drawing GRAC illustrates the source selected for each value in the microprogram PAD field.

As indicated by these charts, the multiplexer control is required to gate seven sources through a 4-input multiplexer. For four cases, the GDAM and GSAM multiplexers operate alike:

a. For a microprogram PAD field value of 0, both multiplexers select the register specified by the source field, using the A inputs.

b. For a value of 2, both multiplexers address register 5, using the C inputs.

c. For a value of 5, the address is taken from the destination field, using the B inputs.

d. For a value of 7, the D inputs generate an address of 6.

If the microprogram PAD field contains a value of 6, all the multiplexers are disabled. As a result, the address of register 0 is selected.

For the two remaining PAD field values, 1 and 4, the operation of the multiplexers is altered. If the PAD field value is 1, destination register address multiplexer GDAM uses input B, but source register address multiplexer GSAM uses input A. This is done by forcing the S0 input of the source multiplexer to a 0. If the PAD field value is 4, GRAC PLUS 1 L is generated. This signal is ORed with the least-significant bit of the source field from the IR to force an odd register address. This is used in the MUL, DIV, and ASHC instructions. GRAC PLUS 1 L is generated for other control field values, but these values select other multiplexer inputs and are not affected by this signal.

7.2.3.2 **General Register Set Selection** – The most-significant bit of the scratchpad address selects which general register set is used. This selection is, in general, done by the multiplexer, but in several special cases the processor forces the selection of general register set 0, which includes the kernel mode stack pointer.

Part of the general register set 1 selection logic is shown on drawing GRAB. General register set 1 can be selected as the source set by bit 11 of the processor status word if R0 through R5 is specified. These same requirements apply to selection of general register set 1 as the destination set. If the general register selection bit in the PSW is set, then general register set 1 is used for both source and destination when general registers R0 through R5 are specified as source or destination. In the user and supervisor modes, specifying R6 as the source or destination will select general register set 1. This forms general register address 16 in supervisor mode. If the processor is in user mode, bit 0 is forced to create general register address 17. If general register 6 is specified in a Move From Previous Space (MFP) instruction, the register address used is determined by the previous processor mode, as indicated by bits 13 and 12 of the PS.

NOTE

In an MFP instruction, the source is always specified in the field normally designated as destination. The destination is the current mode stack.

For the address sources that are variable (the source and destination fields), the register set is selected by a corresponding variable signal. That is, when the register address is taken from the source field, GRAB SRC SET 1 L selects the register set, and when the address is taken from the destination field, GRAB DST SET 1 L selects the register set. For the constant input 6, the register set is selected by current processor mode, to select the correct stack pointer; for the constant input 5, the register set is selected by PS11, which indicates which general register set is in use. (This input is used for the MARK instruction.) These inputs are forced to 0 during the console operations, register examine and register deposit, so that the console operations can explicitly select the desired register set.

When console operations that access the general registers are performed (REG DEP, REG EXAM), the register is selected by the four least-significant bits of the switch register. The switch register is loaded into the IR, so that the destination field inputs to the address multiplexers (which are taken from IR ⟨02:00⟩) can be used to select the register within a set. The set is selected by gating IR03 directly to the most-significant bit of the pad address.

7.2.3.3 **General Register Control Signals** – Refer to drawing GRAC. GRAC GD6 L is used to indicate that the processor is using the stack pointer; this signal qualifies the stack limit logic.

GRAC T6 L and GRAC GATE T6 H are used to clock data after the end of a machine cycle. Because the processor timing cycle can stop for bus operations, condition code clocking must be done by this signal to avoid losing the new condition code values before the processor is restarted.

GRAC GRWE HIB L and GRAC GRWE LOB L are the write-enable signals for the high and low bytes of the general registers, respectively. These signals are generated directly from the PAD write-enable bits in the main microprogram word. A conditional write operation is done only if the conditions are met. An unconditional write operation is done unless inhibited by the memory management unit, which inhibits changes to processor data if it aborts an instruction. The conditions for generating each write-enable signal are shown by charts on drawing GRAC. Writing into the general registers is done at the end of a machine cycle (indicated by the T5 pulse). The write-enable signal is latched to provide sufficient time for the write operation to take effect, and the latches are then cleared by the T6 pulse.

GRAC GRA0 L through GRAC GRA3 L are sent to the KT11-C Memory Management Unit to indicate which registers have been altered during the execution of an instruction. This information is stored in memory management status register SR1 and can be used by the processor to recover from an instruction abort.

7.2.4 General Registers, Source and Destination Multiplexers, and Registers

Refer to drawings GRAD, GRAE, GRAF, and GRAH.

7.2.4.1 General Registers – The processor uses two copies of the two sets of general registers. These two copies are provided to enable the processor to read two different general registers simultaneously. This is done when the processor reads the two registers specified by the source and destination fields in an instruction. The two copies of the general registers are therefore called the general source (GS) and the general destination (GD) registers. The register sets operate separately only for reading; when data is written into a register, it is written into both the GS and the GD register simultaneously.

The data input to the general registers is the output of the shifter (SHFR). The SHFR outputs are also brought directly to the source and destination multiplexers.

The general registers are implemented in two sets of four 64-bit random access memories that are arranged in sixteen 4-bit words. Each general register is made up of one word from each of four memories, so that the same word selection signals are sent to all four memories for one copy of the registers. A different set of selection signals can be sent to the second copy of the registers while reading, but this must not occur when data is being written.

7.2.4.2 Source and Destination Multiplexers – The data input to the general registers is the output of the shifter (SHFR). The SHFR outputs are also brought directly to the source and destination multiplexers, which can select either the SHFR data or the general register output data. Because the register memories output complemented data, the SHFR data is inverted before going to the multiplexers, which invert the data to return it to normal polarity.

7.2.4.3 Source Register (SR) – The outputs of each multiplexer are connected directly to the corresponding register. The source register (SR) is clocked on the pulse TIGC T1 H if enabled by the microprogram bit RACA USRK H. The outputs of the SR are routed to the ALU input multiplexers and to the bus address multiplexer. Bit 0 of the SR is also sent to the IRC module for use in one of the microprogram address generation circuits, the fork C, for odd-byte source branches.

7.2.4.4 Destination Register (DR) – The destination register (DR) can be loaded with a left shift of one bit, a right shift of one bit, or no shift. The shift inputs are used when the processor must operate on two words of data at the same time (for example, during a multiply or divide instruction) and the operation includes shifting. The DR is implemented with 4-bit registers that have six input signals. Each bit of storage can be loaded from

one of three of the six signals; the three inputs for each bit overlap with the three inputs for the next bit. Which input is loaded into the DR is selected by the signals RACA UDRK00 H and RACA UDRK01 H from the microprogram DRK field.

The outputs of the six low-order bits of the destination register, DR ⟨05:00⟩, are routed to the shift counter input The shift counter is used in multiple step instructions, including multiply, divide, and arithmetic shift (ASH or ASHC) to count the number of steps that are done; for the arithmetic shift instructions, the desired shift count is loaded from the six low-order bits of the destination word and the shift is performed on registers specified by the instruction.

7.2.4.5 **Control Logic** – The source multiplexer (SRMX) and destination multiplexer (DRMX) are controlled by GRAC SRMX SEL and GRAC DRMX SEL. Each signal is generated by a logic circuit controlled by two microprogram bits which combine with the register selection field of the current instruction to select either the SHFR output or the general register output. When the instruction selects register 7 (the PC), the multiplexer selects the SHFR input because the data is read from the PCB register through the SHFR, not from the general registers. In addition, the DRMX control bits can also select an all 0s input to the DR. This is implemented by directly clearing the DR. The signal GRAD CLRDR L is generated during the period between time pulses 3 and 5, if multiplexer control signals RACA UDRX00 H and RACA UDRX01 H are asserted.

7.2.4.6 **Special Signals** – Refer to drawing GRAE. GRAE SR EQ ONE L is asserted if the value in the SR is a positive 1 (0 000 000 000 000 001). The two flip-flops shown check for all 0s in data bits 1 through 15. The outputs of the general registers are inverted, so a low signal represents a 1; any 1 clears the flip-flop because the input is inverted. The flip-flops are clocked by the same signal that clocks the SR. The 1 bit in SR00 is taken from the SR output and ANDed with the flip-flop outputs. This signal is routed to the RAC module for use in the microprogram branch control. GRAE SR LEQ ZERO H is generated if the contents of the SR are either all 0s (SR00 does not contain a 1 and the two flip-flops are both 1s) or negative (SR15, the sign bit, is a 1).

7.2.4.7 **SR15 and DR15** – Refer to drawing GRAH. Both SR15 and DR15 are available in normal and complemented form. GRAH SR15 L is used on drawing GRAE to generate GRAE SR LEQ ZERO H, and is used on drawing GRAJ to generate SHFR DATA for the multiply instruction and to control the operation of the divide instruction. This signal is also routed to the RAC module as one of the microprogram branch conditions.

GRAH DR15 is routed to the IRC module for use in generating the carry (C) condition code during a multiply instruction.

7.2.5 Shift Counter

Refer to drawing GRAJ. The shift counter (SC) is used to count the repetitive cycles of data manipulation in the multiply (MUL), divide (DIV), arithmetic shift (ASH), and arithmetic shift combined (ASHC) instructions. The SC is used with the microprogram branch facility. When the branch-enable bits of the current microprogram word select the SC sensing signals, the next microprogram word selected is a function of the present SC contents. SC loading and counting is under direct control of the SHC bits in the current microprogram word. The SC can be loaded with a value from the six least-significant bits of the DR (for ASH or ASHC instructions) or with a constant 17_8 (for MUL or DIV instructions).

The actual value loaded into the counter is the 1's complement of the selected input. The selection of inputs is done by a wired-OR for the four least-significant bits (note that the OR forces these inputs to all 0s, not all 1s), while the four most-significant bits are blocked (to force all 1s) for the constant input. When the variable input is selected, the inputs to the five least-significant bits are inverted; the three most-significant bits are an extended sign.

The direction of counting is dependent on the current sign of the SC. If the SC has been loaded with a positive value (because the SC always contains the complement of the desired number, the sign is negative), the SC counts up to bring the complement closer to 0.

The test for SC=0 tests SC05 directly, and can only be true if SC05=1. The MAX/MIN output of the lower four bits of the SC is dependent on the UP/DOWN (DN) input. When the input is low, the output is true if the lower four bits are all 1s. Therefore, the signal GRAJ SC=0 L is generated only when the contents of the SC are all 1s (the 1's complement of 0).

The SC is clocked by a signal generated from two of the central processor timing pulses and enabled by the micro program shift counter control. The clock signal is a pulse lasting from the beginning of the T3 timing pulse to the beginning of the T5 pulse. This circuit generates a long pulse that is needed in driving the counters used to implement the SC.

7.3 IRC MODULE M8102

The Instruction Register Decode and Condition Code (IRC) Module M8102 contains the instruction register and condition-code logic elements. The fork B and fork C decode logic is also included on the IRC module.

7.3.1 Instruction Register (IR)

Refer to drawing IRCA. The 16-bit IR is made up of 74S74 D-type flip-flops. Data inputs are applied on bus multiplexer lines PDRA BRMX ⟨15:00⟩ H. The IR clock, IRCA CLKIR L, is only enabled when the microprogram IRK field is logic 1. When enabled, data will be clocked into the IR at time pulse T1 for data path control or by UBCB BUS LOAD for bus long pause DATI Unibus cycles.

7.3.2 Fork B Logic

Drawing IRCB illustrates logic that decodes the contents of the IR, and logic that generates microprogram addresses for two groups of instruction types. The instruction register decoding is in two parts; several binary-to-decimal decoders that recognize specific values in IR fields, and combinational logic that converts specific field values into instruction groups. The instruction groups are given mnemonics that represent the individual instructions and classes of instructions that generate each signal. These mnemonics, and the instructions for which each signal is generated, are listed on the drawing.

7.3.2.1 Fork B Instructions – These signals are used to generate microprogram addresses through the fork B logic (drawing IRCB) and through the fork C logic, shown on drawings IRCC and IRCD. In addition, these signals are used to control the data paths through signals generated by logic shown on drawing IRCC.

The fork B logic generates microprogram addresses that are used to select the next machine state after the destination operand has been fetched. For each instruction that operates on a destination operand, there is a unique microprogram word that controls the execution of the operation for that instruction. The majority of these instructions are included in the P Class group. The P Class instructions are executed by a single microprogram word that is stored in ROM location 031, with the exception of the NEG, ASRB, and RORB instructions. The exceptions are made because these instructions can not do a byte swap during the execution cycle, and must use other machine states that permit a separate byte-swap operation for odd-byte data.

7.3.2.2 Fork B Multiplexer – The fork B addresses are generated by a 74S157 2-input, 4-bit multiplexer, and by two additional gates. IRCB B0 RAB04 L is connected to ROM address bits 4 and 5, to generate ROM addresses ranging from 60 to 67. IRCB B0 RAB03 L is connected to ROM address bits 3 and 4, to generate ROM addresses ranging from 31 to 36. The ROM addresses used by the fork B and the instructions executed by each address, are listed in Table 7-2.

Table 7-2
Fork B Instructions and Addresses

Instruction	ROM Address	Instruction	ROM Address
P Class	031	MUL	060
TST.B + BIT B + CMP B	032	DIV	061
(-FAST)		ASH	062
TST.B + BIT B + CMP B	033	ASHC	063
(FAST)		ASRB + RORB	064
JSR	034	MFU	066
JMP	035	NEG	067
F Class	036		

The fork B multiplexer operates in an inverted mode. When an input is low, the output is a 1. Thus, when the multiplexer is disabled for a NEG instruction, the outputs are all 1s; this generates address 67. For all other addresses, the inputs are selected by a signal that is generated for the MUL, DIV, ASH, ASHC, ASRB, RORB, and MFP instructions. When this signal is asserted, the B inputs of the multiplexer are used; RAB04 is forced to a logic 1 by a 0V input. Conversely, the A inputs are used for F Class, J Class, K Class, and most P Class instructions; RAB04 is forced to a 0 by a +3V input. The instructions that use the A inputs of the multiplexer also assert IRCB B0 RAB03 L.
IRCB B0 RAB ⟨02:00⟩ L are generated by connecting the instruction group signals to the multiplexer inputs in the order required for each signal. Table 7-3 lists the signals generated for each ROM address.

Table 7-3
Fork B Address Generation

Address	A Inputs	B Inputs	Other
031	RAB00		RAB03
032	RAB01		RAB03
033	RAB01		B1 RAB00, RAB03
034	RAB02		RAB03
035	RAB02		B1 RAB00, RAB03
036	RAB01, RAB02		RAB03
060		RAB04	
061		RAB00, RAB04	
062		RAB01, RAB04	
063		RAB00, RAB01, RAB04	
064		RAB02, RAB04	
065	not used	not used	
066		RAB01, RAB02, RAB04	
067	forced	forced	

7.3.3 Fork C Logic

Refer to drawing IRCC. The logic shown on this drawing decodes the address modes and register specifications of the current instruction, and generates signals that control register selection and address calculation in the processor. The logic also generates addresses for the fork C microprogram address logic. The fork C selects the address of the next microprogram address when a destination operand must be fetched.

7.3.3.1 Fork C Instruction – Two 8251-1 BCD-to-Decimal Decoders are used to recognize the source and destination modes, respectively, by decoding each 3-bit IR field. The source and destination modes determine the operations performed in the fetching of operands; these signals are used throughout the IRC module. Destination mode 0 is also used to separate the fork C addresses for this mode and all other destination modes, by connecting IRCC DSTM0 L to the fork C input for bit 7 of the ROM address (as shown on drawing RACL) and connecting IRCC DSTM0 H to the input for bit 6. In this manner, the fork C generates microprogram addresses ranging from 202 to 211 for destination mode 0, and microprogram addresses ranging from 110 to 117 for other destination modes.

The exact address generated by the fork C logic depends on:

a. the specific destination mode and the necessity for an odd byte swap, for a non-zero address mode

b. whether the instruction is a floating-point type instruction, the necessity for an odd byte swap, and, if the instruction is not a floating-point type, whether the PC is used as the register for destination mode 0

7.3.3.2 Fork C Multiplexer – The fork C multiplexer is a 74S157 4-bit 2-Line-to-1-Line Multiplexer that is controlled by IRCC DSTM0 L. Recognition of destination mode 0 generates the four low-order bits of the microprogram address for the fork C. The two high-order bits are directly controlled by the destination mode and bits 4 and 5 are always 0. Bit 3 of the address is always a 1 if the destination mode is not 0 (the input is a ground which generates a low output, which asserts the input to the microprogram address assembly logic on drawing RACL). For destination mode 0, bit 3 is controlled by the instruction class; the bit is set for F Class instructions and clear for all others. Table 7-4 summarizes the fork C multiplexer outputs.

Table 7-4
Fork C Multiplexer Outputs

Destination Mode	Fork C Multiplexer Input	Fork C ROM Address								Microprogram Address	
		DST M0 L		Constant L		C0 RAB					
		07	06	05	04	03	02	01	00	State	Loc
0	A	H	L	L	L	L	L	H	L	D07.00	202
0	A	H	L			L	L	H	H	D07.10	203
0	A	H	L			L	H	L	L	D00.80	204
0	A	H	L			L	H	L	H	D00.90	205
0	A	H	L			H	L	L	H	F0P.50	211
1, 2	B	L	H	L	L	H	L	L	L	D12.90	110
1, 2	B	L	H			H	L	L	H	D12.80	111
3	B	L	H			H	L	H	L	D30.90	112
3	B	L	H			H	L	H	H	D30.80	113
4, 5	B	L	H			H	H	L	L	D45.90	114
4, 5	B	L	H			H	H	L	H	D45.80	115
6, 7	B	L	H			H	H	H	L	D67.90	116
6, 7	B	L	H			H	H	H	H	D67.80	117

7.3.4 CCL Decoding

The condition code load (CCL) field of the ROM is decoded as shown on drawing IRCF to determine how the PSW condition code bits are to be altered by each microprogram. The CCL field is summarized in the following chart:

RACA UCCL			Output Asserted IRCF:	Function
02	01	00		
0	0	0	CC NON AFF L	No change
0	0	1	CC INSDEP H	Instruction-dependent. Condition codes determined by subsidiary CC CNTL ROM.
0	1	0	(IRCH SETCC H)*	Set or clear CC; dependent upon IR.
0	1	1	CCFP LOAD L	Load CCs from floating-point processor
1	0	0	CCLD4	Z and N: ACC SHFR C and V: 0
1	0	1	CCLD5	Z and N: ACC SHFR C: AMX15 V: V old + (AMX ∀ ALU)
1	1	0	CCLD6	N, C, and V: not affected Z: Z* SHFR = 0
1	1	1	CCLD7	Z, N, and V: not affected C: carry

* Generated on drawing IRCH.

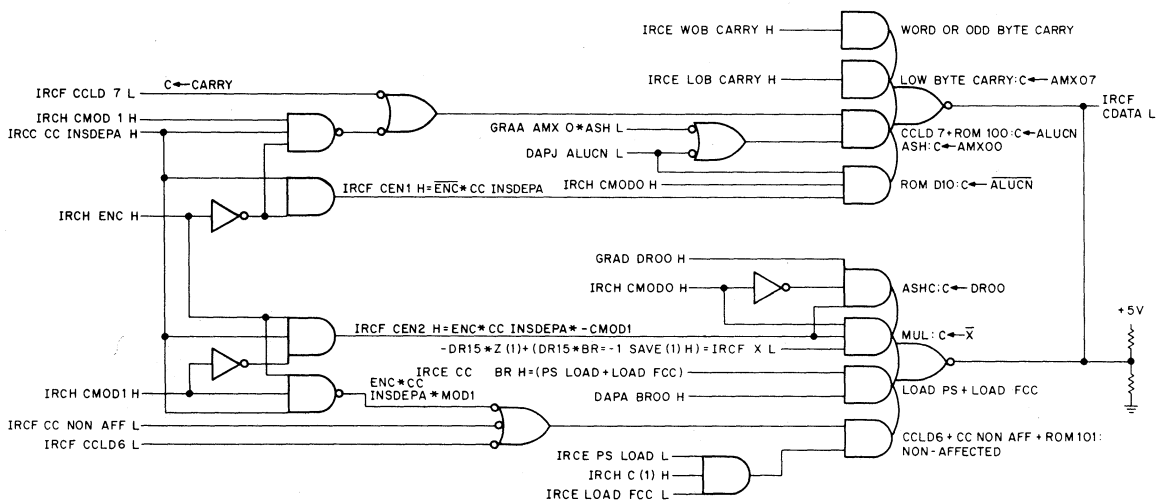


Figure 7-1 Sources of C Bit Data, Simplified Diagram

7.3.5 C Bit Data

The C (carry) bit of the PSW is set when a processor operation causes a carry out of the most-significant bit. The logic that generates the C bit data is shown on drawing IRCF. Figure 7-1 is a simplified diagram of the logic that asserts IRCF CDATA L. Each AND gate input covers a group of instructions that could cause a carry. The notation adjacent to each AND gate indicates the conditions or instructions that enable the gate and the resultant C bit source that asserts IRCF CDATA L.

Table 7-5 lists the instruction-dependent CC CNTL ROM outputs that control the C bit for each group of instructions. IRCE WOB CARRY H and IRCE LOB CARRY H are derived from a 74S153 multiplexer. These C bit inputs are determined from AMX 00, AMX 07, or AMX 15, as listed in Paragraph 7.3.8.

Table 7-5
C Bit Data Sources

Instruction	CC Control ROM			IRCF CDATA L Source
	CMOD1	CMOD0	ENC	
ROR.B, ASR.B	0	0	0	C ← AMX00 (VMOD0=1)
ROL.B, ASL.B	0	0	0	C ← AMX08 (WORD) C ← AMX08 (OB)
ASHC	0	0	1	C ← DR00
COM.B, NEG.B, SBC.B SUB	0	1	0	C ← -ALUCN
MUL	0	1	1	C ← -X
CLR.B, ADC.B TST.B CMP.B, ADD	1	0	0	C ← ALUCN
ASH	1	0	0	C ← AMX00
MFP, MTP, SXT INC.B, DEC.B MOV.B, BIT.B, BIC.B BIS.B, XOR	1	0	1	non-affected
DIV	1	1	0	C ← 1 C ← 0 if -DR15
SWAB				C ← 0
Condition-Code Load Signals				
IRCF CCLD4				C ← 0
IRCF CCLD5				C ← AMX15
IRCF CCLD6				non-affected
IRCF CCLD7				C ← ALUCN

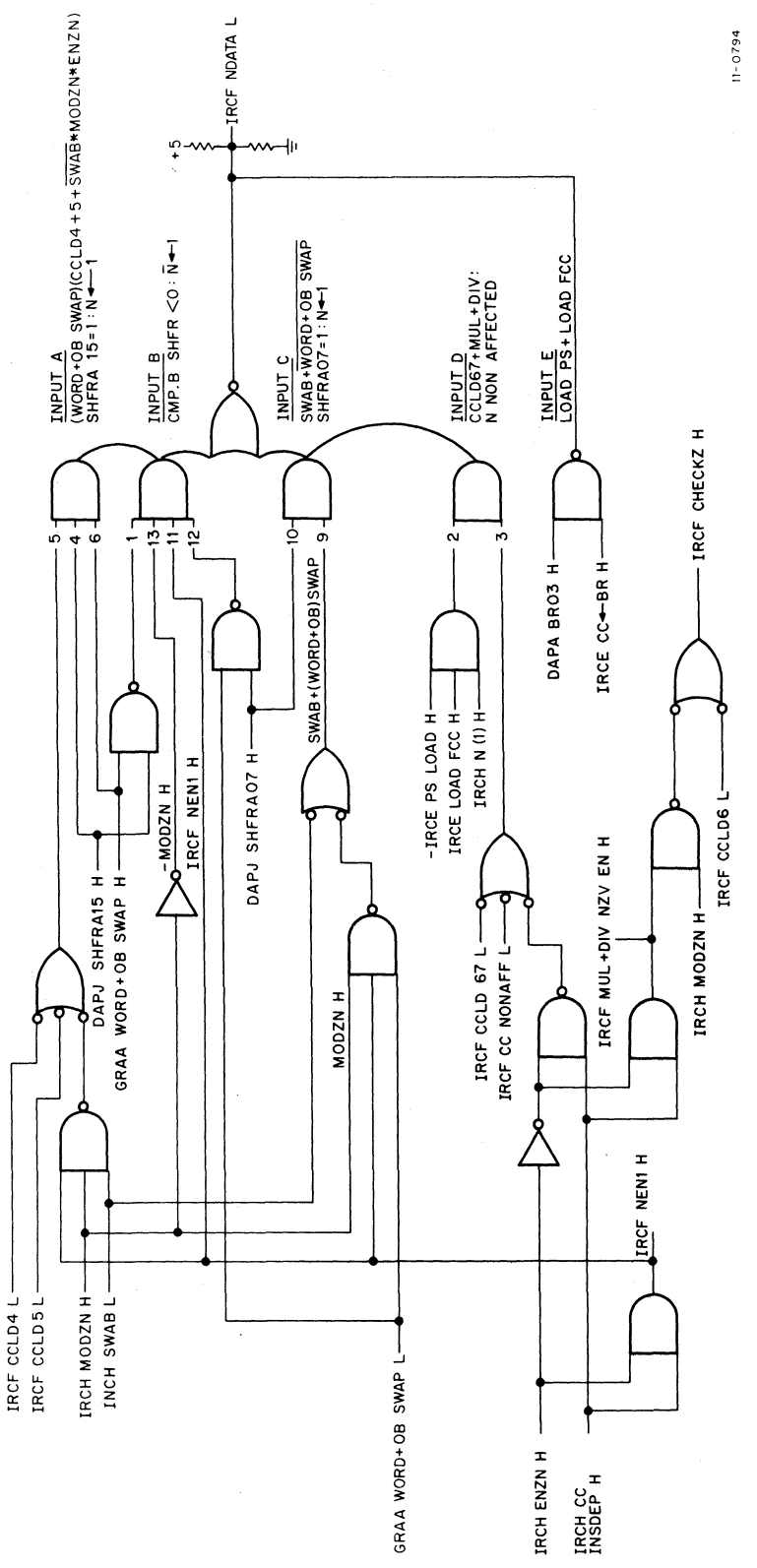


Figure 7-2 Sources of N Bit Data, Simplified Diagram

7.3.6 N Bit Data

The N (negative) bit of the PSW is set when a negative result is produced by a processor operation. The logic that generates the N bit data is shown on drawing IRCF. Figure 7-2 is a simplified diagram of the logic that asserts IRCF NDATA L. Each AND gate input decodes a particular group of instructions or processor operations for which a negative result might be obtained.

