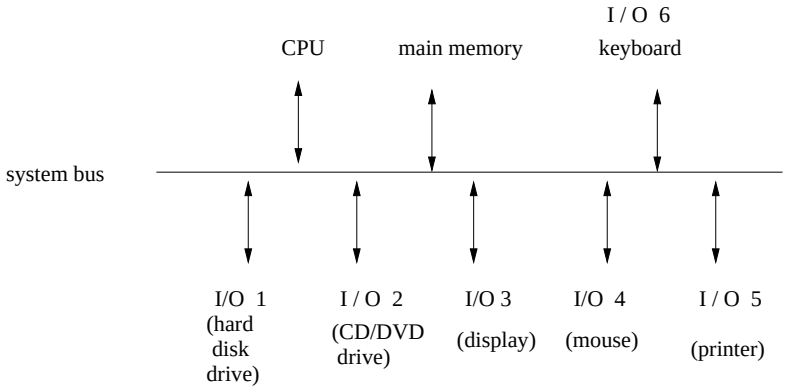
Up to now in the course, we have concentrated on the CPU and memory. In the final few lectures, we will briefly consider some of the basic ideas of how the CPU and memory interact with the I/O devices. The *I/O devices* we will consider are the keyboard and mouse (inputs), and monitor and printer (outputs). The harddisk drive is also sometimes considered an I/O device and is both input and output. We will not consider the issues of commputer networks, which is obviously also included in I/O but which is a more advanced topic.

Our discussion of I/O will be relatively general and untechnical, in contrast to what we have seen up to now. Also, we will not consider current technologies, which are quite complex and diverse and would be inappropriate for an introductory course such as this.

(Shared) System bus

The classical way to connect the CPU, main memory, and I/O devices is to use a shared set of lines called a *system bus*. A "bus" is just a set of wires for carrying bits which is shared by a number of devices. The classical system bus connects the major units in the computer: the CPU, the main memory, the I/O devices. For now, let's assume there is just one system bus. (Later we will see that more recent computers have a hierarchy of buses.)



The main advantage of the system bus is that it reduces the number of connections in the computer. Rather than the CPU (or main memory) having a direct connection with many lines each to each of the I/O devices, all the components can share the same lines. There are obviously disadvantages to such a system bus as well. Because the bus is shared, it can only carry one signal at a time. This tends to reduce performance because one component (CPU, an I/O device, main memory) may need to wait until the system bus is free before it can put signals on the system bus. Also, the various components need to communicate with each other to decide who gets to use the system bus at any given time and who is talking to whom. These communications also take time.

Another disadvantage of the system bus is that it needs a clock speed that is much slower than the CPU. Components that share the system bus are relatively far away from each other. The clock speed of the system bus is constrained by the furthest (worst case) connection between components. That is, any signal that is put on the bus has to travel between any two components that share the bus in any single clock cycle. If we have one system bus shared by all components, then the clock speed might have to be 100 MHz (a period of 10 ns), as opposed to a typical CPU clock speed of 1 GHz (a period of 1 ns).

To be concrete about the limits here, consider that light (or more generally, electromagnetic waves i.e. the voltages that drive our gates) travel at a speed of $3 \times 10^8 ms^{-1}$. If we have a 3 GHz processor, then this would mean that the voltages can travel at most $10cm$ per clock pulse, which is smaller than the distances between many pairs of components in a desktop workstation. And this doesn't include the time needed for the components to read/write from/to the bus and for circuits to stabilize. Hence, the system bus runs at a slower clock speed, giving more time for the 0/1 signals on the bus to stabilize.

There are three components to the system bus. There is a *data bus* which carries data only. There is an *address bus* which carries addresses only. And there is a *control bus* that carries control signals only.

How is the system bus used? Let's take an example: the `lw` instruction. Suppose that a cache miss occurs and so a block containing the word must be brought from main memory into the cache. Here is what we would like to happen.[1]

- The CPU puts the (physical) address of the new block onto the address bus.
- The CPU sets control signals (ReadMem = 1 and WriteMem = 0) on the control bus.
- The control signal causes a block to be read from main memory, i.e. main memory puts the requested block onto the data bus.
- The CPU reads the data bus and the block is written into the data cache.

There is more to it than this, however. For example, it would take several system bus clock cycles for this to happen since each word of the block needs to be sent. Moreover, the above assumes that the system bus was not being used by any other component. But when you have a shared resource, you are not guarenteed that it will be available. We will discuss how the system bus is shared next lecture.

