chapter

8

I/O

Up to now, we have paid little attention to input/output (I/O). We did note (in Chapter 4) that input/output is an important component of the von Neumann model. There must be a way to get information into the computer in order to process it, and there must be a way to get the result of that processing out of the computer so humans can use it. Figure 4.1 depicts a number of different input and output devices.

We suggested (in Chapter 5) that input and output can be accomplished by executing the TRAP instruction, which asks the operating system to do it for us. Figure 5.17 illustrates this for input (at address x3002) and for output (at address x3010).

In this chapter, we are ready to do I/O by ourselves. We have chosen to study the keyboard as our input device and the monitor display as our output device. Not only are they the simplest I/O devices and the ones most familiar to us, but they have characteristics that allow us to study important concepts about I/O without getting bogged down in unnecessary detail.

8.1 I/O Basics

8.1.1 Device Registers

Although we often think of an I/O device as a single entity, interaction with a single I/O device usually means interacting with more than one *device register*. The simplest I/O devices usually have at least two device 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 still busy processing the most recent I/O task.

8.1.2 Memory-Mapped I/O versus Special Input/Output 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, light years ago—1965) is an example of a computer that used special input and output instructions. The 12-bit PDP-8 instruction contained a 3-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, in which the source address is that of an input device register, is an input instruction. Similarly, a store instruction 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. Addresses x0000 to xFDFF are allocated to 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 sales of LC-3 microprocessors may require the expansion of device register address assignments as new and exciting applications emerge!

8.1.3 Asynchronous versus 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 300 MHz. Even for a microprocessor operating at only 300 MHz, a clock cycle lasts only 3.3 nanoseconds. Suppose a processor executed one instruction at a time, and it took the processor 10 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 33 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? Assume the average word length is six characters. See Exercise 8.3.

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 30 million cycles. This would require a typing speed of 100 words/minute, 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. This latter characteristic we call *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 display. In the case of the keyboard, we will need a 1-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 1-bit status register to indicate whether or not the most recent character sent to the monitor has been displayed.

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 300 million characters/second. Each time the typist types a character, the Ready bit is set. 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 (for example, one character every 30 million cycles), then we would not need the Ready bit. The computer would simply know, after 30 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,* or the input activity was synchronous.

8.1.4 Interrupt-Driven versus 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* versus *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 is called *polling,* since the Ready bit is polled by the processor, asking if any key has been struck.

Section 8.2.2 describes how the polling method works. Section 8.5 explains interrupt-driven I/O.

8.2 Input from the Keyboard

8.2.1 Basic Input Registers (the KBDR and the 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, KBDR is assigned to xFE02; KBSR is assigned to xFE00.

Even though a character needs only eight 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. 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 8.1 shows the two device registers needed by the keyboard.

8.2.2 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

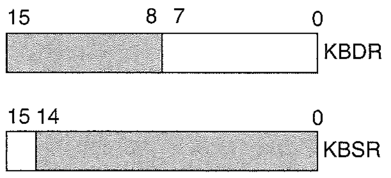

Figure 8.1 Keyboard device registers

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.

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 to 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.

8.2.3 Implementation of Memory-Mapped Input

Figure 8.2 shows the additional data path required to implement memory-mapped input. You are already familiar, from Chapter 5, with the data path required to

