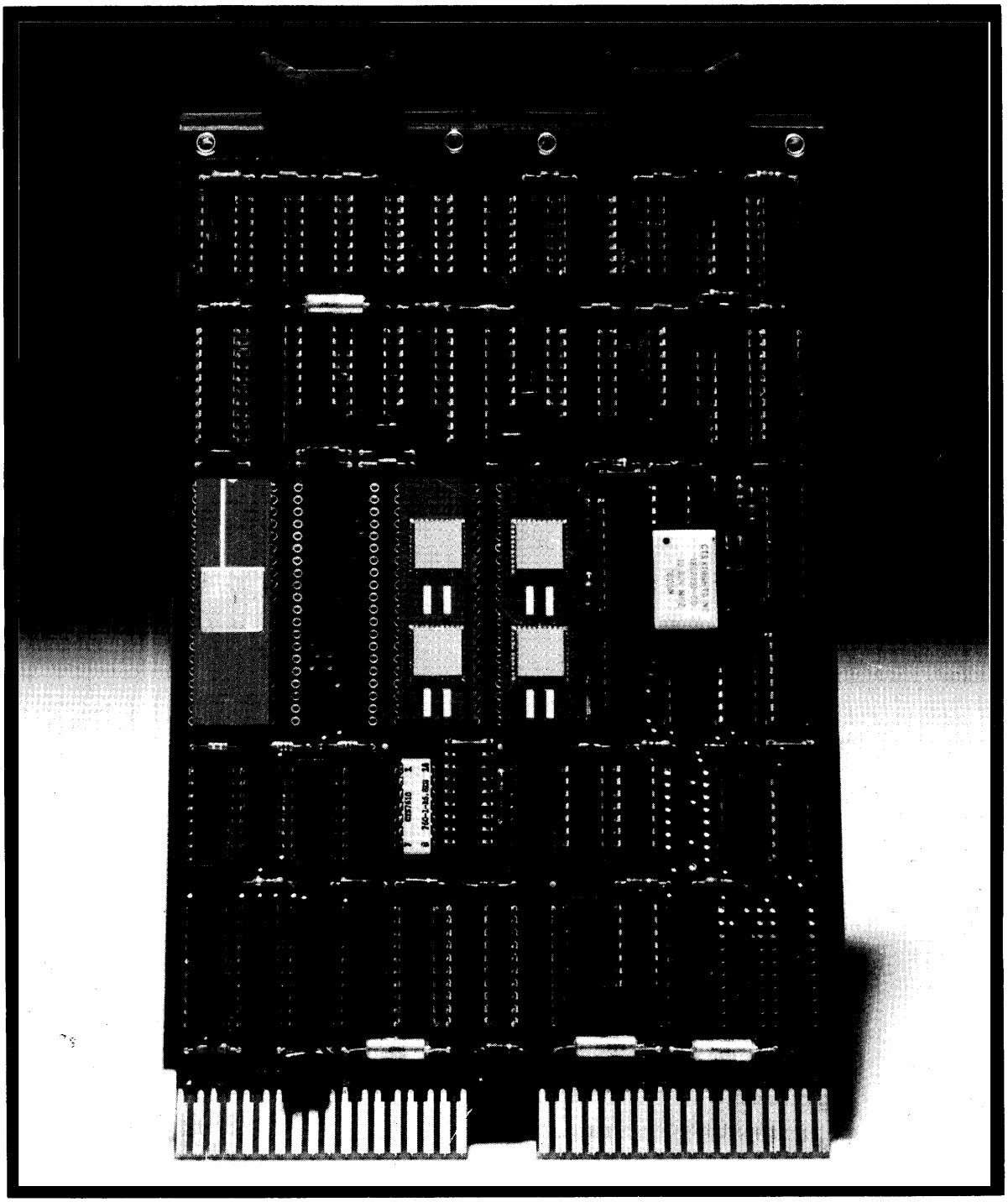
KDF11-AA USER'S GUIDE



digital

PRELIMINARY

EK-KDF11-UG-PR2

KDF11-AA
USER'S GUIDE

digital equipment corporation • marlboro, massachusetts

Preliminary Edition, January 1979
Revised Preliminary, March 1979

Printed in U.S.A.

6-80 15

PREFACE

This manual describes the KDF11-AA processor. The following versions of the KDF11 exist.

1. KDF11-AA processor with memory management unit (MMU)
2. KDF11-AB processor with MMU and floating point
3. KDF11-AC processor without MMU or floating point

The LSI-11/23 processor consists of one of the above KDF11 versions and can be ordered in conjunction with memory modules or in system boxes.

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

FIGURES

FIGURES (Cont)

FIGURES (Cont)

TABLES

CHAPTER 1
SPECIFICATIONS

1.1 INTRODUCTION

The KDF11-AA is a 16-bit, high-performance microprocessor contained on one dual-height multilayer module (M8186). Figure 1-1 shows the module with its major components highlighted. Utilizing the latest MOS/LSI technology, the KDF11-AA brings the full PDP-11/34 functionality to a microprocessor that communicates along the LSI-11 bus. The KDF11-AA contains memory management as a standard feature and offers floating point as an option (KEF11-A).

The processor uses the LSI-11 bus with a new 4-level interrupt bus protocol and parity check feature. The KDF11-AA is compatible with existing LSI-11 processors and devices.

The LSI-11 bus was built around LSI technology requirements consistent with low cost, high performance and small board form factors. Low cost and high performance are realized, in part, by using multifunction lines such as the data/address lines (DAL) that reduce the number of pins to the bus. Other lines, such as the I/O page address decode line, eliminate hardware by removing the need for identical page decoders on each interface module. A detailed description of the LSI-11 bus is contained in Chapter 4.

The KDF11-AA is software-compatible with the PDP-11 family. A wide range of software is available, including programming languages, diagnostic software, and operating systems.

1.2 FEATURES

The KDF11-AA contains the following features.

- Four-level vectored interrupts provide for fast interrupt response without device polling.
- Memory management for 128K of protected, multiuser program space.
- Memory parity errors are recognized during every data-in bus cycle.
- Over 400 instructions for powerful and convenient programming.
- 16-bit word or 8-bit byte addressable locations.
- Eight internal general-purpose registers for use as accumulators and for operand addressing.
- Stack processing for easy handling of structured data, subroutines, and interrupts.

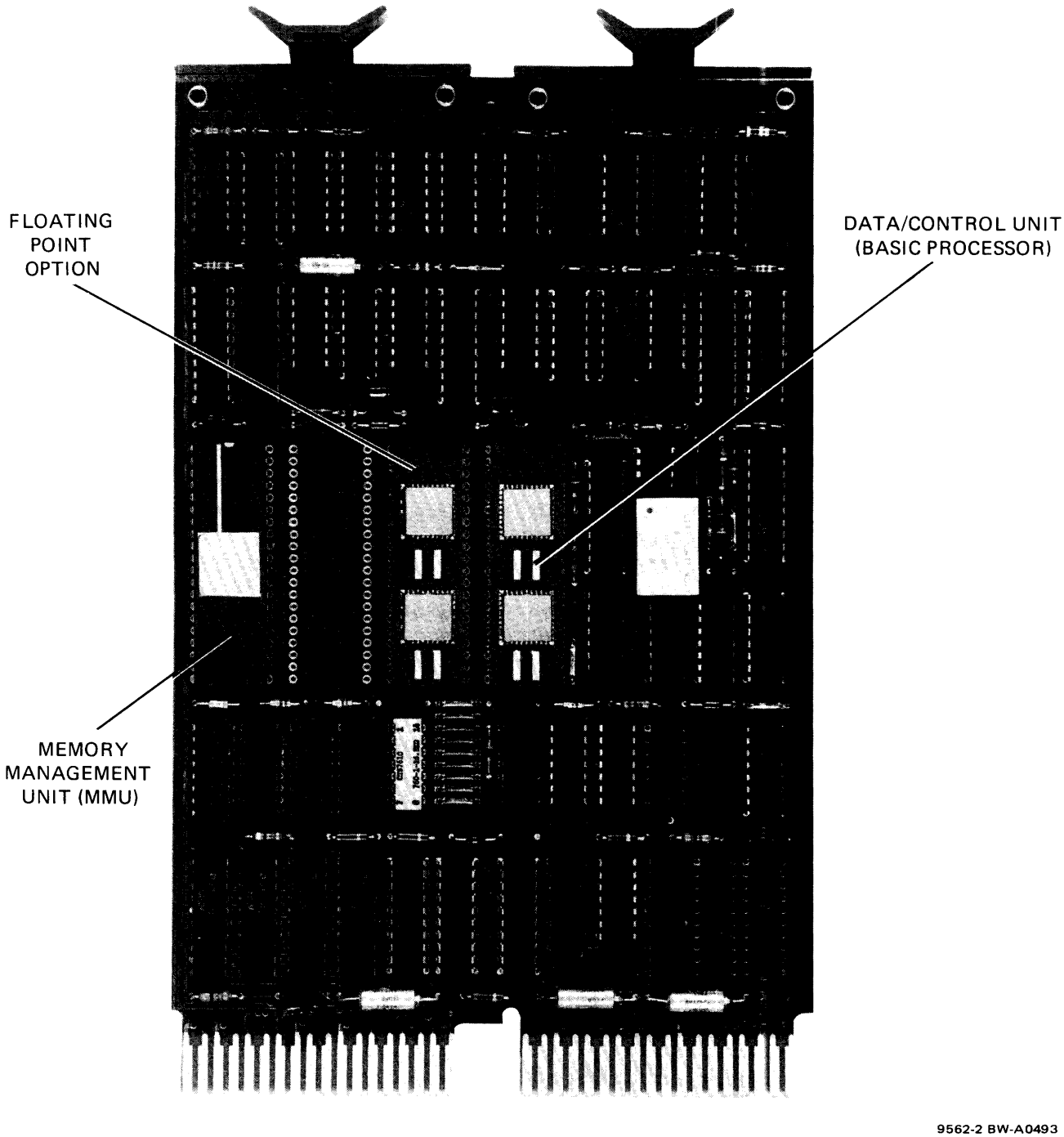


Figure 1-1 KDF11-AA Processor Module (M8186)
(Shown with Optional Floating Point)

- Asynchronous bus operation allows processor and system components (memory and peripherals) to run at their highest possible speed.
- Direct memory access (DMA) allows peripherals to access memory without interrupting processor operation.
- Modular component design allows systems to be easily configured and upgraded.
- Power fail and automatic restart hardware detect and protect against ac power fluctuations.
- Compact, double-height module size for versatile packaging.
- ODT console emulator for ease of program debugging.

1.3 SPECIFICATIONS

Identification	M8186
Size	Double
Dimensions	13.34 cm X 21.59 cm (5.25 in X 8.5 in)
Power Requirements	+5 V ± 5%, 2.0 A +12 V ± 5%, 0.2 A
Bus Loads	ac 2 unit loads dc 1 unit load
Environmental	
Storage	40° C to 65° C (104° F to 149° F) 10% to 90% relative humidity, noncondensing
Operating	5° C to 60° C, (41° F to 140° F) Maximum outlet temperature rise of 5° C (9° F) above 60° C (140° F) Derate maximum temperature by 1° C (1.8° F) for each 305 m (1000 ft) above 2440 m (8000 ft). 10% to 90% relative humidity, noncondensing

Timing (Based on 300 ns CPU microcycle time)

(Refer to Appendix B for detailed listing of instruction times.)

Interrupt Latency (based on MSV11-D without parity, add 500 ns worst case with parity)	
Worst Case	55.7 microseconds (for infrequently used instructions)
	10.8 microseconds (for more frequently used group)
Typical	6.0 microseconds
Interrupt Service Time	8.2 microseconds
DMA Latency	3.49 microseconds (worst case)

1.4 PROCESSOR HARDWARE

The KDF11-AA processor is implemented using three chips. Two MOS/LSI chips, data and control, implement the basic processor. The memory management unit (MMU), the third chip, provides a PDP-11/34 software-compatible memory management scheme.

The data chip (DC302) performs all arithmetic and logical functions, handles data and address transfers with the external world, and coordinates most interchip communication. The control chip (DC303) does microprogram sequencing for PDP-11 instruction decoding and contains the control store ROM. The data and control chips are both contained in one 40-pin package (refer to Figure 1-1). The MMU chip (DC304) contains the registers for 18-bit memory addressing and also includes the FP11 floating point registers and accumulators. Optional floating point requires the MMU chip. Data and control chips do not need the MMU chip for 16-bit addressing.

1.4.1 General-Purpose Registers

The data chip contains eight 16-bit general-purpose registers that provide for a variety of functions. These registers can serve as accumulators, index registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Arithmetic operations can be from one general register to another, from one memory location or device register to another, between memory locations, or between a device register and a general register. Figure 1-2 identifies the eight 16-bit general registers R0 through R7.

Registers R6 and R7 are dedicated. R6 serves as the stack pointer (SP) and contains the location (address) of the last entry in the stack. Register R7 serves as the processor's program counter (PC) and contains the address of the next instruction to be executed. It is normally used for addressing purposes only and not as an accumulator. Register operations are internal to the processor and do not require bus cycles (except for instruction fetch); all memory and peripheral device data transfers do require bus cycles and longer execution time. Thus, general registers used for

processor operations result in faster execution times. The bus cycles required for memory and device references are described in Paragraph 1.4.2.

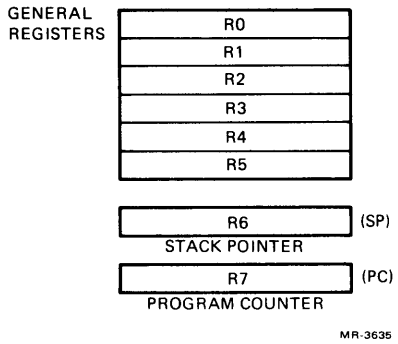


Figure 1-2 General Register

1.4.2 Bus Cycles

The bus cycles (with respect to the processor) are as follows.

DATI	Data word transfer input	Equivalent to read operation
DATO	Data word transfer output	Equivalent to word write operation
DATOB	Data word transfer output	Equivalent to byte write operation
DATIO	Data word transfer input followed by word transfer output	Equivalent to word read/ modify/write
DATIOB	Data word transfer input followed by byte transfer output	Equivalent to byte read/ modify/write

Every processor instruction requires one or more bus cycles. The first operation required is a DATI, which fetches an instruction from the location addressed by the program counter (R7). If no further operands are referenced in memory or in an I/O device, no additional bus cycles are required for instruction execution. If memory or a device is referenced, however, one or more additional bus cycles is required. DMA operations may occur between individual bus cycles, since these operations do not change the state of the processor.

Note the distinction between interrupts and DMA operations: interrupts, which may change the state of the processor, can occur only between processor instructions. For more details on bus operations refer to Chapter 4.

1.4.3 Addressing Memory and Peripherals

The KDF11-AA processor uses 16-bit data paths throughout. These same data paths are also used to construct operand and instruction addresses. Octal notation is used to describe information on the data paths.

A processor word is divided into a high byte and a low byte as shown in Figure 1-3.

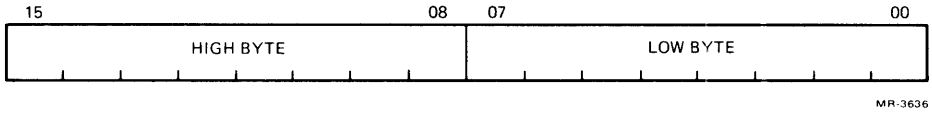


Figure 1-3 High and Low

Word addresses are always even-numbered. Byte addresses can be either even- or odd-numbered. Low bytes are stored at even-numbered memory locations and high bytes at odd-numbered memory locations. Thus, it is convenient to view the memory as shown in Figure 1-4.

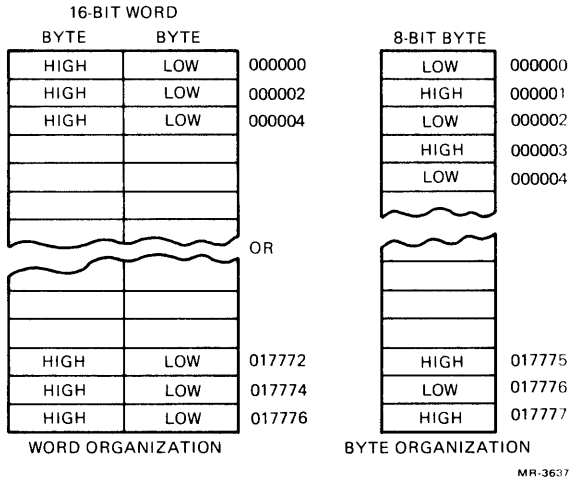


Figure 1-4 Word and Byte Addresses for First 4K

The full 16-bit data path allows a program to specify operand addresses (i.e., virtual addresses) anywhere within a 64K byte range or 32K word range. This virtual address range is fixed by the instruction format and cannot be changed by the user.

For applications that require more than 32K words of physical address, such as multiprogramming and/or timesharing applications, two additional addressing bits are available. These bits allow up to 128K memory words to be physically addressed by the processor. This additional addressing capability is part of the standard memory management within the KDF11-AA architecture.

1.4.4 Memory Management

Memory management has the following three major features.

1. Two software modes that are useful for multiuser (timesharing) systems
2. Extended physical addressing (greater than 32K words, up to 128K words) for allowing more than one program to reside in memory at the same time
3. Memory protection for controlling user program access to system resources (e.g., memory, I/O)

The first feature has two software modes, kernel and user. Kernel mode is employed by the operating system to control system resources and allows full privileges, while user mode is employed for executing a user program and restricts processor privileges (e.g., HALT instruction cannot be executed). The second feature utilizes mapping registers to map the 32K-word virtual address space anywhere in the 128K-word physical address space. The third feature allows restricted access to virtual memory pages (a page is between 0 and 4K words long). This permits the operating system software rather than user programs to control system resources. Chapter 8 contains a complete discussion of memory management.

1.4.5 Processor Status Word (PS)

The processor status word (PS) is in the data chip and contains information on the current processor status. As shown in Figure 1-5, this includes: the condition codes describing the arithmetic or logical results of the last instruction, a trace bit that forces a trap at the end of instruction execution (used during program debug), the current processor priority, an indicator of the previous memory management mode, and an indicator of the current memory management mode.

1.4.5.1 Condition Codes (PS bits <3:0>) - The condition codes contain information on the result of the last CPU operation. The bits are set after execution of all arithmetic or logical single-operand or double-operand instructions. The bits are set as follows.

N = 1 if the result was negative.

Z = 1 if the result was 0.

V = 1 if the operation resulted in an arithmetic overflow.

C = 1 if the operation resulted in a carry from the MSB (most significant bit) or a 1 was shifted from MSB or LSB (least significant bit).

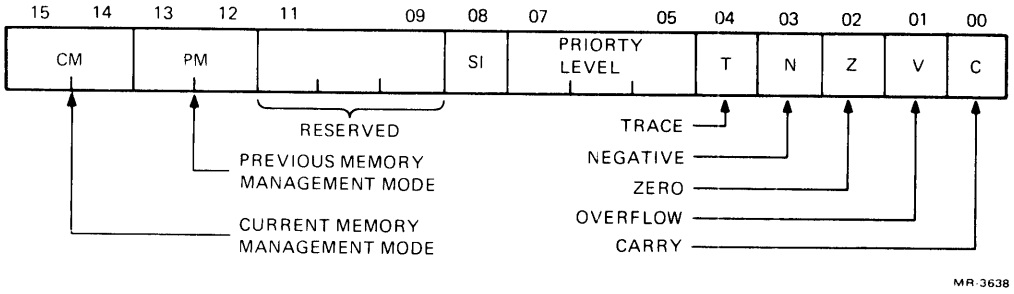


Figure 1-5 Processor Status Word (PS)

1.4.5.2 Trace Bit (PS bit 4) - The trace bit is used in debugging programs since it allows programs to be single-instruction stepped.

1.4.5.3 Priority Level (PS bits <7:5>) - These bits are used by software to determine which interrupts will be processed.

Octal Value of PS<7:5>	Interrupt Level Acknowledged*
7	none
6	7,
5	7, 6,
4	7, 6, 5,
3	7, 6, 5, 4
2	7, 6, 5, 4
1	7, 6, 5, 4
0	7, 6, 5, 4

*Higher levels acknowledged first.

1.4.5.4 Suspended Instruction (SI) (PS bit 8) - This bit is reserved for DIGITAL use and is intended for future optional instruction sets. This bit is read/write and has no protection mechanism. Refer to Paragraph 8.3.3.2 for more details.

1.4.5.5 Previous Mode (PS bits <13:12>) - These bits are used with memory management to indicate what the last memory management mode was. They are read/write bits and are present even without the memory management option.

1.4.5.6 Current Mode (PS bits <15:14>) - These bits indicate what the present memory management mode is. They are read/write and are present even without the memory management option.

1.5 INSTRUCTION SET

The KDF11-AA instruction set provides over 400 powerful instructions. As a comparison, consider that most other (i.e., accumulator-oriented) 16-bit processors require three separate instructions to execute a common double-operand instruction (e.g., ADD).

Conventional Approach

LDA A Load contents of memory location A into accumulator.

ADD B Add contents of memory location B to accumulator.

STA B Store result at location B.

By contrast, the KDF11-AA can fetch both operands, execute, and store the result in one instruction.

KDF11-AA Approach

ADD A, B Add contents of location A to location B; store results at location B.

This greater efficiency not only saves memory space and time, but also improves processor speed since fewer instruction fetches are required.

Another major advantage to the KDF11-AA instruction set is the absence of special-purpose input/output instructions. Special I/O instructions are unnecessary since peripheral device registers are accessed in the same way as main memory locations. This approach to handling I/O devices allows the normal instruction set to be used to test and/or manipulate the various I/O device register bits. For example, a compare instruction can test status bits directly in the I/O device register without bringing them into memory or disturbing any of the general registers; control bits can be set, cleared, or shifted as is most convenient; and peripheral data can be arithmetically or logically altered when received at the device register and before being stored in memory. Refer to Chapter 7 for a complete description of the instruction set and its utilization.

Addressing Modes
Much of the flexibility of the KDF11-AA is derived from its wide range of addressing capabilities. Addressing modes include sequential forward or backward addressing, address indexing, indirect addressing, absolute 16-bit word and 8-bit byte addressing, and stack addressing. Variable-length instruction formatting allows a minimum number of words to be used for each addressing mode. The result is efficient use of program storage space. For more details on addressing modes refer to Chapter 6.

1.6 FLOATING POINT OPTION

Forty-six floating point instructions are available as a microcode option (KEF11-A) on the KDF11-AA processor. These instructions supplement the integer arithmetic instructions (e.g., MUL, DIV, etc.) in the basic instruction set. The floating point option allows floating point operations to be executed 5 to 10 times faster than equivalent software routines and provides for both single precision (32-bit) and double precision (64-bit) operands. This option also conserves memory space, since floating point routines are executed in microcode instead of software. This option implements the same floating point instruction set found on the PDP-11/34, -11/60, and -11/70. For a complete description refer to Chapter 9.

1.7 MEMORIES AND PERIPHERALS

DIGITAL provides a wide range of memories and peripherals to allow maximum flexibility when configuring systems. A detailed list and description can be found in the Memories and Peripherals handbook in the Microcomputer Handbook Series.

1.8 RELATED DOCUMENTS

The following is a list of documents containing additional information of possible interest to KDF11-AA microprocessor users.

Title	Document Number
Memories and Peripherals Handbook	EB 15114 78
Microcomputer Processors Handbook	EB 15115 78
PDP-11 Processor Handbook	EB-09340-20
PDP-11 Software Handbook	EB-09798-20

These documents can be ordered from:

Digital Equipment Corporation
444 Whitney Street
Northboro, MA 01532

Attention: Communications Services (NR2/M15)
Customer Services Section

CHAPTER 2
INSTALLATION

2.1 INTRODUCTION

Items that must be considered when installing a KDF11-AA processor include the following.

1. Configuration of jumpers for operation of user-selectable features
2. Selection of an LSI-11 bus-compatible backplane, mounting, and installation
3. Selection of LSI-11 bus-compatible options and accessories
4. Knowledge of system differences if replacing an LSI-11 or LSI-11/2 with a KDF11-AA

This chapter discusses these procedures in detail. Refer to Paragraph 1.8 for the order number of documents referred to in this chapter.

2.2 JUMPER CONFIGURATIONS

Several jumpers on the processor module provide user-selectable features. Table 2-1 lists the jumper configurations and Figure 2-1 shows the location of these jumpers. Paragraphs 2.2.1 through 2.2.6 describe the jumper functions. Jumpers not discussed are reserved for use by DIGITAL and should not be used.

Table 2-1 Jumper Configurations

Jumper	Name	In	Out
W1	Master clock	Enable internal master clock	Do not remove Manufacturing use only
W2	Reserved for DIGITAL use	Factory-installed	Do not remove
W4	Event line enable	Disabled	Enabled
W5, W6	Power-up mode selector	See text	See text
W7	Halt/trap option	Trap to 10_8 on halt	Enter console ODT on halt

Table 2-1 Jumper Configurations (Cont)

Jumper	Name	In	Out
W8	Conventional bootstrap start address, enable if power-up mode 2 is selected	Power-up to bootstrap address 173000_8	Power-up to boootstrap address selected by jumpers W9-W15
W9-W15	User-selectable bootstrap starting address for power-up mode 2	See text	See text
W16	Reserved for DIGITAL use	Must be installed	Do not remove
W17	Reserved for DIGITAL use	Must be installed	Do not remove
W18	Reserved for DIGITAL use	Must be installed	Do not remove

2.2.1 Master Clock - W1

The internal 13.8 MHz oscillator is disconnected from the clock circuitry if W1 is removed. This jumper is used by DIGITAL manufacturing and is not to be removed by the user.

2.2.2 Event Line - W4

The bus signal BEVENT L causes the event line flip-flop to be set. When the processor enters the service state the request will be honored if the PS<07:05> is 5 or less. (BEVENT is a level 6 interrupt.) This causes the microcode to clear the request flip-flop and trap to the line clock vector (location 100_8). If W4 is inserted, the request flip-flop is disabled and therefore the BEVENT signal is disabled. Users would disable BEVENT, which is normally used as a 60 Hz real-time clock, if they have a programmable clock on the LSI-11 bus.

NOTE

The LSI-11 and LSI-11/2 processors treat a BEVENT interrupt at a different priority level than the KDF11-AA. Refer to Paragraph 2.6.

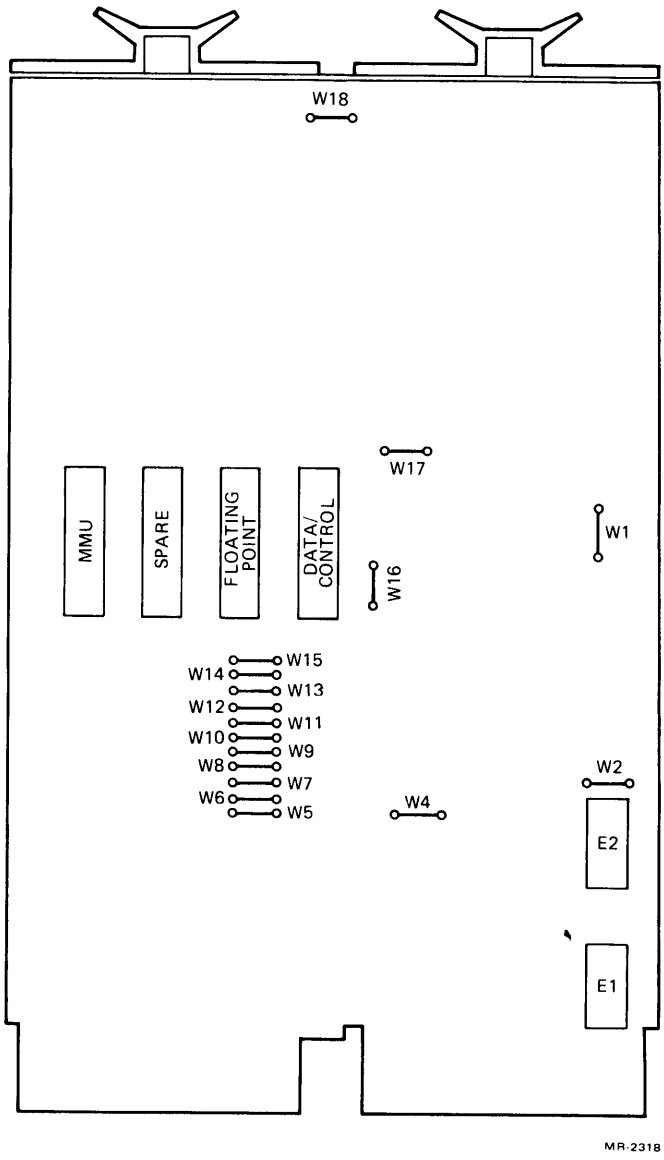


Figure 2-1 KDF11-AA Jumper Locations

2.2.3 Power-Up Mode Selection - W5 and W6

Four power-up modes are available for user selection. Selection is made by removal or insertion of jumpers W5 and W6 as shown in the following listing.

Mode	Name	W6*	W5*
0	PC@24, PS@26	R	R
1	Console ODT	R	I
2	Bootstrap	I	R
3	Extended microcode	I	I

Only the power-up mode is affected, not the power-down sequence. The following paragraphs describe the sequence of events after executing common power-up, when selecting each of the four modes. The state of bus signal BHALT L is significant in power-up mode operation.

2.2.3.1 Power-Up Mode 0 (PC@24, PS@26) - This mode causes the microcode to fetch the contents of memory locations 24_8 and 26_8 and loads their contents into the PC and PS, respectively. The microcode then examines BHALT L. If BHALT L is asserted, the processor enters console ODT mode. If BHALT L is not asserted, the processor begins program execution by fetching an instruction from the location pointed to by the PC. This mode is useful when power fail/auto restart capability is desired.

2.2.3.2 Power-Up Mode 1 (Console ODT) - This mode causes the processor to enter console ODT mode immediately after power-up regardless of the state of any service signals. This mode is useful in a program development or hardware debug environment, giving the user immediate control over the system after power-up.

2.2.3.3 Power-Up Mode 2 (User Bootstrap Starting Address Shown by W8-W15) - This mode causes the processor to internally generate a bootstrap starting address by looking at jumpers W8 through W15 (see Paragraphs 2.2.5 and 2.2.6). This address is loaded into the PC. The processor sets the PS to 340_8 (PS<07:05> = 7_8) to inhibit interrupts before the processor is ready for them. If BHALT L is asserted, the processor enters console ODT mode. If not, the processor begins execution by fetching an instruction from the location pointed to by the PC. This mode is useful for turnkey applications where the system automatically begins operation without operator intervention.

2.2.3.4 Power-Up Mode 3 (User Microcode - For Future Use) - This mode causes the microcode to jump to optional control chip 37_8, location 76_8, and begin microcode execution. This mode is reserved for future DIGITAL use and is not recommended for customer usage. If it is erroneously selected, the processor will treat it as a reserved instruction trap to location 10_8.

*R = jumper removed; I = jumper installed.

2.2.4 Halt/Trap Option - W7

If the processor is in kernel mode and decodes a HALT instruction, BPOK H is tested. If BPOK H is negated, the processor will continue to test for BPOK H. The processor will perform a normal power-up sequence if BPOK H becomes asserted sometime later. If BPOK H is asserted after the HALT instruction decode, the halt/trap jumper (W7) is tested. If the jumper is removed, the processor enters console ODT mode. If the jumper is installed a trap to location 10_8 will occur.

NOTE

In user mode a HALT instruction execution will always result in a trap to location 10_8.

This feature is intended for situations, such as unattended operation, where recovery from erroneous HALT instructions is desirable.

2.2.5 Starting Address 173000_8 - W8

When power-up mode 2 is selected, the processor examines jumper W8 to determine the starting address for program execution. If W8 and a compatible bootstrap module such as BDV-11 are installed in the system, the microcode will begin execution at 173000_8 (conventional starting address for DIGITAL systems). If W8 is removed, a trap to 4_8 (nonexistent address) will occur. If W8 is removed, the processor looks at jumpers W9 through W15 for the starting address.

2.2.6 Selectable Starting Address - W9 through W15

If the user wishes to start execution from an address other than 173000_8, jumpers W9 through W15 can be used to specify the high byte <15:09> of the starting address. Jumpers W15 through W9 correspond to address bits <15:09>, respectively. Bits <08:00> of the starting address are set to 0 by the processor. Jumpers are installed for logic 1, removed for logic 0. The starting address can reside on any 256-word boundary in the lower 32K of memory address space.

2.3 MODULE CONTACT FINGER IDENTIFICATION

DIGITAL plug-in modules, including the KDF11-AA, all use the same contact finger (pin) identification system. The LSI-11 bus is based on the use of double-height modules that plug into a 2-slot bus connector. Each slot contains 36 lines (18 each on component and solder sides of the circuit board).

Slots, shown as row A and row B in Figure 2-2, include a numeric identifier for the side of the module. The component side is designated side 1 and the solder side is designated side 2. Letters ranging from A through V (excluding G, I, O, and Q) identify a particular pin on a side of a slot. A typical pin is designated as follows.

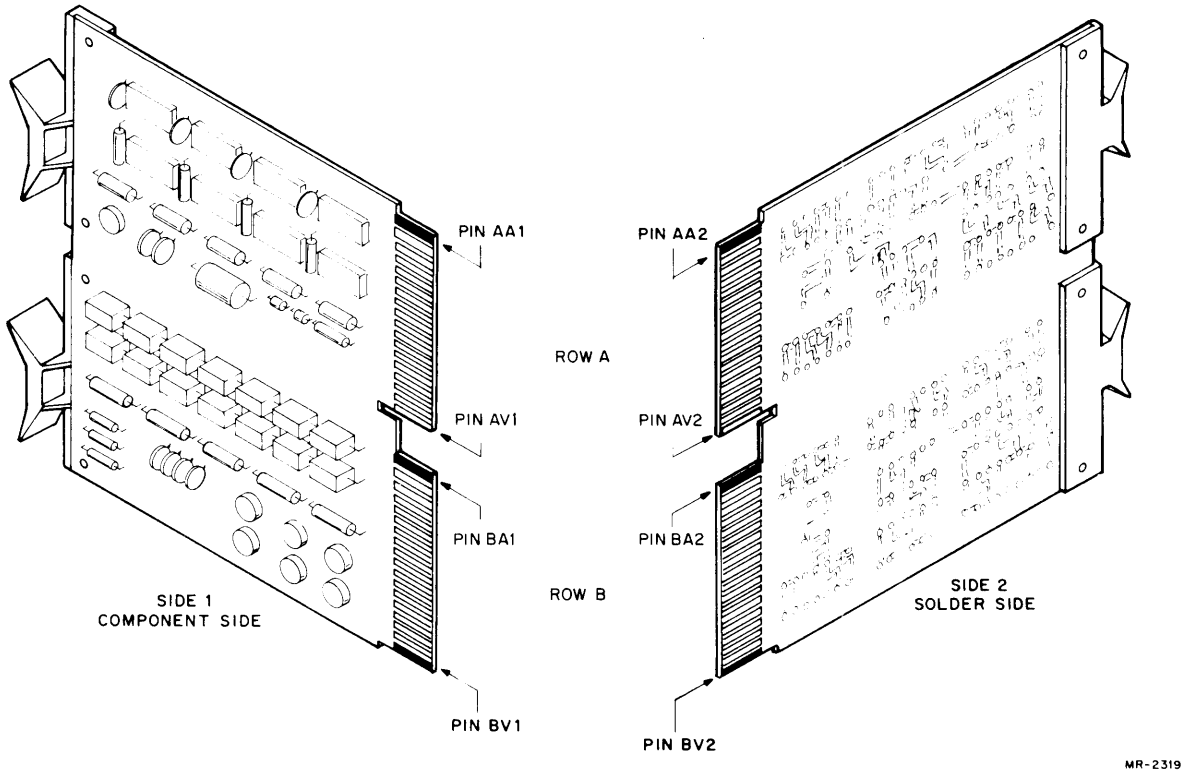
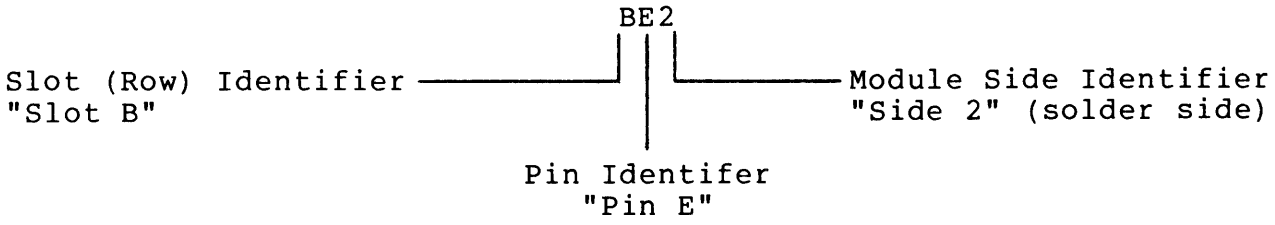


Figure 2-2 Double-Height Module Contact Finger Identification



The positioning notch between the two rows of pins mates with a protrusion on the connector block for correct module positioning.

2.4 BACKPLANE PIN ASSIGNMENTS AND THEIR KDF11-AA UTILIZATION

When configuring a system with the KDF11-AA, the module may be inserted in one of several available backplanes. Refer to Paragraph 2.5 for information on the types available. Using the H9270 backplane as an example, Figure 2-3 shows the backplane pin identification. Individual connector pins shown are viewed from the underside (wiring side). Only pins for one bus location (two slots) are shown in detail. This pin pattern is repeated eight times on this backplane, allowing the user to install several double-height modules.

Table 2-2 lists the backplane pin assignments and describes their use by the KDF11-AA processor module.

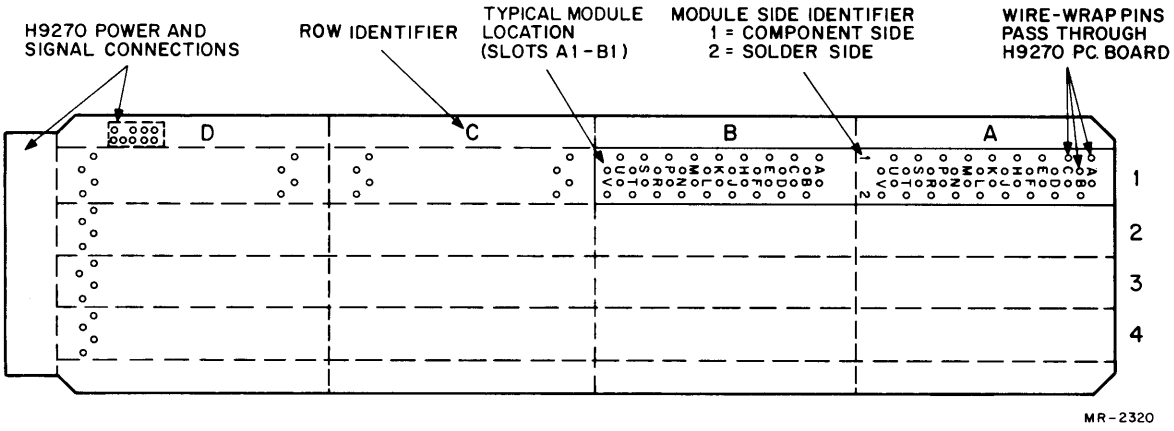


Figure 2-3 H9270 Backplane Pin Identification (Pin Side View Shown)

Table 2-2 Backplane Pin Assignments/ KDF11-AA Processor Utilization

Bus Pin	Mnemonic	Description
AA1	BIRQ5 L	Interrupt Request priority level 5
AB1	BIRQ6 L	Interrupt Request priority level 6
AC1 AD1	BDAL16 L BDAL17 L	Extended address bits (also used for parity)
AE1	SSI	Single-step input (Reserved for DIGITAL use)
AF1	SRUN L	Run light signal
AH1	SRUN L	Run light signal
AJ1	GND	Ground - System signal ground and dc return.
AK1 AL1	MSPAREA MSPAREB	Maintenance Spare - Normally connected together on the backplane at each option location (not bused connection).
AM1	GND	Ground - System signal ground and dc return.

Table 2-2 Backplane Pin Assignments/
KDF11-AA Processor Utilization (Cont)

Bus Pin	Mnemonic	Description
AN1	BDMRL	Direct Memory Access (DMA) Request - A device asserts this signal to request bus mastership. The processor arbitrates bus mastership between itself and all DMA devices on the bus. If the processor is not bus master (it has completed a bus cycle and BSYNC L is not being asserted by the processor), it grants bus mastership to the requesting device by asserting BDMGO L. The device responds by negating BDMR L and asserting BSACK L.
AP1	BHALT L	Processor Halt - When BHALT L is asserted, the processor responds by going into console ODT mode.
AR1	BREF L	Memory Refresh - This signal is not used by KDF11-AA; however, it is terminated.
AS1	+12B	+12 V Battery Power - Secondary +12 V power connection. Battery power can be used with certain devices. Not used by KDF11-AA.
AT1	GND	Ground - System signal ground and dc return.
AU1	PSPARE1	Spare (Not assigned. Customer usage not recommended.)
AV1	+5B	+5 V Battery Power - Secondary +5 V power connection. Battery power can be used with certain devices. Not used on KDF11-AA.
BA1	BDCOK H	DC Power OK - Power supply-generated signal that is asserted when there is sufficient dc voltage available to sustain reliable system operation. This signal is also driven by the "wake-up" circuit on the KDF11-AA.
BB1	BPOK H	Power OK - Asserted by the power supply when primary power is normal. When negated during processor operation, a power fail trap sequence is initiated.
BC1	MMU DAL18 H	(Reserved for DIGITAL use)
BD1	MMU DAL19 H	
BE1	MMU DAL20 H	
BF1	MMU DAL21 H	
BH1		Clock Disable - Reserved for DIGITAL use

Table 2-2 Backplane Pin Assignments/ KDF11-AA Processor Utilization (Cont)

Bus Pin	Mnemonic	Description
BJ1	GND	Ground - System signal ground and dc return.
BK1 BL1	MSPAREB MSPAREB	Maintenance Spare - Normally connected together on the backplane at each option location (not a bused connection).
BM1	GND	Ground - System signal ground and dc return.
BN1	BSACK L	This signal is asserted by a DMA device in response to the processor's BDMGO L signal, indicating that the DMA device is bus master.
BP1	BIRQ7 L	Interrupt request priority level 7
BR1	BEVNT L	External Event Interrupt Request - When asserted, the processor arbitrates as a level 6 interrupt. A typical use of this signal is a line time clock interrupt.
BS1	PSPARE 4	Spare (Not assigned. Customer use not recommended.
BT1	GND	Ground - System signal ground and dc return.
BU1	PSPARE2	Spare (Not assigned. Customer use not recommended.)
BV1	+5	+5 V Power - Normal +5 Vdc system power
AA2	+5	+5 V Power - Normal +5 Vdc system power
AB2	-12	-12 V Power - -12 Vdc (optional) power for devices requiring this voltage. **NOTE** **Modules that require negative voltages contain an inverter circuit (on each module) which generates the required voltage(s). Hence, -12 V power is not required with DIGITAL-supplied options including the KDF11-AA processor.**
AC2	GND	Ground - System signal ground and dc return
AD2	+12	+12 V Power - +12 Vdc system power

Table 2-2 Backplane Pin Assignments/
KDF11-AA Processor Utilization (Cont)

Bus Pin	Mnemonic	Description
AE2	BDOUT L	Data Output - BDOUT, when asserted, implies that valid data is available on BDAL<0:15> L and that an output transfer, with respect to the bus master device, is taking place. BDOUT L is deskewed with respect to data on the bus. The slave device responding to the BDOUT L signal must assert BRPLY L to complete the transfer.
AF2	BRPLY L	Reply - BRPLY L is asserted in response to BDIN L or BDOUT L and during IAK transaction. It is generated by a slave device to indicate that it has placed its data on the BDAL bus or that it has accepted output data from the bus.
AH2	BDIN L	Data Input - BDIN L is used for two types of bus operation: 1. When asserted during BSYNC L time, BDIN L implies an input transfer with respect to the current bus master, and requires a response (BRPLY L). BDIN L is asserted when the master device is ready to accept data from a slave device. 2. When asserted without BSYNC L, it indicates that an interrupt operation is occurring. The master device must deskew input data from BRPLY L.
AJ2	BSYNC L	Synchronize - BSYNC L is asserted by the bus master device to indicate that it has placed an address on the bus. The transfer is in process until BSYNC L is negated.
AK2	BWTBT L	Write/Byte - BWTBT L is used in two ways to control a bus cycle: 1. It is asserted during the leading edge of BSYNC L to indicate that an output sequence is to follow (DATO or DATOB), rather than an input sequence. 2. It is asserted during BDOUT L, in a DATOB bus cycle, for byte addressing.

Table 2-2 Backplane Pin Assignments/ KDF11-AA Processor Utilization (Cont)

Bus Pin	Mnemonic	Description
AL2	BIRQ4 L	Interrupt request priority level 4
AM2	MMU STR H	(Reserved for DIGITAL use)
AN2	BIAKO L	Interrupt Acknowledge Output - This signal is generated by the processor in response to an interrupt request (BIRQ L). The processor asserts BIAKO L, which is routed to the BIAKI L pin of the first device on the bus. Refer to Chapter 3 for the proper interrupt protocol.
AP2	BBS7 L	Bank 7 Select - The bus master asserts BBS7 L when an I/O device address (upper 4K address range) is placed on the bus. BSYNC L is then asserted and BBS7 L remains active for the duration of the addressing portion of the bus cycle.
AR2	UB MAP L	(Reserved for DIGITAL use)
AS2	BDMGO L	DMA Grant Output - This is the processor-generated daisy-chained signal that grants bus mastership to the highest priority DMA device along the bus. The processor generates BDMGO L, which is routed to the BDMGI L pin of the first device on the bus. If it is requesting the bus, it will inhibit passing BDMGO L. If it is not requesting the bus, it will pass the BDMGI L signal to the next (lower priority) device via its BDMGO L pin. The device asserting BDMR L is the device requesting the bus, and it responds to the BDMGI L signal by negating BDMR, asserting BSACK L, assuming bus mastership, and executing the required bus cycle.
AT2	BINIT L	Initialize - BINIT is asserted by the processor to initialize or clear all devices connected to the I/O bus. The signal is generated in response to a power-up condition (the negated condition of BDCOK H), or by executing a RESET instruction.

Table 2-2 Backplane Pin Assignments/ KDF11-AA Processor Utilization (Cont)

Bus Pin	Mnemonic	Description
AU2 AV2	BDAL0 L BDAL1 L	Data/Address Lines - These two lines are part of the 8-line data/address bus over which address and data information are communicated. Address information is first placed on the bus by the bus master device. The same device then either receives input data from, or outputs data to the addressed slave device or memory over the same bus lines.
BA2	+5	+5 V Power - Normal +5 Vdc system power
BB2	-12	-12 V Power - -12 Vdc (optional) power for devices requiring this voltage. Not used by KDF11-AA.
BC2	GND	Ground - System signal ground and dc return
BD2	+12	+12 V Power - +12 Vdc system power
BE2 BF2 BH2 BJ2 BK2 BL2 BM2 BN2 BP2 BR2 BS2 BT2 BU2 BV2	BDAL2 L BDAL3 L BDAL4 L BDAL5 L BDAL6 L BDAL7 L BDAL8 L BDAL9 L BDAL10 L BDAL11 L BDAL12 L BDAL13 L BDAL14 L BDAL15 L	Data/Address Lines - These 14 lines are part of the 8-line data/address bus previously described.

2.5 HARDWARE OPTIONS

KDF11-AA systems can be configured using a variety of backplanes, power supplies, enclosures, memories, peripherals, etc.

2.5.1 Backplanes

Any of the following LSI-11 bus-compatible backplanes can be used with the KDF11-AA.

1. H9270 - Accepts quad- or double-height modules
2. H9273-A - Accepts quad- or double-height modules
3. H9281 - Accepts double-height modules only
4. DDV11-B - Accepts quad- or double-height modules

Refer to the Memories and Peripherals handbook for a complete description of each backplane and installation information.

2.5.1.1 H9270 Backplane - The H9270 consists of an 8-slot backplane with a card guide assembly. As shown in Figure 2-4, this backplane is designed to accept up to eight double-height modules (including processor), four quad modules, or a combination of quad- and double-height modules. When used for bus expansion in multiple backplane systems, the H9270 provides space for up to six option modules, plus the required expansion cable connector module(s) and/or terminator module.

VIEW FROM MODULE SIDE OF BACKPLANE

PROCESSOR (HIGHEST PRIORITY LOCATION)	PROCESSOR OR OPTION 1	1
OPTION 3	OPTION 2	2
OPTION 4	OPTION 5	3
OPTION 7 (LOWEST PRIORITY LOCATION)	OPTION 6	4

MR-1152

Figure 2-4 H9270 Options Positions

2.5.1.2 H9273-A Backplane - The H9273-A backplane logic assembly consists of a 9 X 4 backplane (nine rows of four slots each) and a card frame assembly. Power and signals are supplied to the backplane through connectors J7 and J8.

The H9273-A backplane is designed to accept both double-height and quad-height modules with the exception of the MMV11-A core memory module. The backplane structure is unique in that it provides two distinct buses: the LSI-11 bus signals (slots A and B) and the CD bus (slots C and D). The connectors that comprise this backplane are arranged in nine rows. Each connector has two slots, each of which contains 36 pins, 18 on either side of the slot.

Three jumpers (W1, W2, and W3) are shown in Figure 2-5. Jumper W1 enables the line-time clock when inserted and disables it when removed.

NOTE

Only one BA11-N mounting box in any system may have the line-time clock enabled.

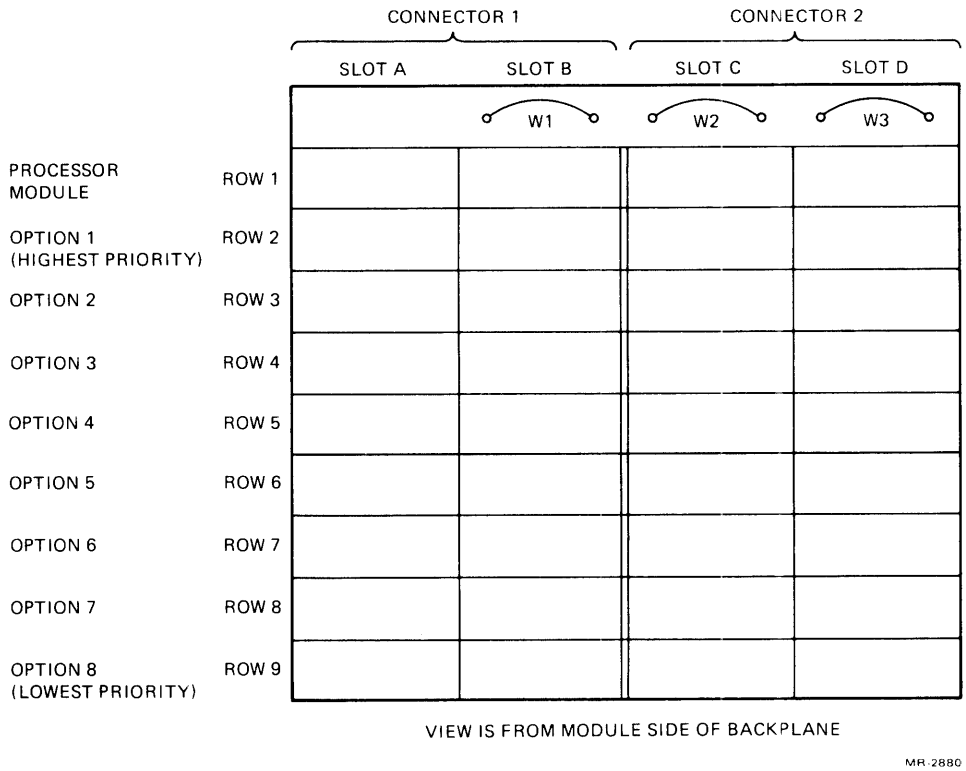


Figure 2-5 H9273-A Option Positions

When inserted, jumpers W2 and W3 allow the LSI-11 quad-height CPU to run in row 1. Jumpers W2 and W3 are removed when the backplane is used as an expansion backplane in a system.

The connectors designated "Connector 1" in Figure 2-5 are wired according to the LSI-11 bus specification. Slots A and B carry the LSI-11 bus signals and are termed the LSI-11 bus slots. The connectors designated "Connector 2" are wired for +5 V and ground, and have no connections to the LSI-11 bus; instead, C- and D-slot pins on side 2 of each row are connected to the C- and D-slot pins on side 1 in the next lower row. For details of the CD interconnection scheme see the Memories and Peripherals handbook.

2.5.1.3 H9281 Backplane (Figure 2-6) - The H9281 backplanes are designed to accept double-height modules only. The H9281 2-slot backplane is available in six options as listed below. These backplanes allow the user to configure compact LSI-11 bus systems that most efficiently utilize available system space.

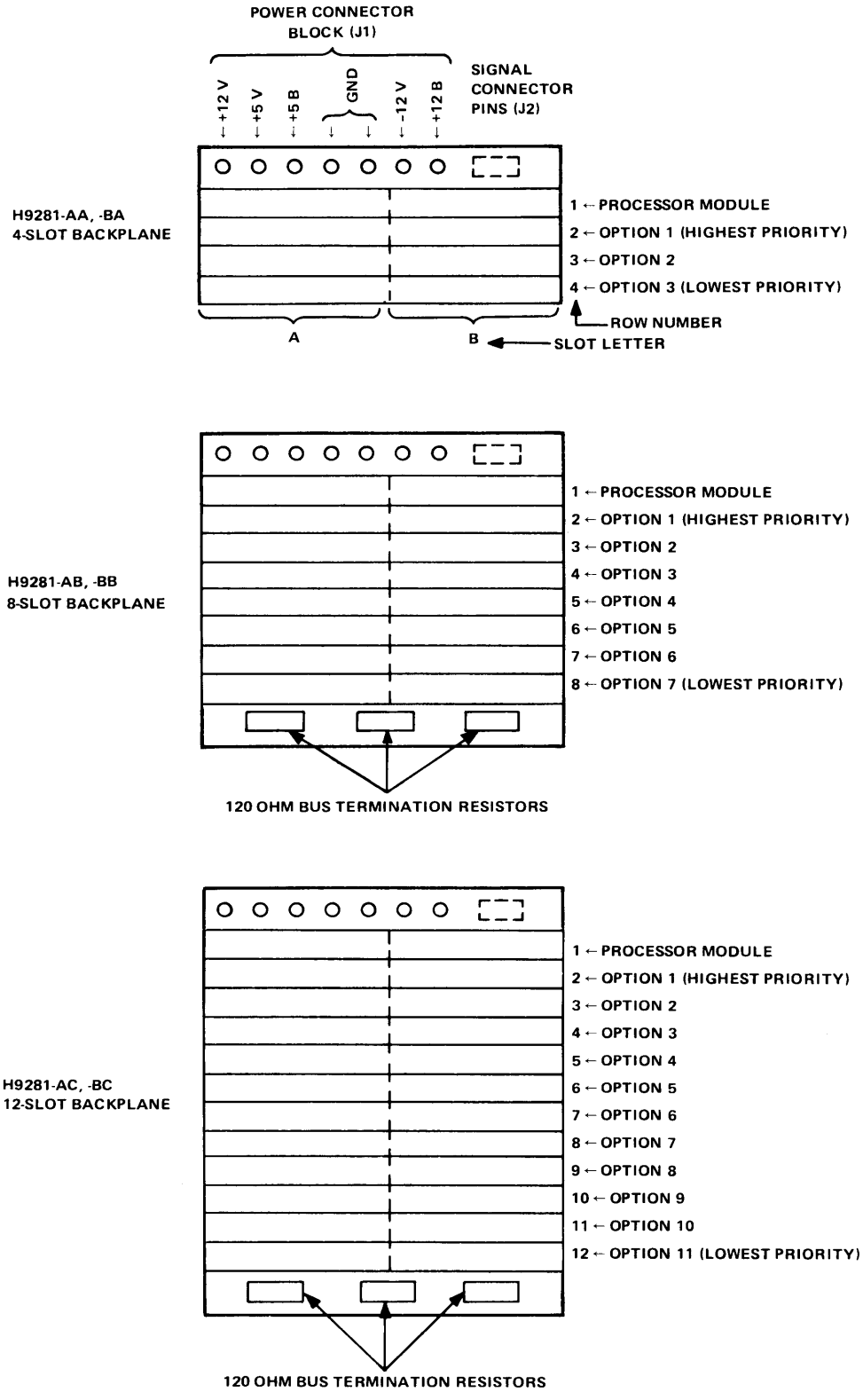


Figure 2-6 H9281 Option and Connector Locations (Module Side)

Backplane Option Designation	Description
H9281-AA	4-module backplane
H9281-AB	8-module backplane
H9281-AC	12-module backplane
H9281-BA	4-module backplane and card cage assembly
H9281-BB	8-module backplane and card cage assembly
H9281-BC	12-module backplane and card cage assembly

NOTE

Some options are too large to be installed in an H9281 backplane. Refer to Memories and Peripherals handbook for a complete list.

Bus Terminations

Backplane models H9281-AB, -BB, -AC, and -BC include 120 ohm bus termination resistors at the electical end of the bus; therefore, it is not necessary to install a separate 120 ohm bus terminator module in these backplanes.

2.5.1.4 DDV11-B Backplane - The DDV11-B is an optional LSI-11 bus expansion backplane for use when additional logic space is required. The DDV11-B is a 9 X 6, 54-slot backplane with a 9 X 4 slot section (18 individual double-height or 9 quad-height module slots) prebused specifically for LSI-11 bus signal and power and ground connections. The remaining 9 X 2 slot section is provided with +5 Vdc, GND, and -12 Vdc power connections only; this leaves the remaining pins free for use with any special double-height logic modules to be used in conjunction with the LSI-11 family of modules and bus requirements.

Module Slot Assignments

Figure 2-7 shows the slot location assignments of the DDV11-B. Rows A, B, C, and D are dedicated to the LSI-11 bus. Any module that conforms to the LIS-11 bus specifications may be used in this portion of the DDV11-B. The position numbers indicate the bus grant wiring scheme with respect to the processor module. The bus grant signals propagate through the slot locations in the position order shown in Figure 2-7 until they reach the requesting device. To provide bus grant signal continuity, any unused slots must be jumpered or unused locations must occur only in the highest position-numbered locations.

Rows E and F contain the 18 user-defined slots with power and ground connections provided.

2.5.1.5 Device Priority Within Backplanes - All LSI-11 bus backplanes are priority-structured. Daisy-chained grant signals for DMA and interrupt requests propagate away from the processor from the first (highest priority device) to successively lower

priority devices. Processor module locations and device (option) priorities are shown in Figures 2-4, 2-5, 2-6, and 2-7. For further discussion, see Paragraph 2.5.5.

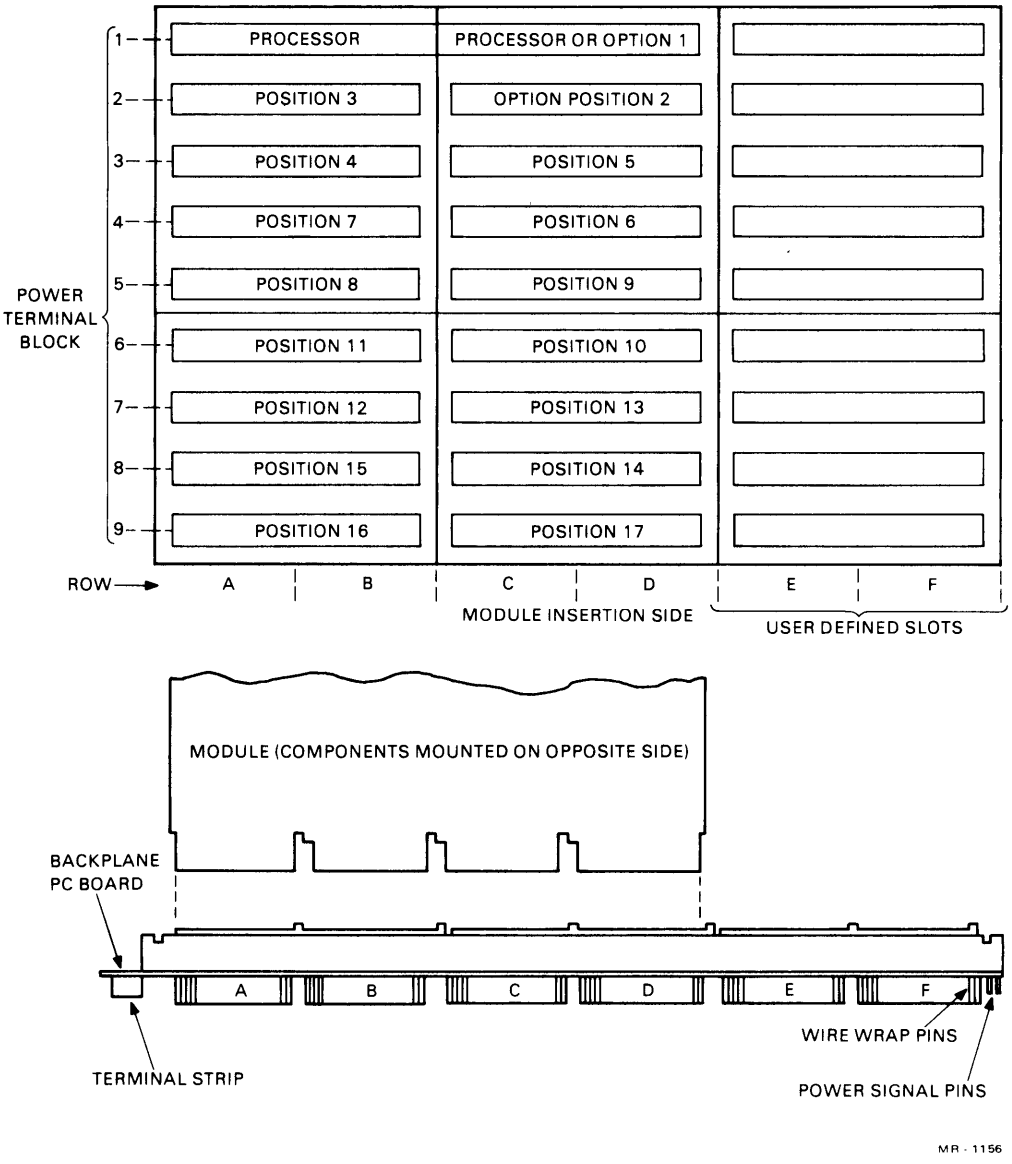


Figure 2-7 DDV11-B Module Installation and Slot Assignments

2.5.2 Power Supplies

Both the H780 and the H786 power supplies can be used when configuring a KDF11-AA system. The H780 power supply is described in detail in the Memories and Peripherals handbook. The H786 is not available separately, only as part of the BA11-N enclosure.

2.5.3 Enclosures

The BA11-M mounting box, which includes an H9270 backplane and an H780 power supply, or the BA11-N mounting box, which includes an H9273 backplane and an H786 power supply, can be used in a system with the KDF11-AA processor. The Memories and Peripherals handbook contains details on both boxes.

2.5.4 Memory Modules

Several memory modules are available for use with the KDF11-AA systems. However, modules such as MSV11-C or MSV11-D that perform memory refresh locally are required, since the KDF11-AA does not perform memory refresh itself. MSV11-C memories will work if provision is made for refresh with some other bus option such as REV11; however, this will degrade system performance and is not recommended. The Memories and Peripherals handbook contains further information on LSI-11 bus-compatible memories.

2.5.5 Peripheral Options

All LSI-11 bus-compatible peripheral devices may be used in KDF11-AA systems. DMA peripherals should be installed with the faster throughput devices physically closest to the processor and slower ones farther away. The user must ensure that faster devices have adequate access to the bus; otherwise, data drop errors may occur. Interrupt-driven peripherals can be installed in one of the following ways. If all peripherals use the single-level scheme, they must be installed with faster interrupting devices physically closest to the processor. All current DIGITAL LSI-11 bus peripheral devices must use this method. Future peripheral devices, or customer-designed devices, can take advantage of the new 4-level interrupt scheme. With the new scheme, peripherals that are designed to perform distributed interrupt arbitration, and that are on different interrupt levels, can be installed in any order. Multiple peripherals on the same request level and peripherals that do not perform distributed arbitration must be installed with the highest priority, or faster, devices closest to the processor. The Memories and Peripherals handbook contains more information on available devices and their installation. For further discussion of the 4-level interrupt system, see Paragraphs 4.4.2 and 4.4.3.

2.6 SYSTEM DIFFERENCES

A number of minor differences exist between the KDF11-AA processor and the LSI-11 (KD11-F) or LSI-11/2 (KD11-HA) processor. The following is a list of system differences that exist because of the KDF11-AA's advanced design.

1. KDF11-AA has no boot loader in microcode.

2. Console ODT functions are different in the KDF11-AA.

3. KDF11-AA does not perform memory refresh.

4. The EVENT line is on level 6 in KDF11-AA; KD11-F and KD11-HA have it on level 4.

The following paragraphs contain more details on these differences. Also refer to Appendixes C, F, and G for additional comparison information.

In systems that used the KD11-F, the ODT command "L" could be used to automatically enter the bootstrap loader. Console ODT in the KDF11-AA does not contain a bootstrap loader command. Users who are down-line loading to KDF11-AAs must change their host software to enter the 14 memory-word bootstrap loader via console ODT. The REV11 refresh/boot module cannot be used to boot a KDF11-AA system. However, the refresh portion of the REV11 can be used to perform refresh for older MSV11-B type memories. This will cause a degradation of system performance and is not recommended. If this method of refreshing memories is employed, the bootstrap/diagnostic functionality of the REV11 must be disabled by removing/installing the appropriate jumpers. The BDV11 bootstrap/diagnostic module may be employed for automatic bootstrap function. The "L" command in the KD11-F also automatically sizes memory. KDF11-AA users whose memory size varies will have to create a program to self-size the system or use console ODT.

For improved performance the KDF11-AA was designed without memory refresh (as was the KD11-HA). The newer memories such as MSV11-C and MSV11-D perform refresh locally.

In the KDF11-AA, as in all multi-level interrupt PDP-11 systems, the event line is on level 6. In the KD11-F it is on level 4. Users whose own software locked out the event line by just setting PS<07:05> to 4 (priority level 4) will have to modify their software to set PS<07:05> to 6 (priority level 6) when installing a KDF11-AA into their present system. DIGITAL software is unaffected.

2.7 MODULE INSTALLATION PROCEDURE

Proceed as directed below.

1. Ensure that there is no dc power applied to the backplane.

2. Remove all modules from the backplane.

3. It is recommended that a single switch be used to apply +5 V and +12 V to the backplane. Simultaneous application of +5 V and +12 V is recommended.

4. Turn power on.

5. At the backplane, check for the following voltages with respect to GND (pin C2 in any backplane slot):

   Row 1, Slot A, Pin A2: +5 V
   Row 1, Slot A, Pin D2: +12 V
   Row 1, Slot A, Pin V1: +5 V

CAUTION
Do not plug in modules with power applied to backplane.

6. Turn power off.

7. Ensure that the system is properly configured as described in Paragraphs 2.2 - 2.6.

8. Insert module into backplane.

9. Turn on system power. Observe that the console device responds as described in Table 2-3.

10. If the BDV-11 is used as a system bootstrap/diagnostic device, the user must consider the following.

    a. The diagnostic portion of the BDV-11 will exercise most legal PDP-11 basic instructions at least once.

    b. The diagnostics were originally created for the KD11-F. In the KDF11-AA the BDV-11 diagnostics will not:

        (1) Perform any memory management or floating point-related tests

        (2) Exercise any memory present above 32K words.

11. Significant differences exist between console ODT responses generated by the KD11-F and the KDF11-AA (see Appendix F). Users familiar with the KD11-F (LSI-11) or users not familiar with the operation of console ODT should refer to Chapter 3.

12. As a quick check of proper system operation, the following short exerciser program can be used. It prints a continuous stream of ASCII characters on the terminal. Use console ODT to enter the following program.

```
Location        Data            Macro Code

1000            005000          CLR R0
1002            12701           MOV #177564, R1
1004            177564
1006            105711    LOOP: TSTB (R1)
1010            100376          BPL LOOP
1012            110061          MOVB R0, 2 (R1)
1014            2
1016            005200          INC R0
1020            000137          JMP @#1006
1022            001006
```

Enter "1000 G" to console ODT and a continuous stream of ASCII characters should be printed on the terminal.

13. For a more thorough check of the KDF11-AA, processor diagnostics are available to do the following.

    a. Exercise the basic instruction set

    b. Exercise the traps and interrupts

    c. Exercise the memory management and extended addressing functions

    d. Exercise the floating point hardware registers and the floating point instruction set.

    The diagnostics are as follows.

    a. Basic Instruction Set, EIS, Traps and Interrupts Test - CJKDBA

    b. MMU Diagnostic - CJKDAA

    c. Floating Point Tests

       Test 1 - CJKDCA
       Test 2 - CJKDDA

Table 2-3 Console Power-Up Printout (or Display) (Note 3)

Conditions	Mode 0	Mode 1	Mode 2	Mode 3
BHALT L (unasserted)	Processor will execute program using contents of location 24 as the PC value.	Terminal will print out a random 6-digit number, which is the contents of the program counter.	Processor will execute program at location 173000. (See Note 2.)	No printout at terminal. (See Note 1.)
BHALT L (asserted)	Terminal will print out contents of memory location 024.	Terminal will print out a random 6-digit number, which is the contents of the program counter.	Terminal will print out "173000." (See Note 2.)	No printout at terminal. (See Note 1.)

NOTES

1. If mode 3 is selected, and user microcode is not implemented, the processor will trap to memory location 010 and start program execution using the contents of location 10 as the PC value and location 12 as the PS value.

2. Normal mode for use with the BDV-11 option. If jumpers W15 through W9 are used, that address will be printed.

3. The terminal printout will consist of 6 octal digits as specified in the table, followed by a carriage return, line feed, and "@" prompt character in all cases.

3.1 INTRODUCTION

Console octal debugging technique (ODT) exists as a portion of the processor microcode that allows the processor to respond to commands and information entered via the terminal. The terminal addresses are 777560_8 through 777566_8. They are generated in microcode and cannot be changed. Console ODT is very useful as an aid in running and debugging programs. Communication between the user and processor is via a stream of ASCII characters interpreted by the processor as console commands. These commands are a subset of ODT-11. The differences in use of console ODT in the KDF11-AA as compared to the LSI-11 are listed in Appendix F.

3.2 TERMINAL INTERFACE

The minimum hardware requirements for a serial line interface permitting a terminal to communicate with console ODT are contained in the following paragraphs. The intent is to describe the minimum hardware for users who design their own serial line interface. The necessary console ODT hardware is a subset of that needed to operate system software. For system software/hardware requirements refer to the DLV11 section in the Memories and Peripherals handbook of the Microcomputer Handbook Series.

3.2.1 Receiver Control and Status Register (RCSR)

The RCSR (Figure 3-1) must exist at address 777560_8 for character input to console ODT. Console ODT does not execute DATO bus cycles to this address; therefore, the RCSR only needs to respond to DATI bus cycles. However, system software causes DATO cycles in order to affect certain bits, such as Interrupt Enable (bit 6), which console ODT does not use.

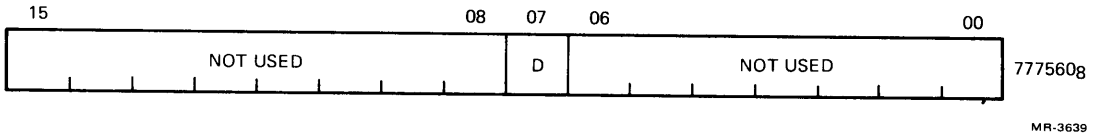


Figure 3-1 Receiver Status Register

Bit	Description
<7>	Done flag. After a character is assembled and exists in the receiver buffer register (RBUF), the Done flag must be set to a 1. When a DATI is performed to the RBUF (i.e. - to pick up the character), the Done flag must be cleared by hardware. Also bus signal BINITL must clear this bit.

Bit	Description
<6:0> <15:8>	Unused. These bits are don't cares and can be in any state since console ODT mode does not use them. In DIGITAL interfaces, these bits may be defined.

3.2.2 Receiver Buffer Register (RBUF)

The RBUF (Figure 3-2) must exist at address 777562_8 for character input to console ODT. This register only needs to respond to DATI bus cycles since console ODT does not execute DATO bus cycles to this address. System software interfaces similarly but DIGITAL diagnostics may cause a DATO cycle and not operate properly.

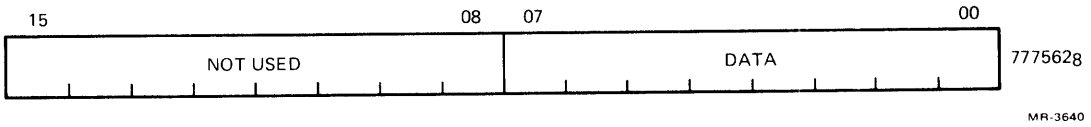


Figure 3-2 Receiver Buffer Register

Bit	Description
<7:0>	ASCII character. These eight bits are read by the processor and interpreted as a console ODT command. When bit 7 of RCSR is a 1, the processor does a DATI to the RBUF. After the DATI, the hardware must clear bit 7 of RCSR to 0.
<15:8>	Unused. These bits are don't cares and can be in any state since console ODT does not use them. In DIGITAL interfaces, these bits may be defined.

