More Verilog Examples

BBM233 Logic Design Lab - Fall 2020

Example of Behavioral Design
Verilog Sequential Circuits Lab Experiment 5
BBM233 Fall 2019

HACETTEPE UNIVERSITY
Computer Engineering Department
BBM233 Logic Design Laboratory
Fall 2019

Sequential Circuits in Verilog

AIM

In this experiment you will design a sequential circuit and implement it in Verilog HDL.

LAB EXPERIMENT

By following the steps discussed above, you will design a **Synchronous 3-Bit Binary Up/Down Counter** which counts either up or down depending on the counting mode.

- Counter outputs will be from 000 to 111 (from 0 to 7).
- The initial state of the counter is 000. There is a single input M (mode selecting signal) such that:
    - When M = 0, the counter counts down (once 0 is reached, it should start counting down from 7 again - cyclically),
    - When M = 1, the counter counts up (once 7 is reached, it should start counting up from 0 again - cyclically).

Experiment Steps:

1. Follow the given steps for designing sequential circuits starting with drawing the state transition diagram. For each step show your work clearly, as it is shown in these instructions with the example of sequence detector. Use D flip-flops in your implementation.

2. Once you have designed the circuit, write a Verilog code implementing the counter. Use behavioral design approach as shown in the alternative implementation. Write an appropriate testbench which clearly shows that your counter works properly.

3. Write a report that includes all design steps, final circuit design, Verilog codes, and proof of correct results (e.g. screenshots of waveform, variable changes from the console, etc.).

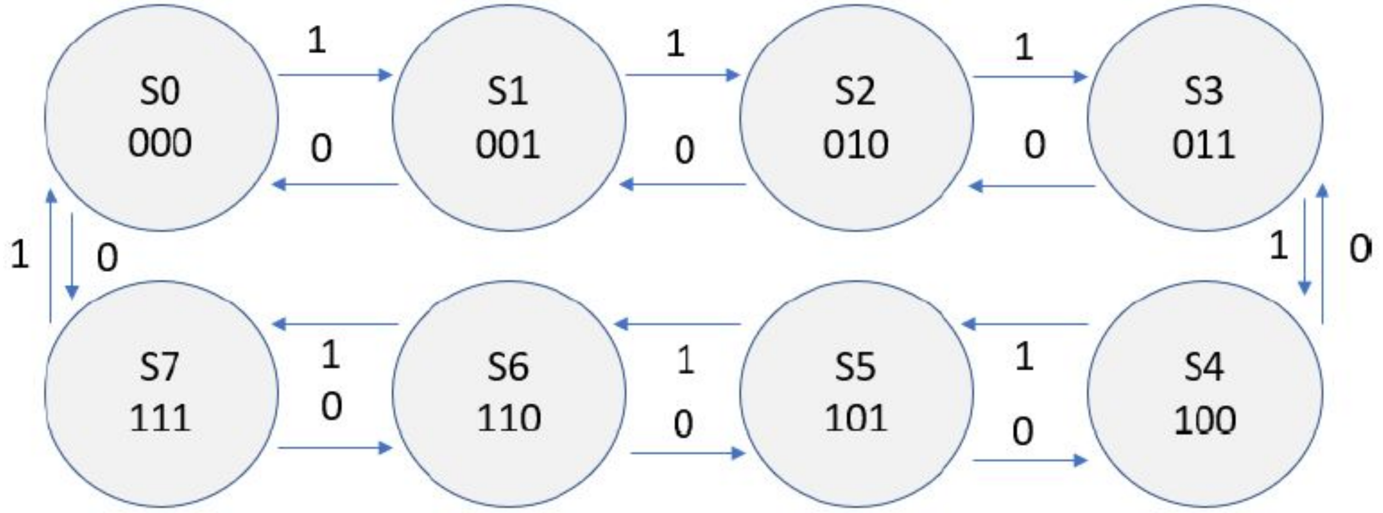
Report Submission:

The deadline for submission is Wednesday, 11.12.2019 at 13:00 for all sections. Zip your files (not .rar, only .zip files are supported by the system) and submit your work through https://submit.cs.hacettepe.edu.tr/index.php with the following file hierarchy:

```
- <studentID>.zip
   - counter.v
   - counter_testbench.v
   - report.pdf
```

A Student Solution - graded with 100

We created a state transition diagram for our three-bit counter. We need 8 states for the representation of the digits (0-7). If the input is one, we should go to next stage (counter counts up). If the input is zero we should go to previous stage. (counter counts down)



We convert our state transition diagram into a state transition table (binary coded state table).

- 3 bit input for previous stage
- One bit input for M (mode selecting signal).
- 3 bit output for next stage

Previous Stage			Input	Next Stage		
0	0	0	0	1	1	1
0	0	0	1	0	1	1
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	0	1	1
0	1	1	0	0	1	0
0	1	1	1	1	0	0
1	0	0	0	0	1	1
1	0	0	1	1	0	1
1	0	1	0	1	0	0
1	0	1	1	1	1	0
1	1	0	0	1	0	1
1	1	0	1	1	1	1
1	1	1	0	1	1	0
1	1	1	1	0	0	0

We choose D-type flip-flops. Since there are eight states, we need three flip-flops, And we label their outputs as A,B and C. The characteristic equation of the D flip-flop is Q(t+1) = D ,which means that the next state values in the state table specify the D input condition for the flip-flop. From the state transition table we obtain the following input equations:

AB\CM	**00**	**01**	**11**	**10**
00	1	0	0	0
01	0	0	1	0
11	1	1	0	1
10	0	1	1	1

DA= A'B'C'M'+A'BCM+ABC'+AB'M+ACM'

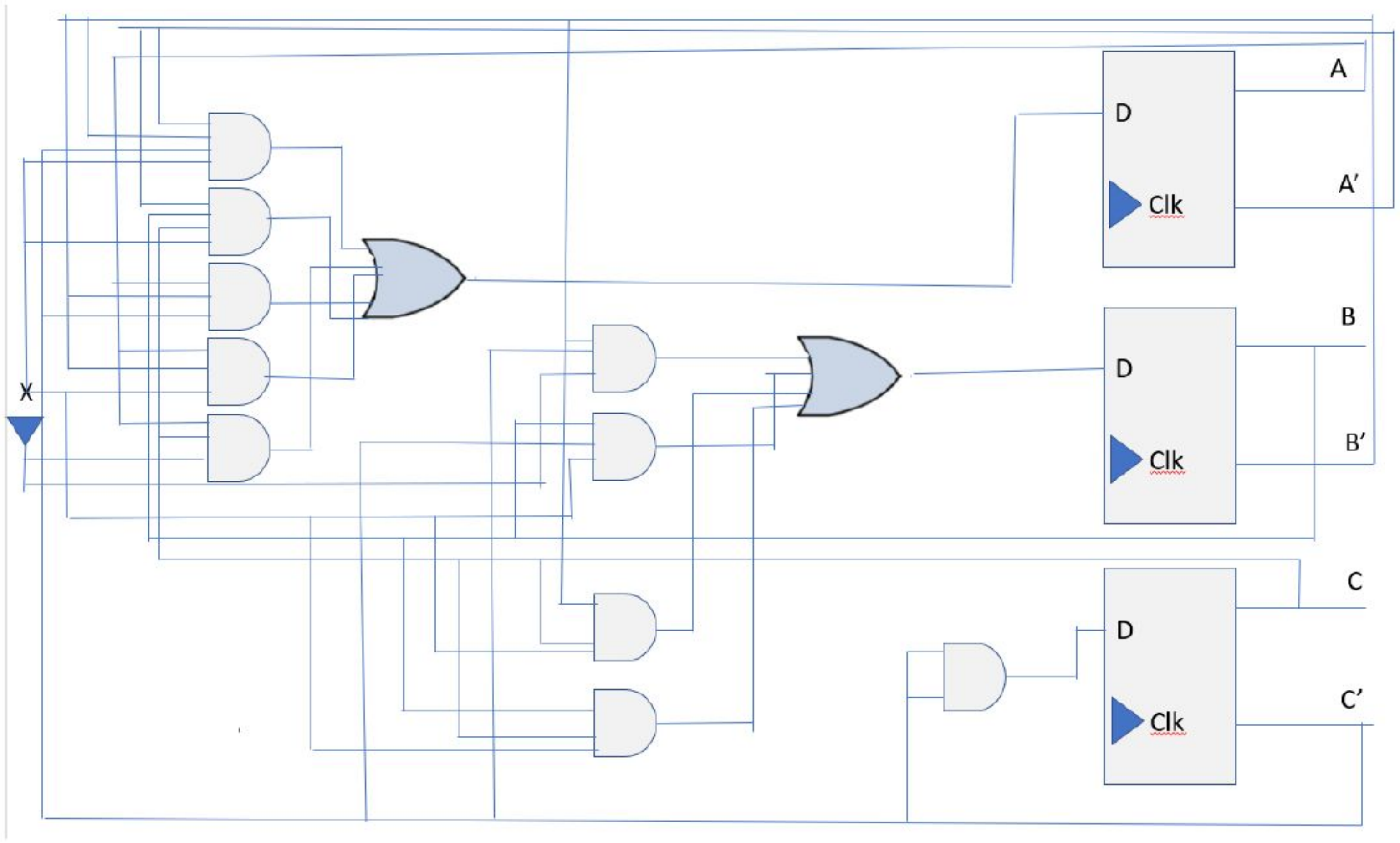
AB\CM	**00**	**01**	**11**	**10**
00	1	0	1	0
01	0	1	0	1
11	0	1	0	1
10	1	0	1	0

DB= B'C'M'+BC'M+B'CM+BCM'

AB\CM	**00**	**01**	**11**	**10**
00	1	1	0	0
01	1	1	0	0
11	1	1	0	0
10	1	1	0	0

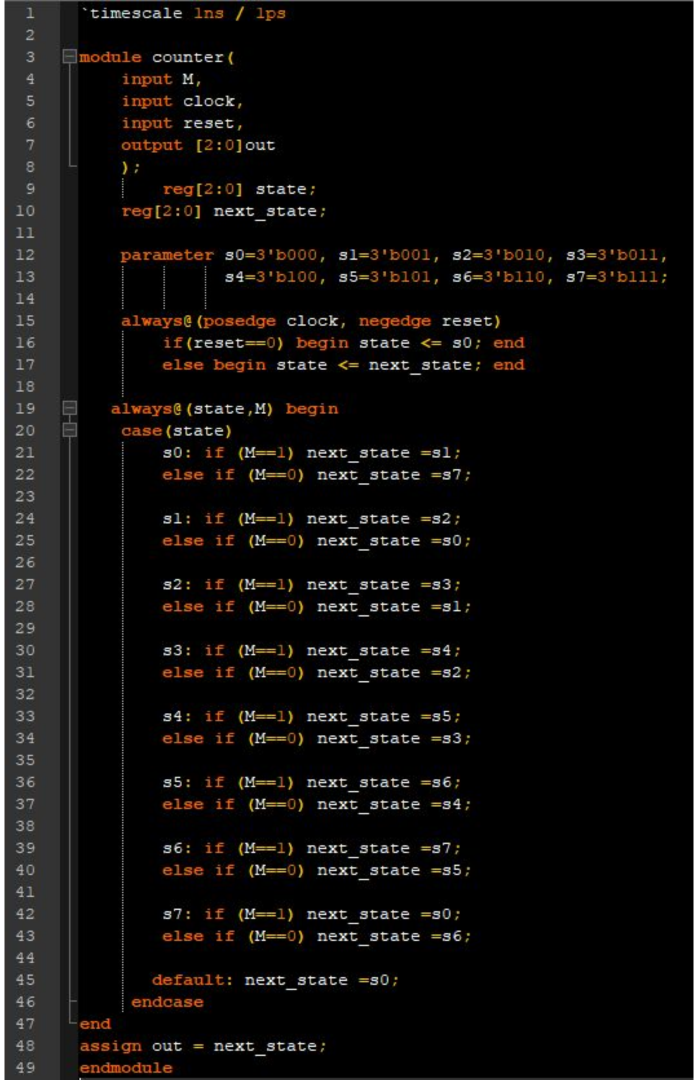
DC= C'

We used simplified functions to design our sequential circuit.



