Date 13/02/025 M T W T F S

DSD

Mid Term week # 01

Lecture # 01

- Verilog is HDL -
- use to describe hardware in verilog

1) Analyze the circuit
2) identify the input and outputs -
3) Label input and output -
4) Truth table
5) Simplify equation using K-map.
6) Implementation - (logic circuit)

easy-way
Model the behavior -

Half adder in verilog

```
module HA (S, Cy, A, B);   // name of the module; because continuous driven
output S, Cy;              // order is not important.
input A, B;                // wire
assign {Cy, S} = A+B;      // wire, reg
                           // {Cy, S} : Cy = high bit, S = lower bit
endmodule                  // model the Combinational circuit almost
```

we define hardware between the module and end module.

Common data types wire and reg

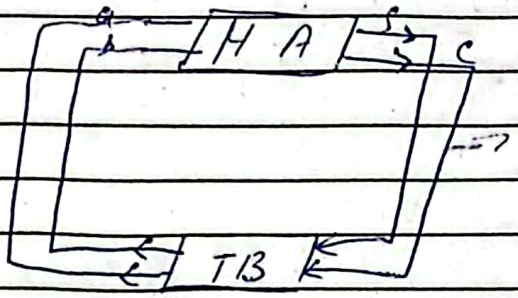
implementation gate-level

synthesized by translation of the description to hardware

Date _____ M T W T F S

assign use to concatinate-
→ translated into actual hardware
{ }, +, ^ synthesizable operator-

Now we dut (design under test)



reg
↓
not physical device
it is datatype.

```
module TST_HA (S, Cy, A, B))
input S, Cy;
output A, B,
reg A, B;  → kisi ko value assign karni ho to initial block mein to reg type use karna hoga-
initial begin
#10 A=0; B=0;
#10 A=0; B=1;
#10 A=1; B=0;
#10 A=1; B=1;
end
```

delay

→ vector generation part-

```
initial
$monitor ("%d, A=%b, B=%b, S=%b, Cy=%b" $time, A, B, S, Cy);
endmodule
```

print

initial block is not synthisizable

→ when time change it will print-
simulation time

$display → display only one
$monitor → when time change it will print-

Date _____ M T W T F S

Connection: Testbench and design

module top-level; // to connect two modules

wire A, B, S, Cy;
HA ha(S, Cy, A, B);
Test_HA t_ha(S, Cy, A, B);
endmodule

data-flow model

xor (o, i1, i2) ← primitive

we can omit instance but in user-defin it will must-

→ Full-adder
→ ripple-carry adder.

Lecture # 2

Data level flow

it is a type of behavior modeling - behavior mein different level of abstraction hai -

module CS, Cy, A, B);
input A, B;
output S, Cy;

assign {Cy, S} = A+B; ← abstract behavior

assign S = A ^ B;
assign Cy = A & B; → data flow

xor x1(S, A, B);
and a1(Cy, A, B); → gate level
endmodule

Date _____ M T W T F S

→ Design men synthesizable cheezen use hon gee.

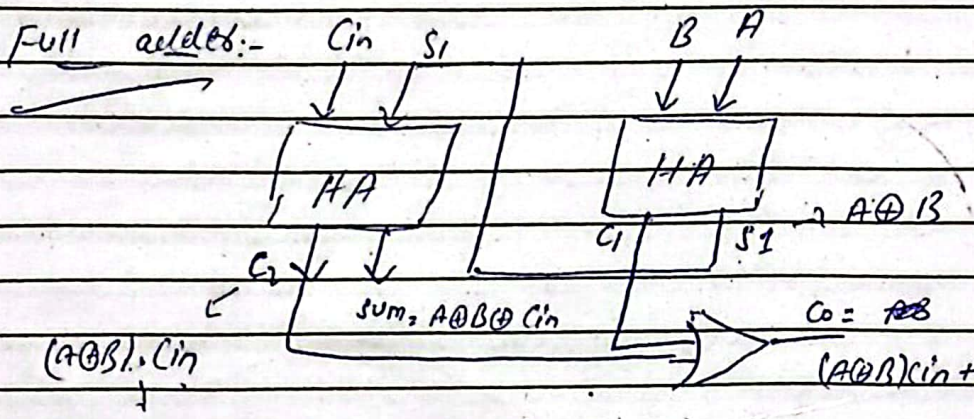
if inputs are fewer the exhaustive verification is possible.