3.2.3 Transmitter Control and Status Register (XCSR)

The XCSR (Figure 3-3) must exist at address 777564_8 for character output from console ODT. ODT does not execute DATO bus cycles to this address; therefore, the XCSR only needs to respond to DATI bus cycles. However, system software causes DATO cycles to affect certain bits (e.g., Interrupt Enable).

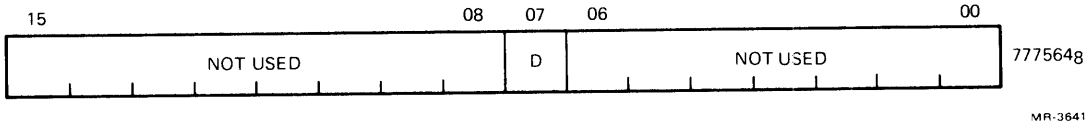


Figure 3-3 Transmitter Control and Status Register

Bit	Description
<7>	Done flag. In the idle state, this bit is a 1 indicating that the hardware is ready to print a character. After a DATO to the transmitter buffer register by the processor (i.e., a character loaded), this bit must be cleared to 0 by the hardware. After the character is printed, the hardware sets this bit to 1. During power-up this bit is set to 1. Bus signal BINIT L must set this bit to 1.
<6:0> <15:8>	Unused. These bits are don't cares and can be in any state since console ODT mode does not use them. In DIGITAL interfaces, these bits may be defined.

3.2.4 Transmitter Buffer Register (XBUF)

The XBUF (Figure 3-4) must exist at address 777566_8 for character output from console ODT. This register only needs to respond to DATO bus cycles since console ODT does not execute DATI bus cycles to this address. System software interfaces similarly but DIGITAL diagnostics may cause a DATI cycle and not operate properly.

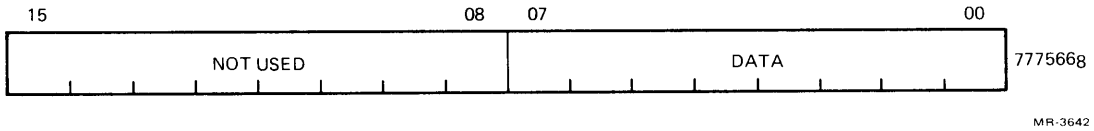


Figure 3-4 Transmitter Buffer Register

Bit	Description
<7:0>	ASCII character. These eight bits are written by the processor with the ASCII character to be printed. When bit 7 of XCSR is a 1, the processor does a DATO to the XBUF. After the DATO, the hardware must clear bit 7 of XCSR to 0.
<15:8>	Unused. These bits are don't cares and can be in any state since console ODT does not use them. In DIGITAL interfaces, these bits may be defined.

3.3 CONSOLE ODT OPERATION

The processor's microcode operates the serial line interface in half-duplex mode. Program I/O techniques are used rather than interrupts. When the console ODT microcode is busy printing characters using the transmit side of the interface, the microcode is not monitoring the receive side for incoming characters. Any characters coming in at this time are lost. The interface may post overrun errors, but the microcode does not check for any error bit in the interface. Therefore users should not type ahead

to ODT because those characters are not recognized. In addition, if another processor is at the other end of the interface, it must obey half-duplex operation. No input characters should be sent until console ODT has finished outputting.

3.3.1 Console ODT Entry Conditions

ODT may be entered as follows.

1. Execution of a HALT instruction in kernel mode, provided the HALT TRAP jumper is not installed.

2. Assertion of the BHALT L signal on the LSI-11 bus. BHALT L is a level, not edge-triggered. The signal must be asserted long enough so that it is seen at the end of a macroinstruction by the service state in the processor.

3. If option 1 has been selected, ODT is entered upon power-up.

> **NOTE**
>
> Unlike the KD11-F and KD11-HA, the KDF11-AA does not enter console ODT upon occurrence of a double bus error (i.e., R6 points to nonexistent memory during a bus timeout trap). The KDF11-AA creates a new stack at location 2 and continues to trap to 4. Since the KDF11-AA does not perform memory refresh, a bus timeout during refresh cannot take place. This differs from the KD11-F, which enters console ODT upon such an occurrence. If a bus timeout occurs while getting an interrupt vector, the KDF11-AA ignores it and continues execution of the program, whereas the KD11-F and KD11-HA enter console ODT. Refer to Appendix F for a listing of console ODT differences.

3.3.2 Console ODT Input Sequence

Upon entry to console ODT, the RBUF register is read using a DATI and the character present in the buffer is ignored. This is done so that erroneous characters or user program characters are not interpreted by console ODT as a command, especially when a program is halted.

The input sequence for console ODT is as follows.

1. Read and ignore character in RBUF.

2. Output a <CR><LF> to terminal.

3. Output contents of PC (program counter R7) in six digits to terminal.

4. Output a <CR><LF> to terminal.

5. Output the prompt character, @, to terminal.

6. Enter a wait loop for terminal input. The Done flag, bit 7 in RCSR, is tested using a DATI. If it is 0, the test continues.

7. If RCSR bit 7 is a 1, then low byte of RBUF is read using a DATI.

3.3.3 Console ODT Output Sequence

The output sequence for ODT is as follows.

1. Test XCSR bit 7 (Done flag) using a DATI and if a 0, continue testing.

2. If XCSR bit 7 is a 1, write character to low byte of XBUF using a DATO (high byte is ignored by interface).

3.4 CONSOLE ODT COMMAND SET

The console ODT command set, listed in Table 3-1, is described in the following paragraphs. The commands are a subset of ODT-11 and use the same command character. Console ODT has ten internal states. For each state only specific characters are recognized as valid inputs; other inputs invoke a "?" response. These states are described in Table 3-2.

Table 3-1 Console ODT Commands

Command	Symbol	Use
Slash	/	Prints the contents of a specified location.
Carriage Return	<CR>	Closes an open location.
Line Feed	<LF>	Closes an open location and then opens the next contiguous location.
Internal Register Designator	$ or R	Opens a specific processor register.
Processor Status Word Designator	S	Opens the PS - must follow an $ or R command.
Go	G	Starts program execution.
Proceed	P	Resumes execution of a program.
Binary Dump	Control-Shift-S	Manufacturing use only.
	H	Reserved for DIGITAL use.

Table 3-2 Console ODT States and Valid Input Characters

State	Example of Terminal Output	Valid Input	Comment
1	@	0-7 R, S G P Control-Shift-S	
2	@R or @$	0-7 S	
3	@1000/123456	0-7 CR LF	
4	@R1/123456	0-7 CR LF	
5	@1000	0-7 / G	
6	@RI or @RS	0-7 S /	
7	@1000/123456 1000	0-7 CR LF	
8	@R1/123456 1000	0-7 CR LF	
9	@	/	Previous location was opened
10	@ Control-Shift-S	2 binary bytes	

The parity bit (bit 7) on all input characters is ignored (i.e., not stripped) by console ODT and if the input character is echoed, the state of the parity bit is copied to the output buffer (XBUF). Output characters internally generated (e.g., <CR>) by ODT have the parity bit equal to 0. All commands are echoed except for <LF>. Where applicable, upper- and lowercase of command characters are recognized.

In order to describe the use of a command, other commands are mentioned before they have been defined. For the novice user, these paragraphs should be scanned first for familiarization and

then reread for detail. The word "location," as used in this paragraph, refers to a bus address, processor register, or processor status word (PS).

NOTE

In the examples the response from the processor is underlined, while the user's entry is not.

3.4.1 /(ASCII 057) Slash

This command is used to open an LSI-11 bus address, processor register, or processor status word and is normally preceded by other characters which specify a location. In response to /, console ODT prints the contents of the location (i.e., six characters) and then a space (ASCII 40). After printing is complete, console ODT waits for either new data for that location or a valid close command. The space character is issued so that the location's contents and possible new contents entered by the user are legible on the terminal.

Example: @001000/012525<SPACE>

where:

@	= console ODT prompt character.
001000	= octal location in the LSI-11 bus address space desired by the user (leading 0s are not required).
/	= command to open and print contents of location.
012525	= contents of octal location 1000.
<SPACE>	= space character generated by console ODT.

The / command can be used without a location specifier to verify the data just entered into a previously opened location. The / is recognized only if it entered immediately after a prompt character. A / issued immediately after the processor enters ODT mode causes a ?<CR><LF> to be printed because a location has not been opened.

Example: @1000/012525<SPACE> 1234 <CR><CR><LF>
@/001234<SPACE>

where:

first line	= new data of 1234 entered into location 1000 and location closed with <CR>

second line = a / was entered without a location specifier and the previous location was opened to reveal that the new contents were correctly entered into memory.

3.4.2 <CR> (ASCII 15) Carriage Return

This command is used to close an open location. If a location's contents are to be changed, the user should precede the <CR> with the new data. If no change is desired, <CR> closes the location without altering its contents.

Example: @R1/004321<SPACE> <CR> <CR><LF>
@

Processor register R1 was opened and no change was desired so the user issued<CR>. In response to the <CR>, console ODT printed <CR><LF>@.

Example: @R1/004321<SPACE> 1234 <CR> <CR><LF>
@

In this case the user desired to change R1, so new data, 1234, was entered before issuing the <CR>. Console ODT deposited the new data in the open location and then printed <CR><LF>@.

Console ODT echoes the <CR> entered by the user and then prints an additional <CR>, followed by a <LF>, and @.

3.4.3 <LF> (ASCII 12) Line Feed

This command is used to close an open location and then open the next contiguous location. LSI-11 bus addresses and processor registers are incremented by 2 and 1 respectively. If the PS is open when a <LF> is issued, it is closed and a <CR><LF>@ is printed; no new location is opened. If the open location's contents are to be changed, the new data should precede the <LF>. If no data is entered, the location is closed without being altered.

Example: @R2/123456<SPACE> <LF> <CR><LF>
@R3/054321<SPACE>

In this case, the user entered <LF> with no data preceding it. In response, console ODT closed R2 and then opened R3. When a user has the last register, R7, open, and issues <LF>, console ODT opens the beginning register, R0. When the user has the last LSI-11 bus address open of a 32K word segment and issues <LF>, console ODT opens the first location of that same segment. If the user wishes to cross the 32K word boundary, he must reenter the address for the desired 32K word segment (i.e., console ODT is modulo 32K word). This operation is the same as that found on all other PDP-11 consoles.

Example: @R7/000000<SPACE> <LF> <CR><LF>
@R0/123456<SPACE>

or

@577776/000001<SPACE> <LF> <CR><LF>
@477776/125252<SPACE>

Unlike other commands, console ODT does not echo the <LF>. Instead it prints <CR>, then <LF> so that terminal printers operate properly. In order to make this easier to decode, console ODT does not echo ASCII 0, 2 or 10, but responds to these three characters with ?<CR><LF>@.

3.4.4 $ (ASCII 044) or R (ASCII 122) Internal Register Designator

Either character when followed by a register number, 0 to 7, or PS designator, S, will open that specific processor register.

The $ character is recognized to be compatible with ODT-11. The R character was introduced for the convenience of one key stroke and because it is representative of what it does.

Example: @$0/000123<SPACE>

or

@R7/000123<SPACE> <LF>
@R0/054321<SPACE>

If more than one character is typed (digit or S) after the R or $, console ODT uses the last character as the register designator. There is an exception, however: if the last three digits equal 077 or 477, ODT opens the PS rather than R7.

3.4.5 S (ASCII 123) Processor Status Word

This designator is for opening the PS (processor status word) and must be employed after the user has entered an R or $ register designator.

Example: @RS/100377<SPACE> 0 <CR> <CR><LF>
@/000010<SPACE>

Note the trace bit (bit 4) of the PS cannot be modified by the user. This is done so that PDP-11 program debug utilities (e.g., ODT-11), which use the T bit for single-stepping, are not accidentally harmed by the user.

If the user issues a <LF> while the PS is open, the PS is closed and ODT prints a <CR><LF>@. No new location is opened in this case.

3.4.6 G (ASCII 107) Go

This command is used to start program execution at a location entered immediately before the G. This function is equivalent to the LOAD ADDRESS and START switch sequence on other PDP-11 consoles.

Example: @ 200 <NULL><NULL>

The console ODT sequence for a G, after echoing the command character, is as follows.

1. Print two nulls (ASCII 0) so the LSI-11 bus initialize that follows does not flush the G character from the double-buffered UART chip in the DLV11 serial line interface.

2. Load R7 (PC) with the entered data. If no data is entered, 0 is used. (In the above example, R7 is equal to 200 and that is where program execution begins).

3. The PS, and floating point status register if the MMU is present, is cleared to 0.

4. The LSI-11 bus is initialized by the processor asserting BINIT L for 12.6 microseconds (at 300 ns microcycle), negating BINIT L, and then waiting for 110 microseconds (at 300 ns microcycle).

5. The service state is entered by the processor. If there is anything to be serviced, it is processed. If the BHALT L bus signal is asserted, the processor reenters the console ODT state. This feature is used to initialize a system without starting a program (R7 is altered). If the user wants to single-step his program he issues a G and then successive P commands, all done with the BHALT L bus signal asserted.

3.4.7 P (ASCII 120) Proceed

This command is used to resume execution of a program and corresponds to the CONTINUE switch on other PDP-11 consoles. No programmer-visible machine state is altered using this command.

Example: @ P

Program execution resumes at the address pointed to by R7. After the P is echoed, the console ODT state is left and the processor immediately enters the state to fetch the next instruction. If the BHALT L bus signal is asserted, it is recognized at the end of the instruction (during the service state) and the processor enters the console ODT state. Upon entry, the content of the PC (R7) is printed. In this fashion, a user can single-instruction step through a program and get a PC "trace" displayed on his terminal.

3.4.8 Control-Shift-S (ASCII 23) Binary Dump

This command is used for manufacturing test purposes and is not a normal user command. It is described here to explain the machine's response if accidentally invoked. It is intended to more efficiently display a portion of memory compared to using the "/" and <LF> commands. The protocol is as follows.

1. After a prompt character, console ODT receives a control-shift-S command and echoes it.

2. The host system at the other end of the serial line must send two 8-bit bytes which console ODT interprets as a starting address. These two bytes are not echoed.

   The first byte specifies starting address <15:08> and the second byte specifies starting address <07:00>. Bus address bits <17:16> are always forced to be 0; the dump command is restricted to the first 32K words of address space.

3. After the second address byte has been received, console ODT outputs 12 octal bytes to the serial line starting at the address previously specified. When the output is finished, console ODT prints <CR><LF>@.

   If a user accidentally enters this command, it is recommended, in order to exit from the command, that two @ characters (ASCII 100) be entered as a starting address. After the binary dump, an @ prompt character is printed.

3.4.9 Reserved Commands

An ASCII H is reserved for future DIGITAL use. If it is accidentally typed, console ODT will echo the H and print a prompt character rather than a "?" which is the invalid character response. No other operation is performed.

3.5 ADDRESS SPECIFICATION

All I/O addresses (124K to 128K) must be entered by users with all 18 bits specified, regardless of whether the MMU is present or not. For example, if a user desires to open the RCSR of the DLV11, he must enter 777560, not 177560. With an MMU present, 18-bit addresses must be used to access memory greater than 32K words.

3.5.1 Processor I/O Addresses

Certain processor and MMU registers have I/O addresses assigned to them for programming purposes. If referenced in console ODT, the PS responds to its bus address, 777776. Processor registers R0 through R7 do not respond (i.e., timeout occurs) to bus addresses 777700 through 777707 if referenced in console ODT.

The MMU contains status registers and PAR/PDR pairs. Any of these registers can be accessed from console ODT by entering its bus address.

Example: @777572/000001<SPACE>

In this case, memory management status register 0 is opened and the memory management enable is set.

Accessing kernel and user stack pointer registers is accomplished in the following way. Whenever R6 is referenced in ODT, it accesses the stack pointer specified by the PS current mode bits (PS<15:14>). This is done for convenience. If a program operating in kernel mode (PS<15:14> = 00) is halted and R6 is opened, the kernel stack pointer is accessed.

3.5.2 Stack Pointer Selection

Similarly, if a program is operating in user mode, "R6" accesses the user stack pointer. If a specific stack pointer is desired, PS<15:14> must be set by the user to the appropriate value and then the "R6" command can be used. If an operating program has been halted, the original value of PS<15:14> must be restored in order to continue execution.

```
Example: PS = 140000
         @R6/123456<SPACE>
```

The user mode stack pointer has been opened.

```
         @RS/140000<SPACE> 0 <CR> <CR><LF>
         @R6/123456<SPACE> <CR> <CR><LF>

         @RS/000000<SPACE> 140000<CR> <CR><LF>
         @P
```

In this case, the kernel mode stack pointer was desired. The PS was opened and PS<15:14> were set to 00 (kernel mode). Then R6 was examined and closed. The original value of PS<15:14> was restored and then the program was continued using the P command.

If PS<15:14> are set to 01, another unique register exists in the processor, but is reserved for future DIGITAL use.

The floating point accumulators, which are also in the MMU chip, cannot be accessed from console ODT. Only floating point instructions can access these registers.

3.6 ENTERING OF OCTAL DIGITS

When the user is specifying an address or data, console ODT will use the last six octal digits if more than six have been entered. The user need not enter leading 0s for either address or data; console ODT forces 0s as the default. If an odd address is entered, the low-order bit is ignored and full 16-bit words are displayed.

3.7 ODT TIMEOUT

If the user specifies a nonexistent address or causes a parity error, console ODT responds to the error by printing ?<CR><LF>@.

3.8 INVALID CHARACTERS

Console ODT will recognize upper- and lowercase characters as commands. Any character that console ODT does not recognize during a particular sequence is echoed (with the exception of ASCII 0, 2, 10, or 12 as noted earlier) and console ODT prints a ?<CR><LF>@. Console ODT has ten internal states, each of which has its own set of valid input characters. When in a particular state, only commands specific to that state are valid (see Table 3-2). This was done to lower the probability of a user unintentionally destroying a program by pressing the wrong key.

CHAPTER 4
LSI-11 BUS

4.1 INTRODUCTION

The processor, memory and I/O devices communicate via 38 bidirectional signal lines that constitute the LSI-11 bus. Addresses, data, and control information are sent along these signal lines, some of which contain time-multiplexed information. The lines are functionally divided as follows.

18	Data/address lines - BDAL<17:00>
6	Data transfer control lines - BBS7, BDIN, BDOUT, BRPLY, BSYNC, BWTBT
3	Direct memory access control lines - BDMG, BDMR, BSACK
6	Interrupt control lines - BEVNT, BIAK, BIRQ4, BIRQ5, BIRQ6, BIRQ7
5	System control lines - BDCOK, BHALT, BINIT, BPOK, BREF

Most LSI-11 bus signals are bidirectional and use terminations for a negated (high) signal level. Devices connect to these lines via high-impedance bus receivers and open collector drivers. The asserted state is produced when a bus driver asserts the line low. Although bidirectional lines are electrically bidirectional (any point along the line can be driven or received), certain lines are functionally unidirectional. These lines communicate to or from a bus master (or signal source), but not both. Interrupt Acknowledge (BIACK) and Direct Memory Access Grant (BDMG) signals are physically unidirectional in a daisy-chain fashion. These signals originate at the processor output signal pins. Each is received on device input pins (BIAKI or BDMGI) and conditionally retransmitted via device output pins (BIAKO or BDMGO). These signals are received from higher priority devices and are retransmitted to lower priority devices along the bus. (Priorities are discussed in Paragraphs 4.3 and 4.4.1.)

Master/Slave Relationship

Communication between devices on the bus is asynchronous. A master/slave relationship exists throughout each bus transaction. At any time, there is one device that has control of the bus. This controlling device is termed the "bus master." The master device controls the bus when communicating with another device on the bus, termed the "slave." The "bus master" (typically the KDF11-AA processor or a DMA device) initiates a bus transaction. The "slave device" responds by acknowledging the transaction in progress and by receiving data from, or transmitting data to, the bus master. LSI-11 bus control signals transmitted or received by the bus master or bus slave device must complete the sequence according to bus protocol.

The processor controls bus arbitration (i.e., who becomes bus master at any given time). A typical example of this relationship is the processor, as master, fetching an instruction from memory (which is always a slave). Another example is a disk, as master, transferring data to memory as slave. Any device except the processor can be master or slave depending on the circumstances. Communication on the LSI-11 bus is interlocked so that for each control signal issued by the master device, there must be a response from the slave in order to complete the transfer. It is the master/slave signal protocol that makes the LSI-11 bus asynchronous. The asynchronous operation precludes the need for synchronizing with, and waiting for, clock pulses.

Since bus cycle completion by the bus master requires response from the slave device, each bus master must include a timeout error circuit that will abort the bus cycle if the slave device does not respond to the bus transaction within 10 microseconds. The KDF11-AA has a bus timer to restart the clock when no device responds to BDIN L or BDOUT L within 10 microseconds. An immediate trap to location 4_8 occurs.

The actual time before a timeout error occurs must be longer than the reply time of the slowest peripheral or memory device on the bus.

4.2 DATA TRANSFER BUS CYCLES

Data transfer bus cycles are as follows.

Bus Cycle Mnemonic	Description	Function (with respect to the bus master)
DATI	Data word input	Read
DATO	Data word output	Write
DATOB	Data byte output	Write byte
DATIO	Data word input/output	Read-modify-write
DATIOB	Data word input/byte output	Read-modify-write byte

These bus cycles, executed by bus master devices, transfer 16-bit words or 8-bit bytes to or from slave devices. The following bus signals are used in a data transfer operation.

Mnemonic	Description	Function
BDAL<17:00> L	18 Data/address lines	BDAL<15:00> L are used for word and byte transfers. BDAL<17:16> L are used for extended addressing, memory parity error, and memory parity error enable functions.

Mnemonic	Description	Function
BSYNC L	Synchronize	Strobe signals
BDIN L	Data input strobe	
BDOUT L	Data output strobe	
BRPLY L	Reply	
BWTBT L	Write/byte control	Control signals
BBS7 L	Bank 7 select	

Data transfer bus cycles can be reduced to three basic types; DATI, DATO(B) and DATIO(B). These transactions occur between the bus master and one slave device selected during the addressing portion of the bus cycle.

4.2.1 Bus Cycle Protocol

Before initiating a bus cycle, the previous bus transaction must have been completed (BSYNC L negated) and the device must become bus master. The bus cycle can be divided into two parts, an addressing portion, and a data transfer portion. During the addressing portion, the bus master outputs the address for the desired slave device (memory location or device register). The selected slave device responds by latching the address bits and holding this condition for the duration of the bus cycle (until BSYNC L becomes negated). During the data transfer portion, the actual data transfer occurs. Paragraphs 4.2.1.2 through 4.2.1.4 describe the data transfer portion of the bus cycle.

4.2.1.1 Device Addressing - The device addressing portion of a data transfer bus cycle comprises an address setup and deskew time and an address hold/deskew time. During the address setup and deskew time the bus master does the following.

1. Asserts BDAL<17:00> L with the desired slave device address bits
2. Asserts BBS7 L if a device in the I/O page is being addressed
3. Asserts BWTBT L if the cycle is a DATO(B) bus cycle

During this time the address, BBS7 L, and BWTBT L signals are asserted at the slave bus receiver for at least 75 ns before BSYNC goes active. Devices in the I/O page ignore the five high-order address bits BDAL<17:13> and instead decode BBS7 L along with the thirteen low-order address bits. An active BWTBT L signal indicates that a DATO(B) operation follows, while an inactive BWTBT L indicates a DATI or DATIO(B) operation.

The address hold/deskew time begins after BSYNC L is asserted.

The slave device uses the active BSYNC L bus receiver output to clock BDAL address bits, BBS7 L and BWTBT L, into its internal logic. BDAL<17:00> L, BBS7 L, and BWTBT L will remain active for 25 ns (minimum) after the BSYNC L bus receiver goes active. BSYNC L remains active for the duration of the bus cycle.

Memory and peripheral devices are addressed similarly except for the way the slave device responds to BBS7 L. Addressed peripheral devices must not decode address bits on BDAL<17:13> L. Addressed peripheral devices may respond to a bus cycle only when BBS7 L is asserted (low) during the addressing portion of the cycle. When asserted, BBS7 L indicates that the device address resides in the I/O page (the upper 4K address space). Memory devices generally do not respond to addresses in the I/O page; however, some system applications may permit memory to reside in the I/O page for use as DMA buffers, read-only-memory bootstraps or diagnostics, etc.

4.2.1.2 DATI (Figures 4-1 and 4-2) - The DATI bus cycle is a read operation. During DATI data is input to the bus master. Data consists of 16-bit word transfers over the bus. During the data transfer portion of the DATI bus cycle the bus master asserts BDIN L 100 ns minimum after BSYNC L is asserted. The slave device responds to BDIN L active in the following ways.

1. Asserts BRPLY L after receiving BDIN L and 125 ns (maximum) before BDAL bus driver data bits are valid

2. Asserts BDAL<17:00> L with the addressed data and error information

When the bus master receives BRPLY L, it does the following.

1. Waits at least 200 ns deskew time and then accepts input data at BDAL<17:00> L bus receivers. BDAL<17:16> L are used for transmitting parity errors to the master. Refer to Paragraph 4.2.2 for more details.

2. Negates BDIN L 150 ns (minimum) to 2 microseconds (maximum) after BRPLY L goes active.

The slave device responds to BDIN L negation by negating BRPLY L and removing read data from BDAL bus drivers. BRPLY L must be negated 100 ns (maximum) prior to removal of read data. The bus master responds to the negated BRPLY L by negating BSYNC L.

Conditions for the next BSYNC L assertion are as follows.

1. BSYNC L must remain negated for 200 ns (minimum).

2. BSYNC L must not become asserted within 300 ns of previous BRPLY L negation.

NOTE

Continuous assertion of BSYNC L retains control of the bus by the bus master, and the previously addressed slave device remains selected. This is done for DATIO(B) bus cycles where DATO or DATOB follows a DATI without BSYNC L negation and a second device addressing operation. Also, a slow slave device can hold off data transfers to itself by keeping BRPLY L asserted, which will cause the master to keep BSYNC L asserted.

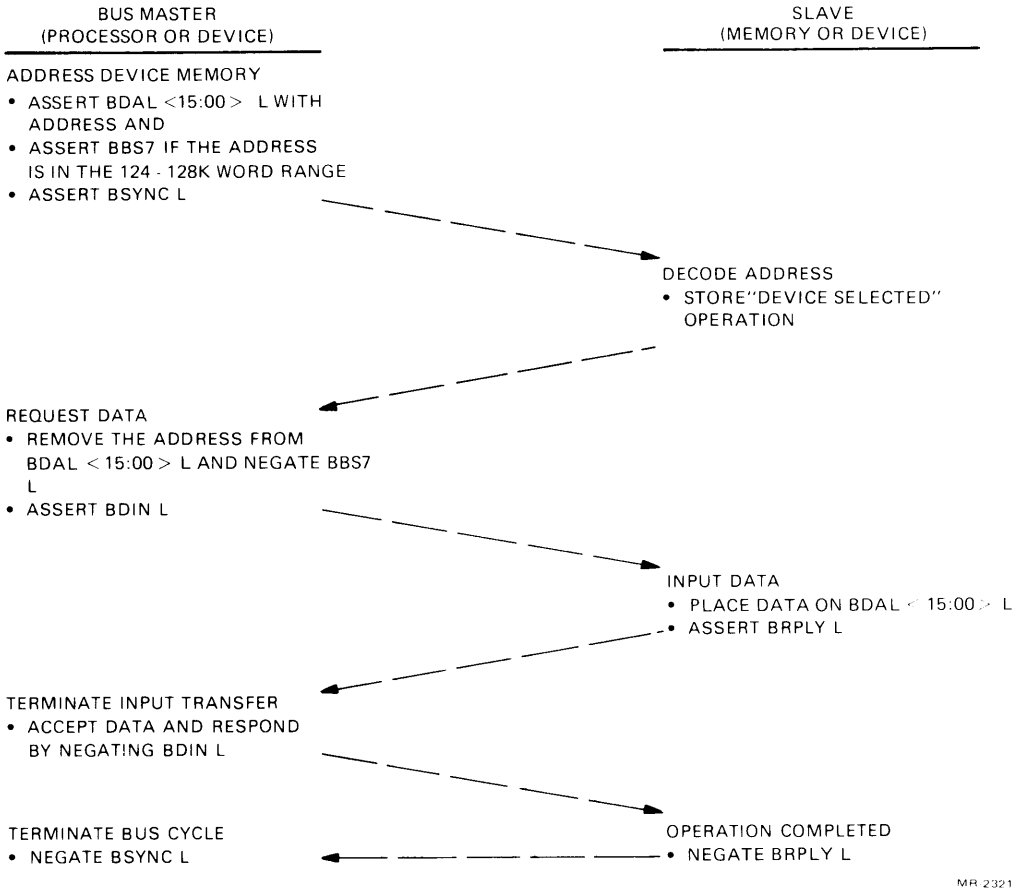
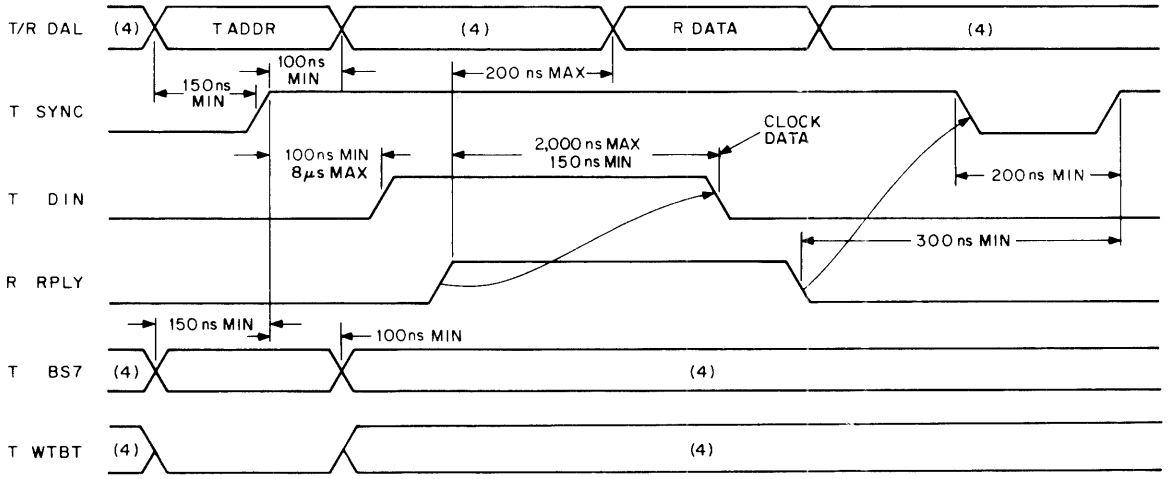
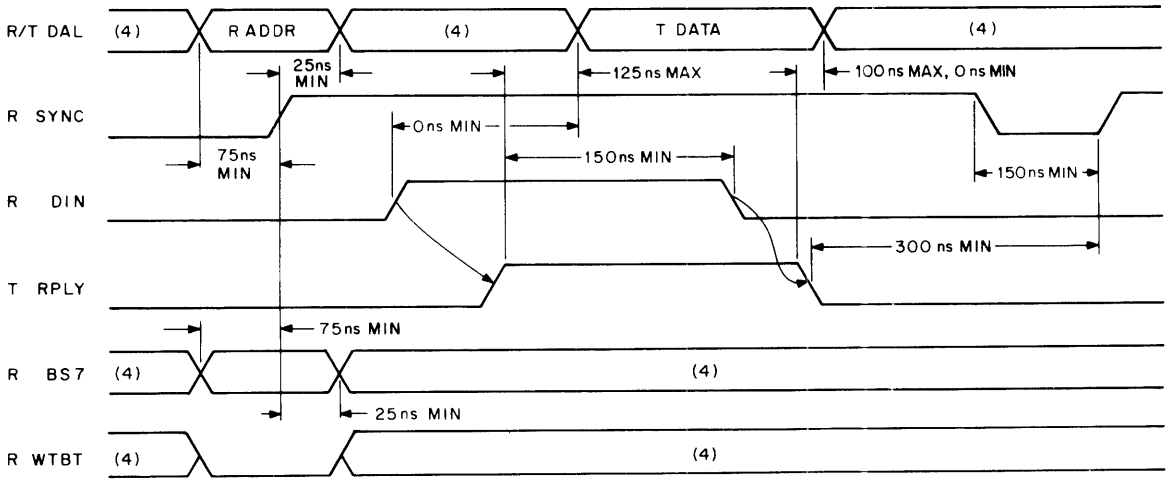


Figure 4-1 DATI Bus Cycle

4.2.1.3 DATO(B) (Figures 4-3 and 4-4) - DATO(B) is a write operation. Data is transferred in 16-bit words (DATO) or 8-bit bytes (DATOB) from the bus master to the slave device. The data transfer output can occur after the addressing portion of a bus cycle when BWTBT L had been asserted by the bus master, or immediately following an input transfer part of a DATIO(B) bus cycle.



TIMING AT MASTER DEVICE



TIMING AT SLAVE DEVICE

NOTES:

1. Timing shown at Master and Slave Device Bus Driver inputs and Bus Receiver Outputs.
2. Signal name prefixes are defined below:
   T = Bus Driver Input
   R = Bus Receiver Output
3. Bus Driver Output and Bus Receiver Input signal names include a "B" prefix.
4. Don't care condition.

MR-2322

Figure 4-2 DATI Bus Cycle Timing

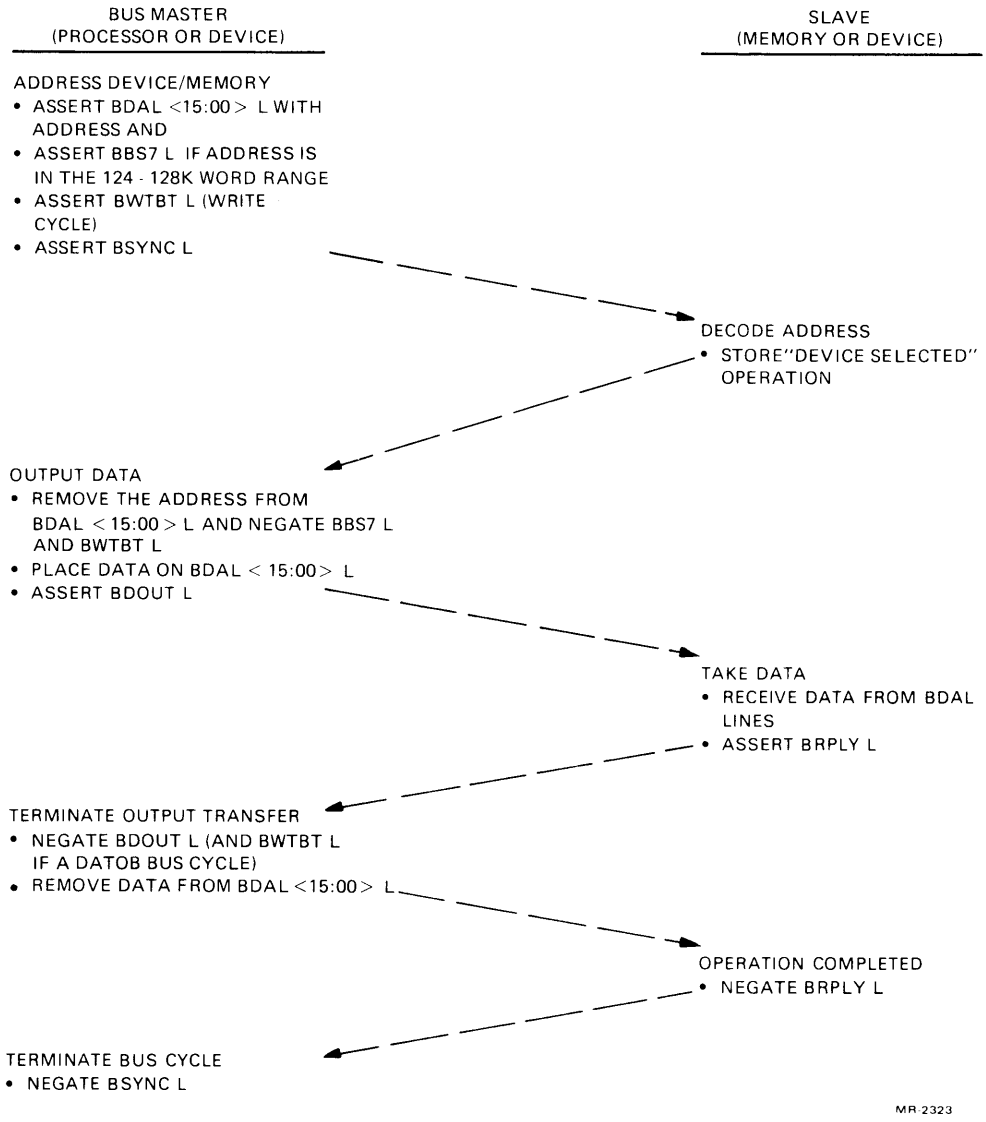
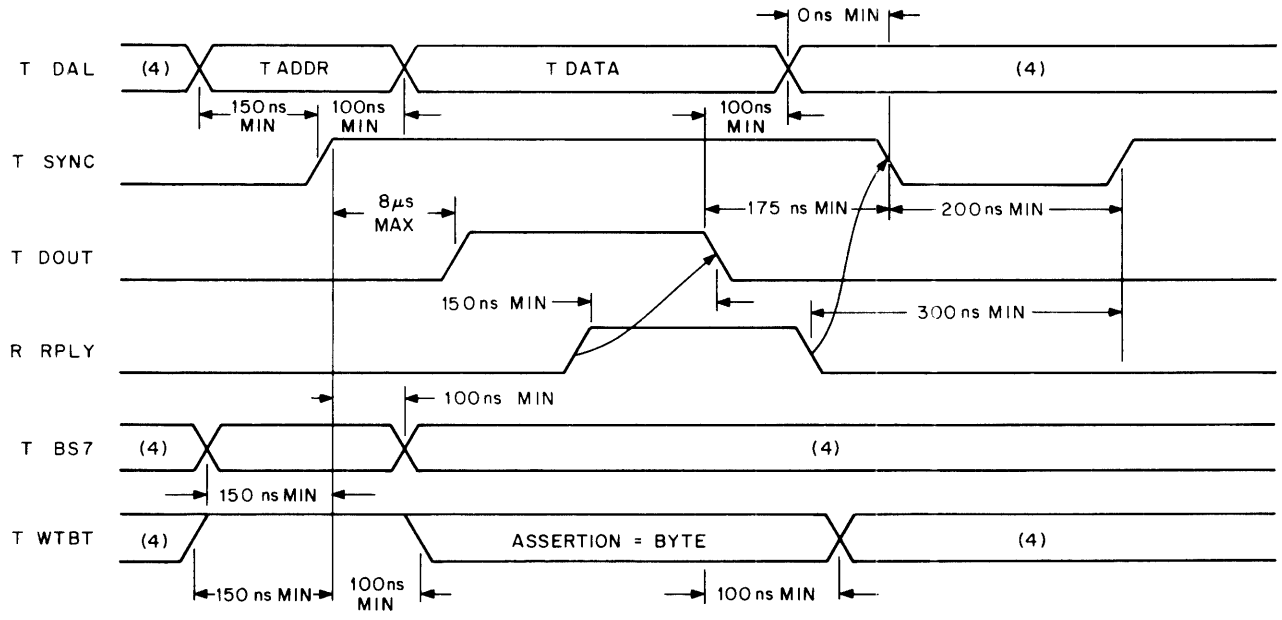


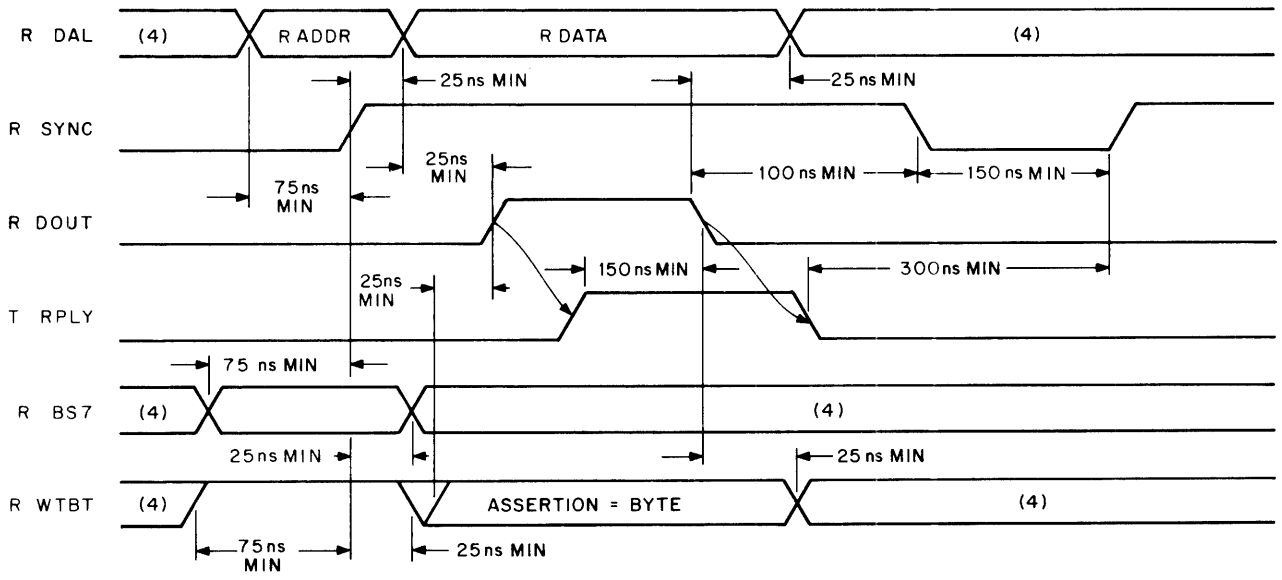
Figure 4-3 DATO or DATOB Bus Cycle

The data transfer portion of a DATO(B) bus cycle comprises a data setup and deskew time and a data hold and deskew time.

During the data setup and deskew time, the bus master outputs the data on BDAL<16:00> L at least 100 ns after BSYNC L is asserted if the transfer is a word transfer. If it is a word transfer, the bus master negates BWTBT L at least 100 ns after BSYNC L assertion. BWTBT L remains negated for the length of the bus cycle. If the transfer is a byte transfer, BWTBT L remains asserted. If it is the output of a DATIOB, BTWBT L becomes asserted and lasts the duration of the bus cycle. During a byte



TIMING AT MASTER DEVICE



TIMING AT SLAVE DEVICE

NOTES:

1. Timing shown at Master and Slave Device Bus Driver Inputs and Bus Receiver Outputs.
2. Signal name prefixes are defined below:
   T = Bus Driver Input
   R = Bus Receiver Output
3. Bus Driver Output and Bus Receiver Input signal names include a "B" prefix.
4. Don't care condition.

Figure 4-4 DATO or DATOB Bus Cycle Timing

transfer, BDAL 00 L selects the high or low byte. This occurs while in the addressing portion of the cycle. If asserted, the high byte (BDAL<15:08> L) is selected; otherwise, the low byte (BDAL<07:00> L) is selected. An asserted BDAL 16 L at this time will force a parity error to be written into memory if the memory is a parity-type memory. BDAL 17 L is not used for write operations. The bus master asserts BDOUT L at least 100 ns after BDAL and BWTBT L bus drivers are stable. The slave device responds by asserting BRPLY L within 10 microseconds to avoid bus timeout. This completes the data setup and deskew time.

During the data hold and deskew time the bus master receives BRPLY L and negates BDOUT L. BDOUT L must remain asserted for at least 150 ns from the receipt of BRPLY L before being negated by the bus master. BDAL<17:00> L bus drivers remain asserted for at least 100 ns after BDOUT L negation. The bus master then negates BDAL inputs.

During this time, the slave device senses BDOUT L negation. The data is accepted and the slave device negates BRPLY L. The bus master responds by negating BSYNC L. However, the processor will not negate BSYNC L for at least 175 ns after negating BDOUT L. This completes the DATO(B) bus cycle. Before the next cycle BSYNC L must remain unasserted for at least 200 ns.

4.2.1.4 DATIO(B) (Figures 4-5 and 4-6) - The protocol for a DATIO(B) bus cycle is identical to the addressing and data transfer portions of the DATI and DATO(B) bus cycles. After addressing the device, a DATI cycle is performed as explained in Paragraph 4.2.1.2; however, BSYNC L is not negated. BSYNC L remains active for an output word or byte transfer [DATO(B)]. The bus master maintains at least 200 ns between BRPLY L negation during the DATI cycle and BDOUT L assertion. The cycle is terminated when the bus master negates BSYNC L, which is the same as described for DATO(B).

4.2.2 Parity Protocol

The KDF11-AA recognizes memory parity errors and traps to location 114_8 if one occurs. A parity error detection occurs during every DATI or DATI portion of a DATIO(B) cycle. The processor samples BDAL 16 L and BDAL 17 L after the 200 ns REPLY deskew time similar to BDAL<15:00> L. BDAL 16 L is interpreted as a parity error signal from memory and BDAL 17 L is interpreted as a parity error enable signal from an external parity controller module. BDAL 17 L is used by software to enable parity detection which is done by addressing a parity status register on the LSI-11 bus. Parity status register hardware then asserts BDAL 17 L during the BDIN L portion of DATI cycles to inform the processor or bus master that detection is enabled. BDAL 16 L is used to indicate a parity error and is asserted by the selected memory at REPLY time. Upon system power-up, memory may contain random data and erroneous parity error signals may be issued (BDAL 16 L asserted). Until known data is written into memory, software keeps BDAL 17 L

negated, to avoid false traps. After known data and correct parity have been written into memory, software can enable parity detection in the parity status register. If both BDAL 16 L and BDAL 17 L are asserted at REPLY time, an abort and trap to location 114_8 will occur. The assertion of BDAL 16 L during BDOUT L will cause memory to write wrong parity as a diagnostic tool for maintenance purposes.

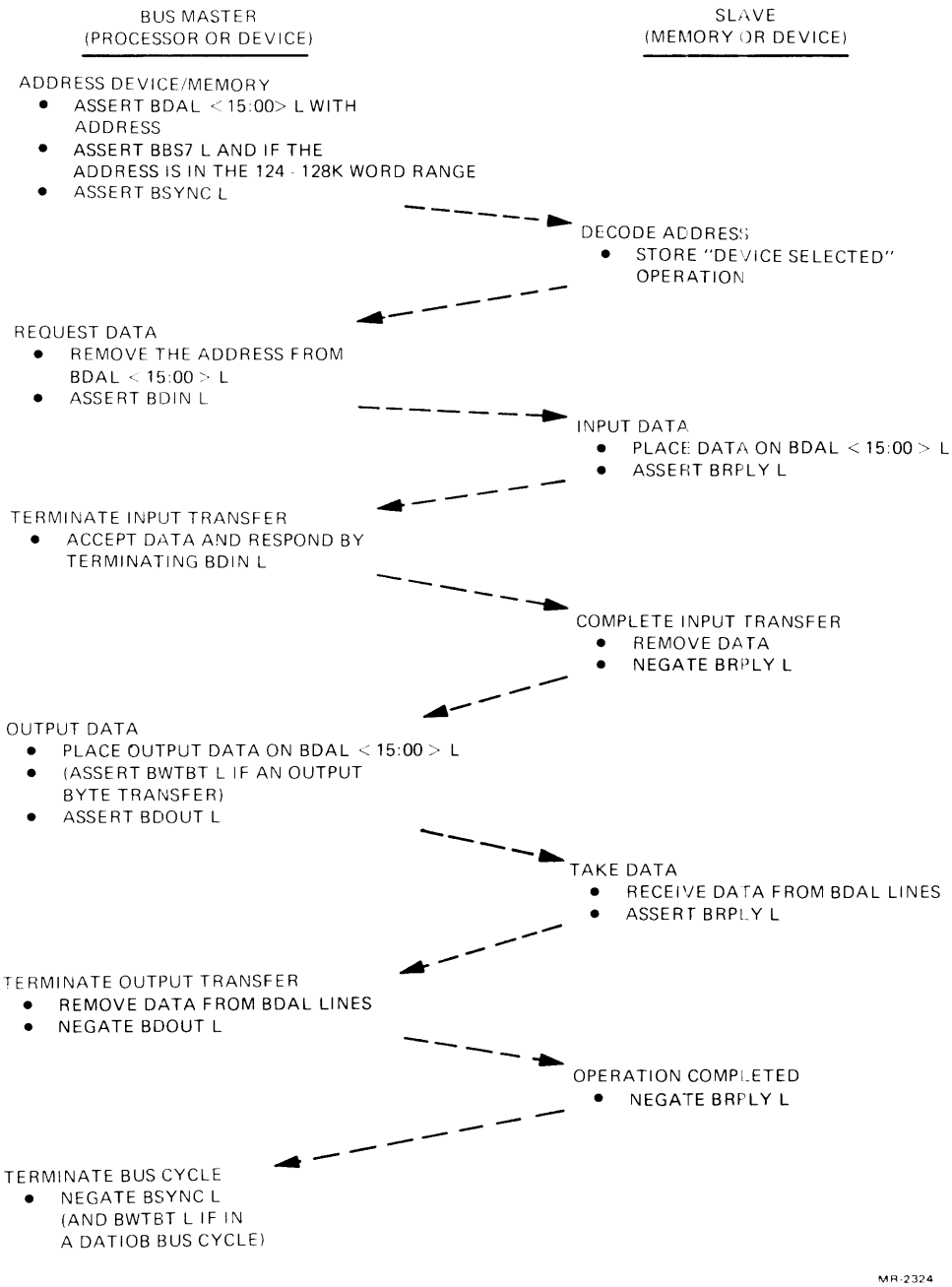
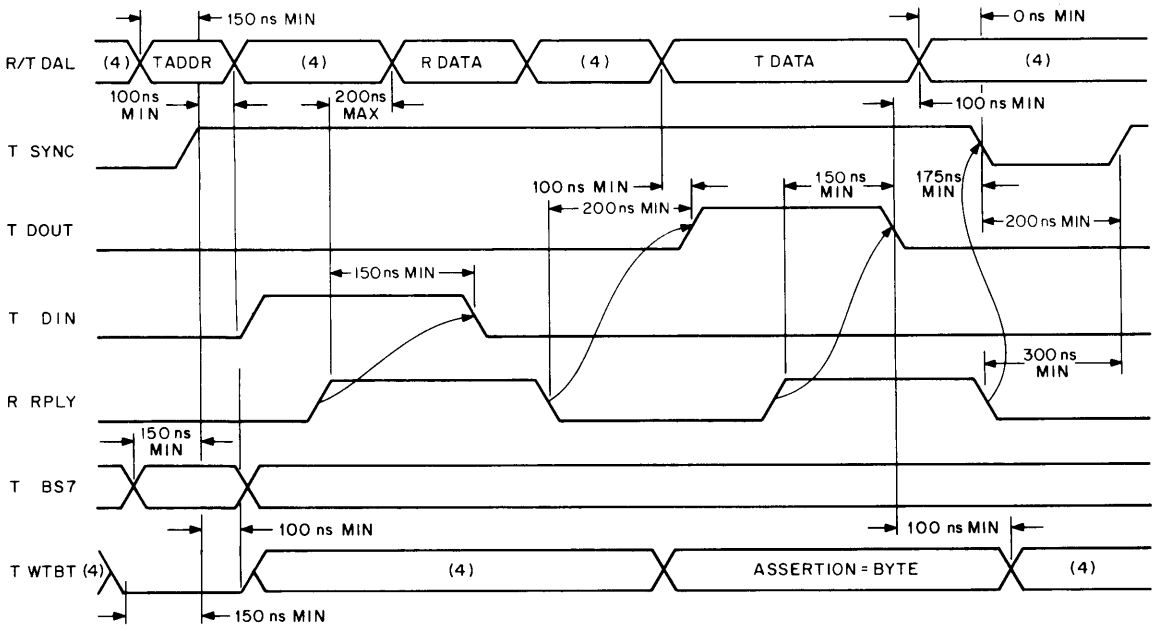
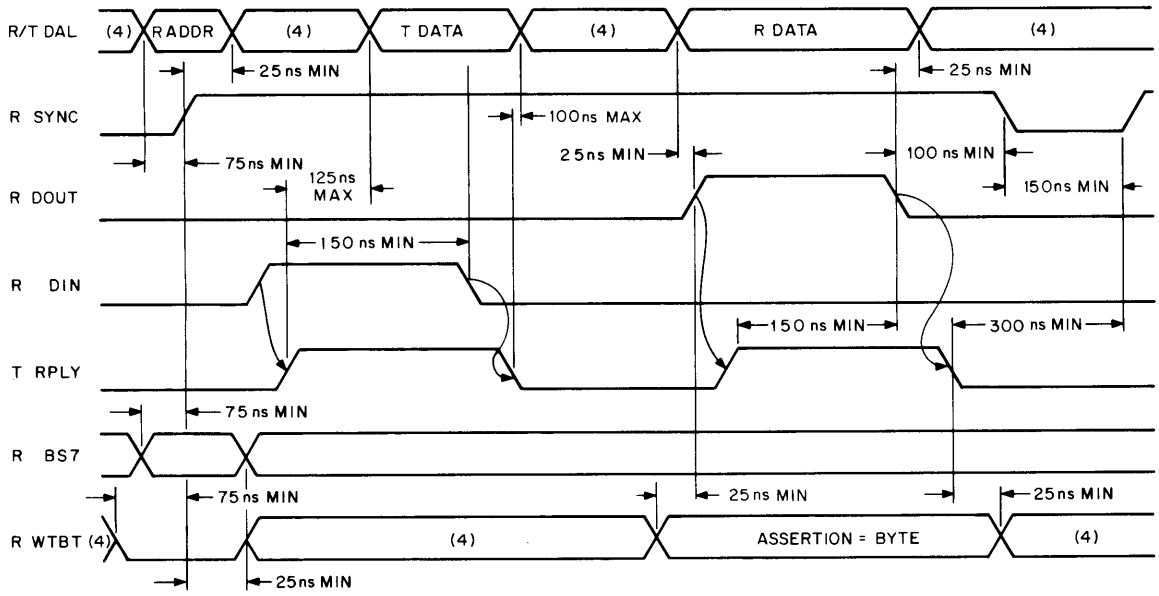


Figure 4-5 DATIO or DATIOB Bus Cycle



TIMING AT MASTER DEVICE



TIMING AT SLAVE DEVICE

NOTES:

1. Timing shown at Requesting Device
   Bus Driver Inputs and Bus Receiver Outputs.
2. Signal name prefixes are defined below:
   T = Bus Driver Input
   R = Bus Receiver Output
3. Bus Driver Output and Bus Receiver Input signal names include a "B" prefix.
4. Don't care condition

Figure 4-6 DATIO or DATIOB Bus Cycle Timing

4.3 DIRECT MEMORY ACCESS

The direct memory access (DMA) capability allows direct data transfers between I/O devices and memory. This is useful when using mass storage devices (e.g., disks) that move large blocks of data to and from memory. A DMA device only needs to know the starting address in memory, the starting address in mass storage, the length of the transfer and whether the operation is read or write. When this information is available the DMA device can transfer data directly to (or from) memory. Since most DMA devices must perform data transfers in rapid succession or lose data, DMA devices are provided the highest priority.

DMA is accomplished after the processor (normally bus master) has passed bus mastership to the highest priority DMA device that is requesting the bus. The processor arbitrates all requests and grants the bus to the DMA device located electrically closest to the processor. A DMA device remains bus master indefinitely until it relinquishes its mastership. The following control signals are used during bus arbitration.

BDMGI L	DMA Grant Input
BDMGO L	DMA Grant Output
BDMR L	DMA Request Line
BSACK L	Bus Grant Acknowledge

DMA Protocol (Figures 4-7 and 4-8)

A DMA transaction can be divided into three phases: the bus mastership acquisition phase, the data transfer phase, and the bus mastership relinquish phase.

During the bus mastership acquisition phase, a DMA device requests the bus by asserting BDMR L. The processor arbitrates the request and initiates the transfer of bus mastership by asserting BDMGO L. The maximum time between BDMR L assertion and BDMGO L assertion is DMA latency. This time is processor-dependent and is 3.5 microseconds for the KDF11-AA. BDMGO L/BDMGI L is one signal that is daisy-chained through each module in the backplane. It is driven out of the processor on the BDMGO L pin, enters each module on the BDMGI L pin and exits on the BDMGO L pin. This signal passes through the modules in descending order of priority until it is stopped by the requesting device. The requesting device blocks the output of BMDGO L and asserts BSACK L. If no device responds to the DMA grant the processor will clear the grant and rearbitrate the bus. If BDMR L is continuously asserted, the bus will be hung (the grant signal will keep passing down the bus, be cleared after no BSACK L occurs, and be driven again after the bus is rearbitrated).

NOTE

The KDF11-AA uses a no-SACK timer which clears BDMGO L after 12 microseconds if no BSACK L has been received.

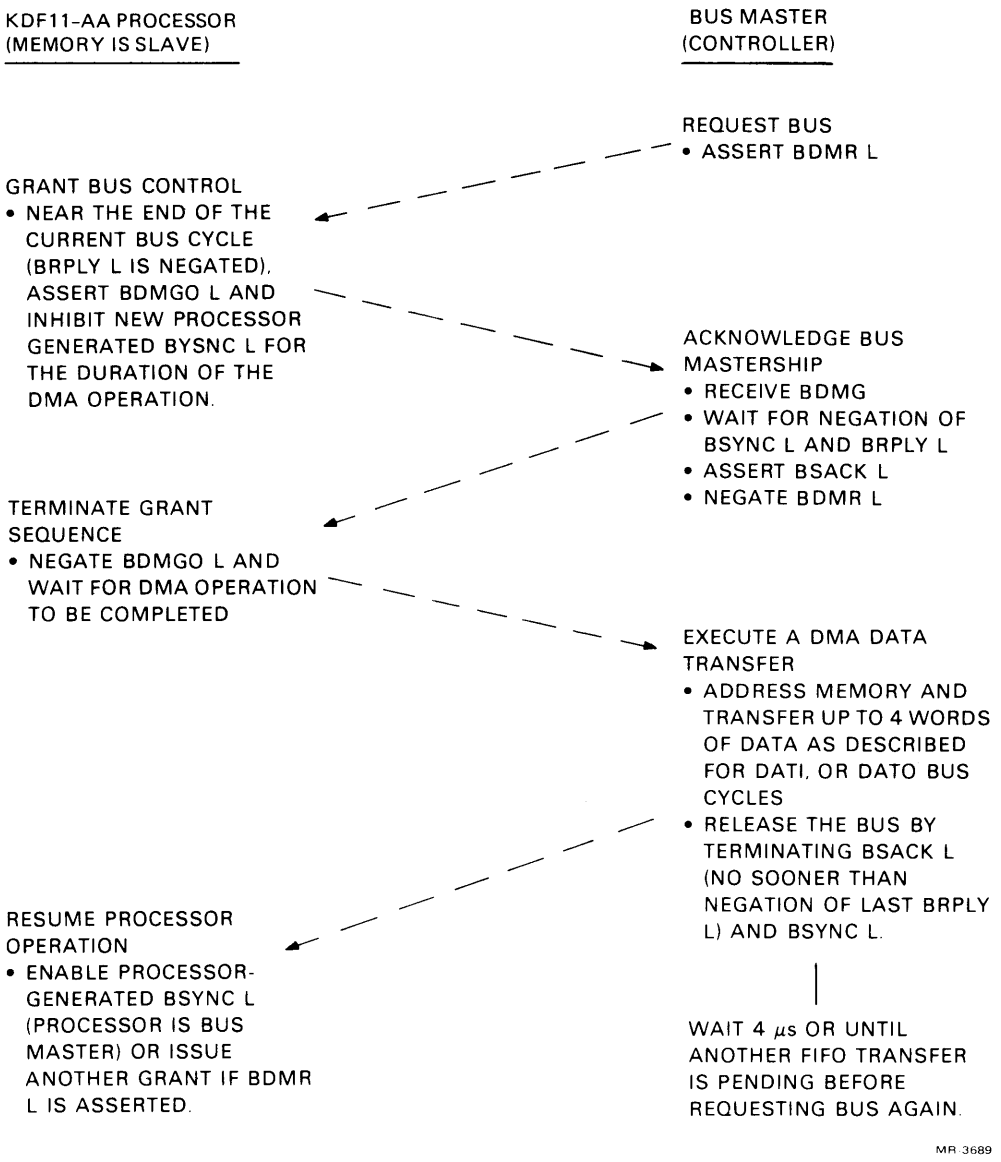


Figure 4-7 DMA Request/Grant Sequence

During the data transfer phase, the DMA device continues asserting BSACK L. The actual data transfer is performed as described in Paragraphs 4.2.1.2 through 4.2.1.4.

NOTE

If multiple-data transfers are performed during this phase, consideration must be given to the use of the bus for other system functions, such as memory refresh (if required).

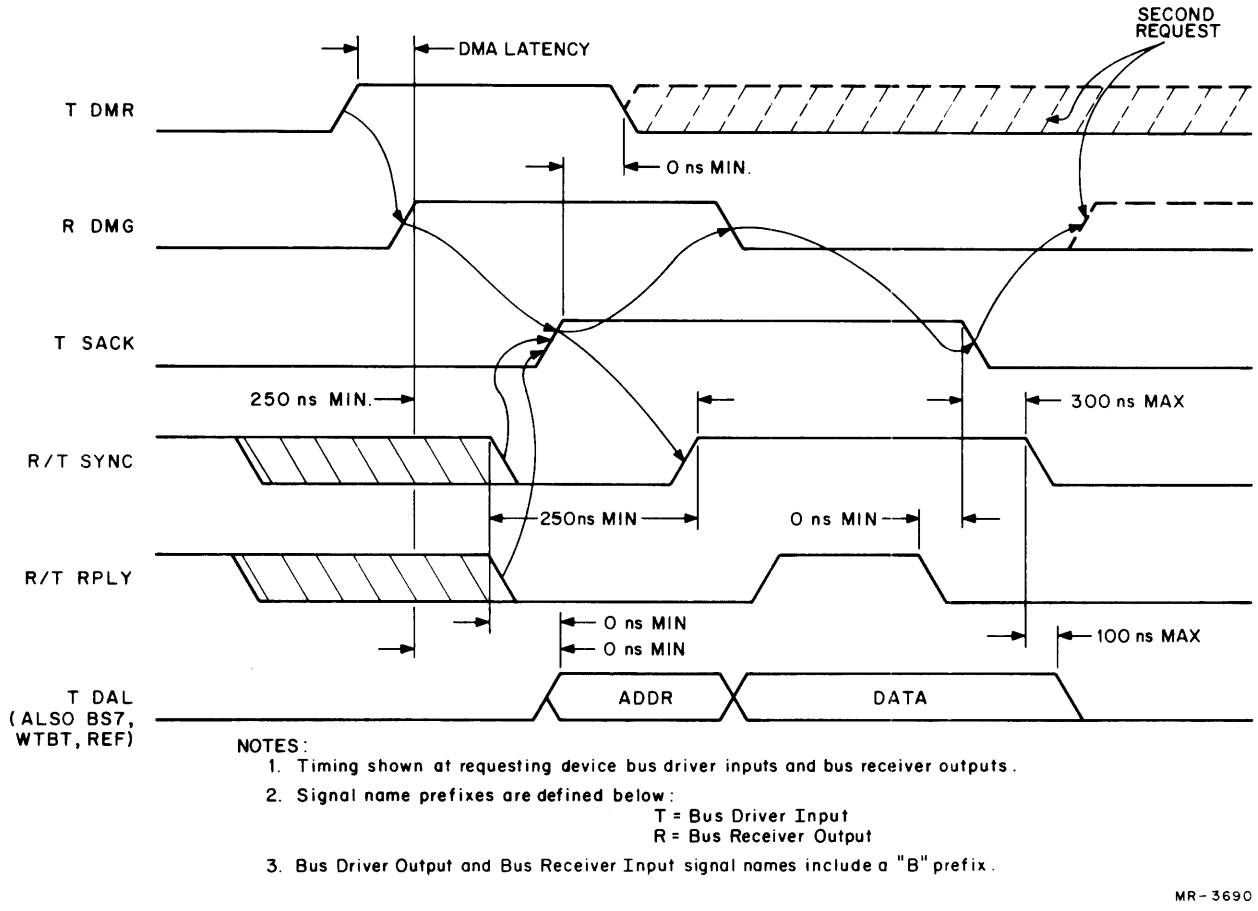


Figure 4-8 DMA Request/Grant Timing

The DMA device can assert BSYNC L for a data transfer 250 ns (minimum) after it receives BDMGI L and its BSYNC L bus receiver becomes negated.

During the bus mastership relinquish phase the DMA device relinquishes the bus by negating BSACK L. This occurs after completing (or aborting) the last data transfer cycle (BRPLY L negated). BSACK L may be negated up to 300 ns (maximum) before negating BSYNC L.

4.4 INTERRUPTS

The interrupt capability of the LSI-11 bus allows any I/O device to temporarily suspend (interrupt) current program execution and divert processor operation to service the requesting device. The processor inputs a vector from the device to start the service routine (handler). Like the device register address, hardware fixes the device vector at locations within a designated range

(below location 001000). The vector indicates the first of a pair of addresses. The content of the first address is read by the processor and is the starting address of the interrupt handler. The content of the second address is a new processor status word (PS). The new PS can raise the interrupt priority level, thereby preventing lower level interrupts from breaking into the current interrupt service routine. Control is returned to the interrupted program when the interrupt handler is ended. The original (interrupted) program's address (PC) and its associated PS are stored on a "stack." The original PC and PS are restored by a return from interrupt (RTI or RTT) instruction at the end of the handler. The use of the stack and the LSI-11 bus interrupt scheme can allow interrupts to occur within interrupts (nested interrupts), depending on the PS.

Interrupts can be caused by LSI-11 bus options. Interrupt operations can also originate from within the processor. These interrupts are called "traps." Traps are caused by programming errors, hardware errors, special instructions and maintenance features.

The LSI-11 bus signals that are used in interrupt transactions are the following.

BIRQ4 L	Interrupt request priority level 4
BIRQ5 L	Interrupt request priority level 5
BIRQ6 L	Interrupt request priority level 6
BIRQ7 L	Interrupt request priority level 7
BIAKI L	Interrupt acknowledge input
BIAKO L	Interrupt acknowledge output
BDAL<15:00> L	Data/address lines
BDIN L	Data input strobe
BRPLY L	Reply

4.4.1 Device Priority

The LSI-11 bus supports the following two methods of device priority.

1. Distributed arbitration - Priority levels are implemented on the hardware. When devices of equal priority level request an interrupt, priority is given to the device electrically closest to the processor.

2. Position-defined arbitration - Priority is determined solely by electrical position on the bus. The closer a device is to the processor, the higher its priority is.

The KDF11-AA uses both the distributed arbitration method with four levels of priority and position-defined arbitration within each level. Interrupts on these priority levels are

enabled/disabled by bits in the processor status word (PS<07:05>). Single-level interrupt (position-defined) devices can also be used in KDF11-AA systems.

4.4.2 Interrupt Protocol (Figures 4-9 and 4-10)

Interrupt protocol has three phases: interrupt request phase, interrupt acknowledge and priority arbitration phase, and interrupt vector transfer phase.

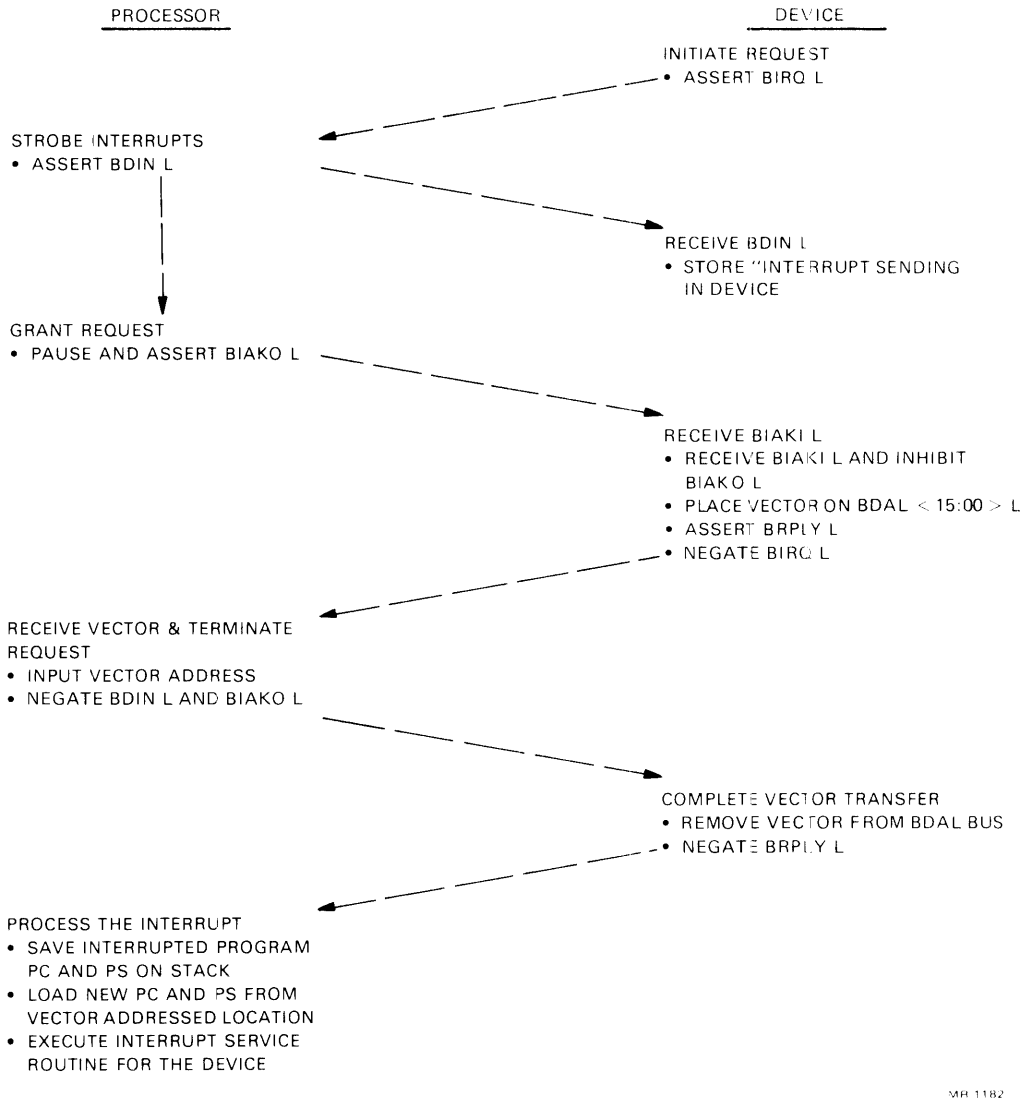


Figure 4-9 Interrupt Request/Acknowledge Sequence

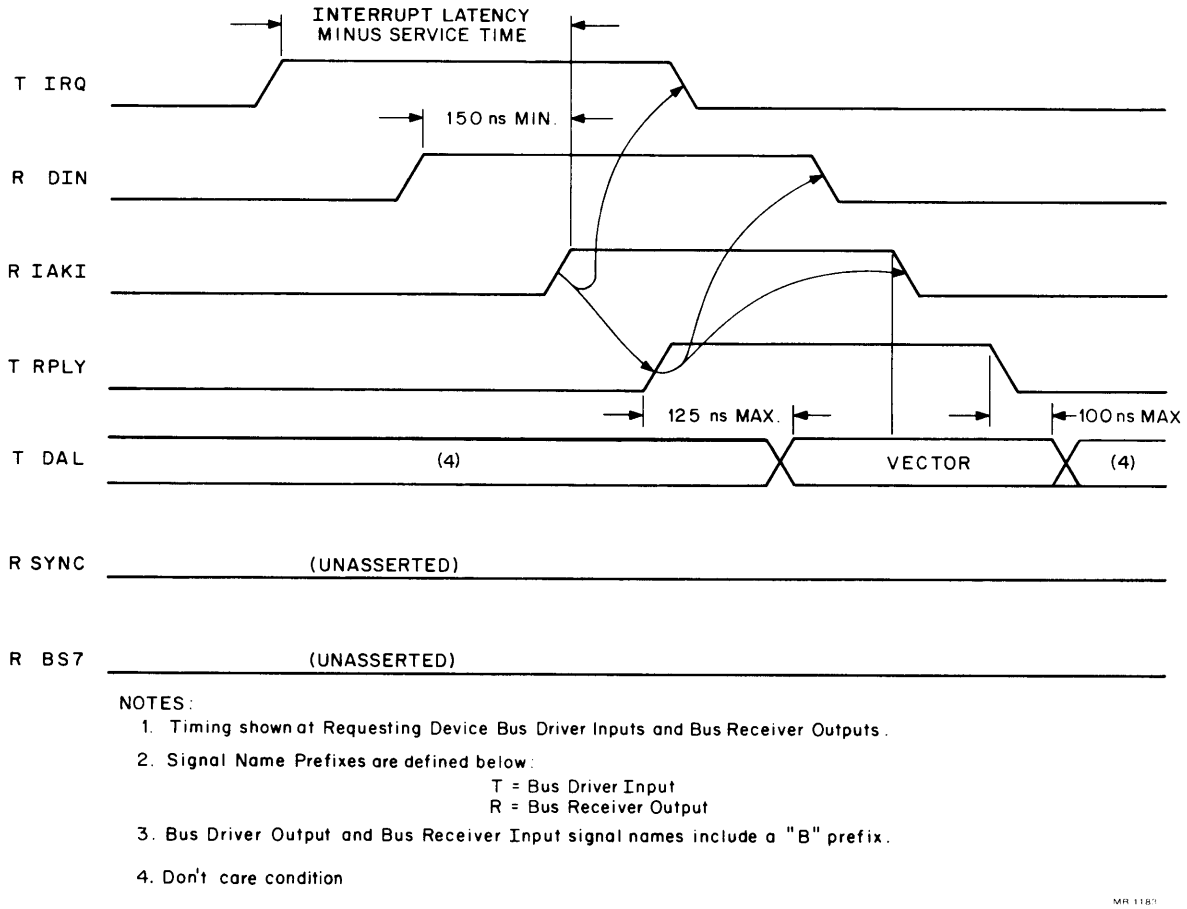


Figure 4-10 Interrupt Protocol Timing

The interrupt request phase begins when a device meets its specific conditions for interrupt requests. For example, the device is "ready," "done," or an error has occurred. The interrupt enable bit in a device status register must be set. The device then initiates the interrupt by asserting the interrupt request line(s). BIRQ4 L is the lowest hardware priority level and is asserted for all interrupt requests for compatibility with previous LSI-11 processors. The level a device is configured at must also be asserted. A special case exists for level 7 devices which must also assert level 6. See item 2 of the arbitration discussion involving the 4-level scheme (below) for an explanation.

