Overview of Input/Output Mechanisms

Norman Matloff
University of California, Davis
\copyrigth{2001}, N. Matloff

February 5, 2001

Contents

1 Introduction

During the early days of computers, input/output (I/O) was of very little importance. Instead computers, as their name implied, were used to **compute.** A typical application might involve huge mathematical calculations needed for some problem in physics. The I/O done in such an application would be quite minimal, with the machine grinding away for hours at a time with little or no I/O being done.

Today, we have just the opposite situation. I/O plays a crucial role in most modern computer applications. In fact, applications in which I/O plays an important role can arguably be viewed as the most powerful "driving force" behind the revolutionary rise to prominence of computers in the last decade or so. Today we make use of computers in numerous aspects of our business and personal lives–and in the vast majority of such uses, the programs center around I/O operations.

In some of these applications, the use of a computer is fairly obvious: First and foremost, the Internet, but also credit-card billing, airline reservations, word processors, spreadsheets, automatic teller machines, real estate databases, drafting programs for architects, video games, and so on. Possibly less obvious is the existence of computers in automobiles, and in such household items as washing machines and autofocus cameras. However, whether the existence of computers in these applications is obvious or hidden, **the common theme behind them is that they all are running programs in which I/O plays an important role, usually the dominant role.**

2 Our Mythical Machine Architecture

In this tutorial, you will learn about I/O on a mythical machine described in the Appendix below. However, **the principles are similar for almost any machine.**

3 I/O Ports and Device Structure

An I/O device connects to the system bus through **ports**. I/O ports are similar in structure to CPU registers. However, a port is not in the CPU, being instead located between the I/O device and the system bus.

Ports are usually 8 bits in width. They have addresses, just like words in memory do, and these addresses are given meaning through the address bus, just as is the case for words in memory.

4 Program Access to I/O Ports

Note that a potential ambiguity arises. Suppose we wish to communicate with port 50, which is connected to some I/O device. The CPU will place the value 50 on the address bus. Since all items which are connected to the bus will see what is on the address bus, how can they distinguish this 50, intended for an I/O port, from a value of 50 meant to address word 50 of memory? In other words, how can the CPU indicate that it wants to address I/O port 50 rather than memory word 50? There are two fundamental systems for handling this problem, described next.

4.1 I/O Address Space Approach

Recall that the system bus of a computer consists of an address bus, a data bus, and a control bus. The control bus in many computers, such as those based on Intel CPUs, has a special line or lines to indicate that the current value on the address bus is meant to indicate an I/O port, rather than a word in memory. For our mythical computer here we will assume four such lines, MR (memory read), MW (memory write), IOR (I/O read) and IOW (I/O write).

To access I/O port 50, the CPU will assert either IOR or IOW (depending on whether it wants to read from or write to that port), while to access word 50 in memory, the CPU will assert either MR or MW. In either case the CPU will place the value 50 on the address bus, but the values it simultaneously places on these lines in the control bus will distinguish between I/O port 50 and word 50 of memory.

The programmer himself or herself controls which of these lines will be asserted, by choosing which instruction to use.

```
MOV R2,[50]
```

and

```
IN R5,[50]
```

will both cause the value 0x50 to go out onto the address bus, and in both cases the response will be sent back via the data bus. (Here R2 and R5 are CPU registers.) But the MOV will assert the MR line in the control bus, while the IN instruction will assert the IOR line in that bus. As a result, the MOV will result in copying the memory byte at address 50 to R2, while the IN will copy the byte from I/O port 50 to R5.

Since the lines IOR/IOW and MR/MW allow us to distinguish between I/O ports and memory words having identical addresses, we can describe this by saying that the I/O ports have a separate **address space** from that of memory. Thus we will call this the **I/O address space approach**.

In our machine the keyboard has ports with addresses 60 and 61. For example, the instruction

IN R3,[60]

would copy a character from the keyboard's data port, which has address 60, to the R3 register.