① Design
② specification
③ Synthesis
↳ implementation
④ verification
⑤ partitioning.

reg x → mean unknown
wire z → mean high impedence

In wave x is shown by red line.

Full adder:-



→ Internal output declare using wire
→ output are special wire using type output.

```
module FA(Co, sum, a, b, ci);

  output Co, sum;
  input a, b, ci;

  wire s1, c1, c2;

  HA ha1(s1, c1, a, b);
  HA ha2(sum, c2, s1, ci);
```

positional association

→ structural level
↓
Blocks and interconnection.

Date MTWTFS

or o1(c0, c1, c2);

endmodule

module tst_FA (co, sum, a, b, ci);
input co, sum;
output a, b, ci;

reg a, b, ci;

initial begin
#5 a=0; b=0; ci=1
#5 a=1; b=1; ci=1
#5 a=1; b=0; ci=0
end
initial
$monitor ("%d, a=%b, b=%b; ci=%b, ~~time~~
"sum=%b"; "cout=%b", $time, a, b, ci, sum, co)
endmodule.

top
module top-level;
wire co, sum, a, b, ci;
tst_FA tst (co, sum, a, b, ci);
FA fa (co, sum, a, b, ci);
endmodule

In ~~full~~ this top level is full adder and the lower level is half adder.

HA ha1 (.S(sum), .Cy(C2), .A(C1), .B(Cin))

Date _____ M T W T F S

Week # 03

Lecture 03

RCA3 — orders not matter

module F.A (cout, sum, A, B, cin); explicit association

HA ha0 (.A(A), .B(B), .S(C1), .Cy(C0));

HA ha1 (sum, C2, C0, cin);

OR o1 (cout, C1, C2);

module RCA3 (Sum, Cout, A, B, Cin);

//vector → after synthesis it will translated to bus

input [2:0] A, B;

input cin;

output [2:0] sum;

output Cout;

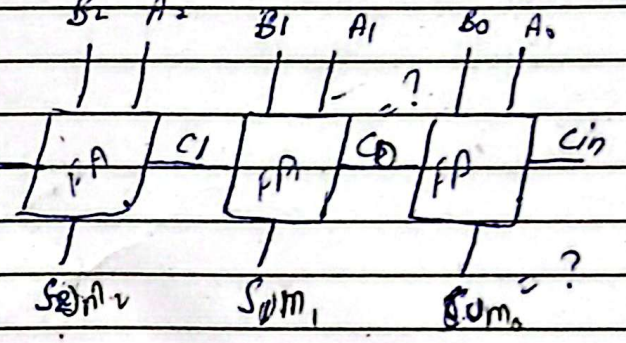
wire C0, C1;

FA fa0(sum[0], C0, A[0], B[0], 1'b0)

↘ collection of wire

↘ ground

↙ size binary



FA fa1 (sum[1], C1, A[1], B[1], C0);

FA fa2 (sum[2], Cout, A[2], B[2], C1);

endmodule

↙ structural level modeling

Page No. | Date | MTWTFS

If Cin not here it will not be re-usable.

→ In verilog we have two block initial and always.

initial → only one time → not (synthesizable) → translate it actual hardware

always → always change when event occur. → always running

we can write

reg cin; → not physical register

initial cin = 1'b0;

it's just type

```
module RCA3_tb(sum, Cout, A, B, Cin);
input [2:0] sum; input cout;
output [2:0] A, B; output cin;

reg [2:0] A, B; reg cin;
initial begin
  A = 3'b101; B = 3'b001; Cin = 1'b0;
#7 A = 3'b101; B = 3'b110; Cin = 1'b0;
#10 A = 3'b110; B = 3'b001; Cin = 1'b0;

initial
$monitor("%d, A = %b, B = %b, sum = %b, Cout = %b",
          $time, A, B, sum, Cout);
endmodule
```

2^7 = 128 possible exhausted verification

[2:0 , [3:0)

