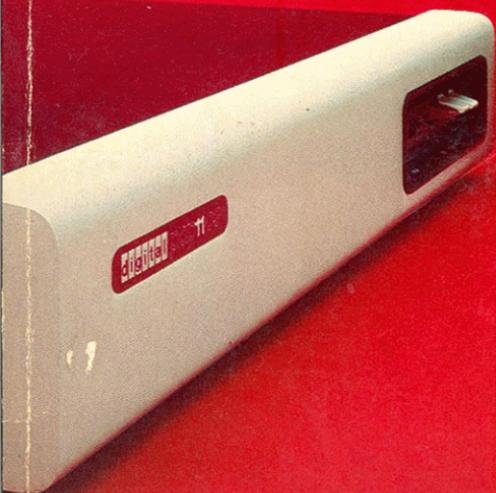


lsim  
pdp11/03  
processor  
handbook



digital

digital

lsim

pdp11/03

processor  
handbook

digital equipment corporation

Copyright © 1975 by Digital Equipment Corporation

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 handbook.

The following are trademarks of  
Digital Equipment Corporation, Maynard, Massachusetts:

DEC

PDP

FLIP CHIP

FOCAL

DIGITAL

DECUS

LSI-11

# CONTENTS

<b>CHAPTER 1 INTRODUCTION</b> .....	1-1
1.1 The LSI-11 Concept .....	1-1
1.2 PDP-11/03 .....	1-1
1.3 Features .....	1-1
1.4 System Architecture .....	1-3
1.5 Microcomputer .....	1-4
1.5.1 General Registers .....	1-5
1.5.2 The Processor Status Word .....	1-7
1.5.3 Instruction Set .....	1-8
1.6 LSI-11 Memory Organization .....	1-9
1.7 LSI-11 BUS .....	1-10
1.7.1 Bidirectional Lines .....	1-12
1.7.2 Master Slave Relation .....	1-12
1.7.3 Interlocked Communication .....	1-12
<b>CHAPTER 2 SPECIFICATIONS</b> .....	2-1
2.1 LSI-11 Operating Specifications .....	2-1
2.2 PDP-11/03 Operating Specifications .....	2-2
2.3 H9270 Backplane Packaging and Mounting .....	2-2
2.4 PDP-11/03 Packaging and Mounting .....	2-2
<b>CHAPTER 3 ADDRESSING MODES</b> .....	3-1
3.1 Single Operand Addressing .....	3-3
3.2 Double Operand Addressing .....	3-3
3.3 Direct Addressing .....	3-5
3.3.1 Register Mode .....	3-5
3.3.2 Autoincrement Mode .....	3-7
3.3.3 Autodecrement (Mode 4) .....	3-8
3.3.4 Index Mode (Mode 6) .....	3-9
3.4 Deferred (Indirect) Addressing .....	3-11
3.5 Use of the PC as a General Register .....	3-13
3.5.1 Immediate Mode .....	3-14
3.5.2 Absolute Addressing .....	3-15
3.5.3 Relative Addressing .....	3-16
3.5.4 Relative Deferred Addressing .....	3-16
3.6 Use of Stack Pointer as General Register .....	3-17
3.7 Summary of Addressing Modes .....	3-17
3.7.1 General Register Addressing .....	3-17
3.7.2 Program Counter Addressing .....	3-19
<b>CHAPTER 4 INSTRUCTION SET</b> .....	4-1
4.1 Introduction .....	4-1
4.2 Instruction Formats .....	4-2
4.3 List of Instructions .....	4-4
4.4 Single Operand Instructions .....	4-6
4.5 Double Operand Instructions .....	4-24
4.6 Program Control Instructions .....	4-34

<b>CHAPTER 5</b>	<b>PROGRAMMING TECHNIQUES</b>	5-1
5.1	The Stack	5-1
5.2	Subroutine Linkage	5-5
5.2.1	Subroutine Calls	5-5
5.2.2	Argument Transmission	5-6
5.2.3	Subroutine Return	5-9
5.2.4	LSI-11 Set Subroutine Calls	5-9
5.3	Interrupts	5-10
5.3.1	General Principles	5-10
5.3.2	Nesting	5-11
5.4	Programming Peripherals	5-13
5.5	Device Registers	5-14
<b>CHAPTER 6</b>	<b>EXTENDED ARITHMETIC OPTION</b>	6-1
6.1	General	6-1
6.2	Fixed Point Arithmetic (EIS)	6-1
6.3	Floating Point Arithmetic (FIS)	6-6
<b>CHAPTER 7</b>	<b>CONSOLE OPERATION</b>	7-1
7-1	General	7-1
7.2	Interfacing	7-1
7.3	ODT/Console Microcode	7-2
<b>APPENDIX A</b>	<b>Memory Map</b>	A-1
<b>APPENDIX B</b>	<b>Instruction Timing</b>	B-1
<b>APPENDIX C</b>	<b>LSI-11 PDP-11 Family of Computers</b>	C-1
<b>APPENDIX D</b>	<b>Instruction Index</b>	D-1
<b>APPENDIX E</b>	<b>Summary of LSI-11 Instructions</b>	E-1

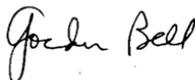
# digital

The LSI-11 (PDP11/03) is the smallest member of the PDP-11 family of computer systems. It offers the user minicomputer performance in a microcomputer package that crosses traditional industry barriers. Therefore, the user can truly add computer power in systems previously too small for computer application. Yet for our traditional user, the boxed version of the LSI-11, the PDP11/03, offers a completely integrated smaller systems tool at lower cost without sacrificing performance. The LSI-11 (PDP11/03) maintains traditional PDP-11 architectural compatibility. This includes programs up to 64K bytes and the use of the (optional) floating instruction set (FIS) and extended instruction set (EIS).

Since the main design objective of the PDP-11 family has always been to optimize total system performance, the interaction of software and hardware have been carefully considered at every step in the design process. The initial PDP-11 was specified in the ISP language and extensively simulated and benchmarked.

The effort of bringing the LSI-11 to the marketplace has required extensive design engineering ranging over many disciplines. The basic N-channel MOS semiconductor densities are state-of-the-art, since the PDP-11 requires many circuits. In fact, the 4-chips that comprise the processor have approximately 50,000 active elements in an area of about  $4 \times 200$  mils  $\times$  200 mils (0.16 square inches). Extensive circuit simulation was carried out for the basic circuits that comprise those elements—using a DECSYSTEM-10. The basic logic masks were laid out using an Applicon system for computer-aided-design, which is based on a PDP-11. Two levels of simulation were carried out on a DECSYSTEM-10 to check the logic (Gate Level), and the behavior of the microprograms (Register-Transfer Level). Finally, complete systems were simulated (including I/O equipment).

All the preparation in the design and production of computers, we believe, is necessary to support our main goal: providing high performance quality computers at the lowest cost by which you, our users, can apply them.



C. Gordon Bell  
Vice President, Engineering  
Digital Equipment Corporation



# INTRODUCTION

### 1.1 THE LSI-11 CONCEPT

DIGITAL introduced the first PDP-11 Processor in 1970. Since then, a family of PDP-11 Computer products has been constantly evolving—not just a family of processors, but a family of peripherals, software, and services. Today, the PDP-11 family is the broadest family of compatible computer products on the market, with one of the latest additions being the LSI-11.

The LSI-11 is a 16-bit microcomputer with the speed and instruction set of a minicomputer. Due to its size and unique capabilities, it can fit into almost any instrumentation, data processing, or controller configuration.

### 1.2 PDP-11/03

The PDP-11/03, a 3½"H x 19"W x 13½"D boxed version of the LSI-11, is designed as an off-the-shelf microcomputer system. It consists of an LSI-11 microcomputer, a modular power supply, and a mounting box. The mounting box is designed to mount in a standard 19" cabinet. For a description of PDP-11/03 specifications, refer to Chapter 2.

### 1.3 FEATURES

The LSI-11 has the following features:

- **400 Plus Instruction Set**  
More than 400 instructions make up the LSI-11's extensive instruction set. This instruction set (also used by the PDP-11/35,40) permits the user to take advantage of standard PDP-11 software. The only departure from the standard software is the addition of two new instructions, used to explicitly access the processor status word (PSW). Development programs as in the PDP-11 family include assemblers, linkers, editors, loaders, utility packages, operating systems, and higher level languages.
- **Extensive Compute Power and Small Processor Size**  
The processor module is built around a set of four N-channel metal oxide semiconductor (MOS) chips, which include control and data elements as well as two microcoded read-only memories (microms). The latter are programmed to emulate the powerful PDP-11/35,40 instruction set, along with routines for on-line debugging techniques (ODT), operator interfacing, and boot-strap loader capability. The processor also contains a 16-bit buffered parallel input/output (I/O) bus, a 4096-word MOS random-access memory (RAM), a real-time clock input, priority interrupt control logic, power-fail/auto restart, and other features to provide stand-alone operation. The entire processor, plus all of the above-mentioned features, are contained on one 8.5-by-10-inch printed circuit board.

- **Modularity**  
The processor, memory, device interfaces, backplane, and interconnecting hardware are all modular in design. Module selection, such as the type and size of memory, and device interfaces, enable custom tailoring to meet specific application requirements.
- **Serial and Parallel I/O Modules**  
Serial and parallel I/O modules are available for interfacing the processor bus with external devices. These modules simplify connection to peripherals when and if required, and also facilitate assembly of prototype systems without penalizing later development of customized interfaces.
- **Choice of Memory**  
Memory modules are offered for applications requiring more storage than is available with the 4096-word MOS random-access memory on the processor board. Included are a non-volatile 4096-word core memory, a 1024-word static RAM, a 4096-word dynamic RAM which can be automatically refreshed by central processor microcode, and read-only memory (PROM/ROM) with capacity to a maximum of 4096 words in 512-word increments (2048 words in 256-word increments).
- **16-Bit Word (Two 8-Bit Bytes)**  
Direct addressing of 32K 16-bit words.
- **Word or Byte Processing**  
Very efficient handling of 8-bit characters without the need to rotate, swap, or mask.
- **Asynchronous Operation**  
System components run at their highest possible speed; replacement with faster devices means faster operation without other hardware or software changes.
- **Stack Processing**  
Hardware sequential memory manipulation makes it easy to handle structured data, subroutines, and interrupts.
- **Direct Memory Access (DMA)**  
Inherent in the architecture is direct memory access for multiple devices.
- **8 General-Purpose Registers**  
For accumulators or address generation.
- **Priority-Structured I/O System**  
Daisy-chained grant signals provide a priority-structured I/O system.
- **Vectored Interrupts**  
Fast interrupt response without device polling.
- **Single and Double Operand Instructions**  
Powerful and convenient set of programming instructions.
- **Power-Fail/Auto Restart**  
Whenever DC power sequencing signals indicate an impending AC power loss, a microcoded power-fail sequence is initiated. When power is restored, the processor can automatically return to the run state. Four options are available for power up sequencing.

## 1.4 SYSTEM ARCHITECTURE

A complete and powerful microcomputer system can be configured by utilizing the KD11-F microcomputer, appropriate memory, I/O devices, and interconnection hardware. The LSI-11 bus (implemented on the H9270 card guide backplane assembly) is the interface which enables a complete system to be configured.

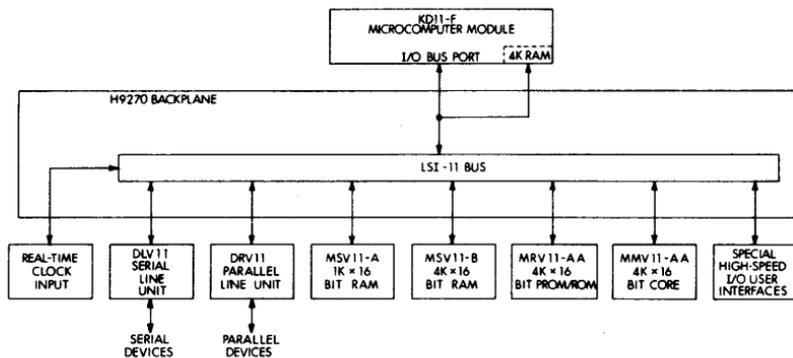


Figure 1-1 LSI-11 System Configuration

All LSI-11 modules connected to this common bus structure receive the same interface signals. LSI-11 bus control and data lines are bidirectional, open-collector lines which are asserted when low. All transactions on the bus are asynchronous. The bus is composed of 16 multiplexed data/address lines, six data transfer control lines, six system control lines, and five interrupt and direct memory access (DMA) control lines.

Interrupt and DMA are implemented with two daisy-chained grant signals which provide a priority-structured I/O system. The highest priority device is the module electrically located closest to the microcomputer module. Only when a device is not asserting a request does it pass grant signals to lower priority devices.

The LSI-11 bus provides a vectored interrupt interface for any device. Device polling is not required in processing interrupt requests. When an interrupting device receives a grant, the device passes to the processor an interrupt vector which points to a new processor status word and the starting address of an interrupt service routine for the device.

The H9270 backplane assembly contains all of the wiring for the LSI-11 bus, plus standard power and system control wires.

## 1.5 MICROCOMPUTER

The microcomputer connected to the LSI-11 bus controls the time allocation of the LSI-11 bus for peripherals and performs arithmetic and logic operations and instruction decoding. It contains multiple high-speed, general-purpose registers which can be used as accumulators, address pointers, index registers, and other specialized functions. The processor does both single and double operand addressing and handles both 16-bit word and 8-bit byte data. The bus permits data transfers directly between I/O devices and memory without disturbing the processor registers.

The microcomputer processor is implemented with four LSI 40-pin chips. The four chips are the control chip, the data chip, and two microm (microcode read-only memory) chips.

### Control Chip

This chip provides the microinstruction address sequence, for the microm and control for the data access port. It contains the following features:

- Programmable Translation Array (PTA)—Provides a decoding mechanism for generating microinstruction addresses from macroinstructions.
- Location Counter (LC)—Stores the address in the microm from which accesses are being made.
- Return Register (RR)—Used to hold a microsubroutine return address.
- Data Transfer Control Logic—Provides control and timing signals for data/address port.
- Interrupt Logic—Provides control over three internal flags for the processor and four external flags for the system.

### Data Chip

The data chip incorporates the paths, registers, and logic to execute microinstructions. It offers the following features:

- Register File—Provides multiple registers for storage of frequently required data.
- Arithmetic and Logic Unit (ALU)—Performs the arithmetic and logic operations necessary for instruction execution.
- Condition Flags Logic—Monitors the status of the result from the ALU section.
- Data/Address Port—Provides access to the data address lines.

### Microm Chips

The microm chips provide storage of the microcode for emulation of the basic PDP-11/35,40 instruction set, resident ODT (octal debugging technique) firmware, resident ASCII/console routine, and bootstrap.

An optional fifth chip (third microm) can be added to the LSI-11 processor, via a socket available on the microcomputer module, to extend the instruction set to include fixed and floating point arithmetic instructions.

### 1.5.1 General Registers

The LSI-11 central processor module contains eight 16-bit general-purpose registers that can perform a variety of functions. These registers can serve as accumulators, index registers, autoincrement registers, auto-decrement registers, or as stack pointers for temporary storage of data. Arithmetic operations can be from one general register to another, from one memory location or device register to another, or between memory locations or a device register and a general register. The following illustration identifies the eight 16-bit general registers R0 through R7.

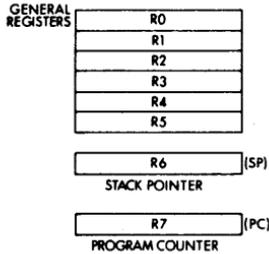


Figure 1-2 General Register Identification

Registers R6 and R7 in the LSI-11 are dedicated. R6 normally serves as the Stack Pointer (SP) and contains the location (address) of the last entry in the stack. Register R7 serves as the processor's Program Counter (PC) and contains the address of the next instruction to be executed. It is normally used for addressing purposes only and not as an accumulator. Register operations are internal to the processor and do not require bus cycles (except for instruction fetch); all memory and peripheral device data transfers do require bus cycles and longer execution time. Thus, general registers used for processor operations result in faster execution times. The bus cycles required for memory and device references are described below.

#### Bus Cycles

The bus cycles (with respect to the processor) are:

DATI	Data word transfer in	Equivalent to Read operation
DATIO	Data word transfer in, followed by word transfer out	Equivalent to Read/Modify Write
DATIOB	Data word transfer in, followed by byte transfer out	Equivalent to Read/Modify Write
DATO	Data word transfer out	Equivalent to Write operation
DATOB	Data byte transfer out	Equivalent to Write operation

Every processor instruction requires one or more bus cycles. The first operation required is a DATI, which fetches an instruction from the location addressed by the Program Counter (R7). If no further operands are referenced in memory or in an I/O device, no additional bus cycles are required for instruction execution. If memory or a device is referenced, however, one or more additional bus cycles are required.

Note the distinction between interrupts and DMA operations: Interrupts, which may change the state of the processor, can occur only between processor instructions; DMA operations can occur between individual bus cycles since these operations do not change the state of the processor.

### Addressing Memory and Peripherals

The maximum direct address space of the LSI-11 is 32K 16-bit words of memory. LSI-11's memory locations and peripheral device registers are addressed in precisely the same manner. The upper 4096 addresses (28K-32K) are usually reserved by convention for peripheral device addressing. However, the user does not need to dedicate the entire 4K space to I/O; he can implement only what he needs.

An LSI-11 word is divided into a high byte and a low byte as shown below.

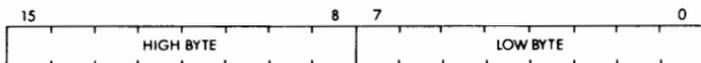


Figure 1-3 High and Low Byte

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

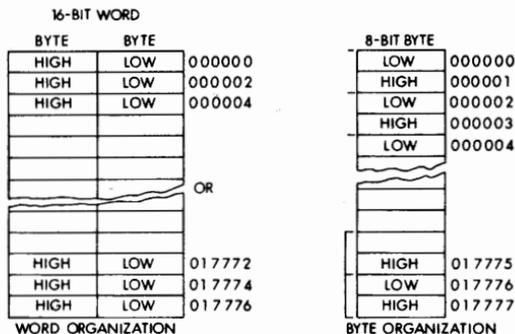


Figure 1-4 Word and Byte Addresses for First 4K Bank

Certain memory locations have been reserved by convention for interrupt and trap handling and peripheral device registers. Addresses from 0 to  $376_8$  are usually reserved for trap and device interrupt vector locations. Several of these are reserved in particular for system (processor initiated) traps.

### 1.5.2 The Processor Status Word (PSW)

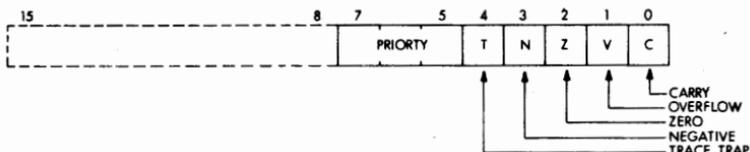


Figure 1-5 Processor Status Word (PSW)

The Processor Status Word (PSW) contains information on the current processor status. This information includes the current processor priority, the condition codes describing the arithmetic or logical results of the last instruction, and an indicator for detecting the execution of an instruction to be trapped during program debugging. The PS word format is shown above. Certain instructions allow programmed manipulation of condition code bits and loading or storing (moving) the PSW. The two instructions for explicitly accessing the PSW are described in Chapter 4.

#### Priority Interrupt Bit

The processor operates with interrupt priority PSW bit 7 asserted (1) or cleared (0). When PSW bit 7 = 1, an external device cannot interrupt the processor with a request for service. The processor must be operating at PSW bit 7 = 0 for the device's request to take effect. As compared to other PDP-11's, the LSI-11 operates at 1 line multi level priority.

#### Condition Codes

The condition codes contain information on the result of the last CPU operation. The bits are set as follows: (The bits are set after execution of all arithmetic or logical single operand or double operand instructions.)

Z = 1, if the result were zero

N = 1, if the result were negative

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

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

#### Trap (T Bit)

The program can only set or clear the trap bit (T) by popping a new PSW off the stack. When set, a processor trap will occur through location 14 at completion of the current instruction execution, and a new processor status word will be loaded from location 16. This T bit is especially useful in debugging programs as an efficient method of installing break-points.

### 1.5.3 Instruction Set

Implementing the PDP-11 instruction repertoire in the LSI chip set permits the user to take advantage of Digital Equipment Corporation's years of experience with the PDP-11 family—more than 17,000 units installed, with all associated application notes, software, documentation, training, reliability, customer references, and the DECUS library of application programs.

The instruction complement uses the flexibility of the general-purpose registers to provide more than 400 powerful hard-wired instructions—the most comprehensive and powerful instruction repertoire of any computer in the 16-bit class. Unlike conventional 16-bit computers, which usually have three classes of instructions (memory reference instructions, operate or accumulator control instructions, and I/O instructions), all data manipulation operations in the LSI-11 are accomplished with one set of instructions. Since peripheral device registers can be manipulated as flexibly as memory by the central processor, instructions that are used to manipulate data in memory can be used equally well for data in peripheral device registers. For example, data in an external device register can be tested or modified directly by the CPU without bringing it into memory or disturbing the general registers. One can add or compare data logically or arithmetically in a device register.

The basic order code of the LSI-11 uses both single and double operand address instructions for words or bytes. The LSI-11 therefore performs very efficiently in one step such operations as adding or subtracting two operands or moving an operand from one location to another.

#### LSI-11 Approach

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

#### Conventional Approach

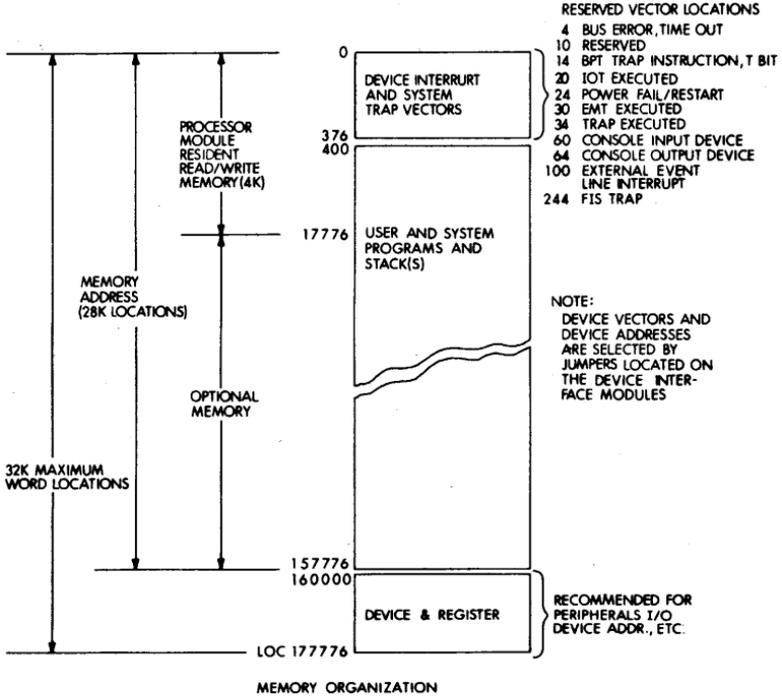
LDA A	Load contents of memory location A into accumulator
ADD B	Add contents of memory location B to accumulator
STA B	Store result at location B

#### Addressing

Much of the power of the LSI-11 is derived from its wide range of addressing capabilities. LSI-11 addressing modes include sequential forward or backward addressing, address indexing, indirect addressing, 16-bit word addressing, 8-bit byte addressing, and stack addressing. *Variable-length instruction formatting allows a minimum number of words to be used for each addressing mode. The result is efficient use of program storage space.*

### 1.6 LSI-11 MEMORY ORGANIZATION

The LSI-11 processor organization and addressing, register, memory, and device addresses are shown.



