Introduction to Digital Logic

Norman Matloff
University of California at Davis
© 1999, 2003, N.S. Matloff

September 4, 2003

Contents

1 Overview **3**

2 Combinational Logic **3**

2.1 A Few Basic Gates 3

2.1.1 AND Gates 3

2.1.2 OR Gates 4

2.1.3 NOT Gates 4

2.1.4 NAND Gates 4

2.1.5 NOR Gates 5

2.1.6 XOR Gates 5

2.2 Some MSI Combinational Components 6

2.2.1 Multiplexors 6

2.2.2 Decoders 7

2.3 Examples 7

2.3.1 Half Adder 7

2.3.2 Full Adder 7

2.3.3 2-Bit Adder 8

2.4 Timing 9

3 Sequential Logic **9**

3.1 Latches and Flip-Flops 10
3.2 Edge-Triggering 11
3.3 Example: A 2-Bit Ripple Counter 12
3.4 Example: Tracking Counts Mod and Div 5 13

4 Bus-Based Circuits 14

5 Example: Memory Chips and Systems 17

5.1 An SRAM Memory Chip 17
5.2 A Memory System 20
5.3 Memory Interleaving 21
5.4 DRAMs 21

6 Example: A Simple CPU 21

1 Overview

As you know, all information inside a computer is processed and stored as 0-1 bits. Here we will look at the basic building blocks used to manipulate this 0-1 information.

2 Combinational Logic

The term **combinational logic** refers to circuitry that transforms bits, as opposed to storing bits. For example, the ALU portion of a CPU transforms data, e.g. transforming two input word-sized bit strings into an output which is the sum of the two inputs.

2.1 A Few Basic Gates

2.1.1 AND Gates

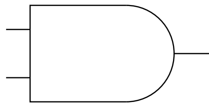
A basic AND gate has two inputs and one output.[1] Let's call the two inputs X and Y, and the output Z. Then Z = 1 if and only if X = 1 and Y = 1, hence the name "AND."

The AND operation is represented in **boolean equation** settings by multiplication, i.e. we write

$$Z = X\,Y \tag{1}$$

As you can see, this equation does succinctly summarize the AND operation: For example, if X and Y are both 1, then since $1 \times 1 = 1$, then Z will be 1 too. If on the other hand, for instance, X = 1 but Y = 0, Z will be 0.

The standard symbol for an AND gate is



Note that for any X (0 or 1),

$$0\,X = 0 \tag{2}$$

and

$$1\,X = X \tag{3}$$

[1] Versions with **fan-in** of more than two, i.e. having more than two inputs, exist too.

2.1.2 OR Gates

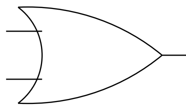
Again, a basic OR gate will have two inputs, but in this case Z = 1 if and only if X = 1 or Y = 1 (which includes the case in which both X and Y are 1).

The boolean equation is

$$Z = X + Y \tag{4}$$

where the '+' is standard addition except that 1 + 1 is taken to be 1.

The standard symbol for an OR gate is



Note that for any X (0 or 1),

$$0 + X = X \tag{5}$$

and

$$1 + X = 1 \tag{6}$$

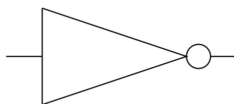
2.1.3 NOT Gates

A NOT gate has one input X and one output Z, with the output being the logical negation of the input. In other words, an input of 1 produces an output of 0, and vice versa.

In boolean equations, a NOT operation is indicated by an overbar:

$$Z = \overline{X} \tag{7}$$

The standard symbol for a NOT gate is:



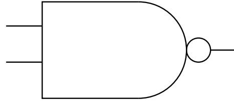
2.1.4 NAND Gates

Here there are two inputs X and Y, and one output Z. The term "NAND" stands for "not-and," meaning that Z = 1 if the statement "X = 1 and Y = 1" is not true.

The boolean equation is

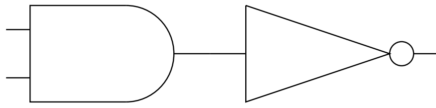
$$Z = \overline{(X\,Y)} \tag{8}$$

The standard symbol for a NAND gate is



Note that the little circle here means “not.”

Obviously, if on the day on which we shopped at the Gates ’R Us store they were out of NAND gates, we could synthesize a NAND by using an AND together with a NOT:



But of course, this would not be so desirable as using a real NAND. The synthesized version would probably have more transistors than the real one, and thus would be slower and take up more space on a chip, thus reducing the total number of gates we could put on the chip.

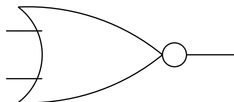
2.1.5 NOR Gates

Again, inputs X and Y, output Z, with Z being equal to 1 if the statement “X = 1 or Y = 1” is not true.

The boolean equation is

$$Z = \overline{(X + Y)} \tag{9}$$

The symbol for a NOR gate is:



Again, the same effect could be synthesized by leading the output of an OR into a NOT.

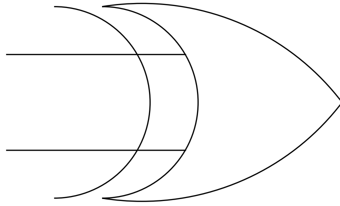
2.1.6 XOR Gates

Here we have inputs X and Y, output Z, with Z being equal to 1 if the statement “X = 1 or Y = 1 but not both” is true. The term used for this is “exclusive-or,” abbreviated to XOR.

The boolean equation is

$$Z = \overline{X}Y + X\overline{Y} \tag{10}$$

The symbol for an XOR gate is:



Again, the same effect could be synthesized by using two NOT gates, two AND gates and an OR gate.

2.2 Some MSI Combinational Components

"MSI" stands for "medium-scale integration." We are integrating a moderate number of gates to form some frequently used building blocks.

2.2.1 Multiplexors

A multiplexor, or MUX, selects one of its data inputs and copies that input to the output, with the selection being made according to its address input.

As a simple example, consider a MUX having two 1-bit data inputs, D1 and D0. To indicate which one we want to be copied to our output, we need another input, A; A = 1 will mean we want D1, and A = 0 will mean we want D0.

Let's call the output Z. Then the equation for Z is

$$Z = A\ D1 + \overline{A}\ D0 \tag{11}$$

(Make SURE this makes sense to you.)

While it won't be drawn here, you should be able to see above that we could construct this MUX by using two AND gates, one NOT, and one OR.

A MUX with four data inputs, D3, D2, D1 and D0 would need two selector bits, A1 and A0, and the output would be

$$Z = A1\ A0\ D3 + A1\ \overline{A0}\ D2 + \ \overline{A1}\ A0 D1 + \overline{A1}\ \overline{A0}\ D0 \tag{12}$$

(Again, make SURE this makes sense to you.)

2.2.2 Decoders

This is best explained by example, say for a 3-to-8 decoder. Let's call the 3-bit input lines X2, X1 and X0, and the output lines Z7, Z6, ..., Z1 and Z0. The 3-bit input can be considered the binary coding for one of the numbers 0-7. The output lines then tell us which one of the numbers 0-7 is represented by the input. For example, suppose X2 = 0, X1 = 1 and X0 = 1, representing the number 3; then Z3 will be 1, to indicate that fact, and all the other Zi will be 0.

