chapter

5

The LC-3

In Chapter 4, we discussed the basic components of a computer—its memory, its processing unit, including the associated temporary storage (usually a set of registers), input and output devices, and the control unit that directs the activity of all the units (including itself!). We also studied the six phases of the instruction cycle—FETCH, DECODE, ADDRESS EVALUATION, OPERAND FETCH, EXECUTE, and STORE RESULT. We are now ready to introduce a "real" computer, the LC-3. To be more nearly exact, we are ready to introduce the instruction set architecture (ISA) of the LC-3. We have already teased you with a few facts about the LC-3 and a few of its instructions. Now we will examine the ISA of the LC-3 in a more comprehensive way.

Recall from Chapter 1 that the ISA is the interface between what the software commands and what the hardware actually carries out. In this chapter and in Chapters 8 and 9, we will point out the important features of the ISA of the LC-3. You will need these features to write programs in the LC-3's own language, that is, in the LC-3's *machine language.*

A complete description of the ISA of the LC-3 is contained in Appendix A.

5.1 The ISA: Overview

The ISA specifies all the information about the computer that the software has to be aware of. In other words, the ISA specifies everything in the computer that is available to a programmer when he/she writes programs in the computer's own machine language. Thus, the ISA also specifies everything in the computer that is available to someone who wishes to translate programs written in a high-level language like C or Pascal or Fortran or COBOL into the machine language of the computer.

The ISA specifies the memory organization, register set, and instruction set, including opcodes, data types, and addressing modes.

5.1.1 Memory Organization

The LC-3 memory has an address space of 2^{16} (i.e., 65,536) locations, and an addressability of 16 bits. Not all 65,536 addresses are actually used for memory locations, but we will leave that discussion for Chapter 8. Since the normal unit of data that is processed in the LC-3 is 16 bits, we refer to 16 bits as one *word*, and we say the LC-3 is *word-addressable*.

5.1.2 Registers

Since it usually takes far more than one machine cycle to obtain data from memory, the LC-3 provides (like almost all computers) additional temporary storage locations that can be accessed in a single machine cycle.

The most common type of temporary storage locations and the one used in the LC-3 is the general purpose register set. Each register in the set is called a *general purpose register* (GPR). Registers have the same property as memory locations in that they are used to store information that can be retrieved later. The number of bits stored in each register is usually one word. In the LC-3, this means 16 bits.

Registers must be uniquely identifiable. The LC-3 specifies eight GPRs, each identified by a 3-bit register number. They are referred to as R0, R1, ... R7. Figure 5.1 shows a snapshot of the LC-3's register set, sometimes called a *register file*, with the eight values 1, 3, 5, 7, −2, −4, −6, and −8 stored in R0, ... R7, respectively.

Register 0	(R0)	0000000000000001
Register 1	(R1)	0000000000000011
Register 2	(R2)	0000000000000101
Register 3	(R3)	0000000000000111
Register 4	(R4)	1111111111111110
Register 5	(R5)	1111111111111100
Register 6	(R6)	1111111111111010
Register 7	(R7)	1111111111111000

Figure 5.1 The register file before the ADD instruction

Register 0	(R0)	0000000000000001
Register 1	(R1)	0000000000000011
Register 2	(R2)	0000000000000100
Register 3	(R3)	0000000000000111
Register 4	(R4)	1111111111111110
Register 5	(R5)	1111111111111100
Register 6	(R6)	1111111111111010
Register 7	(R7)	1111111111111000

Figure 5.2 The register file after the ADD instruction

Recall that the instruction to ADD the contents of R0 to R1 and store the result in R2 is specified as

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1
	ADD				R2			R0						R1	

where the two *sources* of the ADD instruction are specified in bits [8:6] and bits [2:0]. The *destination* of the ADD result is specified in bits [11:9]. Figure 5.2 shows the contents of the register file of Figure 5.1 AFTER the instruction ADD R2, R1, R0 is executed.

5.1.3 The Instruction Set

An instruction is made up of two things, its *opcode* (what the instruction is asking the computer to do) and its *operands* (who the computer is expected to do it to). The instruction set of an ISA is defined by its set of opcodes, *data types*, and *addressing modes*. The addressing modes determine where the operands are located.

You have just seen an example of one opcode ADD and one addressing mode *register mode*. The operation the instruction is asking the computer to perform is 2's complement integer addition, and the locations where the computer is expected to find the operands are the general purpose registers.

5.1.4 Opcodes

Some ISAs have a very large set of opcodes, one for each of a large number of tasks that a program may wish to carry out. Other ISAs have a very small set of opcodes. Some ISAs have specific opcodes to help with processing scientific calculations. For example, the Hewlett Packard *Precision Architecture* has an instruction that performs a multiply, followed by an add $(A \cdot B) + C$ on three source operands. Other ISAs have instructions that process video images obtained from the World Wide Web. The Intel x86 ISA added a number of instructions Intel calls *MMX* *instructions* because they eXtend the ISA to assist with MultiMedia applications that use the Web. Still other ISAs have specific opcodes to help with handling the tasks of the operating system. For example, the VAX architecture, popular in the 1980s, had an opcode to save all the information associated with one program that was running prior to switching to another program. Almost all computers prefer to use a long sequence of instructions to ask the computer to carry out the task of saving all that information. Although that sounds counterintuitive, there is a rationale for it. Unfortunately, the topic will have to wait for a later semester. The decision as to which instructions to include or leave out of an ISA is usually a hotly debated topic in a company when a new ISA is being specified.

The LC-3 ISA has 15 instructions, each identified by its unique opcode. The opcode is specified by bits [15:12] of the instruction. Since four bits are used to specify the opcode, 16 distinct opcodes are possible. However, the LC-3 ISA specifies only 15 opcodes. The code 1101 has been left unspecified, reserved for some future need that we are not able to anticipate today.

There are three different types of instructions, which means three different types of opcodes: *operates*, *data movement*, and *control*. Operate instructions process information. Data movement instructions move information between memory and the registers and between registers/memory and input/output devices. Control instructions change the sequence of instructions that will be executed. That is, they enable the execution of an instruction other than the one that is stored in the next sequential location in memory.

Figure 5.3 lists all the instructions of the LC-3, the bit encoding [15:12] for each opcode, and the format of each instruction. The use of these formats will be further explained in Sections 5.2, 5.3, and 5.4.

5.1.5 Data Types

A *data type* is a representation of information such that the ISA has opcodes that operate on that representation. There are many ways to represent the same information in a computer. That should not surprise us. In our daily lives, we regularly represent the same information in many different ways. For example, a child, when asked how old he is, might hold up three fingers, signifying he is 3 years old. If the child is particularly precocious, he might write the decimal digit *3* to indicate his age. Or, if he is a CS or CE major at the university, he might write 0000000000000011, the 16-bit binary representation for 3. If he is a chemistry major, he might write $3.0 \cdot 10^0$. All four represent the same entity: 3.

If the ISA has an opcode that operates on information represented by a data type, then we say the ISA **supports** that data type. In Chapter 2, we introduced the only data type supported by the ISA of the LC-3: 2's complement integers.

5.1.6 Addressing Modes

An addressing mode is a mechanism for specifying where the operand is located. An operand can generally be found in one of three places: in memory, in a register, or as a part of the instruction. If the operand is a part of the instruction, we refer to it as a *literal* or as an *immediate* operand. The term *literal* comes from the

	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 5.3 Formats of the entire LC-3 instruction set. NOTE: $^+$ indicates instructions that modify condition codes

fact that the bits of the instruction literally form the operand. The term *immediate* comes from the fact that we have the operand immediately, that is, we don't have to look elsewhere for it.

The LC-3 supports five addressing modes: immediate (or literal), register, and three memory addressing modes: *PC-relative*, *indirect*, and *Base+offset*. We will see in Section 5.2 that operate instructions use two addressing modes: register and immediate. We will see in Section 5.3 that data movement instructions use all five modes.

5.1.7 Condition Codes

One final item will complete our overview of the ISA of the LC-3: condition codes. Almost all ISAs allow the instruction sequencing to change on the basis of a previously generated result. The LC-3 has three single-bit registers that are set (set to 1) or cleared (set to 0) each time one of the eight general purpose registers is written. The three single-bit registers are called *N*, *Z*, and *P*, corresponding to their meaning: negative, zero, and positive. Each time a GPR is written, the N, Z, and P registers are individually set to 0 or 1, corresponding to whether the result written to the GPR is negative, zero, or positive. That is, if the result is negative, the N register is set, and Z and P are cleared. If the result is zero, Z is set and N and P are cleared. Finally, if the result is positive, P is set and N and Z are cleared.

Each of the three single-bit registers is referred to as a *condition code* because the condition of that bit can be used by one of the control instructions to change the execution sequence. The x86 and SPARC are two examples of ISAs that use condition codes to do this. We show how the LC-3 does it in Section 5.4.

5.2 Operate Instructions

Operate instructions process data. Arithmetic operations (like ADD, SUB, MUL, and DIV) and logical operations (like AND, OR, NOT, XOR) are common examples. The LC-3 has three operate instructions: ADD, AND, and NOT.

The **NOT** (opcode = 1001) instruction is the only operate instruction that performs a *unary* operation, that is, the operation requires one source operand. The NOT instruction bit-wise complements a 16-bit source operand and stores the result of this operation in a destination. NOT uses the register addressing mode for both its source and destination. Bits [8:6] specify the source register and bits [11:9] specify the destination register. Bits [5:0] must contain all 1s.

If R5 initially contains 0101000011110000, after executing the following instruction:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	1	0	1	1	1	1	1	1	1
	NOT				R3			R5							

R3 will contain 1010111100001111.

