Verilog Data flow modeling

Introduction: For small circuits, the gate-level modeling works very well due to limited number of gates, and the designer can instantiate and connect every gate individually. However, in complex designs the number of gates is very large and gate level design is not efficient. Verilog Dataflow modeling provides a powerful way to implement a design at a higher level of abstraction than gate level. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data.

Features of data Flow Modeling:

- Data flow modeling uses continuous assignment statement. Continuous assignment are made using verilog keyword ***assign***
- Powerful design technique that allows designers to concentrate on optimizing the circuit in terms of data flow.
- Logic synthesis tools can be used to create a gate level netlist
- RTL (register transfer level) is a combination of dataflow and behavioral modeling

Verilog Data flow modeling

Continuous assignment statement(synthesizable): The *assign* statement is used to make continuous assignment in the dataflow modeling. In simplest form syntax is:

Syntax: ***assign #<delay> net = expression;***

- The LHS of assign statement must always be a scalar or vector net (declared using verilog keyword *wire*) or a concatenation. It cannot be a register.
- Registers or nets or function calls can come in the RHS of the assignment.
- Continuous statements are always active. The assignment expression is evaluated as soon as one of the RHS operands changes and the value is assigned to the LHS net.
- Delays(optional) can be specified. Delay values are used to control the time when a net is assigned the evaluated value.It is very useful in modeling timing behavior in real circuits.

Verilog Data flow modeling

Important Points to Remember:

- Continuous assignments cannot be used within procedural blocks (procedural blocks uses procedural assignment like **always** and **initial**).
- LHS of a continuous assignment must be a net data type(can not be register)
- A wire should be assigned only once using continuous assignment.
- An assign statement is used for modeling only combinational logic and it is executed continuously.
- Continuous assignment overrides any procedural assignments
- Commonly used to model tri-state buffers **(assign out = (enable) ? data : 1'bz;)**

Verilog Data flow modeling

Examples of Continuous assignment:
// out is a scalar net (a and b are also scalar nets).

assign out = a + b; // a + b is evaluated and assigned to out

assign out = sel ? Input1 : Input0; // 2:1 Mux implementation

//Continuous assignment with vector nets

assign data[15:0] = data1_bits[15:0] ^ data2_bits[15:0]

//Continuous assignment with concatenation

assign {C_OUT, sum[7:0]} = A[7:0] + B[7:0] + C_IN; // full adder

reg y; // y declared as reg data type
assign y = Input1 ^ Input2; // LHS of Assignment y can not be a register

Verilog Data flow modeling

Regular & Implicit Continuous Assignment:

Regular continuous assignment means, the declaration of a net and its continuous assignments are done in two different statements. But in implicit assignment, continuous assignment can be done on a net when it is declared itself. In the below example, valid is declared as wire during the assignment. If signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred. In the below code dout is not declared as net, but it is inferred during assignment.

```
// Regular Continuous assignment

wire out; // declaration of net

assign out = in1 & in2; //continuous assignment
```

```
// Implicit Continuous assignment

// same result with implicit assignment
wire out = in1 & in2;
```

Verilog Data flow modeling

```
//Mux1 implementation using Data Flow
module mux4_1( sel, d, q );
input[1:0] sel;
input[3:0] d;
output q;
wire q;
wire[1:0] sel;
wire[3:0] d;
assign q =  ( sel == 0 ) ? d[0]
          : ( sel == 1 ) ? d[1]
          : ( sel == 2 ) ? d[2]
          :  d[3];
//another way to write
//assign q = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```

```
//Same implementation different logic

module mux4_1( sel, d, q );

input[1:0] sel;
input[3:0] d;
output    q;

wire       q;
wire[1:0] sel;
wire[3:0] d;

assign q = d[sel];

endmodule
```

Verilog Data flow modeling

Delays(Non-Synthesizable): Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. there are three ways of specifying delays in continuous assignment statements:

- Regular assignment Delay
- Implicit continuous assignment delay
- Net declaration delay

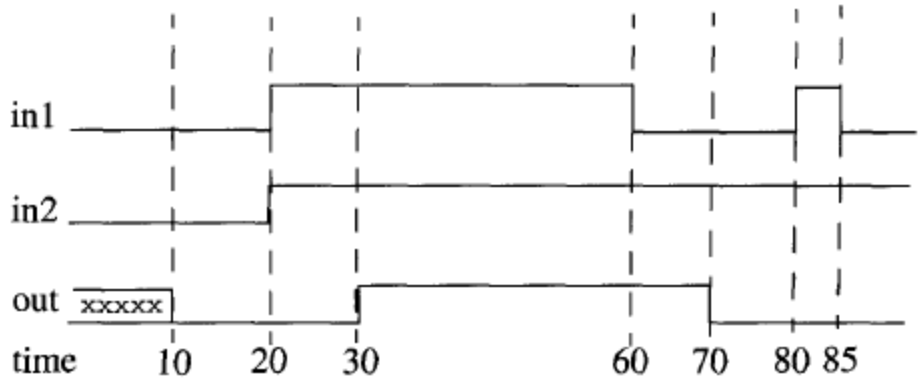
Regular assignment Delay: The delay value is specified after the keyword assign.

/* If there is any change in the operands in the RHS (in1 or in2), then RHS expression will be evaluated after 10 units of time. Let's say at time t, if there is change in one of the operands in the example below, then the expression is calculated at t+10 units of time and result will be assigned to out */

```
assign #10 out = in1 & in2; // Delay in a continuous assignment
```

Verilog Data flow modeling

Example1: Simulated output from **assign #10 out = in1 & in2;**



- When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
- When in1 goes low at 60, out changes to low at 70. However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed. Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.

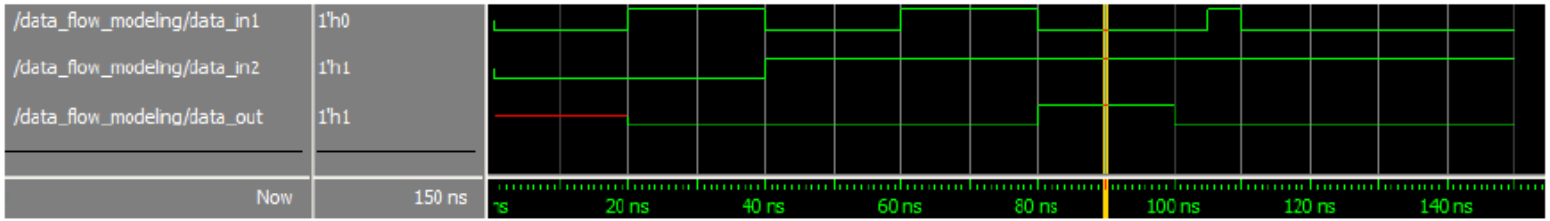
Data flow modeling: assign #20 data_out = data_in1 & data_in2;

```
module data_flow_modeling;
reg data_in1, data_in2; // inputs defined as reg
wire data_out; // output must be wire in a assignment statement
assign #20 data_out = data_in1 & data_in2; //continuous assignment statement
initial
  begin
    $monitor($time," data_in1=%b data_in2=%b data_out=%b",data_in1, data_in2, data_out);
    data_in1 = 1'b0; data_in2 = 1'b0;
    #20 data_in1 = 1'b1; data_in2 = 1'b0;
    #20 data_in1 = 1'b0; data_in2 = 1'b1;
    #20 data_in1 = 1'b1; data_in2 = 1'b1;
    #20 data_in1 = 1'b0; data_in2 = 1'b1;
    #25 data_in1 = 1'b1; data_in2 = 1'b1;
    #5 data_in1 = 1'b0; data_in2 = 1'b1;
    #40 $stop;
  end
endmodule
```

assign #20 data_out = data_in1 & data_in2;

Simulation output

```
#                0 data_in1=0 data_in2=0 data_out=x
#               20 data_in1=1 data_in2=0 data_out=0
#               40 data_in1=0 data_in2=1 data_out=0
#               60 data_in1=1 data_in2=1 data_out=0
#               80 data_in1=0 data_in2=1 data_out=1
#              100 data_in1=0 data_in2=1 data_out=0
#              105 data_in1=1 data_in2=1 data_out=0
#              110 data_in1=0 data_in2=1 data_out=0
```



Verilog Data flow modeling

Implicit assignment Delay:

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

```
// implicit continuous assignment delay

wire #10 out = in1 | in2;

// is same as

wire out;

assign #10 out = in1 | in1;
```

Verilog Data flow modeling

Net assignment Delay:

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly.

