• Verilog

→ Hardware description language used to design the digital logics / for digital design.

→ Module endmodule → Basic building blocks.

→ Process of creating objects from module is instantiations and objects are instances.

→ For example → 4-TFF is instantiated to design a 4-bit counter.

→ Design block & Stimulus block → used to test the design
↓
ex:- Testbench → module

→ Verilog HDL is a case-sensitive language. All keywords are in lower case.

→ (\b), (\t), (\n) → whitespaces

→ whitespaces are ignored in verilog except when it separates tokens.

Yashna

→ whitespace is not ignored in string.

→ /* */ → multiple line comment

→ // → one line coment

→ -6'd3 // 8-bit negative number stored as 2's complement of 3

4'd-2 // Illegal specification.

→
- 23456 // 32-bit number (by default decimal)
- 'hC3 // " " hexadecimal
- 'b01 // " " " binary
- 6'd3 // 6 bit decimal number

→ Strings are treated as a sequence of one-byte ASCII values.

→ Identifiers are names given to objects.

ex:- reg value
↙ ↘ identifier
Keyword

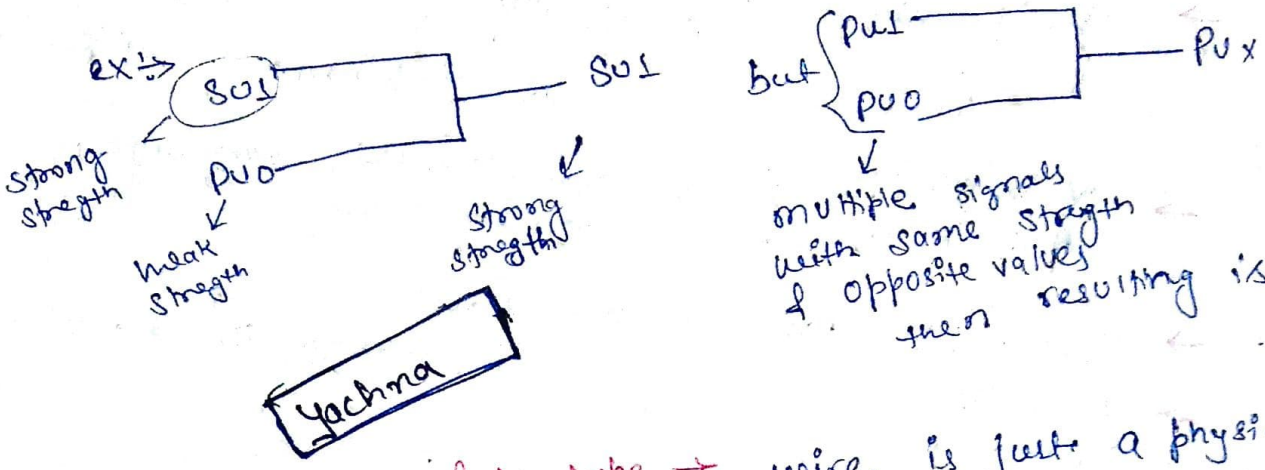
→ Identifiers can be start with alphabets or (_). but it cannot be start with $ or number.

→ Parameters & constant names must be given in upper case. ⇒ (eg. → DELAY, PI).

- Value Set ⇒

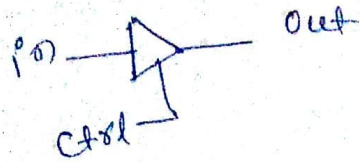
0	logic zero
1	logic one
X	unknown value
Z	High impedance, floating state

- Logic Strength values can be used to resolve signal contention one nets that have multiple drivers.

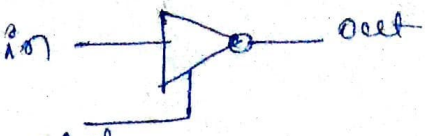


- Reg vs wire data type → wire is just a physical connection. Defaut value of net is 'Z'.
  wire
  Reg means register data type, It represents storage elements.
  Here Register ≠ edge triggered ff(register)
  Here reg means that can hold a value. A register doesnot need a driver like net. Reg → defaut value → X.
- Integer →
  → 32-bit data type
  → Signed type.
- Real → 3e6, 3.14 etc (keyword → real)
  ↳ If it is assigned to Integer data type then it is rounded off to nearest integer.
- time → 64-bit, used to store simulation time.
  → register data type.
- tri → nets have multiple drivers
- wire → nets with single driver

- Verilog data types ⇒
  - Nets
  - Registers
  - vectors → `wire [7:0] bus;` `reg [0:30] b;`
  - Integer
  - Real
  - Time
  - Arrays → `reg [4:0] port [7:0];` `integer count [0:7];`
  - parameters
  - Strings

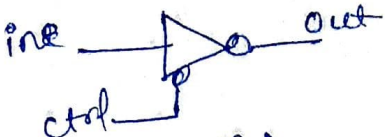


(bufif1)



(notif1)

(bufif0)



