Lectures #29 & 30 – April 22 & 23, 2004

Input Output & Interrupts

In this section we continue the discussion of input/output approaches, but this time from the standpoint of the processor rather than abstract devices. Four concepts are critical here, in two somewhat overlapping pairs: polling vs. interrupts, and port based I/O vs. memory-mapped I/O.

In ***memory-mapped*** I/O, an external device shares some of the primary memory with the CPU and can generally access that memory at the same time without conflicts. For example, the original graphics card for the IBM-PC™ called the Color Graphics Adapter (CGA) shared a 16K segment of RAM with the main 8088 processor. Depending on the selected video mode, placing a byte value into the 16K segment would either cause a text character or a pixel to appear on screen. The CGA card would read the values out of that segment and paint the appropriate items on the screen. Other graphics modes for early video cards operated in similar ways.

The original CGA had a problem when both it and the 8088 accessed the 16K RAM block at the same time: multicolored "snow" appeared on the screen. This indicated a conflict in synchronizing the video card with the main memory bus. Programmers either lived with the problem or they wrote code to "watch" for the video retrace signal, when the electron beam of the display tube was moving back to the start of a line or to the top of the frame and no information was being displayed.

This retrace information was available to a program by reading an input ***port*** rather than a location in primary memory. Reading a port is a lot like loading a value from memory in the sense that every port has a unique "address" (*not* the same as a memory address), and an input port places its value onto the data bus when its address is referenced by a port instruction (the `IN` instruction on the 8088). The value placed on the bus comes from wires connected to devices in the outside world. Similarly, writing to a port is a lot like storing a value into memory; the port address and data value are put onto their respective buses with a port instruction (the `OUT` instruction on the 8088) and the 1s and 0s of the value are sent to the appropriate wires in the outside world.

Watching a port for a value on one specific bit usually means loading a register from the port, masking off all bits except the one of interest (usually by ANDing the register with 1 for the bit of interest and 0 for all other bits), and then comparing the masked value against zero. This repeats until the masked value is correct, a process called ***polling***. For example, watching the rightmost bit of port 12 until it becomes 1 is written in a high-level pseudocode as:

```
Repeat
     R0 := IN(12) AND 00000001
Until R0 <> 0
```

Suppose you were sitting in a comfy chair reading a book, with a telephone on a table next to you. With polling, you would pick up the telephone at regular intervals and ask "hello?", and if no one was there you put the telephone down and continue reading. This is the inherent problem with polling: a certain amount of processing time is spent seeing if there is work to do, and little else can be done until the watched-for event actually happens. These so-called ***busy-waits*** may be necessary in some circumstances, but in general they are very inefficient uses of processor time.

Instead, you read your book continuously, wasting no processor time until the phone actually rings. At this point, you put a bookmark in your book, answer the phone, and when the conversation is over you hang up and return to the book *at the place you left off*. At this point everything is restored to the point where you could handle another call should one come in. These ***interrupts*** are asynchronous since you do not know when they will occur.

Interrupts on a computer can be generated by asynchronous external events such as button-pushes or timer tics, or they can be generated through software instructions. When an interrupt occurs, the minimum processor state that must be saved is the value of the program counter at the time of the interrupt and the processor status flags. This amount of information is automatically saved by the hardware when an interrupt occurs. It then becomes the responsibility of the ***interrupt service routine*** to save any other necessary registers. An interrupt service routine is similar to a subroutine, but upon exit it must return to the *exact* state of the processor at the time of the interrupt event. The processor status flags must be kept transparent along with all other registers, since we cannot predict their values at the time of the interrupt and instructions after the point of the interrupt will depend on them.

On the 6502 there are three external lines into the chip. Those lines are called IRQ (***interrupt request***), NMI (***non-maskable interrupt***), and RES (***reset***). Activating any of them causes the processor to jump indirectly through the appropriate one of three locations at the end of high memory. These locations, called ***interrupt vectors***, are fixed in the address space so that the hardware always knows where to go. Since they must contain valid values on power-up, these memory locations must be in ROM. The RES line resets the processor and jumps through its vector to the processor start-up code, so it really isn't an interrupt in the strictest sense. The IRQ line is a true interrupt, which may be enabled or disabled (or ***masked***) under software control, and since the IRQ can be disabled this is an interrupt *request*. When activated an IRQ pushes both the return address and the processor status flags onto the stack, and also disables the interrupt. This allows the interrupt service routine to do its job without being interrupted again (unless the service routine reactivates the interrupt system). The NMI is nearly identical to the IRQ except that it may *not* be deactivated under software control. This is why the interrupt is called ***non-maskable***. In this system, the NMI has a higher priority than IRQ. Normally an IRQ cannot interrupt the service routine for an NMI, but an NMI *can* interrupt the service routine for an IRQ. The `BRK` instruction generates an IRQ internally under software control. The `RTI` (return from interrupt) instruction is different from a normal `RTS` (return from subroutine) instruction in that it must pop the status register from the stack along with the return address.

On the 8088, the first 256 4-byte locations in memory (the first 1K of RAM) contain interrupt vectors. Every `INT` instruction also contains a 1-byte number which indicates which vector to use. In MS-DOS, the built-in functions of the BIOS (Basic Input/Output System) are all driven by interrupts. Thus, if I want to call some predefined function on an IBM-PC I put parameter values in the appropriate registers and generate an interrupt. This approach allows the BIOS to be updated in the next version of the operating system without breaking any user code. You cannot guarantee between versions of the BIOS where the service routines will actually appear in memory, but the interrupt vectors *will* point to the correct addresses. The user code simply refers to the correct interrupt vectors without knowing where the service routines are located. Accidentally overwriting the interrupt vectors is usually disastrous.

On the ARM, the `SWI` instruction contains space for a large number. This number is used by the interrupt service routine to determine which function to invoke (such as `SWI &11` to exit an ARMulator program and `SWI &0` to print a character, for example, functions which will certainly have different meanings on other systems). On the ARM, invoking an interrupt causes a set of auxiliary registers to be electrically swapped in for the program counter and processor status register (along with a small number of general purpose registers). Returning from an interrupt simply swaps the original registers back into their normal places.

Almost all of the concepts presented here were employed in the MS-DOS based QuickCam camera example presented in class. Setting the video mode requires the use of BIOS functions called through software interrupts. Painting graphics on screen requires the use of memory-mapped I/O, since the video screen shares primary memory with the CPU. Input from the camera came in via input ports, and camera events were detected by polling.

Timer interrupts are used today to break out of application programs and return control to an operating system. The OS then can activate another program for a while, until the next timer interrupt, and the application programs need not contain special code for calling the OS. This approach is called ***pre-emptive multitasking***. In contrast, Windows 3.1 forced application programs to continually call the OS so it could switch tasks equitably; this ***cooperative multitasking*** only works if all the application programs cooperate!