

## APPENDIX



# The LC-3 ISA

## A.1 Overview

The instruction set architecture (ISA) of the LC-3 is defined as follows:

**Memory address space** 16 bits, corresponding to  $2^{16}$  locations, each containing one word (16 bits). Addresses are numbered from 0 (i.e., x0000) to 65,535 (i.e., xFFFF). Addresses are used to identify memory locations and memory-mapped I/O device registers. Certain regions of memory are reserved for special uses, as described in Figure A.1.

Locations x0000 to x2FFF comprise privileged memory and are only accessible if the process is executing in Supervisor mode (PSR[15]=0).

Locations x3000 to xFDFF comprise memory available to User mode and

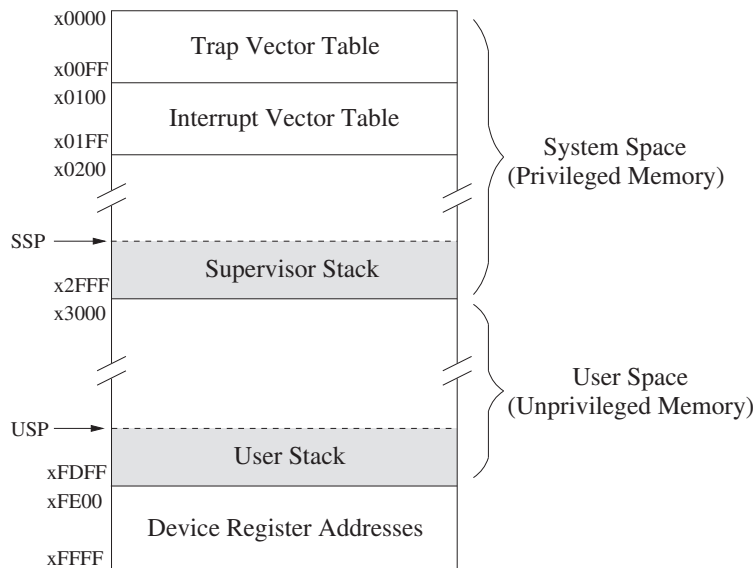


Figure A.1 Memory map of the LC-3

Table A.1 Device Register Assignments		
Address	I/O Register Name	I/O Register Function
xFE00	Keyboard status register (KBSR)	The ready bit (bit [15]) indicates if the keyboard has received a new character.
xFE02	Keyboard data register (KBDR)	Bits [7:0] contain the last character typed on the keyboard.
xFE04	Display status register (DSR)	The ready bit (bit [15]) indicates if the display device is ready to receive another character to print on the screen.
xFE06	Display data register (DDR)	A character written in bits [7:0] will be displayed on the screen.
xFFFC	Processor Status Register (PSR)	Contains privilege mode, priority level and condition codes of the currently executing process.
xFFFE	Machine control register (MCR)	Bit [15] is the clock enable bit. When cleared, instruction processing stops.

data. Addresses xFE00 to xFFFF specify input and output device registers and special internal processor registers that are also only accessible if the process is executing in Supervisor mode (PSR[15]=0). For purposes of controlling access to these device registers, their addresses are also considered part of privileged memory.

**Memory-mapped I/O** Input and output are handled by load/store (LD/ST, LDI/STI, LDR/STR) instructions using memory addresses from xFE00 to xFFFF to designate each device register. Table A.1 lists the input and output device registers and internal processor registers that have been specified for the LC-3 thus far, along with their corresponding assigned addresses from the memory address space.

**Bit numbering** Bits of all quantities are numbered, from right to left, starting with bit 0. The leftmost bit of the contents of a memory location is bit 15.

**Instructions** Instructions are 16 bits wide. Bits [15:12] specify the opcode (operation to be performed); bits [11:0] provide further information that is needed to execute the instruction. The specific operation of each LC-3 instruction is described in Section A.2.

**Illegal opcode exception** Bits [15:12] = 1101 has not been specified. If an instruction contains 1101 in bits [15:12], an illegal opcode exception occurs. Section A.3 explains what happens.

**Program counter** A 16-bit register containing the address of the next instruction to be processed.

**General purpose registers** Eight 16-bit registers, numbered from 000 to 111 (R0 to R7).

**Condition codes** Three 1-bit registers: N (negative), Z (zero), and P (positive). Load instructions (LD, LDI, and LDR) and operate instructions (ADD, AND, and NOT) each load a result into one of the eight general purpose registers. The condition codes are set, based on whether that result, taken as a 16-bit 2’s complement integer, is negative (N = 1; Z, P = 0), zero (Z = 1; N, P = 0), or positive (P = 1; N, Z = 0). All other LC-3 instructions leave the condition codes unchanged.

**Interrupt processing** I/O devices have the capability of interrupting the processor. Section A.3 describes the mechanism.

**Priority level** The LC-3 supports eight levels of priority. Priority level 7 (PL7) is the highest, PL0 is the lowest. The priority level of the currently executing process is specified in bits PSR[10:8].

**Processor status register (PSR)** A 16-bit register, containing status information about the currently executing process. Seven bits of the PSR have been defined thus far. PSR[15] specifies the privilege mode of the executing process. PSR[10:8] specifies the priority level of the currently executing process. PSR[2:0] contains the condition codes. PSR[2] is N, PSR[1] is Z, and PSR[0] is P.

**Supervisor mode** The LC-3 specifies two modes of operation, Supervisor mode (privileged) and User mode (unprivileged). Interrupt service routines and trap service routines (i.e., system calls) execute in Supervisor mode. The privilege mode is specified by PSR[15]. PSR[15]=0 indicates Supervisor mode; PSR[15]=1 indicates User mode.

**Privilege mode exception** The RTI instruction executes in Supervisor mode. If the processor attempts to execute the RTI instruction while in User mode, a privilege mode exception occurs. Section A.3 explains what happens.

**Access Control Violation (ACV) exception** An ACV exception occurs if a process attempts to access a location in privileged memory (either a location in system space or a device register having an address from xFE00 to xFFFF) while operating in User mode. Section A.3 explains what happens.

**Supervisor stack** A region of memory in system space accessible via the Supervisor Stack Pointer (SSP). When PSR[15]=0, the stack pointer (R6) is SSP. When the processor is operating in User mode (PSR[15]=1), the SSP is stored in Saved\_SSP.

**User stack** A region of memory in user space accessible via the User Stack Pointer (USP). When PSR[15]=1, the stack pointer (R6) is USP. When the processor is operating in Supervisor mode (PSR[15]=0), the USP is stored in Saved\_USP.

## A.2 The Instruction Set

The LC-3 supports a rich, but lean, instruction set. Each 16-bit instruction consists of an opcode (bits[15:12]) plus 12 additional bits to specify the other information that is needed to carry out that instruction. Figure A.2 summarizes the 15 different opcodes in the LC-3 and the specification of the remaining bits of each instruction. The 16th four-bit opcode is not specified but is reserved for future use.

In the following pages, the instructions will be described in greater detail. Table A.2 is provided to help you to understand those descriptions. For each instruction, we show the assembly language representation, the format of the 16-bit instruction, the operation of the instruction, an English-language description of its operation, and one or more examples of the instruction. Where relevant, additional notes about the instruction are also provided.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD <sup>+</sup>	0001				DR			SR1			0	00		SR2		
ADD <sup>+</sup>	0001				DR			SR1			1	imm5				
AND <sup>+</sup>	0101				DR			SR1			0	00		SR2		
AND <sup>+</sup>	0101				DR			SR1			1	imm5				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR	0100				1	PCoffset11										
JSRR	0100				0	00		BaseR			000000					
LD <sup>+</sup>	0010				DR			PCoffset9								
LDI <sup>+</sup>	1010				DR			PCoffset9								
LDR <sup>+</sup>	0110				DR			BaseR			offset6					
LEA	1110				DR			PCoffset9								
NOT <sup>+</sup>	1001				DR			SR			111111					
RET	1100				000			111			000000					
RTI	1000				000000000000											
ST	0011				SR			PCoffset9								
STI	1011				SR			PCoffset9								
STR	0111				SR			BaseR			offset6					
TRAP	1111				0000			trapvect8								
reserved	1101															

Figure A.2 Format of the entire LC-3 instruction set. *Note:* + indicates instructions that modify condition codes

**Table A.2 Notational Conventions**

Notation	Meaning
xNumber	The number in hexadecimal notation. Example: xF2A1
#Number	The number in decimal notation. Example #793
bNumber	The number in binary. Example b10011
A[l:r]	The field delimited by bit [l] on the left and bit [r] on the right, of the datum A. For example, if PC contains 0011001100111111, then PC[15:9] is 0011001. PC[2:2] is 1. If l and r are the same bit number, we generally write PC[2].
BaseR	Base Register; one of R0..R7, specified by bits [8:6] of the instruction, used in conjunction with a six-bit offset to compute Base+offset addresses (LDR and STR), or alone to identify the target address of a control instruction (JMP and JSRR).
DR	Destination Register; one of R0..R7, which specifies the register a result should be written to.
imm5	A five-bit immediate value (bits [4:0] of an instruction), when used as a literal (immediate) value. Taken as a five-bit, 2's complement integer, it is sign-extended to 16 bits before it is used. Range: -16..15.
INTV	An eight-bit value, supplied along with an interrupting event; used to determine the starting address of an interrupt service routine. The eight bits form an offset from the starting address of the interrupt vector table. The corresponding location in the interrupt vector table contains the starting address of the corresponding interrupt service routine. Range 0..255.
LABEL	An assembly language construct that identifies a location symbolically (i.e., by means of a name, rather than its 16-bit address).
mem[address]	Denotes the contents of memory at the given address.
offset6	A six-bit signed 2's complement integer (bits [5:0] of an instruction), used with the Base+offset addressing mode. Bits [5:0] are sign-extended to 16 bits and then added to the Base Register to form an address. Range: -32..31.
PC	Program Counter; 16-bit register that contains the memory address of the next instruction to be fetched. For example, if the instruction at address A is not a control instruction, during its execution, the PC contains the address A + 1, indicating that the next instruction to be executed is contained in memory location A + 1.
PCOffset9	A nine-bit signed 2's complement integer (bits [8:0] of an instruction), used with the PC+offset addressing mode. Bits [8:0] are sign-extended to 16 bits and then added to the incremented PC to form an address. Range -256..255.
PCOffset11	An eleven-bit signed 2's complement integer (bits [10:0] of an instruction), used with the JSR opcode to compute the target address of a subroutine call. Bits [10:0] are sign-extended to 16 bits and then added to the incremented PC to form the target address. Range -1024..1023.
PSR	Processor Status Register. A 16-bit register that contains status information of the process that is executing. Seven bits of the PSR have been specified. PSR[15] = privilege mode. PSR[10:8] = Priority Level. PSR[2:0] contains the condition codes. PSR[2] = N, PSR[1] = Z, PSR[0] = P.
Saved_SSP	Saved Supervisor Stack Pointer. The processor is executing in either Supervisor mode or User mode. If in User mode, R6, the stack pointer, is the User Stack Pointer (USP). The Supervisor Stack Pointer (SSP) is stored in Saved_SSP. When the privilege mode changes from User mode to Supervisor mode, Saved_USP is loaded with R6 and R6 is loaded with Saved_SSP.
Saved_USP	Saved User Stack Pointer. The User Stack Pointer is stored in Saved_USP when the processor is executing in Supervisor mode. See Saved_SSP.
setcc()	Indicates that condition codes N, Z, and P are set based on the value of the result written to DR.
SEXT(A)	Sign-extend A. The most significant bit of A is replicated as many times as necessary to extend A to 16 bits. For example, if A = 110000, then SEXT(A) = 1111 1111 1111 0000.
SP	The current stack pointer. R6 is the current stack pointer. There are two stacks, one for each privilege mode. SP is SSP if PSR[15] = 0; SP is USP if PSR[15] = 1.
SR, SR1, SR2	Source register; one of R0..R7 that specifies the register from which a source operand is obtained.
SSP	The Supervisor Stack Pointer.
trapvect8	An eight-bit value (bits [7:0] of an instruction), used with the TRAP opcode to determine the starting address of a trap service routine. Bits [7:0] are taken as an unsigned integer and zero-extended to 16 bits. This is the address of the memory location containing the starting address of the corresponding service routine. Range 0..255.
USP	The User Stack Pointer.
ZEXT(A)	Zero-extend A. Zeros are appended to the leftmost bit of A to extend it to 16 bits. For example, if A = 110000, then ZEXT(A) = 0000 0000 0011 0000.

