

# ECE253 Abridged

Aman Bhargava

September 2019

# Contents

<b>1</b>	<b>Review: Bit Manipulation</b>	<b>3</b>
1.1	Converting to and from Different Bases . . . . .	3
1.1.1	Converting from base 10 $\rightarrow$ base 2 . . . . .	3
1.1.2	Converting from base 2 $\rightarrow$ base 16 . . . . .	3
1.1.3	Converting from base 10 $\rightarrow$ base 16 (and vice versa) . .	4
<b>2</b>	<b>Logic Functions and Logic Gates</b>	<b>5</b>
2.1	Or Gate . . . . .	5
2.2	And Gate . . . . .	5
2.3	Inverter . . . . .	5
2.4	XOR . . . . .	5
<b>3</b>	<b>Boolean Algebra</b>	<b>6</b>
3.1	Useful Boolean Expression Rules . . . . .	6
3.2	Sum-of-Products (SOP) . . . . .	7
3.3	Product of Sums (POS) . . . . .	7
3.4	NAND and NOR Logic Networks (TB 2.7) . . . . .	8
3.5	Three-Way Light Control (TB 2.8.1) . . . . .	8
<b>4</b>	<b>How 2 Verilog</b>	<b>9</b>
4.0.1	Code: 3-Way Multiplexer . . . . .	9
4.1	Full Adder . . . . .	10
4.2	7-Segment Display . . . . .	13
4.2.1	Displaying Numbers . . . . .	13
4.3	FPGA's . . . . .	14
<b>5</b>	<b>Karnaugh Maps</b>	<b>16</b>
5.1	Motivation . . . . .	16
5.2	Review of Terminology . . . . .	16
5.3	Procedure for Minimum Cost Cover . . . . .	17
5.4	5 Variable Karnaugh Map . . . . .	17

<b>6</b>	<b>Storage Elements</b>	<b>18</b>
6.1	Introduction . . . . .	18
6.2	RS Latches . . . . .	18
6.2.1	Behavior . . . . .	18
6.3	Gated RS Latch . . . . .	19
6.3.1	Synchronous Reset . . . . .	19
6.3.2	Behavior . . . . .	19
6.4	Gated D Latch . . . . .	19
6.4.1	Behavior: . . . . .	19
6.5	D Flip-Flops . . . . .	19
6.5.1	Master-Slave Flip Flop . . . . .	20
6.5.2	Behavior . . . . .	20
6.5.3	Why it's Useful . . . . .	20
6.6	T Flip-Flop . . . . .	20
6.7	Flip-Flop Reset/Preset . . . . .	21
6.8	Summary of Objects . . . . .	21
6.9	Verilog Implementations . . . . .	21
6.9.1	Gated D-Latch . . . . .	21
6.9.2	Edge-Triggered Flip Flop . . . . .	21
6.9.3	Synchronous Reset (Active Low) . . . . .	22
<b>7</b>	<b>Finite State Machine</b>	<b>23</b>
7.1	Review . . . . .	23
7.1.1	State Diagrams . . . . .	23
7.1.2	Lmao what is an always block actually . . . . .	23
7.1.3	Case Statements . . . . .	24
7.1.4	State Machine Diagrams and Sequential Diagrams . . . . .	24
7.1.5	Case Statements for State Machines . . . . .	24

# 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 remainders 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[jinti]`
- Likewise, if you want outputs to be registered to LED's, assign them `LEDR[jinti]`

### 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  $c$ .

	$x$	$y$	$C$	$s$
	0	0	0	0
Let's see the truth table:	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  $C_{out}$ .