For most of the instructions, the CC CNTL ROM outputs IRCH MODZN H and IRCH ENZN H are asserted. These control outputs condition the NDATA logic to examine the SHFR output to determine when the N bit should be set. For word or odd-byte operations, the input A logic tests SHFRA15, and sets N accordingly. For byte operations, the input C logic tests SHFRA07. These inputs control the N bit for most operations.

The input B logic tests for CMP.B instructions. Under these conditions, if SHFRA15 is 0, the N bit is set, and if SHFRA15 is 1, the N bit is cleared. Input D covers all cases where the N bit is not affected by the current operation, and is therefore reloaded with the previous content, IRCH N(1) H. Input E allows IRCF NDATA L to be asserted by BR03 for load PS and load FCC functions. Table 7-6 summarizes the sources of N bit data.

Table 7-6
N Bit Data Sources

Instruction	CC Control ROM		IRCF NDATA L Source
	MODZN	ENZN	
CMP.B	0	1	N ← 1 if -SHFRA15 = 1 N ← 0 if SHFRA15 = 1
DIV	0	0	non-affected
MUL	1	0	non-affected
all other instruction-dependent codes	1	1	N ← 1 if SHFRA15 = 1 (word or odd byte) N ← 1 if SHFRA07 = 1 (byte)
SWAB			N ← 1 if SHFRA08 = 1
Condition-Code Load Signal			
IRCF CCLD4			N ← if SHFR = 0
IRCF CCLD5			N ← if SHFR = 0
IRCF CCLD6			non-affected
IRCF CCLD7			non-affected

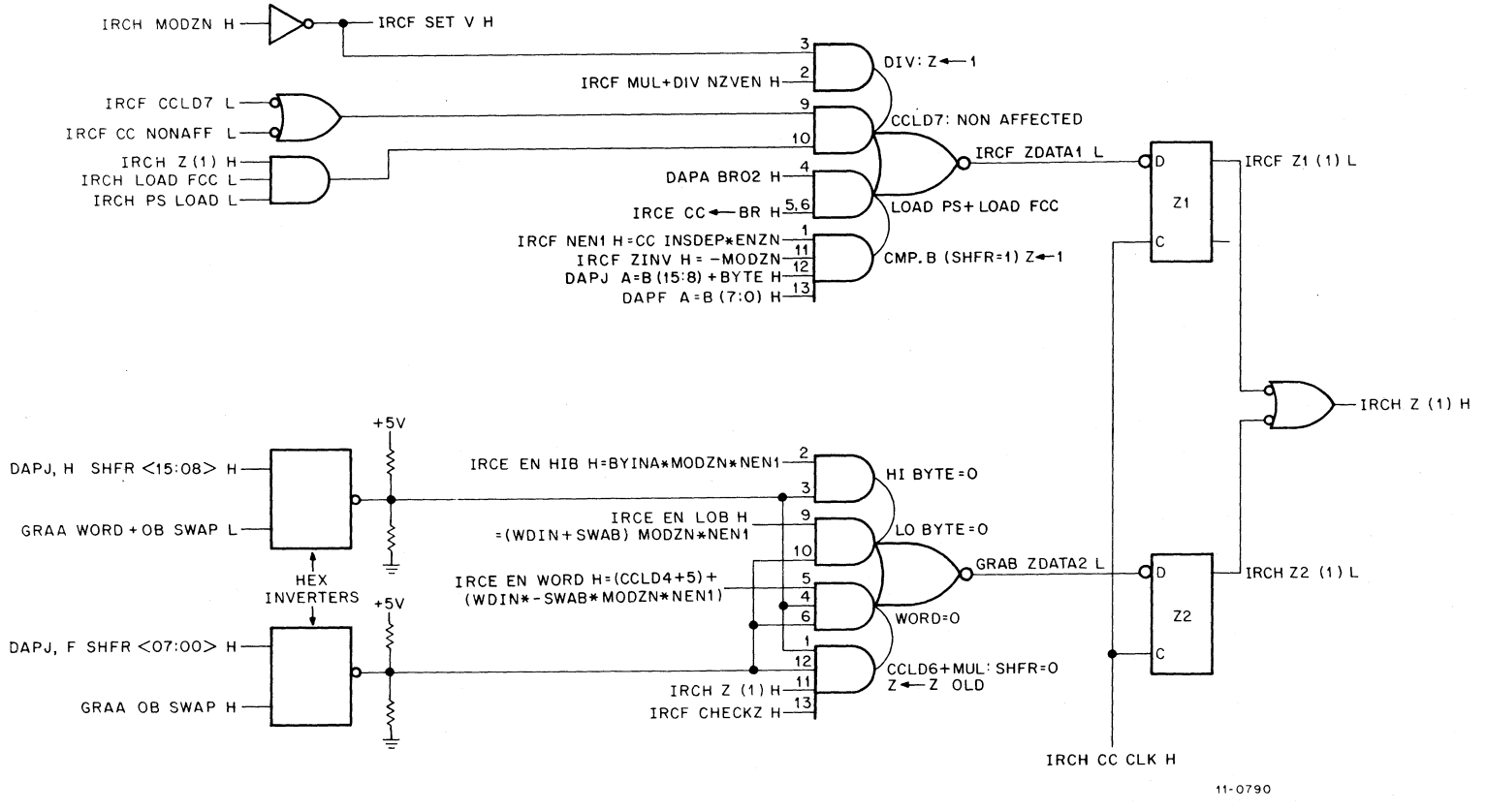


Figure 7-3 Sources of Z Bit Data, Simplified Diagram

7.3.7 Z Bit Data

The Z (zero) bit of the PSW is set when the result of a processor operation is 0. The Z bit data that controls the condition code is generated by logic on drawings IRCF and GRAB.

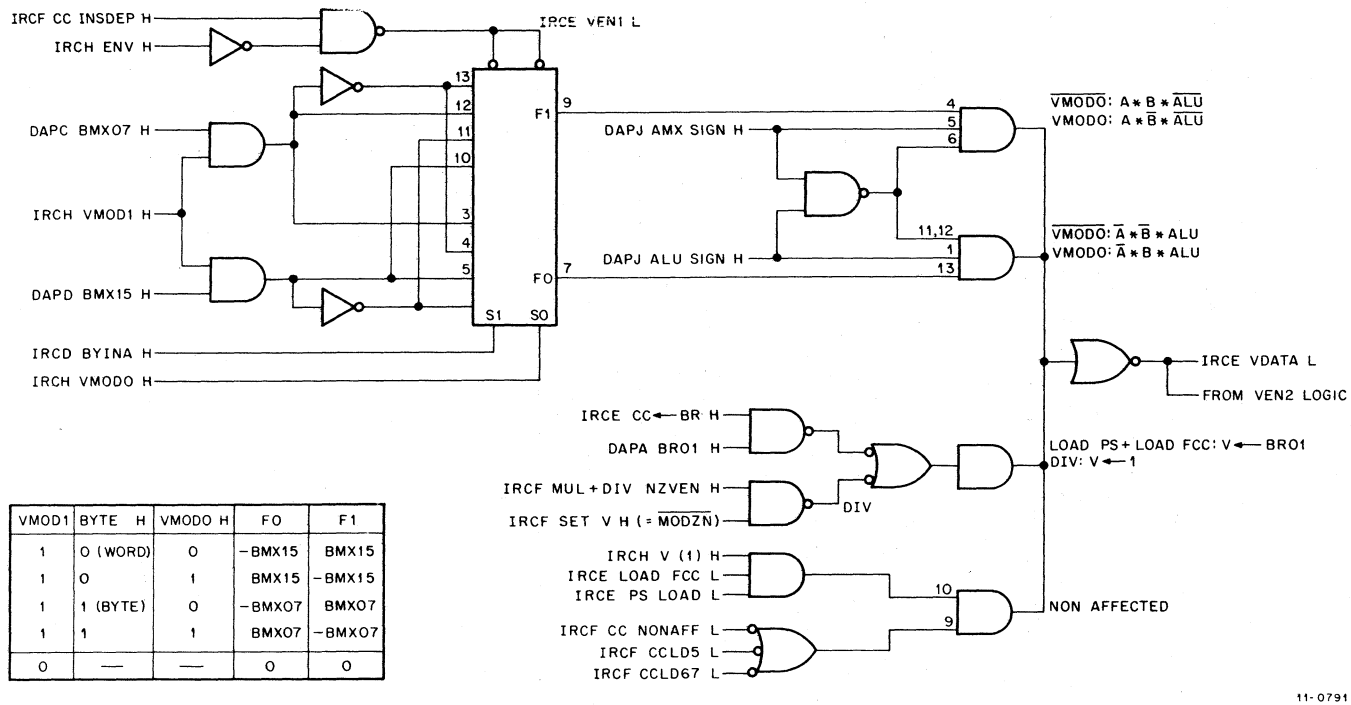
Figure 7-3 is a simplified diagram of the logic that asserts IRCF ZDATA1 L and GRAB ZDATA2 L. These outputs are clocked into the Z1 and Z2 flip-flops, whose contents are ORed to provide the Z bit of the PSW condition code.

7.3.7.1 ZDATA1 Sources – The input gates that assert IRCF ZDATA1 L cover the special conditions that control the Z bit, independent of the SHFR outputs being equal to 0. For example, during the DIV instruction execution, MODZN and ENZN are both low and the Z bit is set. For the special case of the CMP.B instruction, the logic tests for the SHFR output = 1 condition to determine the Z bit. The other input gates that assert IRCF ZDATA1 L test for load PS or load FCC operations and operations that have no affect on the Z bit. Under the former conditions, the Z bit is loaded from BR02 and under the latter conditions, the Z bit is unchanged [Z(1) H controls ZDATA1]. These special conditions are summarized in Table 7-7.

7.3.7.2 ZDATA2 Sources – The logic that generates GRAB ZDATA2 L tests the SHFR output for 0. The open-collector inverters function as 0 detectors for SHFR ⟨15:08⟩ and SHFR ⟨07:00⟩. The enabling inputs IRCE EN HIB H, IRCE EN LOB H, and IRCE EN WORD H are used to test each byte of the SHFR separately, or together. The additional GRAB ZDATA2 gate tests the SHFR output word for 0 under CCLD6 or MUL conditions. If the SHFR output is 0, the previous Z bit condition, Z(1)H, controls the new Z bit.

Table 7-7
Z Bit Data Sources

Instruction	CC Control ROM		Z Data Source
	MODZN	ENZN	
CMP.B	0	1	Z ← 1 if SHFR = 1
MUL	1	0	Z ← 2(1)H if SHFR = 0
DIV	0	0	Z ← 1
SWAB			Z ← 1 if SHFR ⟨07:00⟩ = 0
all other instruction-dependent codes	1	1	Z ← 1 if SHFR = 0
Condition-Code Load Signals			
IRCF CCLD4			Z ← 1 if SHFR = 0
IRCF CCLD5			Z ← 1 if SHFR = 0
IRCF CCLD6			Z ← Z(1)H if SHFR = 0
IRCF CCLD7			non-affected



VMOD1	BYTE H	VMODO H	FO	F1
1	0 (WORD)	0	−BMX15	BMX15
1	0	1	BMX15	−BMX15
1	1 (BYTE)	0	−BMXO7	BMXO7
1	1	1	BMXO7	−BMXO7
0	—	—	0	0

Figure 7-4 VEN1 Sources of V Data Bit, Simplified Diagram

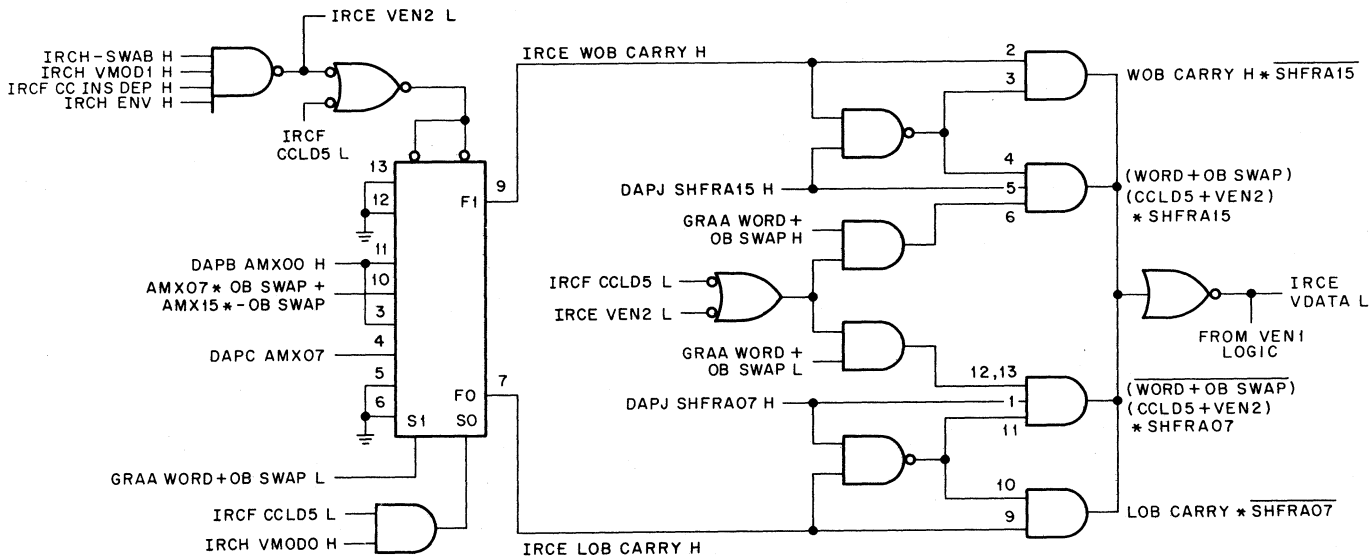
7.3.8 V Bit Data

The V (overflow) bit of the PSW is set when a processor operation results in an arithmetic overflow. The logic that generates the V bit data is shown on drawing IRCE. The V bit is affected by two broad categories of instructions: arithmetic operations, and word or byte operations. The results of these operations and other special cases determine IRCE VDATA L. To simplify the description, arithmetic operations, and special cases are grouped as VEN1 inputs (Paragraph 7.3.8.1). Word and byte operations are grouped as VEN2 inputs (Paragraph 7.3.8.2). Table 7-8 summarizes the V bit data sources of both groups.

Table 7-8
V Bit Data Sources

Instruction	CC Control ROM			IRCE VDATA L Source*
	VMOD1	VMOD0	ENV	
VEN1				
INC.B, ADC.B	0	0	0	V ← -A*ALU15
DEC.B, SBC.B	0	1	0	V ← A*-ALU15
NEG.B, ADD	1	0	0	V ← A*B*-ALU15 + -A*-B*ALU15
SUB, CMP.B	1	1	0	V ← A*-B*-ALU15 + -A*B*ALU15
VEN2				
MFP, MTP, SXT, CLR.B, COM.B, TST.B, MOV.B, BIT.B, BIC.B, BIS.B, MUL, ASH, ASHC, XOR	0	0	1	V ← 0
DIV	0	0	1	V ← 1
ROL.B, ASL.B	1	0	1	V ← SHFRA15 ∀ AMX15
ROR.B, ASR.B	1	1	1	V ← SHFRA15 ∀ AMX00
Condition-Code Load Signals				
IRCF CCLD4				V ← 0
IRCF CCLD5		(VEN2)		V ← V old + (SHFRA15 ∀ AMX15)
IRCF CCLD6		(VEN1)		non-affected
IRCF CCLD7		(VEN1)		non-affected

*A = DAPJ AMX SIGN H
B = DAPD BMX15 H (word) or DAPC BMX07 H (byte)
ALU15 = DAPJ ALU SIGN H



WORD OR ODD BYTE SWAP	VMODO * $\overline{\text{CCLD5}}$	IRCE LOB CARRY	IRCE WOB CARRY
YES	NO	0	AMX07 (ODD BYTE) AMX15 (WORD)
YES	YES	0	AMX00
NO	NO	AMX07	0
NO	YES	AMX00	0

11-0792

Figure 7-5 VEN2 Sources of V Data Bit, Simplified Diagram

7.3.8.1 **VEN1** – Figure 7-4 is a simplified diagram of the V bit data sources that are grouped in the VEN1 category. A 74S153 Dual 4-Line-to-1-Line Multiplexer is used to select the most-significant BMX bit for the arithmetic operations that involve the B input. These are NEG.B, ADD, SUB, and CMP.B, as indicated in Table 7-8. For these instruction-dependent codes, the CC CNTL ROM asserts IRCH VMOD1 H, which gates the BMX outputs to the multiplexer inputs, and IRCE VEN1 L, which enables the multiplexer. IRCD BYINA H selects BMX15 or BMX07 as the most-significant bit. IRCH VMOD0 H selects the BMX bit or its complement at each output, as shown on the multiplexer truth table in Figure 7-4.

The notation on Figure 7-4 indicates the conditions and functions for which each AND gate input asserts IRCE VDATA L.

For INC.B, ADC.B, DEC.B, and SBC.B instruction-dependent codes, CC CNTL ROM output IRCH VMOD1 H is low. As a result, the BMX multiplexer outputs are always 0. For these instructions, B is eliminated from the source function, as listed in the source column of Table 7-8.

7.3.8.2 **VEN2** – Figure 7-5 is a simplified diagram of the V bit data sources that are grouped in the VEN2 category. A 74S153 Dual 4-Line-to-1-Line Multiplexer selects the most-significant AMX bit for the word, odd-byte, or byte operations. The multiplexer truth table is shown on Figure 7-5. The multiplexer is only enabled by CCLD5 or those instruction-dependent codes for which the CC CNTL ROM asserts IRCH VMOD1 H and IRCH ENV H. As indicated in Table 7-8, these instructions include ROL.B, ASL.B, ROR.B, and ASR.B. For these instructions, the notation on Figure 7-5 indicates the conditions and functions for which each AND gate input asserts IRCE VDATA L.

For the majority of the instructions included in the VEN2 group of Table 7-8, VMOD1 is low. As a result, the AMX multiplexer is not enabled and none of the AND gate inputs will be enabled because IRCE VEN2 L is not asserted. Therefore, processing these instructions clears the V bit.

7.3.9 Condition Code Storage

Refer to drawing IRCH. The circuits shown on the top half of this drawing are used to store the processor condition codes; the remainder of the drawing shows circuits concerned with the subsidiary ROMs used in condition code calculation, instruction decoding, and arithmetic and logic unit (ALU) control.

The four condition-code bits, N, Z, V, and C, are stored in the four least-significant bits of the processor status (PS) word. The remaining bits of the PS, and the PS loading and reading logic, are on the PDR module and are shown on drawing PDRD. The condition codes are normally loaded to reflect the result of each instruction that operates on data. When this is done (by clocking the data inputs to each flip-flop), each bit takes on the value of the corresponding signal from the condition code generation logic on drawings IRCE and IRCF. Two Z bit flip-flops, provided to avoid the delay of a final stage OR gate before the clock time, are shown on drawing IRCF.

7.3.9.1 **Clocked Inputs** – IRCH CCLK H clocks the condition-code flip-flops immediately following each ROM cycle (T6 is the T1 of the following cycle) except when the clock is inhibited by a value of 2 in the condition code (CCL) bits in the microprogram. In many cases where the condition codes are clocked, individual bits may remain unaffected by loading the bit from itself, through the combinational logic that generates the condition codes.

7.3.9.2 **BR Inputs** – The condition code flip-flops can be loaded directly from the BR. This is done whenever the bus address transmitted by the processor addresses the low byte of the processor status (PS) word. UBCC CC DATA (1) H indicates this condition and is used to gate the BR bits into the direct-set and direct-clear inputs of the flip-flops. Complements are applied to set and clear inputs, so that each flip-flop is correctly set or reset.

7.3.9.3 **IR Inputs** – A third method of modifying the condition codes allows bits to be set or cleared directly from the instruction. The four least-significant bits of the IR are connected to either the set or the clear inputs of the flip-flops, but not both. The selection of inputs is done by two enabling signals that are generated from opposite polarities of IR04. The same polarity inputs from the IR are used for either setting or clearing; only bits which are 1s in the IR are altered, and the remaining bits are not affected.

When the condition codes are set or cleared from the IR, the normal clocking of the flip-flops is inhibited. When the condition codes are loaded from the BR, the loading signal is present beyond the time when the data inputs are clocked, so the BR inputs take precedence. Unless one of these two conditions is true, the normal clocked input is used.

The Z bit is stored in two flip-flops shown on drawing IRCF. The flip-flop outputs are ORed to generate the value of the condition-code bit. If either flip-flop contains a 1, the Z bit is considered to be a 1. Both flip-flops are set or cleared together when either the BR or the IR bits are transferred to the condition codes. The signal DPCC Z (0) H is not used.

7.3.9.4 **Condition Code Subsidiary ROMs** – The logic on the lower half of drawing IRCH is used to generate addresses for the subsidiary ROMs (CC CNTL, INSTR DECODE, and ALU CNTL). The subsidiary ROMs contain values for different instructions, so the addresses that are generated correspond to individual instructions. The IR provides all inputs to the address generation logic.

Each subsidiary ROM contains 32 8-bit words. The 32 addresses are organized as follows:

a. Addresses 0–7 are used for instructions with op codes containing 06 in IR ⟨14:09⟩. These include the rotates, shifts, MARK, MFP, MTP, and SXT.

b. Addresses 10–17 are used for instructions with op codes containing 07 in IR ⟨14:09⟩. These are the unary instructions.

c. Addresses 20–27 are used for binary instructions (IR ⟨14:12⟩ contains any value from 1 to 6).

d. Addresses 30–37 are used for the register destination instructions, which have a 7 in IR ⟨14:12⟩. These include multiply and divide, the long shifts, and XOR.

NOTE
All addresses for the subsidiary ROM words are in octal.

7.3.9.5 ROM Address Multiplexer – The ROM address is generated by two 74S153 multiplexers for the low four bits, and by an OR gate for the most-significant bit. The address signals are IRCH SUBROMA4 H through IRCH SUBROMA0 H. For the 05 and 06 class instructions, the four least-significant address lines are driven directly from IR ⟨09:06⟩ through the C3 inputs of the multiplexers, and SUBROMA4 is not asserted. For the register destination instructions, SUBROMA4 is asserted, SUBROMA3 is driven by a +3V input to the multiplexer, and the remaining three address bits take on the value of IR ⟨11:09⟩ through the C2 inputs of the multiplexer. For binary instructions, the C1 inputs of the multiplexer are used; SUBROMA4 is asserted and SUBROMA 3 is clear. This data is summarized in Table 7-9.

Table 7-9
Subsidiary ROM Address Sources

Instruction Class IRCA IR ⟨14:12⟩	ROM Address Multiplexer Select		Input Selected	Subsidiary ROM Address Source				
	S1	S0		A4	A3	A2	A1	A0
05 and 06 class IR ⟨14:12⟩ (0) H	H	H	C3	0	IR09	IR08	IR07	IR06
register destination IR ⟨14:12⟩ (1) H	H	L	C2	1	1	IR11	IR10	IR09
binary class -IR ⟨14:12⟩ (0) H -IR ⟨14:12⟩ (1) H	L	H	C1	1	0	IR14	IR13	IR12
not used	L	L	C0					

The SUB instruction is treated specially, to separate the ADD and SUB instructions when generating ROM addresses. Both SUB and ADD would normally generate ROM address 26 (the op codes differ only in bit 15). When the SUB instruction is decoded, the four least-significant bits of the ROM address are forced to 0s to generate address 20. Addresses 27, 35, and 36 are not used. For the SWAB instruction, which is not in any of the four groups that generate ROM addresses, the contents of the IR generate the same ROM address that is used for the ASL instruction. The signal IRCH SWAB L is used to distinguish between the two instructions. The UALU signals are used to recognize that the ALU control is instruction-dependent, and that the outputs of the ALU control ROM on drawing GRAA are active.

7.3.9.6 Subsidiary ROMs – The CC CNTL and INSTR DECODE ROMs shown on drawing IRCH generate signals that are used to control the condition-code generation only when the main microprogram ROM contains a 1 in the CCL bits and are used for further instruction decoding. Each ROM is a DM8598 integrated circuit that stores 256 bits in 32 8-bit words. A complete list of subsidiary ROM addresses and functions is shown on drawing IRCJ.

7.4 PDR MODULE M8104

The Processor Data and Unibus Registers (PDR) Module M8104 contains the logic elements that form the processor data registers and the Unibus registers.

7.4.1 Bus Register Multiplexer

Refer to drawing PDRA. The bus register multiplexer (BRMX) selects one of four inputs to the BR. These inputs, PDRA BRMX ⟨15:00⟩ H, are also used to load the IR. Table 7-10 lists the BRMX inputs.

