Module 9

Understanding the Simulation Cycle

cadence

This module examines in detail simulator execution of procedural blocks. It reviews procedural blocks and event controls, looks again at blocking and nonblocking procedural assignments, introduces the simulation cycle, and presents level-based event controls and simple delays.

Module Objective

In this module, you:

- Produce higher quality code that is less subject to nondeterminism and race conditions

Topics

- Procedural blocks and event control review
- Blocking procedural assignment review
- Making nonblocking procedural assignments
- Nonblocking assignments and the simulation cycle
- Synchronizing procedures
  - With event controls
  - With level-sensitive event controls
  - With procedural delay controls

cadence

Your objective is to produce higher-quality code less subject to indeterminacy and race conditions. To do that, you need to understand something about how the simulator executes processes and their statements. Some terminology definitions are as below:

- Simulation Time: Time value maintained by the simulator to model the actual time it would take for the circuit being simulated
- Process: Verilog code to evaluate, e.g., always and initial blocks and continuous assignments
- Evaluation Event: Evaluation of a process
- Update Event: Change in value of a net or variable
- Event Queue: Queue of events, ordered by simulation time
- Scheduling an Event: Putting an event onto the event queue
- Simulation Cycle: Processing all active events in the event queue

Procedural Blocks and Event Control Review

An always construct is a procedural block that loops continuously throughout the simulation.

An initial construct is a procedural block that executes once.

The "@" token introduces an event control, blocking further execution for either:

- A single event identifier.
- An event expression.
  - Can create an event list with **or** and comma "**,**" operators
  - Can qualify an expression term with **posedge** or **negedge**

Can give a block a name.

Combinational logic can use the blocking assignment "**=**" operator.

Sequential logic must use the nonblocking assignment "**<=**" operator.

Multiple procedures execute "concurrently".

```
module adder (
  input      [3:0] a, b, c,
  output reg [4:0] o, p
);
  always @(a or b or c)
    begin : ADDING
      o = a + b;
      p = a + c;
    end
endmodule
```

```
module flop (
  output reg  q,
  input  wire d, clk, clr
);
  always @(posedge clk)
    if (clr) // CLOCKING
      q <= 0;
    else
      q <= d;
endmodule
```

159

cadence

An always construct is a procedural block that loops continuously throughout the simulation.

An initial construct is a procedural block that executes once.

The at (@) token introduces an event control, which blocks further execution until an event occurs. The event control can utilize a single event identifier or it can utilize an event expression that may be a list of event expressions separated by the or token or separated by commas with terms that can be qualified with posedge or negedge qualifiers.

You can name a sequential block by appending a colon and identifier to the begin keyword. A named block creates another level of scope that can declare its own variables. Of course you can always just make a pertinent comment about the statement's purpose.

A procedural block that represents combinational logic can use the blocking assignment (=) operator.

A procedural block that represents sequential logic must use the nonblocking assignment (<=) operator.

Multiple procedural blocks execute in a manner that appears concurrent to the user. Multiple procedural blocks scheduled to execute in the same simulation cycle can execute in any order.

Blocking Procedural Assignment Review

- Blocking assignment "=" operator
- Blocks execution of subsequent statements until assignment completes:
  - By default immediately
  - Later slides show how to delay
- Almost all examples so far have used blocking assignments.
- This causes problems with some code structures.

```
reg [7:0] byte=8'b00001111;
...
// try to swap nibbles
byte[3:0] = byte[7:4];
byte[7:4] = byte[3:0];
// what is the result?
...
```

Byte is now 00000000

cadence

The procedural assignments you have seen up to this point are blocking assignments (=). They are called blocking assignments because they block execution of the next statement until they complete. This ensures that future statements can use the new value of the updated variable. You will often see it written that these assignments are immediate, but that is not necessarily true, as you will later see how you can deliberately delay it.

This illustration attempts to use blocking assignments to swap the upper and lower halves of a byte. It first assigns the upper half to the lower half and then assigns the lower half to the upper half. This does not have the desired affect, as the simulator completes the first assignment before it starts the second assignment. At the end of the first assignment, both halves have the same value.

You can work around such affects by utilizing a temporary variable, but an even better solution performs both assignments in a way that appears concurrent.