Interrupt Level	Lines Asserted by Device
4	BIRQ4 L
5	BIRQ4 L, BIRQ5 L
6	BIRQ4 L, BIRQ6 L
7	BIRQ4 L, BIRQ6 L, BIRQ7 L

The interrupt request line remains asserted until the request is acknowledged.

During the interrupt acknowledge and priority arbitration phase the KDF11-AA will acknowledge interrupts under the following conditions.

1. The device interrupt priority is higher than the current PS<07:05>.

2. The processor has completed instruction execution and no additional bus cycles are pending.

The processor acknowledges the interrupt request by asserting BDIN L, and 150 ns (minimum) later asserting BIAKO L. The device electrically closest to the processor receives the acknowledge on its BIAKI L bus receiver.

At this point the two types of arbitration must be discussed separately. If the device that receives the acknowledge uses the 4-level interrupt scheme, it reacts as follows.

1. If not requesting an interrupt, the device asserts BIAKO L and the acknowledge propagates to the next device on the bus.

2. If the device is requesting an interrupt it must check to see that no higher level device is currently requesting an interrupt. This is done by monitoring higher level request lines. The table below lists the lines that need to be monitored by devices at each priority level.

   In addition to asserting levels 7 and 4, level 7 devices must drive level 6. This is done to simplify the monitoring and arbitration by level 4 and 5 devices. In this protocol, level 4 and 5 devices need not monitor level 7 since level 7 devices assert level 6. Level 4 and 5 devices will become aware of a level 7 request since they monitor the level 6 request. This protocol has been optimized for levels 4, 5, and 6 devices, since level 7 devices very seldom are necessary.

Device Priority Level	Line(s) Monitored
4	BIRQ5, BIRQ6
5	BIRQ6
6	BIRQ7
7	-

3. If no higher level device is requesting an interrupt, the acknowledge is blocked by the device. (BIAKO L is not asserted). Arbitration logic within the device uses the leading edge of BDIN L to clock a flip-flop that blocks BIAKO L. Arbitration is won and the interrupt vector transfer phase begins.

4. If a higher level request line is active, the device disqualifies itself and asserts BIAKO L to propagate the acknowledge to the next device along the bus.

Signal timing must be carefully considered when implementing 4-level interrupts. Refer to Figure 4-10 for interrupt protocol timing.

If a single-level interrupt device receives the acknowledge, it reacts as follows.

1. If not requesting an interrupt, the device asserts BIAKO L and the acknowledge propagates to the next device on the bus.

2. If the device was requesting an interrupt, the acknowledge is blocked using the leading edge of BDIN L and arbitration is won. The interrupt vector transfer phase begins.

The interrupt vector transfer phase is enabled by BDIN L and BIAKI L. The device responds by asserting BRPLY L and its BDAL<15:00> L bus driver inputs with the vector address bits. The BDAL bus driver inputs must be stable within 125 ns (maximum) after BRPLY L is asserted. The processor then inputs the vector address and negates BDIN L and BIAKO L. The device then negates BRPLY L and 100 ns (maximum) later removes the vector address bits. The processor then enters the device's service routine.

NOTE

BSYNC L and BBS7 L are not asserted by the processor during the interrupt arbitration and vector address transfer operations.

The device must assert BRPLY L within 10 microseconds (maximum) after the processor asserts BDIN L. If the device does not reply, the KDF11-AA aborts the interrupt transaction and resumes

program execution. The aborted transaction is transparent to the program. LSI-11 and LSI-11/2 processors halt in this situation.

4.4.3 4-Level Interrupt Configurations

Users who have high-speed peripherals and desire better software performance can use the 4-level interrupt scheme. Both position-independent and position-dependent configurations can be used with the 4-level interrupt scheme.

The position-independent configuration is shown in Figure 4-11. This allows peripheral devices that use the 4-level interrupt scheme to be placed in the backplane in any order. These devices must send out interrupt requests and monitor higher level request lines as described in Paragraph 4.4.2. The level 4 request is always asserted by a requesting device regardless of priority, to allow compatibility if an LSI-11 or LSI-11/2 processor is in the same system. If two or more devices of equally high priority request an interrupt, the device physically closest to the processor will win arbitration. Devices that use the single-level interrupt scheme must be modified or placed at the end of the bus for arbitration to properly function.

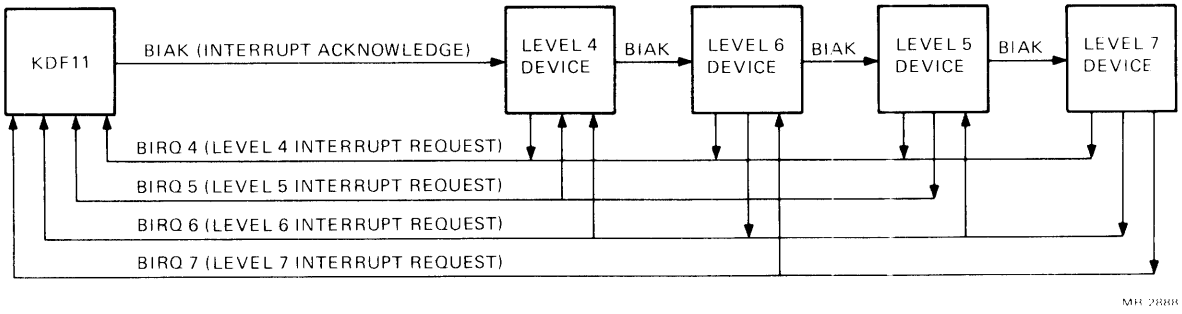


Figure 4-11 Position-Independent Configuration

The position-dependent configuration is shown in Figure 4-12. This configuration is simpler to implement. A constraint is that peripheral devices must be inserted with the highest priority device located closest to the processor and the remaining devices placed in the backplane in decreasing order of priority, with the lowest priority devices farthest from the processor. With this configuration each device only has to assert its own level and level 4 (for compatibility with an LSI-11 or LSI-11/2). Monitoring higher level request lines is unnecessary. Arbitration is achieved through the physical positioning of each device on the bus. Single-level interrupt devices on level 4 should be positioned last on the bus.

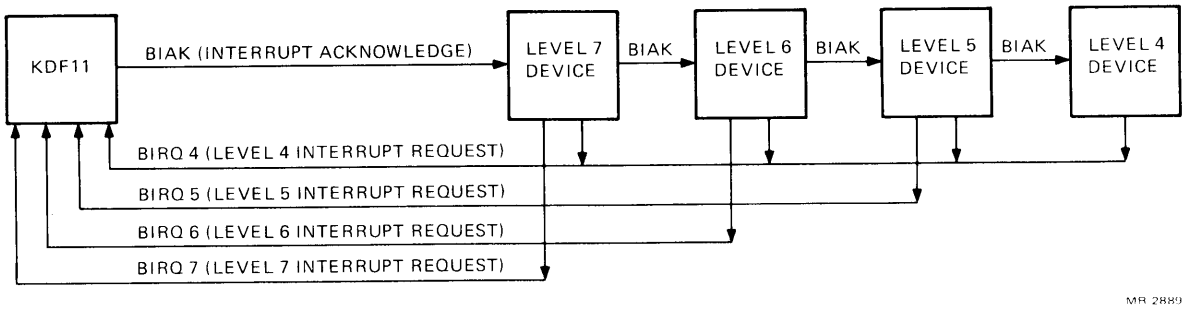


Figure 4-12 Position-Dependent Configuration

4.5 CONTROL FUNCTIONS

The following LSI-11 bus signals provide control functions.

BREF L	Memory refresh
BHALT L	Processor halt
BINIT L	Initialize
BPOK H	Power OK
BDCOK H	dc power OK

4.5.1 Memory Refresh

If BREF is asserted during the address portion of a bus data transfer cycle, it causes all dynamic MOS memories to be simultaneously addressed. The sequence of addresses required for refreshing the memories is determined by the specific requirements for each memory. The complete memory refresh cycle consists of a series of refresh bus transactions. A new address is used for each transaction. A complete memory refresh cycle must be completed within 1 or 2 ms. Multiple data transfers by DMA devices must be avoided since they could delay memory refresh cycles. The KDF11-AA does not perform memory refresh. For systems needing memory refresh, an REV11-A or -C may be used to perform this function.

4.5.2 Halt

Assertion of BHALT L stops program execution and forces the processor unconditionally into console ODT mode.

4.5.3 Initialization

Devices along the bus are initialized when BINIT L is asserted. The processor can assert BINIT L as a result of executing a RESET instruction or as part of a power-up sequence. BINIT L is asserted for approximately 10 microseconds when RESET is executed.

4.5.4 Power Status

Power status protocol is controlled by two signals, BPOK H and BDCOK H. These signals are driven by some external device (usually the power supply) and are defined as follows.

BDCOK H
The assertion of this line indicates that dc power has been stable for at least 3 ms. Once asserted this line remains asserted until the power fails.

The negation of this line is the first event in the power-fail sequence. It indicates that only 5 microseconds of dc power reserve remains. Once BDCOK H is negated it must remain in this state for at least 1 microsecond before being asserted again.

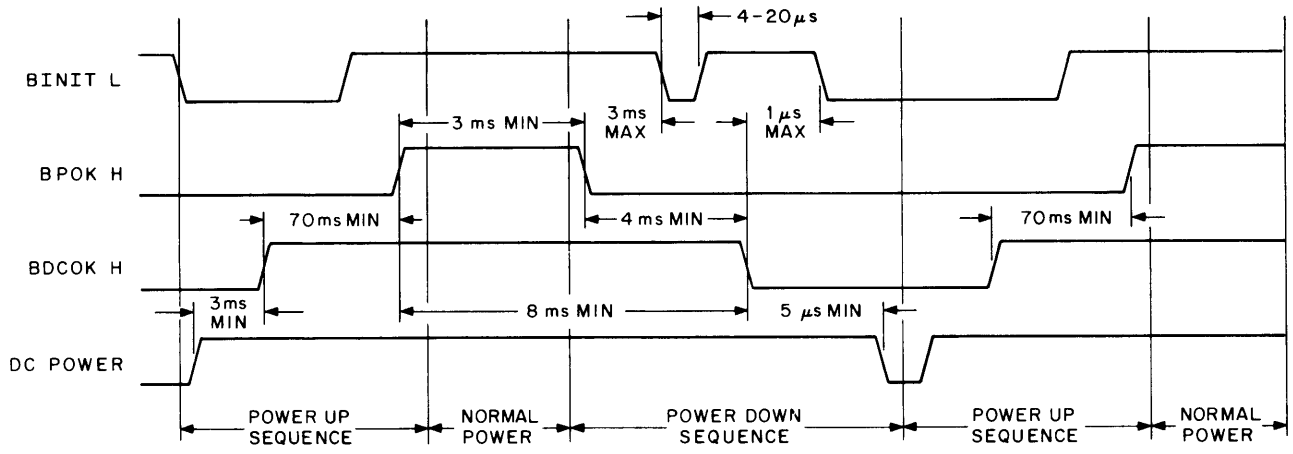
BPOK H
The assertion of this line indicates that there is at least an 8 ms reserve of dc power and that BDCOK H has been asserted for at least 70 ms. Once BPOK H has been asserted, it must remain asserted for at least 3 ms.

The negation of this line indicates that power is failing and that only 4 ms of dc power reserve remains.

Power-Up/Down Protocol (Figure 4-13)
Power-up protocol for the KDF11-AA begins when the power supply applies power with BDCOK H negated. This forces the processor to assert BINIT L. When the dc voltages are stable, the power supply or other external device asserts BDCOK H. The processor responds by clearing the PS, floating point status register (FPS), and floating point exception register (FEC). BINIT L is asserted for 12.6 microseconds and then negated for 110 microseconds. The processor continues to test for BPOK H until it is asserted. The power supply asserts BPOK H 70 ms (minimum) after BDCOK H is asserted. The processor then performs its power-up sequence. Normal power must be maintained at least 3.0 ms before a power-down sequence can begin. The KDF11-AA has four power-up jumper options. Refer to Paragraph 2.2 for details.

A power-down sequence begins when the power supply negates BPOK H. When the current instruction is completed, the processor traps to a power-down routine. The KDF11-AA traps to location 24_8. Location 24_8 contains the PC that points to the power-down routine. The end of the routine is terminated with a HALT instruction to avoid any possible memory corruption as the dc voltages decay.

When the processor executes the HALT instruction, it tests the BPOK H signal. If BPOK H is negated, the processor enters the power-up sequence. It clears internal registers, generates BINIT L and continues to check for the assertion of BPOK H. If it is asserted and dc voltages are still stable, the processor will perform the rest of the power-up sequence.



NOTE:
Once a power down sequence is started, it must be completed before a power-up sequence is started.

Figure 4-13 Power-Up/Power-Down Timing

4.6 BUS ELECTRICAL CHARACTERISTICS

This paragraph contains information about the electrical characteristics of the LSI-11 bus.

4.6.1 Signal Level Specification

Input Logic Levels
TTL Logical Low: 0.8 Vdc maximum
TTL Logical High: 2.0 Vdc minimum

Output Logic Levels
TTL Logical Low: 0.4 Vdc maximum
TTL Logical High: 2.4 Vdc minimum

4.6.2 AC Load Definition

AC loads comprise the maximum capacitance allowed per signal line to ground, as specified in Paragraph 4.7. A unit load is defined as 9.35 pF of capacitance.

4.6.3 DC Load Definition

DC loads are defined as maximum current allowed with a signal line driver asserted or unasserted. These limitations are specified in Paragraph 4.7.

4.6.4 120 Ohm LSI-11 Bus

The electrical conductors interconnecting the bus device slots are treated as transmission lines. A uniform transmission line, terminated in its characteristic impedance, will propagate an electrical signal without reflections. Insofar as bus drivers,

receivers and wiring connected to the bus have finite resistance and nonzero reactance, the transmission line impedance becomes nonuniform, and thus introduces distortions into pulses propagated along it. Passive components of the LSI-11 bus (such as wiring, cabling and etched signal conductors) are designed to have a nominal characteristic impedance of 120 ohms.

The maximum length of interconnecting cable excluding wiring within the backplane is limited to 4.88 m (16 ft).

NOTES

1. The KDF11-AA processor (as well as all standard DIGITAL-supplied LSI-11 interfaces) connects to the bus via special drivers and receivers, described in Paragraphs 4.6.5 and 4.6.6.

2. The KDF11-AA processor provides resistive (220 ohm) pull-up (on all bused lines) to 3.4 Vdc for this wired-OR interconnecting scheme.

4.6.5 Bus Drivers

Devices driving the 120 ohm LSI-11 bus must have open collector outputs and meet the following specifications.

DC Specifications

Output low voltage when sinking 70 mA of current: 0.7 V maximum

Output high leakage current when connected to 3.8 Vdc: 25 uA (even if no power is applied to them, except for BDCOK H and BPOK H)

These conditions must be met at worst-case supply voltage, temperature, and input signal levels.

AC Specifications

Bus driver output pin capacitive load: Not to exceed 10 pF

Propagation delay: Not to exceed 35 ns

Skew (difference in propagation time between slowest and fastest gate): Not to exceed 25 ns

Rise/Fall Times: Transition time from 10% to 90% for positive transition, and from 90% to 10% for negative transition, must be no faster then 5 ns.

4.6.6 Bus Receivers

Devices that receive signals from the 120 ohm LSI-11 bus must meet the following requirements.

DC Specifications

Input low voltage (maximum): 1.3 V

Input high voltage (minimum): 1.7 V

Maximum input current when connected to 3.8 Vdc: 80 uA even if no power is applied to them.

These specifications must be met at worst-case supply voltage, temperature, and output signal conditions.

AC Specifications

Bus receiver input pin capacitance load: Not to exceed 10 pF

Propagation delay: Not to exceed 35 ns

Skew (difference in propagation time between slowest and fastest gate): Not to exceed 25 ns

4.6.7 Bus Termination (Figure 4-14)

The 120 ohm LSI-11 bus must be terminated at each end by an appropriate terminator. This is to be done as a voltage divider with its Thevenin equivalent equal to 120 ohms and 3.4 V nominal. This type of termination is provided by an REV11-A refresh/boot/terminator, or the BDV11-AA.

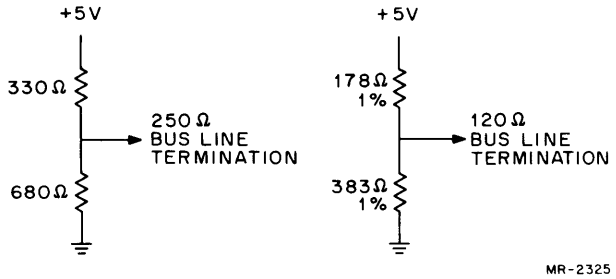


Figure 4-14 Bus Line Terminations

Each of the several LSI-11 bus lines (all signals whose mnemonics start with the letter B) must see an equivalent network with the following characteristics at each end of the bus, except BIAKI L/BIAKO L and BDMGI L/BDMGO L, which do not require termination.

Input impedance (with respect to ground): Z = 120 ohm +5%, -15%

Open circuit voltage: 3.4 Vdc +5%

Capacitance Load: Not to exceed 30 pF

NOTE
The resistive termination may be provided by the combination of two modules (i.e., the processor module supplies 220 ohms to ground). Both of these two terminators must be physically resident within the same backplane.

4.6.8 Bus Interconnecting Wiring

This paragraph contains the electrical characteristics of the bus interface.

4.6.8.1 Backplane Wiring - The wiring that interconnects all device interface slots on the LSI-11 must meet the following specifications.

1. The conductors must be arranged such that each line exhibits a characteristic impedance of 120 ohms (measured with respect to the bus common return).

2. Crosstalk between any two lines must be no greater than 5%. Note that worst-case crosstalk is manifested by simultaneously driving all but one signal line and measuring the effect on the undriven line.

3. DC resistance of signal path, as measured between near-end terminator and far-end terminator module (including all intervening connectors, cables, backplane wiring, connector-module etch, etc.) must not exceed 2 ohms.

4. DC resistance of common return path, as measured between near-end terminator and far-end terminator module (including all intervening connectors, cables, backplane wiring, connector-module etch, etc.) must not exceed an equivalent of 2 ohms per signal path. Thus, the composite signal return path dc resistance must not exceed 2 ohms divided by 40 bus lines, or 50 milliohms. Note that although this common return path is nominally at ground potential, the conductance must be part of the bus wiring; the specified low impedance return path must be provided by the bus wiring as distinguished from common system or power ground path.

4.6.8.2 Intra-Backplane Bus Wiring - The wiring that interconnects the bus connector slots within one contiguous backplane is part of the overall bus transmission line. Due to implementation constraints, the nominal characteristic impedance

of 120 ohms may not be achievable. Distributed wiring capacitance in excess of the amount required to achieve the nominal 120 ohm impedance may not exceed 60 pF per signal line per backplane.

4.6.8.3 **Power and Ground** - Each bus interface slot has connector pins assigned for the following dc voltages.*

+5 Vdc - Three pins (4.5 A maximum per bus device slot)

+12 Vdc - Two pins (3.0 A maximum per bus device slot)

Ground - Eight pins (shared by power return and signal return).

NOTE
Power is not bused between backplanes on any interconnecting bus cables.

4.6.8.4 **Maintenance and Spare Pins**

Maintenance Pins - There are four MSPARE pins per bus device slot assigned to maintenance (AK1, AL1, BK1, BL1). The maintenance pins on the basic LSI-11 system are not bused from module to module. Instead, at each bus device slot, the maintenance pins are shorted together as pairs. These pins must be shorted together for any module to operate. This allows the module to use these pins during initial testing as two separate points. This is used by DIGITAL for manufacturing tests only.

Spare Pins - Spare pins are allocated on the backplane as follows.

SSPARE - These eight pins are reserved for the particular use of a module or set of modules, either as test points or as intermodule communication. For intermodule communication, appropriate wires must be added to the backplane since these pins are not interconnected in any way. SSPARE lines cannot be used as bus connections.

PSPARE - These four pins are similar to SSPARE, except that they are located in a manner such that dc voltages will appear on the pins if a module is inserted backwards. Use of these pins is not recommended.

4.7 SYSTEM CONFIGURATIONS

LSI-11 bus systems can be divided into two types.

1. Systems containing one backplane
2. Systems containing multiple backplanes

---

*The maximum allowable current per pin is 1.5 A. +5 Vdc must be regulated to ±5%; maximum ripple: 100 mV pp. +12 Vdc must be regulated to ±3%; maximum ripple: 200 mV pp.

Before configuring any system, three characteristics for each module in the system must be known. These characteristics include:

1. Power consumption - +5 Vdc and +12 Vdc current requirements.

2. AC bus loading - the amount of capacitance a module presents to a bus signal line. AC loading is expressed in terms of ac loads where one ac load equals 9.35 pF of capacitance.

3. DC bus loading - the amount of dc leakage current a module presents to a bus signal when the line is high (undriven). DC loading is expressed in terms of dc loads where one dc load equals 105 microamperes (nominal).

Power consumption, ac loading, and dc loading specifications for each module are included in the Memories and Peripherals handbook.

NOTE

The ac and dc loads and the power consumption of the processor module, terminator module, and backplane must be included in determining the total loading of a backplane.

4.7.1 Rules for Configuring Single Backplane Systems (Figure 4-15)

1. The bus can accommodate modules that have up to 20 ac loads (total) before an additional termination is required. The processor has on-board termination for one end of the bus. If more than 20 ac loads are included, the other end of the bus must be terminated with 120 ohms.

2. A terminated bus can accommodate modules comprising up to 35 ac loads (total).

3. The bus can accommodate modules up to 20 dc loads (total).

4. The bus signal lines on the backplane can be up to 35.6 cm (14 in) long.

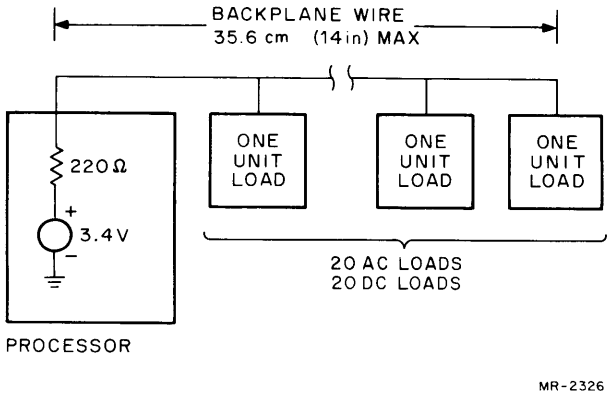


Figure 4-15 Single Backplane Configuration

4.7.2 Rules for Configuring Multiple Backplane Systems (Figure 4-16)

1. a. Up to three backplanes may compose the system.

   b. The signal lines on each backplane can be up to 25.4 cm (10 in) long.

2. Each backplane can accommodate modules that have up to 20 ac loads (total). Unused ac loads from one backplane may not be added to another backplane if the second backplane loading will exceed 20 ac loads. It is desirable to load backplanes equally, or with the highest ac loads in the first and second backplanes.

3. DC loading of all modules in all backplanes cannot exceed 20 loads (total).

4. Both ends of the bus must be terminated with 120 ohms. This means that the first backplane must have an impedance of 120 ohms (obtained via the processor 220 ohm terminations and a separate 220 ohm terminator), and the last backplane must have a termination of 120 ohms.

5. a. The cable(s) connecting the first two backplanes is 61 cm (2 ft) or greater in length.

   b. The cable(s) connecting the second backplane to the third backplane is 22 cm (4 ft) longer or shorter than the cable(s) connecting the first and second backplanes.

   c. The combined length of both cables cannot exceed 4.88 m (16 ft).

   d. The cables used must have a characteristic impedance of 120 ohms.

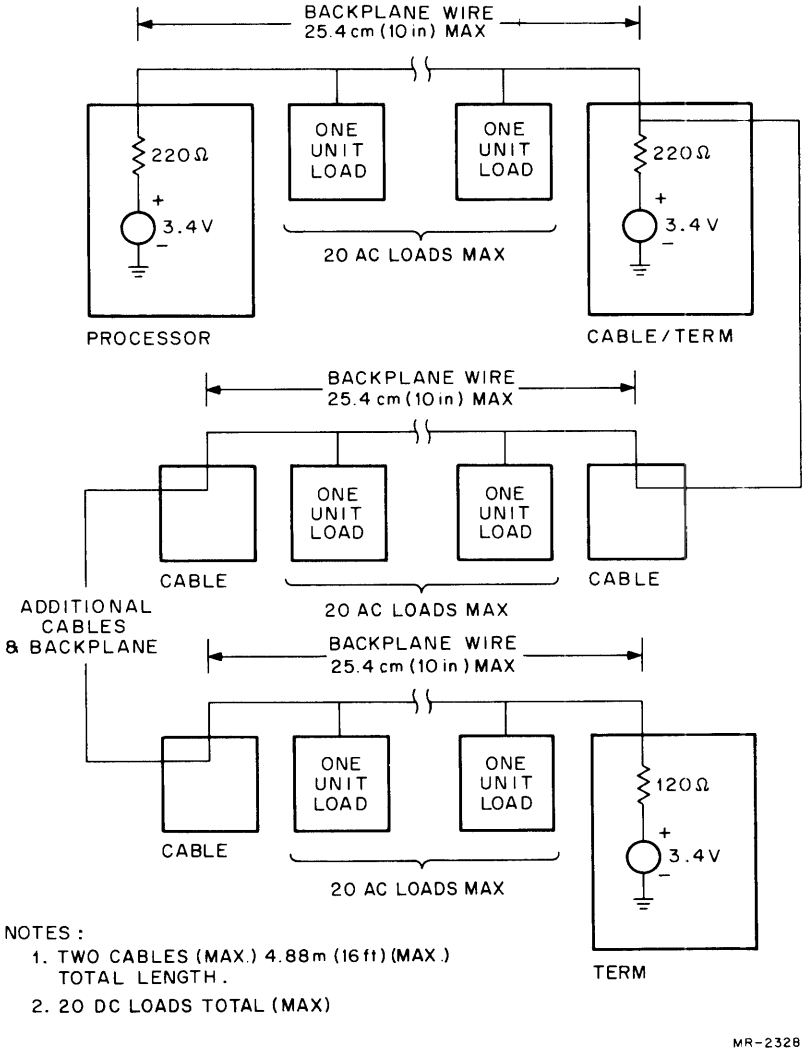


Figure 4-16 Multiple Backplane Configuration

4.7.3 Power Supply Loading

Total power requirements for each backplane can be determined by obtaining the total power requirements for each module in the backplane. Obtain separate totals for +5 V and +12 V power. Power requirements for each module are specified in the Memories and Peripherals handbook.

When distributing power in multiple backplane systems, do not attempt to distribute power via the LSI-11 bus cables. Provide separate, appropriate power wiring from each power supply to each backplane. Each power supply should be capable of asserting BPOK H and BDCOK H signals according to bus protocol; this is required if automatic power fail/restart programs are implemented, or if specific peripherals require an orderly power-down halt sequence. The proper use of BPOK H and BDCOK H signals is strongly recommended.

CHAPTER 5
PROCESSOR FUNCTIONAL DESCRIPTION

5.1 INTRODUCTION

A function block diagram of the KDF11-AA is shown in Figure 5-1. The heart of the processor is contained on three MOS/LSI chips. They are the data chip, the control chip and the memory management unit (MMU). The data and control chips are combined in a single 40-pin package. The MMU is packaged as one 40-pin chip.

The MOS chips communicate over two internal buses: the microinstruction bus (MIB) and the data address lines (DAL). The MIB is used for communication and control among the three MOS chips and to control the logic circuitry on the processor board. The DAL are used for transferring data between the MOS chips, and for transferring data to and from the processor and the LSI-11 bus.

This chapter discusses the functions of the logic contained on the processor board.

5.2 DATA CHIP

The data chip contains the PDP-11 general registers, the processor status word (PS), several working registers, the arithmetic and logic unit (ALU), and conditional branching logic. The data chip does the following.

1. Performs all arithmetic and logical functions

2. Handles all data and address transfers with the LSI-11 bus (except relocation, which is handled by the MMU; see Paragraph 5.4)

3. Generates most of the signals used for interchip communication and external system control

A typical microinstruction cycle starts when the data chip receives a 16-bit microinstruction from the control chip on the time-multiplexed, bidirectional MIB. During the first half of the cycle the register file is precharged, and the selected register(s) are read and sent part way through the ALU chain (i.e., operands are latched into the propagate and generate latches). Also during the first half of the cycle, control information is decoded from the microinstruction and output on the MIB for use by other chips and external logic. During the second half of the cycle the ALU operation is completed and the result is written into the appropriate register.

Output operations occur during the first half of the cycle when the contents of the selected source register are bused around the ALU logic directly to the output buffers. Input data is strobed into the data chip during the first half of the cycle, although it is not written into the register file until the second half.

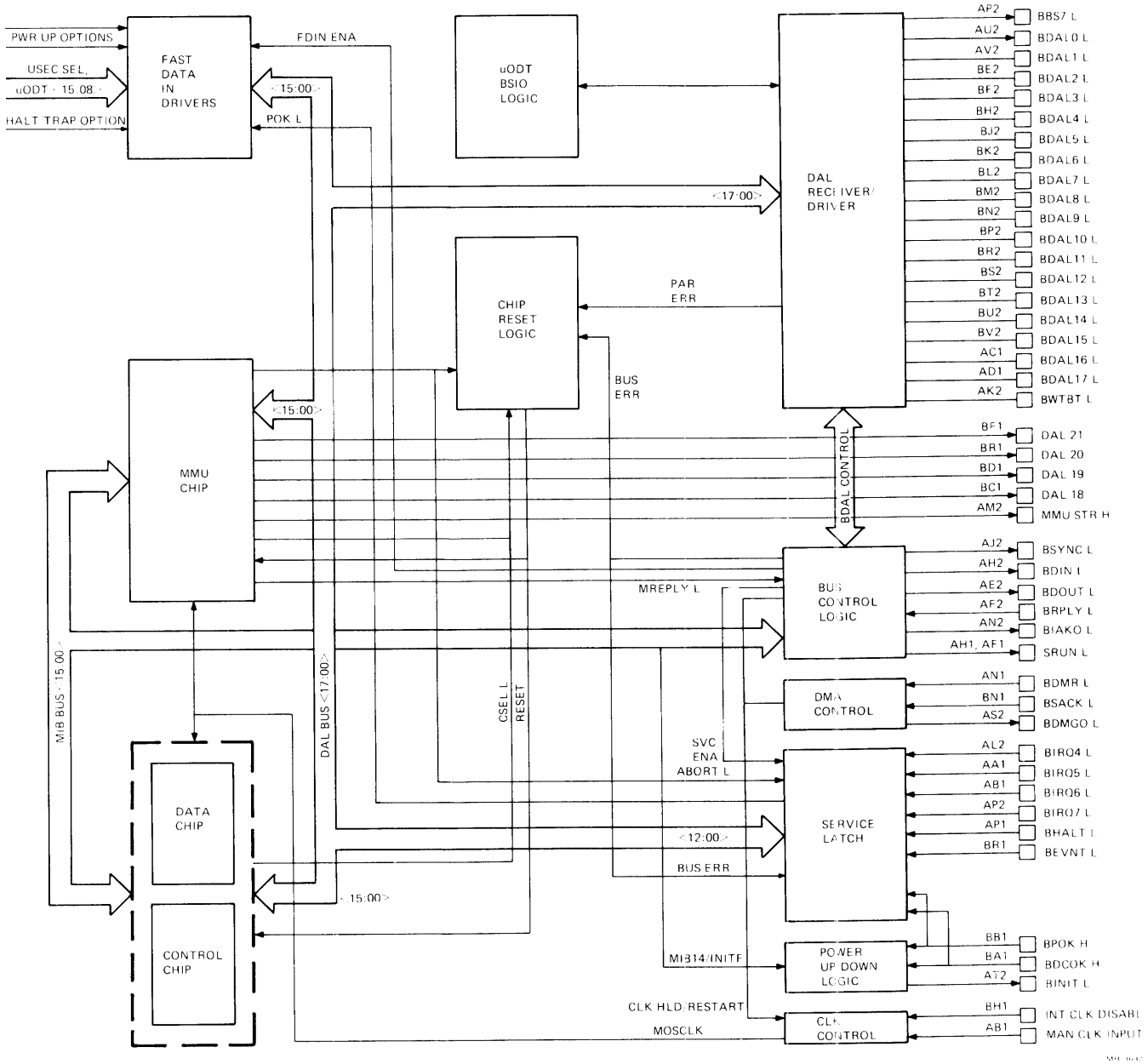


Figure 5-1 Processor Functional Block Diagram

5.3 CONTROL CHIP

The control chip contains the microprogram sequence logic and 552 words of microprogram storage in programmable logic arrays (PLA) and read-only memory (ROM) arrays.

During the course of a normal microinstruction cycle, the control chip accesses the appropriate microinstruction in the PLA or ROM, sends it along the MIB to the data and MMU chips for execution, and then generates the address for the next microinstruction to be accessed. The next address is constructed from either a next address field associated with the current microinstruction or, if a microprogrammed branch is to be executed, the target address contained within the microinstruction itself. The control chip operation is pipelined for better performance so that the next microinstruction is being accessed while the current one is being executed. This next address is then used in conjunction with various internal status and external service inputs to determine the microprogram sequence. The control chip accesses only its local storage. However, multiple chips (up to 32) can be cascaded with external buffering to provide additional microstore.

Chip Select (CSEL)

CSEL is an open collector line with a pull-up resistor. CSEL is routed to all MOS chips on the board except the MMU. The active control chip holds the line low. If a nonexistent control chip is selected by the microcode, the line is pulled high. This causes a control chip error and a trap to location 10_8.

5.4 MMU CHIP

The MMU chip serves two purposes: it provides the memory management function, and it provides storage for the FP11 floating point accumulators and status registers. This chip provides dual mode (user and kernel) address relocation of 18 bits. Sixteen-bit virtual addresses are received from the data chip via the data address lines (DAL), relocated to the appropriate 18-bit physical address, and then sent on the DAL to replace the original virtual address for transmission to the external system bus. The MMU chip contains the status registers and active page registers (PAR/PDR register pairs), as well as access protection and error detection capability. The MMU chip also provides the thirty-six 16-bit registers needed for operand storage, scratchpad areas, and status information storage during floating point operations.

The MMU chip is controlled by information received on the microinstruction bus (MIB) from both the data chip and the control chip, and by several discrete control inputs.

The KDF11-AA can operate without the MMU chip; however, the memory would be limited to 32K words and the floating point registers would not be available. For complete details of memory management capabilities refer to Chapter 8.

5.5 DATA-ADDRESS LINES (DAL)

The DAL bus is routed between all the MOS chips, along the processor board, and to the LSI-11 bus transceivers. The 16-bit DAL bus is time-multiplexed. During clock-high time, the DAL bus transfers data from the data chip to the other MOS chips or between the processor board and the MOS chips. During clock-low time, the DAL bus transfers service data (external and internal interrupt requests) from the board to the control chip. (The control chip receives service information and determines whether to interrupt or fetch the next instruction.)

5.6 MICROINSTRUCTION BUS (MIB)

The 16-bit microinstruction bus is common to all data and control chips. A subset of the MIB is routed to the MMU because it does not need access to all MIB control signals. A different subset of the MIB controls the processor board logic. These control functions are discussed in Paragraphs 5.6.1 - 5.6.6.

The MIB is time-multiplexed and is used for different functions during clock high and low times. During clock-high time, the MIB transfers control information from the data chip to all control chips, the MMU and the board logic. During clock-low time, the MIB transfers microinstructions from the active control chip to other control chips and the data chip.

5.6.1 MIB15/Memory Management Enable (MME)

During clock-high time, MIB15 carries MME from the MMU chip. MME is an active low signal. After being pulled low by the MMU chip, MME indicates to the processor board logic that a relocated-address microcycle should be performed. MME is also asserted low by the processor board during console ODT to allow access to greater than 32K words of memory without using the MMU chip.

5.6.2 MIB14/Initialize (INIT F)

During clock-high time, MIB14 contains an active low initialize signal (INIT F) used by the board logic to generate BINIT L. At the end of every clock-high time, the processor monitors INIT F. If INIT F is asserted low, the processor generates BINIT L onto the LSI-11 bus. DINIT L holds the INIT F flip-flop in the 0 state during power-up so that BINIT L is constantly driven onto the LSI-11 bus until DCOK H from the power supply goes high. (Refer to Paragraph 4.5.4 for power protocol.)

5.6.3 MIB13/Interrupt Acknowledge (IAK)

MIB13 contains IAK during clock-high time, and is used to generate BIAK L onto the LSI-11 bus. The highest priority device that is requesting an interrupt uses BIAK L and BDIN L as a signal to assert its interrupt vector on the LSI-11 bus. IAK occurs only during an input vector microcycle.

5.6.4 MIB12, 9, 8/Address-Input-Output (AIO) Codes

These three control lines along with two other signals, BUS CYC H and SYNC/DMA ENA H, are fed into the bus control PROM (Figure

5-2). The PROM decodes them to determine the type of microcycle currently executing within the MOS chips. The PROM outputs control various signals and perform the following functions.

1. CLK HOLD H stops the clock generator in the high state for asynchronous data transfers. This signal is used to free up the bus during DMA or when bus cycle is in progress.

2. BUS ENA H enables LSI-11 bus drivers during address and data-out bus cycles only.

3. DIN CYC H drives the BDIN L bus driver.

4. DOUT CYC H drives the BDOUT L bus driver.

5. WTBT H drives BWTBT L bus signal whenever an address microcycle is followed by a data-out microcycle and whenever a byte data transfer is in progress.

6. CLK STUT H, for clock control, is used to extend the clock-high time of address microcycles and nonbus data-in and data-out microcycles (refer to Paragraph 5.10.2.).

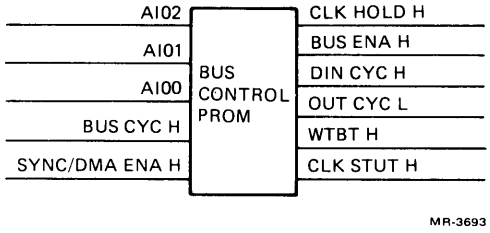


Figure 5-2 Bus Control PROM

5.6.4.1 BUS CYC H - This signal is a function of the sync signal from the data chip (SYNCF). If a data transfer to or from the data chip is internal to the MOS chip set, then BUS CYC H is low. If it is an external bus transfer, then BUS CYC H is high. In the case of internal data transfers, the clock is lengthened one clock tick to allow the chip set more time to complete its internal transfer. In the case of bus-type data transfers, the bus drivers (DOUT transfers) or receivers (DIN transfers) are enabled, and the master clock is halted in the high state, waiting for BREPLY L from the bus.

5.6.4.2 SYNC/DMA ENA H - SYNC/DMA ENA H indicates that another peripheral is still bus master or that the last bus cycle is not yet complete. Its function is to prevent the MOS chip set from attempting to use the LSI-11 bus when the bus is still being used.

The master clock is halted during LSI-11 bus data transfers while transferring data to the peripheral or receiving data from the peripheral. Once this is accomplished the master clock starts up again and microinstructions are again executed. Concurrently, the processor is terminating the previous bus cycle. Because the processor cannot terminate the cycle until BREPLY has been deasserted by the peripheral (there is no time limit on this action taking place according to LSI-11 bus protocol), it is possible for the previous bus cycle to still be active when the chip set is ready for the next bus cycle. SYNC/DMA ENA H causes the clock to stop in the address cycle in this case and halts the chip set in the address microcycle until the previous bus cycle is properly completed (BSYNC L negated).

5.6.5 MIB03/GPO 3

Control code GPO 3, driven by the data chip, is detected by the GPO decode logic (Figure 5-3) and properly timed to produce FDIN ENA L. This signal is used to gate power-up information from jumpers on the processor board. Refer to Chapter 2 for information on jumpers.

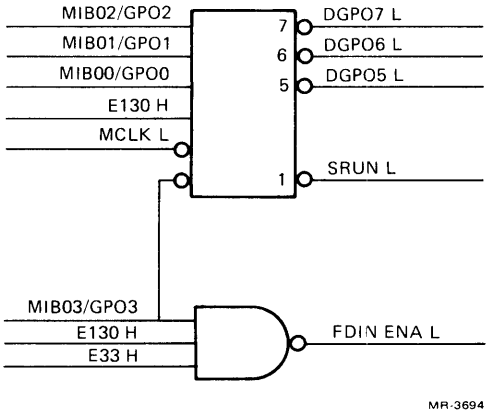


Figure 5-3 GPO Decode Logic

5.6.6 MIB02, 01, 00/GPO 2, 1, 0

GPO 2, 1, and 0 are driven by the data chip during clock-high time and perform control functions on the processor board. These signals are decoded by the logic shown in Figure 5-3. The decoded output is shown in Table 5-1.

5.7 BSYNC L LOGIC

The logic shown in Figure 5-4 controls the assertion of B SYNC L onto the LSI-11 bus. The start of all bus cycles <DATI, DATO(B), DATIO(B)> is signaled by SYNCF L going low on MIB07 of the data chip during clock-high time. SYNCF L is clocked into both the BUS CYC flip-flop and the SYNCF flip-flop at the end of clock-high

time. A set BUS CYC flip-flop indicates to the DMA logic that the processor is going to use the bus, and therefore a DMA request cannot be granted.

Table 5-1 General-Purpose Output Signals

GPO2	GPO1	GPO0	Output Name	Function
1	1	1	DGP07 L	Loads the two highest order address bits into a latch while in micro-ODT. This allows 18-bit addressing to be accomplished without using the memory management unit while in ODT.
1	1	0	DGP06 L	Clears the power-fail flip-flop after the power-fail sequence has been executed in microcode.
1	0	1	DGP05 L	Clears the event flip-flop after the event interrupt has been serviced in microcode.
0	0	1	SRUN L	Generates a low-going pulse that is routed directly to edge fingers AF1, AH1 whenever an instruction fetch occurs. The pulse also occurs whenever a character is received from the serial line unit while in micro-ODT. This signal can be used to cause a steady RUN indication while the processor is executing microinstructions and a flashing indication when typing characters in console-ODT.

The SYNCF flip-flop feeds the BSYNC flip-flop. This flip-flop is strobed every microcyle, 33 ns after the start of clock-high time. Thus, the BSYNC flip-flop will be set 33 ns into clock-high time of the microcycle after the address microcycle. This delay is necessary to allow sufficient address set-up time on the bus. Once the BSYNC flip-flip is set, it drives the bus transceiver and asserts BSYNC L onto the LSI-11 bus.

Once the BSYNC flip-flop is set, it remains set until changed by a low level from the BSYNC RESET logic on its reset line. The SYNCF signal from the data chip clears on the data-in or data-out microcycle that completes the bus cycle. The BSYNC flip-flop is cleared by BRPLY from the LSI-11 bus. SYNC REP L and RESTART END are both functions of BRPLY L. The BSYNC RESET logic uses SYNC REP L and RESTART END to generate a pulse on the rising edge of BRPLY L. BUS CYC L and DOUT BLOCK L block the BSYNC flip-flop from being cleared after the DATI portion of a DATIO cycle. These

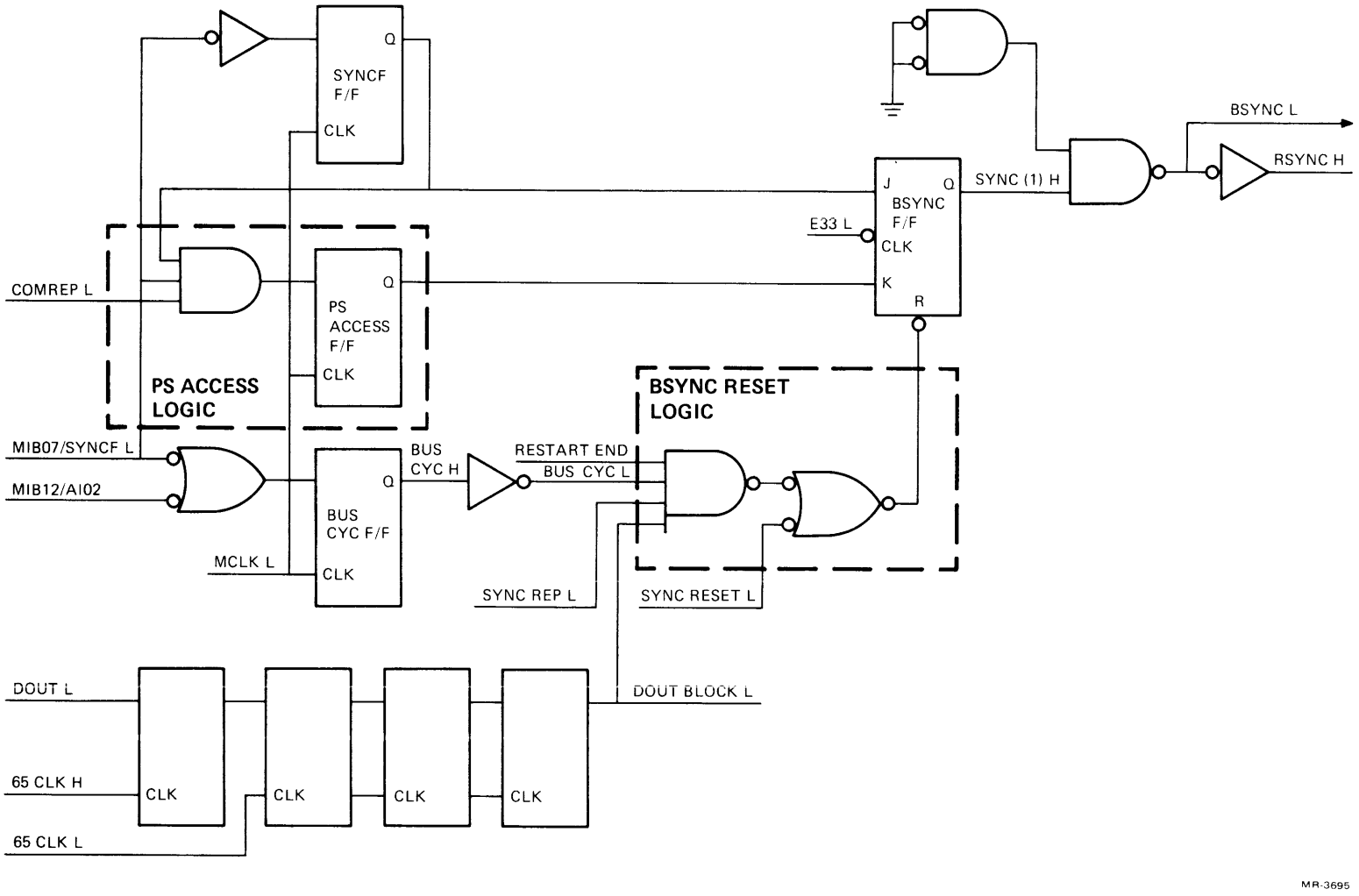


Figure 5-4 BUS SYNC Logic

signals also prevent the BSYNC flip-flop from being cleared for at least 175 ns after BDOUT L is cleared (as per bus specifications). SYNC RESET L clears the BSYNC flip-flop on power-up if a bus timeout occurs, and prevents it from setting when an MMU abort occurs.

PS Access Logic

The PS (processor status word) access logic feeds the K input of the BSYNC flip-flop and is used only when the PS is accessed. The PS is contained in the data chip. When 777776_8 (the address of the PS in the data chip) appears on the DAL during an address microcycle, the data chip decodes the address and access to the PS is allowed. The bus cycle is terminated by deasserting the SYNCF line without allowing a DATI or DATO AIO code.

The PS access flip-flop stores this condition until the start of the next clock-high time. This signal is fed to the K input of the BSYNC flip-flop and resets BSYNC at the start of the next microcycle.

5.8 DIRECT MEMORY ACCESS (DMA)

DMA on the KDF11-AA board allows peripherals to gain control of the LSI-11 bus from the processor and transfer data directly between a peripheral and memory. In this way, data transfers can occur at the full memory speed rather than having the processor transfer data words one at a time between the peripheral and memory. A speed gain of about 12 to 1 over regular programmed transfers is gained by this technique.

The signals required for the DMA logic are the following.

BDMR L — This is the DMA request signal. A peripheral device asserts this line when it is ready to use the bus for a DMA transfer. This line is common to all peripheral devices.

BDMGO L — This DMA grant signal is issued by the processor in response to a DMA request. By asserting this line, the processor indicates that it will halt processing as soon as the current bus cycle is completed. The processor will also disable all bus control lines and data-address lines (BDAL) so that the peripheral device can use them to control the bus. The BDMR line is common to all peripheral devices. BDMGO L is a daisy-chained signal. Any memory or peripheral device that does not want to use the bus simply passes the signal on. The first (physically closest to the processor) device on the bus desiring to use the bus "takes the grant;" i.e., blocks the signal from being passed on. Therefore the peripheral closest to the processor requesting the bus at

the time the grant is issued gets to use the bus. In order to prevent hogging of the bus by peripheral devices nearest the processor, DMA transfer time must be as short as possible.

BSACK L — This DMA acknowledge signal is issued by the peripheral device taking control of the bus. This signal completes the handshake between the processor and the peripheral device and indicates to the processor that a device has taken the bus.

No SACK Timeout — In LSI-11 bus systems there is a possibility that a device can request use of the bus and then not take the DMA grant signal. The no SACK timeout feature clears the DMA grant signal and returns bus mastership to the processor if no peripheral device has issued BSACK L within 18 microseconds after the processor has issued a grant. This prevents a potential bus lockup problem in which the processor has given up the bus but no one has taken the grant.

5.8.1 DMA Logic

The DMA logic is shown in Figure 5-5. BDMR L signals are received from the bus on edge pin AN 1 and synchronized with the processor high-frequency clock through a high-speed synchronizer. This signal is called SYDMR for "synchronized DMR." The SYDMR signal is gated with the signal BUS CYC L to block DMA requests from reaching the DMA ENA flip-flop when a bus cycle is in progress. The gated signal is called GADMR for "gated DMR." The DMA ENA flip-flop samples the GADMR line at the beginning of every clock-high time (about every 290 ns). When a valid GADMR is latched into the DMA ENA flip-flop, the DMA cycle is started. Note that DMA request is always taken unless the processor is currently in a bus cycle. This is necessary to provide fast response to DMA requests. (Refer to Paragraph 5.8.2, DMA Latency.)

Once DMA ENA is latched, DMA grant is issued on the LSI-11 bus approximately 65 ns later by the DMA ENA H being clocked into the DMA grant flip-flop. Granting the DMA request also starts the timer which is set for 18 microseconds. At exactly the same time DMA grant is enabled, the DMA bus disable flip-flop disables the BDAL bus drivers on the processor board. The DMA ENA (1) L signal also blocks any further clock restarts from occurring until the DMA cycle that is just starting is completed. It does this by blocking the AND inputs to the clock restart logic.

Once DMA grant is issued, the processor board waits for a BSACK L signal indicating that a peripheral device has taken the DMA grant. The BSACK L line is monitored by a bus receiver; an active BSACK L resets the no SACK timeout timer which clocks a 1 into the DMA restart flip-flop. The DMA restart flip-flop is now armed.

As soon as the bus is given up by the current DMA master, this flip-flop will allow the DMA rearbitration process to restart. This occurs when BSACK L and BSYNC L are deasserted as the bus master gives up the bus. These signals, along with the armed DMA restart flip-flop, satisfy the logic which feeds the rearbitration logic and a restart/rearbitration takes place.

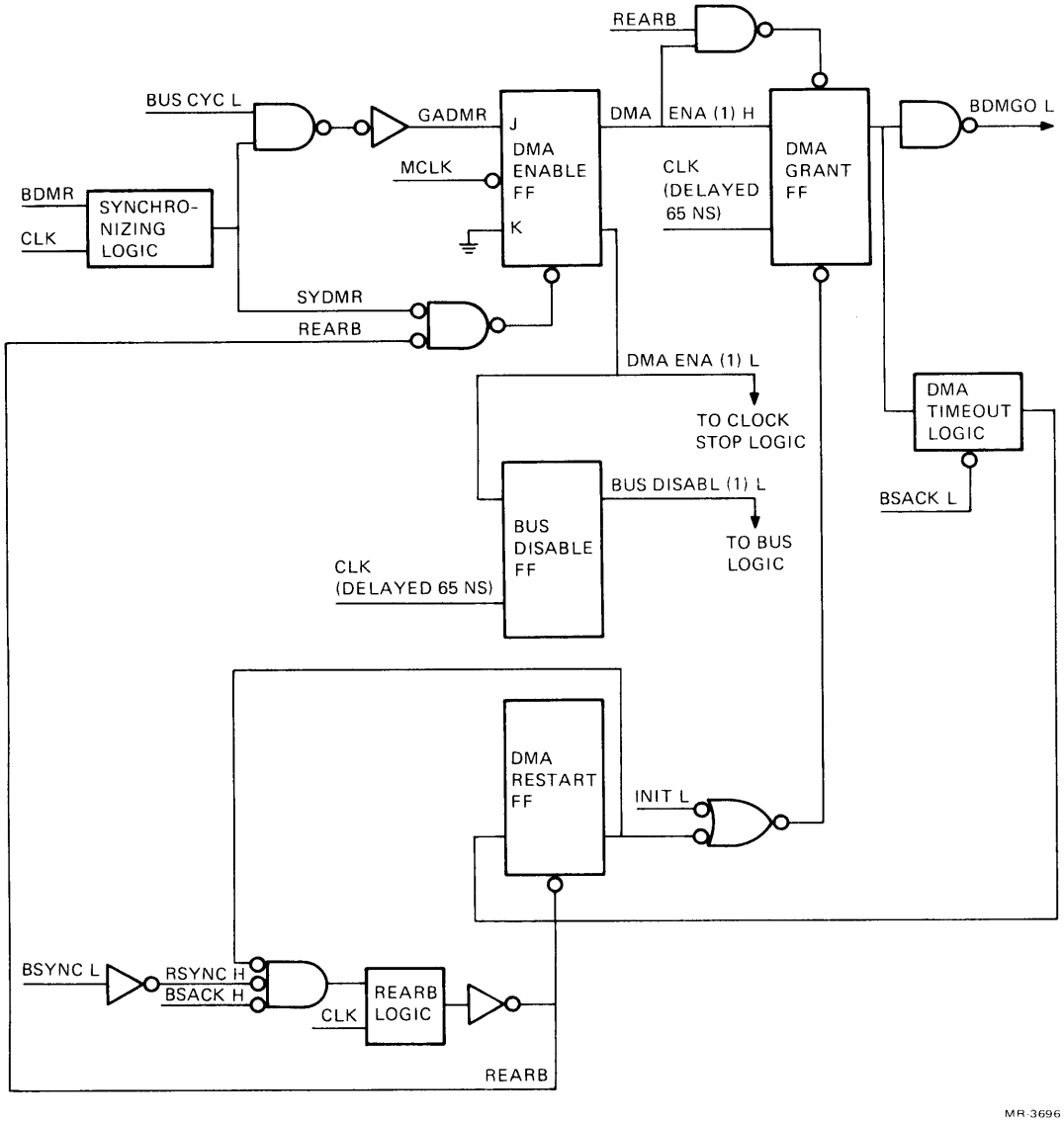


Figure 5-5 DMA Logic

According to system protocol the processor is the lowest priority bus master. When a bus master gives up the bus, the processor should immediately check for another pending request. If another request is pending, another BDMGO is reissued and a new peripheral

takes control. In the KDF11-AA, rearbitration takes place each time the bus is given up. If DMA requests are arriving at too great a rate, it is possible to have the processor constantly arbitrating among bus masters. This effect can be illustrated by holding the BDMR L line low which blocks any instruction fetches by the processor.

The rearbitration logic synchronizes the input signal with the high-frequency clock. The output is inverted to become REARB. Low-going REARB is gated with the synchronized DMA request (SYDMR). If there is no new DMA request, SYDMR is low and the REARB signal clears the DMA enable flip-flop. This event allows MCLK to be restarted (if stopped) on the next tick of the high-frequency clock. Since the DMA enable flip-flop is cleared, the next clock pulse does not set the DMA grant flip-flop and the processor is bus master again.

If a new DMA request is pending, the sequence is different. REARB is blocked from clearing the DMA enable flip-flop by a high SYDMR signal. Sixty-five ns later the DMA grant flip-flop is set and the DMA sequence starts again. Since the DMA enable flip-flop is never reset, the DMA ENA (1) L signal is constantly low and MCLK is never restarted.

5.8.2 DMA Latency

DMA latency is the time from when the DMA request arrives at the processor until BDMGO is put on the bus. The maximum DMA latency is important because of data loss problems. For example, once the heads of a disk drive are over the proper sector, the disk controller must become bus master within a certain period of time. If it does not, information will overflow the temporary data buffers in the disk drive interface and cause data-late errors. Since the KDF11-AA does not grant bus mastership during ongoing bus cycles, worst-case DMA latency occurs when the DMA request arrives just before the start of the longest bus cycle (DATIO). In this case the grant will be issued after the cycle has completed.

5.9 CLOCK GENERATOR CIRCUITRY

The KDF11 chip set clock can be suspended in the high state indefinitely, but can only remain in the clock low state for a limited period of time to avoid loss of internal chip data. A twisted ring oscillator, shown in Figure 5-6, is used with a high-frequency crystal clock input to generate the required clock signals that control the MOS/LSI chips. The TTL level output of the ring oscillator (MCLK H) is driven through a high-voltage clock buffer/driver to produce the high-voltage CHIP CLK that drives the MOS chips.

5.9.1 Initialization

When the processor receives +5 Vdc and +12 Vdc, the ring oscillator is initialized and held in this state until BDCOK H is asserted by the power supply (or the wake-up circuit). The initialization circuitry is shown in Figure 5-7. The output of

the second stage of the DCOK H synchronizer circuit holds START H low. The processor board initializes with MCLK H = 1 and all three stages of the ring oscillator also equal 1 (E65H, E130H, E195H). When DCOK H goes high, it is first synchronized with the high-frequency clock (65CLK H) and then releases the ring oscillator from its initialized state. The synchronizer is necessary because DCOK H is asynchronous to any circuitry on the processor board and feeding DCOK H directly into the ring oscillator could lead to a truncated first cycle of the processor. Once the oscillator is freed, it immediately causes MCLK H to go low and enters the clock-low state.

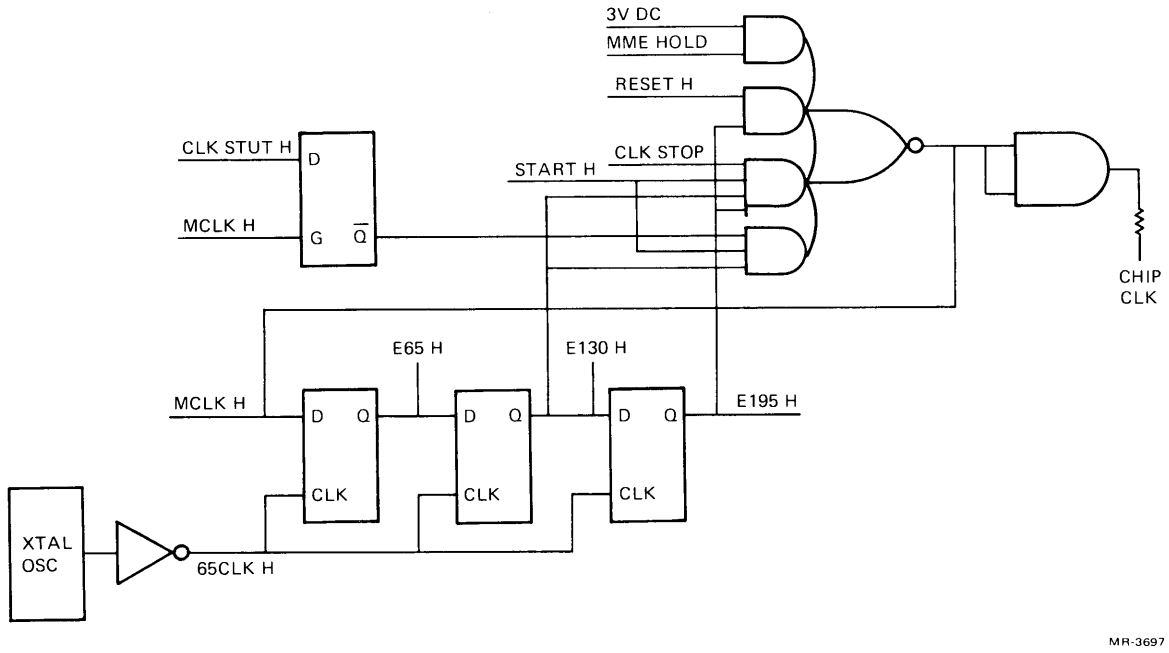


Figure 5-6 Clock Generator

5.9.2 Wake-Up Circuit

The "wake-up" circuit on the KDF11-AA module consists of a diode, a resistor, a capacitor, and a Schmidt trigger inverter, all shown at the left in Figure 5-7. This circuit provides automatic generation of BDCOK H 50 ms after the +5 V supply is turned on. For the circuit to function, the +12 V must be applied before or at the same time the +5 V is applied, and the rise time of the +5 V supply must be no greater than 50 ms.

5.9.3 Single-Step Circuit

The single-step circuit is shown in the lower portion of Figure 5-7. This circuit can be used in conjunction with an external circuit to stop the processor (i.e., hold the clock indefinitely high) in the bus data-in or data-out part of the cycle at a selected address. The external circuit must monitor the BDAL lines and compare the address issued by the processor at BSYNC L time with a desired stop address or addresses. If a valid compare

occurs, the external circuit should pull SINGLE STEP to a logic low level as soon as BDIN L or BDOUT L appears on the bus. The processor will then stop in the bus data-in or data-out microcycle and the data driven from the processor (in the case of data-out transfers) or received from the selected address (in the case of data-in transfers) can be observed on the BDAL lines and any other internal points of the system can be probed manually. The processor can be released from this state by releasing the single-step line and will resume executing instructions.

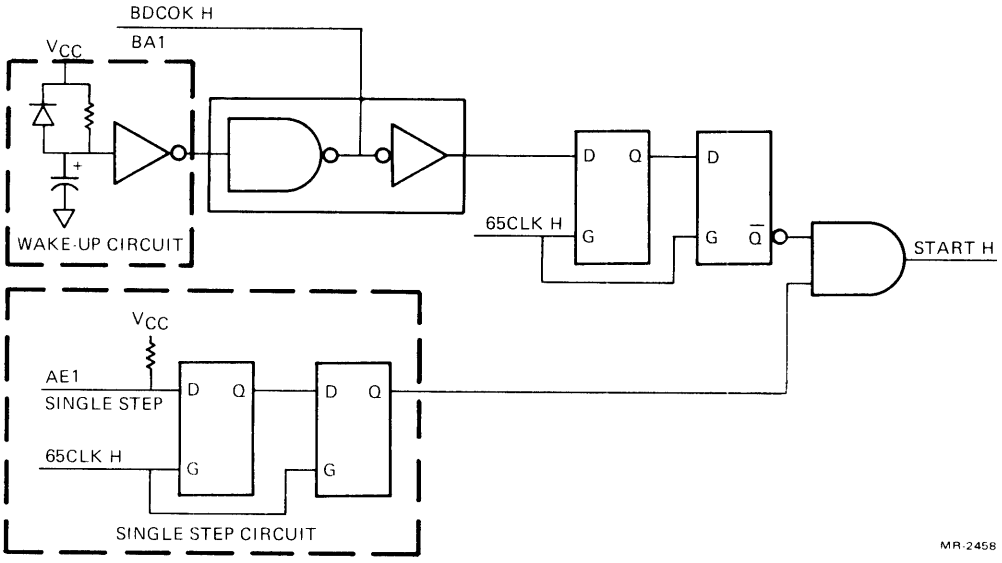


Figure 5-7 Clock Generator Initialization Circuitry

5.10 CLOCK GENERATOR CYCLES

The clock generator is capable of producing a normal cycle and four variations of the normal cycle used for special functions.

5.10.1 Normal Cycle

The normal cycle consists of two cycles of the high-frequency clock in the high state and two cycles in the low state. For this type of cycle, START H is constantly high, RESET H is low, and CLK STOP is low. Figure 5-8 shows this cycle.

5.10.2 Clock Stutter Cycle

The clock stutter cycle is generated on all address microcycles and for all internal data transfers among the MOS chips. It is the same as the normal cycle discussed above except that the clock-high time is extended from two cycles of the high-frequency clock to three. This stretched or "stuttered" clock time allows the DAL lines to settle before the address is driven out onto the bus. The cycle also allows extra time for data transfers between MOS chips.

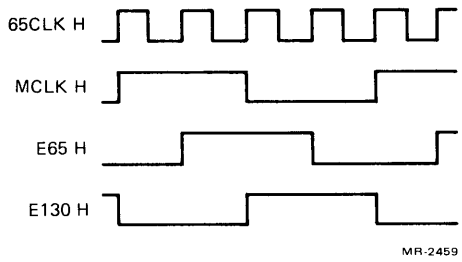


Figure 5-8 Normal Clock Cycle

The cycle is generated by the CLK STUT H signal from the bus control PROM (see Paragraph 5.6.4) being fed through a transparent latch that is enabled during phase time. The output of the latch inhibits the E130 H input to the feedback loop from causing MCLK H to go low. Instead, the ring oscillator output drops when E195 H goes high, one cycle of the high-frequency clock later. The stutter cycle is shown in Figure 5-9.

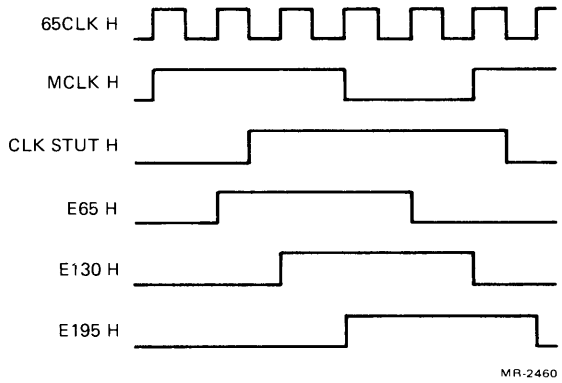


Figure 5-9 Clock Stutter Cycle

5.10.3 Clock Stop Cycle

The clock stop cycle is generated during bus data-in and bus data-out transfers when the chip set must wait for a REPLY from the LSI-11 bus before it can continue. It is also used to prevent the chip set from continuing past the address microcycle portion of a bus cycle when a DMA device has bus "mastership." For a clock stop cycle the bus control PROM generates CLK STUT H and CLK HOLD H. The CLK STUT H signal stretches the clock-high time from two to three high-frequency clock cycles. The CLK HOLD H signal is clocked into a flip-flop (the CLK STOP flip-flop) every cycle after two cycles of the high-frequency clock. The output of this flip-flop, CLK STOP, goes low and holds MCLK H in the high state until the CLK STOP flip-flop is cleared. In the case of a bus

data-in or data-out cycle, the flip-flop is cleared 200 ns after REPLY has been received from the addressed device, or, in the DMA case, 130 ns after the DMA device has given bus mastership back to the processor. This cycle is shown in Figure 5-10.

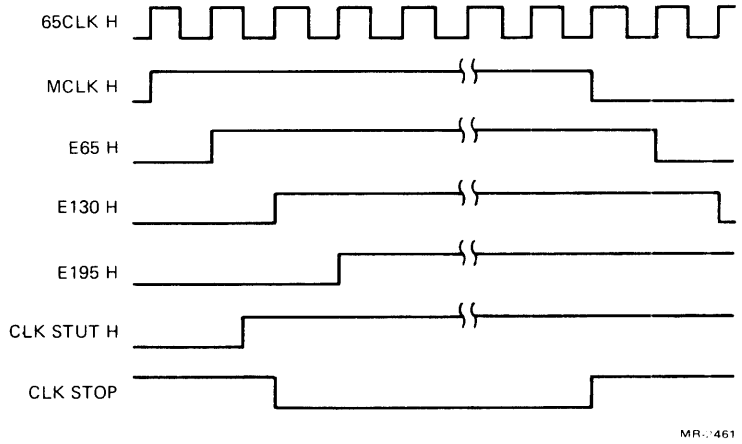


Figure 5-10 Clock Stop Cycle

5.10.4 Memory Management Cycle

This cycle occurs during address microcycles when the memory management chip is present and is enabled to do address relocation (enabling of the MMU is under software control). The MMU chip signals to the processor board that it wants to do address relocation by asserting the MIB line MME L at the end of clock-high time of an address microcycle. The relocation circuit, shown in Figure 5-11, detects the MME L signal and causes MME HOLD to be asserted high 65 ns into clock-low time of the address microcycle. MME HOLD holds MCLK in the clock low state for a total of five high-frequency clock periods or 325 ns. A pulse is produced 195 ns into clock-low time which passes through the OR gate and causes DALFF CLK to latch the relocated address, driven out of the MMU chip onto the DAL bus at this time, into the DAL driver flip-flops. Since the BDAL bus is continuously enabled during this time, the relocated address is immediately driven onto the BDAL lines. The relocation timing circuitry automatically clears itself after five high-frequency clock periods and releases MME HOLD which immediately allows MCLK H to go high, ending clock-low time.

5.10.5 Reset Cycle

The final variation of the basic cycle is when a CHIP RESET occurs. CHIP RESET is generated by the circuit shown in Figure 5-12 and occurs for any one of five error conditions that warrant immediate attention by the chip set. RESET H is enabled 65 ns

into clock-low time and causes the ring oscillator to stretch clock-low time from two periods of the high-frequency clock to three. This extended clock-low time allows CHIP RESET to initialize the MOS/LSI chips.

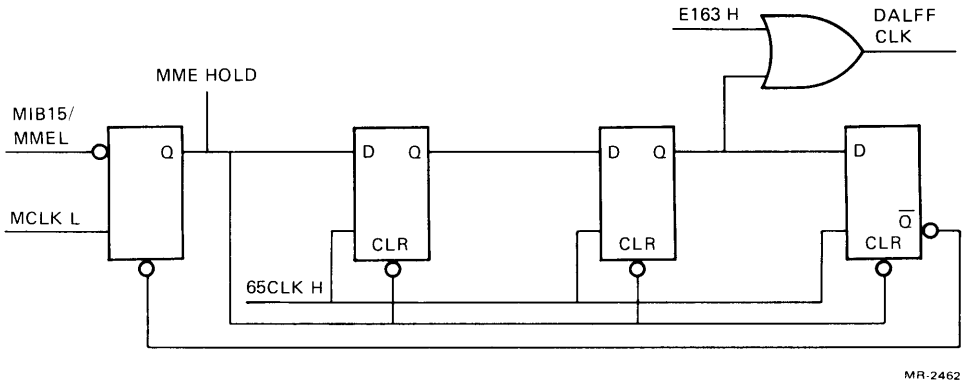


Figure 5-11 Relocation Timing Circuit

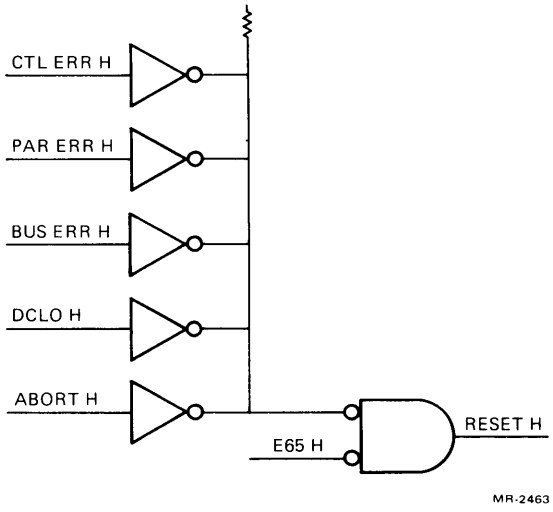


Figure 5-12 Reset Circuit

Chip Reset/RESET

RESET is routed to all MOS chips except the MMU. If an interrupt requiring immediate attention occurs, the line is asserted high. The following five interrupts require immediate attention.