Table 7-10
BRMX Input Sources

Input	Control	Description
DAPF, H, J SHFR ⟨15:00⟩ H	RACA UBRX L	output of the shifter, contains results of last data manipulation by the processor
SMCE MEM D ⟨15:00⟩ H	TMCF SELMEM L	data from high speed semiconductor memory
BUS INTD ⟨15:00⟩ H	TMCF SELINT L	data from internal processor or memory management unit, switch register, or floating-point processor
BUSA BUS D ⟨15:00⟩ L	none	Unibus data

The BRX bit in the microprogram selects the shifter output when 1. TMCF SELMEM L and SELINT L are derived from a Fastbus device or address response. The Unibus is selected by default if no other bus is directly selected.

The drawing also shows the inverters that receive the internal bus data. Both the internal bus and the fast memory bus are terminated at the multiplexer; the Unibus is received through standard Unibus receivers and is terminated separately by an M930 Bus Terminator module.

7.4.2 Bus Register A and Light Register

Refer to drawing PDRB. The output of BRMX is loaded into BR and IR. The primary BR, which is used as a source of data input to the arithmetic and logic unit (ALU), is on the DAP module and is shown on drawing DAPA. However, for timing considerations and to reduce the number of pins used on the DAP module, a copy of the BR (called BRA) is made and distributed to:

a. KT11-C Memory Management Unit
b. FP11 Floating-Point Processor
c. MS11 Semiconductor Memory System
d. Operating System Tester (option)
e. Unibus (via a multiplexer on PDRE)

The four least-significant bits of the BRA are also routed to the IRC module for use in direct loading of the processor condition codes; the condition codes are part of the processor status (PS) word, and the remainder of the PS is shown on drawing PDRD. All active bits of the PS can be loaded from the BRA.

The light register (LR) is primarily a maintenance tool. It is directly loaded from the BRA whenever a DATO data transfer to the bus address of the switch register (177570) takes places. The contents of the LR can be displayed in the console data lights by setting the DATA display select switch to DISPLAY REGISTER.

Drawing PDRB also illustrates a set of inverters used to provide buffered outputs from the BRA to the many inputs it drives in the memory management unit and the semiconductor memory system.

7.4.3 Program Break Register

Refer to drawing PDRC. The program break (PB) register is used as a maintenance aid, to enable checkout of the microprogram operation and to allow control of the processor operation by stopping the processor in a specified microprogram state.

The PB is an 8-bit register that is loaded by moving data to physical address 777770. The contents of this register are continuously compared to the microprogram ROM address RAR ⟨07:00⟩ by two 7485 4-bit comparators. The comparators generate the signal PDRC PB CMP H whenever the two numbers are equal. When the processor is being controlled by a maintenance module, this signal can be used to stop the processor at T2 of the specific microprogram state. This allows examination of certain specific machine states without manually clocking the processor through all the intervening states.

PDRC PB CMP H is ANDed with TIGA T1 (1) H to provide TIGB PB SYNCH H, which can be used as a synchronization point for oscilloscope loops during maintenance tests.

7.4.4 Stack Limit Register

Refer to drawing PDRC. The stack limit (SL) register is an 8-bit register that is loaded by moving data to physical address 777775. The contents of this register are compared with the eight most-significant bits of the bus address being transmitted by the processor. These eight bits specify a 256-byte (128-word) boundary, below which the kernel stack must not store any data. This means that (ignoring the effects of virtual memory) whenever an address is transmitted while the processor is using the kernel stack pointer register 6 as the address source, and the operation is not DATI, this address must not refer to any location below the boundary set by the SL.

The comparator that checks the address against the SL generates one of two signals: PDRC STACK LIMIT H or PDRC RED ZONE H.

PDRC STACK LIMIT H indicates that the address is addressing a 128-word block beginning at the stack limit. If the error detection circuit on the TMC module determines that the address is to one of the 16 words at the top of this block (with the highest addresses), the reference is considered to be a yellow zone, or non-fatal reference. Upon completion of the instruction or trap sequence, the processor will trap, using the trap vector stored in kernel virtual location 04, to a routine that must correct the stack problem. However, if the address refers to a location below the top 16 words of the block, the reference is considered a red zone, or fatal error; the processor performs an emergency recovery by:

a. aborting the instruction or trap sequence immediately
b. storing the current PC and PS in locations 0 and 2
c. using the trap vector at location 4

PDRC RED ZONE H indicates that the processor is addressing a location below the stack limit (in a 128-word block below the boundary). This is always a red zone error.

Note that for both signals, the data transfer operation is considered an error only if the address is derived from the kernel stack pointer register 6. This is determined by the logic on the TMC module (Paragraph 7.6.5.1).

When the memory management unit is in operation, any bus error or memory management abort that occurs during an address reference derived from kernel stack pointer register 6 is treated as a red zone stack error, regardless of the value in the stack limit register.

The SL is initialized to a value of 0. With this value, any stack reference to an address below 000400 is an error. References to addresses between 000377 and 000340 are yellow zone errors, and references to addresses 000337 or below are red zone errors.

7.4.5 Program Interrupt Register

Refer to drawing PDRD. The program interrupt register (PIR) provides a means of scheduling software routines through the same priority structure used to control hardware interrupts from peripheral devices.

The PIR is divided into two parts: the 7-bit PIR and the encoded value of the highest request level, the PIA. The PIR register is used to generate interrupt requests at seven of the eight possible priority levels in the PDP-11 System. The requests are compared with the current processor priority stored in PS ⟨07:05⟩; if a PIR request is higher than the current processor priority, the processor traps at the end of the current instruction, using the interrupt vector at location 240. The PIR request levels relate to processor priority levels 1 through 7. There is no PIR request corresponding to priority 0 because the processor priority can never be lower than 0, and such a request can not be honored. A PIR request is of higher priority than any bus request at the same or lower priority.

The PIA encoder generates a 3-bit number that represents the highest level request stored in the PIR. This number is transmitted on two sets of data lines whenever the processor reads the PIR word. The three PIA signals are connected to the DMX inputs (drawing PDRE) for bits 7 through 5, so that the programmer can move the low byte of the PIR word into the processor status register and thus set the processor priority to the level of the request honored if desired. This locks out all requests on the same level or below. The same three bits are also routed to the DMX inputs for bits 3 through 1; in this position the encoded value can be used as an index constant in dispatching to an interrupt service routine for the appropriate priority level request.

7.4.6 Processor Status Register

Refer to drawing PDRD. The PS stores several types of data that are dependent on the process being performed. This data must be stored whenever the processor changes processes; typically, this occurs every time there is an interrupt or a trap. Because the contents of the PS control many parts of the operation of the processor, modifications of the contents are carefully controlled.

The four fields of information in the PS are:

a. the processor condition codes
b. the trace (T) bit
c. the processor priority
d. the processor mode control and register set selection bits

7.4.6.1 Condition Codes – Refer to drawings IRCF and IRCH. The condition codes occupy bits 3 through 0 of the PS register. The logic that senses the data conditions and stores the selected indications is on the IRC module; the gates that control the reading of the condition codes onto the internal data bus are shown on drawing PDRD. When the PS is explicitly addressed at physical address 777776, the data transfer is on the Unibus; the internal bus is used only under direct microprogram control.

The condition codes are loaded automatically with the results of most data manipulations. In addition, the codes can be manipulated by a microcoded instruction that can set or clear individual condition code bits. Any operation that transmits data directly to the processor status word inhibits the setting of the condition codes because the data transmitted is loaded into PS ⟨03:00⟩ directly. This is done for move instructions that address the PS, RTI instructions that pop a value off the hardware stack into the PS, or interrupt service sequences that load the PS from the interrupt vector.

7.4.6.2 T Bit – The trace (T) bit is provided as a software diagnostic aid. When this bit is set, a processor trap will be vectored through location 14. This trap occurs at the end of the instruction that is being performed when the T bit is being set, unless:

a. The instruction is a return from trap (RTT) instruction. In this case, the trap is delayed until the end of the following instruction.

b. Some other trap or interrupt condition is honored. In this case, the PS containing the T bit is pushed onto the stack and all trace operations are deferred until the PS word is popped off the stack at the end of the trap or interrupt service routine.

The T bit can not be set by moving data to the PS; the only way the T bit can be set is by popping a word off the hardware stack with bit 4 set. This can be done with an RTI or RTT instruction. The purpose of inhibiting other methods of loading the T bit is to protect the user from inadvertently setting the T bit while changing the processor priority or condition codes.

7.4.6.3 Priority Bits – The processor priority is stored in PS ⟨07:05⟩. The 3-bit priority field is interpreted as one of eight priority levels. This level is compared with other requests for control of the system. These requests can be external to the processor, in the case of Unibus requests, or internal, in the case of program interrupt requests. In general, the purpose of requesting control of the system is to interrupt the current processor program and run a service routine or higher priority program before returning control to the interrupted program. Devices that need the use of the Unibus request Unibus control on a non-processor request (NPR) level that is effectively higher than any processor priority; but these devices must not perform the INTR bus operation without gaining control of the Unibus via a BR level.

The processor priority level may be set by directly transferring data to the PS, by popping a new PS from the hardware stack, or by loading the PS from an interrupt or trap vector. In addition, the processor priority may be explicitly set by the set priority level (SPL) instruction. While in all other cases the priority is set from data bits 7 through 5, in the SPL instruction, the priority is loaded from bits 2 through 0. A 2-input multiplexer controls the loading of the priority flip-flops from the appropriate source. In user mode, the processor priority can only be changed by a transfer to the explicit address of the PS (777776). This is possible only if memory management mapping allows it.

7.4.6.4 General Register Set Bit – This bit indicates general register set 0 (when cleared) or general register set 1 (when set). For an explicit reference to the PSW, it is loaded from BR11 and may be set or cleared. For implicit operations, such as RTI, it can only be set, allowing the kernel some control over which register set user or supervisor mode programs can use.

PDRB BR11A H is used to direct-set PS11 at time T4 if the microprogram IBS value is 2.

7.4.6.5 Previous Mode Bits – PS bits PS13 and PS12 store the processor mode previous to the last interrupt or trap. Data is clock-set or direct-set into these flip-flops from BRA or PS ⟨15:14⟩. For example, PS13 and PS12 can be set or cleared by operations that explicitly reference the PSW; they are loaded from BR13 and BR12. They can only be clocked and set by implicit references, unless operating in the kernel mode. This allows a kernel mode program to return to kernel, supervisor, or user mode; a supervisor mode program to return to supervisor or user mode; and a user mode program to only return to user mode. A user or supervisor mode program can not use the RTI instruction to enter the kernel mode. When a new PS is loaded from the trap or interrupt vector, the old contents of PS15 and PS14 are loaded into PS13 and PS12.

7.4.6.6 Current Mode Bits – PS bits PS15 and PS14 control and indicate the current processor mode. The source of input data is always BR15A and BR14A, whether the PS is loaded by an RTT or RTI instruction, or if a new PS is loaded from a trap or interrupt vector, or explicitly referenced. These bits can only be set by implicit references.

7.4.6.7 Read PS – The entire PS word can be gated to the internal data bus by PDRD READ PS H, which is generated by a microprogram IBS field value of 3. This value is used in microstates RSD.00, RSD.02, BRK.20, BRK.80, TRP.00, TRP.01, TRP.02, and HLT.00 to get the current PS into the BR.

7.4.7 Unibus A Data Multiplexer

Drawing PDRE shows the Unibus A data multiplexer (DMX), which selects one of five data sources within the processor for transmission on the Unibus A data lines, BUSA D ⟨15:00⟩ L. The five registers that can be read by the Unibus are:

a. bus register A PDRB BR ⟨15:00⟩ A H

b. stack limit register PDRC SL ⟨07:00⟩ H (HI BYTE) and program break register PDRC PB ⟨07:00⟩ H (LO BYTE)

c. program interrupt register PDRD PIR ⟨15:09⟩ (1) H and PIA code PDRD PIA ⟨02:00⟩ H

d. processor status register PDRD PS ⟨15:11⟩ (1) H, priority bits, and condition codes

One of the five registers is selected by selecting one of the four multiplexer inputs and then enabling one or both bytes of the DMX. The DMX selection chart is shown on drawing PDRE. When the BR, the PIR, or the PS is addressed, the logic on the TMC module selects the corresponding DMX input and enables both bytes; for the SL or the PB, the same DMX input is used, but only the high byte is enabled for the SL and only the low byte for the PB.

The byte enabling signals also prevent the processor from putting any data on the Unibus A data lines except when one of these registers is selected.

7.4.8 Display Multiplexer

Drawing PDRF shows the display multiplexer that selects one of four inputs for the data display lights on the console provided with the KB11-A processor. The display multiplexer is implemented with eight 74S153 Dual 4-Line-to-1-Line Multiplexers that are controlled by two display-select signals from a 4-position switch on the console.

The display selected for each switch position is described in Table 7-11.

7.4.9 Console Interconnections

Drawing PDRH shows the interconnections between the processor and the switch register and data display lights on the console.

Connector J1 is the physical connection; there are 18 signal lines from the console switches, 16 signal lines to the console from the display multiplexer, and two signals from the multiplexer control switch on the console. SSRB SR0 MODE 0 H is one of the inputs that indicate the mode to the console.

The 16 least-significant bits of the switch register SWR ⟨15:00⟩ can be gated onto the internal data bus. This is done whenever an instruction attempts to read data from the switch register address (TMCD SW ADRS L) or during console operations when the switch register is gated onto the internal data bus by TMCF READ SW L (IBS = 1).

Table 7-11
Display Multiplexer

Switch Position	PDRH DISP DATA		Description
	SEL1	SEL0	
BUS REGISTER	H	H	bus register A PDRB BR ⟨15:00⟩ A H
DATA PATHS	H	L	shifter outputs DAPH SHFR ⟨15:00⟩ H
DISPLAY REGISTER	L	H	light register, which is provided for maintenance purposes and is loaded by addressing the switch register
μADRS FPP/CPU	L	L	processor ROM address RACD RAR ⟨07:00⟩ AH into low byte
			floating-point processor ROM address FRMB CRAR ⟨7:0⟩ (1) H into high byte

7.5 RAC MODULE M8103

The ROM and ROM Control (RAC) Module M8103 contains the main microprogram ROM elements and ROM address control logic, including the fork A instruction decode logic, the A fork instruction register (AFIR), and the microprogram branch control logic.

7.5.1 ROM Address Register (RAR)

Refer to drawings RACA through RACD. There are three copies of the RAR. In addition to the two copies (RARB and RARA) used to provide sufficient fanout for the 16 ROM ICs, a third copy (RAR) is used to transmit the current microprogram word address to the KT11-C ROM. Each of the three copies has seven clearable bits; the 21 clearable bits are stored in four of the 6-bit registers, leaving three bits unused. Drawing RACA shows one of the four registers, containing five bits of the RARB copy. The remaining two bits are in a register shown on drawing RACC. Each of the three most-significant bits is implemented by a single flip-flop that can be direct-set by the signal RACA ZAP L. This signal also clears all other RAR bits.

The RAR is normally loaded from inputs generated by the microprogram address selection logic shown on drawing RACL. Under exceptional circumstances, the RAR is forced to address 200 by clearing all but the most-significant of the eight bits, and setting that bit. To permit setting the most-significant bit, it is implemented by a separate flip-flop. The remaining seven bits are implemented by 6-bit registers of the same type used for the ROM output buffer.

RACA ZAP L is the signal used to force the processor into a known state to start the processing of traps, power-up sequences, and various types of aborts. Among the conditions that can generate this signal are:

a. power-up sequence or start sequence (ROM INIT)

b. bus errors or processor traps (TMCC ABORT H and RACB UBSD01 H indicate that the processor is pausing for a bus operation. The signal TIGD TS2 L remains asserted longer than the pulse TIGC T3 L that clocks the RAR, and ensures that the ZAP signal overrides the normal address.)

7.5.2 Microprogram ROM and Buffer Register

All control signals that are dependent only on the machine state (i.e., that are not dependent on asynchronous signals or on data inputs) are derived directly from the outputs of the microprogram ROM. The ROM contains 256 64-bit words; during each processor cycle, one word is fetched from the ROM and stored in a buffer register. The outputs of the buffer register are transmitted to the other modules of the processor to act as control signals or to be used in combinational logic that generates control signals for all processor operations.

The ROM is implemented by 16 type 74187 256-word X 4-bit read-only memories. The buffer register is implemented primarily by 74S174 D-type hex flip-flop registers. (Some bits are implemented by individual flip-flops to provide separate input clocking or greater output load capacity.)

The ROM itself is implemented with open collector outputs that require termination by resistive dividers which maintain a +3V signal level when the ROM outputs are not low.

Various ROM bits are clocked into the output buffer register at different times. Most bits are clocked by the processor T1 pulse, while others are clocked by the T2 pulse. Certain bits are clocked on the trailing edge of the T1 pulse to allow slightly more time for the processor to complete operations started by the previous machine cycle (Figure 7-6 and Table 7-12).

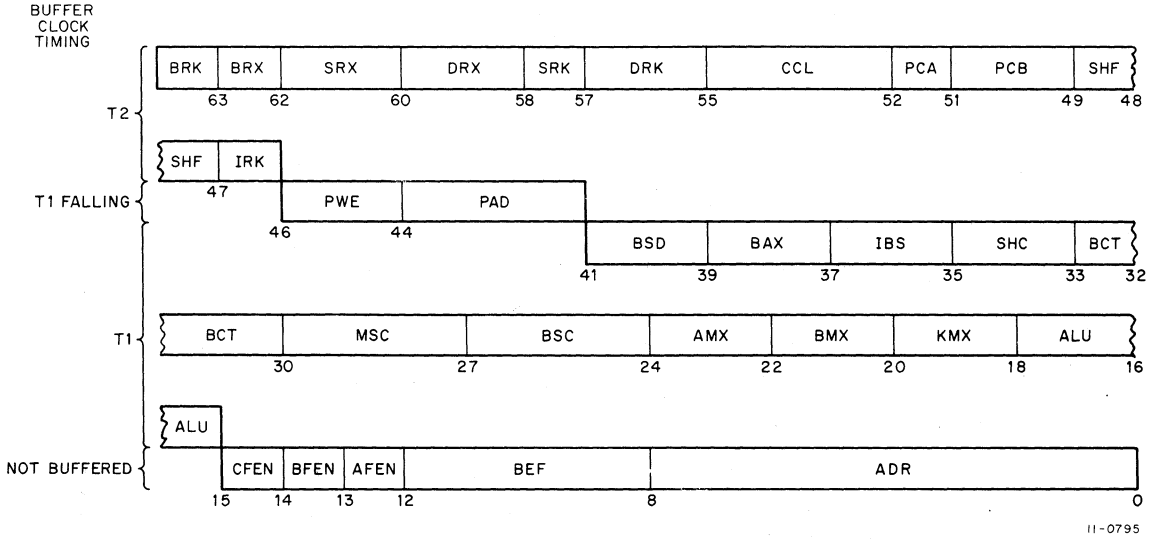


Figure 7-6 Sequence of ROM Bit Clocking

The buffer register shown on drawing RACA is clocked by the T2 pulse; none of the control signals transmitted from the 18 bits of storage on this drawing can be assumed to have settled before the T3 pulse.

Five output signals are derived from the contents of a buffer register that is clocked by the falling edge of the T1 pulse, rather than the leading edge (drawing RACB). These signals (two pad write-enable and three pad address lines) control the writing of information into the processor general registers. The data is transferred into the registers by clocking it on the T1 pulse, so these signals must not change until after the T1 pulse has occurred.

One of the 6-bit registers shown on drawing RACC stores the output of bits 34 and 32 through 28 of the ROM. Bit 33 is stored in a separate flip-flop. This permits the buffer register to transmit both polarities of the signal

USHC00, with no additional signal delays. The shift counter logic, which is shown on drawing RACM, is on the same module as the ROM, so the USHC signals are not brought to pins. Bit 27 of the ROM, which generates UMSC00, is also stored in a separate flip-flop to generate both polarities.

Table 7-12
Microprogram Bit Usage

Bit Positions	Contents	Clocked At
63	bus register clock (UBRK)	T2
62	bus register multiplexer (UBRX)	T2
61–60	source register MUX (UBRX)	T2
59–58	destination register MUX (UDRX)	T2
57	source register clock (USRK)	T2
56–55	destination register clock (UDRK)	T2
54–52	condition-code load (UCCL)	T2
51	program counter A CLK (UPCA)	T2
50–49	program counter B CLK (UPCB)	T2
48–47	shifter control (USHF)	T2
46	instruction register CLK (UIRK)	T2
45–44	pad write-enable (UPWE)	T1 falling
43–41	scratchpad address (UPAD)	T1 falling
40–39	bus delay (UBSD)	T1
38–37	bus address multiplexer (UBAX)	T1
36–35	internal bus (UIBS)	T1
34–33	shift counter (USHC)	T1
32–30	bus control (UBCT)	T1
29–27	miscellaneous control (UMSC)	T1
26–24	bus conditions (UNSC)	T1
23–22	A multiplexer (UAMX)	T1
21–20	B multiplexer (UBMX)	T1
19–18	constant multiplexers (UKMX)	T1
17–15	arithmetic logic unit cont (UALU)	T1
14	fork C enable (UCFEN)	not buffered
13	fork B enable (UBFEN)	not buffered
12	fork A enable (UAFEN)	not buffered
11–08	branch-enable (UBEF)	not buffered
07–00	microprogram address (UADR)	not buffered

The microprogram bits which are used to calculate the new ROM address are used only on the RAC module, so they are not brought to module pins. However, several of the branch-enable signals are required either in both polarities or with greater fanout capacity; UBEF03, UBEF01, and UBEF00 are buffered by more than one gate.

7.5.3 Fork A Instruction Decoding

Normally, the address of the next microprogram word is derived from the contents of the microaddress field (UADR) in bits 7 through 0 of the current microprogram word. Two branch selectors allow 2-way or 4-way branches on the conditions of various processor circuits and on the contents of various data registers. For most decision points encountered during the flow of machine states, this branch capability is sufficient.

In certain situations, particularly after an instruction or data has been fetched by a state sequence that is common to many instructions, it is necessary to select a next machine state that is unique to one or a small class of instructions. This requires a much wider branching capability. In the KB11-A processor, this capability is provided by the fork logic. Each of the three forks generates one of a large number of possible addresses, based on the decoding of the instruction, the address modes, and various processor status indications. When a fork is enabled by the corresponding fork-enable bit of the microprogram, the address generated by the fork is loaded into the ROM address register instead of the contents of the microaddress field.

7.5.3.1 **Decode Logic** – Refer to drawing RACE. The logic illustrated on this drawing is part of fork A. This fork operates as the instruction decoder of the processor. Immediately after the instruction has been loaded into the instruction register (IR), fork A begins to generate an address. Because this address must be available within one machine cycle, fork A is designed to operate at maximum speed. Therefore, the amount of decoding is minimized; classes of instructions are recognized and the bits that differentiate members of the class are used directly as low-order bits of the generated address.

This technique can be understood by examining the address utilization by the forks. As an example, consider the selection of addresses by fork A for the group of instructions ranging from HALT to RTT. The binary op codes for all these instructions are identical except for the three least-significant bits. When the fork A decode logic recognizes that all but the three least-significant bits are 0, bit 3 of the ROM address is set, and the three least-significant bits of the op code become the three least-significant bits of the address.

7.5.3.2 **Address Bit Generation** – The logic shown on drawing RACE generates address bits for certain classes of instructions. These bits are then ORed with other signals that generate the same bits for other classes of instructions to generate the fork A address. The address is then combined with the address from the microprogram in a bit-clear operation that is explained in Paragraph 7.5.8 and shown on drawing RACL.

The signal names indicate the use of each logic circuit as follows:

a. The fork signals that are connected to the microaddress logic on drawing RACL have names that include RAB (for ROM address bit), followed by the number of the address bit to which the signal is connected.

b. In some cases, a signal is connected to more than one address bit because the same conditions generate both bits, as described in Paragraph 7.5.3.5 and shown on drawing RACL.

c. Many RAB signals are connected to the same address bit. They are distinguished by a letter that tells which fork generates the bit, and where more than one signal can be generated for the same fork. Thus, the signal RACE A0 RAB00 is one of several signals used by the fork A logic to generate bit 0 of the address.

7.5.3.3 **RACE A0 RAB ⟨02:00⟩** – RACE A0 RAB00 L, RACE A0 RAB01 L, and RACE A0 RAB02 L are used to generate microprogram addresses 001 through 007. No other fork A bits are enabled when these gates are enabled. The enabling conditions for all three signals are identical, except that each signal corresponds to a different bit of the instruction register. The IR bits passed through the AND-NOR gates are the destination mode bits.

These three signals generate addresses for a class of instructions that require destination address calculation (DAC), but no source address calculation. If the destination mode is 0, the destination data is in the destination register and no address calculation is required.

This group of microprogram words is used for the following groups of instructions:

a. All single operand instructions (with op codes of 05 or 06); this includes the instruction group from CLR to ASL (in both word and byte forms), the variable address-space moves, SXT, and XOR. These instructions are recognized by their op codes and generate the signal RACE RCLASS H.

b. The register and memory instruction group, which includes MUL, DIV, ASH, and ASHC. When one of these instructions is decoded, the signal RACE (MUL:ASHC + MFP) H is generated.

c. Any double operand instruction with a source mode of 0. Because the source data is already in the source register, it is not necessary to do the source data fetch. These instructions generate the signal RACE BIN*SM0 H.

d. The three instructions JMP, JSR, or SWAB. These three instructions use the same address calculation as the single operand instructions. The signal RACE JMP + JSR + SWAB H is generated.

7.5.3.4 RACE A0 RAB03 – RACE A0 RAB03 L is generated for the following groups of instructions:

a. Branch instructions accompanied by a bus request (BRQ); these instructions generate fork A addresses ranging from 330 to 336.

b. The six instructions which have 16-bit op codes, ranging from 000000 to 000006; these instructions range from HALT to RTT and use microprogram addresses 010 through 017 (017 is for a reserved op code that traps through location 4).

c. Any DAC class instruction (except JMP or JSR, and MFP or NEG) with a destination mode of 0; these instructions generate microprogram addresses 030 and 050 through 053.

