**VLSI**

**PROJECT**

BY :

SHIVANG SHARMA

**ABSTRACT :**

##### Basic circuits written in Verilog HDL, simulated and implemented on the FPGA.

**OBJECTIVE** **:**

To implement and observe the output waveforms using Verilog and Xilinx vivado

of the following circuits:

* Half adder
* Full adder
* Half subtracter
* Full subtracter
* Encoder
* Decoder
* Multiplexer
* Demultiplexer
* Comparator
* All basic gates
* D flip flop
* JK flip flop

**INTRODUCTION :**

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

We will use Verilog to implement the circuits on FPGA board using Xilinx vivado. Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks.

Here we do not have a FPGA board so we will implement the circuits in Xilinx vivado itself.

**METHODOLOGY :**

First we need to make truth table of the circuit, then using this truth table derive the Boolean expression of the output using simplification or K-Map. In Verilog, we have to benches namely design bench and test bench.

DESIGN BENCH : Give the name of the module and specify all the input and output port.

Then, with the help of the Boolean expression derived above, design circuit by any of the modelling techniques namely Gate level, Dataflow and behavioral modelling.

TEST BENCH : Mention all the input and output of the circuit as register , wire ,etc.

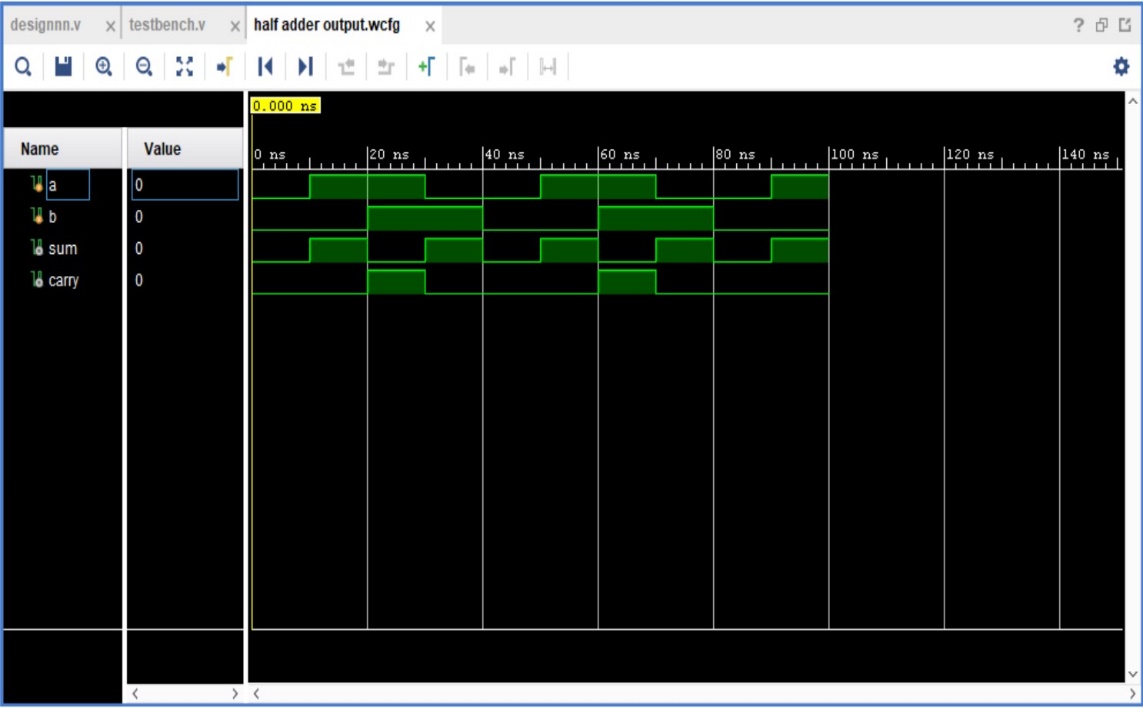
Set initial values of input under INITIAL block. Now to verify the circuit, check the output with different values of input which can be done under ALWAYS block.

Finally run stimulation and observe the waveform to verify the working of the circuit.

**CODE :**

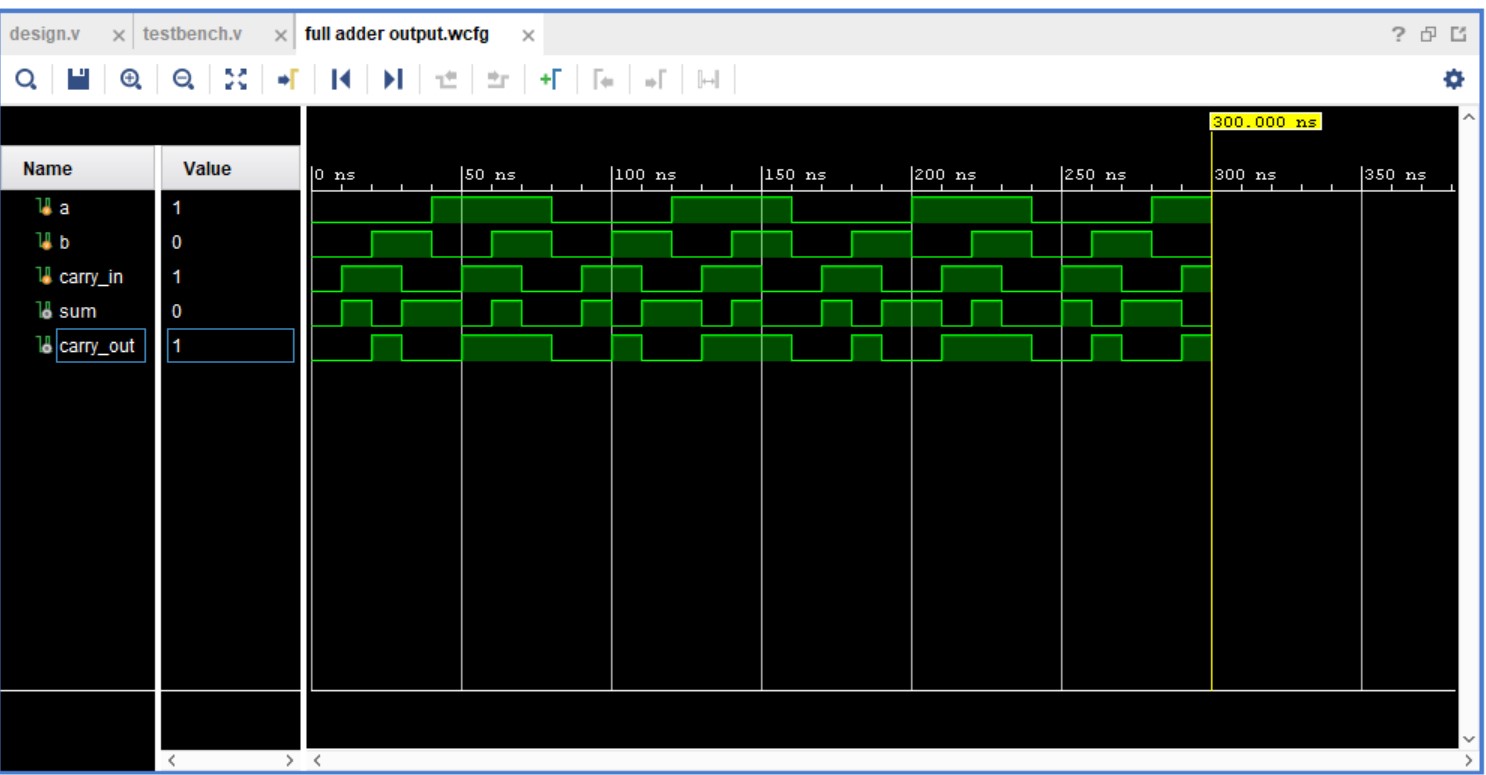
* **HALF ADDER**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg a;  reg b;  wire sum;  wire carry;  half\_adder uut(.a(a) , .b(b), .sum(sum), .carry(carry));  initial  begin  a=0;  b=0;  end  always  begin  #10 a=~a;  #10 b=~b;  end  always  #100 $finish;  endmodule | module half\_adder(  input a,  input b,  output sum,  output carry  );  xor(sum,a,b);  and(carry,a,b);  endmodule |



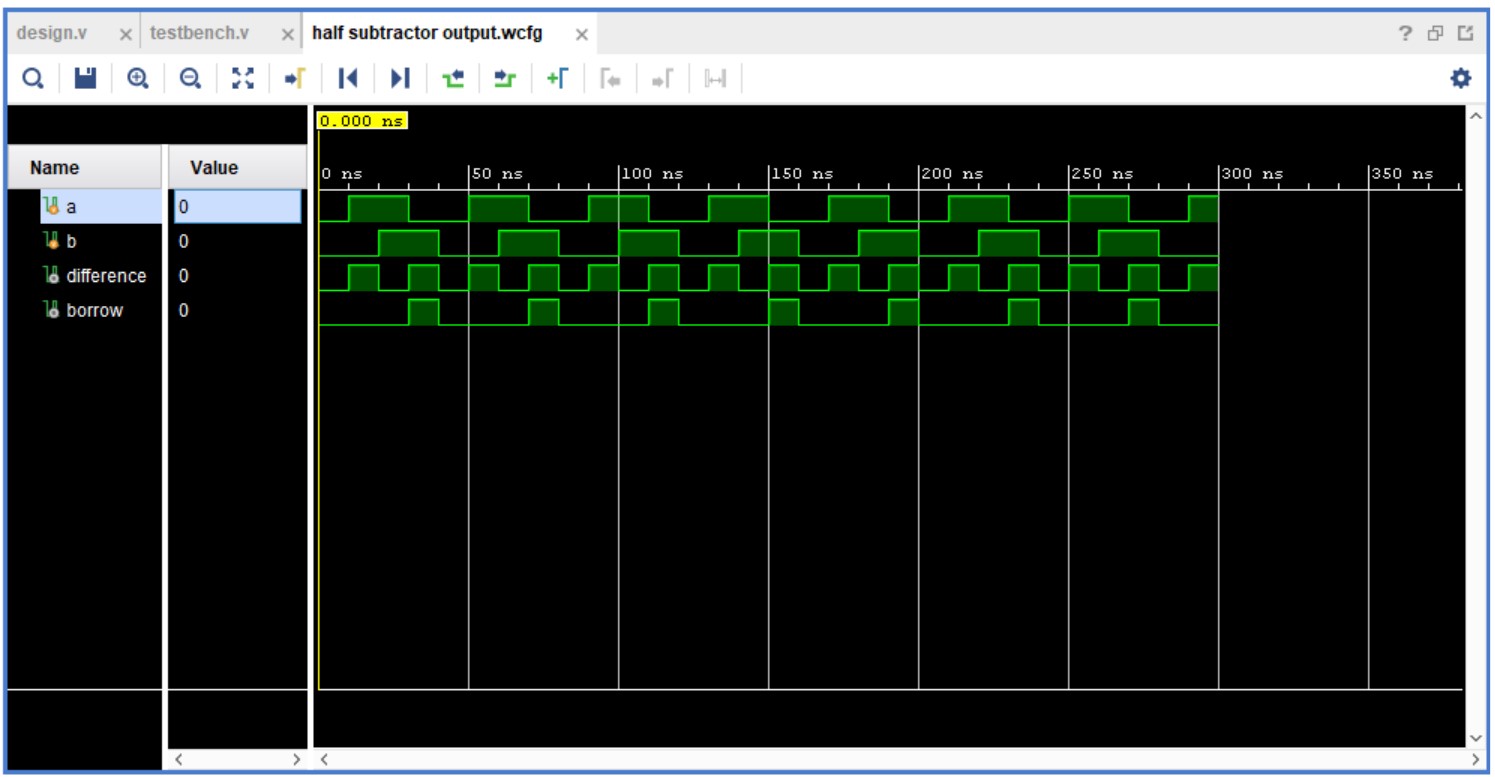
* **FULL ADDER**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg a;  reg b;  reg carry\_in;  wire sum;  wire carry\_out;  full\_adder uut(.a(a),.b(b),.carry\_in(carry\_in),.sum(sum),  .carry\_out(carry\_out));  initial  begin  a=0;  b=0;  carry\_in=0;  end  always  begin  #10 carry\_in=~carry\_in;  #10 b=~b;  end  always  #40 a=~a;  always  #300 $finish;  endmodule | module full\_adder(  input a,  input b,  input carry\_in,  output sum,  output carry\_out  );  assign sum = a^b^carry\_in;  assign carry\_out = (a&b)+((carry\_in)&(a^b));  endmodule |

