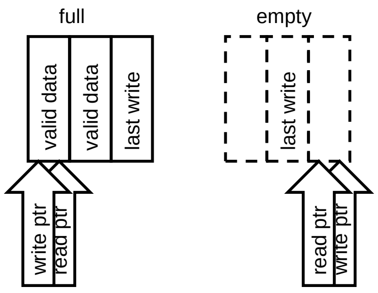
FIFO Outputs

Derive the FIFO status from the pointer addresses.

- Full – Addresses are equal and last operation was write
- Empty – Addresses are equal and last operation was read.

For read:

- Continuously drive current data.



```
// functional code
...


assign empty = (rptr == wptr) && !wrote;
assign full  = (rptr == wptr) &&  wrote;

endmodule
```

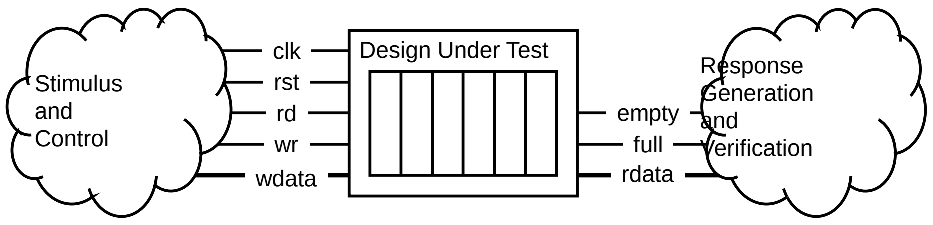
Continuously present to the data output port the register array data addressed by the read pointer.

Generate the FIFO status flags by comparing the read and write addresses. At any point where the read address is the same as the write address, you know that the FIFO is either empty or full. It is full if the most recent operation was to write and empty if the most recent operation was to read.

Note that signal “wrote” in the code is “last write” performed to the FIFO.

FIFO Testbench

- Instantiates and connects to the Design Under Test (DUT)
- Applies stimuli to the DUT input data and control ports
- Monitors the DUT output ports to verify correct behavior



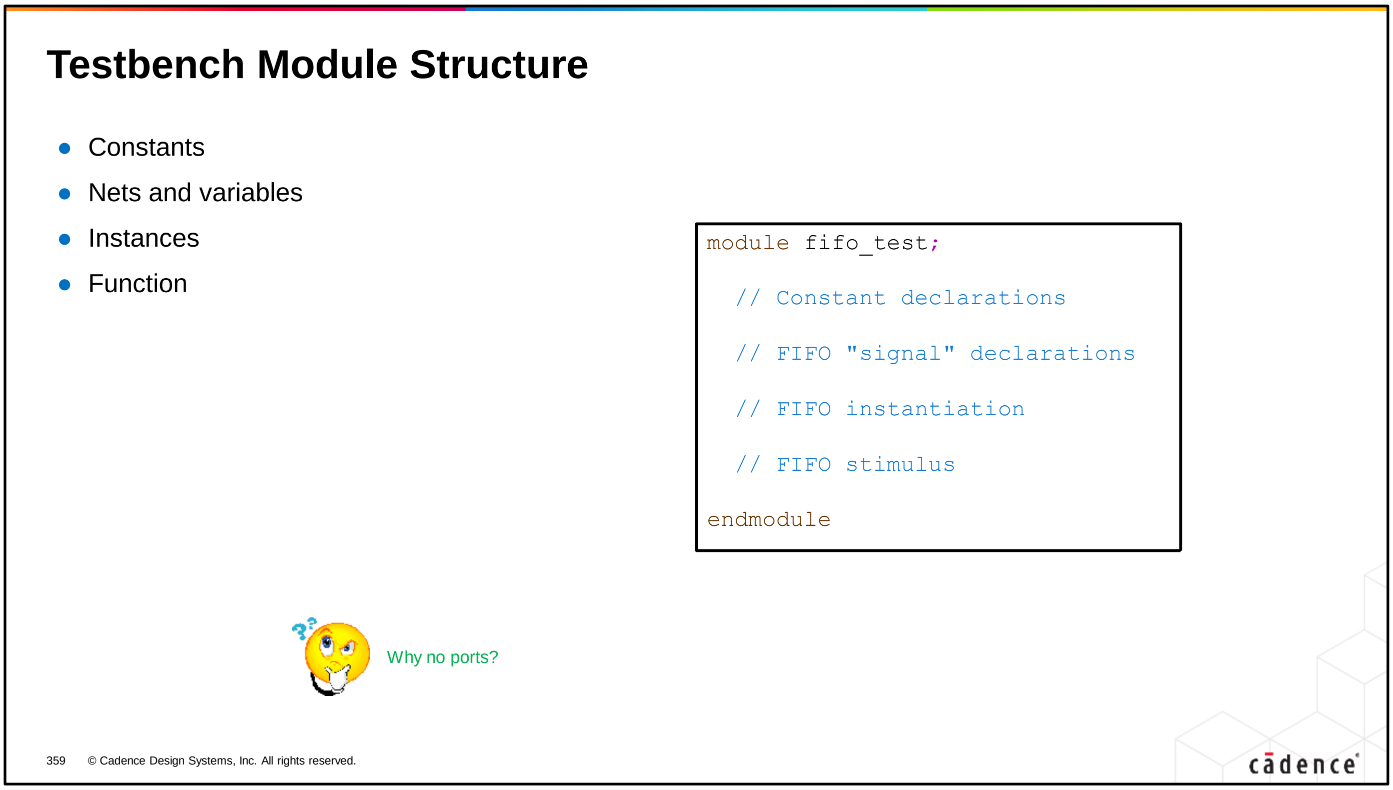
To verify your FIFO model, you need to present it with stimulus and verify that its response is correct. In general, any testbench must do the following:

- Instantiate the Design Under Test (DUT) and connect its ports to local nets and variables.
- Apply stimuli to the input data and control (clock, reset) ports.
- Monitor the output ports to capture the response of the design to the applied stimuli.

A simple testbench records the stimulus and response vectors for subsequent analysis. Such analysis can for this design be a simple visual check of the transition data.

A more sophisticated testbench automatically compares the response to an internally generated response that is known to be correct, and reports whether the DUT failed or passed the test.

Modern testbenches tend to be extremely sophisticated. The majority of development effort is now utilized for verification.



Testbench Module Structure

- Constants
- Nets and variables
- Instances
- Function

```
module fifo_test;

  // Constant declarations

  // FIFO "signal" declarations

  // FIFO instantiation

  // FIFO stimulus

endmodule
```

Why no ports?

359 © Cadence Design Systems, Inc. All rights reserved.

cadence

The top level of the simulation model is a testbench module. The top-level module does not have module parameters or ports because it is not instantiated (otherwise it would not be the top level). The top-level module declares design and test configuration constants, signals to connect to the DUT instance, and the DUT instance. A top-level module can declare procedures to provide stimulus and monitor response, and can instantiate additional test modules to also do some of that work

Testbench Declarations

- Testbench constants
- FIFO *signals*

```
module fifo_test;

  // Verilog-2001 local constants
  localparam dwidth = 8;
  localparam awidth = 5;
  localparam depth = 2 ** awidth;

  // Variables for FIFO inputs
  reg [dwidth-1:0] wdata;
  reg clk, wr, rd, rst;

  // Nets for FIFO outputs
  wire full;
  wire empty;
  wire [dwidth-1:0] rdata;

  // FIFO instantiation
  // FIFO stimulus

endmodule
```

Declare local constants for the FIFO data width and address width and clock period.

Using these constants, declare variables to provide stimulus and nets to receive the response.

