This lecture we will look at several methods for sending data between an I/O device and either main memory or the CPU. (Recall that we are considering the hard disk to be an I/O device.)

Programmed I/O: isolated vs. memory-mapped

When we discussed physical addresses of memory, we considered main memory and the hard disk. The problem of indexing physical addresses is more general that that, though. Although I/O controllers for the keyboard, mouse, printer, monitor are not memory devices per se, they do have registers and local memories that need to be addressed too. We are not going to get into details of particular I/O controllers and their registers, etc. Instead, we'll keep the discussion at a general (conceptual) level.

From the perspective of assembly language programming, there are several general methods for addressing an I/O device (and the registers and memory within the I/O device).

When program MIPS in MARS, you used `syscall` to do I/O. `syscall` causes your program to branch to the kernel. The appropriate exception handler is then run, based on the code number of the `syscall`. (You don't get to see the exception handler when you use the MARS simulator.) MIPS `syscall` is an example of *isolated I/O*, where one has special instructions for I/O operations.

A second and more subtle method by which an assembly language program can address an I/O device is called *memory mapped I/O* (MMIO). With memory-mapped I/O, the addresses of the registers or memory in each I/O device are in a dedicated region of the kernel's virtual address space. This allows the same instructions to be used for I/O as are used for reading from and writing to memory. (Real MIPS processors use MMIO, and use `lw` and `sw` to read and write, respectively, as we will see soon.) The advantage of memory-mapped I/O is that it keeps the set of instructions small. This is, as you know, one of the design goals of MIPS i.e. reduced instruction set computer (RISC).

MARS does allow some tools for programming the kernel, so let's briefly consider memory mapped I/O in MARS, which is a very simple version of MMIO in MIPS. There is only one input device (keyboard) and one output device (display). Both the input and output device each have two registers. The addresses are in the kernel's address space.

```
0xffff0000   input control register   (only the two LSB's are used)
0xffff0004   input data register      (holds one byte)
0xffff0008   output control register  (only the two LSB's are used)
0xffff000c   output data register     (holds one byte)
```

The LSB of each control register indicates whether the corresponding data register is "ready". In the input case, ready means that a character has been typed at the keyboard and is sitting in the low order byte of the data register. In the output case, ready means that the output data register is free to receive a byte from the CPU. e.g. in the case of a (non-buffered) printer, the device might not yet have printed the previous byte that was sitting in the register and hence might not be ready. [Here is the notes I say the registers hold only one byte, whereas in the lecture and in the instructions below I imply that they hold a word each. It doesn't really matter.]

Let's look at an example. Suppose your program should read bytes from the input device and load each byte into register `$s2`. Here's how the kernel might try to do it.

```
lui  $t0, 0xffff
lw   $s2,  4( $t0)    # load from address 0xffff0004
```

From the kernel programmer's perspective, this is quite simple. However, from the underlying hardware perspective, it is more complicated. The MIPS hardware must recognize that the address is in the memory mapped I/O region and handle it properly. In particular, when the kernel program executes the above `lw` instruction, the hardware must detect that the desired address does not correspond to an item in the data cache but rather is some word in the I/O device's memory. This result is similar to a cache miss, in that the process must stop and the kernel must use the system bus to get something. But now it must get the desired word from the I/O device (input) instead of a block from main memory (cache refill).

The CPU puts an address on address line of the system bus, e.g. the address of the I/O device[1] and sets certain control lines on the system bus to indicate that the CPU wants to read from that device. The I/O device controller reads the bus (always) and sees the address and control and then puts the requested data item onto the data bus.

I emphasize: this is the same general mechanism that would be used for a main memory access in the case of a cache miss. Now it is an I/O device that provides the data to the CPU.

Similarly, for an output, the `sw` instruction would be used and the address where the word should be stored would be within a reserved (memory mapped) region of the kernel's address space. The CPU would put that address on the bus (after translating this virtual address into a physical address that indicates the I/O device number) and set a write control signal on the control bus. Again, the mechanism is similar to storing a word to main memory. The main difference is that the addresses on the address bus indicates that the CPU is communicating with the I/O controller, not with main memory.

