UNIT - II

Behavioral Description:

Highlights:

=) The data-flow simulations were implemented to describe digital systems for known digital structures like adders, multiplexers and latches.

⇒ The behavioral description is a powerful tool to describe systems for which the digital logic structures are not known or are hard to generate.

Ex:- Complex arithmetic units
Computer control units
biological mechanisms - to describe the physiological action of certain organs, such as the kidney or heart.

* It describes the system by showing how the outputs behave according to changes in the inputs.

* For this description, we do not need to know the logic diagram of the system; we must know how the output behaves in response to change in the input.

* In VHDL → behavioral description statement is "process"

In verilog → "always" and "initial"

* For VHDL, the statements inside the process are sequential. Verilog → statements are concurrent.

Structure of the HDL behavioral description:

⇒ The predefined word "process" is used in VHDL.
⇒ Every VHDL behavioral description has to include a process.

VHDL:

```
entity half_adder is
port (I1, I2 : in bit ; O1, O2 : out bit);
end half_adder;
architecture behave_ex of half_adder is
begin
process (I1, I2)
begin
  O1 <= I1 xor I2 after 10ns;
  O2 <= I1 and I2 after 10ns;
end process;
end behave_ex;
```

Verilog:

```
module half_add (I1, I2, O1, O2);
input I1, I2;
output O1, O2;
reg O1, O2;
always @ (I1, I2)
 begin
   #10 O1 = I1 ^ I2;
   #10 O2 = I1 & I2;
 end
end module
```

Process - VHDL:

=) The statement process (S1, S2) is concurrent statement.

* The process statement is executed (activated) only on event occurs on any element of the sensitivity list. Otherwise the process remains inactive.

* If the process has no sensitivity list, then the process is executed continuously.

=) All statements inside the body of the process are executed sequentially.

=) The sequential execution means sequential calculation.

* The calculation of a statement will not wait until the preceding statement is assigned - only until the calculation is done.

Verilog :- (with always)

=) In contrast to VHDL, all verilog statements inside always are treated as concurrent.

=) Any signal that is declared as an O/p should also be declared as "register" (reg) if it appears inside always.

VHDL Variable - Assignment statement:

=) The use of "variables" inside process is common practice in VHDL behavioral descriptions.

=) Consider the two assignment statements inside a process.

```
sig : process (t1)
begin
st1 : S1 <= t1;
st2 : S2 <= not S1;
end process;
```

label:

⇒ sig, st1 & st2 are the lables.

⇒ These lables ~~are~~ does not used for compilation or simulation.

⇒ They are used to refer a certain statement.

⇒ In above example 'S1' is appears in both statements left-hand side in S1 & right-hand side in S2.

* At simulation time T0,

$t_1 = 0$ and $S1 = 0$

* At simulation time T1,

t_1 changes from 0 to 1. (event)

⇒ Process is activated now, and S1 is calculated as '1'. But, S1 does not acquire this new value of '1' at time T1, but rather at $T_1 + \Delta$.

⇒ In st2, S2 at T1 is calculated using the old value of S1 (0).

⇒ Now, if we use variable assignment statements as follows:

```
vaib : process (t1)
Variable temp1, temp2 : bit ;  -- Variable declaration
begin
St 1 : temp1 := t1 ;  -- Variable assignment.
St 2 : temp2 := not temp1 ;
St 3 : S1 <= temp1 ;
St 4 : S2 <= temp2 ;
end process ;
```

⇒ Variable assignment statements are calculated and assigned immediately with no delay time b/w calculation and assignment.

⇒ Assignment operator is ":="

⇒ If t1 acquires a new value of 1 at T_1, then immediately temp1 = 1 and temp2 = 0.

⇒ For statement 3 & 4, S1 acquires temp1 (= 1) at $T_1 + \Delta$ and S2 acquires temp2 (0) at $T_1 + \Delta$.

Sequential statements: (If, case & loop)

⇒ These are several statements associated with behavioral descriptions.

⇒ These statements have to appear inside "process" in VHDL or inside always or initial in verilog.

(1) If Statement:

⇒ IF is a sequential statement, that appears inside "process" or "always & initial".
⇒ It has several formats.

(i) VHDL IF Formats:

```
if (Boolean Expression) then
Statement 1;
Statement 2;
Statement 3;
.....
else
Statement a;
Statement b;
Statement c;
.....
end if;
```

Verilog IF Formats

```
if (Boolean Expression)
begin
  Statement 1;
  statement 2;
end
else
begin
  Statement a;
  Statement b;
  Statement c;
  .....
end
```

if only one statement then, begin and end can omitted

Example:

VHDL

```
if (clk = '1') then
temp := S1;
else
temp := S2;
end if;
```

Verilog:

```
if (clk == 1)
temp = S1;
else
temp = S2;
end
```

(ii) 'If' as a Latch

=> The else statement can be eliminated, and in this case the 'if' statement simulates a latch.