```
//Net Delays
wire # 10 out;
assign out = in1 & in2;

//The above statement has the same effect as the following
wire out;
assign #10 out = in1 & in2;
```

Behavioral modeling

Introduction: Behavioral modeling represents the behavior of the logic at a very high level of abstraction. Behavioral constructs provide the designer with a great amount of flexibility to design efficiently. The term RTL (logic is modeled at register level) is commonly used for a combination of dataflow and behavioral modeling.

Procedural statements: There are two basic types of procedural statements in Verilog: always and initial. All other behavioral statements can appear only inside these structured procedure statements.

initial *statement(Non-synthesizable)**:***

All statements inside an initial statement constitute an initial block. An initial block starts at time 0, executes exactly once during a simulation. If there are multiple initial blocks, each block starts to execute concurrently at time 0 and each block finishes execution independently of other blocks. Multiple behavioral statements must be grouped, typically using the keywords begin and end.

Behavioral modeling

```
module initial_example();
reg clk,reset,enable,data;
initial
    clk = 1'b0; // single statement does not need to be grouped
initial
    begin // multiple statement need to be grouped
        reset = 1'b0;
        enable = 1'b0;
        #10 data = 1'b0; // executed after delay of 10 time units
    end
initial
    #100 $finish; //simulation finished at #100 time units
endmodule
```

Note: In the above example, the three initial statements start to execute in parallel at time 0 or executed after delay time units after the current simulation time.

Behavioral modeling

Points to remember regarding initial statement:

- initial can use any of the constructs used in always
- initial blocks are only for implementing testbench code
- The initial blocks are typically used for initialization, monitoring and other processes that must be executed only once during the entire simulation run.
- initial blocks are ignored by synthesis
- initial executes only once, but not necessarily first
- Within the initial block statements are executed sequentially
- It can execute in zero time, or be timed with delays(#delay value).
- If there is more than one initial block,then all the initial blocks are executed concurrently.

Behavioral modeling

always statement(Synthesizable): All statements inside an always statement constitute a always block. An always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion. This statement is used to model a block of activity that is repeated continuously in a digital circuit. The always statement is structured like this:

```
always @ (sensitivity list)
    begin
        <procedural statements>
    end
```

Example:

```
module clk_gen;
reg clk;
initial
    clk = 1'b0; //Initialize clock at time zero
    always #20 clk = ~clk; //toggle clk every half-cycle (time period = 40)
initial
    #400 $finish; //finish simulation at 400 time units
endmodule
```

Behavioral modeling

Points to remember regarding always statement:

- Whenever a variable in the sensitivity list of always block changes, the always block wakes up and executes its enclosed statements. This process of waiting and executing on event is repeated till simulation stops.
- If variables are omitted from the sensitivity list, the always block will not wake up and execute the statements.
- Within the always block, statements are executed sequentially.
- Variables on the LHS, inside the always block must be of type reg. Procedural assignment statements assign values to reg, integer, real, or time variables and can not assign values to nets (wire data types).
- Reg variables retain only the last assigned values within always block.

Behavioral modeling

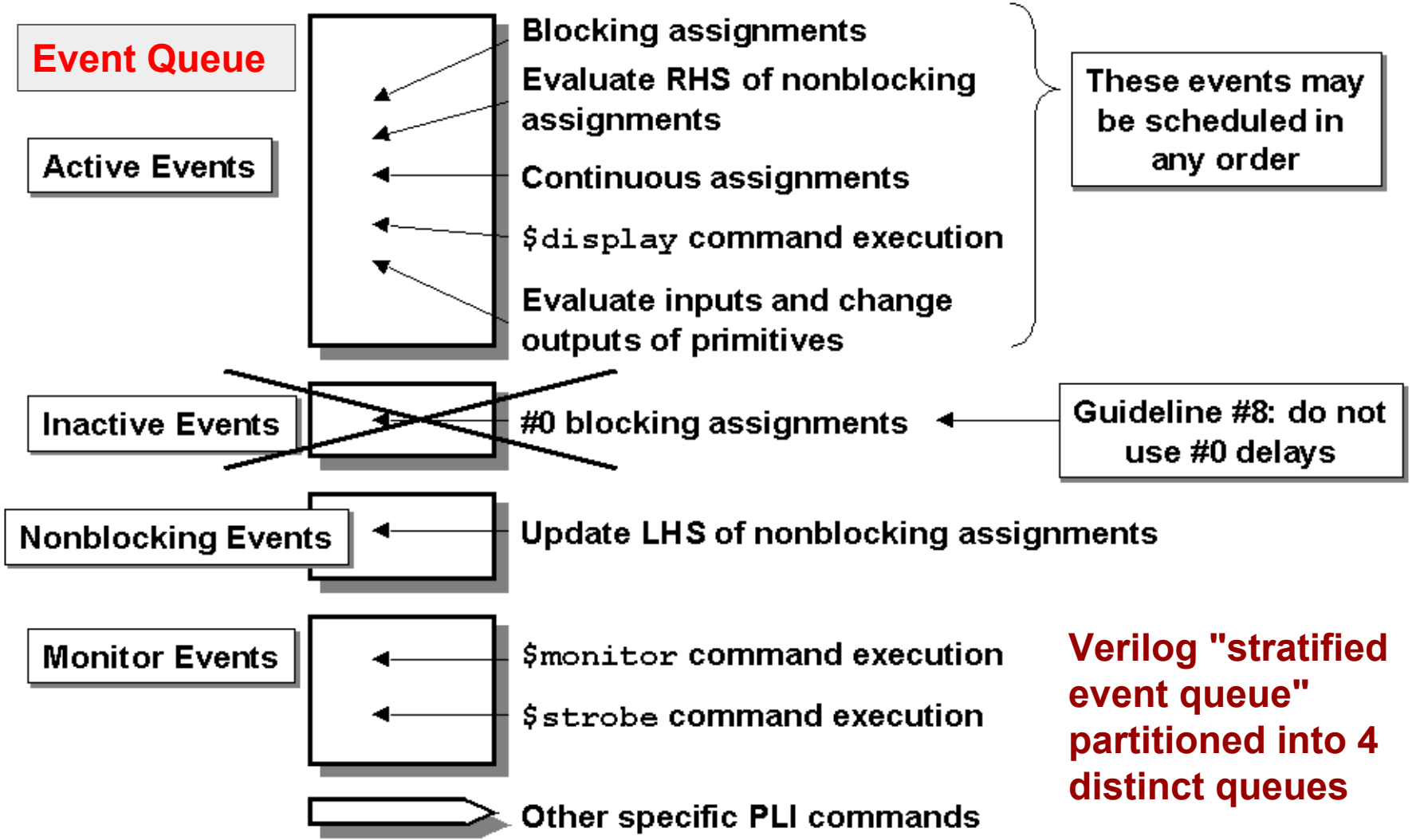
Example:

```
//complete sensitivity list
always @(b, c, d)
        begin
                a = ( b & c);
                e = ( c | d);
        end
```