1. Control error - Nonexistent control chip selected by the microcode. A trap to location 10_8 occurs.

2. Bus error - Nonexistent memory location accessed. A trap to location 4_8 occurs.

3. Parity error - A parity error detected on a current read from memory. A trap to location 114_8 occurs.

4. MMU abort - The MMU has aborted a mapped reference. A trap to location 250_8 occurs for any of the following reasons.

   - o The memory location referenced is not present in the current user's protected address space.
   - o An attempt is made to modify a write-protected location.
   - o The user is exceeding his allotted page boundary.

5. DC Power-Up - Upon power-up the processor forces two sequential RESETS to the chip set to initialize all internal chip registers. The dc power-up line then clears and is not activated again while dc power is on. (Refer to Paragraph 4.5.4 for power protocol.)

CHAPTER 6
ADDRESSING MODES

6.1 INTRODUCTION

In the KDF11-AA all memory reference addressing is accomplished using the eight general-purpose registers. In specifying an address of the data (operand address), one of the eight registers and one of several addressing modes are selected. Each memory reference instruction specifies the following.

1. Function to be performed (operation code)

2. General-purpose register to be used when locating the source and/or destination operand

3. Addressing mode, which specifies how the selected registers are to be used.

Many capabilities are provided by the combination of the addressing modes and the instruction set. The KDF11-AA is designed to handle structured data efficiently and with flexibility. The general-purpose registers implement these functions in the following ways.

1. Act as accumulators: they hold the data to be manipulated

2. Act as pointers: the content of the register is the address of the operand rather than the operand itself, allowing automatic stepping through memory locations.

3. Act as index registers: the content of the register is added to the second word of the instruction to produce the address of the operand. This capability allows easy access to variable entries in a list.

Utilization of the registers for both data manipulation and address calculation results in a variable length instruction format. If registers alone are used to specify the data source, only one memory word is required to hold the instruction. In certain modes, two or three words may be utilized to hold the basic instruction components. Special addressing mode combinations enable temporary data storage for convenient dynamic handling of frequently accessed data. This is known as stack addressing. Programming techniques utilizing the stack are discussed in Chapter 10. Register 6 is always used as the hardware stack pointer, or SP. Register 7 is used by the processor as its program counter (PC). Thus, the register arrangement to be considered in conjunction with instructions and with addressing modes is: registers 0-5 are general-purpose registers, register 6 is the hardware stack pointer, and register 7 is the program counter. The full KDF11-AA instruction set and instruction formats are explained in Chapter 7.

For the purpose of clearly illustrating the use of the various addressing modes, the following instructions and symbols are used in this chapter.

Mnemonic	Description	Octal Code
CLR	Clear (Zero the specified destination.)	0050DD
CLRB	Clear Byte (Zero the byte in the specified destination.)	1050DD
INC	Increment (Add 1 to contents of destination.)	0052DD
INCB	Increment Byte (Add 1 to the contents of the destination byte.)	1052DD
COM	Complement (Replace the contents of the destination by their logical 1's complements; each 0 bit is set and each 1 bit is cleared.)	0051DD
COMB	Complement Byte (Replace the contents of the destination bytes by their logical 1's complements; each 0 bit is set and each 1 bit is cleared.)	1051DD
ADD	Add (Add source operand to destination operand and store the result at destination address.)	06SSDD

DD = destination field (6 bits)
SS = source field (6 bits)
() = contents of

6.2 INSTRUCTION FORMATS

The instruction format for the first word of all single operand instructions (such as clear, increment, test) is shown in Figure 6-1.

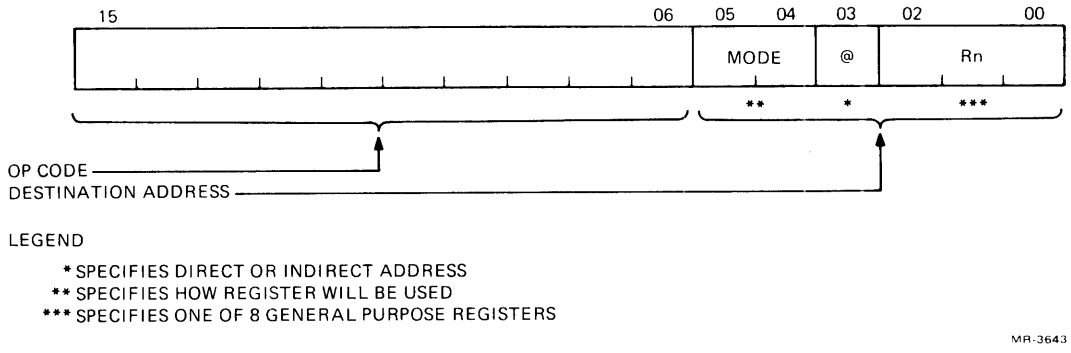


Figure 6-1 Single Operand Instruction Format

The instruction format for the first word of the double operand instruction is shown in Figure 6-2.

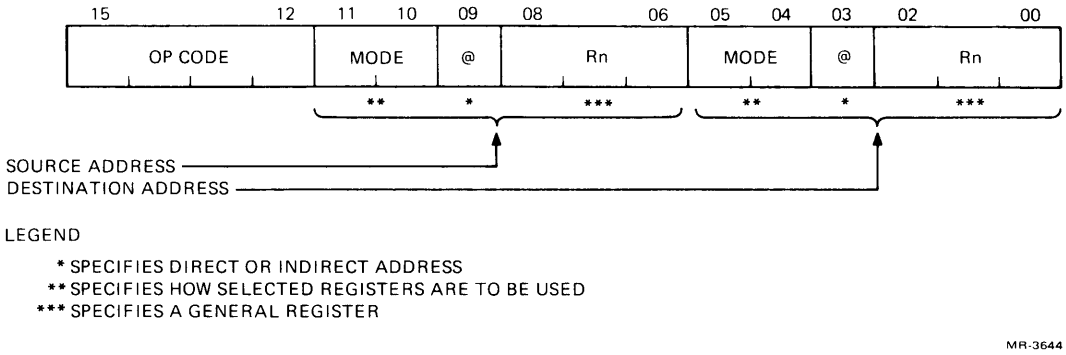


Figure 6-2 Double Operand Instruction Format

6.3 ADDRESSING MODES

Instruction bits <5:3> specify the binary code of the addressing mode chosen. The four direct addressing modes are the following.

1. Register
2. Autoincrement
3. Autodecrement
4. Index

When bit 3 of the instruction is set, indirect addressing is specified and the four basic modes become deferred modes. In a register-deferred mode, the content of the selected register is taken as the address of the operand. In the other deferred modes, the content of the register specifies the address of the operand, rather than the operand itself. Prefacing the register operand(s) with an @ sign or placing the register in parentheses indicates to the MACRO-11 assembler that deferred addressing mode is being used.

The indirect addressing modes are as follows.

1. Register deferred
2. Autoincrement deferred
3. Autodecrement deferred
4. Index deferred

Program counter (PC or register 7) addressing modes are as follows.

1. Immediate
2. Absolute
3. Relative
4. Relative deferred

The KDF11-AA addressing modes are explained and shown in examples in the following pages. They are summarized in Paragraphs 6.3.10 - 6.3.13.

6.3.1 Register Mode MODE 0 Rn

Register mode provides faster instruction execution. There is no need to reference memory to retrieve an operand. Any of the general registers can be used as accumulators. The operand is contained in the selected register. Assembler syntax requires that a general register be defined as follows.

R0 = %0
R1 = %1
R2 = %2

% sign indicates register definition.

Register Mode Examples

Symbolic	Instruction Octal Code	Description
INC R3	005203	Add 1 to the contents of R3.

The example is shown in Figure 6-3.

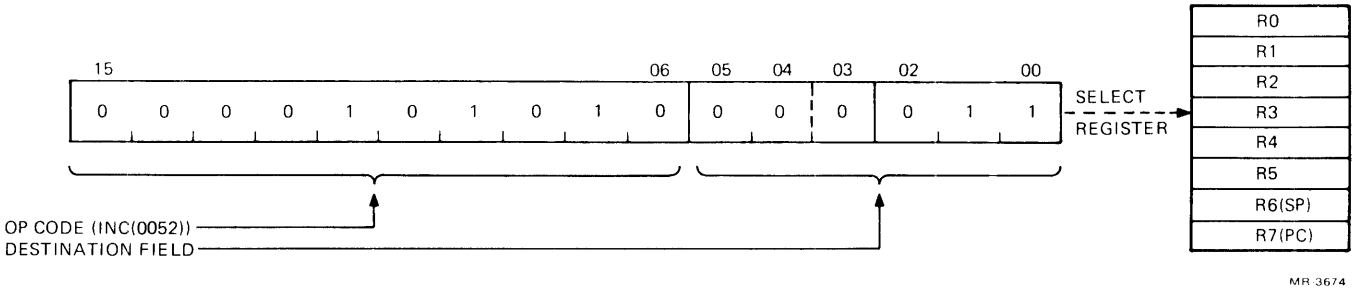


Figure 6-3 Register Mode Increment Example

Symbolic	Instruction Octal Code	Description
ADD R2, R4	060204	Add the contents of R2 to the contents of R4, replacing the original contents of R4 with the sum.

The example is shown in Figure 6-4.

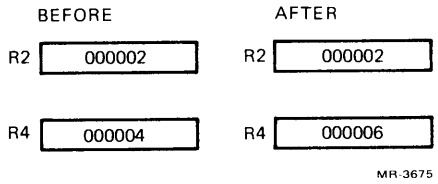


Figure 6-4 Register Mode Add Example

6.3.2 Register Deferred Mode MODE 1 (Rn)

In register deferred mode, the address of the operand is stored in a general-purpose register. The address contained in the general-purpose register directs the CPU to the operand. The operand is located outside the CPU, either in memory, or in an I/O register.

This mode is used for sequential lists, indirect pointers in data structures, top of stack manipulations, and jump tables.

Register Deferred Mode Example

Symbolic	Instruction Octal Code	Description
CLR (R5)	005015	The contents of the location specified in R5 are cleared.

The example is shown in Figure 6-5.

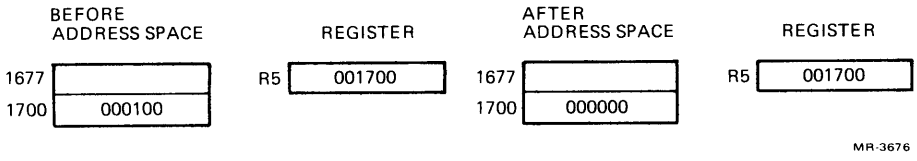


Figure 6-5 Register Deferred Mode Example

6.3.3 Autoincrement Mode MODE 2 (Rn)+

In autoincrement mode, the register contains the address of the operand, and the address is automatically incremented after the operand is retrieved. The address then references the next sequential operand. This mode allows automatic stepping through a list or series of operands stored in consecutive locations. When an instruction calls for mode 2, the address stored in the

register is autoincremented each time the instruction is executed. It is autoincremented by 1 if byte instructions are being used, by 2 if word instructions are being used.

Autoincrement Mode Example

Symbolic	Instruction Octal Code	Description
CLR (R5)+	005025	Contents of R5 are used as the address of the operand. Clear selected operand and then increment the contents of R5 by 2.

The example is shown in Figure 6-6.

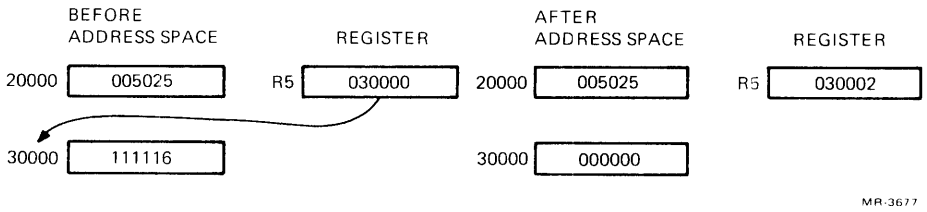


Figure 6-6 Autoincrement Mode Example

6.3.4 Autoincrement Deferred Mode MODE 3 @(Rn)+

In autoincrement deferred mode, the register contains a pointer to an address. The + indicates that the pointer in R2 is incremented by 2 after the address is located. Mode 2, autoincrement, is used to access operands that are stored in consecutive locations. Mode 3, autoincrement deferred, is used to access lists of operands stored anywhere in the system; i.e., the operands do not have to reside in adjoining locations. Mode 2 is used to step through a table of volumes; mode 3 is used to step through a table of addresses.

Autoincrement Deferred Example

Symbolic	Instruction Octal Code	Description
INC @(R2)+	005232	Contents of R2 are used as the address of the address of the operand. The operand is increased by 1, and contents of R2 are incremented by 2.

The example is shown in Figure 6-7.

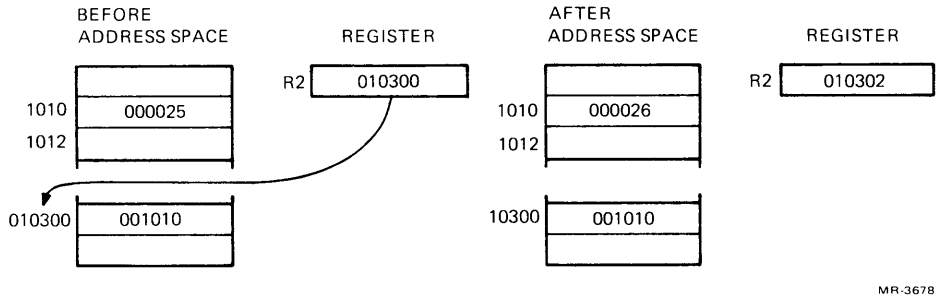


Figure 6-7 Autoincrement Deferred Mode Example

6.3.5 Autodecrement Mode MODE 4 -(Rn)

In autodecrement mode, the register contains an address that is automatically decremented; the decremented address is used to locate an operand. This mode is similar to autoincrement mode, but allows stepping through a list of words or bytes in reverse order. The address is autodecremented by 1 for bytes, by 2 for words.

Autodecrement Mode Example

Symbolic	Instruction Octal Code	Description
INCB -(R0)	105240	The contents of R0 are decremented by 1, then used as the address of the operand. The operand byte is increased by 1.

The example is shown in Figure 6-8.

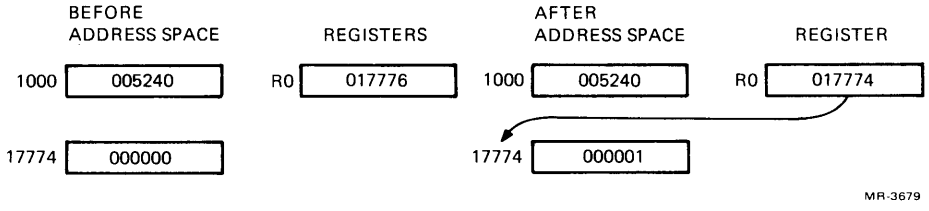


Figure 6-8 Autodecrement Mode Example

6.3.6 Autodecrement Deferred Mode MODE 5 @-(Rn)

In autodecrement deferred mode, the register contains a pointer. The pointer is first decremented by 2, then the new pointer is used to retrieve an address stored outside the CPU. This mode is similar to autoincrement deferred, but allows stepping through a table of addresses in reverse order. Each address then redirects the CPU to an operand. Note that the operands do not have to reside in consecutive locations.

Autodecrement Deferred Mode Example

Symbolic	Instruction Octal Code	Description
COM @-(R0)	005150	The contents of R0 are decremented by 2 and then used as the address of the address of the operand. The operand is 1's complemented.

The example is shown in Figure 6-9.

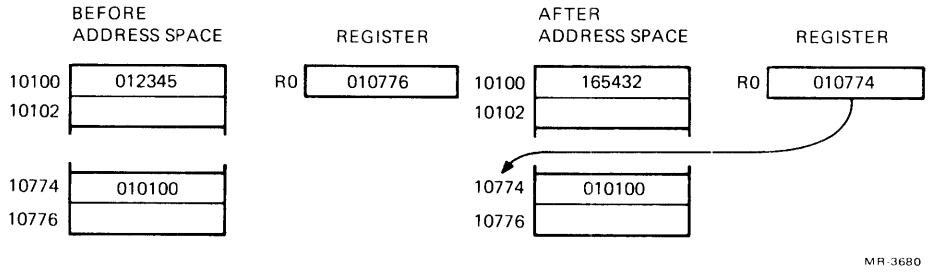


Figure 6-9 Autodecrement Deferred Mode Example

6.3.7 Index Mode MODE 6 ±X(Rn)

In index mode, a base address is added to an index word to produce the effective address of an operand; the base address specifies the starting location of table or list. The index word then represents the address of an entry in the table or list relative to the starting (base) address. The base address may be stored in a register. In this case, the index word follows the current instruction. The locations of the base address and index word may be reversed (index word in the register, base address following the current instruction).

Index Mode Example

Symbolic	Instruction Octal Code	Description
CLR 200(R4)	005064 000200	The address of the operand is determined by adding 200 to the contents of R4. The location is then cleared.

The example is shown in Figure 6-10.

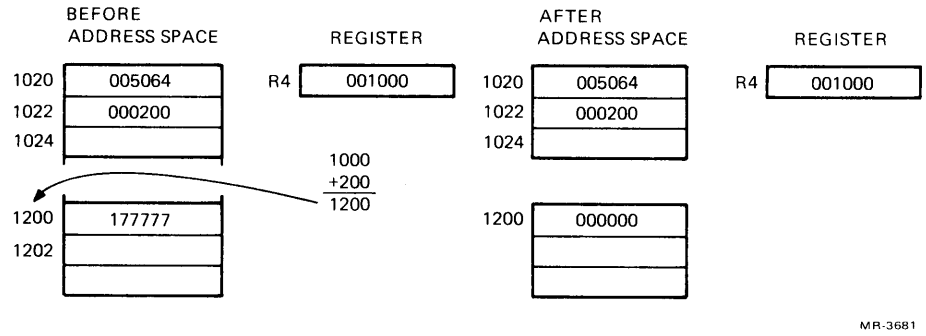


Figure 6-10 Index Mode Example

6.3.8 Index Deferred Mode MODE 7 @X(Rn)

In index deferred mode, a base address is added to an index word. The result is the address of a pointer to the address of the source operand, rather than the address of the source operand. This mode is similar to mode 6, except that it produces a pointer to an address. The content of that address then redirects the CPU to the desired operand. Mode 7 provides for the random access of operands using a table of operand addresses.

Index Deferred Mode Example

Symbolic	Instruction Octal Code	Description
Add @1000(R2), R1	067201 001000	1000 and the contents of R2 are summed to produce the address of the source operand, the contents of which are added to the contents of R1. The result is stored in R1.

The example is shown in Figure 6-11.

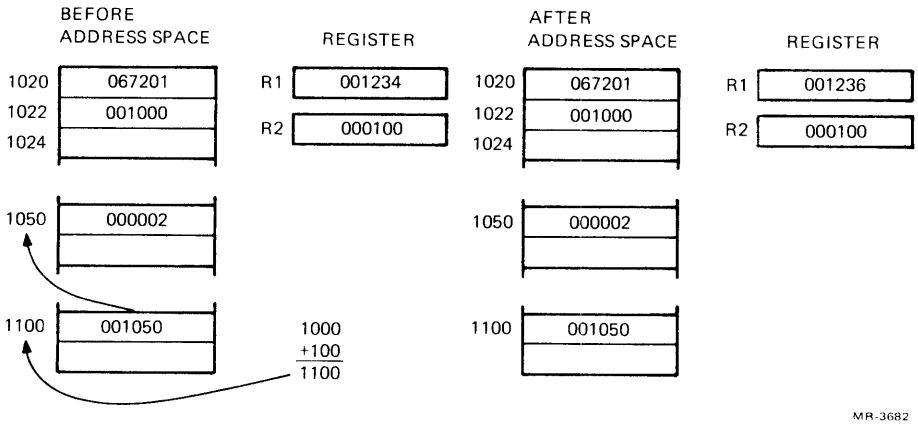


Figure 6-11 Index Deferred Mode Example

6.3.9 Use of the PC as a General Register

Register 7 is both a general-purpose register and the program counter. When the CPU uses the PC to access a word from memory, the PC is automatically incremented by 2 to contain the address of the next word in the instruction being executed or the address of the next instruction to be executed. When the program uses the PC to access byte data, the PC is still incremented by 2.

The PC can be used with all the addressing modes. There are four modes in which the PC can provide advantages for handling position-independent code (see Chapter 10) and unstructured data. These modes are termed immediate, absolute (or immediate deferred), relative, and relative deferred. The remaining modes operate normally when used with the PC. However, they have no practical use in normal programming.

6.3.9.1 PC Immediate Mode MODE 2 #n

Immediate mode is equivalent to using the autoincrement mode with the PC. It provides time improvements for accessing constant operands by including the constant in the memory location immediately following the instruction word.

PC Immediate Mode Example

Symbolic	Instruction Octal Code	Description
ADD #10, R0	062700 000010	The value 10 is located in the second word of the instruction and is added to the contents of R0. Just before this instruction is fetched and executed, the PC points to the

first word of the instruction. The processor fetches the first word and increments the PC by 2. The source operand mode is 27 (autoincrement the PC). Thus, the PC is used as a pointer to fetch the operand (the second word of the instruction) before being incremented by 2 to point to the next instruction.

The example is shown in Figure 6-12.

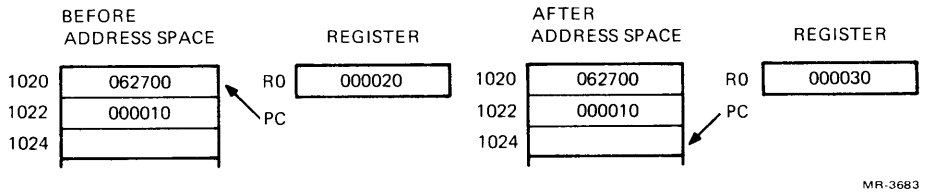


Figure 6-12 PC Immediate Mode Example

6.3.9.2 PC Absolute Mode — MODE 3 @#A

This mode is the equivalent of immediate deferred or autoincrement deferred mode using the PC. The contents of the location following the instruction are taken as the address of the operand. Immediate data is interpreted as an absolute address (i.e., an address that remains constant no matter where in memory the assembled instruction is executed).

PC Absolute Mode Example

Symbolic	Instruction Octal Code	Description
CLR @#1100	005037 001100	Clears the contents of location 1100.

The example is shown in Figure 6-13.

6.3.9.3 PC Relative Mode — MODE 6 A

This mode is index mode 6 using the PC. The operand's address is calculated by adding the word that follows the instruction (called an "offset") to the updated contents of the PC.

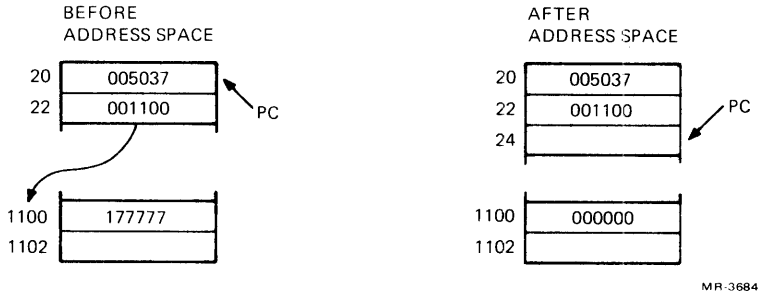


Figure 6-13 PC Absolute Mode Example

PC+2 directs the CPU to the offset that follows the instruction. PC+4 is summed with this offset to produce the effective address of the operand. PC+4 also represents the address of the next instruction in the program.

With the relative addressing mode, the address of the operand is always determined with respect to the updated PC. Therefore, when the instruction is relocated, the operand remains the same relative distance away.

The distance between the updated PC and the operand is called an offset. After a program is assembled, this offset appears in the first word location that follows the instruction. This mode is useful for writing position-independent code (see Chapter 10).

PC Relative Mode Example

Symbolic	Instruction Octal Code	Description
INC A	005267 000054	To increment location A, contents of memory location in the second word of the instruction are added to PC to produce address A. Contents of A are increased by 1.

The example is shown in Figure 6-14.

6.3.9.4 PC Relative Deferred Mode MODE 7 @A

This mode is index deferred (mode 7), using the PC. A pointer to an operand's address is calculated by adding an offset (that follows the instruction) to the updated PC.

This mode is similar to the relative mode, except that it involves one additional level of addressing to obtain the operand. The sum of the offset and updated PC (PC+4) serves as a pointer to an address. When the address is retrieved, it can be used to locate the operand.

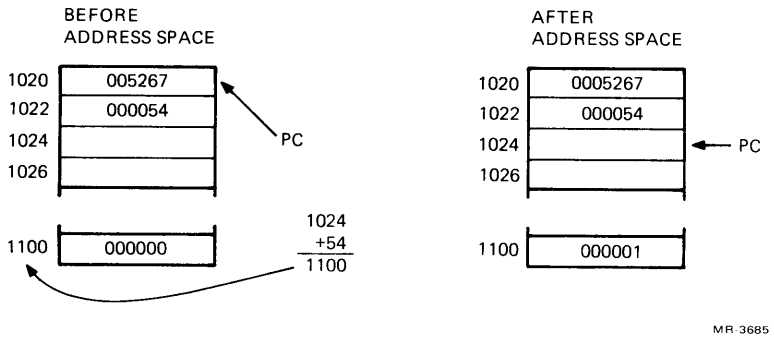


Figure 6-14 PC Relative Mode Example

PC Relative Deferred Mode Example

Symbolic	Instruction Octal Code	Description
CLR @A	005077 000020	Adds the second word of the instruction to PC to produce the address of the address of the operand. Clears operand.

The example is shown in Figure 6-15.

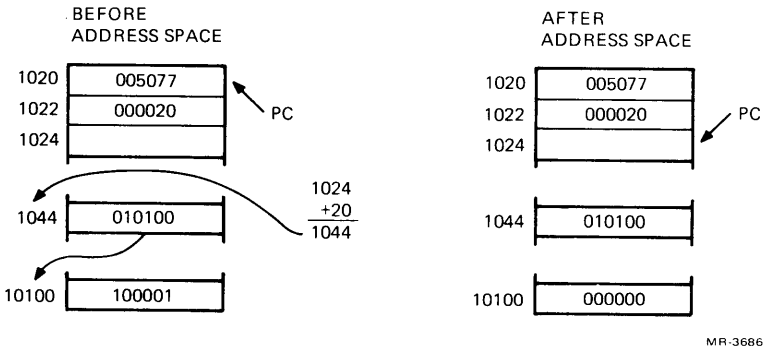


Figure 6-15 PC Relative Deferred Mode Example

6.3.10 Direct Addressing Modes Summary

Table 6-1 summarizes the four basic modes used with direct addressing.

6.3.11 Indirect Addressing Modes Summary

Table 6-2 summarizes the same four basic modes used with indirect addressing.

Table 6-1 Direct Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
000	0	Register	Rn	Register contains operand.
010	2	Autoincrement	(Rn)+	Register is used as a pointer to sequential data, then incremented.
100	4	Autodecrement	-(Rn)	Register is decremented and then used as a pointer to sequential data.
110	6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) is modified.

Table 6-2 Indirect Addressing Modes

Binary Code Code	Mode	Name	Symbolic	Function
001	1	Register Deferred	@Rn or (Rn)	Register contains the address of the operand.
011	3	Autoincrement Deferred	@(Rn)+	Register is first used as a pointer to a word containing the address of the operand, then incremented (always by 2, even for byte instructions).
101	5	Autodecrement Deferred	@-(Rn)	Register is decremented (always by 2, even for byte instructions) and then used as a pointer to a word containing the address of the operand.
111	7	Index Deferred	@X(Rn)	Value X (located in a word contained in the instruction) and (Rn) are added and the sum is used as a pointer to a word containing the address of the operand. Neither X nor (Rn) is modified.

6.3.12 PC Register Addressing Modes Summary

When used with the PC, these modes are termed immediate, absolute (or immediate deferred), relative, and relative deferred. They are summarized in Table 6-3.

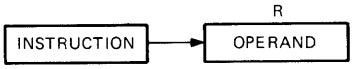
Table 6-3 PC Register Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
010	2	Immediate	#n	Operand is contained in the instruction.
011	3	Absolute	@#A	Absolute address is contained in the instruction.
110	6	Relative	A	Address of A, relative to the instruction, is contained in the instruction.
111	7	Relative Deferred	@A	Address of location containing address of A, relative to the instruction, is contained in the instruction.

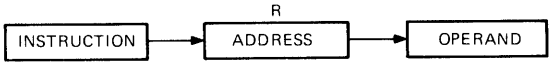
6.3.13 Graphic Summary of Addressing Modes

Figures 6-16 and 6-17 provide a graphic summary of general register addressing modes and program counter addressing modes.

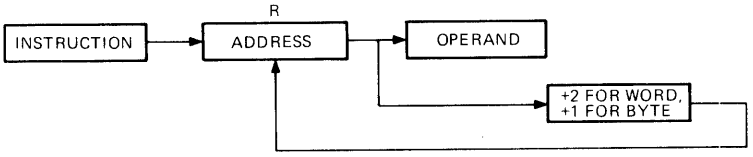
Mode 0	**Register**	OPR R	R contains operand.



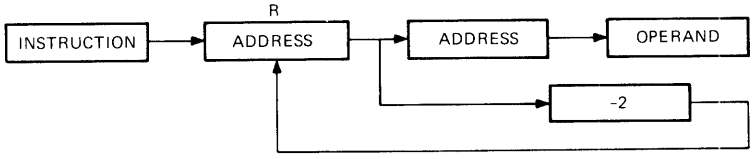
Mode 1	**Register deferred**	OPR (R)	R contains address.



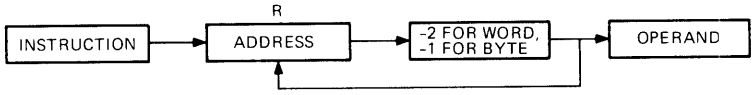
Mode 2	**Autoincrement**	OPR (R)+	R contains address, then increment (R).



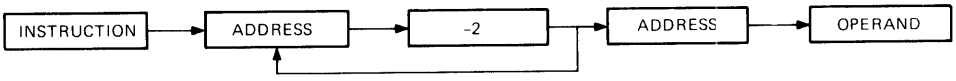
Mode 3	**Autoincrement deferred**	OPR @(R)+	R contains address of address, then increment (R) by 2.



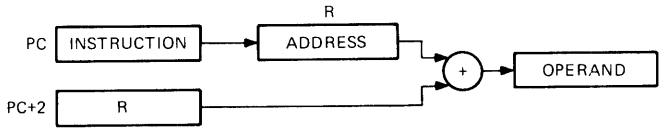
Mode 4	**Autodecrement**	OPR -(R)	Decrement (R), then R contains address.



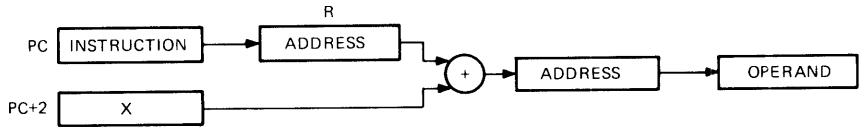
Mode 5	**Autodecrement deferred**	OPR @- (R)	Decrement (R) by 2, then R contains address of address.



Mode 6	**Index**	OPR X(R)	(R)+X is address, second word of instruction.



Mode 7	**Index deferred**	OPR @X(R)	(R)+X is address (second word) of address.



R is a general register, 0 to 7.
(R) is the contents of that register.

MR-3687

Figure 6-16 General Register Addressing Modes

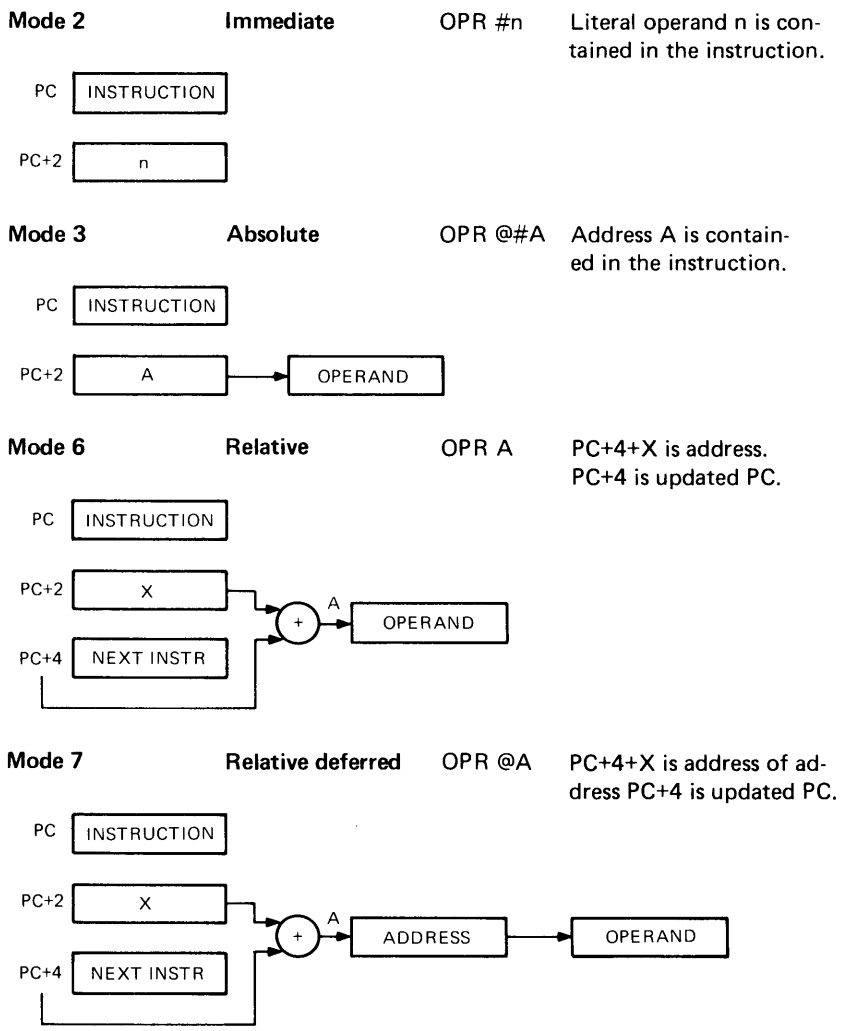


Figure 6-17 Program Counter Addressing Modes

CHAPTER 7
INSTRUCTION SET

7.1 INTRODUCTION

The KDF11-AA instruction set and addressing modes produce over 400 unique instructions. The instruction set offers a wide choice of operations, so that a single instruction will frequently accomplish a task that would require several instructions in a traditional computer. KDF11-AA instructions allow byte and word addressing in both single and double operand formats. This saves memory space and simplifies the implementation of control and communications applications. The use of double operand instructions makes it possible to perform several operations with a single instruction. For example, ADD A,B adds the contents of location A to location B and stores the result in location B. Traditional computers would implement this instruction in the following way.

```
LDA A
ADD B
STR B
```

The instruction set contains a full set of conditional branches, eliminating excessive use of jump instructions. All instructions fall into one of three categories.

1. Single Operand - One part of the word specifies the operation, referred to as "op code," and the second part provides information for locating the operand.

2. Double Operand - The first part of the word specifies the operation to be performed; the remaining two parts provide information for locating two operands.

3. Program Control - The first part of the word specifies the operation to be performed; the second part indicates where the action is to take place in the program.

7.1.1 Single Operand Instructions

The following is a list of single operand instructions.

General

Mnemonic	Instruction
CLR(B)	clear destination
COM(B)	1's complement destination
INC(B)	increment destination
DEC(B)	decrement destination
NEG(B)	2's complement negate destination
TST(B)	test destination

Shift and Rotate

Mnemonic	Instruction
ASR(B)	arithmetic shift right
ASL(B)	arithmetic shift left
ROR(B)	rotate right
ROL(B)	rotate left
SWAB	swap bytes

Multiple Precision

Mnemonic	Instruction
ADC(B)	add carry
SBC(B)	subtract carry
SXT	sign extend

Processor Status

Mnemonic	Instruction
MFPS	move byte from processor status
MTPS	move byte to processor status

Instruction Format - The instruction format for single operand instructions, as shown in Figure 7-1, is described as follows.

1. Bits 15-6 indicate the operation code, which specifies the operation to be performed. (Bit 15 indicates word or byte operation.)

2. Bits 5-0 indicate the destination address, which provides information for locating the operand.

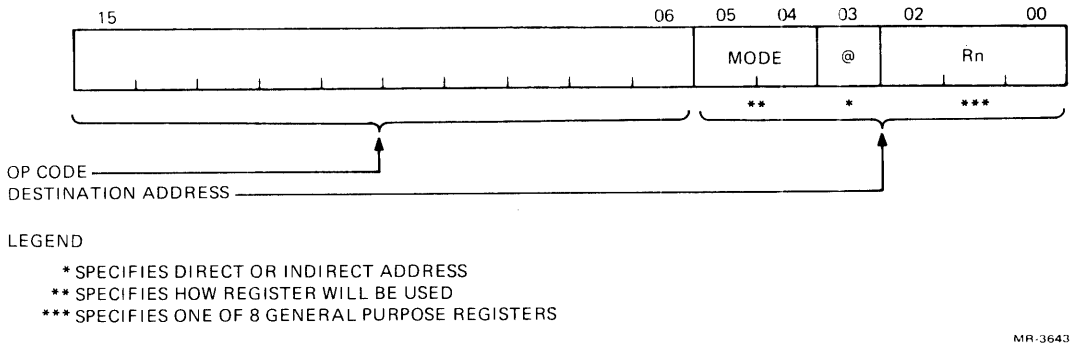


Figure 7-1 Single Operand Instruction Format

7.1.2 Double Operand Instructions

The following is a list of double operand instructions.

General

Mnemonic	Instruction
MOV(B)	move source to destination
ADD	add source to destination
SUB	subtract source from destination
ASH	shift arithmetically
ASHC	arithmetic shift combined
MUL	integer multiply
DIV	integer divide

Logical

Mnemonic	Instruction
BIT(B)	bit test
BIC(B)	bit clear
BIS(B)	bit set
XOR	exclusive OR

7.1.2.1 Double Operand Instruction Format - The format of most double operand instructions (see Figure 7-2) is similar to that of single operand instructions except that they have <u>two</u> fields for locating operands. One field is called the source field, the other is called the destination field. Each field is further divided into addressing mode and selected register. Each field is completely independent. The mode and register used by one field may be completely different than the mode and register used by another field.

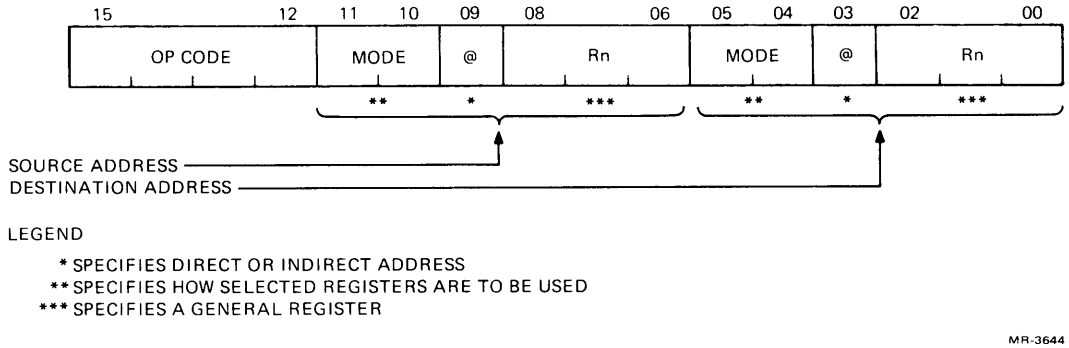


Figure 7-2 Double Operand Instruction Format

Bit 15 indicates word or byte operation <u>except</u> when used with op code 6. Then it indicates an ADD or SUBtract instruction.

Bits 14-12 indicate the op code, which specifies the operation to be done.

Bits 11-6 indicate the source address, which contains information for locating the source operand.

Bits 5-0 indicate the destination address, which contains information for locating the source operand.

7.1.2.2 **Byte Instructions** - Byte instructions are specified by setting bit 15. Thus, in the case of the MOV instruction, bit 15 is 0; when bit 15 is set, the mnemonic is MOVB. There are no byte operations for ADD and SUB; i.e., no ADDB or SUBB. In order to perform the equivalent of an ADDB or SUBB, the MOVB instruction can be used along with an ADD or SUB. The MOVB instruction, when the destination address mode is 0, sign-extends the byte operand through the high byte of the register. This feature can be used by executing a MOVB to get the first byte operand and place it in one general register, and another MOVB to get the second byte operand and place it in a second general register. Then an ADD or SUB is performed on both general registers.

```
Example:  MOVB A,R0
          MOVB B,R1
          ADD R0,R1
```

The condition codes will be affected based upon the byte result.

7.1.3 Program Control Instructions

This paragraph discusses program control instructions.

7.1.3.1 **Branch Instructions** - The following is a list of branch instructions and a discussion of the branch instruction format.

Branch

Mnemonic	Instruction
BR	branch (unconditional)
BNE	branch if not equal to 0
BEQ	branch if equal to 0
BPL	branch if plus
BMI	branch if minus
BVC	branch if overflow is clear
BVS	branch if overflow is set
BCC	branch if carry is clear
BCS	branch if carry is set

Signed Conditional Branch

Mnemonic	Instruction
BGE	branch if greater than or equal to 0
BLT	branch if less than 0
BGT	branch if greater than 0
BLE	branch if less than or equal to 0
SOB	subtract one and branch if not equal to 0

Unsigned Conditional Branch

Mnemonic	Instruction
BHI	branch if higher
BLOS	branch if lower or same
BHIS	branch if higher or same
BLO	branch if lower

Branch Instruction Format (Figure 7-3)
The high byte (bits 8-15) of the instruction is an op code specifying the conditions for the branch to take place.

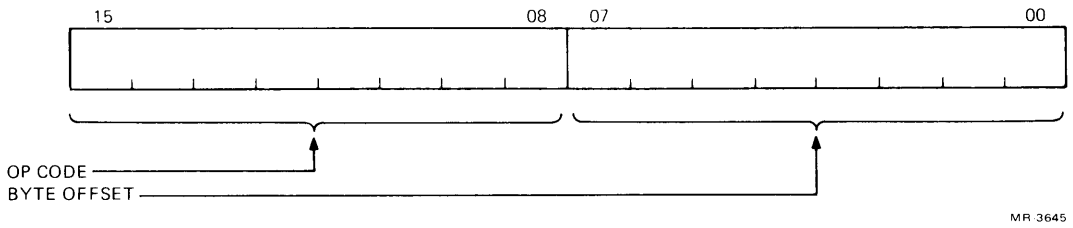


Figure 7-3 Branch Instruction Format

The low byte (bits 0-7) of the instruction is the offset value in words that determines the new program location if the branch is taken. The low byte is treated as an 8-bit signed integer and since the CPU is byte-organized, the integer must be converted from words to bytes. This is done during execution by sign-extending the low byte and then shifting the 16-bit word left one position to create the offset in bytes. Then the offset is added to the current value of the PC to form the new program location if the branch is taken. Since the PC is always incremented by two bytes immediately after the instruction is fetched, the current value of the PC, when the new program location is formed, points to the next location after the branch. Hence an unconditional branch to its own location is 000777_8, rather than 00040_8, which is a branch to the next location.

7.1.3.2 **Jump and Subroutine Instructions** - The following is a list of jump and subroutine instructions, and a discussion of their formats. A list of related interrupt and trap instructions is also provided along with a list of ways to exit from a main program.

Jump and Subroutine

Mnemonic	Instruction
JMP	jump
JSR	jump to subroutine
RTS	return from subroutine

JSR Instruction Format (Figure 7-4)
Bits 9-15 are always octal 004 indicating the op code for JSR.

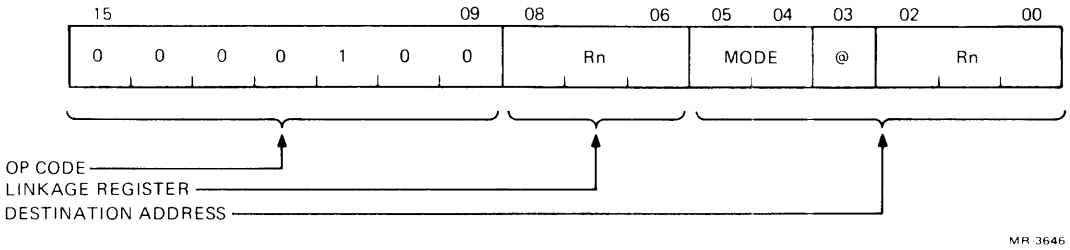


Figure 7-4 JSR Instruction Format

Bits 6-8 specify the link register. Any general purpose register may be used in the link, except R6.

Bits 0-5 designate the destination address that consists of addressing mode and general register fields. This specifies the starting address of the subrroutine.

Register R7 (program counter) is frequently used for both the link and the destination. For example, JSR R7, SUBR, which is coded 004767 may be used. R7 is the only register that can be used for both the link and destination, the other general-purpose registers (GPRs) cannot. Thus, if the link is R5, any register except R5 can be used in the destination field.

RTS Instruction Format (Figure 7-5)
The RTS (return from subroutine) instruction uses the link to return control to the main program once the subroutine is finished.

Bits 3-15 always contain octal 00020, which is the op code for RTS.

Bits 0-2 specify any one of the general-purpose registers.

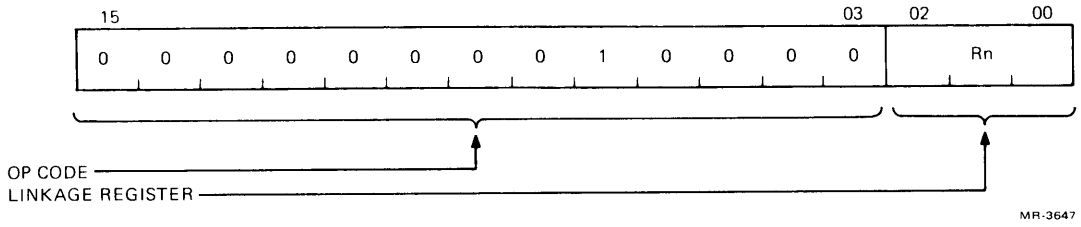


Figure 7-5 RTS Instruction Format

The register specified by bits 0-2 must be the same register as the one used in the JSR which called the subroutine.

Interrupts and Traps

Mnemonic	Instruction
EMT	emulator trap
TRAP	trap
BPT	breakpoint trap
IOT	input/output trap
RTI	return from interrupt
RTT	return from trace trap

Exiting from a Main Program
There are three ways of leaving a main program.

1. Software Exit - The program specifies a jump to some subroutine.

2. Trap Exit - Internal processor hardware executes certain instructions (e.g., EMT) which cause a jump to special software routines.

3. Interrupt Exit - External hardware forces a jump to an interrupt service routine.

In all of the above cases, there is a jump to another program. Once that program has been executed, control is returned to the proper point in the main program.

7.1.3.3 Condition Code Instructions - The following is a list of instructions that affect the condition codes in the PS, and their format. How the condition codes are affected is also discussed.

Mnemonic	Instruction
CLC,CLV,CLZ,CLN,CCC	clear selected condition code
SEC,SEV,SEZ,SEN,SCC	set selected condition code

Instruction Format

The format of the condition code operators, shown in Figure 7-6, is as follows.

1. Bits 15-5 - The operation code.

2. Bit 4 - The "operator" which indicates set or clear with the values 1 and 0 respectively. If set, any selected bit is set; if clear, any selected bit is cleared.

3. Bits 3-0 - The "select" field. Each of these bits corresponds to one of the four condition code bits. When one of these bits is set, then the corresponding condition code bit is set or cleared depending on the state of the "operator" (bit 4).

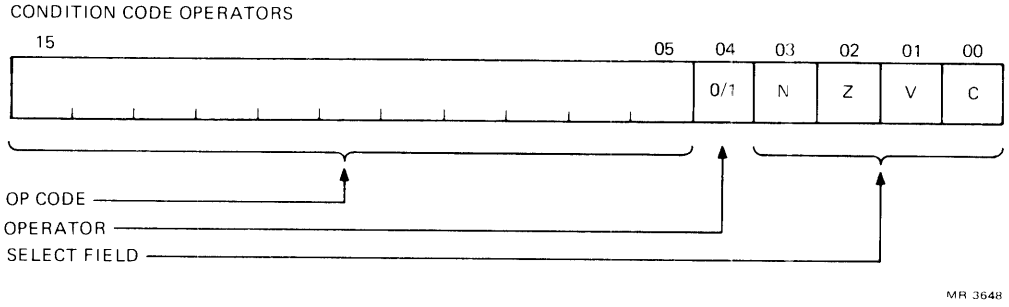


Figure 7-6 Condition Code Operators Format

More than one condition code can be set by a particular instruction. For example, both a carry and an overflow condition may exist after instruction execution.

Condition Codes

There are four condition code bits.

1. N indicates a negative condition when set to 1.
2. Z indicates a zero condition when set to 1.
3. V indicates an overflow condition when set to 1.
4. C indicates a carry condition when set to 1.

These four bits are part of the processor status word (PS). The result of any single operand or double operand instruction affects one or more of the four condition code bits. A new set of condition codes is usually created after execution of each instruction. Some condition codes are not affected by the execution of certain instructions. Branch instructions may test the condition codes after execution of single or double operand instruction. The condition codes are used by the various instructions to check software conditions.

N Bit
The CPU looks only at the sign bit of the result. If the sign bit is set, indicating a negative value, the CPU sets the N bit. If the sign bit is clear, indicating a positive value, then the CPU clears the N bit. When an overflow occurs (V bit is set), the N bit does not indicate the true sign of the result since the N bit is equal to bit 15 of the result.

Z Bit
Whenever the CPU sees that the result of an instruction is 0, it sets the Z bit. If the result is not 0, it clears the Z bit. There are a number of ways of obtaining a 0 result.

1. Adding two numbers equal in magnitude but different in sign

2. Comparing two numbers of equal value

3. Using the CLR instruction.

V Bit
The V bit is set to indicate that an overflow condition exists. An overflow means that the result of an instruction is too large to be represented in 2's complement format. There are two methods the hardware used to check for an overflow condition.

One way is for the CPU to test for a change of sign.

1. When using single operand instructions, such as INC, DEC, or NEG, a change of sign indicates an overflow condition.

2. When using double operand instructions, such as ADD, SUB, or CMP, in which both the source and destination have like signs, a change of sign in the result indicates an overflow condition.

Another method used by the CPU is to test the N bit and C bit when dealing with shift and rotate instructions.

1. If only the N bit is set, an overflow exits.

2. If only the C bit is set, an overflow exists.

3. If both the N and C bits are set, there is no overflow condition.

C Bit
The CPU sets the C bit automatically when the result of an instruction has caused a carry-out of the most significant bit of the result. When the instruction results in a carry-out of the most significant bit of the result, the carry itself is usually moved into the C bit. Othewise, the C bit is cleared. During

rotate instructions (ROL and ROR), the C bit forms a buffer between the most significant bit and the least significant bit of the word. A carry of 1 sets the C bit while a carry of 0 clears the C bit. However, there are exceptions.

1. SUB and CMP set the C bit when there is no carry to indicate that a borrow occurred.

2. Logical operations (e.g., BIT) do not affect the C bit since they are not arithmetic in nature.

3. COM always sets the C bit, TST always clears the C bit.

7.1.3.4 Miscellaneous Instructions - The following is a list of miscellaneous program control instructions.

Mnemonic	Instruction
HALT	halt
WAIT	wait for interrupt
RESET	reset I/O
MTPD	move to previous data space
MTPI	move to previous instruction space
MFPD	move from previous data space
MFPI	move from previous instruction space
MTPS	move byte to processor status word
MFPS	move byte from processor status word

7.1.4 Examples

The following examples and explanations illustrate the use of the various types of instructions in a program.

7.1.4.1 Single Operand Instruction Example - This routine uses a tally to control a loop, which clears out a specific block of memory. The routine has been set up to clear 30_8 byte locations beginning at memory address 600.

```
(R0) = 600
(R1) = 30

LOOP:    CLRB(R0)+
         DEC R1
         BNE LOOP
         HALT
```

Program Description

The CLRB (R0)+ instruction clears the contents of the location specified by R0.

R0 is the pointer.

Because the autoincrement addressing mode is used, the pointer automatically moves to the next memory location after execution of the CLRB instruction.

Register R1 indicates the number of locations to be cleared and is, therefore, a counter. Counting is performed by the DIGITAL R1 instruction. Each time a location is cleared, it is counted by decrementing R1.

The Branch If Not Zero, BNE, instruction checks for done. If the counter is not 0, the program branches back to start to clear another location. If the counter is 0, indicating done, then the program executes the next instruction, HALT.

7.1.4.2 Double Operand Instruction Example - This routine prints out a portion of a payroll program for review by the supervisor. It is known that 76 locations are to be printed and the locations start at address 600.

```
INIT:     MOV #600,R0
          MOV #76,R1

START:    TSTB I/O
          BPL START
          MOVB (R0)+,I/O+2
          DEC R1
          BNE START
          HALT
```

Program Description
MOV is the instruction normally used to set up the initial conditions. Here, the first MOV places the starting address (600) into R0, which will be used as a pointer. The second MOV sets up R1 as a counter by loading the desired number of locations (76) to be printed.

The TSTB instruction tests the Done or Ready flag (bit 7) of the printer. The BPL instruction causes a loop to start if the state of the printer-ready flag is cleared.

The MOVB instruction moves a byte of data to the printer (I/O) for printing. The data comes from the location specified by R0. The pointer R0 is then incremented to point to the next sequential location.

The counter (R1) is then decremented to indicate one byte has been transferred.

The program then checks the loop for done with the BNE instruction. If the counter has not reached 0, this indicates that more transfers must take place. The BNE causes a branch back to START and the program continues.

When the counter (R1) reaches 0, indicating all data has been transferred, the branch does not occur and the program executes the next instruction, HALT.

7.1.4.3 Branch Instruction Example

NOTE
Branch instruction offsets are limited from $+177_8$ to -200_8 words.

A payroll program has set up a series of words to identify each employee by his badge number. The high byte of the word contains the employee's badge number; the low byte contains an octal number ranging from 0 to 13 which represents his salary. These numbers represent steps within three wage classes to identify which employees get paid weekly, monthly, or quarterly. It is time to make out weekly paychecks. Unfortunately, employee information has been stored in a random order. The problem is to extract the names of only those employees who receive a weekly paycheck. Employee payroll numbers are assigned as follows: 0 to 3 - wage class I (weekly), 4 to 7 - wage class II (monthly), 10 to 13 - wage class III (quarterly).

600 is the starting address of memory block containing the employee payroll information. 1264 is the final address of this data area. The following program searches through the data area and finds all numbers representing wage class I, and, each time an appropriate number is found, stores the employee's badge number (just the high byte) on a "last-in/first-out" stack which begins at location 4000.

```
INIT:       MOV #600, R0
            MOV #400, R1

START:      CMPB(R0)+,#3
            BHI CONT

STACK:      MOVB (R0),-(R1)

CONT:       INC R0
            CMP # 1264, R0
            BHIS START
            HALT
```

Program Description
R0 becomes the address pointer, R1 the stack pointer.

Compare the contents of the first low byte with the number 3 and go to the first high byte. If the number is more than 3, branch to continue. If no branch occurs, it indicates that the number is 3 or less. Therefore, move the high byte containing the employee's number onto the stack as indicated by stack pointer R1.

R0 is advanced to the next low byte.

If the last address has not been examined (1264), this instruction produces a result equal to or greater than zero. If the result is equal to or greater than zero, examine the next memory location.

7.2 INSTRUCTION SET

The KDF11-AA instruction set is described in this paragraph. For ease of reference, the instructions are presented alphabetically.

A number of special symbols are used to describe certain features of individual instructions. The commonly used symbols are explained in Table 7-1.

Table 7-1 Instruction Symbols

Symbol	Meaning
SO	Single operand instruction
DO	Double operand instruction
PC	Program control instruction
MS	Miscellaneous instruction
CC	Condition code
()	Indicates the contents of. For example, (R5) means "the contents of R5."
src	Source address
dst	Destination address
<-	Becomes, or moves into. For example, (dst) <- (src) means that the source becomes the destination or that the source moves into the destination location.
(SP)+	Popped or removed from the hardware stack
-(SP)	Pushed or added to the hardware stack
^	Logical AND
v	Logical inclusive OR (either one or both)
⩛	Logical exclusive OR (either one, but not both)
~	Logical NOT
Reg or R	Register
B	Byte

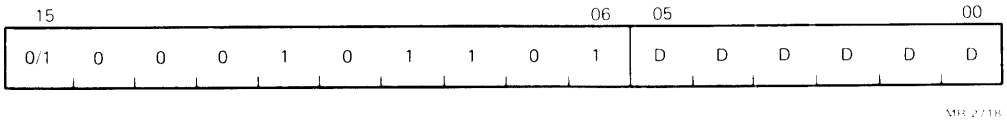
NOTE

Condition code bits are considered to be cleared unless they are specifically listed as set.

ADC/ADCB

Add Carry

0055DD
1055DD



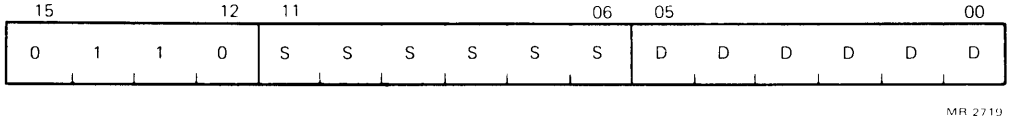
Type: SO

Operation: (dst) <-- (dst) + C

Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if (dst) is 077777 and C = 1
C: set if (dst) is 177777 and C = 1

Description: Adds the contents of the C bit into the destination. This permits the carry from the addition of the low-order words/bytes to be carried into the high-order result, such as in performing double precision arithmetic.

Add 06SSDD



Type: DO

Operation: (dst) <-- (src) + (dst)

Condition Codes:

N: set if result < 0

Z: set if result = 0

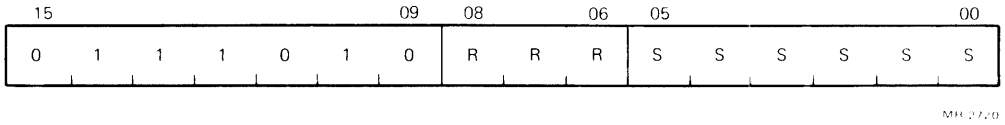
V: set if there is arithmetic overflow as a result of the operation; that is, both operands were of the same sign and the result is of the opposite sign

C: set if there is a carry from the most significant bit of the result

Description: Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. 2's complement addition is performed.

ASH

Arithmetic Shift 072RSS



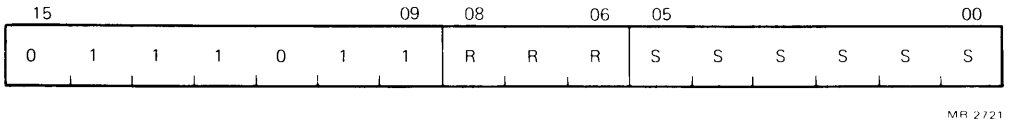
Type: DO

Operation: R <-- R shifted arithmetically NN places to the right or left where NN = (src)

Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if sign of register changed during shift
C: loaded from last bit shifted out of register

Description: The contents of the register are shifted right or left the number of times specified by the source operand. The shift count is taken as the low-order 6 bits of the source operand. This number ranges from -32 to +31. Negative is a right shift and positive is a left shift.

Arithmetic Shift Combined 073RSS



Type: DO

Operation: R, Rvl <-- R, Rvl
The double word is shifted NN places to the right or left, where NN = (src).

Condition Codes: N: set if result < 0

Z: set if result = 0

V: set if sign bit changes during the shift

C: loaded with high-order bit when left; loaded with low-order bit when right shift (loaded with the last bit shifted out of the 32-bit operand)

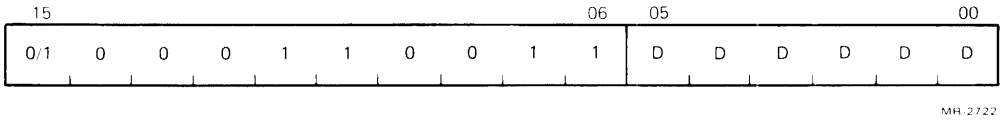
Description: The contents of the register and the register OR-ed with 1 are treated as one 32-bit word. Rvl (bits 0-15) and R (bits 16-31) are shifted right or left the number of times specified by the shift count. The shift count is taken as the low-order six bits of the source operand. This number ranges from -32 to +31. Negative is a right shift and positive is a left shift.

When the register chosen is an odd number, the register and the register OR-ed with 1 are the same. In this case, the right shift becomes a rotate. The 16-bit word is rotated right the number of bits specified by the shift count.

ASL/ASLB

Arithmetic Shift Left

0063DD
1063DD



Type: SO

Operation: (dst) <-- (dst) shifted one place to the left

Condition Codes:

N: set if high-order bit of the result < 0

Z: set if the result = 0

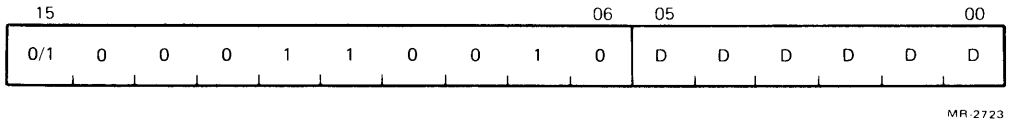
V: loaded with the exclusive OR of the N bit and C bit (as set by the completion of the shift operation)

C: loaded with the high-order bit of the destination

Description: Shifts all bits of the destination left one place. The low-order bit is loaded with a 0. The C bit of the status word is loaded from the high-order bit of the destination. ASL performs a signed multiplication of the destination by 2 with overflow indication.

Arithmetic Shift Right

0062DD
1062DD



Type: SO

Operation: (dst) <-- (dst) shifted one place to the right

Condition Codes:

N: set if the high-order bit of the result is set (result < 0)

Z: set if the result = 0

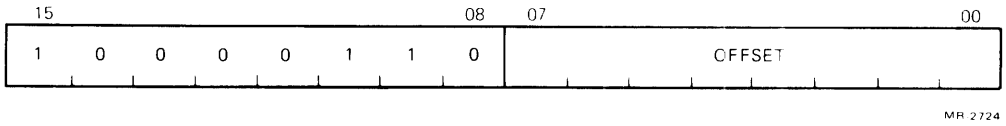
V: loaded from the exclusive OR of the N bit and C bit (as set by the completion of the shift operation)

C: loaded from low-order bit of the destination

Description: Shifts all bits of the destination right one place. The high-order bit is replicated. The C bit is loaded from the low-order bit of the destination. ASR performs signed division by 2.

BCC

Branch if carry clear 103000



Type: PC

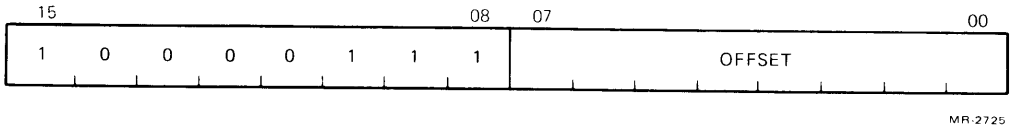
Operation: PC <-- PC + (2 X offset) if C = 0

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the C bit and causes a branch if C is clear.

Branch if carry set

103400



Type: PC

Operation: PC <-- PC + (2 X offset) if C = 1

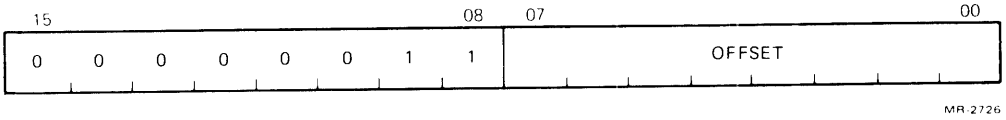
Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the C bit and causes a branch if C is set. Used to test for a carry in the result of a previous operation.

BEQ

Branch if equal

001400



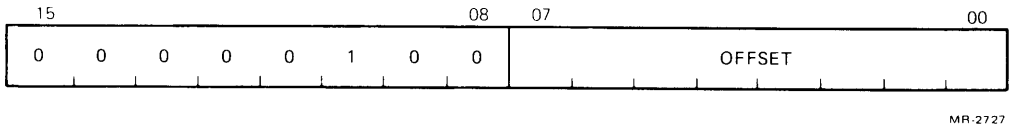
Type: PC

Operation: PC <-- PC + (2 X offset) if Z = 1

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the Z bit and causes a branch if Z is set. As an example, it is used to test equality following a CMP operation, to test that no bits set in the destination were also set in the source following a BIT operation, and, generally, to test that the result of the previous operation was 0.

Branch if greater than or equal 002000



Type: PC

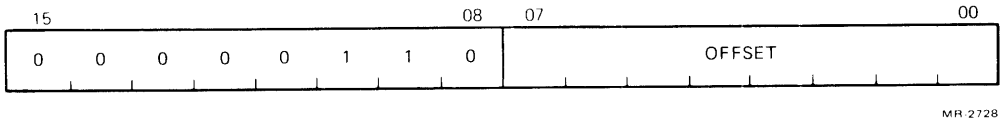
Operation: PC <-- PC + (2 X offset) if N⊻V = 0

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes a branch if N and V are either both clear or both set. BGE is the complementary operation to BLT. Thus, BGE always causes a branch when it follows an operation that caused addition of two positive numbers. BGE also causes a branch on a 0 result.

BGT

Branch if greater than 003000



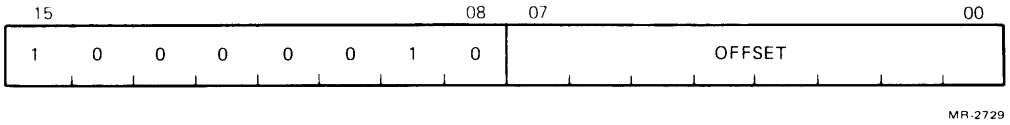
Type: PC

Operation: PC <-- PC + (2 X offset) if Zv (N ∀ V) = 0

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes a branch if the exclusive OR of the N and V bits is 1. Thus, BGT always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BGT always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BGT never causes a branch when it follows a CMP instruction operating on a positive source and negative destination. BGT does not cause a branch if the result of the previous operation was 0 (without overflow).

Branch if higher 101000



Type: PC

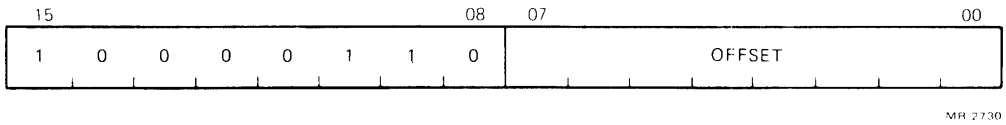
Operation: PC <-- PC + (2 X offset) if C = 0 and Z = 0

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes a branch if the previous operation causes neither a carry nor a 0 result. This will happen in comparison (CMP) operations as long as the source has a higher unsigned value than the destination.

BHIS

Branch if higher than the same 103000



Type: PC

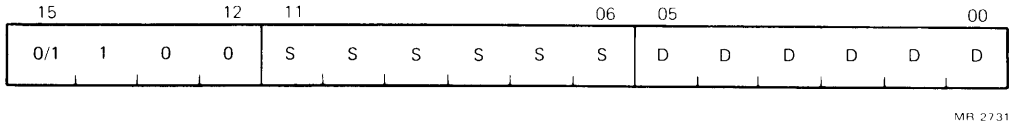
Operation: PC <-- PC + (2 X offset) if C = 0

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the C bit and causes a branch if C is cleared.

Bit Clear

04SSDD
14SSDD



Type: DO

Operation: (dst) <-- ~ (src) ^ (dst)

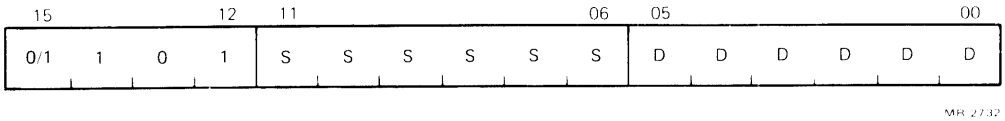
Condition Codes:
N: set if high-order bit of result set
Z: set if result = 0
V: cleared
C: not cleared

Description: Clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are unaffected.

BIS/BISB

Bit Set

05SSDD
15SSDD



Type: DO

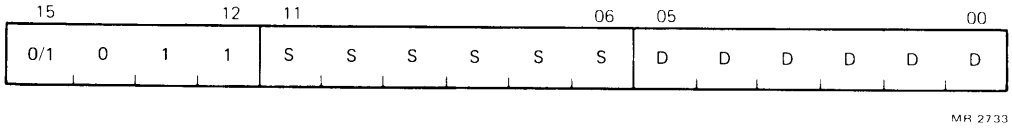
Operation: (dst) <-- (src) v(dst)

Condition Codes:
N: set if high order bit of result set
Z: set if result = 0
V: cleared
C: not affected

Description: Performs inclusive OR operation between the source and destination operands and leaves the result at the destination address; i.e., corresponding bits set in the source are set in the destination. The original contents of the destination are lost.

Bit Test

03SSDD
13SSDD



Type: DO

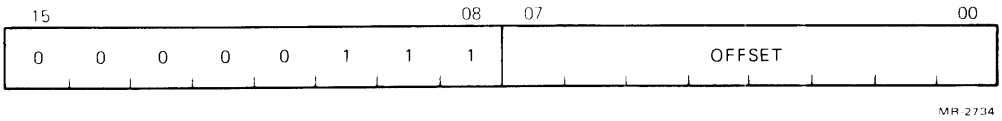
Operation: (dst) ^ (src)

Condition Codes:
N: set if high-order bit of result set
Z: set if result = 0
V: cleared
C: not affected

Description: Performs logical AND comparison of the source and destination operands and modifies condition codes accordingly. Neither the source nor destination operands and modifies condition codes accordingly. Neither the source nor destination operands are affected. The BIT instruction may be used to test whether any of the corresponding bits that are set in the destination are clear in the source.

BLE

Branch if less than or equal to 003400



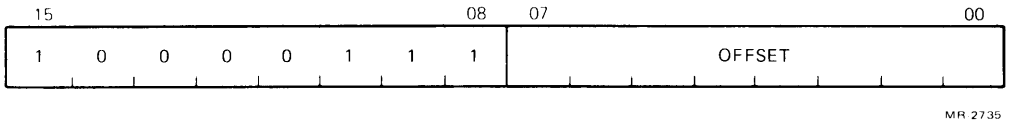
Type: PC

Operation: PC <-- PC + (2 X offset) if Zv(N⊻V) = 1

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes a branch if the exclusive OR of the N and V bits is 1. Thus, BLE always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BLE always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLE never causes a branch when it follows a CMP instruction operating on a positive source and negative destination. BLE does not cause a branch if the result of the previous operation was 0 (without overflow).

Branch if lower 103400



Type: PC

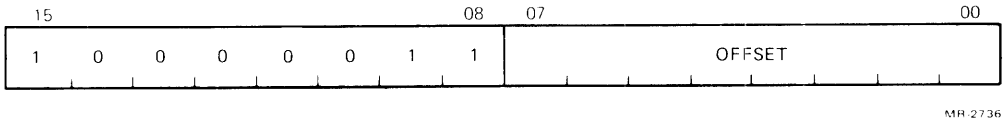
Operation: PC <-- PC + (2 X offset) if C = 1

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the C bit and causes a branch if C is set. Used to test for a carry in the result of a previous operation.

BLOS

Branch if lower or same 101400



Type: PC

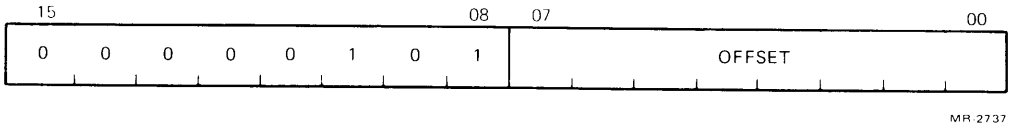
Operation: PC <-- PC + (2 X offset) if CvZ = 1

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes a branch if the previous operation caused either a carry or a 0 result. BLOS is the complementary operation to BHI. The branch occurs in comparison operations as long as the source is equal to or has a lower unsigned value than the destination.

Branch if less than

002400



Type: PC

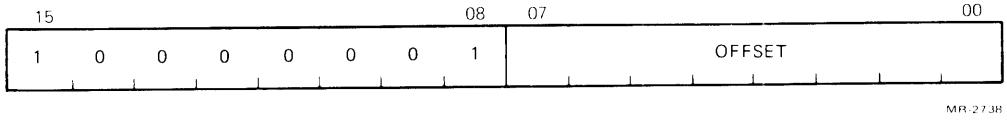
Operation: PC <-- PC + (2 X offset) if N⊻V = 1

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes a branch if the exclusive OR of the N and V bits is 1. Thus, BLT always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BLT always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLT never causes a branch when it follows a CMP instruction operating on a positive source and negative destination. BLT does not cause a branch if the result of the previous operation was 0 (without overflow).