Ex:-

VHDL

```
if clk = '1' then
  temp := S1;
end if;
```

Verilog:

```
if (clk == 1)
begin
  temp = S1;
end
```

* If clk is high then the value S1 is assigned to temp. If clk is not high, temp retains its current value as latch.

(iii) If as Else-If:

VHDL

```
if (Boolean Exp 1) then
  Statement 1 ; Statement 2 ;
elsif (Boolean Exp 2) then
  Statement a ; statement b ;
else
  Statement x ; Statement y ;
end if;
```

Verilog:

```
if (Boolean exp 1) --
begin
  Statement 1; statement 2;
end
else (Boolean exp 2)
begin
  Statement a; statement b;
end
else
begin
  statement x; statement y;
end
```

Ex:-

```
if signal1 = '1' then
temp := S1;
elsif
```

Eg:-

VHDL

```
If signal1 = '1' then
temp := S1;
elsif signal2 = '1' then
temp := S2;
else
 temp := S3;
end if;
```

Verilog

```
if (signal1 == 1)
temp = S1;
else if (signal2 == 1)
temp = S2;
else
temp = S3;
```

After execution :-

signal1	signal2	temp
0	0	S3
0	1	S2
1	0	S1
1	1	S1

⇒ Boolean expression may specify other relational operators, such as inequality or greater than or less than.

Behavioral description of a 2x1 multiplexer with tristate output low Enable signal:

⇒ To describe the behavior of the output of a multiplexer with a change in input, we develop a flowchart for the multiplexer.

⇒ The flowchart shows how the output behaves with the input.

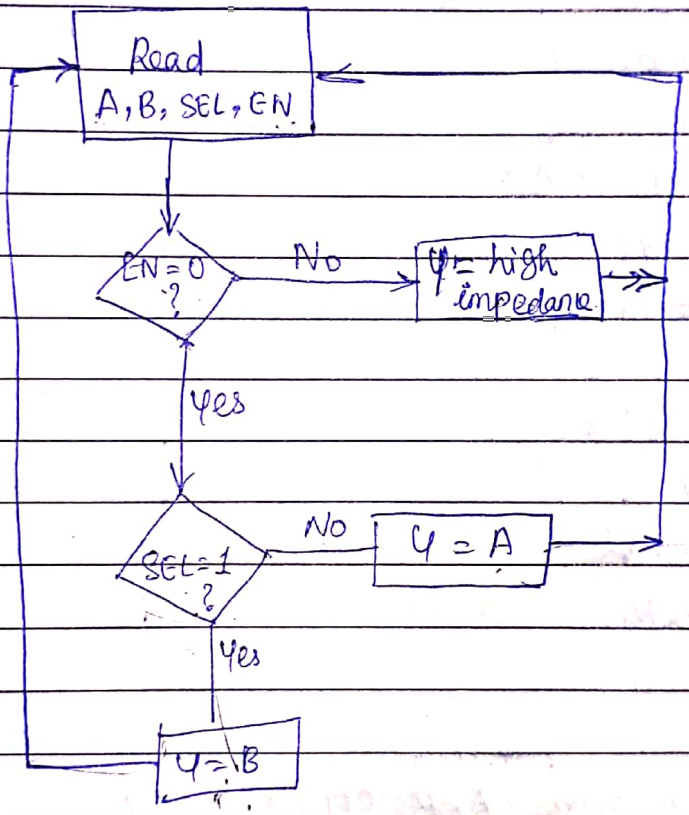
EN → high
o/p → 'z'

EN - low
o/p → B ; Sel-high
→ A ; Sel-low

⇒ The output is high impedance if the enable is high when the enable is low,

⇒ The output is equal to input B if select is high, otherwise the output is equal to A.

⇒ we do not need to know the logic diagram of the multiplexer to write the HDL behavioral ~~diagram~~ description.