Nonblocking Procedural Assignment Review

Nonblocking assignment "<=" operator:

- RHS expression value is calculated.
- LHS variable update is *scheduled*.

```
reg [7:0] byte= 8'b00001111;
...
// try to swap nibbles
byte[3:0] <= byte[7:4];
byte[7:4] <= byte[3:0];
// what is the result?
...
```

Byte is still 00001111

Byte is still 00001111

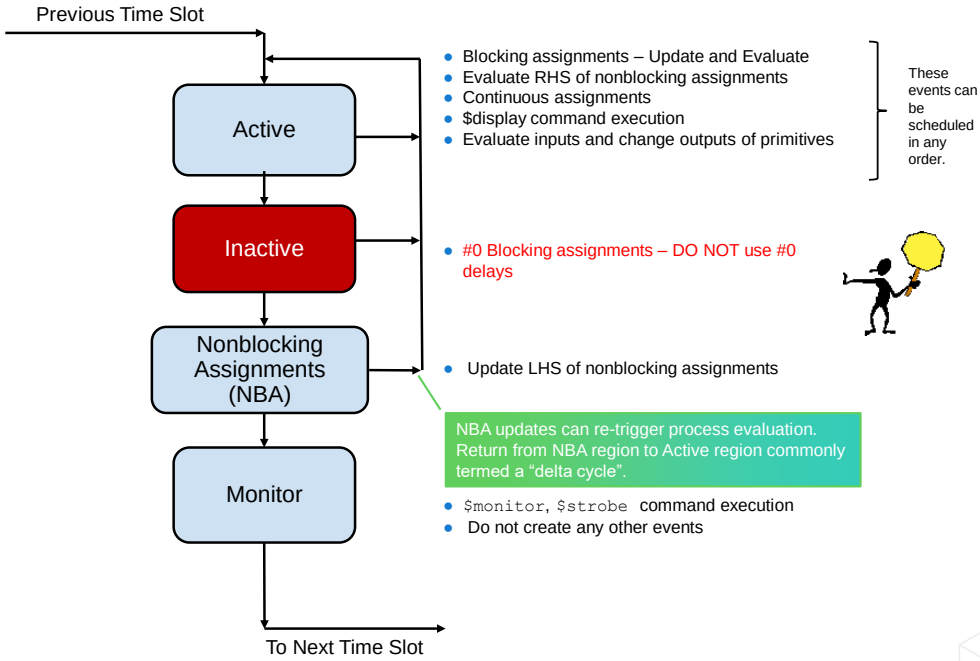
Byte is 11110000
on next delta cycle.

You can alternatively make nonblocking assignments. They are called nonblocking assignments because their completion is scheduled and they do not block execution of the next statement. The simulator instead calculates and retains the new value, but schedules the actual variable update for a point in the simulation where all currently triggered blocks have executed up to the point where they are all now all blocked. This ensures that any other triggered block that reads the variable reads the old value and not the new value.

Nonblocking assignments use the same token as the less-than-or-equal-to (<=) operator. It may be helpful to you to remember that the nonblocking word is longer than the blocking word and that the nonblocking operator is longer than the blocking operator.

This illustration uses nonblocking assignments to swap the upper and lower halves of a byte. It first assigns the upper half to the lower half and then assigns the lower half to the upper half. This does have the desired affect, as the simulator evaluates the first assignment after completing the second assignment. The simulator does not update the value of the variable until the procedural block next blocks at the event control.

