



Under The Hood

Programming Club

SnT Summer Project

MidTerm Evaluation Report

20 June, 2022

Contents

1	PHASE I - BOOLEAN ALGEBRA	3
1.1	BINARY REPRESENTATION OF NUMBERS	3
1.1.1	Unsigned Integers	3
1.1.2	Signed Integers	3
1.2	BOOLEAN OPERATORS	3
1.2.1	Basic Boolean operators	3
1.2.2	Useful Properties of Boolean Operators	4
1.3	TRUTH TABLE	4
1.3.1	Truth Table Diagram	4
1.4	LOGIC GATES	5
1.4.1	AND Gate	5
1.4.2	OR Gate	5
1.4.3	NOT Gate	6
1.4.4	NAND Gate	6
1.4.5	NOR Gate	6
1.4.6	XOR Gate	7
1.4.7	XNOR Gate	7
1.4.8	Universal Gates	7
1.5	FUNCTIONS	7
1.6	K-MAPS	8
1.6.1	Simplifying K-Maps	8
1.6.2	Examples	8
1.7	ELECTRONIC DEVICES	9
1.7.1	Adder	9
1.7.2	Subtractor	10
1.7.3	MUX	10
1.7.4	Comparator	11
1.7.5	Clock	11
1.7.6	Memory	12
2	PHASE II - VERILOG PROGRAMMING	13
2.1	BASICS	13
2.1.1	Intro	13
2.1.2	Variable types	13
2.1.3	Timescale	13
2.2	VERILOG LANGUAGE FEATURES	13
2.2.1	Modules	13
2.2.2	Initial Block	14
2.2.3	Always Block	14
2.2.4	If-else Block	15
2.3	TESTING A VERILOG PROGRAM	15
2.3.1	Creating a Test bench	15
2.3.2	Running the simulation	16
2.4	FINITE STATE MACHINE(FSM)	17

2.4.1	Principle	17
2.4.2	An example: Elevator Control	17
3	MENTORS MENTEES	19

1 PHASE I - BOOLEAN ALGEBRA

1.1 BINARY REPRESENTATION OF NUMBERS

1.1.1 Unsigned Integers

In binary representation, numbers are stored in bits where each bit can be either 0 or 1. Let X be any number and $a_0, a_1, a_2, \dots, a_{n-1}$ denote the n digits of an n -bit binary representation of X where a_0 is the LSB and a_{n-1} is the MSB, then decimal representation of X is given by:

$$X = a_0 2^0 + a_1 2^1 + a_2 2^2 + \dots + a_{n-1} 2^{n-1}$$

Note: An n -bit unsigned binary number can store 2^n values, from 0 to $2^n - 1$.

1.1.2 Signed Integers

In signed integers, 1-bit has to be reserved to store the sign. There are two major ways of storing signed integers in binary.

1's complement: Flip all the bits of a number to get the negative of that number.

Major Drawbacks with this method:

1. Zero has 2 different representations.
2. Only $2^n - 1$ numbers can be stored.

2's complement: Negative of a number is equal to its 1's complement + 1.

An easy way to find 2's complement of a number is that, starting from the LSB copy the bits until the first 1 is encountered, and flip every bit after that, The number we get is the 2's complement of the given number.

1.2 BOOLEAN OPERATORS

1.2.1 Basic Boolean operators

Boolean algebra mainly consists of 3 Boolean operators, which are AND, OR and NOT. Using these three operators, other operators like NAND, NOR, XOR and XNOR can be constructed.

1. AND operator: It is denoted by (\cdot) in boolean algebra. It takes two operands, if both the operands of AND are 1 then it gives 1 as output, otherwise it gives output 0.
2. OR operator: It is denoted by $(+)$ in boolean algebra. It takes two operands, if either or both the operands of OR are 1 then it gives 1 as output, otherwise, if both the operands are 0 then the output is 0.
3. NOT operator: It is denoted by $(A'$ or $\bar{A})$ in boolean algebra. It acts on only one operand and flips its value.