(notif0)

Yashma

- and #(5) a1 (out, i1, i2); // Delay of 5 for all transitions
- and #(4,6) a2 (out, i1, i2); // Rise = 4, Fall = 6
- bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off = 5

- Min/typ/max values are used to model devices ⇒ because delay varies within min & max range ⇒ bcz of process variations

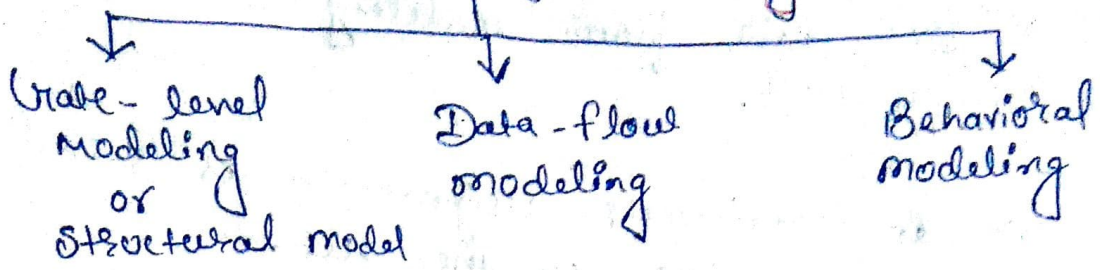
```
// one delay
and #(4:5:6) a1 (out, i1, i2); // if +mindelays, delay = 4
                               // if +maxdelays, delay = 6
                               // if +type delays, delay = 5

// two delay
and #(3:4:5, 5:6:7) a2 (out, i1, i2); // if +mindelays
                                      rise = 3  fall = 5  turnoff = min(3,5)

// three delay
and #(2:3:4, 3:4:5, 4:5:6) a3 (out, i1, i2);
                           // if +mindelays  rise = 2  fall = 3  turnoff = 4
```

- // Invoke simulation with maxm delay
  > vcs test.v +maxdelays

In Verilog, modeling style ⇒



• Gate-level Modeling ⇒

→ Primitives (Predefined modules like AND, OR, NOT etc. gates in library) are used to design any digital logic circuits.

→ So to model gate-level, designer should aware of structure of any digital circuit so that circuit can be modeled in terms of logic gates, mixes etc.

→ For bigger circuit it is very difficult.

• Data-flow modeling ⇒

→ For this modeling designer is aware of how data or signal flows between hardware and how the data is processed in the design.

→ In this modeling, circuits are designed in terms of its function.

→ Boolean equations & operators are used in this modeling.

→ By using this modeling, we can optimize the code.

→ For data flow modeling, continuous assignment is used.

→ continuous assignment is used to assign the value to the net.

→ Syntax:-

assign <net expression> = [delay] <expression of different signal.

ex:-

assign a = c & b;

• Behavioral modeling :-

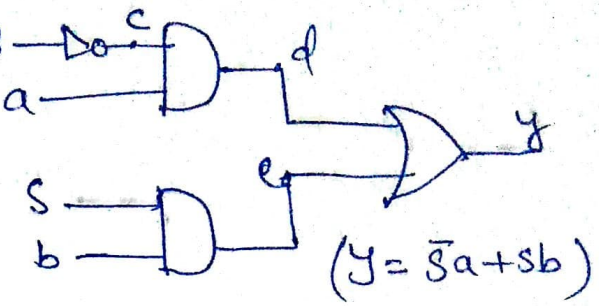
→ This modeling defines the behavior of digital circuit.

→ It is similar to writing the algorithm of any design as in C-language.

→ It doesn't give actual information of internal hardware of design.

• 2x1 Mux using gate-level Modeling :->

```
module mux_2to1(s,a,b,y);
  input s,a,b;
  output y;
  wire c,d,e;
  not (c,s);
  and a1(d,c,a);
  and a2(e,s,b);
   or (y,d,e);
endmodule
```



$(Y = \bar{S}a + Sb)$

• 2x1 mux using data-flow modeling :->

```
module mux_2to1 (a,b,s,y);
  input a,b,s;
  output y;
  wire y;
assign y = s ? b : a;
endmodule
```

• 2x1 mux using Behavioral modeling :->

```
module mux_2to1 (a,b,s,y);
  input a,b,s;
  output reg y;
always @ (a or b or s)
begin if (s)
   y = b;
  else
   y = a;
end
endmodule
```

- Continuous assignment → used in data flow modelling.
  → Always active
  → "assign" keyword is used.
  → Expression is evaluated as soon as one of the right-hand side variable changes.
  → LHS must be Scalar/vector net or concatenation of Scalar or vector net.
  → LHS cannot be scalar/vector reg.
  → RHS can be Scalar/vector net or reg or function calls.
- Implicit continuous assignment → wire out = i1 & i2;

Yachna

- Delay :-

```
wire #10 out;
assign out = i1 & i2;
```

```
wire out;
assign #10 out = i1 & i2;
```

both have same effect.