BMI

Branch if minus 100400



Type: PC

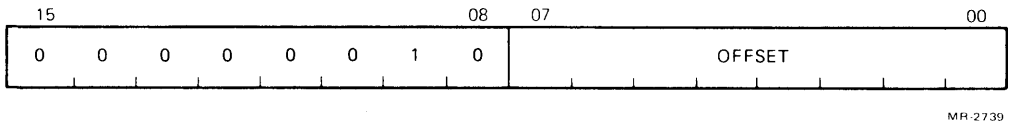
Operation: PC <-- PC + (2 X offset) if N = 1

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the N bit and causes a branch if N is set. Used to test the sign (most significant bit) of the result of the previous operation.

BNE

Branch if not equal 001000



Type: PC

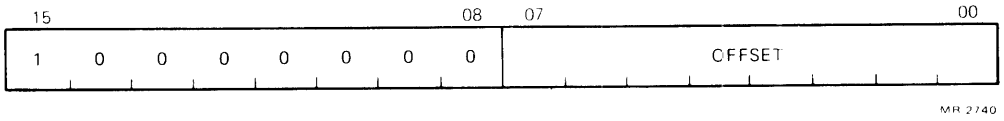
Operation: PC <-- PC + (2 X offset) if Z = 0

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the Z bit and causes a branch if the Z bit is clear. BNE is the complementary operation to BEQ. It is used to test inequality following a CMP, to test that some bits set in the destination were also in the source, following a bit, and, generally, to test that the result of the previous operation was not 0.

BPL

Branch if plus 100000



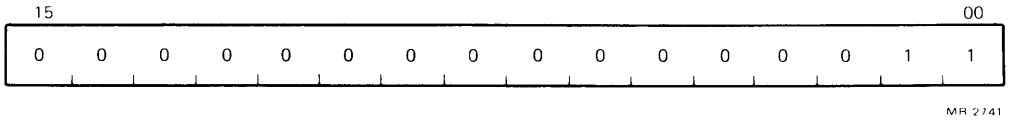
Type: PC

Operation: PC <-- PC + (2 X offset) if N = 0

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the N bit and causes a branch if N is clear. BPL is the complementary operation of BMI.

Breakpoint Trap 000003



Type: PC

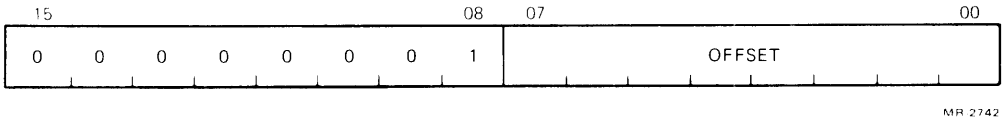
Operation:
- (SP) <-- PS
- (SP) <-- PC
PC <-- (14)
PS <-- (16)

Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description: Performs a trap sequence with a trap vector address of 14. Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids. No information is transmitted in the low byte.

BR

Branch 000400



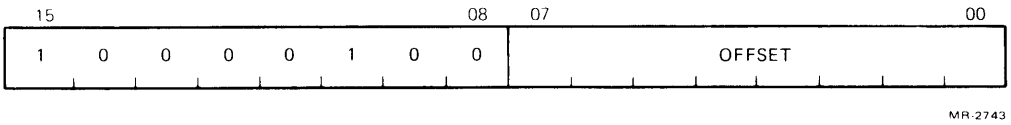
Type: PC

Operation: PC <-- PC + (2 X offset)

Condition Codes: N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Provides a way of transferring program control within a range of -128 to +127 words with a 1-word instruction. An unconditonal branch.

Branch if V bit clear 102000



Type: PC

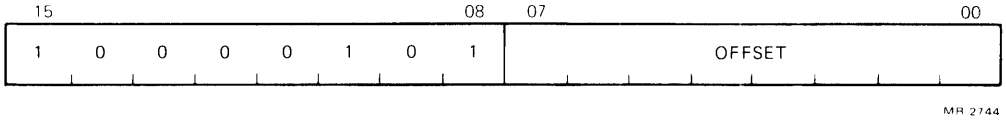
Operation: PC <-- PC + (2 X offset) if V = 0

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of the V bit and causes a branch if the V bit is clear. BVC is the complementary operation to BVS.

BVS

Branch if V bit set 102400



Type: PC

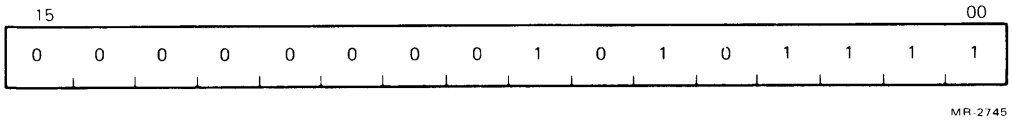
Operation: PC <-- PC + (2 X offset) if V = 1

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Tests the state of V bit (overflow) and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.

CCC

Clear all condition code bits 000257

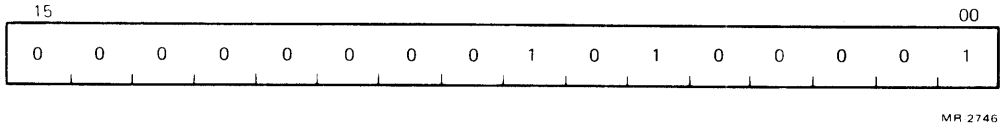


Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

CLC

Clear C 000241

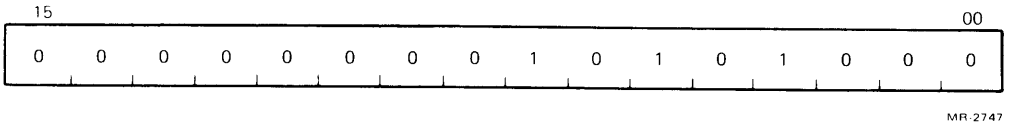


Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

Clear N

000250



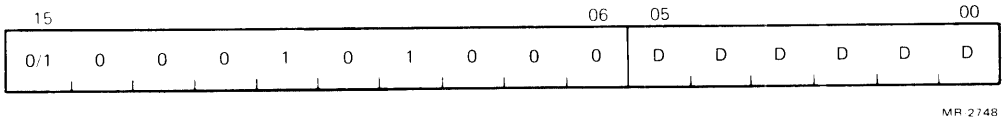
Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

CLR/CLRB

Clear

0050DD
1050DD



Type: SO

Operation: (dst) <-- 0

Condition Codes:
N: cleared
Z: set
V: cleared
C: cleared

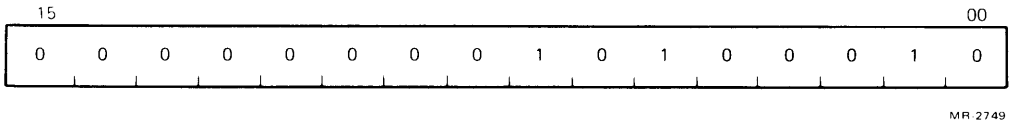
Description: Contents of specified destination are replaced with 0s.

NOTE

As a performance optimization, the last bus cycle of a CLR (or CLRB) is a DATO (or DATOB). Previous LSI-11 processors performed a DATIO cycle for the last bus cycle as a "don't care" for hardware minimization.

Clear V

000242

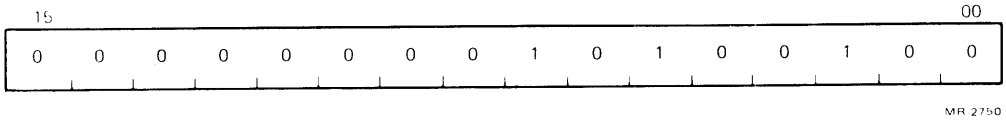


Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bit 4 = 0.

CLZ

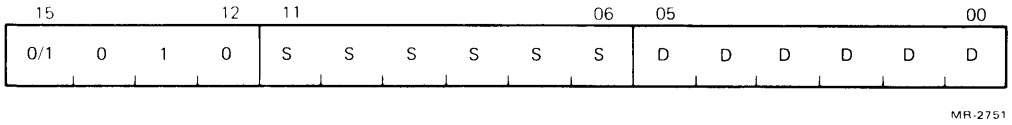
Clear Z 000244



Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

Compare 02SSDD
12SSDD



Type: DO

Operation: (src) - (dst) [in detail (src) + (dst) + 1]

Condition Codes:

N: set if result < 0

Z: set if result = 0

V: set if there is arithmetic overflow; i.e., operands of opposite signs and the sign of the destination are the same as the sign of the result

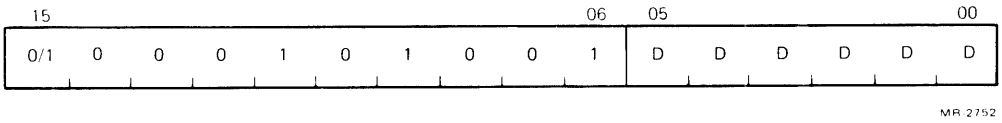
C: cleared if there is a carry from the most significant bit of the result

Description: Compares the source and destination operands and sets the condition codes, which may then be used for arithmetic and logical conditional branches. Both operands are unaffected. The only action is to set the condition codes. The compare is customarily followed by a conditional branch instruction.

COM/COMB

Complement

0051DD
1051DD



Type: SO

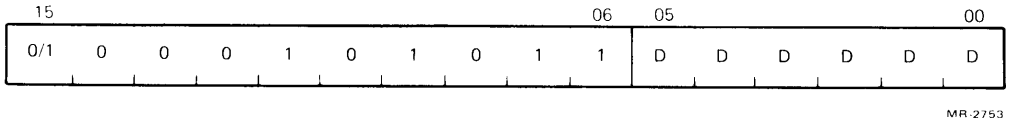
Operation: (dst) <-- ~ (dst)

Condition Codes:
N: set if most significant bit of result = 0
Z: set if result = 0
V: cleared
C: set

Description: Replaces the contents of the destination address by their logical complements (each bit equal to 0 set and each bit equal to 1 cleared).

Decrement

0053DD
1053DD



Type: SO

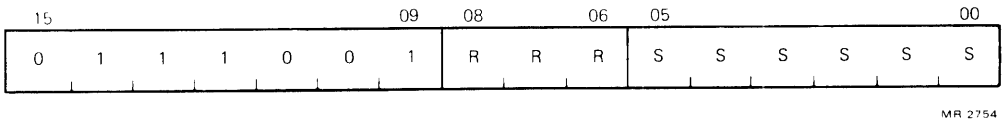
Operation: (dst) <-- (dst) - 1

Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if (dst) was 100000
C: not affected

Description: Subtract 1 from the contents of the destination.

DIV

Divide 071RSS



Type: DO

Operation: R, Rvl <-- R, Rvl/(src)

Condition Codes: N: set if quotient < 0

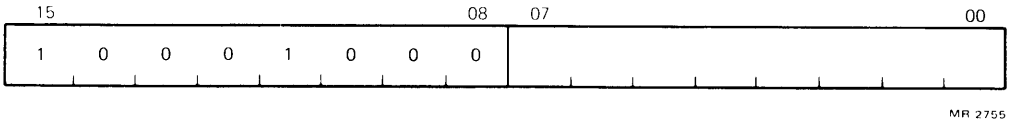
Z: set if quotient = 0

V: set if source = 0 or if the absolute value of the register is larger than the absolute value of instruction is the source. (In this case the instruction is aborted because the quotient would exceed 15 bits.)

C: set if divide by 0 attempted

Description: The 32-bit 2's complement integer in R and Rvl is divided by the source operand. The quotient is left in R; the remainder is of the same sign as the dividend. R must be even.

Emulator Trap 104000



Type: PC

Operation:
-(SP) <-- PS
-(SP) <-- PC
PC <-- (30)
PS <-- (32)

Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

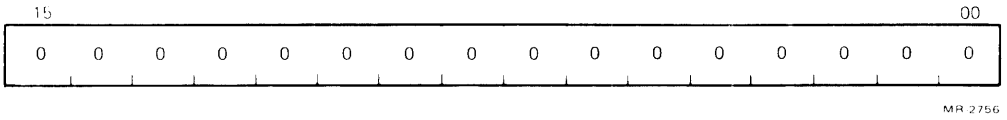
Description: All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit information to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new processor status (PS) is taken from the word at address 32.

CAUTION

EMT is used frequently by DIGITAL system software and is therefore not recommended for general use.

HALT

000000



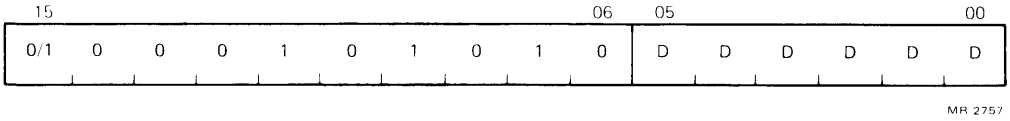
Type: MS

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes program execution to cease and enters console ODT (if memory management is present, program execution ceases only if in kernel mode; a trap to location 10 occurs if in user mode). Additionally if jumper W7 on the KDF11 module is inserted, a trap to 10 will occur unconditionally.

Increment

0052DD
1052DD



Type: SO

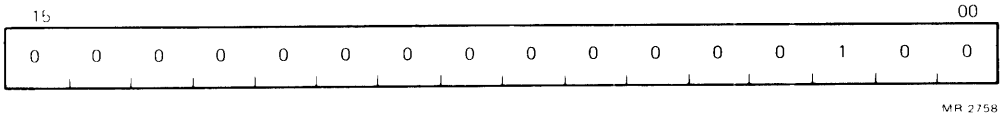
Operation: (dst) <-- (dst) + 1

Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if dst was 077777
C: not affected

Description: Adds 1 to the contents of the destination.

IOT

I/O Trap 000004



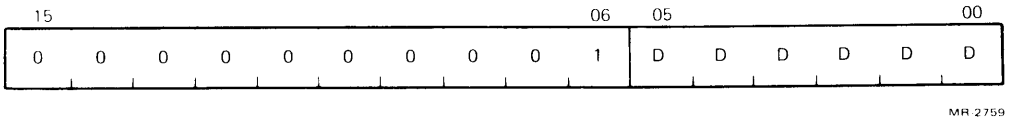
Type: PC

Operation:
-(SP) <-- PS
-(SP) <-- PC
PC <-- (20)
PS <-- (22)

Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description: Performs a trap sequence with a trap vector address of 20. Used to call the I/O executive routine IOX in the paper tape software system and for error reporting in the disk operating system. No information is transmitted in the low byte.

Jump 0001DD



Type: PC

Operation: PC <-- (dst)

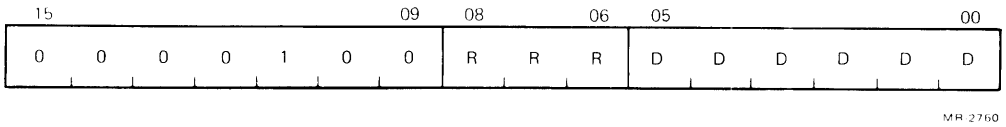
Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: JMP provides more flexible program branching then provided with the branch instruction. It is not limited to $+177_8$ and -200_8 words as are branch instructions. JMP does generate a second word, which makes it slower than branch instructions. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes with the exception of register mode 0. Execution of a jump with mode 0 will cause an illegal instruction condition and a trap to location 4. (Program control cannot be transferred to a register.) Register-deferred mode is legal and will cause program control to be transferred to the address held in the specified register.

NOTE
Instructions are word data and therefore must be fetched from an even-numbered address.

JSR

Jump to Subroutine 004RDD



Type: PC

Operation: (tmp) <-- (dst) (tmp is an internal processor register) -(SP) <-- reg

(push reg contents onto processor stack)

reg <-- PC (PC holds location following JSR; this address now put in reg)

PC <-- (tmp) (PC now points to subroutine address)

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: In execution of the JSR, the old contents of the specified register (the linkage pointer) are automatically pushed onto the processor stack and new linkage information placed in the register. Thus, subroutines nested within subroutines to any depth may all be called with the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include instructions in each routine to save and restore the linkage pointer. Further, since all linkages are saved in a re-entrant manner on the processor stack, execution of a subroutine may be interrupted, and the same subroutine re-entered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called nesting) can proceed to any level.

JSR PC, dst is a special case of the subroutine call suitable for subroutine calls that transmit parameters. JSR PC saves the use of

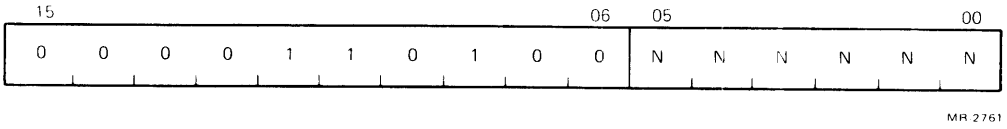
an extra register.

In both JSR and JMP the address is used to load the program counter, R7. Thus, for example, a JSR is destination mode 1 for general register R1 (where (R1) = 100) will access a subroutine at location 100. This is effectively one level less of deferral than operate instructions such as ADD.

A JSR with mode 0 will result in an illegal instruction and a trap through the trap vector address 4.

MARK

0064NN



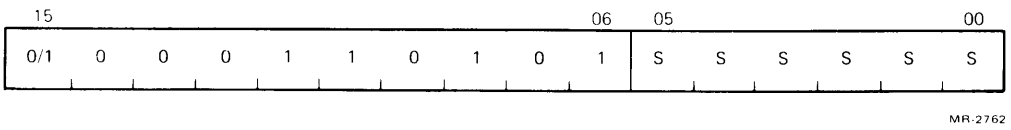
Type: PC

Operation:
SP <-- PC + 2 X NN
PC <-- R5
R5 <-- (SP) +
nn = number of parameters

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Used as part of the standard subroutine return convention. MARK facilitates the stack clean-up procedures involved in subroutine exit. Assembler format is: MARK N

Move from Previous Data Space 0065SS
Move from Previous Instruction Space 1065SS



Type: MS

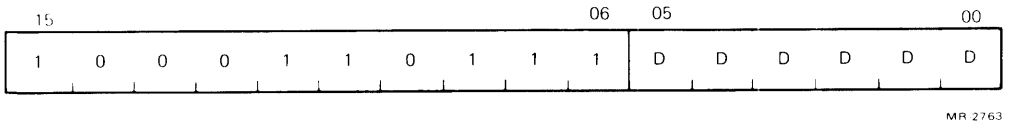
Operation:
(tmp) <-- (src)
-(SP) <-- (temp)

Condition Codes:
N: set if the source < 0
Z: set if the source = 0
V: cleared
C: unaffected

Description: Pushes a word onto the current stack from an address in previous space. The source address is computed using the current registers and memory map. Since data space does not exist in the KDF11, MFPD executes the same as a MFPI.

MFPS

1067DD



Type: MS

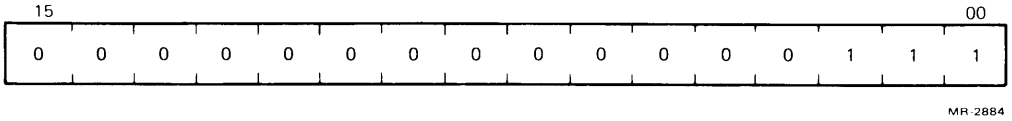
Operation: (dst) <-- PS
dst lower 8 bits

Condition Codes: N: set if PS bit 7 = 1
Z: set if PS <0:7> = 0
V: cleared
C: unaffected

Description: The 8-bit contents of the PS are moved to the effective destination. If destination mode is 0, PS bit 7 is sign-extended through upper byte of the register. The destination operand is treated as a byte address.

The KDF11 implements the PS address, 777776, which can be used as another method of accessing the PS. This method can be used on all PDP-11s except previous LSI-11 processors.

Move From Processor Type 000007



Type: MS

Operation: R0 <-- 000003

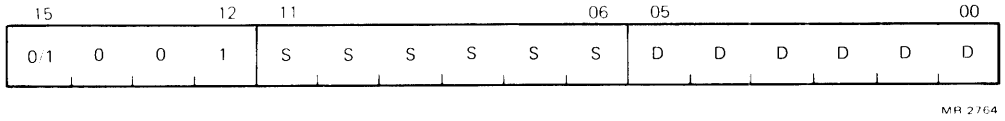
Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: A unique number assigned to each PDP-11 processor model is loaded into general register R0. The KDF11-AA processor number is 000003 and can be used to indicate which processor a program is being executed on. LSI-11 and LSI-11/2 processors treat this opcode as a reserved instruction trap.

MOV/MOVB

Move

01SSDD
11SSDD



Type: DO

Operation: (dst) <-- (src)

Condition Codes:
N: set if (src) < 0
Z: set if (src) = 0
V: cleared
C: not affected

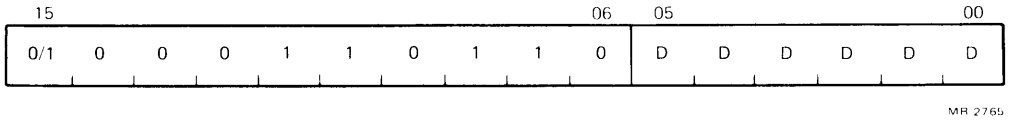
Description: Moves the source operand to the destination location. The previous contents of the destination are lost. The source operand is not affected.

Byte: Same as MOV. The MOVB to a register (mode 0) (unique among byte instructions) extends the most significant bit of the low-order byte (sign extension) into the high byte of the selected register. Otherwise MOVB operates on bytes exactly as MOV operates on words.

NOTE

As a performance optimization, the last bus cycle of a MOV (or MOVB) is a DATO (or DATOB). Previous LSI=11 processors performed a DATIO cycle for MOVB as a "don't care" for hardware minimization.

Move to Previous Data Space 1066SS
Move to Previous Instruction Space 0066SS



Type: MS

Operation: (temp) <-- (SP) +
(dst) <-- (temp)

Condition Codes: N: set if the source < 0
Z: set if the source = 0
V: cleared
C: unaffected

Description: This instruction pops a word off the current stack determined by PS (bits 15, 14) and stores that word into an address in previous space PS (bits 13, 12). The destination address is computed using the current registers and memory map.

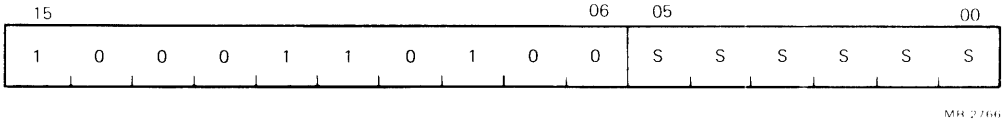
Since data space does not exist in the KDF11, MTPD executes the same as MTPI.

NOTE

As a performance optimization, the last bus cycle of a MTPD and MTPI is a DATO. This instruction was not implemented on previous LSI-11 processors.

MTPS

1064SS



Type: MS

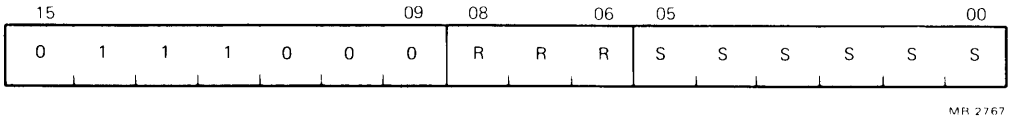
Operation: PS <-- (SRC)

Condition Codes:
N: set according to effective src operand 0-3
Z: same
V: same
C: same

Description: The eight bits of the effective operand replace the current low byte contents of the PS, if in kernel mode. Only PS bits 0 through 3 are affected if in user mode. The source operand address is treated as a byte address. Note that PS bit 4 (T bit) cannot be set with this instruction in either kernel or user mode. The src operand remains unchanged.

The KDF11 implements the PS address, 777776, which can be used as another method of accessing the PS. This method can be used on all PDP-11s except previous LSI-11 processors.

Multiply 070RSS



Type: DO

Operation: R, Rvl <-- R X (src)

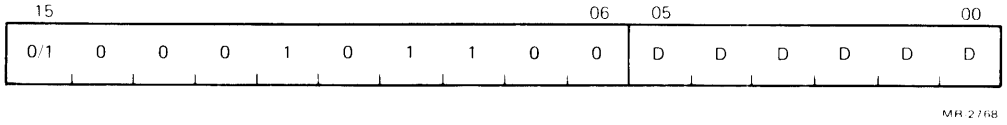
Condition Codes:
N: set if product < 0
Z: set if product = 0
V: cleared
C: set if the result is less than -2^{15} or greater than or equal to $2^{15}-1$.

Description: The contents of the destination register and source taken as 2's complement integers are multiplied and stored in the destination register and the succeeding register (if R is even). If R is odd, only the low-order product is stored. Assembler syntax is: MUL S, R. (note that the actual destination is R, Rvl, which reduces to just R when R is odd.)

NEG/NEGB

Negate

0054DD
1054DD



Type: SO

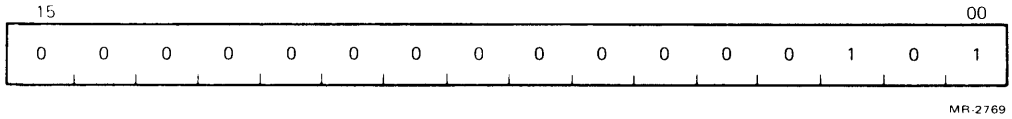
Operation: (dst) <-- (dst)

Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if result = 100000
C: cleared if result = 0

Description: Replaces the contents of the destination address by its 2's complement. Note that 100000 is replaced by itself.

RESET

000005



Type: MS

Operation: PC(SP)
PS(SP)

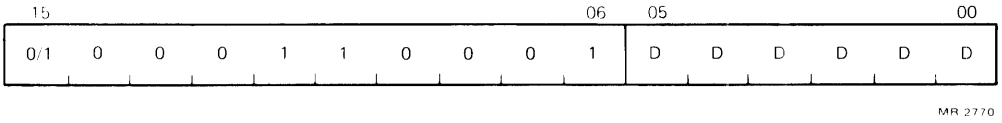
Condition Codes: N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes bus signal BINITL to be asserted for 10 microseconds and then unasserted for 90 microseconds. Used to initialize I/O devices attached to the bus. In addition, memory management status registers SR0 and SR3 are cleared.

ROL/ROLB

Rotate Left

0061DD
1061DD



Type: SO

Operation: (dst) <-- (dst)
rotate left one place

Condition Codes:

N: set if the high-order bit of the result word is set (result > 0)

Z: set if all bits of the result word = 0

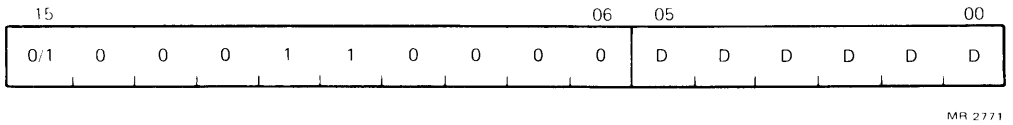
V: loaded with the exclusive OR of the N bit and C bit (as set by the completion of the rotate operation)

C: loaded with the high-order bit of the destination

Description: Rotates all bits of the destination left one place. The high-order bit is loaded into the C bit of the status word and the previous contents of the C bit are loaded into the low-order bit of the destination.

Rotate Right

0060DD
1060DD



Type: SO

Operation: (dst) <-- (dst)
rotate right one place

Condition Codes:
N: set if high-order bit of the result is set

Z: set if all bits of result are 0

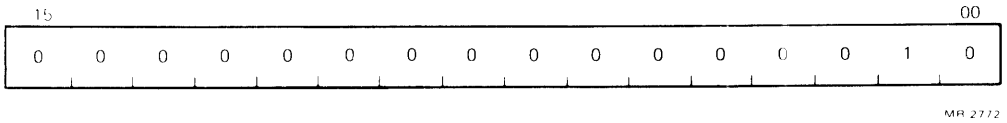
V: loaded with the exclusive OR of the N bit and the C bit as set by ROR

C: loaded with the low-order bit of the destination

Description: Rotates all bits of the destination right one place. The low-order bit is loaded into the C bit and the previous contents of the C bit are loaded into the high-order bit of the destination.

RTI

000002



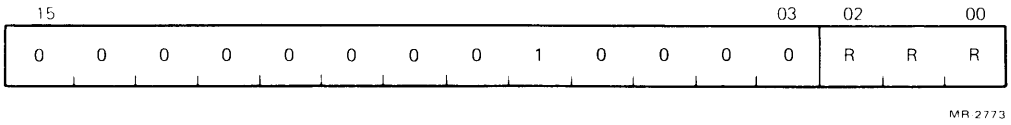
Type: MS

Operation:
PC <-- (SP) +
PS <-- (SP) +

Condition Codes:
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: Used to exit from an interrupt or trap service routine. The PC and PS are restored (popped) from the processor stack. If the RTI sets the T bit in the PS, a trace trap will occur prior to executing the next instruction.

Return from Subroutine 00020R



Type: PC

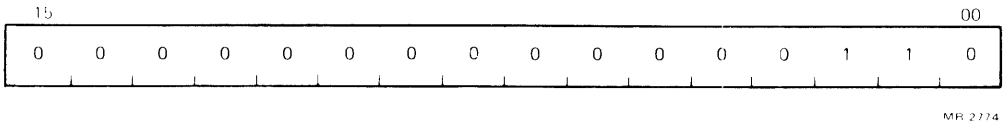
Operation: PC <-- (reg)
(reg) <-- SP +

Condition Codes: N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Loads contents of register into PC and pops the top element of the processor stack into the specified register. Return from a non-reentrant subroutine is typically made through the same register that was used in its call. Thus, a subroutine called with a JSR PC, dst exits with an RTS PC, and a subroutine called with a JSR R5, dst may pick up parameters with addressing modes (R5)+, X(R5), or @X (R5) and finally exit, with an RTS R5.

RTT

000006



Type: MS

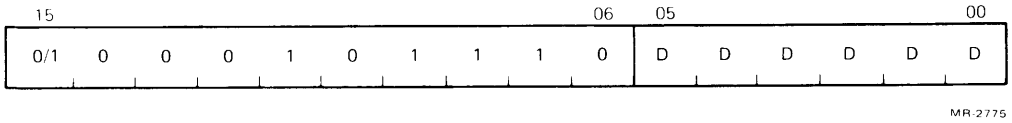
Operation:
PC <-- (SP) +
PS <-- (SP) +

Condition Codes:
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: Used to exit from a trace trap (T bit) service routine and executes the same as the RTT instruction with one exception. If the RTT sets the T bit in the PS, the next instruction will be executed and then the trace trap will be processed. However, if an RTI sets the T bit in the PS, a trace trap will occur before the next instruction is executed.

Subtract Carry

0056DD
1056DD



Type: SO

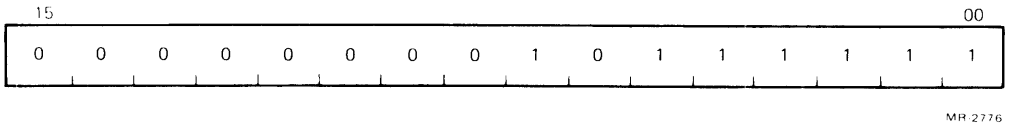
Operation: (dst) <-- (dst) - C

Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if (dst) = 100000 and C = 1
C: cleared if (dst) = 0 and C = 1

Description: Subtract the contents of the C bit from the destination. This permits the carry from the subtraction of the low-order words/bytes to be subtracted from the high-order part of the result in order to perform double precision subtraction.

SCC

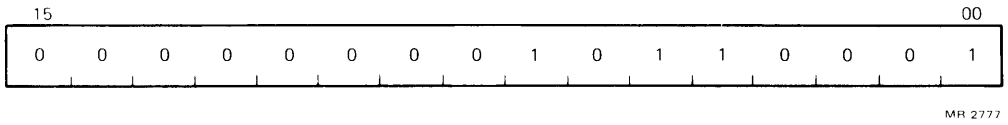
Set all Condition Code Bits 000277



Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

Set C 000261

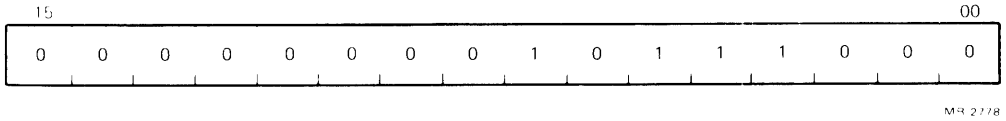


Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

SEN

Set N 000270

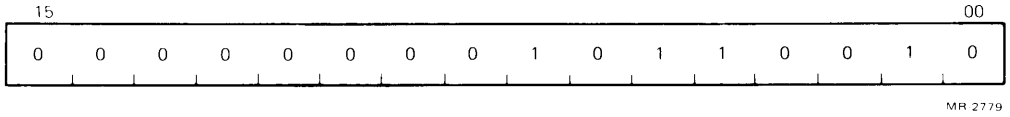


Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

Set V

000262

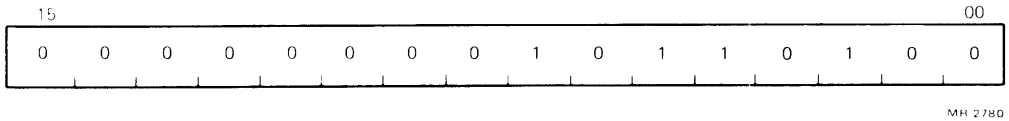


Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

SEZ

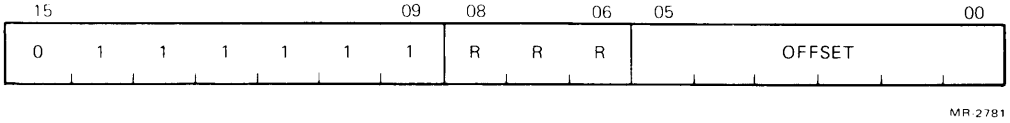
Set Z 000264



Type: CC

Description: Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., the program sets the bit specified by bit 0, 1, 2, 3, if 4 is a 1. Clears corresponding bits if bit 4 = 0.

Subtract one and branch if not equal to 0 — 077R00 plus offset



Type: PC

Operation: R <-- R - 1; if this result does not = 0 then PC <-- PC - (2 X offset)

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

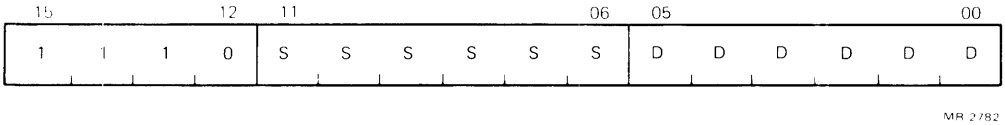
Description: The register is decremented. If it is not equal to 0, twice the offset is subtracted from the PC (now pointing to the following word). The offset is interpreted as a 6-bit positive number. This instruction provides a fast efficient method of loop control. Assembler syntax is:

SOB R, A

where A is the address to which transfer is to be made if the decremented R is not equal to 0. Note that the SOB instruction cannot be used to transfer control in the forward direction.

SUB

Subtract 16SSDD



Type: DO

Operation: (dst) <-- (dst) - (src)

Condition Codes:

N: set if result < 0

Z: set if result = 0

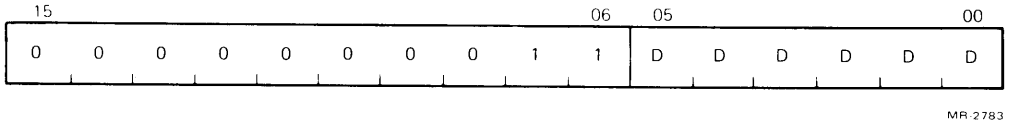
V: set if there is arithmetic overflow as a result of the operation, i.e., if the operands were of opposite signs and the sign of the source is the same as the sign of the result

C: cleared if there is a carry from the most significant bit of the result

Description: Subtracts the source operand from the destination operand and leaves the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. For double precision arithmetic, the C bit, when set, indicates a borrow.

Swap Byte

0003DD



Type: SO

Operation: Byte 1/Byte 0
Byte 0/Byte 1

Condition Codes:
N: set if high-order bit of low-order byte (bit 7) of result is set

Z: set if low-order byte of result = 0

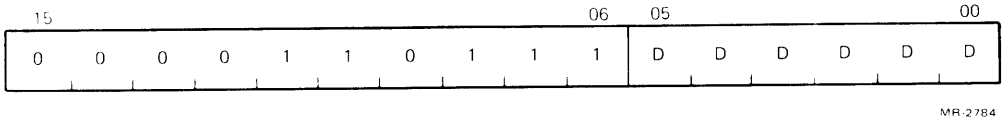
V: cleared

C: cleared

Description: Exchanges high-order byte and low-order byte of the destination which must be a word address.

SXT

Sign Extend 0067DD



Type: SO

Operation: (dst) <-- 0 if N is clear
(dst) <-- - 1 if N bit is set

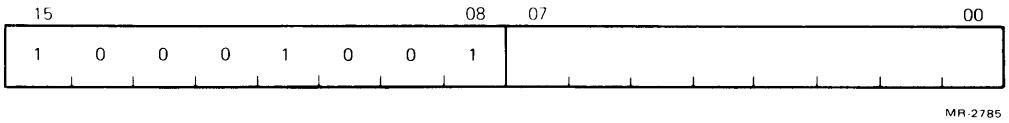
Condition Codes: N: unaffected
Z: set if N bit clear
V: cleared
C: unaffected

Description: If the condition code bit N is set, then a -1 is placed in the destination operand; if N bit is clear, then a 0 is placed in the destination operand. This instruction is particularly useful in multiple precision arithmetic because it permits the sign to be extended through multiple words.

NOTE

As a performance optimization, the last bus cycle of a SXT is a DATO. Previous LSI-11 processors performed a DATIO cycle for the last bus cycle as a "don't care" for hardware minimization.

104400 - 104777



Type: PC

Operation:
- (SP) <-- PS
- (SP) <-- PC
PC <-- (34)
PS <-- (36)

Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description: Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vecotr for TRAP is at address 34.

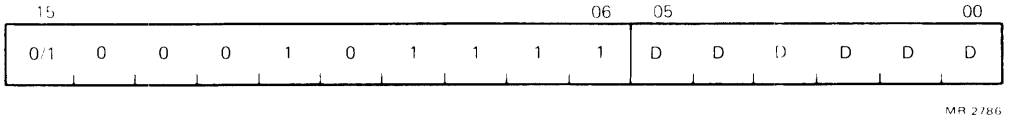
NOTE

Since DIGITAL software makes frequent use of EMT, the TRAP instruction is recommended for general use.

TST/TSTB

Test

0057DD
1057DD



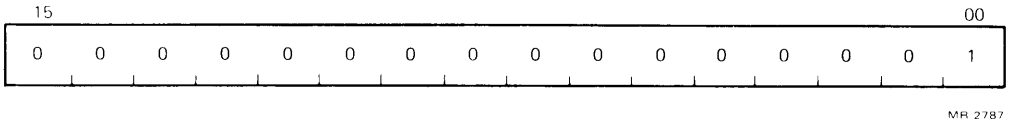
Type: SO

Operation: (dst) <-- (dst)

Condition Codes:
N: set if result < 0
Z: set if result = 0
V: cleared
C: cleared

Description: Sets the condition codes N and Z according to the contents of the destination address.

000001



Type: MS

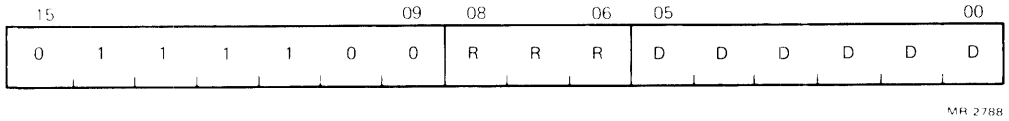
Operation:

Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt. Having been given a WAIT command, the processor will not compete for the instructions or operands from memory. This permits higher transfer rates between device and memory, since no processor-induced latencies will be encountered by bus requests from the device. In WAIT, as in all instructions, the PC points to the next instruction following the WAIT operation. Thus, when an interrupt causes the PC and PS to be pushed onto the stack, the address of the next instruction following the WAIT is saved. The exit from the interrupt routine (i.e., execution of an RTI instruction) will cause resumption of the interupted process at the instruction following the WAIT.

XOR

074RDD



Type: DO

Operation: (dst) <-- Rv(dst)

Condition Codes:
N: set if the result < 0
Z: set if result = 0
V: cleared
C: unaffected

Description: The exclusive OR of the register and destination operand is stored in the destination address. Contents of register are unaffected. Assembler format is XOR R, D.

CHAPTER 8
MEMORY MANAGEMENT

8.1 INTRODUCTION

The KDF11-AA processor implements a 128K word physical address space. This improves the 32K word maximum physical address space previously available in LSI-11 processors. The mapping or translation of 16-bit virtual addresses to 18-bit physical addresses is implemented in one MOS/LSI integrated circuit. This chip is designated the memory management unit (MMU). The memory management functionality is software-compatible with other PDP-11 processors (e.g. PDP-11/34, -11/60 and -11/70). Eight programmable relocation registers are used to accomplish the mapping function. These registers are added to the 16-bit virtual address to form an 18-bit physical address. The actual transformation occurs transparently to an executing program.

8.1.1 Programming

The memory management hardware has been designed for a multiprogramming environment. The processor can operate in two modes, kernel and user.

When in kernel mode, software has complete control and can execute all instructions. Monitors and supervisory programs are executed in this mode.

In a multiprogramming environment several user programs are resident in memory at any given time. The kernel software normally does the following.

1. Controls execution of the various user programs
2. Allocates memory and peripheral device resources
3. Safeguards the integrity of the system as a whole by careful control of each user program

When in user mode, software is executed in a restricted environment and is prevented from executing certain instructions that could be destructive to the entire software system. Some restricted instructions could cause the following.

1. Modification of the kernel program
2. Halting the computer
3. Using memory space assigned to the kernel or to other users

In a multiprogramming system, the memory management unit assigns pages (relocatable memory segments) to a user's program and prevents the user from making any unauthorized access to those

pages outside his assigned area. Thus, a user can effectively be prevented from accidental or willful destruction of any other user program or of the system executive program.

Hardware-implemented features enable the operating system to dynamically allocate memory upon demand while a program is being run.

8.1.2 Basic Addressing

The PDP-11 family word length is 16 bits wide; however, the LSI-11 bus and the KDF11-AA addressing logic are 18 bits wide. While a 16-bit word can generate virtual address references up to 32K words (64K bytes), the CPU and LSI-11 bus can reference physical 18-bit addresses up to 128K words (256K bytes). The extra two bits of addressing logic provide the basic framework for expanding memory references.

The uppermost 4K of address space is reserved for I/O device registers. The 128K physical words that can be referenced with memory management consist of 124K words of user memory and 4K words of I/O device registers.

8.1.3 Active Page Registers

The memory management unit uses two sets of eight 32-bit active page registers (APR) (see Figure 8-1). An APR is actually a pair of 16-bit registers: a page address register (PAR) and a page descriptor register (PDR). These registers are always used as a pair and contain all the information needed to describe and relocate the currently active memory pages.

One set of APRs is used in kernel mode, and the other in user mode. The set to be used is determined by the current CPU mode contained in the processor status word, bits 15 and 14.

8.1.4 Capabilities Provided by Memory Management

Memory Size (words):	128K (124K words plus 4K for I/O Page)
Address Space:	Virtual (16 bits) Physical (18 bits)
Modes of Operation:	Kernel and User
Stack Pointers:	2 (one for each mode)
Memory Relocation	
Number of Pages:	16 (8 for each mode)
Page Length:	32 to 4,096 words
Memory Page Protection:	No access Read-only Read/write

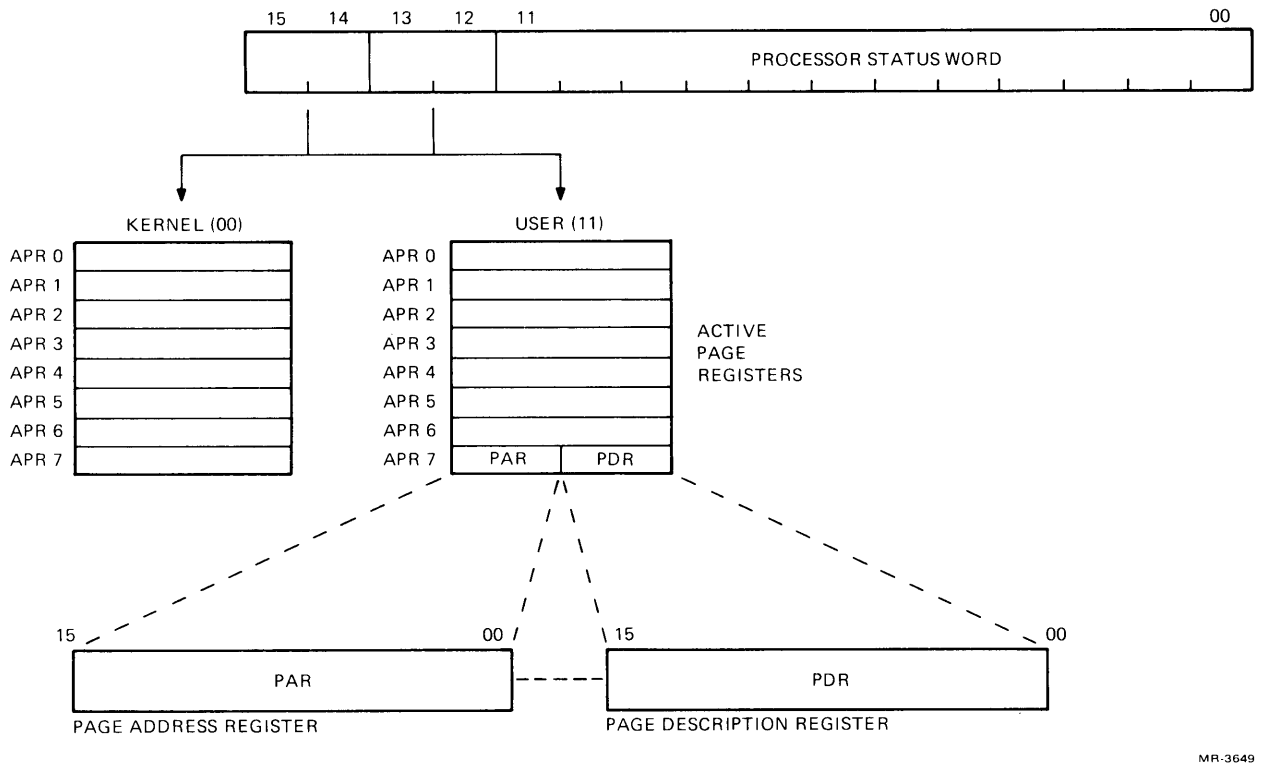


Figure 8-1 Active Page Registers

8.2 MEMORY RELOCATION

When the memory management unit is operating, the normal 16-bit direct byte address is no longer interpreted as a direct physical address (PA) but as a virtual address (VA) containing information to be used in constructing a new 18-bit physical address. The information contained in the virtual address is combined with relocation and description information contained in the active page register to yield an 18-bit physical address.

Because addresses are relocated automatically, the computer may be considered to be operating in virtual address space. This means that regardless of where a program is loaded into physical memory, it will not have to be relinked; it always appears to be at the same virtual location in memory.

The virtual address space is divided into eight 4K-word pages. Each page is relocated separately. This is a useful feature in multiprogrammed timesharing systems. It permits a new large program to be loaded into discontinuous blocks of physical memory.

A basic function is to perform memory relocation and provide extended memory addressing capability for systems with more than 32K words of physical memory. Two sets of page address registers are used to relocate virtual addresses to physical addresses in

memory. These sets are used as hardware relocation registers that permit several users' programs, each starting at virtual address 0, to reside simultaneously in physical memory.

8.2.1 Program Relocation

The page address registers are used to determine the starting physical address of each relocated program in physical memory. Figure 8-2 shows a simplified example of the relocation concept.

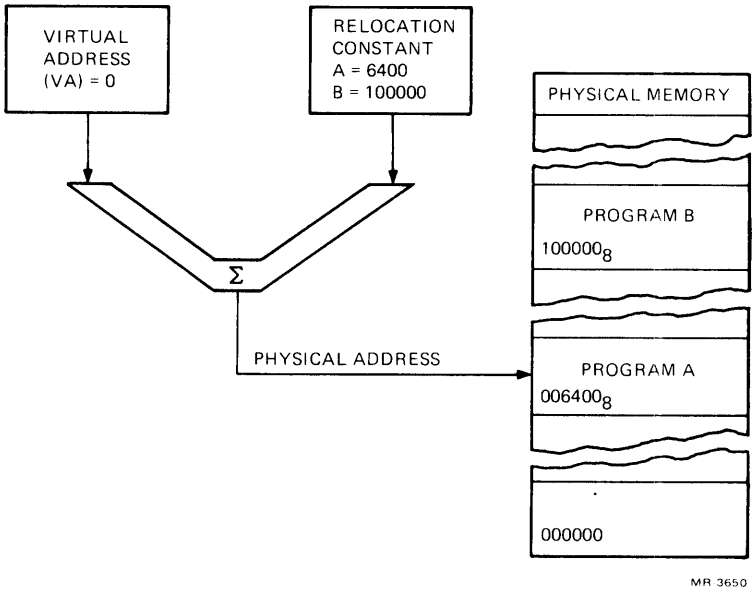


Figure 8-2 Simplified Memory Relocation

Program A starting address 0 is relocated by a constant to provide physical address 6400_8.

If the next program virtual address is 2, the relocation constant will then cause physical address 6402_8, which is the second item of program A, to be accessed. When program B is running, the relocation constant is changed to 100000_8. Then program B virtual addresses starting at 0 are relocated to access physical addresses starting at 100000_8. Using the active page address registers to provide relocation eliminates the need to relink a program each time it is loaded into a different physical memory location. The program always appears to start at the same address.

A program is relocated in pages consisting of from 1 to 128 blocks. Each block is 32 words in length. Thus, the maximum length of a page is 4096 (128 X 32) words. Using all of the eight available active page registers in a set, a maximum program length of 32,768 words can be accommodated. Each of the eight pages can be relocated anywhere in the physical memory, as long as each relocated page begins on a boundary that is a multiple of 32 words. However, for pages that are smaller then 4K words, only the memory actually allocated to the page may be accessed.

The relocation example shown in Figure 8-3 illustrates several points about memory relocation.

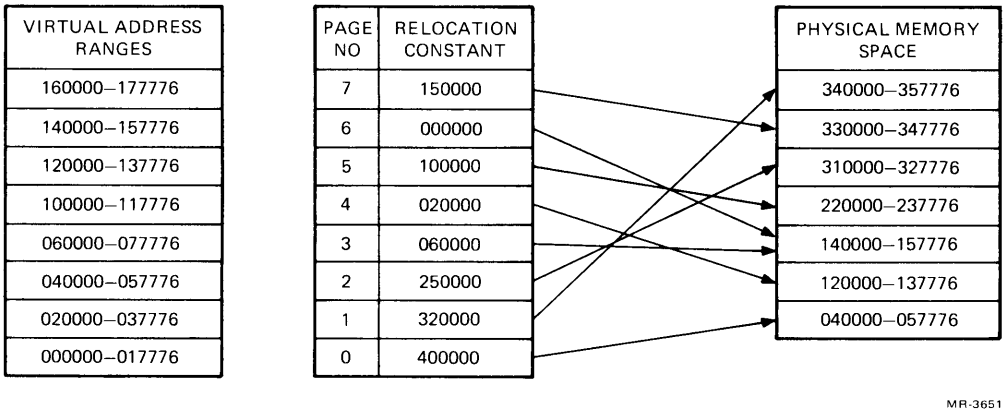


Figure 8-3 Relocation of a 32K-Word Program into 124K-Word Physical Memory

1. Although the program appears to the processor to be in contiguous address space, the 32K-word physical address space is actually scattered through several separate areas of physical memory. As long as the total available physical memory space is adequate, a program can be loaded.

2. Pages may be relocated to higher or lower physical addresses with respect to their virtual address ranges. In the example in Figure 8-3, page 1 is relocated to a higher range of physical addresses, page 4 is relocated to a lower range, and page 3 is not relocated at all (even though its relocation constant is non-0).

3. All the pages shown in the example start on 32-word boundaries.

4. Each page is relocated independently. There is no reason why two or more pages could not be relocated to the same physical memory space. Using more than one page address register in the set to access the same space would be one way of providing different memory access rights to the same data, depending on which part of the program was referencing that data.

8.2.2 Memory Units

Block:	32 words
Page:	1 to 128 blocks (32 to 4,096 words)
No. of pages:	8 per mode
Size of relocatable memory:	32,768 words, max. (8 X 4,096)

8.3 PROTECTION

A timesharing system performs multiprogramming; i.e., it allows several programs to reside in memory simultaneously and to execute sequentially. Access to these programs, and the memory space they occupy, must be strictly defined and controlled. A timesharing system requires several types of memory protection.

1. User programs must not be allowed to expand beyond their allocated space unless authorized by the system.

2. Users must be prevented from modifying common subroutines and algorithms that are resident for all users.

3. Users must be prevented from gaining control of or modifying the operating system software.

4. Users must be prevented from accessing or modifying memory occupied by other users.

Memory management provides the hardware facilities to implement all the above types of memory protection.

8.3.1 Inaccessible Memory

Each page has a 2-bit access control key associated with it. The key is part of the page descriptor register (PDR). The key is assigned under operating system control. When the key is set to 0, the page is defined as nonresident. Any attempt by a user program to access a nonresident page is prevented by an immediate halt. Using this feature to provide memory protection, only those pages associated with the current program are set to legal access keys. The access control keys of all other program pages are set to 0, which prevents illegal memory references.

8.3.2 Read-Only Memory

The access control key for a page can be set to 2, which allows read (fetch) memory references to the page, but immediately halts

any attempt to write into that page. This read-only type of memory protection can be afforded to pages that contain common data, subroutines, or shared algorithms. This type of memory protection allows the access rights to a given memory area to be user-dependent. That is, the access right to a memory area may be varied for different users by altering the access control key.

A page address register in each of the sets (kernel and user modes) may be set up to reference the same physical page in memory and each may be keyed for different access rights. For example, the user access control key might be 2 (read-only access for user programs), and the kernel access control key might be 4 (allowing complete read/write access for the operating system).

8.3.3 Multiple Address Space

There are two complete PAR/PDR sets provided, one set of registers for kernel mode and one set for user mode. This affords the operating system software another type of memory protection. The mode of operation is specified by the processor status word current mode field, or previous mode field, as determined by the current instruction. Each mode has its own corresponding stack pointer (R6) for protection as well as software considerations.

A user mode program is relocated by its own PAR/PDR set, as is a kernel programs. This makes it impossible for a program running in one mode to reference space allocated to another mode accidentally when the active page registers are set correctly. For example, a user cannot transfer to kernel space. The kernel mode address space may be reserved for resident system monitor functions, such as the basic input/output control routines, memory management trap handlers, and timesharing scheduling modules. By dividing the types of timesharing system programs functionally between the kernel and user modes, a minimum of space control housekeeping is required as the timeshared operating system sequences from one user program to the next. For example, only the user PAR/PDR set needs to be updated as each new user program is serviced. The two PAR/PDR sets implemented in the memory management unit are shown in Figure 8-4 and Figure 8-5.

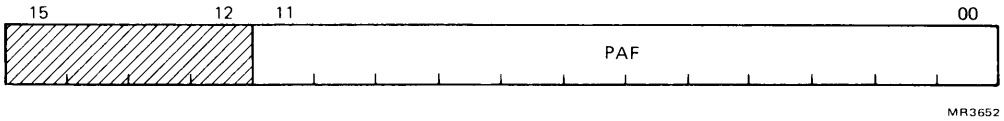


Figure 8-4 Page Address Register

8.3.3.1 Mode Specification in Processor Status Word - PS<15:14> specify the current memory management mode. These bits are used to select the corresponding PAR/PDR set to be used for the currently executing program. PS<13:12> specify the previous memory management mode. These bits are used by the memory

management instructions to communicate between kernel and user address spaces. When an implicit mode change occurs, the previous mode bits (PS<13:12>) are loaded by hardware with the contents of the current mode bits (PS<15:14>). This change can occur whenever an interrupt or trap is processed. PS<15:12> are cleared when power is applied. Clearing these bits selects kernel mode. PS<15:12> are encoded as shown below.

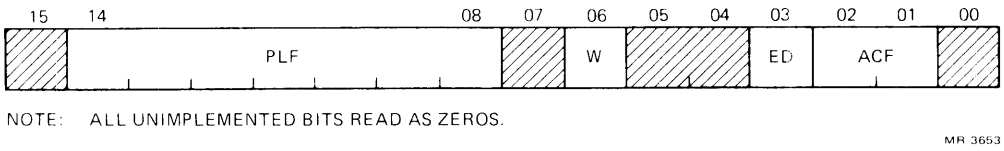


Figure 8-5 Page Descriptor Register

PS<15:14> or PS<13:12>	PAR/PDR Set Enabled	Stack Pointer Selected
00	Kernel	Kernel (KSP)
01	Reserved for future DIGITAL use. Specifies supervisor mode on some PDP-11s. Does not cause a halt.	Supervisor (SSP) - Reserved for future DIGITAL use.
10	Illegal. Does not cause a halt.	Reserved for future DIGITAL use.
11	User	USER (USP)

Each mode selects its own corresponding stack pointer. Thus all program references to register R6 use a different register as specified by PS<15:14>. Stack pointer selection occurs whether the MMU is enabled or not (SR0 bit 0 is a 1). The different stack pointers are initialized by loading the appropriate mode value in PS<15:14>, and can be examined by console ODT.

8.3.3.2 **Processor Status Word Protection** - There are various software methods of affecting PS<15:00>. Since kernel mode is defined to allow software access to all hardware features, free access to the PS is allowed. Since user mode is defined for operating user programs and thus protecting the operating system software, certain PS bits such as the mode and priority level fields are protected. Table 8-1 shows how all PS bits are affected.

Table 8-1 Processor Status Word Protection

PS Bits	RTI, RTI		Traps & Interrupts		Explicit PS Access		MTPS		Power Up
	User	Kernel	User	Kernel	User	Kernel	User	Kernel	
Condition Code PS ⟨3:0⟩	Loaded From Stack	Loaded From Stack	Loaded From Vector	Loaded From Vector	Loaded From Source	Loaded From Source	Loaded From Source	Loaded From Source	Cleared
Trap Bit PS 4	Loaded From Stack	Loaded From Stack	Loaded From Vector	Loaded From Vector	Unchanged	Unchanged	Unchanged	Unchanged	Cleared
Interrupt Priority PS ⟨7:5⟩	Unchanged	Loaded From Stack	Loaded From Vector	Loaded From Vector	Loaded From Source	Loaded From Source	Unchanged	Loaded From Source	Cleared
SI PS 8	Loaded From Stack	Loaded From Stack	Loaded From Vector	Loaded From Vector	Loaded From Source	Loaded From Source	MTPS Non-Accessible	Non-Accessible	Cleared
Previous Mode	Unchanged	Loaded From Stack	Copied From PS ⟨15:14⟩	Copied From PS ⟨15:14⟩	Loaded From Source	Loaded From Source	Non-accessible	Non-accessible	Cleared
Current Mode	Unchanged	Loaded From Stack	Loaded From Vector	Loaded From Vector	Loaded From Source	Loaded From Source	Non-accessible	Non-accessible	Cleared

8.3.3.3 User Mode Restrictions - User mode is intended for executing user programs. While in user mode the program is restricted from using those hardware features that could disrupt system integrity. The following hardware features are protected in user mode.

1. HALT instruction - Instead of entering console ODT, a HALT instruction causes a trap to kernel location 10_8. The intent is not to allow a user program to halt the operating system.

2. RESET instruction - Instead of causing a BUS initialize, a RESET instruction is executed as an NOP instruction. The intent is to prevent the user program from initializing I/O devices.

3. Access to PS<03:00> only. All other PS bits are vital to system operations and cannot be affected.

8.3.3.4 Interrupt and Trap Processing - All interrupt and trap vectors are forced by hardware to always be used in kernel mode when the new PC and PS are fetched. The processor's first step in processing the interrupt or trap is to fetch the new PS value from the interrupt or trap location plus 2. This determines which mode, and consequently which stack pointer, to use for pushing the old PC and PS. The KDF11-AA copies the old PS into a temporary register and then loads the new PS value. PS<15:14> are loaded from the memory location to select the new current mode. PS<13:12> (previous mode) are loaded with the old value in PS<15:14>, to keep a record of what the previous mode was. This is the only place where the PS previous mode bits copy the current mode bits.

This process allows communication between mode address spaces using the memory management instructions. The remaining PS bits are loaded from the memory location. Thus, interrupt and trap service routines can be executed in either kernel or user mode, depending on the contents of the vector plus 2 locations.

8.4 PAGE ADDRESS REGISTER (PAR)

The page address register (PAR), shown in Figure 8-4, contains the 12-bit page address field (PAF) that specifies the base address of the page.

Bits 15-12 are implemented but are reserved for future DIGITAL use.

The page address register may be thought of a relocation constant, or a base register containing a base address. Either interpretation indicates the basic function of the page address register (PAR) in the relocation scheme.

8.5 PAGE DESCRIPTOR REGISTER (PDR)

The page descriptor register (PDR), shown in Figure 8-5, contains information relative to page expansion, page length, and access control. Table 8-2 shows PAR/PDR address assignments.

Table 8-2 PAR/PDR Address Assignments

Kernel Active Page Registers			User Active Page Registers		
No.	PAR	PDR	No.	PAR	PDR
0	772340	772300	0	777640	777600
1	772342	772302	1	777642	777602
2	772344	772304	2	777644	777604
3	772346	772306	3	777646	777606
4	772350	772310	4	777650	777610
5	772352	772312	5	777652	777612
6	772354	772314	6	777654	777614
7	772356	772316	7	777656	777616

8.5.1 Access Control Field (ACF)

This 2-bit field, bits 2 and 1 of the PDR, describes the access rights to this particular page. The access bits specify the manner in which a page may be accessed and whether or not a given access should result in a halt of the current operation. A memory reference that causes a halt is not completed and is terminated immediately. Halts are caused by attempts to access nonresident pages, by page length errors, or by access violations such as attempting to write into a read-only page.

In the context of access control, the term "write" indicates the action of any instruction that modifies the contents of any addressable word. Table 8-3 lists the ACF keys and their functions. The ACF is written into the PDR under program control.

Table 8-3 Access Control Field Keys

ACF	Key	Description	Function
00	0	Nonresident	Halt any attempt to access this nonresident page
01	2	Resident read-only	Halt any attempt to write into this page.
10	4	Unused	Halt all accesses.
11	6	Resident read/write	Read or write allowed. No halt occurs.

8.5.2 Expansion Direction (ED)

The ED bit located in PDR bit position 3 indicates the authorized direction in which the page can expand. A logic 0 in the bit (ED = 0) indicates the page can expand upward from relative zero. A logic 1 in this bit (ED = 1) indicates the page can expand downward toward relative zero. The ED bit is written into the PDR under program control. When the expansion direction is upward (ED = 0), the page length is increased by adding blocks with higher relative addresses. Upward expansion is usually specified for program or data pages to add more program or table space. An example of page expansion upward is shown in Figure 8-6.

When the expansion direction is downward (ED = 1), the page length is increased by adding blocks with lower relative addresses. Downward expansion is specified for stack pages so that more stack space can be added. An example of page expansion downward is shown in Figure 8-7.

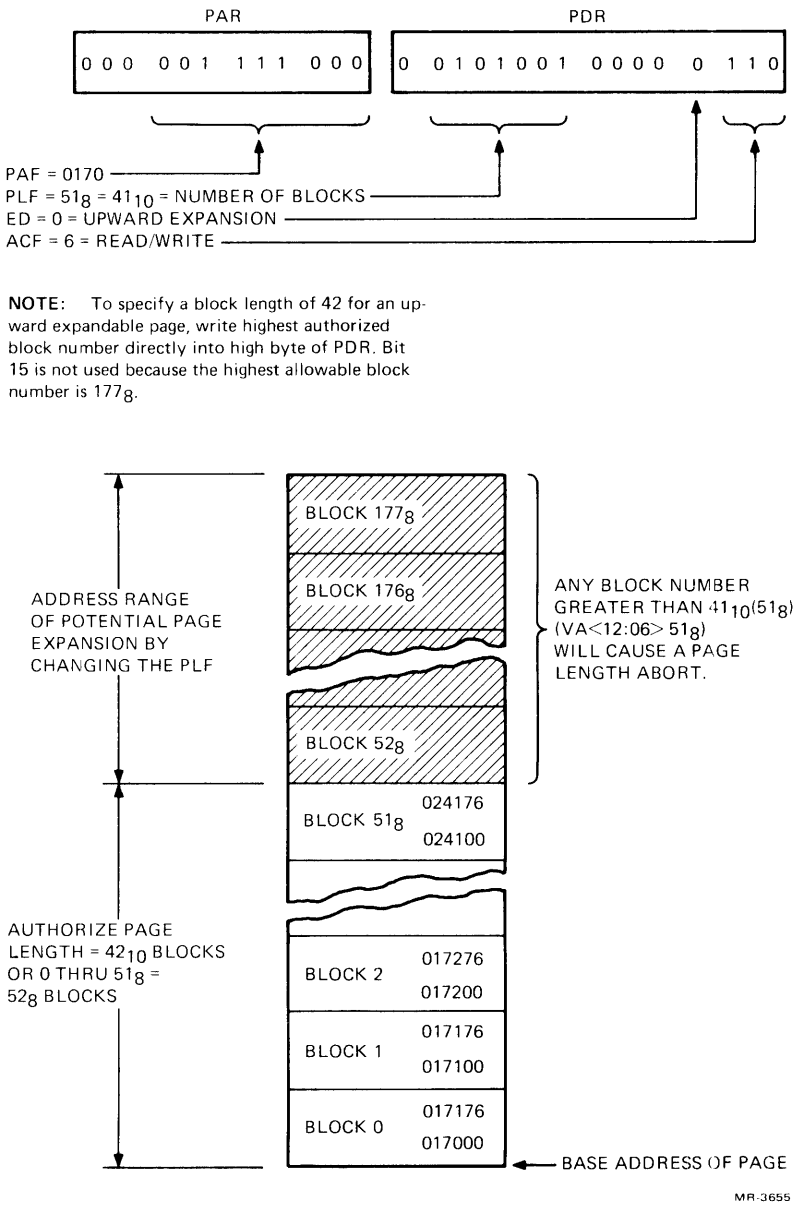


Figure 8-6 Example of an Upward-Expandable Page

8.5.3 Written Into (W)

The W bit located in PDR bit position 6 indicates whether the page has been written into since it was loaded into memory. W = 1 is affirmative. The W bit is automatically cleared when the PAR or PDR of that page is written into. It can be set only by the control logic.

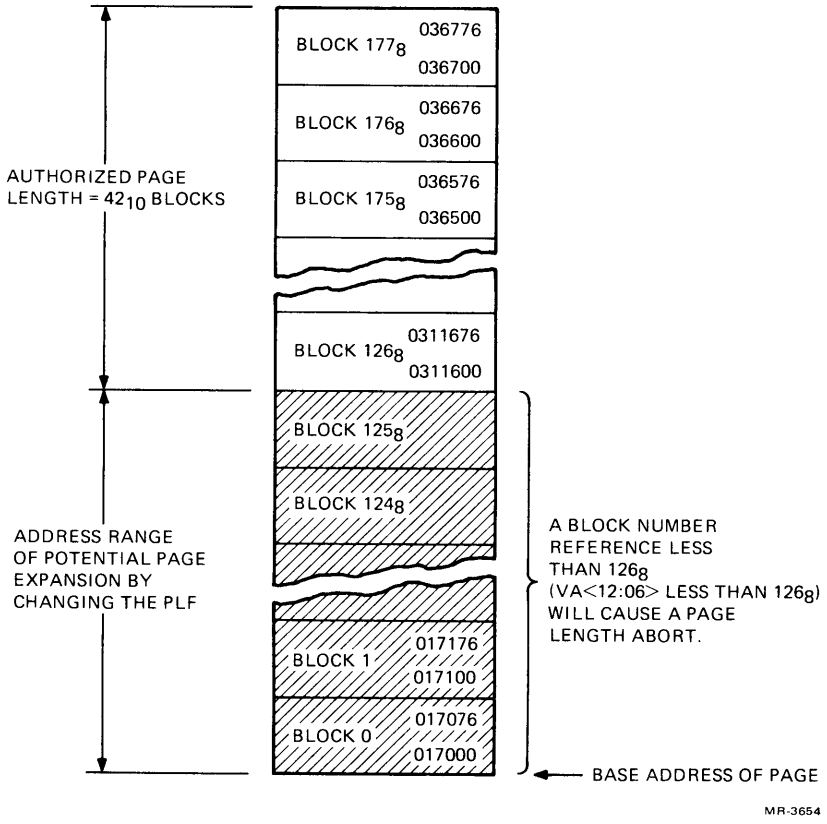


Figure 8-7 Example of a Downward-Expandable Page

In disk swapping and memory overlay applications, the W bit (bit 6) can be used to determine which pages in memory have been modified by a user. Those that have been written into must be saved in their current form. Those that have not been written into (W = 0) need not be saved and can be overlayed with new pages, if necessary.

8.5.4 Page Length Field (PLF)

The 7-bit PLF located in PDR<14:08> specifies the authorized length of the page in 32-word blocks. The PLF holds block numbers from 0 to 177_8, thus allowing any page length from 1 to 128_{10} blocks. The PLF is written into the PDR under program control.

8.5.4.1 PLF For an Upward-Expandable Page - When the page expands upward, the PLF must be set to one less than the intended number of blocks authorized for that page. For example, if 52_8 (42_{10}) blocks are authorized, the PLF is set to 51_8 (41_{10}) (Figure 8-6). The hardware compares the virtual address block number, VA<12:06> with the PLF to determine if the virtual address is within the authorized page length.

When the virtual address block number is less than or equal to the PLF, the virtual address is within the authorized page length. If the virtual address is greater than the PLF, a page length fault (address too high) is detected by the hardware and a halt occurs. In this case, the virtual address space legal to the program is noncontiguous because the three most significant bits of the virtual address are used to select the PAR/PDR set.

8.5.4.2 PLF For a Downward-Expandable Page - The capability of providing downward expansion for a page is intended specifically for those pages that are to be used as stacks. A stack starts at the highest location reserved for it and expands downward toward the lowest address as items are added to the stack.

When the page is to be downward expandable, the PLF must be set to authorize a page length, in blocks, that starts at the highest address of the page. That is always block 177_8. Refer to Figure 8-7, which shows an example of a downward-expandable page. A page length of 42_{10} blocks is arbitrarily chosen so that the example can be compared with the upward-expandable example shown in Figure 8-6.

NOTE

The same PAF is used in both examples. This is done to emphasize that the PAF, as the base address, always determines the lowest address of the page, whether it is upward- or downward-expandable.

To specify page length for a downward-expandable page, write the complement of the blocks required into high byte of PDR.

In this example, a 42-block page is required.
PLF is derived as follows:

$42_{10} = 52_8$; 2's complement = 126_8

The calculation for complementing the number of blocks required to obtain the PLF is as follows:

Maximum Block No.	Minus	Required Length	Equals	PLF
177_8	-	52_8	=	125_8
127_{10}	-	42_{10}	=	85_{10}

8.6 VIRTUAL AND PHYSICAL ADDRESSES

The memory management unit is located between the central processor unit and the LSI-11 bus address lines. When memory management is enabled, the processor ceases to supply physical address information to the bus. Instead, virtual addresses are sent to the memory management unit where they are relocated by various constants computed within the memory management unit.

8.6.1 Construction of a Physical Address

The basic information needed for the construction of a physical address (PA) comes from the virtual address (VA), which is illustrated in Figure 8-8, and the appropriate APR set.

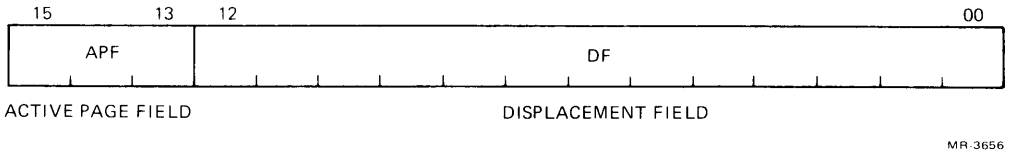


Figure 8-8 Interpretation of a Virtual Address

The virtual address consists of the following.

1. The active page field (APF) - This 3-bit field determines which of eight active page registers (APR0-APR7) will be used to form the physical address (PA).

2. The displacement field (DF) - This 13-bit field contains an address relative to the beginning of a page. This permits page lengths up to 4K words (2^{13} = 8K bytes). The DF is further subdivided into two fields as shown in Figure 8-9.

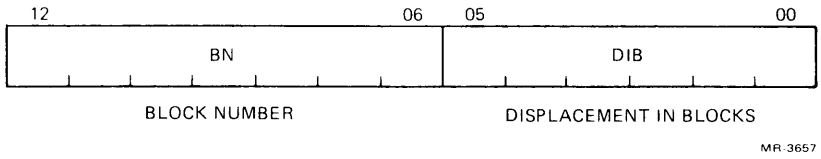


Figure 8-9 Displacement Field of Virtual Address

The displacement field (DF) consists of the following.

1. The block number (BN). This 7-bit field is interpreted as the block number within the current page.

2. The displacement in block (DIB). This 6-bit field contains the displacement within the block referred to by the block number.

The remainder of the information needed to construct the physical address comes from the 12-bit page address field (PAF) (part of the active page register) and specifies the starting address of the memory which that APR describes. The PAF is actually a block number in the physical memory; e.g., PAF = 3 indicates a starting address of 96 (3 X 32 = 96) in physical memory.