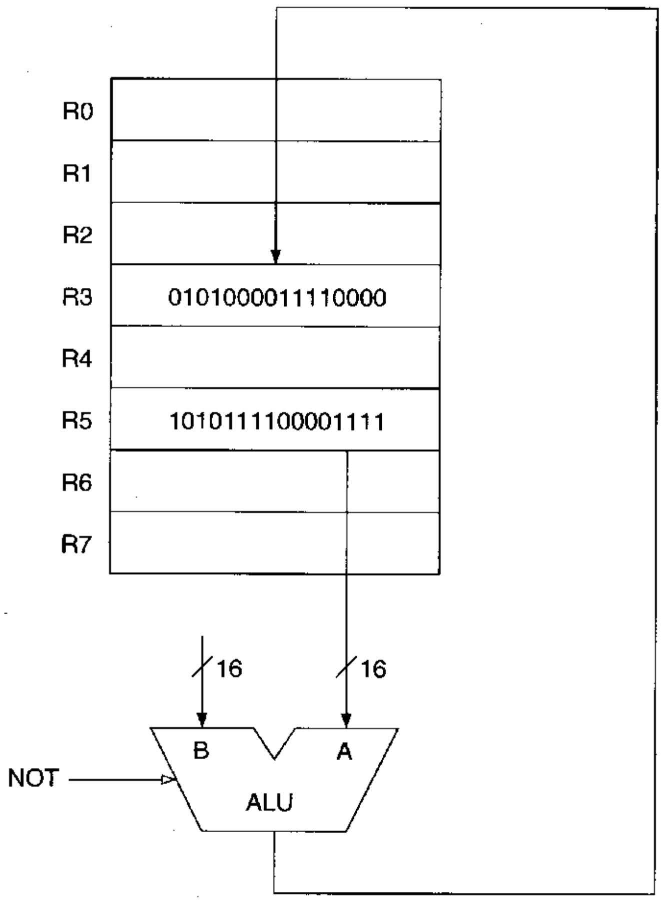

Figure 5.4 Data path relevant to the execution of NOT R3, R5

Figure 5.4 shows the key parts of the data path that are used to perform the NOT instruction shown here. Since NOT is a unary operation, only the A input of the ALU is relevant. It is sourced from R5. The control signal to the ALU directs the ALU to perform the bit-wise complement operation. The output of the ALU (the result of the operation) is stored into R3.

The **ADD** (opcode = 0001) and **AND** (opcode = 0101) instructions both perform *binary* operations; they require two 16-bit source operands. The ADD instruction performs a 2's complement addition of its two source operands. The AND instruction performs a bit-wise AND of each pair of bits in its two 16-bit operands. Like the NOT, the ADD and AND use the register addressing mode for one of the source operands and for the destination operand. Bits [8:6] specify the source register and bits [11:9] specify the destination register (where the result will be written).

The second source operand for both ADD and AND instructions can be specified by either register mode or as an immediate operand. Bit [5] determines which is used. If bit [5] is 0, then the second source operand uses a register, and bits [2:0] specify which register. In that case, bits [4:3] are set to 0 to complete the specification of the instruction.

For example, if R4 contains the value 6 and R5 contains the value −18, then after the following instruction is executed

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	1
	ADD				R1			R4						R5	

R1 will contain the value −12.

If bit [5] is 1, the second source operand is contained within the instruction. In fact, the second source operand is obtained by sign-extending bits [4:0] to 16 bits before performing the ADD or AND. Figure 5.5 shows the key parts of the data path that are used to perform the instruction ADD R1, R4, #−2.

Since the immediate operand in an ADD or AND instruction must fit in bits [4:0] of the instruction, not all 2's complement integers can be immediate operands. Which integers are OK (i.e., which integers can be used as immediate operands)?

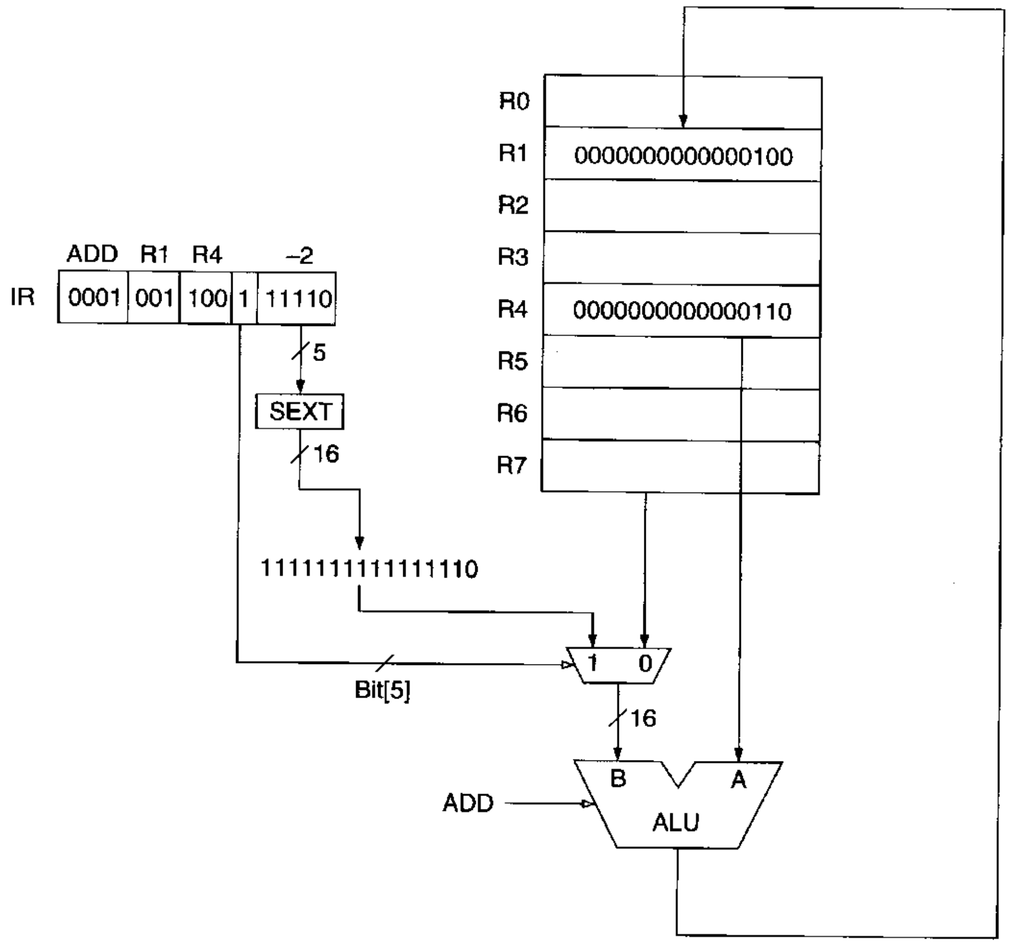

Figure 5.5 Data path relevant to the execution of ADD R1, R4, #-2

Example 5.1

What does the following instruction do?

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0

ANSWER: Register 2 is cleared (i.e., set to all 0s).

Example 5.2

What does the following instruction do?

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	1	0	1	0	0	0	0	1

ANSWER: Register 6 is incremented (i.e., R6 ← R6 + 1).

Note that a register can be used as a source and also as a destination in the same instruction. This is true for all the instructions in the LC-3.

Example 5.3

Recall that the 2's complement of a number can be obtained by complementing the number and adding 1. Therefore, assuming the values A and B are in R0 and R1, what sequence of three instructions performs "A minus B" and writes the result into R2?

ANSWER:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 ← NOT(B)
	NOT				R1			R1								
0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	1	R2 ← -B
	ADD				R2			R1					1			
0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	R2 ← A +(-B)
	ADD				R2			R0						R2		

Question: What distasteful result is also produced by this sequence? How can it easily be avoided?

5.3 Data Movement Instructions

Data movement instructions move information between the general purpose registers and memory, and between the registers and the input/output devices. We will ignore for now the business of moving information from input devices to registers and from registers to output devices. This will be the major topic of Chapter 8 and an important part of Chapter 9 as well. In this chapter, we will confine ourselves to moving information between memory and the general purpose registers.

The process of moving information from memory to a register is called a *load*, and the process of moving information from a register to memory is called a *store*. In both cases, the information in the location containing the source operand remains unchanged. In both cases, the location of the destination operand is overwritten with the source operand, destroying the prior value in the destination location in the process.

The LC-3 contains seven instructions that move information: LD, LDR, LDI, LEA, ST, STR, and STI.

The format of the load and store instructions is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode				DR or SR			Addr Gen bits								

Data movement instructions require two operands, a source and a destination. The source is the data to be moved; the destination is the location where it is moved to. One of these locations is a register, the second is a memory location or an input/output device. As we said earlier, in this chapter the second operand will be assumed to be in memory. We will save for Chapter 8 the cases where the second operand specifies an input or output device.

Bits [11:9] specify one of these operands, the register. If the instruction is a load, *DR* refers to the destination register that will contain the value after it is read from memory (at the completion of the instruction cycle). If the instruction is a store, *SR* refers to the register that contains the value that will be written to memory.

Bits [8:0] contain the *address generation bits*. That is, bits [8:0] encode information that is used to compute the 16-bit address of the second operand. In the case of the LC-3's data movement instructions, there are four ways to interpret bits [8:0]. They are collectively called *addressing modes*. The opcode specifies how to interpret bits [8:0]. That is, the LC-3's opcode specifies which addressing mode should be used to obtain the operand from bits [8:0] of the instruction.