- Verilog is concurrent in nature while C-language is sequential.
- Activity flows in verilog seen in parallel rather than in sequence.
- Each always & initial block are separate activity flow in verilog. Each activity flow starts in Simulation time '0'.

• Procedural blocks

Always

- each always block executes concurrently.
- executes in loop.
- Synthesizable

Initial

- each initial block executes concurrently.
- They execute only once.
- Non-Synthesizable

```
module clock_gen;
  reg clock;
initial
  clock = 1'b0;
always
  #10 clock = ~clock;
initial
  #1000 $finish;
endmodule
```

⇒ If clock initialisation is put inside always block, then clock will be initialized every time the always is entered.

⇒ Simulation must be halted inside separate initial statement.

If no $stop or $finish then it will run forever.

Yachna

• Procedural assignments ⇒

- It updates the values of real, reg, integer, time variables
- → Procedural assignment is always modelled inside procedural block.

• Difference in Procedural assignment & continuous assignment ⇒

Procedural assignment	continuous assignment
The values placed on left hand side variable will remain unchanged until another procedural assignments updates the variable with different value.	continuous assignment cause the value of RHS expression to be continuously placed on to the left-hand side net.

• Procedural assignment

Blocking (=) (BA)

- Blocking Assignments are executed in the order as they are specified in a procedural block.
- RHS of statement is evaluated and after assigning to its LHS it goes to execute the next statement.

Non-blocking (<=) (NBA)

- NBA doesn't block the execution of next statement in that procedural block.
- All RHS are evaluated and then assigned to LHS simultaneously.

- It is recommended that BA & NBA are not mixed for same always/procedural block. (Simulator may allow, but this is not good in design practice).

- BA & NBA can be used to model both Sequential as well as combinational logic.
  Best for good design practice
  or Sometimes to avoid error in circuit, it is good to use BA for combinational circuit & NBA for sequential logic (specially for synchronous sequential logic).

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

  (x = x+5;
    x<=. y;) X → wrong

Yaehna

- Timing controls

  - Delay based
    - #10 y=1 ; // delay the execution of y=1 by 10 units.
      (Inter assignment delay)
    - y = #5 x+z ; // Take the value of x & z at time=0 & evaluate x+z and wait 5 time units to assign the value to y.
      (Intra assignment delay)
  - Event based
    - @(clock) q=d;
      // q=d executed when clock signal changes the value
    - q = @(posedge clk)d;
  - Level-sensitive
    - wait (count_enable)
      #20 count = count+1;
      'wait' keyword is used for level sensitive Timing control.
      If count_enable=0 then it will not enter in the statement.

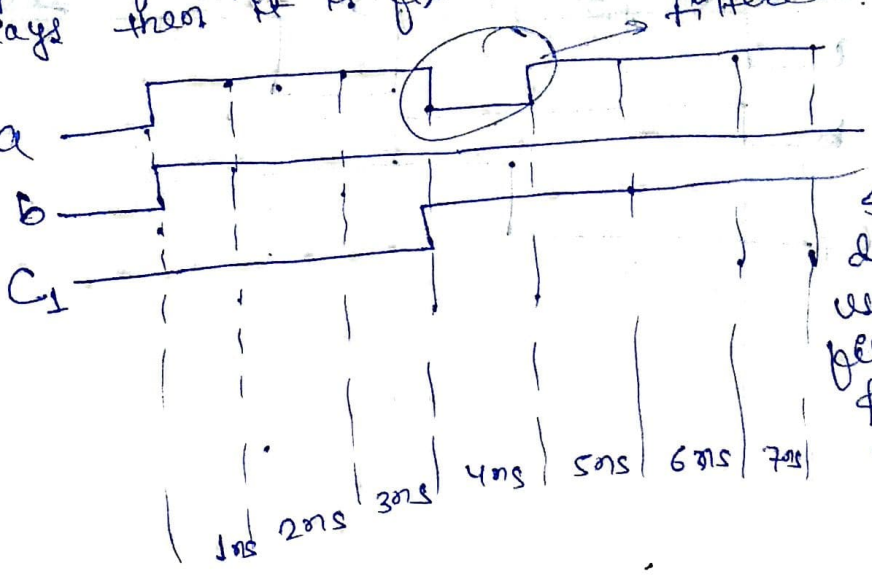
• Inertial delay → Inter assignment delay
→ Gate in-built delay in combinational circuit

⇒

```
always @ (a or b)
  #3  c1 = a+b;
```

// If a or b changes, always block is activated. Then after 3ns (a & b) is evaluated and assigned to c. Then even (a or b) will be checked for change again. If a or b changes in interim period (i.e., within 3ns), then block is not activated. So module doesn't give correct output. If input pulses are shorter than specified delays then it is filtered out.