#### NOTE

There is 32K of users memory space available; however 0-28K is recommended for memory address locations, and 28K-32K for peripherals I/O device addresses, etc.

Figure 1-6 Memory Organization

## 1.7 LSI-11 BUS

The LSI-11 bus is a simple, fast, easy-to-use interface between LSI-11 modules. All LSI-11 modules connected to this common bidirectional bus structure receive the same interface signal lines. A typical system application in which the processor module, memory modules, and peripheral device interface modules are connected to the bus is shown in Figure 1-7.

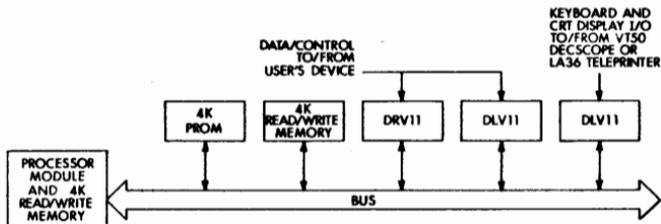


Figure 1-7 Typical Bus Application

Bus data and control lines are bidirectional open-collector lines that are asserted low. The bus is comprised of 16 data/address lines (BDAL 0-15), and 17 control/synchronization signal lines, and system function lines.

Control signal lines include two daisy-chained grant signals (four signal pins), which provide a priority-structured I/O system. The highest priority device is the module located electrically closest to the microcomputer module. Higher priority devices pass a grant signal to lower priority devices only when not requesting service. For example, "Module A," shown in figure 1-8, is the highest priority device, and is capable of interrupting processor operation and/or executing DMA transfers. Modules B and C have lower priorities, respectively. Module B can receive a grant signal when Module A is not asserting a request. Similarly, Module C can receive a grant signal when both Modules A and B are not asserting a request.

Both 16-bit address and 16-bit data words (or data bytes) are multiplexed over the 16 BDAL lines of the LSI-11 bus. For example, during a programmed data transfer, the processor will assert an address on the bus for a fixed time.

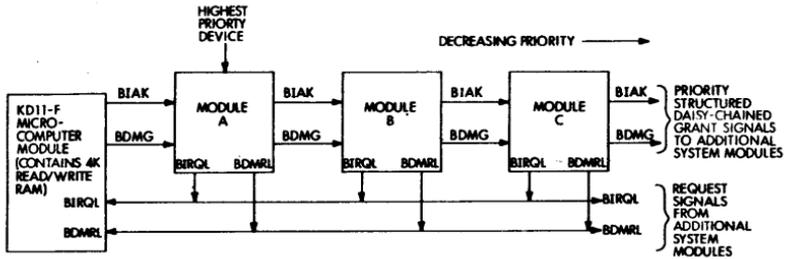


Figure 1-8 Bus Priority Structure

After the address time has been completed, the processor initiates the programmed input or output data transfer. The actual data transfer is asynchronous and requires a reply from the addressed device; bus synchronization and control signals provide this function.

The processor module is capable of driving six device slots (double-height) along the bus without additional termination, as provided with the H9270 backplane. Devices or memory can be installed in any location along this bus, as long as the desired priority order of the devices is maintained. Position 1 (figure 1-9) has the highest priority, position 6 the lowest.

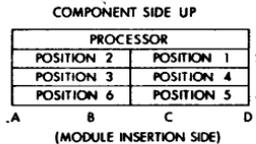


Figure 1-9 Devices Priority Guide

The bus protocol allows for a vectored interrupt by the device. Hence, device polling is not required in interrupt processing routines. This results in a considerable savings in processing time when many devices requiring interrupt service are interfaced along the bus. When an interrupting device receives an interrupt grant signal, the device passes to the processor, an interrupt vector. The vector points to two addresses which contain a new processor status word and the starting address of the interrupt service routine for the particular device.

One bus signal line (BEVNT) functions as an external event interrupt line via the processor module. This signal line can be connected to a 60 Hz line frequency source, and can be used as a real-time interrupt. A wire wrap connection on the processor module enables or inhibits this function. When enabled, the device connected to this line has the highest interrupt priority external to the processor. Interrupt vector  $100_8$  is reserved for this function, and an interrupt request via the external event line causes new PC and PS words to be loaded from locations  $100_8$  and  $102_8$ .

### **1.7.1 Bidirectional Lines**

With bidirectional and asynchronous communications on the LSI-11 bus, devices can send, receive, and exchange data at their own rates. The bidirectional nature of the bus allows utilization of common bus interfaces for different devices, and simplifies the interface design.

### **1.7.2 Master Slave Relation**

Communication between two devices on the bus is in the form of a master-slave relationship. At any point in 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." A typical example of this relationship is the processor, as master, fetching an instruction from memory (which is always a slave). Another example is a DMA device interface, as master, transferring data to memory, as slave. Bus master control is dynamic. The bus arbitrator on the processor module, for example, may pass bus control to a DMA device. The DMA device, as master, could then communicate with a slave memory bank.

Since the LSI-11 bus is used by the processor and all I/O devices, there is a priority structure to determine which device gets control of the bus. Every device on the LSI-11 bus which is capable of becoming bus master is assigned a priority according to its position along the bus. When two devices which are capable of becoming a bus master request use of the bus simultaneously, the device with the higher priority position will receive control.

### **1.7.3 Interlocked Communication**

Data transfer on the LSI-11 bus is interlocked so that communication is independent of the physical bus length and the response time of the slave device. The asynchronous operation precludes the need for synchronizing with, and waiting for, clock impulses. Thus, each device is allowed to operate at the maximum possible speed.

Full 16-bit words or 8-bit bytes of information can be transferred on the bus between a master and a slave. The information can be instructions, addresses, or data. This type of information transfer occurs when the processor, as master, is fetching instructions, operands, and data from memory, and storing the results into memory after execution of the instruction.

## CHAPTER 2

# SPECIFICATIONS

### 2.1 LSI-11 OPERATING SPECIFICATIONS

Tables 2-1 and 2-2 list the electrical and mechanical specifications of the LSI-11 options. All LSI-11 modules will operate at temperatures of 41°F to 122°F (5°C to 50°C) with a relative humidity of 10% to 95% (no condensation), with adequate airflow across the modules.

**TABLE 2-1 LSI-11 ELECTRICAL SPECIFICATIONS**

LSI-11 Nomenclature	Module Description	Power Requirements*	
		+5 V $\pm$ 5%**	+12 V $\pm$ 3%**
KD11-F	Microcomputer with 4K $\times$ 16 RAM	1.8A (Typ)	0.8A (Typ)
		2.4A (Max)	1.1A (Max)
MSV11-A	1K $\times$ 16 RAM	0.8A (Typ)	0.1A (Typ)
		1.8A (Max)	0.1A (Max)
MSV11-B	4K $\times$ 16 RAM	0.6A (Typ)	0.3A (Typ)
		1.1A (Max)	0.6A (Max)
MRV11-AA	4K $\times$ 16 PROM/ROM (OK implemented)	0.2A (Typ)	
		0.4A (Max)	
	(4K implemented)	2.8A (Typ)	
		4.1A (Max)	
MMV11-A	4K $\times$ 16 Core	3.0A (Stby) (Max)	0.2A (Stby) (Max)
		7.0A (Optg) (Max)	0.6A (Optg) (Max)
DLV11	Serial Line Unit	1.0A (Typ)	0.180A (Typ)
		1.6A (Max)	0.250A (Max)
DRV11	Parallel Line Unit	0.8A (Typ)	
		1.3A (Max)	

\*Preliminary

\*\*At the module connector

**For all Modules:**

**Electrical Input Logic Levels:**

Bus Low: 1.3 Vdc Max  
Bus High: 1.7 Vdc Min

**Electrical Output Logic Levels:**

Bus Low: 0.8 Vdc Max  
Bus High: 2.7 Vdc Min

**TABLE 2-2 MECHANICAL SPECIFICATIONS**

LSI-11 Nomenclature	Module Dimension (Tolerance $\pm 0.05''$ )
KD11-F	10.436 $\times$ 8.50 $\times$ 0.5"
KD11-J	10.436 $\times$ 8.50 $\times$ 0.9"
	10.436 $\times$ 8.50 $\times$ 0.5"
MSV11-A	5.187 $\times$ 8.50 $\times$ 0.5"
MSV11-B	5.187 $\times$ 8.50 $\times$ 0.5"
MRV11-AA	5.187 $\times$ 8.50 $\times$ 0.5"
MMV11-A	10.436 $\times$ 8.50 $\times$ 0.9"
DLV11	5.187 $\times$ 8.50 $\times$ 0.5"
DRV11	5.187 $\times$ 8.50 $\times$ 0.5"
H9270	11.15 $\times$ 11.0 $\times$ 2.80"

**2.2 PDP-11/03 OPERATING SPECIFICATIONS**

Table 2-3 lists the environmental and electrical specifications of the PDP-11/03.

**TABLE 2-3 PDP 11/03 OPERATING SPECIFICATIONS**

Temperature	41°F to 122°F (5°C to 50°C)
Relative Humidity	10% to 95% (no condensation)
Input Voltage:	
PDP-11/03-AA, BA	90-132 Vac, 115 Vac nominal, 47-63 Hz
PDP-11/03-AB, BB	180-264 Vac, 230 nominal, 47-63 Hz
Input Power:	
PDP-11/03-AA, AB, BA, BB	210 watts max at full load, 190 watts typical at full load

**2.3 H9270 BACKPLANE PACKAGING AND MOUNTING**

The H9270 Backplane (Figure 2-1) is designed to accept the KD11-F or KD11-J microcomputer and up to six I/O interface modules, or memory modules. Mounting of the H9270 backplane can be accomplished in any one of three planes, as shown in Figure 2-1.

**2.4 PDP-11/03 PACKAGING AND MOUNTING**

The PDP-11/03 shown in Figure 2-2 is offered in the following versions:

Designation	Description
PDP-11/03-AA	4K RAM Configuration (KD11-F), 115 Vac
PDP-11/03-AB	4K RAM Configuration (KD11-F), 230 Vac
PDP-11/03-BA	4K Core Configuration (KD11-J), 115 Vac
PDP-11/03-BB	4K Core Configuration (KD11-J), 230 Vac

The PDP-11/03 is designed with a removable front panel. Removing the front panel exposes the LSI modules and cables. This enables replacement or installation of a module from the front of the PDP-11/03. The 11/03 power supply is located on the right-hand side of the PDP-11/03 when viewed from the front. The power supply contains three front panel switches and indicators which are accessible through a cutout in the front panel. Therefore, when the front panel is removed, the lights and switches are still attached and functional.

The PDP-11/03 is designed to mount in a standard 19" cabinet (Figure 2-3). A standard 19" cabinet has two rows of mounting holes in the front, spaced  $18\frac{3}{16}$ " apart. The holes are located  $\frac{1}{2}$ " or  $\frac{5}{8}$ " apart from each other. Standard front panel increments are  $1\frac{3}{4}$ ".

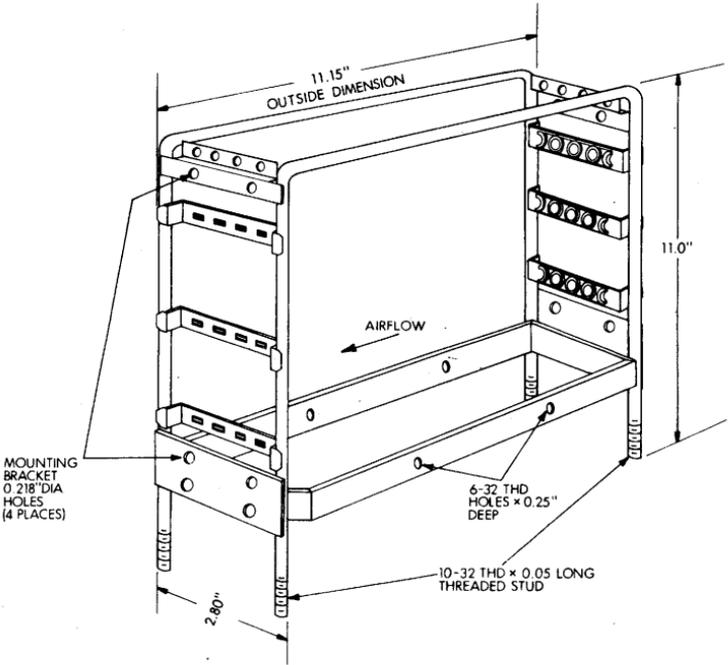
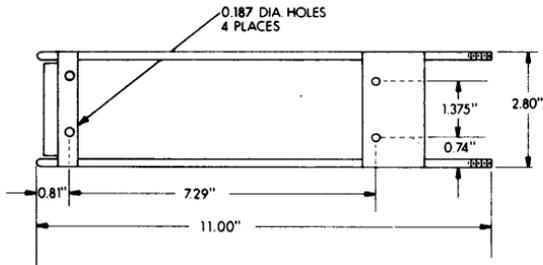
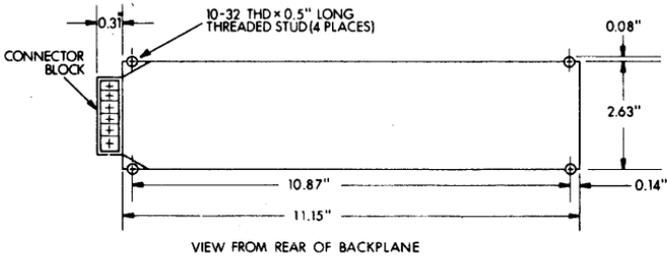


Figure 2-1 H9270 Backplane Mounting (SHT 1 of 2)

### SIDE MOUNTING



### REAR MOUNTING



### TOP AND BOTTOM MOUNTING

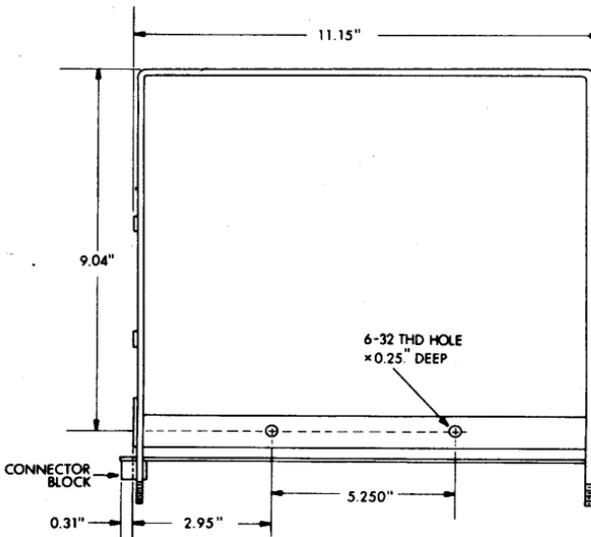


Figure 2-1 Backplane Mounting (SHT 2 of 2)

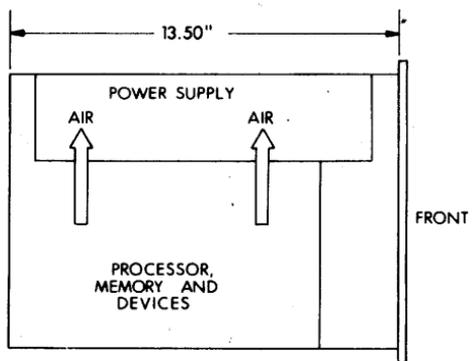
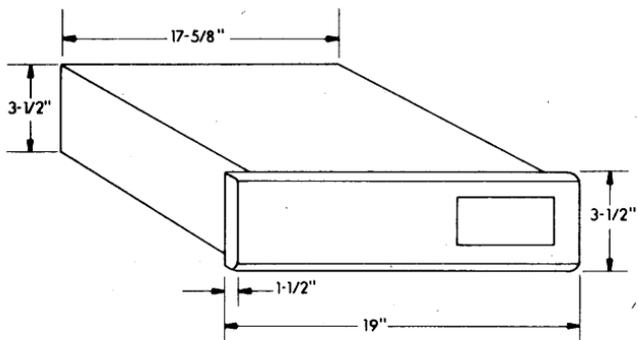


Figure 2-2 PDP-11/03 Assembly Unit

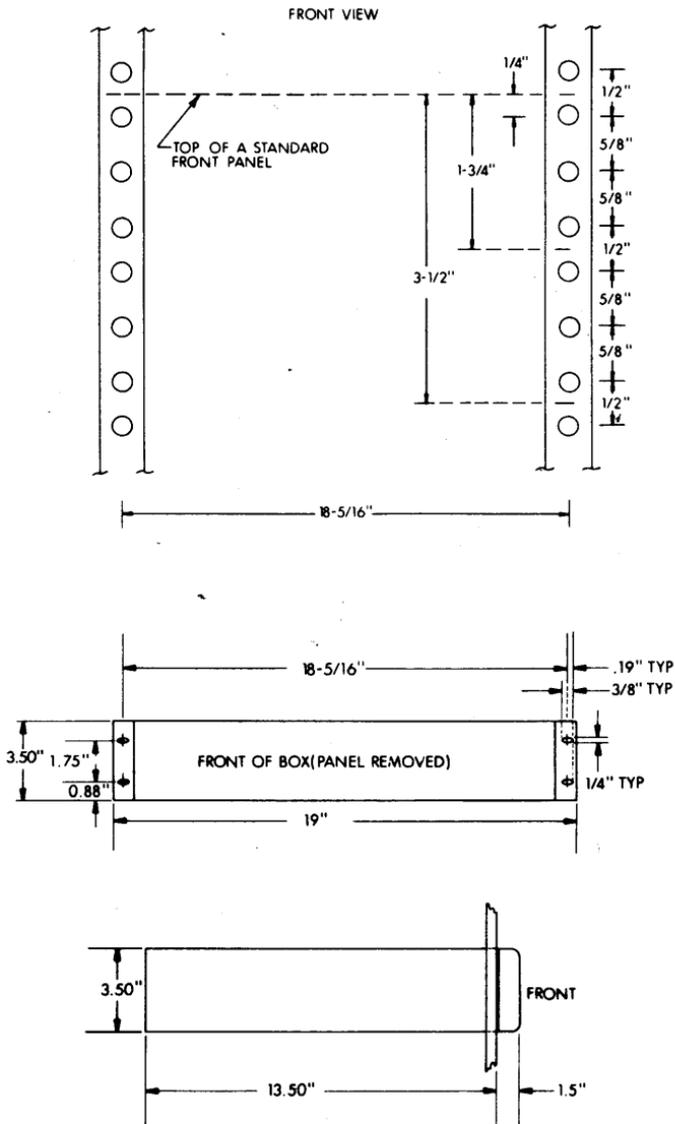


Figure 2-3 PDP-11/03 Cabinet Mounting

# ADDRESSING MODES

Data stored in memory must be accessed and manipulated. Data handling is specified by an LSI-11 instruction (MOV, ADD, etc.), which usually indicates:

- The function (operation code).
- A general-purpose register is to be used when locating the source operand and/or a general-purpose register to be used when locating the destination operand.
- An addressing mode (to specify how the selected register(s) is/are to be used).

A large portion of the data handled by a computer is usually structured (in character strings, arrays, lists, etc.). LSI-11's addressing modes provide for efficient and flexible handling of structured data.

The general registers may be used with an instruction in any of the following ways:

- As accumulators. The data to be manipulated resides within the register.
- As pointers. The contents of the register is the address of the operand, rather than the operand itself.
- As pointers which automatically step 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.
- As index registers. 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 LSI-11 feature, which should be considered in conjunction with the addressing modes, is the register arrangement:

- Six general-purpose registers (R0 – R5)
- A hardware Stack Pointer (SP), register (R6)
- A Program Counter (PC), register (R7)

Registers R0 through 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, 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 LSI-11 also has instruction addressing mode combinations that facilitate temporary data storage structures. This can be used for convenient handling of data which must be frequently accessed. This is known as stack manipulation. The register used to keep track of stack manipulation is known as the stack pointer. 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, it always points to the top of the stack.
- The hardware stack is used during trap or interrupt handling to store information allowing the processor to return to the main 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.

The next section is divided into seven major categories:

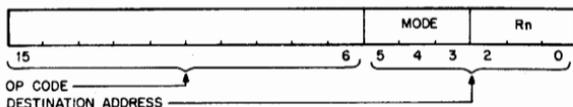
- **Single Operand Addressing**—One part of the instruction word specifies a register; the second part provides information for locating the operand.
- **Double Operand Addressing**—part of the instruction word specifies the registers; the remaining parts provide information for locating two operands.
- **Direct Addressing**—The operand is the contents 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 Register**—The PC is unique 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 Stack Pointer as General Register**—Can be used for stack operations.
- **Summary of Addressing Modes**

#### NOTE

Instruction mnemonics and address mode symbols are sufficient for writing assembly language programs. The programmer need not be concerned about conversion to binary digits; this is accomplished automatically by the assembler program.

### 3.1 SINGLE OPERAND ADDRESSING

The instruction format for all single operand instructions (such as clear, increment, test) is:



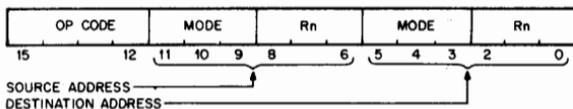
Bits 15 through 6 specify the operation code that defines the type of instruction to be executed.

Bits 5 through 0 form a six-bit field called the destination address field. This consists of two subfields:

- Bits 0 through 2 specify which of the eight general purpose registers is to be referenced by this instruction word.
- Bits 3 through 5 specify how the selected register will be used (address mode). Bit 3 is set to indicate deferred (indirect) addressing.

### 3.2 DOUBLE OPERAND ADDRESSING

Operations which imply two operands (such as add, subtract, move and compare) are handled by instructions that specify two addresses. The first operand is called the source operand, the second 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:



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.

Examples in this section and further in this chapter use the following sample LSI-11 instructions. A complete listing of the LSI-11 instructions is located in the appendix.