7.5.3.5 RACE A0 RAB04 – RACE A0 RAB04 L is generated for any branch instruction. This signal is an input to bits 6 and 7 of the microprogram address, and to bit 4; as a result, all branch instructions generate fork A addresses with these three bits set (addresses between 320 and 336).

7.5.3.6 RACE A0 RAB05 – RACE A0 RAB05 L is generated for MUL, DIV, ASH, and ASHC instructions with a destination mode of 0, and for SOB instructions. RACE BIN L eliminates the binary instructions from UCLASS This RAB signal is also connected to RAB03 to generate addresses ranging from 050 to 057.

7.5.4 Fork A Circuits

The logic illustrated on drawing RACF is a part of fork A that generates microprogram addresses during instruction decoding (Paragraph 7.5.3). RACE A1 RAB00 L, RACE A1 RAB01 L, and RACE A1 RAB02 L generate the three least-significant bits of the ROM address for the classes of instructions described in the following paragraphs.

7.5.4.1 HALT Through Op Code 7 – The seven instructions are all op code (HALT through Op Code 7). These instructions generate microprogram addresses ranging from 010 to 017; the 1 in bit 3 of the address is generated by RACE A0 RAB03 L.

7.5.4.2 X Class – The X Class instructions, MARK, MFP with a destination mode of 0, and MTP, generate addresses of 074, 046, and 045, respectively. RAB02 is forced to a 1, and the two low-order bits are the complements of the corresponding bits from the instruction register. This inversion is done to permit sharing the group of microprogram addresses with the RTS through condition-code operate (CCOP) instructions; both CCOP and MARK have an op code with the least-significant octal digit a 4. Bit 5 of the address is set by RACF A2 RAB05 L.

7.5.4.3 U Class – U Class instructions include two groups: the binary instructions (with any source mode except 0, because that source mode is handled by either the DAC or the E Class groups, depending on the destination mode); and the MUL, DIV, ASH, and ASHC instructions with a destination mode of 0. The binary instructions use four microprogram addresses, one for each pair of source modes (source mode 1 is treated as a pair, although source mode 0 is handled separately). The address for each pair, except source mode 1, has a 0 in the least-significant bit and the two most-significant bits of the source mode in address bits 1 and 2. Source mode 0 has a 1 in bit 0, and the other two bits are treated in the same manner. Bit 4 of the addresses (which range from 021 to 026) is set by the signal RACH A1 RAB04 L. For the remaining four instructions, the least-significant octal digit of the op code (IR ⟨11:9⟩) generates the three least-significant bits of the address (050 through 053); bits 3 and 5 of the address are set by the signal RACE A0 RAB05 L.

7.5.4.4 RTS Through CCOP – For the instructions ranging from RTS to the condition-code operators (CCOPs), the least-significant octal digit of the op code (IR ⟨5:3⟩) generates the three least-significant address bits. The logic is complicated because a CCOP instruction may have any value between 4 and 7 in IR ⟨5:3⟩; therefore, the two least-significant bits are forced to 0 if any of these values is present. Address bit 5 is set by the signal RACF A1 RAB05 L; this signal is not generated if IR ⟨5:3⟩ contains a 1, because this is not a valid op code and the corresponding ROM address is not generated.

7.5.4.5 RACF A2 RAB03 – RACF A2 RAB03 L distinguishes between E Class instructions by generating address 030, instead of address 020, for instructions with either a bus request or a destination register 7.

7.5.4.6 TRUE 1:2 – RACF TRUE1 H and RACF TRUE2 H are used in the generation of addresses for branch instructions. These signals are mutually exclusive because they are generated for opposite polarities of IR15. In each case, if the branch condition specified by IR ⟨10:09⟩ is met, as determined by the four AND gates, the signal is asserted. Neither signal is asserted for a BR instruction, which is recognized directly because IR08 is a 0 and neither TRUE signal is asserted (Paragraph 7.5.5).

7.5.5 Fork A Logic

The logic on drawing RACH is a part of the fork A logic that generates the next microprogram address during instruction decoding.

The IR09A and IR10A flip-flops shown on this drawing are used to provide additional fanout capacity for the A fork instruction register (AFIR) (drawing RACJ), which is used to provide for the loading of the IR signals used by fork A logic. The data inputs and clocking signal for these flip-flops are identical to the signals used in the corresponding AFIR bits. The outputs of these flip-flops are distinguished from the outputs of the corresponding AFIR bits by the A following the bit number.

RACH PSWAB H is used to distinguish between the SWAB instruction and the JMP and JSR instructions; these three instructions are generally treated together.

RACH BIN*(-SM01) L is used to recognize binary instructions with source modes 2 through 7. Source mode 0 is handled as a DAC class instruction, as shown on drawing RACE.

RACH NEG.B*DM0 H is generated for a NEG or NEGB instruction with destination mode 0, because the NEG instruction is executed separately from all other single operand instructions. This signal directly generates address bit 7 by asserting RACH A0 RAB07 L, and also asserts address bit 6 by generating RACH A2 RAB00 L, which is an input to RACL RADR06 H. This signal generates ROM address 300.

RACH DF7 + BRQ H is used for E Class instructions.

RACH A2 RAB00 L is generated for floating-point instructions (address 101); branch instructions that generate the TRUE1 signal (addresses 321, 325, 331, and 335); and the NEG.B instructions (address 300), as discussed previously. This signal also generates bit 6 of the address.

RACH A1 RAB04 L is generated for binary instructions with: both source and destination modes 0 (addresses 20 and 30; any source mode except 0 (addresses 21, 22, 24, and 26); R Class instructions with destination mode 0, except MFP and the NEGB instructions (addresses 20 or 30); or SWAB instructions with a destination mode of 0 (also addresses 20 or 30).

7.5.5.1 Branch Instruction Address Generation – RACH A2 RAB02 L and RACH A2 RAB01 L are used to generate addresses for branch instructions. IR08 is used in branch instructions to negate the branch conditions specified by IR ⟨10:09⟩; when this bit is set, only branch instructions which generate neither TRUE1 nor TRUE2 succeed. Because TRUE1 and TRUE2 are mutually exclusive, there are only three possible combinations of the two signals; these three are then treated differently depending on the value of IR08, generating six possible values of RAB02 through RAB00. If IR08 is a 0 and either TRUE signal is asserted, the branch fails; if IR08 is a 1 and a TRUE signal is asserted, the branch succeeds. The result is reversed for opposite combinations. The six possibilities are further divided by the presence or absence of a bus request to form 12 separate cases. Each of these 12 cases generates a different microprogram address. This is done only to allow the fork A logic to operate at maximum speed; many of the ROM words thus addressed have common contents.

7.5.5.2 Disable BUST – After the processor fetches an instruction, a second bus transfer is usually started to fetch the word in the address following the instruction because this word is usually needed by the processor. However, if this word is not needed, a new bus cycle with a different address is started in a later machine state. If a bus cycle is started unnecessarily during the instruction decode (IRD.00) state, and the address selected is in the semiconductor memory, the memory must complete the unnecessary memory cycle before it can start any other cycle. The signal RACH DIS BUST L is used to prevent starting the unnecessary cycle in cases where it is known that the cycle is unnecessary. There are three conditions under which this signal is generated:

a. The current instruction is a double operand (BIN) instruction, and the source mode is 1, 2, or 3.

b. The current instruction is a branch or conditional branch (BR INST) instruction, and there is a bus request waiting to be honored.

c. The next machine state must begin the fetch of a destination operand in destination mode 1, 2, or 3.

7.5.6 A Fork Instruction Register

Drawing RACJ shows the A fork instruction register (AFIR), which is the second copy of the IR in the processor. The primary IR is on the IRC module and is shown on drawing IRCA. The AFIR is used to provide the fanout capability needed by the fork A logic, and to provide slightly faster operation by eliminating the signal transmission delays for signals from a register on another module.

The AFIR clock signal is generated by a logic circuit identical to the IR clock circuit. The duplication is for purposes of speed and loading. The IR is clocked whenever the IR clock microprogram bit is set and either a T1 or a BUS LOAD pulse is generated.

7.5.7 Microprogram Branch Logic

The KB11-A processor is controlled by words fetched from a microprogram ROM; each word represents a machine state. The sequence of machine states is controlled by the sequence of ROM words fetched. Normally, each ROM word contains the address of the next word to be fetched. When it is necessary to provide for

alterations in the sequence of machine states, two bits of the address contained in the current ROM word can be altered by inputs that sense processor conditions and data values. The altered bits select different addresses depending on their final values, so that up to four different addresses can be selected. This 4-way branch permits a wide variety of machine state sequences to use the same microprogram words.

In the KB11-A, the two bits that can be altered by branch conditions are bits 5 and 4 of the microprogram address. Therefore, when a branch is used, the addresses selected for different conditions differ by 20 or 40. There are 16 sets of branch conditions. One of the 16 sets is selected by the four branch-enable bits in the current microprogram word.

The outputs of the branch logic are two signals; each signal is ORed with the corresponding bit of the microprogram address from the current ROM word. Normally, when the 4-way branch is used, bits 5 and 4 of the stored address are both 0s, and the two branch signals select one of four addresses. If only a 2-way branch is desired, one of the stored address bits is set to a 1, and the corresponding branch bit is ignored, because the result of the OR is always a 1.

If all of the branch-enable bits in the current microprogram word are 0s, no branch conditions are used. The branch signals are always 0s, so the final address bits reflect only the state of the stored address. In effect, this disables the branch logic.

For 12 of the remaining 15 branch-enable values, there are two signals representing processor conditions, such as reset in progress (RIP) or step counter negative [SC05(1)], or data values such as source register negative (SR15). A 16-way multiplexer, implemented by two levels of 4-way multiplexers, selects one pair of input signals to become the branch signals, RACK BRCAB 05 L and RACK BRCAB 04 L. The first level of multiplexers is controlled by three of the four branch-enable bits; the most-significant bit selects one of two, 4-way, 2-bit multiplexers, and the two least-significant bits select one of four inputs for each of the two outputs. There are only two outputs active at a time because one of the multiplexers is disabled. The remaining branch-enable bit selects two of the four outputs from the first level multiplexers; one from each multiplexer. Only one of these can be active, so that input becomes the branch-enable bit. Table 7-13 lists the source of RACK BRCAB 05 L and RACK BRCAB 04 L.

When the branch-enable codes have a value of 3 or 15, no branch inputs are supplied (the multiplexer inputs are grounded), so no branching occurs.

When the branch-enable bits have a value of 14, the normal branch logic again produces no branch signals (the inputs to the multiplexer are grounded) but the console branch logic is enabled. This logic varies the values of address bits 7, 6, 2, 1, and 0, depending on the console operation being performed. Logic on the UBC module encodes the operation selected by the console switches, and this value selects the appropriate address. Console operations use microprogram addresses 070 through 077 and 270.

7.5.8 Microprogram Address Assembly

Refer to drawing RACL. The logic on this drawing combines the five sources of microprogram address information to generate the address of the next microprogram word. These five sources are:

a. the three forks that generate different addresses during instruction decoding and operand fetching
b. the branch inputs from the microprogram branch and console branch logic shown on drawing RACK
c. the microaddress bits from the current microprogram word

Each of the three fork inputs is controlled by a separate fork-enable bit in the current microprogram FEN field. The fork A enable is unconditional, while the fork B and C enables are conditional if certain branch-enable states exist. Only one fork is enabled at a time, and during most machine cycles all the forks are disabled. The branch

inputs, on the other hand, are never disabled; when branching is not desired, all the branch inputs are forced to a non-interferring (0) state.

Each bit of the microprogram address is generated by a negative input NOR-AND gate. The gate has four input NOR gates; one is used for each fork, and the last gate is used for the stored address and the branch inputs. All four gates must be qualified to assert the output of the AND gate; therefore, if any one of the input gates is disabled (has all inputs high), the AND gate has a 0 output.

Table 7-13
Branch Signal Sources

RACK BRCAB 05 L	RACK BRCAB 04 L	RACD UBEF INPUTS			
		03	02	01	00
Always asserted (GND09)	Always asserted (GND09)	0	0	0	0
IRCD DM357 H	GRAE SR EQ ONE L	0	0	0	1
IRCF Z2 (1) H	UBCC (PWRF + INTR) L	0	0	1	0
GRAJ SC = 0 L	GRAJ SC05 L	0	0	1	1
GRAJ DIV SUB L	IRCH N (1) H	0	1	0	0
GRAB OBD (0) H	GRAJ DIV QUIT L	0	1	0	1
DAPA BR14 L	SSRA PS RESTORE (1) H	0	1	1	0
Always asserted (GND09)	TMCB BRQ * - (T + CONF) L	0	1	1	1
RACK SYNC BRC 10 (1) H	RACK FP REQ L + RACK SYNC BRC10 (0) H	1	0	0	0
GRAJ SC = 0 L	GRAD DR00 H	1	0	0	1
TMCA CONF (1) H	TMCB BRQ TRUE L	1	0	1	0
TMCB PF(0) * (SF + TF) H	TMCB PF(0) * (SF + -TF) H	1	0	1	1
Always asserted (GND09)	Always asserted (GND09)	1	1	0	0
IRCB FJ CLASS L	IRCC 0 CLASS L	1	1	0	1
GRAD DR00 H	GRAH SR15 H	1	1	1	0
RACK SYNC BRC10 (1) H	RACK SYNC BRC10 (0) H + FRMF FP REQ WR L	1	1	1	1

When a fork is disabled, the fork-enable signal is a low, enabling the corresponding input of each AND gate. If all three forks are disabled, the corresponding three input gates on each AND gate are enabled, and the AND gate takes the state of the fourth input gate. This gate generates the OR of the stored address and any active branch inputs. If the branch inputs are all 0s (low), the address assembly logic transmits the stored address unchanged (except for the inversion performed by the NOR gate).

As a general rule, all but one of the four input gates are forced to a 1, and the AND gate follows the state of the remaining input gate. When a fork is enabled, the stored address inputs must be forced to all 1s to avoid forcing the corresponding output bit to a 0.

NOTE

All fork B addresses are between 0 and 077; therefore, in some cases where fork B is enabled, bits 6 and 7 of the stored address may be 0 without affecting the operation of the fork.

There is some interaction between the fork and branch logic. The fork B enable signal is unconditional, except when the branch-enable bits in the current microprogram word have the value 15 or 05. If the branch-enable bits have the value of 15, the fork B enable signal is generated only during the execution of F Class (floating-point) or J Class (JMP or JSR) instructions; if the branch enable is a 5, the fork B enable is generated only if the destination operand is not an odd byte. Similarly, the fork C enable is unconditional unless the branch-enable value is 14; in that case, the fork C is enabled for direct address modes (modes 2, 4, and 6) and disabled for indirect address modes (3, 5, and 7). The conditional enabling of forks B and C is usually used to permit one extra machine state before the use of the fork address. The fork is conditionally enabled with a branch condition. If the branch fails, the fork address is used, but if the branch succeeds, the fork is unconditionally enabled in the new microprogram word.

In many cases, particularly for fork A, there is more than one input signal for the same input gate. Only one of the inputs to a particular gate is active at a time, because the logic that generates the input signals has mutually exclusive conditions. The NOR gate acts as the last stage of a combinational network. The primary reason for combining this OR function with the address assembly is to preserve the speed of operation of the forks.

For more information on the fork and branch logic, and on the source of the stored microaddress, see the text accompanying the drawings listed in Table 7-14.

Table 7-14
Address Assembly Sources

Input	Drawings	Paragraph
fork A	RACE, RACF, RACH	7.5.3–7.5.5
fork B	IRCB	7.3.2
fork C	IRCC	7.3.3
branch	RACK	7.5.7
microaddress	RACD	7.5.1

7.6 TMC MODULE M8105

The Trap and Miscellaneous Control (TMC) Module M8105 provides the trap and miscellaneous control logic functions for the KB11-A. These functions include priority arbitration logic (drawings TMCA and TMCB) and most of the control logic required to execute the break conditions shown on flow diagram 12. Stack error and bus error detection logic is shown on drawings TMCC and TMCD. In addition, the logic required for numerous miscellaneous control signals used throughout the system is included on the TMC module.

7.6.1 Request Storage

The request storage register is made up of three 74S174 chips, shown on drawings TMCA and TMCB. The console flag (CONF) flip-flop (drawing TMCA) can be considered as part of the request storage register. The bus requests that are stored are listed in Table 7-15 in the order of their assigned priority. The priority arbitration logic decides which trap, program interrupt, or bus request to honor. If an abort or power-fail condition is detected, the bus request storage register will be cleared.

7.6.1.1 **BRQ Clock** – Refer to drawing TMCE. When the BRQ STROBE control signal is decoded from the current microprogram MSC field, the processor strobes all end-of-instruction requests. TMCE BRQ STROBE H is gated with TMCC STROBE INH L and TS3 to generate TMCE BRQ CLK H. This clock pulse strobes any request into the priority arbitration network.

7.6.1.2 **Priority Clear** – When an abort condition occurs, the priority arbitration storage flip-flops are cleared to ensure that only the abort will be serviced. However, if a power-fail trap is being serviced and a stack-limit-red condition occurs, the requests are not cleared. Under these conditions, the power-fail vector location is still used,

Table 7-15
Processor Service in Order of Priority

Order	Condition	Input	Output*	Result*
1	console flag	UBCF S/INST L	TMCA CONF (1) H	do console control function
2	memory management traps	SSRD MEM MGMT TRAP L	TMCB SEGT L TMCA HONOR SEGT H	trap (250)
3	warning stack violation	TMCD SL YEL	TMCA HONOR SLY H	trap (4)
4	power fail	UBCE PDNF (1) H	TMCA HONOR PWRF L	trap (24)
5	floating-point exception trap	FP EXC TRAP L FRHH	TMCA HONOR FPTRAP L	trap (224)
	CP LEV 7			
6	priority interrupt request PIRQ7	PDRD PIR15 (1) H	TMCA HONOR PIR7 L	trap (240)
7	bus request, level 7 interrupt	BUSA BR7 L	TMCA HONOR BR7 L	interrupt
	CP LEV 6			
8	priority interrupt request PIRQ6	PDRD PIR14 (1) H	TMCA HONOR PIR6 L	trap (240)
9	bus request, level 6 interrupt	BUSA BR6 L	TMCA HONOR BR6 L	interrupt
	CP LEV 5			
10	priority interrupt request PIRQ5	PDRD PIR13 (1) H	TMCA HONOR PIR5 L	trap (240)
11	bus request, level 5 interrupt	BUSA BR5 L	TMCA HONOR BR5 L	interrupt
	CP LEV 4			
12	priority interrupt request PIRQ4	PDRD PIR12 (1) H	TMCA HONOR PIR4 L	trap (240)
13	bus request, level 4 interrupt	BUSA BR4 L	TMCB HONOR BR4 L	interrupt
	CP LEV 3			
14	priority interrupt request PIRQ3	PDRD PIR11 (1) H	TMCB HONOR PIR3 L	trap (240)
	CP LEV 2			
15	priority request PIRQ2	PDRD PIR10 (1) H	TMCB HONOR PIR2 L	trap (240)
	CP LEV 1			
16	priority request PIRQ1	PDRD PIR09 (1) H	TMCB HONOR PIR1 L	trap (240)
17	T bit set and not RTT	PDRD PS04 (1) H and -(IRCD RTT L)	TMCB HONOR T L	trap (14)

* Only if no higher priority request has been received.

but the old PS and PC are pushed into locations 2 and 0. (Refer to the stack errors path of branch enable BE13, shown on the break conditions flow diagram 12.) TMCC ABORT CLR, which generates TMCC PRIORITY CLR, also sets the BLOCK STROBE flip-flop. TMCC BLOCK STROBE (1) is used to inhibit the BRQ STROBE that occurs during microstate ZAP.00 from clocking in any new requests prior to acknowledgement of the abort. Once UBCN ACKN B H occurs for all aborts other than a stack error or power fail, BLOCK STROBE is cleared to allow service of an interrupt upon completion of the abort service routine (SVC.90) prior to the fetch of the next instruction.

For stack errors and power fail aborts, BLOCK STROBE remains set until the fetch of the next instruction, blocking BRQ STROBE in both ZAP.00 and SVC.90 microstates, which prevents any requests from being strobed prior to the fetch of the next instruction. BLOCK STROBE is cleared in the FET.00 microstate by the BCT value 3 (clear flags). Thus, the BRQ STROBE in ZAP.00 is used only during the power-up, to check for the console S INST/S BUS CYCLE switch in the S INST position and the console HALT/ENABL switch in the HALT position. In this case, the console flag (CONF) is set and the processor is placed in console mode, instead of executing the power-up sequence.

7.6.1.3 Power Fail Clear – UBCE PF CLR (1) L is used to clear the priority request register upon completion of honoring a power fail to allow the next instruction to be fetched.

7.6.1.4 Internal Bus Initialization – UBCE INT BUS INIT L clears the request storage register as a function of initialization and the RESET instruction.

7.6.2 Priority Arbitration

The priority arbitration logic shown on drawings TMCA and TMCB ensures that a trap, program interrupt, or bus request will only be honored if no higher-priority request is present. The results of the priority gating are listed in Table 7-15.

The processor priority field of the processor status word [PDRD PS ⟨07:05⟩ (1) H] is decoded on TMCB to block levels so that external devices on those levels can not interrupt the processor with a request for service. For example, if the priority field contains 5, TMCB BLOCK LEV ⟨5:2⟩ L outputs are asserted. These outputs inhibit any service request below BR6 from being honored. Only external devices that have a priority higher than 5 can interrupt the current processor operation.

7.6.3 Control Logic

Refer to drawing TMCB.

7.6.3.1 **BRQ TRUE** – At least once per instruction (except an SPL), the processor, during numerous microstates (FET.00, for example), checks for the BRQ condition. If a request is to be honored, the processor will then branch to BRK.90; otherwise, it continues through the normal flow sequence. The logic that generates the BRQ condition is shown on TMCB. When any of the request-honor outputs listed in Table 7-15 go low, TMCB BRQ TRUE is asserted. This signal is used to determine Branch Enable 12 throughout the flow diagrams.

7.6.3.2 **Enable Vector** – ENB VEC (1) H is ANDed with the honored trap request to provide an output that will generate the vector address for that trap. ENB VEC is set upon entering the INTR PAUSE state, provided the processor is not servicing a bus request. The vector used during the servicing of a bus request is clocked into the BR during the INTR sequence on the Unibus. Table 7-16 lists the output asserted for each honored trap request and the ultimate trap vector that is generated.

Table 7-16
Trap Vectors Enabled

Trap Request Honored	Output	Trap Vector*
TMCA HONOR FPTRAP H	TMCB FPTRAP L	244
TMCA HONOR SEGT H	TMCB SEGT L	250
TMCA HONOR PWRF H	TMCB PWRF L	24
TMCB HONOR T H	TMCB TOK L	14
TMCB HONOR PIRQ H (OR of PIR ⟨7:1⟩)	TMCB PIRQ L	240

* Trap vector generator is shown on drawing DAPE.

When ENB VEC is set, TMCB TRAP INH L is asserted. This output inhibits instruction trap vectors from being gated into the K1MX while the non-instruction trap vector is taken. Any abort condition will also assert TMCB TRAP INH L with the TMCC BLOCK STROBE (1) H signal. ENB VEC is cleared by UBCB ACKN B H once the vector is clocked into BR.

7.6.3.3 **Branch Enable 13 (BE13)** – The logic that controls microbranch-enable BE13 is shown on TMCB. All of the errors and requests that might be honored to cause an internal trap are ORed to provide an output called TF (and its complement, -TF). The 74H50 gates provide the following two outputs: TMCB PF (0) * (SF + TF) H and TMCB PF (0) * (SF + -TF) H. These outputs control which of four microbranch paths will be followed:

a. PUPF (1) – If the power-up flag is set, neither output will be asserted. Microstate PUP.00 (100) will be entered.

b. TF – When the power-up and stack error flags are both cleared [PUPF (0) L and -SERF (1) L] and a trap condition exists, only the TMCB PF (0) * (SF + TF) H output will be asserted. This output causes microstate BRK.80 (140) to be entered.

c. -TF – When the power-up and stack error flags are both cleared and no internal trap conditions are present (-TF), only the TMCB PF (0) * (SF + -TF) H output will be asserted. This causes microstate BRK.20 (120) to be entered.

d. SF – If the stack error flag is set, SERF (1) L will assert both outputs. This will cause the SER.00 microstate (160) to be entered.

The following chart summarizes BE13 microbranch control

Condition	Output*		ROM Address
	A	B	
PUPF (1) H	0	0	100 – power up
-TF	0	1	120 – interrupts, passive release, and console continue
TF	1	0	140 – internal traps
SERF (1) H	1	1	160 – stack red errors

*A = TMCB PF (0) * (SF + TF) H
B = TMCB PF (0) * (SF + -TF) H

7.6.4 Odd Address Error

The odd address error detection logic is shown on drawing TMCC. An odd address error will be detected if an attempt is made to reference a word at an odd address. There are exceptions for certain operations where an odd address is legal during the execution of a word instruction.

When the address is odd, byte address bit DAPB BAMX00 H is high. If IRCD BY IN H is not asserted, indicating that the instruction is not a byte instruction, TMCC ODD ADRS ERR L is asserted. Thus, during any non-byte instruction, a high byte address will be detected as an odd address error.

If the ROM bus condition field is decoded to indicate the bus transaction is not BSOP1 or BSOP2, and is not SRC1 DATI or SRC2 DATI, then it must be either DATI (BSC 0), kernel DATI (BSC 2), FC (BSC 4), or DATO (BSC 5). Under any of these conditions, a high byte address will produce an ODD ADRS ERR, even during byte instructions. During DATI or SRC1 DATI with SM357, an odd address error is detected if BAMX00 is high. An odd address is legal only during BSOP1, BSOP2, SRC2 DATI, or SRC1 DATI* – SM357.