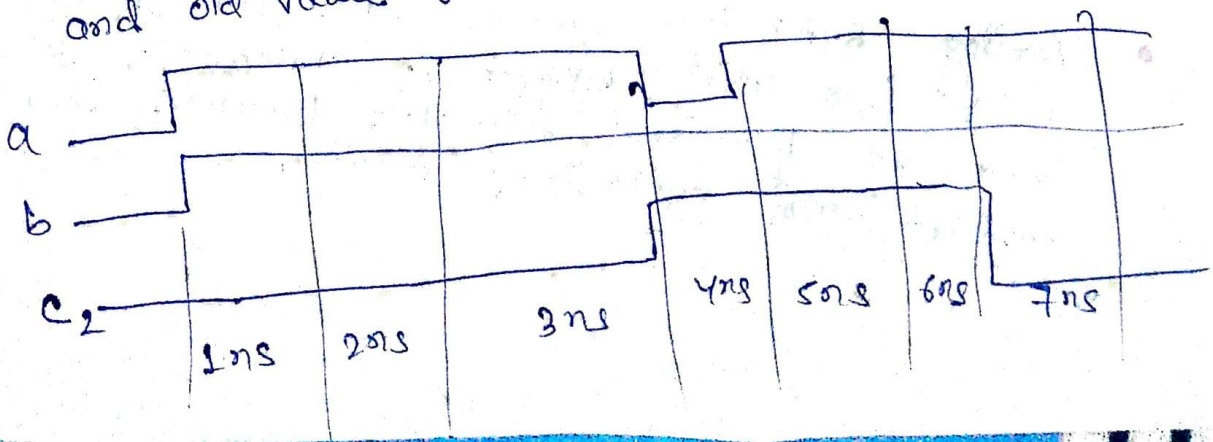
Ex:⇒



Hence, Inertial delay is used to filter spikes & transients on signal.

Yachna

• C_2 = #3 (a+b)
// Inertial assignment delay
// Here (a&b) is evaluated immediately and stored in simulator queue but assigned to C_2 after 3ns.
// only after delay assigned to C_2, event (a or b) checked for change. If 'a' or 'b' changes in 3ns, the new value doesn't effect the output and old value remains.

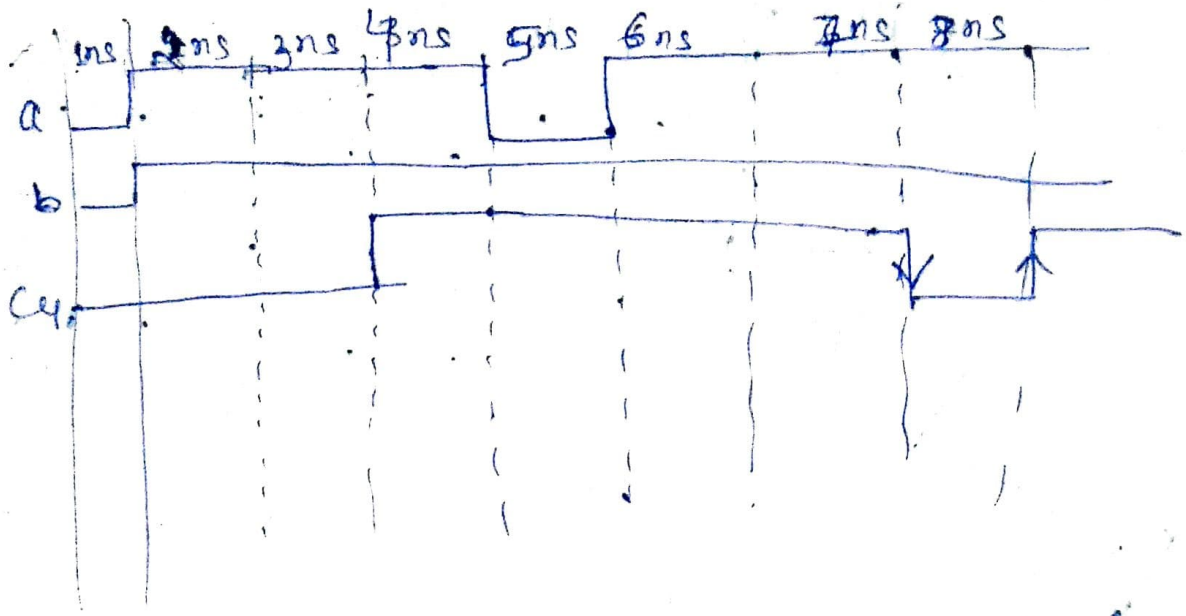


- #3 c_3 <= a+b // same operation as for c_1

- c_4 <= #3 (a+b); ⇒ Transport delay

// Here a + b are evaluated and assigned to c_4 after 3ns but within 3ns it is checked that a or b changes or not, if changes, then new output value is updated.

- Transport delay ⇒ Pure propagation delay.



ex:⇒ #200 a = #100 b; // After 200ns 'b' is evaluated and is assigned to 'a' after 100ns.
So, at 300ns 'a' will get 'b' value.

Yachna

- Verilog synthesizer ignores the delays specified in a procedural assignment statement. It may lead to functional mismatch between design modeled and synthesized netlist.

