HDL : Hardware Description Lang.

__/__/____

VERILOG

~~Design Idea~~

~~Behavioral~~

Design idea

Behavioral design → Flow Graph / Pseudo Code

Data Path design → Bus / register structure

Logic Design → Gate / F.F Netlist

Physical design → Transistor Layout

Manufacturing

Chip / Board

instantiation

Test Bench

stimulus → Design under test (DUT) → monitor

type

reg

$5 - Register

$1 - MUX

initial

begin

%b - bit / binary

$monitor - intelligent print statements

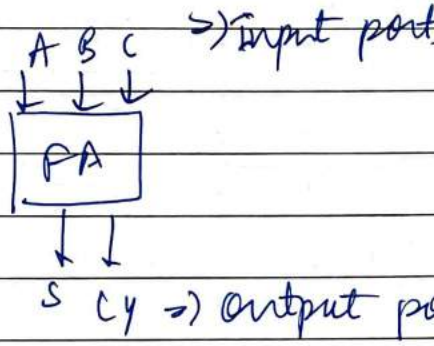
print only when status changes

#5 - after time one apply these inputs

__/__/____

module mod-name (list_of-ports);
input / output declarations
local net declarations
Parallel statements
endmodule

Ex



Ex // A simple AND function
module ~~stand~~ simple and (f, x, y);
input x, y;
output f;
assign f = x & y;
endmodule

This is a behavioral description
Synthesis tool will decide how to realize f:
1. using single AND
2. Using NAND gate followed by NAND

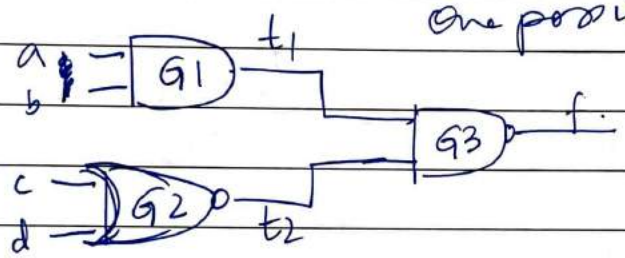
__/__/____

ex: /* 2 level combinational dct */

```
module two_level (a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2;  // intermediate lines
    assign t1 = a & b;
    assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule
```

also behavioral description

One possible gate realization:



t1, t2 → wire data type

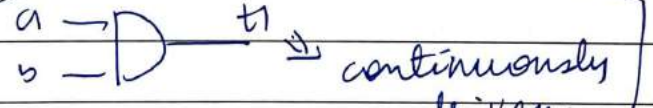
* assign

var expression.

assign t1 = a & b;

models behavioral description

to model combinational dcts



LHS must be "net" type var → here, ex:- wire type

RHS → net type (OR) register type

__/__/____

Data types in Verilog

Net
- continuously driven
- cannot be used to store a value
- to model connections b/w continuous assignments and instantiations

Register
- retains last value assigned to it.
- often used to rep storage elements, but sometimes comb. ckts also.

- reg
- integer
- real : (for floating pt)
- time

- 1 bit values (by default) unless declared as a vector
- Default value of a net is "z" (high impedance / tri-state)

Ex: wire, wor, wand, tri, supply0, supply1 etc.

- wire ≡ tri : ~~multiple~~ drivers o/p are shorted together in case of multiple driver i/p
- wor, wand: inputs OR or AND at connection
- supply 0, 1 – model power supply connection

__/__/__

Data values & signal strengths

4 8.

value level	Rep.
0	Logic 0
1	Logic 1
x	Unknown Logic state
z	High impedance state

Initialization:

- All unconnected nets are set to "z"
- All reg var. set to "x"

Strength	Type
supply	Driving
strong	Driving
pull	Driving
large	Storage
weak	Driving
medium	Storage
small	Storage
high z	High impedance

↑ Strength increases

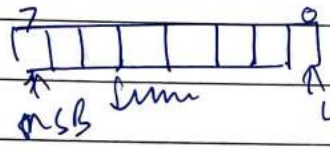
- if 2 signals of unequal str. get driven on a wire, strongr sig will prevail

__/__/____

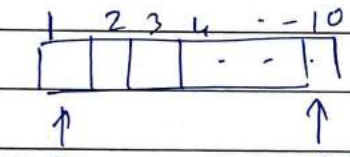
Vectors

Type [range1 : range2] name ;
↓ MSB ↓ LSB

Ex: wire X,y,z ; // Single bit vars
wire [7:0] sum // MSB = sum[7]
LSB = sum[0]



reg [31:0] MDR;
reg [1:10] data ;



Parameter
(kind of like # define)
- cannot specify size
- Ex: parameter H1=25, L0=5;

_/_/_

List of primitive gates:

```
and G ( out, in1, in2);       // and G (a, b, c, d, e, f);
nand G (out, in1, in2);              ↑      ⏟
or  G  (out, in1, in2);           output   inputs
nor     — , —
xor     — , —
xnor    — , —
not G( out, in);
buf G (out, in);


bufif1 G ( out, in, ctrl);
bufif0    — , —
notif0    — —
notif1    — , —
```

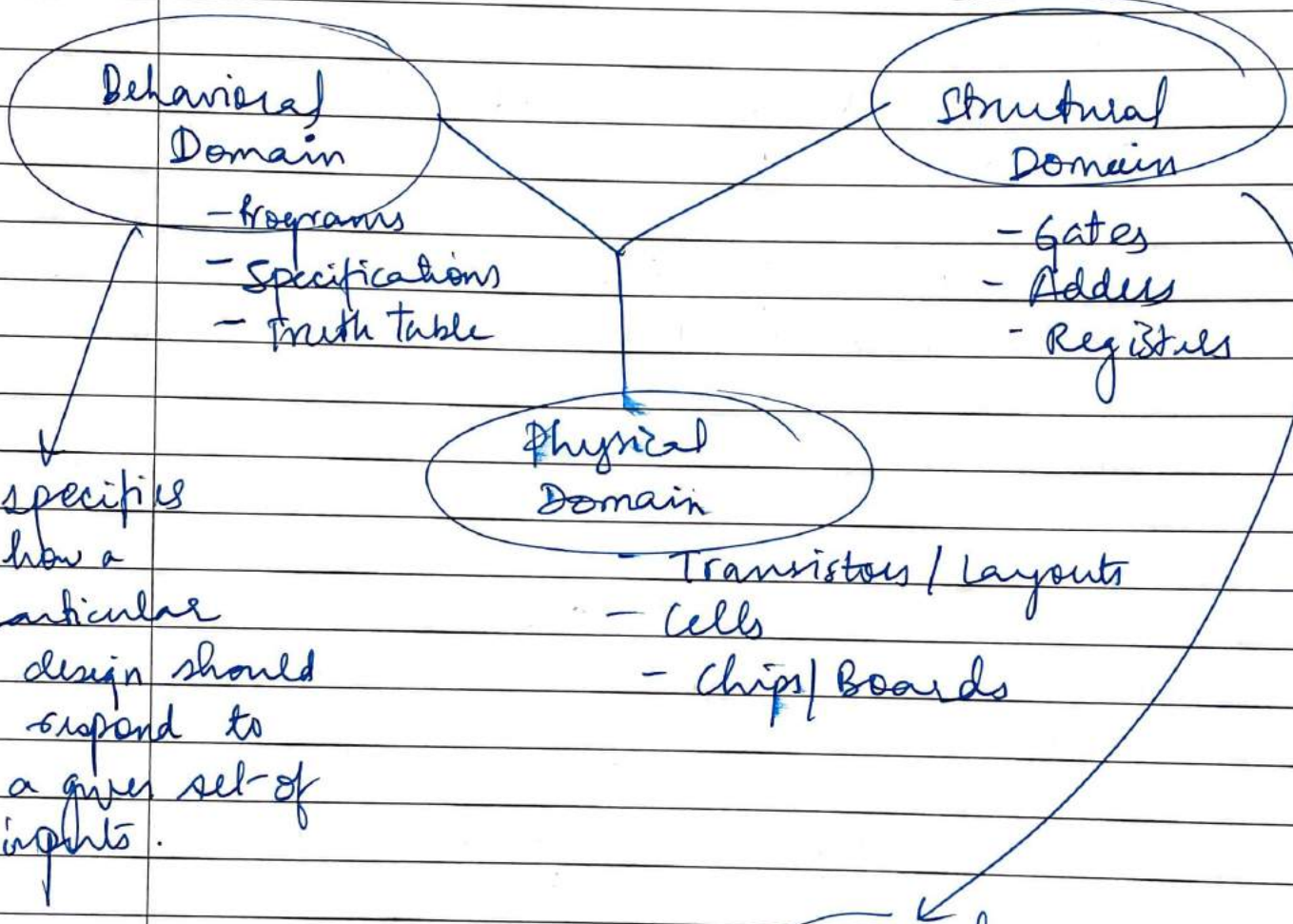
netlist - some building blocks and their interconnections
(in laymans terms)

__/__/____

RISS
LinkedIn
Sombit Dey - IIT K
Shiven Mian

GSOC

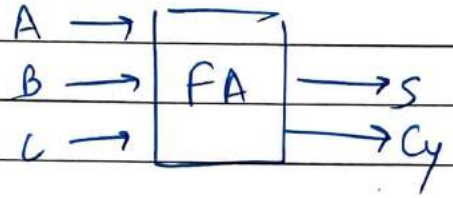
(intro - contd.)



__/__/____

Behavioral rep:: Example → Full Adder

- two operand inputs A and B
- carry input C
- carry output Cy
- sum output S



$$S = A \oplus B \oplus C$$
$$Cy = A.B + A.C + B.C$$

- Verilog

```
module carry ( S, Cy, A, B, C);
   input A, B, C ;
   output S, Cy ;
   assign S = A ^ B ^ C ;
   assign Cy = (A & B) | (B & C) | (C & A);
endmodule
```

- express in verilog in terms of truth table

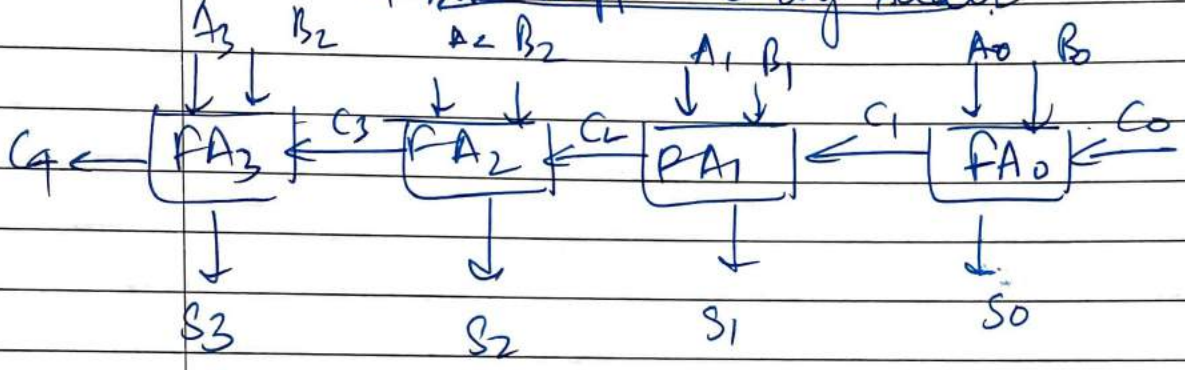
```
primitive carry ( Cy, A, B, C);
   input A, B, C ;
   output Cy ;
   table
   //  A   B   C     :  Cy   ;
       1   1   ?     :  1    ;
       1   ?   1     :  1    ;
       ?   1   1     :  1    ;
       0   0   ?     :  0    ;
       0   ?   0     :  0    ;
       ?   0   0     :  0    ;
   endtable
endprimitive
```

