

Chapter 6: Bus organization and memory design

Objectives

When you are finished with this chapter you will be able to:

- understand the need for bus organization
- use the principles of tri-state logic to design bus oriented systems,
- design the memory decoding circuitry for a modern microprocessor; and
- design a memory system of any width or depth using the address, data and control I/O pins of modern memory circuits.

Bus Organization

In lesson 2 on Computer Hardware you learned that the outputs of logic gates connect to the inputs of other logic gates. It is a general rule; outputs must go to inputs. You can simultaneously connect one output to a number of inputs so that when the output changes its state, all of the inputs connected to that output see the change at the same time (Of course, it still must be within the constraints imposed by the speed of light.). This is called a **one to N** circuit configuration.

You cannot, however, connect a number of inputs together unless they are tied to an output, because there needs to be a signal of some kind present to drive the inputs to either the 1 or 0 states. Without an output to drive them, inputs will tend to drift around, randomly switching between 1 and 0, creating noisy signals in the computer. In general, all unused inputs are “tied” to either ground or the power supply voltage (Vcc). As you’ll see shortly, a similar problem exists if we try to connect two or more outputs together. What do you think happens if an output that is in the logic one state is connected to an output that is in the logic zero state? Do we end up with the average, 0.5?

In order to see what might happen, you can take a 1.5 volt battery (such as an AA or AAA) cell and, with a piece of wire, touch the positive terminal of the battery to the negative terminal. If you have made good electrical contact, you should see sparks and perhaps, even a puff of smoke. In general, computer designers do not like little puffs of smoke coming from inside of their computers. So hopefully, this experiment has convinced you that connecting outputs together is not a good idea.

As you might surmise from the above discussion, connecting two outputs together is similar to an electrical short circuit. In fact, we might actually damage the circuit elements because each one is trying to force the other to change, sort of like when I first got married. But that’s another story. Figure 6.1 introduces us to the basic dilemma of computer design; there are a lot of wires running around.

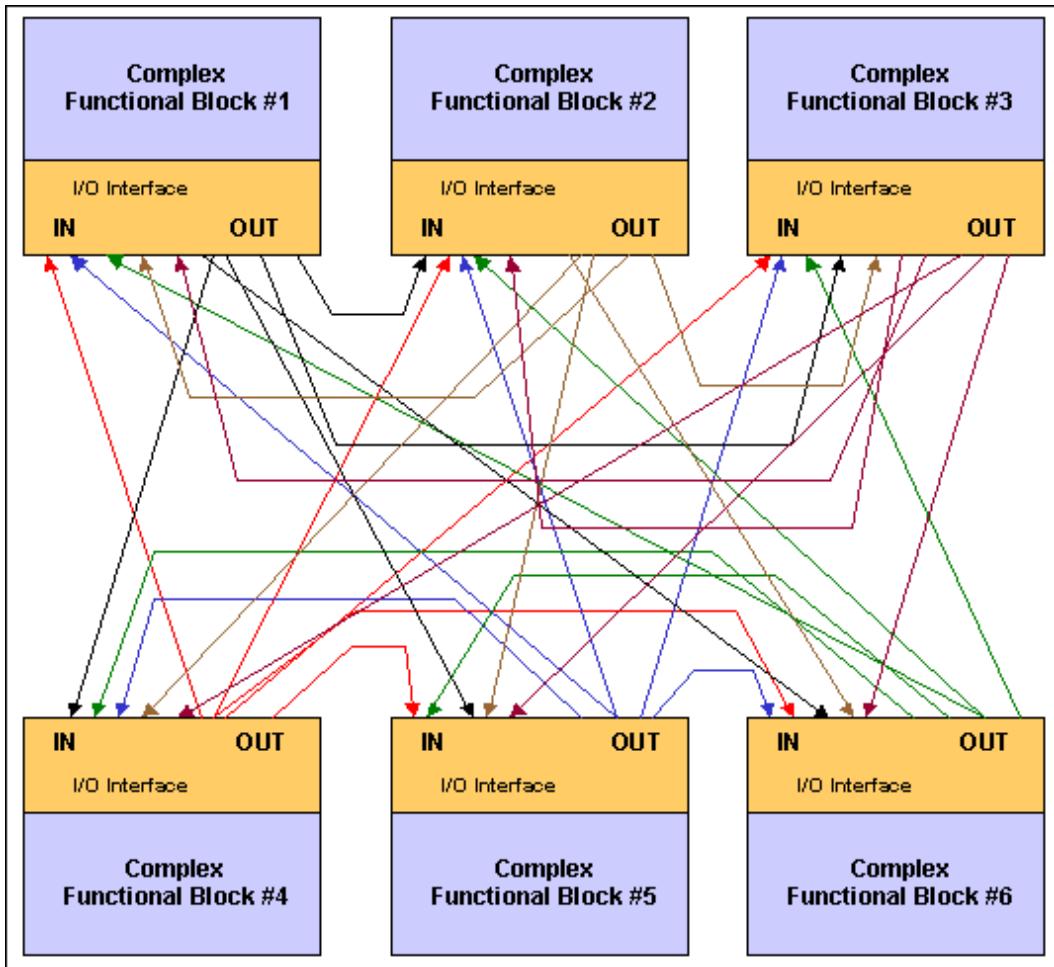


Figure 6.1: Computer organization based upon point-to-point wiring. The diagram shows the number of signal lines required to connect one data bit between six different internal components of a typical computer system.

Figure 6.1 illustrates the “rat’s nest” of wires that are required to interconnect six computer circuits to each other. In this example we don’t care what these functional blocks really are, we’re just interested in how they’re wired together. Also, the interconnection is designed to show just one data bit. We would have to multiply this maze of wires by 32 for a real computer system and modern computers have many more than six function blocks that need to be interconnected.

Each functional block connects its output signal to the five other blocks’ inputs. The color of the wire indicates which functional block is sending the signal and the arrowheads indicate the direction of information flow. Within each functional block there must be some kind of input/output (I/O) interface and control circuit so that the computer can synchronize which block is sending and which block is receiving, because it is customary that only one block sends and only one receives at any point in time. Thus, all the blocks might have 1’s and 0’s on their outputs, but at the right time, one of the blocks should be listening to only one of the possible outputs, which means that the I/O interface circuit must have some way to decide which functional block it’s supposed to listen to, and which to ignore.

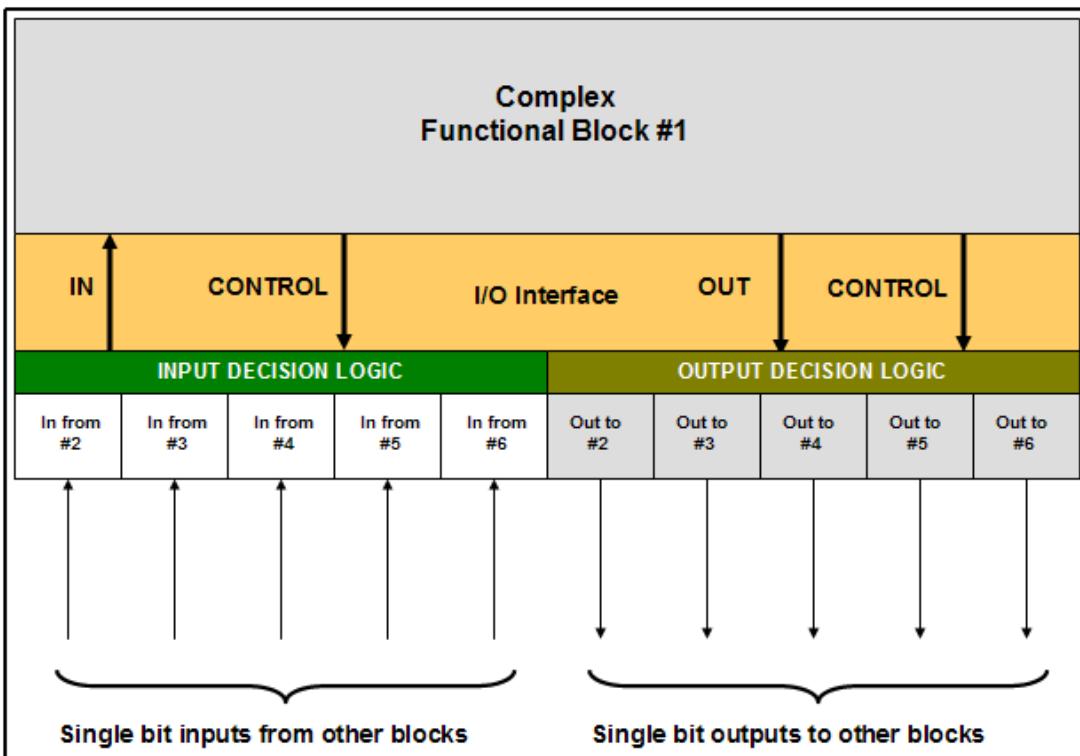


Figure 6.2: Detailed block view of circuitry necessary to implement a single bit communications protocol for 6 functional blocks using point-to-point wiring.

Figure 6.2 shows the I/O organization in somewhat more detail. The output from each block is simultaneously driven to all of the inputs of the other blocks in a "1 to 5" organization. On the input side, each block must have logic to decide which of the outputs it is going to accept at any point in time. Thus, it needs the decision logic, called a ***multiplexer***, or MUX, to do a "5 to 1" reduction so that the correct data can be read by the input and then passed into the logic of the complex functional block. This is a lot of complexity. We'll be drowning in wires if we can't come up with a better solution. What would be ideal is if we could have the circuit connected as shown in figure 6.3.

In figure 6.3, we've managed to dramatically simplify our design. Only one wire comes out of each functional block and it carries the data out as well as the data in. Remember, this is still only one bit of data. This may be simple, but you're correct in observing that it won't work because we've got outputs and inputs all tied together. This is exactly the problem that we considered earlier. We're trying to send a 1 from block #1 to block #6 (green dashed arrow). All of the other outputs are 0 (red dashed arrows) so the data cannot be sent. We need to be able to somehow manage the traffic flow (note the clever clip art) through the circuit. The ways in which we'll do it is by organizing our data paths into ***busses***, and then make use of the fourth fundamental gate structure that we considered in chapter 2, the tri-state buffer.

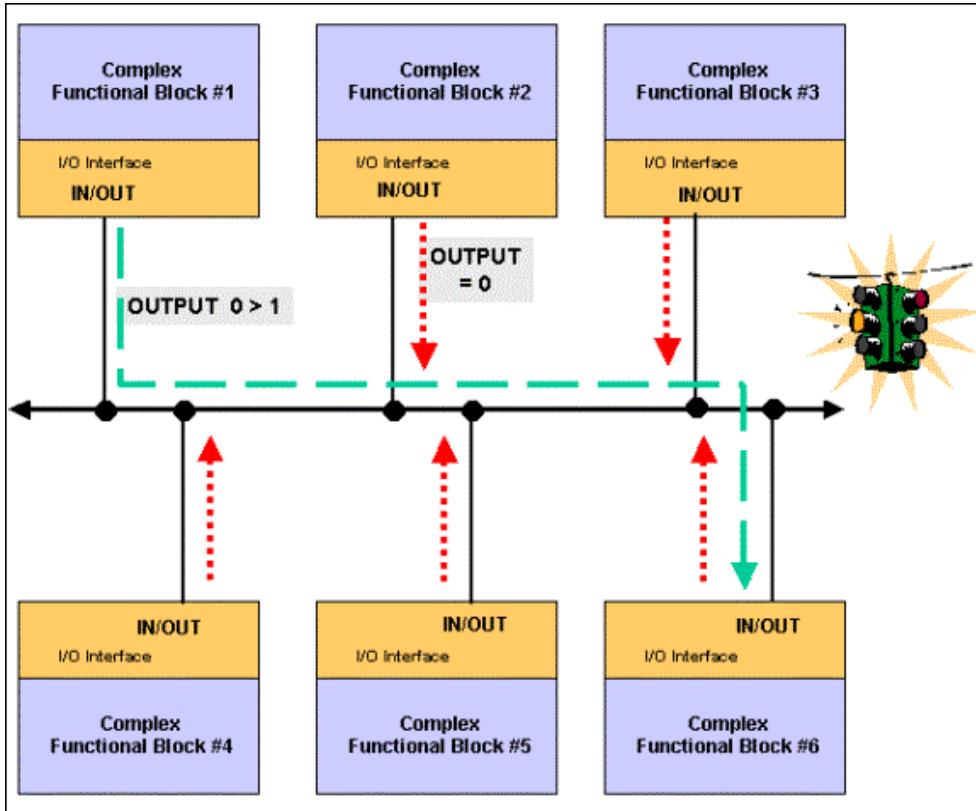


Figure 6.3: Using a bus protocol to interconnect the functional blocks. A problem has been created because inputs and outputs are tied together.

Busses were invented as a way to simplify the organization and flow of data within the computer system. We use busses to allow many devices to connect to the same data path at the same time. A bus is a grouping of similar signals. We'll look at busses in more detail in a moment, but for now, let's focus on figuring out how to connect those pesky signals together in the first place without blowing a fuse.

In most computers there are three main busses called the ***address bus***, the ***data bus***, and the ***status bus***. We'll be discussing these busses in more detail in the next chapter, but for now, let's consider the problem of the outputs once again. Figure 6.4 shows us the dilemma. The AND gate on the left has a 1 on both inputs, so its output is a 1, or 5 volts.

The logic gate on the right has a 1 and a 0 for its two inputs so its output is a 0, or 0 volts. The resulting signal on the bus is indeterminate. In order to fix the problem we need some way to separate the logic functions from the interface to the bus. We can do that by dividing the circuits of the logical devices that connect to the bus into two parts: the logic function and the bus interface unit. We saw the need for the interface circuit in figures 6.1 and 6.2. However, in this case the bus interface unit will not be a multiplexer, which selected among the various input signals coming into the device; the interface logic will be much simpler than that. It will be a simple switch that connects or disconnects the outputs from the bus.

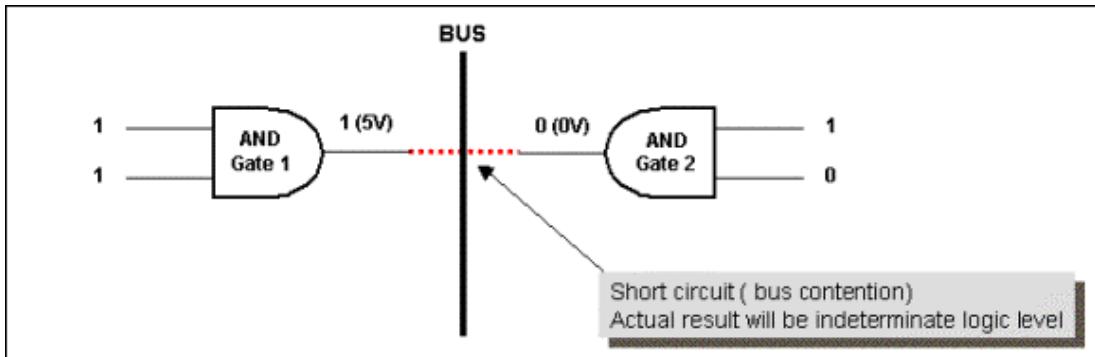


Figure 6.4: Problem with trying to tie two logic gate outputs to the same data bus. What is the resulting logic level that we would see on the bus?

Figure 6.5 shows this schematically. The interface to the bus is an electronic switch. As you already know, that electronic switch is the tri-state buffer. A bus control signal can rapidly activate the switch to either connect or disconnect the output from the bus. If all of the outputs of the various functional blocks are disconnected except one, then the one output that is connected to the bus can “drive the bus” either high or low. Since it is the only “talker” and every other device is a “listener” (one to N) there will not be any conflicts from other output devices. All of the other outputs, which remain disconnected by their electronic switches, have no impact on the state of the bus signal.

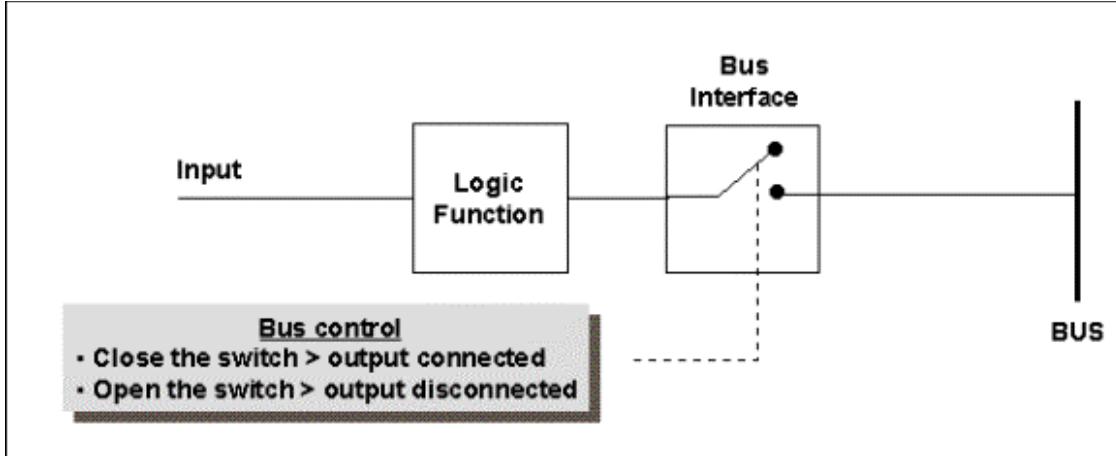


Figure 6.5: Mechanical switch representation of bus interface logic.

As you know, with the tri-state buffers, we aren't really breaking the electrical continuity of the gates to the bus; we are simply making a rapid change in the electrical conductivity of a small portion of the circuit that resides between the logical output of the gate and the bus signal that it is connected to the circuit. In figure 6.5, we simplify this bit of circuit magic by drawing the connect/disconnect portion of the circuit as if it was a physical switch, like the light switch on the wall of a room. However, keep in mind that we cannot open and close a real mechanical switch nearly as quickly or as cleanly as we can go from high impedance (no signal flow) to low impedance (connect to the bus).

It is generally true that the bus control signal will be active low. Whatever logic state (1 or 0) we want to place on the bus, we must first bring the bus control for that gate, function block, memory cell, etc. low to connect the gate to the bus so the output of the logical device can be connected to the bus. This signal has several names. Sometimes it is called ***output enable (/OE)***, ***chip enable (/CE)*** or ***chip select (/CS)***. These signals are not all the same, and, as we'll soon see, may perform different tasks within the device, but they all have the common property of disabling the output of the device and disconnecting it from the bus.