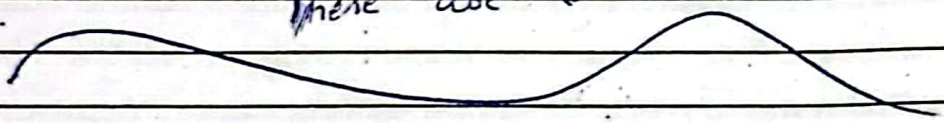
Date MTWTFS

```
module Top.
    wire [2:0] A, B, sum;
    wire Cin, Cout;

    RCA3 dut (sum, Cout, A, B, Cin);
    RCA3_tb tb(sum, Cout, A, B, Cin);
endmodule
```

6-bit RCA

```
module RCA6 (sum, Ct, A, B);   // Ct → final carry

    output [5:0] sum; output Ct;
    input [5:0] A, B;
    wire [illegible] C;

    RCA3 inst0 (sum[2:0], C, A[2:0], B[2:0], 1'b0);
    RCA3 inst1 (sum[5:3], Ct, A[5:3], B[5:3], C);
endmodule.
```

↓ These are two blocks



→ gate level is also called structural level.

→ we describe algorithm

→ behavioral level. mean discription known.

⇒ scalers are wire

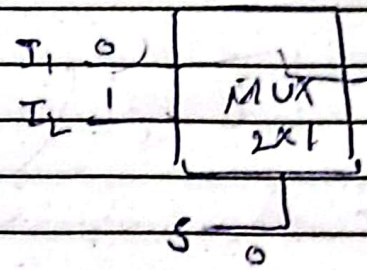
→ vectors are buses

next lecture
decoder and mux

Date _____ M T W T F S

Week # 03

Lecture # 02

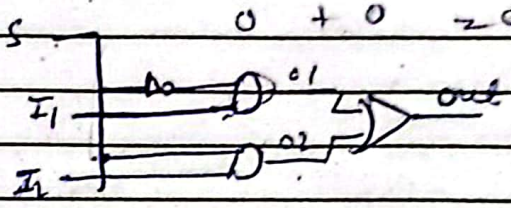


expression

$$O = \bar{S} I_1 + S I_2$$

$$1(0) + 0(1)$$

$$0 + 0 = 0$$



→ for behavior level only need to know the function of the MUX.

```
module m2to1 (out, , I1, I2, s);

    output out;
    input I1, I2, s;

    wire sbar, o1, o2;

① // assign out = !s?I1:I2;        → behavior level
② // assign out = ~S&I1 | S&I2;    → Data flow
③ or g1(out, o1, o2);              explicit structural
   and g2(o2, s, I2);                 code
   not n1(sbar, s);
   and g1(o1, sbar, I1);           → B
```

Date _____ M T W T F S

→ In verilog statements are executed in parallel orders not matter.

(4) method

always running or
B: reg out; ↑ → when I1 change
always @ (I1, I2, S) block will execute
event triggers

→ only input.

if (S)
    out = I2;
else
    out = I1;

→ alternate always @(*)

→ Synthesizable

endmodule.

★★★

→ if if statement is completly satisfied the circuit is combinational circuit.
if not the sequential circuit will be ~~sattified~~ - synthesize.

(3th) method → we don't write output here

always @(I1, I2, S)
    case (S)
        1'b0: out = I1;
        1'b1: out = I2;
    endcase

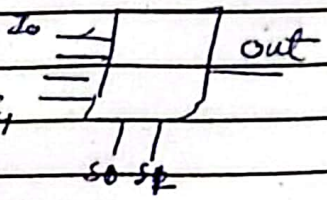
execute in sequence (combinational)
↑ blocking
=
<= → Non Blocking
↓
execute in parallel (sequential)

→ Both blocking and unblocking use in always block.
→ Blocking use in combinational circuit.
→ Non-Blocking use in sequential circuit.
→ we can't write assign in always block because it is concurrent.