	$x$	$y$	$C_{in}$	$C_{out}$	$s$
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
Let's see the truth table:	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 than 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

- $C_{out} = xy + C_{in}X + C_{in}Y$
- $Sum = x \oplus y \oplus z$

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 circuit 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:

	$x_1$	$x_2$	$h_6$	$h_5$	$h_4$	$h_3$	$h_2$	$h_1$	$h_0$
	0	0	1	0	0	0	0	0	0
<b>Truth Table:</b>	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:**

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

$$h_1 = 0$$

$$h_2 = x_1 \cdot !x_0$$

$$h_3 = !x_1 \cdot x_0$$

$$h_4 = x_0$$

$$h_5 = x_1 | x_0$$

$$h_6 = !x_1$$

## 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:** Design a 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 intermediate 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 Definition:**  $n_{gates} + n_{inputs}$

## 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.

# Chapter 6

## Storage Elements

### 6.1 Introduction

Until now, the values coming in have depended on the state (high or low) of the input wires to the circuit. There's another option where the inputs depend on past states of the circuit.

**Storage elements** represent a **state** of a circuit. When inputs change, new inputs either leave circuit in the same state or change the state. These circuits are called **sequential** circuits. Memory elements can be created with logic gates.

### 6.2 RS Latches

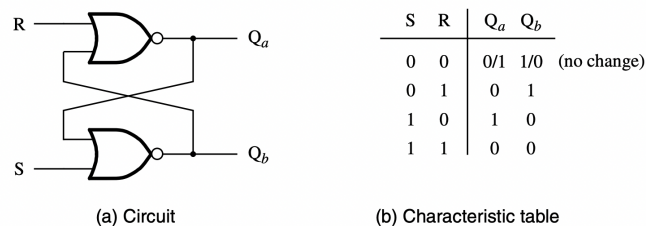


Figure 6.1: Basic Latch

#### 6.2.1 Behavior

You use  $S$  to set the output  $Q_a$  to 1 and  $R$  to set  $Q_a$  to 0.

$S$	$R$	$Q_a$	$Q_b$
0	0	No change	
0	1	0	1
1	0	1	0
1	1	0	0

$S$  functions as the 'set' signal.

## 6.3 Gated RS Latch

Add an and gate to the inputs so that you can only enter signals to the latch when the  $clk$  signal is 1.

### 6.3.1 Synchronous Reset

This is when you can only use the reset when the clock is at 1 (basically you just AND the  $D$  and the  $R_n$  for an 'active low' reset). Reset behavior is synchronized to the rising edge.

### 6.3.2 Behavior

Same as an RS latch, but you can only send  $R$  and  $S$  signals when the  $clk$  is logic 1.

## 6.4 Gated D Latch

Also called 'Transparent Latch' or 'Level-Sensitive Latch'.

### 6.4.1 Behavior:

When the clock is 1, then  $Q = D$ .  $Q$  changes as  $D$  changes and then when the clock is turned back to 0, then  $Q$  persists in it's last state.

## 6.5 D Flip-Flops

**Symbol:** Same as the D Latch, but there's a triangle instead of  $clk$ . If it's a **negative edge** triggered flip-flop, the symbol has a bubble next to the clock. **Positive edge** appears to be the default.

**Edge-Triggered Flip Flop:** The flip flop only changes when the clock is on a rising edge or a falling edge.

### 6.5.1 Master-Slave Flip Flop

You connect a master D-Latch to a slave D-Latch. The clock signals of one is the inverse of the clock symbol going into the other one.

This setup yields a system where you can only change the value of  $Q$  for the slave ( $Q_s$ ) on a rising/falling edge.

### 6.5.2 Behavior

On rising or falling clock edge, the value of  $D$  is stored and appears in  $Q$ .

**Work-Through demonstration:** Let's say you have the inverted clock going to the slave. When you're in logic 1 for the clock, the master is in **transparent** mode.  $Q_m = D$ . When you switch, the value of the master is stuck and the value of the slave is set to  $Q_m$ . You can't reset the slave value after the transition because the **opaque** mode of the master.

Pretty cool property!

### 6.5.3 Why it's Useful

You can put these in the middle of circuits to save values. Then, you can re-inject inputs while you wait for the last half of the circuit to finish processing. It's like splitting the circuit into two pieces that can run on different elements simultaneously.