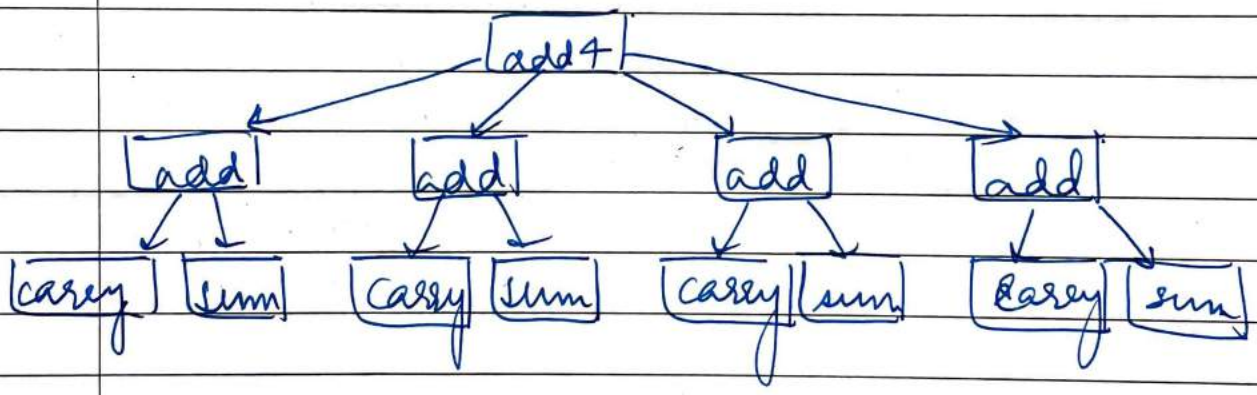
__/__/____

Structural level ; Levels of abstraction are
- the module (functional) level
- the gate level
- the transistor level
- Any combination of above

size of netlist ↑

Ex:: 4 bit Ripple carry Adder





carry = A.B + B.C + C.A

sum = A ⊕ B ⊕ C

```
module add (cy-out, sum, a, b, cy-in);
  input a, b, cy-in;
  output sum, cy-out;
  sum s1 (sum, a, b, cy-in);
  carry c1 (cy-out, a, b, cy-in);
endmodule
```

- invmlog :-

```
module add4 (s, cy4, cy-in, x, y);
  input [3:0] x, y;
  input cy_in;
  output [3:0] s;
  output cy4;
  wire [2:0] cy_out;
  add B0 (cy-out[0], s[0], x[0], y[0], ci);
  add B1 (cy-out[1], s[1], x[1], y[1], cy-out[0]);
  add B2 (cy-out[2], s[2], x[2], y[2], cy-out[1]);
  add B3 (cy-out[2], s[2], x[2], y[2], cy-out[1]);
endmodule
```

instantiation

4 copies of F.A.

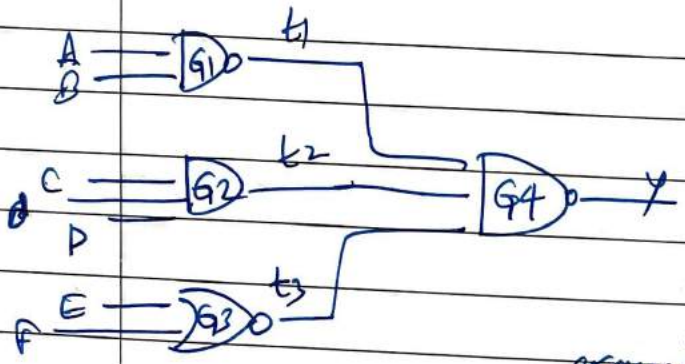
```
module carry (cy-out, a, b, cy-in);
  input a, b, cy-in;
  output cy-out;
  wire t1, t2, t3;
  and g1 (t1, a, b);
  and g2 (t2, a, c);
  and g3 (t3, b, c);
  or g4 (cy-out, t1, t2, t3);
endmodule
```

```
module sum (sum, a, b, cy-in);
  input a, b, cy-in;
  output sum;
  wire t;
  xor x1 (t, a, b);
  xor x2 (sum, t, cy-in);
endmodule
```

__/__/____

VERILOG (contd)

An Example:



example.v

```
module example (A,B,C,D,E,F,Y);
  input A,B,C,D,E,f;
  output Y;
  wire t1, t2, t3;

  nand #1 G1 (t1, A, B);
  and #2 G2 (t2, C, ~B, D);
  nor #1 G3 (t3, E, F);
  nand #1 G4 (Y, t1, t2, t3);
endmodule
```

example-test.v

```
module testbench;
  reg A,B,C,D,E,F; wire Y;
  example DUT (A,B,C,D,E,F,Y);
```

(Contd) →

```
initial
 begin
  $monitor ($time, " A=%b, B=%b,
                    D=%b, E=%b, F=%b, Y=%b",
                    A, B, C, D, E, F, Y);
  #5 A=1; B=0; C=0; D=1; E=0; F=0;
  #5 A=0; B=0; C=1; D=1; E=0; F=0;
  #5 A=1; C=0;
  #5 F=1;
  #5 $finish;
 end
endmodule
```

Command in iverilog

a) iverilog -o mysim example.v example-test.v
b) vvp mysim

⇒ after begin add these 2 lines:

```
$dumpfile ("example.vcd");
$dumpvars (0, testbench);
$monitor ....
```

to display the waveforms:

gtkwave example.vcd

__/__/____

Verilog operators

- Arithmetic
  - + , - , * , / , % , ** (power)
  - many binary
- Logical
  - ! , && , ||
- Relational
  - != , == , >= , <= , > , <
- Reduction (bitwise)
  - &
  - |
  - ~&
  - ~|
  - ^
  - ~^
- shift operators
  - << , >> , >>>
- Conditional operators
  - ~~cond ? true~~ cond_exp ? true_exp : false_exp

__/__/____

Operator precedence

```
+ - ! ~   (unary)
* *
* / %
<< >>  >>>
< <= > >=
== !=  === !==
&  ~&
^  ~^
|  ~|
&&
||
?:          ( R→L )
{}, {{}}  (concat & repetition)
```

Note: operators (L→R) except conditional. (R→L)

:

(1) presence of a 'z' or 'x' in a reg or wire being used in an arithmetic expression results in whole expression being unknown ('x')

(2) logical operators (!, &&, ||) all evaluate to a 1 bit result (0, 1 or x).

~~(3) The~~

(3) Boolean — true ≅ 1'b1
false ≅ 1'b0

__/__/____

System tasks and functions

- **Task**
  can be invoked from diff. parts of the design
  zero or more return values

- **Function:**
  return only one value
  delays not allowed

- **System task**

  begins with $, no delays
  Ex: $display("Hello");
  $time → returns current simulation time

1'b0 → 1 bit no expressed in binary format with value = 0

4'b1011 → 4 bit value in binary that is equivalent to magnitude = 11

__/__/____

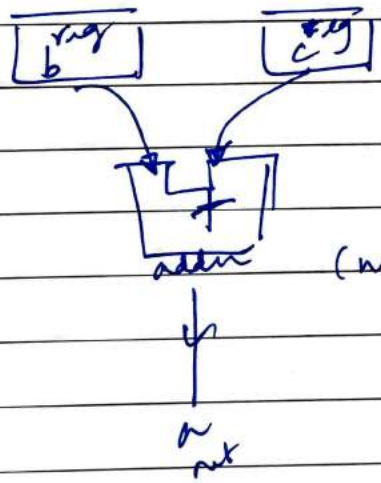
Verilog description styles

- Dataflow
  - continuous assignment (using assignment statements)
- Behavioural
  - Procedural assignment
    - Blocking
    - Non-blocking

(using procedural statements similar to a program in a high level language)

Data flow Style

```
assign a = b+c
assign sign = Z[15];
```



(no concept of clock)

- forms a static binding b/w. net (type & RHS (net or reg) (LHS)

- assignment is continuously active:
  - mostly used to model combinational ckts

__/__/____

Examples → using assign statement to describe a sequential logic element (D Latch) SR Latch.
see in modelling part ✓

PROCEDURAL ASSIGNMENT / Behavioral style

begin-end only in case of multiple statements inside the block

2 kinds in verilog:

- the "initial" block.
  - Executed once at the beginning of simulation
  - Used only in test benches, can't be used in synthesis

initial begin end

- the "always" block
  - continuous loop that nvr terminates

Procedural block defines:
- region of code containing sequential statements
- the statements execute in the order they are written

always (. . . .)
begin
  ⋮
end

Can be used to update variables of type (LHS)
- reg
- integer
- real
- time

__/__/____

- some shortcuts in declarations:

1) "output" and "reg" can be declared together.

Ex: output reg [7:0] data;
instead of output [7:0] data; reg [7:0] data;

2) A variable block can be initialized when it is declared:
Ex reg clock = 0;
instead of reg clock; initial clock = 0;

- the 'always' block

- multiple statements are grouped together using "begin ... end"
- "always" statement starts at time 0 and executes the statements inside the block repeatedly, and nvr stops

Ex - clock signal

- We can specify delays for simulations however, for real ckts the clock generator will be active as long as there is power supply.

* 'initial' and 'always' blocks can co-exist within the same 'verilog module

- They all execute concurrently;
  initial - only once
  always - indefinitely

* Basic syntax of always block:

```
always @(event - expression)
  begin
    —
    —
    —
  end
```

* Only "reg" type variable can be assigned within an "initial" or "always" block

Reason:
- Sequential "always" block executes only when the event expression triggers.
- At other times block is doing nothing
- Obj must remember last value assigned (not continuously driven)

event expression can have any type of var (reg, wire etc)

Sequential statements in verilog

(a) begin ... end.

(b) if ... else

```
if (<exp>)
  seq-stmnt(s);
else
  seq-stmnt(s);
```

use begin-end for group.

(c) case

```
case (<exp>)
  exp1 : seq-stmnt;
  exp2 : seq-stmnt;
  :
  exp#n : seq-stmnt;
  default : default-stmnt;
endcase
```

2 variations :- casex and casez :

(1) "casez" statement treats all 'z' values in the case alternatives or case exp as DONT CARES

(2) "casex" treats all 'x' and 'z' as DONT CARES

__/__/____

(d) "while" loop.

while (<exp>)
    seq-stmnt(s);

(e) "for" loop

for (expr1 ; expr2 ; expr3)
    seq - stmnt;

(f) "repeat" loop → can be constant / var / signal value

repeat (<exp>)
    seq-stmnt;

then evaluated only when the loop starts & not during execution of loop.

// in case of while - loop runs as long as condition is true

// in repeat - ~~we specify no of times the loop needs to run~~ executes the loop a fixed no of times

Ex. repeat (a) ← initial = 10
    begin
    =
    = // say a gets updated to 15
    end

but still loop will only execute 10 times (initial value)

__/__/____

g) forever loop

forever
  seq-stmt;

→ executes forever until $finish is encountered in the test bench.

Note

The forever loop is typically used along with a timing specifier (delay)

- If delay is not specified, the simulator would execute this stmt indefinitely without advancing $time
- Rest of the design will never be executed

Ex

```
// clock generation using "forever"
reg clk;

initial
  begin
    clk = 1'b0;
    forever #5 clk = ~clk // clock period of 10 units
  end
```

⇒ Other constructs Available

* #(time-value)
  - makes a block suspend for time-value units of time
  - time unit can be specified using 'timescale command

__/__/____

* @(event_expression)

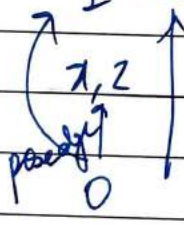
→ makes block suspend until "event_expression" triggers

→ event can be :

1. change of signal value
2. positive or negative edge occuring on signal (posedge or negedge)
3. List of above mentioned events, separated by "or" or comma.

note : @(*) // any variable changes.

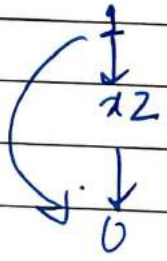
→ posedge



transition from {0, x, z} to 1
or
from 0 to {x, z}

Ex always @ (posedge clk)
{

→ negedge



Types of procedural statements

__/__/____

BLOCKING / NON-BLOCKING ASSIGNMENTS

↓ denoted by ("=")

↓ denoted by ("<=")

(1) Blocking Assignment

```
var_name = [delay_or_event_control] expression;
```

→ Blocking assignment statements are executed in the order they are specified in a procedural block.
- target of an assignment gets updated before the next sequential statement in the procedural block is executed.
- They do not block execution of statements in other procedural blocks.

→ recommended style for comb. logic.

→ eg. logic when case is used (incomplete info then)

RHS will be evaluated first

② Non-blocking

```
var_name <= [delay or event control] expression;
```

→ allows scheduling of assignments without blocking execution of statements that follow within the procedural block.

- the assignment to the target gets scheduled for the end of simulation cycle (at end of procedural block)

- Statements subsequent to the instruction under consideration are not blocked by the assignment.

- Allows concurrent procedural assignment, suitable for seq. logic.