Date MTWTFS

for 4x1

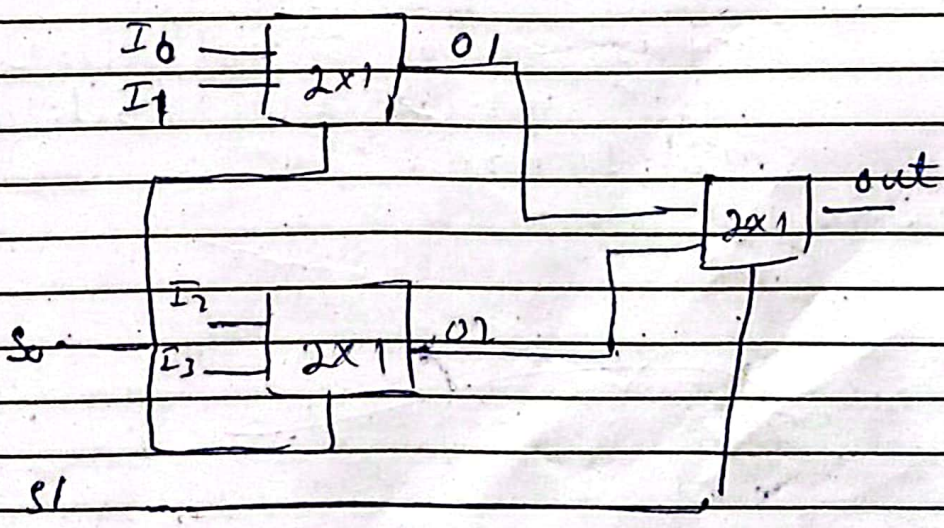
~~always @ (I1, I2, I3, I4, S)~~
~~case (S)~~
~~2'b00 : out = I1 ; I4~~




```
module m4to1 (out, I, s);
output out;
input [3:0] I;
input [1:0] s;
wire o1, o2;
m2to1 m0 (o1, I[0], I[1], s[0]);
reg out;
always @ (I, s)
    case (s)
        2'b00 : out = I[0];
        2'b01 : out = I[1];
        2'b10 : out = I[2];
        2'b11 : out = I[3];
    endcase
endmodule.
```

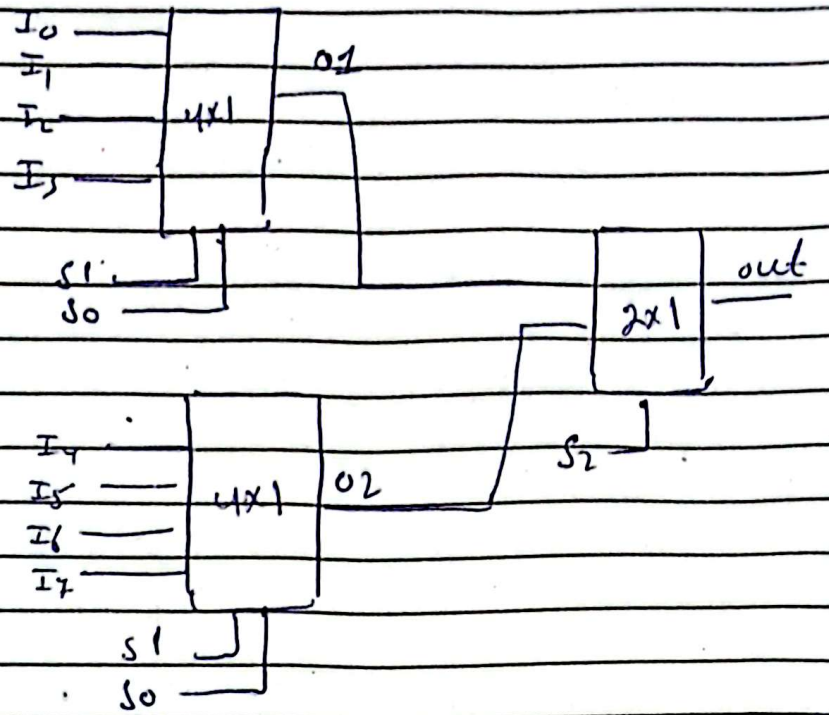
out = $\bar{S}_0\bar{S}_1 I_0 + \bar{S}_0 S_1 I_1 + S_0\bar{S}_1 I_2 + S_0 S_1 I_3$

```
m2to1 m1 (o2, I[2], I[3], s[0]);
m2to1 m2 (out, o1, o2, s[1]);
```