Just as with memory chips, the addresses of I/O ports are determined by combinations of gates whose outputs feed into something like a Chip Select pin in a given port.[1] Denoting the Chip Select pin by CS, we would have in the case of the keyboard port at address 52

$$CS = \overline{A7}A6\overline{A5}A4\overline{A3A2}A1\overline{A0}$$

4.2 Memory-Mapped I/O Approach

Another approach to the "duplicate port/memory address" problem described above is called **memory-mapped I/O."**[2]

[1]Or more typically, a set of ports. The keyboard ports in our example here would typically be mounted on one common I/O board, providing a common interface for them to the bus.

[2]Unfortunately, many terms in the computer field are "overloaded." It has become common on personal computers to refer to the mapping of video monitor pixels to memory addresses also as "memory-mapped I/O." However, this is not the original use of the term, and should not be confused with what we are discussing here.

Under this approach, there are no special lines in the control bus to indicate I/O access versus memory access (though there still must be one or two lines to distinguish a read from a write, of course), and thus no special I/O instructions such as IN and OUT. One simply uses ordinary instructions such as MOV, whether one is accessing I/O ports or memory.

Note that we can use the memory-mapped approach even if we do have special I/O instructions and control lines, just by ignoring them. In our machine here, for instance, we retain the addresses 60 and 61 for the keyboard ports but connect their read/write pins to the MR and MW lines instead of IOR and IOW. To read a character from the Keyboard Data Port 60 and copy it to a memory location CHARBUFFER:

```
MOV CHARBUFFER,[60]
```

Since this is an ordinary MOV instruction, the CPU is being “fooled” into thinking that the keyboard data port is physically a memory location, which the MOV is reading. But of course it is not memory.

One advantage of the memory-mapped approach to I/O port addressing is that one can easily access the ports from a high-level language such as C directly, instead of using assembly language, by taking advantage of the fact that the ***** operator in C allows access to specific memory locations. For example, the above operation could be done in a C program by having the declaration

```
char CharBuffer,*KDataPtr;
```

and then using the code

```
KDataPtr = 0x60;
CharBuffer = *KDataPtr;
```

We could not do this directly with the I/O address space approach, because we have no way in the C language to make the compiler produce IN or OUT instructions. (The above code would produce a MOV.) However, compilers on machines which use the I/O address space approach usually include library functions which can be called from C to do these operations; the authors of these functions wrote them in assembly language.

5 I/O Programming

Consider the case of reading a character from the keyboard. The program which will read the character has no idea when the user will hit a key on the keyboard. How should the program be written to deal with this time problem? (Note that we are for the moment discussing programs which directly access the keyboard, rather than going through the operating system, as is the case with programs you usually write; more on this point later.) There are three basic approaches, covered in the following subsections.

5.1 Wait-Loop I/O

As described in the Appendix to this document, Bit 0 in our machine's Keyboard Status Port (KSP) tells us whether a character is ready to read or not; a value of 1 in this bit means a character is ready, and 0 means the user has not yet typed a character. **Wait-loop I/O** then consists of writing a loop, in which the program keeps testing Bit 0 of the KSP:

```
LP: IN R4,[61]   ; get value of KSP
       AND R4,1     ; check Bit 0
       JZ LP        ; if that bit is 0, try again
 DONE: IN R4,[60]   ; finally, get the character
```

During the time before the user hits a key, the value read from the KSP will always have a 0 in Bit 0, so that the AND results in 0, and we jump back to LP. But eventually the user will hit a key, resulting in Bit 0 of the KSP being 1, and we leave the loop. We then pick up the character at DONE.

5.2 Interrupt-Driven I/O

Wait-loop I/O is very wasteful. Usually the speed of I/O device is very slow compared to CPU speed. This is particularly true for I/O devices which are mainly of a mechanical rather than a purely electronic nature, such as printers and disk drives, and thus are usually orders of magnitude slower than CPU actions. It is even worse for a keyboard: Not only is human typing extremely slow relative to CPU speeds, but also a lot of the time the user is not typing at all; he/she may be thinking, taking a break, or whatever.