```
//Incomplete sensitivity lists
always @(a) // b missing from sensitivity list
        f = a & b;

always // a, b missing from sensitivity list
        f = a & b; // a and b missing
```

Solution: Use always@(*) for combinational logic

In the above example, whenever there is a change in b, c, or d the always block will be executed. Here the list b, c, and d is called the *sensitivity list*. we can replace always @(b,c,d) with always @(*), it is equivalent to include all input signals, used in the always block. This is very useful when always blocks is used for implementing the combinational logic.


Event Queue
Active Events
Blocking assignments
Evaluate RHS of nonblocking assignments
Continuous assignments
$display command execution
Evaluate inputs and change outputs of primitives
These events may be scheduled in any order
Inactive Events
#0 blocking assignments
Guideline #8: do not use #0 delays
Nonblocking Events
Update LHS of nonblocking assignments
Monitor Events
$monitor command execution
$strobe command execution
Other specific PLI commands
Verilog "stratified event queue" partitioned into 4 distinct queues

Event scheduling and re-triggering

Active events Region: The active events queue is where most Verilog events are scheduled.The "stratified event queue" is logically partitioned into four distinct queues for the current simulation time and additional queues for future simulation times.

- Execute all module blocking assignments.
- Evaluate the Right-Hand-Side (RHS) of all nonblocking assignments and schedule updates into the NBA (Non Blocking Assignment)region.
- Execute all module continuous assignments
- Evaluate inputs and update outputs of Verilog primitives.
- Execute the $display and $finish commands.

Inactive Events Region: This region is where #0 blocking assignments are scheduled. Most engineers that continue to use #0 assignments are trying to defeat a race condition that might exist in their code due to assignments made to the same variable from more than one always block.

Note: Engineers should not make #0 (Zero-delay)RTL procedural assignments.

Event scheduling and re-triggering

Nonblocking Assignment Events Region (NBA): The LHS variables of nonblocking assignments are not updated in the active events queue but instead are placed in the nonblocking assign update events queue(NBA queue), where they remain until they are activated (moved into the active events queue).The principal function of this region is to execute the updates to the Left-Hand-Side (LHS) variables that were scheduled in the Active region for all currently executing nonblocking assignments.

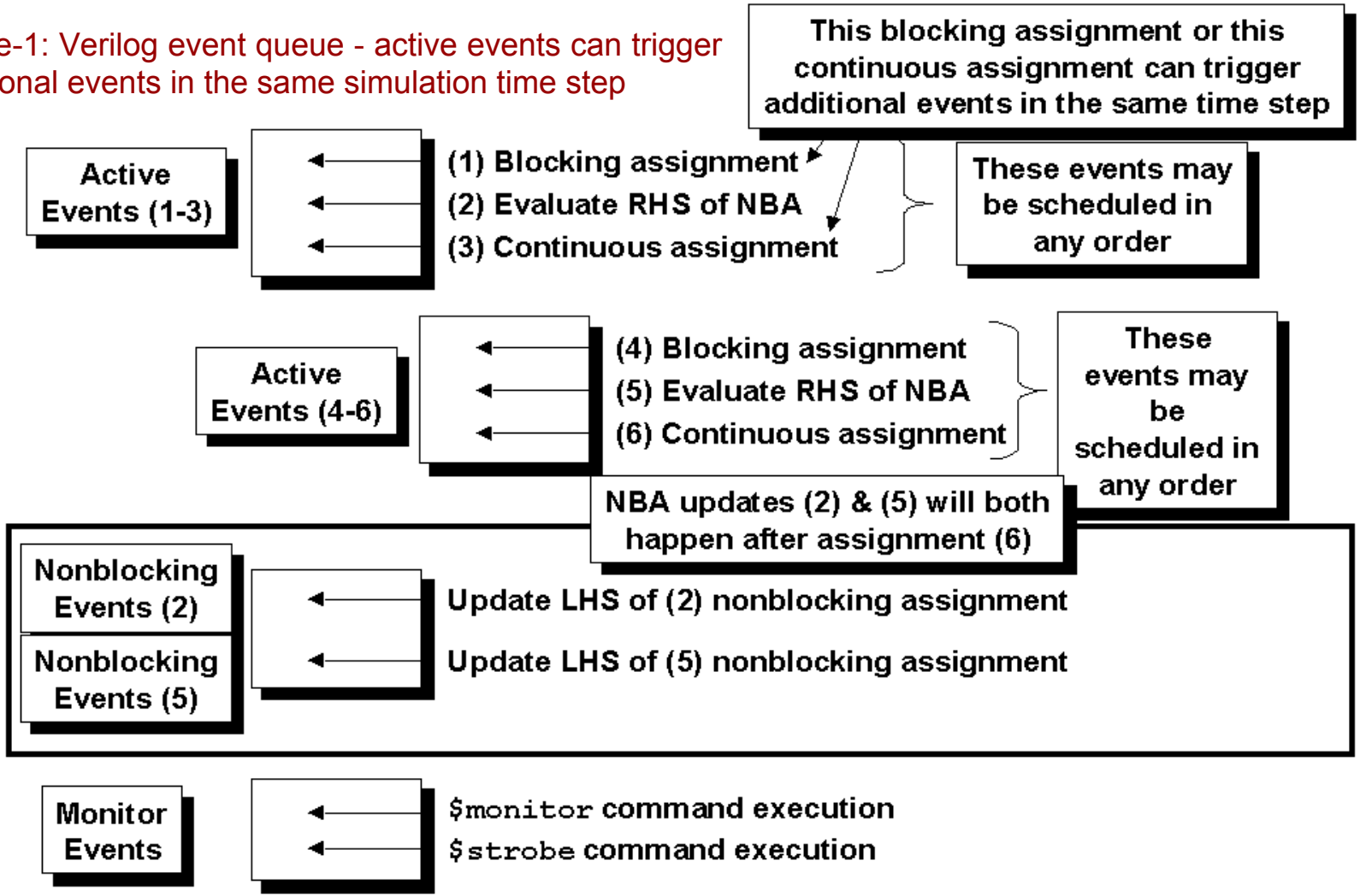
Activating the nonblocking events means to take all of the events from the nonblocking update events queue and put them in the active events queue. When these activated events are executed, they may cause additional processes to trigger and cause more active events and more nonblocking update events to be scheduled in the same time step. Activity in the current time step continues to iterate until all events in the current time step have been executed and no more processes, that could cause more events to be scheduled, can be triggered. At this point, all of the **$monitor** and **$strobe** commands would display their respective values and then the simulation time T can be advanced.

Event scheduling and re-triggering

Active events (shown in figure-1) such as blocking assignments and continuous assignments can trigger additional assignments and procedural blocks causing more active events and nonblocking assign update events to be scheduled in the same time step.Under these circumstances, the new active events would be executed before activating any of the nonblocking assign update events.

After the nonblocking (shown in figure-2) assign updates events are activated, the LHS of the nonblocking assignments are updated, which can trigger additional assignments and procedural blocks, causing more active events and nonblocking assign update events to be scheduled in the same time step. Simulation time does not advance while there are still active events and nonblocking assign update events to be processed in the current simulation time.

Figure-1: Verilog event queue - active events can trigger additional events in the same simulation time step



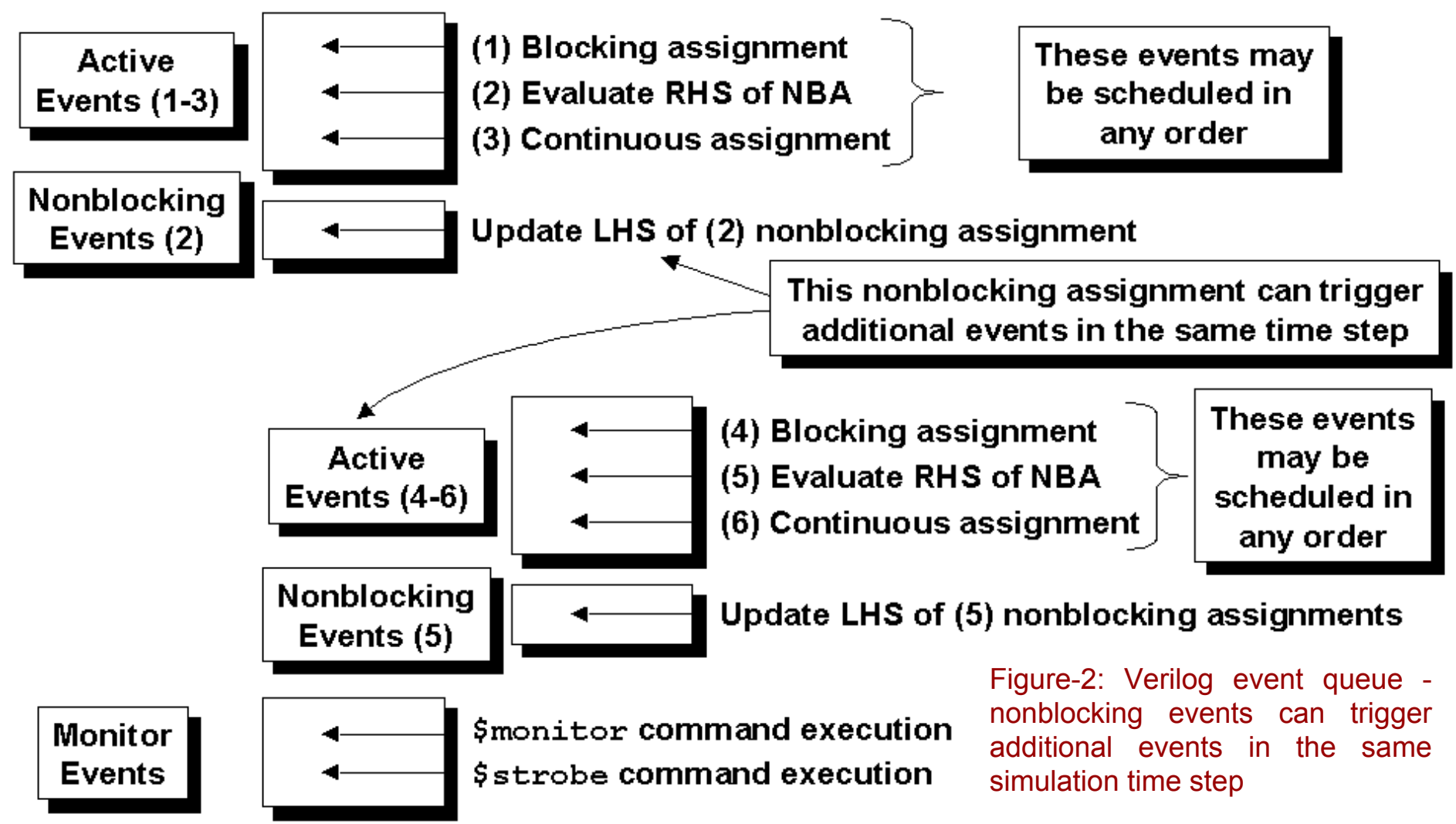


Figure-2: Verilog event queue - nonblocking events can trigger additional events in the same simulation time step

Behavioral modeling

Procedural Assignments: Procedural Assignment can drive only **reg, integer, real** and **time** data type. Which means left-side data type cannot be nets(wire data type). The value placed on a variable will remain unchanged until another procedural assignment overrides the variable with a different value. It can be used to model both combinational and sequential logic.

The LHS of a procedural assignment can be any one of the following:

- *reg*, *integer*, *real*, or *time* register variable or memory elements (**can not be wire**).
- A bit-select these variables (e.g., data[0]).
- A part select of these variables (e.g., data[15:7])
- A concatenation of any of the above

There are two types of procedural assignment statements:

- **blocking**
- **nonblocking**

Behavioral modeling

Blocking Statement (operator = used):

- A blocking assignment statements are executed in the order they are specified in a sequential block (keywords **begin** and **end** are used to group statements in a sequential blocks). The execution of next statement begin only after the completion of the present blocking assignments in a sequential block. **Blocking assignments are executed sequentially.**
- A blocking assignment will not block the execution of the next statement in a parallel block (Parallel blocks are specified by keywords **fork** and **join**).

