100daysof RTL

Part-1(Verilog)

Ummidi Chandrika

WELCOME TO MY 100DAYSOFRTL

HERE IS THE LIST OF DAY WISE RTL CODES:

DAY 1 : CLOCK DIVIDER

DAY 2 : JOHNSON COUNTER

DAY 3: RING COUNTER

DAY 4: 5 INPUT MAJORITY CIRCUIT

DAY 5: PARITY GENERATOR

DAY 6: BINARY TO ONE HOT ENCODER

DAY 7: 4-BIT BCD SYNCHRONOUS COUNTER

DAY 8: 4-BIT CARRY LOOKAHEAD ADDER

DAY 9: N-BIT COMPARATOR

DAY 10: SERIAL IN SERIAL OUT SHIFT REGISTER

DAY 11: SERIAL IN PARALLEL OUT SHIFT REGISTER

DAY 12: PARALLEL IN PARALLEL OUT REGISTER

DAY 13: PARALLEL IN SERIAL OUT REGISTER

DAY 14: BIDIRECTION SHIFT REGISTER

DAY 15: PRBS SEQUENCE GENERATOR

DAY 16: 8-BIT SUBTRACTOR

DAY 17: 8-BIT ADDER/SUBTRACTOR

DAY 18: 4-BIT MULTIPLIER

DAY 19: FIXED POINT DIVISION

DAY 20: MASTER SLAVE JK FLIP FLOP

DAY 21: POSITIVE EDGE DETECTOR

DAY 22: BCD ADDER

DAY 23: 4-BIT CARRY SELECT ADDER

DAY 24: MOORE FSM 1010 SEQUENCE DETECTOR

DAY 25: N:1 MUX

DAY 26: BCD TIMECOUNT

DAY 27: 3-1 MUX

DAY 28: BCD TO SEVEN SEGMENT DISPLAY

DAY 29: D LATCH USING 2:1 MUX

DAY 30: 8-BIT BARREL SHIFTER
DAY 31: 1-BIT COMPARATOR USING 4X1 MUX
DAY 32: LOGICAL, ALGEBRAIC, AND ROTATE SHIFT OPERATIONS
DAY 33: ALU
DAY 34: 4-BIT ASYNCHRONOUS DOWN COUNTER
DAY 35: MOD-N UPDOWN COUNTER
DAY 36: UNIVERSAL BINARY COUNTER
DAY 37: UNIVERSAL SHIFT REGISTER
DAY 38: CN(CHANGE-NO CHANGE FLIPFLOP) USING 2:1 MUX
DAY 39: FREQUENCY DIVIDER BY ODD NUMBERS
DAY 40: GREATEST COMMON DIVISOR USING BEHAVIOURAL MODELLING
DAY 41: GREATEST COMMON DIVISOR VIA FSM
DAY 42: SINGLE PORT RAM
DAY 43: DUAL PORT RAM
DAY 44: CLOCK BUFFER
DAY 45: SYNCHRONOUS FIFO
DAY 46: PRIORITY ENCODER
DAY 47: SEVEN SEGMENT DISPLAY USING ROM
DAY 48: SERIAL ADDER
DAY 49: FIXED PRIORITY ARBITER
DAY 50: ROUND ROBIN ARBITER

100 Days of RTL

Part-1 (Verilog)

Day-1: Clock Divider

A clock divider circuit creates lower frequency clock signals from an input clock source. The divider circuit counts input clock cycles, and drives the output clock low and then high for some number of input clock cycles. where fout = fin / n and "n" is an integer. Frequency dividers are used for both analog and digital applications.

Approach:

The code is simple to understand. All you need to do is to set the output clock to 0 at the time of reset(0). In this example the reset was required to be synchronous. The code is simple as all we need to do is invert the output clock at each of its rising edge.

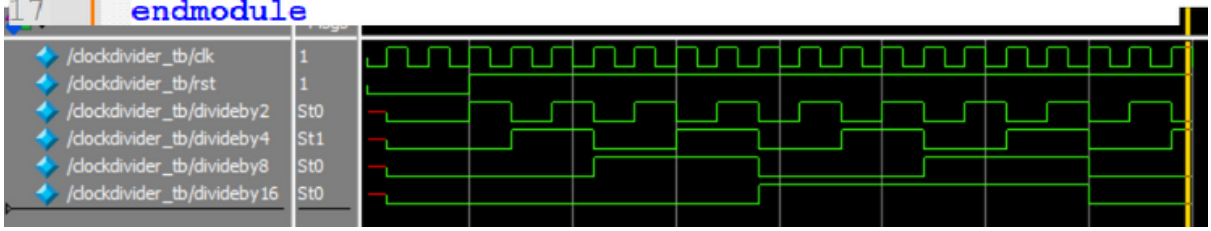
Take a temporary 4-bit register as count[3:0].The count is made to increment by 1 for every positive edge of the clock. Then it will acts as a counter from 0-15

The count[0] acts as a Divideby2 circuit,

count[1] acts as a Divideby4 circuit,
count[2] acts as a Divideby8 circuit,
count[3] acts as a Divideby16 circuit

RTL Code:

```
module clockdivider(clk,divideby2,divideby4,
divideby8,divideby16,rst);
input clk,rst;
reg [3:0]count;
output reg divideby2,divideby4,divideby8,divideby16;
always@(posedge clk)
begin
if(rst==0)
count=4'b0000;
else
count=count+1;
  divideby2=count[0];
  divideby4=count[1];
  divideby8=count[2];
  divideby16=count[3];
end
endmodule
```



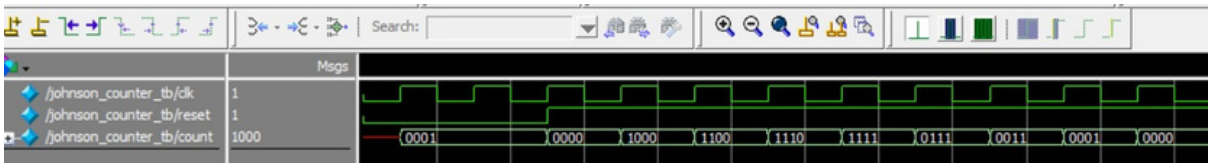
Day-2 Johnson Counter

A Johnson counter is a digital circuit with a series of flip flops connected in a feedback manner.
The circuit is a special type of shift register where the last flip flop's complement output is fed back to the first flip flop's input. This is almost similar to the ring counter with a few extra advantages.

The Johnson counter's main advantage is that it only needs half the number of flip-flops compared to the standard ring counter, and then it's modulo number is halved. So an n-stage Johnson counter will circulate a single data bit, giving a sequence of 2n different states and can therefore be considered a mod-2n counter.

RTL Code:

```
1   module johnson_counter(clk,reset,count);
2   parameter WIDTH=4;
3   input clk,reset;
4   output reg [WIDTH-1:0] count;
5
6   always@(posedge clk)
7   begin
8   if(reset)
9   count={~count[0],count[WIDTH-1:1]};
10  else
11  count=4'b0001;
12  end
13  endmodule
```

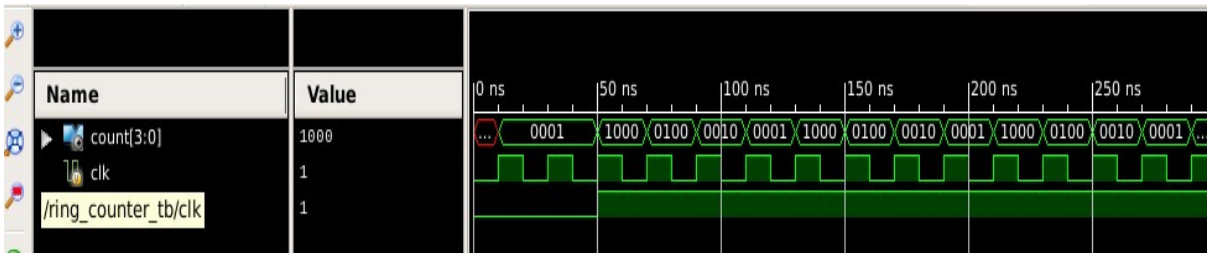


Day-3 Ring Counter

A ring counter is a special type of application of the Serial IN Serial OUT Shift register. The only difference between the shift register and the ring counter is that the last flip flop outcome is taken as the output in the shift register. But in the ring counter, this outcome is passed to the first flip flop as an input. All of the remaining things in the ring counter are the same as the shift register.In the Ring counter, No. of states in Ring counter = No. of flip-flop used.

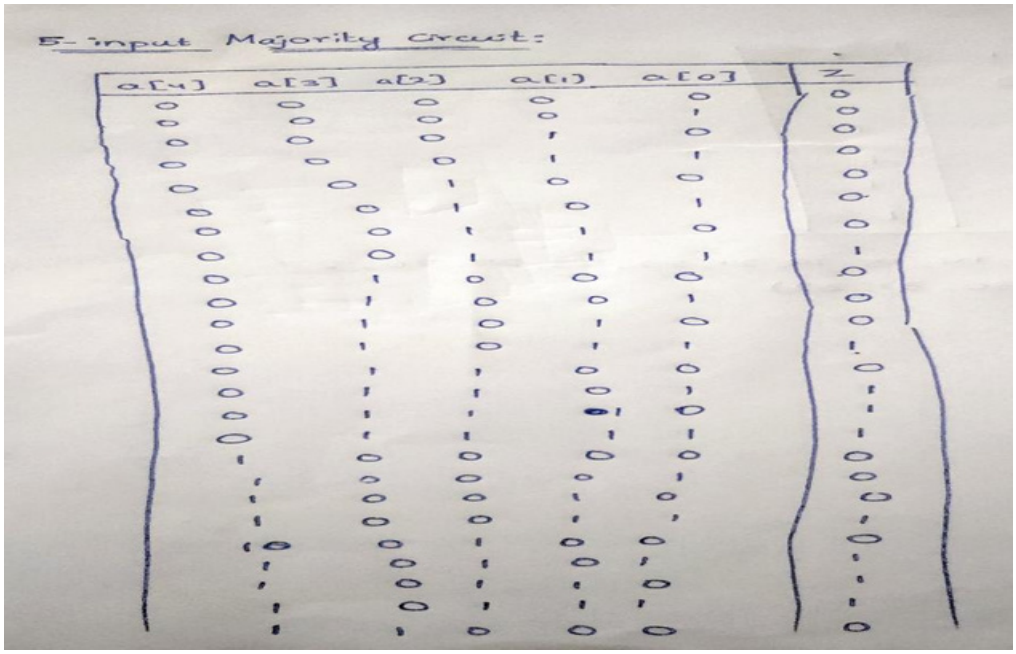
RTL Code:

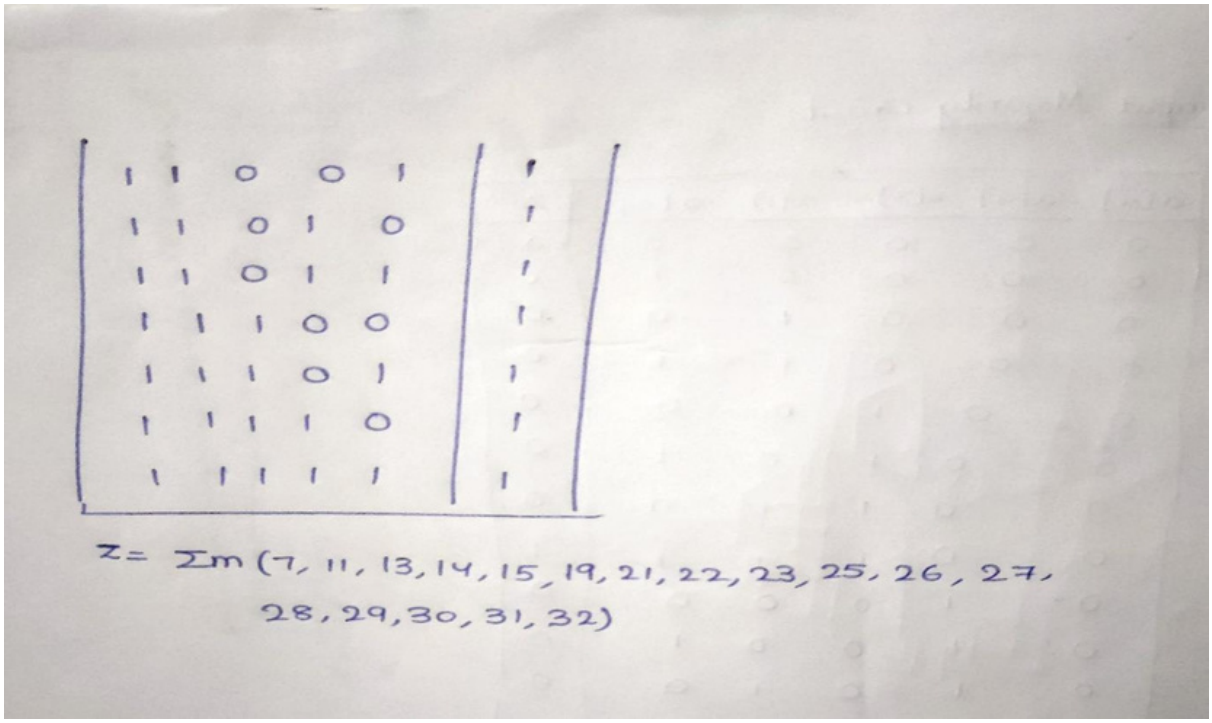
```
1   module johnson_counter(clk,reset,count);
2   parameter WIDTH=4;
3   input clk,reset;
4   output reg [WIDTH-1:0] count;
5
6   always@(posedge clk)
7   begin
8   if(reset)
9   count={~count[0],count[WIDTH-1:1]};
10  else
11  count=4'b0001;
12  end
13  endmodule
```



Day-4 : 5 Input Majority Circuit

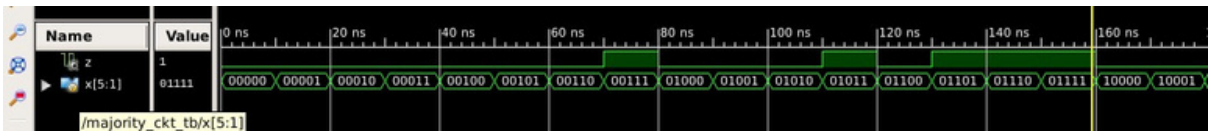
The design of a majority circuit using built-in primitives. The output of a majority circuit is a logic 1 if the majority of the inputs is a logic 1; otherwise, the output is a logic 0. Therefore, a majority circuit must have an odd number of inputs in order to have a majority of the inputs at the same indicates that the majority of the inputs is a logic 1.





RTL Code:

```
module majority_ckt(input [5:1] x, output z);
wire [9:0] w;
// instantiate with And Gates
and and1(w[0],x[3],x[4],x[5]);
and and2(w[1],x[2],x[4],x[5]);
and and3(w[2],x[2],x[3],x[5]);
and and4(w[3],x[2],x[4],x[3]);
and and5(w[4],x[1],x[4],x[5]);
and and6(w[5],x[1],x[3],x[5]);
and and7(w[6],x[1],x[4],x[3]);
and and8(w[7],x[2],x[1],x[5]);
and and9(w[8],x[2],x[4],x[1]);
and and10(w[9],x[2],x[1],x[3]);

or or1(z,w[0],w[1],w[2],w[3],w[4],
w[5],w[6],w[7],w[8],w[9]);

endmodule
```

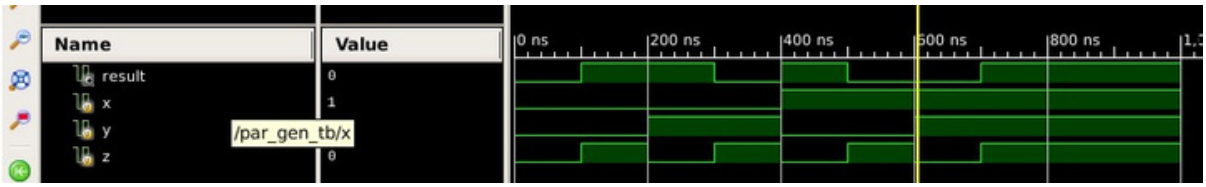


Day-5 Parity Generator

A Parity Generator designed is Odd Parity Generator circuit, it maintains the binary data in an odd number of 1's, for example, the data taken is in even number of 1's, this odd parity generator is going to maintain the data as an odd number of 1's by adding the extra 1 to the even number of 1's. This is the combinational circuit whose output is always dependent upon the given input data.

RTL Code:

```
module par_gen(x,y,z,result);
input x,y,z;
output result;
xor (result,x,y,z);
endmodule
```

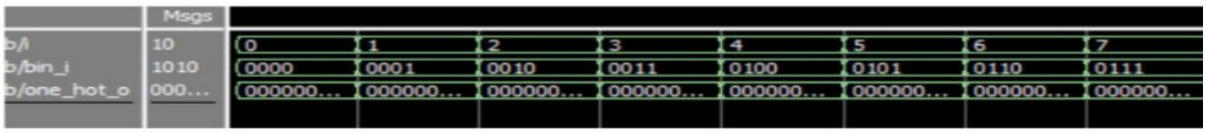


Day 6- Binary to One Hot Encoder

Binary To One-Hot Converter generates an output bit vector of up to 2^N bits with one bit set representing the N-bit input binary value. One-hot encoding is particularly advantageous for FPGA implementations. The width of the output vector is limited by the Verilog implementation to about a million bits, which means a maximum input binary width of 20 bits.

RTL Code:

```
module one_hot(bin_i,one_hot_o);
parameter BIN_W=4;
parameter ONE_HOT_W=16;
input   [BIN_W-1:0]     bin_i;
 output  reg [ONE_HOT_W-1:0] one_hot_o;

  assign one_hot_o = 1'b1<<bin_i;

endmodule
```



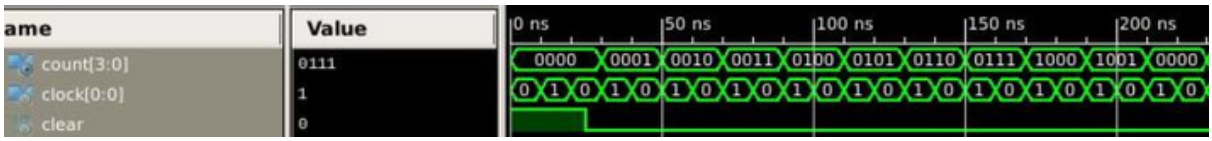

```
VSIM 3> run
# Binary Value: 0000 | One hot encoded value : 0000000000000001
# Binary Value: 0001 | One hot encoded value : 0000000000000010
# Binary Value: 0010 | One hot encoded value : 0000000000000100
# Binary Value: 0011 | One hot encoded value : 0000000000001000
# Binary Value: 0100 | One hot encoded value : 0000000000010000
# Binary Value: 0101 | One hot encoded value : 0000000000100000
# Binary Value: 0110 | One hot encoded value : 0000000001000000
# Binary Value: 0111 | One hot encoded value : 0000000010000000
# Binary Value: 1000 | One hot encoded value : 0000000100000000
# Binary Value: 1001 | One hot encoded value : 0000001000000000
# Binary Value: 1010 | One hot encoded value : 0000010000000000
# Binary Value: 1011 | One hot encoded value : 0000100000000000
# Binary Value: 1100 | One hot encoded value : 0001000000000000
# Binary Value: 1101 | One hot encoded value : 0010000000000000
# Binary Value: 1110 | One hot encoded value : 0100000000000000
# Binary Value: 1111 | One hot encoded value : 1000000000000000
```

Day 7- 4-Bit Synchronous BCD Counter:

A BCD counter is one of the 4-bit binary counters, which counts from 0 to a pre-determined count with an applied clock signal. When the count reaches the predetermined count value, it resets all the flip-flops and starts to count again from 0. This type of counter is designed by using 4 JK flip flops and counts from 0 to 9, and the result is represented in digital form. After reaching the count of 9 (1001), it resets and starts again.