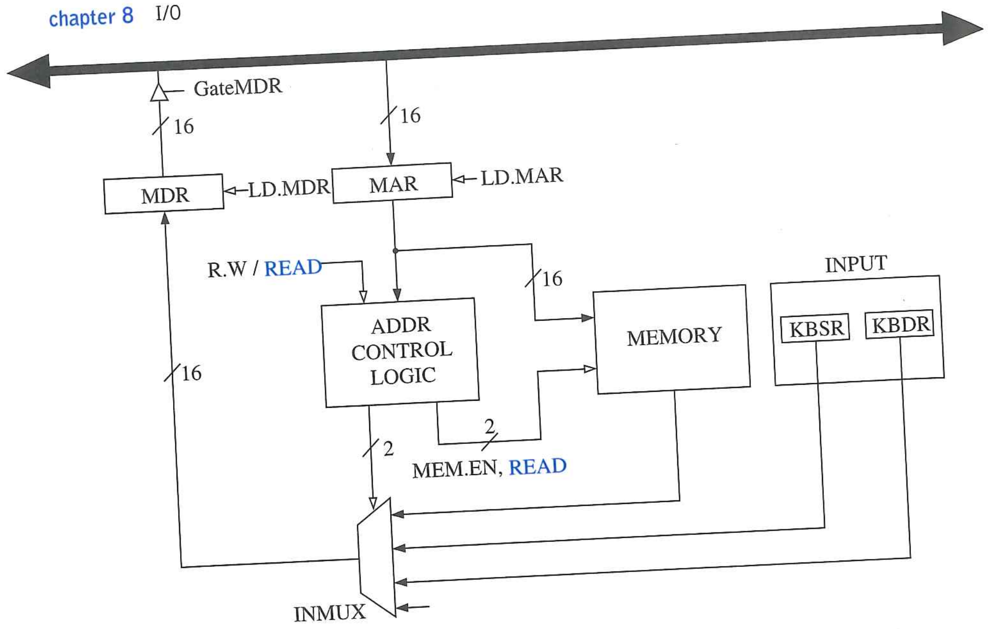

Figure 8.2 Memory-mapped input

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 set of 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.

8.3 Output to the Monitor

8.3.1 Basic Output Registers (the DDR and the 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.

As is the case with input, even though an output character needs only eight 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

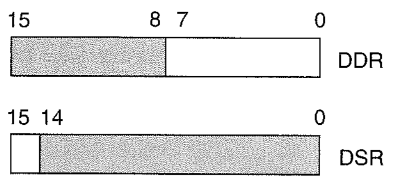

Figure 8.3 Monitor device registers

case of DSR, bit [15] contains the synchronization mechanism, that is, the Ready bit. Figure 8.3 shows the two device registers needed by the monitor.

8.3.2 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), then 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 8.2.2, lines 01 and 02 repeatedly poll DSR[15] to see if the monitor electronics is finished yet 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.

8.3.3 Implementation of Memory-Mapped Output

Figure 8.4 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 8.3.2 that before the DDR could be loaded, the Ready