7.6.5 Fatal Stack Violation

The SL RED signal is asserted when a fatal stack violation (red) is detected. The stack limit protection logic is shown on drawing TMCD. The three conditions that will cause the TMCD SL RED L error signal to be asserted are shown in Figure 7-7.

7.6.5.1 Red Zone or Stack Limit Violation – The high byte of the virtual address of all stack-pointer-related DATO, DATOB, and DATIP operations performed in kernel mode is compared with the contents of the SL (stack limit) register. If the high byte is less than the contents of the SL, a fatal stack violation is detected. Under these conditions, PDRC RED ZONE H is asserted. If the stack error flag is not already set, or the BLOCK STROBE flip-flop is not set, (Paragraph 7.6.7.4), TMCD SL RED L will be asserted when timing pulse TIGD T5 H goes high. This is done to prevent red zone error detection while the emergency stack (locations 2 and 0) is being used during the service of a red zone stack error. A latching gate keeps TMCD SL RED L low until the stack error service routine is completed and UBCB ABORT ACKN L goes low.

If the high-order virtual address is equal to the SL contents, PDRC STACK LIMIT H is asserted. If the BAMX ⟨07:05⟩ H bits are not equal to 7 (16 words), the virtual address is below the 16-word yellow zone boundary. The -TMCD YEL ZONE H level is ANDed with PDRC STACK LIMIT H to qualify that input, and TMCD SL RED L will be asserted when timing pulse TIGD T5 H goes high.

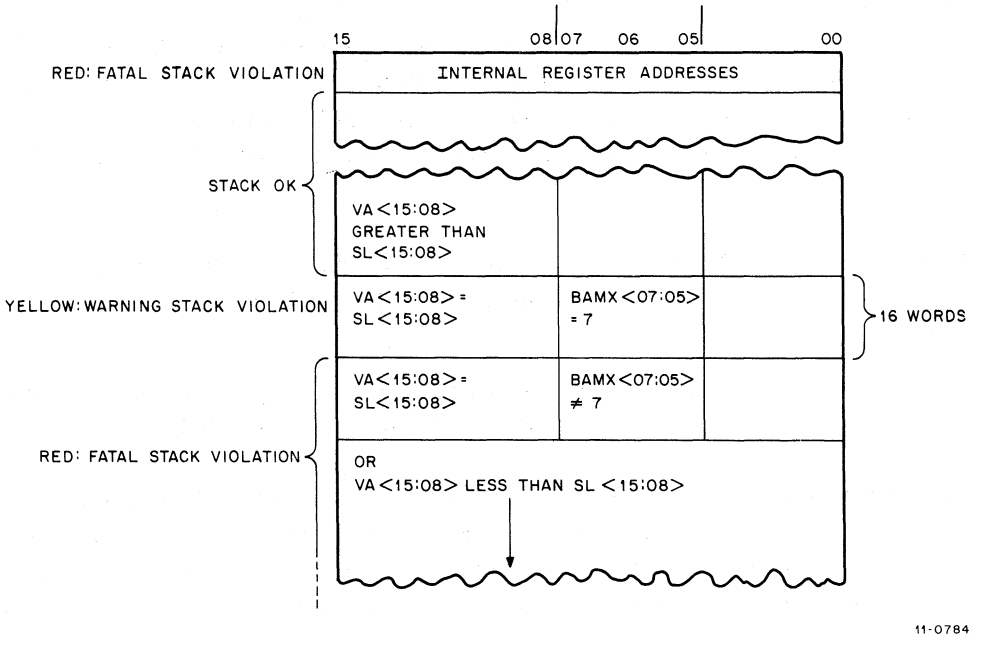


Figure 7-7 Red and Yellow Stack Violations

7.6.5.2 **Internal Address Violation** – When the address of any internal register is decoded from the PA ⟨17:09⟩ H and BAMX ⟨08:01⟩ H inputs to the 74155 chip, TMCD INTERNAL ADRS H is asserted. This signal is used to assert TMCD SL RED L (Figure 7-7). This protects the internal register locations from being inadvertently included in the stack.

7.6.6 Warning Stack Violation

The yellow warning stack violation detection logic is shown on TMCD. This stack limit protection logic is only enabled during stack-pointer-related DATO, DATOB, and DATIP bus operations, as is the case for SL RED. Under these conditions, TMCC KERNEL R6 (1) H and RACB UBSD01 H are both high (Figure 7-7). If the high-order byte of the virtual address VA ⟨15:08⟩ is equal to the contents of the SL, PDRC STACK LIMIT H is high. If BAMX ⟨07:05⟩ H is 7, the address is within the 16-word yellow zone. As a result, the SL YEL flip-flop is set at T2.

7.6.7 Abort Detection

The five conditions that cause an immediate abort are:

a. odd address error
b. fatal stack violation (red)
c. memory management abort
d. timeout
e. parity error

The logic that detects odd address errors and fatal stack violations is located on the TMC module and described in preceding paragraphs. Parity and timeout error detection logic is located on the UBC module and described in Paragraphs 7.7.7 and 7.7.2.9. The memory management abort logic is located on the SAP and SSR modules and described in the KT11-C Memory Management Unit Maintenance Manual.

The five abort-causing inputs are ORed, as shown on TMCC, to assert TMCC ABORT H. Instruction execution is interrupted at T2 of the next pause cycle if any of these inputs is asserted. Any of the five conditions that assert TMCC ABORT H will also prevent the MSYN flip-flop on UBCB from being direct-set and inhibit the CONTROL OK from semiconductor memory. The TMCC ABORT H level is applied to UBCA to assert UBCA SSYN RESTART H after the address deskew delay. At the same time, TMCC ABORT H generates ROM address 200, ZAP.00.

7.6.7.1 **KERNEL R6** – The KERNEL R6 flip-flop detects that a reference to the kernel's stack is being made and used to enable the stack limit error detection logic. All DATO, DATOB, and DATIP references are gated against the following conditions:

a. The PAD address being referenced is 6 and the SR or DR is being used as the input to the BAMX.
b. The ROM bus control code indicates a stack reference during a JSR (BCT = 5).

7.6.7.2 **Address Error Flag (AERF)** – The purpose of the address error flag (AERF) flip-flop is to test for any type of address error, including odd address error, Unibus timeout, or memory management aborts. Odd address, timeout, and memory management abort signals are ORed. If any of these types of address errors are detected, or the CNSL ACT and KERNEL R6 flip-flops are not set, the AERF will be direct-set at time state TS2 of the pause or long pause cycle.

7.6.7.3 **Stack Error Flag (SERF)** – The SERF flip-flop is set when any stack error is detected. Red fatal stack violations (SL RED) and yellow warning stack violations (SL YEL) are two stack error conditions. In addition, memory management abort conditions or bus errors detected during the pause cycle of a kernel R6 reference will also set SERF. Red fatal stack violations, memory management aborts, or bus errors will set SERF immediately at time state TS2 of the current pause cycle, when TMCC CLK FLAGS H goes high. These conditions demand immediate stack error service. If a yellow warning stack violation is detected, the SERF will not be set until the current instruction is completed; then TMCA HONOR SLY H and UBCB ACKN B H will clock and set SERF. SERF (1) H is one of the inputs that control the BE13 microbranch-enable logic, directing the processor to the stack errors path if a red error is detected. For yellow errors, it prevents any BRQ STROBE from clocking in the new request until after the next instruction is fetched.

7.6.7.4 **Block Strobe** – BLOCK STROBE is used to inhibit BRQ STROBE from strobing any requests if an abort condition occurs. It is used with PRIORITY CLR to prevent any trap vectors from being asserted or any traps from being honored while an abort condition is being serviced. It prevents the BRQ STROBE from occurring in either microstate ZAP.00 or in SVC.90. The reason BRQ STROBE is inhibited during ZAP.00 is to prevent requests from being strobed into the priority arbitration logic after PRIORITY CLR has cleared the storage register for service of an abort. The reason BRQ STROBE is inhibited during microstate SVC.90 is to disable the servicing of any requests once a stack error or power fail has occurred until after the execution of the next instruction.

If a fatal stack error occurs during the execution of a power-fail sequence, the PS and PC associated with the fatal stack violation are saved in locations 0 and 2. A new PS and PC are established as determined by the power-fail vector at location 24. The power-fail trap vector (24) is generated because TMCA HONOR PWRF H and SERF (1) H assert TMCB PWRF L.

7.6.8 Internal Address Decoder

Refer to drawing TMCD. A 74155 Dual 2-Line-to-4-Line Decoder is used to decode internal register addresses from PA ⟨17:06⟩ H and BAMX ⟨05:01⟩ H. For any internal register address (77777X), the A1 input goes low

when UBCA CPBSY B H goes high. The A0 input will not be qualified because BAMX 07 H is high. DAPB BAMX ⟨02:01⟩ H are applied to the select inputs. Depending upon these two low-order address bits, the A1 input will be demultiplexed to one of four outputs as indicated in the following chart.

Register Address	DAPB BAMX		Output Selected	Function Asserted
	02	01		
777770	L	L	1F0	PB ADRS L
777772	L	H	1F1	PIR ADRS L
777774	H	L	1F2	SL ADRS L
777776	H	H	1F3	ST ADRS L

If the switch register address (777570) is decoded, DAPC BAMX 07 H will be low. Under these conditions, if TMCE GET OFF H is not asserted, a high level will be applied at the A0 input and the A1 input will also be high (disqualified). The low-order bits DAPB BAMX ⟨02:01⟩ H will be low and the 2F0 output at pin 7, TMCD SW ADRSL, will be asserted. TMCD SW ADRS H is gated with SSRK KT11C FAST (1) L to inhibit switch addresses from the internal bus during KT11-C register operations, because the physical address lines can change while KT11-C registers are being addressed.

7.6.9 DMX Select

Refer to drawing TMCD. The decoded internal address outputs are used to provide TMCD DMX S1:S0 H, which select the appropriate register to be gated onto the Unibus during a DATI or DATIP transaction, as shown on drawing PDRE. The selection lines are used with TMCD HI BYTE EN H and TMCD LO BYTE EN H to multiplex five registers onto the Unibus, as indicated in the following chart.

TMCD DMX		Register
S1	S0	
L	L	BR
L	H	SL (HI BYTE EN only)
L	H	PB (LOW BYTE EN only)
H	L	PIR
H	H	PS

7.6.10 Bus Condition Multiplexer

Refer to drawing TMCE. The bus condition multiplexer decodes the class of the current instruction and the Unibus condition-code field of the ROM to provide TMCE C1 H and TMCE C0 H. These outputs are used to generate Unibus control signals C1 and C0, as shown on drawing UBCC. BSCMX is a 74S153 multiplexer shown on drawing TMCE. RACC UBSC ⟨02:00⟩ H are used to generate TMCE ⟨C1:C0⟩ H as shown in the following chart.

BSC Field	TMCE		Conditions
	C1	C0	
0–3	0	0	BSC = 0–3 are all DATI operations. The bus condition multiplexer is not enabled.
4	1	0	DATO if FPC1 = 1
	0	0	DATI if FPC1 = 0
5	1	0	BSC = 5 is always DATO and C1 is forced to 1.
6	0	1	P Class asserts C0 for DATIP.
	1	0	O Class asserts C1 for DATO.
7	1	0	BSC = 7 always forces C1 to 1 for DATO.
	1	1	BYIN asserts C0 for DATOB.

7.6.11 Miscellaneous Control and Bus Delay Signals

The miscellaneous control signals generated by the combinational logic shown on drawing TMCE are used throughout the TMC module and are issued to other modules in the processor, memory management unit, and floating-point processor. The RACC UMSC ⟨02:00⟩ outputs of the ROM MSC field are clocked out at time state

T1 and the appropriate inputs will be available when TMCC TS3 CLKA H goes high. The logic is straightforward and the conditions required to assert each miscellaneous control signal output can easily be determined by inspection of input signal mnemonics. Table 7-17 lists the functions of the control and bus delay signals generated by the logic shown on drawing TMCE.

7.6.12 Internal Bus Signals

The IBS field is decoded from the ROM to provide RACB UIBS ⟨01:00⟩, which control the internal processor bus. If either of these bits are high, the internal bus is required for internal processor operation and TMCF GET OFF H will be asserted. This prevents external devices such as the memory management unit from placing data on the internal bus.

When the IBS field is 1, and UBCJ DDC STOP L is asserted and TMCF DDC ATTN L is sent to the DEC Data Center (DDC) terminal. This enables the DDC to put its address onto the Unibus D lines to be strobed into the BR. If DDC STOP L is high, TMCF READ SW L will be asserted. This signal gates SWR ⟨15:00⟩ input from the console switch register to the internal bus.

7.6.13 Bus Register Multiplexer Control

The logic for the BRMX select is generated on drawing TMCF. Because the internal selection is a function of the address presently being asserted, there are cases where the bus input is needed and the address is not known. These cases occur during the INTR transaction on the Unibus and during DDC transactions, when the address is asserted on the Unibus. Thus, during INTR PAUSE or DDC ATTN, TMCF SELINT L and TMCF SELMEM L signals are inhibited. The SELMEM signal is also inhibited when an internal address or internal bus command is issued. The bus register multiplexer is shown on drawing PDRA.

Table 7-17
TMCE Control and Bus Delay Signal Functions

Signal	Function
Miscellaneous Control Signals	
TMCE BRQ CLK	Strobes end-of-instruction requests into the priority arbitration logic shown on TMCA and TMCB.
TMCE BEND CLR	Stops Unibus and Fastbus memory cycles when further address calculation is requred.
TMCE SET PRIORITY	Enables loading BR⟨2:0⟩ into PS⟨7:5⟩ for SPL (Set Priority Level) instruction.
TMCE BUST OUT	Generated by ROM to start bus cycle.
TMCE FLOATING ATTN	Signals FPP that, address, instruction or data is ready to be taken.
TMCE SET CONF	Sets CONF if HALT in kernel mode, parity error, or console reset occurs.
Bus Delay Signals	
TMCE BUS LONG PAUSE	Indicates second half of bus cycle and that address lines will change immediately upon exiting present ROM state. Stops timing in T5 of Unibus operation to allow address deskew to be completed.
TMCE INTR PAUSE+	Signifies beginning of service routine. Enables bus grants to occur or trap vector to be enabled.
TMCE PAUSES	Indicates second half of bus cycle. Enables signals such as MSYN and CONTROL OK. Causes timing generator to wait for SSYN at T2 of Unibus cycle or at T5 of Fastbus cycle.
TMCE INTR CLR	Clears CP BSY if bus operation is aborted and notifies FPP that CPU is servicing interrupt.

7.7 UBC MODULE M8106

The Unibus and Console Control (UBC) Module M8106 provides Unibus, Fastbus, and console control logic functions for the KB11-A processor. The Unibus control functions are compatible with all Unibus devices. Complete descriptions of the Unibus are provided in the *PDP-11 UNIBUS Interface Manual.* A short summary of Unibus transactions is provided in Appendix D of the *PDP-11 Handbook.* These manuals provide general overall descriptions of Unibus operations. The UBC module also contains control logic for the Fastbus interface with floating-point and high-speed metal-oxide semiconductor (MOS) and bipolar memory.

7.7.1 Bus Control Introduction

When the KB11-A processor requires a memory reference to fetch or execute an instruction, the Unibus and Fastbus control sequences are initiated. Fastbus devices will decode the address lines while Unibus control is being obtained and before the 150-ns deskew delay is completed. If the address applies to a Fastbus device, that device will respond in time to inhibit the Unibus MSYN signal and a Fastbus control sequence will occur. The Fastbus control sequence is described in Paragraphs 7.7.4 through 7.7.6. If no Fastbus device responds before deskew is completed, the Unibus MSYN signal is asserted and the Unibus control sequence continues. The Unibus control sequence is described in Paragraph 7.7.2. All KB11-A memory references consist of BUST and PAUSE cycles.

7.7.1.1 **BUST (Bus Start) Cycle** – When BUST is decoded from the ROM MSC field of a microstate, address lines are asserted and a memory management delay is provided. An attempt to gain control of the Unibus is initiated and when, if successful, an address deskew delay is initiated. During this cycle, the processor and memory management logic tests for odd address and page address errors and fatal stack limit violations.

7.7.1.2 **PAUSE Cycle** – Errors are acted upon during the PAUSE cycle. If an error has been detected, MSYN (Unibus) and CONTROL OK (Fastbus) will not be asserted. The error condition restarts the timing generator and forces the ROM to ZAP.00 (200). If no errors are detected, MSYN or CONTROL OK is issued and the data transfer occurs. The two types of PAUSE cycles are bus pause and bus long pause; the type is determined by the ROM BSD field for each machine state.

a. **bus pause**: If the address and C lines are to remain the same upon entering the next ROM machine state, bus pause (BSD = 2) is specified because the Unibus address deskew can be completed during the next ROM state.

b. **bus long pause**: If the address and C lines are to be changed upon entering the next ROM state, bus long pause (BSD = 3) is specified. Under these conditions, all Unibus address deskew delays are completed before leaving the current ROM state (T5 restart).

7.7.1.3 **Unibus Control** – BUST is decoded from the ROM at T1. A delay is provided when the KT11-C Memory Management Unit option is implemented. During this time, TIGA STOP T3 L is asserted. During this delay, the processor asserts the address lines and control lines.

7.7.2 DATI and DATIP Unibus Transactions

The UBC logic that initiates and controls DATI and DATIP Unibus transactions is described in the following paragraphs. The descriptions are presented to follow DATI and DATIP sequences. Much of the logic is common to DATO and DATOB Unibus transactions, which are described in Paragraph 7.7.3.

ECO KB11-A No. 13 ("Speed-up ECO"), in conjunction with Revision C or higher of the PDR Module (M8104), has changed the data transfer operations. Explanations of both versions are presented in this paragraph.

7.7.2.1 **CPBSY** – TMCE BUST H and TIGD T3 B H set GET BUS. When all current NPR requests have been honored, UBCA UNIBUS RELEASE H will be high. UBCA UNIBUS RELEASE = –(D SACK + NPR + NPG + BBSY). Under these conditions, GET BUS sets CPBSY. The processor now has control of the Unibus and asserts BUSA BBSY L.

7.7.2.2 **Address Deskew** – CPBSY and GET BUS will assert UBCA DESKEW ADRS H when SSYN from the previous Unibus transaction is negated (–SSYN) and the current Unibus transaction is a DATO or DATOB, or immediately if the transaction is DATI or DATIP.

The deskew register shown on drawing UBCB consists of six D-type flip-flops provided by a 74S174 IC. The UBCA DESKEW ADRS H signal is applied to the D5 input. The flip-flops are clocked by TIGC TF L. The period of TIGC TFL is 30 ns. The UBCA DESKEW ADRS H input is propagated through the deskew register by five successive clock pulses so that after a 150-ns delay, UBCB DESKEW COMP H is asserted.

7.7.2.3 **MSYN** – The delay provided by the deskew register allows for worst-case signal skew and allows time for internal logic in slave devices to decode the address. UBCC SSYN B L assures completion of the previous Unibus transactions. (For DATI transactions, address deskew may be completed prior to the removal of SSYN from the preceding bus cycle.) As soon as UBCB DESKEW COMP H is asserted, the MSYN flip-flop will be direct set, if no errors are detected.

The MSYN flip-flop and error condition gates are shown on UBCB. TMCE PAUSES B H and UBCA TS2 CLK H must be asserted. MSYN will be set if none of the following errors has occurred:

a. odd address error
b. memory management abort
c. stack limit red
d. parity error

If any of the above errors is detected, MSYN cannot be set. An error condition restarts the timing generator and forces the ROM to ZAP.00. Also, MSYN cannot be set if the Fastbus is in use (TMCF FAST L must be high). As previously stated, when a Fastbus device is addressed, that device will assert TMCF FAST L and inhibit MSYN.

The conditions that cause the MSYN flip-flop to be cleared are determined by the type of bus transaction that is in progress. Typically, for DATO bus transactions, MSYN is cleared when the SSYN signal is received from the slave device. For DATI and DATIP transactions, MSYN is cleared by the BUS LOAD signal (if bus long pause), at T1 of the next ROM cycle (if bus pause), or at T3 if KB11-A ECO No. 13 is in the CPU. These conditions are described in the following paragraphs. If a nonexistent memory is addressed, MSYN will be cleared by UBCB TIMEOUT H.

7.7.2.4 **Bus Pause and DATI or DATIP, Early Units** –

a. *Bus Pause and DATI* – In the pause cycle (UBSD01 H) of a Unibus transaction (–TMCF FAST L), the TIGA STOP T3 L signal is asserted to stop the timing generator and wait for the slave device to respond with SSYN (Figure 7-8A). When the slave device receives MSYN, it completes a read cycle, outputs data on the D lines, and asserts SSYN. When the processor receives BUS SSYN L, it asserts UBCC SSYN B H.

   Refer to UBCA. UBCC SSYN B H is ANDed with UBCB MSYN (1) H to provide UBCA SSYN RESTART H, which restarts the timing generator by inhibiting TIGA STOP T3 L. As the timing generator advances to time state TS1, TIGB T1 B L asserts UBCB CLK BR H to clock the data from the slave device into the BR. The ROM state BRK bit must be a 1 to allow the BR CLK to be asserted. The TIGB TS1 L signal asserted at this time is used as the clock to clear the MSYN flip-flop. CPBSY is cleared on the following T4.

b. *Bus Long Pause and DATI or DATIP* – During a bus long pause DATI or DATIP Unibus transaction, the timing generator is restarted when BUSA SSYN L is received from the slave. However, because a bus long pause is in effect, TIGA STOP T1 L is asserted. RACB UBSD 00 H and TIGA UBSD 01 H are high until UBCA BLP DESKEW is true, at which time the TIG resynchronizes and issues T1, which clears CPBSY and removes the A and C lines. The purpose is to allow an additional 75 ns for address deskew after MSYN is cleared during a bus long pause Unibus transaction. The sequence is described in the following paragraphs and shown in the DATI bus timing diagram (Figure 7-8B).

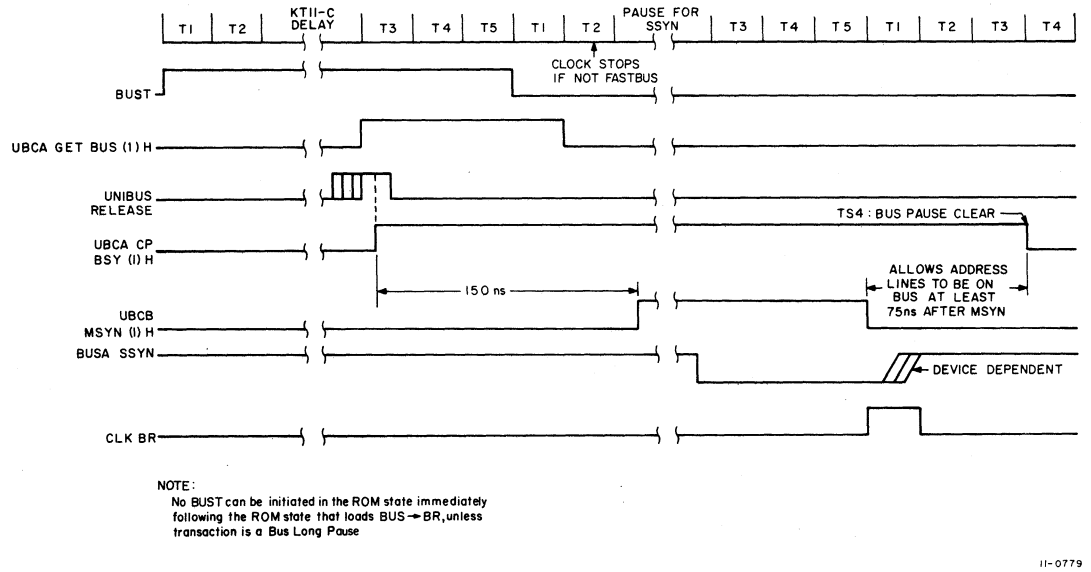


Figure 7-8A DATI Unibus Timing Diagram

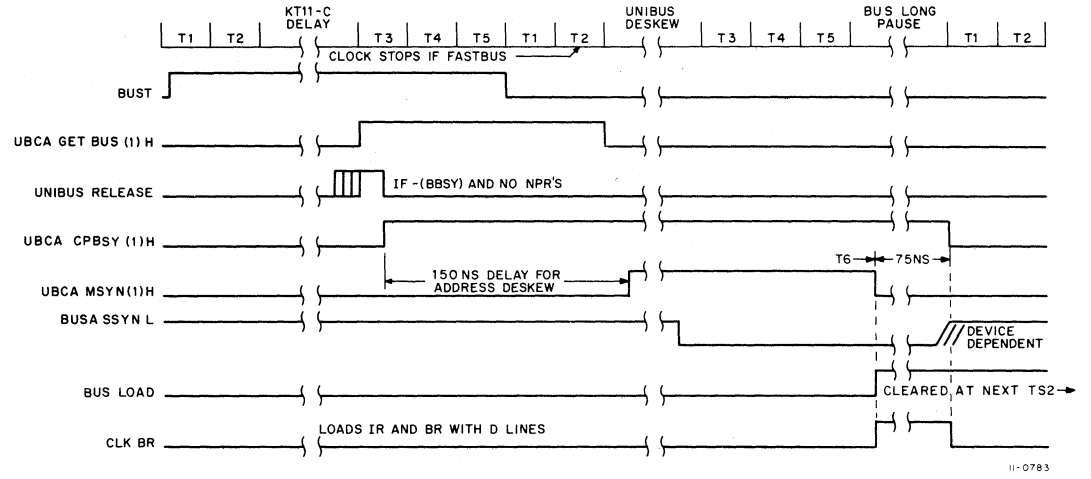


Figure 7-8B Unibus DATI Bus Long Pause Cycle, Control Timing

BUS LOAD – During a DATI or DATIP with a bus long pause, MSYN remains set until the BUS LOAD flip-flop is set. Refer to UBCB. BUS LOAD is set one clock pulse after T5 (GRAC T6 L) to allow completion of write-to-scratchpad for the previous cycle before the IR and BR are loaded with new input data. UBCB BUS LOAD H direct-clears MSYN. When this happens, all inputs are qualified to assert UBCA BLP DESKEW H.

