UNIT-3
BEHAVIORAL MODELING

Syllabus: Introduction, Operations and Assignments ,Functional Bifurcation ,*Initial* Construct, *Always* Construct ,Assignments with Delays *Wait* Construct ,Multiple Always Blocks ,Designs at Behavioural Level ,Blocking and Non-Blocking Assignments ,The case statement ,Simulation Flow, *if* and *if-else* constructs ,*assign–deassign* construct, *repeat* construct, *for* loop, the *disable* construct, *while* loop, *forever* loop, parallel blocks, *force-release* construct, Event.

Introduction: Behavioural level modelling constitutes design description at an abstract level. One can visualize the circuit in terms of its key modular functions and their behaviour; it can be described at a functional level itself instead of getting bogged down with implementation details.

Operations and Assignments The design description at the behavioural level is done through a sequence of assignments. These are called 'procedural assignments' – in contrast to the continuous assignments at the data flow level. All the procedural assignments are executed sequentially in the same order as they appear in the design description.

The procedure assignment is characterized by the following:

1. The assignment is done through the "=" symbol
2. An operation is carried out and the result assigned through the "=" operator to an operand specified on the left side of the "="

   for example, $N = \sim N$;

   Here the content of **reg** *N* is complemented and assigned to the reg *N* itself. The assignment is essentially an updating activity.

3. The operation on the right can involve operands and operators. The operands can be of different types – logical variables, numbers – real or integer and so on.

4. The operands on the right side can be of the net or variable type. They can be scalars or vectors.

5. It is necessary to maintain consistency of the operands in the operation expression –

   e.g., $N = m / l$;

Here *m* and *l* have to be same types of quantities – specifically a **reg**, **integer**, **time**, **real**, **real time**, or memory type of data.

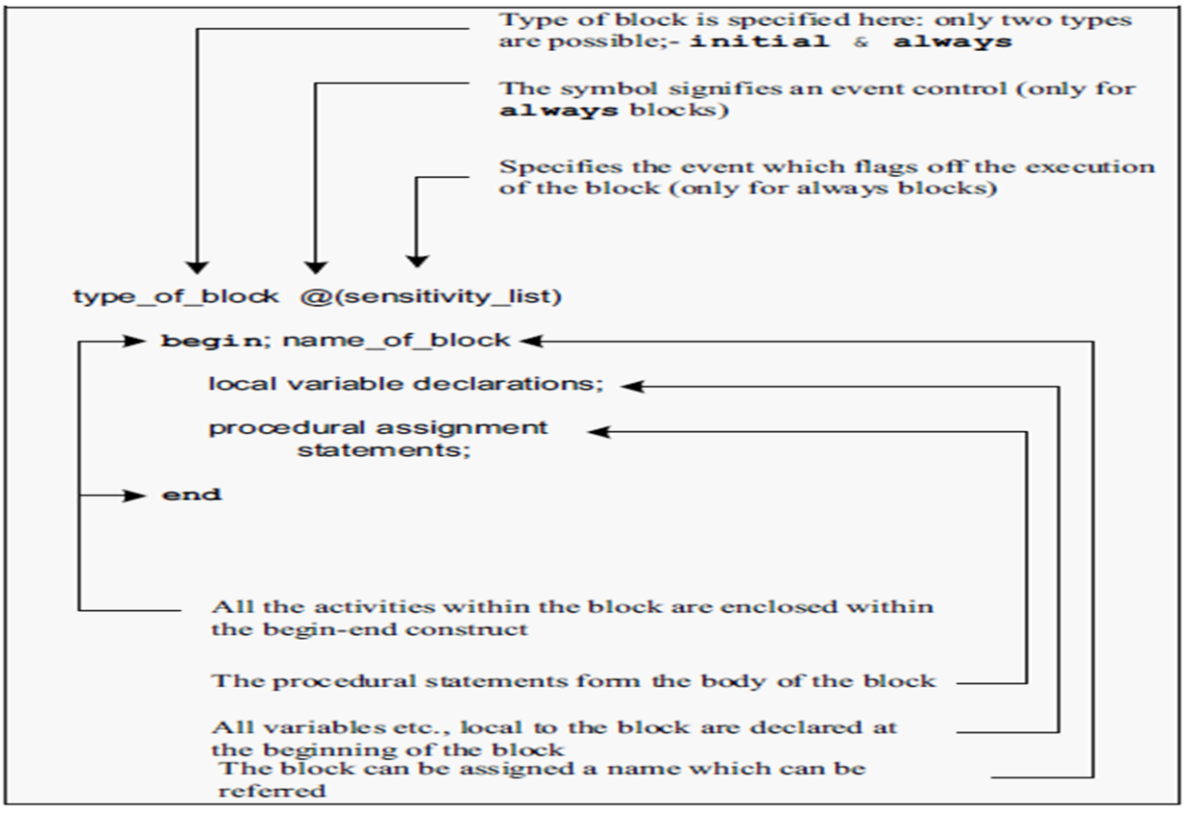
6. The operand to the left of the "=" operator has to be of the variable (*e.g.*, **reg**) type. It has to be specifically declared accordingly. It can be a scalar, a vector, a part vector, or a concatenated vector.

Functional Bifurcation Design description at the behavioural level is done in terms of procedures of two types;

1. one involves ***functional description and interlinks of functional units.*** It is carried out through a series of blocks under an **"always".**
2. The second concerns ***simulation*** – its starting point, steering the simulation flow, observing the process variables, and stopping of the simulation process
3. all these can be carried out under the "always" banner, an "initial" banner, or their combinations.