The formation of the physical address is illustrated in Figure 8-10.

The logical sequence involved in constructing a physical address is as follows.

1. Select a set of active page registers depending on current mode specified by PS<15:14>.

2. The active page field of the virtual address is used to select an active page register (APR0-APR7).

3. The page address field of the selected APR contains the starting address of the currently active page as a block number in physical memory.

4. The block number from the virtual address is added to the block number from the page address field to yield the number of the block in physical memory which will contain the physical address being constructed.

5. The displacement in blocks from the displacement field of the virtual address is joined to the physical block number to yield an 18-bit physical address.

8.6.2 Determining the Program Physical Address - A 16-bit virtual address can specify up to 32K words, in the range from 000000_8 to 177776_8 (word boundaries are even numbers). The three most significant virtual address bits designate the PAR/PDR pair to be referenced during page address relocation. Table 8-4 lists the virtual address ranges that specify each of the PAR/PDR sets.

Table 8-4 Relating Virtual Address to PAR/PDR Set

Virtual Address Range	PAR/PDR Set
000000-17776	0
020000-37776	1
040000-57776	2
060000-77776	3
100000-117776	4
120000-137776	5
140000-157775	6
160000-177776	7

NOTE

Any use of page lengths of less than 4K words causes unaddressable holes in the virtual address space.

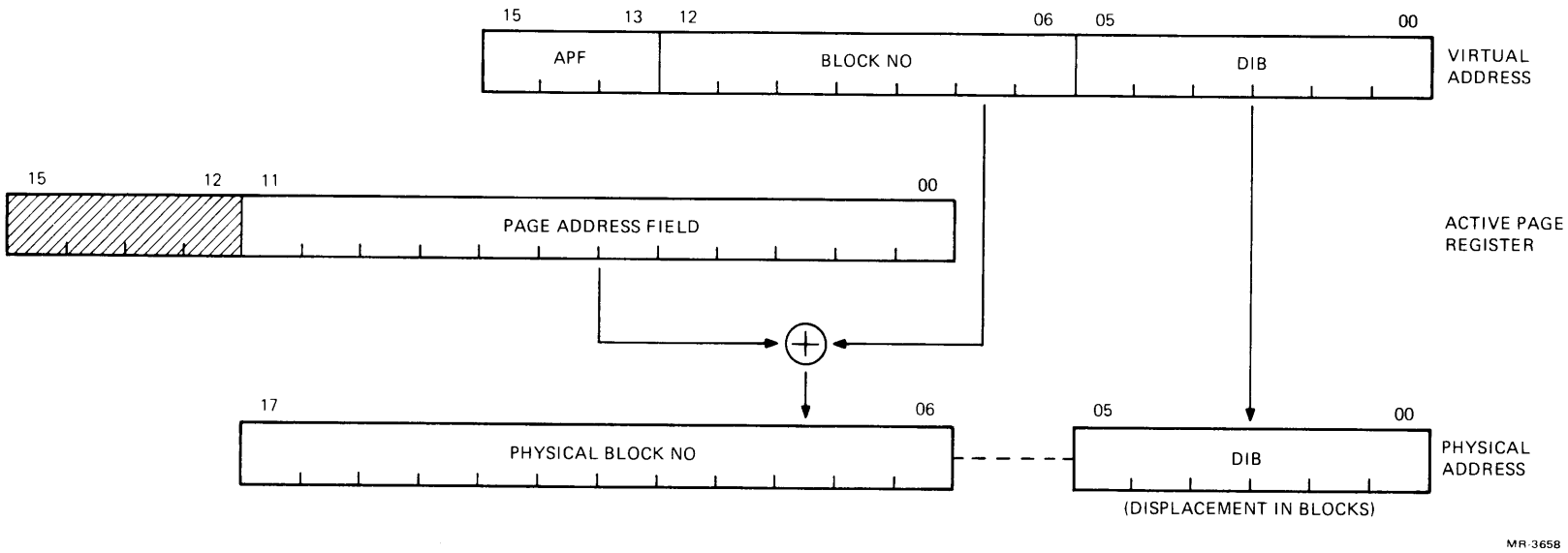


Figure 8-10 Construction of a Physical Address

8.7 STATUS REGISTERS

Halts generated by protection hardware are vectored through kernel virtual address 250. Status registers are used to determine why the halt occurred. Note that a halt to a location which is itself an invalid address will cause another halt. Thus the kernel program must ensure that kernel virtual address 250_8 is mapped into a valid physical address, otherwise an infinite loop requiring operator intervention will occur.

8.7.1 Status Register 0 (SR0)

SR0 contains halt error flags, memory management enable, and other essential information required by an operating system to recover from a halt or service a memory management trap. The SR0 format is shown in Figure 8-11. Its address is 777572_8. This register is cleared by application of power or a RESET instruction.

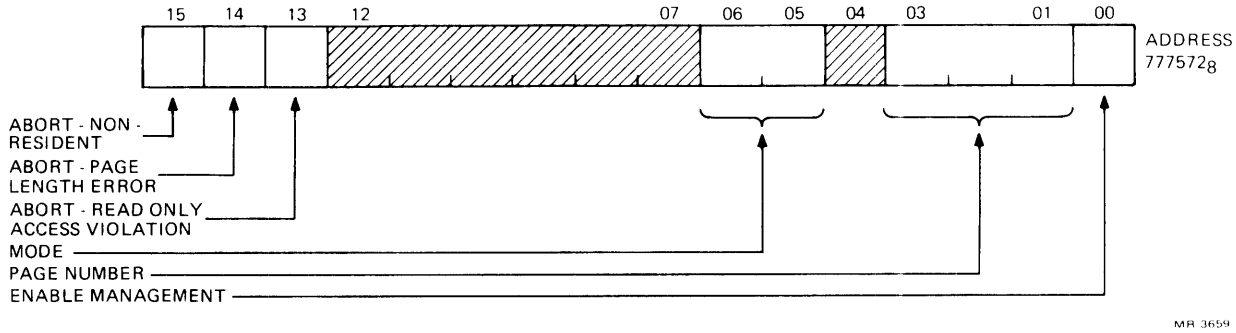


Figure 8-11 Format of Status Register 0 (SR0)

Bits 15-13 are the halt flags. They may be considered to be in priority order in that flags to the right are less significant and should be ignored. For example, a nonresident halt service routine would ignore page length and access control flags. A page length halt service routine would ignore an access control fault.

NOTE

Bit 15, 14, or 13, when set (halt conditions) cause the logic to freeze the contents of SR0 bits 1 to 6 and status register SR2. This is done to facilitate recovery from the halt.

Note that only SR0 bit 0 can be set under program control to provide memory management control information. Only that information which is automatically written into the remaining bits as a result of hardware actions is useful as a monitor of the status of the memory management unit. Setting bits 15-13 under program control will not cause traps to occur. These bits, however, must be reset to 0 by software after a halt or trap has occurred in order to resume monitoring memory management.

8.7.1.1 **Halt Nonresident** - Bit 15 is the halt nonresident bit. It is set by attempting to access a page with an access control field (ACF) key equal to 0 or 4.

8.7.1.2 **Halt Page Length** - Bit 14 is the halt page-length bit. It is set by attempting to access a location in a page with a block number (virtual address bits 12-6) that is outside the area authorized by the page length field (PLF) of the PDR for that page.

8.7.1.3 **Halt Read Only** - Bit 13 is the halt read-only bit. It is set by attempting to write in a read-only page, access key 2.

NOTE

There are no restrictions against halt bits being set simultaneously by the same access attempt.

8.7.1.4 **Mode of Operation** - Bits 5 and 6 indicate the CPU mode (user or kernel) associated with the page causing the halt (kernel = 00, user = 11).

8.7.1.5 **Page Number** - Bits 3-1 contain the virtual page number referenced. Pages, like blocks, are numbered from 0 upwards. The page number field is used by the error recovery routine to identify the page being accessed if a halt occurs.

8.7.1.6 **Enable Relocation and Protection** - Bit 0 is the enable bit. When it is 1, all addresses are relocated and protected by the memory management unit. When bit 0 is set to 0, the memory management unit is disabled and addresses are neither relocated nor protected.

8.7.2 Status Register 1 (SR1)

SR1 is implemented on some PDP-11 computers to provide additional capability. The KDF11-AA does not implement this register but does respond to its bus address, 777574_8, and reads it as all 0s. This information is provided here for clarity only.

8.7.3 Status Register 2 (SR2)

SR2 is loaded with the 16-bit virtual address (VA) at the beginning of each instruction fetch, but is not updated if the instruction fetch fails. SR2 is read-only; a write attempt will not modify its contents. SR2 is the virtual address program counter. Upon an halt, the result of SR0 bits 15, 14, or 13 being set will freeze SR2 until the SR0 halt flags are cleared. The address of SR2 is 777576_8. (See Figure 8-12.)

8.7.4 Status Register 3 (SR3)

SR3 is implemented on some PDP-11 computers to provide additional capability. The KDF11-AA implements a portion of SR3 which is reserved for future DIGITAL use. SR3 bit 4 enables I/O mapping and SR3 bit 5 enables 22-bit mapping. The address of SR3 is 772516_8. Figure 8-13 shows the format of SR3.

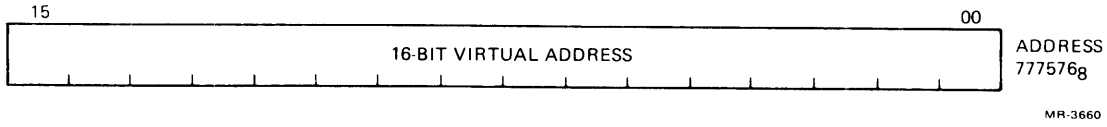


Figure 8-12 Format of Status Register 2 (SR2)

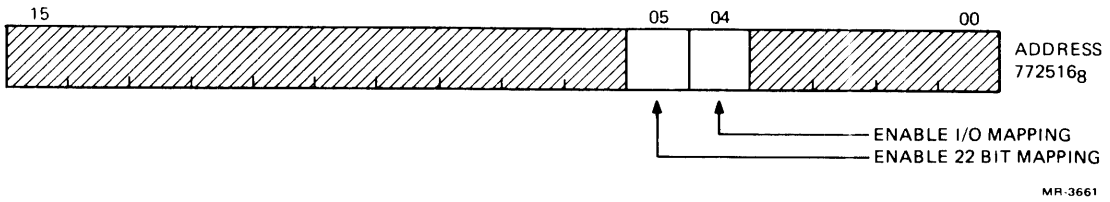


Figure 8-13 Format of Status Register 3 (SR3)

8.8 MEMORY MANAGEMENT INSTRUCTIONS

Memory management provides communication between two spaces, as determined by the current and previous modes of the processor status word (PS). The following instructions are directly applicable to memory management.

Mnemonic	Instruction	Op Code
MFPI	move from previous instruction space	0065SS
MTPI	move to previous instruction space	0066DD
MFPD	move from previous data space	1065SS
MTPD	move to previous data space	1066DD

Refer to Chapter 7, the instruction set, for a more detailed description. These instructions are directly compatible with other PDP-11 computers.

8.9 PROGRAMMING EXAMPLES

MTPI and MFPI, mode 0, register 6 are unique in that these instructions enable communications to and from the previous user stack.

```
;MFPI, MODE 0, NOT REGISTER 6

     MOV      #KM+PUM, PSW      ;KERNEL MODE, PREV USER
     MOV      #-1,-2(6)         ;MOVE -1 ON KERNEL STACK -2
     CLR      %0
     INC      @#SR0             ;ENABLE MEM MGT
     MFPI     %0                ;-(KSP)<R0 CONTENTS
```

The -1 in the kernel stack is now replaced by the contents of R0 which is 0.

```
;MFPI, MODE 0, REGISTER 6

     MOV       #UM+PUM, PSW
     CLR       %6                ;SET R16 = 0
     MOV       #KM+PUM, PSW      ;KERNEL MODE, PREV USER
     MOV       #-1, -2 (6)
     INC       @#SR0             ;ENABLE MEM MGT
     MFPI      %6                ;-(KSP)<R16 CONTENTS
```

The -1 in the kernel stack is now replaced by the contents of the user stack pointer which is 0.

To obtain information from the user stack if the status is set to kernel mode, previous user, two steps are needed.

```
     MFPI      %6                ;GET CONTENTS OF USER POINTER
     MFPI      @(6)+             ;GET USER POINTER FROM KERNEL
                                 ;STACK
                                 ;USE ADDRESS OBTAINED TO GET DATA
                                 ;FROM USER MODE USING THE PREVIOUS
                                 ;MODE
```

The desired data from the user stack is now in the kernel stack and has replaced the user stack address.

```
;MTPI, MODE 0 ,NOT REGISTER 6

     MOV       #KM+PUM, PSW      ;KERNEL MODE, PREV USER
     MOV       #TAGX, (6)        ;PUT NEW PC ON STACK
     INC       @#SR0             ;ENABLE
     MTPI      %7                ;%7<(6)+
     HLT                         ;ERROR
TAGX:CLR       @#SR0             ;DISABLE MEM MGT
```

The new PC is popped off the current stack, and since this is mode 0 and not register 6, the destination is register 7.

```
;MTPI, MODE 0, REGISTER 6

     MOV       #UM+PUM, PSW      ;USER MODE, PREVIOUS USER
     CLR       %6                ;SET USER SP=0 (R16)
     MOV       #KM+PUM, PSW      ;KERNEL MODE, PREV USER
     MOV       #-1, -(6)         ;MOVE -1 INTO K STACK (R6)
     INC       @#SR0             ;ENABLE MEM MGT
     MTPI      %6                ;%16 <(6)+
```

The 0 in R16 is now replaced with -1 from the contents of the kernel stack.

To place information on the user stack if the status is set to kernel mode, previous user mode, three separate steps are needed.

```
MFPI    %6              ;GET CONTENT OF R16=USER POINTER
MOV     #DATA, -(6)     ;PUT DATA ON CURRENT STACK
MTPI    @(6)+           ;@(6)+(FINAL ADDRESS RELOCATED)<
                        ;(R6)+
```

The data desired is obtained from the kernel stack, then the destination address is obtained from the kernel stack and relocated through the previous mode.

CHAPTER 9
FLOATING POINT

9.1 INTRODUCTION

The floating point processor (FPP) is a microcode option (KEF11-A) for use with the KDF11-AA. The KEF11-A FPP is completely software-compatible with the FP11-A used on the PDP-11/34, the FP11-E used on the PDP-11/60 and the FP11-C used on the PDP-11/70. Both single and double precision floating point capability are available together with other features including floating-to-integer and integer-to-floating conversion.

The FPP microcode resides in two MOS/LSI chips contained in one 40-pin package. The FPP requires the MMU chip, in addition to the base MOS/LSI chips, because all the floating point accumulators and status registers reside in the MMU.

9.2 FLOATING POINT DATA FORMATS

Mathematically, a floating point number may be defined as having the form (2 ** K) * f, where K is an integer and f is a fraction. For a nonvanishing number, K and f are uniquely determined by imposing the condition $1/2 \leq f < 1$. The fractional part (f) of the number is then said to be normalized. For the number 0, f must be assigned the value 0, and the value of K is indeterminate.

The FPP floating point data formats are derived from this mathematical representation for floating point numbers. Two types of floating point data are provided. In single precision, or floating mode, the data is 32 bits long. In double precision, or double mode, the data is 64 bits long. Sign magnitude notation is used.

9.2.1 Nonvanishing Floating Point Numbers

The fractional part (f) is assumed normalized, so that its most significant bit must be 1. This 1 is the "hidden" bit: it is not stored explicitly in the data word, but the microcode restores it before carrying out arithmetic operations. The floating and double modes reserve 23 and 55 bits, respectively, for f. These bits, with the hidden bit, imply effective word lengths of 24 bits and 56 bits.

Eight bits are reserved for storage of the exponent K in excess 128 (200_8) notation (i.e., as K + 200_8), giving a biased exponent. Thus exponents from -128 to +127 could be represented by 0 to 377_8, or 0 to 255_{10}. For reasons given below, a biased exponent of 0 (true exponent of -200_8), is reserved for floating point 0. Thus exponents are restricted to the range -127 to +127 inclusive (-177_8 to $+177_8$) or, in excess 200_8 notation, 1 to 377_8.

The remaining bit of the floating point word is the sign bit. The number is negative if the sign bit is a 1.

9.2.2 Floating Point Zero

Because of the hidden bit, the fractional part is not available to distinguish between 0 and nonvanishing numbers whose fractional part is exactly 1/2. Therefore the FPP reserves a biased exponent of 0 for this purpose and any floating point number with a biased exponent of 0 either traps or is treated as if it were an exact 0 in arithmetic operations. An exact or clean 0 is represented by a word whose bits are all 0s. A dirty 0 is a floating point number with a biased exponent of 0 and a nonzero fractional part. An arithmetic operation for which the resulting true exponent exceeds 277_8 is regarded as producing a floating overflow; if the true exponent is less than -177_8, the operation is regarded as producing a floating underflow. A biased exponent of 0 can thus arise from arithmetic operations as a special case of overflow (true exponent = -200_8). (Recall that only eight bits are reserved for the biased exponent.) The fractional part of results obtained from such overflow and underflow is correct.

9.2.3 The Undefined Variable

The undefined variable is defined as any bit pattern with a sign bit of 1 and a biased exponent of 0. The term "undefined variable" is used, for historical reasons, to indicate that these bit patterns are not assigned a corresponding floating point arithmetic value. Note that the undefined variable is frequently referred to as -0 elsewhere in this specification.

A design objective of the FPP was to assure that the undefined variable would not be stored as the result of any floating point operation in a program run with the overflow and underflow interrupts disabled. This is achieved by storing an exact 0 on overflow and underflow, if the corresponding interrupt is disabled. This feature, together with an ability to detect reference to the undefined variable (implemented by the FIOV bit discussed later), is intended to provide the user with a debugging aid: if the presence of -0 occurs, it did not result from a previous floating point arithmetic instruction.

9.2.4 Floating Point Data

Floating point data is stored in words of memory as illustrated in Figures 9-1 and 9-2.

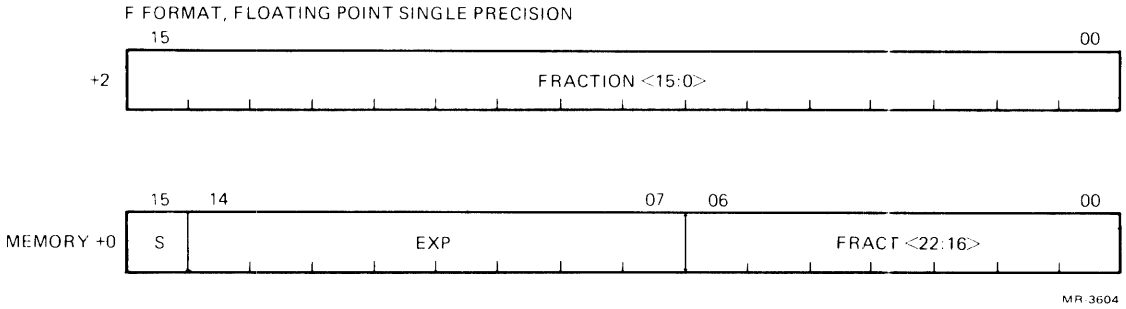


Figure 9-1 Single Precision Format

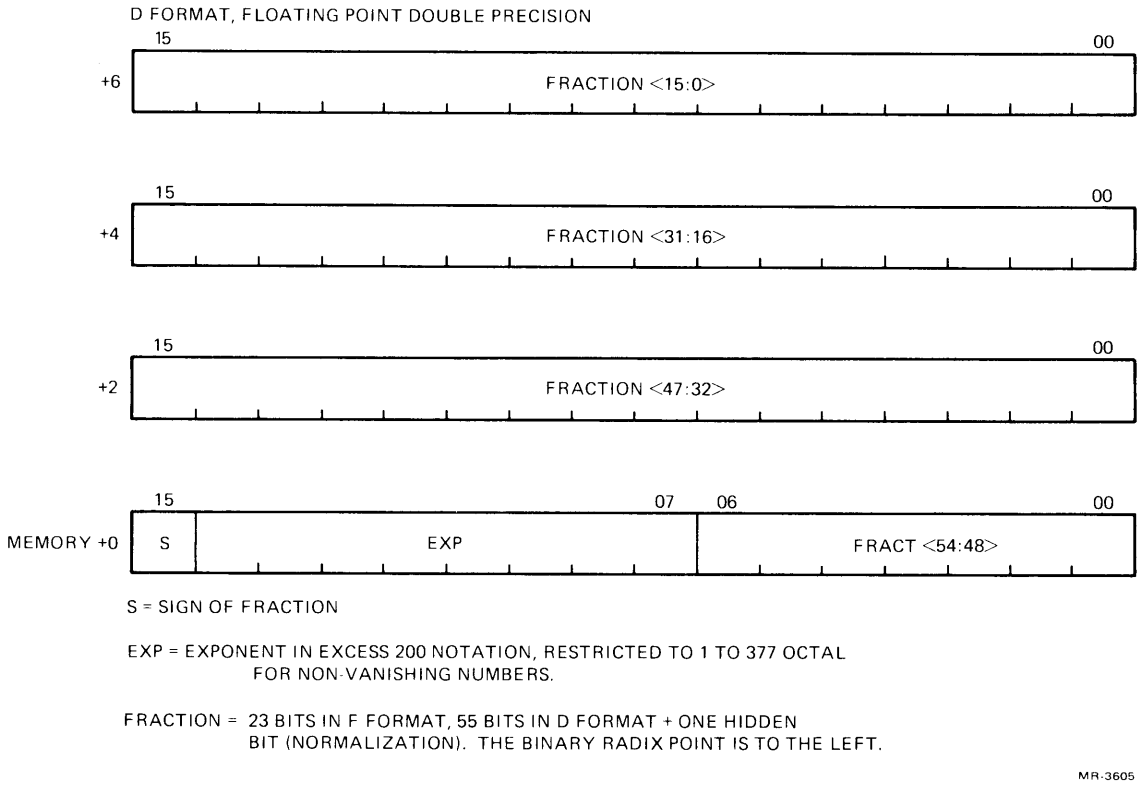


Figure 9-2 Double Precision Format

The FPP provides for conversion of floating point to integer format and vice-versa. The processor recognizes single precision integer (I) and double precision integer long (L) numbers, which are stored in standard 2's complement form. (See Figure 9-3.)

9.3 FLOATING POINT STATUS REGISTER (FPS)

This register provides mode and interrupt control for the floating point unit and conditions resulting from the execution of the previous instruction. (See Figure 9-4.)

For the purposes of discussion a set bit = 1, and a reset bit = 0. Three bits of the FPS register control the modes of operation.

Single/Double: Floating point numbers can be either single or double precision.

Long/Short: Integer numbers can be 16 bits or 32 bits.

Chop/Round: The result of a floating point operation can be either chopped or rounded. The term "chop" is used instead of "truncate" in order to avoid confusion with truncation of series used in approximations for function subroutines.

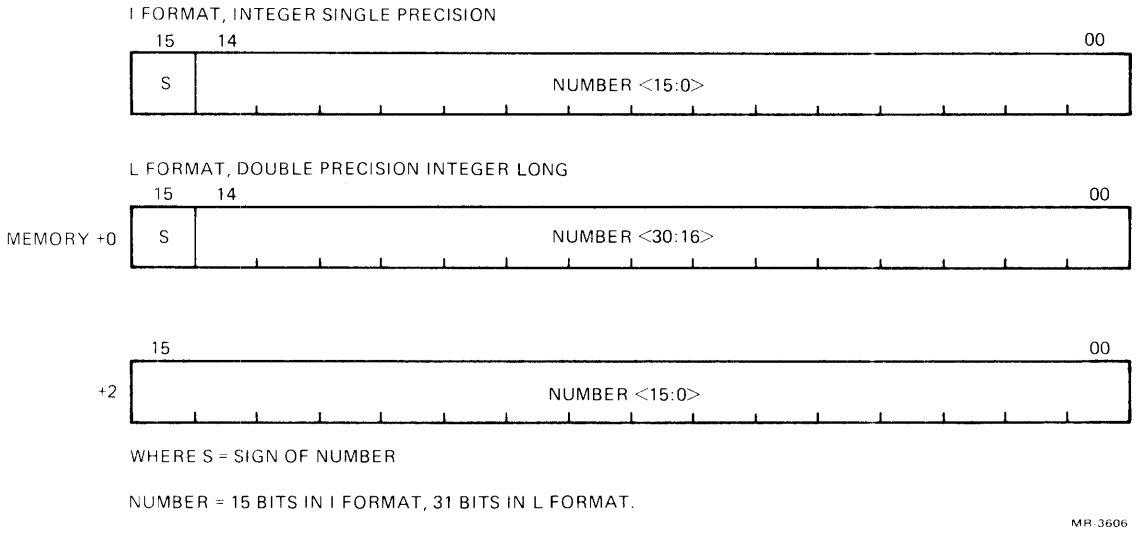


Figure 9-3 2's Complement Format

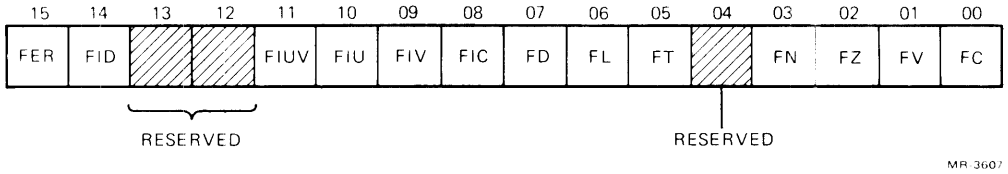


Figure 9-4 Floating Point Status Register

The FPS register contains an error flag and four condition codes (5 bits): carry, overflow, zero, and negative, which are analogous to the processor status condition codes.

The FPP recognizes six floating point exceptions:

Detection of the presence of the undefined variable in memory
Floating overflow
Floating underflow
Failure of floating to integer conversion
Attempt to divide by 0
Illegal floating opcode.

For the first four of these exceptions, bits in the FPS register are available to individually enable and disable interrupts. An interrupt on the occurrence of either of the last two exceptions can be disabled only by setting a bit which disables interrupts on all six of the exceptions, as a group.

Of the thirteen FPS bits, five are set by the FPP as part of the output of a floating point instruction: the error flag and condition codes. Any of the mode and interrupt control bits may be set by the user; the LDFPS instruction is available for this purpose. These thirteen bits are stored in the FPS register as shown in Figure 9-4. The FPS register bits are described in Table 9-1.

Table 9-1 FPS Register Bits

Bit	Name	Description
15	Floating Error (FER)	The FER bit is set by the FPP if 1. Division by zero occurs 2. Illegal opcode occurs 3. Any one of the remaining occurs and the corresponding interrupt is enabled. Note that the above action is independent of whether the FID bit is set or clear. Note also that the FPP never resets the FER bit. Once the FER bit is set by the FPP, it can be cleared only by an LDFPS instruction (note the RESET instruction does not clear the FER bit). This means that the FER bit is up to date only if the most recent floating point instruction produced a floating point exception.
14	Interrupt Disable (FID)	If the FID bit is set, all floating point interrupts are disabled. NOTES 1. The FID bit is primarily a maintenance feature. It should normally be clear. In particular, it must be clear if one wishes to assure that storage of -0 by the FPP is always accompanied by an interrupt. 2. Throughout the rest of this chapter, it is assumed that the FID bit is clear in all discussions involving overflow, underflow, occurrence of -0, and integer conversion errors.

Table 9-1 FPS Register Bits (Cont)

Bit	Name	Description
13		Reserved for future DIGITAL use.
12		Reserved for future DIGITAL use.
11	Interrupt on Undefined Variable (FIUV)	An interrupt occurs if FIUV is set and a -0 is obtained from memory as an operand of ADD, SUB, MUL, DIV, CMP, MOD, NEG, ABS, TST, or any LOAD instruction. The interrupt occurs before execution on the KEF11-A except on NEG, ABS, and TST for which it occurs after execution. When FIUV is reset, -0 can be loaded and used in any FPP operation. Note that the interrupt is not activated by the presence of -0 in an AC operand of an arithmetic instruction; in particular, trap on -0 never occurs in mode 0. The KEF11-A will not store a result of -0 without the simultaneous occurrence of an interrupt.
10	Interrupt on Underflow (FIU)	When the FIU bit is set, floating underflow will cause an interrupt. The fractional part of the result of the operation causing the interrupt will be correct. The biased exponent will be too large by 400_8, except for the special case of 0, which is correct. An exception is discussed later in the detailed description of the LDEXP instruction. If the FIU bit is reset and if underflow occurs, no interrupt occurs and the result is set to exact 0.
9	Interrupt on Overflow (FIV)	When the FIV bit is set, floating overflow will cause an interrupt. The fractional part of the result of the operation causing the overflow will be correct. The biased exponent will be too small by 400_8. If the FIV is reset and overflow occurs, there is no interrupt. The FPP returns exact 0.

Table 9-1 FPS Register Bits (Cont)

Bit	Name	Description
		Special cases of overflow are discussed in the detailed descriptions of the MOD and LDEXP instruction.
8	Interrupt on Integer Conversion Error (FIC)	When the FIC bit is set and a conversion to integer instruction fails, an interrupt will occur. If the interrupt occurs, the destination is set to 0, and all other registers are left untouched. If the FIC bit is reset, the result of the operation will be the same as detailed above, but no interrupt will occur. The conversion instruction fails if it generates an integer with more bits than can fit in the short or long integer word specified by the FL bit (bit 6).
7	Floating Double Precision Mode (FD)	The FD bit determines the precision that is used for floating point calculations. When set, double precision is assumed; when reset, single precision is used.
6	Floating Long Integer Mode (FL)	The FL bit is active in conversion between integer and floating point format. When set, the integer format assumed is double precision 2's complement (i.e., 32 bits). When reset, the integer format is assumed to be single precision 2's complement (i.e., 16 bits).
5	Floating Chop Mode (FT)	When the FT bit is set, the result of any arithmetic operation is chopped (or truncated). When reset, the result is rounded.
4		Reserved for future DIGITAL use.
3	Floating Negative (FN)	FN is set if the result of the last operation was negative, otherwise it is reset.
2	Floating Zero (FZ)	FZ is set if the result of the last operation was 0, otherwise it is reset.

Table 9-1 FPS Register Bits (Cont)

Bit	Name	Description
1	Floating Overflow (FV)	FV is set if the last operation resulted in an exponent overflow, otherwise it is reset.
0	Floating Carry (FC)	FC is set if the last operation resulted in a carry of the most significant bit. This can only occur in floating or double to integer conversions.

9.4 FLOATING EXCEPTION CODE AND ADDRESS REGISTERS

One interrupt vector is assigned to take care of all floating point exceptions (location 244_8). The six possible errors are coded in the 4-bit floating exception code (FEC) register as follows:

```
 2    Floating opcode error
 4    Floating divide by 0
 6    Floating to integer conversion error
 8    Floating overflow
10    Floating underflow
12    Floating undefined variable.
```

The address of the instruction producing the exception is stored in the floating exception address (FEA) register.

The FEC and FEA registers are updated only when one of the following occurs:

1. Divide by 0
2. Illegal opcode
3. Any of the other four exceptions with the corresponding interrupt enabled.

This implies that when and only when the FER bit is set by the FPP are the FEC and FEA registers updated.

NOTES

1. If one of the last four exceptions occurs with the corresponding interrupt disabled, the FEC and FEA are not updated.
2. Inhibition of interrupts by the FID bit does not inhibit updating of the FEC and FEA, if an exception occurs.

3. The FEC and FEA do not get updated if no exception occurs. This means that the STST (store status) instruction will return current information only if the most recent floating point instruction produced an exception.

4. Unlike the FPS register, no instructions are provided for storage into the FEC and FEA registers.

9.5 FLOATING POINT PROCESSOR INSTRUCTION ADDRESSING

Floating point processor instructions use the same type of addressing as the central processor instructions. A source or destination operand is specified by designating one of eight addressing modes and one of eight central processor general registers to be used in the specified mode. The modes of addressing are the same as those of the central processor except mode 0. In mode 0 the operand is located in the designated floating point processor accumulator, rather than in a central processor general register. The modes of addressing are as follows.

0 = FPP accumulator
1 = Deferred
2 = Autoincrement
3 = Autoincrement deferred
4 = Autodecrement
5 = Autodecrement deferred
6 = Indexed
7 = Indexed deferred

Autoincrement and autodecrement operate on increments and decrements of 4 for F format and 10_8 for D format.

In mode 0, the user can make use of all six FPP accumulators (AC0-AC5) as his source or destination. Specifying FPP accumulators AC6 or AC7 will result in an illegal opcode trap. In all other modes, which involve transfer of data to or from memory or the general registers, the user is restricted to the first four FPP accumulators (AC0-AC3). When reading or writing a floating point number from or to memory, the low memory word contains the most significant word of the floating point number and the high memory word the least significant word.

9.6 ACCURACY

General comments on the accuracy of the FPP are presented here. The descriptions of the individual instructions include the accuracy at which they operate. An instruction or operation is regarded as "exact" if the result is identical to an infinite precision calculation involving the same operands. The a priori accuracy of the operands is thus ignored. All arithmetic instructions treat an operand whose biased exponent is 0 as an exact 0 (unless FIUV is enabled and the operand is -0, in which

case an interrupt occurs). For all arithmetic operations, except DIV, a 0 operand implies that the instruction is exact. The same statement holds for DIV if the 0 operand is the dividend. But if it is the divisor, division is undefined and an interrupt occurs.

For nonvanishing floating point operands, the fractional part is binary normalized. It contains 24 bits or 56 bits for floating mode and double mode, respectively. For ADD, SUB, MUL, and DIV, two guard bits are necessary and sufficient for the general case to guarantee return of a chopped or rounded result identical to the corresponding infinite precision operation chopped or rounded to the specified word length. Thus, with two guard bits, a chopped result has an error bound of one least significant bit (LSB); a rounded result has an error bound of 1/2 LSB. These error bounds are realized by the KEF11-A of all instructions. Both the FP11-A and the FP11-E have an error bound greater than 1/2 LSB for ADD and SUB.

In the rest of this specification, an arithmetic result is called exact if no nonvanishing bits would be lost by chopping. The first bit lost in chopping is referred to as the "rounding" bit. The value of a rounded result is related to the chopped result as follows.

1. If the rounding bit is 1, the rounded result is the chopped result incremented by an LSB.

2. If the rounding bit is 0, the rounded and chopped results are identical.

It follows that:

1. If the result is exact,
   rounded value = chopped value = exact value

2. If the result is not exact, its magnitude

   a. is always decreased by chopping
   b. is decreased by rounding if the rounding bit is 0
   c. is increased by rounding if the rounding bit is 1.

Occurrence of floating point overflow and underflow is an error condition: the result of the calculation cannot be correctly stored because the exponent is too large to fit into the eight bits reserved for it. However, the internal hardware has produced the correct answer. For the case of underflow, replacement of the correct answer by 0 is a reasonable resolution of the problem for many applications. This is done by the KEF11-A if the underflow interrupt is disabled. The error incurred by this action is an absolute rather than a relative error; it is bounded (in absolute value) by 2**(-128). There is no such simple resolution for the case of overflow. The action taken, if the overflow interrupt is disabled, is described under FIV (bit 9) of Paragraph 9.3.

The FIV and FIU bits (of the floating point status word) provide the user with an opportunity to implement his own correction of an overflow or underflow condition. If such a condition occurs and the corresponding interrupt is enabled, the microcode stores the fractional part and the low eight bits of the biased exponent. The interrupt will take place and the user can identify the cause by examination of the FV (floating overflow) bit of the FEC (floating exception) register. The reader can readily verify that (for the standard arithmetic operations ADD, SUB, MUL, and DIV) the biased exponent returned by the instruction bears the following relation to the correct exponent generated by the microcode.

1. On overflow, it is too small by 400_8.

2. On underflow, if the biased exponent is 0, it is correct. If it is not 0, it is too large by 400_8.

Thus, with the interrupt enable, enough information is available to determine the correct answer. The user may, for example, rescale his variables (via STEXP and LDEXP) to continue a calculation. Note that the accuracy of the fractional part is unaffected by the occurrence of underflow or overflow.

9.7 FLOATING POINT INSTRUCTIONS

Each instruction that references a floating point number can operate on either single or double precision numbers depending on the state of the FD mode bit. Similarly, there is a mode bit FL that determines whether a 32-bit integer (FL = 1) or a 16-bit integer (FL = 0) is used in conversion between integer and floating point representations. FSRC and FDST operands use floating point addressing modes (see Figure 9-5); SRC and DST operands use CPU addressing modes.

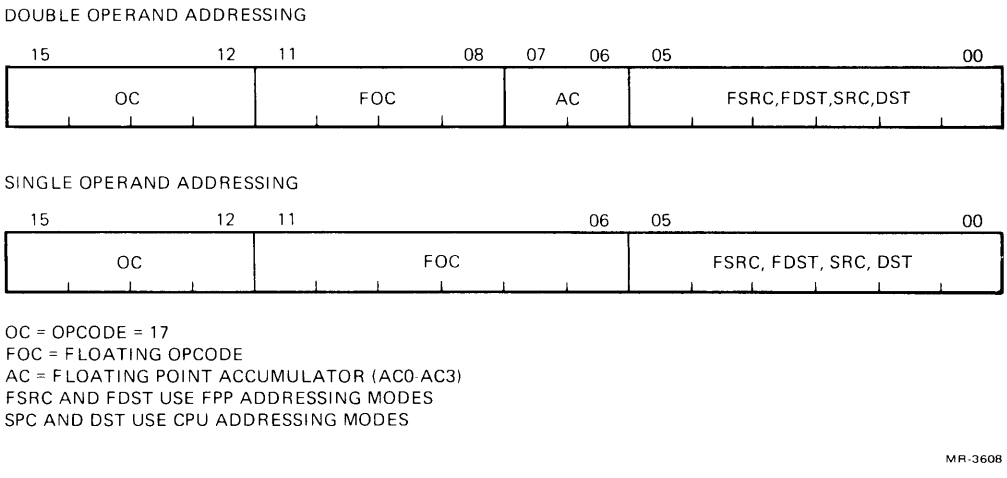


Figure 9-5 Floating Point Addressing Modes

Terms Used in Instruction Definitions

XL = largest fraction that can be represented:
    1 - 2 ** (-24), FD = 0; single precision
    1 - 2 ** (-56), FD = 1; double precision

XLL = smallest number that is not identically zero =
    2 ** (-128) - (2 ** (-127)) * 1/2

XUL = largest number that can be represented =
    2 ** (127) * XL

JL = largest integer that can be represented:
    2 ** (15) - 1; FL = 0; short integer
    2 ** (31) - 1; FL = 1; long integer

ABS (address) = absolute value of (address)

EXP (address) = biased exponent of (address)

.LT. = "less than"

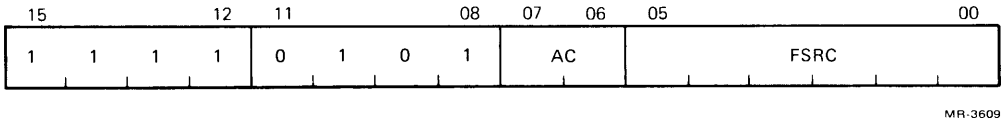
.LE. = "less than or equal to"

.GT. = "greater than"

.GE. = "greater than or equal to"

LSB = least significant bit

Load Floating/Double 172(AC+4)FSRC



Format: LDF FSRC,AC

Operation: AC <-- (FSRC)

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: Load single or double precision number into AC.

Interrupts: If FIUV is enabled, trap on -0 occurs before AC is loaded. However, the condition codes will reflect a fetch of -0 regardless of the FIUV bit.

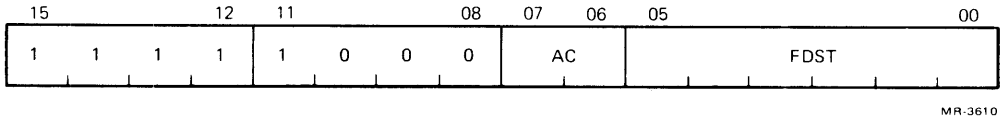
Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: These instructions permit use of -0 in a subsequent floating point instruction if FIUV is not enabled and (FSRC) = -0.

STF/STD

Store Floating/Double 174(AC)FDST



Format: STF AC,FDST

Operation: (FDST) <-- AC

Condition Codes:
FC <-- FC
FV <-- FV
FZ <-- FZ
FN <-- FN

Description: Store single or double precision number from AC.

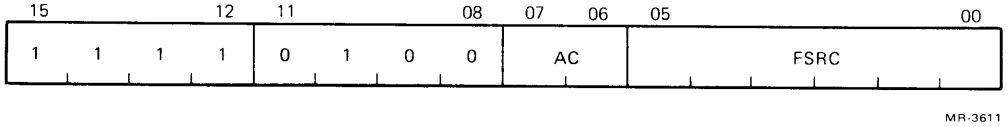
Interrupts: These instructions do not interrupt if FIUV is enabled, because the -0, if present, is in AC, not in memory.

Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: These instructions permit storage of a -0 in memory from AC. There are two conditions in which -0 can be stored in AC of the KEF11-A. One occurs when underflow or overflow is present and the corresponding interrupt is enabled. A second occurs when an LDF, LDD, LDCDF, or LDCFD instruction is executed and the FIUV bit is disabled.

Add Floating/Double 172(AC)FSRC



Format: ADDF FSRC,AC

Operation: Let SUM = (AC) + (FSRC).

If underflow occurs and FIU is not enabled, AC <-- exact 0.

If overflow occurs and FIV is not enabled, AC <-- exact 0.

For all others cases, AC <-- SUM.

Condition Codes:
FC <-- 0
FV <-- 1 if overflow occurs, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: Add the contents of FSRC to the contents of AC. The addition is carried out in single or double precision and is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

1. Overflow with interupt disabled

2. Underflow with interrupt disabled.

For these exceptional cases, an exact 0 is stored in AC.

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution.

If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

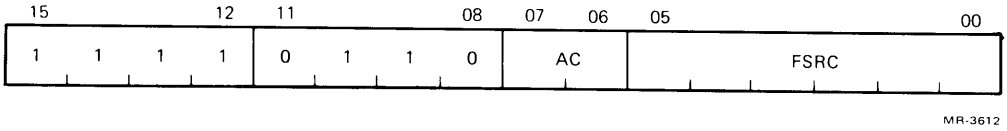
Accuracy: Errors due to overflow and underflow are described above. If neither occurs, then: for oppositely signed operands with exponent difference of 0 or 1, the answer returned is exact if a loss of significance of one or more bits can occur. Note that these are the only cases for which loss of significance of more than one bit can occur. For all other cases the result is inexact with error bounds of:

1. LSB in chopping mode with either single or double precision

2. 1/2 LSB in rounding mode with either single or double precision.

Special Comment: The undefined variable -0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

Subtract Floating/Double 173(AC)FSRC



Format: SUBF FSRC,AC

Operation: Let DIFF = (AC) - (FSRC).

If underflow occurs and FIU is not enabled, AC <-- exact 0.

If overflow occurs and FIV is not enabled, AC <-- exact 0.

For all others cases, AC <-- DIFF.

Condition Codes:
FC <-- 0
FV <-- 1 if overflow occurs, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: Subtract the contents of FSRC from the contents of AC. The subtraction is carried out in single or double precision and is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

1. Overflow with interrupt disabled

2. Underflow with interrupt disabled.

For these exceptional cases, an exact 0 is stored in AC.

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution.

If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

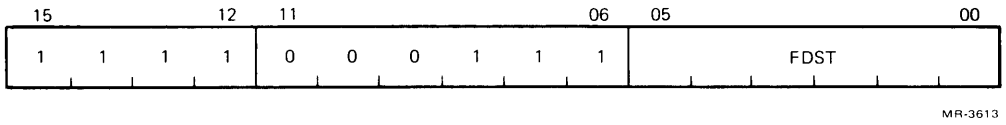
Accuracy: Errors due to overflow and underflow are described above. If neither occurs, then: for like signed operands with exponent difference of 0 or 1, the answer returned is exact if a loss of significance of one or more bits can occur. Note that these are the only cases for which loss of significance of more than one bit can occur. For all other cases the result is inexact with error bounds of:

1. LSB in chopping mode with either single or double precision

2. 1/2 LSB in rounding mode with either single or double precision.

Special Comment: The undefined variable -0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

Negate Floating/Double 1707 FDST



Format: NEGF FDST

Operation: (FDST) <-- (FDST) if (FDST) <> 0, else (FDST) <-- exact 0.

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (FDST) = 0, else FZ <-- 0
FN <-- 1 if (FDST) < 0, else FN <-- 0

Description: Negate single or double precision number, store result in same location (FDST).

Interrupts: If FIUV is enabled, trap on -0 occurs after execution.

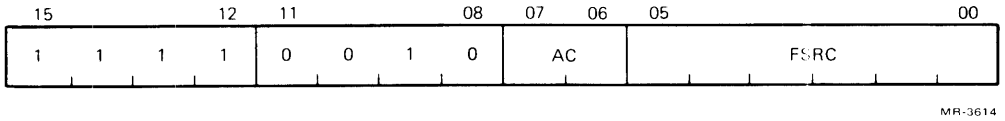
Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: If a -0 is present in memory and the FIUV bit is enabled, then the KEF11-A stores an exact 0 in memory. The condition codes reflect an exact 0 (FZ <-- 1).

MULF/MULD

Multiply Floating/Double 171(AC)FSRC



Format: MULF FSRC,AC

Operation: Let PROD = (AC) * (FSRC).

If underflow occurs and FIU is not enabled, AC <-- exact 0.

If overflow occurs and FIV is not enabled, AC <-- exact 0.

For all others cases, AC <-- PROD.

Condition Codes:
FC <-- 0
FV <-- 1 if overflow occurs, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: If the biased exponent of either operand is 0, (AC) <-- exact 0. For all other cases PROD is generated to 32 bits for floating mode and 64 bits for double mode. The product is rounded or chopped for FT = 0 and 1, respectively, and is stored in AC except for:

1. Overflow with interrupt disabled

2. Underflow with interrupt disabled.

For these exceptional cases, an exact 0 is stored in AC.

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution.

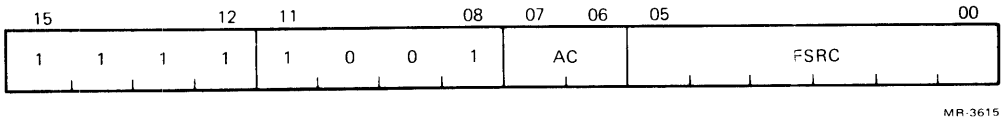
If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow and underflow are described above. If neither occurs, the error incurred is bounded by 1 LSB in chopping mode and 1/2 LSB in rounding mode.

Special Comment: The undefined variable -0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

DIVF/DIVD

Divide Floating/Double 174(AC+4)FSRC



Format: DIVF FSRC,AC

Operation: If EXP(FSRC) = 0, (AC) <-- (AC) and the instruction is aborted.

If EXP(AC) = 0, (AC) <-- exact 0.

For all other cases, let QUOT = (AC)/(FSRC).

If underflow occurs and FIU is not enabled, AC <-- exact 0.

If overflow occurs and FIV is not enabled, AC <-- exact 0.

For all others cases, AC <-- QUOT.

Condition Codes: FC <-- 0
FV <-- 1 if overflow occurs, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: If either operand has a biased exponent of 0, it is treated as an exact 0. For FSRC this would imply division by 0; in this case the instruction is aborted, the FEC register is set to 4 and an interrupt occurs. Otherwise the quotient is developed to single or double precision with two guard bits for correct rounding. The quotient is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in the AC except for:

1. Overflow with interrupt disabled

2. Underflow with interrupt disabled.

For these exceptional cases, an exact 0 is stored in AC.

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution.

If (FSRC) = 0, interrupt traps on attempt to divide by 0.

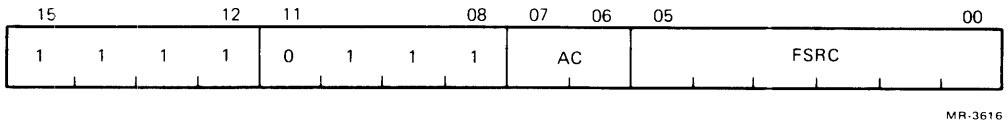
If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow and underflow are described above. If none of these occurs, the error in the quotient will be bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

Special Comment: The undefined variable -0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

CMPF/CMPD

Compare Floating/Double 173(AC+4)FSRC



Format: CMPF FSRC,AC

Operation: (FSRC) -- (AC)

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (FSRC) = 0, else FZ <-- 0
FN <-- 1 if (FSRC) < 0, else FN <-- 0

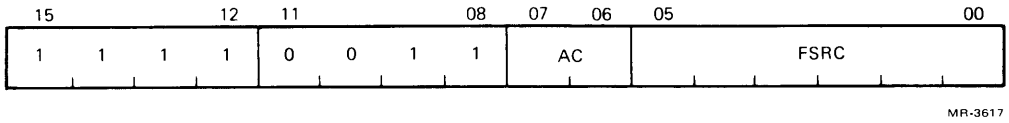
Description: Compare the contents of FSRC with the accumulator. Set the appropriate floating point condition codes. FSRC and the accumulator are left unchanged except as noted below.

Interrupts: If FIUV is enabled, trap on -0 occurs before execution.

Accuracy: These instructions are exact.

Special Comment: An operand which has a biased exponent of 0 is treated as if it were an exact 0. In this case where both operands are 0, the FPP will store an exact 0 in AC.

Multiply and Separate Integer
and Fraction Floating/Double 171(AC+4)FSRC



Format: MODF FSRC,AC

Description and Operation: This instruction generates the product of its two floating point operands, separates the product into integer and fractional parts and then stores one or both parts as floating point numbers.

Let PROD = (AC) * (FSRC) so that in

Floating point: ABS(PROD) = (2 ** K) * f

where

1/2 .LE. f .LT. 1 and

EXP(PROD) = (200 + K) octal

Fixed point binary: PROD = N + g with

n = INT(PROD) = the integer part of PROD and

g = PROD - INT(PROD) = the fractional part of PROD with 0 .LE. F .LT. 1.

Both N and f have the same sign as PROD. They are returned as follows:

If AC is an even-numbered accumulator (0 or 2), N is stored in AC+1 (1 or 3), and f is stored in AC.

If AC is an odd-numbered accumulator, N is not stored and g is stored in AC.

The two statements above can be combined as follows:
N is returned to ACv1 and g is returned to AC, where v means OR.

Five special cases occur, as indicated in the

following formal description with L = 24 for floating mode and L = 56 for double mode.

1. If PROD overflows and FIV is enabled, ACvl <-- N, chopped to L bits, AC <-- exact 0

   Note that EXP(N) is too small by 400_8 and that -0 can get stored in ACvl.

   If FIV is not enabled, ACvl <-- exact 0, AC <-- exact 0, and -0 will never be stored.

2. If 2 ** L .LE. ABS(PROD) and no overflow, ACvl <-- N, chopped to L bits, AC <-- exact 0

   The sign and EXP of N are correct, but low-order bit information, such as parity, is lost.

3. If 1 .LE. ABS(PROD) .LT. 2 ** L, ACvl <-- N, AC <-- g

   The integer part N is exact. The fractional part g is normalized, and chopped or rounded in accordance with FT. Rounding may cause a return of +/-unity for the fractional part. For L = 24, the error in g is bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode. For L = 56, the error in g increases from the above limits as ABS(N) increases above 8 because only 64 bits of PROD are generated.

   If 2 ** p .LE. ABS(N) .LT. 2 ** (p+1), with p>7, the low order p-7 bits of g may be in error.

4. If ABS(PR)D) .LT. 1 and no underflow, ACvl <-- exact 0 and AC <-- g.

   There is no error in the integer part. The error in the fractional part is bounded by 1 LSB in chopping mode and 1/2 LSB in rounding mode. Rounding may cause a return of +/-unity for the fractional part.

5. If PROD underflows and FIU is enabled, ACvl <-- exact 0 and AC <-- g.

   Errors are as in case 4, except that EXP(AC) will be too large by 400_8 (if EXP = 0, it is correct). Interrupt will occur and -0 can be stored in AC.

If FIU is not enabled, ACvl <-- exact 0 and AC <-- exact 0.

For this case the error in the fractional part is less than 2 ** (-128).

Condition Codes:
FC <-- 0
FV <-- 1 if PROD overflows, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Interrupts: If FIUV is enabled, trap on -0 in FSRC occurs before execution.

Overflow and underflow are discussed above.

Accuracy: Discussed above.

Applications:

1. Binary to decimal conversion of a proper fraction. The following algorithm, using MOD, will generate decimal digits D(1), D(2) ... from left to right.

```
Initialize: I <-- 0;
            X <-- number to be converted;
            ABS(X) < 1;
While X <> 0 do
Begin PROD <-- X * 10;
  I <-- I + 1;
  D(I) <-- INT(PROD);
  X <-- PROD -- INT(PROD);
  End;
```

   This algorithm is exact. It is case 3 in the description because the number of nonvanishing bits in the fractional part of PROD never exceeds L, and hence neither chopping nor rounding can introduce error.

2. To reduce the argument of a trigonometric function.

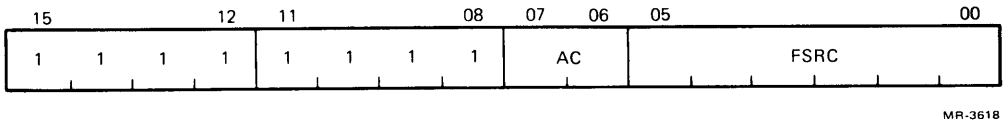
   ARG * 2/PI = N + g. The low two bits of N identify the quadrant, and g is the argument reduced to the first quadrant. The accuracy of N+g is limited to L bits because of the factor 2/PI. The accuracy of the reduced argument thus depends on the size of N.

3. To evaluate the exponential function e ** x, obtain x * (log e base 2) = N + g,
   then e ** x = (2 ** N) * (e ** (g * ln 2)).

   The reduced argument is g * ln2 < 1 and the

factor 2 ** N is an exact power of 2, which may be scaled in at the end via STEXP, ADD N to EXP and LDEXP. The accuracy of N+g is limited to L bits because of the factor (log e base 2). The accuracy of the reduced argument thus depends on the size of N.

Load and Convert from Double to Floating
and from Floating to Double 177(AC+4)FSRC



Format: LDCDF FSRC,AC

Operation: If EXP(FSRC) = 0, AC <-- exact 0.

If FD = 1, FT = 0, FIV = 0 and rounding causes overflow, AC <-- exact 0.

In all other cases, AC <-- Cxy(FSRC), where Cxy specifies conversion from floating mode x to floating mode y.

x = D, y = F if FD = 0 (single) LDCDF
y = F, y = D if FD = 1 (double) LDCFD

Condition Codes: FC <-- 0
FV <-- 1 if conversion produces overflow, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: If the current mode is floating mode (FD = 0), the source is assumed to be a double precision number and is converted to single precision. If the floating chop bit (FT) is set, the number is chopped, otherwise the number is rounded.

If the current mode is double mode (FD = 1), the source is assumed to be a single precision number and is loaded left-justified in AC. The lower half of AC is cleared.

Interrupts: If FIUV is enabled, trap on -0 occurs before execution. However, the condition codes will reflect a fetch of -0 regardless of the FIUV bit.

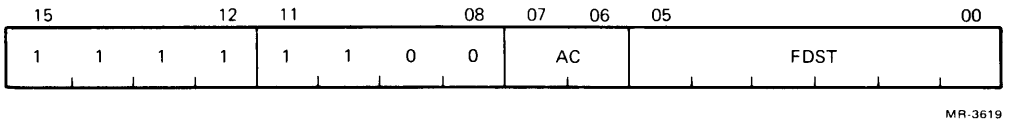
Overflow cannot occur for LDCFD.

A trap occurs if FIV is enabled, and if rounding with LDCDF causes overflow. AC <-- overflowed result. This result must be +0 or -0.

Underflow cannot occur.

Accuracy: LDCFD is an exact instruction. Except for overflow, described above, LDCDF incurs an error bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

Store and Convert from Floating to Double
and from Double to Floating 176(AC)FDST



Format: STCFD AC,FDST

Operation: If (AC) = 0, (FDST) <-- exact 0.

If FD = 1, FT = 0, FIV = 0 and rounding causes overflow, (FDST) <-- exact 0.

In all other cases, (FDST) <-- Cxy(AC), where Cxy specifies conversion from floating mode x to floating mode y.

x = F, y = D if FD = 0 (single) STCFD
x = D, y = F if FD = 1 (double) STCDF

Condition Codes: FC <-- 0
FV <-- 1 if conversion produces overflow, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: If the current mode is single precision, the accumulator is stored left-justified in FDST and the lower half is cleared.

If the current mode is double precision, the contents of the accumulator are converted to single precision, chopped or rounded depending on the state of FT, and stored in FDST.

Interrupts: Trap on -0 will not occur even if FIUV is enabled because FSRC is an accumulator.

Underflow cannot occur.

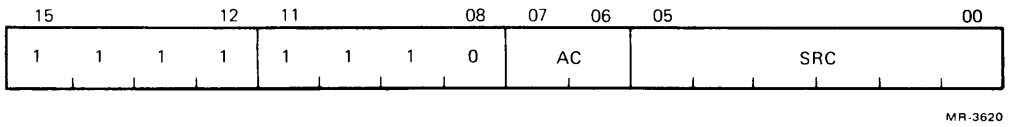
Overflow cannot occur for STCFD.

A trap occurs if FIV is enabled, and if rounding with STCDF causes overflow. (FDST) <-- overflowed result. This must be +0 or -0.

Accuracy: STCFD is an exact instruction. Except for

overflow, described above, STCDF incurs an error bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

Load and Convert Integer or Long Integer
to Floating or Double Precision 177(AC)SRC



Format: LDCIF SRC,AC

Operation: AC <-- Cjx(SRC), where Cjx specifies conversion from integer mode j to floating mode y.

j = I if FL = 0, j = L if FL = 1
x = F if FD = 0, x = D if FD = 1

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (Ac) < 0, else FN <-- 0

Description: Conversion is performed on the contents of SRC from a 2's complement integer with precision j to a floating point number of precision x. Note that j and x are determined by the state of the mode bits FL and FD.

If a 32-bit integer is specified (L mode) and (SRC) has an addressing mode of 0 or immediate addressing mode is specified, the 16 bits of the source register are left-justified and the remaining 16 bits loaded with 0s before conversion.

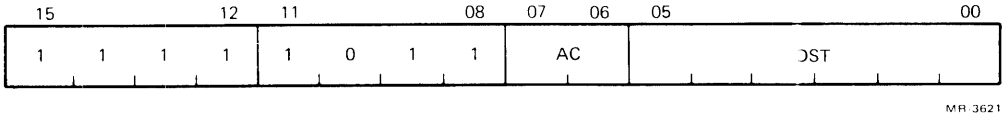
In the case of LDCLF, the fractional part of the floating point representation is chopped or rounded to 24 bits for FT = 1 and 0 respectively.

Interrupts: None; SRC is not floating point, so trap on -0 cannot occur.

Accuracy: LDCIF, LDCID, and LDCLD are exact instructions. The error incurred by LDCLF is bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

STCFI/STCFL/STCDI/STCDL

Store and Convert from Floating or Double
to Integer or Long Integer 175(AC+4)DST



Format: STCFI AC,DST

Operation: (DST) <-- Cxj(AC) if -JL-1 < Cxj(AC) < JL+1, else (DST) <-- 0, where Cjx specifies conversion from floating mode x to integer mode j.

```
j = I if FL = 0, j = L if FL = 1
x = F if FD = 0, x = D if FD = 1
```

JL is the largest integer.

```
2 ** 15 - 1 for FL = 0
2 ** 32 - 1 for FL = 1
```

Condition Codes:
C, FC <-- 0 if -JL-1 < Cxj(AC) < JL+1, else C, FC <-- 1
V, FV <-- 0
Z, FZ <-- 1 if (DST) = 0, else Z, FZ <-- 0
N, FN <-- 1 if (DST) < 0, else N, FN <-- 0

Description: Conversion is performed from a floating point representation of the data in the accumulator to an integer representation.

If the conversion is to a 32-bit word (L mode) and an addressing mode of 0 or immediate addressing mode is specified, only the most significant 16 bits are stored in the destination register.

If the operation is out of the integer range selected by FL, FC is set to 1 and the contents of the DST are set to 0.

Numbers to be converted are always chopped (rather than rounded) before conversion. This is true even when the chop mode bit FT is cleared in the FPS register.

Interrupts: These instructions do not interrupt if FIUV is enabled, because the -0, if present, is in AC,

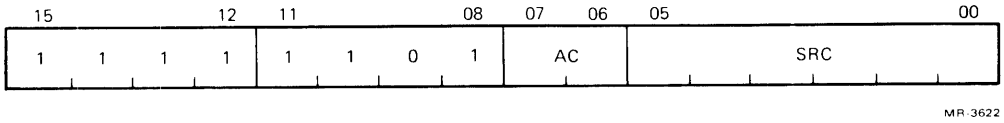
not in memory.

If FIC is enabled, trap on conversion failure will occur.

Accuracy: These instructions store the integer part of the floating point operand, which may not be the integer most closely approximating the operand. They are exact if the integer part is within the range implied by FL.

LDEXP

Load Exponent 176(AC+4)SRC



Format: LDEXP SRC,AR

Operation: Note: 177 and 200, appearing below, are octal numbers.

If -200 < SRC < 200, EXP(AC) <-- SRC + 200 and the rest of AC is unchanged.

If (SRC) > 177 and FIV is enabled, EXP(AC) <-- [(SRC) + 200]<7:0>.

If (SRC) > 177 and FIV is disabled, AC <-- exact 0.

If <SRC) < -177 and FIU is enabled, EXP(AC) <-- [(SRC) + 200]<7:0>.

If (SRC) < -177 and FIU is disabled, AC <-- exact 0.

Condition Codes:
FC <-- 0
FV <-- 1 if (SRC) > 177, else FV <-- 0
FZ <-- 1 if (AC) = 0, else FZ <-- 0
FN <-- 1 if (AC) < 0, else FN <-- 0

Description: Change AC so that its unbiased exponent = (SRC). That is, convert (SRC) from 2's complement to excess 200 notation and insert it in the EXP field of AC. This is a meaningful operation only if ABS(SRC) LE 177.

If SRC > 177, the result is treated as overflow. If SRC < -177, the result is treated as underflow. Note that the KEF11-A does not treat these abnormal conditions the same as the FP11C and FP11B but it does treat them the same as the FP11A and FP11E.

Interrupts: No trap on -0 in AC occurs, even if FIUV is enabled.

If SRC > 177 and FIV is enabled, trap on

overflow will occur.

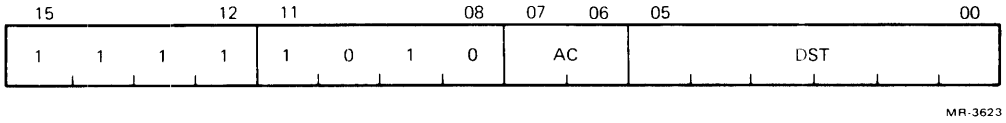
If SRC < -177 and FIU is enabled, trap on underflow will occur.

Accuracy: Errors due to overflow and underflow are described above. If EXP(AC) = 0 and (SRC) <> -200, AC changes from a floating point number treated as 0 by all floating arithmetic operations to a non-0 number. This is because the insertion of the "hidden" bit in the microcode implementation of arithmetic instructions is triggered by a nonvanishing value of EXP.

For all other cases, LDEXP implements exactly the transformation of a floating point number (2 ** K) * f into (2 ** (SRC)) * f where 1/2 .LE. ABS(f) .LT. 1.

STEXP

Store Exponent 175(AC)DST



Format: STEXP AC,DST

Operation: (DST) <-- EXP(AC) -200_8

Condition Codes:
C, FC <-- 0
V, FV <-- 0
Z, FZ <-- 1 if (DST) = 0, else Z, FZ <-- 0
N, FN <-- 1 if (DST) < 0, else N, FN <-- 0

Description: Convert AC's exponent from excess 200 notation to 2's complement and store the result in DST.

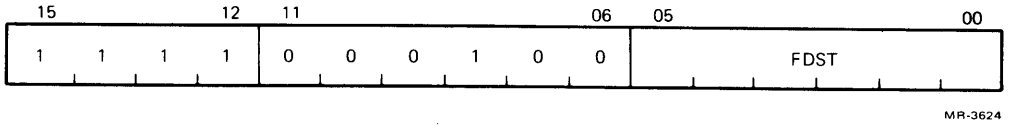
Interrupts: This instruction will not trap on -0.

Overflow and underflow cannot occur.

Accuracy: This instruction is always exact.

Clear Floating/Double

1704 FDST



Format: CLRF FDST

Operation: (FDST) <-- exact 0

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1
FN <-- 0

Description: Set FDST to 0. Set FZ condition code and clear other condition code bits.

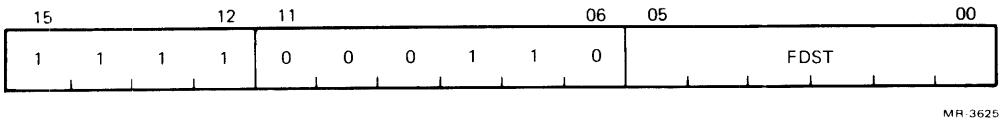
Interrupts: No interrupts will occur.

Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

ABSF/ABSD

Make Absolute Floating/Double 1706 FDST



Format: ABSF FDST

Operation: If (FDST) < 0, (FDST) <-- -(FDST).

If EXP(FDST) = 0, (FDST) <-- exact 0.

For all other cases, (FDST) <-- (FDST).

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (FDST) = 0, else FZ <-- 0
FN <-- 0

Description: Set the contents of FDST to its absolute value.

Interrupts: If FIUV is enabled, trap on -0 occurs after execution.

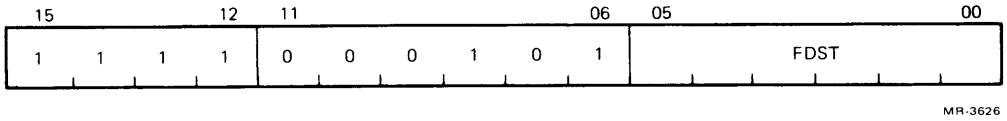
Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: If a -0 is present in memory and the FIUV bit is enabled, then an exact 0 is stored in memory. The condition codes reflect an exact 0 (FZ <-- 1).

Test Floating/Double

1705 FDST



Format: TSTF FDST

Operation: (FDST)

Condition Codes:
FC <-- 0
FV <-- 0
FZ <-- 1 if (FDST) = 0, else FZ <-- 0
FN <-- 1 if (FDST) < 0, else FN <-- 0.

Description: Set the FPP condition codes according to the contents of FDST.

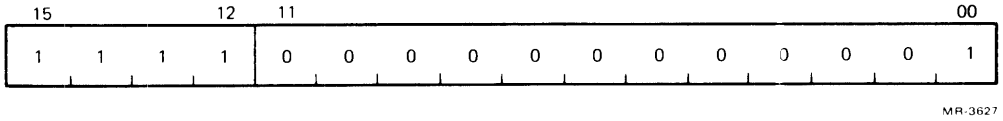
Interrupts: If FIUV is set, trap on -0 occurs after execution.

Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

SETF

Set Floating Mode 170001

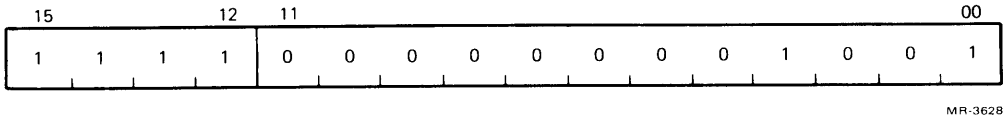


Format: SETF

Operation: FD <-- 0

Description: Set the FPP in single precision mode.

Set Floating Double Mode 170011



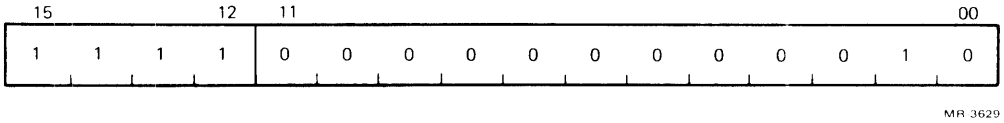
Format: SETD

Operation: FD <-- 1

Description: Set the FPP in double precision mode.

SETI

Set Integer Mode 177002



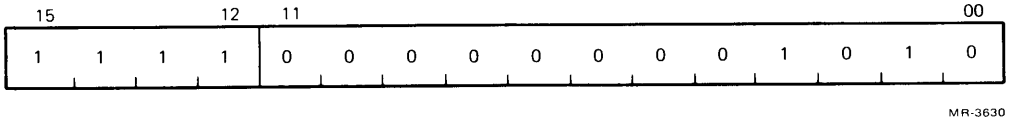
Format: SETI

Operation: FL <-- 0

Description: Set the FPP for Short Integer Data.

Set Long Interger Mode

177012



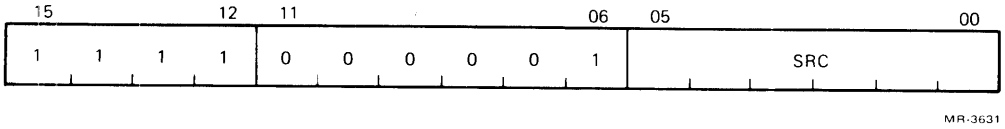
Format: SETL

Operation: FL <-- 1

Description: Set the FPP for long integer data.

LDFPS

Load FPP's Program Status 1701 SRC



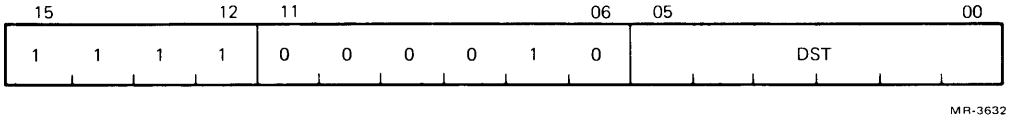
Format: LDFPS SRC

Operation: FPS <-- (SRC)

Description: Load FPP's status register from SRC.

Special Comment: The user is cautioned not to use bits 13, 12, and 4 for his own purposes, since these bits are not recoverable by the STFPS instruction.

Store FPP's Program Status 1702 DST



Format: STFPS DST

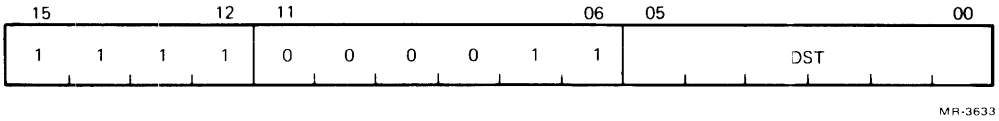
Operation: (DST) <-- FPS

Description: Store FPP's status register in DST.

Special Comment: Bits 13, 12, and 4 are loaded with 0. All other bits are the corresponding bits in the FPS.

STST

Store FPP's Status 1703 DST



Format: STST DST

Operation: (DST) <-- FEC
(DST + 2) <-- FEA

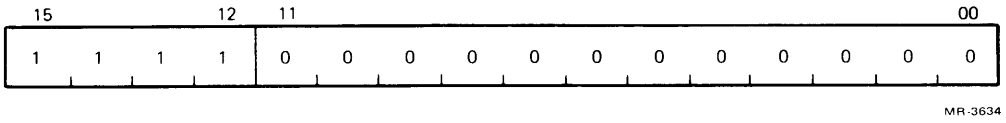
Description: Store the FEC and FEA in DST and DST+2.

NOTES

1. If the destination mode specifies a general register or immediate addressing, only the FEC is saved.

2. The information in these registers is current only if the most recently executed floating point instruction caused a floating point exception.

Copy Floating Condition Codes 170000



Format: CFCC

Operation:
C <-- FC
V <-- FV
Z <-- FZ
N <-- FN

Description: Copy the FPP condition codes into the CPU's condition codes.

CHAPTER 10
PROGRAMMING TECHNIQUES

10.1 INTRODUCTION

The KDF11-AA offers a great deal of programming flexibility and power. Utilizing the combination of the instruction set, the addressing modes, and the programming techniques makes it possible to develop new software or to utilize old programs effectively. The programming techniques in this chapter show the capabilities of the KDF11-AA. The techniques discussed are: postition-independent coding (PIC), stacks, subroutines, interrupts, reentrancy, coroutines, recursion, processor traps, programming peripherals, and conversion.

10.2 POSITION-INDEPENDENT CODE

The output of a MACRO-11 assembly is a relocatable object module. The task builder or linker binds one or more modules together to create an executable task image. Once built, a task can only be loaded and executed at the virtual address specified at link time. This is because the linker has had to modify some instructions to reflect the memory locations in which the program is to run. Such a body of code is considered position-dependent (i.e., dependent on the virtual addresses to which it was bound).

The KDF11-AA processor offers addressing modes that make it possible to write instructions that are not dependent on the virtual addresses to which they are bound. This type of code is termed position-independent and can be loaded and executed at any virtual address. Position-independent code can improve system efficiency, both in use of virtual address space and in conservation of physical memory.

