KDF11-BA CPU Module User's Guide

Prepared by Educational Services
of
Digital Equipment Corporation

1st Edition, January 1982

Copyright © 1982 by Digital Equipment Corporation

All Rights Reserved

The material in this manual is for informational purposes and is subject to change without notice.

Digital Equipment Corporation assumes no responsibility for any errors which may appear in this manual.

Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation.

DIGITAL	DECsystem-10	MASSBUS
DEC	DECSYSTEM-20	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EduSystem	RSTS
UNIBUS	VAX	RSX
DECLAB	VMS	IAS
		MINC-11

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

FIGURES

FIGURES (Cont)

FIGURES (Cont)

FIGURES (Cont)

TABLES

TABLES (Cont)

PREFACE

This guide is meant to familiarize you with the purpose and uses of the KDF11-BA Central Processor Unit (CPU) module. Included are explanations of the features, options, capabilities, and technical characteristics of the module, as well as general reference data. Specifically, this guide presents:

- Information needed to configure, install, and operate the CPU module in a computer system.
- An explanation of the module's configuration requirements and a definition of the factory configuration.
- The module's hardware and software operating features.
- A functional description of the module's major logic elements (using block diagrams).
- General reference information and the differences between the KDF11-BA CPU module and previous LSI-11 CPU modules (Appendices A through F).

CHAPTER 1
SPECIFICATIONS

1.1 INTRODUCTION

The KDF11-BA is a quad-height PDP-11 CPU module (M8189). This module contains a central processor, memory management unit (MMU), a line frequency clock, a BDV11-compatible bootstrap and diagnostic ROM, and two serial-line units. Three extra 40-pin sockets are provided for optional floating-point and commercial instruction sets. The central processor and memory management units are functionally compatible with the KDF11-AA CPU and MMU.

The KDF11-BA CPU supports up to 256K bytes of memory on a traditional LSI-11 bus backplane (18 address bits) or up to 4 megabytes of memory when the module is installed in an extended LSI-11 bus backplane (H9276 or H9275). The extended LSI-11 bus backplane adds four address lines to the LSI-11 bus to provide a 22-bit addressing capability when the KDF11-BA is used with the MSV11-P (M8067) memory module. The extended LSI-11 bus will be referred to throughout this manual as the LSI-11 bus except in those cases where a distinction must be made between it and the traditional LSI-11 bus.

The central processor uses the LSI-11 bus or extended LSI-11 bus with 4-level interrupt bus protocol. The KDF11-BA is compatible with existing LSI-11 processors and peripheral devices.

The LSI-11 bus is built based on LSI-technology requirements consistent with low-cost, high-performance and small-board-form factors. Low cost and high performance are realized, in part, through use of multifunction lines such as the data/address lines (DALs), which 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 extended LSI-11 bus is contained in Chapter 4.

The KDF11-BA is software-compatible with the PDP-11 family. A wide range of software is available, including programming languages, diagnostic software, and operating systems. Note, however, that not all PDP-11 family software uses the extended addressing capability (22 bits) of the KDF11-BA.

1.2 FEATURES

The KDF11-BA CPU module (M8189) has the following features.

KDF11-AA-Compatible CPU

- Instruction set with over 400 instructions
- 4-level vectored interrupts
- 16-bit word or 8-bit byte addressable locations
- Multiple general-purpose registers
- Stack processing
- Direct memory access (DMA)
- Power-fail/autorestart hardware
- 18-bit ODT console emulator

KDF11-AA-Compatible Memory Management

- 18- or 22-bit address
- Kernel and user modes only (no supervisor mode)
- I-space only (no D-space)

Optional Floating-Point Instruction Set

Optional Commercial Instruction Set

On-Board Peripherals

- Line frequency clock
- BDV11-compatible boot and diagnostic
- Console serial-line unit
- Second serial-line unit

Extended LSI-11 Bus Interface (AB Rows)

1.3 SPECIFICATIONS

Identification	M8189
Size	Quad
Dimensions	26.6 cm × 22.8 cm (10.5 in × 8.9 in)
Power Consumption	+5 V ± 5% at 6.4 A (maximum), 4.5 A (typical)
	+12 V ± 5% at 0.7 A (maximum), 0.3 A (typical)
AC Bus Loads	2 unit loads
DC Bus Loads	1 unit load
Environmental	
Storage	−40° C to 65° C (−40° F to 150° F) 10% to 90% relative humidity, noncondensing
Operating	5° C to 60° C (41° F to 140° F) 10% to 90% relative humidity, noncondensing
	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).
Instruction Timing	Based on 75 ns intervals (See Appendix B.)

Interrupt Latency	5.7 μs 12.600 μs, maximum (except EIS) 54.225 μs, maximum (including EIS)
Interrupt Service	8.625 μs (memory management off) 9.750 μs (memory management on)
DMA Latency	1.35 μs, maximum

NOTE

Interrupt and DMA latencies assume a KDF11-BA with Memory Management Enabled and using MSV11-P memory.

1.4 PROCESSOR HARDWARE

The KDF11-BA processor is implemented using three chips. Two MOS/LSI chips, data and control on a single hybrid package, 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 contained in one 40-pin package. The MMU chip (DC304) contains the registers for 18-bit or 22-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 nine 16-bit general-purpose registers that provide for a variety of functions. Note, however, that only eight of these registers may be used at any given time. 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 1-1 identifies the general registers R0 through R7.

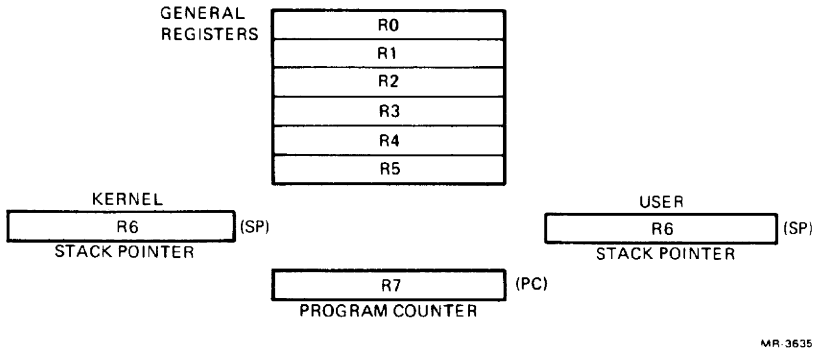


Figure 1-1 General-Purpose Registers

Registers R6 and R7 are dedicated. The KDF11-BA contains two R6 registers which are selected by the processor status word (PS) so that only one is accessible at any given time. 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. Register R7 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, however, and longer execution time. Thus, general registers used for processor operations result in faster execution times.

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

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 more 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, while a DMA operation can occur between bus cycles. For more details on bus operations refer to Chapter 4.

1.4.3 Addressing Memory and Peripherals

The KDF11-BA processor uses 16-bit data paths throughout. These 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 1-2. Word addresses are always even-numbered. Byte addresses can be either even- or odd-numbered. Low bytes are stored at even-numbered memory locations, high bytes at odd-numbered memory locations. Thus, it is convenient to view the memory as shown in Figure 1-3.

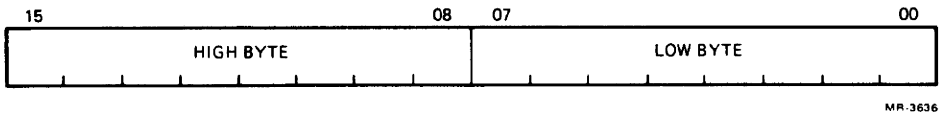


Figure 1-2 High and Low Bytes of a Processor Word

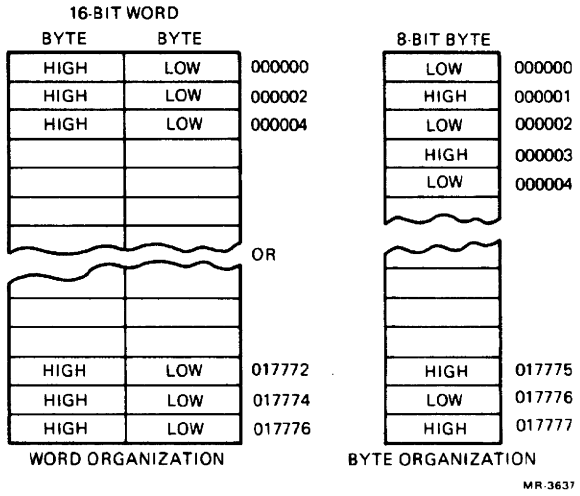


Figure 1-3 Word and Byte Addresses for First 4K Words of Memory

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 addresses, such as multiprogramming and/or timesharing applications, six additional addressing bits are available. These bits allow up to 2 megawords of memory to be physically addressed by the processor. This additional addressing capability is part of the standard memory management within the KDF11-BA architecture.

1.4.4 Memory Management

The memory management has the following three major features.

1. Two software modes that are useful for multiuser (timesharing) systems.

2. Extended memory addressing (greater than 32K words, up to 2 megawords) to allow 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 provides a kernel and user mode to allow efficient segmentation of memory for multiuser environments. Kernel mode is employed by the operating system to control system resources and allows full privileges of the entire system. The user mode is employed for executing a user program and restricts processor privileges. In user mode, the processor is inhibited from executing certain instructions (e.g., the HALT instruction cannot be executed).

The second feature provides a full 22-bit memory addressing capability. Mapping registers are used to map (relocate) the 32K-word virtual address space anywhere in the 2 megaword physical address space.

The third feature allows restricted access to virtual memory pages (a page is defined as 4K words long). This permits the operating system software rather than the user program to control system resources in a multiprogramming environment. This feature ensures that no user operating in user mode can cause a failure of the entire system. 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 status of the processor. As shown in Figure 1-4, 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. The processor status word is located at physical address 17777776.

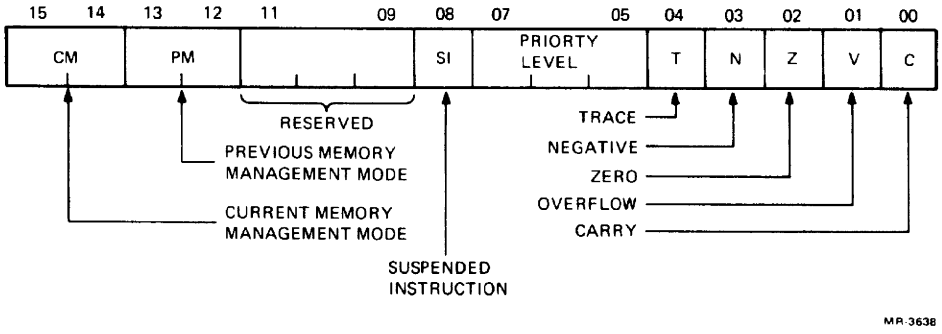


Figure 1-4 Processor Status Word (PS) Format

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 the MSB or LSB (least significant bit).

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

1.4.5.3 Priority Level (PS bits <7:5>) - These bits are used by the 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 use by DIGITAL and is intended for options such as the commercial instruction set (CIS). This bit is read/write and has no protection mechanism. Refer to Paragraph 8.5.3.2 for more details.

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

1.5 INSTRUCTION SET

The KDF11-BA instruction set provides over 400 powerful instructions. As a comparison with other instruction sets, consider that most other (for example, accumulator-oriented) 16-bit processors require three separate instructions to execute a common double-operand instruction (e.g., ADD). The following is the conventional approach to a simple operation.

```
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-BA can fetch both operands, execute, and store the result in one instruction.

```
ADD A, B  Add contents of location A to location B; store result 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 of the KDF11-BA 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-BA 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-AA) on the KDF11-BA 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 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, PDP-11/60, and PDP-11/70. For a complete description refer to Chapter 9.

1.7 COMMERCIAL INSTRUCTION SET OPTION

The commercial instruction set (CIS) is a microcode option (KEF11-BB) that adds character string instructions to the basic PDP-11 instruction set. The character string operations conveniently implement most of the common, as well as time consuming functions that are encountered in commercial data and text processing applications. The microcode option is completely compatible with the standard PDP-11 commercial instruction set. The CIS microcode resides in six MOS/LSI chips mounted on a single double-width 40-pin carrier.

1.8 MEMORIES AND PERIPHERALS

Digital Equipment Corporation provides a wide range of memories and peripherals to allow maximum flexibility in configuring systems. A detailed list and descriptions can be found in the *Microcomputer and Memories Handbook* and the *Microcomputer Interfaces Handbook*.

1.9 RELATED DOCUMENTS

Table 1-1 lists documents containing additional information of possible interest to KDF11-BA processor users.

Table 1-1 Related Documentation

Title	Document Number
Microcomputer Interfaces Handbook	EB-20175-20/80
Microcomputer and Memories Handbook	EB-18451-20/80
PDP-11 Processor Handbook	EB-09402-20/81
PDP-11 Software Handbook	EB-08687-20/80
PDP-11/23B Mounting Box Technical Manual	EK-1123B-TM-001
PDP-11/23B User's Guide	EK-1123B-UG-001
KDF11-B Field Maintenance Print Set	MP-01236

These documents can be ordered from:

Digital Equipment Corporation
Printing and Circulation Services
444 Whitney Street
Northboro, MA 01532

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

CHAPTER 2
INSTALLATION

2.1 INTRODUCTION

This chapter discusses the basic considerations and requirements for configuring and installing the KDF11-BA processor in LSI-11 systems using an extended LSI-11 bus backplane as well as existing LSI-11 systems using one of the LSI-11 bus backplanes. The items that must be considered fall into four basic categories.

1. Configuration of jumpers and switches for operation of user-selectable features.
2. Selection of an LSI-11 bus-compatible backplane and mounting box.
3. Selection of LSI-11 bus-compatible options and accessories.
4. Knowledge of system differences if replacing an LSI-11, LSI-11/2 or LSI-11/23 (KDF11-AA) processor with an LSI-11/23B (KDF11-BA) processor.

See Paragraph 1.9 for information on ordering documents referred to in this chapter.

2.2 JUMPER AND SWITCH CONFIGURATION

The KDF11-BA contains two DIP (dual in-line package) switch units (E102 and E114) and a number of jumpers that allow the user to select the module features desired. The location of the switch units and jumpers is shown in Figure 2-1. The boot/diagnostic switch unit (E102) consists of eight switches that let the user select boot and diagnostic programs. The second switch unit (E114) selects the baud rate for the console SLU (serial-line unit) and the second SLU. The module contains both wirewrap jumper stakes and soldered-in jumpers. The jumpers are divided into the following functional groups.

1. Test jumpers
2. CPU (central processor unit) option jumpers
3. Device selection jumpers
4. Boot and diagnostic ROM jumpers
5. SLU character format jumpers
6. Internal/external SLU clock jumpers
7. Bus grant continuity jumpers

2.2.1 Test Jumpers

The test jumpers described in the following paragraphs are used for tests performed by manufacturing and field service.

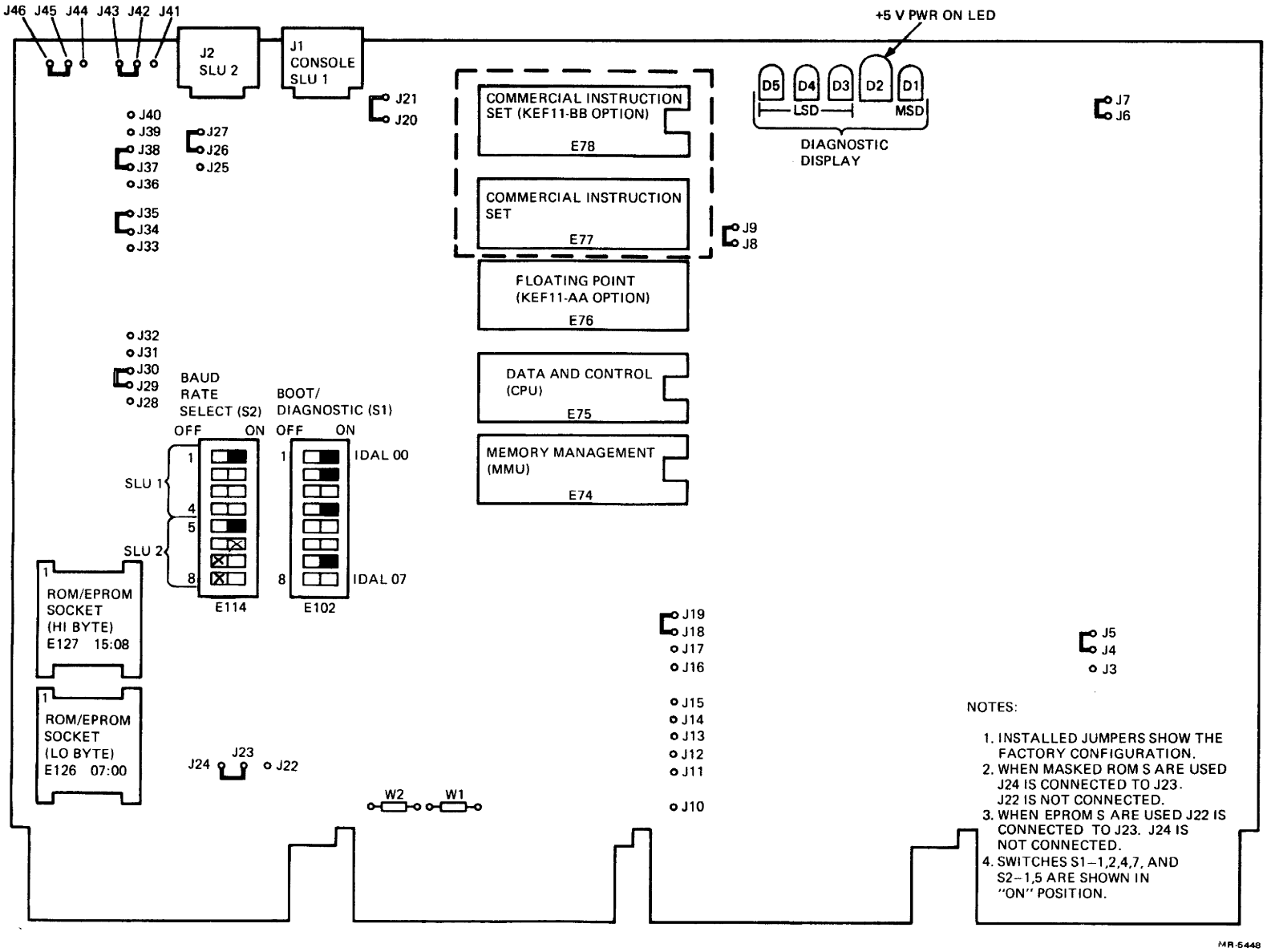


Figure 2-1 KDF11-BA Jumper, Switch, and Diagnostic Display Locations

2.2.1.1 Manufacturing Test Jumpers – Three wirewrap jumpers are provided for manufacturing tests. The jumpers are removed while the tests are performed and must be installed for normal operation. Table 2-1 lists the manufacturing test jumpers.

Table 2-1 Manufacturing Test Jumpers

Jumper From	To	Function
J6	J7	Connects the system oscillator to the CPU and LSI-11 bus timing circuits.
J8	J9	Connects the PHASE signal to the input of the F11 chip clock drivers.
J20	J21	Connects the baud rate crystal oscillator to the SLU baud rate generator and the −12 V charge pump circuit.

2.2.1.2 UART Test Jumper – For normal operation the bus initialize signal (BINIT) will clear the UARTs on the console and second SLUs. If a character is in either of the UARTs' buffers and a RESET instruction is executed before the character is read, the character is lost. The three jumper stakes (J33–J35) allow the console UART to be configured so it will be cleared by power-up and system restart only. Currently this feature is not used by DIGITAL in manufacturing or field service testing. Table 2-2 describes the jumper configuration for the UART test jumper.

CAUTION
Standard field service SLU diagnostics will FAIL if the reset disabled configuration is selected. Normal system and diagnostic operation requires that this feature not be selected.

Table 2-2 UART Test Jumper

Jumper From	To	Function	Reset Disabled	Normal Operation
J35	J34	Connects LINITF(1) H to the console SLU UART reset input.	R	I
J33	J34	Connects DCOKC2B L to the console SLU UART reset input.	I	R

R = removed; I = installed.

2.2.1.3 Field Service Test Jumper – This jumper allows field service personnel to check out the console terminal and its cable independently of the processor. When the jumper is installed in the test configuration, the serial input from the console is looped through the console SLU connector (J1) back to the console. Table 2-3 describes the jumper configuration for normal operation and field service testing.

Table 2-3 Field Service Test Jumper

Jumper From	To	Function	Field Service	Normal Operation
J27	J26	Connects the output of the console serial-line driver to the console serial-output line.	R	I
J25	J26	Connects the serial-line input from the console connector to the console connector serial-line output.	I	R

R = removed; I = installed.

2.2.2 CPU Option Jumpers

Four wirewrap stakes provide user-selectable features associated with the operation of the CPU. The ground stake can be connected to any combination of the other three stakes to select the available features. Two power-up mode stakes select one of four power-up modes. The halt/trap stake selects the halt/trap option.

2.2.2.1 Power-Up Mode Selection – The four power-up modes are selected by installing or removing in various combinations the wirewrap jumpers between jumper stakes J17 and J19 and the ground stake (J18). The jumper configurations for the modes are listed in Table 2-4.

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 for each of the four modes. The state of bus signal BHALT L is significant in power-up mode operation.

Table 2-4 Power-Up Mode Jumper Configurations

Jumper J18 to J19	Jumper J18 to J17	Mode	Name
R	R	0	PC@24, PS@26
R	I	1	Console ODT
I	R	2	Bootstrap
I	I	3	Extended microcode

R = removed; I = installed.

Power-Up Mode 0 – 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 it is not, 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, but is valid only when used with nonvolatile memory.

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

Power-Up Mode 2 – This mode causes the processor to generate internally a 16-bit bootstrap start address of 173000_8 (the conventional start address for DIGITAL systems). This address is loaded into the PC. The processor sets the PS to 340_8 (PS<7:5> = 7) to inhibit interrupts before the processor is ready for them. If BHALT L is asserted, the processor enters console ODT mode; if it is not, the processor begins execution by fetching an instruction from the location pointed to by the PC. This mode is useful for turn-key applications where the system automatically begins operation without operator intervention.

Power-Up Mode 3 – This mode causes the microcode to jump to optional control chip number 37_8, location 76_8, and begin microcode execution. This mode is reserved for future microcode expansion by DIGITAL 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.

2.2.2.2 Halt/Trap Option – 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 (J16) 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 where recovery from erroneous HALT instructions is desirable, such as unattended operation. Table 2-5 lists the halt/trap jumper functions for kernel and user processor modes.

Table 2-5 Halt/Trap Jumper Configuration

Jumper J18 to J16	Processor Mode	Function
R	Kernel	Processor enters console ODT microcode when it executes a HALT instruction.
I	Kernel	Processor traps to location 10_8 when it executes a HALT instruction.
X	User	HALT instruction decode results in a trap to location 10_8 regardless of the status of the halt/trap jumper.

R = removed; I = installed; X = "Don't care."

2.2.3 On-Board Device Selection Jumpers

Six wirewrap stakes on the KDF11-BA module are used to select which on-board peripheral devices are to be enabled or disabled. The ground stake can be connected to any combination of the other five stakes to obtain the desired configuration. The jumper functions are described in Table 2-6.

Table 2-6 On-Board Device Selection Jumpers

<table>
<tr><th>Stake Number</th><th>Stake Name</th><th>Function</th></tr>
<tr><td>J10</td><td>Ground</td><td>This wirewrap stake provides a ground source for the other five wirewrap stakes in this group.</td></tr>
<tr><td>J15</td><td>BDK DISJ L</td><td>When grounded, this signal disables the boot/diagnostic registers, the boot/diagnostic ROMs, and the line clock register.</td></tr>
<tr><td>J11</td><td>LTC ENBJ L</td><td>When grounded, this signal forces the line clock interrupt enable flip-flop to be set and allows the LSI-11 bus BEVNT signal to request program interrupts unconditionally.</td></tr>
<tr><td>J14</td><td>DL1 DISJ L</td><td>When grounded, this signal disables the console serial-line registers. When ungrounded, the device and vector addresses for the console SLU are the following.

Device Addresses | Interrupt Vectors
RCSR 17777560 | Receiver 060
RBUF 17777562 | Transmitter 064
XCSR 17777564
XBUF 17777566

NOTE
If DL1 DISJ L is grounded, the break-on-halt feature must also be disabled (Paragraph 2.2.5.3).</td></tr>
<tr><td>J13</td><td>DL2 DISJ L</td><td>When grounded, this signal disables the second serial-line registers. When ungrounded, the device and vector addresses for the second SLU are determined by the status of the DL2 ADRJ L jumper.</td></tr>
<tr><td>J12</td><td>DL2 ADRJ L</td><td>When DL2 ADRJ L is ungrounded, the second SLU device and its vector addresses are as follows.

Device Addresses | Interrupt Vectors
RCSR 17776500 | Receiver 300
RBUF 17776502 | Transmitter 304
XCSR 17776504
XBUF 17776506

When DL2 ADRJ L is grounded, the device and vector addresses are as follows.

Device Addresses | Interrupt Vectors
RCSR 17776540 | Receiver 340
RBUF 17776542 | Transmitter 344
XCSR 17776544
XBUF 17776546</td></tr>
</table>

2.2.4 Bootstrap/Diagnostic Switches and Jumpers

A 16-pin DIP switch pack (E102) and two jumpers on the KDF11-BA module provide switch-selectable bootstrap and diagnostic programs for hard disks and diskettes or the customer's own bootstrap program. The KDF11-BA will have BDV11 functionality only if the BDV11 2K × 8 diagnostic/bootstrap ROMs or EPROMs containing DIGITAL programs are installed in sockets E126 and E127. The switch and jumper functions are described in Paragraphs 2.2.4.1 and 2.2.4.2 and their locations are shown in Figure 2-1.

2.2.4.1 Bootstrap/Diagnostic Configuration Switches – Boot and diagnostic configuration register bits <07:00> reflect the status of the eight switches of the S1 switch pack (E102). Switches S1-1 through S1-4 are used to select a diagnostic and/or a bootstrap program. Switches S1-5 through S1-8 are used in conjunction with switches S1-3 and S1-4 to select the specific bootstrap program desired. The switch configurations when using the BDV11 2K × 8 diagnostic bootstrap ROMs (DIGITAL) are listed in Tables 2-7 and 2-8.

Table 2-7 Diagnostic/Bootstrap Program Selection

CDAL Bit	Switch Number	Switch Position	Function
00	S1-1	On	Execute CPU diagnostic upon power-up or restart.
01	S1-2	On	Execute memory diagnostic upon power-up or restart.
02	S1-3	On	DECnet boot (S1-4 through S1-7 are arguments*).
03	S1-4	On	Console test and dialogue (S1-3 Off).
03	S1-4	Off	Turn-key boot dispatched by S1-5 through S1-8 configuration (S1-3 Off).

* DECnet boot arguments are:

Boot Device†	Switch Positions S1-4	S1-5	S1-6	S1-7
DUV11	On	X	X	X
DLV11-E	Off	On	X	Off
DLV11-F	Off	On	X	On

† DLV11-E CSR = 17775610
DLV11-F CSR = 17776500
DUV11 CSR = 17760040 if there are no devices from 17760010 to 17760036

X = "Don't care."

Table 2-8 Bootstrap Program Selection

Device Mnemonic*	Switches: CDAL Bit:	S1–5 04	S1–6 05	S1–7 06	S1–8 07	Program Selected
DKn;n < 8*		Off	Off	Off	On	RK05 boot
DLn;n < 4		Off	Off	On	Off	RL01 or RL02 boot
DDn;n < 2		Off	Off	On	On	TU58 (SLU) at 776500 boot
DXn;n < 2		Off	On	Off	Off	RX01 boot
DYn;n < 2		Off	On	On	Off	RX02 boot

*n = unit number

All bootstrap programs other than the DECnet boots above are controlled by the bit patterns in switches S1-5 through S1-8. The bit patterns are described in Table 2-8. If the console test is selected (S1-4 On, S1-3 Off), the bootstrap program is controlled by a device mnemonic and unit number supplied by the console operator. These device mnemonics are also described in Table 2-8.

The console test prompts the operator with

```
XXXX.KW
START?
```

where XXXX is the decimal multiple of 1024 words of RAM found in the system when sized from 0 up in consecutive 1024-word increments. The first word of each 1024-word segment is read and written back to itself.

The console operator responses are a 2-character device mnemonic with a 1-digit octal unit number or one of two special single-character mnemonics. If no 1-digit unit number is specified, the unit 0 is selected. The response must be followed by a <CR> (carriage return). The special single-character mnemonics are

Y Use switch settings to determine boot device
N HALT – enter ODT microcode

2.2.4.2 Bootstrap/Diagnostic ROM Jumpers – Two 24-pin sockets (E126 and E127) are provided for the installation of 2K × 8 ROMs or EPROMs. When EPROMs are inserted into the two ROM sockets, +5 V must be applied to pin 21 of each socket. For all other ROMs used in this option, ROM address bit 13 (BTRA 13 H) must be applied to pin 21. This pin is a chip select input for 2K ROMs. Table 2-9 describes the jumper configurations when using ROMs or EPROMs. Figure 2-1 shows the location of jumper stakes J22, J23 and J24.

Table 2-9 ROM (or EPROM) Jumpers

Jumper From	To	Memory Type ROM	EPROM	Function
J24	J23	I	R	Connects BTRA 13 H to pin 21 of the two ROM sockets.
J22	J23	R	I	Connects +5 V to pin 21 of the two ROM sockets.

I = installed; R = removed.

2.2.5 Console SLU Switch and Jumper Configurations

Four switches of a 16-pin DIP switch pack (E114) and four jumpers provide user-selectable features associated with the operation of the console serial-line unit. The switch and jumper functions are described in Paragraphs 2.2.5.1 through 2.2.5.3 and Paragraph 2.2.7.

2.2.5.1 Console SLU Baud Rates – Switches 1–4 of the S2 switch pack (E114) select 1 of 16 possible SLU baud rates if the internal baud rate generator is used as the clock source. If the KDF11-BA is configured to operate the SLU with an external clock, the positions of these switches are meaningless. Paragraph 2.2.7 describes the jumper configuration for internal/external baud rate clock selection.

The SLU transmits and receives at the selected baud rate. Split baud operation is not provided. The switch configuration for selecting any one of the available baud rates is described in Table 2-10.

Table 2-10 Console SLU Baud Rate Selection

Switch Position S2-4	S2-3	S2-2	S2-1	Baud Rate
On	On	On	On	50
On	On	On	Off	75
On	On	Off	On	110
On	On	Off	Off	134.5
On	Off	On	On	150
On	Off	On	Off	300
On	Off	Off	On	600
On	Off	Off	Off	1200
Off	On	On	On	1800
Off	On	On	Off	2000
Off	On	Off	On	2400
Off	On	Off	Off	3600
Off	Off	On	On	4800
Off	Off	On	Off	7200
Off	Off	Off	On	9600
Off	Off	Off	Off	19200

2.2.5.2 Console SLU Character Formats – Five wirewrap stakes are used to select options for establishing the console SLU character format. The ground stake can be connected to any combination of the other four stakes to configure the character format for the following options.

- One or two stop bits
- Seven data bits plus parity
- Eight data bits without parity
- Odd or even parity

The jumper stake functions are described in Table 2-11 and the jumper configurations are described in Table 2-12.

Table 2-11 Console SLU Character Format Jumpers

Stake Number	Stake Name	Function
J38	Ground	This wirewrap stake provides a ground source for the other four wirewrap stakes in this group.
J39	DL1 CH7J L	When grounded, this signal causes the UART to transmit and receive 7-bit characters. Otherwise, the UART is formatted for 8-bit characters.
J37	DL1 ST1J L	When grounded, this signal causes the UART to transmit and receive one stop bit. Otherwise, it is formatted for two stop bits.
J36	DL1 PARJ L	When grounded, this signal enables UART parity generation and checking. Otherwise, parity is disabled.
J40	DL1 ODDJ L	When DL1 PARJ L and DL1 ODDJ L are both grounded, odd parity is selected. If only DL1 PARJ L is grounded, even parity is selected.

Table 2-12 Character Jumper Configurations

Jumper From	To J38	Character Format Option
J39	IN	7-bit characters
	OUT	8-bit characters
J37	OUT	Two stop bits
	IN	One stop bit
J36*	IN	Parity check enabled
	OUT	Parity check disabled
J40	IN	Odd parity if J36 is in.
	OUT	Even parity if J36 is in.

NOTE: If 8-bit characters (J39 OUT) are selected, parity check must be disabled (J36 OUT).

2.2.5.3 Break-on-Halt Jumpers – Two jumpers enable and disable the break-on-halt feature. If this feature is enabled, the detection of a break condition by the console UART causes the processor to halt and enter the on-line debugging technique (ODT) microcode. If this feature is disabled, there is no response to the break condition. Table 2-13 lists the jumper configurations for selecting the break-on-halt feature.

Table 2-13 Break-on-Halt Jumper Configuration

Jumper From	To	Function	Break Feature Enabled	Disabled
J5	J4	Connects ground to RQ HLT H.	R	I
J3	J4	Connects DL1 FE H to RQ HLT H.	I	R

R = removed; I = installed.
J3 = DL1 FE H
J4 = RQ HLT H
J5 = Ground

2.2.6 Second SLU Switch and Jumper Configurations

The second SLU is configured in the same manner as the console SLU except that a different set of switches and jumpers are used to select the available SLU features. The switch and jumper functions for the second SLU are described in Paragraphs 2.2.6.1 and 2.2.6.2.

2.2.6.1 Second SLU Baud Rates – Switches 5 through 8 of the S2 switch pack (E114) select 1 of 16 baud rates for the second SLU, if the internal baud rate generator is used as the clock source. The second SLU will transmit and receive at the same selected baud rate. The switch configurations for selecting any of the available baud rates are listed in Table 2-14.

Table 2-14 Second SLU Baud Rate Selection

Switch Position S2-8	S2-7	S2-6	S2-5	Baud Rate
On	On	On	On	50
On	On	On	Off	75
On	On	Off	On	110
On	On	Off	Off	134.5
On	Off	On	On	150
On	Off	On	Off	300
On	Off	Off	On	600
On	Off	Off	Off	1200
Off	On	On	On	1800
Off	On	On	Off	2000
Off	On	Off	On	2400
Off	On	Off	Off	3600
Off	Off	On	On	4800
Off	Off	On	Off	7200
Off	Off	Off	On	9600
Off	Off	Off	Off	19200

2.2.6.2 Second SLU Character Formats – Five wirewrap stakes are used to select options for establishing the second SLU character format. The ground stake can be connected to any combination of the other four stakes to configure the character format for the following options.

- One or two stop bits
- Seven data bits plus parity
- Eight data bits without parity
- Odd or even parity

The jumper stake functions are described in Table 2-15 and the jumper configurations are listed in Table 2-16.

Table 2-15 Second SLU Character Format Jumpers

Stake Number	Stake Name	Function
J30	Ground	This wirewrap stake provides a ground source for the other four wirewrap stakes in this group.
J31	DL2 CH7J L	When grounded, this signal causes the UART to transmit and receive 7-bit characters. Otherwise, the UART is formatted for 8-bit characters.
J29	DL2 ST1J L	When grounded, this signal causes the UART to transmit and receive one stop bit. Otherwise, it is formatted for two stop bits.
J28	DL2 PARJ L	When grounded, this signal enables UART parity generation and checking. Otherwise, parity is disabled.
J32	DL2 ODDJ L	When DL2 PARJ L and DL2 ODDJ L are both grounded, odd parity is selected. If only DL2 PARJ L is grounded, even parity is selected.

Table 2-16 Character Jumper Configurations

Jumper From	To J30	Character Format Option
J31	IN	7-bit characters
	OUT	8-bit characters
J29	OUT	Two stop bits
	IN	One stop bit
J28	IN	Parity check enabled
	OUT	Parity check disabled
J32	IN	Odd parity if J28 is in.
	OUT	Even parity if J28 is in.

2.2.7 Internal/External SLU Clock Jumpers

Two sets of jumpers are provided to select an internal or external clock for the console SLU and the second SLU. If the internal clock jumpers are installed, the SLU clocks are obtained from the internal baud rate generator. When the external clock jumpers are installed, external clocks are routed to the SLUs through pin 1 of the J1 and J2 SLU connectors. Table 2-17 lists the internal/external SLU clock jumper configurations.

Table 2-17 Internal/External SLU Clock Jumper Configurations

Jumper From	To	Function	Selected Clock Internal	External
J43	J42	Connects internal baud rate generator to the console SLU UART. (Normal configuration)	I	R
J41	J42	Connects external clock to the console SLU UART.	R	I
J46	J45	Connects internal baud rate generator to the second SLU UART. (Normal configuration)	I	R
J44	J45	Connects external clock to the second SLU UART.	R	I

R = removed; I = installed.

2.2.8 Bus Grant Continuity Jumpers

Two jumpers must be installed when the KDF11-BA is used in an LSI-11/LSI-11 bus backplane. An LSI-11/LSI-11 bus backplane (e.g., an H9275 or H9270) is one that carries the LSI-11 bus signals on backplane rows C and D as well as rows A and B. The jumpers provide continuity for the interrupt acknowledge (BIAK) and direct memory access grant (BDMG) LSI-11 bus signals. The jumpers are described in Table 2-18.

Table 2-18 Bus Grant Continuity Jumpers

Jumper*	Function
W1	Connects backplane pins CM2 and CN2, providing continuity for BIAK L.
W2	Connects backplane pins CR2 and CS2, providing continuity for BDMG L.

*Must be installed when the KDF11-BA is used in an LSI-11/LSI-11 bus backplane; otherwise, the jumper installation is optional.

NOTE

If the KDF11-BA is installed in an LSI-11/CD backplane (H9273 or H9276) and the W1 and W2 jumpers are installed, pin CM1 is shorted to CN1 and pin CR1 is shorted to CS1 on slot 2.

2.3 FACTORY SWITCH AND JUMPER CONFIGURATIONS

Users may reconfigure the module jumpers and switches to select the KDF11-BA options required for the particular system application. All switches and all jumpers except those jumpers reserved for manufacturing and field service testing may be reconfigured. Therefore, the factory configuration as shipped is described below to assist users in determining the jumper and switch changes that are required to select the module options for their systems. Table 2-19 lists the factory jumper configurations. Tables 2-20 and 2-21 list the bootstrap/diagnostic switch and SLU baud rate switch configurations, respectively.

Table 2-19 Factory Jumper Configurations

Jumper From	To	Jumper State	Function
			The manufacturing and field service test jumpers are described below in Paragraph 2.2.1.
J6	J7	I	Master clock; enables internal oscillator.
J8	J9	I	Phase; connects signal to F11 chip clock drivers.
J20	J21	I	XTAL; connects baud rate oscillator.
J35	J34	I	LINITF (1) H; connects reset to console UART.
J33	J34	R	Installed reset disabled test feature (only after removing jumper J35-J34).
J27	J26	I	Connects console SLU serial output to connector J1.
J25	J26	R	Installed for field service wraparound testing (only after removing jumper J27-J26).
			The CPU option jumpers are described below in Paragraph 2.2.2.
J19 J17	J18 J18	I R	Power-up mode 2 (jumper J19-J18 installed; jumper J17-J18 removed) causes the processor to begin executing the bootstrap code at start address 173000.
J16	J18	R	Processor enters console ODT microcode when it executes a kernal mode HALT instruction.
			The on-board device selection jumpers are described in Paragraph 2.2.3.
J11	J10	R	LTC ENJ L. BEVENT can request interrupts only if the processor program has set bit 6 of the line clock register (17777546).
J12 J13	J10 J10	R R	The second SLU is enabled with an RCSR address of 17776500 and interrupt vector addresses of 300 and 304.
J14	J10	R	The console SLU is enabled.

R = removed; I = installed.

Table 2-19 Factory Jumper Configurations (Cont)

Jumper From	To	Jumper State	Function
J15	J10	R	The BDV ROMs and registers, as well as the line clock register, are enabled.
			The boot and diagnostic ROM jumpers are described in Paragraph 2.2.4.2.
J22	J23		NOTE: When ROMs are used, jumper J22-J23 is installed and jumper J24-J23 is removed.
J24	J23		NOTE: When EPROMs are used, jumper J22-J23 is removed and jumper J24-J23 is installed.
			The console SLU character formats are described in Paragraph 2.2.5.2.
J36	J38	R	Console SLU parity check is disabled.
J37	J38	I	Console SLU character contains one stop bit.
J39	J38	R	Console SLU character contains eight bits.
J40	J38	R	No effect; console parity already disabled.
			The break-on-halt jumpers are described in Paragraph 2.2.5.3.
J3 J5	J4 J4	R I	Break-on-halt feature is disabled. The break key on the console SLU does not halt the processor. This feature may be enabled by removing jumper J5-J4 and then installing jumper J3-J4.
			The second SLU character formats are described in Paragraph 2.2.6.2.
J28	J30	R	Second SLU parity check is disabled.
J29	J30	I	Second SLU character contains one stop bit.
J31	J30	R	Second SLU character contains eight bits.
J32	J30	R	No effect; second SLU parity already disabled.
			The internal/external SLU clock jumpers are described in Paragraph 2.2.7.
J41 J43	J42 J42	R I	The on-board baud rate generator is connected to the console SLU. The external clock input from connector J1 is disabled.

R = removed; I = installed.

Table 2-19 Factory Jumper Configurations (Cont)

Jumper From	To	Jumper State	Function
J44	J45	R	The on-board baud rate generator is connected to the second SLU.
J46	J45	I	The external clock input from connector J2 is disabled.
			The bus grant continuity jumpers are described in Paragraph 2.2.8.
W1		I	Provides bus grant continuity for the BIAK signal.
W2		I	Provides bus grant continuity for the BDMG signal.

R = removed; I = installed.

Table 2-20 Bootstrap/Diagnostic Factory Switch Configurations

Switch S1 (E102) Number	Position	Function*
1	On	Execute CPU diagnostic
2	On	Execute memory diagnostic
3	Off	DECnet boot disabled
4	On	Console test and dialogue
5	Off	–
6	Off	–
7	On	RL01/RL02 bootstrap program
8	Off	–

*With the switch configurations shown, the KDF11-BA, upon power-up or restart, will execute the CPU diagnostic, the memory diagnostic, and then enter the console test. If the operator wishes to terminate the memory diagnostic and immediately enter the console test, control/C must be entered on the console terminal.

Table 2-21 SLU Baud Rate Factory Switch Configurations

Switch S2 (E114) Number	Position	Function
1	On	Console SLU set for 9600 baud per Table 2-10.
2	Off	
3	Off	
4	Off	
5	On	Second SLU set for 9600 baud per Table 2-14.
6	Off	
7	Off	
8	Off	

2.4 MODULE CONTACT FINGER IDENTIFICATION

Digital Equipment Corporation's plug-in modules, including the KDF11-BA, all use the same contact (pin) identification system. Figure 2-2 shows the contact finger identification for a typical quad-height module. The LSI-11 bus signals are carried on rows A and B. Each row contains 36 lines (the component and solder sides of the circuit board having 18 lines each).

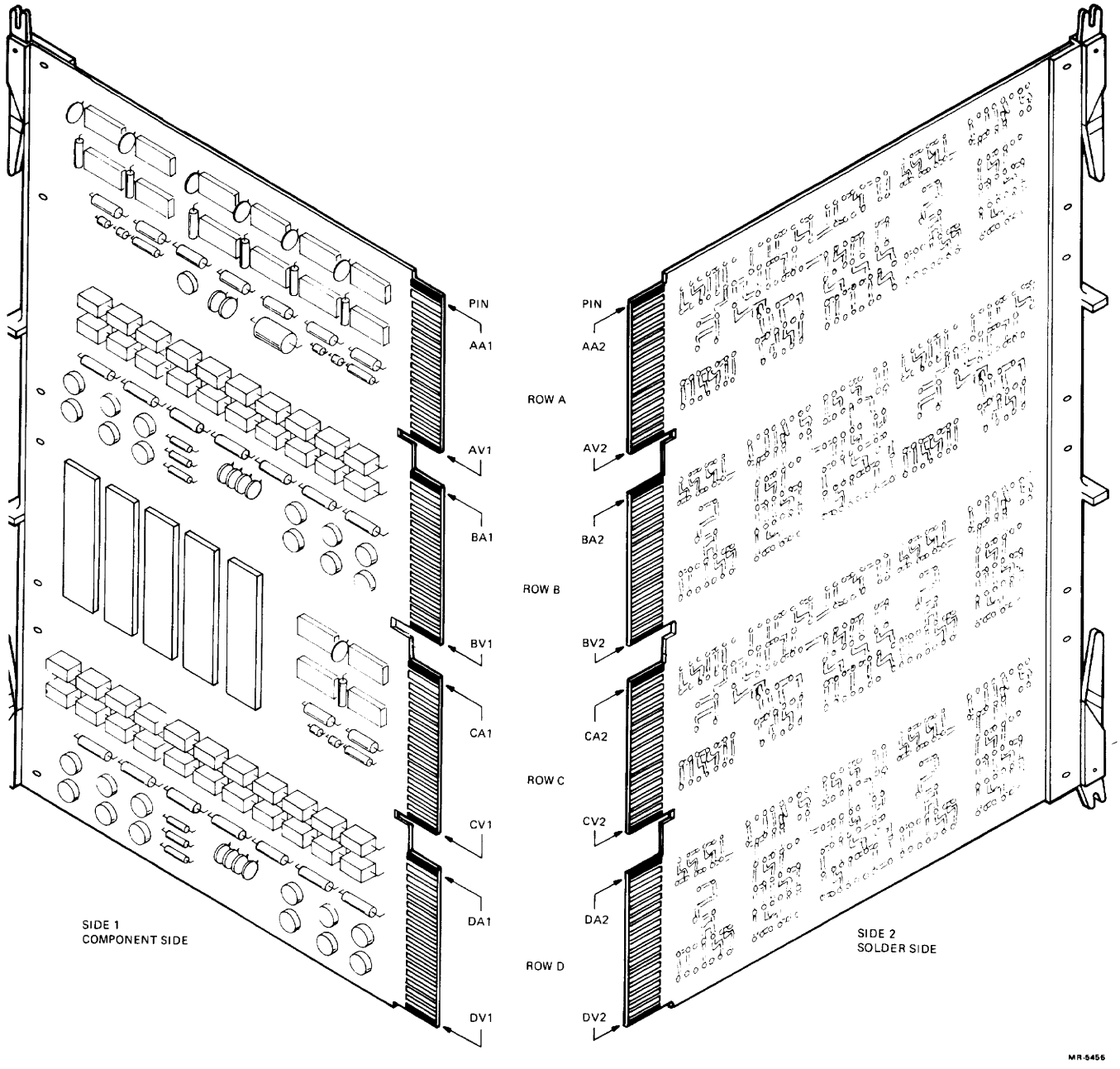
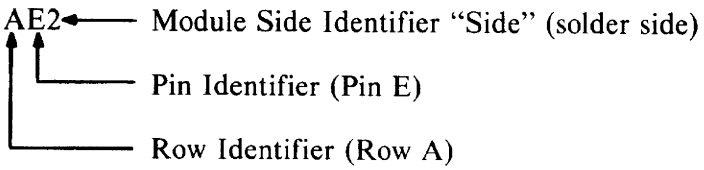


Figure 2-2 Quad Module Contact Finger Identification

Row A, shown in Figure 2-2, includes 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 particular pins on a side of a slot. A typical pin is designated as follows.



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

2.5 BACKPLANE PIN ASSIGNMENTS AND THEIR KDF11-BA UTILIZATION

When configuring a system with the KDF11-BA, the module may be inserted in one of several available backplanes. (Refer to Paragraph 2.6.1 for information on the types of backplanes available.) As an example, Figure 2-3 shows the pin identifications of an H9276 backplane. Individual connector pins shown are viewed from the module insertion side. Only pins for one slot location are shown in detail. This pin pattern is repeated 36 times on the backplane, allowing the user to install several double-height or quad-height modules.

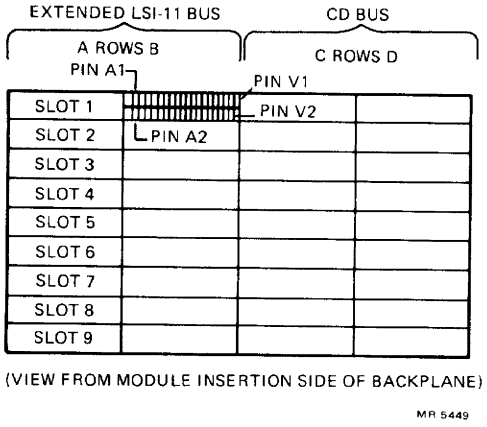


Figure 2-3 H9276 Backplane Pin Identifications

The KDF11-BA backplane pin assignments for rows A and B (extended LSI-11 bus signals) add four BDAL lines that extend the physical address space to four megabytes. The extended bus pin assignment additions are listed in Table 2-22. (Backplane pin assignment and signal pin functions for the remaining pins on rows A and B are described in Appendix F.) A comparison of the KDF11-BA, KDF11-AA, KD11-HA, and KD11-F processors' backplane pin assignments appears in Appendix D.

2.6 HARDWARE OPTIONS

KDF11-BA systems can be configured with a variety of backplanes, power supplies, enclosures, memories, peripherals, etc. Figure 2-4 shows a typical configuration for a KDF11-BA system with 512KB bytes of memory capacity.

Table 2-22 KDF11-BA Extended Address Lines

Bus Pin	Signal Mnemonic	Signal Function
BC1	BDAL 18 L	Data/address line 18
BD1	BDAL 19 L	Data/address line 19
BE1	BDAL 20 L	Data/address line 20
BF1	BDAL 21 L	Data/address line 21

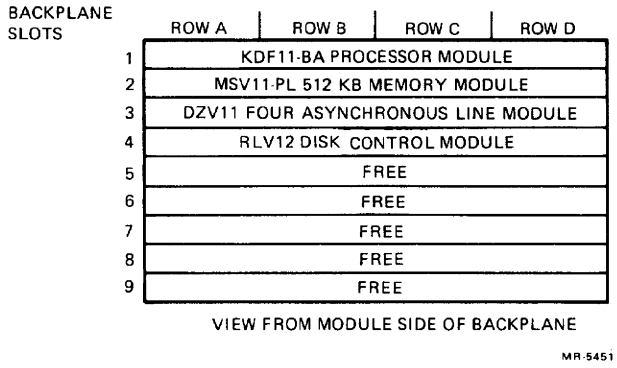


Figure 2-4 Typical KDF11-BA 512K-Byte System

2.6.1 Backplanes

The KDF11-BA is designed to run in any LSI-11 bus-compatible backplane that accepts quad-height modules. The KDF11-BA provides 18-bit addressing in backplanes that feature the traditional LSI-11 bus, or 22-bit addressing in backplanes that feature the extended LSI-11 bus. The following LSI-11 bus and extended LSI-11 bus backplanes are available.

- H9276 – A 9-slot × 4-row backplane that supports 22-bit addressing for up to nine quad- or dual-height modules. The AB slots are bused in accordance with the wiring scheme of the extended LSI-11 bus; the CD slots are bused in accordance with the wiring scheme of the CD bus.

- H9273 – A 9-slot × 4-row backplane that supports 18-bit addressing for up to nine quad- or dual-height modules. The AB slots are bused in accordance with the wiring scheme of the LSI-11 bus; the CD slots are bused in accordance with the wiring scheme of the CD bus.

- H9275 – A 9-slot × 4-row backplane that supports 22-bit addressing. Each slot may contain one quad- or two dual-height modules. The AB and CD slots are bused in accordance with the wiring scheme of the extended LSI-11 bus.

- H9270 – A 4-slot × 4-row backplane that supports 18-bit addressing. Each slot may contain one quad- or two dual-height modules. The AB and CD slots are bused in accordance with the wiring scheme of the LSI-11 bus.

- DDV11-B – A 9-slot × 4-row backplane that supports 18-bit addressing. The AB and CD slots are bused in accordance with the wiring scheme of the LSI-11 bus. The EF slots are available for user-defined interconnections.

Refer to the *PDP-11/23B Mounting Box Technical Manual* for a complete description of the H9276 backplane and the *Microcomputer Interfaces Handbook* for a complete description of the other backplanes listed.

2.6.2 Enclosures

The KDF11-BA may be installed in a variety of enclosures, including, but not limited to, the following.

- BA11-S Mounting Box – Contains the H9276 backplane and the H7861 power supply. It supports 22-bit addressing for up to nine quad- or dual-height modules. The H7861 power supply provides 36 A at +5 V and 5 A at +12 V.

- BA11-N Mounting Box – Contains the H9273 backplane and the H786 power supply. It supports 18-bit addressing for up to nine quad- or dual-height modules. The H786 power supply provides 22 A at +5 V and 11 A at +12 V.

- BA11-M Mounting Box – Contains the H9270 backplane and the H780 power supply. It supports 18-bit addressing for four slots, each of which may contain one quad- or two dual-height modules. The H780 power supply provides 18 A at +5 V and 3.5 A at +12 V.

Refer to the *PDP-11/23B Mounting Box Technical Manual* for a complete description of the BA11-S mounting box and the *Microcomputer Interfaces Handbook* for a complete description of the BA11-N and BA11-M.

2.6.3 Memory Modules

The KDF11-BA is compatible with a wide variety of memories, including, but not limited to, the ones that follow.

- MSV11-P Quad-Height Memory Module – Provides up to 512K bytes of 22-bit addressable memory.

- MSV11-L Dual-Height Memory Module – Provides up to 256K bytes of 22-bit addressable memory.

- MSV11-D Dual-Height Memory Module – Provides up to 64K bytes of 18-bit addressable memory.

The MSV11-B memory module, which does not have on-board refresh logic and only decodes 16-bit addresses, is not recommended for use with the KDF11-BA.

2.6.4 Peripheral Options

The KDF11-BA is designed to be compatible with all peripheral options designed to the LSI-11 bus specification. However, it is incompatible with two types of older options used in systems that support 22-bit addressing.

1. Direct Memory Access (DMA) devices that provide 18-bit DMA addresses, such as the RLV11, RXV21, and DRV11-B, can only access up to 256K bytes of memory.

2. Peripheral devices that use pins BC1, BD1, BE1, and/or BF1 for test signals cannot be used in extended LSI-11 bus backplanes that bus these pins for BDAL21–18 L.

The RLV12 disk controller is a single quad-height module that provides 22-bit addressing. It replaces the RLV11 as an interface to the RL01 and RL02 disk drives.

Non-DMA peripheral devices are generally not affected by 22-bit addressing because they monitor BBS7 L instead of BDAL21–13 to decode I/O Page addresses. Refer to the *PDP-11/23 Mounting Box Manual* for a listing of peripheral options compatible with the extended LSI-11 bus backplanes.

2.7 SYSTEM DIFFERENCES

A number of minor differences exist between the KDF11-AA or KDF11-BA processors and the LSI-11 (KD11-F) or LSI-11/2 (KD11-HA) processors. The following is a list of these differences.

1. The KDF11-BA and KDF11-AA do not have a boot loader in console ODT microcode.

2. The KDF11-BA and KDF11-AA console ODT functions are a subset of the KD11-F and KD11-HA ODT functions.

3. KDF11-BA, KDF11-AA, and KD11-HA do not perform memory refresh.

4. The EVENT line is on level 6 for the KDF11-BA and KDF11-AA; KD11-F and KD11-HA have the EVENT line on level 4.

5. The REV11-C refresh/boot module cannot be used to boot the KDF11-BA system.

Refer to Appendices C, D, and E for additional comparisons among these LSI-11 processors.

2.8 MODULE INSTALLATION PROCEDURE

Certain guidelines should be followed when installing or replacing a KDF11-BA module.

1. Verify dc power before inserting the module in a backplane.

2. Ensure that no dc power is applied to the backplane when removing or inserting the module.

3. Verify the configuration of option jumpers and switches as specified under Paragraphs 2.2 and 2.3.

4. It is recommended that a single switch be used to apply +5 V and +12 V to the system.

The KDF11-BA module's response to power-up depends on the power-up mode, as detailed in Table 2-23.

The following diagnostics are available for checking out the KDF11-BA module.

- CJKDB CPU Diagnostic – Tests the basic instruction set, EIS, and processor traps.

- CJKDA MMU Diagnostic – Checks out the memory management and extended addressing functions.

- CVM8A BDV11 Diagnostic – Checks out KDF11-BA BDV functionality.
- CJDLA SLU Diagnostic – Checks out the KDF11-BA serial-line units.
- CJKDC and CJKDD Floating-Point Diagnostics – Check out the floating-point option.
- CJKDH CIS Diagnostic – Checks out the CIS option.

Table 2-23 Console Power-Up Printout (or Display)*

Power-up Mode	BHALT L State	Console Response
0	Unasserted	Processor will execute the program using the contents at location 24_8 as the PC value.
	Asserted	Terminal will print and enter micro-ODT.
1	Unasserted	Terminal will print out a random 6-digit number, which is the contents of the program counter.
	Asserted	Terminal will print out a random 6-digit number, which is the contents of the program counter, and enter micro-ODT.
2	Unasserted	Processor will execute the program at location 773000.†
	Asserted	Terminal will print out 173000 and enter micro-ODT.†
3	Unasserted	Mode 3 causes microcode to jump to optional control chip 37_8, location 76_8, and begin microcode execution. This mode is reserved for future use by DIGITAL and is not recommended for customer use. If this mode is erroneously selected, the processor will treat it as a reserved instruction trap to location 10_8.
	Asserted	A normal printout-terminal will print out the contents of memory location 10_8 and enter micro-ODT.

*The terminal printout consists of six octal digits as specified in the table, followed by a carriage return, line feed, and @ prompt in all cases.
†Normal mode for use with the BDV11 bootstrap/diagnostic ROMs.

CHAPTER 3
CONSOLE ON-LINE DEBUGGING TECHNIQUE (ODT)

3.1 INTRODUCTION

A portion of the microcode in the KDF11-BA processor emulates the capability normally found on a programmer's console. Since the KDF11-BA does not have a programmer's console (one with lights and switches) or a console switch register at bus address 777570, the terminal at the standard bus address of 777560 is used to perform console functions. Communication between the processor and the user is via a stream of ASCII characters interpreted by the processor as console commands. The console terminal addresses 777560 through 777566 are generated in microcode and cannot be changed.

This feature is called the microcode on-line debugging technique, or micro-ODT. The KDF11-BA micro-ODT accepts 18-bit addresses, allowing it to access 248K bytes of memory, plus the 8K-byte I/O page. A PDP-11 software version of ODT, macro-ODT, is necessary to access memory beyond these limits. Macro-ODT provides a more sophisticated range of debugging techniques, including access of memory locations by virtual address.

The differences in use of console ODT in the KDF11-BA as compared with that in the KD11-F (LSI-11) and the KD11-HA (LSI-11/2) are listed in Appendix E.

3.2 TERMINAL INTERFACE

The hardware interface between a terminal (serial-line unit) and ODT is the on-board console serial-line unit. The terminal is connected to the serial-line unit via connector J1 on the module. Refer to Paragraph 5.14 for a description of the console serial-line unit.

3.3 CONSOLE ODT ENTRY CONDITIONS

The ODT console mode can be entered by the following ways.

1. Execution of a HALT instruction in kernel mode, provided the HALT TRAP jumper (J16 to J18) is not installed.

2. Assertion of the BHALT signal on the bus. Note that the signal must be asserted long enough that it is seen at the end of a macroinstruction by the SERVICE state in the processor. BHALT is level-triggered, not edge-triggered. Typically, BHALT remains asserted until the processor enters ODT.

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

4. From the console serial-line unit if the halt-on-break feature is enabled. Refer to Paragraph 2.2.5.3.

NOTE

Unlike the KD11-F and KD11-HA, the KDF11-BA does not enter console ODT upon occurrence of a double bus error (for example, when R6 points to nonexistent memory during a bus timeout trap). The KDF11-BA creates a new stack at location 2 and continues to trap to 4. If a bus timeout occurs while getting an interrupt vector, the KDF11-BA ignores it and continues execution of the program, whereas in such case the KD11-F and KD11-HA enter console ODT. Refer to Appendix E for a listing of the different ways certain processors interpret the same console ODT commands.

ODT causes the following processor initialization upon entry.

1. Performs a DATI from RBUF (input data buffer at 777562_8) and then ignores the character present in the buffer. This operation prevents the ODT from interpreting erroneous characters or user program characters as a command.

2. Prints a <CR> and <LF> on the console terminal.

3. Prints the contents of the PC (program counter R7) in six digits.

4. Prints a <CR> and <LF>.

5. Prints the prompt character @.

6. Enters a wait loop for the console terminal input. The DONE flag (bit 7) in the RCSR at 777560_8 is constantly being tested via a DATI by the processor for a 1. If bit 7 is a 0, the processor keeps testing.

3.4 ODT OPERATION OF THE CONSOLE SERIAL-LINE INTERFACE

The processor's microcode operates the serial-line interface in half-duplex mode by using program I/O techniques rather than interrupts. This means that when the ODT microcode is busy printing characters using the output side of the interface, the microcode is not monitoring the input side for incoming characters. Any characters coming in while the ODT microcode is printing characters are lost. Overrun errors detected by the universal asynchronous receiver/transmitter (UART) will be ignored because the microcode does not check any error bits in the serial-line interface registers.

Therefore, the user should not "type ahead" to ODT because those characters will not be recognized. More importantly, if another processor is at the end of the serial line, it must obey half-duplex operation. In other words, no input characters should be sent from the console terminal until the processor's ODT output has finished. This restriction does not pertain to echoed characters, however.

3.4.1 Console ODT Input Sequence

The input sequence for ODT follows. (Upon entry to ODT, the RBUF register at 777562 is read but the character is ignored to prevent the character from being interpreted as a command by the console ODT.)

1. Test RCSR bit 7 (DONE flag) of RCSR at 777560_8 using a DATI bus cycle; if it is a 0, continue testing.

2. If RCSR bit 7 is a 1, read the low byte of RBUF at 777562_8 using a DATI bus cycle.

3.4.2 Console ODT Output Sequence

The output sequence of ODT is as follows.

1. Test bit 7 (DONE flag) of the XCSR at 777564_8 using a DATI busy cycle; if it is a 0, continue testing.

2. If XCSR bit 7 is a 1, write to the XBUF at 777566_8 using a DATO bus cycle. The desired character is in the low byte. The data in the high byte is undefined and is ignored by the serial-line interface.

If the interrupt enable (bit 6) in the XCSR is a 1, an interrupt will be created to the software when the proceed (P) console ODT command is used. If a go (G) command is used, all interrupt enables in peripherals are cleared and an interrupt will not occur.

3.5 CONSOLE ODT COMMAND SET

The ODT command set is listed in Table 3-1 and described in Paragraphs 3.5.1 through 3.5.9. The commands are a subset of ODT-11 and use the same command characters. ODT has 10 internal states. Each state recognizes certain characters as valid input and responds with a question mark (?) to all others.

Table 3-1 Console ODT Commands

Command	Symbol	Function
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 execution of a program.
Proceed	P	Resumes execution of a program.
Binary dump	Control-shift-S	Manufacturing use only.
(Reserved)	H	Reserved for DIGITAL use.

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 by ODT (e.g., <CR>) have the parity bit equal to 0. All commands are echoed except for <LF>.

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 the following paragraphs, refers to a bus address, processor register, or processor status word (PS).

The descriptions of the ODT commands include examples of the printouts that the processor will output to the console terminal in response to the commands entered by the user. In the examples given, the processor output is underlined.

3.5.1 / (ASCII 057) - Slash

This command is used to open a bus address, processor register, or processor status word and is normally preceded by other characters that specify a location. In response to /, ODT will print the contents of the location (six characters) and then a space (ASCII 40). After printing is complete, ODT will wait 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: @00001000/012525 <SPACE>

where:

@ = ODT prompt character.

00001000 = octal location in the QBus 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 ODT.

The / command can be used without a location specifier to verify the data just entered into a previously opened location. The / produces this result only if it is entered immediately after a prompt character. A / issued immediately after the processor enters ODT mode will cause ? <CR>, <LF>, to be printed because a location has not yet 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 was correctly entered into memory.

3.5.2 <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> will close 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>, ODT printed <CR>,<LF>, and @.

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

@

In this case, the user desired to change R1. The new data, 1234, was entered before the <CR>. ODT deposited the new data into the open location and then printed <CR>,<LF>, and @. ODT echoes the <CR> entered by the user before it prints <CR>, <LF>, and @.

3.5.3 <LF> (ASCII 12) – Line Feed

This command is used to close an open location and then open the next contiguous location. Bus addresses and processor registers will be incremented by two and one, respectively. If the PS is open when an <LF> is issued, it will be closed and <CR>, <LF>, @ will be printed; no new location will be 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, ODT closed R2 and then opened R3. When a user has the last register, R7, open, and issues <LF>, ODT will "roll over" to the first register, R0. When the user has the last bus address of a 32K-word open segment and issues <LF>, ODT will open the first location of that segment. If the user wishes to cross the 32K-word boundary, he/she must reenter the address for the desired 32K-word segment (i.e., ODT is modulo 32K words).

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

@R0/123456 <SPACE>

or

Example: @577776/000001 <SPACE> <LF> <CR> <LF>

@477776/125252 <SPACE>

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

3.5.4 $ (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 the processor register specified. The $ character is recognized to be compatible with ODT-11, and the R character was introduced for its being one key stroke representative of its function.

Examples: @$0 /000123 <SPACE>

@R7/000123 <SPACE> <LF>

@R0/054321 <SPACE>

If more than one character (digit or S) follows the R or $, ODT will use the last character as the register designator. An exception: if the last three digits equal 077 or 477, ODT will open the PS rather than R7.

3.5.5 S (ASCII 123) – Processor Status Word Designator

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

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

@/000010 <SPACE>

Note that the trace bit (bit 4) of the processor status word cannot be modified by the user. This is so in order that PDP-11 program debugging utilities (e.g., ODT-11), which use the T bit for single-stepping, will not be accidentally harmed by the user. If the user issues an <LF> while the processor status word is open, the word is closed and ODT will print a <CR>, <LF>, @: no new location is opened in this case.

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

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

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

3. The PS, and FPS (floating-point status) register will be cleared to 0.

4. The LSI-11 bus is initialized by the processor asserting BINIT L for 12.6 μs, negating BINIT L, and then waiting for 110 μs.

5. The service state is entered by the processor. Anything to be serviced 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 a program, he/she issues a G and then successive P commands, all done with the BHALT L bus signal asserted.

3.5.7 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 machine state visible to the programmer is altered using this command.

Example: @P

Program execution resumes at the place pointed to by R7. After the P is echoed, the ODT state is left and the processor immediately enters the state to fetch the next instruction. If a HALT request is asserted, it is recognized at the end of the instruction (during the service state) and the processor will enter the ODT state. Upon entry, the contents of the PC (R7) will be printed. In this fashion, a user can single-step through a program and get a PC "trace" displayed on his/her terminal.

3.5.8 Control-Shift-S (ASCII 23) – Binary Dump

This command is used for manufacturing test purposes and is not a normal user command. It is intended to display a portion of memory more efficiently than the / and <LF> commands do. The protocol is as follows.

1. After a prompt character, 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 ODT will interpret as a starting address. These two bytes are not echoed. The first byte specifies starting address <15:8> and the second byte specifies starting address <7:0>. Bus address bits <21:16> are always forced to 0; the DUMP command is restricted to the first 32K words of address space.

3. After the second address byte has been received, ODT outputs 10_8 bytes to the serial line, starting at the address previously specified. When the output is finished, ODT will print <CR>, <LF>, @.

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

3.5.9 Reserved Command

An ASCII H (110) is reserved for future use by DIGITAL. If it is accidentally typed, ODT will echo the H and print a prompt character rather than a ?, which is the invalid character response. No other operation is performed.

3.6 KDF11-BA ADDRESS SPECIFICATION

The KDF11-BA micro-ODT accepts 18-bit addresses, allowing it to access 248K bytes of memory, plus the 8K-byte I/O page. All I/O page addresses must be entered by users with a full 18 bits specified. For example, if a user wishes to open the RCSR of the SLU (serial-line unit), he/she must enter 777560, not 177560.

3.6.1 Processor I/O Addresses

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

The MMU contains status registers and PAR/PDR pairs. These registers can be accessed from ODT by entering their bus address.

Example: @777572/000001 <SPACE>

In this case, memory management status register 0 is opened to show the memory management enable bit set.

The FP11 accumulators, which are also in the MMU chip, cannot be accessed from ODT. Only FP11 instructions can access these registers.

3.6.2 Stack Pointer Selection

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.

Similarly, if a program is operating in user mode (PS<15:14> = 11), the R6 command accesses the user stack pointer. If a different 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> was 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 within the processor is accessed. This register is reserved for future use by DIGITAL.

3.6.3 Entering of Octal Digits

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

3.6.4 ODT Timeout

If the user specifies a nonexistent address, ODT will respond to the bus timeout by printing ?, <CR>, <LF>, @.

3.7 INVALID CHARACTERS

In general, any character that ODT does not recognize during a particular sequence is echoed (with the exception of ASCII codes 0, 2, 10, and 12 as noted earlier) and ODT will print ?, <CR., <LF>, @. ODT has 10 internal states, with each state having its own set of valid input characters. Some commands are allowed only when in certain states or sequences; thus an attempt has been made to lower the probability of a user's unconsciously destroying himself by pressing the wrong key. Table 3-2 defines the ODT states and valid input characters.

Table 3-2 Console ODT States and Valid Input Characters

State	Example of Terminal Output	Valid Input
1	@ R, S G P Control-shift-S	0–7
2	@R or @$ S	0–7
3	@1000/123456 <CR> <LF>	0–7
4	@R1/123456 <CR> <LF>	0–7
5	@1000 / G	0–7
6	@R1 or @RS S /	0–7
7	@1000/123456 1000 <CR> <LF>	0–7
8	@R1/123456 1000 <CR> <LF>	0–7
9*	@	/
10	@ Control-shift-S	2 binary bytes

*Indicates previous location was opened.

CHAPTER 4
EXTENDED LSI-11 BUS

4.1 INTRODUCTION

The processor, memory and I/O devices communicate via signal lines that constitute the extended LSI-11 bus. The extended LSI-11 bus contains 4 additional address lines (BDAL<21:18>) in addition to the 38 lines of the original LSI-11 bus. The four additional address lines extend the 256K-byte physical address space of the LSI-11 bus to 4 megabytes. Addresses, 8-bit bytes or 16-bit data words, bus synchronization, and control signals are sent along these 42 lines. Addresses may be either 16, 18, or 22 bits wide, depending on the addressing capability of the processor installed in the system. The 16-bit data and the first 16 address bits are time-multiplexed over the same 16 data/address lines. Two additional address bits (<17:16>) and the memory parity bits are also time-multiplexed over two signal lines. The signal lines are functionally divided as listed in Table 4-1. Refer to Appendix F for a detailed list of the extended LSI-11 bus signal functions.

The LSI-11 bus lines may be considered transmission lines that are terminated in their characteristic impedance (Z_0) at both the near and far ends of the bus. The near end of the bus is defined as the first bus interface slot in the backplane, the far end is the last bus interface slot.

Table 4-1 Summary of Signal Line Functions

Quantity	Function	Bus Signal Mnemonic
16	Data/address lines	BDAL<15:00>
2	Memory parity/address lines	BDAL<17;16>
4	Address lines	BDAL<21:18>
6	Address and data transfer control lines	BSYNC, BDIN, BDOUT, BWTBT, BBS7, BRPLY
3	Direct memory access (DMA) control lines	BDMR, BDMG, BSACK
5	Interrupt control lines	BIRQ4, BIRQ5, BIRQ6, BIRQ7, BIAK
6	System control lines	BPOK, BDCOK, BINIT, BHALT, BREF, BEVNT

Most LSI-11 bus signals are bidirectional and use a terminating resistor network connected between +5 V and ground to provide a negated (high) signal level. Devices may be connected to any point along the bus to receive signals from the near or far end of the bus via high-impedance bus receivers, or to transmit signals to the near or far end through gated open-collector bus drivers. A bus driver asserts a signal by causing the line to go from a high level (approximately 3.4 V) to a low level (approximately 0.5 V). Although bidirectional lines are electrically bidirectional, certain lines carry signals that are functionally unidirectional. The functionally unidirectional lines carry signals that are required to travel in only one direction. For example, when a device asserts a bus request signal (BIRQ), the signal always travels from the requesting device to the processor and never in the reverse direction.

The interrupt acknowledge (BIAK) and direct memory access grant (BDMG) signals are physically unidirectional signals that are wired to each LSI-11 bus slot in a daisy-chain scheme. These signals are generated by the processor in response to interrupt and direct memory access requests and are transmitted to the bus via output signal pins. Each of the output signals (BIAKO or BDMGO) is received on a device input pin (BIAKI or BDMGI) and conditionally retransmitted via a device output pin (BIAKO or BDMGO). These signals are received from higher-priority devices and retransmitted to lower-priority devices on the bus. DMA and I/O interrupt priorities are discussed in Pargaraphs 4.4 and 4.5.1.

Bus 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-BA 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. The extended LSI-11 bus control signals transmitted or received by the bus master or bus slave device must complete the sequence according to the protocol established for transferring address and data information. The processor controls bus arbitration (i.e., "decides" which device is to be bus master at any given time).