• Swapping values of two variables 'a' & 'b' :→

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

⟹ Race condition between two ~~always~~ BA block.

At time 't = 0' both always block will execute concurrently but it depends on simulator that either a=b will execute first or b=a will execute first.

So, output may be 'a' or 'b'.

yachna

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

→ correctly swapped

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

⇒ 'b' gets 'a' but at the end 'a' will not get 'b' so swapping is not done.

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

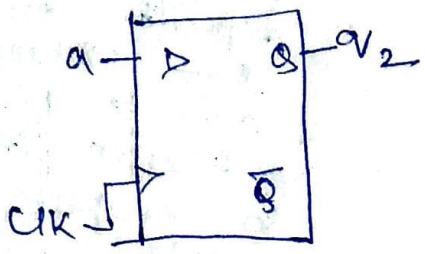
→ correctly swapped

So, for swapping through BA, one extra register is required.

Q.) what will be output circuit?

```
always @ (posedge clk)
  begin
    v1 = a;
    v2 = v1
  end
```

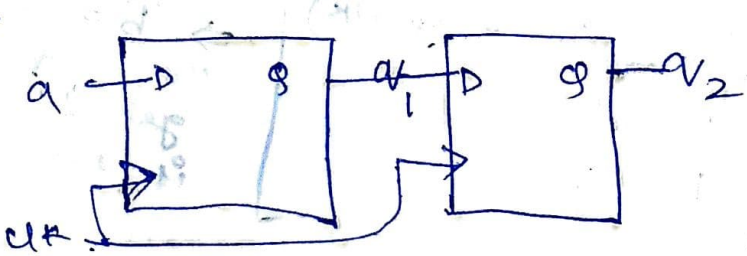
Ans. →



Q.) what will be output circuit?

```
always @ (posedge clk)
  begin
    v2 = v1;
    v1 = a;
  end
```

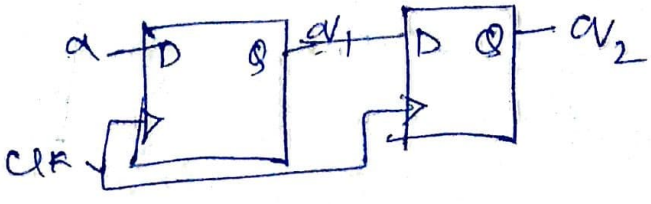
Ans. →



Q.) what will be output circuit?

```
  begin
    v1 <= a;
    v2 <= v1;
  end
```

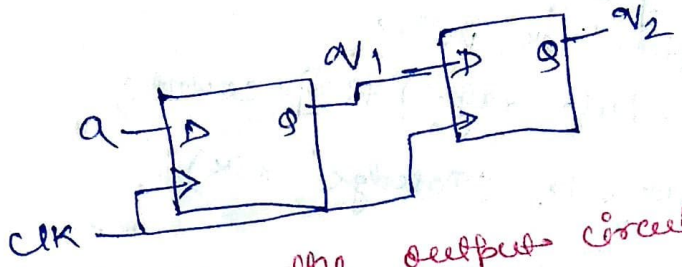
Ans. →



(Q.) what will be the output circuit?

```
always @ (posedge clk)
    v2 <= v1;
always @ (posedge clk)
    v1 <= a;
```

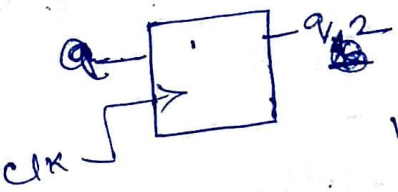
Ans. →



(Q.) what will be the output circuit?

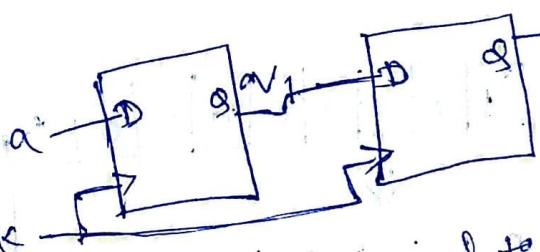
```
always @ (posedge clk)
    v1 = a;
always @ (posedge clk)
    v2 = v1;
```

Ans. →



⇒ if 1st always block execute
↳ v_2 will get 'a'

yachma

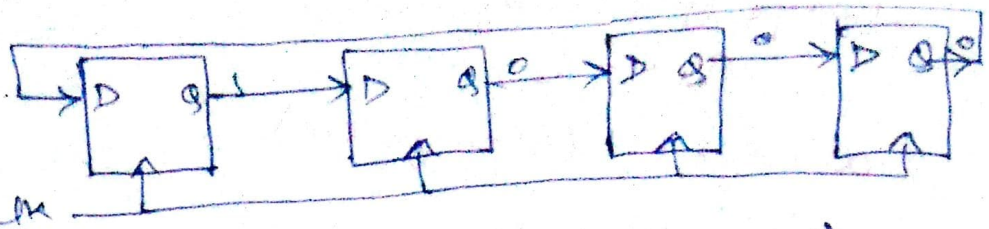


⇒ If 2nd always block will execute first.