Note

A variable cannot appear as the target of both a blocking and non blocking assignment.

- This is not allowed:

```
x = x + 5;
x <= y;
```

__/__/____

Ex: swapping 2 var a & b

Blocking

```
always @(posedge clk)
  a = b;
always @(posedge clk)
  b = a;
```

either a=b will execute 1st or vice-versa depending on simulator implementation

Both registers will get the same value (either a or b) (Race condition)

Non-blocking

```
always @(posedge clk)
  a <= b;
  b <= a;
```

correct swap

correct swap →

```
always @(posedge clk)
  begin
    ta = a;
    tb = b;
    a = tb;
    b = ta;
  end
```

some ex:

①

```
begin
  a = #5 b;
  c = #5 a;
end
```

BA

	a	b	c
#5	b (a=b)	b	c
10	b	b	b.

c will be assigned val = b after 10 time units.

```
begin
  a <= #5 b;
  c <= #5 a;
end
```

NBA.

	a	b	c
5	b	b	b

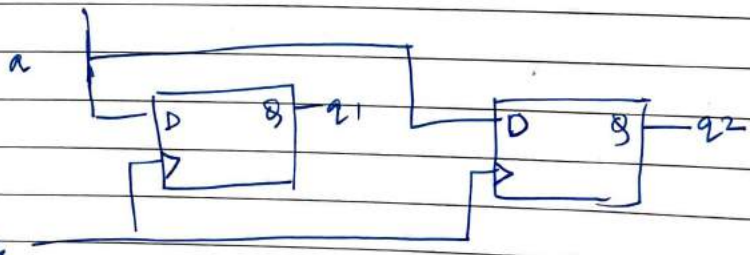
c will be assigned val = b after 5 time units.

②

```
always @(posedge clk)
  begin
    q1 = a;
    q2 = q1;
  end
```

synthesis tool will generate ↓

→ Parallel FFs



but check

try the same using NBA and see what synthesis tool generates?

↓

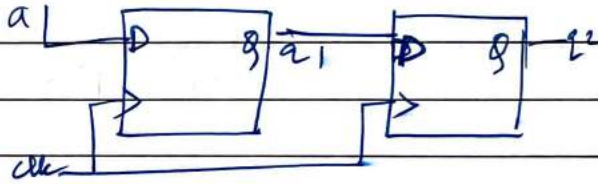
should directly connect $q1 \rightarrow D_2$ || shift Reg

↑ check if Dff is only generated by writing test-bench

__/__/____

③
```
always @(posedge clk)
  begin
    q2 = q1;
    q1 = a;
  end
```

```
always @(posedge clk)
  begin
    q1 <= a;
    q2 <= q1;
  end
```



both will generate ⇒ shift reg

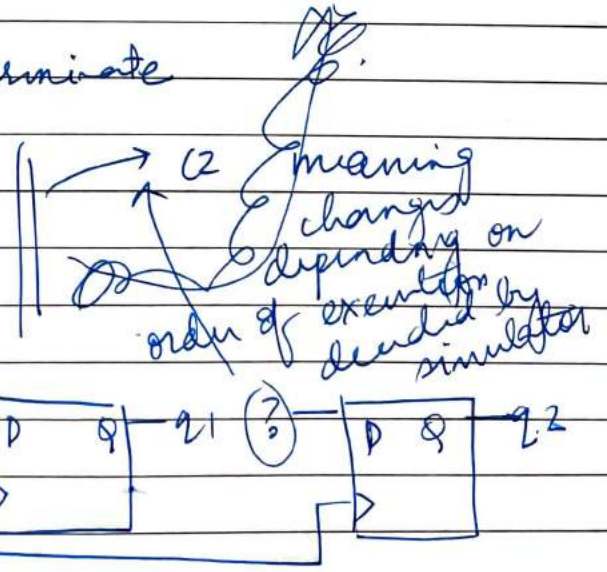
④
```
always @(posedge clk)
  q2 <= q1;
always @(posedge clk)
  q1 <= a;
```

⇒ will generate SR

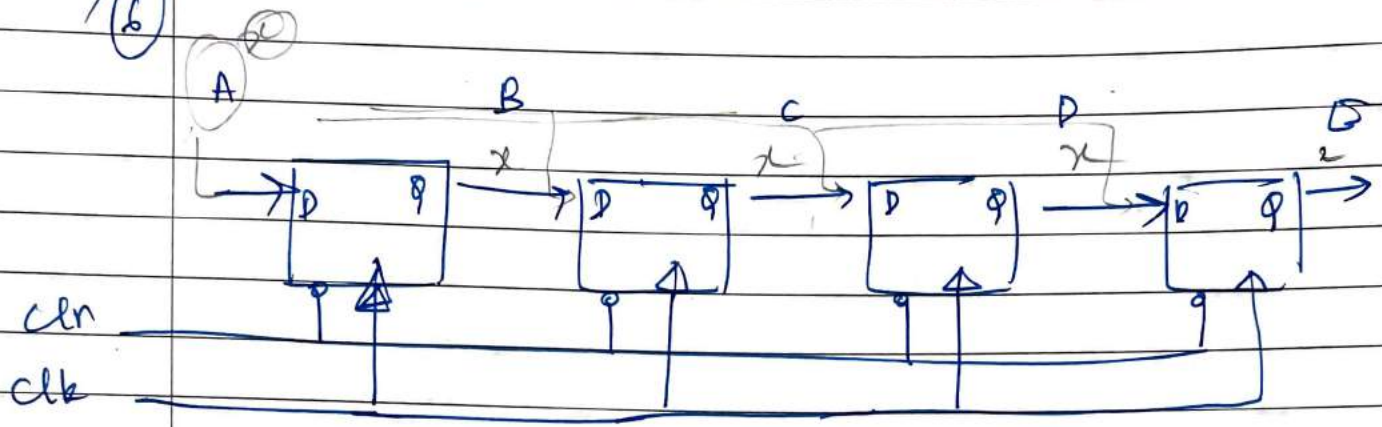
q2 will be indeterminate

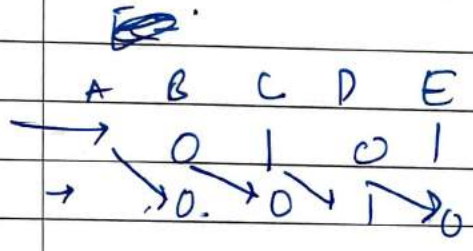
⑤
```
always @(posedge clk)
  q1 = a;
always @(posedge clk)
  q2 = q1;
```

don't know which will execute first



(6)





E = D
D = C
C = B
B = A

interchanging with prev value ✓

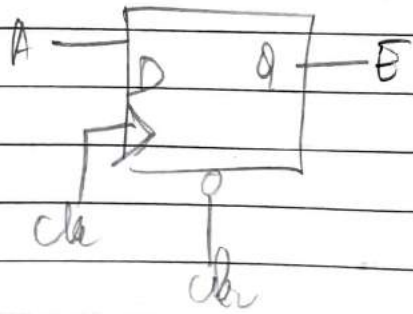
if done using NBA
both of these will end up in SR only

B = A
C = B ✗
D = C
E = D

→ all will get value = x.

↓ same value
E = A

↳ this will instead end up in →



__/__/____

generate blocks

→ "generate" statements allow Verilog code to be generated dynamically before the simulation or synthesis begins.

- very convenient to create parameterized module descriptions.
  ex (N) bit ripple carry adder.
  arbitrary

→ "generate" , "endgenerate"

→ special "genvar" variables:

- genvar - to declare variables that are only used in the evaluation of generation block.
- these var ~~the~~ DNE during simulation/synthesis.
- value of "genvar" can only be defined in a generate loop.
- every generate loop is assigned a name, so that variables inside the generate loop can be referenced hierarchically.

__/__/____

USER DEFINED PRIMITIVES (UDP)

- used to define custom Verilog primitives by the use of lookup tables.

- They can specify:
  - truth table for comb. fxns
  - state table for seq. fxns
  - Don't care, rising & falling edges etc

- for comb fxn, the truth table entries are specified as: ↳ use '?' for don't care

<input1> <I/P 2> ... <I/P N> : <O/P> ;

- for seq. fxn, state table entries:

<I/P1> ... <I/P N> : <present_state> : <next_state>

* Rules for using UDPs

① I/P terminals to a UDP can only be scalar variables
- Multiple I/P terminals can be used.
- Input entries in the table must be in the same order as "input" terminal list

② Only one scalar O/P terminal
- O/P terminal must appear in the beginning of terminal list.

__/__/____

- UDPs
  - comb :- output terminal declared as 'output'
  - seq :- declared as 'reg'

(3) for seq UDP, state can be initialized with an "initial" statement. (optional)

Some guidelines

(1) UDPs model functionality only. They do not model timing or process technology. (delays)

(2) A functional block can be modelled as UDP only if it has exactly one output.
- If a block has >1 output - model as module
- or use multiple UDPs.

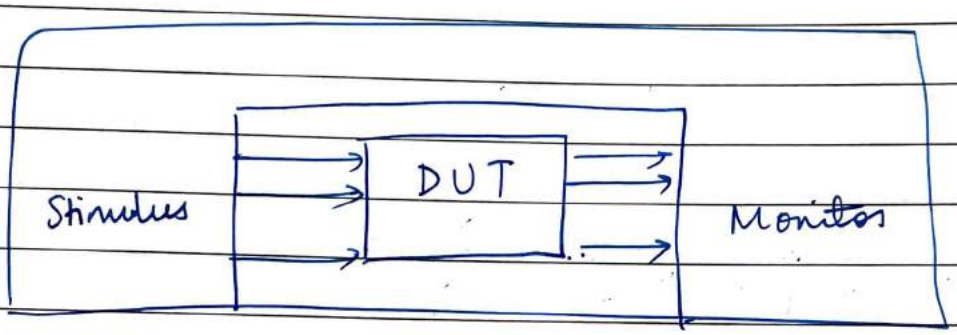
(3) Inside simulator, UDP is implemented as a lookup table in memory.

(4) UDP state tables should be specified as completely as possible.
- for unspecified cases, output is set to "x".

__/__/__

Verilog Test Bench-

- procedural block that executes only once
- Test bench generates clock, reset, and the required test vectors for a given design - under-test (DUT).



input in main module ⟶ reg in test bench
output —"— ⟶ wire —"—

$display
$monitor
$finish
$dumpfile (<filename>) ; ← extension .vcd.
$dumpoff
$dumpon
$dumpvars (level, list_of_var_or_module)
$dumpall
$dumplimit (filesize);

__/__/__

Modelling state machines (FSMs)

Mealy & Moore FSM types

A deterministic FSM can be mathematically defined as a 6 tuple:

$$(\Sigma, \Gamma, S, s_0, \delta, w)$$

Σ - set of I/P combinations
Γ - set of O/P combinations
S - finite set of states
$s_0 \in S$ - initial state
δ - state transition function
w - output function.

Here $\delta : S \times \Sigma \rightarrow S$

- Present state (PS) and present input determine next state (NS).

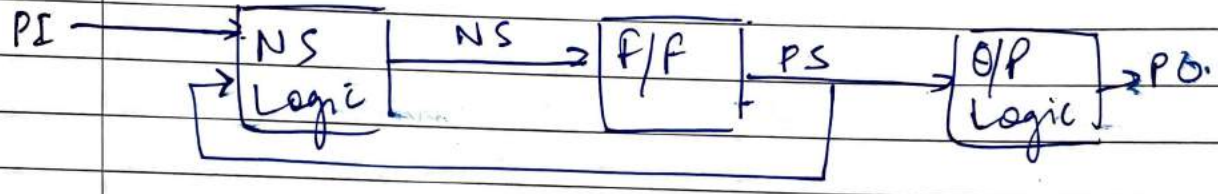
Mealy Machine	Moore Machine
$w : S \times \Sigma \rightarrow \Gamma$	$w : S \rightarrow \Gamma$
O/P depends on state + input	O/P depends only on state.

__/__/____

Pictorial depiction

① Mealy

② Moore



Datapath and controller Design

In complex digital system, h/w is ~~divided~~ partitioned into 2 parts:

a) Datapath

consists of functional units where all computations are carried out.