```
module sequential();
        reg a;
        initial begin
        #10 a = 0; #11 a = 1;
        #12 a = 0; #13 a = 1;
        #14 $finish;
        end
endmodule
```

```
Simulation Output:
 0     a = x // a assigned x at time 0
 10 a = 0   // a assigned 0 at time 10
 21 a = 1   // a assigned 1 at time 21
 33 a = 0   // a assigned 0 at time 33
 46 a = 1   // a assigned 1 at time 46
Simulation completed at time 60
```

Behavioral modeling

Nonblocking Statement(operator <= used):

- Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block (because all the statements execution starts at time 0).
- Nonblocking assignments are executed concurrently.

```
// nonblocking assignment in a sequential block
module sequential();
        reg a;
        initial begin
        #10 a <= 0;
        #11 a <= 1;
        #5   a <= 0;
        #13 a <= 1;
        #20 $finish;
        end
endmodule
```

Simulation Output:

```
 0     a = x // a assigned x at time 0
 10   a = 0   // a assigned 0 at time 10
 11   a = 1   // a assigned 1 at time 11
  5    a = 0   // a assigned 0 at time 5
 13   a = 1   // a assigned 1 at time 13

Simulation completed at time 20
```

Behavioral modeling

Order of blocking assignment matters: Use blocking statement for Combinational

Order of blocking statements matters

Assume a = 1, b = 2, c = 4, d = 4, e = 5 initially

```
//Example1
always @(a, b, c)  // use always@*
   begin
       c = a + b;
       d = c + e;
   end
```

Example1 output: c = 3, d = 8

Order of blocking statements matters

Assume a = 1, b = 2, c = 4, d = 4, e = 5 initially

```
//Example2
always @(a, b, c) // use always@*
   begin
       d = c + e;
       c = a + b;
   end
```

Example2 output: c = 3, d = 9

Behavioral modeling

Order of non-blocking assignment does not matter: Use non blocking statement for sequential

```
//ordering does not matter in
nonblocking assignment

Assume a = 1, b = 2, c = 4, d = 4, e =
5 initially

//Example1
always @(a, b, c)  // use always@*
   begin
      c <= a + b;
      d <= c + e;
   end

Example1 output: c = 3, d = 9
```

```
//ordering does not matter in
nonblocking assignment

Assume a = 1, b = 2, c = 4, d = 4, e =
5 initially

//Example2
always @(a, b, c) // use always@*
   begin
      d <= c + e;
      c <= a + b;
   end

Example2 output: c = 3, d = 9
```

Behavioral modeling

Summary of Guidelines When to use blocking vs nonblocking procedural statements:

- *Guideline #1: When modeling sequential logic, use nonblocking assignments.*
- *Guideline #2: When modeling latches, use nonblocking assignments.*
- *Guideline #3: When modeling combinational logic with an always block, use blocking assignments.*
- *Guideline #4: When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.*
- *Guideline #5: Do not mix blocking and nonblocking assignments in the same always block.*
- *Guideline #6: Do not make assignments to the same variable from more than one always block.*
- *Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments.*
- *Guideline #8: Do not make assignments using #0 delays.*

Behavioral modeling

Why and When to use blocking and nonblocking assignment in Verilog?

- The answer is simulation related and how Verilog blocking and nonblocking assignments are scheduled to be executed by the simulator.

Execution of blocking assignments can be viewed as a one-step process:

- Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement.

Example: *Blocking assignment: evaluation and assignment are immediate*

```
always @ (a or b or c) // same as always @ (*)
begin
    x = a | b;        // Evaluate a | b, assign result to x
    y = a ^ b ^ c;    // Evaluate a ^ b ^ c, assign result to y
    z = b & ~c;       // Evaluate b &(~c), assign result to z
end
```

Behavioral modeling

Another Example of blocking assignment?

```
/*  Assume initial values of variables are a = 1, b=4, c=6, d =8 (before blocking
assignment) */
always @(posedge clk)
      begin // blocking: procedural assignment
            b = a;
            c = b;
            d = c;
       end
```

New values after blocking assignment executed: a =1, b=1, c=1, d=1

Behavioral modeling

Execution of nonblocking assignments can be viewed as a two-step process:

- Evaluate the RHS of nonblocking statements at the beginning of the time step.
- Update the LHS of nonblocking statements at the end of the time step.

Nonblocking assignments are only made to register data types and are therefore only permitted inside of procedural blocks, such as **initial** blocks and **always** blocks. Nonblocking assignments are not permitted in continuous assignments.

Example:

Nonblocking assignment: all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

```
always @ (a or b or c)
begin
    x <= a | b;          //1. Evaluate a | b but defer assignment of x
    y <= a ^ b ^ c;  //2. Evaluate a ^ b ^ c but defer assignment of y
    z <= b & ~c;     //3. Evaluate b&(~c) but defer assignment of z
end                  //4. Assign x, y, and z with their new values
```

Behavioral modeling

Another Example of Non-blocking assignment?

