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.

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

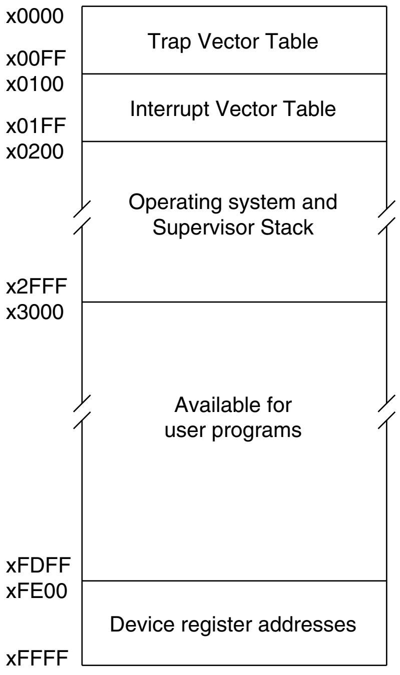

x0000
Trap Vector Table
x00FF
x0100
Interrupt Vector Table
x01FF
x0200
Operating system and Supervisor Stack
x2FFF
x3000
Available for user programs
xFDFF
xFE00
Device register addresses
xFFFF

Figure A.1 Memory map of the LC-3

needed to execute the instruction. The specific operation of each LC-3 instruction is described in Section A.3.

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.4 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.

Condition codes Three 1-bit registers: N (negative), Z (zero), and P (positive). Load instructions (LD, LDI, LDR, and LEA) 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.

Memory-mapped I/O Input and output are handled by load/store (LDI/STI, LDR/STR) instructions using memory addresses to designate each I/O device register. Addresses xFE00 through xFFFF have been allocated to represent the addresses of I/O devices. See Figure A.1. Also, Table A.3 lists each of the relevant device registers that have been identified for the LC-3 thus far, along with their corresponding assigned addresses from the memory address space.

Interrupt processing I/O devices have the capability of interrupting the processor. Section A.4 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.

Privilege mode The LC-3 specifies two levels of privilege, Supervisor mode (privileged) and User mode (unprivileged). Interrupt service routines 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 an RTI instruction while in User mode, a privilege mode exception occurs. Section A.4 explains what happens.

Supervisor Stack A region of memory in supervisor space accessible via the Supervisor Stack Pointer (SSP). When PSR[15] = 0, the stack pointer (R6) is 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.

A.2 Notation

The notation in Table A.1 will be helpful in understanding the descriptions of the LC-3 instructions (Section A.3).

A.3 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 the work of 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 4-bit opcode is not specified, but is reserved for future use. In the following pages, the instructions will be described in greater detail. 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.

Table A.1 Notational Conventions

Notation	Meaning
xNumber	The number in hexadecimal notation.
#Number	The number in decimal notation.
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, the notation is usually abbreviated PC[2].
BaseR	Base Register; one of R0..R7, used in conjunction with a six-bit offset to compute Base+offset addresses.
DR	Destination Register; one of R0..R7, which specifies which register the result of an instruction should be written to.
imm5	A 5-bit immediate value; bits [4:0] of an instruction when used as a literal (immediate) value. Taken as a 5-bit, 2's complement integer, it is sign-extended to 16 bits before it is used. Range: −16..15.
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 6-bit value; bits [5:0] of an instruction; used with the Base+offset addressing mode. Bits [5:0] are taken as a 6-bit signed 2's complement integer, 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, during execution of the instruction at address A, the PC contains address A + 1, indicating the next instruction is contained in A + 1.
PCoffset9	A 9-bit value; bits [8:0] of an instruction; used with the PC+offset addressing mode. Bits [8:0] are taken as a 9-bit signed 2's complement integer, sign-extended to 16 bits and then added to the incremented PC to form an address. Range −256..255.
PCoffset11	An 11-bit value; bits [10:0] of an instruction; used with the JSR opcode to compute the target address of a subroutine call. Bits [10:0] are taken as an 11-bit 2's complement integer, sign-extended to 16 bits and then added to the incremented PC to form the target address. Range −1024..1023.
PSR	Processor Status Register; 16-bit register that contains status information of the process that is running. PSR[15] = privilege mode. PSR[2:0] contains the condition codes. PSR[2] = N, PSR[1] = Z, PSR[0] = P.
setcc()	Indicates that condition codes N, Z, and P are set based on the value of the result written to DR. If the value is negative, N = 1, Z = 0, P = 0. If the value is zero, N = 0, Z = 1, P = 0. If the value is positive, N = 0, Z = 0, P = 1.
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 which specifies the register from which a source operand is obtained.
SSP	The Supervisor Stack Pointer.
trapvect8	An 8-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.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD^{+}	0001				DR			SR1			0	00		SR2		
ADD^{+}	0001				DR			SR1			1	imm5				
AND^{+}	0101				DR			SR1			0	00		SR2		
AND^{+}	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^{+}	0010				DR			PCoffset9								
LDI^{+}	1010				DR			PCoffset9								
LDR^{+}	0110				DR			BaseR			offset6					
LEA^{+}	1110				DR			PCoffset9								
NOT^{+}	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															