****

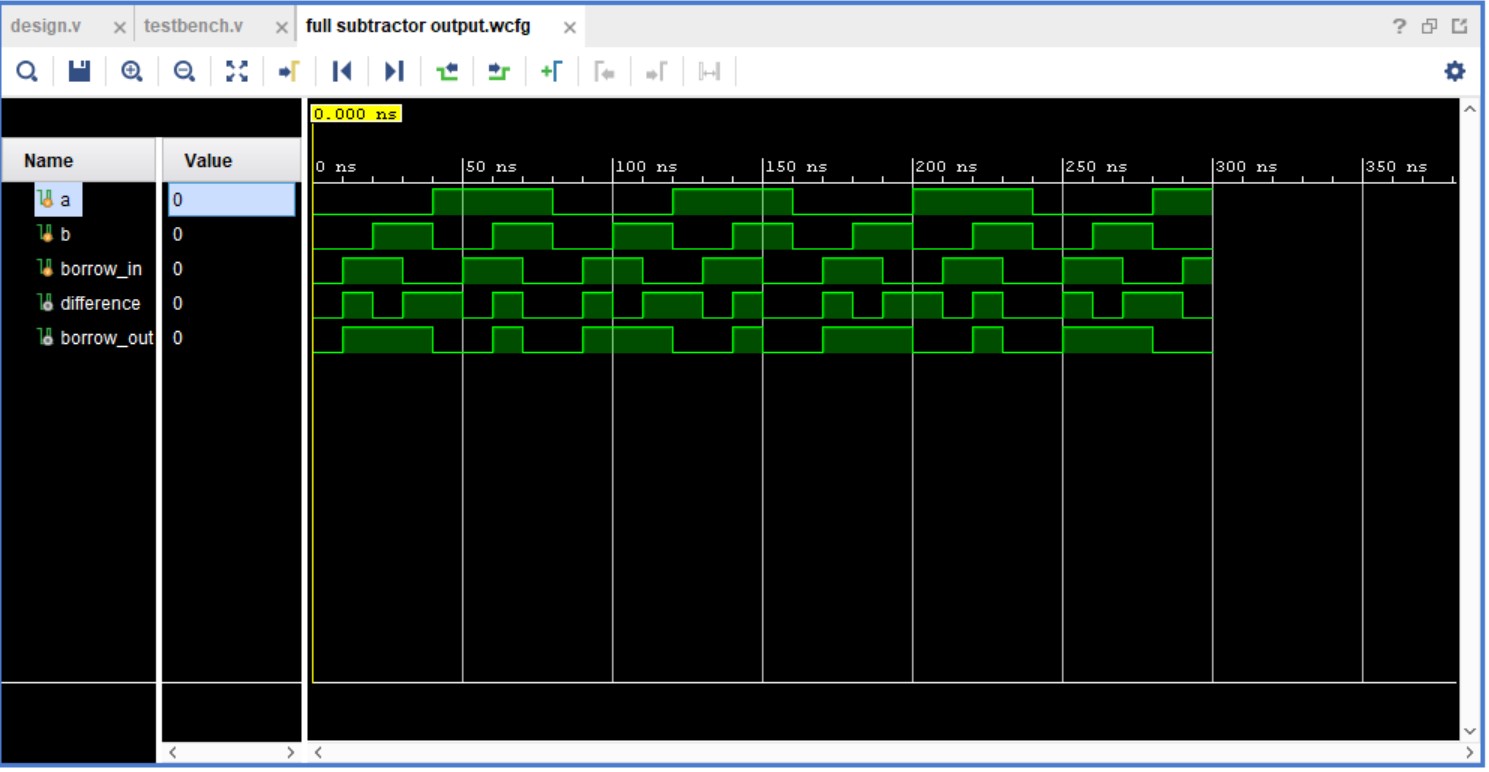
* **HALF SUBTRACTER**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg a;  reg b;  wire difference;  wire borrow;  half\_subtractor uut(.a(a),.b(b),.difference(difference),  .borrow(borrow));  initial  begin  a=0;  b=0;  end  always  begin  #10 a=~a;  #10 b=~b;  end  always  #300 $finish;  endmodule | module half\_subtractor(  input a,  input b,  output difference,  output borrow  );  assign difference = a^b;  assign borrow =(~a)&b;  endmodule |

****

* **FULL SUBTRACTER**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg a;  reg b;  reg borrow\_in;  wire difference;  wire borrow\_out;  full\_subtractor uut(.a(a),.b(b),.borrow\_in(borrow\_in),  .difference(difference),.borrow\_out(borrow\_out));  initial  begin  a=0;  b=0;  borrow\_in=0;  end  always  begin  #10 borrow\_in=~borrow\_in;  #10 b=~b;  end  always  #40 a=~a;  always  #300 $finish;  endmodule | module full\_subtractor(  input a,  input b,  input borrow\_in,  output difference,  output borrow\_out  );  assign difference = a^b^borrow\_in;  assign borrow\_out = ((~a)&b)+((borrow\_in)&(~(a^b)));  endmodule |

****

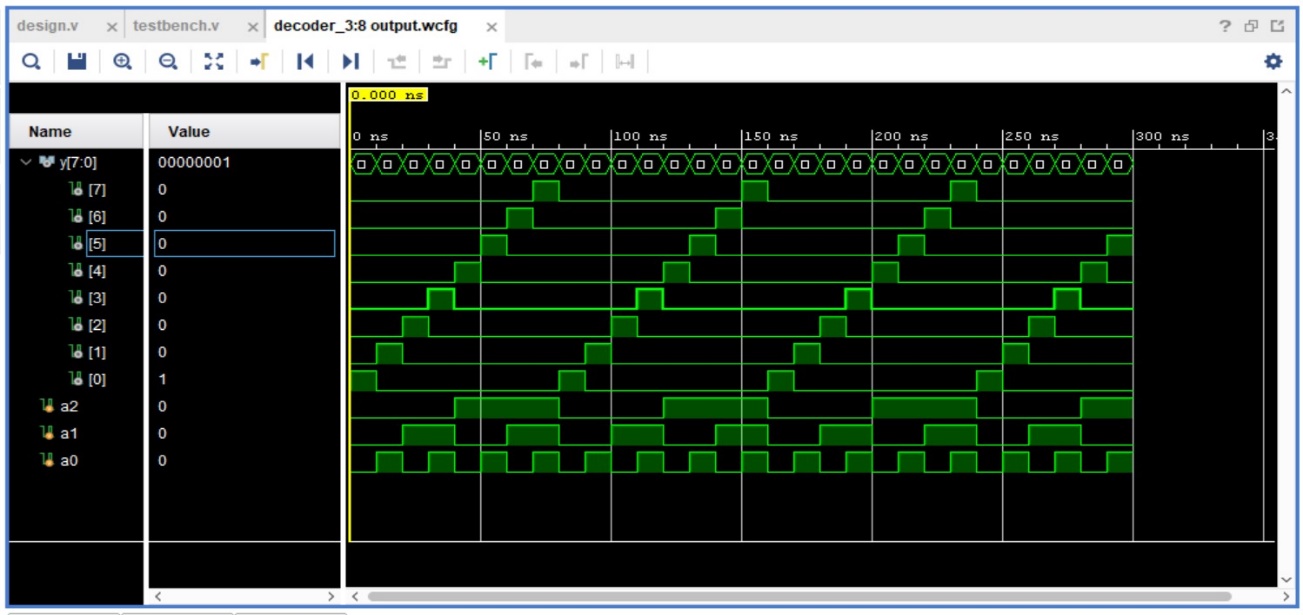
* **ENCODER**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg [7:0]y;  wire a2;  wire a1;  wire a0;  encoder\_8to3 uut(.y(y),.a2(a2),.a1(a1),.a0(a0));  initial  begin  y[0]=0;  y[1]=0;  y[2]=0;  y[3]=0;  y[4]=0;  y[5]=0;  y[6]=0;  y[7]=0;  end  always  begin  #10 y=8'b00000001;  #10 y=8'b00000010;  #10 y=8'b00000100;  #10 y=8'b00001000;  #10 y=8'b00010000;  #10 y=8'b00100000;  #10 y=8'b01000000;  #10 y=8'b10000000;  end  always  #300 $finish;  endmodule | module encoder\_8to3(  input [7:0]y,  output a2,  output a1,  output a0  );  assign a2=y[7]+y[6]+y[5]+y[4];  assign a1=y[7]+y[6]+y[3]+y[2];  assign a0=y[7]+y[5]+y[3]+y[1];  endmodule |