5.3.1 PC-Relative Mode

LD (opcode = 0010) and **ST** (opcode = 0011) specify the *PC-relative* addressing mode. This addressing mode is so named because bits [8:0] of the instruction specify an offset relative to the PC. The memory address is computed by sign-extending bits [8:0] to 16 bits, and adding the result to the incremented PC. The incremented PC is the contents of the program counter after the FETCH phase; that is, after the PC has been incremented. If a load, the memory location corresponding to the computed memory address is read, and the result loaded into the register specified by bits [11:9] of the instruction.

If the instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	0	1	0	1	1	1	1
LD				R2			x1AF								

is located at x4018, it will cause the contents of x3FC8 to be loaded into R2.

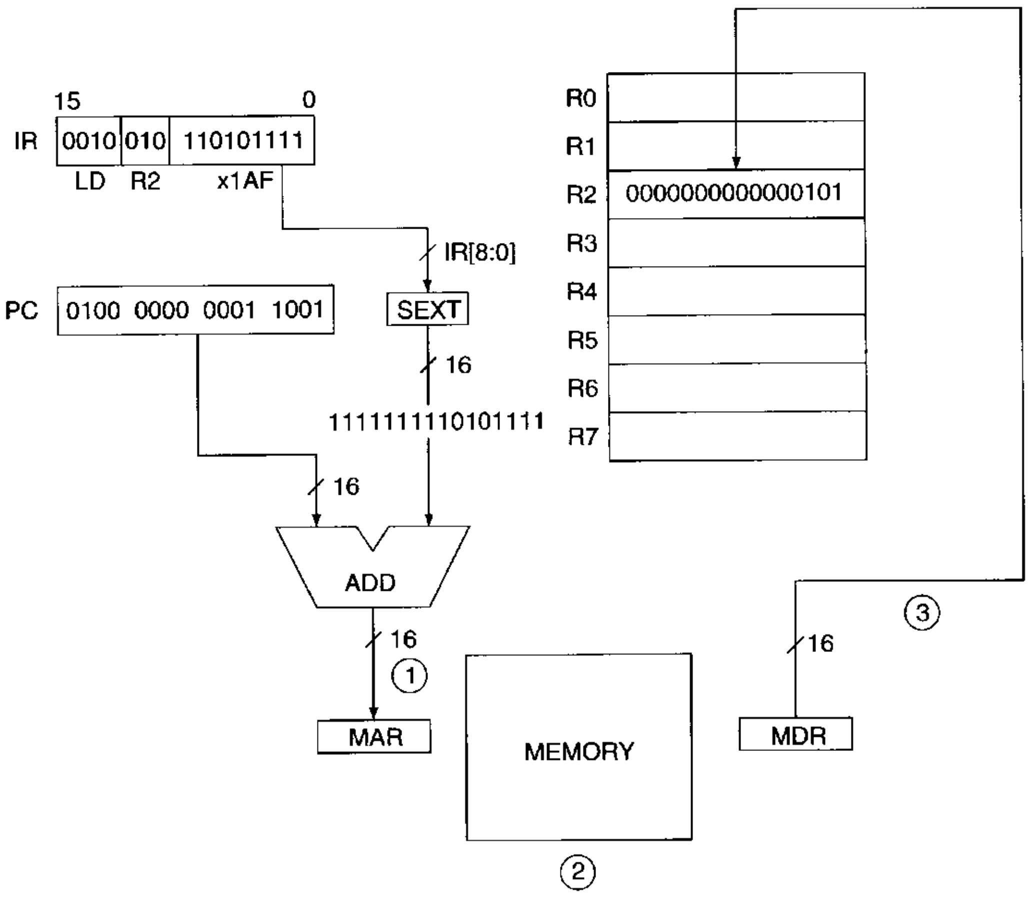

Figure 5.6 Data path relevant to execution of LD R2, x1AF

Figure 5.6 shows the relevant parts of the data path required to execute this instruction. The three steps of the LD instruction are identified. In step 1, the incremented PC (x4019) is added to the sign-extended value contained in IR[8:0] (xFFAF), and the result (x3FC8) is loaded into the MAR. In step 2, memory is read and the contents of x3FC8 are loaded into the MDR. Suppose the value stored in x3FC8 is 5. In step 3, the value 5 is loaded into R2, completing the instruction cycle.

Note that the address of the memory operand is limited to a small range of the total memory. That is, the address can only be within +256 or −255 locations of the LD or ST instruction since the PC is incremented before the offset is added. This is the range provided by the sign-extended value contained in bits [8:0] of the instruction.

5.3.2 Indirect Mode

LDI (opcode = 1010) and **STI** (opcode = 1011) specify the *indirect* addressing mode. An address is first formed exactly the same way as with LD and ST. However, instead of this address **being** the address of the operand to be loaded or stored, it **contains** the address of the operand to be loaded or stored. Hence the name *indirect*. Note that the address of the operand can be anywhere in the computer's memory, not just within the range provided by bits [8:0] of the instruction as is the case for LD and ST. The destination register for the LDI and the source register for STI, like all the other loads and stores, are specified in bits [11:9] of the instruction.

If the instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	1	1	1	1	0	0	1	1	0	0
LDI				R3			x1CC								

is in x4A1B, and the contents of x49E8 is x2110, execution of this instruction results in the contents of x2110 being loaded into R3.

Figure 5.7 shows the relevant parts of the data path required to execute this instruction. As is the case with the LD and ST instructions, the first step consists of adding the incremented PC (x4A1C) to the sign-extended value contained in IR[8:0] (xFFCC), and the result (x49E8) loaded into the MAR. In step 2, memory is read and the contents of x49E8 (x2110) is loaded into the MDR. In step 3, since x2110 is not the operand, but the address of the operand, it is loaded into the MAR. In step 4, memory is again read, and the MDR again loaded. This time the MDR is loaded with the contents of x2110. Suppose the value −1 is stored in memory location x2110. In step 5, the contents of the MDR (i.e., −1) are loaded into R3, completing the instruction cycle.

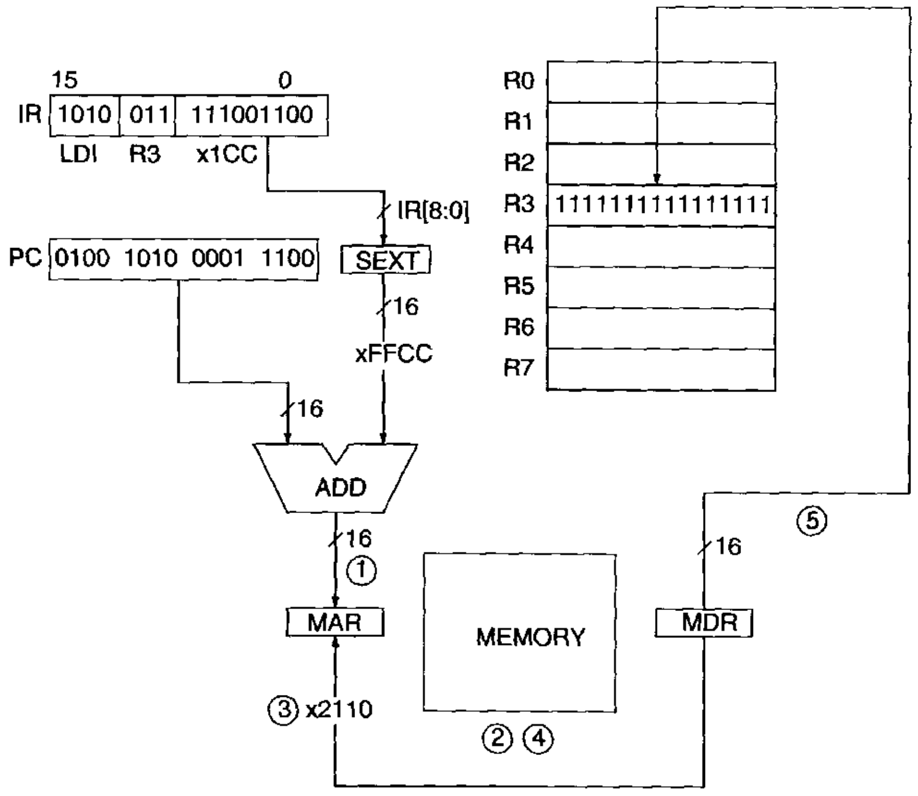

Figure 5.7 Data path relevant to the execution of LDI R3, x1CC

5.3.3 Base+offset Mode

LDR (opcode = 0110) and **STR** (opcode = 0111) specify the *Base+offset* addressing mode. The Base+offset mode is so named because the address of the operand is obtained by adding a sign-extended 6-bit offset to a base register. The 6-bit offset is **literally** taken from the instruction, bits [5:0]. The base register is specified by bits [8:6] of the instruction.

The Base+offset addressing uses the 6-bit value as a 2's complement integer between −32 and +31. Thus it must first be sign-extended to 16 bits before it is added to the base register.

If R2 contains the 16-bit quantity x2345, the instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	1
LDR				R1			R2			x1D					

loads R1 with the contents of x2362.

Figure 5.8 shows the relevant parts of the data path required to execute this instruction. First the contents of R2 (x2345) are added to the sign-extended value contained in IR[5:0] (x001D), and the result (x2362) is loaded into the MAR. Second, memory is read, and the contents of x2362 are loaded into the MDR. Suppose the value stored in memory location x2362 is x0F0F. Third, and finally, the contents of the MDR (in this case, x0F0F) are loaded into R1.