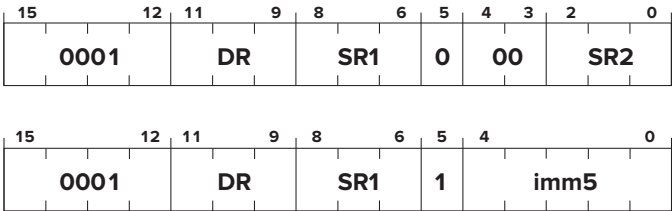
ADD

Addition

Assembler Formats

```
ADD DR, SR1, SR2
ADD DR, SR1, imm5
```

Encodings



Operation

```
if (bit[5] == 0)
    DR = SR1 + SR2;
else
    DR = SR1 + SEXT(imm5);
setcc();
```

Description

If bit [5] is 0, the second source operand is obtained from SR2. If bit [5] is 1, the second source operand is obtained by sign-extending the imm5 field to 16 bits. In both cases, the second source operand is added to the contents of SR1 and the result stored in DR. The condition codes are set, based on whether the result is negative, zero, or positive.

Examples

```
ADD R2, R3, R4    ; R2 ← R3 + R4
ADD R2, R3, #7     ; R2 ← R3 + 7
```

# AND

## Bit-wise Logical AND

### Assembler Formats

```
AND DR, SR1, SR2
AND DR, SR1, imm5
```

### Encodings

15	12	11	9	8	6	5	4	3	2	0
0101				DR		SR1		0	00	SR2

15	12	11	9	8	6	5	4			0
0101				DR		SR1		1	imm5	

### Operation

```
if (bit[5] == 0)
    DR = SR1 AND SR2;
else
    DR = SR1 AND SEXT(imm5);
setcc();
```

### Description

If bit [5] is 0, the second source operand is obtained from SR2. If bit [5] is 1, the second source operand is obtained by sign-extending the imm5 field to 16 bits. In either case, the second source operand and the contents of SR1 are bit-wise ANDed and the result stored in DR. The condition codes are set, based on whether the binary value produced, taken as a 2's complement integer, is negative, zero, or positive.

### Examples

```
AND R2, R3, R4    ;R2 ← R3 AND R4
AND R2, R3, #7     ;R2 ← R3 AND 7
```

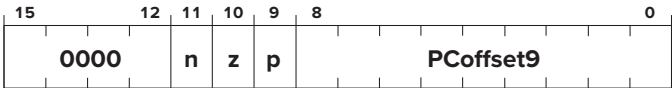
BR

Conditional Branch

Assembler Formats

BRn	LABEL	BRzp	LABEL
BRz	LABEL	BRnp	LABEL
BRp	LABEL	BRnz	LABEL
BR <sup>†</sup>	LABEL	BRnzp	LABEL

Encoding



Operation

if ((n AND N) OR (z AND Z) OR (p AND P))  
PC = PC<sup>‡</sup> + SEXT(PCoffset9);

Description

The condition codes specified by bits [11:9] are tested. If bit [11] is 1, N is tested; if bit [11] is 0, N is not tested. If bit [10] is 1, Z is tested, etc. If any of the condition codes tested is 1, the program branches to the memory location specified by adding the sign-extended PCoffset9 field to the incremented PC.

Examples

BRzp LOOP ; Branch to LOOP if the last result was zero or positive.  
BR<sup>†</sup> NEXT ; Unconditionally branch to NEXT.

<sup>†</sup>The assembly language opcode BR is interpreted the same as BRnzp; that is, always branch to the target address.  
<sup>‡</sup>This is the incremented PC.



# JMP

# RET

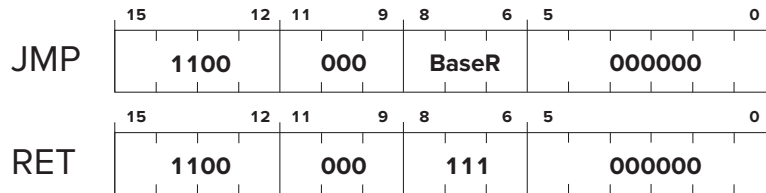
Jump

Return from Subroutine

## Assembler Formats

```
JMP BaseR
RET
```

## Encoding



## Operation

$PC = \text{BaseR};$

## Description

The program unconditionally jumps to the location specified by the contents of the base register. Bits [8:6] identify the base register.

## Examples

```
JMP R2    ; PC ← R2
RET        ; PC ← R7
```

## Note

The RET instruction is a special case of the JMP instruction, normally used in the return from a subroutine. The PC is loaded with the contents of R7, which contains the linkage back to the instruction following the subroutine call instruction.

JSR

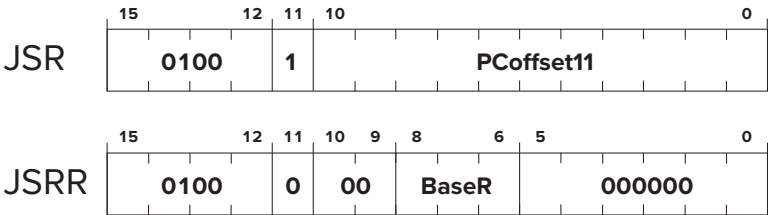
JSRR

Jump to Subroutine

Assembler Formats

```
JSR LABEL
JSRR BaseR
```

Encoding



Operation

```
TEMP = PC;†
if (bit[11] == 0)
    PC = BaseR;
else
    PC = PC† + SEXT(PCoffset11);
R7 = TEMP;
```

Description

First, the incremented PC is saved in a temporary location. Then the PC is loaded with the address of the first instruction of the subroutine, which will cause an unconditional jump to that address after the current instruction completes execution. The address of the subroutine is obtained from the base register (if bit [11] is 0), or the address is computed by sign-extending bits [10:0] and adding this value to the incremented PC (if bit [11] is 1). Finally, R7 is loaded with the value stored in the temporary location. This is the linkage back to the calling routine.

Examples

```
JSR QUEUE ; Put the address of the instruction following JSR into R7;
           ; Jump to QUEUE.
JSRR R3   ; Put the address of the instruction following JSRR into R7;
           ; Jump to the address contained in R3.
```

†This is the incremented PC.

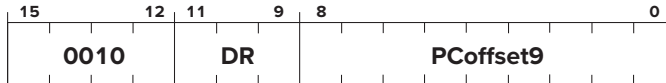
# LD

Load

## Assembler Format

```
LD DR, LABEL
```

## Encoding



## Operation

```
if (computed address is in privileged memory AND PSR[15] == 1)
    Initiate ACV exception;
else
    DR = mem[PC† + SEXT(PCoffset9)];
    setcc();
```

## Description

An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. If the address is to privileged memory and PSR[15]=1, initiate ACV exception. If not, the contents of memory at this address is loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

## Example

```
LD R4, VALUE ; R4 ← mem[VALUE]
```

---

<sup>†</sup>This is the incremented PC.

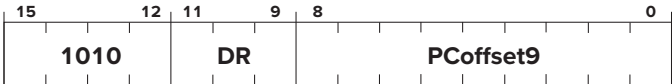
LDI

Load Indirect

Assembler Format

LDI DR, LABEL

Encoding



Operation

```
if (either computed address is in privileged memory AND PSR[15] == 1)
    Initiate ACV exception;
else
    DR = mem[mem[PC† + SEXT(PCoffset9)]];
    setcc();
```

Description

An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. What is stored in memory at this address is the address of the data to be loaded into DR. If either address is to privileged memory and PSR[15]=1, initiate ACV exception. If not, the data is loaded and the condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

LDI R4, ONEMORE ; R4 ← mem[mem[ONEMORE]]

†This is the incremented PC.

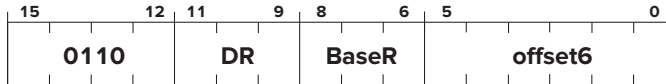
# LDR

## Load Base+offset

### Assembler Format

LDR DR, BaseR, offset6

### Encoding



### Operation

```

If (computed address is in privileged memory AND PSR[15] == 1)
    Initiate ACV exception;
else
    DR = mem[BaseR + SEXT(offset6)];
    setcc();
  
```

### Description

An address is computed by sign-extending bits [5:0] to 16 bits and adding this value to the contents of the register specified by bits [8:6]. If the computed address is to privileged memory and PSR[15]=1, initiate ACV exception. If not, the contents of memory at this address is loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

### Example

LDR R4, R2, #-5 ; R4 ← mem[R2 - 5]

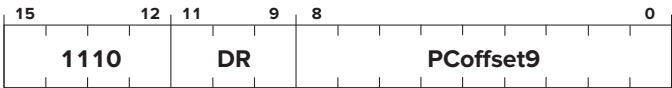
LEA

Load Effective Address

Assembler Format

LEA    DR, LABEL

Encoding



Operation

$DR = PC^{\dagger} + \text{SEXT}(\text{PCoffset9});$

Description

An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. This address is loaded into DR.<sup>‡</sup>

Example

LEA    R4, TARGET        ; R4 ← address of TARGET.

<sup>†</sup>This is the incremented PC.

<sup>‡</sup>The LEA instruction computes an address but does NOT read memory. Instead, the address itself is loaded into DR.

# NOT

## Bit-Wise Complement

### Assembler Format

NOT DR, SR

### Encoding

15	12	11	9	8	6	5	4	3	2	0
1001				DR		SR		1	1111	

### Operation

DR = NOT(SR);  
setcc();

### Description

The bit-wise complement of the contents of SR is stored in DR. The condition codes are set, based on whether the binary value produced, taken as a 2's complement integer, is negative, zero, or positive.

### Example

NOT R4, R2 ; R4 ← NOT(R2)

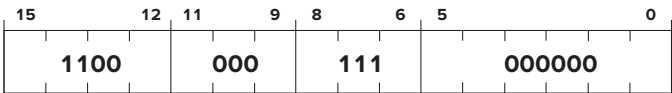
RET

Return from Subroutine

Assembler Format

RET<sup>†</sup>

Encoding



Operation

PC = R7;

Description

The PC is loaded with the value in R7. Its normal use is to cause a return from a previous JSR(R) instruction.

Example

RET ; PC ← R7

<sup>†</sup>The RET instruction is a specific encoding of the JMP instruction. See also JMP.

