Use pencil!

Last time

In addition to an overview of the class, we discussed a few topics:

- What is engineering?
  - Didn't give a complete definition last time, but touched on a few components. A full one I like can be found in the American Heritage dictionary:
    Someone who applies scientific and mathematical principles to practical ends such as the design, manufacture, test, and operation of efficient and economical structures, machines, processes, and systems.

- And what is computer engineering in particular?
  - *The design and low-level use of computers* is the definition I generally use. We discussed VLSI, computer architecture, embedded systems and system software.

- The fact that engineering generally involves abstraction.
  - Basic idea is that you can think at one level and largely ignore issues at other levels. Probably will be easier to truly understand later in today's lecture!

- Digital logic
  - We went over truth tables, AND, OR, NOT and XOR functions. We discussed how to represent those logical statements both with symbols (e.g. + for OR and * for AND) and gates.

- Number representation in binary
  - I provided some basic coverage of binary numbers and the fact that they can be added using digital logic (gates).
  - Mike went into a lot more detail in lab including using 2's complement numbers and how to create an adder of more than 1 bit.

Logic and number representation review questions:

1. Using 4-bit "unsigned" numbers write 4 and 12 in binary.

2. Using 4-bit two's complement numbers, write 4, -4, and 12 in binary.

3. Draw gates which implement the following equation: $\overline{\bar{A} + B \cdot C}$

4. Create a truth table for: $\overline{\bar{A} + B}$

5. If we were to create a truth table for the equation in #3, how many rows would it have?

6. If we were to make a truth table for a 4-bit adder (adds 2 4-bit numbers) how many rows would it have?

Digital logic and computers

In a modern digital device, everything is represented by "1s" and "0s". Typically we use voltage levels (<1 Volt is a zero, >1 is a one would be one scheme…) to specify the 1s and 0s. As mentioned last time, one could use a number of other things to indicate a 0 or 1 (a light being on or off, how full a bucket is, if a domino[1] is standing or not, etc.)

Digital circuits are about manipulating bits. We've seen the idea of using gates (AND, OR, etc.) to compute values. Three *other* things we can do with bits are:

1. Group bits into an array of bits. If we had a group of 4 bits, we could represent _______ different values. Thinking of them as numbers, we could represent numbers from 0 to 15.

   A "word" is an array of bits of the default size for a given circuit. For example, on most computers a word is generally 32 or 64 bits. In this class, we'll often use 16-bit words. Again, that's just the default size in a given circuit/application.

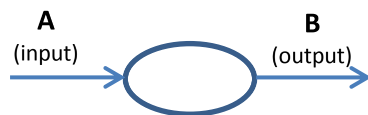
2. We can move a bit value from one place to another with a wire. Nothing too fancy here, just pointing out we *can* do this. In the circuit on the next page, each AND gate's output is connected to the OR gate's input by a wire.

   An array of wires can communicate multiple bits. Such an array of wires is often called a "bus"[2].

3. Store a bit to use later. We call a device that stores one or more bits a "register".

Schematic representation

We can represent a logical function that takes some input and generates some output in a schematic.



We've seen this before with gates (NOT, AND, etc.), but here we are being more generic. We could think of this as a function B=f(A). The box if f(). A is an independent variable and B is the dependent variable.

Unlike a mathematical function, there is some delay in a real system—when A changes B will change a little later. This is called "propagation delay".

This type of box (oval…), is called "combinational logic". The output depends only on the current combination of inputs.

---

[1] http://tinyurl.com/DomLogic

[2] I'll buy a candy bar for the first person who can find *why* a group of wires is called a bus and provide reasonable evidence (post on Piazza). No submissions before 4:30pm today will be accepted (don't start searching now…)

With a truth table we can specify the *behavior* of the combinational logic. We aren't saying how to implement it. We can use a box without knowing how it's implemented. That's what we mean by abstraction—we are working at the level of knowing what the box does, but not caring about how it does it. It's key to note that you have to start at some level of abstraction and leave the implementation of that level to someone else! A carpenter doesn't need to know how or where a tree grew to be able to use a board from the lumber yard.

Hardware description languages—an introduction to Verilog

Say you want to use a specific combinational logic box. So for each input you want a specific output. Say you don't know how to build it, but you have a friend that does know how. How could you tell your friend what you want?

- You could use a truth table and list all inputs and their outputs.
  - But we've seen truth tables can get quite large.
- You can often find an English description. So perhaps B=2*A or S=A+B.

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
```

The always @* notation means that this block of logic should always be computing its value. Put another way, it's how we tell Verilog we want combinational logic.

Misc. notes about Verilog

There are a few other facts that you need to know about Verilog.

Only the value at the end matters.

One very important thing to be aware of is that we are just implementing a truth table. All that matters is that we have the right value at the end. So we can do something like the code on the right. B will be equal to not A by the time we finish the module. And that's all that matters.

