INTRODUCTION

Behavioral level modeling constitutes design description at an abstract level. One can visualize the circuit in terms of its key modular functions and their behavior; 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 behavioral level is done through a sequence of assignments. These are called ‗procedural assignments' The procedure assignment is characterized by the following:

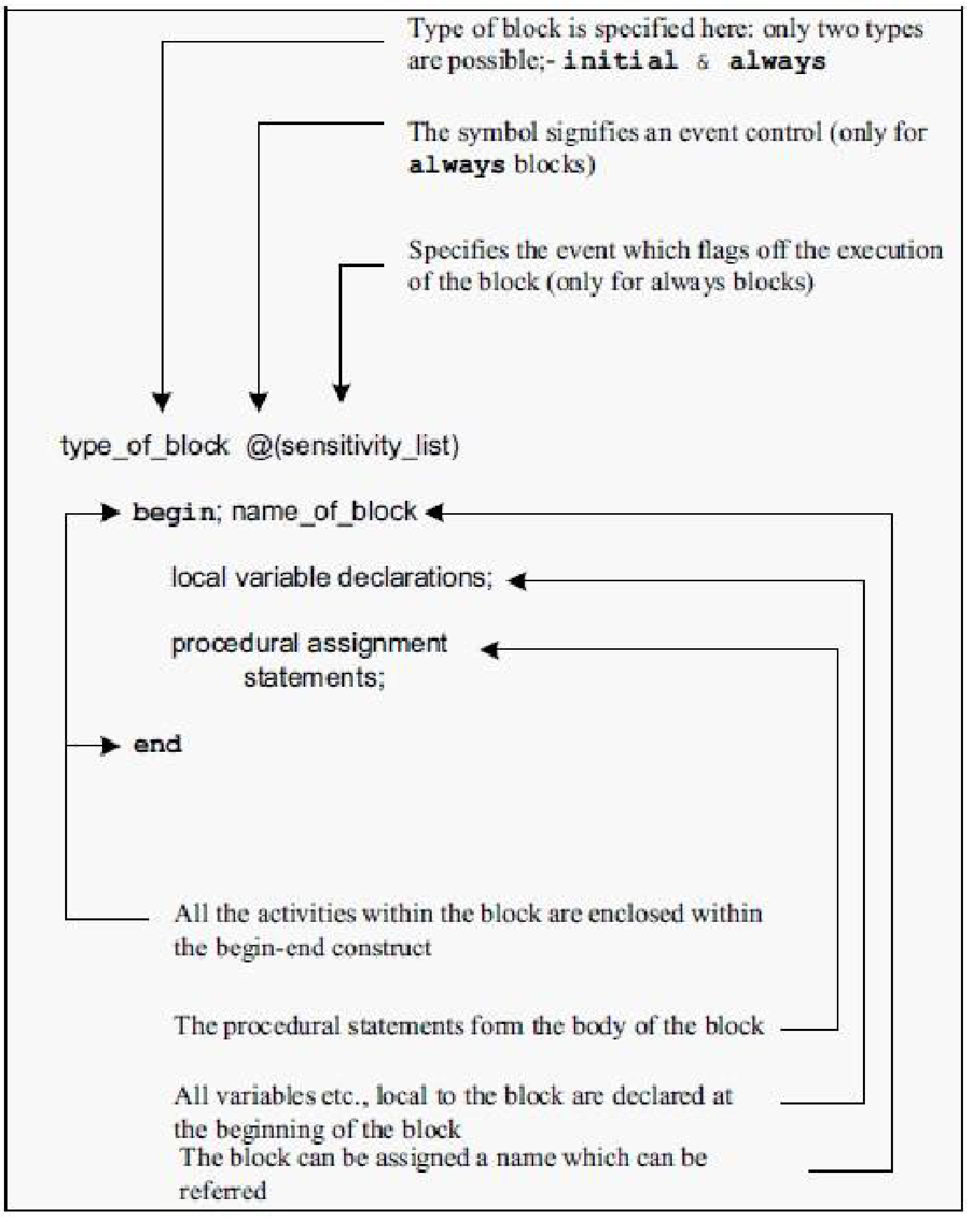
- The assignment is done through the "=‖"symbol (or the "<="symbol) as was the case with the continuous assignment earlier.
- An operation is carried out and the result assigned through the "=" operator to an operand specified on the left side of the "="sign – for example, *N* = ~*N*; Here the content of reg *N* is complemented and assigned to the reg *N* itself. The assignment is essentially an updating activity.
- 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.
- All the operands are given in Tables 6.1 to 6.9. The format of using them and the rules of precedence remain the same.
- The operands on the right side can be of the net or variable type. They can be scalars or vectors.
- 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, realtime, or memory type of data – declared in advance.
- 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.
- Procedural assignments are very much like sequential statements in C. Normally they are carried out one at a time sequentially. As soon as a specified operation on the right is carried out, the result is assigned to the quantity on the left – for example *N* = *m* + *l*; *N*1 = *N* * *N*;The above form a set of two procedures placed within an always block. Generally they are carried out sequentially in the order specified

The sequential nature of the assignments requires the operands on the left of the assignment to be of reg (variable) type.