RTL Code:

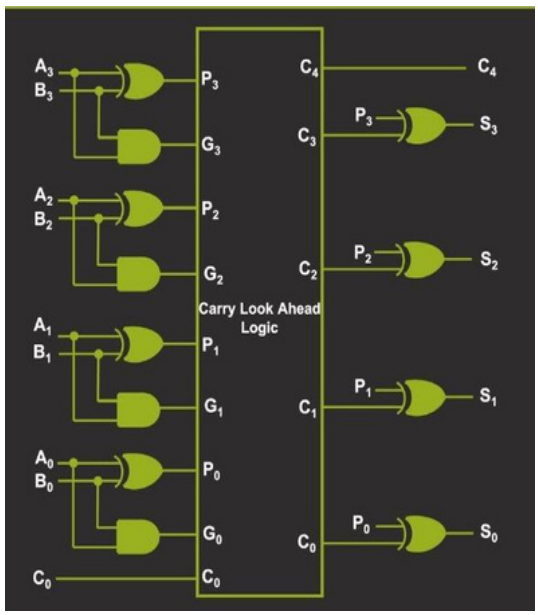
```
module bcd_counter(input clock, clear,
                   output reg [3:0] count);
  reg [3:0] t;
always @ (posedge clock) begin
    if (clear)
    begin
       t <= 4'b0000;
       count <= 4'b0000;
    end
    else
    begin
       t <= t + 1;
       if (t == 4'b1001)
       begin
          t <= 4'b0000;
       end
       count <= t;
    end
end
  endmodule
```



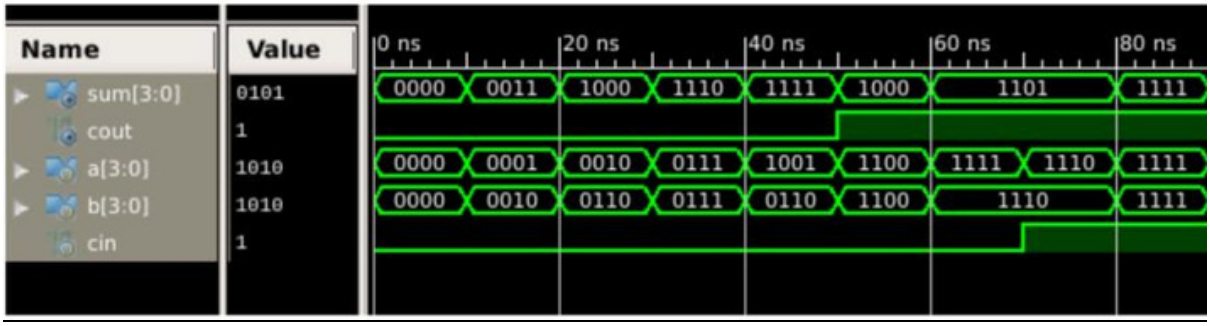
Day 8 : 4-Bit Carry Look Ahead Adder:

A carry look-ahead adder reduces the propagation delay by introducing more complex hardware. In this design, the ripple carry design is suitably transformed such that the

```
module cla_4bit (a, b, cin, sum, cout);
input [3:0] a, b; //define inputs and outputs
input cin;
output [3:0] sum;
output cout;
//design the logic for the generate functions
and (g0, a[0], b[0]),
(g1, a[1], b[1]),
(g2, a[2], b[2]),
(g3, a[3], b[3]);
//design the logic for the propagate functions
xor (p0, a[0], b[0]),
(p1, a[1], b[1]),
(p2, a[2], b[2]),
(p3, a[3], b[3]);
//design the logic for the sum equations
xor (sum[0], p0, cin),
(sum[1], p1, c0),
(sum[2], p2, c1),
(sum[3], p3, c2);//design the logic for the carry
//using the continuous assign statement
assign c0 = g0 | (p0 & cin),
c1 = g1 | (p1 & g0) | (p1 & p0 & cin),
c2 = g2 | (p2 & g1) | (p2 & p1 & g0)
 | (p2 & p1 & p0 & cin),
c3 = g3 | (p3 & g2) | (p3 & p2 & g1)
 | (p3 & p2 & p1 & g0)
 |(p3 & p2 & p1 & p0 & cin);
//design the logic for cout using assign
assign cout = c3;
endmodule
```



carry logic over fixed groups of bits of the adder is reduced to two-level logic.

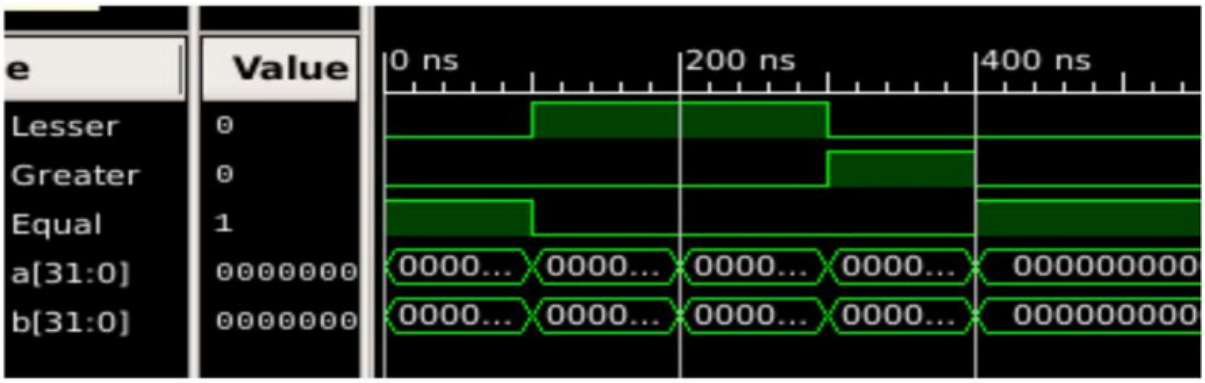


Day 9: N-bit Comparator:

A magnitude digital Comparator is a combinational circuit that compares two digital or binary numbers in order to find out whether one binary number is equal, less than, or greater than the other binary number.

RTL Code:

```
module comparator_nbit(a,b,Lesser,
Greater,Equal);
parameter n=32;
input [n-1:0]a,b;
output Lesser,Greater,Equal;
reg Lesser=0,Greater=0,Equal=0;
always @ (a,b)
begin
if(a>b)
begin
Lesser=0;Equal=0;Greater=1;
end
else if (a<b)
begin
Lesser=1;Equal=0;Greater=0;
end
else
begin
Lesser=0;Equal=1;Greater=0;
end
end
endmodule
```



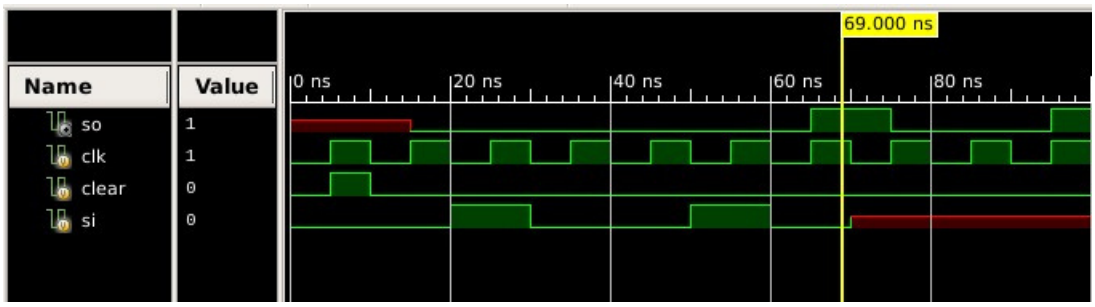

```
Finished circuit initialization process.
  A=     2  B=     2  Lesser=0   Greater=0   Equal=1
  A=    22  B=   444  Lesser=1   Greater=0   Equal=0
  A=   444  B=   555  Lesser=1   Greater=0   Equal=0
  A=   777  B=   111  Lesser=0   Greater=1   Equal=0
  A=  8888  B=  8888  Lesser=0   Greater=0   Equal=1
```

Day 10: Serial In Serial Out Shift Register

SISO shift register circuit accepts serial data on its input pin and shifts it out serially on its output pin. The number of bits that can be shifted out before the next bit arrives depends on the speed of the clock signal that controls the operation of the shift register. This type of shift register can be used as a buffer between two asynchronous devices that communicate with each other using signals with different frequencies or phases.

RTL Code:

```
module siso_reg(clk,clear,si,so);
input clk,si,clear;
output  so;
reg [3:0] tmp;
always @(posedge clk )
begin
if (clear)
tmp <= 4'b0000;
else
tmp <= tmp >>1 ;
tmp[3] <= si;
end
assign so = tmp[0];
endmodule
```



Day 11- Serial in parallel Out Shift Register

A serial-in, parallel-out shift register is similar to the serial-in, serial-out shift register in that it shifts data into internal storage elements and shifts data out at the serial-out, data-out, pin.

It is different in that it makes all the internal stages available as outputs. Therefore, a serial-in, parallel-out shift register converts data from serial format to parallel format.If four data bits are shifted in by four clock pulses via a single wire at data-in, below, the data becomes available simultaneously on the four Outputs QA to QD after the fourth clock pulse.

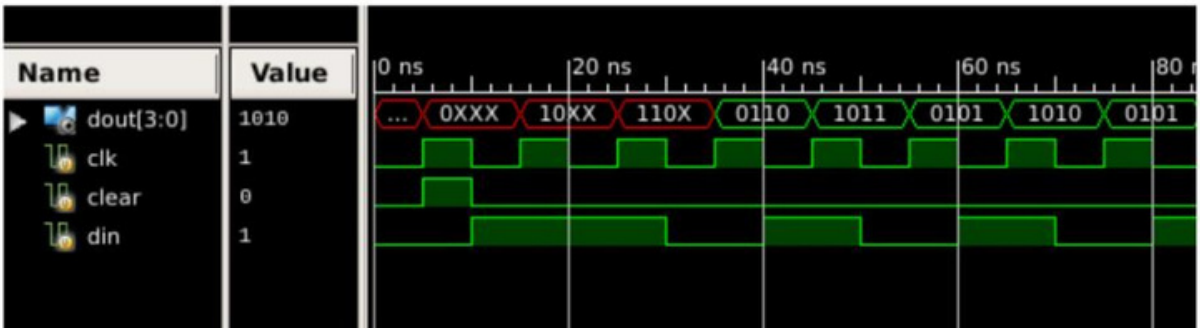
• For 'n' bit serial input data which need to be stored, the number of clock pulse required are equal to 'n'.

• For 'n' bit parallel output data which need to be stored, the number of clock pulse required is zero. As no clock pulse are required for this operation.

RTL Code:

```
module pipo_reg(clk,clear,d,q);
input clk,clear;
output reg [3:0]  q;
input [3:0] d;
always @(posedge clk )
begin
if (clear)
q<=  4'b0000;
else
q<=d;
end
endmodule
 end
  endmodule
```

```
Data Input = 0 | Data output =xxxx
Data Input = 0 | Data output =0xxx
Data Input = 1 | Data output =0xxx
Data Input = 1 | Data output =10xx
Data Input = 1 | Data output =10xx
Data Input = 1 | Data output =110x
Data Input = 0 | Data output =110x
Data Input = 0 | Data output =0110
Data Input = 1 | Data output =0110
Data Input = 1 | Data output =1011
Data Input = 0 | Data output =1011
Data Input = 0 | Data output =0101
```

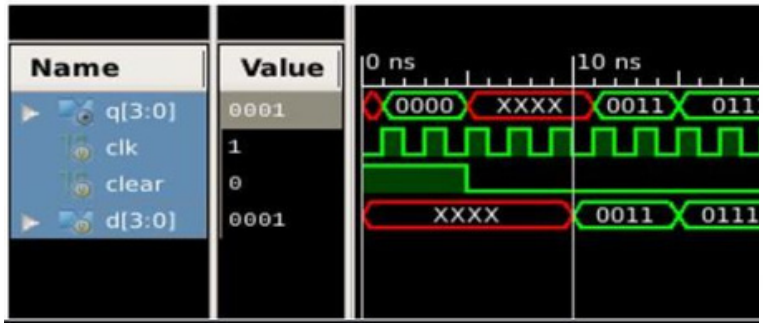


Day 12- Parallel In Parallel Out Shift Register

In "Parallel IN Parallel OUT", the inputs and the outputs come in a parallel way in the register. The inputs A0, A1, A2, and A3, are directly passed to the data inputs D0, D1, D2, and D3 of the respective flip flop. The bits of the binary input is loaded to the flip flops when the negative clock edge is applied. The clock pulse is required for loading all the bits. At the output side, the loaded bits appear.

- For parallel In data, Number of clock pulse needed are equal to 1.
- For parallel Out data, Number of Clock pulse needed are equal to 0.

RTL Code:



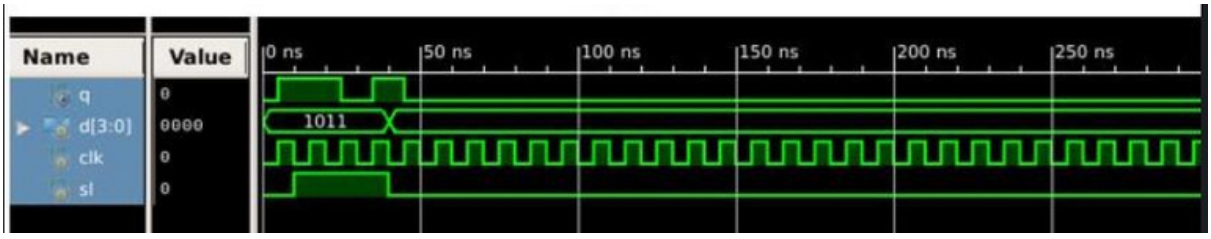

```
Data In = xxxx | Data out=xxxx
Data In = 0011 | Data out=xxxx
Data In = 0011 | Data out=0011
Data In = 0111 | Data out=0111
Data In = 1011 | Data out=0111
Data In = 1011 | Data out=1011
Data In = 1001 | Data out=1001
Data In = 1111 | Data out=1001
Data In = 1111 | Data out=1111
Data In = 0001 | Data out=0001
```

Day-13: Parallel In Serial Out Shift Register

A PISO shift register is a digital circuit that can accept parallel data and output serial data. It is made up of a succession of flip-flops, with each flip-flop capable of storing one bit of data. Unlike PIPO shift registers, which offer parallel input and output, a PISO shift register accepts data in parallel and outputs it sequentially, or serially. It is parallel data storage register. To store 'n' bit; number of clock pulse required is equal to 1.

• To provide serial data as output, number of clock pulse needed are equal to '(n-1)'.

```
module sl(a, b, sl ,q);
     input a,b,sl;
     output q;
     assign q=(~sl&b)|(sl&a);
endmodule
module dff(d,clk,q);
     input d,clk;
     output q;
     reg q=0;
     always @ (posedge clk)
     begin
     q<=d;
     end
endmodule

module piso_reg(d, clk, sl, q);
     input [3:0]d;
input clk,sl;
     output q;
     wire q1,q2,q3,d1,d2,d3;
     dff a(d[3],clk,q1);
     sl al(q1,d[2],sl,d1);
     dff b(d1,clk,q2);
     sl bl(q2,d[1],sl,d2);
     dff c(d2,clk,q3);
     sl cl(q3,d[0],sl,d3);
     dff dd(d3,clk,q);
endmodule
```

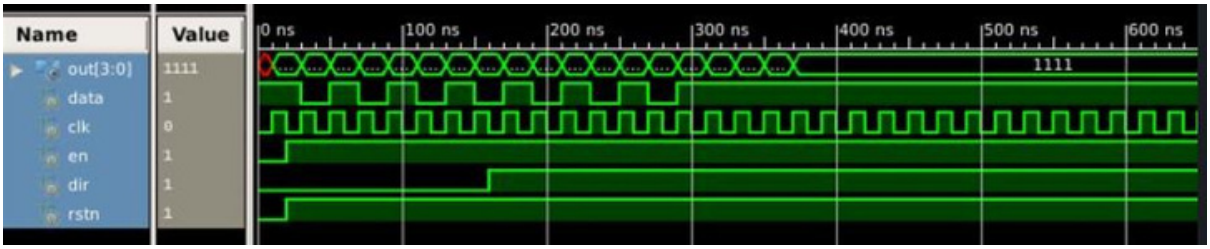


Day- 14 : Bidirectional Shift Register

Bidirectional shift registers are the storage devices capable of shifting the data either right or left, depending on the mode selected.Dir control line is made either low or high to opt for either left-shift or right-shift of the data bits, respectively.

RTL Code:

```
module bishift_reg  #(parameter MSB=4)
                    (  input d,
                       input clk,
                       input en,
                       input dir,
                       input rstn,
                       output reg [MSB-1:0] out);
    always @ (posedge clk)
       if (!rstn)
          out <= 0;
       else begin
          if (en)
             case (dir)
                0 :  out <= {out[MSB-2:0], d};
                1 :  out <= {d, out[MSB-1:1]};
             endcase
          else
             out <= out;
       end
endmodule
```

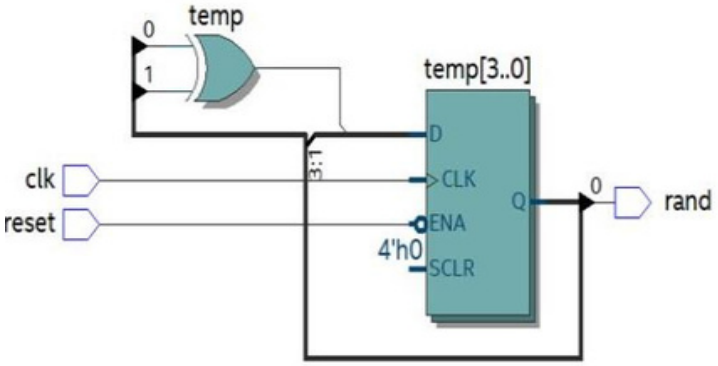


Day 15 : Psuedo Bit Random Sequence Generator

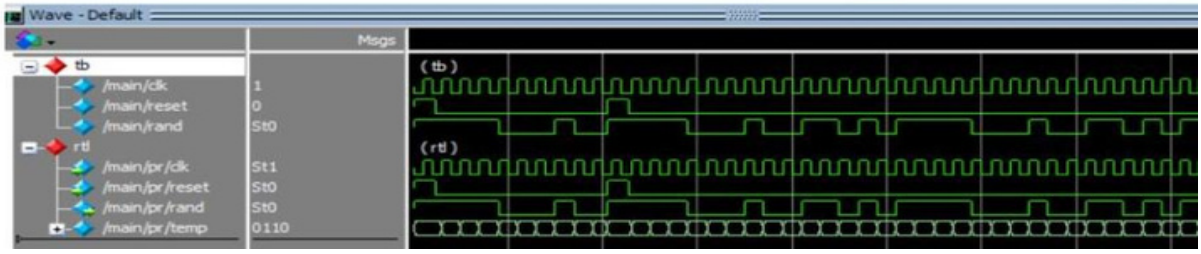
A PRBS is a periodic, deterministic signal consisting of a series of digital ones and zeros.A pseudo-random bit sequence (PRBS) is a pattern test that appears to be random, but is actually a predictable and repeatable sequence with a very long interval (i.e. billions of repeating bits), depending upon the pattern. A PRBS bit stream can be generated by using a linear feedback shift register (LFSR).