HDL description of 2x1 multiplexer using if-else

VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux_if is
port ( A, B, SEL, EN : in std_logic; Y: out std_logic);
end mux_if;
```

```
architecture MUX_bh of MUX_if is
begin
process (SEL, A, B, EN)          -- Sensitivity list
variable temp : std_logic;       -- Common practice to
begin                                avoid the timing error
                                     due to sequential
                                     execution
if EN = '0' then
   if SEL = '1' then
      temp := B;
   else
      temp := A;
   end if;
   Y <= temp;
else
   Y <= 'Z';
end if;
end process;
end mux_bh;
```

Verilog :-

```
module MUX2X1 (A, B, SEL, EN, Y);
input A, B, SEL, EN;
output Y;
reg Y;
always @ (SEL, A, B, EN)
begin
   if (EN == 1)
   Y = 1'bz;
   else
   begin
      if (SEL)
         Y = B;
      else
         Y = A;
   end
end
endmodule
```

// procedural assignments are used to assign values to variables declared as regs. It can appear inside always, initial, fn's and tasks //

Simulation wave form:

HDL description of 2x1 multiplexer using Else-if:

VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity muxBH is
  port (A, B, SEL, EN: in std_logic; y: out std_logic);
end muxBH;
architecture mux_bh of muxBH is
begin
process (SEL, A, B, EN)
Variable temp : std_logic;
begin
  if (EN = '0') and (SEL = '1') then
  temp := B;
  elsif (EN = '0') and (SEL = '0') then
  temp := A;
  else
  temp := 'Z';
  end if;
  y <= temp;
  end process;
  end mux_bh;
```

Verilog:

```
module MUXBH (A, B, SEL, EN, Y);
input A, B, SEL, Gbar;
output Y;
reg Y;
always @ (SEL, A, B, EN)
begin
  if (EN == 0 & SEL == 1)
  begin
    Y = B;
  end
  else if (EN == 0 & SEL == 0)
    begin
      Y = A;
    end
    else
  begin Y = 1'bz;
end
end module
```

Signal and Vauiable assignment:

⇒) A process is written based on signal-assignment statements, and the other process is written based on variable-assignment statements.

⇒) A comparision of the simulation waveforms of these two processes will highlight the differences b/w the two assignment statements.

Ex: 2 D- latch using vaiible and Signal Assignments

Functionality of D-latch :

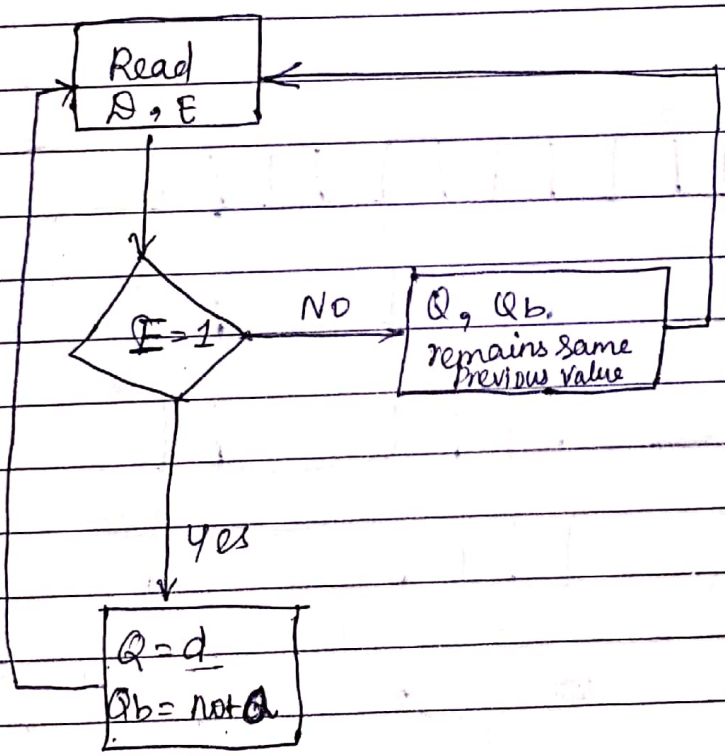
if EN is active

O/p → follows the i/p 'D'.

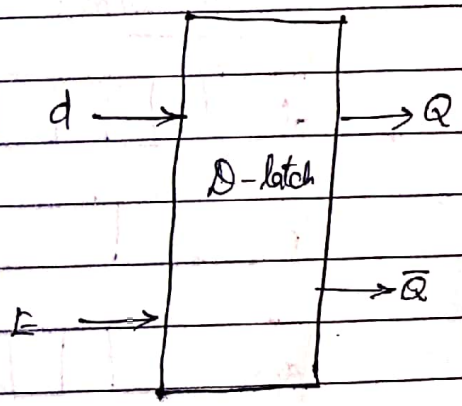
EN is inactive

O/p remains same (unchanged).