Behavioral design for our three-bit counter in Verilog:

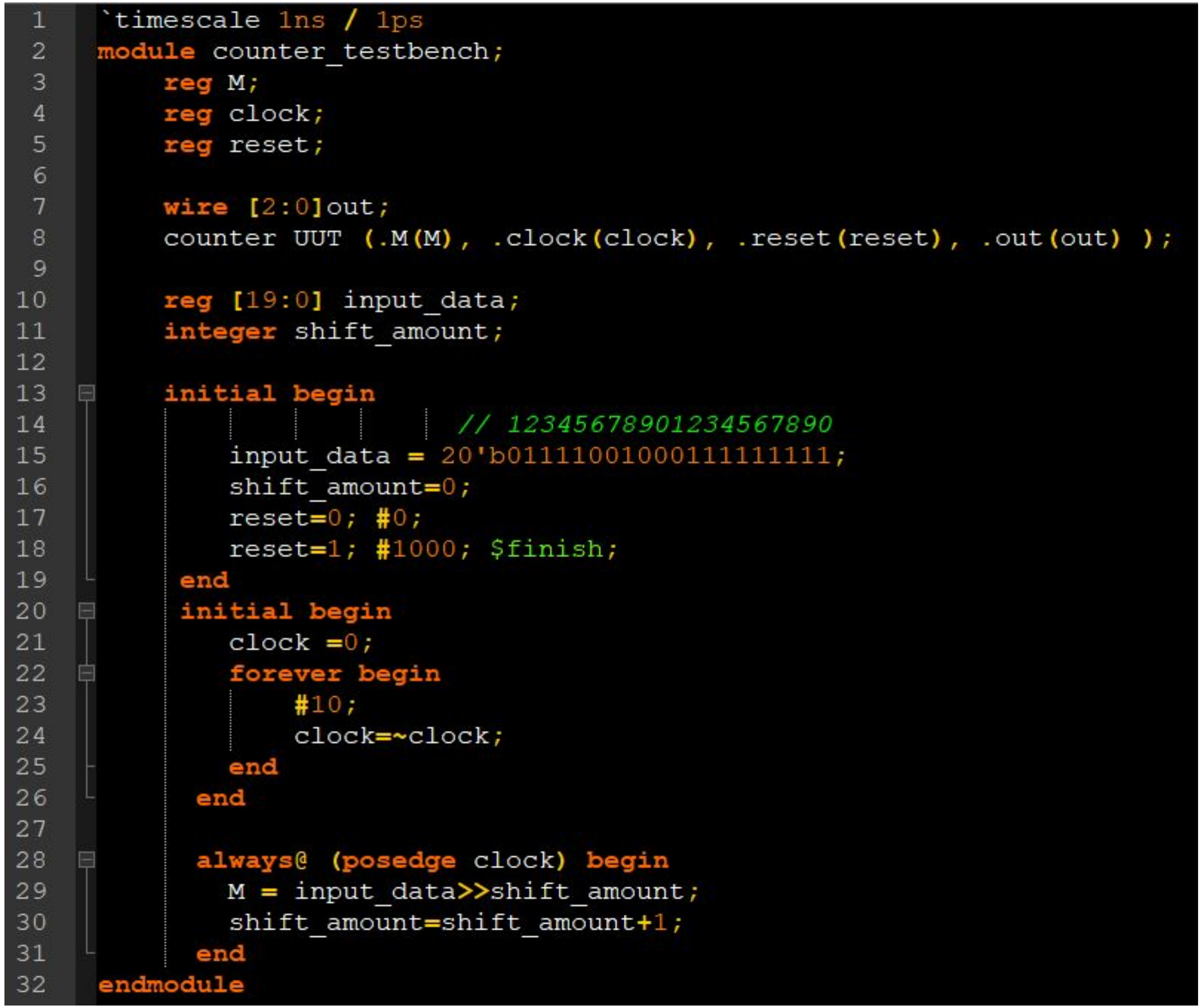
Firstly, we identify our inputs (M, clock, reset) and 3’bit output out. Then we made two register for our state and next state information. We use parameter to determine state’s values. In first always block, we select what we do. If the reset is zero we chose state0 as a initial state. Otherwise we assign next state to our current state. In second always block we have a case block. We use if else to determine which state we will go next. Finally, we assign our output to next state.



```
`timescale 1ns / 1ps

module counter(
    input M,
    input clock,
    input reset,
    output [2:0]out
    );
        reg[2:0] state;
    reg[2:0] next_state;

    parameter s0=3'b000, s1=3'b001, s2=3'b010, s3=3'b011,
                  s4=3'b100, s5=3'b101, s6=3'b110, s7=3'b111;

    always@(posedge clock, negedge reset)
        if(reset==0) begin state <= s0; end
        else begin state <= next_state; end

   always@(state,M) begin
    case(state)
        s0: if (M==1) next_state =s1;
        else if (M==0) next_state =s7;

        s1: if (M==1) next_state =s2;
        else if (M==0) next_state =s0;

        s2: if (M==1) next_state =s3;
        else if (M==0) next_state =s1;

        s3: if (M==1) next_state =s4;
        else if (M==0) next_state =s2;

        s4: if (M==1) next_state =s5;
        else if (M==0) next_state =s3;

        s5: if (M==1) next_state =s6;
        else if (M==0) next_state =s4;

        s6: if (M==1) next_state =s7;
        else if (M==0) next_state =s5;

        s7: if (M==1) next_state =s0;
        else if (M==0) next_state =s6;

      default: next_state =s0;
     endcase
end
assign out = next_state;
endmodule
```

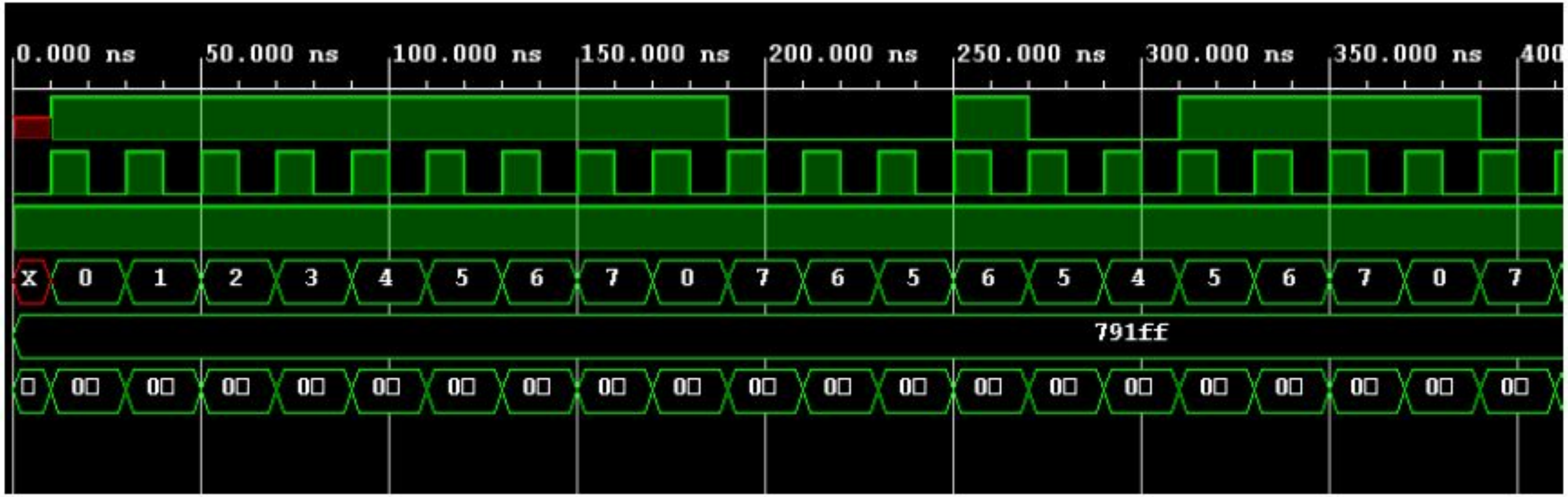
Our testbench:

In out testbench we have M clock and reset as a register. And three-bit output as a wire. We use UUT (Unit under test). Then we gave initial values to our inputs To control the counter we write an “input_data” which will give 1 or 0 to M input value. While giving the values, we use shifting.

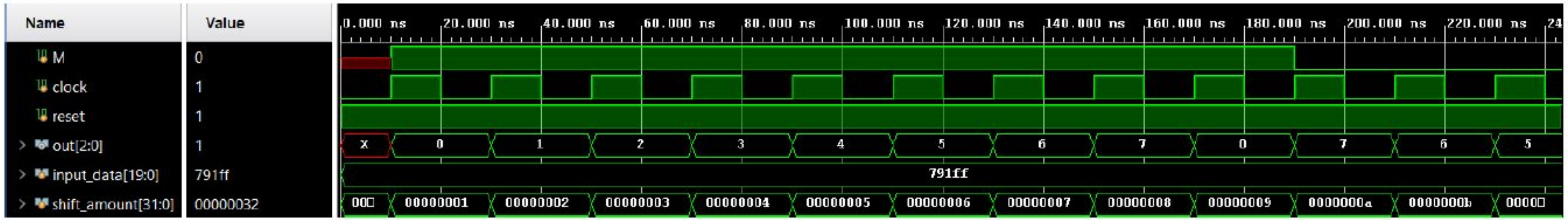


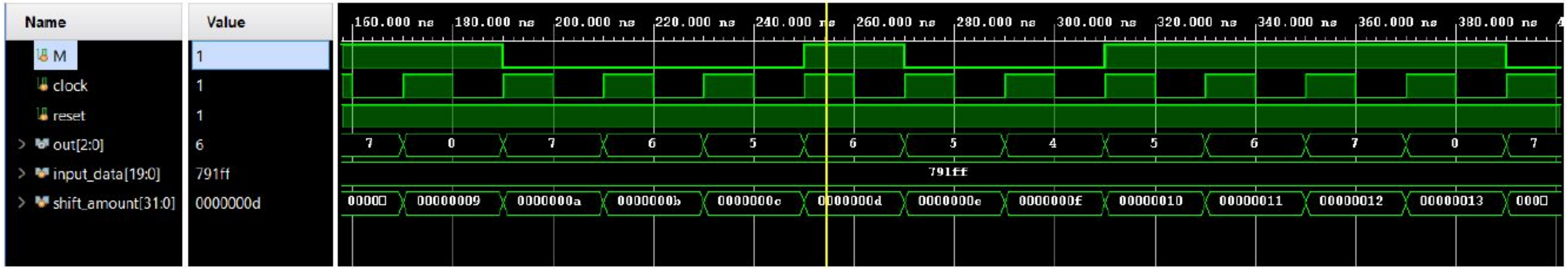
```
`timescale 1ns / 1ps
module counter_testbench;
    reg M;
    reg clock;
    reg reset;

    wire [2:0]out;
    counter UUT (.M(M), .clock(clock), .reset(reset), .out(out) );

    reg [19:0] input_data;
    integer shift_amount;

    initial begin
                //  12345678901234567890
        input_data = 20'b01111001000111111111;
        shift_amount=0;
        reset=0; #0;
        reset=1; #1000; $finish;
     end
     initial begin
        clock =0;
        forever begin
            #10;
            clock=~clock;
        end
      end

      always@ (posedge clock) begin
        M = input_data>>shift_amount;
        shift_amount=shift_amount+1;
      end
endmodule
```

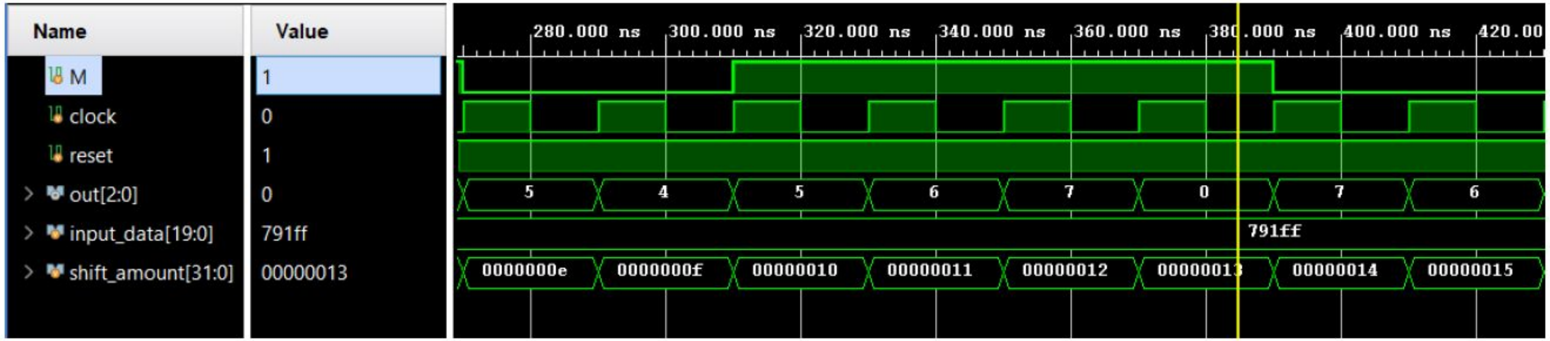
To start from state 0 we gave our reset 0 in the 0. Nanosecond. Then the reset fix to 1. Then we create our clock. In the last always we get the next input.