RTL Code:

```
module prbs_gen (rand, clk, reset);
input clk, reset;
output rand;
wire rand;
reg [3:0] temp;
always @ (posedge reset)
 begin
  temp <= 4'hf;
end
always @ (posedge clk) begin
 if (~reset) begin
  temp <= {temp[0]^temp[1],temp[3],temp[2],temp[1]};
 end
end
assign rand = temp[0];
endmodule
```




```
VSIM 9> run
#  Random bit : 1 | Reset=1 | Clock=0
#  Random bit : 1 | Reset=1 | Clock=1
#  Random bit : 1 | Reset=1 | Clock=0
#  Random bit : 1 | Reset=0 | Clock=0
#  Random bit : 1 | Reset=0 | Clock=1
#  Random bit : 1 | Reset=0 | Clock=0
#  Random bit : 1 | Reset=0 | Clock=1
#  Random bit : 1 | Reset=0 | Clock=0
#  Random bit : 1 | Reset=0 | Clock=1
#  Random bit : 1 | Reset=0 | Clock=0
#  Random bit : 0 | Reset=0 | Clock=1
#  Random bit : 0 | Reset=0 | Clock=0
#  Random bit : 0 | Reset=0 | Clock=1
#  Random bit : 0 | Reset=0 | Clock=0
#  Random bit : 0 | Reset=0 | Clock=1
#  Random bit : 0 | Reset=0 | Clock=0
#  Random bit : 1 | Reset=0 | Clock=1
#  Random bit : 1 | Reset=0 | Clock=0
#  Random bit : 0 | Reset=0 | Clock=1
#  Random bit : 0 | Reset=0 | Clock=0
#  Random bit : 0 | Reset=0 | Clock=1
```

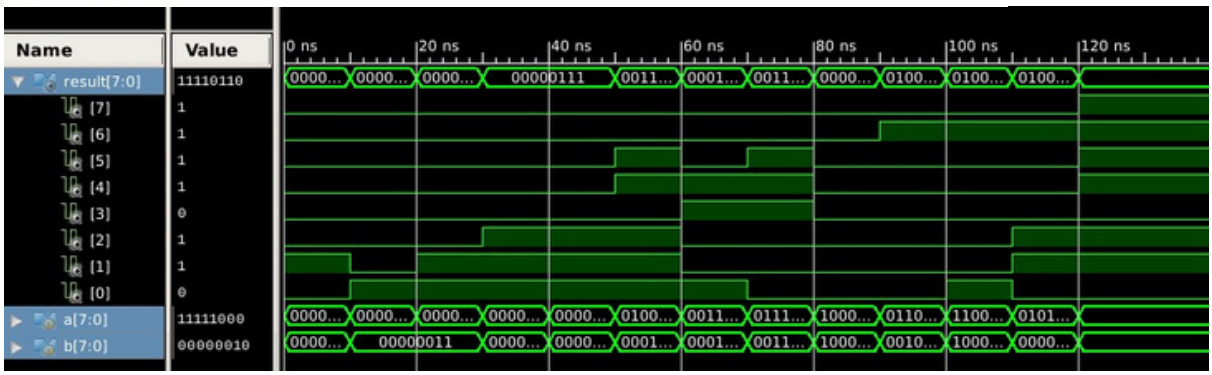


Day 16: 8-Bit Subtractor

The advantage of 2's complement over 1's complement is that if any carry remains after addition then there is no any need to add that carry in the end results.whereas in the case of 1's comement whenever carry remains after addition then it means that the result is positive and the final result will be obtained by adding 1 in the result.

RTL Code:

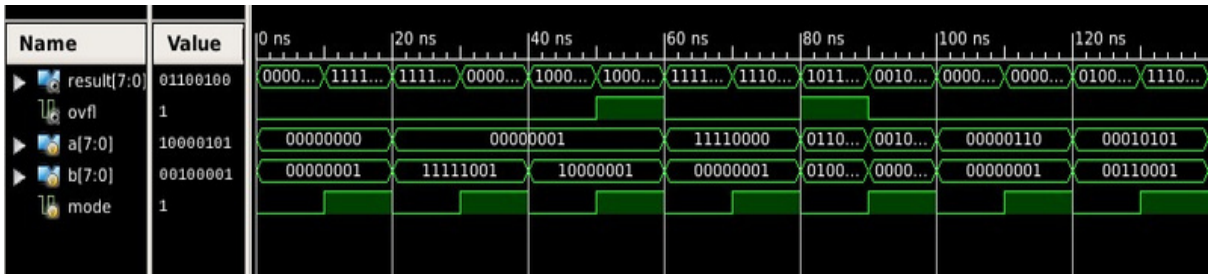
```
module subtract_8bit (a, b, result);
//define inputs and outputs
input [7:0] a, b;
output [7:0] result;
//variables used in always are declared as reg
reg [7:0] result;
//neg_b is used in the subtract operation
reg [7:0] neg_b;
always @ (a or b)
begin
 neg_b = ~b + 1;
result = a + neg_b;
end
endmodule
```



Day 17: 8-Bit Adder Subtractor

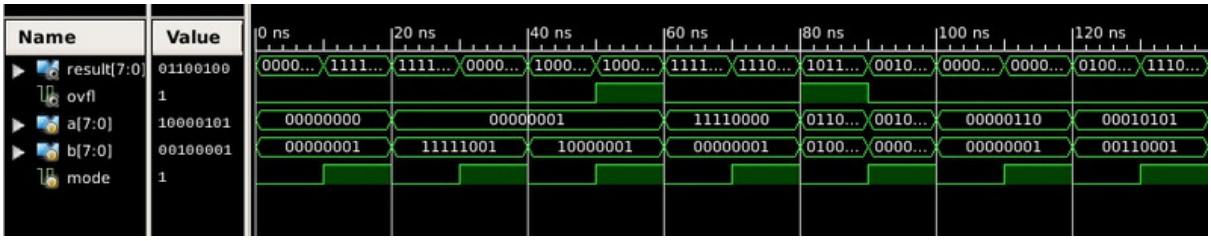
When the mode input (M) is at a low logic, i.e. '0', the circuit act as an adder and when the mode input is at a high logic, i.e. '1', the circuit act as a subtractor.

```
module adder_subtractor (a, b, mode, result, ovfl);
input [7:0] a, b; //define inputs and outputs
input mode;
output [7:0] result;
output ovfl;
//variables rslt and ovfl are left-hand side targets
//in the always block and are declared as type reg
reg [7:0] result;
reg ovfl;
wire [7:0] a, b; //since inputs default to wire
wire mode; //the type wire is not required
//neg_b = ~b + 1 specifies an internal register
reg [7:0] neg_b ;
always @ (a or b or mode)
begin
if (mode == 0) //add
begin
result = a + b;
ovfl =(a[7] & b[7] & ~result[7]) |
(~a[7] & ~b[7] & result[7]);
end
else //subtract
begin
neg_b=~b+1;
result = a + neg_b;
ovfl =(a[7] & neg_b[7] & ~result[7]) |
(~a[7] & ~neg_b[7] & result[7]);
end
endmodule
```



Day -18 : 4-Bit Multiplier

```
module multiplier_4bit(a,b,product);
input [3:0] a,b;
output [7:0] product;
 wire [3:0] m0;
 wire [4:0] m1;
 wire [5:0] m2;
 wire [6:0] m3;

 //middle terms
 wire [7:0] s1,s2,s3;

 assign m0={4{a[0]}}&b[3:0];
 assign m1={4{a[1]}}&b[3:0];
 assign m2={4{a[2]}}&b[3:0];
 assign m3={4{a[3]}}&b[3:0];

 assign s1= m0+(m1<<1);
 assign s2= s1+(m2<<1);
 assign s3= s2+(m3<<1);
 assign product=s3;
 endmodule
```

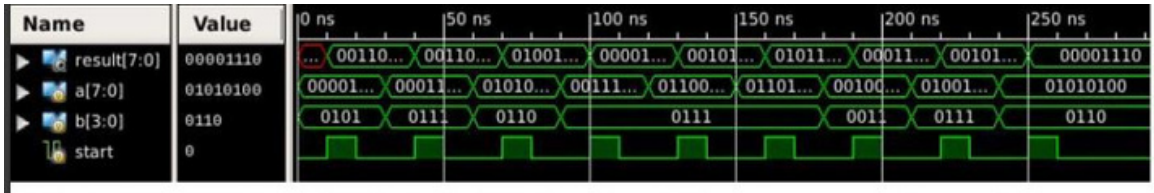


Day 19: Fixed Point Restoring Division

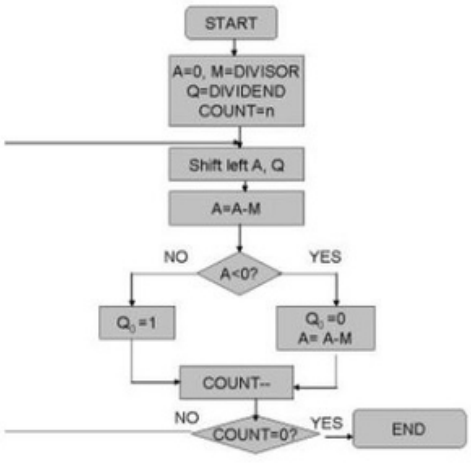
Restoring division examines the state of the carry-out when the dividend is subtracted from the partial remainder. This determines the relative magnitudes of the divisor and partial remainder. If the carry-out = 0, then the partial remainder is restored to its previous value by adding the divisor to the partial remainder. If the carry-out = 1, then there is no restore operation. The partial remainder (high-order half of the dividend) and the low-order half of the dividend are then shifted left one bit position and the process repeats for each bit in the divisor.

RTL Code:

```
module restoring_division(a,b,result,start);
input [7:0] a;
input [3:0] b;
output [7:0] result;
input start;
wire [3:0] b_bar;
reg [3:0] b_neg;
reg [7:0] result;
reg [3:0] count;
assign b_bar=~b;
always @(b_bar)
b_neg=b_bar+1;
always @(posedge start)
begin
result=a;
count =4'b0100;
if ((a!=0) && (b!=0))
while(count)
begin
result=result<<1;
result={(result[7:4]+b_neg),result[3:0]};
if(result[7]==1)
begin
result= {(result[7:4] + b), result[3:1], 1'b0};
count=count-1;
end
else
begin
result={result[7:1],1'b1};
count=count-1;
end
end
end
endmodule
```



Dividend = 00011000, Divisor = 0111, Quotient = 0010, Remainder = 0011
Dividend = 00011000, Divisor = 0111, Quotient = 0011, Remainder = 0011
Dividend = 01010010, Divisor = 0110, Quotient = 0011, Remainder = 0011
Dividend = 01010010, Divisor = 0110, Quotient = 1101, Remainder = 0100
Dividend = 00111000, Divisor = 0111, Quotient = 1101, Remainder = 0100
Dividend = 00111000, Divisor = 0111, Quotient = 1000, Remainder = 0000
Dividend = 01100100, Divisor = 0111, Quotient = 1000, Remainder = 0000
Dividend = 01100100, Divisor = 0111, Quotient = 1110, Remainder = 0010
Dividend = 01101110, Divisor = 0111, Quotient = 1110, Remainder = 0010
Dividend = 01101110, Divisor = 0111, Quotient = 1111, Remainder = 0101
Dividend = 00100101, Divisor = 0011, Quotient = 1111, Remainder = 0101
Dividend = 00100101, Divisor = 0011, Quotient = 1100, Remainder = 0001
Dividend = 01001000, Divisor = 0111, Quotient = 1100, Remainder = 0001
Dividend = 01001000, Divisor = 0111, Quotient = 1010, Remainder = 0010
Dividend = 01010100, Divisor = 0110, Quotient = 1010, Remainder = 0010
Dividend = 01010100, Divisor = 0110, Quotient = 1110, Remainder = 0000

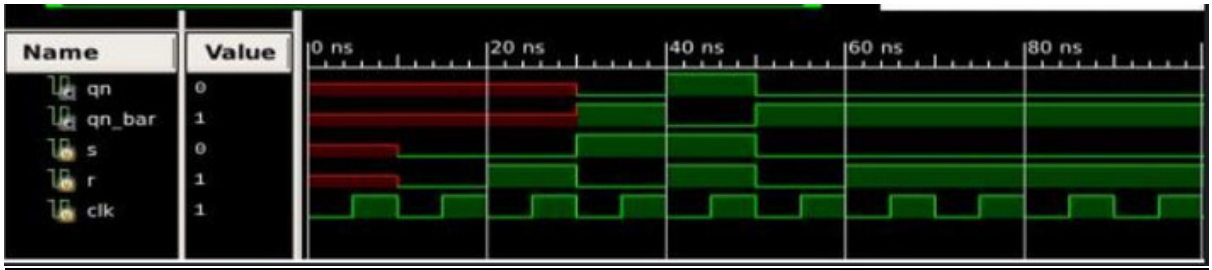


Day- 20: Master Slave JK Flip Flop

Master slave configuration removes the possibility of race around condition from JK flip flop when we put both J and K as 1.
In normal JK flip flop when we put both J and K as 1 and as clock level becomes high the output changes continuously from 0 to 1 & 1 to 0 till the clock is high.However in master slave configuration when we put both J & K as 1 the present output will be compliment of past output i.e the race around is changes to toggling. The output in master slave changes when negative edge of clock reaches i.e negative triggers at negative edge which also removes problem due to propagation delay. This toggling effect helps in designing counters.

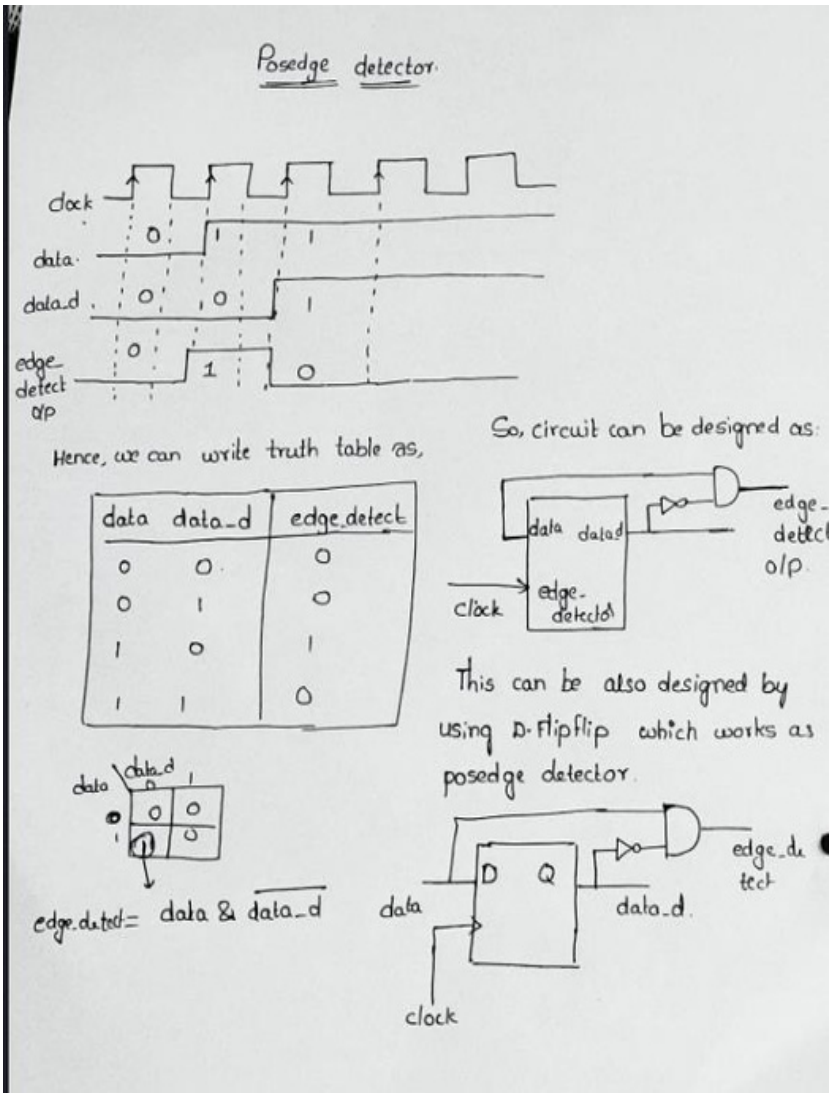
RTL Code:

```
module jk_ff(j,k,clk,q,q_bar);
input j,k,clk;
output  q,q_bar;
reg q;
assign q_bar= ~q;
always @(posedge clk)
begin
case({j,k})
2'b00: q<=q;
2'b01: q<=0;
2'b10: q<=1;
2'b11: q<=~q;
endcase
end
endmodule

module master_slave(s,r,clk,qn,qn_bar,);
input s,r,clk;
output qn,qn_bar;
wire mq;
wire mq_bar;
wire mclk;
assign mclk= ~clk;
jk_ff Master(s,r,clk,mq,mq_bar);
jk_ff Slave(mq,mq_bar,mclk,qn,qn_bar);
endmodule
```

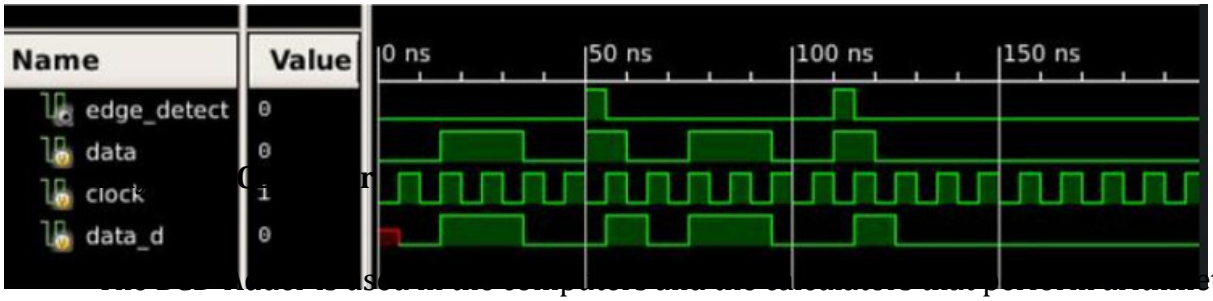


Day 21- Positive Edge Detector

A positive edge Detector will send out a pulse. whenever the signal it is monitoring changes from 0 to 1. The idea behind a Positive Edge Detector is to delay original clock signal by one clock cycle, take its inverse and perform a logical AND with the original signal.



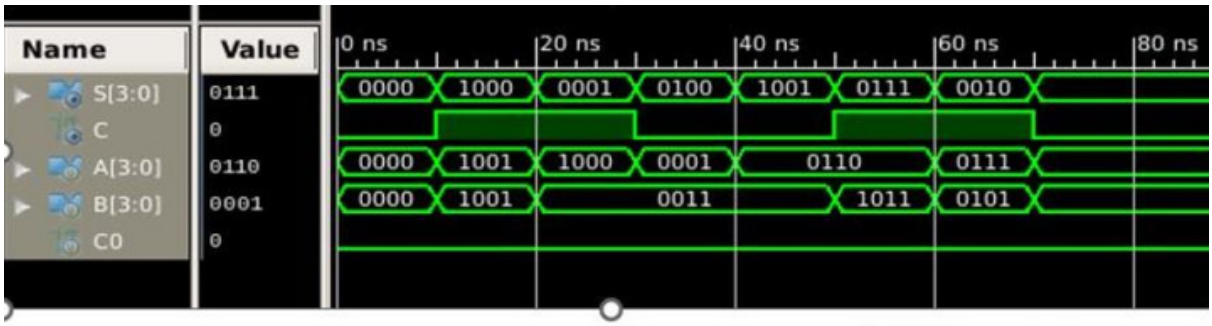

```
//////////////////////////////////////////////////////////////////
module edge_detector(
    input  data,
     input clock,
    output edge_detect
    );
        reg data_d;

    always @ (posedge clock) begin
        data_d <= data;
    end
    assign edge_detect = data & ~data_d;
endmodule
```



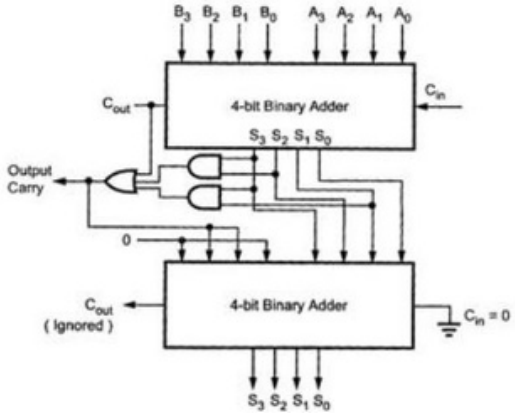
C[illegible]r

[illegible]tic operation directly in the decimal number system. The BCD-Adder accepts the binary-coded form of decimal numbers. The output will vary from 0 to 18 if we are not considering the carry from the previous sum. But if we are considering the carry, then the maximum value of output will be 19 (i.e. 9+9+1 = 19). When we are simply adding

A and B, then we get the binary sum. Here, to get the output in BCD form, we will use BCD Adder.



