ECEN 651

LAB NO. 1

LOGIC LEVEL MODELING IN VERILOG

TA: Mr. Ye Wang

DATE: 09/04/2019

Student Name: Dhiraj Dinesh Kudva

UIN: 829009538

Objective: The objectives of this lab are

1. To learn Xilinx Vivado: Through this lab, I will learn how to use Xilinx Vivado software and its various tools useful for design simulation.

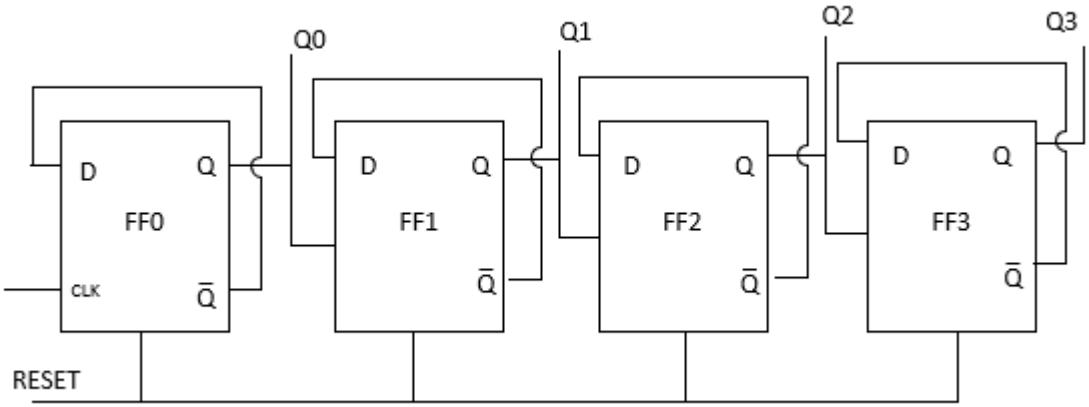
2. Implementation of Ripple counter and 4:1 MUX using Verilog: This lab will also provide an insight of coding in Verilog. With the help of this, I will learn to implement Ripple Carry Counter and 4:1 MUX using Verilog.

3. To learn Testbench simulation: This lab will also teach me how to design the logic for a given circuit and check the output of the design using Verilog.

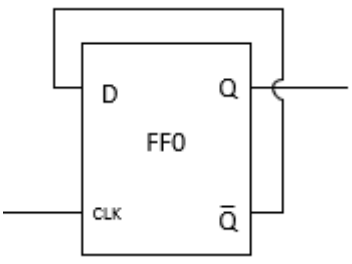
Design:

1. RIPPLE COUNTER

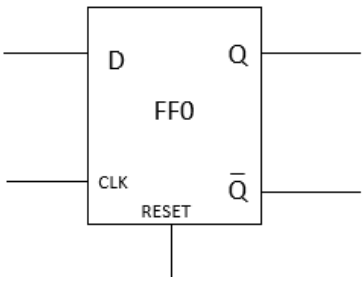
The ripple counter is executed using T flipflop and T flipflop is executed using D flipflop.



Ripple Carry Counter



T Flipflop



D Flipflop

Steps:

1. Open the Xilinx Vivado software.
2. Create new project and in that create new source files in that project.
3. Create separate module for D flip flop, T flip flop and Ripple carry counter.

4. Write Verilog codes for D flip flop. The D flip flop module is then inherited by the T flip flop, which is then inherited by Ripple Carry Counter. (Codes for each module is written below)
5. After this step, write test bench for simulating the ripple carry counter.
6. Mention the delay and clock cycle in this test bench. Initialization of the variables should be done in the test bench.