Procedure- Block Structure

A procedure-block of either type – initial or always

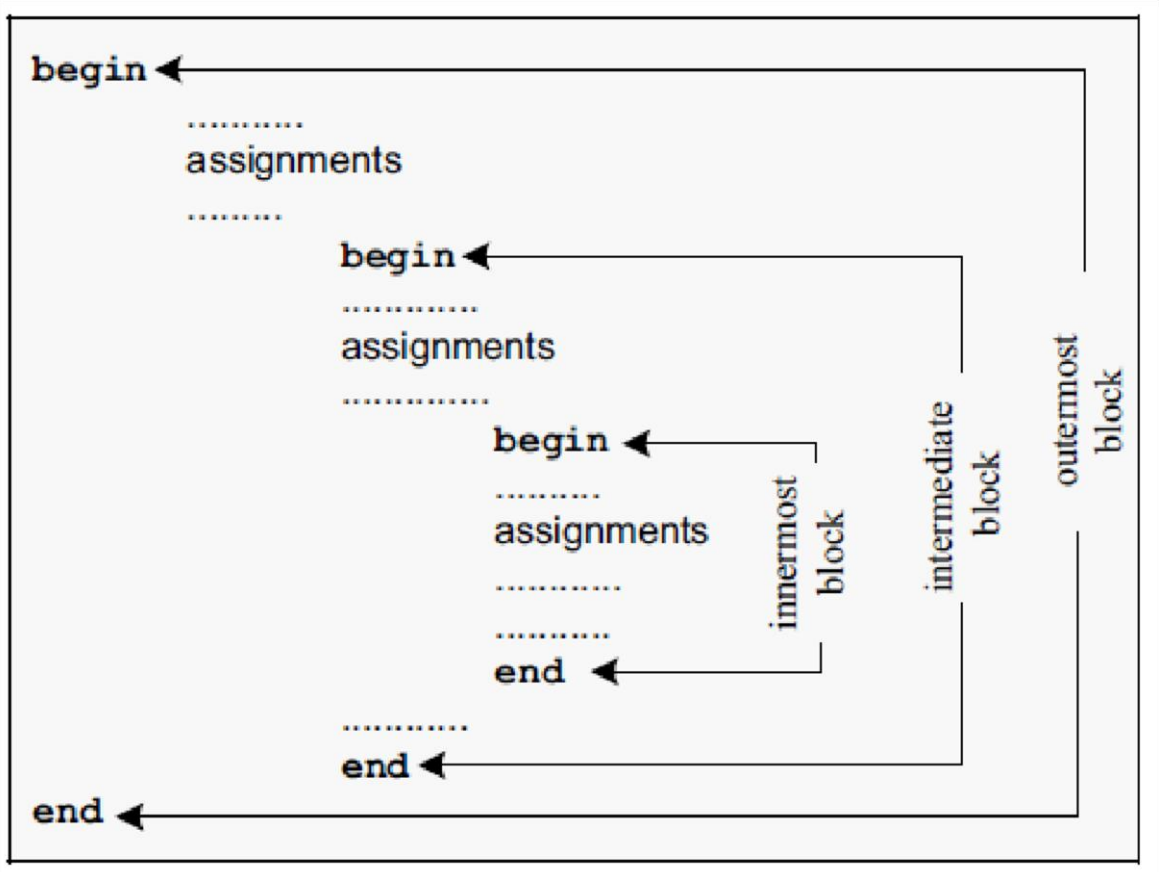


Begin-End Construct

- If a procedural block has only one assignment to be carried out, it can be specified as **initial #2 a=0;**
- More often more than one procedural assignment is to be carried out in an initial block.

- All such assignments are grouped together between "begin" and "end" declarations.
- Every **begin** declaration must have its associated **end** declaration.
- begin – end constructs can be nested as many times as desired.

Nested Begin –End Blocks



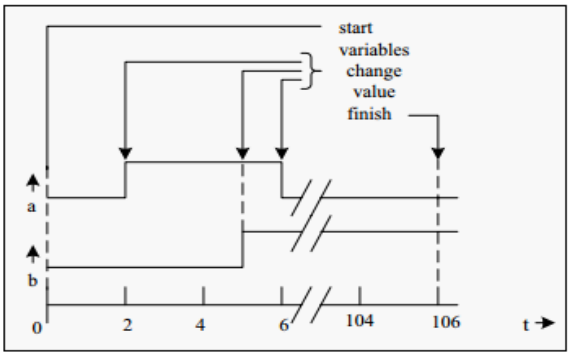
Assigning names to blocks serves different purposes:

- Registers declared within a block are local to it and are not available outside. However, during simulation they can be accessed for simulation, *etc*., by proper dereferencing
- Named blocks can be disabled selectively when desired

Initial Construct

- A set of procedural assignments within an initial construct are executed only Once
- In any assignment statement the left-hand side has to be a storage type of element (and not a net). It can be a reg, integer, or real type of variable. The right-hand side can be a storage type of variable or a net.

```
reg a,b;
initial
        begin
                a = 1'b0;
                b = 1'b0;
        #2      a = 1'b1;
        #3      b = 1'b1;
        #1      a = 1'b0;
        #100$stop;
        end
```



Multiple Initial Blocks

A module can have as many **initial** blocks as desired. All of them are activated at the start of simulation. The time delays specified in one **initial** block are exclusive of those in any other block.

```
module nil1;
initial
reg a, b;
begin
        a = 1'b0;
        b = 1'b0;
        $display ($time,"display:  a = %b,  b = %b", a, b);
  #2    a = 1'b1;
  #3    b = 1'b1;
  #1    a = 1'b0;
end
initial  #100$stop;
initial $monitor ($time, "monitor:  a = %b,  b = %b", a, b);
initial
        begin
   #2   b = 1'b1;
        end
endmodule
```

Always Construct

- The **always** process signifies activities to be executed on an "always basis."
- Its essential characteristics are:
  - Any behavioral level design description is done using an always block.
  - The process has to be flagged off by an event or a change in a net or a reg. Otherwise it ends in a stalemate.
  - The process can have one assignment statement or multiple assignment statements.
  - Normally the statements are executed sequentially in the order they appear.