Simplified Stratified Event Queue



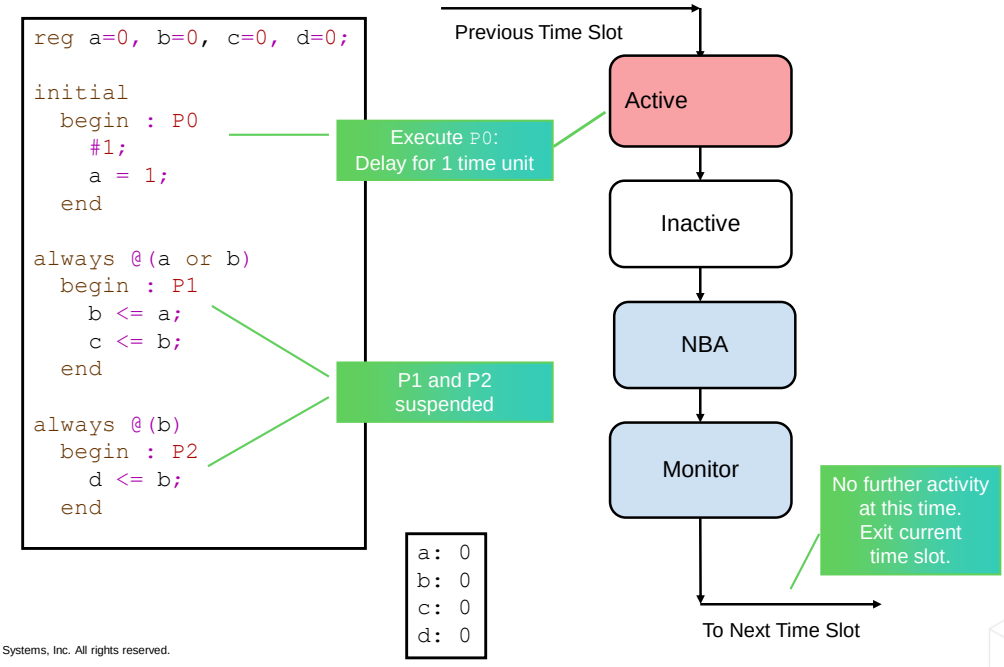
A simulation timeslot is divided into ordered regions to provide a predictable interaction between design constructs.

The Verilog event schedule has four regions for each simulation time:

- The Active region is for executing process statements.
- The Inactive region is for executing process statements postponed with a zero (#0) procedural delay.
- The NBA region is for updating nonblocking assignments.
- The Monitor region is for executing $monitor and $strobe and for calling user routines registered for execution during this read-only region. You cannot create additional events within this region.

The first three of these regions are iterative. They can schedule events that require return to the Active region. When no more events exist for the current simulation time, the simulator executes Monitor statements and then advances simulation time to the next time for which events are scheduled. The simulation terminates when no such future events exist.

Simulation Cycle: 1/6

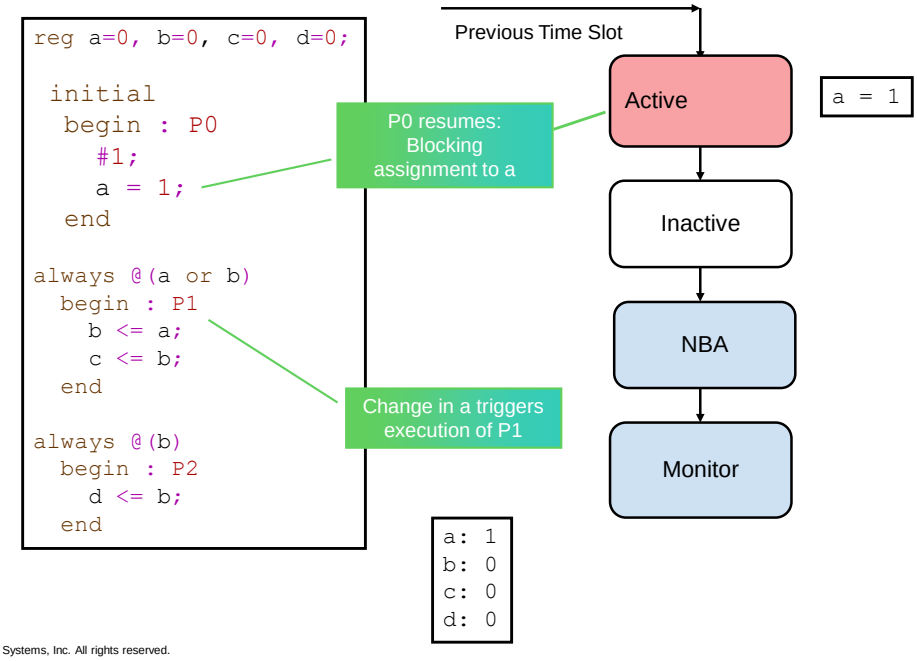


To demonstrate the behavior of nonblocking assignments and the simulation cycle, consider a module with three procedures, P0, P1 and P2. P1 is an always procedure triggered by an event on a or b. When either event occurs, procedure P1 makes nonblocking assignments to variable b and variable c, and returns to the top of the loop to again block. P2 also is an always procedure triggered by an event on variable b. When the event occurs, procedure P2 makes a nonblocking assignment to variable d, and returns to the top of the loop to again block. When an event occurs on variable b, the P1 and P2 processes will resume in either order. This order is predetermined by the simulator implementation and you cannot know or affect it.