Building a decoder from gates is quite straightforward from the equations, which themselves are also straightforward. For instance, from the example above you can see that the equation for Z3 is

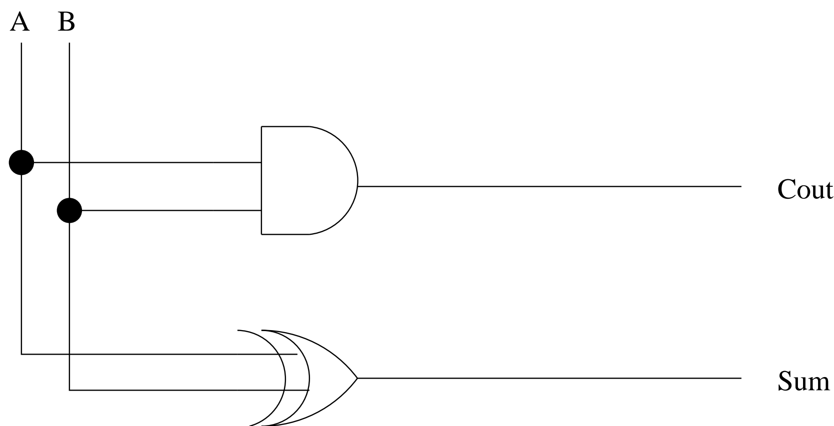
$$Z3 = \overline{X2}\, X1\, X0 \tag{13}$$

2.3 Examples

2.3.1 Half Adder

Here we will design logic to add two 1-bit numbers A and B together. Let's call the sum bit Sum. But note also that there may be a carry generated (this will happen if both addends are equal to 1); we will call this Cout, for "carry output."

The logic will look like this:



(The dark circles represent wire connections. If two lines cross in the picture but there is no dark circle at their intersection, then they do not touch each other.

2.3.2 Full Adder

A full adder has one more input than does a half adder. We will call this input Cin, for "carry in." The reason we need this extra input is that we will be using a full adder as a building block to do multi-bit addition. For example, consider the following addition of two 3-bit numbers, 011 and 001:

```
11
011
001
---
100
```

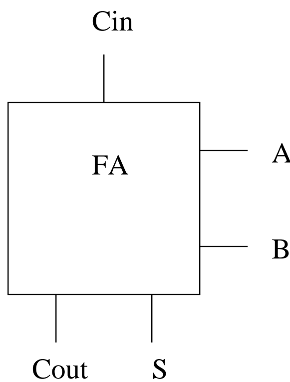
Let's refer to the bit positions as 2, 1 and 0, from left to right. The point then is that the addition at position 0 resulted in a carry into position 1 (shown in the picture), and that carry must be incorporated into the sum performed at position 1. That carry would be the Cin for position 1 (and the Cout for position 0).

We will not draw the logic, but here are the equations (remember, we are now back to a single bit, even though the logic will be used below as a building block for a multi-bit adder):

$$Sum = \overline{A}\,\overline{B}\,Cin + \overline{A}\,B\,\overline{Cin} + A\,\overline{B}\,\overline{Cin} + A\,B\,Cin \tag{14}$$

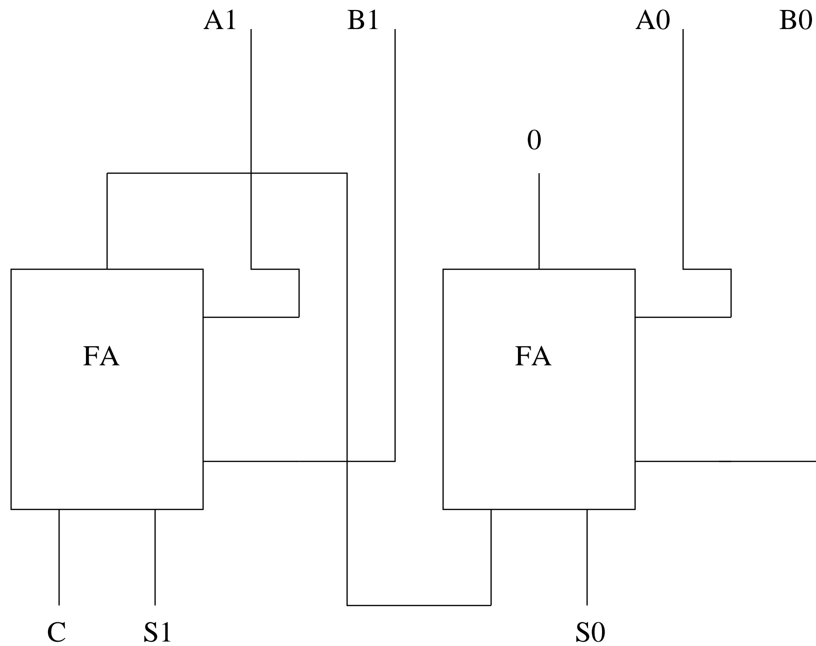
$$Cout = \overline{A}\,B\,Cin + A\,\overline{B}\,Cin + A\,B\,\overline{Cin} + A\,B\,Cin \tag{15}$$

Let's use the following (not standard) symbol for a full adder:



2.3.3 2-Bit Adder

We can put two full adders together to form a 2-bit adder, i.e. logic which will add together two 2-bit inputs, producing a 2-bit sum and a possible carry:



Here (A1,A0) forms the first 2-bit addend, and (B1,B0) forms the second. The sum is (S1,S0), and the carry (into bit position 3) is C. Note that a constant 0 is hardwired into the Cin input of the full adder on the right.

2.4 Timing

The delay of a typical gate is on the order of 10 nanoseconds (ns), i.e. 10 billionths of a second.[2] This sounds extremely fast, almost beyond human imagination, but in view of the fact that computers perform tens of millions of operations per second, gate delays do add up into tangible amounts of time, and thus directly affect the overall speed of the machine. To get the fastest machine, digital logic must be optimized. In other words, even though several sets of gates may be equivalent in effecting a certain function (say addition), their timings may differ considerably, and it is desire to find an optimal set. Many techniques, such as **Karnaugh maps**, exist to do this.

Note that in the 2-bit adder example above, the overall delay is approximately double the delay of a single full adder, since the left full adder must wait for the outputs of the right one to be valid; before that time, the outputs of the left one are garbage. There are other adder designs, such as **carry-lookahead adders**, which aim to circumvent this problem.

3 Sequential Logic

Sequential logic stores data. Registers in a CPU, RAM and so on store data.

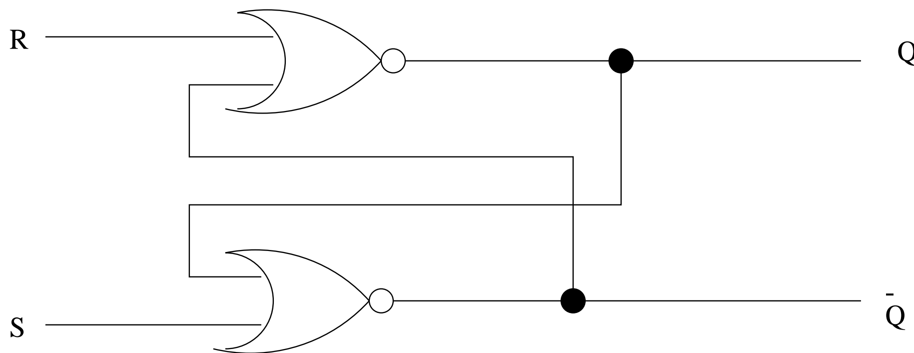
[2]It can be faster than this, depending on the **technology** used, meaning the type of electronics used to implement the gate. Since we do not address anything at the electronics level here, we will not pursue this any further.

3.1 Latches and Flip-Flops

We first look at the **S-R latch**. It has two inputs, R and S, and two outputs, Q and $\overline{Q}$. Its function is that of a 1-bit memory, with Q being the bit currently stored.[3]

Whenever we want to store a new bit in the latch, replacing the old one, we simply pulse a 1 on R or S momentarily, depending on whether we want to store a 0 or a 1 in the latch. (R and S stand for "reset" and "set.") As long as R and S stay at 0, the stored value will remain as is.