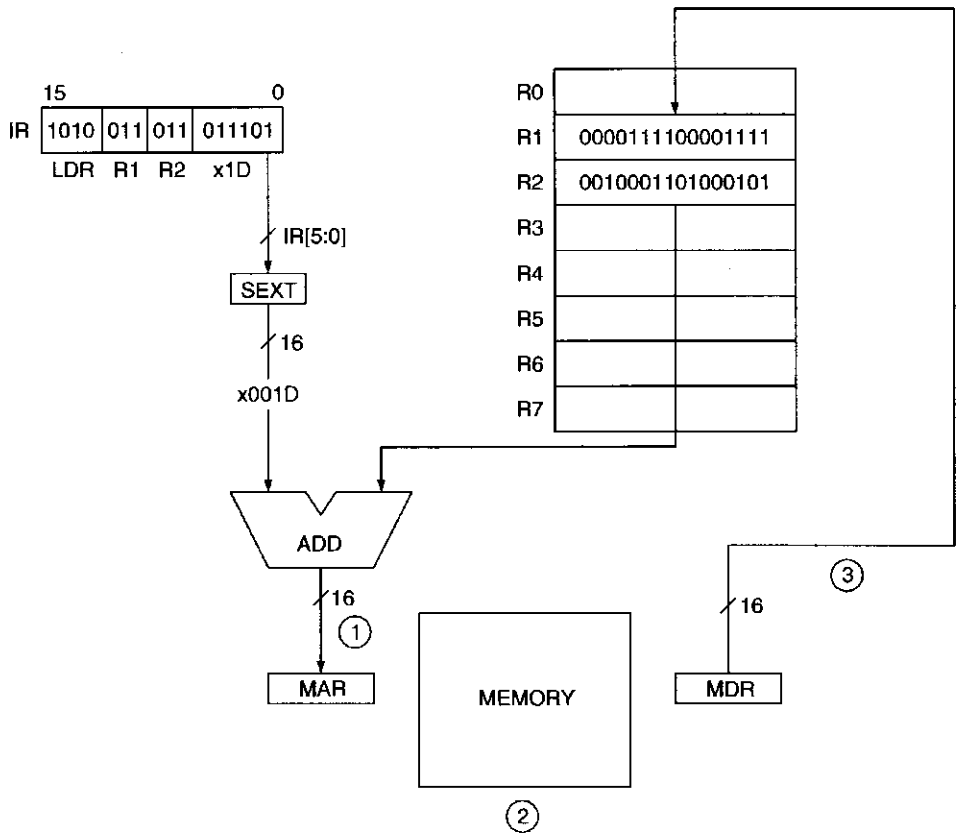

Figure 5.8 Data path relevant to the execution of LDR R1, R2, x1D

Note that the Base+offset addressing mode also allows the address of the operand to be anywhere in the computer's memory.

5.3.4 Immediate Mode

The fourth and last addressing mode used by the data movement instructions is the *immediate* (or, *literal*) addressing mode. It is used only with the load effective address (LEA) instruction. **LEA** (opcode = 1110) loads the register specified by bits [11:9] of the instruction with the value formed by adding the incremented program counter to the sign-extended bits [8:0] of the instruction. The immediate addressing mode is so named because the operand to be loaded into the destination register is obtained immediately, that is, without requiring any access of memory.

The LEA instruction is useful to initialize a register with an address that is very close to the address of the instruction doing the initializing. If memory location x4018 contains the instruction LEA R5, #−3, and the PC contains x4018,

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	1	1	1	1	1	0	1
LEA				R5			−3								

R5 will contain x4016 after the instruction at x4018 is executed.

Figure 5.9 shows the relevant parts of the data path required to execute the LEA instruction. Note that no access to memory is required to obtain the value to be loaded.

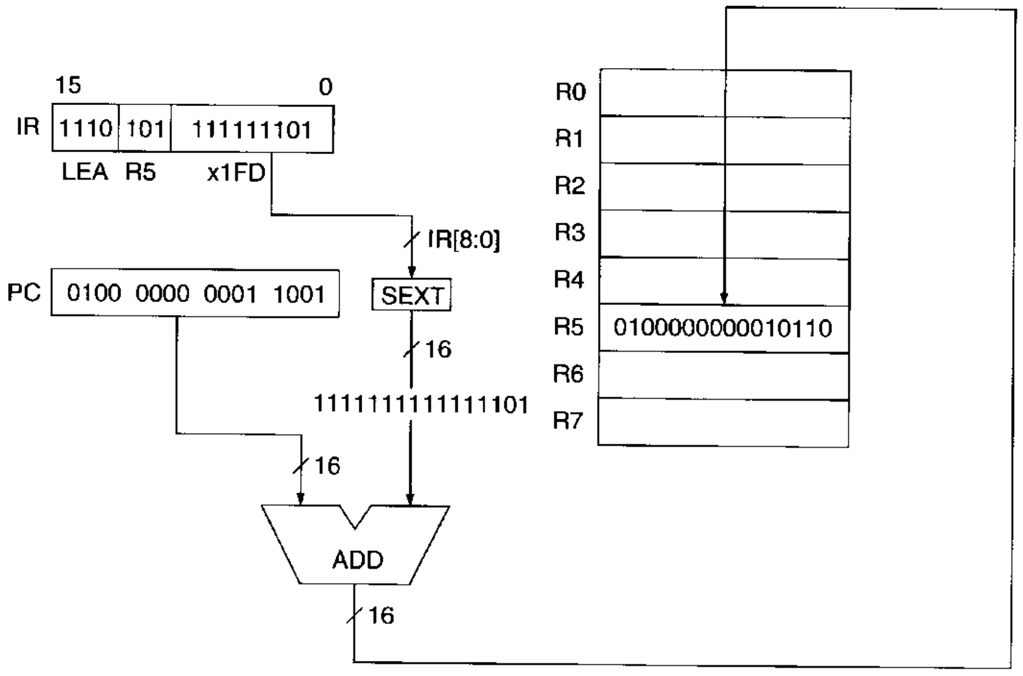

Figure 5.9 Data path relevant to the execution of LEA R5, #−3

Again, LEA is the *only* load instruction that does not access memory to obtain the information it will load into the DR. It loads into the DR the address formed from the incremented PC and the address generation bits of the instruction.

5.3.5 An Example

We conclude our study of addressing modes with a comprehensive example. Assume the contents of memory locations x30F6 through x30FC are as shown in Figure 5.10, and the PC contains x30F6. We will examine the effects of carrying out the instruction cycle seven consecutive times.

The PC points initially to location x30F6. That is, the content of the PC is the address x30F6. Therefore, the first instruction to be executed is the one stored in location x30F6. The opcode of that instruction is 1110, which identifies the load effective address instruction (LEA). LEA loads the register specified by bits [11:9] with the address formed by sign-extending bits [8:0] of the instruction and adding the result to the incremented PC. The 16-bit value obtained by sign-extending bits [8:0] of the instruction is xFFFD. The incremented PC is x30F7. Therefore, at the end of execution of the LEA instruction, R1 contains x30F4, and the PC contains x30F7.

The second instruction to be executed is the one stored in location x30F7. The opcode 0001 identifies the ADD instruction, which stores the result of adding the contents of the register specified in bits [8:6] to the sign-extended immediate in bits [4:0] (since bit [5] is 1) in the register specified by bits [11:9]. Since the previous instruction loaded x30F4 into R1, and the sign-extended immediate value is x000E, the value to be loaded into R2 is x3102. At the end of execution of this instruction, R2 contains x3102, and the PC contains x30F8. R1 still contains x30F4.

The third instruction to be executed is stored in x30F8. The opcode 0011 specifies the ST instruction, which stores the contents of the register specified by bits [11:9] of the instruction into the memory location whose address is computed using the PC-relative addressing mode. That is, the address is computed by adding the incremented PC to the 16-bit value obtained by sign-extending bits [8:0] of the instruction. The 16-bit value obtained by sign-extending bits [8:0] of the instruction is xFFFB. The incremented PC is x30F9. Therefore, at the end of

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x30F6	1	1	1	0	0	0	1	1	1	1	1	1	1	1	0	1	**R1<- PC-3**
x30F7	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	**R2<- R1+14**
x30F8	0	0	1	1	0	1	0	1	1	1	1	1	1	0	1	1	**M[x30F4]<- R2**
x30F9	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	**R2<- 0**
x30FA	0	0	0	1	0	1	0	0	1	0	1	0	0	1	0	1	**R2<- R2+5**
x30FB	0	1	1	1	0	1	0	0	0	1	0	0	1	1	1	0	**M[R1+14]<- R2**
x30FC	1	0	1	0	0	1	1	1	1	1	1	1	0	1	1	1	**R3<- M[M[x3F04]]**

Figure 5.10 Addressing mode example

execution of the ST instruction, memory location x30F4 contains x3102, and the PC contains x30F9.

At x30F9, we find the opcode 0101, which represents the AND instruction. After execution, R2 contains the value 0, and the PC contains x30FA.

At x30FA, we find the opcode 0001, signifying the ADD instruction. After execution, R2 contains the value 5, and the PC contains x30FB.

At x30FB, we find the opcode 0111, signifying the STR instruction. The STR instruction (like the LDR instruction) uses the Base+offset addressing mode. The memory address is obtained by adding the contents of the register specified by bits [8:6] (the BASE register) to the sign-extended offset contained in bits [5:0]. In this case, bits [8:6] specify R1. The contents of R1 are still x30F4. The 16-bit sign-extended offset is x000E. Since x30F4 + x000E is x3102, the memory address is x3102. The STR instruction stores into x3102 the contents of the register specified by bits [11:9], that is, R2. Recall that the previous instruction (at x30FA) stored the value 5 into R2. Therefore, at the end of execution of this instruction, location x3102 contains the value 5, and the PC contains x30FC.