Testbench DUT Instance

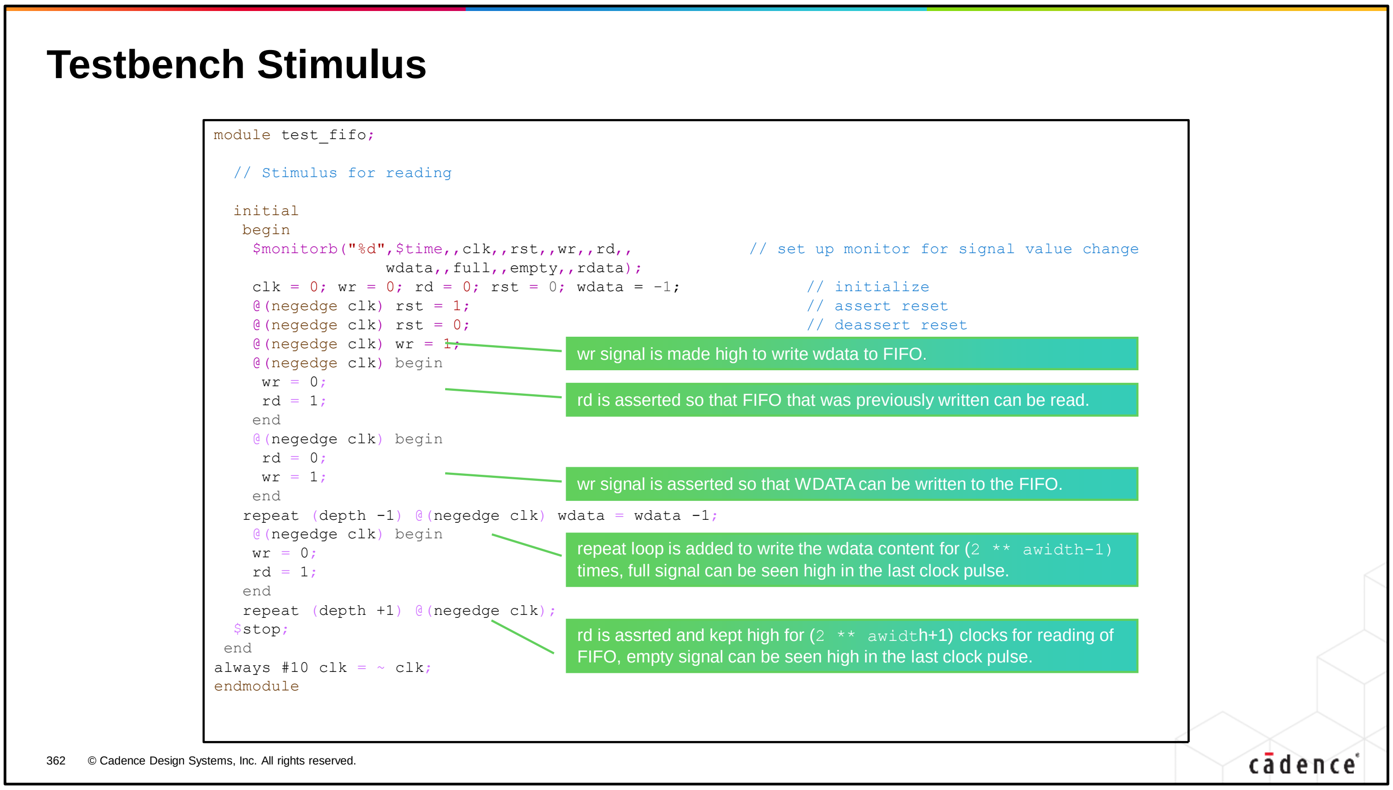
- The module definition name is fifo.
- Override the module parameters.
- The module instance name is dut.
- Use named port connections to bind FIFO ports to local *signals.*

```
module test_fifo;

  // Declarations

  // FIFO instantiation
  fifo
  #(
    .awidth(awidth),
    .dwidth(dwidth)
   )
  fifo
   (
    .wdata  (wdata),
    .rdata (rdata),
    .clk      (clk),
    .rst      (rst),
    .wr    (wr),
    .rd    (rd),
    .full     (full),
    .empty    (empty)
   );

  // FIFO stimulus

endmodule
```

Verilog-2001 list of named parameter assignments

Verilog-1995 list of named port connections

Instantiate the fifo DUT and override its module parameters and connect its ports to local nets and variables.