Referring back to figure 2.6, the diagram of the tri-state logic gate, we see that when the ~OE, or bus control signal, is LOW, the output of the bus interface (BUS I/F) portion of the circuit follows the input. When the /CS signal is HIGH, the output of the BUS I/F goes to Hi Z, effectively removing the gate from the circuit. One final point needs to be emphasized. The tri-state part of the circuit doesn't change the logical state of the gate, or memory device, or whatever else is connected to it. Its only function is to isolate the output of the gate from the bus so that another device may take control and send a signal on the bus.

Let's now look at how we can build our single bit bus into a real data bus. Refer to figure 6.6. Here we see that the bus is actually a grouping of eight similar signals. In this case it represents eight data bits. So, if we were to take apart our VCR and look at the microprocessor inside of it, we might find an 8-bit microprocessor inside. The number of bits of the data bus refers to the size of a number that we can fetch in one operation. Recall that 8-bits allow us to represent numbers from 0 to 255, or 2^8 .

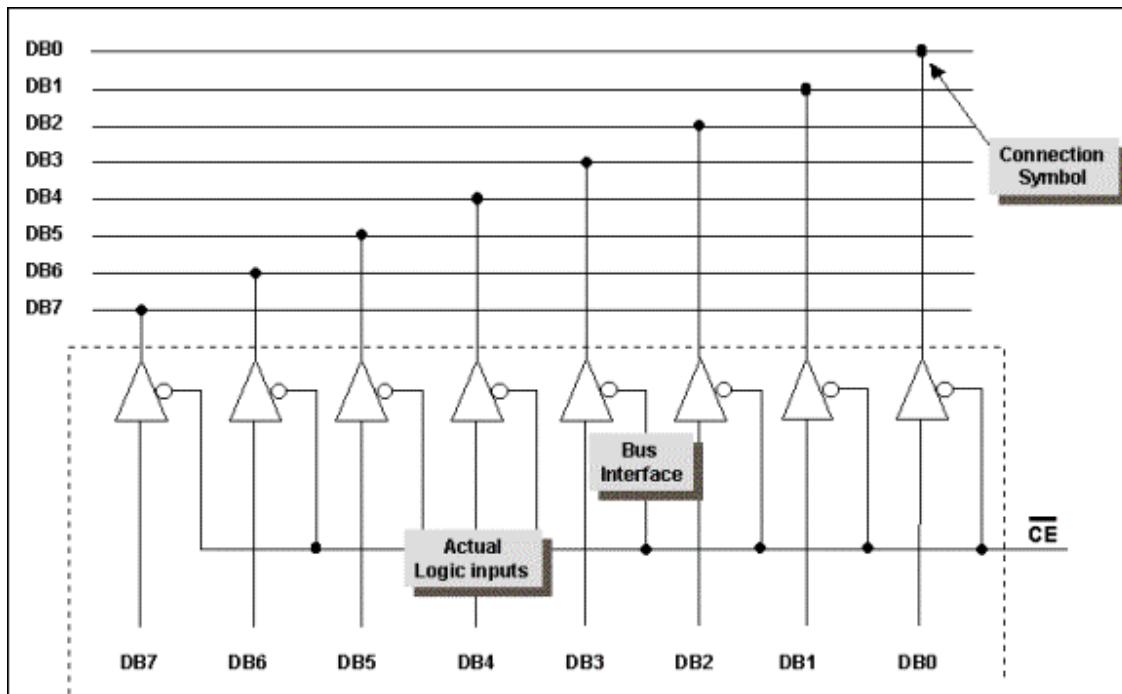


Figure 6.6: Tri-state bus organization for an 8-bit wide data bus.

The bus itself is still made up of eight individual signal wires that are electrically isolated from each other. This is important to remember because, in order to keep our drawings simple, we'll usually draw a bus as one line, rather than 8, 16, or 32 lines. Each wire of the bus is labeled with the corresponding data bit, DB0 through DB7, with DB0 representing the number 2^0 , or 1, and DB7 representing the number 2^7 , or 128.

Inside the dotted line is our device that is connected to the bus. Each data bit inside the device connects with the corresponding data line on the bus through the tri-state buffer. Notice how the $\sim\text{CE}$ signal is connected to all of the tri-state buffers at once. By bringing $\sim\text{CE}$ LOW, all eight individual data bits are simultaneously connected to the data bus. Also note that the eight tri-state buffers (drawn as triangles with the circle on the side) in no way change the value of the data. They simply connect the signals on the eight data bits, DB0 . . . DB7, inside of the functional block, to the corresponding data lines outside of the functional block. Don't confuse the tri-state buffer with the NOT gate. The NOT gate inverts the input signal while the tri-state buffer controls whether or not the signal is allowed to propagate through the buffer.

Figure 6.7 takes this concept one step further. Here we have four 32-bit storage registers connected to a 32-bit data bus. Notice how the data bits, D0 . . . D31 are drawn to show that the individual bits become the 32-bit bus. Figure 6.7 also shows a new logic element, the block labeled "2:4 Address Decoder." Recall that we need some way to decide which device can put its data onto the bus because only one output may be on at a time. The decoder circuit does just that. Imagine that two signals, A0 and A1, come from some other part of the circuit and are used to determine which of the 4, 32-bit registers shown in the figure should be connected to the bus at any given time. The two "address" input bits, labeled A0 and A1, give us 4 possible combinations. Thus, we can then create some relatively simple logic gate circuit to convert the combinations of our inputs, A0 and A1, to 4 possible outputs, the correct chip select bit, /CS0, /CS1, CS2 or /CS3. This circuit design is a bit different from what you're already accustomed to because we want our "TRUE" output to go low, rather than go high. This is a consequence of the fact that tri-state logic is usually asserted with a low-going signal.

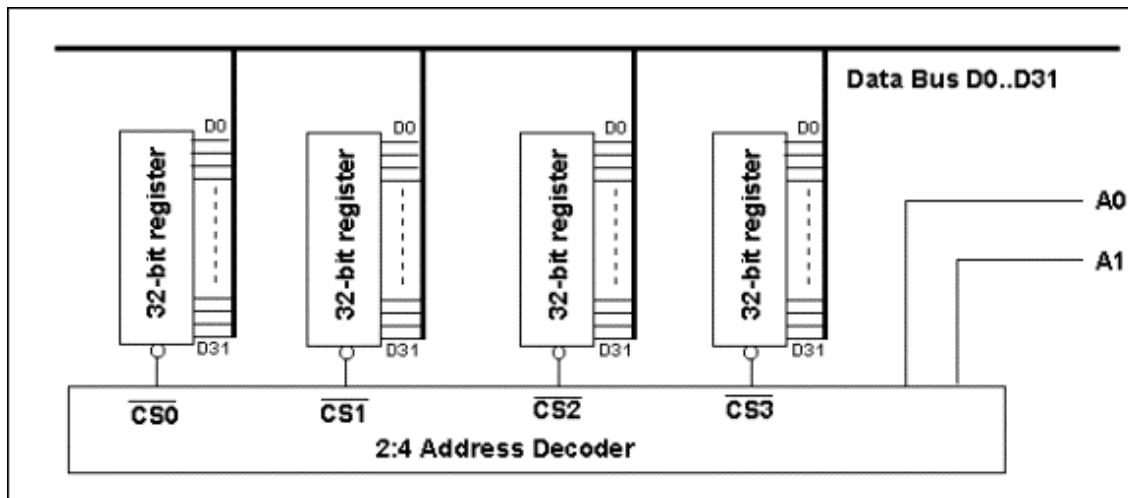


Figure 6.7: Four 32-bit storage registers connected to a bus. The 2:4 decoder takes the 2 input variables, A0 and A1, and selects only 1 of the 4 outputs, $\sim\text{CS}0..\sim\text{CS}3$ to enable.

Table 6.1 is the truth table for the 2:4 decoder circuit shown in figure 6.7.

As we've previously discussed, it is common to see the circle drawn at the input to the tri-state gate. This means that the signal is **active low**. Just as the circle on the outputs of the inverter gate—NAND gate and NOR gate—indicate a signal inversion, the circle on the input to a gate indicates that the signal is asserted (TRUE) when it is LOW. This goes back to our earlier discussion about 1 and 0 being rather arbitrary in terms of what is TRUE and what is FALSE.

A0	A1	$\sim CS_0$	$\sim CS_1$	$\sim CS_2$	$\sim CS_3$
0	0	0	1	1	1
1	0	1	0	1	1
0	1	1	1	0	1
1	1	1	1	1	0

Table 6.1: Truth table for a 2:4 decoder

Before we move on to look at memory organization, let's revisit the Algorithmic State Machine from the previous chapter. With the introduction of the concept of busses in this chapter, we now have a better basis of understanding to see how the operation of a bus-based system might be controlled by the state machine. Figure 6.8 shows us a simplified schematic diagram of part of the control system of a computing machine. Each of the registers shown, Register A, Register B, Temporary Register, and the Output Register have individual controls to read data into the register on a rising edge to the clock inputs and an Output Enable ($\sim OE$) signal to allow the register to place data on the common data bus that connects all of the functional elements inside of the computer.

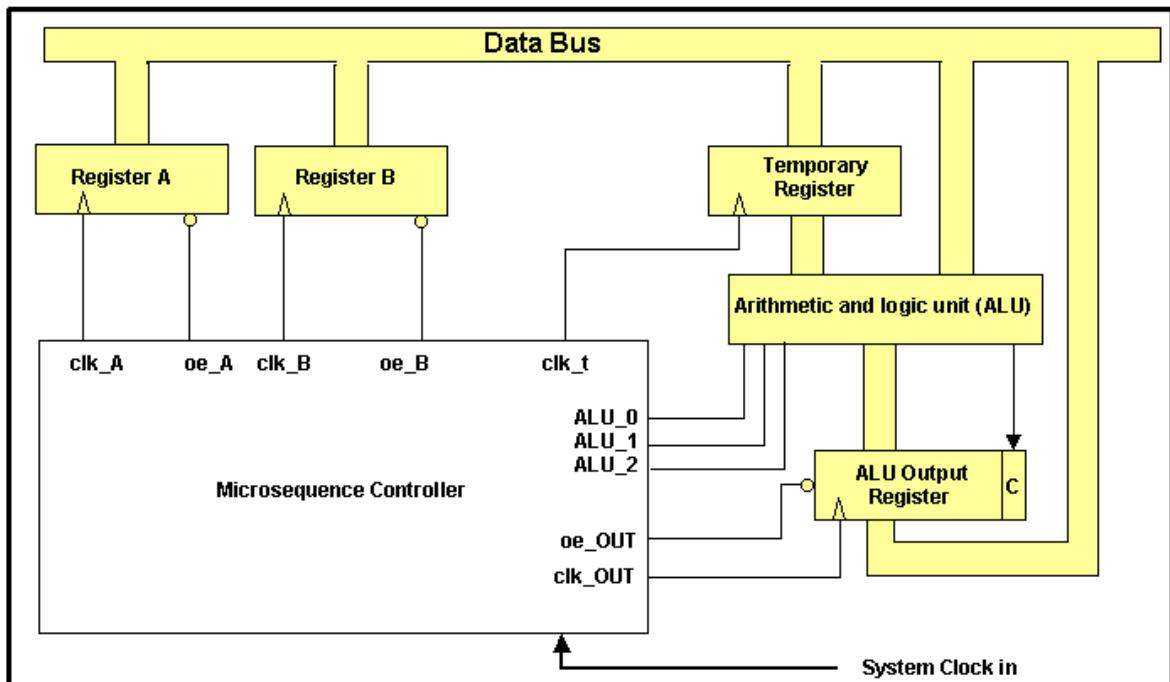


Figure 6.8 Schematic diagram of part of a computing engine. The registers are interconnected using tri-state logic and busses. The Microsequence Controller must sequence the **Output Enable ($\sim OE$)** of each register properly so that there are no bus contentions.

In order to place data into Register A we would first have to enable another source of the data, perhaps memory, and put the data onto the Data Bus. When the data is stable, the clock signal, clk_A, would become active with a low-to-high transition and the data would be stored in Register A. Remember that Register A is just an aggregate of D-FF's with a common clock. Now suppose that we want to add the contents of Register A and Register B, respectively. The Arithmetic and Logic Unit (ALU) requires two input sources to add the numbers together. It has a temporary register and an output register associated with it because the ALU itself is an asynchronous gate design. That is, if the inputs were to change, the outputs would also immediately change because there is no D register there to synchronize its behavior.

Figure 6.9 shows a portion of the truth table that controls the behavior of the Microsequence Controller. In order to add two numbers together, such as the contents of A and B, we might issue the *assembly language instruction*:

ADD B,A

This instruction tells the computer to add together the contents of Register A and Register B and place the result back into Register A. If there is any carry generated by the addition operation, it should be placed in the carry bit position, which is shown as the "C" bit attached to the ALU Output Register. Thus, we can begin to see that we would need several steps in order to add these numbers together. Let's describe the process in words:

1. Copy the data from memory in the A Register. This will be operand #2.
2. Copy operand #1 from memory into the B register.
3. Move the data from the A register into the temporary register.
4. Move the results of the addition in the ALU into the ALU Output Register and Carry bit.
5. Move the data from the Output register back into register A.

Referring to the portion of the truth table shown in Figure 6.9 we can follow the instruction's flow through the controller:

- Clock pulse 1: There is a low to high transition of the Register A input clock. This will store the data that is currently being held on the Data Bus in the register.
- Clock pulse 2: There is a low to high transition of the Register B input clock. This will store the data that is being held on the Data Bus into Register B.
- Clock pulse 3: The input to clock B returns to 0 and the Output Enable signal for Register A becomes active. This puts the data that was previously stored in Register A back onto the Data Bus. There is no effect of the clock signal to Register B going low because this is a falling edge.
- Clock pulse 4: There is a rising edge of the clock to the Temporary Register. This stores the data that was previously stored in Register A and is now on the Data Bus. The Output Enable signal of the A register is turned off and the Output Enable of Register B is turned on. At this point the data stored in Register B is on the Data Bus and the Data stored in register A is stored in

the Temporary Register. The ALU input signals, ALU_0, ALU_1 and ALU_2 are set for addition¹, so the output of the ALU is the sum of A and B plus any carry generated.

- Clock pulse 5: The sum is stored in the Output Register on the rising edge of the clk_out signal.
- Clock pulse 6: The Output Enable of B register is turned off, the clock input to the ALU Output Register is returned to 0, and the data in the ALU Output Register is put on the Data Bus.
- Clock pulse 7: The data is clocked into the A register and the Output Enable of the ALU Output Register is turned off.
- Clock pulse 8: The system returns to the original state.

clock	clk_A	oe_A	clk_B	oe_B	clk_T	clk_OUT	oe_OUT	ALU_0	ALU_1	ALU_2		clk_A	oe_A	clk_B	oe_B	clk_T	clk_OUT	oe_OUT	ALU_0	ALU_1	ALU_2
1	0	1	0	1	0	0	1	0	0	0		1	1	0	1	0	0	1	0	0	0
2	1	1	0	1	0	0	1	0	0	0		0	1	1	1	0	0	1	0	0	0
3	0	1	1	1	0	0	1	0	0	0		0	0	0	1	0	0	1	0	0	0
4	0	0	0	1	0	0	1	0	0	0		0	1	0	0	1	0	1	0	0	0
5	0	1	0	0	1	0	1	0	0	0		0	1	0	0	0	0	1	1	0	0
6	0	1	0	0	0	1	1	0	0	0		0	1	0	1	0	0	0	0	0	0
7	0	1	0	1	0	0	0	0	0	0		1	1	0	1	0	0	0	1	0	0
8	1	1	0	1	0	0	1	0	0	0		0	1	0	1	0	0	1	0	0	0

Figure 6.9 Truth Table for the Microsequence Controller of Figure 6.8 showing the before and after logic level changes with each clock pulse. The left-hand side of the table shows the state of the outputs before the clock pulse and the right hand side of the table shows the state of the outputs after the clock pulse. The areas shown in blue are highlighted to show the changes that occur on each clock pulse.

Now, before you go off and assume that you are a qualified CPU architect let me warn you that it's a lot more involved than this simple example might lead you to believe. However, the principles are the same. For starters, we never discussed just how the instruction, **ADD B,A** actually was transferred (*fetched*) by the computer and how the computer then decoded that this particular bit pattern (*op-code*) is an ADD instruction, and not something else. Also, we didn't discuss how we established the contents of the A and the B registers in the first place. However, for at least that portion of the state machine that actually does the addition process, it probably is pretty close to how the circuitry really works.

¹ * The ALU is a circuit that can perform up to 8 different arithmetic or logical operations based upon the state of the three input variables, ALU_0..ALU_2. In this example, we assume that the ALU code, 000, sets up the ALU to perform the operation of addition.

You've now seen two examples of how the state machine is used to sequence a series of logical operations and how these operations form the basis of the execution of an instruction in a computer. Without the concept of a bus-based architecture and the tri-state logical gate design, this would be very difficult or impossible to accomplish.

Memory System Design

We have already been introduced to the concept of the flip-flop. In particular we saw how the "D" type flip-flop could be used to store a single bit of information. Recall that the data present at the D input to the D-flop will be stored within the device and the stored value will be transferred to the Q output on the rising edge of the clock signal. Now, if the clock signal goes away, we would still have the data present on the Q output. In other words, we've just stored one bit of data. Our D-flop circuit is a memory cell.

Historically, there have been a large number of different devices used to store information as a computer's **random access memory**, or RAM. Magnetic devices, such as **core memories**, were very important until integrated circuits (IC) became widely available. Today, one IC memory device can hold 512 million bits of data. With this type of miniaturization, magnetic devices couldn't keep up with the semiconductor memory in terms of speed or capacity. However, memories based on magnetic storage have one ongoing advantage over IC memories; they don't forget their data when the power is turned off. Thus, we still have hard disk drives and tape storage as our secondary memory systems because they can hold onto their data even if the power is removed.

Many industry analysts are foretelling the demise of the hard drive. Modern FLASH memories, such as the ones that you may use in your PDA, digital camera or MP3 players are already at 16 gigabyte of storage capacity. FLASH memories retain their information even with power removed, and are faster and more rugged than disk drives. However, disk drives and tape drives still win on capacity and price per bit. At the time of this re-editing (spring of 2011) a modern disk drive with a capacity of 2 terabytes can be purchased for about \$80 if you are willing to send in all of the rebate information and the company actually sends you back your rebate check. But that's another story for another time.