Event Control

- The always block is executed repeatedly and endlessly. It is necessary to specify a condition or a set of conditions, which will steer the system to the execution of the block. Alternately such a flagging-off can be done by specifying an event preceded by the symbol "@".

- @(negedge clk) :executes the following block at the negative edge of clk.
- @(posedge clk) : executes the following block at the positive edge of the clk.
- @(prt or clr) :
- @(posedge clk1 or negedge clk2) :
- @ (a or b or c) can also write as @ (a or b or c)

@ (a, b, c)
@ (a, b or c)
@ (a or b, c)

Example 1: UP Counter

```
module counterup(a,clk,N);
input clk;  input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a=4'b0000;
always@(negedge clk) a=(a==N)?4'b0000:a+1'b1;
endmodule
```

Example 2: Down counter

```
module counterdn(a,clk,N);
input clk;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
always@(negedge clk) a=(a==4'b0000)?N:a-1'b1;
endmodule

module tst_counterdn();//TEST_BENCH reg clk;
reg[3:0]N; wire[3:0]a;
counterdn cc(a,clk,N);
initial
begin
        N= 4'b1010;
        Clk = 0;
end
always #2 clk=~clk;
initial $monitor($time,"a=%b,clk=%b,N=%b",a,clk,N); initial #55 $stop;
endmodule
```

Design module of an up down counter and a test bench

```
module updcounter(a,clk,N,u_d);
input clk,u_d;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
always@(negedge clk)
a=(u_d)?((a==N)?4'b0000:a+1'b1):((a==4'b0000)?N:a-
1'b1);
endmodule

module tst_updcounter();//TEST_BENCH
reg clk,u_d;
reg[3:0]N;
wire[3:0]a;
updcounter c2(a,clk,N,u_d);
initial
begin
     N    = 4'b0111;
     u_d = 1'b0;
     clk = 0;
end
always #2 clk=~clk;
always #34u_d=~u_d;
initial $monitor
($time,"clk=%b,N=%b,u_d=%b,a=%b",clk,N,u_d,a);
initial #64 $stop;
endmodule
```

Design module of a shift register with facility for right or left shift and a test bench

module shifrlter(a,clk,r_l); input clk,r_l;
output [7:0]a;
reg[7:0]a; initial a= 8'h01;
always@(negedge clk)
begin
a=(r_l)?(a>>1'b1):(a<<1'b1);
end endmodule

module tst_shifrlter;//test-bench reg clk,r_l;
wire [7:0]a;
shifrlter shrr(a,clk,r_l); initial
begin
clk =1'b1; r_l = 0;
end
always #2 clk =~clk; initial #16 r_l =~r_l; initial
$monitor($time,"clk=%b,r_l = %b,a =%b ",clk,r_l,a); initial #30 $stop;
endmodule

Design module of a D-flip-flop and a test bench

```
module dff(do,di,clk);
output do;
input di,clk;
reg do;
initial
do=1'b0;
always@(negedge clk) do=di;
endmodule

module tst_dffbeh();//test-bench
reg di,clk;
wire do;
dff d1(do,di,clk);
initial
begin
     clk=0;
     di=1'b0;
end
always #3clk=~clk;
always #5 di=~di;
initial
$monitor($time,"clk=%b,di=%b,do=%b",clk,di,do);
initial #35 $stop;
endmodule
```

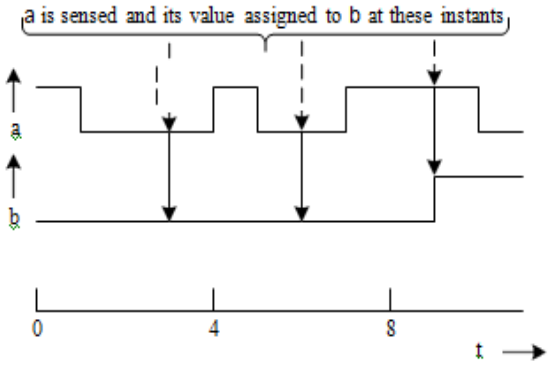
Design module of a D-latch and a test bench

```
module dffen(do,di,en); // d-latch
output do;
input di,en;
reg do;
initial
do=1'b0;
always@(di or en)
if(en)
do=di;
endmodule

module tst_dffbehen;//test-bench
reg di,en;
wire do;
dffen d1(do,di,en);
initial
begin
     en=0;
     di=1'b0;
end
always#7 en =~en;
always#4 di=~di;
initial
$monitor($time,"en=%b,di=%b,do=%b",en,di,do);
initial #50 $stop;
endmodule
```

ASSIGNMENT WITH DELAYS

- Specific delays can be associated with procedural assignments.
  Consider the assignment **always #3 b = a;**
- simulator encounters this at zero time and posts the entire activity to be done 3 ns later.
- The always nature of the activity, the assignment is scheduled to be repeated every 3 ns, irrespective of whether a changes in the meantime. Values of a at the 3rd, 6th, 9th, *etc*., ns are sampled and assigned to b.

```
module del1;
reg a,b;
always #3 b=a;
Initial

begin
                a = 1'b1;
                b = 1'b0;
        #1      a = 1'b0;
        #3      a = 1'b1;
        #1      a = 1'b0;
        #2      a = 1'b1;
        #3      a = 1'b0;
end
initial $monitor($time," a = %d, b = %d", a, b);
initial #20 $finish;
endmodule
```