As the simulation starts at time 0, the simulator moves all procedural blocks to the active evaluate events queue, and evaluates them in any order:

- As the simulator evaluates procedure P0, it immediately encounters a simple delay of #1, so schedules P0 resumption for the later time.
- As the simulator evaluates procedures P1 and P2, it immediately encounters event controls, so blocks further execution until one of the associated events occurs.

Simulation Cycle: 2/6



After exhausting all events at the current time, the simulator jumps to the next event time, and resumes execution of P0.

Here, the P0 procedure makes an immediate blocking assignment to register a, after which P0 completes.

When the simulator reaches the end of the active region, it checks to see if any procedural blocks have been triggered by the executed blocking assignments. Here a change in the value of a triggers P1.

Simulation Cycle: 3/6

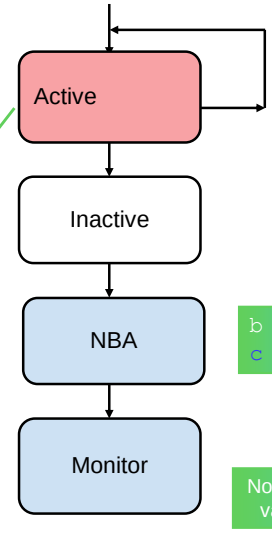
```
reg a=0, b=0, c=0, d=0;

 initial
  begin : P0
    #1;
    a = 1;
  end

always @(a or b)
  begin : P1
    b <= a;
    c <= b;
  end

always @(b)
  begin : P2
    d <= b;
  end
```

Active re-entered to execute P1. Nonblocking assignments to b and c scheduled.




```
b <= 1
c <= 0
```

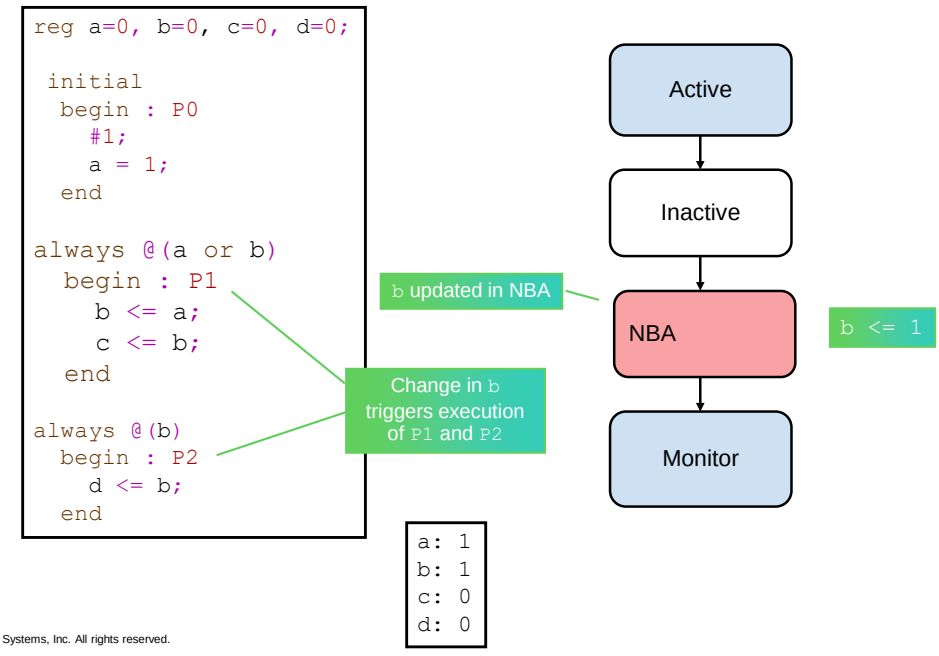
No change in value for c

```
a: 1
b: 0
c: 0
d: 0
```


cadence

The simulator re-enters the Active region to execute P1. P1 makes nonblocking assignments to b and c. These are scheduled for the NBA region. Remember nonblocking assignments are evaluated using the current values of variables, therefore c is assigned to the current value of b which is 0. This is not a new value for c, so the assignment can be ignored.