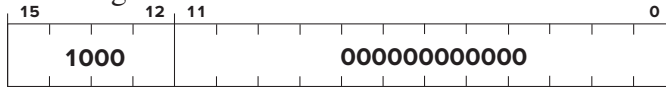


# RTI

## Return from Trap or Interrupt

### Assembler Format

#### RTI Encoding



### Operation

```

if (PSR[15] == 1)
    Initiate a privilege mode exception;
else
    PC=mem[R6]; R6 is the SSP, PC is restored
    R6=R6+1;
    TEMP=mem[R6];
    R6=R6+1; system stack completes POP before saved PSR is restored
    PSR=TEMP; PSR is restored
    if (PSR[15] == 1)
        Saved_SSP=R6 and R6=Saved_USP;
  
```

### Description

If the processor is running in User mode, a privilege mode exception occurs. If in Supervisor mode, the top two elements on the system stack are popped and loaded into PC, PSR. After PSR is restored, if the processor is running in User mode, the SSP is saved in Saved\_SSP, and R6 is loaded with Saved\_USP.

### Example

RTI ; PC, PSR ← top two values popped off stack.

### Note

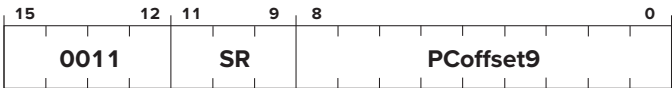
RTI is the last instruction in both interrupt and trap service routines and returns control to the program that was running. In both cases, the relevant service routine is initiated by first pushing the PSR and PC of the program that is running onto the system stack. Then the starting address of the appropriate service routine is loaded into the PC, and the service routine executes with supervisor privilege. The last instruction in the service routine is RTI, which returns control to the interrupted program by popping two values off the supervisor stack to restore the PC and PSR. In the case of an interrupt, the PC is restored to the address of the instruction that was about to be processed when the interrupt was initiated. In the case of an exception, the PC is restored to either the address of the instruction that caused the exception or the address of the following instruction, depending on whether the instruction that caused the exception is to be re-executed. In the case of a TRAP service routine, the PC is restored to the instruction following the TRAP instruction in the calling routine. In the case of an interrupt or TRAP, the PSR is restored to the value it had when the interrupt was initiated. In the case of an exception, the PSR is restored to the value it had when the exception occurred or to some modified value, depending on the exception. See also Section A.3.

# ST

## Assembler Format

ST SR, LABEL

## Encoding



## Operation

```
if (computed address is in privileged memory AND PSR[15] == 1)
    Initiate ACV exception;
else
    mem[PC† + SEXT(PCoffset9)] = SR;
```

## Description

If the computed address is to privileged memory and PSR[15]=1, initiate ACV exception. If not, the contents of the register specified by SR is stored in the memory location whose address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC.

## Example

ST R4, HERE ; mem[HERE] ← R4

<sup>†</sup>This is the incremented PC.

# STI

## Store Indirect

### Assembler Format

STI SR, LABEL

### Encoding



### Operation

```

if (either computed address is in privileged memory AND PSR[15] == 1)
    Initiate ACV exception;
else
    mem[mem[PC† + SEXT(PCOffset9)]] = SR;

```

### Description

If either computed address is to privileged memory and PSR[15]=1, initiate ACV exception. If not, the contents of the register specified by SR is stored in the memory location whose address is obtained as follows: Bits [8:0] are sign-extended to 16 bits and added to the incremented PC. What is in memory at this address is the address of the location to which the data in SR is stored.

### Example

```
STI R4, NOT_HERE ; mem[mem[NOT_HERE]] ← R4
```

---

<sup>†</sup>This is the incremented PC.

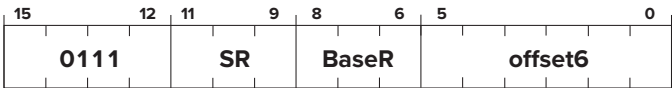
STR

Store Base+offset

Assembler Format

STR SR, BaseR, offset6

Encoding



Operation

```
if (computed address is in privileged memory AND PSR[15] == 1)
    Initiate ACV exception;
else
    mem[BaseR + SEXT(offset6)] = SR;
```

Description

If the computed address is to privileged memory and PSR[15]=1, initiate ACV exception. If not, the contents of the register specified by SR is stored in the memory location whose address is computed by sign-extending bits [5:0] to 16 bits and adding this value to the contents of the register specified by bits [8:6].

Example

STR R4, R2, #5 ; mem[R2+5] ← R4

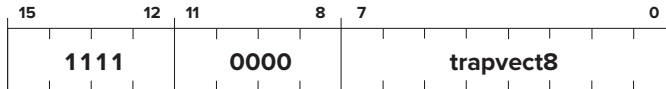
# TRAP

## System Call

### Assembler Format

TRAP trapvector8

### Encoding



### Operation

```
TEMP=PSR;
if (PSR[15] == 1)
    Saved_USP=R6 and R6=Saved_SSP;
    PSR[15]=0;
Push TEMP,PC† on the system stack
PC=mem[ZEXT(trapvect8)];
```

### Description

If the the program is executing in User mode, the User Stack Pointer must be saved and the System Stack Pointer loaded. Then the PSR and PC are pushed on the system stack. (This enables a return to the instruction physically following the TRAP instruction in the original program after the last instruction in the service routine (RTI) has completed execution.) Then the PC is loaded with the starting address of the system call specified by trapvector8. The starting address is contained in the memory location whose address is obtained by zero-extending trapvector8 to 16 bits.

### Example

```
TRAP x23 ; Directs the operating system to execute the IN system call.
          ; The starting address of this system call is contained in
          ; memory location x0023.
```

### Note:

Memory locations x0000 through x00FF, 256 in all, are available to contain starting addresses for system calls specified by their corresponding trap vectors. This region of memory is called the Trap Vector Table. Table A.3 describes the functions performed by the service routines corresponding to trap vectors x20 to x25.

<sup>†</sup>This is the incremented PC.

Unused Opcode

Assembler Format

none

Encoding



Operation

Initiate an illegal opcode exception.

Description

If an illegal opcode is encountered, an illegal opcode exception occurs.

*Note:*

The opcode 1101 has been reserved for future use. It is currently not defined. If the instruction currently executing has bits [15:12] = 1101, an illegal opcode exception occurs. Section A.3 describes what happens.

**Table A.3** Trap Service Routines

Trap Vector	Assembler Name	Description
x20	GETC	Read a single character from the keyboard. The character is not echoed onto the console. Its ASCII code is copied into R0. The high eight bits of R0 are cleared.
x21	OUT	Write a character in R0[7:0] to the console display.
x22	PUTS	Write a string of ASCII characters to the console display. The characters are contained in consecutive memory locations, one character per memory location, starting with the address specified in R0. Writing terminates with the occurrence of x0000 in a memory location.
x23	IN	Print a prompt on the screen and read a single character from the keyboard. The character is echoed onto the console monitor, and its ASCII code is copied into R0. The high eight bits of R0 are cleared.
x24	PUTSP	Write a string of ASCII characters to the console. The characters are contained in consecutive memory locations, two characters per memory location, starting with the address specified in R0. The ASCII code contained in bits [7:0] of a memory location is written to the console first. Then the ASCII code contained in bits [15:8] of that memory location is written to the console. (A character string consisting of an odd number of characters to be written will have x00 in bits [15:8] of the memory location containing the last character to be written.) Writing terminates with the occurrence of x0000 in a memory location.
x25	HALT	Halt execution and print a message on the console.

## A.3 Interrupt and Exception Processing

As has been discussed in detail in Chapter 9, events external to the program that is running can interrupt the processor. A common example of an external event is interrupt-driven I/O. It is also the case that the processor can be interrupted by exceptional events that occur while the program is running that are caused by the program itself. An example of such an “internal” event is the presence of an unused opcode in the computer program that is running.

Associated with each event that can interrupt the processor is an eight-bit vector that provides an entry point into a 256-entry *interrupt vector table*. The starting address of the interrupt vector table is x0100. That is, the interrupt vector table occupies memory locations x0100 to x01FF. Each entry in the interrupt vector table contains the starting address of the service routine that handles the needs of the corresponding event. These service routines execute in Supervisor mode.

Half (128) of these entries, locations x0100 to x017F, provide the starting addresses of routines that service events caused by the running program itself. These routines are called *exception service routines* because they handle exceptional events, that is, events that prevent the program from executing normally. The other half of the entries, locations x0180 to x01FF, provide the starting addresses of routines that service events that are external to the program that is running, such as requests from I/O devices. These routines are called *interrupt service routines*.

### A.3.1 Interrupts

At this time, an LC-3 computer system provides only one I/O device that can interrupt the processor. That device is the keyboard. It interrupts at priority level PL4 and supplies the interrupt vector x80.

An I/O device can interrupt the processor if it wants service, if its interrupt enable (IE) bit is set, and if the priority of its request is greater than the priority of any other event that wants to interrupt and greater than the priority of the program that is running.

Assume a program is running at a priority level less than 4, and someone strikes a key on the keyboard. If the IE bit of the KBSR is 1, the currently executing program is interrupted at the end of the current instruction cycle. The interrupt service routine is **initiated** as follows:

1. The PSR of the interrupted process is saved in TEMP.
2. The processor sets the privilege mode to Supervisor mode (PSR[15]=0).
3. The processor sets the priority level to PL4, the priority level of the interrupting device (PSR[10:8]=100).
4. If the interrupted process is in User mode, R6 is saved in Saved\_USP and R6 is loaded with the Supervisor Stack Pointer (SSP).
5. TEMP and the PC of the interrupted process are pushed onto the supervisor stack.
6. The keyboard supplies its eight-bit interrupt vector, in this case x80.
7. The processor expands that vector to x0180, the corresponding 16-bit address in the interrupt vector table.
8. The PC is loaded with the contents of memory location x0180, the address of the first instruction in the keyboard interrupt service routine.

The processor then begins execution of the interrupt service routine.

The last instruction executed in an interrupt service routine is RTI. The top two elements of the supervisor stack are popped and loaded into the PC and PSR registers. R6 is loaded with the appropriate stack pointer, depending on the new value of PSR[15]. Processing then continues where the interrupted program left off.

### A.3.2 Exceptions

At this time, the LC-3 ISA specifies three exception conditions: privilege mode violation, illegal opcode, and access control violation (ACV). The privilege mode violation occurs if the processor attempts to execute the RTI instruction while running in User mode. The illegal opcode exception occurs if the processor attempts to execute an instruction having the unused opcode (bits [15:12] = 1101). The ACV exception occurs if the processor attempts to access privileged memory (i.e., a memory location in system space or a device register having an address from xFE00 to xFFFF while running in User mode).

Exceptions are handled as soon as they are detected. They are *initiated* very much like interrupts are initiated, that is:

1. The PSR of the process causing the exception is saved in TEMP.



2. The processor sets the privilege mode to Supervisor mode (PSR[15]=0).
3. If the process causing the exception is in User mode, R6 is saved in Saved\_USP and R6 is loaded with the SSP.
4. TEMP and the PC of the process causing the exception are pushed onto the supervisor stack.
5. The exception supplies its eight-bit vector. In the case of the privilege mode violation, that vector is x00. In the case of the illegal opcode, that vector is x01. In the case of the ACV exception, that vector is x02.
6. The processor expands that vector to x0100, x0101, or x0102, the corresponding 16-bit address in the interrupt vector table.
7. The PC is loaded with the contents of memory location x0100, x0101, or x0102, the address of the first instruction in the corresponding exception service routine.

The processor then begins execution of the exception service routine.

The details of the exception service routine depend on the exception and the way in which the operating system wishes to handle that exception.

In many cases, the exception service routine can correct any problem caused by the exceptional event and then continue processing the original program. In those cases, the last instruction in the exception service routine is RTI, which pops the top two elements from the supervisor stack and loads them into the PC and PSR registers. The program then resumes execution with the problem corrected.

In some cases, the cause of the exceptional event is sufficiently catastrophic that the exception service routine removes the program from further processing.