Hard disks and tape systems are **electromechanical systems**, with motors and other moving parts. The mechanical components cause them to be far less reliable and much slower than IC memories, typically 10,000 times slower than an IC memory device. It is for this reason that we don't use our hard disks for the main memory in our computer system because they're just too slow. However, as you'll see in a later lesson, the ability of the hard drive to provide almost limitless storage enables an operating system such as Linux or Windows to give the user the impression that every application that is opened on your desktop always has as much memory as it needs.

In this section, we will start from the D-flop as an individual device and see how we can interconnect many of them to form a memory array. In order to see how data can be written to the memory and read from the memory along the same signal path (although not at the same instant in time), consider figure 6.10.

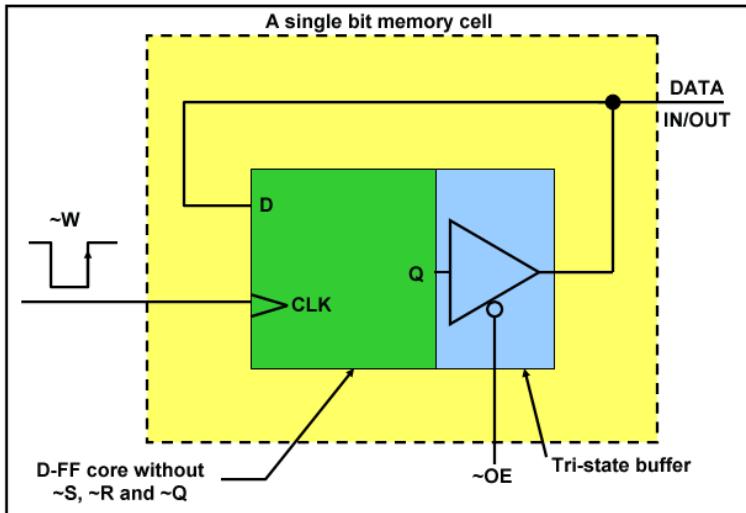


Figure 6.10 Schematic representation of a single bit of memory. The tri-state buffer on the output of the cell controls when the Q output may be connected to the bus.

The green box is just a slightly simplified version of the basic D flip-flop. We've eliminated the $\sim S$, $\sim R$ inputs and $\sim Q$ output. The blue box is the tri-state buffer, which is controlled by a separate $\sim OE$ (output enable) input. When $\sim OE$ is HIGH, the tri-state buffer is disabled, and the Q output of the memory cell is isolated (Hi-Z state) from the data lines (DATA I/O line). However, the data line is still connected to the D input of the cell, so it is possible to write data to the cell, but the new data written to the cell is not immediately visible to someone trying to read from the cell until the tri-state buffer is enabled. When we combine the basic FF cell with the tri-state buffer, we have all that we need to make a 1-bit memory cell. This is indicated by the yellow box surrounding the two elements that we've just discussed.

The write signal is a bit misleading so we should discuss it. We know that data is written into the D-FF on the rising edge of a pulse, which is indicated by the up-arrow on the write pulse ($\sim W$) in figure 6.10. So why is the write signal, $\sim W$, written as if it was an active low signal? The reason is that we normally keep the write signal in a 1 state. In order to accomplish a write operation, the $\sim W$ must be brought low, and then returned high again. It is the low-to-high transition that accomplishes the actual data write operation, but since we must bring the write line to a low state in order to accomplish the actual writing of the data, we consider the write signal to be active low. Also, you should infer from this discussion that you would never activate the $\sim W$ line and the $\sim OE$ lines at the same time. Either you bring $\sim W$ low and keep $\sim OE$ high, or vice versa. They never are low at the same time. Now, let's return to our analysis of the memory array.

We'll take another step forward in complexity and build a memory out of tri-state devices and D-flops. Figure 6.11 shows a simple (well, maybe not so simple) 16-bit memory,

organized as four, 4-bit nibbles. Each storage bit is a miniature D-flop that also has a tri-state buffer circuit inside of it so that we can build a bus system with it.

Each row of 4 D-FF's has two common control lines that provide the clock function (write) and the output enable function for placing data onto the I/O bus. Notice how the corresponding bit position from each row is physically tied to the same wire. This is why we need the tri-state control signal, /OE, on each bit cell (D-FF). For example, if we want to write data into row 2 of D-FF's, the data must be placed on the DB0 through DB3 from the outside device and the W2 signal must go high to store the data. Also, to write data into the cells, the /OE signal must be kept in the HIGH state in order to prevent the data already

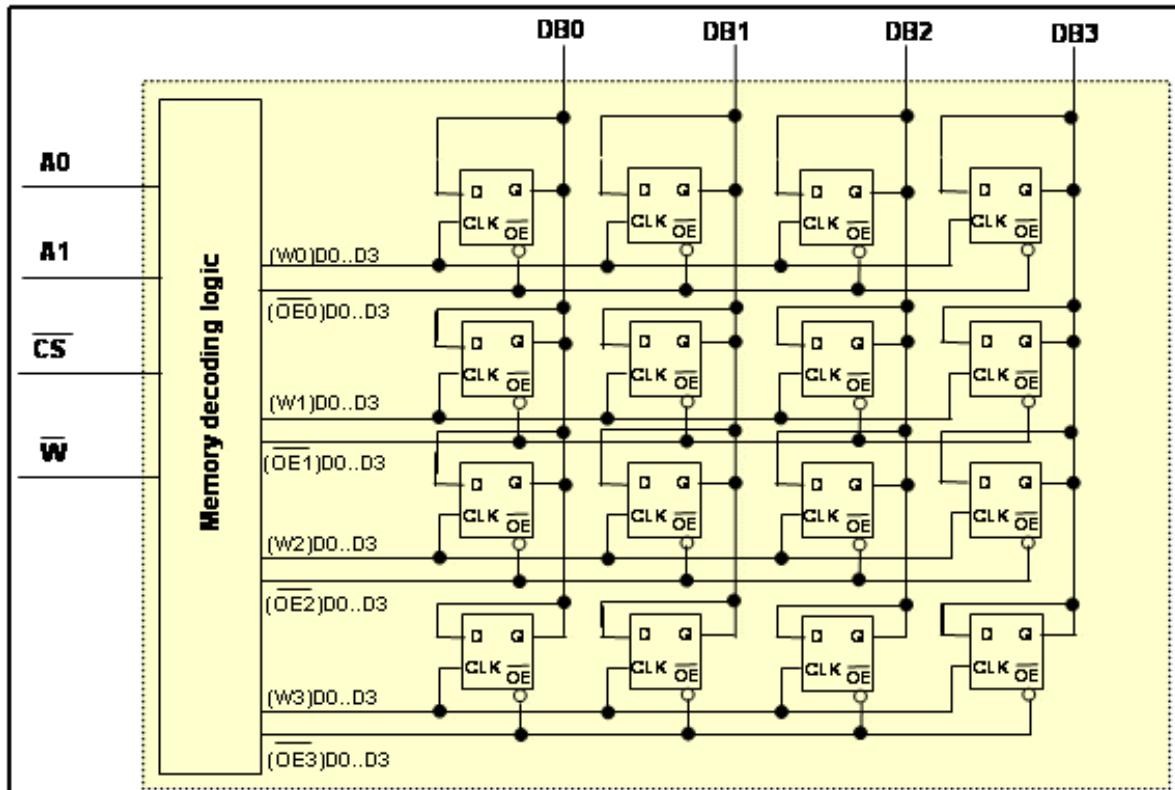


Figure 6.11 16-bit memory built using discrete "D" flip-flops. We would access the top row of the four possible rows if we set the address bits, A0 and A1 to 0. In a similar vein, $(A0, A1) = (1, 0)$, $(0, 1)$ or $(1, 1)$ would select rows 1, 2 and 3, respectively.

stored in the cell from being placed on the data lines and corrupting the new data being written into a cell.

The control inputs to the 16-bit memory are shown on the left of figure 6.11. The data input and output, or I/O, is shown on the top of the device. Notice that there is only one I/O line for each data bit. That's because data can flow in or out on the same wire. In other words, we've used bus organization to simplify the data flow into and out of the device. Let's define each of the control inputs:

A0 and A1	Address inputs used to select which row of the memory is being addressed for input or output operations. Since we have 4 rows in the device, we need two
------------------	--

address lines.

- | | |
|------------|--|
| ~CS | Chip select. This active low signal is the master switch for the device. You cannot write into it or read from it if ~CS is HIGH. |
| ~W | If the ~W line is HIGH, then the data in the chip may be read by the external device, such as the computer chip. If the ~W line is low, data is going to be written into the memory. |

The signal ~CS (chip select) is, as you might suspect, the master control for the entire chip. Without this signal, none of the Q outputs from any of the 16 D-FF's could be enabled, so the entire chip would remain in the Hi-Z state, as far as any external circuitry was concerned. Thus, in order to read the data in the first row, not only must $(A_0, A_1) = (0,0)$, we also need $\sim CS = 0$. But wait, there's more!

We're not quite done because we still have to decide if we want to read from the memory or write to it. If we want to read from it, we would want to enable the Q output of each of the 4 D-flops that make up one row of the memory cell. This means that in order to read from any row of the memory, we need the following conditions to be TRUE:

- READ FROM ROW 0 > $(A_0 = 0) \text{ AND } (A_1 = 0) \text{ AND } (\sim CS = 0) \text{ AND } (\sim W = 1)$
- READ FROM ROW 1 > $(A_0 = 1) \text{ AND } (A_1 = 0) \text{ AND } (\sim CS = 0) \text{ AND } (\sim W = 1)$
- READ FROM ROW 2 > $(A_0 = 0) \text{ AND } (A_1 = 1) \text{ AND } (\sim CS = 0) \text{ AND } (\sim W = 1)$
- READ FROM ROW 3 > $(A_0 = 1) \text{ AND } (A_1 = 1) \text{ AND } (\sim CS = 0) \text{ AND } (\sim W = 1)$

Suppose that we want to write four bits of data to ROW 1. In this case we don't want the individual ~OE inputs to the D-flops to be enabled because that would turn on the tri-state output buffers and cause a conflict with the data we're trying to write into the memory. However, we'll still need the master ~CS signal because that enables the chip to be written to. Thus, to write four bits of data to ROW 1, we need the following equation:

$$\text{WRITE TO ROW 1} > (A_0 = 1) \text{ AND } (A_1 = 0) \text{ AND } (\sim CS = 0) \text{ AND } (\sim W = 0)$$

Figure 6.12 is a simplified schematic diagram of a commercially available memory circuit from NEC®, a global electronics and semiconductor manufacturer headquartered in Japan. The device is a $\mu\text{PD}444008^1$ 4M-Bit CMOS Fast Static RAM (SRAM) organized as 512K by 8-bit wide words (bytes). The actual memory array is composed of an X-Y matrix 4,194,304 individual memory cells. This is just like the 16-bit memory that we discussed earlier, only quite a bit larger. The circuit has 19 address lines going into it, labeled $A_0 \dots A_{18}$. We need that many address lines because $2^{19} = 524,288$, so

19 address lines will give us the right number of combinations that we'll need to access every memory word in the array.

The signal named /WE is the same as the ~W signal of our earlier example. It's just labeled differently, but still requires a low- to-high transition to write the data. The /CS signal is the same as our ~CS in the earlier example. One difference is that the commercial part also provides an explicit /OE signal for controlling the tri-state output buffers during a read operation. In our example, the ~OE operation is implied by the state of the ~W input. In actual use, the ability to independently control ~OE makes for a more flexible part, so it is commonly added to memory chips such as this one. Thus, you can see that our 16-bit memory is operationally the same as the commercially available part.

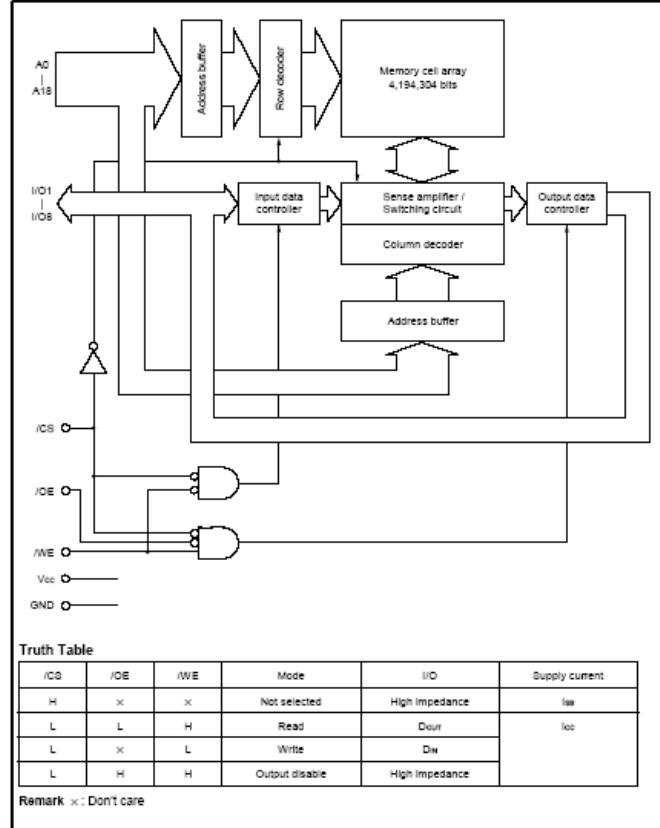


Figure 6.12 Logical diagram of an NEC μPD444008 4M-Bit CMOS Fast Static RAM. *Diagram courtesy of NEC Corporation.*

Let's return to figure 6.11 for a moment before we move on. Notice how each row of D-flops has two control signals going to each of the chips. One signal goes to the /OE tri-state controls and the other goes to the CLK input. What would the circuit inside of the block on the left actually look like? Right now, you have all of the knowledge and information that you need to design it. Figure 6.13 is the truth table.

You can see that the control logic for a real memory device, such as the μPD444008 in figure 6.12 could become significantly more complex as the number of bits increases from 16 to 4 million, but the principles are the same. Also, if you refer to figure 6.13 you should see that the decoding logic is highly regular and scalable. This would make the design of the hardware much more straight-forward.

A0	A1	R/~W	~CS	W0	~OE0	W1	~OE1	W2	~OE2	W3	~OE3
0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1
1	1	0	0	1	1	1	1	1	1	0	1
0	0	1	0	1	0	1	1	1	1	1	1
1	0	1	0	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	0	1	1
1	1	1	0	1	1	1	1	1	1	1	0
0	0	0	1	1	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1

Figure 6.13: Truth table for 16-bit memory decoder.

Data Bus Width and Addressable Memory

Before we move on to look at memory system designs of higher complexity, we need to stop and catch our breath for a moment, and consider some additional information that will help to make the upcoming sections more comprehensible. We need to put two pieces of information into their proper perspective:

1. Data bus width
2. Addressable memory

The width of a computer's data bus determines the size of the number that it can deal with in one operation or instruction. If we consider embedded systems as well as desktop PC's, servers, workstations, and mainframe computers, we can see a spectrum of data bus widths going from 4 bits up to 128 bits wide, with data buses of 256 bits in width just over the horizon. It's fair to ask, "Why is there such a variety?" The answer is speed versus cost. A computer with an 8-bit data path to memory can be programmed to do everything a processor with a 16-bit data path can do, except it will take longer to do it.

Consider this example. Suppose that we want to add two 16-bit numbers together to generate a 16-bit result. The numbers to be added are stored in memory and the result will be stored in memory as well. In the case of the 8-bit wide memory, we'll need to store each

16-bit word as two successive 8-bit bytes. Anyway, here's the algorithm for adding the numbers.

Case 1: 8-bit wide data bus

1. Fetch lower byte of first number from memory and place in an internal storage register.
2. Fetch lower byte of second number from memory and place in another internal storage register.
3. Add the lower bytes together.
4. Write the low order byte to memory.
5. Fetch upper byte of first number from memory and place in an internal storage register.
6. Fetch upper byte of second number from memory and place in another internal storage register.
7. Add the two upper bytes together with the carry (if present) from the prior add operation.
8. Write the upper byte to the next memory location from the low order byte.
9. Write the carry (if present) to the next memory location.

Case 2: 16-bit wide data bus

1. Fetch the first number from memory and place in an internal storage register.
2. Fetch the second number from memory and place in another internal storage register.
3. Add the two numbers together.
4. Write the result to memory.
5. Write the carry (if present) to memory.

As you can see, **Case 1** required almost twice the number of steps as **Case 2**. The efficiency gained by going to wider data buses is dependent upon the algorithm being executed. It can vary from as little as a few percent improvement to almost four times the speed, depending upon the algorithm being implemented.

Here's a summary of where the various bus widths are most common.

- 4,8 bits: appliances, modems, simple applications
- 16 bits: industrial controllers, automotive applications
- 32 bits: telecommunications, laser printers, desktop PC's
- 64 bits: high end PC's, UNIX workstations, games (Nintendo 64)
- 128 bits: high performance video cards for gaming
- 128, 256 bits: next generation, very long instruction word (VLIW) machines

Sometimes we try to economize by using a processor with a wide internal data bus with a narrower memory. For example, the Motorola 68000 processor that we'll study in this class has a 16-bit external data bus and a 32-bit internal data bus. It takes two memory fetches to

bring in a 32-bit quantity from memory, but once it is inside the processor it can be dealt with as a single 32-bit value.

Address Space

The next consideration in our computer design is how much addressable memory the computer is equipped to handle. The amount of externally accessible memory is defined as the *address space* of the computer. This address space can vary from 1024 bytes for a simple device to over 60 gigabytes for a high performance machine. Also, the amount of memory that a processor can address is independent of how much memory you actually have in your system. The Pentium processor in your PC can address over four billion bytes of memory, but most users rarely have more than 1 Gigabyte of memory inside their computer. Here are some simple examples of addressable memory:

- A simple microcontroller, such as the one inside of your Mr. Coffee® machine, might have 10 address lines, A0 . . . A9, and is able to address 1024 bytes of memory ($2^{10} = 1024$).
- A generic 8-bit microprocessor, such as the one inside your burglar alarm, has 16 address lines, A0 . . . A15, and is able to address 65,536 bytes of memory ($2^{16} = 65,536$).
- The original Intel 8086 microprocessor that started the PC revolution has 20 address lines, A0 . . . A19, and is able to address 1,048,576 bytes of memory ($2^{20} = 1,048,576$).
- The Motorola 68000 microprocessor has 24 address lines, A0 . . . A23, and is able to address 16,777,216 bytes of memory ($2^{24} = 16,777,216$).
- The Pentium microprocessor has 32 address lines, A0 . . . A31, and is able to address 4,294,967,296 bytes of memory ($2^{32} = 4,294,967,296$).