An S-R latch can be constructed as follows:



Suppose, for instance, that currently Q = 1, and we wish to change Q to 0. That would mean momentarily pulsing the R line to 1. Let's see that this does indeed work:

Since Q = 1, $\overline{Q}$ will be 0. Thus the two inputs to the upper NOR gate will be 1 (from R) and 0 (from $\overline{Q}$), making the output 0. In other words, Q does change to 0, as desired.

What about $\overline{Q}$? It will stay at 0 for a short time, but as soon as Q changes to 0 and feeds that value back into the lower NOR, the output of that lower NOR will now be 1 (since S = 0). In other words, $\overline{Q}$ does indeed change to 1.

But that is not the end of the story, for we must make sure that those new values of Q and $\overline{Q}$ are maintained. Well, the feedback nature of the circuit has exactly that result. For example, after $\overline{Q}$ changes to 1, that ensures that the output of the upper NOR gate is 0 (regardless of what R is), so Q will indeed continue to stay at 0. Similarly, you should check that the design ensures that $\overline{Q}$ will stay at 1, until such time as S is pulsed.

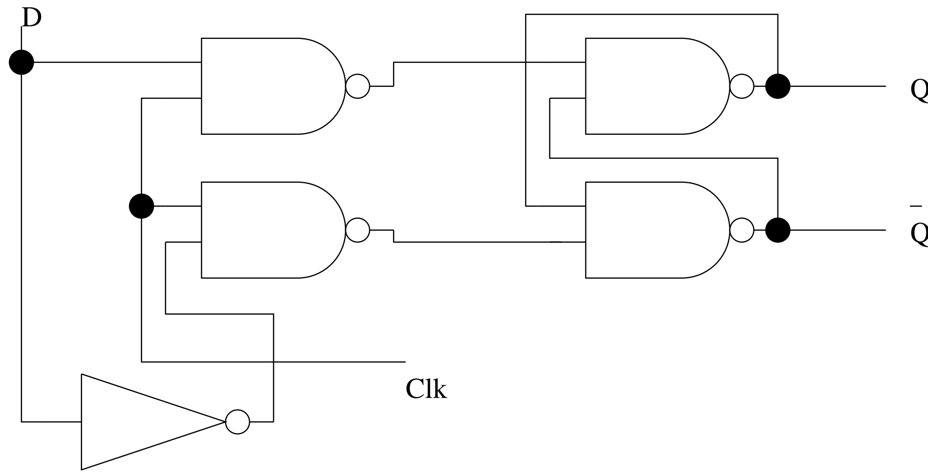
Flip-flops are like latches, except that they are **clocked**, so that they accept new input only at certain times. A clock is a crystal device that pulses at regular intervals, sending 1,0,1,0,1,.... For example, a 1 gigaherz PC has a clock which pulses 1 billion times per second.[4]

[3] $\overline{Q}$ is then the negation of the bit being stored. Since this is generated anyway, due to the nature of the logic used in construction we get it "for free," and thus might as well formally make it one of the outputs. That way if $\overline{Q}$ happens to be needed elsewhere in our machine, we would be able to save a NOT gate there.

[4] Of course, the faster the clock, the faster the machine. However, in choosing the clock speed, we have to account for all gate delays, signal propagation times along wires, and so on, so that all signals reach their destinations within one clock cycle. In other words, we must choose the clock cycle to accommodate the longest delay in the overall circuit. Actually, if the designer is extremely careful, it may be possible to exceed this limitation, but you can see that one cannot arbitrarily increase clock speed on a given chip.

Flip-flops, by virtue of having clocked input acceptance, allow the designer much more convenient control as to when we allow Q to change, as you will see below.

A D flip-flop can be constructed as follows:



The D input ("data") is the new value to be stored at the time of the clock pulse. You should "walk through" an instance of the operation of this circuit, again say where the value 1 had been stored originally (Q = 1), but in which we want the new value to be 0 (D = 0). Trace through the sequence of events which will occur when the clock pulse comes; you will find that $\overline{Q}$ is the first to change, becoming 1, with its feedback into the upper-right NAND making Q change to 0, as desired.

Keep in mind the role of the clock here. At many times the value at D will be garbage, and we don't want it to affect Q. The only time that it can affect Q is when the clock pulses. That already gives us a bit more control, but even more importantly, we can design our circuit so that the clock pulse itself is not connected to the clock input of a flip-flop, but rather that pulse is AND-ed with some other wires that represent conditions under which the input data D is valid. That way we insure that Q will change only when D is valid. You will see an example of this later in this tutorial, where we build a RAM circuit.

3.2 Edge-Triggering

Many flip-flops are **edge-triggered**. What this means is that they are designed in such a way that an input value (labeled D in the picture above) will have effect on the flip-flop only during a narrow window of time, specifically the time during which the clock pulse is rising or falling.[5]

This is done to avoid feedback problems in complex circuits. The output of a flip-flop may be routed through a series of gates and ultimately fed back in to the same flip-flop as an input. For instance, consider the Intel assembly-language instruction[6]

```
addl %ebx, %eax
```

[5]These are called the **leading edge** and **trailing edge** or **falling edge** of the clock pulse.
[6]Using the UNIX syntax, i.e. AT&T.

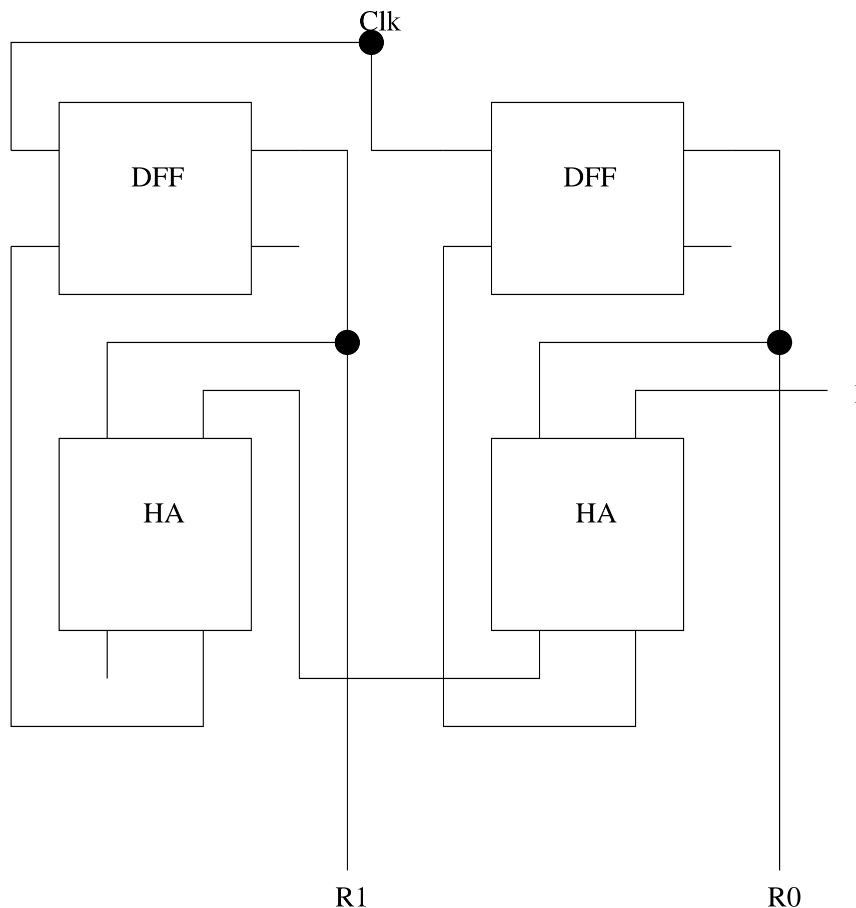
(If you have not worked with Intel machines before, this instruction adds the values in the EAX and EBX registers, and stores the sum back into EAX.)