Another difference between the handling of interrupts and the handling of exceptions is the priority level of the processor during the execution of the service routine. In the case of exceptions, we normally do not change the priority level when we service the exception. The priority level of a program is the urgency with which it needs to be executed. In the case of the exceptions specified by the LC-3 ISA, the urgency of a program is not changed by the fact that a privilege mode violation occurred or there was an illegal opcode in the program or the program attempted to access privileged memory while it was in User mode.





# The Microarchitecture of the LC-3

**W**e have seen in Chapters 4 and 5 the several stages of the instruction cycle that must occur in order for the computer to process each instruction. If a microarchitecture is to implement an ISA, it must be able to carry out this instruction cycle for every instruction in the ISA. This appendix illustrates one example of a microarchitecture that can do that for the LC-3 ISA. Many of the details of the microarchitecture and the reasons for each design decision are well beyond the scope of an introductory course. However, for those who want to understand **how** a microarchitecture can carry out the requirements of each instruction of the LC-3 ISA, this appendix is provided.

## C.1 Overview

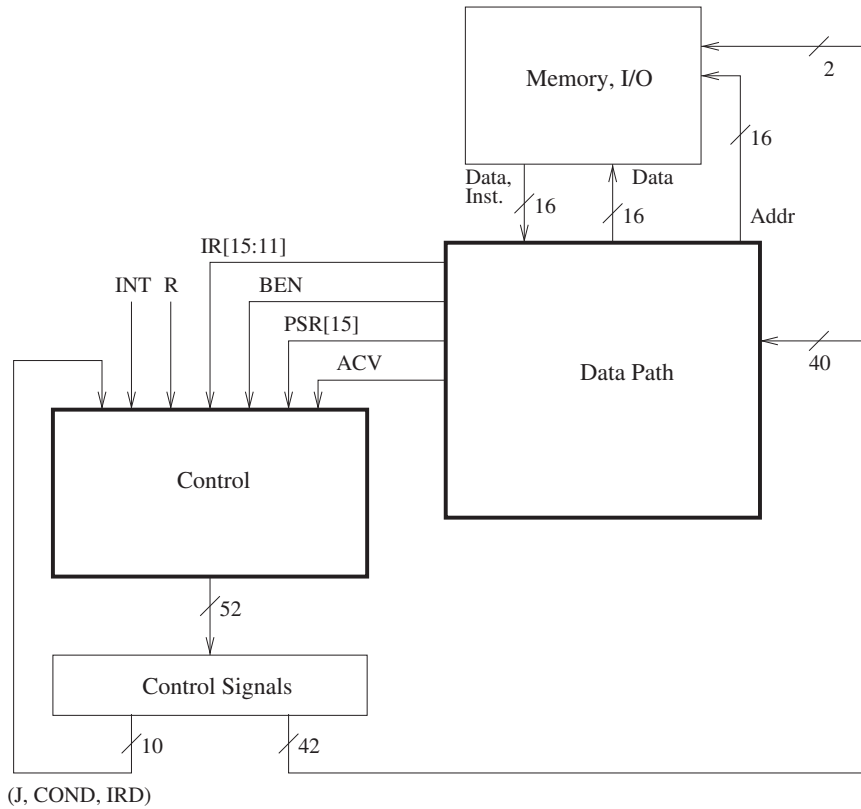
Figure C.1 shows the two main components of a microarchitecture: the *data path*, which contains all the components that actually process the instructions, and the *control*, which contains all the components that generate the set of control signals that are needed to control the processing at each instant of time.

We say, “at each instant of time,” but we really mean **during each clock cycle**. That is, time is divided into *clock cycles*. The cycle time of a microprocessor is the duration of a clock cycle. A common cycle time for a microprocessor today is 0.33 nanoseconds, which corresponds to 3 billion clock cycles each second. We say that such a microprocessor is operating at a frequency of 3 gigahertz, or 3 GHz.

At each instant of time—or, rather, during each clock cycle—the 52 control signals (as shown in Figure C.1) control both the processing in the data path and the generation of the control signals for the next clock cycle. Processing in the data path is controlled by 42 bits, and the generation of the control signals for the next clock cycle is controlled by 10 bits.

Note that the hardware that determines which control signals are needed each clock cycle does not operate in a vacuum. On the contrary, the control signals needed in the “next” clock cycle depend on the following:

1. The control signals that are present during the current clock cycle.
2. The LC-3 instruction that is being executed.



**Figure C.1** Microarchitecture of the LC-3, major components.

3. The privilege mode of the program that is executing, and whether the processor has the right to access a particular memory location.
4. If that LC-3 instruction is a BR, whether the conditions for the branch have been met (i.e., the state of the relevant condition codes).
5. Whether or not an external device is requesting that the processor be interrupted.
6. If a memory operation is in progress, whether it is completing during this cycle.

Figure C.1 identifies the specific information in our implementation of the LC-3 that corresponds to these six items. They are, respectively:

1. J[5:0], COND[2:0], and IRD—ten bits of control signals provided by the current clock cycle.
2. inst[15:12], which identifies the opcode, and inst[11:11], which differentiates JSR from JSRR (i.e., the addressing mode for the target of the subroutine call).
3. PSR[15], bit [15] of the Processor Status Register, which indicates whether the current program is executing with supervisor or user privileges, and ACV, a signal that informs the processor that a process operating in User

mode is trying to access a location in privileged memory. ACV stands for Access Control Violation. When asserted, it denies the process access to the privileged memory location.

4. BEN to indicate whether or not a BR should be taken.
5. INT to indicate that some external device of higher priority than the executing process requests service.
6. R to indicate the end of a memory operation.

## C.2 The State Machine

The behavior of the LC-3 microarchitecture during a given clock cycle is completely determined by the 52 control signals, combined with ten bits of additional information (inst[15:11], PSR[15], ACV, BEN, INT, and R), as shown in Figure C.1. We have said that during each clock cycle, 42 of these control signals determine the processing of information in the data path and the other ten control signals combine with the ten bits of additional information to determine which set of control signals will be required in the next clock cycle.

We say that these 52 control signals specify the *state* of the control structure of the LC-3 microarchitecture. We can completely describe the behavior of the LC-3 microarchitecture by means of a directed graph that consists of nodes (one corresponding to each state) and arcs (showing the flow from each state to the one[s] it goes to next). We call such a graph a *state machine*.

Figure C.2, combined with Figure C.7, is the state machine for our implementation of the LC-3. The state machine describes what happens during each clock cycle in which the computer is running. Each state is active for exactly one clock cycle before control passes to the next state. The state machine shows the step-by-step (clock cycle-by-clock cycle) process that each instruction goes through from the start of its FETCH phase to the end of its instruction cycle, as described in Section 4.3.2. Each node in the state machine corresponds to the activity that the processor carries out during a single clock cycle. The actual processing that is performed in the data path is contained inside the node. The step-by-step flow is conveyed by the arcs that take the processor from one state to the next.

Let's start our study of Figure C.2 by examining the FETCH phase of the instruction cycle. As you know, every instruction goes through the same FETCH phase in its instruction cycle. Recall from Chapter 4 that the FETCH phase starts with a memory access to read the instruction at the address specified by the PC. Note that in the state numbered 18, the MAR is loaded with the address contained in PC, and the PC is incremented in preparation for the FETCH of the next LC-3 instruction after the current instruction finishes its instruction cycle. If the content of MAR specifies privileged memory, and PSR[15] = 1, indicating User mode, the access of the instruction will not be allowed. That would be an access control violation, so ACV is set. Finally, if there is no interrupt request present (INT = 0), the flow passes to the state numbered 33. We will describe in Section C.7 the flow of control if INT = 1, that is, if an external device is requesting an interrupt.

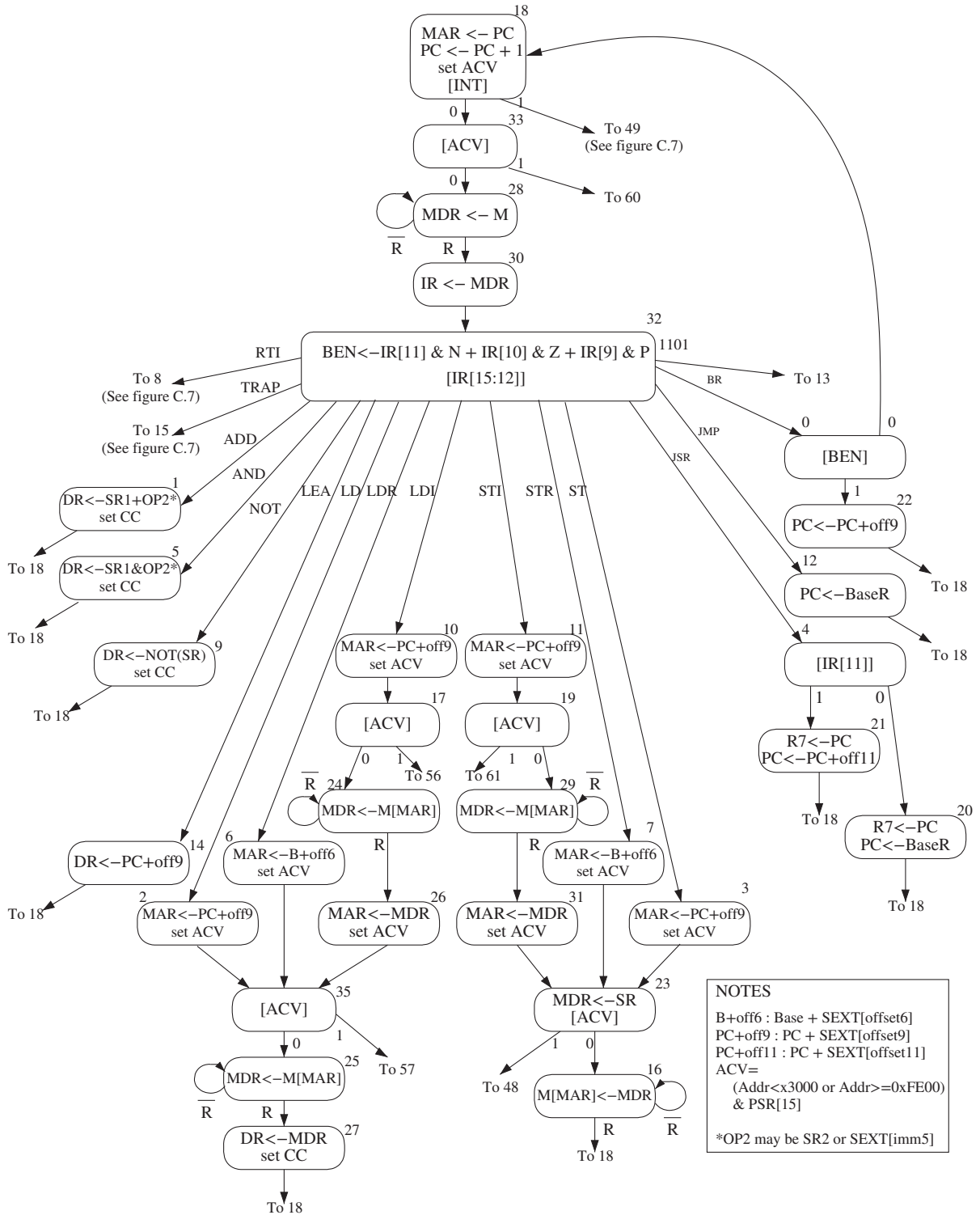


Figure C.2 A state machine for the LC-3.