****

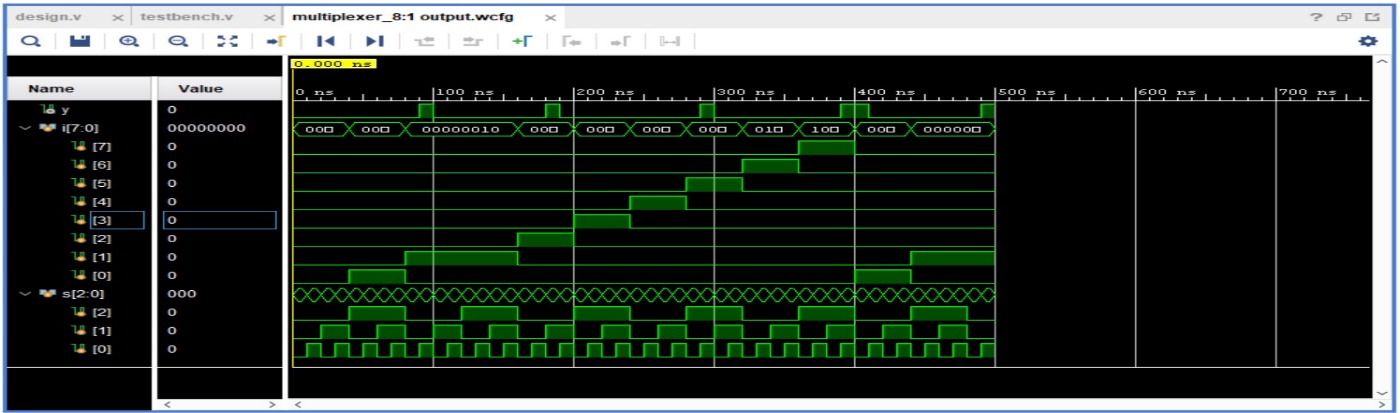
* **DECODER**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg a0;  reg a1;  reg a2;  wire [7:0]y;  decoder\_3to8 uut(.y(y),.a2(a2),.a1(a1),.a0(a0));  initial  begin  a0=0;  a1=0;  a2=0;  end  always  begin  #10 a0=~a0;  #10 a1=~a1;  a0=~a0;  #10 a0=~a0;  #10 a2=~a2;  a1=~a1;  a0=~a0;  end  always  #300 $finish;  endmodule | module decoder\_3to8(  input a0,  input a1,  input a2,  output [7:0]y  );  assign y[7]=a0&a1&a2;  assign y[6]=(~a0)&a1&a2;  assign y[5]=a0&(~a1)&a2;  assign y[4]=(~a0)&(~a1)&a2;  assign y[3]=a0&a1&(~a2);  assign y[2]=(~a0)&a1&(~a2);  assign y[1]=a0&(~a1)&(~a2);  assign y[0]=(~a0)&(~a1)&(~a2);  endmodule |

****

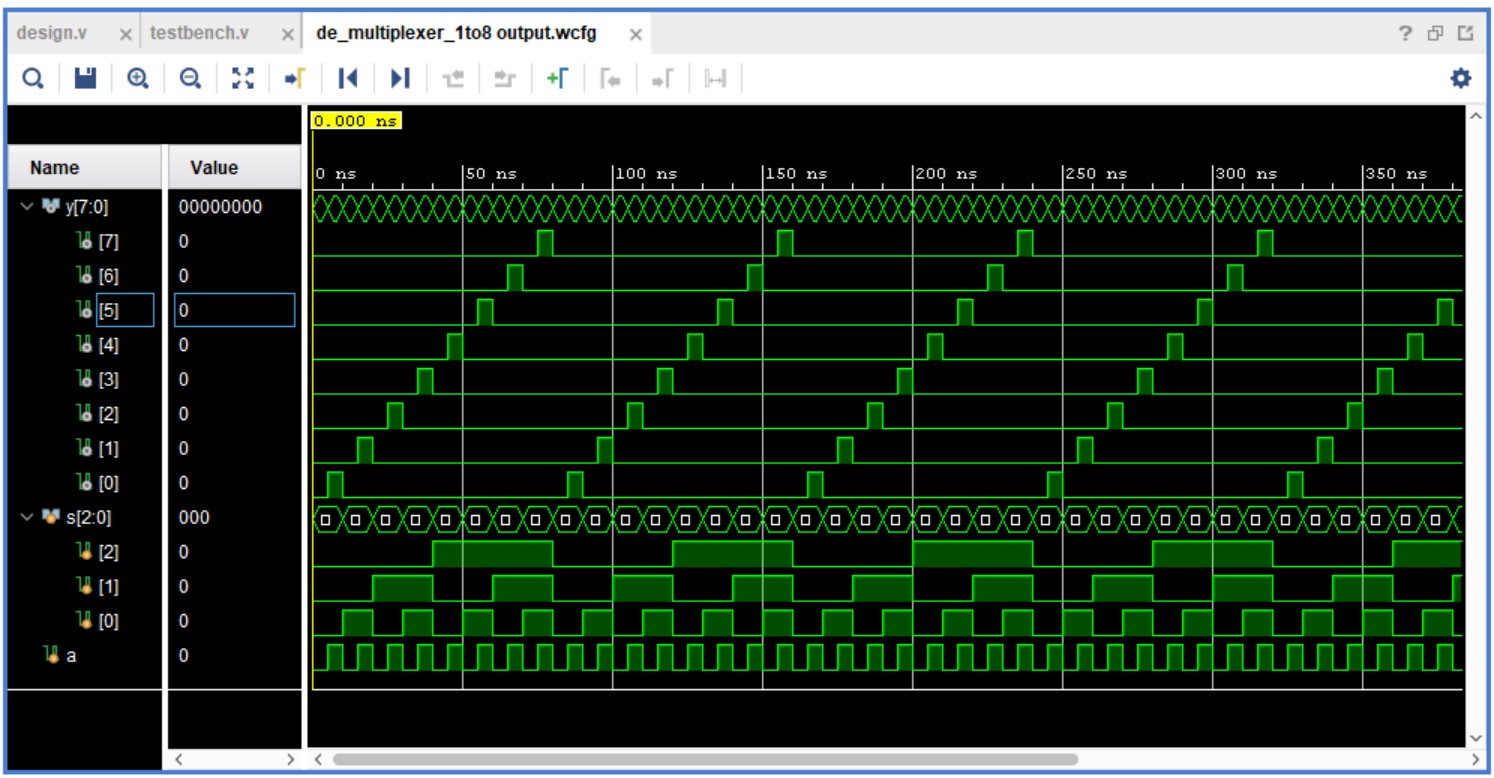
* **MULTIPLEXER**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg [7:0]i;  reg [2:0]s;  wire y;  multiplexer\_8to1 uut(.y(y),.s(s),.i(i));  initial  begin  s=3'b000;  i=8'b00000000;  end  always  begin  #10 s[0]=~s[0];  #10 s[1]=~s[1];  s[0]=~s[0];  #10 s[0]=~s[0];  #10 s[2]=~s[2];  s[1]=~s[1];  s[0]=~s[0];  end  always  begin  #40 i=8'b0000\_0001;  #40 i=8'b0000\_0010;  #40 i=8'b0000\_0010;  #40 i=8'b0000\_0100;  #40 i=8'b0000\_1000;  #40 i=8'b0001\_0000;  #40 i=8'b0010\_0000;  #40 i=8'b0100\_0000;  #40 i=8'b1000\_0000;  end  always  #500 $finish;  endmodule | module multiplexer\_8to1(  input [7:0]i,  input [2:0]s,  output y  );  reg y;  always@(s or i)  begin  if(s==3'b000)  y=i[0];  else if(s==3'b001)  y=i[1];  else if(s==3'b010)  y=i[2];  else if(s==3'b011)  y=i[3];  else if(s==3'b100)  y=i[4];  else if(s==3'b101)  y=i[5];  else if(s==3'b110)  y=i[6];  else if(s==3'b111)  y=i[7];  end  endmodule |

****

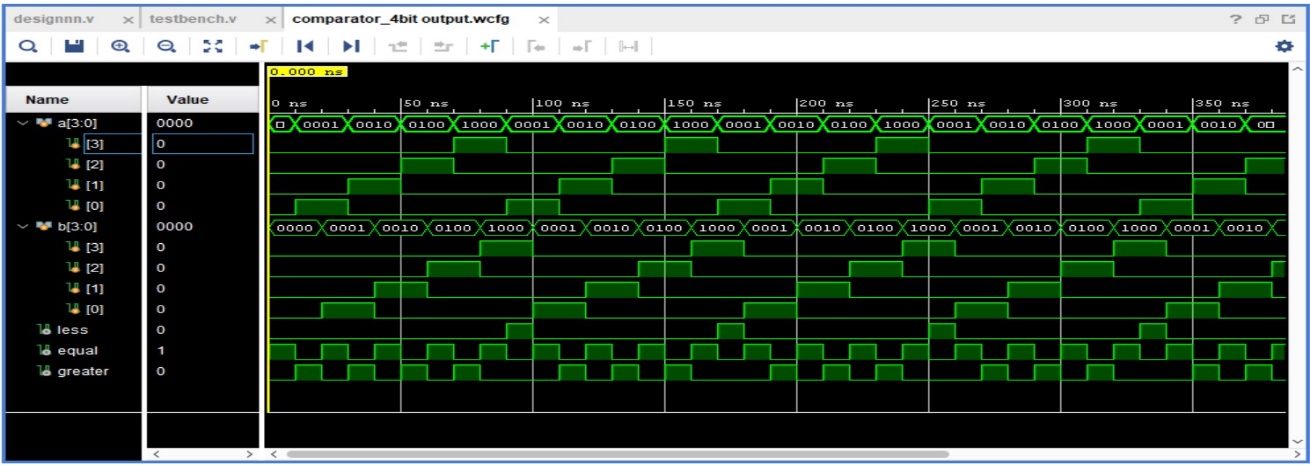
* **DEMULTIPLEXER**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg a;  reg [2:0]s;  wire [7:0]y;  de\_multiplexer\_1to8 uut(.y(y),.s(s),.a(a));  initial  begin  s=3'b000;  a=0;  end  always  begin  #10 s[0]=~s[0];  #10 s[1]=~s[1];  s[0]=~s[0];  #10 s[0]=~s[0];  #10 s[2]=~s[2];  s[1]=~s[1];  s[0]=~s[0];  end  always  begin  #5 a=~a;  end  always  #500 $finish;  endmodule | module de\_multiplexer\_1to8(  input a,  input [2:0]s,  output [7:0]y  );  assign y[0]=((~s[0])&(~s[1])&(~s[2]))&(a);  assign y[1]=((s[0])&(~s[1])&(~s[2]))&(a);  assign y[2]=((~s[0])&(s[1])&(~s[2]))&(a);  assign y[3]=((s[0])&(s[1])&(~s[2]))&(a);  assign y[4]=((~s[0])&(~s[1])&(s[2]))&(a);  assign y[5]=((s[0])&(~s[1])&(s[2]))&(a);  assign y[6]=((~s[0])&(s[1])&(s[2]))&(a);  assign y[7]=((s[0])&(s[1])&(s[2]))&(a);  endmodule |