The XOR Operator is denoted by $A \oplus B$ and can be constructed from the above operators as $A \oplus B = A \cdot \bar{B} + \bar{A} \cdot B$ XOR operator has a unique property while checking equality that is $A \oplus B = 0 \iff A = B$

1.2.2 Useful Properties of Boolean Operators

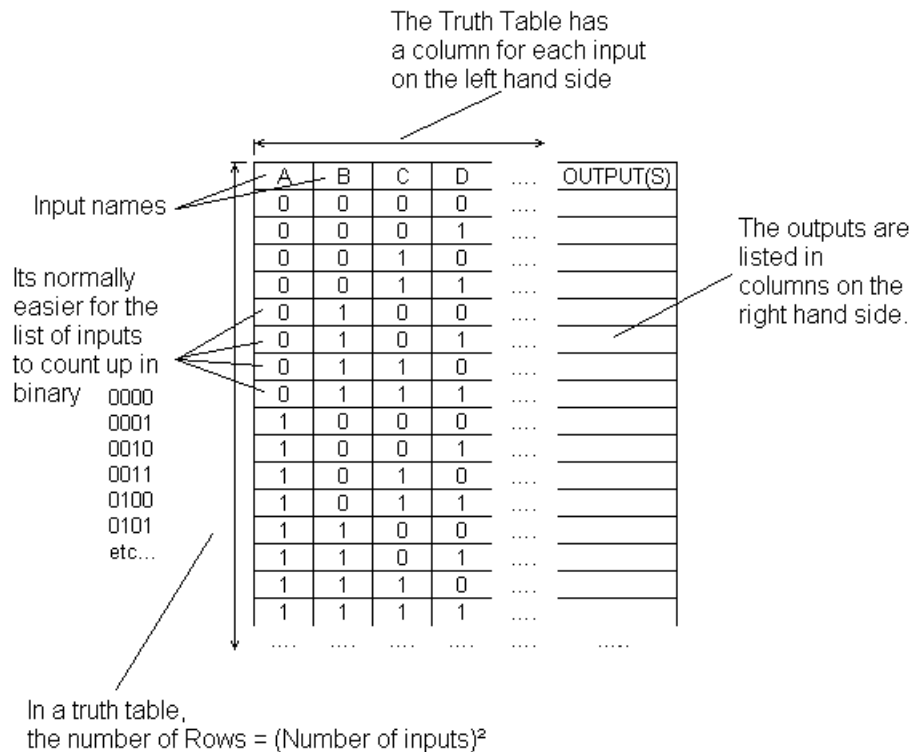
$$\begin{array}{lll}
 A.1 = A & A.0 = 0 & A + 1 = 1 \\
 A + 0 = A & A.\bar{A} = 0 & A + \bar{A} = 1 \\
 A.(B + C) = A.B + A.C & \overline{A + B} = \bar{A}.\bar{B} & \overline{A.B} = \bar{A} + \bar{B}
 \end{array}$$

1.3 TRUTH TABLE

Truth tables are used to help show the function of a logic gate. Truth tables help understand the behaviour of logic gates.

- They show how the input of a logic gate relate to its output.
- The gate inputs are shown in the left columns of the table with all the different possible input combinations. This is normally done by making the inputs count up in binary.
- The gate outputs are shown in the right hand side column.

1.3.1 Truth Table Diagram



1.4 LOGIC GATES

Boolean functions are practically implemented by using electronic gates (aka logic gates). Electronic gates require a power supply. Gate INPUTS are driven by voltages having two nominal values, e.g. 0V and 5V representing logic 0 and logic 1 respectively. The OUTPUT of a gate provides two nominal values of voltage only, e.g. 0V and 5V representing logic 0 and logic 1 respectively.

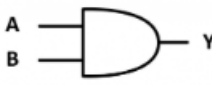
Digital systems are said to be constructed by using logic gates.

The basic gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations of these gates are described below with the aid of truth tables:

1.4.1 AND Gate

The AND gate is an electronic circuit that gives a high output (1) only if all its inputs are high. The AND gate is so named because, if 0 is called "false" and 1 is called "true," the gate acts in the same way as the logical "and" operator.


Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1


$$Y = A.B$$

A dot (\cdot) is used to show the AND operation i.e. $A.B$. This dot is sometimes omitted i.e. AB

1.4.2 OR Gate

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1


$$Y = A+B$$

The OR gate is an electronic circuit that gives a high output (1) if one or more of its inputs are high. A plus (+) is used to show the OR operation.

1.4.3 NOT Gate

Input	Output
A	Y
0	1
1	0

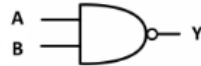


$$Y = \overline{A}$$

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an inverter. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs.

1.4.4 NAND Gate

Inputs		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



$$Y = \overline{A \cdot B}$$

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if any of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

1.4.5 NOR Gate

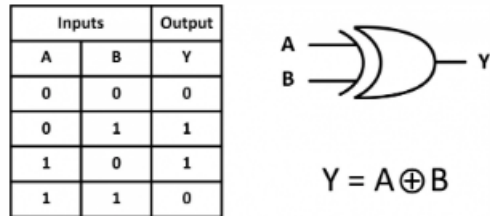
Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



$$Y = \overline{A + B}$$

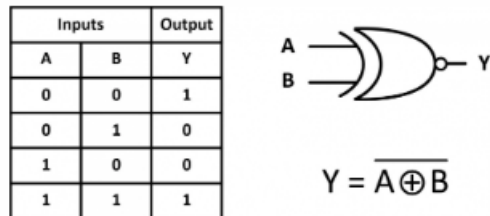
This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if any of the inputs are high. The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

1.4.6 XOR Gate



The 'Exclusive-OR' gate is a circuit which will give a high output if either, but not both, of its two inputs are high. An encircled plus sign (\oplus) is used to show the XOR operation.

1.4.7 XNOR Gate



The 'Exclusive-NOR' gate circuit does the opposite to the XOR gate. It will give a low output if either, but not both, of its two inputs are high. The symbol is an XOR gate with a small circle on the output. The small circle represents inversion.

1.4.8 Universal Gates

The NAND and NOR gates are called universal functions since with either one the AND and OR functions and NOT can be generated.

1.5 FUNCTIONS

Any truth table involving n variables with 2^n outputs can be expressed as a function having input, a combination of n bits mapping to particular output bit. This function involves those n Boolean variables in a logical expression. At times these expressions can be very complicated involving many complex relations between numerous variables, to simplify it we use *K-Maps*.

1.6 K-MAPS

In digital circuits, to find expression with minimum variables, we minimize and simplify Boolean expressions of 3, 4 variables easily using K-map (Karnaugh Maps). K-map is table like representation but it gives more information than TRUTH TABLE.

A boolean function can be represented in two forms (i) Sum of Product (SOP) and (ii) Product of Sum (POS).

1.6.1 Simplifying K-Maps

To simplify expression using K-Maps:

1. Select K-map according to the number of variables.
2. Identify minterms or maxterms as given in problem.
3. For SOP put 1's in blocks of K-map respective to the minterms (0's elsewhere).
4. For POS put 0's in blocks of K-map respective to the maxterms(1's elsewhere).
5. Make rectangular groups containing total terms in power of two like 2,4,8 ..(except 1) and try to cover as many elements as you can in one group.
6. From the groups made in step 5 find the product terms and sum them up for SOP form.

K-Map for 3 variables is as follows:

x	y	z	F
0	0	0	m0
0	0	1	m1
0	1	0	m2
0	1	1	m3
1	0	0	m4
1	0	1	m5
1	1	0	m6
1	1	1	m7

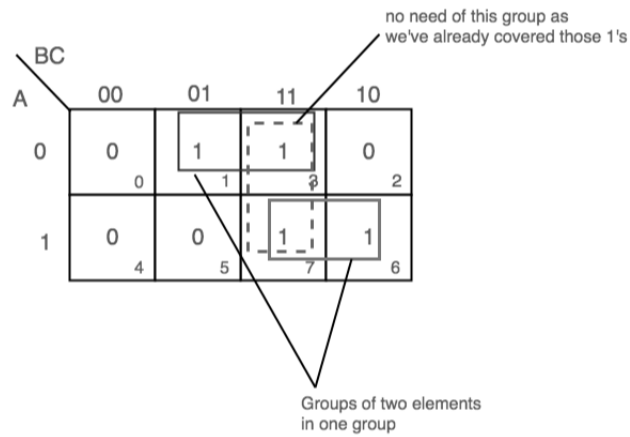
3-variable Truth table of F

		yz			
		y'z'	y'z	yz	yz'
x	x'	m0 x'y'z' 0	m1 x'y'z 1	m3 x'yz 3	m2 x'yz' 2
	y	m4 xy'z' 4	m5 xy'z 5	m7 xyz 7	m6 xyz' 6

3-Variable K-Map, minterm and cell position

1.6.2 Examples

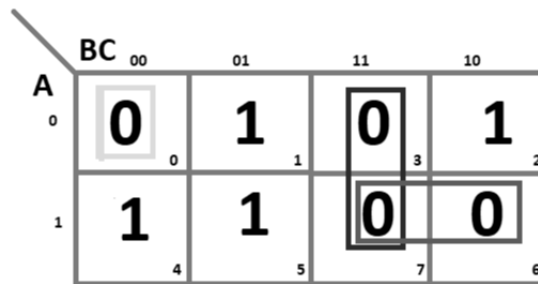
- (i) SOP form - $f(A, B, C) = \Sigma(1, 3, 6, 7)$



From the groups we get terms $\bar{A}C$ and AB . To get SOP form, we sum these product terms, and get the final expression:

$$\bar{A}C + AB$$

(ii) POS form - $f(A, B, C) = \Pi(0, 3, 6, 7)$



After finding terms from the groups and complimenting and adding them we get $\bar{A} + \bar{B}$, $\bar{B} + \bar{C}$, $A + B + C$. To get POS form, we will take product of these three terms:

$$(\bar{A} + \bar{B}).(\bar{B} + \bar{C}).(A + B + C)$$

1.7 ELECTRONIC DEVICES

1.7.1 Adder

An adder adds binary numbers and accounts for values carried in as well as out. A one-bit adder adds three one-bit numbers, often written as A, B, and

C_{in} ; A and B are the operands, and C_{in} is a bit carried in from the previous less-significant stage.

Output carry and sum typically represented by the signals C_{out} and S. Adder is implemented in the following way:

$$\begin{aligned} S &= A \oplus B \oplus C_{in} \\ C_{out} &= AB + C_{in}B + C_{in}A \end{aligned}$$

When performing addition of multi-bit numbers an adder is used for each i^{th} bit and the corresponding output carry of i-th adder becomes input carry for (i+1)th bit adder i.e

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_{in} \\ C_{i+1} &= A_i B_i + C_i B_i + C_i A_i \end{aligned}$$

where C_i represents input carry for adder of i-th bit. Initial carry for adder (c_0) is 0.

A	B	Cin	Cout	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

1.7.2 Subtractor

Subtractors are usually implemented using the same approach as that of an adder, by using the standard *two's complement* notation, by providing an addition/subtraction selector to the initial carry and to invert the second operand.

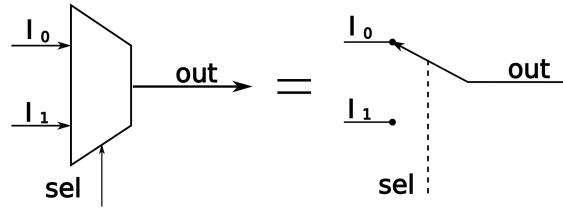
By definition of 2's complement

$$\begin{aligned} -B &= \bar{B} + 1 \\ A-B &= A + \bar{B} + 1 \end{aligned}$$

Addition/subtraction selector can be provided by giving A, ($B \oplus \text{op}$), and op as input, where it acts as adder when op = 0 and as subtractor when op = 1.

