EK-KDJ1A-UG-001

KDJ11-A CPU Module User's Guide

Prepared by Educational Services
of
Digital Equipment Corporation

Preliminary Edition, January 1984
1st Edition, May 1984

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.

The manuscript for this book was created using a DIGITAL Word Processing System and, via a translation program, was automatically typeset on DIGITAL's DECset Integrated Publishing System. Book production was done by Educational Services Development and Publishing in Marlboro and Bedford, MA.

The following are trademarks of Digital Equipment Corporation.

digital	MASSBUS	RSTS
DEC	MicroPower/PASCAL	RSX
DECmate	MINC-11	RT-11
DECnet	OMNIBUS	TOPS-10
DECUS	OS/8	TOPS-20
DECsystem-10	PDP	UNIBUS
DECSYSTEM-20	PDT	VAX
DECwriter	P/OS	VMS
DIBOL	Professional	VT
EduSystem	QBus	Work Processor
IAS	Rainbow	

CONTENTS

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 user's guide is intended to support the users of the KDJ11-A CPU module by providing them with architecture, programming, diagnostic and configuration information. The architecture is described in Chapter 1 and is supported by the functional theory description in Chapter 4. The diagnostics and booting procedures are described in Chapter 9, and Chapter 3 provides the techniques used for on-line debugging (ODT). The configuration requirements for both the module and system applications are described in Chapter 2. Chapter 5 provides the information on the LSI-11 bus used in most system applications.

The KDJ11-A module uses the standard instruction set described in Chapter 6 and the floating-point instruction set described in Chapter 7. Also described in Chapter 6 are the addressing modes which are supported by the programming techniques described in Chapter 8. The detailed timing information is provided in Appendix A and the differences between other LSI-11 and PDP-11 microprocessors are listed in Appendix B.

CHAPTER 1
ARCHITECTURE

1.1 DESCRIPTION

The KDJ11-A is a dual-height processor module for LSI-11 type bus systems. It is designed for use in high-speed, real-time applications and for multiuser, multitasking environments.

The KDJ11-A module executes the complete PDP-11 integer and FP-11 floating-point instruction sets. Full 22-bit memory management is provided for both instruction references and data references in three protection modes – kernel, supervisor, and user. The KDJ11-A module is fully downward compatible with older PDP-11 models which have 18-bit memory management or no memory management.

The three protection modes provide the ability to implement layered software protection. Memory management separately manages each mode, allowing each mode to access different sections of main memory. Furthermore, each section can have different access protection rights. Each mode uses a separate system stack pointer that offers an additional degree of isolation. The protection modes are organized so that a higher protection mode can always enter a lower protection mode, while a lower protection mode can never accidentally enter a higher protection mode. Kernel mode has full privileges and can execute all instructions. Supervisor mode and user mode, the two lower privileged modes, cannot execute certain instructions.

The module interfaces to the extended LSI-11 bus and can address up to 4 megabytes of main memory. Block mode DMA transfers, which are allowed on the extended bus, are supported by the KDJ11-A. The 22-bit extended LSI-11 bus is fully downward compatible with the standard 18-bit LSI-11 bus.

The KDJ11-A module supports console emulation (micro octal debugging tool or ODT). This allows users to interrogate and write main memory and CPU registers as if a console switch panel and display lights were available.

The module contains an 8 Kbyte write-through direct map cache (set size one, block size one). The cache is transparent to all programs and acts as a high-speed buffer between the processor and main memory. The data stored in the cache represents the most active portion of the main memory being used. The processor accesses main memory only when data is not available in the cache.

The user-visible registers are shown in Figure 1-1 and are classified as general purpose, system control, memory system, floating point and memory management registers.

Self-diagnostic LEDs are provided on the KDJ11-A module and indicate the status of the module and system when the module is powered-up. The LEDs aid in troubleshooting module failures.

The KDJ11-A module can run RT-11 V5.1, RSX-11M, RSX-11M PLUS, RSTS/E, UNIX, and micro-power PASCAL operating systems.

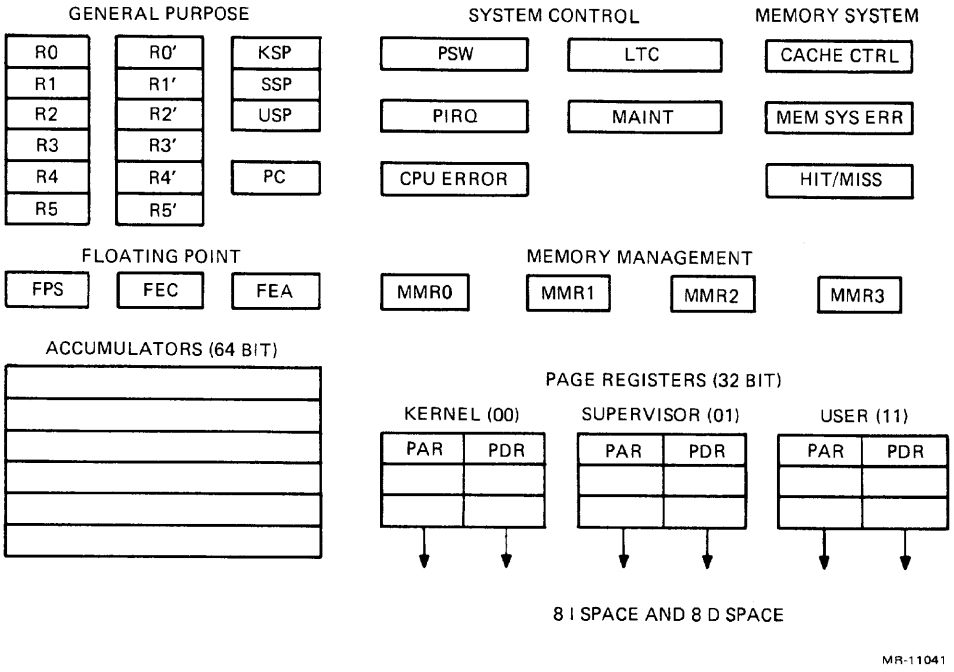


Figure 1-1 Programming Model

1.2 GENERAL PURPOSE REGISTERS

There are 16 general purpose registers (GPR), as listed in Table 1-1, but only 8 are visible to the user at any given time. All these registers can be used as accumulators, deferred addresses, index references, autoincrement, autodecrement, and stack pointers.

1.2.1 Registers

There are two groups of six registers designated R0–R5 and R0'–R5'. The group currently being used is selected by bit 11 in the processor status word (PSW). When bit 11 is set (1), the R0'–R5' group is selected, and when bit 11 is cleared (0), the R0–R5 group is selected.

Table 1-1 General-Purpose Registers

Register Number	Designation	
0	R0	R0'
1	R1	R1'
2	R2	R2'
3	R3	R3'
4	R4	R4'
5	R5	R5'
6	KSP	SSP
7	PC	USP

1.2.2 Stack Pointer

Register six (R6) is designated as the system stack pointer. There are three stack pointers available, one for each corresponding protection mode. However, only one is visible to the user at a given time. The processor status bits 14 and 15 select the active stack pointer used for all instructions except MFPI, MFPD, MTPI, and MTPD. When these instructions select R6 as the destination register, bits 12 and 13 of the processor status word select the active stack pointer. In both cases, the 2-bit selection code is encoded as described in Table 1-2 to select the active register.

Table 1-2 Stack Pointer (PSW 15, 14 or 13, 12)

Code	Selected R6
00	Kernel stack pointer (KSP)
01	Supervisor stack pointer (SSP)
11	User stack pointer (USP)
10	Illegal – User stack pointer selected

1.2.3 Program Counter

The program counter (PC) contains the 16-bit address of the next instruction stream word to be accessed. It is designated as R7 and controls the sequencing of instructions. The PC is directly addressable by single- and double-operand instructions and is a general purpose register, although it is normally not used as an accumulator.

1.3 SYSTEM CONTROL REGISTERS

The processor status word (PSW), program interrupt request (PIRQ), CPU error register, line clock register, and the maintenance register are designated as the system control registers. These registers are used by the module to control system-oriented functions.

1.3.1 Processor Status Word (Address: 17 777 776)

The processor status word (PSW) provides the current and previous operational modes, the general purpose register group being used, the current priority level, the condition code status, and the trace trap bit used for program debugging. The PSW is initialized at power-up and is cleared with a console start. The PSW register is defined in Figure 1-2 and is described in Table 1-3.

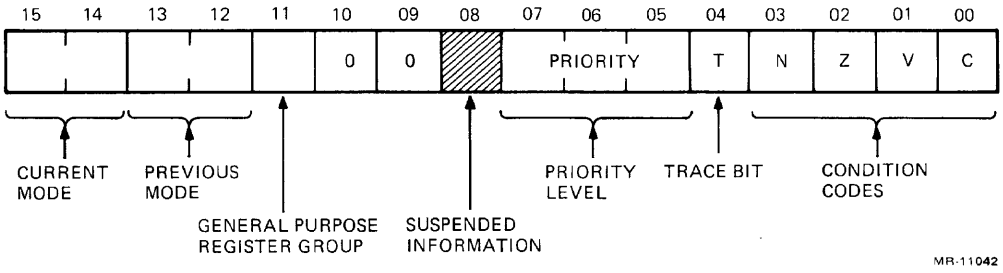


Figure 1-2 Processor Status Register

Table 1-3 Processor Status Bit Description

Bit	Name	Status	Description
15, 14	Current mode	R/W	Indicates the current operating mode and is coded as follows. **Bits 15 14 Mode** 0 0 Kernel 0 1 Supervisor 1 0 Illegal 1 1 User
13, 12	Previous mode	R/W	Indicates the previous operating mode and is coded the same as bits 15, 14.
11	Register set	R/W	Selects the group of general purpose registers being used. When the bit is set, the R0'–R5' group is selected and when cleared, the R0–R5 group is selected.
10, 09	N/A	R	Not used.
08	Suspended information	R/W	Reserved.
07:05	Priority	R/W	Indicates the current priority level of the processor and is coded as follows. **Bits 7 6 5 Priority Level** 1 1 1 7 1 0 0 6 1 0 1 5 1 0 0 4 0 1 1 3 0 1 0 2 0 0 1 1 0 0 0 0
04	Trap*	R/W	The trap bit is inactive when it is cleared. When set, the processor traps to location 14 at the end of the current instruction. It is useful for debugging programs and setting breakpoints.
03	Negative	R/W	Condition code N is set when the previous operation result was negative.
02	Zero	R/W	Condition code Z is set when the previous operation result is zero.
01	Overflow	R/W	Condition code V is set when the previous operation resulted in an arithmetic overflow.
00	Carry	R/W	Condition code C is set when the previous operation caused a carry out.

* The T-bit cannot be set by explicitly writing to the PSW. It can only be changed by the RTI/RTT instructions.

1.3.2 CPU Error Register (Address: 17 777 766)

The CPU error register identifies the source of any trap or abort condition that caused a trap through location 4. Six separate error conditions are identified in Figure 1-3 and are described in Table 1-4. The register is cleared by any write reference, power-up, or by console start. It is not changed by the RESET instruction.

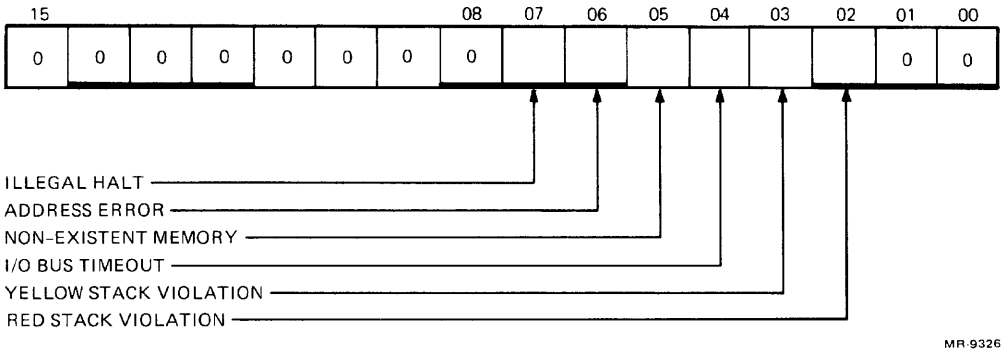


Figure 1-3 CPU Error Register

Table 1-4 CPU Error Register Bit Description

Bit	Name	Status	Function
15:08	Not used	–	–
07	Illegal HALT	Read only	Set when execution of a HALT instruction is attempted in user or supervisor mode.
06	Address error	Read only	Set when word access to an odd byte address or an instruction fetch from an internal register is attempted.
05	Nonexistent memory	Read only	Set when a reference to main memory times out
04	I/O bus timeout	Read only	Set when a reference to the I/O page times out.
03	Yellow stack violation	Read only	Set on a yellow zone stack overflow trap. (Kernel mode stack reference less than 400 octal).
02	Red stack violation	Read only	Set on a red stack trap – a kernel stack push abort during an interrupt, abort, or trap sequence.
01, 00	Not used	–	–

1.3.3 Program Interrupt Request Register (Address: 17 777 772)

The program interrupt request register (PIRQ) implements a software interrupt facility. A request is initiated by setting one of the bits <15:09>, which corresponds to a program interrupt request for priority levels 7–1. Bits <07:05> and <03:01> are set by hardware to the encoded value of the highest pending request set. When the interrupt is acknowledged, the processor vectors to address 240 for a service routine. It is the responsibility of the service routine to clear the interrupt request. The PIRQ register is defined in Figure 1-4 and is described in Table 1-5. The PIRQ register is cleared at power-up, by a console start, or by the RESET instruction.

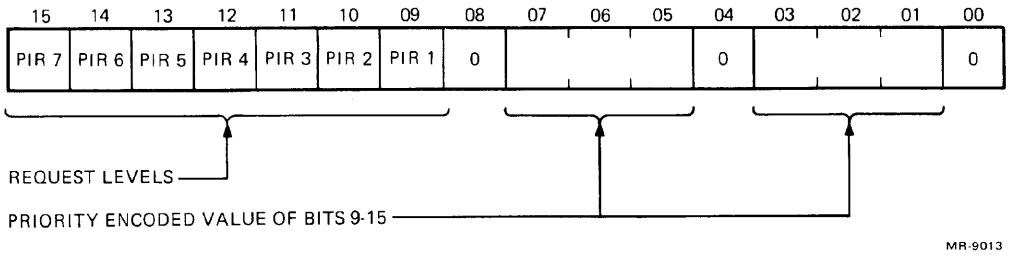


Figure 1-4 Program Interrupt Request Register (PIRQ)

Table 1-5 PIRQ Bit Descriptions

Bit	Name	Status	Function
15	Level 7	Read/write	Requests an interrupt priority of level 7
14	Level 6	Read/write	Requests an interrupt priority of level 6
13	Level 5	Read/write	Requests an interrupt priority of level 5
12	Level 4	Read/write	Requests an interrupt priority of level 4
11	Level 3	Read/write	Requests an interrupt priority of level 3
10	Level 2	Read/write	Requests an interrupt priority of level 2
09	Level 1	Read/write	Requests an interrupt priority of level 1
07:05	Encoded value	Read only	Bits <07:05> represent the encoded value of highest priority level set in bits <15:09>
03:01	Encoded value	Read only	Bits <03:01> represent the encoded value of the highest priority level set in bits <15:09>. Same as bits <07:05>.

1.3.4 Line Time Clock Register (Address: 17 777 546)

The line time clock register (LTC) controls the recognition of the LSI-11 bus BEVNTL signal. When bit 06 of the register is set (1), the BEVNTL signal can be recognized and will generate the highest possible level 6 interrupt request through address location 100. The BEVNTL input is disabled when bit 06 of the register is cleared (0). The BEVNTL input can be permanently disabled by installing the W9 jumper. The register is defined in Figure 1-5 and is described by Table 1-6. The register is cleared at power-up, by a console start, or by the RESET instruction.

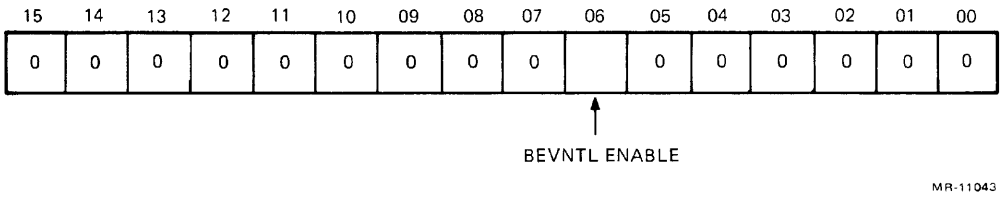


Figure 1-5 Line Time Clock Register (BEVNT)

Table 1-6 Line Time Clock (LTC) Register Bit Descriptions

Bit	Name	Status	Function
15:07	Not used	-	-
06	BEVNT ENABLE	Read/write	When this bit is set (1), the LSI-11 BEVNT L signal can be recognized (unless W9 is installed).
05:00	Not used	-	-

1.3.5 Maintenance Register (Address: 17 777 750)

The maintenance register provides a way for software to determine the power-up options selected by the user. It also indicates if a floating-point accelerator (FPA) is available. The register is defined in Figure 1-6 and is described by Table 1-7.

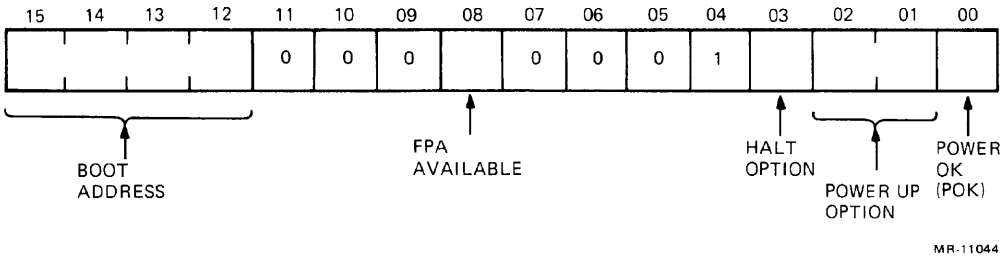


Figure 1-6 Maintenance Register

Table 1-7 Maintenance Register Bit Description

Bit	Name	Status	Function
15:12	Boot address	Read only	These bits read the user's selected boot address. The address is selected by jumpers, W1 (bit 15), W2 (bit 14), W4 (bit 13) and W6 (bit 12). A "1" indicates the jumper is inserted and a "0" indicates the jumper is removed.
11:09	Not used	–	–
08	FPA available	Read only	The bit is set (1) if a floating-point accelerator (FPA) is installed on the module.
07:04	Module ID	–	The "0001" code identifies this module as a KDJ11-A microprocessor.
03	HALT option	Read only	The option determines how the HALT instruction is used in the kernel mode. If W5 is removed, the bit is set (1) and the processor will set up an emergency stack at location 4 and then trap through vector address 4. If W5 is installed, the bit is cleared (0) and the processor will enter console ODT mode.
02, 01	Power-up	Read only	These bits read the power-up mode for the processor. Bit 2 is set (1) by removing jumper W3 and bit 01 is set (1) by removing jumper W7. The following power-up options are available.

Bit 02	Bit 01	Option
0	0	PC at 24, PS at 26
0	1	Micro-ODT, PS = 0
1	0	PC = 173000, PS = 340
1	1	User Bootstrap, PS = 340

00	BPOK H	Read only	The bit is set (1) when the LSI-11 bus signal BPOK H is asserted, indicating that the ac power is okay.

1.4 INTERRUPTS

The KDJ11-A module uses a variety of trap, hardware, and software interrupts, described in Tables 1-8 and 1-9. Four interrupt request lines allow external hardware to interrupt the processor on four interrupt levels using an externally supplied vector. Seven levels of software interrupt requests are supported through use of the PIRQ register. Finally, a variety of internally vectored traps are provided to flag error conditions.

Table 1-8 Asynchronous Interrupts

Interrupt	Internal or External	Vector Address	Priority Level*
Red stack trap (CPU error register, bit 02)	Internal	4	NM
Address error (CPU error register, bit 06)	Internal	4	NM
Memory management violation (MMR0, bits <13:15>)	Internal	250	NM
Timeout/nonexistent memory (CPU error register, bits <04:05>)	Internal	4	NM
Parity error (PARITY, ABORT)	External	114	NM
Trace (T-bit) Trap (PSW, bit 04)	Internal	14	NM
Yellow stack trap (CPU error register, bit 03)	Internal	4	NM
Power fail (PWRF)	External	24	NM
FP exception (FPE)	External	244	NM
PIR 7 (PIRQ, bit 15)	Internal	240	7
IRQ 7	External	User-defined	7
PIR 6 (PIRQ, bit 14)	Internal	240	7
BEVNT	External	100	6
IRQ 6	External	User-defined	6
PIR 5 (PIRQ, bit 13)	Internal	240	5
IRQ 5	External	User-defined	5
PIR 4 (PIRQ, bit 12)	Internal	240	4
IRQ 4	External	User-defined	4
PIR 3 (PIRQ, bit 11)	Internal	240	3
PIR 2 (PIRQ, bit 10)	Internal	240	2
PIR 1 (PIRQ, bit 09)	Internal	240	1
Halt line (HALT)†	External	None – places system in console mode.	

* NM = Non-maskable

† The halt line usually has the lowest priority, however, it has highest priority during vector reads. This allows the user to break out of potential infinite loops. An infinite loop could occur if a vector has not been properly mapped during memory management operations.

Table 1-9 Synchronous Interrupts

Interrupt	Vector Address
FP instruction exception	244
TRAP (trap instruction)	34
EMT (emulator trap instruction)	30
IOT (I/O trap instruction)	20
BPT (breakpoint trap instruction)	14
CSM (call to supervisor mode instruction)	10
HALT instruction*	4
WAIT (wait-for-interrupt instruction)	

* Execution of the HALT instruction performs different operations, depending on jumper W5 and the protection mode. Jumper W5 determines the operation of a HALT instruction in the kernel mode. If it is installed, the processor enters the ODT mode, and, if it is removed, the processor sets up an emergency stack at location 4 and traps to location 4. The HALT instruction in the supervisor or user mode is an illegal instruction and the processor traps to location 4. This condition also sets bit 07 of the CPU error register.

1.5 MEMORY MANAGEMENT

KDJ11-A memory management provides the hardware for complete memory management and protection. It is designed to be a memory management facility for accessing all of physical memory and for multiuser, multiprogramming systems where memory protection and relocation facilities are necessary.

In multiprogramming environments, several user programs are resident in memory at any given time. The tasks of the supervisory program include the following.

1. Control the execution of the various user programs
2. Manage the allocation of memory and peripheral device resources
3. Safeguard the integrity of the system as a whole by control of each user program

In a multiprogramming system, memory management provides the means for assigning memory pages to a user program and preventing that user from making any unauthorized access to pages outside his assigned area. Thus, a user can effectively be prevented from accidental or willful destruction of any other user program or the system executive program.

The following are the basic characteristics of KDJ11-A memory management.

- 16 user mode memory pages
- 16 supervisor mode memory pages
- 16 kernel mode memory pages
- 8 pages in each mode for instructions
- 8 pages in each mode for data
- Page lengths from 64 to 8192 bytes
- Each page provided with full protection and relocation
- Transparent operation
- 3 modes of memory access control
- Memory access to 4 megabytes.

1.5.1 Memory Mapping

The processor can perform 16-bit, 18-bit or 22-bit address mapping. The I/O page, which is the uppermost 4 K words of memory, always uses the physical addresss locations 17 760 000 to 17 777 777.

1.5.1.1 16-Bit Mapping – There is a direct mapping relocation from virtual to physical addresses. The lowest 28 K virtual addresses are the same corresponding physical addresses. The I/O page physical addresses are located in the upper 4 K block as shown in Figure 1-7.

1.5.1.2 18-Bit Mapping – Each of the three modes; kernel, supervisor, and user, are allocated 32 K words that are mapped into 128 K words of physical address space. The lowest 124 K words of physical memory or the I/O page can be referenced as shown in Figure 1-8.

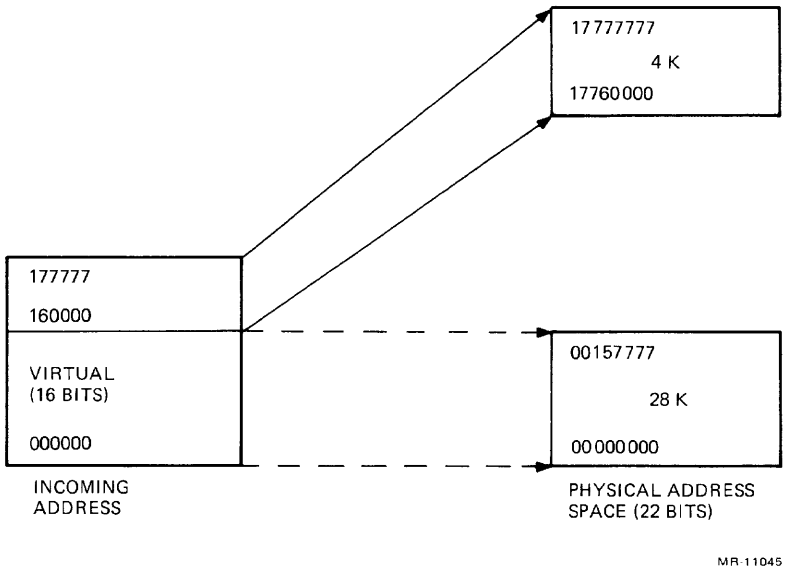


Figure 1-7 16-Bit Mapping

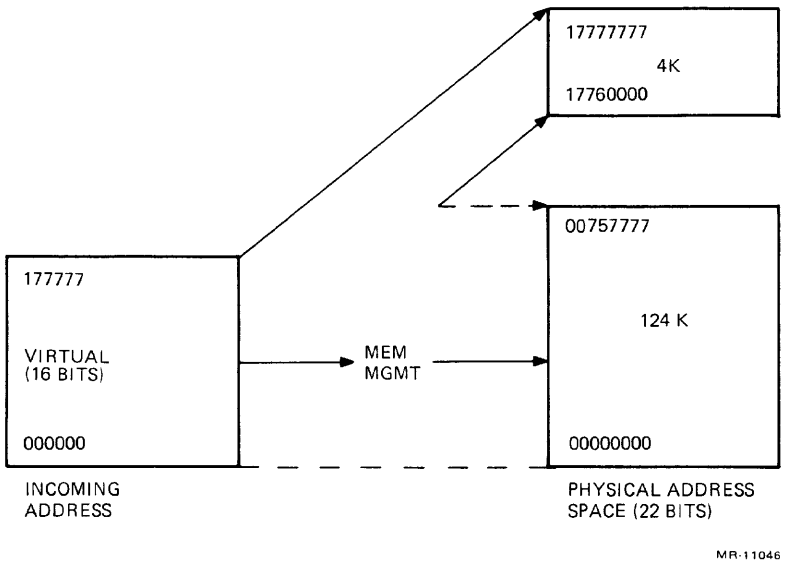


Figure 1-8 18-Bit Mapping

1.5.1.3 22-Bit Mapping – This mode uses the full 22-bit addresses to access all of the physical memory. The upper 4 K block is still the I/O page as shown in Figure 1-9.

1.5.2 Compatibility

The operation of 16-, 18-, and 22-bit mapping can be used to provide compatibility among other PDP-11 computers. This means that software written and developed for any PDP-11 computer can be run on the KDJ11-A without modification. Refer to Table 1-10.

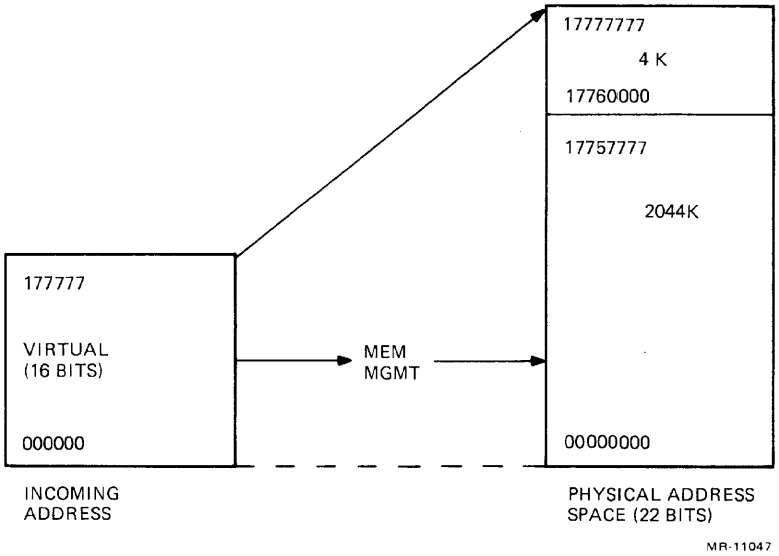


Figure 1-9 22-Bit Mapping

Table 1-10 KDJ11-A Compatibility

Mapping	Memory Management	System
16-bit	Off	PDP-11/05, 11/10, 11/15, 11/20, 11/03
18-bit	On	PDP-11/35, 11/40, 11/45, 11/50, 11/23
22-bit	On	PDP-11/70, 11/44, 11/24, 11/23 plus

1.5.3 Virtual Addressing

When memory management is operating, the normal 16-bit address is no longer interpreted as a direct physical address but as a virtual address containing information to be used in constructing a new 22-bit physical address. The information contained in the virtual address is combined with relocation information contained in the page address register to yield a 22-bit physical address as shown in Figure 1-10. Using memory management, memory can be dynamically allocated in pages, each composed of from 1 to 128 integral blocks of 64 bytes.

The starting physical address for each page is an integral multiple of 64 bytes, and each page has a maximum size of 8192 bytes. Pages may be located anywhere within the physical address space. The determination of which set of 16 pages registers is used to form a physical address is made by the current mode of operation (i.e., kernel, supervisor, or user mode), and if the reference is for instructions or data.

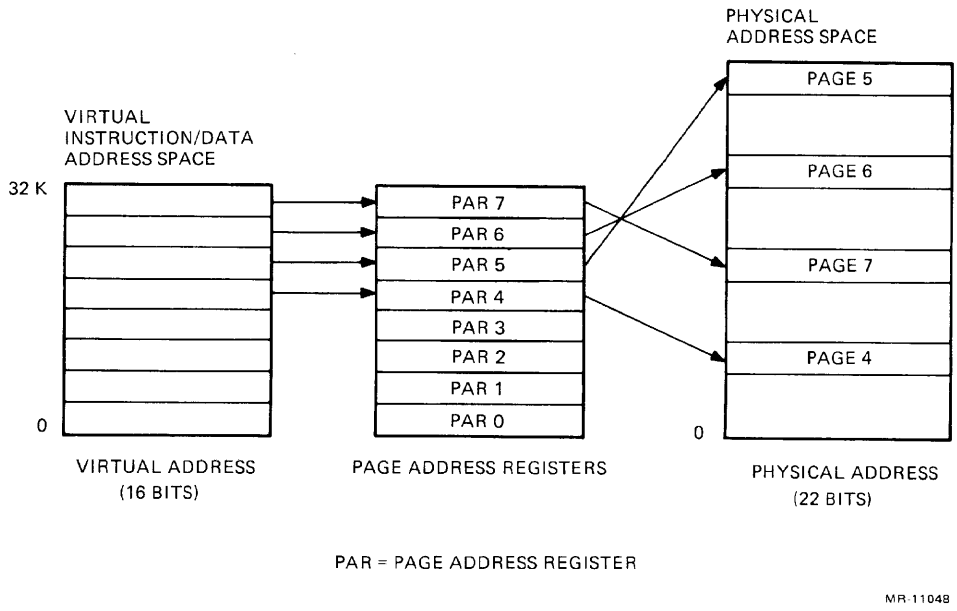


Figure 1-10 Virtual Address Mapping into Physical Address

1.5.4 Interrupt Conditions Under Memory Management Control

Memory management relocates all addresses. When it is enabled, all traps, aborts, and interrupt vectors are mapped using the kernel mode data space mapping registers. Therefore, when a vectored transfer occurs, the new program counter (PC) and processor status word (PS) are obtained from two consecutive words physically located at the trap vector and are mapped using kernel mode data space registers.

The stack used for the "push" of the current PC and PSW is specified by bits 14 and 15 of the new PSW. The PSW mode bits also determine the new mapping register set. This allows the kernel mode program to have complete control over servicing all traps, aborts or interrupts. The kernel program may assign the service of some of these conditions to a supervisor or user mode program by simply setting the mode bits of the new PSW in the vector to return control to the appropriate mode.

1.5.5 Construction of a Physical Address

All addresses with memory relocation enabled either reference information in instruction (I) space or data (D) space. I space is used for all instruction fetches, index words, absolute addresses, and immediate operands; D space is used for all other references. I space and D space each have eight page address registers (PARs) in each mode of CPU operation (kernel, supervisor, and user). Memory management register 3, can disable D space and map all references (instructions and data) through I space, or can enable D space and map all references through both I and D space.

The basic information needed for the construction of a physical address comes from the virtual address, which is illustrated in Figure 1-11, and the appropriate PAR set.

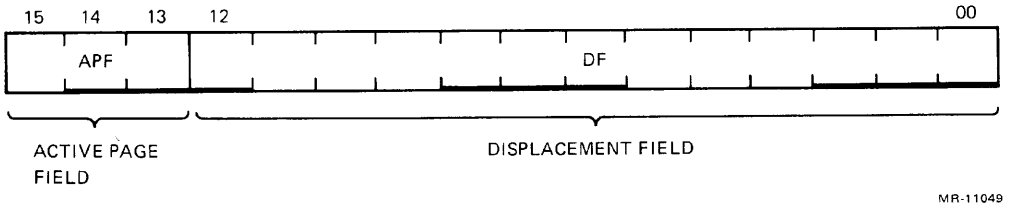


Figure 1-11 Interpretation of a Virtual Address

The virtual address consists of:

1. The active page field. This 3-bit field determines which of 8 page address registers from the PAR set (PAR0–PAR7) will be used to form the physical address.

2. The displacement field. This 13-bit field contains an address relative to the beginning of a page. The longest page length is 8 Kbytes (2^{13} = 8 Kbytes). The DF is further subdivided into two fields as shown in Figure 1-12.

The displacement field consists of:

1. The block number. This 7-bit field is interpreted as the block number within the current page.

2. The displacement in block. This 6-bit field contains the displacement within the block referred to by the block number.

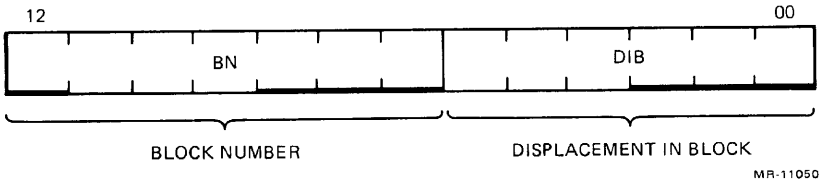


Figure 1-12 Displacement Field of a Virtual Address

The remainder of the information needed to construct the physical address comes from the contents of the PAR referenced by the page address field. This 16-bit register specifies the starting address of the memory page. The PAF is actually a block number in the physical memory. For instance, PAF = 3 indicates a starting address of 96 (3 × 32) words in physical memory.

The construction of the physical address is illustrated in Figure 1-13.

The logical sequence involved in constructing a physical address (PA) is as follows.

1. Select a set of page address registers. This depends on the space being referenced and the protection mode being used.

2. The active page field of the virtual address selects one of eight page address registers (PAR0–PAR7) from the appropriate set.

3. The page address field of the selected page address register contains the starting address of the currently active page as a block number in physical memory.

4. The block number from the virtual address is added to the page address field to yield the number of the block in physical memory. This is bits <21:06> of the physical address being constructed.

5. The displacement in block from the displacement field of the virtual address is joined to the physical block number to yield a true 22-bit physical address.

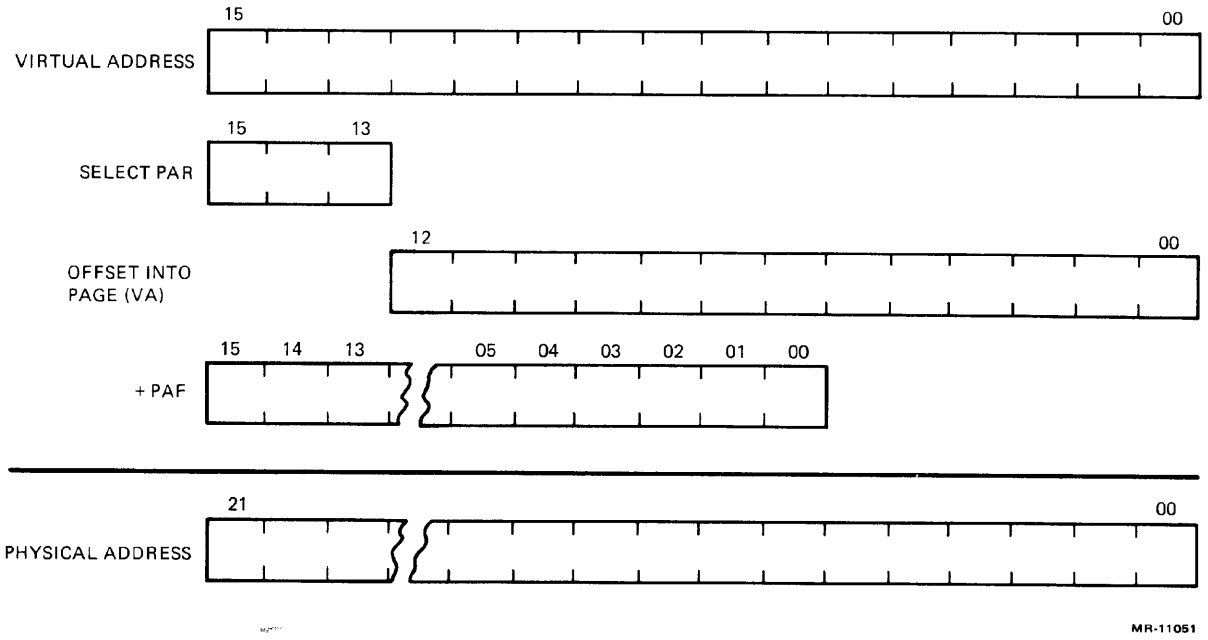


Figure 1-13 Construction of a Physical Address

1.5.6 Memory Management Registers

Memory management implements 3 sets of 32 16-bit registers as shown in Figure 1-14. One set of registers is used in kernel mode, another in supervisor mode, and the other in user mode. The protection mode in use determines which set is to be used. Each set is subdivided into two groups of 16 registers. One group is used for references to instruction (I) space, and one to data (D) space. The I space group is used for all instruction fetches, index words, absolute addresses, and immediate operands. The D space group is used for all other references, providing it has not been disabled by memory management register 3. Each group is further subdivided into two parts of eight registers. One part is the page address register (PAR) whose function was described previously. The other part is the page descriptor register (PDR). PARs and PDRs are always selected in pairs by the top three bits of the virtual address. A PAR/PDR pair contains all the information needed to describe and locate a currently active memory page.

The memory management registers are located in the uppermost 8 Kbytes of physical address space, which is designated as the I/O page. The addresses allocated to the memory management registers are listed in Table 1-11.

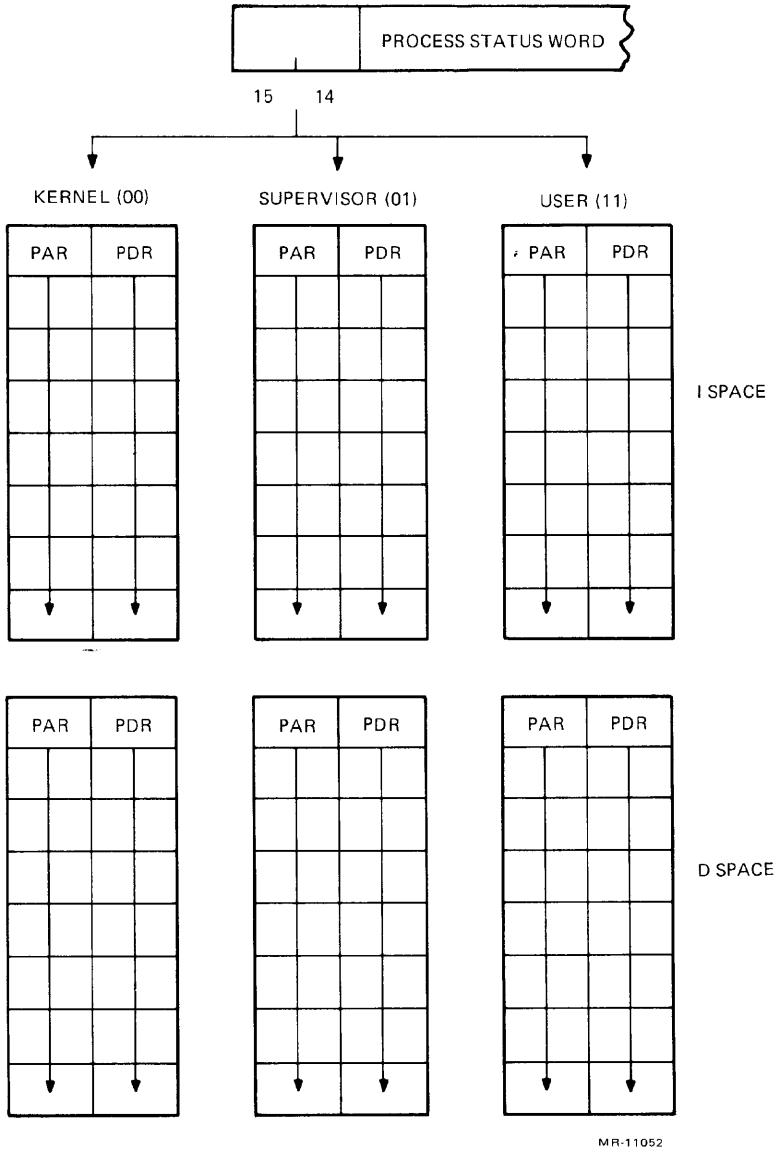


Figure 1-14 Active Page Registers

Table 1-11 Memory Management Register Addresses

Register	Address
Memory management register 0(MMR0)	17 777 572
Memory management register 1(MMR1)	17 777 574
Memory management register 2(MMR2)	17 777 576
Memory management register 3(MMR3)	17 772 516
User I space descriptor register (UISDR0)	17 777 600
.	.
.	.
.	.
User I space descriptor register (UISDR7)	17 777 616
User D space descriptor register (UDSDR0)	17 777 620
.	.
.	.
.	.
User D space descriptor register (UDSDR7)	17 777 636
User I space address register (UISAR0)	17 777 640
.	.
.	.
.	.
User I space address register (UISAR7)	17 777 656
User D space address register (UDSAR0)	17 777 660
.	.
.	.
.	.
User D space address register (UDSAR7)	17 777 676
Supervisor I space descriptor register (SISDR0)	17 772 200
.	.
.	.
.	.
Supervisor I space descriptor register (SISDR7)	17 772 216
Supervisor D space descriptor register (SDSDR0)	17 772 220
.	.
.	.
.	.
Supervisor D space descriptor register (SDSDR7)	17 772 236

Register	Address
Supervisor I space address register (SISAR0)	17 772 240
.	.
.	.
.	.
Supervisor I space address register (SISAR7)	17 772 256
Supervisor D space address register (SDSAR0)	17 772 260
.	.
.	.
.	.
Supervisor D space address register (SDSDR7)	17 772 276
Kernel I space descriptor register (KISDR0)	17 772 300
.	.
.	.
.	.
Kernel I space descriptor register (KIDSR7)	17 772 316
Kernel D space descriptor register (KDSDR0)	17 772 320
.	.
.	.
.	.
Kernel D space descriptor register (KDSDR7)	17 772 336
Kernel I space address register (KISAR0)	17 772 340
.	.
.	.
.	.
Kernel I space address register (KISAR7)	17 772 356
Kernel D space address register (KDSAR0)	17 772 360
.	.
.	.
.	.
Kernel D space address register (KDSAR7)	17 772 376

1.5.6.1 Page Address Registers – The page address register (PAR) contains the page address field (PAF), a 16-bit field that specifies the starting address of the page as a block number in physical memory.

The page address register (see Figure 1-15) contains the page address field that may be alternatively thought of as a relocation register containing a relocation constant, or as a base register containing a base address. These registers are not changed by either console starts or the reset instruction. They are undefined at power-up.

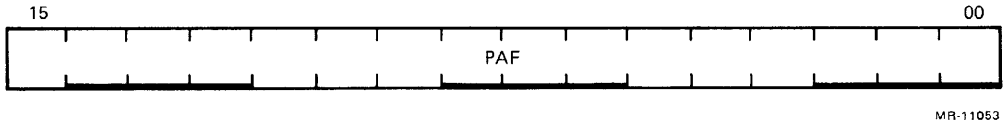


Figure 1-15 Page Address Register (PAR)

1.5.6.2 Page Descriptor Register – The page descriptor register contains information relative to page expansion, page length, and access control. The register is shown in Figure 1-16 and is described in Table 1-12.

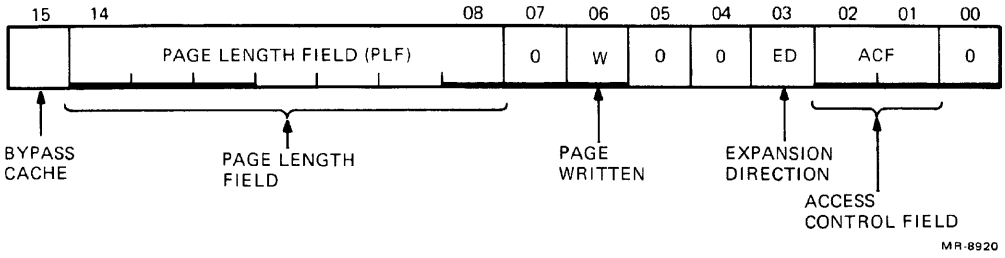


Figure 1-16 Page Descriptor Register (PDR)

1.5.7 Fault Recovery Registers

Aborts generated by the memory management hardware are vectored through kernel virtual location 250. Memory management registers 0, 1, 2, and 3 are used to determine why the abort occurred and to allow for program restarting.

NOTE

An abort to a location which is itself an invalid address will cause another abort. Thus, the kernel program must ensure that kernel virtual address 250 is mapped into a valid address; otherwise, a loop will occur that will require console intervention.

Table 1-12 Page Descriptor Bit Description

Bit	Name	Status	Function
15	Bypass cache	Read/write	This bit implements a conditional cache bypass mechanism. If the PDR accessed during a relocation operation has this bit set, the reference will go directly to main memory. Read or write hits will result in invalidation of the accessed cache location.
14:08	Page length field	Read/write	This field specifies the block number which defines the page boundary. The block number of the virtual address is compared against the page length field to detect length errors. An error occurs when expanding upwards if the block number is greater than the page length field, and when expanding downwards if the block number is less than the page length field.
07	Not used	-	-
06	Page written	Read only	The written into (W) bit indicates whether the page has been written into since it was loaded in memory. When this bit is set, it indicates a modified page. The W-bit is automatically cleared when the PAR or PDR of that page is written.
05, 04	Not used	-	-
03	Expansion direction	Read/write	This bit 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.
02, 01	Access control field	Read/write	This field contains the access code for this particular page. The access code specifies 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. Implemented codes are: 00 Nonresident - abort all accesses 01 Read only - abort on write 10 Not used - abort all accesses 11 Read/write access
00	Not used	-	-

1.5.7.1 Memory Management Register 0 (Address: 17 777 572) – Memory management register 0 (MMR0) provides MMU control and records MMU status. The register contains abort and status flags as shown in Figure 1-17 and described in Table 1-13.

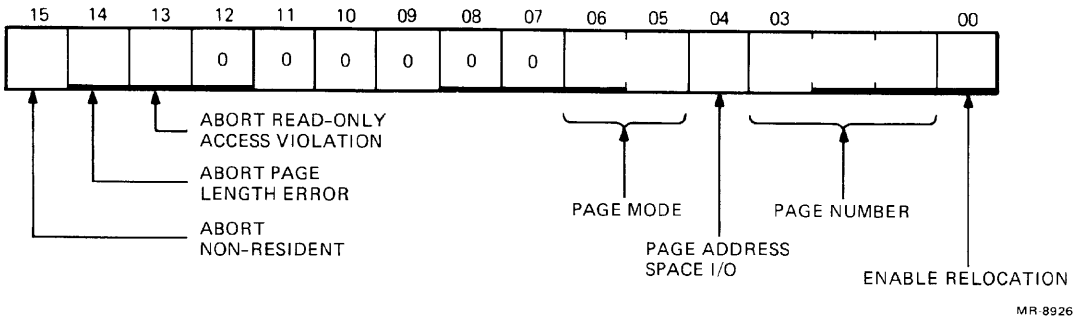


Figure 1-17 Memory Management Register 0 (MMR0)

Table 1-13 MMR0 Bit Descriptions

Bit	Name	Status	Function
15*	Nonresident abort	Read/write	Bit 15 is set by attempting to access a page with an access control field key equal to 0 or 2. It is also set by attempting to use memory relocation with a processor mode (PS<15:14>) of 2.
14*	Page length abort	Read/write	Bit 14 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 of the page descriptor register for that page.
13*	Read only abort	Read/write	Bit 13 is set by attempting to write in a read-only page. Read-only pages have access keys of 1.
12:07	Not used	–	–
06, 05	Processor mode	Read only	Bits <06:05> indicate the processor mode (kernel, supervisor, user, illegal) associated with the page causing the abort (kernel = 00, supervisor = 01, user = 11, illegal = 10). If the illegal mode is specified, an abort is generated and bit 15 is set.
04	Page space	Read only	Bit 04 indicates the address space (I or D) associated with the page causing the abort (0 = I space, 1 = D space).
03:01	Page number	Read only	Bits <03:01> contain the page number of the page causing the abort.
00	Enable relocation	Read/write	Bit 00 enables relocation. When it is set to 1, all addresses are relocated. When bit 00 is set to 0, memory management is inoperative and addresses are not relocated.

* Bits <15:13> can be set by an explicit write; however such an action does not cause an abort. Whether set explicitly or by an abort, setting any bit in bits <15:13> causes memory management to freeze the contents of MMR0 <06:01>, MMR1, and MMR2. The status registers remain frozen until MMR0 <15:13> is cleared by an explicit write.

1.5.7.2 Memory Management Register 1 (Address: 17 777 574) – Memory management register 1 (MMR1) records any autoincrement or autodecrement of a general purpose register, including explicit references through the PC. The increment or decrement amount by which the register was modified is stored in 2's complement notation. The lower byte is used for all source operand instructions and the destination operand may be stored in either byte, depending on the mode and instruction type. The register is cleared at the beginning of each instruction fetch. The register is defined in Figure 1-18.

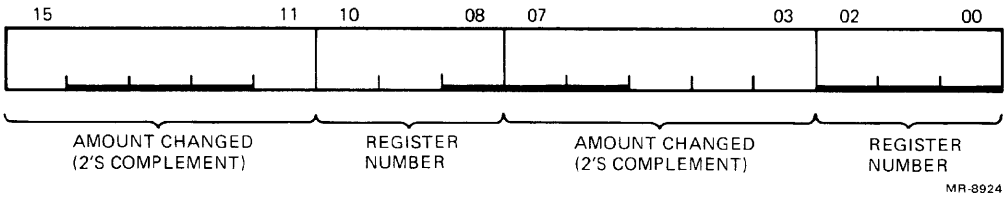


Figure 1-18 Memory Management Register 1 (MMR1)

1.5.7.3 Memory Management Register 2 (Address: 17 777 576) – Memory management register 2 (MMR2) is loaded with the program counter of the current instruction and is frozen when any abort condition is posted in MMR0.

1.5.7.4 Memory Management Register 3 (Address: 17 772 516) – Memory management register 3 (MMR3) enables the data space for the kernel, supervisor, and user operating modes. It also selects either 18-bit or 22-bit mapping and enables the request for the supervisor macroinstruction (CSM). The register is shown in Figure 1-19 and is defined in Table 1-14. MMR3 is cleared during power-up, by a console start, or by a RESET instruction.

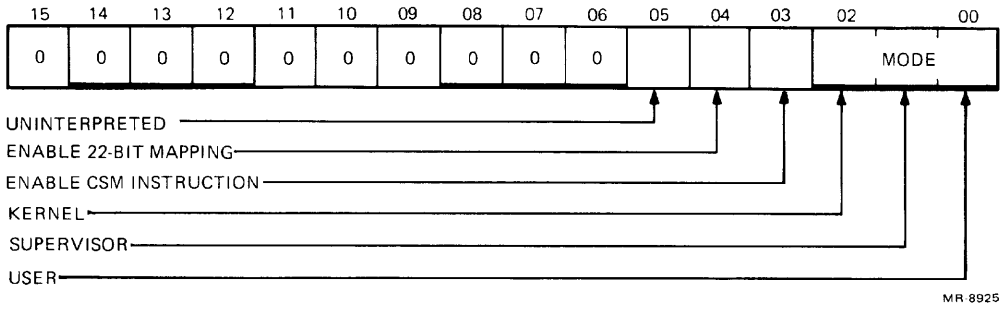


Figure 1-19 Memory Management Register 3 (MMR3)

Table 1-14 MMR3 Bit Description

Bit	Name	Status	Function
15:06	Not used	–	–
05	Uninterpreted	Read/write	This bit can be set or cleared under program control, but it is not interpreted by the KDJ11-A.
04	Enable 22-bit mapping	Read/write	This bit enables 22-bit memory addressing (the default is 18-bit addressing).
03	Enable CSM instruction	Read/write	This bit enables recognition of the call supervisor mode instruction.
02	Kernel data space	Read/write	This bit enables the data space mapping for the kernel operating mode.
01	Supervisor data space	Read/write	This bit enables the data space mapping for the supervisor operating mode.
00	User data space	Read/write	This bit enables the data space mapping for the user operating mode.

1.5.7.5 Instruction Back-Up/Restart Recovery – The process of "backing up" and restarting a partially completed instruction involves the following.

1. Performing the appropriate memory management tasks to alleviate the cause of the abort (e.g., loading a missing page).

2. Restoring the general purpose registers indicated in MMR1 to their original contents at the start of the instruction by subtracting the "modify value" specified in MMR1.

3. Restoring the PC to the "abort-time" PC by loading R7 with the contents of MMR2, which contains the value of the virtual PC at the time the "abort-generating" instruction was fetched.

Note that this back-up/restart procedure assumes that the general purpose register used in the program segment will not be used by the abort recovery routine. This is automatically the case if the recovery program uses a different general purpose register set.

1.5.7.6 Clearing Status Registers Following Abort – At the end of a fault service routine, bits <15:13> of MMR0 must be cleared (set to 0) to resume error checking. On the next memory reference following the clearing of these bits, the various registers will resume monitoring the status of the addressing operations. MMR2 will be loaded with the next instruction address, MMR1 will store register change information, and MMR0 will log memory management status information.

1.5.7.7 Multiple Faults – Once an abort has occurred, any subsequent errors that occur while the memory management registers are frozen will not change MMR0, MMR1 or MMR2. The information saved in MMR0 through MMR2 will always refer to the first abort that it detected.

1.5.8 Typical Usage Examples

The memory management unit provides a general purpose memory management tool. It can be used in a manner as simple or complex as desired. It can be anything from a simple memory expansion device to a complete memory management facility.

The variety of possible and meaningful ways to use the facilities offered by the memory management unit means that both single-user and multiprogramming systems have complete freedom to make whatever memory management decisions best suit their individual needs. Although a knowledge of what most types of computer systems seek to achieve may indicate that certain methods of using the memory management unit will be more common than others, there is no limit to the ways to use these facilities.

In most typical applications, the control over the actual memory page assignments and their protection resides in a supervisory type program which operates in kernel mode. This program sets access keys in such a way as to protect itself from willful or accidental destruction by other supervisor or user mode programs. The facilities are also provided such that the kernel mode program can dynamically assign memory pages of varying sizes in response to system needs.

1.5.8.1 Typical Memory Page – When the memory management unit is enabled, the kernel mode program, a supervisor mode program, and a user mode program each have eight active pages described by the appropriate page address registers and page descriptor registers for data and eight pages for instructions. Each segment is made up of from 1 to 128 blocks and is pointed to by the page address field of the corresponding page address register as illustrated in Figure 1-20.

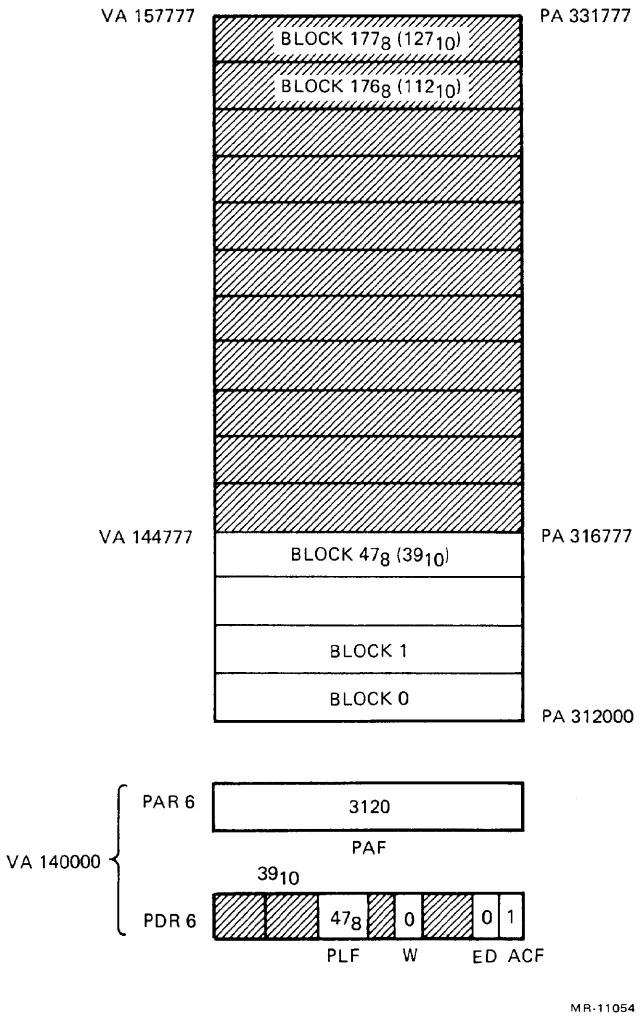


Figure 1-20 Typical Memory Page

The memory segment illustrated in Figure 1-20 has the following attributes.

1. Page length: 40 blocks
2. Virtual address range: 140000–144777
3. Physical address range: 312000–316777
4. Nothing has been modified (i.e., written) in this page
5. Read-only protection
6. Upward expansion

These attributes were determined according to the following scheme.

1. Page address register (PAR6) and page descriptor register (PDR6) were selected by the active page field (APF) of the virtual address. (Bits <15:13> of the VA = 6_8.)

2. The initial address of the page was determined from the page address field of PAR6 (312000 = 3120_8 blocks $\times 40_8$ (32_{10}) words per block $\times$ 2 bytes per word).

   NOTE

   The PAR that contains the PAF constitutes what is often referred to as a base register containing a base address or a relocation register containing a relocation constant.

3. The page length ($47_8 + 1 = 40_{10}$ blocks) was determined from the page length field (PLF) contained in page descriptor register PDR6. Any attempts to reference beyond these 40_{10} blocks in this page will cause a "page length error," which will result in an abort, vectored through kernel virtual address 250.

4. The physical addresses were constructed according to the scheme illustrated in Figure 1-13.

5. The written (W) bit indicates that no locations in this page have been modified (i.e., written). If an attempt is made to modify any location in this particular page, an access control violation abort will occur. If this page were involved in a disk swapping or memory overlay scheme, the W-bit would be used to determine whether it had been modified and, thus, required saving before overlay.

6. This page is read-only protected; i.e., no locations in this page may be modified. The mode of protection was specified by the access control field of PDR6.

7. The direction of expansion is upward (ED = 0). If more blocks are required in this segment, they will be added by assigning blocks with higher relative addresses.

The attributes which describe this page can be determined under software control. The parameters describing the page are loaded into the appropriate page address register (PAR) and page descriptor register under program control. In a normal application, the particular page, which itself contains these registers, would be assigned to the control of a kernel mode program.

1.5.8.2 Nonconsecutive Memory Pages – Higher virtual addresses do not necessarily map to higher physical addresses. It is possible to set up the page address fields of the PARs so that higher virtual address blocks may be located in lower physical address blocks as illustrated in Figure 1-21.

Although a single memory page must consist of a block of contiguous locations, consecutive virtual memory pages do not have to be located in consecutive physical address locations. The assignment of memory pages is not limited to consecutive nonoverlapping physical address locations.

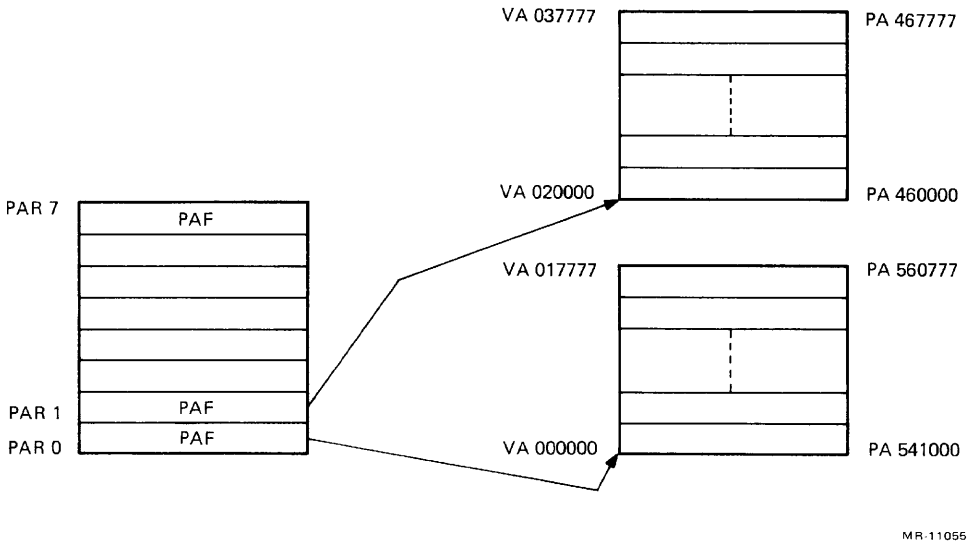


Figure 1-21 Nonconsecutive Memory Pages

1.5.8.3 Stack Memory Pages – When constructing programs, it is often desirable to isolate all program variables from *pure code* (i.e., program instructions) by placing them on a register indexed stack. These variables can then be "pushed" or "popped" from the stack area as needed. (See Chapter 6.) Since stacks expand by adding locations with lower addresses, when a memory page which contains "stacked" variables needs more room, it must "expand down," i.e., add blocks with lower relative addresses to the current page. This mode of expansion is specified by setting the expansion direction bit of the appropriate page descriptor register to a 1. Figure 1-22 illustrates a typical stack memory page. This page will have the following parameters.

PAR6: PAF = 3120
PDR6: PLF = 175_8 or $125_{10}(128_{10}-3)$
ED = 1
W = 0 or 1
ACF = nnn (to be determined by programmer as necessary)

NOTE
The W-bit will be set by hardware.

In this case the stack begins 128 blocks above the relative origin of this memory page and extends downward for a length of three blocks. A page length error abort will be generated by the hardware when an attempt is made to reference any location below the assigned area, i.e., when the block number from the virtual address is less than the page length field of the appropriate page descriptor register.

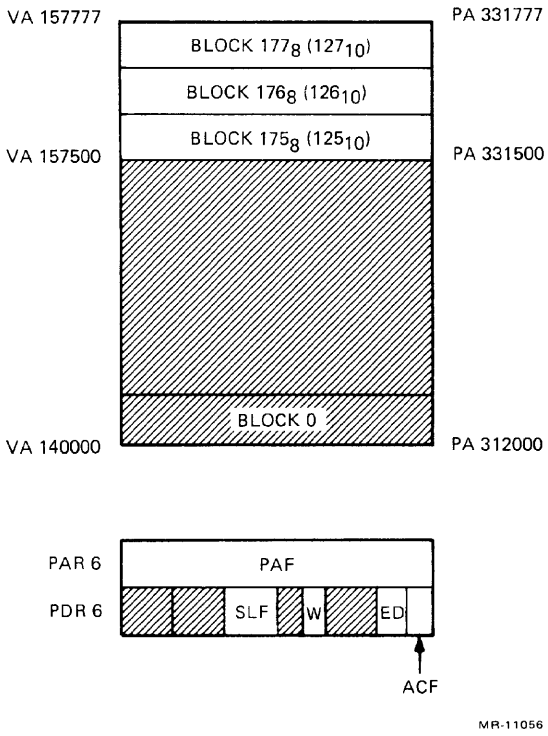


Figure 1-22 Typical Stack Memory Page

1.5.9 Transparency

In a multiprogramming application, it is possible for memory pages to be allocated such that a program appears to have a complete 64 Kbyte memory configuration. Using relocation, a kernel mode supervisory-type program can perform all memory management tasks entirely transparent to a supervisor or user mode program. In effect, a system can use its resources to provide maximum throughput and response to a number of users, each of whom seems to have a powerful system "all to himself."

1.6 CACHE MEMORY

The statistics from executing programs clearly indicate that at any given moment, a program spends most of its time within a relatively small section of code. The KDJ11-A cache memory exploits this phenomenon by using a small amount of high-speed memory to store the most recently accessed memory locations. Cached code will execute much faster than noncached code because of the large difference between the access times of the cache memory and the LSI-11 bus main memory.

The following illustrates how the KDJ11-A cache is constructed. It is a direct map (set size one; block size one), 8 Kbyte cache. Each physical address is logically subdivided into a 9-bit label, 12-bit index, and 1-bit byte select field as shown in Figure 1-23.

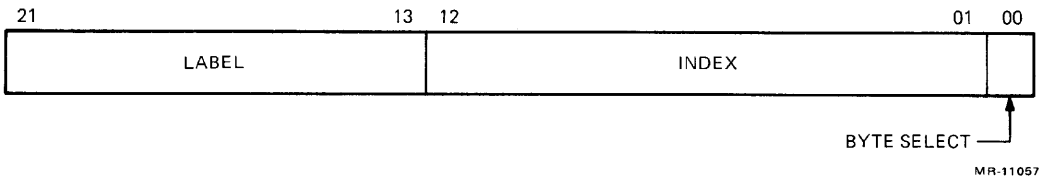


Figure 1-23 Cache Physical Address

The index field is used to select the cache entry. The index is 12 bits long, selecting one of 4096 separate cache entries. Each cache entry contains a 9-bit tag field (TAG), tag parity bit (P), tag valid bit (V), two bytes of cache data (B0 and B1) and two corresponding byte parity bits (P0 and P1). (See Figure 1-24.)