(registers / muxs / buses / adders / multipliers / counters / other functional blocks)

we are not specifying - telling exactly what to do with these hardware

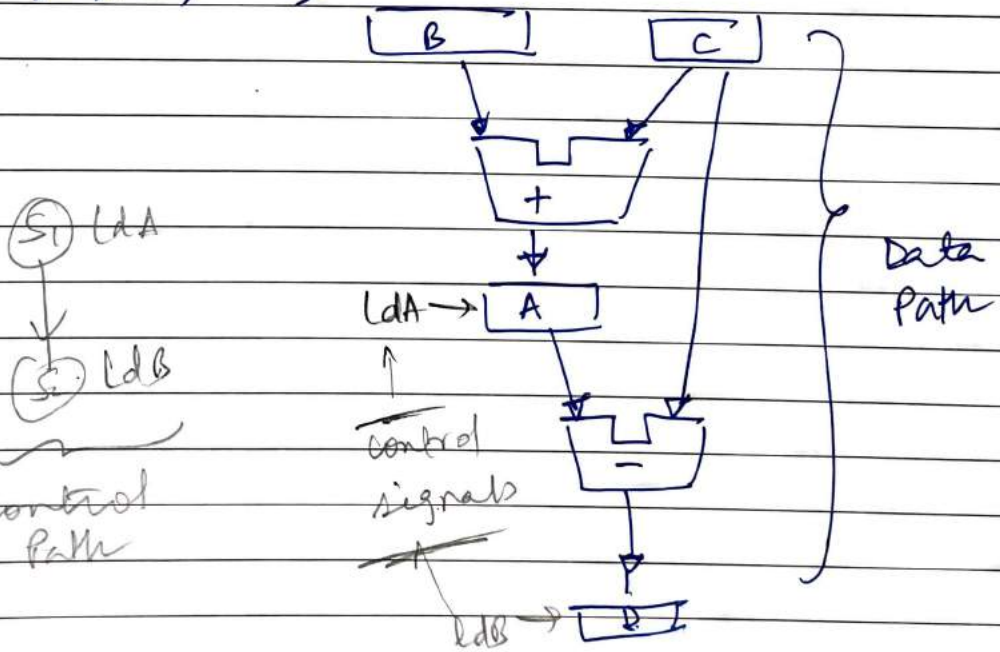
(b) Control Path

implements FSM and provides control signal to data path in proper sequence.

- In response to control signal, various operations are carried out by the data path.
- Also takes input from the data path regarding various status info.

Ex (simple)

```
reg [15:0] A, B, C, D;
A = B + C;
D = A - C;
```

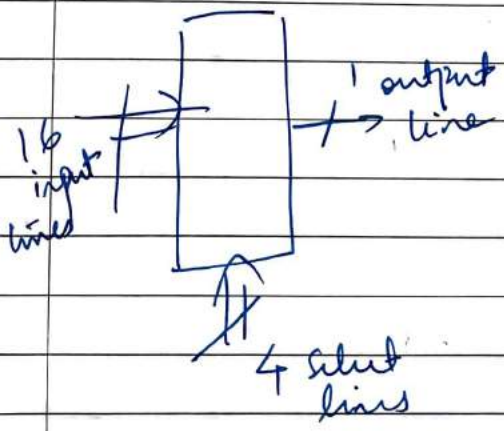


(Modelling)

__/__/__

Verilog ex1 - MUX

16 to 1 mux



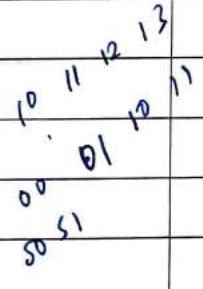
Steps

(1) Using pure behavioral modelling

(2) structural modelling using 4 to 1 mux specified using behavioral model

(3) Make structural modeling of 4 to 1 mux, using behavioral modeling of 2 to 1 mux

(4) Make structural gate level modelling of 2 to 1 mux, to have complete hierarchical description



(1)

```
module mux16to1 ( in , sel, out);
    input [15:0] in;
    input [3:0] sel;
    output out;

    assign out = in [sel];
endmodule
```

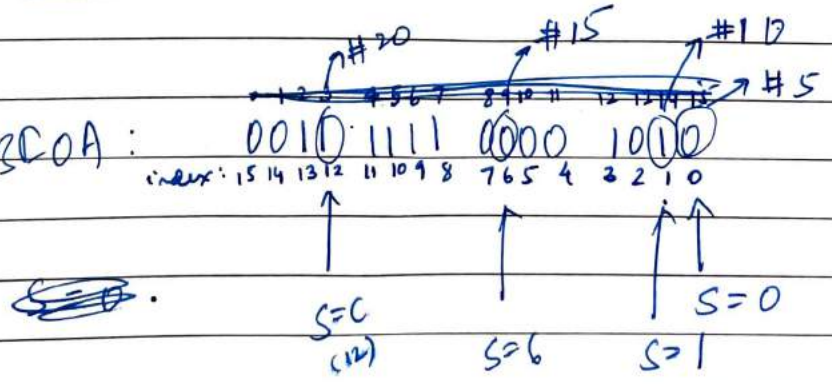
__/__/____

```
module muxtest;
  reg [15:0] A; reg [3:0] S; wire F;
  mux16to1 M(.in(A), .sel(S), .out(F))

  initial
    begin
      $dumpfile("mux16to1.vcd");
      $dumpvars(0, muxtest);
      $monitor($time, "A=%0h, S=%0h, F=%0b", A, S, F)
      #5 A=16'h3f0a; S=4'h0;
      #5 S=4'h1;
      #5 S=4'h6;
      #5 S=4'hC;
      #5 $finish;
    end
  endmodule
```

for GTK wave only

o/p

```
0 A=xxxx, S=X, F=X
5 A=3f0a, S=0, F=0
10 A=3f0a, S=1, F=1
15 A=3f0a, S=6, F=0
20 A=3f0a, S=C, F=1
```

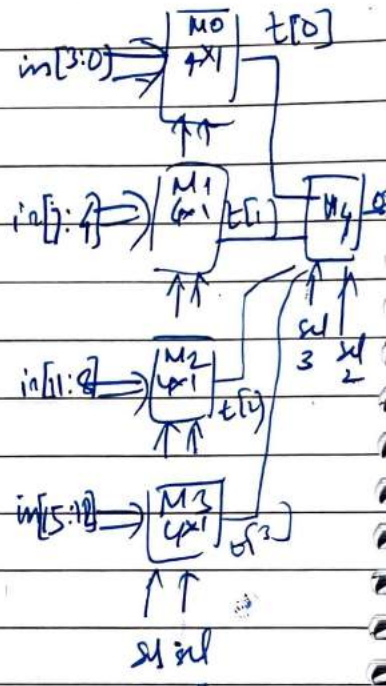


(1)

```
module mux4to1(in, sel, out);
    input [3:0] in;
    input [1:0] sel;
    output out;

    assign out = in[sel];
endmodule
```

```
module mux16to1 (in, sel, out);
    input [15:0] in;
    input [3:0] sel;
    output out;
    wire [3:0] t;

    mux4to1  M0  ( in[3:0], sel[1:0], t[0] );
    mux4to1  M1  ( in[7:4], sel[1:0], t[1] );
    mux4to1  M2  ( in[11:8], sel[1:0], t[2] );
    mux4to1  M3  ( in[15:12], sel[1:0], t[3] );
    mux4to1  M4  ( t, sel[3:2], out );
```

instantiate 5 times



(2)

```
module mux2to1 (in, sel, out);
    input [1:0] in;
    input sel;
    output out;

    assign out = in[sel];
endmodule
```

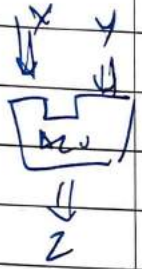
```
module mux4to1 (in, sel, out);
    input [3:0] in;
    input [1:0] sel;
    output out;

    mux2to1 M0 (in[1:0]
                sel[0], t[0]);
    mux2to1 M1 (in[3:2]
                sel[0], t[1]);
    mux2to1 M2 (t, sel[1], out);
endmodule
```

purpose of zero flag? → in operations like for loop → we might need to detect a zero

__/__/__

Verilog Ex2. ADDER

Generation of status flags: sign
zero → whether result is 0.
carry
parity
overflow → whether sum can fit in 16 bits




```
module ALU(X,Y,Z,sign,zero,carry,Parity,Overflow);
input [15:0] X,Y;
output [15:0] Z;
output sign,zero,carry,Parity,Overflow;

assign {carry,Z} = X+Y;   //16bit addition
assign sign = Z[15]; //MSB
assign zero = ~|Z;
assign Parity = ~^Z;
```

in behavioral fashion
concat operation
16+1=17 bits
reduction operator NOR
reduction operator
XNOR

```
assign Overflow = (X[15] & Y[15] & ~Z[15]) |
                  (~X[15] & ~Y[15] & Z[15]);

endmodule
```

__/__/____

```
module alutest;
  reg [15:0] X,Y;
  wire [15:0] Z;
   wire S,ZR,CY,P,V;
  ALU DUT (X,Y,Z,S,ZR,CY,P,V);

  initial
    begin
      $dumpfile("alut.vcd");
       $dumpvars(0,alutest);
      $monitor($time,"X=%0h,Y=%0h,Z=%0h,S=%0b
                Z=%0b,CY=%0b,P=%0b,V=%0b",
                X,Y,Z,S,ZR,CY,P,V);
      #5 X=16'h8fff; Y=16'h8000;
      #5 X=16'hfffe; Y=16'h0002;
      #5 X=16'hAAAA; Y=16'h5553;
      #5 $finish;
    end
endmodule
```

GTK

Obs:

#5

16'h8fff.

x= 1000 1111 1111 1111

16'h8000;

y= 1000 0000 0000 0000.

z= 0111 1111 1111 1111

z= 0fff

s = 0

#10

1111 1111 1111 1110

1 0000 0000 0000 0010

0000 0000 0000 0000

z= 0000 s>0.

#15

1010 1010 1010 1010

0101 0101 0101 0101

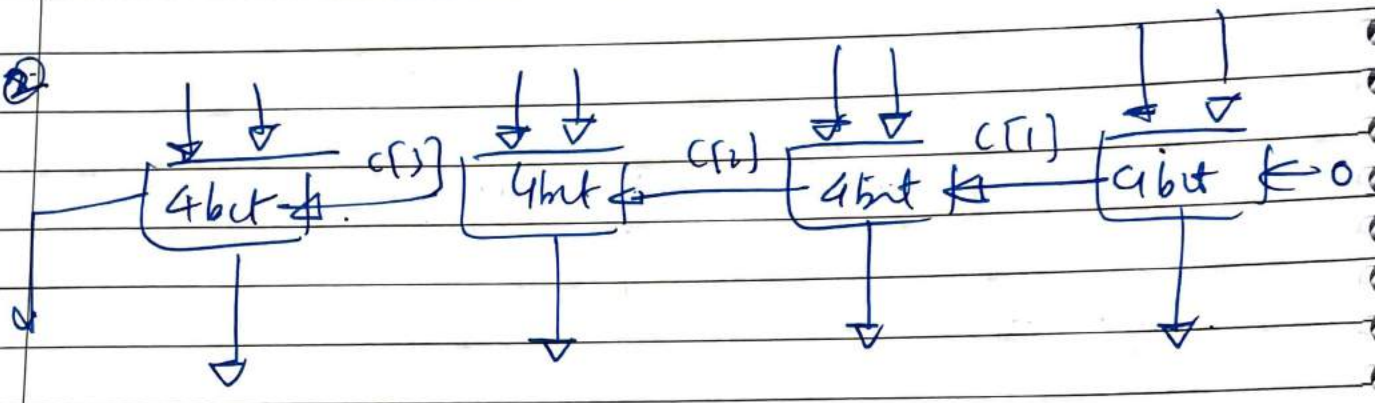
1111 1111 1111 1111

z= ffff

s = 1

__/__/____

②




```
module adder4( S, cout , A, B, cin);
    input [3:0] A, B;
    input cin;
    output [3:0] S;
    output cout;

    assign {cout,S} = A + B+cin;
endmodule
```

module ALU

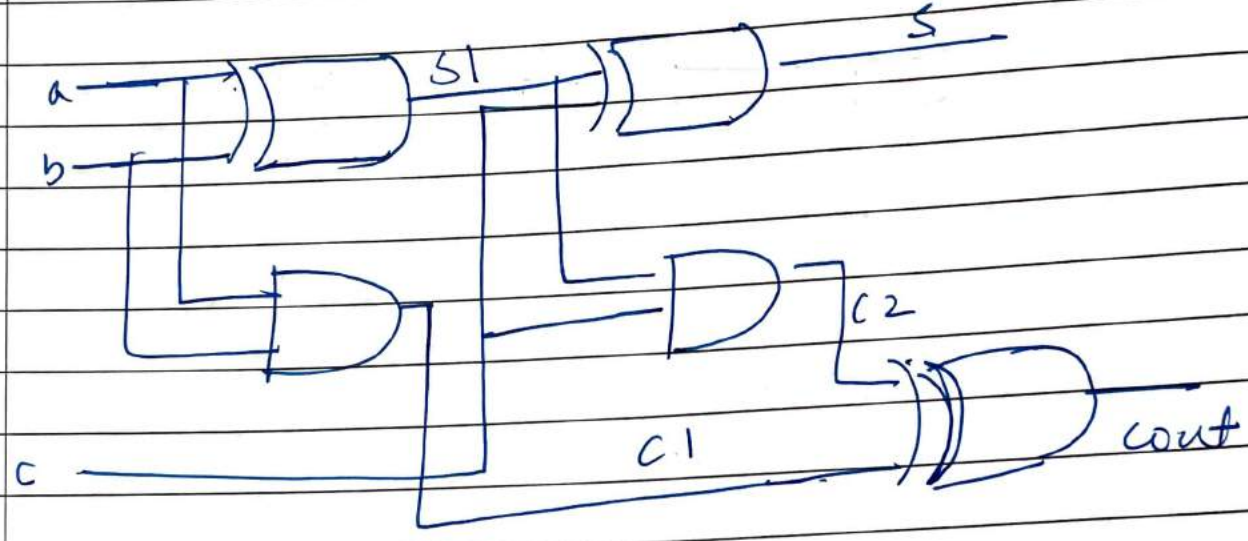
use c[3:1]

remove the concat operation.