A=0000| B=0000| Cin=0|BCD Sum=0000|Cout=0
A=1001| B=1001| Cin=0|BCD Sum=1000|Cout=1
A=1000| B=0011| Cin=0|BCD Sum=0001|Cout=1
A=0001| B=0011| Cin=0|BCD Sum=0100|Cout=0
A=0110| B=0011| Cin=0|BCD Sum=1001|Cout=0
A=0110| B=1011| Cin=0|BCD Sum=0111|Cout=1
A=0111| B=0101| Cin=0|BCD Sum=0010|Cout=1
A=0110| B=0001| Cin=0|BCD Sum=0111|Cout=0



RTL Code:

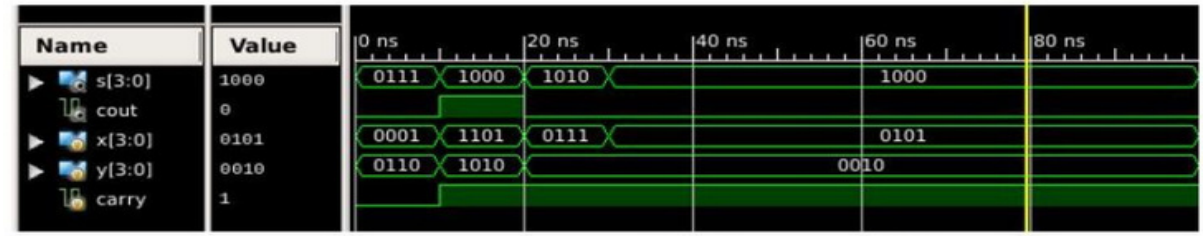
```
module full_adder(a,b,c,s,co);
input a,b,c;
output s,co;
assign s=a^b^c;
assign co=(a&b)|(b&c)|(c&a);
endmodule

module four_bit_adder(x,y,cin,sum,cout);
input [3:0] x,y;
input cin;
output [3:0] sum;
output cout;
wire c0,c1,c2;

full_adder fa0(.a(x[0]),.b(y[0]),.c(cin),.s(sum[0]),.co(c0));
full_adder fa1(.a(x[1]),.b(y[1]),.c(c0),.s(sum[1]),.co(c1));
full_adder fa2(.a(x[2]),.b(y[2]),.c(c1),.s(sum[2]),.co(c2));
full_adder fa3(.a(x[3]),.b(y[3]),.c(c2),.s(sum[3]),.co(cout));
endmodule
module BCD_adder(S, C, A, B, C0);
input [3:0] A, B;
input C0;
output [3:0] S;
output C;
wire C1, C2, C3, C4, C5;
wire [3:0]X, Z;
and (C1, Z[3], Z[2]);
and (C2, Z[3], Z[1]);
or (C, C3, C1,C2);
xor (C5, C, C);
assign X[2] = C;
assign X[1] = C;
assign X[3] = C5;
assign X[0] = C5;
//four bit adder body from instance of full_adder
four_bit_adder F_1 ( A, B, C0,Z,C3);
four_bit_adder F_2 (X, Z, C0,S,C4);
endmodule
```

Day 23- 4-Bit Carry Select Adder

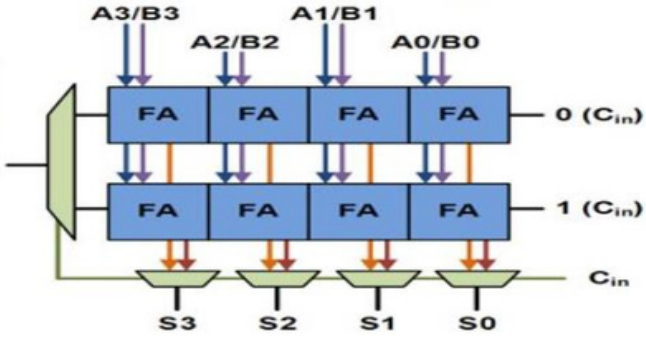
A carry select adder is an arithmetic combinational logic circuit which adds two N-bit binary numbers and outputs their N-bit binary sum and a 1-bit carry.The idea behind a N-bit carry select adder is to avoid propagating the the carry from bit to bit in sequence. If we have two adders in parallel: one with a carry input of 0, the other with a carry input of 1, then we could use the actual carry input generated to select between the outputs of the two parallel adders. This means all adders could be performing their calculations in parallel. Having two adders for each result bit is quite wasteful so we could configure the N-bit adder to use 2*N/M-1 M-bit ripple carry adders in parallel. Note that the adder for the least significant bits will always have a carry input of 0 so no parallel addition is needed in this case.

RTL Code:

```
module full_adder( A, B, Cin, S, Cout);
 input wire A, B, Cin;
 output reg S, Cout;
 always @(A or B or Cin)
  begin
   S = A ^ B ^ Cin;
   Cout = A&B | B&Cin | Cin&A ;
  end
endmodule
module mux(A,B,S,Y);
input A,B,S;
output reg Y;
always@(A,B,S)
begin
Y=~S&A| S&B;
end
endmodule
module carry_select (x,y,carry,s,cout);
input [3:0]x,y;
input carry;
output [3:0]s;
output cout;
wire w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16;
full_adder fao (x[0],y[0],1'b0,w1,w2);
full_adder fa1 (x[1],y[1],w2,w3,w4);
full_adder fa2 (x[2],y[2],w4,w5,w6);
full_adder fa3 (x[3],y[3],w6,w7,w8);
full_adder fa4 (x[0],y[0],1'b1,w9,w10);
full_adder fa5 (x[1],y[1],w10,w11,w12);
full_adder fa6 (x[2],y[2],w12,w13,w14);
full_adder fa7 (x[3],y[3],w14,w15,w16);
mux mu0(w1,w9,carry,s[0]);
mux mu1(w3,w11,carry,s[1]);
mux mu2(w5,w13,carry,s[2]);
mux mu3(w7,w15,carry,s[3]);
mux mu4(w8,w16,carry,cout);
endmodule
```




```
VSIM 8> run
# x=0001,y=0110,carry=0,s=0111,cout=0
# x=1101,y=1010,carry=1,s=1000,cout=1
# x=0111,y=0010,carry=1,s=1010,cout=0
# x=0101,y=0010,carry=1,s=1000,cout=0
```

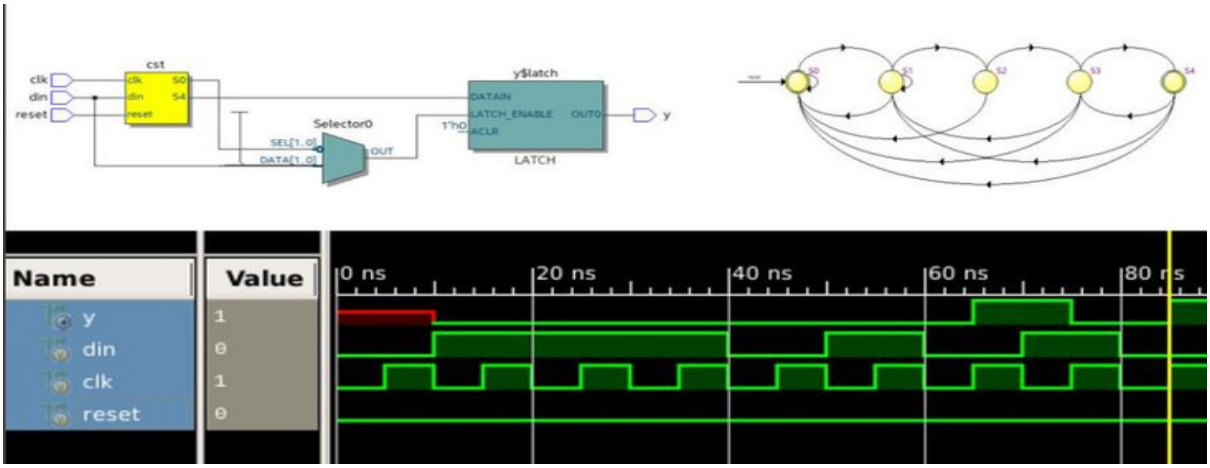


Day 24-Moore FSM which detects 1010 Sequence

Moore machine is a finite-state machine whose current output values are determined only by its current state. This is in contrast to a Mealy machine, whose output values are determined both by its current state and by the values of its inputs. Like other finite state machines, in Moore machines, the input typically influences the next state.
Why Moore is better than Mealy?
A Mealy Machine changes its output on the basis of its present state and current input. A Moore Machine's output depends only on the current state. It does not depend on the current input.Mealy machines react faster to inputs. They generally react in the same clock cycle.

```
module moore_fsm(din, reset, clk, y);
input din;
input clk;
input reset;
output reg y;
reg [2:0] cst, nst;
localparam S0 = 3'b000,
           S1 = 3'b001,
           S2 = 3'b010,
           S3 = 3'b100,
             S4 = 3'b101;
always @(cst or din)
 begin
 case (cst)
   S0: if (din == 1'b1)
         begin
        nst = S1;
         y=1'b0;
         end
     else nst = cst;
   S1: if (din == 1'b0)
         begin
       nst = S2;
         y=1'b0;
         end
      else
         begin
         nst = cst;
         y=1'b0;
         end
```

```
    S2: if (din == 1'b1)
            begin
           nst = S3;
            y=1'b0;
            end
              else
            begin
            nst = S0;
            y=1'b0;
            end
    S3: if (din == 1'b0)
            begin
           nst = S4;
            y=1'b0;
            end
         else
            begin
            nst = S1;
            y=1'b0;
            end
 S4: if (din == 1'b0)
            begin
           nst = S1;
            y=1'b1;
            end
            else
            begin
            nst = S3;
            y=1'b1;
            end
     default: nst = S0;
   endcase
end
 always@(posedge clk)
begin
            if (reset)
            cst <= S0;
            else
              cst <= nst;
end
 endmodule
```

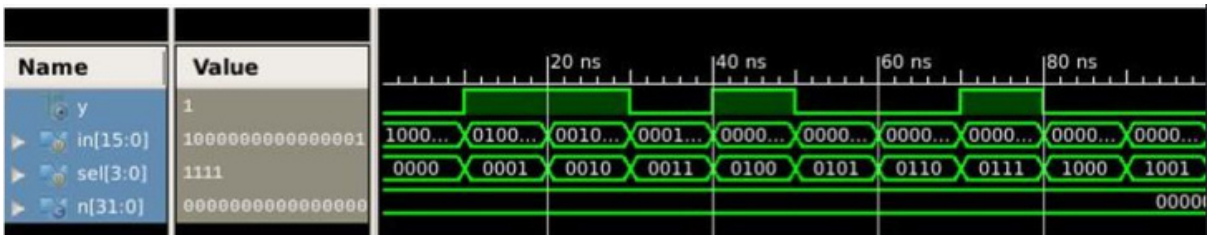


Day 25- N:1 Multiplexer

Multiplexers are also known as “Data n selector, parallel to serial convertor, many to one circuit, universal logic circuit”. Multiplexers are mainly used to increase amount of the data that can be sent over the network within certain amount of time and bandwidth.

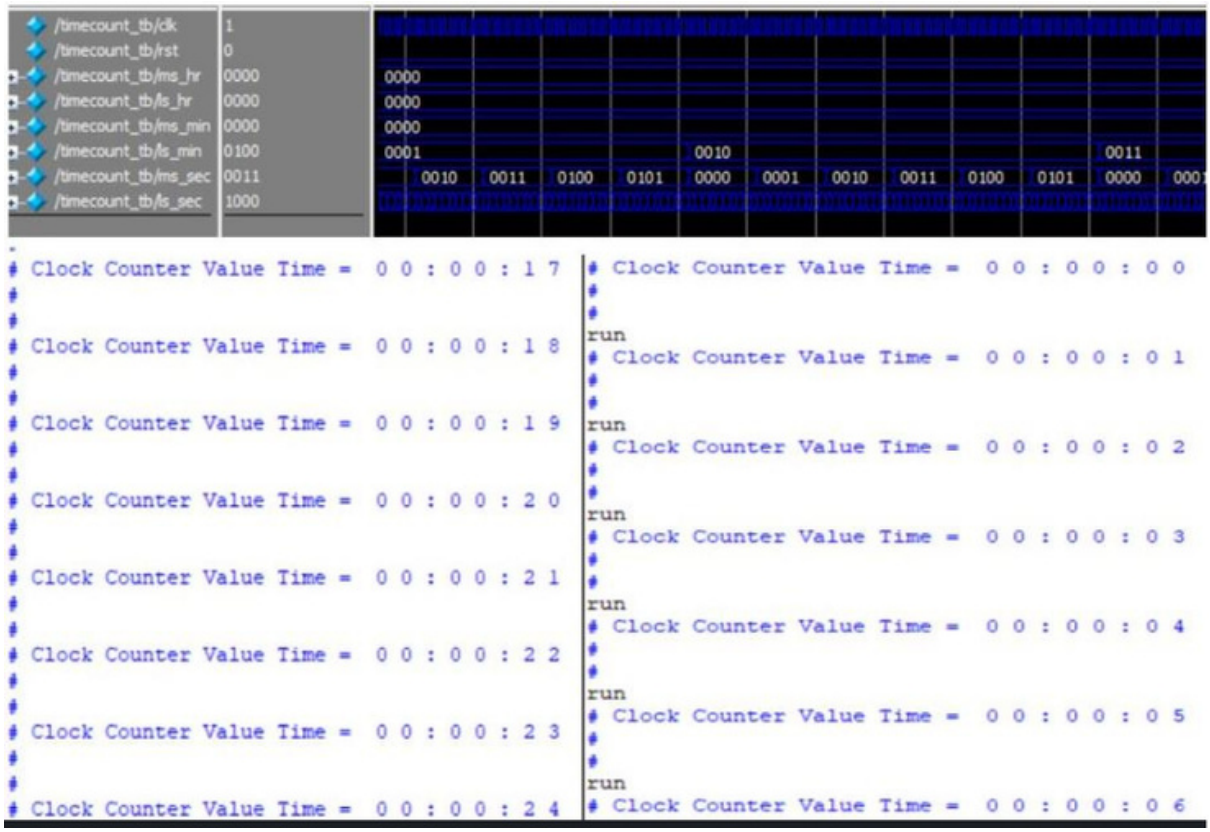
RTL Code:

```
module n_1mux(in,sel,y);
parameter n=16;
input [n-1:0] in;
input 3:0 sel;
output reg y;

always @(in or sel)
begin
case(sel)
4'b0000: y= in[0];
4'b0001: y= in[1];
4'b0010: y= in[2];
4'b0011: y= in[3];
4'b0100: y= in[4];
4'b0101: y= in[5];
4'b0110: y= in[6];
4'b0111: y= in[7];
4'b1000: y= in[8];
4'b1001: y= in[9];
4'b1010: y= in[10];
4'b1011: y= in[11];
4'b1100: y= in[12];
4'b1101: y= in[13];
4'b1110: y= in[14];
4'b1111: y= in[n-1];
default: y=4'b0000;
endcase
end
endmodule
```



Day 26: Write RTL code for a BCD counter that displays time in a 24hr format as shown here HH:MM:SS with the following specification details:

I. Input Clock Frequency is 1Hz
II. Inputs: clock (posedge), reset(Active high synchronous)
III. Outputs: ms_hr<3:0>, ls_hr<3:0>, ms_min<3:0>, ls_min<3:0>, ms_sec<3:0>, ls_sec<3:0>

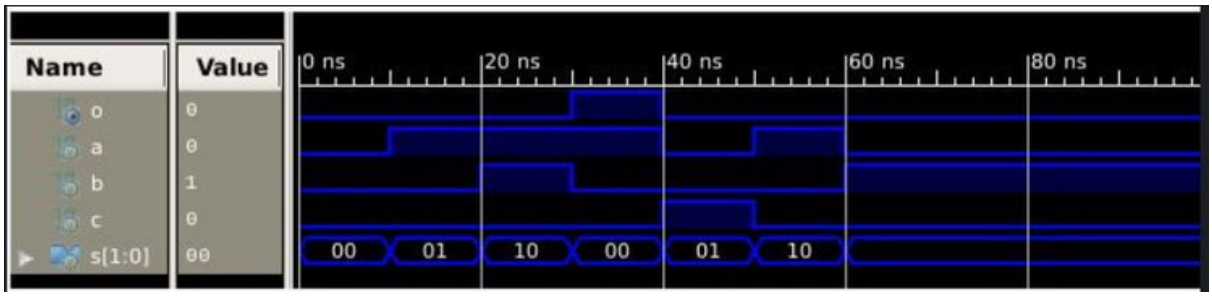


Day 27 : 3-to-1 1-bit MUX with a 1-bit latch

A 3:1 mux has 2 select lines and 3 inputs. As a mux with 2 select lines can represent at max 4 inputs, a 3:1 mux repeats some inputs for 2 combinations.

RTL Code:

```
module mux (a, b, c,  s, o);
        input         a, b, c ;
        input  [1:0] s;
        output        o;
        reg           o;
        always @(a or b or c  or s)
        begin
           if (s == 2'b00)
              o = a;
           else if (s == 2'b01)
              o = b;
           else if (s == 2'b10)
              o = c;
            else
            o=c;
        end
        endmodule
```



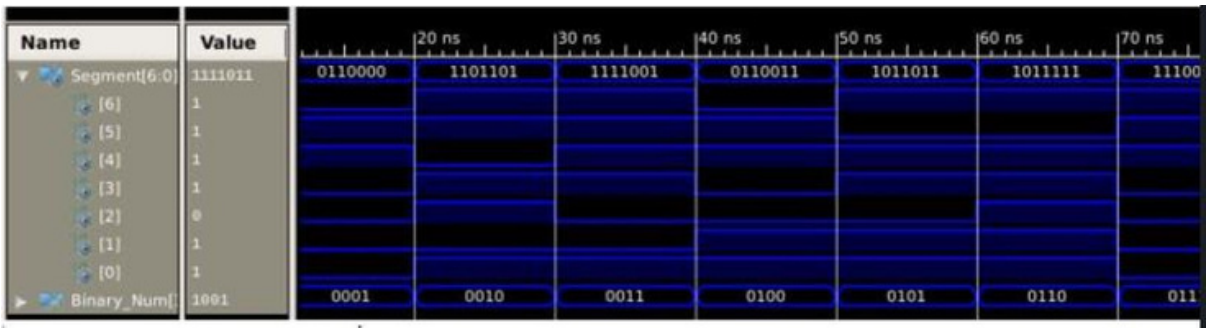
Day 28: BCD to Seven Segment Display

A Seven-Segment Display is an indicator commonly used by FPGA designers to show information to the user. Code to convert from binary to seven-segment display compatible can be done easily in Verilog. There are many applications that can require the use of one or more seven-segment displays such as:

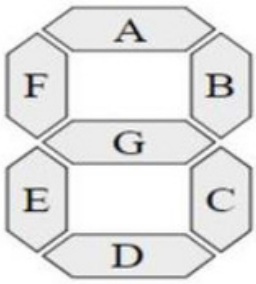
-->Alarm Clock
-->Stop Watch
-->Button Count Indicator
-->Voltage Measurements (from Analog to Digital Converter)

RTL Code:

```
module Binary_7Segment
  (

   input [3:0] Binary_Num,
   output [6:0]     Segment

   );

  reg [6:0] Segment;
  always @(Binary_Num)
    begin
      case (Binary_Num)
            0 : Segment = 7'b1111110;
            1 : Segment = 7'b0110000;
            2 : Segment = 7'b1101101;
            3 : Segment = 7'b1111001;
            4 : Segment = 7'b0110011;
            5 : Segment = 7'b1011011;
            6 : Segment = 7'b1011111;
            7 : Segment = 7'b1110000;
            8 : Segment = 7'b1111111;
            9 : Segment = 7'b1111011;
            //switch off 7 segment character
            default : Segment = 7'b000000;
      endcase
    end


endmodule // Binary_To_7Segment
```



Binary Num=0000 | ABCDEFG=1111110
Binary Num=0001 | ABCDEFG=0110000
Binary Num=0010 | ABCDEFG=1101101
Binary Num=0011 | ABCDEFG=1111001
Binary Num=0100 | ABCDEFG=0110011
Binary Num=0101 | ABCDEFG=1011011
Binary Num=0110 | ABCDEFG=1011111
Binary Num=0111 | ABCDEFG=1110000
Binary Num=1000 | ABCDEFG=1111111
Binary Num=1001 | ABCDEFG=1111011

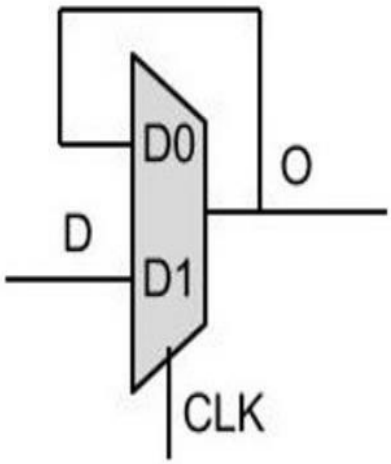


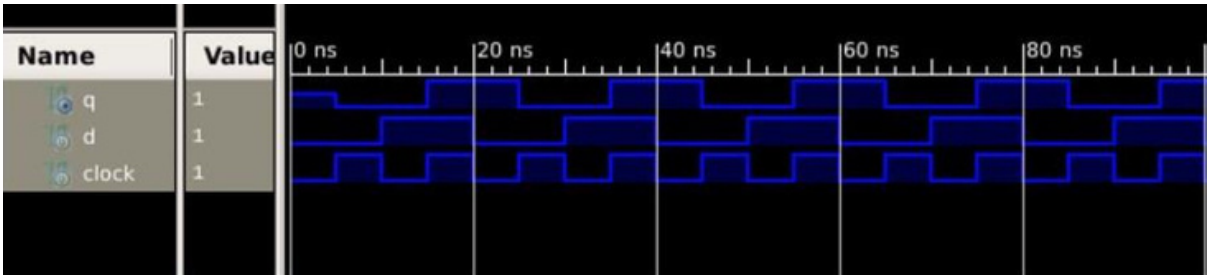
Day 29: D Latch using 2:1 Mux

To build a positive level sensitive latch from a multiplexer, short the output with IN0 pin of the multiplexer and connect data input to IN1 and Clock input to SEL pin of multiplexer. A negative level latch can also be built similarly.