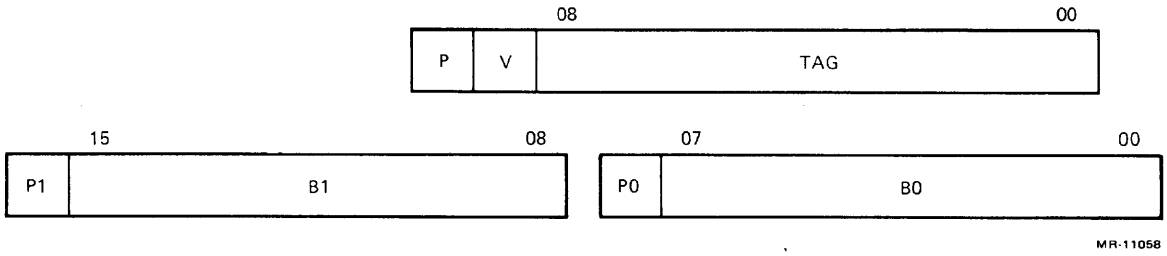


Figure 1-24 Cache Data Format

A physical address is considered cached when the tag field of the cache entry specified by the index field equals the label field, the valid bit is set, and no parity errors are seen. When a cache read hit occurs, i.e., the address is cached during a read operation, B1 and B0 are used as the source of the data. When a cache read miss occurs, i.e., the address is not cached, main memory is accessed to obtain the data.

A physical address is stored in the cache whenever the cache is allocated. To allocate the cache, the tag field of a cache entry specified by the index field is set equal to the label field, the V-bit is set, B1 and B0 are loaded with the fresh data, and the parity bits are correctly calculated. This guarantees that the next access to this address will report a cache hit. It should be noted that allocating the cache typically destroys a previously allocated valid cache entry. The cache is allocated whenever a read miss or word write miss occurs.

Write cycles are separated into word write and byte write operations. Main memory is always updated during writes. A cache hit will cause the proper byte(s) to be written in both the cache and in main memory. This is called *writing through* the cache. A cache miss during a word write will allocate the cache; however, since two bytes are allocated together, a byte write only updates main memory. The cache response matrix is summarized in Table 1-15.

The I/O page (top 8 Kb) is never cached and therefore always reports misses. This is because the I/O page contains dynamic status registers which, when read, must always convey the latest information.

When the system is powered up, the cache must be cleared and correct parity written into each entry. This is called flushing the cache.

Table 1-15 Cache Response Matrix

	DMA		CPU	
Operation	**Hit**	**Miss**	**Hit**	**Miss**
Read	Read memory– no cache change	Read memory– no cache change	Read cached data	Read memory– allocate cache
Write word	Invalidate cache– update memory	Update memory– no cache change	Write through cache to memory	Write memory– allocate cache
Write byte	Invalidate cache– update memory	Update memory– no cache change	Write through cache to memory	Write memory– no cache change
Read bypass	–	–	Read memory– invalidate cache	Read memory– no cache change
Write bypass	–	–	Write memory– invalidate cache	Write memory– no cache change
Read force miss	–	–	Read memory– no cache change	Read memory– no cache change
Write force miss	–	–	Write memory– no cache change	Write memory– no cache change

A potential stale data problem can occur when a DMA device writes to a cached location. The overwritten cache entry must be invalidated. To avoid this problem, the cache system monitors each DMA transaction to determine when the DMA transaction invalidates the cache. This also includes block mode DMA which is possible on the 22-bit LSI-11 bus.

For both diagnostic and availability reasons, it is important to be able to turn off the cache via software. The cache is disabled by setting either of the force cache miss bits, 02 and 03, in the cache control register. When disabled, all references are forced to miss the cache. That is, main memory is always accessed, cache parity errors are ignored, and no cache allocation is performed. The cache is essentially removed from the system. This is different than bypassing the cache. Bypass references access the main memory, check cache parity, and invalidate the cache entry if previously allocated. Read references that bypass the cache check for parity errors and will invalidate any address hits.

1.6.1 Parity

The KDJ11-A module has a main memory parity error detection mechanism. The BDAL<16> and <17> data lines are sampled when BDIN L is negated and the microprocessor initiates a memory read. The BDAL<16> bit is the parity error signal and the BDAL<17> bit is the parity abort error signal. When both are asserted (1), an abort occurs through the vector at virtual address 114 in kernel D space.

The cache memory also has a parity error detection mechanism. A parity error in the cache is not considered fatal because the main memory system has a backup copy of the data. The cache uses even parity for the even data bytes stored in the cache memory and odd parity for the odd data bytes stored in the cache memory. It also uses even parity for the tag field stored in the cache memory.

1.6.1.1 Parity Errors – A parity error indicates that a single bit error has occurred. Parity errors can occur in either the main memory or the cache memory. A main memory parity error is always fatal since the data stored in this memory is wrong and it cannot be restored. This type of parity error will always cause an abort through virtual address 114 in the kernel D space. Cache parity errors are not considered to be fatal since the data in the cache memory can be updated with the correct data from the main memory. When they occur, the KDJ11-A module will either abort, interrupt, or continue without an abort or interrupt. The action is determined by the state of bits 07 and 00 in the cache control register as defined in Table 1-16.

Table 1-16 Cache Parity Errors

CCR <07>	CCR <00>	Action
0	0	Update cache, interrupt through 114
0	1	Update cache only
1	X	Update cache, abort through 114 should only be used for diagnostics

1.6.1.2 Multiple Cache Parity Errors – If a cache parity error occurs while the error status from a previous cache parity error is not cleared from the memory system error register, then no abort or interrupt occurs. The main memory is accessed again to retrieve the correct data and the corrupted cache entry data is updated with the correct data. This prevents a cache hardware failure from generating an infinite series of interrupt or abort service loops.

1.6.2 Memory System Registers

The memory system registers consist of the cache control register, the memory system error register, and the hit/miss register. These registers are used by modules to control the memory system and report any errors that occur.

1.6.2.1 Cache Control Register (Address: 17 777 746) – The cache control register (CCR) controls the operation of the cache memory. The cache bypass, abort, and force miss functions can be controlled by software via this register. The cache control register is shown in Figure 1-25 and is described in Table 1-17. The register is cleared by either power-up or a console start. It is unaffected by the RESET instruction.

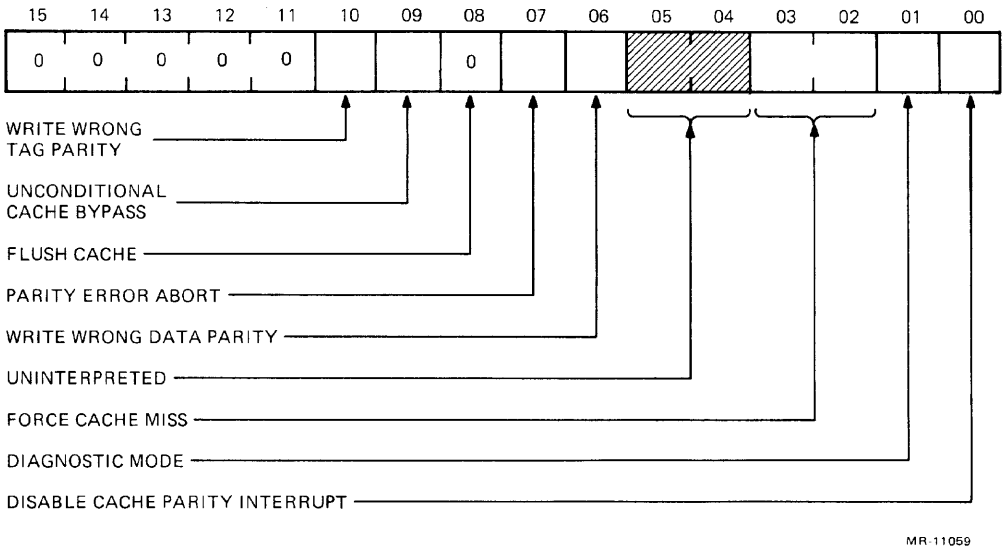


Figure 1-25 Cache Control Register (CCR)

Table 1-17 Cache Control Register Description

Bit	Name	Status	Function
15:11	Not used	–	–
10	Write wrong tag parity	Read/write	When set (1), this bit causes the cache tags to be written with wrong parity on all update cycles. This will cause a cache tag parity error to occur on the next access to that location.
09	Bypass cache	Read/write	When set (1), this bit forces all CPU memory references to go directly to main memory. Read hits will result in invalidation of accessed locations in the cache.
08	Flush cache*	Write only	When set (1), this bit causes the entire contents of the cache to be declared invalid. Writing a 0 into this bit will have no effect.
07	Enable parity error abort	Read/write	This bit is used with bit 0 to define the action taken as a result of a parity error. This bit is reserved for diagnostic purposes only.
06	Write wrong data parity	Read/write	When set (1), this bit causes high and low parity bytes to be written with wrong parity on all update cycles. This will cause a cache parity error to occur on the next access to that location.
05:04	Uninterpreted	–	These bits can be set or cleared under program control, but are not interpreted by the KDJ11-A.
03:02	Force miss	Read/write	When either is set, they force all CPU memory references to go directly to main memory. The cache tag and data stores are not changed. The parity is not checked. When set (1) these bits remove the cache memory from the system.
01	Diagnostic mode	Read/write	When set (1), all non-bypass and non-forced miss word writes will allocate the cache, irrespective of nonexistent memory (NXM) errors. In addition, NXM writes will not trap.
00	Disable cache parity interrupt	Read/write	Bits <07:00> specify the action to take following a cache parity error. If both bits are cleared (0) and a parity error occurs, an interrupt through vector 114 is generated. If bit 07 is cleared and bit 00 is set, a cache parity error neither aborts the reference nor generates an interrupt. In any case, all cache parity errors force a memory reference and update the cache with the fresh data.

* It takes approximately 1 millisecond to flush the cache. During this time DMA and interrupt requests are not serviced and no data processing occurs.

1.6.2.2 Hit/Miss Register (Address: 17 777 752) – The hit/miss register (HMR) records the status of the most recent cache accesses. The HMR is a shift register that records a hit as a 1 and a miss as a 0 for the most recent memory reads. A hit represents data located in the cache memory and a miss means the data is located in the main memory. Bit 00 represents the most recent memory access and is shifted to the left on successive memory access. The HMR is a read-only register and is shown in Figure 1-26.

1.6.2.3 Memory System Error Register (Address: 17 777 744) – The memory system error register (MSER) is a read-only register that is cleared by any write reference. The register monitors parity error aborts and records the type of parity error. The register is shown in Figure 1-27 and is described in Table 1-18. The memory system register is cleared by any write reference, during power-up, and by a console start. It is unaffected by the RESET instruction.

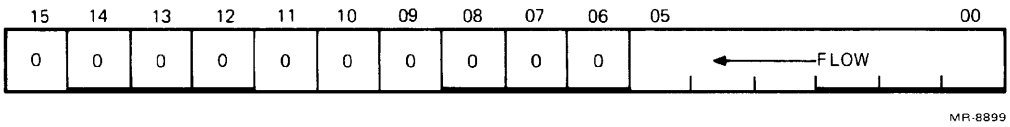


Figure 1-26 Hit/Miss Register (HMR)

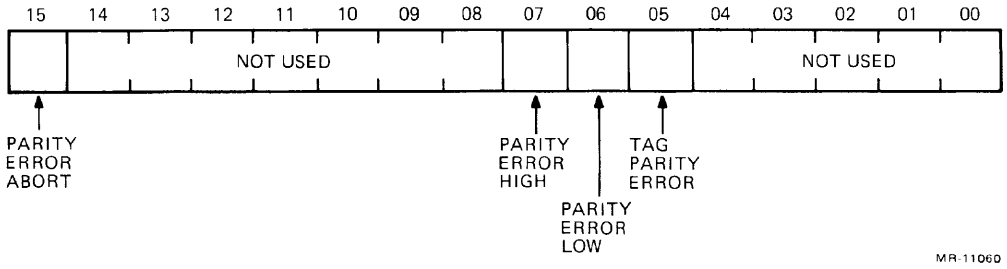


Figure 1-27 Memory System Error Register (MSER)

Table 1-18 Memory System Error Register

Bit	Name	Status	Description
15	Parity error abort	Read only	This bit is set (1) when cache or memory parity error aborts on instruction. Parity aborts occur on all main memory parity errors and when bit 07 of the CCR is set. A cache parity error occurs on a non-prefetch bus cycle.
14:08	Not used	–	–
07*	Parity error high	Read only	This bit is set (1) when the parity error was caused by the high byte data.
06*	Parity error low	Read only	This bit is set (1) when the parity error was caused by the low byte data.
05*	Tag parity error	Read only	This bit is set (1) when the parity error was caused by the tag field.
04:00	Not used	–	–

* Bits <07:05> are individually set when a cache parity error occurs and CCR bit 07 is set. All three bits are set when the CCR bit 07 is cleared and a cache parity error occurs irrespective of where the error occurred.

1.7 FLOATING-POINT

The KDJ11-A uses the floating-point instruction set to perform all floating-point arithmetic operations and converts data between integer and floating-point formats. It uses similar address modes and the same memory management facilities of the processor. The floating-point instructions can reference the floating-point accumulators, the general registers, or any location in memory.

1.7.1 Floating-Point Data Formats

Mathematically, a floating-point number may be defined as having the form (2 ** K) * f, where K is an integer and f is a fraction. For a nonvanishing number, K and f are uniquely determined by imposing the condition $1/2 \leq f < 1$. The fractional part (f) of the number is then said to be *normalized*. For the number 0, f must be assigned the value 0, and the value of K is indeterminate.

The 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.

1.7.1.1 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 processor 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 fractions 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.

1.7.1.2 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 floating-point processor (FPP) reserves a biased exponent of 0 for this purpose, and any floating-point number with a biased exponent of 0 either traps or is treated as if it were an exact 0 in arithmetic operations. An exact or "clean" 0 is represented by a word whose bits are all 0s. A "dirty" 0 is a floating-point number with a biased exponent of 0 and a nonzero fractional part. An arithmetic operation for which the resulting true exponent exceeds 277_8 is regarded as producing a "floating overflow;" if the true exponent is less than -177_8, the operation is regarded as producing a "floating underflow." A biased exponent of 0 can thus arise from arithmetic operations as a special case of overflow (true exponent = -200_8). (Recall that only eight bits are reserved for the biased exponent.) The fractional part of results obtained from such overflow and underflow is correct.

1.7.1.3 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 ensure 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 FIUV bit discussed later), is intended to provide the user with a debugging aid: if −0 occurs, it did not result from a previous floating-point arithmetic instruction.

1.7.1.4 Floating-Point Data – Floating-point data is stored in words of memory as illustrated in Figures 1-28 and 1-29.

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 1-30.)

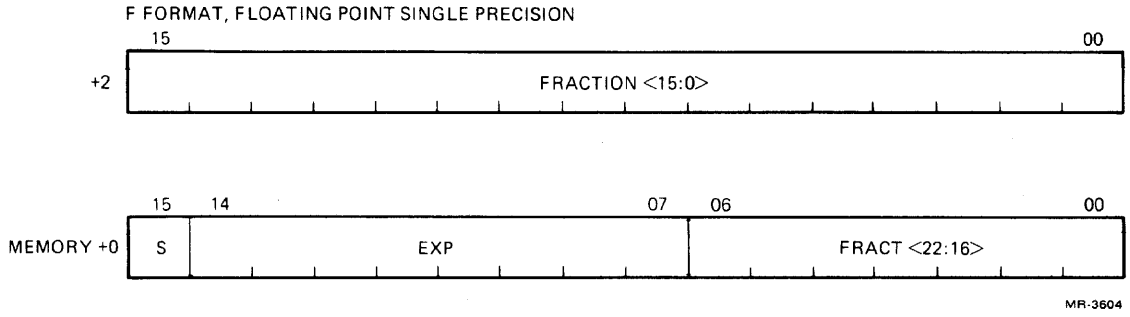


Figure 1-28 Single-Precision Format

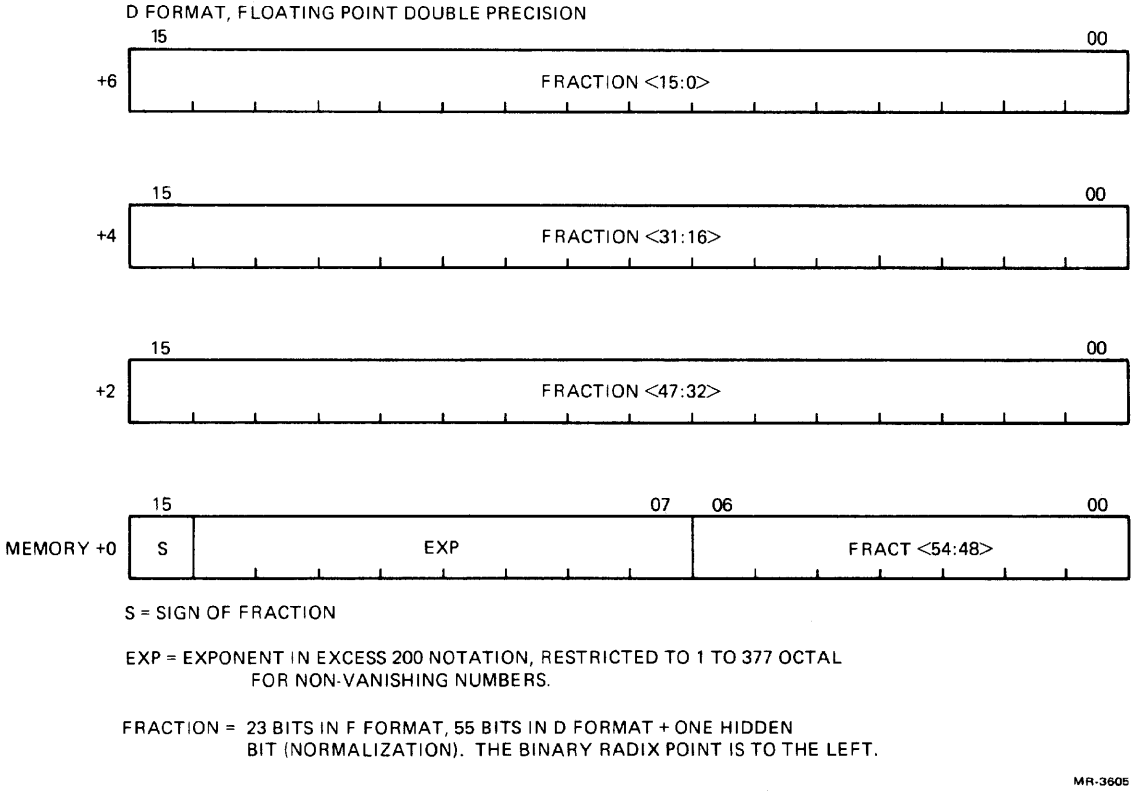


Figure 1-29 Double-Precision Format

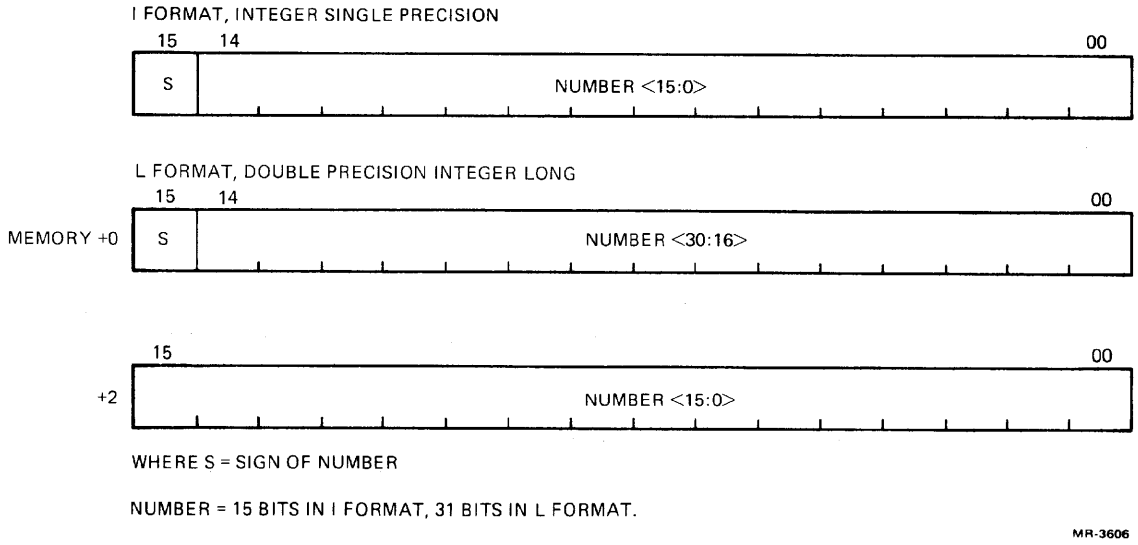


Figure 1-30 2's Complement Format

1.7.2 Floating-Point Registers

The floating-point registers are defined as six accumulators, the floating-point status register, the floating-point exception address register, and the floating-point exception code register, as shown in Figure 1-1.

1.7.2.1 Floating-Point Accumulator – Six 64-bit accumulators (AC0–AC5) are implemented for the temporary storage and manipulation of 32-bit and 64-bit floating-point data types.

1.7.2.2 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.

For the purposes of discussion, a set bit = 1 and a reset bit = 0. Three bits of the FPS register control the modes of operation as follows.

- Single/Double: floating-point numbers can be either single- or double-precision.
- Short/Long: integer numbers can be 16 bits or 32 bits.
- Chop/Round: the result of a floating-point operation can be either chopped or rounded. The term *chop* is used instead of *truncate* to avoid confusion with truncation of series used in approximations for function subroutines.

The FPS register contains an error flag and four conditions codes (five bits): carry, overflow, zero, and negative, which are equivalent to the CPU condition codes.

The floating-point operation 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 zero
- Illegal floating op code

For the first four of these exceptions, bits in the FPS register are available to enable or disable interrupt individually. An interrupt on the occurrence of either of the last two exceptions can be disabled only by setting a bit which disables interrupts of all six of the exceptions as a group.

Of the 13 FPS bits described above, the error flag and condition codes are set by the FPP as part of the output of a floating-point instruction. Any of the mode and interrupt control bits may be set by the user; the LDFS instruction is available for this purpose. The FPS register is shown in Figure 1-31 and described in Table 1-19.

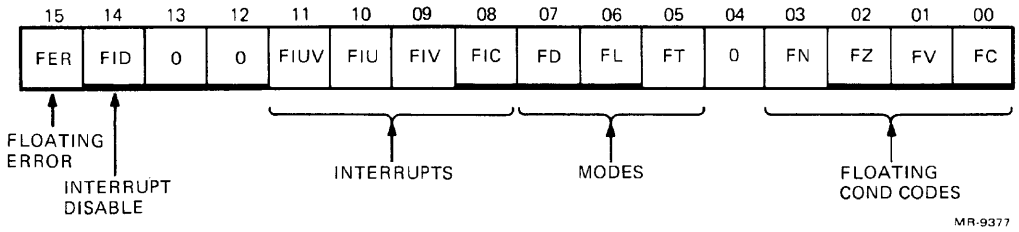


Figure 1-31 Floating-Point Status Register

Table 1-19 Floating-Point Status Bit Description

Bit	Name	Function
15	Floating error (FER)	This bit is set by a floating-point instruction if: • Division by zero occurs • Illegal op code occurs • Any of the remaining errors occur and the corresponding interrupt is enabled. This action is independent of the FID bit status. Also note 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 (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 this bit is set, all floating-point interrupts are disabled. 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 a FPP is always accompanied by an interrupt. Throughout the rest of this chapter, it is assumed that the FID bit is clear in all discussions involving overflow, underflow, occurrence of −0, and integer conversion errors.
13, 12	Not used	–

Table 1-19 Floating-Point Status Bit Description (Cont)

Bit	Name	Function
11	Interrupt on undefined variable (FIUV)	An interrupt occurs when this bit 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. 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 any AC operand of an arithmetic instruction; in particular, trap on −0 never occurs in mode 0. The FPP will not store a result of −0 without a simultaneous interrupt.
10	Interrupt on underflow (FIU)	When this 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 (octal) 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.
09	Interrupt on overflow (FIV)	When this 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 (octal). If the FIV is reset and overflow occurs, there is no interrupt. The FPP returns to exact 0. Special cases of overflow are dis- cussed in the detailed descriptions of the MOD and LDEXP instructions.
08	Interrupt on integer conversion (FIC)	When this bit is set and conversion to integer instruction fails, an interrupt will occur. If the interrupt occurs, the destination is set to 0, and all other registers are left untouched. If the FIC bit is reset, the result of the operation will be the same as detailed above, but no interrupt will occur. The conversion instruction fails if it generates an integer with more bits than can fit in the short or long integer word specified by the FL bit (bit 06)
07	Floating double-precision mode (FD)	This bit determines the precision that is used for floating-point calculations. When set, double-precision is assumed; when reset, single-precision is used.
06	Floating long integer mode (FL)	This bit is used in conversion between integer and floating-point format. When set, the integer format assumed is double-precision 2's complement (i.e., 32 bits). When reset, the integer format is assumed to be single-precision 2's complement (i.e., 16 bits).
05	Floating chop mode (FT)	When this bit is set, the result of any arithmetic operation is chopped (or truncated). When reset, the result is rounded.
04	Not used	-
03	Floating negative (FN)	This bit is set if the result of the last floating-point operation was negative; otherwise, it is reset.
02	Floating zero (FZ)	This bit is set if the result of the last floating-point operation was 0; otherwise, it is reset.
01	Floating overflow (FV)	This bit is set if the last floating-point operation resulted in an exponent overflow; otherwise, it is reset.
00	Floating carry (FC)	This bit is set if the last operation resulted in a carry of the most significant bit. This can only occur in a floating or double-to-integer conversion.

1.7.2.3 Floating-Point Exception Registers (FEC, FEA) – One interrupt vector is assigned to take care of all floating-point exceptions (location 244). 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 zero error
6	Floating or double-to-integer conversion error
8	Floating overflow error
10	Floating underflow error
12	Floating undefined variable error

The address of the instruction producing the exception is stored in the floating exception address (FEA) register.

The FEC and FEA registers are updated when one of the following occurs.

- Divide by zero
- Illegal op code
- Any of the other four exceptions with the corresponding interrupt enabled

If one of the four exceptions occurs with the corresponding interrupt disabled, the FEC and FEA are not updated. Inhibition of interrupts by the FID bit does not inhibit updating of the FEC and FEA, if an exception occurs. The FEC and FEA are not updated if no exception occurs. This means that the store status (STST) instruction will return current information only if the most recent floating-point instruction produced an exception. Unlike the FPS register, no instructions are provided for storage into the FEC and FEA registers.

1.7.3 Floating-Point Instruction Addressing

Floating-point 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.

1.7.4 Accuracy

General comments on the accuracy of the floating-point are presented here. The descriptions of the individual instructions, including the accuracy at which they operate, are listed in Chapter 7. An instruction or operation is regarded as "exact" if the result is identical to an infinite precision calculation involving the same operands. The prior 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 FPP for all instructions.

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 FPP 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 09) of the status register.

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 floating overflow (FV) bit of the floating exception (FEC) 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 enabled, 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.

1.8 SOFTWARE SYSTEMS

The KDJ11-A module can run the RT-11, RSX-11 V5.1, RSX-11 PLUS, RSTS/E, UNIX, and micropower PASCAL operating systems. These systems are described in the *PDP-11 Software Handbook* (EB 18687-20/80).

CHAPTER 2
INSTALLATION

2.1 INTRODUCTION

This chapter discusses the considerations and requirements to configure and install a KDJ11-A module in an LSI-11 system. The module can be installed in systems using the extended LSI-11 bus backplane as well as existing systems that use one of the standard LSI-11 backplanes. The items that must be considered before installing the module are as follows.

1. Configuration of the user selectable features.
2. Selection of an LSI-11 compatible backplane and mounting box.
3. Selection of LSI-11 options compatible with the KDJ11-A.
4. Knowledge of system differences when replacing an LSI-11 processor with the KDJ11-A module.

2.2 CONFIGURATION

The KDJ11-A has nine jumpers for the user selectable features. The locations of these jumpers are shown in Figure 2-1 and their functions are described in Table 2-1. A jumper is installed by pushing an insulated jumper wire (P/N 12-18783-00) onto the two wirewrap pins provided on the module.

Table 2-1 KDJ11-A Jumper Identification

Jumper	Function
W1	Bootstrap address bit 15
W2	Bootstrap address bit 14
W3	Power-up option selection bit 02
W4	Bootstrap address bit 13
W5	HALT trap option bit 03
W6	Bootstrap address bit 12
W7	Power-up option selection bit 01
W8	Wakeup disable
W9	BEVNT recognition

2.2.1 Power-Up Options

There are four power-up options available for the user to select. These options are selected by jumpers W7 and W3. The bits are set (1) when the jumpers are removed. A power-up option is selected by configuring W3 and W7, as described in Table 2-2. A description of each option is provided below.

Table 2-2 Power-Up Options

Option	W3	W7	Power-Up Mode
0	Installed	Installed	PC at 24, PS = 26
1	Installed	Removed	Micro-ODT, PS = 0
2	Removed	Installed	PC at 173000, PS = 340
3	Removed	Removed	Users bootstrap, PS at 340

2.2.1.1 Power-Up Option 0 – The processor reads physical memory locations 24 and 26 and loads the data into the PC and PS, respectively. The processor either services pending interrupts or starts program execution, beginning at the memory location pointed at by the PC.

2.2.1.2 Power-Up Option 1 – The processor unconditionally enters micro-ODT with the PS cleared. Pending service conditions are ignored.

2.2.1.3 Power-Up Option 2 – The processor sets the PC to 173000 and the PS to 340. The processor then either services pending interrupts or starts program execution, beginning at the memory location pointed at by the PC. This option is used for the standard bootstrap.

2.2.1.4 Power-Up Option 3 – The processor reads the four bootstrap address jumpers and loads the result into PC<15:12>. PC<11:00> are set to zero, and the PS is set to 340. The processor then either services pending interrupts, or starts program execution, beginning at the memory location pointed at by the PC.

2.2.2 HALT Option

The HALT option determines the action taken after a HALT instruction is executed in the kernel mode. At the end of a HALT instruction, the processor checks the BPOK bit 00 before checking the HALT option bit 03. If BPOK is set, the processor will recognize the HALT option, which is controlled by the W5 jumper. When the jumper is removed, bit 03 is set (1) and the processor will trap to location 4 in the kernel data space and set bit 07 of the CPU error register. When the jumper is installed, bit 03 reads as a zero and the processor enters the micro-ODT mode. If BPOK bit 00 is not set when the processor checks, the option is not recognized and the processor loops until BPOK is asserted and the power-up sequence is initiated.

2.2.3 Boot Address

The boot address jumpers selects the starting address for the user's bootstrap program when power-up option 3 is selected. The state of the highest four bits, <15:12>, is determined by jumpers W1, W2, W4, and W6, respectively. A bit will be set (1) when the respective jumper for that bit is installed and the bit will be read as a zero when the jumper is removed. During the power-up sequence, the processor reads the address determined by bits <15:12> and forces the remaining bits to read as zeros. Therefore, the user's bootstrap program can reside on any 2048 word boundary.

2.2.4 Wakeup Disable

The KDJ11-AA module has an onboard wakeup circuit to properly sequence the BDCOK signal. When jumper W8 is removed, the wakeup circuit is enabled and the module will properly sequence the BDCOK signal. The wakeup circuit will be disabled when W8 is installed and external logic must be used to properly sequence the BDCOK signal.

2.2.5 BEVNT Recognition

The LSI-11 bus signal BEVNT provides an external event interrupt request to the processor. This feature is disabled when the W9 jumper is installed and disables the line time clock register. When the jumper is removed, the BEVNT input is recognized and is under control of the line time clock register. Specifically, the signal is recognized by the module when bit 06 of the line time clock register is set (1) and is disabled when bit 06 is not set (0). The line time clock register address is 17 777 546 and is a read/write register.

2.2.6 Factory Configuration

The factory or shipped configuration is described in Table 2-3. The user should review these features and change them accordingly to match the requirements of the system using the module.

Table 2-3 Factory Configuration

Jumper	Status	Function
W1	Installed	Bit 15 set (1)
W2	Installed	Bit 14 set (1)
W3	Removed	Selects power-up option 2
W4	Installed	Bit 13 set (1)
W5	Removed	HALT instruction traps to location 4
W6	Installed	Bit 12 set (1)
W7	Installed	Selects power-up option 2
W8	Removed	Wakeup circuit is enabled
W9	Removed	BEVNT register is enabled

2.3 DIAGNOSTIC LEDS

The module has four LEDs that monitor the status of the module. The LEDs are designated as D1 through D4 and are located on the edge of the module, as shown in Figure 2-1. The D1 LED is turned on only when the module is operating in the micro-ODT mode. LEDS D2–D4 are used with the diagnostics and run during the power-up sequence. These LEDs are turned on at the beginning of the sequence and are turned off upon the successful pass of the diagnostic. Each LED monitors a primary function of the module operation, as described in Table 2-4. When troubleshooting the system, the LEDs indicate the most probable failure, as described in Table 2-5.

Table 2-4 LED Functions

LED On	Test Conditions
D1	Micro-ODT is entered.
D2	Module could not do a write and read transaction to the CPU error register. Indicates the microcode is not running.
D3	Module attempted to read location 17 777 560 and timed out. Indicates SLU is not responding.
D4	Module attempted to read location 0 and timed out or attempted to read location 17 777 700 and did not time out. Indicates the memory system is not responding.

Table 2-5 Probable System Failure

LEDs D1	D2	D3	D4	Probable Failure
X	On	On	On	CPU module
X	Off	On	On	LSI-11 bus
X	On	Off	On	CPU module
X	Off	Off	On	LSI-11 bus or memory
X	On	On	Off	CPU module
X	Off	On	Off	SLU module
X	On	Off	Off	CPU module
X	Off	Off	Off	Console terminal

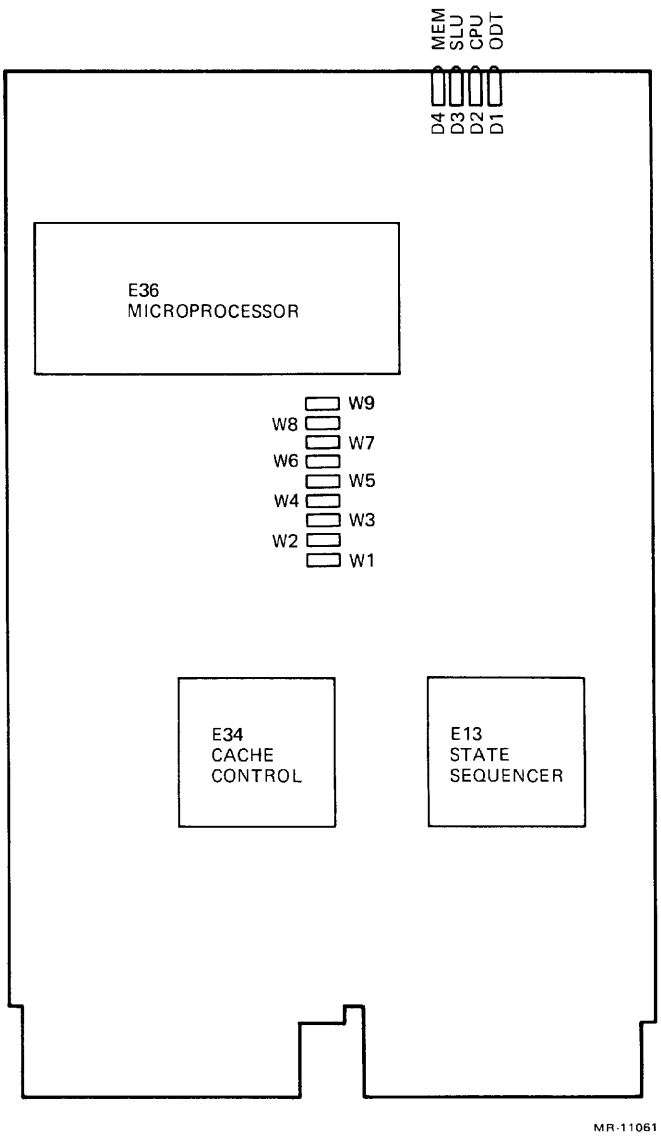


Figure 2-1 KDJ11-A Jumper Locations

2.4 MAINTENANCE REGISTER (ADDRESS 17 777 750)

The contents of the maintenance register is primarily determined by the user's selection of jumpers W1 through W7. In addition to these, the register bit 00 monitors the status of the LSI-11 bus signal BPOK, and bit 08 monitors the availability of a floating-point accelerator. The register is defined in Figure 2-2 and its contents are described in Table 2-6. It is a read-only register.

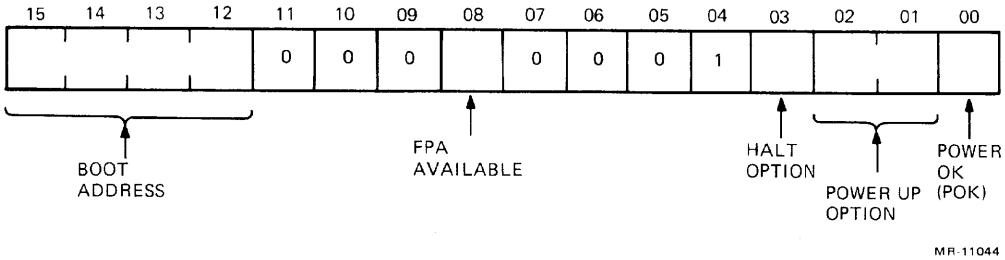


Figure 2-2 Maintenance Register

Table 2-6 Maintenance Register Bit Description

Bit	Name	Status	Function
15:12	Boot address	Read only	These bits read the user's boot address selected by jumpers W1, W2, W4, and W6. A 1 indicates the jumper is installed and a 0 indicates the jumper is removed.
11:09	Not used	Read only	Read as zeros
08	FPA available	Read only	A 1 indicates the presence of a floating-point accelerator and a 0 indicates that an accelerator is not installed.
07:04	Module ID	Read only	The 0001 code identifies to the microprocessor that this is a KDJ11-A module.
03	HALT	Read only	This bit reads the status of the W5 jumper. A 1 indicates the jumper is removed and a 0 indicates the jumper is installed.
02:01	Power-up	Read only	These bits read the user's power-up mode selected by jumpers W3 and W7. A 1 indicates the jumper is removed and a 0 indicates the jumper is installed.
01	POK	Read only	Reads as a 1 when BPOK H is asserted and the power supply is okay.

2.5 POWER-UP SEQUENCE

The power-up sequence for the module is shown in Figure 2-3.

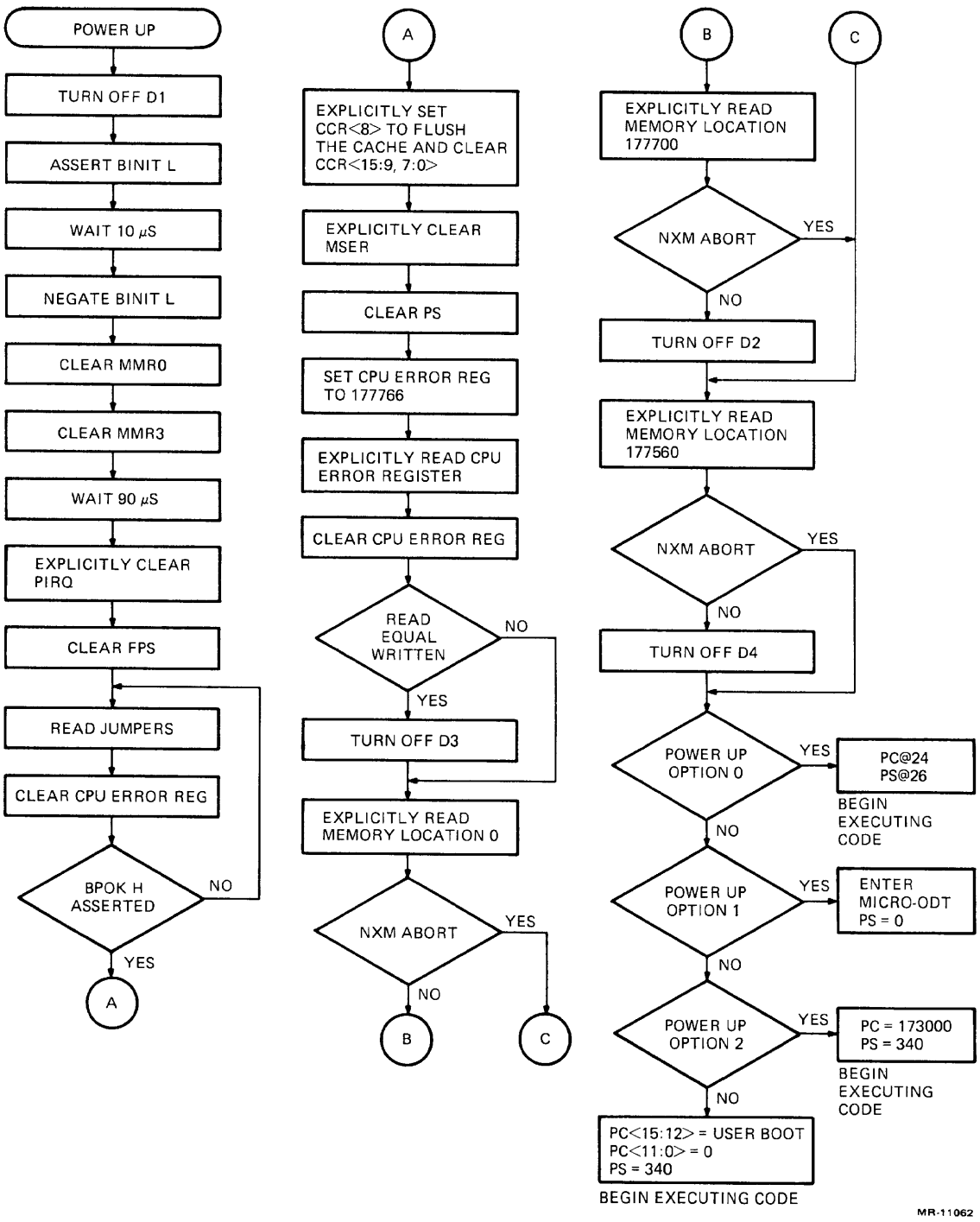


Figure 2-3 KDJ11-A Power-Up Sequence

2.6 POWER-DOWN SEQUENCE

The power-down sequence for the module is shown in Figure 2-4.

2.7 EXIT MICRO-ODT SEQUENCE

The micro-ODT mode is exited by the G command and the module sequence is shown in Figure 2-5.

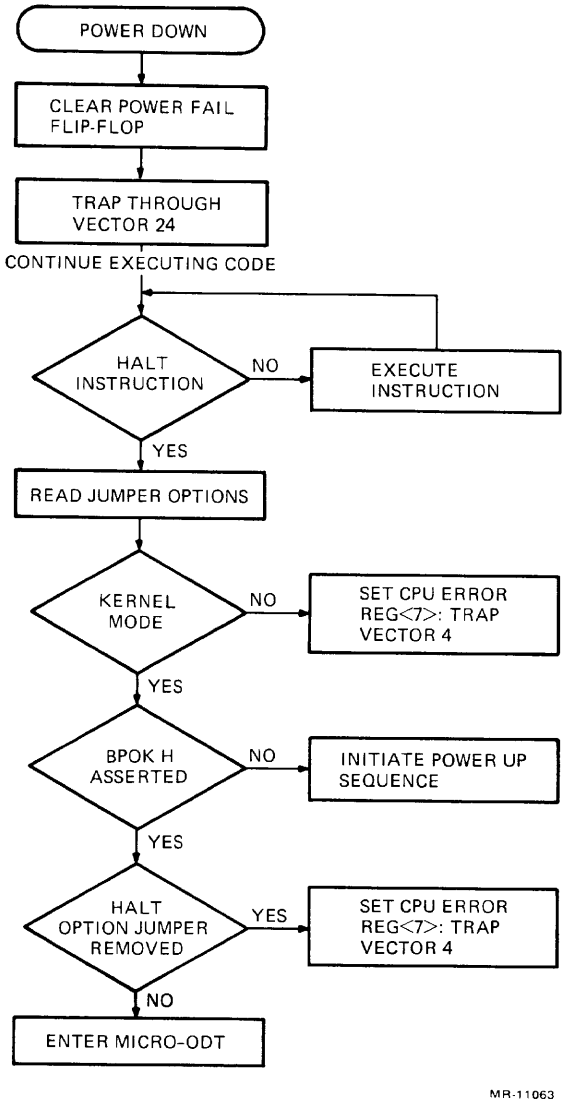


Figure 2-4 KDJ11-A Power-Down Sequence

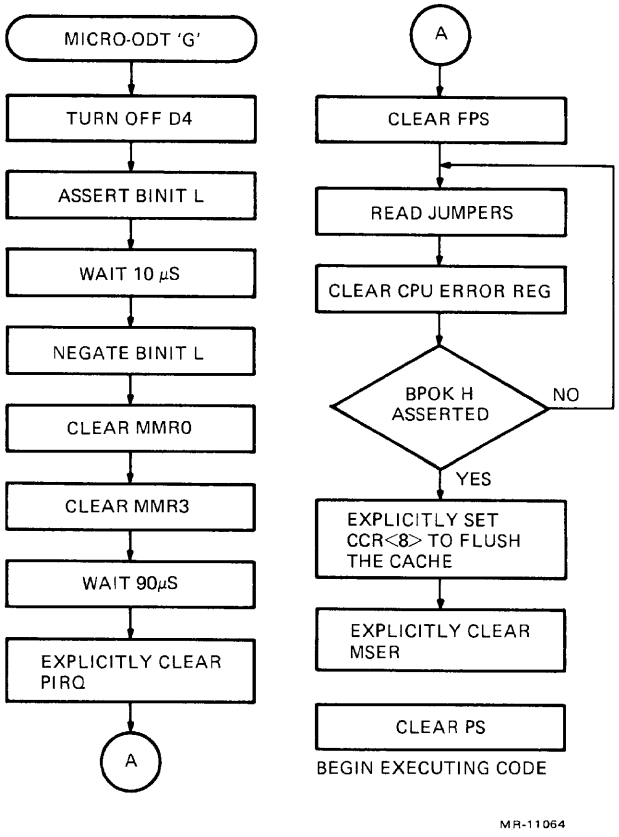


Figure 2-5 Micro-ODT Exit Sequence

2.8 MODULE CONTACT FINGER IDENTIFICATION

The LSI-11 type modules, including the KDJ11-A, all use the same contact (pin) identification system. Figure 2-6 identifies the contacts used on a dual-height module. The LSI-11 bus signals are carried on rows A and B, each with 18 contacts on the component side and the solder side. The KDJ11-A signals are identified along with the LSI-11 bus signals in Table 2-7. The pins are identified as follows.

AE2 Module Side Identifier Side (solder side)

Pin Identifier (Pin E)

Row Identifier (Row A)

The positioning notch between the two rows of pins mates with a protrusion on the connector block for the correct module positioning. A complete description of the backplane and bus operation is provided in Chapter 5.

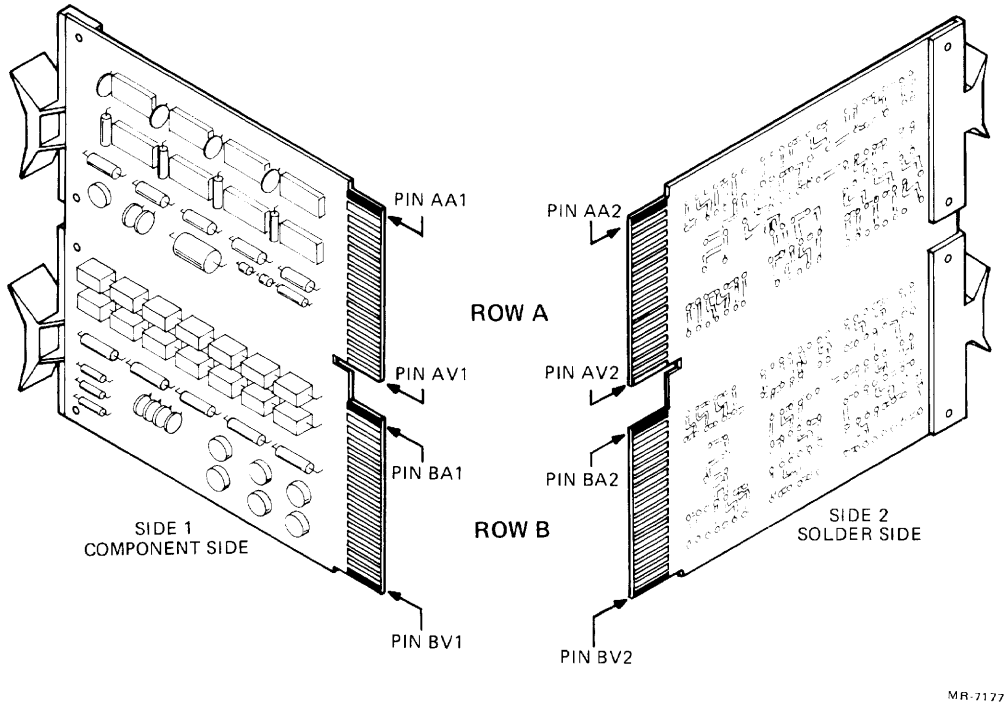


Figure 2-6 KDJ11-A Module Contacts

Table 2-7 KDJ11-A Module Signals

Component Side Pin	LSI-11 Bus	KDJ11-A	Solder Side Pin	LSI-11 Bus	KDJ11-A
AA1	BIRQ 5 L	BIRQ 5 L	AA2	+5	+5
AB1	BIRQ 6 L	BIRQ 6 L	AB2	−12	Not used
AC1	BDAL 16 L	BDAL 16 L	AC2	GND	GND
AD1	BDAL 17 L	BDAL 17 L	AD2	+12	Not used
AE1	SSPARE 1	Not used	AE2	BDOUT L	BDOUT L
AF1	SSPARE 2	SRUN L*	AF2	BRPLY L	BRPLY L
AH1	SSPARE 3	Not used	AH2	BDIN L	BDIN L
AJ1	GND	GND	AJ2	BSYNC L	BSYNC L
AK1	MSPARE A	Not used	AK2	BWTBT L	BWTBT L
AL1	MSPARE A	Not used	AL2	BIRQ L	BIRQ 4 L
AM1	GND	GND	AM2	BIAKI L	Not used
AN1	BDMR L	BDMR L	AN2	BAILO L	BIAK L
AP1	BHALT L	BHALT L	AP2	BBS 7 L	BBS 7 L
AR1	BREF L	Not used	AR2	BDMGI L	Not used
AS1	+12 B	Not used	AS2	BDMGO L	BDMG L
AT1	BND	GND	AT2	BINIT L	BINIT L
AU1	PSPARE 1	Not used	AU2	BDAL 0 L	BDAL 0 L
AV1	+5 B	+5 B	AV2	BDAL 1 L	BDAL 1 L
BA1	BDCOK H	BDCOK H	BA2	+5	+5
BB1	BPOK H	BPOK H	BB2	−12	Not used
BC1	SSPARE 4	BDAL 18 L	BC2	GND	GND
BD1	SSPARE 5	BDAL 19 L	BD2	+12	Not used
BE1	SSPARE 6	BDAL 20 L	BE2	BDAL 2 L	BDAL 2 L
BF1	SSPARE 7	BDAL 21 L	BF2	BDAL 3 L	BDAL 3 L
BH1	SSPARE 8	Not used	BH2	BDAL 4 L	BDAL 4 L
BJ1	GND	GND	BJ2	BDAL 5 L	BDAL 5 L
BK1	MSPARE B	Not used	BK2	BDAL 6 L	BDAL 6 L
BL1	MSPARE B	Not used	BL2	BDAL 7 L	BDAL 7 L
BM1	GND	GND	BM2	BDAL 8 L	BDAL 8 L
BN1	BSACK L	BSACK L	BN2	BDAL 9 L	BDAL 9 L
BP1	BIRQ 7 L	BIRQ 7 L	BP2	BDAL 10 L	BDAL 10 L
BR1	BEVNT L	BEVENT L	BR2	BDAL 11 L	BDAL 11 L
BS1	PSPARE 4	Not used	BS2	BDAL 12 L	BDAL 12 L
BT1	GND	GND	BT2	BDAL 13 L	BDAL 13 L
BU1	PSPARE 2	Not used	BU2	BDAL 14 L	BDAL 14 L
BV1	+5	+5	BV2	BDAL 15 L	BDAL 15 L

* The SRUN L signal is primarily used to drive a panel run light indicator. It is used for BA11-N and later systems. It indicates the processor is executing instructions.

2.9 HARDWARE OPTIONS

The KDJ11-A module can be configured into an operating system using a variety of backplanes, power supplies, enclosures, and LSI-11 type modules.

2.9.1 LSI-11 Options

The LSI-11 options that are compatible with the KDJ11-A module are listed in Table 2-8. These options meet the following requirements and may be used in any KDJ11-A system configuration.

1. The backplanes, memory, and I/O devices must support 22-bit addressing.

2. These devices must use backplane pins BC1, BD1, BE1, BF1 and DC1, DD1, DE1, DF1, for the BDAL bits <18:21> only.

Table 2-8 LSI-11 Compatible Options

Name	Option	Identification
Backplanes		
H9275	4 × 9	LSI-11/LSI-11 backplane
H9276	4 × 9	LSI-11/CD backplane
Micro/PDP-11	4 × 8	LSI-11/CD and 4 × 5 LSI-11/LSI-11 backplane
Memory		
MCV11-D-D	M8631	CMOS nonvolatile memory
MSV11-D-L	M8059	MOS memory
MSV11-P	M8067	MOS memory
MXV11-B	M7915	Multifunction module
MRV11-D	M8578	PROM/ROM module
Options		
AAV11-C	A6008	D/A converter
ADV11-C	A8000	A/D converter
AXV11-C	A0028	D/A and A/D combination converter
DLV11	M7940	Asynchronous serial line interface
DLV11-E	M8017	Asynchronous serial line interface
DLV11-F	M8028	Asynchronous serial line interface
DLV11-J	M8043	Four asynchronous serial line interfaces (CS Rev. E or later, ECO M8043-MR002 installed)
DMV11-AC	M8053-MA	Synchronous communications interface
DMV11-AF	M8064-MA	Synchronous communications interface
DPV11	M8020	Programmable synchronous EIA line
DRV11	M7941	Parallel interface
DRV11-J	M8049	Parallel interface
DUV11	M7951	Programmable synchronous EIA Line
DZV11	M7957	4-line asynchronous EIA multiple
IBV11-A	M7954	IEEE instrument bus interface
KPV11-A	M8016	Power-fail and LTC generator (KPV11-B and -C are not compatible)
KWV11-C	A4002	Programmable real-time clock
LAV11	M7949	LA180 line printer interface
LPV11	M8027	LA180/LP05 printer interface
RLV12	M8061	RL01/2 controller
RQDX1	M8639	MSCP controller for RX50 floppy disk and RD51 Winchester
RXV11	M7946	RX01 interface
TSV05	M7196	Magnetic tape interface
Bus Cable Cards		
M9404		Cable connector
M9404-YA		Cable connector with 240 Ω terminators
M9405		Cable connector
M9405-YA		Connector with 120 Ω terminators
Boot ROMs		
MXV11-B2		Boot ROMs

2.9.2 Restricted LSI-11 Options

The LSI-11 options that are not compatible or restricted for use with the KDJ11-A module are listed in Table 2-9. Backplanes, memories, or I/O devices that are not capable of 22-bit addressing may generate or decode erroneous addresses if they are used in systems that implement 22-bit addressing. Memory and memory-addressing devices which implement only 16- or 18-bit addressing may be used in a 22-bit backplane, but the size of the system memory must be restricted to the address range of these devices (32 KW for systems with a 16-bit device, and 128 KW for systems with an 18-bit device).

Any device that uses backplane pins BC1, BD1, BE1, BF1 or DC1, DD1, DE1, DF1 for purposes other than BDAL <18:21> is electrically incompatible with the 22-bit bus and may not be used without modification to the hardware.

NOTE

Eighteen-bit DMA devices can potentially work in Q22 systems by buffering I/O in the 18-bit address space.

Table 2-9 Restricted or Noncompatible LSI-11 Options

Name	Option	Identification
Backplanes		
DDV11-B	6 × 9	Backplane (18-bit addressing only)
H9270	4 × 4	Backplane (18-bit addressing only)
H9273-A	4 × 9	Backplane (18-bit addressing only)
H9281-A, -B, -C	2 × n	Dual-height backplane n = 4, 8, and 12 (18-bit addressing only)
VT103 B.P.	4 × 4	Backplane (54-14008) (18-bit addressing only)
Memories		
MMV11-A	G653	Core memory (16-bit addressing only, Q-Bus required on C/D backplane connectors)
MRV11-AA	M7942	ROM (16-bit addressing only)
MRV11-BA	M8021	UV PROM-RAM (16-bit addressing only)
MRV11-C	M8048	PROM/ROM (18-bit addressing only)
MSV11-B	M7944	MOS (16-bit addressing only)
MSV11-C	M7955	MOS (18-addressing only)

Table 2-9 Restricted or Noncompatible LSI-11 Options (Cont)

Name	Option	Identification
MSV11-D/E	M8044/M8045	MOS (18-bit addressing only)
MXV11-A	M8047	Multifunction module (18-bit addressing only on memory, the memory can be disabled)
Options		
AAV11	A6001	D/A converter (Use of BC1 for purposes other than BDAL 18)
ADV11	A012	A/D converter (Use of BC1 for purposes other than BDAL 18)
BDV11	M8012	Bootstrap/terminator (CS Revision D or later for use with KDF11-A, or KDF11-B, EDD M8012-ML0002. CS Revision E or later for use in 22-bit systems, ECO M8012-ML005)
DLV11-J	M8043	Serial line interface (CS Revision E or later for use with KDF11-A, or KDF11-B, ECO M8043-M8002)
DRV11-B	M7950	DMA interface (18-bit DMA only)
KPV11-B, -C	M8016-YB, -YC	Power-fail/line-time clock terminator (Termination for 18 bits only)
KUV11	M8018	WCS (For use with KD11-B and KD11-BA processors only)
KWV11-A	M7952	Programmable real-time clock (Use of BC1 for purposes other than BDAL 18)
REV11	M9400	Terminator, DMA refresh, bootstrap (Bootstrap for use with KD11-B and KL11-HA processors only. Termination for 18 bits only. DMA refresh may be used in any system.)
RKV11-D	M72609	RK05 controller interface (16-bit DMA only)
RLV11	M8013 + M8014	RL01, 2 controller (18-bit DMA only, use of BC1 and BL1 for purposes other than BDAL 18 AND BDAL 19, requires CD-interconnect on backplane C/D connectors)
RXV21	M8029	RX02 interface (18-bit DMA only)
TEV11	M9400-YB	Terminator (Termination for 18 bits only)
VSV11	M7064	Graphics display (18-bit DMA only)

Table 2-9 Restricted or Noncompatible LSI-11 Options (Cont)

Name	Option	Identification
Bus Cable Cards		
M9400-YD		Cable connector (18-bit bus only)
M9400-YE		Cable connector with 240 Ω terminators (18-bit bus only)
M9401		Cable connector (18-bit bus only)
Boot ROMs		
MXV11-A2		Boot ROMs

2.9.3 Enclosures

The KDJ11-A module 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 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 mounting boxes.

2.10 SYSTEM DIFFERENCES

The KDJ11-A module does not have a bootstrap loader, serial line interface, I/O bus map, real-time clock, or memory. A complete listing of the differences between the module and other LSI-11 type processor modules are listed in Appendix B.

Several key system differences between the KDF11-A and KDJ11-A modules are highlighted below.

1. The KDJ11-A contains an on-board line time clock register (LTC). No LSI-11 bus cycle is started when the LTC register is accessed at its bus address of 17 777 546. The access is completely contained on board the KDJ11-A and does not use the LSI-11 bus. Therefore, an LSI-11 bus option register addressable at 17 777 546 can never be accessed.

   An example of a problem this causes with options can be found in the BDV11 option (M8012). The BDV11 contains an LTC register which disables recognition of the LSI-11 bus signal BEVNT by continually asserting BEVNT. Since only the negative edge of BEVNT triggers the interrupt through location 100, recognition of BEVNT is disabled by this action. The LTC register on the BDV11 powers-up with BEVNT disable and will only release its grip when a programmer writes to the register. When the BDV11 is used with a KDJ11-A, the BDV11's copy of the LTC can never be written and, therefore, unless the BDV11 is configured with switch B5 in the off position, all BEVNT interrupts are forever blocked. Switch B5 disconnects the BEVNT signal from the BDV11.

   In general, no option should contain a register at address 17 777 546.

2. Bit 11 in the processor status (PS) word selects the alternate register set in the KDJ11-A. This bit is not implemented in the KDF11-A. Interrupt vectors should not specify the alternate register set.

3. Odd word addresses cause addressing error traps (through location 4) in the KDJ11-A. The KDF11-A does not generate any error condition when word references are addressed with odd addresses. Any existing code which generates odd word addresses will not work on the KDJ11-A. The existing BDV11 has code that generates odd word addresses.

   The BDV11 generates the error in the ROM diagnostics. The BDV11 can bypass the error code if the diagnostics are eliminated (switches A1 and A2 off).

4. BDAL <21:13> are driven as "110000111" during I/O references (BBS7 asserted). The KDF11-A drives these bits differently: "000000111" when memory management is turned off, "000011111" when 18-bit memory management is selected, and "111111111" when 22-bit memory management is selected.

2.11 KDJ11-A SYSTEM

A KDJ11-A module can be installed to upgrade an existing Digital system or a custom-built system using LSI-11 components. The existing system must be either a KDF11-A or KDF11-B processor. There are three considerations that must be addressed to upgrade a system.

1. The boot mechanism
2. 18- or 22-bit addressing system
3. Single or multiple box system

If the system processor is not a KDF11-A or KDJ11-A, such as the 11/03 and 11/03L, it should not be considered for upgrade.

In the following upgrade descriptions, the systems have been labeled as being field serviceable or not. A system which is field serviceable has a bootstrap which meets Field Service requirements. However, there is no guarantee that the overall system will be field serviceable.

NOTE

It is recommended that the ac and dc loading for the final configuration be checked for conformance with the Q-bus loading rules. It is also recommended to check for overloading on the +5 V and +12 V power supplies.

For each system upgrade, Table 2-10 lists the parameters for both the old system and the upgraded system.

2.12 MODULE INSTALLATION PROCEDURE

Certain guidelines should be followed when installing or replacing a KDJ11-A 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.
4. Insert the KDJ11-A module into the first slot or position in the backplane with the component side facing up.
5. Ensure that either the module or the selected system components provide the power-up protocol.
6. Use a single switch to apply all power to the system.

Table 2-10 Upgrade Choices

Current System	KDJ11-A/MXV11-B or MRV11-D w/B2 ROM Field Serviceable	KDJ11-A/MXV11-A Not Field Serviceable	KDJ11-A BDV11 (1) Not Field Serviceable
18-Bit Systems			
Component upgrades			
KDF11-A/MXV11-A			
1 box	X	X	
Multibox	X	X	
KDF11-A/BDV11			
1 box	X	X(8)	X
Multibox	X(2)	X(2)	X(6)
PDP-11/23S system upgrades			
KDF11-BA (boot on CPU)			
1 slot required			
1 box	X	X	X
Multibox (3)	X(2)	X(2)	X(6)
PDP-11/23A system upgrades			
KDF11-A			
Same as component upgrades			
22-Bit Systems			
Component upgrades			
KDF11-A/MXV11-A (4)			
1 box	X	X	
Multibox (10)			
PDP-11/23 PLUS or MICRO/PDP-11 (7, 9)			
KDF11-B/BE (boot on CPU)			
1 slot required			
1 box	X	X(4)	X(5)
Multibox (3, 10)			

NOTES:

1. Disable the Processor and Memory test and also the BEVNT register on the BDV11.
2. Use BCV1A and BDV1B expansion cables.
3. It is not currently possible to expand out of the PDP-11/23-S or MICRO/PDP-11 box while maintaining FCC compliance.
4. Memory must be disabled.
5. Must have BDV11 ECO M8012-ML005 installed.
6. Use BCV2B cable set between the first and second box and BCV1A or BCV2B between second and third box. In a 3-box system, expansion cable set lengths must differ by 4 feet.
7. Neither the BDV11 nor the MXV11-A boot code support the RD51 (10 megabyte Winchester) or the RX50 5-1/4 inch diskettes.
8. Check ac loading, since termination was removed when the BDV11 was removed from the system.
9. The PDP-11/23 PLUS and MICRO/PDP-11 system upgrades will require an extra backplane slot to accommodate the additional boot module.
10. Not currently configurable with Digital equipment.

For further information regarding upgrade parts, contact your local Field Service Representative.

2.13 SPECIFICATIONS

Identification	M8192
Size	Dual
Dimensions	13.2 cm × 22.8 cm (5.2 in × 8.9 in)
Power Consumption	+5 V ±5% at 4.5 A (maximum)
AC Bus Loads	3.4 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	For ambient temperatures above 55°C, sufficient air flow must be provided to limit the module temperature to less than 65°C. For inlet temperatures below 55° C, air flow must be provided to limit temperature rise across the module to 10° C. Derate maximum temperature by 1°C (1.8°F) for each 305 m (1000 ft) above 2440 m (8000 ft).
Instruction Timing	See Appendix A.
DMA Latency	DMA latency is defined as the time between receiving a DMA request (MDMRL) and granting the request (BDMGL). The worst case DMA latency is 2.2 microseconds.

CHAPTER 3
CONSOLE ON-LINE DEBUGGING TECHNIQUE (ODT)

3.1 INTRODUCTION

A portion of the microcode in the KDJ11-A module emulates the capability normally found on a programmer's console. Since the KDJ11-A does not have a programmer's console (one with lights and switches) or a console switch register at bus address 17777570, the terminal at the standard bus address of 17777560 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 17777560 through 17777566 are generated in microcode and cannot be changed.