We can see in the simulator that when the input M becomes 1 it counts up. When we changed the input value as a 0 it starts the count down. The counter is cyclic. We can see in the simulator that after 7 0 is comes.


Name
Value
M 0
clock 1
reset 1
out[2:0] 1
input_data[19:0] 791ff
shift_amount[31:0] 00000032
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000 ns
180.000 ns
200.000 ns
220.000 ns
X 0 1 2 3 4 5 6 7 0 7 6 5
791ff
000 00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000a 0000000b 0000


Name
Value
M 1
clock 1
reset 1
out[2:0] 6
input_data[19:0] 791ff
shift_amount[31:0] 0000000d
160.000 ns
180.000 ns
200.000 ns
220.000 ns
240.000 ns
260.000 ns
280.000 ns
300.000 ns
320.000 ns
340.000 ns
360.000 ns
380.000 ns
7 0 7 6 5 6 5 4 5 6 7 0 7
791ff
0000 00000009 0000000a 0000000b 0000000c 0000000d 0000000e 0000000f 00000010 00000011 00000012 00000013 0000


Name
Value
M 1
clock 0
reset 1
out[2:0] 0
input_data[19:0] 791ff
shift_amount[31:0] 00000013
280.000 ns
300.000 ns
320.000 ns
340.000 ns
360.000 ns
380.000 ns
400.000 ns
420.00
5 4 5 6 7 0 7 6
791ff
0000000e 0000000f 00000010 00000011 00000012 00000013 00000014 00000015

Example of Structural Design
Verilog Sequential Circuits Lab Experiment 6
BBM233 Fall 2019

HACETTEPE UNIVERSITY
Computer Engineering Department
BBM233 Logic Design Laboratory
Fall 2019

Verilog Sequential Circuits Lab Experiment

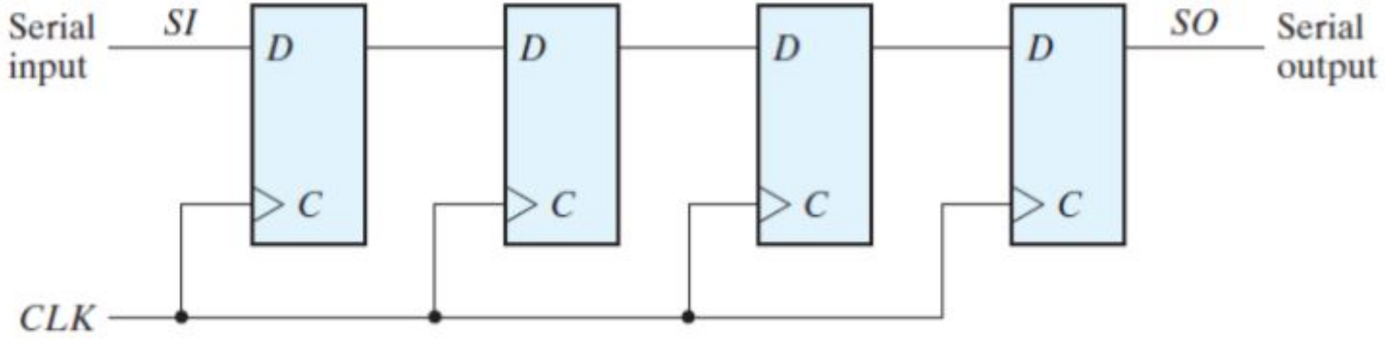
Registers

AIM

In this experiment you will design a serial adder and implement it in Verilog HDL.

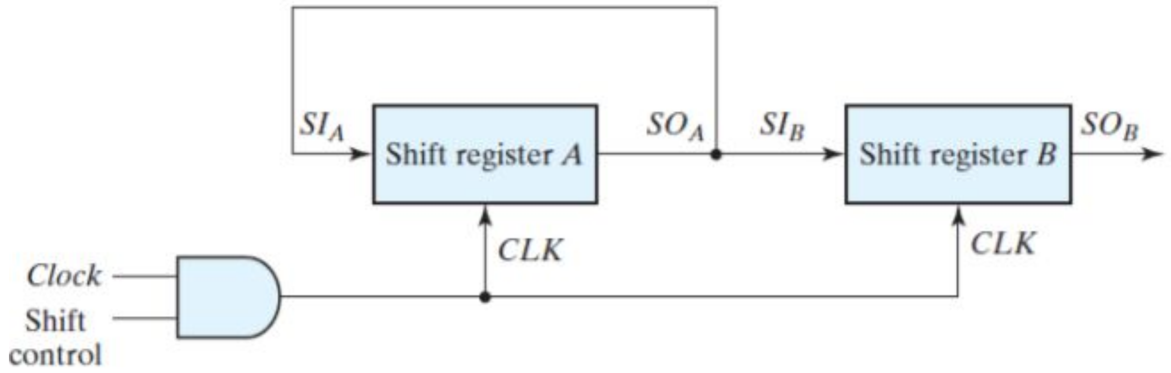
BACKGROUND

Shift registers are registers that are capable of shifting binary information held in each cell to its neighboring cell, either to the left or to the right. The simplest possible shift register is one that uses only flip-flops. A 4-bit unidirectional (left-to-right) shift register is shown in the figure below.

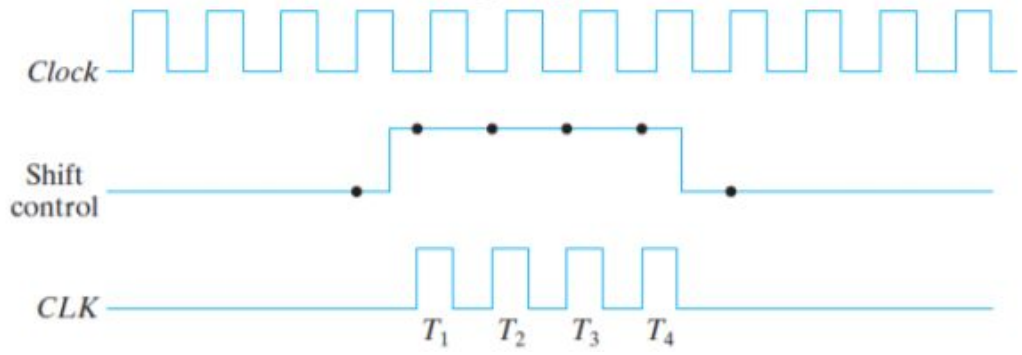


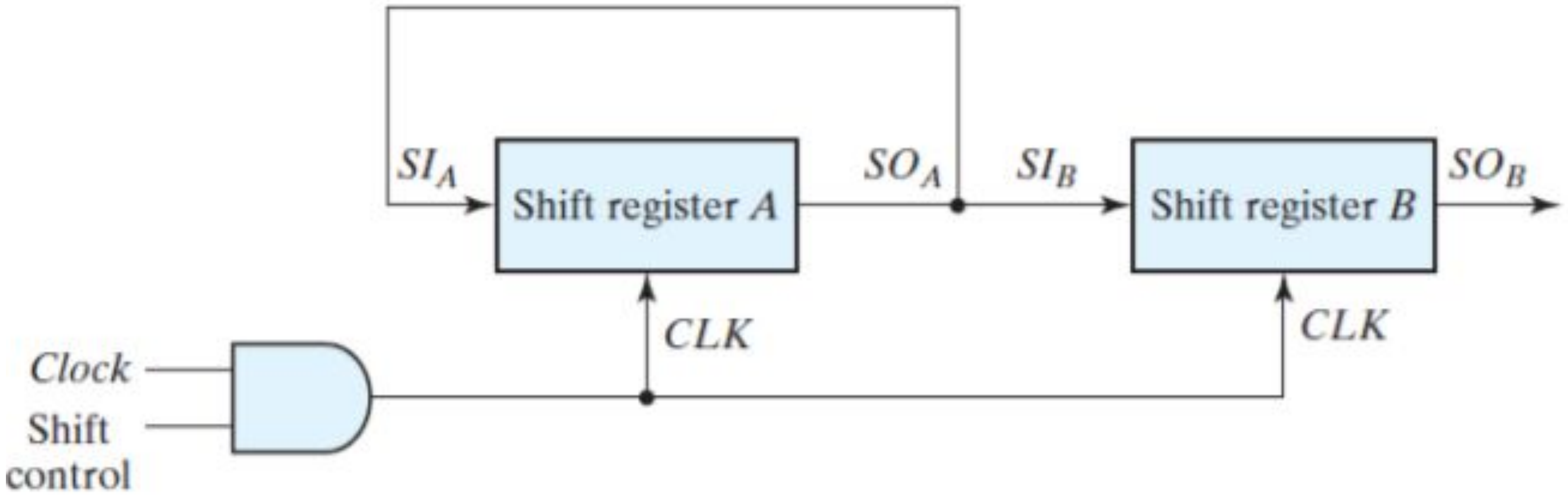
An example application would be serial transfer. A digital system is said to operate in a serial mode when information is manipulated one bit at a time. Computer may operate in a serial mode, a parallel mode, or a combination of both. Serial operations are slower, because information is manipulated one bit at a time. However, serial computers require less hardware, because one common circuit can be used over and over again to manipulate the bits coming out of shift registers.

In serial transfer, information is transferred one bit at a time by shifting the bits out of a source register into a destination register. The serial transfer of information from register A to register B is done with shift registers, as shown in figure below.

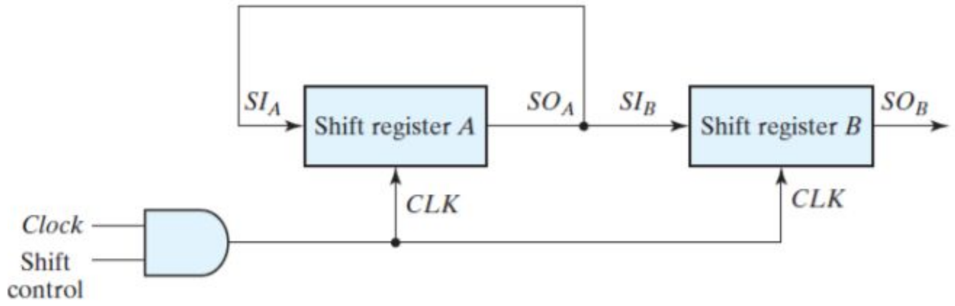


Timing Diagram:

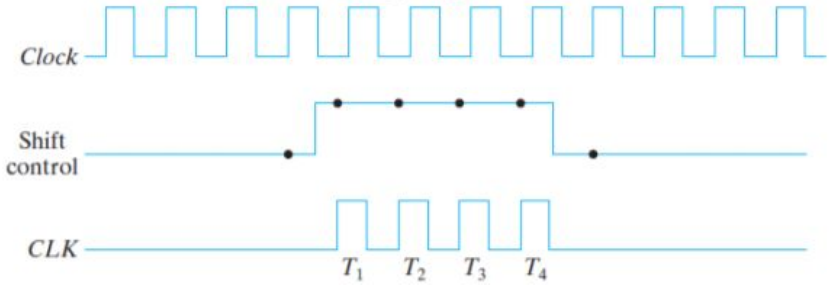




The serial output (SO) of register A is connected to the serial input (SI) of register B. To prevent the loss of information stored in the source register, the information in register A is made to circulate by connecting the serial output to its serial input. The initial content of register B is shifted out through its serial output and is lost unless it is transferred to a third shift register.