A typical example of a master/slave relationship has the processor, as master, fetching an instruction from memory which is always a slave). Another example is a disk drive, as master, transferring data to memory, again, as the slave. Any device except the processor can be master or slave depending on the circumstances. Communication on the extended LSI-11 bus is interlocked; therefore, 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 extended LSI-11 bus asynchronous. The asynchronous operation allows both fast and slow devices to use the bus and eliminates the need for synchronizing clock pulses between the bus master and slave device.

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 μs. The KDF11-BA has a bus timer that restarts the clock when no device responds to BDIN L or BDOUT L within 10 μs. An immediate trap to location 4_8 occurs. The slowest peripheral or memory device must respond in less than 10 μs to prevent a bus timeout error.

4.2 BUS SIGNAL NOMENCLATURE

Throughout the following protocol specifications, bus signals are referred to in several different ways.

1. In general discussions where timing, polarity, and physical location are unimportant, the base signal name without any prefixes or suffixes is used. For example:

   SYNC, WTBT, BS7, DAL<21:00> or the DAL lines

2. Most signals on the backplane etch are asserted low and referred to with a prefix character B, and a suffix (space) L. For example:

   BSYNC L, BWTBT L, BBS7 L, BDAL<21:00> L

   BPOK H and BDCOK H are asserted high.

3. Receivers and drivers are considered part of the bus. Signal inputs to drivers are referred to with a prefix character T for transmit. For example:

   TSYNC, TWTBT, TBS7, TDAL<21:00>

4. Signal outputs of receivers are referred to with a prefix character R for received. For example:

   RSYNC, RWTBT, RBS7, RDAL<21:00>

Whenever timing is important, the designations in items 3 and 4 above are used to reference timing to a receiver output or driver input. For example, after receipt of the negation of RDIN, the slave negates its TRPLY (0 ns minimum, 8000 ns maximum). It must maintain data valid on its TDAL lines until 0 ns (minimum) after the negation of RDIN, and must negate its TDAL lines 100 ns (maximum) after the negation of its TRPLY.

4.3 DATA TRANSFER BUS CYCLES

Data is transferred between a bus master and slave device to accomplish various functions. The data transfer bus cycles and their functions are described in Table 4-2.

Table 4-2 Data Transfer Bus Cycles

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 data to be written in the destination byte during byte output operations is valid on the appropriate BDAL lines. For example, BDAL<15:08> contains the high byte, and BDAL<07:00> contains the low byte. Table 4-3 describes the bus signals used in a data transfer operation.

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.

Table 4-3 Data Transfer Bus Signals

Mnemonic	Description	Function
BDAL<21:00> L	22 Data/address lines	BDAL<21:18> L are used for 22-bit extended addressing; BDAL<17:16> L are used for 18-bit extended addressing, memory parity error, and memory parity error enable functions; BDAL<15:00> L are used for 16-bit addressing, word and byte transfers.
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	

4.3.1 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 is 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 of the bus cycle, the operations performed will vary slightly, depending on the type of data transfer desired. Paragraphs 4.3.1.2 through 4.3.1.4 describe the data transfer portion of the various bus cycles.

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

1. Asserts TDAL<21:00> with the desired slave device address bits.
2. Asserts TBS7 if a device in the I/O page is being addressed.
3. Asserts TWTBT if the cycle is a DATO(B) bus cycle.
4. Asserts TSYNC 150 ns (minimum) after gating TDAL, TBS7, and TWTBT onto the bus.

During this time the address, RBS7, and RWTBT signals are asserted at the slave bus receiver for at least 75 ns before RSYNC becomes active. Devices in the I/O page ignore the 9 high-order address bits RDAL<21:13> and instead decode RBS7 along with the 13 low-order address bits. An active RWTBT signal indicates that a DATO(B) operation follows, while an inactive RWTBT indicates a DATI or DATIO(B) operation.

The address hold/deskew time begins after RSYNC is asserted. The slave device uses the active RSYNC to clock RDAL address bits, RBS7 and RWTBT, into its internal logic. RDAL<21:00>, RBS7, and RWTBT will remain active for 25 ns (minimum) after the RSYNC becomes active. RSYNC remains active for the duration of the bus cycle.

Memory and peripheral devices are addressed similarly, except for the way they respond to RBS7. Addressed peripheral devices must not decode address bits on RDAL<17:13>. Addressed peripheral devices may respond to a bus cycle only when RBS7 is asserted during the addressing portion of the cycle. When asserted, RBS7 indicates that the device address resides in the I/O page (the upper 8K-byte 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.

4.3.1.2 DATI – The DATI bus cycle is a read operation that inputs data from the slave device to the bus master. The operations performed by the bus master and slave device during a DATI are shown in Figure 4-1. The DATI bus cycle timing is shown in Figure 4-2. Data consists of 16-bit word transfers over the bus. During the data transfer portion of the DATI bus cycle, the bus master asserts TDIN 100 ns (minimum) after it asserts TSYNC. The slave device responds to RDIN active by asserting:

1. TRPLY after receiving RDIN and 125 ns (maximum) before TDAL bus driver data bits are valid.

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

When the bus master receives RRPLY, it does the following.

1. Waits at least 200 ns deskew time and then accepts input data at RDAL<15:00> bus receivers. RDAL<17:16> are monitored for a possible parity error indication.

2. Negates TDIN 150 ns (minimum) after RRPLY becomes active.

The slave device responds to RDIN negation by negating TRPLY and removing read data from TDAL bus drivers. TRPLY must be negated 100 ns (maximum) prior to removal of read data. The bus master responds to the negated RRPLY by negating TSYNC.

Conditions for the next TSYNC assertion are as follows.

1. TSYNC must remain negated for 200 ns (minimum).

2. TSYNC must not become asserted within 300 ns of the previous RRPLY negation.

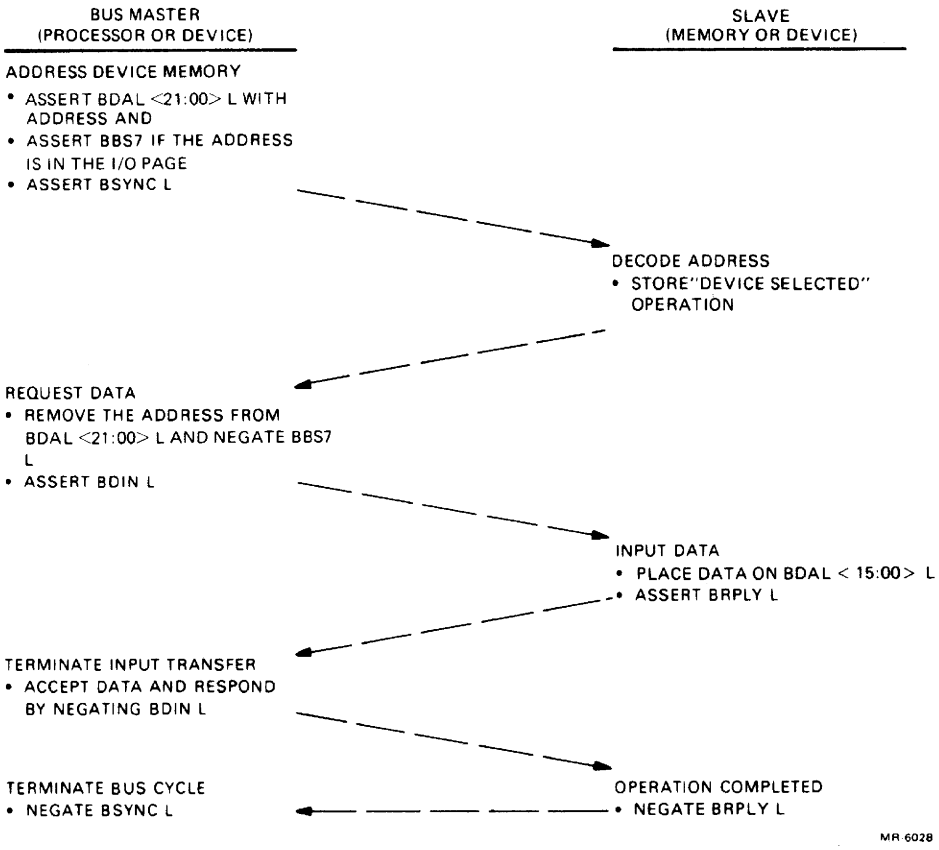
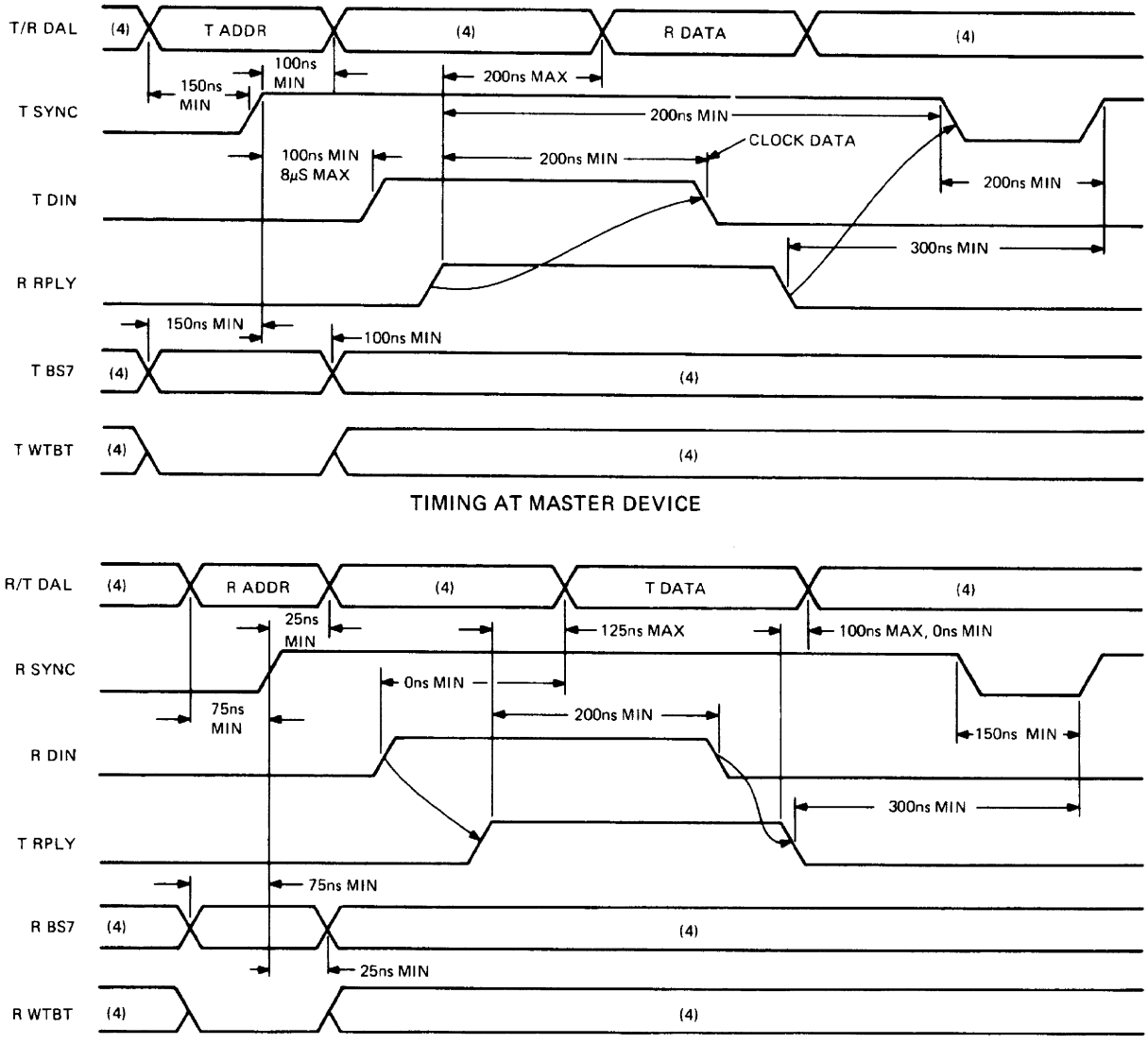


Figure 4-1 DATI Bus Cycle



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-6037

Figure 4-2 DATI Bus Cycle Timing

4.3.1.3 DATO(B) – 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 TWTBT has been asserted by the bus master, or immediately following an input transfer part of a DATIO(B) bus cycle. The operations performed by the bus master and slave device during a DATO(B) bus cycle are shown in Figure 4-3. The DATO(B) bus cycle timing is shown in Figure 4-4.

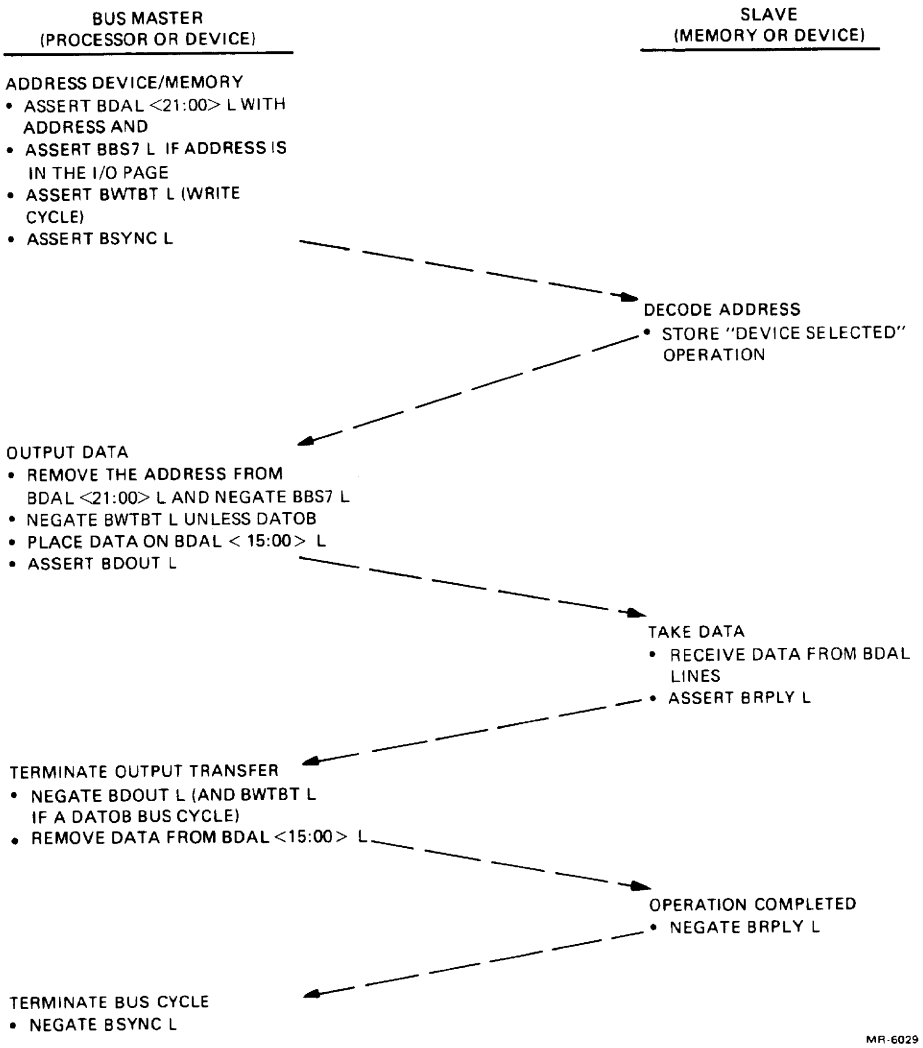
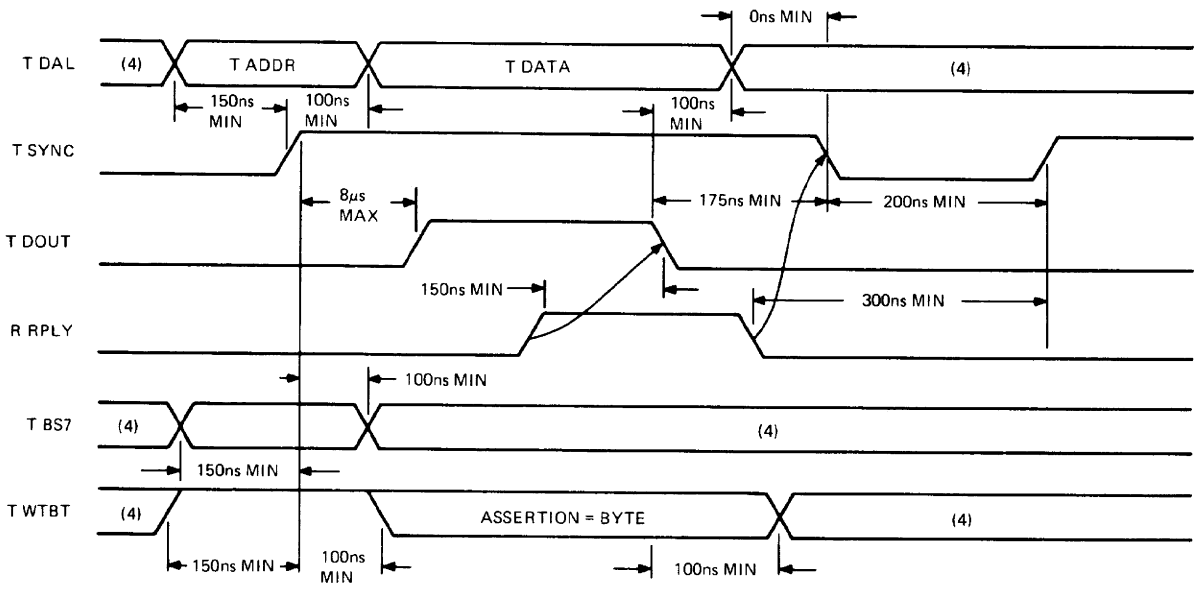
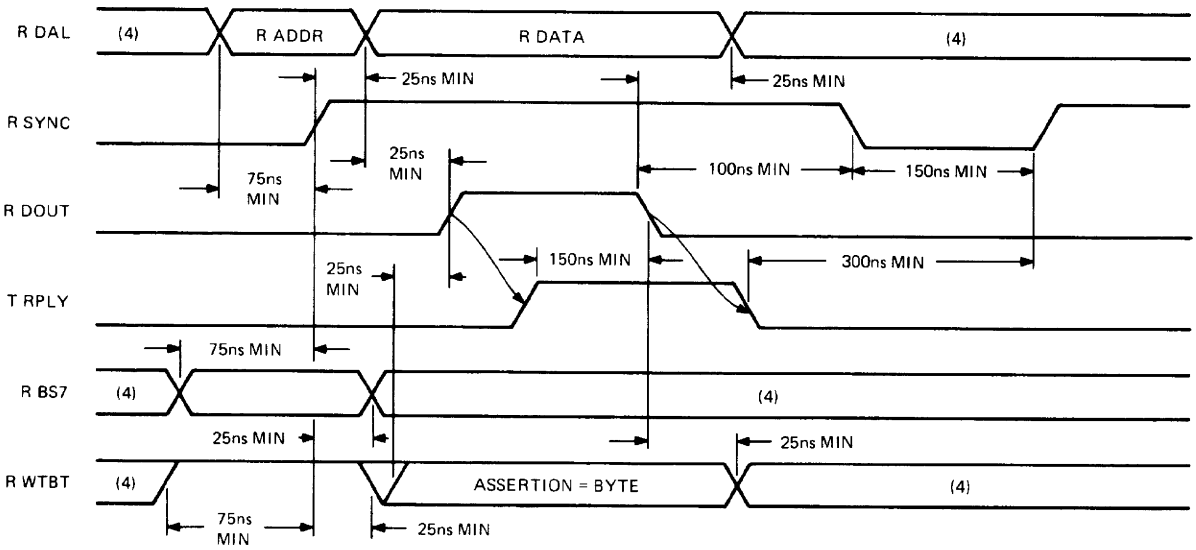


Figure 4-3 DATO or DATO(B) Bus Cycle



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-1179

Figure 4-4 DATO or DATO(B) Bus Cycle Timing

The data transfer portion of a DATO(B) bus cycle comprises a data setup/deskew time and a data hold/deskew time. During the data setup/deskew time, the bus master outputs the data on TDAL<15:00> 100 ns (minimum) after TSYNC is asserted. If it is a word transfer, the bus master negates TWTBT while gating data onto the bus. If the transfer is a byte transfer, the bus master asserts TWTBT while gating data onto the bus. During a byte transfer, the condition of BDAL 00 L during the address cycle selects the high or low byte. 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 data transfer 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 TDOUT L 100 ns (minimum) after the TDAL and TWTBT bus driver inputs are stable. The slave device responds to RDOUT by accepting the input data and asserting TRPLY (8 μs maximum to avoid bus timeout). This completes the data setup/deskew time. During the data hold/deskew time the bus master negates TDOUT 150 ns (minimum) after the assertion of RRPLY. TDAL<21:00> bus drivers remain stable for at least 100 ns after TDOUT negation. The bus master then negates TDAL inputs.

During this time, the slave device senses RDOUT negation and negates TRPLY. The bus master responds by negating TSYNC. However, the processor will not negate TSYNC for at least 175 ns after negating TDOUT. This completes the DATO(B) bus cycle. Before the next cycle, TSYNC must remain unasserted for at least 200 ns. Also, TSYNC may not assert until 300 ns (minimum) after RRPLY negates.

4.3.1.4 DATIO(B) – 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 4.3.1.2; however, TSYNC is not negated. TSYNC remains active for an output word or byte transfer [DATO(B)]. The bus master maintains at least 200 ns between RRPLY negation during the DATI cycle and TDOUT assertion. The cycle is terminated when the bus master negates TSYNC, which follows the same protocol as described for DATO(B). The operations performed by the bus master and slave device during a DATIO or DATIO(B) bus cycle are shown in Figure 4-5. The DATIO and DATIO(B) bus cycle timing is shown in Figure 4-6.

4.4 DIRECT MEMORY ACCESS (DMA)

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., disk drives) 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 requests are assigned the highest priority level.

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 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

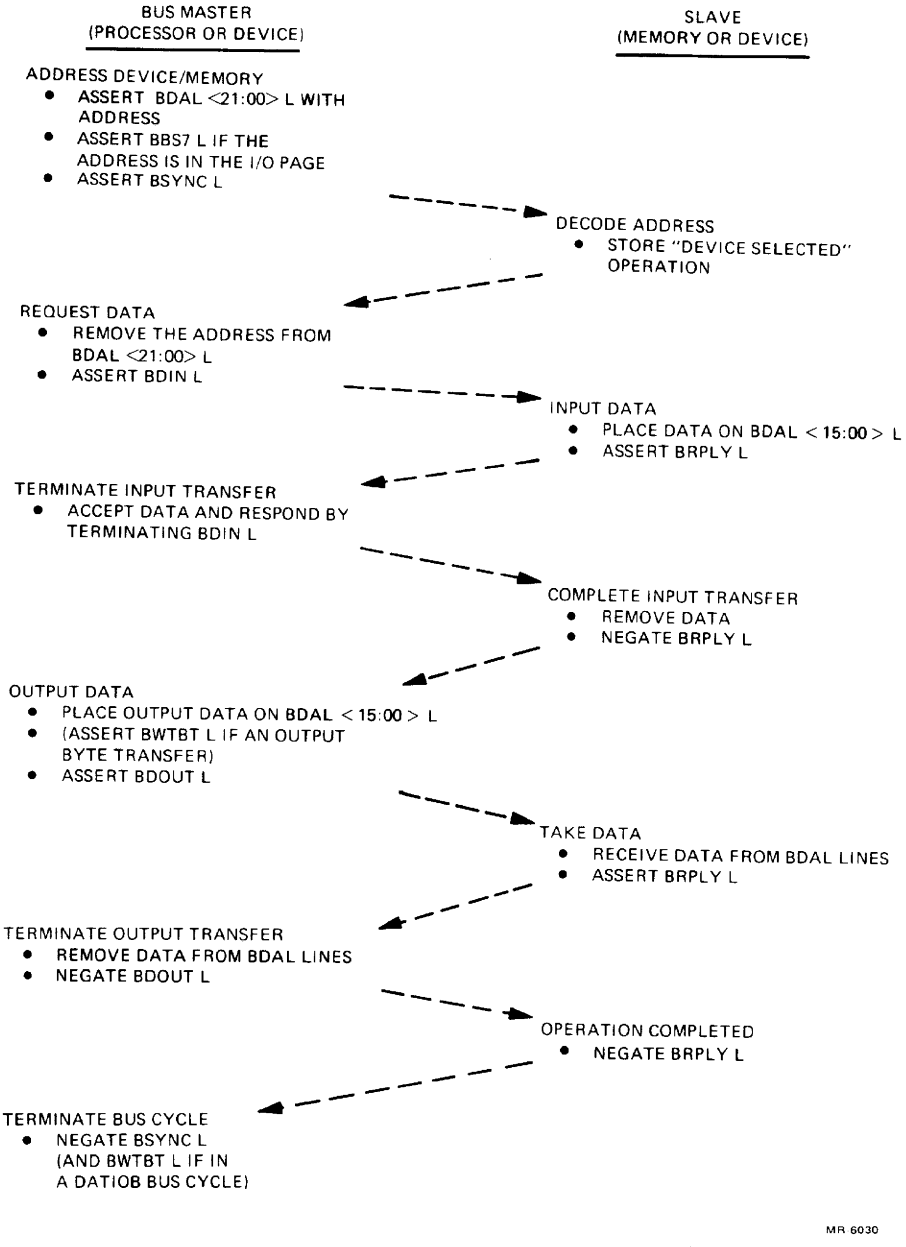
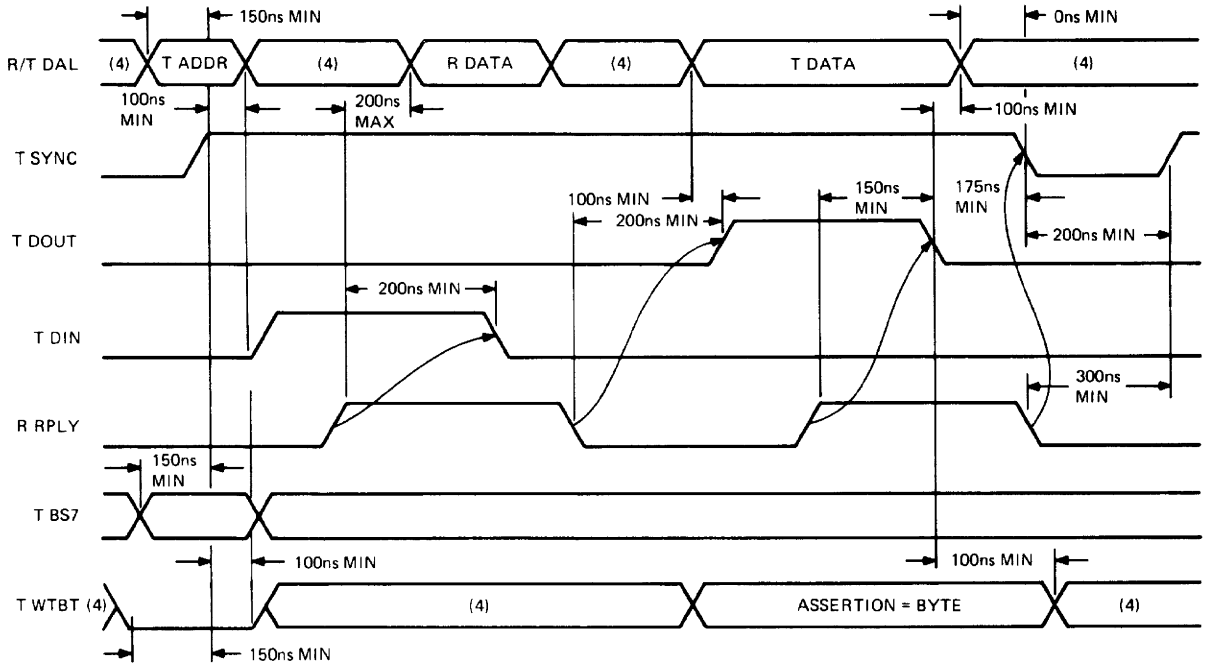
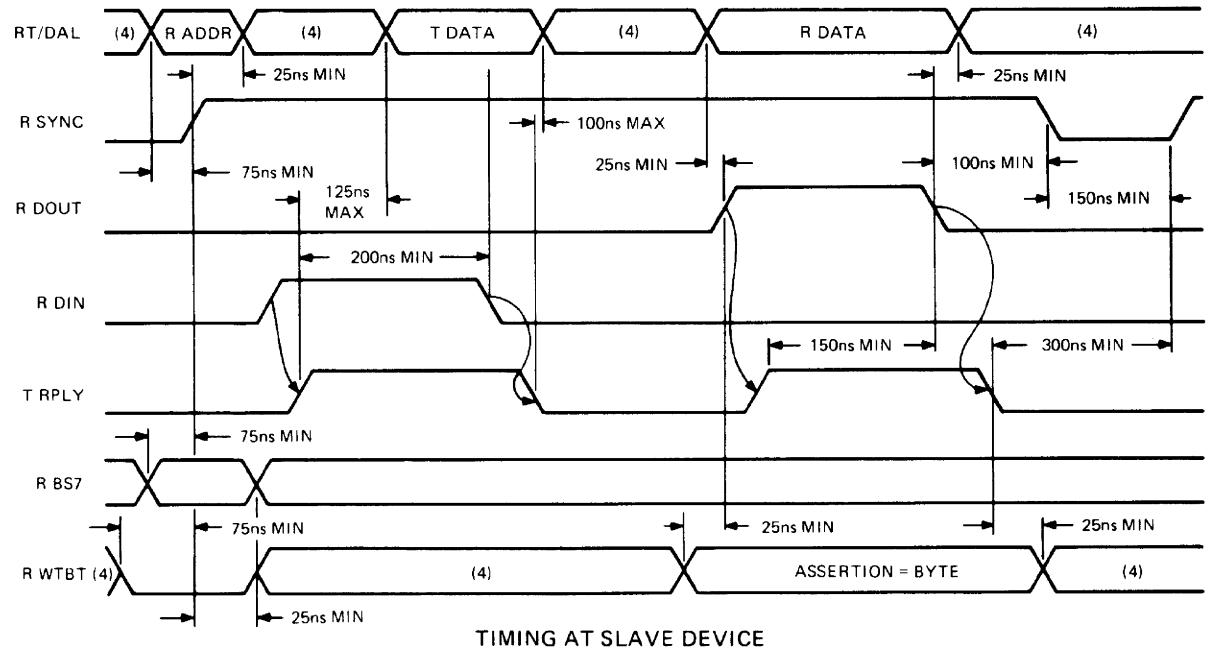


Figure 4-5 DATIO or DATIO(B) Bus Cycle



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.

MR 6036

Figure 4-6 DATIO or DATIO(B) Bus Cycle Timing

A DMA transaction is divided into three phases: the bus mastership acquisition phase, the data transfer phase, and the bus mastership relinquish phase. The operations performed by the processor and bus master during the DMA request/grant sequence are shown in Figure 4-7. The DMA request/grant bus cycle timing is shown in Figure 4-8.

During the bus mastership acquisition phase, a DMA device requests the bus by asserting TDMR. The processor arbitrates the request and initiates the transfer of bus mastership by asserting TDMGO. The maximum time between BDMR L assertion by the DMA device and BDMGO L assertion by the processor is DMA latency. This time is processor-dependent. The KDF11-BA asserts TDMGO 1.4 μs (maximum) after the assertion of RDMR.

BDMGO L/BDMGI L is one of two signals that are daisy-chained through each module in the backplane. The signal 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 BDMGO L and asserts TSACK. If no device responds to the DMA grant, the processor will clear the grant and rearbitrate the bus.

NOTE

The KDF11-BA uses a "NO-SACK" timer, which clears BDMGO L if BSACK L is not received from the DMA device within 10 μs.

During the data transfer phase, the DMA device continues asserting BSACK L. 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). The actual data transfer is performed in the same manner as the data transfer portion of DATI, DATO(B) and DATIO(B) bus cycles described in Paragraphs 4.3.1.2 through 4.3.1.4.

The DMA device can assert TSYNC L for a data transfer 0 ns (minimum) after it receives RDMGI L, 250 ns (minimum) after RSYNC is negated, and 300 ns (minimum) after RRPLY is negated.

During the bus mastership relinquish phase the DMA device relinquishes the bus by negating TSACK. This occurs after the last data transfer cycle (RRPLY negated) is completed (or aborted). TSACK may be negated up to 300 ns (maximum) before negating TSYNC.

4.5 INTERRUPTS

The interrupt capability of the LSI-11 bus allows any I/O device to suspend temporarily (interrupt) current program execution and divert processor operation for service of the requesting device. The processor inputs a vector from the device to start the service routine (handler). As with a device register address, the hardware fixes the device vector at locations within a designated range of addresses between 000 and 777_8. The vector indicates the first of a pair of addresses. The content of the first address is read by the processor; it is the starting address of the interrupt handler. The content of the second address is a new processor status word (PS). The PS bits <07:05> can be programmed to a priority level from 0 to 7_8. Only interrupts on a level higher than the number in the priority level field of the PS are serviced by the processor. If the interrupt priority level of the new PS is higher than that of the original PS, the new PS raises the interrupt priority level and thus prevents lower-level interrupts from breaking into the current interrupt service routine. Control is returned to the interrupted program when the interrupt service routine is completed. 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 instruction (RTI or RTT) at the end of the service routine. The use of the stack and the LSI-11 bus interrupt scheme can allow interrupts to occur within interrupts (nested interrupts) if the requesting interrupt has a higher priority level than the interrupt currently being serviced.

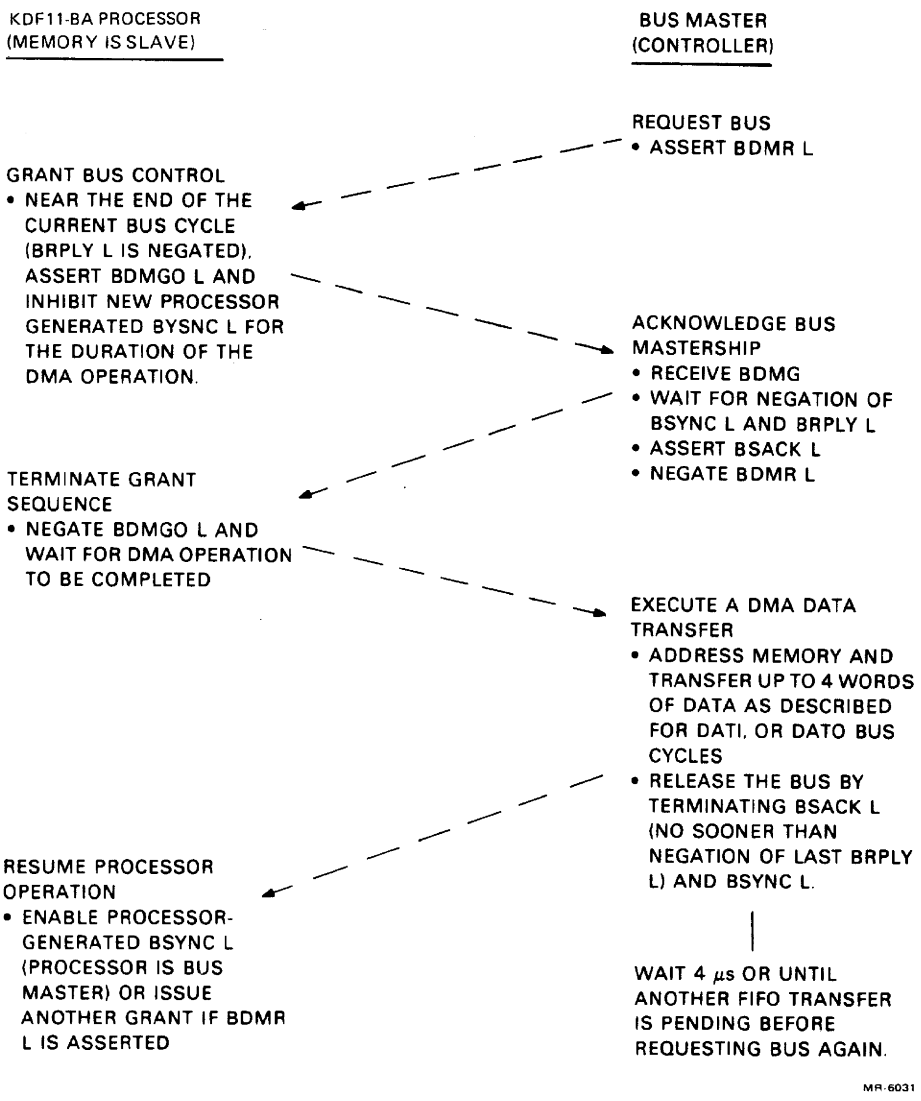


Figure 4-7 DMA Request/Grant Sequence

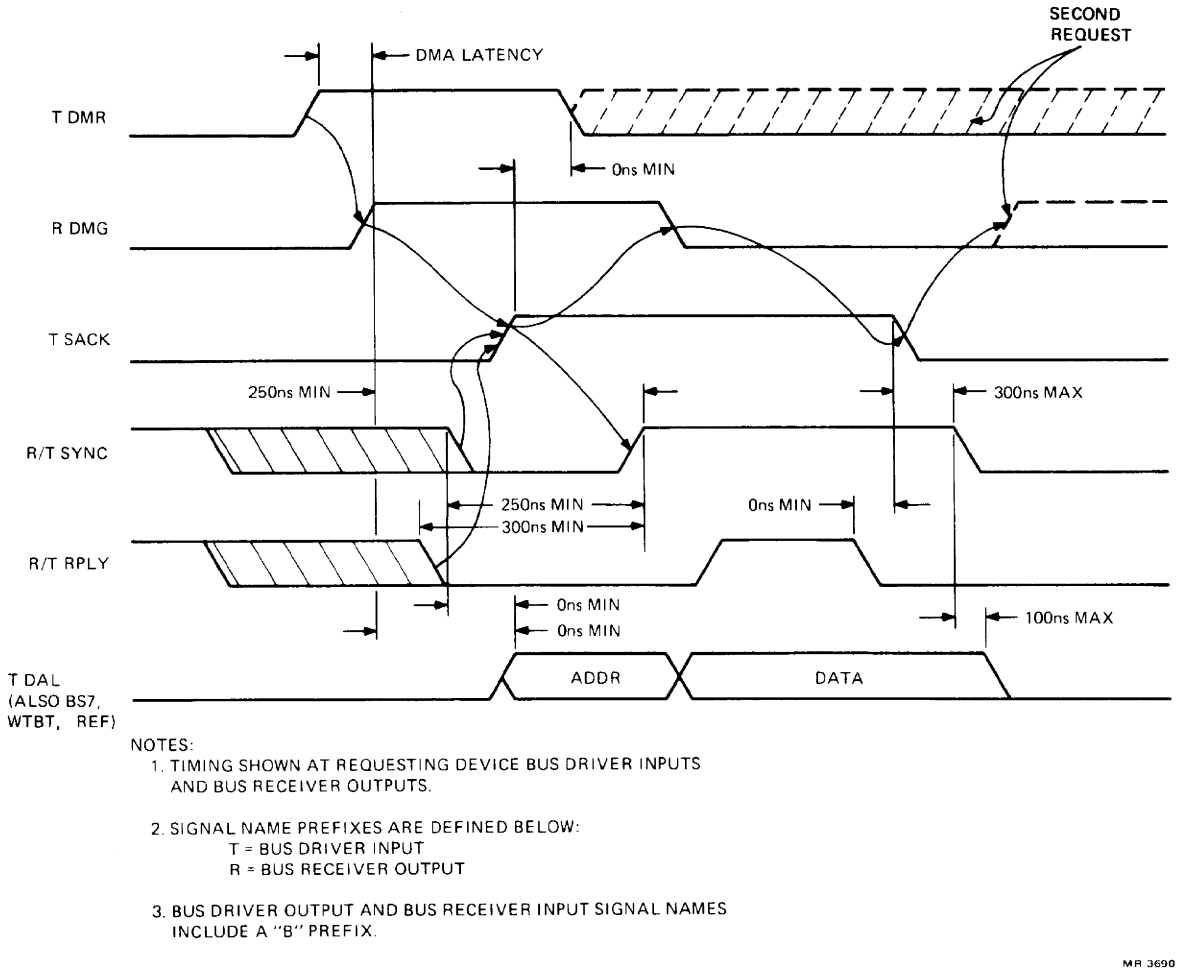


Figure 4-8 DMA Request/Grant Bus Cycle Timing

Interrupts can be caused by LSI-11 bus options and can also originate in the processor. Interrupts originating in the processor are called "traps" and are caused by programming errors, hardware errors, special instructions, and maintenance features. The following are the LSI-11 bus signals used in interrupt transactions.

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

4.5.1 Device Priority

The LSI-11 bus supports the following two methods of determining 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 device closest to the processor has the highest priority while the device at the far end of the bus has the lowest priority.

The KDF11-BA uses both methods – distributed arbitration, 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 that interrupt on BIRQ4 can also be used in KDF11-BA systems but must be placed in a bus slot following the last bus slot in which a position-independent device is installed.

4.5.2 Interrupt Protocol

Interrupt protocol has three phases: the interrupt request phase, the interrupt acknowledge and priority arbitration phase, and the interrupt vector transfer phase. The operations performed by the processor and interrupting device are shown in Figure 4-9. Interrupt protocol timing is shown in Figure 4-10.

The interrupt request phase begins when a device meets its specific conditions for interrupt requests; for example, when the device is "ready," "done," or when 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 at which a device is configured must also be asserted. (A special case exists for level-7 devices that must also assert level 6.) The interrupt request line remains asserted until the request is acknowledged.

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

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

1. The device interrupt priority is higher than the current priority level stored in 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 TDIN and, 225 ns (minimum) later, asserting TIAKO. The device electrically closest to the processor receives the acknowledge on its RIAKI bus receiver.

On the leading edge of RDIN, each bus option capable of requesting interrupts decides whether to accept or to pass on the RIAKI signal. A device that does not support position-independent, multilevel interrupts will accept RIAKI if it is requesting an interrupt when RDIN asserts. A device that does support position-independent, multilevel interrupts accepts RIAKI if it is requesting an interrupt and if there is no higher-priority request pending when RDIN asserts. This decision must be clocked into a flip-flop, which settles within 150 ns of TDIN.

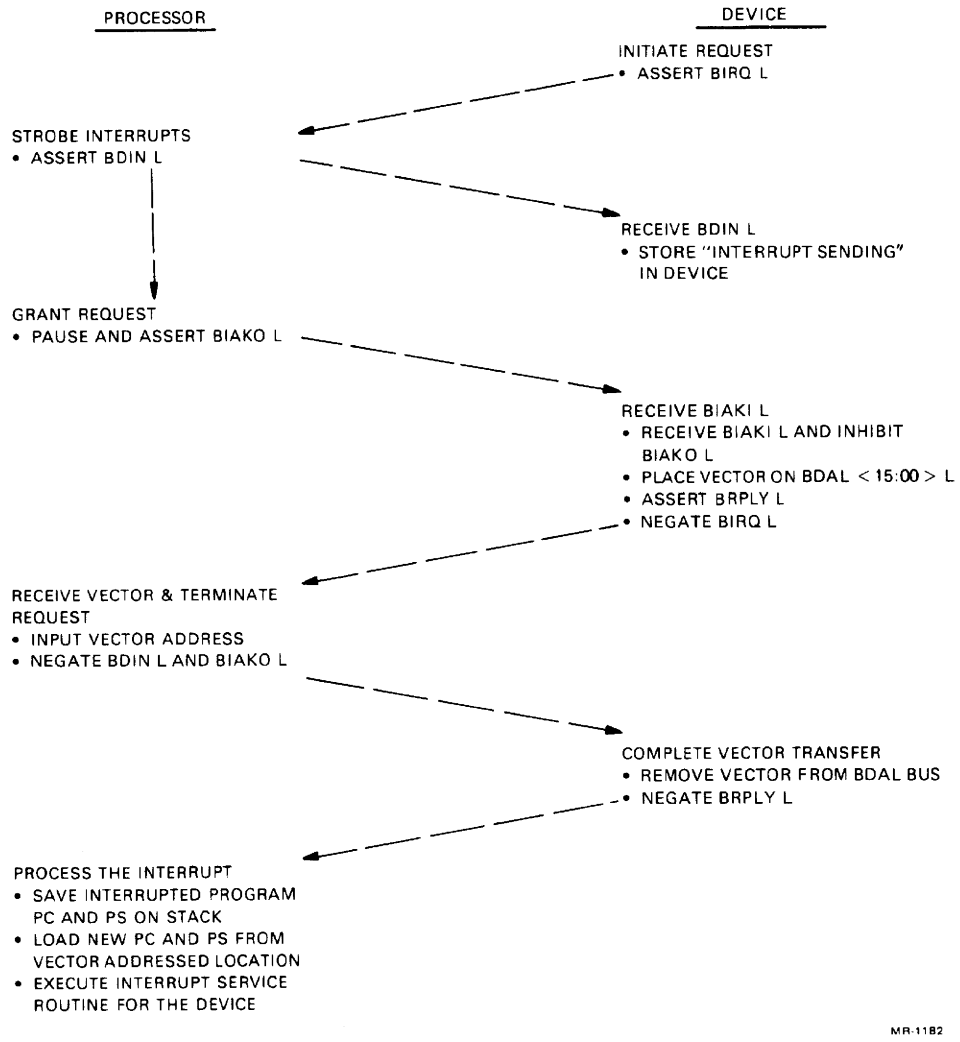
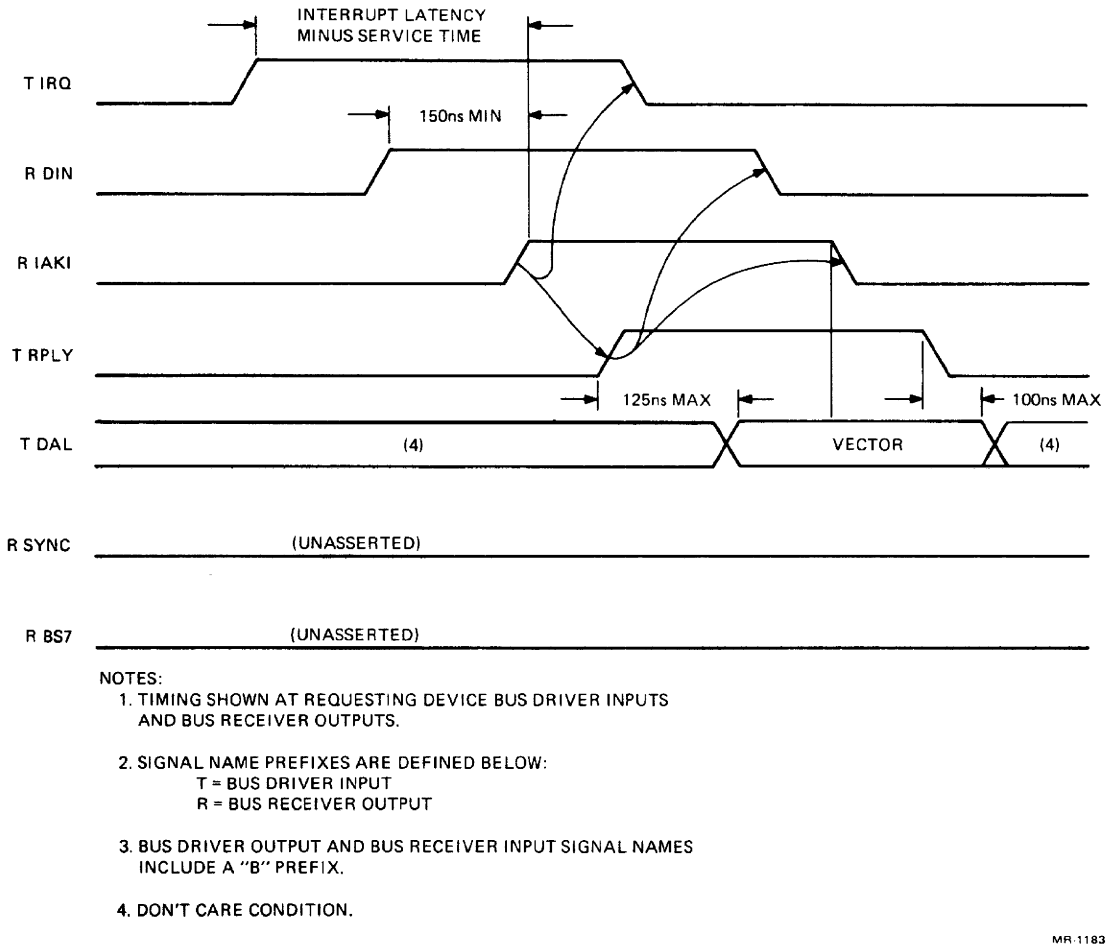


Figure 4-9 Interrupt Request/Acknowledge Sequence



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.

MR-1183

Figure 4-10 Interrupt Protocol Timing

Devices that support position-independent, multilevel interrupts assert from one to three IRQ lines when requesting an interrupt. Table 4-4 presents the IRQ lines a device at each level must assert in order to request an interrupt, and the lines it must monitor to determine whether a higher-priority device is requesting an interrupt.

Table 4-4 Position-Independent, Multilevel Device Requirements

Interrupt Level	IRQ Lines Asserted	IRQ Lines Monitored
4	TIRQ4	RIRQ5, RIRQ6
5	TIRQ4, TIRQ5	RIRQ6
6	TIRQ4, TIRQ6	RIRQ7
7	TIRQ4, TIRQ6, TIRQ7	

During the interrupt vector transfer phase, the responding interrupt device receives RIAKI and then asserts TRPLY. The vector address must be stable at TDAL<08:02> 125 ns (maximum) after TRPLY is asserted. The processor receives the assertion of RRPLY, and 200 ns (minimum) later it inputs the vector address and negates both TDIN and TIAKI. The interrupting device negates TRPLY after the negation of RIAKI and removes the vector address from TDAL<08:02> 100 ns (maximum) after TRPLY negates. Since vector addresses are constrained to be between 000 and 774_8, none of the remaining TDAL lines are used.

4.5.3 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 configuration 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 4.5.2. 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 function properly.

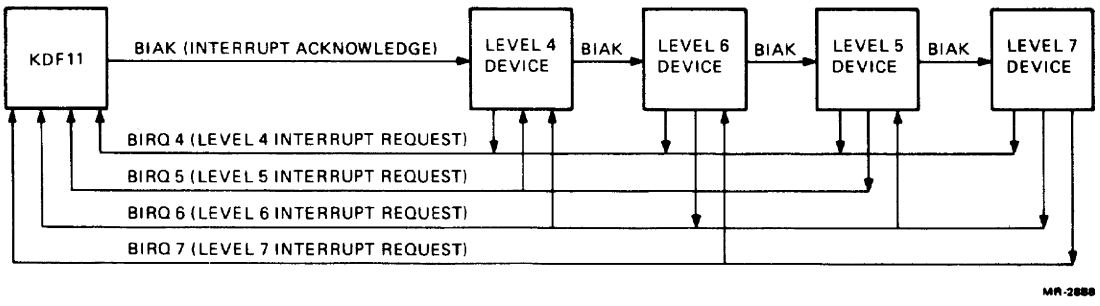


Figure 4-11 Position-Independent Configuration

The position-dependent configuration is shown in Figure 4-12. This configuration is simpler to implement, with the following constraint, however. Peripheral devices must be ordered so that the highest-priority device is located closest to the processor with the remaining devices placed in the backplane in decreasing order of priority. With this configuration each device must only 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.

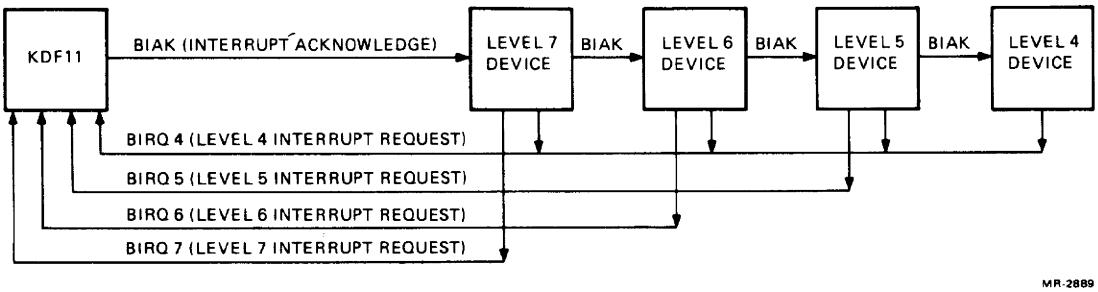


Figure 4-12 Position-Dependent Configuration

4.6 CONTROL FUNCTIONS

The following LSI-11 bus signals provide system control functions.

BREF L	Memory refresh
BHALT L	Processor halt
BINIT L	Initialize
BPOK H	Power OK
BDCOK H	DC power OK
BEVENT L	External event interrupt request

4.6.1 Memory Refresh

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

4.6.2 Halt

Assertion of BHALT L stops program execution and forces the processor unconditionally into console ODT mode. The processor does not assert the BHALT L bus line when it comes to a programmed HALT.

4.6.3 Initialization

Devices along the bus are initialized when BINIT L is asserted. The processor asserts the BINIT L signal under the following conditions.

1. During a power-down sequence.
2. During a power-up sequence.
3. During the execution of a RESET instruction.

4. After detection of a G character in ODT mode (if the processor features an ODT mode and a G command within it), and before execution of the code starting at the address that preceded the G command.

4.6.4 Power Status

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

4.6.4.1 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.

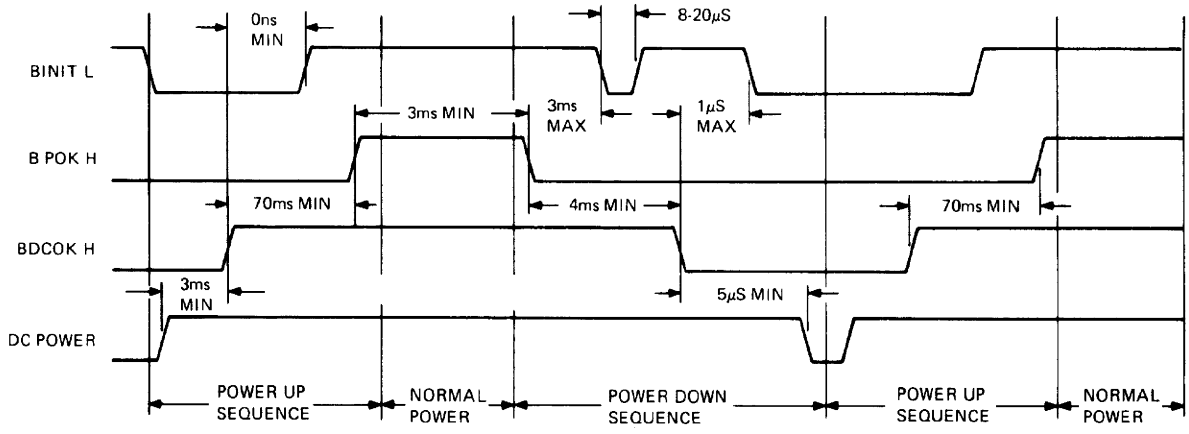
4.6.4.2 BPOK H – The assertion of this line indicates that there is at least an 8-ms reserve of dc power and that BDCOK H has been asserted for at least 70 ms. Once BPOK H has been asserted, it must remain asserted for at least 3 ms.

The negation of this line indicates that power is failing and that only 4 ms of dc power reserve remains. The negation of this line during processor operation initiates a power-fail trap sequence.

4.6.4.3 Power-Up – The following events occur during a power-up sequence.

1. Logic associated with the power supply negates BDCOK H during power-up and asserts BDCOK H 3 ms (minimum) after dc power is restored to voltages within specification.
2. The processor asserts BINIT L after receiving nominal power and negates BINIT L 0 ns (minimum) after the assertion of BDCOK H.
3. Logic associated with the power supply negates BPOK H during power-up and asserts BPOK H 70 ms (minimum) after the assertion of BDCOK H. If power does not remain stable for 70 ms, BDCOK H will be negated; therefore, devices should suspend critical actions until BPOK H is asserted.
4. BPOK H must remain asserted for a minumum of 3 ms. BDCOK H must remain asserted 4 ms (minimum) after the negation of BPOK H.

The timing diagram for the power-up/power-down sequence is shown in Figure 4-13.



NOTE:
ONCE A POWER DOWN SEQUENCE IS STARTED, IT MUST BE COMPLETED BEFORE A POWER UP SEQUENCE IS STARTED.

MR-6032

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

4.6.4.4 Power-Down – The following events occur during a power-down sequence.

1. If the ac voltage to a power supply drops below 75% of the nominal voltage for one full line cycle (15–24 ms), BPOK H is negated by the power supply. Once BPOK H is negated, the entire power-down sequence must be completed.

   A device that requested bus mastership before the power failure that has not become bus master should maintain the request until BINIT L is asserted or the request is acknowledged (in which case regular bus protocol is followed).

2. Processor software should execute a RESET instruction 3 ms (minimum) after the negation of BPOK H. This asserts BINIT L for from 8 to 20 μs. Processor software executes a HALT instruction immediately following the RESET instruction.

3. BDCOK H must be negated a minimum of 4 ms after the negation of BPOK H. This 4 ms allows mass storage and similar devices to protect themselves against erasures and erroneous writes during a power failure.

4. The processor asserts BINIT L 1 μs (minimum) after the negation of BDCOK H.

5. DC power must remain stable for a minimum of 5 μs after the negation of BDCOK H.

6. BDCOK H must remain negated for a minimum of 3 ms.

4.6.5 BEVENT L

The BEVENT L signal is an external line clock interrupt request to the processor. When BEVENT L is asserted, the processor internally assigns location 100_8 as the vector address for the BEVENT service routine. Because the vector is internally assigned, the processor does not execute the protocol for reading-in the interrupt vector address as is the case for other external interrupt requests.

4.7 BUS ELECTRICAL CHARACTERISTICS

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

4.7.1 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)

4.7.2 AC Bus Load Definition

AC bus loading is the amount of capacitance a module presents to a bus signal line. This capacitance is measured between each module signal line and ground. AC bus loading is expressed in ac unit loads where each unit load is defined as 9.35 pF.

4.7.3 DC Bus Load Definition

DC bus loading is the amount of leakage current a module presents to a bus signal line. A dc unit load is defined as 105 μA flowing into a module device when the signal line is in the unasserted (high) state.

4.7.4 120 Ω 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 Ω.

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

NOTE

1. **The KDF11-BA processor (as well as all standard DIGITAL-supplied LSI-11 interfaces) connects to the bus via special drivers and receivers, described in Paragraphs 4.7.5 and 4.7.6.**

2. **The KDF11-BA processor provides resistive (120 Ω) pull-up (on all bused lines) to 3.4 Vdc for this wired-OR interconnecting scheme.**

4.7.5 Bus Drivers

Devices driving the 120 Ω LSI-11 bus must have open collector outputs and meet the specifications that follow.

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

V_{CC} can vary from 4.75 V to 5.25 V.

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

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

AC Specifications

Bus driver output pin capacitance load: Not to exceed 10 pF.

Propagation delay: Not to exceed 35 ns.

Driver skew (difference in propagation time between slowest and fastest bus driver): 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.

4.7.6 Bus Receivers

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

DC Specifications (These conditions must be met at worst-case supply voltage, temperature, and output signal conditions.)

V_{CC} can vary from 4.75 V to 5.25 V.

Input low voltage: 1.3 V (maximum).

Input high voltage: 1.7 V (minimum).

Maximum input leakage current when connected to 3.8 Vdc: 80 μA with V_{CC} between 0.0 and 5.25 V.

AC Specifications

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

Propagation delay: Not to exceed 35 ns.

Receiver skew (difference in propagation time between slowest and fastest receiver): Not to exceed 25 ns.

4.7.7 Bus Termination

The 120 Ω LSI-11 bus must be terminated at each end by an appropriate resistive termination. A pair of resistors in series from +5.0 V to ground is used to establish a voltage for each bidirectional line when that line is not being driven (negated). The parallel impedance of this pair of resistors is 120 Ω. The terminating resistors are shown in Figure 4-14. The KDF11-BA contains terminating resistor networks in 16-pin dual-in-line packages to provide the 120 Ω terminations for the data/address, synchronization, and control lines at the processor end of the bus.

Some system configurations do not require terminating resistors at the far end of the bus. If the system configuration does require such termination, it is typically provided by a M9404-YA cable connector module. Rules for configuring single- and multiple-backplane systems are described in Paragraphs 4.8.1 and 4.8.2.

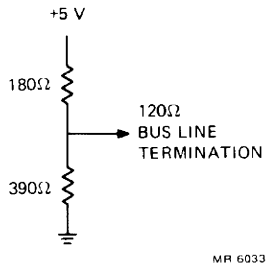


Figure 4-14 Bus Line Termination

4.7.8 Bus Interconnection Wiring

This paragraph contains the electrical characteristics of the bus interface. The bus interface for the module connectors is provided by one, two or three backplanes, depending on the system configuration. Since each backplane contains 9 slots, a system may have a maximum of 27 module interfaces to the bus.

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

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

2. Crosstalk from a pulse-driven line to an undriven line to which a constant 5 V is applied must be less than 5% of the 5 V. 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 a bus segment signal path, as measured between the near-end terminator and far-end terminator modules (including all intervening connectors, cables, backplane wiring, connector-module etch, etc.) must not exceed 2 Ω.

4. DC resistance of a bus segment common return path, as measured between the near-end terminator and far-end terminator modules (including all intervening connectors, cables, backplane wiring, connector-module etch, etc.) must not exceed an equivalent of 2 Ω per signal path. Thus, the composite signal return path dc resistance must not exceed 2 Ω divided by 40 bus lines, or 50 mΩ. 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.

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

4.7.8.3 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

The maximum allowable current per pin is 1.5 A. The +5 Vdc must be regulated to ±5% and the maximum ripple should not exceed 100 mV peak-to-peak. The +12 Vdc must be regulated to ±3% and the maximum ripple should not exceed 200 mV peak-to-peak.

NOTE

Power is not bused between backplanes on any interconnecting LSI-11 bus cables.

4.7.8.4 Maintenance and Spare Pins

Maintenance Pins – There are four M SPARE 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 some modules to operate. This allows a module to use these pins during initial testing as two separate points. This feature is used by DIGITAL for manufacturing tests only.

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

> **S SPARES** – These four pins, AE1, AH1, BH1, AF1 (with the exception of AF1 in slot 1), are reserved for the particular use of a module or set of modules. They may be used as test points or for intermodule connection. Appropriate wires must be added for intermodule communication since these pins are not connected in any way. The processor uses AF1 in slot 1 as an output pin for the SRUN signal. S SPARE lines cannot be used as bus connections.
>
> **P SPARES** – These two pins, AU1 and BU1 are similar to the S SPARE pins except that they are located in a manner that causes dc voltages to appear on them if a module is inserted backwards. Use of these pins is not recommended.

4.8 SYSTEM CONFIGURATIONS

LSI-11 bus systems can be divided into two types. The first type comprises those systems that use only one backplane, the second type comprising those systems that use multiple backplanes. Two sets of rules must be followed when configuring a system to accommodate the different electrical characteristics of the two types of systems. These rules are listed in Paragraphs 4.8.1 and 4.8.2.

Three characteristics of each component in an LSI-11 bus system must be known before configuring any system:

1. Power consumption – The total amount of current drawn from the +5 Vdc and +12 Vdc power supplies by all modules in the system.

2. AC bus loading – The amount of capacitance a module presents to a bus signal line. AC loading is expressed in ac unit loads, where one ac unit 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 unit loads, where one dc unit load equals 105 μA (nominal).

Power consumption, ac loading, and dc loading specifications for each module are included in the *Microcomputer Interfaces 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 bus loading of a backplane.**

4.8.1 Rules for Configuring Single-Backplane Systems

The following rules apply only to single-backplane systems. Any extension of the bus off the backplane is considered a multiple-backplane system and must be configured accordingly. A single-backplane configuration diagram is shown in 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 Ω.

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.

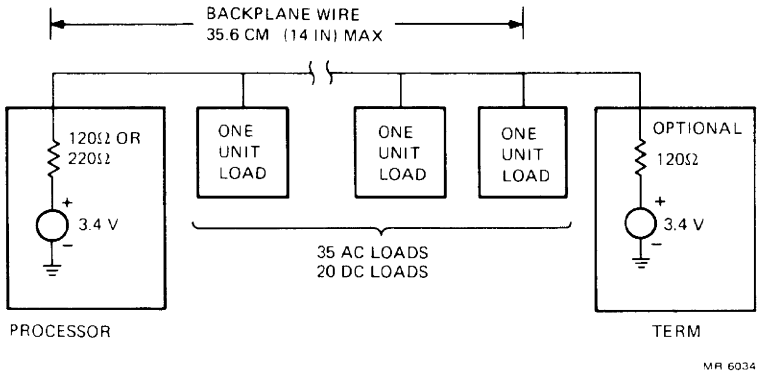


Figure 4-15 Single-Backplane Configuration

4.8.2 Rules for Configuring Multiple-Backplane Systems

Multiple-backplane systems may contain a maximum of three backplanes. A configuration diagram for a multiple-backplane system is shown in Figure 4-16.

1. 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 15 loads (total).

4. The first backplane must have an impedance of 120 Ω (obtained via the processor module). The second backplane is terminated by 120 Ω resistor networks contained on the cable connector inserted in the third backplane.

5. The cables connecting the backplanes must observe the following rules.

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

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

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

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

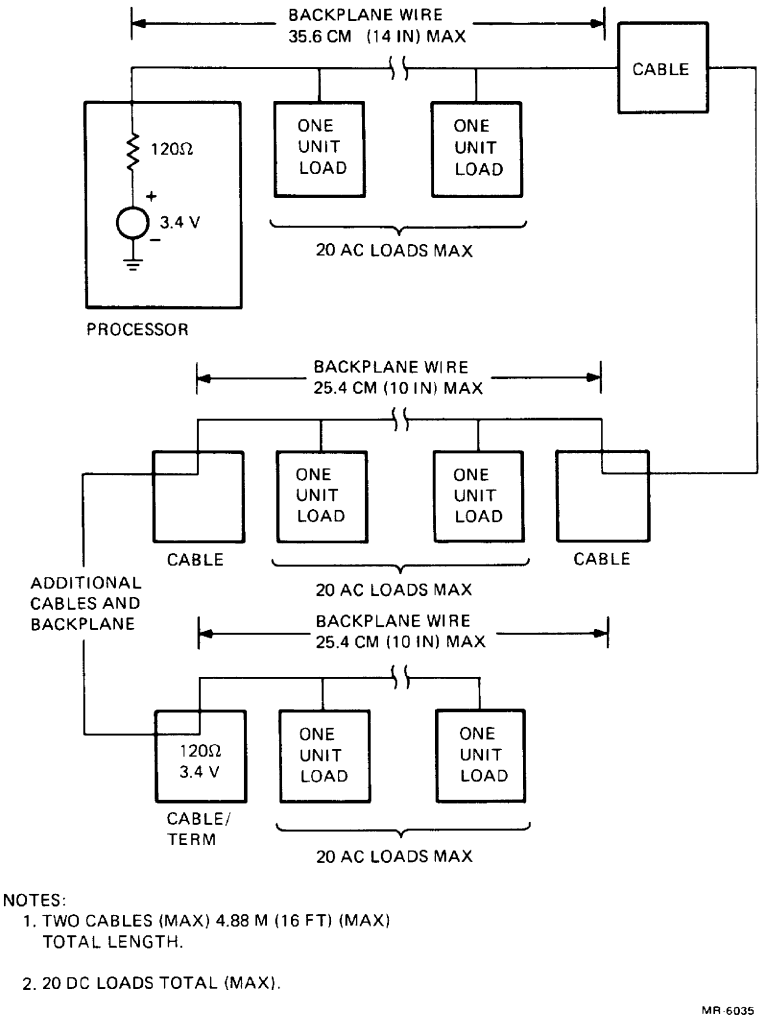


Figure 4-16 Multiple-Backplane Configuration

4.8.3 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 *Microcomputer Interfaces Handbook*.

Do not attempt to distribute power via the LSI-11 bus cables in multiple-backplane systems. 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 the BPOK H and BDCOK H signals is strongly recommended.

CHAPTER 5
FUNCTIONAL DESCRIPTION

5.1 INTRODUCTION

This chapter describes the control logic and data flow of the KDF11-BA. Figure 5-1 (Sheets 1 and 2) is a functional block diagram that shows the major logical subunits of the KDF11-BA and their relation to the LSI-11 bus and the three internal processor buses. The three internal buses are the chip/data address lines (CDALs), the microinstruction bus (MIB), and the internal data/address lines (IDALs).

The functions of the logic subunits are described at the block diagram level. The block diagrams may be used in conjunction with the KDF11-B Field Maintenance Print Set to locate the detailed circuit logic for the logic subunits. Each block in a diagram contains the letter K followed by a number 1 through 10. This alphanumeric designator refers to the specific drawing number of the KDF11-B Field Maintenance Print Set that contains the detailed circuit logic for that block.

The KDF11-BA central processor unit is contained on two LSI chips (control and data) which reside on a single 40-pin carrier. The optional memory management unit (MMU) is contained on one LSI chip, which also resides on a 40-pin carrier. The KDF11-BA has sockets for these two carriers and three extra sockets for the commercial instruction set (CIS) or floating-point (FP) options. The control and data chips, the MMU chip, and the optional CIS and FPP (floating-point processor) chips are referred to in this discussion as the F11 chip set. The F11 chips communicate among themselves and with external KDF11-BA logic over the MIB <15:00> and CDAL <21:00> bus. KDF11-BA logic interfaces the F11 chip set with the internal IDAL <15:00> bus and the external LSI-11 bus. The KDF11-BA bootstrap and diagnostic line clock and serial-line units reside on the IDAL bus. Memory and additional peripherals interface with the LSI-11 bus. Bidirectional interfaces (CDAL/IDAL transceivers and CDAL/BDAL transceivers) on the KDF11-BA module connect the CDAL <21:00> bus with the IDAL <15:00> bus and with the LSI-11 data/address lines BDAL <21:00>.

KDF11-BA logic supporting the F11 chip set includes the master clock control logic, MIB decode logic, fixed data logic, service logic, reset logic and ODT logic. Logic pertaining to the LSI-11 bus includes the bus control logic, bus synchronizer and the CDAL/BDAL transceivers. Logic pertaining to the IDAL bus peripherals includes the bus control logic, CDAL/IDAL transceivers, the IDAL address decode, bootstrap/diagnostic and line clock logic, the console and second SLU logic, the baud rate generator, and the −12 V charge pump logic.

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.