instantiate 4 adders in module ALU

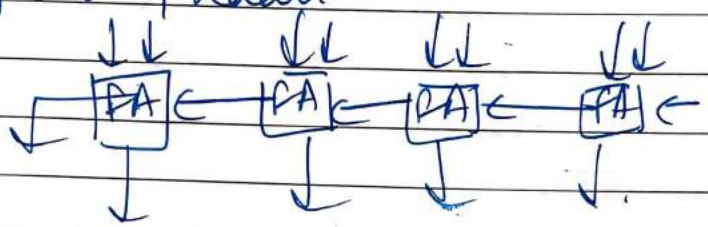
```
A0 ( Z[3:0] , c[1], X[3:0] , Y[3:0] , 1'b0 ).
A1 ( Z[7:4] , c[2] , X[7:4] , Y[7:4] , c[1] )
A2 ( Z[11:8] , c[3] , X[11:8] , Y[11:8] , c[2] ),
A3 ( Z[15:12] , Carry , X[15:12] , Y[15:12] , c[3] );
```

(3) structural modelling of Ripple carry Adder

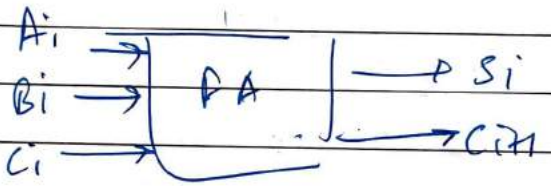



```
module add4 ( S, cout, A, B, cin );
    input [3:0] A, B;
    input cin;
    output [3:0] S;
    output cout;
    wire c1, c2, c3;

    fulladd FA0 ( S[0], c1, A[0], B[0], cin);
        — . — 1 ( S[1], c2, A[1], B[1], c1);
        — . — 2 ( S[2], c3, A[2] B[2], c2);
        —— - 3 ( S[3], cout, A[3], B[3], c3);
endmodule
module full adder ( s, cout, a, b, c);
    input a, b, c;
    output s, Cout;
    wire s1, c1, c2;
    xor G1 (s1, a, b), G2 ( s, s1, c), G3(Cout, c2, c1);
    and G4 (c1, a, b), G5 ( c2, s1, c);
endmodule
```

ripple carry adder:



slower than carry lookahead adder :→ generate all carries in 11 st



$C_{i+1} = g_i + p_i \cdot C_i$

carry generate
$g_i = A_i \cdot B_i$

carry propagate
$p_i = A_i \oplus B_i$

__/__/____

Ex 3: Level sensitive D latch

```
module level-sensitive_latch (D, Q, En);
    input D, En;
    output Q;
    assign Q = En ? D : Q ;
endmodule
```

enable

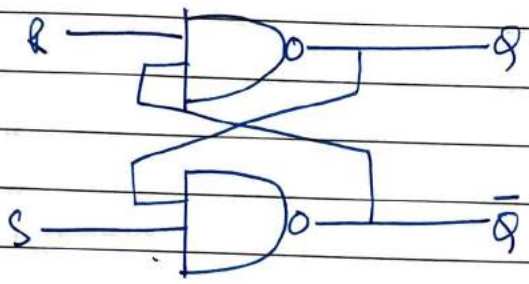
if enable - true
Q → D

else
Q will remember Q
(memory)

En	D	Q
0	x	Q_{n-1}
1	0	0
1	1	1.

Generates D-type latch

Ex 4: ~~Ex 4~~ Simple S-R latch



S	R	Q_n
1	1	Q_{n-1}
0	1	0
1	0	1
0	0	?

```
module sr_latch (Q, Qbar, S, R);
    input S, R;
    output Q, Qbar;

    assign Q = ~(R & Qbar);
    assign Qbar = ~(S & Q);
endmodule
```

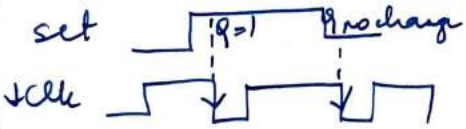
__/__/__

```
module latchtest ;
  reg S, R ;
  wire Q, Qbar ;
  sr_Latch LAT (Q, Qbar, S, R);
  initial
    begin
    $monitor ($time , "S = %b, R = %b, Q = %b,
                       Qbar = %b", S, R, Q, Qbar);
    S = 1'b0 ; R = 1'b1 ;
  #5  S = 1'b1 ; R = 1'b1 ;
  #5  S = 1'b1 ; R = 1'b0 ;
  #5  S = 1'b1 ; R = 1'b1 ;
  #5  S = 1'b0 ; R = 1'b0 ;
  #5  S = 1'b1 ; R = 1'b1 ;
    end
endmodule
```

O/P:

0	S=0,	R=1,	Q=0,	Qbar=1
5	1	1	0	1
10	1	0	1	0
15	1	1	1	0
20	0	0	1	1

and then simulator hangs



__/__/__

Ex5: D Flip Flop with synchronous set & reset

changes take place at the active edge of clk

```
module dff (q, qbar, d, set, reset, clk);
    input d, set, reset, clk;
    output reg q; output qbar;
    assign qbar = ~q;

    always @ (posedge clk)
      begin
        if (reset == 0)  q <= 0;
        else if (set == 0)  q <= 1;
        else q <= d;
      end
endmodule
```

DFF with asynchronous set & reset

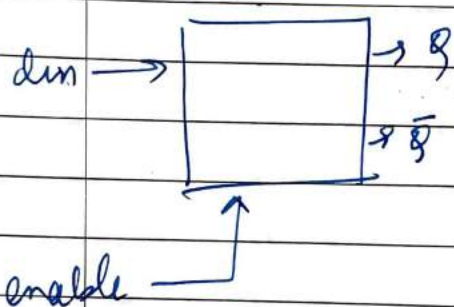
```
module dff (q, qbar, d, set, reset, clk);
    input d, set, reset, clk;
    output reg q;  output qbar;
    assign qbar = ~q;

    always @ ( posedge clk or negedge set or
                negedge reset)
      begin
        if (reset == 0)  q <= 0;
        else if (set == 0)  q <= 1;
        else q <= d;
      end
endmodule
```

__/__/____

Sequential DFF

```
module dff_negedge (D, clk, Q, Qbar);
    input D, clk;
    output reg Q, Qbar;

    always @(negedge clock)
        begin
            Q = D;
            Qbar = ~D;
        end
endmodule
```

Ex 6: Transparent latch with enable



latch → level triggered

```
module latch (q, qbar, din, enable);
  input din, enable;
  output reg q; output qbar;

  assign qbar = ~q;
  always @ (din or enable)
    begin
      if (enable) q = din;
    end
endmodule
```

__/__/____

Eg 7: 4 bit counter with asynchronous reset. (0 to 15)

```
module counter (clk, rst, count);
    input clk, rst;
    output reg [3:0] count;

always @(posedge clk or posedge rst)
  begin
   if(rst)
      count<=0
   else
      count <= count + 1
  end
endmodule
```

~~round~~

__/__/__

Ex 8: Sequential logic ex

```
module incomp_state_type ( curr_state, flag);
    input [0:1] curr_state;
    output reg [0:1] flag;

    always @ (curr_state)
      case ( curr_state)
          0,1 : flag=2;
          3   : flag=0;
      endcase
endmodule
```

curr_state	flag
0	2
1	2
2	?
3	0

[I]

Latch will be generated for flag but why sequential ?

Verilog simulator will assume :
if some of the input descriptions are missing in the case statement (here 2)

⇒ if curr_state is 2, then flag will not change
↳ flag will map to a storage element (latch).

__/__/____

try to avoid unnecessary generation of latch.

in the prev program, modify by adding

```
reg=0
after begin
```

this will also be the state at case2.

no latch will be generated as now it is a comb. ckt.

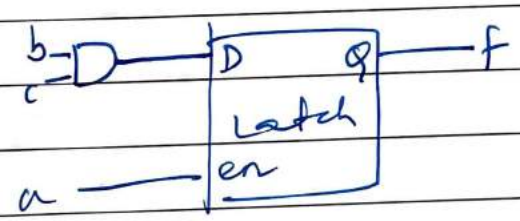
Ex 2:

→
```
module xyz (input a,b,c, output reg f)
  always @(*)
    if(a==1) f=b&c;
endmodule
```

↳ for a=0, value of f unspecified

↓

Latch is generated

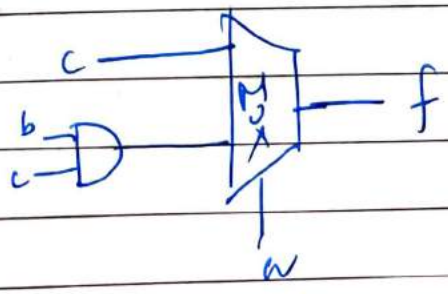


→
```
module xyz( input a,b,c , output reg f)
  always @(*)
    begin
      f=c;
      if(a==1) f=b&c;
    end
endmodule
```

← when a=0

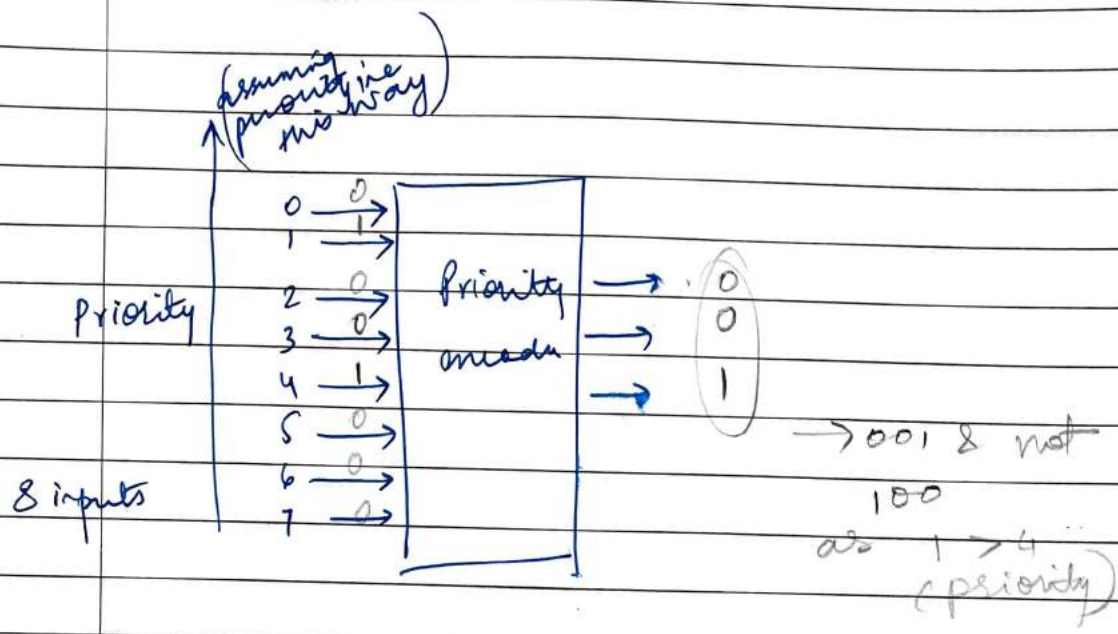
will generate mux

↓



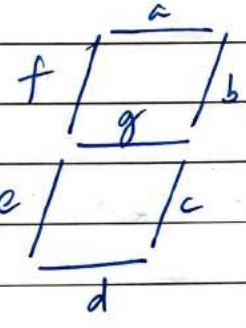
__/__/__

ex 9 Priority encoder




```
module priority_encoder(in, code);
    input [7:0] in;
    output [2:0] code;
                          ← whenever there is change in input
    always @(in)
    begin
        if (in[0])        code = 3'b000
        else if (in[1])   code = 3'b001
        else if (in[2])   code = 3'b010
        else if (in[3])   code = 3'b011
        else if (in[4])   code = 3'b100
        else if (in[5])   code = 3'b101
        else if (in[6])   code = 3'b110
        else if (in[7])   code = 3'b111
        else              code = 3'bxxx
    end
endmodule
```