RTL Code:

```
module mux(a,b,s,y);
input a,b,s;
output reg  y;
always @(a or b or s)
begin
  y= (~s&a) | (s&b);
end
endmodule


module d_latch(d,clock,q);
input d,clock;
inout q;

mux mux1(.a(q),.b(d),.s(clock),.y(q));

endmodule
```





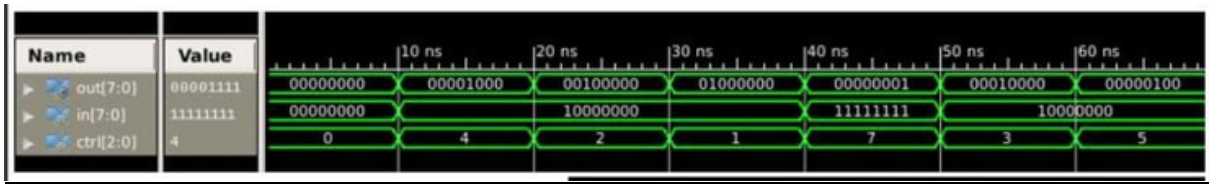
Day 30: 8-Bit Barrel Shifter

A barrel shifter is a logic circuit for shifting a word by a varying amount. Its has a control input that specifies the number of bit positions that it shifts by. A barrel shifter is implemented with a sequence of shift multiplexers, each shifting a word by 2k bit positions for different values of k .A barrel shifter is able to complete the shift in a single clock cycle, giving it a great advantage over a simple shifter which can shift n bits in n clock cycles. It is used in conjunction with a processor's arithmetic logic unit (ALU) or otherwise embedded in the ALU itself.

RTL Code:

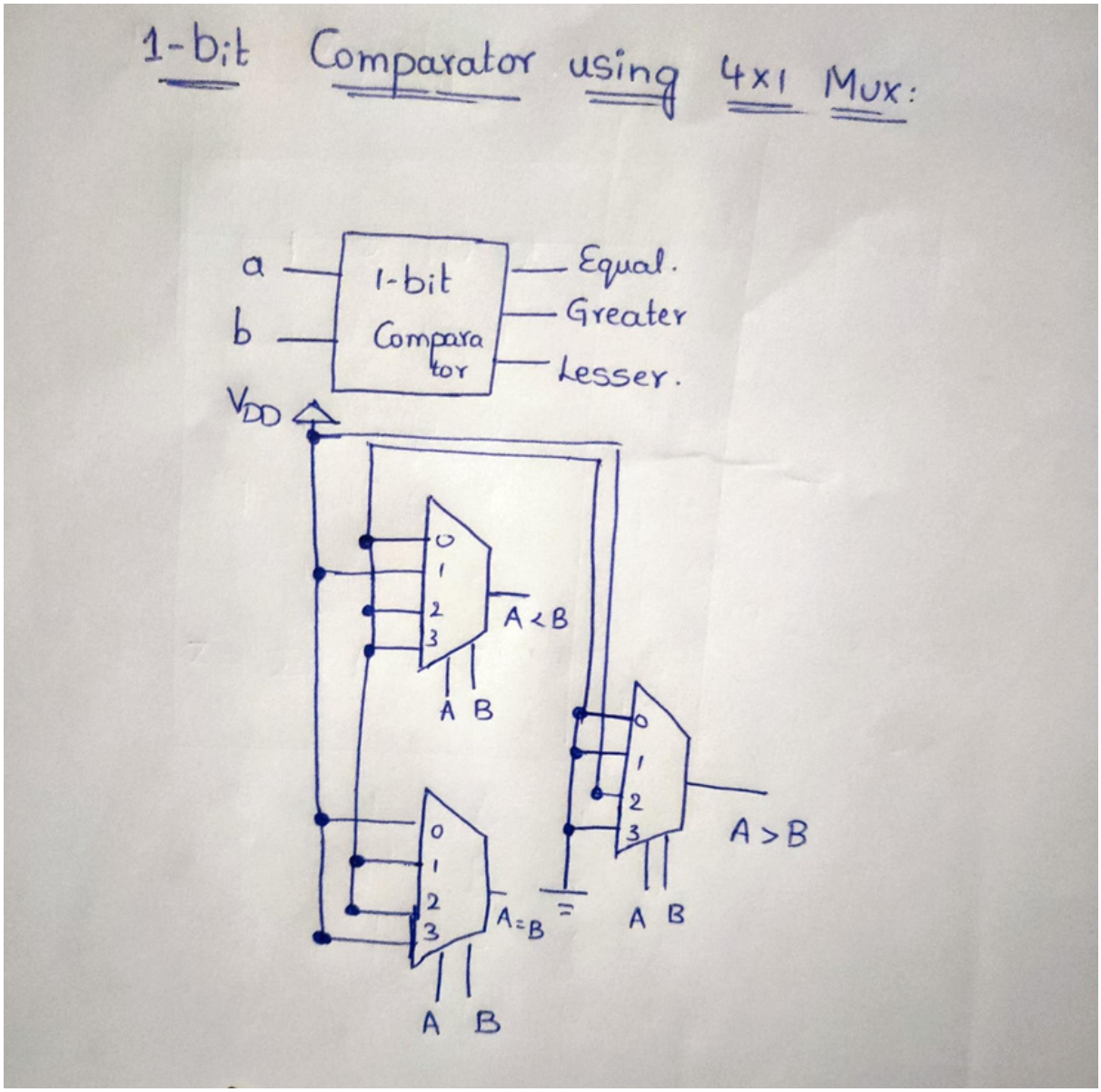
```
module barrelshifter(in, ctrl, out);
  input  [7:0] in;
  input [2:0] ctrl;
  output [7:0] out;
  wire [7:0] x,y;

//4bit shift right
mux2X1  ins_17 (.in0(in[7]),.in1(1'b0),.sel(ctrl[2]),.out(x[7]));
mux2X1  ins_16 (.in0(in[6]),.in1(1'b0),.sel(ctrl[2]),.out(x[6]));
mux2X1  ins_15 (.in0(in[5]),.in1(1'b0),.sel(ctrl[2]),.out(x[5]));
mux2X1  ins_14 (.in0(in[4]),.in1(1'b0),.sel(ctrl[2]),.out(x[4]));
mux2X1  ins_13 (.in0(in[3]),.in1(in[7]),.sel(ctrl[2]),.out(x[3]));
mux2X1  ins_12 (.in0(in[2]),.in1(in[6]),.sel(ctrl[2]),.out(x[2]));
mux2X1  ins_11 (.in0(in[1]),.in1(in[5]),.sel(ctrl[2]),.out(x[1]));
mux2X1  ins_10 (.in0(in[0]),.in1(in[4]),.sel(ctrl[2]),.out(x[0]));

//2 bit shift right

mux2X1  ins_27 (.in0(x[7]),.in1(1'b0),.sel(ctrl[1]),.out(y[7]));
mux2X1  ins_26 (.in0(x[6]),.in1(1'b0),.sel(ctrl[1]),.out(y[6]));
mux2X1  ins_25 (.in0(x[5]),.in1(x[7]),.sel(ctrl[1]),.out(y[5]));
mux2X1  ins_24 (.in0(x[4]),.in1(x[6]),.sel(ctrl[1]),.out(y[4]));
mux2X1  ins_23 (.in0(x[3]),.in1(x[5]),.sel(ctrl[1]),.out(y[3]));
mux2X1  ins_22 (.in0(x[2]),.in1(x[4]),.sel(ctrl[1]),.out(y[2]));
mux2X1  ins_21 (.in0(x[1]),.in1(x[3]),.sel(ctrl[1]),.out(y[1]));
mux2X1  ins_20 (.in0(x[0]),.in1(x[2]),.sel(ctrl[1]),.out(y[0]));

//1 bit shift right
mux2X1  ins_07 (.in0(y[7]),.in1(1'b0),.sel(ctrl[0]),.out(out[7]));
mux2X1  ins_06 (.in0(y[6]),.in1(y[7]),.sel(ctrl[0]),.out(out[6]));
mux2X1  ins_05 (.in0(y[5]),.in1(y[6]),.sel(ctrl[0]),.out(out[5]));
mux2X1  ins_04 (.in0(y[4]),.in1(y[5]),.sel(ctrl[0]),.out(out[4]));
mux2X1  ins_03 (.in0(y[3]),.in1(y[4]),.sel(ctrl[0]),.out(out[3]));
mux2X1  ins_02 (.in0(y[2]),.in1(y[3]),.sel(ctrl[0]),.out(out[2]));
mux2X1  ins_01 (.in0(y[1]),.in1(y[2]),.sel(ctrl[0]),.out(out[1]));
mux2X1  ins_00 (.in0(y[0]),.in1(y[1]),.sel(ctrl[0]),.out(out[0]));

endmodule
```

```
module mux2X1( in0,in1,sel,out);
input in0,in1;
input sel;
output out;
assign out=(sel)?in1:in0;
endmodule
```



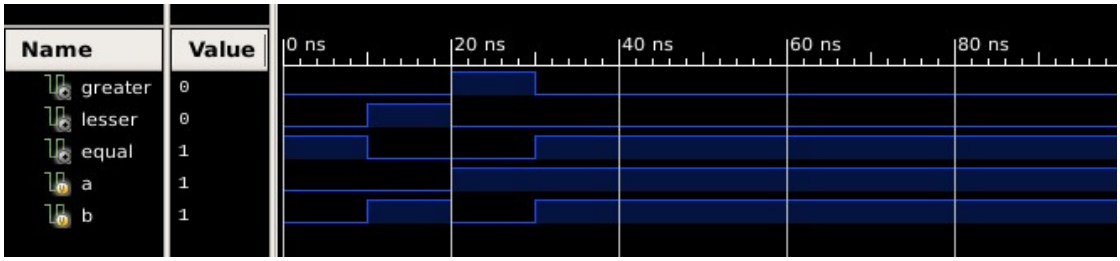
Day 31: 1-Bit Comparator using 4X1 Mux

IMPLEMENTATION:



RTL Code:

```
module mux4X1( in0,in1,in2,in3,s1,s0,out);
input in0,in1,in2,in3;
input   s1,s0;
output  out;
assign out = s1 ? (s0 ? in3 : in2) : (s0 ? in1 : in0);

endmodule


module comparator(a,b,greater,lesser,equal);
input a,b;
output  greater,lesser,equal;

mux4X1 mux1(1'b0,1'b1,1'b0,1'b0,a,b,lesser);
mux4X1 mux2(1'b1,1'b0,1'b0,1'b1,a,b,equal);
mux4X1 mux3(1'b0,1'b0,1'b1,1'b0,a,b,greater);

endmodule
```

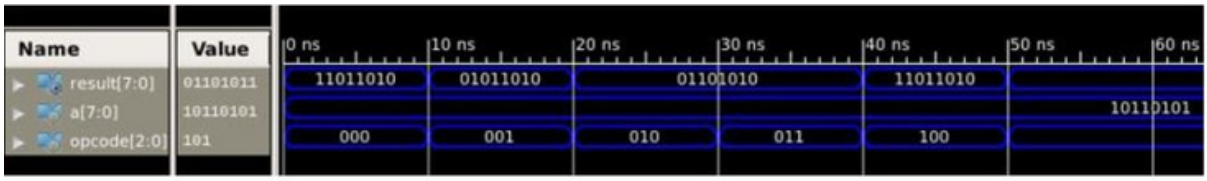


Day 32- Logical, Algebraic, and Rotate Shift Operations

Shift registers that perform the operations of shift left logical (SLL), shift left algebraic (SLA), shift right logical (SRL), shift right algebraic (SRA), rotate left (ROL), and rotate right (ROR) will be Coded.

RTL Code:

```
module shift_rotate (a, opcode, result);
//list inputs and outputs
input [7:0] a;
input [2:0] opcode;
output [7:0] result;

//specify wire for input and reg for output
wire [7:0] a;
wire [2:0] opcode;
reg [7:0] result;


//define the opcodes
parameter sra_op = 3'b000, //shift right arithmetic
srl_op = 3'b001,           //Shift right logical
sla_op = 3'b010,           //Shift left algebraic
sll_op = 3'b011,           //Shift left logical
ror_op = 3'b100,           //rotate right
rol_op = 3'b101;           //rotate left

//execute the operations
always @ (a or opcode)
begin
case (opcode)
sra_op : result = {a[7], a[7], a[6], a[5],
 a[4], a[3], a[2], a[1]};
srl_op : result = a >> 1;
sla_op : result = {a[6], a[5], a[4], a[3],
 a[2], a[1], a[0], 1'b0};
 sll_op : result = a << 1;
ror_op : result = {a[0], a[7], a[6], a[5],
a[4], a[3], a[2], a[1]};
rol_op : result = {a[6], a[5], a[4], a[3],
a[2], a[1], a[0], a[7]};
default : result = 0;
endcase
end
endmodule
```



Day 33: ALU

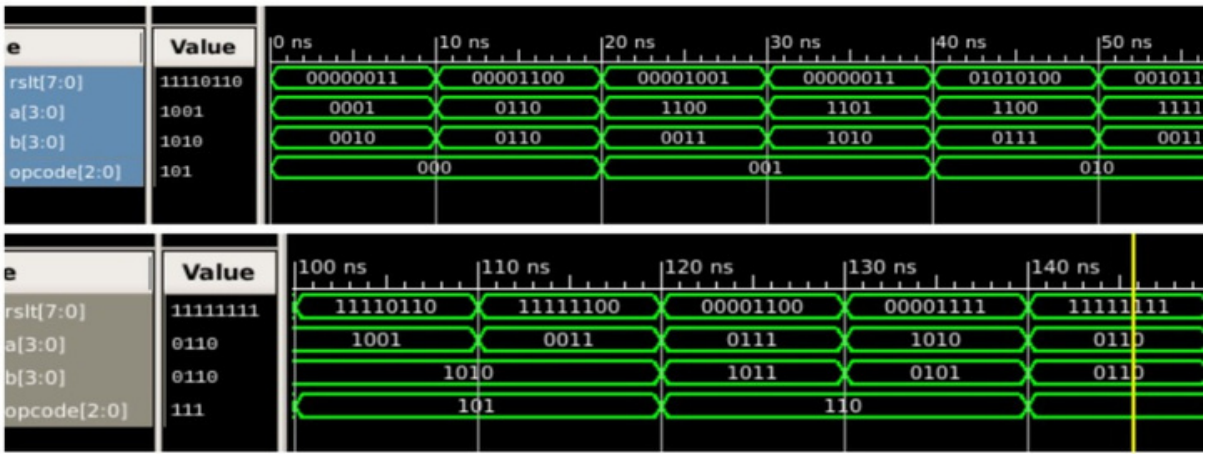
Arithmetic and logic units (ALUs) perform the arithmetic operations and logical operations. Here Opcode decides the type of operation to be performed on the operands.If the opocde is

0-Addition

1- Subtraction
2-Multiplication
3-And operation
4-Or operation
5-Not operation
6-Xor operation
7-Xnor operation

RTL Code:

```
module alu (a, b, opcode, rslt);
//define inputs and output
input [3:0] a, b;
input [2:0] opcode;
output [7:0] rslt;
//the rslt is left-hand side target in always
//and is declared as type reg
reg [7:0] rslt;
//define operation codes
//parameter defines a constant
parameter add_op = 3'b000,
sub_op = 3'b001,
mul_op = 3'b010,
and_op = 3'b011,
or_op = 3'b100,
not_op = 3'b101, //negation
xor_op = 3'b110,
xnor_op = 3'b111;
//perform the operations
always @ (a or b or opcode)
begin
case (opcode)
add_op: rslt = a + b;
sub_op: rslt = a - b;
mul_op: rslt = a * b;
and_op: rslt = a & b; //also ab
or_op: rslt = a | b;
not_op: rslt = ~a; //also ~b
xor_op: rslt = a ^ b;
xnor_op: rslt = ~(a ^ b);
endcase
end
endmodule
```

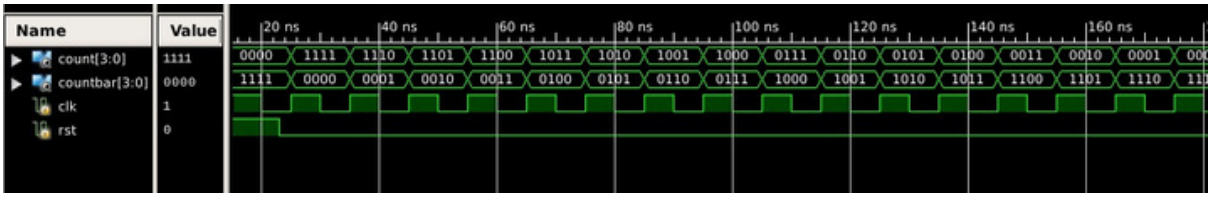


Day-34: 4-Bit Asynchronous Down Counter

In the asynchronous counter, an external clock pulse is provided for only the first flip flop, thereafter the output of the 1st FF acts as a clock pulse for the second FF and so on. In the case of synchronous FFs, all the flip flops are triggered simultaneously by an external clock pulse.

RTL code:

```
module dff(q,qbar,clk,rst,d);
    output reg q;
    output qbar;
    input clk, rst;
    input d;

    assign qbar = ~q;

    always @(posedge clk, posedge rst)
    begin
        if (rst)
            q <= 0;
        else
            q <= d;
    end
endmodule

module async_counter(count,countbar,clk,rst);
  input clk, rst;
  output [3:0] count, countbar;
    dff dff1(count[0],countbar[0],clk     ,rst,countbar[0]);
  dff dff2(count[1],countbar[1],count[0],rst,countbar[1]);
 dff dff3(count[2],countbar[2],count[1],rst,countbar[2]);
  dff dff4(count[3],countbar[3],count[2],rst,countbar[3]);

endmodule
```



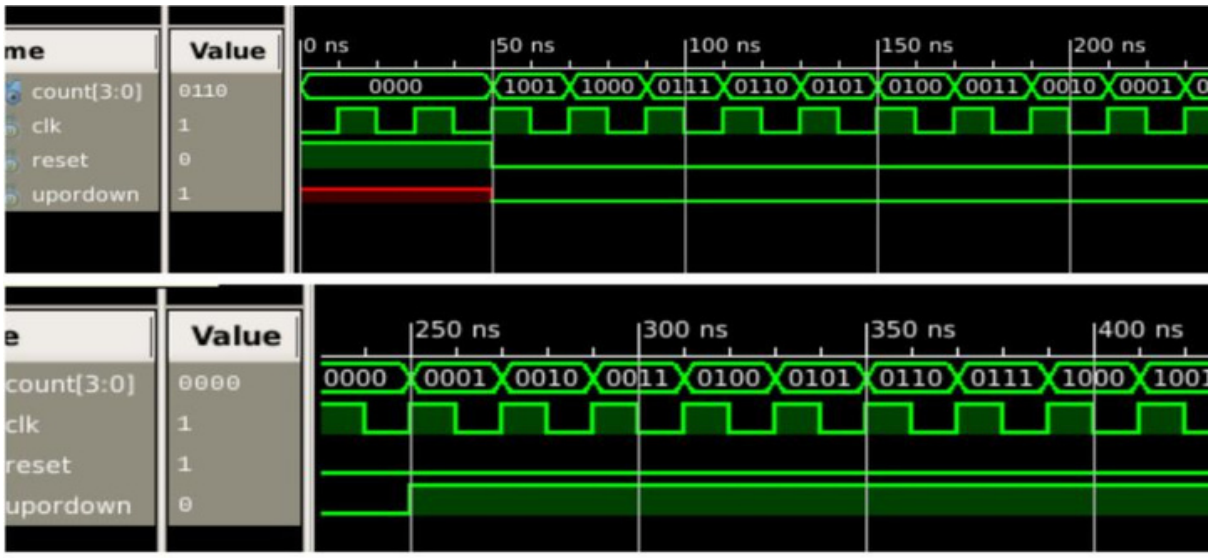
Day 35 : Mod-N UpDown Counter

A bidirectional counter is a synchronous up/down binary counter that has the ability to count in both directions either to or from some preset value as well as zero.
As well as counting “up” from zero and increasing or incrementing to some preset value, it is sometimes necessary to count “down” from a predetermined value to zero allowing us to produce an output that activates when the zero count or some other pre-set value

is reached.The upordown signal is considered and if it is 0, then it will act as down counter and if upordown is 1 it will act as up counter.