Flow chart



Logic symbol



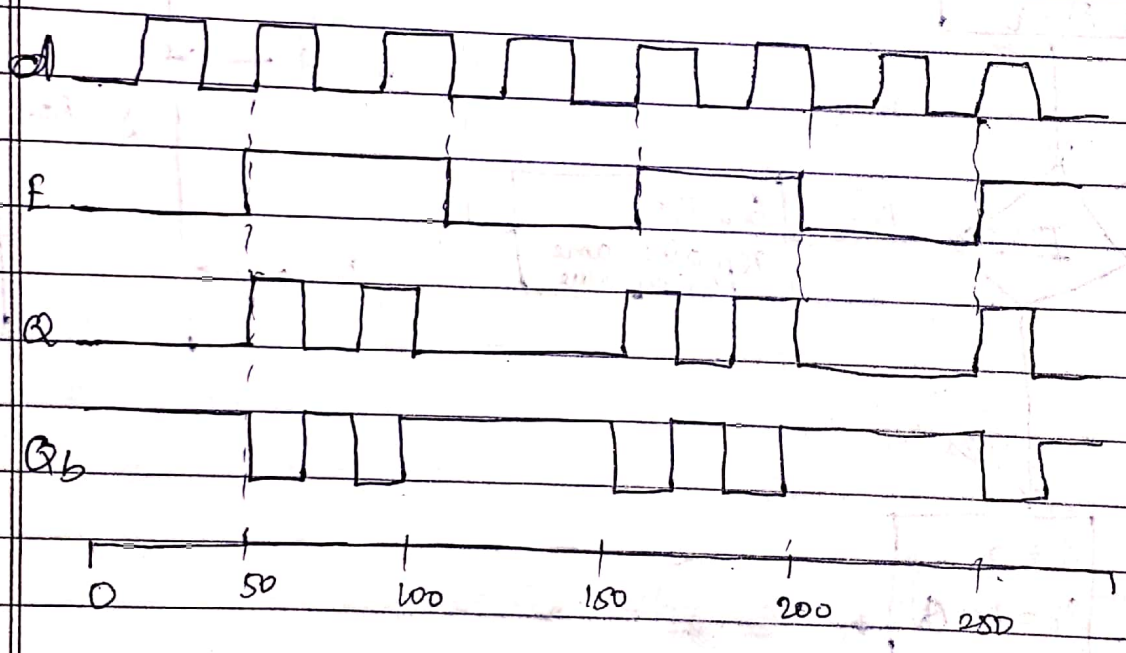
VHDL Code using Vaiable assignment statements:

VHDL

```
entity DL_Vai is
port (d, E : in bit ; Q, Qb : out bit);
end DL_Vai;
architecture DL_behav of DL_Vai is
begin
VAR : process (d, E)
Vaiable temp1, temp2 : bit ;
```

```
begin
  if E = '1' then
  temp1 := d ;
  temp2 := not temp1 ;
  end if ;
  Qb <= temp2 ;
  Q <= temp1 ;
  end process VAR ;
  end ~~BEHAV~~ DL_behav ;
```

waveform :

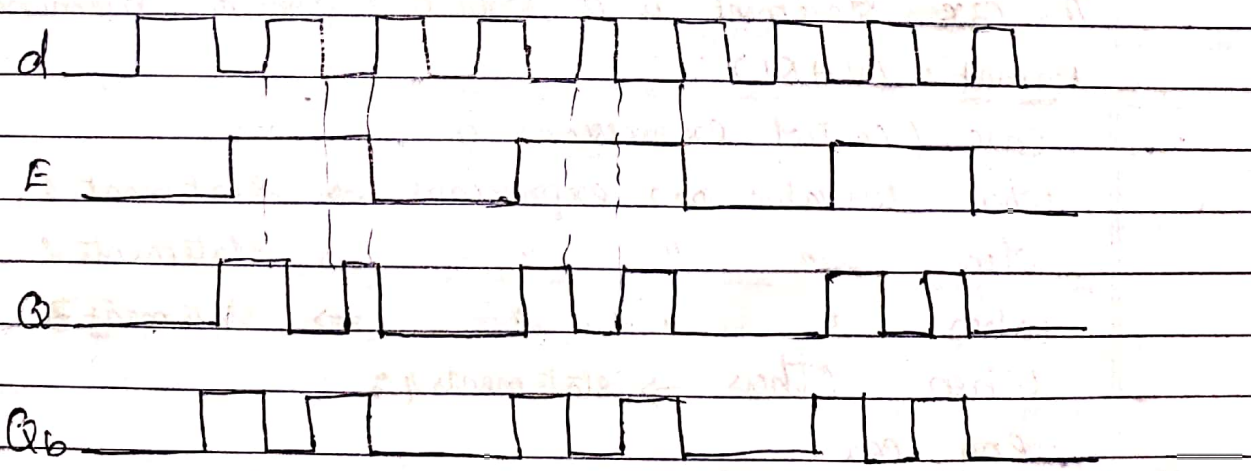


Clearly from the waveform, the code correctly represents a D-latch where Q follows 'd' when E is high ; other wise retains its previous value. Also Qb is the invert of Q at all times.