1.7.3 MUX

A MUX (we see, 2-to-1 multiplexer) consists of two inputs I_0 and I_1 , one selector input (sel, S) and one output *out*. Depending on the selector value, the output (out, Y) is either of the inputs (I_0 or I_1).



MUX can be equated to a controlled switch; a hardware level analog of if-else. Truth table is as follows

<i>sel</i>	<i>out</i>
0	$I(0)$
1	$I(1)$

and logical expression is

$$Y = I_0\bar{S} + I_1S$$

1.7.4 Comparator

A comparator takes two numbers as input and as the name suggests, it compares them. It will have three possible outputs: greater, equal, and smaller, and only one of them will be set to 1. The outputs depicts $G = 1$ when $A > B$, $E = 1$ when $A = B$ and $L = 1$ when $A < B$.

Eg:

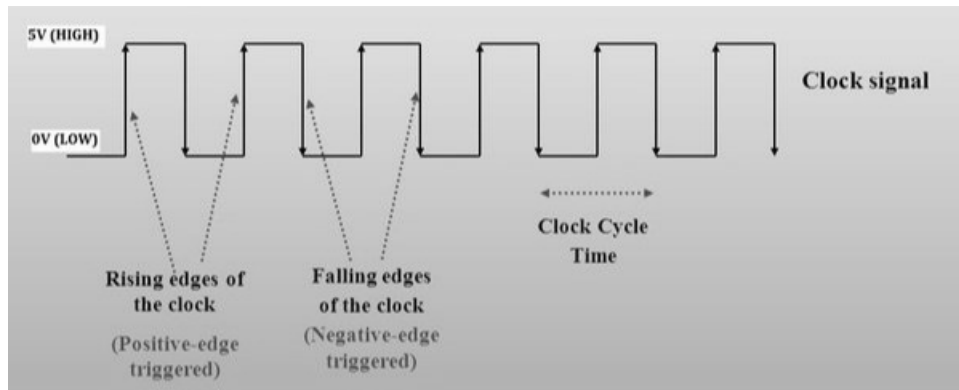
<i>A</i>	0	0	1	1
<i>B</i>	0	1	0	1
<i>G</i>	0	0	1	0
<i>E</i>	1	0	0	1
<i>L</i>	0	1	0	0

From this we observe,

$$\begin{aligned} G &= A \bar{B} \\ E &= A \odot B \\ L &= \bar{A}B \end{aligned}$$

1.7.5 Clock

Clock is used to bring a sense of chronology in computers. To do so, they require an external signal (an alternating boolean variable), apart from standard inputs to operate, such external signal is referred as clock or clock pulse. The idea of clock is to control the output even if the input is given.



- There are two levels, namely logic High and logic Low in clock signal. The signal stays at a *logic level*, either high 5V or low 0V, for an equal amount of time.
- In clock signal, two types of transitions occur, i.e. transition either from Low to High or High to Low. These transitions are called *edge*.
- The transition from low to high is called positive edge, and that from high to low is called negative edge. These transitions can be used for *triggering*.

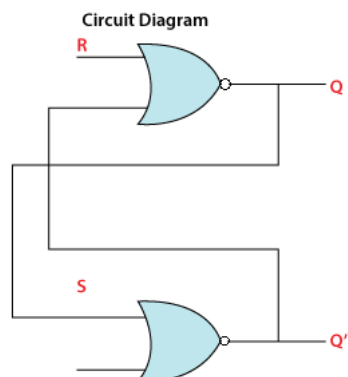
1.7.6 Memory

Latches and flip flops are elements to mimic memory.

1. LATCHES

Latches are *level triggered*.

SR Latch:



Truth table for SR Latch:

	S	R	Q_t	Q_{t+1}
<i>hold</i>	0	0	0	0
<i>hold</i>	0	0	1	1
<i>reset</i>	0	1	0	0
<i>reset</i>	0	1	1	0
<i>set</i>	1	0	1	1
<i>set</i>	1	0	1	1
<i>race</i>	1	1	0	?
<i>race</i>	1	1	1	?

2. FLIP-FLOPS

Flip-flops are *edge triggered*.