Mnemonic	Description	Octal Code
CLR	clear (zero the specified destination)	0050DD
CLRB	clear byte (zero the byte in the specified destination)	1050DD
INC	increment (add 1 to contents of destination)	0052DD
INCB	increment byte (add 1 to the contents of destination byte)	1052DD
COM	complement (replace the contents of the destination by their 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 their logical complement; each 0 bit is set and each 1 bit is cleared).	1051DD
ADD	add (add source operand to destination operand and store the result at destination address)	06SSDD

DD = destination field (6 bits)

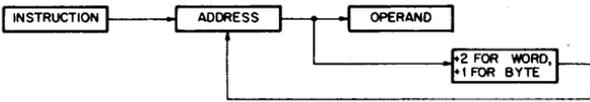
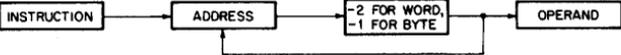
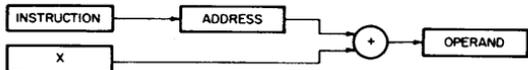
SS = source field (6 bits)

( ) = contents of

### 3.3 DIRECT ADDRESSING

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

#### DIRECT MODES

Mode	Name	Assembler Syntax	Function
0	Register	Rn	Register contains operand
			
2	Autoincrement	(Rn) +	Register is used as a pointer to sequential data then incremented
			
4	Autodecrement	-(Rn)	Register is decremented and then used as a pointer.
			
6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) are modified.
			

#### 3.3.1 Register Mode

OPR Rn

With register mode any of the general registers may be used as simple accumulators and the operand is 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.

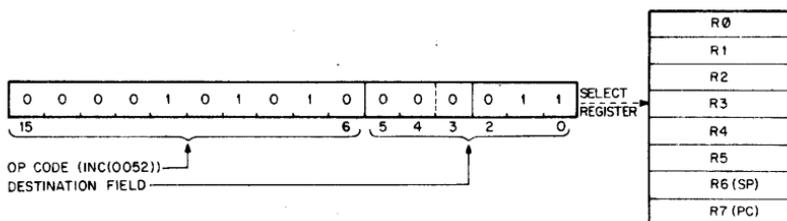
Registers are typically referred to by name as R0, R1, R2, R3, R4, R5, R6 and R7. However R6 and R7 are also referred to as SP and PC, respectively.

### Register Mode Examples

(all numbers in octal)

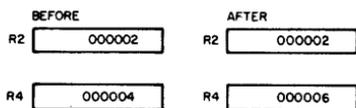
	Symbolic	Octal Code	Instruction Name
1.	INC R3	005203	Increment

Operation: Add one to the contents of general register 3



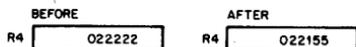
2.	ADD R2,R4	060204	Add
----	-----------	--------	-----

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



3.	COMB R4	105104	Complement Byte
----	---------	--------	-----------------

Operation: One's complement bits 0-7 (byte) in R4. (When general registers are used, byte instructions only operate on bits 0-7; i.e. byte 0 of the register)



### 3.3.2 Autoincrement Mode

OPR (Rn)+

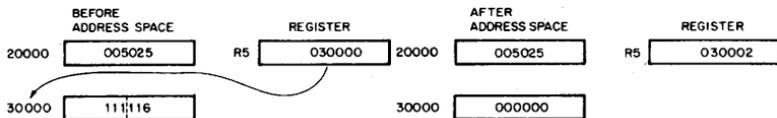
This mode provides for automatic stepping of a pointer through sequential elements of a table of operands. It assumes the contents of the selected general register to be the address of the operand. Contents of registers are stepped (by one for bytes, by two for words, always by two for R6 and R7) to 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

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

Operation:

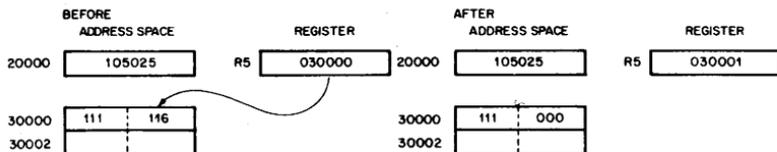
Use contents of R5 as the address of the operand. Clear selected operand and then increment the contents of R5 by two.



2.	CLRB (R5) +	105025	Clear Byte
----	-------------	--------	------------

Operation:

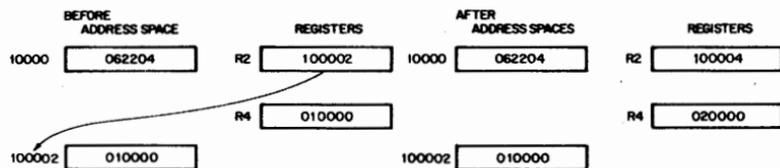
Use contents of R5 as the address of the operand. Clear selected byte operand and then increment the contents of R5 by one.



3. 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.



### 3.3.3 Autodecrement Mode (Mode 4)

OPR-(Rn)

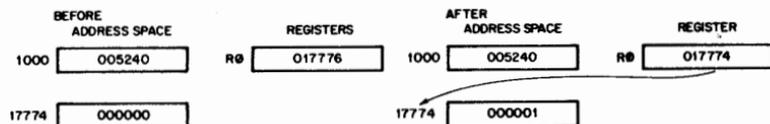
This mode is useful for processing data in a list in reverse direction. The contents of the selected general register are decremented (by two for word instructions, by one for byte instructions) and then used as the address of the operand. The choice of postincrement, predecrement features for the LSI-11 were not arbitrary decisions, but were intended to facilitate hardware/software stack operations.

#### Autodecrement Mode Examples

- |    | Symbolic | Octal Code | Instruction Name |
|----|----------|------------|------------------|
| 1. | INC-(R0) | 005240     | Increment        |

Operation:

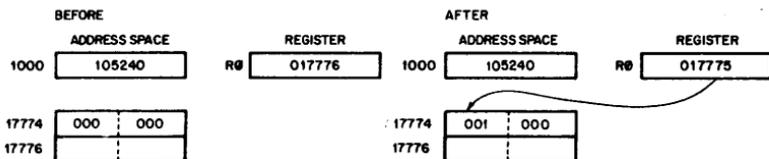
The contents of R0 are decremented by two and used as the address of the operand. The operand is incremented by one.



- |    |           |        |                |
|----|-----------|--------|----------------|
| 2. | INCB-(R0) | 105240 | Increment Byte |
|----|-----------|--------|----------------|

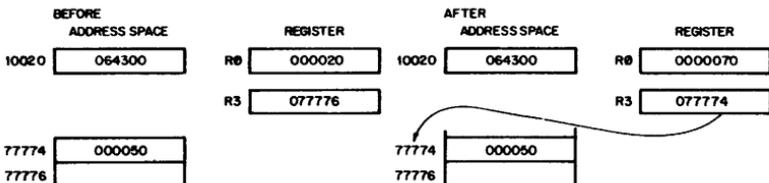
Operation:

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



3.        **ADD-(R3),R0        064300        Add**

**Operation:**                    The contents of R3 are decremented by 2 then used as a pointer to an operand (source) which is added to the contents of R0 (destination operand).



### 3.3.4 Index Mode (Mode 6)

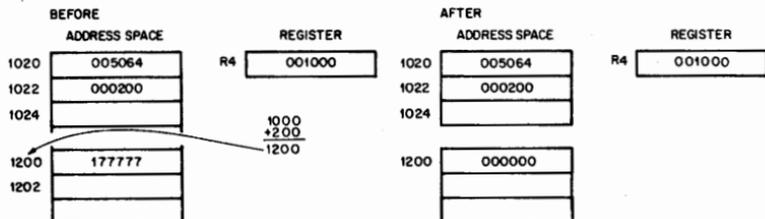
#### OPR X(Rn)

The contents of the selected general 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 and is located in the memory location following the instruction word and Rn is the selected general register.

#### Index Mode Examples

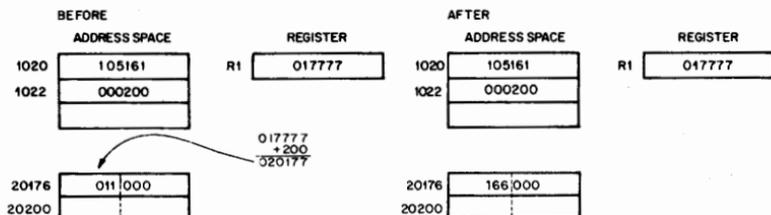
	Symbolic	Octal Code	Instruction Name
1.	CLR 200(R4)	005064 000200	Clear

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



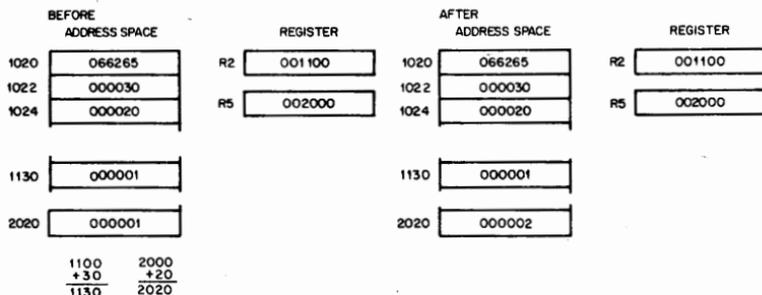
2. **COMB 200(R1) 105161 Complement Byte**  
**000200**

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



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

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



### 3.4 DEFERRED (INDIRECT) ADDRESSING

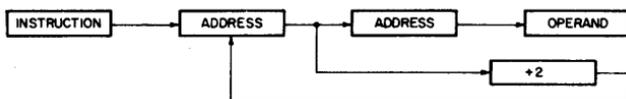
The four basic modes may also be used with deferred addressing. Whereas in the register mode the operand is the contents of the selected register. In the register deferred mode the contents of the selected register is the address of the operand.

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

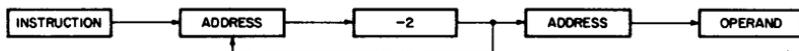
Mode	Name	Assembler' Syntax	Function
1	Register Deferred	@Rn or (Rn)	Register contains the address of the operand



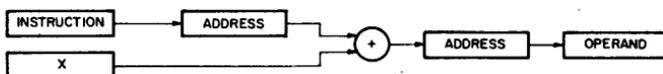
3	Autoincrement Deferred	@(Rn) +	Register is first used as a pointer to a word containing the address of the operand, then incremented (always by 2; even for byte instructions).
---	------------------------	---------	--



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.
---	------------------------	--------	---



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

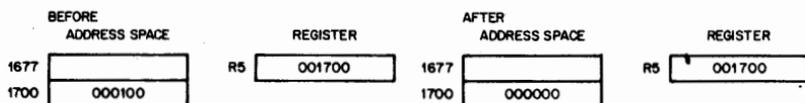


The following examples illustrate the deferred modes.

### Register Deferred Mode Example

Symbolic	Octal Code	Instruction Name
CLR @R5	005015	Clear

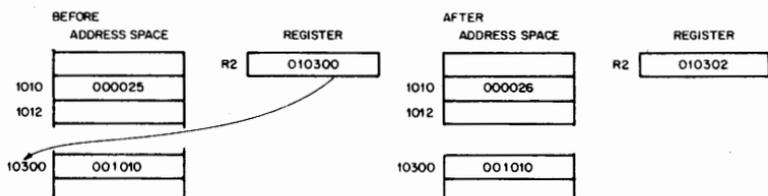
Operation: The contents of location specified in R5 are cleared.



### Autoincrement Deferred Mode Example (Mode 3)

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

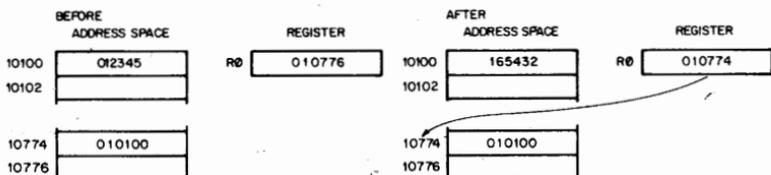
Operation: The contents of R2 are used as the address of the operand. Operand is increased by one. Contents of R2 is incremented by 2.



### Autodecrement Deferred Mode Example (Mode 5)

Symbolic	Octal Code	Complement
COM @-(R0)	005150	

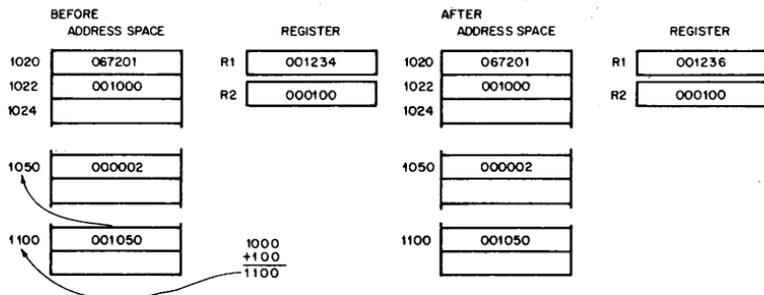
Operation: The contents of R0 are decremented by two and then used as the address of the address of the operand. Operand is one's complemented. (i.e. logically complemented)



### Index Deferred Mode Example (Mode 7)

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

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



### 3.5 USE OF THE PC AS A GENERAL REGISTER

Although Register 7 is a general purpose register, it doubles in function as the Program Counter for the LSI-11. 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 LSI-11 addressing modes. However, there are four of these modes with which 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, and 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 following the instruction) is the relative address for the address of the operand.

The reader should remember that the special PC modes are the same as modes described in 3.3 and 3.4, but the general register selected is R7, the program counter.

When a standard program is available for different users, it often is helpful to be able to load it into different areas of memory and run it there. LSI-11's 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.

### 3.5.1 Immediate Mode

OPR #n,DD

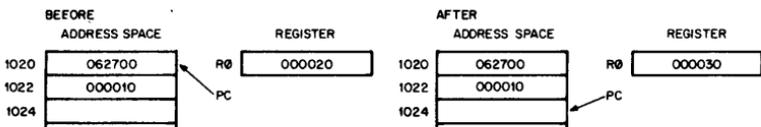
Immediate mode is equivalent to using the autoincrement mode with the PC. It provides time improvements for accessing constant operands by including the constant in the memory location immediately following the instruction word.

#### Immediate Mode Example

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 being incremented by two to point to the next instruction.



### 3.5.2 Absolute Addressing

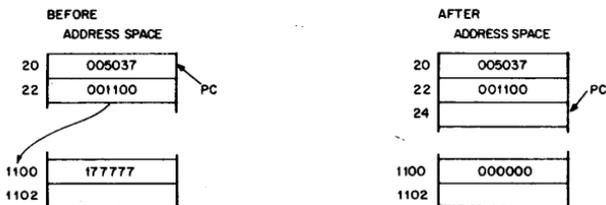
OPR @ #A

This mode 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

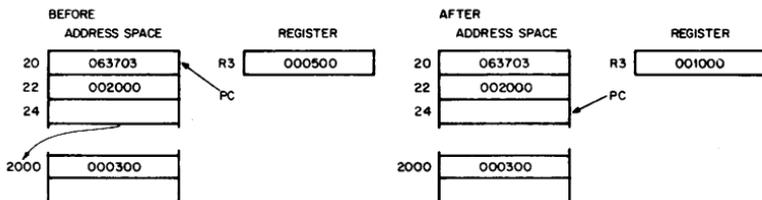
	Symbolic	Octal Code	Instruction Name
1.	CLR @ # 1100	005037 001100	Clear

Operation: Clear the contents of location 1100.



2.	ADD @ # 2000,R3	063703 002000
----	-----------------	------------------

Operation: Add contents of location 2000 to R3.



### 3.5.3 Relative Addressing

OPR A or OPR X (PC)  
 , where X is the location of A relative to the instruction.

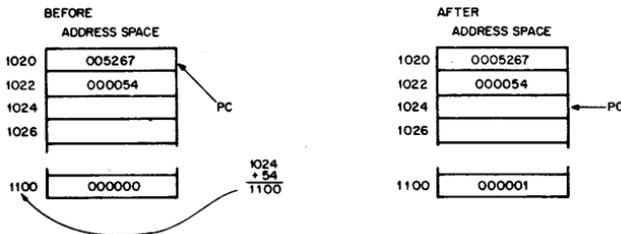
This mode is assembled as index mode using R7. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand, but the number which, when added to the (PC), becomes the address of the operand. This mode is useful for writing position independent code (see Chapter 5) 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.

#### Relative Addressing Example

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.



### 3.5.4 Relative Deferred Addressing

OPR @A or  
 OPR @X(PC), where x is location containing address of A, relative to the instruction.

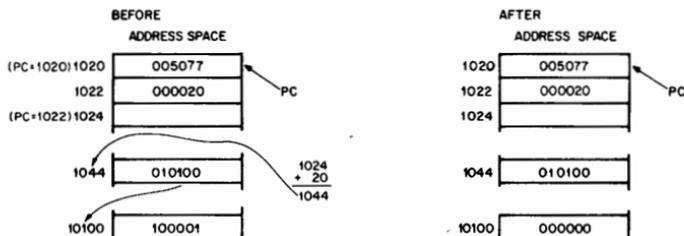
This mode is similar to the relative mode, except that the second word of the instruction, when added to the PC, contains the address of the address of the operand, rather than the address of the operand.

#### Relative Deferred Mode Example

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.



### 3.6 USE OF STACK POINTER AS GENERAL REGISTER

The processor stack pointer (SP, Register 6) is in most cases the general register used for the stack operations related to program nesting. Auto-decrement with Register 6 "pushes" data on to the stack and autoincrement with Register 6 "pops" data off the stack. Index mode with SP permits random access of items on 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.

### 3.7 SUMMARY OF ADDRESSING MODES

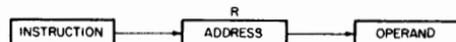
#### 3.7.1 General Register Addressing

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

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

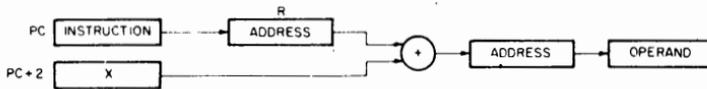


**Mode 1**                      **Register deferred**                      OPR (R)                      R contains address





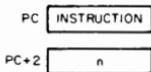
**Mode 7 Index deferred** OPR @X(R) (R) + X is address of address



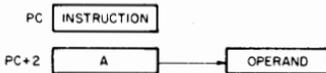
### 3.7.2 Program Counter Addressing

Register = 7

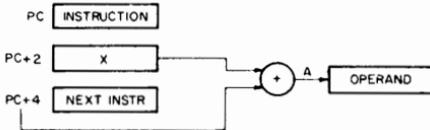
**Mode 2 Immediate** OPR #n Operand n follows instruction



**Mode 3 Absolute** OPR @#A Address A follows instruction

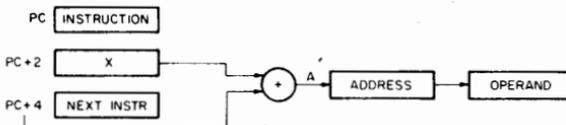


**Mode 6 Relative** OPR A  $\underbrace{PC + 4 + X}$  is address updated PC



**Mode 7 Relative deferred** OPR @A

$\underbrace{PC + 4 + X}$  is address of address updated PC





## INSTRUCTION SET

## 4.1 INTRODUCTION

The specification for each instruction includes the mnemonic, octal code, binary code, a diagram showing the format of the instruction, a symbolic notation describing its execution and the effect on the condition codes, a description, special comments, and examples.

**MNEMONIC:** This is indicated at the top corner of each page. When the word instruction has a byte equivalent, the byte mnemonic is also shown.

**INSTRUCTION FORMAT:** A diagram accompanying each instruction shows the octal op code, the binary op code, and bit assignments. (Note that in byte instructions the most significant bit (bit 15) is always a 1.)

**SYMBOLS:**

( ) = contents of

SS or src = source address

DD or dst = destination address

loc = location

← = becomes

↑ = "is popped from stack"

↓ = "is pushed onto stack"

∧ = boolean AND

v = boolean OR

⊕ = exclusive OR

~ = boolean not

Reg or R = register

B = Byte

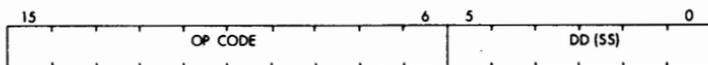
■ =  $\begin{cases} 0 & \text{for word} \\ 1 & \text{for byte} \end{cases}$

, = concatenated

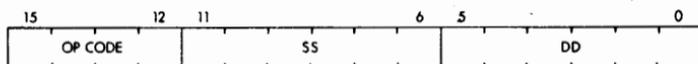
## 4.2 INSTRUCTION FORMATS

The following formats include all instructions used in the LSI-11. Refer to individual instructions for more detailed information.

- Single Operand Group (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, XOR)

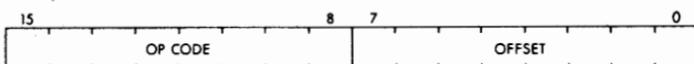


- Double Operand Group (BIT, BITB, BIC, BICB, BIS, BISB, ADD, SUB, MOV, MOVB, CMP, CMPB)

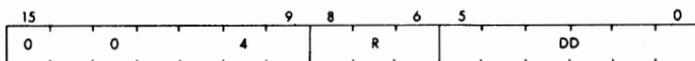


- Program Control Group

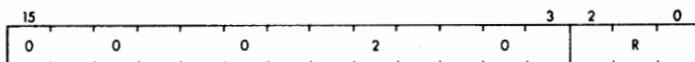
- Branch (all branch instructions)



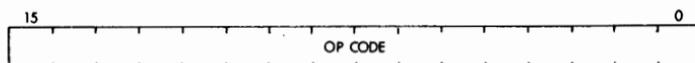
- Jump To Subroutine (JSR)



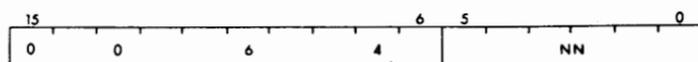
- Subroutine Return (RTS)



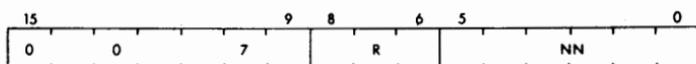
- Traps (break point, IOT, EMT, TRAP, BPT)



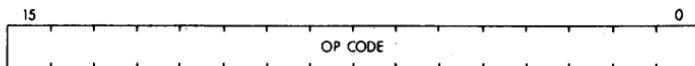
- Mark (MARK)



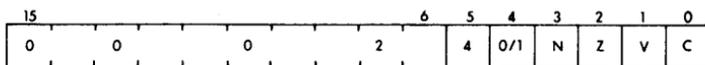
- Subtract 1 and branch (if = 0)(SOB)



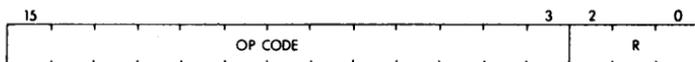
4. Operate Group (HALT, WAIT, RTI, RESET, RTT, NOP)



5. Condition Code Operators (all condition code instructions)

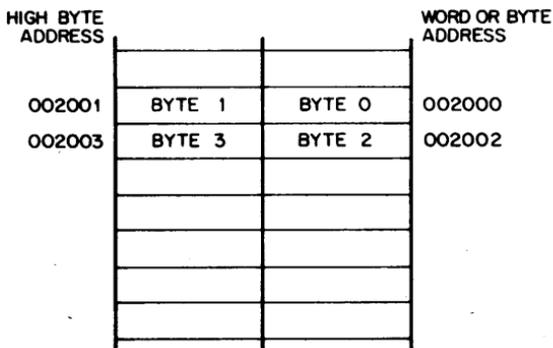


6. Fixed and Floating Point Arithmetic (optional EIS/FIS) (FADD, FSUB, FMUL, FDIV, MUL, DIV, ASH, ASHC)



**Byte Instructions**

The LSI-11 includes a full complement of instructions that manipulate byte operands. Since all LSI-11 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 LSI-11 to perform as either a word or byte processor. The numbering scheme for word and byte addresses in memory is:



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

Example:

Symbolic	Octal	
CLR	0050DD	Clear Word
CLRB	1050DD	Clear Byte

### 4.3 LIST OF INSTRUCTIONS

The LSI-11 instruction set is shown in the following sequence.