This feature is called the microcode on-line debugging technique, or micro-ODT. The KDJ11-A micro-ODT accepts 22-bit addresses, allowing it to access 4088 M bytes of memory, plus the 8 Kbyte I/O page. Micro-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 KDJ11-A 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 KDJ11-A does not provide a serial line interface on the module. Therefore, the console must interface with an LSI-11 serial line interface module connected into the backplane. This allows the console to communicate with the KDJ11-A via the LSI-11 bus.

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 (W5) is 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.

NOTE

Unlike the KD11-F and KD11-HA, the KDJ11-A 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 KDJ11-A creates a new stack at location 2 and continues to trap to 4. If a bus timeout occurs while getting an interrupt vector, the KDJ11-A 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 17777562_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 carriage return <CR> and line feed <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 07) in the RCSR at 17777560_8 is constantly being tested via a DATI by the processor for a 1. If bit 07 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 17777562 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 07 (DONE flag) of RCSR at 17777560_8 using a DATI bus cycle; if it is a 0, continue testing.

2. If RCSR bit 07 is a 1, read the low byte of RBUF at 17777562_8 using a DATI bus cycle.

3.4.2 Console ODT Output Sequence

The output sequence of ODT is as follows.

1. Test bit 07 (DONE flag) of the XCSR at 17777564_8 using a DATI bus cycle; if it is a 0, continue testing.

2. If XCSR bit 07 is a 1, write to the XBUF at 17777566_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 06) 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 07) 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 Q-Bus address space desired by the user (leading 0s are not required).

/ = command to open and print contents of location.

012525 = contents of octal location 1000.

<SPACE> = space character generated by 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 32 K word open segment and issues <LF>, ODT will open the first location of that segment. If the user wishes to cross the 32 K word boundary, the user must reenter the address for the desired 32 K word segment (i.e., ODT is modulo 32 K 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>, so that teletype printers will operate properly. 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, $ or R, 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. The R character was introduced for its being a one key stroke representation 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 04) of the processor status word cannot be modified by the user. This is to prevent the PDP-11 program debugging utilities (e.g., ODT-11), which use the T-bit for single-stepping, from being 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 G <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 floating-point status (FPS) register and the PS will be cleared to 0.

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

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 <15:08> and the second byte specifies starting address <07:00>. Bus address bits <21:16> are always forced to 0; the DUMP command is restricted to the first 32 K 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 KDJ11-A ADDRESS SPECIFICATION

The KDJ11-A micro-ODT accepts 22-bit addresses, allowing it to access 4088 M bytes of memory, plus the 8 Kbyte I/O page. All I/O page addresses must be entered by users with a full 22 bits specified. For example, if a user wishes to open the RCSR of the serial-line unit (SLU), he/she must enter 17777560, 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, 17777776. Processor registers R0 through R7 will not respond (i.e., timeout will occur) to bus addresses 17777700 through 17777707 if referenced in ODT.

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

Example: @17777572/000001 <SPACE>

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

The FP11 accumulators 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.

3.6.3 Entering of Octal Digits

In general, when the user is specifying an address or data, ODT will use the last eight digits if more than eight 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 data 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
FUNCTIONAL THEORY

4.1 INTRODUCTION

The KDJ11-A is a dual-height microprocessor module on a multilayer printed circuit board for use in an LSI-11 type system. Figure 4-1 shows the interconnecting data paths between the major functional blocks of the module which include the following.

- the DCJ11 microprocessor
- the cache data path and memory
- the state sequencer
- the input/output control circuits
- the bus interface input/output transceivers

The module uses a DCJ11 microprocessor CMOS chip to execute the PDP-11 instruction set described in Chapter 6, control the memory management, support the console micro-ODT and the other architectural features described in Chapter 1. The DCJ11 initiates all the KDJ11-A data transfers and operations. The cache data path contains the line time clock register and the memory system error register (MSER). The maintenance register is an on-board register that allows software to read the options selected by the user. The KDJ11-A provides an interface between the DCJ11 and the LSI-11 bus via the A-bus and B-bus data paths. The state sequencer is a 68-pin gate array that controls the module data transfers using the data paths. These include the read and write transactions to the cache memory and the system memory by sequencing the hand shake signals that control the LSI-11 bus.

An on-board 8 Kbyte direct map cache memory is provided. The cache data path chip is a 68-pin gate array that contains the control logic to support the cache memory. The cache memory is transparent to all programs and is designed with high-speed RAM memory. The memory locations currently being accessed from the system memory are automatically stored in the cache memory. The next time these locations are accessed, the data is retrieved from the cache memory and eliminates the time-consuming LSI-11 bus transaction. Full parity protection is provided for the cache memory and much of the parity calculations are done by the cache data path chip. The KDJ11-A monitors DMA writes into the system memory to ensure that the cache data does not become stale. Each DMA write address is checked to see if the address is cached, and if it is, the cached data is invalidated.

There are four LEDs on the module that provide a visual indication and monitor the status of the module. Three of the indicators are set during the power-up sequence to indicate when a hardware failure occurs. The fourth indicator is set when the module is in the micro-ODT mode. There is also a 40-pin socket provided on the module for a future floating-point accelerator option.

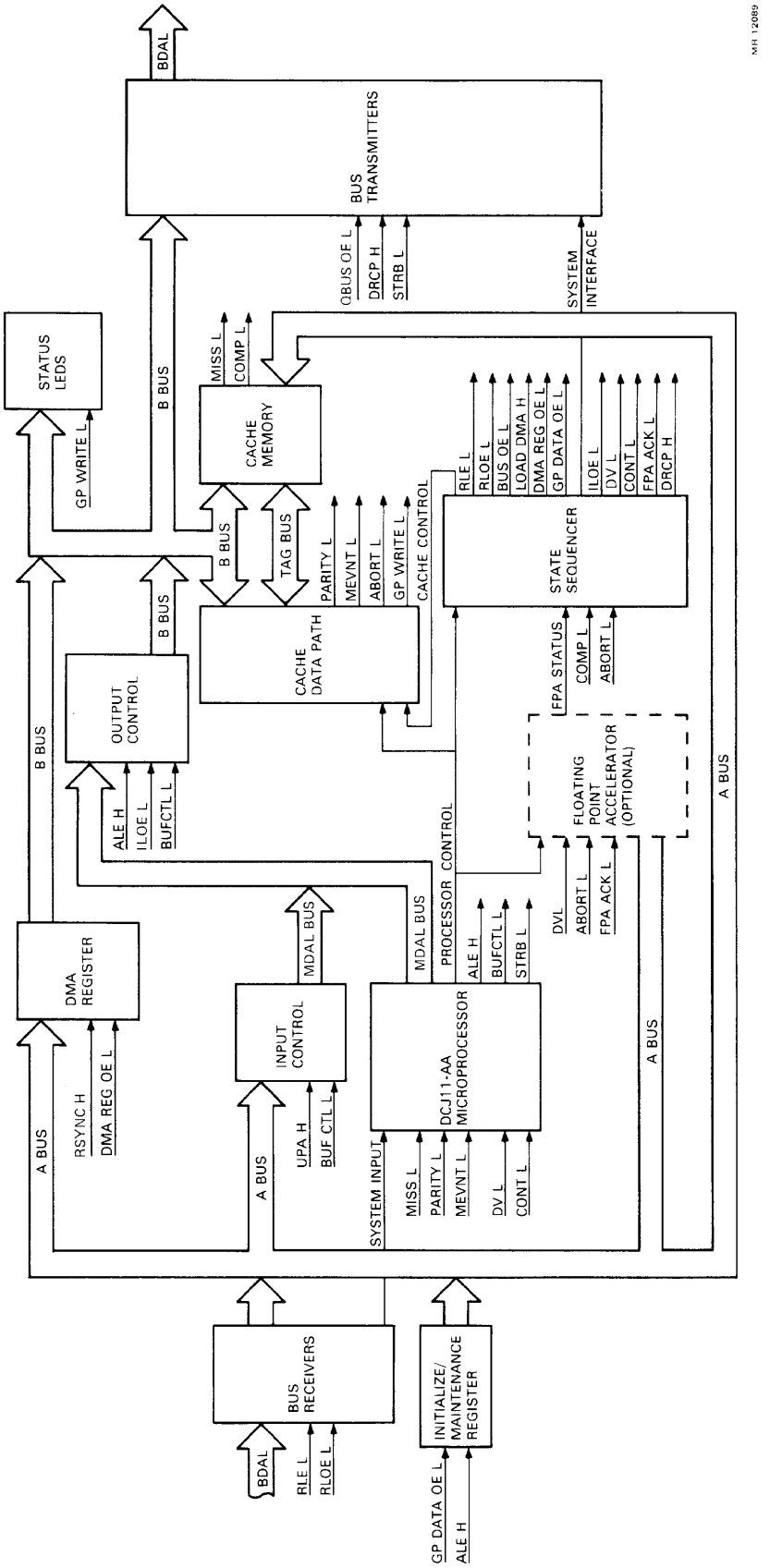


Figure 4-1 Functional Block Diagram

4.2 DCJ11 MICROPROCESSOR

The DCJ11 is a microprocessor contained on a 60-pin VLSI chip. The input/output pins are shown in Figure 4-2 and the signals are described below.

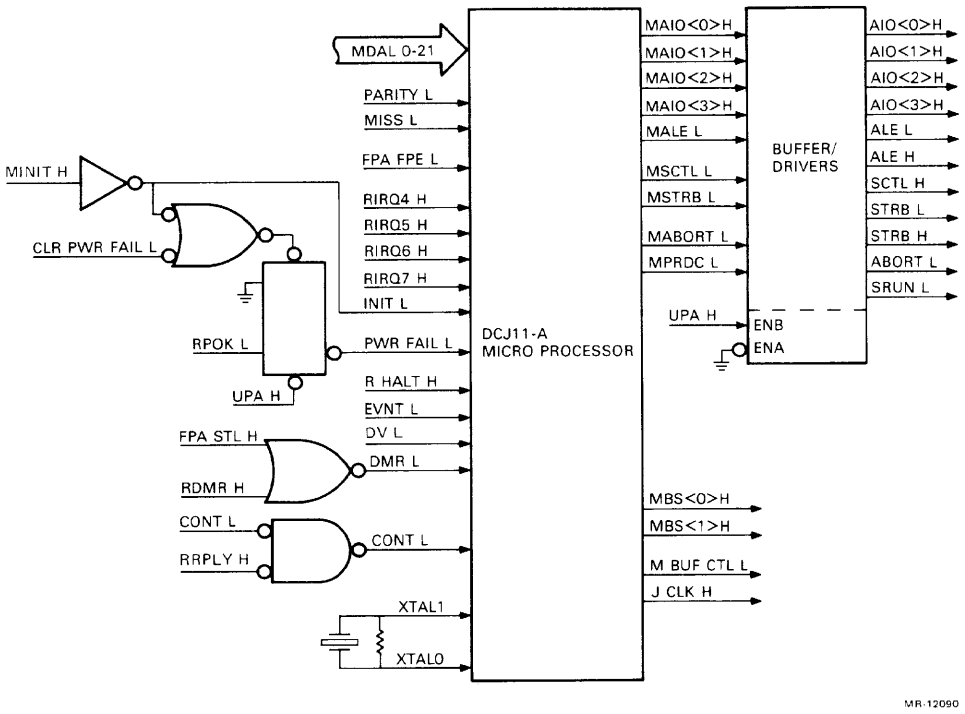


Figure 4-2 DCJ11-A Microprocessor

4.2.1 Initialization (MINIT L)

The MINIT L input is asserted by the BDCOK bus signal which must be asserted for a minimum of 1.5 microsecond. BDCOK H is asserted by the KDJ11-A when jumper W8 is removed. If jumper W8 is inserted, BDCOK H must be asserted externally in order to start the KDJ11-A. The DCJ11 starts the power-up sequence (described in Chapter 2) after MINIT L is asserted. MINIT L also clears the PWR FAIL circuit, initializes the state sequencer, asserts the LSI-11 bus initialization signal BINIT L, and turns on the diagnostic LEDs.

4.2.2 Output Signals

The DCJ11 output signals control the various module functions and are described below.

4.2.2.1 Address Input/Output (AIO<03:00> H) – These four signals classify the current transaction as a bus read, bus word write, bus byte write, GP read, GP write, interrupt acknowledge, or NOP as shown in Table 4-1.

Table 4-1 AIO Coding

AIO SIGNAL				
3	2	1	0	Type of Transaction*
1	1	1	1	Non I/O (NOP)
1	1	1	0	General-purpose read
1	1	0	1	Interrupt acknowledge (read vector)
1	1	0	0	Instruction stream request read
1	0	1	1	Read-modify-write, no bus lock
1	0	1	0	Read-modify-write, bus lock
1	0	0	1	Data stream read
1	0	0	0	Instruction stream demand read
0	1	0	-	General-purpose word write
0	0	1	-	Bus byte write
0	0	0	-	Bus word write

* The NOP, IACK, bus and general-purpose (GP) transactions are defined as follows.

1. A NOP transaction is an internal operation that does not require a bus transfer.
2. A bus transaction uses the DAL bus to access memory, I/O devices or explicit addressable registers.
3. A general-purpose transaction is used to access interface devices that are not directly addressable by the DAL bus.
4. Interrupt acknowledge (IACK) transactions are in response to the DCJ11 granting an interrupt request.

4.2.2.2 Bank Select, (BS1 H, BS0 H) – These signals are time multiplexed during the transaction. During the first portion of a bus transaction, they are used to define the type of address on the MDAL bus. The addresses identified by the BS0 H and BS1 H signals are defined in Table 4-2.

The memory types are all addresses below 17 600 000. The system register types are bus addressable registers in the address range of 17 777 740 to 17 777 751. The internal register types are addressable registers that reside within the DCJ11. The external I/O types are addresses greater than 17 577 777 which are neither internal registers nor system registers.

During the second half of the transaction, the BS1 H signal indicates the cache bypass status and the BS0 signal indicates the cache force miss status as described below.

BS1 H Asserted – Cache bypass
Negated – No cache bypass

BS0 H Asserted – Cache force miss
Negated – No cache force miss

Table 4-2 Bank Select Address Codes

BS1	BS0	Address Type
0	0	Memory
0	1	System register
1	0	External I/O
1	1	Internal register

4.2.2.3 Address Latch Enable (ALE L) – The ALE L output is asserted at the start of a transaction and latches the physical address, the AIO code and the BS1 H, BS0 H code. The negation of ALE L latches the cache hit/miss calculated data.

4.2.2.4 Stretch Control (SCTL L) – The SCTL L is asserted for the stretched portion of a transaction and negated when the DCJ11 receives CONT L input. When SCTL L is asserted, it generates the LSI-11 bus signal BSYNC L that is used for the LSI-11 bus read and write transactions. It also activates the ABORT L input/output signal.

4.2.2.5 Strobe (STRB L) – This signal is asserted at the end of the second DCJ11 clock period and is negated at the end of the transaction. The address is latched into the cache data path and the LSI-11 bus drivers when STRB L is asserted. The negation of STRB L clears the parity error flip-flop that drives the PARITY L input to the DCJ11.

4.2.2.6 Buffer Control (BUFCTL L) – The BUFCTL L is asserted to enable the input control logic for the A-bus to drive the MDAL bus. It is negated to enable the output control logic for the MDAL bus to drive the B-bus. The signal is asserted when the DCJ11 is reading data from the A-bus and negated when the DCJ11 is writing address or data information onto the B-bus.

4.2.2.7 Predecode Strobe (PRDC L) – The signal is asserted for the first two DCJ11 clock periods of any transaction that decodes a PDP-11 instruction. It also drives the SRUN L output of the module.

4.2.2.8 Clock (CLK H) – The CLK H output initiates and continuously clocks the timeout logic circuits used to detect nonexistent memory and the no BSACK L error condition.

4.2.3 Input Signals

The DCJ11 receives status and control information from a variety of input signals. These signals and their associated functions are described below.

4.2.3.1 MISS L – The MISS L input reports the cache memory hit and miss status during bus read and write transactions.

4.2.3.2 Data Valid (DV L) – The DV L input is generated by the state sequencer and is used to latch in read data from the MDAL bus.

4.2.3.3 Continue (CONT L) – The CONT L input is generated by the state sequencer and the LSI-11 bus signal BRPLY L to indicate that the current stretched transaction can end. It is only asserted when both the state sequencer enables the continue output, and the bus signal BRPLY L is negated on the LSI-11 bus.

4.2.3.4 DMA Request (DMR L) – The DMR L input is used to stall the DCJ11 by stretching the next transaction. It is asserted by the FPA STL L signal from the floating-point accelerator socket or by the LSI-11 bus signal BDMR L. The input is sampled at the beginning of the current transaction, and, when present, it will stretch the next transaction until the DMA or FPA transfer is complete.

4.2.3.5 IRQ <07:04> H – These inputs are coded priority levels from external devices that drive the LSI-11 bus signals BIRQ<07:04> L. The IRQ<07:04> H inputs are interrupt requests to the DCJ11 and are coded to determine a priority level. The acknowledgement of these inputs is dependent on the current priority level of the processor status word.

4.2.3.6 HALT H – The HALT H input is driven by the LSI-11 bus signal BHALT L and is the lowest interrupt priority for an external device.

4.2.3.7 EVNT H – The EVNT H input is driven by the LSI-11 bus signal BEVNT L and has a level-6 priority. This signal can be disabled by installing the W9 jumper or by software clearing bit 6 of the line time clock (LTC) register.

4.2.3.8 PWR FAIL L – This input is asserted by the power fail flip-flop which is set by the negation of the LSI-11 bus signal BPOK H. The flip-flop is reset by either MINIT L or CLR PWR FAIL L signals. This input is a nonmaskable interrupt to the DCJ11.

4.2.3.9 PARITY L – The PARITY L input is driven by the cache data path when a parity error is detected. This input is a nonmaskable interrupt to the DCJ11.

4.2.3.10 ABORT L – The ABORT L signal is an input/output line that can be driven by the DCJ11 or an external device such as the cache data path. The signal is used in conjunction with the PARITY L input to determine when the DCJ11 aborts the current transaction.

4.2.3.11 FPA FPE L – The FPA FPE L input is driven by the floating-point accelerator socket and is a nonmaskable interrupt request.

4.2.4 MDAL<21:00>

The MDAL<21:00> bus is a time-multiplexed data/address bus. The basic bus consists of DAL bits <15:00> and is bidirectional. DAL bits <21:16> are outputs only and used as the extended bus. The data being transmitted or received is dependent on the type of transaction being performed by the DCJ11.

4.2.5 DCJ11 Timing

The DCJ11 controls the type of transaction being executed and indicates this to the module circuits by coding the AIO<03:00> signals. There are six basic transactions performed and these are described as follows.

4.2.5.1 NOP – This transaction performs a DCJ11 internal operation and does not require the use of the MDAL bus. The normal transaction is shown in Figure 4-3. The stretched transaction (Figure 4-4) occurs when DMR is asserted early in the transaction and remains stretched until the CONT input is asserted to end the transaction.

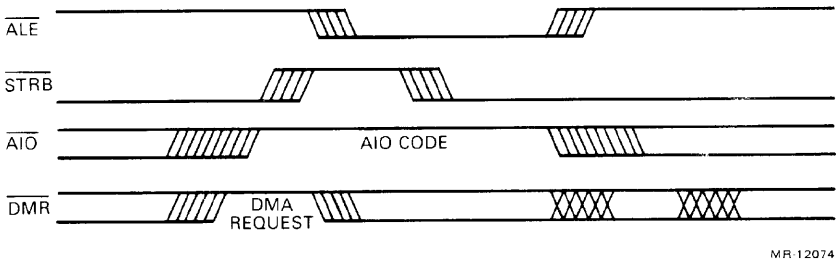


Figure 4-3 NOP Transaction

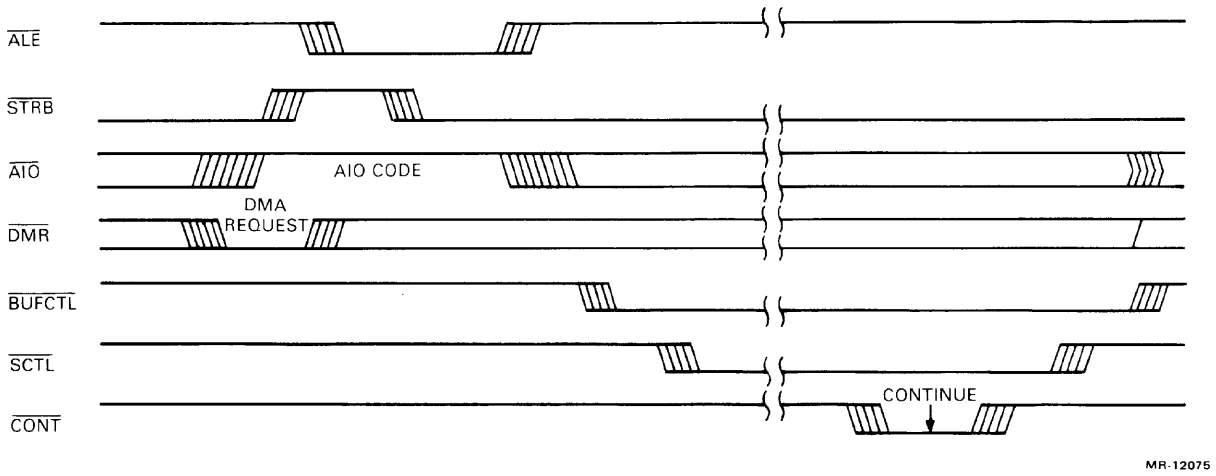


Figure 4-4 Stretched NOP Transaction

4.2.5.2 Bus Read – The bus read transaction uses the MDAL bus to read data from cache memory, main memory, input/output devices or the addressable module registers. These transactions occur during instruction stream reads, data stream reads and the read portion of read-modify-write. The transaction reads complete words and if only a byte is required, the DCJ11 ignores the excess byte. A cache bus read transaction (Figure 4-5) occurs when the physical address scores a hit in the cache memory. The DCJ11 will abort the transaction if any memory management or address errors assert the ABORT L signal. When this happens, all current information is ignored and the transaction is immediately aborted.

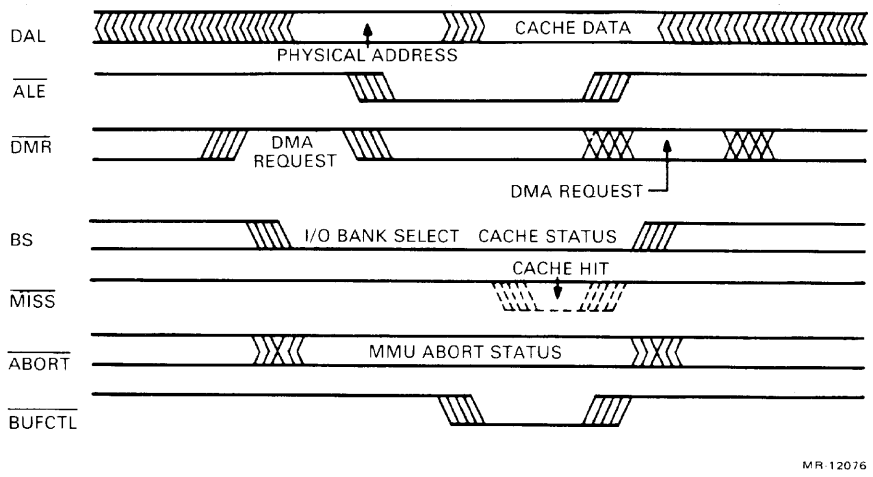


Figure 4-5 Bus Read Transaction

The non-cache or stretched bus read transaction (Figure 4-6) is used when the data must be accessed via the LSI-11 bus. This occurs when any of the following conditions exist.

1. Either BS1 H or BS0 H is set to one indicating an I/O address
2. Cache bypass is indicated
3. Cache force miss is indicated
4. DMR L is asserted
5. Cache MISS is reported

The BUFCTL L and SCTL L outputs are asserted during the stretched portion of the read transaction. The data is read by the DCJ11 when data valid (DV L) is asserted. When the transaction is stretched only because the DMR input was asserted, then DV L is not asserted because it will overwrite the valid data received from the cache. The transaction will remain stretched until the CONT L input is asserted to end the transaction.

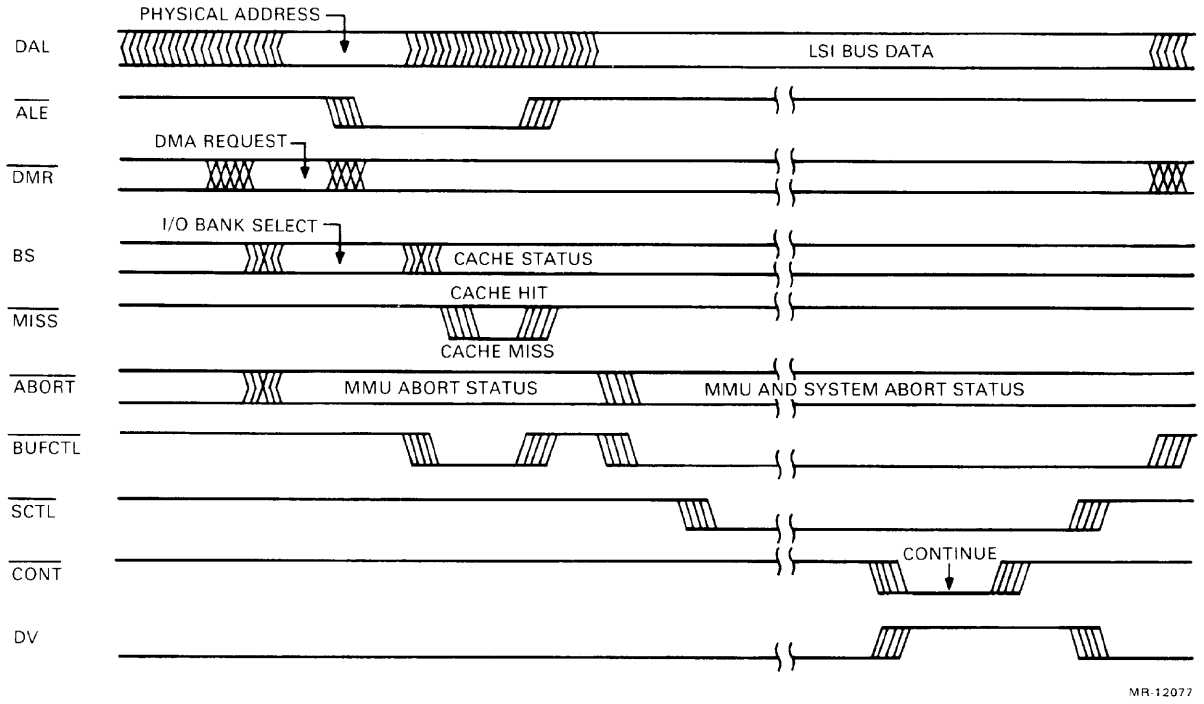


Figure 4-6 Stretched Bus Read Transaction

4.2.5.3 Bus Write – The bus write transaction writes data to memory, I/O devices, or other addressable registers via the DAL bus. The transaction can write either bytes or words as determined by the AIO code. The DCJ11 reports any memory management or address errors by enabling the ABORT L signal. This causes the transaction to be terminated immediately and all data should be ignored.

The write transaction as shown in Figure 4-7 and all bus write transactions are stretched. The SCTL L signal is asserted and the write data is on the bus during the stretched portion of the transaction. For byte writes, an even address selects the low byte and an odd address selects the high byte. The data for the remaining byte is not used.

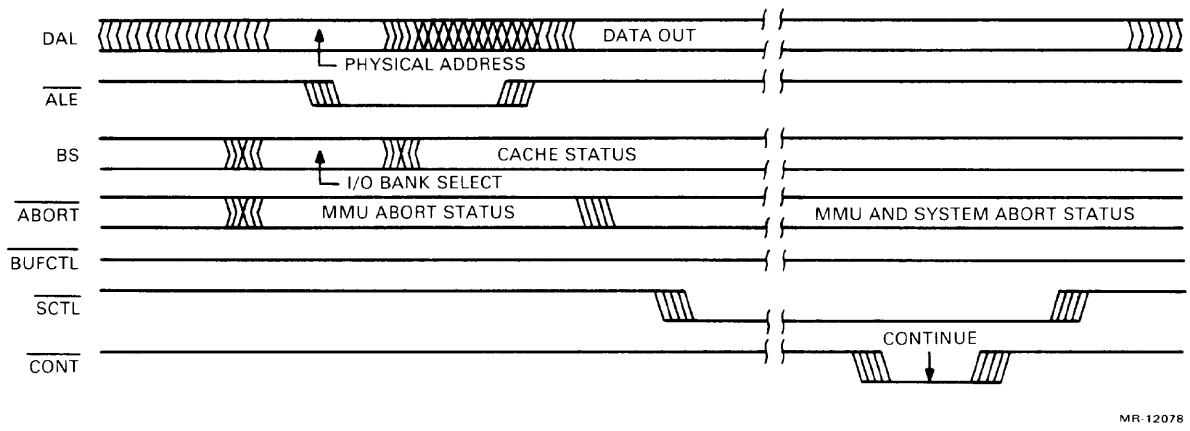


Figure 4-7 Bus Write Transaction

4.2.5.4 General-Purpose Read – The general-purpose read transaction accesses non-user-addressable module hardware. The MDAL address used for general-purpose reads is in the form of 17 777 XXX, where the "XXX" bits represent the general-purpose read code described in Table 4-3. The codes use MDAL bits <07:00> to access the hardware.

All general-purpose read transactions (Figure 4-8) are stretched. The DCJ11 reads the data when DV L is asserted. The transaction is stretched until CONT L is asserted to end the transaction.

Table 4-3 General-Purpose Read Codes

Code	Function
000	Reads the maintenance register during power up and determines the options selected by the user.
001	Reserved
003	Reserved

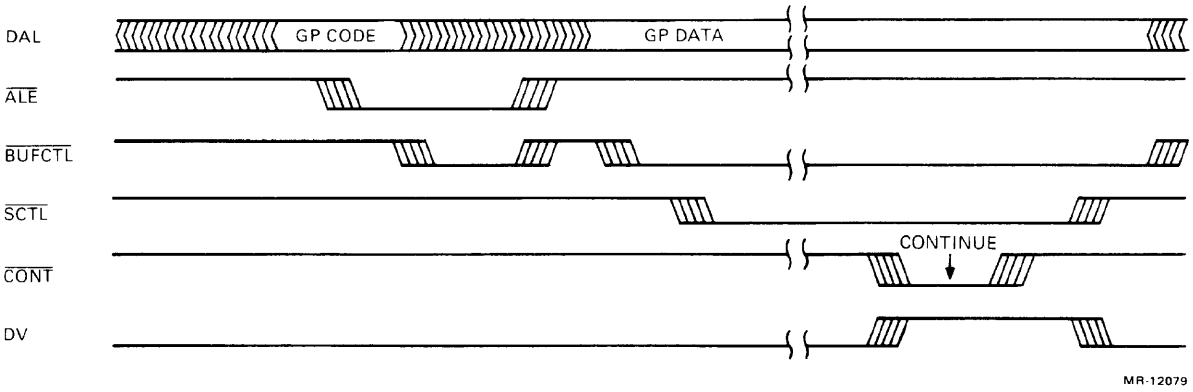


Figure 4-8 General-Purpose Read Transaction

4.2.5.5 General-Purpose Write – The general-purpose write transaction accesses non-user-addressable module hardware. The MDAL address used for general-purpose writes is in the form 17 777 XXX, where the "XXX" bits represent the general-purpose write code described in Table 4-4. The codes use MDAL bits <07:00> to access the hardware.

All general-purpose write transactions (Figure 4-9) are stretched. The DCJ11 writes the data when SCTL is asserted during the stretched portion of the transaction. The transaction is stretched until CONT L is asserted to end the transaction.

Table 4-4 General-Purpose Write Codes

Code	Function
003	Reserved
014	Asserts bus reset signal
034	Indicates exit from console (ODT) mode
040	Reserved for future use
100	Acknowledges EVENT interrupt
114	Negates bus reset signal
140	Acknowledges power fail
220	Microdiagnostic test 1 passed
224	Microdiagnostic test 2 passed
230	Microdiagnostic test 3 passed
234	Indicates entrance into console (ODT) mode

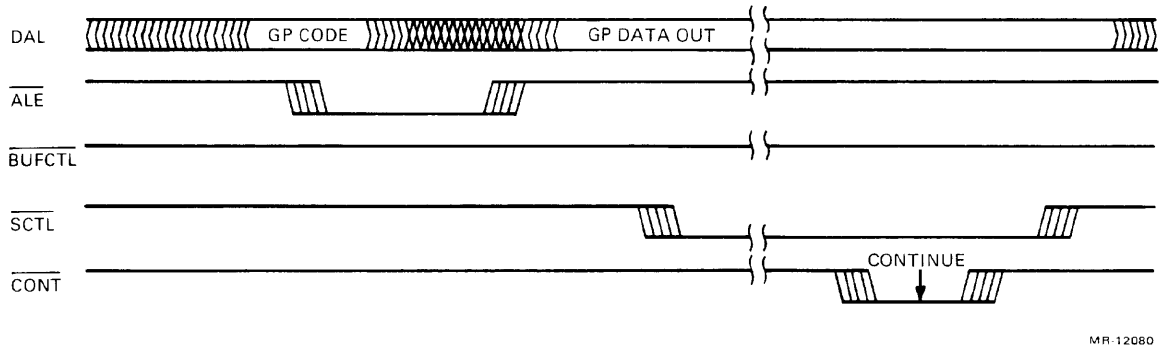


Figure 4-9 General-Purpose Write Transaction

4.2.5.6 IACK – The read interrupt vector transaction acknowledges an interrupt request received on one of the IRQ<03:00> inputs by reading a device interrupt vector. All interrupt vector transactions (Figure 4-10) are stretched. The device interrupt vector is latched by the DCJ11 when the DV L input is asserted.

4.3 STATE SEQUENCER

The state sequencer (Figure 4-11) controls the routing of address and data information on the KDJ11-A module and the LSI-11 bus handshaking signals. The module data path buses consist of the A-bus, B-bus and the MDAL bus. The MDAL bus is bidirectional; it interfaces with the A-bus by the input control logic and the B-bus by the output control logic. These data paths allow the DCJ11 to transmit addressing and data information on the B-bus to the LSI-11 bus, and receive read data on the A-bus from the LSI-11 bus. The A-bus and B-bus are also connected to the DMA register, which allows DMA addresses to connect to the B-bus.

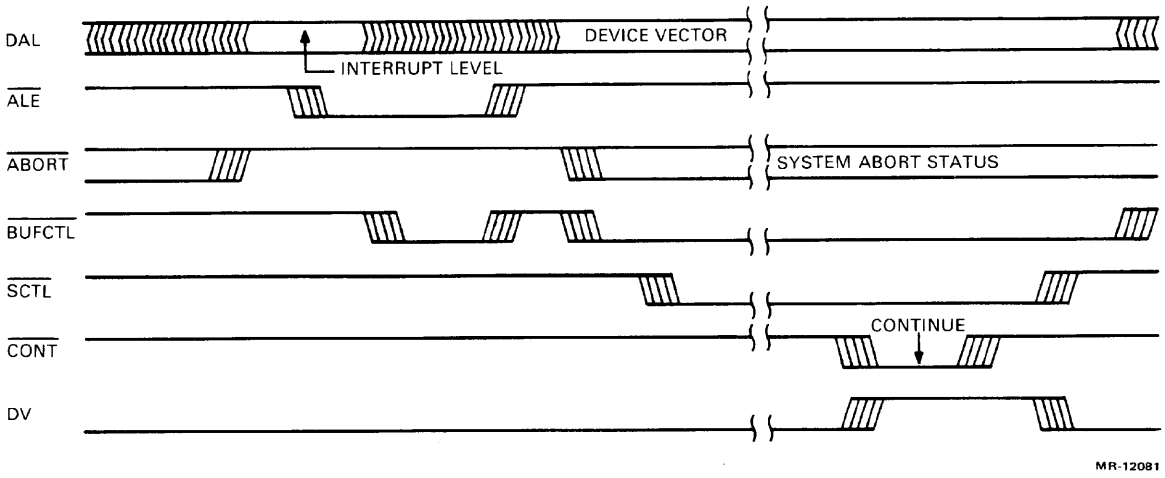


Figure 4-10 Interrupt Acknowledge Transaction

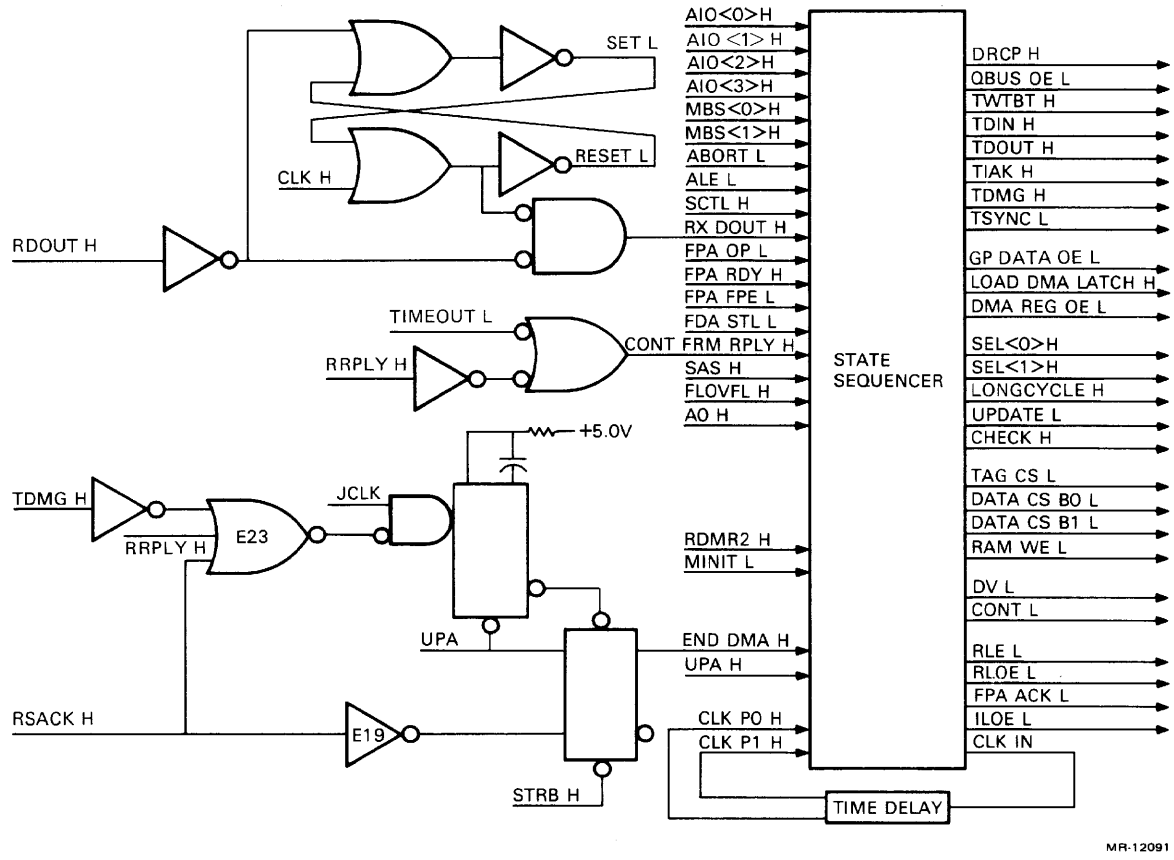


Figure 4-11 State Sequencer

The steady or quiescent state of the sequencer sets up the module data paths for high-speed cache memory read operation. When a transaction is stretched, the state sequencer leaves the steady state to control the module functions and the LSI-11 bus. This allows the module to perform memory read/write, interrupt vector reads, board register read/write, floating-point accelerator memory I/O, general purpose I/O, or DMA arbitration. A stretched transaction is initiated when SCTL L is asserted. This starts the state sequencer's clock and, if necessary, generates the LSI-11 bus signal BSYNC L. The CLK H output drives the external delay line to generate two delayed clock inputs of 40 ns and 60 ns. These are used to determine the cycle time of the sequencer and provide short periods of 80 ns or long periods of 120 ns. The state sequencer decodes the AIO inputs to identify the type of transaction and the BS1 H, BS0 H inputs to classify the address. The state sequencer provides control signals to the functional areas of the module to support the transaction being performed.

4.3.1 DCJ11

The state sequencer informs the DCJ11 when valid data is on the MDAL bus by asserting DV L. It also asserts the CONT L input to the DCJ11 when the transaction is completed. It receives the ABORT L and ALE L inputs from the DCJ11.

4.3.2 LSI-11 Bus Signals

The state sequencer provides the handshaking control signals when the module is transmitting or receiving data via the LSI-11 bus. These signals are TWTBT H, TDIN H, TDOUT H, TIAK H, TDMG H and TSYNC. The use of these signals and the LSI-11 bus protocol are described in Chapter 5.

4.3.3 LSI-11 Bus Receivers

The LSI-11 bus data is latched into the bus receivers when RLE L is asserted and this data is driven onto the A-bus when RLOE L is asserted.

4.3.4 LSI-11 Bus Transmitters

The LSI-11 bus data is latched into the bus transmitters from the B-bus when the DRCP H signal is asserted and driven onto the LSI-11 bus when the Q-BUS OE L signal is asserted.

4.3.5 Maintenance Register

The maintenance register data is placed on the A-bus when GP DATA OE L signal is asserted.

4.3.6 DMA Register

The DMA register receives an address from the LSI-11 bus via the A-bus and latches it into the register when LOAD DMA LATCH H is asserted. The address is driven onto the B-bus to check it against the addresses in the cache memory when DMA REG OE L is asserted.

4.3.7 Cache Data Path

The cache data path provides the SAS H, FLOVFL H and A<0> H inputs to the state sequencer and receives the SEL <01:00> H, LONGCYCLE H, UPDATE L and CHECK H from the state sequencer. The special address status (SAS H) is asserted whenever the maintenance or LTC registers are addressed. The A<00> H input represents the status of address bit zero. The flush counter overflow status (FLOVFL H) input is asserted when the cache memory is being flushed. The LONGCYCLE H output is asserted each time a location is flushed and increments the address stored in the flush counter to the next location. The SEL<01:00> H provide the select output code used to drive the contents of a register selected in the cache data path onto the B-bus. The select codes are described in Table 4-5. The UPDATE L and CHECK H signals are used by the cache data path to control the tag parity function.

Table 4-5 Select Codes

SEL 1	SEL 0	Selections
0	0	The cache data path DAL outputs are tristated.
0	1	The contents of the address register is driven on the DAL outputs.
1	0	The status of the memory system error register is driven on the DAL outputs, except when the LTC register is specifically addressed.
1	1	The current address/or contents of the flush counter is driven on the DAL outputs.

4.3.8 Cache Memory

The cache memory asserts the COMP L input when an address scores a cache memory miss. The memory read/write functions are controlled by the TAG CS L, DATA CS B1-B0 L and the RAM WE L outputs. The tag chip select (TAG CS L) signal is asserted to select the 11-bit TAG memory. The high byte data chip select (DATA CSB1 H) and the low byte data chip select (DATA CSB0 H) signals are asserted to select words or bytes stored in the cache memory. The RAM write enable signal (RAMWE L) is asserted to write data, or negated to read data into the selected memory.

4.3.9 Floating-Point Accelerator

The floating-point accelerator (FPA) socket provides the FPA RDY H, FPA STL L, FPA OP L and FPA FPE L inputs and receives the FPA ACK L and DV L outputs. The FPA RDY H input is asserted when the FPA is ready to proceed. The FPA STL L input is asserted when the FPA wishes to stall the DCJ11. The FPA FPE L is asserted to exit the stall condition. The FPA OP L is asserted when the FPA is writing data on the A-bus. The state sequencer enables the FPA option by asserting the FPA ACK L output. The FPA latches data from the DCJ11 when the state sequencer asserts DV L.

4.3.10 Bus Traffic

The on-board buses transfer the addresses and the read/write data to and from the DCJ11. They also provide communications between the on-board functions and the system I/O. An overview of the bus traffic flow is described below.

4.3.10.1 Address Busing – The DCJ11 uses the B-bus to address cache memory, main memory, and the I/O devices. The address flow pattern is shown in Figure 4-12.

4.3.10.2 Read Data – The DCJ11 uses the A-bus to read data from the FPA, cache memory, maintenance register, main memory, and the I/O devices. The read pattern is shown in Figure 4-13.

4.3.10.3 Write Data – The DCJ11 uses the A-bus and B-bus to write data to the FPA, cache memory, status LEDs, main memory, and the I/O devices. The write data pattern is shown in Figure 4-14.

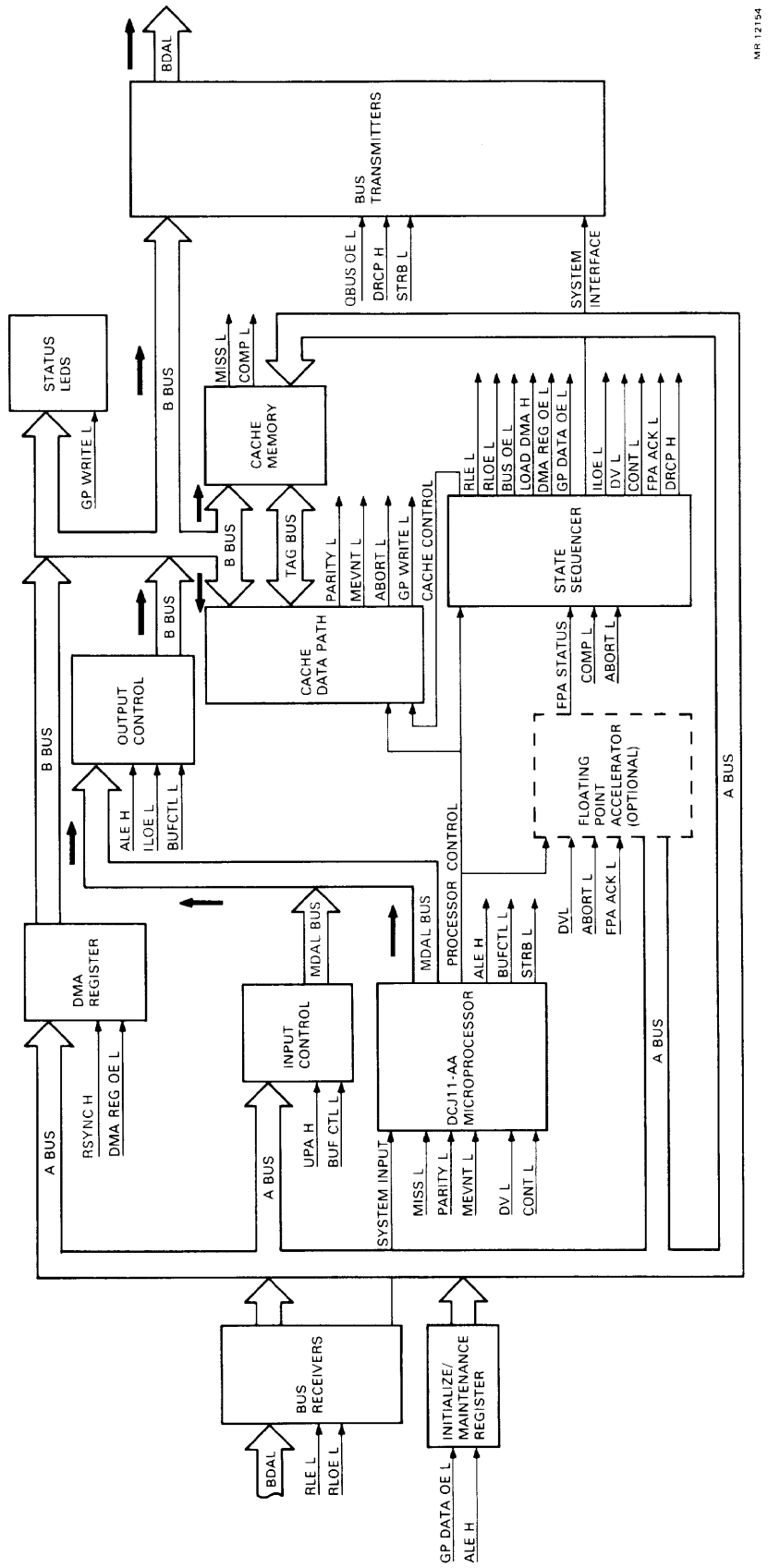


Figure 4-12 Address Traffic Pattern

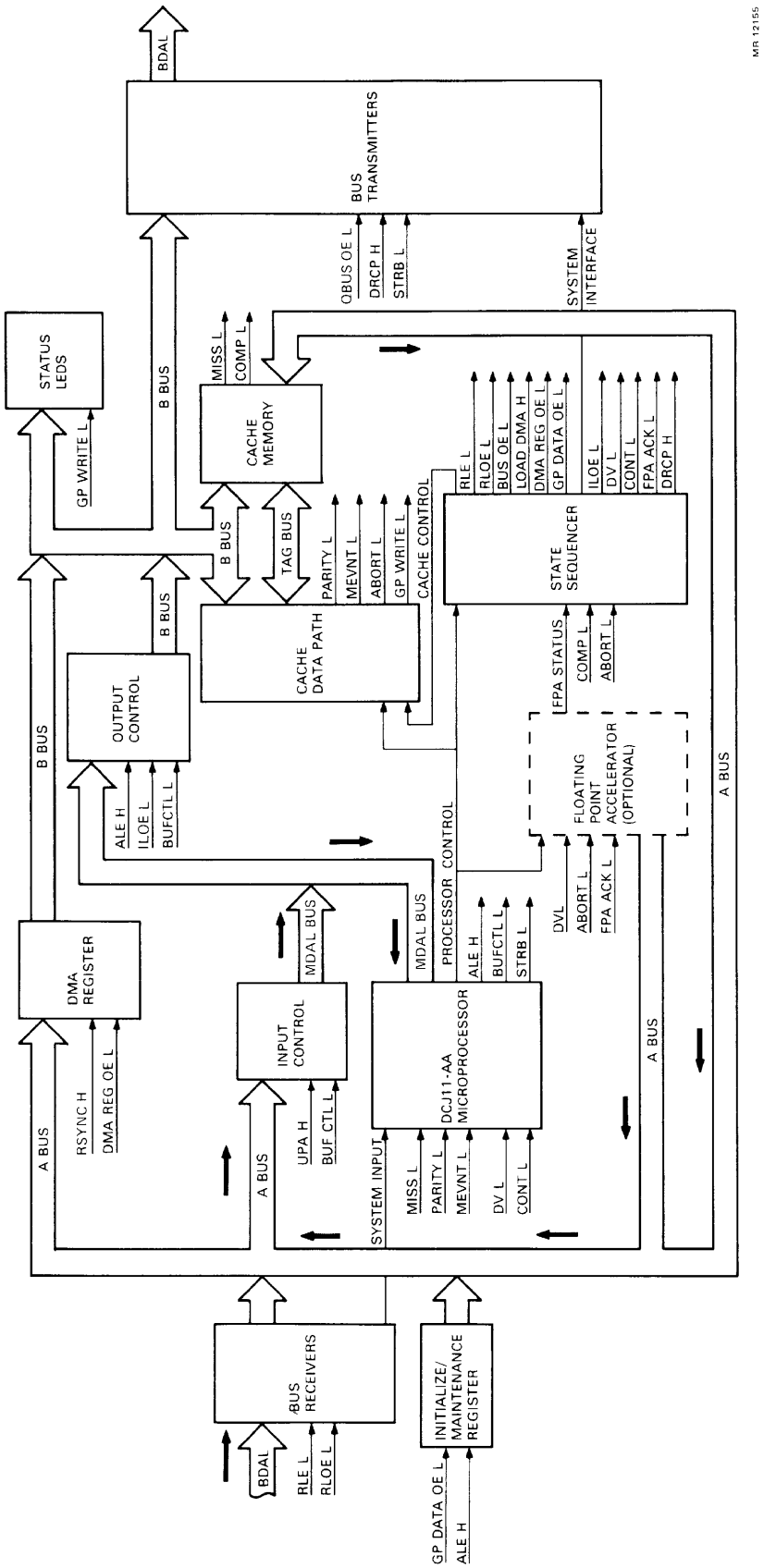


Figure 4-13 Read Data Busing

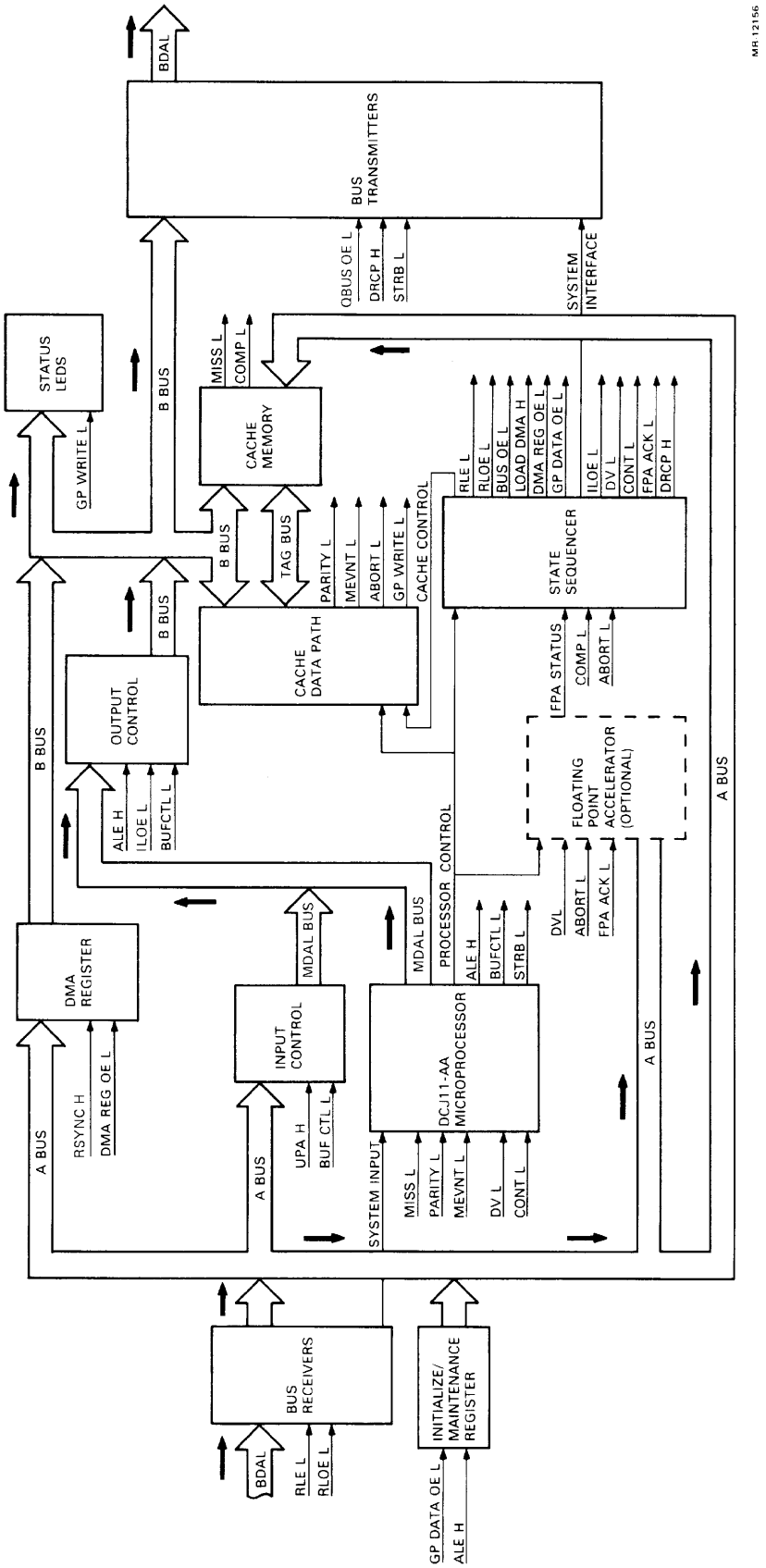


Figure 4-14 Write Data Busing

4.4 CACHE DATA PATH

The cache data path is a multifunction gate array (Figure 4-15) that controls the 8 Kbyte direct map cache memory. It generates B-bus bits <21:13> as TAG data for the cache memory during cache write transaction. Parity for the TAG data is generated, predicted, and checked by the gate array. The LTC, memory system error, and address registers are contained within the array. It also contains the flush address counter used to clear or flush the cache memory.

4.4.1 DCJ11 Input Signals

The cache data path decodes the AIO input to identify the transaction and the BS<01:00> H inputs to identify the type of address. The SEL<01:00> H inputs selects the contents of an internal register or counter as described in Table 4-6.

The cache data path receives the ALE L, STRB L and SCTL L signals to synchronize and control the cache operation. The assertion of ALE L latches the BS<01:00> H data and gates the GP WRITE L output. The assertion of STRB L latches the address data into the address register. The negation of STRB L clears the parity error latch and enables the GP WRITE L output. The assertion of SCTL L enables the ABORT L output and latches the write data. The negation of SCTL L clears the flush counter and disables the ABORT L output.

Table 4-6 Output Select Codes

SEL 1	SEL 0	Selections
0	0	The DAL output are tristated
0	1	The contents of the address register
1	0	Either memory system error or BEVNT register
1	1	Flush counter

4.4.2 State Sequencer Inputs

The cache data path receives CHECK H, UPDATE L and LONGCYCLE H signals to control the cache memory. The CHECK H and UPDATE L inputs control the generation, checking and prediction of the TAG parity as described in Table 4-7. The cache data path predicts the parity of address bits <21:13> in the same way it calculates the TAG parity bit. The predicted parity is driven as the PREDICT PAR H output signal and compared with the stored TAG parity bit by the data parity logic to determine a hit or miss. The TAG parity bit is calculated for bits <21:13> and stored with the TAG data. The parity is checked when the predicted parity and the stored parity bits are compared within the cache data path to enable the PERR L output when an error is detected. The LONGCYCLE H input is asserted to increment the address stored in the flush counter.

Table 4-7 TAG Parity

Update L	Check H	Function
Negated	Negated	Predict TAG parity
Negated	Asserted	Check TAG parity
Asserted	Negated	Generate TAG parity
Asserted	Asserted	Undefined

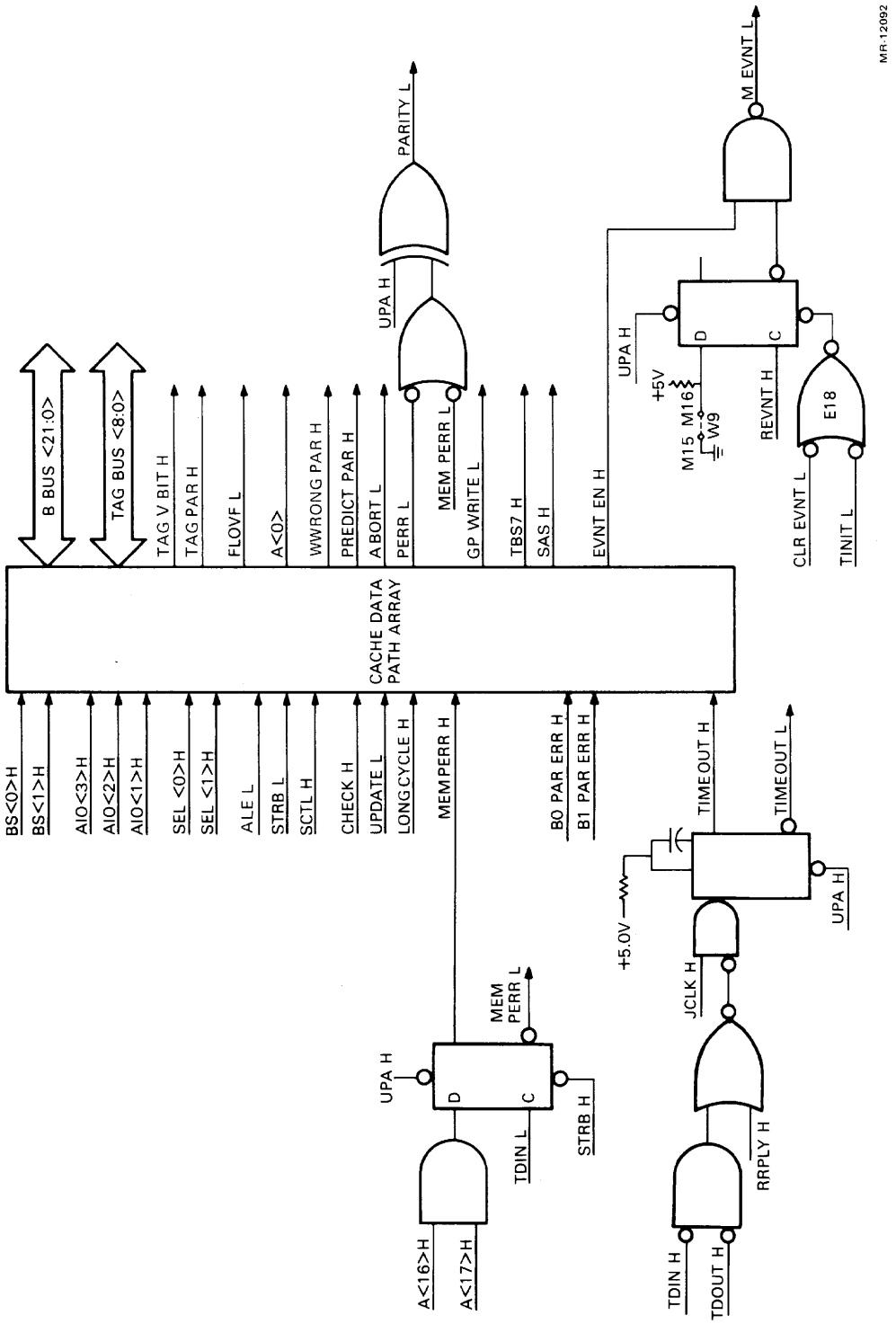


Figure 4-15 Cache Control Logic

4.4.3 System Memory Parity

The system memory parity data is transmitted to the module via A-bus bits <17,16>. These inputs are monitored and when asserted, a parity error is detected. The MEM PERR H input is asserted and enable either on ABORT L or PERR L output.

4.4.4 Cache Memory Parity

The cache memory parity error inputs B0 PAR ERR H and B1 PAR ERR H are asserted when a parity error is detected in the cache data memory. The low byte is monitored by B0 PAR ERR H and sets bit 06 of the MSER. The high byte is monitored by BI PAR ERR H and sets bit 07 of the MSER. Either input can enable the PERR L or ABORT L output.

4.4.5 Timeout

The TIMEOUT H input is enabled when the LSI-11 bus fails to assert the RRPLY H input within 10 microseconds after the TDIN H or TDOUT H signal was asserted by the module. When TIMEOUT is asserted, it causes the ABORT L output to be asserted and aborts the transaction.

4.4.6 Cache Control Register

The cache control register in the cache data path is shadow copied when the CCR register in the DCJ11 is written and its contents are used to control the cache memory system. The cache data path logic only interprets bits 10, 08, 07, 06, 01, and 00. The write wrong parity logic is enabled by bit 10 being set (1) and it inverts the current TAG parity bit. This will force a TAG parity error the next time that location is accessed. When bit 08 is set (1), the FLOVFL H output is asserted to flush the cache and the flush counter is enabled. The bit is reset when the flush counter overflows and SCTL L is negated. The parity error abort, bit 07, is used with the disable cache parity interrupt, bit 00 to determine the action taken in response to parity errors. The conditions for bits 07 and 00 are summarized in Table 4-8. The write wrong data parity logic is enabled when bit 06 is set (1) and it inverts both of the data parity bits. This changes the high byte even parity to odd and the low byte odd parity to even. This causes a data parity error the next time that location is accessed. The cache diagnostic mode is enabled when bit 01 is set (1) and the cache is allocated on all write transactions, regardless of ABORT L, except when bypassing or forcing a cache miss.

Table 4-8 Parity Error Action

Bit 7	Bit 0	Action
1	0	Abort through vector 114, update cache
1	1	Abort through vector 114, update cache
0	0	Interrupt through vector 114, update cache
0	1	Update cache only

4.4.7 Memory System Error Register

The memory system error register is a read-only register that uses bits 15, 07, 06, and 05 to store parity error data for the memory system. The register is cleared by any write into it. The parity abort, bit 15, is set whenever a parity abort occurs. A parity abort is defined as any parity error or memory error occurring during a demand read with the cache control register bit 07 set. When this occurs, bits 07, 06, and 05 are individually set to identify the type of parity error. Bit 07 is set for a high byte data parity error, bit 06 is set for a low byte parity error, and bit 05 is set for a tag parity error. However, if the cache control register bit 07 is not set, then any type of parity error in the cache sets all three bits. The register is read when the SEL <01, 00> bits are set to 1 and 0, respectively, and the LTC register address is not selected.

4.4.8 LTC Register

The LTC register is a read/write register that allows software to set bit 06 and enable the EVNT EN output. The EVNT EN H signal allows the bus BEVNT L input to be routed to the microprocessor as an external event interrupt. The BEVNT L input can be disabled by the user inserting the W9 jumper. When enabled, the flip-flop is clocked by REVNT H and the output is gated with EVNT EN H to enable the MEVNT L signal. The flip-flop is reset by either CLR EVNT L or TINIT L.

4.4.9 Flush Counter

The contents of the cache memory is flushed or cleared during power-up and whenever bit 08 of the cache control register is set. This requires each address location in the cache to be addressed and cleared. The process is initiated by the cache control chip asserting FLOVFL H to the state sequencer and zeroing the flush counter. The contents of the flush counter is used to address the cache memory via the B-bus bits <12:01>. Every time an address is cleared, the counter is incremented to the next address by the LONGCYCLE H input from the state sequencer. Flushing the cache memory takes up to 1.3 microseconds and during this time, no DMA or processor activity is performed. The counter contains 12 bits and when the cache memory is completely flushed, the counter overflows. This causes the cache control chip to negate the FLOVFL H signal to the state sequencer, indicating the cache flush operation is complete.

4.4.10 Address Register

The address register latches the address received via the B-bus during the early portion of the transaction. The A<00> output is driven directly from address bit 00. During the later portion of the transaction, the SEL <01, 00> H code enables the address to be driven via the B-bus to the main memory and the cache memory. All 22 bits are used to address the main memory and bits <12:01> are used to address the cache memory. Register bits <21:13> are placed on the TAG bus as data for storage in the cache memory when the UPDATE L input is asserted.

