Chapter 9: Buses and Tri-state Logic

Overview

Communication is at the heart of any computer or computer application. Sooner or later, if you want the computer to be useful, it *must* exchange data with another device. In a laboratory setting, this could be an ADC, which converts a voltage into a digital signal and then sends it to the Central Processing Unit (CPU) for analysis. Alternatively, the CPU might generate a digital signal to send out to a DAC, which in turn will operate some piece of apparatus. This week you will learn some basics about buses, which are central to all computer-based communication hardware, and about the tri-state logic which buses use.

Buses

All interface elements, from your keyboard to your mouse, eventually transport digital information from the external world to the CPU. There are two obvious ways to do this.

1. ***Serial communication*** is the process where clock pulses key each bit (binary digit) in sequence into a shift register. This will seriously slow a computer that expects to receive data in 8, 16, or 32 bit pieces. It has the benefit of only requiring one line for input and one line for output. Sometimes input and output occurs on the same line, although obviously not at the same time.

2. ***Parallel communication*** transfers 8, 16, or 32 bits one each clock pulse. Clearly this is much faster, but it requires many simultaneous data lines. This is really the only choice for very fast processes, like sending data from the CPU to external Random Access Memory (RAM). The number of data lines could quickly become ridiculous if you have many separate devices that communicate with the CPU.

A method to employ parallel communication, while keeping the number of lines to a minimum, is to define a *bus*. Any bus uses a modest number of lines to allow the CPU to communicate with all of its external devices simultaneously. Of course, this can only work if each device knows when to speak and when to listen. Otherwise the devices could try to talk to the CPU at the same time leading to chaos (called *bus contention*). The device that determines who will speak and who will listen is called the *bus master*. Usually, the CPU is the bus master, but some clever buses can allow other devices to take over when necessary.

Input and output are always designated relative to the bus master. A READ means data is going into the bus master. A WRITE means that data is going out from the bus master.

Each bus has several lines dedicated to one of three separate duties.

1. ***Control lines*** are (mostly) unidirectional. They coordinate data transfer.

2. ***Address lines*** are unidirectional. The bus master uses them to designate which device it wants to communicate with.

3. ***Data lines*** are bidirectional so that they can carry binary data into or out from the bus master.

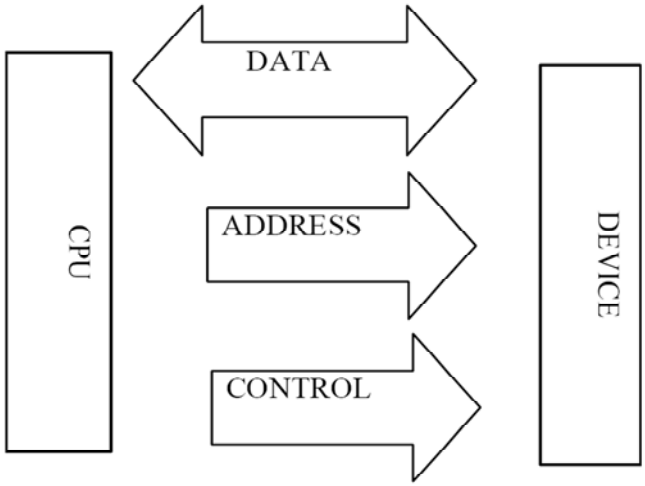

Figure 10-1: The bus.

Defining how many physical lines and their logical duties defines the bus. A bus defined without a widely recognized convention, or specification, is called a *private bus architecture*. Such an architecture would be designed for specific application and will not be generally applicable for many different uses. In most cases, one uses a well know and defined standard for bus applications (e.g. USB, PCI, SCSI, Firewire…).

Control Lines

Control lines are mostly output lines from the bus master to the external devices. Since these lines are bus master output lines, they can all be simultaneously tied to many external device inputs. Only the bus master can assert a level on one of these lines (the master talks on these lines and all the other devices listen to these lines). These control lines might tell an external device if it should read data or write data and when the transfer should happen.

In a PC, the Input/Output Write (IOW) goes LOW to signify that the bus master has written data to the bus; Input/Output Read (IOR) goes LOW to signify that an external device should write its data to the bus so that the bus master can read from the bus. In both cases, the falling edge on the control line triggers the timing for the transfer. Other buses have levels for the Read and Write lines, and a separate *strobe* line to control timing.