Eg 10: BCD to 7 segment decoder



segment bit assignment:
(a, b, c, d, e, f, g)

A segment glows when corresponding bit of seg is 0.

```
module bcd-to-7seg (bcd, seg);
    input [3:0] bcd;
    output reg [6:0] seg;
    always @(bcd)
      case
          0 : seg = 7'b 0000001
          1 : seg = 6'b 1001111
          2 : seg = 6'b
          3 : seg = 6'b
          4 : seg = 6'b
          5 : seg = 6'b
          6 : seg = 6'b
          7 : seg = 6'b
          8 : seg = 6'b
```

```
        9:  seg = 6'b0000100;
    default: seg = 6'b1111111;
endcase
endmodule
```

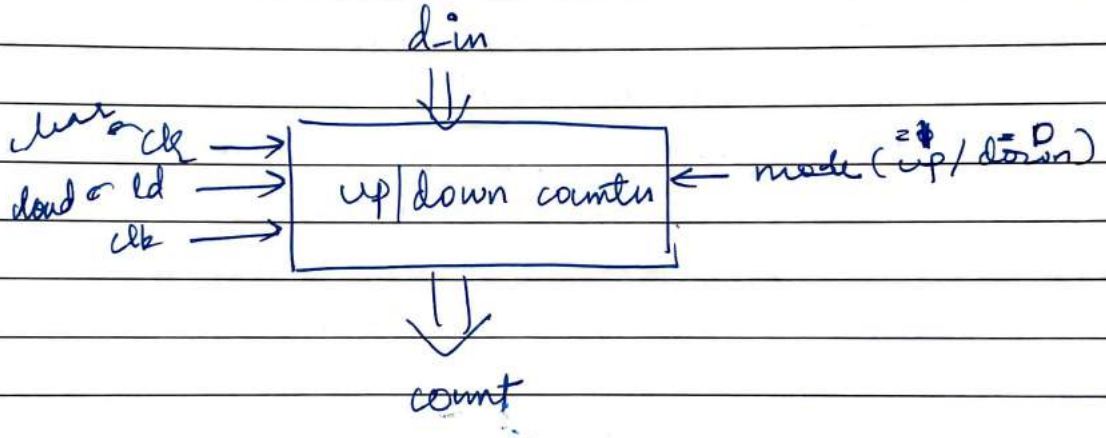
ex 11: n bit comparator

↳ for actual synthesis, it is common to have a structural design representation of the comparator

```
module compare(A, B, lt, gt, eq);
  parameter word_size=16;
  input [word_size-1:0] A, B;
  output lt, gt, eq;

  always @(*)
    begin
      gt=0; lt=0; eq=0;
      if (A>B) gt=1;
      else if (A<B) lt=1;
      else eq=1;
    end
endmodule
```

__/__/____

0x12: (Synchronous ^ clear) up-down counter.




```
module counter(mode, clr, ld, d_in, clk, count);
    input  mode, clr, ld, clk;
    input  [0:7] d_in;
    output reg [0:7] count;

    always @ (posedge clk)
        if (ld)          count <= d_in;
        else if (clr)    count <= 0;
        else if (mode)   count <= count + 1;
        else             count <= count - 1;
endmodule
```

__/__/____

parameter values are substituted before simulation or synthesis.

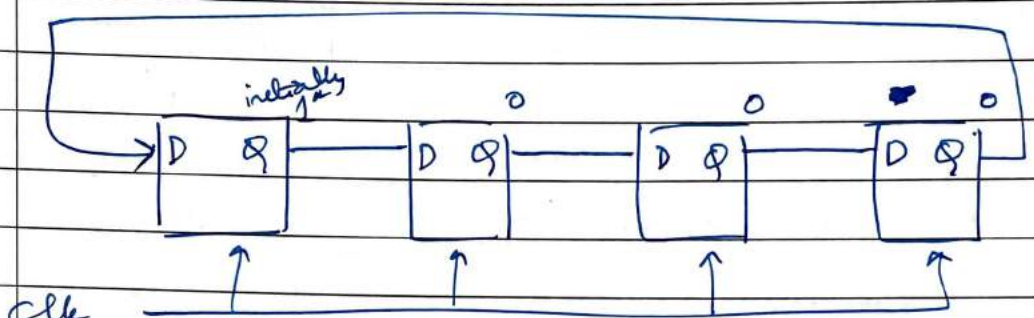
ex13: n bit counter - Parameterized design

```
module counter(clear, clk, count);
    parameter N=7;
    input clear, clk;
    output reg [0:N] count;

    always @(negedge clk)
        if(clear)
            count <=0;
        else
            count <= count+1;
endmodule
```

__/__/____

ex14: Ring counter

4 bit RC →



1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0
⋮

8 bit RC ↓

```
module ring_counter (clk, init, count);
    input clk, init;
    output reg [7:0] count;
    always @(posedge clk)
    begin
        if(init) count = 8'b10000000;
        else begin
            count <= count << 1;   // (if <= is used)
            count[0] <= count[7];
        end
    end
endmodule
```

initial 10000000
00000000.
Rotation not happening

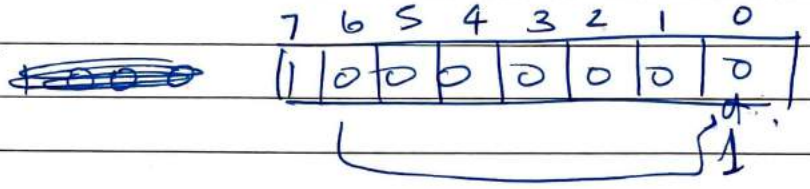
if(=' is used)

count <= count <<1

count[0] <= count[7]

1.

when non-blocking is used.



// RC (modified) → with blocking

```
module RC (clk, init, count);
    input clk, init;
    output reg [7:0] count;
    always @(posedge clk)
      begin
        if (init)   count = 8'b10000000;
        else
            count = {count[6:0], count[7]};
                          ↑ concat
      end
endmodule
```

a: 0101
b: 0011
EXOR: f: 0110

__/__/____

ex 15: Bitwise XOR

```
module xor_bitwise (f, a, b);
  parameter N=16;
  input [N-1:0] a, b;
  output [N-1:0] f;
  genvar p;

  generate for (p=0; p<N; p=p+1)
    begin xorlp // label for loop
      xor XG (f[p], a[p], b[p]);
    end
  endgenerate
endmodule
```

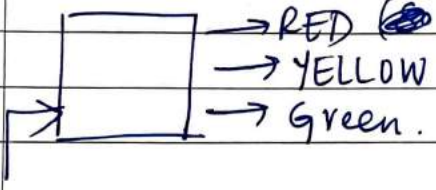
xorlp[0].XG
xorlp[1].XG

```
module generate_test;
reg [15:0] x, y;
wire [15:0] out;

xor_bitwise G(.f(out), .a(x), .b(y));

initial
  begin
    $monitor ("x: %b, y: %b, out: %b",
                x, y, out);
    x = 16'haaaa; y = 16'h00ff;
    #10 x = 16'h0f0f; y = 16'h3333;
    #20 $finish;
  end
endmodule
```

↑ in this ex, synthesis tool generates 5 FFs — 2 for state, 3 for light

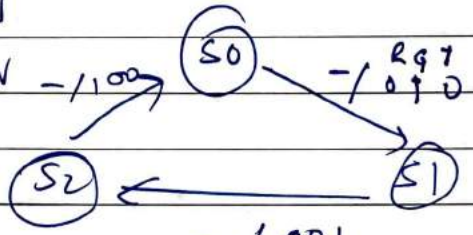
__/__/__

Ex16: Moore machine (Traffic Lights)



no I/P Lines -/(write)

00 S0 → RED
01 S1 → ~~YELLOW~~ GREEN
10 S2 → ~~GREEN~~ YELLOW



3 states

→ we need 2 bits to rep. the state.

```
module cyclic_lamp(clk, light);
    input clk
    output reg [0:2] light;
    parameter S0=0, S1=1, S2=2;
    parameter RED=3'b100, GREEN=3'b010,
      YELLOW=3'b001;
    reg [0:1] state;
    always @(posedge clk)
      case (state)
        S0: begin          // S0-RED
            light <= GREEN;
            state <= S1;
          end
```

__/__/____

```
S1:  begin                    // GREEN
        light <= YELLOW;
        state <= S2;
     end
S2:  begin                    // YELLOW
        light <= RED;
        state <= S0;
     end
default : begin
            light <= RED;
            state <= S0;
          end
endcase
endmodule

module test_cyclic_lamp;
   reg clk;
   wire [0:2] light;
   cyclic_lamp LAMP (clk, light);
   always #5 clk = ~clk;
   initial
     begin
       clk = 1'b0;
       #100 $finish;
     end
initial
begin
$dump
endmodule
```

__/__/____

Note

In the prev code, we do not need separate ffs for the outputs, as the outputs can be directly generated from state.

How to achieve this?

- modify code s.t. all assignments to light is made in a sep "always" block.
- use - blocking assignment triggered by state / change rather than clk.

will lead to synthesizer using comb. ckt.

```
always @ (posedge clk)
  case (state)
    S0: state <= S1;
    S1: state <= S2;
    S2: state <= S0;
    default: state <= S0;
  endcase
```

→ for this synthesis tool will generate 2 ffs

```
always @ (state)
  case (state)
    S0: light = RED;
    S1: light = GREEN;
    S2: light = YELLOW;
    default: light = RED;
  endcase
endmodule
```

→ comb ckt that takes state as I/P & produces light as O/P

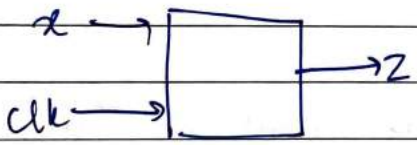
__/__/____

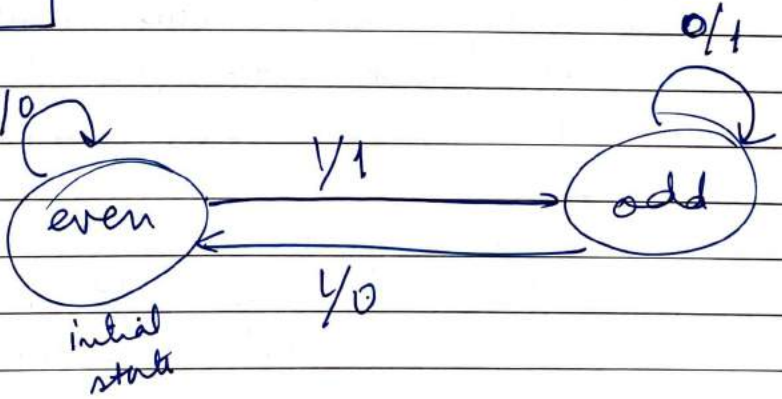
State ($S_1 S_0$)	Light (RGY)	
S_0 : 00	100	$R = S_0' S_1'$
S_1 : 01	010	$G = S_0$
S_2 : 10	001	$Y = S_1$
11	XX	

Ex 17: Design serial parity adder

- Continuous stream of bits is fed to a circuit in synchronism with a clock. The circuit will be generating a bit stream as output, where 0 → even no of 1's seen so far
1 → odd no of 1's seen so far.

