CHAPTER 3
BASIC COMPUTER ORGANIZATION

Section 1. Introduction

In Chapter 1 we studied circuit design at what is commonly referred to as the digital logic (or gate) level. Here the fundamental unit for processing is a binary, digital, electronic signal, and the basic components of circuits are gates. Connections among gates made with individual conducting lines.

As part of our study of digital logic circuits, however, we designed circuits capable of accepting multi-bit inputs, producing multi-bit output, and undergoing multi-bit changes of state. Among some of these circuits are multi-bit gates, multiplexers, decoders, bit-sliced adders, various forms of registers (parallel in/out, counters, shift registers), etc. These circuits are among the fundamental components for a level of circuit design immediately above the digital logic level in a digital system design hierarchy. In recognition of the omnipresence of registers at this level, it is commonly referred to as the *register level* (or *register-transfer level*). In Chapter 2 we discussed the organization of main memory, one of the critical components of a computer system. In Chapter 4 we shall discuss the organization and design of a simple central processing unit,

In this chapter we will examine how these fundamental system components can be brought together in a basic computer system. Missing from our discussion, however, will be any detailed discussion of any of the input/output (I/O or IO) devices or secondary storage devicesof a computer system. We do this because there are just too many varieties of such devices. As a result we just note their presence in a system and address other issues as they are pertinent to our discussion. Sections 2.2 and 2.3 of your textbook describe many of these devices and issues involved in transferring data among them and memory. We leave this for you to read to become more familiar with these devices. We will discuss some of the issues involved with data transfers in this chapter, however.

Recall that the central processing unit (CPU) performs many of the processing tasks of the of the computer system and generally controls the system. Memory is used to store programs being executed by the CPU as well as any data immediately needed by, or generated by, the programs. The IO devices generally allow the CPU to interact with the computer's environment (where by “environment” we mean anything other than memory). The overall functioning of a computer system involves the transfer of data among the system components and the exchange of control signals to coordinate these transfers.

While it is possible to organize the connections among the register-level components so that each pair of elements has its own connections, as the number of components increases the number of dedicated connections soon becomes unwieldy. Instead register-level design uses sets of shared connection, known as *buses*, for transferring data to or from an associated set of register-level components. We will discuss buses in more detail in the next section, but for now we show some possible interconnection structures for computer systems.

Section 2. Buses

As we just noted a *bus* is a common electrical pathway between two or more devices. When applied to computer system (and also the design of computer components) buses are intended to transfer all bits of an n-bit word from a specified source to a specified destination, or destinations. Although our emphasis in this chapter is on data transfers at the register level, bus structures are applicable for data transfers at both the register level and at the system level. While data transfers among system level components are generally more complex because of the greater variety of system level components, many of the issues we consider at the register level also exist at the system level and are dealt with similarly.

A bus may be *unidirectional* (capable of word transfers in one direction only) or *bidirectional*. Although there are various approaches to bus design, in any bus design we can identify the following types of lines:

1. *data lines* for carrying the data being transferred between the registers.

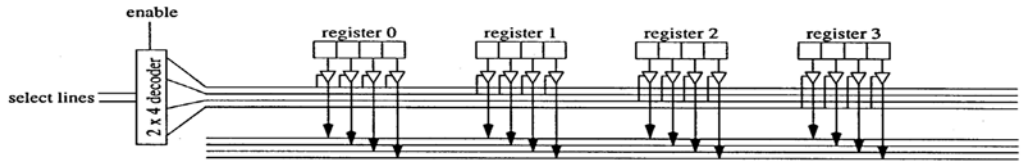
2. *address* (or select) *lines* to indicate the source/destination registers of a transfer

3. *control lines* to control access to, and the use of, the data and address lines. Minimally the control lines indicate the direction of the data transfer (from source to bus, or from bus to destination). In addition there may be an additional line/lines to coordinate the activities of a transfer.