```
/*  Assume initial values of variables are a = 1, b=4, c=6, d =8 (before non-
blocking assignment) */
always @(posedge clk)
      begin // non-blocking: procedural assignment
            b <= a;
            c <= b;
            d <= c;
       end
```

New values after nonblocking assignment executed: a =1, b=1, c=4, d=6

Note: In “always” block sequential logic, use nonblocking assignment.

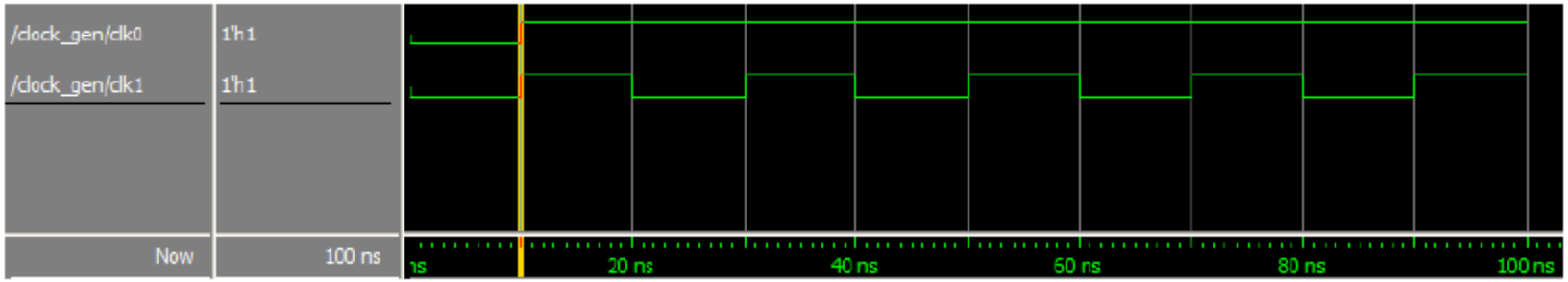
Behavioral modeling

Let's look at the Clock Generation logic in this example:

```
//Self-triggering always block with blocking assignment
Example1:
module clock_gen;
reg clk0; // clk0 declared as reg
Initial
      begin
            clk0 = 1'b0;
            #100 $stop;
      end
always@(clk0) #10 clk0 = ~clk0;
endmodule
```

```
//Self-triggering always block nonblocking assignment
Example2:
module clock_gen;
reg clk1; // clk1 declared as reg
Initial
      begin
            clk1 = 1'b0;
            #100 $stop;
      end
always @(clk1) #10 clk1 <= ~clk1;
endmodule
```

Behavioral modeling Simulation output:



Verilog Simulation Behavior: Verilog always block cannot trigger itself. Example1 uses blocking assignments. Blocking assignments evaluate their RHS expression and update their LHS value without interruption. The blocking assignment must complete before the @(clk0) edge-trigger event can be scheduled. By the time the trigger event has been scheduled, the blocking clk0 assignment has completed; therefore, there is no trigger event from within the always block to trigger the @(clk0) trigger.

Behavioral modeling

In contrast, Example2 uses nonblocking assignments. After the first @(clk1) trigger, the RHS expression of the nonblocking assignment is evaluated and the LHS value scheduled into the nonblocking assign updates event queue. Before the nonblocking assign updates event queue is "activated," the @(clk1) trigger statement is encountered and the always block again becomes sensitive to changes on the clk1 signal. When the nonblocking LHS value updated later in the same time step, the @(clk1) is again triggered.

Example of Race condition using blocking assignment:

A problem with blocking assignments occurs when the RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block and both equations are scheduled to execute in the same simulation time step, such as on the same clock edge. If blocking assignments are not properly ordered, a **race condition** can occur. When blocking assignments are scheduled to execute in the same time step, the order execution is unknown.

Behavioral modeling

```
module osc1 (y1, y2, clk, rst);
    output y1, y2;
    input  clk, rst;
    reg    y1, y2;


    always @(posedge clk or posedge rst)
       if (rst) y1 = 0;  // reset
       else     y1 = y2;

    always @(posedge clk or posedge rst)
       if (rst) y2 = 1;  // preset
       else     y2 = y1;
  endmodule
```

According to the IEEE Verilog Standard, the two "always" blocks can be scheduled in any order.

If the first always block executes first after a reset, both y1 and y2 will take on the value of 1.

If the second "always" block executes first after a reset, both y1 and y2 will take on the value 0.

This clearly represents a Verilog race condition.

Behavioral modeling

The race problem in previous blocking assignment will be solved by nonblocking assignment.

```
module osc1 (y1, y2, clk, rst);
output y1, y2;
input  clk, rst;
reg    y1, y2;

always @(posedge clk or posedge rst)
  if (rst) y1 <= 0;  // reset
  else     y1 <= y2;

always @(posedge clk or posedge rst)
  if (rst) y2 <= 1;  // preset
  else     y2 <= y1;
endmodule
```

At the positive edge of clock, the values of all right-hand-side variables are "read," and the right-hand-side expressions are evaluated and stored in temporary variables.

During the write operation, the values stored in the temporary variables are assigned to the left-hand-side variables.

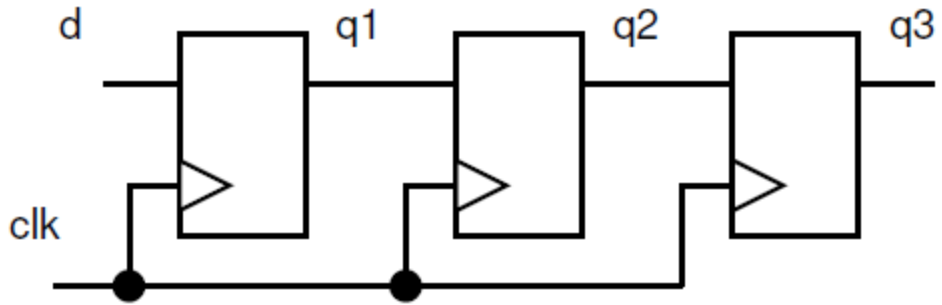
Separating the read and write operations ensures that the values of registers y1 and y2 are swapped correctly, regardless of the order in which the write operations are performed.

Behavioral modeling

Let's see how simulator processes the Nonblocking Assignments in previous example.

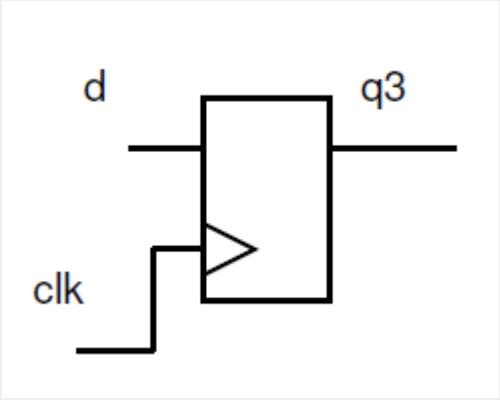
```
//Process nonblocking assignments by using temporary variables
module swap;
reg y1, y2, temp-y1, temp-y2;

        always @(posedge clock)
                begin
                //read operation,store values of right-hand-side expressions in temporary variable:
                        temp-y1 = y1 ;
                        temp-y2 = y2;

                //Write operation, Assign values of temporary variables to left-hand-side variables
                        y1 = temp-y2;
                        y2 = temp-y1 ;
                end
endmodule
```

Sequential Pipeline modeling Example implemented with blocking assignment:




```
module pipeb1 (q3, d, clk);
output q3;
input d;
input clk;
reg q3, q2, q1;
always @(posedge clk) begin
q1 = d;
q2 = q1;
q3 = q2;
end
endmodule
```

On every clock edge, the input value is transferred directly to the q3-output without delay. Actual synthesized result

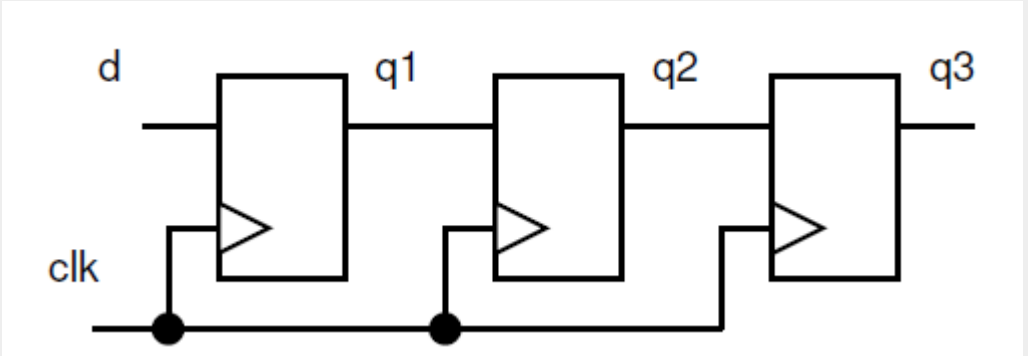


Sequential Pipeline modeling Example implemented with nonblocking assignment:

If the blocking-assignment is replaced with nonblocking assignments, each will simulate correctly and synthesize the desired pipeline logic.