Most buses also have interrupt control lines, which are input lines to allow an external device to signal the bus master. Since interrupts are input lines to the bus master, one must be careful to insure that only one external device writes to an interrupt line at a time.

Address Lines

Address lines are also output lines that carry information from the bus master to all external devices simultaneously. Again, you can connect these address lines to the input of logic of all the external devices simultaneously because only the bus master will write to them. Usually, the address lines go to a *decoder* before going to each device. A decoder is a chip that recognizes when the address lines have been set for the specific

device. Older devices had “jumpers” to set the address that the decoder would recognize. More modern devices have their addresses set by software. This provides greater versatility.

In some buses, there are address lines that reach all of memory, and a subset of address lines that reach external devices. An additional control line determines whether the address means internal (RAM) or an external *port*. In a PC, this line is called *Address-Enable* (*AE*).

Data Lines and Tri-State logic

Data lines serve as input and output lines depending on whether the bus master wants to read or to write. The data lines are always connected to each device. As you know, it is illegal to connect the outputs of logic devices together. If one output wanted to force H, while the other wanted to force L, the result would be a lot of power dissipation and a dead chip. Consequently, all devices that might write to the data bus lines ***must*** do so through tri-state logic. As you might guess, tri-state logic has three states: H, L, and disconnect (which is sometimes called *high-impedance* or *High Z* state). If an output is in the High-Z state, it will not draw current from any other outputs to force them into any state. It behaves just as if the gate has been disconnected from the bus. Any particular device that wants to write to the data lines must have its Input/Output Write (IOW) line asserted before its output leaves the High Z state. It is the core duty of the bus master to insure that only one device is *IOW enabled* at a time.

The 2-Bit Bus

Consider a very simple bus that has two bits for address lines and a single data line. This bus can connect the bus master to four devices, numbered 00, 01, 10, and 11 (in binary, of course!), which can write data to the bus. This bus can carry data values of 0 and 1. Finally, this bus would have a read-enable (IOR). As usual, the enable line should be active-low.

This is a total of 4 lines, and it provides four commands (four output options). If we were to add write options to the bus via an IOW line, the number of commands would double. It should not surprise you that this number grows quickly as the number of address and/or data bits increases.

Read from bus (external device writes data to bus)

To command an external device to write to the bus so that the bus master can read in, you must do three things, in order:

1. Write the address of the external device to the address lines.

2. Bring the IOR line low. This, in combination with the correct address brings the device out of high impedance mode. The device will respond by writing out its data to the data lines (since only that device’s output lines will now be active).

3. After you (or a CPU) read the data, you return the IOR to high.

In all buses, there are standards for timing of these operations. Examples of standards include that the address and a data might be required to be set for 100 ns before the IOR goes Low, and the data is required to stay constant for another 5 ns after IOR returns high. As a second example, external devices might be required to make their data available no later than 550 ns after an IOR goes Low.

Buses in PCs

In the previous section, we considered a simple bus to show how a CPU could use just a few lines to exchange data with several different external devices. If you have direct access to the bus, this is always the fastest way to do data transfer. In fact, most buses have extra control lines that enable Direct Memory Access (DMA) controllers to exchange data with a computer's Random Access Memory (RAM) sequentially to minimize the CPU's operations. This produces the data transfers almost as fast as the bus clock.

However, most modern operating systems *multitask*, which is to say that they run several computer programs at once, jumping between each program as the need arises. Sometimes, the CPU will copy the contents of one part of its memory onto an external hard disk so that it can temporarily use that part to do something else. In the old days of small memory, this process, called *overlaying* or *virtual memory*, was often necessary even to run a single program. You, as the programmer, were required to remember what was where, so that you never used a variable that was not actually in memory. Modern operating systems now do that bookkeeping for you, although all the switching back and forth slows down the processor's speed. The net result of multitasking is that most operating systems do not let you have direct control of any specific part of memory or any specific external device address. If you want that control, you must write a *device driver*, which is a separate program that handles communications between your device and the operating system. Then, other programs will use the external device by calling the device driver. This is a continuing problem, since every time the operating system changes, you must write a new device driver for each external device.

An alternative is to communicate through a universal and simple bus, and require that the operating system always come with a standard driver for that bus. This slows down your data transfer, but it makes it far easier to implement a new external device, which is usually called a *peripheral device*.