Before we get into what happens during the clock cycle when the processor is in the state numbered 33, we should explain the numbering system—that is, why are states numbered 18 and 33. Recall, from our discussion of finite state machines in Chapter 3, that each state must be uniquely specified and that this unique specification is accomplished by means of state variables. Our state machine that implements the LC-3 ISA requires 59 distinct states to implement the entire behavior of the LC-3. Figure C.2 shows 31 of them plus pointers to seven others (states 8, 13, 15, 48, 49, 57, and 60). Figure C.7 shows the other 28 states (including the seven that are pointed to in Figure C.2). We will visit all of them as we go through this appendix. Since  $k$  logical variables can uniquely identify  $2^k$  items, six state variables are needed to uniquely specify 59 states. The number next to each node in Figure C.2 and Figure C.7 is the decimal equivalent of the values (0 or 1) of the six state variables for the corresponding state. Thus, for example, the state numbered 18 has state variable values 010010.

Now, back to what happens after the clock cycle in which the activity of state 18 has finished. As we said, if no external device is requesting an interrupt, the flow passes to state 33 (i.e., 100001). From state 33, control passes to state 60 if the processor is trying to access privileged memory while in User mode, or to state 28, if the memory access is allowed, that is, if there is no ACV violation. We will discuss what happens if there is an ACV violation in Section C.7.

In state 28, since the MAR contains the address of the instruction to be processed, this instruction is read from memory and loaded into the MDR. Since this memory access can take multiple cycles, this state continues to execute until a ready signal from the memory (R) is asserted, indicating that the memory access has completed. Thus, the MDR contains the valid contents of the memory location specified by MAR. The state machine then moves on to state 30, where the instruction is loaded into the instruction register (IR), completing the fetch phase of the instruction cycle.

The state machine then moves to state 32, where DECODE takes place. Note that there are 13 arcs emanating from state 32, each one corresponding to bits [15:12] of the LC-3 instruction. These are the opcode bits that direct the state machine to one of 16 paths to carry out the instruction cycle of the particular instruction that has just been fetched. Note that the arc from the last state of each instruction cycle (i.e., the state that completes the processing of that LC-3 instruction) takes us to state 18 (to begin the instruction cycle of the next LC-3 instruction).

## C.3 The Data Path

The data path consists of all components that actually process the information during each clock cycle—the functional units that operate on the information, the registers that store information at the end of one cycle so it will be available for further use in subsequent cycles, and the buses and wires that carry information from one point to another in the data path. Figure C.3, an expanded version of what you have already encountered in Figure 5.18, illustrates the data path of our microarchitecture of the LC-3.

Note the control signals that are associated with each component in the data path. For example, ALUK, consisting of two control signals, is associated with

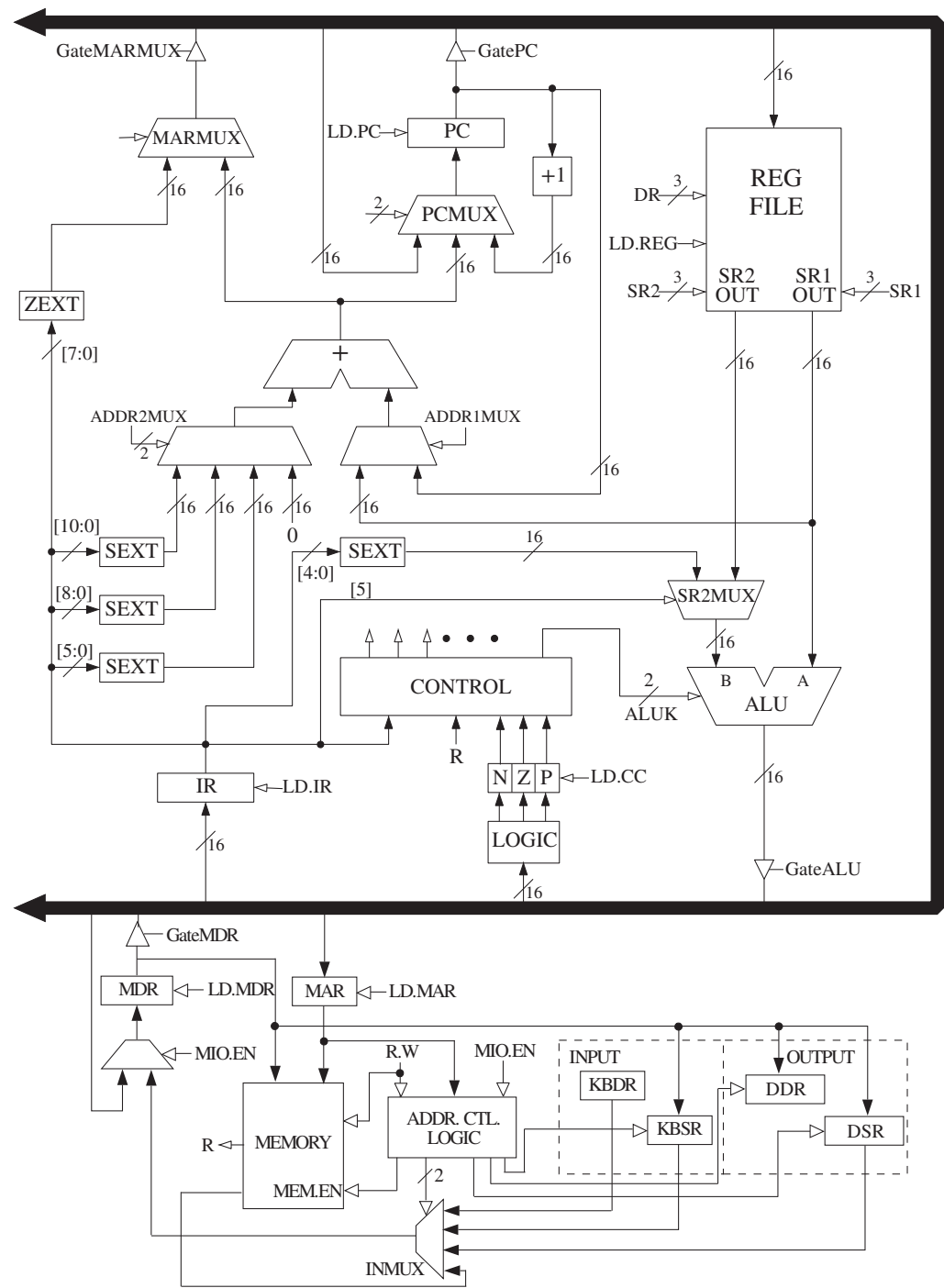


Figure C.3 The LC-3 data path.



**Table C.1 Data Path Control Signals**

Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
LD.Priv/1:	NO, LOAD	
LD.Priority/1:	NO, LOAD	
LD.SavedSSP/1:	NO, LOAD	
LD.SavedUSP/1:	NO, LOAD	
LD.ACV/1:	NO, LOAD	
LD.Vector/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateVector/1:	NO, YES	
GatePC-1/1:	NO, YES	
GatePSR/1:	NO, YES	
GateSP/1:	NO, YES	
PCMUX/2:	PC+1 BUS ADDER	;select pc+1 ;select value from bus ;select output of address adder
DRMUX/2:	11.9 R7 SP	;destination IR[11:9] ;destination R7 ;destination R6
SR1MUX/2:	11.9 8.6 SP	;source IR[11:9] ;source IR[8:6] ;source R6
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO offset6 PCoffset9 PCoffset11	;select the value zero ;select SEXT[IR[5:0]] ;select SEXT[IR[8:0]] ;select SEXT[IR[10:0]]
SPMUX/2:	SP+1 SP-1 Saved SSP Saved USP	;select stack pointer+1 ;select stack pointer-1 ;select saved Supervisor Stack Pointer ;select saved User Stack Pointer
MARMUX/1:	7.0 ADDER	;select ZEXT[IR[7:0]] ;select output of address adder
TableMUX/1:	x00, x01	
VectorMUX/2:	INTV Priv.exception Opc.exception ACV.exception	
PSRMUX/1:	individual settings, BUS	
ALUK/2:	ADD, AND, NOT, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
Set.Priv/1:	0 1	;Supervisor mode ;User mode

the ALU. These control signals determine how that component (the ALU) will be used each cycle. Table C.1 lists the set of 42 control signals that control the elements of the data path and the set of values that each control signal can have. (Actually, for readability, we provide a symbolic name for each value, rather than the binary value.) For example, since ALUK consists of two bits, it can have one of four values. Which value it has during any particular clock cycle depends on whether the ALU is required to ADD, AND, NOT, or simply pass one of its inputs to the output during that clock cycle. PCMUX also consists of two control signals and specifies which input to the MUX is required during a given clock cycle. LD.PC is a single-bit control signal and is a 0 (NO) or a 1 (YES), depending on whether or not the PC is to be loaded during the given clock cycle.

During each clock cycle, corresponding to the “current state” in the state machine, the 42 bits of control direct the processing of all components in the data path that are required during that clock cycle. As we have said, the processing that takes place in the data path during that clock cycle is specified inside the node representing the state.

## C.4 The Control Structure

The control structure of a microarchitecture is specified by its state machine. As described earlier, the state machine (Figure C.2 and Figure C.7) determines which control signals are needed each clock cycle to process information in the data path and which control signals are needed each clock cycle to direct the flow of control from the currently active state to its successor state.

Figure C.4 shows a block diagram of the control structure of our implementation of the LC-3. Many implementations are possible, and the design considerations that must be studied to determine which of many possible implementations should be used is the subject of a full course in computer architecture.

We have chosen here a straightforward microprogrammed implementation. Each state of the control structure requires 42 bits to control the processing in the data path and 10 bits to help determine which state comes next. These 52 bits are collectively known as a *microinstruction*. Each microinstruction (i.e., each state of the state machine) is stored in one 52-bit location of a special memory called the control store. There are 59 distinct states. Since each state corresponds to one microinstruction in the control store, the control store for our microprogrammed implementation requires six bits to specify the address of each microinstruction. Those six bits correspond to the state number associated with each state in the state machine. For example, the microinstruction associated with state 18 is the set of 52 control signals stored in address 18 of the control store.

Table C.2 lists the function of the ten bits of control information that help determine which state comes next. Figure C.5 shows the logic of the microsequencer. The purpose of the microsequencer is to determine the address in the control store that corresponds to the next state, that is, the location where the 52 bits of control information for the next state are stored.