#### SINGLE OPERAND

Mnemonic	Instruction	Op Code	Page
<b>General</b>			
CLR(B)	clear dst .....	■050DD	4-6
COM(B)	complement dst .....	■051DD	4-7
INC(B)	increment dst .....	■052DD	4-8
DEC(B)	decrement dst .....	■053DD	4-9
NEG(B)	negate dst .....	■054DD	4-10
TST(B)	test dst .....	■057DD	4-11
<b>Shift &amp; Rotate</b>			
ASR(B)	arithmetic shift right .....	■062DD	4-13
ASL(B)	arithmetic shift left .....	■063DD	4-14
ROR(B)	rotate right .....	■060DD	4-15
ROL(B)	rotate left .....	■061DD	4-16
SWAB	swap bytes .....	0003DD	4-17
<b>Multiple Precision</b>			
ADC(B)	add carry .....	■055DD	4-19
SBC(B)	subtract carry .....	■056DD	4-20
SXT	sign extend .....	0067DD	4-21
<b>PS WORD OPERATORS</b>			
MFPS	move byte from PS .....	1067DD	4-22
MTPS	move byte to PS .....	1064SS	4-23

#### DOUBLE OPERAND

<b>General</b>			
MOV(B)	move source to destination .....	■1SSDD	4-25
CMP(B)	compare src to dst .....	■2SSDD	4-26
ADD	add src to dst .....	06SSDD	4-27
SUB	subtract src from dst .....	16SSDD	4-28
<b>Logical</b>			
BIT(B)	bit test .....	■3SSDD	4-30
BIC(B)	bit clear .....	■4SSDD	4-31
BIS(B)	bit set .....	■5SSDD	4-32
XOR	exclusive or .....	072RDD	4-33

## PROGRAM CONTROL

Mnemonic	Instruction	Op Code or Base Code	Page
<b>Branch</b>			
BR	branch (unconditional) .....	000400	4-35
BNE	branch if not equal (to zero) .....	001000	4-36
BEQ	branch if equal (to zero) .....	001400	4-37
BPL	branch if plus .....	100000	4-38
BMI	branch if minus .....	100400	4-39
BVC	branch if overflow is clear .....	102000	4-40
BVS	branch if overflow is set .....	102400	4-41
BCC	branch if carry is clear .....	103000	4-42
BCS	branch if carry is set .....	103400	4-43
<b>Signed Conditional Branch</b>			
BGE	branch is greater than or equal (to zero) .....	002000	4-45
BLT	branch if less than (zero) .....	002400	4-46
BGT	branch if greater than (zero) .....	003000	4-47
BLE	branch if less than or equal (to zero) ..	003400	4-48
<b>Unsigned Conditional Branch</b>			
BHI	branch if higher .....	101000	4-50
BLOS	branch if lower or same .....	101400	4-51
BHIS	branch if higher or same .....	103000	4-52
BLO	branch if lower .....	103400	4-53
<b>Jump &amp; Subroutine</b>			
JMP	jump .....	0001DD	4-54
JSR	jump to subroutine .....	004RDD	4-56
RTS	return from subroutine .....	00020R	4-58
MARK	mark .....	006400	4-59
SOB	subtract one and branch (if $\neq 0$ ) .....	077R00	4-61
<b>Trap &amp; Interrupt</b>			
EMT	emulator trap .....	104000—104377	4-63
TRAP	trap .....	104400—104777	4-64
BPT	breakpoint trap .....	000003	4-65
IOT	input/output trap .....	000004	4-66
RTI	return from interrupt .....	000002	4-67
RTT	return from interrupt .....	000006	4-68
<b>MISCELLANEOUS</b>			
HALT	halt .....	000000	4-71
WAIT	wait for interrupt .....	000001	4-72
RESET	reset external bus .....	000005	4-73
<b>RESERVED INSTRUCTIONS</b>			
		00021R	4-74
		00022N	4-75

## CONDITION CODE OPERATORS

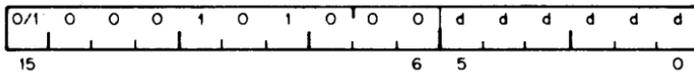
CLC	clear C .....	000241	4-76
CLV	clear V .....	000242	4-76
CLZ	clear Z .....	000244	4-76
CLN	clear N .....	000250	4-76
CCC	clear all CC bits .....	000257	4-76
SEC	set C .....	000261	4-76
SEV	set V .....	000262	4-76
SEZ	set Z .....	000264	4-76
SEN	set N .....	000270	4-76
SCC	set all CC bits .....	000277	4-76
NOP	no operation .....	000240	4-76

## 4.4 SINGLE OPERAND INSTRUCTIONS

### CLR CLRB

clear destination

■050DD



**Operation:** (dst) ← 0

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

**Description:** Word: Contents of specified destination are replaced with zeroes.  
Byte: Same

**Example:**

CLR R1

Before  
(R1) = 177777

After  
(R1) = 000000

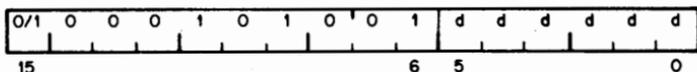
NZVC  
1111

NZVC  
0100

# COM COMB

complement dst

■051DD



**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:** 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  
 (R0) = 013333

After  
 (R0) = 164444

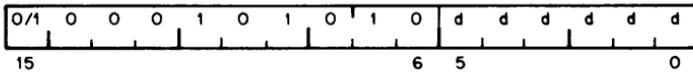
NZVC  
 0110

NZVC  
 1001

# INC INCB

increment dst

■052DD



**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 one to contents of destination  
 Byte: Same

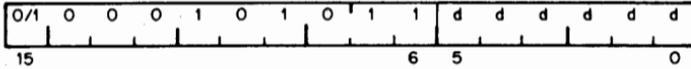
**Example:** INC R2

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

# DEC DEC B

decrement dst

■053DD



**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

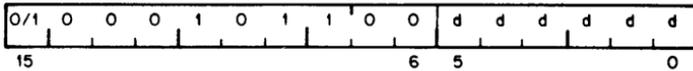
**Example:** DEC R5

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

# NEG NEGB

negate dst

■054DD



**Operation:** (dst) ← -(dst)

**Condition Codes:** N: set if the result is <0; cleared otherwise  
 Z: set if result is 0; cleared otherwise  
 V: set if the result is 10000; cleared otherwise  
 C: cleared if the result is 0; set otherwise

**Description:** Word: Replaces the contents of the destination address by its two's complement. Note that 10000 is replaced by itself (in two's complement notation the most negative number has no positive counterpart).  
 Byte: Same

**Example:**

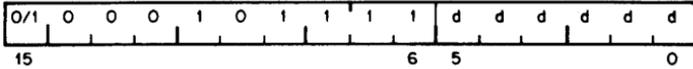
NEG R0

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

# TST TSTB

test dst

■057DD



**Operation:** (dst) ← (dst)

**Condition Codes:** N: set if the 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, contents of dst remains unmodified  
 Byte: Same

**Example:** TST R1

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

### **Shifts**

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 0 in shifts to the left. Bits shifted out of the C bit, as shown in the following examples, are lost.

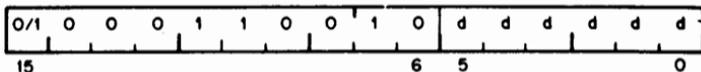
### **Rotates**

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

arithmetic shift right

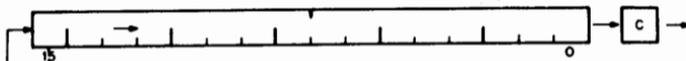
■062DD



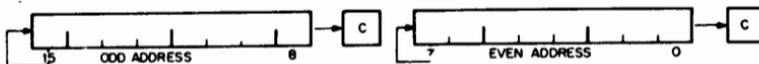
**Operation:** (dst) ← (dst) shifted one place to the right

**Condition Codes:** N: set if the high-order bit of the result is set (result < 0); cleared otherwise  
 Z: set if the result = 0; cleared otherwise  
 V: loaded from the Exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)  
 C: loaded from low-order bit of the destination

**Description:** 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 two.  
 Word:



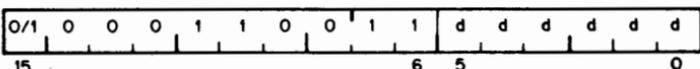
Byte:



# ASL ASLB

arithmetic shift left

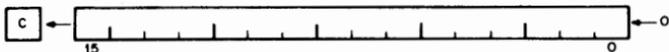
■063DD



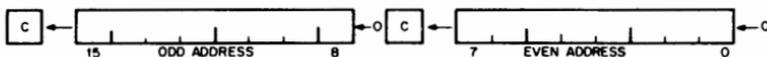
**Operation:** (dst) ← (dst) shifted one place to the left

**Condition Codes:** N: set if high-order bit of the result is set (result < 0); cleared otherwise  
 Z: set if the result = 0; cleared otherwise  
 V: loaded with the exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)  
 C: loaded with the high-order bit of the destination

**Description:** Word: Shifts all bits of the destination left one place. Bit 0 is loaded with an 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:



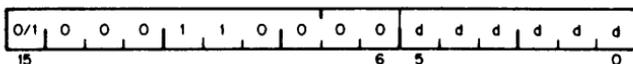
Byte:



# ROR RORB

rotate right

■060DD



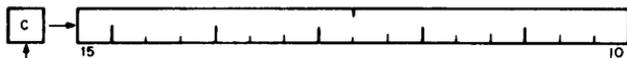
**Operation:** (dst) ← (dst)  
rotate right one place

**Condition Codes:** N: set if the high-order bit of the result is set (result < 0); cleared otherwise  
Z: set if all bits of result = 0; cleared otherwise  
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)  
C: loaded with the low-order bit of the destination

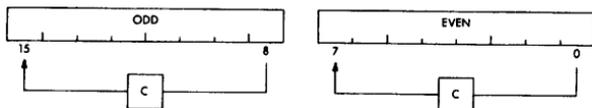
**Description:** Rotates all bits of the destination right one place. Bit 0 is loaded into the C-bit and the previous contents of the C-bit are loaded into bit 15 of the destination.  
Byte: Same

**Example:**

Word:



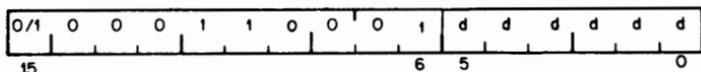
Byte:



# ROL ROLB

rotate left

■061DD



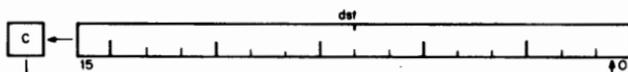
**Operation:**  $(dst) \leftarrow (dst)$   
rotate left one place

**Condition Codes:** N: set if the high-order bit of the result word is set (result < 0); cleared otherwise  
Z: set if all bits of the result word = 0; cleared otherwise  
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)  
C: loaded with the high-order bit of the destination

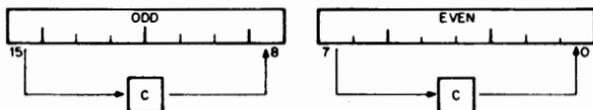
**Description:** Word: Rotate all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into Bit 0 of the destination.  
Byte: Same

**Example:**

Word:



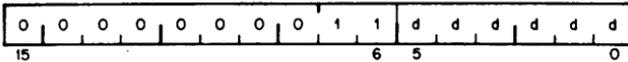
Bytes:



# SWAB

swap bytes

0003DD



**Operation:** Byte 1/Byte 0  $\leftarrow$  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 (destination must be a word address).

**Example:**

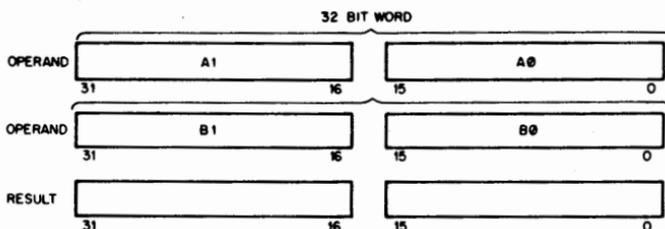
SWAB R1

Before	After
(R1) = 077777	(R1) = 177577
NZVC	NZVC
1111	0000

### Multiple Precision

It is sometimes necessary to do arithmetic on operands considered as multiple words or bytes. The LSI-11 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 = 3777777777

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

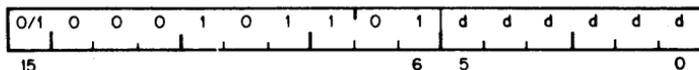
```
ADD R1,R2
ADC R3
ADD R4,R3
```

1. After (R1) and (R2) are added, 1 is loaded into the C bit
2. ADC instruction adds C bit to (R3); (R3) = 0
3. (R3) and (R4) are added
4. Result is 3777777776 or -2

# ADC ADCB

add carry

■055DD



**Operation:**  $(dst) \leftarrow (dst) + (C \text{ bit})$

**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:** Adds the contents of the C-bit into the destination. This permits the carry from the addition of the low-order words to be carried into the high-order result.  
Byte: Same

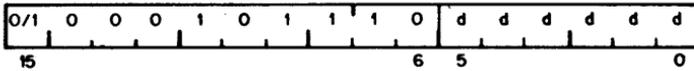
**Example:** Double precision addition may be done with the following instruction sequence:

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

# SBC SBCB

subtract carry

■056DD



**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

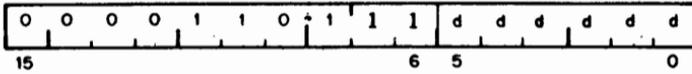
**Example:** Double precision subtraction is done by:

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

# SXT

sign extend

0067DD



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

**Condition Codes:** N: unaffected  
Z: set if N bit clear  
V: cleared  
C: unaffected

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

**Example:**

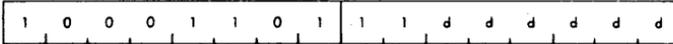
SXT A

	Before		After
(A) =	012345	(A) =	177777
	NZVC		NZVC
	1000		1000

# MFPS

Move byte From Processor Status word

1067DD



**Operation:** (dst) ← PSW  
dst lower 8 bits

**Condition Code**

**Bits:** N = set if PSW bit 7 = 1; cleared otherwise  
Z = set if PS <0:7> = 0; cleared otherwise  
V = cleared  
C = not affected

**Description:** The 8 bit contents of the PS are moved to the effective destination. If destination is mode 0, PS bit 7 is sign extended through upper byte of the register. The destination operand address is treated as a byte address.

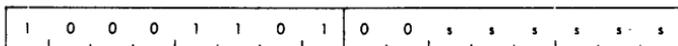
**Example:** MFPS R0

before	after
R0 [0]	R0 [000014]
PS [000014]	PS [000000]

# MTPS

Move byte To Processor Status word

1064SS



**Operation:** PSW  $\leftarrow$  (SRC)

**Condition Codes:** Set according to effective SRC operand bits 0-3

**Description:** The 8 bits of the effective operand replaces the current contents of the PSW. The source operand address is treated as a byte address.  
Note that the T bit (PSW bit 4) cannot be set with this instruction. The SRC operand remains unchanged. This instruction can be used to change the priority bit (PSW bit 7) in the PSW

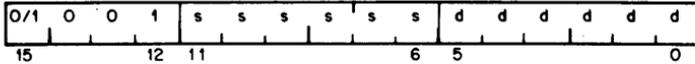
#### **4.5 DOUBLE OPERAND INSTRUCTIONS**

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

# MOV MOVB

move source to destination

■1SSDD



**Operation:** (dst) $\leftarrow$ (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. The 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)

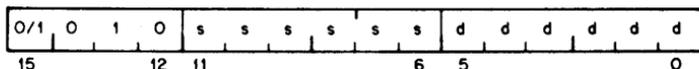
MOV R1,R3 ; performs an inter register transfer

MOVB @ #177562, @ #177566 ; moves a character from terminal keyboard buffer to terminal printer buffer.

# CMP CMPB

compare src to dst

■2SSDD



**Operation:** (src)-(dst)

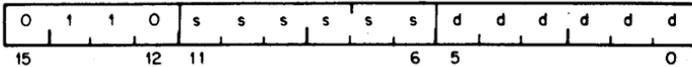
**Condition Codes:** N: set if result <0; cleared otherwise  
Z: set if result = 0; cleared otherwise  
V: set if there was arithmetic overflow; that is, operands were of opposite signs and the sign of the destination was the same as the sign of the result; cleared otherwise  
C: cleared if there was a carry from the most significant bit of the result; 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 unaffected. The only action is to set the condition codes. The compare is customarily followed by a conditional branch instruction. Note that unlike the subtract instruction the order of operation is (src)-(dst), not (dst)-(src).

# ADD

add src to dst

06SSDD



**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 most significant bit of the result; cleared otherwise

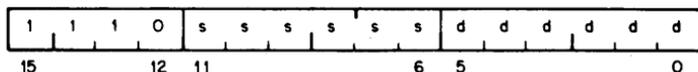
**Description:** Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. Two's complement addition is performed.  
Note: There is no equivalent byte Mode.

**Examples:** 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

subtract src from dst

16SSDD



**Operation:**  $(dst) \leftarrow (dst) - (src)$

**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 most significant bit of the result; 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".

**Example:**

SUB R1,R2

Before  
(R1) = 011111  
(R2) = 012345

After  
(R1) = 011111  
(R2) = 001234

NZVC  
1111

NZVC  
0000

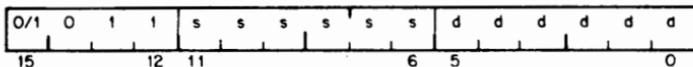
**Logical**

These instructions have the same format as the double operand arithmetic group. They permit operations on data at the bit level.

# BIT BITB

bit test

■3SSDD



**Operation:** (src) A (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 destination operands are affected. The BIT instruction may be used to test whether any of the corresponding bits that are set in the destination are 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 3 and 4 of R3 to see  
 if both are off

R3 = 0 000 000 000 011 000

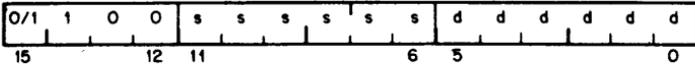
**Before**  
 NZVC  
 1111

**After**  
 NZVC  
 0001

# BIC BICB

bit clear

■4SSDD



**Operation:**  $(dst) \leftarrow \sim(src) \wedge (dst)$

**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 unaffected.

**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

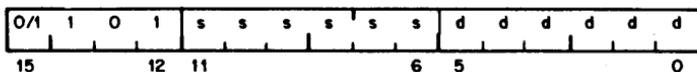
**After:** (R4)=0 000 000 001 000 001

# BIS

## BISB

bit set

■5SSDD



**Operation:** (dst) ← (src) v (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 "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

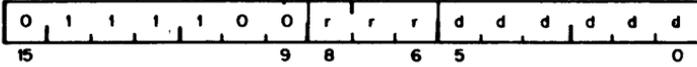
**Before:** (R0)=0 000 001 010 011 100  
 (R1)=0 000 001 001 001 001

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

# XOR

exclusive OR

074RDD



**Operation:** (dst) $\leftarrow$ Rv(dst)

**Condition Codes:** N: set if the result <0; cleared otherwise  
Z: set if result =0; cleared otherwise  
V: cleared  
C: unaffected

**Description:** The exclusive OR of the register and destination operand is stored in the destination address. Contents of register are unaffected. Assembler format is: XOR R,D

**Example:** XOR R0,R2

	Before	After
(R0)	= 001234	= 001234
(R2)	= 001111	= 000325
NZVC		NZVC
	1111	0001
<b>Before:</b>	(R0)=0 000 001 010 011 100	(R2)=0 000 001 001 001 001
<b>After:</b>	(R2)=0 000 000 011 010 101	

## 4.6 PROGRAM CONTROL INSTRUCTIONS

### 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:

- a) the branch instruction is unconditional
- b) 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 two 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. Similarly 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<sub>10</sub> words (400 bytes) from the current PC, and in the forward direction by 177<sub>10</sub> words (376 bytes) from the current PC.

The PDP-11 assembler handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form:

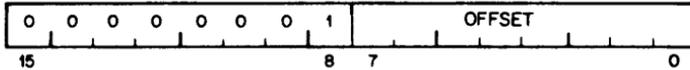
Bxx loc

Where "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 NO OP's.

# BR

branch (unconditional)

000400 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$

**Condition Codes:** Unaffected

**Description:** Provides a way of transferring program control within a range of  $-128_{10}$  to  $+127_{10}$  words with a one word instruction.

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

Updated PC = address of branch instruction + 2

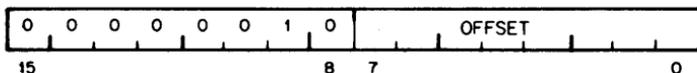
**Example:** With the Branch instruction at location 500, the following offsets apply.

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

branch if not equal (to zero)

001000 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $Z = 0$

**Condition Codes:** Unaffected

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

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

will branch to C if  $A \neq B$

and the sequence

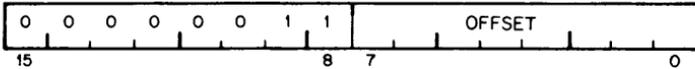
`ADD A,B ; add A to B`  
`BNE C ; Branch if the result is not equal to 0`

will branch to C if  $A + B \neq 0$

# BEQ

branch if equal (to zero)

001400 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $Z = 1$

**Condition Codes:** Unaffected

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

**Example:**

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

will branch to C if  $A = B$  (  $A - B = 0$  )  
and the sequence

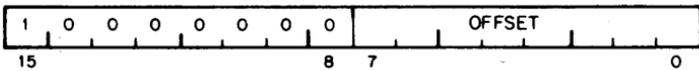
```
ADD  A,B           ; add A to B
BEQ  C             ; branch if the result = 0
```

will branch to C if  $A + B = 0$ .