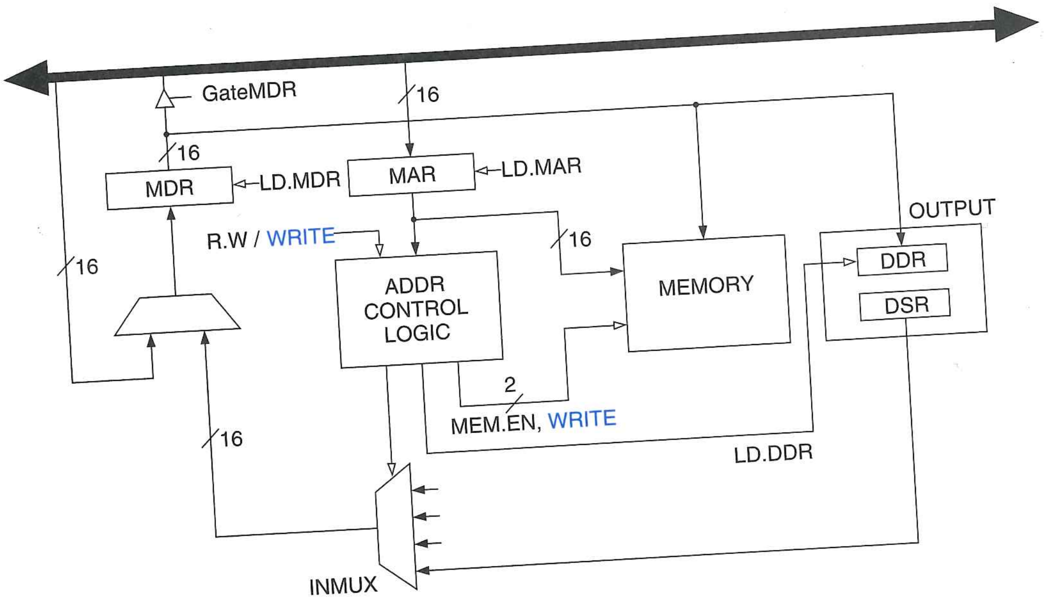

Figure 8.4 Memory-mapped output

bit had to be in state 1, indicating that the previous character had already been 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.

8.3.4 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 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
```

8.4 A More Sophisticated Input Routine

In the example of Section 8.2.2, 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 symbol that is displayed by your operating system (for example, % or **C:**) or by your editor (for example, **:**) are examples of prompts.

The program fragment shown in Figure 8.5 obtains keyboard input via polling as we have shown in Section 8.2.2 already. 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 in parts.

You are already familiar with lines 13 through 19 and lines 25 through 28, which correspond to the code in Section 8.3.4 for inputting a character via the

```
01  START    ST      R1,SaveR1    ; Save registers needed
02           ST      R2,SaveR2    ; by this routine
03           ST      R3,SaveR3
04  ;
05           LD      R2,Newline
06  L1       LDI     R3,DSR
07           BRzp    L1           ; Loop until monitor is ready
08           STI     R2,DDR       ; Move cursor to new clean line
09  ;
0A           LEA     R1,Prompt    ; Starting address of prompt string
0B  Loop     LDR     R0,R1,#0     ; Write the input prompt
0C           BRz     Input        ; End of prompt string
0D  L2       LDI     R3,DSR
0E           BRzp    L2           ; Loop until monitor is ready
0F           STI     R0,DDR       ; Write next prompt character
10           ADD     R1,R1,#1     ; Increment prompt pointer
11           BRnzp   Loop         ; Get next prompt character
12  ;
13  Input    LDI     R3,KBSR
14           BRzp    Input        ; Poll until a character is typed
15           LDI     R0,KBDR      ; Load input character into R0
16  L3       LDI     R3,DSR
17           BRzp    L3           ; Loop until monitor is ready
18           STI     R0,DDR       ; Echo input character
19  ;
1A  L4       LDI     R3,DSR
1B           BRzp    L4           ; Loop until monitor is ready
1C           STI     R2,DDR       ; Move cursor to new clean line
1D           LD      R1,SaveR1    ; Restore registers
1E           LD      R2,SaveR2    ; to original values
1F           LD      R3,SaveR3
20           BRnzp   NEXT_TASK    ; Do the program's next task
21  ;
22  SaveR1   .BKLW   1            ; Memory for registers saved
23  SaveR2   .BKLW   1
24  SaveR3   .BKLW   1
25  DSR      .FILL   xFE04
26  DDR      .FILL   xFE06
27  KBSR     .FILL   xFE00
28  KBDR     .FILL   xFE02
29  Newline  .FILL   x000A        ; ASCII code for newline
2A  Prompt   .STRINGZ ``Input a character>''
```

Figure 8.5 The input routine for the LC-3 keyboard

keyboard and echoing it on the monitor. Lines 01 through 03, lines 1D through 1F, and lines 22 through 24 recognize that this input routine needs to use general purpose registers R1, R2, and R3. Unfortunately, they most likely contain values that will still be needed after this routine has finished. To prevent the loss of those values, the ST instructions in lines 01 through 03 save them in memory locations SaveR1, SaveR2, and SaveR3, before the input routine starts its business. These three memory locations have been allocated by the .BLKW pseudo-ops in lines 22 through 24. After the input routine is finished and before the program branches unconditionally to its NEXT_TASK (line 20), the LD instructions in lines 1D through 1F restore the original values saved to their rightful locations in R1, R2, and R3.

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 x0A is written to DDR for outputting (line 8).

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 program knows that the entire input prompt has been written to the screen and branches to the code that handles the actual keyboard input (starting at line 13).

After the person at the keyboard has typed 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.

8.5 Interrupt-Driven I/O

In Section 8.1.4, 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 8.2, 8.3, and 8.4, 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 it 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.

8.5.1 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 that