4.4.11 CDP Outputs

The cache data path transmits and receives address and data information via the B-bus <21:00> and the TAG bus <10:00> including the TAG V bit and TAG parity bit. The FLOVFL H output is asserted while the cache memory is being flushed and negated when flushing cycle is completed. The A<00> H output is asserted whenever the B-bus bit 00 is set (1). The WR WRONG PAR H output is asserted whenever bit 06 of the CCR is set and writes the wrong parity into the cache memory. The PREDICT PAR H output is the predicted TAG parity of B-bus bits <21:13> and it is compared with the stored TAG parity to determine the hit/miss results. The PERR L and ABORT L outputs are generated by the parity logic and interpreted by the DCJ11 as described in Table 4-9. The GP WRITE L output is asserted when the AIO coded input specifies a GP write transaction. The output is used to externally latch the GP data. The TBS7 H output is asserted when the BS <01, 00> H code specifies an external I/O address during the early portion of the transaction and during the later portion of the transaction, or if the transaction is bypassing the cache or forcing a cache miss. The SAS H output is asserted whenever the maintenance register or the LTC register is being addressed. The EVNT EN H output is described in Paragraph 4.4.8.

Table 4-9 Abort and Parity Response

Abort	Parity	DCJ11 Action
Negated	Negated	No interrupt or abort
Negated	Asserted	Interrupt; vector to location 114
Asserted	Asserted	Abort; vector to location 114
Asserted	Negated	Abort; vector to location 4

4.5 CACHE MEMORY

The cache memory (Figure 4-16) consists of RAM memory for data, TAG and parity, the data parity logic, and the hit/miss logic. The cache memory is used to temporarily store data received from the system memory that the processor is currently using. This allows the DCJ11 to quickly access on-board data without performing external bus transactions. The physical address is divided into three sections as shown in Figure 4-17. The byte bit is used to access either high or low bytes of data. The index bits are used as the address of the cache memory. The label bits are stored as TAG data for valid cache entries. Each cache entry is organized as shown in Figure 4-18. The high and low bytes of data are stored as data. The low byte parity (P0) is stored as even parity and the high byte parity (P1) is stored as odd parity. The label bits with a tag valid bit (V) and the tag parity bit (P2), stored as even parity are stored as TAG data. The byte parity is calculated by the data parity logic and the hit/miss logic interprets the physical address as a valid cache address.

The cache memory is controlled by the state sequencer signals DATA CS BO, BIL, TAG CS L, UPDATE L, and the write enable signal RAM WE L. The WR WRONG PAR H, PREDICT PAR H signals and the TAG data are controlled by the cache data path chip. The physical addresses are received via the B-bus, the data is read/written via the A-bus and the TAG data is read/written via the TAG bus.

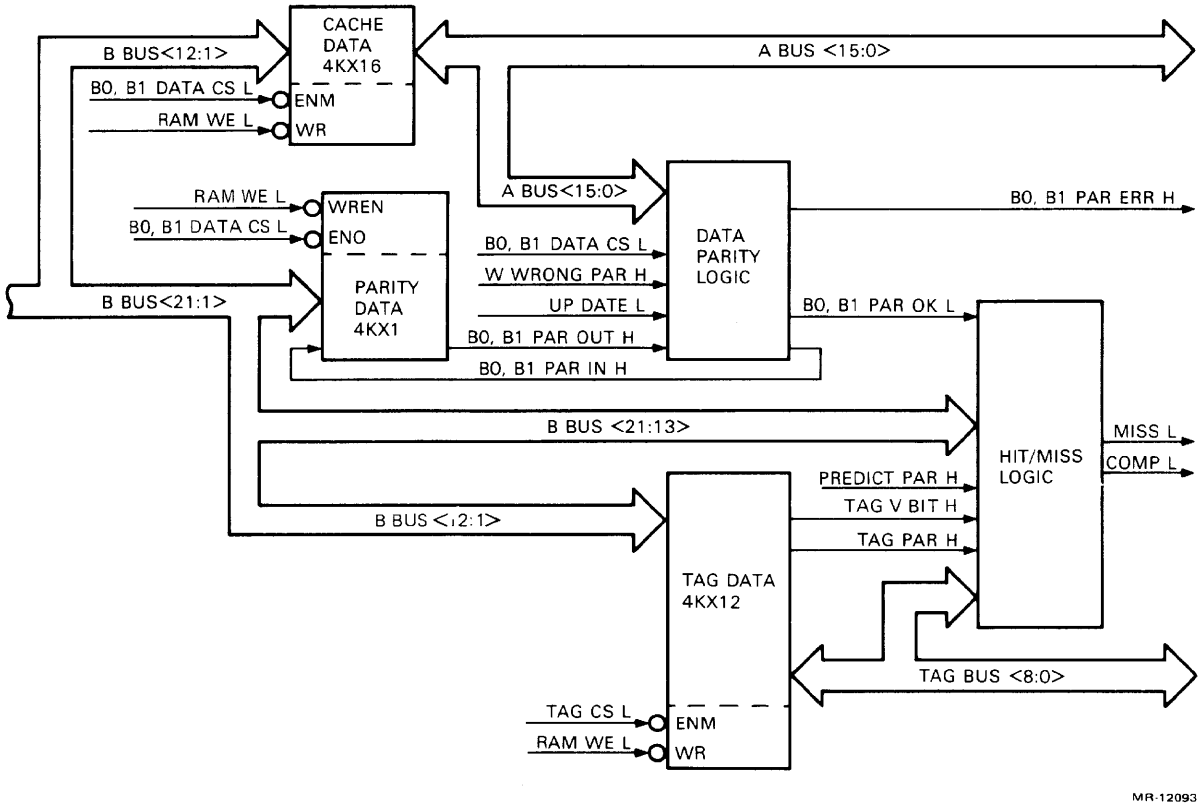


Figure 4-16 Cache Memory

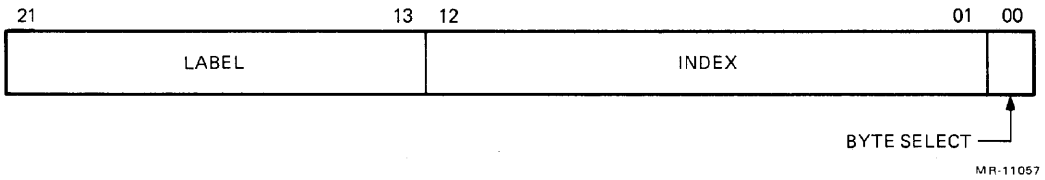


Figure 4-17 Cache Memory Physical Address

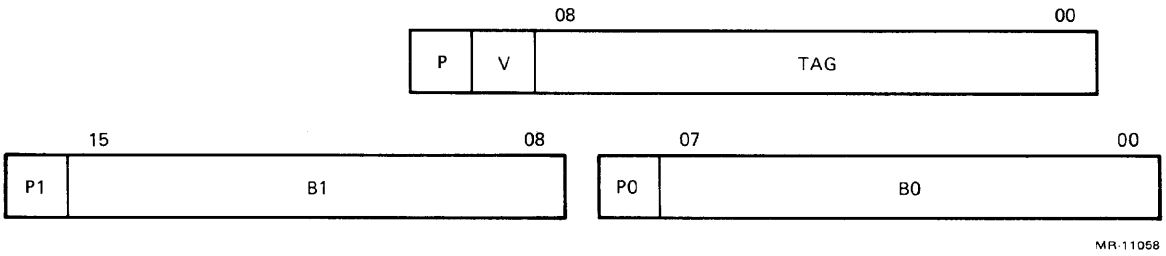


Figure 4-18 Cache Data

4.5.1 Cache Data

The cache data RAM is 8 Kbytes of read/write memory that is addressed by the index field, B-bus bits <12:01>. These bits will always access the data stored in an address location, but the data is not validated until the label field of the address is verified as the TAG data.

The read/write operations are controlled by the state sequencer. The low byte of cache data is read when the DATA CS BO L input is asserted and is written when both the DATA CS BO L and RAM WE L inputs are asserted. The high byte of cache data is read when the DATA CS B1 L input is asserted and is written when both the DATA CS BI L and RAM WE L inputs are asserted. The data is routed via the A-bus to the DCJ11.

4.5.2 Data Parity Logic

The data parity logic generates parity bits for the high and low bytes of data. The same logic is used to check the parity bits when data is read from the cache memory. The high byte stores odd parity and the low byte stores even parity. The parity logic is shown in Figure 4-19.

The parity logic uses the selected byte data and the UPDATE L signal from the state sequencer to generate data parity. The UPDATE L input enables the parity generator. The parity generator determines the number of high inputs and generates a parity bit for the high and low bytes. The low byte stores the status of the parity bit as B0 PAR IN H, and the high byte stores the status of the parity bit as B1 PAR IN H when the data is written into the cache memory. The cache data path can invalidate the data entry by enabling the WR WRONG PAR H input. This signal uses the exclusive-OR gate to invert the generated parity bit and store the error in the parity RAM.

The parity bit of the data is checked when the cache memory is accessed. The data is received by the parity generator and the UPDATE L input is not asserted at this time. The parity data is accessed, the low byte parity bit is received as B0 PAR OUT H, and the high byte parity bit is received as B1 PAR OUT H. The NAND gate is enabled and functions as an inverter for the B0, B1 PAR OUT H signals. The DATA CS B0, B1 L inputs, check the even output for the low byte (B0) and the odd output for the high byte (B1) to set the PAR OK L outputs low.

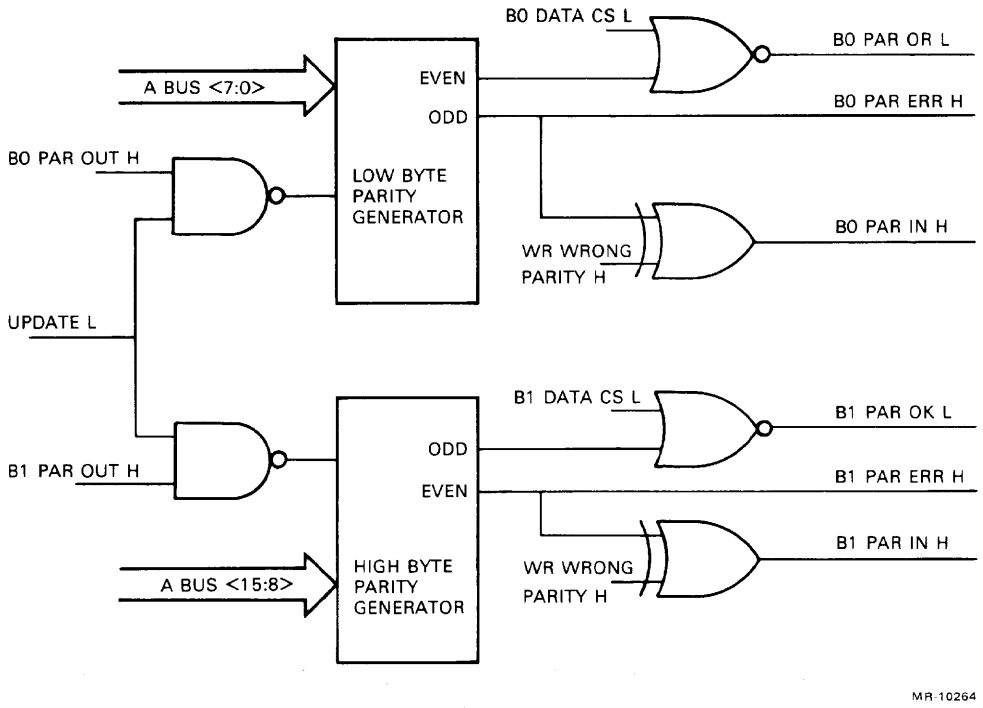


Figure 4-19 Cache Data Parity Logic

4.5.3 Parity Data

The parity RAM has 8 Kbytes of read/write RAM memory that stores the high and low byte data parity bit. The low byte parity bit is read when DATA CS BO L input is asserted and is written when both the DATA CS BO L and RAM WE L are asserted. The high byte parity bit is read when DATA CS BI L input is asserted and is written when both the DATA CS BI L and RAM WE L are asserted. The data parity bits are generated and used by the data parity logic.

4.5.4 TAG RAM

The TAG RAM is a 4 K × 12 read/write memory that stores 11 bits of data and one bit that is not used. The data consists of the 9-bit label field (address bits <21:13>), the TAG valid bit (VBIT), and the TAG parity bit (TAG PAR). The data is received from the cache data path. The data is read when TAG CS input is asserted and is written when both TAG CS and RAM WE inputs are asserted. These signals are controlled by the state sequencer.

4.5.5 Hit/Miss Logic

The hit/miss logic (Figure 4-20) compares the TAG stored data and bits <21:13> of the current address on the B-bus for a match condition. The TAG valid bit is also checked. When a match occurs, the current address is recognized as a valid cache entry and sets the comparator outputs low. If they do not match, the comparator outputs are set high. The TAG PAR H bit is checked with the PREDICT PAR H bit by the exclusive-OR gate and the output is low when a match occurs. The MISS L and COMP L gates are identical and monitor the two comparator outputs, the two data PAR OK L bits, and the output of the TAG PAR H gate. When all five inputs are low, the MISS L and COMP L outputs are high to indicate a hit. The MISS L signal goes to the DCJ11 and the COMP L signal goes to the state sequencer to indicate that the current address is stored in the cache memory. If MISS L and COMP L outputs are low, indicating one of the inputs is invalid, then the current address is not a valid cache entry and the data is retrieved from the system memory.

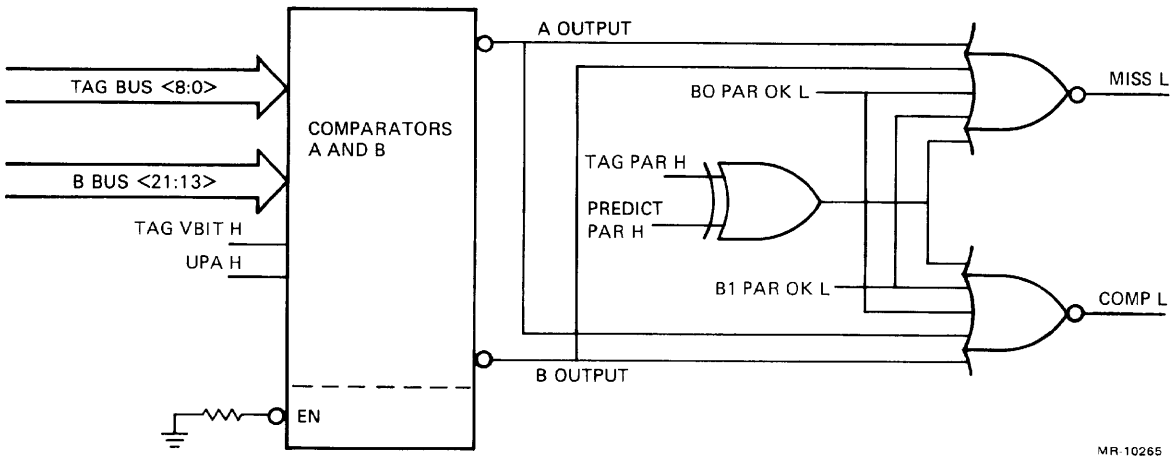


Figure 4-20 Cache HIT/MISS Logic

4.6 BUS RECEIVERS

The module receives addresses and data from the LSI-11 bus via six 2908 bus transceivers as shown in Figure 4-21. The state sequencer provides the control signals RLE L and RLOE L that transfer LSI-11 bus data to the module A-bus. The data is latched when RLE L is asserted. The output drivers are then enabled by RLOE L and transmits the LSI bus data to the module A-bus.

The LSI-11 bus control signals are transmitted to the module by the input transceivers. These signals are used by the module to control the LSI-11 bus interface.

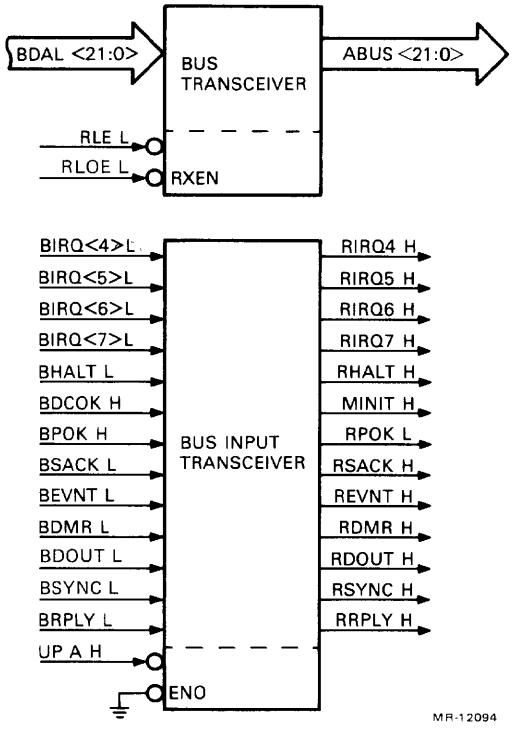


Figure 4-21 KDJ11-A Bus Receivers

4.7 BUS TRANSMITTERS

The module transmits addresses and data to the LSI-11 bus via six 2908 bus transceivers as shown in Figure 4-22. The address and data inputs are controlled by the LATCH H input. The address is clocked into the transceiver when the STRB L input from the DCJ11 is asserted. Write data is checked into the transceiver when DRCP L (normally low) is pulsed from high to low. The DRCP L input is generated by the state sequencer. The state sequencer enables the QBUS OE L input to transmit the data over the LSI-11 bus. When TBS7 H (Bank Select) signal is asserted to indicate the reference is to the I/O page, bits <19:16> are driven as zeros. This allows the KDJ11-A module to work in a 64 Kbyte system with the older MSV11-D memories.

The LSI bus control signals are transmitted by the output transceivers. The state sequencer provides most of the handshake protocol with the LSI bus. The WAKEUP H signal is enabled by removing the W9 jumper to generate the BDCOK H initialization pulse at power-up.

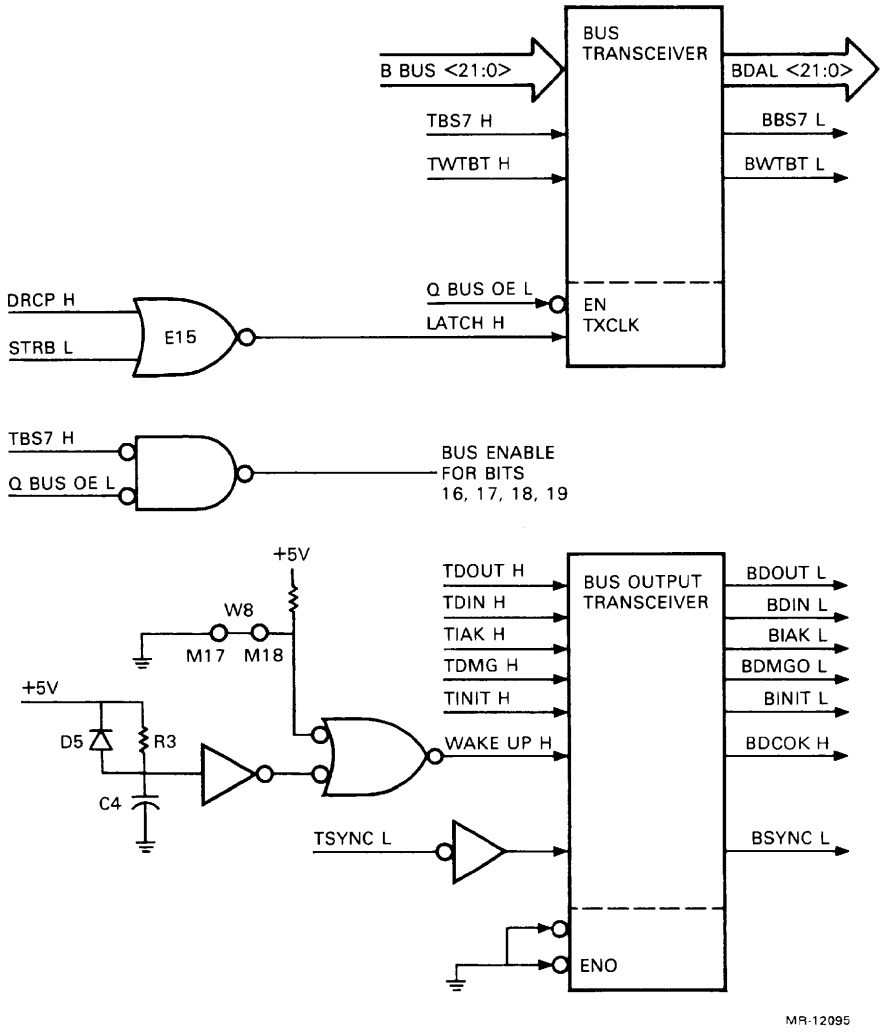


Figure 4-22 KDJ11-A Bus Transmitters

4.8 OUTPUT CONTROL

The output control logic (Figure 4-23) has 22 D-type latch circuits with output drivers that transfer the address or data on the MDAL bus to the B-bus. The ILOE L signal from the state sequencer enables the drivers to the B-bus. A decoder circuit uses the DCJ11 outputs, BUFCTL L and ALE L, to control the latches. When BUFCTL L and ALE L are negated, the output latches are opened. When either ALE L or BUFCTL L are asserted, the latches are closed.

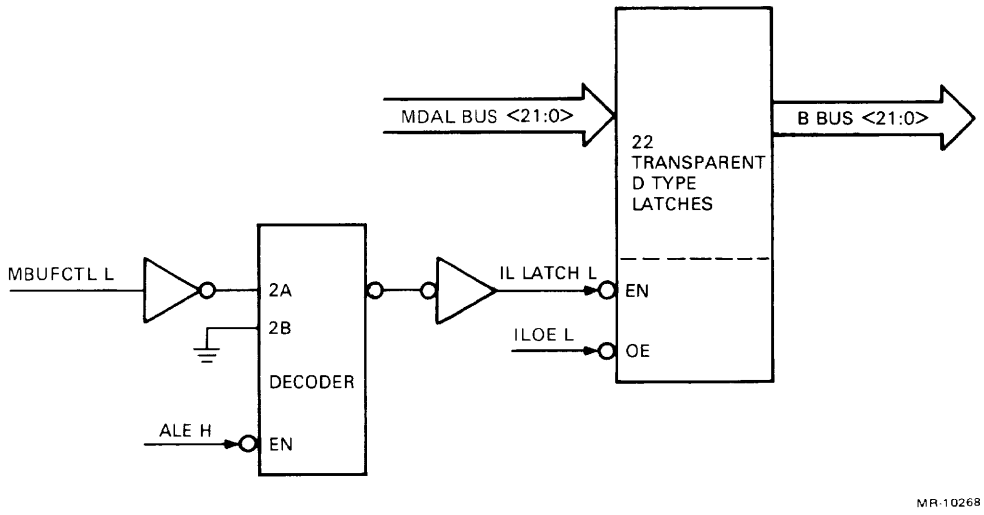


Figure 4-23 DCJ11-A Output Control

4.9 INPUT CONTROL

The input control logic (Figure 4-24) uses 16 D-type latch circuits to transfer data from the A-bus to the MDAL bus. The latches are used as buffers (latches are always opened) and are enabled when the BUFCTL L input is asserted.

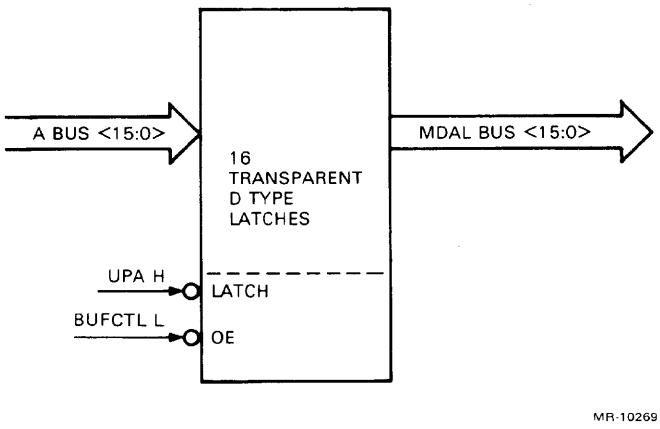


Figure 4-24 DCJ11-A Input Control

4.10 DMA MONITOR REGISTER

The KDJ11-A does not perform direct DMA transfers, but it does monitor DMA transfers when the system memory is being updated via block DMA. This ensures that the data stored in the cache memory is not being changed in the system memory. During a DMA transfer, the initial address of the DMA transaction is transferred over the A-bus. It is clocked into the DMA monitor register when RSYNC H is asserted. For DMA, DATO, DATIO and DATOB bus cycles, this register is used to address the cache memory in order to determine if the referenced location is in the cache memory. If it is, the cache data is invalidated. Successive block mode DMA write cycles (DATOB) are also monitored. Address bits <04:01> of the initial DMA address are clocked into the DMA monitor register when RSYNC H is asserted. These bits are incremented to the next address when RDOUT H is negated. Therefore, an entire 16-word aligned block mode transfer can be monitored. The four-bit incrementor with bits 00 and 05 are designed into the FPLA shown in Figure 4-25. The remaining 16 bits are controlled by the D-type flip-flops. The DMA REG OE L signal is controlled by the state sequencer and the INC/LOAD DMA ADR H input is controlled by the DMA LSI-11 bus signals BSYNC L and BDOUT L.

4.11 INITIALIZATION/MAINTENANCE REGISTER

The initialization/maintenance register allows the user to select the options available as described in Chapter 2. This register (Figure 4-26) is read by the DCJ11 during the power-up sequence and can be read by software accessing location 17 777 750 to determine which options were selected. The register uses jumpers W1 to W7 to determine the input state. The W3, W5, and W7 jumpers read as "1" when the jumper is removed; W1, W2, W4 and W6 jumpers read as "1" when the jumper is inserted. The UPA input is pulled up to +5 Vdc representing a "1" for bit 04 and a "0" for bits <11:09>. The grounded inputs represent a "0" for bits <07:05>. The FPA OP L input will be a "1" if a FPA is mounted on the module and the PWR OK H input is a "1" when the LSI-11 bus signal is asserted. The BDCOK H signal indicates the ac power is set to its proper value.

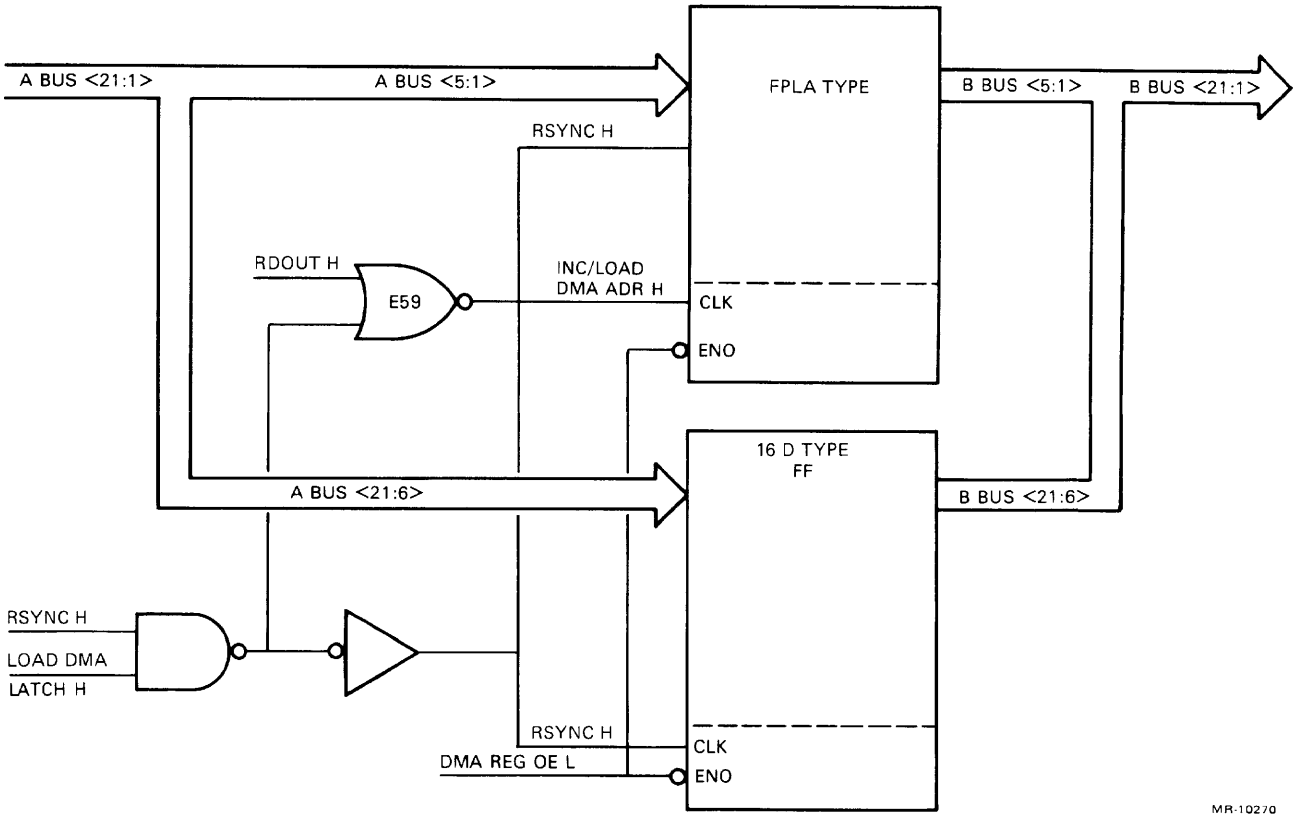


Figure 4-25 DMA Monitor Register

The low byte of the register is implemented by using eight D-type latches. The data is clocked by the assertion of ALE L from the DCJ11. The high byte of the register is implemented by using eight buffer drivers. The entire register is read onto the A-bus by GP DATA OE L input from the state sequencer.

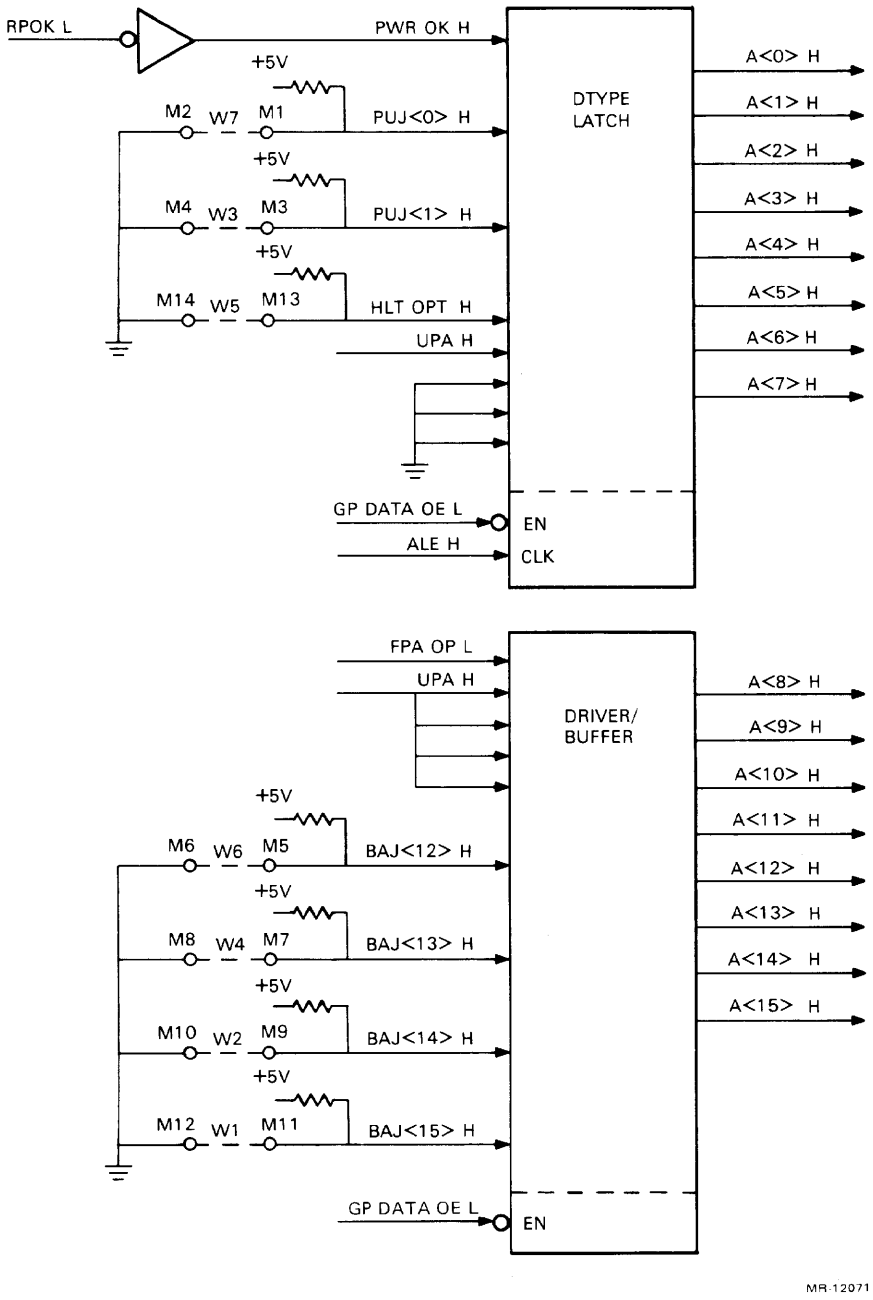


Figure 4-26 Initialization/Maintenance Register Logic

4.12 STATUS LEDs

The status LEDs logic (Figure 4-27) uses an addressable latch circuit for the LED display and a decoder circuit to reset either EVENT or PWR FAIL. The DCJ11 controls these functions by performing GP writes on the B-bus.

The EVENT or PWR FAIL conditions are cleared by GP write codes 100 and 140. The decoder circuit decodes B-bus bits 05 and 06 and is enabled by the GP WRITE L signal from the cache data path. When both bits are set, the CLR PWR FAIL L output is enabled and when bit 06 is set and bit 05 negated, the CLR EVENT L output is enabled.

The status LEDs are controlled by an addressable latch circuit. The circuit is reset by the MINIT L signal generated at power-up. MINIT L latches all the outputs low, thereby turning on the three diagnostic LEDs and turning off the ODT LED. It also enables the TINIT L output to initialize the module. During the initialization period the DCJ11 performs diagnostics, and upon the successful completion, it issues GP write codes to turn off the LEDs. GP code 220 turns off the SLU LED, GP code 224 turns off the MEMORY OK LED and GP code 230 turns off the SEQUENCING LED. After the initialization period, the DCJ11 enters its start up mode. If it enters ODT then GP write code 234 is issued and turns on the ODT LED. The LED functions are described in Chapter 2.

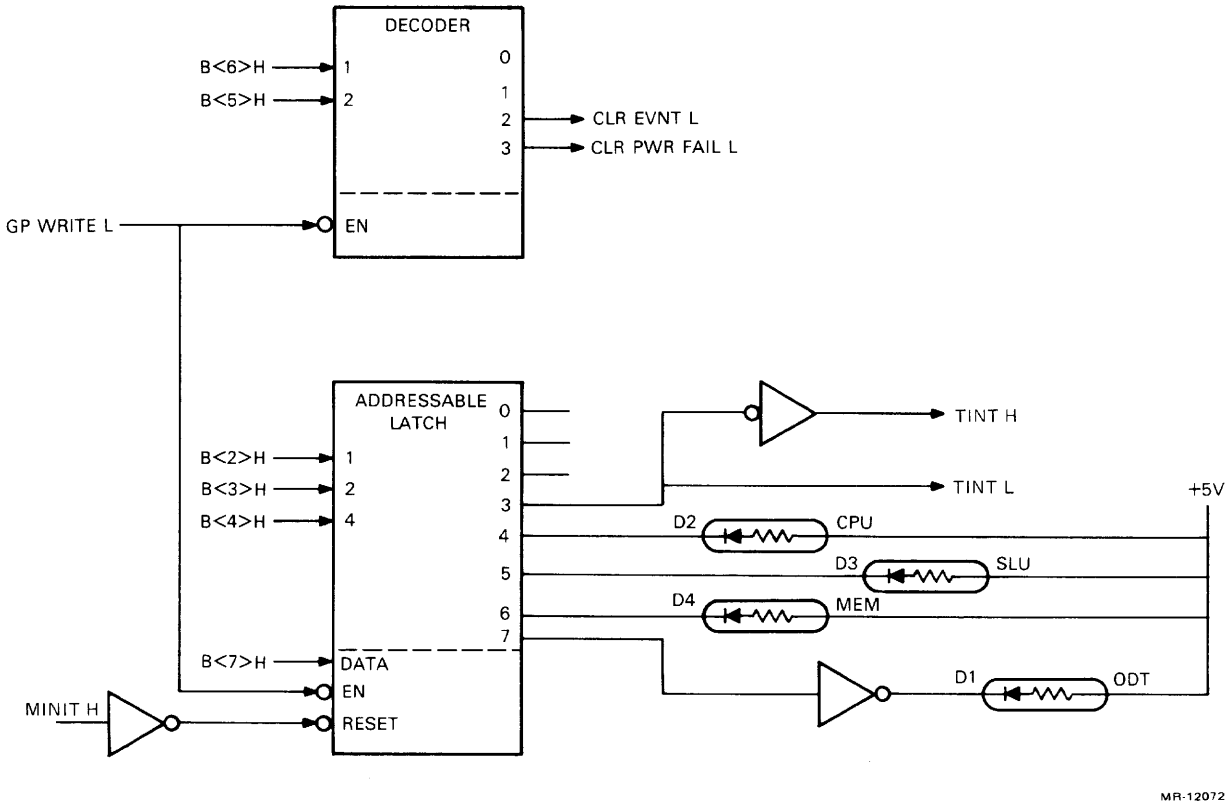


Figure 4-27 Status LEDs Logic

CHAPTER 5
EXTENDED LSI-11 BUS

5.1 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 256 Kbyte 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 5-1. Refer to Chapter 2 for a list of the extended LSI-11 bus signals.

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 5-1 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 5.4 and 5.5.1.

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 KDJ11-A 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., it "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 KDJ11-A 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.

5.2 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 the 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.

5.3 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 5-2.

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 5-3 describes the bus signals used in a data transfer operation.

Table 5-2 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

Table 5-3 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	

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.

5.3.1 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 5.3.1.2 through 5.3.1.4 describe the data transfer portion of the various bus cycles.

5.3.1.1 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. It asserts TDAL<21:00> with the desired slave device address bits.
2. It asserts TBS7 if a device in the I/O page is being addressed.
3. It asserts TWTBT if the cycle is a DATO(B) bus cycle.
4. It 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 8 Kbyte 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.

5.3.1.2 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 5-1. The DATI bus cycle timing is shown in Figure 5-2. 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.

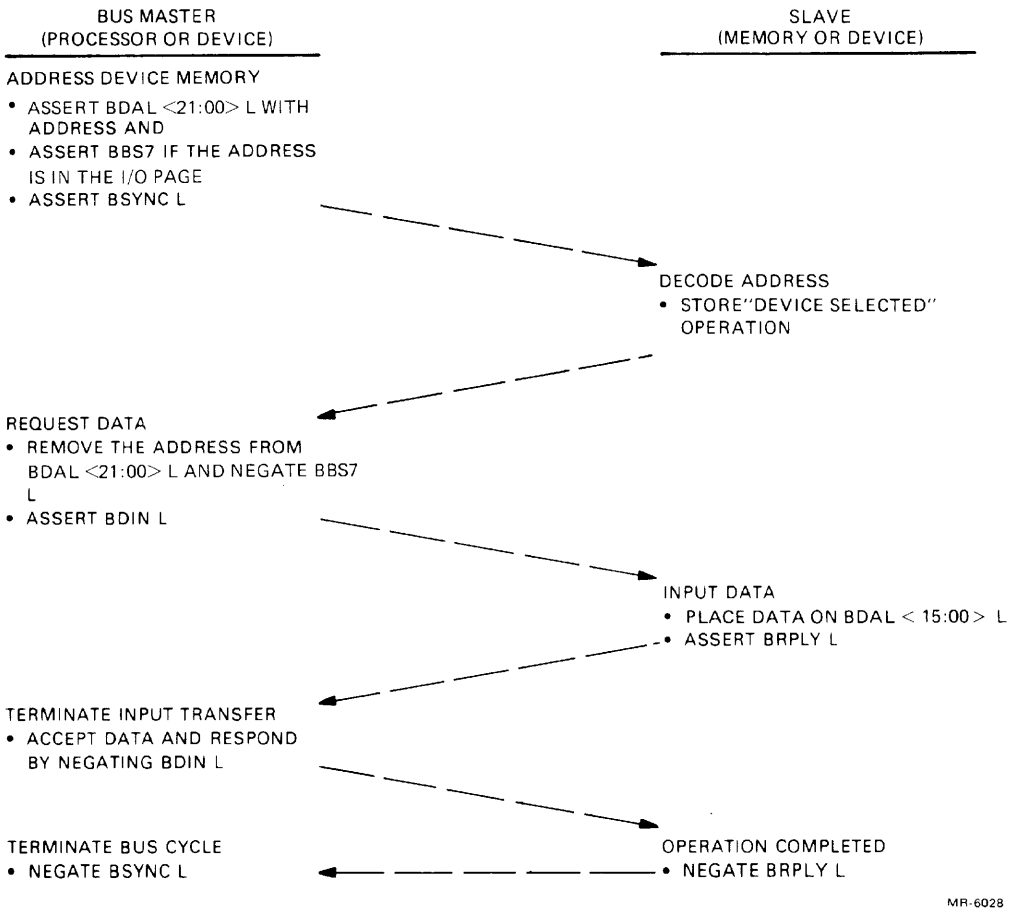
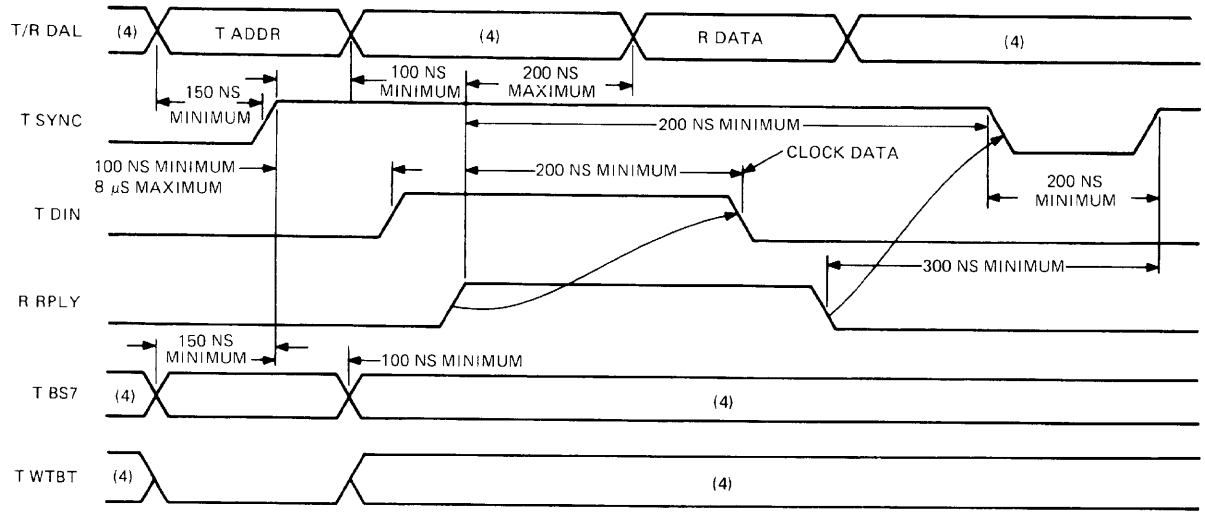
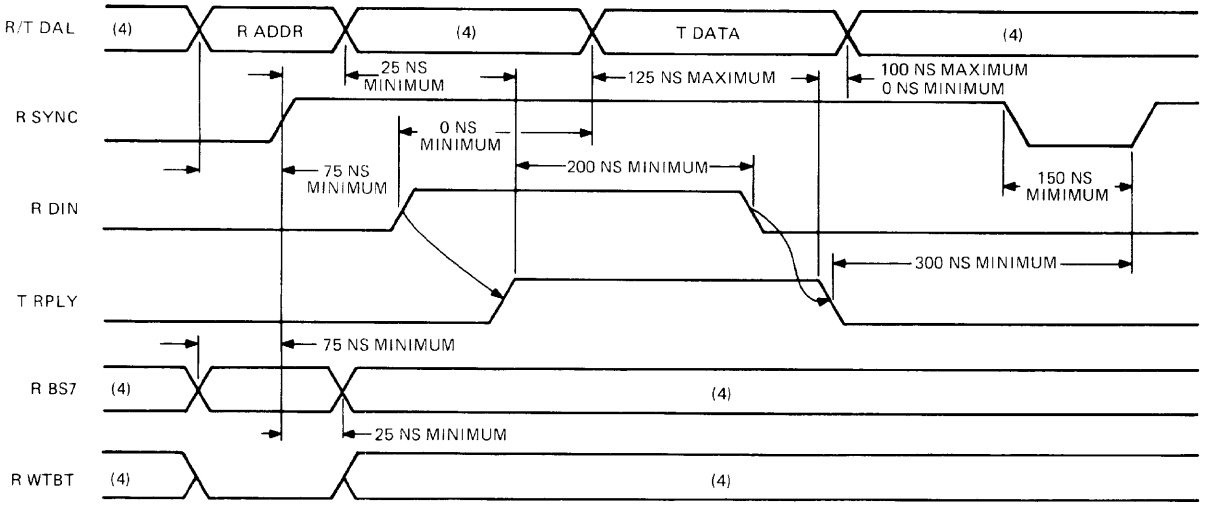


Figure 5-1 DATI Bus Cycle



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-6037

Figure 5-2 DATI Bus Cycle Timing

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

1. It 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. It 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.

5.3.1.3 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 5-3. The DATO(B) bus cycle timing is shown in Figure 5-4.

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.

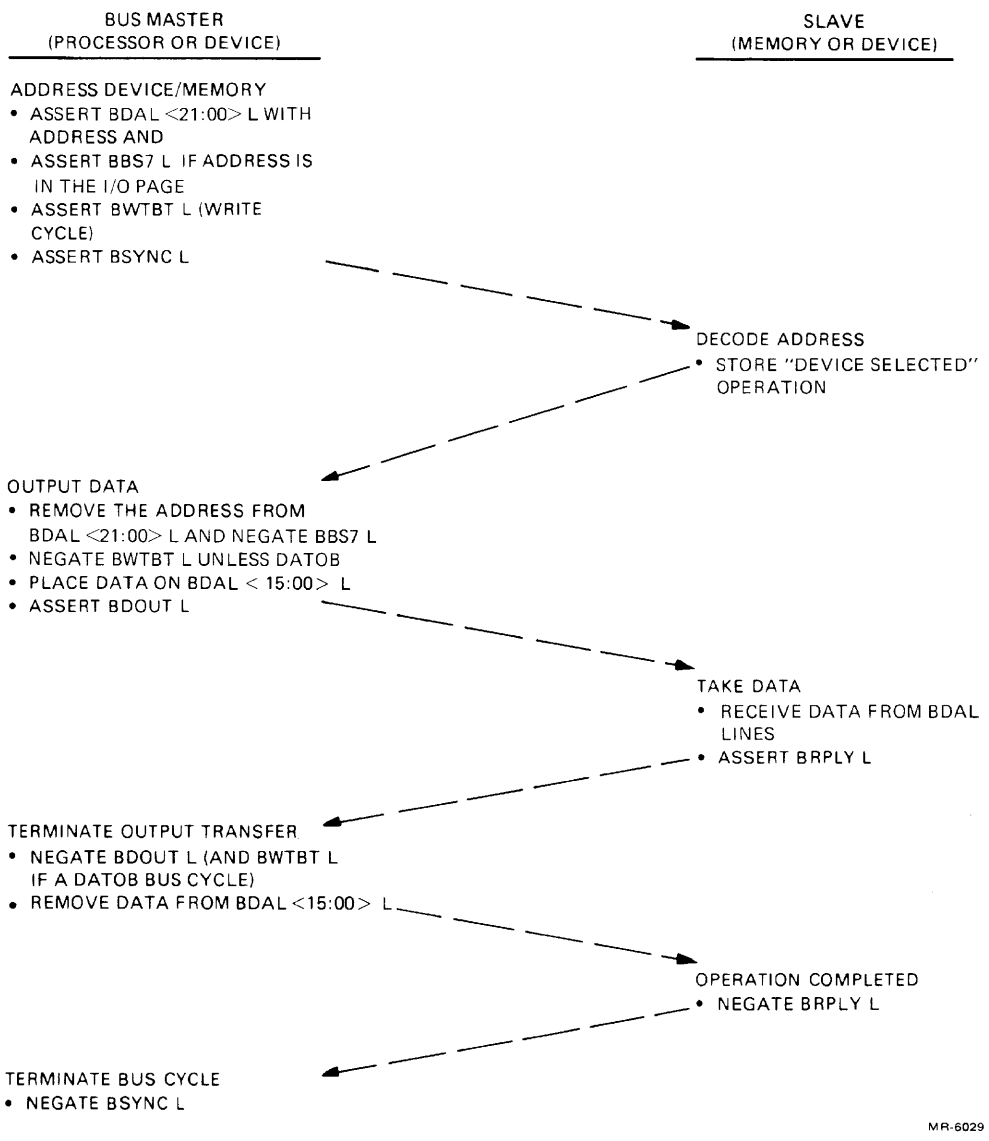
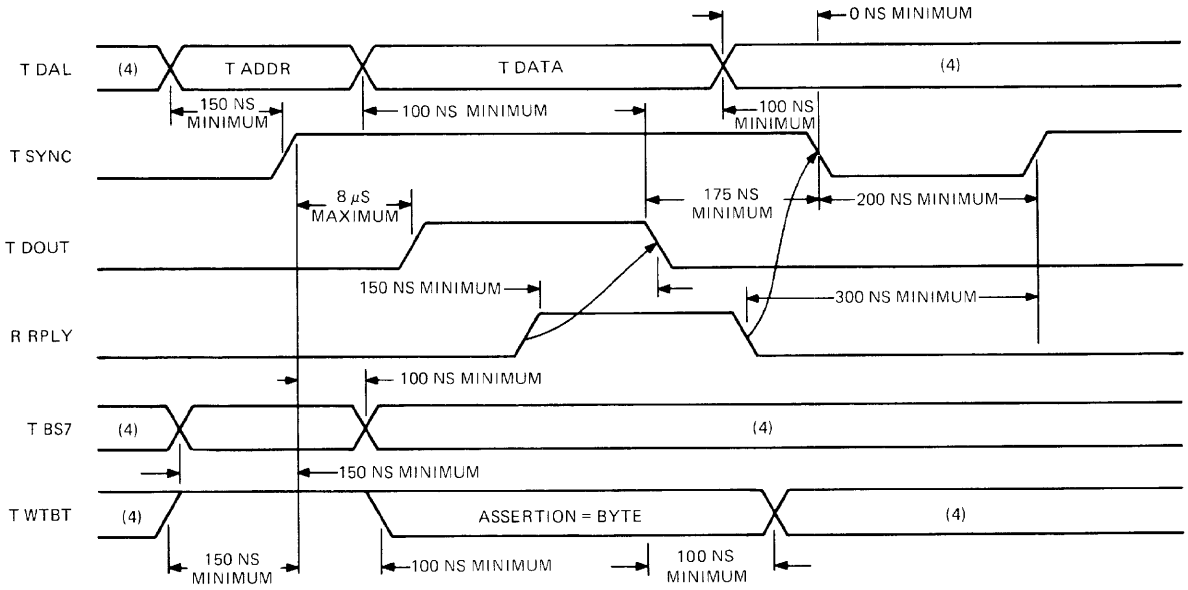
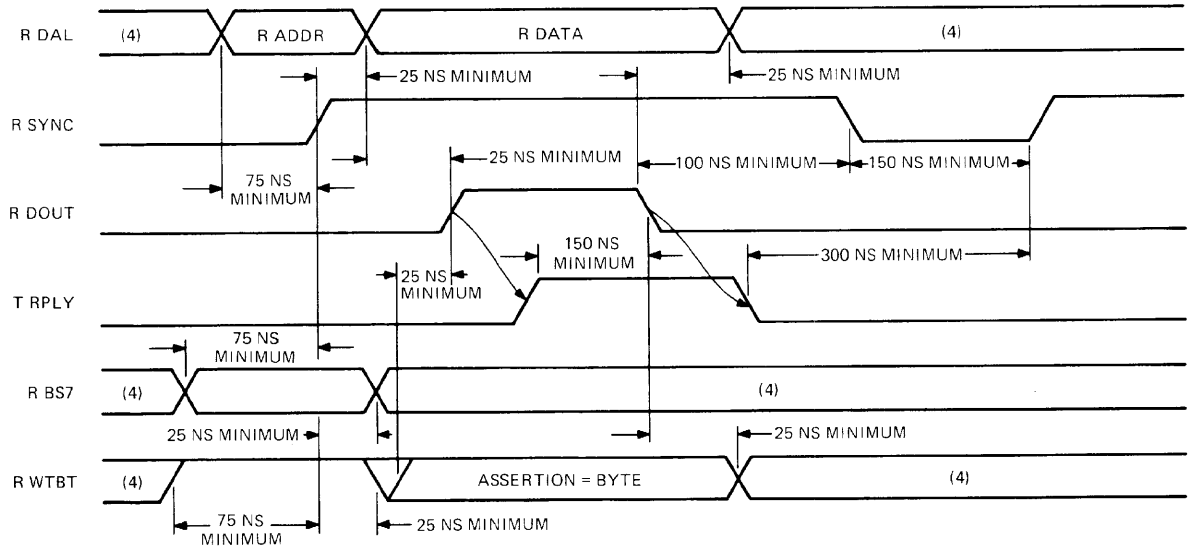


Figure 5-3 DATO or DATO(B) Bus Cycle



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 5-4 DATO or DATO(B) Bus Cycle Timing

5.3.1.4 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 5.3.1.2; 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 5-5. The DATIO and DATIO(B) bus cycle timing is shown in Figure 5-6.

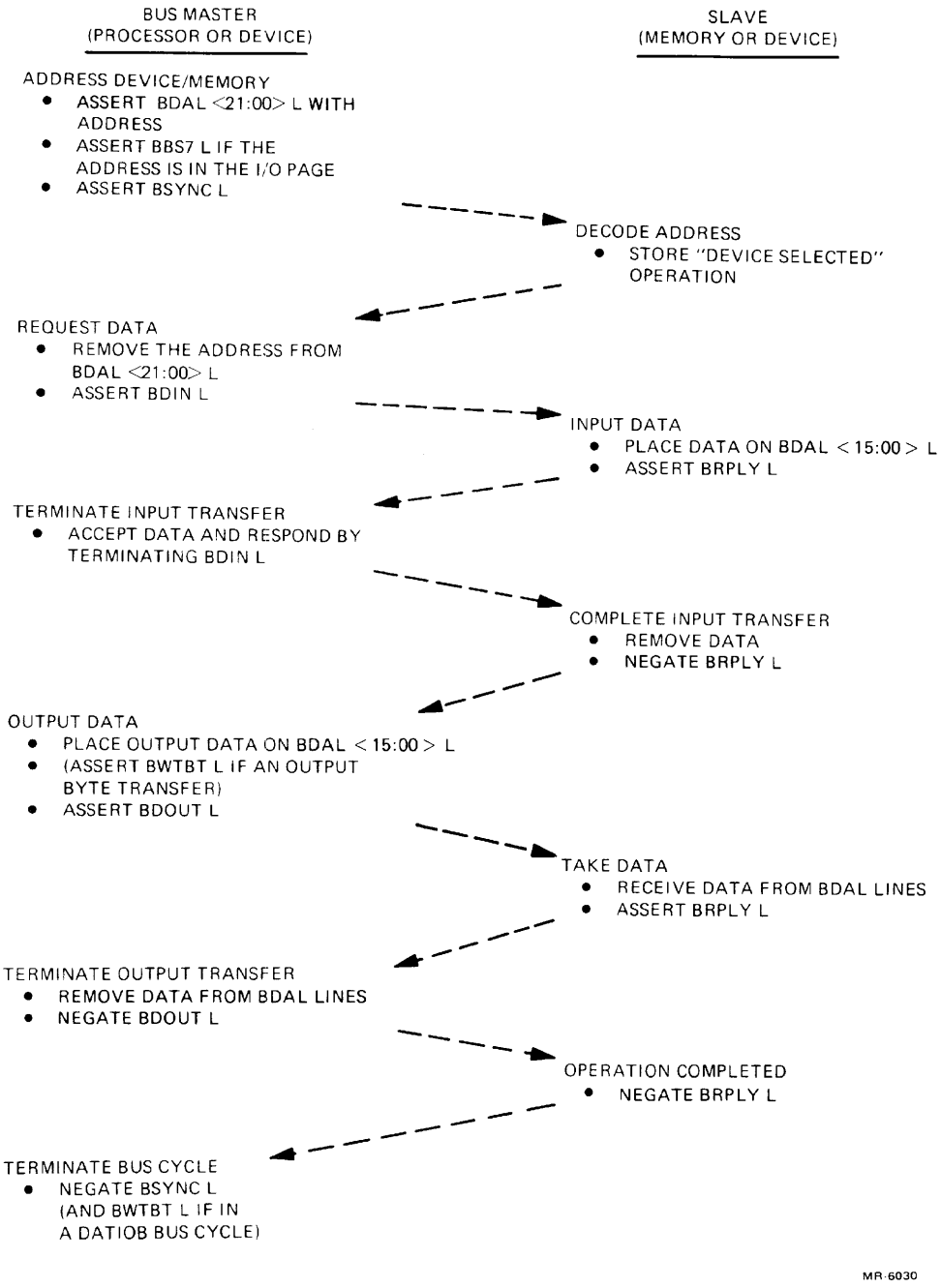
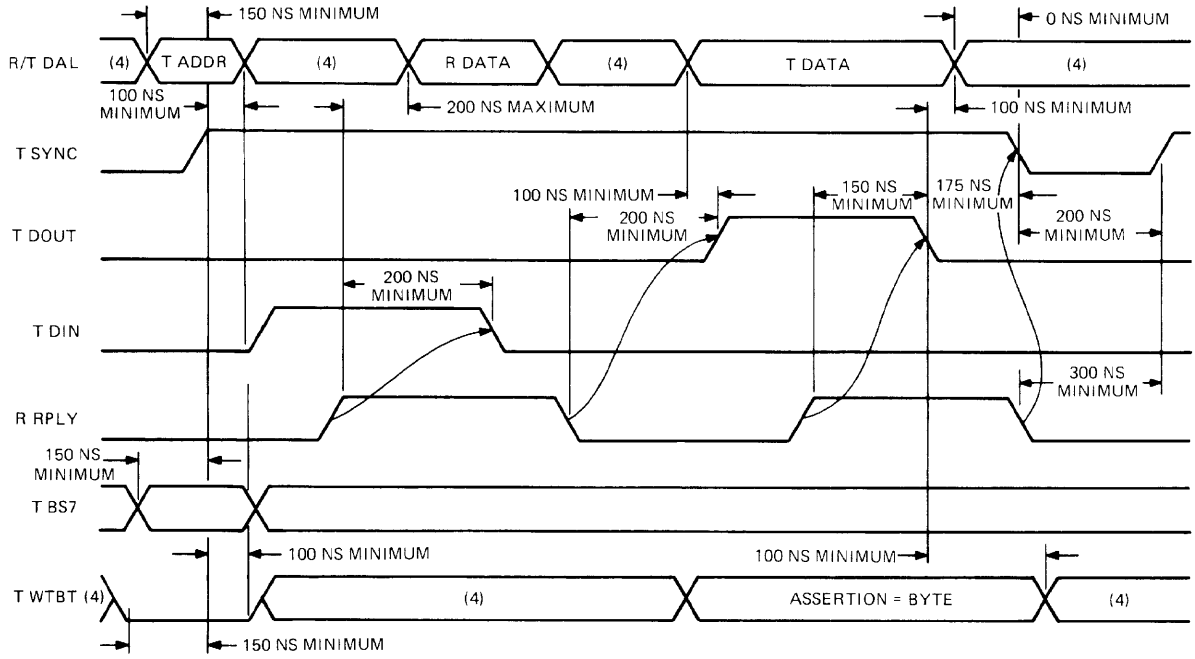
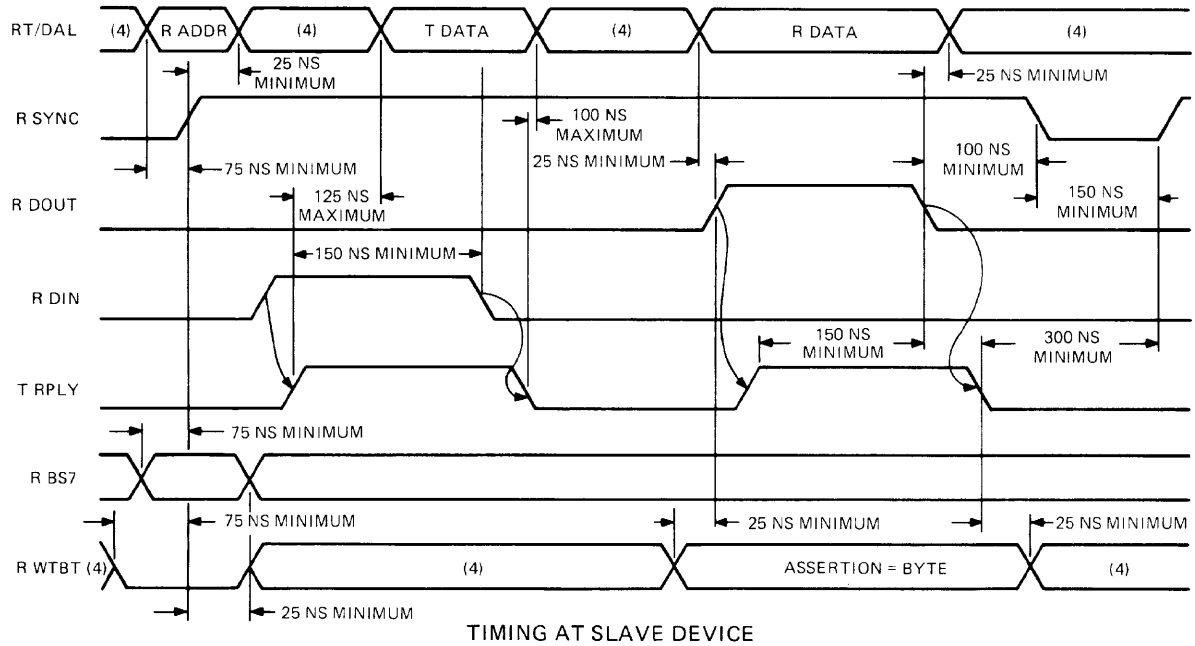


Figure 5-5 DATIO or DATIO(B) Bus Cycle



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 5-6 DATIO or DATIO(B) Bus Cycle Timing

5.4 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.

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

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 5-7. The DMA request/grant bus cycle timing is shown in Figure 5-8.

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 TDMG. 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 KDJ11-A asserts TDMG 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 KDJ11-A 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 5.3.1.2 through 5.3.1.4.

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.

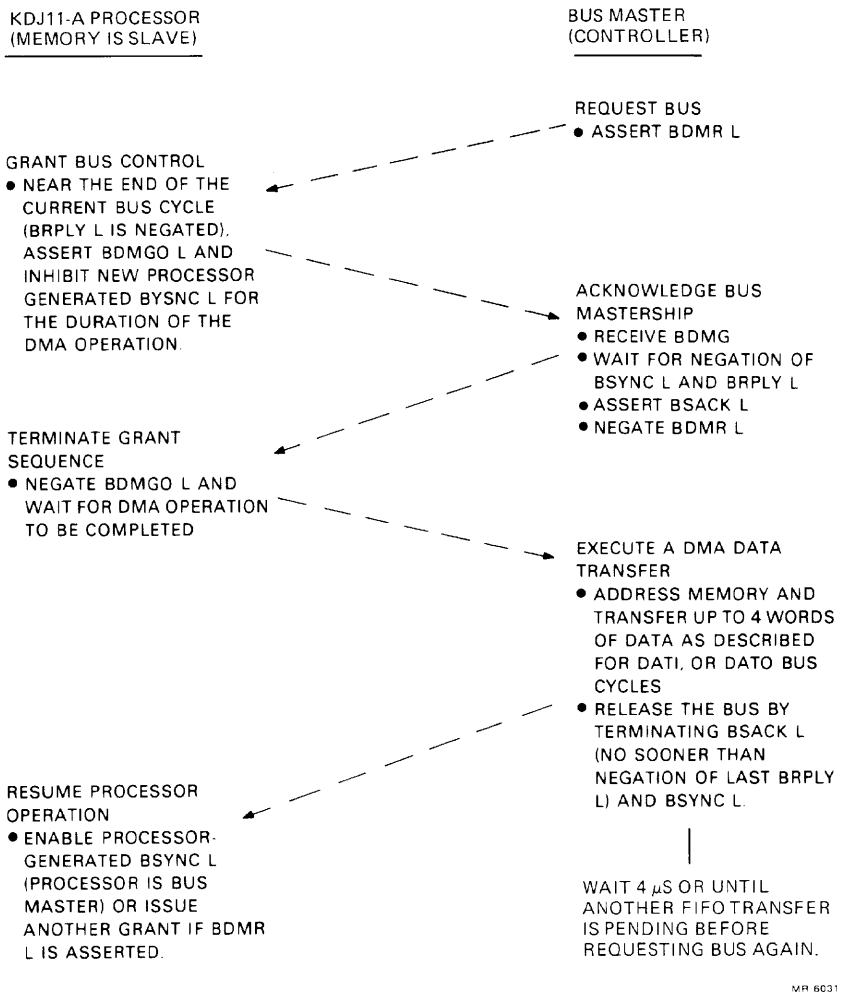
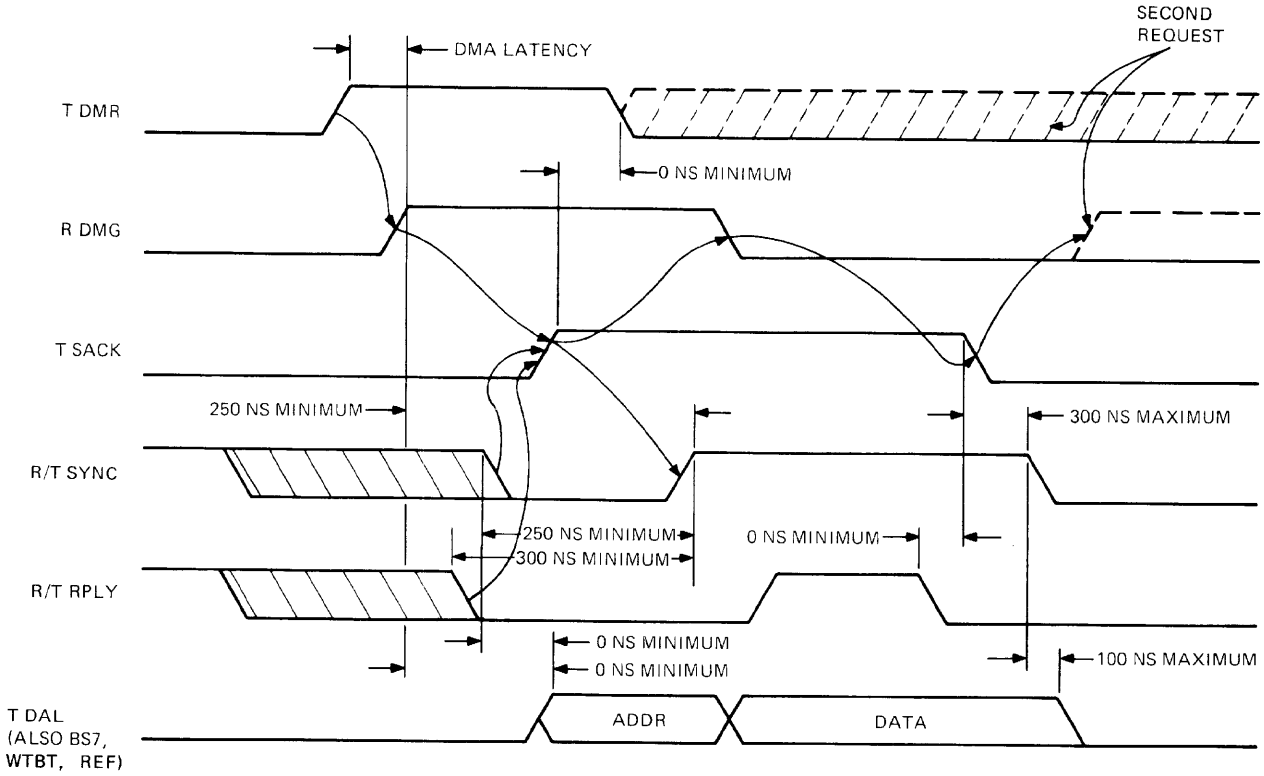


Figure 5-7 DMA Request/Grant Sequence



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.