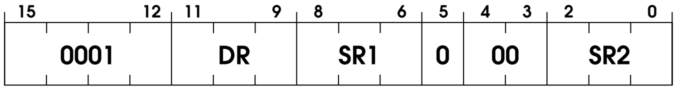
Figure A.2 Format of the entire LC-3 instruction set. **Note:** + indicates instructions that modify condition codes

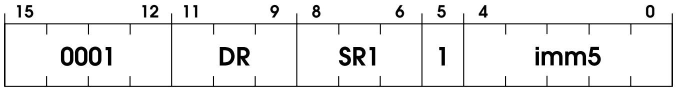
ADD Addition

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
```

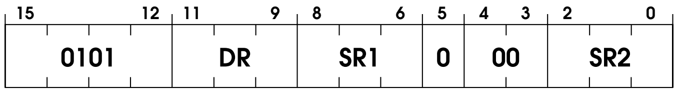
AND

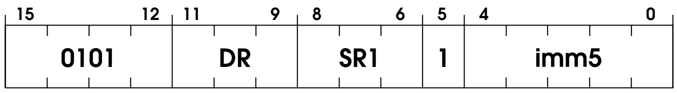
Bit-wise Logical AND

Assembler Formats

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

Encodings





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

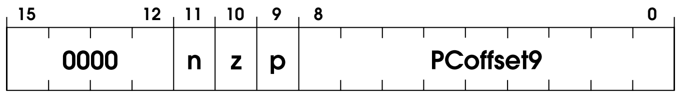
AND	R2, R3, R4	;R2 $\leftarrow$ R3 AND R4
AND	R2, R3, #7	;R2 $\leftarrow$ R3 AND 7

BR Conditional Branch

Assembler Formats

BRn	LABEL	BRzp	LABEL
BRz	LABEL	BRnp	LABEL
BRp	LABEL	BRnz	LABEL
BR[†]	LABEL	BRnzp	LABEL

Encoding



Operation

```
if ((n AND N) OR (z AND Z) OR (p AND P))
   PC = PC‡ + SEXT(PCoffset9);
```

Description

The condition codes specified by the state of bits [11:9] are tested. If bit [11] is set, N is tested; if bit [11] is clear, N is not tested. If bit [10] is set, Z is tested, etc. If any of the condition codes tested is set, the program branches to the 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[†]	NEXT	; Unconditionally branch to NEXT.

[†] The assembly language opcode BR is interpreted the same as BRnzp; that is, always branch to the target address.

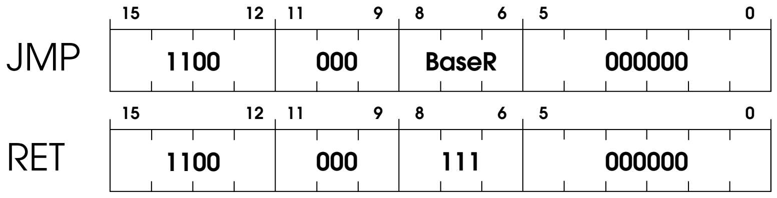
[‡] This is the incremented PC.

JMP / RET — Jump / Return from Subroutine

Assembler Formats

JMP BaseR
RET

Encoding



Operation

```
PC = 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 $\leftarrow$ R2
RET		; PC $\leftarrow$ R7

Note

The RET instruction is a special case of the JMP instruction. 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

	15 12	11	10 0
JSR	0100	1	PCoffset11

	15 12	11	10 9	8 6	5 0
JSRR	0100	0	00	BaseR	000000

Operation

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

Description

First, the incremented PC is saved in R7. This is the linkage back to the calling routine. Then the PC is loaded with the address of the first instruction of the subroutine, causing an unconditional jump to that address. 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).