```
      .
      .
      .
  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
2: The needs of the I/O device start being carried out
2: The needs of the I/O device are being carried out
2: The needs of the I/O device are being carried out
2: The needs of the I/O device are being carried out
2: The needs of the I/O device have been carried out
3: Program A is brought back to life
  Program A is executing instruction n+3
  Program A is executing instruction n+4
      .
      .
      .
```

Figure 8.6 Instruction execution flow for interrupt-driven I/O

program to stop, (2) have the processor carry 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 8.6.

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
      .
      .
      .
```

8.5.2 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 8.1

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 1,000 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!)

We could obtain each character input by polling, as in Section 8.2. If we did, we would waste a lot of time waiting for the "next" character to be typed. It would take $100 \cdot 0.125$ 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 10-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 1,000 sequences that is the difference between 7 hours and $3\frac{1}{2}$ hours.

8.5.3 Generation of the Interrupt Signal

There are two parts to interrupt-driven I/O, (1) the enabling mechanism that allows an I/O device to interrupt the processor when it has input to deliver or is ready to accept output, and (2) the mechanism that manages the transfer of the I/O data. The two parts can be briefly described as:

1. generating the interrupt signal, which stops the currently executing process, and
2. handling the request demanded by this signal.

The first part we will study momentarily. We will examine the various things that must come together to force the processor to stop what it is doing and pay attention to the interrupt request.

The second part, unfortunately, we will have to put off until Section 10.2. To handle interrupt requests, the LC-3 uses a stack, and we will not get to stacks until Chapter 10.

Now, then, part 1. Several things must be true for an I/O device to actually interrupt the processor:

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 and takes care of the interrupt.

The Interrupt Signal from the Device

For an I/O device to generate an interrupt request, the first two elements in the previous list must be true: The device must want service, and it must have the right to request that service.

The first element we have discussed at length 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 second element is an interrupt enable bit, which can be set or cleared by the processor, 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 8.7, the IE bit is bit [14]. The **interrupt request from the I/O device** is the logical AND of the IE bit and the Ready bit, as is also shown in Figure 8.7.

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. 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

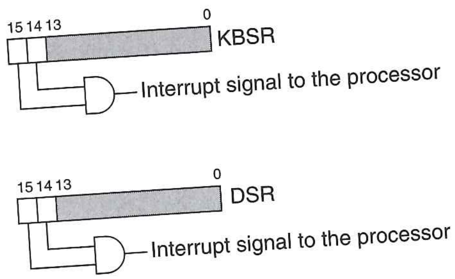
15 14 13
0
KBSR
Interrupt signal to the processor
15 14 13
0
DSR
Interrupt signal to the processor

Figure 8.7 Interrupt enable bits and their use

last character), bit [15] is set. This, in turn, asserts the output of the AND gate, causing an interrupt request to be generated from the I/O device.

The Importance of Priority

The third element in the list of things that must be true for an I/O device to actually interrupt the processor is whether the request is sufficiently urgent. Every instruction that the processor executes, it does with a stated level of urgency. The term we give for the urgency of execution is *priority.*

We say that a program is being executed at a specified priority level. Almost all computers have a set of priority levels that programs can run at. The LC-3 has eight priority levels, PL0, .. 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 seeks access to the computer, the lower-priority program suspends processing until the higher-PL program executes and satisfies that 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.

We will see momentarily that the processor will stop executing its current program and service an interrupt request if the INT signal is asserted. Figure 8.8 shows what is required to assert the INT signal and where the notion of priority level comes into play. Figure 8.8 shows the status registers of several devices operating at various priority levels. 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 and the executing program is stopped.