At x30FC, we find the opcode 1010, signifying the LDI instruction. The LDI instruction (like the STI instruction) uses the indirect addressing mode. The memory address is obtained by first forming an address as is done in the PC-relative addressing mode. In this case, the 16-bit value obtained by sign-extending bits [8:0] of the instruction is xFFF7. The incremented PC is x30FD. Their sum is x30F4, which is the address of the operand address. Memory location x30F4 contains x3102. Therefore, x3102 is the operand address. The LDI instruction loads the value found at this address (in this case 5) into the register identified by bits [11:9] of the instruction (in this case R3). At the end of execution of this instruction, R3 contains the value 5 and the PC contains x30FD.

5.4 Control Instructions

Control instructions change the sequence of the instructions that are executed. If there were no control instructions, the next instruction fetched after the current instruction finishes would be the instruction located in the next sequential memory location. As you know, this is because the PC is incremented in the FETCH phase of each instruction. We will see momentarily that it is often useful to be able to break that sequence.

The LC-3 has five opcodes that enable this sequential flow to be broken: conditional branch, unconditional jump, subroutine (sometimes called *function*) call, TRAP, and return from interrupt. In this section, we will deal almost exclusively with the most common control instruction, the *conditional branch*. We will also introduce the unconditional jump and the TRAP instruction. The TRAP instruction is particularly useful because, among other things, it allows a programmer to get information into and out of the computer without fully understanding the intricacies of the input and output devices. However, most of the discussion of the TRAP instruction and all of the discussion of the subroutine call and the return from interrupt we will leave for Chapters 9 and 10.

5.4.1 Conditional Branches

The format of the conditional branch instruction (opcode = 0000) is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	N	Z	P	PCoffset								

Bits [11], [10], and [9] correspond to the three condition codes discussed in Section 5.1.7. Recall that in the LC-3, **all** instructions that write values into the general purpose registers set the three condition codes (i.e., the single-bit registers N, Z, P) in accordance with whether the value written is negative, zero, or positive. These instructions are ADD, AND, NOT, LD, LDI, LDR, and LEA.

The condition codes are used by the conditional branch instruction to determine whether to change the instruction flow; that is, whether to depart from the usual sequential execution of instructions that we get as a result of incrementing PC during the FETCH phase of each instruction.

The instruction cycle is as follows: FETCH and DECODE are the same for all instructions. The PC is incremented during FETCH. The EVALUATE ADDRESS phase is the same as that for LD and ST: the address is computed by adding the incremented PC to the 16-bit value formed by sign-extending bits [8:0] of the instruction.

During the EXECUTE phase, the processor examines the condition codes whose corresponding bits in the instruction are 1. That is, if bit [11] is 1, condition code N is examined. If bit [10] is 1, condition code Z is examined. If bit [9] is 1, condition code P is examined. If any of bits [11:9] are 0, the corresponding condition codes are not examined. If any of the condition codes that are examined are in state 1, then the PC is loaded with the address obtained in the EVALUATE ADDRESS phase. If none of the condition codes that are examined are in state 1, the PC is left unchanged. In that case, in the next instruction cycle, the next sequential instruction will be fetched.

For example, if the last value loaded into a general purpose register was 0, then the current instruction (located at x4027) shown here

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	1
		BR		n	z	p					x0D9				

would load the PC with x4101, and the next instruction executed would be the one at x4101, rather than the instruction at x4028.

Figure 5.11 shows the data path elements that are required to execute this instruction. Note the logic required to determine whether the sequential instruction flow should be broken. In this case the answer is yes, and the PC is loaded with x4101, replacing x4028, which had been loaded during the FETCH phase of the conditional branch instruction.

If all three bits [11:9] are 1, then all three condition codes are examined. In this case, since the last result stored into a register had to be either negative, zero, or positive (there are no other choices), one of the three condition codes must be in state 1. Since all three are examined, the PC is loaded with the address obtained in the EVALUATE ADDRESS phase. We call this an *un*conditional branch since

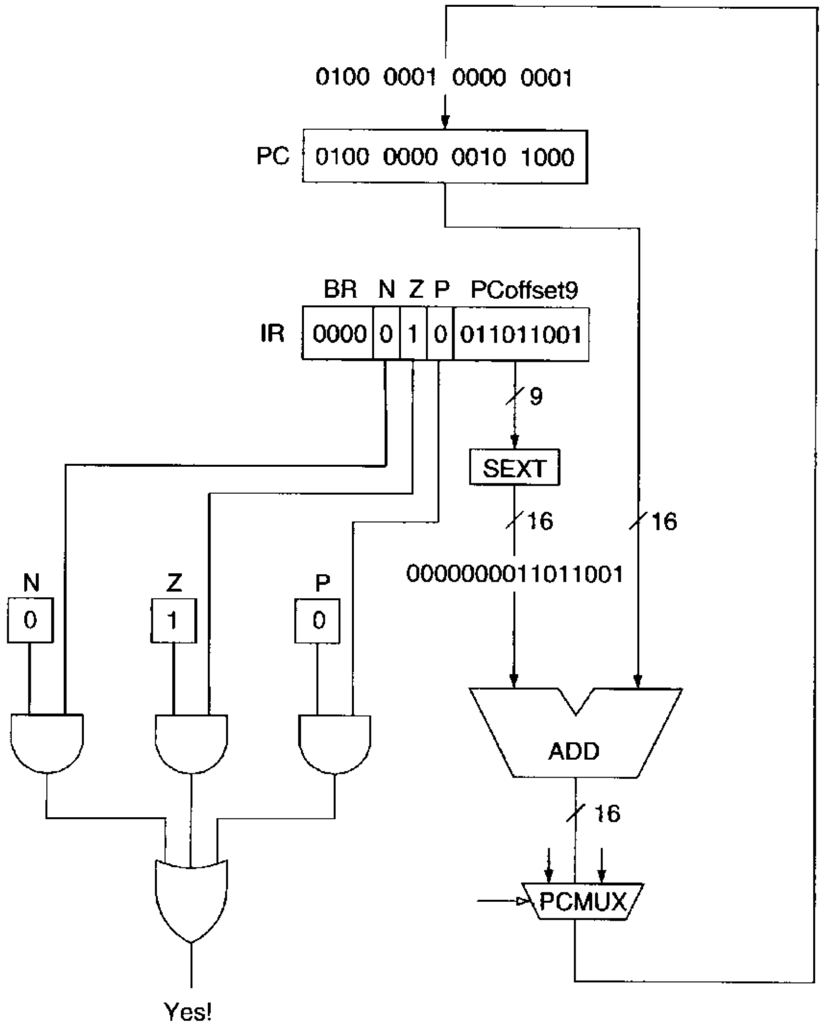

Figure 5.11 Data path relevant to the execution of BRz x0D9

the instruction flow is changed unconditionally, that is, independent of the data that is being processed.

For example, if the following instruction,

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	0	0	0	0	1	0	1
BR				n	z	p	x185								

located at x507B, is executed, the PC is loaded with x5001.

What happens if all three bits [11:9] in the BR instruction are 0?

5.4.2 An Example

We are ready to show by means of a simple example the value of having control instructions in the instruction set.

Suppose we know that the 12 locations x3100 to x310B contain integers, and we wish to compute the sum of these 12 integers.

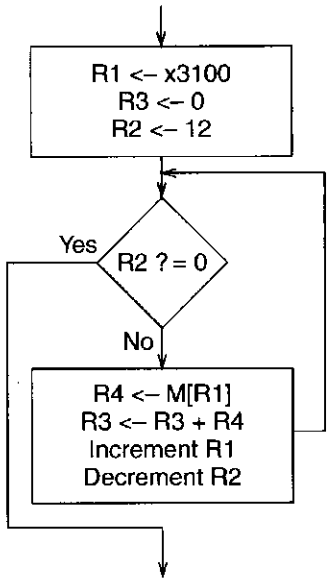

Figure 5.12 An algorithm for adding 12 integers

A flowchart for an algorithm to solve the problem is shown in Figure 5.12.

First, as in all algorithms, we must *initialize our variables*. That is, we must set up the initial values of the variables that the computer will use in executing the program that solves the problem. There are three such variables: the address of the next integer to be added (assigned to R1), the running sum (assigned to R3), and the number of integers left to be added (assigned to R2). The three variables are initialized as follows: The address of the first integer to be added is put in R1. R3, which will keep track of the running sum, is initialized to 0. R2, which will keep track of the number of integers left to be added, is initialized to 12. Then the process of adding begins.

The program repeats the process of loading into R4 one of the 12 integers, and adding it to R3. Each time we perform the ADD, we increment R1 so it will point to (i.e., contain the address of) the next number to be added and decrement R2 so we will know how many numbers still need to be added. When R2 becomes zero, the Z condition code is set, and we can detect that we are done.

The 10-instruction program shown in Figure 5.13 accomplishes the task.

The details of the program execution are as follows: The program starts with PC = x3000. The first instruction (at location x3000) loads R1 with the address x3100. (The incremented PC is x3001; the sign-extended PCoffset is x00FF.)

The instruction at x3001 clears R3. R3 will keep track of the running sum, so it must start off with the value 0. As we said previously, this is called *initializing* the SUM to zero.

The instructions at x3002 and x3003 set the value of R2 to 12, the number of integers to be added. R2 will keep track of how many numbers have already been added. This will be done (by the instruction contained in x3008) by decrementing R2 after each addition takes place.