INTRA ASSIGNMENT DELAYS

- The "intra-assignment" delay carries out the assignment in two parts.
- An assignment with an intra- assignment has the form
  A = # dl expression;
- Here the expression is scheduled to be evaluated as soon as it is encountered. However, the result of the evaluation is assigned to the right-hand side quantity after a delay specified by dl.
- In case it is an expression, it is evaluated and execution delayed by the number of time steps. If the number evaluates to a negative quantity, the same is interpreted as a 2's complement value. In the statement
  always #b a = a + 1;
  a and b are variables. The execution incrementing a is scheduled at b ns. If b changes, the execution time also changes accordingly.
- As another example consider the procedural assignment
  always #(b + c) a = a + 1;
  Here a, b, and c are variables. The algebraic addition of variables b and c is to be done. The scheduler schedules the incrementing of a and reassigning the incremented values back to a with a time delay of (b + c) ns.
- As an additional example consider the assignment below with an intra-assignment delay.
  always #(a + b) a = #(b + c) a +1;

  Here the simulator evaluates (a + b) during simulation. After a lapse of (a + b) ns, execution of the statement is taken up; (a + 1) is evaluated and assigned as the new value of a – but the assignment is delayed by (b + c) ns.

Zero Delay: A delay of 0 ns does not really cause any delay. However, it ensures that the assignment following is executed last in the concerned time slot.

```
always
begin a=1;
#0  a=0;
end
```

WAIT construct

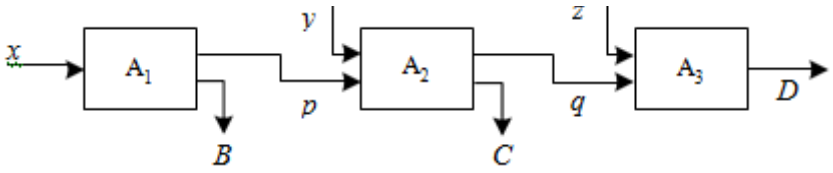
- The **wait** construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment or group of assignments.
- Its syntax has the form **wait (alpha) assignment1;**
  alpha can be a variable, the value on a net, or an expression involving them.
- If alpha is an expression, it is evaluated; if true, assignment1 is carried out. One can also have a group of assignments within a block in place of assignment1.
- The activity is level-sensitive in nature, in contrast to the edge-sensitive nature of event specified through @. Specifically the procedural assignment
  @clk a = b;
  assigns the value of b to a when clk changes;
  if the value of b changes when clk is steady, the value of a remains unaltered.
- In contrast, with **wait**(clk) #2 a = b;
  the simulator waits for the clock to be high and then assigns b to a with a delay of 2 ns. The assignment will be refreshed as long as the clk remains high.

Example: The counter module uses a **wait** construct. It has an enable input En. The counter is active and counts only when En = 1

```
module ctr_wt(a,clk,N,En);
input clk,En;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a=4'b1111;
always
begin
        wait(En)
        @(negedge clk)
        a=(a==N)?4'b0000:a+1'b1;
end
endmodule
```

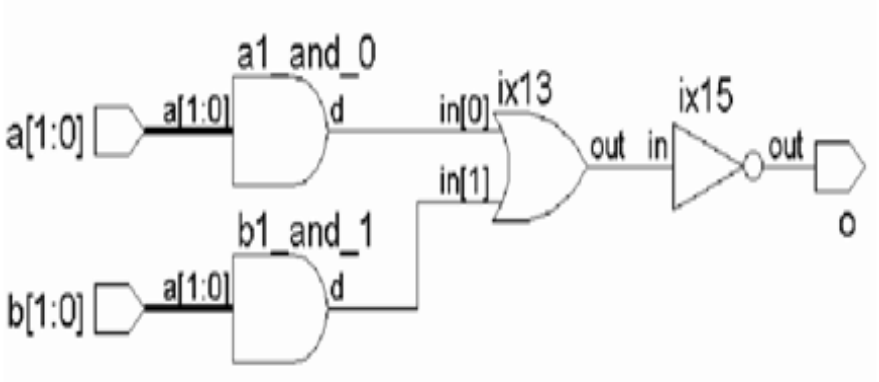
Multiple Always Blocks:

- All the activities within an always block are scheduled for sequential execution.
- The activities can be of a combinational nature, a clocked sequential nature, or a combination of these two. (A design description involving such combinations is conventionally called the 'Register Transfer Level' description.)
- Basically, any circuit block whose end-to-end operation can be described as a continuous sequence can be described within an **always** block.



- A typical circuit block conforming to the above description has three activities termed A1, A2, and A3. These three are to be done in that order. Activity A1 accepts *x* as input, and it generates output *B* and *p*. *p* and *y* form inputs to activity A2.
- Similarly activity A2 generates outputs *c* and *q* after activity A1 is completed. *q* and *z* form outputs of A2. After activity A2 is completed, activity A3 is scheduled. It accepts *z* and *q* as inputs and generates *D* as output.
- Here if A1, A2, and A3 are logical activities, the whole block can be synthesized as a combinational logic unit. If one or more of these are clocked events, execution may be sequential.