RTL Code:

```
module modN_counter
  # (parameter N = 10,
     parameter WIDTH = 4)

  ( input   clk,
    input   reset,
    input upordown,
    output  reg[WIDTH-1:0] count);

  always @ (posedge clk )
  begin
    if (reset==1)
     count <= 0;

    else
    if(upordown==1)       //Up Mode is selected
            if (count == N-1)
       count <= 0;
            else
        count<=count+1;  //increment counter

    else                 //Down Mode is selected
      if(count==0)
    count<=N-1;
       else
       count<=count-1;     //Decrement the counter
  end
endmodule
```



Day 36: Universal Binary Counter

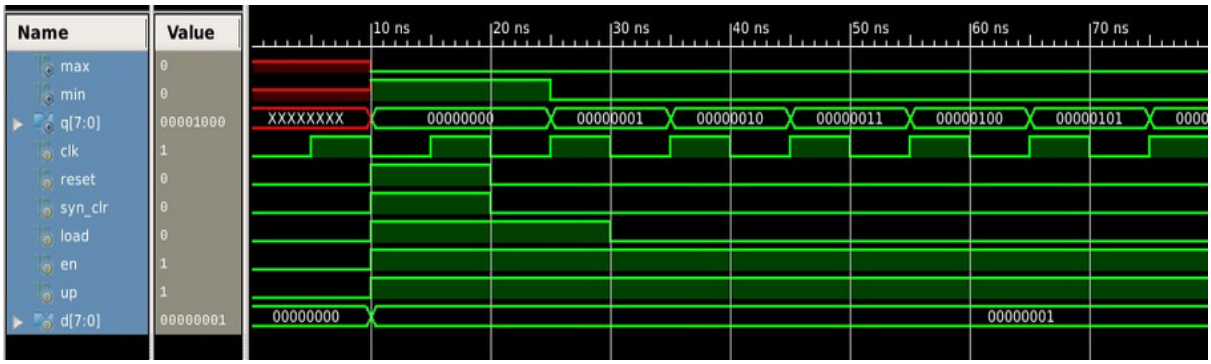
A universal binary counter is more versatile. It can count up or down, pause, be loaded

with a specific value, or be synchronously cleared. Its functions are summarized in below Table

Sync Clear	Load	Enable	Up	Q(Output)	Operation
1	-	-	-	00..00	Synchronous Clear
0	1	-	-	Data	Parallel load
0	0	1	1	Q+1	Count Up
0	0	1	0	Q-1	Count Down
0	0	0	-	Q	Pause

RTL Code:

```
module univ_bin_counter
#(parameter N=8)
(
input wire clk, reset,
input wire syn_clr , load, en, up,
input wire [N-1:0] d,
output wire max, min,
output wire [N-1:0] q
);
//signal declaration
reg [N-1:0] r_reg, r_next;
// body
// register
always @(posedge clk, posedge reset)
if (reset)
r_reg <= 0; //
else
 r_reg <= r_next;
// next-state logic
always @(*)
if (syn_clr)
r_next = 0;
else if (load)
r_next = d;
else if (en & up)
r_next = r_reg + 1;
else if (en & ~up)
r_next = r_reg - 1;
else
r_next = r_reg;
 // output logic
assign q = r_reg;
assign max = (r_reg==2**N-1) ? 1'b1 : 1'b0;
assign min = (r_reg==0) ? 1'b1 : 1'b0;
endmodule
```



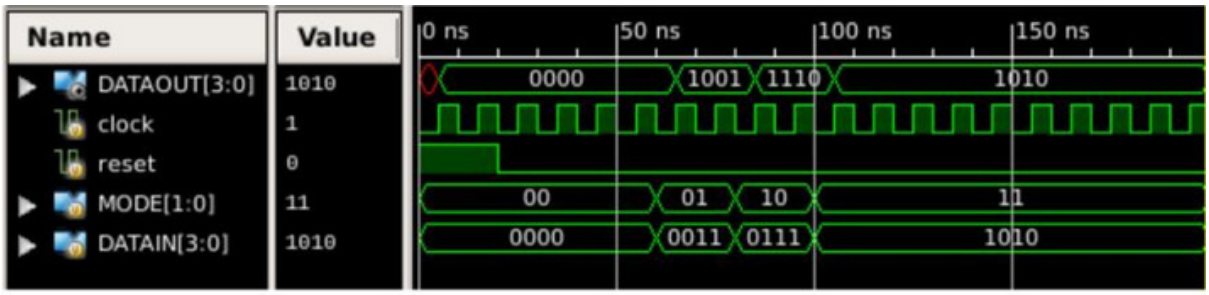
Day 37: Universal Shift Register

A Universal Shift Register is a register which can shift its data in both direction i.e. left and right directions. In other Words , a universal shift register is a bidirectional shift register .It is combination of design of bidirectional shift register and a unidirectional shift register with the parallel load provisions. It can perform parallel to serial operation (first loading parallel input and then shifting.It can also perform serial to parallel operation (first shifting and then retrieving parallel output .The desired operation is then specified by a 2 bit control signal as shown in the Below table.

S0	S1	Register Operation
0	0	No changes
0	1	Shift Right
1	0	Shift Left
1	1	Parallel Load

RTL Code:

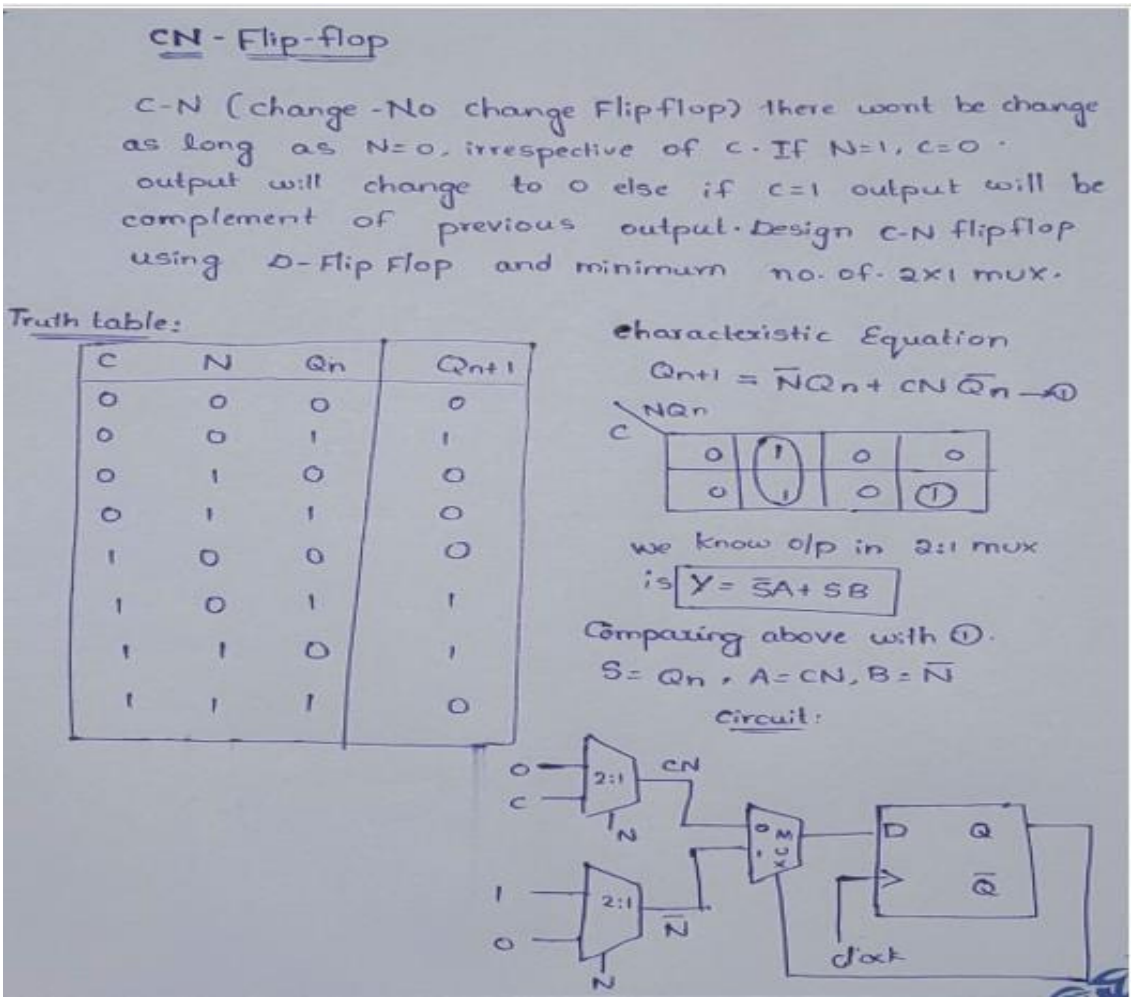
```
module universal_shiftreg(DATAOUT, clock, reset, MODE, DATAIN);
  output reg [3:0] DATAOUT;
  input clock, reset;
  input [1:0] MODE;
  input [3:0] DATAIN;

  always @(posedge clock)
  begin
    if(reset)
      DATAOUT <= 0;
    else
      begin
        case(MODE)
          2'b00 : DATAOUT <= DATAOUT;       // locked mode, do nothing
          2'b01 : DATAOUT <= {DATAIN[0], DATAIN[3:1]};//DATAOUT >> 1;
          2'b10 : DATAOUT <= {DATAIN[2:0], DATAIN[3]};//DATAOUT << 1;
          2'b11 : DATAOUT <= DATAIN;        // parallel in parallel out
        endcase
      end
  end

endmodule
```



Day 38: CN- Flipflop (Change -No Change Flip Flop) using DFF and 2:1 Mux

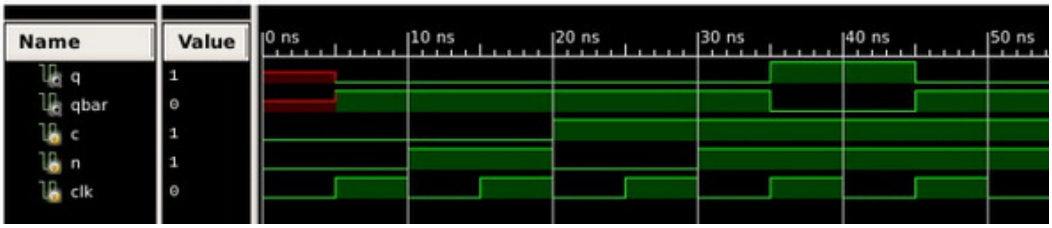
In C-N (Change – No change) flip-flop, there won't be any change in output as long as N is 0, irrespective of C. If N=1, then if C=0 output will change to 0 else if C=1 output will be the compliment of previous output. Design C-N flip-flop using D flip flop and minimum number of 2 x 1 multiplexer. The characteristic table and design of the above flip-flop is shown below.

C	N	Q(N+1)	State
0	0	Q(n)	No change
0	1	0	Reset
1	0	Q(n)	No Change
1	1	Q'(n)	Toggle



RTL Code:

```
module mux2X1(a,b,s,y);
input a,b,s;
output reg y;
always @(a or b or s)
begin
case(s)
0: y=a;
1:y=b;
default: y=1'b0;
endcase
end
endmodule

module d_ff(d,clk,reset,q);
input d,clk,reset;
output reg q;
always @(posedge clk)
begin
if(reset)
q=0;
else
q=d;
end
endmodule

module cn_flipflop(c,n,clk,q,qbar);
input c,n,clk;
output q,qbar;
wire cn,n_bar,d_wire;
mux2X1 mux1(1'b0,c,n,cn);
mux2X1 mux2(1'b1,1'b0,n,n_bar);
mux2X1 mux3(cn,n_bar,q,d_wire);
d_ff dff1( .d(d_wire),.clk(clk),.reset(),.q(q));
assign qbar=~q;
endmodule
```



Day 39: Frequency Divider by any Odd Number (Here I Used N=5)

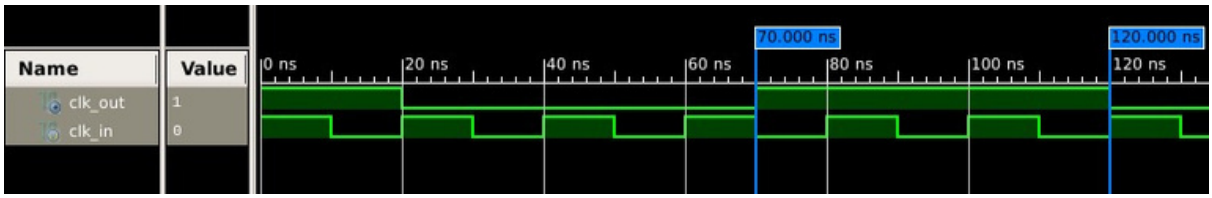
Frequency or clock dividers are among the most common circuits used in digital systems. Things get a little more complicated when we try to divide the frequency by an odd number, since we can't simply divide the number of input clock cycles by 2. If we observe the timing diagram describing a frequency divider by 5 in below figure , it give the output with 50% Duty Cycle.

RTL Code:

```
module clk_div_odd
#(parameter N=5)
(
 input    clk_in,
 output    clk_out
    );

reg  [3:0] count = 4'b0;         //4-bit counter
reg      A1 = 0;
reg        B1 = 0;
reg      Tff_A = 0;
reg     Tff_B = 0;
wire     clock_out;
wire     wTff_A;
wire     wTff_B;

//Connects registers to wires for combinational logic
assign    wTff_A  = Tff_A;
assign    wTff_B  = Tff_B;

assign    clk_out = wTff_B ^ wTff_A; //XOR gate

//Counter for division by N
always@(posedge clk_in)
 begin
  if(count == N-1) //Count to N-1.
   begin // Example: Use 4 to divide by 5
    count <= 4'b0000;
   end
  else
   begin
    count <= count + 1;
   end
 end
```

```
//Set A to high for one clock cycle when counter is 0
always@(posedge clk_in)
 begin
  if(count == 4'b0000)
   A1 <= 1;
  else
   A1 <= 0;
 end

//Sets B to high for one clock cycle when counter is (N+1)/2
always@(posedge clk_in)
 begin
  if(count == (N+1)/2) //Use (N+1)/2
   B1 <= 1; //Ex: (5+1)/2 = 3
  else
   B1 <= 0;
 end
```

```
//T flip flop toggles
always@(negedge A1) // Toggle signal Tff_A
 begin   //whenever A1 goes from 1 to 0
  Tff_A <= ~Tff_A;
 end

always@(negedge clk_in)
 begin
  if(B1) // Toggle signal Tff_B whenever
   begin //B1 is 1
    Tff_B <= ~Tff_B;
   end
 end

endmodule
```



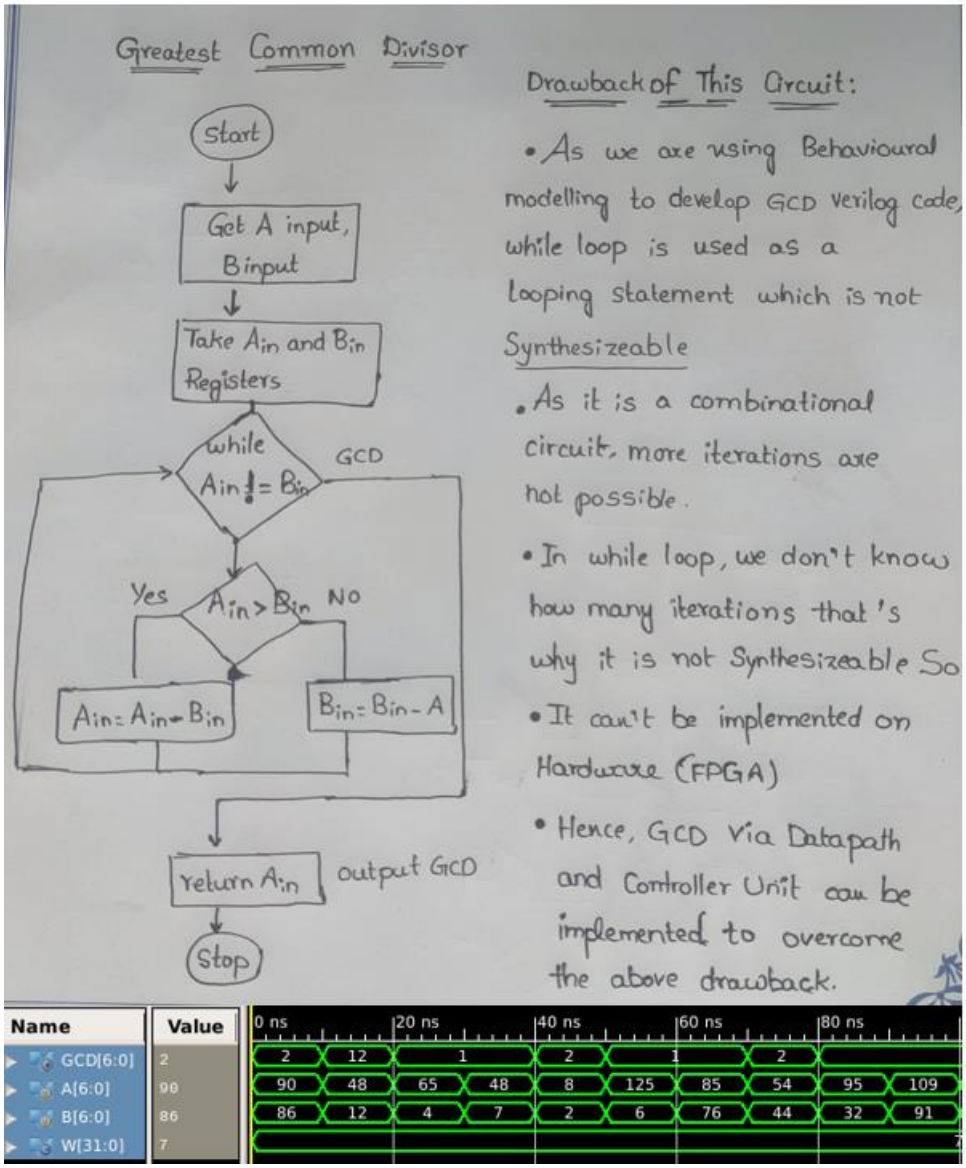
Day 40: Greatest Common Divisor Using Behavioural Modelling

In this GCD calculation , I have used simple algorithm using repeated subtraction. It is used to calculate the Greatest Common Divisor of any two numbers and the operation of the circuit can be explained by flowchart below. But it is not synthesizeable as we are using While loop.

Let's overcome the limitation of this Non-Synthesizeable circuit in the next code.