FUNCTIONAL BIFURCATION

Design description at the behavioral level is done in terms of procedures of two types; one involves functional description and interlinks of functional units. It is carried out through a series of blocks under an ―always The second concerns simulation – its starting point, steering the simulation flow, observing the process variables, and stopping of the simulation process; all these can be carried out under the ―always‖ banner, an ―initial banner, or their combinations. However, each always and each initial block initiates an activity flow during simulation In general the activity with all such blocks starts at the simulation time and flows concurrently during the whole simulation process

A procedure-block of either type – **initial** or


Type of block is specified here: only two types are possible;- initial & always
The symbol signifies an event control (only for always blocks)
Specifies the event which flags off the execution of the block (only for always blocks)
type_of_block @(sensitivity_list)
begin; name_of_block
local variable declarations;
procedural assignment statements;
end
All the activities within the block are enclosed within the begin-end construct
The procedural statements form the body of the block
All variables etc., local to the block are declared at the beginning of the block
The block can be assigned a name which can be referred

begin – end Construct

If a procedural block has only one assignment to be carried out, it can be specified as below

initial #2 a=0;
If 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. The following are to be noted here

- Every **begin** declaration must have its associated **end** declaration.
- **begin** – **end** constructs can be nested as many times as desired.

Name of the Block

Any block can be assigned a name, but it is not mandatory. Only the blocks which are to be identified and referred by the simulator need be named. 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

Local Variables

Variables used exclusively within a block can be declared within it. Such a variable need not be declared outside, in the module encompassing the block. Such local declarations conserve memory and offer other benefits too. Regs declared and used within a block are static by nature. They retain their values at the time of leaving the block. The values are modified only at the next entry to the block.

INITIAL CONSTRUCT

A set of procedural assignments within an **initial** construct are executed only once – and, that too, at the times specified for the respective assignments The **initial** process is characterized by the following

- 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 (**reg**, **integer**, or **real** type of variable) or a net.
- All the procedural assignments appear within a **begin–end** block
- All the procedural assignments are executed sequentially – in the same order as they appear in the design description. The **initial** block above does three controlling activities during the simulation run.
- Initialize the selected set of **reg**'s at the start.
- Change values of **reg**'s at predetermined instances of time. These form the inputs to the module(s) under test and test it for a desired test sequence.
- Stop simulation at the specified time

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.

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.

The process can have one assignment statement or multiple assignment statements. In the latter case all the assignments are grouped together within a ―**begin** – **end** construct.

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 the **reg** (variable) clk.
@(**posedge** clk) : executes the following block at the positive edge of the **reg** (variable) clk. @clk : executes the following block at both the edges of clk.

- The events can be changes in **reg**, **integer**, **real** or a signal on a net. These should be declared beforehand.
- No algebra or logic operation is permitted as an event. The OR'ing signifies —execute the block if any one of the events takes place.
- The positive transition for a reg type single bit variable is a change from 0 to1.
- For a logic variable it is a transition from false to true.

The **posedge** transition for a signal on a net can be of three different types:

- 0 to1
- 0 to x or z
- x or z to 1

The **negedge** transition for a signal on a net can be of three different types:-

- 1 to 0
- 1 to x or z
- x or z to 0

If the event specified is in terms of a multibit **reg**, only its least significant bit is considered for the transition. Changes in the other bits are ignored. The event-based flagging-off of a block is applicable only to the **always** block. According to the recent version of the LRM, the comma operator (,) plays the same role as the keyword **or**. The two can be used interchangeably or in a mixed form. Thus the following are identical: @ (a **or** b **or** c) @ (a **or** b, c) @ (a, b, c) @ (a, b **or** c)

EXAMPLES

Versatile 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
TEST_BENCH
module tst_counterup;
reg clk;
reg[3:0]N;
 wire[3:0]a;
counterup c1(a,clk,N);
 initial
begin
clk = 0;
N = 4'b1011;
end
always #2 clk=~clk;
 initial $monitor($time,"a=%b,clk=%b,N=%b",a,clk,N);
endmodule