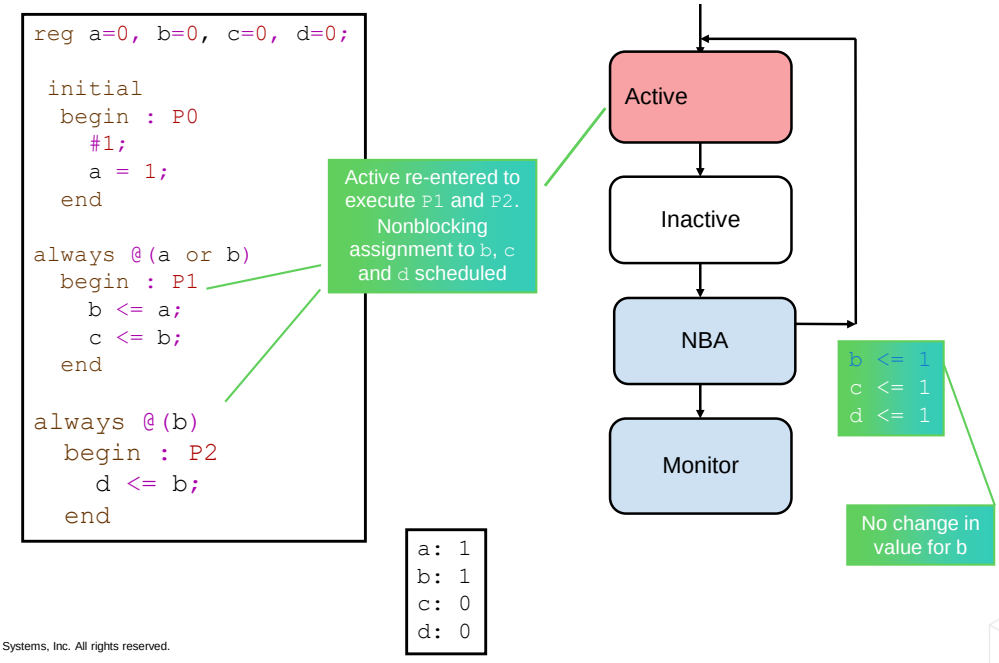
Simulation Cycle: 4/6



There are no more procedural blocks triggered by activity in the Active region, therefore the simulator can advance to the NBA region to make the scheduled assignment to b.

When the simulator has completed all the nonblocking assignments in the NBA region, it checks to see if any procedural blocks have been triggered by the assignments. Here a change in the value of b triggers P1 and P2.

Simulation Cycle: 5/6



The simulator re-enters the Active region to execute P1 and P2.

P1 makes nonblocking assignments to b and c. These are scheduled for the NBA region. Only c receives a new value, therefore the assignment to b can be ignored.

P2 makes nonblocking assignments to c. This is scheduled for the NBA region.

The execution loop of re-entering the Active region from the NBA region is called a delta cycle. There may be many delta cycles at a given time slot before the simulating reaches a steady state.

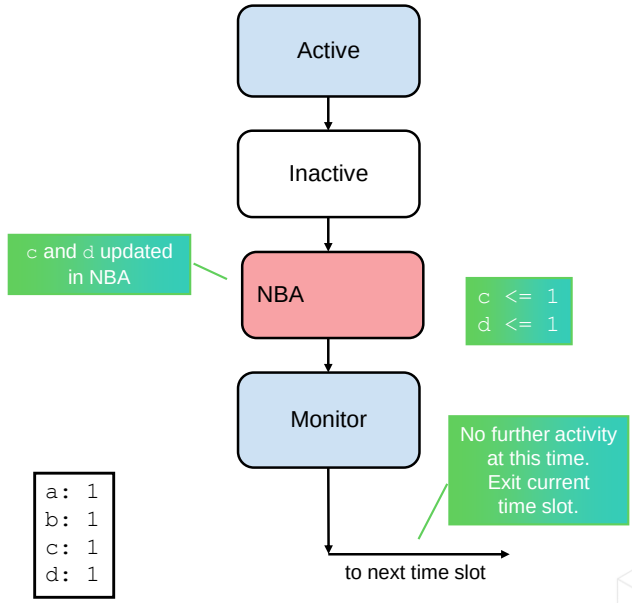
Simulation Cycle: 6/6

```
reg a=0, b=0, c=0, d=0;

initial
  begin : P0
    #1;
    a = 1;
  end

always @(a or b)
  begin : P1
    b <= a;
    c <= b;
  end

always @(b)
  begin : P2
    d <= b;
  end
```



There are no more procedural blocks triggered by activity in the Active region, therefore the simulator can advance to the NBA region to make the scheduled assignments to c and d.