```
module pipen1 (q3, d, clk);
output q3;
input d;
input clk;
reg q3, q2, q1;
always @(posedge clk)
        begin
                q1 <= d;
                q2 <= q1;
                q3 <= q2;
        end
endmodule
```

Actual Synthesized circuit



Mixed sequential & combinational logic - use nonblocking assignments

When combining combinational and sequential logic into a single always block, code the always block as a sequential always block with nonblocking assignments.

Example:

```
module nbex1 (q, a, b, clk, rst_n);
output q;
input clk, rst_n;
input a, b;
reg q;
always @(posedge clk or negedge rst_n)
if (!rst_n)
        q <= 1'b0;
else
        q <= a ^ b; // a ^ b is combo logic
endmodule
```

```
//implemented with 2 separate always
module nbex1 (q, a, b, clk, rst_n);
output q;
input clk, rst_n;
input a, b;
reg q, y;
always @(a or b) // combo always block
y = a ^ b;
//sequential always block
always @(posedge clk or negedge rst_n)
        if (!rst_n) q <= 1'b0;
        else        q <= y;
endmodule
```

Multiple assignments to the same variable: Bad coding style(race condition)

Making multiple assignments to the same variable from more than one always block is a Verilog race condition, even when using nonblocking assignments. In example below, two always blocks are making assignments to the q-output, both using nonblocking assignments. Since these always blocks can be scheduled in an order, the simulation output is a race condition.

```
module badcode1 (q, d1, d2, clk, rst_n);
output q; input d1, d2, clk, rst_n;
reg q;
always @(posedge clk or negedge rst_n)
        if (!rst_n)  q <= 1'b0;
        else         q <= d1;
always @(posedge clk or negedge rst_n)
        if (!rst_n)  q <= 1'b0;
        else         q <= d2;
endmodule
```

When Synopsys tools read this type of coding example, the following warning message is issued:

Warning: In design 'badcode1', there is 1 multiple-driver net with unknown wired-logic type.

The pre-synthesis simulation does not even closely match the post-synthesis simulation(produces 2 flip flops driving 2 input and gate) in this example.

Behavioral Modeling: Nonblocking assignments & $display

- Use $strobe to display values that have been assigned using nonblocking assignments.
- Nonblocking assignments are updated after all $display commands

Example:

```
module display_cmds;
reg a;
initial
   $monitor("\$monitor: a = %b", a);
initial
  begin
    $strobe ("\$strobe : a = %b", a);
    a = 0;
    a <= 1;
    $display ("\$display: a = %b", a);
    #1 $finish;
  end
endmodule
```

The below output from the above simulation shows that the $display command was executed in the active events queue, before the nonblocking assign update events were executed.
Simulation output:

$display: a = 0
$monitor: a = 1
$strobe : a = 1

Use $strobe to display values that have been assigned using nonblocking assignments.

Behavioral Modeling:#0-delay assignments

- #0 delay forces an assignment to the end of a time step (that is executed last, after all other statements in that simulation time are executed). This is used to eliminate race conditions.
- if there are multiple zero delay statements, the order between them is nondeterministic.

Example:

```
initial
        begin
                X = 0 ;
                y = 0 ;
        end

initial
        begin
                #0 X = 1; //zero delay control
                #0 y = 1;
        end
```

The four statements- X = 0, y = 0, X = 1, y = 1 are scheduled to be executed at simulation time 0. However, since X = 1 and y = 1 have #0, they will be executed last.

Thus, at the end of time 0, X will have value 1 and y will have value 1. The order in which X = 1 and y = 1 are executed is not deterministic.

Behavioral Modeling:Multiple nonblocking assignments to the same variable

Making multiple nonblocking assignments to the same variable in the same always block is defined by the Verilog Standard. The last nonblocking assignment to the same variable wins!

Nonblocking assignments shall be performed in the order the statements were executed. Consider the following example:

```
initial
      begin
            a <= 0;
            a <= 1;
      end
```

Both the statements are scheduled to be executed at simulation time 0. First a is assigned 0 and then a is assigned 1 and hence at the end of time step, a will be 1(last nonblocking statement to variable a <= 1 wins).

Behavioral Modeling:Timing Control

The time at which procedural statements will get executed shall be specified using timing controls.The statements within a sequential block are executed in order, but, in the absence of any delay, they all execute at the same simulation time (the current **time step)**. In reality there are delays that are modeled using a timing control. There are three methods of timing control used in Verilog.

- **Delay Based Timing Control**
  - Regular Delay Control
  - Intra-assignment Delay Control
  - Zero Delay Control
- **Event Based Timing Control**
  - Regular event control
  - Named Event Control
  - Event OR control
- **Level Sensitive Timing Control**

Behavioral Modeling:Timing Control

Delay Based Timing Control(Non-Synthesizable): In this, timing control is achieved by specifying waiting time to execution, when the statement is encountered. The symbol "**#**" is used to specify the delay. There are 3 ways, delay based timing control can be specified.

Regular Delay Control: Delays the execution of a procedural statement by specific simulation time. Syntax for delay based timing control is*: **#< time > < statement >;***
Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment. Example is given below:

```
parameter latency = 20; //define parameters
parameter delta = 2;
reg x, y, z, p, q; //define register variables
initial
begin x = 0; // no delay control
        #10 y = 1; // delay control with a number. Delay execution of y = 1 by 10 units
        #latency z = 0; //delay control with identifier latency (20 units)
        #(latency + delta) p = 1; // Delay control with expression
       #(4:5:6) q = 0; // minimum,typical and maximum delay values.
end
```

Behavioral Modeling:Timing Control

Intra-assignment Delay Control(Non-Synthesizable): In this case delays will be specified on the right hand side of the assignment operation. The RHS expression will be evaluated at the current time and the assignment will be occurred only after the delay.

Example:

```
//define register variables
reg a, b, c, y;
initial
    begin
        a=0; c=0;
        //intra assignment delays
        y = #10 a + c; /*Take value of a and c  at the time=0, evaluate
                        a+c and then wait 10 time units to assign value to y */
    end
```

Behavioral Modeling:Timing Control

Zero Delay Control(non-synthesizable): The order of execution of procedural statements in different always-initial blocks is nondeterministic. Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed. This is used to eliminate race conditions. However, if there are multiple zero delay statements, the order between them is nondeterministic.

Example:

```
initial
    begin
        a = 2 ;
        b = 3 ;
    end
initial
    begin
        #0 a = 5; //zero delay control
        #0 b = 6;
    end
```

since a = 5 and b = 6 have #0, they will be executed last. Thus, at the end of time 0, a will have value 5 and b will have value 6. The order in which a = 5 and b = 6 are executed is not deterministic.

Behavioral Modeling:Timing Control

Event-Based Timing Control(Synthesizable): An event is the change in the value on a register or a net. In this, execution of a statement or a block of a statement is controlled by an event.

Regular event control: Execution of statement will happen on changes in signal or at a positive or negative transitions of signals.

Example:

```
@(clock) q = d; //q = d is executed whenever signal clock changes value

@(posedge clock) q = d; /*q = d is executed whenever signal clock does
                          a positive transition ( 0 to 1,x or z, x to 1, z to 1 )*/

@(negedge clock) q = d; /* q = d is executed whenever signal clock does
                           a negative transition ( 1 to 0,x or z, x to 0, z to 0) */

q = @(posedge clock) d; /* d is evaluated immediately and assigned to q at the
                           positive edge of clock */
```

Behavioral Modeling:Timing Control

Named event control(non-synthesizable): A named event is declared by the keyword ***event***. An event is triggered by the symbol **->**. The triggering of the event is recognized by the symbol @.

Example:

```
//This is an example of a data buffer storing data after the last packet of data has arrived.
    event received-data; // define an event called received-data
    always @(posedge clock) //check at each positive clock edge
        begin
            if(last-datapacket) //If this is the last data packet
                ->received-data; //trigger the event received-data
        end
    always @(received-data) //Await triggering of event received-data,
    // When event is triggered, store all four packets of received data in data buffer
    data-buf = {data_pkt [0] , data_pkt [1], data_pkt [2], data_pkt [3] };
```

Behavioral Modeling:Timing Control

Event OR control(Synthesizable): Sometimes a transition signals or events can trigger the execution of a statement or a block of statements. This is expressed as an OR of events or signals. The list of events or signals expressed as an OR is also known as a *sensitivity list*. The keyword ***or*** is used to specify multiple triggers.

Example:

```
//A level-sensitive latch with asynchronous reset
always @( reset or clk or d) //Wait for reset or clk or d to change
begin
    if (reset) //if reset signal is high, set q to 0.
        q <= 1'b0;
    else if (clk) //if clk is high, latch input
        q <= d;
end
```

Behavioral Modeling: Timing Control

Level-Sensitive Timing Control(wait is Non-Synthesizable):

Verilog allows level-sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed. The keyword **wait** is used for level-sensitive constructs.

Example:

```
always
    wait (cnt-enable) #20 count = count + 1;
```

In the above example, the value of cnt-enable is monitored continuously. If cnt-enable is 0, the statement is not entered. If it is logical 1, the statement count = count + 1 is executed after 20 time units. If cnt-enable stays at 1, count will be incremented every 20 time units.

Behavioral Modeling:Sequential Blocks

Block Statements: Block statements are used to group two or more statements together, so that they act as one statement. There are two types of blocks:

- Sequential blocks
- Parallel blocks

Sequential blocks: The sequential block is defined using the keywords ***begin*** and ***end*** and has following characteristics:

- The statements in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution (except for nonblocking assignments with intra-assignment timing control).
- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

Behavioral Modeling: Sequential Blocks Example

```
// Sequential block without delay
        reg x, y;
        reg [1:0] z, w;
        initial
//completes at simulation time 0
        begin
          x = 1'b0;
          y = 1'b1;
          z = {x, y};
          w = {y, x};
        end
```

```
//Sequential blocks with delay.
reg x, y;
reg [1:0] z, w;
initial
begin
x = 1'b0; //completes at simulation time 0
#5 y = 1'b1; //completes at simulation time 5
#10 z = {x, y}; //completes at simulation time 15
#20 w = {y, x); //completes at simulation time 35
end
```

Behavioral Modeling: Parallel blocks

Parallel Blocks(fork join non synthesizable): The parallel block is defined using the keywords ***fork*** and ***join*** and has the following characteristics:

- Statements in a parallel block are executed concurrently.
- Ordering of statements is controlled by the delay or event control assigned to each statement.
- If delay or event control is specified, it is relative to the time the block was entered.

Example: //Parallel blocks with delay.

```
reg x, y;
reg [1:0] z, w;
initial
fork
    x = 1'b0; //completes at simulation time 0
    #5 y = 1'b1; //completes at simulation time 5
    #10 z = {x, y}; //completes at simulation time 10
    #20 W = {y, x}; //completes at simulation time 20
join
```

All statements in a parallel block start at the time when the block was entered. Thus, the order in which the statements are written in the block is not important.

Behavioral Modeling: Special Features of Blocks

We discuss three special features available with block statements: *nested blocks, named blocks,* and *disabling of named blocks.*

Nested blocks: Blocks can be nested. Sequential and parallel blocks can be mixed as shown below.

Example:

```
//Nested blocks
initial
begin
    x = 1'b0;
        fork
            #5 y = 1'b1;
            #10 z = {x, y};
        join
    #20 w = {y, x};
end
```

Behavioral Modeling: Special Features of Blocks

Named blocks(Synthesizable): Blocks can be given names.

- Local variables can be declared for the named block.
- Named blocks are a part of the design hierarchy. Variables in a named block can be accessed by using hierarchical name referencing.
- Named blocks can be disabled, i.e., their execution can be stopped.

```
module top;//Named blocks
initial
begin: block1 //sequential block named  block1
integer i; //integer i is static and local to block1
// can be accessed by hierarchical name, top.
block1.i
...
...
end
```

```
initial
fork: block2 //parallel block named block2
reg i; // register i is static and local to block2
// can be accessed by hierarchical name,
top.block2.i
...
...
join
```

Behavioral Modeling: Special Features of Blocks

Disabling named blocks: The keyword **disable** provides a way to terminate the execution of a block. Disabling a block causes the execution control to be passed to the statement immediately succeeding the block.

Example: integer i; reg [15:0] flag;

```
initial
begin
        flag[15:0] = 16'b0010_0000_0000_0000;
        i = 0;
  begin: block1 //The main block inside while is named block1
        while (i < 16)
                begin
                        if (flag[i])
                                begin
                                        $display("Encountered a TRUE bit at element number %d", i);
                                        disable block1; //disable block1 because you found true bit.
                                end
                        i=i+1;
                end
   end
end
```

Behavioral Modeling

Conditional Statements(Synthesizable): The condition (if-else if - else) is used to make a decision whether a statement is executed or not. The keywords ***if*** and ***else*** are used to make conditional statement. The conditional statement can appear in following form:

```
//Type-1 conditional statement(no else statement).
      if (<expression>) true-statement ; //If <expression> is true(1), true-statement executed

//Type-2 conditional statement(one else statement)
      if (<expression>) true-statement ; /*If <expression> is true(1),
        else false-statement ; true-statement executed else false-statement gets executed*/

//Type 3 conditional statement (nested if-else-if).
      if (<expression1>) true-statement1;// Choice of multiple statements,
      else if (<expression2>) true-statement2 ; // but only one gets executed.
      else if (cexpression3>) true-statement3 ;
      else default-statement ;
```

Behavioral Modeling: (

```
//Simple if
module simple_if();

reg latch;
wire enable,din;

always @ (enable or din)
if (enable)
        latch <= din;

endmodule
```

//Synthesized Hardware of "if": Latch

```
// if-else
module if_else();
reg dff;
wire clk,din,reset_n;

always @ (posedge clk)
if (!reset_n)
        dff <= 0;
else
        dff <= din;

endmodule
```

```
module nested_if();
reg [3:0] counter;
reg clk,reset,enable,up_en,down_en;
always @ (posedge clk)
if (!reset_n)
  counter <= 4'b0000;
//if both enable and up_en are 1
else if (enable && up_en)
  counter <= counter + 1'b1;
//if both enable and down_en are 1
end else if (enable && down_en)
  counter <= counter - 1'b1;
else
  counter <= counter;
endmodule
```

Multiway Branching

The case statement (Synthesizable)is a multi-way decision statement that tests whether an expression matches one of the expressions and branches accordingly. Keywords ***case*** and ***endcase*** are used to make a case statement. The case statement syntax is as follows.

```
case (expression)
    case1: statement1; // if expression matches with case1 statement1 executed
    case2: statement_2;// if expression matches with case2 statement2 executed
    case3: statement_3;// if expression matches with case3 statement3 executed
    ...
    ...
    default: default_statement;//if none of the alternative matches default executed
endcase
```

Note: Default statement is optional

Multiway Branching

Case statement example:

```
module mux (a,b,c,d,sel,y); //4:1 Mux
input a, b, c, d; /Mux inputs
input [1:0] sel; //Mux select inputs
output y; //Mux output
reg y; // declared as reg
always @ (a or b or c or d or sel)
case (sel[1:0])
        2'b00 : y = a;
        2;b01 : y = b;
        2'b10 : y = c;
        2'b11: y = d;
        default : y = x;
endcase
endmodule
```

```
//Another example

case (alu_ctr)
        2'b00: aluout = a + b;
        2'b01: aluout = a - b;
        2'b10: aluout = a & b;
        default:aluout = 1'bx; /* Treated as don't
            cares for minimum logic generation*/
endcase
//default case is if alu_ctr is 11, or contain 'x's
```

Multiway Branching

The casex and **casez** statements:(Synthesizable) Special versions of the case statement allow the x ad z logic values to be used as "don't care":

- casez : Treats z and ? as don't care
- casex : Treats x, ? and z as don't care

```
module casez_example();
reg [3:0] opcode;
reg [1:0] a,b,c;
reg [1:0] out;
always @ (opcode or a or b or c)
casez(opcode)
  4'b1zzx : out = a; /* Don't care about lower 2:1
                        bit, bit 0 match with x */
  4'b01?? : out = b; // bit 1:0 is don't care
  4'b001? : out = c;  // bit 0 is don't care
    default : out = a;
endcase
```

```
module casex_example();
reg [3:0] opcode;
reg [1:0] a,b,c;
reg [1:0] out;
always @ (opcode or a or b or c)
casex(opcode)
  4'b1zzx : out = a; // Don't care  2:0 bits
  4'b01?? : out = b; // bit 1:0 is don't care
  4'b001? : out = c; // bit 0 is don't care
    default : out = a;
endcase
```

Multiway Branching

casex more examples:

In casex(a) example below, the case choices constant “a” may contain z, x or ? which are used as don’t cares for comparison. With case the corresponding simulation variable would have to match a tri-state, unknown, or either signal. In short, case uses x to compare with an unknown signal. Casex uses x as a don’t care which can be used to minimize logic.