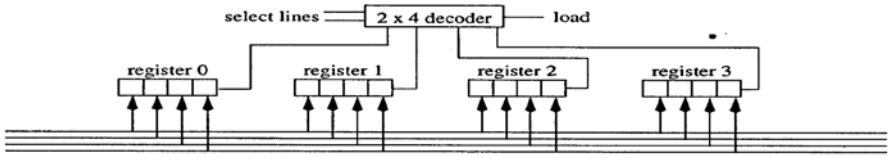
We now indicate how it is possible for multiple devices to be attached to common data lines without causing their respective signals to become mixed. We shall focus exclusively on connecting a number of registers (rather than entire components) to common data lines. This is not unnecessarily restrictive for our discussion since in practice all transfers of data within a computer system take place at the register level.

Implementation of Register-level Buses

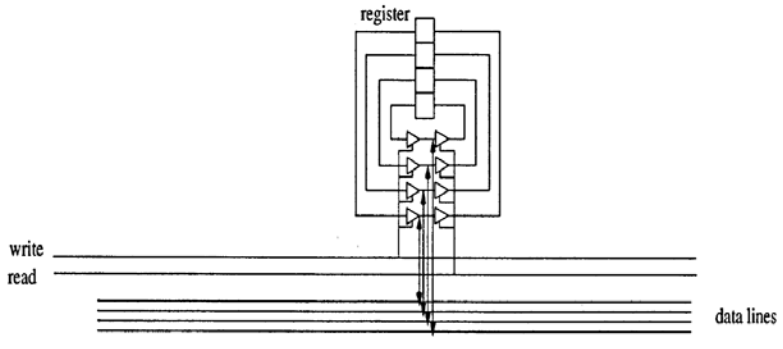
The first problem we encounter is: If several potential sources are physically connected to the bus, how can we keep the data from all of them from becoming intermingled when the data from only one source is to be placed on the bus? A solution is to attach the output lines from each device to the data lines of the system bus with *tri-state devices* (typically tri-state buffers or tri-state inverters). We introduced tri-state devices in the previous chapter.



Now we show how it is possible to transfer data from a single register to a bus. Transferring data from a bus to a register can be done without using tri-state devices, as shown below



For bi-directional transfers, however, a more complex arrangement is required. We show below the connections for one register. For multiple registers the output of as decoder (for register selection) could be ANDed with the write line and with the read line to read from or write to a selected register.



Section 3. Bus Operations

In this section we examine more technical issues related to bus implementations. We begin by making a distinction between those devices that can initiate bus transfers, so-called *bus masters*, and other devices that respond to request – *bus slaves*.

- Some devices may be bus masters in some instances and bus slaves in others.
- See the textbook (page 189) for examples of bus masters and bus slaves.

Bus Clocking – Synchronous and Asynchronous Buses

Buses are classified as either synchronous or asynchronous

- *Synchronous bus* – one of the control lines is a (master) clock line which receives its signals from a clock source (typically a crystal oscillator). All bus activities between devices attached to the bus are coordinated with pulses (clock cycles) on this line and take an integral number of such cycles. Note that typically some activities take place on the rising edge of a clock pulse, while others take place on the falling edge.

  One complication that must be considered in using a synchronous bus is that of *clock skew*. This arises when the lengths of some the paths between two different data elements are sufficiently long that there are delays between the times when a clock signal reaches the different elements. If this delay is significant it may be possible for one element to change and cause input values to a second element to be changed before that element has received the clock signal.

- *Asynchronous bus* – does not use a clock line

Example – Asynchronous bus transfer from a master device to a slave device – a read operation from memory to a CPU.

Again using general concepts, the activities involved in performing a memory read to a CPU over an asynchronous bus are as follows:

1. The CPU places an address on the bus' address lines (ADDR)
2. The CPU indicates the address is in memory (MREQ = 0) and that a read operation will take place (RD = 0).
3. The CPU (as transfer master) asserts a master synchronization signal (MSYNC) to alert the slave device (memory) to commence the transfer
4. Memory reads data from the appropriate word and places it on the bus' data lines.
5. Once the data is on the data lines, memory activates a slave synchronization signal (SSYNC) to let the CPU know that data is available.

Example – Synchronous bus transfer from a master device to a slave device -- a read operation from memory to a CPU.

The accompanying handouts illustrate, and then describe a protocol for transferring synchronously from memory to a cpu. Note that some events occur with rising edges of clock pulses and some with falling edges.