Examples

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

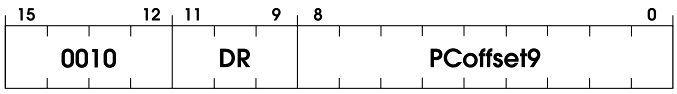
† This is the incremented PC.

LD Load

Assembler Format

LD DR, LABEL

Encoding



Operation

```
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. The contents of memory at this address are loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

LD R4, VALUE ; R4 $\leftarrow$ mem[VALUE]

† This is the incremented PC.

LDI Load Indirect

Assembler Format

LDI DR, LABEL

Encoding

15 12	11 9	8 0
1010	DR	PCoffset9

Operation

```
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. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

LDI R4, ONEMORE ; R4 $\leftarrow$ mem[mem[ONEMORE]]

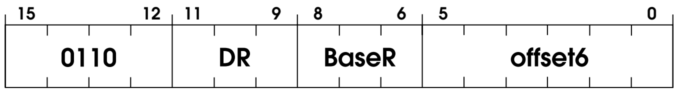
† This is the incremented PC.

LDR Load Base+offset

Assembler Format

LDR DR, BaseR, offset6

Encoding



Operation

```
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]. The contents of memory at this address are loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

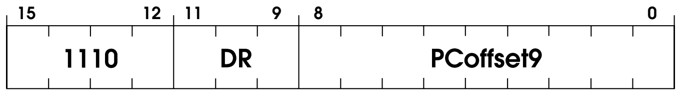
LDR R4, R2, #−5 ; R4 $\leftarrow$ mem[R2 − 5]

LEA — Load Effective Address

Assembler Format

LEA DR, LABEL

Encoding



Operation

```
DR = 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. This address is loaded into DR.[‡] The condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

LEA R4, TARGET ; R4 ← address of TARGET.

[†] This is the incremented PC.

[‡] The LEA instruction does not read memory to obtain the information to load into DR. 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	11111

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 $\leftarrow$ NOT(R2)

RET[†] Return from Subroutine

Assembler Format

RET

Encoding

15 12	11 9	8 6	5 0
1100	000	111	000000

Operation

```
PC = R7;
```

Description

The PC is loaded with the value in R7. This causes a return from a previous JSR instruction.

Example

RET ; PC ← R7

[†] The RET instruction is a specific encoding of the JMP instruction. See also JMP.

RTI Return from Interrupt

Assembler Format

RTI

Encoding

15 12	11 0
1000	000000000000

Operation

```
if (PSR[15] == 0)
   PC = mem[R6]; R6 is the SSP
   R6 = R6+1;
   TEMP = mem[R6];
   R6 = R6+1;
   PSR = TEMP; the privilege mode and condition codes of
   the interrupted process are restored
else
   Initiate a privilege mode exception;
```

Description

If the processor is running in Supervisor mode, the top two elements on the Supervisor Stack are popped and loaded into PC, PSR. If the processor is running in User mode, a privilege mode violation exception occurs.

Example

RTI ; PC, PSR $\leftarrow$ top two values popped off stack.

Note

On an external interrupt or an internal exception, the initiating sequence first changes the privilege mode to Supervisor mode (PSR[15] = 0). Then the PSR and PC of the interrupted program are pushed onto the Supervisor Stack before loading the PC with the starting address of the interrupt or exception service routine. Interrupt and exception service routines run 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 an interrupt, 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.4.

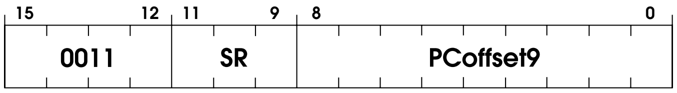
If the processor is running in User mode, a privilege mode violation exception occurs. Section A.4 describes what happens in this case.