Suppose EAX and EBX originally contain the values 5 and 2, respectively. The new value in EAX should be 7. But if the circuitry were poorly designed, the addition might continue, with the 7 being added to 2, thus putting 9 into EAX, and so on. By limiting the sensitivity of flip-flops[7] to very narrow windows of time, this feedback problem cannot occur.

3.3 Example: A 2-Bit Ripple Counter

Recall that an n-bit string can represent (unsigned) integers in the range 0, 1, 2, ..., $2^n - 1$. An n-bit **ripple counter** is simply a counter, which will continually cycle through these values. For example, a 2-bit ripple counter will cycle through 0, 1, 2, 3, 0, 1, 2, 3, 0,..., or in bit form, 00, 01, 10, 11, 00, 01, 10, 11, 00, ...

We can construct such a counter from two D flip-flops, two half adders, and a clock:



(In the DFF boxes, the two left inputs are D and Clk, while the two right outputs are Q and $\overline{Q}$.)

[7]Again, remember that registers are made up of flip-flops.

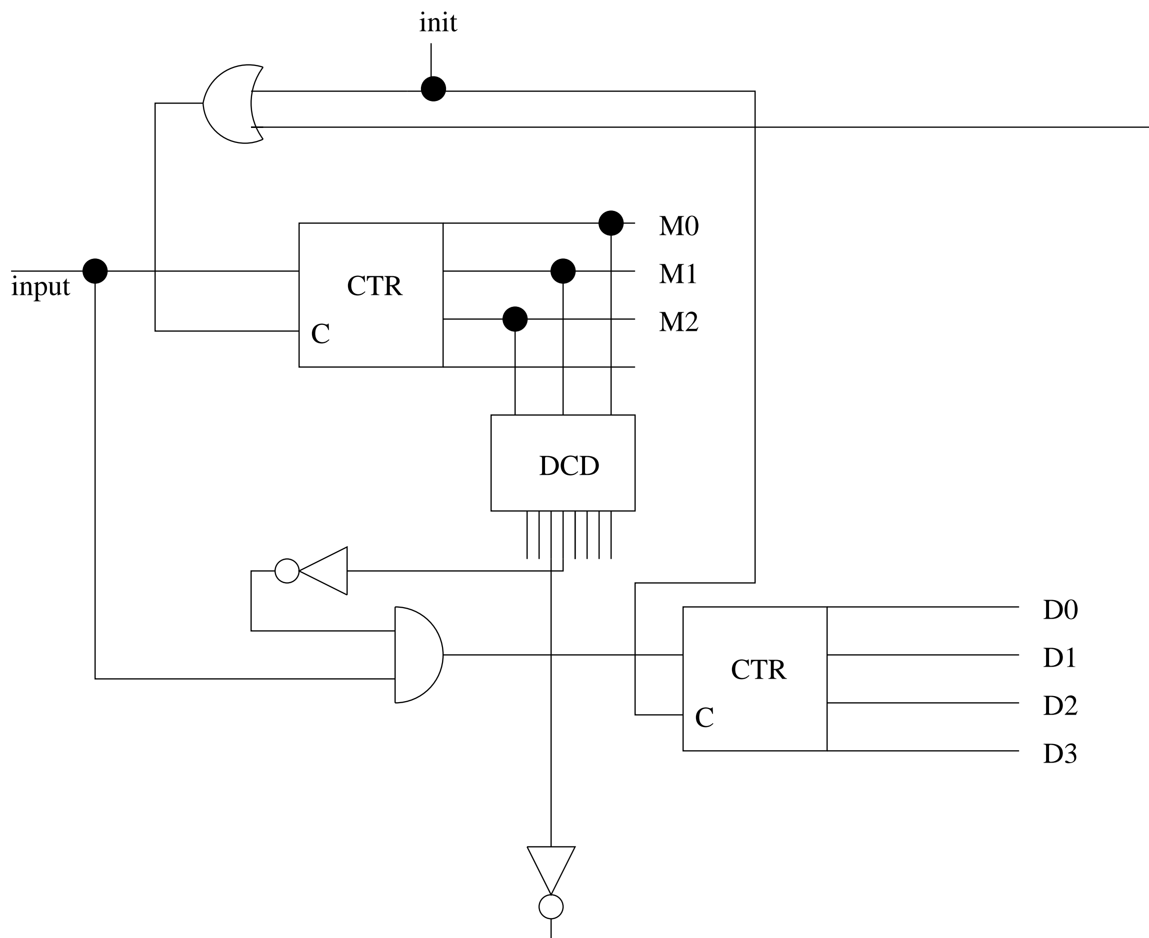
Note the following:

- the constant 1 is hard-wired as one of the inputs to the right-hand half adder
- we are ignoring the $\overline{Q}$ outputs of the flip-flops
- the outputs are R1 and R0; e.g. when the count is 2, we will have R1 = 1 and R0 = 0, since 10 is the binary representation of 2
- the input labeled Clk does not actually have to be connected to a clock, and in most applications will not be; instead, the input will simply be a line which we have arranged (elsewhere in the circuit) to pulse to a 1 every time some particular event of interest occurs

3.4 Example: Tracking Counts Mod and Div 5

The circuit to be designed here will have an input line like a ripple counter does, but instead of tracking the raw count c of the number of times the input line is pulsed, we will track c mod 5 and c div 5.

Here is the circuit:



The input line is visible at the left of the picture. Its quiescent state is 0, but sometimes pulses, i.e. 1s, come in on that line. (The pulses might come at regular intervals, such as from a clock, or at irregular times, depending on what application we needed this circuit for, i.e. depending on what we are counting.) The output pins are M0, M1 and M2, which contain c mod 5, and D0, D1, D2 and D3, which contain c div 5 (though only up to 15 for the latter).

There are two boxes labeled CTR; these are 4-bit ripple counters. The pins labeled C in them are for clearing, i.e. resetting; if this pin is pulsed, all bits of the ripple counter will be reset to 0. Similarly, there is an "init" input at the top of the picture, to clear both the mod and div counters.

The box labeled DCD is a 3-to-8 decoder. Its output pins are **active low**, meaning that 0 means "yes" and 1 means "no." For example, let's label these pins Z7,Z6,...,Z0 from left to right. Then Z7 will be equal to 0 if and only if the three input pins contain 111, the binary representation of 7.

Whenever the mod counter reaches 4, Z4 will be 0, which means that the output of the NOT gate we've connected to Z4 will be 1. This 1 is then ANDed with the circuit's input line. So, the next time the input line is pulsed—which will be a count which is a multiple of 5—a pulse will be felt at the c div 5 counter. That counter will then increment, exactly what we want.

Note that the gate delay in DCD is helping us here. When the count is 4, the next pulse will change the count momentarily to 5 (and the rest of the circuit will change that to 0). However, even when the count first becomes 5, Z4 will still be equal to 0 (indicating a count of 4, not 5) for a short period of time equal to the gate delay in DCD. This delay is good, because when the pulse comes at count 4, we want Z4 to stay at 0 long enough so that it makes CTR increment. This illustrates the delicate timing issues which can arise in digital circuits.