When the simulator has completed all the nonblocking assignments in the NBA region, it checks to see if any procedural blocks have been triggered by the assignments. Here there are no triggered blocks, therefore the simulator can exit the current time slot.

Simulator time will advance to the next nearest event, or if no events are scheduled, the simulator will terminate.

Simulation Cycle Summary

```
reg a=0, b=0, c=0, d=0;

 initial
  begin : P0
    #1;
    a = 1;
  end

always @(a or b)
  begin : P1
    b <= a;
    c <= b;
  end

always @(b)
  begin : P2
    d <= b;
  end
```

```
...
always @(a or b)
  begin : P1
    b = a;
    c = b;
  end

 always @(b)
  begin : P2
    d = b;
  end
```

Cycle	Eval	Event	Update
cyc	P0	a=1	
	P1		b=1
cyc+1	P1 P2		c=1 d=1

Note P1 is executed twice to reach a steady state.

- For combinational code, blocking assignments are sufficient.
- Nonblocking assignments are not needed and thus inefficient.

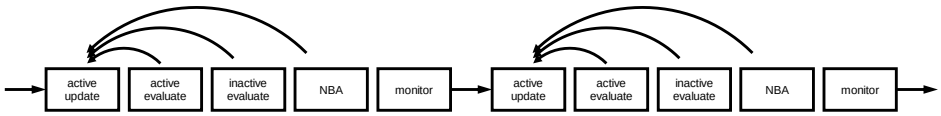
cadence

Here is a summary of the two delta cycles which propagate the transition of a to 1. The first delta cycle evaluates procedure P0, which generates an update event for the assignment to a. That update event triggers evaluation of procedure P1, which schedules a nonblocking update to b which occurs later in the same delta cycle. The nonblocking update to b triggers evaluation of procedures P1 and P2, which occur in the next delta cycle. These evaluations schedule nonblocking assignments to c and d that do not affect these procedures.

Synchronizing Procedures

You suspend execution of a procedural statement with:

- The @ event control (wait for simulation events)
- The wait level-sensitive event control (conditionally wait for events)
- The # delay control (wait for simulation time)
  - Schedules process resumption into the inactive evaluate queue for the current or some future simulation time



time τ_1 time τ_2

Timing controls are not statements. They apply to the one immediately subsequent statement.

Verilog provides procedural timing controls for stepping execution of procedural blocks. Timing controls are not statements. Timing controls apply to the one immediately subsequent statement. In a sequential block, this of course also delays all following statements. Because a timing control is not a statement, it cannot appear at the end of a block unless a statement follows it. Conveniently, that following statement may be a null statement. A null statement is simply a semicolon (;).

Synchronizing Procedures: Event Controls

Use an event control for edge-sensitive timing control in behavioral code:

```
@ event_identifier
@ ( event_expression )
@*
@ (*)
```

- An event is any transition of the specified ports, nets or variables.
- Execution resumes when any of the events occur.
- Verilog-2001 added the comma and wildcard operators.
  - The wildcard operator adds all signals read within the block and any arguments of functions called from the block.

```
module reg_adder (
  output reg [3:0] out,
  input      [2:0] a, b,
  input            clk
);
  reg [3:0] sum;

  always @(a or b)
    sum = a + b;

  always @(negedge clk)
    out <= sum;

endmodule
```

Wait for a transition on a or b

Wait for a positive transition on clk

```
initial
  begin
    repeat (12)
      begin
        @(posedge clk)
          tog <= ~tog;
      end
  end
```

Timing control can be anywhere in block!

Use an event control for edge-sensitive timing control in behavioral code. This is the most commonly used timing control. An event control starts with the at (@) character and then follows with either a wildcard character, a single event identifier, or a parenthesized event expression. The event expression can be a list of event expressions separated by the or keyword or by the comma (,) character. The or keyword in an event expression is a separator between event expressions and is not an operator in the usual sense. You can further qualify an expression with the posedge or negedge keywords.

This example uses a combinational block to maintain the value of an intermediate sum variable and a sequential block to assign the sum value to the output variable upon every negative edge of the clock.