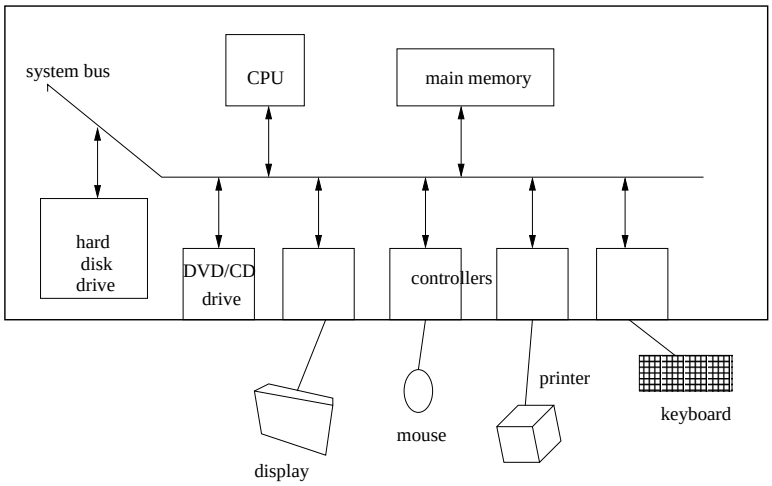
I/O controllers

Each of the I/O devices may have its own complicated electronics (and mechanics, in the case of a printer or scanner). Each I/O device interfaces with an *I/O controller* which is inside the computer case. The controller is responsible for reading/writing on the system bus. Each I/O controller might have its own specialized processor e.g. own clock, its own special registers, its own special ROM circuits, its own memory, and its own little program counter (PC).

Do not confuse I/O controllers with *device drivers*. A driver for an I/O device is a specialized piece of software that knows all about the particular device and its I/O controller. In particular, the driver controls the reads and and writes for the registers and other circuits in the I/O controller. The drivers are part of the operating system (kernel) and they sit in main memory or on disk.

Note that many I/O devices (printer, keyboard, mouse, monitor, etc) are external to the case of the computer. We use the term *peripheral* to describe the hardware that is external to the computer, that is, outside the *case*. These devices don't plug directly into the system bus. Rather, each plugs into a *port*. Each port is typically part of the I/O controller that sits on the motherboard.

[1] We ignore the possible write-back e.g. if the cache entry is dirty.



Let's next consider a few examples of how various components can share the system bus. Recall the tri-state buffer circuit element from lecture 6. This was a mechanism for allowing many wires to put signals on a single wire. We will also need to use a tri-state buffers for the system bus. The reason is that we have many devices that can put signals onto the bus, and so only one of them can write to the bus at any time.

Example (input device): Keyboard

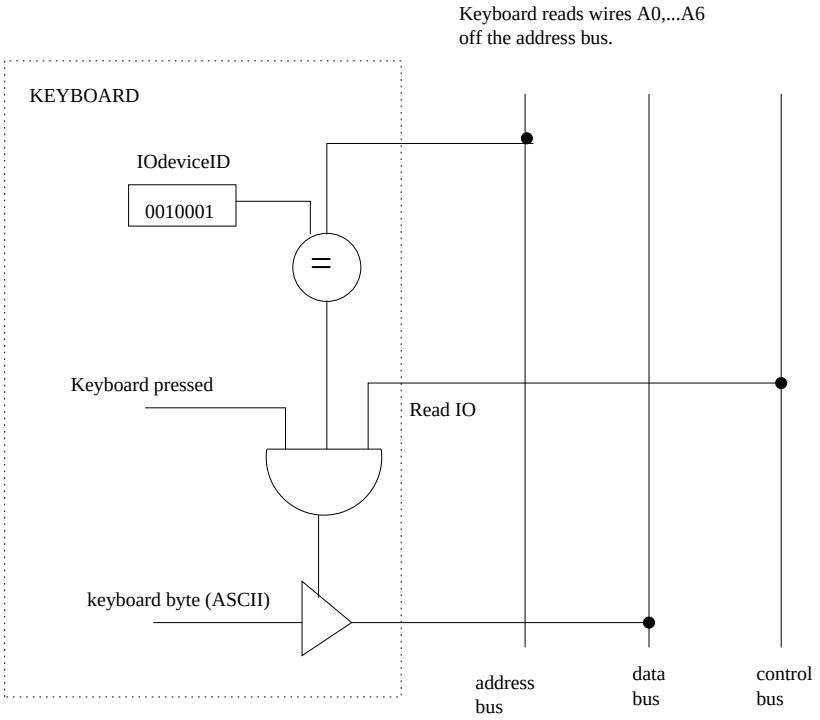
A keyboard is an input device that enters one byte of data at a time. Suppose there are a maximum of 128 (2^7) I/O devices recognized by the CPU and that this keyboard happens to be I/O device number 17. Suppose further than there is a control line on the system bus, called `ReadIO` that is set to 1 by the CPU when it tries to read from one of the I/O devices. Then, the keyboard would put its byte of data on the bus if the ReadIO control signal is 1 AND the address of the I/O device matches the lower 7 bits (that is, $2^7 = 128$) of the address on the address bus AND one of the keys was pressed.