As you'll soon see, we generally refer to addressable memory in terms of bytes (8-bit values) even though the memory width is greater than that. This creates all sorts of memory addressing ambiguities that we'll soon get into.

Paging

Suppose that you're reading a book. In particular, this book is a very strange book. It has exactly 100 words on every page and each word on each page is numbered from 0 to 99. The book has exactly 100 pages, also numbered from 0 to 99. A quick calculation tells you that the book has 10,000 words (100 words/page x 100 pages). Also, next to every word on every page is the absolute number of that word in the book, with the first word on page 0 given the number 0000 and the last word on the last page given the number 9,999. This is a very strange book indeed!

However, we notice something quite interesting. Every word on a page can be uniquely identified in the book in one of two ways:

1. Give the absolute number of the word from 0000 to 9,999

- Give the page number that the word is on, from 00 to 99 and then give the position of the word on the page, from 00 to 99.

Thus, the 45th word on page 36 could be numbered as 3644 in absolute addressing or as page = 36, offset = 44. As you can see, however we choose to form the address, we get to the correct word. As you might expect, this type of addressing is called ***paging***. Paging requires that we supply two numbers in order to form the correct address of the memory location we're interested in:

- Page number*** of the page in memory that contains the data,
- Page offset*** of the memory location in that page.

Figure 6.14 shows such a scheme for a microprocessor (sometimes we'll use the Greek letter "mu" and the letter "P" together, μP , as a shorthand notation for microprocessor). The microprocessor has 20 address lines, A0 . . . A19, so it can address 1,048,576 bytes of memory. Unfortunately, we don't have a memory chip that is just the right size to match the memory address space of the processor. This is usually the case, so we'll need to add additional circuitry (and multiple memory devices) to provide enough memory so that every possible address coming out of the processor has a corresponding memory location to link to.

Since this memory system is built with 64K byte memory devices, each of the 16 memory chips has 16 address lines, A0 through A15. Therefore, each of the address lines of the address bus, A0 through A15, goes to each of the address pins of each memory chip.

The remaining four address lines coming out of the processor, A16 through A19 are used to select which of the 16 memory chips we will be addressing. Remember that the 4 most significant address lines, A16 through A19 can have 16 possible combinations of values from 0000 to 1111, or 0 through F in hexadecimal.

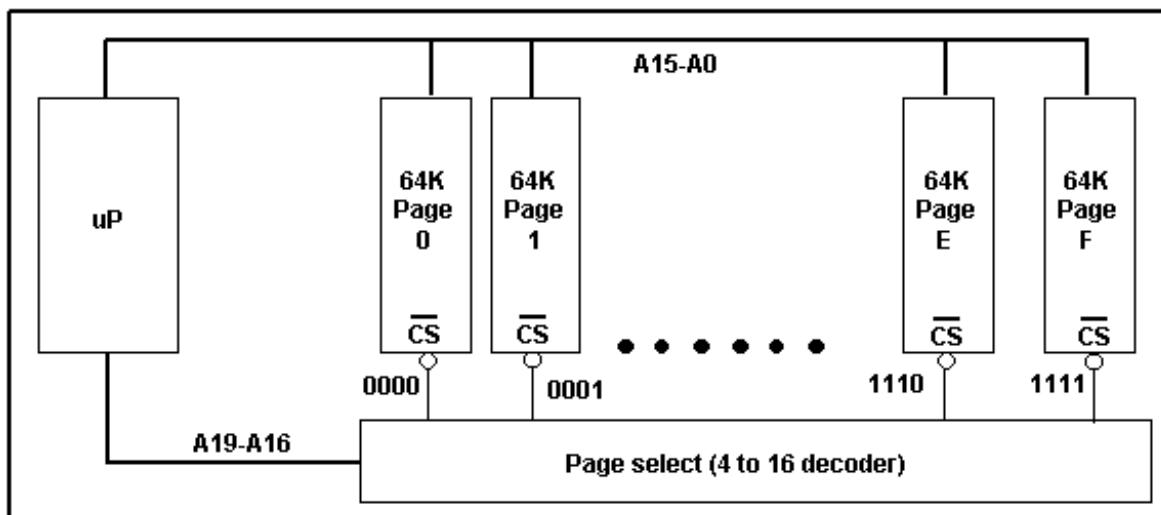


Figure 6.14: Memory organization for a 20-bit microprocessor. The memory space is organized as 16, 64K byte memory pages.

Consider the microprocessor in figure 6.14. Let's assume that it outputs the hexadecimal address 9A30D. The least significant address lines A0 through A15 from the processor go to each of the corresponding address inputs of the 16 memory devices. Thus, each memory device sees the hexadecimal address value A30D. Address bits A16 through A19 go to the page select circuit. So, we might wonder if this system will work at all. Won't the data stored in address A30D of each of the memory devices interfere with each other and give us garbage?

The answer is no, thanks to the /CS inputs on each of the memory chips. Assuming that the processor really wants the byte at memory location 9A30D, the remaining four address lines coming out of the processor, A16 through A19 are used to select which of the 16 memory chips we will be addressing. Remember that the 4 most significant address lines, A16 through A19 can have 16 possible combinations of values from 0000 to 1111, or 0 through F in hexadecimal.

This looks suspiciously like the decoder design problem we discussed earlier. This memory design has a 4:16 decoder circuit to do the page selection with the most significant 4 address bits selecting the page and the remaining 16 address bits form the page offset of the data in the memory chips. Notice that the same address lines, A0 through A15, go to each of the 16 memory chips, so if the processor puts out the hexadecimal address E3AB0, all 16 memory chips will see the address 3AB0. Why isn't there a problem? As I'm sure you can all chant in unison by now it is the tri-state buffers which enable us to connect the 16 pages to a common data bus. Address bits A16 through A19 determine which one of the 16 ~CS signals to turn on. The other 15 remain in the HIGH state, so their corresponding chips are disabled and do not have an effect on the data transfer.

Paging is a fundamental concept in computer systems. It will appear over and over again as we delve further into the operation of computer systems. In figure 6.14 we organized the 20-bit address space of the processor as 16, 64K byte pages. We probably did it that way because we were using 64K memory chips. This was somewhat arbitrary, as we could have organized the paging scheme in a totally different way depending upon the type of memory devices we had available to us. Figure 6.15 shows other possible ways to organize the memory. Also, we could build up each page of memory from multiple chips, so the pages themselves might need to have additional hardware decoding on them.

It should be emphasized that the type of memory organization used in the design of the computer will, in general, be transparent to the software developer. The hardware design specification will certainly provide a memory map to the software developer, providing the address range for each type of memory, such as RAM, ROM, FLASH, etc. However, the software developer need not worry about how the memory decoding is organized.

From the software designer's point of view, the processor puts out a memory address and it is up to the hardware design to correctly interpret it and assign it to the proper memory device or devices.

Page address	Page address bits	Page offset	Offset address bits	
NONE	NONE	0 to 1,048,575	A0 to A19	Linear address
0 to 1	A19	0 to 524,287	A0 to A18	
0 to 3	A19-A18	0 to 262,143	A0 to A17	
0 to 7	A19-A17	0 to 131,071	A0 to A16	
0 to 15	A19-A16	0 to 65,535	A0 to A15	
0 to 31	A19-A15	0 to 32,767	A0 to A14	
0 to 63	A19-A14	0 to 16,383	A0 to A13	

Our example

Figure 6.15: Possible paging schemes for a 20-bit address space.

Paging is important because it is needed to map the *linear address space* of the microprocessor into the physical capacity of the storage devices. Some microprocessors, such as the Intel 8086 and its successors, actually use paging as their primary addressing mode. The external address is formed from a page value in one register and an offset value in another. The next time your computer crashes and you see the infamous "Blue Screen of Death" look carefully at the funny hexadecimal address that might look like

BD48:0056

This is a 32-bit address in page-offset representation. Note that disk drives use paging as their only addressing mode. Each disk is divided into 512 byte sectors (pages). A four gigabyte disk has 8,388,608 pages.

Designing a Memory System

You may not agree, but we're ready to put it all together and design a real memory system for a real computer. OK, maybe we're not quite ready, but we're pretty close. Close enough to give it a try. Figure 6.16 is a schematic diagram for a computer system with a 16-bit wide data bus.

First, just a quick reminder that in binary arithmetic, we use the shorthand symbol "K" to represent 1024, and not 1000, as we do in most engineering applications. Thus, by saying 256K you really mean 262,144 and not 256,000. Usually, the context would eliminate the ambiguity; but not always, so beware.

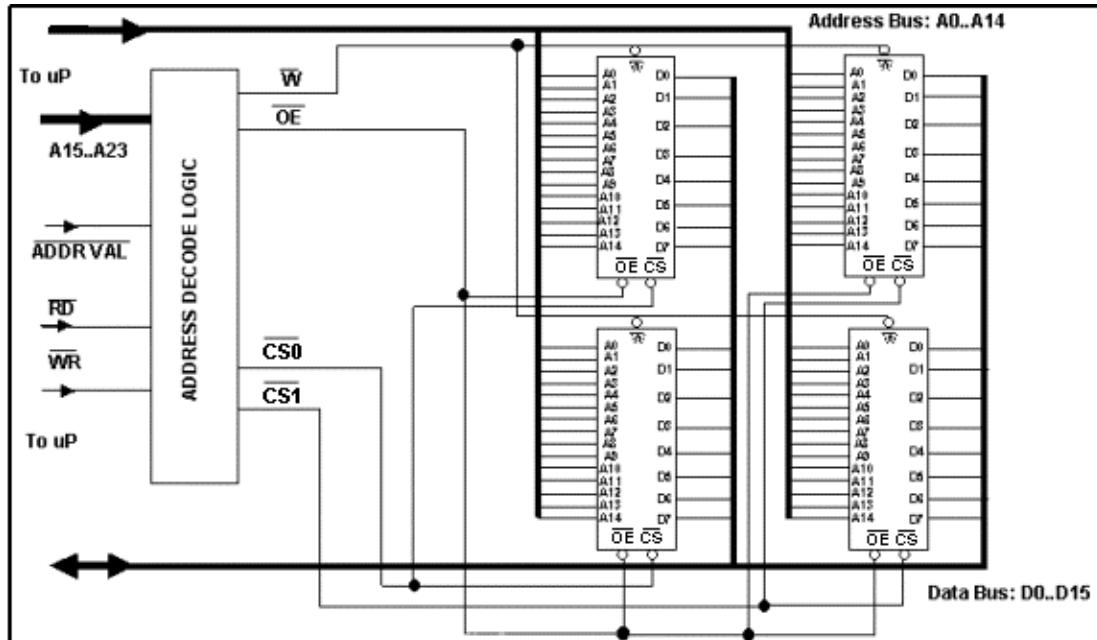


Figure 6.16: Schematic diagram for a 64K by 16 memory system built from four 32K by 8 memory chips.

The circuit in figure 6.16 looks a lot more complicated than anything we've considered so far, but it really isn't very different than what we've already studied. First, let's look at the memory chips. Each chip has 15 address lines going into it, implying that it has 32K unique memory addresses because $2^{15} = 32,768$. Also, each chip has eight data input/output (I/O) lines going into it. However, you should keep in mind that the data bus in Figure 6.16 is actually 16 bits wide (D0..D15) so we would actually need two, 8-bit wide, memory chips in order to provide the correct memory width to match the width of the data bus. We'll discuss this point in greater detail when we discuss figure 6.17.

There is another subtle point that we need to mention in passing before we look further into memory design and organizational issues. The circuit of Figure 6.16 has a limitation that we would not tolerate in a real memory design. Storing a single byte in memory requires the same amount of storage (16-bits) as storing a 16-bit value. Thus, every time we store a string variable in memory, we are throwing away 50% of the memory's storage capacity. What we are lacking is a mechanism to store quantities smaller than the width of the data bus in such a way that we make full use of the available storage capacity of the memory.

In Figure 6.16, this translates to a mechanism for independently accessing either the upper or lower memory chip of each page. With that addition, we could write a single byte to one or the other memory device and not corrupt the data in the other device.

Thus, with additional circuitry, we should be able to add a mechanism which would allow us to overcome the design limitation. For example, let's assume that the processor wants to store a byte in one of the memory chips in Figure 6.16. It could first read the data from both chips into an internal storage area, then write the byte into the correct half of the 16-bit word and finally, it could write the result back out to both chips. This is an extremely costly

operation to write a single byte. In practice, and as you'll see in Chapter 7, we need to add a bit more to this circuit design so that byte-addressability within a larger memory word can be realized.

The internal organization of the four memory chips in figure 6.16 is identical to the organization of the circuits we've already studied except these devices contain 256K memory cells and the memory we studied in figure 6.11 had 16 memory cells. It's a bit more complicated, but the idea is the same. Also, it would have taken me more time to draw 256K memory cells than to draw 16, so I took the easy way out.

This memory chip arrangement of 32K memory locations with each location being 8-bits wide is conceptually the same idea as our 16-bit example in figure 6.11 in terms of how we would add more devices to increase the size of our memory in both width (size of the data bus) and depth (number of available memory locations). In figure 6.11 we discussed a 16-bit memory organized as four memory locations with each location being 4-bits wide. In figure 6.16 there are a total of 262,144 memory cells in each chip because we have 32,768 rows by 8 columns in each chip.

Each chip has the three control inputs, $\sim\text{OE}$, $\sim\text{CS}$ and $\sim\text{W}$. In order to read from a memory device we must do the following steps:

1. Place the correct address of the memory location we want to read on A0 through A14.
2. Bring $\sim\text{CS}$ LOW to turn on the chip.
3. Keep $\sim\text{W}$ HIGH to disable writing to the chip.
4. Bring $\sim\text{OE}$ LOW to turn on the tri-state output buffers.

The memory chips then put the data from the corresponding memory location onto data lines D0 through D7 from one chip, and D8 through D15 from the other chip. In order to write to a memory device we must do the following steps:

1. Place the correct address of the memory location we want to write on A0 through A14.
2. Bring $\sim\text{CS}$ LOW to turn on the chip.
3. Bring $\sim\text{W}$ LOW to enable writing to the chip.
4. Keep $\sim\text{OE}$ HIGH to disable the tri-state output buffers.
5. Place the data on data lines D0 through D15. With D0 through D7 going to one chip and D8 through D15 going to the other.
6. Bring $\sim\text{W}$ from LOW to HIGH to write the data into the corresponding memory location.

Now that we understand how an individual memory chip works, let's move on to the circuit as a whole. In this example our microprocessor has 24 address lines, A0 through A23. A0 through A14 are routed directly to the memory chips because each chip has an address space of 32K bytes. The nine most significant address bits, A15 through A23 are needed to provide the paging information for the decoding logic block. These nine bits tell us that this memory space may be divided up into 512 pages with 32K address on each page. However,

the astute reader will immediately note that we only have a total of four memory chips in our system. Something is definitely wrong! We don't have enough memory chips to fill 512 pages. Oh drat, I hate it when that happens!

Actually, it isn't a problem after all. It means that out of a possible 512 pages of addressable memory, our computer has 2 pages of real memory, and space for another 510 pages. Is this a problem? That's hard to say. If we can fit all of our code into the two pages we do have, then why incur the added costs of memory that isn't being used? I can tell you from personal experience that a lot of sweat has gone into cramming all of the code into fewer memory chips to save a dollar here and there.

The other question that you might ask is this: "OK, so the addressable memory space of the μ P is not completely full. So where's the memory that we do have positioned in the address space of the processor?" That's a very good question because we don't have enough information right now to answer that. However, before we attempt to program this computer and memory system, we must design the hardware so that the memory chips we do have are correctly decoded at the page locations they are designed to be at. We'll see how that works in a little while.

Let's return to figure 6.16. It's important to understand that we really need two memory chips for each page of memory because our data bus is 16-bits wide, but each memory chip is only eight data bits wide. Thus, in order to build a 16-bit wide memory, we need two chips. We can see this in figure 6.17. Notice how each memory device connects to a separate group of eight wires in the data bus. Of course, the address bus pins, A0 through A14 must connect to the same wires of the address bus, because we are addressing the same address location both memory chips.

Now that you've seen how the two memory chips are "stacked" to create a page in memory that is 32K by 16, it should not be a problem for you to design a 32K by 32 memory using four chips.

You may have noticed that the microprocessor's clock was nowhere to be seen in this example of a memory system. Surely, one of the most important links in a computer system, the memory to processor, needs a clock signal in order to synchronize the processor to the memory. In fact, many memory systems do not need a clock signal to insure reliable performance. The only thing that needs to be considered is the timing relationship between the memory circuits and the processor's bus operation. In the next chapter we'll look at a processor bus cycle in more detail, but here's a preview. The NEC μ PD444008 comes in three versions. The actual part numbers are:

- μ PD444008-8
- μ PD444008-10
- μ PD444008-12

The numerical suffixes, 8, 10 and 12, refer to the maximum **access time** for

each of the chips. The access time is basically a specification which determines how quickly the chip is able to reliably return data once the control inputs have been properly established. Thus, assuming that the address to the chip has stabilized, $\sim\text{CS}$ and $\sim\text{OE}$ are asserted, then after a delay of 8, 10 or 12 nanoseconds (depending upon the version of the chip being used) the data would be available for reading into the processor. The chip manufacturer, NEC, guarantees that the access time will be met for the entire temperature range that the chip is designed to operate over. For most electronics, the commercial temperature range is 0 degrees Celsius to 70 degrees Celsius.