Polling

One issue that arises with the above example is that it only makes sense to read from the input device when the input device is ready, e.g. when there is a byte of data to be read. (This issue is independent of whether one uses isolated I/O or memory mapped I/O.)

To solve this problem, one can use a method called *polling*. Before the CPU can read from the input device, it checks the status of the input device and see if it is "ready," meaning that the CPU checks the "ready" bit in the input control register `0xffff0000`, which is bit 0 (the least significant bit). In particular, the CPU needs to wait until this bit has the value 1. This can be implemented in MIPS with a small *polling loop*.

```
          lui   $t0, 0xffff
Wait:     lw    $t1,  0($t0)      # load from the input control register
          andi  $t1, $t1, 0x0001  # reset (clear) all bits except LSB
          beq   $t1, $zero,  Wait # if not yet ready, then loop back
          lw    $s2,  4( $t0)     # input device is ready, so read
```

[1] and/or some register number or some offset within the local memory of that device

A similar polling loop is used for the output device:

```
          lui    $t0, 0xffff
Wait:     lw     $t1,  8($t0)     # load the output control register
          andi   $t1, $t1, 0x0001 # reset all bits except LSB
          beq    $t1, $zero, Wait # if not ready, then loop back
          sw     $s2,  12( $t0 )  # output device is ready, so write
```

Obviously polling is not an efficient solution. The CPU would waste many cycles looping and waiting for the ready bit to turn on. Imagine the CPU clock clicking along at 2 GHz waiting for a human to respond to some prompt and press <ENTER>. People are slow; the delay could easily be several billion clock pulses.

This inefficiency problem is solved to some extent by limiting each process to some finite stretch of time. There are various ways to implement it. The simplest would just be to use a finite for-loop, instead of an infinite loop. The number of times you go through the loop might depend on various factors, such as the number of other processes running and the importance of having the I/O done as soon as the device is available.

I emphasize that, while this polling example uses memory-mapped I/O, it could have used isolated I/O instead. i.e. Polling isn't tied to one of these methods.

Example: buffered input with a keyboard

[**Added April 10:** While I am presenting this example of buffered input in the context of polling, there is nothing in this example that restricts it to polling *per se*. Rather, the example illustrates what *buffering* is, and why it is useful when there is a bus that is being shared. Buffering can be used for other I/O schemes as well, e.g. DMA and interrupts.]

Suppose a computer has a single console display window for the user to enter instructions. The user types on a keyboard and the typed characters are echoed in the console (so that the user can see if he/she made typing mistakes).

The user notices that there sometimes there is a large time delay between when the user enters characters and when the characters are displayed in the console but that the console eventually catches up. What is happening here?

First, assume that the keyboard controller has a large buffer – a *circular array* [2] – where it stores the values entered by the user. The controller has two registers, `front` and `back`, which hold the numbers of characters that have been entered by the user and the number of characters that have been read by the CPU, respectively. Each time a character is entered or read, the registers are incremented. Certain situations have special status, which could be detected (see slides for details) :

- The character buffer is "full" when the difference of these two indices is N and in this case keystrokes are ignored.
- The keyboard controller is "ready" to provide a character to the CPU when the difference of the two indices is greater than 0.

[2] In case you didn't cover it in COMP 250, a circular array is an array of size N where the index i can be any positive integer, and the index is computer with $i \ mod \ N$.

In each case there might be a circuit which tests for these conditions.

With this buffer in mind, we can see what is causing the delay in echoing to the console. Suppose the user is typing into the keyboard, but the system bus is being used by some I/O device. The buffer starts to fill up, and the CPU only reads from the buffer when the bus is available for it to do so. The CPU grabs a short time slice, and quickly reads a character and writes a character (to the console i.e. output) and does this until it empties the buffer or until it gives up the bus or pauses the process and continues with another process.

Direct memory access (DMA)

Another mechanism for I/O is *direct memory access* (DMA). This is a specialized method which involves communication between memory and an I/O device. The idea is for an I/O controller (say the hard disk or the printer controller) to have its own specialized set of circuits, registers, and local memory which can communicate with main memory via the system bus. Such an I/O controller is called a *DMA controller*. The advantage is that it can take over much of the work of the CPU in getting the data onto and off of the system bus and to/from main memory. The CPU would first tell the DMA controller what it should do, and then the CPU can continue executing other processes while the DMA controller uses the bus.