****

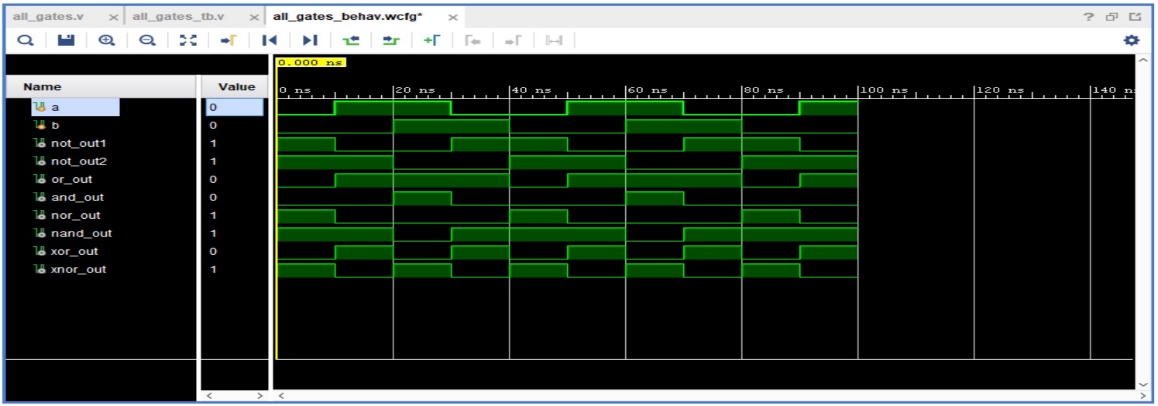
* **COMPARATOR**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg [3:0] a;  reg [3:0] b;  wire less;  wire equal;  wire greater;  comparator\_4bit uut (  .a(a),  .b(b),  .less(less),  .equal(equal),  .greater(greater)  );  initial  begin  a=4'b0000;  b=4'b0000;  end  always  begin  #10 a=4'b0001;  #10 b=4'b0001;  #10 a=4'b0010;  #10 b=4'b0010;  #10 a=4'b0100;  #10 b=4'b0100;  #10 a=4'b1000;  #10 b=4'b1000;  end  always  #500 $finish;  endmodule | module comparator\_4bit(  a,b,less,equal,greater  );  input [3:0] a;  input [3:0] b;  output less;  output equal;  output greater;  reg less;  reg equal;  reg greater;  always @(a or b)  begin  if(a > b)  begin  less = 0;  equal = 0;  greater = 1;  end  else if(a == b)  begin  less = 0;  equal = 1;  greater = 0;  end  else  begin  less = 1;  equal = 0;  greater =0;  end  end  endmodule |

****

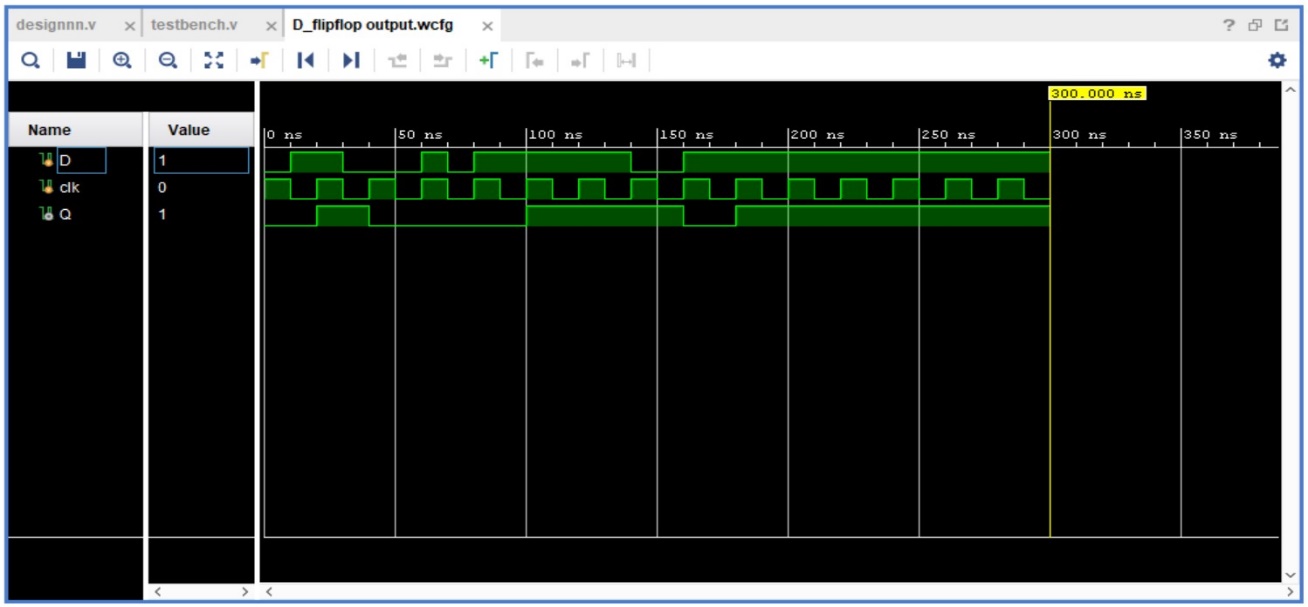
* **ALL GATES**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module all\_gates\_tb();  reg a;  reg b;  wire not\_out1;  wire not\_out2;  wire or\_out;  wire and\_out;  wire nor\_out;  wire nand\_out;  wire xor\_out;  wire xnor\_out;  all\_gates uut( .a(a),  .b(b),  .not\_out1(not\_out1),  .not\_out2(not\_out2),  .or\_out(or\_out),  .and\_out(and\_out),  .nor\_out(nor\_out),  .nand\_out(nand\_out),  .xor\_out(xor\_out),  .xnor\_out(xnor\_out));  initial  begin  a=0;  b=0;  end  always  begin  #10 a=~a;  #10 b=~b;  end  always  #100 $finish;  endmodule | module all\_gates(  input a,  input b,  output not\_out1,  output not\_out2,  output or\_out,  output and\_out,  output nor\_out,  output nand\_out,  output xor\_out,  output xnor\_out  );  not(not\_out1,a);  not(not\_out2,b);  or(or\_out,a,b);  and(and\_out,a,b);  nor(nor\_out,a,b);  nand(nand\_out,a,b);  xor(xor\_out,a,b);  xnor(xnor\_out,a,b);  endmodule |

****

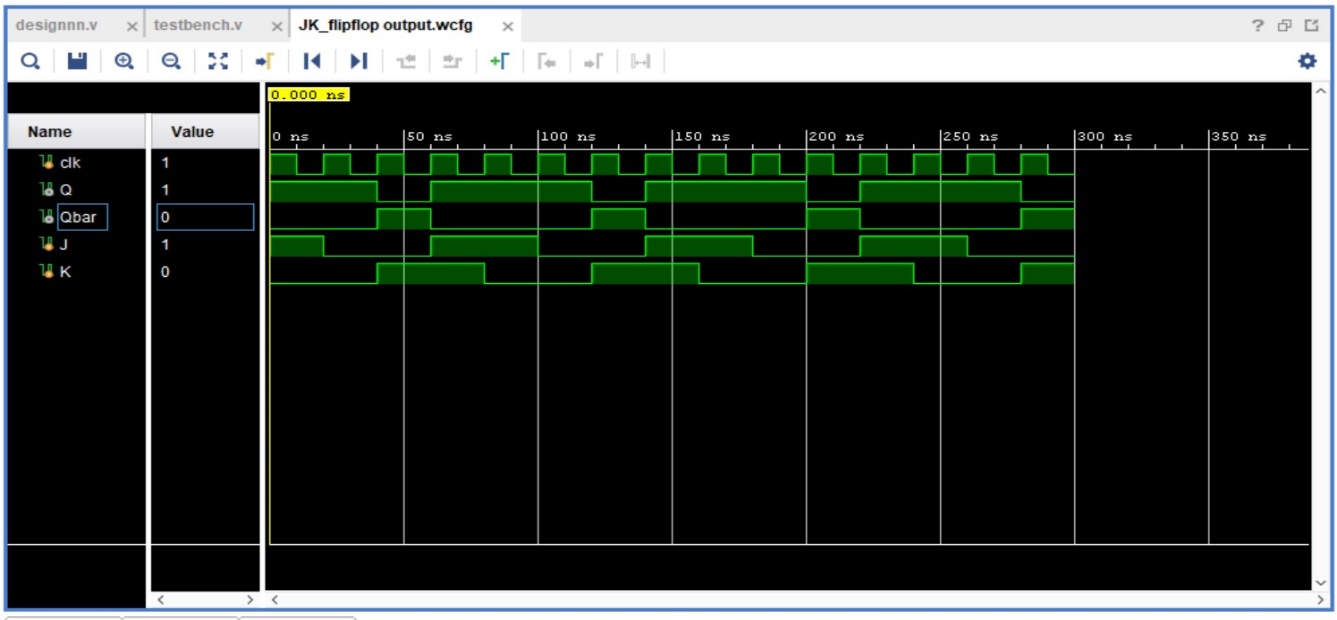
* **D FLIPFLOP**

|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg D;  reg clk;  wire Q;  possitive\_edge\_Dflipflop uut(.D(D),.clk(clk),.Q(Q));  initial  begin  clk=1;  D=0;  end  always  begin  #10 clk=~clk;  end  initial  begin  #10 D<=1;  #20 D<=0;  #30 D<=1;  #10 D<=0;  #10 D<=1;  #60 D<=0;  #20 D<=1;  end;  always  #300 $finish;  endmodule | module possitive\_edge\_Dflipflop(D,clk,Q);  input D;  input clk;  output Q;  reg Q;  always @(posedge clk)  begin  Q <= D;  end  endmodule |