So, v_1 is assigned to v_2 then a is assigned to v_1. so, v_2 will get v_1.

So, ambigious or cannot be said that what will be the value of 'v_2'.

• Ring Counter :->

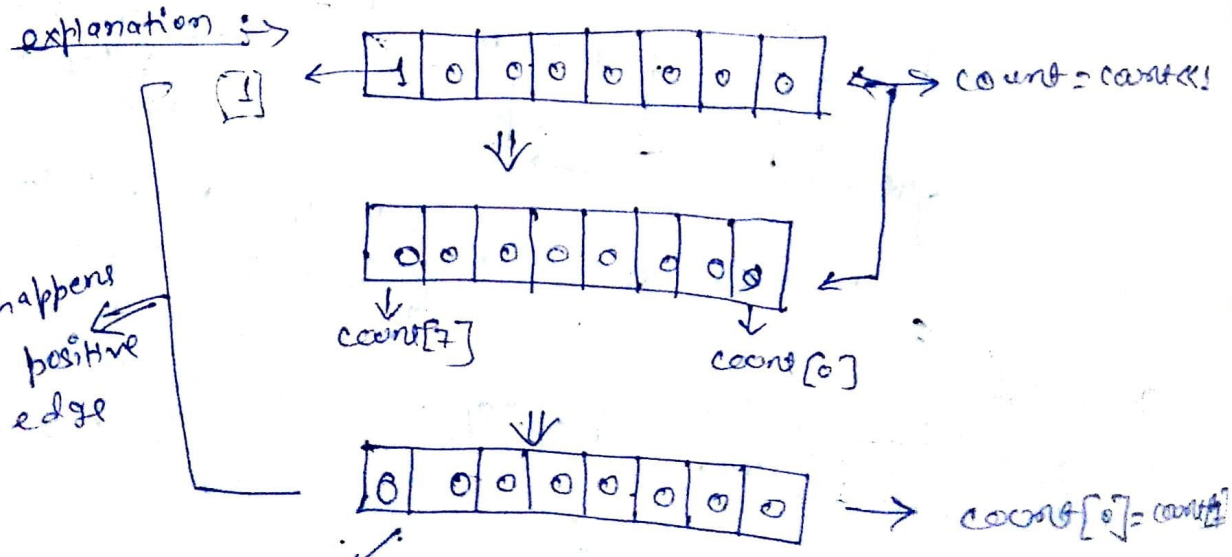



```
module ring_counter (clk, init, count);
input clk, init;
output reg [7:0] count;
always @ (posedge clk)
  begin
    if (init)
    count = 8'b10000000;
  else
  begin
    count = count << 1;
    count[0] = count[7];
  end
end
endmodule
```

⇒ do not model a ring counter.

yochna

explanation :->



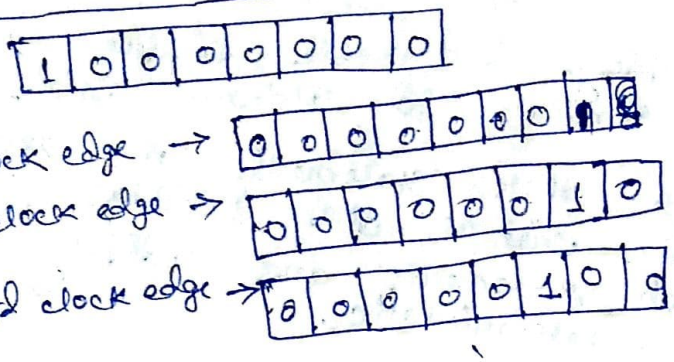
After getting this again same above operation at next positive clock-edge.

since, only '0' is there, no '1' is stored so, this doesn't form a ring counter.

```
module ring_counter (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]};
  end
endmodule
```

Vachna

∴



} This works as ring counter

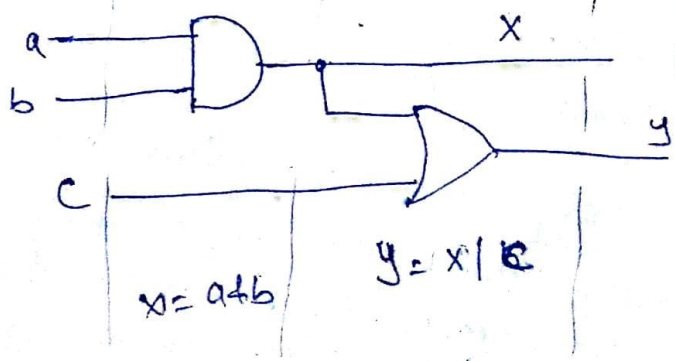
(or)

```
if (init)
  count = 8'b10000000;
else begin
  count <= count << 1;
  count[0] <= count[7];
```

} → This also works as ring counter

- Ordered execution mimics the inherent logic flow of combinational logic.
  Hence blocking assignments generally work better for combinational logic.

ex:-



• System tasks

- $ <keyword>
- $display, $monitor, $strobe, $time, $stime, $realtime, $reset, $write, $stop, $finish, $random, $dumpfile, etc.