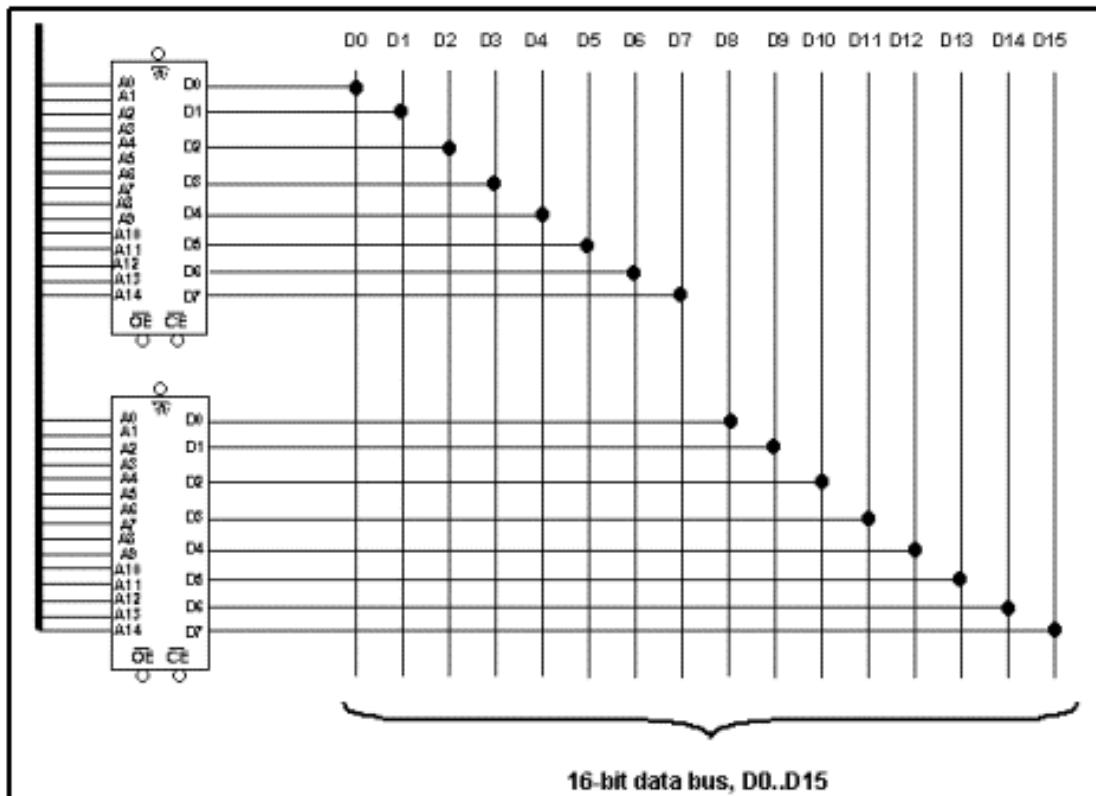


Figure 6.17: Expanding a memory system by width.

Let's do a simple example to see what this means. We'll actually look into this in more detail later on, but it can't hurt to prepare ourselves for things to come. Suppose that we have a processor with a 500 MHz clock. You know that this means that each clock period is 2 ns long. Our processor requires 5 clock cycles to do a memory read, with the data being read into the processor on the falling edge of the 5th clock cycle. The address and control information comes out of the processor on the rising edge of the first clock cycle. This means that the processor requires 4.5×2 , or 9 ns to do a memory read operation.

However, we're not quite done with our calculation. Our decoding logic circuit also introduces a time delay. Assume that it takes 1ns from the time the processor asserts the control and address signal to the time that the decoding logic provides the correct signals to the memory system. This means that we actually have 8 ns, not 9 ns, to read the data. Thus, only the fastest version of the part (generally this means the most expensive version) would work reliably in this design.

Is there anything that we can do? We could slow down the clock. Suppose that we changed the clock frequency from 500 MHz to 400 MHz. This lengthens the period to 2.5 ns per clock cycle. Now 4.5 clock cycles take 11.25 ns instead of 9 ns. Subtracting 1 ns for the propagation delay through the decoding logic, we would need a memory that was 10.25 ns or faster to work reliably. That looks pretty encouraging. We could slow the clock down even more so we could use even cheaper memory devices. Won't the Project Manager be pleased! Unfortunately, we've just made a trade-off. The trade-off is that we've just slowed our processor down by 20%. Everything the processor does will now take 20% longer. Can we live with that? At this point we probably don't know. We'll need to do some careful measurements of code execution times and performance requirements before we can answer the question completely; and even then we may have to make some pretty rough assumptions.

Anyway, the key to the above discussion is that there is no explicit clock in the design of the memory system. The clock dependency is implicit in the timing requirements of the memory-to-processor interface, but the clock itself is not required. In this particular design, our memory system is asynchronously connected to the processor.

Today, most PC memory designs are synchronous designs. The clock signal is an integral part of the control circuitry of the processor-to-memory interface. If you've ever added a

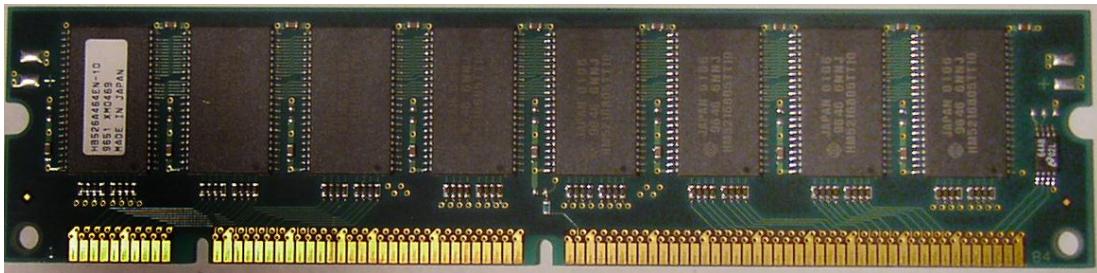


Figure 6.18: Typical SDRAM memory module.

memory "stick" to your PC then you've upped the capacity of your PC using ***synchronous dynamic random access memory or SDRAM*** chips. The printed circuit board (the stick) is a convenient way to mechanically connect the memory chips to the PC motherboard.

Figure 6.18 is a photograph of a 64 Megabyte (Mbyte) SDRAM memory module. This module holds 64 Mbytes of data organized as 1M by 64. There are a total of 16 memory chips on the module (front and back) each chip has a capacity of 32 Mbits, organized as 8M by 4. We'll look at the differences between asynchronous, or ***static*** memory systems, and synchronous, or ***dynamic***, memory systems later on in this chapter.

Paging in real memory systems

Our four memory chips of figure 6.16 will give us two 32K-by-16 memory pages. This leaves us 510 possible memory pages that are empty. How do we know where we'll have these two memory pages and where we will just have empty space? The answer is that it is up to you (or the hardware designer) to specify where the memory will be. As you'll soon see, in the 68000 system we want non-volatile memory (such as ROM or FLASH) to reside

from the start of memory and go up from there. Let's state for the purpose of this exercise that we want to locate our two available pages of real memory at page 0 and at page 511.

Let's assume that the processor has 24 address bits. This corresponds to about 16M of addressable memory (2^{24} address locations). It is customary to locate RAM memory (read/write) at the top of memory, but this isn't required. In most cases, it will depend upon the processor architecture. In any case, in this example we need to figure out how to make one of the two real memory pages respond to addresses from 0x000000 through 0x007FFF. This is the first 32K of memory and corresponds to page 0. The other 32K words of memory should reside in the memory region from 0xFF8000 through 0xFFFFFFF, or page 511. How do we know that? Simple, it's paging. Our total system memory of 16,777,216 words may be divided up into 512 pages with 32K on each page.

Since we have nine bits for the paging we can divide the absolute address up as shown in table 6.2.

Page Number (binary)	Page number (hex)	Absolute address range (hex)
A23.....A15		
000000000	000	000000 to 007FFF
000000001	001	008000 to 00FFFF
000000010	002	010000 to 017FFF
000000011	003	018000 to 01FFFF
.	.	.
111111111	1FF	FF8000 to FFFFFFF

Table 6.2 Page numbers and memory address ranges for a 24-bit addressing system.

We want the two highlighted memory ranges to respond by asserting the ~CS0 or ~CS1 signals when the memory addresses are within the correct range and the other memory ranges to remain unasserted. The decoder circuit for page 1FF is shown in figure 6.19. The circuit for page 000 is left as an exercise for you.

Notice that there is a new signal called ~ADDRVAL (Address Valid). The Address Valid signal (or some other similar signal) is issued by the processor in order to notify the external memory that the current address on the bus is stable. Why is this necessary?

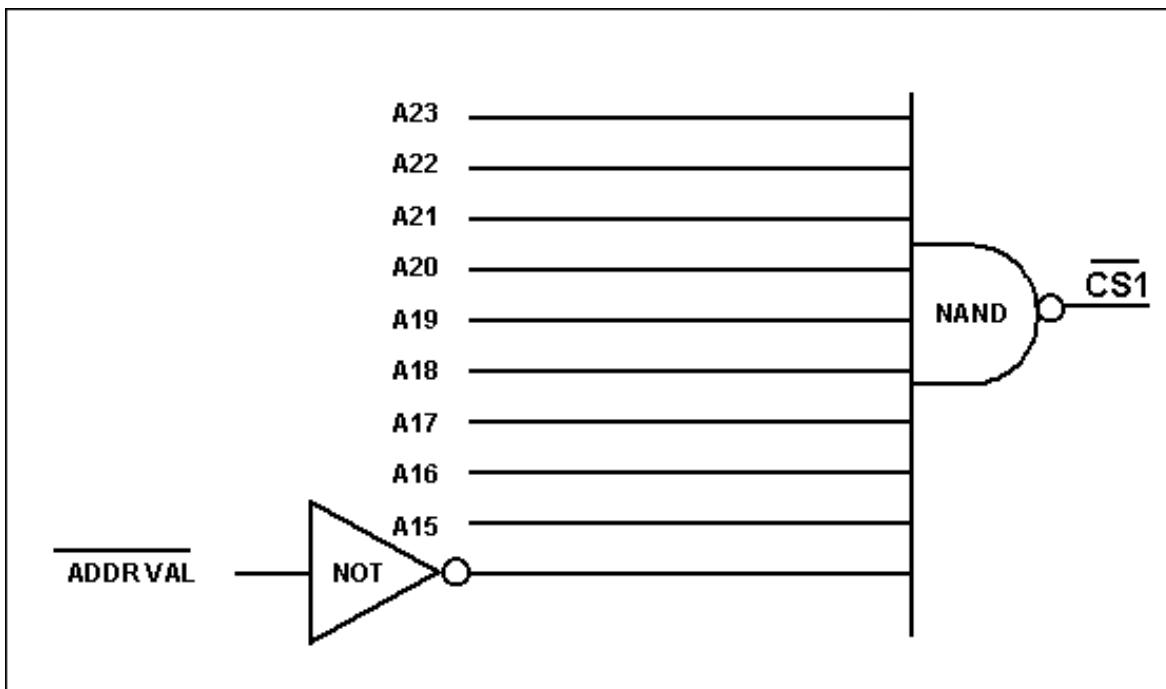


Figure 6.19: Schematic diagram for a circuit to decode the top page of memory of figure 6.16.

Keep in mind that the addresses on the address bus are always changing. Just executing one instruction may involve five or more memory accesses with different address values. The longer an address stays around, the worse the performance of the processor will be. Therefore, the processor must signal to the memory that the current value of the address is correct and the memory may respond to it. Also, some processors may have two separate signals $\sim\text{RD}$ and $\sim\text{WR}$, to signify read and write operations, respectively. Others just have a single line $\text{R}/\sim\text{W}$. There are advantages and disadvantages to each approach and we won't need to consider them here. For now, let's assume that our processor has two separate signals, one for a read operation and one for a write operation.

As you can see from figure 6.16 and from the discussion of how the memory chips work in our system, it is apparent that we can express the logical conditions necessary to read and write to memory as:

$$\text{MEMORY READ} = \sim\text{OE} * \sim\text{CS} * \text{WR}$$

$$\text{MEMORY WRITE} = \text{OE} * \sim\text{CS} * \sim\text{WR}$$

In both cases, we need to assert the $\sim\text{CS}$ signal in order to read or write to memory. It is the control of the chip enable (or chip select) signal that allows us to control where in the memory space of the processor a particular memory chip will become active.

With the exception of our brief introduction to SDRAM memories we've considered only ***static RAM (SRAM)*** for our memory devices. As you've seen, static RAM is derived from the D Flip-flop. It is a relatively simple interface to the processor because all we need to do is present an address and the appropriate control signals, wait the correct amount of time,

and then we can read or write to memory. If we don't access memory for long stretches of time there's no problem because the feedback mechanism of the flip-flop gate design keeps the data stored properly as long as power is applied to the circuit. However, we have to pay a price for this simplicity. A modern SRAM memory cell requires five or six transistors to implement the actual gate design. When you're talking about memory chips that store 256 million bits of data, a six transistor memory cell takes up a lot of valuable room on the silicon chip (die).

Today, most high-density memory in computers, like your PC, uses a different memory technology called ***dynamic RAM***, or ***DRAM***. DRAM cells are much smaller than SRAM cells, typically taking only one transistor per cell. One transistor is not sufficient to create the feedback circuit that is needed to store the data in the cell, so DRAM's use a different mechanism entirely. This mechanism is called ***stored charge***.

If you've ever walked across a carpet on a dry winter day and gotten a shock when you touched some metal, like the refrigerator, you're familiar with stored charge. Your body picked up excess charge as you walked across the carpet (now you represent a logical 1 state) and you returned to a logical 0 state when you got zapped as the charge left your body. DRAM cells work in exactly the same way. Each DRAM cell can store a small amount of charge that can be detected as a 1 by the DRAM circuitry. Store some charge and the cell stores a 1. Remove the charge and it stores a 0. (However, just like the charge stored on your body, if you don't do anything to replenish the charge, it will eventually leak away.) It's a bit more complicated than this, and the stored charge might actually represent a 0 rather than a 1, but it will be sufficient for our understanding of the concept.

In the case of a DRAM cell, the way that we replenish the charge is to periodically read the cell. Thus, DRAM's get their name from the fact that we are constantly reading them, even if we don't actually need the data stored in them. This is the ***dynamic*** portion of the

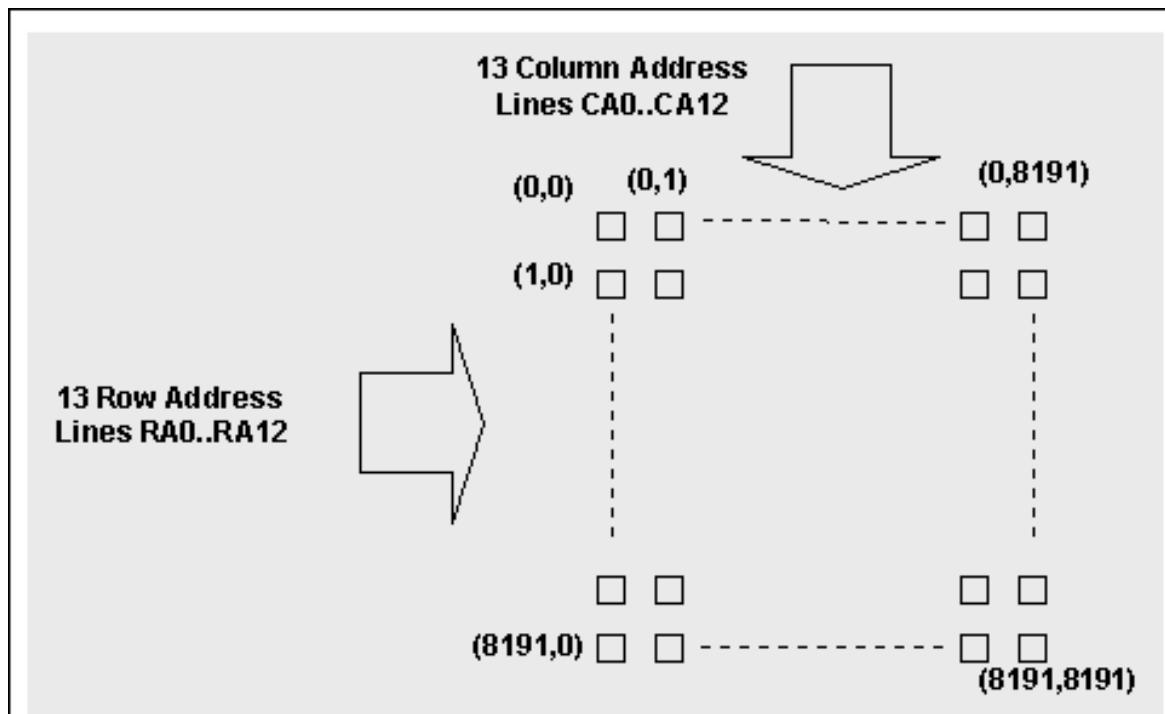


Figure 6.20: Organization of a 64 Megabit DRAM memory

DRAM's name. The process of reading from the cell is called a *refresh cycle*, and must be carried out at intervals. In fact, every cell of a DRAM must be refreshed every few milliseconds or the cell will be in danger of losing its data. Figure 6.20 shows a schematic representation of the organization of a 64 Mbit DRAM memory.

The memory is organized as a matrix with 8192 rows by 8192 columns (2^{13}). In order to uniquely address any one of the DRAM memory cells, a 26-bit address is required. Since we've already created it as a matrix, and 26 pins on the package would add a lot of extra complexity, the memory is addressed by providing a separate row address and a separate column address to the XY matrix. Fortunately for us, the process of creating these addresses is handled by the special chip sets on your PC's motherboard. Let's return to the refresh problem. Suppose that we must refresh each of the 64 million cells at least once every 10 milliseconds. Does that mean that we must do 64 million refresh cycles? Actually no; it is sufficient to just issue the row address to the memory and that guarantees that all of the 8192 cells in that row get refreshed at once. Now our problem is more tractable. If, for example the specification allows us 16.384 milliseconds to refresh 8192 rows in the memory, then we must, on average, refresh one row every $16.384 \times 10^{-3} / 8.192 \times 10^3$ seconds, or one row every two microseconds.

If this all seems very complicated, it certainly is. Designing a DRAM memory system is not for the beginning hardware designer. The DRAM introduces several new levels of complexity:

- We must break the full address down into a row address and a column address,
- We must stop accessing memory every microsecond or so and do a refresh cycle,
- If the processor needs to use the memory when a refresh also needs to access the memory, we then need some way to synchronize the two competing processes.

This makes the interfacing DRAM to modern processors quite a complex operation. Fortunately the modern support chip sets have this complexity well in hand. Also, if the fact that we must do a refresh every two microseconds seems excessive to you, remember that your 3 GHz Athlon or Pentium processor issues 6,000 clock cycles every two microseconds. So we can do a lot of processing before we need to do a refresh cycle.

The problem of conflicts arising because of competing memory access operations (read, write and refresh) are mitigated to a very large degree because modern PC processors contain on-chip memories called *caches*. Cache memories will be discussed in much more detail in a later chapter, but for now we can see the effect of the cache on our off-chip DRAM memories by greatly reducing the processor's demands on the external memory system.