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 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 clrupdcou(a,clr,clk,N,u_d);

input clr,clk,u_d;

input[3:0]N;

output[3:0]a;

 reg[3:0]a;

 initial a =4'b0000;

always@(negedge clk or posedge clr) a = (clr) ? 4'h0 : ( (u_d) ? ( (a==N) ? 4'b0000 : a+1'b1)
 :( (a == 4'b0000) ? N : a - 1'b1));

endmodule
```

Example

Shift Register 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
```

ASSIGNMENTS WITH DELAYS

The delay refers to the specific activity it qualifies. A variety of possibilities of specifying delays to assignments exist. 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 assignment is scheduled to be repeated every 3 ns, irrespective of whether a changes in the Meantime

Intra-assignment Delays

In contrast, the intra-assignment‖ delay carries out the assignment in two parts

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 a after a delay specified by dl. dl can be an integer or a constant expression

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. Often it is used to avoid indecision in the precedence of execution of assignments

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.
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

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
```

BLOCKING AND NONBLOCKING ASSIGNMENTS

These are executed sequentially – that is, one statement is executed, and only then the following one is executed. Such assignments block the execution of the following lot of assignments at any time step. Hence they are called ―blocking assignments‖. A facility called the ―nonblocking assignment is available for such situations. The symbol ―<=‖ signifies a non-blocking assignment. The same symbol signifies the ―less than or equal to‖ operator in the context of an operation. The context decides the role of the symbol. The main characteristic of a nonblocking assignment is that its execution is concurrent with that of the following assignment or activity.

Nonblocking Assignments and Delays

Delays – of the assignment type and the intra-assignment type – can be associated with nonblocking assignments also. The principle of their operation is similar to that with blocking assignments.

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. Format of the **case** construct is

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

If the evaluated value matches ref1, statement1 is executed; and the simulator exits the block; 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. A statement or a group of statements is executed if and only if there is an exact – bit by bit – match between the evaluated expression and the specified ref1, ref2, *etc*.

For any of the matches, one can have a block of statements defined for execution. The block should appear within the **begin-end** construct.

There can be only one **default** statement or **default** block. It can appearanywhere in the case statement.

One can have multiple signal combination values specified for the same statement for execution. Commas separate all of them.

```
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
```

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

```
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
```

SIMULATION FLOW

Verilog has to be an inherently parallel processing language. The fact that all the elements of a digital circuit (or any electronic circuit for that matter) function and interact continuously conforming to their interconnections demands parallel processing. In Verilog theparallel processing is structured through the following [IEEE]: Simulation time: Simulation is carried out in simulation time. The simulator functions with simulation time advancing in (equal) discrete steps.

At every simulation step a number of active events are sequentially carriedout.

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. A number of active events can be present for execution at any simulation time step; all may vie for ―attention.‖ Amongst these, an event specified at #0 time is scheduled for execution at the end

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 – explained above.

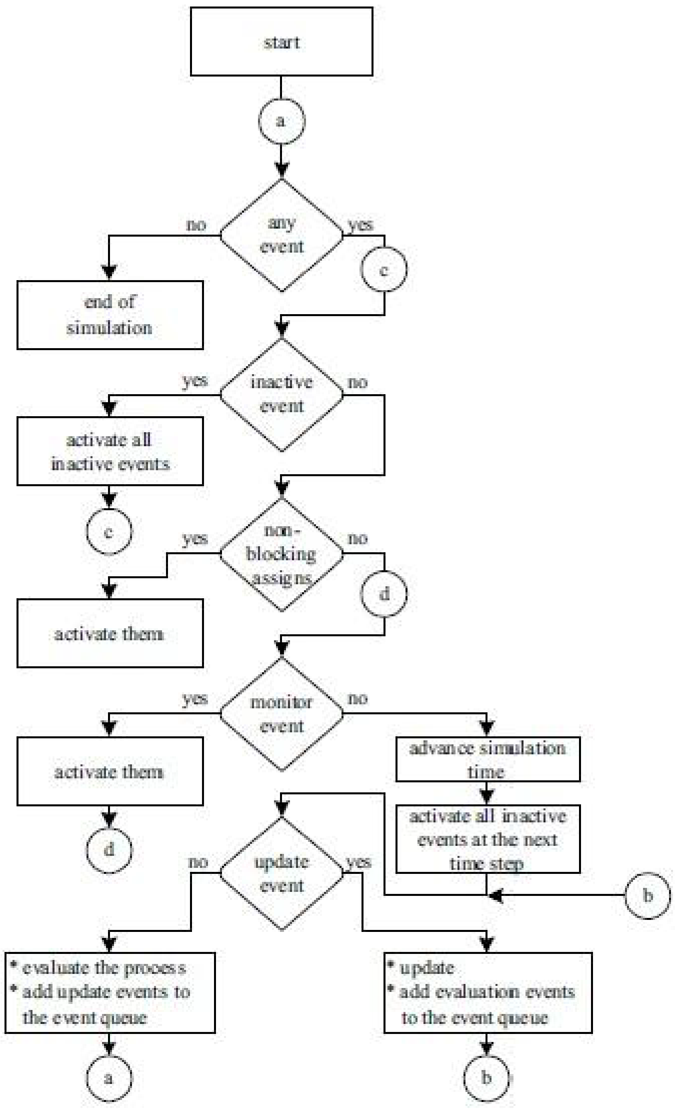
Inactive events – The inactive events are the events lined up for execution immediately after the execution of the active events. Events specified with zero delay are all inactive events.

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 nonblocking assignment events.

Future events – Events scheduled to occur at some future simulation time are the future events.

The simulation process conforming to the stratified event queue is shown in flowchart form in Figure



if AND if-else CONSTRUCTS

The **if** construct checks a specific condition and decides execution based on the result. the structure of a segment of a module with an **if** statement. After execution of assignment1, the condition specified is checked. If it is satisfied, assignment2 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. . . .

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

Use of the if–else construct

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

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