• $display → $display ("Hello you can learn");
o/p → Hello you can learn
$ display ($time)
o/p → 150

& $strobe

→ Both $display & $strobe displays whatever written inside it. Similar to Printf in 'c'.

→ If many other statements are executed in same time unit as the $display task, the order in which statements and $display are executed is non-deterministic.

Yachna

→ If many other statements and $strobe are executed in same time unit, then $strobe will execute after execution of other statments in same time unit.

→ So, $ strobe provides synchronisation mechanism to ensure that data is displayed only after other statements.

$monitor → used to monitor a signal when its value changes.

→ It has the functionality of event-driven print.

→ Executes only once.

→ Only one $ monitor is activated at a time.

→ If there are more than one $monitor statemeat, then last $ monitor statement will be active statement.

$monitoron → monitoring on

$monitoroff → monitoring off

$time → Returns current simulation time as a 64-bit integer.

$stime → Returns current simulation time as a 32-bit integer.

$realtime → Returns current simulation time as a real number.

$write → Print a set of values on a single line.

$reset → Resets the simulation back to time 0.

$stop → Halts the simulator and puts it in interactive mode where the user can enter commands

$finish → Terminate or stop the simulation.

Yachna

Ex:-

```
initial begin
clock = 0;
reset = 1;
#200 $stop;
#600 $finish;
```

→ suspends the simulation at t=200.

→ Terminates the simulation at t=800.

$random → Generates 32-bit random integer number. It can be +ve or -ve.

$dumpfile ("file name.vcd") → To dump the file
↓
means value change dump

$dumpvar → To dump the variables in the design.

$dumpvar (level , module name) :->
↓
If level=0, then all variables of that module will be dumped.

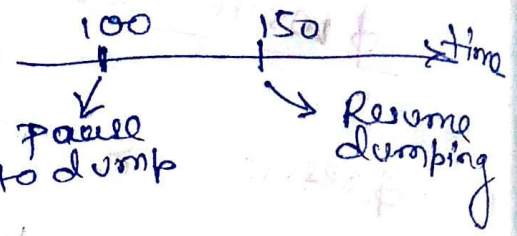
ex:-> $dumpvars (0, module1 , module2);
↓
all variables of these modules are dumped.

If level = 1, the selected variables are dumped. ex:-> $dumpvars (1, a, b, c, d);

$dumpall → All variables of that file are dumped.

$dumplimit (file size); → To limit the maximum size of .vcd file.

$dumpoff → stop dumping $dumpon → Initiate dumping.

Ex:-
```
initial begin
  #100 $dumpoff
  #150 $dumpon
```


Yachna

$readmemb (file name, memname, Start-addr, stopaddr)

$readmemh (" " " " ")

↓ To read in binary format

↓ To read in hexadecimal format

→ If start addr & stop addr are omitted, then entire memory is read.

ex:- $readmemh ("mem.dat", mem);

↓ memory will start to read the data from mem.dat file.

$readmemb ("mem.dat", mem, 200, 50);

↓ i.e., reading from location 200 to 50.

• Grouping the statements

using
begin
- - -
end

↓

• sequential blocks

[i.e., all statement will execute one after another except for blocking assignments.]

• synthesizable

using
fork
- - -
join

• parallel blocks
↓
execution will be in concurrent

• Non-synthesizable

• It is not used to model combinational circuit.

• **Nested blocks** →

```
initial begin
  #10  y = z;
  fork
    #5  z = 15;
    #10  x = z;
  join
end
```

yachna

• Compiler directives ⇒ All directives are defined by `<keyword>`.

ex ⇒ • define , • include , • time scale.

• define → • Similar to #define in c-language.
• It is used to define text-macros or constants.

ex ⇒ • `define WORD_SIZE 32`
• `define S $stop`

• include → • It is used to include entire contents of verilog source file in another verilog file during compilation.

• timescale → • For Accuracy & Precision.

ex ⇒ • `timescale 10ns/100ns`
- - -
1.5

1.5 × 10 = 15 = 20ns

→ 10ns
→ 20ns
→ 30ns
→ 40ns

• Disable → • To get out of loop.
• Similar to break statement in C.

difference =>

disable	break
disable any named block in the design.	breaks the current loop only.

Ex=>

```
initial
  begin
  flag = 16'b 1000-0000-----------;
    i=0;
  begin : block1
  while (i<16)
    begin
  if (flag[i])
  begin
  $display ("Encountered a True bit at
            element number %0d", i);
  disable block1;
  end
  i=i+1;
end
end
end
```

yachna

disable block bez you found true bit.

