ECE253 Abridged

Aman Bhargava

September 2019

Contents

Chapter 1

Review: Bit Manipulation

Have you ever wanted to be a cool computer person who does things with ones and zero's instead of actual letters and numbers like a normal person? If so, this is the right chapter for you!

1.1 Converting to and from Different Bases

Base 10, 2, and 16 are most commonly used. Base 16 is just a way to read base 2 in a more efficient manner. In order to work with bits it's pretty important to know how to convert back and forth because the test is all on paper.

1.1.1 Converting from base 10 $\rightarrow$ base 2

You keep dividing by two, keeping track of the remainder. Eventually the number you will be trying to divide by two will be 1. You keep going until it's zero + remainder(1). Then you read the reaminders upward from that final 1.

1.1.2 Converting from base 2 $\rightarrow$ base 16

Any hex number can be expressed as 4 binary digits. Make a correspondence table between quadruplets of binary numbers and hex (1-f, inclusive). To convert to base 16 subdivide from right to left in groups of four binary digits. Pad the leftmost part with leading zeros and convert using the table.

1.1.3 Converting from base 10 $\rightarrow$ base 16 (and vice versa)

Just go through base 2 fam.

Chapter 2

Logic Functions and Logic Gates

2.1 Or Gate

1. Symbols
2. Switch structure
3. Truth table

2.2 And Gate

2.3 Inverter

2.4 XOR

2.5 Boolean Algebra

Here are the axioms of Boolean Algebra:

1. $0 \cdot 0 = 0$
2. $1 \cdot 1 = 1$
3. $0 \cdot 1 = 1 \cdot 0 = 0$
4. if $x = 0$, $!x = 1$

Dual Form

1. $1 + 1 = 1$
2. $0 + 0 = 0$
3. $1 + 0 = 0 + 1 = 1$
4. if $x = 1$, $!x = 0$

Duality: In a given logic expression, you can swap $1 \to 0$ and $\cdot \to +$ and the expression is still valid.

2.6 Useful Boolean Expression Rules

- $x \cdot 0 = 0$
- $x \cdot 1 = x$
- $x \cdot x = x$
- $x \cdot !x = 0$
- $x \cdot 0 = 0$
- $!!x = x$
- $x + 1 = 1$
- $x + 0 = x$
- $x + x = x$
- $x + !x = 1$

Distributive Properties:

$$x \cdot (y + z) = xy + xz$$

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

2.7 Sum-of-Products (SOP)

'Sum' means boolean OR while 'product' means boolean AND.

Min term: for n variables, term where all variables appear once is a 'minterm'. Note that variables can either be complimented or uncomplimented.

Any boolean function can be represented by the sum of products of minterms - this is just done by simply converting the truth table and OR-ing each truth.

Then, you can simplify. Pretty common sense. 'Canonical' just means it's a bunch of midterms separated by OR's.

2.8 Product of Sums (POS)

Max term: where all n variables appear OR-d. They can be complimented or un-complimented.

You can pretty easily make all the valid max terms from a truth table. To generate a POS expression, you multiply (AND) the maxterms that sum to zero.

2.9 NAND and NOR Logic Networks (TB 2.7)

2.10 Three-Way Light Control (TB 2.8.1)

2.11 How 2 Verilog

A 3-input multiplexer can be made with two 2-input multiplexers. Now let's implement this without using two pre-made multiplexers. Let:

-

Notes on Implementation

- If you want inputs to be registered from switches, you need to assign them SW[¡int¿]
- Likewise, if you want outputs to be registered to LED's, assign them LEDR[¡int¿]

2.11.1 Code: 3-Way Multiplexer

```
module mux2b2to1 (SW, LEDR); //Two bit 2 to 1 multiplexer
   input[4:0] SW, //[4:0] sets switches 0-4 to inputs(?)
   output[1:0] LEDR, //

   wire S;
   wire[1:0] a, b, z; //'two bit wide vector'?

   assign a = SW[1:0],
   assign b = SW[3:2],
   assign s = SW[4],
   assign LEDR = z,

   assign z[0] = (~s&a[0]) | (s&b[0]);
   assign z[1] = (~s & a[1]) | (s&b[1]);

endmodule;
```