```
 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
```

assign−deassign CONSTRUCT

The **assign** – **deassign** constructs allow continuous assignments within a behavioral block. **always**@(**posedge** clk) a = b; By way of execution, at the positive edge of clk the value of b is assigned to variable a, and a remains frozen at that value until the next positive edge of clk. Changes in b in the interval are ignored. 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: The assignment here is akin to a direct (one way) electrical connection to c from d established at the positive edge of clk. Consider an enhanced version of the above block as

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.

In short, assign allows a variable or a net change in the sensitivity list to mandate a subsequent continuous assignment within. **deassign** terminates the assignment done through the **assign**-based statement.

```
 module demux1(a0,a1,a2,a3,b,s);
 output a0,a1,a2,a3;
 input b;
input [1:0]s;
 reg a0,a1,a2,a3;
always@(s) if(s==2'b00)
 assign {a0,a1,a2,a3}={b,3'oz};

else if(s==2'b01)
```

```
 assign {a0,a1,a2,a3}={1'bz,b,2'bz};
 else if(s==2'b10)
 assign {a0,a1,a2,a3}={2'bz,b,1'bz};
 else if(s==2'b11)
assign {a0,a1,a2,a3}={3'oz,b};
endmodule
```

D Flip-Flop through assign – deassign Constructs

```
 module dffassign(q,qb,di,clk,clr,pr);
 output q,qb;
input di,clk,clr,pr;
reg q;
assign qb=~q;
always@(clr or pr)
 begin
 if(clr)
assign q = 1'b0;//asynchronous clear
and if(pr)
assign q = 1'b1;// preset of FF overrides
 else
deassign q;// the synchronous behaviour

end
 always@(posedge clk) q = di;//synchronous (clocked)value assigned to q
endmodule
```

repeat CONSTRUCT

The repeat construct is used to repeat a specified block a specified number of times. The quantity a can be a number or an expression evaluated to a number. As soon as the repeat statement is encountered, a is evaluated. The following block is executed ―a‖ times. If ―a‖ evaluates to 0 or **x** or **z**, the block is not executed. Structure of a **repeat** block. …

```
repeat (a)
 begin assignment1;
 assignment2;
end
```

… A module to illustrate the use of the
repeat construct

```
module trial_8b;
 reg[7:0] m[15:0];
 integer i;
 reg clk;
 always begin repeat(8) begin @(negedge clk) m[i]=i*8;
 i=i+1;
end
repeat(8)
 begin @(negedge clk) i=i-1;
$display("t=%0d, i=%0d, m[i]=%0d", $time,i,m[i]);
end
```

```
end
initial
 begin
 clk = initial
begin clk = 1'b0;
 i=0; #70 $stop;
end
 always #2 clk=~clk;
endmodule
```

for LOOP

The **for** loop in Verilog is quite similar to the **for** loop in C; the format of the **for** loop is . . . . **for**(assignment1; *expression*; assignment 2) statement; . . . 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.

An adder module using the **for** loop.