# BPL

branch if plus

100000 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $N = 0$

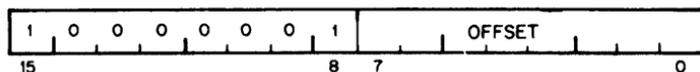
**Condition Codes:** Unaffected

**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

branch if minus

100400 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $N = 1$

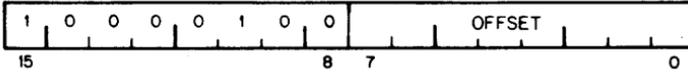
**Condition Codes:** Unaffected

**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

branch if overflow is clear

102000 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $V = 0$

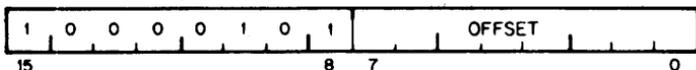
**Condition Codes:** Unaffected

**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

branch if overflow is set

102400 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $V = 1$

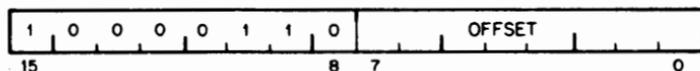
**Condition Codes:** Unaffected

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

## BCC

branch if carry is clear

103000 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $C = 0$

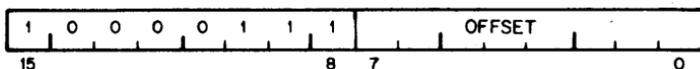
**Condition Codes:** Unaffected

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

## BCS

branch if carry is set

103400 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $C = 1$

**Condition Codes:** Unaffected

**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.

### 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 (two's complement) values.

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

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

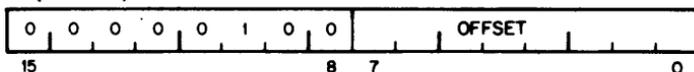
whereas in unsigned 16-bit arithmetic the sequence is considered to be

highest	177777
	.
	.
	.
	.
	000002
	000001
lowest	000000

## BGE

branch if greater than or equal  
(to zero)

002000 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $N \vee V = 0$

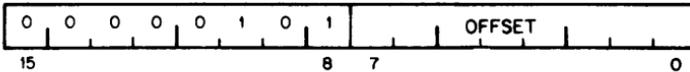
**Condition Codes:** Unaffected

**Description:** Causes a branch if N and V are either both clear or both set. BGE is the complementary operation to BLT. Thus BGE 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 zero result.

# BLT

branch if less than (zero)

002400 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $N \vee V = 1$

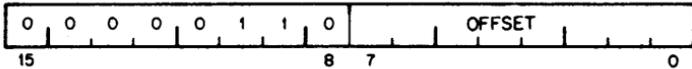
**Condition Codes:** Unaffected

**Description:** Causes a branch if the "Exclusive Or" of the N and V bits are 1. 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 zero (without overflow).

# BGT

branch if greater than (zero)

003000 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $Z \vee (N \neq V) = 0$

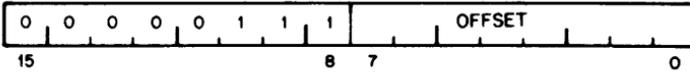
**Condition Codes:** Unaffected

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

# BLE

branch if less than or equal (to zero)

003400 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $Z \vee (N \vee V) = 1$

**Condition Codes:** Unaffected

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

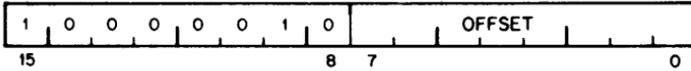
**Unsigned Conditional Branches**

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

# BHI

branch if higher

101000 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $C=0$  and  $Z=0$

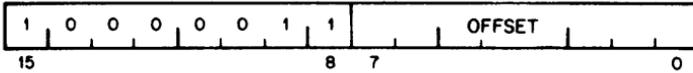
**Condition Codes:** Unaffected

**Description:** Causes a branch if the previous operation caused neither a carry nor a zero 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



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $C \vee Z = 1$

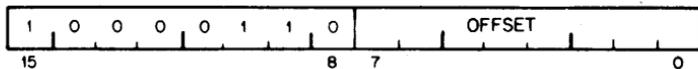
**Condition Codes:** Unaffected

**Description:** Causes a branch if the previous operation caused either a carry or a zero result. BLOS is the complementary operation to 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



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $C = 0$

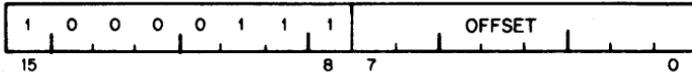
**Condition Codes:** Unaffected

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

# BLO

branch if lower

103400 Plus offset



**Operation:**  $PC \leftarrow PC + (2 \times \text{offset})$  if  $C = 1$

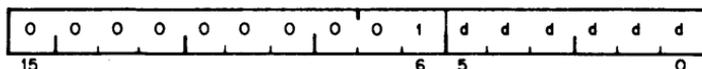
**Condition Codes:** Unaffected

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

# JMP

jump

0001DD



**Operation:** PC ← (dst)

**Condition Codes:** unaffected

**Description:** JMP provides more flexible program branching than provided with the branch instructions. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes, with the exception of register mode 0. Execution of a jump with mode 0 will cause an "illegal instruction" condition, and will cause the CPU to trap to vector address 4. (Program control cannot be transferred to a register.) Register deferred mode is legal and will cause program control to be transferred to the address held in the specified register. Note 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:**

	JMP	FIRST	; Transfers to First
	.....		
<b>First:</b>	.....		
	JMP	@LIST	; Transfers to location pointed to at LIST
	.....		
<b>List:</b>	FIRST	; pointer to FIRST	
	JMP	@(SP)+	; Transfer to location pointed to by the top of the stack, and remove the pointer from the stack

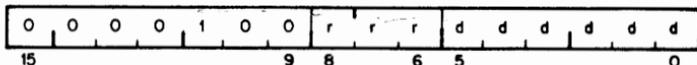
**Subroutine Instructions**

The subroutine call in the PDP-11 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, thus providing for reentrancy. This allows one copy of a subroutine to be shared among several interrupting processes.

# JSR

jump to subroutine

004RDD



- Operation:**
- $\uparrow(\text{SP}) \leftarrow \text{reg}$  (push reg contents onto processor stack)
  - $\text{reg} \leftarrow \text{PC}$  (PC holds location following JSR; this address now put in reg)
  - $\text{PC} \leftarrow (\text{dst})$  (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 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 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.

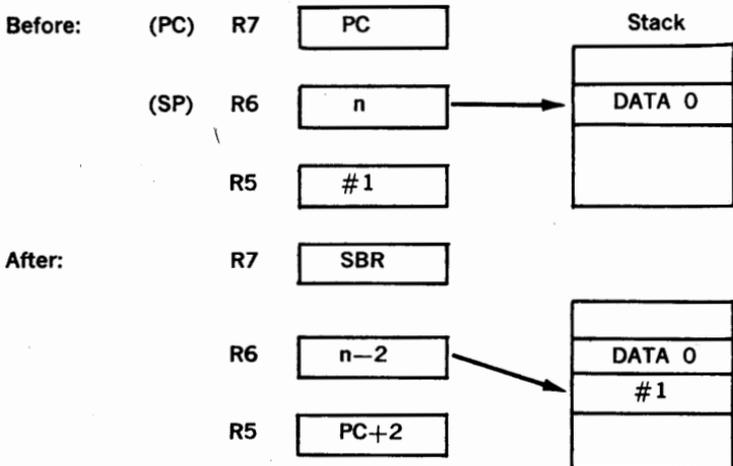
JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general registers. The SP and the PC are the only registers that may be modified by this call.

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

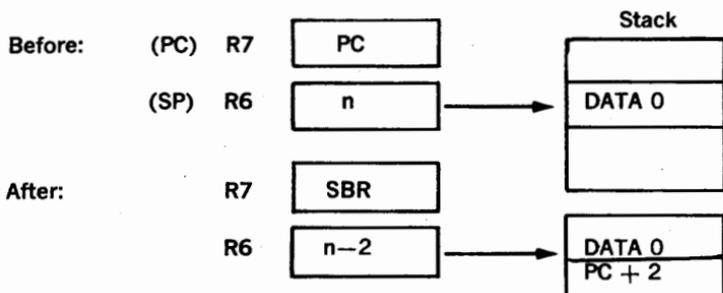
Return from a subroutine is done by the RTS instruction. RTS reg loads the contents of reg into the PC and pops the top element of the processor stack into the specified register.

**Example:**

JSR R5, SBR



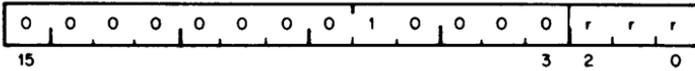
JSR PC, SBR



# RTS

return from subroutine

00020R

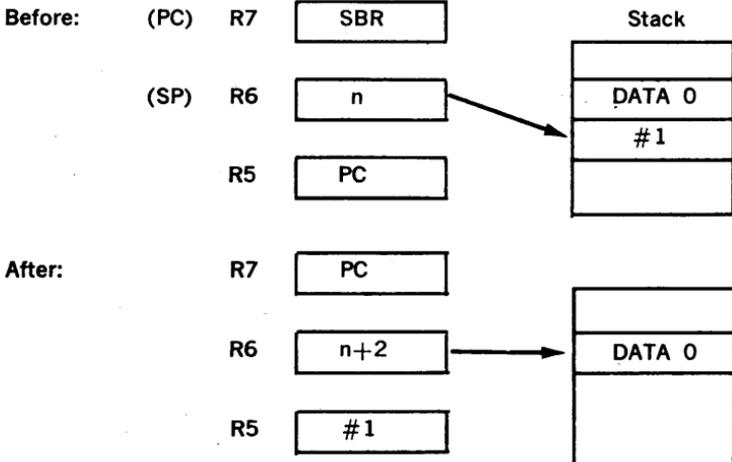


**Operation:** PC ← (reg)  
(reg) ← (SP) ↑

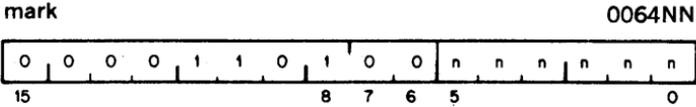
**Description:** Loads contents of reg into PC and pops the top element of the processor stack into the specified register. Return from a non-reentrant subroutine is typically made through the same register that was used in its call. Thus, a subroutine called with a JSR PC, dst exits with a RTS PC and 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



# MARK



**Operation:** SP ← updated PC + 2 + 2n n = number of parameters  
 PC ← R5  
 R5 ← (SP) ↓

**Condition Codes:** unaffected

**Description:** Used as part of the standard PDP-11 subroutine return convention. MARK facilitates the stack clean up procedures involved in subroutine exit. Assembler format is: MARK N

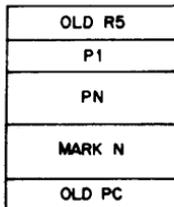
**Example:**

```

MOV R5, -(SP)           ;place old R5 on stack
MOV P1, -(SP)          ;place N parameters
MOV P2, -(SP)          ;on the stack to be
                       ;used there by the
                       ;subroutine

MOV PN, -(SP)
MOV #MARKN, -(SP)      ;places the instruction
                       ;MARK N on the stack
MOV SP, R5             ;set up address at Mark N in-
                       ;struction
JSR PC, SUB            ;jump to subroutine
  
```

At this point the stack is as follows:



And 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

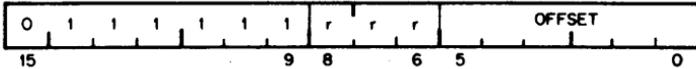
causes the contents of R5 to be placed in the PC which then results in the execution of the instruction MARK N. The contents of old PC are placed in R5

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) contents of the old R5 to be popped into R5 thus completing the return from subroutine.

# SOB

subtract one and branch (if  $\neq 0$ )

077R00



**Operation:**  $(R) \leftarrow (R) - 1$ ; if this result  $\neq 0$  then  $PC \leftarrow PC - (2 \times \text{offset})$  if  $(R) = 0$ ;  $PC \leftarrow PC$

**Condition Codes:** unaffected

**Description:** The register is decremented. If it is not equal to 0, twice the offset is subtracted from the PC (now pointing to the following word). The offset is interpreted as a sixbit positive number. This instruction provides a fast, efficient method of loop control. Assembler syntax is:

SOB R,A

Where A is the address to which transfer is to be made if the decremented R is not equal to 0. Note that the SOB instruction can not be used to transfer control in the forward direction.

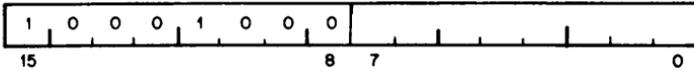
**Traps**

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 two-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

emulator trap

104000—104377



**Operation:**

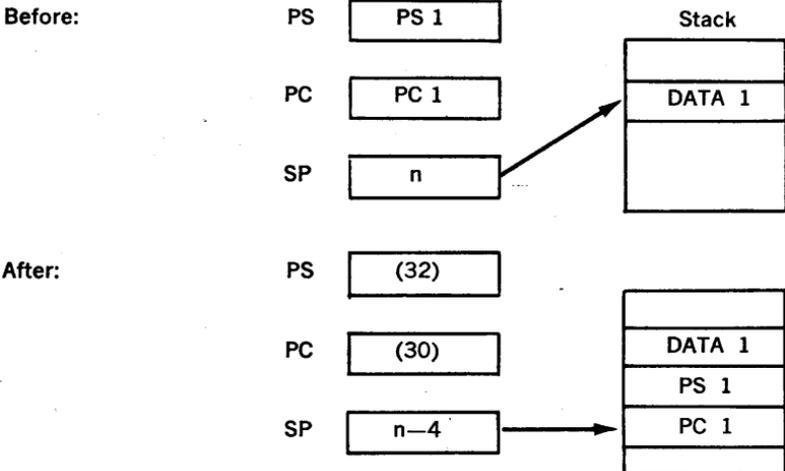
- ▼(SP)←PS
- ▼(SP)←PC
- PC←(30)
- PS←(32)

**Condition Codes:**

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

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

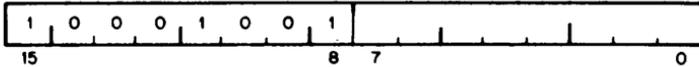
**Caution:** EMT is used frequently by DEC system software and is therefore not recommended for general use.



# TRAP

trap

104400—104777



**Operation:**       $\nabla(SP) \leftarrow PS$   
                      $\nabla(SP) \leftarrow PC$   
                      $PC \leftarrow (34)$   
                      $PS \leftarrow (36)$

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

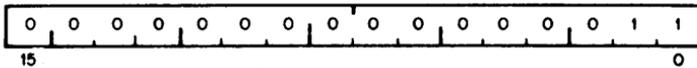
**Description:**        Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34.

**Note:** Since DEC software makes frequent use of EMT, the TRAP instruction is recommended for general use.

# BPT

breakpoint trap

000003



**Operation:**         $\downarrow(\text{SP}) \leftarrow \text{PS}$   
                      $\downarrow(\text{SP}) \leftarrow \text{PC}$   
                      $\text{PC} \leftarrow (14)$   
                      $\text{PS} \leftarrow (16)$

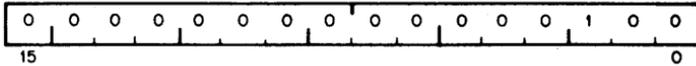
**Condition Codes:** N: loaded from trap vector  
                     Z: loaded from trap vector  
                     V: loaded from trap vector  
                     C: loaded from trap vector

**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 de-  
                     bugging aids.  
                     (no information is transmitted in the low byte.)

# IOT

input/output trap

000004



**Operation:**             $\downarrow(\text{SP}) \leftarrow \text{PS}$   
                          $\downarrow(\text{SP}) \leftarrow \text{PC}$   
                          $\text{PC} \leftarrow (20)$   
                          $\text{PS} \leftarrow (22)$

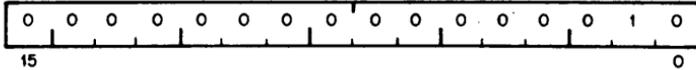
**Condition Codes:**    N:loaded from trap vector  
                            Z:loaded from trap vector  
                            V:loaded from trap vector  
                            C:loaded from trap vector

**Description:**        Performs a trap sequence with a trap vector address of 20.  
                                (no information is transmitted in the low byte)

# RTI

return from interrupt

000002



**Operation:** PC  $\leftarrow$  (SP)<sup>M</sup>  
PS  $\leftarrow$  (SP)<sup>A</sup>

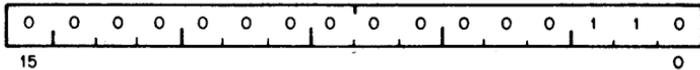
**Condition Codes:** N: loaded from processor stack  
Z: loaded from processor stack  
V: loaded from processor stack  
C: loaded from processor stack

**Description:** Used to exit from an interrupt or TRAP service routine. The PC and PS are restored (popped) from the processor stack. If a trace trap is pending, the first instruction after RTI will not be executed prior to the next T traps.

# RTT

return from interrupt

000006



**Operation:** PC  $\leftarrow$  (SP)  $\uparrow$   
PS  $\leftarrow$  (SP)  $\uparrow$

**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 while RTI permits trace trap. If new PS has T bit set, trap will occur after execution of first instruction after RTT.

**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, and trap to vector address 4. Reserved instructions trap to vector address 10.

**Bus Error Traps**—Bus Error Traps are time-out errors; attempts to reference addresses on the bus that have made no response within a certain length of time. In general, these are caused by attempts to reference non-existent memory, and attempts to reference non-existent peripheral devices. Bus error traps cause processor traps through the trap vector address 4.

**Trace Trap**—Trace Trap is enabled by bit 4 of the PSW 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.

The following are special cases of the T-bit and are detailed in subsequent paragraphs.

1. The traced instruction cleared the T-bit.
2. The traced instruction set the T-bit.
3. The traced instruction caused an instruction trap.
4. The traced instruction caused a bus error trap.
5. The processor was interrupted between the time the T-bit was set and the fetching of the instruction that was to be traced.
6. The traced instruction was a WAIT.
7. The traced instruction was a HALT.
8. The traced instruction was a Return from Interrupt.

#### NOTE

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

**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 execution of this instruction. The status word on the stack, however, will have a clear T-bit.

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

**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 exists 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.

**An instruction that caused a Bus Error Trap**—This is treated as an Instruction Trap. The only difference is that the error service is not as likely to exit with an RTI, so that the trace trap may not occur.

Note that interrupts may be acknowledged immediately after the loading of the new PC and PS at the trap vector location. To lock out all interrupts, the PSW at the trap vector should set Bit 7.

**A WAIT**—T bit trap is not honored during a wait.

**A HALT**—The processor halts. The PC points to the next instruction to be executed. The trap will occur immediately following execution resumption.

**A Return from Interrupt**—The return from interrupt instruction either clears or sets the T-bit. If the T-bit was set and RTT is the traced instruction, the trap is delayed until completion of the next instruction.

**Power Failure Trap**—Occurs when AC power fail signal is received while processor is in run mode. Trap vector for power failure is location 24 and 26. Trap will occur if an RTI instruction is executed in power fail service routine.

**Trap Priorities**—In case of internal and external multiple processor trap conditions, occurring simultaneously, the following order of priorities is observed (from high to low):

- Bus Error Trap
- Memory Refresh
- Instruction Traps
- Trace Trap
- Halt Line
- Power Fail Trap
- Event Line Interrupt
- Device (Bus) Interrupt Request

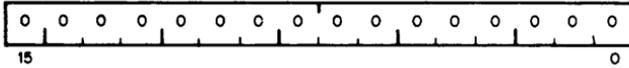
If a bus error is caused by the trap process handling instruction traps, trace traps, or a previous bus error, the processor is halted. This is called a double bus error.

## 4.7 MISCELLANEOUS

# HALT

halt

000000



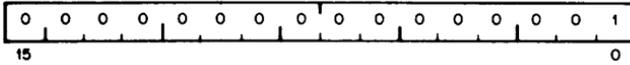
**Condition Codes:** not affected

**Description:** Causes the processor to leave RUN mode. The PC points to the next instruction to be executed. The processor goes into the HALT mode. The console mode of operation is enabled.

# WAIT

wait for interrupt

000001



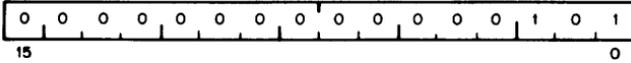
**Condition Codes:** not affected

**Description:** Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt request. Having been given a WAIT command, the processor will not compete for bus use by fetching instructions or operands from memory. This permits higher transfer rates between a device and memory, since no processor-induced latencies will be encountered by interrupt requests from devices. 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.

# RESET

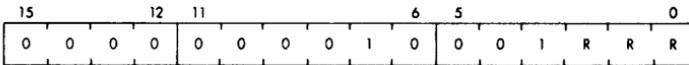
reset external bus

000005



**Condition Codes:** not affected

**Description:** Sends INIT on the BUS for 10  $\mu$ sec. All devices on the BUS are reset to their state at power-up. The processor remains in an idle state for 90  $\mu$ sec following issuance of INIT.



**Operation:** (R) ← gets contents of 5 internal 16 bit registers R ← R + 12 at end of inst.

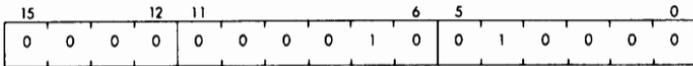
**Condition Codes:** Unaffected

**Description:** Contents of register R (low order 3 bits of inst.) is used as a pointer. The contents of the internal hidden temporary registers are consecutively written into memory and the contents of R are incremented by 2 until the five 16 bit registers have been written.  
 (R) ← (R) + 12. Primarily used as a maintenance aid in diagnostic routines. The interpretation of the five words in memory is as follows:

Memory Location	Microlevel Register Symbol	Function
(R)	RBA	Bus address Register. It contains the last non-instruction fetch bus address for destination modes, 3, 5, 6 and 7.
(R)+2	RSRC	Source Operand Register. It contains the last source operand of a double operand instruction. The high byte may not be correct if it was source mode 0.
(R)+4	RDST	Destination operand register. It contains the last destination operand fetched by the processor.
(R)+6	RPSW	PSW and Scratch Register. The top 4 bits are PSW bits 4 thru 7. The remaining bit interpretation is a function of the last instruction and may not be that useful for all cases.
(R)+10	RIR	Instruction Register. It contains the present, not past, instruction being executed, and will always be 36R where R is the register in the format. The 360 is a result of firmware instruction decoding and is caused by 150 being added to the opcode (21R+150=36R).