- event_control ::=
  @ event_identifier
  | @ (event_expression)
  | @*
  | @ (*)
- event_expression ::=
  expression
  | hierarchical_identifier
  | posedge expression
  | negedge expression
  | event_expression or event_expression
  | event_expression , event_expression

Note: Wildcard event list adds all signals read within the block and any arguments of functions called from the block. If you read a signal in a function via side-effects (i.e., read directly and not passed by the argument list) then that signal is not included in the wildcard event list which is a feature of Verilog 2001.

Synchronizing Procedures: Level-Sensitive Event Controls

Use a wait statement for level-sensitive timing control in behavioral code:

```
wait (expr) statement
```

- When the statement executes:
  - If the expression is known and true, it does not suspend the process.
  - If the expression is unknown or false, it does suspend the process until the expression is known and true.

```
module latch_adder (
  output reg [3:0] out,
  input      [2:0] a, b,
  input            enb
);

  always @(a or b)
    begin
      // continue if/when enabled
      wait (enb)
        out = a + b;
    end

endmodule
```


Use a wait statement for level-sensitive timing control in behavioral code. If upon executing the wait statement the expression is already true the statement does not block. If upon executing the wait statement the expression is not true then the statement does block, and then unblocks if and when the expression becomes true.

In this example, the procedural block blocks upon every execution until an event occurs on its a or b inputs. When one of these events occurs, the procedural block tests the state of the enable (enb) input, and if the enable input is not 1 then the procedural block again blocks until the enable input becomes 1. While the procedural block is waiting for the enable input to become 1 it misses any further events on the a or b inputs, as it is not waiting for them.

- wait (expr) statement

The wait statement blocks if the expression is not true and then unblocks when the expression becomes true.

Synchronizing Procedures: Delay Control

Use a delay control for a simple time delay:

```
# delay_value
# ( expr )
# ( expr:expr:expr )
```

min:typ:max

- For modeling propagation delay
- For generating a clock
- For stepping the testbench

```
wire a, b; reg y;
...
always @(a or b) #10 y = a & b;
```

```
module clock (output reg clk);
  parameter PERIOD = 10;

  initial clk = 0;
  always #(PERIOD/2) clk = ~clk;

endmodule
```

```
reg a, b;
...
initial
  begin
    // wait 10 time units after
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
  end
```


cadence

Use a delay control for a simple time delay for modeling propagation delay, generating a clock and stepping the testbench. If the delays are constants then you can use module parameters to propagate the constants through the hierarchy.

---

delay_value

(expr)

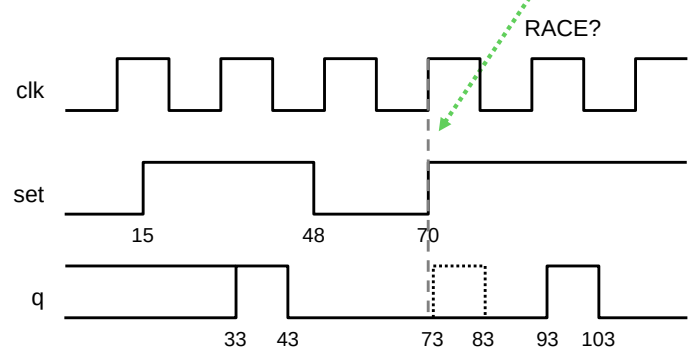
(expr:expr:expr)

Example Timing Controls

```
// clock
parameter PERIOD = 20;
reg clk = 0;
always #(PERIOD/2)
  clk = ~clk;
```

```
// stimulus
reg set = 0;
initial
  begin
    #15 set = 1;
    #33 set = 0;
    #22 set = 1;
  end
```

```
// response
reg q;
always
  begin
    wait (set);
    @(posedge clk);
    #3  q = 1;
    #10 q = 0;
    wait (!set);
  end
```



This example illustrates a collection of timing controls.

The response procedure does the following:

- 1st: It waits for “set” to be 1, ignoring the positive clock edge at time 10.
- 2nd: After “set” is 1 at time 15, it waits for the positive clock edge.
- 3rd: After the positive clock edge at time 30 it waits 3 time units before setting “q” to 1.
- 4th: After setting “q” to 1 at time 33 it waits 10 time units before setting “q” to 0.
- 5th: After setting “q” to 0 at time 43 it waits for “set” to be 0.
- 6th: After “set” is 0 at time 48 it again waits for “set” to be 1.
- 7th: “set” is 1 at time 70, coincident with the positive clock edge. Both assignment statements are on the active evaluate queue. Update to the “set” variable unblocks the response procedure and places it on the active evaluate queue, normally but not necessarily at the end of the queue, so the response procedure normally resumes execution after the positive clock edge has occurred and must wait for the next positive clock edge.