As we'll see, the probability that the instruction or data that a processor requires will be in the cache is usually greater than 90%, although the exact probability is influenced by the algorithms being run at the time. Thus, only 10% of the time will the processor need to go to external memory in order to access data or instructions not in the cache. In modern processors, data is transmitted between the external memory systems and the processor in

bursts, rather than one byte or word at a time. Burst accesses can be very efficient ways to transfer data. In fact, you are probably already very familiar with the concept because so many other systems in your PC rely on burst data transfers. For example, your hard drive transfers data to memory in bursts of a sector of data at a time. If your computer is connected to a 10BaseT or 100BaseT network then it is processing packets of 256 bytes at a time. It would be just too inefficient and wasteful of the system resources to transmit data a byte at a time.

SDRAM memory is also designed to efficiently interface to a processor with on-chip caches and is specifically designed for burst accesses between the memory and the on-chip caches of the processor. Figure 6.21 is an excerpt from the data sheet for an SDRAM memory device from Micron Technology, Inc.®, a semiconductor memory manufacturer located in Boise, ID. The timing diagram is for the MT48LC128MXA2² family of SDRAM memories. The devices are 512 Mbit parts organized as 4, 8 or 16 bit wide data paths. The ‘X’ is a placeholder for the organization (4, 8 or 16 bit wide). Thus, the MT48LC128M4A2 is organized as 32M by 4 while the MT48LC128M16A2 is organized as 8M by 16.

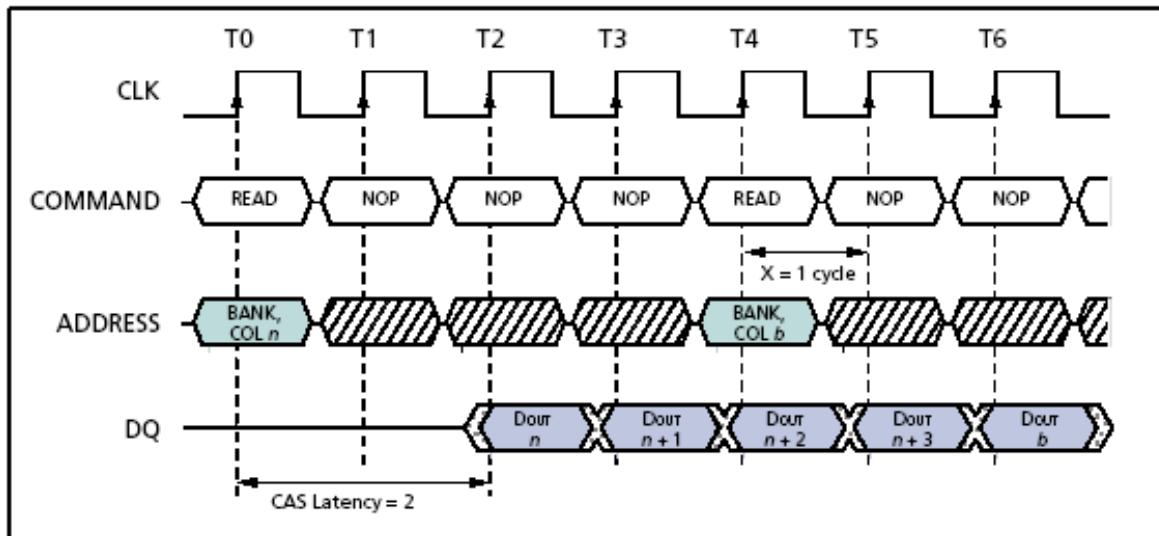


Figure 6.21: Timing diagram of a burst memory access for a Micron Technology Inc. part number MT48LC128MXA2 SDRAM memory chip. *Diagram courtesy of Micron Technology.*

These devices are far more complicated in their operation than the simple SRAM memories we've looked at so far. However, we can see the fundamental burst behavior in figure 6.21.

The fields marked **COMMAND**, **ADDRESS** and **DQ** are represented as bands of data, rather than individual bits. This is a simplification that allows us to show a group of signals, such as 14 address bits, without having to show the state of each individual signal. The band is used to show where the signal must be stable and where it is allowed to change. Notice how the signals are all synchronized to the rising edge of the clock. Once the **READ** command is issued and the address is provided for where the burst is to originate, there is a two clock cycle latency and sequentially stored data in the chip will then be available **on**

every successive clock cycle. Clearly, this is far more efficient than reading one byte at a time.

When we consider cache memories in greater detail we'll see that the on-chip caches are also designed to be filled from external memory in bursts of data. Thus, we incur a penalty in having to set-up the initial conditions for the data transfer from external memory to the on-chip caches, but once the data transfer parameters are loaded, the memory to memory data transfer can take place quite rapidly. For this family of devices, the data transfer takes place at a maximum clock rate of 133 MHz.

Newer SDRAM devices, called ***double data rate, or DDR*** chips, can transfer data on both the rising and falling edges of the clock. Thus, a DDR chip with a 133 MHz clock input can transfer data at a speedy 266 MHz. These parts are designated, for reasons unknown, as PC2700 devices. Any SDRAM chip capable of conforming to a 266 MHz clock rate are PC2700.

Modern DRAM design takes many different forms. We've been discussing SDRAM because this is the most common form of DRAM in a modern PC. Your graphics card contains video DRAM. Older PC's contained ***extended data out, or EDO DRAM***. Today, the most common type of SDRAM is DDR SDRAM. The amazing thing about all of this is the incredibly low cost of this type of memory. At this writing (summer of 2004) you can purchase 512 Mbytes of SDRAM for about 10 cents per megabyte. A memory with the same capacity, built in static RAM would cost well over \$2000.

Memory to processor interface

The last topic that we'll tackle in this chapter involves the details of how the memory system and the processor communicate with each other. Admittedly, we can only scratch the surface because there are so many variations on a theme when there are over 300 commercially available microprocessor families in the world today, but let's try to take a general overview without getting too deeply enmeshed in individual differences.

In general, most microprocessor-based systems contain three major bus groupings:

- Address bus: A unidirectional bus from the processor out to memory.
- Data bus: A bi-directional bus carrying data from the memory to the processor during read operations and from the processor to memory during write operations.
- Status bus: A heterogeneous bus comprised of the various control and housekeeping signals needed to coordinate the operation of the processor, its memory and other peripheral devices. Typical status bus signals include:
 - a- RESET,
 - b- interrupt management,
 - c- bus management,
 - d- clock signals,
 - e- read and write signals.

This is shown schematically in figure 6.22 for the Motorola®* MC68000 processor. The 68000 has a 24-bit address bus and a 16-bit external data bus. However, internally, both address and data can be up to 32 bits in length. We'll discuss the interrupt system and bus management system later on in this section.

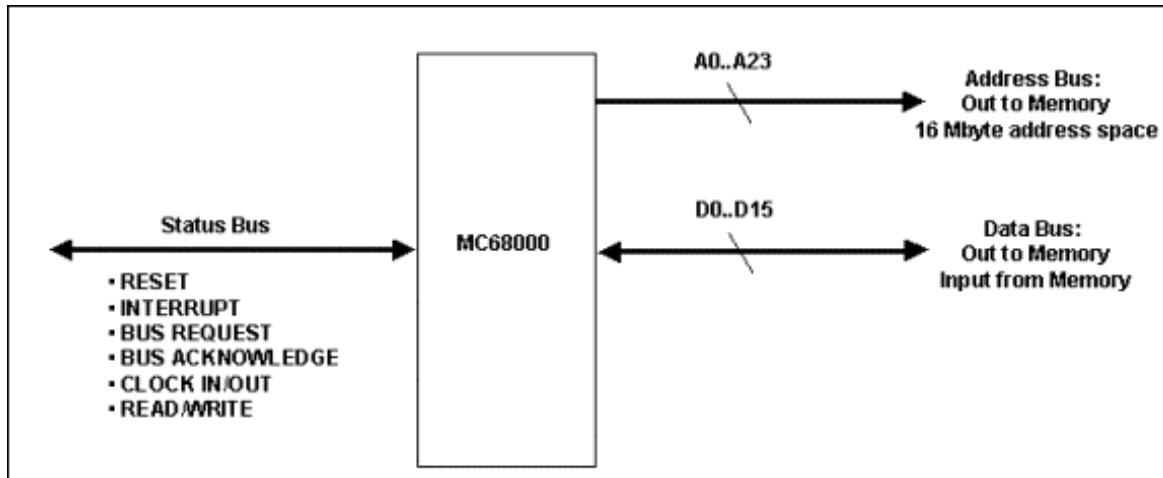


Figure 6.22: Figure 4.9: Three major busses of the Motorola 68000 processor

The Address Bus is the aggregate of all the individual address lines. We say that it is a ***homogeneous bus*** because all of the individual signals that make up the bus are address lines. The address bus is also unidirectional. The address is generated by the processor and goes out to memory. The memory does not generate any addresses and send them to the processor over this bus.

The Data Bus is also homogeneous, but it is bidirectional. Data goes out from memory to the processor on a read operation and from the processor to memory on a write operation. Thus, data can flow in either direction, depending upon the instruction being executed.

The Status Bus is ***heterogeneous***. It is made up of different kinds of signals, so we can't group them in the same way that we do for address and data. Also, some of the signals are unidirectional, some are bidirectional. The Status Bus is the "housekeeping" bus. All of the signals that are also needed to control system operation are grouped into the Status Bus.

* The Motorola Corporation has recently spun off its Semiconductor Products Sector (SPS) to form a new company, **Freescale®, Inc.** However, old habits die hard, so we'll continue to refer to processors derived from the 68000 architecture as the Motorola MC68000.

Let's now look at how the signals on these busses work together with memory so that we may read and write. Figure 6.23 shows us the processor side of the memory interface.

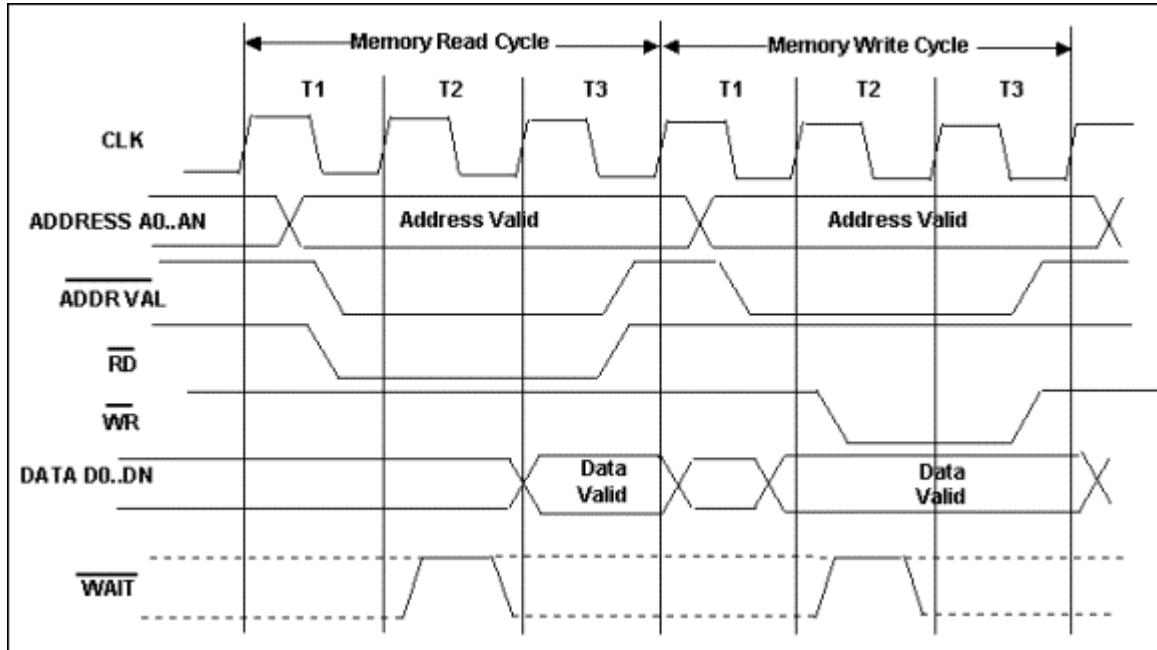


Figure 6.23: Figure 4.9: Timing diagram for a typical microprocessor

Now we can see how the processor and the clock work together to sequence the accessing of the memory data. While it may seem quite bewildering at first, it is actually very straightforward. Figure 6.23 is a “simplified” timing diagram for a processor. We’ve omitted many additional signals that may be present or absent in various processor designs and tried to restrict our discussion to the bare essentials.

The Y-axis shows the various signals coming from the processor. In order to simplify things, we’ve grouped all the signals for the address bus and the data bus into a “band” of signals. That way, at any given time, we can assume that some are 1 and some are 0, but the key is that we must specify when they are valid. The crossings, or X’s in the address and data busses are a symbolic way to represent points in time when the addresses or data on the busses may be changing, such as an address changing to a new value, or data coming from the processor.

Since the microprocessor is a state machine, everything is synchronized with the edges of the clock. Some events occur on the positive going edges and some may be synchronized with the negative going edges. Also, for convenience, we’ll divide the bus cycles into identifiable time signatures called “T states.” Not all processors work this way, but this is a reasonable approximation of how many processors actually work. Keep in mind that the processor is always running these bus cycles. These operations form the fundamental method of data exchange between the processor and memory. Therefore, we can answer a

question that was posed at the beginning of this chapter. Recall that the state machine truth table for the operation, ADD B,A, left out any explanation of how the data got into the registers in the first place, and how the instruction itself got into the computer.

Thus, before we look at the timing diagram for the processor/memory interface, we need to remind ourselves that the control of this interface is handled by another part of our state machine. In algorithmic terms, we do a "function call" to the portion of the state machine that handles the memory interface, and the data is read or written by that algorithm.

Let's start with a READ cycle. During the falling edge of the clock in T1 the address becomes stable and the /ADDR VAL signal is asserted LOW. Also, the /RD signal goes LOW to indicate that this is a read operation. During the falling edge of T3 the READ and ADDRESS VALID signals are de-asserted, indicating to memory that that the cycle is ending and the data from memory is being read by the processor. Thus, the memory must be able to provide the data to the processor within two full clock cycles (all of T2 plus half of T1 and half of T3).

Suppose the memory isn't fast enough to guarantee that the data will be ready in time. We discussed this situation for the case of the NEC static RAM chip and decided that a possible solution would be to slow the processor clock until the access time requirements for the memory could be guaranteed to be within specs. Now we will consider another alternative. In this scenario, the memory system may assert the ~WAIT signal back to the processor. The processor checks the state of the ~WAIT signal on the falling edge of the clock during T2 cycle. If the ~WAIT signal is asserted, the processor generates another T2 cycle and checks again. As long as the ~WAIT signal is LOW, the processor keeps marking time in T2. Only when ~WAIT goes high will the processor complete the bus cycle. This is called a **wait state**, and is used to synchronize slower memory to faster processors.

The write cycle is similar to the read cycle. During the falling edge of the clock in T1 the address becomes valid. During the rising edge of the clock in T2 the data to be written is put on the data bus and the write signal goes low, indicating a memory write operation. ~WAIT signal has the same function in T2 on the write cycle. During the falling edge of the clock in T3 the ~WR signal is de-asserted, giving the memory a rising edge to store the data. ~ADDR VAL also is de-asserted and the write cycle ends.

There are several interesting concepts buried in the previous discussion that require some explanation before we move on. The first is the idea of a state machine that operates on both edges of the clock, so let's consider that first. When we input a single clock signal to the processor in order to synchronize its internal operations, we don't really see what happens to the internal clock. Many processors will internally convert the clock to a **2-phase clock**. A timing diagram for a two phase clock is shown in figure 6.24.

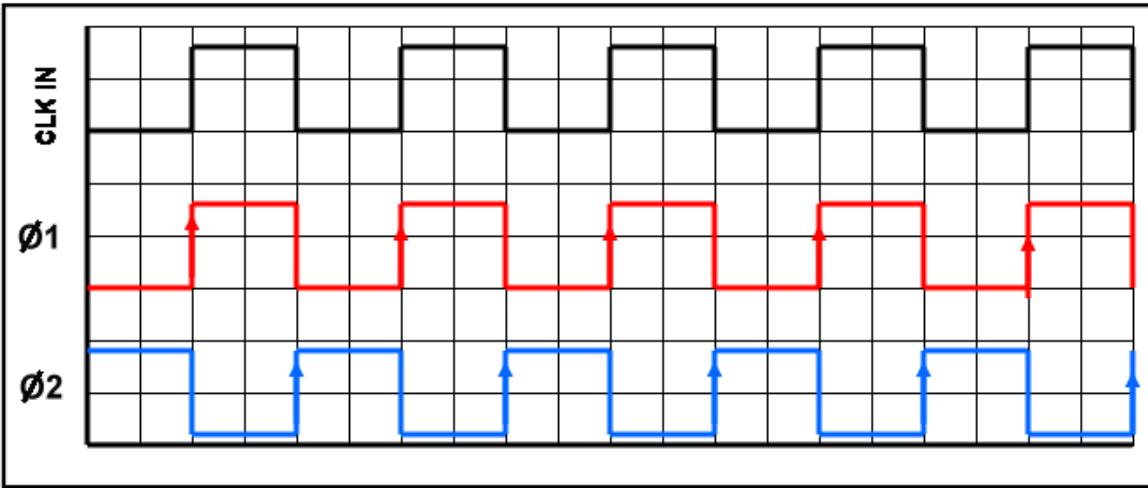


Figure 6.24: Timing diagram for a two-phase clock.

The input clock, which is generated by an external oscillator, is converted to a 2-phase clock, labeled ϕ_1 and ϕ_2 . The two clock phases now 180 degrees out of phase from each other, so that every rising or falling edge of the CLK IN signal generates an internal rising clock edge. How could we generate a 2-phase clock? You actually already know how to do it, but there's a piece of information that we first need to place in context. Figure 6.25³ is a circuit which can be used to generate a 2-phase clock.