Broadly speaking, the activities involved in performing a memory read to a CPU over a synchronous bus are as follows:

1. The CPU (master) places an address on the bus' address lines (ADDR). This address becomes active with the rising edge of the clock
2. Control lines for the device (in this case $\overline{MREQ}$ and desired operation ($\overline{RD}$) is asserted (low in our cases).

- Note, in some buses there may be special lines such as MEMR (memory read), MEMW (memory write), IOR (IO device read), IOW (IO device write), etc. for this task.

3. The CPU waits for an appropriate number of clock pulses for the memory operation to complete. A "wait state" signal may be activated so the CPU can (continue to) wait.
4. Memory (the slave device) reads data from the appropriate word and places it on the bus' data lines.
5. Once the memory data has been read into a CPU register the read and memory request lines can be returned to their unasserted values.

Because all of these activities must happen in coordination with clock cycles, the frequency of the master clock, the read/write times for memory, and other timing factors must all be taken into account.

The accompanying handout gives a detailed discussion of a synchronous memory read transfer..

Block Transfers

Where it is necessary to transfer multiple words between a master device and a slave device it is possible to modify the synchronous scheme to allow a count value to be transferred to the slave device and then to transfer a block of data values at a time (say one word per clock cycle). This can reduce the time to transfer a block of data significantly compared to transferring the block one word at a time.

Bus Arbitration

Bus arbitration is a mechanism for resolving simultaneous requests from several bus masters for control of the bus for a data transfer. Special bus arbitration circuitry is needed for this, often as a special chip.
In the simplest scheme a master device asserts a bus request line that goes to the bus arbiter. Bus requests from each possible master go to the same request line to the bus arbiter, Upon recognizing that a bus request signal has arrived, the bus arbiter returns a bus grant signal on a separate line to which the bus masters are attached in a daisy chain manner (see page 185 of the text). In this way the device that sent a bus request that is closest to the bus arbiter in this daisy chain arrangement will be given control of the bus.

- In a variation of this arrangement, the bus arbiter may accept signals on several (prioritized) bus request lines, each of which has an associated bus grant line. As in the above case, the devices associated with a given bus request/grant pair are attached to the bus grant line in a daisy-chain manner. Simultaneous requests for a bus are resolved first by the priority of the request line and secondarily (if necessary) by position in the daisy chain.
- Because of the role of a bus arbiter, the above mechanism is known as *centralized bus arbitration*. It is also possible to use a simpler, decentralized scheme in which the devices compete among themselves and monitor a busy line before attempting to gain bus access..

Bus Standards

Computer components such as CPUs, memory, and various IO devices will be manufactured by different companies, each having their own unique internal characteristics. In order that they might communicate among themselves via buses within a computer system, however, various bus standards (or protocols) have been established and used over the years. These standards include not only specifications for the number of address and data lines, and types of control lines present, but also encompass mechanical and electrical specifications so that when the bus is incorporated into printed circuit boards the physical connectors, voltages, and timing signal will all be compatible. Here we review briefly some noteworthy bus standards:

- ISA (Industry Standard Architecture) bus: A bus standard adopted by most of the personal computer industry in the early 1980s. Originally it had 20 address lines and 8 data lines, later expanded to 16 data lines. It used an 8.33 MHz clock. Among its control signals are ones for asserting memory reads, memory

writes, IO reads, IO writes, etc. ISA was extended to 32 bits in the late 80s, giving us Extended Industry Standard Architecture (EISA). The maximum data transfer rate (bandwidth) on ISA was 16.7 MB/sec, and with EISA it was 33.3 MB/sec.

- PCI (Peripheral Component Interface): A standard developed in the early 1990s to have a higher bandwidth than EISA. It is intended for attaching high-speed peripherals to a computer. Later versions of PCI supported up to 133MHz clock speeds and either 32 bit or 64 bit address and data lines (shared). This standard is described in your book in detail starting on page 204.
- USB (Universal Serial Bus): A standard developed in the early 1990s for attaching low-speed IO devices (such as keyboard, mice, scanners, etc.) to a computer. It is intended for serial transmission (bit-at-a-time) and uses only four lines – two for data, one for power, and one for ground. UISB 2.0, adopted in 1998, allows a data transfer speed of up to 480 Mbps.