Can you make the assignment more efficient? What if you do this:

```
   // assign z[0] = (~s&a[0]) | (s&b[0]);
   // assign z[1] = (~s & a[1]) | (s&b[1]);
   assign z = (~s&a) | (s&b); // NOT CORRECT
```

Because s is only 1-bit and a, b are two bits, s is extended with a 0, which makes the logic incorrect!

General Notes on how Syntax Works:

- **What is 'assign'?** 'assign' just means you're making a connection (alias?) b/w the two. **Question:** is this necessary for instantiating the variable?
- **Assignment arith.** When you create a verilog wire/input with $x = [a : b]$, the number of bits in x is $b - a + 1$
- **Bit Access:** To access bits, you say $x[n]$ or $x[n : m]$ where $n < m$. The length of the slice is $n - m + 1$
- **Concatenation:** If you want to stitch together multiple bits, you use $x = SW[9 : 8], SW[1 : 0]$. That statement stitches together two 2-bit chunks ($SW[9 : 8], SW[1 : 0]$ to make one 4-bit chunk).

- **Order of Operations:** Basically nobody knows... just use parenthesis when you're not sure. And goes before or, though.

2.12 Full Adder

Description: Adding in binary is the same as in decimal, just with fewer options. When adding two single bits, we have three output possibilities: $00, 01, 10$. We call the least significant bit the sum s and the most significant digit the carry c.

Let's see the truth table:

x	y	C	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

As you can see, the sum bit is just $x \oplus y$ and the carry bit is xy. This is a **half adder** because it doesn't accept a carry from the last calculation.

A full adder accepts a carry in C_{in}, x, and y, and outputs sum s and carry out C_{out}.

Let's see the truth table:

x	y	C_{in}	C_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Therefore

$$C_{out} = xy + xC_{in} + yC_{in}$$

and

$$s = x \oplus y \oplus C_{in}$$

Ripple Carry Adder: Then you can string them together by assigning one of these adders to each bit of the output and passing the carry out to the carry in of the next bit!

Specifics for Ripple Carry Adder:

- Max size for output is one more then the two input's sizes because one more order of magnitude in binary is just doubling the number, and the

most you can do when adding two numbers of equal length is double them.

- Basically just string them together and you're golden.

Review of Outcomes

- $Cout = xy + CinX + CinY$
- $Sum = x \oplus y \oplus z$

Now let's make it in verilog!

```
module fulladder(x, y, Cin, S, Cout);
   input x, y, Cin;
   output S, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x&y) | (Cin&x) | (Cin&y); // Why no wires here?
      Bc no physical IO's.
```

Now we make a 3-bit adder out of full adders in Verilog!

```
module adder3bit(X, Y, S)
   input[2:0] X, Y;
   output[3:0] S;
   wire[3:0] C; // to connect full adders together

   fulladder U0(X[0], Y[0], C[0], C[1]);
   fulladder U1(X[1], Y[1], C[1], C[2]);
   fulladder U2(X[2], Y[2], C[2], C[3]);

   assign S[3] = C[3]; // Final carry bit is the most significant
      bit of the sum.
   assign C[0] = 1'b0; // Weird syntax: 1 bit, equal to 0.
endmodule;
```

This is structural verilog - we can't immediately see the bigger picture. We see wires and modules stitched together and we have to figure out what it all means.

Weird constant syntax:

- $1'b0$: 1 bit constant, in binary, equal to 0.
- $4'hF$: 4 bit constant, in hex, equal to F.
- $4'd9$: 4 bit constant, in decimal, equal to 9.
- $8'h1E$: 8 bit constant, in hex, equal to 1E.