UBCB BUS LOAD L asserts UBC CLK BR H at this time, provided the ROM state BRK bit is 1.

BLP DESKEW – The purpose of BLP DESKEW is to allow 75 ns for address deskew before address lines are removed from the Unibus. The logic that initiates bus long pause deskew timing is shown on drawing TIGA. With TMCE BUS LONG PAUSE H asserted, the signal UBCA BLP DESKEW H will be asserted as soon as the MSYN flip-flop is cleared, because GET BUS is cleared and CPBSY is still set. The UBCA BLP DESKEW H signal is used on TIGA to disable STOP T1 and restart the timing generator. As a result, a 75- to 100-ns delay is provided. This period of time is allowed for address deskew before timing restarts because the A and C lines will change upon exiting the pause cycle. When UBCA BLP DESKEW H is asserted, the next T1 time state sets BLP CLR, which clocks and clears CPBSY (drawing UBCA).

BUSA BBSY L – The BUSA BBSY L signal is asserted when the CPBSY flip-flop is set, as shown on drawing UBCA. While the BUSA BBSY signal is asserted, the UNIBUS RELEASE H signal is inhibited. During the bus long pause cycle of a bus transaction (other than DATIP), the following T1 clears CPBSY and allows UBCA UNIBUS RELEASE to be asserted. During a bus pause, CPBSY is cleared after the timing generator is restarted when TIGB TS4 L causes UBCA TS4 CLK H of the next ROM state to be asserted. The removal of BUSA BBSY from the Unibus allows a previously granted device to take control.

7.7.2.5 Bus Pause and DATI or DATIP in Later Units – The following description applies to machines that incorporate ECO KB11-A No. 13 and Revision C of the PDR module M8104. This ECO effectively eliminates the Bus Long Pause and its deskew in T5 at the end of the bus cycle (Figure 7-9).

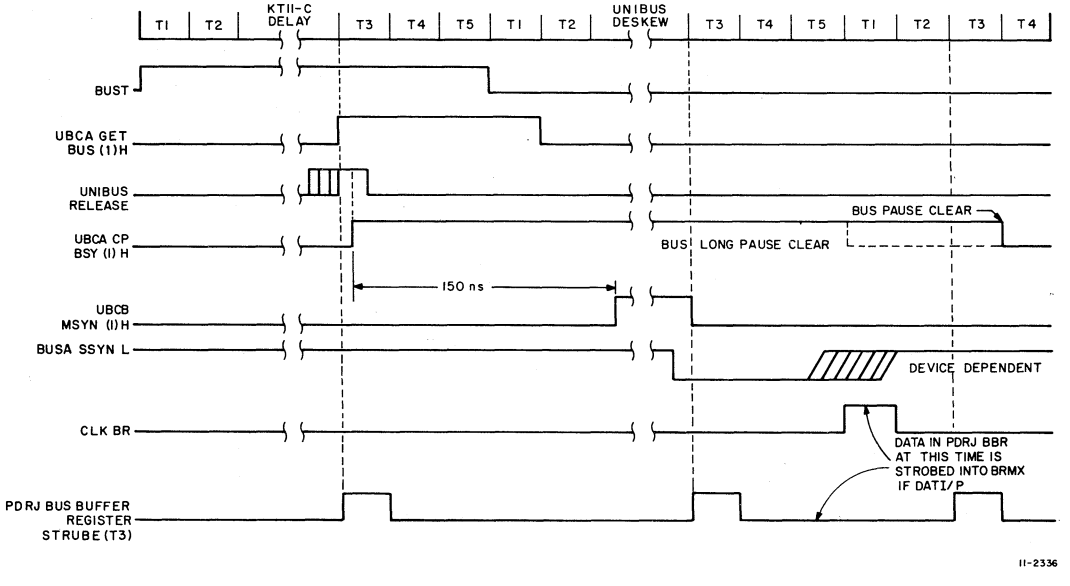


Figure 7-9 Unibus Timing Diagram

TMCE BUST H is asserted at T1. T3 does not occur until the end of the KT11-C delay, at which time UBCA GET BUS is set. If there is no other bus operation, UBCA UNIBUS RELEASE H is asserted, and UBCA CPBSY is set. This starts the address deskew, at the end of which UBCB MSYN is set. The CPU now waits for BUSA SSYN L from the Unibus; when it occurs, UBCC SSYN B L [in conjunction with UBCB MSYN (1) H] generates UBCA SSYN RESTART H. This, in turn, starts the synchronizer on TIGA.

A minimum of 75 ns is required to restart the clock at T3; this allows for data deskew. This same T3 strobes the bus data into the PDRJ Bus Buffer Register and resets UBCB MSYN. At the next T1, if the flows call for a Bus Long Pause, or at the T4 after that, if a Bus Pause, UBCA CPBSY, is reset. This is the only difference between a Pause and a Long Pause in the newer KB11-As. The time from the end of MSYN to the T1 following it allows for address deskew.

7.7.2.6 **TIMEOUT** – The TIMEOUT logic shown on drawing UBCB terminates the Unibus transaction if an SSYN response is not received within 10 or 5 μs after the MSYN flip-flop is set.

The UBCB MSYN SET H signal, which is asserted when MSYN sets, is applied to the D input of the TIMEOUT flip-flop. At the same time, it is applied to the 74123 one-shot. If the 74123 is not direct-cleared within 10 or 5 μs, the output will clock the TIMEOUT flip-flop. If the slave device returns the SSYN signal within a normal time, UBCC SSYN B L will clear the one-shot and prevent the TIMEOUT flip-flop from being set.

If BUSA SSYN L is not received within 10 or 5 μs, the TIMEOUT flip-flop gets set and a trap to location 4 occurs. Once the trap vector has been clocked into the DR, the next ROM state BCT field asserts UBCB ACKN L, which clears TIMEOUT.

7.7.3 DATO and DATOB Unibus Transactions

7.7.3.1 **Early Machines** – Initial Unibus control functions for DATO and DATOB transactions are identical to those described for DATI and DATIP. The sequence is initiated by BUST at TS1. Refer to the DATO/DATOB timing diagram in Figure 7-10. During the KT11-C delay, the Unibus address and control lines are asserted. After the KT11-C delay, the timing generator starts and at TS3, TMCE BUST OUT initiates the processor's attempt to get control of the Unibus by setting GET BUS. If the Unibus is already busy, there will be a delay until UNIBUS RELEASE H is asserted before CPBSY will set. When BUSA SSYN L from the previous Unibus transaction is negated, UBCA DESKEW ADRS H is asserted. The deskew delay lasts for 150 ns. During this time, the Unibus address, control and data lines are asserted. If the address is that of a Fastbus device, or internal register, or if a bus error or page address error occurs, MSYN cannot be set. Otherwise, MSYN is set 150 ns after CPBSY.

The addressed slave device receives the control and MSYN signals, strobes in the word (or byte) of data, and asserts BUS SSYN L. When the processor receives BUSA SSYN L, it asserts UBCC SSYN B L, which in turn asserts UBCA SSYN RESTART H to restart the timing generator.

UBCC C1 B H is asserted for DATO or DATOB. Therefore, when TIGB TS4 L goes low, UBCC MSYN CLR H is asserted; this clears MSYN. Because the timing generator must resynchronize when BLP DESKEW is asserted, three time periods (90 ns) will elapse before T1 is issued. This allows the slave device deskew time before the address, data, and control lines will be removed. When the slave device sees MSYN fall, it drops SSYN.

7.7.3.2 **DATO and DATOB with ECO KB11-A No. 13** – The DATO and DATOB operation in recent systems is identical to that of DATI, which is explained in Paragraph 7.7.2.5, with the exception that the data strobed into the PDRJ Bus Buffer Register is not used.

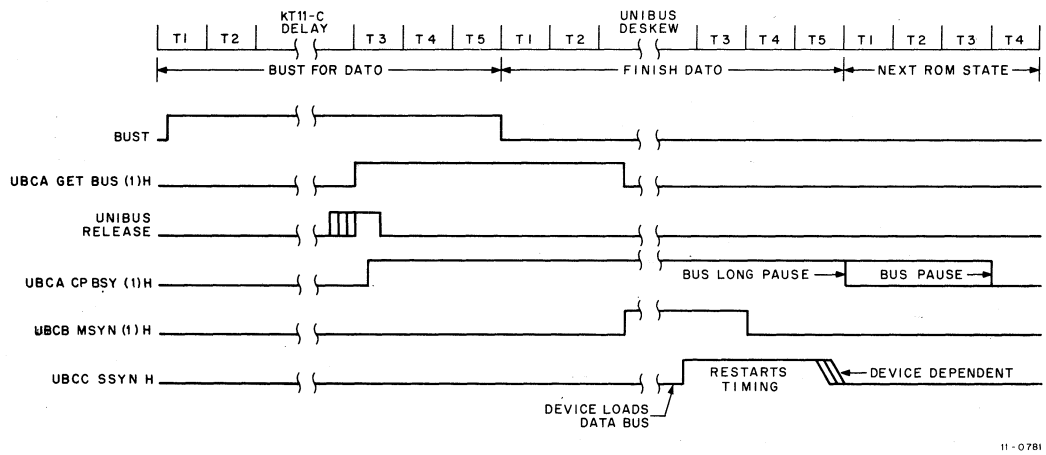


Figure 7-10 DATO and DATOB Timing Diagram

7.7.4 Fastbus Transactions

Fastbus transactions are initiated in the same manner as Unibus transactions. When BUST is decoded from the MSC field of the ROM for a particular microstate, STOP T3 L is asserted for a KT11-C delay. During this time, the Fastbus address and control lines are asserted. If the address is that of a Fastbus device (such as semiconductor memory, KT11-C, or an internal register), TMCF FAST L asserts UBCB FAST H, the signal UBCB NOT UNI L asserts UBCB CLR UNI H, and CPBSY is cleared at T2. The Unibus control sequence is terminated and TMCF FAST L prevents MSYN from being set.

UBCB CLR DESKEW L is used to restart A + D + C deskew for special-case DATO transactions. For semiconductor memory storage, the data need not be present in the BR until CONTROL OK is issued. However, for Unibus operations, the data must be present prior to the start of the 150-ns address deskew. For certain DATO operations (SHR.10, EXC.00), the data is not loaded until T1 of the pause cycle; the Unibus deskew must not be restarted. Therefore, for BSOP2 (DATO and DATOB) and BSOP1 DATO operations, if the BRK ROM bit was true in the previous cycle, UBCB CLR DESKEW L is asserted at T2 and the deskew is restarted.

7.7.5 Fastbus DATI and DATIP

After the KT11-C delay, the timing generator advances to TS3 and TMCE BUST OUT L is asserted. Refer to the timing diagram shown in Figure 7-11. Assume that the address was located in a semiconductor memory. During the remainder of the BUST cycle, the semiconductor memory control decodes the address and responds with SMCF MEM L. When the processor receives this input, it asserts TMCF MEM H and TMCF FAST L.

The processor enters the pause cycle and checks for errors. If no bus errors, stack limit red errors, or parity errors are detected, TMCE BEND ERR L will be high. If no page address abort condition is detected by the KT11-C logic, SSRC KT11C ABORT FLG L is asserted (drawing UBCA). As a result, CONTROL OK is set at T3 of the pause cycle. The CONTROL OK output indicates to the semiconductor memory control that the control lines are stable. The signal is used to latch decoders in the semiconductor memory control. The memory proceeds to do a read operation and then asserts the SMCA MEM SYNC (B) L, indicating that the data is ready. TIGA STOP T1 L is asserted until SMCA MEM SYNC (B) L is received. The timing generator is suspended in TS5. When SMCA MEM SYNC (B) L is asserted, the timing generator restarts. TIGB T1 L clears CONTROL OK. When it does, the semiconductor memory control logic drops SMCA MEM SYNC (B) L.

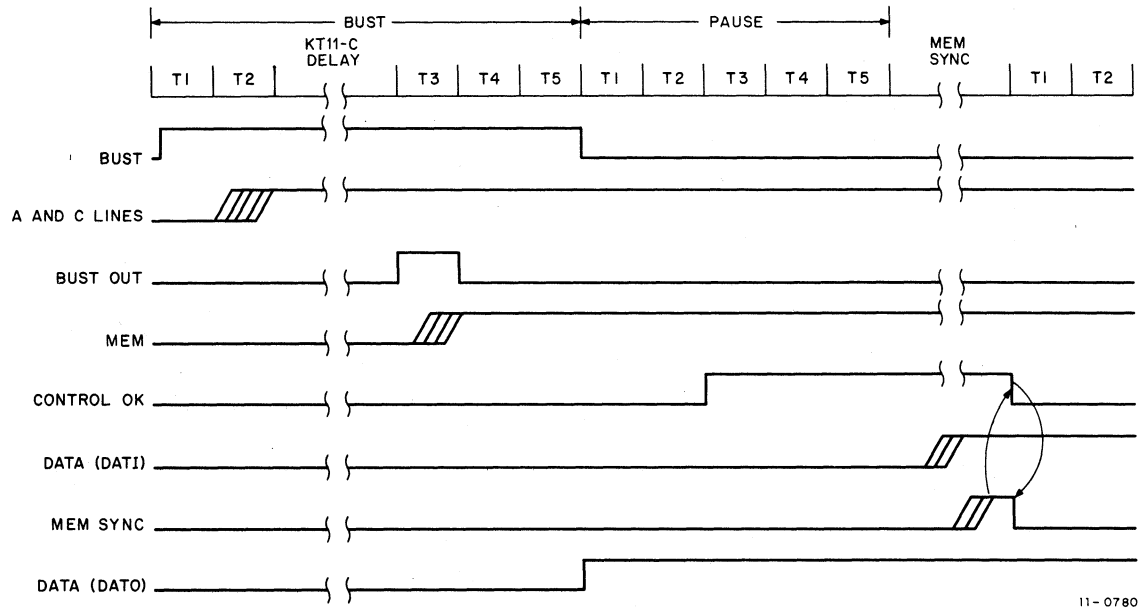


Figure 7-11 DATI and DATO Fastbus Control

7.7.6 Fastbus DATO and DATOB

The DATO cycle for Fastbus transactions is similar to the DATI cycle; the difference is that data lines are asserted at time T1 of the ROM state with the BRK bit set to 1. SMCA MEM SYNC (B) L is asserted by the memory control logic as soon as the data is taken. If this happens prior to TS5 of the pause cycle, TIGA STOP T1 L will not be asserted and the processor timing generator is not suspended in TS5.

7.7.7 Parity Error Logic

A parity error on Unibus A is indicated by BUSA PA L high and BUSA PB L low. This condition causes the UNIPERF flip-flop on UBCB to set when MSYN is cleared. A semiconductor memory parity error causes SMCB PERF L to set another flip-flop (E87 pins 8 and 9) on UBCB. The output of this flip-flop and that of UNIPERF are ORed to assert UBCB PARITY ERR H.

At the end of the PAUSE ROM cycle, both UBCB PARITY ERR L and UBCB PE ABORT L are asserted. The first of these signals generates trap vector 114 on DAPE. The second one asserts RACA ZAP L at the next TS2; it also clears the TMCA priority network through TMCC ABORT CLR L and TMCC PRIORITY CLR L, if no stack limit error is pending (TMCD SL RED L). This abort occurs one ROM state after the PAUSE ROM state. Note that all other aborts occur directly after the PAUSE state.

UBCB ABORT ACK H clears both UNIPERF and the semiconductor memory error flip-flops.

7.7.8 NPR and NPG

Refer to drawing UBCD, which shows the NPR (non-processor request) and NPG (non-processor grant) logic. Any Unibus A device can assert the BUSA NPR L input, which takes precedence over any processor request for use of the Unibus. UBCC DATIP L and UBCE ACL0 B L inhibit NPRs. DATIP is used to inhibit setting the NPR flip-flop to improve the NPR latency. TMCC FREE CLK (1) H, which is independent of any processor time state, will set NPR when UBCD NPR RQ L is asserted.

If the processor is not in the process of servicing a bus request, NPG will be set on the next UBCD GRANT CLK L pulse. Thus, the processor asserts the UBCD PROC NPG H onto the Unibus to the device that is requesting Unibus control simultaneously. The NPG flip-flop remains set until the device responds with a SACK signal. The UBCA D SACK (delayed SACK) then clears the NPR flip-flop and the device scheduled to become the new master waits for the BBSY to be cleared so that it can become master and assert BBSY. At the time NPG is set, UBCD GRANT L is asserted to begin the NO SACK timing sequence described in Paragraph 7.7.9.1.

7.7.9 Priority Bus Request

The priority arbitration logic on TMCA and TMCB checks bus requests against the priority established by the processor status word. When a bus request on one of levels 4 through 7 is honored, UBCD EXT BRQ H is asserted as shown on drawing UBCD. Nothing happens until the processor enters time state TS2 of the interrupt pause cycle. When no NPR is present, the SERVICE BR flip-flop will be set by the clock pulse. As a result, the GRANT BR flip-flop will be set on the following TMCE FREE CLK (1) H transition. UBCD GRANT BR (0) H is the select input to a 74157 data selector. When GRANT BR is set, the B inputs (honor bus requests) are selected to provide the processor bus grant outputs. Thus, if the priority arbitration logic issued TMCA HONOR BR 5 L, UBCD PROC BG 5 H will then be asserted.

7.7.9.1 NO SACK – At the time GRANT BR or NPG is set, UBCD GRANT L is asserted to initiate the NO SACK timeout sequence. UBCD GRANT H is applied to a 74123 one-shot. A 5- to 10-μs delay is provided before the 74123 output clocks and sets the NO SACK flip-flop. If no BUS SACK signal is received by this time, the UBCD NO SACK (1) L signal clears the NPG and GRANT BR flip-flops.

7.7.9.2 INTR RESTART – There are two ways to restart timing after a device gains bus control with one of the BR levels. The logic that generates INTR RESTART H is shown on drawing UBCA. The UBCA INTR RESTART H signal is usually asserted by UBCC INTR B L. Under normal conditions, the device may perform several transfers before asserting BUS INTR L.

If a device does not respond to BG by asserting SACK within 5 to 10 μs, the NO SACK flip-flop will clear. The bus grant signal is cleared and restart is accomplished.

A passive release will also cause the timing to restart. When GRANT BR is set, UBCD GRANT BR H will set the PASSIVE flip-flop. The delayed SACK response from the device, D SACK L, clears the GRANT BR flip-flop. When the device drops BBSY, upon completion of the last data transfer, the UBCA INTR RESTART H signal is asserted. The PASSIVE flip-flop will be cleared by TIGD T5 L.

7.7.10 Interrupt Flag

Refer to drawing UBCC. When a device which has been selected as bus master asserts BUSA INTR L, it initiates an interrupt bus transaction. When the processor receives BUSA INTR L, the INTRF (interrupt flag) flip-flop is set, which asserts UBCC (PWRF + INTR) L. At the same time, UBCC INTR B L restarts timing by asserting UBCA INTR RESTART H.

The processor waits until T1 for deskew to ensure that all bits of the interrupt vector address are available on the D lines. When TIGC T1 B H goes high, the interrupt vector is loaded into the BR from the D lines and the INT SSYN (internal slave sync) flip-flop is set. When INT SSYN sets, it asserts BUSA SSYN L, which is sent to the interrupting device. As a result, the interrupting device clears BUSA INTR L, the D lines, and BUSA BBSY L. This is an active release of the Unibus by the interrupting device.

When the BUSA INTR L input to the processor goes high, UBCC INTR B L goes high. Because the MSYN flip-flop is cleared at this time, the INT SSYN flip-flop will be direct-cleared when UBCC INTR B L goes high. Therefore, the processor clears BUSA SSYN L and enters the interrupt sequence.

7.7.11 Internal SSYN

Refer to drawing UBCC. Any DATI or DATO transaction that involves the processor internal registers (PS, PIRQ, SL, PB) is initiated in the same way as a Unibus transaction with an external device. However, the processor Unibus control logic must also provide the SSYN responses.

The internal register address is ANDed with UBCA CPBSY B H on TMCD to assert TMCD INTERNAL ADRS H. After time is allowed for address deskew, the MSYN flip-flop will be set, and therefore, UBCB MSYN SET H is asserted. UBCC CLK SSYN H clocks BR and a 50-ns delay is provided to allow the priority arbitration logic to settle, if the processor status word was changed. Then, the INT SSYN flip-flop will be set, which causes the BUSA SSYN L signal to be asserted. Then, UBCC SSYN B H asserts UBCA SSYN RESTART H to restart timing. MSYN is cleared at time state TS1. When UBCB MSYN (0) L goes low, the INT SSYN flip-flop is direct-cleared.

7.7.12 Data Transfer Control Decoding

The Unibus control logic decodes control signals TMCE C0 H and TMCE C1 H to assert the BUSA C0 L and BUSA C1 L control signals. The control line bus drivers are shown on drawing UBCC. The processor only controls these Unibus control lines when the CPBSY flip-flop is set. The inputs are ANDed with UBCA CPBSY B H. Note that the drivers for the Fastbus control signals UBCC MEM BUS C0 L and UBCC MEM BUS C1 L are always enabled. Decoder logic is included to provide the DATI, DATIP, DATO, and DATOB control signals required on the UBC module. Decoding is summarized in the following chart.

TMCE C1 H	TMCE C0 H	Output Asserted
L	L	UBCC DATI L: data in
L	H	UBCC DATIP L: data in, pause
H	L	UBCC DATO L: data out
H	H	UBCC DATOB L: data out, byte

7.7.12.1 **HI BYTE/LO BYTE** – When a data-out bus transaction is in progress, UBCC DATO L always asserts both the UBCC HI BYTE H and UBCC LO BYTE H signals (drawing UBCC). If the bus transaction is a data-out byte, only one of these signals is asserted. If the byte address bit DAPB BAMX 00 H is high, the high-order byte is addressed and UBCC HI BYTE H is asserted. Conversely, if DAPB BAMX 00 H is low, the low-order byte is addressed and UBCC LO BYTE H is asserted.

7.7.12.2 **CC DATA** – The purpose of the CC DATA flip-flop is to jam-set the N, Z, V, and C condition code bits into the PS register when an explicit reference is made to the PS word. When UBCC LO BYTE H is asserted (DATO or DATOB) and the PS address is decoded, the CC DATA flip-flop D input goes low. As soon as MSYN sets, CC DATA is clocked and set. As a result, the BR bits ⟨03:00⟩ are jam-set into the PS bits ⟨03:00⟩. Thus, the arithmetic result of the operation is not clocked into the condition codes; instead, the data being read into the PS is maintained true while the normal T1 condition-code clock is overridden.

7.7.13 Power Control

The power supply provides two control signals to the processor on the Unibus, BUSA AC LO L and BUSA DC LO L. The BUSA AC LO L is asserted by the power supply to indicate that power failure is imminent. From the time that BUSA AC LO L is asserted, the power supply can continue at full load for about 2 ms. The BUSA DC LO L is an independently-detected signal that indicates one or more of the regulated dc output levels is dropping. In the event of AC power failure, the BUSA DC LO L signal follows the BUSA AC LO L signal within a few milliseconds. The BUSA AC LO L and BUSA DC LO L signals control the power-down and power-up logic shown on drawing UBCE.

7.7.13.1 Power Down – Refer to the timing diagram shown in Figure 7-12. When BUSA ACLO L is asserted during normal operation, the power down flag PDNF is set, because BLOCK DOWN is reset at the end of the previous power sequence. PDNF is applied to the priority arbitration logic on TMCA; the first BRQ strobe generates TMCE BRQ CLK H, which clocks the interrupt flags into the priority logic. If no higher priority flag is up (CONF or SL YEL), TMCA HONOR PWRF L is then generated. At the end of the current instruction, the ROM branches to the service flows.

At microstate BRK.20, UBCB ACKN B H goes high at TS3 and sets TMCC BLOCK STROBE (1) H. At microstate SVC.90, if no aborts are pending, this signal and TMCE CLK CONF H generate TMCC AC CLEAR L, which is ANDed with TMCA HONOR PWRF L to set the UBC PF CLR (1) H flip-flop.