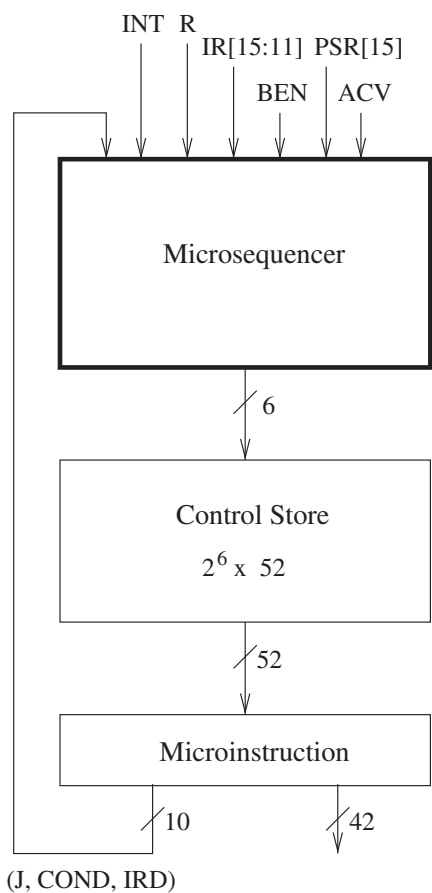
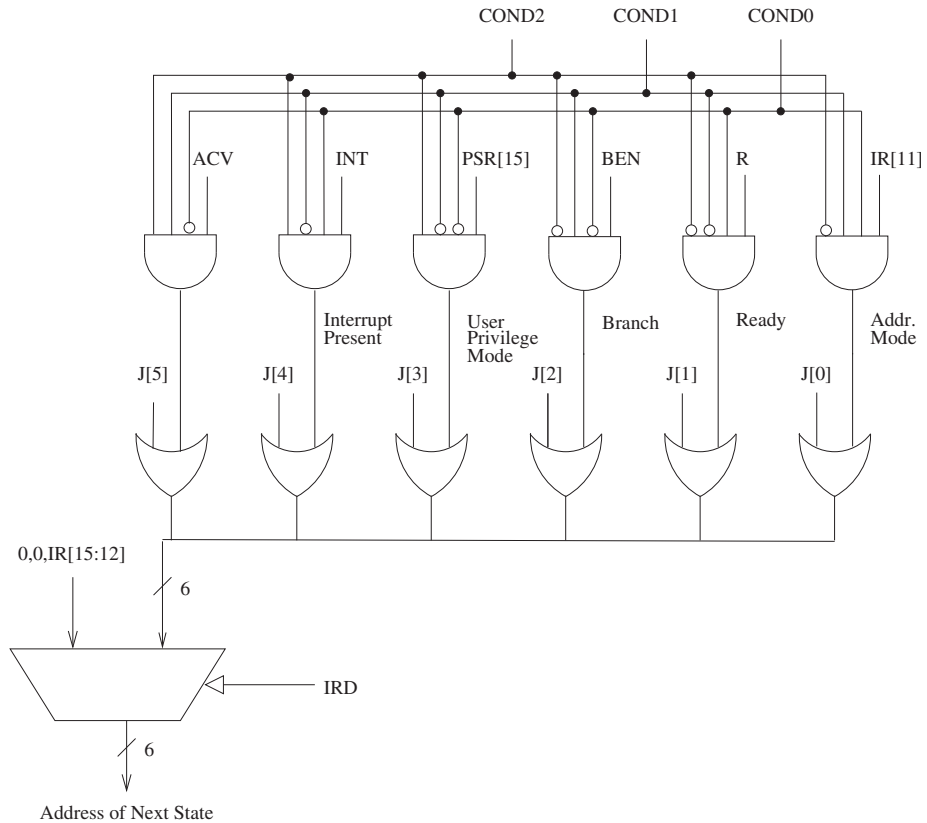


Figure C.4 The control structure of a microprogrammed implementation, overall block diagram.

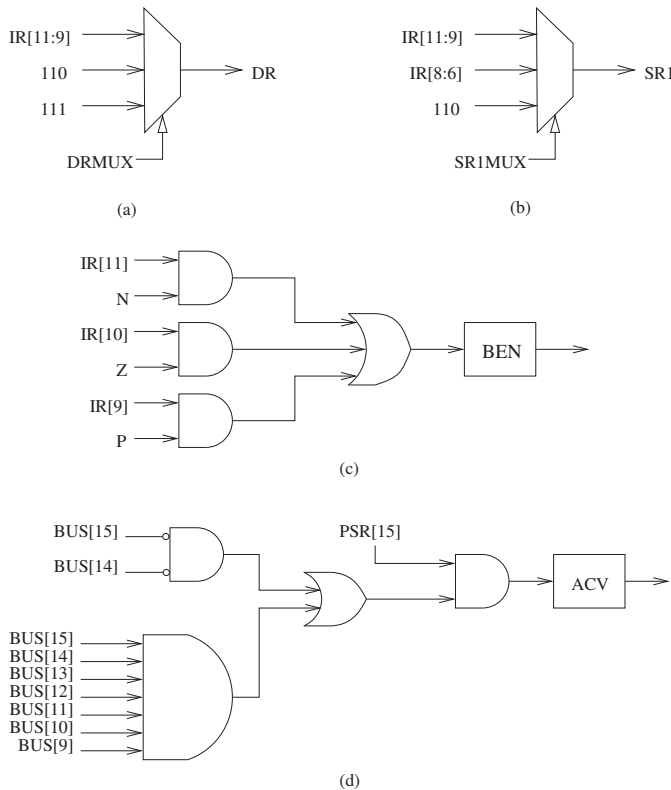
Table C.2 Microsequencer Control Signals	
Signal Name	Signal Values
J/6:	
COND/3:	COND0 ;Unconditional
	COND1 ;Memory Ready
	COND2 ;Branch
	COND3 ;Addressing Mode
	COND4 ;Privilege Mode
	COND5 ;Interrupt test
	COND6 ;ACV Test
IRD/1:	NO, YES



**Figure C.5** The microsequencer of the LC-3.

As we said, state 32 of the state machine (Figure C.2) performs the DECODE phase of the instruction cycle. It has 16 “next” states, depending on the LC-3 instruction being executed during the current instruction cycle. If the IRD control signal in the microinstruction corresponding to state 32 is 1, the output MUX of the microsequencer (Figure C.5) will take its source from the six bits formed by 00 concatenated with the four opcode bits IR[15:12]. Since IR[15:12] specifies the opcode of the current LC-3 instruction being processed, the next address of the control store will be one of 16 addresses, corresponding to the 15 opcodes plus the one unused opcode, IR[15:12] = 1101. That is, each of the 16 next states after state 32 is the first state to be carried out after the instruction has been decoded in state 32. For example, if the instruction being processed is ADD, the address of the next state is state 1, whose microinstruction is stored at location 000001. Recall that IR[15:12] for ADD is 0001.

If, somehow, the instruction inadvertently contained IR[15:12] = 1101, the unused opcode, the microarchitecture would execute a sequence of microinstructions, starting at state 13. These microinstructions would respond to the fact that an instruction with an illegal opcode had been fetched. Section C.7.3 describes what happens in that case.



**Figure C.6** Additional logic required to provide control signals.

Several signals necessary to control the data path and the microsequencer are not among those listed in Tables C.1 and C.2. They are DR, SR1, BEN, INT, ACV, and R. Figure C.6 shows the additional logic needed to generate DR, SR1, BEN, and ACV.

The INT signal is supplied by some event external to the normal instruction processing, indicating that normal instruction processing should be interrupted and this external event dealt with. The interrupt mechanism was described in Chapter 9. The corresponding flow of control within the microarchitecture is described in Section C.7.

The remaining signal, R, is a signal generated by the memory in order to allow the LC-3 to operate correctly with a memory that takes multiple clock cycles to read or store a value.

Suppose it takes memory five cycles to read a value. That is, once MAR contains the address to be read and the microinstruction asserts READ, it will take five cycles before the contents of the specified location in memory is available to be loaded into MDR. (Note that the microinstruction asserts READ by means of two control signals: MIO.EN/YES and R.W/RD; see Figure C.3.)

Recall our discussion in Section C.2 of the function of state 28, which accesses an instruction from memory during the FETCH phase of each instruction cycle. If the memory takes five cycles to read a value, for the LC-3 to operate correctly, state 28 must execute five times before moving on to state 30. That is, until MDR contains valid data from the memory location specified by the contents of MAR, we want state 28 to continue to re-execute. After five clock cycles, the memory has completed the “read,” resulting in valid data in MDR, so the processor can move on to state 30. What if the microarchitecture did not wait for the memory to complete the read operation before moving on to state 30? Since the contents of MDR would still be garbage, the microarchitecture would put garbage into the IR in state 30.

The ready signal (R) enables the memory read to execute correctly. Since the memory knows it needs five clock cycles to complete the read, it asserts a ready signal (R) throughout the fifth clock cycle. Figure C.2 shows that the next state is 28 (i.e., 011100) if the memory read will not complete in the current clock cycle and state 30 (i.e., 011110) if it will. As we have seen, it is the job of the microsequencer (Figure C.5) to produce the next state address.

The ten microsequencer control signals for state 28 are:

```
IRD/0      ; NO
COND/001   ; Memory Ready
J/011100
```

With these control signals, what next state address is generated by the microsequencer? For each of the first four executions of state 28, since  $R = 0$ , the next state address is 011100. This causes state 28 to be executed again in the next clock cycle. In the fifth clock cycle, since  $R = 1$ , the next state address is 011110, and the LC-3 moves on to state 30. Note that in order for the ready signal (R) from memory to be part of the next state address, COND had to be set to 001, which allowed R to pass through its four-input AND gate.

## C.5 The TRAP Instruction

As we have said, each LC-3 instruction follows its own path from state 32 to its final state in its instruction cycle, after which it returns to state 18 to start processing the next instruction. As an example, we will follow the instruction cycle of the TRAP instruction, shown in Figure C.7.

Recall that the TRAP instruction pushes the PSR and PC onto the system stack, loads the PC with the starting address of the trap service routine, and executes the service routine from privileged memory.

From state 32, the next state after DECODE is state 15, consistent with the TRAP instruction opcode 1111. In state 15, the Table register, which will be used to form MAR[15:8] of the trap vector table entry, is loaded with x00, the PC is incremented (we will see why momentarily), and the MDR is loaded with the PSR in preparation for pushing it onto the system stack. Control passes to state 47.

In state 47, the trap vector (IR[7:0]) is loaded into the eight-bit register Vector, PSR[15] is set to Supervisor mode since the trap service routine executes in privileged memory, and the state machine branches to state 37 or 45, depending on whether the program that executed the TRAP instruction was in User mode or Supervisor mode. If in User mode, state 45 saves the User Stack Pointer in Saved\_USP, loads the stack pointer from Saved\_SSP, and continues on to state 37, where the processor starts pushing PSR and PC onto the stack. If the program executing the TRAP instruction is already in Privileged mode, state 45 is not necessary.

In states 37 and 41, the PSR is pushed onto the system stack. In states 43, 46, and 52, the PC is pushed onto the system stack. Note that in state 43, the PC is decremented before being pushed onto the stack. This is necessary in the case of dealing with interrupts and exceptions, which will be explained in Section C.7. This is not necessary for processing the TRAP instruction, which is why PC is incremented in state 15.

The only thing remaining is to load PC with the starting address of the trap service routine. This is done by loading MAR with the address of the proper entry in the trap vector table, obtained by concatenating Table and Vector (in state 54), loading the starting address from memory into MDR (in state 53), and loading the PC (in state 55). This completes the execution of the TRAP instruction, and control returns to state 18 to begin processing the next instruction – in this case, the first instruction of the trap service routine.

The last instruction in every trap service routine is RTI (return from trap or interrupt). From DECODE in state 32, the next state of RTI is state 8, consistent with its eight-bit opcode 1000. In states 8, 36, and 38, the PC is popped off the system stack and loaded into PC. In states 39, 40, 42, and 34, the PSR is popped off the system stack and loaded into PSR. This returns the PC and PSR to the values it had before the trap service routine was executed. Finally, if the program that invoked the TRAP instruction was in User mode, PSR[15] must be returned to 1, the Supervisor Stack Pointer saved, and the User Stack Pointer loaded into SP. This is done in state 59, completing the instruction cycle for RTI.

## C.6 Memory-Mapped I/O

As you know from Chapter 9, the LC-3 ISA performs input and output via memory-mapped I/O, that is, with the same data movement instructions that it uses to read from and write to memory. The LC-3 does this by assigning an address to each device register. Input is accomplished by a load instruction whose effective address is the address of an input device register. Output is accomplished by a store instruction whose effective address is the address of an output device register. For example, in state 25 of Figure C.2, if the address in MAR is xFE02, MDR is supplied by the KBDR, and the data input will be the last keyboard character typed. On the other hand, if the address in MAR is a legitimate memory address, MDR is supplied by the memory.