To check if the address of the I/O device is correct, the keyboard controller matches a particular set of lines on the address bus to the values in an IOdeviceID register (see figure). Since the keyboard is device number 17 (say) the binary number 0010001 is stored there. See the sketch on the next page.

Example (input device): mouse

Another example of an input device is a mouse. A mouse marks the (x,y) position of a cursor. Dragging the mouse causes the (x,y) values to change. Moving to the right and left causes the x value to increase and decrease, respectively. Moving forward and backward causes the y value to increase and decrease, respectively. The mouse also has buttons and a scroll option.

The mouse itself contains electronics which convert the mechanical action into "packets" of data which are sent to the mouse I/O controller. The controller (inside the computer case) decodes these packets and writes the results into various registers. The I/O controller then communicates the data with the system bus. I will sketch out more details on how this is done in coming lectures.



Output devices

We will look briefly at two types of output devices: printers and displays (monitors). The first distinction to be aware of in thinking about the information sent to an output device is whether the data describes how to make an image (called *vector graphics*) or whether the data describes a square grid of pixels, in particular, their intensities i.e. RGB values (called *raster graphics*).

Example : printer

Many printers are built to understand instructions in the (Adobe) Postscript^{TM} language. An application such as Word or Adobe Reader or a web browser issues a print command. The PostScript file to be printed is sent from main memory to the I/O (printer) controller. Then individual PostScript instructions are sent from the controller to the printer. PostScript printers contain software and hardware which together interpret the instructions, and convert them into a rasterized image (i.e. a square grid of pixels). *See the slides for some very basic examples of postscript instructions.*

A printer is an output device, and so it does not put anything on the data bus. Hence, it does not need a tri-state buffer. To check whether the data on the data bus is meant for the printer, a similar mechanism could be used as in the keyboard example, i.e. read from the address bus and control bus. The file containing the printer instructions (PostScript) would be put on the data bus.

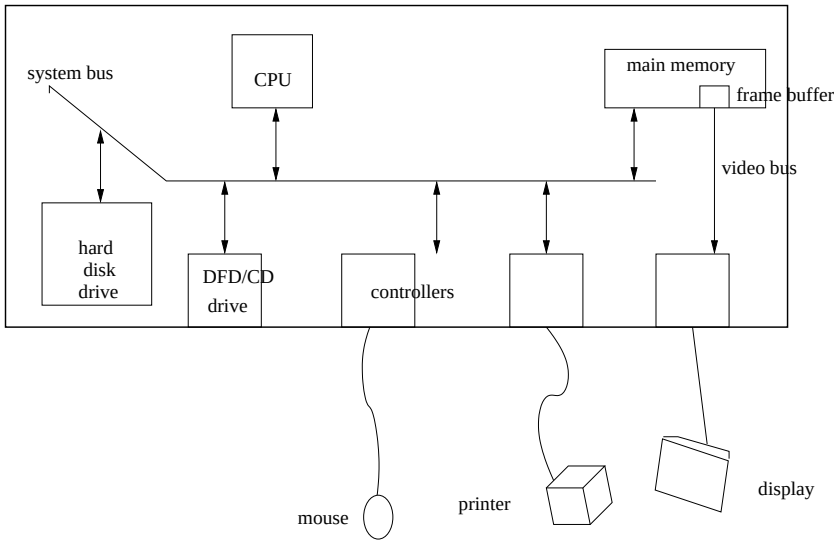
The printer determines whether it should read from the data bus by looking at the address on the address bus. Continuing with our earlier example, if the lower 7 bits match the address of this printer, then the data is meant for this printer and is written to the printer. Otherwise, the printer does not take data from the data bus.

Example: monitor

Our next example is the monitor. The monitor screen is a rectangular array of points called *pixels* (picture elements). For each pixel (x,y), there are three numbers stored which represent the R,G, and B intensities of the point. Typically, one byte is used for each of these intensities. That is, intensity codes goes from 0 to 255 (bigger means brighter).

A example image size is 1024 x 768 pixels. If we consider 3 bytes per pixel – one byte for each of R,G,B – then we need about 3 MBs of memory just for the image that is on your monitor at any instant.