VHDL Code Of D-latch using Signal-Assignment statements

```
entity D-latch is
port (d, E : in bit ; Q : buffer bit ; Qb : out bit);
end D_latch;
architecture Dlatch_behave of D latch is
begin
process (d, E)
begin
   if E = '1' then
   Q <= d ;          -- Signal assignment
   Qb <= not Q;
  end if;
end process;
end Dlatch_behave;
```

wave form:-



the wave form shows the simulation of D-latch.
=> The figure shows, Q following Qb, which is an error because Qb should be the invert of Q.

Verilog code for D-latch:

```
module D-latch (d, E, Q, Qb);
input  d, E;
output Q, Qb;
reg    Q, Qb;
always @ (d, E)
begin
    if (E == 1)
    begin
    Q = d;
     Qb = ~Q;
    end
endmodule
```

(2) Case Statement:

The case statement is a sequential control statement.

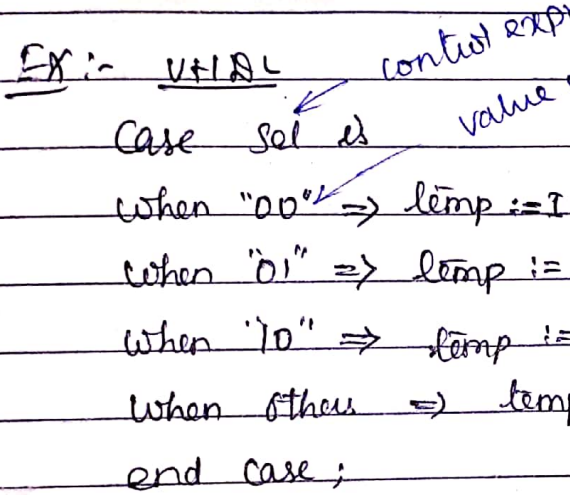
Format :- (VHDL)

```
Case (control expression) is
when testvalue (or) expression1 => statements 1;
When   "    "    "              => statements 2;
when   "    "    "              => statement 3;
When  Others => statements 4;
end case;
```

Verilog:

```
Case (control-expression)
test value1 : begin Statements 1 ; end
test value2 : begin statements 2 ; end
test value3 : begin statements 3 ; end
default : begin default Statements end
endcase
```

Ex :- VHDL




```
Case sel is
when "00" => temp := I1;
when "01" => temp := I2;
when "10" => temp := I3;
when others => temp := I4;
end case;
```

Verilog:

```
Case sel
2'b00 : temp = I1;
2'b01 : temp = I2;
2'b10 : temp = I3;
default : temp = I4;
endcase
```

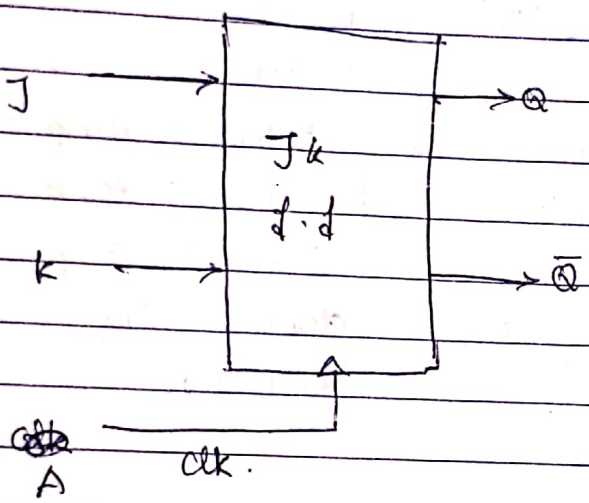
⇒ the control is "sel";

If sel = 00, then temp = I1;
If sel = 01, then temp = I2;
If sel = 10, then temp = I3;
if sel = 11 or other values then temp = I4.

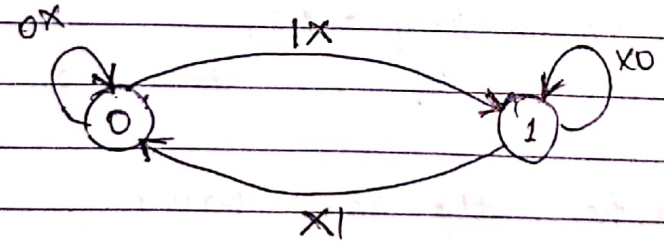
⇒ All four values have the same priority.
(i.e) if sel = '10', then the 3rd statement is executed directly without checking the first & second expressions.