In multiprogramming systems like RSX-11M, it is important that many tasks be able to share a single physical copy of common code; for example, a library routine. To make the optimum use of a task's virtual address space, shared code should be position-independent. Code that is not position-independent can also be shared, but it must appear in the same virtual locations in every task using it. This restricts the placement of such code by the task builder and can result in the loss of virtual addressing space.

10.2.1 Use of Addressing Modes in the Construction of Position-Independent Code

The construction of position-independent code is closely linked to the proper use of addressing modes. The remainder of this explanation assumes the reader is familiar with the addressing modes described in Chapter 6.

All addressing modes involving only register references are position-independent. These modes are as follows.

```
R         register mode
(R)       register deferred mode
(R)+      autoincrement mode
@*R)+     autoincrement deferred mode
-(R)      autodecrement mode
@-(R)     autodecrement deferred mode
```

When employing these addressing modes, the user is guaranteed position independence, providing the contents of the registers have been supplied independent of a particular virtual memory location.

The relative addressing modes are position-independent when a relocatable address is referenced from a relocatable instruction. These modes are as follows.

```
A         relative mode
@A        relative deferred mode
```

Relative modes are not position-independent when an absolute address (that is a nonrelocatable address) is referenced from a relocatable instruction. In this case, absolute addressing (i.e., @#A) may be employed to make the reference position independent.

Index modes can be either position-independent or position-dependent, according to their use in the program. These modes are as follows.

```
X(R)      index mode
@X(R)     index deferred mode
```

If the base, X, is an absolute value (e.g., a control block offset), the reference is position-independent. The following is an example.

```
    MOV       2(SP),R0      ;POSITION-INDEPENDENT
N=4
    MOV       N(SP),R0      ;POSITION-INDEPENDENT
```

If, however, X is a relocatable address, the reference is position-dependent, as the following example shows.

```
    CLR       ADDR(R1)      ;POSITION-DEPENDENT
```

Immediate mode can be either position-independent or not, according to its use. Immediate mode references are formatted as follows.

```
#N        immediate mode
```

When an absolute expression defines the value of N, the code is position-independent. When a relocatable expression defines N, the code is position-dependent. That is, immediate mode references are position-independent only when N is an absolute value.

Absolute mode addressing is position-independent only in those cases where an absolute virtual location is being referenced. Absolute mode addressing references are formatted as follows.

```
    @#A         absolute mode
```

An example of a position-independent absolute reference is a reference to the processor status word (PS) from a relocatable instruction, as in this example.

```
    MOV         @#PSW,R0      ;RETRIEVE STATUS AND PLACE IN REGISTER
```

10.2.2 Position-Dependent/Position-Independent Comparative Example

The RSX-11 library routine, PWRUP, is a FORTRAN-callable subroutine to establish or remove a user power failure asynchronous system trap (AST) entry point address. Imbedded within the routine is the actual AST entry point which saves all registers, effects a call to the user-specified entry point, restores all registers on return, and executes an AST exit directive. The following examples are excerpts from this routine. The first example has been modified to illustrate position-dependent references. The second example is the position-independent version.

Position-Dependent Code

```
PWRUP::
        CLR     -(SP)           ;ASSUME SUCCESS
        CALL    .X.PAA          ;PUSH (SAVE)
                                ;ARGUMENT ADDRESSES
                                ;ONTO STACK
        .WORD   1.,$PSW         ;CLEAR PSW, AND
                                ;SET R1=R2SP
        MOV     $OTSV,R4        ;GET OTS IMPURE
                                ;AREA POINTER
        MOV     (SP)+,R2        ;GET AST ENTRY
                                ;POINT ADDRESS
        BNE     10$             ;IF NONE SPECIFIED,
                                ;SPECIFY NO POWER
        CLR     -(SP)           ;RECOVERY AST SERVICE
        BR      20$             ;
10$:                            ;
        MOV     R2,F.PF(R4)     ;SET AST ENTRY POINT
        MOV     #BA,-(SP)       ;PUSH AST SERVICE
                                ;ADDRESS
```

```
20$:                                    ;
        CALL    .X.EXT                  ;ISSUE DIRECTIVE, EXIT.
        .BYTE   109.,2.                 ;
        .
        .
        .
BA:     MOV     R0,-(SP)                ;PUSH (SAVE) R0
        MOV     R1,-(SP)                ;PUSH (SAVE) R1
        MOV     R2,-(SP)                ;PUSH (SAVE) R2

Position-Independent Code
PWRUP::
        CLR     -(SP)                   ;ASSUME SUCCESS
        CALL    .X.PAA                  ;PUSH ARGUMENT
                                        ;ADDRESSES ONTO
                                        ;STACK
        .WORD   1.,$PSW                 ;CLEAR PSW, AND
                                        ;SET R1=R2-SP.
        MOV     @#$OTSV,R4              ;GET OTS IMPURE
                                        ;AREA POINTER
        MOV     (SP)+,R2                ;GET AST ENTRY
                                        ;POINT ADDRESS
        BNE     10$                     ;IF NONE SPECIFIED,
                                        ;SPECIFY NO POWER
        CLR     -(SP)                   ;RECOVERY AST SERVICE
        BR      20$
10$:                                    ;
        MOV     R2,F.PF(R4)             ;SET AST ENTRY POINT
        MOV     PC,-(SP)                ;PUSH CURRENT LOCATION
        ADD     #BA-.,(SP)              ;COMPUTE ACTUAL LOCATION
                                        ;OF AST
20$:
        CALL    .X.EXT                  ;ISSUE DIRECTIVE, EXIT.
        .BYTE   109.,2.
;
;ACTUAL AST SERVICE ROUTINE:
;
;       1) SAVE REGISTERS
;       2) EFFECT A CALL TO SPECIFIED SUBROUTINE
;       3) RESTORE REGISTERS
;       4) ISSUE AST EXIT DIRECTIVE
;
BA:     MOV     R0,-(SP)                ;PUSH (SAVE) R0
        MOV     R1,-(SP)                ;PUSH (SAVE) R1
        MOV     R2,-(SP)                ;PUSH (SAVE) R2
```

The position-dependent version of the subroutine contains a relative reference to an absolute symbol ($OTSV) and a literal reference to a relocatable symbol (BA). Both references are bound by the task builder to fixed memory locations. Therefore, the routine will not execute properly as part of a resident library if its location in virtual memory is not the same as the location specified at link time.

In the position-independent version, the reference to $OTSV has been changed to an absolute reference. In addition, the necessary code has been added to compute the virtual location of BA based upon the value of the program counter. In this case, the value is obtained by adding the value of the program counter to the fixed displacement between the current location and the specified symbol. Thus, execution of the modified routine is not affected by its location in the image's virtual address space.

10.3 STACKS

The stack is part of the basic design architecture of the KDF11-AA. It is an area of memory set aside by the programmer or by the operating system for temporary storage and linkage. It is handled on a LIFO (last-in/first-out) basis, where items are retrieved in the reverse of the order in which they were stored. A stack starts at the highest location reserved for it and expands linearly downward to a lower address as items are added to the stack.

It is not necessary to keep track of the actual locations into which data is being stacked. This is done automatically through a stack pointer. To keep track of the last item added to the stack, a general register is used to store the memory address of the last item in the stack. Any register except register 7 (the PC) may be used as a stack pointer under program control; however, instructions associated with subroutine linkage and interrupt service automatically use register 6 as a hardware stack pointer. For this reason, R6 is frequently referred to as the system SP. Stacks may be maintained in either full word or byte units. This is true for a stack pointed to by any register except R6, which must be organized in full word units only. Byte stacks (Figure 10-1) require instructions capable of operating on bytes rather than full words.

10.3.1 Pushing Onto a Stack

Items are added to a stack using the autodecrement addressing mode. Adding items to the stack is called PUSHing, and is accomplished by the following instructions.

```
MOV     Source,-(SP)      ;MOV contents of source word
                          ;onto the stack
                              or
MOVB    Source,-(SP)      ;MOVB source byte onto
                          ;the stack
```

Data is thus PUSHed onto the stack.

10.3.2 Popping From a Stack

Removing data from the stack is called a POP (popping from the stack). This operation is accomplished using the autoincrement mode.

```
MOV     (SP)+,Destination    ;MOV destination word
                             ;off the stack
                                   or
MOVB    (SP)+,Destination    ;MOVB destination byte
                             ;off the stack
```

After an item has been popped, its stack location is considered free and available for other use. The stack pointer points to the last-used location, implying that the next lower location is free. Thus, a stack may represent a pool of sharable temporary storage locations. (See Figure 10-2.)

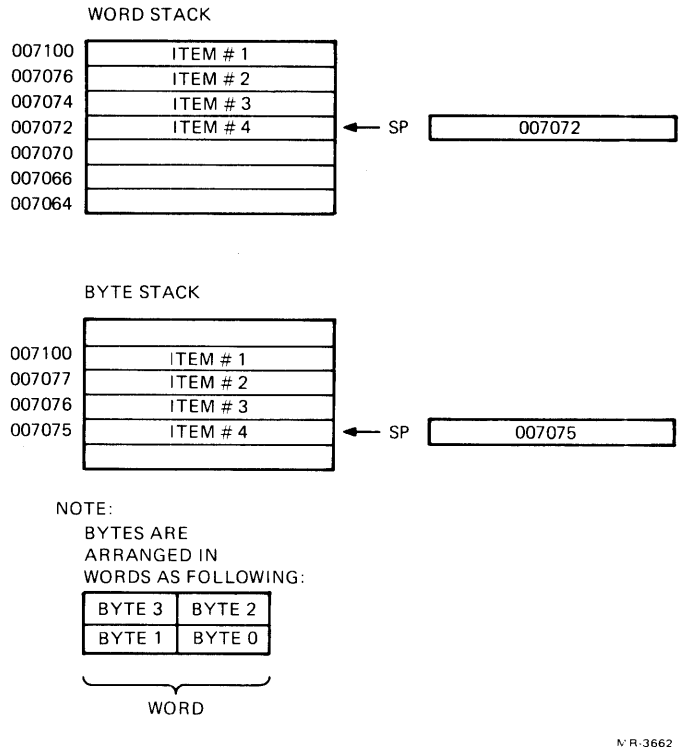


Figure 10-1 Word and Byte Stacks

10.3.3 Deleting Items From a Stack

The following techniques may be used to delete from a stack.

To delete one item use the following.

```
INC SP or TSTB(SP)+ for a byte stack
```

To delete two items use the following.

```
ADD#2,SP or TST(SP)+ for word stack
```

To delete fifty items from a word stack use the following.

```
ADD#100.,SP
```

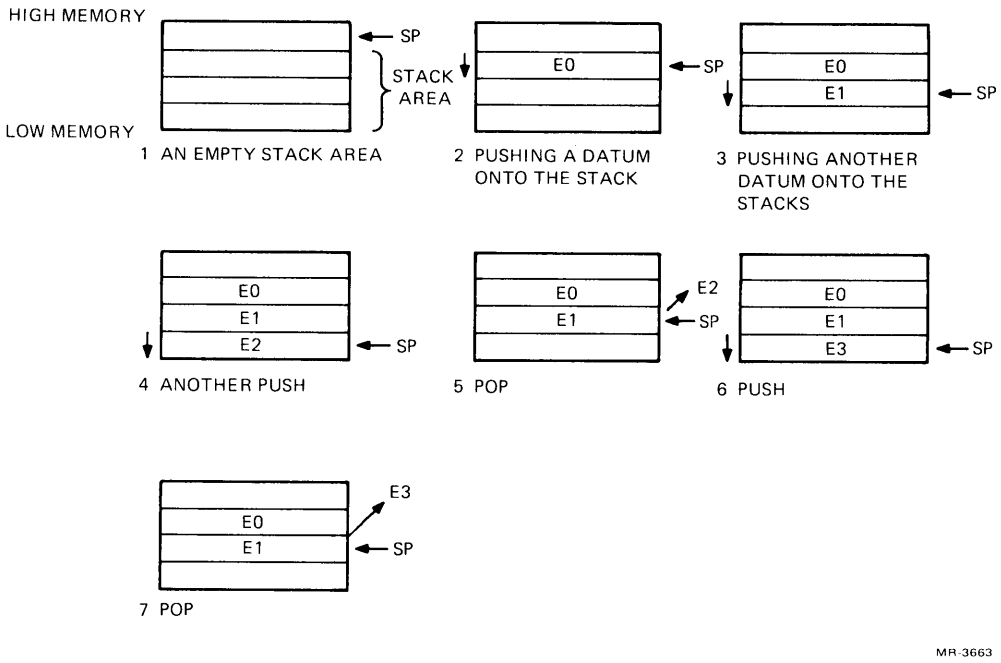


Figure 10-2 Illustration of Push and Pop Operations

10.3.4 Stack Uses

Stacks are used in the following ways.

1. Often one of the general purpose registers must be used in a subroutine or interrupt service routine and then returned to its original value. The stack can be used to store the contents of the registers involved.

2. The stack is used in storing linkage information between a subroutine and its calling program. The JSR instruction, used in calling a subroutine, requires the specification of a linkage register along with the entry address of the subroutine. The content of this linkage register is stored on the stack, so as not to be lost, and the return address is moved from the PC to the linkage register. This provides a pointer back to the calling program so that successive arguments may be transmitted easily to the subroutine.

When a subroutine returns, it is necessary to "clean up" the stack by eliminating or skipping over the subroutine arguments. One way this can be done is by insisting that the subroutine keep the number of arguments as its first stack item. Returns from subroutines then involve calculating the amount by which to reset the stack pointer, resetting the stack pointer, then storing the original contents of the register which were used as the copy of the stack pointer.

5. Stack storage is used in trap and interrupt linkage. The program counter and the processor status word of the executing program are pushed on the stack.

6. When the system stack is being used, nesting of subroutines, interrupts, and traps to any level can occur until the stack overflows its legal limits.

7. The stack method is also available for temporary storage of any kind of data. It may be used as a LIFO list for storing inputs, intermediate results, etc.

10.3.5 Stack Use Examples

As an example of stack use, consider this situation: a subroutine (SUBR) wants to use registers 1 and 2, but these registers must be returned to the calling program with their contents unchanged. The subroutine could be written as follows.

Not using the Stack:

```
Address     Octal Code      Assembler Syntax        Comments

076322      010167 SUBR:    MOV R1,TEMP1        ;save R1
076324      000074          *
076326      010267          MOV R2,TEMP2        ;save R2
076330      000072          *
.           .               .
.           .               .
.           .               .
076410      016701          MOV TEMP1,R1        ;restore R1
076412      000006          *
076414      0167902         MOV TEMP2,R2        ;restore R2
076416      000004          *
076420      000297          RTS PC
076422      000000          TEMP1:0
076424      000000          TEMP2:0
```

*Index constants

Using the Stack:
R3 has been previously set to point to the end of an unused block of memory.

Address	Octal Code	Assembler Syntax	Comments
010020	010143 SUBR:	MOV R1,-(R3)	;push R1
010022	010243	MOV R2,-(R3)	;push R2
.	.	.	
.	.	.	
.	.	.	
.	.	.	
010130	012302	MOV (R3)+,R2	;pop R2
010132	012301	MOV (R3)+,R1	;pop R1
010134	000207	RTS PC	

Note: In this case R3 was used as a stack pointer.

The second routine uses four fewer words of instruction code and two words of temporary "stack" storage. Another routine could use the same stack space at some later point. Thus, the ability to share temporary storage in the form of a stack is a way to save on memory use.

As another example of stack use, consider the task of managing an input buffer from a terminal. As characters come in, the user may wish to delete characters from the line; this is accomplished very easily by maintaing a byte stack containing the input characters. Whenever a backspace is received, a character is "popped" off the stack and eliminated from consideration. In this example, "popping" characters to be eliminated can be done by using either the MOVB (MOVE BYTE) or INC (INCREMENT) instructions.

Note that in this case the increment instruction (INC) is preferable to MOVB, since it accomplishes the task of eliminating the unwanted character from the stack by readjusting the stack pointer without the need for a destination location. Also, the stack pointer (SP) used in this example cannot be the system stack pointer (R6) because R6 may point only to word (even) locations. (See Figure 10-3.)

10.3.6 Subroutine Linkage

The contents of the linkage register are saved on the system stack when a JSR is executed. The effect is the same as if a MOV reg,-(R6) had been performed. Following the JSR instruction, the same register is loaded with the memory address (the contents of the current PC), and a jump is made to the entry location specified.

Figure 10-4 gives the before and after conditions when executing the subroutine instructions JSR R5, 1064.

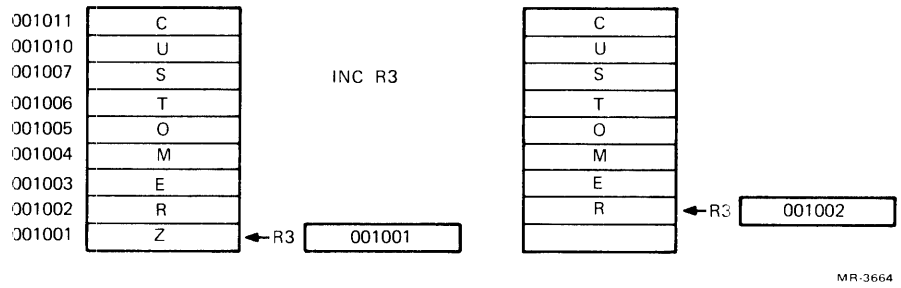


Figure 10-3 Byte Stack Used as a Character Buffer

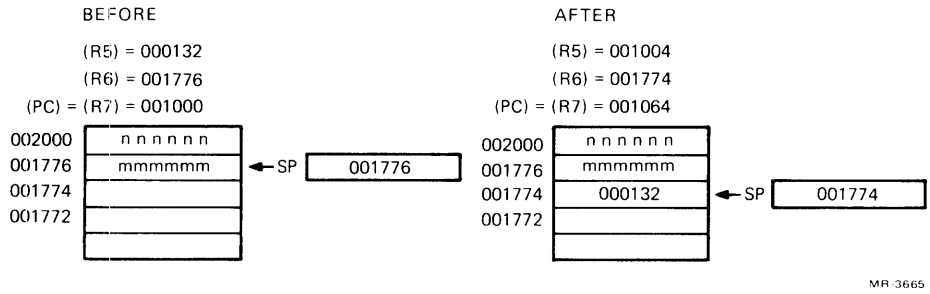


Figure 10-4 JSR Example

Because hardware already uses general-purpose register R6 to point to a stack for saving and restoring PC and PS (processor status word) information, it is convenient to use this same stack to save and restore intermediate results and to transmit arguments to and from subroutines. Using R6 this way permits nesting subroutines and interrupt service routines.

10.3.6.1 Return from a Subroutine - An RTS instruction provides for a return from the subroutine to the calling program. The RTS instruction must specify the same register as the one the JSR instruction used in the subroutine call. When the RTS is executed, the register specified is moved to the PC, and the top of the stack to be placed in the register specified. Thus, an RTS PC has the effect of returning to the address specified on the top of the stack.

10.3.6.2 Subroutine Advantages - There are several advantages to the subroutine calling procedure affected by the JSR instruction.

1. Arguments can be passed quickly between the calling program and the subroutine.

2. If there are no arguments, or the arguments are in a general register or on the stack, the JSR PC,DST mode can be used so that none of the general-purpose registers are used for linkage.

3. Many JSRs can be executed without the need to provide any saving procedure for the linkage information, since all linkage information is automatically pushed onto the stack in sequential order. Returns can be made by automatically popping this information from the stack in the order opposite to the JSRs.

Such linkage address bookkeeping is called automatic "nesting" of subroutine calls. This feature enables construction of fast, efficient linkages in a simple, flexible manner. It also permits a routine to call itself.

10.3.7 Interrupts

An interrupt is similar to a subroutine call, except that it is initiated by the hardware rather than by the software. An interrupt can occur after the execution of an instruction.

Interrupt-driven techniques are used to reduce CPU waiting time. In direct program data transfer, the CPU loops to check the state of the Done/Ready flag (bit 7) in the peripheral interface. Using interrupts, the CPU can handle other functions until the peripheral initiates service by setting the Done bit in its control status register. The CPU completes the instruction being executed and then acknowledges the interrupt, and vectors to an interrupt service routine. The service routine will transfer the data and may perform calculations with it. After the interrupt service routine has been completed, the computer resumes the program that was interrupted by the peripheral's high-priority request.

10.3.7.1 Interrupt Service Routines - With interrupt service routines, linkage information is passed so that a return to the main program can be made. More information is necessary for an interrupt sequence than for a subroutine call because of the random nature of interrupts. The complete machine state of the program immediately prior to the occurrence of the interrupt must be preserved in order to return to the program without any noticeable effects. This information is stored in the processor status word (PS). Upon interrupt, the contents of the program counter (PC) (address of next instruction) and the PS are automatically pushed onto the R6 system stack. The effect is the same as if:

```
MOV PS,-(SP)      ;Push PS
MOV PC,-(SP)      ;Push PC
```

had been executed.

The new contents of the PC and PS are loaded from two preassigned consecutive memory locations which are called "vector addresses."

The first word contains the interrupt service routine entry address (the address of the service routine program sequence), and the second word contains the new PS which will determine the machine status, including the operational mode and register set to be used by the interrupt service routine. The contents of the vector address are set under program control.

After the interrupt service routine has been completed, an RTI (return from interrupt) is performed. The top two words of the stack are automatically "popped" and placed in the PC and PS respectively, thus resuming the interrupted program.

Interrupt service programming is intimately involved with the concept of CPU and device priority levels.

10.3.7.2 Nesting - Interrupts can be nested in much the same manner that subroutines are nested. It is possible to nest any arbitrary mixture of subroutines and interrupts without any confusion. When the respective RTI and RTS instructions, are used, the proper returns are automatic. (See Figure 10-5.)

10.3.8 Reentrancy

Other advantages of the KDF11-AA stack organization occur in programming systems that handle several tasks. Multi-task program environments range from simple single-user applications which manage a mixture of I/O interrupt service and background data processing, as in RT-11, to large complex multi-programming systems that manage an intricate mixture of executive and multi-user programming situations, as in RSX-11. In all these situations, using the stack as a programming technique provides flexibility and time/memory economy by allowing many tasks to use a single copy of the same routine with a simple straightforward way of keeping track of complex program linkages.

The ability to share a single copy of a program among users or among tasks is called reentrancy. Reentrant program routines differ from ordinary subroutines in that it is not necessary for reentrant routines to finish processing a given task before they can be used by another task. Multiple tasks can exist at any time in varying stages of completion in the same routine. Thus the situation as shown in Figure 10-6 may occur.

10.3.8.1 Reentrant Code - Reentrant routines must be written in pure code, (any code that consists exclusively of instructions and constants). The value of using pure code whenever possible is that the resulting code has the following characteristics.

1. It is generally considered easier to debug.

2. It can be kept in read-only memory (is read-only protected).

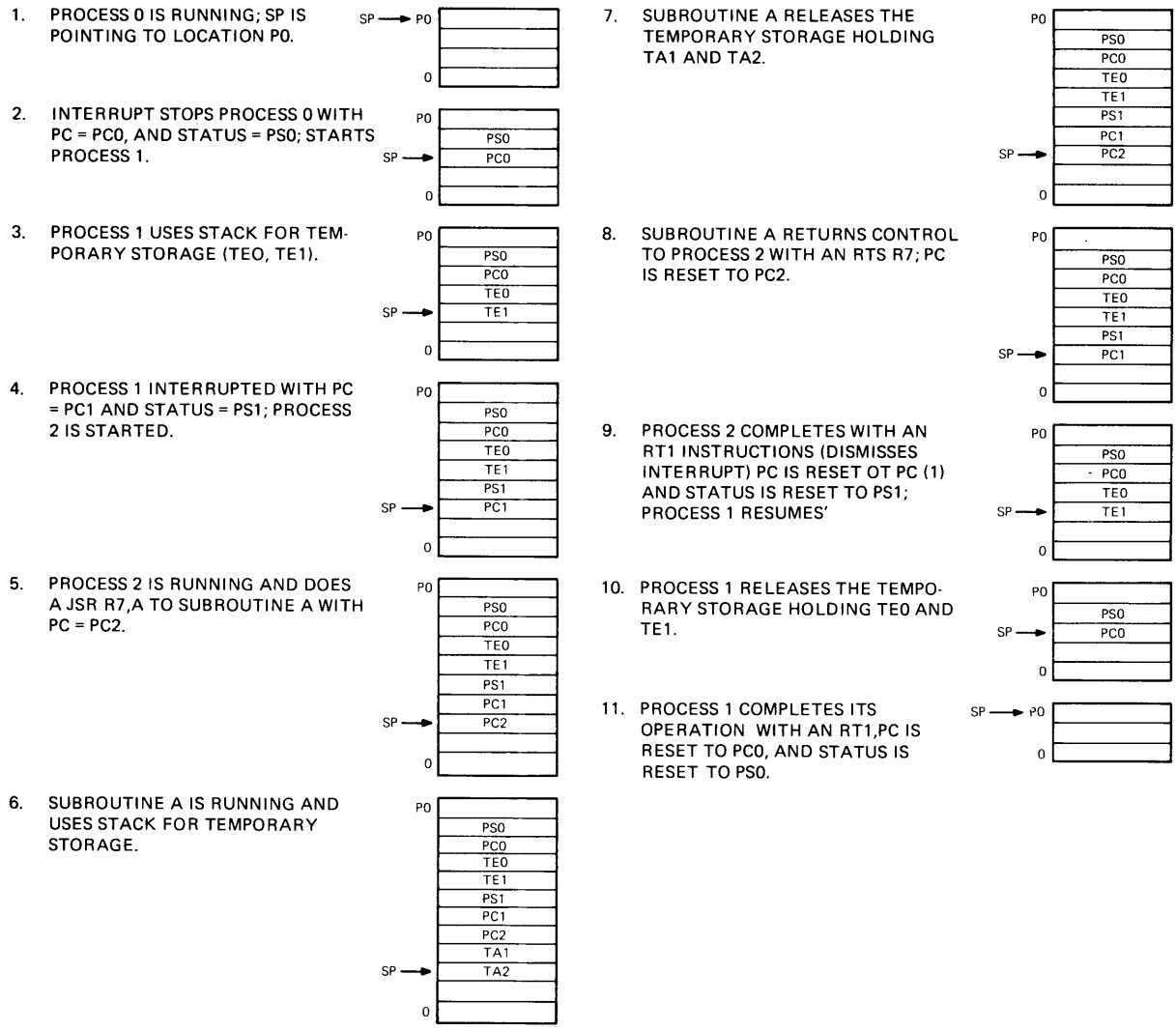


Figure 10-5 Nested Interrupt Service Routines and Subroutines

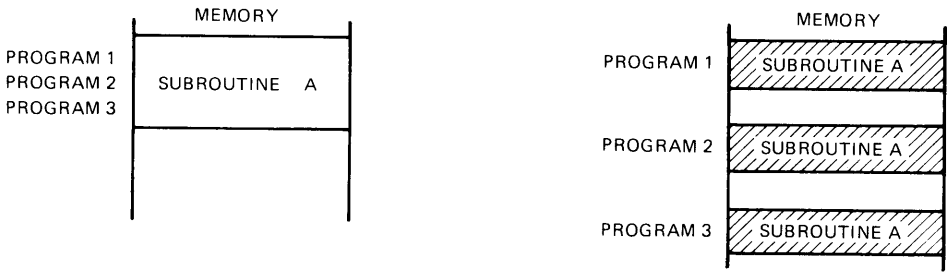


Figure 10-6 Reentrant Routines

Using reentrant code, control of a routine can be shared as follows. (See Figure 10-7.)

1. Task A requests processing by reentrant routine Q.

2. Task A temporarily relinquishes control of reentrant routine Q before it completes processing.

3. Task B starts processing the same copy of reentrant routine Q.

4. Task B completes processing by reentrant routine Q.

5. Task A regains use of reentrant routine Q and resumes where it stopped.

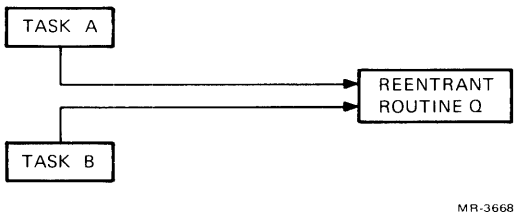


Figure 10-7 Sharing Control of a Routine

10.3.8.2 Writing Reentrant Code - In an operating system environment, when one task is executing and is interrupted to allow another task to run, a context switch occurs which causes the processor status word and current contents of the general-purpose registers (GPRs) to be saved and replaced by the appropriate values for the task being entered. Therefore, reentrant code should use the GPRs and the stack for any counters, pointers, or data that must be modified or manipulated in the routine.

The context switch occurs whenever a new task is allowed to execute. It causes all of the GPRs, the PS, and often other task-related information to be saved in an impure area, then reloads these registers and locations with the appropriate data for the task being entered. Notice that one consequence of this is that a new stack pointer value is loaded into R6, thereby causing a new area to be used as the stack when the second task is entered.

The following should be observed when writing reentrant code.

1. All data should be in or pointed to by one of the general purpose registers.

2. A stack can be used for temporary storage of data or pointers to impure areas within the task space. The pointer to such a stack would be stored in a GPR.

3. Parameter addresses should be used by indexing and indirect reference rather than by putting them into instructions within the code.

4. When temporary storage is accessed within the program, it should be by indexed addresses, which can be set by the calling task in order to handle any possible recursion.

10.3.9 Coroutines

In some programming situations it happens that several program segments or routines are highly interactive. Control is passed back and forth between the routines, each going through a period of suspension before being resumed. Since the routines maintain a symmetric relationship with each other, they are called coroutines.

Coroutines are two program sections, either subordinate to the other, which can call each other. The nature of the call is "I have processed all I can for now, so you can execute until you are ready to stop, then I will continue."

The coroutine call and return are identical, each being a jump to subroutine instruction with the destination address being on top of the stack and the PC serving as the linkage register, as follows.

```
JSR PC,@(R6)+
```

10.3.9.1 Coroutine Calls - The coding of coroutine calls is made simple by the stack feature. Initially, the entry address of the coroutine is placed on the stack and from that point the

```
JSR PC,@*R6)+
```

instruction is used for both the call and the return statements. The result of this JSR instruction is to exchange the contents of the PC and the top element of the stack, and so permit the two routines to swap control and resume operation where each was terminated by the previous swap. An example is shown in Figure 10-8.

Notice that the coroutine linkage cleans up the stack with each transfer of control.

ROUTINE A	STACK		ROUTINE B	COMMENTS
.				. LOC IS PUSHED
.				. ONTO THE STACK
.				. TO PREPARE FOR
MOV #LOC,-(SP)	LOC	←SP		THE COROUTINE
.				CALL.
.				
			LOC:	
JSR PC,@(SP)+	PC0	←SP		. WHEN THE CALL
(PC0)				. IS EXECUTED,
				. THE PC FROM
				ROUTINE A IS
				PUSHED ON THE
				STACK AND EXE-
				CUTION CONTIN-
				UES AT LOC.
			JSR PC,@(SP)+	ROUTINE B CAN
.	PC1	SP	(PC1)	RETURN CONTROL
.				. TO ROUTINE A
.				. BY ANOTHER
				. COROUTINE CALL.
				PC0 IS POPPED
				FROM THE STACK
				AND EXECUTION
				RESUMES IN
				ROUTINE A JUST
				AFTER THE CALL
				TO ROUTINE B,
				I.E., AT PC0.
				PC1 IS SAVED
				ON THE STACK
				FOR A LATER
				RETURN TO
				ROUTINE B.

MR-3669

Figure 10-8 Coroutine Example

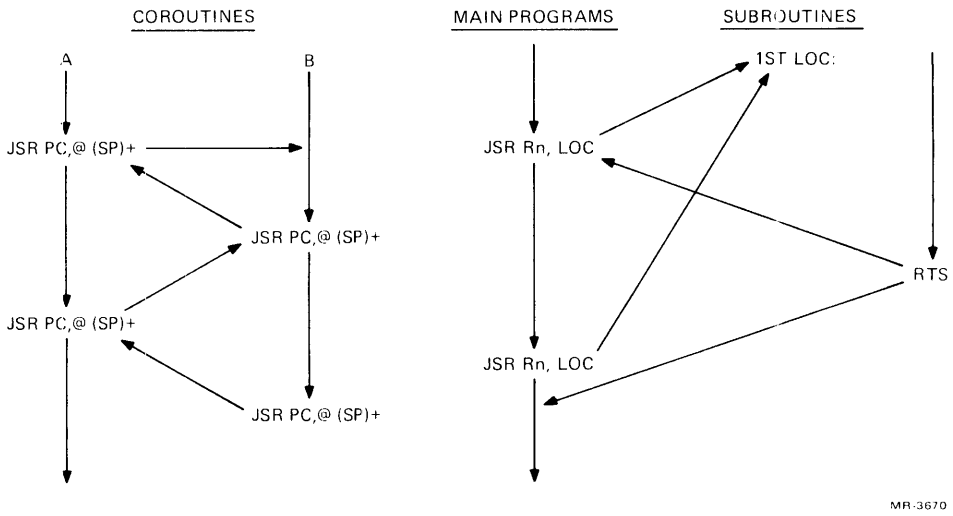


Figure 10-9 Coroutines vs. Subroutines

10.3.9.2 **Coroutines Versus Subroutines** - Coroutines can be compared to subroutines in the following ways.

1. A subroutine can be considered to be subordinate to the main or calling routine, but a coroutine is considered to be on the same level, as each coroutine calls the other when it has completed current processing.

2. A subroutine executes, when called, to the end of its code. When called again, the same code will execute before returning. A coroutine executes from the point after the last call of the other coroutine. Therefore, the same code will not be executed each time the coroutine is called. An example is shown in Figure 10-9.

3. The call and return statements for coroutines are the same, as follows.

```
JSR PC,@(SP)+
```

   This one instruction also cleans up the stack with each call.

   The last coroutine call will leave an address on the stack that must be popped if no further calls are to be made.

4. Each coroutine call returns to the coroutine code at the point after the last exit with no need for a specific entry point label, as would be required with subroutines.

10.3.9.3 **Using Coroutines** - Coroutines should be used in the following situations.

1. Coroutines should be used whenever two tasks must be coordinated in their execution without obscuring the basic structure of the program. For example, in decoding a line of assembly language code, the results at any one position might indicate the next process to be entered. Where a label is detected, it must be processed. If no label is present, the operator must be located, etc.

2. Coroutines should be employed to add clarity to the process being performed, to ease in the debugging phase, etc.

An assembler must perform a lexicographic scan of each assembly language statement during pass 1 of the assembly process. The various steps in such a scan should be separated from the main program flow to add to the program clarity and to aid in debugging by isolating many details. Subroutines would not be satisfactory

here, as too much information would have to be passed to the subroutine each time it was called. This subroutine would be too isolated. Coroutines could be effectively used here with one routine being the assembly-pass-one routine and the other extracting one item at a time from the current input line. Figure 10-10 illustrates this example.

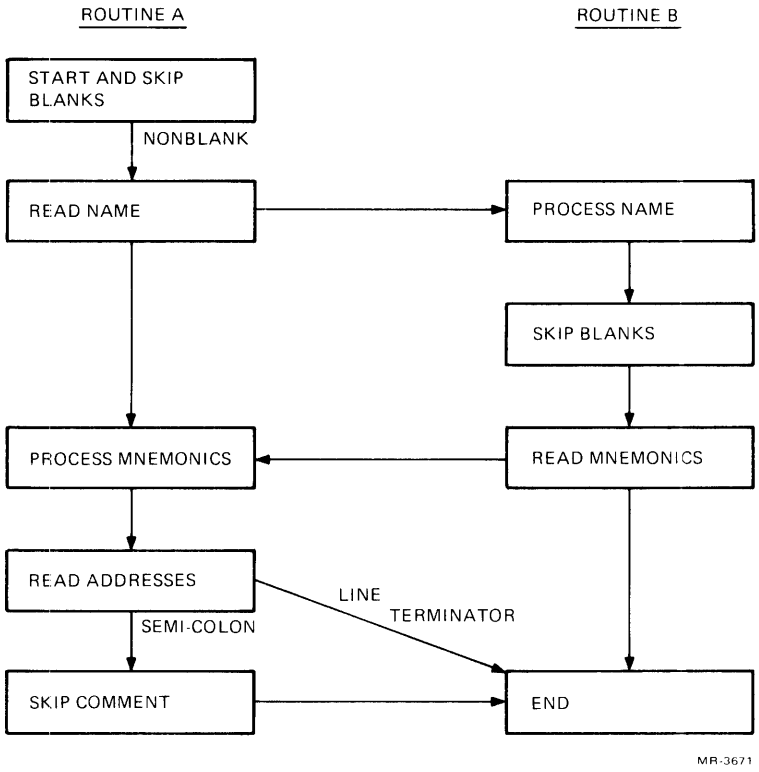


Figure 10-10 Coroutine Path

Coroutines can be utilized in I/O processing. The example above shows coroutines used in double-buffered I/O using IOX. The flow of events might be described as:

```
       Write 01
       Read I1          concurrently
       Process I2
then
       Write 02
       Read I2          concurrently.
       Process I1
```

Figure 10-11 illustrates a coroutine swapping interaction.

ROUTINE #1 IS OPERATING, IT THEN EXECUTES:

```
MOV #PC2,-(R6)
JSR PC,@(R6)+
```

WITH THE FOLLOWING RESULTS:

1. PC2 IS POPPED FROM THE STACK AND THE SP AUTOINCREMENTED.
2. SP IS AUTODECREMENTED AND THE OLD PC (I.E., PC1) IS PUSHED.
3. CONTROL IS TRANSFERRED TO THE LOCATION PC2 (I.E., ROUTINE #2).

ROUTINE #2 IS OPERATING, IT THEN EXECUTES:

```
JSR PC,@(R6)+
```

WITH THE RESULT THAT PC2 IS EXCHANGED FOR PC1 ON THE STACK AND CONTROL IS TRANSFERRED BACK TO ROUTINE #1.

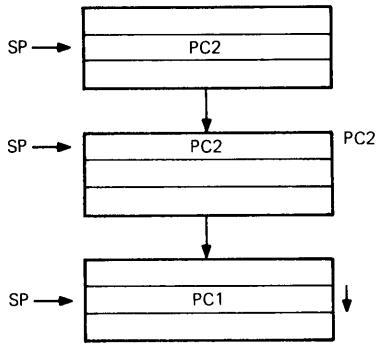


Figure 10-11 Coroutine Interaction

Routine #1 is operating, it then executes:

```
MOV #PC2,-(R6)
JSR PC,@(R6)+
```

with the following results.

1. PC2 is popped from the stack and the SP autoincremented.
2. SP is autodecremented and the old PC (i.e. PC1) is pushed.
3. Control is tranferred to the location PC2 (i.e. routine 2).

Routine 2 is operating, it then executes:

```
JSR PC,@(R6)+
```

with the result that PC2 is exchanged for PC1 on the stack and control is transferred back to routine 1.

10.3.10 Recursion

An interesting aspect of a stack facility, other than its providing for automatic handling of nested subroutines and interrupts, is that a program may call on itself as a subroutine just as it can call on any other routine. Each new call causes

the return linkage to be placed on the stack, which, as it is a last-in/first-out queue, sets up a natural unraveling to each routine just after the point of departure.

Typical flow for a recursive routine might be something like that shown in Figure 10-12.

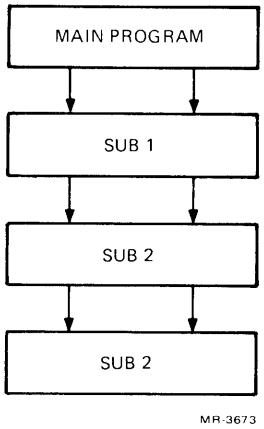


Figure 10-12 Recursive Routine Flow

The main program calls function 1, SUB 1, which calls function 2, SUB 2, which recurses once before returning.

Example:

```
DNCF:   ,
        ,
        ,
        BEQ 1$          ;TO EXIT RECURSIVE LOOP
        JSR R5,DNCF     ;RECURSE
1$      ,
        ,
        ,
        RTS R5          ;RETURN TO 1$ FOR
                        ;EACH CALL, THEN TO
                        ;MAIN PROGRAM
```

The routine DNCF calls itself until the variable tested becomes equal to 0, then it exits to 1$ where the RTS instruction is executed, returning to the 1$ once for each recursive call and one final time to return to the main program.

In general, recursion techniques will lead to slower programs than the corresponding interactive techniques, but the recursion will give shorter programs in memory space used. Both the brevity and clarity produced by recursion are important in assembly language programs.

Uses of Recursion - Recursion can be used in any routine in which the same process is required several times. For example, a function to be integrated may contain another function to be integrated, i.e., to solve for XM

where:

$$SM = 1 + F(X)$$

and:

$$F(X) = G(X)$$

Another use for a recursive function could be in calculating a factorial function because

$$FACT(N) = FACT(N-1)*N$$

Recursion should terminate when N = 1.

The macroprocessor within MACRO-11, for example, is itself recursive, as it can process nested macrodefinitions and calls. For example, within a macrodefinition, other macros can be called. When a macro call is encountered within definition, the processor must work recursively; i.e., to process one macro before it is finished with another, then to continue with the previous one. The stack is used for a separate storage area for the variables associated with each call to the procedure.

As long as nested definitions of macros are available, it is possible for a macro to call itself. However, unless conditionals are used to terminate this expansion, an infinite loop could be generated.

10.3.11 Processor Traps

Certain errors and programming conditions cause the KDF11-AA processor to enter the "service" state and trap to a fixed location. A trap is an interrupt generated by software. Pending conditions are arbitrated according to a priority. The following list describes the priority from highest to lowest.

Condition	Description
Memory Management Violation* (MMUERR)	A memory management violation causes an abort and traps to location 250_8.
Timeout Error* (BUSERR)	No response from a bus device during a bus transaction causes an abort and traps to location 4_8.
Parity Error* (PARERR)	A parity error signal received by the processor during a bus transaction causes an abort and traps to location 114_8.

Trace (T) Bit*	If PSW bit 4 is set at the end instruction execution, the processor traps to location 14_8.
Stack Overflow* (STKOVF)	If the kernel stack pointer was pushed below 400_8 during an instruction execution, the processor traps to location 4_8 at the end of the instruction.
Power Fail* (PFAIL)	If bus signal Power OK (BPOKH) became negated during instruction execution, the processor traps to location 24_8 at the end of the instruction.
Interrupt Level 7(BIRQ7) (Maskable by PS<07:05>) Interrupt Level 6 (BIRQ6) (Maskable by PS<07:05>) Interrupt Level 5 (BIRQ5) (Maskable by PS<07:05>) Interrupt Level 4 (BIRQ4) (Maskable by PS<07:05>)	If device interrupt requests are asserted and PS<07:05> are properly set, the processor at the end of the present instruction execution will initiate an interrupt vector sequenced on the bus.
Halt line	If BHALTL bus signal is asserted during the service state, the processor will enter ODT mode.

PS<07:05>	Levels Inhibited
7	All
6	6, 5, 4
5	5, 4
4	4
0->3	None

*Nonmaskable-software cannot inhibit the condition. CTLERR, MMUERR, BUSERR, PARERR are mutally exclusive when the processor is executing a program.

10.3.11.1 Trap Instructions - Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters. When a trap occurs, the contents of the current program counter (PC) and program status word (PS) are pushed onto the processor stack and replaced by the contents of a 2-word trap vector containing a new PC and new PS. The return sequence from a trap involves executing an RTI or RTT instruction which restores the old PC and old PS by popping them from the stack. Trap vectors are located at permanently assigned fixed addresses.

The EMT (trap emulator) and TRAP instructions do not use the low-order byte of the word in their machine language representation. This allows user information to be transferred in the low-order byte. The new value of the PC loaded from the

vector address of the TRAP or EMT instructions is typically the starting address of a routine to access and interpret this information. Such a routine is called a trap handler.

The trap handler must accomplish several tasks. It must save and restore all necessary GPRs, interpret the low byte of the trap instruction and call the indicated routine, serve as an interface between the calling program and this routine by handling any data that needs to be passed between them, and, finally, cause the return to the main routine.

The trap handler can be useful as a patching technique. Jumping out to a patch area is often difficult because a 2-word jump must be performed. However, the 1-word TRAP instruction may be used to dispatch to patch areas. A sufficient number of slots for patching should first be reserved in the dispatch table of the trap handler. The jump can then be accomplished by placing the address of the patch area into the table and inserting the proper TRAP instruction where the patch is to be made.

10.3.11.2 Use of Macro Calls - The trap handler can be used in a program to dispatch execution to any one of several routines. Macros may be defined to cause the proper expansion of a call to one of these routines, as in this example.

```
.MACRO SUB2 ARG
MOV ARG, R0
TRAP +1
.ENDM
```

When expanded, this macro sets up the one argument required by the routine in R0 and then causes the trap instruction with the number 1 in the lower byte. The trap handler should be written so that it recognizes a 1 as a call to SUB2. Notice that ARG here is being transmitted to SUB2 from the calling program. It may be data required by the routine or it may be a pointer to a longer list of arguments.

In an operating system environment like RT-11, the EMT instruction is used to call system or monitor routines from a user program. The monitor of an operating system necessarily contains coding for many functions, such as I/O, file manipulation, etc. This coding is made accessible to the program through a series of macro calls which expand into EMT instructions with low bytes, indicating the desired routine or group of routines to which the desired routine belongs. Often a GPR is designated to be used to pass an identification code to further indicate to the trap handler which routine is desired. For example, the macro expansion for a resume execution command in RT-11 is as follows.

```
.MACRO .RSUM
CM3, 2.
.ENDM
```

CM3 is defined as follows.

```
        .MACRO CM3 CHAN, CODE
        MOV #CODE *400,R0
.IIF NB CHAN,BISB CHAN,R0
        EMT 374
        .ENDM
```

Note that the EMT low byte is 374. This is interpreted by the EMT handler to indicate a group of routines. Then the contents of R0 (high byte) are tested by the handler to identify exactly which routine within the group is being requested, in this case routine number 2. (The CM3 call of the .RSUM is set up to pass the identification code.)

10.3.12 Conversion Routines

Almost all assembly language programs require the translation of data or results from one form to another. Coding that performs such a transformation will be called a conversion routine in this manual. Several commonly used conversion routines are included in the following pages.

Almost all assembly language programs involve some type of conversion routines: octal to ASCII, octal to decimal, and decimal to ASCII are a few of the most widely used.

Arithmetic multiply and divide routines are fundamental to many conversion routines.

Division is typically approched in one of two ways.

1. The division can be accomplished through a combination of rotates and subtractions.

   Examples:
   Assume the following code and register data; to make the example easier, also assume a 3-bit word.

```
DIV:    MOV #3,-(SP)        ;SET UP DIGIT COUNTER
        CLR -(SP)           ;CLEAR RESULT
1$      ASL (SP)
        ASL R1
        ROL R0
        CMP R0,R3
        BLT 2$
        SUB R3,R0           ;R0 CONTAINS REMAINDER
        INC (SP)            ;INCREMENT RESULT
2$      DEC 2 (SP)          ;DECREMENT COUNTER
        BNE $1
```

Therefore, to divide 7 by 2:

```
R0=000          remainder
R1=111          7-multiplicand
R3=010          2-multiplier
C bit=0

STACK
011             counter
000             quotient
```

Following through the coding, the quotient, remainder, and dividend all shift left, manipulating the most significant digit first, etc.

At the conclusion of the routine:

```
R0=001          remainder
R1=000
R3=010

STACK
000             counter
011             quotient
```

2. A second method of division occurs by repeated subtraction of the powers of the divisor, keeping a count of the number of subtractions at each level.

Example:
To divide 221_{10} by 10, first try to subtract powers of 10 until a nonnegative value is obtained, counting the number of subtractions of each power.

```
  221
-1000
```

negative so go to next lower power, count for $10^3 = 0$.

```
  221
 -100

  121    count for 10^2 = 1.
 -100

   21    count = 2
 -100
```

negative, so reduce power.
count for 10^2 = 2

```
 21
-10

 11    count for 10_1 = 1.

 11
-10

  1     count = 2
-10
```

negative, so count for 10^1 = 2.

No lower power, so remainder is 1.

Answer = 022, remainder 1.

Multiplication can be done through a combination of rotates and additions or through repetitive additions.

Example:
Assume the following code and a 3-bit word.

```
        CLR R0              ;HIGH HALF OF ANSWER
        MOV #3,CNT          ;SET UP COUNTER
        MOV R1,MULT;        ;MULTIPLICAND

        MORE:               ROR R2
                            BCC NOW
                            ADD MULT,R0 ;IF INDICATED,
ADD
                            ;MULTIPLICAND
        NOW;                ROR R0
                            R04 R1
                            DEC CNT
                            BNE MORE
        MULT:               0
        CNT:                0
```

The following conditions exist for 6 times 3:

R0 = 000 - high-order half of result
R1 = 110 - multiplicand
R3 = 011 - multiplier

After the routine is executed:

R0 = 010 - high-order half of result
R1 = 010 - low-order half of result
R2 = 100
CNT = 0
MULT = 110

Example:
Multiplication of R0 by 50_8(101000).

```
MUL50:      MOV R0,-(SP)
            ASL R0
            ASL R0
            ADD (SP)+,R0
            ASL R0
            ASL R0
            ASL R0
            RETURN
```

If R0 contains 7:

R0 = 111

After execution:

R0 = 100011000
($7*50_8 = 430_8$).

ASCII Conversions - The conversion of ASCII characters to the internal representation of a number as well as the conversion of an internal number to ASCII in I/O operations presents a challenge. The following routine takes the 16-bit word in R1 and stores the corresponding six ASCII characters in the buffer addressed by R2.

```
OUT:    MOV     #5,R0           ;LOOP COUNT
LOOP:   MOV     R1,-(SP)        ;COPY WORD INTO STACK
        BIC     #177770,@SP     ;ONE OCTAL VALUE
        ADD     #'0,@SP         ;CONVERT TO ASCII
        MOVB    (SP)+,-(R2)     ;STORE IN BUFFER
        ASR     R1              ;SHIFT
        ASR     R1              ;RIGHT
        ASR     R1              ;THREE
        DEC     R0              ;TEST IF DONE
        BNE     LOOP            ;NO, DO IT AGAIN
        BIC     #177776,R1      ;GET LAST BIT
        ADD     #'0,R1          ;CONVERT TO ASCII
        MOVB    R5,-(R2)        ;STORE IN BUFFER
        RTS     PC              ;DONE,RETURN
```

10.4 PROGRAMMING THE PROCESSOR STATUS WORD

The current processor status can be read and written using several programming techniques on the PS. The PS has an I/O address of 777776. The KDF11 and other PDP-11 processors implement this address, whereas LSI-11 and LSI-11/2 processors do not.

One technique is to use the I/O address as a source or destination address with any instruction.

```
CLR @#177776
MOV @#177776, R0
```

The first instruction clears the PS and the second instruction moves the contents of the PS to general register R0.

The PS explicit address (777776) can be accessed on a word or byte basis. The KDF11 will recognize the PS odd address (777777) and the access result will be identical to an odd memory address reference.

Another technique is to use the two dedicated PS instructions, MTPS and MFPS. These instructions only reference the even byte.

If memory management is enabled certain PS bits are protected. Refer to Paragraph 8.3.3.2 for more details.

10.5 PROGRAMMING PERIPHERALS

Programming of LSI-11 bus compatible modules (devices) is simple. A special class of instructions to deal with input/output operations is unnecessary. The bus structure permits a unified addressing structure in which control, status, and data registers for devices are directly addressed as memory locations. Therefore, all operations on these registers, such as tranferring information into or out of them or manipulating data within them, are performed by normal memory reference instructions.

The use of all memory reference instructions on device registers greatly increases the flexibility of input/output programming. For example, information in a device register can be compared directly with a value and a branch made on the result.

```
CMP RBUF,       #101
BEQ SERVICE
```

In this case, the program looks for 101 in the DLV11 receiver data buffer register (RBUF) and branches if it finds it. There is no need to transfer the information into an intermediate register for comparison.

When the character is of interest, a memory reference instruction can transfer the character into a user buffer in memory or to another peripheral device. The instruction:

```
MOV DRINBUF LOC
```

transfers a character from the DRV11 Data Input Buffer (DRINBUF) into a user-defined location.

All arithmetic operations can be performed on a peripheral device register. For example, the instruction ADD #10, DROUT BUF will add 10 to the DRV11's Output Buffer.

All read/write device registers can be treated as accumulators. There is no need to funnel all data transfers, arithmetic operations, and comparisons through a single or small number of accumulator registers.

10.6 PDP-11 PROGRAMMING EXAMPLES

The programming examples on the following pages show how the instruction set, the addressing modes, and the programming techniques can be used to solve some simple problems. The format used is either PAL-11 or MACRO-11.

```
Program Program
Address Contents Label  Op Code   Operand     Comments

                        ;PROGRAMMING EXAMPLE
                        ;SUBTRACT CONTENTS OF LOCS 700-710
                        ;FROM CONTENTS OF LOCS 1000-1010

        000000          R0=%0
        000001          R1-%1
        000002          R2=%2
        000003          R3=%3
        000004          R4=%4
        000005          R5=%5
        000006          SP=%6
        000007          PC=%7

        000500          .=500
000500  012706  START:  MOV       #.,SP       ;INIT STACK POINTER
        000500
000504  012701          MOV       #700,R1
        000700
000510  012702          MOV       #712,R2
        000712
000514  012703          MOV       #1000,R3
        001000
000520  012704          MOV       #1012,R4
        001012
000524  005000          CLR       R0
000526  005005          CLR       R5
```

```
000430  062105   SUM1:  ADD       (R1)+,R5     ;START ADDING
000532  020102          CMP       R1,R2        ;FINISHED ADDING?
000534  001375          BNE       SUM1         ;IF NOT BRANCH BACK
000536  062300   SUM2:  ADD       (R3)+,R0     ;START ADDING
000540  020304          CMP       R3,R4        ;FINISHED ADDING?
000542  001375          BNE       SUM2         ;IF NOT BRANCH BACK

000544  160500   DIFF:  SUB       R5,R0        ;SUBTRACT RESULTS

000546  000000          HALT                   ;THAT'S ALL FOLKS

        000700          =700
000700  000001          WORD 1,2,3,4,5
000702  000002
000704  000003
000706  000004
000710  000005

        001000          =1000
001000  000004          WORD 4,5,6,7,8
001002  000005
001004  000006
001006  000007
001010  000010

                                               A-30
        000500          END
```

```
        ;PROGRAM TO COUNT NEGATIVE NUMBERS
        ;IN A TABLE
        ;20. SIGNED WORDS
        ;BEGINNING AT LOC VALUES
        ;COUNT HOW MANY ARE NEGATIVE IN R0

        R0=%0
        R1=%1
        R2=%2
        SP=%6
        PC=%7

        .=500

START:  MOV#.,SP                ;SET UP STACK
        MOV #VALUE,R1           ;SET UP POINTER
        MOV #VALUES+40.,R2      ;SET UP COUNTER
        CLR R0
```

```
CHECK:    TST (R1)+                ;TEST NUMBER
          BPL NEXT                 ;POSITIVE?
          INC R0                   ;NO, INCREMENT
                                   ;COUNTER
NEXT:     CMP R1,R2                ;YES, FINISHED?
          BNE CHECK                ;NO, GO BACK
          HALT                     ;YES, STOP

VALUES:   0
          .END
```

```
          ;PROGRAM TO COUNT ABOVE AVERAGE QUIZ SCORES
          ;LIST OF 16. QUIZ SCORES
          ;BEGINNING AT LOC SCORES
          ;KNOWN AVERAGE IN LOC AVERAGE
          ;COUNT IN R0 SCORES ABOVE AVERAGE

          R0=%0
          R1=%1
          R2=%2
          R3=%3
          SP=%6
          PC=%7

          .=500

START:    MOV #.,SP                ;SET UP STACK
          MOV #16.,R1              ;SET UP COUNTER
          MOV #SCORES,R2           ;SET UP POINTER
          MOV #AVERAGE,R3
          CLR R0

CHECK:    CMP (R2)+,(R3)           ;COMPARE SCORE AND AVERAGE
          BLE NO                   ;LESS THAN OR EQUAL
                                   ;TO AVERAGE?
          INC R0                   ;NO, COUNT
NO:       DEC R1                   ;YES, DECREMENT COUNTER
          BNE CHECK                ;FINISHED? NO, CHECK
          HALT                     ;YES, STOP
AVERAGE:  65.

SCORES*   25.,70.,100.,60.,80.,80.,40.
          55.,75.,100.,65.,90.,70.,65.,70.

          .END
```

```
        ;PROGRAMMING EXAMPLE
        ;ACCEPT (IMMEDIATE ECHO) AND
        ;STORE 20. CHARS
        ;FROM THE KEYBOARD, OUTPUT CR & LF
        ;ECHO ENTIRE STRING FROM STORAGE

        R0=%0
        R1=%1
        SP=%6
        CR=15
        LF=12
        TKS=177560
        TKB=TKS+2
        TPS=TKB+2
        TPB=TPS+2

        .TITLE ECHO

        .=1000
START:  MOV     #.,SP           ;INITIALIZE STACK POINTER
        MOV     #SAVE+2,R0      ;SA OF BUFFER
                                ;BEYOND CR & LF
        MOV     #20.,R1         ;CHARACTER COUNT

IN:     TSTB    @#TKS           ;CHAR IN BUFFER?
        BPL     IN              ;IF NOT BRANCH BACK
                                ;AND WAIT
ECHO:   TSTB    @#TPS           ;CHECK TELEPRINTER
                                ;READY STATUS
        BPL     ECHO
        MOVB    @#TKB,@#TPB     ;ECHO CHARACTER
        MOVB    @#TKB,(R0)+     ;STORE CHARACTER AWAY
        DEC     R1
        BNE     IN              ;FINISHED INPUTTING?

        MOV     #SAVE,R0        ;SA OF BUFFER INCLUDING
                                ;CR & LF
        MOV     #22.,R1         ;COUNTER OF BUFFER
                                ;INCLUDING CR & LF

OUT:    TSTB    @#TPS           ;CHECK TELEPRINTER
                                ;READY STATUS
        BPL     OUT
        MOVB    (R0)+,@#TPB     ;OUTPUT CHARACTER
        DEC     R1
        BNE     OUT             ;FINISHED OUTPUTTING?
        HALT

SAVE:   .BYTE   CR,LF
        .=.+20,

        .END
```

```
          ;PROGRAMMING EXAMPLE
          ;SUBROUTINE TO INPUT TEN VALUES
INPUT:    MOV #BUFFER,R0            ;SET UP SA OF
                                    ;STORAGE BUFFER
          MOV #-10.,R1              ;SET UP COUNTER
IN:       TSTB @#TKS                ;TEST KYBD READY STATUS
          BPL IN
OUT:      TSTB @#TPS                ;TEST TTO READY STATUS
          BPL OUT
          MOVB @#TKB,@#TPB          ;ECHO CHARACTER
          MOVB @#TKB,(R0)+          ;STORE CHARACTER
          INC R1                    ;INC COUNTER
          BNE IN
          RTS PC                    ;EXIT
```

```
          ;PROGRAMMING EXAMPLE
          ;SUBROUTINE TO SORT TEN VALUES
SORT:     MOV #-10.,R4
NEXT:     MOV COUNT,R3
          MOV #BUFFER+9.,R0
          ADD R3,R0
          MOVB (R0)+,R1
LOOP:     CMPB (R0)+,R1
          BGE GT
LT:       MOVB -(R0),R2
          MOVB R1,(R0)+
          MOV R2,R1
GT:       INC R3
          BNE LOOP
INSERT:   MOVB R1,BUFFER+10.(R4)
          INC R4
          INC COUNT
          BNE NEXT
          MOV #-9.,COUNT            ;RESTORE LOCATION COUNT
          RTS PC                    ;EXIT
```

```
COUNT:    .WORD -9.
LINE1:    .ASCII/INPUT ANY TEN SINGLE-DIGIT VALUES (0-9); I'LL/
          .ASCII/SORT AND OUTPUT THEM IN/
LINE2:    .ASCII/SMALLEST TO LARGEST ORDER./
BUFFER:   .=.+10.
          .END INITSP               ;FINISHED!!!
```

```
;PROGRAMMING EXAMPLE
;SUBROUTINE EXAMPLE
;INPUT TEN VALUES, SORT, AND
;OUTPUT THEM IN SMALLEST TO LARGEST ORDER

R0=%0
R1=%1
R2=%2
R3=%3
R4=%4
R5=%5
SP=%6
PC=%7
TKS=177560
(address of terminal control status register)
TKB=TKS+2 - (terminal data buffer register)
TPS=TKB+2
(terminal output control and status registers)
TPB=TPS+2 - (terminal output data buffer)

.=3000

INITSP: MOV #.,SP               ;INITIALIZE STACK POINTER
        JSR PC,CRLF             ;GO TO CRLF SUBROUTINE
        JSR R5, OUTPUT          ;GOT TO OUTPUT SUBROUTINE
        LINE1                   ;SA OF LINE 1 BUFFER
        69.                     ;NUMBER OF OUTPUTS
        JSR PC,CRLF             ;GO TO CRLF SUBROUTINE
        JSR R5,OUTPUT           ;GO TO OUTPUT SUBROUTINE
        LINE2                   ;SA OF LINE 2 BUFFER
        26.                     ;NUMBER OF OUTPUTS
        JSR PC,CRLF             ;GO TO CRLF SUBROUTINE
        JSR PC,INPUT            ;GO TO INPUT SUBROUTINE
        JSR PC,SORT             ;GO TO SORT SUBROUTINE
        JSR PC,CRLF             ;GO TO CRLF SUBROUTINE
        JSR R5,OUTPUT           ;GO TO OUTPUT SUBROUTINE
        BUFFER                  ;INPUT BUFFER AREA
        10.                     ;NUMBER OF OUTPUTS
        JSR PC,CRLF
        HALT                    ;THE END!!!
```

```
          ;PROGRAMMING EXAMPLE
          ;SUBROUTINE TO OUTPUT A CR & LF
CRLF:     TSTB @#TPS              ;TEST TTO READY STATUS
          BPL CRLF
          MOVB #15,@#TPB          ;OUTPUT CARRIAGE RETURN
LNFD:     TSTB @#TPS              ;TEST TTO READY STATUS
          BPL LNFD
          MOVB #12,@#TPB          ;OUTPUT LINE FEED
          RTS PC                  ;EXIT
```

```
          ;SUBROUTINE TO OUTPUT A
          ;VARIABLE LENGTH MESSAGE
OUTPUT:   MOV (R5)+,R0            ;PICK UP SA OF DATA BLOCK
          MOV (R5)+,R1            ;PICK UP NUMBER OF OUTPUTS
          NEG R1                  ;NEGATE IT
AGAIN:    TSTB @#TPS              ;TEST TTO READY STATUS
          BPL AGAIN
          MOVB (R0)+,@#TPB        ;OUTPUT CHARACTER
          INC R1                  ;BUMP COUNTER
          BNE AGAIN
          RTS R5
```

10.7 LOOPING TECHNIQUES

Looping techniques are illustrated in the program segments below. The segments are used to clear a 50-word table.