Timing Diagram:



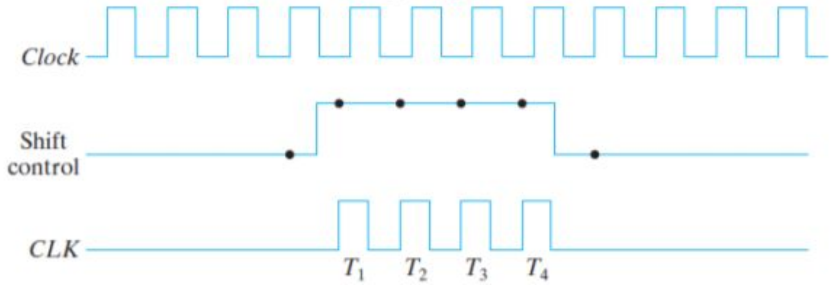
Let's assume that the shift registers are 4 bit wide. The control unit that supervises the transfer of data must be designed in such a way that it enables the shift registers, through the shift control signal, for a fixed time of four clock pulses in order to pass an entire word. When the shift control signal is active, the output of the AND gate connected to the CLK inputs produces four pulses: T1, T2, T3, and T4. Each rising edge of the pulse causes a shift in both registers. After the fourth pulse, the shift control is changed to 0, and the shift registers are disabled.

Assume that the binary content of A before the shift is 1011 and that of B is 0010. The serial transfer from A to B occurs in four steps, as shown in the table below.

Serial-Transfer Example

Timing Pulse	Shift Register A	Shift Register B
Initial value	1 0 1 1	0 0 1 0
After T_1	1 1 0 1	1 0 0 1
After T_2	1 1 1 0	1 1 0 0
After T_3	0 1 1 1	0 1 1 0
After T_4	1 0 1 1	1 0 1 1

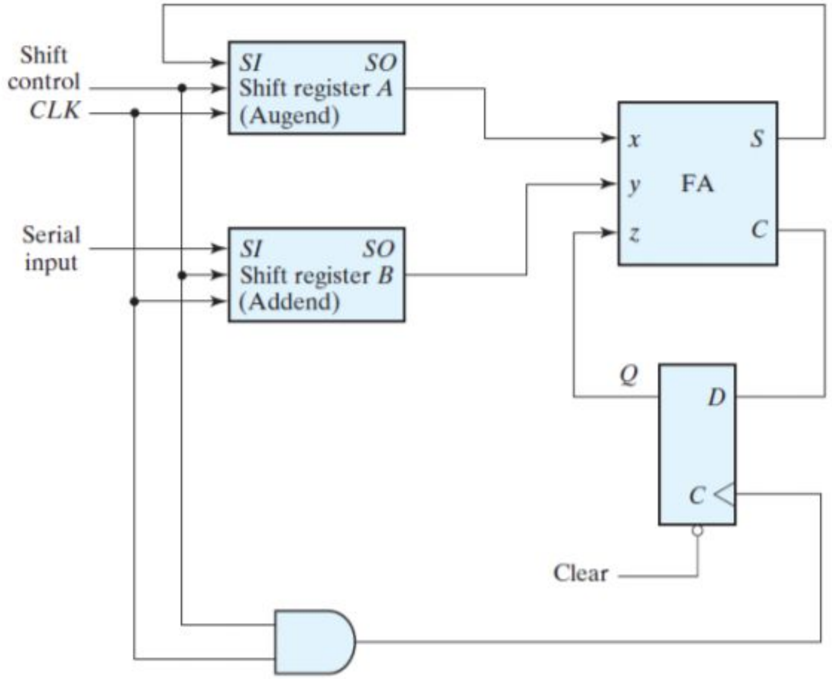
Timing Diagram:



LAB EXPERIMENT

You will implement a Serial Adder in Verilog using structural design approach.

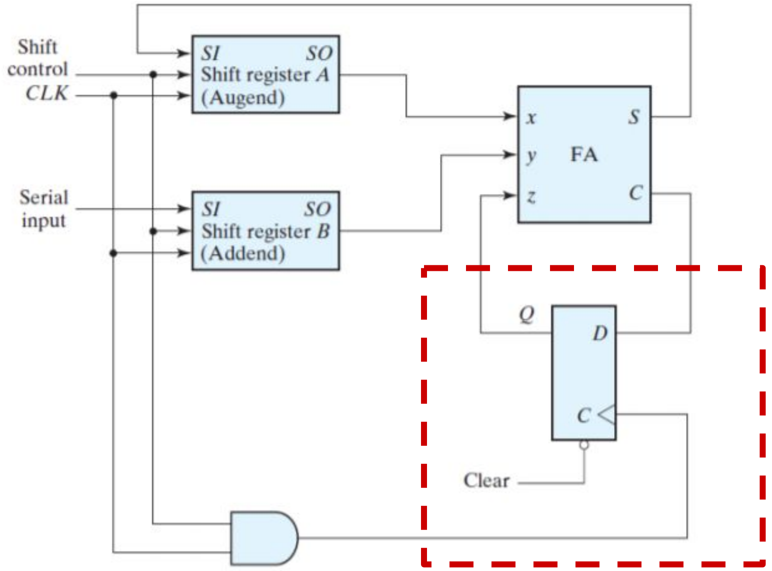
The two binary numbers to be added serially are stored in two shift registers. Beginning with the least significant pair of bits, the circuit adds one pair at a time through a single full-adder (FA) circuit, as shown in the figure below.



The carry out of the full adder is transferred to a D flip-flop, the output of which is then used as the carry input for the next pair of significant bits. The sum bit from the S output of the full adder could be transferred into a third shift register. However, by shifting the sum into A while the bits of A are shifted out, it is possible to use one register for storing both the augend and the sum bits.

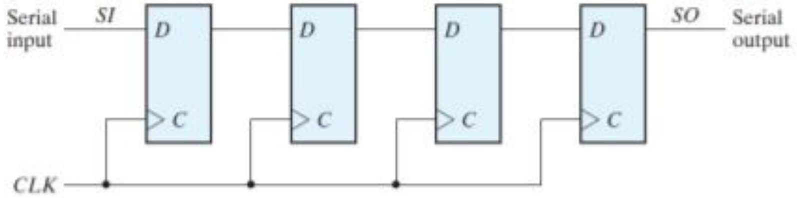
The operation of the serial adder is as follows: Initially, register A holds the augend, register B holds the addend, and the carry flip-flop is cleared to 0. The outputs (SO) of A and B provide a pair of significant bits for the full adder at x and y. Output Q of the flip-flop provides the input carry at z. The shift control enables both registers and the carry flip-flop, so at the next clock pulse, both registers are shifted once to the right, the sum bit from S enters the leftmost flip-flop of A, and the output carry is transferred into flip-flop Q. The shift control enables the registers for a number of clock pulses equal to the number of bits in the registers. For each succeeding clock pulse, a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right. This process continues until the shift control is disabled. Thus, the addition is accomplished by passing each pair of bits together with the previous carry through a single full-adder circuit and transferring the sum, one bit at a time, into register A.

Note that, unlike parallel adders (e.g. Ripple-Carry Adder) which require the number of full-adder circuits to be the same as the number of bits in the binary numbers (a 4-bit ripple-carry adder requires four full adder circuits), a serial adder requires only one full-adder circuit and a carry flip-flop. Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit which consists of a full adder and a flip-flop that stores the output carry. This design is typical in serial operations because the result of a bit-time operation may depend not only on the present inputs, but also on previous inputs that must be stored in flip-flops.



Experiment Steps:

1. Implement a D flip-flop in Verilog.
2. Implement a Full Adder in Verilog. You may use any design approach you wish.
3. Implement a 4-bit Shift Register in Verilog using D flip-flip module and a structural design approach based on the figure below.



4. Use the figure of a Serial Adder circuit given above to implement a 4-bit serial adder in Verilog using structural design approach. Use D flip-flop, full adder, and shift register modules you implemented in the previous steps as necessary.
5. Write testbenches that test your Serial Adder module for the following initial states of the shift registers A and B:
    a. Shift-register-A = 4'b0001, Shift-register-B = 4'b1110
    b. Shift-register-A = 4'b0101, Shift-register-B = 4'b0011
    c. Shift-register-A = 4'b1111, Shift-register-B = 4'b0001
    d. Shift-register-A = 4'b1111, Shift-register-B = 4'b1111
6. Write a report that explains your design steps in detail, Verilog codes, and proof of correct results (e.g. screenshots of waveforms, variable changes from the console, etc.).

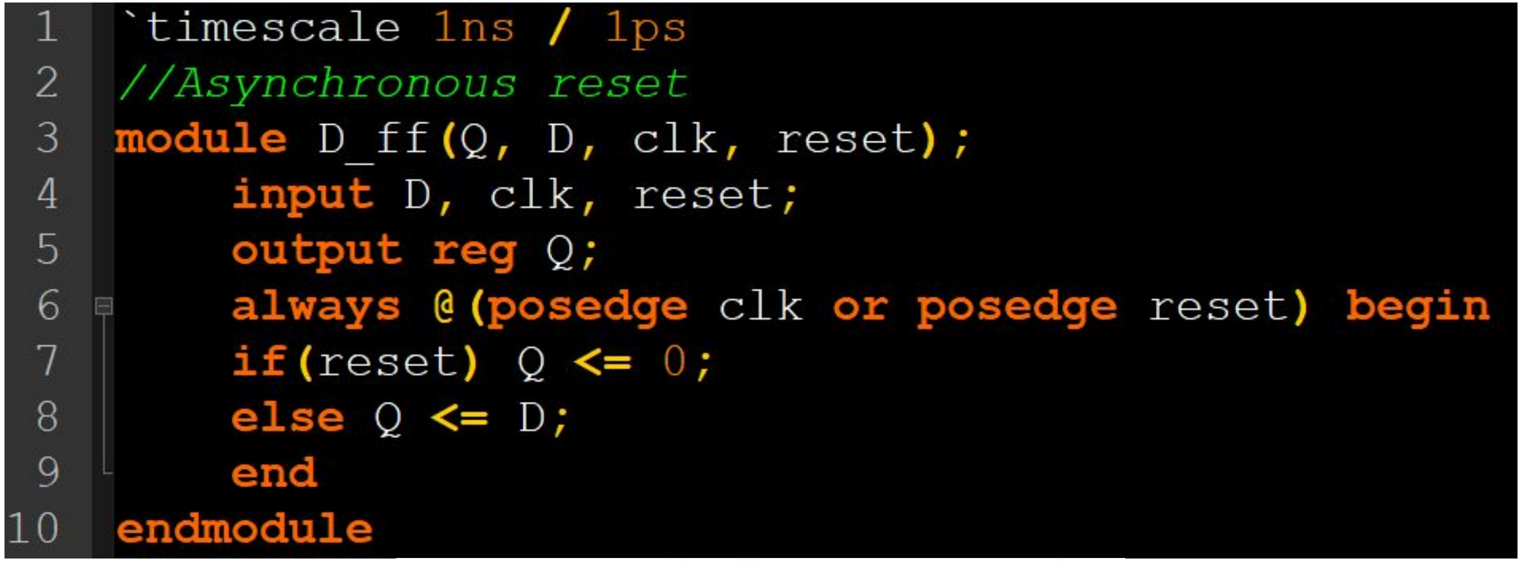
Report Submission:

The deadline for submission is Friday, 03.01.2020 at 22:00 for all sections. Zip your files (not .rar, only .zip files are supported by the system) and submit your work through https://submit.cs.hacettepe.edu.tr/index.php with the following file hierarchy:

```
- <studentID>.zip
  - D_ff.v
  - full_adder.v
  - shift_register.v
  - serial_adder.v
  - serial_adder_tb.v
  - report.pdf
```

A Student Solution - graded with 100

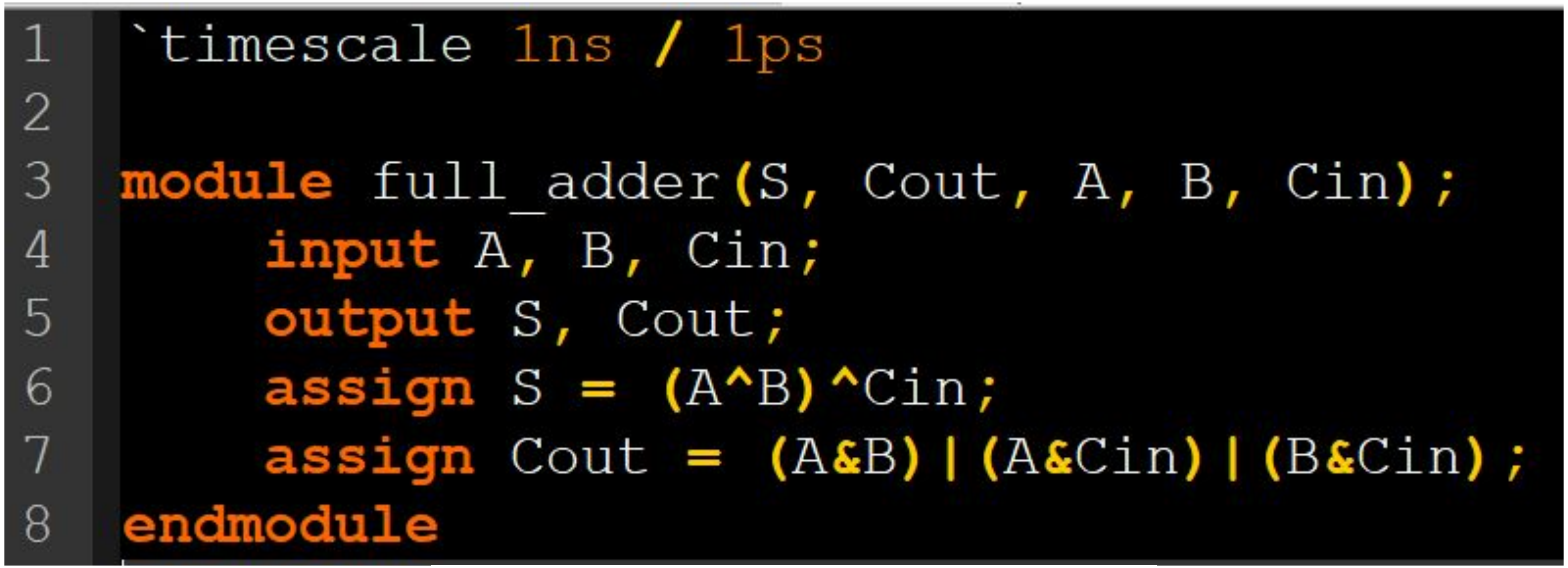
Step 1 - Implementing D flip-flop: A D-ff with asynchronous reset is implemented.



```verilog
`timescale 1ns / 1ps
//Asynchronous reset
module D_ff(Q, D, clk, reset);
    input D, clk, reset;
    output reg Q;
    always @(posedge clk or posedge reset) begin
    if(reset) Q <= 0;
    else Q <= D;
    end
endmodule
```

Figure.1 D flip-flop Verilog code

Step 2 – Implementing a Full Adder: Behavioral design approach is used.

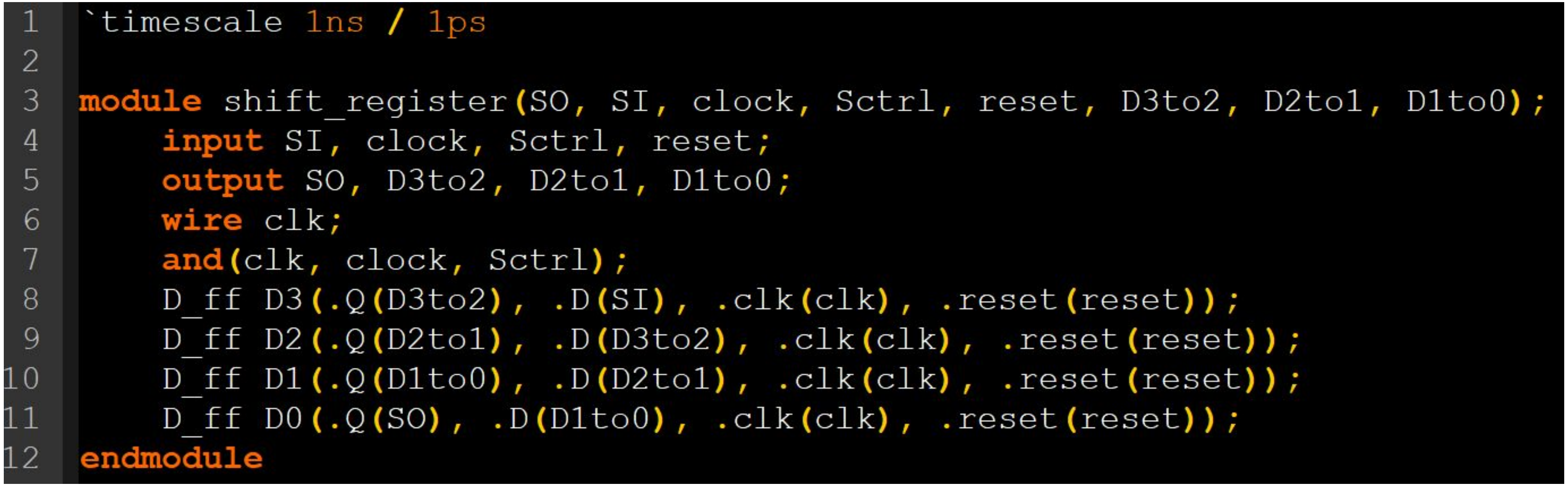


```
`timescale 1ns / 1ps

module full_adder(S, Cout, A, B, Cin);
    input A, B, Cin;
    output S, Cout;
    assign S = (A^B)^Cin;
    assign Cout = (A&B)|(A&Cin)|(B&Cin);
endmodule
```

Figure.2 Full Adder Verilog Code

Step 3 – Implementing a Shift Register: D3 flip-flop represents the most significant bit and D0 represents the least significant bit. We connected the flip-flops to each other by wires. We have resets in all D flip-flops in order to set the register to 0000 in the beginning, otherwise it will be XXXX. Shift register doesn’t work on every clock rise. It requires clock and Sctrl to be ON together which is named clk in the code. SO is the serial output which is the least significant bit. In order to see what is stored in the registers we also took D3to2, D2to1, D1to0 wires as output, together with SO they represent the 4 bits stored in the register.

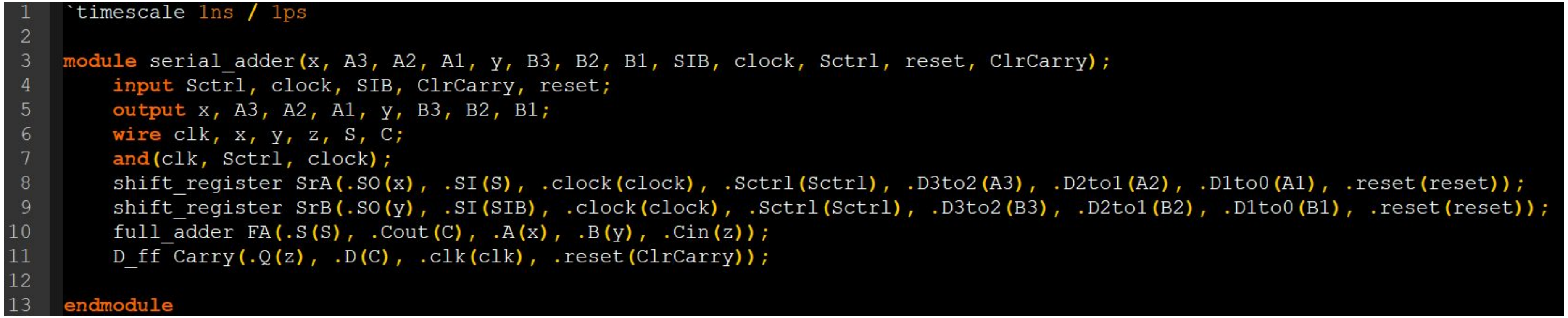
Step 3 – Implementing a Shift Register:



```verilog
`timescale 1ns / 1ps

module shift_register(SO, SI, clock, Sctrl, reset, D3to2, D2to1, D1to0);
    input SI, clock, Sctrl, reset;
    output SO, D3to2, D2to1, D1to0;
    wire clk;
    and(clk, clock, Sctrl);
    D_ff D3(.Q(D3to2), .D(SI), .clk(clk), .reset(reset));
    D_ff D2(.Q(D2to1), .D(D3to2), .clk(clk), .reset(reset));
    D_ff D1(.Q(D1to0), .D(D2to1), .clk(clk), .reset(reset));
    D_ff D0(.Q(SO), .D(D1to0), .clk(clk), .reset(reset));
endmodule
```