Designs at Behavioral level




```
module aoibeh(o,a,b);
output o;
input[1:0]a,b;
reg o,a1,b1,o1;
always@(a[1] or a[0]or b[1]or b[0])
begin
       a1=&a;
       b1=&b;
       o1=a1||b1;
       o=~o1;
end
endmodule
```

```
module aoibeh1(o,a,b);
output o;
input[1:0]a,b;
reg o;
always@(a[1]ora[0]or b[1]orb[0]) o=~((&a)||(&b));
endmodule
```

```
module aoibeh2(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
and g1(a1,a[1],a[0]),g2(b1,b[1],b[0]);
always@(a1 or b1)
o=~(a1||b1);
endmodule
```

```
module aoibeh3(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
assign a1=&a,b1=&b;
always@(a1 or b1)o=~(a1||b1);
endmodule
```

Blocking and Non Blocking Assignments:

- All assignment within an initial or an always block done through an equality ("=") operator. These are executed **sequentially.** Such assignments block the execution of the following lot of assignments at any time step. Hence they are called "blocking assignments".
- If the assignments are to be effected concurrently A facility called the "non blocking assignment" is available for such situations. The symbol "<=" signifies a non-blocking assignment. The main characteristic of a non blocking assignment is that its execution is **concurrent**

```
A = 2'b00;
B = 2'b01;
A <= B;
B <= A;
```

Swapping variable values through non blocking assignments.

```
A = 2'b00;
B = 2'b01;
A = B;
B = A;
```

Another group of blocking assignments.

•First A is assigned the binary value 00, and then B is assigned the value 01.

•These two assignments are sequential.

•The subsequent two assignments are concurrent. The assignment

A <= B

"reads" the value of B, stores it separately, and then assigns it to A. The new value of a is 01.

•The assignment **B <= A ;**
takes the value of A– i.e., 00 – stores it separately and assigns it to B. Thus the new value of B is 00.
• After the block is executed, A has the value 01 while B has the value 00.

so

•All four assignments here are sequential in nature.
•The third one, namely

A = B;

assigns the value 01 to a;
• subsequently the fourth and following assignment **B = A ;**
assigns the present value of A (*i.e.*, 01) to b; The value of b remains at 01 itself.

Blocking Assignments and Delays

```
module nil1 (c1, a, b);
output c1;
input a, b;
reg c1;
always @(a or b)
        #3      c1 = a&b;
endmodule
```

Program1: A time delay in an evaluation

```
module nil2 (c2, a, b);
output c2;
input a, b;
reg c2;
always  @(a or b)
                c2 = #3 a&b;
endmodule
```

Program2: An intra-assignment delay

•In Program 1, which has a delay of 3 ns for the blocking assignment to c1.
•If a or b changes, the always block is activated. Three ns later, (a&b) is evaluated and assigned to c1.
•The event "(a or b)" will be checked for change or trigger again. If a or b changes, all the activities are frozen for 3 ns.
• If a or b changes in the interim period, the block is not activated.
• Hence the module does not depict the desired output.

•In Program2, with an intra-assignment delay of 3 ns to the assignment to c2.
•The always block is activated if a or b changes. (a & b) is evaluated immediately but assigned to c2 only after 3 ns.
•However, the behavior is not acceptable on two counts:
a)The output assignment has to wait for 3 ns after the change.
b)Only after the delayed assignment to c2, the event (a or b) checked for change.
• If a or b changes in the interim period, the block is not activated.

Non Blocking Assignments and Delays

```
module nil3 (c3, a, b);
output c3;
input a, b;
reg c3;
always @(a or b)
        #3      c3 <= a&b;
endmodule
```

```
module nil4 (c4, a, b);
output c4;
input a, b;
reg c4;
always @(a or b)
        c4 <= #3 a&b;
endmodule
```

Program 3: A time delay in a non- blocking assignment Program 4: An intra-assignment delay in a non blocking assignment.

- In Program 3, has a blocking delay of 3 ns; but the assignment is of the non blocking type.
- The block is entered if the value of a or b changes but the evaluation of a&b and the assignment to c3 take place with a time delay of 3 ns.
- If a or b changes in the interim period, the block is not activated.

In Program 4, possibly represents the best alternative with time delay.

- The always block is activated if a or b changes. (a&b) is evaluated immediately and scheduled for assignment to c4 with a delay of 3 ns.
- Without waiting for the assignment to take effect (*i.e.*, at the same time step as the entry to the block), control is returned to the event control operator. Further changes to a or b – if any are again taken cognizance of.
- The assignment is essentially a delay operation.

The **CASE** statement

- The **case** statement is an elegant and simple construct for multiple branching in a module.
- The keywords **case**, **endcase**, and **default** are associated with the **case** construct.
- First expression is evaluated. If the evaluated value matches ref1, statement1 is executed; and the simulator exits the block;

```
Case (expression)
Ref1 : statement1;
Ref2 : statement2;
Ref3 : statement3;
...
...
default: statementd;
endcase
```

- else expression is compared with ref2 and in case of a match, statement2 is executed, and so on.

- If none of the ref1, ref2, *etc*., matches the value of expression, the **default** statement is executed.

Example : Consider the module for a 2-to-4 decoder. The test bench is also included in the figure. One of the 4 output bits goes high, depending on the binary value of {i1, i2}. If i1, i2, or both take x or z values, there is no match and the **default** block is executed.

```
module dec2_4beh(o,i);
output[3:0]o;
input[1:0]i;
reg[3:0]o;
always@(i)
begin
case(i)
        2'b00:o=4'h0;
        2'b01:o=4'h1;
        2'b10:o=4'h2;
        2'b11:o=4'h4;
default:
begin
        $display ("error");
        o=4'h0;
end
endcase
end
endmodule
```

```
//test bench
module tst_dec2_4beh();
wire [3:0]o;
reg[1:0] i;
//reg en;
dec2_4beh dec(o,i);
initial
begin
         i =2'b00;
       #2i =2'b01;
       #2i =2'b10;
       #2i =2'b11;
       #2i =2'b11;
       #2i =2'b0x;
end
initial $monitor ($time   ,   " output o =   %b   , input i
= %b " , o ,i);
endmodule
```

```
output
# 0 output o =   0000   , input i   = 00
# 2 output o =   0001   , input i   = 01
# 4 output o =   0010   , input i   = 10
# 6 output o =   0100   , input i   = 11
# error
# 10 output o =   0000   , input i = 0x
```