Date M T W T F S

8x1 MUX



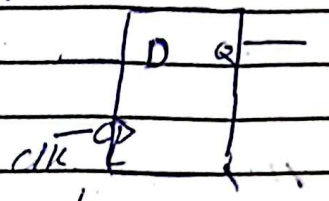

```
module m8to1 (out, I, S);
    output out;
    input [7:0] I;
    input [2:0] S;
    wire O1, O2;

    m4to1 m0 (O1, I[0:3], S[1:0]);
    m4to1 m1 (O2, I[4:7], S[1:0]);
    m2to1 m2 (out, O1, O2, S[2]);
endmodule
```

concatinate operator

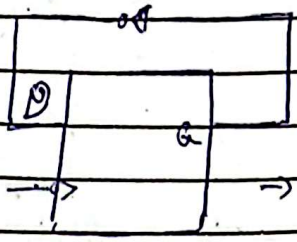
{I[0], I[1], I[2], I[3]}

Date _____ Week # 04 M T W T F S

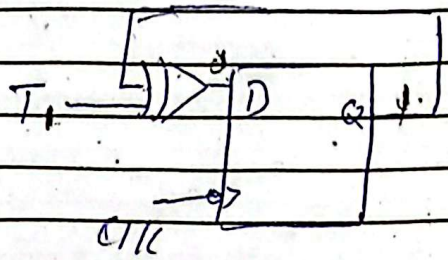
Lecture # 01:



↓
negative edge triggered



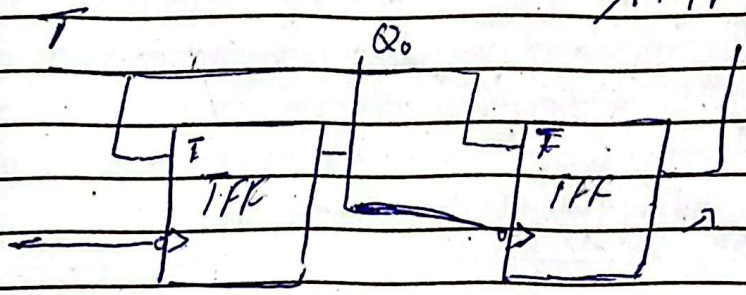
→ T using D-flip flop



T Flip-flop

→ negative edge pr toggle hoga
→ possitive edge pr previous state maintain hogee

→ Ripple counter



→ control counter because of T input

Ripple
Two bit counter 0 – 3

Date M T W T F S

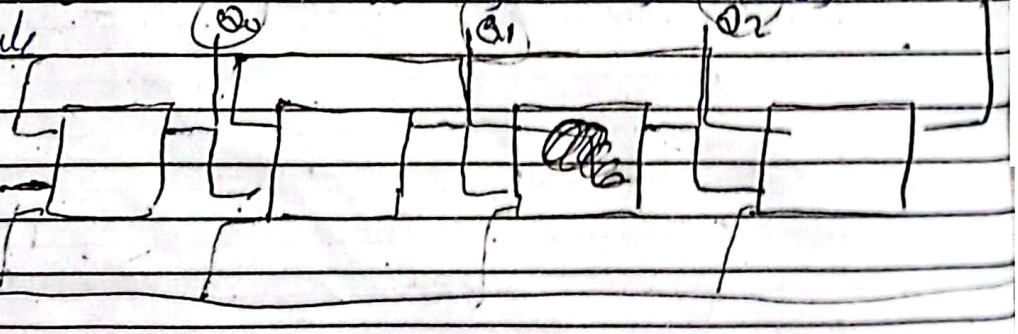
```
module D-FF (q, d, clock, reset);
output q;
input d, clock, reset;
reg q;

always @(negedge clock or posedge reset)
    if (reset)
        q = 1'b0;
    else
        q = d;

endmodule
```

(Asynchronous)

Isko hata den to ye synchronous bn jaye ga.