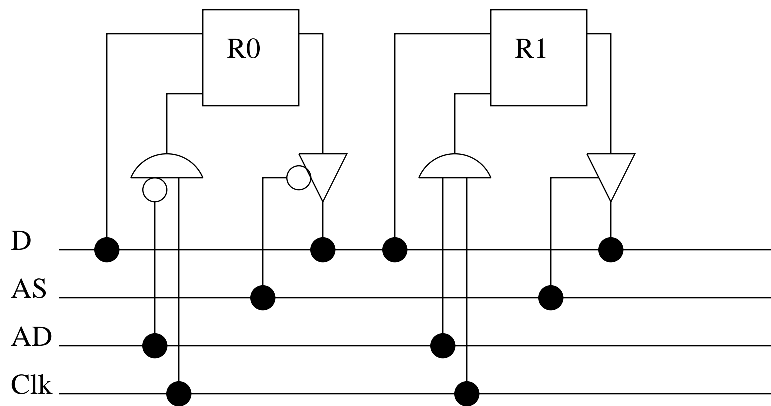
4 Bus-Based Circuits

A **bus** is a set of parallel wires used for transfer of data among various components. You are already familiar with the idea of a **system bus** for a computer, which connects components such as the CPU, memory and I/O devices.[8]

Since several components are attached to the same bus, how do we make sure that only one of them actually is connected to the bus at a time? The answer is that we use **tri-state buffers** which can connect a component to a bus, or electrically isolate the component away from the bus.

To illustrate this, consider the design of a very simple CPU which, for simplicity of exposition, will have only two registers, R0 and R1, each only one bit wide, implemented as a DFF:

[8] In this context, the term **bus** often connotes not just the wires themselves, but also standards for the roles played by the wires, electrical characteristics, and so on.

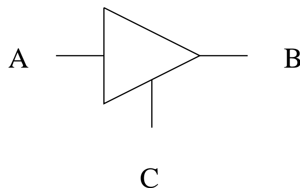


In each register, the upper left input is for data, the lower left for the clock, and output is out the right side.

Again, keep in mind that the bus shown here is inside the CPU, different from the system bus; the purpose of the bus here is for data transfer from one register (the **source**) to another (the **destination**). (There also would be an ALU, etc., but we do not show other items here.)

We see data (D) and address (AS, AD) lines here, just as we would for a system bus. However, due to the simple nature of our example, in which our word size is only one bit (!), we only need one data line; if we had say, 32-bit words, we would need 32 data lines, D31,D30,...,D0. Similarly, since we have only two registers, we only need one address line for the source register, and one for the destination register; if we had say, 16 registers, we would need four address lines each for source and destination. I have chosen the name "AS" for "address of source," and "AD" for "address of destination," thinking of R0 and R1 having addresses 0 and 1, respectively.

What is new here, though, is the presence of triangles which look like NOT gates but are instead tri-state buffers; the latter are distinguished from the former by the presence of an extra input line coming in at the side of the triangle, i.e. entering at a sloped leg of the triangle:



The operation is very simple: If C = 1, then the input A is copied to B. If C = 0, then B is entirely unaffected by A.

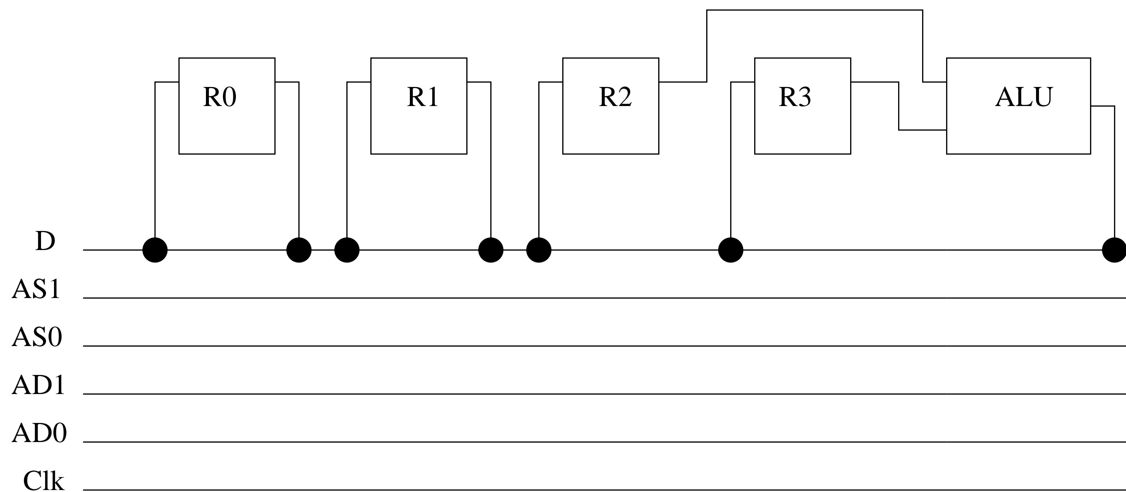
Suppose we wish to copy the contents of R1 to R0. Then we will put 1 on the AS line and 0 on the AD line. What will happen when the clock pulse comes?

Look at the tri-state buffer just below and to the right of R1. Its input ("A") is R1's Q value, i.e. the valued stored in register R1. Since AS = 1, the tri-state buffer will allow its input to flow through, i.e. R1 will be copied to the D line. Meanwhile, R0 will not be copied to the D line, because the input to R0's tri-state buffer is 0. (The little circle to the left of that buffer represents an inverter, i.e. a NOT gate. Since AS = 1, the inverter converts the signal inputted to R0's tri-state buffer to a 0.) Of course, that is what we want; we would have chaos if both R0 and R1 were copied to D at the same time.

You should now verify on your own that because we put 0 on the AD line, when the clock pulses, the value on the D line (which was from R1, as we saw above) will flow into R0. So we will indeed have copied R1 to R0, as desired.

You might be wondering what will control which values go onto AS and AD. Well, recall that a CPU executes machine language instructions. Each instruction is implemented by a portion of the CPU's **control unit**, a combinational circuit which has as its input the op code from the current instruction, and which has AS and AD (among other things) as its output.

Let us add an ALU to the two-register, one-bus system depicted above:



We do not have room to show the connecting lines, gates and tri-state buffers; suffice it to say that they are similar to those shown earlier. (You should try drawing some for yourself, though, as a check of your understanding.) For simplicity, we are continuing to assume a one-bit word size.

The ALU has two inputs on its left, and one output on its right. Again, remember that the ALU is a combinational circuit, for example performing addition, using properly-chosen gates as we have seen before.

Note that we have added two new registers, R2 and R3. They are known as **private registers**, "private" in the sense that they will not be visible to the programmer, i.e. this CPU's assembly/machine language instructions are not capable of specifying using these registers.

Instead, R2 and R3 will serve as temporary storage cells which save data destined for the ALU. As you can see, I've designed their Q outputs to feed into the ALU, rather than being connected back to the D bus line as with R0 and R1.

Now consider a machine instruction ADD R0,R1 for this hypothetical CPU, meaning that the old value of R0+R1 will now become the new value of R0. This instruction would require three clock cycles:

> first clock: copy R0 to R2
>
> second clock: copy R1 to R3
>
> third clock: enable the tri-state buffer connecting the output of the ALU to D, and let D flow into R0

There are also inputs, again not shown, to the ALU determining which operation we want to perform, e.g. add, subtract, logical-and, logical-or, etc. Thus in the third clock cycle the control logic would also be putting the code for "add" on the control inputs to the ALU.[9] During the third cycle we would also enable the connection from the output of the ALU to the bus. Note that this also means that there needs to be circuitry which counts clock pulses.