What does the CPU need to tell a DMA controller in order for the DMA controller to take over this work? Take the example of a page fault. The kernel program (page fault handler) needs to modify the page table, and it also needs to bring a page from the hard disk to main memory (and maybe swap a page out too). How is this done?

Let's take the case that a page is swapped out, which is what I sketched in class. The kernel needs to specify the following parameters to the DMA controller of hard disk:

- a physical page address in main memory to be swapped out
- a physical page address on the hard disk (where the page goes)
- control signals (e.g. copy one page from main memory to hard disk)

The DMA controller then initiates the memory transfer. It could do so by executing a sequence of instructions, something like this.

```
initialize baseReg to address where page will be put in the local
memory of the controller (the "disk cache")

while offset = 0 to pagesize -1 {
   put address AddrMainMem on address but, and put control signals on the
     control bus so main memory writes a word onto data bus
   read word from data bus and put into local memory at
              address (baseReg + offset)
   AddrMainMem = AddMainMem + 4  //  1 word
}
```

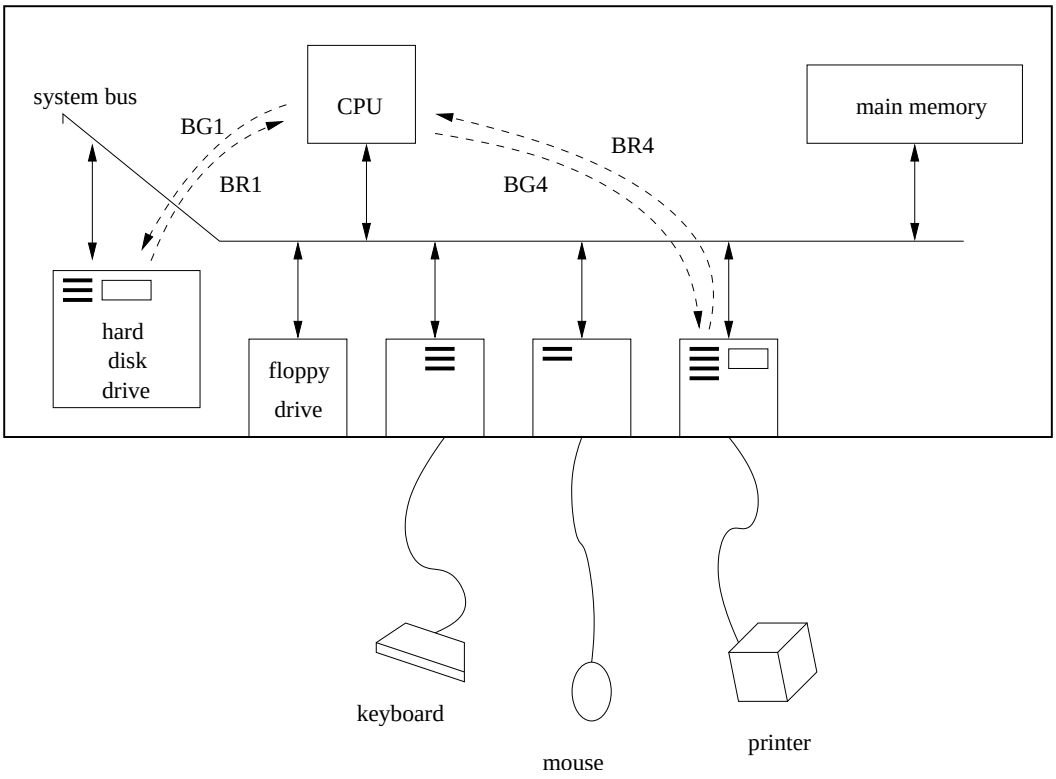
Then, when the entire page has been read, the page can be written the hard disk itself. For this, the controller uses the value AddrHardDisk which would be the address where the page goes on the hard disk. A similar sequence of instructions as above could be used for this.