****

* **JK FLIPFLOP**

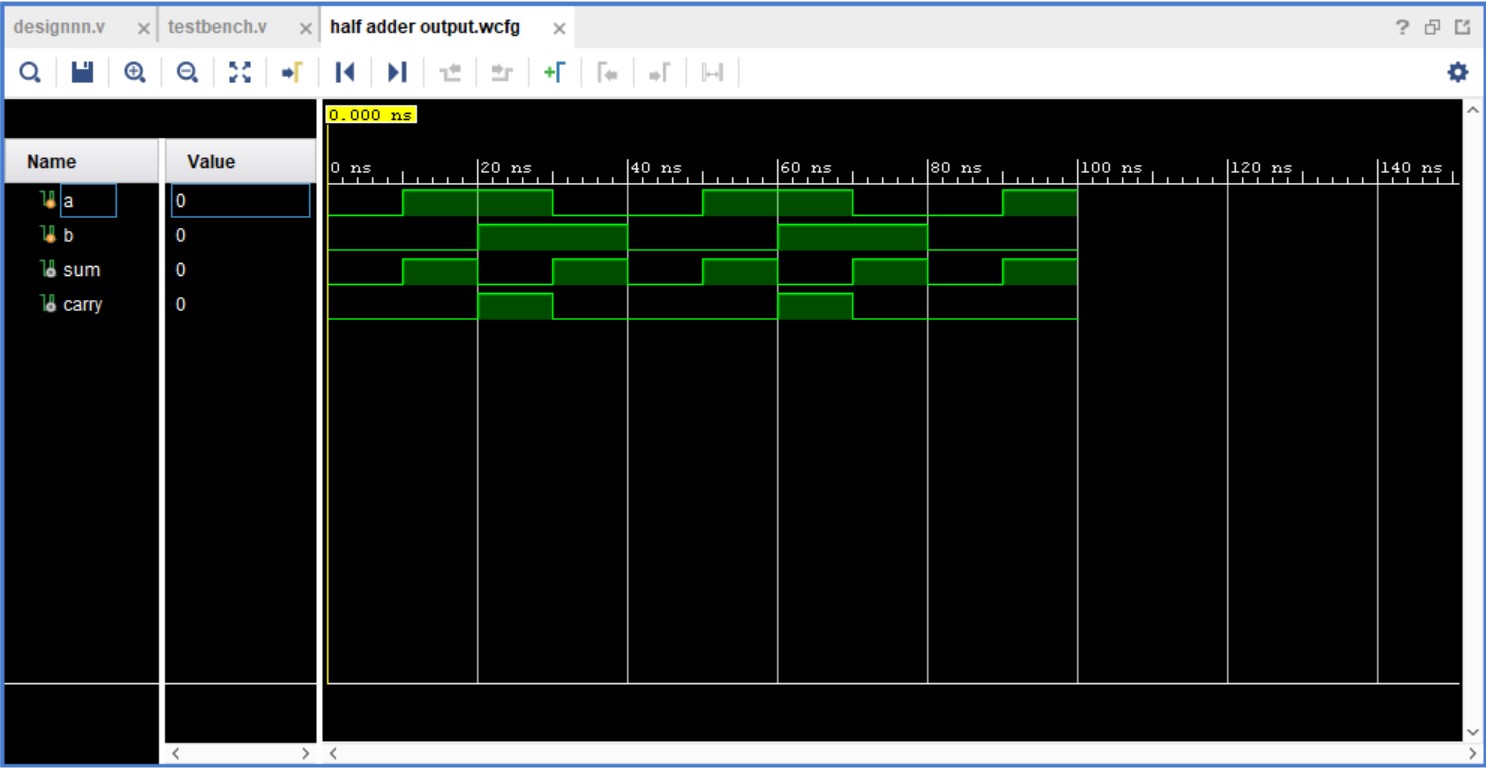
|  |  |
| --- | --- |
| **TEST BENCH** | **DESIGN BENCH** |
| module testbench();  reg J;  reg K;  reg clk;  wire Q;  wire Qbar;  possitive\_edge\_JKflipflop uut(.J(J),.K(K),.clk(clk),.Q(Q),.Qbar(Qbar));  initial  begin  clk=1;  J=1;  K=0;  end  always  begin  #10 clk=~clk;  end  always  begin  #20 J=~J;  #20 K=~K;  end  always  #300 $finish;  endmodule | module possitive\_edge\_JKflipflop(J,K,clk,Q,Qbar);  input J;  input K;  input clk;  output Q;  output Qbar;  reg Q;  always @ (posedge clk)  case ({J,K})  2'b00 : Q <= Q;  2'b01 : Q <= 0;  2'b10 : Q <= 1;  2'b11 : Q <= ~Q;  endcase  assign Qbar=~Q;  endmodule |