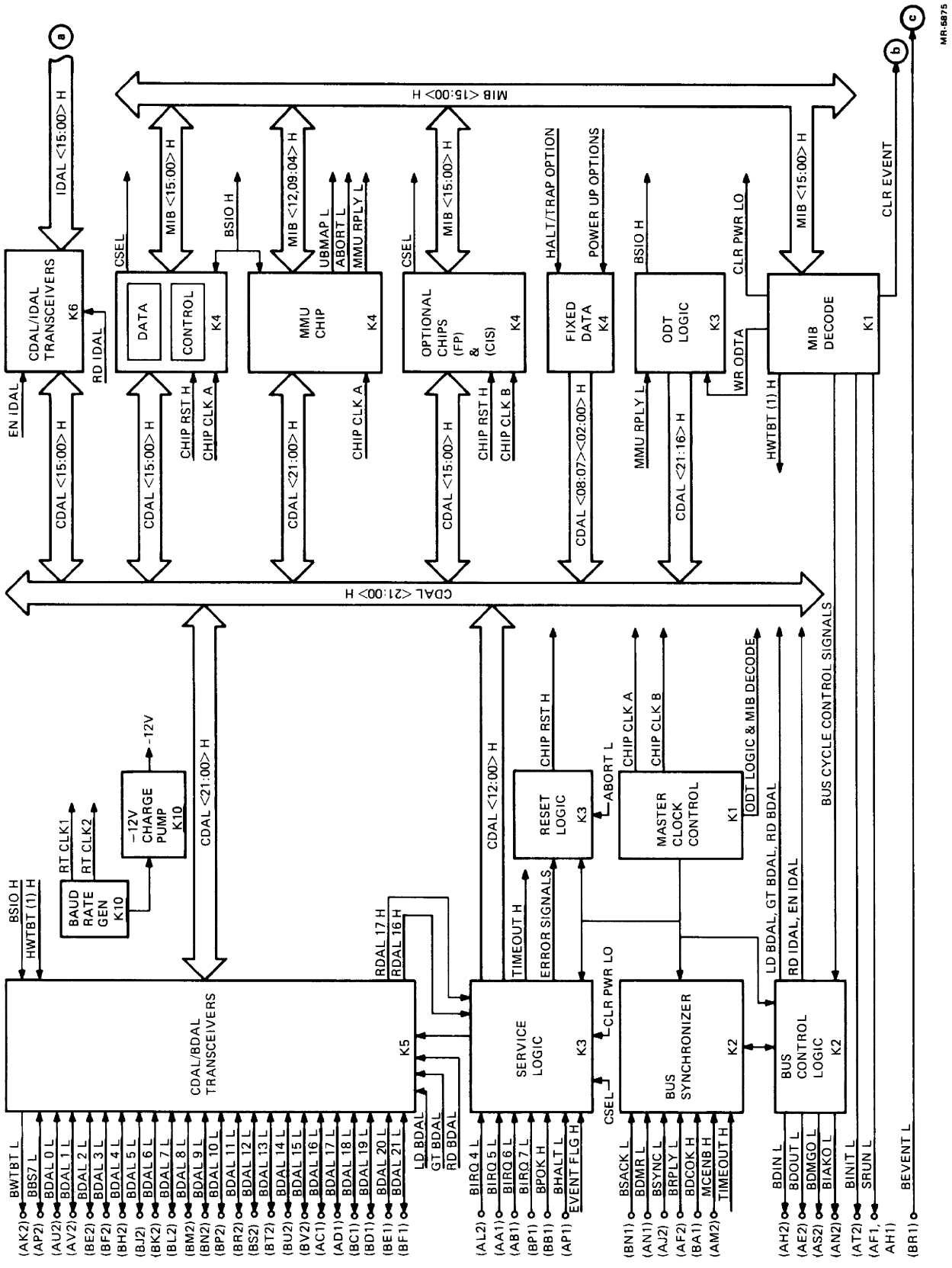


Figure 5-1 KDF11-BA Processor (Sheet 1 of 2)

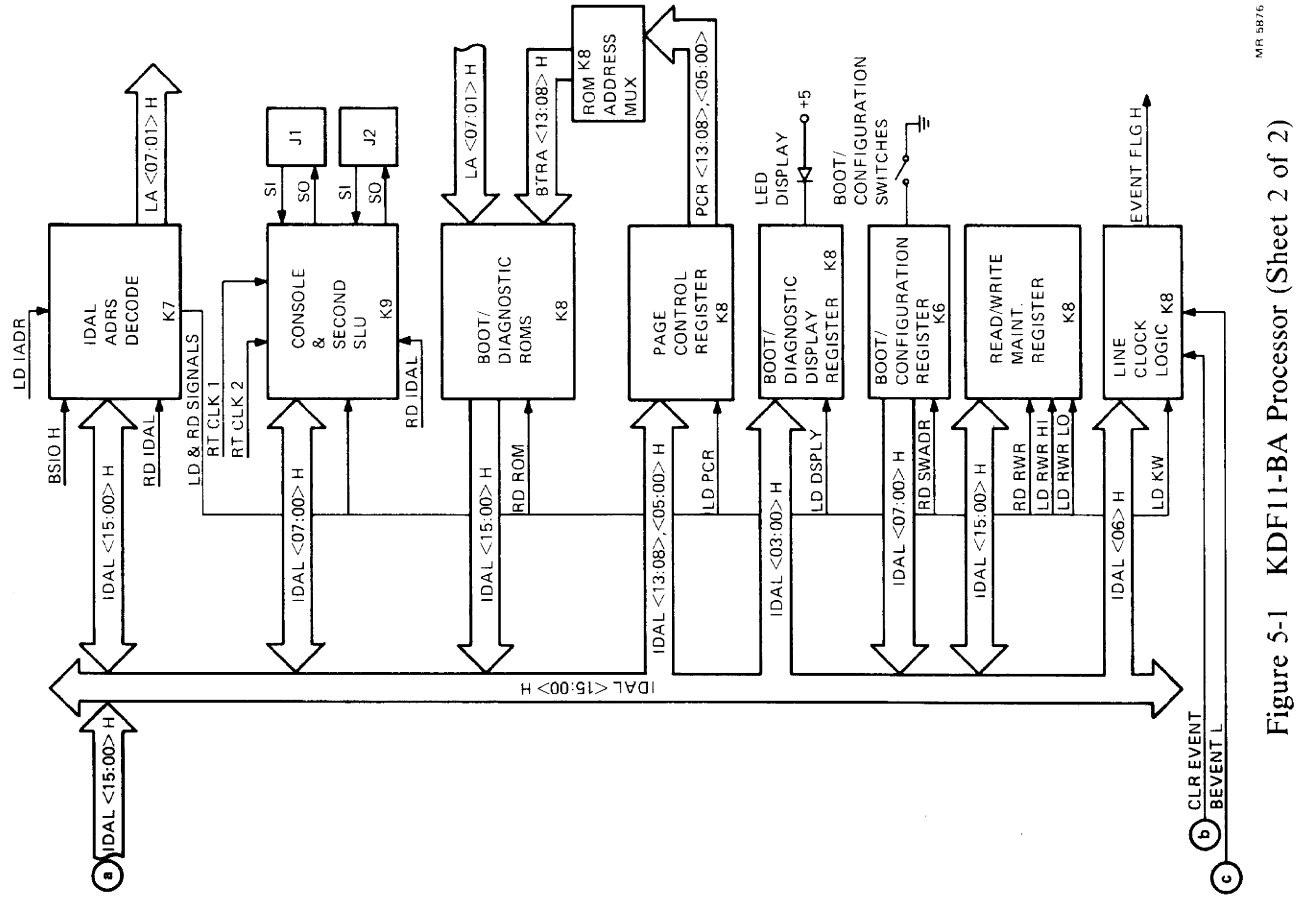


Figure 5-1 KDF11-BA Processor (Sheet 2 of 2)

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 (phase time) 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 (phase-bar time) 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.

5.3 CONTROL CHIP

The control chip contains the microprogram sequence logic and 552 words of microprogram storage in programmable logic arrays (PLAs) 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 microprogram storage.

Chip Select (CSEL) – CSEL is an open collector line with a pull-up resistor. CSEL is routed to all F11 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 storage for the FP11 floating-point accumulators and status registers. This chip provides dual mode (user and kernel) address relocation of 22 bits. Sixteen-bit virtual addresses are received from the data chip via the CDAL bus, relocated to the appropriate 18- or 22-bit physical address, and then sent on the CDAL bus to replace the original virtual address for transmission to the external LSI-11 bus. The MMU chip contains the status register and active page registers (PAR/PDR register pairs), as well as access protection and error detection capability. The MMU chip also provides the 36 16-bit registers needed for operand storage, scratchpad areas, and status information storage during floating-point operation.

The MMU chip is controlled by information received on the microinstruction bus (MIB) from the data chip and control chip and by several discrete control inputs. Complete details of memory management capabilities are described in Chapter 8.

5.5 BASE TIMING LOGIC

The base timing for the KDF11-BA is performed by the master clock control logic. Figure 5-2 shows the major logic blocks of the master clock control, the outputs of the master clock control logic, and the interface to the KDF11-BA logic.

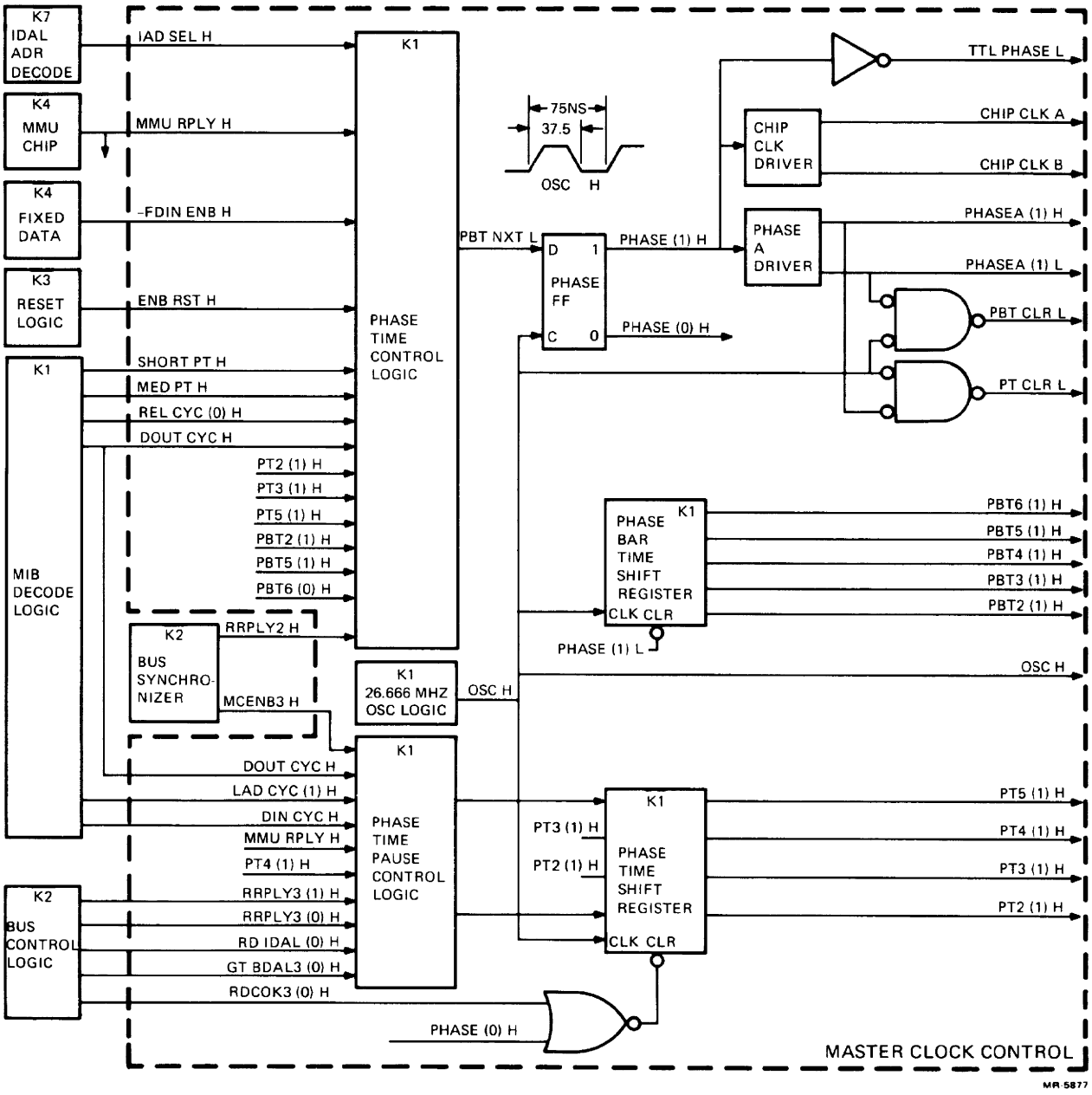


Figure 5-2 Base Timing Interface

A 26.666 MHz crystal oscillator toggles a flip-flop whose buffered output (OSC H) drives the timing logic for the F11 chip set, the control logic for the LSI-11 bus, and the IDAL bus. OSC H has a period of 75 ns and a half period of 37.5 ns.

The chip clock driver uses the PHASE (1) H flip-flop output to produce the +12 V F11 chip set clock signals (CHIP CLK A and CHIP CLK B). Each chip set clock cycle consists of a phase time [PHASE (1) H set] and a phase-bar time [PHASE (1) H clear]. The F11 chip set is semistatic and loses information if it remains in phase-bar time (PBT) for longer than 500 ns. The F11 chip set can remain in phase time (PT) indefinitely.

Two shift registers [PT2 (1) H through PT5 (1) H and PBT2 (1) H through PBT6 (1) H] operate as state machines during phase time and phase-bar time, respectively. If PHASE (1) H is set, but PT2 (1) H through PT5 (1) H are clear, the logic is in phase time one. If PHASE (1) H and PT2 (1) H are set, but PT3 (1) H through PT5 (1) H are clear, the logic is in phase time two. Similarly, if PHASE (1) H is clear and PBT2 (1) H through PBT5 (1) H are clear, the logic is in phase-bar time one. If PBT2 (1) H through PBT4 (1) H are set, but PHASE (1) H and PBT5 (1) H are clear, the logic is in phase-bar time four.

The PHASE (1) H flip-flop and the two shift registers are clocked on the leading edge of OSC (1) H. When PHASE (1) H is clear, the logic typically advances from one phase-bar time to the next. Usually, it advances from phase-bar time two to phase time one. However, there are two exceptions.

1. During address relocation cycles [REL CYC (0) H negated], the logic enters phase time one after phase-bar time five.

2. During reset cycles (ENB RST L clear), the logic enters phase time one after phase-bar time six.

When PHASE (1) H is set, the logic typically advances from one phase time to the next. However, there are the following exceptions.

1. The logic pauses in phase time one if microcycle enable [MCENB3 (1) H] is clear.

2. The logic pauses in phase time one if the last phase time was an address cycle [LAD CYC (1) H] and the address has not yet settled on the LSI-11 bus [GT BDAL 3 (0) H].

3. The logic enters phase-bar time one after phase time two during a chip set micro-NOP cycle (SHORT PT H).

4. The logic enters phase-bar time one after phase time three if both MED PT H and −FDIN ENB H are asserted. MED PT H is asserted during an F11 chip set address cycle or an F11 chip set data cycle that does not reference the LSI-11 bus, the IDAL bus, or an MMU register. In these two cases, MED PT H is asserted. −FDIN ENB H is negated during a fixed data cycle.

5. During an LSI-11 bus DOUT cycle, the logic pauses in phase time four until RRPLY3 (1) H clears. This only affects LSI-11 bus DATIO timing.

6. During an LSI-11 bus, IDAL bus, or MMU DIN cycle, the logic pauses in phase time four until it receives a reply signal. It then proceeds to phase time five.

7. During an LSI-11 bus, IDAL bus, or MMU DOUT cycle, the logic pauses in phase time five until it receives a reply signal. It then proceeds to phase-bar time one.

The PBT CLR L signal clears flip-flops that gate data onto the CDAL lines during phase time. That data must remain there for one-half an OSC period into phase-bar time. The PT CLR L signal clears flip-flops that gate data onto the CDAL lines during phase-bar time. That data must remain there for one-half an OSC period into phase time.

5.6 MIB DECODE LOGIC

The 16-bit microinstruction bus MIB <15:00> is common to all data and control chips. The MIB is time-multiplexed and is used for different functions during the clock cycle. During the clock phase-bar time, the MIB contains the current microinstruction provided by one of the F11 control chips. During the clock phase time, the MIB lines contain control information provided by the F11 data chip. The KDF11-BA logic monitors some MIB lines during phase time, some at the end of phase time, and some at the end of phase-bar time. The MIB decode logic is shown in Figure 5-3.

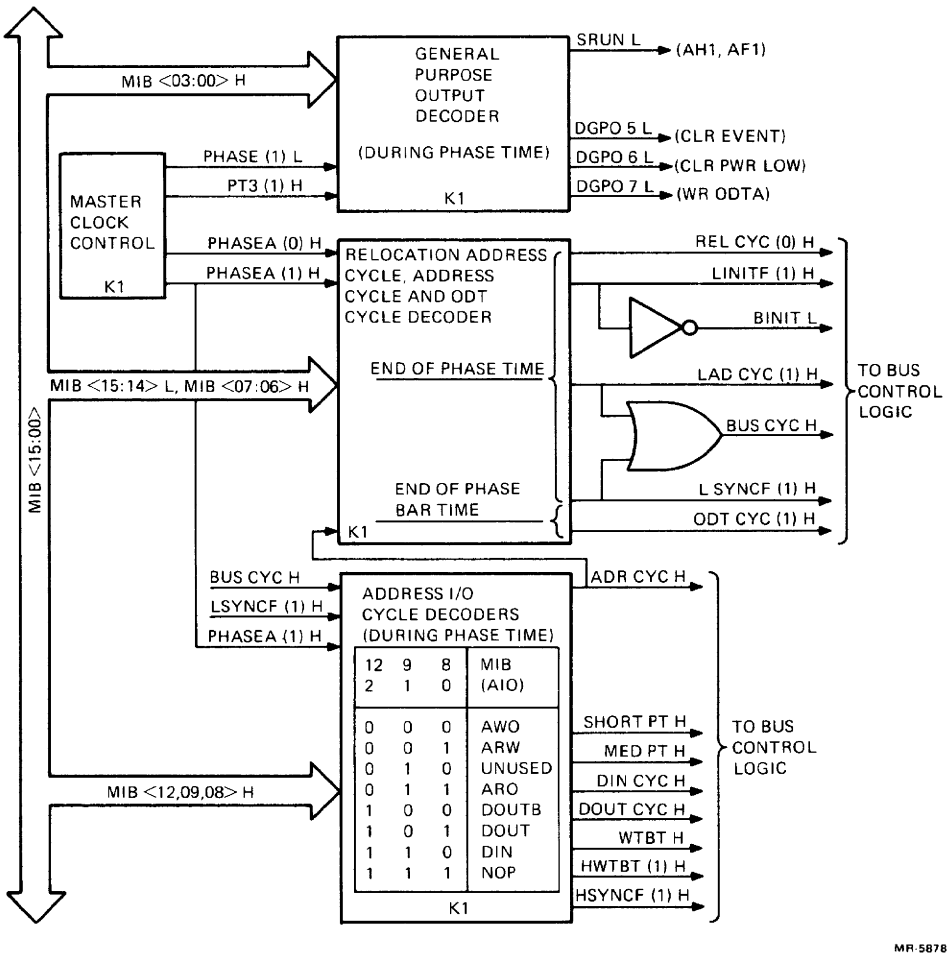


Figure 5-3 MIB Decode Logic

5.6.1 MIB Decode During Phase Time

During phase time, MIB lines <12,09,08> contain the address/input/output signals AIO <2:0> while MIB lines <03:00> contain the general-purpose outputs DGPO <3:0> L.

The AIO bits are decoded to determine whether the current cycle is an address cycle (ADR CYC H), a bus-type data-in cycle (DIN CYC H), or a bus-type data-out cycle (DOUT CYC H). Bus-type DIN and DOUT cycles are decoded only if BUS CYC H is asserted. The write/byte signal (WTBT H) is also decoded, as are two signals that determine whether the logic enters phase-bar time either after phase time two (SHORT PT H), phase time three (MED PT H), or phase time five (SHORT PT H and MED PT H both clear). Table 5-1 describes the decoded general-purpose output signals derived from MIB <02:00> when MIB 03 is negated.

The assertion of MIB 03 is used to set the read fixed data [RD FIXDT (1) L] flip-flop during phase time three. When the RD FIXDT (1) L flip-flop is set, the jumper-selected power-up mode and HALT/TRAP option information is gated onto the CDAL bus at the same time CDAL <08> is negated to specify boot address 173000.

Table 5-1 Decoded General-Purpose Output

GPO2 (MIB02)	GPO1 (MIB01)	GPO0 (MIB00)	Output Name	Function
1	1	1	DGP07 L	Loads the two highest order address bits into a latch while in micro-ODT. These two bits are necessary for 18-bit addressing because the memory management unit is disabled 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. This signal can be used to cause a steady RUN indication while the processor is fetching instructions, and a flashing indication when typing characters in console ODT.

5.6.2 MIB Decode at the End of Phase Time

The following MIB lines are clocked into flip-flops by PHASEA (0) H at the end of phase time.

1. MIB 15 H is clocked into relocation cycle [REL CYC (0) H].
2. MIB 12 H is inverted to produce address cycle (ADR CYC H), which is clocked into latched address cycle [LAD CYC (1) H].
3. MIB 07 H is inverted to produce SYNC H, which is clocked into LSYNCF (1) H. LSYNCF (1) H is used by the bus control logic to generate BSYNC L.
4. MIB 14 H is clocked into LINITF (0) H, which is inverted to produce BINIT L.

5.6.3 MIB Decode at the End of Phase-Bar Time

PHASEA (1) H clocks the ODT CYC (1) H flip-flop at the end of phase-bar time. The ODT CYC (1) H flip-flop sets if MIB 07 H is asserted and MIB 06 H is clear.

5.7 BUS CONTROL LOGIC

The logic described in this section controls the transfer of information between the F11 chip set and the LSI-11 bus, the transfer of information between the F11 chip set and the IDAL bus, and the transfer of LSI-11 bus ownership to DMA devices. Figure 5-4 shows the bus control logic interface to the LSI-11 bus and the internal KDF11-BA logic.

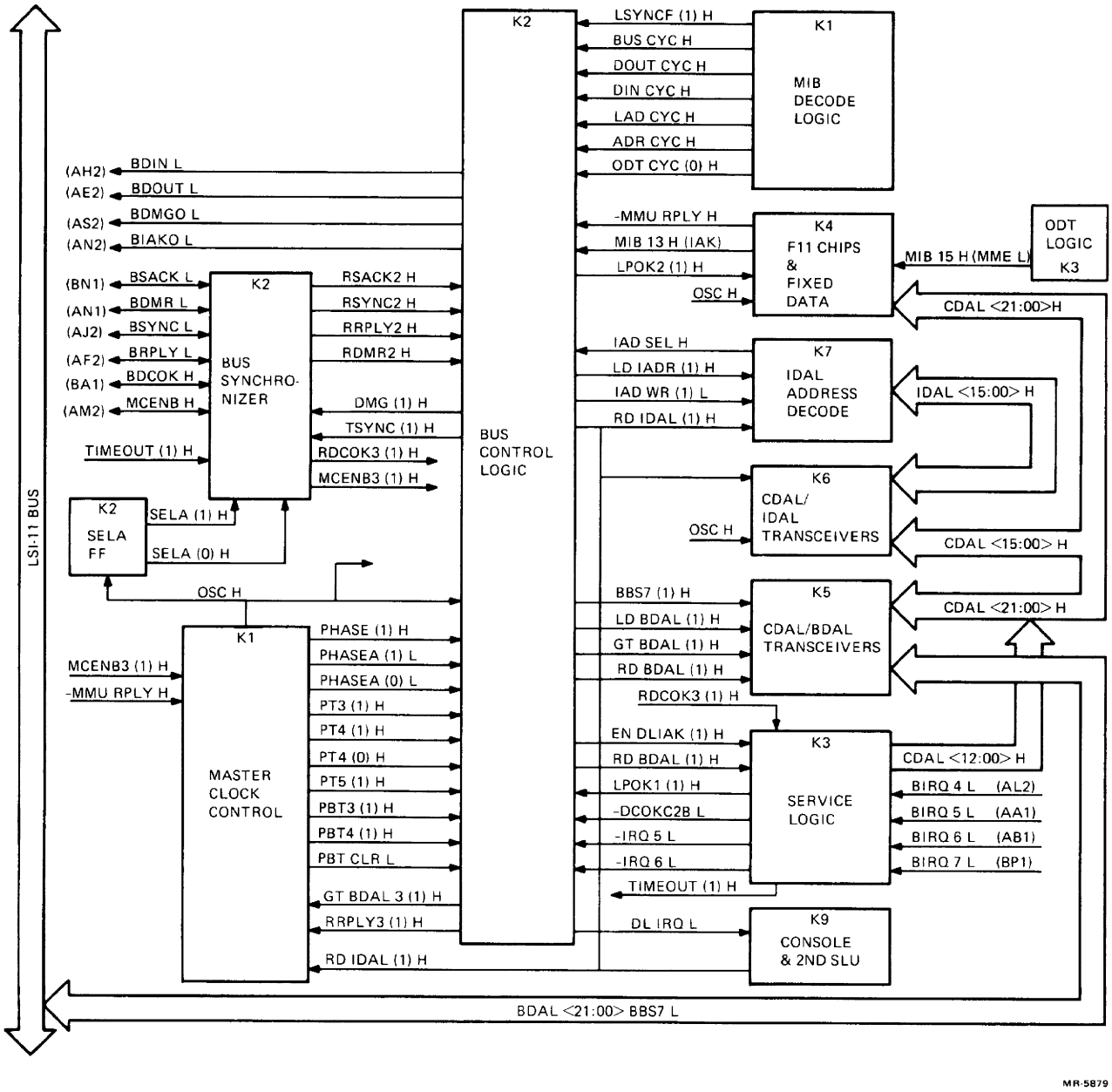


Figure 5-4 KDF11-BA Bus Control Interface

5.7.1 Bus Synchronizer Circuits

Because internal operation of the KDF11-BA is synchronous, asynchronous external signals must be synchronized before they can be used by the KDF11-BA logic. The BRPLY L, BSYNC L, BSACK L, and BDMR L signals received from the LSI-11 bus are time-critical and must be monitored as frequently as possible. At the same time, the registers that receive them must be allowed to settle for at least 100 ns to ensure reliable bus operation. The bus synchronizer contains special circuits that clock these four bus signals every OSC H period (75 ns), but allows them two OSC H periods (150 ns) to settle. A fifth LSI-11 bus signal BDCOK H, and the microcycle enable signal MCENB H, are not as time-critical; they are clocked every two OSC H periods.

5.7.1.1 BRPLY, BSYNC, BSACK, BDMR Synchronization – Each of the four bus signals (BRPLY L, BSYNC L, BSACK L, and BDMR L) is applied in parallel to a pair of D type flip-flops. These flip-flops are called the A and B synchronizer flip-flops. The outputs of the A and B flip-flops are connected to the A and B inputs of a multiplexer. The select input of the multiplexer is obtained from the SELA flip-flop, which is toggled by the leading edge of OSC H. The SELA (1) H output is used to clock the B flip-flops and also for the select input of the multiplexer. The SELA (0) H output is used only to clock the A flip-flops.

The BRPLY L signal is clocked into the B flip-flop by the SELA (1) H signal and into the A flip-flop by the SELA (0) H signal. The SELA (1) H signal connected to the multiplexer selects either the A or B flip-flop output to produce RRPLY2 H. When SELA (1) H is set, RRPLY2 H equals the A flip-flop output, which, by the next OSC H toggle, will have settled for two OSC H periods. This description applies to the BSYNC L, BSACK L, and BDMR L signals, which are used to produce the RSYNC2 H, RSACK2 H, and RDMR2 H signals, respectively.

5.7.1.2 BDCOK and MCENB Synchronization – The BDCOK H signal is applied to two B synchronizer flip-flops, which are clocked by SEL (1) H. The output of the second B flip-flop [(RDCOK3 (1) H)] is used to hold the KDF11-BA logic in the clear condition until RDCOK3 (1) H sets. This means that the KDF11-BA logic will remain in the clear condition for two OSC H periods (150 ns) after BDCOK H is asserted.

The MCENB H signal is applied to two A synchronizer flip-flops, which are clocked by SEL (0) H. The output of the second A flip-flop MCENB3 (1) H is sent to the phase time pause control logic of the master clock control. MCENB3 (1) H will clear two OSC H periods (150 ns) after MCENB H is negated. This signal is negated for manufacturing test purposes only.

5.7.2 Direct Memory Access (DMA) Control

DMA on the KDF11-BA module allows a peripheral to gain control of the LSI-11 bus from the processor and transfer data directly between that peripheral and memory. In this way, data transfers can occur at full memory speed rather than having the processor transfer data words one at a time between the peripheral and memory. Paragraph 4.4 presents the LSI-11 bus specification for granting DMA requests.

DMA Bus Grant Operation

A direct memory access (DMA) device requests control of the LSI-11 bus by asserting BDMR L. The KDF11-BA grants control of the LSI-11 bus to a requesting device by asserting BDMGO L. The following events occur during a KDF11-BA bus grant sequence.

1. BDMR L is received, inverted to RDMR H, synchronized, and delayed by two OSC H periods to become RDMR2 H.

2. After the KDF11-BA has released the LSI-11 bus [GTBDAL 1 (0) H and HLD BUS (0) H both asserted], RDMR2 H asserts DMG ENB H.

3. OSC H clocks DMG ENG H into the DMG (1) H flip-flop, which asserts BDMGO L.

4. DMG (1) H also triggers the NO-SACK timeout circuit in the bus synchronizer. If BSACK L is not asserted by the requesting DMA device within 10 μs after the KDF11-BA issues BDMGO L, the NO-SACK timeout circuit will negate ENB DMR (1) L. The negation of ENB DMR (1) L negates RDMR H, which leads to the negation of BDMGO L.

5. BSACK L is received, inverted to RSACK H, synchronized, and delayed by two OSC periods to become RSACK2 H. RSACK2 H is clocked into the RSACK3 flip-flop (located in the bus control logic) by OSC H to negate RSACK3 (0) H.

6. The negation of RSACK3 (0) H by OSC H causes the negation of BDMGO L to terminate the bus grant sequence.

5.7.3 Address Microcycle Control

The KDF11-BA may perform either a normal-address microcycle or a relocated-address microcycle, depending on whether the memory management unit is enabled or disabled. The memory management unit is enabled or disabled under program control.

A normal-address microcycle is a 16-bit direct byte address that references the first 32K words (64K bytes) of memory, and therefore, does not require the memory management function. A relocated-address microcycle is one that uses the MMU to convert a 16-bit program virtual address (VA) to an 18- or 22-bit physical (PA) address. When the MMU is enabled, the normal 16-bit direct byte address is no longer interpreted as a direct physical address but as a virtual address containing information to be used in constructing a new 18- or 22-bit address that is capable of referencing addresses in a 4 megabyte memory.

Microinstruction bus bit (MIB 15) is the memory management enable (MME L) signal that indicates to the processor logic whether a relocated-address microcycle should or should not be performed. The memory management unit asserts MME L when a relocated-address microcycle should be performed. MME is also asserted low by the KDF11-BA ODT logic during certain ODT address cycles.

The KDF11-BA logic stores the address provided by the F11 data chip during phase time if a normal-address microcycle is to performed or if an ODT cycle has been decoded. During a relocation-address microcycle (MMU asserts MME L), the address provided by the MMU is stored by the KDF11-BA logic during phase-bar time. The following events take place during an address microcycle.

1. The LD BDAL (1) H and LD BBS7 (1) H signals from the bus control logic clock the CDAL address into the BDAL registers at the end of phase time three.

2. The LD IADR (1) H signal from the bus control logic clocks the CDAL address into the latched internal address registers of the IDAL decode logic at the end of phase time three.

3. If no DMA device is using the LSI-11 bus, the GT BDAL (1) L signal from the bus control logic gates the address in the BDAL registers onto the BDAL lines at the end of phase time three. Note that during address relocation cycles, the BDAL registers do not contain a valid address until the end of phase-bar time three.

4. If a DMA device is using the LSI-11 bus, the GT BDAL1 (1) H signal is inhibited until the DMA device releases the bus by negating BSACK L. When the bus is released, OSC H sets GT BDAL1 (1) H, which gates the address in the BDAL registers onto the BDAL lines.

5. LD BDAL (1) H and LD BBS7 (1) H clock the CDAL address into the BDAL registers at the end of phase-bar time three when using the phase-bar time address (relocation address) from the MMU.

6. LD IADR (1) H clocks the CDAL address into the latched internal address registers of the IDAL decode logic at the end of phase-bar time four when using the phase-bar time address from the MMU.

7. GT BDAL1 (1) H remains set until the recognition of a bus DIN cycle (MIB decode logic asserts DIN CYC H), or the end of a bus DOUT cycle [DOUT 2 (1) H negates].

5.7.4 BSYNC Signal

The BSYNC L signal is asserted on the LSI-11 bus when the bus control logic asserts transmit synchronize [TSYNC (1) H]. All address cycles, except those that precede an interrupt-type DIN cycle, assert TSYNC (1) H. According to LSI-11 bus specifications, TSYNC (1) H must be set 150 ns (minimum) after the address is gated onto the BDAL lines. OSC L from the master clock control clocks TSYNC (1) H set if both LSYNCF (1) H and GT BDAL3 (1) H are set. The MIB decode logic will assert LSYNCF (1) H if MIB 07 H is negated at the end of phase time. The GT BDAL 3 (1) H signal in the bus control logic is set 150 ns after the address is gated onto the BDAL lines. If GT BDAL3 (1) H is clear, the logic pauses in phase time until it sets, thus assuring two and one-half oscillator periods between the time the address is gated on the BDAL lines and the assertion of BSYNC L. If LSYNCF (1) H is clear, the logic continues but does not set TSYNC (1) H.

Once set, TSYNC (1) H remains set until the HLD BUS (1) H signal in the bus control logic clears and the slave device negates BRPLY L. HLD BUS (1) H does not clear until GT BDAL2 (1) L and BUS CYC H both clear.

5.7.5 Noninterrupt Bus DIN Cycles

A noninterrupt bus DIN cycle (DATI) is a read operation. During a DIN microcycle of a DATI bus cycle, 16-bit data is input to the F11 chip set from the IDAL bus via the CDAL/IDAL transceivers or from the LSI-11 bus via the CDAL/BDAL transceivers. The MIB 13 H (IAK H) signal is not asserted during a noninterrupt bus DIN microcycle, and thus prevents the assertion of BIAKO L on the LSI-11 bus. The following events take place during a normal bus DIN cycle.

1. The TSYNC (1) H signal from the bus control logic is set one-half period before the end of phase time one and is inverted to assert BSYNC L.

2. BDIN L is asserted by DIN CYC H from the MIB decode logic one-half period into phase time three.

3. GT BDAL (1) H is cleared at the end of phase time three because the MIB decode logic has asserted DIN CYC H.

4. If the IAD SEL H signal from the IDAL address decode logic is asserted, RD IDAL (1) H gates the data on the IDAL lines onto the CDAL lines at the end of phase time four.

5. If the IAD SEL H signal is clear, RD BDAL (1) H gates the data on the BDAL lines onto the CDAL lines at the end of phase time four.

6. The master clock control causes the logic to pause in phase time four until it receives an indication that the data transfer is completed. The completion of the data transfer is indicated by the assertion of the BRPLY L signal, the RD IDAL (0) H signal from the bus control logic, or the MMU RPLY H signal from the memory management unit.

7. The CDAL data is clocked into the F11 chip set at the end of phase time five.

5.7.6 Interrupt-Type Bus DIN Cycles

The KDF11-BA may accept interrupts from either the on-board SLUs or from external devices. If the interrupt request is from an on-board device, the interrupt vector address is input to the F11 chip set via the CDAL/IDAL transceivers. If the interrupt request is from an external device, the input vector address is input to the F11 chip set from the LSI-11 bus via the CDAL/BDAL transceivers.

The F11 chip set asserts MIB 13 H (IAK H) during interrupt type bus DIN cycles. IAK H causes the assertion of the BIAKO L bus signal to acknowledge the honoring of an external or internal interrupt request. The following events take place during an interrupt-type DIN cycle.

1. The KDF11-BA SLUs request an interrupt by asserting DL L. If one of the SLUs is requesting an interrupt, and if no higher interrupt request is pending (BIRQ 5 L and BIRQ 6 L negated), the bus control logic asserts EN DLIAK (1) H at the beginning of phase time one.

2. Because LSYNCF (1) H is clear, TSYNCF (1) H does not set.

3. The assertion of DIN CYC H from the MIB decode logic causes DIN (1) H to set one-half period into phase time three. When DIN (1) H sets, it causes the assertion of BDIN L.

4. GT BDAL1 (1) H in the bus control logic is cleared at the end of phase time three by the assertion of DIN ENB H.

5. If the EN DLIAK (1) H signal in the bus control logic is set, RD IDAL (1) H is set one period into phase time four. When RD IDAL (1) H sets, one of the four KDF11-BA SLU vector addresses placed on the IDAL lines is input to the F11 chip set via the CDAL/IDAL transceivers.

6. If EN DLIAK (1) H is clear, RD BDAL (1) H is set one period into phase time four and TIAK (1) H is clocked set one period after GT BDAL3 (1) H is clocked clear. When TIAK (1) H sets, it causes the assertion of BIAKO L. The vector address input from the external device is then read from the BDAL lines to the CDAL lines.

7. The master clock control causes the logic to pause in phase time four until it receives an indication that the vector address transfer is completed. The completion of the vector transfer is indicated by the assertion of the BRPLY signal or by the negation of the RD IDAL (0) H signal from the bus control logic.

8. The CDAL data is clocked into the F11 chip set at the end of phase time five.

5.7.7 Bus DOUT Cycle

A bus DOUT cycle is a write operation. During a DOUT microcycle of a DATO(B) bus cycle, 16-bit words (DATO) or 8-bit bytes (DATOB) are output by the F11 chip set to an IDAL register via the CDAL/IDAL transceivers, or to an external device via the CDAL/BDAL transceivers. The following events take place during a DOUT cycle.

1. LD BDAL (1) H clocks the CDAL data into the BDAL registers at the end of phase time three.

2. The CDAL data is also gated onto the IDAL lines.

3. The bus control logic clocks the internal address write [IAD WR (1) L] signal on at the end of phase time two and clocked off one period into phase time five. If the PLA in the IDAL address decode logic has selected one of the writable registers, the IAD WR (1) L signal gates the load signal (e.g., LD PCR LO L) to the selected IDAL bus register.

4. The master clock control causes the logic to pause in phase time four until the bus control logic negates RRPLY3 (1) H. This occurs only during DATIO cycles.

5. The DOUT CYC H signal from the MIB decode logic is clocked into DOUT (1) H of the bus control logic one period into phase time five. When DOUT (1) H sets, it causes the assertion of BDOUT L.

6. The master clock control causes the logic to pause in phase time five until the MMU asserts MMU RPLY H, an external device asserts BRPLY L or, if IAD SEL H is asserted, until the bus control logic negates IAD WR (1) H.

7. The negation of GT BDAL (1) H is delayed by the setting and clearing of DOUT2 (1) H. DOUT2 (1) H is set one period after DOUT (1) H sets (after BDOUT L is asserted). DOUT2 (1) H clears one period after DOUT1 (1) H clears.

5.8 CDAL/BDAL INTERFACE

The CDAL/BDAL transceivers transfer information between the LSI-11 bus and the F11 chip set in response to load, gate, and read signals obtained from the bus control logic. The CDAL/BDAL bus interface is shown in Figure 5-5.

The LD BDAL (1) H and LD BBS7 (1) H signals are used to load the information on the CDAL bus into the transceiver registers during a data-out (write) bus cycle. The GT BDAL (1) L signal gates the contents of the transceiver registers onto the LSI-11 bus. The RD BDAL (1) L signal enables the transceiver registers during a data-in (read) bus cycle to transfer the information on the BDAL <18:00> L lines to the CDAL bus.

The MIB decode logic decodes MIB <7:6> and sets or clears the ODT CYC flip-flop at the end of phase-bar time to indicate whether the current cycle is a normal address cycle or an ODT address cycle. The ODT CYC flip-flop is clear during a normal address cycle and set during an ODT cycle.

During normal address cycles, BBS7 L is asserted if BSIO H is asserted by the F11 data chip or the MMU chip. During ODT address cycles, the ODT logic performs an ODT relocation cycle to clear BBS7 L if either ODTA17 (1) H or ODTA16 (1) H is clear. The assertion of BBS7 L indicates that an address references the 8K-byte I/O page.

The HWTBT (1) H signal is loaded into the CDAL/BDAL transceivers and gated onto the LSI-11 bus during the address and data portions of the data-out bus cycle. The MIB decode logic decodes MIB <9:8> to set or clear the HWTBT latch. The HWTBT (1) H signal is set during the address portion of a data-out bus cycle. During the data portion of the data-out bus cycle, HWTBT (1) H is set to indicate a write byte operation and clear to indicate a write word operation. The state of HWTBT is not gated onto the LSI-11 bus during a data-in cycle.

The BDAL <17:16> L bits are used by the service logic for an address parity check. The bits are transferred as RDAL <17:16> H to the service logic by RD BDAL (1) L during a data-in bus cycle. If both RDAL bits are set, an address parity error [PAR ERR (1) H] is generated by the service logic at the end of phase-bar time. PAR ERR (1) H is used by the reset logic to reset all of the F11 chips except the MMU chip. PAR ERR (1) H also causes the program to trap to location 114_8.

5.9 SERVICE, RESET, AND ODT LOGIC

The service logic gates information onto the CDAL <12:07,05:00> lines in response to various external and internal KDF11-BA conditions. The service logic operation is described in Paragraph 5.9.1. The reset logic monitors various error signal inputs and generates a reset signal for the bus control logic and the F11 chip set if an error is detected. The reset logic operation is described in Paragraph 5.9.2. The service and reset logic interface to the LSI-11 bus, CDAL bus, and other KDF11-BA logic is shown in Figure 5-6.

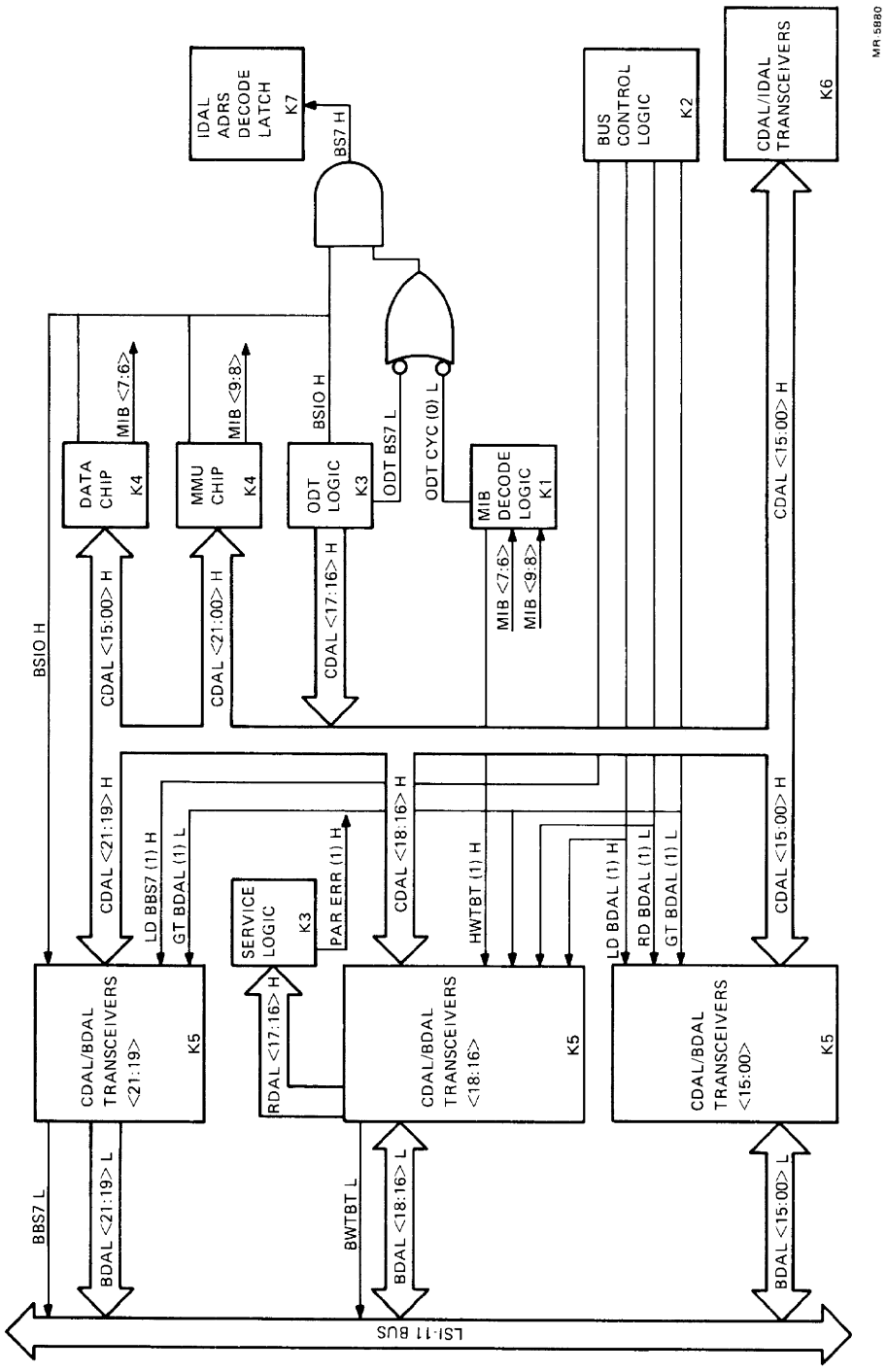


Figure 5-5 CDAL/BDAL Interface

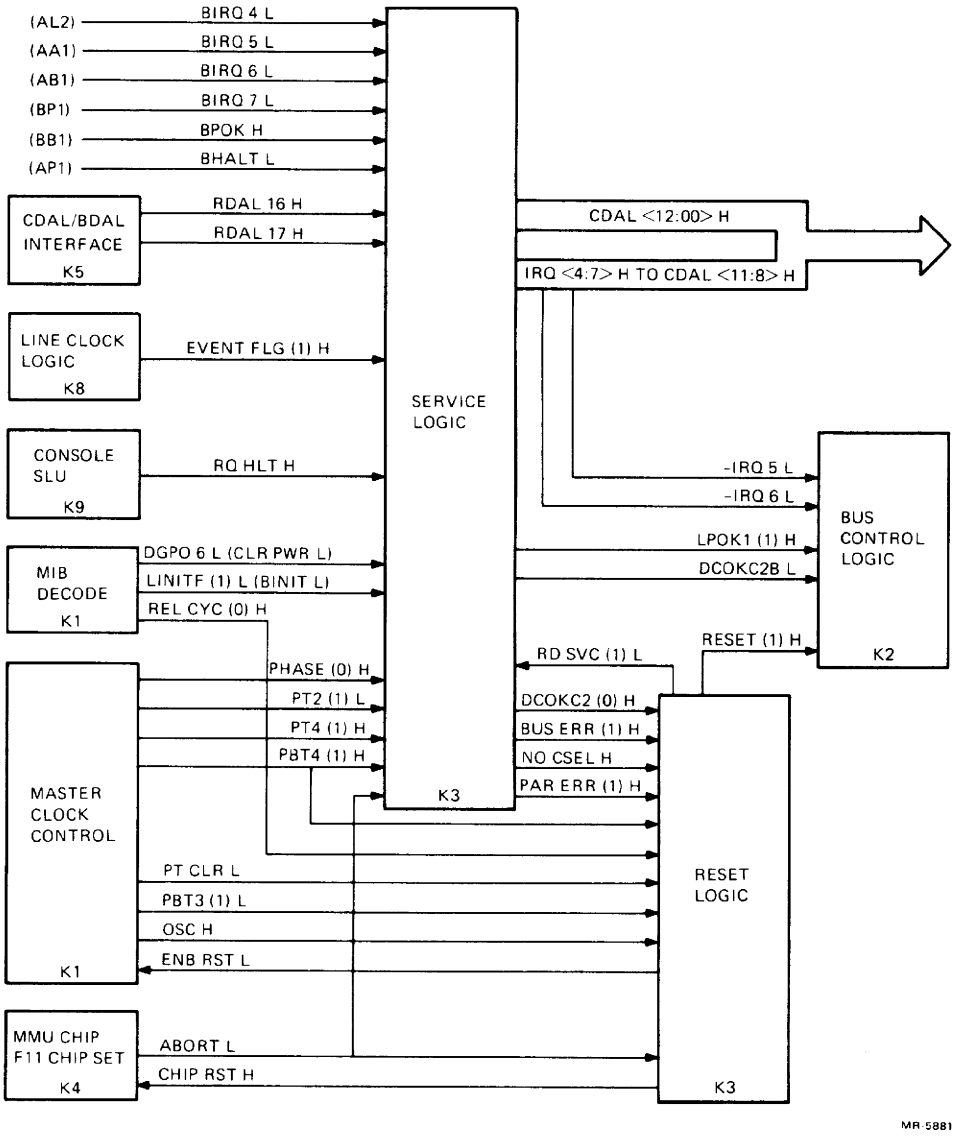


Figure 5-6 Service and Reset Logic Interface

The ODT logic gates the ODT address onto the CDAL <21:16> lines during ODT address cycles. The operation of the ODT logic is described in Paragraph 5.9.3 and the logic interface is shown in Figure 5-7.

5.9.1 Read Service Operation

The CDAL <12:00> lines contain service information during phase-bar time if a relocated address is not on the CDAL bus. The read service RD SVC (1) L signal from the reset logic gates the service information onto the CDAL <12:00> lines at the end of phase-bar time one, or when an MMU abort occurs at the beginning of phase-bar time four. RD SVC (1) L is cleared by phase time clear PT CLR L one-half period into phase time one.

Thirteen service information bits are placed on the CDAL <12:00> lines by tri-state drivers or tri-state registers. Five of the CDAL lines <06,04:02,00> are driven by tri-state drivers when RD SVC (1) L is asserted. The signal names and functions for the CDAL lines are described in Table 5-2.

Table 5-2 Service Logic Bits <06,04:02,00>

CDAL Line	Signal Name	Function
CDAL 06 H	Ground	CDAL 06 H is always asserted.
CDAL 04 H	CTL ERR (1) L	The assertion of this bit indicates that none of the F11 control chips asserted either CSELA L or CSELB L.
CDAL 03 H	ABORT L	Negation of this bit indicates the occurrence of an MMU abort.
CDAL 02 H	PAR ERR (1) L	Assertion of this bit indicates a parity error.
CDAL 00 H	DCOKC3 (1) L	Assertion of this bit indicates that the LSI-11 bus BDCOK H signal has been valid for at least three phase-to-phase-bar transitions.

The remaining eight CDAL lines <12:07,05,01> are driven by a tri-state register. The signals described in Table 5-3 are clocked into the register by PHASE (0) H at the beginning of phase-bar time and placed on the CDAL lines when RD SVC (1) L is asserted.

Table 5-3 Service Logic Bits <12:07,05,01>

CDAL Line	Signal Name	Function
CDAL 12 H	EVENT FLG (1) H	The assertion of this bit posts a line clock interrupt request.
CDAL 11 H	IRQ 4 H	The assertion of this bit posts a level-4 interrupt request.
CDAL 10 H	IRQ 5 H	The assertion of this bit posts a level-5 interrupt request.
CDAL 09 H	IRQ 6 H	The assertion of this bit posts a level-6 interrupt request.
CDAL 08 H	IRQ 7 H	The assertion of this bit posts a level-7 interrupt request.
CDAL 07 H	PWR DWN (1) H	The assertion of this bit posts a power-down interrupt request.
CDAL 05 H	HALT H	This bit reflects the state of the LSI-11 bus BHALT L line.
CDAL 01 H	TIMEOUT (1) H	The assertion of this bit indicates that a bus timeout has occurred.

5.9.2 F11 Chip Reset Operation

The reset logic generates an F11 chip reset (CHIP RST H) signal for any one of five error conditions that require immediate attention by the chip set. The CHIP RST H signal is routed to all the F11 chips except the MMU chip. The CHIP RST H signal is asserted high for any one of the five following conditions.

1. Control error – A nonexistent control chip is selected by the microcode.
2. Bus error – A nonexistent memory location is accessed.
3. Parity error – A parity error is detected on a current read from memory.
4. DC power-up – Upon power-up the processor forces the reset logic to assert CHIP RST H to initialize all internal chip registers. The dc power-up line then clears and is not reactivated while dc power is on.
5. MMU abort – The MMU has aborted a mapped memory reference. The MMU chip will assert ABORT L for any of the following reasons.
   - The memory location referenced is not present in the current user's protected address space.
   - An attempt is made to modify a write-protected location.
   - The user is exceeding his allotted page boundary.

The reset logic input and output signals are shown in Figure 5-6. The CHIP RST H signal is obtained from a RESET flip-flop that is clocked set at the beginning of phase-bar time four if any one of four error signals from the service logic is asserted. The service logic error signals are applied to a NOR gate to produce an enable reset (ENB RST L), which is applied to the D input of the RESET flip-flop and the master clock control. The assertion of ENB RST L extends phase-bar time through phase-bar time six.

The ABORT L signal generated by the MMU chip as a result of a memory error is applied to the set input of the RESET flip-flop when phase-bar time three begins. The signal names and functions of the error signals that cause assertion of CHIP RST H are described in Table 5-4. The RESET flip-flop is cleared by PT CLR L from the master clock control one-half period after phase time one.

Table 5-4 F11 Chip Reset Signals

Signal Name	Function
DCOKC3 (0) H	Assertion of this bit indicates that the LSI-11 bus BDCOK H signal has been valid for less than three phase time-to-phase-bar transitions.
PAR ERR (1) H	Assertion of this bit indicates a parity error; a trap to location 114_8 occurs.
BUS ERR (1) H	Assertion of this bit indicates a bus timeout; a trap to location 4_8 occurs.
NO CSEL H	Assertion of this bit indicates that none of the F11 control chips asserted either CSELA or CSELB; a trap to location 10_8 occurs.
ABORT L	Assertion of this bit indicates an MMU abort; a trap to location 250_8 occurs.

5.9.3 ODT Address Logic

During ODT addressing cycles the processor responds to commands and information entered via the console terminal addresses 777560_8 through 777566_8. The ODT logic interface to the CDAL bus and other KDF11-BA logic is shown in Figure 5-7.

The ODT logic contains a flip-flop (PT2D) that is used to gate the CDAL <21:16> and MIB 15 H drivers. PT2D is clocked set during every address cycle at the beginning of phase time two and cleared one-half period into phase-bar time one by PBT CLR L. The ODT logic also contains an ODT ADR flip-flop that gates the ODT address bits <17:16> onto CDAL <17:16> H, controls ODT address relocation by assertion or negation of MIB 15 H, and enables or disables the MMU by assertion or negation of DMMUS L.

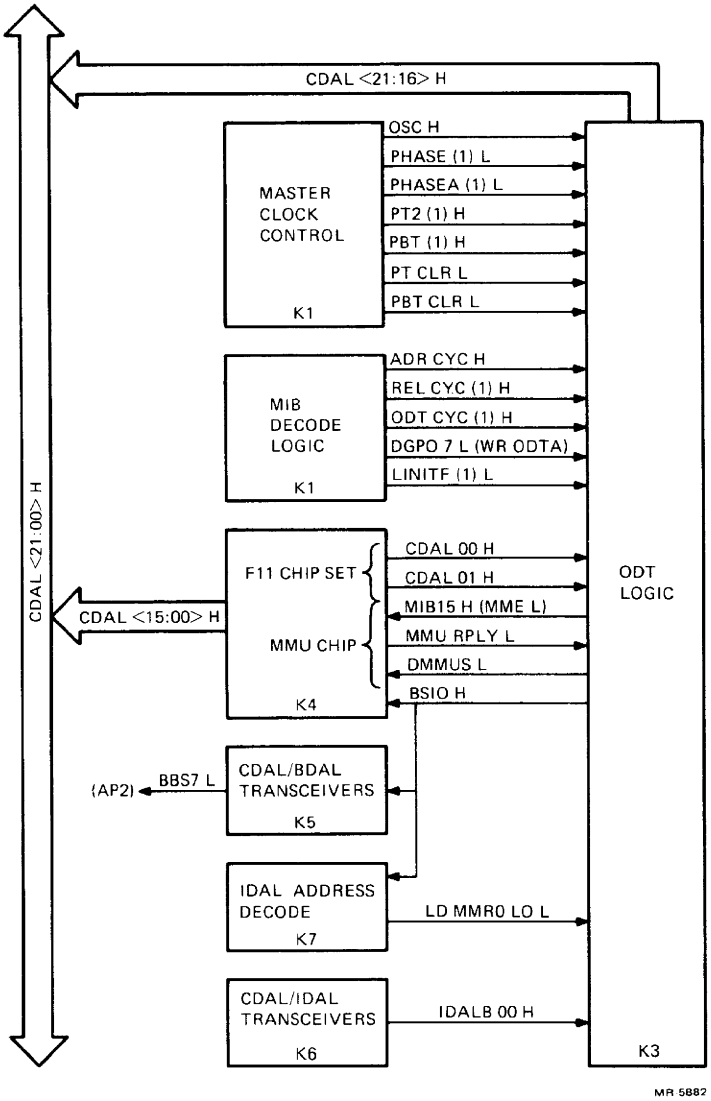


Figure 5-7 KDF11-BA ODT Logic Interface

During ODT address cycles the ODT ADR flip-flop is clocked set by OSC H at the end of phase time two and cleared one-half period into phase-bar time. If the ODT ADR flip-flop is set, the PD2D flip-flop negates CDAL <21:18> H and gates ODT address bits <17:16> onto CDAL <17:16> H. During normal address cycles the ODT ADR flip-flop is clear and the PT2D flip-flop negates CDAL <21:16> H. The remaining address bits (<15:00>) are gated onto CDAL <15:00> H by the F11 data chip.

If the two ODT address bits, ODT17 (1) H and ODT16 (1) H, are both set, the phase time address and BSIO H signal are correct and there is no need to prevent the DAT and MMU chips from responding to their I/O page addresses. However, if either of the ODT address bits is clear, the ODT logic must guarantee that the BBS7 L register bit is negated and that the DAT and MMU chips do not incorrectly respond to an asserted BSIO H during phase time. Therefore, if either ODT17 (1) H or ODT16 (1) H is clear, the ODT ADR flip-flop asserts DMMUS L to disable the MMU registers, and negates MIB 15 H to force an ODT address relocation cycle. The assertion of DMMUS L prevents the MMU from decoding CDAL<12:00> for an MMU register address.

The ODT address relocation cycle negates BSIO H and CDAL <21:19>, but presents meaningless data on CDAL <18:00>. This relocation cycle performs two functions. First, it prevents the DAT chip from incorrectly responding to a phase time PS address of 177776 on CDAL <15:00>. The DAT chip sees a relocated address with BSIO H negated. Second, it loads the negated BSIO H signal into the BBS7 L register bit with the load signal LD BBS7 (1) H. Note that LD BDAL (1) H also updates the BDAL <21:19> L register bits even though they were correctly negated during phase time.

5.10 FIXED DATA DIN CYCLES

The fixed data logic shown in Figure 5-8 gates the jumper-selected power-up mode and HALT/TRAP options onto the CDAL bus during DIN cycles. If MIB 03 H is asserted during phase time, the RD FIXDT flip-flop is set at the end of phase time two and cleared one-half period into phase-bar time one. RD FIXDT L enables tri-state drivers that gate the following status bits onto the CDAL lines.

CDAL Line	Input Signal and Purpose
CDAL 08 H	Ground. When power-up mode 2 is selected (CDAL bits 01–00 below), this bit specifies boot address 773000.
CDAL 07 H	LPOK2 (1) L. The assertion of this bit indicates that the LSI-11 bus BPOK H signal is asserted.
CDAL 02 H	TRAP OPJ L. Assertion of this signal indicates that the trap option jumper has been installed.
CDAL 01 H	PUP CD1J L. Assertion of this signal indicates that the power-up code bit 01 jumper has been installed.
CDAL 00 H	PUP CD0J L. Assertion of this signal indicates that the power-up code bit 00 jumper has been installed.

5.11 CDAL/IDAL INTERFACE

The CDAL/IDAL transceivers transfer address and data information to and from on-board peripheral devices connected to the IDAL bus and other internal KDF11-BA logic and/or the LSI-11 bus via the CDAL bus. The on-board peripheral devices are the console and second serial-line units, the bootstrap/diagnostic ROMs and the line clock. The CDAL/IDAL interface is shown in Figure 5-9.

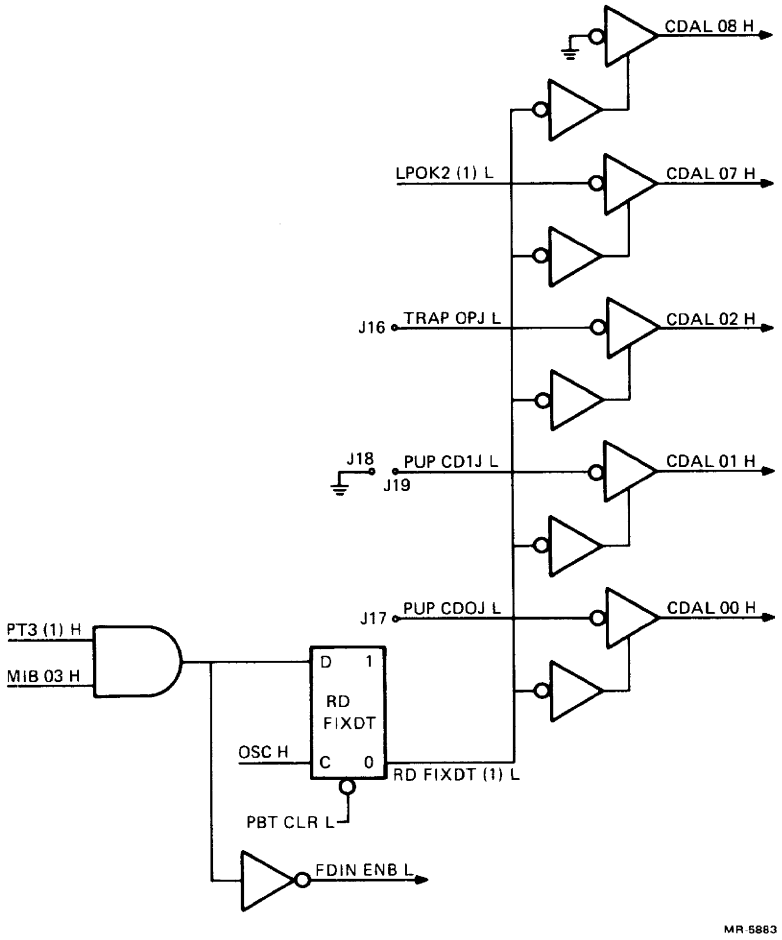


Figure 5-8 Fixed Data Logic

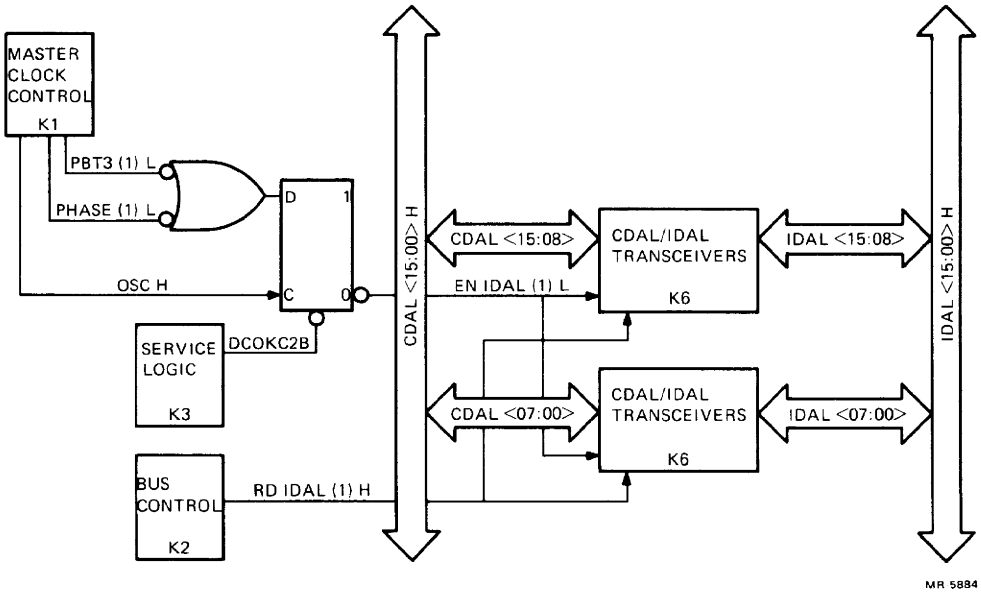


Figure 5-9 CDAL/IDAL Interface

The CDAL/IDAL transceivers are enabled by the EN IDAL (1) L signal. The EN IDAL flip-flop is clocked clear one-half period into phase-bar time one; this disables the CDAL/IDAL transceivers. The CDAL/IDAL transceivers are enabled when the EN IDAL flip-flop is clocked set one-half period into phase-bar three or one-half period into phase time one, whichever occurs first.

The direction of data transfer through the CDAL/IDAL transceivers is controlled by the RD IDAL (1) H signal obtained from the bus control logic. The RD IDAL (1) H signal is normally low, causing the transfer of data from the CDAL bus to the IDAL bus. When an IDAL bus register or vector address is read, RD IDAL (1) H goes high at the end of phase time four and is cleared at the end of phase-bar time one. RD IDAL (1) H causes the transfer of data from the IDAL bus to the CDAL bus.

5.12 IDAL ADDRESS DECODE

The IDAL address decode logic decodes the IDAL <12:00> H address bits and generates read and load signals for the serial-line units, the bootstrap/diagnostic ROMs, the bootstrap/diagnostic registers, and the line clock register. The IDAL address decode logic is shown in Figure 5-10.

The LD IADR signal from the bus control logic clocks BS7 H and IDAL <12:00> H into the latched internal address register at the end of phase time three of a normal address cycle, or at the end of phase-bar time four of a normal address relocation. Because BS7 H reflects BSIO H gated by the assertion of ODT17 (1) H and ODT16 (1) H, ODT relocation cycles do not update the internal address register. The outputs of the latched internal address register are sent to the programmable logic array (PLA), ROM read decode logic, and the load and read decoders.

The PLA decodes address, control, and jumper signals to produce signals that control the loading and reading of various IDAL bus registers. The PLA inputs may be subdivided as follows.

1. Four wirewrap jumpers:

   - J15 BDK DISJ L, when asserted, disables the boot and diagnostic registers, the boot and diagnostic ROMs, and the line clock register.
   - J14 DL1 DISJ L, when asserted, disables the console SLU registers.
   - J13 DL2 DISJ L, when asserted, disables the second SLU registers.
   - J12 DL2 ADRJ L, when asserted, changes the base address of the second SLU from 17776500_8 to 17776540_8.

2. HWTBT H – This signal is always clear during read data transfers, clear for write-word data transfers, and set for write-byte data transfers.

3. HSYNCF H – This signal is clear when reading a vector address and set for a normal read or write.

4. This signal is asserted (low) if LBS7 (1) H, LA12 (1) H, and LA10 (1) H are all asserted. This signal is asserted for all IDAL references. However, it is not decoded for vector address references.

5. This signal is asserted (low) if LA08 (1) H and LA06 (1) H are both asserted while LA07 (1) H is negated. This signal is asserted for all IDAL register references. It is not decoded for either boot and diagnostic ROM references or for vector address references.

6. LA <11,09,05:01> H – These seven address signals are individually decoded.

7. LA <00> H – If HWTBT H is asserted, this address bit determines whether the high byte or low byte is referenced.

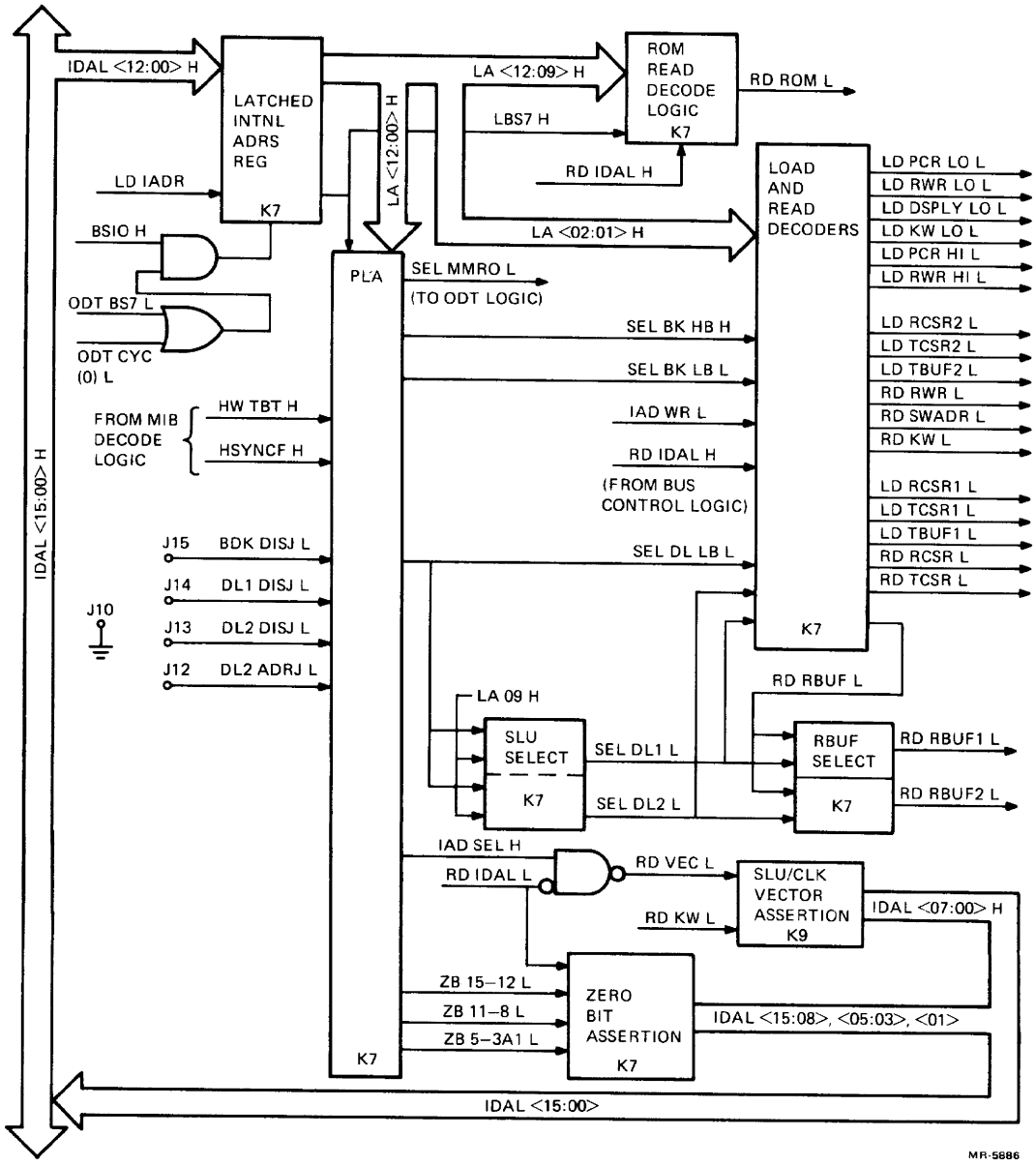


Figure 5-10 IDAL Address Decode Logic

The eight PLA outputs are sent to the load and read decoders, the SLU select logic, the SLU/CLK vector assertion logic, and the zero bit assertion logic. The signal names and functions of the PLA outputs are as follows.

1. SEL MMRO LB L – Asserted for low-byte references to memory management status register (SRO) at address 17777572_8.

2. SEL DL LB L – Asserted for low-byte references to device and vector addresses selected by the following jumper configurations.

   - Device addresses 17777560_8 through 17777566_8 if DL1 DISJ L (J14) is ungrounded.
   - Device addresses 17776500_8 through 17776506_8 if DL2 DISJ L (J13) and DL2 ADRJ L (J12) are both ungrounded.
   - Device addresses 17776540_8 through 17776546_8 if DL2 DISJ L (J13) is ungrounded and DL2 ADRJ L (J12) is grounded.

3. SEL BK HB H – Asserted for high-byte references to addresses 17777521_8 through 17777525_8 and 17777547_8 if BDK DISJ L (J15) is ungrounded.

4. SEL BK LB L – Asserted for low-byte references to addresses 17777520_8 through 17777524_8 and 17777546_8 if BDK DISJ L (J15) is ungrounded.

5. ZB-3A1 L – Asserted for all register references for which, when read, register bits <05:03,01> are always zero.

6. ZB11-08 L – Asserted for all register references for which, when read, register bits <11:08> are always zero.

7. ZB15-12 L – Asserted for all register references for which, when read, register bits <15:12> are always zero.

8. IAD SEL H – Asserted if SEL DL LB L, SEL BK HB H, or SEL BK LB L is asserted. Also asserted for valid DL high-byte references and valid references to the boot and diagnostic ROM addresses.

The various load and read control signals are produced by three 74LS155 decoders and associated logic. The load signals are sent to the SLU registers, the page control register, the read/write maintenance register, the boot/diagnostic display register, and the line clock logic when the IAD WR L signal from the bus control logic is asserted.

The read signals are sent to the SLU registers, the boot configuration register, the SLU/CLK vector assertion logic, and the read/write maintenance register when the RD IDAL H signal from the bus control logic is asserted. The ROM read decode logic asserts RD ROM L when the RD IDAL H signal is asserted. The RBUF select logic uses the RD RBUF L signal from the load and read decoders to read the contents of either the SLU receiver buffer RD RBUF1 L or RD RBUF2 L.

5.13 BOOTSTRAP/DIAGNOSTIC AND LINE CLOCK LOGIC

Figure 5-11 shows the bootstrap/diagnostic and line clock logic.