A very long time ago in the 1980s when the first wave of personal computer occured , image pixel values were stored in a part of main memory called the *frame buffer* (see figure below). That is, your screen shows a large array from main memory. However, this was infeasible for all but the simplest images. Most monitors presents a new image every 1/60 of a second. (A minimum of 30 frames per second is required, otherwise the eye detects flicker and the video looks jittery.) If the image size is about 1000×1000 and even the screen were refreshed every 1/30 of second, then this would still require 100 MB per second (considering RGB) to get put on the system bus. This should make you concerned. The system bus would have to spend a large amount of its time just transfering pixels, which is clearly unacceptable.

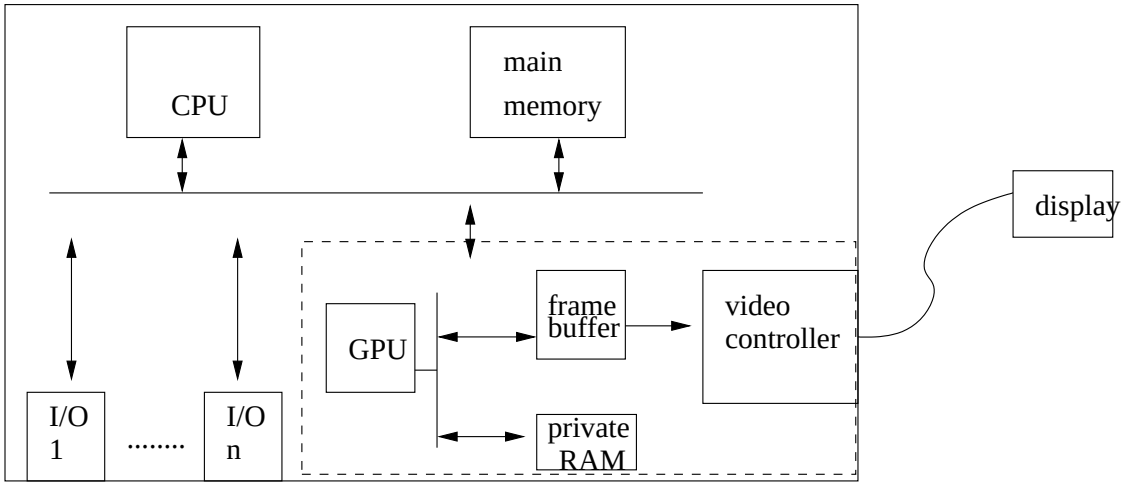


One solution to the system bus traffic jam problem is to have a dedicated (read only) bus to carry bytes from the frame buffer to the display controller. This *video bus* completely freed up the system bus. However, even if we don't use the system bus to transfer pixels to the video controller, the pixel values still need to be computed for each frame and sent to the frame buffer. A long time ago, the CPU was indeed responsible for making the images. However, the graphics capabilities were weak since the CPU had so many other things to do. And it still had to use the system bus to transmit these values to the framebuffer, which clogged up the system bus.

GPU (graphics processing unit)

The solution that evolved since the 1980s is to use a second processor called a *graphics processing unit* (GPU). This is a specialized processor for making images, e.g. drawing polygons and other fantastic things that can be done. (Take COMP 557 Fundamentals of Computer Graphics, if you wish to learn more.) Rather than the CPU performing drawing operations itself and computing the RGB intensity values of each pixel, these specialized computations are performed by the GPU. Only the instructions such as `drawline(x1,x2,y1,y2,R,G,B)` or `drawTriangle(x1,x2,x3,y1,y2,y3,R,G,B)` are sent to the GPU over the system bus. Such instructions are similar is concept (but *not* in detail) to the PostScript instructions mentioned earlier which were sent to the printer.

The GPU together with its private RAM and the framebuffer, are known as the *graphics card* or video card. (See dashed line in figure below.) The GPU executes the instructions it is given by drawing into the frame buffer, which is part of the graphics/video card.



So, problem solved? Not entirely. Relieving the CPU of so much computational burden just passes the problem on. The more operations we require the GPU to do, the more complicated the GPU design must be. It needs its own registers and controls, its own floating point unit(s), its own instructions. Modern graphics cards (video cards) contain thousands of processors and large amounts of memory. They can perform general purpose massively parallel computation as well (if you know how ot program them!) But there is price: the cards are expensive.

That was then... this is now...

The material I have covered in this lecture is appropriate for an introductory course, but it does not even attempt to explain the current technologies. If you wish to read on your own about how things work these days, check out the terms below:

- buses: *PCI bus* (1990s), *PCI Express* (since 2003)
- graphics cards: recent cards have thousands of processors on them, and (drop your jaw) can even be programmed (see *CUDA*) so that your laptop is as powerful as recent supercomputers. This is called GPGPU, for general purpose GPUs.