How could we make this faster? Suppose we had two buses, rather than one, with all components (the registers and the ALU) connected to both buses. Then we could load both R2 and R3 at the same time, i.e. during the same clock cycle: R0 would be copied to R2 via the first bus, while at the same time, R1 would be copied to R2 via the second bus. During the second cycle we would enable the output from the ALU to a bus, say the first one, and enable the input to (in this example) R0. The instruction ADD R0,R1 would then take only two clock cycles, rather than three—a 33% speedup!

And by adding a third bus, we could get the time down to only one cycle. Now all of this is somewhat oversimplified (we have not accounted for time needed to fetch the instruction from memory and decode it, etc.), but you can at least begin to see the principle here—a classic computer science time/space tradeoff. If we are willing to use greater amounts of precious space on the CPU chip (more buses take up more room), we can reap a savings in time.

With this example, the need for edge-triggering or some other anti-feedback mechanism[10] should be more apparent. It would be worthwhile for you now to go back and review that section earlier in this document.

5 Example: Memory Chips and Systems

To illustrate the principles developed here, we will consider the design of simple memory chips and memory systems.

5.1 An SRAM Memory Chip

First consider the design of a "4x2" memory chip, meaning that it contains four two-bit words. (This is much, much smaller than the sizes of typical commercial memory chips, but the principles are the same.) We will call the four words Word 0, Word 1, Word 2 and Word 3. (Remember, though, that these are word numbers within this chip, not within a system constructed from this chip and others; more on this later.)

The chip will have the following pins: address pins A1 and A0 (two pins encode $2^2 = 4$ addresses), which indicate which of the four words is to be accessed; data-in pins DI1 and DI0 (for writing data to the chip); data-out pins DO1 and DO0 (for reading data from the chip); a write-enable pin WE, used to inform the chip that we wish to write to it; an output-enable pin OE, used to inform the chip that we wish to read from it; and a chip-select pin CS, to inform the chip that it, rather than some other memory chip, will be involved in the current memory transaction.

Again, the "4x2" designation for this chip means that the chip contains four words, each two bits wide. Note

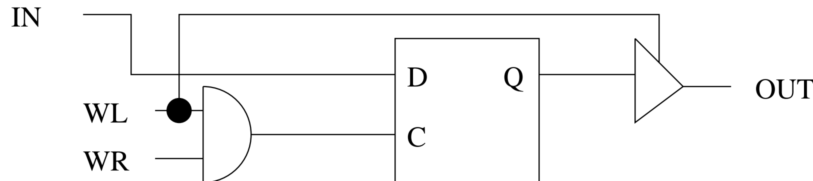
[9]Though I have not set it up this way here, in most designs even a "move" operation is done through the ALU; in the case here, then, during the first and second clock cycles the control logic would put the code for "move" onto the ALU's control input lines.

[10]Such as **master-slave** flip-flops.

that this might be a quite different viewpoint than that held by the CPU of our system. The CPU might, say, view an "8x4" system would consist of eight four-bit words. The memory for such a system could then be constructed by using in combination four 4x2 chips, as we will do later. (The CPU, though, would be unaware of the chip structure of the system.)

The use of separate pins for data input and output here is not standard, but simplifies the design somewhat. If we were to have a single set of pins for both functions (as we will assume later), the design below could be modified in a straightforward way.

To design this chip, let us first design a one-bit cell which will serve as the basic building block for the chip:



Here there is a D flip-flop, which will hold the stored bit, together with three inputs, IN, WL and WR, and one output, OUT.

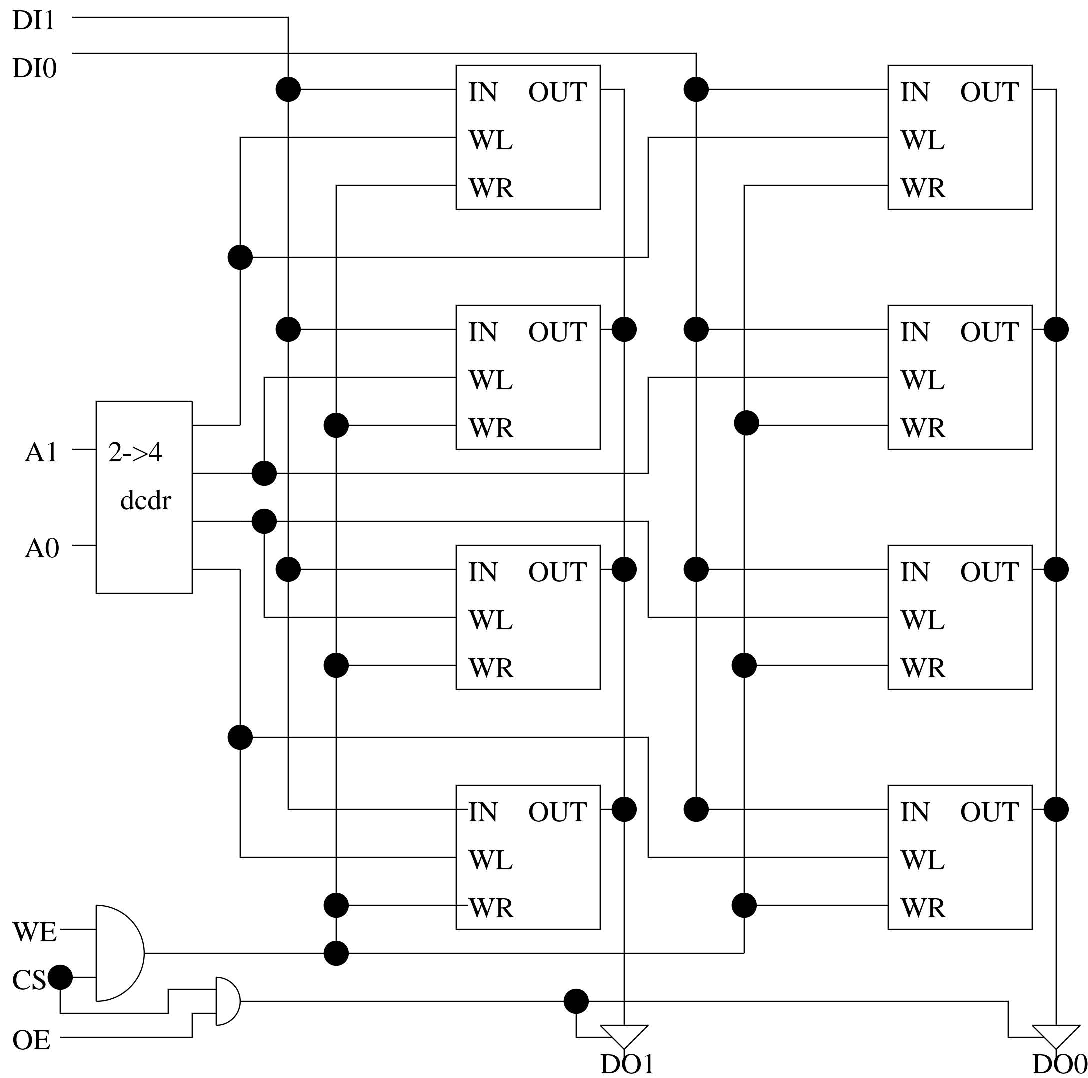
When we write data to this bit, the data will come in on the IN line, and the WR line will be 1. When we read data from the bit, it will come out the OUT line, and WR will be 0.