Ex: Behavioral Description of a positive edge-Triggered Jk-f.f using the Case statement:

=> edge triggered f.f's are sequential circuits.
=> They are triggered by the edge of the clk.
=> (+)ive (or) (-)ive edge f.f's sample the input only at the (+)ive (or) (-)ive edges of the clk.



Symbol



State diagram

=> The State diagram shows the possible states of 'q' state 0 or) state 1.

=> Transition b/w these two states has to occur only at the (+)ive edges of the clk.

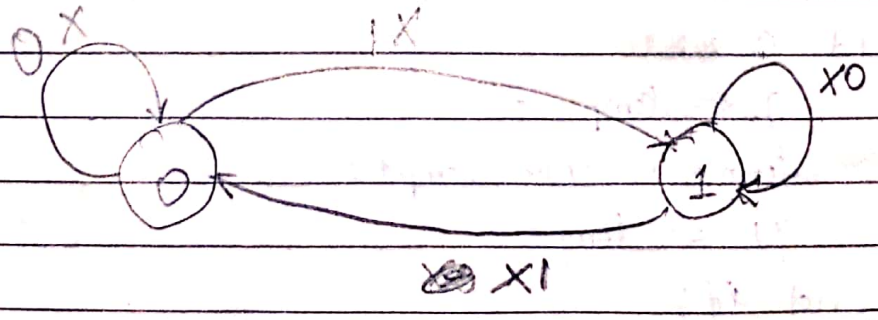
⇒ If the current state is 0 (q = 0), then the next state is 0 (or 1) if,

JK = 0x (1x)

where 'x' is don't care

⇒ If the Ct state is 1 (q = 1), then the next state is 1 (0) if, Jk = x0 (x1).

J	K	Clk	q (next state)
0	0	↑	No change (hold), next = current
1	0	↑	1
0	1	↑	0
1	1	↑	Toggle (next state) = invert of current state
X	X	no active edge	No change (hold), next = current



HDL code for a (+)ve edge-Triggered JK ff using the case statement :-

VHDL:

```
library ieee;
use ieee.std_logic_1164.all;
entity JK_FF is
port ( JK : in bit_vector (1 downto 0);
       clk : in std_logic ; q, qb : out bit);
end JK_FF;
architecture JK_BHE of JK_FF is
begin
process (clk, JK)
Variable temp1, temp2 : bit ;
begin
if rising_edge (clk) then
case JK is
when "01" => temp1 := '0' ;
when "10" => temp1 := '1' ;
when "00" => temp1 := temp1 ;
when "11" => temp1 := not temp1;
end case;
    q <= temp1 ;
    temp2 := not temp1;
    qb <= temp2;
end if;
end process ;
end JK_BHE;
```

verified

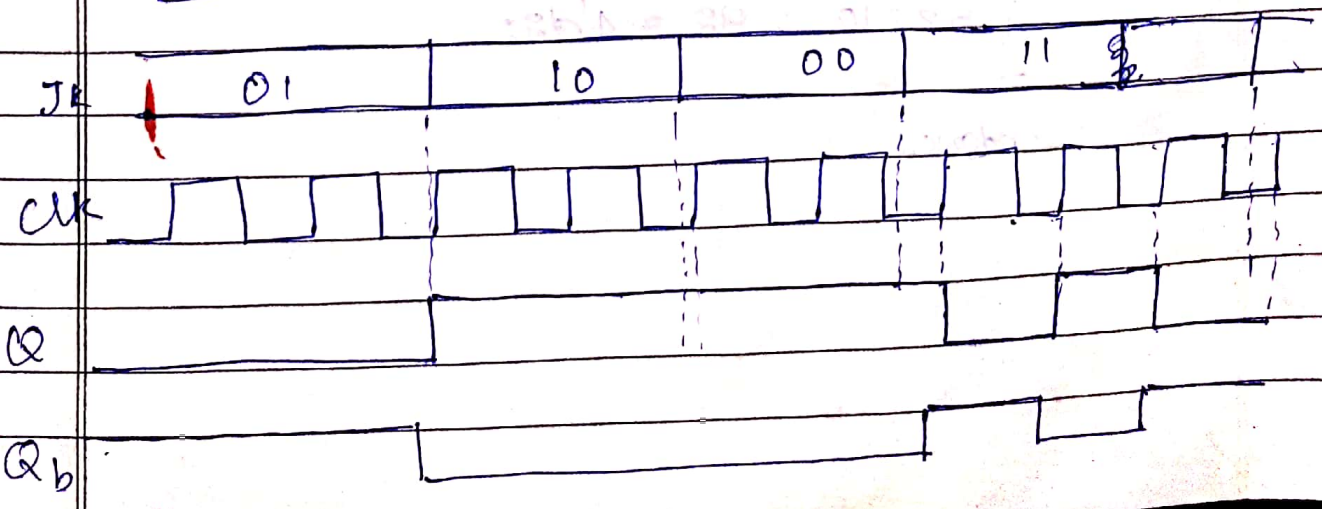
Verilog:

```
module JK_FF (Jk, clk, q, qb);
input [1:0] Jk;
input clk;
output q, qb;
reg q, qb;
always @ (posedge clk, Jk)
begin
case (Jk)
2'd0 : q = q;
2'd1 : q = 0;
2'd2 : q = 1;
2'd3 : q = ~q;
endcase
qb = ~q;
end
endmodule
```