- Moore machine





```
module parity-gen ( x, clk, z );
    input x, clk;
    output reg z;
    reg even-odd;    // machine state
    parameter EVEN=0, ODD=1;

    always @(posedge clk)
        case (even-odd)
            EVEN: begin
                z <= x ? 1 : 0;
                even-odd <= x ? ODD : EVEN;
                end
            ODD: begin
                z <= x ? 0 : 1;
                even-odd <= x ? EVEN : ODD;
                end
            default: even-odd <= EVEN;
        endcase
endmodule
```

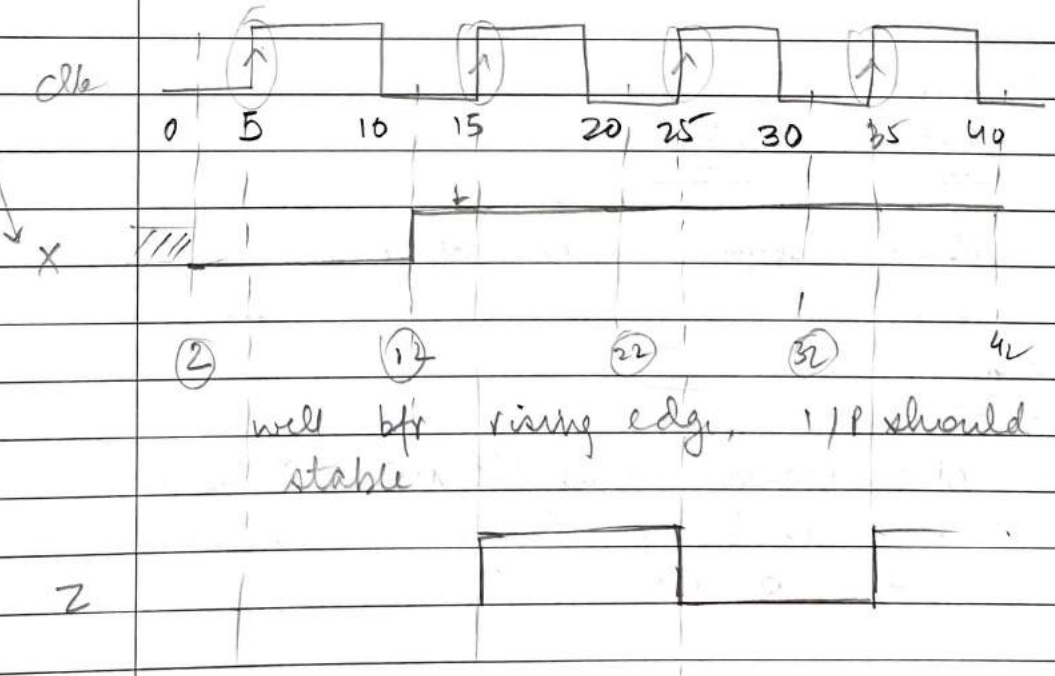
// design will cause synthesis tool to generate a latch for the o/p even-odd, z

(even → o/p=0
odd → o/p=1)

// again, just by knowing state we can tell o/p in this ex, we do not need extra latch to store o/p.

__/__/____

```
module test-parity;
  reg clk, x; wire z;
  parity_gen PAR (x, clk, z);
  initial
    begin
      clk = 1'b0;
    end.
  always #5 clk = ~clk;
  initial
    begin
      #2 x=0; #10 x=1; #10 x=1; #10 x=1;
      #10 x=0; #10 x=1; #10 x=1; #10 x=0;
      #10 x=0; #10 x=1 #10 x=1 #10 x=0;
      #10 $finish;
    end
endmodule
```



well b/f rising edge, I/P should be stable

// modified design → latch is not genrated for z.

```
always @(posedge clk)
  case (even_odd)
    EVEN: even_odd <= x? ODD : EVEN;
    ODD : even_odd <= x? EVEN : ODD;
    default : even_odd <= EVEN;
  endcase

always @(even_odd)
  case (even_odd)
    EVEN : z=0;
    ODD : z=1;
  endcase
endmodule
```

Ex18 Mealy machine.

(State + input dependent)

Sequence detector

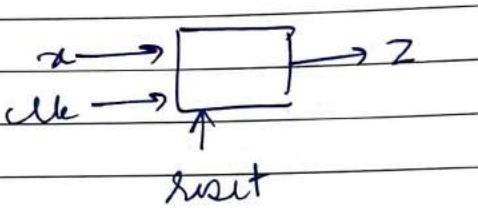
- ckt accepts serial bit stream "x" as i/p and produces serial bit stream 'z' as o/p.

- whenever bit pattern "0110" appears in input stream, it output z=1; at all other times z=0.

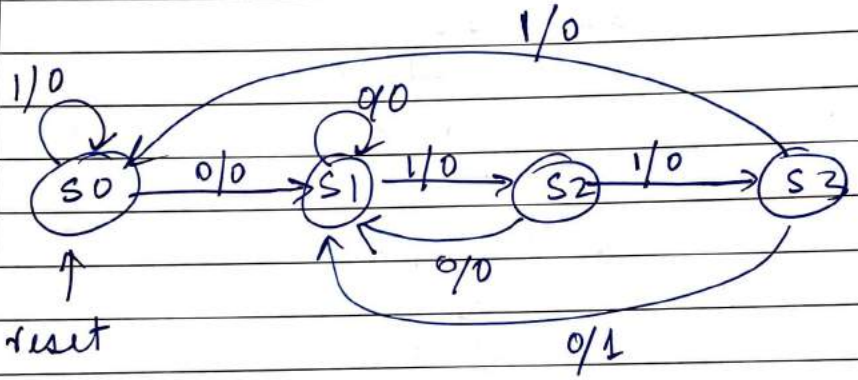
- Overlapping ✓ - Mealy machine

Ex x: 0 110 110

z: 0001001



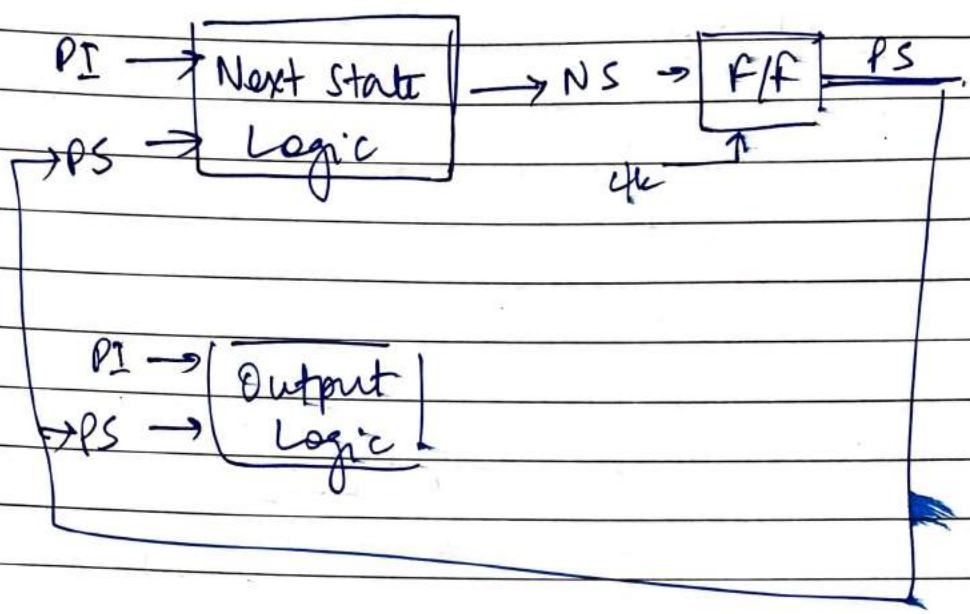
input bits "x" are applied in sync with clk.




```
module seq_detector (x, clk, reset, z);
  input x, clk, reset;
  output reg z;
  parameter S0=0, S1=1, S2=2, S3=3;
  reg [0:1] PS, NS;

always @(posedge clk or posedge reset)
    if (reset)   PS <= S0;
    else         PS <= NS;
```

...

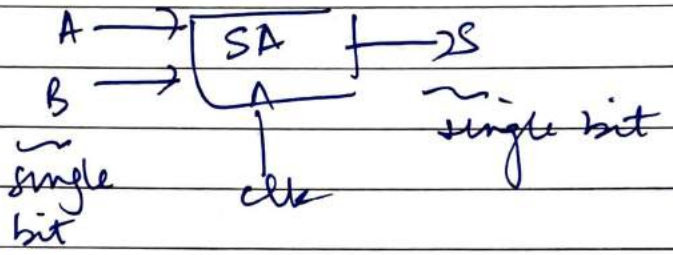
```
always @(PS, X)
    case(PS)
      S0: begin
            Z = X ? 0 : 0;
            NS = X ? S0 : S1;
          end

      S1 : begin
             Z = X ? 0 : 0;
             NS = X ? S2 : S1;
           end

      S2 : begin
             Z = X ? 0 : 0;
             NS = X ? S3 : S1;

      S3 : begin
             Z = X ? 0 : 1;
             NS = X ? S0 : S1;
           end
    endcase
endmodule
```



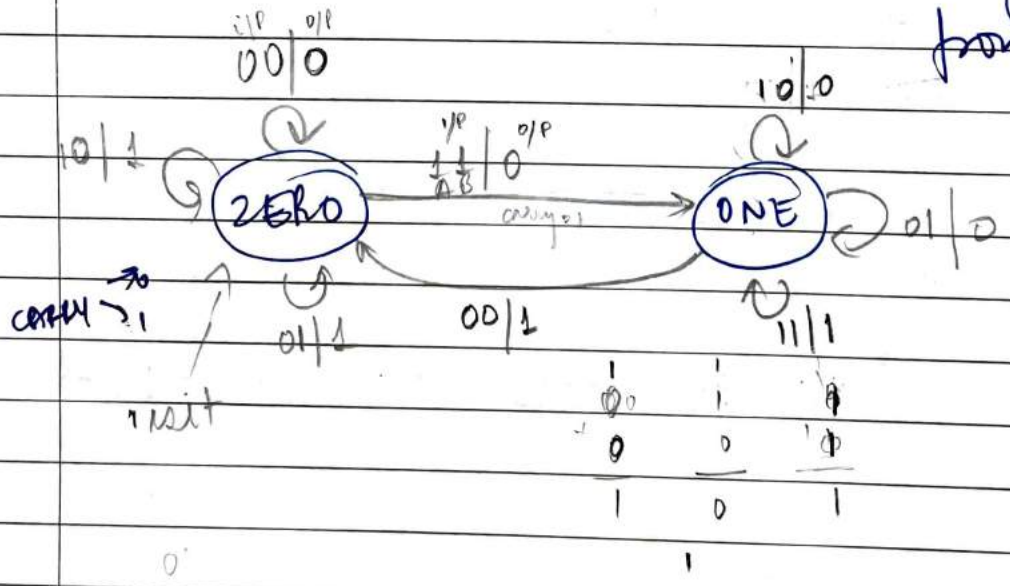

```
module test_seq;
  reg x, clk, reset;  wire z;
  seq_detector SEQ (x, clk, reset, z);

  initial
   begin
    clk = 1'b0; reset = 1'b1;
    #15 reset = 1'b0;
   end

  always #5 clk = ~clk;
   initial
    begin
    #12 x=0; #10 x=0; #10 x=1; #10 x=1;
          0       1       1       0
          0       1       1       0
    #10 $finish
    end
endmodule
```

Q19: Serial Adder

A: '0'110
B: 0010
S: 1000
0



state: CARRY (1 bit) → for every calc, we need to remember only carry from prev sum