(NO ASSIGNED MNEMONIC)

00022N



**Operation:** Causes Micro Instruction Control Transfer to Microlocation 3000

**Condition Codes:** Unaffected

**Description:** This instruction can be used to transfer Microcontrol to Microcode address 3000 in the Microprocessor. If Microaddress 3000 does not exist this opcode will cause a reserved instruction trap through memory location 10.

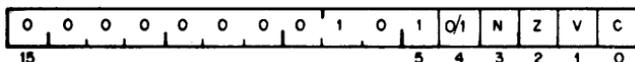
This is a reserved DEC instruction.

## Condition Code Operators

<b>CLN</b>	<b>SEN</b>
<b>CLZ</b>	<b>SEZ</b>
<b>CLV</b>	<b>SEV</b>
<b>CLC</b>	<b>SEC</b>
<b>CCC</b>	<b>SCC</b>

condition code operators

0002XX



**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 0-3) 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 is a 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 CC's	000277
CCC    Clear all CC's	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.

## PROGRAMMING TECHNIQUES

In order to produce programs which fully utilize the power and flexibility of the LSI-11, the reader should become familiar with the various programming techniques which are part of the basic design philosophy of the LSI-11. Although it is possible to program the LSI-11 along traditional lines such as "accumulator orientation" this approach does not fully exploit the architecture and instruction set of the LSI-11.

## 5.1 THE STACK

A "stack," as used on the LSI-11, is an area of memory set aside by the programmer for temporary storage or subroutine/interrupt service linkage. The instructions which facilitate "stack" handling are useful features not normally found in low-cost computers. They allow a program to dynamically establish, modify, or delete a stack and items on it. The stack uses the "last-in, first-out" concept, that is, various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order. On the LSI-11, a stack starts at the highest location reserved for it and expands linearly downward to the lowest address as items are added to the stack.

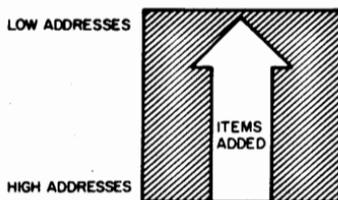


Figure 5-1: Stack Addresses

The programmer does not need to keep track of the actual locations his data is being stacked into. This is done automatically through a "stack pointer." To keep track of the last item added to the stack (or "where we are" in the stack) a General Register always contains the memory address where the last item is stored in the stack. In the LSI-11 any register except Register 7 (the Program Counter-PC) may be used as a "stack pointer" under program control; however, instructions associated with subroutine linkage and interrupt service automatically use Register 6 (R6) as a hardware "Stack Pointer." For this reason R6 is frequently referred to as the system "SP."

Stacks in the LSI-11 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.

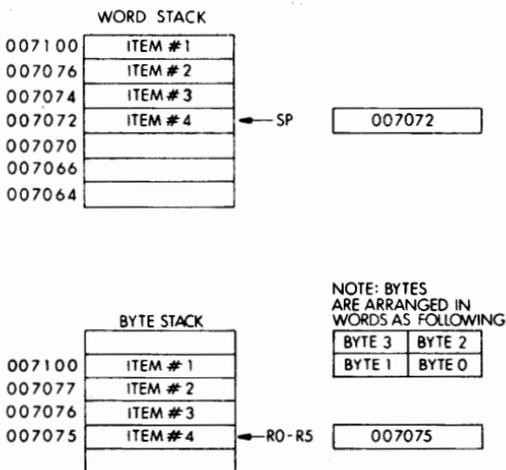


Figure 5-2: Word and Byte Stacks

Items are added to a stack using the autodecrement addressing mode with the appropriate pointer register. (See Chapter 3 for description of the autoincrement/decrement modes).

This operation is accomplished as follows;

**MOV Source,—(SP) ;MOV Source Word onto the stack**  
**or**

**MOVB Source,—(R) ;MOVB Source Byte onto a stack**

This is called a "push" because data is "pushed onto the stack."

To remove an item from a stack the autoincrement addressing mode with the appropriate R is employed. This is accomplished in the following manner:

```
MOV (SP), +, Destination ;MOV Destination Word off the stack
or
```

```
MOVB (R) +, Destination ;MOVB Destination Byte off the stack
```

Removing an item from a stack is called a "pop" for "popping from 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 shareable temporary storage locations.

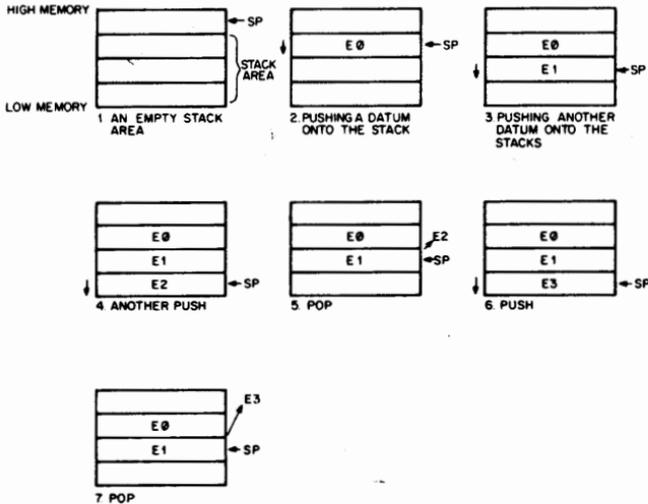


Figure 5-3: Illustration of Push and Pop Operations

As an example of stack usage 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:

Address	Octal Code	Assembler Syntax
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	016702	MOV TEMP2, R2 ;Restore R2
076416	000004	*
076420	000207	RTS PC
076422	000000	TEMP1: 0
076424	000000	TEMP2: 0

\*Index Constants

Figure 5-4: Register Saving Without the Stack

OR: Using a Stack

Address	Octal Code	Assembler Syntax
010020	010143	SUBR: MOV R1, -(R3) ;push R1
010022	010243	MOV R2, -(R3) ;push R2
.	.	.
.	.	.
010130	012301	MOV (R3) +, R2 ;pop R2
010132	012302	MOV (R3) +, R1 ;pop R1
010134	000207	RTS PC

**Note:** In this case R3 was used as a Stack Pointer

Figure 5-5: Register Saving using the Stack

The second routine uses four less 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 very economical way to save on memory usage.



When a JSR is executed, the contents of the linkage register are saved on the system R6 stack as if a MOV reg.-(SP) had been performed. Then the same register is loaded with the memory address following the JSR instruction (the contents of the current PC) and a jump is made to the entry location specified by the DST operand.

Address	Assembler Syntax	Octal Code
001000	JSRR5 SUBR	004567
001002	index constant for SUBR	000060
001064	SUBR MOV A.B	0Innmm

Figure 5-7: JSR using R5

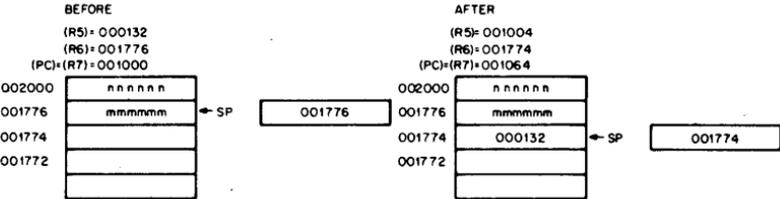


Figure 5-8: JSR

Note that the instruction JSR R6,SUBR is not normally considered to be a meaningful combination.

**5.2.2 Argument Transmission**

The memory location pointed to by the linkage register of the JSR instruction may contain arguments or addresses of arguments. These arguments may be accessed from the subroutine in several ways. Using Register 5 as the linkage register, the first argument could be obtained by using the addressing modes indicated by (R5), (R5) + ,X(R5) for actual data, or @(R5) + , etc. for the address of data. If the autoincrement mode is used, the linkage register is automatically updated to point to the next argument.

Figures 5-9 and 5-10 illustrate two possible methods of argument transmission.

**Address Instructions and Data**

010400	JSR R5,SUBR	
010402	Index constant for SUBR	SUBROUTINE CALL
010404	arg # 1	ARGUMENTS
010406	arg # 2	
.	.	
.	.	
.	.	
020306	SUBR: MOV (R5) + ,R1	:get arg # 1
020310	MOV (R5) + ,R2	:get arg # 2 Retrieve Arguments from SUB

Figure 5-9; Argument Transmission -Register Autoincrement Mode

Address	Instructions and Data	
010400	JSR R5,SUBR	
010402	index constant for SUBR	SUBROUTINE CALL
010404	077722	Address of Arg # 1
010406	077724	Address of Arg. # 2
010410	077726	Address of Arg. # 3
.	.	.
.	.	.
077722	Arg # 1	
077724	arg # 2	arguments
077726	arg # 3	
.	.	.
.	.	.
020306	SUBR: MOV @(R5) + ,R1	;get arg # 1
020301	MOV @(R5) + ,R2	;get arg # 2

Figure 5-10: Argument Transmission-Register Autoincrement Deferred Mode

Another method of transmitting arguments is to transmit only the address of the first item by placing this address in a general purpose register. It is not necessary to have the actual argument list in the same general area as the subroutine call. Thus a subroutine can be called to work on data located anywhere in memory. In fact, in many cases, the operations performed by the subroutine can be applied directly to the data located on or pointed to by a pointer without the need to ever actually move this data into the subroutine area.

Calling Program: MOV POINTER, R1  
JSR PC,SUBR

SUBROUTINE ADD (R1) + ,(R1) ;Add item # 1 to item # 2, place result in item # 2, R1 points to item # 2 now

etc.  
or

ADD (R1),2(R1) ;Same effect as above except that R1 still points to item # 1 etc.

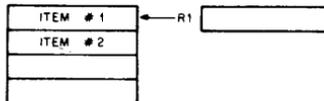


Figure 5-11: Transmitting Stacks as Arguments

Because the LSI-11 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 quite convenient to use this same stack to save and restore intermediate 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 already been saved at the beginning of a subroutine. In the previous example R5 may be used to index the arguments while R6 is free to be incremented and decremented in the course of 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 which occurs.



Figure 5-12: Shifting Indexed Base

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.



Figure 5-13: Constant Index Base Using "R6 Copy"

### 5.2.3 Subroutine Return

In order to provide for a return from a subroutine to the calling program an RTS instruction is executed by the subroutine. This instruction should specify the same register as the JSR used in the subroutine call. When executed, it causes the register, specified, to be moved to the PC and the top of the stack to be then placed in the register specified. Note that if an RTS PC is executed, it has the effect of returning to the address specified by the contents of the top of the stack.

Note that the JSR and the JMP instructions differ in that a linkage register is always used with a JSR; there is no linkage register with a JMP and no way to return to the calling program.

When a subroutine finishes, it is necessary to "clean-up" the stack by eliminating or skipping over the subroutine arguments. One way this can be done is by making the subroutine keep the number of arguments as its first stack item. Returns from subroutines would then involve calculating the amount by which to reset the stack pointer. Resetting the stack pointer then restores the original contents of the register which was used as the copy of the stack pointer. The LSI-11 however, has a specific instruction (MARK instruction) used to perform the clean-up task. The MARK instruction which is stored on a stack in place of "number of argument" information may be used to automatically perform these "clean-up" chores.

### 5.2.4 LSI-11 Set Subroutine Advantages

There are several advantages to the LSI-11 Set subroutine calling procedure.

- a. arguments can be quickly passed between the calling program and the subroutine.
- b. if the user has 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 taken up for linkage.
- c. 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 simply be made by automatically popping this information from the stack in the opposite order of the JSRs.

Such linkage address bookkeeping is called automatic "nesting" of subroutine calls. This feature enables the programmer to construct fast, efficient linkages in a simple, flexible manner. It even permits a routine to call itself in those cases where this is meaningful. It also allows subroutines to be interrupted by external devices without losing the proper return linkage registers.

### 5.2.5 Trap Subroutine Calls

The TRAP instruction may be used to call subroutines. The TRAP instruction is typically used with a package of many different subroutines such as the software floating-point math package. The subroutines in the package are assigned a unique number which is to be included in the TRAP instruction. When a subroutine is called, a "TRAP n" instruction is executed, where "n" is the number ( $0 \leq n \leq 255$ ) which designates

the subroutine to be invoked. Arguments are typically passed on the stack, in the registers, or they may follow the TRAP instruction. The advantages of using the TRAP instruction are that a program using a TRAP subroutine package may be assembled and linked independent of the TRAP package and the subroutine call only requires one word, as opposed to two words which are normally required using the JSR instruction. The disadvantage of using the TRAP instruction is the extra overhead incurred in the software decoding of the TRAP instruction.

Calling Program:	MOV ARG, -(SP)	; Push argument onto the stack
	TRAP 3	; Invoke subroutine #3
Trap handler:	TRAPH: MOV R0, -(SP)	; Save register
		..
	MOV 4(SP), R0	; Copy address of the TRAP instruction +2
	MOVB -2(R0), R0	; Copy subroutine number in TRAP instruction
	BIC #177400, R0	; Clear possible sign extension bits
	ASL R0	; Convert to word offset
	JSR PC,@TRPTBL(R0)	; Call subroutine
	MOV (SP)+, R0	; Restore register
	RTT	; Return to user
	TRPTBL: SUB0	; Table of pointer to subroutines
	SUB1	
	SUB2	
	SUB3	
	..	

## 5.3 INTERRUPTS

### 5.3.1 General Principles

Interrupts are in many respects very similar to subroutine calls. However, they are forced, rather than controlled, transfers of program execution occurring because of some external and program-independent event (such as a stroke on the teleprinter keyboard). Like subroutines, interrupts have linkage information such that a return to the interrupted program can be made. More information is actually necessary for an interrupt than a subroutine 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. (i.e. was the previous operation zero or negative, etc.) 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:

MFPS, -(SP)	; Push PS
MOV R7, -(SP)	; Push PC

had been executed.

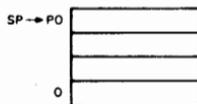
The new contents of the PC and PS are loaded from two preassigned consecutive memory locations which are called an "interrupt vector." The actual locations are chosen by the device interface designer and are located in low memory addresses. The first word contains the interrupt service routine address (the address of the new program sequence) and the second word contains the new PS which will determine the machine status including the operational mode and register set to be used by the interrupt service routine.

After the interrupt service routine has been completed, an RTI (return from interrupt) is performed. The two top words of the stack are automatically "popped" and placed in the PC and PS respectively, thus resuming the interrupted program.

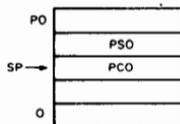
### 5.3.2 Nesting

Interrupts can be nested in much the same manner that subroutines are nested. In fact, it is possible to nest any arbitrary mixture of subroutines and interrupts without any confusion. By using the RTI and RTS instructions, respectively, the proper returns are automatic.

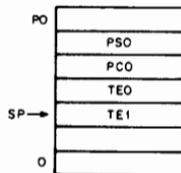
1. Process 0 is running; SP is pointing to location P0.



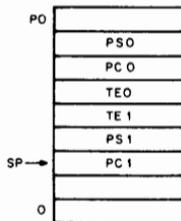
2. Interrupt stops process 0 with PC = PC0, and status = PS0; starts process 1.



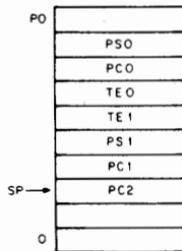
3. Process 1 uses stack for temporary storage (TE0, TE1).



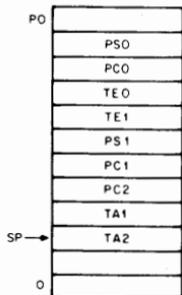
4. Process 1 interrupted with PC = PC1 and status = PS1; process 2 is started



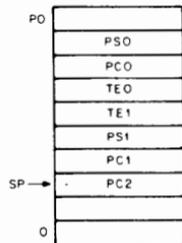
5. Process 2 is running and does a JSR R7,A to Subroutine A with PC = PC2.



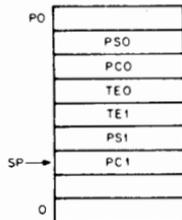
6. Subroutine A is running and uses stack for temporary storage.



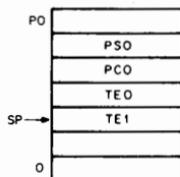
7. Subroutine A releases the temporary storage holding TA1 and TA2.



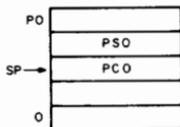
8. Subroutine A returns control to process 2 with an RTS R7,PC is reset to PC2.



9. Process 2 completes with an RTI instruction (dismisses interrupt) PC is reset to PC(1) and status is reset to PS1; process 1 resumes.



10. Process 1 releases the temporary storage holding TE0 and TE1.



11. Process 1 completes its operation with an RTI PC is restored to PC0 and status is reset to PS0.

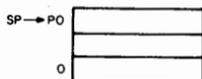


Figure 5-14: Nested Interrupt Service Routines and Subroutines

Note that the area of interrupt service programming is intimately involved with the concept of CPU and device priority levels.

#### 5.4 PROGRAMMING PERIPHERALS

Programming of LSI-11 modules (devices) is simple. A special class of instructions to deal with input/output operations is unnecessary. The bus structure permits a unified addressing structure in which control, status, and data registers for devices are directly addressed as memory locations. Therefore, all operations on these registers, such as transferring 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 DLVII Receiver Data Buffer Register (RBUF) and branches if it finds it. There is no need to transfer the information into an intermediate register for comparison.

When the character is of interest, a memory reference instruction can transfer the character into a user buffer in memory or to another peripheral device. The instruction:

```
MOV DRINBUF LOC
```

transfers a character from the DRV11 Data Input Buffer (DRINBUF) into a user-defined location.

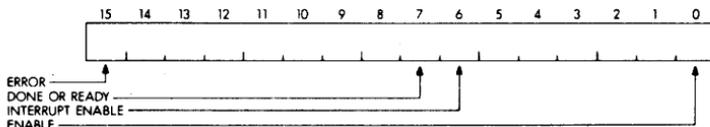
All arithmetic operations can be performed on a peripheral device register. For example, the instruction ADD #10, DROUT BUF will add 10 to the DRV11's Output Buffer.

All read/write device registers can be treated as accumulators. There is no need to funnel all data transfers, arithmetic operations, and comparisons through a single or small number of accumulator registers.

## 5.5 DEVICE REGISTERS

All devices are specified by a set of registers which are addressed as memory and manipulated as flexibly as an accumulator. There are two types of registers associated with each device: (1) control and status registers; (2) data registers. The following examples are general, for specific device register information refer to the applicable manual.

Control and Status Registers—Each device can have one or more control and status registers that contain the information necessary to communicate with that device. The general form, shown below does not necessarily apply to every device, but is presented as a guide.



BIT	NAME	DESCRIPTION
15	Error	Set when an error occurs.
7	Done or Ready	Set when the device is either ready to accept new information, or has completed an operation and has data available.
6	Interrupt Enable	When set, an interrupt will be requested when a done or error condition occurs.
0	Enable	Set to allow the peripheral device to perform a function.

Many devices require less than sixteen status bits. Other devices will require more than sixteen bits and therefore will require additional status and control registers.

Data Buffer Registers—Each device has at least one buffer register for temporarily storing data to be transferred into or out of the processor. The number and type of data registers is a function of the device. The DLV11 for example uses single 8-bit data buffer registers. The DRV11 uses 16-bit data registers and some devices may use more than 1 register for data buffers.

**Interrupt Structure**—If the appropriate interrupt enable bit is set, in the control and status register of a device, transition from 0 to 1 of the READY or ERROR bit, where applicable, should cause an interrupt request to be issued to the processor. Also if READY or ERROR is a 1 when the interrupt enable is turned on, an interrupt request is made. If the device makes the request and the processor's priority is zero, and no higher priority devices are requesting an interrupt, the request is granted, and the interrupt sequence takes place.

- a. the current program counter (PC) and processor status (PS) are pushed onto the processor stack;
- b. the new PC and PS are loaded from a pair of locations (the interrupt vector) in addressed memory, unique to the interrupting device.

Since each device has a unique interrupt vector which dispatches control to the appropriate interrupt handling routine immediately, no device polling is required. The Return from Interrupt Instruction (RTI) is used to reverse the action of the interrupt sequence. The top two words on the stack are popped into the PC and PS, returning control to the interrupted sequence.

**Programming Example**—A DLV11 interrupt routine to service a low-speed paper tape reader, could appear as follows (assume the DLV11's interrupt vector is 60, and PRSER is the service routine for the device):

First the user must initialize the Stack Pointer (R6) and device vector locations. Then the user must initialize the service routine by specifying an address pointer and a word count:

```
INIT: MOV # BUFADR, R0           ; set address pointer into
      MOV # COUNT, COUNTR       ; register
      MOV # 101, RCSR           ; set counter
                                   ; enable DLV11
                                   ; interrupt enable &
                                   ; reader run enable,
                                   ; Program continues until
                                   ; interrupt occurs
```

When the interrupt occurs and is acknowledged, the processor stores the current PC and PS on the stack. Next it goes to the interrupt vector and picks up the new PC and PS location 60, 62. When the program was loaded, the address of PRSER would be put in location 60 and 200, in 72 (to set the processor's priority to 4 and inhibit new interrupts). The next instruction executed is the first instruction of the device service routine at PRSER.

```
PRSER:  MOVB RBUF, (R0) +       ; move character from
                                   ; DLV11's receiver data
                                   ; buffer register to buffer and
                                   ; increment pointer
      DEC COUNTR                ; decrement character count
      BEQ DONE                  ; branch when COUNTR equals 0
      INC RCSR                   ; set reader enable for next
                                   ; character input

DONE:   RTI                      ; return to interrupted program
```



## CHAPTER 6

# EXTENDED ARITHMETIC OPTION

### 6.1 GENERAL

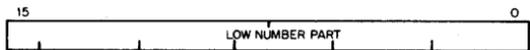
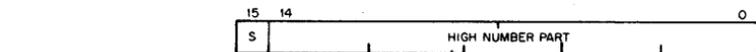
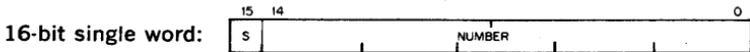
This chapter describes the Extended Arithmetic Chip, which is an option on the KD11-F, KD11-J Microcomputer Module. The KEV11 option allows extended manipulation of fixed point numbers (fixed point arithmetic) and enables direct operations on single precision 32-bit words (floating point arithmetic).

### 6.2 FIXED POINT ARITHMETIC (EIS)

The following instructions apply to fixed point numbers:

Mnemonic	Instruction	Op Code
MUL	multiply	070RSS
DIV	divide	071RSS
ASH	shift arithmetically	072RSS
ASCH	arithmetic shift combined	073RSS

Operand formats are:



S is the sign bit.

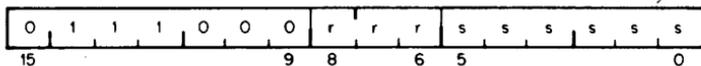
S = 0 for positive quantities

S = 1 for negative quantities; number is in 2's complement notation

# MUL

multiply

070RSS



**Operation:** R, Rv1  $\leftarrow$  R x(src)

**Condition Codes:** N: set if product is  $<0$ ; cleared otherwise  
Z: set if product is 0; cleared otherwise  
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 two'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, Rv1 which reduces to just R when R is odd.)