Accordingly, if we use wait-loop I/O, the CPU must execute the wait loop many times between successive I/O actions. This is wasted time and wasted CPU cycles.

An analogy is the following. Suppose you are expecting an important phone call at the office. Imagine how wasteful it would be if phones didn't have bells—-you would have to repeatedly say, "Hello, hello, hello, ..." into the phone until the call actually came in! This would be a big waste of time. The bell in the phone frees you from this; you can do other things around the office, without paying attention to the phone, because the bell will notify you when a call arrives.

Thus it would be nice to have an analog of a telephone bell in the computer. This does exist, in the form of an **interrupt.** It takes the form of a signal sent to the CPU by an I/O device. This signal forces the CPU to suspend, i.e. "interrupt," the currently-running program, say "X," and switch execution to another procedure, which we term the **interrupt service routine** (ISR) for that I/O device.[3]

Here is a simple example (a bit more complex this time, since we are reading several characters, not just one):

[3]This is called a **hardware interrupt** or **external interrupt**. Those who have done assembly-language programming on PCs should not confuse this with the INT instruction. The latter is almost the same as a procedure CALL instruction, and is used on PCs to implement systems calls, i.e. calls to the operating system.

```
LP: ...             ; do some work in this loop, maybe unrelated to keyboard
       ...
       ...
       ...
       ...
       J LP

       ...
       ...

       ; ISR starts here
       ; the following code must be loaded at address 0x1000000
       PUSH R4          ; we will use R4, etc. here, so must save old ones
       PUSH R5
       PUSH R6
       IN R4,[60]       ; pick up the character from the KSP
       MOV R5,COUNT     ; get current character count, stored in memory
       MOV R6,a(CHARS)  ; where does character array start?
       ADD R6,R5        ; R6 now points to place for next character
       MOV [R6],R4      ; place the character there
       POP R6           ; restore old register values
       POP R5
       POP R4
       IRET             ; back to the interrupted routine
```

Note that the ISR may or may not have any relation to what is in the loop (starting at LP). But the key point is that at some times (whenever someone hits a key, in this case) the ISR will need to be run, and from the point of view of the interrupted program, that time is unpredictable. In the example here, each time a character is typed, a different instruction within the loop might get interrupted.

What kind of code might be in the loop? Well, for example, consider a program to do some printing. For simplicity, let us suppose that the printer is set up so that we give it one character at a time. (Most modern printers would have us give them groups of characters at a time, but the general principle would be similar.) Again, since a printer is a mechanical device, it is slow when compared to CPU speeds. It would be quite a waste to use wait-loop I/O, tying up the CPU between printing of successive characters.

So, for example, we could put some sort of game in the LP loop, so that at least the user would have some entertainment while the printing is being done. And even though the game would be interrupted frequently, the user would never notice it, since the execution time for the ISR for each character would be quite small.

Interrupt systems can get quite complex. What happens, for example, when two I/O devices both issue interrupts at about the same time (i.e. within the same instruction in the LP loop)? Which one gets priority? And even when there is just one interrupt at a time, how can the program or CPU sense which of several I/O devices is the one which “rang the bell” in this case? Since each I/O device will have its own ISR, this issue must be resolved. There are many possible resolutions, but we will not discuss them here.

5.3 Direct Memory Access (DMA)

In our keyboard ISR example above, when a user hit a key, the ISR copied the character from the KDP into a register, then copied the register to memory. This is wasteful; it would be more efficient if we could arrange for the character to go directly to memory from the KDP. This is called **direct memory access** (DMA).

DMA is not needed for devices which send or receive characters at a slow rate, such as a keyboard, but it is often needed for something like a disk drive, which sends/receives characters at very high rates. So in many machines a disk drive is connected to a DMA controller, which in turn connects to the system bus.