Recall that the flip-flop is clocked, and responds to pulses (their leading or trailing edges), rather than to levels of a signal. In our case here, it is likely that the WR line will send such pulses, because it will be the result of AND-ing with a clock. Suppose for example that our system bus (which connects the CPU to memory) contains a W line and a CLOCK line. We could AND those two lines together, and feed the result into WR. Then if the CPU asserts the W line, when the clock pulse occurs, that pulse will appear on WR as well, and have leading and trailing edges.

WL ("word line") has the following function. As you will see below, our full memory chip will consist of a 4x2 array of the one-bit cells we are now examining. Each row of that array consists of one word within the memory chip, i.e. one address within the chip. The WL line for the two bits of a given word will go to 1 when we wish to access that word. If WL is 0, then you can see from the diagram above that no data will be allowed into or out of the bit cell; note the role of the tri-state device for the latter case.

By the way, note that if WR is 1, i.e. we are doing a write to this bit cell, data actually *is* allowed out of the cell, which seems wrong. However, this will be remedied by other tri-state devices in the chip as a whole.

Now here is our design for that chip:



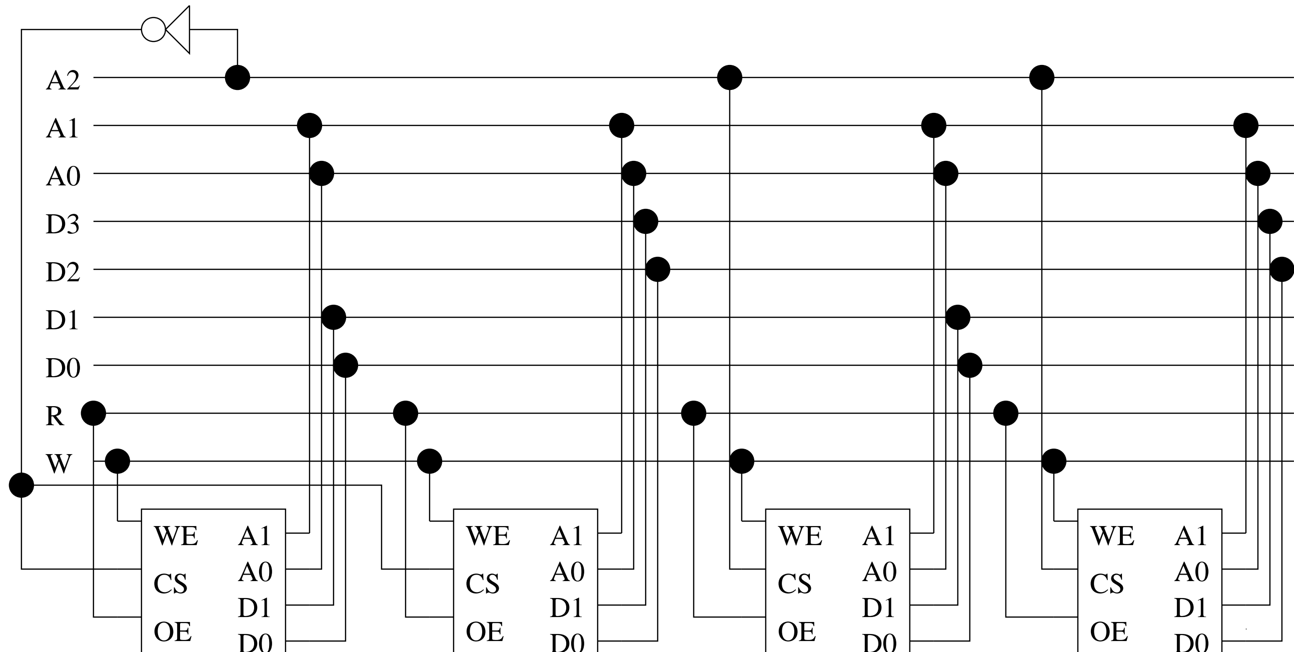
As mentioned earlier, the main part of the chip consists of a 4x2 array of the bit cells designed above. The top row is Word 0 of the chip, the row just below it is Word 1, and so on. A 2-to-4 decoder at the middle left of the diagram selects the proper row of bit cells, i.e. the proper word, depending on which address is desired.

The design is straightforward. Let us confirm, for instance, that if CS is 0, no data will be allowed into or out of this chip. First, note that if CS is 0, then the inputs to WR in the bit cells will also be 0; thus no data will be allowed into any bit cell, exactly as planned. Second, note that if CS is 0, the inputs to the two tri-state devices near the DO1 and DO0 pins will also be 0, insuring that no data flows out of the chip. Again, all of this is important; we will be connecting several of these chips to a system bus, and must insure that we do not have data from two chips flowing onto the same bus lines at the same time.

Consistent with our earlier comment on the 1-bit cell design, the WE input here would likely be set up as the AND-ing together of, say, W and CLOCK lines in the system bus. The same comment applies to our memory system in the following subsystem (even though the CLOCK line is not drawn).

5.2 A Memory System

Now, let us see how a memory system can be constructed from memory chips. Again for simplicity, we will continue to assume 4x2 chips,[11] and we will assume an overall system of eight four-bit words.[12] Here is how we can construct the system from 4x2 chips:



The bus here is a system bus, not the buses internal to CPUs and memory chips which we have discussed earlier. The CPU and I/O devices are also connected to this bus, but are not shown here, nor is the CLOCK line shown. We are assuming that no direct memory-to-memory access is possible; all reads from and writes to memory are performed by the CPU.

Let's call the four chips I, II, III and IV, from left to right. By inspecting which chip pins are connected to bus lines D3-D0, you can see that chips I and III contain the lower two bits of each four-bit word, while chips II and IV contain the higher two bits. By noting the connections of bus line A2 to the CS pins of the chips, you can see that of the entire address space 0-7, **now speaking from the CPU/system viewpoint**, chips I and II contain system words 0-3, and chips III and IV contain system words 4-7.

Suppose, for example, we have a C program with an **int** variable **x**, and that the address of **x** happens to be 3 in this system. Then **x** will be stored in chips I (lower 2 bits of **x**) and II (upper 2 bits of **x**). If the C source code has something like

```
x = 5;
```

then on, say, an Intel CPU the compiler will produce code like

[11]For even more simplicity, we now assume that each data pin does both input and output.

[12]Remember, the CPU will see only the latter, and not know how the system breaks down in terms of chips.

```
movl $5, x
```

In executing this instruction, the CPU will put 011 on lines A2, A1 and A0, and put 0101 on D3-D0, and so on.

5.3 Memory Interleaving

The four-chip system in the above example is said to be **high-order interleaved**, which means that the most significant ("high-order") bits of the address determine which chips (in this case a chip pair) a given address is stored in. With **low-order interleaving**, the least significant bits are used instead. Note that (think about this and make sure you understand it) in a high-order interleaved system consecutive addresses are stored within the same chip (until we reach a chip boundary) while in the low-order case consecutive addresses are stored in consecutive chips (mod the number of chips).