Figure.3 Shift Register Verilog code

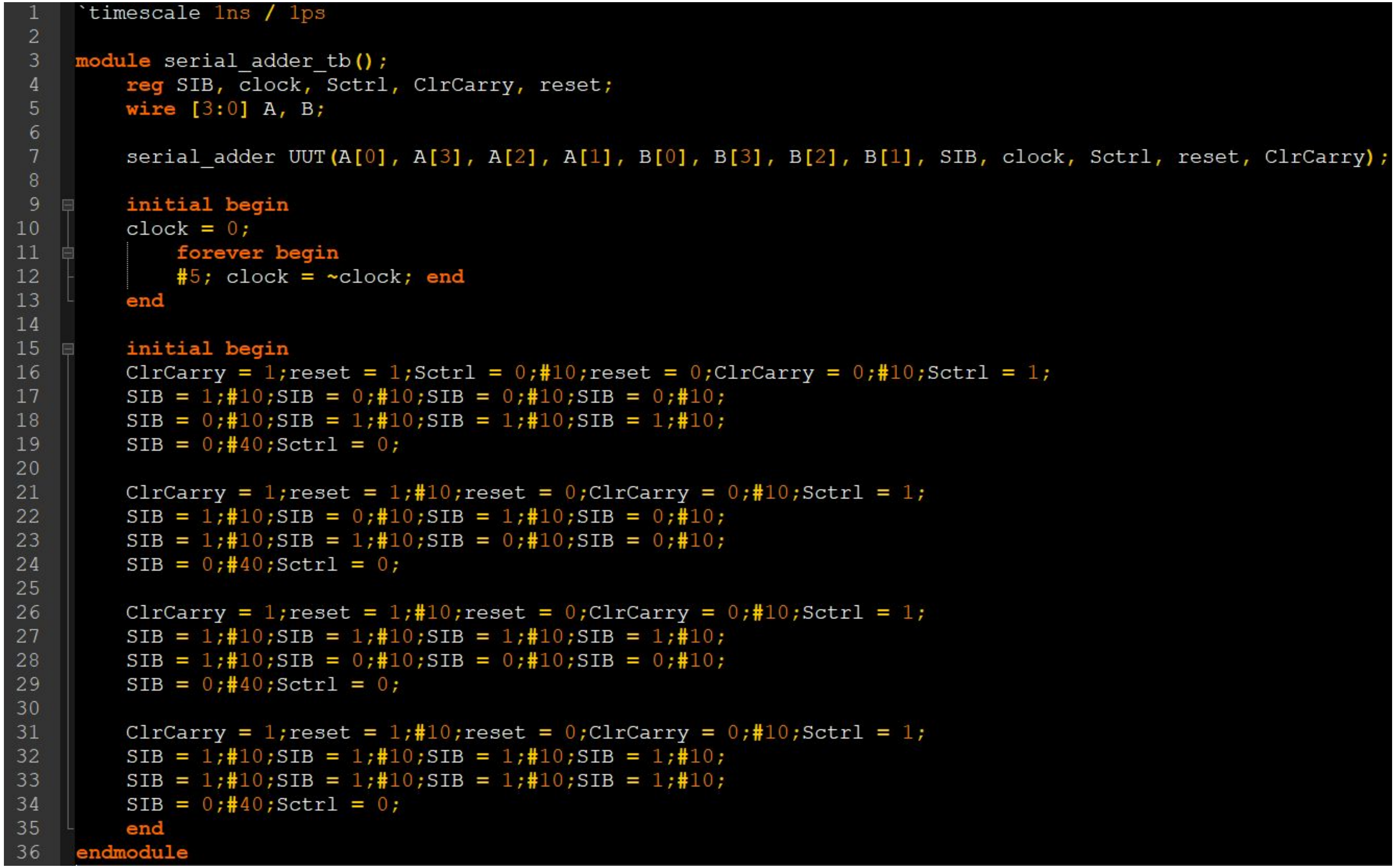
Step 4 – Implementing a Serial Adder: Numbers are input to Shift Register B (SIB) and Shift Register A behaves as the sum of numbers input to register B. The serial output of Shift Register A and B is fed into the full adder together with the carry stored inside Carry D-ff. ClrCarry is used for resetting carry flip-flop and reset is used for resetting both registers to 0000. The adder will work only if Sctrl is on and the clock is ticking.



```
`timescale 1ns / 1ps

module serial_adder(x, A3, A2, A1, y, B3, B2, B1, SIB, clock, Sctrl, reset, ClrCarry);
    input Sctrl, clock, SIB, ClrCarry, reset;
    output x, A3, A2, A1, y, B3, B2, B1;
    wire clk, x, y, z, S, C;
    and(clk, Sctrl, clock);
    shift_register SrA(.SO(x), .SI(S), .clock(clock), .Sctrl(Sctrl), .D3to2(A3), .D2to1(A2), .D1to0(A1), .reset(reset));
    shift_register SrB(.SO(y), .SI(SIB), .clock(clock), .Sctrl(Sctrl), .D3to2(B3), .D2to1(B2), .D1to0(B1), .reset(reset));
    full_adder FA(.S(S), .Cout(C), .A(x), .B(y), .Cin(z));
    D_ff Carry(.Q(z), .D(C), .clk(clk), .reset(ClrCarry));

endmodule
```

Figure.4 Serial Adder Verilog code

Step 5 – Writing the testbench for the Serial Adder: The 4-bit wires A and B are what is stored in Shift register A and B respectively. A clock configured to go on and off every 5 ms. For the given 4 initial states for testing, 4 code block are written, each starting with resetting the registers and the Carry D-ff. And then Shift control is turned on and input is given in to Shift register B (SIB).



```verilog
`timescale 1ns / 1ps

module serial_adder_tb();
    reg SIB, clock, Sctrl, ClrCarry, reset;
    wire [3:0] A, B;

    serial_adder UUT(A[0], A[3], A[2], A[1], B[0], B[3], B[2], B[1], SIB, clock, Sctrl, reset, ClrCarry);

    initial begin
    clock = 0;
        forever begin
        #5; clock = ~clock; end
    end

    initial begin
    ClrCarry = 1;reset = 1;Sctrl = 0;#10;reset = 0;ClrCarry = 0;#10;Sctrl = 1;
    SIB = 1;#10;SIB = 0;#10;SIB = 0;#10;SIB = 0;#10;
    SIB = 0;#10;SIB = 1;#10;SIB = 1;#10;SIB = 1;#10;
    SIB = 0;#40;Sctrl = 0;

    ClrCarry = 1;reset = 1;#10;reset = 0;ClrCarry = 0;#10;Sctrl = 1;
    SIB = 1;#10;SIB = 0;#10;SIB = 1;#10;SIB = 0;#10;
    SIB = 1;#10;SIB = 1;#10;SIB = 0;#10;SIB = 0;#10;
    SIB = 0;#40;Sctrl = 0;

    ClrCarry = 1;reset = 1;#10;reset = 0;ClrCarry = 0;#10;Sctrl = 1;
    SIB = 1;#10;SIB = 1;#10;SIB = 1;#10;SIB = 1;#10;
    SIB = 1;#10;SIB = 0;#10;SIB = 0;#10;SIB = 0;#10;
    SIB = 0;#40;Sctrl = 0;

    ClrCarry = 1;reset = 1;#10;reset = 0;ClrCarry = 0;#10;Sctrl = 1;
    SIB = 1;#10;SIB = 1;#10;SIB = 1;#10;SIB = 1;#10;
    SIB = 1;#10;SIB = 1;#10;SIB = 1;#10;SIB = 1;#10;
    SIB = 0;#40;Sctrl = 0;
    end
endmodule
```

Figure.5 Serial Adder Testbench Verilog code

Simulation Results: Explanation for the first test; the sum of 0001 + 1110 is wanted. To achieve this we fed 0001 into register B and then 1110. To clarify, until 20ms The Serial adder is reset. At 20ms Sctrl is on and input (SIB) starts to be fed into shift register B. From 20 to 60ms 1, 0, 0, 0 is fed (notice the least significant bit is fed first) into SIB and now register B is set to 0001 while reg A is still 0000. The next 4 clocks until 100ms, At the same time 0, 1, 1, 1 is fed into SIB and 0000+0001 is calculated and stored in register A so now reg B is 1110 and reg A is 0001 which is what we wanted. The next 4 clocks simply A and B is added and stored in A (100ms to 140ms). The yellow line represents the final state after addition. Basically, we input 2 4-bit numbers one after the other and the sum is stored in A.