Example : An ALU module along with a test bench. The ALU function has been realized through a block with a **case** construct. Additional functions can be added to the ALU by a direct expansion of the **case** block.

```
module  alubeh(c,s,a,b,f);
output[3:0]c;
output s;
input [3:0]a,b;
input[1:0]f;
reg s;
reg[3:0]c;
always@(a or b or f)
begin
      case(f)
      2'b00:      c=a+b;
      2'b01:      c=a-b;
      2'b10:      c=a&b;
      2'b11:      c=a|b;
      endcase
```

end
endmodule

```
module tst_alubeh;//test-bench
reg[3:0]a,b;
reg[1:0]f;
wire[3:0]c;
wire s;
alubeh aa(c,s,a,b,f);
initial
begin
f=2'b00;a=2'b00;b=2'b00;
end
always
begin
    #2 f=2'b00;a=4'b0011;b=4'b0000;
    #2 f=2'b01;a=4'b0001;b=4'b0011;
    #2 f=2'b10;a=4'b1100;b=4'b1101;
    #2 f=2'b11;a=4'b1100;b=4'b1101;
end
initial $monitor($time,"f=%b,a=%b,b=%b,c=%b",f,a,b,c);
initial #10 $stop;
endmodule
```

Casex and Casez

- The **case** statement executes a multiway branching where every bit of the **case** expression contributes to the branching decision.
- The statement has two variants where some of the bits of the **case** expression can be selectively treated as don't cares – that is, ignored.
- **Casez** allows **z** to be treated as a don't care. "?" character also can be used in place of **z**.
- **Casex** treats **x** or **z** as a don't care.

Example 3: A module for a priority encoder and a test bench. The encoder gives a 2-bit output. The binary output represents the position of the first one bit in the 4-bit input combination.

```
module pri_enc(a,b);
output[1:0]a;
input[3:0]b;
reg[1:0]a;
always@(b)
casez(b)
4'bzzz1:a=2'b00;
4'bzz10:a=2'b01;
4'bz100:a=2'b10;
4'b1000:a=2'b11;
endcase
endmodule
```

```
module pri_enc_tst;//test-bench
reg [3:0]b;
wire[1:0]a;
pri_enc pp(a,b);
initial b=4'bzzz0;
always
begin
     #2 b=4'bzzz1;
     #2 b=4'bzzz1;
     #2 b=4'bzz10;
     #2 b=4'bz100;
     #2 b=4'b1000;
end
initial $monitor($time, "input b =%b,a   =%b ",b,a);
initial #40 $stop;
endmodule
```

SIMULATION FLOW

In Verilog the parallel processing is structured through the following [IEEE]

Simulation time: Simulation is carried out in simulation time.

- At every simulation step a number of active events are sequentially carried out.
- The simulator maintains an event queue – called the "Stratified Event Queue" – with an active segment at its top. The top most event in the active segment of the queue is taken up for execution next.
- The active event can be of an update type or evaluation type. The evaluation event can be for evaluation of variables, values on nets, expressions, *etc*. Refreshing the queue and rearranging it constitutes the update event.
- Any updating can call for a subsequent evaluation and *vice versa*.
- Only after all the active events in a time step are executed, the simulation advances to the next time step.

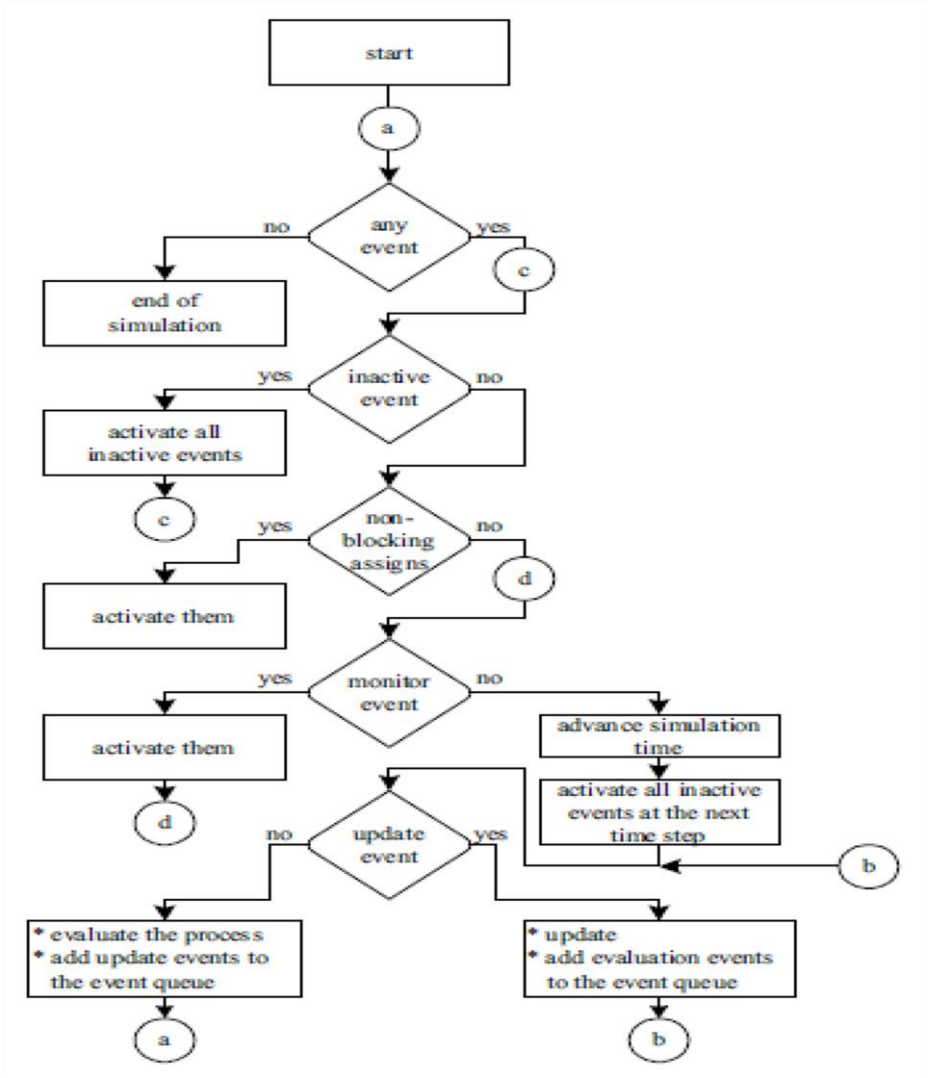
Completion of the sequence of operations above at any time step signifies the parallel nature of the HDL.