**Example:** 16-bit product (R is odd)

```
CLC ;Clear carry condition code
MOV #400,R1
MUL #10,R1
BCS ERROR ;Carry will be set if
;product is less than
;-215 or greater than or equal to 215
;no significance lost
```

Before After

(R1) = 000400 (R1) = 004000

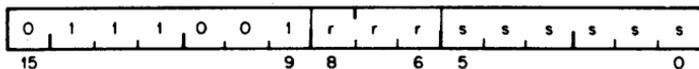
Assembler format for all EIS instructions is:

OPR src, R

# DIV

divide

071RSS



**Operation:** R, Rv1 ← R, Rv1 / (src)

**Condition Codes:** N: set if quotient < 0; cleared otherwise  
Z: set if quotient = 0; cleared otherwise  
V: set if source = 0 or if the absolute value of the register is larger than the absolute value of the source. (In this case the instruction is aborted because the quotient would exceed 15 bits.)  
C: set if divide 0 attempted; cleared otherwise

**Description:** The 32-bit two's complement integer in R and Rv1 is divided by the source operand. The quotient is left in R; the remainder in Rv1. Division will be performed so that the remainder is of the same sign as the dividend. R must be even.

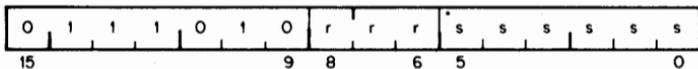
**Example:**  
CLR R0  
MOV #20001,R1  
DIV #2,R0

Before	After	
(R0) = 000000	(R0) = 010000	Quotient
(R1) = 020001	(R1) = 000001	Remainder

# ASH

shift arithmetically

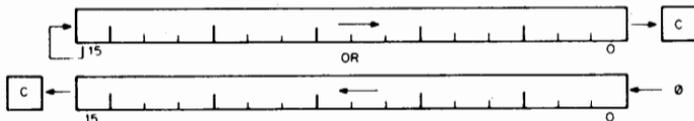
072RSS



**Operation:**  $R \leftarrow R$  Shifted arithmetically NN places to right or left  
Where NN = low order 6 bits of source

**Condition Codes:** N: set if result  $< 0$ ; cleared otherwise  
Z: set if result = 0; cleared otherwise  
V: set if sign of register changed during shift; cleared otherwise  
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 shift count. The shift count is taken as the low order 6 bits of the source operand. This number ranges from -32 to +31. Negative is a right shift and positive is a left shift.



## 6 LSB of source

011111  
000001  
111111  
100000

## Action in general register

Shift left 31 places  
shift left 1 place  
shift right 1 place  
shift right 32 places

## Example:

ASH R0, R3

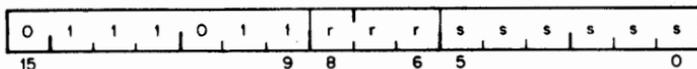
Before  
(R0) = 001234  
(R3) = 000003

After  
(R0) = 012340  
(R3) = 000003

# ASHC

arithmetic shift combined

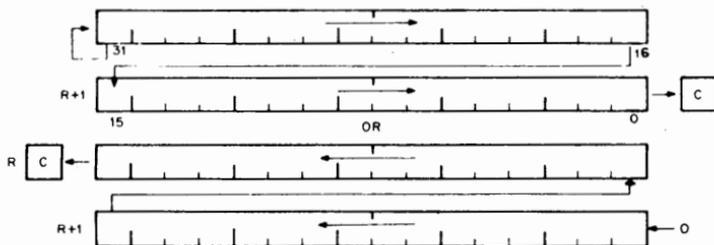
073RSS



**Operation:** R, Rv1 ← R, Rv1 The double word is shifted NN places to the right or left, where NN = low order six bits of source

**Condition Codes:** N: set if result < 0; cleared otherwise  
 Z: set if result = 0; cleared otherwise  
 V: set if sign bit changes during the shift; cleared otherwise  
 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 one are treated as one 32 bit word, R + 1 (bits 0-15) and R (bits 16-31) are shifted right or left the number of times specified by the shift count. The shift count is taken as the low order 6 bits of the source operand. This number ranges from -32 to +31. Negative is a right shift and positive is a left shift. When the register chosen is an odd number the register and the register OR'ed with one are the same. In this case the right shift becomes a rotate (for up to a shift of 16). The 16 bit word is rotated right the number of bits specified by the shift count.

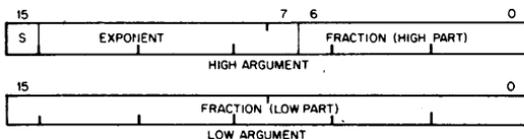


### 6.3 FLOATING POINT ARITHMETIC (FIS)

The Floating Point instructions used are unique to the LSI-11 and PDP-11/35 & 40. However, the OP Codes used do not conflict with any other instructions.

Mnemonic	Instruction	Op Code
FADD	floating add	07500R
FSUB	floating subtract	07501R
FMUL	floating multiply	07502R
FDIV	floating divide	07503R

The operand format is:



S = sign of fraction; 0 for positive, 1 for negative

Exponent = 8 bits for the exponent, in excess (200)<sub>10</sub> notation

Fraction = 23 bits plus 1 hidden bit (all numbers are assumed to be normalized)

The number format is essentially a sign and magnitude representation. The format is identical with the 11/45 for single precision numbers.

#### Fraction

The binary radix point is to the left (in front of bit 6 of the High Argument), so that the value of the fraction is always less than 1 in magnitude. Normalization would always cause the first bit after the radix point to be a 1, such that the fractional value would be between  $\frac{1}{2}$  and 1. Therefore, this bit can be understood and not be represented directly, to achieve an extra 1 bit of resolution.

The first bit to the right of the radix point (hidden bit) is always a 1. The next bit for the fraction is taken from bit 6 of the High Argument. The result of a Floating Point operation is always rounded away from zero, increasing the absolute value of the number.

#### Exponent

The 8-bit Exponent field (bits 14 to 7) allow exponent values between  $-128$  and  $+127$ . Since an excess (200)<sub>10</sub>, or (128)<sub>10</sub> number system is used, the correspondence between actual values and coded representation is as follows:

Actual Value	Representation	
	Octal	Binary
Decimal +127	377	11 111 111
+1	201	10 000 001
0	200	10 000 000
-1	177	01 111 111
-128	000	00 000 000



Traps occur through the vector at location 244. A Floating Point instruction will be aborted if an interrupt request is issued before the instruction is near completion. The Program Counter will point to the aborted Floating instruction so that the Interrupt will look transparent.

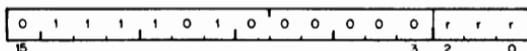
Assembler format is: OPR R

## INSTRUCTIONS

### FADD

floating add

07500R



**Operation:**  $[(R)+4, (R)+6] \leftarrow [(R)+4, (R)+6] + [(R), (R)+2]$ , if result  $\geq 2^{-128}$ ; else  $[(R)+4, (R)+6] \leftarrow 0$

**Condition Codes:** N: set if result  $< 0$ ; cleared otherwise  
Z: set if result = 0; cleared otherwise  
V: cleared  
C: cleared

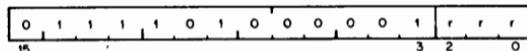
**Description:** Adds the A argument to the B argument and stores the result in the A Argument position on the stack. General register R is used as the stack pointer for the operation.

$A \leftarrow A + B$

### FSUB

floating subtract

07501R



**Operation:**  $[(R)+4, (R)+6] \leftarrow [(R)+4, (R)+6] - [(R), (R)+2]$ , if result  $\geq 2^{-128}$ ; else  $[(R)+4, (R)+6] \leftarrow 0$

**Condition Codes:** N: set if result  $< 0$ ; cleared otherwise  
Z: set if result = 0; cleared otherwise  
V: cleared  
C: cleared

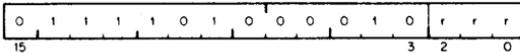
**Description:** Subtracts the B Argument from the A Argument and stores the result in the A Argument position on the stack.

$A \leftarrow A - B$

# FMUL

floating multiply

07502R



**Operation:**  $[(R)+4, (R)+6] \leftarrow [(R)+4, (R)+6] \times [(R), (R)+2]$  if result  $\geq 2^{-128}$ ; else  $[(R)+4, (R)+6] \leftarrow 0$

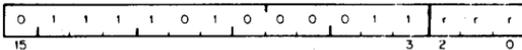
**Condition Codes:** N: set if result  $< 0$ ; cleared otherwise  
Z: set if result = 0; cleared otherwise  
V: cleared  
C: cleared

**Description:** Multiplies the A Argument by the B Argument and stores the result in the A Argument position on the stack.  
 $A \leftarrow A \times B$

# FDIV

floating divide

07503R



**Operation:**  $[(R)+4, (R)+6] \leftarrow [(R)+4, (R)+6] / [(R), (R)+2]$  if result  $\geq 2^{-128}$ ; else  $[(R)+4, (R)+6] \leftarrow 0$

**Condition Codes:** N: set if result  $< 0$ ; cleared otherwise  
Z: set if result = 0; cleared otherwise  
V: cleared  
C: cleared

**Description:** Divides the A Argument by the B Argument and stores the result in the A Argument position on the stack. If the divisor (B Argument) is equal to zero, the stack is left untouched.

$A \leftarrow A/B$



## CONSOLE OPERATION

## 7.1 GENERAL

The LSI-11 can use a standard ASCII terminal or keyboard printer with a 20 mA current loop and resident microcode for console operation. The LA36 is ideally suited for this use and will be described in this chapter.

## 7.2 INTERFACING

Interfacing with the LSI-11 (Figure 7-1) can be accomplished through the DLV11 Serial Line Unit (SLU) and BC08R cable assembly. One end of this cable connects to a 40 pin connector on the DLV11, the other end of the cable is terminated with a Mate-N-Lok connector that is pin-compatible with the following peripheral options:

- LA36 DECwriter
- LT33 Teletypewriter
- LT35 Teletypewriter
- VT05B Alphanumeric Terminal
- VT50 DECscope
- RT02-B Remote Data Entry Terminal

For a detailed description of the DLV11, refer to the LSI-11 user's manual.

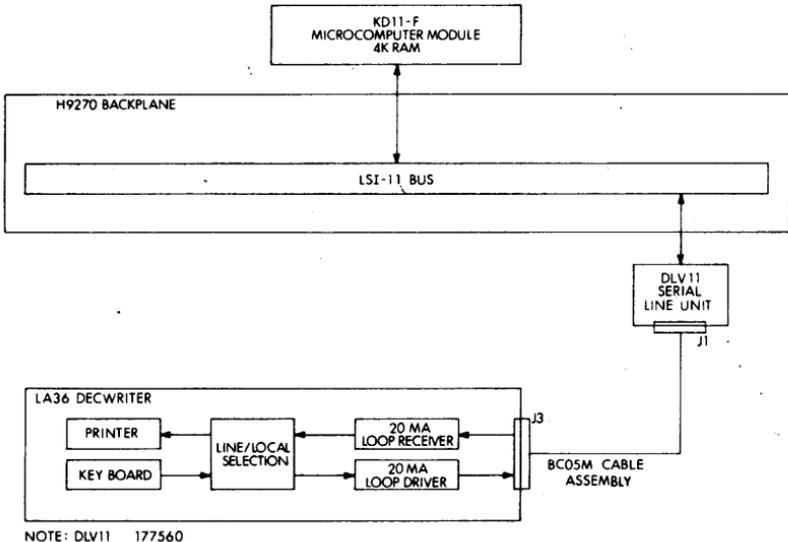


Figure 7-1 Console Interfacing with LA36

### 7.3 ODT/CONSOLE MICROCODE

The LSI-11 does not have an internal or external switch register or control function switch option. In a typical configuration there is no bus device which responds to address 177570 (the SWR address on PDP-11). The function of Load Address, Deposit, Examine, Continue, Start/Halt are implemented with microcode routines that communicate with an operator via a serial stream of ASCII characters. For operation, it requires a serial line interface (e.g., DLV11) at Bus address 177560 and a device that can interpret and display as well as send ASCII characters (e.g., LA36).

The HALT or ODT microcode state of the KD11-F can be entered in five different ways (others are a subset of these) from the RUN state:

- Execution of a LSI-11 HALT instruction
- A double Bus Error (Bus Error trap with SP (R6) pointing to non-existent memory)
- The assertion of a low level on the B HALT line on the Bus
- As a powerup option
- ASCII break with DLV11 framing error asserting the B HALT line (enabled by jumper of DLV11)

Upon entering the HALT state, the KD11-F responds through the console device with an ASCII prompt character sequence. The following prompt sequence is used:

- CR LF nnnnnn CR LF @ (where nnnnnn is the location of the next LSI-11 instruction to be executed and @ is ODT prompt character).

The following is a list of the command character set and its utilization. In each example the operator's entry is not underlined, and the KD11-F response is. Note that in part the character set is a subset of ODT-11. The input character set is interpreted by the KD11-F only when it is in the HALT state.

Note also that all commands and characters are echoed by the KD11-F and that illegal command characters will be echoed and followed by ? (ASCII 012) followed by CR (ASCII 015) followed by LF(ASCII 012) followed by @ (ASCII 100). If a valid command character is received when no location is open (e.g., when having just entered the halt state), the valid command character will be echoed and followed by a ? CR, LF, @. Opening non-existent locations will have the same response. The console always prints six numeric characters; however, the user is not required to type leading zeros for either address or data.

1. **"/" slash (ASCII 057)**

This command is used to open a memory location, general purpose register, or the processor status word. The / command is normally preceded by a location identifier. Before the contents is typed, the console will issue a space (ASCII 40) character.

example:

@ 001000/ 021525

where:

@ = KD11F prompt character (ASCII 100)  
001000 = octal location in address space to be opened  
/ = command to open and exhibit contents of location  
012525 = contents of octal location 1000

**NOTE**

If / used without preceding location identifier, address of last opened location will be used. This feature can be used to verify the data just entered in a location.

2. **"CR" carriage return (ASCII 015)**

This command is used to close an open location. If contents of location are to be changed, CR should be preceded by the new value. If no change to location is necessary then CR will not alter contents.

example:

@ 001000/ 012525 CR LF

@ /012525

OR

example:

@ 001000/ 012525 15126421 CR LF

@ /126421

where:

CR = (ASCII 015) used to close location 1000 in both examples. Note that in second example contents of location 1000 was changed and that only the last 6 digits entered were actually placed in location 1000.

### 3. "LF" line feed (ASCII 012)

This command is also used to close an open location or GPR (general purpose register). If entered after a location has been opened, it will close the open location or GPR and open location + 2 or GPR + 1. If the contents of the open location or GPR are to be modified, the new contents should precede the LF operator.

example:

```
@ 1000/ 012525 LF CR  
001002/ 005252 CR LF  
@
```

where:

LF = (ASCII 012) used to close location 1000 and open location 1002, if used on the PS, the LF will modify the PS if the data has been typed, and close it; then a CR, LF, @ is issued. If LF is used to advance beyond R7, the register name that is printed is meaningless but the contents printed is that of R0.

### 4. "↑" up arrow (ASCII 135)

The "↑" command is also used to close an open location or GPR. If entered after a location or GPR has been opened, it will close the open location or GPR and open location -2, or GPR-1. If the contents of the open location or GPR are to be modified, the new contents should precede the "↑" operator.

example:

```
@ 1000/ 012525↑ CR LF  
000776/ 010101 CR LF  
@
```

where:

"↑" = (ASCII 135) used to close location 1000 and open location 776.

(ASCII 135) up arrow:

If used on the PS, the ↑ will modify the PS if the data has been typed and close it; then CR, LF, @ is issued. If ↑ is used to decrement below R0, the register name that is printed is meaningless but the contents is that of R7.

5. **"@" at sign (ASCII 100)**

The @ command is used once a location has been opened to open a location using the contents of the opened location as a pointer. Also the open location can be optionally modified similar to other commands and if done, the new contents will be used as the pointer.

example:

```
@ 1000/ 000200 @ CR LF
000200/ 000137 CR LF
@
```

where:

@ = (ASCII 100) used to close open location 1000 and open location 200.

Note that the @ command may be used with either GPRs or memory contents.

If used on the PS, the command will modify the PS if data has been typed and close it; however, the last GPR or memory location contents will be used as a pointer.

6. **"←" back arrow (ASCII 137)**

This command is used once a location has been opened to open the location that is the address of the contents of the open location plus the address of the open location plus 2. This is useful for relative instructions where it is desired to determine the effective address.

example:

```
@ 1000/ 000200 ← CR LF
001202/ 002525 CR LF
@
```

where:

"←" = (ASCII 137) used to close open location 1000 and open location 1202 (sum of contents of location 1000, 1000 and 2). Note that this command cannot be used if a GPR or the PS is the open location and if attempted, the command will modify the GPR or PS if data has been typed, and close the GPR or PS; then a CR, LF, @ will be issued.

7. **\$ dollar sign (ASCII 044) or R (ASCII 122) internal register designator:**

Either command if followed by a register value 0—7 (ASCII 060—067) will allow that specific general purpose register to be opened if followed by the / (ASCII 057) command.

example:

@ \$ n/ 012345 CR LF

@

where:

\$ = register designator. This could also be R.

n = octal register 0—7.

012345 = contents of GPR n.

Note that the GPRs once opened can be closed with either the CR, LF, "↑", or @ commands. The "←" command will also close a GPR but will not perform the relative mode operation.

8. **"\$ s" (ASCII 044; ASCII 123) processor status word**

By replacing "n" in the above example with the letter S (ASCII 123) the processor status word will be opened. Again either \$ or R (ASCII 122) is a legal command.

example:

@ \$ S/ 000100 CR LF

@

where:

\$ = GPR or processor status word designator

S = specifies processor status register; differentiates from GPRs.

000200 = eight bit contents of PSW; bit 7 = 1, all other bits = 0.

Note that the contents of the PSW can be changed using the CR command but bit 4 (the T bit) cannot be set using any of the commands.

9. **"G" (ASCII 107)**

The "G" (GO) command is used to start execution of a program at the memory location typed immediately before the "G".

example:

@ 100 G or 100;G

The LSI-11 PC(R7) will be loaded with 100 and execution will begin at that location. Before starting execution, a BUS INIT is issued for 10 μsec followed by 90 μsec of idle time. Note that a semi-colon character (ASCII 073) can be used to separate the address from the G and this is done for PDP-11 ODT compatibility. Since the console is a character-by-character processor, as soon as the "G" is typed, the command is processed and a RUBOUT cannot be issued to cancel the command. If the B HALT L line is asserted,

execution does not take place and only the BUS INIT sequence is done. The machine returns to console mode and prints the PC followed by CR,LF,@.

10. **"P" (ASCII 120)**

The "P" (Proceed) command is used to continue or resume execution at the location pointed to by the current contents of the PC(R7).

example:

@ P or ;P

If the B HALT L line is asserted the INIT line will not be asserted, a single instruction will be executed, and the machine will return to console mode. It will print the contents of the PC followed by a CR,LF,@. In this fashion, it is possible to single instruction step through a user program.

The semicolon is accepted for PDP-11 ODT compatibility. If the semicolon character is received during any character sequence, the console ignores it.

11. **"M" (ASCII 115)**

The "M" (Maintenance) command is used for maintenance purposes and prints the contents of an internal CPU register. This data reflects how the machine got to the console mode.

example:

@ M 000213 CR LF

@

The console prints six characters and then returns to command mode by printing CR,LF,@.

The last octal digit is the only number of significance and is encoded as follows. The value specifies how the machine got to the console mode.

Last Octal Digit Value	Function
0	Halt instruction or B Halt line
1	Bus Error occurred while getting device interrupt vector
2	Bus Error occurred while doing memory refresh
3	Double Bus Error occurred (stack was non-existent value)
4	Non-existent Micro-PC address occurred on internal CPU bus

In the above example, the last octal digit is a "3", which indicates a Double Bus Error occurred.

## 12. "RO" RUBOUT (ASCII 177)

While RUBOUT is not truly a command, the console does support this character. When typing in either address or data, the user can type RUBOUT to erase the previously typed character and the console will respond with a "\ " (Backslash—ASCII 134) for every typed RUBOUT.

example:

```
@ 000100/ 077777 123457 (RUBOUT) \ 6 CR LF
@ 000100/ 123456
```

In the above example, the user typed a "7" while entering new data and then typed RUBOUT. The console responded with a "\ " and then the user typed a "6" and CR. Then the user opens the same location and the new data reflects the RUBOUT. Note that if RUBOUT is issued repeatedly, only numerical characters are erased and it is not possible to terminate the present mode the console is in. If more than six RUBOUTS are consecutively typed, and then a valid location closing command is typed, the open location will be modified with all zeroes.

The RUBOUT command cannot be used while entering a register number. R2 \ 4 / 012345 will not open register R4; however the RUBOUT command will cause ODT to revert to memory mode and open location 4.

## 13. "L" (ASCII 114)

The "L" (Boot Loader) command will cause the processor to self-size memory and then load from the specified device a program that is in Bootstrap Loader Format (e.g.—Absolute Loader). The device is specified by typing in the address of the input control and status register immediately before the "L".

example:

```
@ 177560L
```

First memory is sized, starting at 28K and the device address (177560) is placed in the last location for Absolute Loader compatibility. Then the program will be loaded by setting the "GO" bit in address 177560 and reading a byte of data from 177562.

The loading begins at the address specified in the Bootstrap loader format. The loading is terminated when address XXX775 has been loaded and execution automatically begins at XXX774. It is up to the program being loaded to halt the processor if that is desired. In the case of the Absolute Loader, the processor will halt and the console will print XXX500 (the current PC) followed by CR,LF,@. (XXX = 017 for 4K memory; XXX = 157 for 28K memory).

When loading a program using the "L" command, the B HALT L line is ignored. If a timeout error occurs, the console will terminate the load and print ?,CR,LF,@.

Any device address may be used as long as it is software com-

patible with the DLV11. If no address is typed, address 0 will be used.

**14. "CONTROL-SHIFT-S" (ASCII 23)**

This command is used for manufacturing test purposes and is not a normal user command. It is briefly described here so that in case a user accidentally types this character, he will understand the machine response. If this character is typed, ODT expects two more characters. It uses these two characters as a 16-bit binary address and starting at that address, dumps five locations in binary format to the serial line.

It is recommended that if this mode is inadvertently entered, two characters such as a NULL (0) and @ (ASCII 100) be typed to specify an address in order to terminate this mode. Once completed, ODT will issue a CR, LF, @.