The instruction at x3004 is a conditional branch instruction. Note that bit [10] is a 1. That means that the Z condition code will be examined. If it is set, we know

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	`R1<- 3100`
x3001	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	`R3 <- 0`
x3002	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	`R2 <- 0`
x3003	0	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0	`R2 <- 12`
x3004	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	`BRz x300A`
x3005	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	`R4 <- M[R1]`
x3006	0	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0	`R3 <- R3+R4`
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	`R1 <- R1+1`
x3008	0	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	`R2 <- R2-1`
x3009	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0	`BRnzp x3004`

Figure 5.13 A program that implements the algorithm of Figure 5.12

R2 must have just been decremented to 0. That means there are no more numbers to be added and we are done. If it is clear, we know we still have work to do and we continue.

The instruction at x3005 loads the contents of x3100 (i.e., the first integer) into R4, and the instruction at x3006 adds it to R3.

The instructions at x3007 and x3008 perform the necessary bookkeeping. The instruction at x3007 increments R1, so R1 will point to the next location in memory containing an integer to be added (in this case, x3101). The instruction at x3008 decrements R2, which is keeping track of the number of integers still to be added, as we have already explained, and sets the N, Z, and P condition codes.

The instruction at x3009 is an unconditional branch, since bits [11:9] are all 1. It loads the PC with x3004. It also does not affect the condition codes, so the next instruction to be executed (the conditional branch at x3004) will be based on the instruction executed at x3008.

This is worth saying again. The conditional branch instruction at x3004 follows the instruction at x3009, which does not affect condition codes, which in turn follows the instruction at x3008. Thus, the conditional branch instruction at x3004 will be based on the condition codes set by the instruction at x3008. The instruction at x3008 sets the condition codes depending on the value produced by decrementing R2. As long as there are still integers to be added, the ADD instruction at x3008 will produce a value greater than zero and therefore clear the Z condition code. The conditional branch instruction at x3004 examines the Z condition code. As long as Z is clear, the PC will not be affected, and the next instruction cycle will start with an instruction fetch from x3005.

The conditional branch instruction causes the execution sequence to follow: x3000, x3001, x3002, x3003, x3004, x3005, x3006, x3007, x3008, x3009, x3004, x3005, x3006, x3007, x3008, x3009, x3004, x3005, and so on until the value in R2 becomes 0. The next time the conditional branch instruction at x3004 is executed, the PC is loaded with x300A, and the program continues at x300A with its next activity.

Finally, it is worth noting that we could have written a program to add these 12 integers **without** any control instructions. We still would have needed the LEA instruction in x3000 to initialize R1. We would not have needed the instruction at x3001 to initialize the running sum, nor the instructions at x3002, and x3003 to initialize the number of integers left to be added. We could have loaded the contents of x3100 directly into R3, and then repeatedly (by incrementing R1, loading the next integer into R4, and adding R4 to the running sum in R3) added the remaining 11 integers. After the addition of the twelfth integer, we would go on to the next task, as does the example of Figure 5.13 with the branch instruction in x3004.

Unfortunately, instead of a 10-instruction program, we would have had a 35-instruction program. Moreover, if we had wished to add 100 integers without any control instructions instead of 12, we would have had a 299-instruction program instead of 10. The control instructions in the example of Figure 5.13 permit the reuse of sequences of code by breaking the sequential instruction execution flow.

5.4.3 Two Methods for Loop Control

We use the term *loop* to describe a sequence of instructions that get executed again and again under some controlling mechanism. The example of adding 12 integers contains a loop. Each time the *body* of the loop executes, one more integer is added to the running total, and the counter is decremented so we can detect whether there are any more integers left to add. Each time the loop body executes is called one *iteration* of the loop.

There are two common methods for controlling the number of iterations of a loop. One method we just examined: the use of a counter. If we know we wish to execute a loop *n* times, we simply set a counter to *n*, then after each execution of the loop, we decrement the counter and check to see if it is zero. If it is not zero, we set the PC to the start of the loop and continue with another iteration.

A second method for controlling the number of executions of a loop is to use a *sentinel*. This method is particularly effective if we do not know ahead of time how many iterations we will want to perform. Each iteration is usually based on processing a value. We append to our sequence of values to be processed a value that we know ahead of time can never occur (i.e., the sentinel). For example, if we are adding a sequence of numbers, a sentinel could be a # or a *, that is, something that is not a number. Our loop test is simply a test for the occurrence of the sentinel. When we find it, we know we are done.

5.4.4 Example: Adding a Column of Numbers Using a Sentinel

Suppose in our example of Section 5.4.2, we know the values stored in locations x3100 to x310B are all positive. Then we could use any negative number as a sentinel. Let's say the sentinel stored at memory address x310C is -1. The resulting flowchart for the program is shown in Figure 5.14 and the resulting program is shown in Figure 5.15.

As before, the instruction at x3000 loads R1 with the address of the first value to be added, and the instruction at x3001 initializes R3 (which keeps track of the sum) to 0.

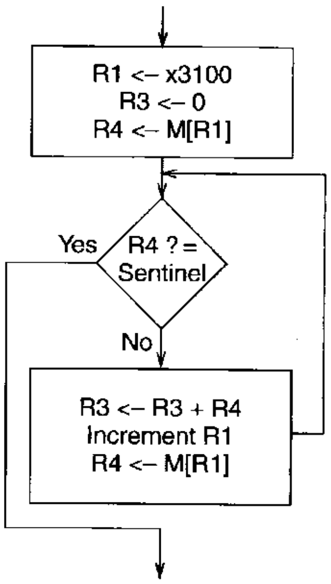

Figure 5.14 An algorithm showing the use of a sentinel for loop control

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	`R1<- x3100`
x3001	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	`R3 <- 0`
x3002	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	`R4 <- M[R1]`
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	`BRn x3008`
x3004	0	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0	`R3 <- R3+R4`
x3005	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	`R1 <- R1+1`
x3006	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	`R4 <- M[R1]`
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	1	`BRnzp x3003`

Figure 5.15 A program that implements the algorithm of Figure 5.14

At x3002, we load the contents of the next memory location into R4. If the sentinel is loaded, the N condition code is set.

The conditional branch at x3003 examines the N condition code, and if it is set, sets PC to x3008 and onto the next task to be done. If the N condition code is clear, R4 must contain a valid number to be added. In this case, the number is added to R3 (x3004), R1 is incremented to point to the next memory location (x3005), R4 is loaded with the contents of the next memory location (x3006), and the PC is loaded with x3003 to begin the next iteration (x3007).

5.4.5 The JMP Instruction

The conditional branch instruction, for all its capability, does have one unfortunate limitation. The next instruction executed must be within the range of addresses that can be computed by adding the incremented PC to the sign-extended offset obtained from bits [8:0] of the instruction. Since bits [8:0] specify a 2's complement integer, the next instruction executed after the conditional branch can be at most +256 or −255 locations from the branch instruction itself. What if we would like to execute next an instruction that is 1,000 locations from the current instruction. We cannot fit the value 1,000 into the 9-bit field; ergo, the conditional branch instruction does not work.

The LC-3 ISA does provide an instruction **JMP** (opcode = 1100) that can do the job. An example follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	JMP							R2							

The JMP instruction loads the PC with the contents of the register specified by bits [8:6] of the instruction. If this JMP instruction is located at address x4000, R2 contains the value x6600, and the PC contains x4000, then the instruction at x4000 (the JMP instruction) will be executed, followed by the instruction located at x6600. Since registers contain 16 bits, the full address space of memory, the JMP instruction has no limitation on where the next instruction to be executed must reside.

5.4.6 The TRAP Instruction

Finally, because it will be useful long before Chapter 9 to get data into and out of the computer, we introduce the TRAP instruction now. The **TRAP** (opcode = 1111) instruction changes the PC to a memory address that is part of the operating system so that the operating system will perform some task in behalf of the program that is executing. In the language of operating system jargon, we say the TRAP instruction invokes an operating system SERVICE CALL. Bits [7:0] of the TRAP instruction form the *trapvector*, which identifies the service call that the program wishes the operating system to perform. Table A.2 contains the trapvectors for all the service calls that we will use with the LC-3 in this book.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	trapvector							

Once the operating system is finished performing the service call, the program counter is set to the address of the instruction following the TRAP instruction, and the program continues. In this way, a program can, during its execution, request services from the operating system and continue processing after each such service is performed. The services we will require for now are

```
* Input a character from the keyboard (trapvector = x23).
* Output a character to the monitor (trapvector = x21).
* Halt the program (trapvector = x25).
```

Exactly how the LC-3 carries out the interaction between operating system and executing programs is an important topic for Chapter 9.

5.5 Another Example: Counting Occurrences of a Character

We will finish our introduction to the ISA of the LC-3 with another example program. We would like to be able to input a character from the keyboard and then count the number of occurrences of that character in a file. Finally, we would like to display that count on the monitor. We will simplify the problem by assuming that the number of occurrences of any character that we would be interested in is small. That is, there will be at most nine occurrences. This simplification allows us to not have to worry about complex conversion routines between the binary count and the ASCII display on the monitor—a subject we will get into in Chapter 10, but not today.

Figure 5.16 is a flowchart of the algorithm that solves this problem. Note that each step is expressed both in English and also (in parentheses) in terms of an LC-3 implementation.