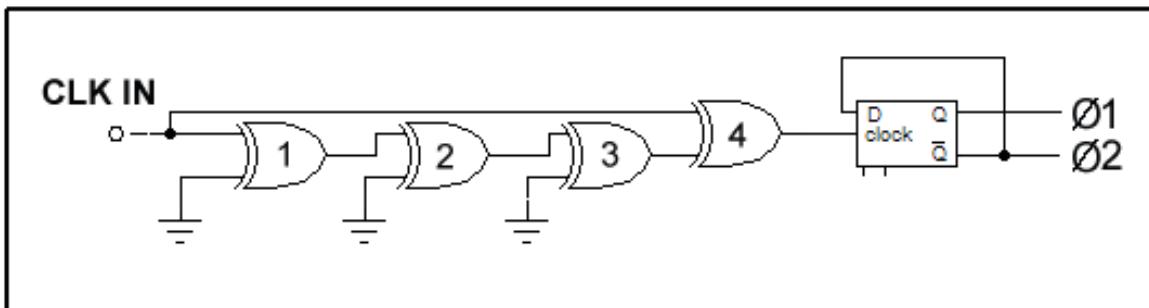


Figure 6.25: A Two-phase clock generation circuit.

The 4 XOR gates are convenient to use because there is a common integrated circuit part which contains 4 XOR gates in 1 package. This circuit makes use of the propagation delays that are inherent in a logic gate. Suppose that each XOR gate has a propagation delay of 10 ns. Assume that the clock input is LOW. One input of XOR gates 1 through 3 is permanently tied to ground (logic LOW). Since both inputs of gate 1 are LOW, its output is also LOW. This situation carries through to gates 2, 3 and 4. Now, the CLK IN input goes to logic state HIGH. The output of gate #4 goes high 10 ns later and toggles the D-FF to change state. Since the Q and \bar{Q} outputs are opposite each other, we conveniently have a source of two alternating clock phases by nature of the divide-by-two wiring of the D-FF.

After a propagation delay of 30 ns the output of gate #3 also goes HIGH, which causes the output of XOR gate #4 to go LOW again because the output of an XOR gate is LOW if both inputs are the same and HIGH if the inputs are different. At some time later, the clock input goes low again and we generate another 30 ns wide positive going pulse at the output of

gate #4 because for 30 ns both outputs are different. This cause the D-FF to toggle at both edges of the clock and the Q and $\sim Q$ outputs give us the alternating phases that we need. Figure 6.26 shows the relevant waveforms.

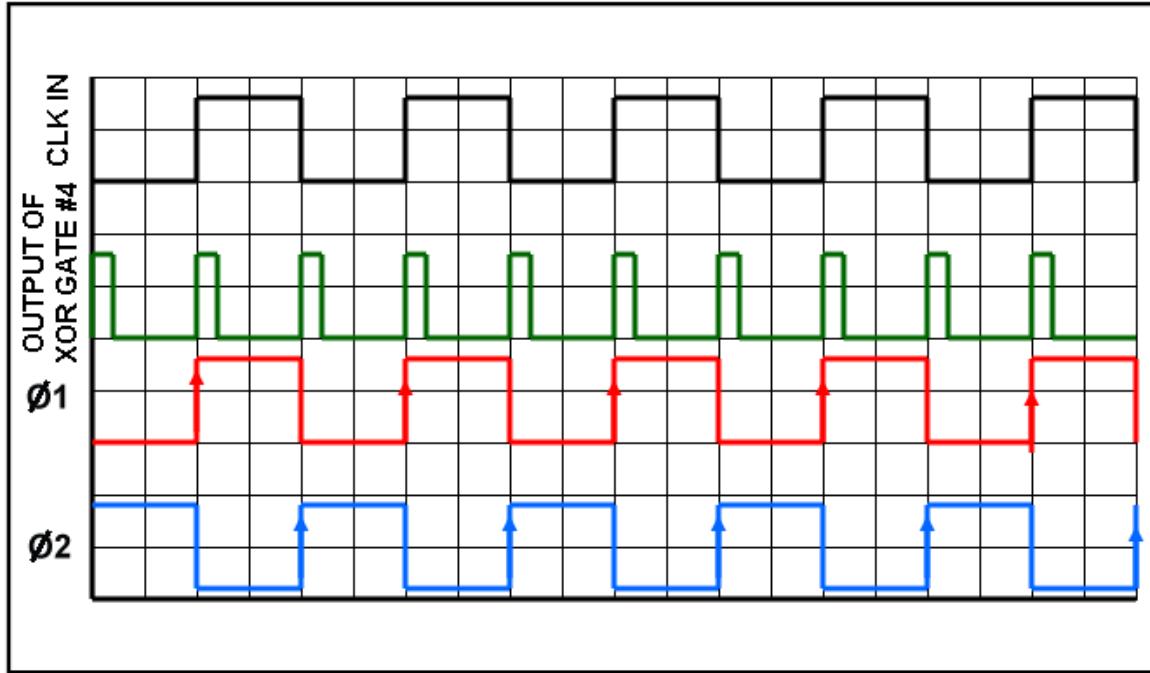


Figure 6.26: Waveforms for the 2-phase clock generation circuit.

This circuit works for any clock frequency that has a period greater than 4 XOR gate delays. Also, by using both outputs of the D-FF, we are guaranteed a two-phase clock output that is exactly 180 degrees out of phase from each other.

Now we can revisit figure 6.23 and see the other subtle point that was buried in the diagram. Since we are apparently changing states on the rising and falling edges of the clock, we now know that the internal state machine of the processor is actually using a 2- phase clock and each of the ‘T’ states is, in reality, two states. Thus, we can redraw the timing diagram for a READ cycle as a state diagram. This will clearly demonstrate the way in which the WAIT state comes into play. Figure 6.27 shows the READ phase of the bus cycle, represented as a state diagram.

Referring to figure 6.27 we can clearly see that in state T21 the processor tests the state of the \sim WAIT input. If the input is asserted LOW, the processor remains in state T21, effectively lengthening the total time for the bus cycle. The advantage of the wait state over decreasing the clock frequency is that we can design our system such that a wait penalty is incurred only when the processor accesses certain memory regions, rather than slowing it for all operations. We can now summarize the entire bus READ cycle as follows:

- T10: READ cycle begins.
Processor outputs new memory address for READ operation.
- T11: Address is now stable and $\sim AD$ VAL goes LOW. $\sim RD$ goes low indicating that a READ cycle is beginning.
- T20: READ cycle continues.
- T21: Processor samples $\sim WAIT$ input. If asserted T21 cycle continues.
- T30: READ cycle continues.
- T31: READ cycles terminates.
 $\sim AD$ VAL and $\sim RD$ are de-asserted and processor inputs the data from memory.

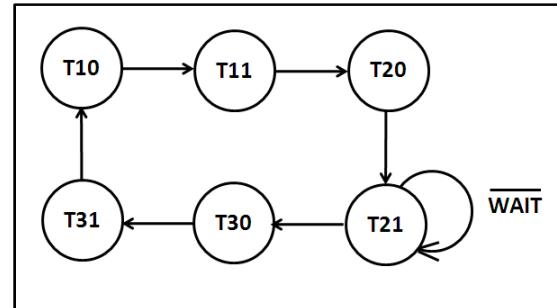


Figure 6.27: State diagram for a processor READ cycle

Direct Memory Access (DMA)

We'll conclude Chapter 6 with a brief discussion of another form of memory access called DMA, or direct memory access. The need for a DMA system is a result of the fact that the memory system and the processor are connected to each other by busses. Since the bus is the only path in and out of the system, conflicts will arise when peripheral devices, such as disk drives or network cards have data for the processor, but the processor is busy executing program code.

In many systems, the peripheral devices and memory share the same busses with the processor. When a device, such as a hard disk drive needs to transfer data to the processor, we could imagine two scenarios.

Scenario #1

- 1 Disk drive: “Sorry for the interrupt boss, I’ve got 512 bytes for you.”
- 2 Processor: “That’s a big 10-4 little disk buddy. Gimme the first byte.”
- 3 Disk drive: “Sure boss. Here it is.”
- 4 Processor: “Got it. Gimme the next one.”
- 5 Disk drive: “Here it is.”

Repeat steps 4 and 5 for 510 times.

Scenario #2

- 1 Disk drive: “Yo, boss. I got 512 bytes and they’re burning a hole in my platter. I gotta go, I gotta go.” (BUS REQUEST)

- 2 Processor: "OK, ok, pipe down lemme finish this instruction and I'll get off the bus. OK, I'm done, the bus is yours, and don't dawdle, I'm busy." (BUS GRANT)
- 3 Disk drive: "Thanks boss. You're a pal. I owe you one. I've got it." (BUS ACKNOWLEDGE)
- 4 Disk drive: "I'll put the data in the usual spot." (Said to itself)
- 5 Disk drive: "Hey boss! Wake up. I'm off the bus."
- 6 Processor: Thanks disk. I'll retrieve the data from the usual spot."
- 7 Disk drive: "10-4. The usual spot. I'm off."

As you might gather from these two scenarios, the second was more efficient because the peripheral device, the hard disk, was able to take over memory control from the processor and write all of its data in a single burst of activity. The processor had placed its memory interface in a tri-state condition and was waiting for the signal from the disk drive that it could return to the bus. Thus, the DMA allows other devices to take over control of the busses and implement a data transfer to or from memory while the processor idles, or processes from a separately cached memory. Also, given that many modern processors have large on-chip caches, the processor loses almost nothing by turning the external bus over to the peripheral device. Let's take the humorous discussion of the two scenarios and get serious for a moment. Figure 6.28 shows the simplified DMA process. You may also gather that I shouldn't quit my day job to become a sit-com writer, but that's a discussion for another time.

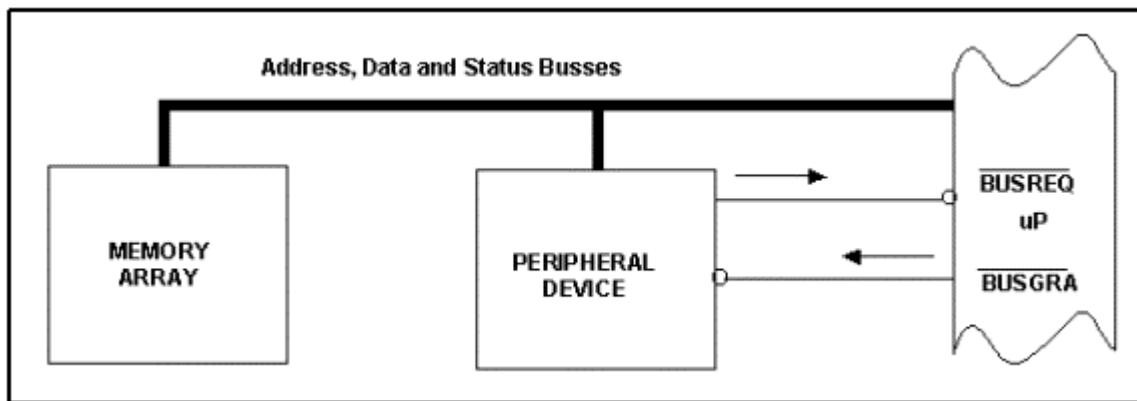


Figure 6.28: Schematic representation of a DMA transfer

In the simplest form, there is a **handshake** process that takes place between the processor and the peripheral device. A handshake is simply an action that expects a response to indicate the action was accepted. The process can be described as follows:

- The peripheral device requests control of the bus from the processor by asserting the **BUS REQUEST** (/BUSREQ) signal input on the processor.

- When processor completes present instruction cycle, and no higher level interrupts are pending, it sends out a **BUS GRANT** (/BUSGRA), giving the requesting device permission to begin its own memory cycles.
- Processor then idles, or continues to process data internally in cache, until BUSREQ signal goes away

Summary of chapter 6

- We looked at the need for bus organization within a computer system and how busses are organized into address, data and status busses.
- Carrying on our discussion of the previous chapter, we saw how the microcode state machine would work with the bus organization to control the flow of data on internal busses.
- We saw how the tri-state buffer circuit enables individual memory cells to be organized into larger memory arrays,
- We introduced the concept of paging as a way to form memory addresses and as a method to build memory systems.
- We looked at the different types of modern memory technology to understand the use of static RAM technology and dynamic RAM technology
- Finally, we concluded the overview of memory with a discussion of direct memory access as an efficient way to move blocks of data between memory and peripheral devices.

Bibliography and references

- 1- <http://www.necel.com/memory/pdfs/M14428EJ5V0DS00.pdf>
- 2- <http://download.micron.com/pdf/datasheets/dram/sdram/512MbSDRAM.pdf>
- 3- Ralph Tenny, *Simple Gating Circuit Marks Both Pulse Edges*, Designer's Casebook, Prepared by the editors of **Electronics**, McGraw Hill, Pg. 27

Exercises for chapter 6

1- Design a 2 input, 4-output memory decoder, given the truth table shown below:

INPUTS		OUTPUTS			
A	B	/O1	/O2	/O3	/O4
0	0	0	1	1	1
1	0	1	0	1	1
0	1	1	1	0	1
1	1	1	1	1	0

2- Refer to figure 6.11. The external input and output (I/O) signals defined as follows:

A0,A1: Address inputs for selecting which row of memory cells (D-flip flops) to read from, or write to.

/CS: Chip Select signal. When low, the memory is active and you read from it or write to it.

R/~W: Read/Write line. When high, the appropriate row within the array may be read by an external device. When low, an external device may write data into the appropriate row. The appropriate row is defined by the state of address bits A0 and A1.

DB0, DB1, DB2, DB3: Bi-directional data bits. Data being written into the appropriate row, or read from the appropriate row, as defined by A0 and A1, are defined by these 4 bits.

The array works as follows:

1- To read from a specific address (row) in the array:

- a- Place the address on A0 and A1
- b- Bring /CS low
- c- Bring R/~W high.
- d- The data will be available to read on D0..D3

2- To write to a specific address in the array:

- a- Place the address on A0 and A1
- b- Bring /CS low
- c- Bring R/~W low.
- d- Place the data on D0..D3.
- e- Bring R/~W high.

3- Each individual memory cell is a standard D flip-flop with one exception. There is a tri-state output buffer on each individual cell. The output buffer is controlled by the

/CS signal on each FF. When this signal is low, the output is connected to the data line and the data stored in the FF is available for reading. When this output is high the output of the FF is isolated from the data line so that data may be written into the device.

Consider the box labeled "Memory Decoding Logic" in the diagram of the memory array. Design the truth table for that circuit, simplify it using K-Maps and draw the gate logic to implement the design.

3- Assume that you have a processor with a 26-bit wide address bus and a 32-bit wide data bus.

a- Suppose that you are using memory chips organized as 512K deep by 8 bits wide (4 Mbit). How many memory chips are required to build a memory system for this processor that completely fills the entire address space, leaving no empty regions?

b- Assuming that we use a page size of 512K, complete the following table for the first three pages of memory:

Page Number	Starting Address (Hex)	Ending Address (Hex)

4- Consider the memory timing diagram from figure 6.23. Assume that the clock frequency is 50 MHz and that you do not want to add any wait states to slow the processor down. What is the slowest memory access time that will work for this processor?

5- Define the following terms in a few sentences:

- a- Direct Memory Access
- b- Tri-state logic
- c- Address bus, data bus, status bus

6- The figure shown below is a schematic diagram of a memory device that will be used in a memory system for a computer with the following specifications:

- 20-bit address bus
- 32-bit data bus
- Memory at pages 0,1 and 7

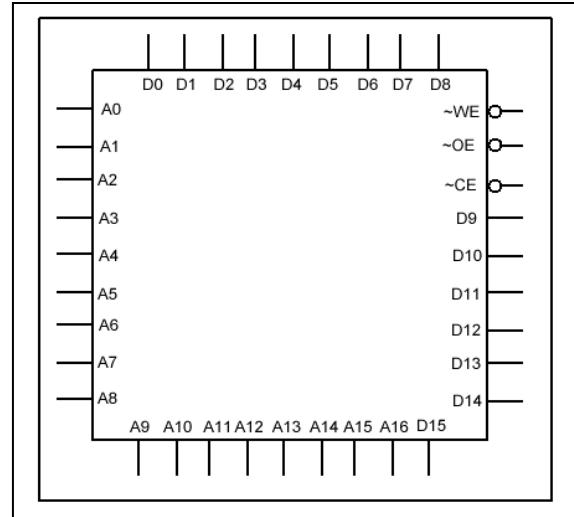
a- How many addressable memory locations are in each memory device?

b- How many bits of memory are in each memory device?

c- What is the address range, in hex, covered by each memory device in the computer's address space? You may assume that each page of memory is the same size as the address range of one memory device.

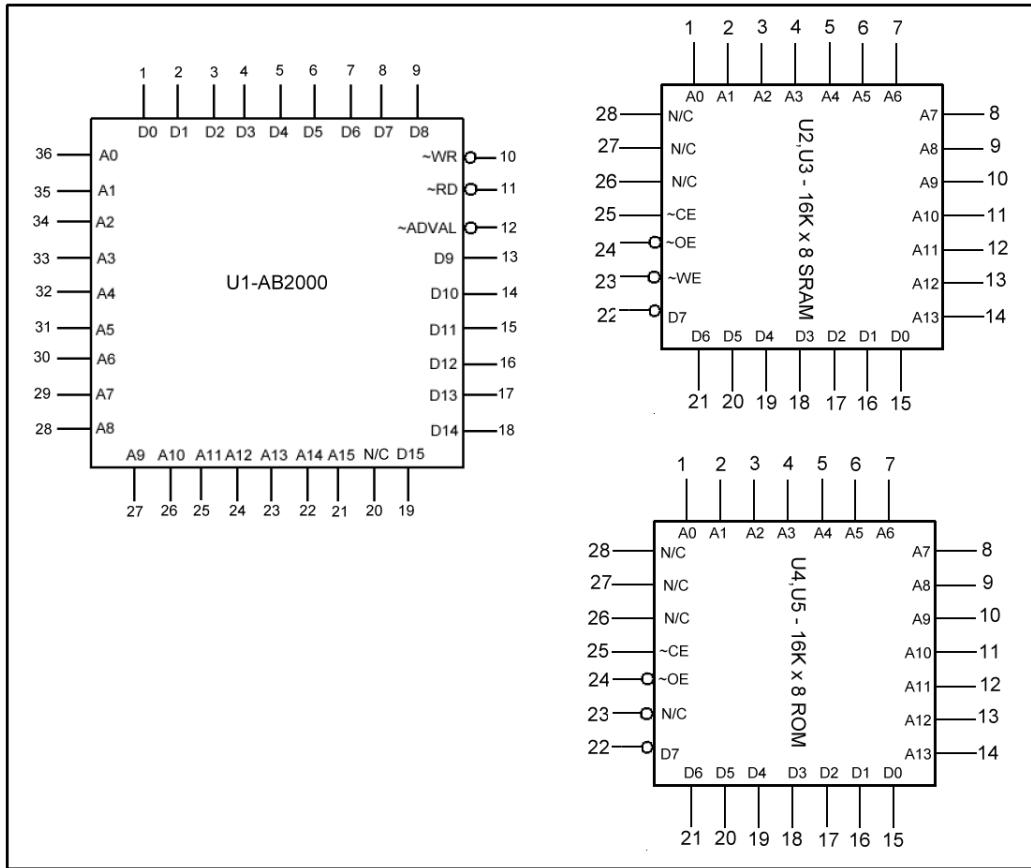
d- What is the total number of memory devices required in this memory design?

e- Why would a memory system design based upon this type of a memory device not be capable of addressing memory locations at the byte-level? Discuss the reason for your answer in a sentence or two.