RTL Code:

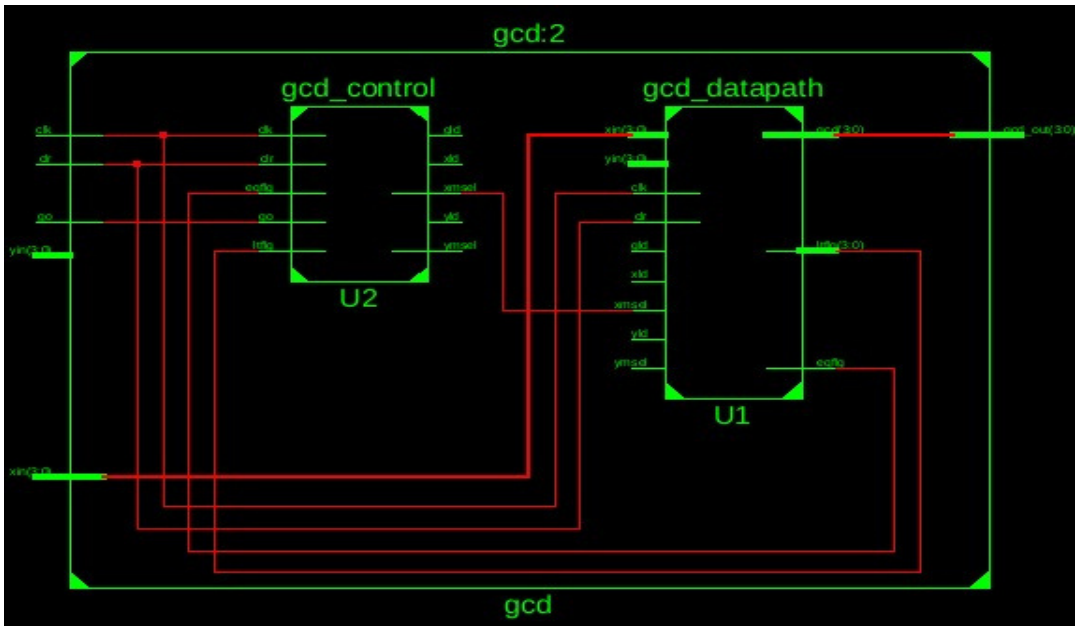
```
module gcd_beh#( parameter W = 7)
)(
input [W-1:0] A, B,
output  [W-1:0] GCD
);
reg [W-1:0] Ain, Bin, GCD;
always @(*)
begin
Ain = A; Bin = B;
while( Ain != Bin)
begin
if ( Ain < Bin )
Bin = Bin-Ain;
else
Ain = Ain - Bin;
end
GCD = Ain;
end
endmodule
```



Day 41: Greatest Common Divisor via Data Path and Controller

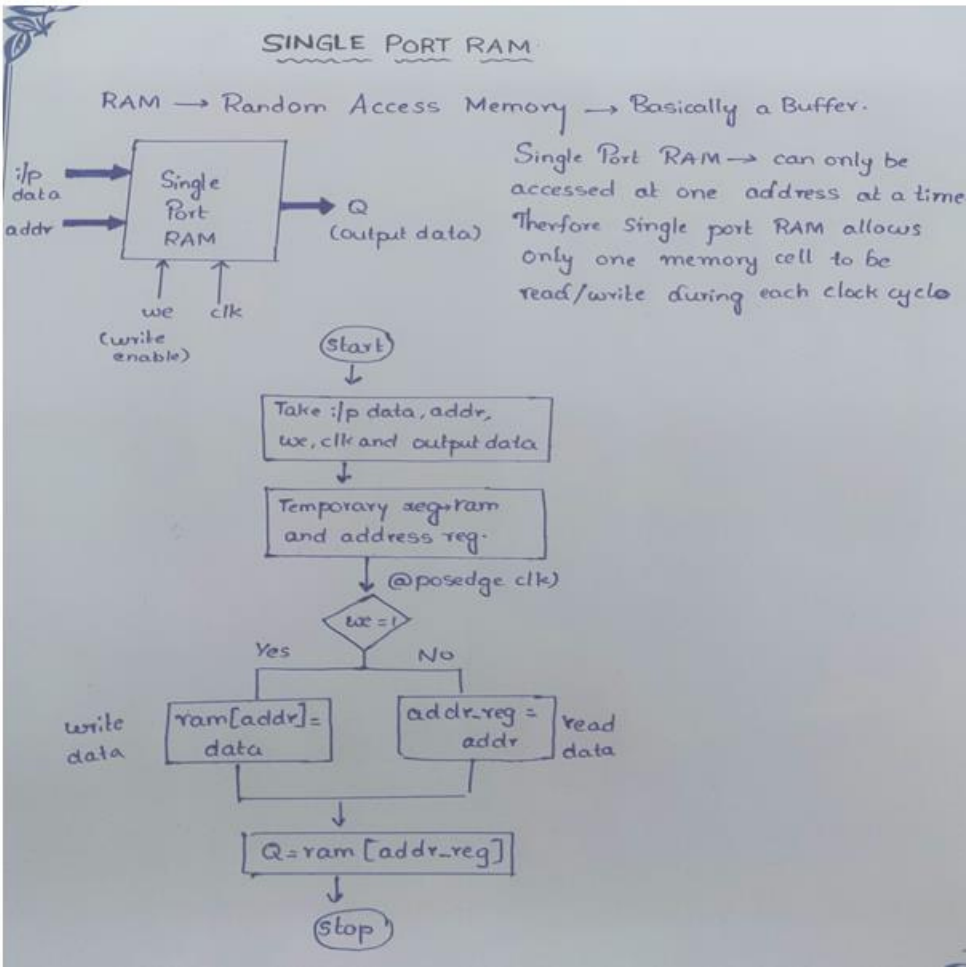
In previous code of Greatest Common Divisor using Behavioral Modelling , the main drawback is Non-Synthesizeable circuit , hence we implemented it using via Data Path and Controller circuit so that the circuit can be synthesized.

The design of the GCD calculator should be divided into 2 parts - a controller and a datapath. The controller is an FSM which issues commands to the datapath based on the current state and the external inputs. This can be a behavioral description. The datapath contains a netlist of functional units like multiplexers, registers, subtractors and a comparator. The Datapath does the actual GCD computation.



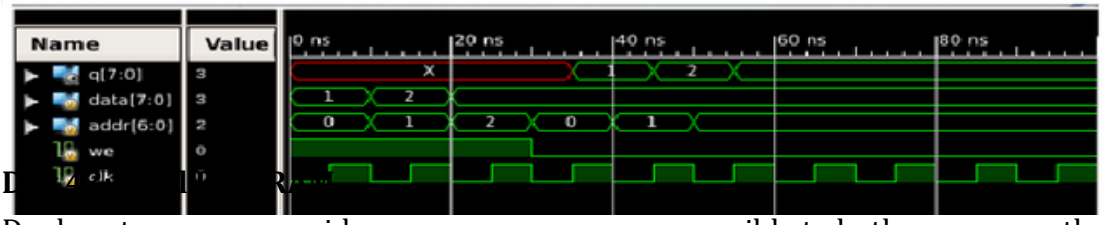
Day 42: Single Port RAM

The Single Port RAM block models RAM that supports sequential read and write operations. If you want to model RAM that supports simultaneous read and write operations, use the Dual Port RAM block.



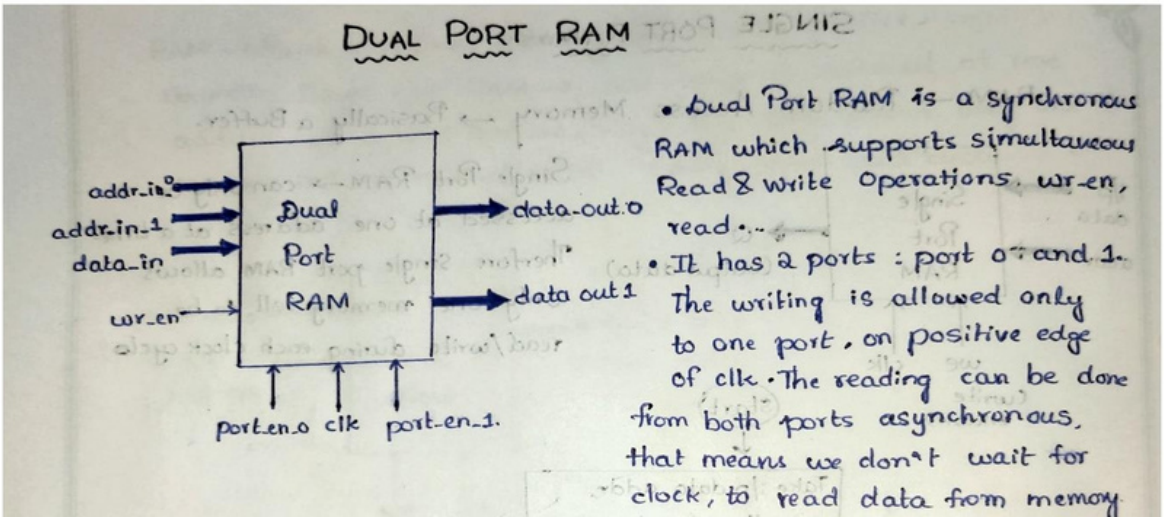
RTL Code:

```
module single_port_ram
#(parameter addr_width = 6,
parameter data_width = 8,
parameter depth = 64)
                (
                input [data_width-1:0] data,   //this is input data
                input [addr_width-1:0] addr,   //address
                input we,clk,                  //we is read and write c
                output [data_width-1:0] q      //q is outptut data
                );


//Declare RAM variable
reg [data_width-1:0] ram [depth-1:0];

//address register
reg [addr_width-1:0] addr_reg;

always @(posedge clk)
begin
if(we)                          //if we=1 write data into RAM
     ram[addr] <=data;
     else                       //if we=0 then read data out
         addr_reg <=addr;   //gets address value from outisde of module
end
assign q= ram[addr_reg];  //read data to q

endmodule
```



D **R**

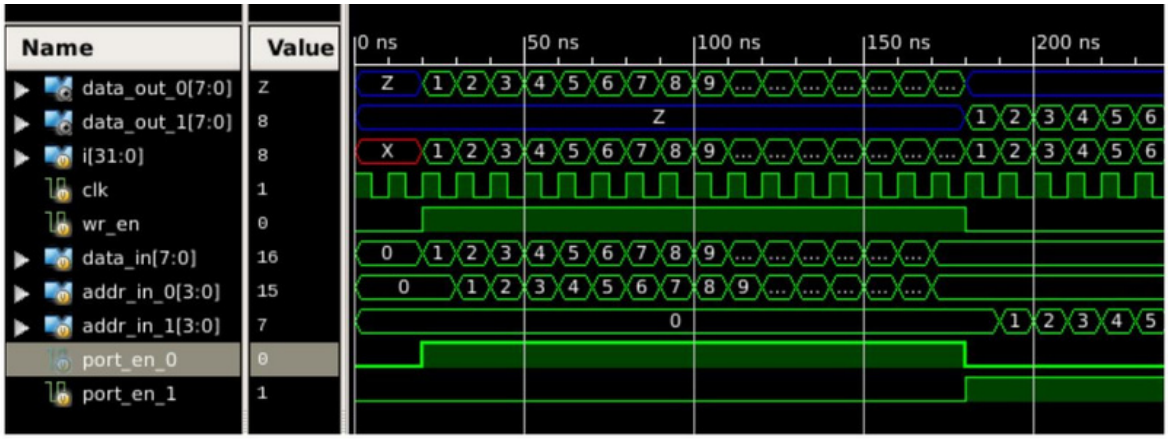
Dual port memory provides a common memory accessible to both processors that can be used to share and transmit data and system status between the two processors



On the positive edge of the clock, when the write operation takes place, if port enable 1 is high , the the data is written into port 1 and if port 0 enable is high, data written into port 0.But the read operation happens asynchronously without waiting for clock.

RTL Code:

```
module dual_port_ram
    # (parameter data_width=8,
       parameter addr_width=4,
         parameter depth=16
         )
    (   input clk, //clock
        input wr_en,    //write enable for port 0
        input [data_width-1:0] data_in,    //Input data to port 0.
        input [addr_width-1:0] addr_in_0,  //address for port 0
        input [addr_width-1:0] addr_in_1,  //address for port 1
        input port_en_0,    //enable port 0.
        input port_en_1,    //enable port 1.
        output [data_width-1:0] data_out_0,    //output data from port 0.
        output [data_width-1:0] data_out_1    //output data from port 1.
    );

//memory declaration.S
reg [data_width-1:0] ram[0:depth-1];

//writing to the RAM
always@(posedge clk)
begin
    if(port_en_0 == 1 && wr_en == 1)    //check enable signal and if write
        ram[addr_in_0] <= data_in;
end

//always reading from the ram, irrespective of clock.
assign data_out_0 = port_en_0 ? ram[addr_in_0] : 'dZ;
assign data_out_1 = port_en_1 ? ram[addr_in_1] : 'dZ;

endmodule
```

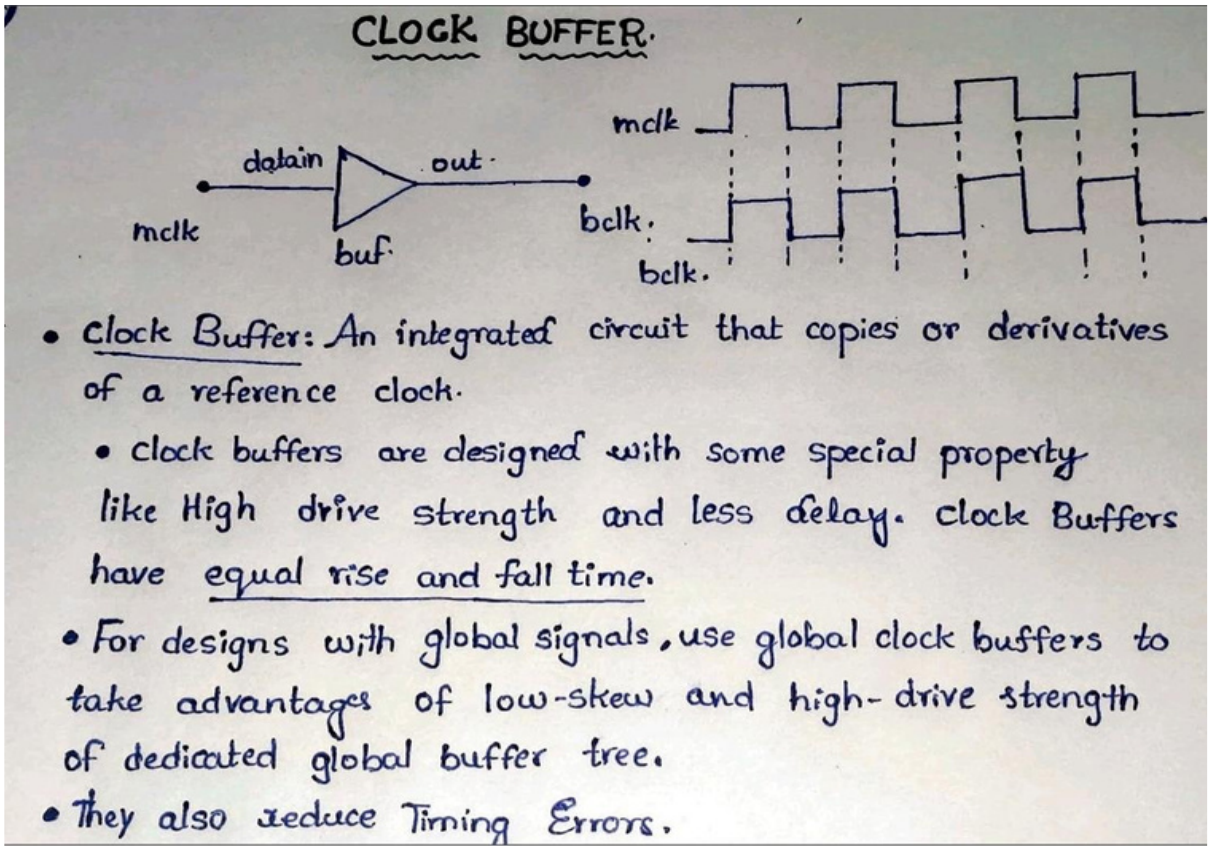


Day 44: Clock Buffer

A Buffer is an element which produces an output signal which is of the same value of the input signal.

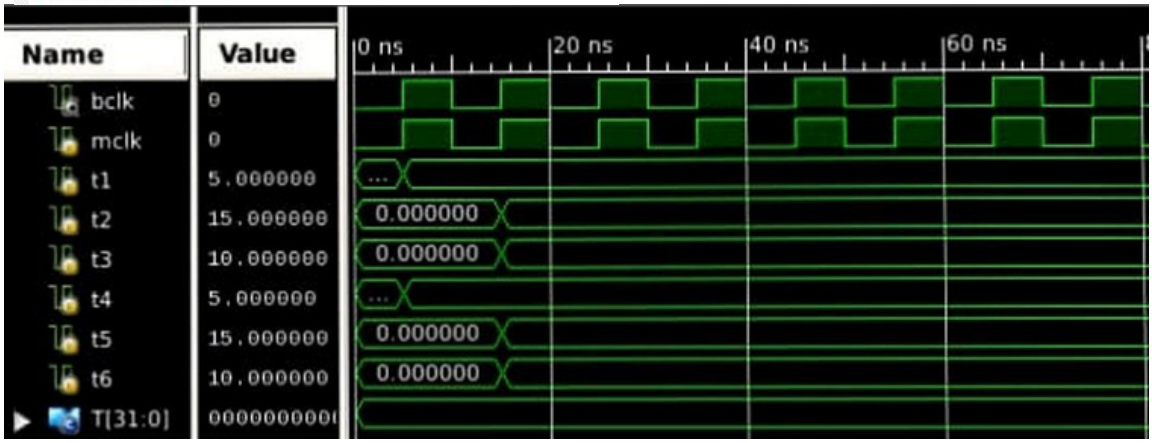
What is clock buffer vs normal buffer?

Clock buffers have equal rise and falltime. This prevents duty cycle of clock signal from changing when it passes through a chain of clock buffers. Normal buffers are designed with W/L ratio such that sum of rise time and fall time is minimum. They too are designed for higher drive strength



RTL Code

```
module clock_buffer(mclk,bclk);
input mclk;
output bclk;
buf b1(bclk,mclk);
endmodule
```



Day 45: Synchronous FIFO

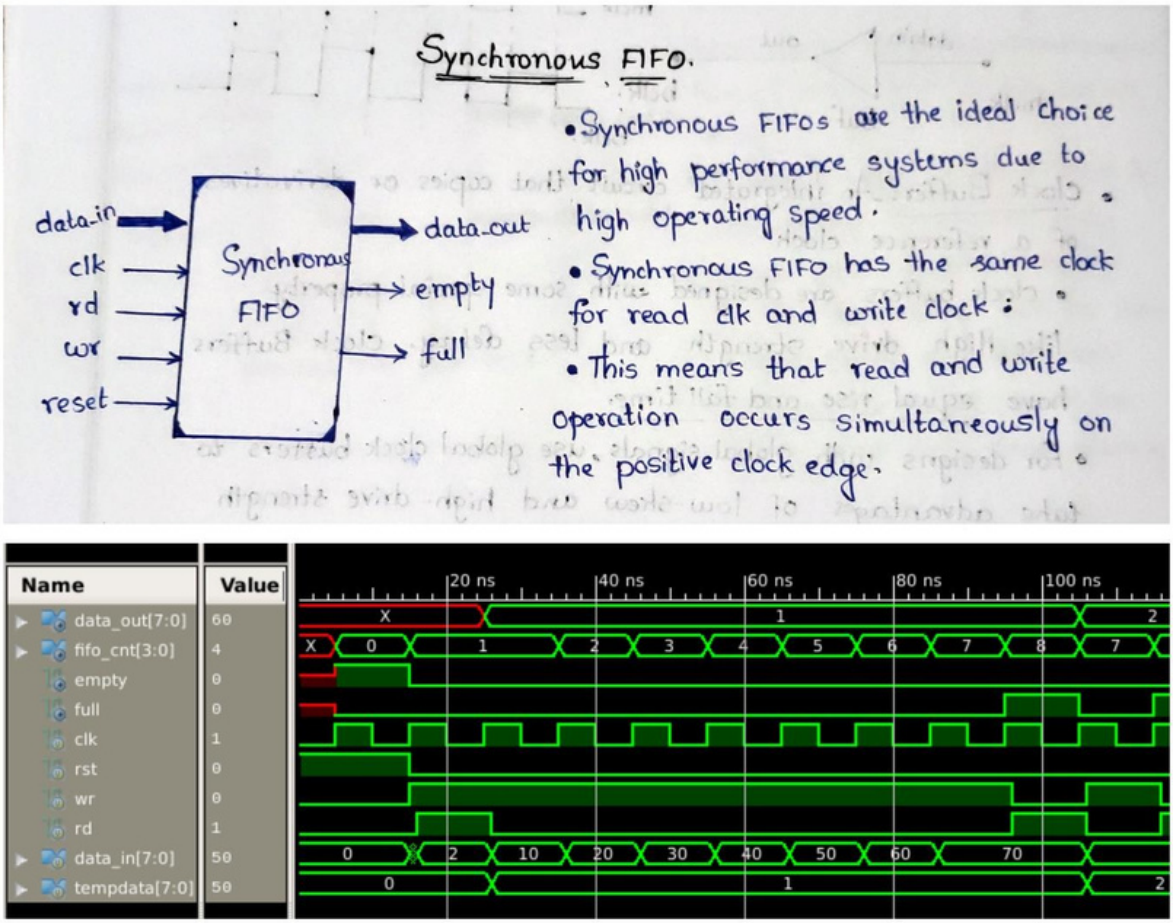
The name FIFO stands for first in first out and means that the data written into the buffer first comes out of it first.FIFOs are used in designs to safely pass multi-bit data words from one clock domain to another, or to control the flow of data between source and destination sides sitting in the same clock domain. If read and write clock domains are governed by the same clock signal, FIFO is said to be synchronous FIFO

Why is synchronous FIFO needed?