This is a bit of a dumb way to think about it, but that's the general idea behind how they are actually used.

## 6.6 T Flip-Flop

Let's take a D Flip Flop component and loop the value of  $Q$  and  $\bar{Q}$  back to the input  $D$ . Let's make  $T$  the new input.  $T$  switches the input of  $D$  between  $Q$  and  $\bar{Q}$ .

## 6.7 Flip-Flop Reset/Preset

## 6.8 Summary of Objects

## 6.9 Verilog Implementations

### 6.9.1 Gated D-Latch

---

```
module D_Latch (D, clock, Q, Qb);
    input D, clock;
    output reg Q, Qb; // 'reg' means it's an always block

    always@(D,clock) //defining a block that's sensitive to changes
        in Q and D.
    begin
        if(clock==1'b1)
            begin // need this begin because you're doing two things in
                the if block.
                    Q = D;
                    Qb = ~D;
            end // begin and end function like curly braces in C/C++/Java
    end //if clock == 0'b1, then verilog maintains previous values.

endmodule
```

---

### 6.9.2 Edge-Triggered Flip Flop

---

```
module D_ft(D, clock, Q, Qb)
    input D, clock;
    output reg Q, Qb;

    // Always blocks tell you what you need to
    always@(posedge clock) // This means you execute this on the
        positive edge (posedge is a keyword for that)
    begin
        Q <= D; // use '<=' for describing flip flops.
        Qb <= ~D; // we may get to why in this course but for now
            it's just a thing...
    end
endmodule
```

---

### 6.9.3 Synchronous Reset (Active Low)

---

```
always@(posedge clock)
begin
    if(resetn == 1'b0)
    begin
        Q <= 1'b0;
        Qb <= 1'b1;
    end
    else
    begin
        Q <= D;
        Qb <= ~D;
    end
end

endmodule;
```

---

# Chapter 7

## Finite State Machine

### 7.1 Review

A finite state machine (FSM) is an object that changes its properties (outputs) based on inputs over time. For example, a human could be modeled as a state machine where the input is water over time and the output is the need to go to the bathroom. **Sequential circuits** are FSM's.

#### 7.1.1 State Diagrams

**Example:**  $w$  can be 0, 1. We are trying to find when  $w$  has gone  $1 \rightarrow 0 \rightarrow 1$ .

**Solution:** We can use a shift register to get all the sequential bits of  $w$  in, then we use a simple logical expression on the outputs  $w_0\dot{w}_1\dot{w}_2$ .

**One-Hot Encoding:** When only one of your bits is one. **one cold** is the opposite.

#### 7.1.2 Lmao what is an always block actually

---

```
always@(A, B, C)
begin

end
```

---



$A, B, C$  are in the sensitivity list, the code inside is ‘sensitive’ to them so as they change the internal block is fired. You can only use **if** statements in here.

### 7.1.3 Case Statements

Same as in pretty much all other programming languages.

---

```
case(A)
  value1:
    do_something;
  value2:
    do_something_else;
  value3: // multiple lines
    begin
      do_something;
      do_something more;
    end
  default:
    default_behavior;
endcase
```

---

### 7.1.4 State Machine Diagrams and Sequential Diagrams

### 7.1.5 Case Statements for State Machines

Quick example:

---

```
module FSM(input w, clock, resetn, output z);
  reg[1:0] y, Y;
  parameter A = 2'b0, B = 2'b01, C = 2'b10, D = 2'b11;
  //state table
  always@(w, y) begin
    case(y) begin
      A: begin
        if(W)
          Y = B;
        else
          Y = A;
        end
      B: begin
        if(w)
          Y = B;
        end
    end
  end
```

```

        else
            Y = C;
        end
    endcase
end

always@(posedge clk) begin
    if(!resetn)
        y <= A;
    else
        y <= Y;
    end

    assign z = (y == D);
endmodule

```

---

- *parameter* lets you define constants.