7- Assume that you are the chief hardware designer for the **Soul of the City Bagel and Flight Control Systems Company**. Your job is to design a computer to memory sub-system for a new, automatic, galley and bagel maker for the next generation of commercial airliners now being designed. The microprocessor is the AB2000, a hot new chip that you've just been waiting to have an opportunity to use in a design.

The processor has a 16-bit address bus and a 16-bit data bus as shown in the figure below (For simplicity, the diagram only shows the relevant processor signals for this exercise problem). Also shown in the figure are the schematic diagrams and pin designations for the ROM and SRAM chips you will need in your design.



The memory subsystem has 3 status signals that you must use to design your memory array:

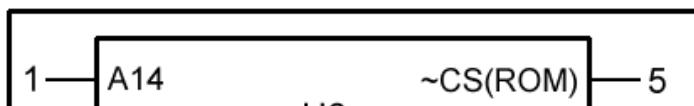
- /WR: Active low, it indicates a write to memory cycle
- /RD: Active low, it indicates a read from memory cycle
- /ADVAL: Active low, it indicates that the address on the bus is stable and may be considered to be a valid address

The memory chips occupy the following regions of memory:

- ROM from 0x0000 through 0x3FFF
- SRAM from 0xC000 through 0xFFFF
- Other stuff (not your problem) from 0x4000 through 0xBFFF

(a) Design the memory decoder circuit that you will need for this circuit. It must be able to properly decode the memory chips for their respective address ranges and also decode the processor's /WR, /RD and /ADVAL signals to create /OE, /CE, and /WR. Hint, refer to the text sections on the memory system to get the equations for /OE, /CE and /WR.

Assume that the circuit that you design will be fabricated into a memory interface chip designed, U6. Since the printed circuit designer needs to know the pin designations for all the parts, you supply the specification of the pin functions for U6 shown, below. You will



then design the memory decoder to these pin designations. Finally, for simplicity, you decide not to implement byte addressability. All memory accesses will be word-width accesses.

(b) Create a Net List for your design. A sample net list is shown in the figure, below. The net list is just a table of circuit interconnections or "nets". A net is just the common connection of all I/O pins on the circuit packages that are connected together. Thus, in this design, you might have a net labeled "ADDR13", which is common to: Pin #23 on chip number U1 (shorthand U1-23), U2-14, U3-14, U4-14, U5-14. The net list is used by printed circuit board manufacturers to actually build your PC board. A start for your net list is shown below:

Net name							
addr0	U1-36	U2-1	U3-1	U4-1	U5-1		
addr1	U1-35	U2-2	U3-2	U4-2	U5-2		
addr2	U1-34	U2-3	U3-3	U4-3	U5-3		
addr3	U1-33	U2-4	U3-4	U4-4	U5-4		
•							
•							
data0	U1-1	U2-15	U4-15				

8- Complete the analysis for each of the following situations:

a- A microprocessor with a 20-bit address range using 8 memory chips with a capacity of 128K each: Build a table showing the address ranges, in hexadecimal, covered by each memory chip.

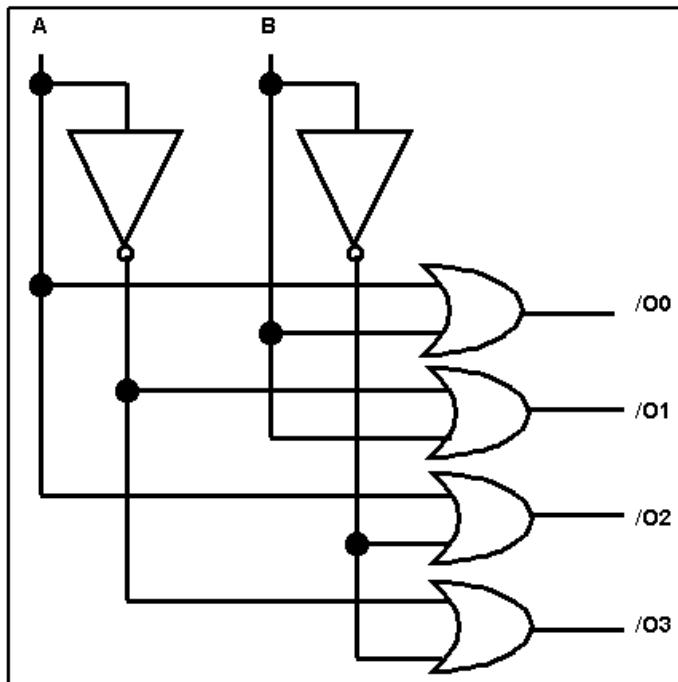
b- A microprocessor with a 24-bit address range and using 4 memory chips with a capacity of 64K each: Two of the memory chips occupy the first 128K of the address range and two chips occupy the top 128K of the address range. Build a table showing the address ranges, in hexadecimal, covered by each of the chips. **Notice that there is a big address range in the middle with no active memory in it.** This is fairly common.

c- A microprocessor with a 32-bit address range and using 8 memory chips with a capacity of 1M each ($1M = 1,048,576$): Two of the memory chips occupy the first 2M of the address range and 6 chips occupy the top 6M of the address range. Build a table showing the address ranges, in hexadecimal, covered by each of the chips.

d- A microprocessor with a 20-bit addressing range and using 8 memory chips of different sizes. Four of the memory chips have a capacity of 128K each and occupy the first 512K consecutive addresses from 00000 on. The other four memory chips have a capacity of 32K each and occupy the topmost 128K of the addressing range. Build a table showing the address ranges, in hexadecimal, covered by each of the chips.

Solution to the Chapter 6 exercises.

1- The gate design is shown below. Note that this is more a problem of conversion from negative to positive logic than anything else.



2- The truth table is shown below:

The first thing to notice is that the output side of the truth table is almost all 1's. This could make the K-Map reduction really tedious. However, there is hope. Suppose that on the output side we change the sense of the logic. Let's make 0 true and 1 false. Suddenly, the truth table gets a lot simpler. In fact, it becomes so simple we don't need a K-Map because each output variable is true for only one set of conditions (one row) of the input variables.

By inspection, the equations are:

$$\sim W_0 = \sim A_0 * \sim A_1 * \sim (R/\sim W) * \sim CS$$

$$\sim W_1 = A_0 * \sim A_1 * \sim (R/\sim W) * \sim CS$$

$$\sim W_2 = \sim A_0 * A_1 * \sim (R/\sim W) * \sim CS$$

$$\sim W_3 = A_0 * A_1 * \sim (R/\sim W) * \sim CS$$

$$\sim CS_0 = \sim A_0 * \sim A_1 * (R/\sim W) * \sim CS$$

$$\sim W_1 = A_0 * \sim A_1 * (R/\sim W) * \sim CS$$

$$\sim W_2 = \sim A_0 * A_1 * (R/\sim W) * \sim CS$$

$$\sim W_3 = A_0 * A_1 * (R/\sim W) * \sim CS$$

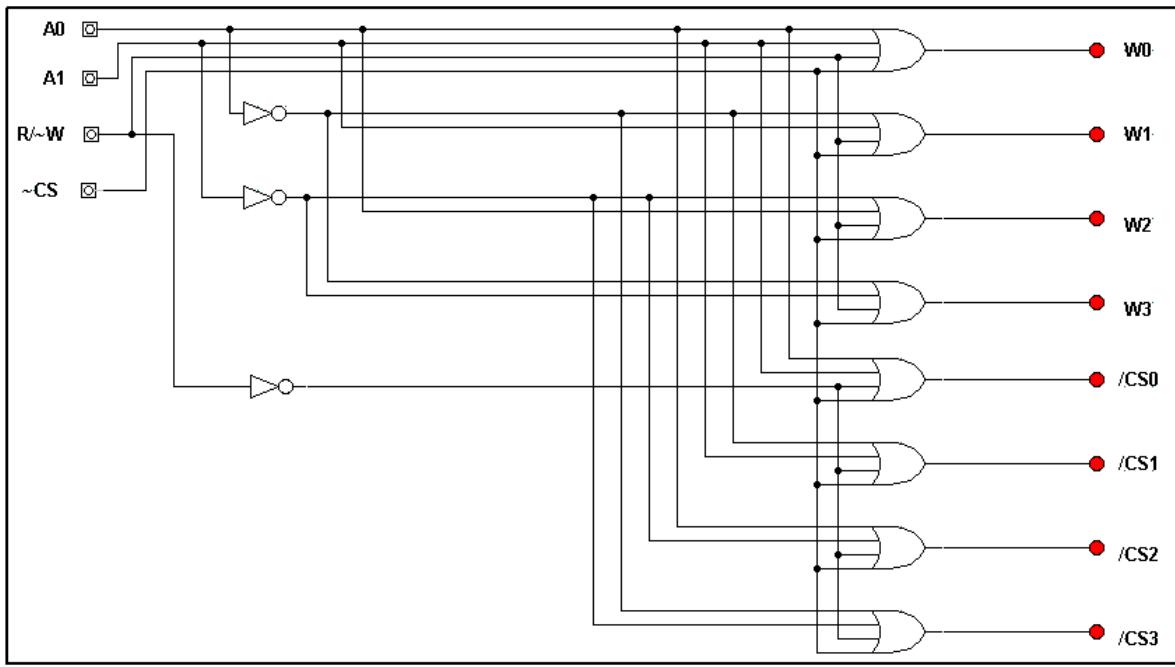
Now, let's see what we can do about simplifying these equations. Recall DeMorgan's Theorem. Consider the logical equation for W_0 .

$$\sim W_0 = \sim A_0 * \sim A_1 * \sim (R/\sim W) * \sim CS$$

By DeMorgan's Theorem, this is the same as: $\sim W_0 = \sim (A_0 + A_1 + R/\sim W + CS)$

$$W_0 = A_0 + A_1 + R/\sim W + CS$$

That's pretty simple, we've gone from an OR'ing or AND terms with lots of NOT gates to a simple OR term.



3- Part a

26 address bits means that the microprocessor can address 64 MBytes of memory. If the memory chip is organized as 512K x 8 bits, then it needs 19 address lines in order to address all 512K memory locations. Since the microprocessor has a 32-bit data bus it needs 4 chips per page with each chip holding 512K bytes. Thus, each page holds 2 MBytes.

Since we have a total addressable memory of 64 MBytes then we must have 32 pages of memory with each page holding 2 Mbytes. Therefore we need to have 4 chips per page multiplied by 32 pages, or **128** chips in order to completely fill the memory.

Part b

Page number	Starting Address(hex)	Ending Address (hex)
0	0000000	007FFFF
1	0080000	00FFFFFF
2	0100000	017FFFF

4- A clock frequency of 50 MHz has a clock period of 20 nanoseconds. From the timing diagram we see that ~RD, ~WR and ~ADDRVAL go low on the falling edge of T1 and they rise again to end the cycle at the falling edge of T3. Since this is two clock periods, the slowest memory that will work reliably must have an access time of at least 40 nanoseconds. Anything with an access time of 40 nanoseconds or less will work in this application.

5a- Direct memory access: A method of improving the efficiency of data transfers between a peripheral device and the computer's memory. The DMA process allows a peripheral device to take control of the memory bus while the processor idles and the peripheral handles the data transfer directly to memory, bypassing the processor.

5b- Tri-state logic: A circuit design that adds an additional output control to memory or other devices that enables their outputs to be tied together on busses. When the tri-state logic turns off the output of the device, the output presents a high impedance to the bus. In other words, it is as if it isn't connected to the bus, thus enabling another output to drive the bus.

5c- Address bus, data bus, status bus: These are the three main busses of the processor. The address bus presents the address of the next memory operation to the memory system. It is unidirectional, that is all signals are outputs from the processor to memory. The data bus is bi-directional. Data flows into the processor and out to memory on the same bus signals. The status bus is heterogeneous. Some signals are input only, some are output only and others are bi-directional. The status bus carries all of the housekeeping signals of the processor.

6a- Since the memory device has 17 address lines, it contains 128K of addressable memory.

6b- Each memory location holds 16 bits, or 2^4 bits. Thus 2^{17} memory locations X 2^4 bits per memory location = 2^{21} bits, or 2M bits.

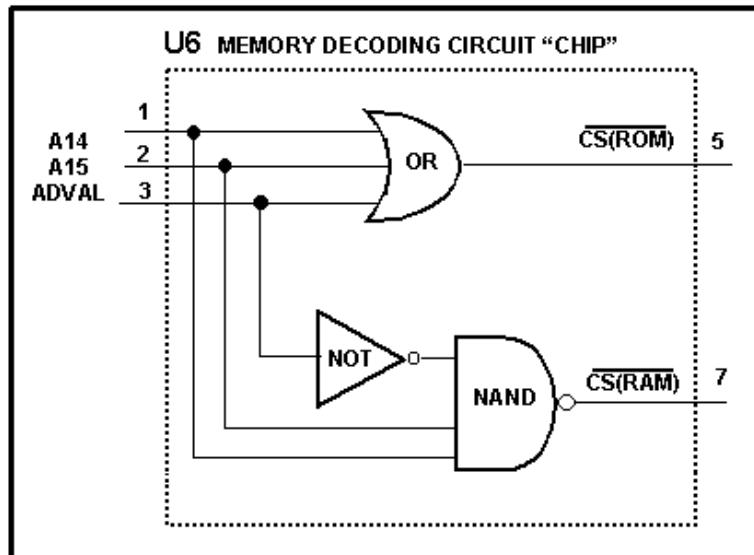
6c-

Page	Starting Address	Ending Address
0	00000	1FFFF
1	20000	3FFFF
7	E0000	FFFFF

6d- 6 Devices (3 pages x 2 devices per page).

6e- Since each memory device has 16 lines for data and only one write line, ~WE, it does not appear to be capable of writing only one byte. Thus, the minimum data quantity that you could read or write from each memory device is a word of data.

7a- The cir



7b- The netlist is shown below:

Net name					
addr0	U1-36	U2-1	U3-1	U4-1	U5-1
addr1	U1-35	U2-2	U3-2	U4-2	U5-2
addr2	U1-34	U2-3	U3-3	U4-3	U5-3
addr3	U1-33	U2-4	U3-4	U4-4	U5-4
addr4	U1-32	U2-5	U3-5	U4-5	U5-5
addr5	U1-31	U2-6	U3-6	U4-6	U5-6
addr6	U1-30	U2-7	U3-7	U4-7	U5-7
addr7	U1-29	U2-8	U3-8	U4-8	U5-8
addr8	U1-28	U2-9	U3-9	U4-9	U5-9
addr9	U1-27	U2-10	U3-10	U4-10	U5-10
addr10	U1-26	U2-11	U3-11	U4-11	U5-11
addr11	U1-25	U2-12	U3-12	U4-12	U5-12
addr12	U1-24	U2-13	U3-13	U4-13	U5-13
addr13	U1-23	U2-14	U3-14	U4-14	U5-14
addr14	U1-22	U6-1			
addr15	U1-21	U6-3			
data0	U1-1	U2-15	U4-15		
data1	U1-2	U2-16	U4-16		
data2	U1-3	U2-17	U4-17		
data3	U1-4	U2-18	U4-18		
data4	U1-5	U2-19	U4-19		
data5	U1-6	U2-20	U4-20		
data6	U1-7	U2-21	U4-21		
data7	U1-8	U2-22	U4-22		
data8	U1-9	U3-15	U5-15		
data9	U1-13	U3-16	U5-16		
data10	U1-14	U3-17	U5-17		
data11	U1-15	U3-18	U5-18		
data12	U1-16	U3-19	U5-19		
data13	U1-17	U3-20	U5-20		
data14	U1-18	U3-21	U5-21		
data15	U1-19	U3-22	U5-22		
~ADVAL	U1-12	U6-3			
~WR	U1-10	U2-23	U3-23		
~RD	U1-11	U2-24	U3-24	U4-24	U5-24
~CSROM	U6-5	U4-25	U5-25		
~CSRAM	U6-7	U2-25	U3-25		

8- a- A microprocessor with a 20-bit address range using 8 memory chips with a capacity of 128K each. The address range of the processor is 00000..FFFFF and the address range of each memory chip is 00000..1FFFF

Memory chip	Address Range
1	00000..1FFFF
2	20000..3FFFF
3	40000..5FFFF
4	60000..7FFFF
5	80000..9FFFF
6	A0000..BFFFF
7	C0000..DFFFF
8	E0000..FFFFF

b- A microprocessor with a 24-bit address range and using 4 memory chips with a capacity of 64K each. Two of the memory chips occupy the first 128K of the address range and two chips occupy the top 128K of the address range.

This is a bit tougher. The address range of the processor is 000000..FFFFFF and each of the memory chips has an address range of 0000..FFFF. The address table looks like this.

Memory chip	Address Range
1	000000..00FFFF
2	010000..01FFFF
3	FE0000..FEFFFF
4	FF0000..FFFFFF

c- A microprocessor with a 32-bit address range and using 8 memory chips with a capacity of 1M each. (1M= 1,048,576). Two of the memory chips occupy the first 2M of the address range and 6 chips occupy the top 6M of the address range. This gets a bit tougher still.

Memory chip	Address Range
1	0000000..000FFFFF
2	0010000..001FFFFF
3	FFA0000..FFAFFFFF
4	FFB0000..FFBFFFFF
5	FFC0000..FFCFFFFF
6	FFD0000..FFDFFFFF
7	FFE0000..FFEEFFFF
8	FFF0000..FFFFFFFF

d- A microprocessor with a 20-bit addressing range and using 8 memory chips of different sizes. Four of the memory chips have a capacity of 128K each and occupy the first 512K consecutive addresses from 00000 on. The other four memory chips have a capacity of 32K each and occupy the topmost 128K of the addressing range

Memory chip	Address Range
1	00000..1FFFF
2	20000..3FFFF
3	40000..5FFFF
4	60000..7FFFF
5	E0000..E7FFF
6	E8000..EFFFF
7	F0000..F7FFF
8	F8000..FFFFFF