The first step is (as always) to initialize all the variables. This means providing starting values (called *initial values*) for R0, R1, R2, and R3, the four registers the computer will use to execute the program that will solve the problem. R2 will keep track of the number of occurrences; in Figure 5.16, it is referred to as *count*. It is initialized to zero. R3 will point to the next character in the file that is being examined. We refer to it as *pointer* since it contains the **address** of the location where the next character of the file that we wish to examine resides. The pointer is initialized with the address of the **first** character in the file. R0 will hold the character that is being counted; we will input that character from the keyboard and put it in R0. R1 will hold, in turn, each character that we get from the file being examined.

We should also note that there is no requirement that the file we are examining be close to or far away from the program we are developing. For example, it is perfectly reasonable for the program we are developing to start at x3000, and the file we are examining to start at x9000. If that were the case, in the initialization process, R3 would be initialized to x9000.

The next step is to count the number of occurrences of the input character. This is done by processing, in turn, each character in the file being examined, until the file is exhausted. Processing each character requires one iteration of a loop. Recall from Section 5.4.3 that there are two common methods for keeping track of iterations of a loop. We will use the sentinel method, using the ASCII code for EOT (End of Text) (00000100) as the sentinel. A table of ASCII codes is in Appendix E.

In each iteration of the loop, the contents of R1 are first compared to the ASCII code for EOT. If they are equal, the loop is exited, and the program moves on to the final step, displaying on the screen the number of occurrences. If not, there is work to do. R1 (the current character under examination) is compared to R0 (the character input from the keyboard). If they match, R2 is incremented. In either case, we get the next character, that is, R3 is incremented, the next character is loaded into R1, and the program returns to the test that checks for the sentinel at the end of the file.

When the end of the file is reached, all the characters have been examined, and the count is contained as a binary number in R2. In order to display the

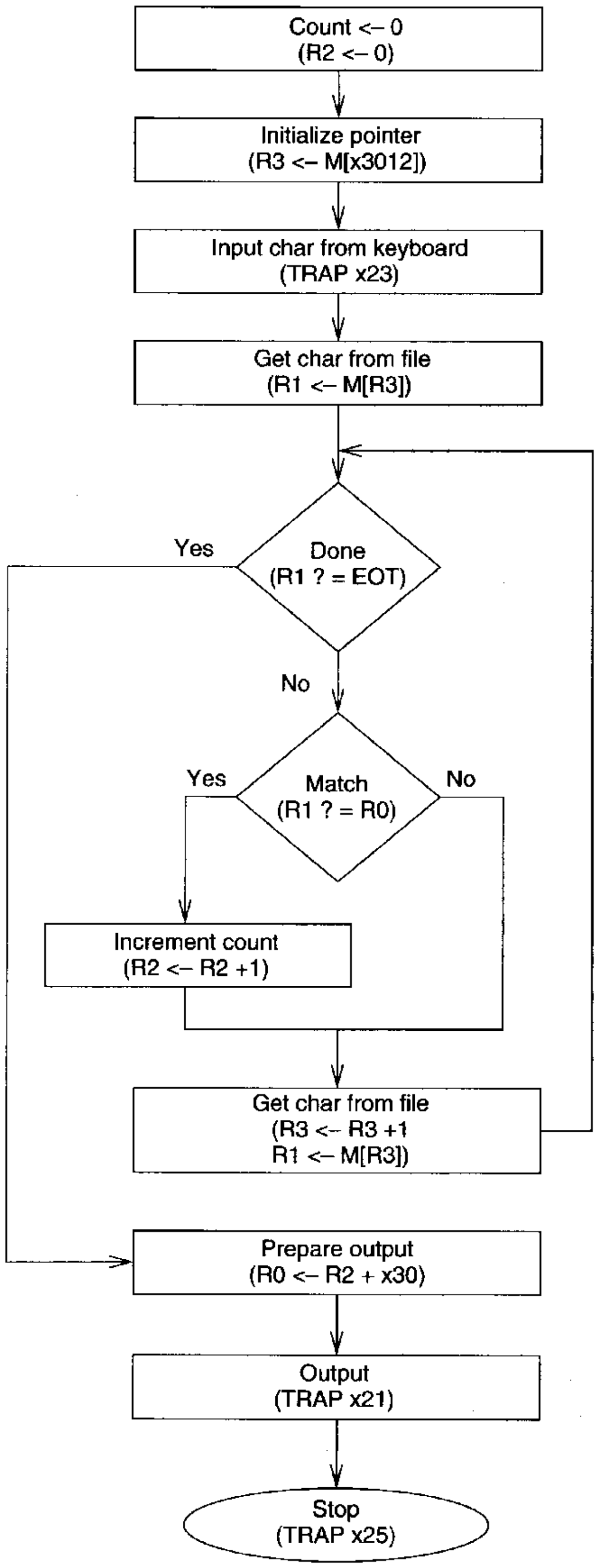

Figure 5.16 An algorithm to count occurrences of a character

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	**R2 <- 0**
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	**R3 <- M[x3012]**
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	**TRAP x23**
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	**R1 <- M[R3]**
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0	**R4 <- R1-4**
x3005	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	**BRz x300E**
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	**R1 <- NOT R1**
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	**R1 <- R1 + 1**
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	**R1 <- R1 + R0**
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	**BRnp x300B**
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	**R2 <- R2 + 1**
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	**R3 <- R3 + 1**
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	**R1 <- M[R3]**
x300D	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	0	**BRnzp x3004**
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	**R0 <- M[x3013]**
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	**R0 <- R0 + R2**
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	**TRAP x21**
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	**TRAP x25**
x3012	Starting address of file																
x3013	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	**ASCII TEMPLATE**

Figure 5.17 A machine language program that implements the algorithm of Figure 5.16

count on the monitor, it is necessary to first convert it to an ASCII code. Since we have assumed the count is less than 10, we can do this by putting a leading 0011 in front of the 4-bit binary representation of the count. Note in Figure E.2 the relationship between the binary value of each decimal digit between 0 and 9 and its corresponding ASCII code. Finally, the count is output to the monitor, and the program terminates.

Figure 5.17 is a machine language program that implements the flowchart of Figure 5.16.

First the initialization steps. The instruction at x3000 clears R2 by ANDing it with x0000; the instruction at x3001 loads the value stored in x3012 into R3. This is the address of the first character in the file that is to be examined for occurrences of our character. Again, we note that this file can be anywhere in memory. Prior to starting execution at x3000, some sequence of instructions must have stored the first address of this file in x3012. Location x3002 contains the TRAP instruction, which requests the operating system to perform a service call on behalf of this program. The function requested, as identified by the 8-bit trapvector 00100011 (or, x23), is to input a character from the keyboard and load it into R0. Table A.2 lists trapvectors for all operating system service calls that can be performed on behalf of a user program. Note (from Table A.2) that x23 directs the operating system to perform the service call that reads the next character struck and loads it into R0. The instruction at x3003 loads the character pointed to by R3 into R1.

Then the process of examining characters begins. We start (x3004) by subtracting 4 (the ASCII code for EOT) from R1, and storing it in R4. If the result is zero, the end of the file has been reached, and it is time to output the count. The instruction at x3005 conditionally branches to x300E, where the process of outputting the count begins.

If R4 is not equal to zero, the character in R1 is legitimate and must be examined. The sequence of instructions at locations x3006, x3007, and x3008 determine if the contents of R1 and R0 are identical. The sequence of instructions perform the following operation:

$$R0 + (NOT\ (R1) + 1)$$

This produces all zeros only if the bit patterns of R1 and R0 are identical. If the bit patterns are not identical, the conditional branch at x3009 branches to x300B, that is, it skips the instruction x300A, which increments R2, the counter.

The instruction at x300B increments R3, so it will point to the next character in the file being examined, the instruction at x300C loads that character into R1, and the instruction at x300D unconditionally takes us back to x3004 to start processing that character.

When the sentinel (EOT) is finally detected, the process of outputting the count begins (at x300E). The instruction at x300E loads 00110000 into R0, and the instruction at x300F adds the count to R0. This converts the binary representation of the count (in R2) to the ASCII representation of the count (in R0). The instruction at x3010 invokes a TRAP to the operating system to output the contents of R0 on the monitor. When that is done and the program resumes execution, the instruction at x3011 invokes a TRAP instruction to terminate the program.

5.6 The Data Path Revisited

Before we leave Chapter 5, let us revisit the data path diagram that we first encountered in Chapter 3 (Figure 3.33). Now we are ready to examine all the structures that are needed to implement the LC-3 ISA. Many of them we have seen earlier in this chapter in Figures 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, and 5.11. We reproduce this diagram as Figure 5.18. Note at the outset that there are two kinds of arrows in the data path, those with arrowheads filled in, and those with arrowheads not filled in. Filled-in arrowheads designate information that is processed. Unfilled-in arrowheads designate control signals. Control signals emanate from the block labeled "Control." The connections from Control to most control signals have been left off Figure 5.18 to reduce unnecessary clutter in the diagram.

