Use pencil!

Last time

- Introduced basic logic and some terms including "bus", "word", register" and "combinational logic".
- Talked about schematic representation.
- Introduced Verilog.
  - Basics of Verilog including @*, wire, reg, etc.

Let's look at how one could implement "NOT" in Verilog. (And yes it's very wordy)

The top part describes the interface. Basically it provides the name, inputs and outputs.

- 4'b0101 means a 4-bit number (5 in this case).
- 4'h9 means a 4-bit number in hex (base 16) is 9.
- What do you think 1'b0 means?

- How would you write the binary number 10010 in Verilog?

We use "begin" and "end" instead of "{" and "}". In fact you can leave them out in the same situations (though we'll generally include them even when not strictly needed).

```
module NOT(
    input wire A,
    output reg B);

    always @* begin
        case (A)
            1'b0: begin
                B = 1'b1;
            end

            1'b1: begin
                B = 1'b0;
            end
        endcase
    end
endmodule
```

  - Discussed how to connect components.

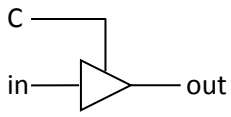
```
module top(
    input wire [3:0] A,
    input wire [3:0] B,
    input wire [3:0] C,
    input wire [3:0] D,
    output wire [3:0] result);

    wire [3:0] mult1_out;
    wire [3:0] mult2_out;

    mult u1 (A, B, mult1_out);
    mult u2 (C, D, mult2_out);
    add u3 (mult1_out, mult2_out, result);

endmodule
```

Review Questions

1. If I have a * and + box, how would I use schematic representation to draw A*B+C*D?
2. What does @* mean?
3. In binary, how would you write 6'd14?
4. Write a "top" module which uses mult and add to find "(A+B)*(C+D)"
5. What are u1, u2, etc?
6. At the end of lecture, I claimed I could reorder the mults and add. Why doesn't order matter here?

Tri-state devices

One other thing that will be useful to us is a "tri-state device". Basically, what this is, is a device that let's us "disconnect" a wire—it's in effect a switch. The standard symbol can be seen to the right. Basically speaking, if C is a "1" then out=in. If C is a "0" then out is connected to nothing at all. The truth table is seen below.



C	in	out
0	0	Not connected
0	1	Not connected
1	0	0
1	1	1

Draw a schematic which uses tristate devices and a NOT to implement the following:

```
if(X=1)
      Y=A
else
      Y=B
```

Administrative break…

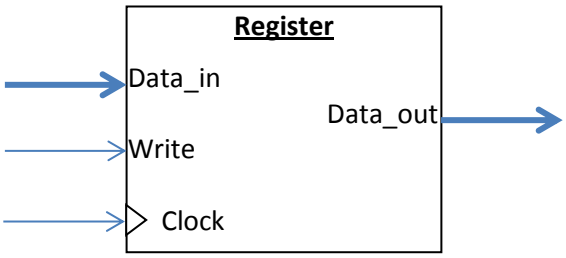
- Your TC2 assignment is due on January 26th. It's a memo about ideas on educational toys.

- Class grade comments
  - You may have noticed that the class is graded out of 1000 points total (230 for the project, 200 for the final, 10 for lab 1, 20 for your TC2 memo, etc.)
  - It is worth noting that there isn't a limit on how many people can get each grade. We will grade straight-scale. So if you are in the 90s you'll get an A of some sort (A+, A, A-). If you are in the 80s, you'll get at least a B of some sort, etc.
    - It's possible we'll be more generous than straight scale, but not less generous.
    - Do recall that you need a C (not a C-) to pass.
- A brief discussion on why TC is important…
- I need to get everyone's pictures I didn't get the first time.
  - Still working on names. I'm slow…

Sequential Logic

So far, we've only used "combinational" logic. To remind you, that means that the output of the device is determined solely by the current inputs. Look at the tri-state truth table above. If you know the inputs, you know the output.