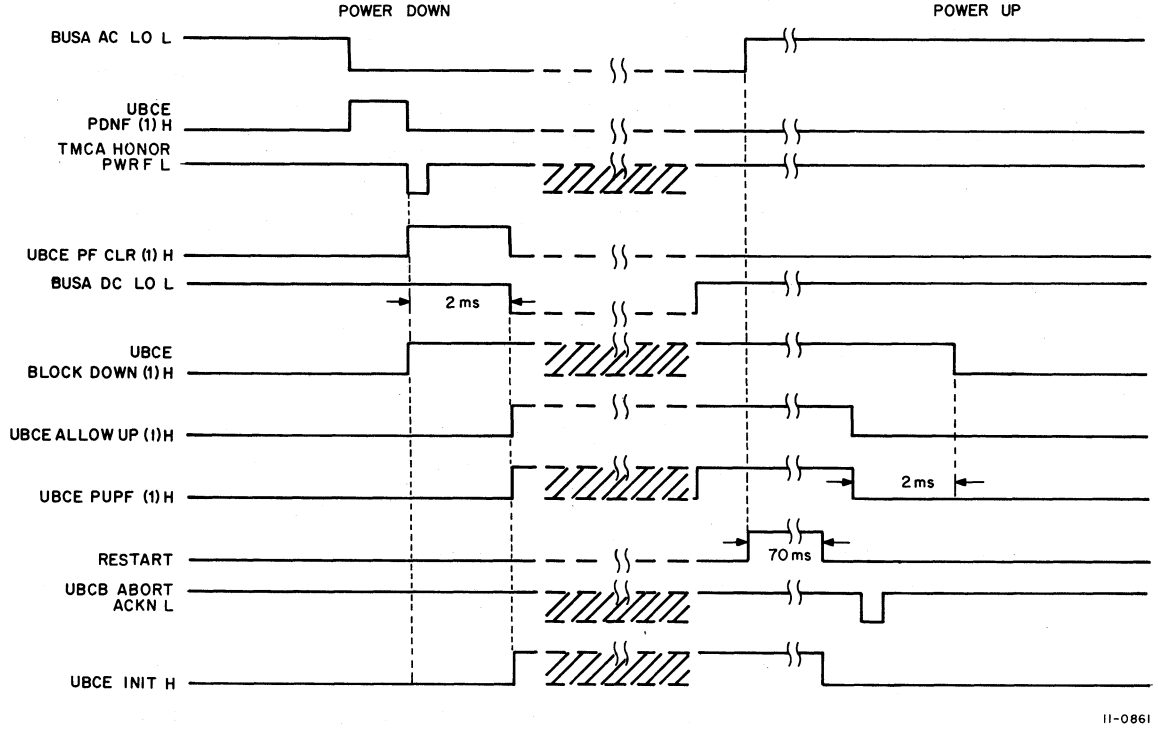


Figure 7-12 Power-Down/Power-Up Sequence

PF CLR does the following:

a. resets the TMCA priority flip-flops;

b. resets UBCE PDNF;

c. starts the 2 ms timer (E47) which, at the end of its delay, fires DCLO 1 one-shot; the pulse thus generated goes out on the UNIBUS as BUSA DCLO L.

By this time, all the internal traps and service routines should have been executed; no further bus transactions can occur, because DCLO asserts the initializing signals:

a. UBCE INT BUS INIT L – clear internal registers PD, PIRQ, SL and PB, and the priority arbitration flip-flops

b. UBCE ROM INIT H – forces ROM to ZAP.00 (200) and sets timing generator to T4

c. UBCE INIT – clears processor, floating-point processor, and KT11-C.

d. BUSA INIT L – initializes Unibus.

In addition to these signals, DCLO sets the ALLOW UP and PUPF (power up) flip-flops, which set up the power-up sequence, should the DCLO signal not be generated by the H742 power supply, or should ACLO be negated before DCLO is asserted by the H742.

7.7.13.2 **Power Up** – When AC power is restored, the power supply will remove the BUSA DC LO L and BUSA AC LO L signals. The BUSA DC LO L signal will go high before BUSA AC LO L. When BUSA AC LO L goes high, the UBCE ALLOW UP (1) H signal starts the RESTART time delay by triggering the 74123 RESTART one-shot. This time delay allows 70 ms for the magnetic core memory internal power supplies to power up. During this delay interval, the low output from the RESTART one-shot asserts all the initializing signals to inhibit all processor activity. After the RESTART interval, the processor proceeds with the power-up microprogram routines. At T2 of PUP.00, ACKN is decoded and at TS3, UBCB ABORT ACKN L will be asserted; this clears ALLOW UP and PUPF. When PUPF is cleared, UBCE PUPF (0) L initiates a 2-ms delay by triggering the 74123 DOWN DLY one-shot. For this period of time, BLOCK DOWN remains set and prevents any BUSA AC LO L assertion from setting PDNF. This ensures that the processor will complete the power-up sequence before another power-down is initiated.

7.7.14 Initialization

The initialization logic is shown on drawing UBCE. There are three basic sources of initialization: the console START switch, the power-down/power-up control logic, and the reset instruction in kernel mode.

7.7.14.1 **Power-Down/Power-Up** – The power-down/power-up control logic asserts the initialization signals as shown in the timing diagram. The UBCE INIT H level is typical; it is asserted from the time BUSA DC LO L is asserted until the RESTART signal goes high.

7.7.14.2 **CNSL RESET** – When the processor is halted and the START switch is asserted, UBCF CNSL RESET L asserts the initialization signals. The primary purpose is for maintenance, if the processor or a device hangs the Unibus or microprogram.

7.7.14.3 START – The console START switch sets the START FLAG flip-flop. It also asserts UBCE STATUS CLR L. During machine state RES.10, the BCD field is decoded to provide code 4, INIT IF PS14 (0). At TS3, therefore, the RESET one-shot is triggered to begin a 10-ms delay. During this time, the true output is ANDed with START (1) H to assert UBCE START INIT L and UBCE INIT H. These initializing signals are used to initialize all processor, memory management, and floating-point modules. At the same time, the 0 output of the RESET one-shot asserts BUSA INIT L to initialize all devices on the Unibus, and UBCE INT BUS INIT L to initialize any registers that use the internal data bus.

7.7.14.4 RESET ABORT – When the 10-ms RESET one-shot is fired, a 1-μs RESET ABORT one-shot is also fired. The output is used to clear the RESET one-shot if a power fail occurs after the first 1 μs of the initialization interval.

7.7.14.5 RESET in Progress – When a reset instruction is executed and state RES.10 is entered in the kernel mode, the RESET one-shot will be triggered. The 0 output will go low to assert UBCE INT BUS INIT L, UBCE RIP + FP SYNC L, and BUS INIT L. UBCE RIP + FP SYNC L is used in microbranch-enable BE00 to keep the ROM cycling in RES.20 until the 10-ms delay is completed. The microbranch-enable is shared with FP SYNC. During FP instruction executions, the processor waits for FP SYNC once FP ATTN has been sent.

7.7.15 Console Switch Inputs

The console switch inputs are connected to the UBC module as shown on drawing UBCJ. The switch control inputs are used to clock and set associated flip-flops in the console control register (drawing UBCF). When any flip-flop is set, UBCF ACT H is asserted. If HALT and S INST CYCLE switches are set, UBCF S/INST L will be asserted. This level is used to condition the CONF flip-flop on TMCA. At the end of an instruction, when the processor checks for bus requests, CONF will set. CONF must be set before the console START switch can clock and set the START bit.

7.7.15.1 DEC Data Center Inputs – The UBC/DEC Data Center interface is shown on drawing UBCJ. When DDC STOP L and DDC BEGIN L are asserted from the DEC Data Center, the START bit of the console control register will be set. When DDC STOP L and DDC LOAD L are asserted from the DEC Data Center, the CNSL07 (load address) bit will be clocked and set.

7.7.15.2 Console Control Register – The console control register will be cleared at the end of each console control sequence (microstate CON.20). At T2 of this microstate, RACC UBCT ⟨02:00⟩ are decoded to assert UBCF ACKN LEVEL H (010 = CNSL.ACKN). When UBCA TS4 CLK H goes high, all console control register flip-flops are direct-cleared.

An interrupt or initialize signal will also clear most of the console control register bits. The only exception is START. UBCE DC LO B L is the only other condition that can clear the START bit.

7.7.16 Console Control Decoder

The states of the console control register flip-flops are decoded to provide the microstate address of the console control function. The console function and associated microstate address bits are tabulated on UBCH. The decoder logic asserts UBCH CNSL ⟨07, 02, 01, 00⟩ H outputs, which are clocked into a storage register at TS3 of the last microstate of a control function sequence. The purpose of this 4-bit register is to hold the microstate address of the previous console operation. EXAM STEP and DEP STEP can not be entered unless enabled by the preceding console operation.

7.7.16.1 **EXAM and STEP EXAM** – The microstate address for these functions is formed indirectly. If no other control register flip-flop is set, the UBCH EXAM + STEP EXAM H output will be asserted when the UBCH STRB FTN H clock input occurs. This allows the EXAM function to be initiated. It also gates UBCF EXAM (1) H to UBCH CNSL00 H to produce the EXAM STEP microstate address as the next console function.

The REG EXAM console function is decoded directly from UBCF REG EXAM (1) H, which asserts microstate address bit 1 (RAD.00 address 072).

7.7.16.2 **DEPOSIT and STEP DEPOSIT** – The DEPOSIT console function can be initiated directly by UBCF DEPOS (1) H. Initially, UBCH STEP DEP + DEP L will be high and microstate address 073 will be asserted when the DEPOSIT flip-flop sets. When it does, microstate address 073 is generated. As a result, the storage register holds the complement 1100. Therefore, UBCH STEP DEP L is asserted. This asserts UBCH STEP DEP + DEP H. The next DEPOSIT control function will actually be a STEP DEPOSIT with microstate address 074 being generated. Under these conditions, the storage register holds the complement 1011. This content will assert UBCH DEP L, which also asserts the UBCH STEP DEP + DEP H. Thus the microstate address will remain 0100 each time a DEPOSIT is initiated.

The REG DEPOSIT (075), START (076), CONTINUE (077), and LD ADRS (270) microstate addresses are encoded directly from the associated control register outputs.

7.8 TIG MODULE M8109

The Timing Generator (TIG) Module M8109 provides the timing generator logic elements for the KB11-A.

7.8.1 Timing Sources

The three sources of KB11-A timing are the crystal clock, the R/C clock, and the MAINT STPR switch S0 (on the maintenance card). These timing sources are shown in drawing TIGB.

7.8.1.1 **Crystal Clock** – The crystal clock provides a constant square wave output of 33 MHz. The oscillator frequency is determined by the LC tuned-collector network and stabilized by the crystal connected between emitters. An offset network in the base circuits ensures that the oscillator will start when +5V is applied to the module. The amplified output, TIGB XTAL H, is a +3.5/0V square wave with a 30-ns period.

7.8.1.2 **R/C Clock** – The R/C clock is provided for maintenance test purposes and is available only when the maintenance card is plugged into the CPU backplane. The frequency of the square-wave output, TIGB RC H, can be adjusted as high as 37 MHz by varying potentiometer R104 in the RC feedback network. Thus, the clock pulse period can be narrowed to approximately 27 ns to test for race conditions in the logic.

7.8.1.3 **MAINT STPR Switch** – The third source of timing is the manually-operated single-step MAINT STPR switch S0, located on the maintenance card. This switch is only enabled when maintenance card switches S2 and S3 are both set to 1.

7.8.2 Source Synchronizer

The timing source synchronizer is shown on drawing TIGB. The purpose of the source synchronizer is to select only one timing source at any given time and inhibit the two remaining sources. The synchronizer prevents cycles of improper length and ensures that TIGB SOURCE CLOCK L starts in the high (non-asserted) state when switching between sources. Timing source selection is determined by the setting of switches S1, S2, and S3 when the maintenance card is plugged in. If the maintenance card is not installed, the crystal clock is the only source of timing. The following paragraphs describe timing source selection when the maintenance card is plugged in.

7.8.2.1 **Crystal Clock Selection** – When maintenance card switch S3 is not set, XMAA S3 L is high. When the RC EN and MS EN flip-flops are not set, the XTAL SYNC flip-flop is set by the next TIGB XTAL L pulse going high. With maintenance card switches S1 and S2 not on, MS EN will be cleared, as will RC SYNC and RC EN. Therefore, XTAL SYNC is set, and on the next TIGB XTAL H going low, XTAL EN flip-flop will be set. As a result, the source multiplexer output, TIGB SOURCE CLOCK L, will follow the XTAL H input to the XTAL EN flip-flop.

Note that XTAL EN (1) L inhibits XMAA S3 H inputs to the RC EN flip-flop. Therefore, the XTAL SYNC flip-flop must be cleared before a timing source change can be accomplished. The RC EN and MS EN gating input to the XTAL SYNC flip-flop ensures that these sources have been disabled before XTAL EN is allowed to gate the XTAL H pulse through the source multiplexer.

7.8.2.2 **RC Clock Selection** – The RC clock is selected as the timing source when maintenance card CLK switch S3 is on RC, and S2 and S1 are both set to 0. The XMAA S3 L input is low and the RC SYNC flip-flop will be set by the next TIGB RC L pulse going high. As a result, the RC EN flip-flop will be set by the following TIGB RC H clock pulse going low. The source multiplexer output, TIGB SOURCE CLOCK L, will then follow the TIGB RC H input. TIGB XTAL EN (0) H and TIGB MS EN (0) H are fed back to inhibit TIGB RC SYNC D inputs to ensure that the enable flip-flops are cleared before the timing source can be changed.

7.8.2.3 **MAINT STPR Selection** – The maintenance card S2 and S1 switches are both set to 1 to allow single timing pulses to be generated by MAINT STPR switch S4. The XMAA S1 L and XMAA S2 L inputs are both low. The resultant input to the MS EN flip-flop D input causes the flip-flop to be set on the next TIGB XTAL H clock input. On the following TIGB XTAL H and TIGB RC H clock pulses, the XTAL SYNC and RC SYNC flip-flops will be reset. Succeeding clock pulses will then reset the XTAL EN and RC EN flip-flops. MS EN (1) H is ANDed with STEP (1) H to assert the TIGB SOURCE CLOCK L output of the source multiplexer. Each time momentary MAINT STPR switch S4 is pressed, the STEP flip-flop toggles.

NOTE
The MAINT STPR switch must be actuated twice to complete a single TIGB SOURCE CLOCK L output pulse.

Removing the S2 or S1 input conditions the MS EN flip-flop to be cleared on the next XTAL H clock pulse, going low. MS EN (0) L direct-clears STEP to condition it for the next time the SING TP function is selected.

7.8.2.4 **Synchronization** – The feature of the source synchronizer is that the output level is maintained high (non-asserted). The sample timing diagram shown in Figure 7-13 shows the TIGB SOURCE CLOCK L output as the maintenance card CLK switch is changed from XTAL to RC. With the XMAA S3 L input low (RC clock

selected), the XTAL SYNC flip-flop is cleared on the next TIGB XTAL L clock pulse going low. One XTAL H clock pulse later, XTAL EN will be cleared, enabling the D input to the RC SYNC flip-flop. The next time TIGB RC H goes low, RC SYNC will be set. The difference in XTAL H and RC H pulse widths is exaggerated in Figure 7-13 to indicate that the clock pulses are completely independent. Note that the SYNC and EN flip-flops are clocked on the trailing edge of the source clocks so that the gating level to the source multiplexer is always removed as the clock input is non-asserted. This provides a clean leading edge for TIGB SOURCE CLOCK L. Note also that only half a clock period is available for the enable flip-flop to change state and gate the associated clock source through the multiplexer. The synchronizer output will remain high the first time the MAINT STPR switch is actuated.

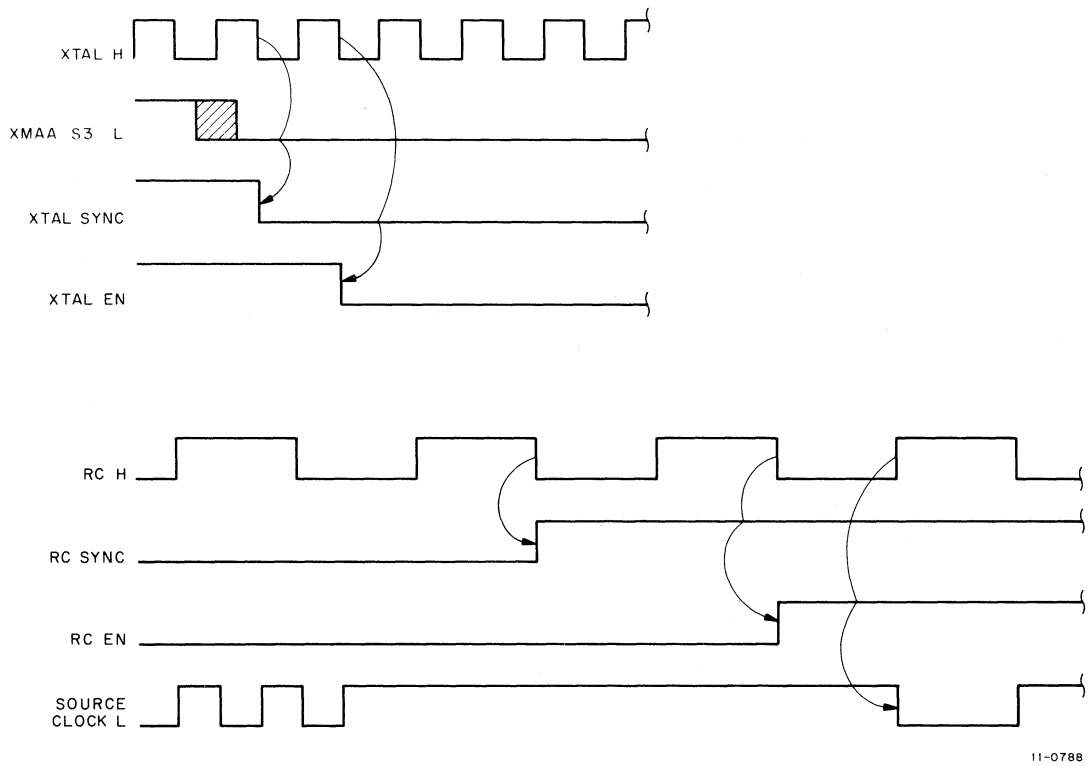


Figure 7-13 Timing Source Synchronization

7.8.3 Phase Splitter/Buffer

The phase splitter/buffer, shown on drawing TIGB, is driven by TIGB SOURCE CLOCK L from the source synchronizer to produce timing pulse outputs TIGB CLOCK L and TIGB CLOCK H. The TIGB CLOCK L output pulses are in phase with TIGB SOURCE CLOCK L.

7.8.3.1 Level Converter – Transistors Q65 and Q66 convert the TIGB SOURCE CLOCK L output to the level required at the phase splitter inputs. A low input at the base of Q65 causes all the emitter current to flow through Q65. With the +V reference voltage applied at the base of Q66, no current flows through Q66 and R122. Thus a low input provides a low output. When TIGB SOURCE CLOCK L goes high, Q65 cuts off, and the +V reference causes current flow through Q66 and R122 to provide a high output.

7.8.3.2 **Phase Splitter** – The phase splitter consists of two emitter-coupled 2N3009 transistors, Q61 and Q62. At rest, with no output signal from the source synchronizer, Q61 is forward-biased. A fixed bias at the Q62 base holds that transistor cut off. Under these conditions, the TIGB CLOCK H output provided by buffers Q53 and Q54 will be low because Q61 will be conducting. Q54 will be turned on.

When TIGB SOURCE CLOCK L starts to go low, as the result of a clock pulse, the base of Q61 goes negative with respect to the Q62 base. More current flows through Q62, causing a greater voltage drop across the Q62 collector resistors, R109–R111. Less voltage is developed across common emitter resistors R89–R96, increasing the forward bias on Q62. As a result when Q62 starts to conduct more current, Q61 just as quickly starts to cut off. The circuit is a differential amplifier that responds to slight changes of the input signal at high speed. When TIGB SOURCE CLOCK L starts to go positive, Q61 turns on and Q62 cuts off just as fast. The switching action of Q61 and Q62 follows the TIGB SOURCE CLOCK L signal with only about 1-ns difference between TIGB CLOCK H and TIGB CLOCK L.

7.8.3.3 **Buffers** – Each buffer stage consists of a 2N3009 and a 2N4258 transistor. When Q61 turns off as a result of a low source synchronizer output, Q53 is biased on and Q54 is cut off. Thus, the TIGB CLOCK H output goes high, which is 180° out-of-phase with the TIGB SOURCE CLOCK L input. At the same time, Q62 turns on and the positive collector cuts off Q55 and forward biases Q56. Therefore, TIGB CLOCK L goes low, which is in phase with the TIGB SOURCE CLOCK L input from the source synchronizer.

7.8.4 Timing Generator

The timing generator is shown on drawing TIGA. It consists of five J-K flip-flops, T1 through T5, connected as a ring counter. The flip-flops are initially direct-cleared by TIGA ROM INITA L, which is asserted by UBCE ROM INIT H. This initializing signal is provided when the START switch is pressed, while the ENABL/HALT switch is in HALT, to produce a system clear. It is also provided if a power failure occurs, as described in Paragraph 7.7.13. A time state diagram is shown in Figure 7-14.

NOTE

Unless otherwise indicated, consider TIGB XTAL H as the timing source for normal processor operation.

When the timing generator has been initially cleared, a high input is provided at the J input of T4. This conditioning input is produced by T1 (0), T2 (0), and T5 (0). When TIGA ROM INIT L sets INIT SYNC, it resets T4. After TIGA ROM INIT L, the next positive-going transition of TIGC TPB L resets INIT SYNC. This allows T4 to be set by the next negative-going transition of TIGC TPB H. T4 will be set by the next TIGB TP H negative-going transition as shown in the timing diagram. The conditional J input to the T4 flip-flop ensures that the counter will remain in sequence. T4 will reset on the next TIGB TP H clock pulse. As long as the conditional inputs STOP T1 L or STOP T3 L are not asserted, the timing generator operates as a synchronous ring counter that continuously cycles through the five time states shown in Figure 7-14. The flip-flop associated with each time state will remain set for one time period (typically 30 ns). The minimum machine cycle time is approximately 150 ns. The purpose of the STOP T1 and STOP T3 signals is to prevent the processor from entering these time states under certain conditions. The various conditions that cause TIG STOP T1 L to be asserted are described in Paragraph 7.8.5. The various conditions that cause TIGA STOP T3 L to be asserted are described in Paragraph 7.8.6.

NOTE

The timing generator counter outputs are not sufficiently accurate for use by the processor. Therefore, high-speed switches are used to provide the accurate timing pulses and time state levels required. The high accuracy timing pulse generators are described in Paragraph 7.8.7. The time state generators are described in Paragraph 7.8.8.

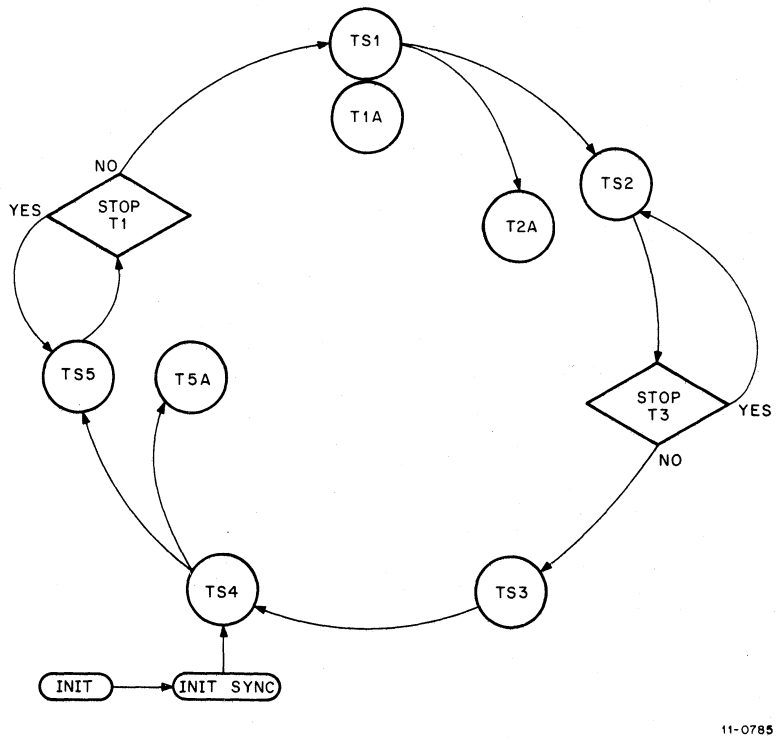


Figure 7-14 Time State Diagram

7.8.5 STOP T1

There are five conditions that can assert the TIGA STOP T1 L signal. If any of these conditions exist, TIGA STOP T1 L remains low, T5 remains set on succeeding clock pulses, and T1 can not be set. The processor is suspended in time state T5 until all conditions that assert TIGA STOP T1 L have been resolved. Then the D input to the T1 SYNC flip-flop goes high and it will be set the next time TIGB TPB L goes high. As a result, the D input to T1 is enabled so that T1 sets on the next TIGB TP B H negative-going transition.

The combinational logic that produces STOP T1 L is shown on drawing TIGA. The following paragraphs describe each of the conditions that can assert TIGA STOP T1 L.

7.8.5.1 **Not In T4 or T5** – If the timing generator is not currently in T4 or T5, TIGA T5 (0) H and TIGA T4 (0) H assert TIGA STOP T1 L, preventing the counter from proceeding into T1. TIGA STOP T1 L is asserted most of the time as a function of the counter.

7.8.5.2 **Semiconductor Memory Delay** – When the semiconductor memory is accessed, TIGA STOP T1 L will be asserted during a bus pause or bus long pause cycle until SMCA MEM SYNC B L is received, provided no abort has occurred.

7.8.5.3 **Conventional Memory Delay** – During a bus long pause cycle on the Unibus, TIGA STOP T1 L will be asserted until UBCA BLP DESKEW H qualifies the D input to UNI DLY and it is set by the next time pulse. An abort will also cause the stop condition to be negated.

7.8.5.4 **Operating System Test** – The OST option will assert the PHKA STOP T1 L input under certain conditions.

7.8.5.5 Single Cycle Mode – When the processor is halted and placed in the S BUS CYCLE mode of operation from the console, the SNGCY flip-flop is direct-set to assert TIGA STOP T1 L and cause the processor to halt after each single bus cycle is completed. When the CONT switch is pressed, the CONT flip-flop is set on the next TIGB TPB L pulse going high. This enables the K input to the SNGCY flip-flop so it will reset on the next TIGB TPB H pulse going high. This allows the processor to enter T1 and proceed through another bus cycle. As soon as T1 is entered, the flip-flop controlled by the CONT switch is reset. The CONT flip-flop resets on the next clock pulse and the SNGCY flip-flop is again set on the trailing edge of that clock pulse. As a result, TIGA STOP T1 L is again asserted to stop the processor after a single bus cycle.