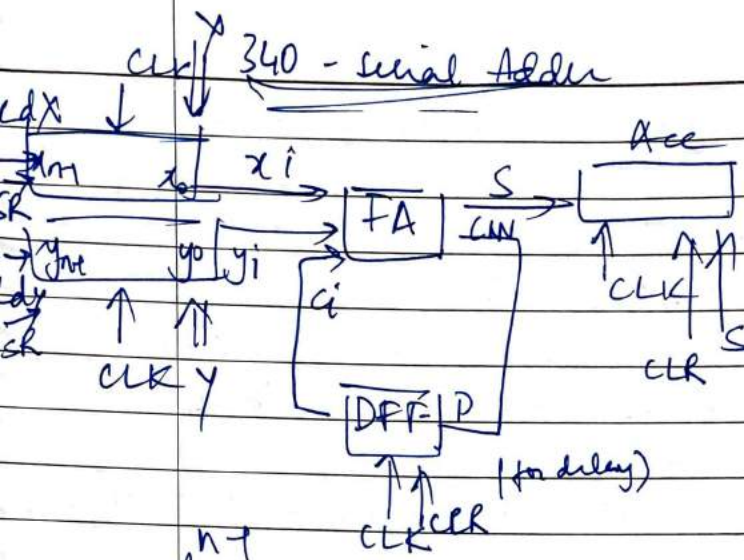
1
0 0 1
0 0 1
1 0 1
1

SR - Shift Right

RTL design

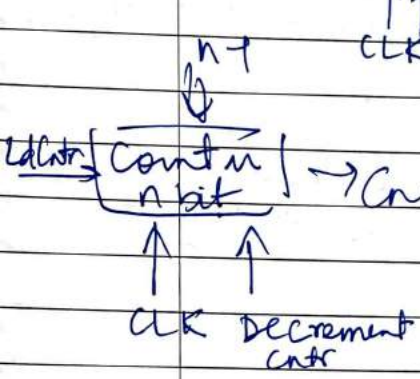
__/__/____

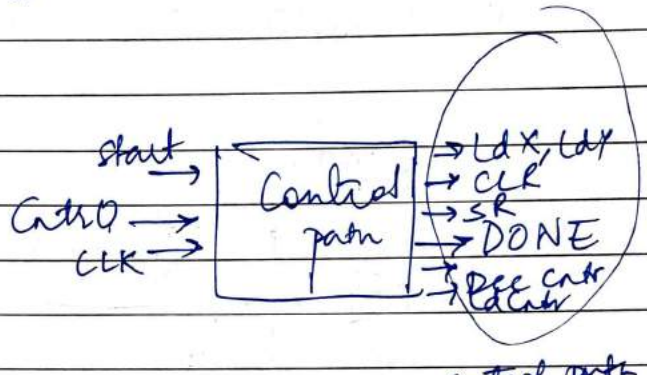
340 - Serial Adder



Two n bit nos.

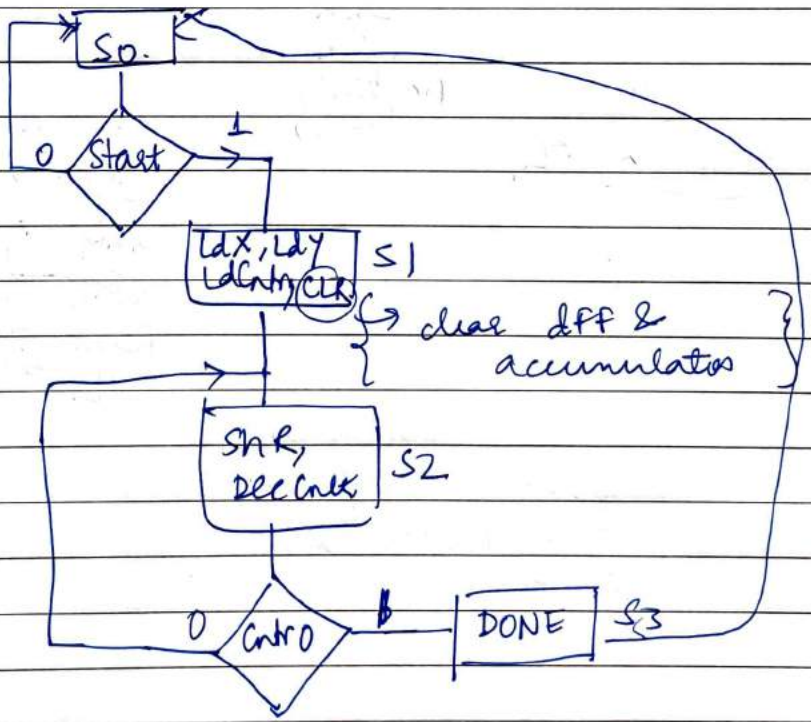
assumption → (not checking overflow, assuming result is also n bit)





control path to data path.

State diagram :



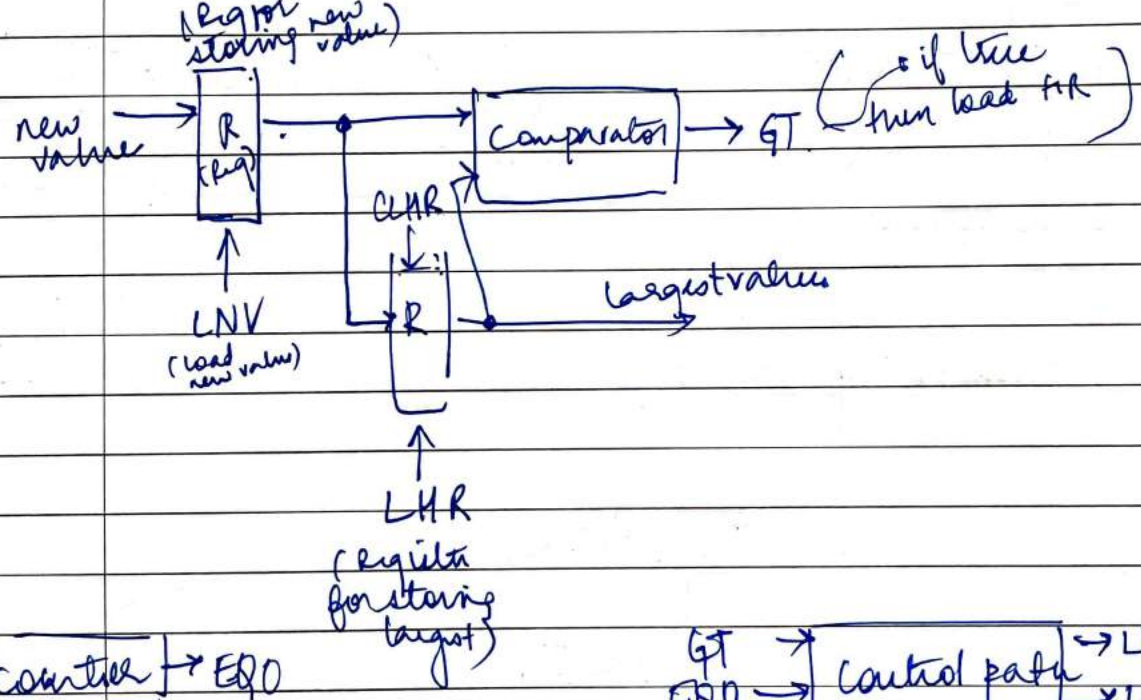
__/__/____

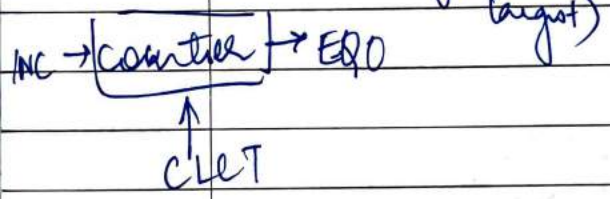
RTL ex ②

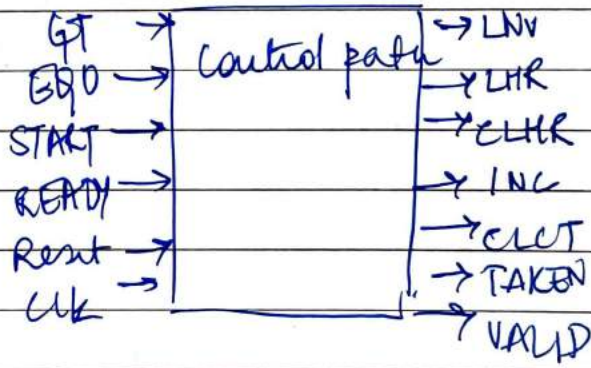
Design a digital system that reads in 256, 16 bit values one at a time and determines the largest one

External control signals

- I/P
  - START - start the operation
  - READY - indicates new value is available
  - TAKEN - indicates new value has been accepted
- O/P
  - VALID - indicates largest value o/p is valid

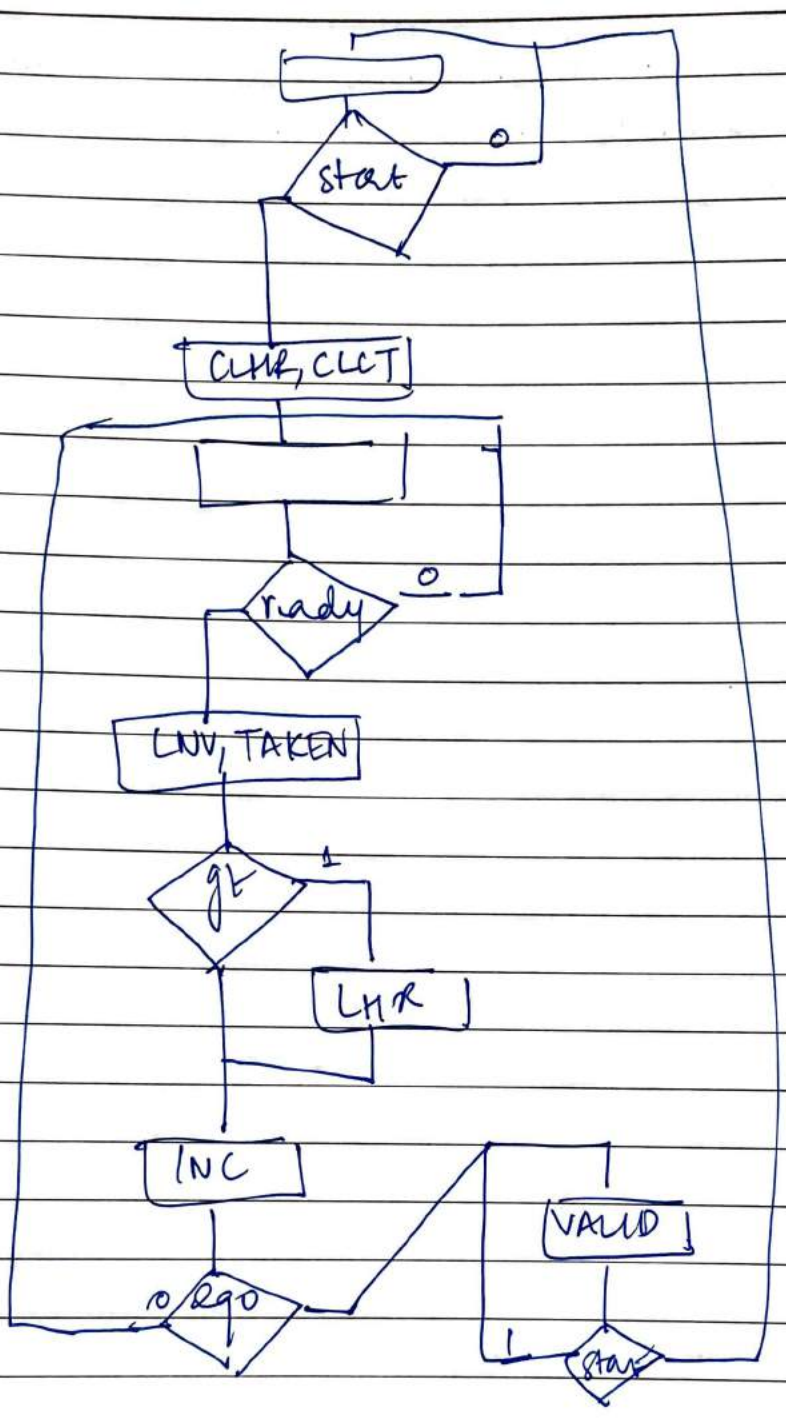






LNV - Load new value
CLHR - clear high reg
GT - greater than
CLCT - clear counter

__/__/____



ASM chart

P1: Sync process defining state transitions
P2: combinational process for NSD & OD
Structural model for components & interface b/w datapath & control path.

abstract level → state chart

__/__/____

State chart allows the operations of control and data path to be specified using a single notation