Verilog Code for Ripple Carry Counter

Ripple Carry Counter

1. D flip flop module

```
module D_FF(q,d,clk,reset);
  output q;                //input output initialization
  input d, clk, reset;
  reg q;
  // logic for d flipflop
  always @(posedge reset or negedge clk)
  if (reset)
  q = 1'b0;
  else
  q=d;
endmodule
```

2. T flipflop module

```
module T_FF (q, clk, reset);
output q;                //input and output initialization
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
//connecting the d flipflop to the negative output to convert it into toggle or
T-Flipflop
not n1(d, q); //not is a verilog provided primitive
endmodule
```

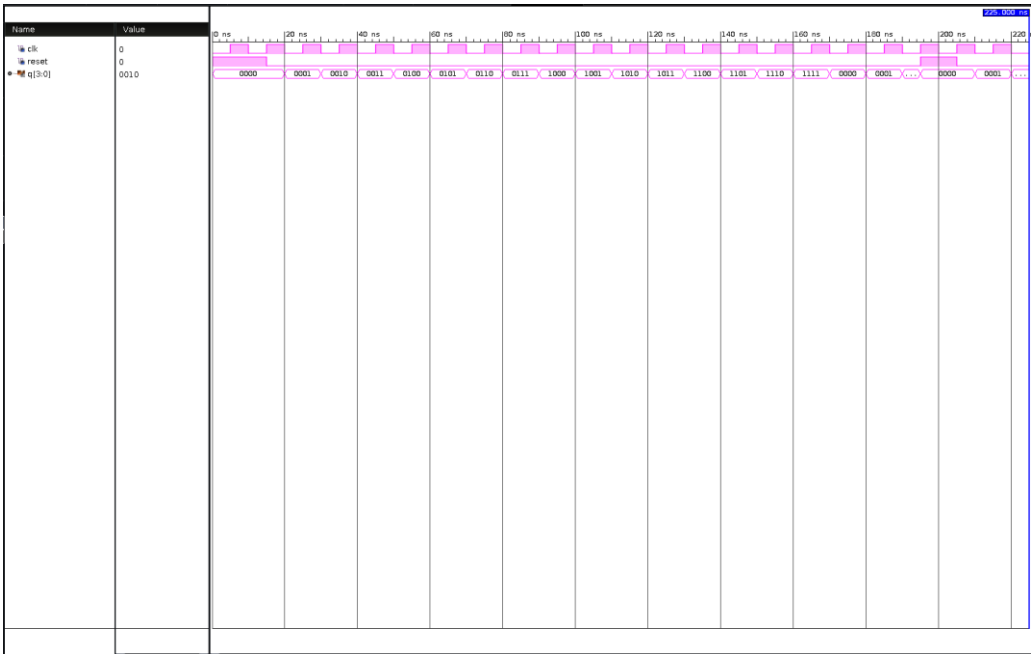
3. Ripple Carry Counter Module

```
module RCC(q,clk,reset); //main module for Ripple Carry counter
 output [3:0] q;              //initializing inputs and outputs

input clk, reset;
 //linking T flipflops : Output of T flipflop is connected to clock of next flipflop
T_FF tff0(q[0], clk, reset);
T_FF tff1(q[1], q[0], reset);
T_FF tff2(q[2], q[1], reset);
T_FF tff3(q[3], q[2], reset);
Endmodule
```

Testbench for Ripple Counter

```
module StimulusBlock;

reg clk;
reg reset;
wire [3:0] q;
// instantiate the design block
RCC r1(q, clk, reset);
// control the clock signal that drives the design block. Cycle time = 10
initial
clk = 1'b0;
always

#5 clk=~clk;  // togle clk every 5 time units
// control the reset signal that drives the design block
// reset is asserted from 0 to 20 and from 200 to 220
initial
begin
reset = 1'b1;
#15 reset = 1'b0;
#180 reset = 1'b1;
#10 reset = 1'b0;
#20 $finish; // teriminate the simulation
end
// Monitor the outputs
initial
$monitor($time, " Clk %b, Output q = %d",clk,q);
endmodule
```