ST Store

Assembler Format

ST SR, LABEL

Encoding



Operation

```
mem[PC† + SEXT(PCoffset9)] = SR;
```

Description

The contents of the register specified by SR are 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

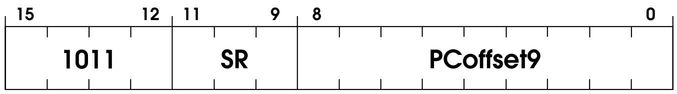
† This is the incremented PC.

STI — Store Indirect

Assembler Format

STI SR, LABEL

Encoding



Operation

```
mem[mem[PC† + SEXT(PCoffset9)]] = SR;
```

Description

The contents of the register specified by SR are 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]] $\leftarrow$ R4

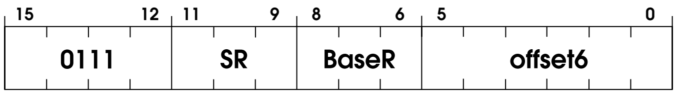
† This is the incremented PC.

STR — Store Base+offset

Assembler Format

STR SR, BaseR, offset6

Encoding



Operation

```
mem[BaseR + SEXT(offset6)] = SR;
```

Description

The contents of the register specified by SR are 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] $\leftarrow$ R4

TRAP — System Call

Assembler Format

TRAP trapvector8

Encoding

15–12	11–8	7–0
1111	0000	trapvect8

Operation

```
R7 = PC;†
PC = mem[ZEXT(trapvect8)];
```

Description

First R7 is loaded with the incremented PC. (This enables a return to the instruction physically following the TRAP instruction in the original program after the service routine 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.2 describes the functions performed by the service routines corresponding to trap vectors x20 to x25.

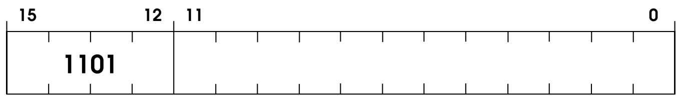
[†] This is the incremented PC.

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.4 describes what happens.

Table A.2 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.

Table A.3 Device Register Assignments

Address	I/O Register Name	I/O Register Function
xFE00	Keyboard status register	Also known as KBSR. The ready bit (bit [15]) indicates if the keyboard has received a new character.
xFE02	Keyboard data register	Also known as KBDR. Bits [7:0] contain the last character typed on the keyboard.
xFE04	Display status register	Also known as 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	Also known as DDR. A character written in the low byte of this register will be displayed on the screen.
xFFFE	Machine control register	Also known as MCR. Bit [15] is the clock enable bit. When cleared, instruction processing stops.

A.4 Interrupt and Exception Processing

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 8-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.4.1 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 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 processor sets the privilege mode to Supervisor mode (PSR[15] = 0).
2. The processor sets the priority level to PL4, the priority level of the interrupting device (PSR[10:8] = 100).
3. R6 is loaded with the Supervisor Stack Pointer (SSP) if it does not already contain the SSP.
4. The PSR and PC of the interrupted process are pushed onto the Supervisor Stack.
5. The keyboard supplies its 8-bit interrupt vector, in this case x80.
6. The processor expands that vector to x0180, the corresponding 16-bit address in the interrupt vector table.
7. 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.4.2 Exceptions

At this time, the LC-3 ISA specifies two exception conditions: privilege mode violation and illegal opcode. The privilege mode violation occurs if the processor encounters the RTI instruction while running in User mode. The illegal opcode exception occurs if the processor encounters the unused opcode (Bits [15:12] = 1101) in the instruction it is is processing.

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

1. The processor sets the privilege mode to Supervisor mode (PSR[15] = 0).
2. R6 is loaded with the Supervisor Stack Pointer (SSP) if it does not already contain the SSP.
3. The PSR and PC of the interrupted process are pushed onto the Supervisor Stack.
4. The exception supplies its 8-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.
5. The processor expands that vector to x0100 or x0101, the corresponding 16-bit address in the interrupt vector table.
6. The PC is loaded with the contents of memory location x0100 or x0101, 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 so 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 two 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.