1. AUTOINCREMENT (POINTER ADDRESS IN GPR)

```
                          R0 = %0
                          MOV #TBL,R0
          LOOP:           CLR (R0)+
                          CMP R0,#TBL+100.
                          BNE LOOP
```

2. AUTODECREMENT (POINTER AND LIMIT VALUES IN GPR)

```
                          R0=%0
                          R1=%1
                          MOV #TBL,R0
                          MOV #TBL+100.,R1
          LOOP:           CLR -- (R1)
                          CMP R1,R0
                          BNE LOOP
```

3. COUNTER (DECREMENTING A GPR CONTAINING COUNT)

```
                    R0=%0
                    R1=%1
                    MOV #TBL,R0
                    MOV #50.,R1
        LOOP:       CLR (R0)+
                    DEC R1
                    BNE LOOP
```

4. INDEX REGISTER MODIFICATION (INDEXED MODE; MODIFYING INDEX VALUE)

```
                    R0=%0
                    CLR R0
        LOOP:       CLR TBL (R0)
                    ADD #2,R0
                    CMP R0,#100.
                    BNE LOOP
```

5. FASTER INDEX REGISTER MODIFICATION (STORING VALUES IN GPR)

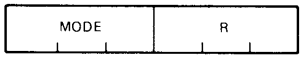
```
                    R0=%0
                    R1=%1
                    R2=%2
                    MOV #2,R1
                    MOV #100.,R2
                    CLR R0
        LOOP:       CLR TBL (R0)
                    ADD R1,R0
                    CMP R0,R2
                    BNE LOOP
```

6. ADDRESS MODIFICATION (INDEXED MODE; MODIFYING BASE ADDRESS)

```
                    R0=%0
                    MOV #TBL,R0
        LOOP:       CLR 0(R0)
                    ADD #2,LOOP+2
                    CMP LOOP+2,#100.
                    BNE LOOP
```

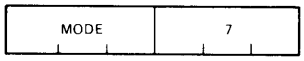
A.1 SUMMARY OF KDF11 INSTRUCTIONS

WORD FORMAT



Mode	Name	Symbolic	Description
0	register	R	(R) is operand [ex. R2 = %2]
1	register deferred	(R)	(R) is address
2	auto-increment	(R)+	(R) is adrs; (R) +(1 or 2)
3	auto-incr deferred	@(R)+	(R) is adrs of adrs; (R) +2
4	auto-decrement	−(R)	(R) −(1 or 2); is adrs
5	auto-decr deferred	@−(R)	(R) −2; (R) is adrs of adrs
6	index	X(R)	(R) + X is adrs
7	index deferred	@X(R)	(R) + X is adrs of adrs

PROGRAM COUNTER ADDRESSING Reg = 7



2	immediate	#n	operand n follows instr
3	absolute	@#A	address A follows instr
6	relative	A	instr adrs + 4 + X is adrs
7	relative deferred	@A	instr adrs + 4 + X is adrs of adrs

LEGEND

Op Codes

■	= 0 for word/1 for byte
SS	= source field (6 bits)
DD	= destination field (6 bits)
R	= gen register (3 bits), 0 to 7
XXX	= offset (8 bits), +127 to −128

Operations

( )	= contents of
s	= contents of source
d	= contents of destination
r	= contents of register
←	= becomes

Op Codes

N = number (3 bits)
NN = number (6 bits)

Operations

X = relative address
% = register definition

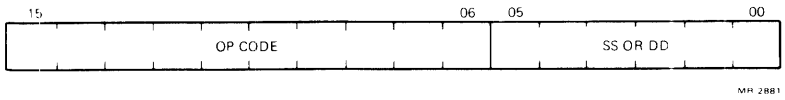
Boolean

< = AND
> = inclusive OR
> = exclusive OR
~ = NOT

Condition Codes

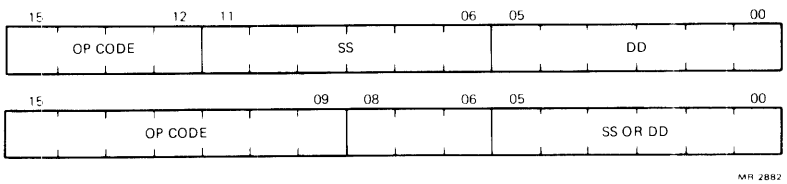
* = conditionally set/cleared
— = not affected
0 = cleared
1 = set

SINGLE OPERAND: OPR dst



Mnemonic	Op Code	Instruction	dst Result	N	Z	V	C
General							
CLR(B)	■ 050DD	clear	0	0	1	0	0
COM(B)	■ 051DD	complement (1's)	~ d	*	*	0	1
INC(B)	■ 052DD	increment	d + 1	*	*	*	—
DEC(B)	■ 053DD	decrement	d — 1	*	*	*	—
NEG(B)	■ 054DD	negate (2's compl)	—d	*	*	*	*
TST(B)	■ 057DD	test	d	*	*	0	0
Rotate & Shift							
ROR(B)	■ 060DD	rotate right	→ C, d	*	*	*	*
ROL(B)	■ 061DD	rotate left	C, d ←	*	*	*	*
ASR(B)	■ 062DD	arith shift right	d/2	*	*	*	*
ASL(B)	■ 063DD	arith shift left	2d	*	*	*	*
SWAB	0003DD	swap bytes		*	*	0	0
Multiple Precision							
ADC(B)	■ 055DD	add carry	d + C	*	*	*	*
SBC(B)	■ 056DD	subtract carry	d — C	*	*	*	*
SXT	0067DD	sign extend	0 or —1	—	*	0	—
Processor Status (PS) Operators							
MFPS	1067DD	move byte from PS	d ← PS	*	*	0	—
MTPS	1064SS	move byte to PS	PS ← s	*	*	*	*

DOUBLE OPERAND: OPR src, dst OPR src, R or OPR R, dst

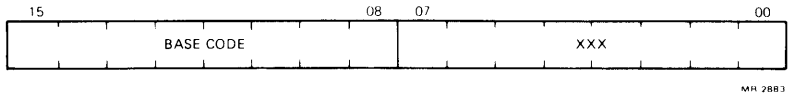


Mnemonic	Op Code	Instruction	Operation	N	Z	V	C
General							
MOV(B)	■ 1SSDD	move	d ← s	*	*	0	–
CMP(B)	■ 2SSDD	comapare	s — d	*	*	*	*
ADD	06SSDD	add	d ← s + d	*	*	*	*
SUB	16SSDD	subtract	d ← d — s	*	*	*	*
Logical							
BIT(B)	■ 3SSDD	bit test (AND)	s d	*	*	0	–
BIC(B)	■ 4SSDD	bit clear	d ← (~s) d	*	*	0	–
BIS(B)	■ 5SSDD	bit set (OR)	d ← s v d	*	*	0	–
XOR	074RDD	exclusive (OR)	d ← r v d	*	*	0	–
EIS							
MUL	070RSS	multiply	r ← r x s	*	*	0	*
DIV	071RSS	divide	r ← r/s	*	*	*	*
ASH	072RSS	shift arithmetically		*	*	*	*
ASHC	073RSS	arith shift combined		*	*	*	*

BRANCH: B—location

If condition is satisfied
Branch to location,
New PC ← Updated PC + (2 x offset)

adrs of br instr + 2



Op Code = Base Code + XXX

Mnemonic	Base Code	Instruction	Branch Condition	
Branches				
BR	000400	branch (unconditional)	(always)	
BNE	001000	br if not equal (to 0)	≠ 0	Z = 0
BEQ	001400	br if equal (to 0)	= 0	Z = 1
BPL	100000	branch if plus	+	N = 0
BMI	100400	branch if minus	—	N = 1
BVC	102000	br if overflow is clear		V = 0
BVS	102400	br if overflow is set		V = 1

Mnemonic	Base Code	Instruction		Branch Condition
BCC	103000	br if carry is clear		C = 0
BCS	103400	br if carry is set		C = 1
Signed Conditional Branches				
BGE	002000	br if greater or equal (to 0)	≥ 0	N v V = 0
BLT	002400	br if less than (0)	< 0	N v V = 1
BGT	003000	br if greater than (0)	> 0	Z v (N v V) = 0
BLE	003400	br if less or equal (to 0)	≤ 0	Z v (N v V) = 1
Unsigned Conditional Branches				
BHI	101000	branch if higher	$>$	C v Z = 0
BLOS	101400	branch if lower or same	$\leq$	C v Z = 1
BHIS	103000	branch if higher or same	$\geq$	C = 0
BLO	103400	branch if lower	$<$	C = 1

JUMP & SUBROUTINE

Mnemonic	Op Code	Instruction	Notes
JMP	0001DD	jump	PC ← dst
JSR	004RDD	jump to subroutine	use same R
RTS	00020R	return from subroutine	use same R
MARK	0064NN	mark	aid in subr return
SOB	077RNN	subtract 1 & br (if ≠ 0)	(R) — 1, then if (R) ≠ 0: PC ← Updated PC — (2 x NN)

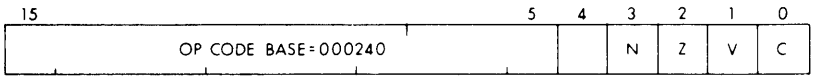
TRAP & INTERRUPT:

Mnemonic	Op Code	Instruction	Notes
EMT	104000 to 104377	emulator trap (not for general use)	PC at 30, PS at 32
TRAP	104400 to 104777	trap	PC at 34, PS at 36
BPT	000003	breakpoint trap	PC at 14, PS at 16
IOT	000004	input/output trap	PC at 20, PS at 22
RTI	000002	return from interrupt	
RTT	000006	return from interrupt	inhibit T bit trap

MISCELLANEOUS:

Mnemonic	Op Code	Instruction
HALT	000000	halt
WAIT	000001	wait for interrupt
RESET	000005	reset external bus
NOP	000240	(no operation)
MFPI	0065SS	move from previous instr space
MTPI	0066DD	move to previous instr space
MFPD	1065SS	move from previous data space
MTPD	1066DD	move to previous data space

CONDITION CODE OPERATORS:

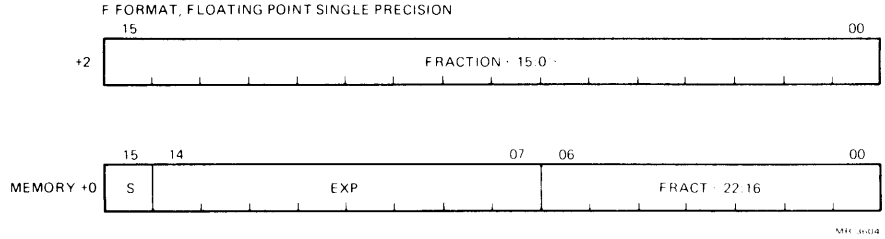


0 = CLEAR SELECTED COND. CODE BITS
1 = SET SELECTED COND. CODE BITS

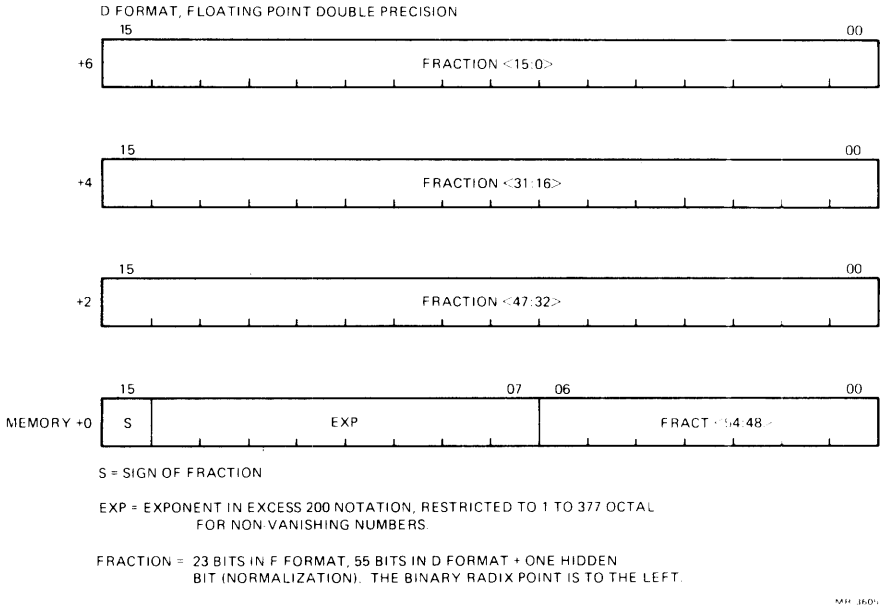
Mnemonic	Op Code	Instruction	N	Z	V	C
CLC	000241	clear C	–	–	–	0
CLV	000242	clear V	–	–	0	–
CLZ	000244	clear Z	–	0	–	–
CLN	000250	clear N	0	–	–	–
CCC	000257	clear all cc bits	0	0	0	0
SEC	000261	set C	–	–	–	1
SEV	000262	set V	–	–	1	–
SEZ	000264	set Z	–	1	–	–
SEN	000270	set N	1	–	–	–
SCC	000277	set all cc bits	1	1	1	1

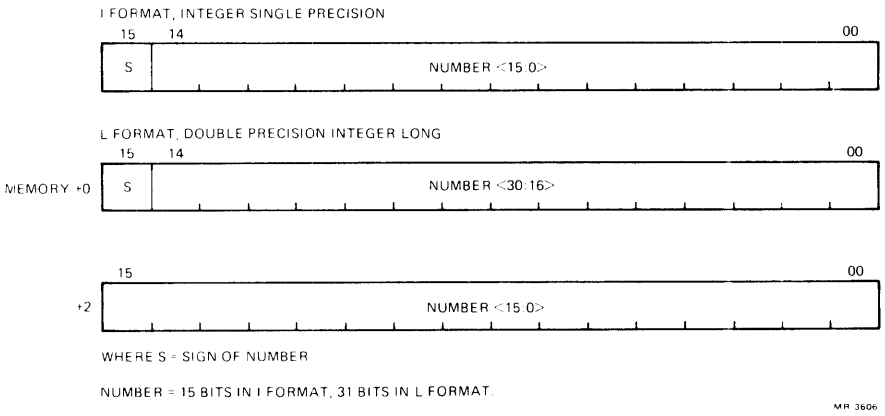
OPTIONAL FLOATING POINT:

Data Formats

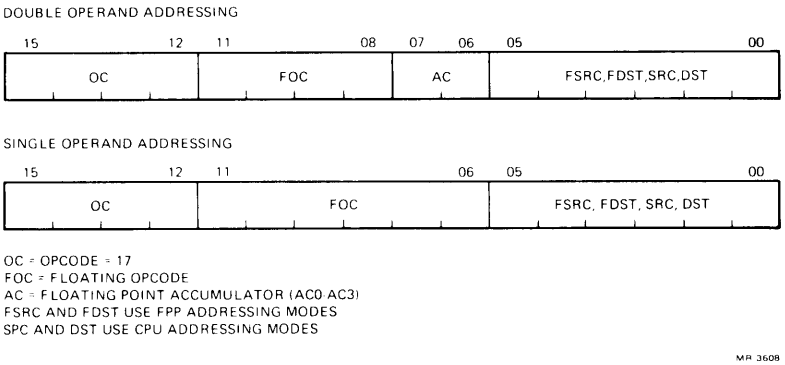


OPTIONAL FLOATING POINT:
Data Formats (Cont)





Addressing Formats



OC = OPCODE = 17
FOC = FLOATING OPCODE
AC = FLOATING POINT ACCUMULATOR (AC0-AC3)
FSRC AND FDST USE FPP ADDRESSING MODES
SPC AND DST USE CPU ADDRESSING MODES

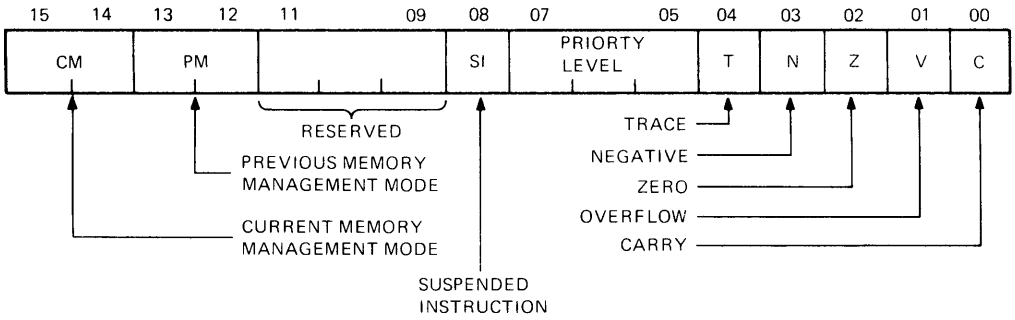
MR-3608

Mnemonic	Op Code	Instruction	Notes
CFCC	170000	copy fl cond codes	
SETF	170001	set floating mode	FD ← 0
SETI	170002	set integer mode	FL ← 0
SETD	170011	set fl dbl mode	FD ← 1
SETL	170012	set long integer mode	FL ← 1
LDFPS	1701 src	load FPP prog status	
STFPS	1702 dst	store FPP prog status	
STST	1703 dst	store (exc codes & adrs)	
CLRF, CLRD	1704 fdst	clear floating/double	fdst ← 0
TSTF, TSTD	1705 fdst	test fl/dbl	
ABSF, ABSD	1706 fdst	make absolute fl/dbl	fdst ← fdst
NEGF, NEGD	1707 fdst	negate fl/dbl	fdst ← −fdst
MULF, MULD	171 (AC) fsrc	multiply fl/dbl	AC ← AC x fsrc
MODF, MODD	171 (AC + 4) fsrc	multiply & integerize	
ADDF, ADDD	172 (AC) fsrc	add fl/dbl	AC ← AC + fsrc
LDF, LDD	172 (AC + 4) fsrc	load fl/dbl	AC ← fsrc
SUBF, SUBD	173 (AC) fsrc	subtract fl/dbl	AC ← AC - fsrc
CMPF, CMPD	173 (AC + 4) fsrc	compare fl/dbl (to AC)	
STF, STD	174 (AC) fdst	store fl/dbl	fdst ← AC
DIVF, DIVD	174 (AC + 4) fsrc	divide fl/dbl	AC ← AC/fsrc
STEXP	175 (AC) dst	store exponent	
STCFI, STCFL STCDI, STCDL	175 (AC + 4) dst	store & convert fl or dbl to int or long int	
STCFD, STCDF	176 (AC) fdst	store & convert (dbl-fl)	
LDEXP	176 (AC + 4) src	load exponent	
LDCIF, LDCIF LDCLF, LDCLD	177 (AC) src	load & convert int or long int to fl or dbl	
LDCDF, LDCFD	177 (AC + 4) fsrc	load & convert (dbl-fl)	

A.2 NUMERICAL OP CODE LIST

Op Code	Mnemonic
00 00 00	HALT
00 00 01	WAIT
00 00 02	RTI
00 00 03	BPT
00 00 04	IOT
00 00 05	RESET
00 00 06	RTT
00 00 07	
00 00 10	MFPT
00 00 77	(unused)
00 01 DD	JMP
00 02 0R	RTS
00 02 10	(reserved)
00 02 27	
00 02 40	NOP
00 02 41	cond codes
00 02 77	
00 03 DD	SWAB
00 04 XXX	BR
00 10 XXX	BNE
00 14 XXX	BEQ
00 20 XXX	BGE
00 24 XXX	BLT
00 30 XXX	BGT
00 34 XXX	BLE
00 4R DD	JSR
00 50 DD	CLR
00 51 DD	COM
00 52 DD	INC
00 53 DD	DEC
00 54 DD	NEG
00 55 DD	ADC
00 56 DD	SBC
00 57 DD	TST
00 60 DD	ROR
00 61 DD	ROL
00 62 DD	ASR
00 63 DD	
00 64 NN	MARK
00 66 SS	MFPI
00 66 DD	MTPi
00 67 DD	SXT
00 70 00	(unused)
00 77 77	
01 SS DD	MOV
02 SS DD	CMP
03 SS DD	BIT
04 SS DD	BIC
05 SS DD	BIS
06 SS DD	ADD
07 0R SS	MUL
07 1R SS	DIV
07 2R SS	ASH
07 3R SS	ASHC
07 4R DD	XOR
07 50 0R	(reserved)
07 50 1R	
07 50 2R	
07 50 3R	
07 50 40	(reserved)
07 67 77	
07 7R NN	SOB
10 00 XXX	BPL
10 04 XXX	BMI
10 10 XXX	BHI
10 14 XXX	BLOS
10 20 XXX	BVC
10 24 XXX	BVS
10 30 XXX	BCC, BHIS
10 34 XXX	BCS, BLO
10 40 00	EMT
10 43 77	
10 44 00	TRAP
10 47 77	
10 50 DD	CLRB
10 51 DD	COMB
10 52 DD	INCB
10 53 DD	DECB
10 54 DD	NEGB
10 55 DD	ADCB
10 56 DD	SBCB
10 57 DD	TSTB
10 60 DD	RORB
10 61 DD	ROLB
10 62 DD	ASRB
10 63 DD	ASLB
10 64 SS	MTPS
10 65 SS	MFPD
10 66 DD	MTPD
10 67 DD	MFPS
11 SS DD	MOVB
12 SS DD	CMPB
13 SS DD	BITB
14 SS DD	BICB
15 SS DD	BISB
16 SS DD	SUB
17 00 00	floating point
17 77 77	

A.3 PROCESSOR STATUS WORD



MR 3638

A.4

ABSOLUTE LOADER

Starting Address: – 500
Memory Size:

4K	017
8K	037
12K	057
16K	077
20K	117
24K	137
28K	157
(or larger)	

BOOTSTRAP LOADER

Address	Contents	Address	Contents
– 744	016 701	– 764	000 002
– 746	000 026	– 766	– 400
– 750	012 702	– 770	005 267
– 752	000 352	– 772	177 756
– 754	000 211	– 774	000 765
– 756	105 711	– 776	177 560 (TTY)
– 760	100 376		or 177 550 (PC11)
– 762	116 162		

773 000 Paper Tape Bootstrap
773 100 Disk/DECtape Bootstrap
773 200 Card Reader Bootstrap
773 300 Cassette Bootstrap
773 400 Floppy Disk Bootstrap

A.5 DEVICE REGISTER ADDRESSES

Device	Registers	Device Address	Interrupt Vector
Line Time Clock (external event) interrupt			100
Console Terminal			
Input Control/Status	RCSR	177560	60
Input Buffer	RBUF	177562	
Output Control/Status	XCSR	177564	64
Output Buffer	XBUF	177566	
LAV11 High-Speed Printer			
Printer Status		177514	200
Printer Buffer		177516	
High-Speed Paper Tape Reader/Punch			
Reader Status		177550	70
Reader Buffer		177552	
Punch Status		177554	74
Punch Buffer		177556	
RXV11 Floppy Disk System			
Status	RXCS	177170	264
Buffer	RXDB	177172	
REV11 ROM Programs		165000–165776, 173000–173776	

A.6 CONSOLE ODT COMMANDS

Command	Symbol	Description
Slash	/	Prints the contents of a specified location.
Carriage Return	<CR>	Closes an open location.
Line Feed	<LF>	Closes an open location and then opens the next contiguous location.
Internal Register Designator	$ or R	Opens a specifiec processor register.
Processor Status Word Designator	S	Opens the PS, must follow an "$" or "R" command.
Go	G	Starts program execution.
Proceed	P	Resumes execution of a program.
Binary Dump	Control-Shift-S	Manufacturing use only.
	H	Reserved for DIGITAL use.

A.7 7-BIT ASCII CODE

Octal Code	Char	Octal Code	Char	Octal Code	Char	Octal Code	Char
000	NUL	040	SP	100	@	140	`
001	SOH	041	!	101	A	141	a
002	STX	042	"	102	B	142	b
003	ETX	043	#	103	C	143	c
004	EOT	044	$	104	D	144	d
005	ENQ	054	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BEL	047	'	107	G	147	g
010	BS	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	*	112	J	152	j
013	VT	053	+	113	K	153	k
014	FF	054	,	114	L	154	l
015	CR	055	-	115	M	155	m
016	SO	056	.	116	N	156	n
017	SI	057	/	117	O	157	o
020	DLE	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	CAN	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB	072	:	132	Z	172	z
033	ESC	073	;	133	[	173	{
034	FS	074	<	134	\	174	|
035	GS	075	=	135	] or ↑	175	}
036	RS	076	>	136	^	176	~
037	US	077	?	137	— or ←	177	DEL

APPENDIX B
INSTRUCTION TIMING

B.1 BASIC INSTRUCTION TIMING

The following are the assumptions used to calculate instruction times.

Instruction times, are calculated using this equation

Instruction Time = Basic Time + Source Time + Destination Time

If memory management is enabled, add .16 microseconds for each memory reference. To arrive at incremental value to add to the above instruction time, use the following equation (select numbers from memory cycle column).

Increment = .16 (Reads + Writes) + .32 (RMW)

All timing is based on the MSV11-D memory (no parity) with the following characteristics. Typical times are shown for a 300 ns microcycle ±10%.

Bus Cycle	Access Time (ns)	Cycle Time (ns)
DATI	210	500
DATO(B)	100	545
DATIO(B)	630	1075

B.1.1 Source Address Time

Instruction	Source Mode	Memory Cycles	Time (Microseconds)
Double Operand	0	0	0
	1	1	1.12
	2	1	1.12
	3	2	2.25
	4	1	1.42
	5	2	2.55
	6	2	2.55
	7	3	3.67

B.1.2 Destination Time

Instruction	Destination Mode	Memory Cycles	Time (Microseconds)
	0	0	0
MOV, CLR, SXT,	1	1	1.84
	2	1	1.84
MFPS, MTPI (D)	3	2	2.66
	4	1	1.84
	5	2	2.96
	6	2	2.96
	7	3	4.09
	0	0	0
CMP, BIT,	1	1	1.42
	2	1	1.42
TST	3	2	2.25
	4	1	1.42
	5	2	2.55
	6	2	2.55
	7	3	3.67
	0	0	0
	1	1	0.22
	2	1	0.22
MTPS, MFPI (D)	3	2	1.05
MUL, DIV,	4	1	1.22
ASH, ASHC	5	2	1.35
	6	2	1.35
	7	3	2.47
	0	0	0
BIC, BIS, ADD, SUB,	1	1	2.66
SWAB, COM, INC,	2	1	2.66
DEC, NEG, ADC,	3	2	3.49
SBC, ROR, ROL,	4	1	2.66
ASR, ASL, XOR	5	2	3.79
	6	2	3.79
	7	3	4.91

B.1.3 Execute and Fetch Time

Instruction	Memory Cycles	Time (Microseconds)
MOV, CMP, BIT, ADD, SUB, BIC, BIS, SXT, CLR, TST, SWAB, COM, INC, DEC, NEG, ADC, SBC, ROR, ROL, ASR, ASL, MFPS	1	1.72

Instruction	Memory Cycles	Time (Microseconds)
MTPS	1	4.72
MFPI (D)	1	4.12
MTPI (D)	2	2.85
MUL	1	24.52
DIV	1	50.62
ASH	1	30.8
ASHC	1	47.02
All branch instructions	1	1.72
SOB (branch)	1	2.62
(no branch)	1	2.32
RTS	2	3.15
MARK	2	4.65
RTI, RTT	3	5.17
Set or Clear N, Z, V, C	1	2.62
HALT	5	-
WAIT	1	2.92
RESET	1	-
EMT, TRAP	5	7.98
IOT, BPT	5	8.85

JUMP INSTRUCTIONS

Instruction	Mode	Memory Cycles	Time (Microseconds)
	1	1	2.02
	2	1	2.32
JMP	3	2	2.85
	4	1	2.32
	5	2	3.15
	6	2	3.15
	7	3	4.27
	1	2	3.86
	2	2	4.16
JSR	3	3	4.69
	4	2	4.16
	5	3	4.99
	6	3	4.99
	7	4	6.11

B.1.4 Latency

Interrupts (BR requests) are acknowledged at the end of the current instruction. Interrupt latency, which is the time from when an interrupt is requested to when it is granted, is 10.79 microseconds (max) (non-EIS) and 55.65 microseconds (max) (EIS) for the KDF11-AA.

Interrupt service time, which is the time from BR acknowledgement to the first subroutine instruction, is 8.18 microseconds max.

NPR (DMA) latency, which is the time from request to bus mastership for the first NPR device, is 3.34 microseconds max.

B.2 FLOATING POINT INSTRUCTION TIMING (OPTION)

The execution time of a floating point instruction is dependent on the following:

1. Type of instruction
2. Type of addressing mode specified
3. Type of memory.

In addition, the execution time of many instructions, such as ADDF, are dependent on the data.

Table B-1 provides the basic instruction times for addressing mode 0 with a microcycle time of 300 ns. Tables B-2 through B-5 show the additional time required, using the MSV11-D memory with parity enabled, for instructions with other than mode 0. Refer to the notes for the execution time variations for the data-dependent instructions.

Table B-1 Instruction Times

Instruction	Micro-cycles	Mode 0 Time (Micro-seconds)	Notes	Modes 1-7
LDF	28	9.15	1,2,19	Use Table 8-2
LDD	36	11.55	1,2,23	
LDCFD	40	12.75	1,4	
LDCDF	55	17.25	1,5	
CMPF	65	20.25	14,15	
CMPD	71	22.05	14,15	
DIVF	301	91.05	1,29,41,43,44	
DIVD	795	239.25	1,30,42,43,44	
ADDF	121	37.05	1,16,17,18,20,25,27,28,41,43,44	
ADDD	139	42.45	1,16,21,22,24,26,27,28,42,43,44	
SUBF	124	37.95	1,16,17,18.20,25,27,28,41,43,44	
SUBD	142	43.35	1,16,21,22,24,26,27,28,42,43,44	
MULF	264	79.95	1,29,31,41,43,44	
MULD	641	193.05	1,30,32,42,43,44	
MODF	682	205.35	1,26,30,32,33,34,35,41,43,44	
MODD	693	208.65	1,26,30,32,33,34,36,42,43,44	
TSTF	28	9.15	1,2,37	
TSTD	32	10.35	1,2,37	

Table B-1 Instruction Times (Cont)

Instruction	Micro-cycles	Mode 0 Time (Micro-seconds)	Notes	Modes 1-7
STF	18	6.15		Use Table 8-3
STD	26	8.55		
STCDF	65	20.25	1,38	
STCFD	48	15.15	1,4	
CLRF	36	11.55		
CLRD	40	12.75		
ABSF	43	13.65	37	Use both Tables 8-2 and 8-3
ABSD	51	16.05	37	
NEGF	42	13.35	1,37	
NEGD	50	15.75	1,37	
LDFPS	11	4.05		Use Table 8-4
LDEXP	38	12.75	1,2,3,37	
LDCIF	60	18.75	6,8	
LDCID	55	17.25	6,8	
LDCLF	60	18.75	6,7,8,9	
LDCLD	55	17.25	6,7,8,9	
STFPS	16	5.55		Use Table 8-5
STST	17	5.85		
STEXP	34	10.95	1,2	
STCFI	58	18.15	11,12,39	
STCDI	59	18.45	11,12,39	
STCFL	55	17.25	10,11,13,40	
STCDL	56	17.55	10,11,13,40	
CFCC	12	4.35		No operands
SETF	14	4.95		
SETD	14	4.95		
SETI	14	4.95		
SETL	14	4.95		

Table B-2 Instruction Times

Addressing Mode	Microcycles*		Read/Write Memory Cycles		Time (Microseconds)	
	Single Precision	Double Precision	Single Precision	Double Precision	Single Precision	Double Precision
1	6,8,11	8,10,13	2/0	4/0	4.81	6.92
2	7,9,11	9,11,14	2/0	4/0	5.11	7.22
2 Immediate	6,8,11	2,4,7	1/0	1/0	4.05	2.85
3	7,9,11	9,11,14	3/0	5/0	5.86	7.97
4	7,9,11	9,11,14	2/0	4/0	5.11	7.22
5	8,10,13	10,12,15	3/0	5/0	6.16	8.27
6	8,10,13	10,12,15	3/0	5/0	6.16	8.27
7	10,12,15	12,14,17	4/0	6/0	7.52	9.63

*Note: The three numbers (of microcycles) in each set represent three different conditions:

1. If the floating point number is positive
2. If the floating point number is negative and non-0
3. If the floating point number is a negative 0 with FIUV flag clear.

Table B-3 Instruction Times

Addressing Mode	Microcycles		Read/Write Memory Cycles		Time (Microseconds)	
	Single Precision	Double Precision	Single Precision	Double Precision	Single Precision	Double Precision
1	3	5	0/2	0/4	2.56	4.82
2	6	8	0/2	0/4	3.46	5.72
2 Immediate	-2	-6	0/1	0/1	0.23	-0.97
3	4	6	1/2	1/4	3.61	5.87
4	6	8	0/2	0/4	3.46	5.72
5	5	7	1/2	1/4	3.91	6.17
6	5	7	1/2	1/4	3.91	6.17
7	7	9	2/2	2/4	5.27	7.53

Table B-4 Instruction Times

Addressing Mode	Microcycles		Read/Write Memory Cycles		Time (Microseconds)	
	Single Integer	Double Integer	Single Integer	Double Integer	Single Integer	Double Integer
1	2	4	1/0	2/0	1.35	2.71
2	3	5	1/0	2/0	1.65	3.01
2 Immediate	1	1	1/0	1/0	1.05	1.05
3	3	5	2/0	3/0	2.41	3.76
4	3	5	1/0	2/0	1.65	3.01
5	4	6	2/0	3/0	2.71	4.06
6	4	6	2/0	3/0	2.71	4.06
7	6	8	3/0	4/0	4.06	5.42

Table B-5 Instruction Times

Addressing Mode	Microcycles		Read/Write Memory Cycles		Time (Microseconds)	
	Single Integer	Double Integer	Short	Long	Single Integer	Double Integer
1	2	4	0/1	0/2	1.43	2.86
2	3	5	0/1	0/2	1.73	3.16
2 Immediate	1	1	0/1	0/1	1.13	1.13
3	3	5	1/1	1/2	2.48	3.91
4	3	5	0/1	0/2	1.73	3.16
5	4	6	1/1	1/2	2.78	4.21
6	4	6	1/1	1/2	2.78	4.21
7	6	8	2/1	2/2	4.13	5.57

NOTES

1. Add 300 ns if result is positive.
2. Add 300 ns if result is non-0.
3. Add 900 ns if SRC > 177 or SRC < -177.
4. Add 900 ns if floating point number = 0.
5. Add 3.3 microseconds if overflow on rounding.
6. Add 300 ns if integer is negative.
7. Add 1.5 microseconds if absolute value of integer < 65,536.

9. Add 1.2 microseconds n times where n = 240 - exp and if absolute value of integer > = 65,536.

10. Add 600 ns if exp < 20.

11. Add 2.1 microseconds n times where n is the smaller of the two absolute values: (310 - exp) or (230 - exp).

12. Add 600 ns if integer is negative.

13. Add 900 ns if integer is negative.

14. Add 1.2 microseconds if floating point numbers are equal.

15. Add 2.1 microseconds if numbers are unequal but the signs are the same.

16. Add 600 ns if FPACC > FPSRC.

17. Add 2.4 microseconds if FPSRC > FPACC.

18. Add 600 ns if adding opposite signs or subtracting like signs.

19. Add 2.4 microseconds if trapped on undefined variable.

20. Add 900 ns and 1.2 microseconds n times where n = exp difference.

21. Add 3.6 microseconds if FPSRC > FPACC.

22. Add 1.2 microseconds if adding opposite signs or subtracting like signs.

23. Add 1.2 microseconds if trapped on undefined variable.

24. Add 900 ns and 1.8 microseconds n times where n = exp difference.

25. Add 1.2 microseconds n times where n = shifts to normalize.

26. Add 1.8 microseconds n times where n = shifts to normalize.

27. Add 3.3 microseconds if underflow.

28. Add 600 ns if overflow.

29. Add 600 ns if need to normalize after multiply or divide.

30. Add 1.2 microseconds if need to normalize after multiply or divide.

31. Add 600 ns for every "1" bit in multiplier (FPSRC).

32. Add 1.2 microseconds for every "1" bit in multiplier (FPSRC).

33. Add 900 ns times n where n = exp module 16 (calculate integer and fraction).

34. Add 300, 600, or 900 ns if exp = 21-40, 41-60 or > 100, or 61-100 respectively.

35. Add 1.8 microseconds if the fractional part = 0.

36. Add 1.2 microseconds if the fractional part = 0.

37. Add 4.5 microseconds if trapped on any of the FP11 interrupts.

38. Add 5.4 microseconds if trapped on overflow.

39. Add 24.3 microseconds if trapped on conversion error.

40. Add 24.9 microseconds if trapped on conversion error.

41. Add 1.2 microseconds if rounding.

42. Add 1.8 microseconds if rounding.

43. Add 8.1 microseconds if trapped on overflow.

44. Add 9.9 microseconds if trapped on underflow.

B.2.1 Interrupt Latency

For all floating point instructions except ADD, SUB, MUL, DIV, and MOD, the interrupt latency is the length of the instruction. The longest execution times for each of the FPP instructions are shown in Table B-6.

In the floating point arithmetic instructions, interrupts may be serviced while in the midst of their execution. Table B-7 shows the longest times between checking for interrupt requests during the floating point instructions. If an interrupt is to be serviced before execution is complete, the instruction is aborted and all the PDP-11 general registers and floating point registers are restored to their original values. After the interrupt is serviced, the floating point instruction is restarted from scratch. This interrupt restore routine takes 6.9 microseconds and the time must be added to the interrupt latency times where execution of an instruction is aborted.

Table B-6 Longest Execution Times

Instruction	Worst Case (Mode 7) No. of Microcycles	Time (Microseconds)
LDF	50	18.77
LDD	56	22.08
LDCFD	57	20.87
LDCDF	91	33.48
CMPF	86	29.57
CMPD	96	34.08
DIVF	350	108.77
DIVD	849	259.98
ADDF	310	96.77
ADDD	596	184.08
SUBF	313	97.67
SUBD	599	184.98
MULF	361	112.07
MULD	919	280.98
MODF	1166	353.57
MODD	1311	398.58
TSTF	58	21.17
TSTD	64	24.48
STF	25	11.42
STD	35	16.08
STCDF	98	34.98
STCFD	54	20.12
CLRF	43	16.82
CLRD	49	20.28
ABSF	72	28.54
ABSD	80	34.11
NEGF	72	28.54
NEGD	80	34.11
LDFPS	17	8.11
LDEXP	64	22.21
LDCIF	127	41.11
LDCID	122	39.61
LDCLF	134	43.97
LDCLD	129	42.47
STFPS	22	9.68
STST	25	11.42
STEXP	42	15.68
STCFI	143	45.98
STCDI	144	46.28
STCFL	145	47.42
STCDL	146	47.72

Table B-6 Longest Execution Times (Cont)

Instruction	Worst Case (Mode 7) No. of Microcycles	Time (Microseconds)
CFCC	12	4.35
SEIF	14	4.95
SETD	14	4.95
SETI	14	4.95
SETL	14	4.95

Table B-7 Longest Interrupt Request Checking Times

Instruction	From	To	Max. No. of Micro-cycles	Max. Time (Micro-seconds)	Max. Latency Time (Micro-seconds)*
ADDF/SUBF	fetch	end	95	32.27	32.27
ADDF/SUBF	fetch	1st svcpt	83	28.67	35.57
ADDF/SUBF	1st svcpt	2nd svcpt	42	12.60	19.50
ADDF/SUBF	fetch	3rd svcpt	71	25.07	31.97
ADDF/SUBF	1st svcpt	3rd svcpt	48	14.40	21.30
ADDF/SUBF	2nd svcpt	3rd svcpt	11	3.30	10.20
ADDF/SUBF	3rd svcpt	4th svcpt	6	1.80	8.70
ADDF/SUBF	3rd svcpt	end	80	24.00	24.00
ADDF/SUBF	4th svcpt	end	79	23.70	23.70
ADDD/SUBD	fetch	end	105	36.78	36.78
ADDD/SUBD	fetch	1st svcpt	103	36.18	43.08
ADDD/SUBD	1st svcpt	2nd svcpt	56	16.80	24.70
ADDD/SUBD	fetch	3rd svcpt	83	30.18	37.08
ADDD/SUBD	1st svcpt	3rd svcpt	64	19.20	26.10
ADDD/SUBD	2nd svcpt	3rd svcpt	13	3.90	10.80
ADDD/SUBD	3rd svcpt	4th svcpt	8	2.40	9.30
ADDD/SUBD	3rd svcpt	end	88	26.40	26.40
ADDD/SUBD	4th svcpt	end	87	26.10	26.10
MULF	fetch	end	89	30.47	30.47
MULF	fetch	svcpt	82	28.37	35.27
MULF	svcpt	end	93	27.90	27.90
MULD	fetch	end	101	35.58	35.58
MULD	fetch	svcpt	91	32.58	39.48
MULD	svcpt	end	106	31.80	31.80
DIVF	fetch	end	89	30.47	30.47
DIVF	fetch	svcpt	73	25.67	32.57
DIVF	svcpt	end	96	28.80	28.80

Table B-7 Longest Interrupt Request Checking Times (Cont)

Instruction	From	To	Max. No. of Micro-cycles	Max. Time (Micro-seconds)	Max. Latency Time (Micro-seconds)*
DIVD	fetch	end	101	35.58	35.58
DIVD	fetch	svcpt	85	30.78	37.68
DIVD	svcpt	end	112	33.60	33.60
MODF	fetch	end	93	31.67	31.67
MODF	fetch	1st svcpt	86	29.57	36.47
MODF	1st svcpt	2nd svcpt	29	8.70	15.60
MODF	2nd svcpt	3rd svcpt	51	15.30	22.20
MODF	2nd svcpt	end	125	37.50	37.50
MODF	3rd svcpt	end	86	25.80	25.80
MODD	fetch	end	107	37.38	37.38
MODD	fetch	1st svcpt	91	32.58	39.48
MODD	1st svcpt	2nd svcpt	29	8.70	15.60
MODD	2nd svcpt	3rd svcpt	49	14.70	21.60
MODD	2nd svcpt	end	135	40.50	40.50
MODD	3rd svcpt	end	96	28.80	28.80

*Note: These times include the register restore time.

APPENDIX C
KDF11/PDP-11 PROGRAM AND OPERATION DIFFERENCE

The following pages contain a chart comparing the programming and operational features of the KDF11-AA, LSI-11 and PDP-11 processors.

ACTIVITY	PDP-11/							
	LSI-11	KDF11	04	34	05/10	15/20	35/40	45
1. OPR%R, (R)+ or OPR%R, -(R) using the same register as both source and destination: contents of R are incremented (decremented) by 2 before being used as the source operand.		X				X	X	
OPR%R, (R)+ or OPR%R, - (R) using the same register as both register and destination: initial contents of R are used as the source operand.	X		X	X	X			X
2. OPR%R, @(R)+ or OPR%R, @-(R) using the same register as both source and destination: contents of R are incremented (decremented) by 2 before being used as the source operand.		X				X	X	
OPR%R, @(R)+ or OPR%R, @-(R) using the same register as both source and destination: initial contents of R are used as the source operand.	X		X	X				X
3. OPR PC, X(R); OPR PC, @ X(R); OPR PC, @ A; OPR PC, A: Location A will contain the PC of OPR +4.		X				X	X	
OPR PC, X(R); OPR PC, @ X(R), OPR PC, A; OPR PC, @ A Location A will contain the PC of OPR +2.	X		X	X	X			
4. JMP (R) + or JSR reg, (R)+: Contents of R are incremented by 2, then used as the new PC address.					X	X		
JMP (R)+ or JSR reg, (R)+: Initial contents of R are used as the new PC.	X	X	X	X			X	X
5. JMP %R or JSR reg, %R traps to 4 (illegal instruction).	X	X	X	X	X	X	X	
JMP %R or JSR reg, %R traps to 10 (illegal instruction).								X
6. SWAB does <u>not</u> change V.						X		
SWAB clears V.	X	X	X	X	X		X	X
7. Register addresses (177700 - 177717) are valid Program addresses when used by CPU.					X			

	ACTIVITY	PDP-11/							
		LSI-11	KDF11	04	34	05/10	15/20	35/40	45
	Register addresses (177700 - 177717) time-out when used as a program address by the CPU. Can be addressed under console operation. Note addresses cannot be addressed under Console for LSI-11 or KDF11.	X	X				X	X	X
8.	Basic instructions noted in PDP-11 processor handbook.	X	X	X	X	X	X	X	
	SOB, MARK, RTT, SXT instructions.	X	X	X	X			X	X
	ASH, ASHC, DIV, MUL	X	X		X			X	X
	XOR instruction.	X	X		X			X	X
	The external option KE11-A provides MUL, DIV and SHIFT operation in the same data format.					X	X		
	The KE11-E (Expansion Instruction Set) provides the instructions MUL, DIV, ASH, and ASHC. These new instructions are 11/45 compatible.							X	
	The KE11-F adds unique stack ordered floating point instructions: FADD, FSUB, FMUL, FDIV.							X	
	The KEV-11 adds EIS/FIS instructions.	X							
	SPL instruction								X
9.	Power fail during RESET instruction is not recognized until after the instruction is finished (70 milliseconds). RESET instruction consists of 70 millisecond pause with INIT occurring during first 20 milliseconds.						X	X	
	Power fail immediately ends the REST instruction and traps if an INIT is in progress. A minimum INIT of 1 microsecond occurs if instruction aborted.								X
	Power fail acts the same as 11/45 (22 milliseconds with about 300 nanoseconds minimum). Power fail during RESET fetch is fatal with no power down sequence.			X	X	X			
	RESET instruction consists of 10 microsecond of INIT followed by a 90 microsecond pause. Power fail is not recognized until the instruction is complete.	X	X						

ACTIVITY	PDP-11/							
	LSI-11	KDF11	04	34	05/10	15/20	35/40	45
10. No RTT instruction					X	X		
If RTT sets the T bit, the T bit trap occurs after the instruction following RTT.	X	X	X	X			X	X
11. If TI sets "T" bit, "T" bit trap is acknowledged after instruction following RTI.					X	X		
If RTI sets "T" bit, "T" bit trap is acknowledged immediately following RTI.	X	X	X	X			X	X
12. If an interrupt occurs during an instruction that has the "T" bit set, the "T" bit trap is acknowledged before the interrupt.	X	X	X	X	X	X	X	
If an interrupt occurs during an instruction and the "T" bit is set, the interrupt is acknowledged before "T" bit trap.								X
13. "T" bit trap will sequence out of WAIT instruction.		X	X	X	X	X	X	
"T" bit trap will not sequence out of WAIT instruction. Waits until an interrupt.	X							X
14. Explicit reference (direct access) to PS can load "T" bit. Console can also load "T" bit				X	X	X		
Only implicit references (RTI, RTT, traps and interrupts) can load "T" bit. Console cannot load "T" bit.	X	X	X				X	X
15. Odd address/nonexistent references using the SP cause a HALT. This is a case of double bus error with the second error occurring in the trap servicing the first error. Odd address trap not in LSI-11 or F-11.	X		X	X	X	X		
Odd address/nonexistent references using the stack pointer cause a fatal trap. On bus error in trap service, new stack created at 0/2.		X					X	X
16. The first instruction in an interrupt routine will not be executed if another interrupt occurs at a higher priority level than assumed by the first interrupt.	X	X	X	X	X		X	X
The first instruction in an interrupt service is guaranteed to be executed.						X		

ACTIVITY	PDP-11/							
	LSI-11	KDF11	04	34	05/10	15/20	35/40	45
17. 8 General purpose registers	X	X	X	X	X	X	X	
16 General purpose registers								X
18. PSW address, 177776, not implemented must use new instructions, MTPS (Move to PS) and MFPS (Move from PS).	X							
PSW address implemented, MTPS and MFPS not implemented.			X		X	X	X	X
PSW address and MTPS and MFPS implemented.		X		X				
19. Only one interrupt level (BR4) exists.	X							
Four interrupt levels exist.		X	X	X	X	X	X	X
20. Stack overflow is not implemented.	X							
Stack overflow below 400 is implemented.		X	X	X	X	X		
Red and yellow zone stack overflow is implemented.							X	X
21. Odd address trap is not implemented.	X	X						
Odd address trap is implemented.			X	X	X	X	X	X
22. FMUL and FDIV instructions implicitly use R6 (one push and pop); hence R6 must be set up correctly.	X							
FMUL and FDIV instructions do not implicitly use R6.							X	
23. Due to their execution time, EIS instructions can abort because of a device interrupt.	X							
EIS instructions do not abort because of a device interrupt.		X					X	X
24. Due to their execution time, FIS instructions can abort because of a device interrupt.	X						X	
25. EIS instructions do a DATIP and DATO bus sequence when fetching source operand.	X							
EIS instructions do a DATI bus sequence when fetching source operand.		X						
26. MOV instruction does just a DATO bus sequence for the last memory cycle.	X	X		X			X	X

ACTIVITY	PDP-11/							
	LSI-11	KDF11	04	34	05/10	15/20	35/40	45
MOV instruction does a DATIP and DATO bus sequence for the last memory cycle.				X	X	X		
27. If PC contains nonexistent memory address and a bus error occurs, PC will have been incremented.	X	X	X	X	X	X		X
If PC contains nonexistent memory address and a bus error occurs, PC will be unchanged.							X	
28. If register contains nonexistent memory address in mode 2 and a bus error occurs, register will be incremented.	X	X			X	X	X	X
Same as above but register is unchanged.			X	X				
29. If register contains an odd value in mode 2 and a bus error occurs, register will be incremented.	X	X					X	X
If register contains an odd value in mode 2 and a bus error occurs, register will be unchanged.			X	X	X	X		
30. Condition codes restored to original values after FIS interrupt abort (EIS does not abort on 35/40).							X	
Condition codes that are restored after EIS/FIS interrupt abort are indeterminate.	X							
31. Op codes 075040 through 075377 unconditionally trap to 10 as reserved Op codes.		X	X	X	X	X	X	X
If KEV-11 option is present, Op codes 75040 through 075377 perform a memory read using the register specified by the low order 3 bits as a pointer. If the register contents are a non-existent address, a trap to 4 occurs. If the register contents are an existent address, a trap to 10 occurs if user microcode is not present. If no KEV-11 option is present, a trap to 10 occurs.	X							
32. Op codes 210 through 217 trap to 10 as reserved Op codes.		X	X	X	X	X	X	X
Op codes 210 through 217 are used as maintenance instruction.	X							
33. Op codes 75040 through 75777 trap to 10 as reserved Op codes.		X	X	X	X	X	X	X

ACTIVITY	PDP-11/							
	LSI-11	KDF11	04	34	05/10	15/20	35/40	45
Only if KEV-11 option is present, Op codes 75040 through 75377 can be used as escapes to user microcode. Op codes 75400 through 75777 can also be used.	X							
Used as escapes to user microcode, and KEV-11 option need not be present. If no user microcode exists, a trap to 10 occurs.								
34. Op codes 170000 through 177777 trap to 10 as reserved instructions.				X	X	X	X	
Op codes 170000 through 177777 are implemented as floating point instructions.			X	X				X
Op codes 170000 through 177777 can be used as escapes to user microcode. If no user microcode exists, a trap to 10 occurs.	X							

APPENDIX D
INTEGRATED CIRCUITS

This appendix contains reference information for the 9643 TTL-to-MOS driver (DIGITAL part number 19-16028-01) and the 2908 bus transceiver (DIGITAL part number 19-15305-00). These two integrated circuits are used on the KDF11-AA processor board.

D.1 9643

The 9643 is a dual positive logic "AND" TTL-to-MOS driver. This device has separate driver address inputs with common strobe and accepts standard TTL and DTL input signals. Provision is made for high-current and high-voltage outputs levels suitable for driving MOS circuits. Figure D-1 shows the chip layout and pin nomenclature.

D.2 2908

The 2908 is an open-collector bus tranceiver with 3-state receivers and parity. Figure D-2 shows the terminal connection diagram and defines the functional terms. Figure D-3 is the logic diagram and Figure D-4 is the associated truth table. Table D-1 lists the parity output functions.

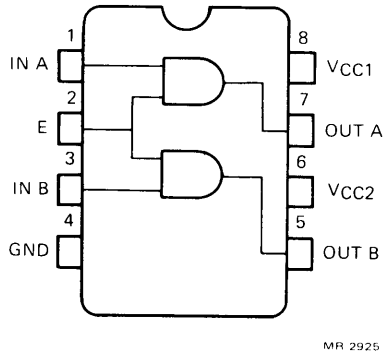
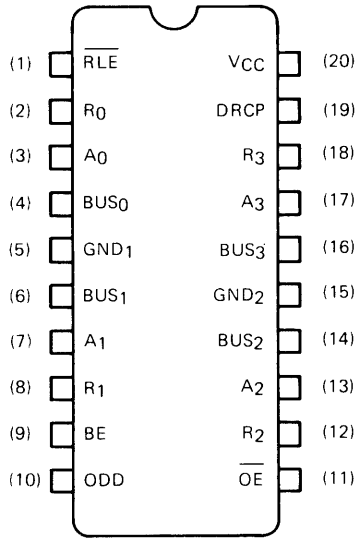


Figure D-1 9643 Connection Diagram

Table D-1 2908 Parity Output Functions

BE	Odd Parity Output
L	Odd = $A_0 \oplus A_1 \oplus A_2 \oplus A_3$
H	Odd = $Q_0 \oplus Q_1 \oplus Q_2 \oplus Q_3$



NOTE:
Pin 1 is Marked for Orientation Purposes. Numbers in ( ) Denote Terminal Numbers.

LEGEND:

$\overline{BE}$: Bus Enable. When the Bus Enable is LOW, the Four Drivers are in the High Impedance State.

BUS_0, BUS_1, BUS_2, BUS_3: The Four Driver Outputs and Receiver Inputs (Data is Inverted).

DRCP: Driver Clock Pulse. Clock Pulse for the Driver Register.

ODD: Odd Parity Output. Generates Parity With the Driver Enabled, Checks Parity With the Driver in the High-Impedance State.

$\overline{OE}$: Output Enable. When the $\overline{OE}$ Input is HIGH, the Four Three-State Receiver Outputs are in the High-Impedance State.

R_0, R_1, R_2, R_3: The Four Receiver Outputs. Data From the Bus is Inverted While Data From the A or B Inputs is Noninverted.

$\overline{RLE}$: Receiver Latch Enable. When $\overline{RLE}$ is LOW, Data on the BUS Inputs is Passed Through the Receiver Latches. When $\overline{RLE}$ is HIGH, the Receiver Latches are Closed and Will Retain the Data Independent of all Other Inputs.

MR-2926

Figure D-2 2908 Connection Diagram, and Definition of Functional Terms

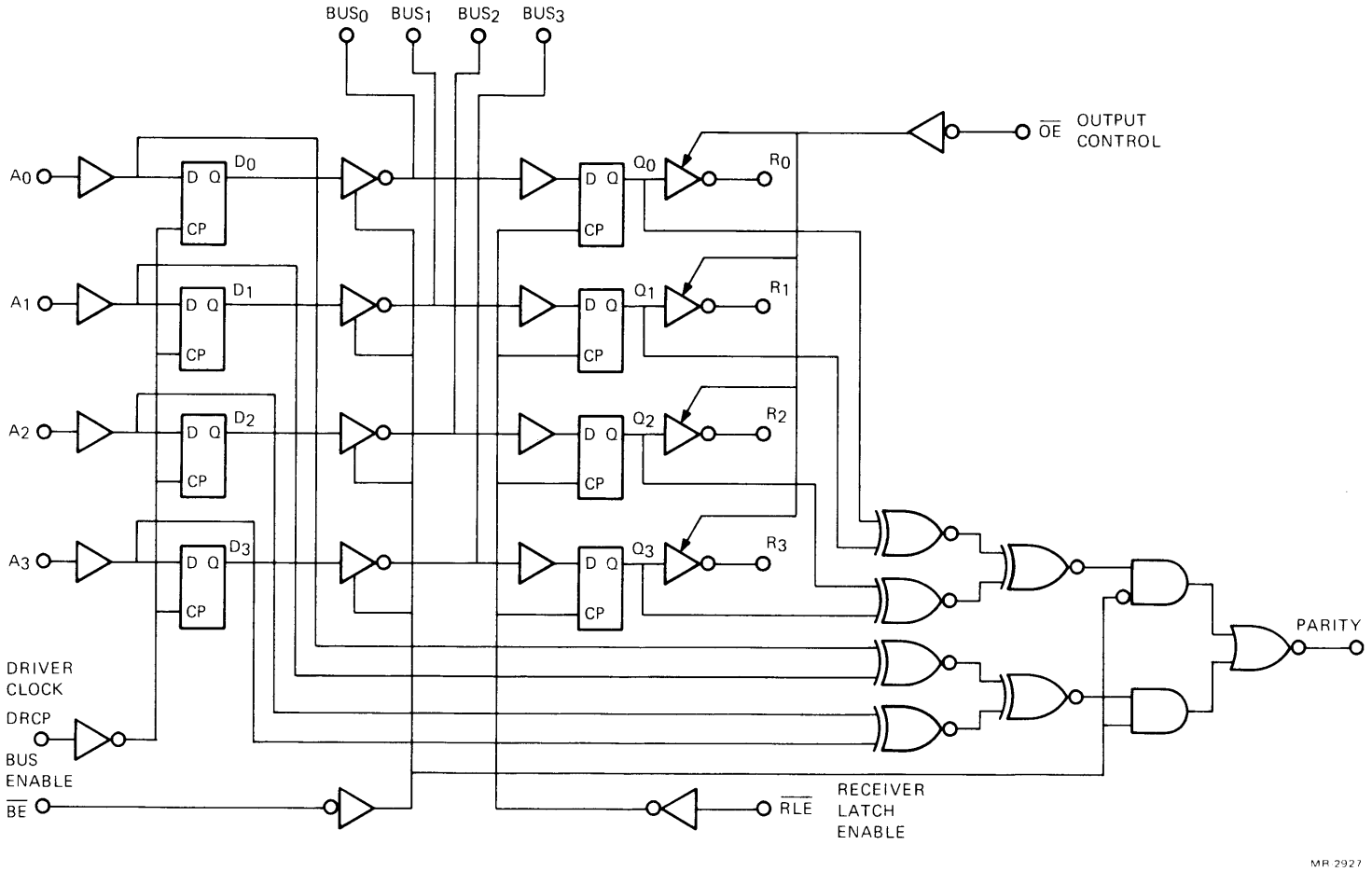


Figure D-3 2908 Logic Diagram

INPUTS					INTERNAL TO DEVICE		BUS	OUTPUT	FUNCTION
A_i	DRCP	$\overline{BE}$	$\overline{RLE}$	$\overline{OE}$	D_i	Q_i	B_i	R_i	
X	X	H	X	X	X	X	H	X	DRIVER OUTPUT DISABLE
X	X	X	X	H	X	X	X	Z	RECEIVER OUTPUT DISABLE
X	X	H	L	L	X	L	L	H	DRIVER OUTPUT DISABLE AND RECEIVE DATA VIA BUS INPUT
X	X	H	L	L	X	H	H	L	
X	X	X	H	X	X	NC	X	X	LATCH RECEIVED DATA
L	↑	X	X	X	L	X	X	X	LOAD DRIVER REGISTER
H	↑	X	X	X	H	X	X	X	
X	L	X	X	X	NC	X	X	X	NO DRIVER CLOCK RESTRICTIONS
X	H	X	X	X	NC	X	X	X	
X	X	L	X	X	L	X	H	X	DRIVE BUS
X	X	L	X	X	H	X	L	X	

H = HIGH Z = HIGH IMPEDANCE X = DON'T CARE i = 0, 1, 2, 3
L = LOW NC = NO CHANGE ↑ = LOW-TO-HIGH TRANSITION

MR-2928

Figure D-4 2908 Truth Table

E.1 RXV11 BOOTSTRAPS

Full Length Version

@1000/000000 12702 <LF>
001002/000000 1002n7 <LF>*
001004/000000 12701 <LF>
001006/000000 177170 <LF>
001010/000000 130211 <LF>
001012/000000 1776 <LF>
001014/000000 112703 <LF>
001016/000000 7 <LF>
001020/000000 10100 <LF>
001022/000000 10220 <LF>
001024/000000 402 <LF>
001026/000000 12710 <LF>
001030/000000 1 <LF>
001032/000000 6203 <LF>
001034/000000 103402 <LF>
001036/000000 112711 <LF>
001040/000000 111023 (LF)
001042/000000 30211 <LF>
001044/000000 1776 <LF>
001046/000000 100756 <LF>
001050/000000 103766 <LF>
001052/000000 105711 <LF>
001054/000000 100771 <LF>
001056/000000 5000 <LF>
001060/000000 22710 <LF>
001062/000000 240 <LF>
001064/000000 1347 <LF>
001066/000000 122702 <LF>
001070/000000 247 <LF>
001072/000000 5500 <LF>
001074/000000 5007 <CR>

Abbreviated Version (DRIVE 0 ONLY):

@1000/000000 5000 <LF>
001002/000000 12701 <LF>
001004/000000 177170 <LF>
001006/000000 105711 <LF>
001010/000000 1776 <LF>
001012/000000 12711 <LF>
001014/000000 3 <LF>
001016/000000 5711 <LF>
001020/000000 1776 <LF>
001022/000000 100405 <LF>
001024/000000 105711 <LF>
001026/000000 100004 <LF>
001030/000000 116120 <LF>
001032/000000 2 <LF>
001034/000000 770 <LF>
001036/000000 0 <LF>
001040/000000 5007 <CR>

*n = 4 for Unit 0
n = 6 for Unit 1
<LF> = Line Feed
<CF> = Carriage Return
Starting address = 1000

E.2 RKV11 BOOTSTRAP (Drive 0 only)

(terminal response underlined)

```
START   @001000/000000 5<LF>
        001002/000000 10061<LF>
        001004/000000 6<LF>
        001006/000000 12761<LF>
        001010/000000 177400<LF>
        001012/000000 2<LF>
        001014/000000 12711<LF>
        001016/000000 5<LF>
        001020/000000 105711<LF>
        001022/000000 100376<LF>
        001024/000000 5007<LF>
        001026/000000 0<CR>
        @R0/XXXXXX 0<LF>
        R1/XXXXXX 177404<CR>
```

APPENDIX F
ODT DIFFERENCES

A number of differences exist between console ODT for the KDF11-AA and console ODT for the KD11-F. The following list describes these differences.

KD11-F	KDF11-AA
1. All characters that are input are echoed except when in the APT command mode where no characters are echoed. An echoed line feed <LF> will be followed by a carriage return <CR> only (no second <LF> or padding nulls). This method creates a potential timing problem with a TTY ASR33 which types the next character before the print head has completely returned.	All characters that are input in any command mode, except the APT mode, are echoed except the octal codes 0, 2, 10, 12, 200, 202, 210, and 212. This suppresses echoing LFs [and nulls (0), STXs (2), BSx (10)] because an automatic <CR> and LF> follow. In the APT command mode, no input characters are echoed.
2. When an address location is open, another location can be opened without explicitly closing the first location, e.g., 1000/123456 2000/054321.	An address location must be explicitly closed by a <CR> or <LF> command before another is opened or else an error (?) will occur and any open location will automatically be closed without altering its contents.
3. "↑" will open the previous location.	"↑" is illegal and micro-ODT prints "?", <CR>, <LF>, "@"
4. "@" will open a location using indirect addressing.	"@" is illegal and micro-ODT prints "?", <CR>, <LF>, "@"
5. "<" will open a location using relative addressing.	"?" is illegal and micro-ODT prints "?", <CR>, <LF>, "@"
6. "M" will print the contents of an internal CPU register.	"M" is illegal and micro-ODT prints "?", <CR>, <LF>, "@"
7. Rubout (ASCII 177) will delete the last character typed in.	Rubout is illegal and micro-ODT prints "?", <CR>, <LF>, "@"
8. "L" is the boot loader command which will load the absolute loader.	"L" is illegal and micro-ODT prints "?", <CR>, <LF>, "@"
9. Control-Shift-S command mode (ASCII 23) accepts 2 bytes forming a 16-bit address and dumps 10 bytes in binary format. The 2 input bytes are not echoed.	Control-Shift-S command mode (ASCII 23) accepts 2 bytes forming an 18-bit address with bits <17:15> always 0s and dumps 10 bytes in binary format. The 2 input bytes are not echoed.
10. Up to a 16-bit address and 16-bit data may be entered. Leading 0s are assigned.	Up to an 18-bit address and 16-bit data may be entered. Leading 0s are assumed.
11. Incrementing (LF) the address 177776 results in the address 000000.	Incrementing (LF) the addresses 177776, 377776, 577776 and 777776 result in the addresses 000000, 200000, 400000 and 600000 respectively, i.e., the upper 2 bits of the 18-bit address are not affected; they must be explicitly set.
12. Incrementing a PDP-11 register from R7 prints out "R8" and the contents of R0.	Incrementing a PDP-11 register from R7 prints out "R0" and the contents of R0.
13. The 10 page is in the address group 17XXXX.	The 10 page is in the address groupt 77XXXX where address bits <17:12> must be explicitly 1s.

KD11-F	KDF11-AA
14. The micro-ODT mode can be entered from the following sources: a. A PDP-11 HALT instruction. b. A double bus error. c. An asserted HALT line. d. A power up option. e. An asserted HALT line caused by a DLV11 framing error. f. A micro-ODT bus error. g. A memory refresh bus error. h. An interrupt vector time-out. i. A nonexistent micro-PC address.	The micro-ODT mode can be entered from the following sources: a. A PDP-11 HALT instruction when in kernel mode, the P0KL line is low and the HALT jumper option strap is present. b. An asserted HALT line. c. A power-up option. d. An asserted HALT line caused by a DLV11 framing error. e. A micro-ODT bus error.
15. A carriage return <CR> is echoed and followed by just a line feed <LF>.	A carriage return <CR>. is echoed and followed by another <CR> and line feed <LF>.

APPENDIX G
KD11-F/KD11-HA/KDF11-AA DETAILED COMPARISON

The KDF11-A module uses five bused spare lines that were reserved for future expansion to implement 18-bit addressing (two lines) and 4-level interrupts (three lines). In addition, the processor uses several of the SSPARE lines for test points or for control functions required during manufacturing testing of the boards. These lines should not cause users any problems unless they have inadvertently bused user signals across the backplane on these pins. For a backplane pin assignment comparison, see Table G-1.

The KDF11-AA uses the LSI-11 bus closer to its specified limits than either the KD11-F or KD11-HA. These bus timing differences, listed in Table G-2, should have no effect on any user of LSI-11 peripherals or memories since a safety margin still exists between actual times and bus limits.

Table G-1 Backplane Pin Assignment Comparison

Line	Backplane Name	KDF11-AA	KD11-F	KD11-HA
AA1	BSPARE1	BIRQ5L	Reserved*	Reserved*
AB1	BSPARE2	BIRQ6L	Reserved*	Reserved*
BP1	BSPARE6	BIRQ7L	Reserved*	Reserved*
AC1	BAD16	BDAL16L	Reserved*	Reserved*
AD1	BAD17	BDAL17L	Reserved*	Reserved*
AE1	SSPARE1	Single Step	Not Used	STOP L
AF1	SSPARE2	SRUNL	SRUNL	SRUNL
AH1	SSPARE3	SRUNL	Not Used	SRUNL
AK1	MSPAREA	Not Used	Not Used	MTOEL
AL1	MSPAREA	Not Used	Not Used	GND
AM2	BIAKIL	MMU STRH	Not Used	Not Used
AR1	BREFL	Not Used†	BREFL	Not Used†
AR2	BDMGIL	UBMAAPL	Not Used	Not Used
BC1	SSPARE4	MMU DAL18H	Not Used	SCLK3H
BD1	SSPARE5	MMU DAL19H	Not Used	SWMIB18H
BE1	SSPARE6	MMU DAL20H	Not Used	SWMIB19H
BF1	SSPARE7	MMU DAL21H	Not Used	SWMIB20H
BH1	SSPARE8	CLK DISL	Not Used	SWMIB21H
BK1	MSPAREB	Not Used	4K RAM BIAS	Not Used
BL1	MSPAREB	Not Used	4K RAM BIAS	Not Used

*Even though these lines are not used on the KD11-F and KD11-HA, they are bused on the backplane and terminated for future bus expansion.

†Not used on the KDF11-AA and KD11-HA but terminated in the inactive state to prevent problems with older memories.

All remaining pins are identical among all three processors.

Table G-2 Comparison of KD11-F, KD11-HA, and KDF11-AA Bus Timing

Interval	Bus Specification (ns)	KD11-F (ns)	DK11-HA (ns)	KDF11-AA (ns)
BSYNC L - BDIN L	100	200	188	144
BSYNC L - BDOUT L	200	300	281	288
BSYNC L - BIAK L	325	600	562	435
Address setup time on bus	150	300	281	180
Address hold time on bus	100	100	100	108
Replay to DIN/DOUT inactive time	200	700+400/-0	675+375/-0	225+72/-0

APPENDIX H
PARITY ON THE LSI-11 BUS

H.1 INTRODUCTION

This appendix describes the method for reporting parity errors to the KDF11-AA or DMA devices, and parity control information to the MSV11-DE memory boards. Two different control circuits are suggested which implement parity error reporting.

The first design emulates the parity registers in a (PDP-11) MS-11 memory. This relatively complicated circuit can be tested with a standard parity diagnostic which also completely exercises the parity memory. This design also allows parity memories and nonparity memories to be mixed in a system.

The second circuit presents a simpler parity controller design. This circuit is sufficient if mixed memories are not present, and there is no need to run a standard parity controller/memory diagnostic.

Both options provide a "write wrong parity" diagnostic feature.

H.2 PARITY CONTROL AND STATUS ON THE LSI-11 BUS

BDAL<17:16>, which are time-multiplexed signals (see Chapter 4), report parity errors to both the KDF11-AA processor and DMA devices, and control information to the MSV11-DE. During the address phase of all bus cycles (150 ns before T SYNC until 100 ns after T SYNC), BDAL<17:16> represent the two highest address bits of an address. BDAL<17:16> only have parity significance during the data phase of a reference (read operations: 200 ns after R EPLY until 25 ns after T DIN; write operation: 100 ns before T DOUT until 100 ns after T DOUT). The MSV11-DE asserts BDAL 16 during the data portion of a read operation to report a parity error to the KDF11-AA or DMA device, and the parity controller asserts BDAL 16 during the data portion of a write operation to send control information (write wrong parity) to the MSV11-DE*. BDAL 17 is only used by the KDF11-AA. The assertion of this bit by the parity controller during the data phase of a read operation will enable processor parity trapping (see Chapter 10). When BDAL<17:16> are both asserted during the data phase of a KDF11-AA read operation, the processor will abort the bus cycle and immediately trap to location 114_8. The actual sampling of BDAL<17:16> by the KDF11-AA is done 250-300 ns after receiving BRPLY from the memory. Table H-1 summarizes the parity control and status information sent on the LSI-11 Bus.

H.3 MS-11 PARITY CONTROL REGISTER EMULATION

The MS-11 is a PDP-11 Unibus MOS memory with parity. It is desirable to emulate this memory since a comprehensive MS-11 parity diagnostic exists to test the memory and parity control

*Care must be taken to make sure that DAL 16 is asserted in a manner that does not violate LSI-11 Bus timing.

logic. The diagnostic can be run in a system that contains nonparity memory. In this case, the parity subtests are bypassed in each nonparity memory bank.

Table H-1 Parity Control and Status Information on the LSI-11 Bus (Data Phase)

	Read Operation		Write Operation	
	From	Function	From	Function
BDAL 16	MSV11-DE	Parity error status for KDF11-AA or DMA devices.	Parity controller	Write wrong parity on MSV11-DE
BDAL 17	Parity controller	Enable KDF11-AA parity trap through location 114_8 (if bus master).	X	No MSV11-DE function

Each MS-11 memory contains 8K words of memory and a parity control register. There can be a maximum of 128K words of memory; therefore, a system can have 16 parity control registers. Each register has the format shown in Figure H-1.

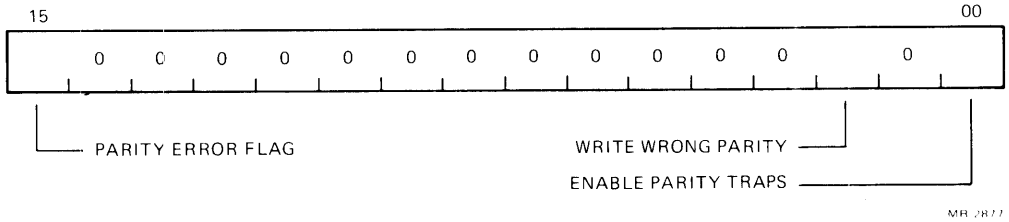


Figure H-1 Parity Control Register Format

These registers respond to addresses 772100-772136. The 8K bank number selects the control register.

The following functionality is designed into the LSI-11 Bus MS-11 parity circuit. When a parity error is detected, the parity error flag is set in the control register that controls the bank of memory in which the parity error occurred. This bit can only be cleared by writing a 0. If the enable parity trap bit is set in the selected control register when the error is detected, BDAL 17 is asserted by the circuit. This causes the KDF11-AA to trap. If bit 2 (write wrong parity) is enabled in the selected register during a write operation, BDAL 16 will be asserted on the LSI-11

Bus and wrong parity will be set in the MSV11-DE. An indication of the memory address that caused the parity error can be obtained by looking at the address pushed onto the stack during the trap. The address on the stack will point to the instruction or data reference that caused the error.

As shown in Figure H-2, a 16 X 4 RAM is used to store the parity control registers. A major portion of the logic is used to clear the RAM at power-up, and every time a bus reset (BINIT) is issued on the LSI-11 bus. Once the initialization is started, the logic sequences a 4-bit counter (Init Address Generation) and, pausing at each count, writes 0s into the selected RAM cells (Init Control).

Following this process, the RAM is explicitly or implicitly involved in every memory cycle. Explicit addressing occurs when any of the parity control registers (772100 through 772136) are accessed. This causes DAL<4:1> to be switched to the RAM address (Implicit/Explicit RAM Address MUX) and DAL<15, 2, 0> to be directed to the RAM input (CSR 15 MUX). The logic detects this mode (Explicit Address Detection) by examining the LSI-11 bus address lines, BBS7 L, and 16 switches which mark each 8K word memory bank as either a parity or nonparity memory. If the selected memory contains no parity, the explicit addressing mode is not generated.

Implicit cycles (nonexplicit) connect address bits<17:14> (Implicit/Explicit RAM Address MUX) to the RAM. This selects the parity control register which controls the addressed memory. In an implicit read cycle, bit 15 of the register is flagged with an error if the selected memory has previously been marked with an error, or if DAL 16 of the current cycle signals a parity error (CSR 15 MUX).

DAL<17:16> are asserted when appropriate (DAL<17:16> Drivers) in the manner previously described.

H.4 SIMPLE PARITY CONTROLLER (Figure H-3)

This circuit is a simplified version of the circuit shown in Figure H-2. The 16 X 4 RAM is replaced by a simple latch. This eliminates all the RAM initialization circuitry (Init Control and Init Address Generation). The explicit addresses of the registers do not have to be decoded. This reduces the complexity by simplifying the decode logic and eliminating the Implicit/Explicit RAM Address MUX. The control logic is simplified as a result.

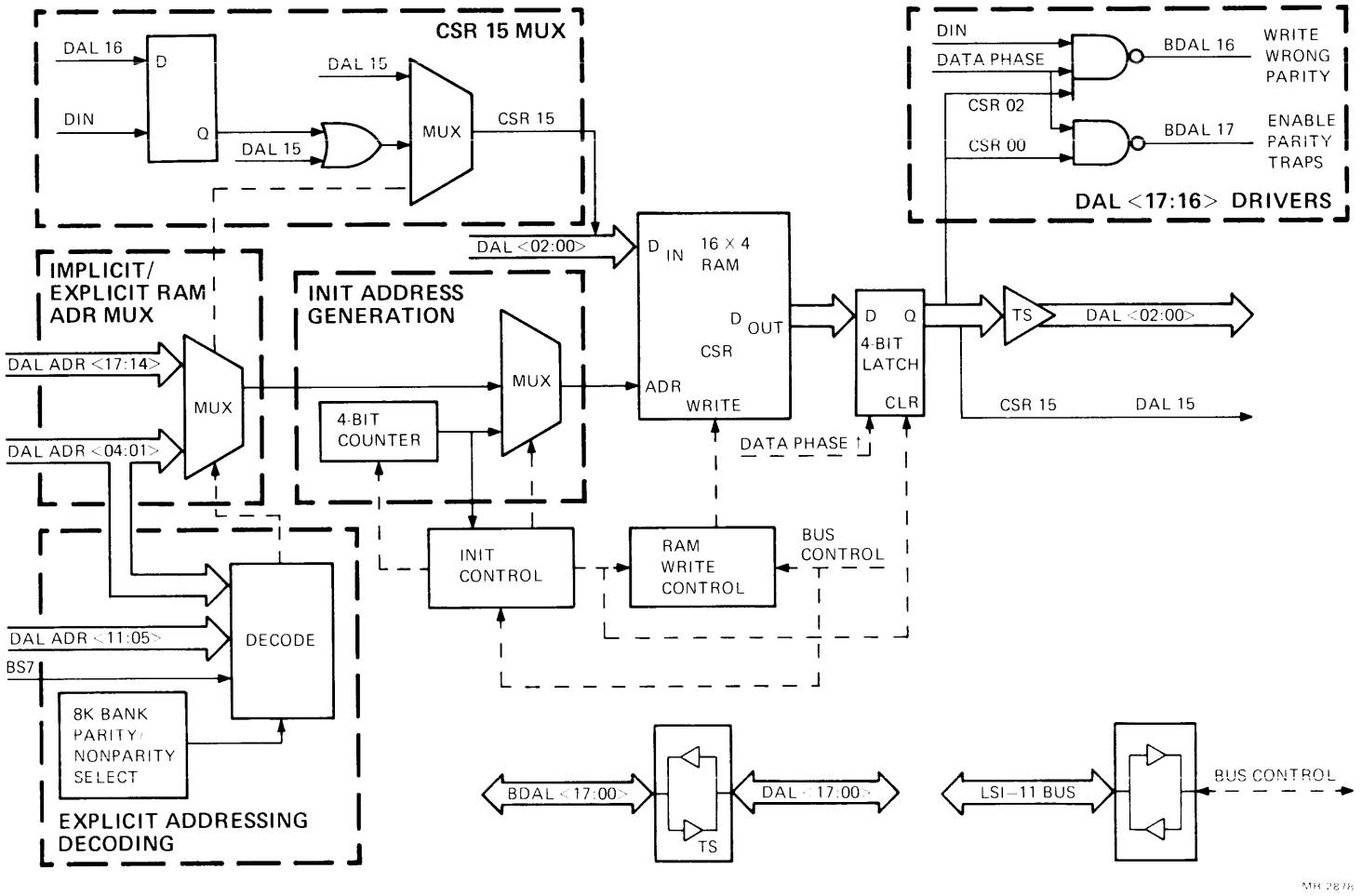


Figure H-2 LSI-11 Bus MS-11 Emulation

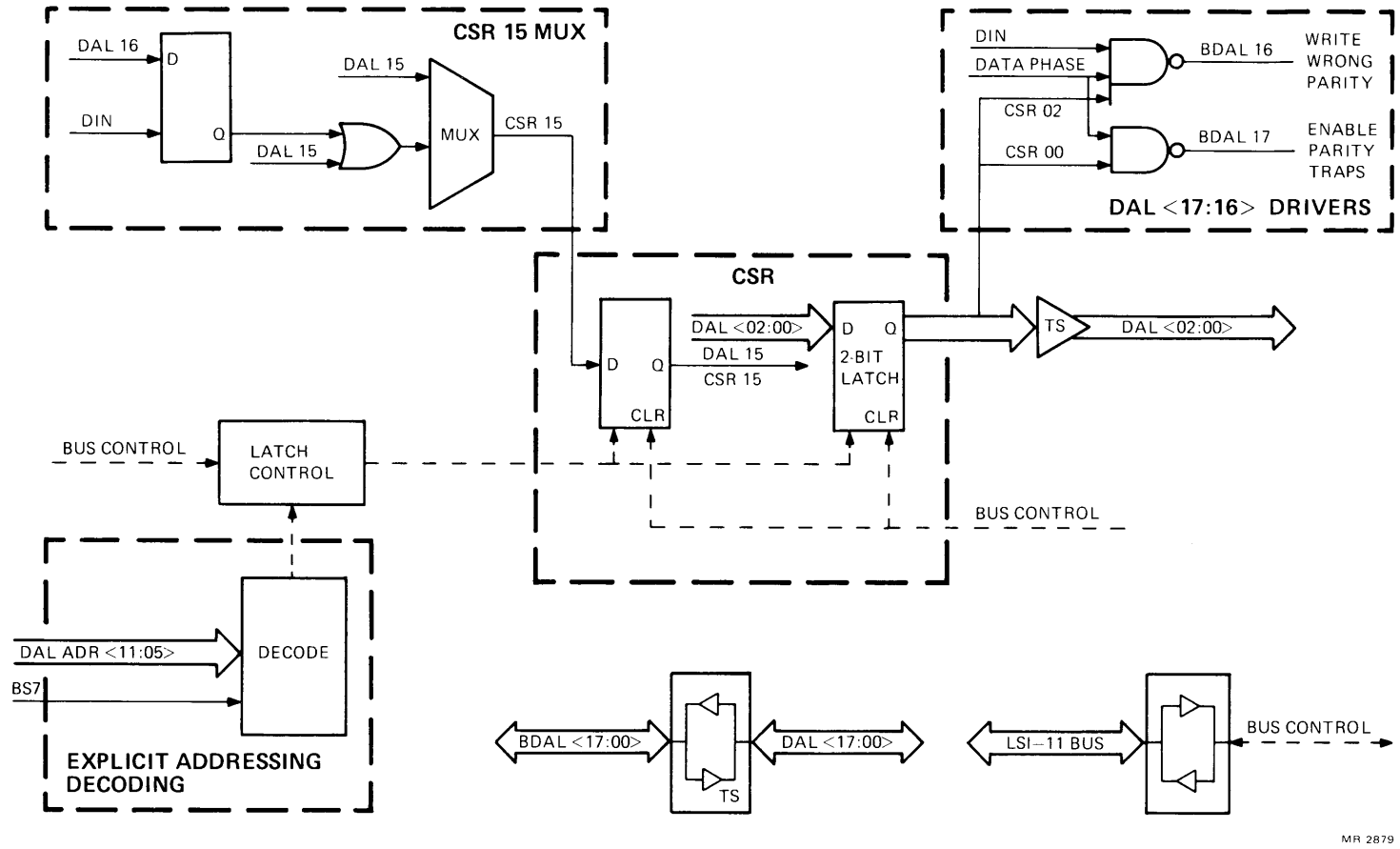


Figure H-3 Simple Parity Controller

KDF11-AA USER'S GUIDE
EK-KDF11-UG-PR2

Reader's Comments

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 or errors have you found in the manual? _____

_____

_____

_____

Does this manual satisfy the need you think it was intended to satisfy? _____

Does it satisfy *your* needs? _____ Why? _____

_____

_____

_____

☐ Please send me the current copy of the *Technical Documentation Catalog*, which contains information on the remainder of DIGITAL's technical documentation.

Name _____ Street _____
Title _____ City _____
Company _____ State/Country _____
Department _____ Zip _____

Additional copies of this document are available from:

Digital Equipment Corporation
444 Whitney Street
Northborough, Ma 01532
Attention: Printing & Circulation Services (NR2/M15)
Customer Services Section

Order No. EK-KDF11-UG

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

Digital Equipment Corporation
Communications Development and Publishing
200 Forest Street (MR1-2/T17)
Marlboro, Massachusetts 01752