**Table C.3 Truth Table for Address Control Logic**

MAR	MIO.EN	R.W	MEM.EN	IN.MUX	LD.KBSR	LD.DSR	LD.DDR
xFE00	0	R	0	x	0	0	0
xFE00	0	W	0	x	0	0	0
xFE00	1	R	0	KBSR	0	0	0
xFE00	1	W	0	x	1	0	0
xFE02	0	R	0	x	0	0	0
xFE02	0	W	0	x	0	0	0
xFE02	1	R	0	KBDR	0	0	0
xFE02	1	W	0	x	0	0	0
xFE04	0	R	0	x	0	0	0
xFE04	0	W	0	x	0	0	0
xFE04	1	R	0	DSR	0	0	0
xFE04	1	W	0	x	0	1	0
xFE06	0	R	0	x	0	0	0
xFE06	0	W	0	x	0	0	0
xFE06	1	R	0	x	0	0	0
xFE06	1	W	0	x	0	0	1
other	0	R	0	x	0	0	0
other	0	W	0	x	0	0	0
other	1	R	1	mem	0	0	0
other	1	W	1	x	0	0	0

The state machine of Figure C.2 does not have to be altered to accommodate memory-mapped I/O. However, something has to determine when memory should be accessed and when I/O device registers should be accessed. This is the job of the address control logic (ADDR.CTL.LOGIC) shown in Figure C.3.

Table C.3 is a truth table for the address control logic, showing what control signals are generated, based on (1) the contents of MAR, (2) whether or not memory or I/O is accessed this cycle (MIO.EN/NO, YES), and (3) whether a load (R.W/Read) or store (R.W/Write) is requested. Note that, for a memory-mapped load, data can be supplied to MDR from one of four sources: memory, KBDR, KBSR, or DSR. The address control logic provides the appropriate select signals to the INMUX. For a memory-mapped store, the data supplied by MDR can be written to memory, KBSR, DDR, or DSR. The address control logic supplies the appropriate enable signal to the corresponding structure.

## C.7 Interrupt and Exception Control

The final piece of the state machine needed to complete the LC-3 story are those states that control the initiation of an interrupt, those states that control the return from an interrupt (the RTI instruction), and those states that control the initiation of one of the three exceptions specified by the ISA.

Interrupts and exceptions are very similar. Both stop the program that is currently executing. Both push the PSR and PC of the interrupted program onto the system stack, obtain the starting address of the interrupt or exception service routine from the interrupt vector table, and load that starting address into the Program Counter. The main difference between interrupts and exceptions is the nature of



the event that causes the program that is executing to stop. Interrupts are events that usually have nothing to do with the program that is executing. Exceptions are events that are the direct result of something going awry in the program that is executing. The LC-3 specifies three exceptions: a privilege mode violation, an illegal opcode, and an ACV exception. Figure C.7 shows the state machine that carries these out. Figure C.8 shows the data path, after adding the additional structures to Figure C.3 that are needed to make interrupt and exception processing work.

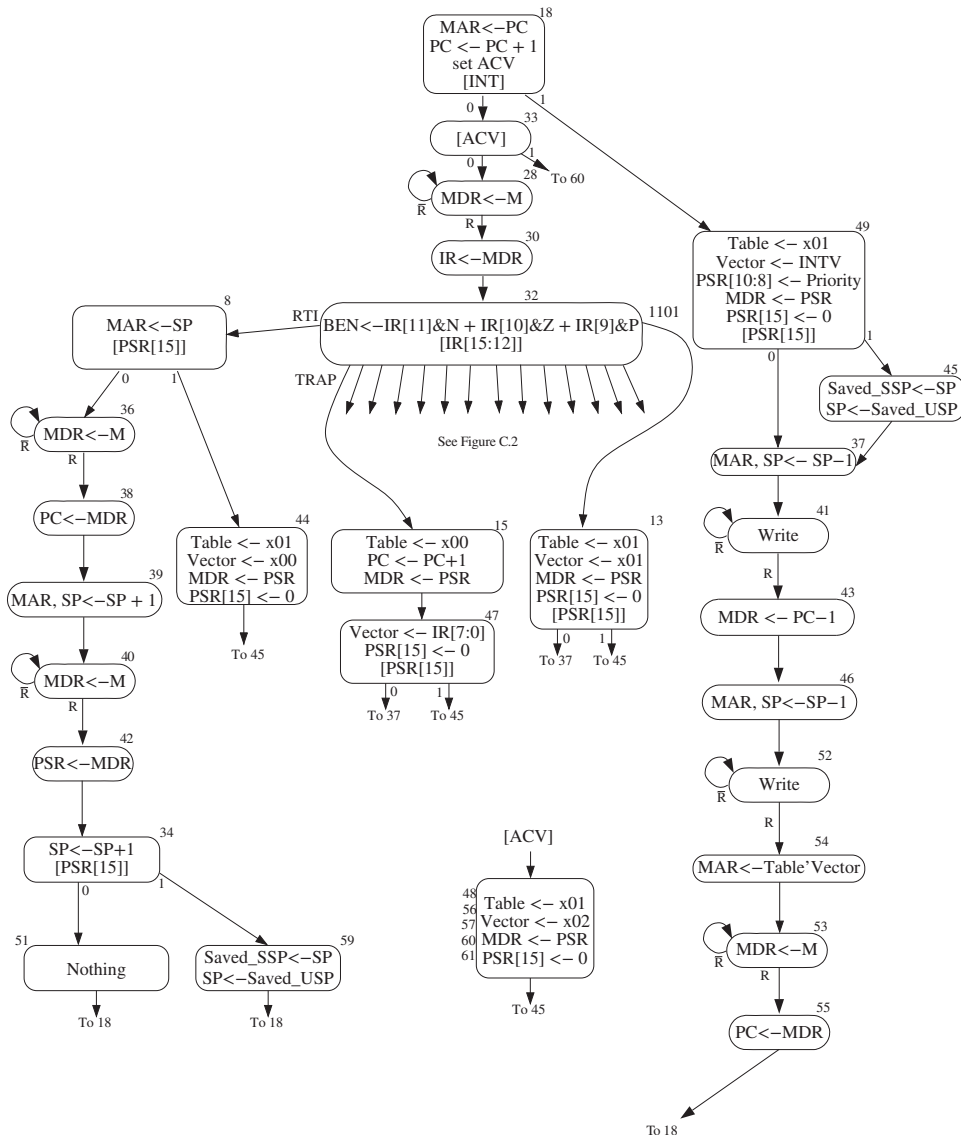


Figure C.7 LC-3 state machine showing interrupt control.

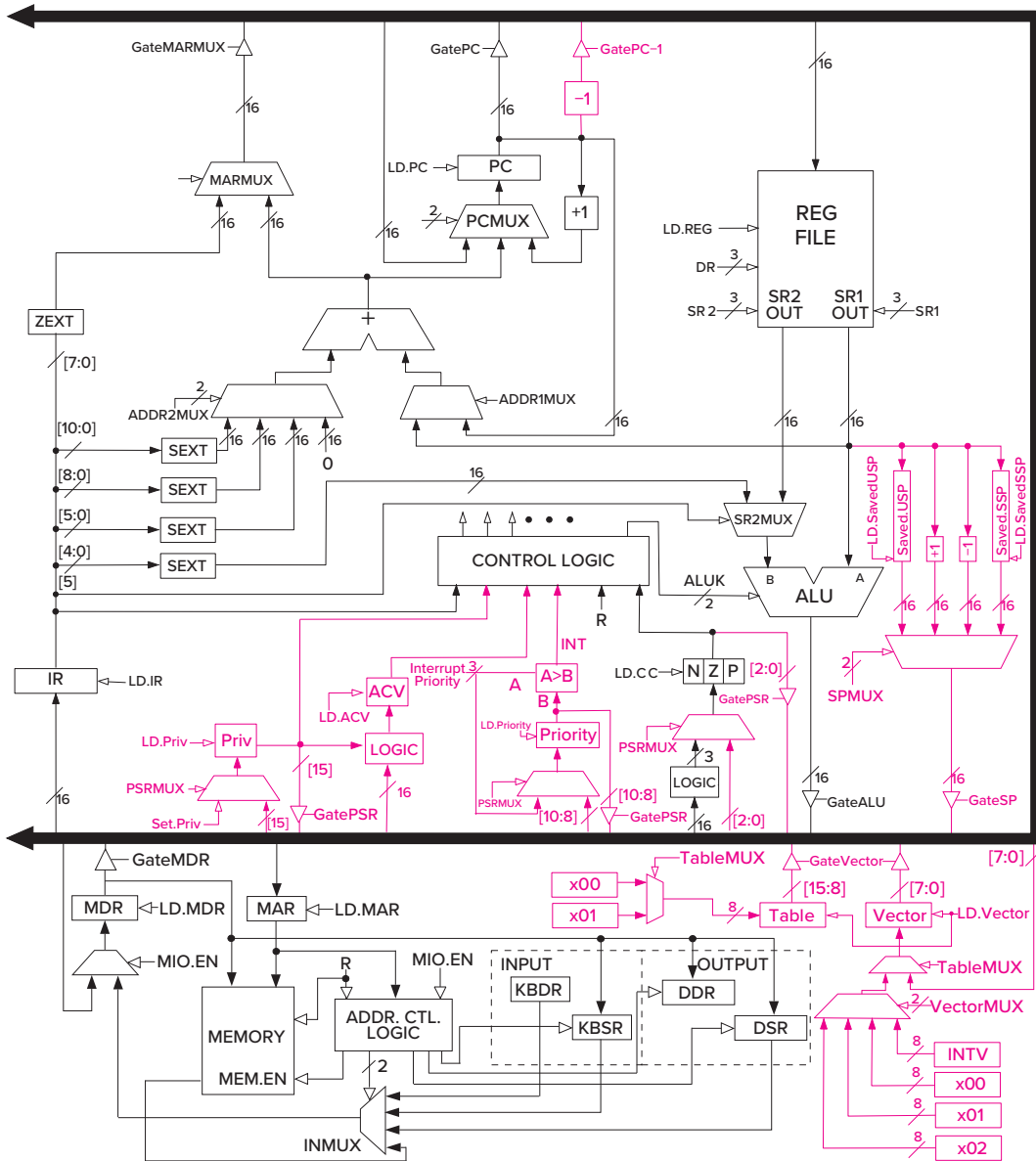


Figure C.8 LC-3 data path, including additional structures for interrupt control.

Section C.7.1 describes the flow of processing required to initiate an interrupt. Section C.7.3 describes the flow of processing required to initiate an exception.

### C.7.1 Initiating an Interrupt

While a program is executing, an interrupt can be requested by some external event so that the normal processing of instructions can be preempted and the control can turn its attention to processing the interrupt. The external event requests

an interrupt by asserting its interrupt request signal. Recall from Chapter 9 that if the priority level of the device asserting its interrupt request signal is higher than both the priority level of the currently executing program and any other external interrupt request asserted at the same time, INT is asserted and INTV is loaded with the interrupt vector corresponding to that external event. The microprocessor responds to INT by initiating the interrupt. That is, the processor puts itself into Supervisor mode if it isn't in Supervisor mode, pushes the PSR and PC of the interrupted process onto the supervisor stack, and loads the PC with the starting address of the interrupt service routine. The PSR contains the privilege mode PSR[15], priority level PSR[10:8], and condition codes PSR[2:0] of a program. It is important that when the processor resumes execution of the interrupted program, the privilege mode, priority level, and condition codes are restored to what they were when the interrupt occurred.

The microarchitecture of the LC-3 initiates an interrupt as follows: Recall from Figure C.2 that in state 18, while MAR is loaded with the contents of PC and PC is incremented, INT is tested.