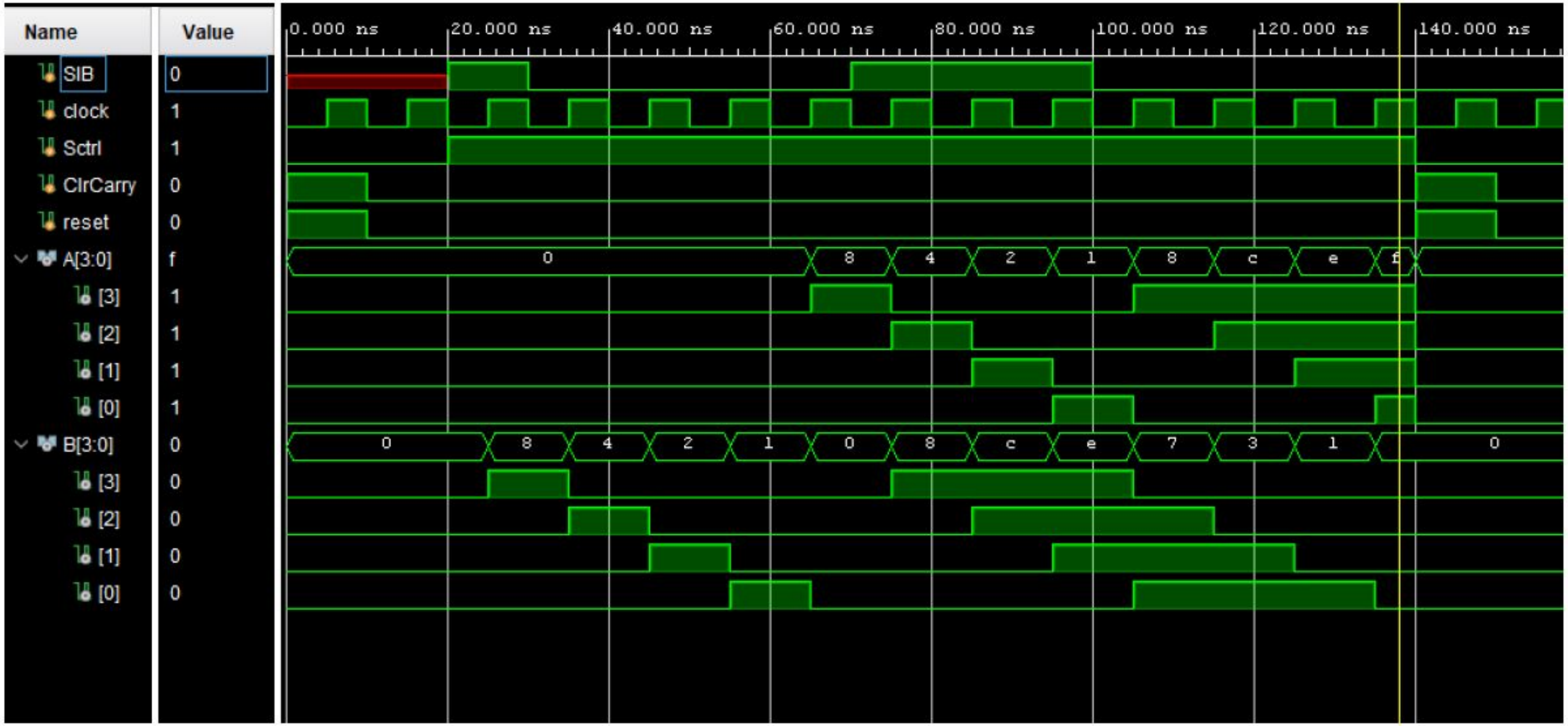


Figure.6 Waveform for 0001 + 1110

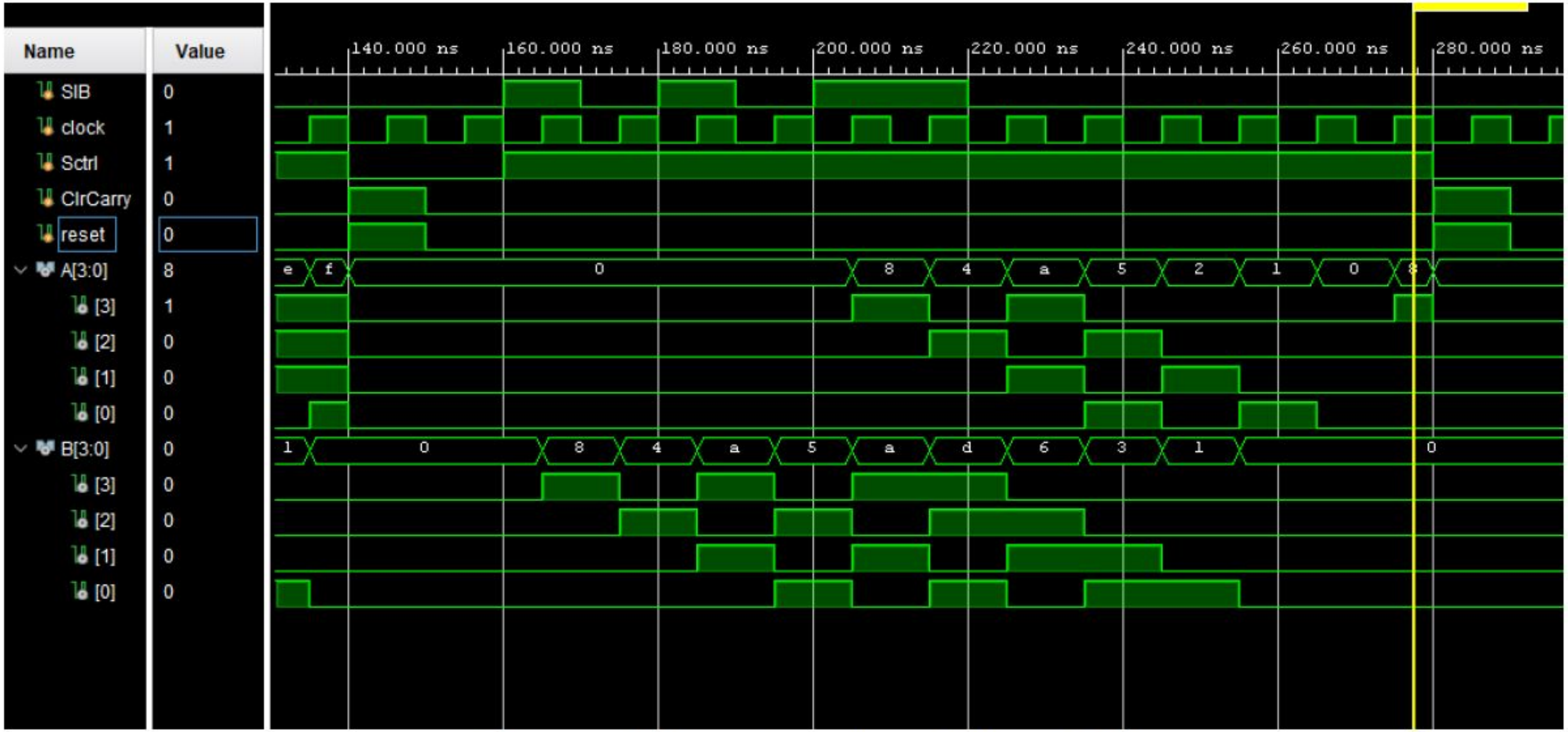
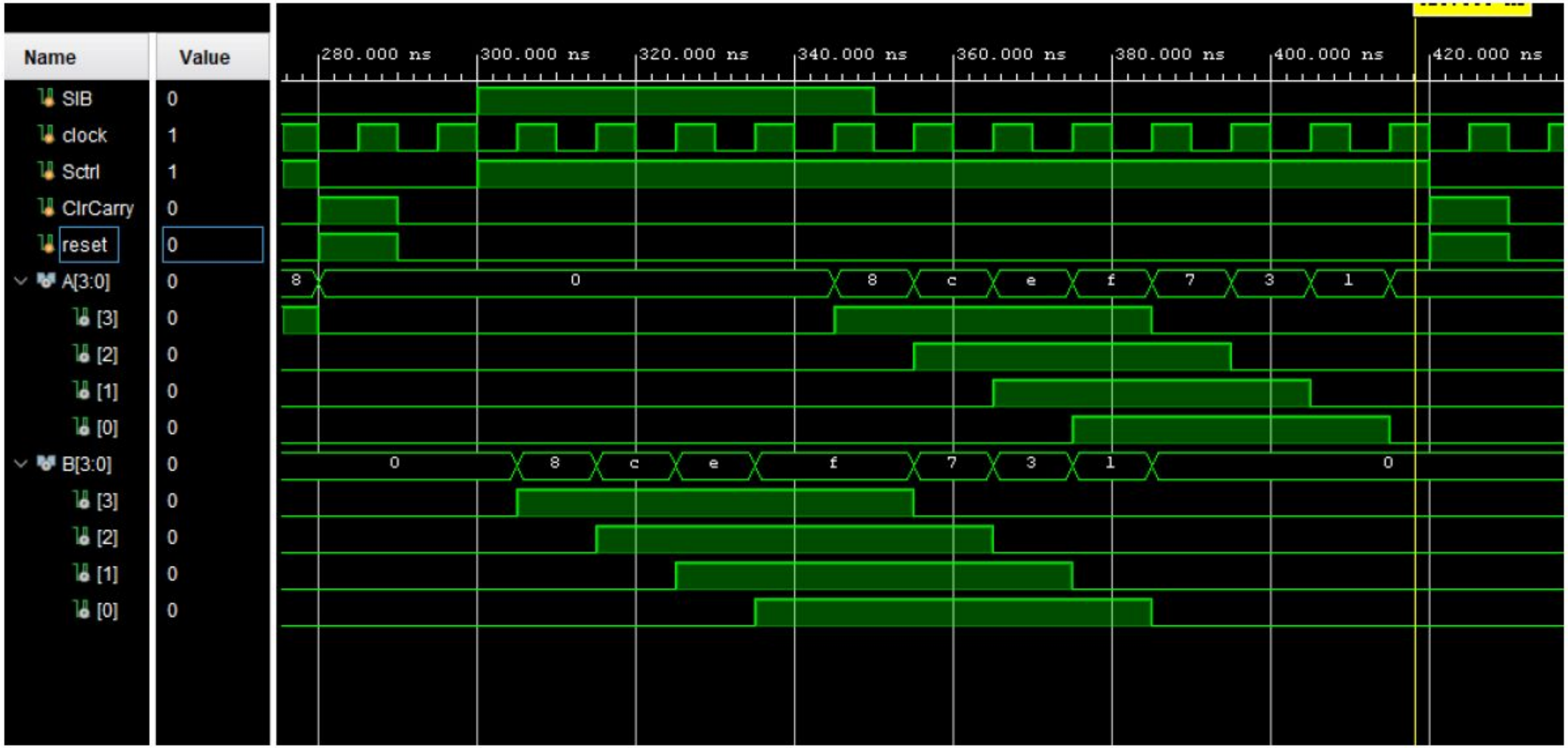


Figure.7 Waveform for 0101 + 0011



Name	Value
SIB	0
clock	1
Sctrl	1
ClrCarry	0
reset	0
A[3:0]	0
[3]	0
[2]	0
[1]	0
[0]	0
B[3:0]	0
[3]	0
[2]	0
[1]	0
[0]	0

Figure.8 Waveform for 1111 + 0011

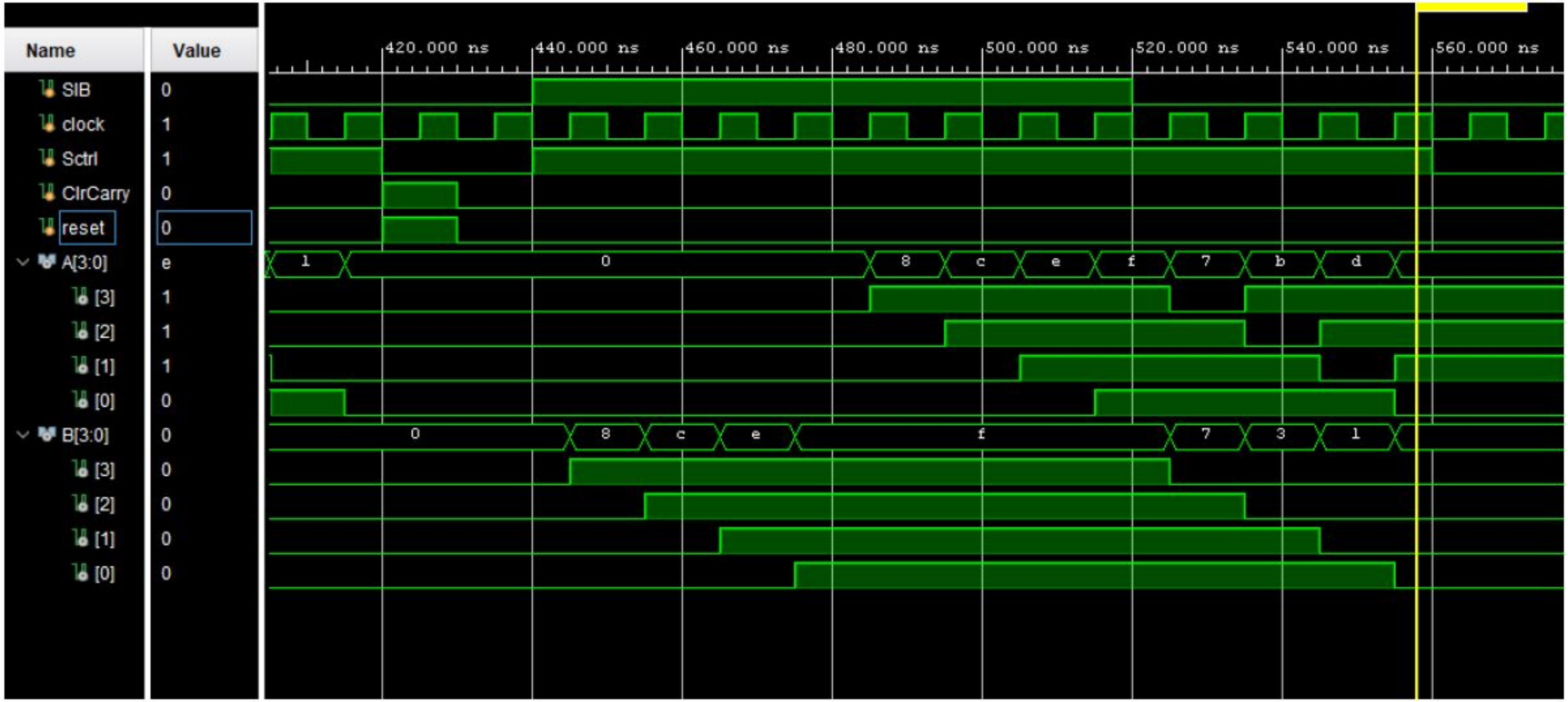
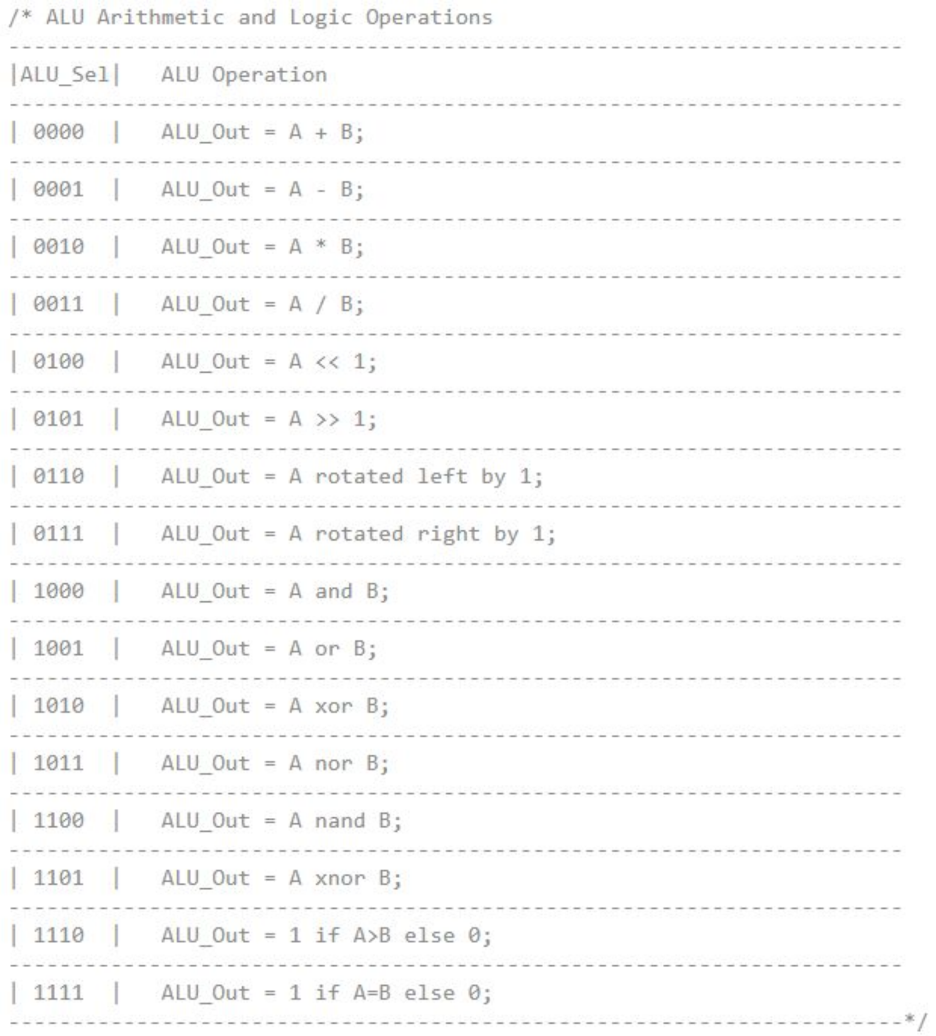