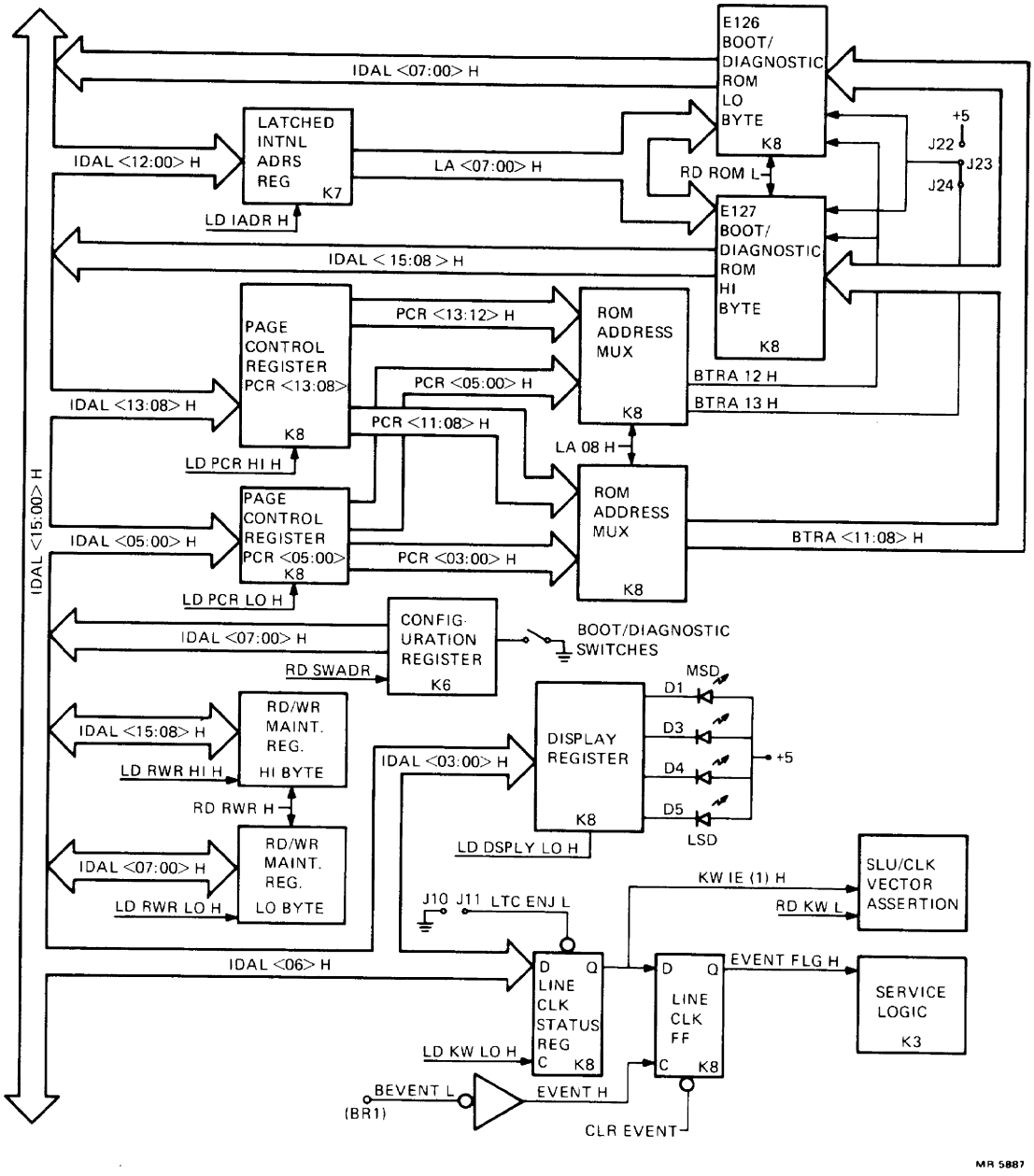


Figure 5-11 Bootstrap/Diagnostic and Line Clock Logic

5.13.1 Boot and Diagnostic Logic

The boot and diagnostic page control register consists of two bytes, PCR 13–08 (1) H and PCR 05–00 (1) H, each located in a hex register. These registers are loaded from the IDAL lines by −LD PCR LO H and by −LD PCR LO H, respectively. A pair of quad multiplexers produce the six most significant ROM address bits, BTRA 13–08 H. If LA 08 (1) H is set, BTRA 13–08 H equals PCR 13–08 (1) H. If LA 08 (1) H is clear, BTRA 13–08 H equals PCR 05–00 (1) H.

The boot and diagnostic ROM sockets accept pin-compatible 2K, 4K, and 8K ROMs. These ROMs are addressed by BTRA 13–08 and by latched internal address bits LA <07:01>. A wirewrap jumper can replace BTRA13 H with +5 V for 2K EPROMs. The ROM data is gated directly onto IDAL <15:00> by RD ROM L. The KDF11-BA uses 2K ROMs that are compatible with the BDV11.

The boot and diagnostic read/write maintenance register consists of two bytes located in a pair of 8-bit universal shift/storage registers. These registers are loaded from IDAL <15:00> by −LD RWR HI H and −LD RWR LO H. RD RWR H gates their contents onto IDAL <15:00>.

The boot and diagnostic write-only display register consists of four bits located in a quad register. −LD DSPLY LO H loads this register with data from IDAL <03:00>. Clearing one of the four display register bits lights a corresponding LED mounted at the top of the KDF11-BA module.

The boot and diagnostic read-only switch register consists of eight switches that are gated onto IDAL <07:00> by RD SWADR L. The remaining IDAL lines are negated by the zero bit logic driven by the PLA outputs ZB 15–12 L and ZB 11–9 L. (See Figure 5-10.)

5.13.2 Line Clock Register

the line clock register contains a single read-write bit, KW IE (1) H. This bit is loaded from IDAL 06 H by −LD KW LO H. KW IE (1) H is held set when the LTC ENJ L signal is asserted by a wirewrap jumper. The LSI-11 bus BEVENT L signal is received as EVENT H. If KW IE (1) H is set, the leading edge of EVENT H sets EVENT FLG (1) H.

The logic for reading the line clock register consists of a quad multiplexer that gates KW IE (1) H onto IDAL 06 H and negates IDAL 07, 02, and 00 H. The remaining IDAL lines are negated by the zero bit logic driven by the PLA outputs ZB 15–12 L, ZB 11–9 L, and ZB 5–3A1 L. (See Figure 5-10.)

5.14 SERIAL-LINE UNITS

Figure 5-12 (Sheets 1 and 2) shows the logic associated with the serial-line units.

5.14.1 Universal Asynchronous Receiver Transmitters

Each serial line unit is based on a universal asynchronous receiver transmitter circuit, contained in a single 40-pin package (Digital Part No. 21-13937-01). Each UART contains a receiver section and a transmitter section.

The receiver section contains receiver data buffer bits 07–00 and receiver status register bit 07. Serial data (SERIAL IN1 H or SERIAL IN2 H) is clocked into a receiver shift register and then transferred to the data buffer. Loading the data buffer sets the status bit (RX1 DONE H or RX2 DONE H). The read control signal (RD RBUF1 L or RD RBUF2 L) gates the data buffer onto IDAL 07–00 H and clears the status bit.

The transmitter section contains transmitter data buffer bits 07–00 and transmitter status register bit 07. The write control signal (LD TBUF1 L or LD TBUF2 L) loads IDAL 07–00 H into the data buffer and clears the status bit (TBMT1 H or TBMT2 H). The contents of the data buffer is loaded into the transmitter shift register (as soon as that register is empty) and then clocked out as serial data (SERIAL OUT1 H or SERIAL OUT2 H). The status bit is set when the data buffer is empty and able to receive another character.

For each UART the receiver and transmitter clock inputs are driven by the same clocking signal (RT CLK1 H or RT CLK 2 H). The clock rate is 16 times the serial data rate.

Character formats are selected by wirewrap jumpers and may consist of seven or eight data bits, one or two stop bits, parity or no parity, and odd or even parity.

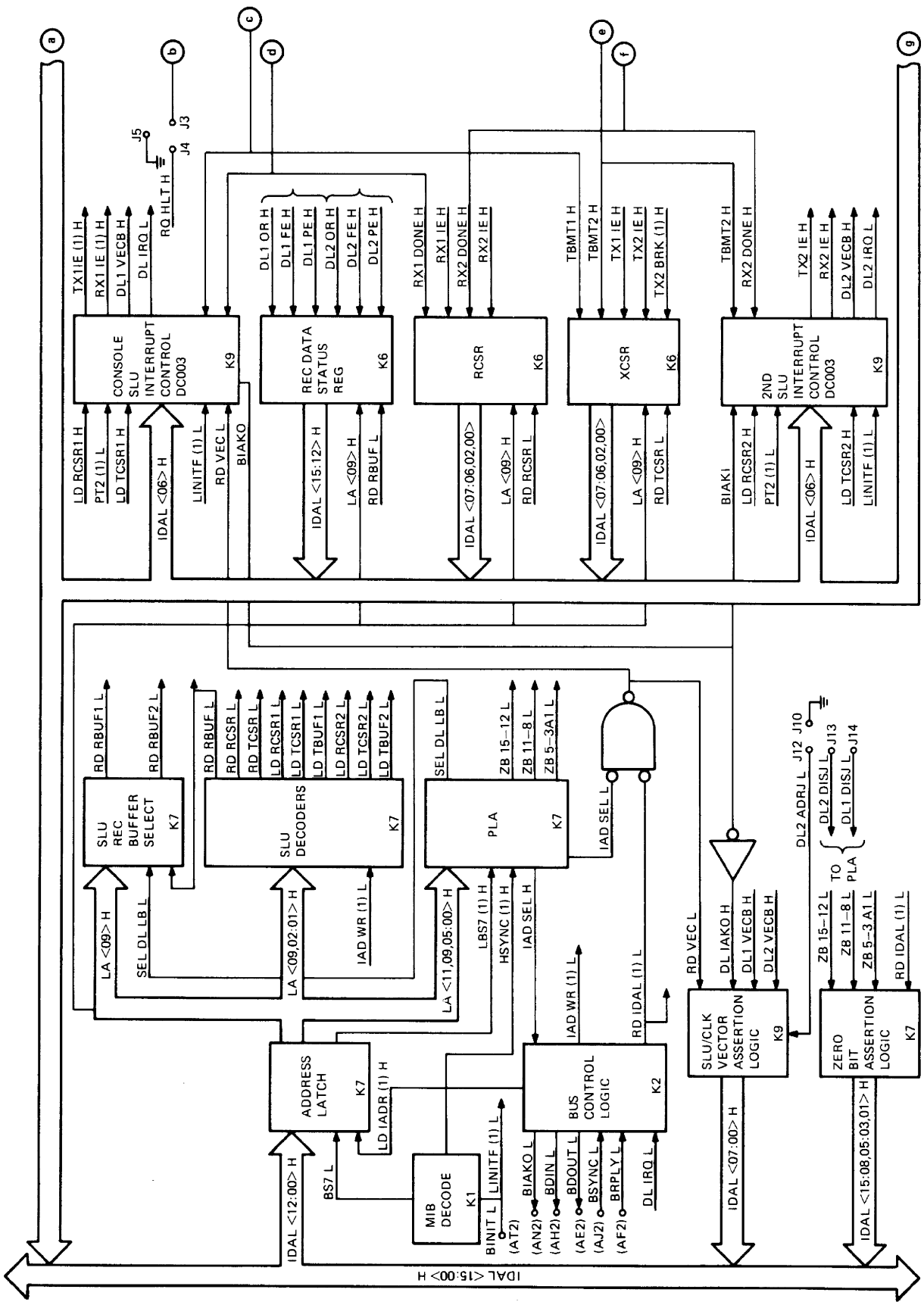


Figure 5-12 Serial-Line Units (Sheet 1 of 2)

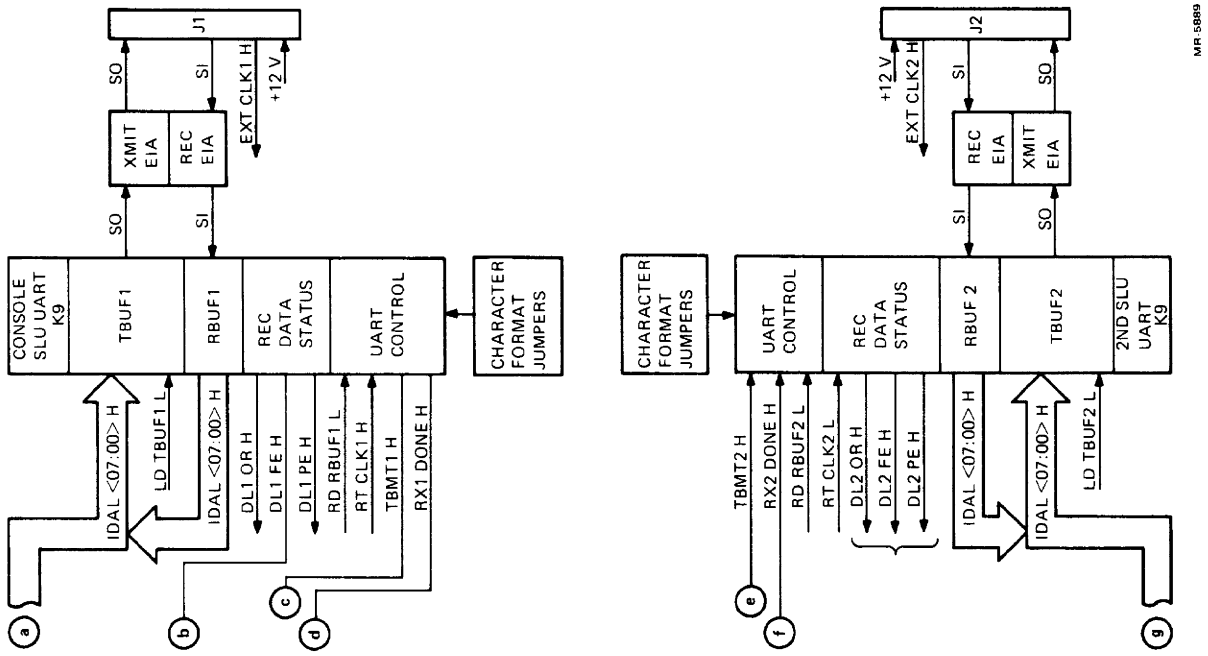


Figure 5-12 Serial-Line Units (Sheet 2 of 2)

5.14.2 The DC003 Interrupt Logic Circuits

Each serial-line unit has an associated DC003 interrupt logic circuit that consists of an 18-pin package. Each DC003 provides two interrupt channels for receiver and transmitter interrupts. The receiver channel has a higher priority than the transmitter channel.

The receiver channel contains a read/write interrupt enable bit [RX1 IE (1) H or RX2 IE (1) H] that is accessed as bit 06 of the receiver status register. LD RCSR1 H or LD RCSR2 H loads IDALB 06 H into this interrupt enable bit.

The transmitter channel contains a read/write interrupt enable bit [TX1 IE (1) H or TX2 IE (1) H] that is accessed as bit 06 of the transmitter status register. LD TCSR1 H or LD TCSR2 H loads K6 IDALB 06 H into the interrupt enable bit.

If the receiver interrupt enable bit and the receiver done bit (RX1 DONE H and RX2 DONE H) are both set, or if the transmitter interrupt enable bit and the transmitter ready bit (TBMT1 H and TBMT2 H) are both set, the open collector interrupt request output is asserted. The interrupt request outputs of the two DC003 circuits are tied together as DL IRQ L.

The F11 chip set and KDF11-BA logic respond to an SLU interrupt request (DL IRQ L asserted) by reading the vector address. PT2 (1) L drives the DC003 BDIN inputs and activates all channels requesting an interrupt at that time. RD VEC L not only gates the vector address onto IDAL 07–00 H, but also asserts the console DC003's BIAKI input. If neither of the interrupt channels in this DC003 were activated by PT2 (1) L, the DC003 asserts its BIAKO output, which drives the BIAKI input of the other DC003. The actual vector address gated onto IDAL 07–00 H depends upon DL1 VECB H, DL1 IAK0 H, DL2 VECB H, and DL2 ADRJ L. The selected interrupt channel is cleared.

5.14.3 Register Read Operations

Control signal RD RBUF1 L or RD RBUF2 L gates one of the receiver data buffers onto IDAL 07–00 H. Simultaneously, the assertions of ZB 15–12 L and ZB 11–08 L cause the zero bit assertion logic to negate IDAL 15–08 H.

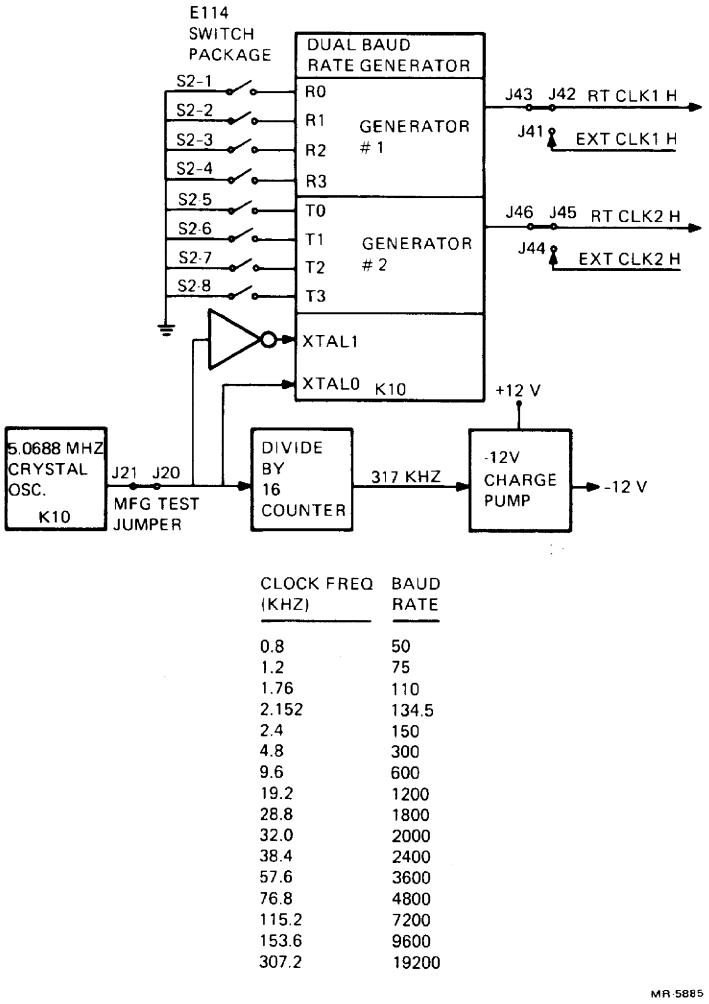
Control signal RD RCSR L gates either RX1 DONE H and RX1 IE (1) H or RX2 DONE H and RX2 IE (1) H onto IDAL 07–06 H and negates IDAL 02 H and IDAL 00 H. If LA 09 (1) H is asserted, the console receiver status register signals are selected. In either case, the assertion of ZB 15–12 L, ZB 11–08 L, and K7 ZB 5–3A1 L causes the zero bit assertion logic to negate the remaining IDAL lines.

Control signal RD TCSR L gates either TBMT1 H and TX1 IE (1) H or TBMT2 H and TX2 IE (1) H onto IDAL 07–06 H and negates IDAL 02 H and IDAL 00 H. If LA 09 (1) H is asserted, the console transmitter status register signals are selected. In either case, the assertion of ZB 15–12 L, ZB 11–08 L, and K7 ZB 5–3A1 L causes the zero bit assertion logic to negate the remaining IDAL lines.

5.14.4 Baud Rate Generator

A dual baud rate generator circuit contained in an 18-pin IC produces the RT CLK1 H and RT CLK2 H clocking signals for the two serial-line units if the J42–J43 and J45–J46 jumpers are installed.

The baud rate generator and −12 V Charge Pump are shown in Figure 5-13. The baud rate generator is driven by a signal provided by a 5.0688 MHz crystal oscillator. The baud rate generator divides the basic crystal oscillator frequency into one of 16 possible SLU receiver-transmitter frequencies. Four switches for each serial line select the desired RT clock frequency, which is 16 times the desired SLU baud rate. The switch configurations for selecting the available baud rates are listed in Table 2-10.



CLOCK FREQ (KHZ)	BAUD RATE
0.8	50
1.2	75
1.76	110
2.152	134.5
2.4	150
4.8	300
9.6	600
19.2	1200
28.8	1800
32.0	2000
38.4	2400
57.6	3600
76.8	4800
115.2	7200
153.6	9600
307.2	19200

Figure 5-13 Baud Rate Generator and −12 V Charge Pump

5.14.5 Charge Pump Circuit

The serial-line EIA transmitter-receiver drivers require +12 V and −12 V operating power. The charge pump circuit supplies the −12 V, thereby eliminating the need for backplane power other than the standard +5 V and +12 V.

The input to the charge pump circuit is a 317 KHz square wave signal obtained from a divide-by-16 counter driven by the 5.0688 MHz crystal oscillator. The 317 KHz signal drives a pair of MH0026 +12 V MOS clock drivers that alternately charge a 0.47 μF capacitor to −12 V.

CHAPTER 6
ADDRESSING MODES

6.1 INTRODUCTION

In the KDF11-BA 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-BA 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 (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-BA instruction set and instruction formats are explained in Chapter 7. To illustrate clearly 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 the source operand to the destination operand and store the result at the 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. The instruction format for the first word of the double-operand instruction is shown in Figure 6-2.

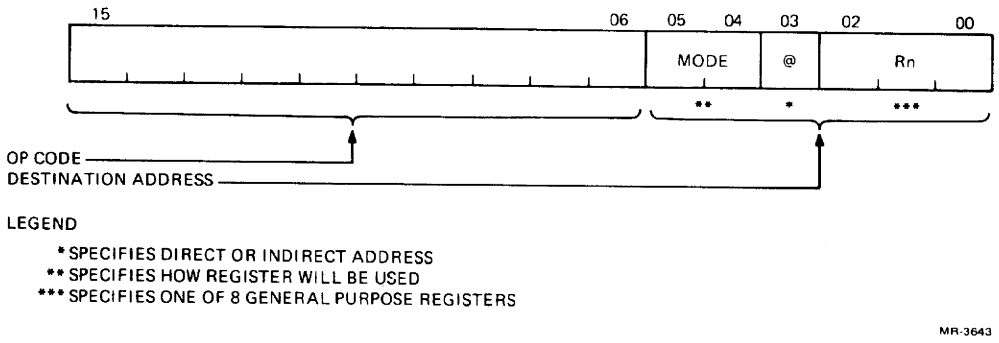


Figure 6-1 Single-Operand Instruction Format

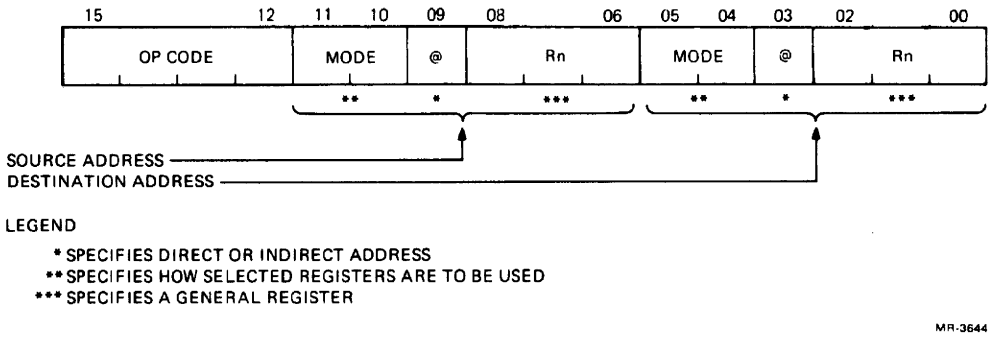


Figure 6-2 Double-Operand Instruction Format

6.3 ADDRESSING MODES

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

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-BA addressing modes are explained and shown in examples in the following pages. They are summarized in Paragraphs 6.3.10 through 6.3.13.

6.3.1 Register Mode (Mode 0) Rn

Register mode provides faster instruction execution since 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

The % sign indicates register definition.

Register Mode Examples (Figures 6-3 and 6-4.)

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

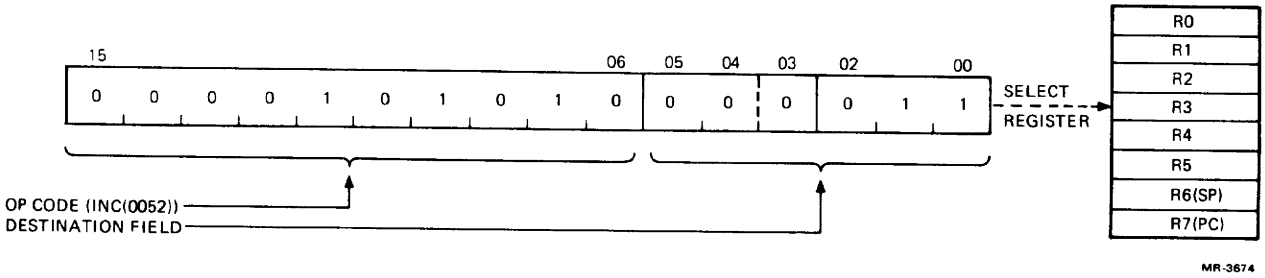


Figure 6-3 Register Mode Increment Example

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.

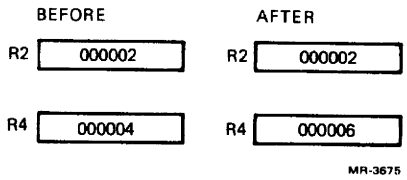


Figure 6-4 Register Mode Add Example

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 (Figure 6-5.)

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

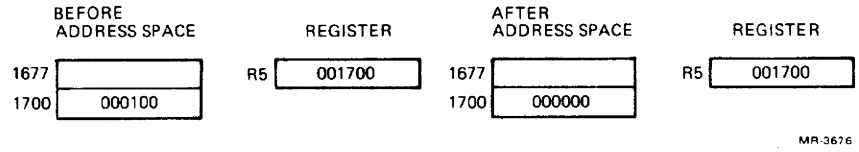


Figure 6-5 Register-Deferred Mode Example

6.3.3 Autoincrement Mode (Mode 2) (Rn)+

In autoincrement mode the register contains the address of the operand; 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, and by 2 if word instructions are being used.

Autoincrement Mode Example (Figure 6-6.)

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.

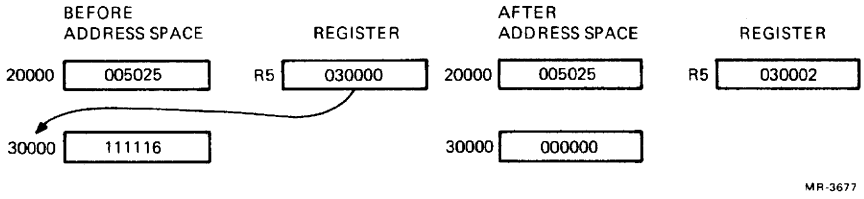


Figure 6-6 Autoincrement Mode Example

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; that is, 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 (Figure 6-7.)

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.

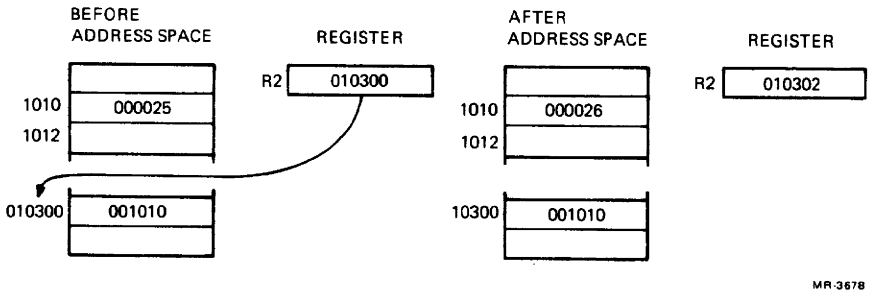


Figure 6-7 Autoincrement-Deferred Mode Example

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 (Figure 6-8.)

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.

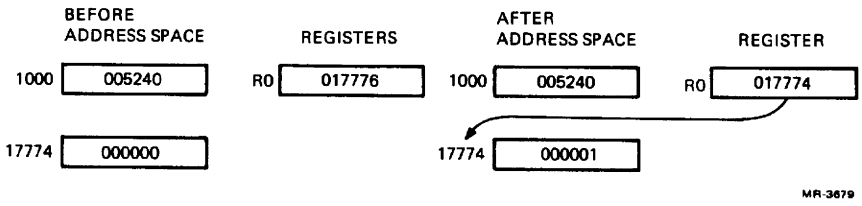


Figure 6-8 Autodecrement Mode Example

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 (Figure 6-9.)

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

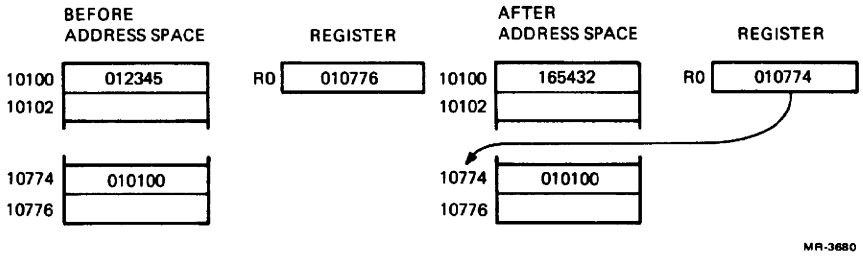


Figure 6-9 Autodecrement-Deferred Mode Example

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 a 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 (Figure 6-10.)

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.

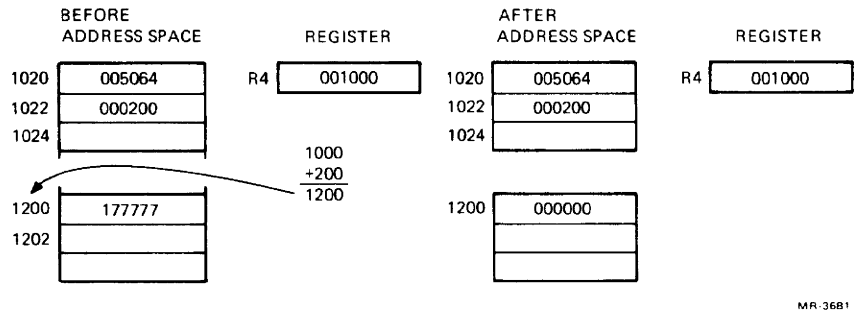


Figure 6-10 Index Mode Example

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 (Figure 6-11.)

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.

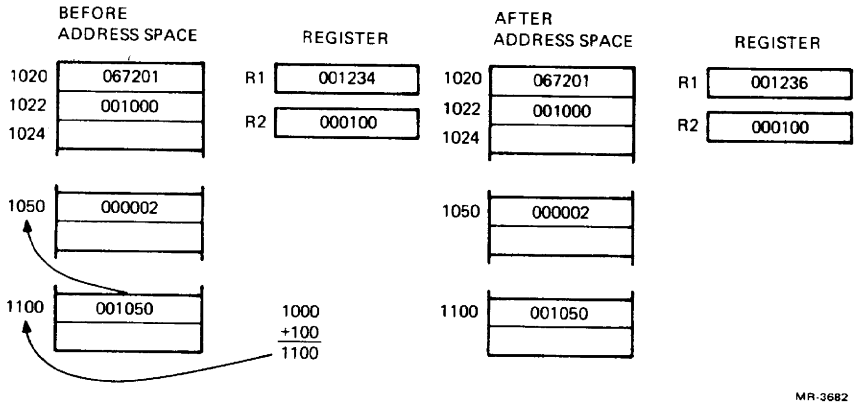


Figure 6-11 Index-Deferred Mode Example

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 (Figure 6-12.)

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.

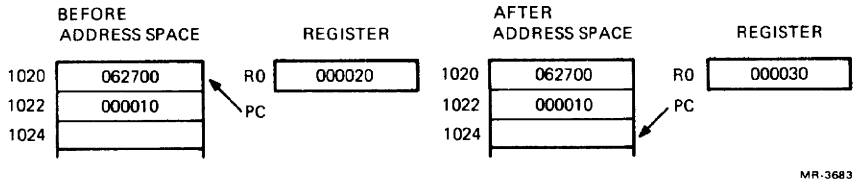


Figure 6-12 PC Immediate Mode Example

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 (Figure 6-13.)

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

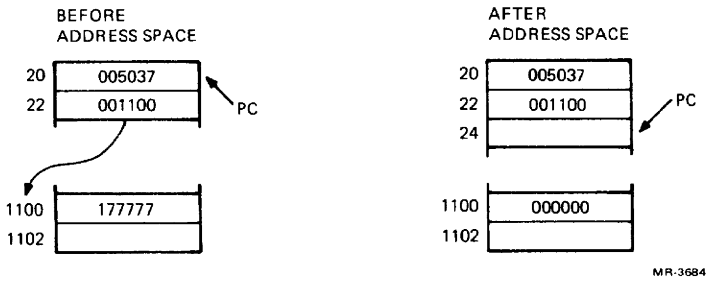


Figure 6-13 PC Absolute Mode Example

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. 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 (Figure 6-14.)

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

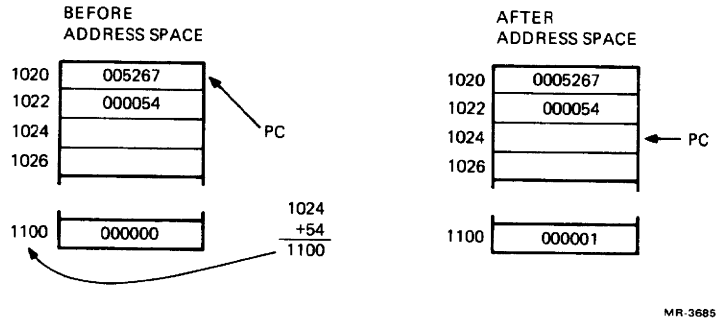


Figure 6-14 PC Relative Mode Example

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.

PC Relative-Deferred Mode Example (Figure 6-15.)

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

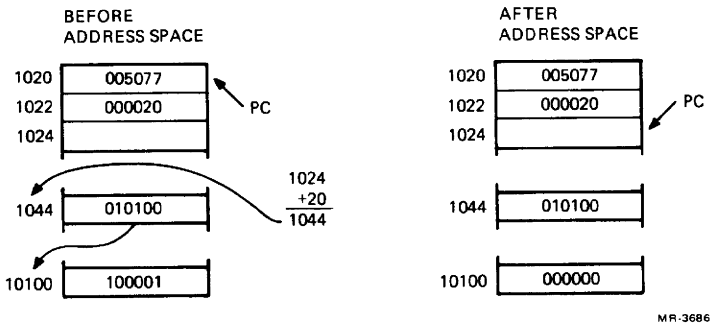


Figure 6-15 PC Relative-Deferred Mode Example

6.3.10 Direct Addressing Modes Summary

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

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 is incremented.
100	4	Autodecrement	–(Rn)	Register is decremented, then is 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.

6.3.11 Indirect Addressing Modes Summary

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

Table 6-2 Indirect Addressing Modes

Binary 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 is incremented (always by 2, even for byte instructions).
101	5	Autodecrement-deferred	@–(Rn)	Register is decremented (always by 2, even for byte instructions), then is 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.

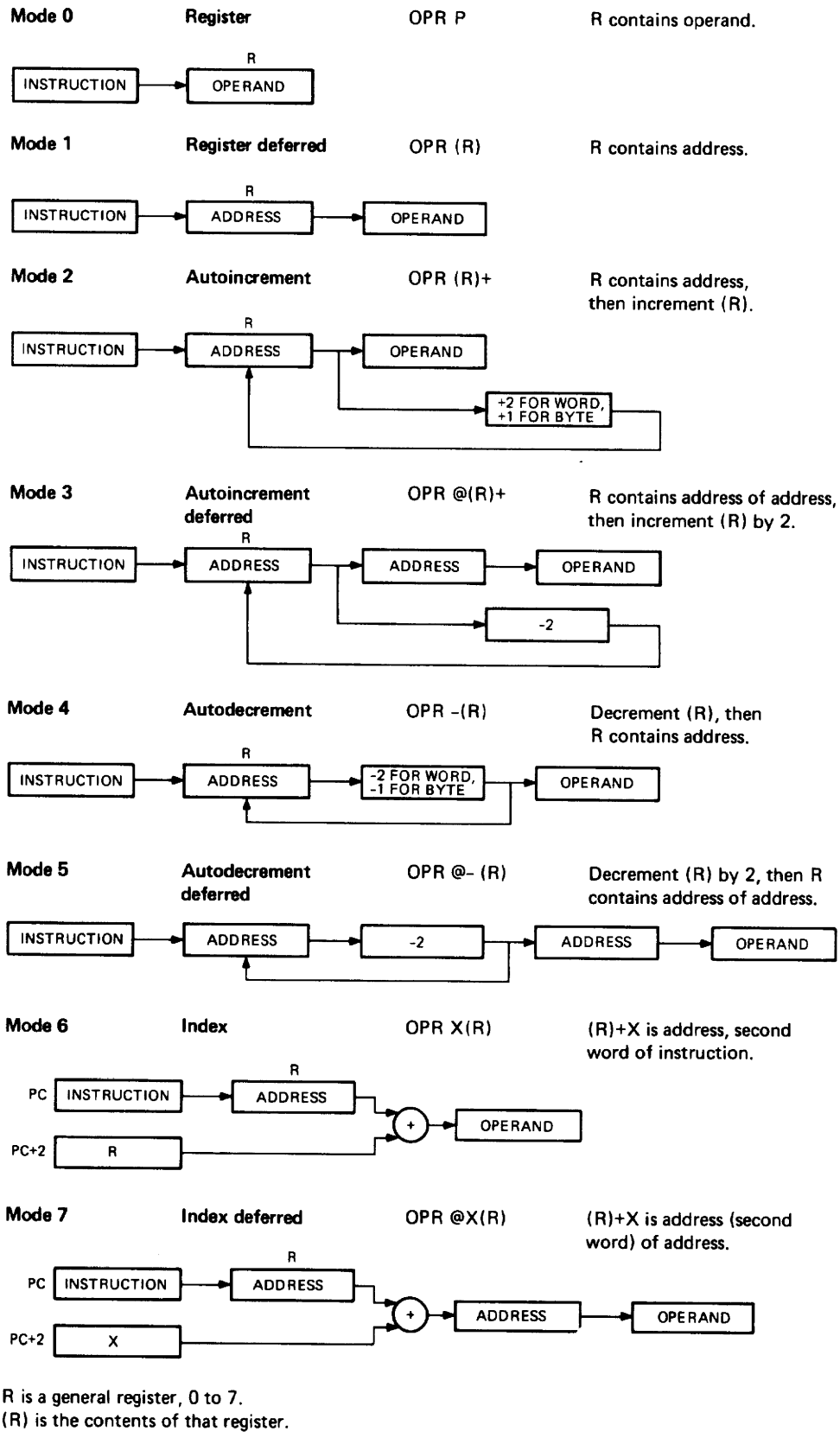


Figure 6-16 General Register Addressing Modes

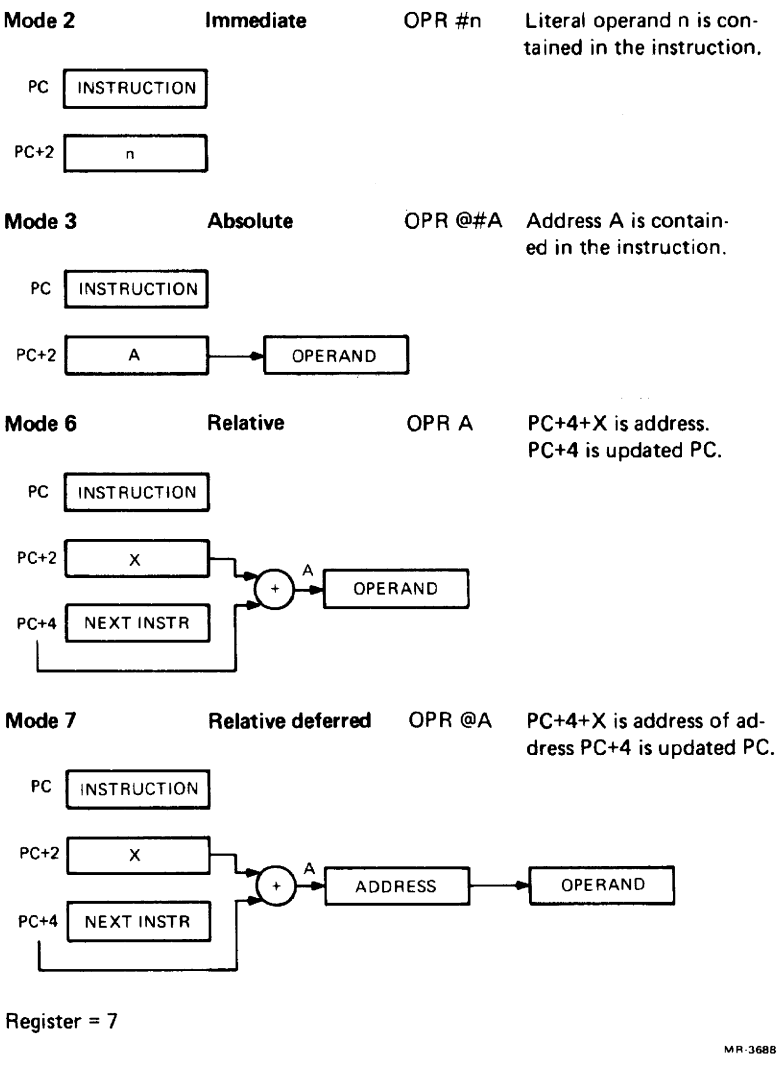


Figure 6-17 Program Counter Addressing Modes

CHAPTER 7
INSTRUCTION SET

7.1 INTRODUCTION

The KDF11-BA instruction set and addressing modes produce over 400 unique instructions. The instruction set offers a wide choice of operations, and often a single instruction will accomplish a task that would require several instructions in a traditional computer.

KDF11-BA 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 these operations with three instructions:

```
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, referred to as "op code," specifies the operation; 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 gives information on locating the operand.

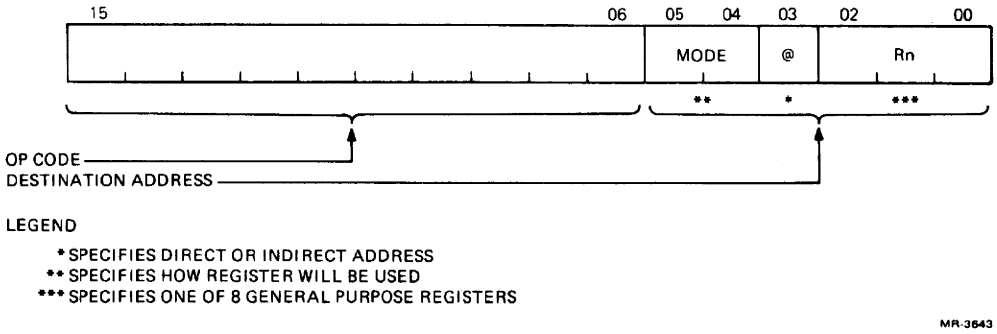


Figure 7-1 Single-Operand Instruction Format

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 the former have *two* 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 also completely independent. The mode and register used by one field may be completely different from the mode and register used by another field.

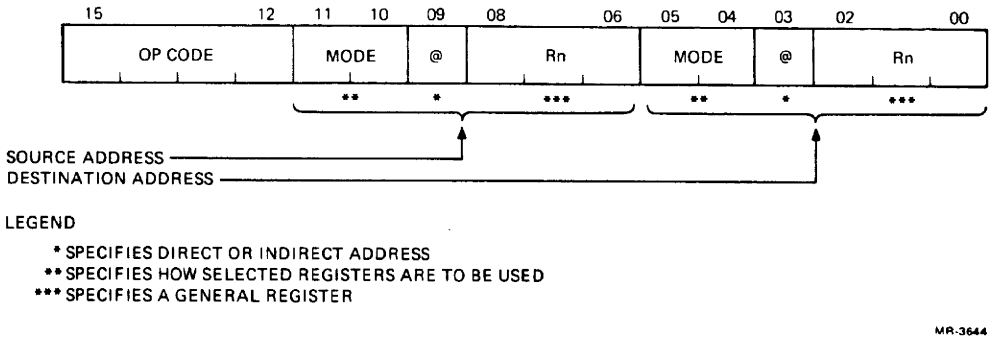


Figure 7-2 Double-Operand Instruction Format

Bit 15 indicates word or byte operation *except* 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 – that is, 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.

```
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 – What follows 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 1 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
The high byte (bits 8–15) of the instruction is an op code specifying the conditions for the branch to take place. Refer to Figure 7-3.

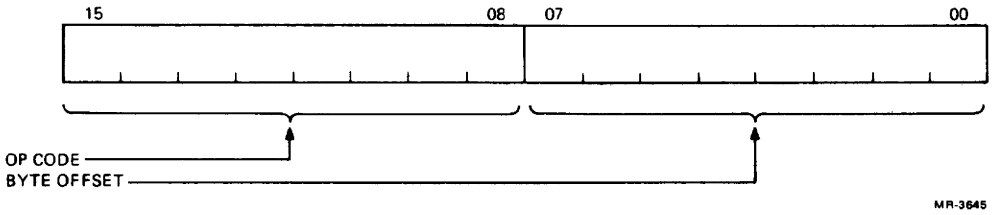


Figure 7-3 Branch Instruction Format

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
Bits 9–15 are always octal 004 indicating the op code for JSR. Refer to Figure 7-4.

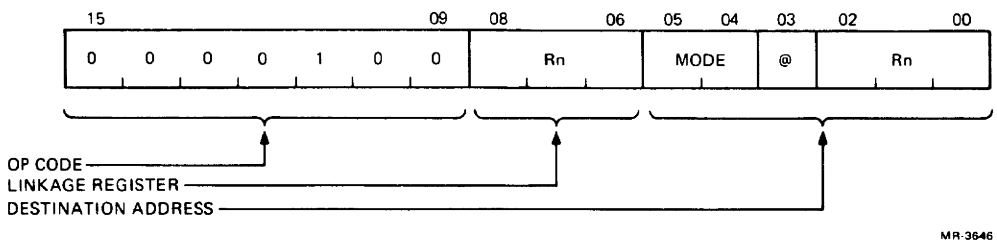


Figure 7-4 JSR Instruction Format

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 subroutine.

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

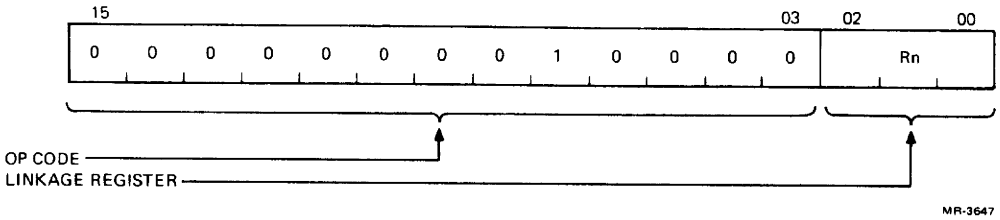


Figure 7-5 RTS Instruction Format

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. The register specified by bits 0–2 must be the same register as the one used in the JSR that 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 to leave 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) that 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 formats. How the condition codes are affected is also discussed.

Mnemonic	Instruction
CLC, CLV, CL2 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" that 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, the corresponding condition code bit is set or cleared depending on the state of the "operator" (bit 4).

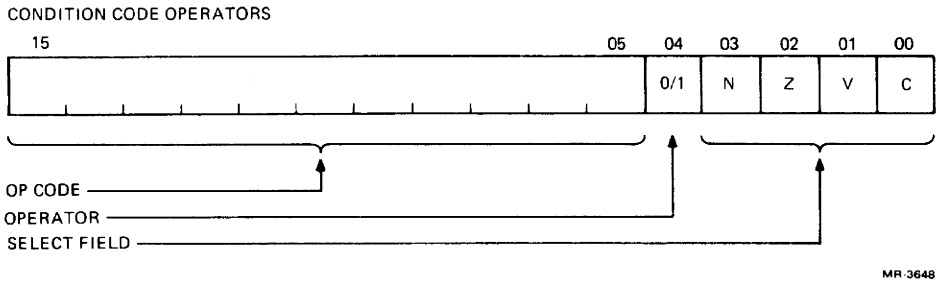


Figure 7-6 Condition Code Operators Format

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 a 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, 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 uses 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 exists.
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. Otherwise, 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 – Miscellaneous program control instructions are listed below.

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 of Single-Operand, Double-Operand, and Branch Instructions

The following examples and explanations show 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 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, 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 incremented to point to the next sequential location, and the counter (R1) is 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, 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 to be from $+177_8$ to -200_8 words.

A payroll program has set up a series of words to identify each employee by his/her 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 that represents his/her salary. These numbers represent steps within three wage classes to identify which employees are 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).

The starting address of the memory block containing the employee payroll information is 600. The final address of this data area is 1264. The following program searches through the data area and finds all numbers representing wage class I. Each time one is found, the program stores the employee's badge number (just the high byte) on a "last-in/first-out" stack that 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, 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 (1264) has not been examined, 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-BA instruction set is described below. 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

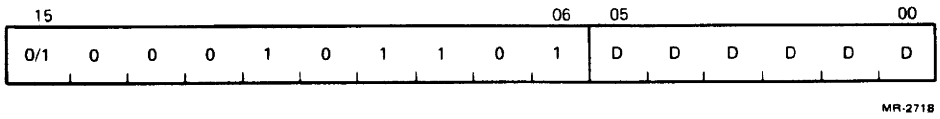
Symbol	Meaning
SO	Single-operand instruction.
DO	Double-operand instruction.
PC	Program control instruction.
MS	Miscellaneous instruction.
CC	Condition code.
()	Indicates "the contents of"; for example, (R5) means "the contents of R5."
src	Source address.
dst	Destination address.
←	Becomes, or moves 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.
∨	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

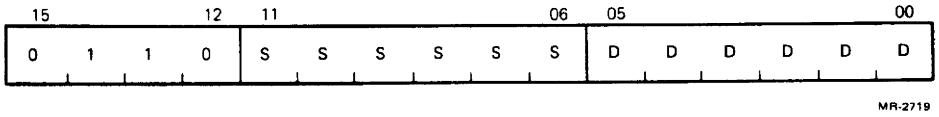
Operation: (dst) ← (dst) + C

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 to 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



Add

Type: DO

Operation: (dst) ← (src) + (dst)

Condition Codes:
N: set if result < 0

Z: set if result = 0

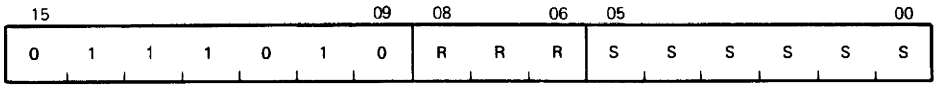
V: set if there is arithmetic overflow as a result of the operation; that is, both operands were of the same sign and the result is of the opposite sign

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

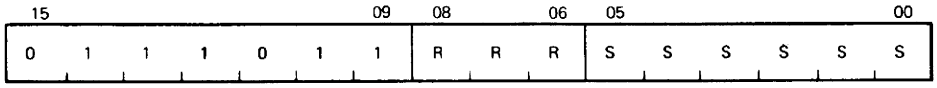
Operation: R ← R shifted arithmetically NN places to the right or left where NN = (src)

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 six bits of the source operand. This number ranges from −32 to +31. Negative is a right shift and positive is a left shift.

ASHC

Arithmetic shift combined 073RSS



Type: DO

Operation: R, R ∨ 1 ← R, R ∨ 1
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 shift; 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 ORed with 1 are treated as one 32-bit word. R ∨ 1 (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

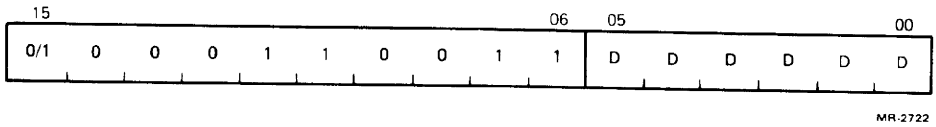
six bits of the source operand. This number ranges from −32 to +31. Negative is a right shift and positive is a left shift.

When the register chosen is an odd number, the register and the register ORed 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



MR-2722

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)

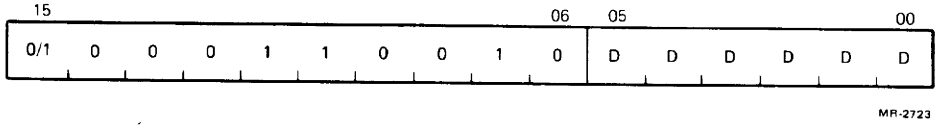
C: loaded with the high-order bit of the destination

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.

ASR/ASRB

Arithmetic shift right

0062DD
1062DD



MR-2723

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

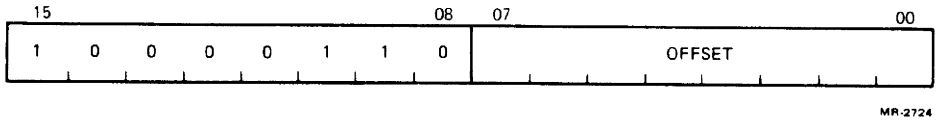
V: loaded from the exclusive OR of the N bit and C bit (as set by the completion of the shift operation)

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

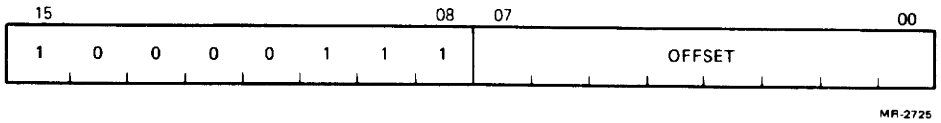
Operation: PC ← PC + (2 × offset) if C = 0

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.

BCS

Branch if carry set 103400



Type: PC

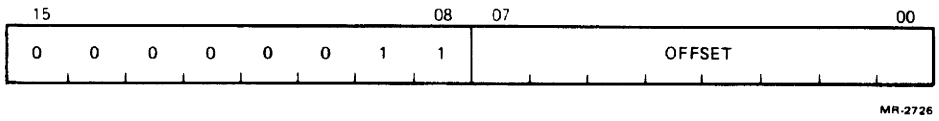
Operation: PC ← PC + (2 × offset) if C = 1

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

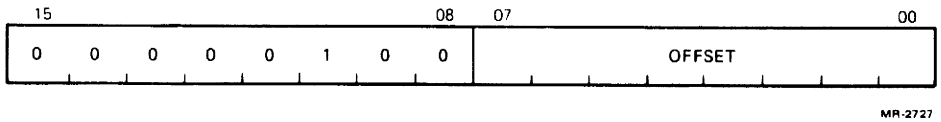
Operation: PC ← PC + (2 × offset) if Z = 1

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.

BGE

Branch if greater than or equal 002000



Type: PC

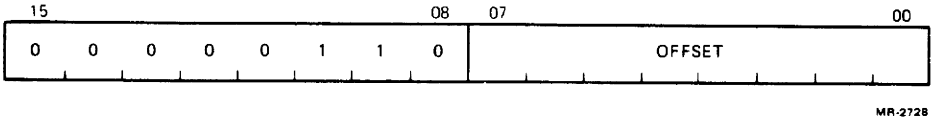
Operation: PC ← PC + (2 × offset) if N ∀ V = 0

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

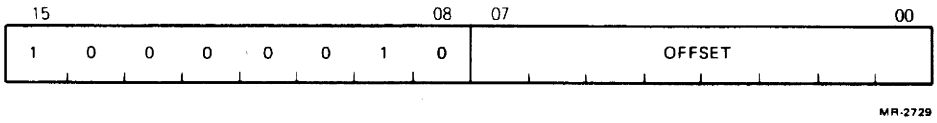
Operation: PC ← PC + (2 × offset) if Z ∨ (N ∀ V) = 0

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 after 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).

BHI

Branch if higher 101000



Type: PC

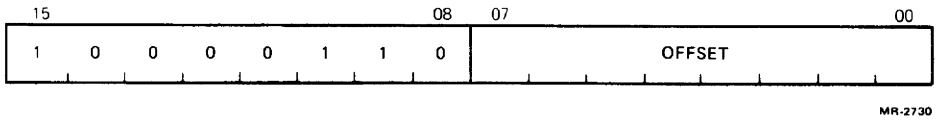
Operation: PC ← PC + (2 × offset) if C = 0 and Z = 0

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

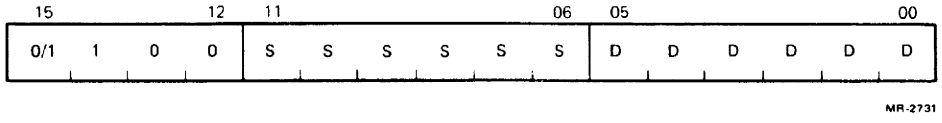
Operation: PC ← PC + (2 × offset) if C = 0

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.

BIC/BICB

Bit clear 04SSDD
14SSDD



Type: DO

Operation: (dst) ← ~ (src) ∧ (dst)

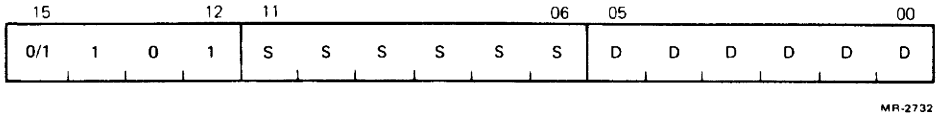
Condition Codes:
N: set if high-order bit of result set
Z: set if result = 0
V: cleared
C: not cleared

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 remain unaffected.

BIS/BISB

Bit set

05SSDD
15SSDD



Type: DO

Operation: (dst) ← (src) ∨ (dst)

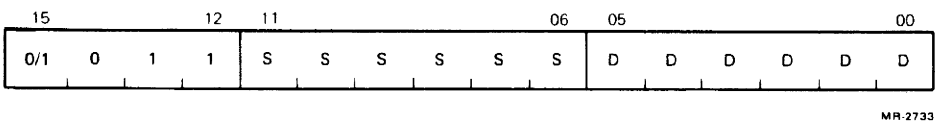
Condition Codes:
N: set if high order bit of result set
Z: set if result = 0
V: cleared
C: not affected

Description: Performs an 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/BITB

Bit test

03SSDD
13SSDD



Type: DO

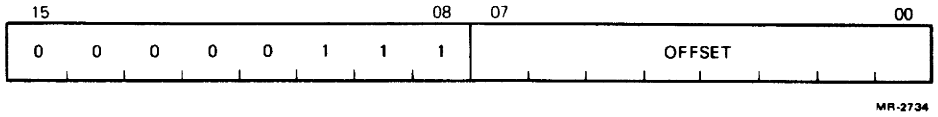
Operation: (dst) ∨ (src)

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

Description: Performs a logical AND comparison of the source and 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

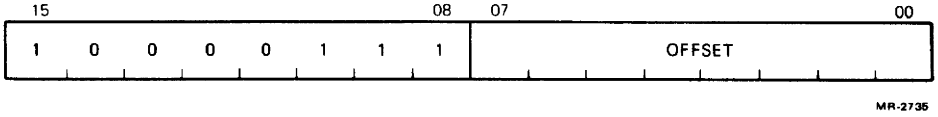
Operation: PC ← PC + (2 × offset) if Z ∨ (N ∀ V) = 1

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 after 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).

BLO

Branch if lower 103400



Type: PC

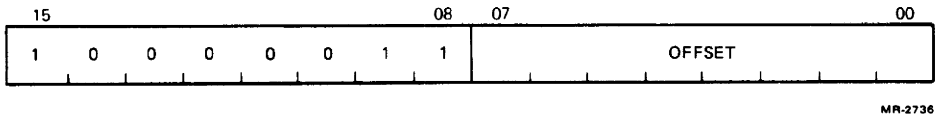
Operation: PC ← PC + (2 × offset) if C = 1

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

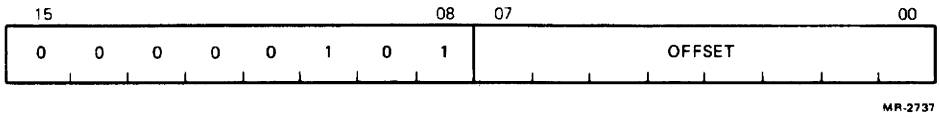
Operation: PC ← PC + (2 × offset) if C ∨ Z = 1

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.

BLT

Branch if less than 002400



Type: PC

Operation: PC ← PC + (2 × offset) if N ∀ V = 1

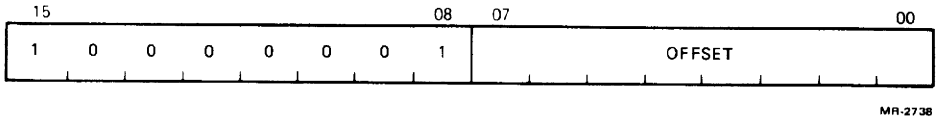
Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes a branch if the exclusive OR of the N and V bits is 1. Thus, BLT always branches after 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 × offset) if N = 1

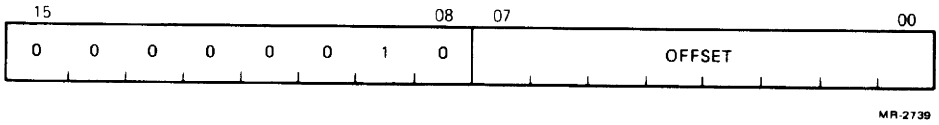
Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

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

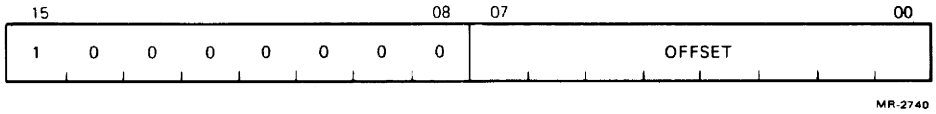
Operation: PC ← PC + (2 × offset) if Z = 0

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

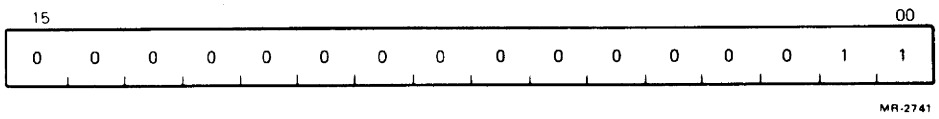
Operation: PC ← PC + (2 × offset) if N = 0

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.

BPT

Breakpoint trap 000003



Type: PC

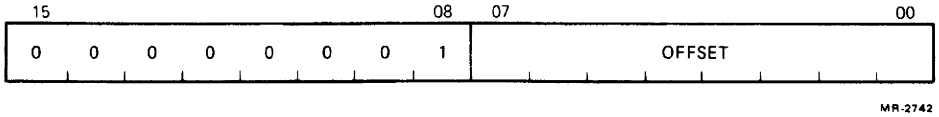
Operation:
– (SP) ← PS
– (SP) ← PC
PC ← (14)
PS ← (16)

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

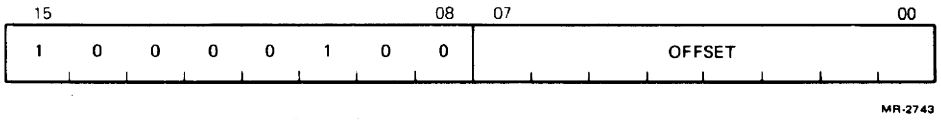
Operation: PC ← PC + (2 × offset)

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 unconditional branch.

BVC

Branch if V bit clear 102000



Type: PC

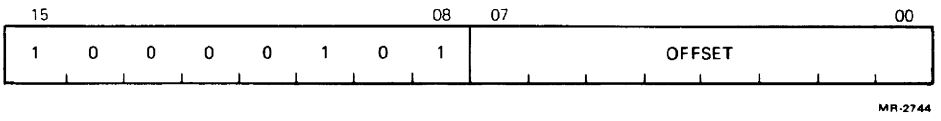
Operation: PC ← PC + (2 × offset) if V = 0

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

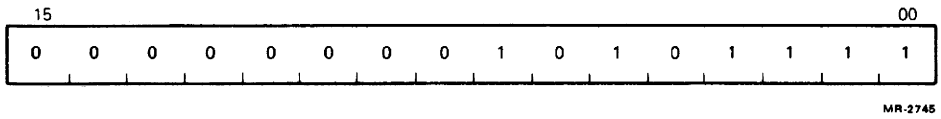
Operation: PC ← PC + (2 × offset) if V = 1

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

Description: Tests the state of the 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

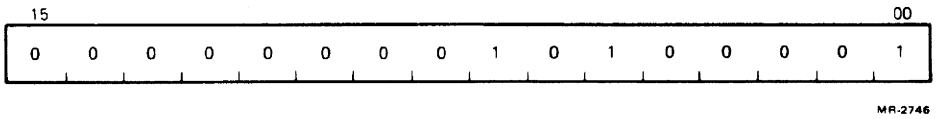


Type: CC

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

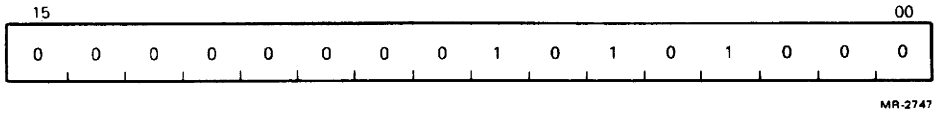


Type: CC

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, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.

CLN

Clear N 000250



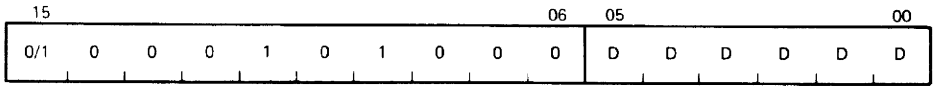
Type: CC

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.

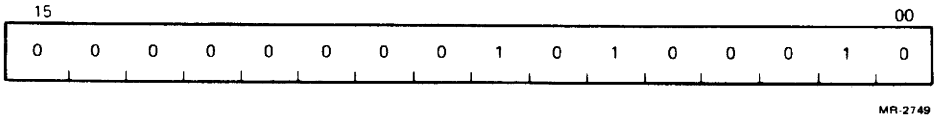
NOTE

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.

CLV

Clear V

000242



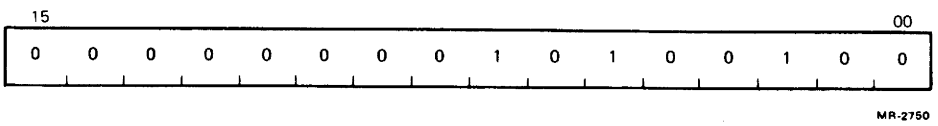
Type: CC

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 bits if bit 4 = 0.

CLZ

Clear Z

000244



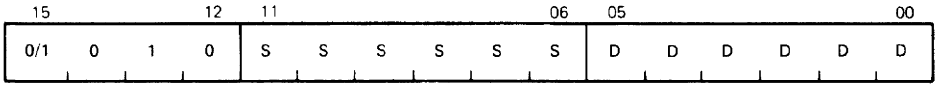
Type: CC

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.

CMP/CMPB

Compare

02SSDD
12SSDD



MR-2751

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., if the operands were of opposite signs and the sign of the destination is the same as the sign of the result

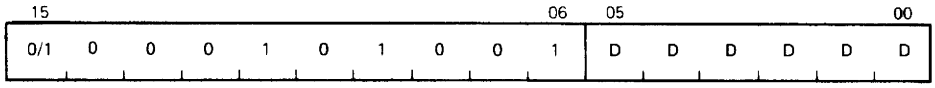
C: cleared if there is a carry from the most significant bit of the result

Description: 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



MR-2752

Type: SO

Operation: (dst) ← ~ (dst)

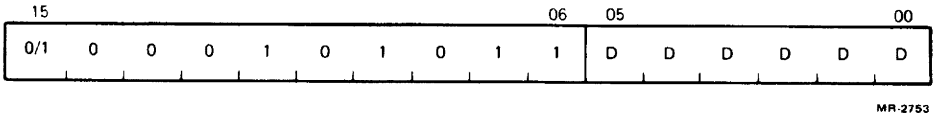
Condition Codes: N: set if most significant bit of result = 0
Z: set if result = 0
V: cleared
C: set

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).

DEC/DECB

Decrement

0053DD
1053DD



Type: SO

Operation: (dst) ← (dst) − 1

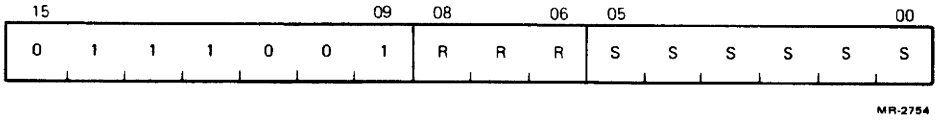
Condition Codes: N: set if result < 0
Z: set if result = 0
V: set if (dst) was 100000
C: not affected

Description: Subtract 1 from the contents of the destination.

DIV

Divide

071RSS



Type: DO

Operation: R, R ∨ 1 ← R, R ∨ 1/(src)

Condition Codes: N: set if quotient < 0

Z: set if quotient = 0

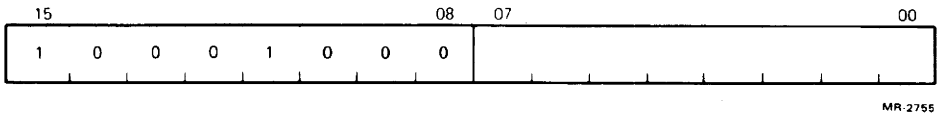
V: set if source = 0 or if the absolute value of the register is larger than the absolute value of the instruction in the source. (In this case the instruction is aborted because the quotient would exceed 15 bits.)

C: set if divide by 0 attempted

Description: The 32-bit 2's complement integer in R and R ∨ 1 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.

EMT

Emulator trap 104000



Type: PC

Operation:
— (SP) ← PS
— (SP) ← PC
PC ← (30)
PS ← (32)

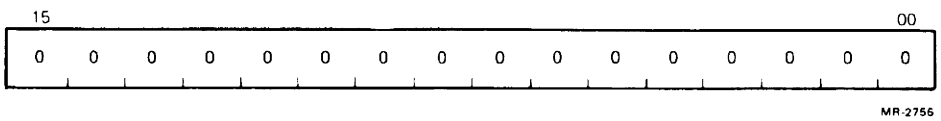
Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

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

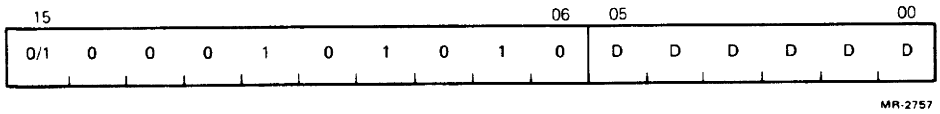
Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

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).

INC/INCB

Increment

0052DD
1052DD



Type: SO

Operation: (dst) ← (dst) + 1

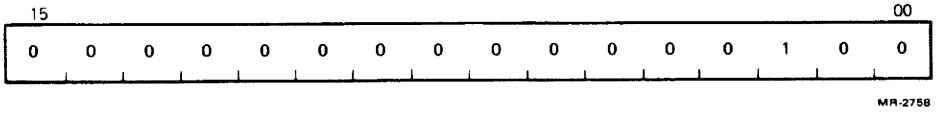
Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if dst was 077777
C: not affected

Description: Adds 1 to the contents of the destination.

IOT

I/O trap

000004



Type: PC

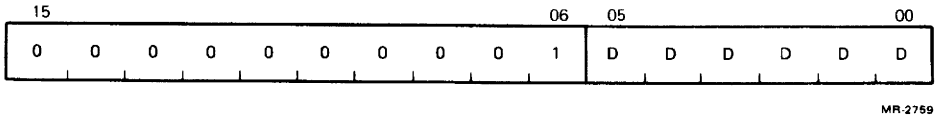
Operation:
– (SP) ← PS
– (SP) ← PC
PC ← (20)
PS ← (22)

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.

JMP

Jump 0001DD



Type: PC

Operation: PC ← (dst)

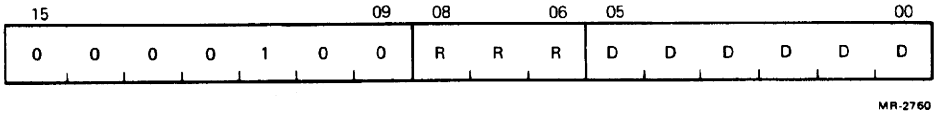
Condition Codes:
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: JMP provides more flexible program branching than provided with the branch instruction. JMP is not limited to $+177_8$ and -200_8 words as are branch instructions. JMP does generate a second word, however, 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 reentrant manner on the processor stack, execution of a subroutine may be interrupted, and the same subroutine reentered 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.

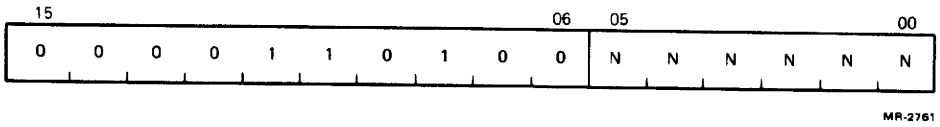
JSR PC, dst is a special case of the subroutine call suitable for subroutine calls that transmit parameters. JSR PC saves the use of an extra register.

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

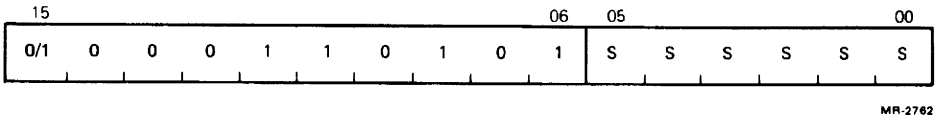
Operation:
SP ← PC + 2 × NN
PC ← R5
R5 ← (SP) +
nn = number of parameters

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

MFPD/MFPI

Move from previous data space 0065SS
Move from previous instruction space 1065SS



Type: MS

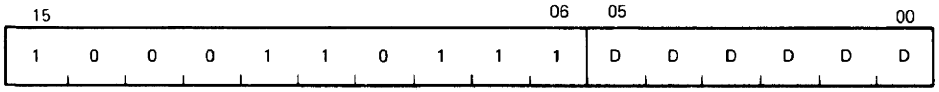
Operation:
(tmp) ← (src)
– (SP) ← (temp)

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 the 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 in the same way as an MFPI does.