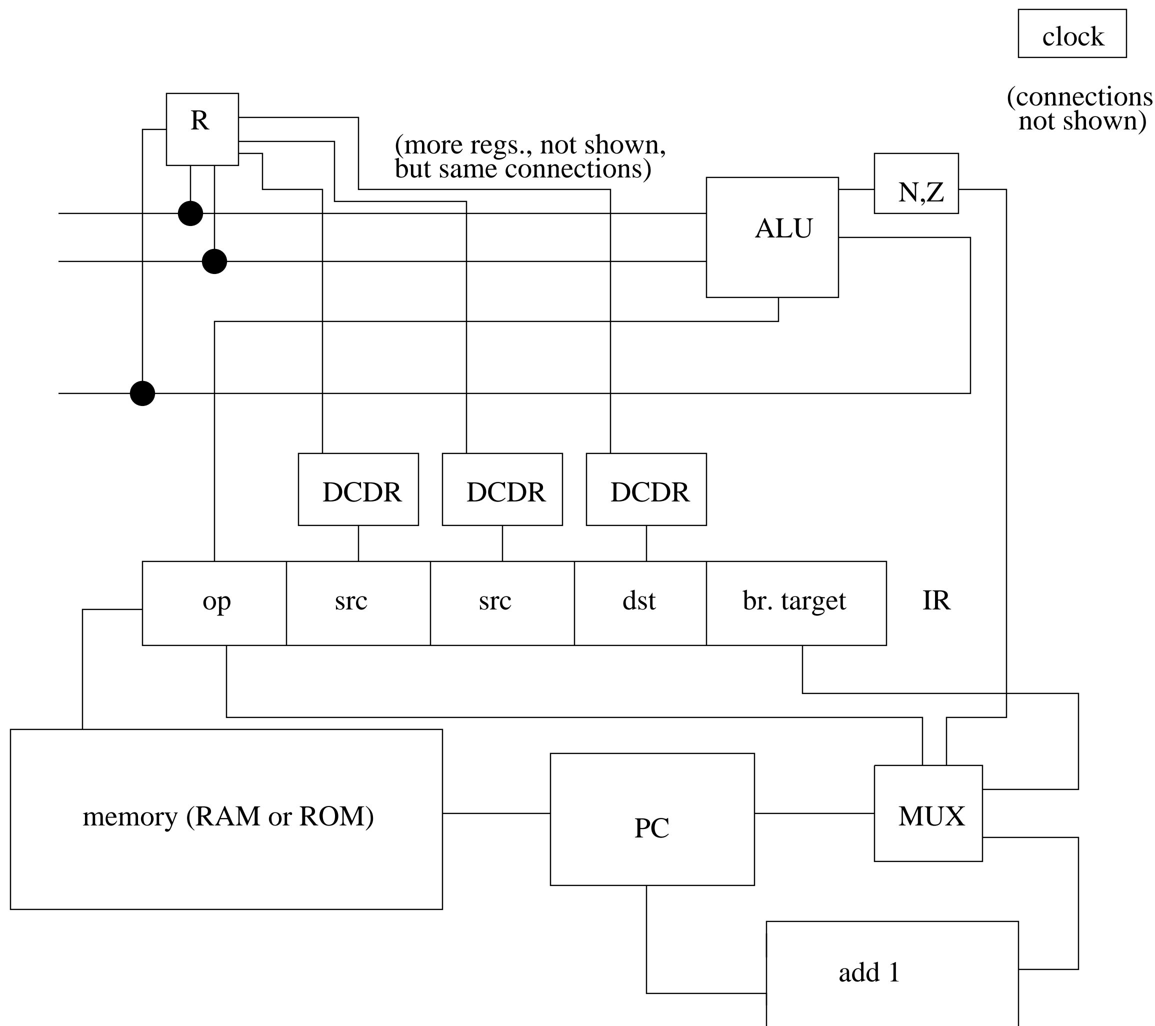
5.4 DRAMs

The kind of memory chip described above, in which each 1-bit cell is implemented as a flip-flop, are known as **static** RAM (SRAM). By contrast, the bit cells in **dynamic** RAM (DRAM) chips consist of tiny capacitors, rather than flip-flops; a charged capacitor represents a 1, while a discharged one represents a 0. The term **static** refers to the fact that each flip-flop in an SRAM continuously maintains itself (as noted when first discussed flip-flops in this tutorial); it will not change as time passes, until we want it to change.

Except for the structure of individual bits, the connections between the bits in a DRAM is very similar to that of an SRAM. One difference, though, arises from the fact that a bit in a DRAM will lose its charge as time passes, so it must be periodically refreshed, i.e. recharged, by additional circuitry. The refresh operation of a DRAM also brings some reduction in system performance, as a bit (or row of bits) is not accessible during the time it is being refreshed. On the other hand, the simplicity of DRAM bit cells means that we can fit more bit cells on a chip, thus saving cost.

6 Example: A Simple CPU

As another illustration of how these principles can be used, we will examine the design of a simple CPU:



We will assume 8 registers. (Only one of them is shown here, but the others all have the same connections as the one shown.) All data accessed by the program will be in the registers, unlike most CPUs, in which data can be either in registers or in memory. The program itself is in memory.

Let's review some terms first: A CPU has a special register, typically called a Program Counter (PC), which specifies the address in memory of the next instruction to be executed. At the beginning of a fetch-execute cycle, the CPU will fetch the instruction from memory which is pointed to by the PC, depositing the instruction in the Instruction Register (IR). There the instruction will be decoded, meaning that the digital logic in the CPU will execute the instruction according to the **op code** and **operands** specified in the instruction. In our case here, the latter are specified in terms of which two registers will be used as sources for the instruction, and which one is to be the destination. Thus each of these register fields will be 3 bits wide. The execution of the instructions is performed mainly by the Arithmetic and Logic Unit (ALU). While execution is being performed, the PC is also being incremented by 1 (except in the case of a branch), to prepare the PC for the next instruction when this one is done.

Referring to the two ALU inputs and its outputs as src1, src2 and dst, respectively, let us set up these op codes:

```
0000      dst = src1
0001      dst = src1 + src2
0010      dst = src1 - src2
0011      dst = src1 AND src2
0100      dst = src1 OR src2
0101      dst = NOT src1
0110      if N go to branch target
0111      if Z go to branch target
1000      go to branch target
```

This design features three buses. (Keep in mind that these are *internal CPU buses*, not a system bus which connects a CPU to memory and I/O devices. Also, note that we have a direct connection here between the CPU and memory, instead of using a bus.) The top two buses lead to the ALU as inputs, while the third bus carries output from the ALU.

Note the box labeled "N,Z" in the figure. This calls for some more review. Recall that CPUs have a register dedicated to **condition codes** (some RISC CPUs allow any register to be used for this), which record certain facts about the output of the last ALU operation: The N ("negative") bit states whether the ALU output was negative, while the Z ("zero") bit records whether that output was 0. In both cases, 1 means yes and 0 means no; for instance, Z = 1 means "Yes, the output was 0."

The condition codes are used for conditional branches. On Intel-CPU machines, for instance, consider the code

```
subl %ebx, %eax
jnz t
```

The first instruction subtracts the EBX register from the EAX register. This will be done by the CPU, and the N and Z bits will record the result, which in turn will be used by the second instruction, JNZ: It says, "If the Z bit is not set, then jump to T."

Now, let's look at the design.[13] The "op" section of IR also feeds into the MUX. The latter must provide the PC either with the incremented previous PC value, or the branch target (T in our example above), depending on (a) the op code (i.e. what kind of branch we are doing, if any) and (b) the values of N and Z.

Study this diagram carefully, and think about how to implement the details.

[13]The diagram does not indicate which lines are inputs and which are outputs. Here is a guide: Observe that in IR, each of the register fields is input to a decoder (3-to-8), the outputs of which control which register is copied to the first bus, which one is copied to the second bus, and which register will be loaded with the ALU output on the third bus. The "op" section of IR is input to the ALU, to control what operation the latter does.

Ins: Left of R; left and bottom of ALU; left of N,Z; bottom of IR; right of memory; left of IR; top and right of MUX; right of add 1.

Outs: Right of ALU; right of N,Z; bottom of R; top of DCDRs top and bottom of IR; left of MUX; top of memory; left of PC; left of add 1.