The Test for INT

The final step in the first part of interrupt-driven I/O is the test to see if the processor should stop and handle an interrupt. Recall from Chapter 4 that the instruction cycle sequences through the six phases of FETCH, DECODE, EVALUATE ADDRESS, FETCH OPERAND, EXECUTE, and STORE RESULT. Recall further that after the sixth phase, the control unit returns to the first phase, that is, the FETCH of the next instruction.

The additional logic to test for the interrupt signal is to replace that last sequential step of **always** going from STORE RESULT back to FETCH, as follows: The STORE RESULT phase is instead accompanied by a test for the interrupt signal INT. If INT is not asserted, then it is business as usual, with the control unit

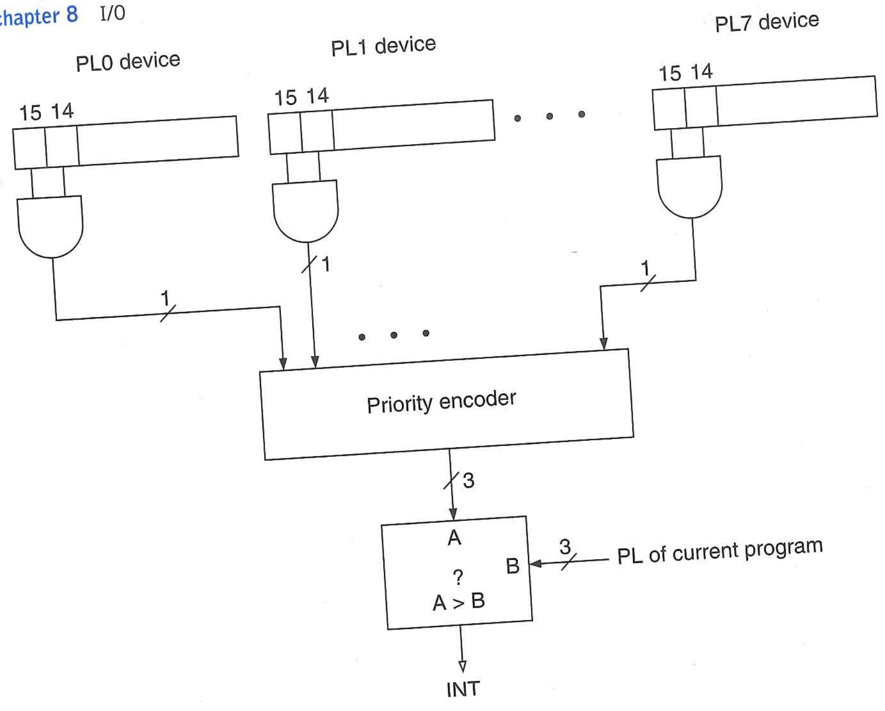

Figure 8.8 Generation of the INT signal

returning to the FETCH phase to start processing the next instruction. If INT is asserted, then the control unit does two things before returning to the FETCH phase. First it saves enough state information to be able to return to the interrupted program where it left off. Second it loads the PC with the starting address of the program that is to carry out the requirements of the I/O device. How it does that is the topic of Section 10.2, which we will study after we learn how stacks work.

8.6 Implementation of Memory-Mapped I/O, Revisited

We showed in Figures 8.2 and 8.4 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 8.9 (reproduced from 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 provides the control signals to direct the transfer of data between the MDR and the memory or I/O registers.