2 PHASE II - VERILOG PROGRAMMING

2.1 BASICS

2.1.1 Intro

Verilog is a Hardware Description Language(HDL) used for designing digital circuits, it has a modular structure and syntax more or less like C.

2.1.2 Variable types

A variable in verilog belongs to two types:

1. Net: This variable type is one which stores continuously driven signals of input or output in verilog. Among many examples of this data type is "wire" which we used in many programs.
2. Reg: This variable type retains the last value assigned to it and so used as a storage element.

2.1.3 Timescale

Typical format to declare an inclusion of timescale in verilog program is 'timescale <time_unit>/<time_precision>. Time unit can be 1ms,1ns etc. which gives a time unit to statements like #3 which means a delay by 3 time units. Time precision is the precision of the time clock, the least count.

2.2 VERILOG LANGUAGE FEATURES

2.2.1 Modules

Modules in verilog are same as functions in any higher level language like C in its working. The format of a module in verilog is as follows:

```

module module_name(list_of_ports);
    io declarations
    parallel statements
    ...
    ...
endmodule

```

A verilog program consists of multiple modules which perform different functions. A typical example of a module(AND gate) would be:

```

module and_Gate(a,b,c);
    input a,b;
    output c;
    assign c = a&b;
endmodule

```

Here all parameters are by default 1-bit numbers. One can easily define a variable to be a vector of multiple bits for say a 16 bit variable can be declared as var_type[15:0]var_name. Bits are stored from MSB: var_name[0] to LSB: var_name[15].

Here we see a new keyword "assign". This keyword is used to represent continuous assignment where the left variable is updated whenever the expression on the right changes. Here the LHS variable is usually a wire and RHS can contain both reg and a wire.

2.2.2 Initial Block

It is a block inside a module used for a specific function of running the code whatever is inside it only once the program is run. **Every such code block has a begin and end statement inside which statements are written:**

```

initial begin
    ...
    statements
    ...
end

```

2.2.3 Always Block

Another useful paradigm in verilog programming is an Always block which runs every time the given condition is satisfied. The condition being typically a sensitivity statement which checks for changes in multiple parameters passed into that statement:

```

always @(parameters) begin

```

```

...
    statements
...
end

```

2.2.4 If-else Block

In verilog if-else statements are declared just like C:

```

if (conditional) begin
    operations;
end else begin
    operations;
end

```

Once we are done with making a program using above features in verilog, we need to simulate the system and verify its operations just like running a program in a high level language by providing inputs. For this we need to create a test bench and run it to see the outputs.

2.3 TESTING A VERILOG PROGRAM

2.3.1 Creating a Test bench

A test bench is yet another module inside which we call another module which we want to test, In verilog calling a module inside another module is called instantiation which actually keeps a copy of that module inside it. Then inside an "initial-block" we pass desired inputs and simulate the outputs to readable format using different system tasks like "monitor" and "display". Monitor function is like a smart `printf` statement which only outputs if any one of the parameters changes while display is just like a `printf` statement. A typical example of a test bench is shown below:


```

module tb;

    reg a, B;
    wire C;
    andMod ans(a, B, C);

    initial begin
        $monitor("a = %b, B = %b, C = %b", a, B,C);
        a = 0;
        B = 0;
        #2;

        repeat(4) begin
            a = $random;
            B = $random;
            #15;
        end
    end
endmodule

```

Here the repeat block does the same thing the specified no. of times and the random statement assigns the variable a value randomly. Note how each block is in between a begin and end statement.