Stratified Event Queue

- The events being carried out at any instant give rise to other events – inherent in the execution process. All such events can be grouped into the following 5 types:
  - ***Active events*** –
  - ***Inactive events*** – The inactive events are the events lined up for execution immediately after the execution of the active events. Events specified with zero delay are all inactive events.
  - ***Blocking Assignment Events*** – Operations and processes carried out at previous time steps with results to be updated at the current time step are of this category.

- ***Monitor Events*** – The Monitor events at the current time step –**$monitor** and **$strobe** – are to be processed after the processing of the active events, inactive events, and non blocking assignment events.
- ***Future events*** – Events scheduled to occur at some future simulation time are the future events.

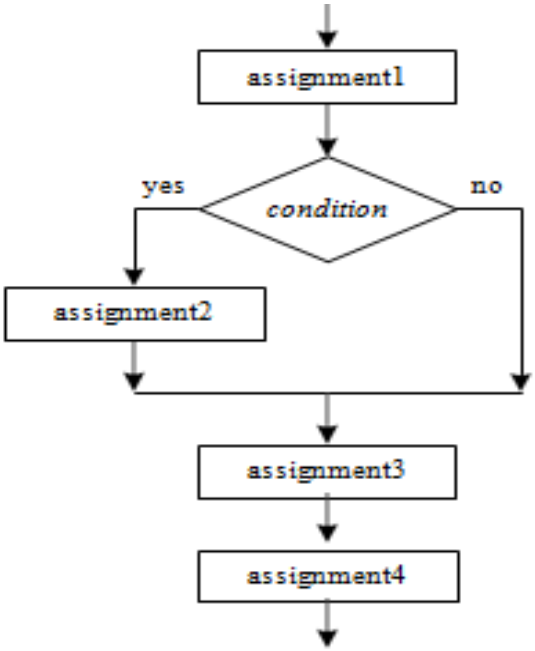
FLOW CHART for SIMULATION FLOW



IF construct:

- The **if** construct checks a specific condition and decides execution based on the result. The following figure shows the structure of a segment of a module with an **if** statement.

```
assignment1;
if (condition) assignment2;
assignment3;
assignment4;
```

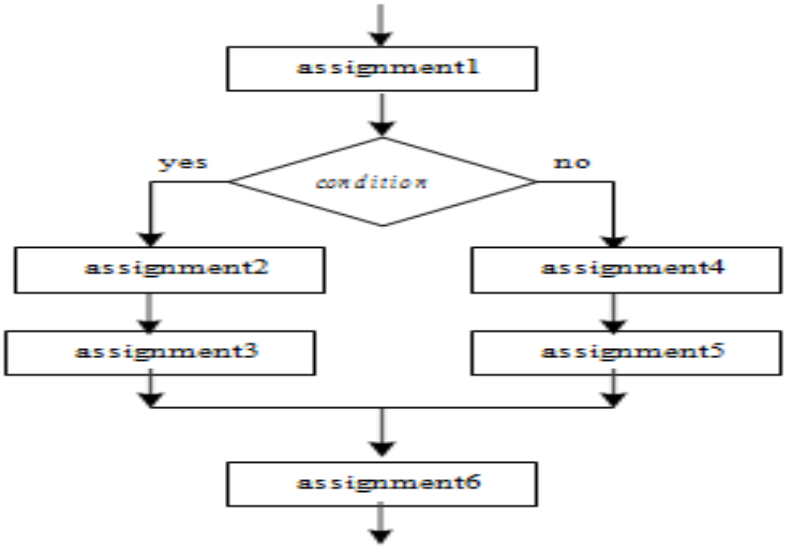


After execution of assignment1, the condition specified is checked. If it is satisfied, assignment 2 is executed; if not, it is skipped. In either case the execution continues through assignment3, assignment4, *etc*. Execution of assignment2 alone is dependent on the condition. The rest of the sequence remains.

IF-ELSE construct

- The **if- else** construct is more common and turns out to be more useful than the **if** construct taken alone.

```
...
assignment1;
if(condition)
        begin           // Altemative 1
                assignment2;
                assignment3;
        end
else
        begin           //altemative 2
                assignment4;
                assignment5;
        end
assignment6;
...
...
```



- The design description has two branches; the alternative taken is decided by the *condition*:
- After the execution of assignment1, if the *condition* is satisfied, alternative1 is followed and assignment2 and assignment3 are executed. Assignment4 and assignment 5 are skipped and execution proceeds with assignment6.
- If the *condition* is not satisfied, assignment2 and assignment3 are skipped and assignment4 and assignment5 are executed. Then execution continues with assignment6.