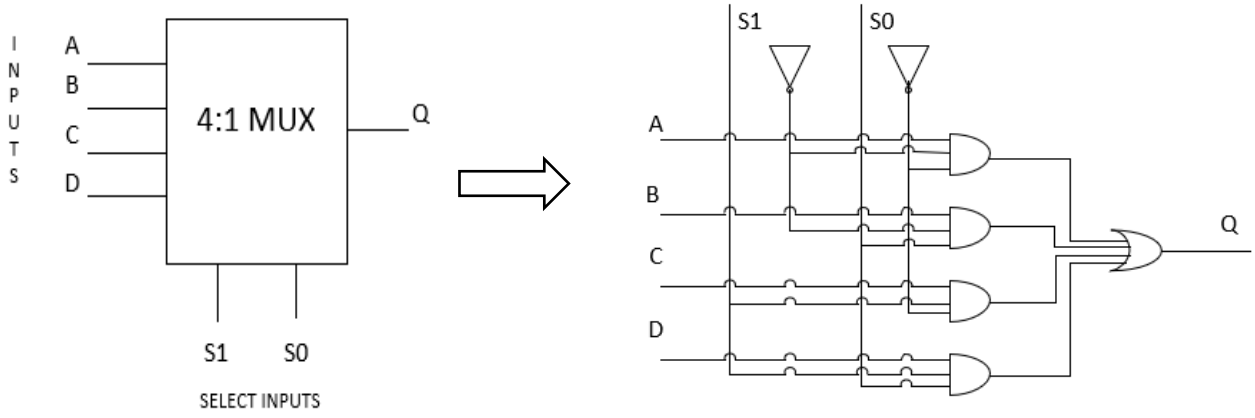
Output Waveform:



2. 4:1 MUX

The 4: 1 Multiplexer has four inputs and single output. The output is selected based on the two select inputs S0 and S1. The figure below is schematic level representation of 4:1 MUX using logic gates.

Schematics: Gate level Schematic



Truth Table for 4:1 MUX

Select Inputs		Outputs
S1	S0	
0	0	A
0	1	B
1	0	C
1	1	D

Steps:

1. Open Xilinx Vivado software.
2. Create new file and source files for 4:1 Mux
3. Based on the above truth table and schematic, generate Verilog code.
4. Create a new test bench and add delays in input so that a stable and clear waveform can be generated at output.
5. Run the simulation file and check the waveform. Compare the output waveform with the truth table.

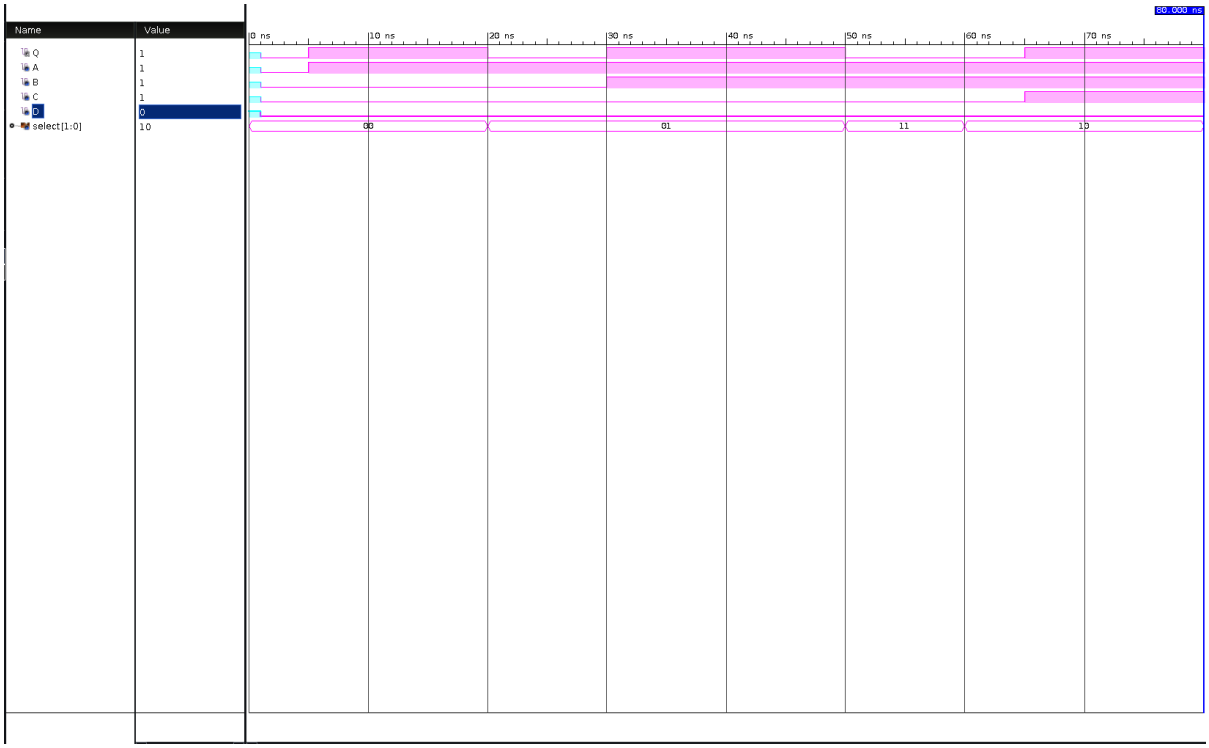
Verilog Code for 4:1 Multiplexer

```
module DEMO1(Q,A,B,C,D,select);
   input A,B,C,D; //defining inputs and outputs
   input [1:0] select;
   output Q;

 //the assign statement complies with the truth table given above
// if S0 & S1 both are zero, then the output matches with input A
// if S0=1 & S1=0, then the output matches with input B
// if S0=0 & S1=1, then the output matches with input C
// if S0 & S1 both are one, then the output matches with input D

 assign Q = select[1]? (select[0]? D:C) : (select[0] ? B: A);
endmodule
```