This illustration provides good arguments for synchronizing stimulus to the clock:

- 1st – The stimulus does not utilize the clock period constant. The stimulus causes greatly different results upon changing the clock period.
- 2nd – The stimulus is not synchronized to the clock. If the stimulus is synchronized to the clock, then we know that when the “set” signal transitions, the clock has already occurred and no race condition can exist.

Compiler Directive `timescale Preview

The `timescale compiler directive specifies the time unit and time precision for all following modules.

```
`timescale unit/precision
```

- Can appear only outside module descriptions.
- Time values in a module are rounded to the precision of that module.
- The overall simulation time precision is the smallest precision that any module specified.

```
`timescale 1ns / 100ps

module test;
reg a, b;
...
initial
  begin
    // wait 10 ns after
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
  end
...
```

Not a statement – no semicolon

175

cadence

The `*timescale* directive specifies the time unit and time precision for subsequent module declarations. Any expression used in a context to mean simulation time will assume the time unit of the current module and be rounded to the time precision of the current module. System tasks that display simulation time will scale the display to the time unit of the current module.

As the standard does not prescribe a default time scale, for interoperability, either all modules must be subject to a time scale or no module can be subject to a time scale. To ensure that all modules are subject to a time scale, you should get used to specifying a time scale at the top of every file, even if you not use time specifications in the file. To modify a time scale, you need to edit the file and recompile that file and all other files that utilized that directive. No method exists to modify a time scale during simulation.

```
`timescale time_unit / time_precision
```

- The *time_unit* argument specifies the assumed units for time values.
- The *time_precision* argument specifies the precision for time values.
- The *time_precision* argument cannot be larger than the *time_unit* argument.
- Time values are rounded to meet the current precision specification.
- The overall simulation time precision is the smallest specified time precision.
- The integer part of the arguments specify a magnitude (1, 100, 1000} and the character string argument suffix is the unit {*s*, *ms*, *us*, *ns*, *ps*, *fs*}.

Module Summary

Now you can produce higher quality code less affected by non-determinism and race conditions.

This module described:

- Procedural **initial** and **always** blocks
- Blocking **=** procedural assignment
- Nonblocking **<=** procedural assignment
- The simulation cycle (delta cycles)
- Timing control:
  - The @ event control
  - The **wait** level-sensitive event control
  - The # delay control

Your objective is to produce higher quality code less subject to indeterminacy and race conditions. To do that, you need to understand something about how the simulator executes processes and their statements.

Module Review

1. When does the variable value change when using a: (i) blocking assignment? (ii) nonblocking assignment?
2. What are the three Verilog timing controls?
3. What Verilog construct do you use in a procedure to advance simulation time?
4. Where can you place the `timescale compiler directive?

cadence

This page does not contain notes.

Module Review Solutions

1. When does the variable value change when using a: (i) blocking assignment? (ii) nonblocking assignment?
   - A blocking assignment blocks execution of subsequent statements of the procedure until the assignment completes, which is by default immediately.
   - A nonblocking assignment schedules assignment completion for the “update” phase of a delta cycle, by default the current cycle.
2. What are the three Verilog timing controls?
   - The @ event control, the wait level-sensitive event control, the # delay control.
3. What Verilog construct do you use in a procedure to advance simulation time?
   - The # delay control suspends the procedure for a specified simulation time.
4. Where can you place the `timescale compiler directive?
   - In the source code outside any module definition. The Verilog standard implies but does not explicitly specify this by saying it applies to “modules that follow” and always showing it outside example modules.

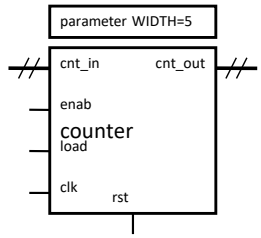
cadence

This page does not contain notes.

Lab

Lab 9-1 Modeling a Generic Counter

- Use blocking and nonblocking assignments while describing a counter.



cadence

Your objective is to produce high-quality deterministic code.

For this lab, you use blocking and nonblocking assignments while describing a counter:

- You place combinational behavior in a combinational procedure.
- You place sequential behavior in a sequential procedure.