=) where rising_edge (VHDL) and posedge (verilog) are predefined words called "attributes".

=) They represent the (+)ive edge of the clk.

Simulation waveform:



Verilog casex and casez:

⇒ Verilog has another 2 variations of "case" statement : casex and casez

⇒ casex ignores the "don't care" values of the control expression.

⇒ casez ignores the "high impedance" values of the control expression.

EX:-

```
casex (a1)
    4'bxxx1 : a2 = 4'd1;
    4'bxx10 : a2 = 4'd2;
    -----
endcase
```

all occurance of x is ignored.
a2 = 1 if, and only if, the LSB of a1 is 1
a2 = 2 if the bits of order 0 & 1 are 10.

EX:-

```
casez (a1)
    4'bzzz1 : a2 = 4'd1;
    4'bzz10 : a2 = 4'd2;
    -----
endcase
```

Ex:- Verilog Description of a priority encoder using Casex:-

=> priority encoder encodes the inputs according to a Priority set by the user.

Ex:-

When the i/p's represent interrupt requests. if, two are more interrupt requests are issued at the same time by devices needing service, and the Central Processing unit (CPU) can serve only one device at a time, then one of these request should be given priority over the others and be serve first.

=> The priority encoder can handle this task.

=> The i/p to the priority encoder is the interrupt requests and the o/p of the priority encoder can be memory addresses where the service routine is located.

Truth table:

input	output
a	b
xxx1	1
xx10	2
x100	4
1000	8
others	0

Verilog description for a 4-bit priority encoder:

```
module encoder_4 (Int_req, Rout_addrs);
input [3:0] int_req;          (a)
output [3:0] Rout_addrs;      (b)
reg [3:0] Rout_addrs;         (b)
always @ (Int_req) (a)
begin
    casex (Int_req) (a)
        4'bxxx1 : Rout_addrs (b) = 4'd1;
        4'bxx10 : Rout_addrs (b) = 4'd2;
        4'bx100 : Rout_addrs (b) = 4'd4;
        4'b1000 : Rout_addrs (b) = 4'd8;
        default : Rout_addrs (b) = 4'd0;
    endcase
end
endmodule
```

simulation output:

(a) Int_req	1111	1110	1000	0011	1100	0101	0000	0110
(b) Rout_addrs	0001	0010	1000	0001	0100	0001	0000	0010

bit 0 of input a has highest priority.

LOOP statement:

=> loop is used to repeat the execution of statements written inside its body.

=> The No of repetitions is controlled by the range of an index parameter.

=> The loop allows the code to be shortened.

(1) For-loop:

General format:

```
for (lower index value) (upper index value) (step)
statement 1; statement 2; statement 3; ....
end loop
```

=> If the value of index is b/w lower & upper, all statements written inside the body of the loop are executed.

=> For each cycle, the index is modified at the end of loop. According to the step.

=> If the value of index is not b/w the lower & upper value, the loop is terminated.

Ex :-

VHDL

```
for i in 0 to 2 loop
if temp(i) = '1' then
result := result + 2**i ;
end if;
end loop;
```

Verilog:

```
for(i=0 ; i<=2 ; i=i+1)
begin
if (temp[i] == 1)
begin
result = result + 2** i;
end
end
```

⇒ The index is 'i' lower value is '0' & the upper value is 2. the step is '1'.

⇒ All statements b/w the for statement and end loop (VHDL) or end (verilog) are executed until the index 'i' goes out of range.

⇒ At the very beginning of the loop, 'i' takes the value of '0' and the statements if and result are executed as:

```
if (temp(0) = '1' then
result := result + 2** 0 ;
```

⇒ When the program encounters the end of the loop it increments i to 1.

⇒ If 'i' is less than or equal to 2, the loop is repeated; otherwise, the program exits the loop and executes statements. statements

⇒ In VHDL, we do not have to declare index 'i'; in verilog, it has to be declared.

(2) While - loop:

general format:

```
while (condition)
statement 1;
statement 2;
.....
end
```

⇒ As long as the condition is true, all statements written before the end of the loop are executed, otherwise the program exits the loop.