MFPS

1067DD



Type: MS

Operation: (dst) ← PS
dst lower 8 bits

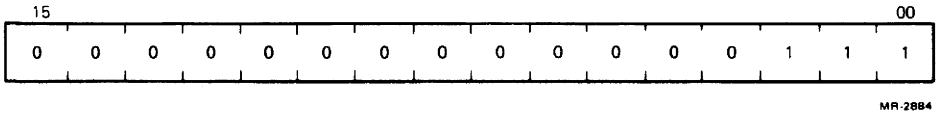
Condition Codes: N: set if PS bit 7 = 1
Z: set if PS <0:7> = 0
V: cleared
C: unaffected

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 the upper byte of the register. The destination operand is treated as a byte address.

The KDF11-BA implements the PS address 17777776, 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.

MFPT

Move from processor type 000007



Type: MS

Operation: R0 ← 000003

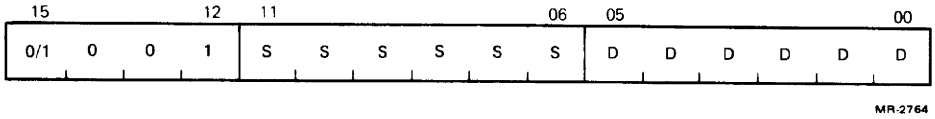
Condition Codes: N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: A unique number assigned to each PDP-11 processor model is loaded into general register R0. The KDF11-BA 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 op code 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), which is 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.

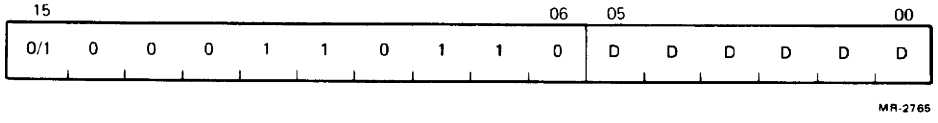
NOTE

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.

MTPD/MTPI

Move to previous data space
Move to previous instruction space

1066SS
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 and 14 and stores that word into an address in the previous space (PS bits 13 and 12). The destination address is computed using the current registers and memory map.

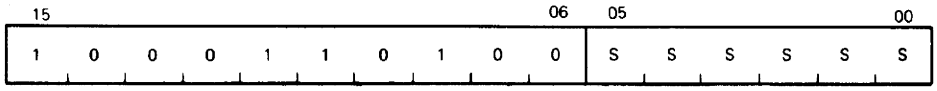
Since data space does not exist in the KDF11, MTPD executes in the same way as MTPI does.

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)

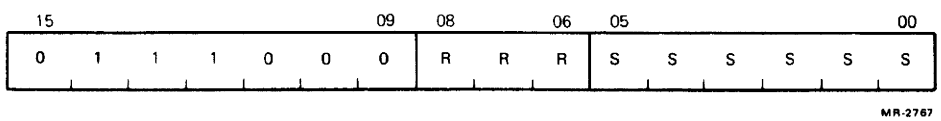
Condition Codes: N: set according to effective src operand bits 0–3
Z: same
V: same
C: same

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-BA implements the PS address 17777776, 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.

MUL

Multiply 070RSS



Type: DO

Operation: R, R ∨ 1 ← R × (src)

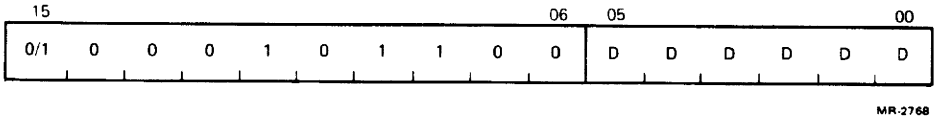
Condition Codes:
N: set if product < 0
Z: set if product = 0
V: cleared
C: set if the result is less than -2^{15} or greater than or equal to $2^{15} - 1$.

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, R ∨ 1, which reduces to just R when R is odd.)

NEG/NEGB

Negate

0054DD
1054DD



Type: SO

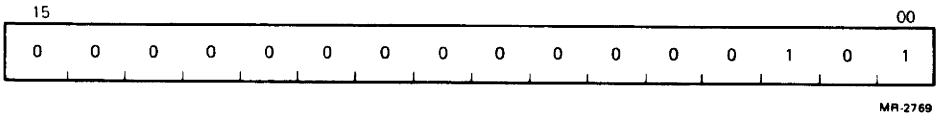
Operation: (dst) ← (dst)

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)

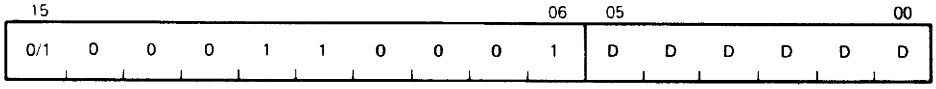
Condition Codes: N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: Causes bus signal BINIT L to be asserted for 10 μs and then unasserted for 90 μs. 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



MR-2770

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)

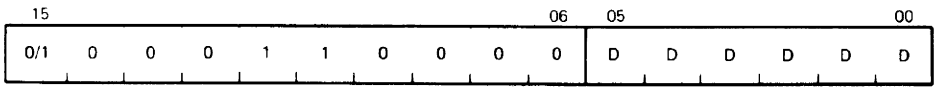
C: loaded with the high-order bit of the destination

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.

ROR/RORB

Rotate right

0060DD
1060DD



MR-2771

Type: SO

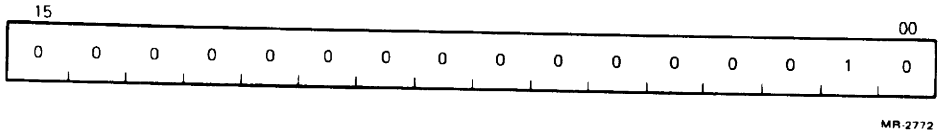
Operation: (dst) ← (dst)
rotate right one place

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) +

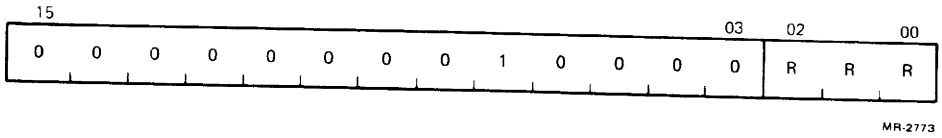
Condition Codes:
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

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.

RTS

Return from subroutine

00020R



Type: PC

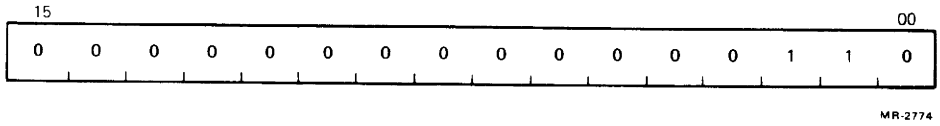
Operation:
PC ← (reg)
(reg) ← SP +

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

Description: Loads the contents of the register into the PC and pops the top element of the processor stack into the specified register. Return from a nonreentrant 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) +

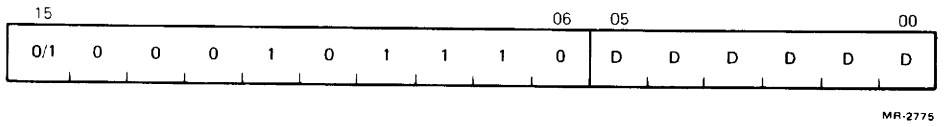
Condition Codes: N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: Used to exit from a trace trap (T bit) service routine and executes in the same way as the RTT instruction does, 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.

SBC/SBCB

Subtract carry

0056DD
1056DD



Type: SO

Operation: (dst) ← (dst) − C

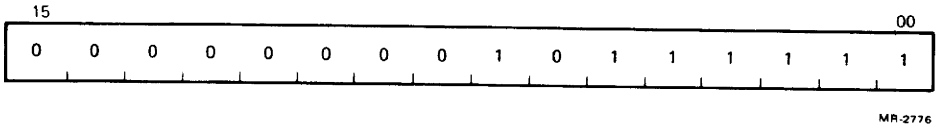
Condition Codes:
N: set if result < 0
Z: set if result = 0
V: set if (dst) = 100000 and C = 1
C: cleared if (dst) = 0 and C = 1

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



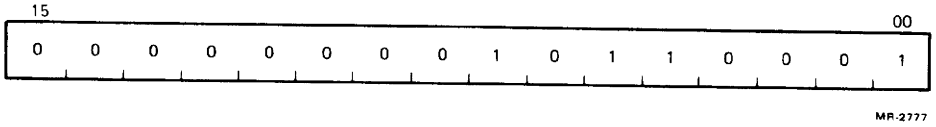
Type: CC

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.

SEC

Set C

000261



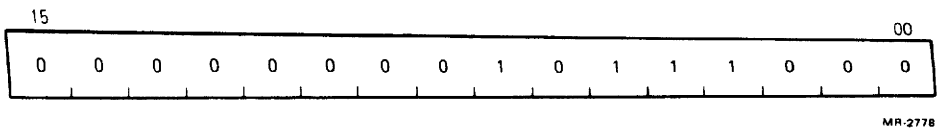
Type: CC

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

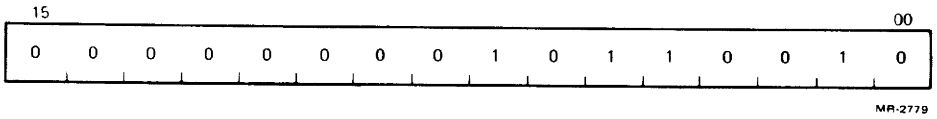


Type: CC

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.

SEV

Set V 000262

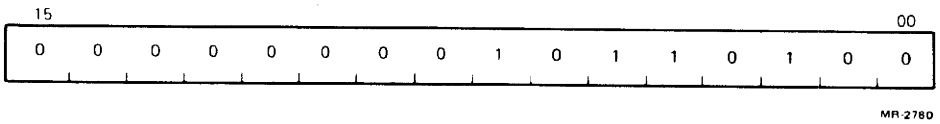


Type: CC

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

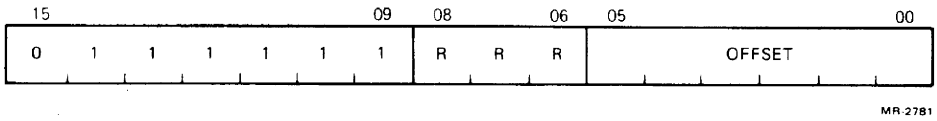


Type: CC

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, or 3, if 4 is a 1. Clears corresponding bits if bit 4 = 0.

SOB

Subtract one and branch if not equal to 0 — 077R00 + offset



Type: PC

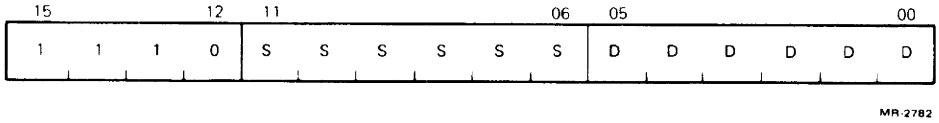
Operation: R ← R − 1; if this result does not = 0 then PC ← PC − (2 × offset)

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

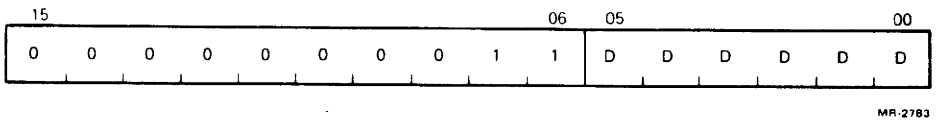
V: set if there is arithmetic overflow as a result of the operation, i.e., if the operands were of opposite signs and the sign of the source is the same as the sign of the result

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 indicates a borrow when set.

SWAB

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

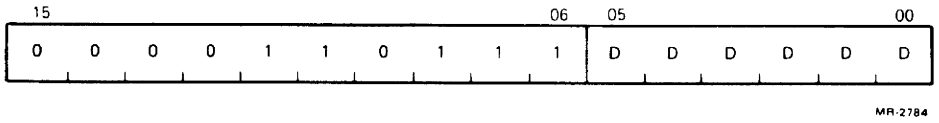
V: cleared

C: cleared

Description: Exchanges the 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

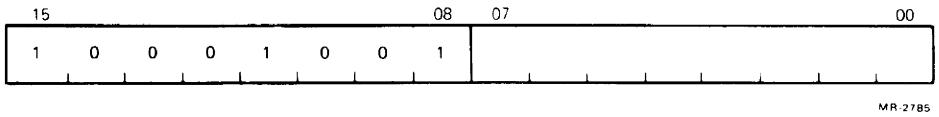
Description: If the condition code bit N is set, a −1 is placed in the destination operand; if N bit is clear, a 0 is placed in the destination operand. This instruction is particularly useful in multiple-precision arithmetic because it permits the sign to be extended through multiple words.

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.

TRAP

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 vector for TRAP is at address 34.

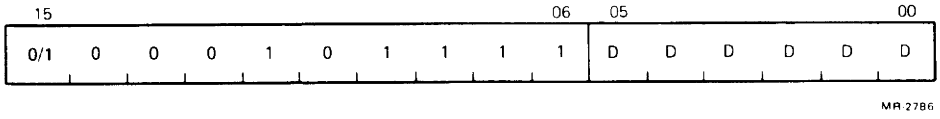
NOTE

Since DIGITAL software makes frequent use of EMT, the TRAP instruction is recommended for general use.

TST/TSTB

Test

0057DD
1057DD



Type: SO

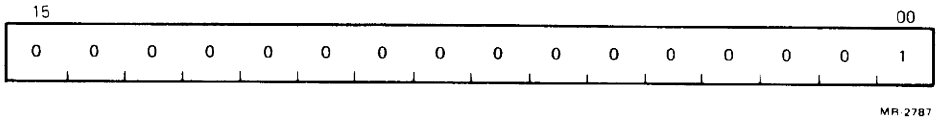
Operation: (dst) ← (dst)

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.

WAIT

000001



Type: MS

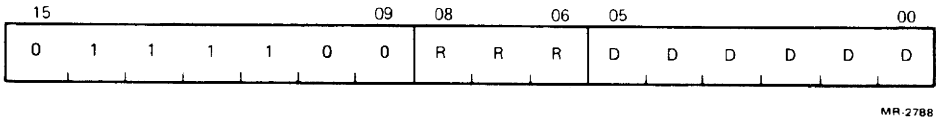
Operation:

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 interrupted process at the instruction following the WAIT.

XOR

074RDD



Type: DO

Operation: (dst) ← R ∨ (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. The contents of the register are unaffected. Assembler format is: XOR R, D.

CHAPTER 8
MEMORY MANAGEMENT

8.1 INTRODUCTION

The KDF11-BA processor implements a 2 megaword physical address space. The mapping or translation of 16-bit virtual addresses to 18- or 22-bit physical addresses is implemented in one 40-pin 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, PDP-11/60 and PDP-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 a 18- or 22-bit physical address. The actual physical address transformation occurs transparently to an executing program. The MMU chip also contains some of the floating-point registers in addition to the relocation registers.

8.1.1 Programming

The memory management hardware has been designed for a multiuser operating system environment. The processor can operate in two modes (kernel and user) to provide memory relocation and protection in a multiuser environment. When in kernel mode, software has complete control and can execute all instructions. Monitors and supervisory programs are executed in this mode.

In a multiuser environment several user programs reside 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. This restricted environment prevents the following.

1. Modification of the kernel program.
2. Halting the computer.
3. Initializing the system.
4. Using memory space assigned to the kernel or to other users.

In a multiuser 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 pages outside his/her assigned area. Thus, a user can effectively be prevented from accidental or willful destruction of any other user's 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; however, the extended LSI-11 bus and the KDF11-BA addressing logic are 22 bits wide. While a 16-bit word can generate up to 32K words (64K bytes) of virtual address references, the CPU and extended LSI-11 bus can reference up to 2 megawords (4 megabytes) of physical 22-bit addresses. The extra six bits of addressing logic provide the basic framework for expanding memory references.

The uppermost 4K words of address space is reserved for I/O device registers. The 2 megawords of physical address space that can be referenced with memory management consist of 2,093,056 words of user memory and 4,096 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 (APRs) (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 (CM) contained in the processor status word, bits 15 and 14.

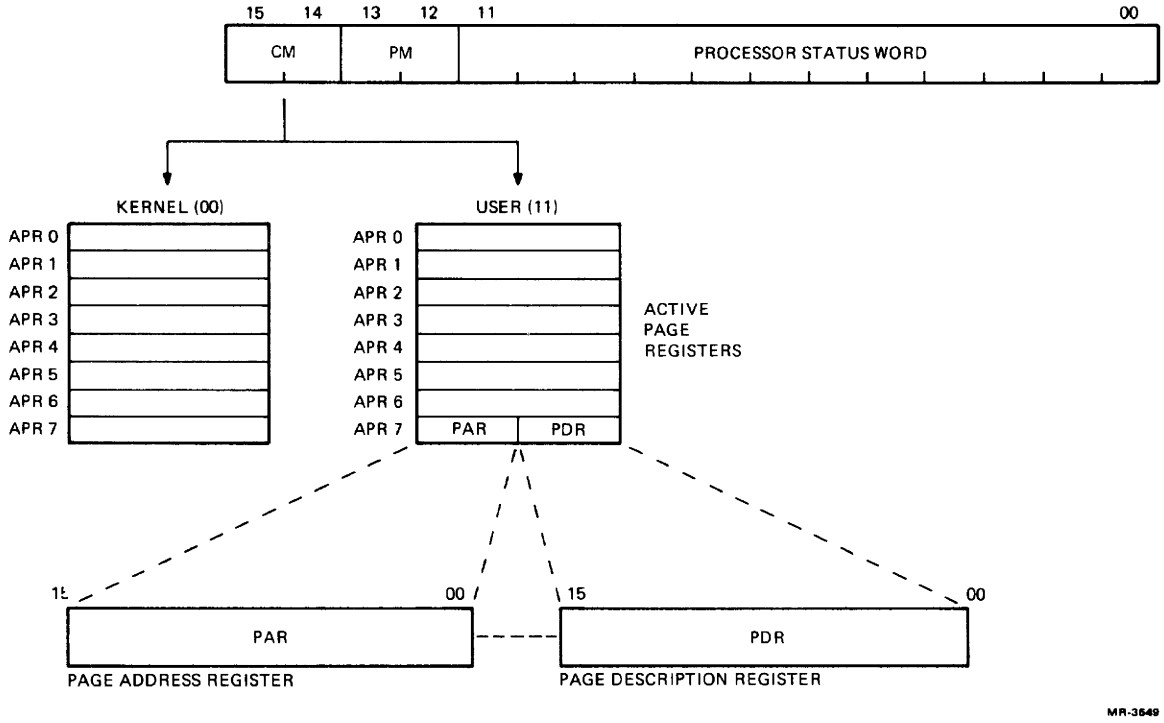


Figure 8-1 Active Page Registers

8.1.4 Capabilities Provided by Memory Management

Memory Size (words)	2 megawords (minus 4K words for I/O Page)
Address Space	Virtual (16 bits) Physical (18 or 22 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

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- or 22-bit physical address. Information contained in the virtual address is combined with relocation and description information contained in the active page register to yield an 18- or 22-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 of the memory management unit 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. 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 (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.

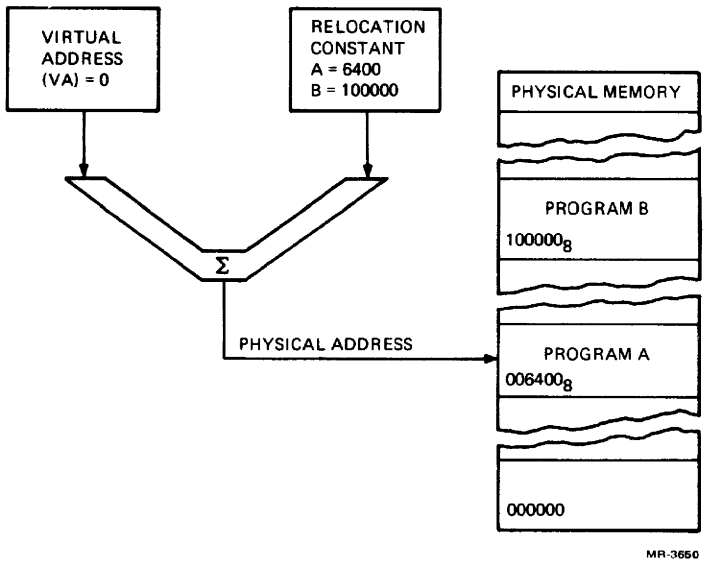


Figure 8-2 Memory Relocation, Simplified Block Diagram

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 × 32) words. Using all 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 physical memory, as long as each relocated page begins on a boundary that is a multiple of 32 words. However, for pages smaller than 4K words, only the memory actually allocated to the page may be accessed. Refer to the relocation example shown in Figure 8-3.

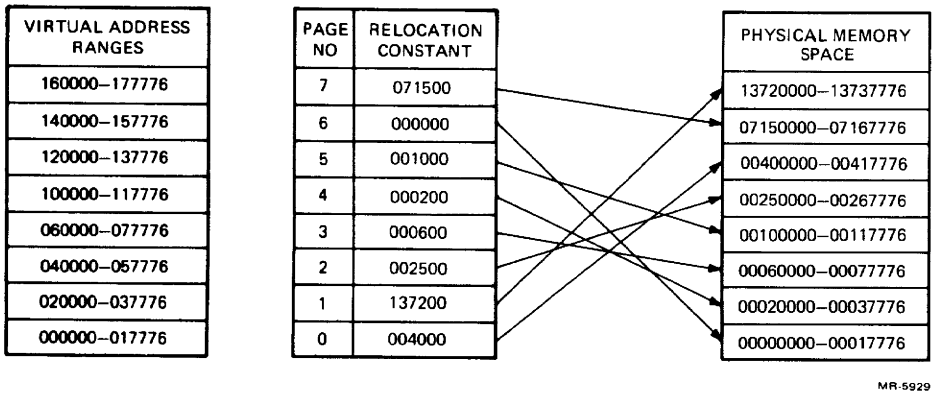


Figure 8-3 Relocation of a 32K-Word Program into 2 Megawords of Physical Memory

Figure 8-3 illustrates several points about memory relocation.

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 physical addresses higher or lower in respect to their virtual address ranges. In this example, page 1 is relocated to a higher range of physical addresses, page 4 is relocated to a lower range.

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)
Number of pages	8 per mode
Size of relocatable memory	32,768 words, maximum (8 × 4,096)

8.3 MEMORY MANAGEMENT REGISTERS

The memory management unit uses two sets of page address registers (PARs) and page descriptor registers (PDRs) referred to as PAR/PDR pairs. One set of PAR/PDR register pairs is used in kernel mode and the other set of register pairs in user mode. The choice of which set is to be used is determined by the current processor mode contained in processor status word (PS) bits <15:12>. The MMU also contains four status registers (SR0 through SR3) that implement various memory management functions. The memory management register functions are described in the following paragraphs.

8.3.1 Page Address Register (PAR)

The page address register contains the 16-bit page address field (PAF), which specifies the starting address of the page as a block number in physical memory. The page address register is shown in Figure 8-4.

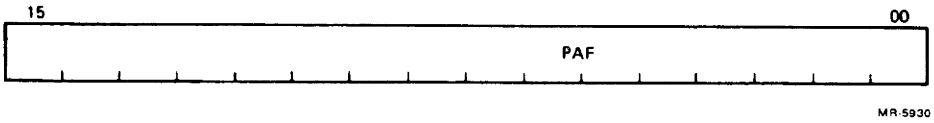


Figure 8-4 Page Address Register

The page address register may be thought of as a relocation constant, or as a base register containing a base address. Either interpretation indicates the basic function of the page address register (PAR) in the relocation scheme.

8.3.2 Page Descriptor Register (PDR)

The page descriptor register is a 16-bit register that contains information relative to page expansion, page length, and access control. The page descriptor register bit assignments are shown in Figure 8-5.

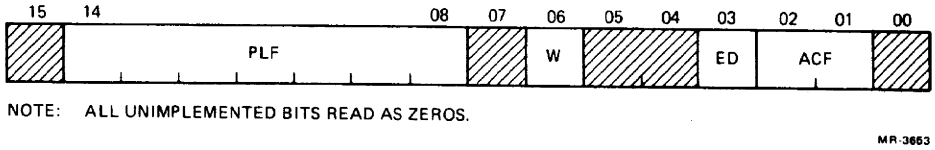


Figure 8-5 Page Descriptor Register

8.3.2.1 Access Control Field (ACF) – This 2-bit field (bits 2 and 1) of the PDR describes the access rights to a particular page. The access codes or keys specify the manner in which a page may be accessed, and whether or not a given access should result in an abort of the current operation. A memory reference that causes an abort is not completed and is terminated immediately.

Aborts are caused by attempts to access nonresident pages, by page-length errors, or by access violations, such as attempts to write into a read-only page. Traps are used as an aid in gathering memory management information.

In the context of access control, the term "write" is used to indicate the action of any instruction that modifies the contents of any addressable word. A write is synonymous with what is usually called a "store" or "modify" in many computer systems. Table 8-1 lists the ACF keys and their functions. The ACF is written into the PDR under program control.

Table 8-1 Access Control Field Keys

AFC	Key	Description	Function
00	0	Nonresident	Abort any attempt to access this nonresident page.
01	2	Resident read-only	Abort any attempt to write into this page.
10	4	(Unused)	Abort all accesses.
11	6	Resident read/write	Read or write allowed; No trap or abort occurs.

NOTE
A memory management abort causes the program to trap to location 250_8.

8.3.2.2 Expansion Direction (ED) – Bit 3 of the page description register (PDR) specifies in which direction the page expands. If ED = 0, the page expands upward from block number 0 to include blocks with higher addresses; if ED = 1, the page expands downward from block number 127 to include blocks with lower addresses. Upward expansion is usually used for program space while downward expansion is used for stack 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 the addition of 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.

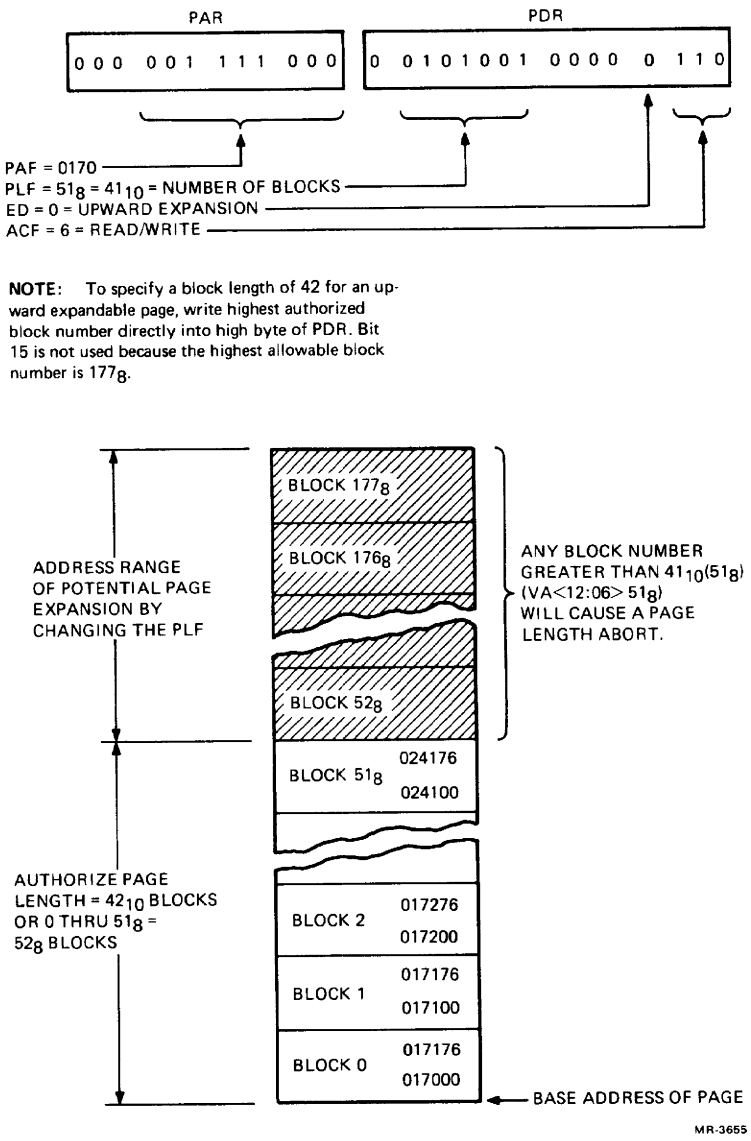


Figure 8-6 Example of an Upward-Expandable Page

8.3.2.3 Write Access Bit (W) (Bit 6) – This bit indicates whether or not this page has been modified (i.e., written into) since either the PAR or PDR was loaded. (W = 1 is affirmative). The W bit is useful in applications that involve disk swapping and memory overlays. It is used to determine which pages have been modified (and hence must be saved in their new form) and which pages have not been modified and can simply be overlaid. Note that the W bit is "reset" to "0" whenever either PAR or PDR is modified (written into).

8.3.2.4 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.

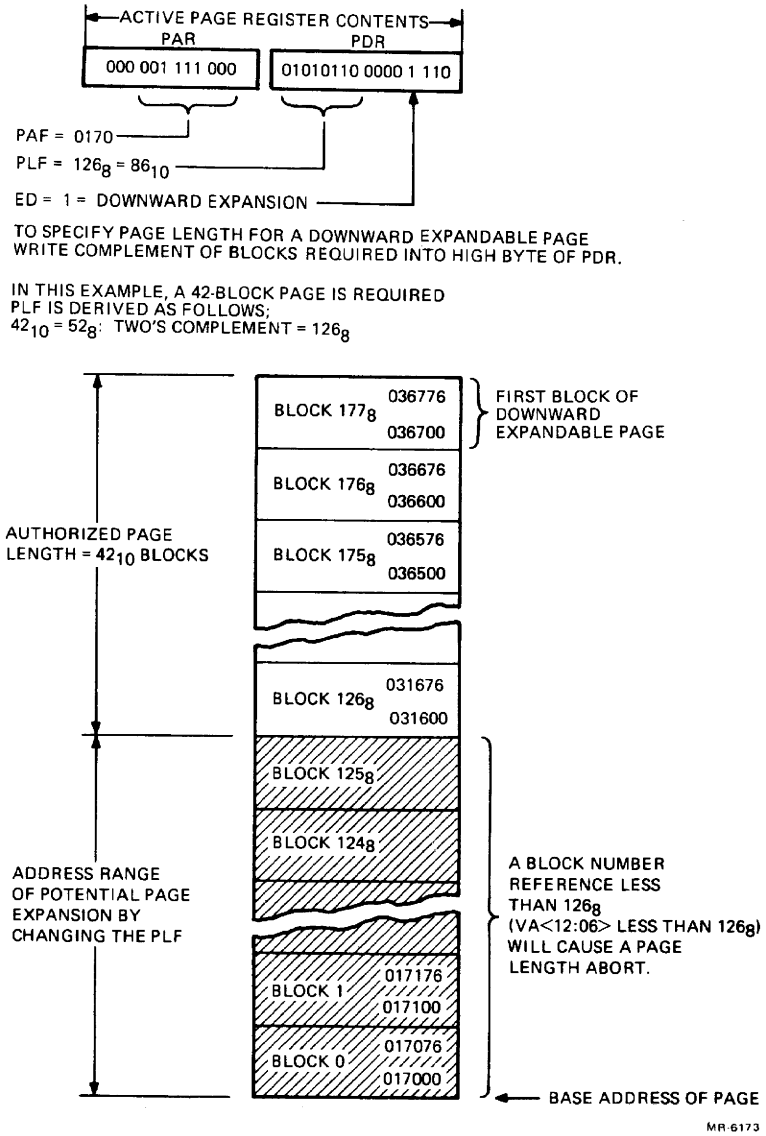


Figure 8-7 Example of a Downward-Expandable Page

When the page expands upward, the PFL must be set to 1 less than the number of blocks authorized for that page. For the example shown in Figure 8-6, since 52_8 (42_{10}) blocks are authorized, the PLF is set to 51_8 (41_{10}). 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 VA<12:06> is less than or equal to the PLF, the virtual address is within the authorized page length. If VA<12:06> is greater than the PLF, a page-length fault (address too high) is detected by the hardware and an MMU abort occurs.

When the page is to be downward-expandable, the PLF must be set to 200_8 (128_{10}) minus the length of the page (in blocks). For the example shown in Figure 8-7, since 52_8 (42_{10}) blocks are authorized, the PLF is set to 126_8 (86_{10}).

When VA<12:06> is greater than or equal to the PLF, the virtual address is within the authorized page length. If VA<12:06> is less than the PLF, a page-length fault (address too low) is detected by the hardware and an MMU abort occurs.

The downward-expandable example in Figure 8-7 uses the same PAF as the upward-expandable example in Figure 8-6. This is so to emphasize that the base address points to the lowest possible address of the 128 block page, whether the page is upward- or downward-expandable. As shown in Figure 8-7, the base address may not even be within the authorized page length of a downward-expandable page.

8.3.3 PAR/PDR Address Assignments

Addresses are assigned to the kernel and user active page registers as PAR/PDR register pairs. The PAR/PDR register addresses are listed in Table 8-2.

Table 8-2 PAR/PDR Address Assignments

Kernel Active Page Registers			User Active Page Registers		
No.	**PAR**	**PDR**	**No.**	**PAR**	**PDR**
0	17772340	17772300	0	17777640	17777600
1	17772342	17772302	1	17777642	17777602
2	17772344	17772304	2	17777644	17777604
3	17772346	17772306	3	17777646	17777606
4	17772350	17772310	4	17777650	17777610
5	17772352	17772312	5	17777652	17777612
6	17772354	17772314	6	17777654	17777614
7	17772356	17772316	7	17777656	17777616

8.3.4 Status Register 0 (SR0) – Address: 17777572_8

SR0 contains abort error flags, memory management enable, and other information essential for an operating system to recover from an abort or to service a memory management trap. The format of SR0 is shown in Figure 8-8.

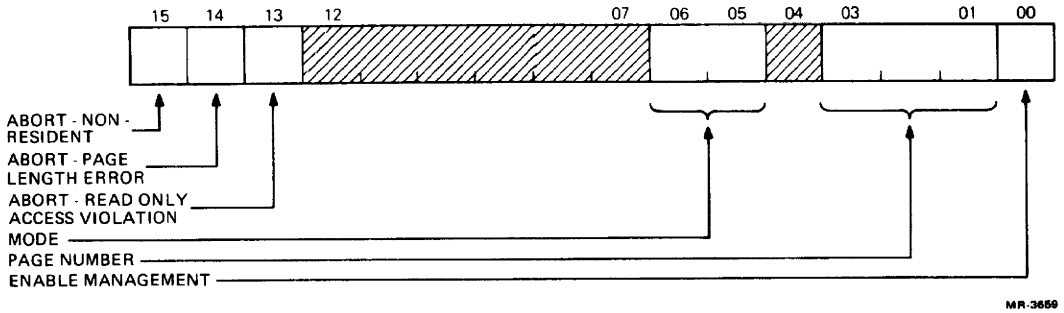


Figure 8-8 Format of Status Register 0 (SR0)

The enable management bit (SR0 bit 0) is set and cleared under program control to enable and disable memory management. The abort flag bits (SR0<15:13>) can also be set and cleared under program control, but they cause an MMU abort only when set automatically by an MMU abort condition. After an MMU abort has occurred, the program must clear SR0<15:13> in order to resume monitoring memory management status.

The abort flags are in priority order in that flags to the right are less significant and should be ignored when a more significant flag is asserted. For example, a nonresident abort service routine would ignore the page-length bit (14) and read-only access violation bit (13). A page-length abort service routine would ignore the read-only access violation bit.

The mode bits (SR0<06:05>) and the page number bits (SR0<03:01>) are loaded automatically when an MMU abort occurs. The status of these bits is frozen whenever one of the abort flags (SR0<15:13>) is set. The SRO is cleared by the RESET instruction, power-up or restart.

8.3.4.1 Abort Nonresident – Bit <15> is the abort nonresident bit. It is set by attempting to access a page with an access control field (ACF) key equal to 0 or 4, or by enabling relocation with an illegal mode in the PSW.

8.3.4.2 Abort Page Length – Bit <14> is the abort page-length bit. It is set by attempting to access a location in a page with a block number (virtual address bits <12:06>) that is outside the area authorized by the page-length field (PLF) of the PDR for that page.

8.3.4.3 Abort Read-Only – Bit <13> is the abort read-only bit. It is set by attempting to write-in a read-only page. Read-only pages have an access control field (ACF) key of 2_8.

NOTE
There are no restrictions against abort bits being set simultaneously by the same access attempt.

8.3.4.4 Mode of Operation – Bits <06:05> indicate the CPU mode (user or kernel) associated with the page causing the abort. (Kernel = 00, user = 11.)

8.3.4.5 Page Number – Bits <03:01> refer to the virtual page number that caused a memory management fault. Pages, like blocks, are numbered from 0 upward. The page number bits are used by the error recovery routine to identify the page being accessed if an abort occurs.

8.3.4.6 Enable Relocation and Protection – Bit <0> is the enable bit. When it is set to 1, all addresses are relocated and protected by the memory management unit. When this bit is set to 0, the memory management unit is disabled and addresses are neither relocated nor protected.

8.3.5 Status Register 1 (SR1) – Address: 17777574_8
SR1 is a read-only register that always reads as zero.

8.3.6 Status Register 2 (SR2) – Address: 17777576_8
SR2 is loaded with a 16-bit virtual address (VA) during 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. The content of SR2 is frozen whenever one of the abort flags (SR0<15:13>) is set. The format of SR2 is shown in Figure 8-9.

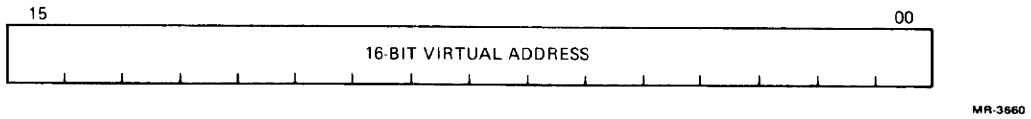


Figure 8-9 Format of Status Register 2 (SR2)

8.3.7 Status Register 3 (SR3) – Address: 17772516_8

SR3 bit <4> enables or disables the memory management 22-bit mapping. If memory management is not enabled (SR0 bit 0 is clear), bit 4 is ignored and the 16-bit address is not mapped. If memory management is enabled (SR0 bit 0 is set) and bit 4 is clear, the computer uses 18-bit mapping. If memory management is enabled and bit 4 is set, the computer uses 22-bit mapping.

SR3 bit <5> is a read/write bit that has no effect on KDF11-BA operation. On systems that contain an I/O map (e.g., the PDP-11/24), bit 5 is set to enable I/O map relocation and is cleared to disable relocation. Status register 3 is cleared by the RESET instruction, power-up or restart. The format of SR3 is shown in Figure 8-10.

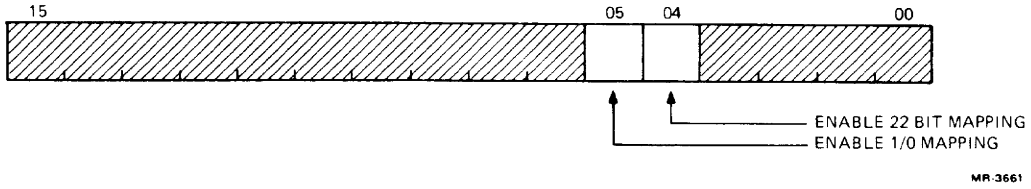


Figure 8-10 Format of Status Register 3 (SR3)

8.4 VIRTUAL AND PHYSICAL ADDRESSES

The memory management unit is located between the central processor unit and the LSI-11 bus address lines. 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- or 22-bit physical address. The information contained in the virtual address (VA) is combined with relocation information to yield an 18- or 22-bit physical address (PA). Using the memory management unit, memory can be dynamically allocated in pages, each page composed of from 1 to 128 integral blocks of 32 words.

The starting physical address of each page is an integral multiple of 32 words, and each page has a maximum size of 4096 words. Pages may be located anywhere within the physical address space. The current mode of the processor (kernel or user) determines which set of 16 PAR/PDR registers is used to construct the physical address.

8.4.1 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 8-11, and the appropriate APR set.

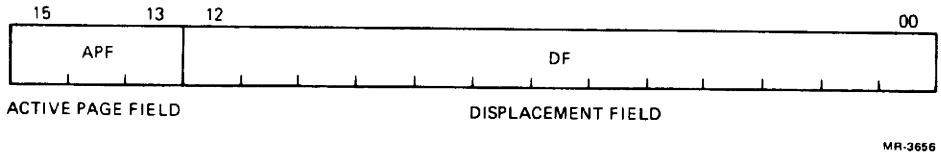


Figure 8-11 Interpretation of a Virtual Address

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 of up to 4K words (2^{13} = 8K bytes). The DF is divided into two fields as shown in Figure 8-12.

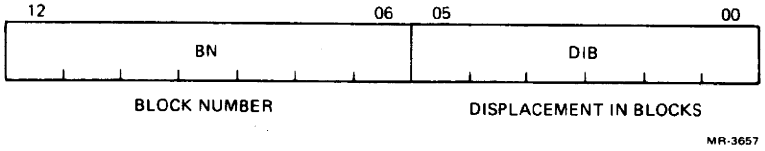


Figure 8-12 Displacement Field of Virtual Address

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- or 16-bit page address field (PAF) (contained in the active page register), specifying the starting address of the memory that APR describes. The PAF is actually a block number in the physical memory; for example, PAF = 3 indicates a starting address of 96 ($3 \times 32 = 96$) in physical memory. The formation of the physical address is illustrated in Figure 8-13.

The logical sequence involved in forming a physical address is as follows.

1. Select a set of active page registers. (Selection depends on the 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.

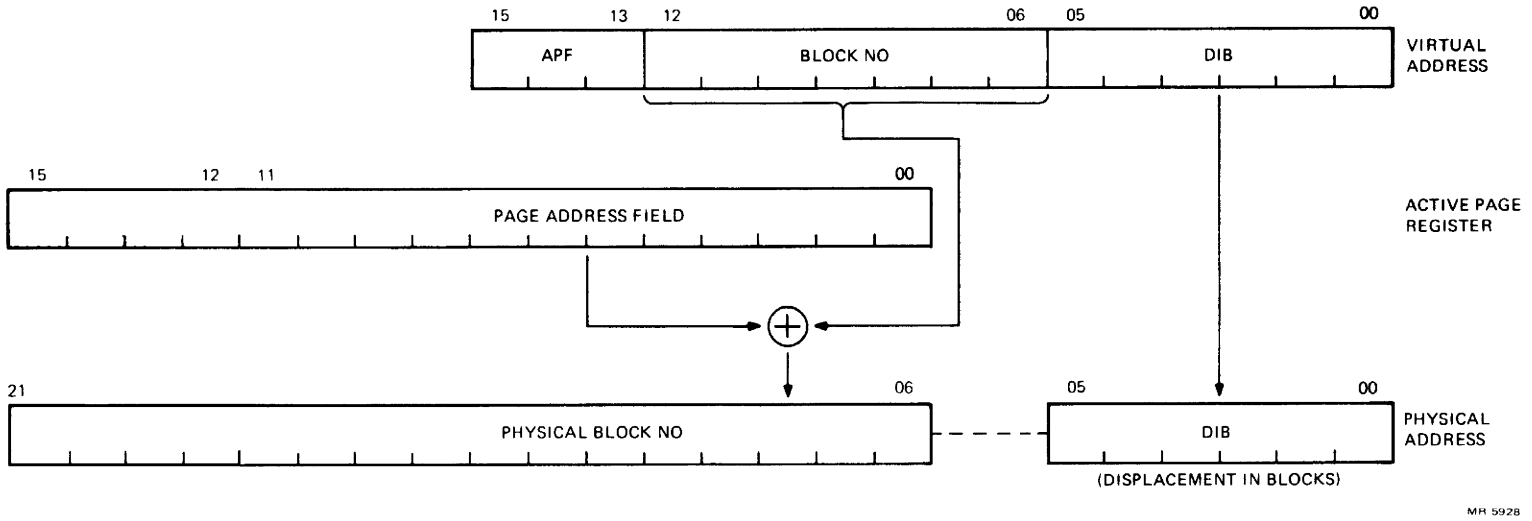


Figure 8-13 Formation of a Physical Address

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 that 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 a 22-bit physical address.

8.4.2 Determining the Program Physical Address

A 16-bit virtual address can specify up to 32K words, in the range of 000000 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 8-3 lists the virtual address ranges that specify each of the PAR/PDR sets.

Table 8-3 Relating Virtual Address Ranges to PAR/PDR Sets

Virtual Address Range	PAR/PDR Set
000000–17776	0
020000–37776	1
040000–57776	2
060000–77776	3
100000–117776	4
120000–137776	5
140000–157776	6
160000–177776	7

NOTE

Any use of page lengths of less than 4K words causes unaddressable "holes" in the virtual address space.

8.5 PROTECTION

A timesharing system performs multiprogramming; that is, it allows several programs to reside in memory simultaneously, executing each 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 to do so 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 types of memory protection listed above.

8.5.1 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 access control key functions are described in Table 8-1.) 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 from doing so by an immediate abort. 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.

8.5.2 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. It also 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 (in 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).

8.5.3 Multiple Address Space

Two complete PAR/PDR sets are provided: one for kernel mode and one 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's 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 program. 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 PAR and PDR register formats are shown in Figures 8-4 and 8-5.)

8.5.3.1 Mode Specification in the 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.

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.

8.5.3.2 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 8-4 shows how all PS bits are affected.

8.5.3.3 User Mode Restrictions – User mode is intended for executing user programs. 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 here 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.

8.5.3.4 Interrupt and Trap Processing – All interrupt and trap vectors are forced by hardware to be used always 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-BA 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.

Table 8-4 Processor Status Word Protection

PS Bits	RTI, RTT User	RTI, RTT Kernel	Traps and Interrupts User	Traps and Interrupts Kernel	Explicit PS Access User	Explicit PS Access Kernel	MTPS User	MTPS Kernel	Power-Up
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	Set when powered up to bootstrap. Cleared when powered up to ODT, location 24, or microcode.
SI PS<8>	Loaded from stack	Loaded from stack	Loaded from vector	Loaded from vector	Loaded from source	Loaded from source	Nonaccessible	Nonaccessible	Cleared
Previous Mode PS<13:12>	Unchanged	Loaded from stack	Copied from PS<15:14>	Copied from PS<15:14>	Loaded from source	Loaded from source	Nonaccessible	Nonaccessible	Cleared
Current Mode PS<15:14>	Unchanged	Loaded from stack	Loaded from vector	Loaded from vector	Loaded from source	Loaded from source	Nonaccessible	Nonaccessible	Cleared

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.

8.6 MEMORY MANAGEMENT INSTRUCTIONS

Memory management provides communications between two spaces, as determined by the current memory management mode bits (PS<15:14>) and previous memory management mode bits (PS<13:12>) 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 for a more detailed description. These instructions are directly compatible with larger PDP-11 computers.

CHAPTER 9
FLOATING-POINT ARITHMETIC

9.1 INTRODUCTION

Forty-six floating-point instructions are available as a microcode option (KEF11-AA) for use with the KDF11-BA processor. The KEF11-AA 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 with other features, including floating-to-integer and integer-to-floating conversion.

The KEF11-AA consists of two MOS/LSI chips contained in one 40-pin package. Operation of the KEF11-AA 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 $1/2 \; f < 1$. 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 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 (the true exponent of -200_8), is reserved for floating-point 0. Therefore, 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 (floating-point processor) 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

An undefined variable is 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 chapter.

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 -0 occurs becomes present, 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.

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.)

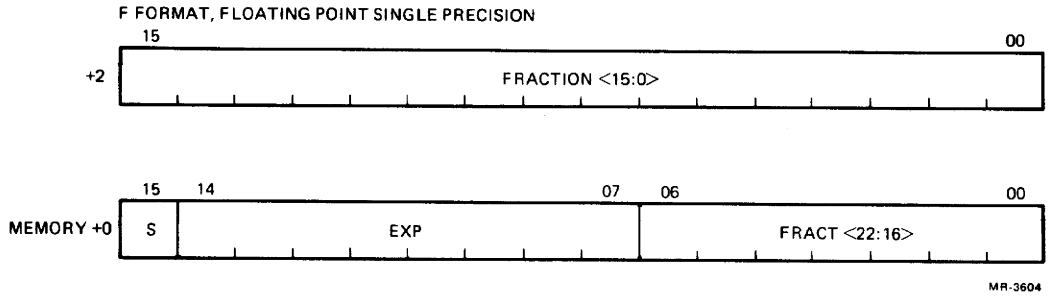


Figure 9-1 Single-Precision Format

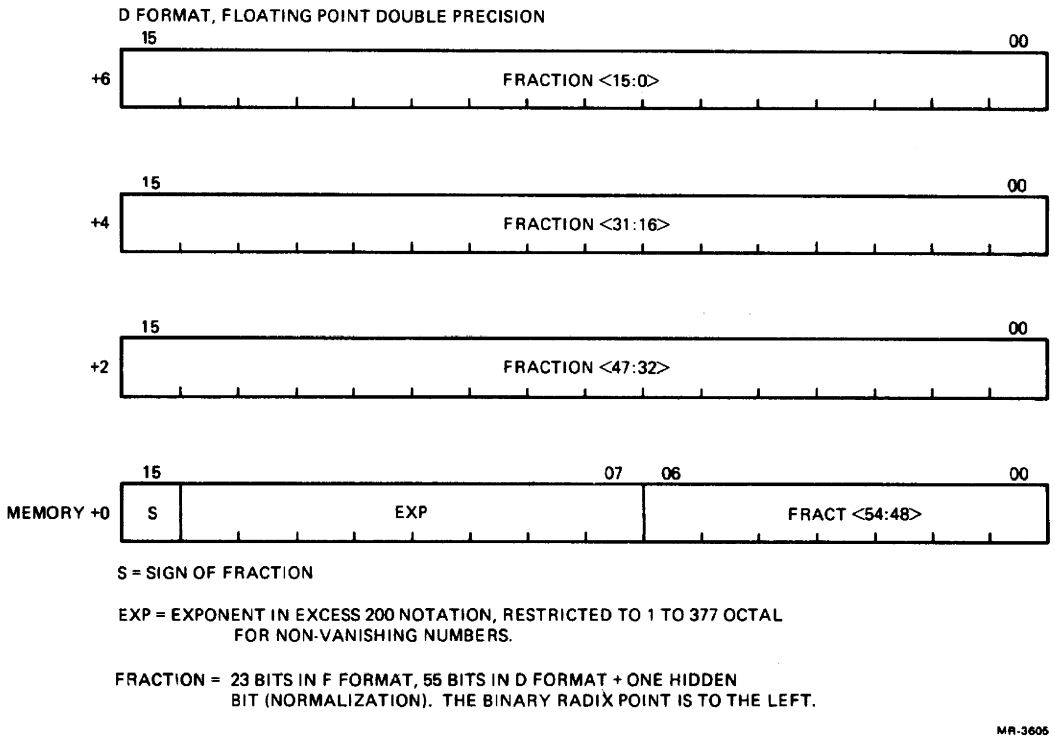


Figure 9-2 Double-Precision Format

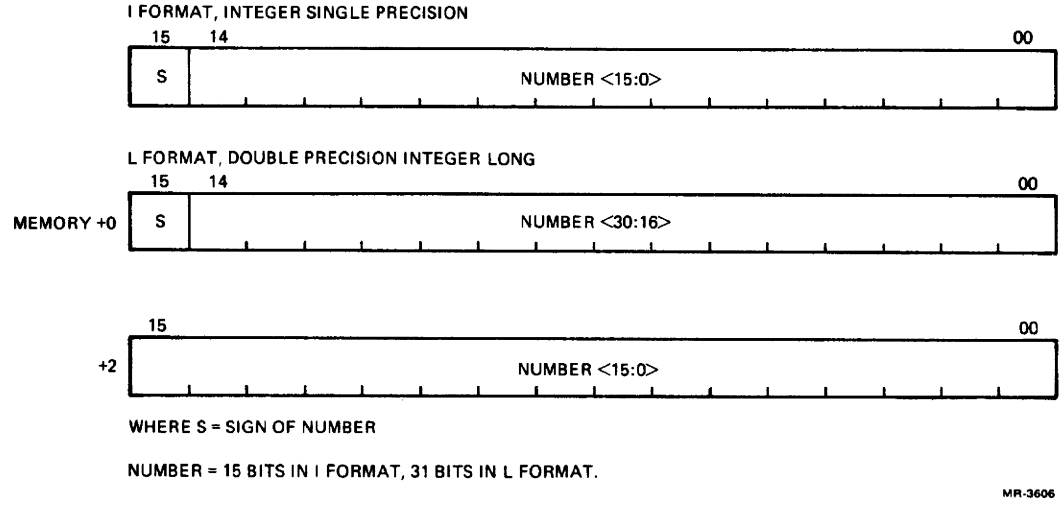


Figure 9-3 2's Complement Format

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.) In this discussion a set bit = 1 and a reset bit = 0. Three bits of the FPS register control the modes of operation.

1. Single/Double – Floating-point numbers can be either single- or double-precision.

2. Long/Short – Integer numbers can be 16 bits or 32 bits.

3. 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.

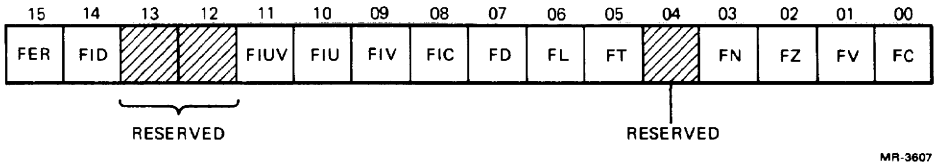


Figure 9-4 Floating-Point Status Register

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 op code

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 that disables interrupts on *all six* of the exceptions, as a group.

Of the 13 FPS bits, 5 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.

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 op-code error
4	Floating divide by 0
6	Floating-to-integer conversion error
8	Floating overflow
10	Floating underflow
12	Floating undefined variable

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. an illegal op code occurs, 3. any one of the remaining floating-point exceptions 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. **NOTE** **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 assume that the FID bit is clear in all discussions involving overflow, underflow, occurrence of −0, and integer conversion errors.**
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-AA will not store a result of −0 without the simultaneous occurrence of an interrupt.

Table 9-1 FPS Register Bits (Cont)

Bit	Name	Description
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. 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.
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 formats. 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).

Table 9-1 FPS Register Bits (Cont)

Bit	Name	Description
5	Floating Chop Mode (FT)	When the FT bit is set, the result of any arithmetic operation is chopped (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.
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.

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. Division by 0.
2. Illegal op code.
3. Any of the other four exceptions with the corresponding interrupt enabled.

This implies that only when the FER bit is set by the FPP are the FEC and FEA registers updated.

NOTE

1. **If one of the last four exceptions occurs with the corresponding interrupt disabled, the FEC and FEA are not updated.**

2. **If an exception occurs, inhibition of interrupts by the FID bit does not inhibit updating of the FEC and FEA.**

3. **The FEC and FEA are not 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, 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 in 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_8 for F format and 10_8 for D format.

In mode 0 users can make use of all six FPP accumulators (AC0–AC5) as their source or destination. Specifying FPP accumulators AC6 or AC7 will result in an illegal op code trap. In all other modes, which involve transfer of data to or from memory or the general registers, users are 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-AA 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 chapter, 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 is:

   a. always decreased by chopping.
   b. decreased by rounding if the rounding bit is 0.
   c. 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) in Table 9-1.

The FIV and FIU bits (of the floating-point status word) provide users with an opportunity to implement their 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 users can identify the cause by examination of the FV (floating overflow) bit of the FEC (floating exception) register. You 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 the biased exponent is not 0, it is too large by 400_8.

Thus, with the interrupt enable, enough information is available to determine the correct answer. Users may, for example, rescale their 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.

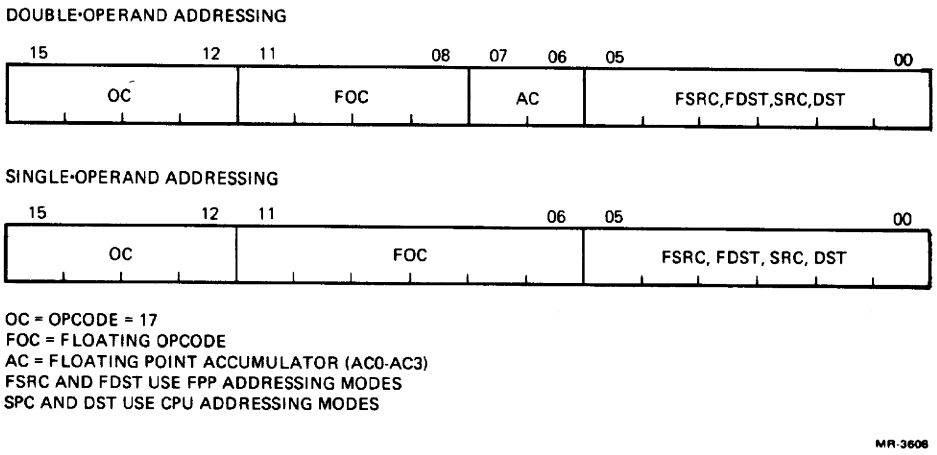


Figure 9-5 Floating-Point Addressing Modes

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

Boolean Symbols

$\wedge$ = AND

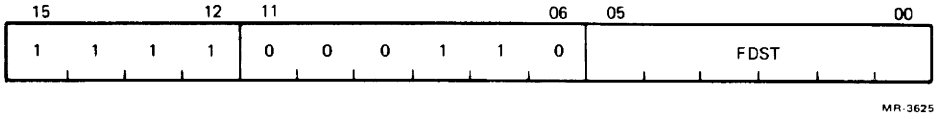
$\vee$ = inclusive OR

$\forall$ = exclusive OR

$\sim$ = NOT

ABSF/ABSD

Make absolute floating/double 1706 FDST



Format: ABSF FDST

Operation: If (FDST) $<$ 0, (FDST) $\leftarrow$ $-$(FDST).

If EXP(FDST) = 0, (FDST) $\leftarrow$ exact 0.

For all other cases, (FDST) $\leftarrow$ (FDST).

Condition Codes: FC $\leftarrow$ 0
FV $\leftarrow$ 0
FZ $\leftarrow$ 1 if (FDST) = 0, else FZ $\leftarrow$ 0
FN $\leftarrow$ 0

Description: Set the contents of FDST to its absolute value.

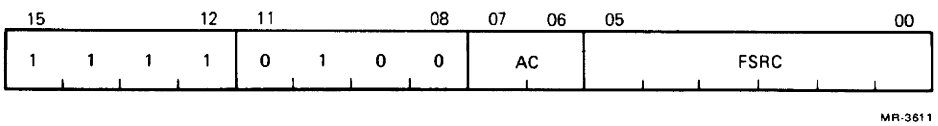
Interrupts: If FIUV is enabled, trap on -0 occurs after execution. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: If a -0 is present in memory and the FIUV bit is enabled, an exact 0 is stored in memory. The condition codes reflect an exact 0 (FZ $\leftarrow$ 1).

ADDF/ADDD

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 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.

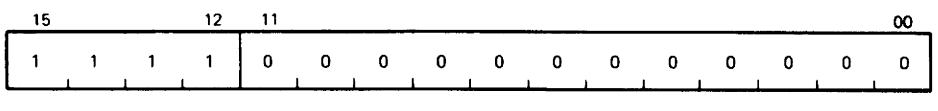
Accuracy: Errors due to overflow and underflow are described above. If neither occurs, then: for oppositely signed operands with exponent difference of 0 or 1, the answer returned is exact if a loss of significance of one or more bits can occur. Note that these are the only cases for which loss of significance 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.

CFCC

Copy floating condition codes 170000



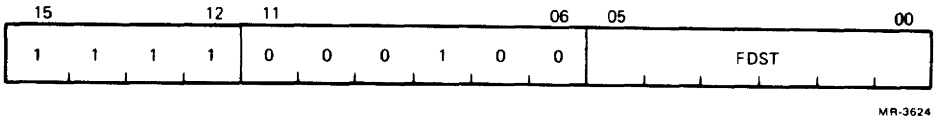
Format: CFCC

Operation:
C ← FC
V ← FV
Z ← FZ
N ← FN

Description: Copy the FPP condition codes into the CPU's condition codes.

CLRF/CLRD

Clear floating/double 1704 FDST



Format: CLRF FDST

Operation: (FDST) ← exact 0

Condition Codes:
FC ← 0
FV ← 0
FZ ← 1
FN ← 0

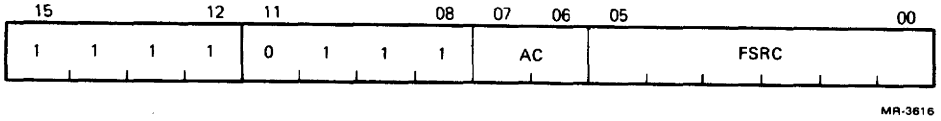
Description: Set FDST to 0. Set FZ condition code and clear other condition code bits.

Interrupts: No interrupts will occur. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

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.

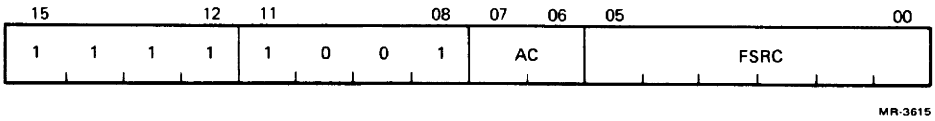
Interrupts: If FIUV is enabled, trap on −0 occurs before execution.

Accuracy: These instructions are exact.

Special Comment: An operand that 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.

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.

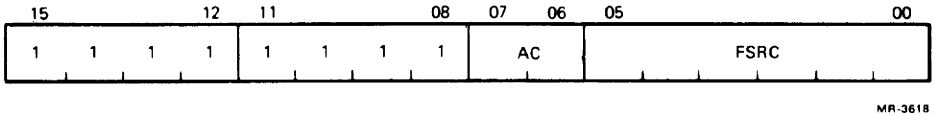
Interrupts: If FIUV is enabled, trap on −0 in FSRC occurs before execution. If (FSRC) = 0, interrupt traps on an attempt to divide by 0. If overflow or underflow occurs, and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. 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.

LDCDF/LDCFD

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.

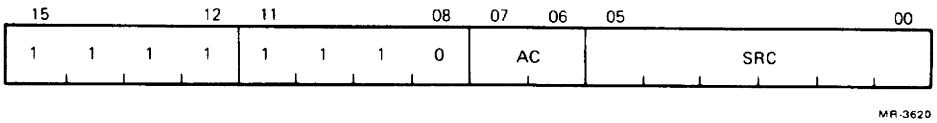
A trap occurs if FIV is enabled, and if rounding with LDCDF causes overflow. AC ← overflowed result. This result must be +0 or −0. Underflow cannot occur.

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.

LDCIF/LDCID/LDCLF/LDCLD

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.

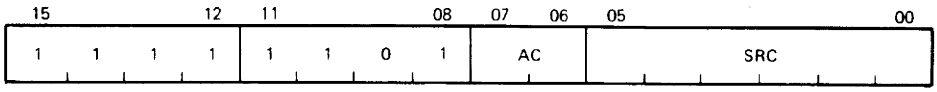
In the case of LDCLF, the fractional part of the floating-point representation is chopped or rounded to 24 bits for FT = 1 or 0, respectively.

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.

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) $\leftarrow$ SRC + 200 and the rest of AC is unchanged.

If (SRC) $>$ 177 and FIV is enabled, EXP(AC) $\leftarrow$ [(SRC) + 200]<7:0>.

If (SRC) $>$ 177 and FIV is disabled, AC $\leftarrow$ exact 0.

If <SRC) < -177 and FIU is enabled, EXP(AC) $\leftarrow$ [(SRC) + 200]<7:0>.

If (SRC) < -177 and FIU is disabled, AC $\leftarrow$ exact 0.

Condition Codes:
FC $\leftarrow$ 0
FV $\leftarrow$ 1 if (SRC) $>$ 177, else FV $\leftarrow$ 0
FZ $\leftarrow$ 1 if (AC) $=$ 0, else FZ $\leftarrow$ 0
FN $\leftarrow$ 1 if (AC) $<$ 0, else FN $\leftarrow$ 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 into 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 FP11-C and FP11-B do, but it does treat them the same as the FP11-A and FP11-E do.

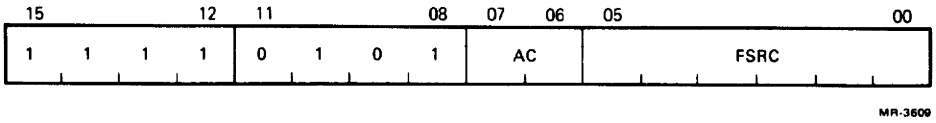
Interrupts: No trap on -0 in AC occurs, even if FIUV is enabled. If SRC $>$ 177 and FIV is enabled, trap on overflow will occur. If SRC < -177 and FIU is enabled, trap on underflow will occur.

Accuracy: Errors due to overflow and underflow are described above. If EXP(AC) = 0 and (SRC) $\neq -200$, AC changes from a floating-point number treated as 0 by all floating arithmetic operations to a non-0 number. This happens 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.

LDF/LDD

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.

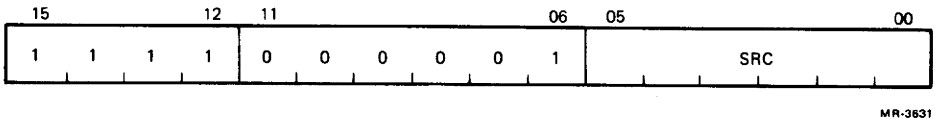
Interrupts: If FIUV is enabled, trap on −0 occurs before AC is loaded. However, the condition codes will reflect a fetch of −0 regardless of the FIUV bit. Overflow and underflow cannot occur.

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.

LDFPS

Load FPP's program status — 1701 SRC



Format: LDFPS SRC

Operation: FPS ← (SRC)

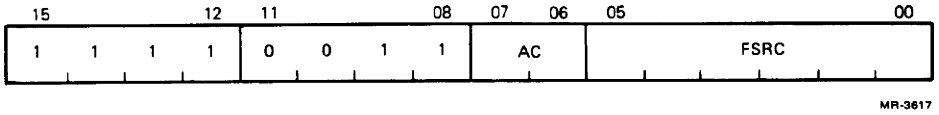
Description: Load FPP's status register from SRC.

Special Comment: Users are cautioned not to use bits 13, 12, and 4 for their own purposes, since these bits are not recoverable by the STFPS instruction.

MODF/MODD

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) = $(200 + K)_8$

Fixed-point binary: PROD = N + g, where

N = INT(PROD) = integer part of PROD, and

g = PROD − INT(PROD) = 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 AC ∨ 1 and g is returned to AC.

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, AC ∨ 1 ← N, chopped to L bits, AC ← exact 0.

   Note that EXP(N) is too small by 400_8 and that −0 can be stored in AC ∨ 1.

   If FIV is not enabled, AC ∨ 1 ← exact 0, AC ← exact 0, and −0 will never be stored.

2. If 2 ** L .LE. ABS(PROD) and no overflow, AC ∨ 1 ← 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, AC ∨ 1 ← 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, AC ∨ 1 ← 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, AC ∨ 1 ← 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, AC ∨ 1 ← 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.

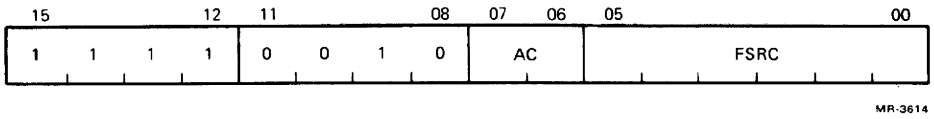
   ARG * 2/PI = N + g. The low two bits of N identify the quadrant, and g is the argument reduced to the first quadrant. The accuracy of N + g is limited to L bits because of the factor 2/PI. The accuracy of the reduced argument thus depends on the size of N.

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.

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 or 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.

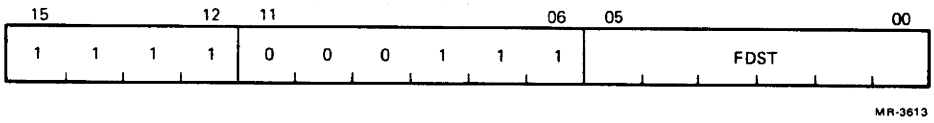
Interrupts: If FIUV is enabled, trap on −0 in FSRC occurs before execution. If overflow or underflow occurs, and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It 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.

NEGF/NEGD

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 the single- or double-precision number; store result in same location (FDST).

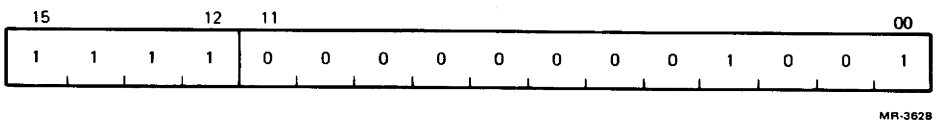
Interrupts: If FIUV is enabled, trap on −0 occurs after execution. Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comment: If a −0 is present in memory and the FIUV bit is enabled, the KEF11-AA stores an exact 0 in memory. The condition codes reflect an exact 0 (FZ ← 1).

SETD

Set floating double mode 170011



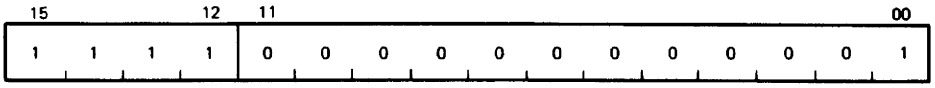
Format: SETD

Operation: FD ← 1

Description: Set the FPP in double-precision mode.

SETF

Set floating mode 170001



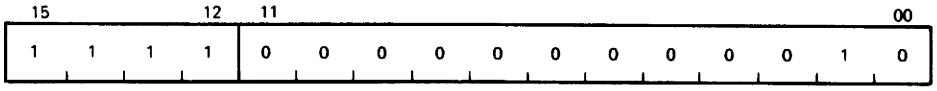
Format: SETF

Operation: FD ← 0

Description: Set the FPP in single-precision mode.

SETI

Set integer mode 177002



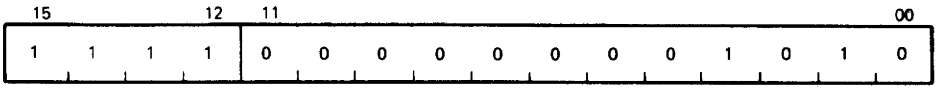
Format: SETI

Operation: FL ← 0

Description: Set the FPP for short-integer data.

SETL

Set long-interger mode 177012



Format: SETL

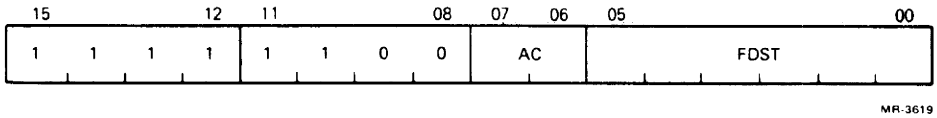
Operation: FL ← 1

Description: Set the FPP for long-integer data.

STCFD/STCDF

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.

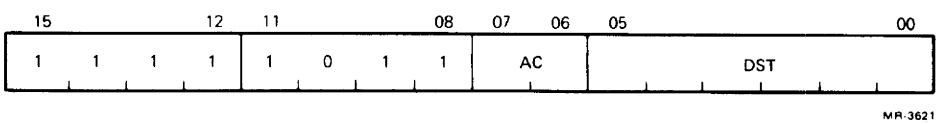
A trap occurs if FIV is enabled, and if rounding with STCDF causes overflow. (FDST) ← overflowed result. This must be +0 or −0.

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.

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.

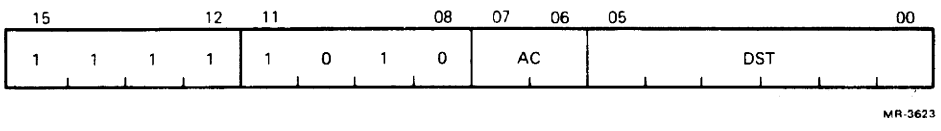
Numbers to be converted are always chopped (rather than rounded) before they are converted. This is true even when the chop mode bit FT is cleared in the FPS register.

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.

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

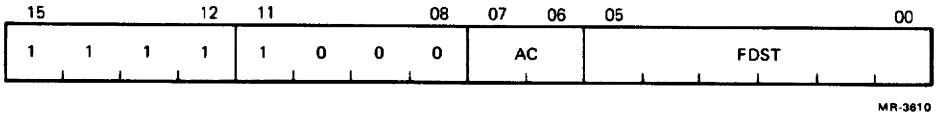
Description: Convert AC's exponent from excess 200 notation to 2's complement and store the result in DST.

Interrupts: This instruction will not trap on −0. Overflow and underflow cannot occur.