Test Bench for 4:1 Mux

```
module testbenchmux;
wire Q;    //input and output initialization
reg A,B,C,D;
reg [1:0] select;

// instantiate the design block
DEMO1 udt(Q,A,B,C,D,select); //module for testbench
// initializing the inputs to zero
initial begin
#1 A = 1'b0;B = 1'b0;C = 1'b0;D = 1'b0;
end
//creating input waveforms for testing the 4:1 MUX Logic by toggling after
stipulated delays
always
begin
#5 A=~A;
#25 B=~B;
#10 C=~C;
#45 D=~D;
end
 // creating waveforms for select inputs
initial
begin
select = 2'b00;
#20select = 2'b01;
#30 select = 2'b11;
#10 select = 2'b10;
#20 $finish; // teriminate the simulation
end
```

Output for 4:1 Multiplexer



Questions:

a. What does the $monitor statement in the provided test bench do? What about $finish?

Ans. The $monitor displays output of the parameters whenever it changes, in the format mentioned in its syntax. The $finish makes the simulator exit and passes the control back to the operating system host.

b. Similarly, what do the # symbols signify in the provided test bench? What about $time?

Ans: The # symbols provided in the testbench signifies delay and the number after the # symbol is the time units after which the statement following it will execute. The $time function returns a 64-bit integer value of the current time, but scaled to time scale unit

c. The 4:1 multiplexer above can be built directly using gates, or it can be constructed using three 2:1 multiplexers, of which are constructed from gates. Provide some intuition as to how the delay through the 4:1 multiplexer would differ for both the aforementioned cases.

Ans: Each 2:1 Mux requires 4 logical gates (1 NOT, 2 AND and 1 OR), while 4:1 MUX requires 7 logical gates (4 nos. of 3 input AND gate, 1 no. of 4 input OR gate and 2 NOT gates). So, if we use three numbers of 2:1 MUX to implement 4:1 MUX, the total number of gates would be 12, 5 more than a standard 4:1 MUX. Each gate has its own latency, that is, the time required to provide output after the input is provided. So, the 4:1 MUX constructed using 2:1 MUX would have latency or time delay of more than 5 gates when compared with standard 4:1 MUX constructed using gates. Hence the delay in 4:1 MUX constructed using 2:1 MUX would be greater.