Synchronous FIFOs are the ideal choice for high-performance systems due to high operating speed. Synchronous FIFOs also offer many other advantages that improve system performance and reduce complexity. These include status flags: synchronous flags, half-full, programmable almost-empty and almost-full flags.

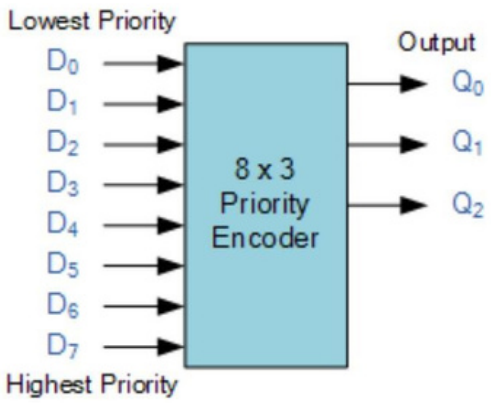
RTL Code:

```
module sync_fifo (input [7:0] data_in, input clk, rst, rd, wr,
output empty, full, output reg [3:0]fifo_cnt,
output reg [7:0] data_out);


reg [7:0] fifo_ram [0:7];
reg [2:0] rd_ptr, wr_ptr;
assign empty= (fifo_cnt==0);
assign full =(fifo_cnt==8);
always @(posedge clk) begin: write
if (wr && ! full)
fifo_ram [wr_ptr] <= data_in;
else if (wr && rd)
fifo_ram [wr_ptr] <= data_in;
end


always @ (posedge clk) begin: read
if (rd && !empty)
data_out <= fifo_ram [rd_ptr];
else if (rd && wr)
data_out <= fifo_ram [rd_ptr];
end
//pointer block
always @(posedge clk) begin: pointer
if (rst) begin
wr_ptr <= 0;
rd_ptr <= 0;
end
else begin
wr_ptr <= ((wr && ! full) || (wr && rd)) ? wr_ptr+1 :
wr_ptr;
rd_ptr <= ((rd && !empty) || (wr && rd)) ? rd_ptr+1:
rd_ptr;
end
end
```



Day 46: Priority Encoder

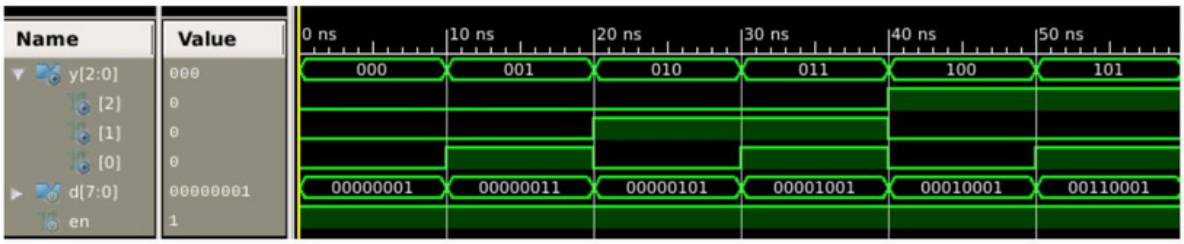
The priority encoder is a circuit that executes the priority function. The logic of the priority encoder is such that two or more inputs appear at an equal time, the input having the largest priority will take precedence. The truth table of a 8*3 priority encoder is given below.



Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

RTL Code:

```
module pri_en(d,y,en);
input [7:0] d;
input en;
output  [2:0] y;
reg [2:0] y;
always @(d or en )
begin
if(en)
begin
casex(d)

8'b00000001:y=3'b000;
8'b0000001x:y=3'b001;
8'b000001xx:y=3'b010;
8'b00001xxx:y=3'b011;
8'b0001xxxx:y=3'b100;
8'b001xxxxx:y=3'b101;
8'b01xxxxxx:y=3'b110;
8'b1xxxxxxx:y=3'b111;
endcase
end
else
begin
y=3'bxxx;
end
end

endmodule
```

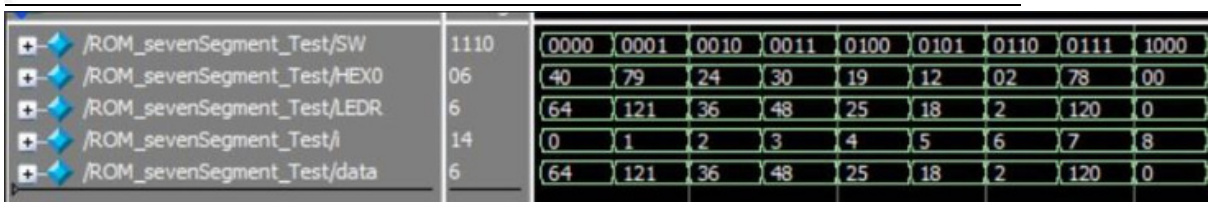


Day 47: Seven segment display pattern using ROM

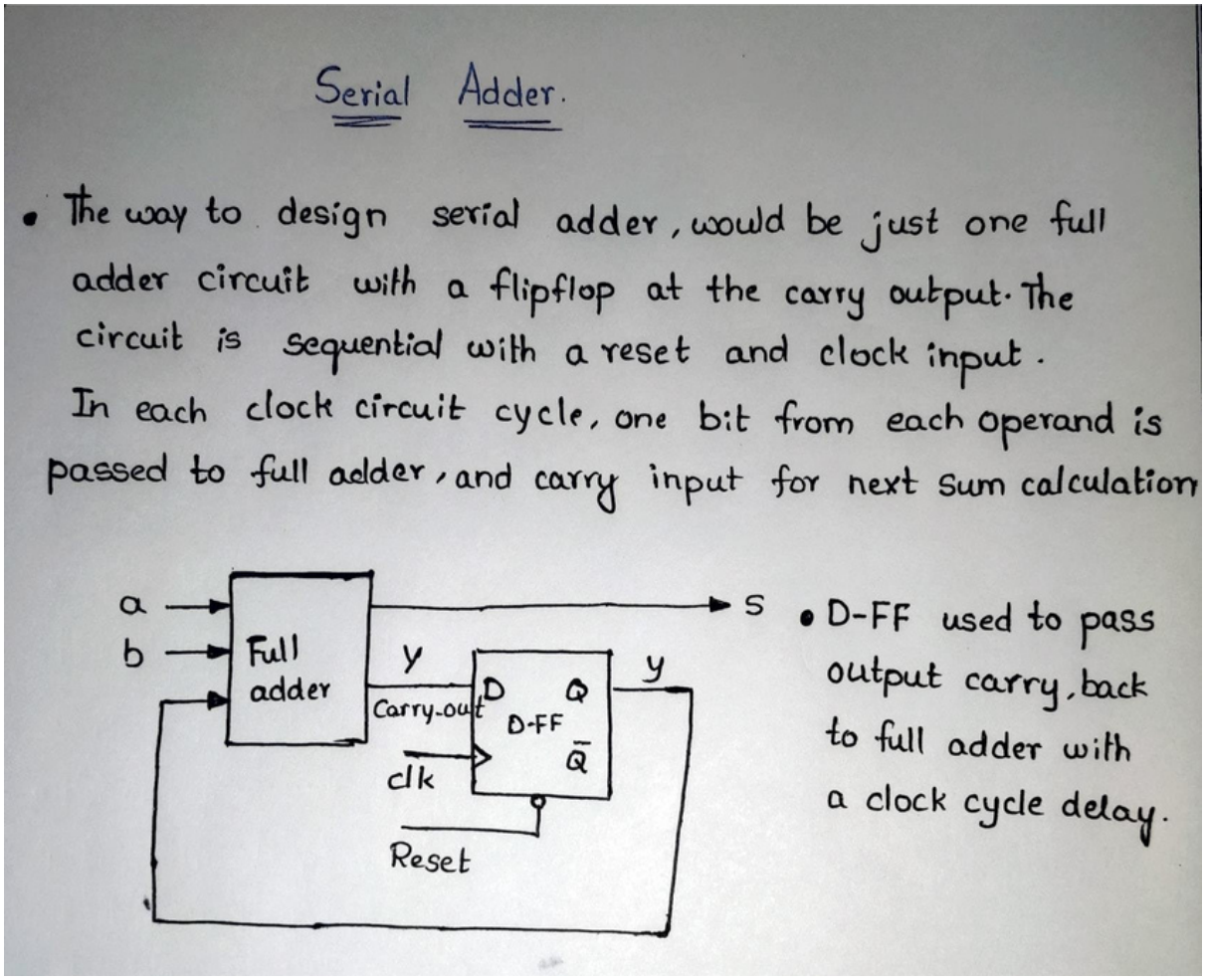
ROMs are the devices which are used to store information permanently.ROM is implemented on verilog to store the display-pattern for seven-segment device

RTL Code:

```
module ROM_sevenSegment
 #(
  parameter addr_width = 16, // store 16 elements
  addr_bits = 4, // required bits to store 16 elements
  data_width = 7 // each element has 7-bits
 )
 (
input wire [addr_bits-1:0] addr,
output reg [data_width-1:0] data // reg (not wire)
);
always @*
 begin
 case(addr)
  4'b0000 : data = 7'b1000000; // 0
  4'b0001 : data = 7'b1111001; // 1
4'b0010 : data = 7'b0100100; // 2
  4'b0011 : data = 7'b0110000; // 3
4'b0100 : data = 7'b0011001; // 4
  4'b0101 : data = 7'b0010010; // 5
  4'b0110 : data = 7'b0000010; // 6
  4'b0111 : data = 7'b1111000; // 7
4'b1000 : data = 7'b0000000; // 8
  4'b1001 : data = 7'b0010000; // 9
  4'b1010 : data = 7'b0001000; // a
    4'b1011 : data = 7'b0000011; // b
    4'b1100 : data = 7'b1000110; // c
  4'b1101 : data = 7'b0100001; // d
  4'b1110 : data = 7'b0000110; // e
  default : data = 7'b0001110; // f
endcase
 end

 endmodule
```

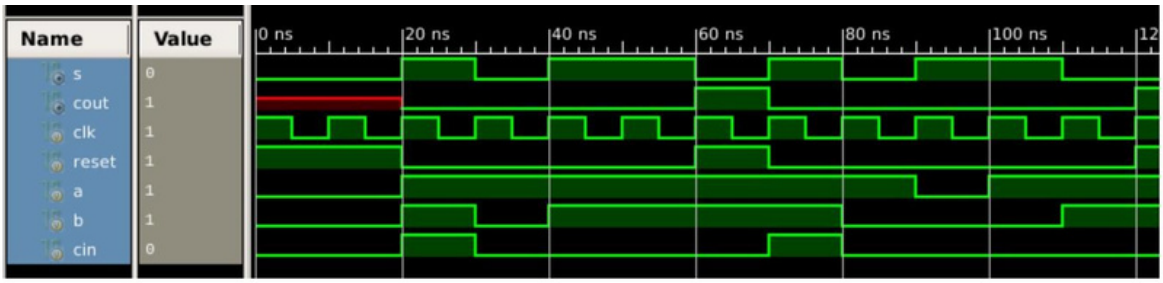


Day 48: Serial Adder



RTL Code:

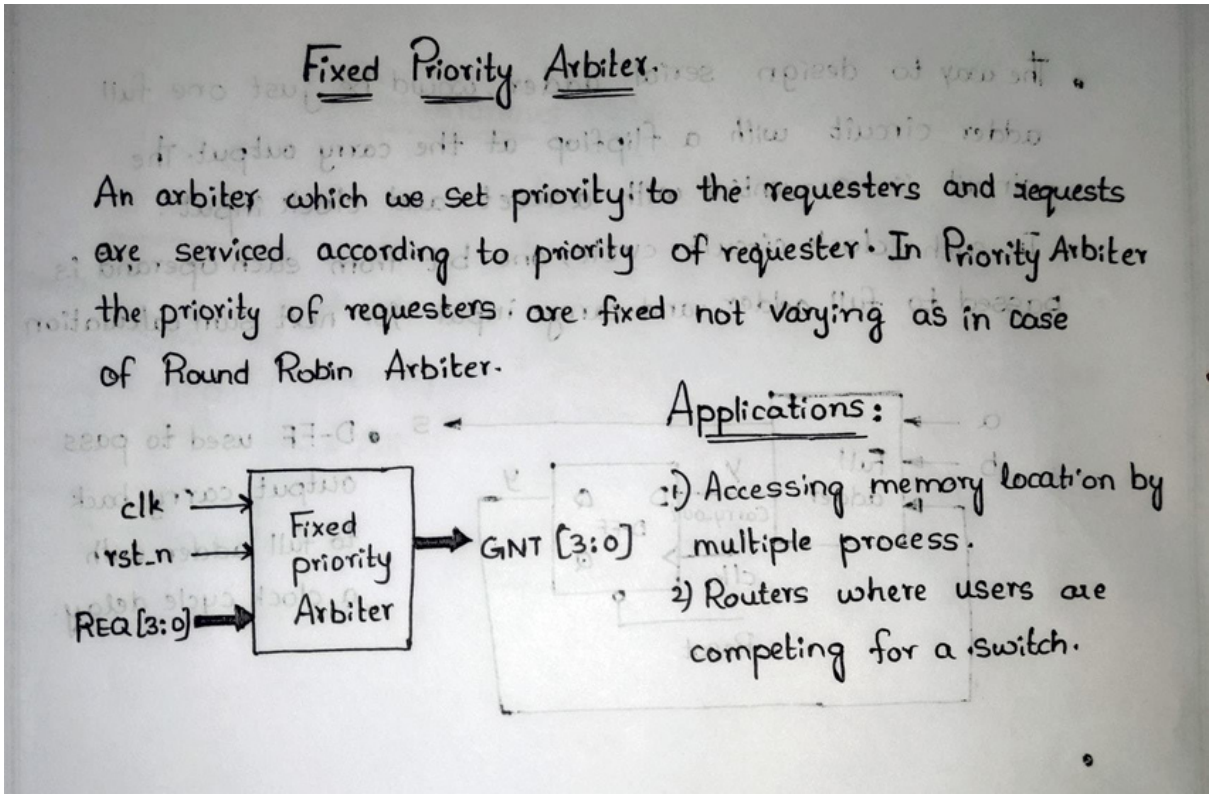
```
module serial_adder
    (   input clk,reset,
        input a,b,cin,
        output reg s,cout
        );

reg c,flag;

always@(posedge clk or posedge reset)
begin
    if(reset == 1) begin //active high reset
        s = 0;
        cout = c;
        flag = 0;
    end else begin
        if(flag == 0) begin
            c = cin;
            flag = 1;
        end
        cout = 0;
        s = a ^ b ^ c;  //SUM
        c = (a & b) | (c & b) | (a & c);  //CARRY
    end
end

endmodule
```



Day 49: Fixed Priority Arbiter

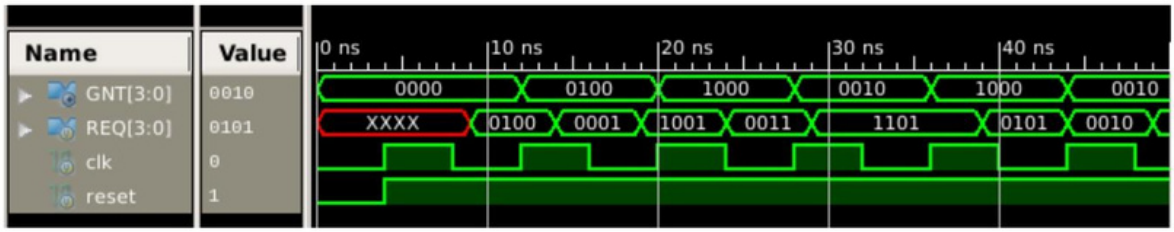
Arbitration means deciding who gets the bus when two or more devices request it. Priority is a means of specifying the relative importance of messages to be sent on the MCB. In this MCB implementation, the priority will be based on the priority of the calling task, and on whether this is a control or monitor request.

In Priority arbiter the priority of requesters are fixed not varying as in case of Round-Robin arbiter



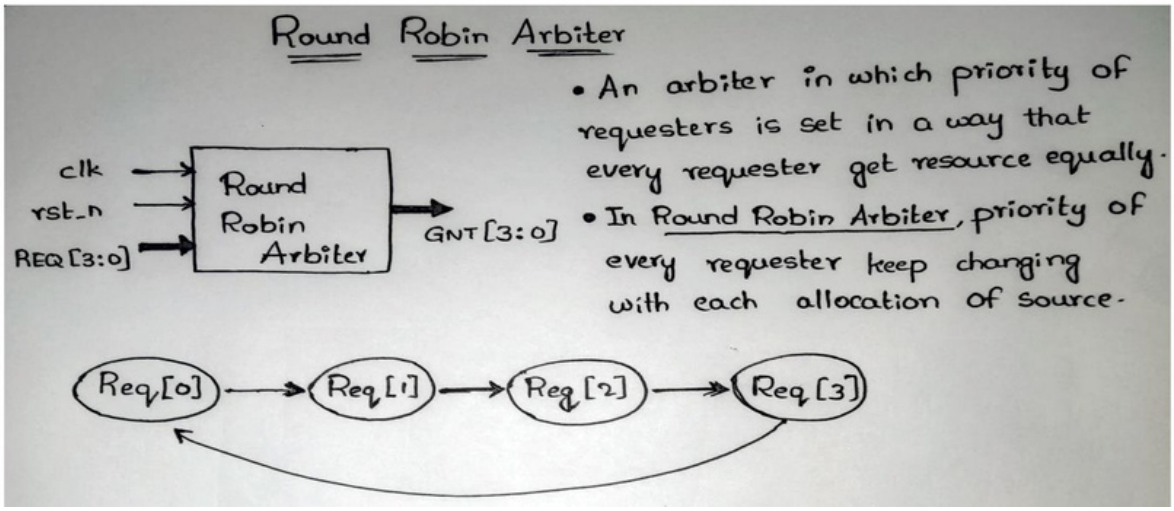
RTL Code:

```
module fixedpriority_arbiter(output reg [3:0] GNT,
            input [3:0] REQ,
            input clk,reset
);
always @(posedge clk or negedge reset)
// PRIORITY 3>1>0>2
    begin
    if(!reset)
    GNT<= 4'b0000;
    else if(REQ[3])
     GNT<= 4'b1000;
    else if(REQ[1])
     GNT<= 4'b0010;
    else if(REQ[0])
     GNT<= 4'b0001;
    else if(REQ[2])
     GNT<= 4'b0100;
     else
     GNT<= 4'b0000;
    end

endmodule
```



Day 50: Round Robin Arbiter

The Round-robin arbiter mechanism is useful when no starvation of grants is allowed. The arbiter quantizes time shares each requestor is allowed to have. A minimal fairness



is guaranteed by granting requestors in Round-robin manner. The requestors can prioritize their time shares by the weight.

RTL Code:

```
module roundrobin_arbiter(
input clk,rst_n,
input [3:0] REQ,
output reg [3:0] GNT
    );
    reg[2:0] pr_state;
    reg[2:0] nxt_state;

    parameter [2:0] Sideal = 3'b000;
    parameter [2:0]      S0 = 3'b001;
    parameter [2:0]      S1 = 3'b010;
    parameter [2:0]      S2 = 3'b011;
    parameter [2:0]      S3 = 3'b100;

    always @(posedge clk or negedge rst_n)

    begin
    if(!rst_n)
     pr_state <= Sideal;
     else
      pr_state <=nxt_state;
     end

    always@(*)
    begin
          case(pr_state)
            Sideal:
                        begin
                            if(REQ[0])
                                nxt_state = S0;
                            else if (REQ[1])
                                 nxt_state = S1;
                            else if (REQ[2])
                               nxt_state = S2;
                            else if (REQ[3])
                               nxt_state = S3;
                             else
                               nxt_state =Sideal;
                        end
```