****

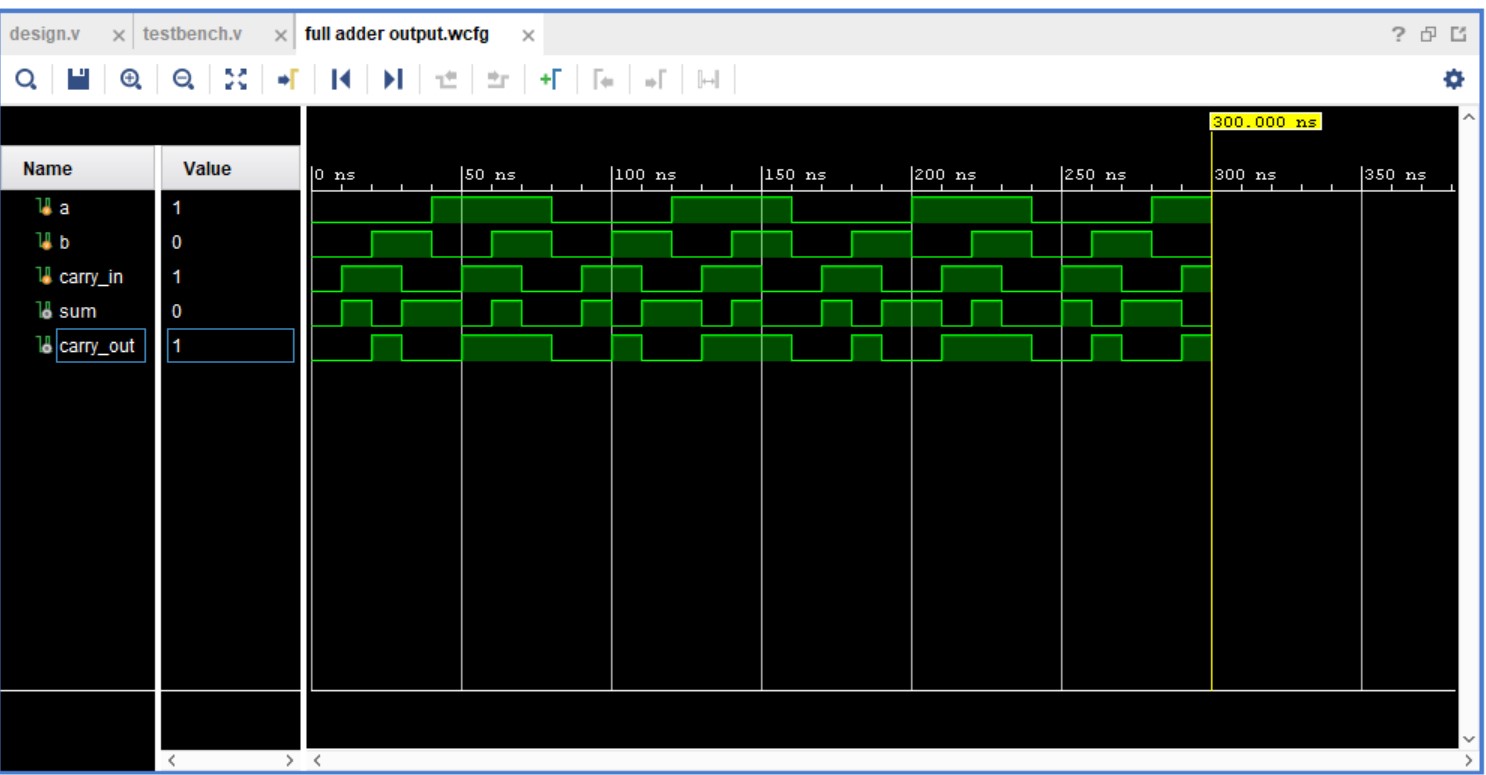
**CONCLUSION :**

Here are the output waveforms of the circuits we implemented on Xilinx vivado

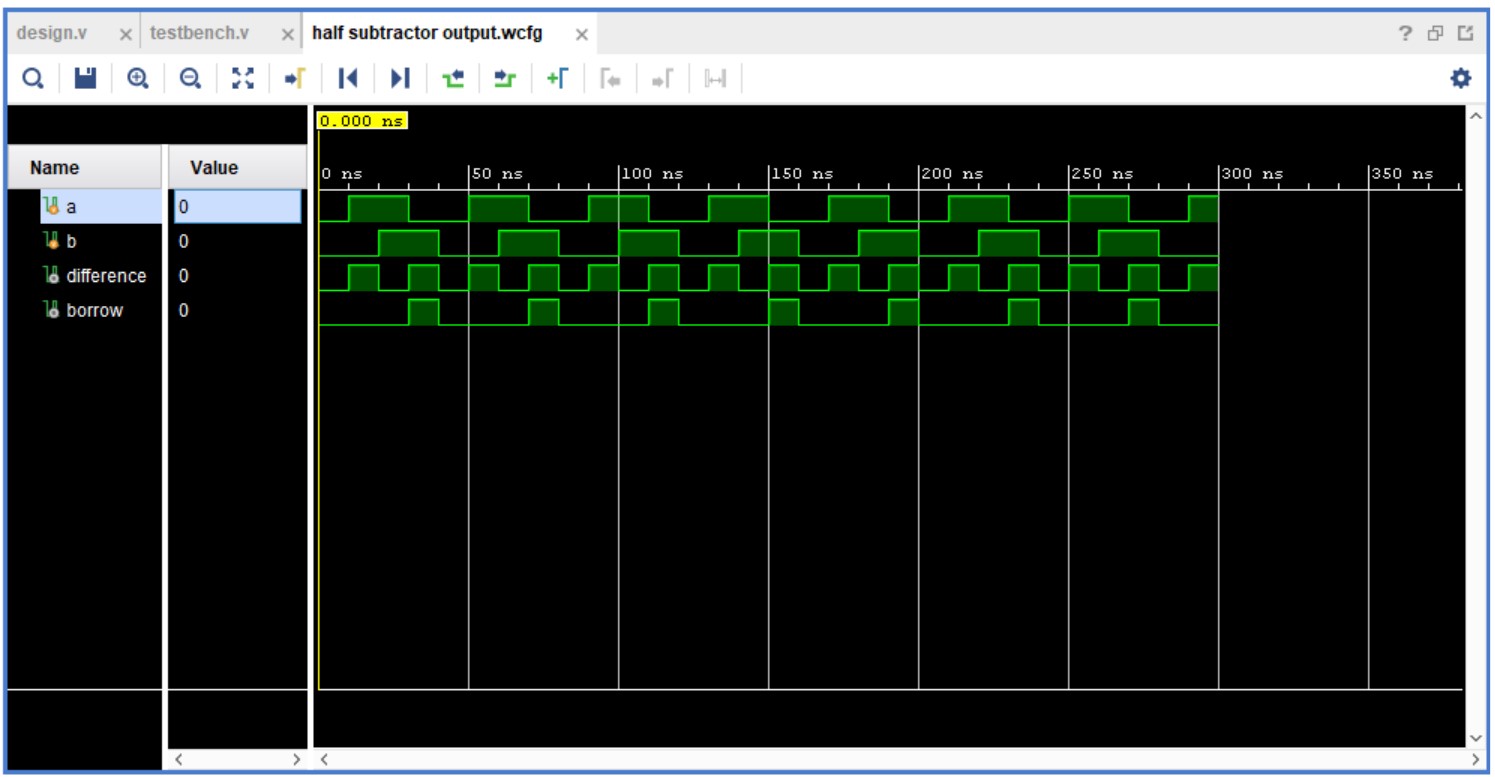
* HALF ADDER



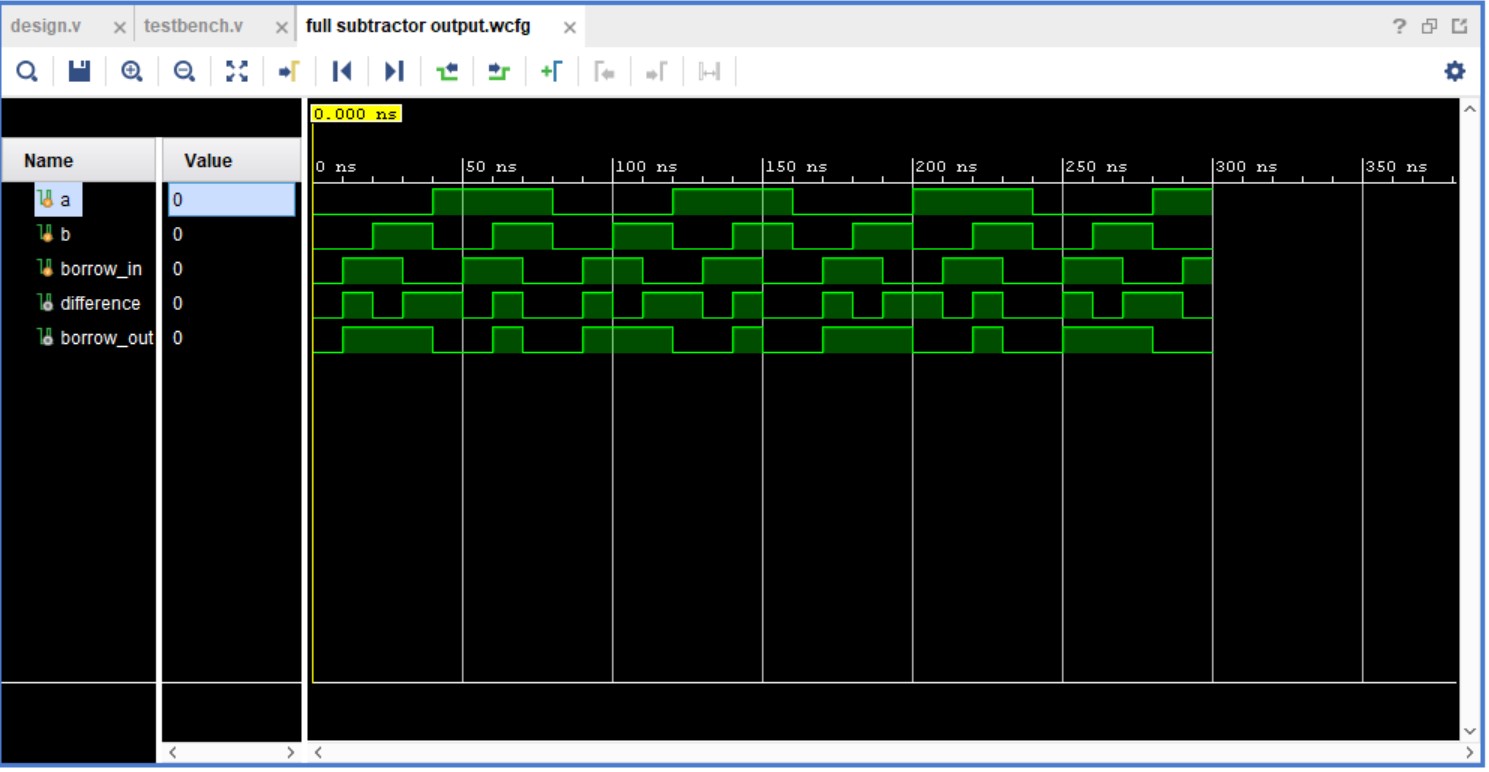
* FULL ADDER



