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

Chapter 3

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 $1 \rightarrow 0$ and $\cdot \rightarrow +$ and the expression is still valid.

3.1 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:

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

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

3.2 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.

3.3 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.

3.4 NAND and NOR Logic Networks (TB 2.7)

3.5 Three-Way Light Control (TB 2.8.1)

Chapter 4

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¿]

4.0.1 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.

4.1 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.

4.2 7-Segment Display

Prompt: Design a c ircuit with two inputs x_1 and x_0 representing a 2-bit number x. Show x on the 7-segment display (ranges from 0-3 inclusive).

Numbering: h_0 is the top segment. Clockwise increases. h_6 is the middle one.

Note on D1-SoC Board: Logic 0 makes the light turn on for h_{0-6} and 1 makes it turn off ('active low')

4.2.1 Displaying Numbers

Sections to light up for 0: h_{0-5} Sections to light up for 1: h_{1-2} Sections to light up for 2: h_{0-1}, h_6, h_{3-4} Sections to light up for 3: h_{0-3}, h_6

With active low: Sections to power for 0: h_6 Sections to power for 1: h_0, h_{3-6} Sections to power for 2: . Sections to power for 3:

Truth Table:

x_1	x_2	h_6	h_5	h_4	h_3	h_2	h_1	h_0
0	0	1	0	0	0	0	0	0
0	1	1	1	1	1	0	0	1
1	0	0	1	0	0	1	0	0
1	1	0	1	1	0	0	0	0

Consolidating the Logic Functions:

$$h_0 = (!x_1) \cdot x_0$$

$$h_1 = 0$$

$$h_2 = x_1.!x_0$$

$$h_3 = !x_1 \cdot x_0$$

$$h_4 = x_0$$

$$h_5 = x_1|x_0$$

$$h_6 = !x_1$$

Verilog Code:

```
module seg7(SW, HEX0)
   input[1:0] X;
   output[6:0] HEX0;

   assign HEX0[0] = ~SW[1] & SW[0];
   // etc.

endmodule
```

Implementation Example: Designa circuit with 4 inputs a, b, c, d and s. if s $= 0$, show $a + b$ on 7-seg display if s $= 1$, show $c + d$ on 7-seg display

$$a, b \rightarrow mux_{2bit*2inputs} \rightarrow fulladd_{cin=0} \rightarrow HEXO \rightarrow output$$

- The carry in digit to the full adder is 0
- The carry out from the full adder is the most significant digit of the HEXO input.
- Honestly you could just use a half adder because you don't need a carry-in. $C_{out} = xy$, $Sum = x \oplus y$

Cool XOR Fact: $\oplus$ returns 1 if the number of true inputs is ODD.

4.3 FPGA's

Can do anything.

- Field programmable gate arrays: Programmable in the field (i.e. not in the factory necessarily).
- 2-D array of programmable blocks.
- Blocks contain lookup tables (LUT)
- Any two input LUT can be programmed for any two-input logic functions.
- More complex logic comes out when you connect the LUTS - this is what Quartus does.

How to encode any two-input thing Let there be 4 sRAM bits p, q, r, s. Let the inputs be a, b.

$$p, q \rightarrow mux_{a1} \rightarrow x$$

$$r, s \rightarrow mux_{a1} \rightarrow y$$

Where x, y are itermediate variables.

$$x, y \rightarrow mux_b \rightarrow output$$

Therefore it takes 4 ram cells for a 2-input LUT.

$$n_{ramcells} = 2^{inputs}$$

There are 85,000 6-LUTS on lab chips.

Chapter 5

Karnaugh Maps

5.1 Motivation

Creating boolean functions to get what you need done is confusing and unlikely to lead to an optimal solution when there are many variables.

Karnaugh maps are a tool for expressing your truth tables in such a way that makes getting optimal solutions easier with many variables.

Example 2x2 Karnaugh Map

	0	1
0	a	b
1	c	d

5.2 Review of Terminology

- **Implicant:** Any product term for which the function is true (think 'implies' logic is 1)
- **Cover:** A set of implicants that covers all 1's of the function.
- **Prime implicant:** Any implicant that, if we delete a literal, is no longer an implicant (largest groups of ones in the K-map)
- **Essential prime implicant:** Prime implicant that covers a 1 covered by no other prime implicant.
- **Min-cost cover:**
- **Cost:**

Example:

	00	01	11	10
0	1	1	1	1
1	1	0	0	0

Let the left column be the z column and the top be products of x, y.
List of implicants:

1. $\bar{z}$
2. $\bar{x}\bar{y}$
3. $\bar{x}\bar{z}$
4. $y\bar{z}$
5. $x\bar{z}$
6. $\bar{y}\bar{z}$
7. $m_{0,1,2,4,5}$

Prime Implicants: Largest group of ones in cardinal maps...

Cost Defininition: $n_{gates} + n_{inputs}$

5.3 Procedure for Minimum Cost Cover

- Find essential prime implicants and include in cover
- Select additional prime implicants to include in the cover until all 1's are covered.

5.4 5 Variable Karnaugh Map

Use two 4-variable K-Maps. One for case where $x_5 = 0$, other for $x_5 = 1$. Min-term indexing goes by the binary representation of x_{0-n} where x_n is the LEAST significant bit.