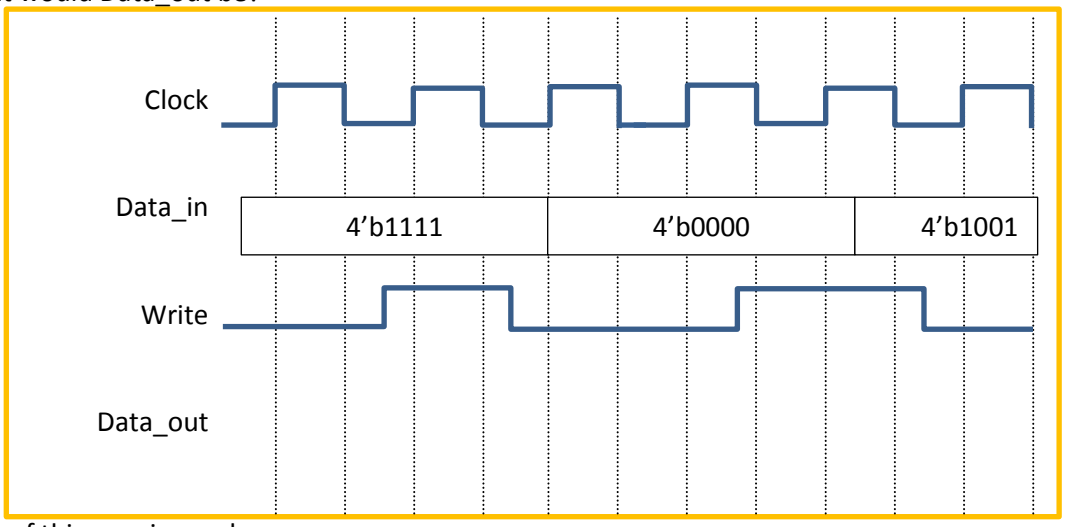
Sequential logic allows for "memory". That is, we can store data and the output of the device depends upon what is being stored. As we discussed before, we generally use the term "register" to indicate a device that stores data.



This device will output (on Data_out) the last value written to it. So how do you write to it? It requires two things to happen.

1. *Write* must be a 1
2. *Clock* must be a "rising edge"

Say that Data for the above register is 4 bits in size (both Data_in and Data_out). Say Data_out starts at 4'b0101. What would Data_out be?

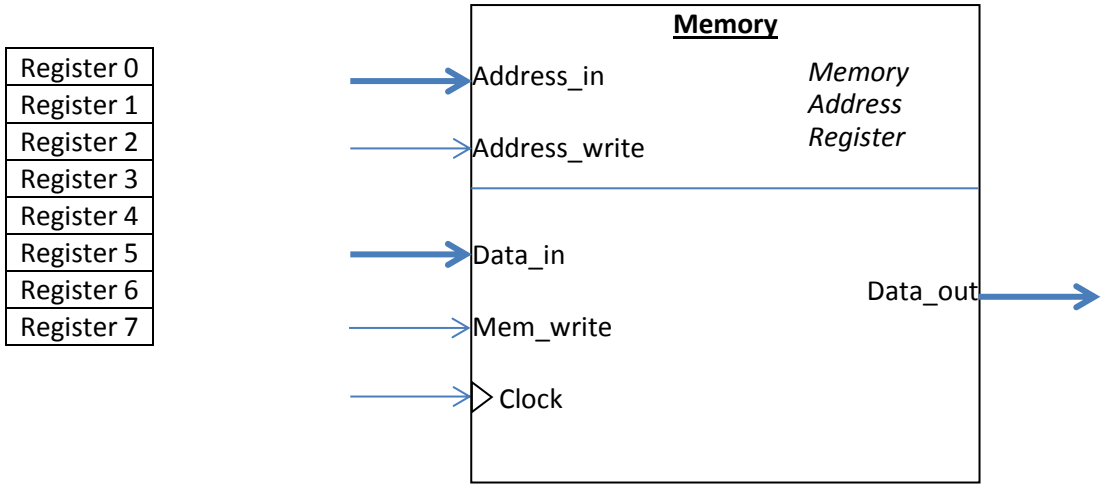


There are a lot of things going on here.