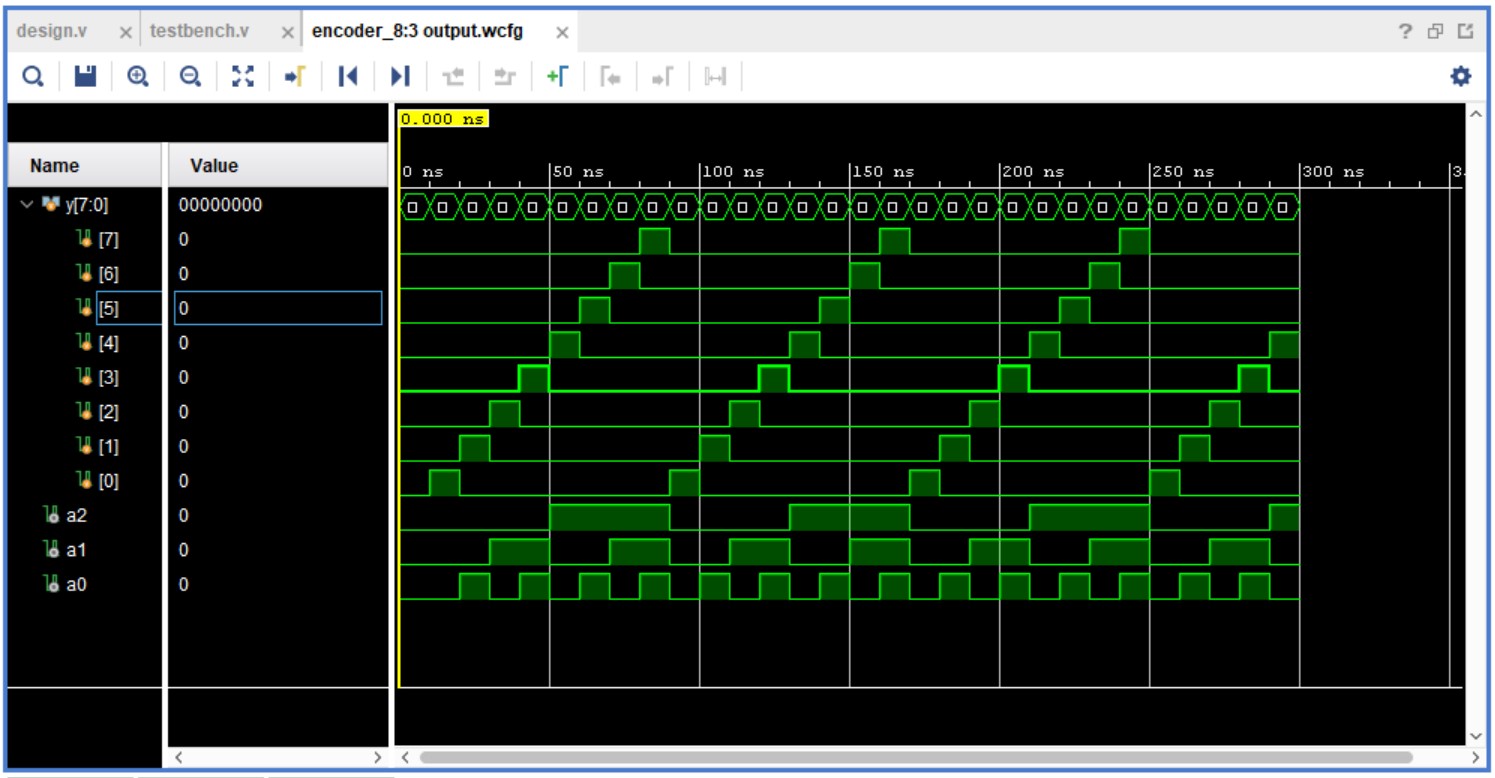
* HALF SUBTRACTER



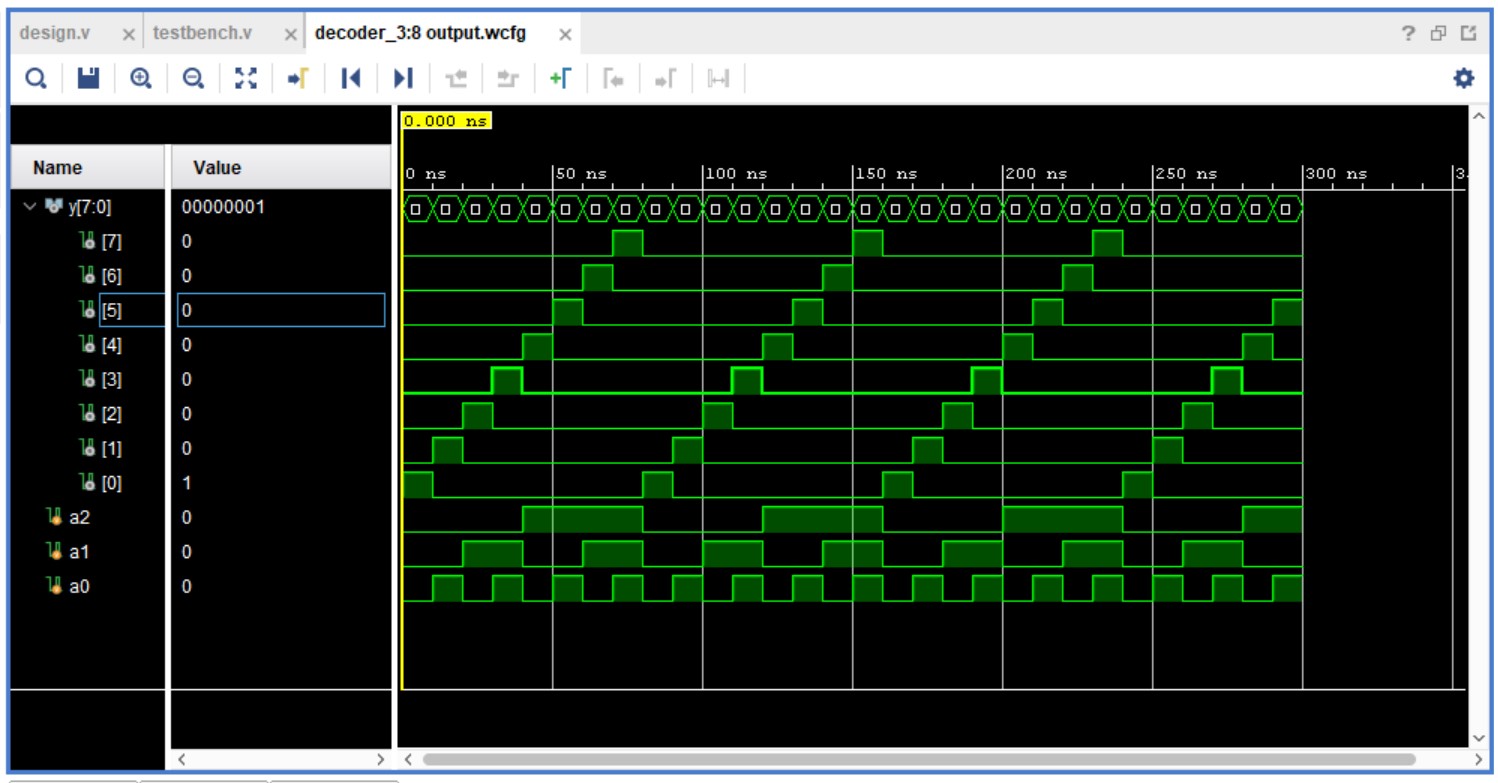
* FULL SUBTRACTER



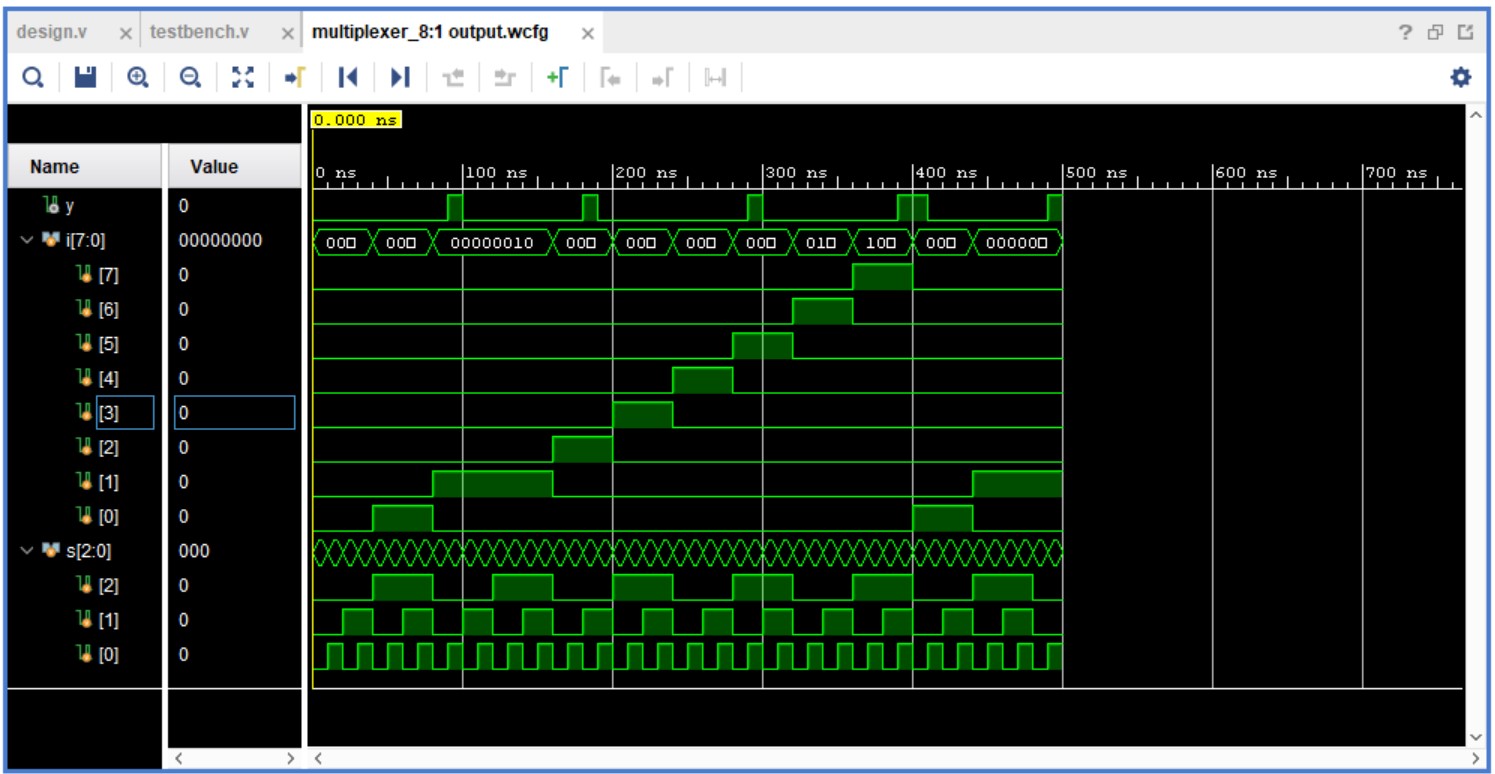
* ENCODER



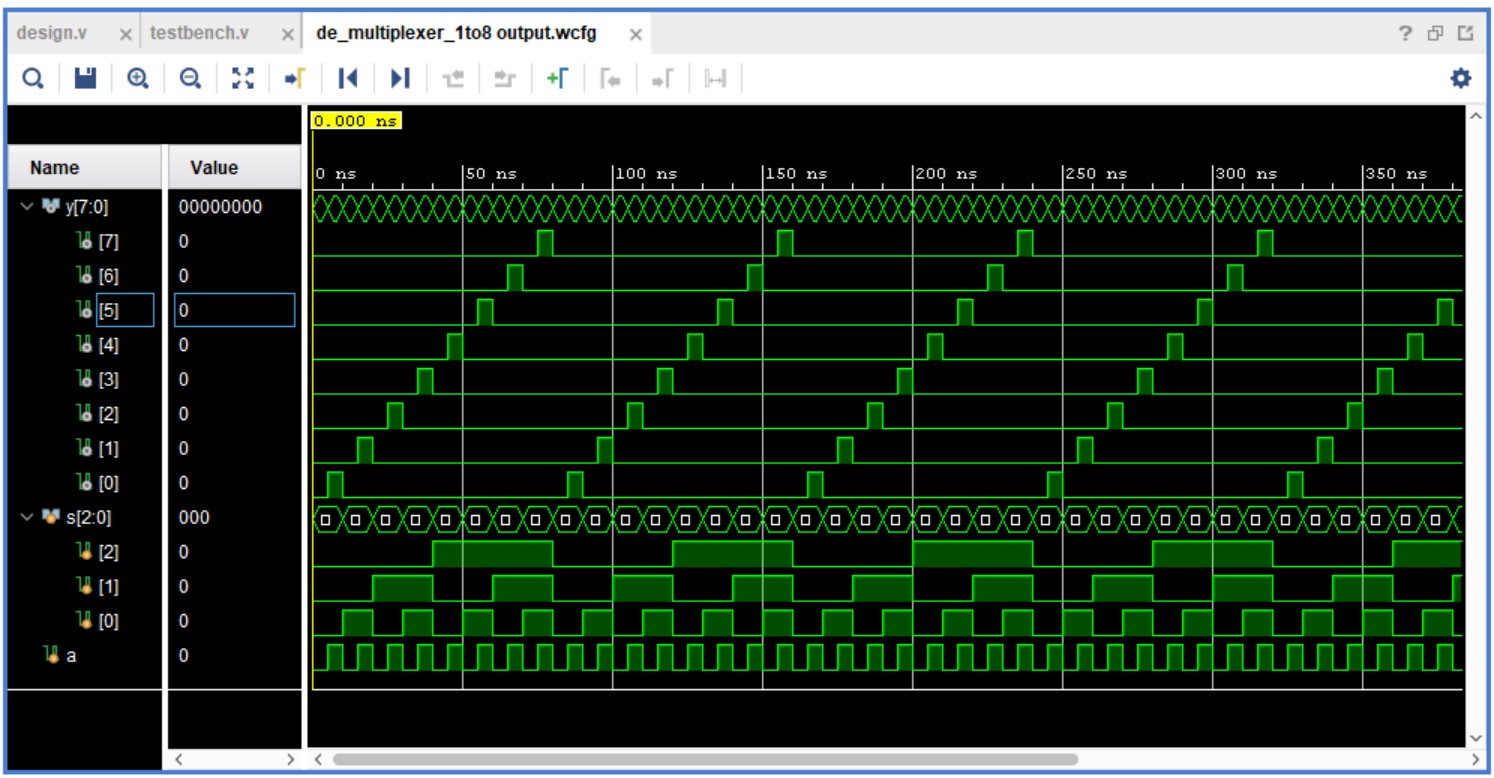
* DECODER



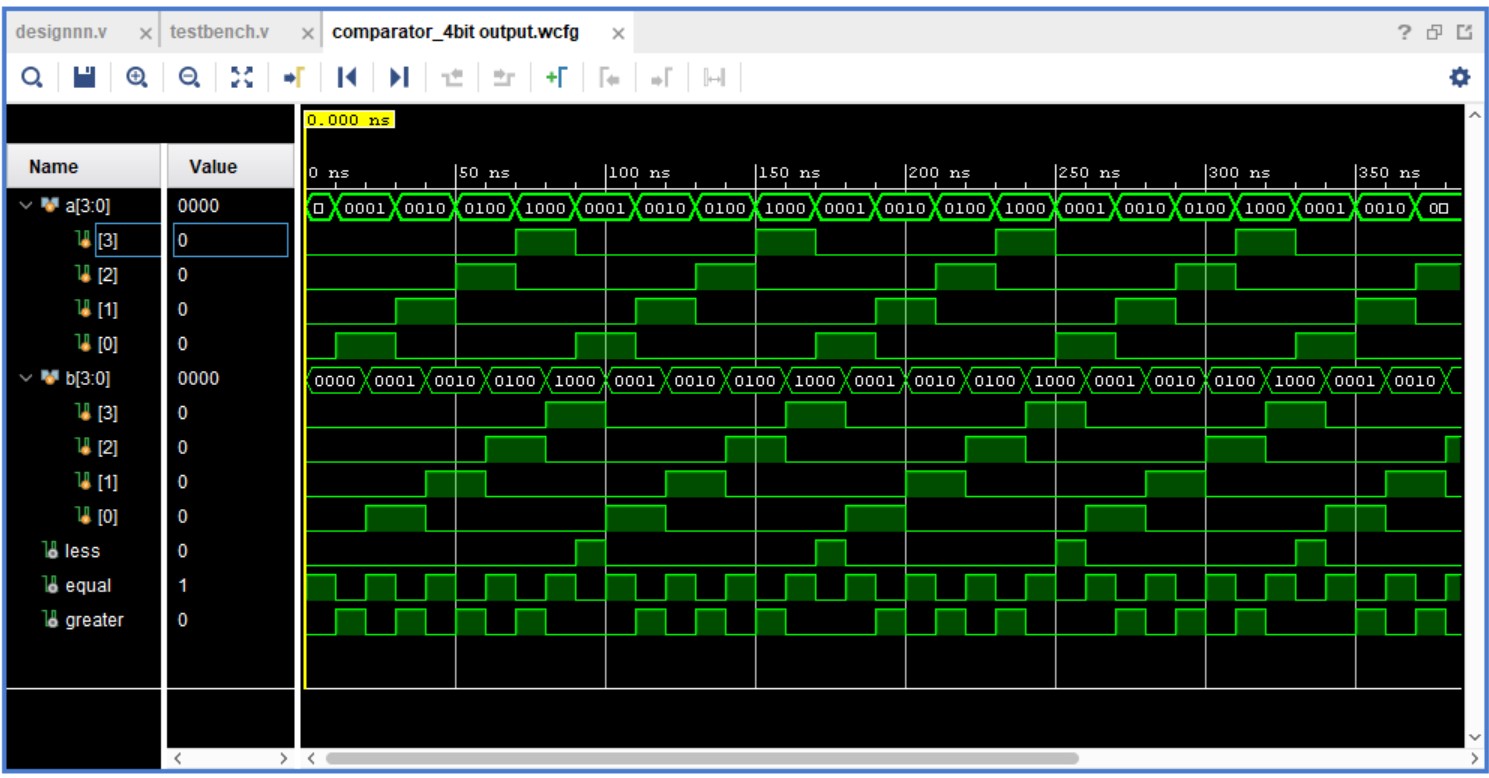