Accuracy: This instruction is exact.

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.

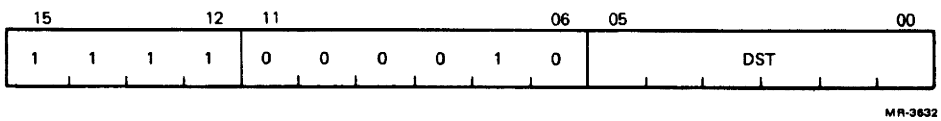
Interrupts: These instructions do not interrupt if FIUV is enabled, because the −0, if present, is in AC, not in memory. Overflow and underflow cannot occur.

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.

STFPS

Store FPP's program status 1702 DST



Format: STFPS DST

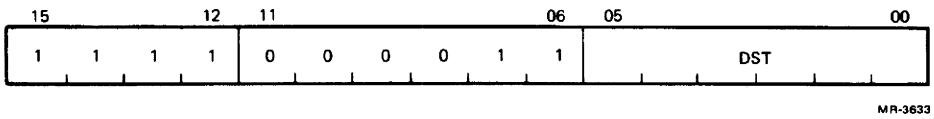
Operation: (DST) ← FPS

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

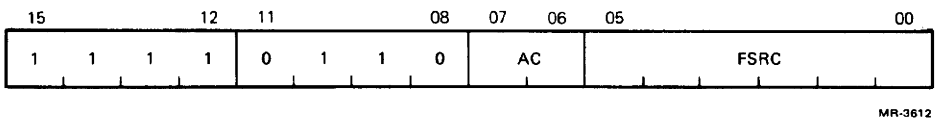
Description: Store the FEC and FEA in DST and DST+2. Note the following.

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.

SUBF/SUBD

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: 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:

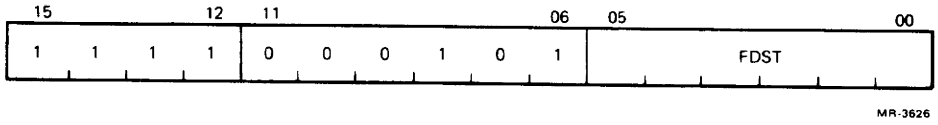
1. LSB in chopping mode with either single- or double-precision.
2. 1/2 LSB in rounding mode with either single- or double-precision.

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.

TSTF/TSTD

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.

CHAPTER 10
PROGRAMMING TECHNIQUES

10.1 INTRODUCTION

The KDF11-BA 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-BA. The techniques discussed involve position-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 so 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-BA processor offers addressing modes that make it possible to write instructions that do not depend 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 (a library routine, for example). 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 you are familiar with the addressing modes described in Chapter 6.

The following addressing modes, which involve only register references, are position-independent.

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 independently of a particular virtual memory location.

Two relative addressing modes are position-independent when a relocatable address is referenced from a relocatable instruction:

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 such 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:

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 Comparison of Position-Dependent and Position-Independent Code

The RSX-11 library routine, PWRUP, is a FORTRAN-callable subroutine for establishing or removing a user power failure asynchronous system trap (AST) entry point address. Imbedded within the routine is the actual AST entry point that 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-BA. It is an area of memory set aside by the programmer or 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 lower addresses as items are added.

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 (see Figure 10-1) require instructions capable of operating on bytes rather than full words.

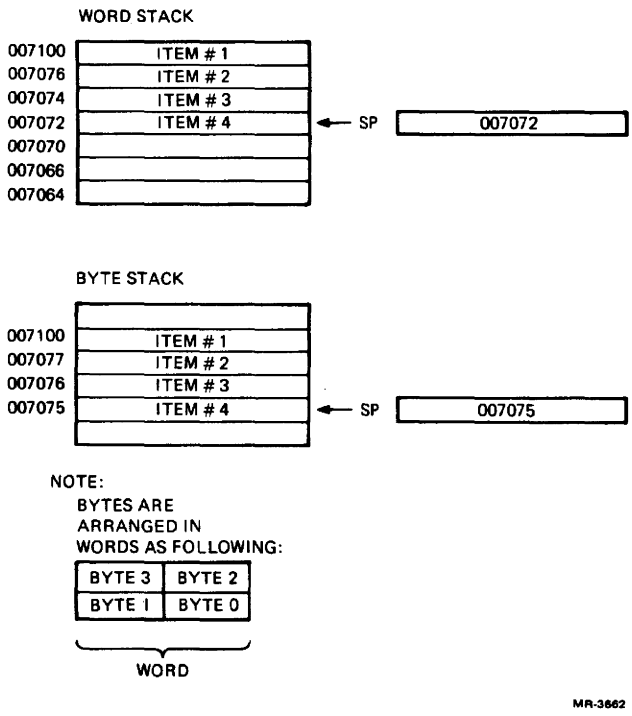


Figure 10-1 Word and Byte Stacks

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 POPping. 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.)

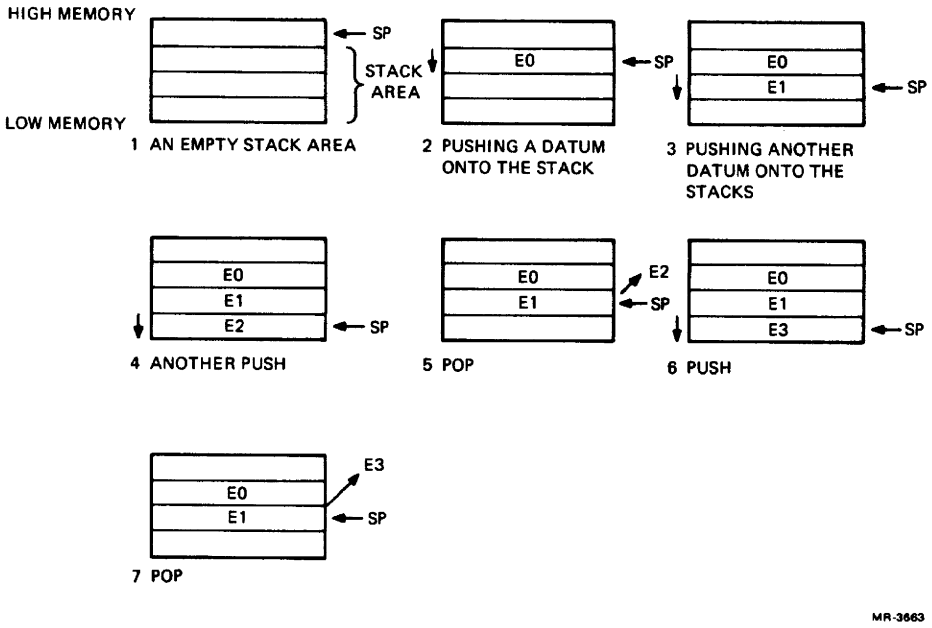


Figure 10-2 Push and Pop Operations

10.3.3 Deleting Items from a Stack

The following techniques may be used to delete from a stack. To delete one item use:

INC SP or TSTB(SP)+ for a byte stack

To delete two items use:

ADD#2,SP or TST(SP)+ for word stack

To delete fifty items from a word stack use:

ADD#100.,SP

10.3.4 Stack Uses

A stack is 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.

3. If no arguments need be passed by stacking them after the JSR instruction, the PC may be used as the linkage register. In this case, the result of the JSR is to move the return address in the calling program from the PC onto the stack and replace it with the entry address of the called subroutine.

4. In many cases, the operations performed by the subroutine can be applied directly to the data located on or pointed to by a stack without the need to move the data into the subroutine area.

   Example:

```
                        ;CALLING PROGRAM
MOV SP,R1               ;R1 IS USED AS THE STACK
JSR PC,SUBR             ;POINTER HERE.

                        ;SUBROUTINE
ADD (R1)+,(R1)          ;ADD ITEM #1 TO #2, PLACE
                        ;RESULT IN ITEM #2,
                        ;R1 POINTS TO
                        ;ITEM #2 NOW
```

   Because the hardware already uses general-purpose register R6 to point to a stack for saving and restoring PC and processor status word (PS) information, it is convenient to use the same stack to save and restore immediate results and to transmit arguments to and from subroutines. Using R6 in this manner permits extreme flexibility in nesting subroutines and interrupt service routines.

   Since arguments may be obtained from the stack by using some form of register-indexed addressing, it is sometimes useful to save a temporary copy of R6 in some other register which has been saved at the beginning of a subroutine. If R6 is saved in R5 at the beginning of the subroutine, R5 may be used to index the arguments. During this time R6 is free to be incremented and decremented while being used as a stack pointer. If R6 had been used directly as the base for indexing and not "copied," it might be difficult to keep track of the position in the argument list, since the base of the stack would change with every autoincrement/decrement that occurred.

   However, if the contents of R6 (SP) are saved in R5 before any arguments are pushed onto the stack, the position relative to R5 would remain constant.

Return from a subroutine also involves the stack, as the return instruction, RTS, must retrieve information stored there by the JSR.

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 that 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 maintaining 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 because R6 may point only to word (even) locations. (See Figure 10-3.)

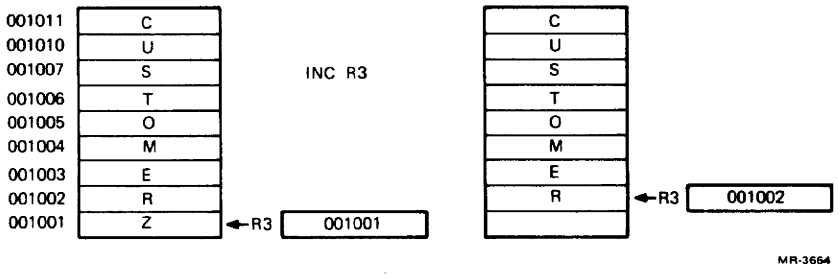


Figure 10-3 Byte Stack Used as a Character Buffer

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 shows the conditions before and after executing the subroutine instructions JSR R5, 1064.

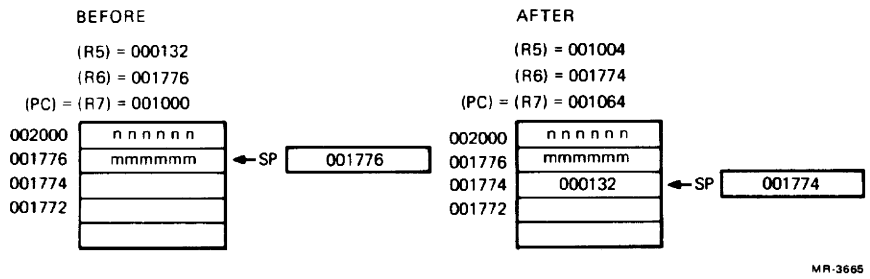


Figure 10-4 JSR Stack Condition Example

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 that 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 is 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 that 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-BA stack organization occur in programming systems that handle several tasks. Multitask program environments range from simple single-user applications that manage a mixture of I/O interrupt service and background data processing (as in RT-11), to large, complex multiprogramming systems that manage an intricate mixture of executive and multiuser 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.

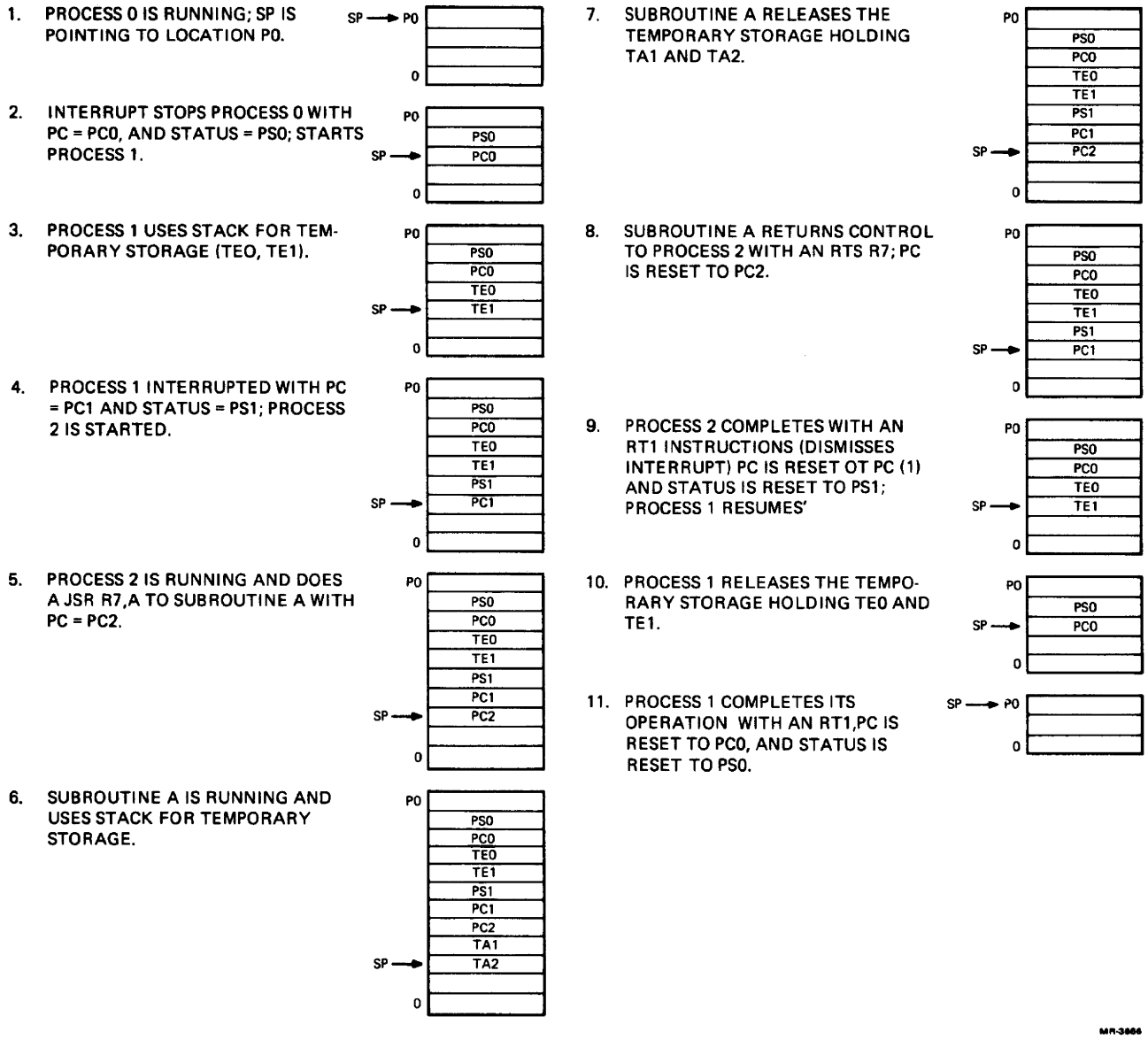


Figure 10-5 Nested Interrupt Service Routines and Subroutines

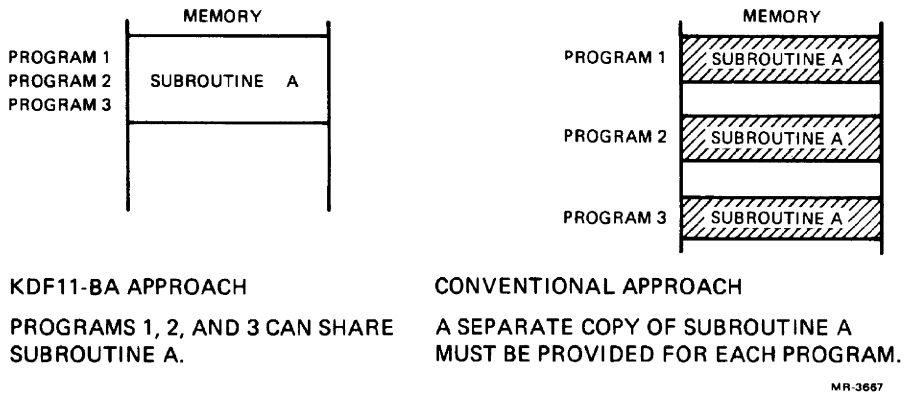


Figure 10-6 Reentrant Routines

10.3.8.1 Reentrant Code – Reentrant routines must be written in pure code (that is, 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).

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 gives up 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.

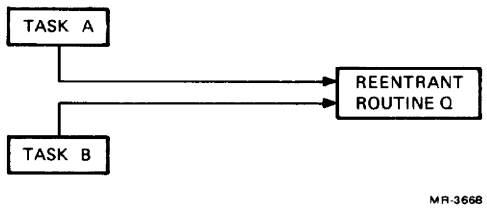


Figure 10-7 Sharing Control of a Routine

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 in which the processor status word and current contents of the general-purpose registers (GPRs) are 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. It 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 call of the 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. This JSR instruction results in an exchange of the contents of the PC and the top element of the stack; this permits 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 control transfer.

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. When called, a subroutine executes 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 instructions for coroutines are the same:

```
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. Refer to Paragraph 10.3.6.1 for information on the return from subroutine instruction.

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.

ROUTINE A	STACK		ROUTINE B	COMMENTS
				LOC IS PUSHED ONTO THE STACK TO PREPARE FOR THE COROUTINE CALL.
MOV #LOC,-(SP)	LOC	←SP		
			LOC:	
JSR PC,@(SP)+ (PC0)	PC0	←SP		WHEN THE CAL' IS EXECUTED, THE PC FROM ROUTINE A IS PUSHED ON THE STACK AND EXE-CUTION CONTIN-UES AT LOC.
	PC1	SP	JSR PC,@(SP)+ (PC1)	ROUTINE B CAN 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

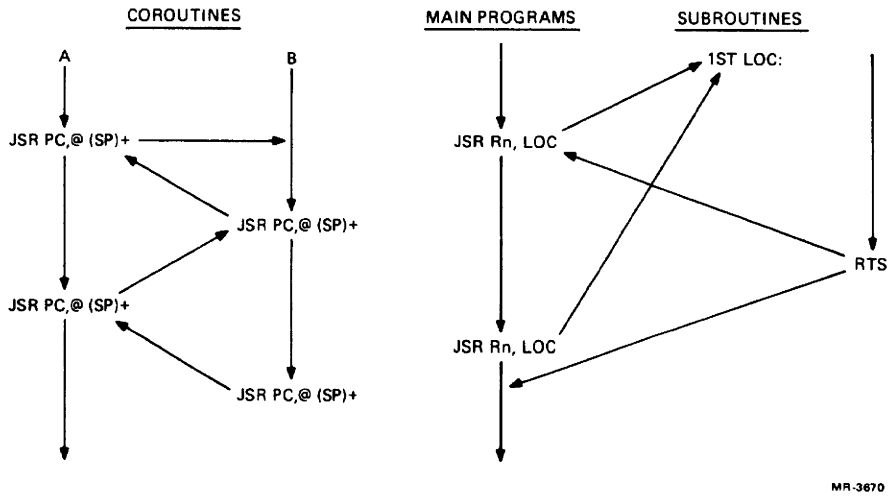


Figure 10-9 Coroutines Versus Subroutines

10.3.9.3 Using Coroutines – Coroutines should be used in the following situations.

1. 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. A detected label must be processed. If no label is present, the operator must be located, etc.
2. 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's 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. Such a subroutine would be too isolated. Coroutines could be effectively used here with one routine being the assembly-pass-1 routine and the other extracting one item at a time from the current input line. Figure 10-10 illustrates this example.

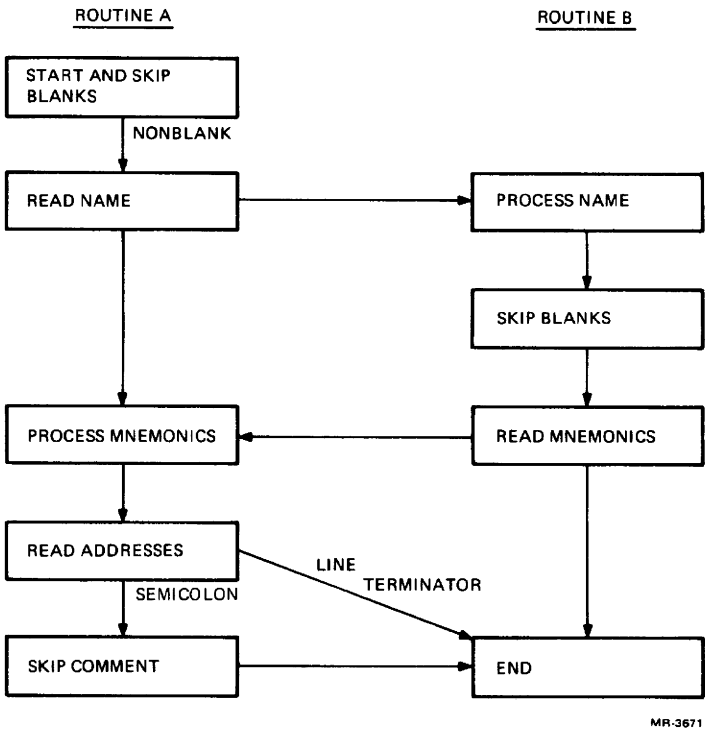


Figure 10-10 Coroutine Path

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
Process I2 } concurrently,

then

Write 02
Read I2
Process I1 } concurrently.

Figure 10-11 illustrates a coroutine swapping interaction.

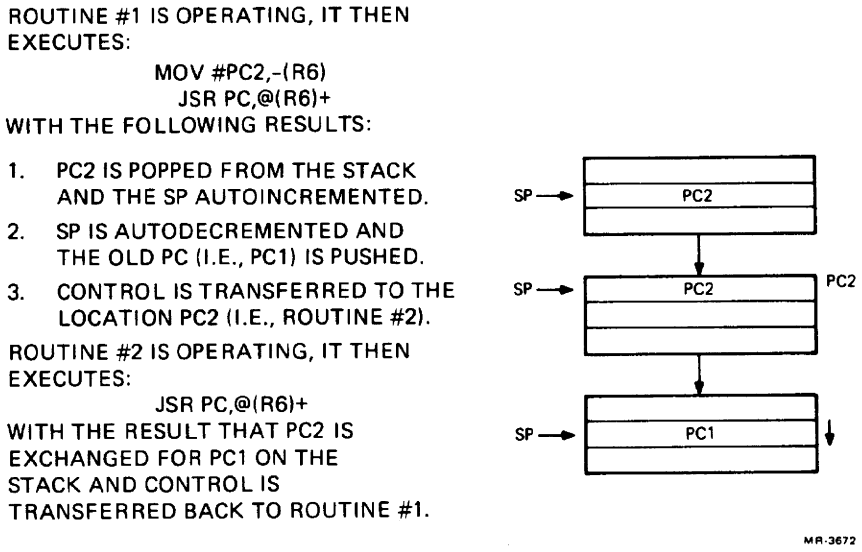


Figure 10-11 Coroutine Interaction

When routine #1 is operating; it 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).

When routine #2 is operating; it 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 resemble that shown in Figure 10-12.

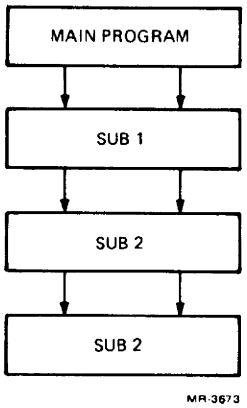


Figure 10-12 Recursive Routine Flow

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 a final time to return to the main program.

In general, recursion techniques will lead to slower programs than the corresponding interactive techniques, but recursion will produce shorter programs, and thus save memory space. 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, as in solving 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 since 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; that is, it must process one macro before it is finished with another, then 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-BA 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 PS bit 4 is set at the end of 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.

(Continued)

*Nonmaskable software cannot inhibit the condition. CTLERR, MMUERR, BUSERR, PARERR are mutually exclusive when the processor is executing a program.

<table>
<tr><th>Condition</th><th>Description</th></tr>
<tr><td>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>)</td><td>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.

PS<07:05> | Levels Inhibited
7 | All
6 | 6, 5, 4
5 | 5, 4
4 | 4
0–3 | None</td></tr>
<tr><td>Halt Line</td><td>If the BHALT L bus signal is asserted during the service state, the processor will enter ODT mode.</td></tr>
</table>

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.***

A 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.

A 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 the example below.

```
.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 that 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:

```
    .MACRO CM3 CHAN, CODE
    MOV #CODE *400,R0
.IIF NB                 HAN,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. Code that performs such a transformation is called a *conversion routine* in this guide. Several commonly used conversion routines follow.

Almost all assembly language programs involve some type of conversion routine. 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.

   Example:

   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. The second method of division works 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 the next lower power, and count for $10^3 = 0$.

```
  221
-100

  121   count for 10^2 = 1
-100

   21    count = 2
-100
```

Negative, so reduce power, and 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 with a combination of rotates and additions or with 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 17777776. The KDF11-BA 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 @#17777776
MOV @#17777776, 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 (17777776) can be accessed on a word or byte basis. The KDF11-BA will recognize the PS odd address (17777777) 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 8.5.3.2 for more details.

10.5 PROGRAMMING PERIPHERALS

Programming LSI-11 bus-compatible modules (devices) is simple. A special class of instructions that 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 one or a 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  Program
Address  Contents  Label     Op Code Operand     Comments

                                                 ;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
```

```
Program  Program
Address  Contents  Label    Op Code Operand     Comments

                                                ;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
```

```
Program  Program
Address  Contents  Label    Op Code Operand     Comments

                                                ;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!!!
```

```
Program  Program
Address  Contents  Label    Op Code Operand     Comments

                                                ;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
```

CHAPTER 11
BOOTSTRAP AND DIAGNOSTIC LOGIC

11.1 INTRODUCTION

The bootstrap and diagnostic logic features three hardware registers and two ROM sockets for 2K, 4K or 8K of 16-bit read-only memory. This 16-bit read-only memory typically contains diagnostic programs and a selection of bootstrap programs. These programs are user-selectable by setting eight switches on a 16-pin DIP switch pack (E102). Programming the bootstrap and diagnostic logic consists of setting the switches for the programs desired and the supplying of inputs by the console operator. The bootstrap/diagnostic switch configurations and console operator responses are described in Chapter 2, Paragraph 2.2.4.1. The diagnostic programs test the processor, the memory and the user's console.

The KDF11-BA includes two BDV11-compatible 2K $\times$ 8 ROMs that are installed in ROM sockets E126 (low byte) and E127 (high byte). The BDV programs include both CPU and memory diagnostics as well as bootstrap programs for loading memory from a variety of LSI-11-compatible peripherals. Paragraphs 2.2.4.1 and 11.4 present specific information on the operation of the BDV ROM programs.

Alternatively, users may install ROMs or EPROMs containing programs of their choice in the ROM sockets. In such case the features of the BDV ROM programs would no longer be applicable unless specifically included in the new ROMs.

11.2 BOOTSTRAP AND DIAGNOSTIC REGISTERS

The bootstrap and diagnostic logic contains three hardware registers that are software-addressable. (One of the registers is a dual-purpose, functioning as the configuration register when read and the display register when written.) These registers are assigned individual addresses that cannot be changed or modified. The designations and addresses of these registers are listed in Table 11-1 The registers and associated logic are described in the following paragraphs.

These three registers, along with the line clock register and the ROM addresses, can be disabled by inserting a jumper from J10 to J15.

Table 11-1 Register Address Assignments

Register	Read/ Write	Bit Size	Address
Page Control	W	12	17777520
Read/Write	R/W	16	17777522
Maintenance Configuration*	R	8	17777524
Display*	W	4	17777524

*Dual-purpose register.

11.2.1 Page Control Register (PCR) – Address: 17777520

The page control register (PCR) is a write-only register that is both word- and byte-addressable. Only bits <13:8> and <5:0> can be loaded. Whenever the KDF11-BA read-only memory is accessed, either the PCR high byte (bits <13:8>) or the PCR low byte (bits <5:0>) is used for the six most significant bits of the ROM address.

The read-only memory is accessed by bus addresses 17773000 through 17773777. The eight least significant bits (bits <7:0>) of the bus address become the low-order bits of the ROM address. If bus address bit 8 is zero (17773000–17773377), PCR bits <5:0> become the six most significant bits of the ROM address. If bus address bit 8 is one (17773400–17773777), PCR bits <13:8> become the six most significant bits of the ROM address. The format for the page control register is shown in Figure 11-1. This register is cleared by power-up and when the system is rebooted.

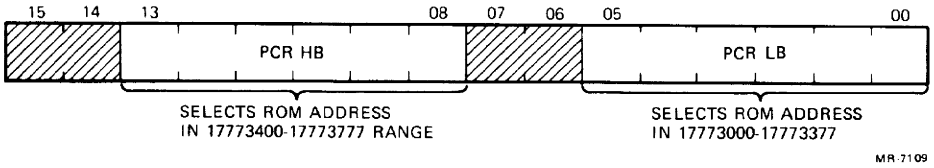


Figure 11-1 Page Control Register Format

11.2.2 Read/Write Maintenance Register (R/W) – Address: 17777522

The read/write maintenance register (R/W) is a 16-bit read/write register that is both word- and byte-addressable. It is used by the ROM diagnostics to test various read/write functions before accessing main memory. This register is cleared by power-up and by system reset.

11.2.3 Configuration and Display Register (CDR) – Address: 17777524

The configuration and display register (CDR) is actually made up of two independent registers that share the same address. The read-only configuration register is accessed when the CDR is read. The write-only display register is loaded by a write transfer to the CDR.

Configuration register bits <15:8> always read as zero; bits <7:0> reflect the status of eight switches on the KDF11-BA module at location E102. The interpretation of these switches is determined by the ROM boot and diagnostic programs. Diagnostic/bootstrap program selection for the KDF11-BA is described in Tables 2-7 and 2-8.

Display register bits <3:0> allow for program control of a diagnostic LED display on the KDF11-BA module. Writing a 0 into one of these bits lights its corresponding LED. Writing a 1 into one of these bits turns its corresponding LED off. Display register bits <15:4> are not used. The display register is cleared (and the four LEDs are lit) by power-up or system restart.

11.3 KDF11-BA ROM MEMORY (ADDRESSES: 17773000–17773777)

The KDF11-BA boot and Diagnostic option has two ROM sockets for either 2K, 4K, or 8K of 16-bit read-only memory.

Addressing ROM Memory

The KDF11-BA ROM memory responds to bus addresses 17773000–17773777. The eight least significant bits of the bus address (bits <7:0>) become the low byte of the ROM address. If bus address bit 8 is zero (17773000–17773377), the PCR bits <5:0> become the six most significant bits of the 14-bit ROM address. If bus address bit 8 is one (17773400–17773777), PCR bits <13:8> become the six most significant bits of the ROM address.

The KDF11-BA includes a pair of 2K × 8 ROMs that only require a 12-bit ROM address. The two most significant ROM address bits (PCR bits <13:12> or <5:4>) must be zero. Figures 11-2 and 11-3 show the formation of the ROM address by the PCR LO byte and the PCR HI byte, respectively.

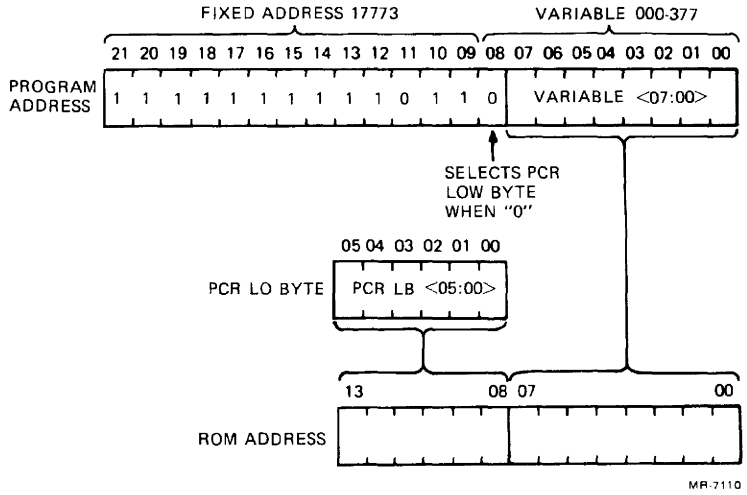


Figure 11-2 ROM Address Format Using PCR LO Byte

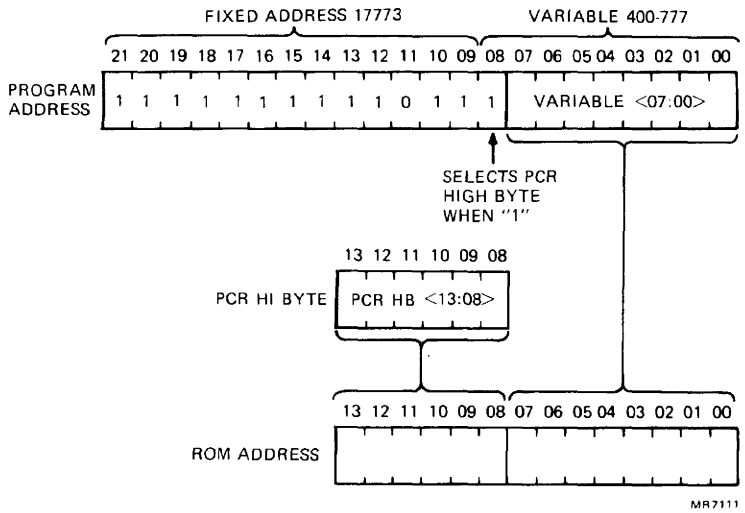


Figure 11-3 ROM Address Format Using PCR HI Byte

11.4 KDF11-BA BOOTSTRAP AND DIAGNOSTIC ROM FUNCTIONALITY

The KDF11-BA ROM programs include both CPU and memory diagnostics as well as bootstrap programs for loading memory from a variety of LSI-11-compatible perpherals. Paragraph 2.2.4.1 describes the use of the configuration register switches in selecting the diagnostic and bootstrap programs. The LED displays and error halts used by the ROM programs are described below.

11.4.1 KDF11-BA LED Display

The KDF11-BA ROM programs use the four red LEDs to indicate which programs and program segments are running. If the program performs an error halt or if it hangs up waiting for data from a peripheral device, these LEDs serve as an error indication.

The four red LEDs present an octal number from 0 (all LEDs off) to 17 (all LEDs on). The most significant LED is separated from the other three LEDs by the green power-on LED. An octal code of 0 indicates that the diagnostics and bootstrap programs have been successfully executed. Codes 1 and 2 are lit during the CPU and memory tests, respectively. Codes 3 and 4 are lit when the ROM programs are typing a message on the console device or waiting for a console input, respectively. Codes 5–12 are lit during various phases of the bootstrap routines. (Code 12 indicates a ROM bootstrap error and should never occur on the KDF11-BA which, unlike the BDV11, does not have sockets for additional ROM boot code.) If the memory diagnostic is disabled, the ROM code still verifies the existence of memory locations 0–6, indicating an error with LED code 13. Code 17 indicates that the ROM programs are unable to begin running, either because the halt switch is on, or because of a hardware failure. Table 11-2 lists the errors indicated by their corresponding LED display pattern.

Table 11-2 KDF11-BA LED Display

Display (Octal)	MSD Bit 3	Bit 2	LSD Bit 1	Bit 0	Type of Error
01	Off	Off	Off	On	CPU test error.
02	Off	Off	On	Off	Memory test error.
03	Off	Off	On	On	Waiting for console terminal transmitter READY flag.
04	Off	On	Off	Off	Waiting for console terminal receiver DONE flag.
05	Off	On	Off	On	Load device status error.
06	Off	On	On	Off	Bootstrap code incorrect;
07	Off	On	On	On	DECnet waiting for response from host.
10	On	Off	Off	Off	DECnet waiting for message completion.
11	On	Off	Off	On	DECnet processing received message.
12	On	Off	On	Off	ROM bootstrap error (not used on KDF11-BA).
13	On	Off	On	On	Special memory test failure on locations 0–6. (Can occur when memory test is disabled.)
17	On	On	On	On	System hung, halt switch on, or not power-up mode 2.

NOTE
The errors indicated above are valid only if the KDF11-BA BDV ROMs (part numbers 23-339E2-00 and 23-340E2-00) are installed in ROM sockets E126 (low byte) and E127 (high byte).

11.4.2 KDF11-BA Error Halts

A failure in a diagnostic test or bootstrap program causes the error to be indicated by the display and an error halt instruction is carried out by the processor. When entering the halt mode, the processor outputs on the console terminal the PC address at the time of the error. The actual error address is one word less than the terminal printout. In halt mode, the processor responds to the console ODT commands and allows the operator to troubleshoot the error. Table 11-3 lists the error halts that can result when the KDF11-BA ROM diagnostics and boostrap programs detect an error condition.

Table 11-3 List of Error Halts

Address of Error*	Display (Octal)	Cause of Error
173036	01	CP1ERR, R0 contains address of error.
173040	05	SLU switch selection incorrect; error in switches.
173046	05	SLU error; CSR address for selected device in error. Check CSR for selected device in floating CSR address area.
173200	12	ROM loader error; checksum on data block.
173232	02	Memory error 2; write address into itself. Test 0–30K words with MMU off if present. R1 = Address in error and expected data R5 = Failing data
173236	01	CP4ERR, R0 points to cause of error.
173240	01	CP3ERR, R0 contains address of error.
173262	02	Memory error 3; odd parity pattern (072527) using byte addressing. Failure in this test usually will indicate problem in byte logic. Test 0–30K words with MMU off if present. R1 = Failing address R4 = Expected data R5 = Failing data
173302	02	Memory error in prememory data test for locations 000–776. R2 = Failing data R3 = Expected data R5 = Failing address (000–776)
173316	02	Memory error; bit 15 set in one of the parity CSRs (772100–772136). Failing memory should have parity error light on. R4 = Address of failing CSR (Contents of failing CSR identifies which 1K-word bank of memory caused error.)

*Contents of R7 after halt.

Table 11-3 List of Error Halts (Cont)

<table>
<tr><th>Address of Error*</th><th>Display (Octal)</th><th>Cause of Error</th></tr>
<tr><td>173364</td><td>12</td><td>ROM loader error; checksum on address block.</td></tr>
<tr><td>173376</td><td>12</td><td>ROM loader error; jump address is odd.</td></tr>
<tr><td>173526</td><td>05</td><td>RL01/RL02 device error.</td></tr>
<tr><td>173652</td><td>05</td><td>RK05 device error.</td></tr>
<tr><td>173654</td><td>01</td><td>Switch mode halt; match was not made with switches.</td></tr>
<tr><td>173660</td><td>02</td><td>Memory error in 0000–2044K words of the 22-bit memory test. This is a common error halt for six different tests.

If R3 = 0, there is an error in test 1–5; R4 determines failing test.
R4 = Expected data
R5 = Failing data

<table>
<tr><th>Contents of R4</th><th>Test No.</th><th>Test Description</th></tr>
<tr><td>20000–27776</td><td>1</td><td>Address test bits 11–0</td></tr>
<tr><td>177777</td><td>2</td><td>Data test</td></tr>
<tr><td>000000</td><td>3</td><td>Data test</td></tr>
<tr><td>072527</td><td>4</td><td>Odd parity pattern test</td></tr>
<tr><td>125125</td><td>5</td><td>Byte addressing test</td></tr>
</table>

For tests 1–5 (R3 = 0), determine 22-bit failing address as follows:

R1 bits 11–00 = failing address bits 11–00
R2 bits 15–06 = failing address bits 21–12

Example:
R2 = 123400 R1 = 027776
R2 = 1234XX R1 = XX7776

Ignore the upper two octal digits of R1 and the lower two octal digits of R2.
Failing 22-bit address = 12347776

Errors in address uniqueness test.
Test checks address bits 21–06. Test 6.
If R3 is not equal to 0, an error is in this test.

R4 = Expected data
R5 = Failing data
R2 = 22-bit failing physical address bits 21–06.
Failing address bits 05–00 are always 0.</td></tr>
</table>

*Contents of R7 after halt.

Table 11-3 List of Error Halts (Cont)

Address of Error*	Display (Octal)	Cause of Error
		Example: R2 = 024566 Failing address = 02456600
173664	02	Memory error in prememory address test for locations 000–776. R2 = Failing data R5 = Failing address and expected data
173670	01	Error CPU Test 9; JSR R3 failed.
173700	01	Error CPU Test 9; JSR PC failed.
173704	05	RX01/RX02 device error.
173714	04	A NO typed in console terminal test.
173736	02	Memory error 1; data test failed.
		Test 0–30K words with MMU off if present. R1 = Failing address R4 = Expected data (either 0 or 177777) R5 = Failing data
173740	01	Error CPU Test 9; RTS return failed.
173742	03/04	Console terminal test; no DONE flag.
173760	05	TU58 error halt.

*Contents of R7 after halt.

CHAPTER 12
LINE FREQUENCY CLOCK

12.1 INTRODUCTION

The line clock logic generates bus request level 6 interrupts to the processor at time intervals determined by the BEVENT L signal. The BEVENT L signal is obtained from the power supply via module pin BR1 at 16 2/3 ms or 20 ms intervals, depending on the line frequency source (60 Hz or 50 Hz, respectively). The line clock logic is shown in Figure 5-11.

Recognition of the BEVENT L signal is typically enabled and disabled under program control using bit 6 of the line clock status register (LKS). When the line clock register is disabled, or if clock interrupts are to be always enabled, recognition of BEVENT L is held enabled by inserting the jumper from J10 to J11.

12.2 LINE CLOCK STATUS REGISTER (LKS) (ADDRESS: 17777546)

The line clock status register (LKS) contains the read/write line clock interrupt enable bit (6), which enables and disables recognition of the BEVENT L line. The remaining bits are not used and always read as zero.

Program recognition of this register, along with the boot and diagnostic registers and ROM memory, can be disabled by inserting a jumper from J10 to J15 on the KDF11-BA module. The line clock status register bit assignment is described in Table 12-1.

Table 12-1 Line Clock Status Register Bit Assignment

Bit	Mnemonic	Meaning and Operation
15:07		Unused.
06	LCIE	Line Clock Interrupt Enable – When set, this read/write bit allows the LSI-11 BEVENT line to initiate program interrupt requests. When this bit is clear, line clock interrupts are disabled. LCIE is cleared by power-up and BINIT. LCIE is held set when the LTC ENJ L (J10 to J11) jumper is installed.
05:00		Unused.

12.3 LINE CLOCK OPERATION

When the line clock interrupt bit is set (either under program control or by a jumper from J10 to J11), assertion of BEVENT L generates an interrupt request at level 6. If the current processor priority is 6 or 7, the processor ignores this request. If the priority is 5 or less, the processor traps to a service routine via vector address 100. Memory location 100 must contain the starting address of the service routine; location 102 contains the new processor status word.

Interrupt vector address: 100
Priority level: 6

CHAPTER 13
SERIAL-LINE UNITS

13.1 INTRODUCTION

The two full-duplex asynchronous serial-line units (console serial-line unit and the second serial-line unit) provide the KDF11-BA with an EIA interface that is compatible with RS-232-C and RS-423. The serial-line baud rates are determined by a clock signal from an internal baud rate generator or an external clock signal via connectors J1 and J2. Jumpers are provided to select either the internal clock or the external clock. If the internal clock is jumper-selected, the serial-line baud rates are switch-selectable from 50 to 19.2K baud. The console serial line and the second serial line may operate at different baud rates, but each serial line will transmit and receive data at the same selected rate. The serial lines provide error indicator bits for overrun error, framing error, and parity error.

The console serial-line unit may be configured to respond to a break signal received from the console terminal. Both serial lines interrupt the processor at bus interrupt priority request level 4 (BR4).

The character format for each of the serial-line units is selected by wirewrap jumpers. The format may consist of seven or eight data bits, one or two stop bits, parity or no parity, and even or odd parity. The wirewrap jumper configuration and baud rate switch configuration for the serial lines are described in Chapter 2.

The console serial-line unit is connected to the console terminal via connector J1. The second serial-line unit is connected to a line printer, the TU58 cassette tape, or an additional terminal via connector J2. A block diagram of the serial-line units is shown in Figure 5-12.

13.2 SERIAL-LINE UNIT REGISTERS

The program communicates with and transfers data to and from the external peripheral devices via four registers associated with each serial line. Two of the registers (RCSR and TCSR) contain control/status information for receiver and transmitter operation. The other two registers (RBUF and TBUF) contain data received from and data to be transmitted to the peripheral device. The addresses assigned to the console and second serial-line registers are listed in Table 13-1.

Register Bit Assignments

The console and second serial-line registers have the same bit assignments with the exception of bit 0 of the TCSR. Bit 0 is used as a transmit break bit (TX BRK) in the second serial-line register (TCSR2) and it is unused in the console serial-line register (TCSR1).

The bit formats for the registers are shown in Figure 13-1. The register bit assignments are described in Tables 13-2 through 13-5.

Table 13-1 Serial-Line Register Addresses

Console Line Register	Address*	Second Serial Line Register	Address	
RCSR1	17777560	RCSR2	17776500**	17776540***
RBUF1	17777562	RBUF2	17776502	17776542
TCSR1	17777564	TCSR2	17776504	17776544
TBUF1	17777566	TBUF2	17776506	17776546

*DL1 DISJ L (J14) must be ungrounded.
**DL2 DISJ L (J13) and DL2 ADRJ L (J12) must be ungrounded.
***DL2 DISJ L (J13) must be ungrounded and DL2 ADRJ L (J12) must be grounded.

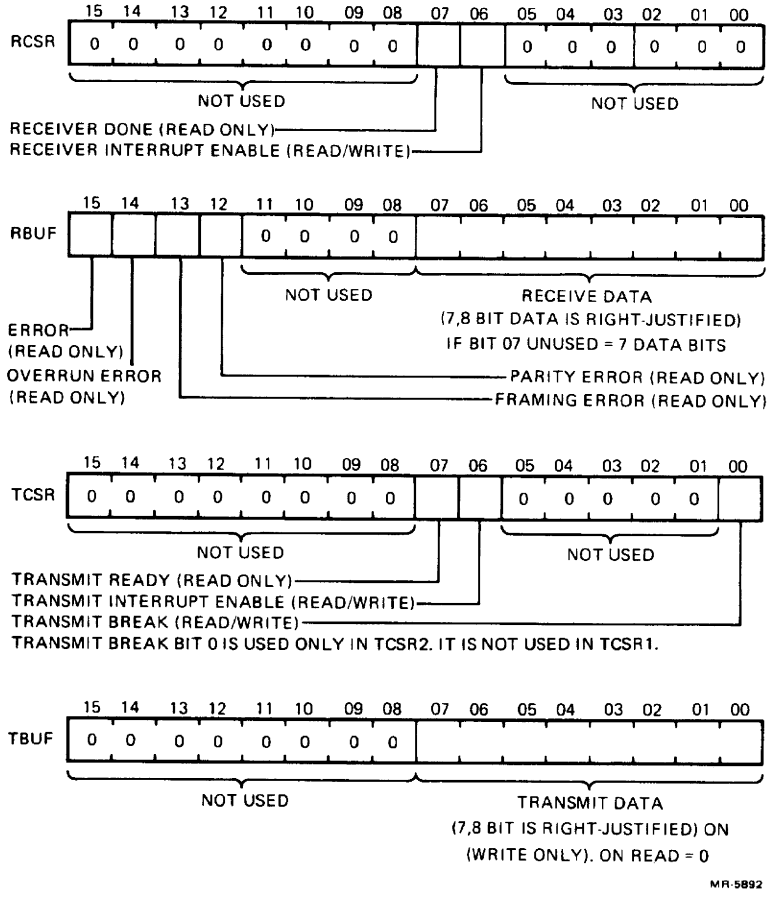


Figure 13-1 Serial-Line Register Formats

Table 13-2 RCSR1 and RCSR2 Bit Assignments

Bits	Mnemonic	Description
15–08		Unused. Read as zeros.
07	RX DONE	Receiver Done. This read-only bit is set when an entire character has been received and is ready to be read from the RBUF Register. This bit is automatically cleared when RBUF is read. It is also cleared by power-up and BUS INIT.
06	RX IE	Receiver Interrupt Enable. This read/write bit is cleared by power-up and BUS INIT. If both RCVR DONE and RCVR INT ENB are set, a program interrupt is requested.
05–00		Unused. Read as zeros.

Table 13-3 RBUF1 and RBUF2 Bit Assignments

Bits	Mnemonic	Description
15	ERR	Error. This read-only bit is set if any RBUF bit (14–12) is set. ERR is clear if all RBUF bits (14–12) are clear. This bit cannot generate a program interrupt.
14	OVR ERR	Overrun Error. This read-only bit is set if a previously received character was not read before being overwritten by the present character.
13	FRM ERR	Framing error. This read-only bit is set if the present character had no valid stop bit. Also used to detect a break condition.
12	PAR ERR	Parity Error. This read-only bit is set if received parity does not agree with expected parity. Always 0 if no parity is selected. NOTE **Error conditions remain in effect until the next character is received, at which point, the error bits are updated. The error bits are cleared by power-up and BUS INIT.**
11–08		Unused. Read as zeros.
07–00		Received Data Bits. These read-only bits contained the last received character. If less than eight bits are selected, the character will be right-justified with the most significant bit(s) reading zero.

Table 13-4 TCSR1 and TCSR2 Bit Assignments

Bits	Mnemonic	Description
15–08		Unused. Read as zeros.
07	TX RDY	Transmitter Ready. This read-only bit is cleared when TBUF is loaded and is set when TBUF can receive another character. XMT RDY is set by power-up and by BUS INIT.
06	TX IE	Transmitter Interrupt Enable. This read/write bit is cleared by power-up and BUS INIT. If both XMT RDY and XMT INT ENB are set, a program interrupt is requested.
02–01		Unused. Read as zeros.
00	TX BRK	Break. When set, this read/write bit transmits a continuous space. This bit is cleared by power-up and SYSTEM INIT. This bit is used only in TCSR2; it is unused in TCSR1.

Table 13-5 TBUF1 and TBUF2 Bit Assignments

Bits	Mnemonic	Description
15–08		Unused. Read as zeros.
07–00	TBUF	TBUF bits 07–00 are write-only bits used to load the transmitted character. If less than eight bits are selected, the character must be right-justified.

13.3 INTERRUPT VECTORS AND INTERRUPT PRIORITY

Two interrupt vectors are provided for the console SLU: one for the SLU transmitter and the other for the SLU receiver. Four interrupt vectors are provided for the second SLU, but only two may be used at any given time. The two vectors that are used by the second SLU depend on the DL2 ADRJ L (J12) jumper configuration. Table 13-6 lists the vectors provided for the console and second serial-line units. The interrupt priority for both SLUs is BR4.

13.4 CONSOLE SLU BREAK RESPONSE

The KDF11-BA console serial-line unit may be configured either to perform a halt operation or to have no response when a break condition is received. A halt operation will cause the processor to halt and enter the on-line debugging technique (ODT) microcode.

If the console SLU is disabled (J14 connected to J10), the halt-on-break feature must also be disabled. The halt-on-break feature is disabled by removing the jumper between J3 and J4 and connecting a jumper between J4 and J5.

13.5 SERIAL-LINE I/O SIGNALS

The two SLUs' input/output signals interface to the console terminal and peripheral device via two connectors (J1 and J2). The connector pin functions for both SLUs are identical and are described in Table 13-7. The 10 pins on each connector (Digital part No. 12-13506-04) are arranged in two rows with five pins in each row.

Table 13-6 Console and Second SLU Interrupt Vectors

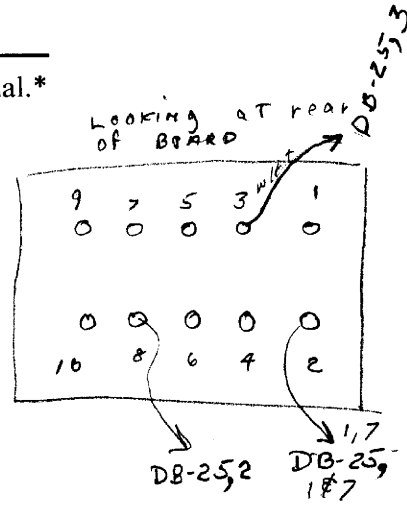
Console SLU		Second SLU*	
Receiver	Transmitter	Receiver	Transmitter
060	064	300**	304
		304***	344

*DL2 DISJ L (J13) must be ungrounded to enable the Second SLU.
**DL2 ADRJ L (12) must be ungrounded.
***DL2 ADRJ L (J12) must be grounded to J10.

Table 13-7 SLU Connector Pin Functions

Pin	Signal	Function
1	EXT CLK	Input for optional external clock signal.*
2	Ground	
3	XMIT+	Transmitter output.
4	Ground	
5	Ground	
6	NC	Key; pin not provided.
7	RCV−	Receiver input (most negative).
8	RCV+	Receiver input (most positive).
9	Ground	
10	+12 V	Power for external options; fused at 1 A.

*Paragraph 2.2.7 describes the internal/external SLU clock jumpers.



DB-25
Jumper
8 → 20
7 → 1
4 → 5 → 6

CHAPTER 14
COMMERCIAL INSTRUCTION SET

14.1 INTRODUCTION

The commercial instruction set (CIS) provided by the KEF11-BB option is a series of instructions for manipulating byte strings in order to improve COBOL performance, text editing, and word processing capabilities. CIS includes instructions that operate on character strings and on decimal numbers. Each generic type of instruction is provided in two forms. The essential difference between the two forms is the manner in which operands are delivered to the instruction. The first form is the "register" form, where operands are implicitly obtained from the general registers. The second form is the "in-line" form, where operands or word address pointers to operands follow the op-code word in the instruction stream. The mnemonic for the in-line form is the mnemonic for the register form suffixed with the letter "I." The condition codes are set identically for both forms. The in-line forms minimize register modifications.

The CIS also includes commercial load descriptor instructions used for instruction control. These instructions augment the character and form instructions by efficiently loading operands (string descriptors) into the general registers. Descriptors consist of the starting address of the string and the length of the string. Two forms of instructions are provided. The first form of the instruction loads two string descriptors into the general registers. The second form loads three string descriptors into the general registers.

The instructions in the PDP-11 CIS consist of the following extended instruction groups.

07602X	Commercial Load 2 Descriptors
07603X	Character String Move
07604X	Character String Search
07605X	Numeric String
07606X	Commercial Load 3 Descriptors
07607X	Packed String
07613X	Character String Move (in-line)
07614X	Character String Search (in-line)
07615X	Numeric String (in-line)
07617X	Packed String (in-line)

14.2 UNPREDICTABLE CONDITIONS

A result of an instruction or the effect of an instruction can be "unpredictable." Unpredictable describes an outcome that is indeterminate and nonrepeatable. When the results of an instruction are unpredictable, the condition codes and destination operands (but not their descriptors) will contain unpredictable values; destinations may not even contain valid results. When the effect of an instruction is unpredictable, the entire user or process state, and not only the portion typically used by the instruction, will be unpredictable. In a machine with multiple modes and address spaces, and unpredictable operation in a less privileged mode will not affect the state of a more privileged mode, nor will it result in accesses to memory from user mode that are outside the mapped limits of the user's program.

Note that architectural constraints exist on unpredictable effects. In particular, an unpredictable effect that manifests itself as a trap must meet all the requirements for the particular trap.

14.3 CHARACTER DATA TYPES

There are three different character data types.

1. A "character," a single byte with an abbreviated string of length 1.
2. A "character string," a contiguous group of bytes in memory.
3. A "character set," a subset of the 256 possible characters that can be encoded in a byte.

14.3.1 Character

A character is an 8-bit byte, as shown in Figure 14-1.

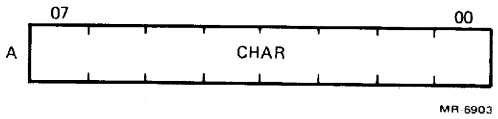


Figure 14-1 8-Bit Byte Character

A character is used as an operand by CIS instructions. When one appears in a general register, it is in the low-order half; the high-order half of the register must be zero. When it appears in the instruction stream, the character is in the low-order half of a word; the high-order half of the word must be zero. If the high-order half of a word that contains a character is nonzero, the effect of the instruction that uses it will be unpredictable.

14.3.2 Character String

A character string is a contiguous sequence of bytes in memory that begins and ends on a byte boundary. It is addressed by its most significant character (lowest address). The highest address is the least significant character. A character string is specified by a 2-word descriptor with the attributes of length and lowest address. The length is an unsigned binary integer that represents the number of characters in the string and may range from 0 to 65,535. A character string with zero length is said to be vacant; its address is ignored. A character string with nonzero length is said to be occupied.

The character string descriptor is used as an operand by CIS instructions. The descriptor appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. Figure 14-2 shows the descriptor for a character string of length "n" starting at address "A" in memory.

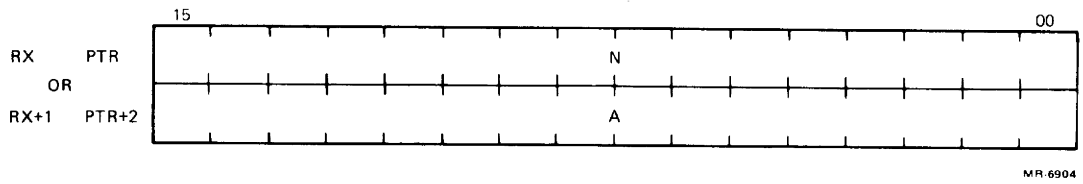


Figure 14-2 Character String Descriptor

Figure 14-3 shows the character string as it would be placed in memory.

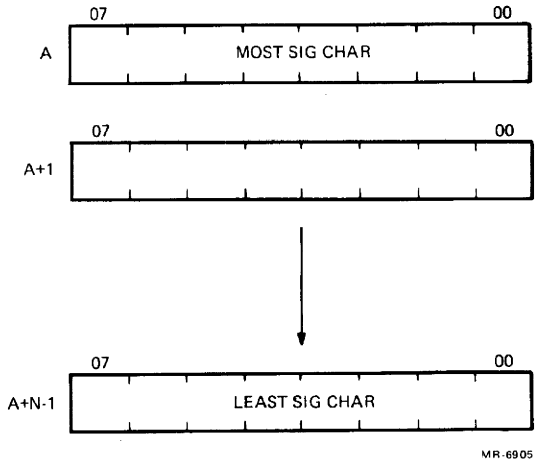


Figure 14-3 Character String in Memory

14.3.3 Character Set

A character set is a subset of the 256 possible characters that can be encoded in a byte. It is specified by a descriptor that consists of the address of a 256-byte table and an 8-bit mask. The address is of byte 0 in the table. Each byte in the table specifies up to eight orthogonal character subsets of which the corresponding character is a member. The mask selects which combinations of these orthogonal subsets comprise the entire character set. In effect, each bit in the mask corresponds to one of eight orthogonal subsets that may be encoded by the table. The mask specifies the union of the selected subsets into the character set. Typical sets would be: uppercase, lowercase, nonzero digits, end-of-line, etc.

Operationally, a character (char) is considered to be in the character set if the evaluation of (M[table.adr+char].mask) is not equal to zero. The character is not in the character set if the evaluation is zero. Each byte in the table indicates of which combination of up to eight orthogonal character subsets (i.e., one for each of the eight bit vectors: 00000001(2), 00000010(2), 00000100(2), 00001000(2), 00010000(2), 00100000(2), 01000000(2) and 10000000(2)) the corresponding character is a member. The mask specifies which union of the eight orthogonal character subsets comprises the total character set. For example, if (a) the 8-bit vector 00000001(2) appearing in the table corresponds to the character subset of all uppercase alphabetic characters, (b) 00000010(2) appearing in the table corresponds to the character subset of all lowercase alphabetic characters, and (c) 00000100(2) appearing in the table corresponds to the decimal digits, then using the mask 00000011(2) with this table specifies the character set of all alphabetic characters, and using the mask 00000111(2) specifies the character set of all alphanumeric characters.

The character set descriptor is used as an operand by CIS instructions. It appears in two consecutive general registers, or in two consecutive words of memory pointed to by a word in the instruction stream. If the high-order half of the first descriptor word is nonnzero, the effect of an instruction that uses a character set will be unpredictable. The character set format is shown in Figure 14-4.

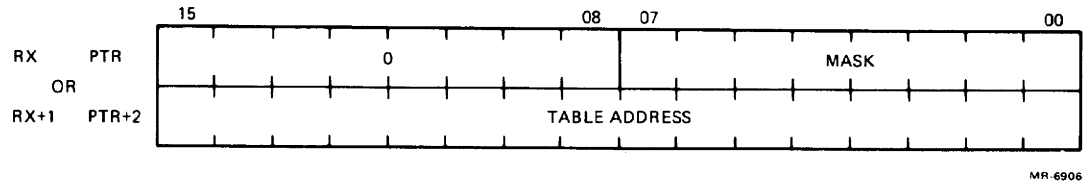


Figure 14-4 Character Set Format

14.3.4 Character String Instructions

Character string operations conveniently provide most of the common, as well as time-consuming, functions that are encountered in commercial data and text processing applications. Instructions are provided to move and to search character strings.

The character string move instructions use character string descriptors as operands. These descriptors specify a source and a destination character string. The contents of the source are moved to the destination with alignment at either the most significant character, as in MOVC(I) and MOVTC(I), or the least significant character, as in MOVRC(I). If the source is longer than the destination, characters are truncated from the side opposite that of the alignment; if the destination is longer than the source, the destination is completed with fill characters on the side opposite that of the alignment. The MOVTC(I) instructions move a translated source string to a destination string. The character string move instructions are summarized below.

Character String Move Instructions

MOVC(I)	Move character
MOVRC(I)	Move reverse-justified character
MOVTC(I)	Move translated character

The character string search instructions use a character string descriptor as one operand. The other operand is either a character, a character string descriptor, or a character set descriptor. These instructions are used to examine the source string to find the presence or absence of characters. The source string is processed from most significant to least significant character. The character string search instructions are summarized below.

Character String Search Instructions

LOCC(I)	Locate character
SKPC(I)	Skip character
SCANC(I)	Scan character
SPANC(I)	Span character
CMPC(I)	Compare character
MATC(I)	Match character

Conceptually, the character string search instructions may be divided into three classes.

1. Character String Searches – CMPC(I) compares two character strings. The condition codes are set according to the comparison of the corresponding most significant unequal characters. MATC(I) finds an object string within a source string. This is the "in-string" function that languages and text processing systems provide.

2. Character Searches – LOCC(I) finds the first occurrence of a given character in a string. SKPC(I) skips to the first nonoccurrence of a given character in a string.

3. Character Set Searches – In these instructions, a string is examined until a member of a character set is either found, as in SCANC(I), or found, as in SPANC(I). This aids the search for one of several delimiters, such as the slash (/), comma (,), CR, LF, FF, etc., or the passing of combinations of characters such as blanks, tabs, etc. LOCC(I) and SKPC(I) are optimizations of SCANC(I) and SPANC(I), in which the set consists of a single character.

The setting of condition codes reflects the results of the character string operations. For character string moves, the condition codes indicate whether the source and destination strings were of equal length, the source was shorter than the destination so that fill characters were used, or the source was longer than the destination so that characters were truncated. This is accomplished by setting the condition codes on the result of an arithmetic comparison of the initial source and destination lengths. For CMPC(I) the condition codes are the result of arithmetically comparing the most significant corresponding pair of unequal characters. For the other search instructions they show whether or not the operand strings were completely examined.

The condition codes for some character string search instructions may be interpreted according to the notion of success or failure. Success is the accomplishment of the instruction's task; failure is the inability to accomplish the task. Since the condition codes are set based on the results of the instruction, there is an indirect correspondence between these settings and success or failure. This correspondence is invariant within an instruction, but it is not the same for all search instructions. Therefore, different branch instructions must be used to test the operation of each instruction. The branch instructions are summarized below.

Instruction	Success	Failure
LOCC(I)	BNE	BEQ
SCANC(I)	BNE	BEQ
CMPC(I)	BEQ	BNE
MATC(I)	BNE	BEQ

The "register form" of character string instructions implicitly finds operands in the general registers. These operands include character, character string descriptor, character set descriptor, and translation table address. If an instruction does not use a register, its contents will be undisturbed. R0–R1 generally contain a source character string descriptor; R2–R3 generally contain a second source character string descriptor, or the destination string descriptor. The low-order half of R4 is used as an explicit character. R4–R5 are used to contain a character set descriptor. R5 contains the starting address of a 256-byte table, which is used for character translation.

When move instructions terminate, R0 contains the number of unmoved source characters, and R1, R2, and R3 are cleared. For search instructions, the registers are updated to represent descriptors for the resulting strings.

The "in-line" form of character string instructions finds operands, or pointers to operands, in the instruction stream immediately following the op-code word. Operands that appear directly in the instruction stream include characters and translation table addresses. Descriptors are represented in the in-

struction stream by a single word whose contents are interpreted as a word address pointer to the 2-word descriptor. These descriptors specify character strings and character sets. Some instructions return a character string descriptor in R0–R1.

In general, all character string instructions are unaffected by the overlapping of source or destination strings. The result of the move instructions is equivalent to having read the entire source string before storing characters in the destination. If the destination string of the MOVTC(I) instructions overlaps the translation table, the characters stored in the destination string will be unpredictable.

14.4 DECIMAL STRING DATA TYPES

Two classes of decimal string data types – numeric strings and packed strings – are defined. Both have similar arithmetic and operational properties; they differ primarily in their representation of signs and the placement of their digits in memory.

The numeric string data types are signed zoned, unsigned zoned, trailing overpunched, leading overpunched, trailing separate, and leading separate. The packed string data types are signed packed and unsigned packed. Instructions that operate on numeric strings permit each numeric string operand to be separately specified; similarly, packed string instructions permit each packed string operand to be separately specified. Thus, within each of the two classes of decimal strings, the operands of an instruction may be of any data type within the appropriate class.

Decimal strings exist in memory as contiguous bytes that begin and end on a byte boundary. They represent numbers consisting of 0 to 31_{10} digits, in either sign-magnitude or absolute-value form. Sign-magnitude strings (signed) may be positive or negative; absolute-value strings (unsigned) represent the absolute value of the magnitude. Decimal numbers are whole integer values with an implied decimal radix point immediately after the least significant digit; they may be extended conceptually with the addition of 0s before the most significant digit.

A 4-bit binary coded decimal representation is used for most digits in decimal strings. A 4-bit half byte is called a "nibble" and may be used to contain a binary bit pattern that represents the value of a decimal digit. The following shows the binary nibble contents associated with each decimal digit.

Digit	Nibble
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Each decimal string data type may have several representations. These representations permit a certain latitude when accepting source operands. Decimal string data types have a "preferred" representation, which is a valid source representation used to construct the destination string. Also, "alternate" representations are provided for some decimal data types when accepting source operands.

Decimal strings used as source operands are not checked for validity. Instructions will produce unpredictable results if a decimal string used as a source operand contains invalid digit encoding, an invalid sign designator, or, in the case of overpunched numbers, invalid sign/digit encoding. When used as a course, decimal strings with zero magnitude are unique, regardless of sign. Thus, positive zero and negative zero have identical interpretations.

Conceptually, decimal string instructions first determine the correct result, then store the decimal string representation of the correct result in the destination string. A result of zero magnitude is considered to be positively signed. If the destination string can contain more digits than are significant in the result, the excess most significant destination string digits have zero digits stored in them. If the destination string cannot contain all significant digits of the result, the excess most significant result digits are not stored; the instruction will indicate decimal overflow. Note that negative zero is stored in the destination string as a side effect of decimal overflow where the sign of the result is negative and the destination is not large enough to contain any nonzero digits of the result.

If the destination string has zero length, no resulting digits will be stored. The sign of the result will be stored in separate and packed strings, but not in zoned and overpunched strings. Decimal overflow will indicate a nonzero result.

14.4.1 Decimal String Descriptors

Decimal strings are represented by a 2-word descriptor. The descriptor contains the length, data type, and address of the string. It appears in two consecutive general registers (in the register form of instructions), or in two consecutive words in memory pointed to by a word in the instruction stream (in the inline form of instructions). The unused bits are reserved by the architecture and must be 0s. The effect of an instruction using a descriptor will be unpredictable if any nonzero reserved field in the descriptor contains nonzero values or a reserved data type encoding is used. The design of the numeric and packed string descriptors are identical:

First Word

(L) length <4:0> — Number of digits specified as an unsigned binary integer.

(T) data type <14:12> — Specifies which decimal data type representation is used.

Second Word

(A) address <15:0> — Specifies the address of the byte that contains the most significant digit of the decimal string.

The descriptor format for a decimal string of data type T whose length is L and whose most significant digit is at address A is shown in Figure 14-5.

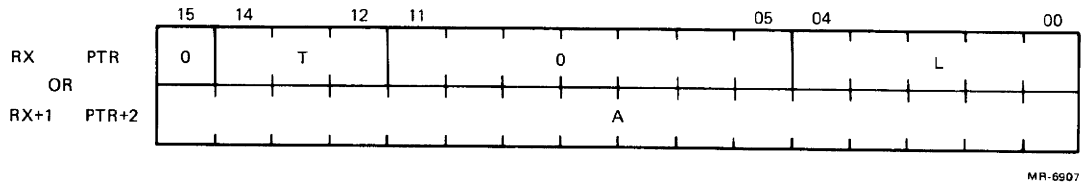


Figure 14-5 Decimal String Descriptor

The encodings (in binary) for the "numeric" string data type field are

000	Signed zoned
001	Unsigned zoned
010	Trailing overpunched
011	Leading overpunched
100	Trailing separate
101	Leading separate
110	Reserved for use by DIGITAL
111	Reserved for use by DIGITAL

The encodings (in binary) for the packed string data type field are

000	Reserved for use by DIGITAL
001	Reserved for use by DIGITAL
010	Reserved for use by DIGITAL
011	Reserved for use by DIGITAL
100	Reserved for use by DIGITAL
101	Reserved for use by DIGITAL
110	Signed packed
111	Unsigned packed

14.4.2 Packed Strings

Packed strings can store two decimal digits in each byte. The least significant (highest addressed) byte contains the sign of the number in bits <3:0> and the least significant digit in bits <7:4>.

Signed Packed Strings – The preferred positive sign designator is 1100_2; alternate positive sign designators are 1010_2, 1110_2 and 1111_2. The preferred negative sign designator is 1101_2; the alternate negative sign designator is 1011_2. Source strings will properly accept both the preferred and alternate designators; destination strings will be stored with the preferred designator.

Unsigned Packed Strings – The unsigned signed designator is 1111_2.

Packed Sign Nibble

Sign Nibble	Preferred Designator	Alternate Designator(s)
Positive	1100_2	1010_2, 1110_2, 1111_2
Negative	1101_2	1011_2
Unsigned	1111_2	

For other than the least significant byte, bytes contain two consecutive digits – the one of lower significance in bits <3:0> and the one of higher significance in bits <7:4>. For numbers whose length is odd, the most significant digit is in bits <7:4> of the lowest addressed byte. Numbers with an even length have their most signicant digit in bits <3:0> of the lowest addressed byte; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000 for destination strings. Numbers with a length of one occupy a single byte and contain their digit in bits <7:4>. The number of bytes that represents a packed string is [length/2] + 1 (integer division where the fractional portion of the quotient is discarded).

The format for a packed string with an odd number of digits is shown in Figure 14-6, and the format for a packed string with an even number of digits is shown in Figure 14-7.

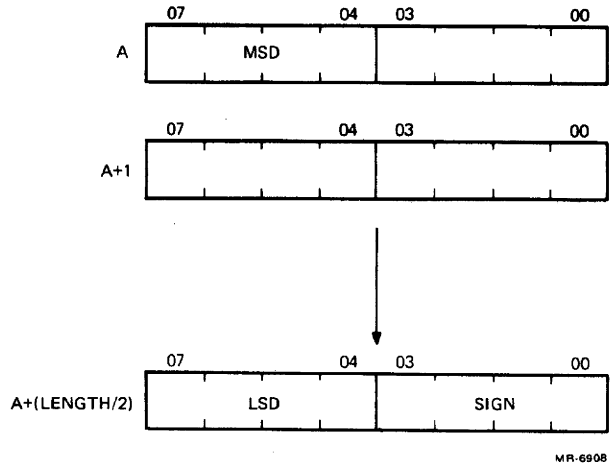


Figure 14-6 Packed String – Odd Digits

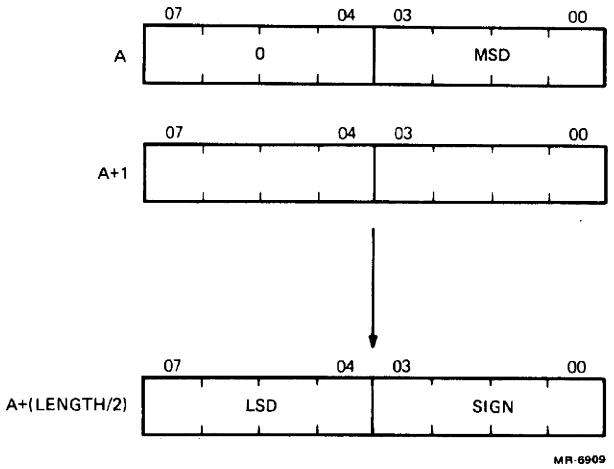


Figure 14-7 Packed String – Even Digits

A zero-length packed string occupies a single byte of storage; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000 for destination strings. Bits <3:0> must be a valid sign for source strings, and are used to store the sign of the result for destination strings. When used as a source, zero-length strings represent operands with zero magnitude. When used as a destination, they can only reflect a result of zero magnitude without indicating overflow. The format for a packed string with a zero length is shown in Figure 14-8.