```
module NOT(
      input wire A,
      output reg B);
      always @* begin
            B = 1'b0;

            case (A)
                  1'b0: begin
                        B = 1'b1;
                  end
            endcase
      end
endmodule
```

Reg and wire

You may have noticed that some values are called "reg" and some "wire". This is an unfortunate quirk of Verilog. The rule you should use is that if a value is assigned in an always block it should be a reg. Otherwise it should be a wire.

Wow, this feels "wordy"

I mean we could express NOT as a very simple truth table, but in Verilog we are using a lot of text (25 "words" actually). That seems crazy. But we won't generally enumerate all cases like this. After all, that would be huge and tedious (an 8-input module would have 256 cases!). So for now, yes it is very wordy, but hopefully you'll see that its more powerful than a truth table once we start working with more complex modules.

Some more symbols

Verilog supports most of the symbols you'll have seen in C and similar languages. && is AND, || is OR, ! is NOT, etc. Also the operators we use in C for comparison, such as ==, !=. >, >= and the like are also supported. ^ is XOR. We can also do addition (+), subtraction (-), and multiplication (*). Numbers will be treated as unsigned values where it matters.

Group example

Now, given all that, let's write a function that takes more than one input and see how we do.

Write a module that models Jim Harbaugh's raise for next year. Let's say he gets a $100,000 raise if Michigan beats MSU (unlikely as that may be ☺) and a $100,000 if he wins a bowl game. If he does both he gets a $300,000 raise. He goes no raise otherwise. Let's create a module which takes two inputs (MSU, bowl) and returns a 2 bits value (raise). Raise will be either 00, 01, or 11. We can declare a two bit output reg as:

```
output reg [1:0] raise
```

and we can do an assignment of "10" to raise as

```
raise = 2'b10
```

A high-level 4-bit adder

```
module add(
      input wire [3:0] in1,
      input wire [3:0] in2,
      output reg [4:0] out);

      always @* begin
         out[4:0] = in1[3:0] + in2[3:0];
       end
endmodule
```

Consider the module on the right. This is a great example of a module where the Verilog is considerably simpler than the truth table.

As a note, this is ***exactly*** the same as if I did the same thing by using the "truth table" style of Verilog we've been using.

```
if ( in1 == 4'b0000 && in2 == 4'b0000) begin
      out = 5'b00000;
end else if ( in1 == 4'b0000 && in2 == 4'b0001) begin
      out = 5'b001;
```

(etc., etc., etc.).

It's not that we would ever write an add module like that, it's that the module in the box is exactly the same as doing that!

Review of combinational logic terminology

We've covered the basics of combinational logic and combinational Verilog! Let's review a few things quickly.

- Combinational logic
- Register
- Bus
- Propagation delay

Connecting components

Let's say we want to compute A*B + C*D. If we have an add module and a * module, how would we do that? Let's just draw the schematic.

Note that all blocks are working at ***all times***. Unlike a program, where you compute A*B then C*D and then add them, here each device is always doing its task! If you change A, there will be some delay before the output changes (that delay is called what again?).

Let's look at how to build this in Verilog (on the next page)

```
module mult(
      input wire [3:0] in1,
      input wire [3:0] in2,
      output reg [3:0] out);

      always @* begin
            out = in1 * in2;
      end
endmodule

module add(
      input wire [3:0] in1,
      input wire [3:0] in2,
      output reg [3:0] out);

      always @* begin
            out = in1 + in2;
      end
endmodule

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);











endmodule
```

Final thoughts

In our design, would it work if we changed the order we instantiated the modules? Say, did the add first?

Yes it would. Wouldn't work in a programing language, but here we are describing hardware.

(Bonus page) Adding numbers as an example of combinational logic[3]

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]

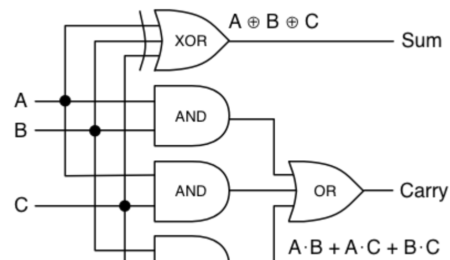


Figure 1: Full adder

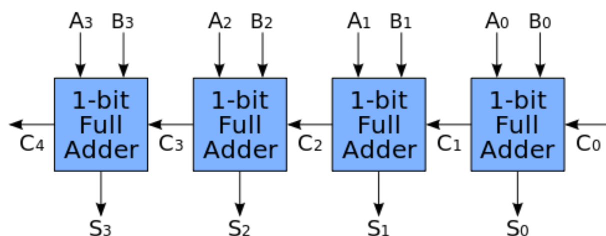


Figure 2: Ripple Carry Adder.
S is Sum and C is Carry!

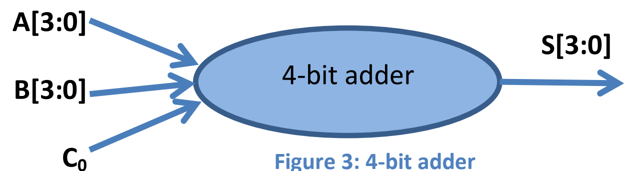


Figure 3: 4-bit adder

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?

[3] 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)