MR-3690

Figure 5-8 DMA Request/Grant Bus Cycle Timing

5.5 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.

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.

Signal	Name
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

5.5.1 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 KDJ11-A 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 KDJ11-A systems but must be placed in a bus slot following the last bus slot in which a position-independent device is installed.

5.5.2 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 5-9. Interrupt protocol timing is shown in Figure 5-10.

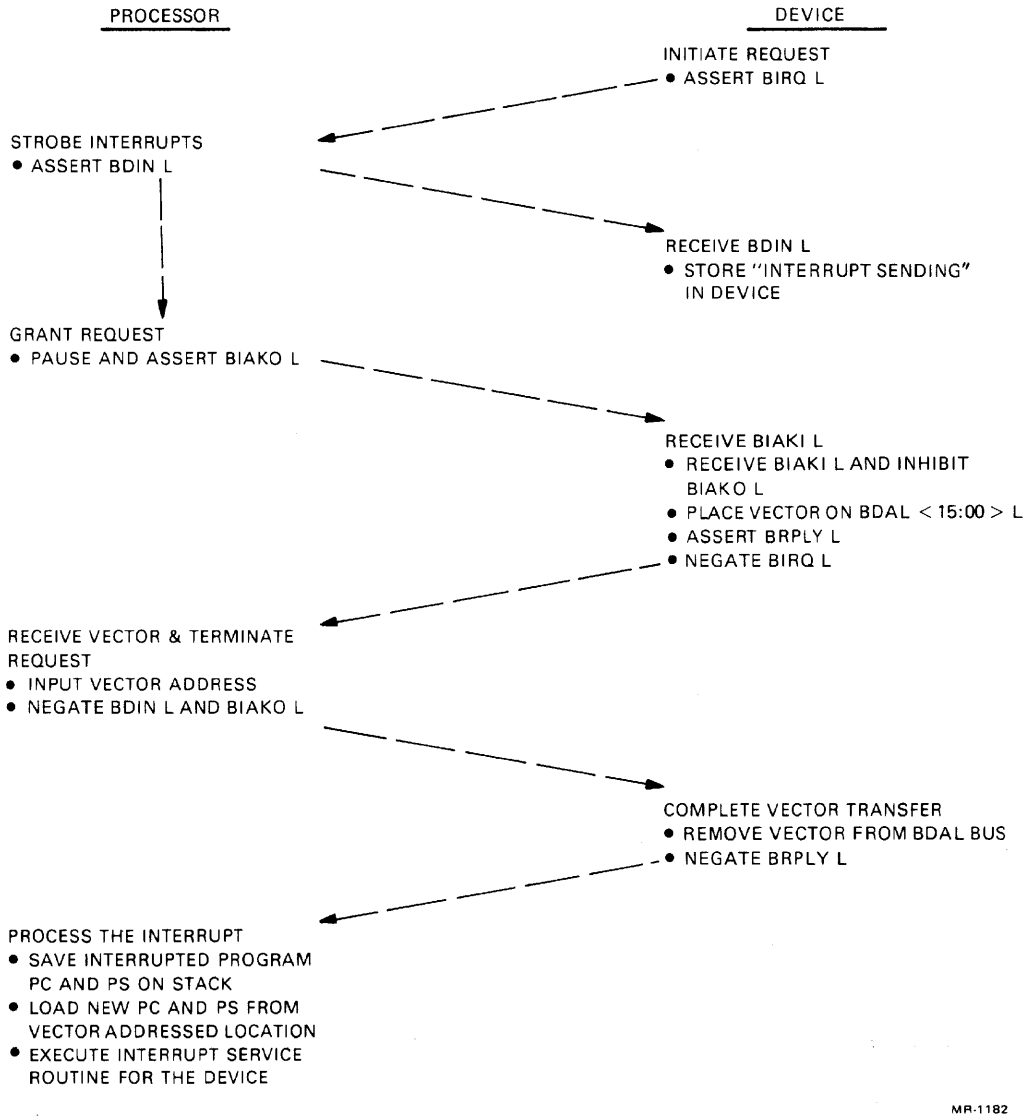


Figure 5-9 Interrupt Request/Acknowledge Sequence

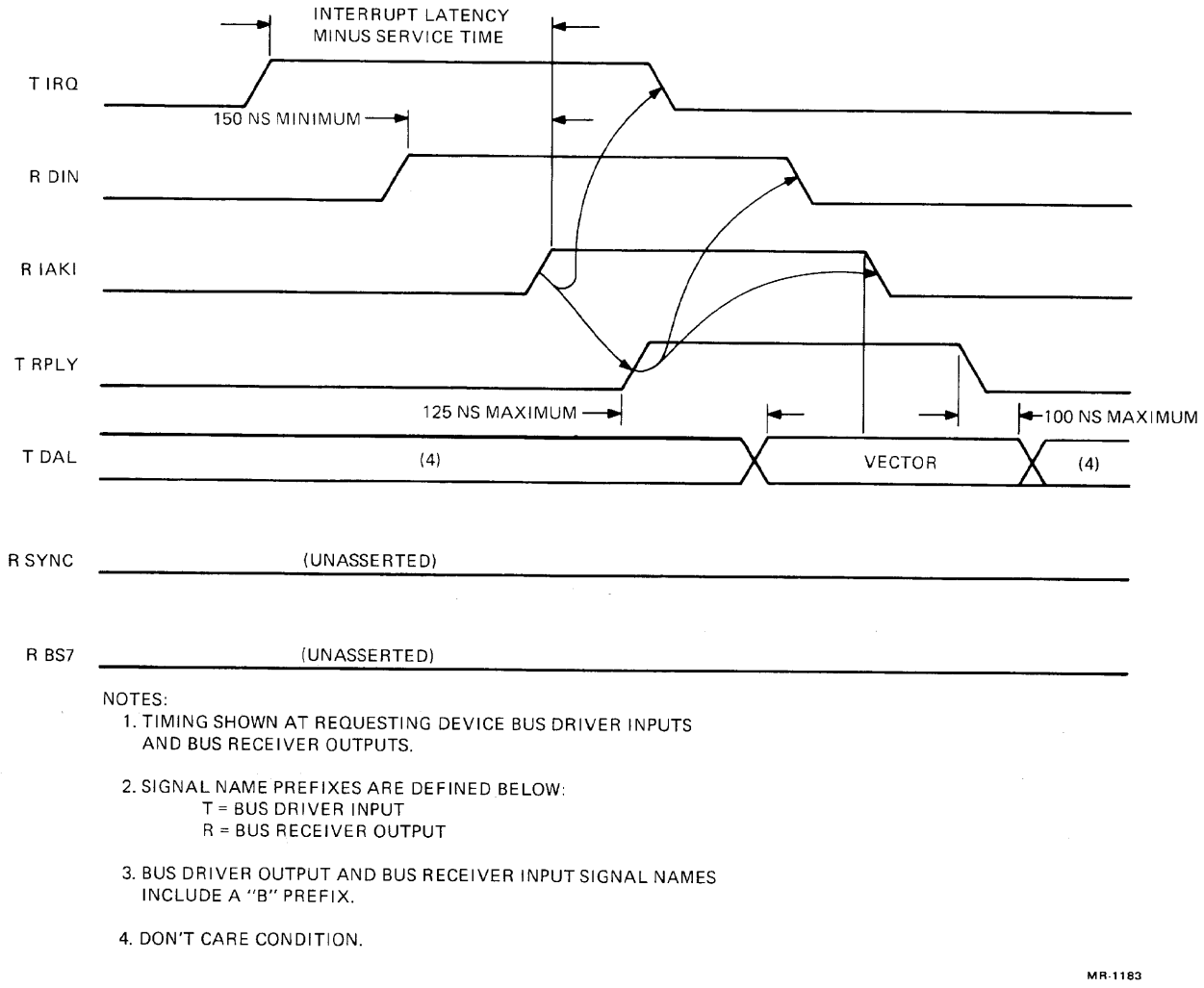


Figure 5-10 Interrupt Protocol Timing

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 KDJ11-A 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, by 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 accepts 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.

Devices that support position-independent, multilevel interrupts assert from one to three IRQ lines when requesting an interrupt. Table 5-4 presents the IRQ lines a device at each level must assert in order to request an interrupt and lists the lines it must monitor to determine whether a higher-priority device is requesting an interrupt.

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.

Table 5-4 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	

5.5.3 4-Level Interrupt Configurations

Users having high-speed peripherals and desiring 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 5-11. 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 5.5.2. 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.

The position-dependent configuration is shown in Figure 5-12. 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.

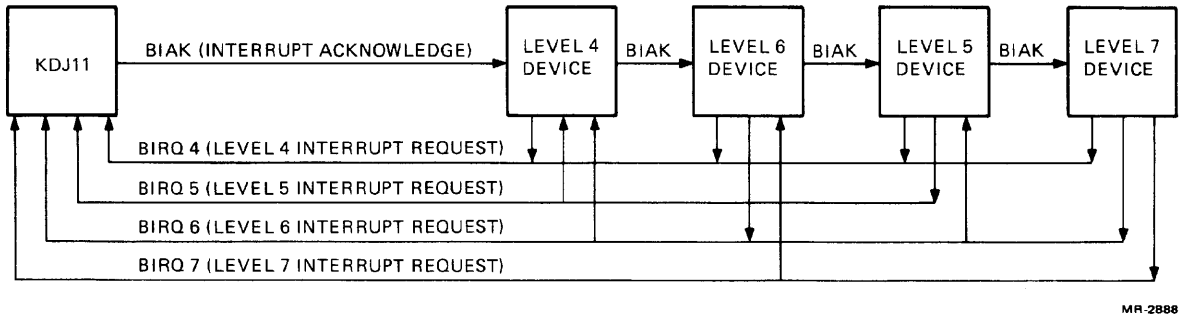


Figure 5-11 Position-Independent Configuration

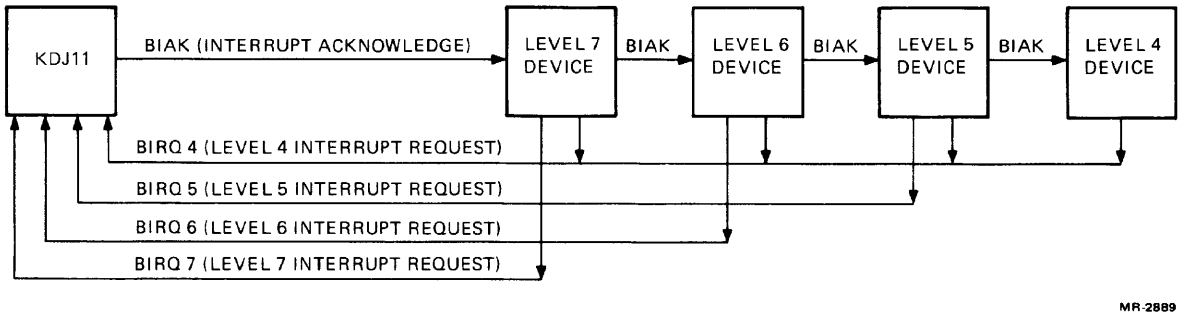


Figure 5-12 Position-Dependent Configuration

5.6 CONTROL FUNCTIONS

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

Signal	Name
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

5.6.1 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 KDJ11-A does not perform memory refresh.

5.6.2 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.

5.6.3 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

5.6.4 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.

5.6.4.1 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.

5.6.4.2 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.

5.6.4.3 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 5-13.

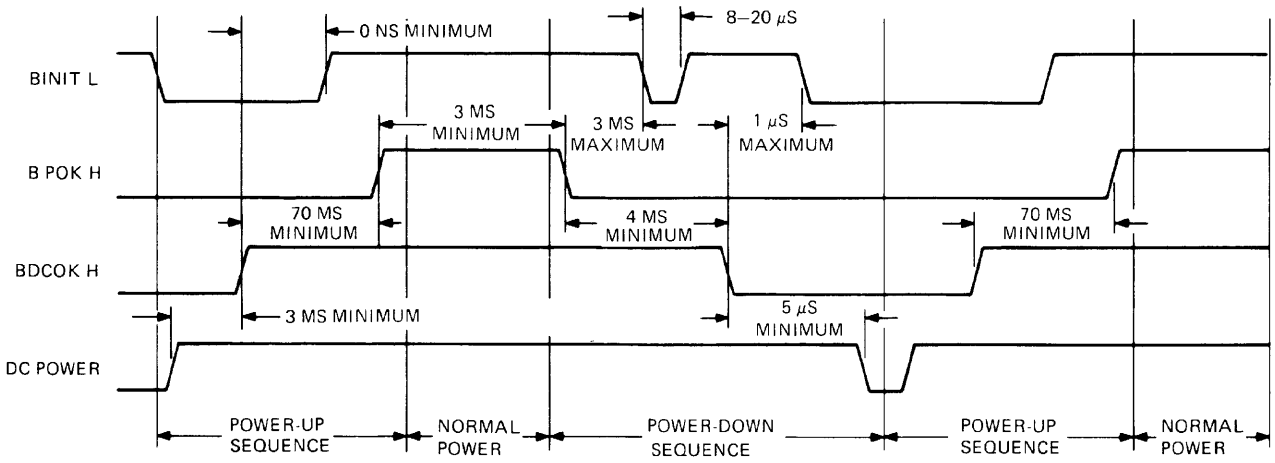


Figure 5-13 Power-Up/Power-Down Timing

5.6.4.4 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.

5.6.5 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.

5.7 BUS ELECTRICAL CHARACTERISTICS

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

5.7.1 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)

5.7.2 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.

5.7.3 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.

5.7.4 120 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).

NOTES

1. **The KDJ11-A processor (as well as all standard DIGITAL-supplied LSI-11 interfaces) connects to the bus via special drivers and receivers, described in Paragraphs 5.7.5 and 5.7.6.**

2. **The KDJ11-A processor provides resistive (250 Ω) pull-up (on all bused lines) to 3.4 Vdc for this wired-OR interconnecting scheme.**

5.7.5 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.

5.7.6 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 V 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.

5.7.7 KDJ11-A 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 250 Ω. The terminating resistors are shown in Figure 5-14. The KDJ11-A contains terminating resistor networks in 18-pin single-in-line packages to provide the 250 Ω terminations for the data/address, synchronization, and control lines at the processor end of the bus.

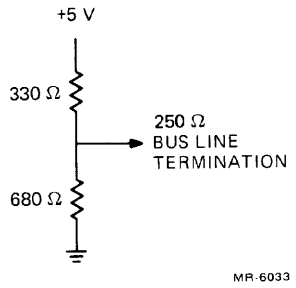


Figure 5-14 Bus Line Termination

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 5.8.1 and 5.8.2.

5.7.8 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.

5.7.8.1 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.

5.7.8.2 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.

5.7.8.3 Power and Ground – Each bus interface slot has connector pins assigned for the following dc voltages.

Voltage	Number of Pins
+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.

5.7.8.4 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.

5.8 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 5.8.1 and 5.8.2.

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.

5.8.1 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 5-15.

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.

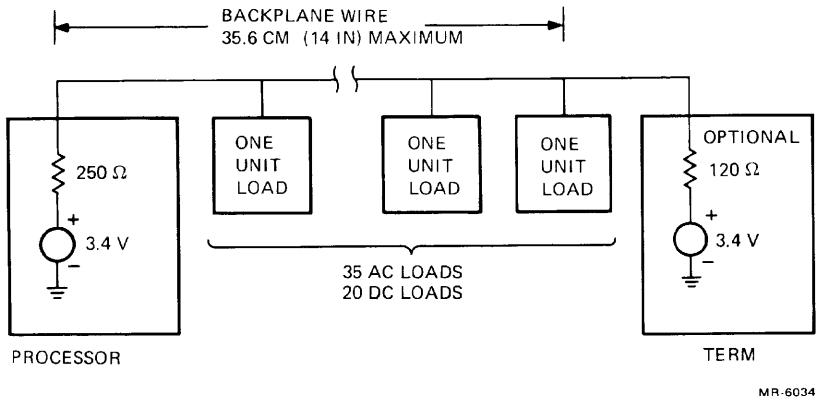


Figure 5-15 Single-Backplane Configuration

5.8.2 Rules for Configuring Multiple-Backplane Systems

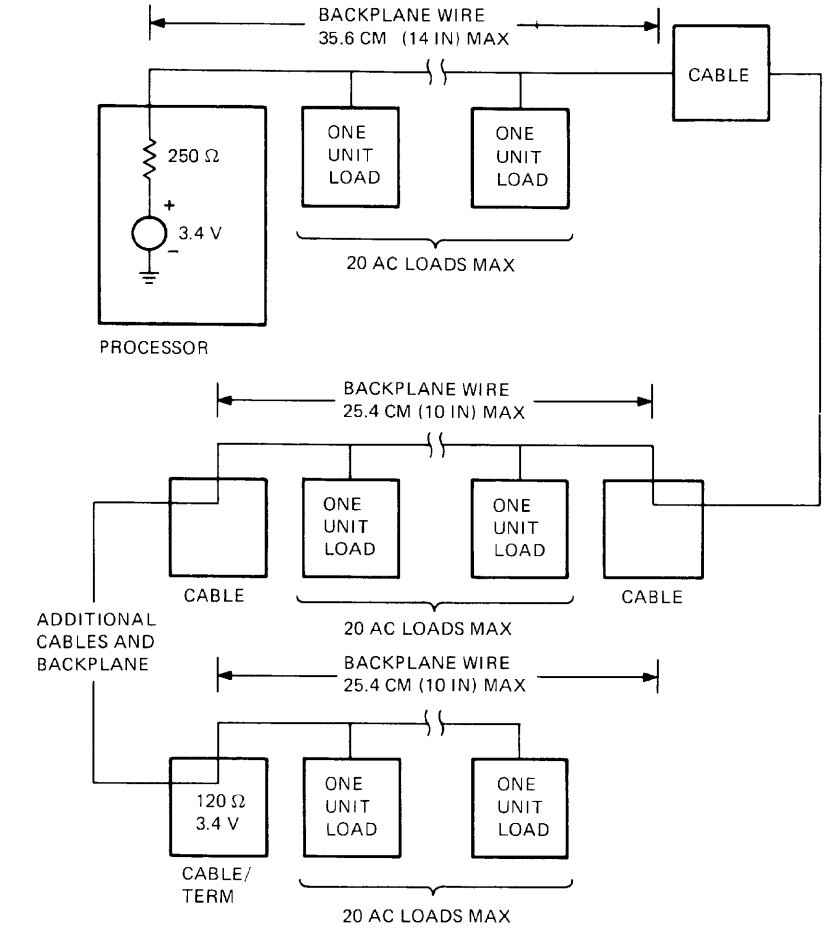
Multiple-backplane systems can contain a maximum of three backplanes. A configuration diagram for a multiple-backplane system is shown in Figure 5-16.

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.



NOTES:
1. TWO CABLES (MAX) 4.88 M (16 FT) (MAX) TOTAL LENGTH.
2. 20 DC LOADS TOTAL (MAX).

MR 6035

Figure 5-16 Multiple-Backplane Configuration

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.

5.8.3 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 6
ADDRESSING MODES AND BASE INSTRUCTION SET

6.1 INTRODUCTION

The first part of this chapter is divided into six major sections as follows.

- Single-Operand Addressing – One part of the instruction word specifies the registers; the other part provides information for locating the operand.
- Double-Operand Addressing – One part of the instruction word specifies the registers; the remaining parts provide information for locating two operands.
- Direct Addressing – The operand is the content of the selected register.
- Deferred (Indirect) Addressing – The contents of the selected register is the address of the operand.
- Use of the PC as a General-Purpose Register – The PC is different from other general-purpose registers in one important respect. Whenever the processor retrieves an instruction, it automatically advances the PC by 2. By combining this automatic advancement of the PC with four of the basic addressing modes, we produce the four special PC modes – immediate, absolute, relative, and relative-deferred.
- Use of the Stack Pointer as a General-Purpose Register – General-purpose registers can be used for stack operations.

The second part of this chapter describes each of the instructions in the KDJ11-A instruction set.

6.2 ADDRESSING MODES

Data stored in memory must be accessed and manipulated. Data handling is specified by a KDJ11-A instruction (MOV, ADD, etc.), which usually specifies the following.

- The function to be performed (operation code)
- The general-purpose register to be used when locating the source operand, and/or destination operand (where required)
- The addressing mode, which specifies how the selected registers are to be used

A large portion of the data handled by a computer is structured (in character strings, arrays, lists, etc.). The KDJ11-A addressing modes provide for efficient and flexible handling of structured data.

A general-purpose register may be used with an instruction in any of the following ways.

1. As an accumulator – The data to be manipulated resides in the register.

2. As a pointer – The contents of the register is the address of an operand, rather than the operand itself.

3. As a pointer that automatically steps through memory locations – Automatically stepping forward through consecutive locations is known as autoincrement addressing; automatically stepping backwards is known as autodecrement addressing. These modes are particularly useful for processing tabular or array data.

4. As an index register – In this instance, the contents of the register and the word following the instruction are summed to produce the address of the operand. This allows easy access to variable entries in a list.

An important KDJ11-A feature, which should be considered with the addressing modes, is the register arrangement.

- Two sets of six general-purpose registers (R0–R5 and R0'–R5')
- A hardware stack pointer (SP) register (R6) for each processor mode (kernel, supervisor, user)
- A program counter (PC) register (R7)

Registers R0–R5 and R0'–R5' are not dedicated to any specific function; their use is determined by the instruction that is decoded.

- They can be used for operand storage. For example, the contents of two registers can be added and stored in another register.
- They can contain the address of an operand or serve as pointers to the address of an operand.
- They can be used for the autoincrement or autodecrement features.
- They can be used as index registers for convenient data and program access.

The KDJ11-A also has instruction addressing mode combinations that facilitate temporary data storage structures. These can be used for convenient handling of data that must be accessed frequently. This is known as stack manipulation. The register that keeps track of stack manipulation is known as the stack pointer (SP). Any register can be used as a stack pointer under program control; however, certain instructions associated with subroutine linkage and interrupt service automatically use register R6 as a "hardware stack pointer." For this reason, R6 is frequently referred to as the SP.

- The stack pointer (SP) keeps track of the latest entry on the stack.
- The stack pointer moves down as items are added to the stack and moves up as items are removed. Therefore, the stack pointer always points to the top of the stack.
- The hardware stack is used during trap or interrupt handling to store information, allowing an orderly return to the interrupted program.

Register R7 is used by the processor as its program counter (PC). It is recommended that R7 not be used as a stack pointer or accumulator. Whenever an instruction is fetched from memory, the program counter is automatically incremented by two to point to the next instruction word.

6.2.1 Single-Operand Addressing

The instruction format for all single-operand instructions (such as CLR, INC, TST) is shown in Figure 6-1.

Bits <15:06> specify the operation code that defines the type of instruction to be executed.

Bits <05:00> form a 6-bit field called the destination address field. The destination address field consists of two subfields:

- Bits <05:03> specify the destination mode. Bit 03 is set to indicate deferred (indirect) addressing.
- Bits <02:00> specify which of the 8 general-purpose registers is to be referenced by this instruction word.

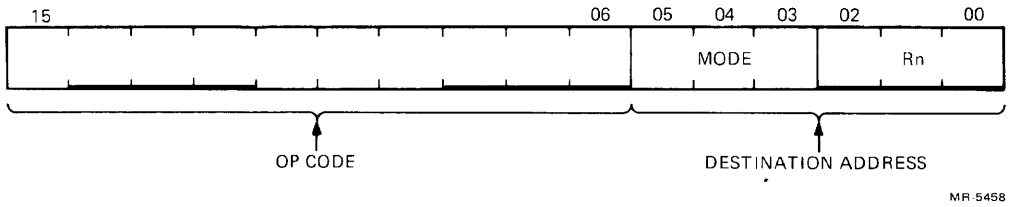


Figure 6-1 Single-Operand Addressing

6.2.2 Double-Operand Addressing

Operations that imply two operands (such as ADD, SUB, MOV, and CMP) are handled by instructions that specify two addresses. The first operand is called the source operand; the second is called the destination operand. Bit assignments in the source and destination address fields may specify different modes and different registers. The instruction format for the double operand instruction is shown in Figure 6-2.

The source address field is used to select the source operand (the first operand). The destination is used similarly, and locates the second operand and the result. For example, the instruction ADD A, B adds the contents (source operand) of location A to the contents (destination operand) of location B. After execution, B will contain the result of the addition and the contents of A will be unchanged.

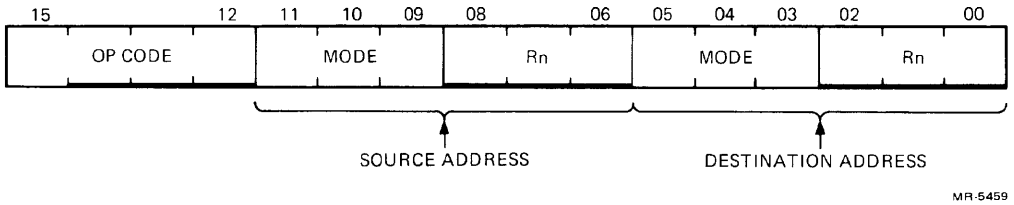


Figure 6-2 Double-Operand Addressing

Examples in this paragraph and the rest of the chapter use the following sample KDJ11-A instructions. (A complete listing of the KDJ11-A instructions appears in Paragraph 6.3.)

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 one to contents of the destination.)	0052DD
INCB	Increment byte. (Add one to the contents of the destination byte.)	1052DD
COM	Complement. (Replace the contents of the destination by its logical complement; each 0 bit is set and each 1 bit is cleared.)	0051DD
COMB	Complement byte. (Replace the contents of the destination byte by its logical complement; 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 (six bits)
SS = source field (six bits)
() = contents of

6.2.3 Direct Addressing

The following summarizes the four basic modes used with direct addressing.

Direct Modes (Figures 6-3 to 6-6)

Mode	Name	Assembler Syntax	Function
0	Register	Rn	Register contains operand.

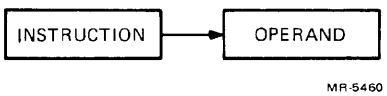


Figure 6-3 Mode 0 Register

Mode	Name	Assembler Syntax	Function
2	Autoincrement	(Rn)+	Register is used as a pointer to sequential data and then incremented.

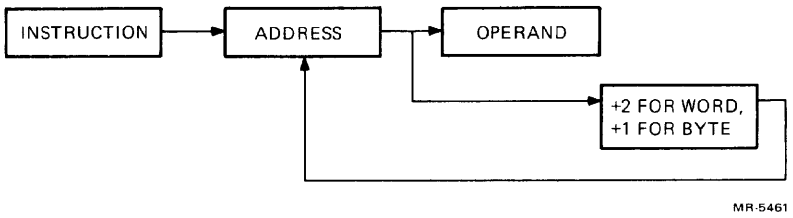


Figure 6-4 Mode 2 Autoincrement

Mode	Name	Assembler Syntax	Function
4	Autodecrement	–(Rn)	Register is decremented and then used as a pointer.

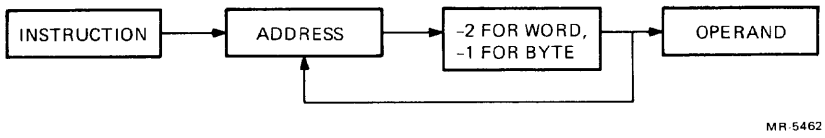


Figure 6-5 Mode 4 Autodecrement

Mode	Name	Assembler Syntax	Function
6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) is modified.

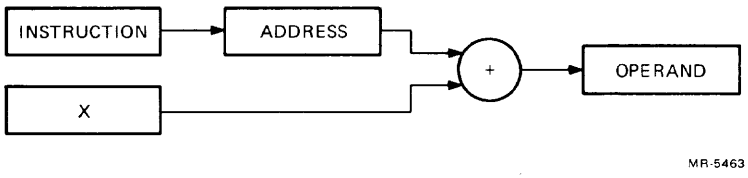


Figure 6-6 Mode 6 Index

6.2.3.1 Register Mode – With register mode any of the general registers may be used as simple accumulators, with the operand contained in the selected register. Since they are hardware registers (within the processor), the general registers operate at high speeds and provide speed advantages when used for operating on frequently accessed variables. The assembler interprets and assembles instructions of the form OPR Rn as register mode operations. Rn represents a general register name or number and OPR is used to represent a general instruction mnemonic. Assembler syntax requires that a general register be defined as follows.

R0 = %0 (% sign indicates register definition)
R1 = %1
R2 = %2, etc.

Registers are typically referred to by name as R0, R1, R2, R3, R4, R5, R6, and R7. However, R6 and R7 are also referred to as SP and PC, respectively.

Register Mode Examples (Figures 6-7 to 6-9)

1.	**Symbolic**	**Octal Code**	**Instruction Name**
	INC R3	005203	Increment

Operation: Add one to the contents of general-purpose register R3.

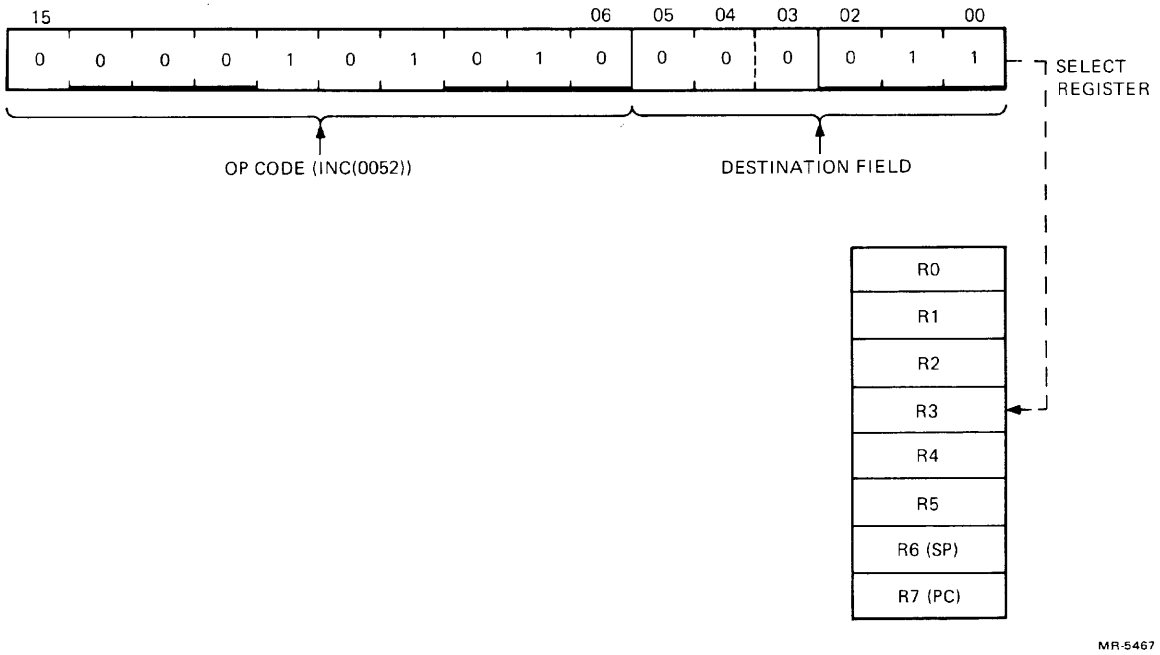


Figure 6-7 INC R3 Increment

2.	**Symbolic**	**Octal Code**	**Instruction Name**
	ADD R2, R4	060204	Add

Operation: Add the contents of R2 to the contents of R4.

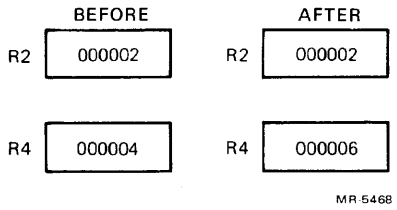


Figure 6-8 ADD R2,R4 Add

3.	**Symbolic**	**Octal Code**	**Instruction Name**
	COMB R4	105104	Complement byte

Operation: 1's complement bits <07:00> (byte) in R4. (When general registers are used, byte instructions operate only on bits <07:00>; i.e., byte 0 of the register.)

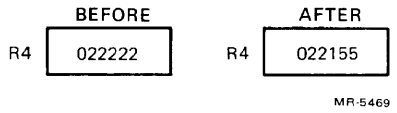


Figure 6-9 COMB R4 Complement Byte

6.2.3.2 Autoincrement Mode [OPR (Rn)+] – This mode (mode 2) provides for automatic stepping of a pointer through sequential elements of a table of operands. It assumes the contents of the selected general-purpose register to be the address of the operand. Contents of registers are stepped (by one for byte instructions, by two for word instructions, always by two for R6 and R7) to address the next sequential location. The autoincrement mode is especially useful for array processing and stack processing. It will access an element of a table and then step the pointer to address the next operand in the table. Although most useful for table handling, this mode is completely general and may be used for a variety of purposes.

Autoincrement Mode Examples (Figures 6-10 to 6-12)

1.	**Symbolic**	**Octal Code**	**Instruction Name**
	CLR (R5)+	005025	Clear

Operation: Use contents of R5 as the address of the operand. Clear selected operand and then increment the contents of R5 by two.

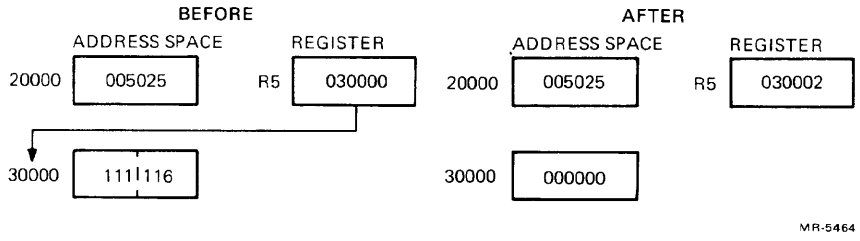


Figure 6-10 CLR (R5)+ Clear

2.	**Symbolic**	**Octal Code**	**Instruction Name**
	CLRB (R5)+	105025	Clear byte

Operation: Use contents of R5 as the address of the operand. Clear selected byte operand and then increment the contents of R5 by one.

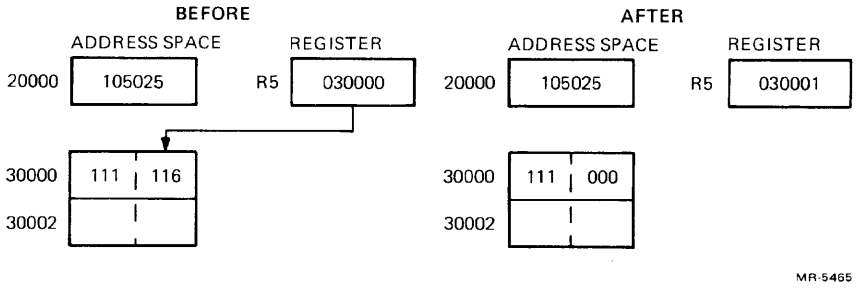


Figure 6-11 CLRB (R5)+ Clear Byte

3.	**Symbolic**	**Octal Code**	**Instruction Name**
	ADD (R2)+,R4	062204	Add

Operation: The contents of R2 are used as the address of the operand, which is added to the contents of R4. R2 is then incremented by two.

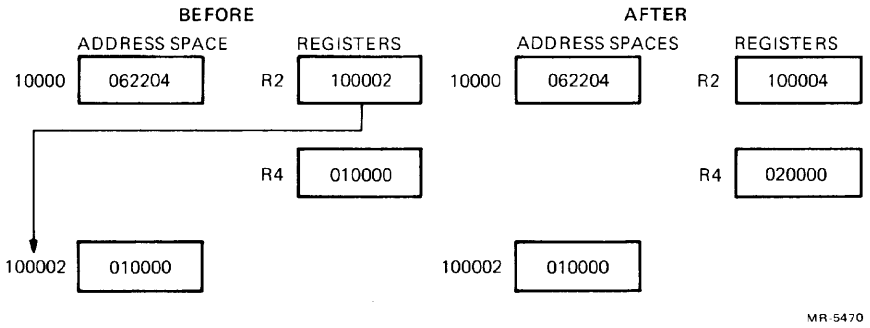


Figure 6-12 ADD (R2)+,R4 Add

6.2.3.3 Autodecrement Mode [OPR-(Rn)] – This mode (mode 4) is useful for processing data in a list in reverse direction. The contents of the selected general-purpose register are decremented (by one for byte instructions, by two for word instructions) and then used as the address of the operand. The choice of postincrement, predecrement features for the KDJ11-A were not arbitrary decisions, but were intended to facilitate hardware/software stack operations.

Autodecrement Mode Examples (Figures 6-13 to 6-15)

1.	**Symbolic**	**Octal Code**	**Instruction Name**
	INC –(R0)	005240	Increment

Operation: The contents of R0 are decremented by two and used as the address of the operand. The operand is incremented by one.

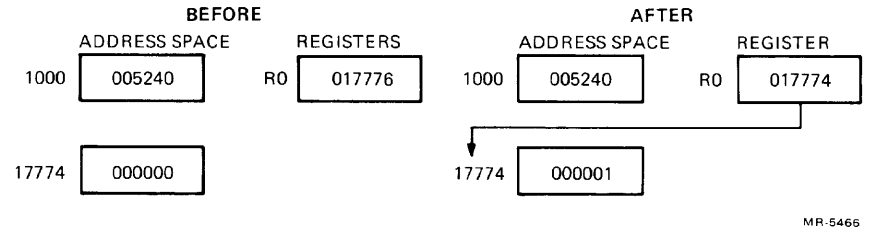


Figure 6-13 INC –(R0) Increment

2.	Symbolic	Octal Code	Instruction Name
	INCB –(R0)	105240	Increment byte

Operation: The contents of R0 are decremented by one and then used as the address of the operand. The operand byte is increased by one.

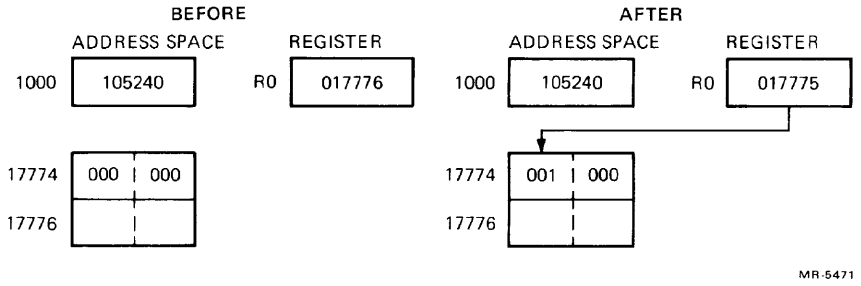


Figure 6-14 INCB –(R0) Increment Byte

3.	Symbolic	Octal Code	Instruction Name
	ADD –(R3),R0	064300	Add

Operation: The contents of R3 are decremented by two and then used as a pointer to an operand (source), which is added to the contents of R0 (destination operand).

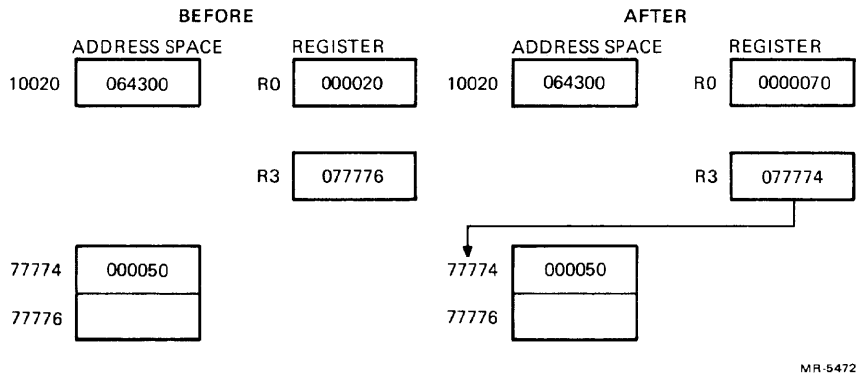


Figure 6-15 ADD –(R3),R0 Add

6.2.3.4 Index Mode [OPR X(Rn)] – In this mode (mode 6) the contents of the selected general-purpose register, and an index word following the instruction word, are summed to form the address of the operand. The contents of the selected register may be used as a base for calculating a series of addresses, thus allowing random access to elements of data structures. The selected register can then be modified by program to access data in the table. Index addressing instructions are of the form OPR X(Rn), where X is the indexed word located in the memory location following the instruction word and Rn is the selected general-purpose register.

Index Mode Examples (Figures 6-16 to 6-18)

1.	**Symbolic**	**Octal Code**	**Instruction Name**
	CLR 200(R4)	005064 000200	Clear

Operation: The address of the operand is determined by adding 200 to the contents of R4. The operand location is then cleared.

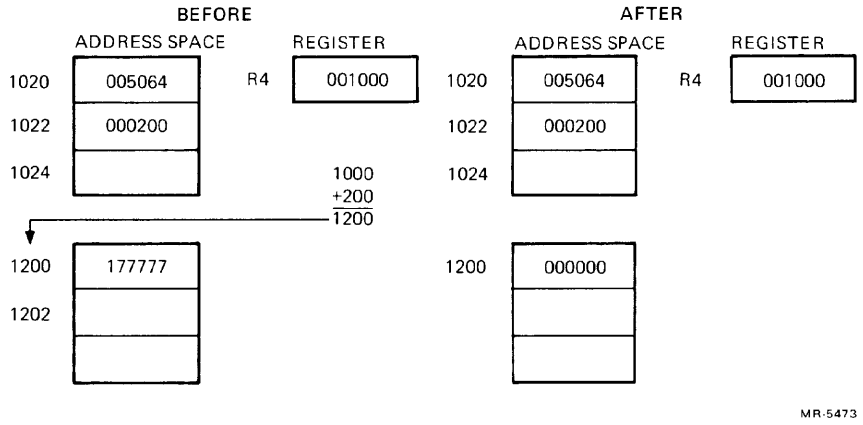


Figure 6-16 CLR 200(R4) Clear

2. **Symbolic** **Octal Code** **Instruction Name**

COMB 200(R1)	105161 000200	Complement byte

Operation: The contents of a location, which are determined by adding 200 to the contents of R1, are 1's complemented (i.e., logically complemented).

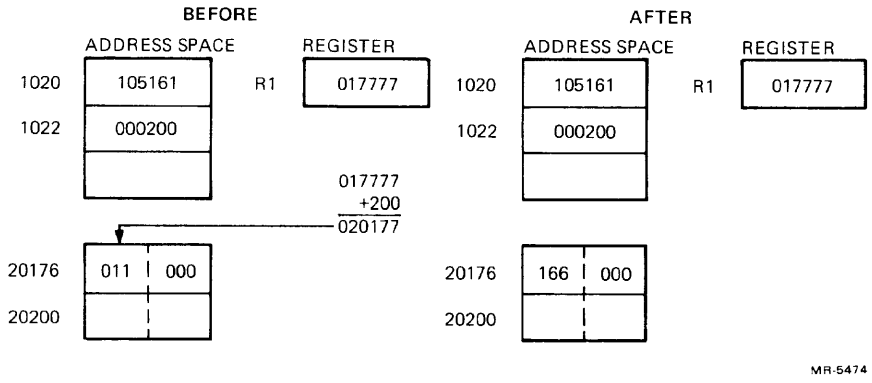


Figure 6-17 COMB 200(R1) Complement Byte

3. **Symbolic** **Octal Code** **Instruction Name**

ADD 30(R2),20(R5)	066265 000030 000020	Add

Operation: The contents of a location, which are determined by adding 30 to the contents of R2, are added to the contents of a location that is determined by adding 20 to the contents of R5. The result is stored at the destination address, that is, 20(R5).

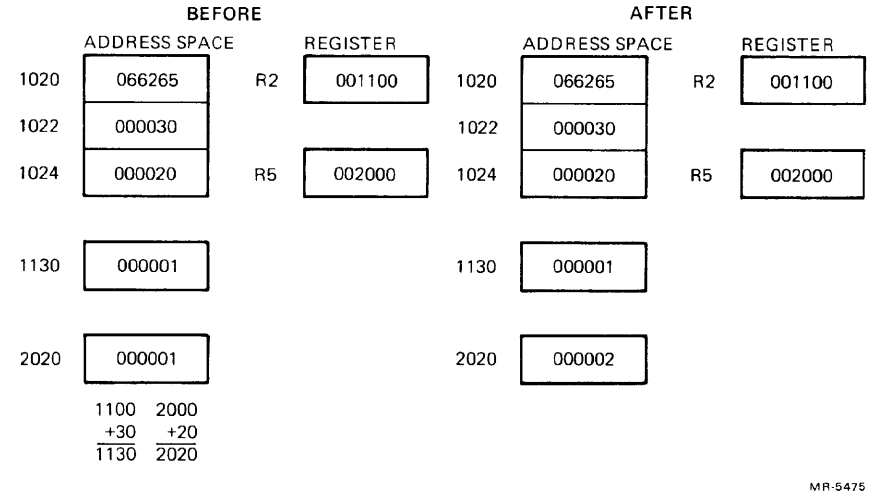


Figure 6-18 ADD 30(R2),20(R5) Add

6.2.4 Deferred (Indirect) Addressing

The four basic modes may also be used with deferred addressing. Whereas in register mode the operand is the contents of the selected register, in register-deferred mode the contents of the selected register is the address of the operand.

In the three other deferred modes, the contents of the register select the address of the operand rather than the operand itself. These modes are therefore used when a table consists of addresses rather than operands. The assembler syntax for indicating deferred addressing is @ [or () when this is not ambiguous]. The following summarizes the deferred versions of the basic modes.

Deferred Modes (Figures 6-19 to 6-22)

Mode	Name	Assembler Syntax	Function
1	Register-deferred	@Rn or (Rn)	Register contains the address of the operand.

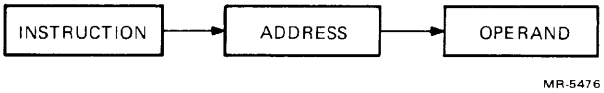


Figure 6-19 Mode 1 Register-Deferred

Mode	Name	Assembler Syntax	Function
3	Autoincrement-deferred	@(Rn)+	Register is first used as a pointer to a word containing the address of the operand and then incremented (always by two, even for byte instructions).

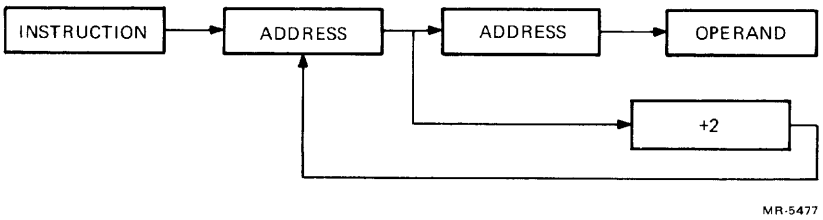


Figure 6-20 Mode 3 Autoincrement-Deferred

Mode	Name	Assembler Syntax	Function
5	Autodecrement-deferred	@–(Rn)	Register is decremented (always by two, even for byte instructions) and then used as a pointer to a word containing the address of the operand.

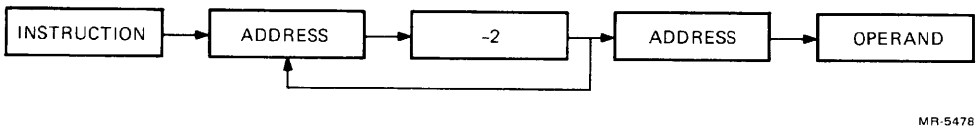


Figure 6-21 Mode 5 Autodecrement-Deferred

Mode	Name	Assembler Syntax	Function
7	Index-deferred	@X(Rn)	Value X (stored in a word following the instruction) and (Rn) are added; the sum is used as a pointer to a word containing the address of the operand. Neither X nor (Rn) is modified.

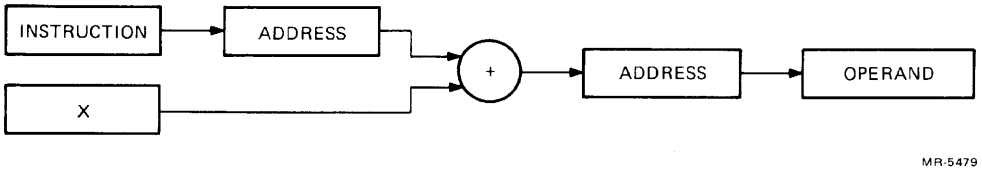


Figure 6-22 Mode 7 Index-Deferred

The following examples illustrate the deferred modes.

Register-Deferred Mode Example (Figure 6-23)

Symbolic	Octal Code	Instruction Name
CLR @R5	005015	Clear

Operation: The contents of location specified in R5 are cleared.

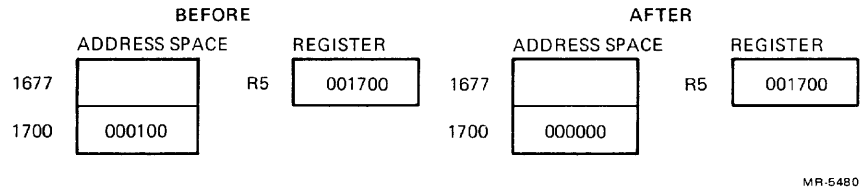


Figure 6-23 CLR @R5 Clear

Autoincrement-Deferred Mode Example (Mode 3) (Figure 6-24)

Symbolic	Octal Code	Instruction Name
INC @(R2)+	005232	Increment

Operation: The contents of R2 are used as the address of the address of the operand. The operand is increased by one; the contents of R2 are incremented by two.

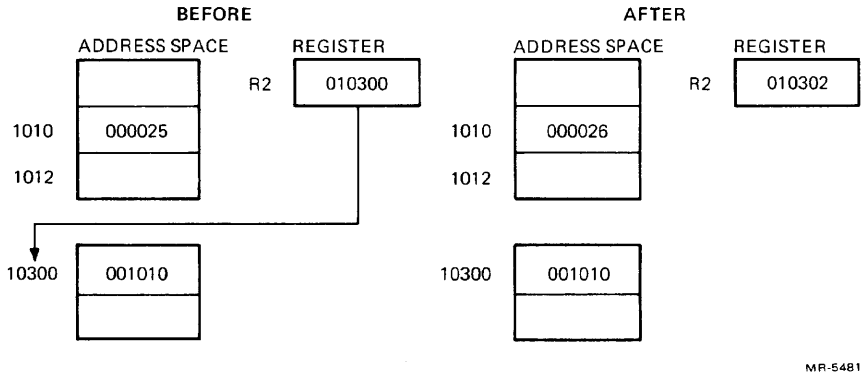


Figure 6-24 INC @(R2)+ Increment

Autodecrement-Deferred Mode Example (Mode 5) (Figure 6-25)

Symbolic	Octal Code
COM @–(R0)	005150

Operation: The contents of R0 are decremented by two and then used as the address of the address of the operand. The operand is 1's complemented (i.e., logically complemented).

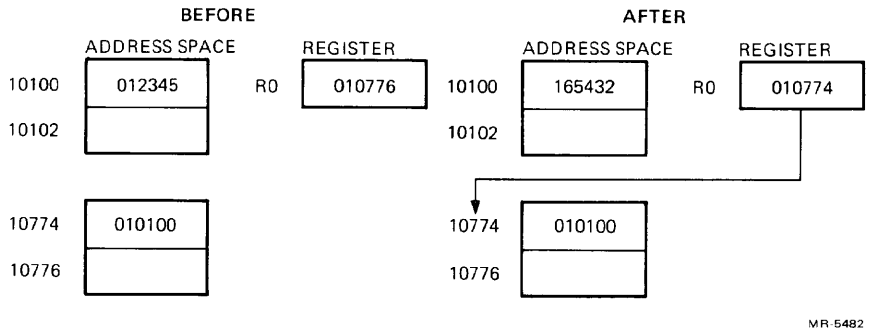


Figure 6-25 COM @–(R0) Complement

Index-Deferred Mode Example (Mode 7) (Figure 6-26)

Symbolic	Octal Code	Instruction Name
ADD @1000(R2),R1	067201 001000	Add

Operation: 1000 and the contents of R2 are summed to produce the address of the address of the source operand, the contents of which are added to the contents of R1; the result is stored in R1.

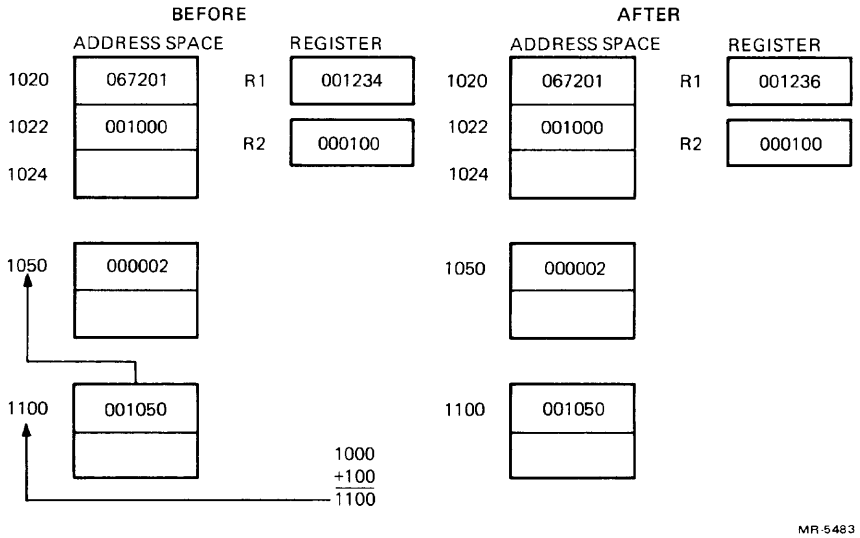


Figure 6-26 ADD @1000(R2),R1 Add

6.2.5 Use of the PC as a General-Purpose Register

Although register 7 is a general-purpose register, it doubles in function as the program counter for the KDJ11-A. Whenever the processor uses the program counter to acquire a word from memory, the program counter is automatically incremented by two to contain the address of the next word of the instruction being executed or the address of the next instruction to be executed. (When the program uses the PC to locate byte data, the PC is still incremented by two.)

The PC responds to all the standard KDJ11-A addressing modes. However, with four of these modes the PC can provide advantages for handling position-independent code and unstructured data. When utilizing the PC, these modes are termed immediate, absolute (or immediate-deferred), relative, and relative-deferred. The modes are summarized below.

Mode	Name	Assembler Syntax	Function
2	Immediate	#n	Operand follows instruction.
3	Absolute	@#A	Absolute address of operand follows instruction.
6	Relative	A	Relative address (index value) follows the instruction.
7	Relative-deferred	@A	Index value (stored in the word after the instruction) is the relative address for the address of the operand.

When a standard program is available for different users, it is often helpful to be able to load it into different areas of memory and run it in those areas. The KDJ11-A can accomplish the relocation of a program very efficiently through the use of position-independent code (PIC), which is written by using the PC addressing modes. If an instruction and its operands are moved in such a way that the relative distance between them is not altered,
the same offset relative to the PC can be used in all positions in memory. Thus, PIC usually references locations relative to the current location.

The PC also greatly facilitates the handling of unstructured data. This is particularly true of the immediate and relative modes.

6.2.5.1 Immediate Mode [OPR #n,DD] – Immediate mode (mode 2) is equivalent in use to 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.

Immediate Mode Example (Figure 6-27)

Symbolic	Octal Code	Instruction Name
ADD #10,R0	062700 000010	Add

Operation: 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 two. 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 it is incremented by two to point to the next instruction.

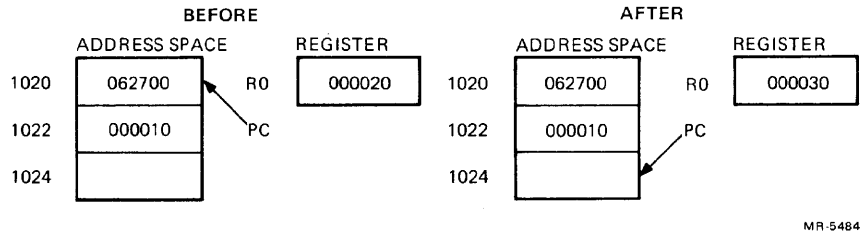


Figure 6-27 ADD #10,R0 Add

6.2.5.2 Absolute Addressing Mode [OPR @#A]– This mode (mode 3) is the equivalent of immediate-deferred or autoincrement-deferred 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).

Absolute Mode Examples (Figures 6-28 and 6-29)

1.	**Symbolic**	**Octal Code**	**Instruction Name**
	CLR @#1100	005037 001100	Clear

Operation: Clear the contents of location 1100.

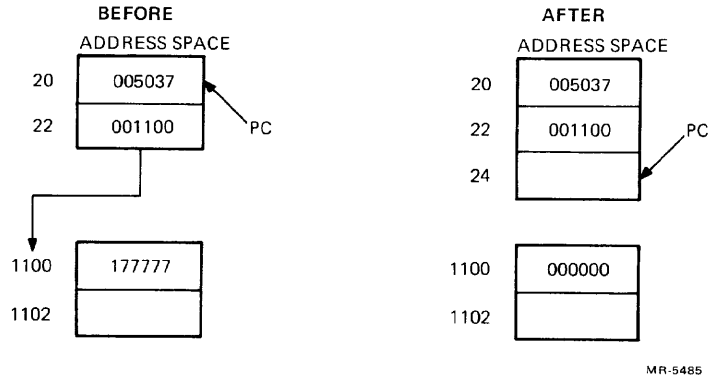


Figure 6-28 CLR @ #1100 Clear

2.	**Symbolic**	**Octal Code**	**Instruction Name**
	ADD @#2000,R3	063703 002000	Add

Operation: Add contents of location 2000 to R3.

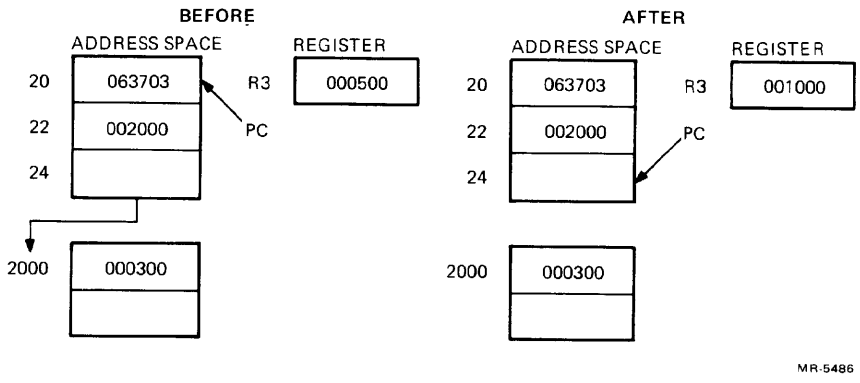


Figure 6-29 ADD @ #2000 Add

6.2.5.3 Relative Addressing Mode [OPR A or OPR X(PC)] – This mode (mode 6) is assembled as index mode using R7. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand, but the number which, when added to the (PC), becomes the address of the operand. This mode is useful for writing position-independent code since the location referenced is always fixed relative to the PC. When instructions are to be relocated, the operand is moved by the same amount. The instruction OPR X(PC) is interpreted as "X is the location of A relative to the PC."

Relative Addressing Mode Example (Figure 6-30)

Symbolic	Octal Code	Instruction Name
INC A	005267 000054	Increment

Operation: To increment location A, contents of memory location immediately following instruction word are added to (PC) to produce address A. Contents of A are increased by one.

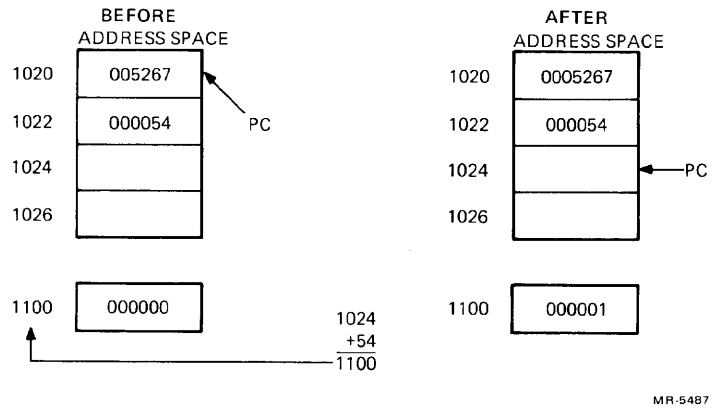


Figure 6-30 INC A Increment

6.2.5.4 Relative-Deferred Addressing Mode [OPR @A or OPR @X(PC)] – This mode (mode 7) is similar to relative mode, except that the second word of the instruction, when added to the PC, contains the address of the address of the operand, rather than the address of the operand. The instruction OPR @X(PC) is interpreted as "X is the location containing the address of A, relative to the PC."

Relative-Deferred Mode Example (Figure 6-31)

Symbolic	Octal Code	Instruction Name
CLR @A	005077 000020	Clear

Operation: Add second word of instruction to updated PC to produce address of address of operand. Clear operand.

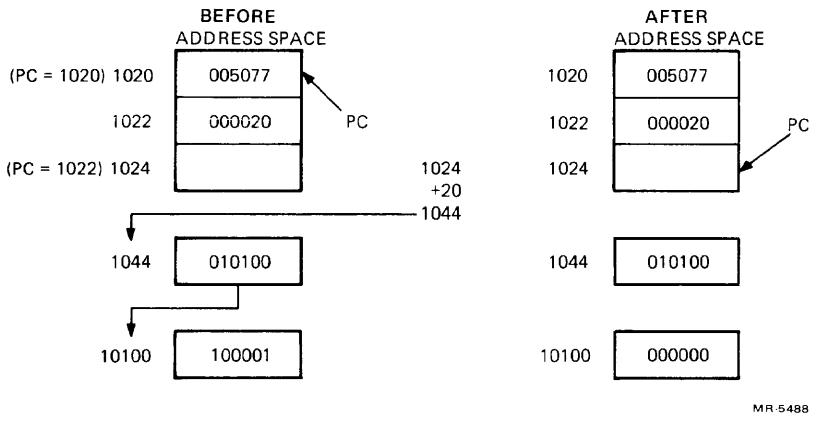


Figure 6-31 CLR @A Clear

6.2.6 Use of the Stack Pointer as a General-Purpose Register

The processor stack pointer (SP, register 6) is, in most cases, the general register used for the stack operations related to program nesting. Autodecrement with register 6 "pushes" data onto the stack, and autoincrement with register 6 "pops" data off the stack. Since the SP is used by the processor for interrupt handling, it has a special attribute: autoincrements and autodecrements are always done in steps of two. Byte operations using the SP in this way leave odd addresses unmodified.

6.3 INSTRUCTION SET

The rest of this chapter describes the KDJ11-A instruction set. The explanation of each instruction includes the instruction's mnemonic, octal code, binary code, a diagram showing the format of the instruction, a symbolic notation describing its execution and effect on the condition codes, a description, special comments, and examples. Each explanation is headed by its mnemonic. When the word instruction has a byte equivalent, the byte mnemonic also appears.

The diagram that accompanies each instruction shows the octal op code, binary op code, and bit assignments. [Note that in byte instructions, the most significant bit (bit 15) is always a one.]

Symbols:

() = contents of	$\vee$ = Boolean OR
SS or src = source address	$\forall$ = exclusive OR
DD or dst = destination address	$\sim$ = Boolean not
loc = location	REG or R = register
$\leftarrow$ = becomes	B = byte
$\uparrow$ = "is popped from stack"	■ = 0 for word, 1 for byte
$\downarrow$ = "is pushed onto stack"	, = concatenated
$\wedge$ = Boolean AND	

6.3.1 Instruction Formats

The following formats include all instructions used in the KDJ11-A. Refer to individual instructions for more detailed information.