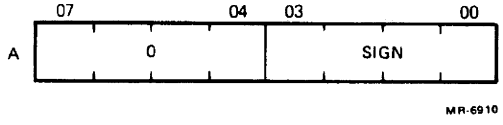


Figure 14-8 Packed String – Zero Length

The following are the characteristics of a valid string.

1. A length of 0 to 31_{10} digits.
2. Every digit nibble is in the range 0000 to 1001_2.
3. For even-length sources, bits <7:4> of the lowest addressed byte are 0000.
4. Signed packed strings – sign nibble is either 1010_2, 1011_2, 1100_2, 1101_2, 1110_2 or 1111_2.
5. Unsigned packed strings – sign nibble is 1111_2.

14.4.3 Zoned Strings

Zoned strings represent one decimal digit in each byte. Each byte is divided into two portions – the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low-order nibble contains the value of the corresponding decimal digit. Zoned strings may be either signed or unsigned. The format for zoned strings is shown in Figure 14-9.

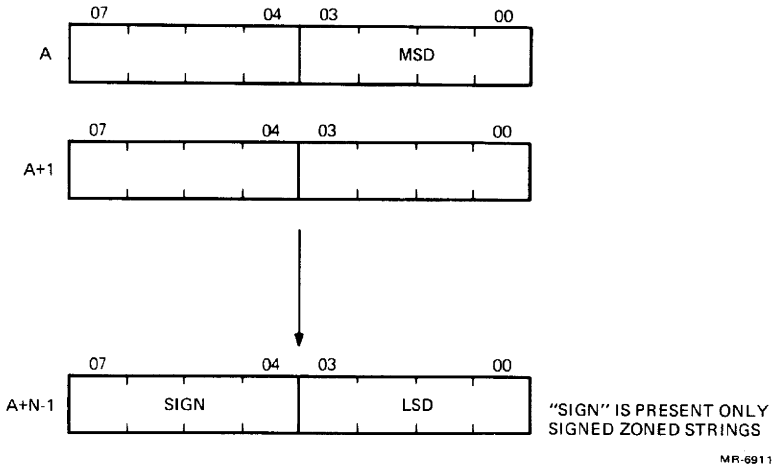


Figure 14-9 Zoned Strings

Signed Zoned Strings – When used as a source string, the high-order nibble of the least significant byte contains the sign of the number; the high-order nibbles of all other bytes are ignored. Destination strings are stored with the sign in the high-order nibble of the least significant byte, and 0011_2 in the high-order nibble of all other bytes. In the high-order nibble 0011_2 corresponds to the ASCII encoding for numeric digits. The positive sign designator is 0011_2; the negative sign designator is 0111_2.

Unsigned Zoned Strings – When used as a source string, the high-order nibbles of all bytes are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all bytes. The number of bytes needed to contain a zoned string is identical to the length of the decimal number.

A zero-length zoned string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero-length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a nonzero result will be indicated by the setting of overflow. The following are the characteristics of a valid zoned string.

1. A length of 0 to 31_{10} digits.

2. The low-order nibbles of each byte are in the range 0000 to 1001_2.

3. Signed zoned strings – The high-order nibble of the least significant byte is either 0011_2 or 0111_2.

14.4.4 Overpunched Strings

Overpunched strings represent one decimal digit in each byte. Trailing overpunched strings combine the encoding of the sign and the least significant digit; leading overpunched strings combine the encoding of the sign and the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions – the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low-order nibble contains the value of the corresponding decimal digit. When used as a source string, the high-order nibble of all bytes that do not contain the sign are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all bytes that do not contain the sign. In the high-order nibble 0011_2 corresponds to the ASCII encoding for numeric digits.

The list below shows the sign of the decimal string and the value of the digit encoded in the sign byte. Source strings will properly accept both the preferred and alternate designators; destination strings will store the preferred designator. The preferred designators correspond to the ASCII graphics A to R, and the open and close brace ({ and }). The alternate designators correspond to the ASCII graphics 0 to 9, the open and close brackets ([and]), a question mark (?), exclamation point (!), and colon (:).

Overpunched Sign/Digit Byte

Overpunched Sign/Digit	Preferred Designator	Alternate Designator(s)
+0	01111011_2	00110000_2, 01011011_2, 00111111_2
+1	01000001_2	00110001_2
+2	01000010_2	00110010_2
+3	01000011_2	00110011_2
+4	01000100_2	00110100_2
+5	01000101_2	00110101_2
+6	01000110_2	00110110_2
+7	01000111_2	00110111_2
+8	01001000_2	00111000_2
+9	01001001_2	00111001_2
−0	01111101_2	01011101_2, 00100001_2, 00111010_2
−1	01001010_2	
−2	01001011_2	
−3	01001100_2	
−4	01001101_2	
−5	01001110_2	
−6	01001111_2	
−7	01010000_2	
−8	01010001_2	
−9	01010010_2	

The number of bytes needed to contain an overpunched string is identical to the length of the decimal number. The format for a trailing overpunched string is shown in Figure 14-10. The leading overpunched string is shown in Figure 14-11.

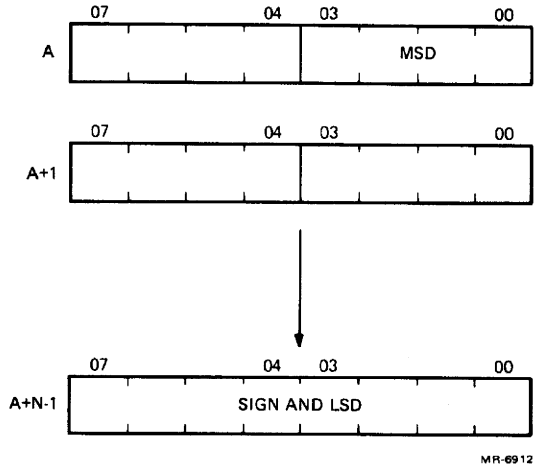


Figure 14-10 Trailing Overpunched String

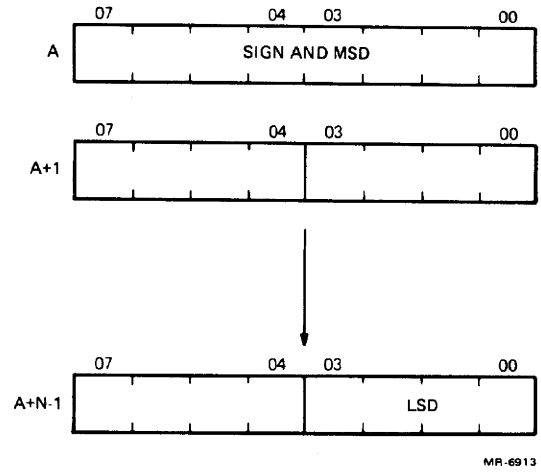


Figure 14-11 Leading Overpunched String

A zero-length overpunched string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero-length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a nonzero result will be indicated by the setting of overflow. The following are the characteristics of a valid overpunched string.

1. A length of 0 to 31_{10} digits.

2. The low-order nibble of each digit byte is in the range 0000 to 1001_2.

3. The encoded sign/digit byte contains values from the above list of preferred and alternate overpunched sign/digit values.

14.4.5 Separate Strings

Separate strings represent one decimal digit in each byte. Trailing separate strings encode the sign in the byte immediately after the least significant digit; leading separate strings encode the sign in the byte immediately before the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions – the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low-order nibble contains the value of the corresponding decimal digit.

When used as a source string, the high-order nibbles of all digit bytes are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all digit bytes. In the high-order nibble 0011_2 corresponds to the ASCII encoding for numeric digits. The preferred positive sign designator is 00101011_2 and the alternate positive sign designator is 00100000_2. The negative sign designator is 00101101_2. These designators correspond to the ASCII encoding for the plus sign (+), a space, and minus sign (−).

Separate Sign Byte

Sign Byte	Preferred Designator	Alternate Designator
Positive	00101011_2	00100000_2
Negative	00101101_2	

The number of bytes needed to contain a leading or trailing separate string is identical to the length + 1. The format for a trailing separate string is shown in Figure 14-12. The leading separate string is shown in Figure 14-13.

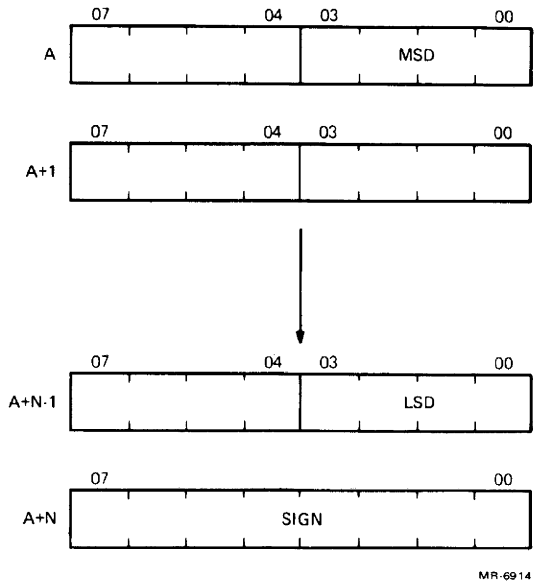


Figure 14-12 Trailing Separate String

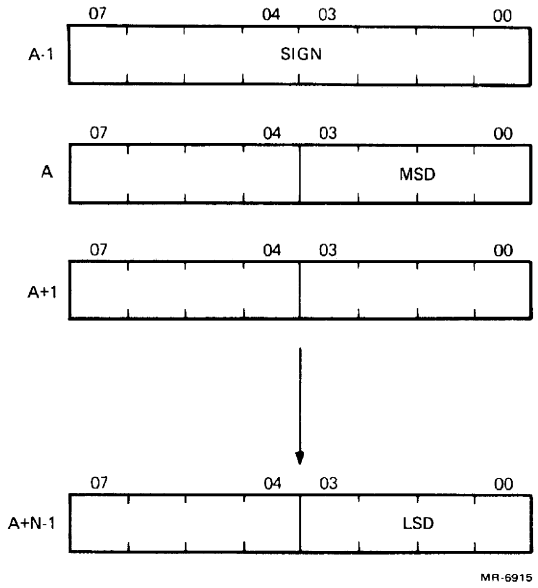


Figure 14-13 Leading Separate String

A zero-length separate string occupies a single byte of memory that contains the sign of the string. When used as a source, zero-length strings provide operands with zero magnitude; when used as a destination, they can only reflect a result of zero magnitude without indicating overflow. The sign of the result is stored.

The format for a zero-length trailing separate string is shown in Figure 14-14. The zero-length leading separate string is shown in Figure 14-15.

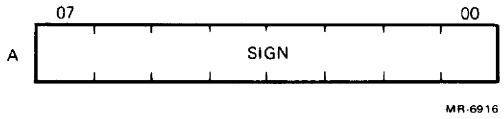


Figure 14-14 Zero-Length Trailing Separate String

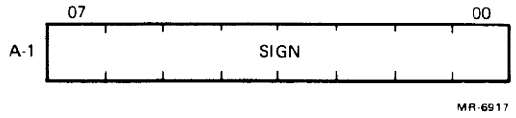


Figure 14-15 Zero-Length Leading Separate String

The following are the characteristics of a valid separate string.

1. A length of 0 to 31_{10} digits.
2. The low-order nibble of each digit byte is in the range 0000 to 1001_2.
3. The sign byte is either 00100000_2, 00101011_2 or 00101101_2.

14.4.6 Long Integer

Long integers are 32-bit binary 2's complement numbers organized as two words in consecutive registers or in memory – no descriptor is used. One word contains the high-order 15 bits. The sign is in bit <15>; bit <14> is the most significant. The other word contains the low-order 16 bits with bit <0> the least significant. The range of numbers that can be represented is −2,147,483,648 to +2,147,483,647.

The register form of decimal convert instructions uses a restricted form of long-integer format with the number in the general register pair R2–R3. The format for the register form of decimal convert instructions is shown in Figure 14-16.

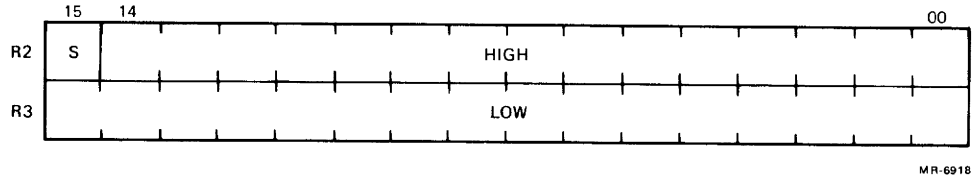


Figure 14-16 Decimal Convert (Register Form)

The in-line form of decimal convert instructions reference the long integer by a word address pointer, which is part of the instruction stream. The format for the in-line form of decimal convert instructions is shown in Figure 14-17.

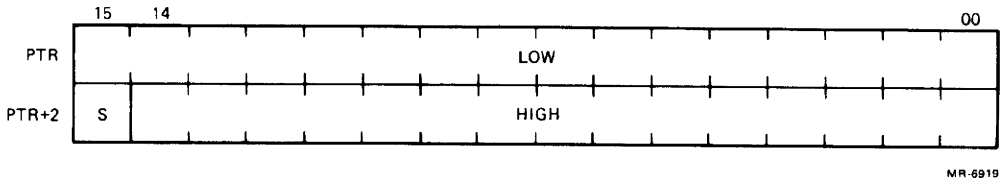


Figure 14-17 Decimal Convert (In-Line Form)

Note that these two representations of long integers differ. There is no single representation of long integers among EAE, EIS, FPP and software. The "register" form was selected to be compatible with EIS; the "in-line" form was selected to be compatible with current standard software usage.

14.4.7 Decimal String Instructions

Decimal string instructions aid in the manipulation of decimal data. Several numeric (byte) and packed decimal data types are supported. Instructions are provided for basic arithmetic operations, as well as for compare, shift, and convert functions.

Each arithmetic, shift and compare instruction operates on a single class of data type. Both numeric and packed string instructions are provided for most operations. Convert instructions have a source operand of one data type and a destination operand of another data type. Decimal string instructions specify to which class each of their decimal string operands belong. The data type supplied as part of each operand's descriptor may be any valid data type of the class. This permits a general mixing of data types within numeric and packed classes. The data types on which an instruction operates are designated by the last letter(s) of the op-code mnemonic. N denotes numeric strings, P denotes packed strings, and L denotes long binary integers.

The arithmetic instructions are ADDN(I), ADDP(I), SUBN(I), SUBP(I), MULP(I) AND DIVP(I). ASHN(I) and ASHP(I) shift a decimal string by a specified number of digit positions (in either direction) with optional rounding, and store the result in the destination string. Thus, they effectively multiply or divide by a power of 10. If the shift count is zero, these shift instructions can be used simply to move decimal strings (destinations are stored with preferred representation). Move negated may be accomplished by using SUBN(I) or SUBP(I). Arithmetic comparison instructions, CMPN(I) and CMPP(I), are provided to examine the relative difference between two decimal strings.

CVTNL(I) and CVTPL(I) convert a decimal string to a long (32-bit) 2's complement integer. CVTLN(I) and CVTLP(I) convert a long integer to a decimal string. CVTNP(I) and CVTPN(I) convert between numeric and packed decimal strings.

The following is a list of the decimal string instructions.

Packed String Instructions

ADDP(I)	Add packed
SUBP(I)	Subtract packed
MULP(I)	Multiply packed
DIVP(I)	Divide packed
ASHP(I)	Arithmetic shift packed
CMPP(I)	Compare packed

Numeric String Instructions

ADDN(I)	Add numeric
SUBN(I)	Subtract numeric
ASHN(I)	Arithmetic shift numeric
CMPN(I)	Compare numeric

Convert Instructions

CVTNL	Convert numeric to long
CVTLN	Convert long to numeric
CVTPL	Convert packed to long
CVTLP	Convert long to packed
CVTNP	Convert numeric to packed
CVTPN	Convert packed to numeric

14.4.8 Condition Codes

For instructions that store a value in a destination string, the N and Z bits reflect the value stored. The N bit indicates a negative destination; the Z bit indicates a destination having zero magnitude. A destination string with zero magnitude is considered to be positive (even if a negative zero was stored as a consequence of decimal overflow). Thus, the setting of N and Z are mutually exclusive.

The V bit indicates whether the destination string accurately represents the result of the instruction. It is also set if division by zero was attempted. If the V bit is set, the destination string will represent the least significant portion of the result (truncated). If the V bit is cleared, the destination represents the true result.

For DIVP(I), C indicates division by zero. Otherwise, C is always cleared. For comparisons using the CMPN(I) and CMPP(I) instructions, the N and Z bits reflect the signed relationship between the source strings. The signed branch instructions can test the result. V and C are cleared.

For instructions that return a long-integer value, N reflects the sign of the 2's complement integer, and Z indicates whether it was zero. V indicates whether the long integer could not contain all significant digits and sign of the result. CVTNL(I) and CVTPL(I) also use C to represent a borrow from a more significant portion of the long binary result. Otherwise, C is cleared.

14.4.9 Operand Delivery

The "register" form of decimal string instructions implicitly finds the operands in the general registers. These operands include decimal string descriptors, long binary integers, and shift descriptor words. If an instruction does not use a register, its contents will be undisturbed. R0–R1 generally contain the first source descriptor, R2–R3 the second source descriptor, and R4–R5 the destination descriptor. ASHN and ASHP use R4 to contain a shift descriptor word. CVTLN, CVTLP, CVTNL and CVTPL use R0–R1 to contain a decimal string descriptor, and R2–R3 for the long integer. When an instruction is completed, the source descriptor registers are cleared.

The "in-line" form of the decimal string instructions finds the operands, or pointers to descriptors, in the instruction stream immediately following the op-code word. Operands that appear directly in the instruction stream are shift descriptor words. Operands that are represented in the instruction stream by a pointer containing the word address of the descriptor are decimal string descriptors and long binary integers. No in-line form of decimal string instructions modifies R0–R6.

14.4.10 Data Overlap

The operation of decimal string instructions is unaffected by an overlap of the source operands, provided that each source operand is a valid representation of the specified data type. The overlap of the destination string and any of the source strings will, in general, produce unpredictable results. However, ADDN(I), ADDP(I), SUBN(I) and SUBP(I) will permit the destination string to overlap either or both source strings only if all corresponding digits of the strings are in coincident bytes in memory. This facilitates 2-address arithmetic.

14.5 COMMERCIAL LOAD DESCRIPTOR INSTRUCTIONS

The commercial load descriptor instructions augment the character and decimal string instructions by efficiently loading the general registers with string descriptors. Two forms of instructions are provided. The first form, the L2Dr instructions, load two string descriptors into the general registers. The first descriptor is loaded into R0–R1 and the second descriptor is loaded into R2–R3. This instruction supports equal-length character string move, equal-length character string compare, character string matching, and decimal string compare. The second form, the L3Dr instructions, take three descriptors. The first is loaded into R0–R1, the second into R2–R3, and the third into R4–R5. The instruction supports 3-address arithmetic. The condition codes are not affected for either form of instruction.

Words containing the addresses of the descriptors (two for L2Dr and three for L3Dr) are in consecutive locations in memory. The descriptor addresses are found by applying the addressing mode @(Rr)+ once for each descriptor. The value of r is encoded as the low-order three bits of the instruction's op-code. If 0 r 5, r can be thought of as the base address of a small table in memory, where each entry in the table contains the address of a descriptor. If r = 6, the instructions effectively pop the addresses of descriptors off the stack. If r = 7, the descriptor addresses are contiguous with the instruction's op-code word.

The string descriptors are two words long. The address of the descriptor is that of the low-order word. It is loaded into the corresponding even register. The high-order word of the descriptor is loaded into the corresponding odd register. Note that although these instructions are described in terms of string descriptors, they are applicable for instances where two consecutive words in memory referenced by a pointer are to be copied into even-odd general register pairs. The following is a list of the commercial load descriptor instructions.

Command	Load 2 descriptors using:
L2D0	@(R0)+
L2D1	@(R1)+
L2D2	@(R2)+
L2D3	@(R3)+
L2D4	@(R4)+
L2D5	@(R5)+
L2D6	@(R6)+
L2D7	@(R7)+
	Load 3 descriptors using:
L3D0	@(R0)+
L3D1	@(R1)+
L3D2	@(R2)+
L3D3	@(R3)+
L3D4	@(R4)+
L3D5	@(R5)+
L3D6	@(R6)+
L3D7	@(R7)+

14.6 INSTRUCTION SUSPENSION

The intent in defining instruction suspendability is to establish a means for providing reasonable interrupt latency and does not presume to endow CIS instructions with an ability to recover from trap conditions from which sequences of basic instructions cannot recover.

Suspension events refer primarily to events that occur asynchronously with the instruction's execution; these are specifically the interrupts generated by I/O peripheral devices, power-fail traps, and floating-point processor exceptions. Secondarily, suspension-events can refer also to those synchronous trap events that occur only for information notification purposes and do not imply that the integrity of the instruction's execution is in jeopardy. Such suspension events include "yellow zone" traps.

Potentially suspendable instructions have a defined architectural mechanism (PS<8> as described below) by which they can be suspended in midexecution to allow the processor to service suspension events. The instructions are subsequently resumed from the point where they had been suspended.

The presence of suspension events may cause certain CIS instructions to be suspended on some processors. If the instruction is suspended, PS<8> will be set, R7 will be backed up to address the op-code word, and the suspension event will be serviced. When the instruction is resumed, PS<8> indicates that execution of the instruction had been in progress.

In order to make these instructions suspendable on all processors, the instruction state is part of the user state saved by interrupt handling routines. The instruction state includes the general registers, condition codes, and memory. This state is processor-dependent when suspended. Software should not attempt to interpret or modify this state; it must only be saved and restored.

Up to 64_{10} words of the internal instruction state may also have been pushed onto the stack. This state must not be modified by software also. The instruction will remove this state from the stack when it is resumed.

If PS<8> is set prior to the execution of a potentially suspendable instruction, the effect of the instruction is unpredictable. PS<8> is cleared upon normal completion of a potentially suspendable instruction. PS<8> represents "instruction suspension" and has the corresponding mnemonic "IS."

All suspendable instructions use PS<8> to indicate instruction suspension. If, when a potentially suspendable instruction is executed, PS<8> is clear, the instruction is being commenced; if the bit is set, the instruction is being resumed. PS<8> is cleared when:

1. A suspended instruction is successfully completed.
2. The processor powers up.
3. A new PS is fetched from a vector location with PS<8> clear.
4. RTI or RTT is executed with a new PS<8> clear.
5. It is explicitly cleared by an instruction.

PS<8> is set when:

1. A potentially suspendable instruction is interrupted and requests to be suspended.
2. A new PS is fetched from a vector location with PS<8> set.
3. RTI or RTT is executed with PS<8> set.
4. It is explicitly set by an instruction.

The setting of this bit has no effect on instructions that are not potentially suspendable; such instructions do implicitly modify this bit.

When an instruction is suspended, the state that follows may contain information vital to the resumption of the instruction. The information must be preserved and restored prior to restarting the suspended instruction. The nature of the information may vary from one execution of the instruction to another; it may comprise any of the following.

1. General registers R0–R5.
2. Condition code bits (PS<3:0>).
3. Up to 64_{10} words on the stack of the context in which the suspended instruction had been executing.
4. Any destinations used by the instruction.

Stack Utilization

CIS instructions may use the R6 stack for temporary "scratch" state storage. The maximum number of additional words an extended instruction may claim on the R6 stack is 64_{10}. The reason for imposing a limit is to ensure that system software can adequately provide for worst-case stack allocation requirements. In addition to the above restriction, the normal PDP-11 stack-limit mechanism remains in effect for extended instructions just as it does for any other instruction.

If insufficient stack space exists, the instruction will terminate by a memory management abort in such a way that if additional stack space is allocated, the instruction will successfully restart.

Notation

dst	Destination string
src1	Source string 1
src2	Source string 2
dscr	Descriptor

14.7 EXTENDED INSTRUCTION DEFINITIONS

The commercial instruction set contains instructions to manipulate various data types and strings, including character, numeric and decimal data. The operations performed include data type conversions, string search operations, block moves, and arithmetic operations. The definitions of the 52 instructions that compose the CIS are described in Paragraphs 14.7.1 through 14.7.20.

14.7.1 ADDN/ADDP/ADDNI/ADDPI

Purpose: Add Decimal

Operation: dst src2 + src1

Condition Codes:
N: set if dst < 0; cleared otherwise
Z: set if dst = 0; cleared otherwise
V: set if dst cannot contain all significant digits of the result; cleared otherwise
C: cleared

Op Codes:

ADDN	076050
ADDP	076070
ADDNI	076150
ADDPI	076170

Description: Src1 is added to src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form - ADDN and ADDP

When the instruction starts, the operands must have been placed in the general registers: the first source descriptor must be in R0–R1, the second source descriptor in R2–R3, and the destination descriptor in R4–R5. The add decimal format is shown in Figure 14-18.

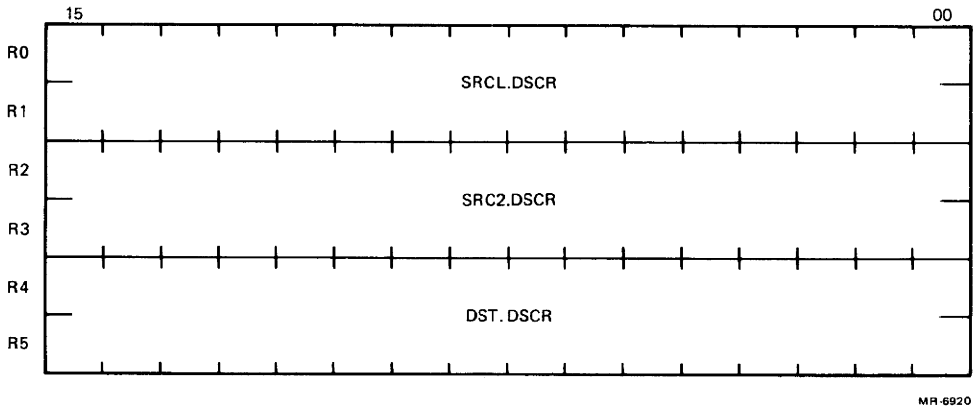


Figure 14-18 Add Decimal Format

When the instruction is completed, the source descriptor registers are cleared, as shown in Figure 14-19.

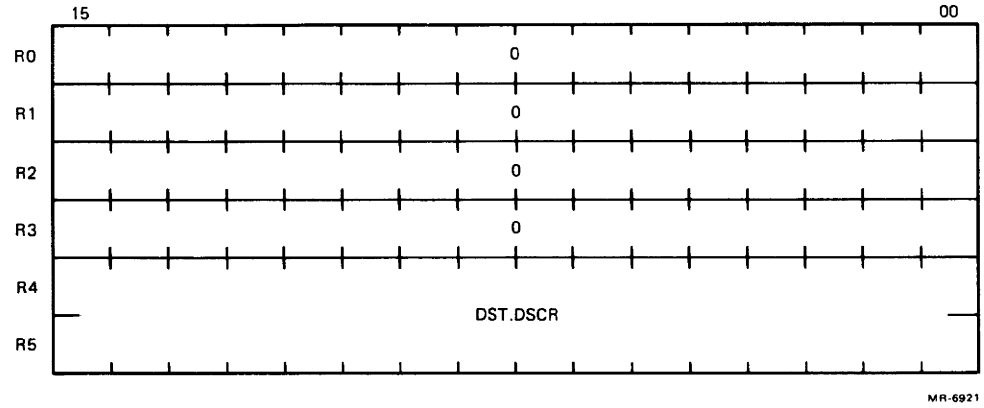


Figure 14-19 Add Decimal Format (Cleared)

In-Line Form - ADDNI and ADDPI
Each word address pointer that follows the op-code word in the instruction stream refers to a 2-word decimal string descriptor. R0–R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings, provided that each source string is a valid representation of the specified data type.

2. Source strings may overlap the destination string only if all corresponding digits of the strings are in coincident bytes in memory.

14.7.2 ASHN/ASHP/ASHNI/ASHPI

Purpose: Arithmetic Shift Decimal

Operation: dst src * 10** (shift count)

Condition Codes:
N: set if dst < 0; cleared otherwise
V: set if dst cannot contain all significant digits of the result; cleared otherwise
Z: set if dst = 0; cleared otherwise
C: cleared

Op Codes:

ASHN	076056
ASHP	076076
ASHNI	076156
ASHPI	076176

Description: The decimal number specified by the source descriptor is arithmetically shifted and stored in the area specified by the destination descriptor. The shifted result is aligned with the least significant digit position in the destination string. The shift count is a 2's complement byte whose value ranges from -128_{10} to $+127_{10}$. If the shift count is positive, a shift in the direction of least to most

significant digits is performed. A negative shift count performs a shift from most to least significant digit. Thus, the shift count is the power of 10 by which the source is multiplied; negative powers of 10 effectively divide. Zero digits are supplied for vacated digit positions. A zero shift count will move the source to the destination. The condition codes reflect the value stored in the destination string, and whether all significant digits are stored.

A negative shift count invokes a rounding operation. The result is constructed by shifting the source the specified number of digit positions. The rounding digit is then added to the most significant digit that was shifted out. If this sum is less than 10_{10} the shifted result is stored in the destination string. If the sum is 10_{10} or greater the magnitude of the shifted result is increased by 1 and then stored in the destination string. If no rounding is desired, the rounding digit should be zero.

The shift count and rounding digit are represented in a single word referred to as the shift descriptor. Bits <15:12> of this word must be zero. The shift descriptor format is shown in Figure 14-20.

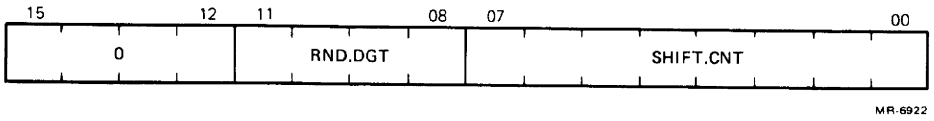


Figure 14-20 Shift Descriptor Format

Register Form – ASHN and ASHP
When the instruction starts, the operands must have been placed in the general registers: the source descriptor must be in R0–R1, the destination descriptor in R2–R3, and the shift descriptor in R4. The arithmetic shift decimal format is shown in Figure 14-21.

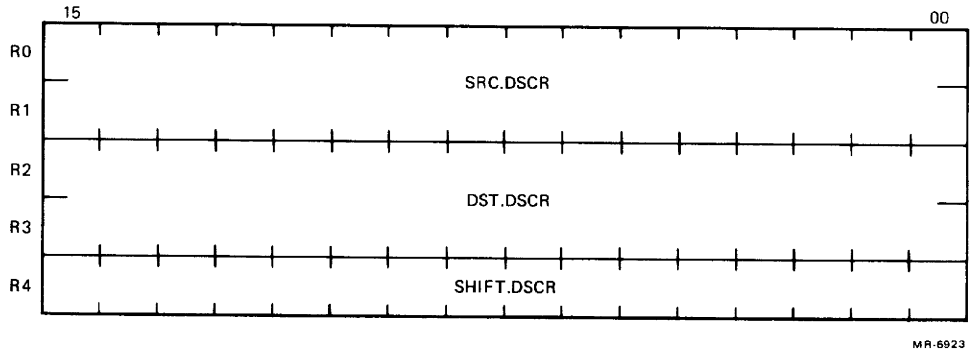


Figure 14-21 Arithmetic Shift Decimal Format

When the instruction is completed, the source descriptor registers and shift descriptor register are cleared, as shown in Figure 14-22.

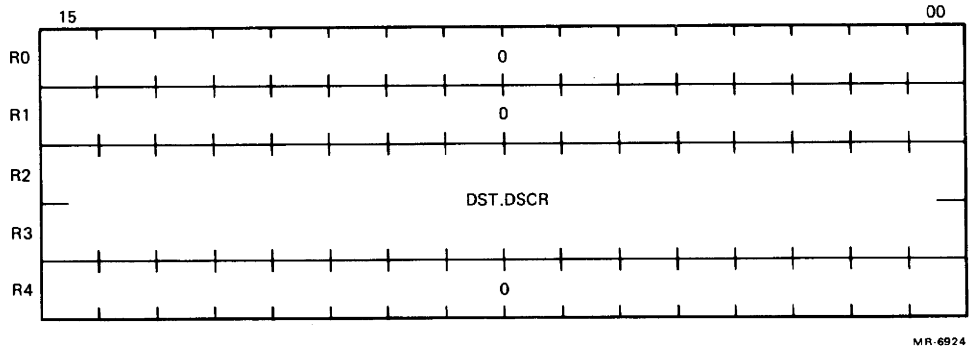


Figure 14-22 Arithmetic Shift Decimal Format (Cleared)

In-Line Form - ASHNI and ASHPI
The words followong the op-code word in the instruction stream are a word address pointer to a 2-word decimal string source descriptor, a word address pointer to a 2-word decimal string destination descriptor, and a shift descriptor word. R0–R6 are unchanged when the instruction is completed.

Notes:

1. If bits <15:12> of the shift descriptor word are not zero, the effect of the instruction is unpredictable.

2. If bits <11:8> of the shift descriptor are not a valid decimal digit, the results of the instruction are unpredictable.

3. Any overlap of the source and destination strings will produce unpredictable results.

14.7.3 CMPC/CMPCI

Purpose: Compare Character

Operation: Src1 is compared with src2 (src1 − src2).

Condition Codes: The condition codes are based on the arithmetic comparison of the most significant pair of unequal src1 and src2 characters (src1.byte − src2.byte)

N: set if dst < 0; cleared otherwise

Z: set if dst = 0; cleared otherwise

V: set if there was arithmetic overflow, that is, src1.byte<7> and src2.byte<7> were different, and src2.byte<7> was the same as bit <7> of <src1.byte − src2.byte); cleared otherwise

C: cleared if there was a carry from the most significant bit of the result; set otherwise

Op Codes:

CMPC	076044
CMPCI	076144

Description: Each character of src1 is compared with the corresponding character of src2 by examining the character strings from most significant to least significant characters. If the character strings are of unequal length, the shorter character string is conceptually extended to the length of the longer character string with fill characters after its least significant character. The instruction terminates when the first corresponding unequal characters are found or when both character strings are exhausted. The condition codes reflect the last comparison, permitting the unsigned branch instructions to test the result.

Register Form - CMPC

When the instruction starts, the operands must have been placed in the general registers: the first source character string descriptor must be in R0–R1, the second source character string descriptor in R2–R3, the fill character in R4<7:0>, and R4<15:8> must be zero. The compare character format is shown in Figure 14-23.

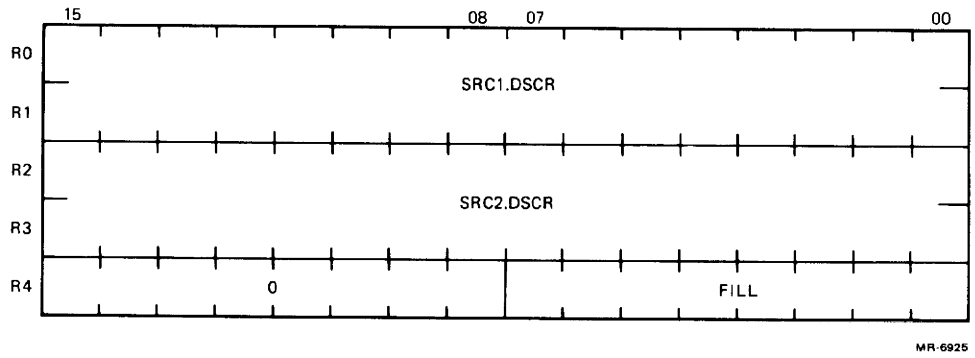


Figure 14-23 Compare Character Format

The instruction terminates with substring descriptors in R0–R1 and R2–R3 that represent the portion of each source character string beginning with the most significant corresponding unequal characters. R0–R1 contain a descriptor for the unequal portion of the original src1 string; R2–R3 contain a descriptor for the unequal portion of the original src2 string. A vacant character string descriptor indicates that the entire source character string was equal to the corresponding portion of the other source character string, including extension by the fill character. The vacant character string descriptor's address is one greater than that of the least significant character of the character string. The compare character format when the instruction terminates is shown in Figure 14-24.

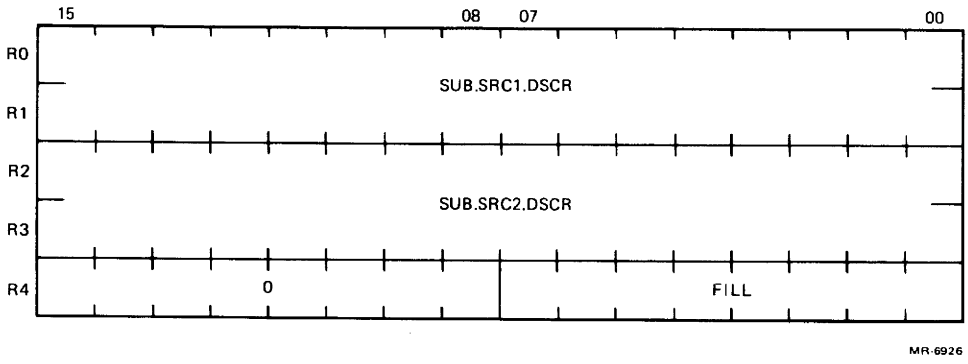


Figure 14-24 Compare Character Termination Format

In-Line Form - CMPCI
The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string src1 descriptor, a word address pointer to a 2-word character string src2 descriptor, and a word whose low-order half contains the fill character and whose high-order half must be zero. R0–R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source character strings.

2. If the src1 character string is vacant, the fill character will be compared with src2. If the src2 character string is vacant, the fill character will be compared with src1. If both character strings are vacant, the condition codes will indicate equality.

3. CMPC – If an initial source character string descriptor is vacant, the resulting substring descriptor is the same as the original character string descriptor.

4. A test for success is BEQ; a test for failure is BNE.

5. When the instruction terminates, the condition codes will be set as if a CMPB instruction operated on the most significant unequal characters. If both strings are initially vacant or are identical, the condition codes will be set as if the last characters to be compared were identical. This results in equality with N, V, and C cleared, and Z set.

6. Both CMPC and CMPCI update the condition codes. CMPC returns substring descriptors.

14.7.4 CMPN/CMPP/CMPNI/CMPPI

Purpose: Compare Decimal

Operation: Src1 is compared with src2 (src1 − src2).

Condition Codes:
N: set if src1 < src2; cleared otherwise
Z: set if src1 = src2; cleared otherwise
V: cleared
C: cleared

Op Codes:

CMPN	076052
CMPP	076072
CMPNI	076152
CMPPI	076172

Description: Src1 is arithmetically compared with src2, with the condition codes reflecting the comparison. The signed branch instruction can be used to test the result.

Register Form - CMPN and CMPP
When the instruction starts, the operands must have been placed in the general registers: the first source descriptor must be in R0–R1, and the second source descriptor in R2–R3. The compare decimal format is shown in Figure 14-25.

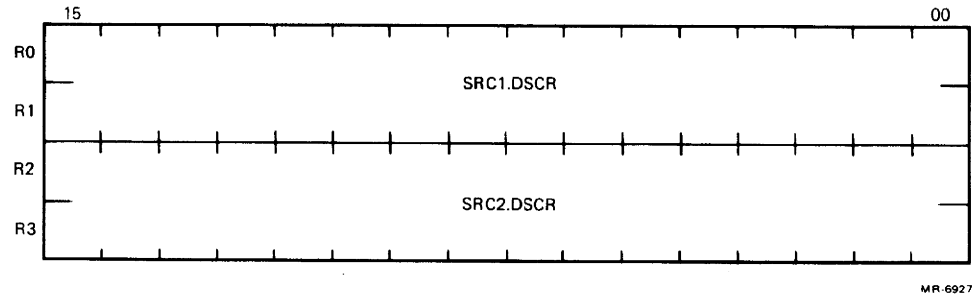


Figure 14-25 Compare Decimal Format

When the instruction is completed, the source descriptor registers are cleared, as shown in Figure 14-26.

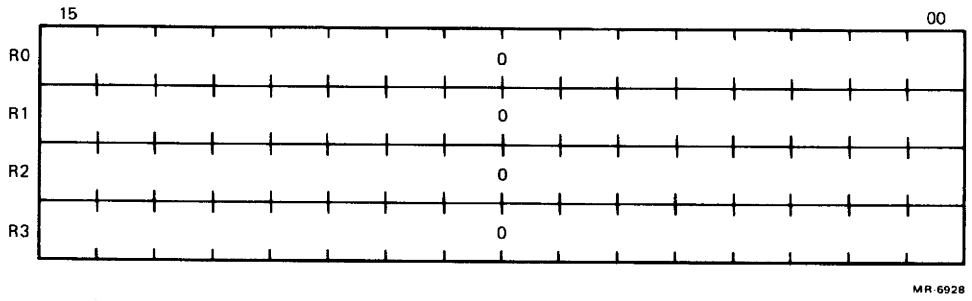


Figure 14-26 Compare Decimal Format (Cleared)

In-Line Form – CMPNI and CMPPI
Each word address pointer following the op-code word in the instruction stream refers to a 2-word decimal string descriptor. R0–R6 are unchanged when the instruction is completed.

Note:

1. The operation of these instructions is unaffected by any overlap of the source strings, provided that each source string is a valid representation of the specified data type.

14.7.5 CVTLN/CVTLP/CVTLNI/CVTLPI

Purpose: Convert Long-to-Decimal

Operation: decimal string long integer

Condition Codes:
N: set if dst < 0; cleared otherwise
Z: set if dst = 0; cleared otherwise
V: set if dst cannot contain all significant digits of the result; cleared otherwise
C: cleared

Op Codes:

CVTLN	076057
CVTLP	076077
CVTLNI	076157
CVTLPI	076177

Description: The source long integer is converted to a decimal string. The condition codes reflect the result stored in the destination decimal string, and whether all significant digits are stored.

Register Form – CVTLN and CVTLP
When the instruction starts, the operands must have been placed in the general registers: the destination descriptor must be in R0–R1, and the source long integer in R2–R3. The convert long-to-decimal format is shown in Figure 14-27.

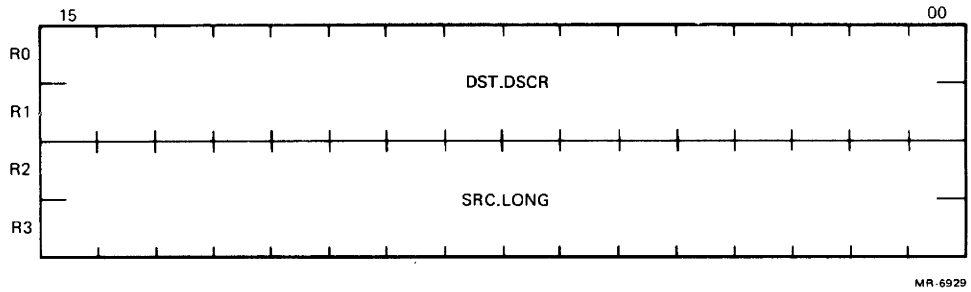


Figure 14-27 Convert Long-to-Decimal Format

When the instruction is completed, the source long-integer registers are cleared, as shown in Figure 14-28.

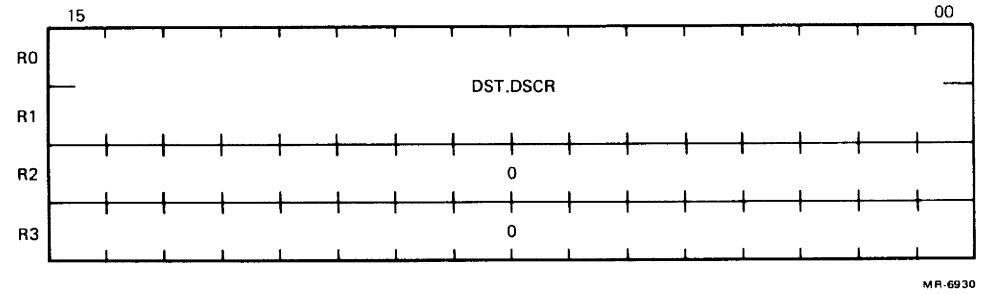


Figure 14-28 Convert Long-to-Decimal Format (Cleared)

In-Line Form – CVTLNI and CVTLPI
The words following the op-code word in the instruction stream are a word address pointer to a 2-word decimal string descriptor, and a word address pointer to a 2-word long-integer source. R0–R6 are unchanged when the instruction is completed.

Notes:

1. Register forms use a long integer oriented with the sign and high-order portion of R2, and the low-order portion in R3.

2. In-line forms use a long integer oriented with the low-order portion in src.long, and the sign and high-order portion in src.long + 2.

14.7.6 CVTNL/CVTPL/CVTNLI/CVTPLI

Purpose: Convert Decimal-to-Long

Operation: long integer decimal string

Condition Codes: The condition codes are based on the long-integer destination and on the sign of the source decimal string.

N: set if long.integer $<$ 0; cleared otherwise

Z: set if long.integer $=$ 0; cleared otherwise

V: set if long.integer dst cannot correctly represent the 2's complement form of the result; cleared otherwise

C: set if src $<$ 0 and long.integer $\neq$ 0; cleared otherwise

Op Codes:

CVTNL	076053
CVTPL	076073
CVTNLI	076153
CVTPLI	076173

Description: The source decimal string is converted to a long integer. The condition codes reflect the result of the operation, and whether significant digits were not converted.

Register Form – CVTNL and CVTPL
When the instruction starts, the operand must have been placed in the general registers: the source decimal string descriptor must be in the R0–R1. The convert decimal-to-long format is shown in Figure 14-29.

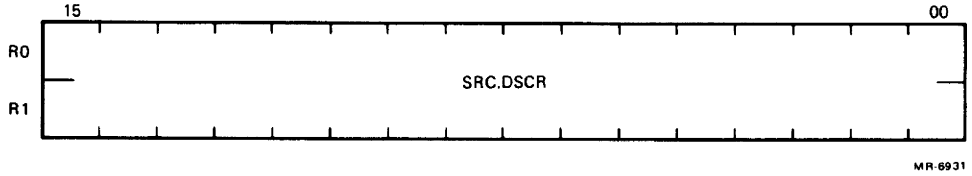


Figure 14-29 Convert Decimal-to-Long Format

When the instruction is completed, the source decimal string descriptors are cleared, and the destination long integer is returned in R2–R3, as shown in Figure 14-30.

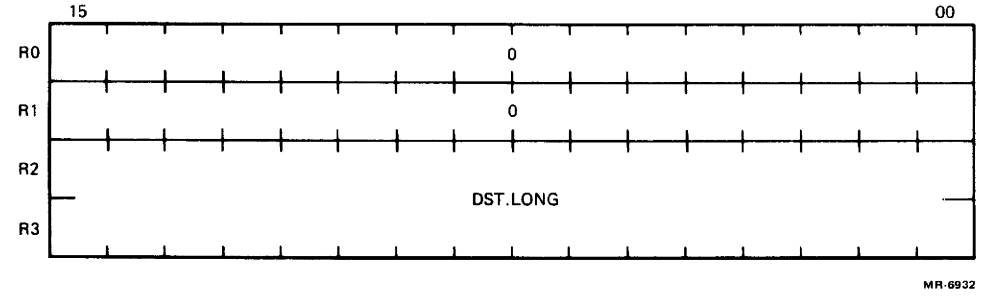


Figure 14-30 Convert Decimal-to-Long Format (Cleared)

In-Line Form – CVTNLI and CVTPLI
The words following the op-code word in the instruction stream are a word address pointer to a 2-word decimal string source descriptor, and a word address pointer to a 2-word long integer destination. R0–R6 are unchanged when the instruction is completed.

Notes:

1. Register forms use a long integer oriented with the sign and high-order portion in R2, and the low-order portion in R3.

2. In-line forms use a long integer oriented with the low-order portion in dst.long, and the sign and high-order portion in dst.long + 2.

3. If the V bit is set, the contents of the long-integer destination are the least significant 32 bits of the result.

4. A source whose value is +231 can be represented as a 32-bit binary integer. However, since the destination is a 2's complement long integer, the resulting condition codes will be: N set, Z cleared, V set, and C cleared.

14.7.7 CVTNP/CVTPN/CVTNPI/CVTPNI

Purpose: Convert Decimal

Operation:
CVTNP/CVTNPI packed string numeric string
CVTPN/CVTPNI numeric string packed string

Condition Codes:
N: set if dst < 0; cleared otherwise
Z: set if dst = 0; cleared otherwise
V: set if dst cannot contain all significant digits of the result; cleared otherwise
C: cleared

Op Codes:

CVTNP	076055
CVTPN	076054
CVTNPI	076155
CVTPNI	076154

Description: These instructions convert between numeric and packed decimal strings. The source decimal string is converted and moved to the destination string. The condition codes reflect the result of the operation, and whether all significant digits were stored.

Register Form – CVTNP and CVTPN
When the instruction starts, the operands must have been placed in the general registers: the source descriptor must be in R0–R1, and the destination descriptor in R2–R3. The convert decimal format is shown in Figure 14-31.

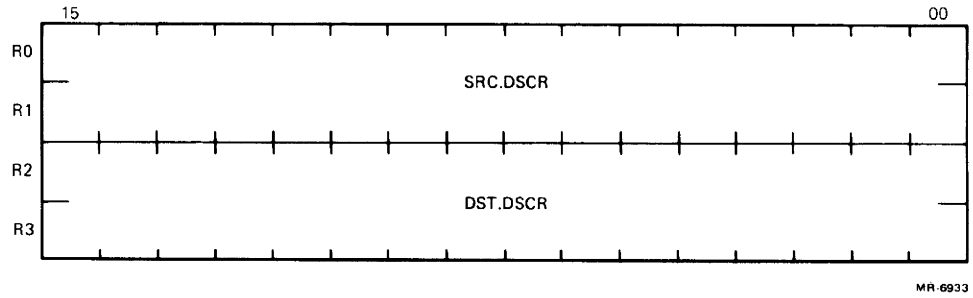


Figure 14-31 Convert Decimal Format

When the instruction is completed, the source descriptor registers are cleared, as shown in Figure 14-32.

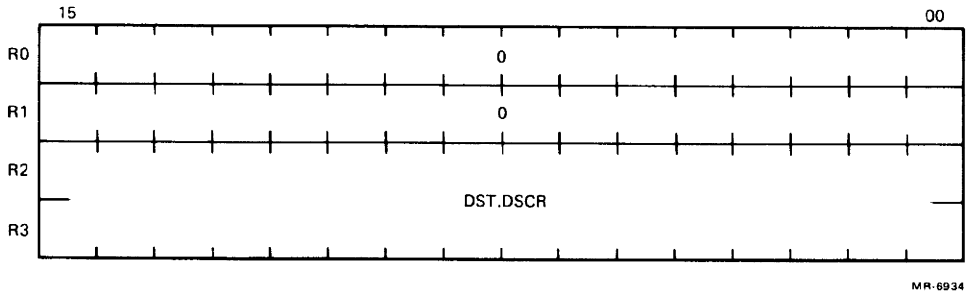


Figure 14-32 Convert Decimal Format (Cleared)

In-Line Form - CVTNPI and CVTPNI
Each word address pointer following the op-code word in the instruction stream refers to a 2-word decimal string descriptor. R0–R6 are unchanged when the instruction is completed.

Notes:

1. The results of the instruction are unpredictable if the source and destination strings overlap.
2. These instructions use both a numeric and packed decimal string descriptor.

14.7.8 DIVP/DIVPI

Purpose: Divide Decimal

Operation: dst src2/src1

Condition Codes: N: set if dst < 0; cleared otherwise

Z: set if dst = 0; cleared otherwise

V: set if dst cannot contain all significant digits of the result or if src1 = 0; cleared otherwise

C: set if src1 = 0; cleared otherwise

Op Codes:

DIVP	076075
DIVPI	076175

Description: Src2 is divided by src1, and the quotient (fraction truncated) is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form – DIVP
When the instruction starts, the operands must have been placed in the general registers: the first source descriptor must be in R0–R1, the second source descriptor in R2–R3, and the destination descriptor in R4–R5. The divide decimal format is shown in Figure 14-33.

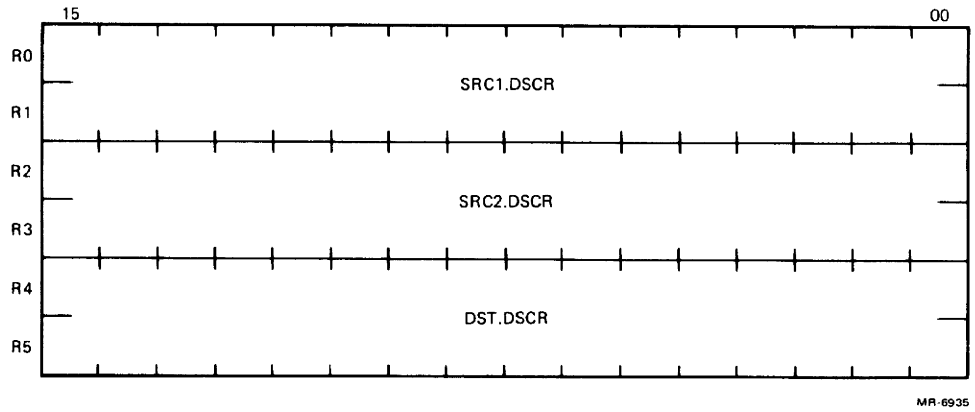


Figure 14-33 Divide Decimal Format

When the instruction is completed, the source descriptor registers are cleared, as shown in Figure 14-34.

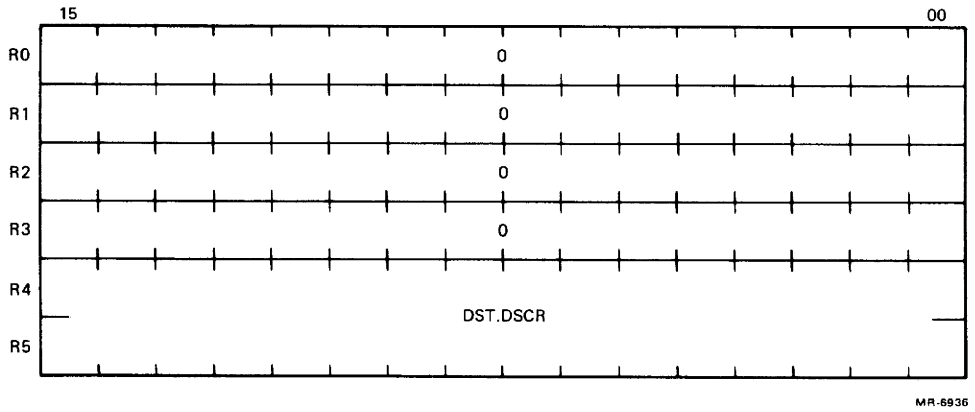


Figure 14-34 Divide Decimal Format (Cleared)

In-Line Form –DIVPI
Each word address pointer following the op-code word in the instruction stream refers to a 2-word decimal string descriptor. R0–R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings, provided that each source string is a valid representation of the specified data type.

2. The results of the instruction are unpredictable if the source and destination strings overlap.

3. Division by zero will set the V and C bits. The destination string, and the N and Z condition code bits will be unpredictable.

4. No numeric string divide instruction is provided.

14.7.9 LOCC/LOCCI

Purpose: Locate Character

Operation: Search source character string for a character.

Condition Codes: The condition codes are based on the final contents of R0.

N: set if R0 <15> is set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

Op Codes:

LOCC	076040
LOCCI	076140

Description: The source character string is searched from most significant to least significant character until the first occurrence of the search character. A character string descriptor is returned in R0–R1 that represents the portion of the source character string beginning with the located character. If the source character string contains only characters not equal to the search character, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the result value in R0.

Register Form – LOCC
When the instruction starts, the operands must have been placed in the general registers: the source character string descriptor must be in R0–R1, the search character in R4 <7:0>, and R4 <15:8> must be zero. The register form of the locate character format is shown in Figure 14-35.

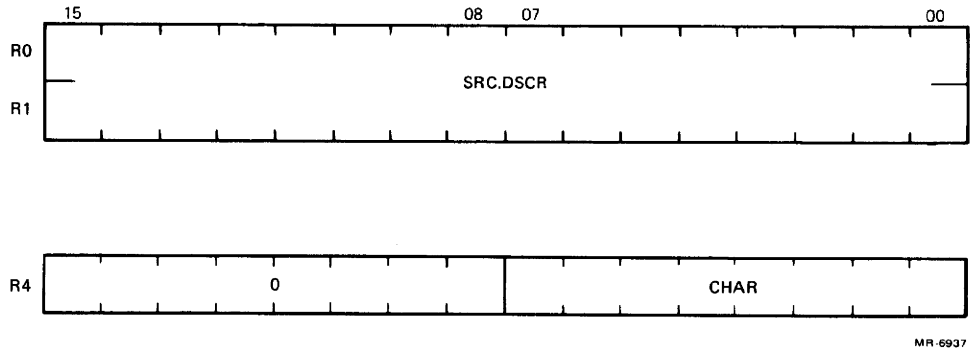


Figure 14-35 Locate Character Format (Register Form)

When the instruction is completed, R0–R1 contain a character set descriptor that represents the substring of the source character string, beginning with the located character, as shown in Figure 14-36.

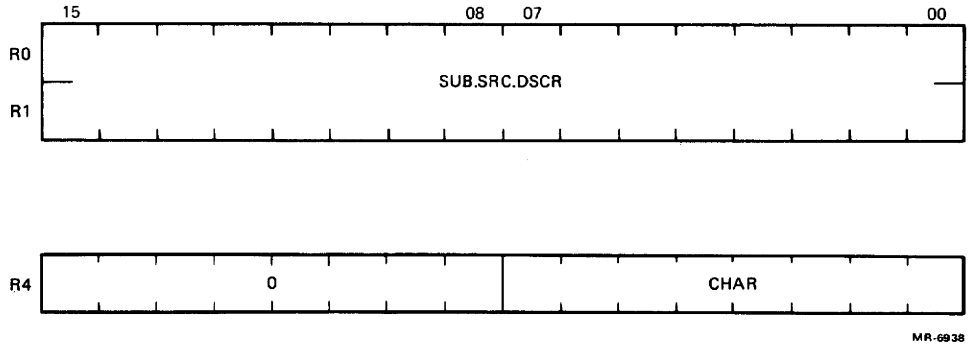


Figure 14-36 Locate Character Termination Format

In-Line Form – LOCCI
The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string source descriptor, and a word whose low-order half contains the search character and whose high-order half must be zero. When the instruction is completed, R0–R1 contain a character string descriptor that represents the substring of the source character string beginning with the located character. R2–R6 are unchanged. The in-line form of the locate character format is shown in Figure 14-37.

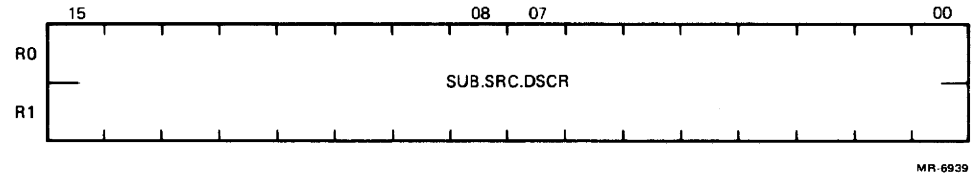


Figure 14-37 Locate Character Format (In-Line)

Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that no match was found. The original source character string descriptor is returned in R0–R1.

2. A test for success is BNE; a test for failure is BEQ.

3. The condition codes will be set as if this instruction were followed by TST R0.

14.7.10 L2DR

Purpose: Load Two Descriptors

Operation: Load word pairs into R0–R1 and R2–R3.

Condition Codes:
N: not affected
Z: not affected
V: not affected
C: not affected

Op Code: L2DR 07602r

Description: This instruction augments the character and decimal string instructions by efficiently loading string descriptors into the general registers. A descriptor "alpha" is loaded into R0–R1; a second descriptor "beta" is loaded into R2–R3. The address of the descriptors is determined by the addressing mode @ (Rr)+ where r is the low-order three bits of the op-code word. The address of the descriptor "alpha" is derived by applying this addressing mode once; the address of the descriptor "beta" is derived by applying this addressing mode a second time. The addressing mode autoincrements the indicated register by two. The addressing mode computation is not affected by the descriptors loaded into the general registers. The words containing the addresses of the descriptors are in consecutive words in memory; the descriptors themselves may be anywhere in memory. The condition codes are not affected.

When the instruction is completed, the "alpha" descriptor is in R0–R1 and the "beta" descriptor is in R2–R3. The load two descriptors format is shown in Figure 14-38.

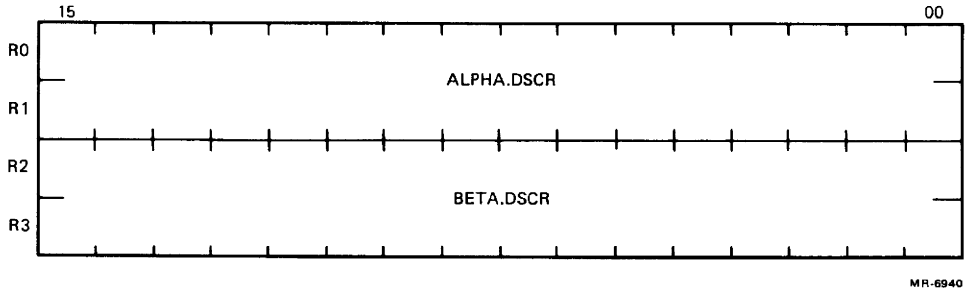


Figure 14-38 Load Two Descriptors Format

14.7.11 L3DR

Purpose: Load Three Descriptors

Operation: Load word pairs into R0–R1, R2–R3, and R4–R5.

Condition Codes:
N: not affected
Z: not affected
V: not affected
C: not affected

Op Code: L3DR 07606r

Description: This instruction augments the character and decimal string instructions by efficiently loading string descriptors into the general registers. A descriptor "alpha" is loaded into R0–R1; a second descriptor "beta" is loaded into R2–R3; a third descriptor "gamma" is loaded into R4–R5. The address of the descriptors is determined by the addressing mode @(Rr)+ where r is the low-order three bits of the op-code word. The address of the descriptor "alpha" is derived by applying this addressing mode once. The address of the descriptor "beta" is derived by applying this addressing mode a second time. The address of the de-

scriptor "gamma" is derived by applying this addressing mode a third time. The addressing mode autoincrements the indicated register by two. The addressing mode computation is not affected by the descriptors loaded into the general registers. The words containing the addresses of the descriptors are in consecutive words in memory; the descriptors themselves may be anywhere in memory. The condition codes are not affected.

When the instruction is completed, the "alpha" descriptor is in R0–R1, the "beta" descriptor is in R2–R3, and the "gamma" descriptor is in R4–R5. The load three descriptors format is shown in Figure 14-39.

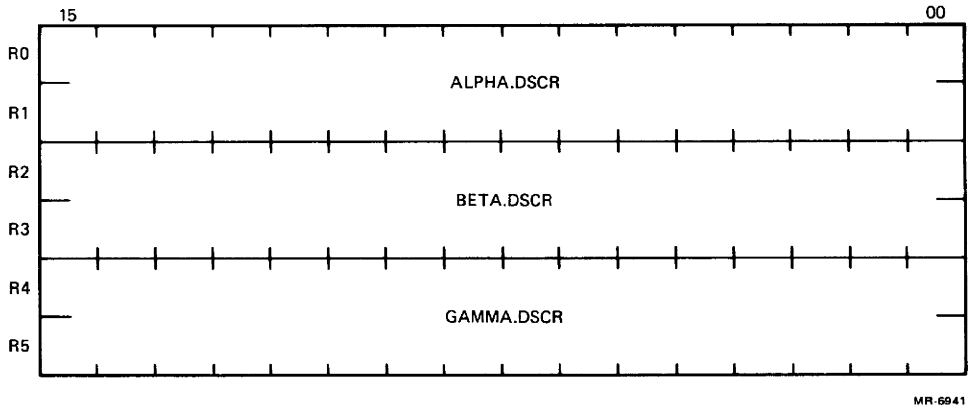


Figure 14-39 Load Three Descriptors Format

14.7.12 MATC/MATCI

Purpose: Match Character

Operation: Search source character string for object character string.

Condition Codes: The condition codes are based on the final contents of R0.

N: set if R0<15> is set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

Op Codes:

MATC	076045
MATCI	076145

Description: The source character string is searched from most significant to least significant character for the first occurrence of the entire object character string. A character string descriptor is returned in R0–R1 that represents the portion of the original source character string, from the most significant character that completely matches the object character string to the end of the source character string. If the object character string does not completely match any portion of the source character string, the character descriptor returned in R0–R1 is vacant, with an address one greater than the least significant character in the source string. The

condition codes reflect the resulting value in R0. If the Z bit is cleared, the entire object character string was successfully matched with the source character string; if the Z bit is set, the match failed.

Register Form - MATC

When the instruction starts, the operands must have been placed in the general registers: the source character string descriptor must be in R0–R1, and the object character string descriptor in R2–R3. The register form of the match character format is shown in Figure 14-40.

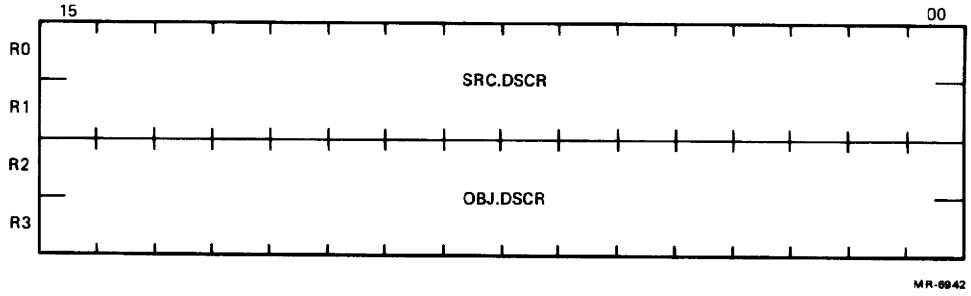


Figure 14-40 Match Character Format (Register Form)

The instruction terminates with a character substring descriptor returned in R0–R1 that represents the portion of the original source character string, beginning with the most significant character to completely match the object character string. The format of the match character after termination is shown in Figure 14-41.

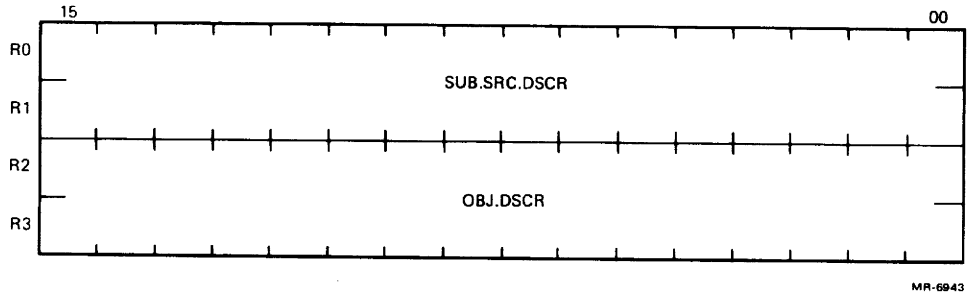


Figure 14-41 Match Character Termination Format

In-Line Form - MATCI

The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string source descriptor, and a word address pointer to a 2-word character string object descriptor. The instruction terminates with a character substring descriptor returned in R0–R1 that represents the portion of the original source character string, beginning with the most significant character to completely match the object character string. R2–R6 are unchanged when the instruction is completed. The in-line form of the match character format is shown in Figure 14-42.

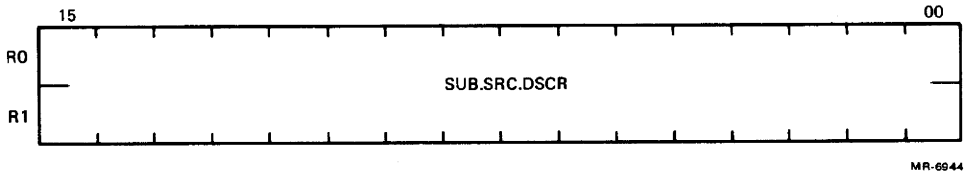


Figure 14-42 Match Character Format (In-Line)

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and object character strings.

2. A vacant object character string matches any nonvacant source character string. A vacant source character string will not match any object character string. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that no match was found. The original source character string descriptor is returned in R0–R1.

3. If the length of the object character string is greater than that of the source character string, no match is found; R0–R1 and the condition codes will be updated.

4. A test for success is BNE; a test for failure is BEQ.

5. The condition codes will be set as if this instruction were followed by TST R0.

14.7.13 MOVC/MOVCI

Purpose: Move Character

Operation: dst src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character string lengths (result = src.len–dst.len).

N: set if result < 0; cleared otherwise

Z: set if result = 0; cleared otherwise

V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len–dst.len); cleared otherwise

C: cleared if there was a carry from the most significant bit of the result; set otherwise

Op Codes:

MOVC	076030
MOVCI	076130

Description: The character string specified by the source descriptor is moved into the area specified by the destination descriptor. It is aligned by the most significant character. The condition codes reflect an arithmetic comparison of the original source and destination lengths. If the source string is shorter than the destination string, the fill character is used to complete the least significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits' being cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This is indicated by the Z bit's being set. The unsigned branch instructions may test the result of the instruction.

Register Form – MOVC

When the instruction starts, the operands must have been placed in the general registers: the source character string descriptor must be in R0–R1, the destination character string descriptor in R2–R3, the fill character in R4<7:0>, and R4<15:8> must be zero. The move character format is shown in **Figure 14-43.**

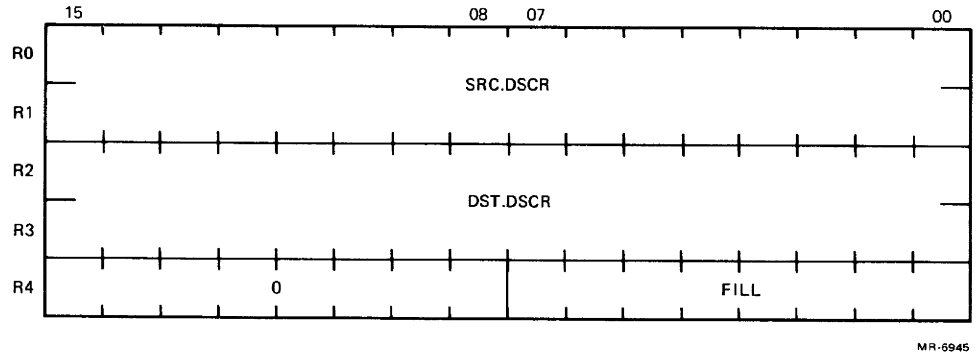


Figure 14-43 Move Character Format

When the instruction is completed, R0 contains the number of unmoved source string characters, and R1–R3 are cleared, as shown in Figure 14-44.

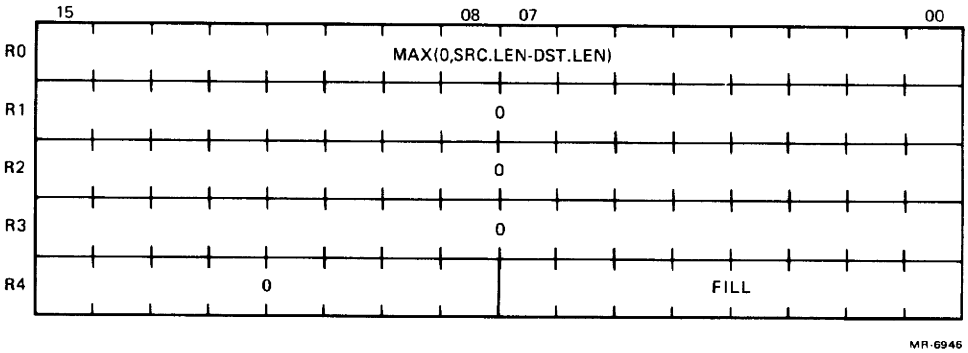


Figure 14-44 Move Character Format (Cleared)

In-Line Form – MOVCI

The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string source descriptor, a word address pointer to a 2-word character string destination descriptor, and a word whose low-order half contains the fill character and whose high-order half must be zero. R0–R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.

2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. The condition codes will be updated. MOVC will update the general registers.

3. MOVC – When the instruction terminates, R0 is zero only if Z or C is set.

4. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.

14.7.14 MOVRC/MOVRCI

Purpose: Move Reverse-Justified Character

Operation: dst reverse-justified src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character strings (result = src.len–dst.len).

N: set if result < 0; cleared otherwise

Z: set if result = 0; cleared otherwise

V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len–dst.len); cleared otherwise

C: cleared if there was a carry from the most significant bit of the result; set otherwise

Op Codes:

MOVRC	076031
MOVRCI	076131

Description: The character string specified by the source descriptor is moved into the area specified by the destination descriptor. It is aligned by the least significant character. The condition codes reflect an arithmetic comparison of the original source and destination lengths. If the source string is shorter than the destination string, the fill character is used to complete the most significant part of the destination string. This is indicated by the C bit's being set. If the source string is longer than the destination string, the most significant characters of the source string are not moved. This is indicated by the Z and C bits' being cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This is indicated by the Z bit's being set. The unsigned branch instructions may test the result of the instruction.