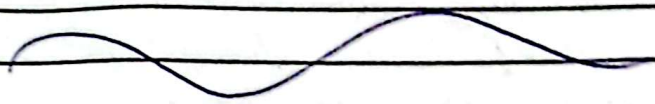
If negedge and posedge not written it will ~~not~~ become combinational or sequential depend on the description.

```
module T-FF (q, d, clock, reset);
output q;
input clock, reset;
wire d;

not n1 (d, q);

D-FF d0 (q, d, clock, reset);

endmodule
```



Date M T W T F S

```
module ripple_counter (q, clock, reset);

    output [3:0] q;
    input clock, reset;

    T_FF tff0 (q[0], clock reset);
    T_FF tff1 (q[1], q[0], reset);
    T_FF tff2 (q[2], q[1], reset);
    T_FF ff3 (q[3], q[2], reset);
endmodule
```

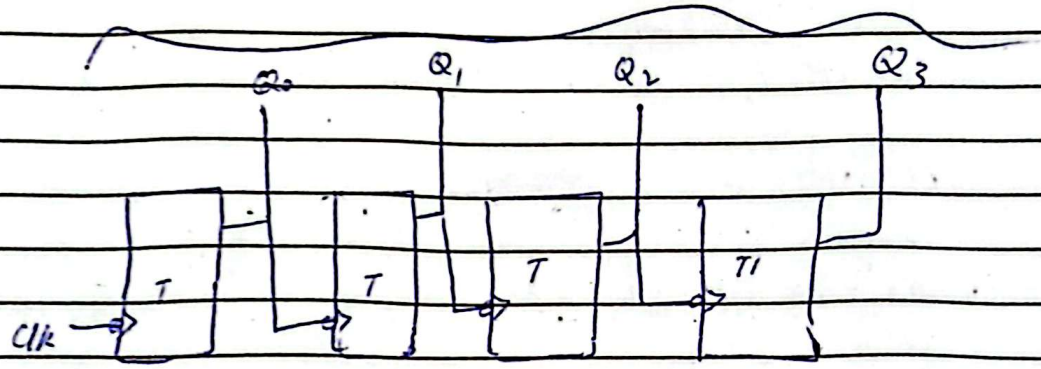


```
module T_FF (q, T, clock, reset);

    output [3:0] q;
    input T, clock, reset;

    wire d;

    xor x1 (d, T, q)
    .
    .
```

```
module stimulus (q, clock, reset);
    input [3:0] q;
    output clock, reset;
    reg clock, reset;

    always
        #5 clock = ~clock;
```

Date _____ M T W T F S

```
initial
begin
        clock = 1'b0;
        reset = 1'b1;
#15 reset = 1'b0;
#500 $finish;
end

// monitor the outputs

initial
        $monitor ("$time" T=%b, reset=%b, output
        =%d", T, reset, v);
endmodule
```