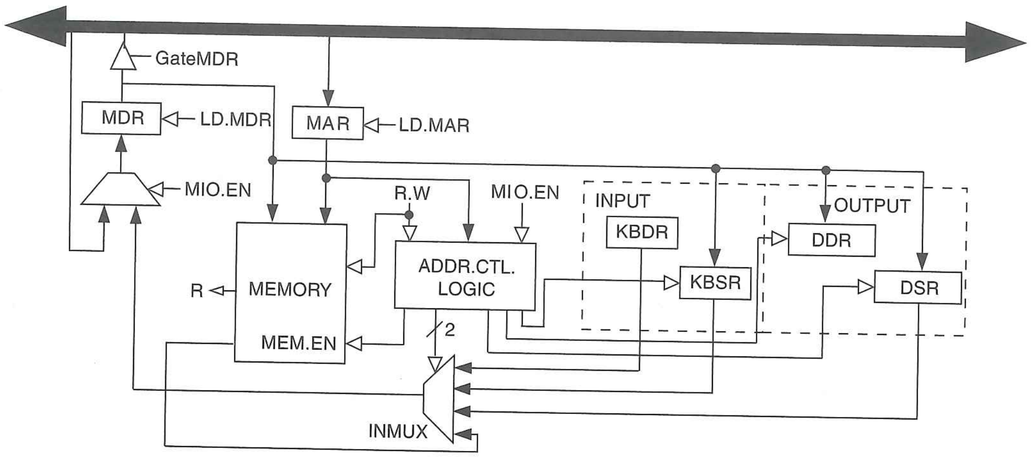

Figure 8.9 Partial data path implementation of memory-mapped I/O

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 are written either to memory or to one of the device registers. The Address Control Logic either enables a write to memory or it asserts the load enable line of the device register specified by the contents of the MAR.

Exercises

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

8.2 Why is a Ready bit not needed if synchronous I/O is used?

8.3 In Section 8.1.3, 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?

8.4 Are the following interactions usually synchronous or asynchronous?
a. Between a remote control and a television set
b. Between the mailcarrier 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?

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

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

8.7 Which of the following combinations describe the system described in Section 8.2.2?

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

8.8 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.

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

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

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

8.12 Adam H. decided to design a variant of the LC-3 that did not need a keyboard status register. Instead, he created a readable/writable keyboard data and status register (KBDSR), which contains the same data as the KBDR. With the KBDSR, a program requiring keyboard input would wait until a nonzero value appeared in the KBDSR. The nonzero value would be the ASCII value of the last key press. Then the program would write a zero into the KBDSR indicating that it had read the key press. Modify the basic input service of Section 8.2.2 to implement Adam's scheme.

8.13 Some computer engineering students decided to revise the LC-3 for their senior project. In designing the LC-4, they decided to conserve on device registers by combining the KBSR and the DSR into one status register: the IOSR (the input/output status register). IOSR[15] is the keyboard device Ready bit and IOSR[14] is the display device Ready bit. What are the implications for programs wishing to do I/O? Is this a poor design decision?

8.14 An LC-3 Load instruction specifies the address xFE02. How do we know whether to load from the KBDR or from memory location xFE02?

8.15 Interrupt-driven I/O:

a. What does the following LC-3 program do?

```
        .ORIG   x3000
        LD      R3, A
        STI     R3, KBSR
AGAIN   LD      R0, B
        TRAP    x21
        BRnzp   AGAIN
A      .FILL    x4000
B      .FILL    x0032
KBSR   .FILL    xFE00
       .END
```

b. If someone strikes a key, the program will be interrupted and the keyboard interrupt service routine will be executed as shown below. What does the keyboard interrupt service routine do?

```
        .ORIG   x1000
        LDI     R0, KBDR
        TRAP    x21
        TRAP    x21
        RTI
KBDR    .FILL   xFE02
        .END
```

NOTE: RTI will be studied in chapter 10.

c. Finally, suppose the program of part *a* started executing, and someone sitting at the keyboard struck a key. What would you see on the screen?

8.16 What does the following LC-3 program do?

```
        .ORIG   x3000
        LD      R0,ASCII
        LD      R1,NEG
AGAIN   LDI     R2,DSR
        BRzp    AGAIN
        STI     R0,DDR
        ADD     R0,R0,#1
        ADD     R2,R0,R1
        BRnp    AGAIN
        HALT
ASCII   .FILL   x0041
NEG     .FILL   xFFB6    ; -x004A
DSR     .FILL   xFE04
DDR     .FILL   xFE06
        .END
```