• Verilog operators :=>

Operator	Symbol	Number of Operands
Arithmetic	+ (Add)	Two
	− (Sub)	"
	* (Mul)	"
	% (modulus)	"
	/ (division)	"
Logical	! (logical negation)	One
	&& (logical and)	Two
	|| (logical or)	"

Relational	> < >= <=	Two " " "
Equality	== (equality) != (inequality) === (case equality) !== (case inequality)	two " " "
Bitwise	~ (bitwise negation) & (bitwise and) | (bitwise or) ^ (bitwise xor) ^~ or ~^ (xnor)	one " " " "
Reduction	& (Reduction and) ~& (Reduction nand) | (" or) ^ (" xor) ^~ or ~^ (" xnor)	one " " " "
shift	logical left shift logical right shift Arithmetic left shift Arithmetic right shift	Two " " "
concatenation	{ }	any no.
Replication	{{ }}	any no.
conditional	? :	three

compares the both x&z also

ex:- X = 4'b1001
&X // 1&0&0&1 = 0

NOTE :- in1 = 4'b10x1; // then sum = in1 + in2 will be 4'bx
in2 = 4'b0010;

- operators should be inside paranthesis, if it is not in paranthesis then it will consider precedence of operators.

Unary → + − ! ~

Multiply → * / %

→ + −

→ << >>

→ < <= > >=

→ == != === !==

Reduction → & ~&

^ ~^

| ~|

Logical → && ||

conditional → ? :

→ Highest precedence

High

Low

→ lowest precedence

Yachna

<Q> Difference in Arithmetic & logical shift operation?

- Arithmetic right shift :→ MSB is copied.

1011 → 1101

1101 → 1110 → (-2) → 2's complement

2's complement (-2)

→ Arithmetic right shift preserves the sign.

- Logical right shift :→

1011 → 0101 → 0010

2's compliment → (-5) | 2's compliment → +5 | (+2)

So, sign changes in logical shift operations.

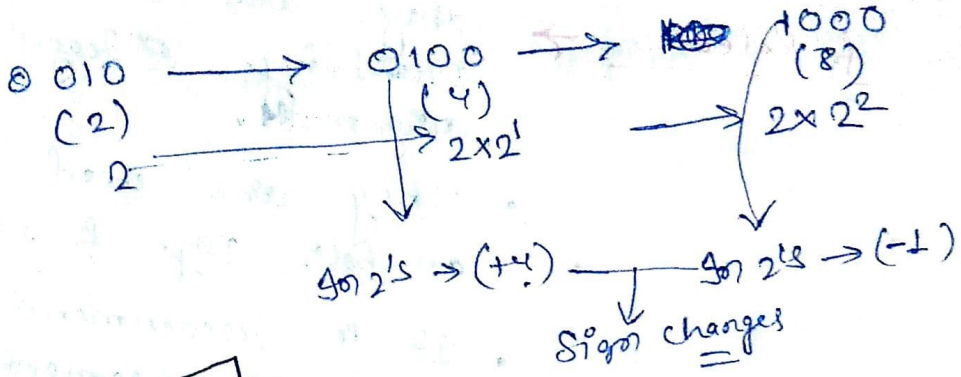
- For positive number logical right shift divide the number by 2^n (n = no. of shifts) & throw out the remainder.

• Logi • Arithmetic left shift =>

1101 ⟶ 1010

Sm 2's → (-3) Sm 2's (-6)

• Logical left shift => multiply the number by 2^n where n = no. of shifts.



Yashna

• Procedural continuous assignment =>

- ~~Procedur~~ PCA are normally used for controlled periods of time
- keywords assign & deassign are used.
- LHS of ~~pro~~ PCA can only be a reg type or concatenation of reg.
- It cannot be part or bit select of net or array of reg.
- PCA override the effect of regular Procedural assignments.

Ex:

```
module edge_dff (q, qbar, d, clk, reset);
  output q, qbar;
  input d, clk, reset;
  reg q, qbar;
always @ (negedge clk)
  begin
    q = d;
    qbar = ~d;
  end
always @ (reset)
  if (reset)
begin
    assign q = 1'b0;   // override
    assign qbar = 1'b1;
end
```

```
else
  begin
    deassign q;          // if reset goes low
    deassign qbar;       deassign or remove the
  end                    overriding values.
endmodule
```

- **Force & Release** → they are also used for overriding regular procedural assignments.
  - they are used to override on both reg & net data types.
  - It is recommended, force & release statements not to be used inside design block.
    they should be used inside stimulus block.

yachna

```
module stimulus;
  edge dff dff (Q, Qbar, D, clk, RESET);
  initial begin
  #50 force dff.q = 1'b1; // at t=50, force
   #50 release dff.q;     // at t=100, release
  end
endmodule
```

Overriding Parameter ⇒

- **defparam** ⇒ Using keyword defparam parameter values can be changed separately in module instances.

ex⇒
```
module hello_word;
  parameter id_num=0;
  initial
    $display("Displaying hello_word id number=%d", id_num);
  endmodule
```