Register Form – MOVRC
When the instruction starts, the operands must have been placed in the general registers: the source character string descriptor must be in R0–R1, the destination character string descriptor in R2–R3, the fill character in R4 <7:0>, and R4<15:8> must be zero. The move reverse-justified character format is shown in Figure 14-45.

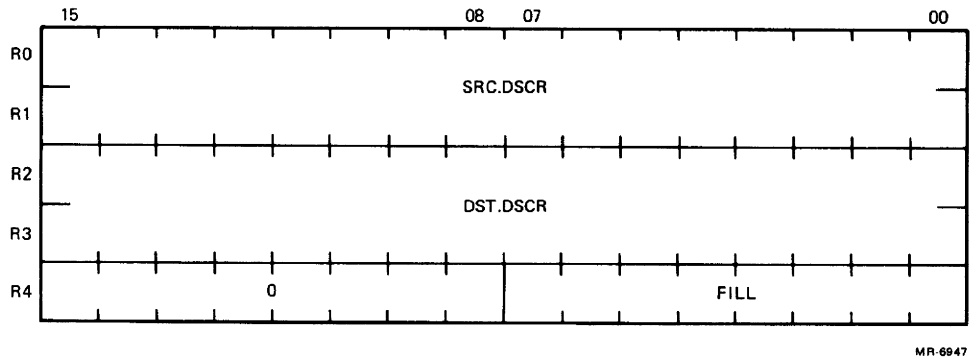


Figure 14-45 Move Reverse-Justified Character Format

When the instruction is completed, R0 contains the number of unmoved source string characters, and R1–R3 are cleared, as shown in Figure 14-46.

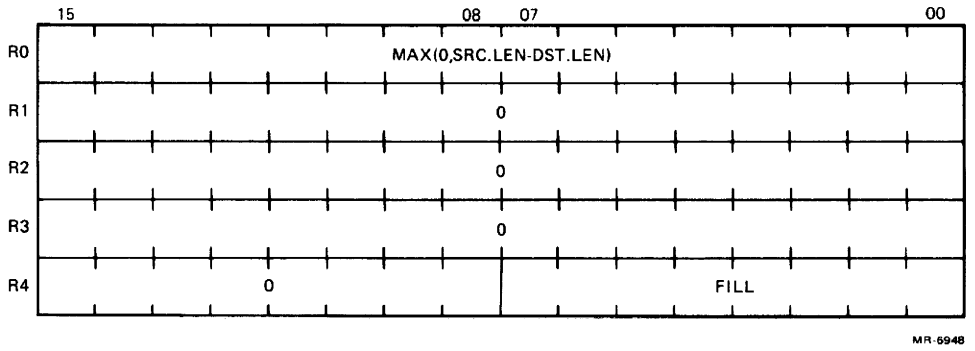


Figure 14-46 Move Reverse-Justified Character Format (Cleared)

In-Line Form - MOVRCI
The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string source descriptor, a word address pointer to a 2-word character string destination descriptor, and a word whose low-order half contains the fill character and whose high-order half must be zero. R0–R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVRC will update the general registers.
3. MOVRC – When the instruction terminates, R0 is zero only if Z or C are set.
4. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.

14.7.15 MOVTC/MOVTCI

Purpose: Move Translated Character

Operation: dst translated src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character string lengths (result = src.len–dst.len).

N: set if result < 0; cleared otherwise

Z: set if result = 0; cleared otherwise

V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len–dst.len); cleared otherwise

C: cleared if there was a carry from the most significant bit of the result; set otherwise

Op Codes:

MOVTC	076032
MOVTCI	076132

Description: The character string specified by the source descriptor is translated and moved into the area specified by the destination descriptor. It is aligned by the most significant character. Translation is accomplished by using each source character as an 8-bit positive integer index into a 256-byte table, the address of which is an operand of the instruction. The byte at the indexed location in the table is stored in the destination string. The condition codes reflect an arithmetic comparison of the original source and destination lengths.

If the source string is shorter than the destination string, the untranslated fill character is used to complete the least significant part of the destination string. This is indicated by the C bit's being set. If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits' being cleared. If the source and destination strings are of equal length, all characters are translated and moved with neither truncation nor filling. This is indicated by the Z bit's being set. The unsigned branch instructions may test the result of the instruction.

Register Form – MOVTC
When the instruction starts, the operands must have been placed in the general registers: the source character string descriptor must be in R0–R1, the destination character string descriptor in R2–R3, the fill character in R4<7:0>, R4<15:8> must be zero, and the translation table address in R5. The move translated character format is shown in Figure 14-47.

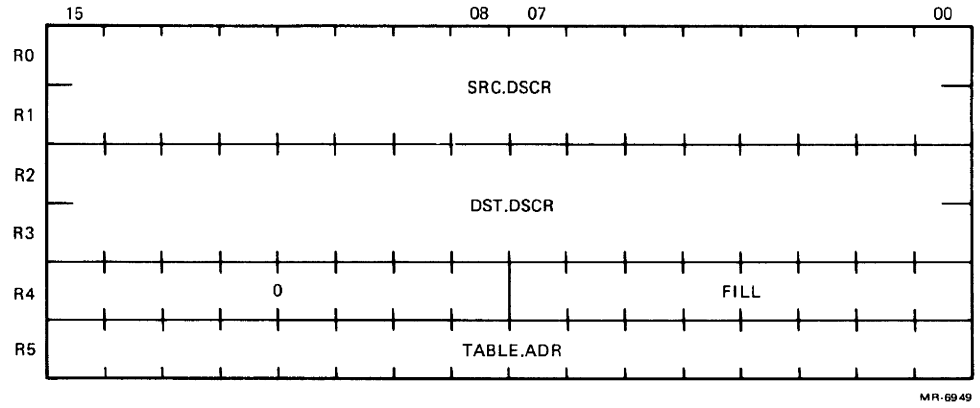


Figure 14-47 Move Translated Character Format

When the instruction is completed, R0 contains the number of unmoved source string characters, and R1–R3 are cleared, as shown in Figure 14-48.

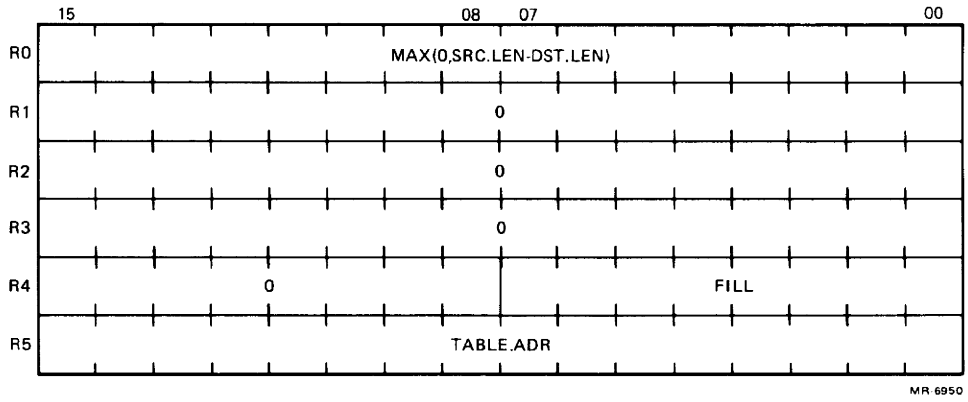


Figure 14-48 Move Translated Character Format (Cleared)

In-Line Form – MOVTCI
The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string source descriptor, a word address pointer to a 2-word character string destination descriptor, a word whose low-order half contains the fill character and whose high-order half must be zero, and a word containing the address of the translation table. R0–R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.
2. If the destination string overlaps the translation table in any way, the results of the instruction will be unpredictable.

3. If the source string is vacant, the untranslated fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVTC will update the general registers.

4. MOVTC – When the instruction terminates, R0 is zero only if Z or C are set.

5. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.

6. The effect of the instruction is unpredictable if the entire 256-byte translation table is not in readable memory.

14.7.16 MULP/MULPI

Purpose: Multiply Decimal

Operation: dst src2 * src1

Condition Codes:
N: set if dst < 0; cleared otherwise
Z: set if dst = 0; cleared otherwise
V: set if dst cannot contain all significant digits of the result; cleared otherwise
C: cleared

Op Codes:

MULP	076074
MULPI	076174

Description: Src1 and src2 are multiplied, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form – MULP
When the instruction starts, the operands must have been placed in the general registers: the first source descriptor must be in R0–R1, the second source descriptor in R2–R3, and the destination descriptor in R4–R5. The multiply decimal format is shown in Figure 14-49.

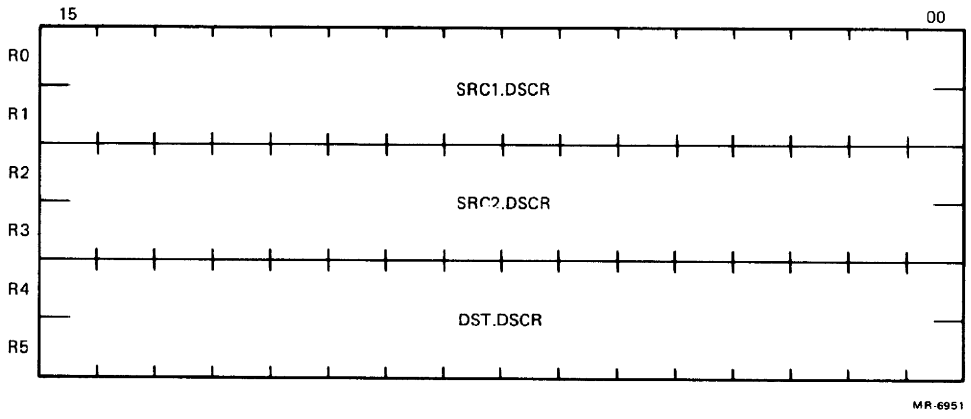


Figure 14-49 Multiply Decimal Format

When the instruction is complete, the source descriptor registers are cleared, as shown in Figure 14-50.

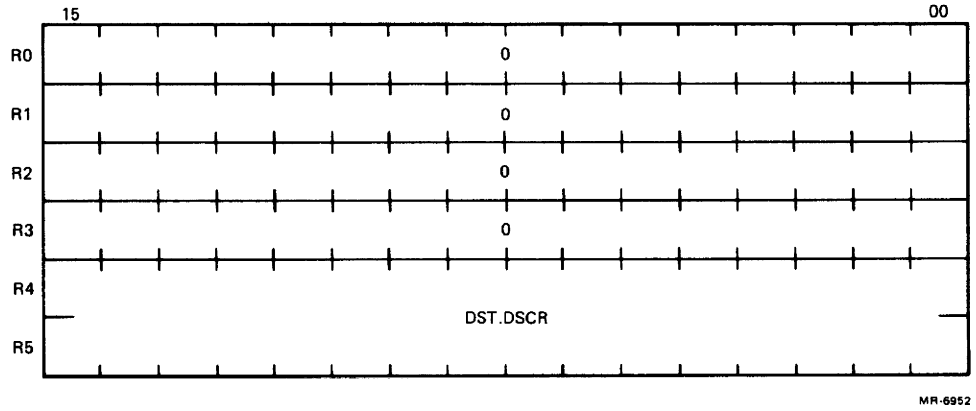


Figure 14-50 Multiply Decimal Format (Cleared)

In-Line Form - MULPI
Each word address pointer following the op-code word in the instruction stream refers to a 2-word decimal string descriptor. R0–R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings, provided that each source string is a valid representation of the specified data type.

2. The results of the instruction are unpredictable if the source and destination strings overlap.

3. No numeric string multiply instruction is provided.

14.7.17 SCANC/SCANCI

Purpose: Scan Character

Operation: Search source character string for a member of the character set.

Condition Codes: The condition codes are based on the final contents of R0.

N: set if R0<15> is set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

Op Codes:

SCANC	076042
SCANCI	076142

Description: The source character string is searched from most significant to least significant character until the first occurrence of a character that is a member of the character set. A character string descriptor is returned in R0–R1 that represents the portion of the source character string, beginning with the located member of the character set. If the source character string contains only characters that are not

in the character set, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form - SCANC
When the instruction starts, the operands must have been placed in the general registers: the source character string descriptor must be in R0–R1, and the character set descriptor in R4–R5. The scan character format is shown in Figure 14-51.

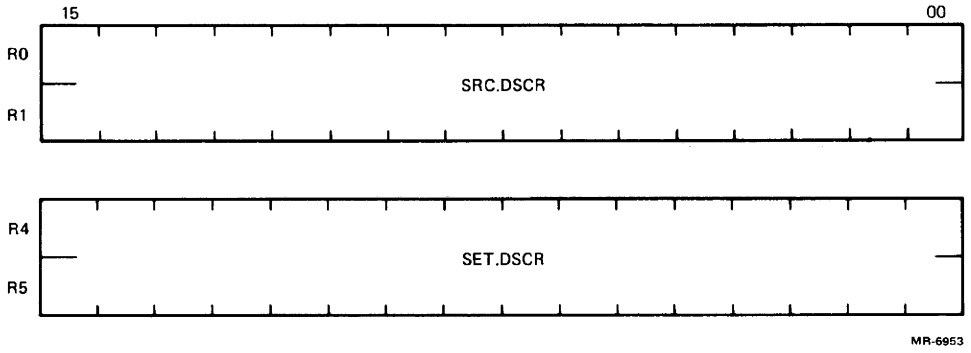


Figure 14-51 Scan Character Format

When the instruction is completed, R0–R1 contain a character string descriptor that represents the substring of the source character string, beginning with the most significant character that is a member of the character set. The format of the scan character after termination is shown in Figure 14-52.

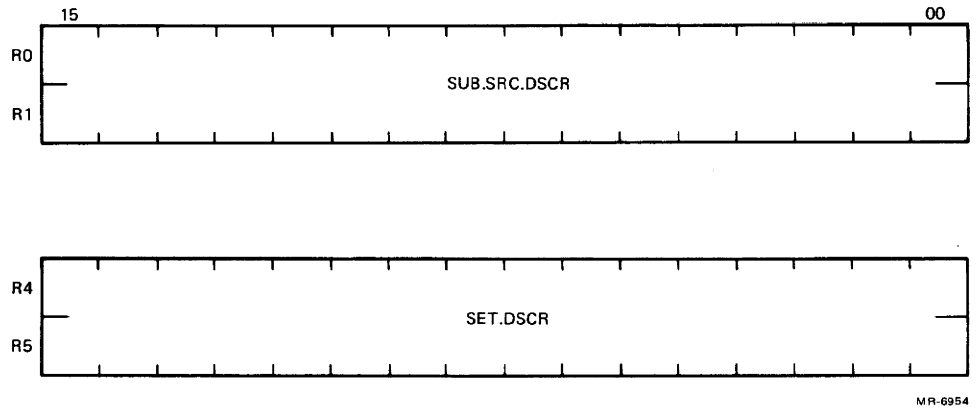


Figure 14-52 Scan Character Termination Format

In-Line Form - SCANCI
The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string source descriptor, and a word address pointer to a 2-word character set descriptor. When the instruction is completed, R0–R1 contain a character string descriptor that represents the substring of the source character string, beginning with the most significant character that is a member of the character set. R2–R6 are unchanged. The in-line format of the scan character is shown in Figure 14-53.

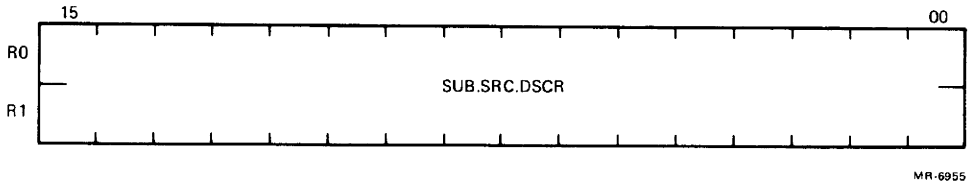


Figure 14-53 Scan Character Format (In-Line)

Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that no character in the set was found. The original source character string descriptor is returned in R0–R1.

2. The source character string and character set table may overlap in any way.

3. A test for success is BNE; a test for failure is BEQ.

4. The condition codes will be set as if this instruction were followed by TST R0.

5. The effect of the instruction is unpredictable if the entire 256-byte character set table is not in readable memory.

14.7.18 SKPC/SKPCI

Purpose: Skip Character

Operation: Search source character string until a character other than the search character is found.

Condition Codes: The condition codes are based on the final contents of R0.

N: set if R0<15> is set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

Op Codes:

SKPC	076041
SKPCI	076141

Description: The source character string is searched from most significant to least significant character until the first occurrence of a character that is not the search character. A character string descriptor is returned in R0–R1 that represents the portion of the source character string, beginning with the most significant character that was not equal to the search character. If the source character string contains only characters equal to the search character, the instruction returns a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form - SKPC
When the instruction starts, the operands must have been placed in the general registers: the source character string descriptor must be in R0–R1, the search character in R4<7:0>, and R4<15:8> must be zero. The format of the register form of a skip character instruction is shown in Figure 14-54.

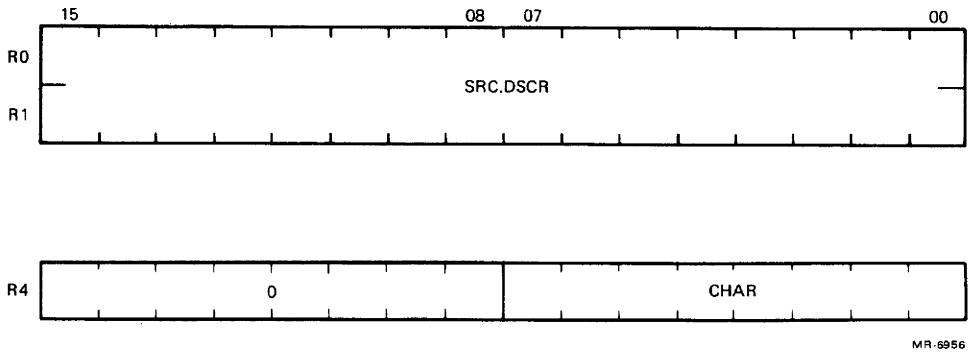


Figure 14-54 Skip Character Format (Register Form)

When the instruction is completed, R0–R1 contain a character string descriptor that represents the substring of the source character string, beginning with the most significant character that was not equal to the search character. The format of the skip character after termination is shown in Figure 14-55.

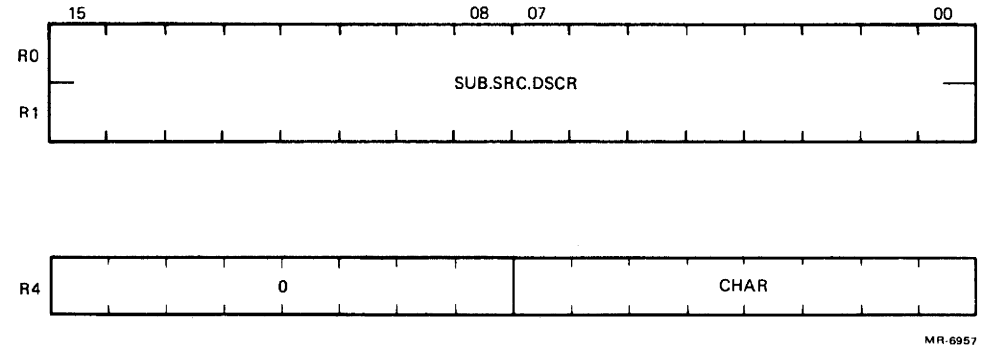


Figure 14-55 Skip Character Termination Format

In-Line Form - SKPCI
The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string source descriptor, and a word whose low-order half contains the search character and whose high-order half must be zero. When the instruction is completed, R0–R1 contain a character string descriptor that represents the substring of the source character string, beginning with the most significant character that was not equal to the search character. R2–R6 are unchanged. The format of the in-line form of the skip character is shown in Figure 14-56.

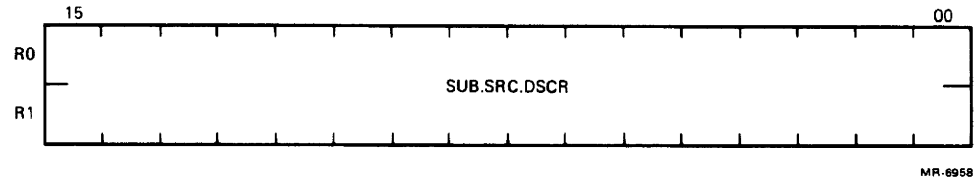


Figure 14-56 Skip Character Format (In-Line)

Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating the character string only contained search characters. The original source character string descriptor is returned in R0–R1.

2. The condition codes will be set as if this instruction were followed by TST R0.

14.7.19 SPANC/SPANCI

Purpose: Span Character

Operation: Search source character string for a character that is not a member of the character set.

Condition Codes: The condition codes are based on the final contents of R0.

N: set if R0<15> is set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

Op Codes:

SPANC	076043
SPANCI	076143

Description: The source character string is searched from most significant to least significant character until the first occurrence of a character that is not a member of the character set. A character string descriptor is returned in R0–R1 that represents the portion of the source character string, beginning with the character that is not a member of the character set. If the source character string contains only characters that are in the character set, the instruction returns a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value of R0.

Register Form - SPANC
When the instruction starts, the operands must have been placed in the general registers: the source character string descriptor must be in R0–R1, and the character set descriptor in R4–R5. The format of the register form of the span character is shown in Figure 14-57.

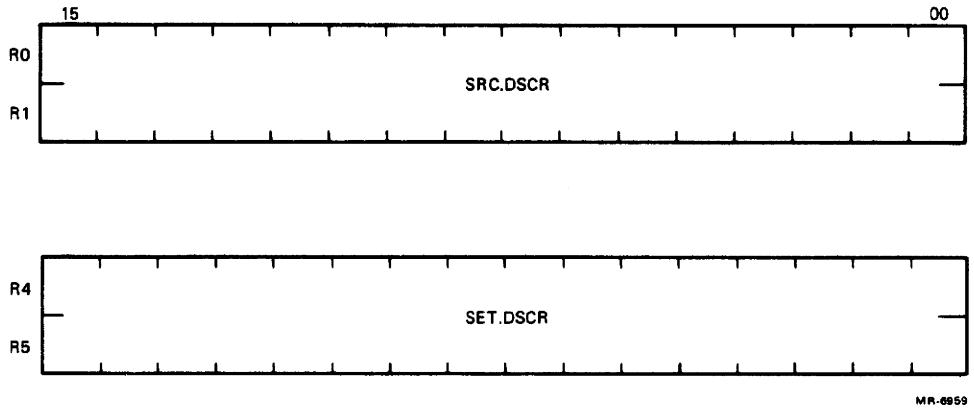


Figure 14-57 Span Character Format (Register Form)

When the instruction is completed, R0–R1 contain a character string descriptor that represents the substring of the source character string, beginning with the most significant character that is not a member of the character set. The format of the span character after termination is shown in Figure 14-58.

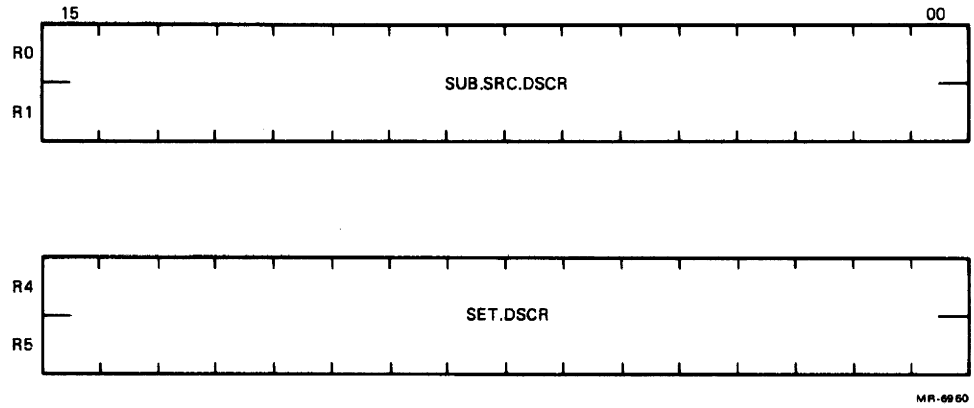


Figure 14-58 Span Character Termination Format

In-Line Form - SPANCI

The words following the op-code word in the instruction stream are a word address pointer to a 2-word character string source descriptor, and a word address pointer to a 2-word character set descriptor. When the instruction is completed, R0–R1 contain a character string descriptor that represents the substring of the source character string, beginning with the most significant character that is not a member of the character set. R2–R6 are unchanged. The format of the in-line form of the span character is shown in Figure 14-59

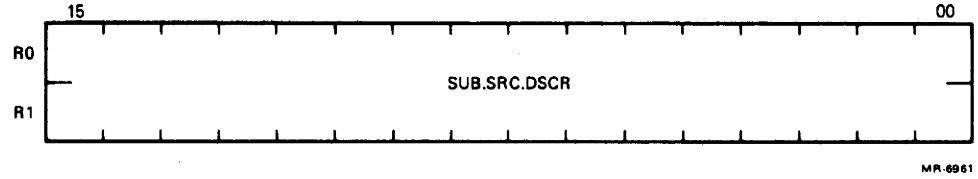


Figure 14-59 Span Character Format (In-Line)

Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that only characters in the set were found. The original source character string descriptor is returned in R0–R1.

2. The source character string and character set table may overlap in any way.

3. The condition codes will be set as if this instruction were followed by TST R0.

4. The effect of the instruction is unpredictable if the entire 256-byte character set table is not in readable memory.

14.7.20 SUBN/SUBP/SUBNI/SUBPI

Purpose: Subtract Decimal

Operation: dst src2 − src1

Condition Codes:
N: set if dst < 0; cleared otherwise
Z: set if dst = 0; cleared otherwise
V: set if dst cannot contain all significant digits of the result; cleared otherwise
C: cleared

Op Codes:

SUBN	076051
SUBP	076071
SUBNI	076151
SUBPI	076171

Description: Src1 is subtracted from src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form - SUBN and SUBP
When the instruction starts, the operands must have been placed in the general registers: the first source descriptor must be in R0–R1, the second source descriptor in R2–R3, and the destination descriptor in R4–R5. The subtract decimal format is shown in Figure 14-60.

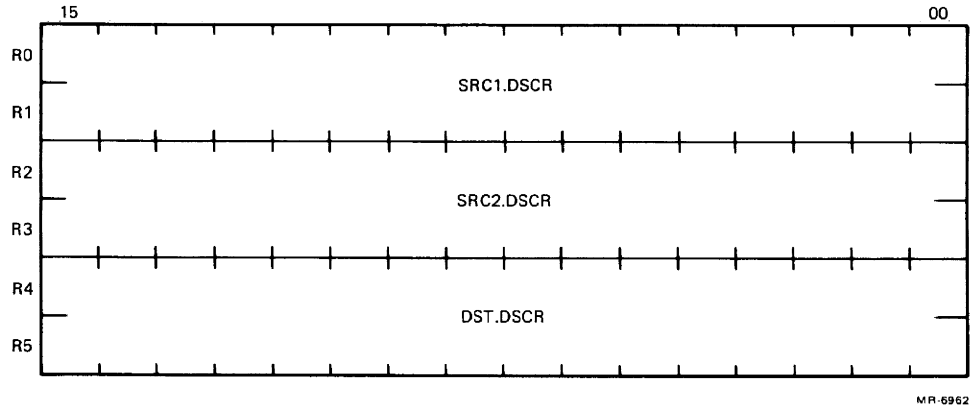


Figure 14-60 Subtract Decimal Format

When the instruction is completed, the source descriptor registers are cleared, as shown in Figure 14-61.

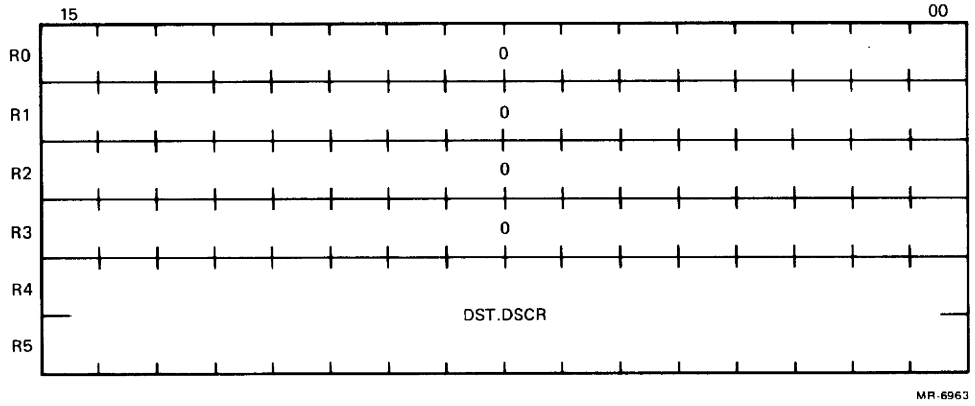


Figure 14-61 Subtract Decimal Format (Cleared)

In-Line Form – SUBNI and SUBPI
Each word address pointer that follows the op-code word in the instruction stream refers to a 2-word decimal string descriptor. R0–R6 are unchanged when the instruction is completed.

Notes:

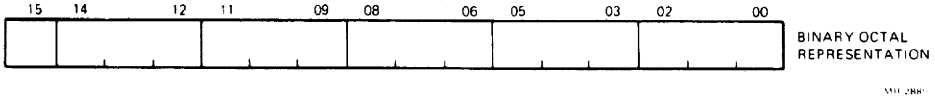
1. The operation of these instructions is unaffected by any overlap of the source strings, provided that each source string is a valid representation of the specified data type.

2. Source strings may overlap the destination string only if all corresponding digits of the strings are in coincident bytes in memory.

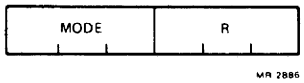
APPENDIX A
GENERAL REFERENCE INFORMATION

A.1 SUMMARY OF KDF11 INSTRUCTIONS

WORD FORMAT

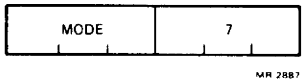


ADDRESSING MODES



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

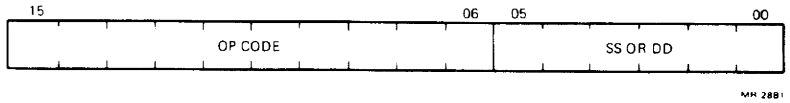
Boolean

< = AND
> = inclusive OR
> = exclusive OR
~ = NOT

Condition Codes

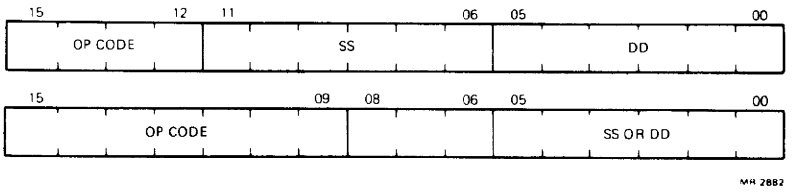
* = conditionally set/cleared
— = not affected
0 = cleared
1 = set

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

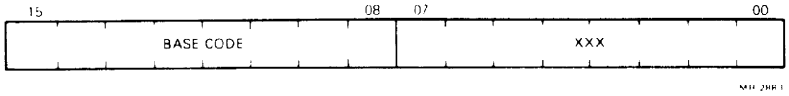


Mnemonic	Op Code	Instruction	Operation	N	Z	V	C
General							
MOV(B)	■ 1SSDD	move	d ← s	*	*	0	–
CMP(B)	■ 2SSDD	comapare	s – d	*	*	*	*
ADD	06SSDD	add	d ← s + d	*	*	*	*
SUB	16SSDD	subtract	d ← d – s	*	*	*	*
Logical							
BIT(B)	■ 3SSDD	bit test (AND)	s d	*	*	0	–
BIC(B)	■ 4SSDD	bit clear	d ← (~s) d	*	*	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)

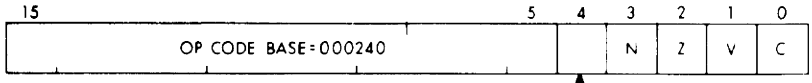
TRAP & INTERRUPT:

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:

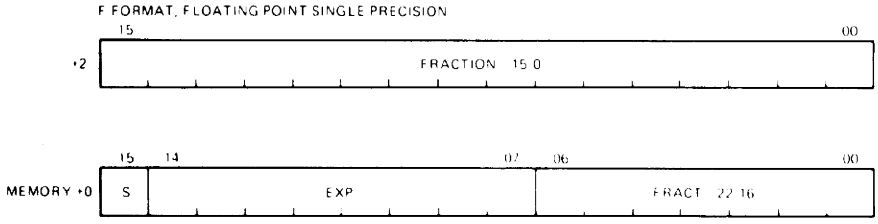


0 = CLEAR SELECTED COND. CODE BITS
1 = SET SELECTED COND. CODE BITS

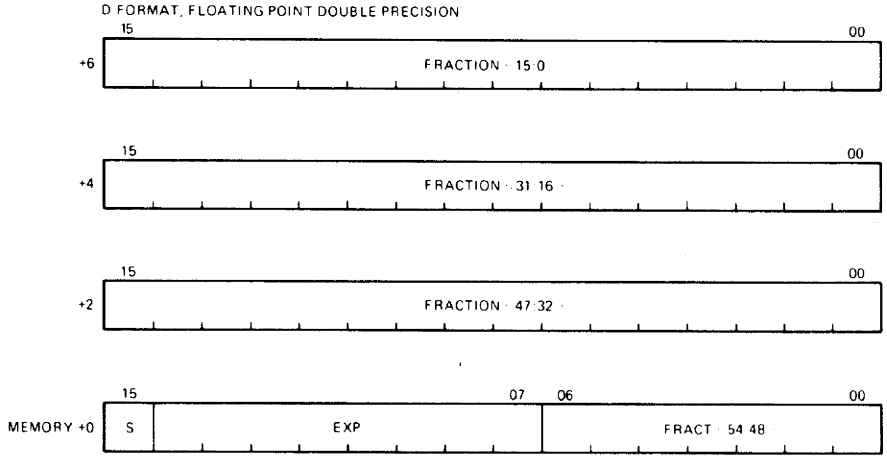
Mnemonic	Op Code	Instruction	N	Z	V	C
CLC	000241	clear C	–	–	–	0
CLV	000242	clear V	–	–	0	–
CLZ	000244	clear Z	–	0	–	–
CLN	000250	clear N	0	–	–	–
CCC	000257	clear all cc bits	0	0	0	0
SEC	000261	set C	–	–	–	1
SEV	000262	set V	–	–	1	–
SEZ	000264	set Z	–	1	–	–
SEN	000270	set N	1	–	–	–
SCC	000277	set all cc bits	1	1	1	1

OPTIONAL FLOATING POINT:

Data Formats



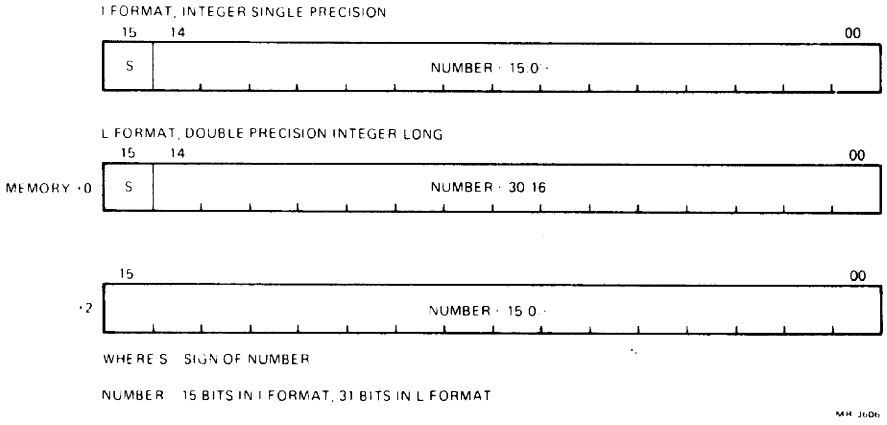
OPTIONAL FLOATING POINT:
Data Formats (Cont)



S = SIGN OF FRACTION

EXP = EXPONENT IN EXCESS 200 NOTATION, RESTRICTED TO 1 TO 377 OCTAL FOR NON-VANISHING NUMBERS

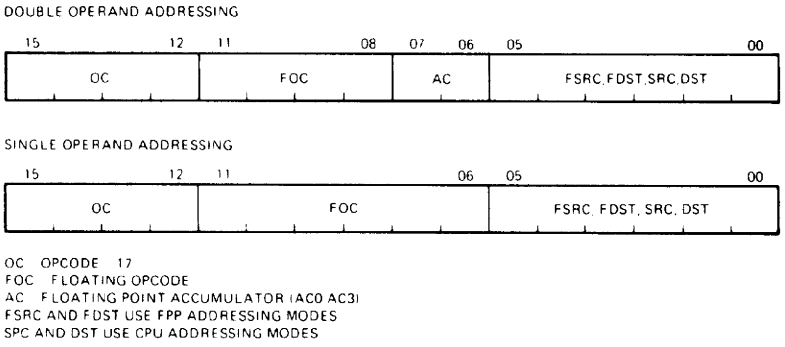
FRACTION = 23 BITS IN F FORMAT, 55 BITS IN D FORMAT + ONE HIDDEN BIT (NORMALIZATION). THE BINARY RADIX POINT IS TO THE LEFT



WHERE S SIGN OF NUMBER

NUMBER 15 BITS IN I FORMAT, 31 BITS IN L FORMAT

Addressing Formats



OC OPCODE 17
FOC FLOATING OPCODE
AC FLOATING POINT ACCUMULATOR (AC0 AC3)
FSRC AND FDST USE FPP ADDRESSING MODES
SRC 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)	

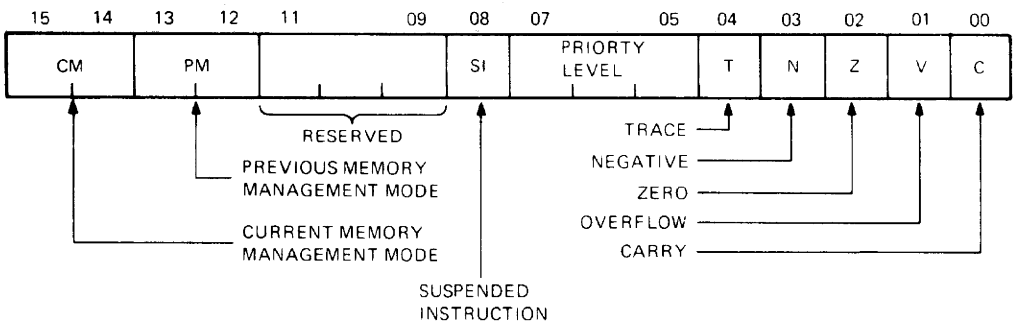
A.2 NUMERICAL OP CODE LIST

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	MFPT
00 00 10 – 00 00 77	(unused)
00 01 DD	JMP
00 02 0R	RTS
00 02 10 – 00 02 27	(reserved)
00 02 40	NOP
00 02 41 – 00 02 77	cond codes
00 03 DD	SWAB
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 – 10 43 77	EMT

Op Code	Mnemonic
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	ASL
00 64 NN	MARK
00 66 SS	MFPI
00 66 DD	MTPi
00 67 DD	SXT
10 44 00 – 10 47 77	TRAP
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

Op Code	Mnemonic
00 70 00 – 00 77 77	(unused)
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	FADD
07 50 1R	FSUB
07 50 2R	FMUL
07 50 3R	FDIV
07 50 40 – 07 67 77	(reserved)
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 – 17 77 77	floating point

A.3 PROCESSOR STATUS WORD (PS) 17777776



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 AND VECTORS

Device	Register	Device Address	Interrupt Vector
Console Terminal			
Input Control/Status	RCSR	17777560	60
Input Buffer	RBUF	17777562	
Output Control/Status	XCSR	17777564	64
Output Buffer	XBUF	17777566	
2nd SLU Terminal			
Input Control/Status	RCSR	17776500*	300
		17776540**	340
Input Buffer	RBUF	17776502*	
		17776542**	
Output Control/Status	XCSR	17776504*	304
		17776544**	344
Output Buffer	XBUF	17776506*	
		17776546**	
KDF11-B Boot/Diagnostic			
Page Control	PCR	17777520	
Read Control	RWR	17777522	
Lights Switches	CDR	17777524	
Unused		17777526	
Boot/Diagnostic ROM		17773000–	
		17774000	
Line Frequency Clock	LKS	17777546	100
RLV12 Disk			160
Status	CSR	17774400	
Bus Address	BAR	17774402	
Disk Address	DAR	17774404	
Multipurpose	MPR	17774406	
Bus Address Extension	BAE	17777546	
LPV11 High Speed Printer			200
Printer Status	LPS	17777514	
Printer Buffer	LPB	17777516	

*J13 and J12 must be ungrounded.
**J13 must be ungrounded and J12 must be grounded.

MMU Status Registers

MMU	Register	Address
Status Register 0	SR0	17777572
Status Register 1	SR1	17777574
Status Register 2	SR2	17777576
Status Register 3	SR3	17772516

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

RESERVED TRAP and INTERRUPT VECTORS

Vector	Description
000	(Reserved)
004	Bus Timeout and Illegal Instructions (e.g., JMP R0) (Odd Address Trap Not Implemented on LSI-11/23)
010	Illegal and Reserved Instruction
014	BPT Instruction and T Bit
020	IOT Instruction
024	Power Fail
030	EMT Instruction
034	TRAP Instruction
060	Console Input Device
064	Console Output Device
100	External Event Line Interrupt
114	Parity Error
160	RLV12
200	LPV 11
244	KEF11-A (Floating Point)
250	Memory Management Abort
264	RXV11, RXV21
300	Floating Vectors start here

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 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.

* All addresses in ODT must be 18-bit addresses between 0 and 777776.

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 GENERAL INFORMATION

The KDF11-BA CPU executes PDP-11 instructions as a series of microcode cycles. A data fetch consists of an address cycle and a bus DIN cycle. A data write consists of an address cycle and a bus DOUT cycle. An instruction fetch consists of an address cycle, a bus DIN cycle, and a non-I/O cycle. The execution of an instruction typically consists of one or more non-I/O cycles. Floating-point instructions also include interchip DIN and DOUT cycles that move data between the DATA and MMU chips. The execution time for an instruction depends on the type of the instruction, the modes of addressing used, the type of memory referenced and whether the memory management unit is enabled or disabled.

Each microcode cycle consists of an integral number of clock pulses, one occurring every 75 ns. The number of clock pulses and the time required to complete the most common microcode cycles are listed in Table B-1. The time required for bus DIN or DOUT microcode cycles accessing either the memory management registers (MMU DIN or DOUT) or the KDF11-BA on-board peripherals (IDAL bus DIN or DOUT) are listed in Table B-2. The KDF11-BA module's peripherals include the bootstrap and diagnostic ROMs, the line clock logic, and the serial-line units.

1. The KDF11-BA detects RRPLY assertion within 112.5 ns of the time it asserts TDIN. (This is typical for peripherals that assert TRPLY as soon as they receive RDIN asserted.)

2. The KDF11-BA detects RRPLY assertion within 337.5 ns of the time it asserts TSYNC. (This is typical for the MSV11-P parity memory.)

3. The KDF11-BA detects RRPLY assertion within 150 ns of the time it asserts TDOUT. (This is typical for peripherals and memories that assert TRPLY as soon as they receive RDOUT asserted. This includes the MSV11-P parity memory.)

Table B-1 KDF11-BA Common Microcode Cycle Times

Type of Cycle	Clock Pulses	Time (ns)
Address (no relocation)	5	375
Address (relocation)	8	600
LSI-11 bus DIN (1)	10	750
LSI-11 bus DIN (2)	11	825
LSI-11 bus DOUT (3)	11	825
Interchip DIN	5	375
Interchip DOUT	5	375
Non-I/O (micro-NOP)	4	300

Table B-2 KDF11-BA Peripheral Microcode Cycle Times

Type of Cycle	Clock Pulses	Time (ns)
IDAL bus DIN	8	600
IDAL bus DOUT	8	600
MMU DIN	7	525
MMU DOUT	7	525

B.2 BASIC INSTRUCTION TIMING

The source, destination, and fetch/execute times for the KDF11-BA basic instruction set appear below. KDF11-BA instruction times are calculated using the following equation:

Instruction Time = Basic Time + Source Time + Destination Time
(Basic Time = Fetch Time + Execute Time)

The basic, source, and destination times were calculated from the microcode cycle times listed in Table B-1. LSI-11 bus DIN (2) and DOUT (3) times of 825 ns were used for the MSV11-P parity memory, which has its specifications listed in Table B-3.

The instruction execution times for systems with memory management enabled or disabled are listed in Tables B-4 through B-7.

Table B-3 MSV11-P Parity Memory

Bus Cycle	Access Time (ns)		Cycle Time (ns)	
	Typical	Maximum	Typical	Maximum
DATI	240	260	560	590
DATO(B)	90	120	610	640
DATIO(B)	660	690	1175	1210

Table B-4 Source Address Times

Instructions	Source Mode	Memory Cycles	Time (μs) with Memory Management	
			Enabled	Disabled
ADD, SUB	0	0	0	0
MOV(B), CMP(B)	1	1	1.425	1.200
BIS(B), BIC(B)	2	1	1.425	1.200
BIT(B)	3	2	2.850	2.400
	4	1	1.725	1.500
	5	2	3.150	2.700
	6	2	3.150	2.700
	7	3	4.575	3.900

Table B-4 Source Address Times (Cont)

Instructions	Source Mode	Memory Cycles	Time (μs) with Memory Management Enabled	Disabled
MUL,DIV	0	0	1.275	1.275
ASH, ASHC	1	1	1.725	1.450
MFPI, MFPD	2	1	1.725	1.450
MTPS	3	2	2.850	2.400
	4	1	1.725	1.500
	5	2	3.150	2.700
	6	2	3.150	2.700
	7	3	4.575	3.900

Table B-5 Destination Address Times

Instructions	Source Mode	Memory Cycles	Time (μs) with Memory Management Enabled	Disabled
MOV(B), CLR(B)	0	0	0	0
SXT, MFPS	1	1	2.025	1.800
MTPI, MTPD	2	1	2.025	1.800
	3	2	3.150	2.700
	4	1	2.025	1.950
	5	2	3.450	3.000
	6	2	3.450	3.000
	7	3	4.875	4.500
CMP(B), BIT(B)	0	0	0	0
TST(B)	1	1	1.725	1.500
	2	1	1.725	1.500
	3	2	2.850	2.400
	4	1	1.725	1.500
	5	2	3.150	2.700
	6	2	3.150	2.700
	7	3	4.575	3.900
ADD, SUB	0	0	0	0
INC(B), DEC(B)	1	1	2.850	2.625
COM(B), NEG(B)	2	1	2.850	2.625
ROR(B), ROL(B)	3	2	4.275	3.825
ASR(B), ASL(B)	4	1	2.850	2.625
BIS(B), BIC(B)	5	2	4.575	4.125
ADC, SBC	6	2	4.575	4.125
XOR, SWAB	7	3	6.000	5.325

Table B-6 Basic (Fetch and Execute) Times

Instructions	Memory Cycles	Time (μs) with Memory Management Enabled	Disabled
MOU, CMP, BIT, BIC, BIS, ADD, SUB, SXT, CLR, TST, COM, INC, DEC, NEG, ADC, SBC, ROR, ROL, ASR, ASL, SWAB, MFPS	1	2.025	1.800
MTPS	1	3.600	3.375
MFPI, MFPD	2	4.050	3.600
MTPI, MTPD	2	4.725	4.275
SOB (NO BRANCH)	1	2.625	2.400
SOB (BRANCH)	1	2.925	2.700
ALL BRANCH	1	2.025	1.800
CLN, CLE, CLV, CLC, SEN, SEZ, SEV, SEC, CCC, SCC	1	2.925	2.700
RTS	2	3.750	3.300
MARK	2	5.325	4.875
RTI	3	6.225	5.550
RTT	3	7.500	6.825
IOT, BPT	5	10.500	9.375
EMT, TRAP	5	9.525	8.850
WAIT	1	3.375	3.150
MUL	1	33.300	33.075
DIV	1	49.650	49.425
ASH	1	24.825	24.600
ASHC	1	46.050	45.825

NOTES

1. The instruction times for MUL, DIV, ASH, and ASHC are operand-dependent and could be less than the values given above.

2. The instruction times for the RESET and HALT instructions are mode/option-dependent.

Table B-7 Jump Instruction Times

Instruction	Dest. Mode	Memory Cycles	Time (μs) with Memory Management Enabled	Disabled
JMP	1	1	2.325	2.100
	2	1	2.625	2.400
	3	2	3.450	3.000
	4	1	2.625	2.400
	5	2	3.750	3.300
	6	2	3.750	3.300
	7	3	5.175	4.500
JSR	1	2	4.350	3.900
	2	2	4.650	4.200
	3	3	5.475	4.800
	4	2	4.650	4.200
	5	3	5.775	5.100
	6	3	5.775	5.100
	7	4	7.200	6.300

B.3 DMA AND INTERRUPT LATENCIES

DMA latency is the time required for the first DMA device to obtain bus mastership after it asserts a direct memory access request (BDMR L). The DMA latency is 1.35 μs (maximum). The maximum DMA latency was calculated for a relocated address cycle followed by a DOUT cycle. The processor disables DMA grant (BDMGO L) from the end of the address cycle phase time for four 75 ns intervals after the DOUT cycle phase time.

Interrupts (BR requests) are acknowledged by the processor at the end of the current instruction. Interrupt latency is defined as the time required by the KDF11-BA to assert an interrupt acknowledge (BIAKO L) after it receives an interrupt request. Interrupt service time is defined as the time required by the processor to fetch the first service routine instruction after the assertion of BIAKO L. The interrupt latency time and the interrupt service time must be added to obtain the total time from the reception of the interrupt request to the fetch of the first service routine instruction. The specifications for interrupt latency and interrupt service times are as follows.

Interrupt Latency	5.475 μs, typical [MOV X(R7),R0]
	12.600 μs, maximum (except EIS)
	54.225 μs, maximum (including EIS)
Interrupt Service	8.625 μs (memory management Off)
	9.750 μs (memory management On)

NOTE

1. **Interrupt and DMA latencies assume a KDF11-BA with memory management enabled and using MSV11-P memory.**

2. **The maximum interrupt latencies were calculated for ADD @X(R7), @Y(R7), and DIV @X(R7).**

APPENDIX C
LSI-11, KDF11/PDP-11 PROGRAMMING AND OPERATIONAL DIFFERENCES

The table on the following pages compares the programming and operational features of the KDF11-BA, KDF11-AA, LSI-11, and PDP-11 processors.

ACTIVITY	LSI-11	KDF11-AA and BA	PDP-11/04	PDP-11/34	PDP-11/05/10	PDP-11/15/20	PDP-11/35/40	PDP-11/45
1. OPR%R, (R)+ or OPR%R, and –(R) using the same register as both source and destination: contents of R are incremented (or decremented) by 2 before being used as the source operand.		X				X	X	
OPR%R, (R)+ or OPR%R, and –(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, and @–(R) using the same register as both source and destination: contents of R are incremented (or decremented) by 2 before being used as the source operand.		X				X	X	
OPR%R, @(R)+ or OPR%R, and @–(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	

ACTIVITY	LSI-11	KDF11-AA and BA	PDP-11/04	PDP-11/34	PDP-11/05/10	PDP-11/15/20	PDP-11/35/40	PDP-11/45
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 *not* 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			
Register addresses (177700–177717) timeout 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

ACTIVITY	LSI-11	KDF11-AA and BA	PDP-11/04	34	05/10	15/20	35/40	45
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 PDP-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 ms). RESET instruction consists of a 70 ms pause with INIT occurring during the first 20 ms.						X	X	
Power-fail immediately ends the RESET instruction and traps if an INIT is in progress. A minimum INIT of 1 μs occurs if instruction aborted.								X
Power-fail acts the same as in the PDP-11/45 (22 ms with about 300 ns minimum). Power-fail during RESET fetch is fatal with no power-down sequence.			X	X	X			
RESET instruction consists of 10 μs of INIT followed by a 90 μs pause. Power-fail is not recognized until the instruction is complete.	X	X						

ACTIVITY	LSI-11	KDF11-AA and BA	PDP-11/04	PDP-11/34	PDP-11/05/10	PDP-11/15/20	PDP-11/35/40	PDP-11/45
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 RTI sets the T bit, T bit trap is acknowledged after the instruction following RTI.					X	X		
If RTI sets the 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 the 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 F11.	X		X	X	X	X		

	ACTIVITY	LSI-11	KDF11-AA and BA	PDP-11/04	34	05/10	15/20	35/40	45
	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		
17.	Eight general-purpose registers.	X	X	X	X	X	X	X	
	Sixteen general-purpose registers.								X
18.	PS address, 177776, not implemented; must use new instructions, MTPS (move to PS) and MFPS (move from PS).	X							
	PS address implemented, MTPS and MFPS not implemented.			X		X	X	X	X
	PS 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

ACTIVITY	LSI-11	KDF11-AA and BA	PDP-11/04	PDP-11/34	PDP-11/05/10	PDP-11/15/20	PDP-11/35/40	PDP-11/45
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 a source operand.	X							
EIS instructions do a DATI bus sequence when fetching a source operand.		X						
26. MOV instruction just does a DATO bus sequence for the last memory cycle.	X	X		X			X	X
MOV instruction does a DATIP and DATO bus sequence for the last memory cycle.				X	X	X		
27. If the PC contains a nonexistent memory address and a bus error occurs, the PC will have been incremented.	X	X	X	X	X	X		X

ACTIVITY	LSI-11	KDF11-AA and BA	PDP-11/04	PDP-11/34	PDP-11/05/10	PDP-11/15/20	PDP-11/35/40	PDP-11/45
If the PC contains nonexistent memory address and a bus error occurs, the PC will be unchanged.							X	
28. If a register contains nonexistent memory address in mode 2 and a bus error occurs, the register will be incremented.	X	X			X	X	X	X
Same as above but the register is unchanged.			X	X				
29. If a register contains an odd value in mode 2 and a bus error occurs, the register will be incremented.	X	X					X	X
If a register contains an odd value in mode 2 and a bus error occurs, the register will be unchanged.			X	X	X	X		
30. Condition codes restored to original values after FIS interrupt abort (EIS does not abort on the PDP-35/40).							X	
Condition codes that are restored after EIS/FIS interrupt abort are indeterminate.	X							
31. Op codes 075040–075377 unconditionally trap to 10 as reserved op codes.		X	X	X	X	X	X	X
If the KEV-11 option is present, op codes 075040–075377 perform a memory read using the register specified by the low-order three bits as a pointer. If the register contents are a nonexistent 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							

ACTIVITY	LSI-11	KDF11-AA and BA	PDP-11/04	PDP-11/34	PDP-11/05/10	PDP-11/15/20	PDP-11/35/40	PDP-11/45
32. Op codes 210–217 trap to 10 as reserved op codes.		X	X	X	X	X	X	X
Op codes 210–217 are used as maintenance instructions.	X							
33. Op codes 075040–075777 trap to 10 as reserved op codes.		X	X	X	X	X	X	X
Only if KEV-11 option is present, op codes 075040–075377 can be used as escapes to user microcode. Op codes 075400–075777 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 170000–177777 trap to 10 as reserved instructions.				X	X	X	X	
Op codes 170000–177777 are implemented as floating-point instructions.			X	X				X
Op codes 170000–177777 can be used as escapes to user microcode. If no user microcode exists, a trap-to-10 occurs.	X							

APPENDIX D
KDF11-BA BACKPLANE PIN ASSIGNMENT COMPARISON

The KDF11-BA module (M8189) uses four bused spare lines that were reserved for future expansion to implement 22-bit addressing. The KDF11-BA also uses two spare pins for the RUN light signal and one spare pin for battery backup control of the power-up code 1 jumper signal (PUP CD1J L). The KDF11-BA uses the AM2 pin in slot 1 for the input of a microcycle enable signal (MCENB H), which may be externally negated to disable the master clock control for testing purposes. Pin AM2 in slots 2 through 9 is used by peripheral option modules as an input pin for the BIAKI signal. Two pairs of CD slot signals can be connected together to provide continuity for the interrupt acknowledge (BIAK) and bus grant (BDMG) signals when the KDF11-BA is used in an LSI-11/LSI-11 backplane.

Certain pins of the A and B backplane rows are used for different functions by the KDF11-BA, KDF11-AA, KD11-HA, and KD11-F processors. A comparison of the backplane pin assignment for the processors is shown in Table D-1. The assignment of the remaining backplane pins of rows A and B are identical for all four processors. The backplane pin assignment for rows C and D of the KDF11-BA module is listed in Table D-2.

Table D-1 Backplane Pin Assignment Comparison (Rows A and B)

Bus Pin	Backplane Signal Name	Backplane Pin Utilization KDF11-BA	KDF11-AA	FD11-HA	KD11-F
AA1	BIRQ5 L	BIRQ5 L	BIRQ5 L	Not used	Not used
AB1	BIRQ6 L	BIRQ6 L	BIRQ6 L	Not used	Not used
BP1	BIRQ7 L	BIRQ7 L	BIRQ7 L	Not used	Not used
AC1	BDAL16 L	BDAL16 L	BDAL16 L	Not used	Not used
AD1	BDAL17 L	BDAL17 L	BDAL17 L	Not used	Not used
AE1	SSPARE1	Not used	Single Step	STOP L	Not used
AF1	SSPARE2	SRUN L	SRUN L	SRUN L	SRUN L
AH1	SSPARE3	SRUN L	SRUN L	SRUN L	Not used
AK1	MSPAREA	Not used	Not used	MTOE L	Not used
AL1	MSPAREA	Not used	Not used	GND	Not used
AM2	BIAKI L	MCENB H	MMU STRH	Not used	Not used
AR1	BREF L	BCTRL L	Not used*	Not used*	BREF L
AR2	BDMGI L	Not used	UBMAP L	Not used	Not used
BC1	SSPARE4	BDAL18 L	MMU DAL18 H	SCLK3 H	Not used
BD1	SSPARE5	BDAL19 L	MMU DAL19 H	SWMIB18 H	Not used

*Not used on the KDF11-AA and KD11-HA but terminated in the inactive state to prevent problems with older memories.

Table D-1 Backplane Pin Assignment Comparison (Rows A and B) (Cont)

Bus Pin	Backplane Signal Name	Backplane Pin Utilization KDF11-BA	KDF11-AA	FD11-HA	KD11-F
BE1	SSPARE6	BDAL20 L	MMU DAL20 H	SWMIB19 H	Not used
BF1	SSPARE7	BDAL21 L	MMU DAL21 H	SWMIB20 H	Not used
BH1	SSPARE8	PUP CD1J L	CLK DISL	SWMIB21 H	Not used
BK1	MSPAREB	Not used	Not used	Not used	4K RAM BIAS
BL1	MSPAREB	Not used	Not used	Not used	4K RAM BIAS

Table D-2 KDF11-BA Backplane Pin Assignment (Rows C and D)

Bus Pin	Pin Utilization	Bus Pin	Pin Utilization	Bus Pin	Pin Utilization	Bus Pin	Pin Utilization
CA1	–	CA2	+5	DA1	–	DA2	+5
CB1	–	CB2	–	DB1	–	DB2	–
CC1	–	CC2	GND	DC1	–	DC2	GND
CD1	–	CD2	–	DD1	–	DD2	–
CE1	–	CE2	–	DE1	–	DE2	–
CF1	–	CF2	–	DF1	–	DF2	–
CH1	–	CH2	–	DH1	–	DH2	–
CJ1	–	CJ2	–	DJ1	–	DJ2	–
CK1	–	CK2	–	DK1	–	DK2	–
CL1	–	CL2	–	DL1	–	DL2	–
CM1	–	CM2	BIAKI L	DM1	–	DM2	–
CN1	–	CN2	BIAKO L	DN1	–	DN2	–
CP1	–	CP2	–	DP1	–	DP2	–
CR1	–	CR2	BDMGI L	DR1	–	DR2	–
CS1	–	CS2	BDMGO L	DS1	–	DS2	–
CT1	GND	CT2	–	DT1	GND	DT2	–
CU1	–	CU2	–	DU1	–	DU2	–
CV1	–	CV2	–	DV1	–	DV2	–

APPENDIX E
MICRO-ODT DIFFERENCES

A number of differences exist between the ways the LSI-11 (KD11-F), LSI-11/2 (KD11-HA), LSI-11/23 (KDF11-A) and LSI-11/23B (KDF11-BA) CPUs interpret the same console ODT commands. Notably, the LSI-11/23 and LSI-11/23B do not support the L command. The following list describes these differences.

In most cases, if you are using ODT from a console terminal, your program will not be affected. However, the slight difference in response to some commands may impact users who have programmed a host computer to emulate a console terminal to down-line load programs to the LSI-11.

LSI-11 and LSI-11/2
(KD11-F and KD11-HA)

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 printhead has completely returned.

2. When an address location is open, another location can be opened without explicitly closing the first location. For example, 1000/123456 2000/054321

3. "↑" will open the previous location.

4. "@" will open a location using indirect addressing.

5. "←" will open a location using relative addressing.

LSI-11/23 and LSI-11/23B
(KDF11-AA and KDF11-BA)

1. 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 <LF>s, nulls (0), STXs (2), and BSx (10) because an automatic <CR> and <LF> follow. In the APT command mode, no input characters are echoed.

2. 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 its contents being altered.

3. "↑" is illegal and micro-ODT prints ?<CR><LF>@.

4. "@" is illegal and micro-ODT prints ?<CR><LF>@.

5. "?" is illegal and micro-ODT prints ?<CR><LF>@.

LSI-11/23 and LSI-11/23B (KDF11-AA and KDF11-BA)

6. "M" will print the contents of an internal CPU register.

7. Rubout (ASCII 177) will delete the last character typed in.

8. "L" is the boot loader command that will load the absolute loader from the specified device.

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.

10. Up to a 16-bit address and 16-bit data may be entered. Leading zeros are assumed.

11. Incrementing <LF>, the address 177776 results in the address 000000.

12. Incrementing a PDP-11 register from R7 prints out "R8" and the contents of R0.

13. The I/O page is in the address group 17XXXX.

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.

LSI-11 and LSI-11/2 (KD11-F and KD11-HA)

6. "M" is illegal and micro-ODT prints ?<CR><LF>@.

7. Rubout is illegal and micro-ODT prints ?<CR><LF>@.

8. "L" is illegal and micro-ODT prints ?<CR><LF>@.

9. Control-shift-S command mode (ASCII 23) accepts 2 bytes forming an 18-bit address with bits <17:16> always zeros and dumps 10 bytes in binary format. The 2 input bytes are not echoed.

10. Up to an 18-bit address and 18-bit data may be entered. Leading zeros are assumed.

11. Incrementing <LF>, the addresses 177776, 377776, 577776 and 777776 result in the addresses 000000, 200000, 400000, and 600000, respectively. That is, 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 "R0" and the contents of R0.

13. The I/O page is in the address group 77XXXX, where address bits <17:12> must be explicit ones.

14. The micro-ODT mode can be entered from the following sources.

    a. A PDP-11 HALT instruction when in kernel mode; the POKL line is low and the HALT jumper option strap is present.

    b. An asserted HALT line.

    c. A power-up option.

LSI-11/23 and LSI-11/23B (KDF11-AA and KDF11-BA)	LSI-11 and LSI-11/2 (KD11-F and KD11-HA)
d. A power-up option.	d. An asserted HALT line caused by a DLV11 framing error.
e. An asserted HALT line caused by a DLV11 framing error.	e. A micro-ODT bus error.
f. A micro-ODT bus error.	(See NOTE*)
g. A memory refresh bus error.	
h. An interrupt vector timeout.	
i. A nonexistent micro PC address.	
15. A carriage return <CR> is echoed and followed by a line feed <LF> only.	15. A carriage return <CR> is echoed and followed by another <CR> and a line feed <LF>.
16. No "H" command.	16. "H" causes the LSI-11/23 to execute microcode routine that, in effect, does nothing. †

*14. The micro-ODT mode can be entered on the LSI-11/23B from the following sources.
1. A PDP-11 HALT instruction when in kernel mode, if the HALT TRAP jumper (J16 to J18) is not installed.
2. An asserted HALT line.
3. Power-up mode option 1 selected.

†16. "H" causes the LSI-11/23B to echo the "H" and print a prompt character rather than a "?", which is the invalid character response. No other operation is performed.

APPENDIX F
FUNCTIONAL DESCRIPTION OF BUS SIGNALS

The following Table F-1 offers a functional description of the extended LSI-11 bus signals.

Table F-1 Extended LSI-11 Bus Signal Functions

Bus Pin	Signal Mnemonic	Signal Function
AA1	BIRQ5 L	Interrupt request priority level 5.
AB1	BIRQ6 L	Interrupt request priority level 6.
AC1	BDAL16 L	Address line 16 during addressing protocol; parity control line during data transfer protocol.
AD1	BDAL17 L	Address line 17 during addressing protocol; parity control line during data transfer protocol.
AE1	SSPARE1 alternate +5B	Special spare – Not assigned or bused in DIGITAL cable or backplane assemblies; available for user connection. This pin may be used optionally for +5 V battery (+5B) backup power to keep critical circuits alive during power failures. A jumper is required on LSI-11 bus options to open (disconnect) the +5B circuit in systems that use this line as SSPARE1.
AF1	SSPARE2 SRUN	Special spare – Not assigned or bused in DIGITAL cable or backplane assemblies; available for user interconnection. In the highest priority device slot, the processor may use this pin for a signal to indicate its RUN state.
AH1	SSPARE3	Special spare – Not assigned nor bused in DIGITAL cable or backplane assemblies; available for user interconnection.
AJ1	GND	Ground – System signal and dc return.
AK1 AL1	MSPAREA MSPAREB	Maintenance spare – Normally connected together on the backplane at each option location (not a slot-to-slot bused connection).
AM1	GND	Ground – System signal and dc return.
AN1	BDMRL	Direct memory access (DMA) request – Device asserts this signal to request bus mastership.

Table F-1 Extended LSI-11 Bus Signal Functions (Cont)

Bus Pin	Signal Mnemonic	Signal Function
AP1	BHALT L	Processor halt – When BHALT L is asserted, the processor responds by going into its halt state (generally console ODT mode.)
AR1	BREF L	Memory refresh – Used during refresh protocol to override memory bank selection decoding and to cause all banks to be selected. Asserted or negated with BRPLY by block mode slave devices to indicate to the bus master whether the slave can accept another block mode DIN or DOUT transfer.
AS1	+5B or +12B battery	+12 or +5 Vdc battery backup power to keep critical circuits alive during power failures. This signal is not bused to BS1 in all DIGITAL backplanes. A jumper is required on all LSI-11 bus options to open (disconnect) the backup circuit from the bus in systems that use this line at the alternate voltage.
AT1	GND	Ground – Systems signal and dc return.
AU1	PSPARE1	Power spare 1 (not assigned a function; not recommended for use) – If a backplane is busing −12 V (on pin BB2) and a module is accidentally inserted upside down in the backplane, −12 Vdc appears on pin AU1. If AU1 is unused on the module, no damage will occur.
AV1	+5B	+5 V battery backup power – For keeping critical circuits alive during power failures.
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.
BB1	BPOK H	AC 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 BD1 BE1 BF1	SSPARE 4 SSPARE 5 SSPARE 6 SSPARE 7	Special spares <7:4> in standard LSI-11 bus systems (16- and 18-address-bit systems). Not assigned or bused in standard LSI-11 bus cable or backplane assemblies. These pins are used to bus address lines <21:18> in extended LSI-11 cable and backplane assemblies. CAUTION **These pins may have been used as test points in some DIGITAL or customer options. These options must be modified or designated incompatible with extended LSI-11 bus backplanes.**
BH1	SSPARE 8	Special spare – Not assigned or bused in DIGITAL cable or backplane assemblies; available for user interconnection.
BJ1	GND	Ground – System signal ground and dc return.

Table F-1 Extended LSI-11 Bus Signal Functions (Cont)

Bus Pin	Signal Mnemonic	Signal Function
BK1 BL1	MSPAREB MSPAREB	Maintenance spares – 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 accepting bus mastership. Device remains bus master until it negates BSACK L.
BP1	BIRQ 7 L	Interrupt request priority level 7.
BR1	BEVNT L	External event interrupt request – The processor latches the leading edge and arbitrates as an interrupt. A typical use of this signal is a line-time clock interrupt.
BS1	+12B	+12 Vdc battery backup power (not bused to AS1 in all DIGITAL backplanes).
BT1	GND	Ground – System signal ground and dc return.
BU1	PSPARE2	Power spare 2 (not assigned a function, not recommended for use) – If a backplane is busing −12 V (on pin AB2) and a module is accidentally inserted upside down in the backplane, −12 Vdc appears on pin BU1. If BU1 is unused on the module, no damage will occur.
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.
AC2	GND	Ground – System signal and dc return.
AD2	+12	+12 V power – Normal +12 Vdc system power.
AE2	BDOUT L	Data output – When asserted, BDOUT implies that valid data is available on BDAL <15:0> 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.
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 will place its data on the BDAL bus or that it will accept data from the bus, according to the appropriate protocol.

Table F-1 Extended LSI-11 Bus Signal Functions (Cont)

Bus Pin	Signal Mnemonic	Signal Function
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) from the addressed slave. 2. The interrupt fielding processor initiates interrupt service by asserting TDIN L followed by TIACK 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. In block mode BSYNC L remains asserted until the last transfer cycle is completed.
AK2	BWTBT L	Write/byte – BWTBT L is used in two ways to control a bus cycle: 1. It is asserted during the address portion of a cycle to indicate that an output cycle is to follow DATO or DATO(B) rather than an input cycle. 2. It is asserted during the data portion of a DATO(B) or DATIO(B) bus cycle, to indicate a byte rather than a word transfer is to take place.
AL2	BIRQ4 L	Interrupt request priority level 4.
AM2 AN2	BIAKI L BIAKO L	Interrupt acknowledge – In accordance with interrupt protocol, the processor asserts BIAKO L to acknowledge an interrupt. The bus transmits this to BIAKI L of the next priority device (electrically closest to the processor). This device accepts the interrupt acknowledge if: 1. The device requested the bus by asserting an interrupt, BIRQX L. 2. The device had the highest priority interrupt request on the bus at the time of the preceding BDIN L assertion. If both these conditions are not met, the device asserts BIAKO L to the next device on the bus. This process continues in a daisy-chain fashion until the device with the highest interrupt priority receives the interrupt acknowledge (IAK) signal and proceeds with interrupt protocol.
AP2	BBS7 L	Bank 7 select – When the bus master asserts TADDR, it asserts BBS7 L to reference the I/O page (including that portion of the I/O page reserved for nonexistent memory). The address on BDAL <12:0> L when BBS7 L is asserted is the address within the I/O page.

Table F-1 Extended LSI-11 Bus Signal Functions (Cont)

Bus Pin	Signal Mnemonic	Signal Function
AR2 AS2	BDMGI L DBMGO L	Direct memory access grant – The bus arbitrator asserts this signal to grant bus mastership to a requesting device, according to bus mastership protocol. The signal is passed in a daisy-chain from the arbitrator (as BDMGO L) through the bus to BDMGI L of the next priority device (electrically closest device on the bus). This device accepts the grant only if it requested to be bus master (by asserting BDMR L). If it did not, the device passes the grant (asserts BDMGO L) to the next device on the bus. This process continues until the requesting device acknowledges the grant by asserting BSACK L after BRPLY L and BSYNC L are both negated.
AT2	BINIT L	Initialize – This signal is used for system reset. All devices on the bus are to return to a known, initial state; that is, registers are reset to zero, all bus drivers are disabled, and logic is reset to state 0, ready to be addressed for operations. Exceptions should be completely documented in programming and engineering specifications for the device.
AU2	BDAL0 L	Data/address line 00 – Specifies high or low byte during address for DATO(B) and DATIO(B) cycles.
AV2	BDAL1 L	Data/address line 01.
BA2	+5	+5 Vdc power.
BB2	−12	−12 Vdc power (optional, not required for DIGITAL LSI-11 or F11 hardware options).
BC2	GND	Power supply return.
BD2	+12	+12 Vdc power.
BE2	BDAL2 L	Data/address line 02.
BF2	BDAL3 L	Data/address line 03.
BH2	BDAL4 L	Data/address line 04.
BJ2	BDAL5 L	Data/address line 05.
BK2	BDAL6 L	Data/address line 06.
BL2	BDAL7 L	Data/address line 07.
BM2	BDAL8 L	Data/address line 08.

Table F-1 Extended LSI-11 Bus Signal Functions (Cont)

Bus Pin	Signal Mnemonic	Signal Function
BN2	BDAL9 L	Data/address line 09.
BP2	BDAL10 L	Data/address line 10.
BR2	BDAL11 L	Data/address line 11.
BS2	BDAL12 L	Data/address line 12.
BT2	BDAL13 L	Data/address line 13.
BU2	BDAL14 L	Data/address line 14.
BV2	BDAL15 L	Data/address line 15.

KDF11-BA CPU Module User's Guide
EK-KDFEB-UG-001

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
Northboro, MA 01532
Attention: Printing and Circulating Service (NR2/M15)
Customer Services Section

Order No. EK-KDFEB-UG

Do Not Tear – Fold Here and Staple

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD, MA.

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Educational Services Development and Publishing
200 Forest Street (MR1-2/T17)
Marlboro, MA 01752