A DMA controller works very much like a CPU, in the sense that it can work as a **bus master** like a CPU does. Specifically, a DMA controller can read from and write to memory, i.e. assert the MR and MW lines in the bus, etc.

So, the device driver program for something like a disk drive will often simply send a command to the DMA controller, and then go dormant. After the DMA controller finishes its assignment, saying writing one disk sector to memory, it causes an interrupt; the ISR here will be the device driver for the disk, though the ISR won't have much work left to do.

A Specifications of Our Mythical Machine

The instruction set will consist of:

```
MOV x,y       copy y to x
ADD x,y       add y to x
SUB x,y       subtract y from x
AND x,y       AND together x and y, placing the result in x
OR x,y        OR together x and y, placing the result in x
NOT x         peform a 1s-complement operation on x
IN x,y        do an input operation from the I/O port y, putting the
              result in x
OUT y,x       do an output operation to the I/O port y, using x as source
JZ t          jump to t if the Z bit is set
JN t          jump to t if the N bit is set
J t           jump unconditionally to t
CALL t        call the subroutine at t
RET           return from subroutine
IRET          return from interrupt service routine
PUSH x        push x onto stack; x must be a register
POP x         pop stack and place the popped value into x; x must be
              a register
```

Here x can be either a register or a memory location, and y can be either a register, memory location or a constant (assumed to be in hex notation), but x and y cannot both be memory locations (or a memory location and a port). Registers are denoted R0, R1, R2 and so on, and also include the stack pointer SP. A memory location can be given either as a label, as with X in

```
MOV R2,X
```

as a hex literal, as in

```
ADD R4,[2500]
```

or with **indirect addressing**, as in

```
ADD R4,[R6]
```

Note that the brackets are used to distinguish labels from constants, and register addressing from indirect addressing. Thus

```
ADD R4,[2500]
```

adds whatever is in memory location 2500 to R4, while

```
ADD R4,2500
```

would add the number 2500 to R4, and

```
ADD R4,[R6]
```

adds whatever is in the memory location pointed to by R6 to R4, while

```
ADD R4,R6
```

adds the contents of R6 itself to R4.

One can specify the address of a lable using the notation a(), e.g. a(X) for the address of X.

With the control-transfer instructions JZ, JN, J and CALL, the target t is specified as with memory addresses above, i.e. either as a label or as a hex literal surrounded by brackets.

The Z (Zero) and N (Negative) bits are stored in a special PS (Program Status) register. They reflect the result of the most recent ALU operation (including MOV, in which the ALU does a copy operation); Z will be 1 or 0, depending on whether the latest ALU operation resulted in a zero value, respectively, and similarly for N.

The stack is assumed to grow toward 0. Thus the CALL instruction will perform the following operations:

```
SP <-- SP-1
copy PC to the memory location pointed to by SP
copy t to the PC
```

An interrupt will cause the following to occur as soon as the current instruction finishes:

```
CPU notices the Interrupt Request
   line in the system bus has
   been asserted by some I/O
   device
CPU pushes current PC value on stack    ; save PC of interrupted routine
CPU pushes current PS value on stack    ; save PS of interrupted routine
CPU does PC <-- 0X1000000                  ; jump to ISR
```

The IRET instruction “undoes” all of this:

```
CPU pops stack and placed popped value ; restore old PS
   in PS
CPU posp stack and placed popped value ; restore old PC
   in PC
```

We assume the machine has a Keyboard Status Port (KSP) at port 60, and a Keyboard Data Port (KDP) at port 61. The KSP’s bit 0 indicates whether a character has arrived (coded 1) or not (coded 0). Before the user hits a key, this bit is 0; when the user finally hits a key, the bit changes to 1; then when the program reads that character by reading the KDP, this bit reverts to 0.

We will not define ports for other machines here, since they will not be needed for examples, but a typical machine would have a number of I/O devices and thus have many ports.