While TIGA STOP T1 L is asserted, T5 will remain set until the stop condition is negated. T5A, however, is not controlled by STOP T1 L and will remain set for only one time period. Thus, only one T5 timing pulse is produced during any stop condition. When the stop condition is negated, T1 will set and T5 will clear.

7.8.6 STOP T3

There are seven conditions that can assert the TIGA STOP T3 L signal. If any of these conditions exist, TIGA STOP T3 L remains low, T2 remains set on succeeding clock pulses, and T3 can not be set. The processor is suspended in time state T2 until all conditions that assert STOP T3 L have been resolved. The combinational control logic that produces STOP T3 L is shown on drawing TIGA. The following paragraphs describe each of the conditions that can assert TIGA STOP T3 L.

7.8.6.1 Not In T2 – Any time the processor is not in time state T2, T2 (0) H will assert TIGA STOP T3 L.

7.8.6.2 Single Cycle – When the SNGCY flip-flop is set, TIGA STOP T3 L is asserted. The flip-flop will be direct-set under control of the S BUS CYCLE and HALT switches on the console. The TIGB SINGLE CY L signal that direct-sets the flip-flop will not be asserted until time state T3 of a bus pause or bus long pause cycle.

7.8.6.3 ROM + UPB – Maintenance card switch inputs XMAA S1 L and XMAA S2 L are decoded by an exclusive-OR gate that provides TIGB ROM + UPB H if either is set. Setting S2 alone selects a single ROM cycle; setting S1 alone selects a microprogram break (UPB). The logic is shown on drawing TIGB. When TIGB ROM + UPB H is asserted, the ROM + UPB flip-flop is set by the next TIGB TPB L clock pulse going low. This conditions a J-K flip-flop on TIGA, which is set on the trailing edge of that clock pulse and causes TIGA STOP T3 L. Each time the CONT switch is pressed, CONT (1) H causes the timing cycle to proceed.

7.8.6.4 Bus Pause or Long Pause Delay – During a bus pause or bus long pause cycle that does not involve fast memory, TIGA STOP T3 L is asserted until the slave device clears SSYN. The processor will not proceed into time state T3 until the data transfer is available to clock into the BR.

7.8.6.5 Interrupt Pause Delay – During an interrupt pause to service an external break request, STOP T3 L is asserted until the UBCA INTR RESTART H signal is asserted. The conditions that cause INTR RESTART are described in Paragraph 7.7.9.2.

7.8.6.6 Operating System Tester – The operating system tester (OST) option will assert the PHKA STOP T3 L input under certain conditions.

7.8.6.7 **KT11-C Delay** – During a BUST, TIGA STOP T3 L will be asserted when the SSR DLY flip-flop is set. The purpose of the delay is to allow time for the physical address to be propagated through the page address path of the KT11-C unit before the processor enters time state T3.

NOTE
If the KT11-C option is not installed, SSRB ENABLE T3 DLY H is grounded by the SJB module and the SSR DLY flip-flop is never set.

When the KT11-C option is enabled and a conditional BUST is sensed by the KT11-C, SSRB ENABLE T3 DLY H will condition SSR DLY to be set. SSR DLY will then be set when the processor enters time state T2. SSR DLY (1) H and TIGA T2A (1) H assert TIGA STOP T3 L. TIGA T2A (1) H is high for one time period. There are several processor operations that cause a conditional BUST cycle to be initiated. If certain conditions are met, RACH DIS BUST L is asserted to discontinue the BUST cycle. Therefore, under these conditions, TIGA STOP T3 L is only asserted for one time period while T2A is high.

If RACH DIS BUST L is not asserted, the BUST cycle is to continue. The duration of STOP T3 is then determined by S0 and S1, operating as a counter to control the delay. Jumper W3 is installed at the factory to provide a delay of three time periods. S1 is set two clock pulses after T2 and T2A. After the third clock pulse, SSR DLY is reset. During that interval, STOP T3 is asserted by SSR DLY (1) H and RACH DIS BUST L high (- RACH DIS BUST H). As a result, T3 is set by the fourth clock pulse after T2 is set, providing three time periods of delay. If the KT11-C is not enabled, SSR DLY will not occur.

7.8.7 Timing Pulse Generators

As indicated in Paragraph 7.8.4, the switching times of the flip-flops used in the timing generator ring counter are not very precise; therefore, the flip-flop states are not used directly for processor timing. Instead, high-speed transistors are used to generate the timing pulses. The timing pulse generator schematics are shown on drawing TIGC.

Each of the timing pulse generators gates the phase splitter/buffer clock pulse output with a time state generator output to generate the timing pulse associated with that state. Figure 7-15 is a sample timing diagram that shows how the T1 time state is gated with CLOCK H and CLOCK L to provide the T1 H and T1 L timing pulse outputs.

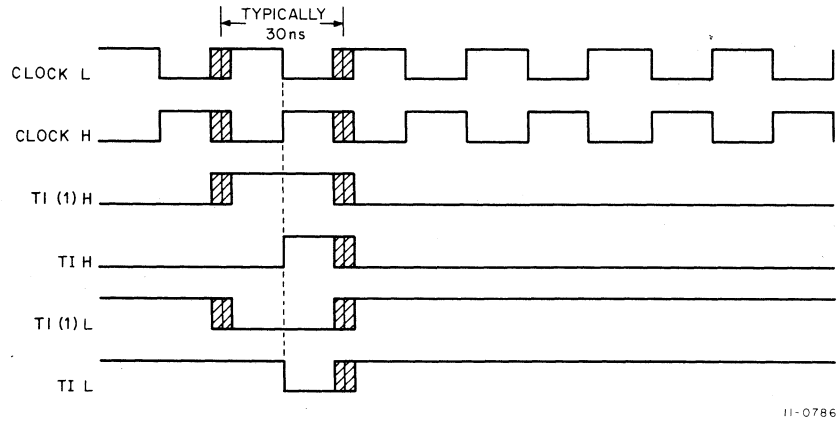


Figure 7-15 Timing Pulse Generation

7.8.7.1 **Positive Timing Pulse Generators** – Refer to TIGC T1 H circuit. In the quiescent state, the driver stage bias holds Q10 cut off and Q9 turned on so the associated TP H output is low. Each timing pulse generator is enabled by the associated time state output. For example, when TIGA T1 (1) H is asserted, Q40 is cut off and Q39 saturates. When TIGB CLOCK H goes high, the positive-going pulse is gated through Q39 to the bases of Q10 and Q9. As a result, Q10 conducts and Q9 is turned off to produce the leading edge of the TIGC T1 H clock pulse.

7.8.7.2 **Negative Timing Pulse Generators** – Refer to TIGC T1 L circuit. The negative pulse generator circuits operate in a manner similar to the positive pulse generator described in Paragraph 7.8.7.1, except that complements of the enable and clock inputs are used and transistor types are reversed. For example, in the quiescent state, Q12 is on and Q11 is off. When TIGA T1 (1) L is asserted, Q42 is cut off and Q41 saturates. When TIGB CLOCK L goes low, the negative-going pulse is gated through to the bases of Q12 and Q11. As a result, Q11 conducts and Q12 is turned off to provide the leading edge of TIGC T1 L.

7.8.8 Timing State Generators

The timing state generators provide the time state signal levels TS1 L through TS5 L used throughout the system. For example, TS2 L through TS5 L are used by the memory management to provide internal relocation timing. Refer to drawing TIGE.

Each time state level is provided by a pair of cross-coupled gates connected to operate as a high-speed flip-flop. Initially, all flip-flops are cleared by TIGA ROM INIT B L. When each timing generator flip-flop sets, the true output is ANDed with TIGB TPB H to set the associated time state flip-flop. Each flip-flop remains set for two clock periods. For example, TIGE TS1 L remains low until T3 (1) H is ANDed with TIGB TPB H to reset the flip-flop. Figure 7-16 illustrates the relationship between the time states. The leading and trailing edges of these time state signals are dependent upon the accurate TPH time pulses rather than the less accurate timing generator flip-flops. In general, the TS levels are used where timing is not critical.

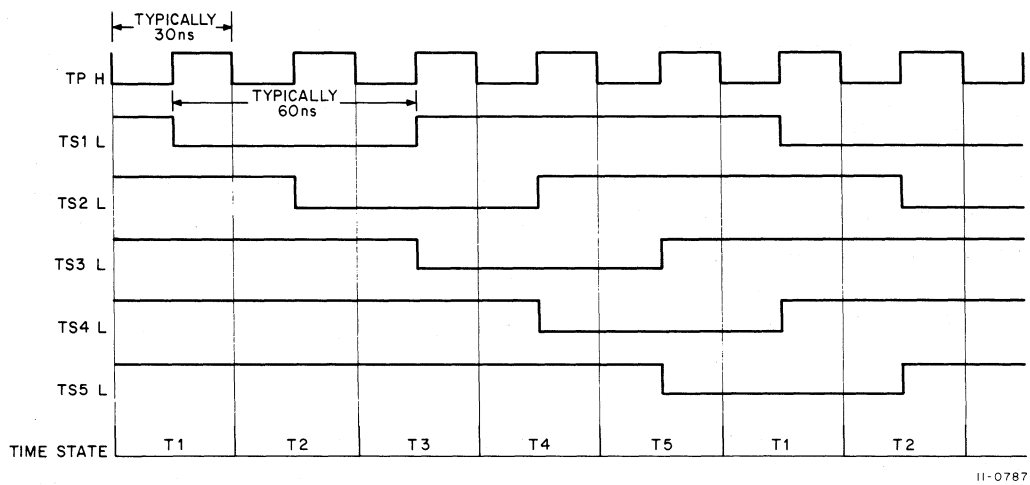


Figure 7-16 Generation of Time States

7.9 CONSOLE LOGIC

The console logic is located on the KNL console module and shown on drawings KNLA, KNLB, KNLC, and KNLD. The following paragraphs describe data address display select and mode select functions controlled from the console.

7.9.1 Switch Register and Data Display

Figure 7-17 is a simplified diagram of console switch register and data display functions. Complete circuit details are shown on drawings KNLA, PDRB, and PDRH.

7.9.1.1 Switch Register Inputs – KNLA SWR ⟨17:00⟩ H are connected to the PDR module by the KNL/PDR interface cable. PDRH SWR ⟨17:16⟩ H are used only for addresses. PDRH SWR ⟨15:00⟩ H are gated to the BRMX by TMCF READ SW L or TMCD SW ADRS L. TMCF READ SW L is asserted only during microstate CON.00. TMCD SW ADRS 5 L is asserted when the switch register address is specified.

7.9.1.2 DATA Display – The data display is controlled by a 4-position rotary switch that determines the select inputs to the 74S153 multiplexers shown on drawing PDRF. The data source selected by each switch position is shown in Figure 7-17 and listed in the following chart.

Switch Position	Data Source	Drawing Reference
BUS REGISTER	bus register A	PDRB
DATA PATHS	shifter output	DAPF, H, J
DISPLAY REGISTER	light register	PDRB
μADRS FPP/CPU	floating-point ROM address (high byte)	FRMB
	central processor ROM address (low byte)	RACD

7.9.2 Address Display and Control

Figure 7-18 is a simplified diagram that shows the sources of the address display bits. Complete circuit details are shown on drawing KNLB. The address display is controlled by the 8-position address select switch that also provides console control of the processor mode.

7.9.2.1 Address Bits ⟨05:00⟩ – The console ADDRESS indicators always display DAPB BAMX ⟨05:00⟩ H. These low-order address bits are not affected by relocation.

7.9.2.2 Address Bits ⟨15:06⟩ – The source of these address display bits is controlled by the address select switch. In the PROG PHY position, physical address bits SAPJ PA ⟨15:06⟩ H are the source. For any other address select switch position, virtual address bits SAPH VA ⟨15:06⟩ H are the source.

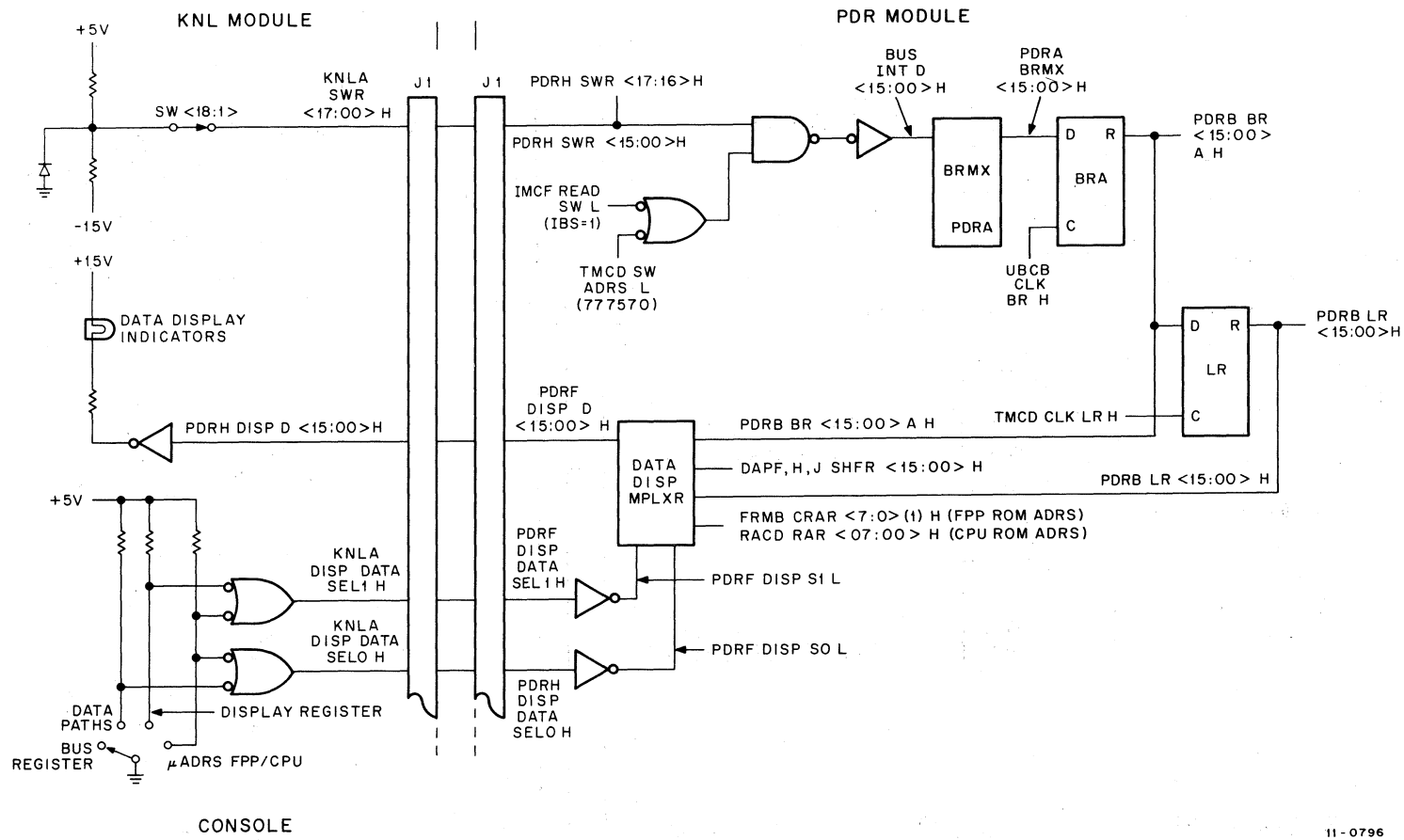


Figure 7-17 Simplified Diagram of Console Switch Register and Data Display Sources

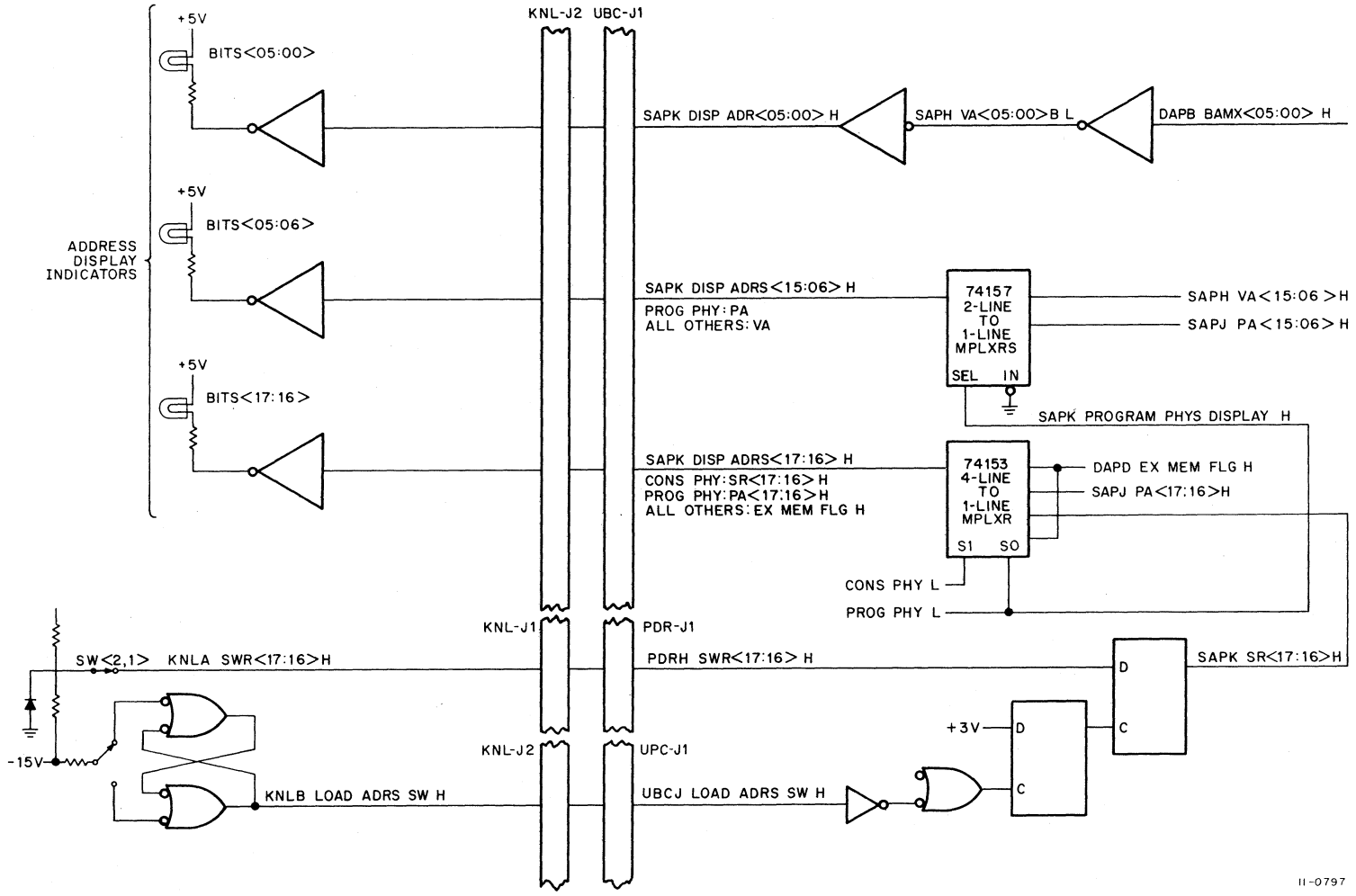


Figure 7-18 Sources of Address Display, Simplified Diagram

7.9.2.3 Address Bits ⟨17:16⟩ – The source of these address bits is determined by the address select switch and the console switches:

a. CONS PHY – Switch register bits KNLA SWR ⟨17:16⟩ H are selected as the address when the address select switch is set to CONS PHY. The current switch settings are stored as SAPK SR ⟨17:16⟩ H when the LOAD ADRS pushbutton is pressed.

b. PROG PHY – When the address select switch is set to PROG PHY, physical address bits SAPJ PA ⟨17:16⟩ H are selected and displayed. For relocated addresses, these bits are provided from the KT11-C PAR. Otherwise these bits are derived from the EX MEM FLG signal. DAPD EX MEM FLG H is asserted only when DAPD BAMX ⟨15:13⟩ H are high.

7.9.3 Console Mode Control

The console address select switch also functions as a mode control. Figure 7-19 is a simplified diagram of the mode control logic. Complete console circuit details are shown on drawing KNLB. Inputs from the console switch are encoded on the KNLB DISP ADRS SEL ⟨2:0⟩ H lines. The select code is listed on drawing KNLB. These lines are decoded on the SAP module, where KERNEL, SUPER, and USER I or D selections are ORed and gated to the appropriate space control flip-flop by UBCF CNSL ACT (1) H.

7.10 SJB MODULE M8116

If the KT11-C Memory Management Unit is not implemented in the PDP-11/45 System, SJB Module M8116 is installed in slot 14 of the CPU backplane. Drawing SJBA shows the address drivers that provide the output levels required by Unibus A and the address display indicators. The control inputs required for proper KB11-A operation without the KT11-C option are generated as shown on drawing SJBB.

When the KT11-C option is implemented, the SJB module is replaced by SAP Module M8107, and SSR Module M8108 is installed in slot 13.

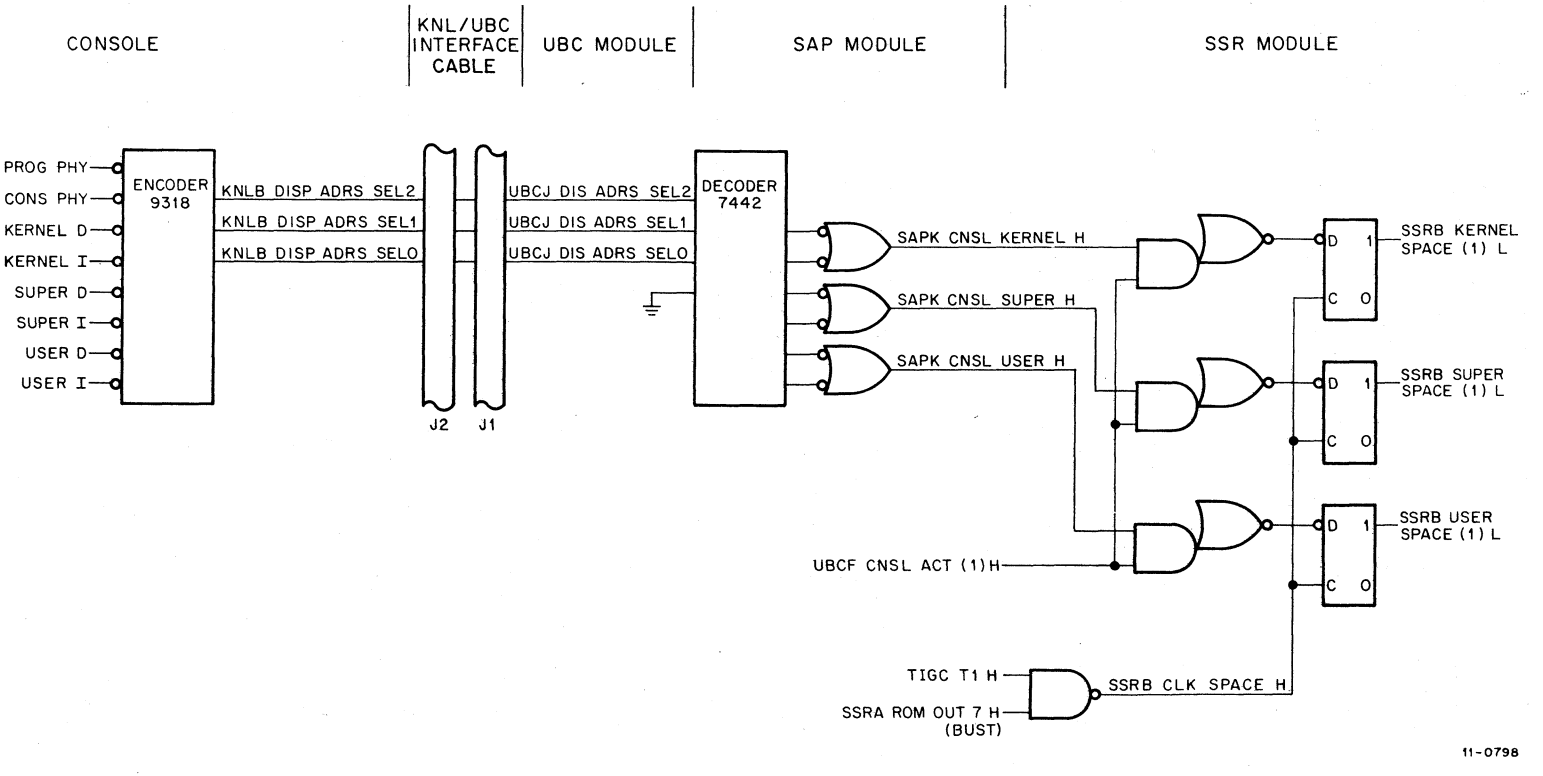


Figure 7-19 Console Mode Control, Simplified Diagram

CHAPTER 8
MAINTENANCE

The KB11-A Central Processor Unit is an integral part of the PDP-11/45 System. Most of the maintenance information that applies to the KB11-A, as well as to the basic PDP-11/45 System and options, is provided in Chapter 4 of the *PDP-11/45 System Maintenance Manual.* That information includes coverage of the diagnostic programs, use of the maintenance card and extender boards, margin tests, and integrated circuit removal and replacement procedures.

Chapter 4 of the *PDP-11/45 System Maintenance Manual* also includes a series of test procedures to be performed if the KB11-A fails to execute the initial diagnostic program, Unconditional Branch Test, MAINDEC-11-DOAA.

READER'S COMMENTS

KB11-A CENTRAL PROCESSOR UNIT
MAINTENANCE MANUAL
DEC-11-HKBAA-B-D

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

Digital Equipment Corporation
Technical Documentation Department
Digital Park, PK3-2
Maynard, Massachusetts 01754

DIGITAL EQUIPMENT CORPORATION
MAYNARD, MASSACHUSETTS 01754