1. Single-Operand Group: (Figure 6-32)

   CLR, CLRB, COM, COMB, INC, INCB, DEC, DECB, NEG, NEGB, ADC, ADCB, SBC, SBCB, TST, TSTB, ROR, RORB, ROL, ROLB, ASR, ASRB, ASL, ASLB, JMP, SWAB, MFPS, MTPS, SXT, TSTSET, WRTLCK, XOR

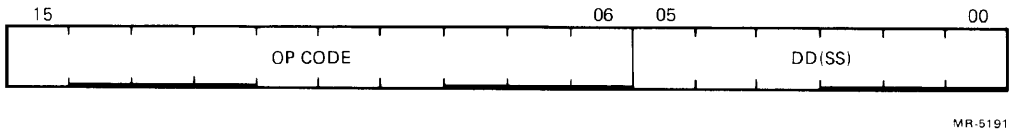


Figure 6-32 Single-Operand Group

2. Double-Operand Group:

   a. Group 1: (Figure 6-33)

   BIT, BITB, BIC, BICB, BIS, BISB, ADD, SUB, MOV, MOVB, CMP, CMPB

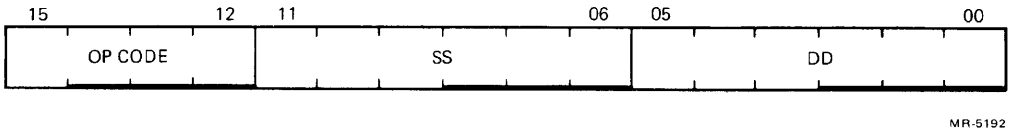


Figure 6-33 Double-Operand Group 1

   b. Group 2: (Figure 6-34)

   ASH, ASHC, DIV, MUL

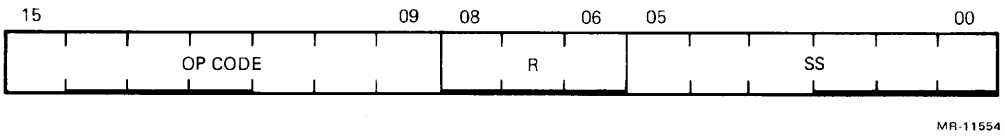


Figure 6-34 Double-Operand Group 2

3. Program Control Group:

a. Branch (all branch instructions) (Figure 6-35)

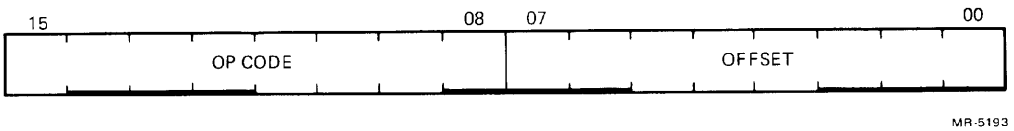


Figure 6-35 Program Control Group Branch

b. Jump to Subroutine (JSR) (Figure 6-36)

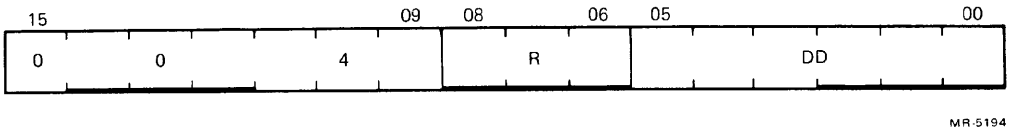


Figure 6-36 Program Control Group JSR

c. Subroutine Return (RTS) (Figure 6-37)

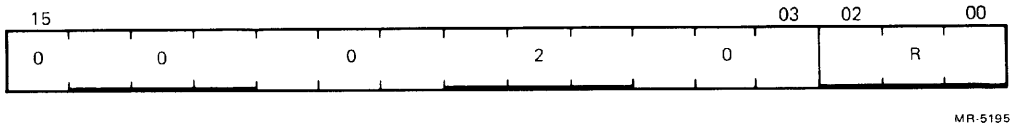


Figure 6-37 Program Control Group RTS

d. Traps (breakpoint, IOT, EMT, TRAP, BPT) (Figure 6-38)

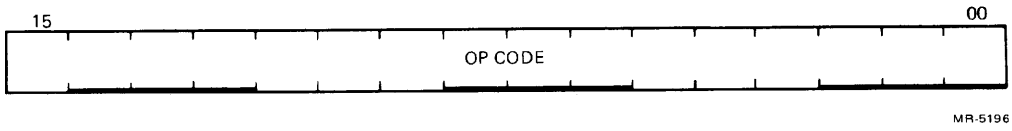


Figure 6-38 Program Control Group Traps

e. Subtract 1 and Branch (if = 0) (SOB) (Figure 6-39)

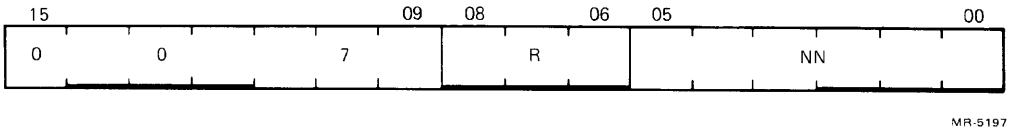


Figure 6-39 Program Control Group Subtract

f. Mark (Figure 6-40)

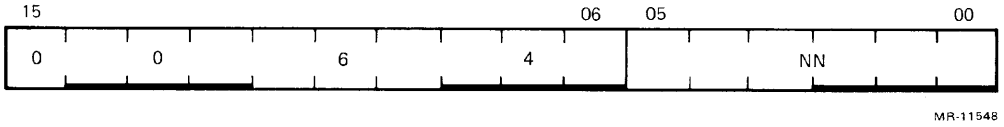


Figure 6-40 Mark

g. Call to Supervisor Mode (CSM) (Figure 6-41)

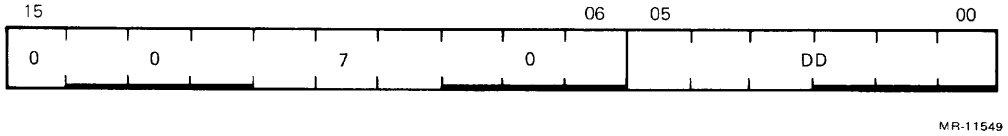


Figure 6-41 Call to Supervisor Mode

h. Set Priority Level (SPL) (Figure 6-42)

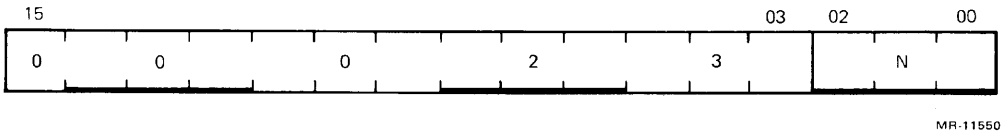


Figure 6-42 Set Priority Level

4. Operate Group: HALT, WAIT, RTI, RESET, RTT, NOP, MFPT
(Figure 6-43)

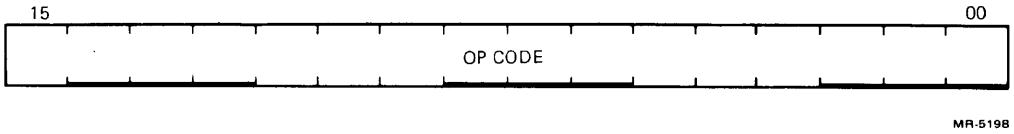


Figure 6-43 Operate Group

5. Condition Code Operators (all condition code instructions)
(Figure 6-44)

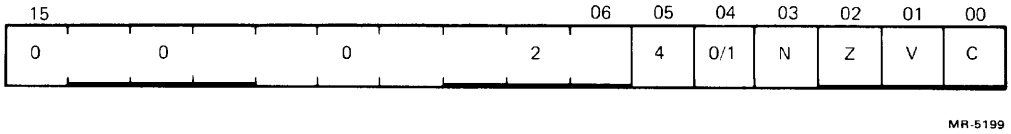


Figure 6-44 Condition Group

6. Move To/From Previous Instruction/Data Space Group: MTPD, MTPI, MFPD, MFPI
(Figure 6-45)

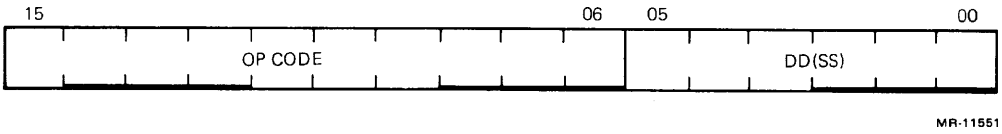


Figure 6-45 Move To And From
Previous Instruction/Data Space Group

6.3.2 Byte Instructions

The KDJ11-A includes a full complement of instructions that manipulate byte operands. Since all KDJ11-A addressing is byte-oriented, byte manipulation addressing is straightforward. Byte instructions with autoincrement or autodecrement direct addressing cause the specified register to be modified by one to point to the next byte of data. Byte operations in register mode access the low-order byte of the specified register. These provisions enable the KDJ11-A to perform as either a word or byte processor. The numbering scheme for word and byte addresses in memory is shown in Figure 6-46.

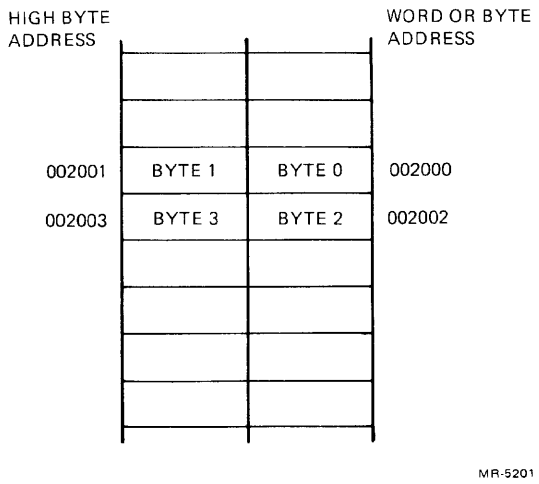


Figure 6-46 Byte Instructions

The most significant bit (bit 15) of the instruction word is set to indicate a byte instruction.

Example:

Symbolic	Octal Code	Instruction Name
CLR	0050DD	Clear word
CLRB	1050DD	Clear byte

6.3.3 List of Instructions

The following is a list of the KDJ11-A instruction set.

SINGLE-OPERAND

General

Mnemonic	Instruction	Op Code
CLR(B)	Clear destination	■050DD
COM(B)	Complement destination	■051DD
INC(B)	Increment destination	■052DD
DEC(B)	Decrement destination	■053DD
NEG(B)	Negate destination	■054DD
TST(B)	Test destination	■057DD
WRTLCK	Read/lock destination, write/unlock R0 into destination	0073DD
TSTSET	Test destination, set low bit	0072DD

Shift and Rotate

Mnemonic	Instruction	Op Code
ASR(B)	Arithmetic shift right	■062DD
ASL(B)	Arithmetic shift left	■063DD
ROR(B)	Rotate right	■060DD
ROL(B)	Rotate left	■061DD
SWAB	Swap bytes	0003DD

Multiple-Precision

Mnemonic	Instruction	Op Code
ADC(B)	Add carry	■055DD
SBC(B)	Subtract carry	■056DD
SXT	Sign extend	0067DD

PS Word Operators

Mnemonic	Instruction	Op Code
MFPS	Move byte from PS	1067DD
MTPS	Move byte to PS	1064SS

DOUBLE-OPERAND

General

Mnemonic	Instruction	Op Code
MOV(B)	Move source to destination	■1SSDD
CMP(B)	Compare source to destination	■2SSDD
ADD	Add source to destination	06SSDD
SUB	Subtract source from destination	16SSDD
ASH	Arithmetic shift	072RSS
ASHC	Arithmetic shift combined	073RSS
MUL	Multiply	070RSS
DIV	Divide	071RSS

Logical

Mnemonic	Instruction	Op Code
BIT(B)	Bit test	■3SSDD
BIC(B)	Bit clear	■4SSDD
BIS(B)	Bit set	■5SSDD
XOR	Exclusive OR	074RDD

PROGRAM CONTROL

Branch

Mnemonic	Instruction	Op Code or Base Code
BR	Branch (unconditional)	000400
BNE	Branch if not equal (to zero)	001000
BEQ	Branch if equal (to zero)	001400
BPL	Branch if plus	100000
BMI	Branch if minus	100400
BVC	Branch if overflow is clear	102000
BVS	Branch if overflow is set	102400
BCC	Branch if carry is clear	103000
BCS	Branch if carry is set	103400

Signed Conditional Branch

Mnemonic	Instruction	Op Code or Base Code
BGE	Branch if greater than or equal (to zero)	002000
BLT	Branch if less than (zero)	002400
BGT	Branch if greater than (zero)	003000
BLE	Branch if less than or equal (to zero)	003400

Unsigned Conditional Branch

Mnemonic	Instruction	Op Code or Base Code
BHI	Branch if higher	101000
BLOS	Branch if lower or same	101400
BHIS	Branch if higher or same	103000
BLO	Branch if lower	103400

Jump and Subroutine

Mnemonic	Instruction	Op Code or Base Code
JMP	Jump	0001DD
JSR	Jump to subroutine	004RDD
RTS	Return from subroutine	00020R
SOB	Subtract one and branch (if $\neq$ 0)	077R00

Trap and Interrupt

Mnemonic	Instruction	Op Code or Base Code
EMT	Emulator trap	104000–104377
TRAP	Trap	104400–104777
BPT	Breakpoint trap	000003
IOT	Input/output trap	000004
RTI	Return from interrupt	000002
RTT	Return from interrupt	000006

Miscellaneous Program Control

Mnemonic	Instruction	Op Code or Base Code
CSM	Call to supervisor mode	0070DD
MARK	Mark	0064NN
SPL	Set Priority Level	00023N

MISCELLANEOUS

Mnemonic	Instruction	Op Code or Base Code
HALT	Halt	000000
WAIT	Wait for interrupt	000001
RESET	Reset external bus	000005
MFPT	Move processor type	000007
MTPD	Move to previous data space	1066SS
MTPI	Move to previous instruction space	0066SS
MFPD	Move from previous data space	0065SS
MFPI	Move from previous instruction space	1065SS

CONDITION CODE OPERATORS

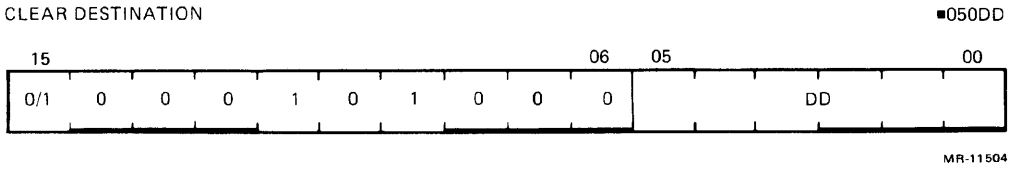
Mnemonic	Instruction	Op Code or Base Code
CLC	Clear C	000241
CLV	Clear V	000242
CLZ	Clear Z	000244
CLN	Clear N	000250
CCC	Clear all CC bits	000257
SEC	Set C	000261
SEV	Set V	000262
SEZ	Set Z	000264
SEN	Set N	000270
SCC	Set all CC bits	000277
NOP	No operation	000240

6.3.4 Single-Operand Instructions

The KDJ11-A instructions that involve only one operand are described in the paragraphs that follow.

6.3.4.1 General –

CLR
CLRB



Operation: (dst) ← 0

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

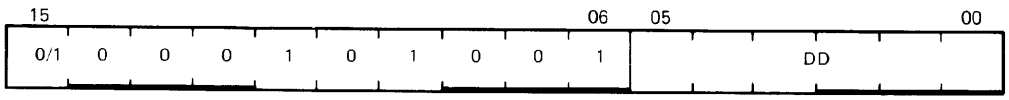
Description: Word: The contents of the specified destination are replaced with 0s.
Byte: Same.

Example: CLR R1

Before	After
(R1) = 177777	(R1) = 000000
N Z V C	N Z V C
1 1 1 1	0 1 0 0

COM
COMB

COMPLEMENT DST ■051DD



MR-11505

Operation: (dst) ← ~ (dst)

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

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

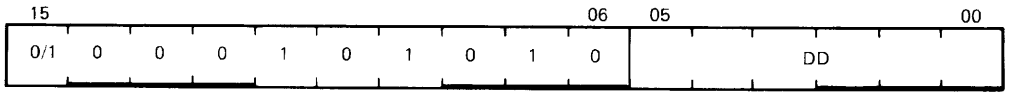
Byte: Same.

Example: COM R0

Before	After
(R0) = 013333	(R0) = 164444
N Z V C	N Z V C
0 1 1 0	1 0 0 1

INC
INCB

INCREMENT DST ■052DD



MR-11506

Operation: (dst) ← (dst) + 1

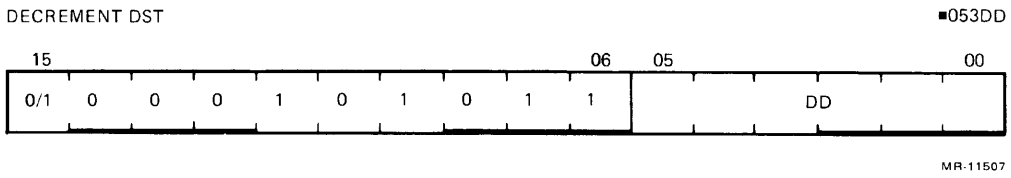
Condition Codes:
N: set if result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) held 077777; cleared otherwise
C: not affected

Description: Word: Add 1 to the contents of the destination.
Byte: Same.

Example: INC R2

Before	After
(R2) = 000333	(R2) = 000334
N Z V C 0 0 0 0	N Z V C 0 0 0 0

DEC
DECB



Operation: (dst) ← (dst) − 1

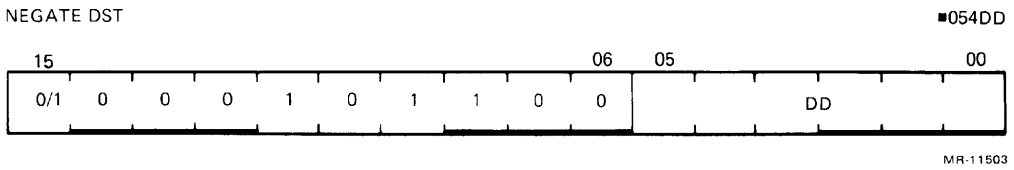
Condition Codes:
N: set if result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) was 100000; cleared otherwise
C: not affected

Description: Word: Subtract 1 from the contents of the destination.
Byte: Same.

Example: DEC R5

Before	After
(R5) = 000001	(R5) = 000000
N Z V C 1 0 0 0	N Z V C 0 1 0 0

NEG
NEGB



Operation: (dst) ← − (dst)

Condition Codes:
N: set if result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if result is 100000; cleared otherwise
C: cleared if result is 0; set otherwise

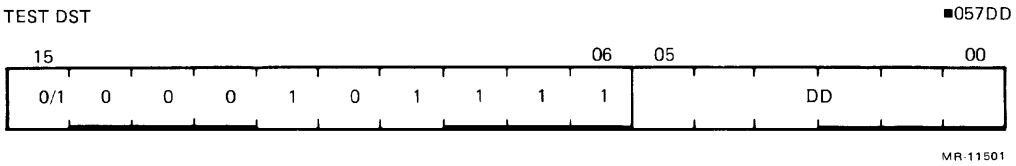
Description: Word: Replaces the contents of the destination address by its 2's complement. Note that 100000 is replaced by itself. (In 2's complement notation the most negative number has no positive counterpart.)

Byte: Same.

Example: NEG R0

Before	After
(R0) = 000010	(R0) = 177770
N Z V C 0 0 0 0	N Z V C 1 0 0 1

TST
TSTB



Operation: (dst) ← (dst)

Condition Codes:
N: set if result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: cleared

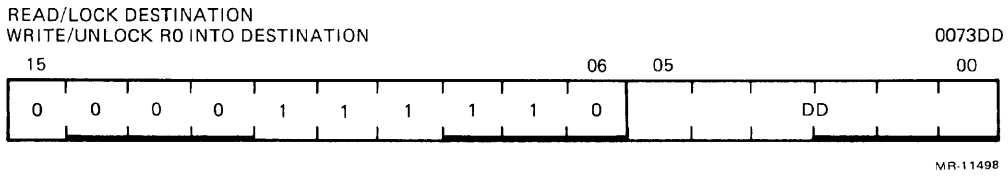
Description: Word: Sets the condition codes N and Z according to the contents of the destination address; the contents of dst remain unmodified.

Byte: Same.

Example:

TST	R1
Before	After
(R1) = 012340	(R1) = 012340
N Z V C 0 0 1 1	N Z V C 0 0 0 0

WRTLCK

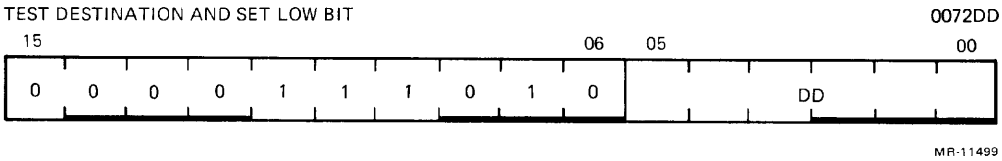


Operation: (dst) ← (R0)

Condition Codes:
N: set if R0 < 0
Z: set if R0 = 0
V: cleared
C: unchanged

Description: Writes contents of R0 into destination using bus lock. If mode is 0, traps to 10.

TSTSET



Operation: (R0) ← (dst), (dst) ← (dst) ∨ 000001 (octal)

Condition Codes:
N: set if R0 < 0
Z: set if R0 = 0
V: cleared
C: gets contents of destination bit 0.

Description: Reads/locks destination word and stores it in R0. Writes/unlocks (R0) ∨ 1 into destination. If mode is 0, traps to 10.

6.3.4.2 Shifts and Rotates – Scaling data by factors of two is accomplished by the shift instructions:

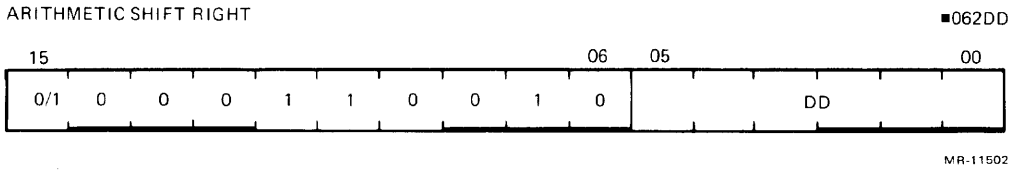
ASR – Arithmetic shift right

ASL – Arithmetic shift left

The sign bit (bit 15) of the operand is reproduced in shifts to the right. The low-order bit is filled with 0s in shifts to the left. Bits shifted out of the C-bit, as shown in the following instructions, are lost.

The rotate instructions operate on the destination word and the C-bit as though they formed a 17-bit "circular buffer." These instructions facilitate sequential bit testing and detailed bit manipulation.

ASR
ASRB



Operation: (dst) ← (dst) shifted one place to the right

Condition Codes:

N: set if high-order bit of result is set (result < 0); cleared otherwise

Z: set if result = 0; cleared otherwise

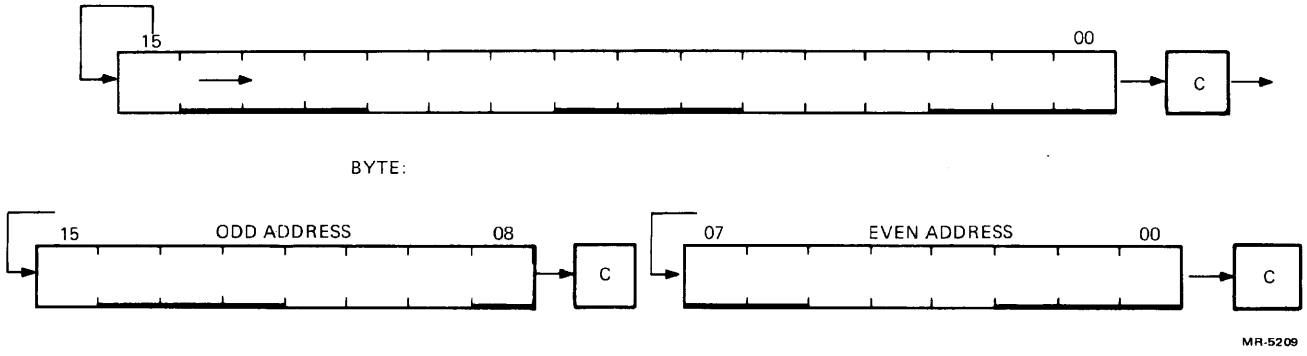
V: loaded from exclusive OR of N-bit and C-bit (as set by the completion of the shift operation)

C: loaded from low-order bit of destination

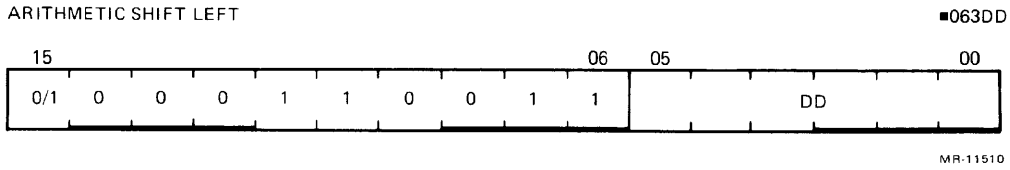
Description: Word: Shifts all bits of the destination right one place. Bit 15 is reproduced. The C-bit is loaded from bit 0 of the destination. ASR performs signed division of the destination by 2.

Byte: Same.

Example:



ASL
ASLB



Operation: (dst) ← (dst) shifted one place to the left

Condition Codes:

N: set if high-order bit of result is set (result < 0); cleared otherwise

Z: set if result = 0; cleared otherwise

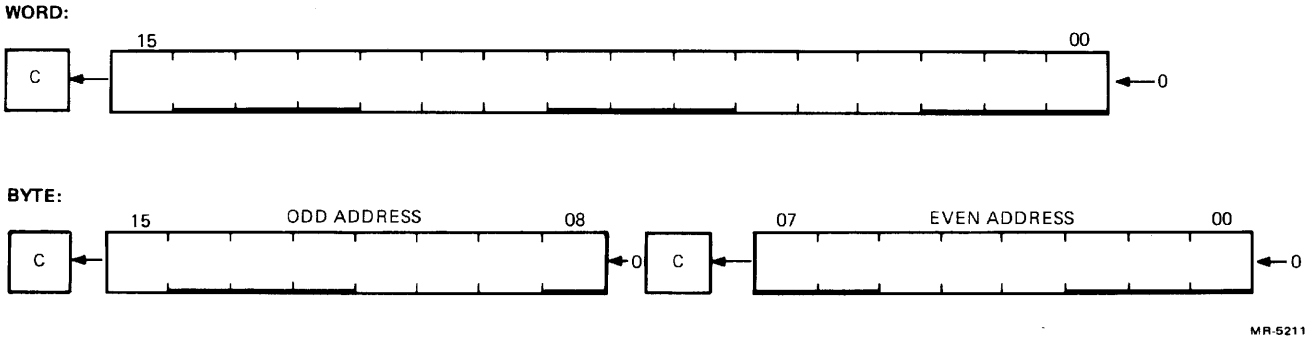
V: loaded with exclusive OR of N-bit and C-bit (as set by the completion of the shift operation)

C: loaded with high-order bit of destination

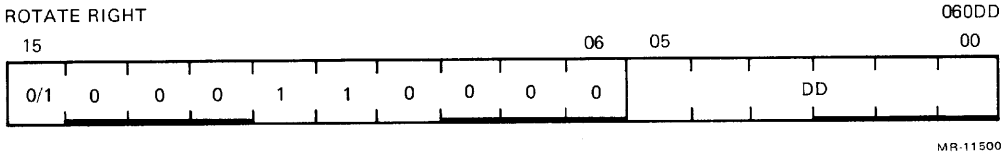
Description: Word: Shifts all bits of the destination left one place. Bit 0 is loaded with a 0. The C-bit of the status word is loaded from the most significant bit of the destination. ASL performs a signed multiplication of the destination by 2 with overflow indication.

Byte: Same.

Example:



ROR
RORB



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

Condition Codes:

N: set if high-order bit of result is set (result < 0); cleared otherwise

Z: set if all bits of result = 0; cleared otherwise

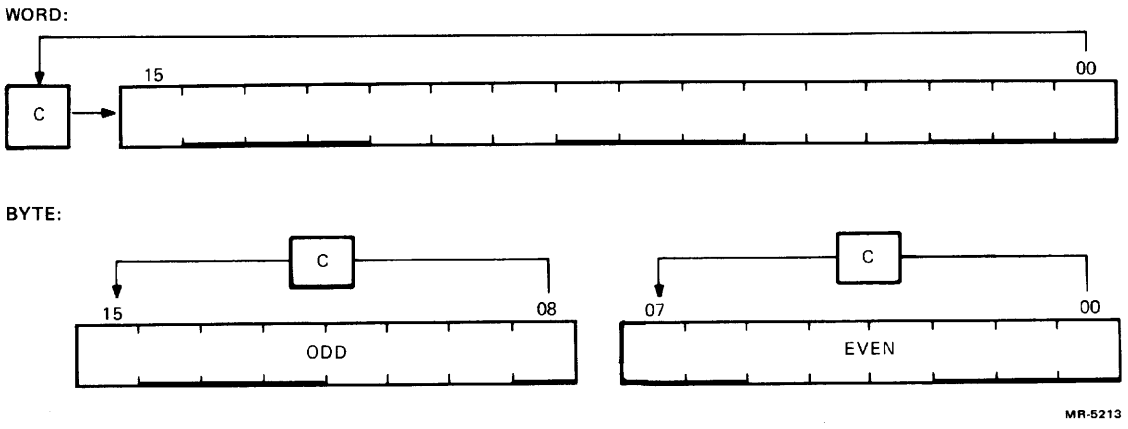
V: loaded with exclusive OR of N-bit and C-bit (as set by the completion of the rotate operation)

C: loaded with low-order bit of destination

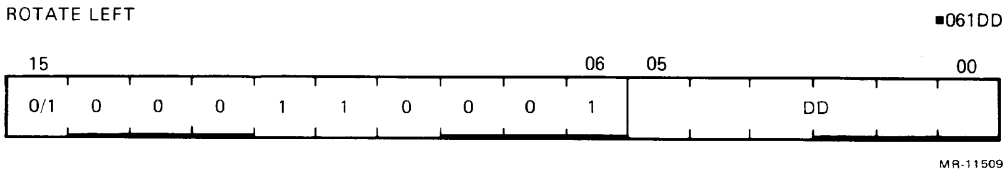
Description: Word: Rotates all bits of the destination right one place. Bit 0 is loaded into the C-bit and the previous contents of the C-bit are loaded into bit 15 of the destination.

Byte: Same.

Example:



ROL
ROLB



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

Condition Codes:

N: set if high-order bit of result word is set (result < 0); cleared otherwise

Z: set if all bits of result word = 0; cleared otherwise

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

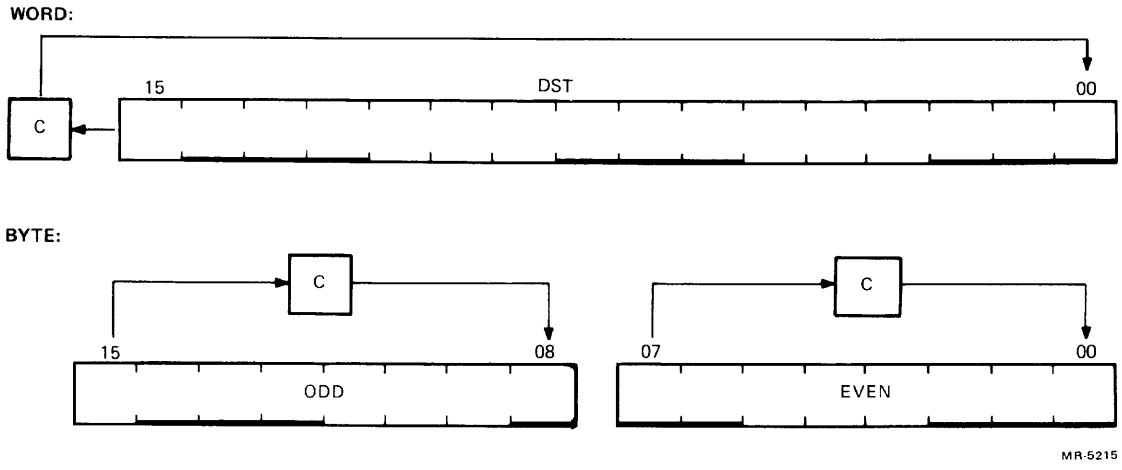
C: loaded with high-order bit of destination

Description:

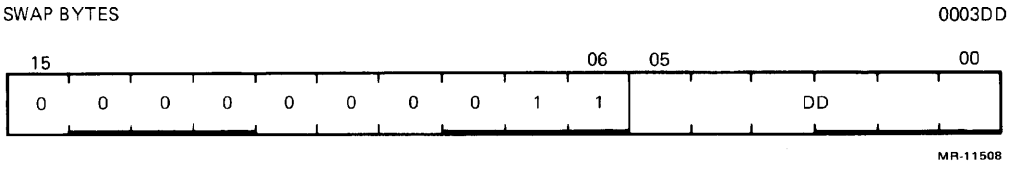
Word: Rotates all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into bit 0 of the destination.

Byte: Same.

Example:



SWAB



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; cleared otherwise

Z: set if low-order byte of result = 0; cleared otherwise

V: cleared

C: cleared

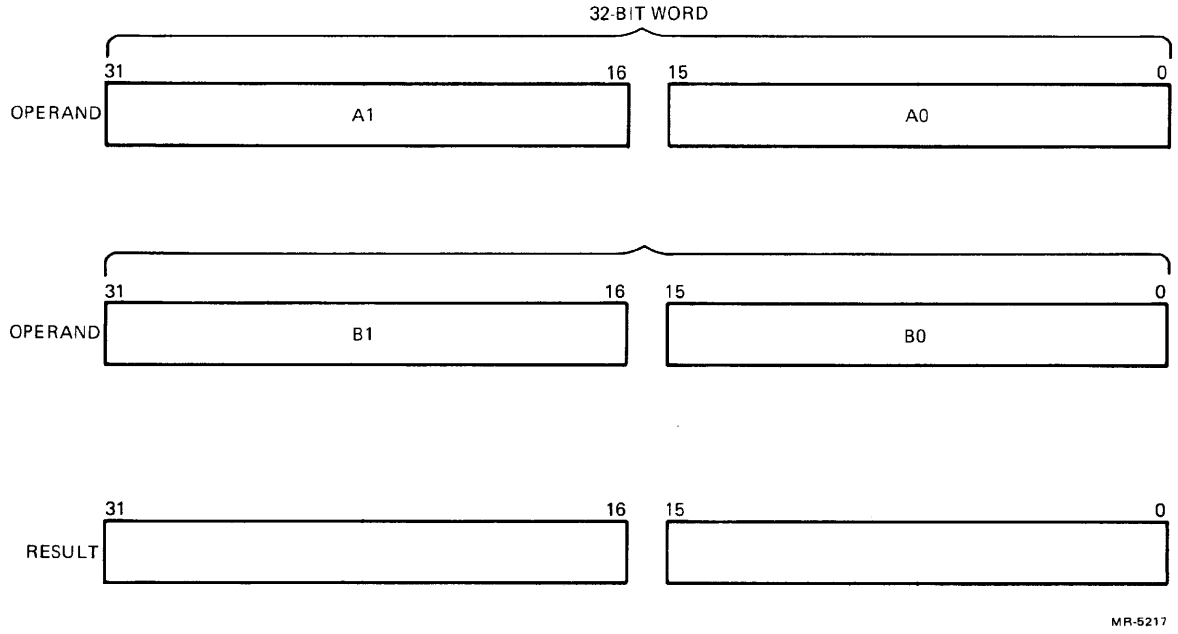
Description: Exchanges high-order byte and low-order byte of the destination word. (The destination must be a word address.)

Example: SWAB R1

Before	After
(R1) = 077777	(R1) = 177577
N Z V C 1 1 1 1	N Z V C 0 0 0 0

6.3.4.3 Multiple-Precision – It is sometimes necessary to do arithmetic operations on operands considered as multiple words or bytes. The KDJ11-A makes special provision for such operations with the instructions ADC (add carry) and SBC (subtract carry) and their byte equivalents.

For example, two 16-bit words may be combined into a 32-bit double-precision word and added or subtracted as shown below.



Example:

The addition of −1 and −1 could be performed as follows.

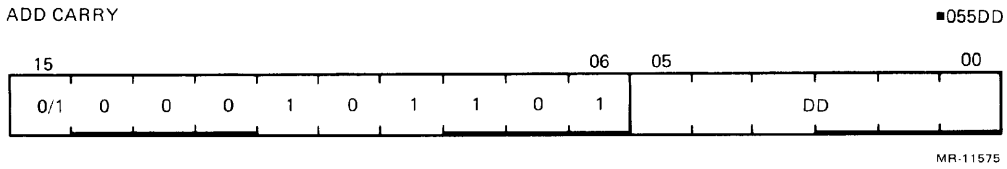
−1 = 37777777777

(R1) = 177777 (R2) = 177777 (R3) = 177777 (R4) = 177777

ADD R1,R2
ADC R3
ADD R4,R3

1. After (R1) and (R2) are added, 1 is loaded into the C-bit.
2. The ADC instruction adds the C-bit to (R3); (R3) = 0.
3. The (R3) and (R4) are added.
4. The result is 37777777776, or −2.

ADC
ADCB



Operation: (dst) ← (dst) + (C-bit)

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

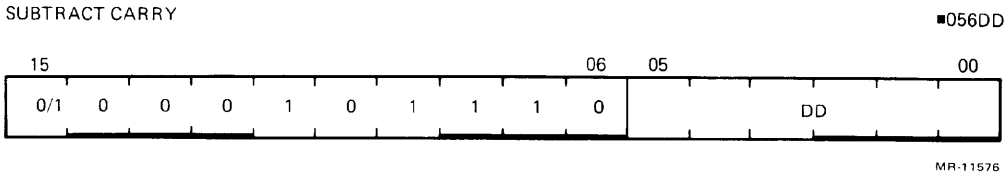
Description: Word: Adds the contents of the C-bit to the destination. This permits the carry from the addition of the low-order words to be carried to the high-order result.

Byte: Same.

Example: Double-precision addition may be done with the following instruction sequence.

```
ADD     A0,B0     ;add low-order parts
ADC     B1        ;add carry into high-order
ADD     A1,B1     ;add high-order parts
```

SBC
SBCB



Operation: (dst) ← (dst) − (C)

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

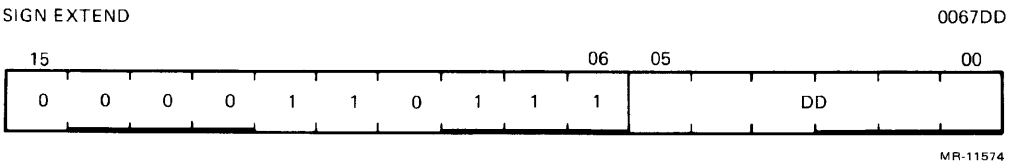
Description: Word: Subtracts the contents of the C-bit from the destination. This permits the carry from the subtraction of two low-order words to be subtracted from the high-order part of the result.

Byte: Same.

Example: Double-precision subtraction is done by:

```
SUB     A0,B0
SBC     B1
SUB     A1,B1
```

SXT



Operation: (dst) ← 0 if N-bit is clear
(dst) ← 1 if N-bit is set

Condition Codes:
N: not affected
Z: set if N-bit is clear
V: cleared
C: not affected

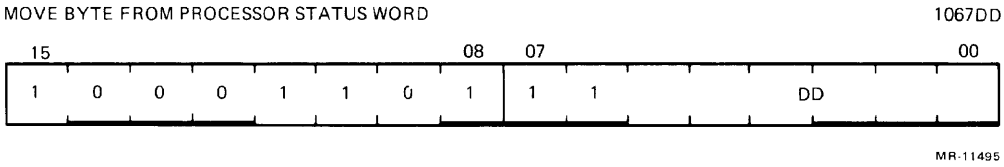
Description: If the condition code bit N is set, a −1 is placed in the destination operand; if the N-bit is clear, a 0 is placed in the destination operand. This instruction is particularly useful in multiple-precision arithmetic because it permits the sign to be extended through multiple words.

Example:

```
SXT A

Before              After

(A) = 012345        (A) = 177777

N Z V C             N Z V C
1 0 0 0             1 0 0 0
```

6.3.4.4 PS Word Operators –

MFPS



Operation: (dst) ← PS
dst lower 8 bits

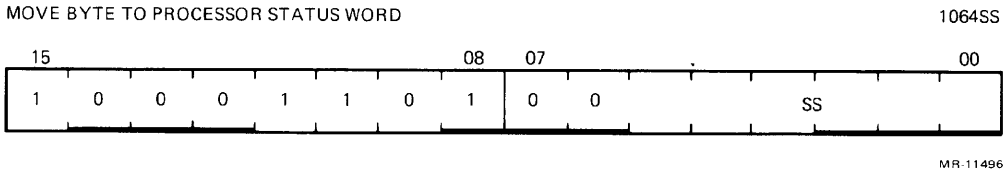
Condition Codes:
N: set if PS <07> = 1; cleared otherwise
Z: set if PS <07:00> = 0; cleared otherwise
V: cleared
C: not affected

Description: The 8-bit contents of the PS are moved to the effective destination. If the destination is mode 0, PS bit 07 is sign-extended through the upper byte of the register. The destination operand address is treated as a byte address.

Example:

```
MFPS R0

Before              After

R0 [0]              R0 [000014]
PS [000014]         PS [000000]
```

MTPS



Operation: PS ← (src)

Condition Codes: Set according to effective SRC operand bits <03:00>

Description: The eight bits of the effective operand replace the current contents of the lower byte of the PS. The source operand address is treated as a byte address. Note: The T-bit (PS bit 04) cannot be set with this instruction. The SRC operand remains unchanged. This instruction can be used to change the priority bits (PS bits <07:05>) in the PS only in kernel mode. If not in kernel mode, PS bits <07:05> cannot be changed.

Example: MTPS R1

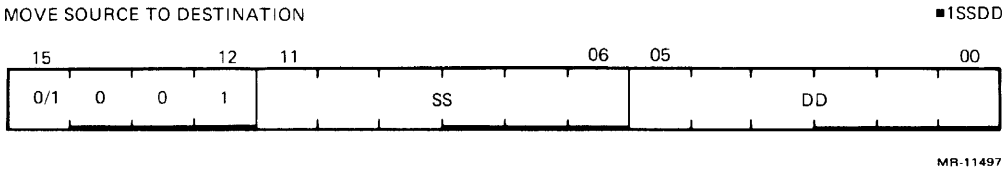
Before	After
(R1) = 000777	(R1) = 000777
(PS) = XXX000	(PS) = XXX357
N Z V C	N Z V C
0 0 0 0	1 1 1 1

6.3.5 Double-Operand Instructions

Double-operand instructions save instructions (and time) since they eliminate the need for "load" and "save" sequences such as those used in accumulator-oriented machines.

6.3.5.1 General –

MOV
MOVB



Operation: (dst) ← (src)

Condition Codes:
N: set if (src) < 0; cleared otherwise
Z: set if (src) = 0; cleared otherwise
V: cleared
C: not affected

Description: Word: Moves the source operand to the destination location. The previous contents of the destination are lost. Contents of the source address are not affected.

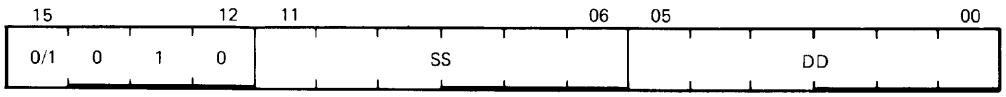
Byte: Same as MOV. The MOVB to a register (unique among byte instructions) extends the most significant bit of the low-order byte (sign extension). Otherwise, MOVB operates on bytes exactly as MOV operates on words.

Example:

MOV XXX,R1	;loads register 1 with the contents of memory location; XXX represents a programmer-defined mnemonic used to represent a memory location
MOV #20,R0	;loads the number 20 into register 0; # indicates that the value 20 is the operand
MOV @#20,–(R6)	;pushes the operand contained in location 20 onto the stack
MOV (R6)+,@#177566	;pops the operand off a stack and moves it into memory location 177566 (terminal print buffer)
MOV R1,R3	;performs an inter-register transfer
MOVB @#177562,@#177566	;moves a character from the terminal keyboard buffer to the terminal printer buffer

CMP
CMPB

COMPARE SRC TO DST ■2SSDD



MR-11562

Operation: (src) − (dst)

Condition Codes:

N: set if result < 0; cleared otherwise

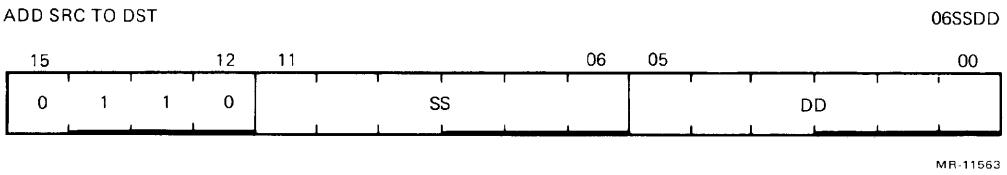
Z: set if result = 0; cleared otherwise

V: set if there was arithmetic overflow; that is, operands were of opposite signs and the sign of the destination was the same as the sign of the result; cleared otherwise

C: cleared if there was a carry from the result's most significant bit; set otherwise

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 not affected. The only action is to set the condition codes. The compare is customarily followed by a conditional branch instruction. Note: Unlike the subtract instruction, the order of operation is (src) − (dst), not (dst) − (src).

ADD



Operation: (dst) ← (src) + (dst)

Condition Codes:

N: set if result < 0; cleared otherwise

Z: set if result = 0; cleared otherwise

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

C: set if there was a carry from the result's most significant bit; cleared otherwise

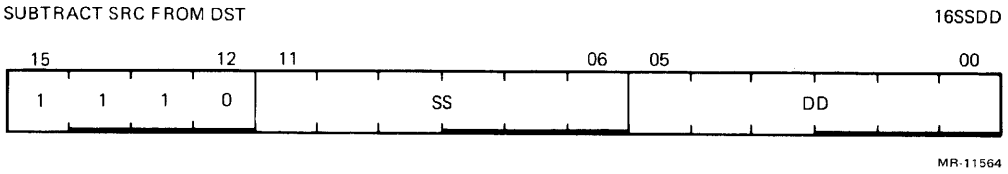
Description: Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. Two's complement addition is performed. Note: There is no equivalent byte mode.

Example:

Add to register:	ADD 20,R0
Add to memory:	ADD R1,XXX
Add register to register:	ADD R1,R2
Add memory to memory:	ADD @#17750,XXX

XXX is a programmer-defined mnemonic for a memory location.

SUB



Operation: (dst) ← (dst) − (src)

Condition Codes:

N: set if result < 0; cleared otherwise

Z: set if result = 0; cleared otherwise

V: set if there was arithmetic overflow as a result of the operation; that is, if operands were of opposite signs and the sign of the source was the same as the sign of the result; cleared otherwise

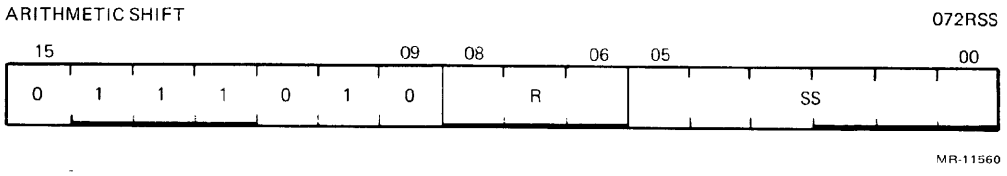
C: cleared if there was a carry from the result's most significant bit; set otherwise

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. In double-precision arithmetic the C-bit, when set, indicates a "borrow." Note: There is no equivalent byte mode.

Example: SUB R1,R2

Before	After
(R1) = 011111	(R1) = 011111
(R2) = 012345	(R2) = 001234
N Z V C	N Z V C
1 1 1 1	0 0 0 0

ASH

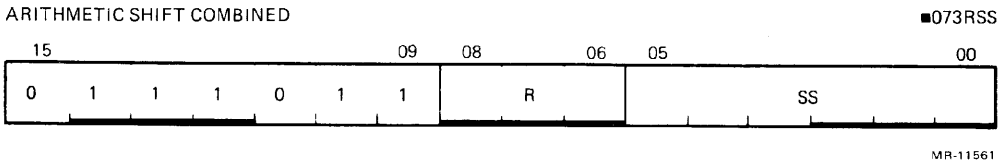


Operation: R ← R shifted arithmetically NN places to the right or left where NN = (src)

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



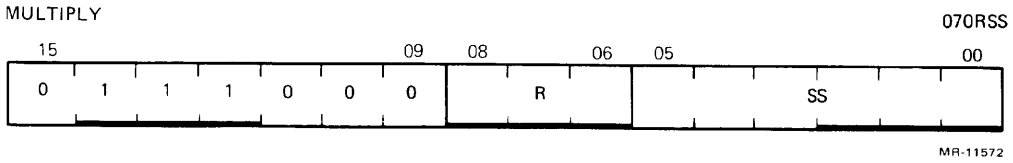
Operation: R, R ∨ 1 ← R, R ∨ 1
The double word is shifted NN places to the right or left where NN = (src)

Condition Codes:
- N: set if result < 0
- Z: set if result = 0
- V: set if sign bit changes during 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 <15:00>) and R (bits <31:16>) 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 times specified by the shift count.

MUL

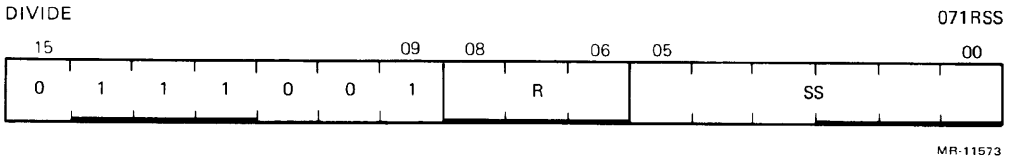


Operation: R, R ∨ 1 ← R × (src)

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.

DIV



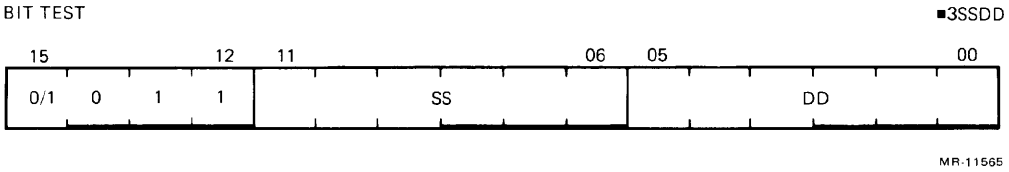
Operation: R, R ∨ 1 ← R, R ∨ 1/(src)

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 zero is 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.

6.3.5.2 Logical – These instructions have the same format as those in the double-operand arithmetic group. They permit operations on data at the bit level.

BIT
BITB



Operation: (src) ∧ (dst)

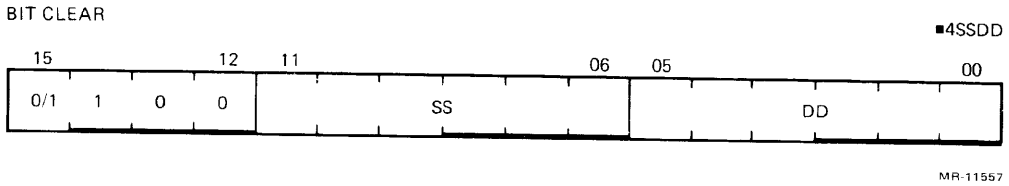
Condition Codes:
N: set if high-order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

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

Example:

```
BIT #30,R3          ;test bits three and four of R3 to see if
                    ;both are off.

R3 = 0 000 000 000 011 000

Before              After

N Z V C             N Z V C
1 1 1 1             0 0 0 1
```

BIC
BICB



Operation: (dst) ← ~(src) ∧ (dst)

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

Description: Clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are not affected.

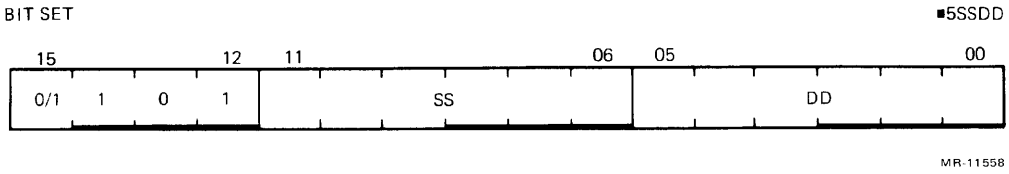
Example: BIC R3,R4

Before	After
(R3) = 001234	(R3) = 001234
(R4) = 001111	(R4) = 000101
N Z V C	N Z V C
1 1 1 1	0 0 0 1

Before: (R3) = 0 000 001 010 011 100
(R4) = 0 000 001 001 001 001

After: (R4) = 0 000 000 001 000 001

BIS
BISB



Operation: (dst) ← (src) ∨ (dst)

Condition Codes:
N: set if high-order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
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; that is, corresponding bits set in the source are set in the destination. The contents of the destination are lost.

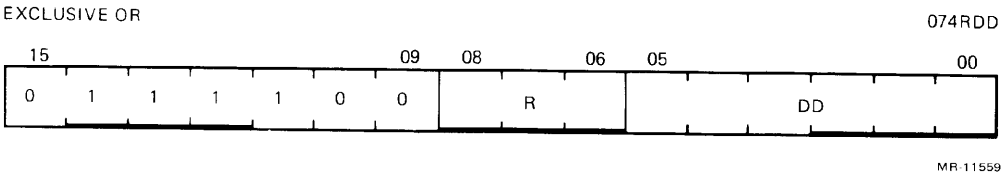
Example: BIS R0,R1

Before	After
(R0) = 001234	(R0) = 001234
(R1) = 001111	(R1) = 001335
N Z V C	N Z V C
0 0 0 0	0 0 0 0

Before: (R0) = 0 000 001 010 011 100
(R1) = 0 000 001 001 001 001

After: (R1) = 0 000 001 011 011 101

XOR



Operation: (dst) ← (reg) ∀ (dst)

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: The exclusive OR of the register and destination operand is stored in the destination address. The contents of the register are not affected. The assembler format is XOR R,D.

Example: XOR R0,R2

Before	After
(R0) = 001234	(R0) = 001234
(R2) = 001111	(R2) = 000325
N Z V C	N Z V C
1 1 1 1	0 0 0 1

Before: (R0) = 0 000 001 010 011 100
(R2) = 0 000 001 001 001 001

After: (R2) = 0 000 000 011 010 101

6.3.6 Program Control Instructions

The following paragraphs describe the KDJ11-A instructions that affect program control.

6.3.6.1 Branches – These instructions cause a branch to a location defined by the sum of the offset (multiplied by 2) and the current contents of the program counter if:

1. The branch instruction is unconditional.
2. It is conditional and the conditions are met after testing the condition codes (NZVC).

The offset is the number of words from the current contents of the PC, forward or backward. Note that the current contents of the PC point to the word following the branch instruction.

Although the offset expresses a byte address, the PC is expressed in words. The offset is automatically multiplied by 2 and sign-extended to express words before it is added to the PC. Bit 7 is the sign of the offset. If it is set, the offset is negative and the branch is done in the backward direction. If it is not set, the offset is positive and the branch is done in the forward direction.

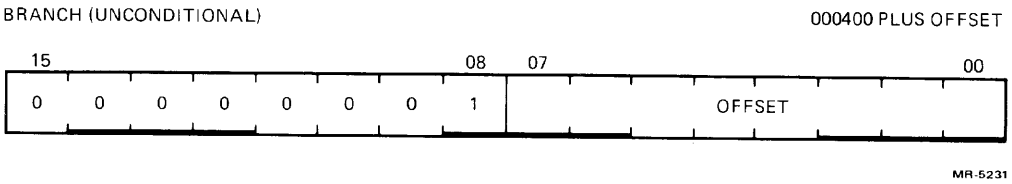
The 8-bit offset allows branching in the backward direction by 200 (octal) words (400 octal bytes) from the current PC, and in the forward direction by 177 (octal) words (376 octal bytes) from the current PC.

The KDJ11-A assembler typically handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form:

Bxx loc

Bxx is the branch instruction and loc is the address to which the branch is to be made. The assembler gives an error indication in the instruction if the permissible branch range is exceeded. Branch instructions have no effect on condition codes. Conditional branch instructions where the branch condition is not met are treated as NOPs.

BR



Operation: PC ← PC + (2 × offset)

Condition Codes: Not affected

Description: Provides a way of transferring program control within a range of −128 to +127 words with a one word instruction.

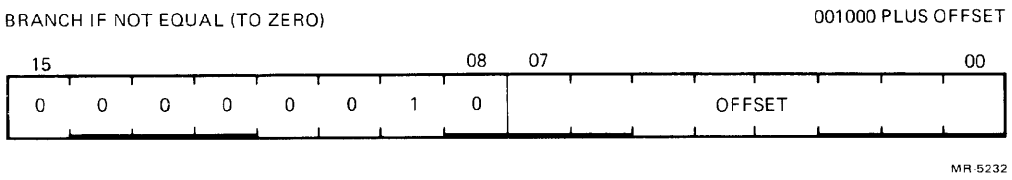
New PC address = updated PC + (2 × offset)

Updated PC = address of branch instruction +2

Example: With the branch instruction at location 500, the following offsets apply.

New PC Address	Offset Code	Offset (decimal)
474	375	−3
476	376	−2
500	377	−1
502	000	0
504	001	+1
506	002	+2

BNE



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

Condition Codes: Not affected

Description: Tests the state of the Z-bit and causes a branch if the Z-bit is clear. BNE is the complementary operation of BEQ. It is used to test: (1) inequality following a CMP, (2) that some bits set in the destination were also in the source following a BIT operation, and (3) generally, that the result of the previous operation was not 0.

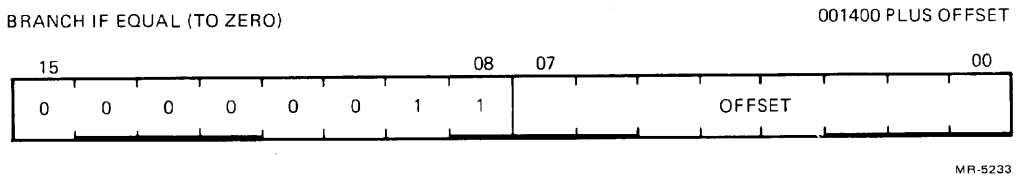
Example: Branch to C if A ≠ B

```
CMP A,B          ;compare A and B
BNE C            ;branch if they are not equal
```

Branch to C if A + B ≠ 0

```
ADD A,B          ;add A to B
BNE C            ;branch if the result is not
                 equal to 0
```

BEQ



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

Condition Codes: Not affected

Description: Tests the state of the Z-bit and causes a branch if Z is set. It is used to test: (1) equality following a CMP operation, (2) that no bits set in the destination were also set in the source following a BIT operation, and (3) generally, that the result of the previous operation was 0.

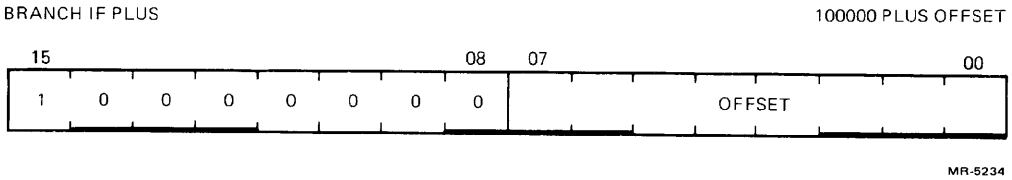
Example: Branch to C if A = B (A − B = 0)

```
CMP A,B          ;compare A and B
BEQ C            ;branch if they are equal
```

Branch to C if A + B = 0

```
ADD A,B          ;add A to B
BEQ C            ;branch if the result = 0
```

BPL

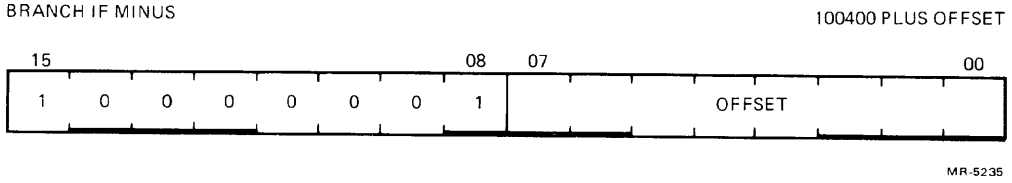


Operation: PC ← PC + (2 × offset) if N = 0

Condition Codes: Not affected

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

BMI

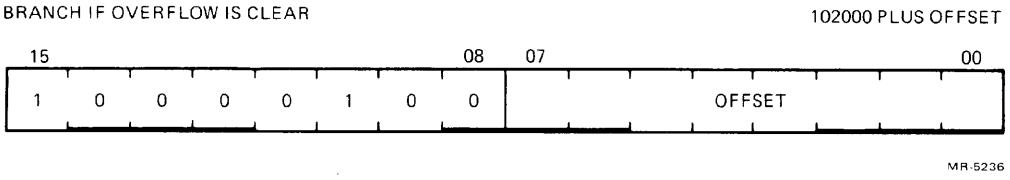


Operation: PC ← PC + (2 × offset) if N = 1

Condition Codes: Not affected

Description: Tests the state of the N-bit and causes a branch if N is set. It is used to test the sign (most significant bit) of the result of the previous operation), branching if negative. BMI is the complementary function of BPL.

BVC

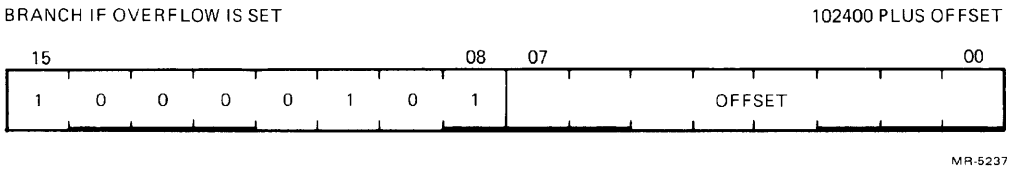


Operation: PC ← PC + (2 × offset) if V = 0

Condition Codes: Not affected

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

BVS

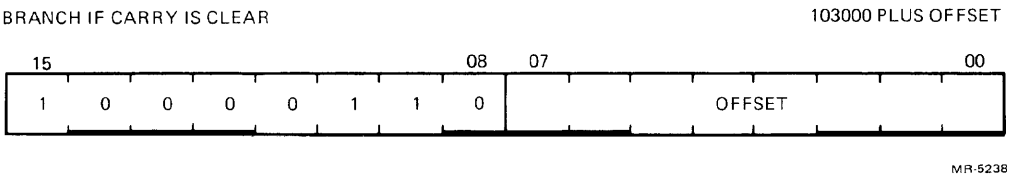


Operation: PC ← PC + (2 × offset) if V = 1

Condition Codes: Not affected

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

BCC

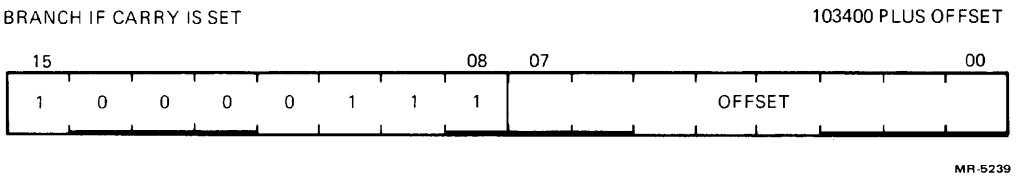


Operation: PC ← PC + (2 × offset) if C = 0

Condition Codes: Not affected

Description: Tests the state of the C-bit and causes a branch if C is clear. BCC is the complementary operation of BCS.

BCS



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

Condition Codes: Not affected

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

6.3.6.2 Signed Conditional Branches – Particular combinations of the condition code bits are tested with the signed conditional branches. These instructions are used to test the results of instructions in which the operands were considered as signed (2's complement) values.

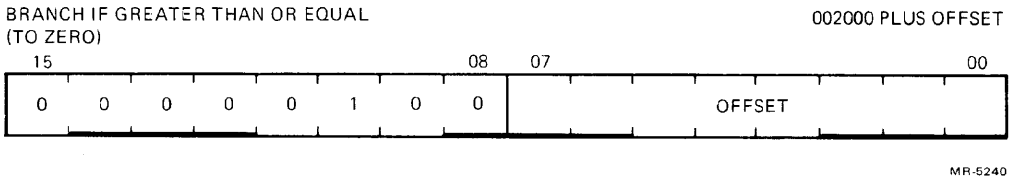
Note that the sense of signed comparisons differs from that of unsigned comparisons in that in signed, 16-bit, 2's complement arithmetic the sequence of values is as follows.

largest	077777
positive	077776
	.
	.
	.
	000001
	000000
	177777
	177776
	.
	.
	.
smallest	100001
negative	100000

Whereas, in unsigned, 16-bit arithmetic, the sequence is considered to be:

highest	177777
	.
	.
	.
	.
	.
	.
	000002
	000001
lowest	000000

BGE

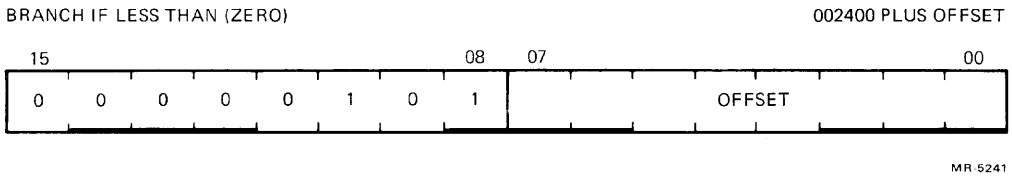


Operation: PC ← PC + (2 × offset) if N ∀ V = 0

Condition Codes: Not affected

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

BLT

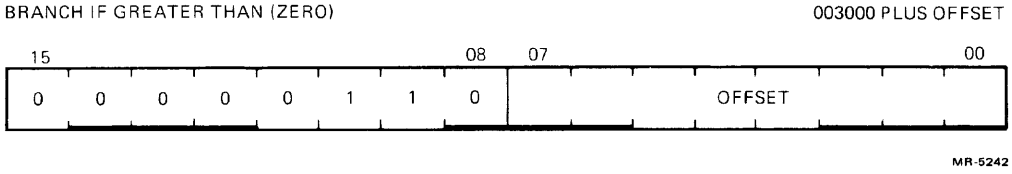


Operation: PC ← PC + (2 × offset) if N ∀ V = 1

Condition Codes: Not affected

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

BGT

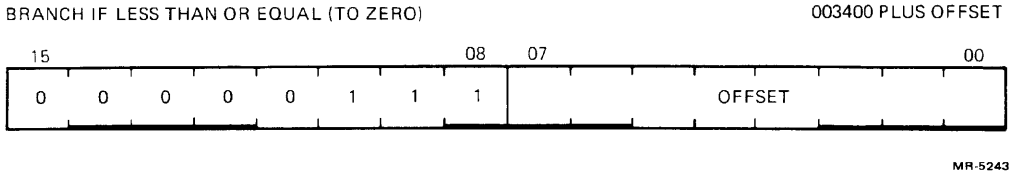


Operation: PC ← PC + (2 × offset) if Z ∨ (N ∀ V) = 0

Condition Codes: Not affected

Description: Operation of BGT is similar to BGE, except that BGT will not cause a branch on a 0 result.