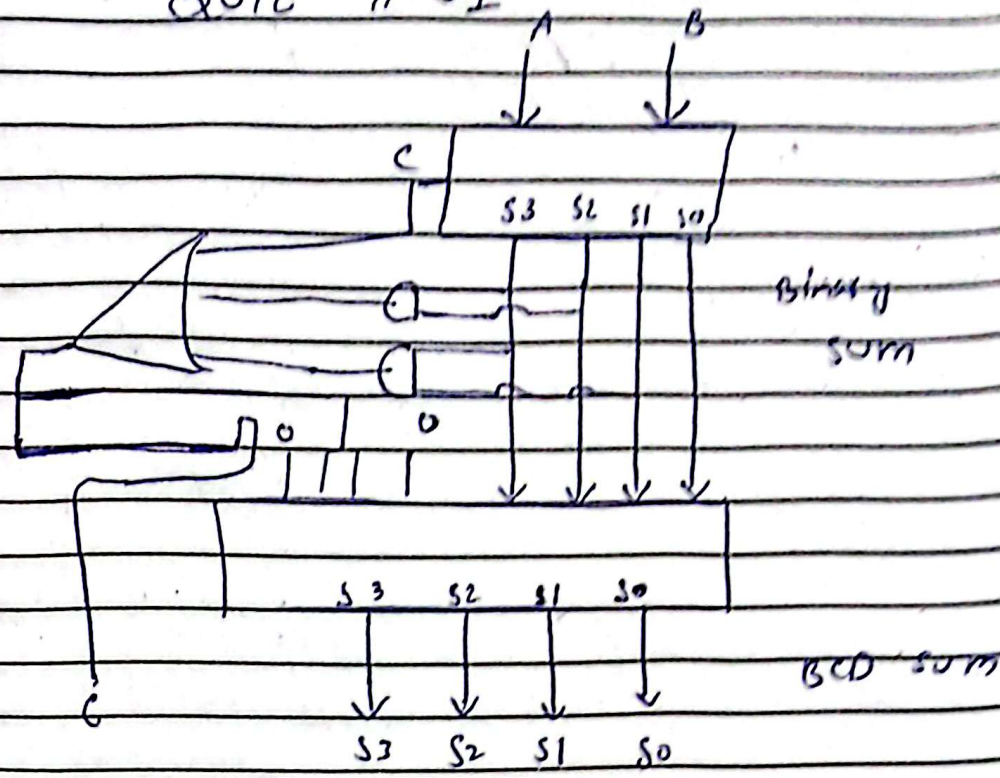


Q_3	Q_2	Q_1	Q_0
0	0	0	0
0	0	0	1
.	.	.	.

A	B	C
0	0	0
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1

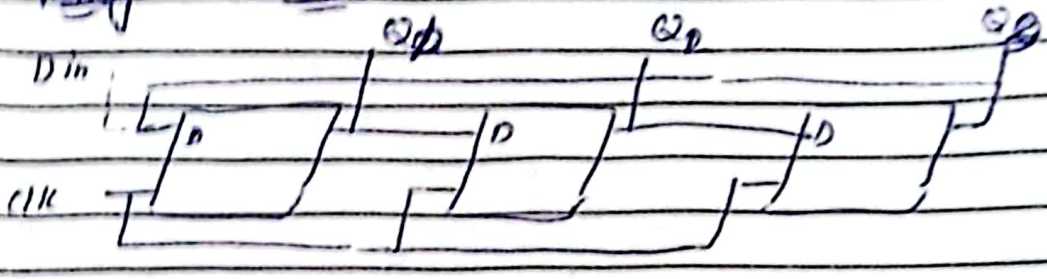
Date M T W T F S

Quiz # 01




```
module add4 (C, S, A, B);

    input [3:0] A, B;
    output [3:0] S;
    output C;
    assign [C, S] = A+B;
endmodule

module BCD (C, S, A, B);
    input [3:0] A, B;
    output [3:0] S;
    output C;

    wire Cy, Cz;
    wire [3:0] sum;
    add4 a1 (Cy, sum, A, B);
    assign C = Cy | (sum[3]&sum[2]) | (sum[3]&
                                       sum[1]);
    add4 a2(Cz, S, {1'b0, C, C, 1'b0}, sum);
endmodule
```

Ring Counter :



3-bit ring counter
serial in serial out
→ 3-bit right shift register

counter have flip flop and register made up from flip flop.

1
4
2
1

Circle 8bit shift left counter

module sc (clk, init, count);
↑ name ↑ serial input ↑ count output

input clk, init;
output [7:0] count;

reg [7:0] count;

always @(posedge init or negedge clk)
if (init)
count = 8'b10000000;
else
count = {count[6:0], count[7]};

→ high active and asynchronous
→ when posedge come it will execute
→ concatination

shift rotation

00000001
00000010
00000100

Date _____ M T W T F S

for shift right

count = 8'b00000001

count = {count[0], count[7:1]};

10000000
01000000
00100000
00010000
00001000
00000100

→ posedge and negedge use with sensitivity list signal.

always @ (posedge clk)
if (~init)
always @ (init)
if (init)
count = 8'b10000000;
else

↓

This is allowed

→ always block executed parallely
→ default value of wire is 0
→ and default value of reg is 1.
→ if is not completly specified no matter because circuit is sequential.

2nd method

M T W T F S

always @ (posedge init or posedge clk)

if (init)

count = 8'b10000000;

begin → blocking

count = count << 1;

count[0] = count[7];

end

if << → new reset

That is why one bit

#200 $finish → use to end simulation.

→ output