State 18 is the only state in which the processor checks for interrupts. The reason for only testing in state 18 is straightforward: Once an LC-3 instruction starts processing, it is easier to let it finish its complete instruction cycle (FETCH, DECODE, etc.) than to interrupt it in the middle and have to keep track of how far along it was when the external device requested an interrupt (i.e., asserted INT). If INT is only tested in state 18, the current instruction cycle can be aborted early (even before the instruction has been fetched), and control directed to initiating the interrupt.

The test is enabled by the control signals that make up COND5, which are 101 only in state 18, allowing the value of INT to pass through its four-input AND gate, shown in Figure C.5, to contribute to the address of the next state. Since the COND signals are not 101 in any other state, INT has no effect in any other state.

In state 18, the ten microsequencer control bits are as follows:

```

IRD/0      ; NO
COND/101   ; Test for interrupts
J/100001

```

If  $INT = 1$ , a 1 is produced at the output of the AND gate, which in turn makes the next state address not 100001, corresponding to state 33, but rather 110001, corresponding to state 49. This starts the initiation of the interrupt (see Figure C.7).

Several functions are performed in state 49. The PSR, which contains the privilege mode, priority level, and condition codes of the interrupted program, are loaded into MDR, in preparation for pushing it onto the supervisor stack. PSR[15] is cleared, reflecting the change to Supervisor mode, since all interrupt service routines execute in Supervisor mode. The three-bit priority level and eight-bit interrupt vector (INTV) provided by the interrupting device are recorded. PSR[10:8] is loaded with the priority level of the interrupting device. The internal register Vector is loaded with INTV and the eight-bit register Table is loaded with x01 in preparation for accessing the interrupt vector table to obtain the starting address of the interrupt service routine. Finally, the processor tests

the old PSR[15] to determine whether the stack pointers must be adjusted before pushing PSR and PC.

If the old PSR[15] = 0, the processor is already operating in Supervisor mode. R6 is the Supervisor Stack Pointer (SSP), so the processor proceeds immediately to states 37 and 41 to push the PSR of the interrupted program onto the supervisor stack. If PSR[15] = 1, the interrupted program was in User mode. In that case, the User Stack Pointer (USP) must be saved in Saved\_USP and R6 must be loaded with the contents of Saved\_SSP before moving to state 37. This is done in state 45.

The control flow from state 49 to either 37 or 45 is enabled by the ten microsequencer control bits, as follows:

```
IRD/0      ; NO
COND/100   ; Test PSR[15], privilege mode
J/100101
```

If PSR[15] = 0, control goes to state 37 (100101); if PSR[15] = 1, control goes to state 45 (101101).

In state 37, R6 (the SSP) is decremented (preparing for the push), and MAR is loaded with the address of the new top of the stack.

In state 41, the memory is enabled to WRITE (MIO.EN/YES, R.W/WR). When the write completes, signaled by R = 1, PSR has been pushed onto the supervisor stack, and the flow moves on to state 43.

In state 43, the PC is loaded into MDR. Note that state 43 says MDR is loaded with PC-1. Recall that in state 18, at the beginning of the instruction cycle for the interrupted instruction, PC was incremented. Loading MDR with PC-1 adjusts PC to the correct address of the interrupted program.

In states 46 and 52, the same sequence as in states 37 and 41 occurs, only this time the PC of the interrupted program is pushed onto the supervisor stack.

The final task to complete the initiation of the interrupt is to load the PC with the starting address of the interrupt service routine. This is carried out by states 54, 53, and 55. It is accomplished in a manner similar to the loading of the PC with the starting address of a TRAP service routine. The event causing the INT request supplies the eight-bit interrupt vector INTV associated with the interrupt, similar to the eight-bit trap vector contained in the TRAP instruction. This interrupt vector is stored in the eight-bit register INTV, shown on the data path in Figure C.8.

The interrupt vector table occupies memory locations x0100 to x01FF. In state 54, the interrupt vector that was loaded into Vector in state 49 is combined with the base address of the interrupt vector table (x0100) and loaded into MAR. In state 53, memory is READ. When R = 1, the read has completed, and MDR contains the starting address of the interrupt service routine. In state 55, the PC is loaded with that starting address, completing the initiation of the interrupt.

It is important to emphasize that the LC-3 supports two stacks, one for each privilege mode, and two stack pointers (USP and SSP), one for each stack. R6 is the stack pointer and is loaded from the Saved\_SSP when privilege changes from User mode to Supervisor mode, and from Saved\_USP when privilege changes

from Supervisor mode to User mode. When the privilege mode changes, the current value in R6 must be stored in the appropriate “Saved” stack pointer in order to be available the next time the privilege mode changes back.

### C.7.2 Returning from an Interrupt or Trap Service Routine, RTI

Interrupt service routines, like trap service routines already described, end with the execution of the RTI instruction. The job of the RTI instruction is to restore the computer to the state it was in before the interrupt or trap service routine was executed. This means restoring the PSR (i.e., the privilege mode, priority level, and the values of the condition codes N, Z, P) and restoring the PC. These values were pushed onto the stack during the initiation of the interrupt or execution of the TRAP instruction. They must, therefore, be popped off the stack in the reverse order.

The first state after DECODE is state 8. Here we load the MAR with the address of the top of the supervisor stack, which contains the last thing pushed (that has not been subsequently popped)—the state of the PC when the interrupt was initiated. At the same time, we test PSR[15] since RTI is a privileged instruction and can only execute in Supervisor mode. If PSR[15] = 0, we can continue to carry out the requirements of RTI.

States 36 and 38 restore PC to the value it had when the interrupt was initiated. In state 36, the memory is read. When the read is completed, MDR contains the address of the instruction that was to be processed next when the interrupt occurred. State 38 loads that address into the PC.

States 39, 40, 42, and 34 restore the privilege mode, priority level, and condition codes (N, Z, P) to their original values. In state 39, the Supervisor Stack Pointer is incremented so that it points to the top of the stack after the PC was popped. The MAR is loaded with the address of the new top of the stack. State 40 initiates the memory READ; when the READ is completed, MDR contains the interrupted PSR. State 42 loads the PSR from MDR, and state 34 increments the stack pointer.

The only thing left is to check the privilege mode of the interrupted program to see whether the stack pointers have to be switched. In state 34, the microsequencer control bits are as follows:

```

IRD/0      ; NO
COND/100   ; Test PSR[15], privilege mode
J/110011

```

If PSR[15] = 0, control flows to state 51 (110011) to do nothing for one cycle. If PSR[15] = 1, control flows to state 59, where R6 is saved in Saved\_SSP and R6 is loaded from Saved\_USP. In both cases, control returns to state 18 to begin processing the next instruction.

### C.7.3 Initiating an Exception

The LC-3 identifies three cases where processing is not allowed to continue normally due to something going awry in the executing program. We refer to these cases as exceptions. They are initiated in the same way interrupts are initiated,

by pushing the PSR and PC onto the system stack, obtaining the starting address of the exception service routine from the interrupt vector table, and loading that address into the PC to initiate the exception service routine.

The three exceptions identified in the LC-3 are (1) a privileged mode exception caused by the program attempting to execute the RTI instruction while in User mode, (2) the illegal opcode exception caused by the program trying to execute an instruction whose opcode is 1101, and (3) an access control violation (ACV) exception caused by the program trying to access a privileged memory location while in User mode.

### C.7.3.1 Privilege Mode Exception

If the processor is in User mode ( $\text{PSR}[15] = 1$ ) and is attempting to execute RTI, a privilege mode exception occurs. The processor pushes the PSR and the address of the RTI instruction onto the supervisor stack and loads the PC with the starting address of the service routine that handles privilege mode violations. Figure C.7 shows the flow, starting with a branch from state 8 to state 44 if  $\text{PSR}[15] = 1$ .

In state 44, the eight-bit Table register is loaded with x01, indicating the address of an entry in the interrupt vector table, and the eight-bit Vector register is loaded with x00, indicating the first entry in the interrupt vector table. The contents of x0100 is the starting address of the service routine that handles privilege mode exceptions. The MDR is loaded with the PSR of the program that caused the exception in preparation for pushing it onto the system stack. Finally,  $\text{PSR}[15]$  is set to 0, since the service routine will execute with supervisor privileges. Then the processor moves to state 45, where it follows the same flow as the initiation of interrupts.

The main difference between this flow and that for the initiation of interrupts is in state 54, where MAR is loaded with  $\text{x01}'\text{Vector}$ . In the case of interrupts, Vector is loaded in state 49 with INTV, which is supplied by the interrupting device. In the case of the privilege mode violation, Vector is loaded in state 44 with x00.

There are two additional functions performed in state 49 that are not performed in state 44. First, the priority level is changed, based on the priority of the interrupting device. We do not change the priority in handling a privilege mode violation. The service routine executes at the same priority as the program that caused the violation. Second, a test to determine the privilege mode is performed for an interrupt. This is unnecessary for a privilege mode violation since the processor already knows it is executing in User mode.

### C.7.3.2 Illegal Opcode Exception

Although it would be a rare situation, it is possible, we suppose, that a programmer writing a program in machine language could mistakenly include an instruction having opcode = 1101. Since there is no such opcode in the LC-3 ISA, the computer cannot process that instruction. State 32 performs the DECODE, and the next state is state 13.

The action the processor takes is very similar to that of a privilege mode exception. The PSR and PC of the program are pushed onto the supervisor stack, and the PC is loaded with the starting address of the Illegal Opcode exception service routine.

State 13 is very similar to state 44, which starts the initiation of a privilege mode exception. There are two differences: (1) Vector is loaded with x01, since the starting address of the service routine for the illegal opcode exception is in x0101. (2) In the case of the privilege mode exception, we know the program is in User mode when the processor attempts to execute the RTI instruction. In the case of an illegal opcode, the processor can be in either mode, so from state 13 the processor goes to state 37 or state 45, depending on whether the program is executing in Supervisor mode or User mode when the illegal opcode instruction is encountered.

Like state 44, the priority of the running program is not changed, since the urgency of handling the exception is the same as the urgency of executing the program that contains it. Like state 49, state 13 tests the privilege mode of the program that contains the illegal opcode, since if the currently executing program is in User mode, the stack pointers need to be switched as described in Section C.7.1. Like state 49, the processor then microbranches either to state 37 if the stack pointer is already pointing to the supervisor stack, or to state 45 if the stack pointers have to be switched. From there, the initiating sequence continues in states 37, 41, 43, etc., identical to what happens when an interrupt is initiated (Section C.7.1) or a privilege mode exception is initiated (Section C.7.3.1). The PSR and PC are pushed onto the supervisor stack and the starting address of the service routine is loaded into the PC, completing the initiation of the exception.

### C.7.3.3 Access Control Violation (ACV) Exception

An Access Control Violation (ACV) exception occurs if the processor attempts to access privileged memory while operating in User mode. The state machine checks for this in every case where the processor accesses memory, that is, in states 17, 19, 23, 33, and 35. If an ACV violation occurs, the next state is respectively states 56, 61, 48, 60, or 57 (see Figure C.2). In all five states, the processor loads Table with x01, Vector with x02, MDR with the PSR, sets PSR[15] to 0, exactly like state 44, with one exception. Vector is set to x02 since the starting address of the ACV exception service routine is in memory location x0102. Processing continues exactly like in state 44, moving first to state 45 to switch to the system stack, and then pushing PSR and PC onto the stack and loading the PC with the starting address of the service routine.

## C.8 Control Store

Figure C.9 completes our microprogrammed implementation of the LC-3. It shows the contents of each location of the control store, corresponding to the 52 control signals required by each state of the state machine. We have left the exact entries blank to allow you, the reader, the joy of filling in the required signals yourself. The solution is available from your instructor.

**Figure C.9** Specification of the control store.