Ex:- VHDL

```
while (i < x) loop
  i := i+1;
  z := i*z;
end loop;
```

Verilog:

```
while (i < x)
begin
  i = i+1;
  z = i*z;
end
```

⇒ In the above example the condition is (i < x). As long as 'i' is less than x, i is incremented, and the product i*z is calculated and assigned to 'z'.

(3) Verilog repeat :

⇒ In Verilog, the sequential statement "repeat" causes the execution of statements b/w its begin and end to be repeated a fixed number of times ; no condition is allowed in repeat.

Ex:-
```
repeat (32)
begin
    #100 i = i+1;
end
```

⇒ 'i' is incremented 32 times with a delay of 100 screen time units.

(A) Verilog Forever :

⇒ The statement "forever" in verilog repeats the loop endlessly.

⇒ One common use of 'forever' is to generate clocks.

Ex:-
```
Initial
begin
clk = 1'b0;
forever #20 clk = ~clk;
end
```

VHDL Next and Exit:

In VHDL, next & exit are two sequential statements associated with loop.

Exit => causes the program to exit the loop
next => causes the program to jump to the end of the loop.

Ex:-

```
for i in 0 to 2 loop
....
....
next when z = '1';
Statement1; Statement2;
end loop;
Statement2;
```

=> In this example, at the beginning of the loop is execution, i takes the value 0; at the statement next when z = 1, the program checks the value of z,
=> if z=1, the statement1 is skipped and incremented to 1.
=> The loop is then repeated with i=1.
=> If 'z' is not equal to '1' the statement 1 is executed and 'i' is incremented to 1, and the loop is repeated.

Behavioral Description of a 4-bit +ive edge Triggered counter

Calculating the factorial using behavioral description with while-loop:

⇒ HDL behavioral description is used to find the factorial of a +ive number 'N'.

⇒ The factorial of N, $N! = N(N-1)(N-2)(N-3)\ldots 1$.

EX:- $4! = 4(3)(2)(1) = 24$.

⇒ Where N - i/p } both are declared as natural;
Z - o/p. }

this restricts the values of N & Z as +ive integers.

VHDL:

```
entity factorial is
port (N: in natural ; Z: out natural);
end factorial;
architecture factorial_behav of factorial is
begin
process (N)
variable Y, i : natural;
begin
    Y := 1;
    i := 0;
```

```
while (i < N) loop
  i := i+1;
  y := y * i;
end loop
z <= y;
end process;
end factorial-behav;
```

N = 4

0 < 4 → 0 < 4

i = 1 → i = 1

y = → y = 1 * 1 = 1

1 < 4

i = 2

y = 1 * 2 = 2

2 < 4

i = 3

y = 2 * 3 = 6

3 < 4

i = 4

y = 6 * 4 = 24

4 < 4

Verilog :

```
module factorial (N, Z);
output [15:0] Z;
input [15:0] N;
reg [15:0] Z;
integer i;
always @ (N)
begin
  z = 1;
  i = 0;
while (i < N)
begin
  i = i+1;
  Z = i * z;
end
end
endmodule
```

Booth's multiplication Algorithm:

=> Applicable for signed binary numbers only.

=> Employs 2's complement scheme to represent all signed binary integer.

=> Reduces number of multiplication steps.

=> uses both add & subtract as well as right shift arithmetic.

left most bit → sign bit.

=> two 4 bit numbers as multiplicand and multiplier → the product is 8 bit's with left most bit sign bit.

~~00101 → right shift only.~~

=> X → multiplicand (4 bit)
y → multiplier (4 bit)

Sum → product (8 bit)
initialized as 0000 0000

=> String :- A string consists of one or more consecutive one's.

The beginig of the stiij is the tiansition from 0 to 1. while the end of the stiij is tiansition from 1 to 0.

⇒ To detect the tiansition, a 1 bit reg 'E' is used to hold '0' initially.

⇒ By compaiij 'E' with the bits of 'X', the begiiij and end of the stiij can be detected.

Ex:- 5 X -5

X = 5 = 1011

Y = -5 = 0101 (2's comp. of 5)

-Y = -(-5) = 5 = 1011 (2's comp of 4)

E = 0

Sum = 0000 0000

	X(i)	E	sum	commands
i=0	1011	0	0000 0000	
			1011	sub y. (add -y)
			1011 0000	
			0101 1000	shift right
		1	1101 1000	sum[7] = sum[6] E = x(0)
i=1	1011	1	0110 1100	shift right
		1	1110 1100	sum(7) = sum[6] E = x[1]
i=2	1011	1	01	add y