5.6.1 Basic Components of the Data Path

The Global Bus

You undoubtedly first notice the heavy black structure with arrowheads at both ends. This represents the data path's global bus. The LC-3 global bus consists of 16 wires and associated electronics. It allows one structure to transfer up to 16 bits of information to another structure by making the necessary electronic connections on the bus. Exactly one value can be transferred on the bus at one time. Note that each structure that supplies values to the bus has a triangle just

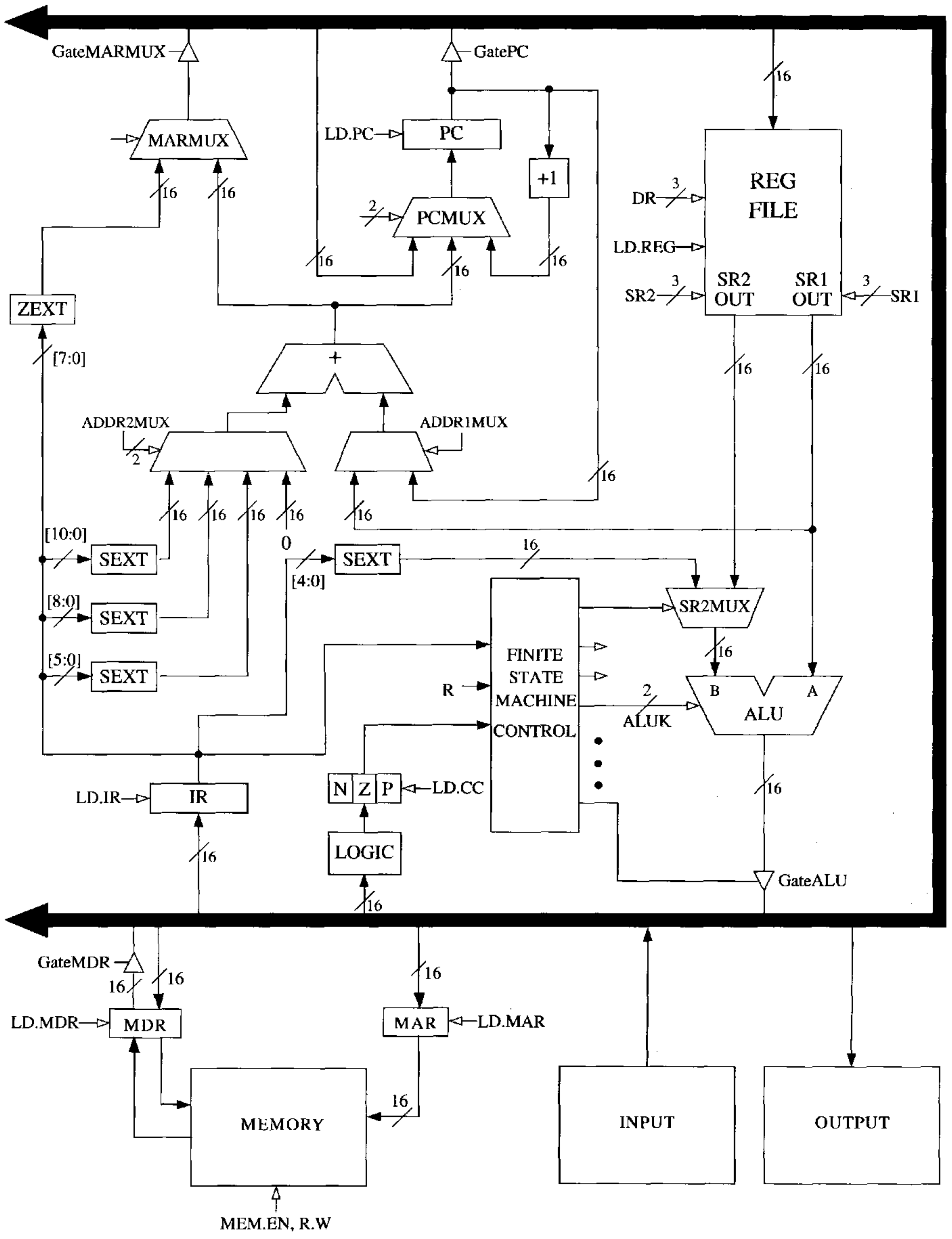

Figure 5.18 The data path of the LC-3

behind its input arrow to the bus. This triangle (called a *tri-state device*) allows the computer's control logic to enable exactly one supplier to provide information to the bus at any one time. The structure wishing to obtain the value being supplied can do so by asserting its LD.x (load enable) signal (recall our discussion of gated latches in Section 3.4.2). Not all computers have a single global bus. The pros and cons of a single global bus is yet another one of those topics that will have to wait for later in your education.

Memory

One of the most important parts of any computer is the memory that contains both instructions and data. Memory is accessed by loading the memory address register (MAR) with the address of the location to be accessed. If a load is being performed, control signals then read the memory, and the result of that read is delivered by the memory to the memory data register (MDR). On the other hand, if a store is being performed, the data to be stored is first loaded into the MDR. Then the control signals specify that WE is asserted in order to store into that memory location.

The ALU and the Register File

The ALU is the processing element. It has two inputs, source 1 from a register and source 2 from either a register or the sign-extended immediate value provided by the instruction. The registers (R0 through R7) can provide two values, source 1, which is controlled by the 3-bit register number SR1, and source 2, which is controlled by the 3-bit register number SR2. SR1 and SR2 are fields in the LC-3 operate instruction. The selection of a second register operand or a sign-extended immediate operand is determined by bit [5] of the LC-3 instruction. Note the mux that provides source 2 to the ALU. The select line of that mux, coming from the control logic, is bit [5] of the LC-3 operate instruction.

The result of an ALU operation is a result that is stored in one of the registers, and the three single-bit condition codes. Note that the ALU can supply 16 bits to the bus, and that value can then be written into the register specified by the 3-bit register number DR. Also, note that the 16 bits supplied to the bus are also input to logic that determines whether that 16-bit quantity is negative, zero, or positive, and sets the three registers N, Z, and P accordingly.

The PC and the PCMUX

The PC supplies via the global bus to the MAR the address of the instruction to be fetched at the start of the instruction cycle. The PC, in turn, is supplied via the three-to-one PCMUX, depending on the instruction being executed. During the FETCH phase of the instruction cycle, the PC is incremented and written into the PC. That is shown as the rightmost input to the PCMUX.

If the current instruction is a control instruction, then the relevant source of the PCMUX depends on which control instruction is currently being processed. If the current instruction is a conditional branch and the branch is taken, then the PC is loaded with the incremented PC + PCoffset (the 16-bit value obtained by sign-extending IR[8:0]). Note that this addition takes place in the special adder and not in the ALU. The output of the adder is the middle input to PCMUX. The third input to PCMUX is obtained from the global bus. Its use will become clear after we discuss the other control instructions in Chapters 9 and 10.

The MARMUX

As you know, memory is accessed by supplying the address to the MAR. The MARMUX controls which of two sources will supply the MAR with the appropriate address during the execution of a load, a store, or a TRAP instruction. The right input to the MARMUX is obtained by adding either the incremented PC or a base register to a literal value or zero supplied by the IR. Whether the PC or a base register and what literal value depends on which opcode is being processed. The control signal ADDR1MUX specifies the PC or base register. The control signal ADDR2MUX specifies which of four values to be added. The left input to MARMUX provides the zero-extended trapvector, which is needed to invoke service calls, as will be discussed in further detail in Chapter 9.

5.6.2 The Instruction Cycle

We complete our tour of the LC-3 data path by following the flow through an instruction cycle. Suppose the content of the PC is x3456 and the content of location x3456 is 0110011010000100. And suppose the LC-3 has just completed processing the instruction at x3455, which happened to be an ADD instruction.

FETCH

As you know, the instruction cycle starts with the FETCH phase. That is, the instruction is obtained by accessing memory with the address contained in the PC. In the first cycle, the contents of the PC are loaded via the global bus into the MAR, and the PC is incremented and loaded into the PC. At the end of this cycle, the PC contains x3457. In the next cycle (if memory can provide information in one cycle), the memory is read, and the instruction 0110011010000100 is loaded into the MDR. In the next cycle, the contents of the MDR are loaded into the instruction register (IR), completing the FETCH phase.

DECODE

In the next cycle, the contents of the IR are decoded, resulting in the control logic providing the correct control signals (unfilled arrowheads) to control the processing of the rest of this instruction. The opcode is 0110, identifying the LDR instruction. This means that the Base+offset addressing mode is to be used to determine the address of data to be loaded into the destination register R3.

EVALUATE ADDRESS

In the next cycle, the contents of R2 (the base register) and the sign-extended bits [5:0] of the IR are added and supplied via the MARMUX to the MAR. The SR1 field specifies 010, the register to be read to obtain the base address. ADDR1MUX selects SR1OUT, and ADDR2MUX selects the second from the right source.

OPERAND FETCH

In the next cycle (or more than one, if memory access takes more than one cycle), the data at that address is loaded into the MDR.

EXECUTE

The LDR instruction does not require an EXECUTE phase, so this phase takes zero cycles.

STORE RESULT

In the last cycle, the contents of the MDR are loaded into R3. The DR control field specifies 011, the register to be loaded.

Exercises

5.1 Given instructions ADD, JMP, LEA, and NOT, identify whether the instructions are operate instructions, data movement instructions, or control instructions. For each instruction, list the addressing modes that can be used with the instruction.

5.2 A memory's addressibility is 64 bits. What does that tell you about the size of the MAR and MDR?

5.3 There are two common ways to terminate a loop. One way uses a counter to keep track of the number of iterations. The other way uses an element called a ____. What is the distinguishing characteristic of this element?

5.4 Say we have a memory consisting of 256 locations, and each location contains 16 bits.

a. How many bits are required for the address?
b. If we use the PC-relative addressing mode, and want to allow control transfer between instructions 20 locations away, how many bits of a branch instruction are needed to specify the PC-relative offset?
c. If a control instruction is in location 3, what is the PC-relative offset of address 10. Assume that the control transfer instructions work the same way as in the LC-3.

5.5 *a.* What is an addressing mode?
b. Name three places an instruction's operands might be located.
c. List the five addressing modes of the LC-3, and for each one state where the operand is located (from part b).
d. What addressing mode is used by the ADD instruction shown in Section 5.1.2?