```
module addfor(s,co,a,b,cin,en);
 output[7:0]s;
 output co;
 input[7:0]a,b;
input en,cin;
reg[8:0]c;
reg co;
reg[7:0]s;
always@( posedge en ) begin c[0] =cin;
for(i=0;i<=7;i=i+1)
begin {c[i+1],s[i]}=(a[i]+b[i]+c[i]);
 end
co=c[8];
end
endmodule
```

THE disable CONSTRUCT

There can be situations where one has 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. Conditional termination of a loop, interrupt servicing, *etc*., are typical contexts for its use. Often the disabling is carried out from within the block itself. The **disable** construct is functionally similar to the *break* in C OR gate module to demonstrate the use of the
disable construct

```
module or_gate(b,a,en);
```

```
 input [3:0]a;
 input en;
output b;
reg b;
 integer i;
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
 endmodule
```

The **disable** statement has to have a block (or task) identifier tagged to it in this respect it differs from ―*break*‖ in C.

Once encountered, it terminates execution of the block; the following statements within the block are not executed.

Typically it can be used to handle exceptions to regularly assigned activities for example, Interrupt, Hold, Reset, *etc*.

while LOOP

The format for the while loop is shown is **while** (*expression*) statement ;

The Boolean *expression* is evaluated. If it is **true**, the statement (or block of statements) is executed and expression evaluated and checked. If the *expression* evaluates to **false**, the loop is terminated and the following statement is taken for execution.

If the *expression* evaluates to **true**, execution of statement (block of statements) is repeated. Thus the loop is terminated and broken only if the *expression* evaluates to false.

To generates a pulse of definite width

```
module while2(b,n,en,clk);
```

```
input[7:0]n;
```

```
input clk,en;
```

output b;

```
reg[7:0]a;
```

```
reg b;
```

```
always@(posedge en)
```

```
begin a=n;
```

```
while(|a)
```

```
begin b=1'b1;
```

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

end

```
b=1'b0;
```

end

```
initial b=1'b0;
```

```
endmodule
```

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).

module to generate a clock waveform using the **forever** construct

```
 module clk;
reg clk, en;
 always @(posedge en)
 forever#2 clk=~clk;
initial
begin
 clk=1'b0;
 en=1'b0;
#1 clk=1'b1;
 #4 en=1'b1;
#30 $stop;
end
initial $monitor("clk=%b, t=%0d, en=%b ", clk,$time,en);
 endmodule
```

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. One can use a fork-join block within a **begin–end** block or vice versa.

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. The **force–release** construct is similar to the **assign–deassign** construct. The latter construct is for conditional assignment in a design description. The **force–release** construct is for ―short time‖ assignments in a test-bench. Synthesis tools will not support the **force–release** constructs.

The **force–release** construct is equally valid for net-type variables and **reg**-type variables. The net–type variables revert to their normal values on release. With **reg**-type variables the value forced remains until another assignment to the reg.

The variable, on which the values are forced during testing, must be properly dereferenced.

In the illustration above, each variable was forced one at a time. It was done only to simplify the illustration sequence and focus attention on the possible use of the construct. In practice, different variables can be forced together before the special debug sequence. Their release too can be together.

OR gate module and its test bench to illustrate the use of **force–release** construct

```
module or_fr_rl(a,b,c);
input b,c;
 output a;
 wire a,b,c;
assign a= b|c;
initial
 begin
#1 $display("display:time=%0d, b=%b, c=%b, a=%b", $time,b,c,a);
#6 force b=1'b1;
 #1 $display("display:time=%0d, b=%b, c=%b, a=%b", $time,b,c,a);
#6 release b;
```

```
#1 $display("display:time=%0d, b=%b, c=%b, a=%b", $time,b,c,a);
```

End

```
 endmodule
```

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. . . .

```
 event change; . . .
always
. . . . . .
 change; . . .
.always@change . .
```

. In the course of execution of an **always** block, the event is triggered. The operator signifies the triggering. Subsequently, another activity can be started in the module by the event change. The **always**@(change) block activates this. The event change can be used in other modules also by proper dereferencing; with such usage an activity in a module can be synchronized to an event in another module. The **event** construct is quite useful, especially in the early stages of a design. It can be used to establish the functionality of a design at the behavioral level; it allows communication amongst different instantiated modules without associated inputs or outputs. A module to illustrate the **event** construct:

A serial data receiver

```
 module rec(a,ddi,clk);
 output[8:1]a;
 input ddi,clk;
reg[8:1] a;
integer j,jj;
event buf_ful;
 always
for (j=0;j<20;j=j+1)
begin
#0 jj=0;

repeat(8)@(negedge clk)
 begin jj=jj+1;
 a[jj]=ddi;
//$display("b=%b",a[jj]);
```

```
 end
#0 ->buf_ful;
 end
endmodule
```