DMA: Bus request and bus granted

The next major issue to be discussed is how the DMA controller and the CPU coordinate who gets to use the bus at a given time. When the CPU sends its instructions to the DMA controller (as in the above example), the DMA controller might not be ready to execute the instructions right away. For example, suppose a page fault occurs and the page fault handler requests a page to be transferred from the hard disk DMA controller to main memory, i.e. it communicates this request on the system bus. There is no need for the CPU to disconnect itself immediately from the system bus after making the request since the HD controller first needs to get the page off the hard disk before it can use the bus. If the hard disk controller is just waiting for the hard disk to swing around, then there is no reason why the hard disk controller needs to have the system bus.

How then can the hard disk controller eventually take over the system bus once it is ready to use it, namely after it has read the page from the disk and put it in its local memory? (Assume there is only one I/O device (the hard disk) and hence only one DMA device. We'll return to the case of multiple DMA devices next lecture.) The hard disk/DMA controller needs to communicate that it is ready to use the bus, and it needs to do so without using the system bus!

A DMA controller and the CPU communicate via two special control lines: BR (bus request) which goes from the DMA controller to the CPU, and BG (bus granted) which goes from CPU to DMA. If the CPU is using the bus, it sets BG to 0. Now, when the DMA controller is ready to use the system bus, the DMA controller sets BR to 1. If the CPU doesn't need the system bus, it temporarily disconnects itself from the system bus (electronically speaking, it shuts off its tri-state buffers and stops writing on the system bus) and it sets BG to 1. If the CPU is using the system bus, it finishes using it before disconnecting from system bus and setting BG to 1.



I emphasize that BR and BG signals cannot be sent on the system bus. These signals are used to decide who uses the system bus, and so they must always be available. The CPU must always be able to communicate the value of BG to the DMA controller, and the DMA controller must always be able to communicate the value of BR to the CPU. Separate direct lines ("point-to-point") between the CPU and the DMA controller are needed to carry these signals.

[ASIDE: The above figure shows two DMA devices, say the printer and the hard drive. I have labelled two sets of BR and BG signals. The figure also shows that the I/O controllers have their own registers and little RAMs. These registers have physical addresses (I/O device number and register number or memory address number within that device). In a memory mapped system, these registers and memory would have corresponding virtual addresses which would need to be translated.]

Hard disk

A few final comments about how to improve hard disk performance....

Recall that the hard disk controller has a local memory (DRAM), sometimes called the *disk cache*. When a page is read from disk, it is fist put into this disk cache, and then transferred to main memory as a subseqent stage. Similarly, when a page is brought from main memory to the hard disk controller, it is put in the disk cache and then later moved to the disk itself (as was described earlier in the lecture).

To improve performance when reading a page from the hard disk, neighboring physical pages could be read as well. This is easy to do because once the hard disk has spun around to the position of the page, the controller can easily read off neighboring pages too. Reading neighboring pages on disk would be useful when adjacent virtual pages are stored on the hard disk as adjacent physical pages. For example, a file might consist of multiple pages – it would be better to store them side by side so that they can be accessed at the same time.

You might have heard of *disk fragmentation*. For example, suppose you are storing a file on your hard disk. The file is an application which consists of many pages of source code. When you first put the file on the hard disk, it would be nice to keep all the pages in order and at consecutive page addresses on the disk for the reason I mentioned above. However, this might not be possible since the disk might not have a big empty region available. The disk might contain lots of free space, but it might consist of many holes e.g. where previous files had been deleted. You can imagine that after years of use, and many adds and file deletes and many page swaps, the number of large empty gaps decreases and the number of small gaps increases. This is fragmentation. The disk doesn't work as well when it is fragmented, because the trick mentioned in the previous paragraph is not as effective.

Finally, another way to improve performance of the disk arises when there are several read/write requests for data on different parts of the disk. Rather than performing the read/write in the order of request, which might take multiple spins of the disk, the hard disk controller could sort the list of pages so that it could access them in one spin. For example, suppose requests were made for page 3,7,5,4 at a time when the head is at position for page 2. Rather that doing 3, 7 and then spinning all the way around for 5 and then all the way around again for 4, it could get the requests in order 3,4,5,7 in a single spin of the disk.