BLE



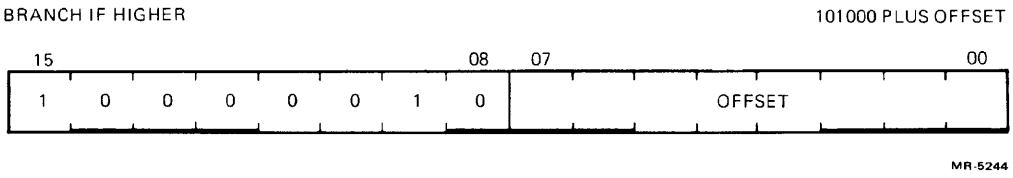
Operation: PC ← PC + (2 × offset) if Z ∨ (N ∀ V) = 1

Condition Codes: Not affected

Description: Operation is similar to BLT, but in addition will cause a branch if the result of the previous operation was 0.

6.3.6.3 Unsigned Conditional Branches – The unsigned conditional branches provide a means for testing the result of comparison operations in which the operands are considered as unsigned values.

BHI



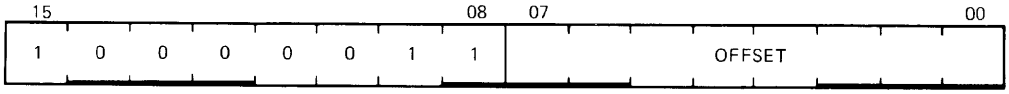
Operation: PC ← PC + (2 × offset) if C = 0 and Z = 0

Condition Codes: Not affected

Description: Causes a branch if the previous operation caused 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.

BLOS

BRANCH IF LOWER OR SAME — 101400 PLUS OFFSET



MR-5245

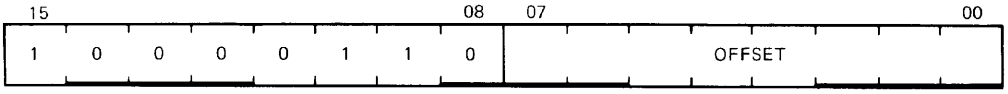
Operation: PC ← PC + (2 × offset) if C ∨ Z = 1

Condition Codes: Not affected

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

BHIS

BRANCH IF HIGHER OR SAME — 103000 PLUS OFFSET



MR-5246

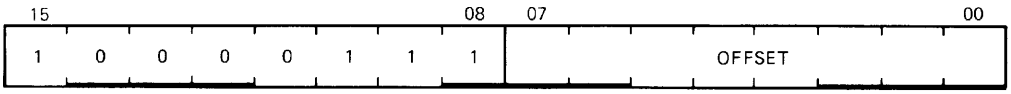
Operation: PC ← PC + (2 × offset) if C = 0

Condition Codes: Not affected

Description: BHIS is the same instruction as BCC. This mnemonic is included for convenience only.

BLO

BRANCH IF LOWER — 103400 PLUS OFFSET



MR-5247

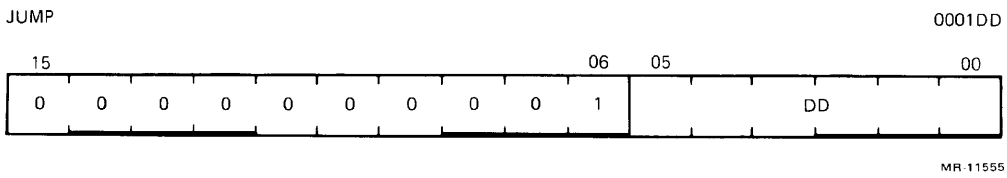
Operation: PC ← PC + (2 × offset) if C = 1

Condition Codes: Not affected

Description: BLO is the same instruction as BCS. This mnemonic is included for convenience only.

6.3.6.4 Jump and Subroutine Instructions – The subroutine call in the KDJ11-A provides for automatic nesting of subroutines, reentrancy, and multiple entry points. Subroutines may call other subroutines (or indeed themselves) to any level of nesting without making special provision for storage of return addresses at each level of subroutine call. The subroutine calling mechanism does not modify any fixed location in memory, and thus provides for reentrancy. This allows one copy of a subroutine to be shared among several interrupting processes.

JMP



Operation: PC ← (dst)

Condition Codes: Not affected

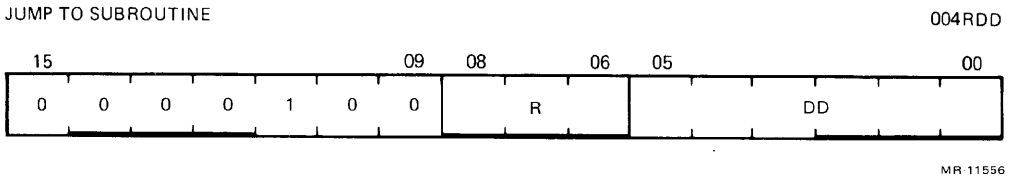
Description: JMP provides more flexible program branching than the branch instructions do. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes, with the exception of register mode 0. Execution of a jump with mode 0 will cause an "illegal instruction" condition, and will cause the CPU to trap to vector address ten. (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 that instructions are word data and must therefore be fetched from an even-numbered address.

Deferred-index mode JMP instructions permit transfer of control to the address contained in a selectable element of a table of dispatch vectors.

Example:

```
First:

JMP FIRST           ;transfers to FIRST
. . . . .
. . . . .
JMP @LIST           ;transfers to location
                    pointed to at LIST
. . . . .

List:

FIRST               ;pointer to FIRST

JMP @(SP)+          ;transfer to location
                    pointed to by the top of
                    the stack, and remove the
                    pointer from the stack
```

JSR



Operation:

(tmp) ← (dst) (tmp is an internal processor register)

↓ (SP) ← reg (Push register contents onto processor stack)

reg ← PC (PC holds location following JSR; this address now put in register)

PC ← (dst) (PC now points to subroutine destination)

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 is 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. The same subroutine may be 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.

A subroutine called with a JSR reg,dst instruction can access the arguments following the call with either autoincrement addressing, (reg) +, if arguments are accessed sequentially, or by indexed addressing, X(reg), if accessed in random order. These addressing modes may also be deferred, @(reg)+ and @X(reg), if the parameters are operand addresses rather than the operands themselves.

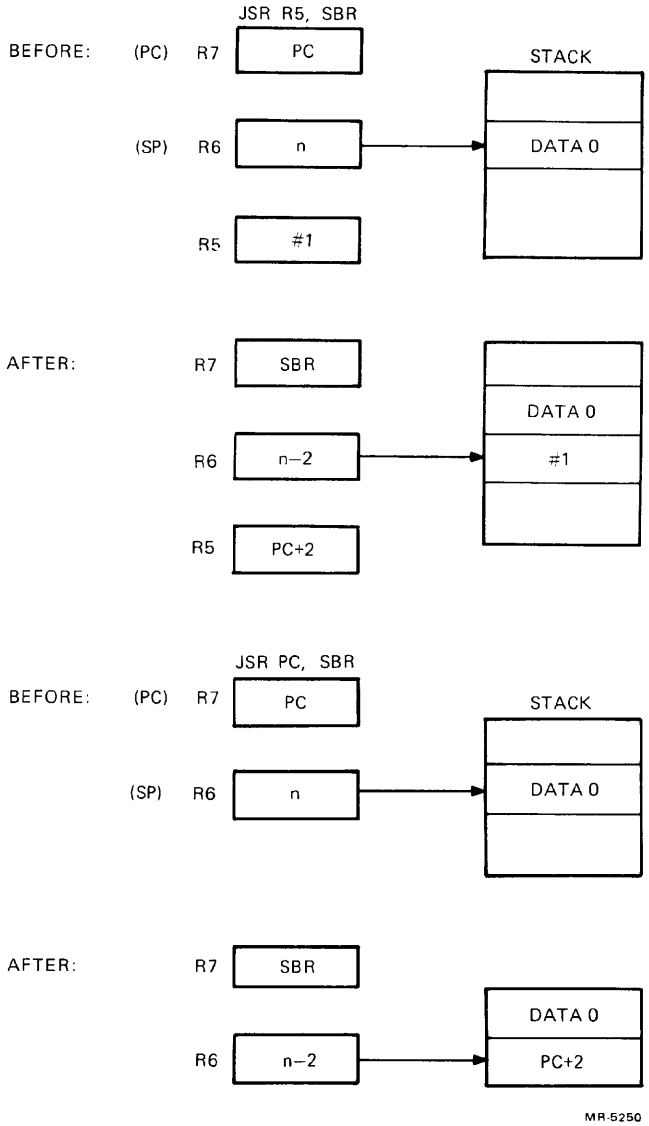
JSR PC, dst is a special case of the KDJ11-A subroutine call suitable for subroutine calls that transmit parameters through the general registers. The SP and the PC are the only registers that may be modified by this call.

Another special case of the JSR instruction is JSR PC,@(SP) +, which exchanges the top element of the processor stack with the contents of the program counter. This instruction allows two routines to swap program control and resume operation from where they left off when they are recalled. Such routines are called *coroutines*.

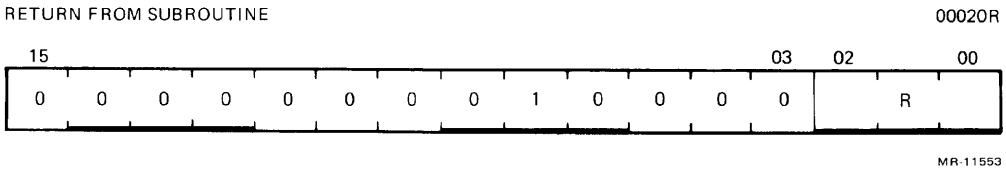
Return from a subroutine is done by the RTS instruction. RTS reg loads the contents of reg into the PC and pops the top element of the processor stack into the specified register.

Example:

		R5	R6	R7
SBCALL:	JSR R5,SBR	#1	n	SBCALL
SBCALL+4:	ARG 1			
	ARG 2			
	.			
	.			
SBCALL+2+2M:	ARG M			
CONT:	Next Instruction	#1	n	CONT
	.			
	.			
SBR:	MOV (R5)+,dst 1	SBCALL+4	n−2	SBR
	MOV (R5)+,dst 2			
	.			
	MOV (R5)+,dst M	SBCALL+2+2M		
	Other Instructions	CONT		
EXIT:	RTS R5	CONT	n−2	EXIT



RTS



Operation:

PC ← (reg)
(reg) ← (SP) ↑

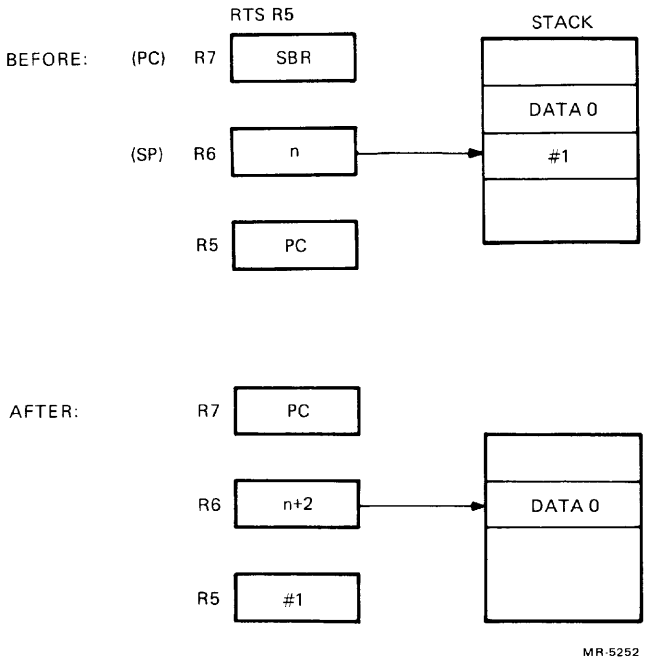
Description:

Loads the contents of the register into PC and pops the top element of the processor stack into the specified register.

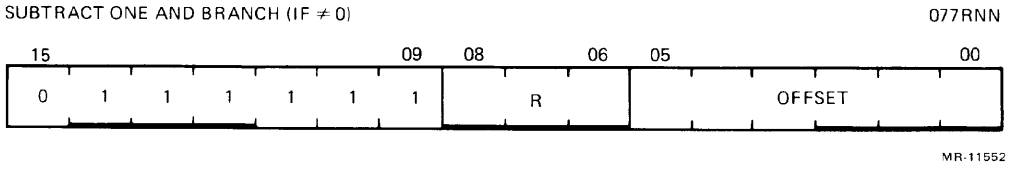
Return from a nonreentrant subroutine is typically made through the same register that was used in its call. Thus, a subroutine called with a JSR PC, dst exits with a RTS PC and a subroutine called with a JSR R5, dst, may pick up parameters with addressing modes (R5) +, X(R5), or @X(R5) and finally exits, with an RTS R5.

Example:

RTS R5



SOB



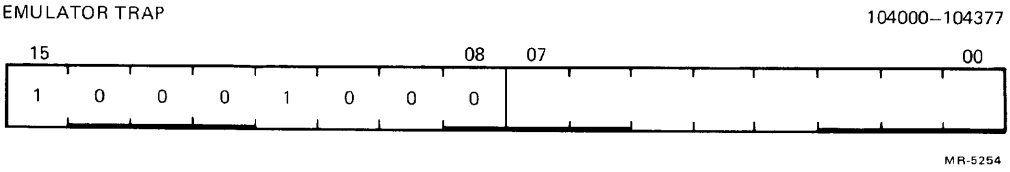
Operation: (R) ← (R) − 1; if this result ≠ 0, then PC ←PC − (2 × offset); if (R) = 0 then PC ← PC

Condition Codes: Not affected

Description: The register is decremented. If the contents does not equal 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. The 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: the SOB instruction cannot be used to transfer control in the forward direction.

6.3.6.5 Traps – Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters. A trap is effectively an interrupt generated by software. When a trap occurs, the contents of the current program counter (PC) and processor 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 instruction vectors are located at permanently assigned fixed addresses.

EMT



Operation:

↓ (SP) ← PS
↓ (SP) ← PC
PC ← (30)
PS ← (32)

Condition Codes:

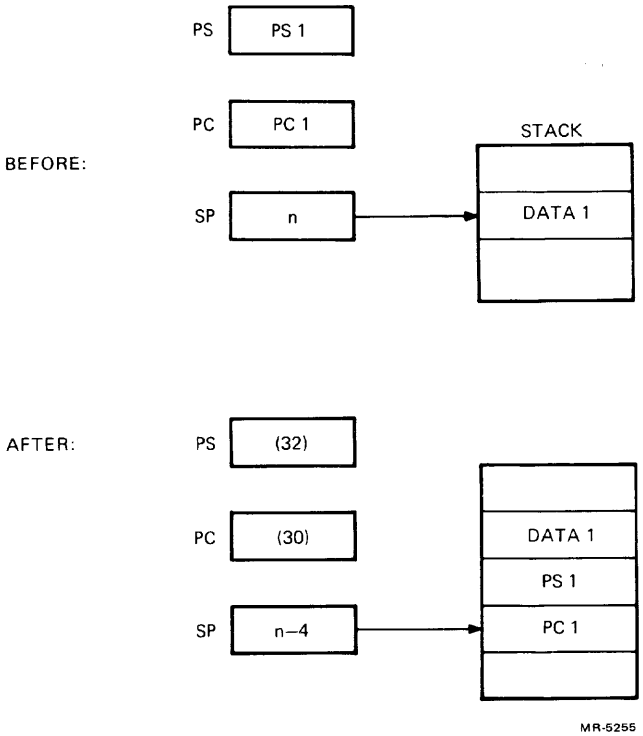
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

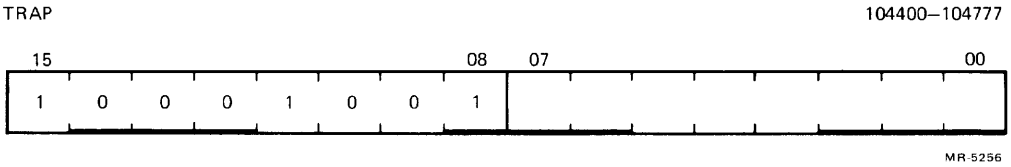
All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit information to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new processor status (PS) is taken from the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

NOTE

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



TRAP



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

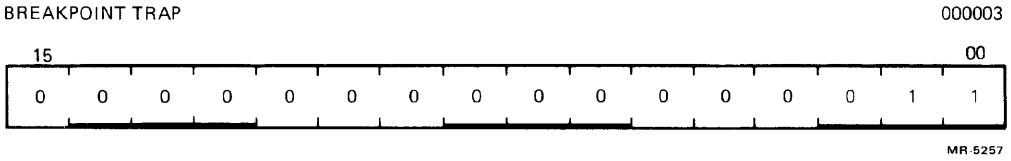
Description:

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.

BPT



Operation:

↓ (SP) ← PS
↓ (SP) ← PC
PC ← (14)
PS ← (16)

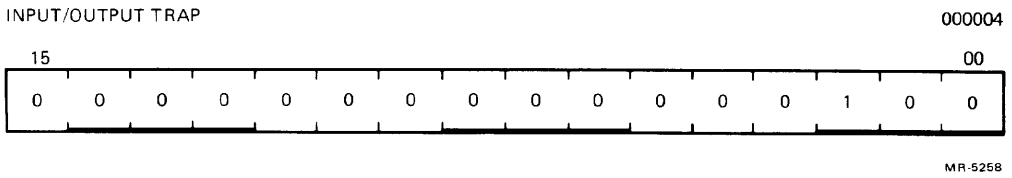
Condition Codes:

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.)

IOT



Operation:

↓ (SP) ← PS
↓ (SP) ← PC
PC ← (20)
PS ← (22)

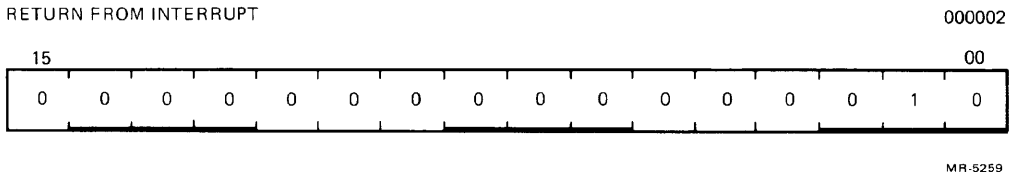
Condition Codes:

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. (No information is transmitted in the low byte.)

RTI



Operation:

PC ← (SP) ↑
PS ← (SP) ↑

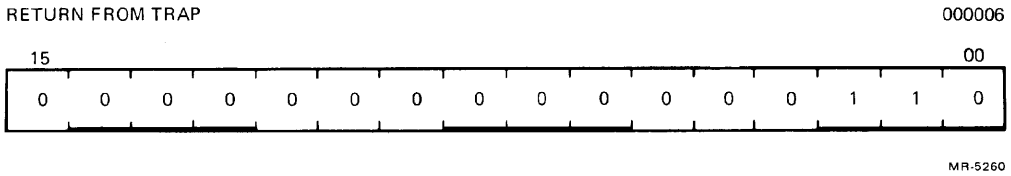
Condition Codes:

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. When executed in supervisor mode, the current and previous mode bits in the restored PS cannot be kernel. When executed in user mode, the current and previous mode bits in the restored PS can only be user. RTI cannot clear PS bit 11 if it was already set.

RTT



Operation:

PC ← (SP)↑
PS ← (SP)↑

Condition Codes:

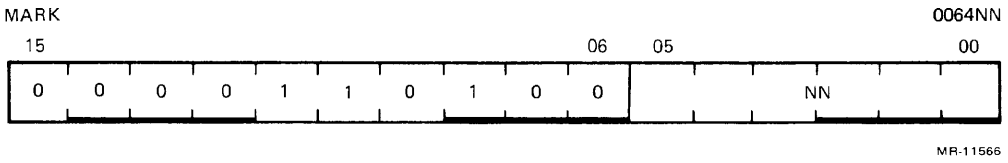
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description:

Operation is the same as RTI except that it inhibits a trace trap, whereas RTI permits a trace trap. If the new PS has the T-bit set, a trap will occur after execution of the first instruction after RTT. When executed in supervisor mode, the current and previous mode bits in the restored PS cannot be kernel. When executed in user mode, the current and previous mode bits in the restored PS can only be user. RTT cannot clear PS bit 11 if it was already set.

6.3.6.6 Miscellaneous Program Control –

MARK



Operation:

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

Condition Codes:

N: unaffected
Z: unaffected
V: unaffected
C: unaffected

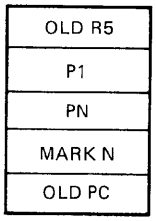
Description:

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.

Example:

```
MOV R5,–(SP)          ;place old R5 on stack
MOV P1,–(SP)          ;place N parameters on
MOV P2,–(SP)          ;the stack to be used
                      ;there by the subroutine
MOV PN,–(SP)
MOV =MARKN,–(SP)      ;place the instruction
                      ;MARK N on the stack
MOV SP,R5             ;set up address at MARK N
                      ;instruction
JSR PC,SUB            ;jump to subroutine
```

At this point the stack is as follows



MR-11569

The program is at the address SUB which is the beginning of the subroutine.

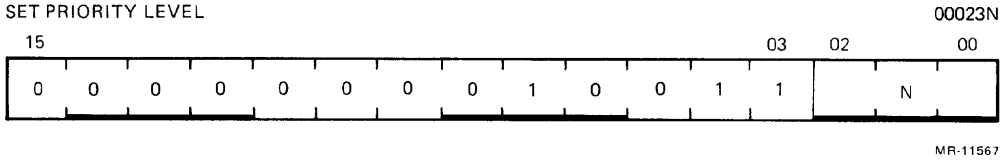
```
SUB:                  ;execution of the
                      ;subroutine itself

RTS R5                ;the return begins:
                      ;this causes the contents
                      ;of R5 to be placed in the
                      ;PC which then results in
                      ;the execution of the
                      ;instruction MARK N. The
                      ;contents of the old PC
                      ;are placed in R5.
```

MARK N causes: (1) the stack pointer to be adjusted to point to the old R5 value; (2) the value now in R5 (the old PC) to be placed in the PC; and (3) the contents of the old R5 to be popped into R5, thus completing the return from the subroutine.

NOTE

If memory management is in use, the stack must be mapped through both I and D space to execute the MARK instruction.

SPL



Operation:

PS bits <07:05> ← priority
(priority = N)

Condition Codes:

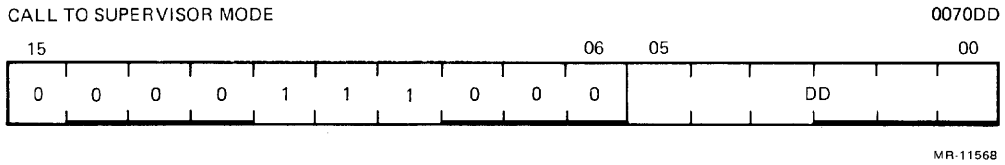
N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description:

In kernel mode, the least significant three bits of the instruction are loaded into the processor status word (PS) bits <07:05>, thus causing a changed priority. The old priority is lost. In user or supervisor modes, SPL executes as a NOP.

Assembler syntax is: SPL N

CSM



Operation:

If MMR3 bit 3 = 1 and current
mode = kernel then
supervisor SP ← current mode SP
temp<15:04> ← PS<15:04>
temp<03:00> ← 0
PS<13:12> ← PS<15:14>
PS<15:14> ← 01
PS 4 ← 0
–(SP) ← temp
–(SP) ← PC
–(SP) ← (dst)
PC ← (10)
otherwise, traps to 10 in kernel mode.

Condition Codes:

N: unaffected
Z: unaffected
V: unaffected
C: unaffected

Description: CSM may be executed in user or supervisor mode, but is an illegal instruction in kernel mode. CSM copies the current stack pointer (SP) to the supervisor mode, switches to supervisor mode, stacks three words on the supervisor stack (the PS with the condition codes cleared, the PC, and the argument word addressed by the operand), and sets the PC to the contents of location 10 (in supervisor space). The called program in supervisor space may return to the calling program by popping the argument word from the stack and executing RTI. On return, the condition codes are determined by the PS word on the stack. Hence, the called program in supervisor space may control the condition code values following return.

6.3.6.7 Reserved Instruction Traps – These are caused by attempts to execute instruction codes reserved for future processor expansion (reserved instructions) or instructions with illegal addressing modes (illegal instructions). Order codes not corresponding to any of the instructions described are considered to be reserved instructions. JMP and JSR with register mode destinations are illegal instructions; they trap to virtual address 4 in kernel data space. Reserved instructions trap to vector address 10 in kernel data space.

6.3.6.8 Trace Trap – Trace trap is enabled by bit 4 of the PS and causes processor traps at the end of instruction execution. The instruction that is executed after the instruction that set the T-bit will proceed to completion and then trap through the trap vector at address 14. Note that the trace trap is a system debugging aid and is transparent to the general programmer.

NOTE

Bit 4 of the PS can only be set indirectly by executing a RTI or RTT instruction with the desired PS on the stack.

The following are special cases of the T-bit.

NOTE

The traced instruction is the instruction after the one that set the T-bit.

1. An instruction that cleared the T-bit – Upon fetching the traced instruction, an internal flag, the trace flag, was set. The trap will still occur at the end of this instruction's execution. The status word on the stack, however, will have a clear T-bit.

2. An instruction that set the T-bit – Since the T-bit was already set, setting it again has no effect. The trap will occur.

3. An instruction that caused an instruction trap – The instruction trap is performed and the entire routine for the service trap is executed. If the service routine exits with an RTI, or in any other way restores the stacked status word, the T-bit is set again, the instruction following the traced instruction is executed, and, unless it is one of the special cases noted previously, a trace trap occurs.

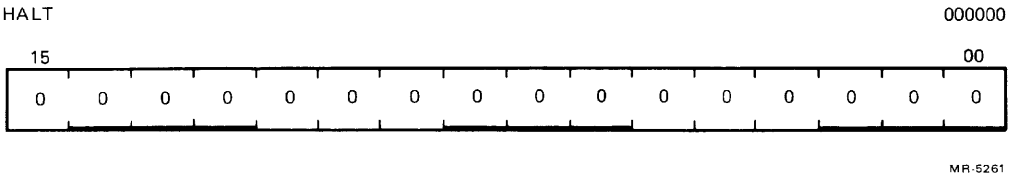
4. An instruction that caused a stack overflow – The instruction completes execution as usual. The stack overflow does not cause a trap. The trace trap vector is loaded into the PC and PS and the old PC and PS are pushed onto the stack. Stack overflow occurs again, and this time the trap is made.

5. An interrupt between setting of the T-bit and fetch of the traced instruction – The entire interrupt service routine is executed and then the T-bit is set again by the exiting RTI. The traced instruction is executed (if there have been no other interrupts) and, unless it is a special case noted above, causes a trace trap.

6. Interrupt trap priorities – See Table 1-8.

6.3.7 Miscellaneous Instructions

HALT



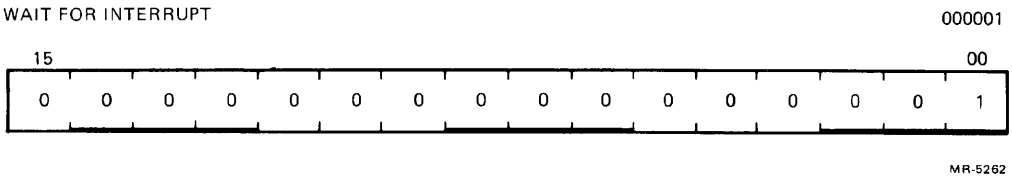
Operation:

↓ (SP) ← PS
↓ (SP) ← PC
PC ← restart address
PS ← 340

Condition Codes: Not affected

Description: The effect of HALT depends upon the CPU operating mode and the halt option currently selected. See Chapter 8 for more details on halt options. In kernel mode, a halt option of 1 (external logic driving a 1 on DAL3 in response to a GP Read with a GP code of 000) causes a trap through location 4 and sets bit 7 of the CPU error register when HALT is executed. If the halt option is 0 in kernel mode, execution of the HALT instruction causes the KDJ11-A into console ODT. Execution of the HALT instruction in user or supervisor mode causes a trap through location 4 and sets bit 7 of the CPU error register.

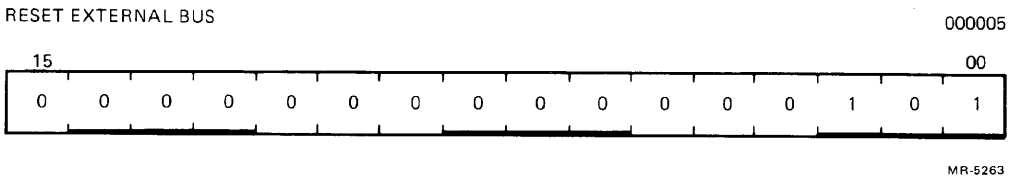
WAIT



Condition Codes: Not affected

Description: In WAIT, as in all instructions, the PC points to the next instruction following the WAIT instruction. Thus, when an interrupt causes the PC and PS to be pushed onto the processor 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. If not in kernel mode, WAIT executes as a NOP.

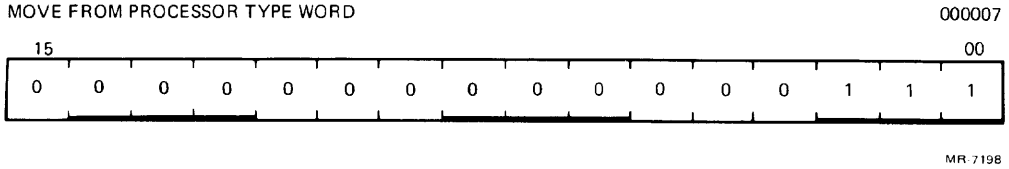
RESET



Condition Codes: Not affected

Description: The following sequence of events occurs: (1) a GP Write cycle is performed and a GP code of 014 is generated; (2) operation is delayed for 69 microcycles; (3) a GP Write is performed and a GP code of 214 is generated; (4) operation is delayed for 600 microcycles delay. If not in kernel mode, RESET operates as a NOP.

MFPT



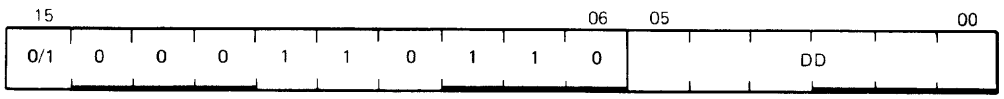
Operation: R0 ← 5

Condition Codes: Not affected

Description: The number 5 is placed in R0, indicating to the system software that the processor type is KDJ11-A.

MTPD/MTPI

MOVE TO PREVIOUS DATA SPACE
MOVE TO PREVIOUS INSTRUCTION SPACE ■066DD



MR-11571

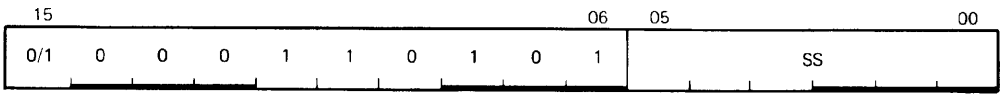
Operation: (temp) ← (SP)+
(dst) ← (temp)

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

Description: The instruction pops a word off the current stack determined by PS bits <15:14> and stores that word into an address in the previous space (PS bits <13:12>). The destination address is computed using the current registers and memory map.

MFPD/MFPI

MOVE FROM PREVIOUS DATA SPACE
MOVE FROM PREVIOUS INSTRUCTION SPACE ■065SS



MR-11570

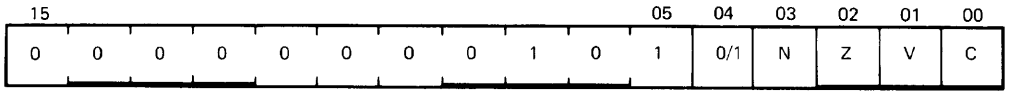
Operation: (temp) ← (src)
−(SP) ← (temp)

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

Description: Pushes a word onto the current stack from an address in the previous space determined by PS<13:12>. The source address is computed using the current registers and memory map. When MFPI is executed and both previous mode and current mode are user, the instruction functions as though it were MFPD.

6.3.8 Condition Code Operators

CLN SEN
CLZ SEZ
CLV SEV
CLC SEC
CCC SCC



MR-5266

Description:

Set and clear 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 <03:00>) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., set the bit specified by bit 0, 1, 2, or 3, if bit 4 = 1. Clear corresponding bits if bit 4 = 0.

Mnemonic	Operation	Op Code
CLC	Clear C	000241
CLV	Clear V	000242
CLZ	Clear Z	000244
CLN	Clear N	000250
SEC	Set C	000261
SEV	Set V	000262
SEZ	Set Z	000264
SEN	Set N	000270
SCC	Set all CCs	000277
CCC	Clear all CCs	000257
	Clear V and C	000243
NOP	No operation	000240

Combinations of the above set or clear operations may be ORed together to form combined instructions.

CHAPTER 7
FLOATING-POINT ARITHMETIC

7.1 INTRODUCTION

The KDJ11-A executes 46 floating-point instructions. The floating-point instruction set is compatible with the FP11 instruction set for PDP-11 computers. Both single- and double-precision floating-point capabilities are available with other features, including floating-to-integer and integer-to-floating conversion.

7.2 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 is 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.

7.2.1 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 200 notation [i.e., as K + 200 (octal)], giving a biased exponent. Thus, exponents from −128 to +127 could be represented by 0 to 377 (base 8), or 0 to 255 (base 10). For reasons given below, a biased exponent of 0 [the true exponent of −200 (octal)], is reserved for floating-point 0. Therefore, exponents are restricted to the range −127 to +127 inclusive (−177 to +177 octal) or, in excess 200 notation, 1 to 377.

The remaining bit of the floating-point word is the sign bit. The number is negative if the sign bit is a 1.

7.2.2 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 KDJ11-A 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 (octal) is regarded as producing a floating overflow; if the true exponent is less than −177 (octal), 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 octal). (Recall that only eight bits are reserved for the biased exponent.) The fractional part of results obtained from such overflow and underflow is correct.

7.2.3 Undefined Variables

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 was to ensure 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 FIUV bit discussed later), is intended to provide the user with a debugging aid: if −0 occurs, it did not result from a previous floating-point arithmetic instruction.

7.2.4 Floating-Point Data

Floating-point data is stored in words of memory as illustrated in Figures 7-1 and 7-2.

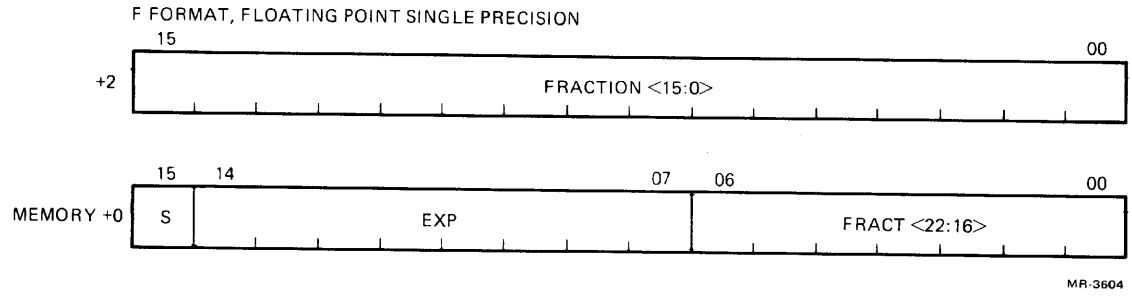


Figure 7-1 Single-Precision Format

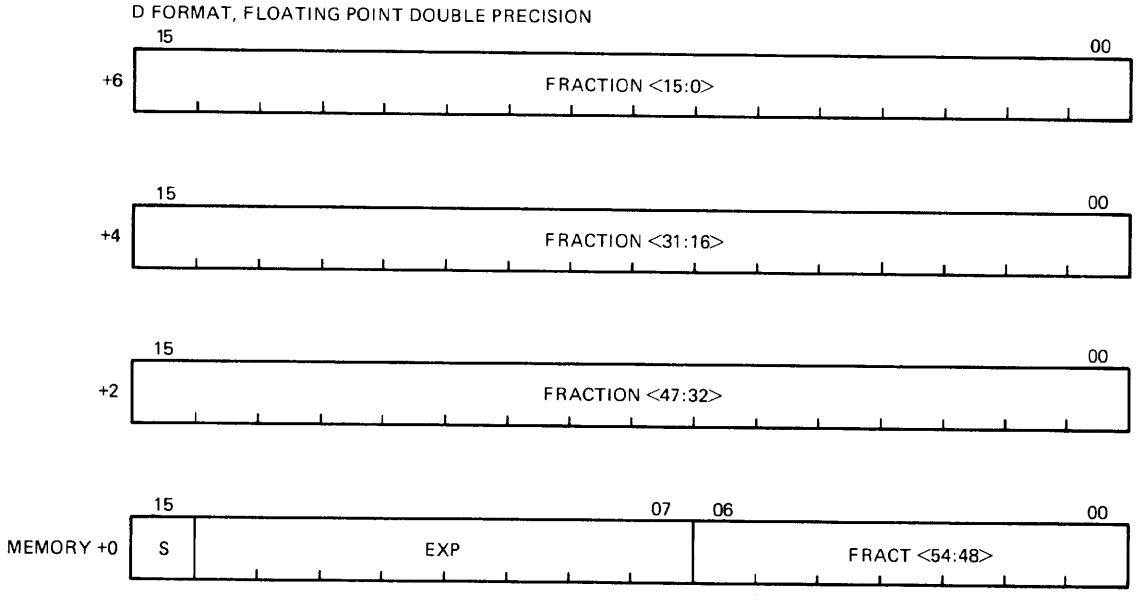


Figure 7-2 Double-Precision Format

The KDJ11-A 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 7-3.)

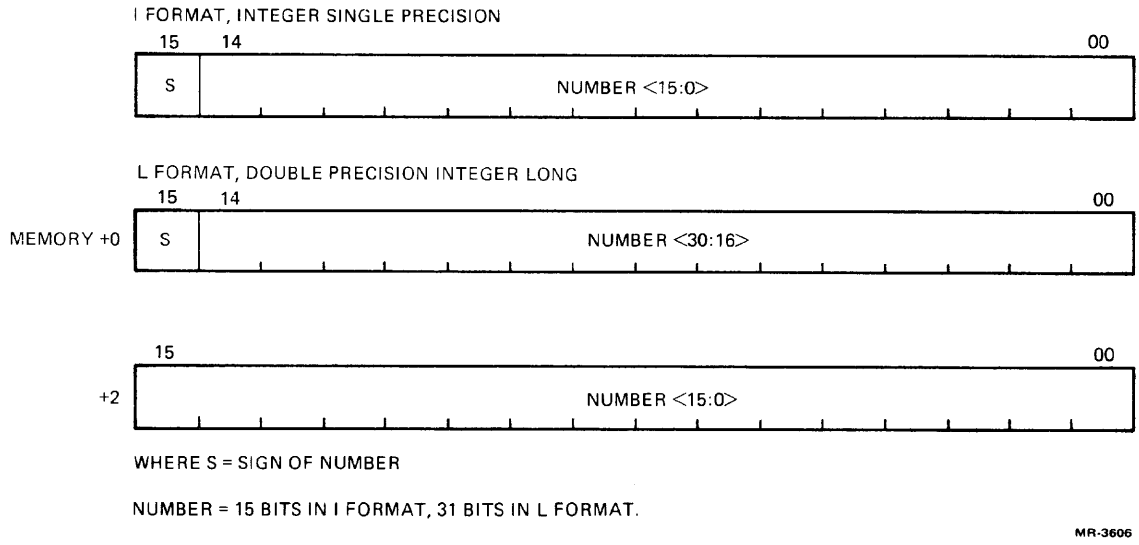


Figure 7-3 2's Complement Format

7.3 FLOATING-POINT STATUS REGISTER (FPS)

This register provides mode and interrupt control for the currently executing floating-point instruction and also reflects conditions resulting from the execution of the previous instruction. (See Figure 7-4.) In this discussion a set bit = 1 and a reset bit = 0. Three bits of the FPS register control the modes of operation as follows.

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" to avoid confusion with truncation of series used in approximations for function subroutines.

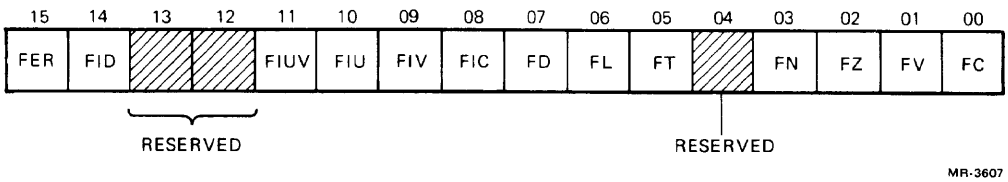


Figure 7-4 Floating-Point Status Register

The FPS register contains an error flag and four condition codes (5 bits): carry, overflow, zero, and negative, which are analogous to the CPU condition codes.

The KDJ11-A 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 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 13 bits are stored in the FPS register as shown in Figure 7-4. The FPS register bits are described in Table 7-1.

Table 7-1 FPS Register Bits

Bit	Name	Description
15	Floating Error (FER)	The FER bit is set by the KDJ11-A 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 KDJ11-A never resets the FER bit. Once the FER bit is set by the KDJ11-A, it can be cleared only by an LDFPS instruction (note the RESET instruction does not clear the FER bit). This means that the FER bit is up-to-date only if the most recent floating-point instruction produced a floating-point exception.
14	Interrupt Disable (FID)	If the FID bit is set, all floating-point interrupts are disabled. **NOTES** **1. The FID bit is primarily a maintenance feature. It should normally be clear. In particular, it must be clear is one wishes to assure that storage of −0 by the KDJ11-A is always accompanied by an interrupt.** **2. Throughout the rest of the chapter, assume that the FID bit is clear in all discussions involving overflow, underflow, occurrence of −0, and integer conversion errors.**

Table 7-1 FPS Register Bits (Cont)

Bit	Name	Description
13		Reserved for future DIGITAL use.
12		Reserved for future DIGITAL use.
11	Interrupt on Undefined Variable (FIUV)	An interrupt occurs if FIUV is set and a −0 is obtained from memory as an operand of ADD, SUB, MUL, DIV, CMP, MOD, NEG, ABS, TST, or any LOAD instruction. The interrupt occurs before execution on all instructions. When FIUV is reset, −0 can be loaded and used in any floating-point operation. Note that the interupt 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. A result of −0 will not be stored without the simultaneous occurrence of an interrupt.
10	Interrupt on Underflow (FIU)	When the FIU bit is set, floating underflow will cause an interrupt. The fractional part of the result of the operation causing the interrupt will be correct. The biased exponent will be too large by 400, except for the special case of 0, which is correct. An exception is discussed later in the detailed description of the LDEXP instruction.
09	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. If the FIV bit is reset and overflow occurs, there is no interrupt. The KDJ11-A returns exact 0. Special cases of overflow are discussed in the detailed descriptions of the MOD and LDEXP instructions.
08	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.
07	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.
06	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).
05	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.
04		Reserved for future DIGITAL use.
03	Floating Negative (FN)	FN is set if the result of the last floating-point operation was negative; otherwise it is reset.

Table 7-1 FPS Register Bits (Cont)

Bit	Name	Description
02	Floating Zero (FZ)	FZ is set if the result of the last floating-point operation was 0; otherwise it is reset.
01	Floating Overflow (FV)	FV is set if the last floating-point operation resulted in an exponent overflow; otherwise it is reset.
00	Floating Carry (FC)	FC is set if the last floating-point operation resulted in a carry of the most significant bit. This can only occur in floating double-to-integer conversions.

7.4 FLOATING EXCEPTION CODE AND ADDRESS REGISTERS

One interrupt vector is assigned to take care of all floating-point exceptions (location 244). 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 zero error
6 Floating-to-integer or double-to-integer conversion error
8 Floating overflow error
10 Floating underflow error
12 Floating undefined variable error

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 zero
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, the FEC and FEA registers are updated.

NOTES

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.**

7.5 FLOATING-POINT INSTRUCTION ADDRESSING

Floating-point 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 = Floating-point accumulator
1 = Deferred
2 = Autoincrement
3 = Autoincrement-deferred
4 = Autodecrement
5 = Autodecrement-deferred
6 = Indexed
7 = Indexed-deferred

Autoincrement and autodecrement operate on increments and decrements of 4 for F format, and 10 (octal) for D format.

In mode 0 users can make use of all six floating-point accumulators (AC0–AC5) as their source or destination. Specifying floating-point 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 floating-point 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.

7.6 ACCURACY

General comments on the accuracy of the KDJ11-A floating-point instructions 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 KDJ11-A for all instructions.

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:

   - always decreased by chopping.
   - decreased by rounding if the rounding bit is 0.
   - 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 KDJ11-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 09) in Table 7-1.

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 floating overflow (FV) bit or the floating exception register (FEC). The reader can readily verify that (for the standard arithmetic operations ADD, SUB, MUL, and DIV) the biased exponent returned by the instruction bears the following relation to the correct exponent.

1. On overflow, it is too small by 400 (octal)

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 (octal).

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.

7.7 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 7-5); SRC and DST operands use CPU addressing modes.

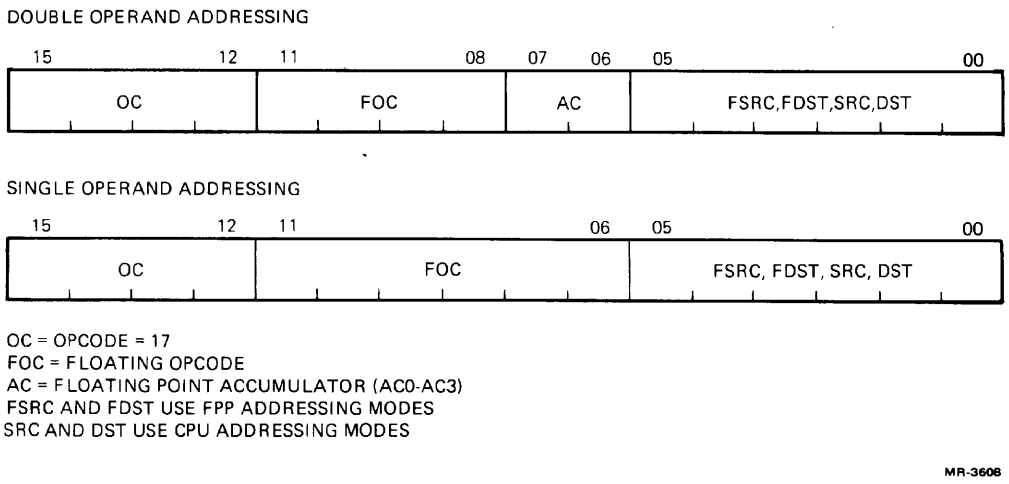


Figure 7-5 Floating-Point Addressing Modes

Terms Used in Instruction Definitions

OC = op code = 17

FOC = floating op code

AC = contents of accumulator, as specified by AC field of instruction

fsrc = address of floating-point source operand

fdst = address of floating-point destination operand

f = fraction

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)

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

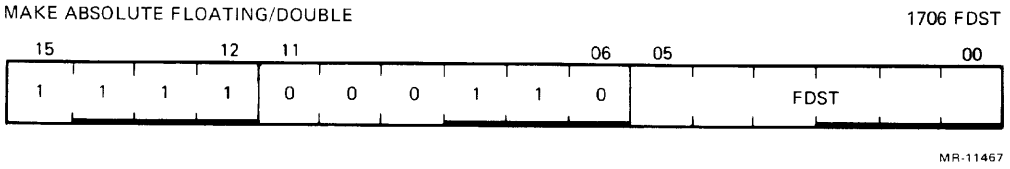
$\wedge$ = AND

$\vee$ = inclusive OR

$\forall$ = exclusive OR

$\sim$ = NOT

ABSF/ABSD



Format: ABSF FDST

Operation: If (FDST) < 0, (FDST) ← − (FDST).

If EXP(FDST) = 0, (FDST) ← exact 0.

For all other cases, (FDST) ← (FDST).

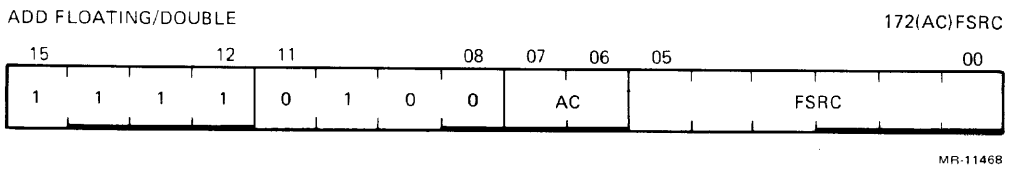
Condition Codes:
FC ← 0
FV ← 0
FZ ← 1 if (FDST) = 0, else FZ ← 0
FN ← 0

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

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

Accuracy: These instructions are exact.

ADDF/ADDD



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.

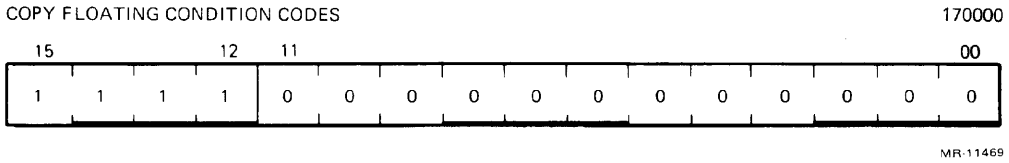
Interrupts: If FIUV is enabled, trap on −0 in FSRC occurs before execution. If overflow or underflow occurs, and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400 for overflow. It is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

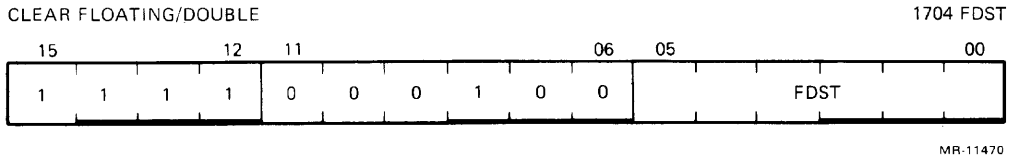


Format: CFCC

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

Description: Copy the floating-point condition codes into the CPU's condition codes.

CLRF/CLRD



Format: CLRF FDST

Operation: (FDST) ← exact 0

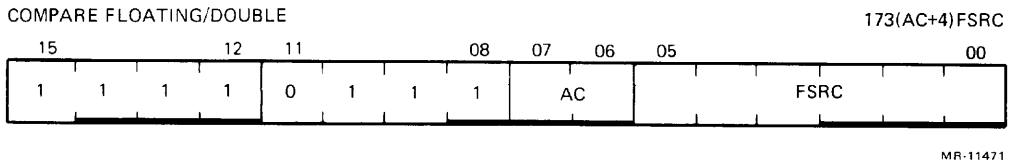
Condition Codes:
FC ← 0
FV ← 0
FZ ← 1
FN ← 0

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



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

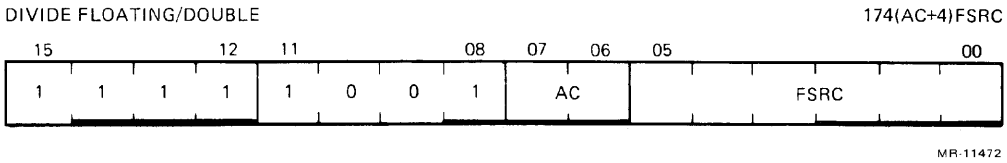
Description: Compare the contents of FSRC with the accumulator. Set the appropriate floating-point condition codes. FSRC and the accumulator are left unchanged except as noted below.

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 KDJ11-A will store an exact 0 in AC.

DIVF/DIVD



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 for overflow. It is too large by 400 for underflow, except for the special case of 0, which is correct.

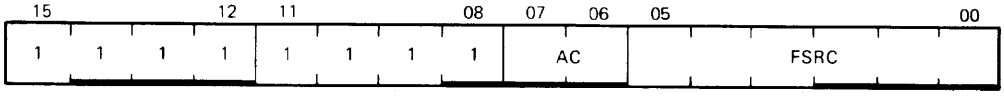
Accuracy: Errors due to overflow and underflow are described above. If none of these occurs, the error in the quotient will be bounded by 1 LSB in chopping mode and by 1/2 LSB in rounding mode.

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



MR-11473

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. Overflow cannot occur for LDCFD.

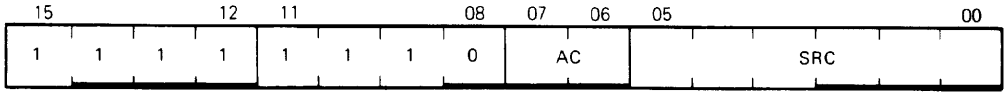
A trap occurs if FIV is enabled, and if rounding with LDCDF causes overflow. AC ← overflowed result. This result must be +0 or −0. Underflow cannot occur.

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



MR-11474

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.

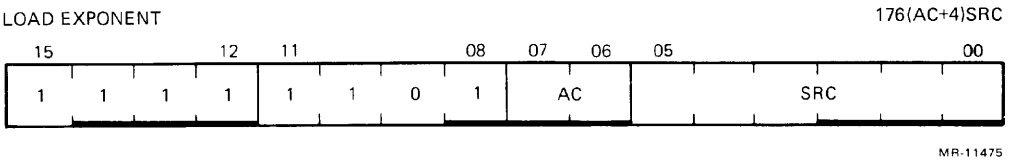
If a 32-bit integer is specified (L mode) and (SRC) has an addressing mode of 0 or immediate addressing mode is specified, the 16 bits of the source register are left-justified and the remaining 16 bits loaded with 0s before conversion.

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



Format: LDEXP SRC,AR

Operation: NOTE: 177 and 200, appearing below, are octal numbers.

If −200 < SRC < 200, EXP(AC) ← SRC + 200 and the rest of AC is unchanged.

If (SRC) > 177 and FIV is enabled, EXP(AC) ← [(SRC) + 200]<07:00>.

If (SRC) > 177 and FIV is disabled, AC ← exact 0.

If (SRC) < −177 and FIU is enabled, EXP(AC) ← [(SRC) + 200]<07:00>.

If (SRC) < −177 and FIU is disabled, AC ← exact 0.

Condition Codes:
FC ← 0
FV ← 1 if (SRC) > 177, else FV ← 0
FZ ← 1 if (AC) = 0, else FZ ← 0
FN ← 1 if (AC) < 0, else FN ← 0

Description: Change AC so that its unbiased exponent = (SRC). That is, convert (SRC) from 2's complement to excess 200 notation and insert it into the EXP field of AC. This is a meaningful operation only if ABS(SRC) LE 177.

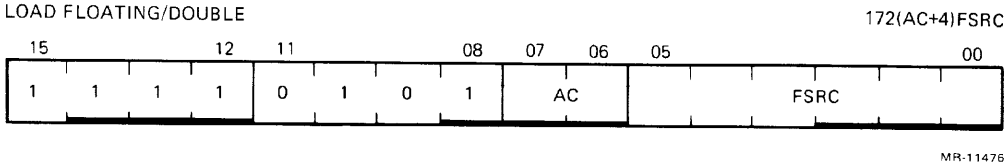
If SRC > 177, the result is treated as overflow. If SRC < −177, the result is treated as underflow.

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

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



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

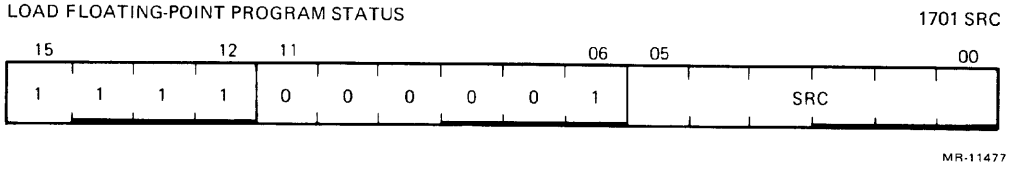
Description: Load single- or double-precision number into AC.

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



Format: LDFPS SRC

Operation: FPS ← (SRC)

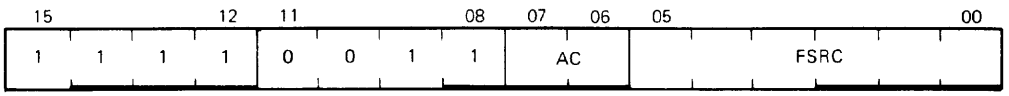
Description: Load floating-point status register from SRC.

Special Comment: Users are cautioned not to use bits 13, 12, and 04 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



MR-11478

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)

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. g .LT. 1.

Both N and g 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 g 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 $\vee$ 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 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 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 59 bits of PROD are generated.

   If 2 ** p .LE. ABS(N) .LT. 2 ** (p + 1), with p > 2, the low order p − 2 bits of g may be in error.

4. If ABS(PROD) .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 4008 (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.

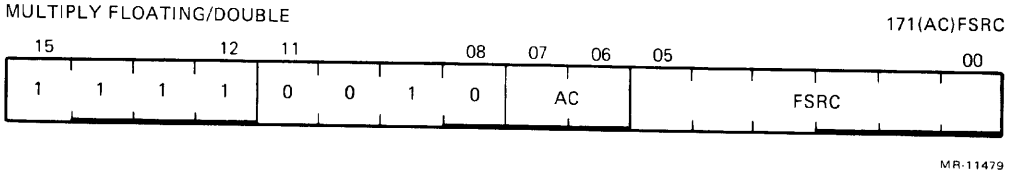
2. To reduce the argument of a trigonometric function.

   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



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 48 bits for floating mode and 59 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

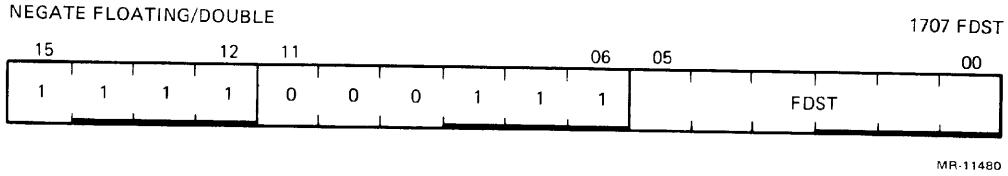
For these exceptional cases, an exact 0 is stored in AC.

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 for overflow. It is too large by 400 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



Format: NEGF FDST

Operation: (FDST) ← − (FDST) if (FDST) = 0, else (FDST) ← exact 0

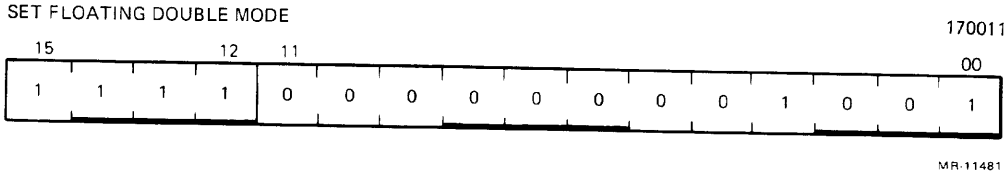
Condition Codes:
FC ← 0
FV ← 0
FZ ← 1 if (FDST) = 0, else FZ ← 0
FN ← 1 if (FDST) < 0, else FN ← 0

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

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

Accuracy: These instructions are exact.

SETD

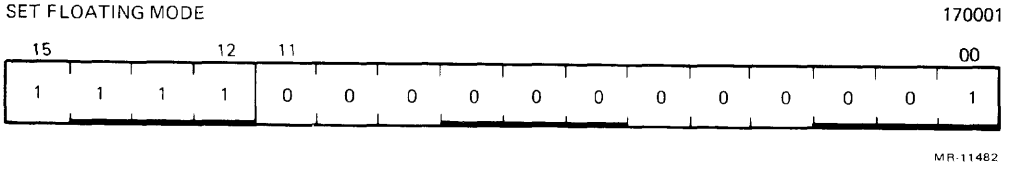


Format: SETD

Operation: FD ← 1

Description: Set the KDJ11-A in double-precision mode.

SETF

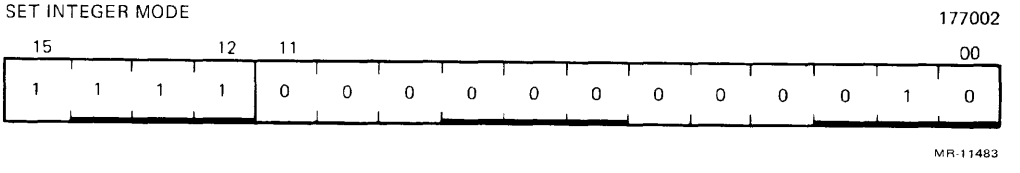


Format: SETF

Operation: FD ← 0

Description: Set the KDJ11-A in single-precision mode.

SETI

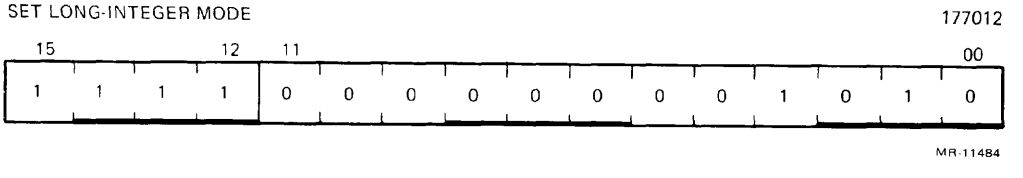


Format: SETI

Operation: FL ← 0

Description: Set the KDJ11-A for short-integer data.

SETL



Format: SETL

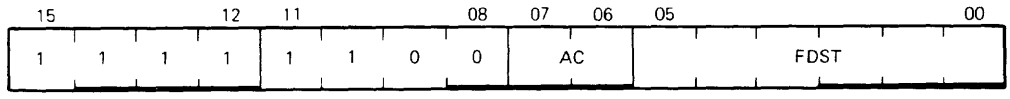
Operation: FL ← 1

Description: Set the KDJ11-A for long-integer data.

STCFD/STCDF

STORE AND CONVERT FROM FLOATING-TO-DOUBLE
AND FROM DOUBLE-TO-FLOATING

176(AC)FDST



MR-11485

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.

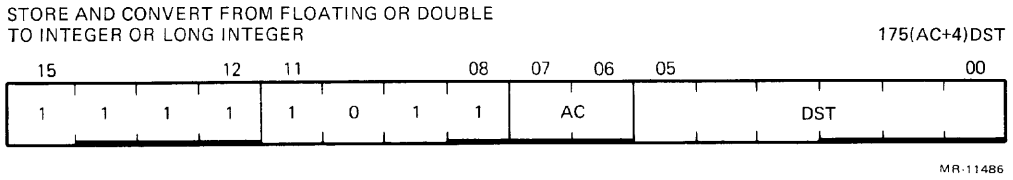
If the current mode is double-precision, the contents of the accumulator are converted to single-precision, chopped or rounded depending on the state of FT, and stored in FDST.

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



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.

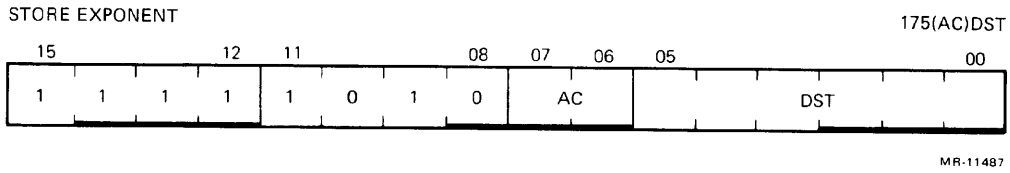
If the operation is out of the integer range selected by FL, FC is set to 1 and the contents of the DST are set to 0.

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



Format: STEXP AC,DST

Operation: (DST) ← EXP(AC) − 200

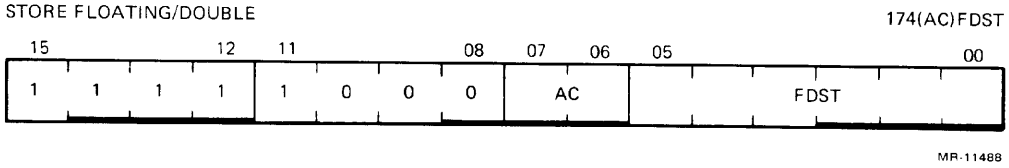
Condition Codes:
C, FC ← 0
V, FV ← 0
Z, FZ ← 1 if (DST) = 0, else Z, FZ ← 0
N, FN ← 1 if (DST) < 0, else N, FN ← 0

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



Format: STF AC,FDST

Operation: (FDST) ← AC

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

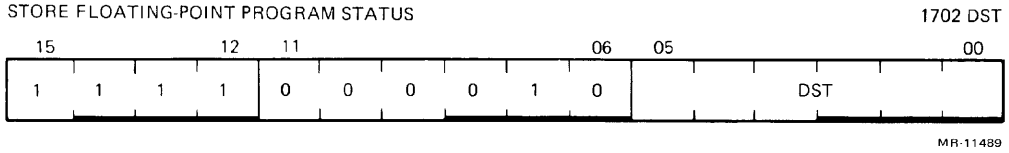
Description: Store single- or double-precision number from AC.