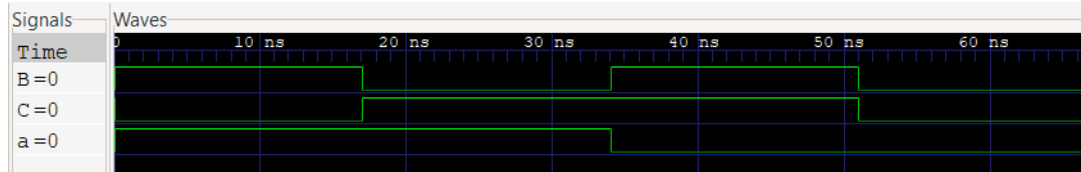
2.3.2 Running the simulation

Now commands iverilog -o name file.v and vvp name gives the output on the terminal. This output can be simulated in gtkwave by adding two lines:

```
$dumpfile("file_name.vcd");
```

```
$dumpvars(0, tb);
```

inside the initial block of test bench. This will create a .vcd file in same directory, now u can type in terminal gtkwave file_name.vcd to open the gtkwave and see the simulation there which typically looks like this:



2.4 FINITE STATE MACHINE(FSM)

2.4.1 Principle

FSM is a reactive system whose response to a particular stimulus (a signal, or a piece of input) is not the same on every occasion, depending on its current “state”. A Finite State Machine is defined by $(\Sigma, S, S_0, \delta, F)$, where:

1. Σ is the input alphabet (a finite, non-empty set of symbols).
2. S is a finite, non-empty set of states.
3. s_0 is an initial state, an element of S .
4. δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$
5. F is the set of final states, a (possibly empty) subset of S .
6. O is the set (possibly empty) of outputs

The Finite State Machine class keeps track of the current state, and the list of valid state transitions. You define each transition by specifying :

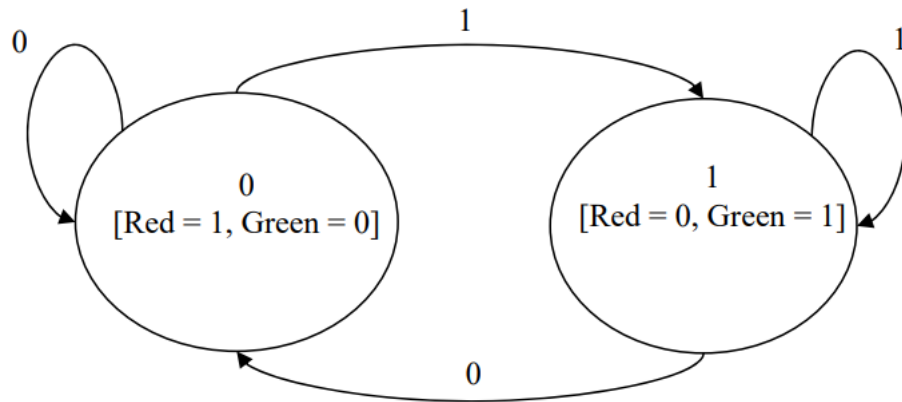
1. FromState - the starting state for this transition
2. ToState - the end state for this transition
3. condition - a callable which when it returns True means this transition is valid.
4. callback - an optional callable function which is invoked when this transition is executed.

2.4.2 An example: Elevator Control

Following example will provide a better picture of FSM:

Step 1: Describing the machine: In this example, we’ll be designing a controller for an elevator. The elevator can be at one of two floors: Ground or First. There is one button that controls the elevator, and it has two values: Up or Down. Also, there are two lights in the elevator that indicate the current floor: Red for Ground, and Green for First. At each time step, the controller checks the current floor and current input, changes floors and lights in the obvious way.

Step 2: Draw FSM diagram.



Step 3: Write the truth table.

CurrentState	Input	NextState	Red	Green
0	0	0	1	0
0	1	1	1	0
1	0	0	0	1
1	1	1	0	1

3 MENTORS MENTEES

Mentors

Aditya Tanwar

Akhil Agrawal

Documentation

Lavesh Gupta

Ujjawal Dubey

Anuj Singhal

Rajat Gattani

Presentation

Mentees

Bornadhya Abhir Rajbongshi

Survati Pal

Aastha Sitpal

Sajal Jain

Nelluru Mourya Reddy

Lavesh Gupta

Rajat Gattani

Ramtej Penta

Devang Kumawat

Madhur Bansal

Hisham Hadi T

Dasari Charithambika

Dhruv

Ujjawal Dubey

Vishant Bhadana

Vaishnavi Singh

Rishi Poonia

Anuj