ASSIGN–DEASSIGN CONSTRUCT

- The assign – deassign constructs allow continuous assignments within a behavioral block.

  always@(posedge clk) a = b;
- At the positive edge of clk the value of b is assigned to a, and a remains frozen at that value until the next positive edge of clk. Changes in b in the interval are ignored.
- As an alternative, consider the block

  always@(posedge clk) assign c = d;

  Here at the positive edge of clk, c is assigned the value of d in a continuous manner; subsequent changes in d are directly reflected as changes in variable c:
-

```
Always
begin
@(posedge clk) assign c = d;
@(negedge clk) deassign c;
end
```

The above block signifies two activities:

1. At the positive edge of clk, c is assigned the value of d in a continuous manner.
2. At the following negative edge of clk, the continuous assignment to c is removed; subsequent changes to d are not passed on to c; it is as though c is electrically disconnected from d.

REPEAT CONSTRUCT: The repeat construct is used to repeat a specified block a specified number of times.

```
…
repeat (a)
begin
        assignment1;
        assignment2;
…
End
…
```

- The quantity **a** can be a number or an expression evaluated to a number.
- The following block is executed "a" times. If "a" evaluates to 0 or x or z, the block is not executed.

FOR LOOP: For loop in Verilog is quite similar to the for loop in C**.** It has four parts; the sequence of execution is as follows:

1. Execute assignment1.
2. Evaluate *expression*.
3. If the *expression* evaluates to the true state (1), carry out statement. Go to step 5.
4. If *expression* evaluates to the false state (0), exit the loop.
5. Execute assignment2. Go to step 2

Structure:

```
. . . .
for(assignment1; expression; assignment 2)
statement;
```

THE DISABLE CONSTRUCT: To break out of a block or loop. The disable statement terminates a named block or task. Control is transferred to the statement immediately following the block. The disable construct is functionally similar to the *break* in C

```
always@(posedge en)
begin:OR_gate
b=1'b0;
for(i=0; i<=3; i=i+1)  if(a[i]==1'b1)
begin  b=1'b1;
disable OR_gate
end
end
```

WHILE LOOP: The Boolean *expression* is evaluated. If it is true, the statements are executed and expression evaluated and checked. If the *expression* evaluates to false, the loop is terminated and the following statement is taken for execution.

```
while(|a)  begin

b=1'b1;

@(posedge clk)  a=a-1'b1;

end

b=1'b0;
```

FOREVER LOOP: Repeated execution of a block in an endless manner is best done with the forever loop (compare with repeat where the repetition is for a fixed number of times).

```
always @(posedge en)

forever#2 clk=~clk;
```

PARALLEL BLOCKS: All the procedural assignments within a begin–end block are executed sequentially. The fork–join block is an alternate one where all the assignments are carried out concurrently (The non-blocking assignments too can be used for the purpose.). One can use a fork- join block within a begin–end block or vice versa.

```
module fk_jn_a;
integer a;
initial
begin
        a=0;
 #1     a=1;
 #2     a=2;
 #3     a=3;
 #4     $stop;
end
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule

//Simulatiom results
# a=0, t=0
# a=1, t=1
# a=2, t=3
# a=3, t=6
```

```
module fk_jn_b;
integer a;
initial
fork
        a=0;
 #1     a=1;
 #2     a=2;
 #3     a=3;
 #4     $stop;
join
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule

//Simulation results
# a=0, t=0
# a=1, t=1
# a=2, t=2
# a=3, t=3
```

FORCE–RELEASE CONSTRUCT

- When debugging a design with a number of instantiations, one may be stuck with an unexpected behavior in a localized area. Tracing the paths of individual signals and debugging the design may prove to be too tedious or difficult.

- In such cases suspect blocks may be isolated, tested, and debugged and *status quo ante* established. The force–release construct is for such a localized isolation for a limited period.

force a = 1'b0;

forces the variable a to take the value 0.

force b = c&d;

forces the variable b to the value obtained by evaluating the expression c&d.

EVENT

- The keyword **event** allows an abstract event to be declared. The event is not a data type with any specific values; it is not a variable (reg) or a net.
- It signifies a change that can be used as a trigger to communicate between modules or to synchronize events in different modules.
- The operator "à" signifies the triggering. Subsequently, another
- activity can be started in the module by the event change.

. . .
event change;

. . .

always

. . .

. . . ⇨ change;

. . . always@change

Example: A segment of a design using the **if** constructs. It is a ring counter, which shifts one bit right at every clock pulse. The shift operation shifts the a byte right by one bit and fills the vacated bit – a[7] – with a zero. It is set to 1 if the bit shifted out last – a[0] – was a 1. The same is carried out through the **if** statement.

```
Reg[7:0] a;
Reg  c;
always@(posedge clk)
begin
        c = a[0];
        a = a>>1'b1; // Since the vacated bit of a is filled with a zero, it need be
        if( c ) a[7] = c;// set only if a[0] =1
end
```

Ring counter description using the if construct

The use of **case** statement to realize mux, demux, direct encoders, and decoders makes the design description simple and direct – in contrast to the use of **if-else-if** construct.

Example: 2 to 4 demux module

```
module demux(a,b,s);
output [3:0]a;
input b;
input[1:0]s;
reg[3:0]a;
always@(b or s)
begin
        if(s==2'b00)
        begin
                a[2'b0]=b;
                a[3:1]=3'bZZZ;
        end
        else if(s==2'b01)
        begin
                a[2'd1]=b;
                {a[3],a[2],a[0]}=3'bZZZ;
        end
else if(s==2'b10)
        begin
                a[2'd2]=b;
                {a[3],a[1],a[0]}=3'bZZZ;
```

```
          end
else
          begin
                    a[2'd3]=b;
                    a[2:0]=3'bZZZ;
          end
end
endmodule
```