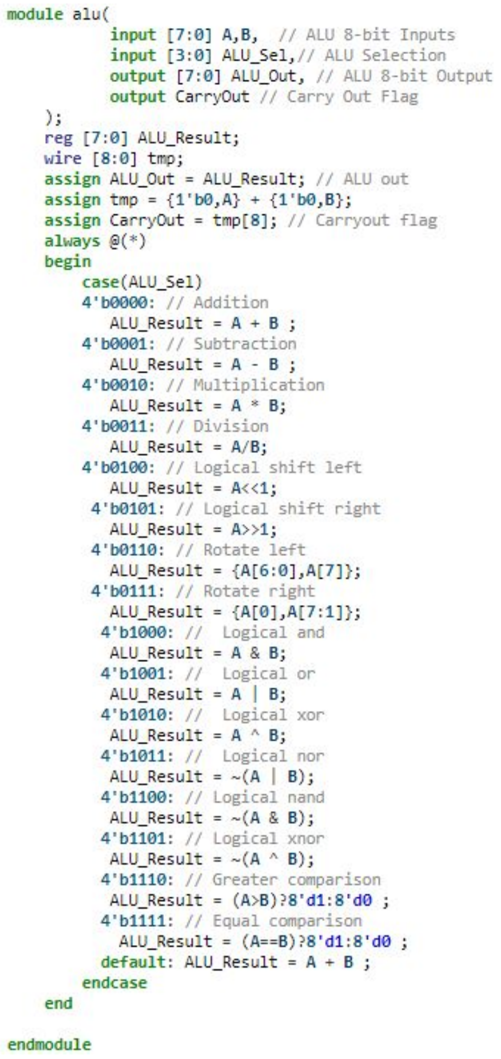
Figure.9 Waveform for 1111+1111

Arithmetic Logic Unit (ALU)
An Example of Behavioral Design

Solution from https://www.fpga4student.com

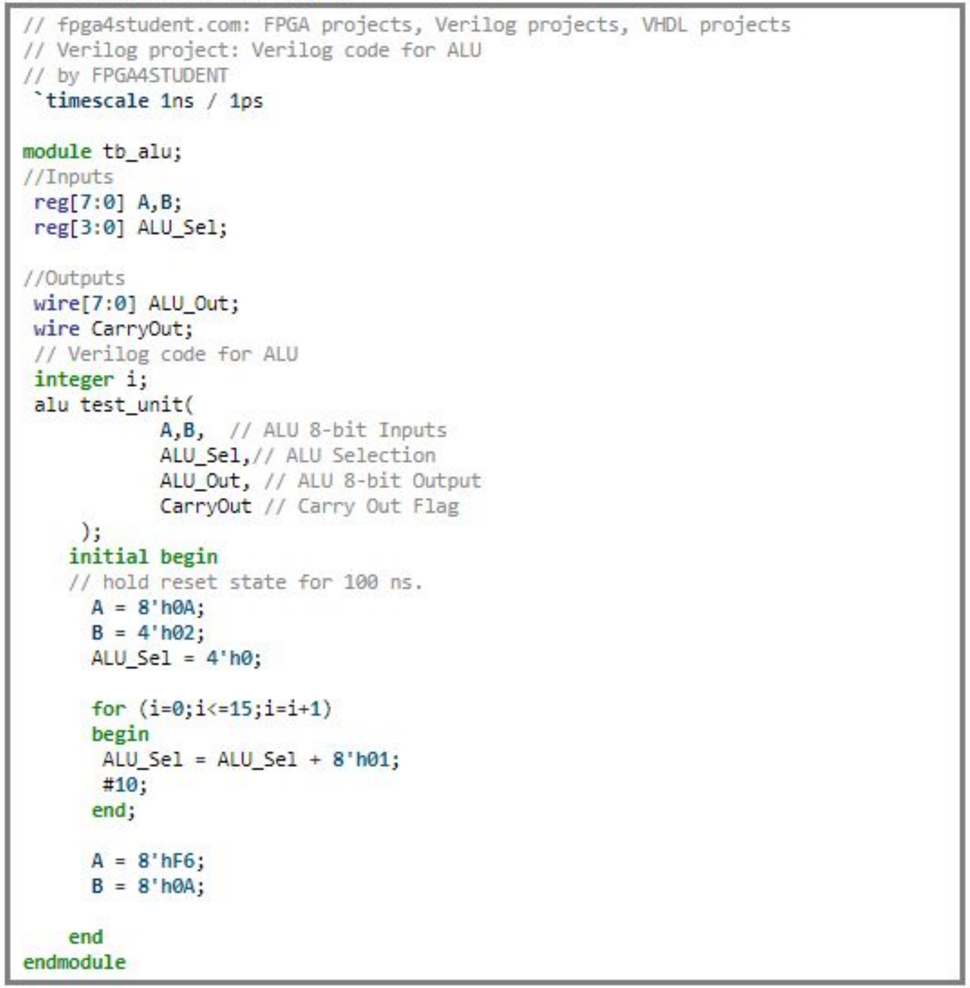


```
/* ALU Arithmetic and Logic Operations
----------------------------------------------------------------------
|ALU_Sel|   ALU Operation
----------------------------------------------------------------------
| 0000  |   ALU_Out = A + B;
----------------------------------------------------------------------
| 0001  |   ALU_Out = A - B;
----------------------------------------------------------------------
| 0010  |   ALU_Out = A * B;
----------------------------------------------------------------------
| 0011  |   ALU_Out = A / B;
----------------------------------------------------------------------
| 0100  |   ALU_Out = A << 1;
----------------------------------------------------------------------
| 0101  |   ALU_Out = A >> 1;
----------------------------------------------------------------------
| 0110  |   ALU_Out = A rotated left by 1;
----------------------------------------------------------------------
| 0111  |   ALU_Out = A rotated right by 1;
----------------------------------------------------------------------
| 1000  |   ALU_Out = A and B;
----------------------------------------------------------------------
| 1001  |   ALU_Out = A or B;
----------------------------------------------------------------------
| 1010  |   ALU_Out = A xor B;
----------------------------------------------------------------------
| 1011  |   ALU_Out = A nor B;
----------------------------------------------------------------------
| 1100  |   ALU_Out = A nand B;
----------------------------------------------------------------------
| 1101  |   ALU_Out = A xnor B;
----------------------------------------------------------------------
| 1110  |   ALU_Out = 1 if A>B else 0;
----------------------------------------------------------------------
| 1111  |   ALU_Out = 1 if A=B else 0;
----------------------------------------------------------------------*/
```



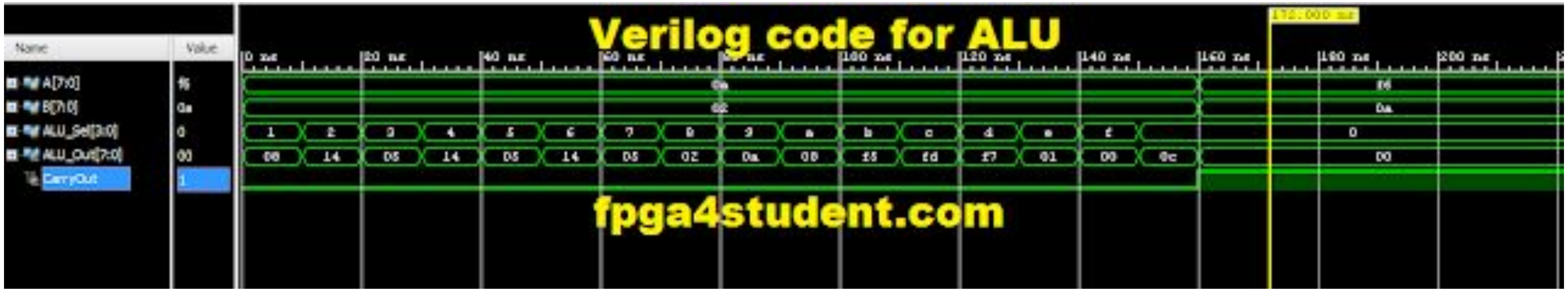
```
module alu(
           input [7:0] A,B,  // ALU 8-bit Inputs
           input [3:0] ALU_Sel,// ALU Selection
           output [7:0] ALU_Out, // ALU 8-bit Output
           output CarryOut // Carry Out Flag
    );
    reg [7:0] ALU_Result;
    wire [8:0] tmp;
    assign ALU_Out = ALU_Result; // ALU out
    assign tmp = {1'b0,A} + {1'b0,B};
    assign CarryOut = tmp[8]; // Carryout flag
    always @(*)
    begin
        case(ALU_Sel)
        4'b0000: // Addition
           ALU_Result = A + B ;
        4'b0001: // Subtraction
           ALU_Result = A - B ;
        4'b0010: // Multiplication
           ALU_Result = A * B;
        4'b0011: // Division
           ALU_Result = A/B;
        4'b0100: // Logical shift left
           ALU_Result = A<<1;
         4'b0101: // Logical shift right
           ALU_Result = A>>1;
         4'b0110: // Rotate left
           ALU_Result = {A[6:0],A[7]};
         4'b0111: // Rotate right
           ALU_Result = {A[0],A[7:1]};
          4'b1000: //  Logical and
           ALU_Result = A & B;
          4'b1001: //  Logical or
           ALU_Result = A | B;
          4'b1010: //  Logical xor
           ALU_Result = A ^ B;
          4'b1011: //  Logical nor
           ALU_Result = ~(A | B);
          4'b1100: // Logical nand
           ALU_Result = ~(A & B);
          4'b1101: // Logical xnor
           ALU_Result = ~(A ^ B);
          4'b1110: // Greater comparison
           ALU_Result = (A>B)?8'd1:8'd0 ;
          4'b1111: // Equal comparison
            ALU_Result = (A==B)?8'd1:8'd0 ;
          default: ALU_Result = A + B ;
        endcase
    end

endmodule
```

Testbench Verilog code for the ALU:



```
// fpga4student.com: FPGA projects, Verilog projects, VHDL projects
// Verilog project: Verilog code for ALU
// by FPGA4STUDENT
 `timescale 1ns / 1ps

module tb_alu;
//Inputs
 reg[7:0] A,B;
 reg[3:0] ALU_Sel;

//Outputs
 wire[7:0] ALU_Out;
 wire CarryOut;
 // Verilog code for ALU
 integer i;
 alu test_unit(
            A,B,  // ALU 8-bit Inputs
            ALU_Sel,// ALU Selection
            ALU_Out, // ALU 8-bit Output
            CarryOut // Carry Out Flag
     );
    initial begin
    // hold reset state for 100 ns.
      A = 8'h0A;
      B = 4'h02;
      ALU_Sel = 4'h0;

      for (i=0;i<=15;i=i+1)
      begin
       ALU_Sel = ALU_Sel + 8'h01;
       #10;
      end;

      A = 8'hF6;
      B = 8'h0A;

    end
endmodule
```

Simulation waveform for the ALU:


Verilog code for ALU
172.000 ns
Name
Value
A[7:0]
f6
B[7:0]
0a
ALU_Sel[3:0]
0
ALU_Out[7:0]
00
CarryOut
1
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
0a
02
f6
0a
1 2 3 4 5 6 7 8 9 a b c d e f
0
08 14 05 14 05 14 05 02 0a 08 f5 fd f7 01 00 0c
00
fpga4student.com