## APPENDIX A

### MEMORY MAP

#### RESERVED VECTOR LOCATIONS

000 (RESERVED)  
004 TIME OUT & OTHER ERRORS  
010 ILLEGAL & RESERVED INSTRUCTION  
014 BPT INSTRUCTION AND T BIT  
020 IOT INSTRUCTION  
024 POWER FAIL  
030 EMT INSTRUCTION  
034 TRAP INSTRUCTION  
060 CONSOLE INPUT DEVICE  
064 CONSOLE OUTPUT DEVICE  
100 EXTERNAL EVENT LINE INTERRUPT  
244 FIS (OPTIONAL)

#### DEVICE ADDRESSES

160000 DEVICE ADDRESSES  
ARE SELECTED BY  
JUMPERS LOCATED  
ON THEIR LINE UNIT  
177776 MODULES.



## INSTRUCTION TIMING

**B.1 LSI-11 INSTRUCTION EXECUTION TIME**

The execution time for an instruction depends on the instruction itself, the modes of addressing used, and the type of memory referenced. In most cases the instruction execution time is the sum of a Basic Time, a Source Address (SRC) Time, and a Destination Address (DST) Time.

$$\text{INSTR TIME} = \text{Basic Time} + \text{SRC Time} + \text{DST Time}$$

$$(\text{BASIC Time} = \text{Fetch Time} + \text{Decode Time} + \text{Execute Time})$$

Some of the instructions require only some of these times. All timing information is in microseconds, unless otherwise noted. Times are typical; processor timing can vary  $\pm 20\%$ .

**SOURCE AND DESTINATION TIME**

MODE	SRC TIME (Word)	SRC TIME (Byte)	DST TIME (Word)	DST TIME (Byte)
0	0	0	0	0
1	1.40 $\mu\text{sec}$	1.05 $\mu\text{sec}$	2.10 $\mu\text{sec}$	1.75 $\mu\text{sec}$
2	1.40	1.05	2.10	1.75
3	3.50	3.15	4.20	4.20
4	2.10	1.75	2.80	2.45
5	4.20	3.85	4.90	4.90
6	4.20	3.85	4.90	4.55
7	6.30	5.95	6.65	7.00

NOTE FOR MODE 2 and MODE 4 if R6 or R7 used with Byte operation, add 0.35  $\mu\text{sec}$  to SRC time and 0.70  $\mu\text{sec}$  to DST time.

**INSTRUCTION TIME**

DOPS (Double Operand)	DM0	DM1-7
MOV	3.50 $\mu\text{sec}$	2.45 $\mu\text{sec}$
ADD,XOR,SUB,BIC,BIS	3.50	4.20
CMP,BIT	3.50	3.15
MOVB	3.85	3.85
BICB,BISB	3.85	3.85
CMPB,BITB	3.15	2.80

SOPS (Single Operand)	DM0	DM1-7
CLR	3.85 $\mu$ sec	4.20 $\mu$ sec
INC,ADC,DEC,SBC	4.20	4.90
COM,NEG	4.20	4.55
ROL,ASL	3.85	4.55
TST	4.20	3.85
ROR	5.25	5.95
ASR	5.60	6.30
CLRB,COMB,NEGB	3.85	4.20
ROLB,ASLB	3.85	4.20
INCB,DECB,SBCB,ADCB	3.85	4.55
TSTB	3.85	3.50
RORB	4.20	4.90
ASRB	4.55	5.95
SWAB	4.20	3.85
SXT	5.95	6.65
MFPS (1067DD)	4.90	6.65
MTPS (1064SS)	7.00	7.00*

\*For MTPS use Byte DST time not SRC time.

\*Add 0.35  $\mu$ sec to instr. time if Bit 7 of effective OPR = 1

JMP/JSR MODE	DST TIME
1	0.70 $\mu$ sec
2	1.40
3	2.10
4	1.40
5	2.80
6	2.80
7	4.90

INSTRUCTION	TIMES
JMP	3.50 $\mu$ sec
JSR (PC=LINK)	5.25
JSR (PC $\neq$ LINK)	6.40
ALL BRANCHES	3.50 (CONDITION MET OR NOT MET)
SOB(BRANCH)	4.90
SOB(NO BRANCH)	4.20
SET CC	3.50
CLEAR CC	3.50
NOP	3.50
RTS	5.25
MARK	11.55
RTI	8.75*
RTT	8.75*+

INSTRUCTION	TIMES
TRAP,EMT	16.80* $\mu$ sec
IOT,RPT	18.55*
WAIT	6.30
HALT	5.60
RESET	5.95 + 10.0 $\mu$ sec. for INIT + 90.0 $\mu$ sec.
MAINT INST. (00021R)	20.30
RSRVD INST. (00022N)	5.95 (TO GET TO UADDRESS 3000)

\* IF NEW PS HAS BIT 4 or BIT 7 SET ADD 0.35  $\mu$ sec FOR EACH  
+ IF NEW PS HAS BIT 4 (T BIT) SET ADD 2.10  $\mu$ sec

### EXTENDED ARITHMETIC (KEV11) INSTRUCTION TIMES

#### EIS Instruction Times

MODE	SRC TIME
0	0.35 $\mu$ sec.
1	2.10
2	2.80
3	3.15
4	2.80
5	3.85
6	3.85
7	5.60

INSTRUCTION	BASIC TIME
MUL	24.0 to 37.0 $\mu$ sec. If both numbers less than 256 in absolute value
DIV	64.0 $\mu$ sec. Worst Case 16 bit multiply 78.0 $\mu$ sec. Worst Case
ASH (RIGHT)	10.1 + 1.75 per shift
ASH (LEFT)	10.8 + 2.45 per shift
ASHC (RIGHT)	10.1 + 2.80 per shift
ASHC (LEFT)	10.1 + 3.15 per shift

#### FIS Instruction Times ( $\mu$ sec)

INST. TIME = BASIC TIME + SHIFT TIME FOR BINARY POINTS + SHIFT TIME FOR NORMALIZATION

INSTRUCTION	BASIC TIME
FADD	42.1 $\mu$ sec
FSUB	42.4

EXPONENT DIFFERENCE	ALIGN BINARY POINTS
0 — 7	2.45 $\mu$ sec per shift
8 — 15	3.50 + 2.45 per shift over 8
16 — 23	7.00 + 2.45 per shift over 16

EXPONENT DIFFERENCE	NORMALIZATION
0 — 7	2.1 $\mu$ sec per shift
8 — 15	2.1 + 2.1 per shift over 8
16 — 23	4.2 + 2.1 per shift over 16

INSTRUCTION	BASIC TIME ( $\mu$ sec)
FMUL	52.2 base time + 3.85 per "1" bit. If either argument has 8 bits of precision
FDIV	93.7 Worst Case 232 Worst Case 151 Typical

CENTRAL PROCESSOR	LSI-11	11/05	11/10	11/15	11/20	11/35	11/40	11/45
Main Market	OEM	OEM	End User	OEM	End User	OEM	End User	OEM & End User
Memory	core, MOS, ROM		core		core		core	bipolar, MOS, core
Reg to Reg Transfer	3.5 us		3.7 us		2.3 us		0.9 us	0.3 0.45 0.9
Max Mem Size (words)	32K		28K		28K 124K		124K	124K
Max Address Space	32K		32K		32K 128K		128K	128K
General Purpose Reg	8		8		8		8	16
Stack Processing	yes		yes		yes		yes	yes
Micro-programmed	yes		yes		no		yes	yes
Instructions	basic set + XOR, SOB, MARK, SXT, RTT, MTPS, MFPS		basic set		basic set		basic set + XOR, SOB, MARK, SXT, RTT	same as 11/40 + MUL, DIV, ASH, ASHC, SPL
Extended Arithmetic (hardware)	option (internal) MUL, DIV, ASH, ASHC		option (external)		option (external)		option (internal) MUL, DIV, ASH, ASHC	standard (int)
Floating Point	option, FADD, FSUB, FMUL, FDIV		software only		software only		hardware option 32-bit word 400 or programmable (option)	hardware option 32 or 64-bit word programmable
Stack Limit Address	none		400 (fixed)		400 (fixed)		option MFPI, MTPI	option MFPI, MFDP MTPI, MTPD
Memory Management	not available		not available		not available		option MFPI, MTPI	option MFPI, MFDP MTPI, MTPD
Modes	1		1		1		1 std, 2 opt	3
Automatic Priority Interrupt	1-line multi-level		4-line multi-level		1-line 4-line multi-lev multi-lev		4-line multi-level	4-line multi-level
Power Fail and Auto-Restart	standard		standard		option standard		standard	8 software levels standard



## APPENDIX D

### INSTRUCTION INDEX

<b>A</b>		FMUL .....	6-9
ADC(B) .....	4-19	FSUB .....	6-8
ADD .....	4-27	<b>H</b>	
ASL(B) .....	4-14	HALT .....	4-71
ASH .....	6-4	<b>I</b>	
ASHC .....	6-5	INC(B) .....	4-8
ASR(B) .....	4-13	IOT .....	4-66
<b>B</b>		<b>J</b>	
BCC .....	4-42	JMP .....	4-54
BCS .....	4-43	JSR .....	4-56
BEQ .....	4-37	<b>M</b>	
BGE .....	4-45	MARK .....	4-59
BGT .....	4-47	MFPS .....	4-22
BHI .....	4-50	MOV(B) .....	4-25
BHIS .....	4-52	MTPS .....	4-23
BIC(B) .....	4-31	MUL .....	6-2
BIS(B) .....	4-32	<b>N</b>	
BIT(B) .....	4-30	NEG(B) .....	4-10
BLT .....	4-46	NOP .....	4-76
BLE .....	4-48	<b>R</b>	
BLO .....	4-53	RESET .....	4-73
BLOS .....	4-51	ROL(B) .....	4-16
BMI .....	4-39	ROR(B) .....	4-15
BNE .....	4-36	RTI .....	4-67
BPL .....	4-38	RTS .....	4-58
BPT .....	4-65	RTT .....	4-68
BR .....	4-35	<b>S</b>	
BVC .....	4-40	SBC(B) .....	4-20
BVS .....	4-41	SOB .....	4-61
<b>C</b>		SUB .....	4-28
CLR(B) .....	4-6	SWAB .....	4-17
CMP(B) .....	4-26	SXT .....	4-21
COM(B) .....	4-7	<b>T</b>	
COND. CODES .....	4-76	TRAP .....	4-64
<b>D</b>		TST(B) .....	4-11
DEC(B) .....	4-9	<b>W</b>	
DIV .....	6-3	WAIT .....	4-72
<b>E</b>		<b>X</b>	
EMT .....	4-63	XOR .....	4-33
<b>F</b>			
FADD .....	6-8		
FDIV .....	6-9		

### NUMERICAL OP CODE LIST

OP Code	Mnemonic	OP Code	Mnemonic	OP Code	Mnemonic	
00 00 00	HALT	00 60 DD	ROR	10 40 00	}	
00 00 01	WAIT	00 61 DD	ROL	10 41 00		
00 00 02	RTI	00 62 DD	ASR	10 42 00		
00 00 03	BPT	00 63 DD	ASL	10 43 77		
00 00 04	IOT	00 64 NN	MARK		}	
00 00 05	RESET	00 67 DD	SXT	10 44 00		
00 00 06	RTT			10 45 00		
00 00 07	(unused)	00 70 00	(unused)	10 46 00		
00 00 77	(unused)	00 77 77	(unused)	10 47 77	TRAP	
00 01 DD	JMP			10 50 DD	CLRB	
00 02 0R	RTS			10 51 DD	COMB	
00 02 10	}	01 SS DD	MOV	10 52 DD	INCB	
00 02 11		(reserved)	02 SS DD	CMP	10 53 DD	DECB
00 02 12			03 SS DD	BIT	10 54 DD	NEGB
00 02 27			04 SS DD	BIC	10 55 DD	ADCB
00 02 40		NOP	05 SS DD	BIS	10 56 DD	SBCB
		06 SS DD	ADD	10 57 DD	TSTB	
00 02 41	}	07 0R SS	MUL	10 60 DD	RORB	
00 02 42		cond	07 1R SS	DIV	10 61 DD	ROLB
00 02 43		codes	07 2R SS	ASH	10 62 DD	ASRB
00 02 77			07 3R SS	ASHC	10 63 DD	ASLB
		07 4R DD	XOR	10 64 SS	MTPS	
				10 67 DD	MFPS	
00 03 DD	SWAB	07 50 0R	FADD	11 SS DD	MOVB	
00 04 XXX	BR	07 50 1R	FSUB	12 SS DD	CMPB	
00 10 XXX	BNE	07 50 2R	FMUL	13 SS DD	BITB	
00 14 XXX	BEQ	07 50 3R	FDIV	14 SS DD	BICB	
00 20 XXX	BGE	07 50 40	}	15 SS DD	BISB	
00 24 XXX	BLT	07 67 77		(unused)	16 SS DD	SUB
00 30 XXX	BGT					
00 34 XXX	BLE					
00 4R DD	JSR	07 7R NN	SOB			
00 50 DD	CLR	10 00 XXX	BPL			
00 51 DD	COM	10 04 XXX	BMI			
00 52 DD	INC	10 10 XXX	BHI			
00 53 DD	DEC	10 14 XXX	BLOS			
00 54 DD	NEG	10 20 XXX	BVC			
00 55 DD	ADC	10 24 XXX	BVS			
00 56 DD	SBC	10 30 XXX	BCC,			
00 57 DD	TST	10 34 XXX	BHIS			
			BCS,			
			BLO			

## SUMMARY OF LSI-11 INSTRUCTIONS

		MODE      R	
Mode	Name	Symbolic	Description
0	register	R	(R) is operand [ex. R2=%02]
1	register deferred	(R)	(R) is address
2	auto-increment	(R)+	(R) is adrs; (R) + (1 or 2)
3	auto-incr deferred	@(R)+	(R) is adrs of adrs; (R) + 2
4	auto-decrement	-(R)	(R) - (1 or 2); is adrs
5	auto-decr deferred	@-(R)	(R) - 2; (R) is adrs of adrs
6	index	X(R)	(R) + X is adrs
7	index deferred	@X(R)	(R) + X is adrs of adrs

## PROGRAM COUNTER ADDRESSING      Reg = 7

		MODE      7	
2	immediate	#n	operand n follows instr
3	absolute	@ #A	address A follows instr
6	relative	A	instr adrs + 4 + X is adrs
7	relative deferred	@A	instr adrs + 4 + X is adrs of adrs

## LEGEND

## Op Codes

■ = 0 for word/1 for byte  
 SS = source field (6 bits)  
 DD = destination field (6 bits)  
 R = gen register (3 bits), 0 to 7  
 XXX = offset (8 bits), +127 to -128  
 N = number (3 bits)  
 NN = number (6 bits)

## Boolean

^ = AND  
 v = inclusive OR  
 ^v = exclusive OR  
 ~ = NOT

## Operations

( ) = contents of  
 s = contents of source  
 d = contents of destination  
 r = contents of register  
 ← = becomes  
 X = relative address  
 % = register definition  
 , = Concatenated with

## Condition Codes

\* = conditionally set/cleared  
 - = not affected  
 0 = cleared  
 1 = set

**SINGLE OPERAND: OPR dst**


Mne-monic	Op Code	Instruction	dst	Result	N	Z	V	C
-----------	---------	-------------	-----	--------	---	---	---	---

**General**

CLR(B)	■ 050DD	clear	0	0	1	0	0	0
COM(B)	■ 051DD	complement (1's)	$\sim d$	*	*	0	1	1
INC(B)	■ 052DD	increment	$d + 1$	*	*	*	*	-
DEC(B)	■ 053DD	decrement	$d - 1$	*	*	*	*	-
NEG(B)	■ 054DD	negate (2's compl)	$-d$	*	*	*	*	*
TST(B)	■ 057DD	test	$d$	*	*	0	0	0

**Rotate & Shift**

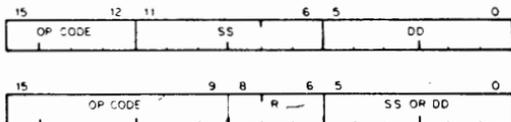
ROR(B)	■ 060DD	rotate right	$\rightarrow C, d$	*	*	*	*	*
ROL(B)	■ 061DD	rotate left	$C, d \leftarrow$	*	*	*	*	*
ASR(B)	■ 062DD	arith shift right	$d/2$	*	*	*	*	*
ASL(B)	■ 063DD	arith shift left	$2d$	*	*	*	*	*
SWAB	0003DD	swap bytes		*	*	0	0	0

**Multiple Precision**

ADC(B)	■ 055DD	add carry	$d + C$	*	*	*	*	*
SBC(B)	■ 056DD	subtract carry	$d - C$	*	*	*	*	*
SXT	0067DD	sign extend	0 or -1	-	*	0	-	-

**Processor Status (PS) Operators**

MFPS	1067DD	move byte from PS	$d \leftarrow PS$	*	*	0	-	-
MTPS	1064SS	move byte to PS	$PS \leftarrow s$	*	*	*	*	*

**DOUBLE OPERAND: OPR src, dst OPR src, R or OPR R, dst**


Mne-monic	Op Code	Instruction	Operation	N	Z	V	C
-----------	---------	-------------	-----------	---	---	---	---

**General**

MOV(B)	■ 1SSDD	move	$d \leftarrow s$	*	*	0	-
CMP(B)	■ 2SSDD	compare	$s - d$	*	*	*	*
ADD	06SSDD	add	$d \leftarrow s + d$	*	*	*	*
SUB	16SSDD	subtract	$d \leftarrow d - s$	*	*	*	*

**Logical**

BIT(B)	■ 3SSDD	bit test (AND)	$s \wedge d$	*	*	0	-
BIC(B)	■ 4SSDD	bit clear	$d \leftarrow (\sim s) \wedge d$	*	*	0	-
BIS(B)	■ 5SSDD	bit set (OR)	$d \leftarrow s \vee d$	*	*	0	-
XOR	074RDD	exclusive OR	$d \leftarrow r \oplus d$	*	*	0	-

### Optional EIS

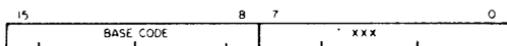
MUL	070RSS	multiply	$r \leftarrow r \times s$	* * 0 *
DIV	071RSS	divide	$r \leftarrow r/s$	* * * *
ASH	072RSS	shift		* * * *
ASHC	073RSS	arithmetically arith shift combined		* * * *

### Optional FIS

FADD	07500R	floating add		* * 0 0
FSUB	07501R	floating subtract		* * 0 0
FMUL	07502R	floating multiply		* * 0 0
FDIV	07503R	floating divide		* * 0 0

### BRANCH: B -- location

If condition is satisfied:  
Branch to location,  
New PC  $\leftarrow$  Updated PC + (2 x offset)  
 $\underbrace{\hspace{10em}}_{\text{adrs of br instr} + 2}$



Op Code = Base Code + XXX

Mne- monic	Base Code	Instruction	Branch Condition
---------------	-----------	-------------	------------------

#### Branches

BR	000400	branch (unconditional)	(always)
BNE	001000	br if not equal (to 0)	$\neq 0$ $Z = 0$
BEQ	001400	br if equal (to 0)	$= 0$ $Z = 1$
BPL	100000	branch if plus	$+$ $N = 0$
BMI	100400	branch if minus	$-$ $N = 1$
BVC	102000	br if overflow is clear	$V = 0$
BVS	102400	br if overflow is set	$V = 1$
BCC	103000	br if carry is clear	$C = 0$
BCS	103400	br if carry is set	$C = 1$

#### Signed Conditional Branches

BGE	002000	br if greater or equal (to 0)	$\geq 0$ $N \nabla V = 0$
BLT	002400	br if less than (0)	$\nabla 0$ $N \nabla V = 1$
BGT	003000	br if greater than (0)	$\nabla 0$ $Z \vee (N \nabla V) = 0$
BLE	003400	br if less or equal (to 0)	$\leq 0$ $Z \vee (N \nabla V) = 1$

#### Unsigned Conditional Branches

BHI	101000	branch if higher	$\nabla$ $C \vee Z = 0$
BLOS	101400	branch if lower or same	$\nabla$ $C \vee Z = 1$
BHIS	103000	branch if higher or same	$\nabla$ $C = 0$
BLO	103400	branch if lower	$\nabla$ $C = 1$





## TRAP VECTORS

000	(reserved)	024	Power Fail
004	Time Out & other errors	030	EMT instruction
010	illegal & reserved instr	034	TRAP instruction
014	BPT instruction	244	FIS (optional)
020	IOT instruction		

## ODT COMMANDS

Format	Description
RETURN	Close opened location and accept next command.
LINE FEED	Close current location; open next sequential location.
↑	Open previous location.
←	Take contents of opened location, index by contents of PC, and open that location. (ASCII 137)
@	Take contents of opened location as absolute address and open that location.
r/	Open the word at location r.
/	Reopen the last location.
\$n/or Rn/	Open general register n (0-7) or S (PS register).
r;G or rG	Go to location r and start program.
nL	Execute bootstrap loader using n as device CSR. Console device address is 177560.
;P or P	Proceed with program execution.
RUBOUT	Erases previous numeric character. Response is a backslash (\).

## 7-BIT ASCII CODE

Octal Code	Char						
000	NUL	040	SP	100	@	140	\
001	SOH	041	!	101	A	141	a
002	STX	042	"	102	B	142	b
003	ETX	043	#	103	C	143	c
004	EOT	044	\$	104	D	144	d
005	ENQ	054	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BEL	047	'	107	G	147	g
010	BS	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	.	112	J	152	j
013	VT	053	+	113	K	153	k
014	FF	054		114	L	154	l
015	CR	055	-	115	M	155	m
016	SO	056	.	116	N	156	n
017	SI	057	/	117	O	157	o
020	DLE	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	CAN	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB	072	:	132	Z	172	z
033	ESC	073	:	133	[	173	{
034	FS	074	<	134	\	174	
035	GS	075	>	135	]	175	}
036	RS	076	>	136	^	176	~
037	US	077	?	137	_	177	DEL









# digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone: (617) 897-5111

## SALES AND SERVICE OFFICES

UNITED STATES—ALABAMA, Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, El Segundo, Los Angeles, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Santa Clara, Stanford, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Lanham, MD) • FLORIDA, Ft. Lauderdale and Orlando • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans (Metairie) • MARYLAND, Odenton • MASSACHUSETTS, Marlborough, Waltham and Westfield • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City (Independence) and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo (Cheektowaga), Long Island (Huntington Station), Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland (Euclid), Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Eugene and Portland • PENNSYLVANIA, Allentown, Philadelphia (Bluebell) and Pittsburgh • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) •

INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver and Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Grenoble and Paris • GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich and Stuttgart • HONG KONG • INDIA, Bombay • INDONESIA, Jakarta • IRELAND, Dublin • ITALY, Milan and Turin • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland • NORWAY, Oslo • PUERTO RICO, Santurce • SINGAPORE • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Bristol, Edinburgh, Leeds, London, Manchester and Reading • VENEZUELA, Caracas •



digital