```
casex (a)
      2’b1x: msb = 1; // msb = 1 if a = 10 or a = 11
      // If this were case(a) then only a=1x would match.
      default: msb = 0;
endcase
```

Multiway Branching

casez more examples:

casez is the same as casex except only ? and z (not x) are used in the case choice constants as don't cares. Casez is favored over casex since in simulation, an inadvertent x signal, will not be matched by a 0 or 1 in the case choice.

Example:

```
casez (d)
    3'b1??: b = 2'b11; // b = 11 if d = 100 or greater
    3'b01?: b = 2'b10; // b = 10 if d = 010 or 011
    default: b = 2'b00;
endcase
```

Loops

There are four types of looping statements in Verilog:

- while
- for
- repeat
- forever

While Loop: Defined using the keyword while and contains an expression. The while loop continues until the expression is true. It terminates when the expression is false. If the calculated value of expression is z or x, it is treated as a false. The value of expression is calculated each time before starting the loop.

Example:

```
integer count;//Increment count from 0 to 63. Exit at count 64
initial
  begin
      count = 0;
      while (count < 64) //Execute loop till count is 63, exit at count 64
          count = count + 1;
  end
```

Loops

for Loop(Synthesizable): Defined using the keyword **for**. The execution of for loop block is controlled by a three step process, as follows:

- An initial condition
- A check to see if the terminating condition is true
- A procedural assignment to change value of the control variable

Example:

```
'define MAX-STATES 16 //Initialize array elements
integer state [0: 'MAX-STATES-1]; //integer array state with elements 0: 15
        integer i;
        initial
          begin
              for(i = 0; i < 16; i = i + 2) //initialize all even locations with 0
                    state[i]= 0;
              for(i = 1; i < 16; i = i + 2) //initialize all odd locations with 1
              state[i] = 1;
          end
```

Loops

Repeat Loop(Non-synthesizable): Defined using the keyword **repeat**. The repeat loop continuously executes the loop for a given number of times. The number of times the loop executes can be mention using a constant or an expression. The expression is calculated only once, before the start of loop and not during the execution of the loop. If the expression value turns out to be z or x, then it is treated as zero, and hence loop block is not executed at all.

Example: integer count; //Increment and display count from 0 to 127

```
initial

    begin
        count = 0;
        repeat (128)
            begin
                $display("Count = %d", count);
                count = count + 1;
            end
    end
```

Loops

Forever loop: Defined using keyword forever. The loop does not contain any expression and executes forever until the $finish task is encountered.A forever loop is typically used in conjunction with timing control constructs.

Example:

```
//Use forever loop instead of always block for clock generation in test bench
        reg clock;
        initial
            begin
                clock = 1'b0;
                forever #20 clock = ~clock; //Clock with period of 40 units
            end
```

Note: while and forever loop must contain @(posedge/negedge clock) statement in order to be synthesizable.

While and forever loop example to be synthesizable

//while loop
To avoid combinational feedback during synthesis, a while loop must be broken with an @(posedge/negedge clock) statement. For simulation a delay inside the loop will suffice.

```
while (!overflow)
  begin
       @(posedge clk);
       a = a + 1;
  end
```

//forever loop
To avoid combinational feedback during synthesis, a forever loop must be broken with an @(posedge/negedge clock) statement. For simulation a delay inside the loop will suffice.

```
forever
begin
       @(posedge clk);
       a = a + 1;
end
```

Verilog Task and Function

Verilog designers are required to implement the same functionality (common procedures) at many places in a behavioral design. In this way, common procedures can be written once and can execute from different places. Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design.

- Both task and function are called from always or initial block and contain only behavioural statements.
- Definition of task and function must be in a module.
- The highlighting difference between task and function is that, only task can handle event, delay or timing control statements and function executes in zero simulation time.
- Tasks have input, output, and inout arguments;
- functions have only input arguments.

Verilog Task and Function Comparison

task

1. A task can enable other tasks and functions
2. Tasks may execute in non-zero simulation time.
3. Tasks may contain delay(#10), event(@posedge/negedge), or timing control statements(wait).
4. Tasks may have zero or more arguments of type input, output or inout.
5. Tasks do not return with a value but can pass multiple values through output and inout arguments.
6. Tasks do not contain always or initial block.

function

1. A function can enable another function but not another task.
2. functions always execute in zero simulation time.
3. Functions can not contain delay, event, or timing control statements.
4. Functions must have at least one input argument. They can have more than one input. They cannot have output or inout arguments.
5. Functions always return a single value.
6. Functions do not contain always or initial block.

Verilog task example1

```
module reverse_bits_Task;
parameter BITWIDTH = 8;
task reverse_bits; // task definition starts here
input [BITWIDTH – 1 : 0] din; // Input declaration
output [BITWIDTH – 1 : 0] dout; //Output declaration
integer i; //Values are passed to and from a task through arguments.
   begin
      for (i=0; i < BITWIDTH; i = i +1)
            dout[BITWIDTH - i] = din[i];
    end
endtask // task definition ends here
reg [BITWIDTH – 1] REG_X, NEW_REG;
always @(REG_X)
      reverse_bits(REG_X,NEX_REG); // task being called
endmodule
```

Verilog task example2

```
module MUX4X1_Using_TASK (Q, IN, SEL);
  input [3:0] IN;
  input [1:0] SEL;
  output Q;
  reg Q;
always @(IN or SEL)
   mux(IN, SEL,Q); //task called

task mux; //task mux defined
input [3:0] in;
input [1:0] sel;
output out;
case (sel)
   2'b00: out = in[0];
   2'b01: out = in[1];
   2'b10: out = in[2];
   2'b11: out = in[3];
endcase
endtask
endmodule
```

```
// test bench
module test;
reg [3:0] muxin;
reg [1:0] msel;
wire mout;

MUX4X1_Using_TASK mux_task(mout,muxin,msel);
initial
   $monitor($time," -->muxin = %b, msel = %b,
                      mout =  %b",muxin,msel,mout);

initial
  begin
           muxin = 4'b0111; msel = 2'b01;
     #10  muxin = 4'b0111; msel = 2'b10;
     #10  muxin = 4'b0110; msel = 2'b00;
     #100 $finish;
  end
endmodule
```

Verilog function example1

```
// Define a module that contains the
//function Parity_Calc
module parity_calc(Addr, Par);
  input [31:0] Addr;
  output Par;
  reg Par;
always @(Addr)
  Par = Parity_Calc(Addr); //function call

function Parity_Calc;// function definition
 input [31:0] addr;
 Parity_Calc_Func = ^addr;
endfunction
endmodule
```

```
module test;
reg [31:0] Addr;
wire Par;
parity_calc pcalc(Addr,Par); //module instantiate
initial
$monitor($time," -->Addr = %b, Par = %b, ",
Addr,Par);
initial
  begin
       Addr = 32'b1111_0000_0011_0111;
       #10  Addr = 32'b1111_0000_0011_0101;
       #10  Addr = 32'b1111_0001_1111_0101;
       #100 $finish;
  end
endmodule
```

Simulation output:

```
0 -->Addr = 00000000000000001111000000110111, Par = 1
10 -->Addr = 00000000000000001111000000110101, Par = 0
20 -->Addr = 00000000000000001111000111110101, Par = 1
```

Verilog function example2

```
module shifter;
'define LEFT-SHIFT 1'b0 //Left shifter
'define RIGHT-SHIFT 1'b1 //Right shifter
reg [31:0] addr, left-addr, right-addr;
reg control;
always @ (addr )// Compute the right- and left-shifted values whenever a new address value appears
        begin // call the function defined below to do left and right shift.
                left-addr = shift(addr, 'LEFT-SHIFT);
                right-addr = shift(addr, 'RIGHT-SHIFT);
        end
function [31: 0] shift; //define shift function. The output is a 32-bit value.
input [31:0] address; input control;
        begin //set the output value appropriately based on a control signal.
                shift = (control == 'LEFT-SHIFT) ?(address << 1) : (address >> 1);
        end
endfunction
endmodule
```