* MULTIPLEXER



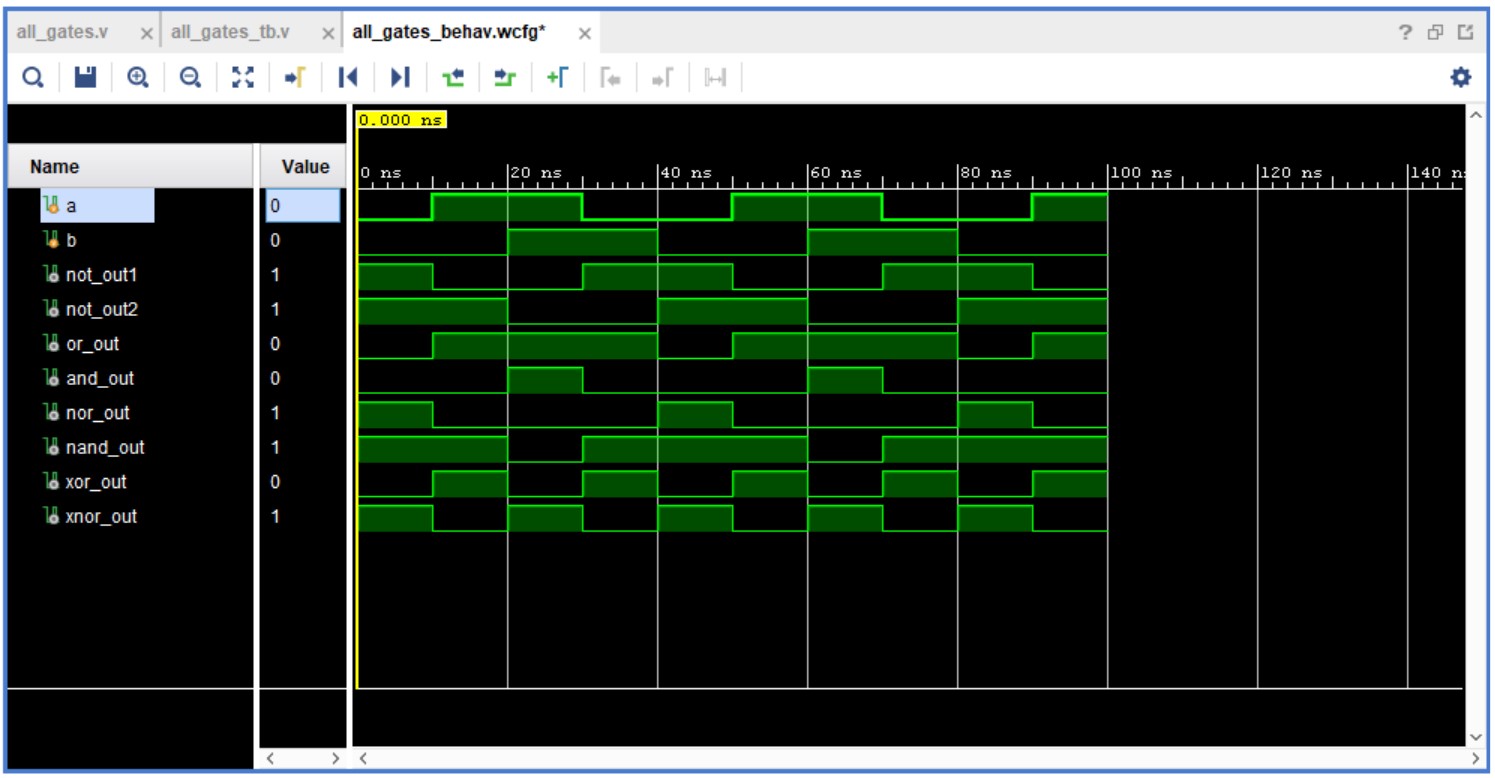
* DEMULTIPLEXER



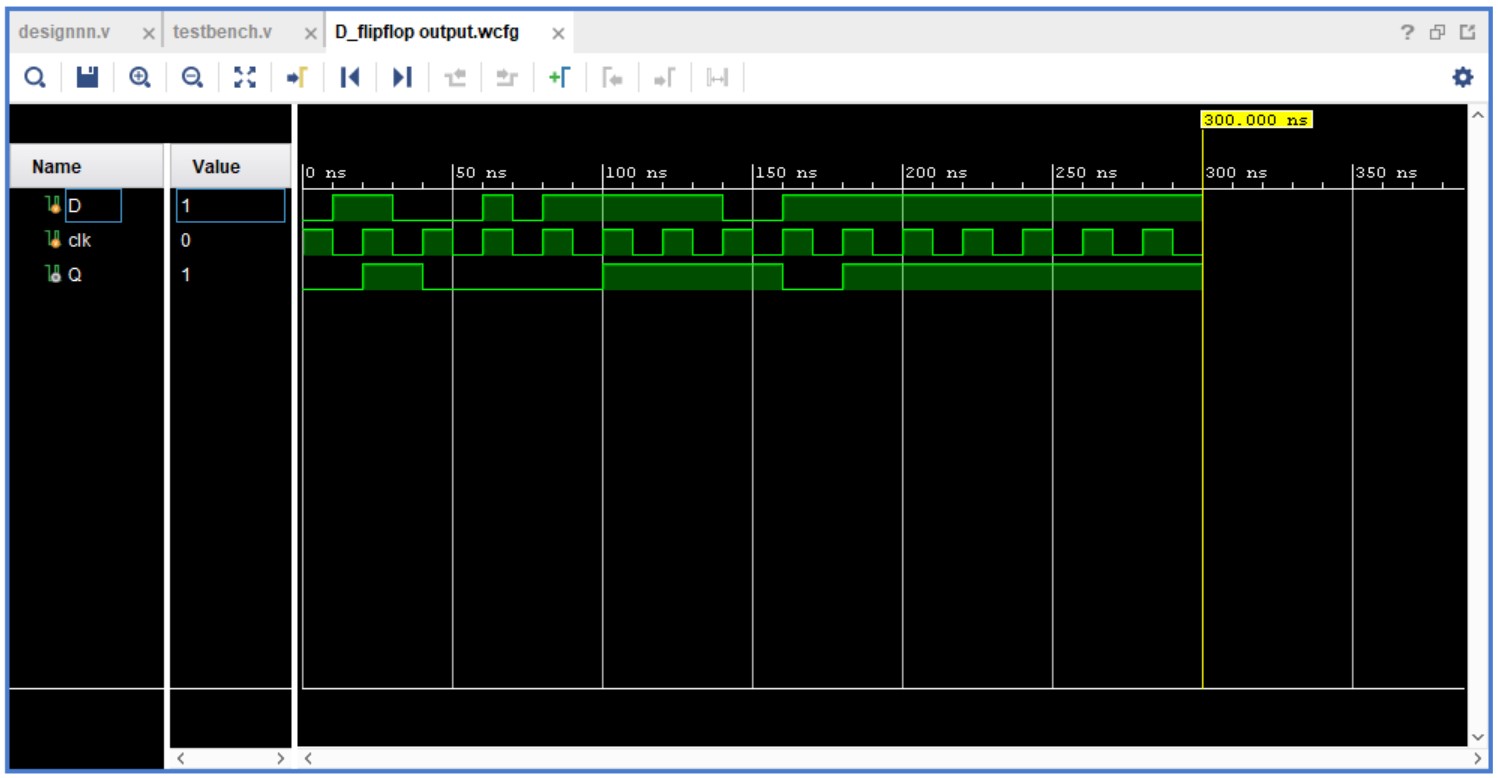
* COMPARATOR



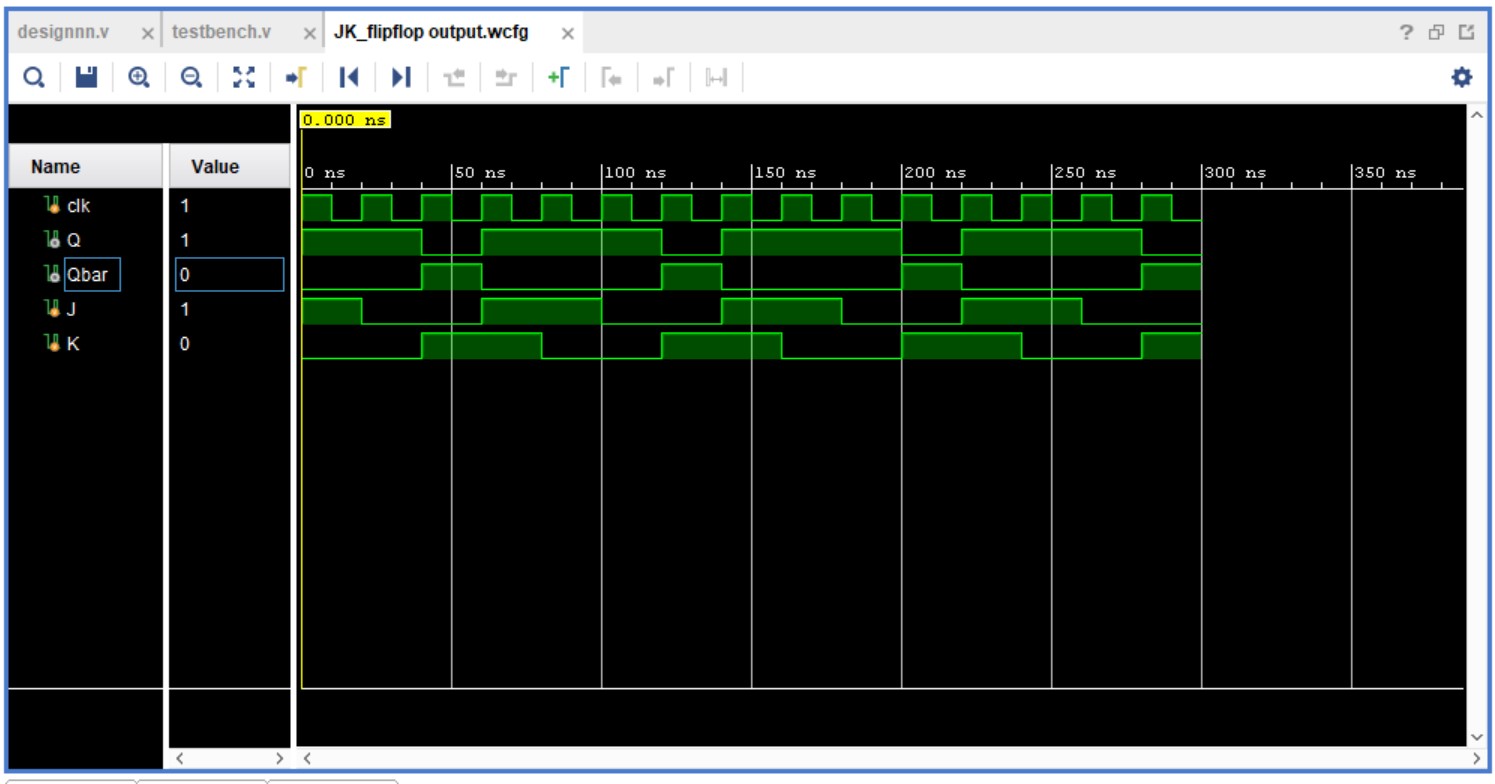
* ALL BASIC GATES



* D FLIP FLOP



* JK FLIP FLOP



THANK

YOU