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 an AC of the KDJ11-A. One occurs when underflow or overflow is present and the corresponding interrupt is enabled. A second occurs when an LDF or LDD instruction is executed and the FIUV bit is disabled.

STFPS



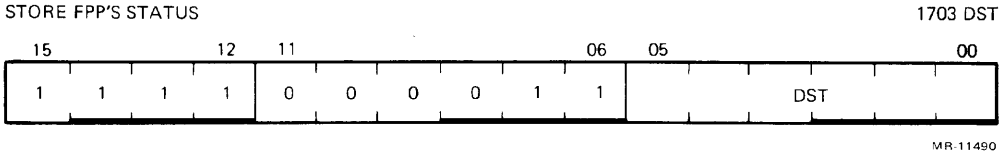
Format: STFPS DST

Operation: (DST) ← FPS

Description: Store the floating-point status register in DST.

Special Comment: Bits 13, 12, and 04 are loaded with 0. All other bits are the corresponding bits in the FPS.

STST



Format: STST DST

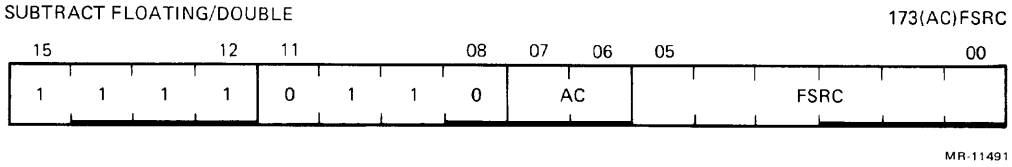
Operation: (DST) ← FEC (DST + 2) ← FEA

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



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.

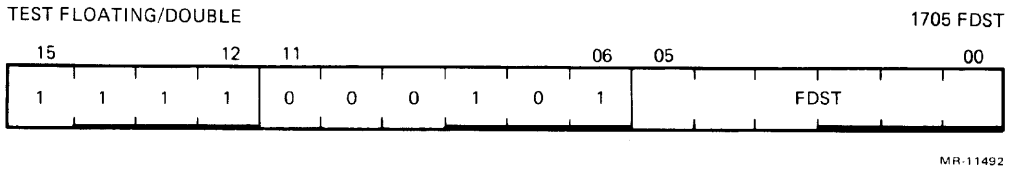
Interrupts: If FIUV is enabled, trap on −0 in FSRC occurs before execution. If overflow or underflow occurs, and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too small by 400 for overflow. It is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



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 floating-point condition codes according to the contents of FDST.

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

Accuracy: These instructions are exact.

CHAPTER 8
PROGRAMMING TECHNIQUES

8.1 INTRODUCTION

The KDJ11-A offers a great deal of programming flexibility and power. Utilizing the combination of the instruction set, the addressing modes, and the programming techniques, it is possible to develop new software or to utilize old programs effectively. The programming techniques in this chapter show the capabilities of the KDJ11-A. The techniques discussed involve position-independent coding (PIC), stacks, subroutines, interrupts, reentrancy, coroutines, recursion, processor traps, programming peripherals, and conversion.

8.2 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 KDJ11-A 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.

8.2.1 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.

The following 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
```

8.2.2 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.

8.3 STACKS

The stack is part of the basic design architecture of the KDJ11-A. 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 8-1) require instructions capable of operating on bytes rather than full words.

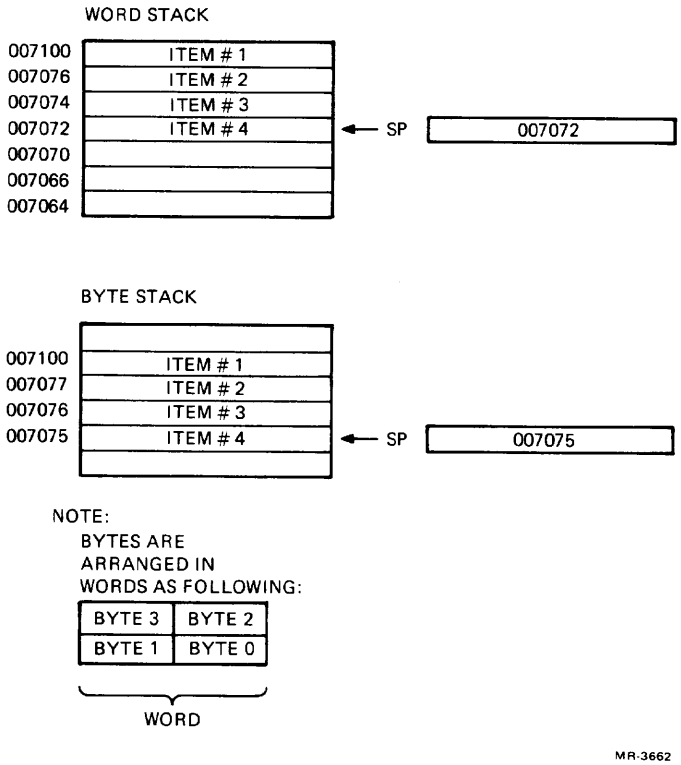


Figure 8-1 Word and Byte Stacks

8.3.1 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.

8.3.2 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 8-2.)

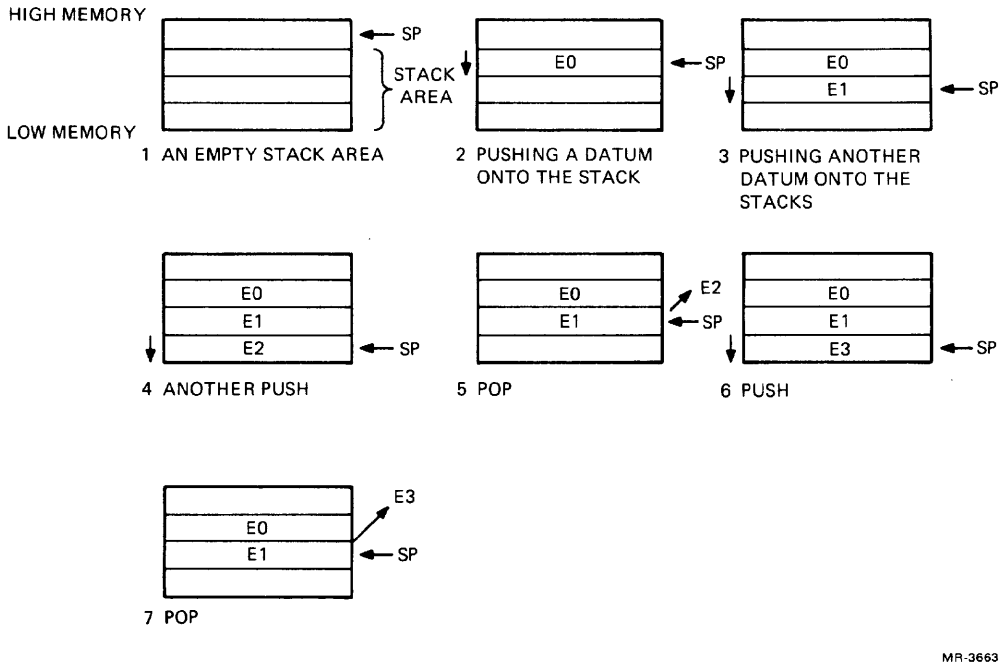


Figure 8-2 Push and Pop Operations

8.3.3 Deleting Items from a Stack

The following techniques may be used to delete items 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 50 items from a word stack use:

ADD#100.,SP

8.3.4 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.

8.3.5 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 usage.

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 8-3.)

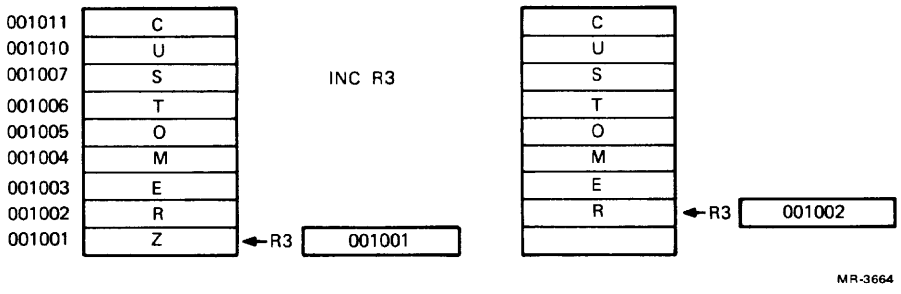


Figure 8-3 Byte Stack Used as a Character Buffer

8.3.6 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 8-4 shows the conditions before and after the subroutine instructions JSR R5, 1064 are executed.

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.

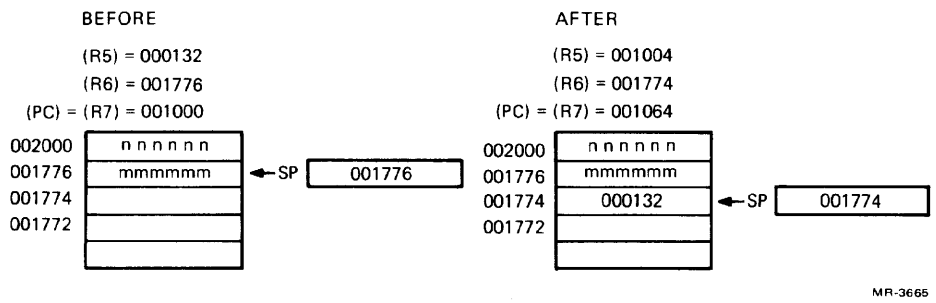


Figure 8-4 JSR Stack Condition Example

8.3.6.1 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.

8.3.6.2 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.

8.3.7 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, 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.

8.3.7.1 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). 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.

8.3.7.2 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 8-5.)

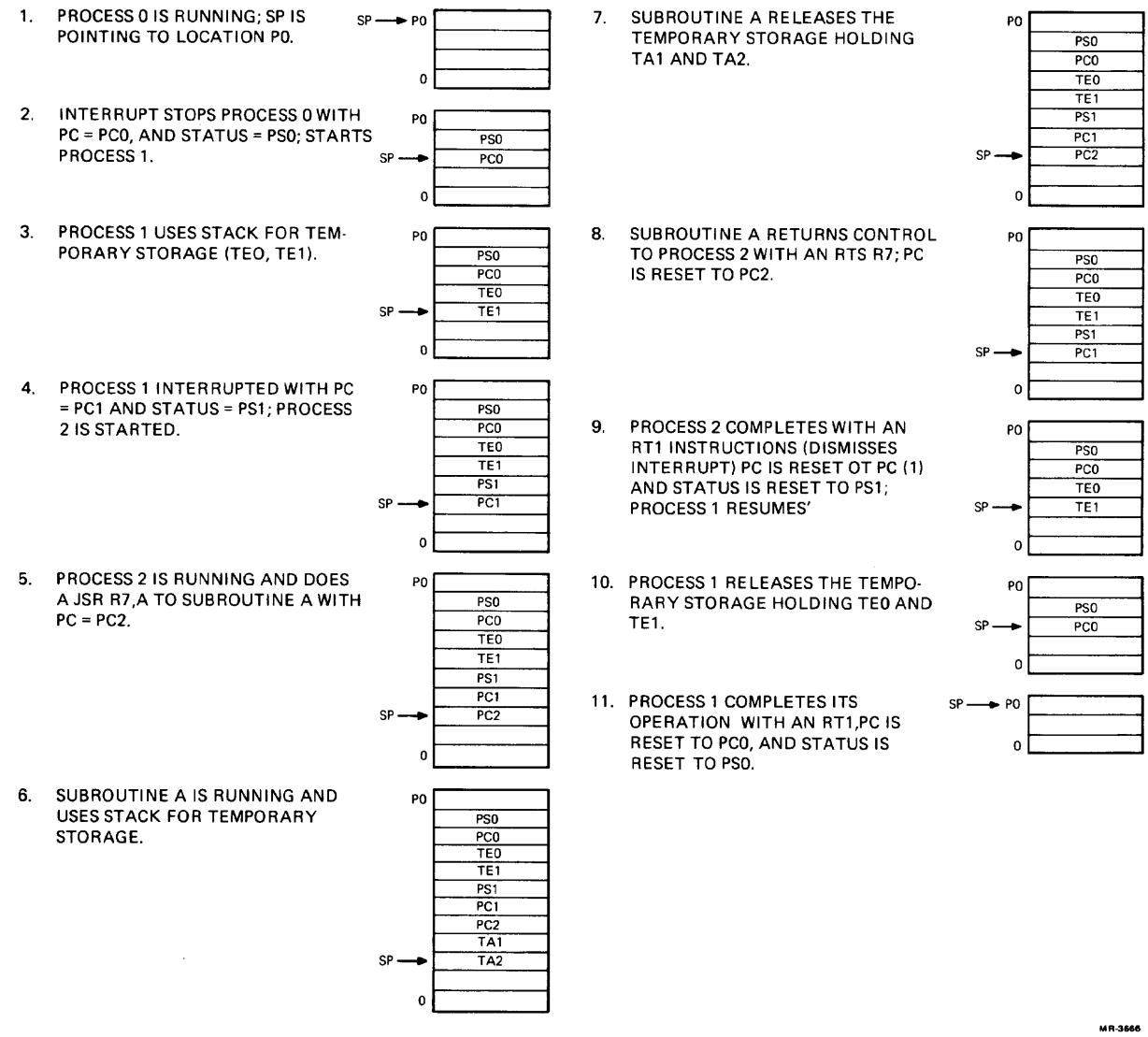


Figure 8-5 Nested Interrupt Service Routines and Subroutines

8.3.8 Reentrancy

Other advantages of the KDJ11-A 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 8-6 may occur.

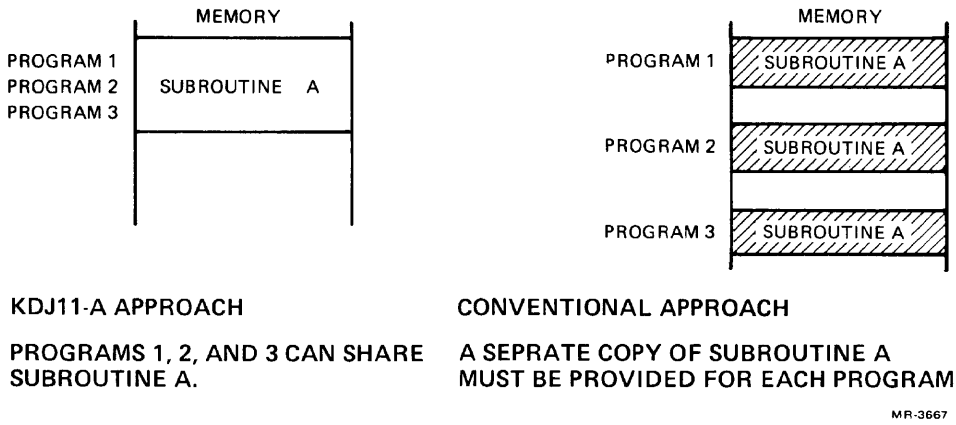


Figure 8-6 Reentrant Routines

8.3.8.1 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 than standard code.
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 8-7.)

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.

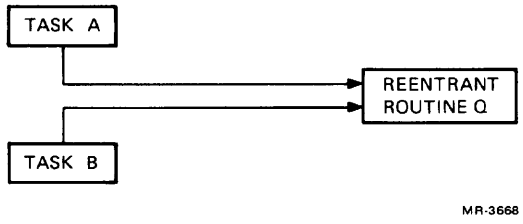


Figure 8-7 Sharing Control of a Routine

8.3.8.2 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.

8.3.9 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)+
```

8.3.9.1 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 8-8. Notice that the coroutine linkage cleans up the stack with each control transfer.

ROUTINE A	STACK		ROUTINE B	COMMENTS
.			.	LOC IS PUSHED
.			.	ONTO THE STACK
.			.	TO PREPARE FOR
MOV #LOC,-(SP)	LOC	←SP		THE COROUTINE
.				CALL.
.				
			LOC:	
JSR PC,@(SP)+	PC0	←SP	.	WHEN THE CALL
(PC0)			.	IS EXECUTED,
			.	THE PC FROM
				ROUTINE A IS
				PUSHED ON THE
				STACK AND EXE-
				CUTION CONTIN-
				UES AT LOC.
			JSR PC,@(SP)+	ROUTINE B CAN
.	PC1	SP	(PC1)	RETURN CONTROL
.			.	TO ROUTINE A
.			.	BY ANOTHER
			.	COROUTINE CALL.
				PC0 IS POPPED
				FROM THE STACK
				AND EXECUTION
				RESUMES IN
				ROUTINE A JUST
				AFTER THE CALL
				TO ROUTINE B,
				I.E., AT PC0.
				PC1 IS SAVED
				ON THE STACK
				FOR A LATER
				RETURN TO
				ROUTINE B.

MR-3669

Figure 8-8 Coroutine Example

8.3.9.2 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 8-9.

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 8.3.6.1 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.

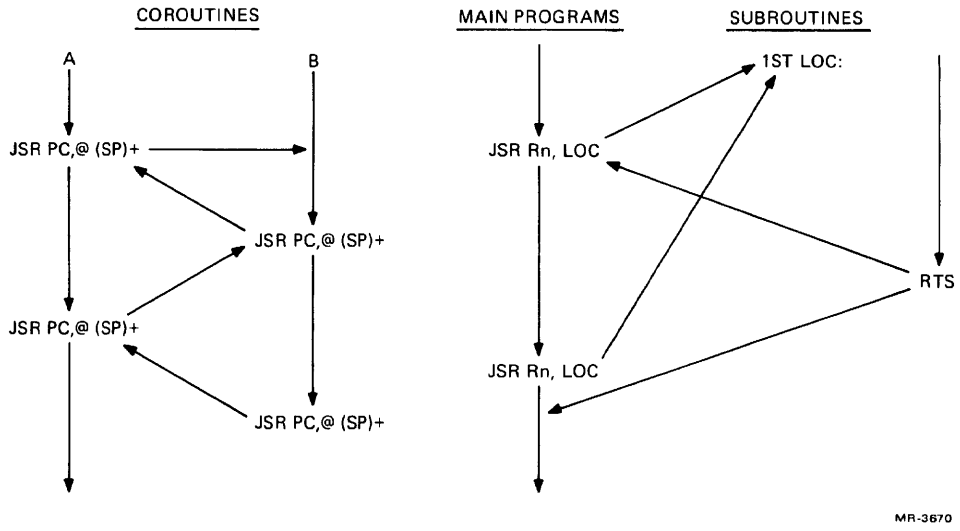


Figure 8-9 Coroutines Versus Subroutines

8.3.9.3 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 8-10 illustrates this example.

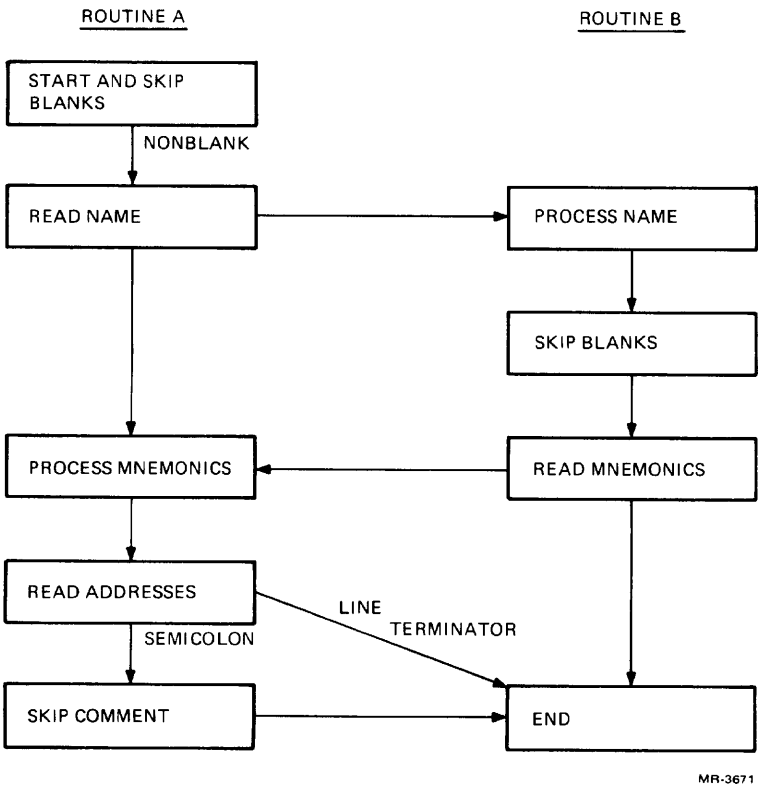


Figure 8-10 Coroutine Path

Coroutines can be utilized in I/O processing. The example above shows coroutines used in double-buffered I/O using IOX. The flow of events might be described as:

```
Write 01
Read I1          concurrently,
Process I2
```

then

```
Write 02
Read I2          concurrently,
Process I1
```

Figure 8-11 illustrates a coroutine swapping 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.

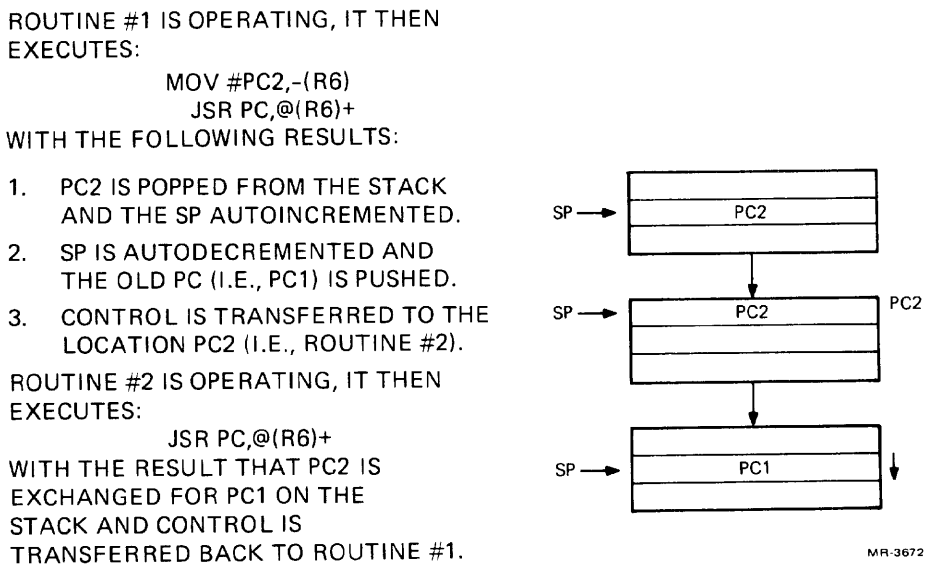


Figure 8-11 Coroutine Interaction

8.3.10 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 8-12.

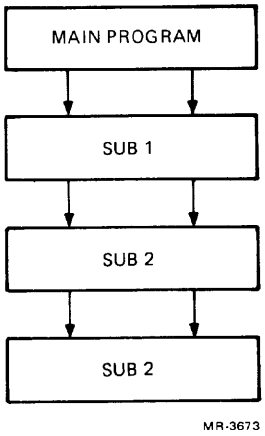


Figure 8-12 Recursive Routine Flow

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.

8.3.11 Processor Traps

Certain errors and programming conditions cause the KDJ11-A 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.

* Nonmaskable software cannot inhibit the condition. CTLERR, MMUERR, BUSERR, PARERR are mutually exclusive when the processor is executing a program.

Condition	Description
Interrupt Level 7 (BIRQ7) Interrupt Level 6 (BIRQ6) Interrupt Level 5 (BIRQ5) Interrupt Level 4 (BIRQ4)	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. These inputs are maskable by PS<07:05>.
Halt Line	If the BHALT L bus signal is asserted during the service state, the processor will enter ODT mode.

PS<07:05>	Levels Inhibited
7	All
6	6, 5, 4
5	5, 4
4	4
0–3	None

8.3.11.1 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.

8.3.11.2 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             CHAN,BISB CHAN,R0
    EMT 374
    .ENDM
```

Note that the EMT low byte is 374. This is interpreted by the EMT handler to indicate a group of routines. Then the contents of R0 (high byte) are tested by the handler to identify exactly which routine within the group is being requested – in this case routine number 2. (The CM3 call of the .RSUM is set up to pass the identification code.)

8.3.12 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 approached 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_8 * 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
```

8.4 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 KDJ11-A 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 KDJ11-A 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.

8.5 PROGRAMMING PERIPHERALS

Programming LSI-11 bus-compatible modules (devices) is simple. A special class of instructions that deals 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.

8.6 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 Address	Program 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				
	000500		END		

Program Address	Program 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 Address	Program Contents	Label	Op Code	Operand	Comments
					;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 Address	Program 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 Address	Program 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!!!

Program Address	Program Contents	Label	Op Code	Operand	Comments
					;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		

8.7 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 9
BOOT ROMS AND DIAGNOSTICS

9.1 INTRODUCTION

The KDJ11-A module may be incorporated into some type of LSI-11 based system using a mass storage device and a system console. The system should contain a multifunction option such as the MXV11-B with a system device bootstrap program that is included in the MXV11-B2 ROM option. These ROMs are required for on-site Field Service support.

The operation of the XXDP+ diagnostics for the KDJ11-A module are described in this section.

9.2 MXV11-B2 ROM SET

The MXV11-B2 ROM set is a bootstrap/diagnostic option for the MXV11-B multifunction module and the MRV11-D universal PROM module. The option performs bootstrap programs for mass storage devices and diagnostic programs on the CPU, memory, and I/O devices during power-up or when manually invoked.

The bootstrap function is automatic at power-up if the CPU is configured for this feature. The system console can be used to boot devices at nonstandard I/O page addresses, select a secondary system device, or run a diagnostic program.

CAUTION

In the event of a power failure, if a system uses battery backup, the user should not power-up using the automatic mode. During the power-up sequence, this mode executes a memory diagnostic and could destroy the data stored. An alternative power-up mode should be selected.

The MXV11-B2 supports turnkey operation so that the user does not have to initiate the bootstrap function. It supports all the system devices currently available for the LSI-11 bus. These include the RLO1, RLO2, TVS05, TU58, RX50/RD51.

9.2.1 Power-Up

The MXV11-B2 performs a memory diagnostic at power-up. On completion of the memory test, a search is conducted for a bootable device. During the power-up sequence, the console port is monitored for a CTRL C command and, if it occurs, the sequence is aborted and the BOOT?> prompt appears on the console.

9.2.2 Automatic Booting

The KDJ11-A will power-up at 17 773 000 when power-up option 2 is selected. The MXV11-B2 option will automatically perform the power-up diagnostics and then search for a bootable device as follows.

RLO1/RLO2 (units 0 through 3)
RX50/RD51* (units 0 through 7)
RX02 (units 0 and 1)
RXO1 (units 0 and 1)
TSVO5 (unit 0 only)
TU58

The MXV11-B2 boots a volume from unit 0 of the first mass storage device found. If unit 0 cannot be booted, it searches through RX and RD units 1–7 in sequence of the same device for a bootable volume. When a bootable volume cannot be located, it proceeds to the next device in sequence and exercises the same routine. A message appears on the console approximately every 30 seconds until a volume is bootstrap loaded. If no devices exist or respond to the booting sequence, then it will try to boot a TU58.

When a bootable volume is found, the MXV11-B2 reads the boot code from the selected mass storage device and unit (logical block 0) into successive memory locations, starting at address 0. It loads the unit number and the device CSR address into registers 0 and 1, respectively.

9.2.3 Manual Booting

Pressing a CTRL C before a device is booted will abort the program and enter the manual mode by issuing the BOOT?> program or ODT prompt "@". The KDJ11-A module allows the user to select a bootstrap address by using power-up option 3. A list of the MXV11-B2 boot commands are listed in Table 9-1.

Table 9-1 MXV11-B2 Boot Commands

Command	Group	Function
CLn	Utility	Clock on/off
mDDn	Boot	Boot TU58
mDLn	Boot	Boot RL01/RL02
mDUn	Boot	Boot MSCP devices (RX50/RD51)*
mDXn	Boot	Boot RX01
mDYn	Boot	Boot RX02
HE	Utility	Help
IN	Utility	Initialize bus
LD	Utility	Load from boot block
MP	Utility	Show memory map
mMSn	Boot	Boot TSV05
n/	Utility	Examine/deposit memory
mNEn	Boot	Boot DECnet via DLV11-E
mNFn	Boot	Boot DECnet via DLV11-F
mNPn	Boot	Boot DECnet via DPV11
mNUn	Boot	Boot DECnet via DUV11
OD	Utility	Enter console ODT
mTCn	Utility	Clock test
TF	Utility	Floating-point test
mTMn	Utility	Test memory
mTSn	Utility	Serial line test

* The boot searches for removable (RX50) disk and then fixed disk (RD51).

* Sequences through MSCP (mass storage control protocol) removable units 0 through 7, then MSCP fixed units 0 through 7.

9.2.4 Error and Help Messages

The MXV11-B2 ROMs will printout on the system console a variety of error and help messages when the system fails to be booted. In the automatic mode, a message is displayed every 30 seconds while it searches for a bootable device, this does not represent a failure. The messages can occur for either the automatic or manual mode. A fatal message is always preceded by BOOTROM-F-; other messages will provide helpful information to the user. The messages are listed in Table 9-2 with suggestions to help the user.

Table 9-2 MXV11-B2 Error Messages

Message*	Cause	Suggested User Action
Automatic Boot Soft Error Message		
No device ready after x tries.	No bootable device or volume available to load. This message repeats at 30-second intervals until 10th message, then repeats at 15-minute intervals (approximately).	Close doors on floppy if system is on RX01 or RX02 media. Make sure that RL01/RL02 READY (white) indicator is on, etc. If problem is not obvious and the message repeats, press CTRL C and try to boot desired device with a keyboard command. More specific messages will appear.
Automatic Boot Fatal Error Messages		
?BOOTROM-F Memory parity error at xxxxxx. ?BOOTROM-F Memory error at xxxxxx.	Defective memory unit or MMU detected.	Record the message and number. Turn power off, then on. If problem remains, service is required. If you wish to bypass the memory test, use manual mode by rebooting system, pressing CTRL C, and then using the LOAD command.
?BOOTROM-F Unknown error - call for help.	Fatal hardware failure detected.	Record all relevant information about the system, including the LED indicators on MXV11-B module (if installed). Service is required.
xxxxxx @	Fatal hardware failure or bad system volume detected.	Try a different system volume, if available (one you know works, if possible). If the problem remains, record information as above. Service is required.
Any partially printed message.	Fatal hardware failure detected, possibly the console.	If possible, try a different console. If the problem remains, record information. Service is required.
General Command Error Messages		
?BOOTROM-F Syntax error in command.	Illegal character or other general input error occurred.	Retype command correctly.
?BOOTROM-F No such command - type HE for help.	Invalid or misspelled command entered.	Refer to manual, or type HE to get a list of all valid commands.
?BOOTROM-F Too many characters.	More than 8 octal digits typed before the 2-letter command, or more than 1 digit following command, or more than 17 letters in command.	Retype command correctly.
?BOOTROM-F Number not octal.	An 8 or 9 was typed.	Determine correct number and retype command.

* XX = device mnemonic, x = octal number

Table 9-2 MXV11-B2 Error Messages (Cont)

Message*	Cause	Suggested User Action
Manual Boot Messages		
You can produce these messages by using one of the commands in the boot group (Table 9-1). Some device-specific messages are listed in the next section of this table.		
Enter a device and unit	Previous command was LD.	If you wish to load a device boot block into memory without executing it, enter a valid command from the boot group. Normal load-and-go operation is restored after the command executes.
XX x boot block read.	Normal termination for a boot group command when the previous command was LD.	Examine or alter the boot block in locations 000000 to 000776 by using console ODT.
No boot block on volume.	The volume has a format that corresponds to a Digital data-only volume.	Remove the volume and replace with correct one, or (if it is not a Digital system volume) boot it with the LD command. (Refer to LD command section.)
Unknown boot block on volume boot anyway?	The volume has a format that does not correspond to any Digital standard.	Type N and retry with a different volume. If it is not a Digital system volume, type Y; this transfers control to secondary boot at location zero.
?BOOTROM-F No XX device at x.	If a CSR was explicitly typed in, it may be incorrect. If none was typed, the device is missing, defective, or configured for a nonstandard I/O page address.	If CSR was incorrect, retype with correct CSR. If not, service is required. (Hardware must be supported by Digital, and device must be part of your system.)
?BOOTROM-F XX x read error.	Error detected in the device or volume.	Try another volume you know is good. If the problem remains, service is required.
?BOOTROM-F XX x error.	Device error detected.	Service may be required, unless there is an obvious solution.
?BOOTROM-F XX x not ready.	Volume not ready to be read by device (for example, not loaded).	The solution depends on the device, and is usually obvious after inspection (for example, volume not inserted into device, floppy drive door open, or RL02 disk cover left out). If the device has a panel of status indicators, they may give a clue. If there is no obvious solution, service may be required.

* XX = device mnemonic, x = octal number

Table 9-2 MXV11-B2 Error Messages (Cont)

Message*	Cause	Suggested User Action
?BOOTROM-F Bad CSR number.	CSR number typed in is greater than 177560, less than 160000, or odd, or specified CSR is that of the console.	Retype the command, using correct CSR address.
?BOOTROM-F Bad Unit number.	Specified unit does not exist in system, or the number is greater than maximum number of units supported by single controller for specified device type.	If device uses unit- number plugs, such as RL disks, they may have been changed or removed without operator knowledge. Check device for plugs and retype command. If not, there may be a hardware fault.
?BOOTROM-F Unknown error - call for help. ?BOOTROM-F Fatal ROM error.	Fatal hardware failure detected.	Record all relevant information about the system, including LED indicators on the MXV11-B module (if installed). Service is required.
xxxxxx @	Fatal hardware failure or a bad system volume detected.	Try a different system volume, if available (one you know works, if possible). If the problem remains, record all relevant information, including the LED indicators on the MXV11-B module (if installed). Service may be required.
Any partially printed message.	Fatal hardware failure detected, possibly the console.	If possible, try a different console. If problem remains, record all information. Service may be required.
?BOOTROM-F Memory cache parity error.	Cache memory parity error or failure detected.	Replace processor module or continue to use system without cache (cache turned off). System simply runs slower.
Device-Specific Manual Boot Messages		
RX02 unit with RX01 volume. Boot anyway? (Occurs with RX02 floppy disk systems.)	RX02 drive loaded with single-density volume.	If you know the volume contains a valid RX02 boot-only block, type Y. If volume is unknown, it may be an RX01 disk.
?BOOTROM-F Comm error. (Occurs only while booting DECnet via a serial line from a keyboard command, such as NE.)	DECnet boot could not be executed due to hardware or software problem in host system, target system, or communication link.	Check the communication line. Service may be required.

* XX = device mnemonic, x = octal number

If the option is installed in the MXV11-B module, the LEDs on the module can indicate errors.

The LEDs read as follows. The single red LED to one side of the green LED is bit 3; the three red LEDs to the other side of the green LED are bits 2 to 0, with bit 2 being the red LED closest to the green LED.

3		2	1	0	
Red	Green	Red	Red	Red	(As seen looking at the edge of the board, with the components up.)

In the following chart, a 1 indicates the LED is on, and 0 indicates the LED is off. The green LED indicates +5 Vdc is applied to RAM memory.

The chart shows which part of the ROM program was executing when the system hung up.

	LEDs			
3	**2**	**1**	**0**	
0	0	0	0	Successful boot
0	0	0	1	Comprehensive memory test
0	0	1	0	Waiting for console input
0	0	1	1	Low memory test (below 2000 octal)
0	1	0	0	MSCP device (RX50/RD51)
0	1	0	1	Not assigned
0	1	1	0	Not assigned
0	1	1	1	RL01/RL02 boot
1	0	0	0	RX01/RX02 boot
1	0	0	1	TSV05 boot
1	0	1	0	Not assigned
1	0	1	1	DPV11 DECnet boot
1	1	0	0	DUV11 DECnet boot
1	1	0	1	DLV11 DECnet boot
1	1	1	0	TU58 boot
1	1	1	1	Power-up initialization

LED indicator codes that are not assigned should never appear when using the MXV11-B2.

NOTE

A 1111 indicator code appears after a successful DECnet boot.

9.3 DIAGNOSTICS

The XXDP+ diagnostic programs help to verify the system is functioning correctly or to isolate a faulty component. These are used for maintenance purposes and not as part of the normal system operation. The XXDP+ diagnostic software consists of a library of diagnostic programs designed to test individual system components. These can be chained together, dependent on the system configuration, to provide an overall system diagnostic. The diagnostics specifically used for the KDJ11-A module are listed in Table 9-3 and are described below.

Table 9-3 KDJ11-A Diagnostics

Name	Function
CZKDJAO	CPU tests
CZKDKAO	Memory management tests
CZKDLAO	Floating-point tests
CZKDMAO	Cache memory tests

The HALT trap option must be disabled by installing the W5 jumper when running these diagnostics. The diagnostic program can be halted by asserting the HALT line. This is done by pressing the BREAK key on the system console for systems configured to assert HALT when BREAK is keyed. They can be restarted by addressing location 152 010 and pressing the G key on the system console. The system monitor "." will prompt and the diagnostic program can be selected by the run command R followed by the diagnostic name. The name will be echoed and the program started. The name of the diagnostic is printed on the first pass and completed tests are identified by the system console printing END PASS. When an error is detected, the diagnostic will halt and print out the error condition as follows.

Error = Specific Function Being Tested
Error = (Unique Error Number)
Error PC = (PC at Time of Error)

9.4 DIAGNOSTIC EXAMPLE

An example of running the diagnostics is described below. The response of the user is underlined and the system response is typed. The W5 jumper must be installed. Comments are listed on the right hand side to further explain the example.

Diagnostic	Comments
28	
START? DL<CR>	Booted DL device
CHMDLC1 XXDP+ DL MONITOR	XXDP+ monitor
BOOTED VIA UNIT 0	
28K UNIBUS SYSTEM	May be LSIBUS or UNIBUS 28K=MEMORY SIZE OR STANDARD
ENTER DATE (DD-MMM-YY): 1-NOV-83	User enters date
RESTART ADDRESS: 152010	Identifies restart address
THIS IS XXDP+. TYPE "H" OR	
"H/L" FOR HELP	
.R CZKDJO<CR>	. = System monitor R = RUN command
CZKDJO.BIC	CZKDJO = Diagnostic <CR> = RETURN key

CZKDJO KDJ11 CPU Diagnostic

END PASS # 1
END PASS # 2
END PASS # 3 — Halt test by pressing break
027622 — Address at HALT

@152010G — Key restart address and G for GO

.R CZKDKO<CR> — Run diagnostic and return

CZKDKO.BIC

SET BIT 8 = 1 FOR 18 BIT SYSTEM — Set bit 8 by 000400
SWR = 000000 NEW = <CR> — Press return

CZKDKO KDJ11 Memory Management

END PASS # 1
END PASS # 2
END PASS # 3
END PASS # 4 — Halt test by pressing break
012404 — address at halt

@152010G — Key restart address and G for GO

.R CZKDLO<CR> — Run diagnostic and return
CZKDLO.BIC

CZKDLO KDJ11 Floating Point

END PASS # 1
END PASS # 2
END PASS # 3
END PASS # 4
END PASS # 5 — Halt by pressing BREAK
022242 — address at HALT

@152010G — Key restart address and G for GO

.R CZKDMO<CR> — Run diagnostic and return
CZKDMO.BIC
SET BIT 8 = 1 FOR 18 BIT SYSTEM — Set bit 8 by 000400
SET BIT 9 = 1 FOR CACHE RAM AND TAG RELIABILITY TESTS — Set bit 9 by 001000 / Set bits 8 and 9 by 001400

SWR = 000000 NEW = <CR> — Press RETURN

CZKDMO KDJ11 Cache Memory System

END PASS # 1
END PASS # 2
END PASS # 3
END PASS # 4
END PASS # 5
END PASS # 6
010152

Halt test by pressing BREAK
address at HALT

@152010G

Key restart address and
G for GO

.R

System monitor and run
command

APPENDIX A
INSTRUCTION TIMING

A.1 GENERAL

The execution time required for the base instruction set and the floating-point instruction set used by the KDJ11-A is described in this appendix. The execution time for an instruction is dependent upon the type of instruction, the addressing mode used, and the type of memory accessed. In general, the total execution time is the sum of the base instruction fetch/execute time and the operand(s) address calculation/fetch time.

The execution time provided for all read instructions assumes that the data is accessed from the module cache memory. When the data is accessed from the main memory, the execution time provided must be degraded. Memory write instructions, indicated by the "+" notation, must have the memory write time added to the listed time in order to determine the total time.

The floating-point instruction execution timing is provided as a range. The actual performance is data dependent and will fall within the described range.

A.2 BASE INSTRUCTION SET TIMING

The execution times for the base instruction set are provided in Tables A-1 through A-6 and are subject to the general notes listed at the end of Table A-6.

Table A-1 Source Address Time: All Double Operand

Instruction	Source Mode	Source Register	Microcode Cycles	Time (ns)	Read Memory Cycles
ADD, SUB,	0	0–7	0	0	0
CMP, BIT,	1	0–7	2	534	1
BIC, BIS,	2	0–6	2	534	1
MOV	2	7	1	267	1
	3	0–6	4	1068	2
	3	7	3	801	2
	4	0–6	3	801	1
	4	7	6	1602	2 (Note 1)
	5	0–6	5	1335	2
	5	7	8	2136	3 (Note 1)
	6	0–7	4	1068	2
	7	0–7	6	1602	3

Table A-2 Destination Address Time: Read-Only Single Operand

Instruction	Destination Mode	Destination Register	Microcode Cycles	Time (ns)	Read Memory Cycles
TST, MUL, DIV,	0	0–7	0	0	0
ASH, ASHC, MTPS,	1	0–7	2	534	1
MFPI, MFPD, CSM	2	0–6	2	534	1
	2	7	1	267	1
	3	0–6	4	1068	2
	3	7	3	801	2
	4	0–6	3	801	1
	4	7	7	1869	2 (Note 2)
	5	0–6	5	1335	2
	5	7	9	2403	3 (Note 3)
	6	0–7	4	1068	2
	7	0–7	6	1602	3

Table A-3 Destination Address Time: Read-Only Double Operand

Instruction	Destination Mode	Destination Register	Microcode Cycles	Time (ns)	Read Memory Cycles
CMP, BIT	0	0–7	0	0	0
	1	0–7	3	801	1
	2	0–6	3	801	1
	2	7	2	534	1
	3	0–6	5	1335	2
	3	7	4	1068	2
	4	0–6	4	1068	1
	4	7	8	1236	2 (Note 2)
	5	0–6	6	1602	2
	5	7	10	2670	3 (Note 3)
	6	0–7	5	1335	2
	7	0–7	7	1869	3

Table A-4 Destination Address Time: Write-Only

Instruction	Destination Mode	Destination Register	Microcode Cycles	Time (ns)	Memory Cycles Read	Memory Cycles Write
MOV, CLR, SXT,	0	0–6	0	0	0	0
MFPS, MTPI, MTPD	0	7	5	1335	1	0
	1	0–6	2	534+	0	1
	1	7	6	1602+	1	1
	2	0–6	2	534+	0	1
	2	7	6	1602+	1	1
	3	0–6	4	1068+	1	1
	3	7	3	801+	1	1
	4	0–6	3	801+	0	1
	4	7	7	1869+	1	1
	5	0–6	5	1335+	1	1
	5	7	9	2403+	2	1
	6	0–7	4	1068+	1	1
	7	0–7	6	1602+	2	1

Table A-5 Destination Address Time: Read-Modify-Write

Instruction	Destination Mode	Destination Register	Microcode Cycles	Time (ns)	Memory Cycles Read	Memory Cycles Write
ADD, SUB, ADC,	0	0–6	0	0	0	0
SBC, BIC, BIS,	0	7	5	1335	1	0
SWAB, NEG, INC,	1	0–6	3	801+	1	1
DEC, COM, XOR,	1	7	7	1869+	2	1
ROR, ROL, ASR,	2	0–6	3	801+	1	1
ASL	2	7	7	1869+	2	1
	3	0–6	5	1335+	2	1
	3	7	4	1068+	2	1
	4	0–6	4	1068+	1	1
	4	7	8	2136+	2	1 (Note 2)
	5	0–6	6	1602+	2	1
	5	7	10	2670+	3	1 (Note 3)
	6	0–7	5	1335+	2	1
	7	0–7	7	1869+	3	1

Table A-6 Execution, Fetch Time

Instruction	Microcode Cycles	Time (ns)	Memory Cycles Read	Memory Cycles Write
Double Operand				
ADD, SUB, CMP, BIT, BIC, XOR, MOV, BIS	1	267	1	0
Single Operand				
SWAB, CLR, COM, INC, DEC, NEG, ADC, SBC, TST, ROL, ROR, ASL, ASR, SXT, MFPS, XOR	1	267	1	0
MFPI, MFPD	5	1335+	1	1
MTPS	8	2136	1	0
MTPI, MTPD	3	801	2	0
CSM	28	7476+	3	3
Extended Instruction Set				
MUL	22	5874	1	0 (Notes 5, 11)
DIV				
By zero	5	1335	1	0 (Note 6)
Other	34	9078	1	0 (Notes 6, 7)
ASH	4	1068	1	0 (Notes 8, 11)
ASHC				
No shift	5	1335	1	0
Left	6	1602	1	0 (Notes 8, 9, 11)
Right	7	1869	1	0 (Notes 8, 10, 11)

Table A-6 Execution, Fetch Time (Cont)

Double Operand Instruction	Microcode Cycles	Time (ns)	Memory Cycles Read	Write
Program Control				
BRANCH				
Not Taken	2	534	1	0
Taken	4	1068	2	0
SOB				
Not Taken	3	801	1	0
Taken	5	1335	2	0
IOT, TRAP, EMT, BPT	20	5340+	4	2
MARK	10	2670	3	0

Instruction	Destination Mode	Destination Register	Microcode Cycles	Time (ns)	Memory Cycles Read	Write
JMP	1	0–7	4	1068	2	0
	2	0–7	6	1602	2	0
	3	0–7	5	1335	3	0
	4	0–7	5	1335	2	0
	5	0–7	6	1602	3	0
	6	0–6	6	1602	3	0
	6	7	5	1335	3	0
	7	0–7	7	1869	4	0
JSR (Note 4)	1	0–7	9	2403+	2	1
	2	0–7	10	2670+	2	1
	3	0–6	10	2670+	3	1
	3	7	9	2403+	3	1
	4	0–7	10	2670+	2	1
	5	0–7	11	2937+	3	1
	6	0–6	10	2670+	3	1
	6	7	9	2403+	3	1
	7	0–7	12	3204+	4	1

Instruction	Microcode Cycles	Time (ns)	Memory Cycles Read	Write
RTS 0–6	6	1602	3	0
RTS 7	5	1335	3	0
RTT, RTI	9	2403	4	0

Table A-6 Execution, Fetch Time (Cont)

Double Operand Instruction	Microcode Cycles	Time (ns)	Memory Cycles Read	Write
Miscellaneous Instructions				
MFPT	2	534	1	0
NOP, SET or CLEAR C, V, N, Z	3	801	1	0
SPL	7	1869	1	0
HALT	TBD			
RESET	TBD			
WAIT	TBD			

General Notes to Tables A-1 through A-6

1. Subtract 534 ns and one read if both source and destination modes autodecrement PC, or if WRITE-ONLY or READ-MODIFY-WRITE mode 07 or 17 is used.

2. READ-ONLY and READ-MODIFY-WRITE destination mode 47 references actually perform 3 read operations. For bookkeeping purposes, one of the reads is accounted for in the EXECUTE, FETCH TIMING.

3. READ-ONLY and READ-MODIFY-WRITE destination mode 57 references actually perform 4 read operations. For bookkeeping purposes one of the reads is accounted for in the EXECUTE, FETCH TIMING.

4. Subtract 267 ns if link register is PC.

5. Add 267 ns if the source operand is negative.

6. Subtract 267 ns if the source mode is not zero.

7. Add 267 ns if the quotient is even.
   Add 534 ns if overflow occurs.
   Add 1335 ns and 1 read if the PC is used as a destination register, but only if source mode 47 or 57 is not used.

8. Add 267 ns per shift.

9. Add 267 ns if source operand<15:6> is not zero.

10. Subtract 267 ns if one shift only.

11. Add 1068 ns and 1 read if the PC is used as a destination register, but only if source mode 47 or 57 is not used.

A.3 FLOATING-POINT INSTRUCTION SET TIMING

The execution time range for the floating-point instruction set is described in Tables A-7 through A-12.

Table A-7 Instruction Execution Times (In Microseconds)

Instruction	Minimum	Typical	Maximum	Non-mode 0 Section
ABSD	6.1		6.4	IV
ABSF	5.1		5.3	IV
ADDD	10.9	12.8	31.7	II
ADDF	8.3	9.3	31.7	II
CFCC	1.3		1.3	-
CLRD	3.7		3.7	III
CLRF	3.2		3.2	III
CMPD	6.4		6.7	II
CMPF	4.8		5.1	II
DIVD	42.7		44.5	II
DIVF	15.7		16.8	II
LDCDF	6.4		6.9	II
LDCFD	5.3		5.6	II
LDCID	8.3		11.2	V
LDCIF	6.9		9.6	V
LDCLD	8.3		13.9	V
LDCLF	6.9		11.7	V
LDD	4.3		4.5	II
LDEXP	4.5		4.8	V
LDF	3.2		3.5	II
LDFPS	1.6		1.6	V
MODD	53.9	51.9	71.5	II
MODF	21.9	25.1	30.1	II
MULD	44.0		46.1	II
MULF	14.9		16.3	II
NEGD	5.9		6.1	IV
NEGF	4.8		5.1	IV
SETD	1.6		1.6	-
SETF	1.6		1.6	-
SETI	1.6		1.6	-
SETL	1.6		1.6	-
STCDF	4.5		5.3	III
STCDI	6.9		10.1	VI
STCDL	6.9		14.4	VI
STCFD	5.1		5.3	III
STCFI	6.1		9.3	VI
STCFL	6.1		13.6	VI
STD	3.2		3.2	III
STEXP	4.3		4.3	VI
STF	2.1		2.1	III
STFPS	2.4		2.4	VI
STST	1.9		1.9	VI
SUBD	12.5	14.7	32.5	II
SUBF	9.9	10.9	27.7	II
TSTD	2.9		3.2	II
TSTF	2.4		2.7	II

Table A-8 Floating Source Modes 1–7

Instruction	Mode	Register	Microcode Cycles	Time (ns)	Memory Read	Memory Write
Single Precision						
ADDF, CMPF,	1	0–7	3	801	2	0
DIVF, LDCDF,	2	0–6	3	801	2	0
LDF, MODF,	2	7	1	267	1	0
MULF, SUBF,	3	0–6	4	1068	3	0
TSTF	3	7	3	801	3	0
	4	0–7	4	1068	2	0
	5	0–7	5	1335	3	0
	6	0–7	4	1068	3	0
	7	0–7	6	1602	4	0
Double Precision						
ADDD, CMPD,	1	0–7	5	1335	4	0
DIVD, LDCFD,	2	0–6	5	1335	4	0
LDD, MODD,	2	7	0	0	1	0*
MULD, SUBD,	3	0–6	6	1602	5	0
TSTD	3	7	5	1335	5	0
	4	0–7	6	1602	4	0
	5	0–7	7	1869	5	0
	6	0–7	6	1602	5	0
	7	0–7	8	2136	6	0

* Mode 27 references only access single word operands. The execution time listed has been compensated in order to accurately compute the total execution time.

Table A-9 Floating Destination Modes 1–7

Instruction	Mode	Register	Microcode Cycles	Time (ns)	Memory Read	Memory Write
Single Precision						
CLRF, STCDF, STF	1	0–7	3	801+	0	2
	2	0–6	3	801+	0	2
	2	7	1	267+	0	1
	3	0–6	4	1068+	1	2
	3	7	3	801+	1	2
	4	0–7	4	1068+	0	2
	5	0–7	5	1335+	1	2
	6	0–7	4	1068+	1	2
	7	0–7	6	1602+	2	2
Double Precision						
CLRD, STCFD, STD	1	0–7	5	1335+	0	4
	2	0–6	5	1335+	0	4
	2	7	0	0	0	1*
	3	0–6	6	1602+	1	4
	3	7	5	1335+	1	4
	4	0–7	6	1602+	0	4
	5	0–7	7	1869+	1	4
	6	0–7	6	1602+	1	4
	7	0–7	8	2136+	2	4

* Mode 27 references only access single word operands. The execution time listed has been compensated in order to accurately compute the total execution time.

Table A-10 Floating Read-Modify-Write Modes 1–7

Instruction	Mode	Register	Microcode Cycles	Time (ns)	Memory Read	Memory Write
Single Precision						
ABSF, NEGF	1	0–7	5	1335+	2	2
	2	0–6	5	1335+	2	2
	2	7	1	267+	1	1*
	3	0–6	6	1602+	3	2
	3	7	5	1335+	3	2
	4	0–7	6	1602+	2	2
	5	0–7	7	1869+	3	2
	6	0–7	6	1602+	3	2
	7	0–7	8	2136+	4	2
Double Precision						
ABSD, NEGD	1	0–7	9	2403+	4	4
	2	0–6	9	2403+	4	4
	2	7	0	0	1	1*
	3	0–6	10	2670+	5	4
	3	7	9	2403+	5	4
	4	0–7	10	2670+	4	4
	5	0–7	11	2937+	5	4
	6	0–7	10	2670+	5	4
	7	0–7	12	3204+	6	4

* Mode 27 references only access single word operands. The execution time listed has been compensated in order to accurately compute the total execution time.

Table A-11 Integer Source Modes 1–7

Instruction	Mode	Register	Microcode Cycles	Time (ns)	Memory Read	Memory Write
Integer						
LDCID, LCDIF,	1	0–7	2	534	1	0
LDEXP, LDFPS	2	0–6	2	534	1	0
	2	7	0	0	1	0*
	3	0–6	3	801	2	0
	3	7	2	534	2	0
	4	0–7	3	801	1	0
	5	0–7	4	1068	2	0
	6	0–7	3	801	2	0
	7	0–7	5	1335	3	0
Long Integer						
LDCLD, LCDLF	1	0–7	4	1068	2	0
	2	0–6	4	1068	2	0
	2	7	0	0	1	0*
	3	0–6	5	1335	3	0
	3	7	4	1068	3	0
	4	0–7	5	1335	2	0
	5	0–7	6	1602	3	0
	6	0–7	5	1335	3	0
	7	0–7	7	1869	4	0

* Mode 27 references only access single word operands. The execution time listed has been compensated in order to accurately compute the total execution time.

Table A-12 Integer Destination Modes 1–7

Instruction	Mode	Register	Microcode Cycles	Time (ns)	Memory Read	Memory Write
Integer						
STCDI, STCFI, STEXP, STFPS	1	0–7	2	534+	0	1
	2	0–6	2	534+	0	1
	2	7	2	534+	0	1
	3	0–6	3	801+	1	1
	3	7	2	534+	1	1
	4	0–7	3	801+	0	1
	5	0–7	4	1068+	1	1
	6	0–7	3	801+	1	1
	7	0–7	5	1335+	2	1
Long Integer						
STCDL, STCFL, STST	1	0–7	4	1068+	0	2
	2	0–6	4	1068+	0	2
	2	7	2	534+	0	1
	3	0–6	5	1335+	1	2
	3	7	4	1068+	1	2
	4	0–7	5	1335+	0	2
	5	0–7	6	1602+	1	2
	6	0–7	5	1335+	1	2
	7	0–7	7	1869+	2	2

APPENDIX B
PROGRAMMING DIFFERENCES

The programming differences between the KDJ11-A processor and the other processors of the PDP-11 family are summarized in Table B-1.

Table B-1 KDJ11-A Programming Differences

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
1. OPR %R, (R) +; OPR %R, – (R) using the same register as both source and destination: contents of R are incremented (decremented) by 2 before being used as the source operand.	X						X	X			X	X
OPR %R, (R) +; OPR %R, – (R) using the same register as both register and destination: initial contents of R are used as the source operand.		X	X	X	X	X			X	X		
2. OPR %R, @ (R) +; OPR %R, @ – (R) using the same register as both source and destination: contents of R are incremented (decremented) by 2 before being used as the source operand.	X						X	X			X	X
OPR %R, @ (R) +; OPR %R, @ – (R) using the same register as both source and destination: initial contents of R are used as the source operand.		X	X	X	X	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			X	X
OPR PC, X (R); OPR PC, @ X (R), OPR PC, A; OPR PC, @ A: location A will contain the PC of OPR +2.		X	X	X	X	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	X	X	X	X

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
5. JMP %R or JSR reg, %R traps to 10 (illegal instruction).	X		X	X	X	X	X	X			X	
JMP %R or JSR reg, %R traps to 4 illegal instruction).		X							X	X		X
6. SWAB does not change V.							X					
SWAB clears V.	X	X	X	X	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) time out when used as a program address by the CPU. Can be addressed under console operation.		X	X	X			X	X	X	X	X	
Register addresses (177700–177717) time out when used as an address by CPU or console.	X				X							X
8. Basic instructions noted in *PDP-11 Processor Handbook*.	X	X	X	X	X	X	X	X	X	X	X	X
SOB, MARK, RTT, SXT instructions*	X	X		X	X			X	X	X	X	X
ASH, ASHC, DIV, MUL, XOR	X	X		X	X			X	X	X	X	X
Floating-point instructions in base machine.											X	X
MFPT instruction.	X	X										X
The external option KE11-A provides MUL, DIV, SHIFT operation in the same data format.						X	X					

*RTT instruction is available in 11/04 but is different than other implementations.

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
The KE11-E (expansion instruction set) provides the instructions MUL, DIV, ASH, and ASHC. These new instructions are 11/45 compatible.								X				
The KE11-F (floating instruction set) adds unique stack ordered oriented point instructions: FADD, FSUB, FMUL, FDIB.								X				
The KEV-11 adds EIS/FIS instructions					X							
MFP, MTP instructions	X	X		X				X		X	X	X
SPL instruction		X							X	X		X
CSM instruction		X										X
9. Power fail during RESET instruction is not recognized until after the instruction is finished (70 milliseconds). RESET instruction consists of 70 millisecond pause with INIT occurring during first 20 milliseconds.							X	X			X	
Power fail immediately ends the RESET instruction and traps if an INIT is in progress. A minimum INIT of microsecond occurs if instruction aborted. PDP11-04/34/44 are similar with no minimum INIT time.		X	X	X					X	X		
Power fail acts the same as 11/45 (22 milliseconds with about 300 nanoseconds minimum). Power fail during RESET fetch is fatal with no power down sequence.						X						

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
RESET instruction consists of 10 microseconds of INIT followed by a 90 microsecond pause. Reset instruction consists of a minimum 8.4 microseconds followed by a minimum 100 nanosecond pause. Power fail not recognized until the instruction completes.	X				X							X
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	X	X	X	X
11. If RTI sets T-bit, T-bit trap is acknowledged after instruction following RTI.						X	X					
If RTI sets T-bit, T-bit trap is acknowledged immediately following RTI.	X	X	X	X	X			X	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	X			X	X
If an interrupt occuurs during an instruction and the T-bit is set, the interrupt is acknowledged before T-bit trap.									X	X		
13. T-bit trap will sequence out of WAIT instruction.	X	X	X	X		X	X	X			X	
T-bit trap will not sequence out of WAIT instruction. Waits until an interrupt.					X				X	X		

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
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	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 implemented in LSI-11, 11/23 or 11/24.		X	X	X	X	X	X					
Odd address/nonexistent references using the stack pointer cause a fatal trap. On bus error in trap service, new stack created at 0/2.	X							X	X	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	X	X	X	X
The first interrupt in an interrupt service is guaranteed to be executed.							X					
17. Single general-purpose register implemented.	X	X	X	X	X	X	X	X			X	
Dual general-purpose register set implemented.									X	X		X

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
18. PSW address, 177776, not implemented; must use instructionns MTPS (move to PS) and MFPS (move from PS).					X							
PSW address implemented, MTPS and MFPS not implemented.		X	X			X	X	X	X	X	X	
PSW address and MTPS and MFPS implemented.	X			X								X
19. Only one interrupt level (BR4) exists.					X							
Four interrupt levels exist.	X	X	X	X		X	X	X	X	X	X	X
20. Stack overflow not implemented.					X							
Some sort of stack overflow implemented.	X	X	X	X		X	X	X	X	X	X	X
21. Odd address trap not implemented.	X				X							
Odd address trap implemented.		X	X	X		X	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				X	X	X	X	X
24. Due to their execution time, FIS instructions can abort because of a device interrupt.					X			X				

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
25. Due to their execution time, FP11 instructions can abort because of a device interrupt.†	X											
FP11 instructions do not abort because of a device interrupt.		X		X					X	X	X	X
26. EIS instructions do a DATIP and DATO bus sequence when fetching source operand.					X							
EIS instructions do a DATI bus sequence when fetching source operand.	X	X		X				X	X	X	X	X
27. MOV instruction does just a DATO bus sequence for the last memory cycle.	X	X		X	X			X	X	X	X	X
MOV instruction does a DATIP and DATO bus sequence for the last memory cycle.			X			X	X					
28. If PC contains nonexistent memory and a bus error occurs, PC will have been incremented.	X	X	X	X	X	X	X		X	X		X
If PC contains nonexistent memory address and a bus error occurs, PC will be unchanged.								X				
29. If register contains nonexistent memory address in mode 2 and a bus error occurs, register will be incremented.	X				X	X	X	X	X	X		X
Same as above but register is unchanged.		X	X	X								

†Integral floating point assumed on 11/23 and 11/24; FP11E assumed for 11/60.

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
30. If register contains an odd value in mode 2 and a bus error occurs, register will be incremented.	X				X			X	X	X		X
If register contains an odd value in mode 2 and a bus error occurs, register will be unchanged.		X	X	X		X	X					
31. Condition codes restored to original values after FIS interrupt abort (EIS does not abort on 35/40).								X				
Condition codes that are restored after EIS/FIS interrupt abort are indeterminate.					X							
32. Op codes 075040 through 0753777 unconditionally trap to 10 as reserved op codes.	X	X	X	X		X	X	X	X	X	X	X
If KEV-11 option is present, op codes 75040 through 07533 perform a memory read using the register specified by the low order 3 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.					X							
33. Op codes 210 through 217 trap to 10 as reserved instructions.	X	X	X	X		X	X	X	X	X	X	X
Op codes 210 through 217 are used as a maintenance instruction.					X							

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
34. Op codes 75040 through 75777 trap to 10 as reserved instructions.	X	X	X	X		X	X	X	X	X	X	X
If KEV-11 options is present, op codes 75040 through 7577 can be used as escapes to user microcode. If no user microcode exists, a trap to 10 occurs.					X							
35. Op codes 170000 through 177777 trap to 10 as reserved instructions.			X			X	X	X				
Op codes 170000 through 177777 are implemented as floating-point instructions.	X	X		X					X	X	X	X
Op codes 170000 through 1777777 can be used as escapes to user microcode. If no user microcode exists, a trap to 10 occurs.					X							
Op code 076600 used for maintenance.											X	
36. CLR and SXT do just a DATO sequence for the last bus cycle.	X											X
CLR and SXT do DATIP-DATO sequence for the last bus cycle.		X	X	X	X	X	X	X	X	X	X	
37. MEM MGT maintenance mode MMR0 bit 8 is implemented.		X		X				X	X	X	X	
MEM MGT maintenance mode MMR0 bit 8 is not implemented.	X											X

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors 23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
38. PS <15:12>, nonkernel mode, nonkernel stack pointer and MTPx and MFPx instructions exist even when MEM MGT is not configured.	X	X							X	X	X	X
PS <15:12>, nonkernel mode, nonkernel stack pointer, and MTPx and MFPx instructions exist only when MEM MGT is configured.								X				
39. Current mode PS bits <15:14> set to 01 or 10 will cause a MEM MGT trap upon any memory reference.	X			X				X				
Current mode PS bits <15:14> set to 10 will be treated as kernel mode (00) and not cause a MEM MGT trap.	X											
Current mode PS bits <15:14> set to 10 will cause a MEM MGT trap upon any memory reference.		X							X	X		X
40. MTPS in user mode will cause MEM MGT trap if PS address 177776 nor mapped. If mapped, PS <07:05> and <03:00> affected.				X								
MTPS in nonuser mode will not cause MEM MGT trap and will only affect PS <03:00> regardless of whether PS address 177776 is mapped.	X											X
41. MFPS in user mode will cause MEM MGT if PS address 177776 not mapped. If mapped, PS <07:00> are accessed.				X								
MTPS in user mode will not trap regardless of whether PS address 177776 is mapped.	X											X

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors 23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
42. Programs cannot execute out of internal processor registers.												X
Programs can execute out of internal processor registers.	X	X		X				X	X	X	X	
43. A HALT instruction in user or supervisor mode will trap through location 4.		X							X	X		X
A HALT instruction in user or supervisor mode will trap through location 10.	X			X				X			X	
44. PDR bit <00> implemented.									X	X		
PDR bit <00> not implemented.	X	X		X				X			X	X
45. PDR bit <07> (any access) implemented.									X	X		
PDR bit <07> (any access) not implemented.	X	X		X				X			X	X
46. Full PAR <15:00> implemented.	X	X								X		X
Only PAR <11:00> implemented.				X				X	X		X	
47. MMR0 <12>–trap-memory management–implemented.									X	X		
MMR0 <12> not implemented.	X	X		X				X			X	X
48. MMR3 <02:00> –D space enable–implemented.		X							X	X		X
MMR3 <02:00> not implemented.	X			X				X			X	

Table B-1 KDJ11-A Programming Differences (Cont)

Feature	Processors											
	23/24	44	04	34	LSI/11	05/10	15/20	35/40	45	70	60	KDJ11-A
49. MMR3 <05:04>–IOMAP, 22-bit mapping enabled–implmented.	X	X								X		X
MMR3 <05:04> not implemented.				X				X	X		X	
50. MMR3 <03>–CSM enable–implemented.		X										X
MMR3 <03> not implemented.	X			X				X	X	X	X	
51. MMR2 tracks instruction fetches and interrupt vectors.									X	X		
MMR2 tracks only instruction fetches.	X	X		X				X			X	X
52. MFPx %6, MTPx when PS <13:12> = 10 gives unpredictable results.	X	X		X				X	X	X	X	
MTPx %6, MTPx %6 when PS <13:12> = 10 uses user stack pointer.												X

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

S

T

W