- The above is a "timing diagram". It shows how things change over time. Generally the x-axis would have some amount of time listed (maybe 1ns per dashed line)
- We are seeing a clock for the first time. Generally speaking clocks will be periodic (as shown above) with some frequency. Your home computer probably uses a 2 to 3 GHz clock for the logic in its processor. Generally speaking, registers will only change on the rising edge of clock.
- I'm showing a 4-bit bus here in a fairly typical way. When the data changes, I just show the new value.

Memories

If you think about a register as a variable (which is a fine analogy) then memories are arrays. You can select which register you want (called the address) and that's the one that you can read or change.



What's going on here is that I have two separate devices: the memory address register and the memory itself. If I want to read from register 3, I first have to write a "3" to the memory address register (using Address_in, Address_write, and clock). Until I again change the memory address register, I'll be reading and writing only from register 3.

1. If there are 8 registers in this memory, how many bits do I need for Address_in?
2. If each register were 6 bits wide, how many bits do I need for data_in and data_out?
3. Assuming both of the above are true, how many bits of storage (not including the Memory Address Regsiter) does this memory need?
    - What if I included the memory address register in that calculation?

And…

That's it. Amazingly, that's all the building blocks we'll need to implement any general algorithm—including a computer. Just combinational logic (including tri-states), registers, memories (which is really just an array of registers and a way to select an element) and wires!

(Bonus page) Adding numbers as an example of combinational logic[1]

Below, in figures 1 and 2, we have a LOT of things going on at once. We've designed a circuit which we claim will add two 4-bit numbers. We'll use the notation A[3:0] to indicate the *bus* that is made up of A_3, A_2, A_1, and A_0. We'll use B[3:0], S[3:0] and C[4:0] in a similar way. Let's assume C_0=0, A[3:0]=0100 and B[3:0]=0110. That is we are performing:

```
  0100
+ 0110
=======
```

```
  +
=======
```

What is the base-10 equivalent of the addition on the left?

For the above addition, what are the values of:

- S[3:0]
- C[4:0]

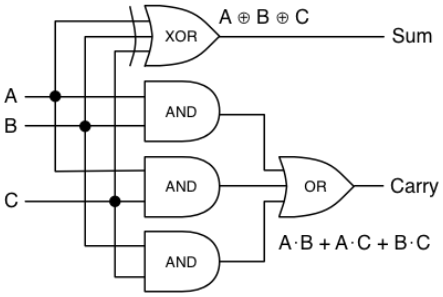


Figure 1: Full adder

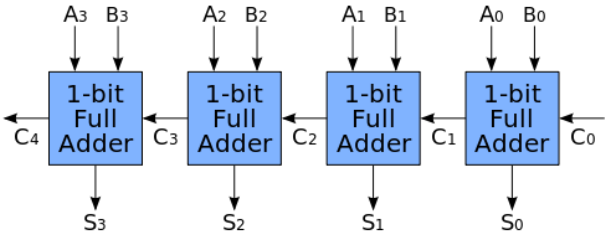


Figure 2: Ripple Carry Adder.
S is Sum and C is Carry!

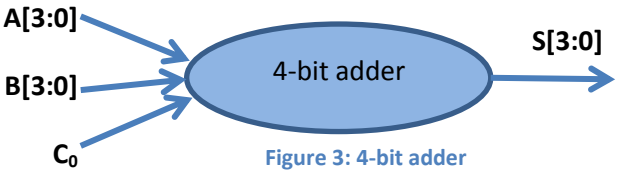


Figure 3: 4-bit adder

Here we are working at a number of levels of abstraction. We've built a 1-bit full adder out of gates and then built a 4-bit adder out of those 1-bit adders. We could just *use* a 4-bit adder and not worry at all about how the adder was built (in fact there are better ways to build an adder!). Instead we just need to know how the device works. In Verilog, we could just use a "+" instead, but sometimes we want to control the implementation and so we might do it this way.

Can you create the full adder and 4-bit ripple carry adder in Verilog as we did the multiply-add circuit above?

[1] Figure for full adder from http://robey.lag.net/2012/11/07/how-to-add-numbers-1.html. Figure for RCA from https://en.wikipedia.org/wiki/Adder_(electronics)