Testbench Stimulus

```
module test_fifo;

  // Stimulus for reading

  initial
   begin
    $monitorb("%d",$time,,clk,,rst,,wr,,rd,,            // set up monitor for signal value change
                  wdata,,full,,empty,,rdata);
    clk = 0; wr = 0; rd = 0; rst = 0; wdata = -1;             // initialize
    @(negedge clk) rst = 1;                                   // assert reset
    @(negedge clk) rst = 0;                                   // deassert reset
    @(negedge clk) wr = 1;
    @(negedge clk) begin
     wr = 0;
     rd = 1;
    end
    @(negedge clk) begin
     rd = 0;
     wr = 1;
    end
   repeat (depth -1) @(negedge clk) wdata = wdata -1;
    @(negedge clk) begin
    wr = 0;
    rd = 1;
   end
   repeat (depth +1) @(negedge clk);
  $stop;
 end
always #10 clk = ~ clk;
endmodule
```

wr signal is made high to write wdata to FIFO.

rd is asserted so that FIFO that was previously written can be read.

wr signal is asserted so that WDATA can be written to the FIFO.

repeat loop is added to write the wdata content for (2 ** awidth-1) times, full signal can be seen high in the last clock pulse.

rd is assrted and kept high for (2 ** awidth+1) clocks for reading of FIFO, empty signal can be seen high in the last clock pulse.

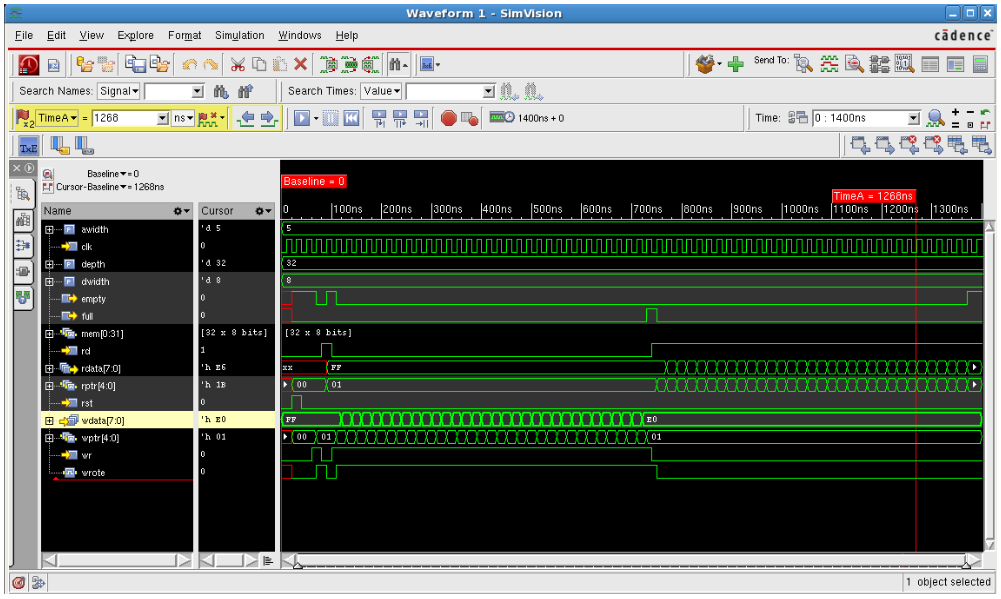
The $monitor family of system tasks output the value of their arguments at the end of each unique simulation time in which any of them has changed.

As the fifo DUT is synchronized to the rising clock edge, the stimulus is synchronized to the falling clock edge. Having done that, the stimulus can utilize blocking assignments without causing clock/data races.

As is typical for a testbench, the stimulus tests the “corner” conditions of the DUT algorithm.

The $stop system task stops the simulation.

Simulation Results



Baseline = 0
Cursor-Baseline = 1268ns

Name	Cursor
awidth	'd 5
clk	0
depth	'd 32
dwidth	'd 8
empty	0
full	0
mem[0:31]	[32 x 8 bits]
rd	1
rdata[7:0]	'h E6
rptr[4:0]	'h 1B
rst	0
wdata[7:0]	'h E0
wptr[4:0]	'h 01
wr	0
wrote	0

Here is a Waveform window of the Cadence SimVision graphical simulation analysis environment displaying the transition history of the testbench signals.

Module Review

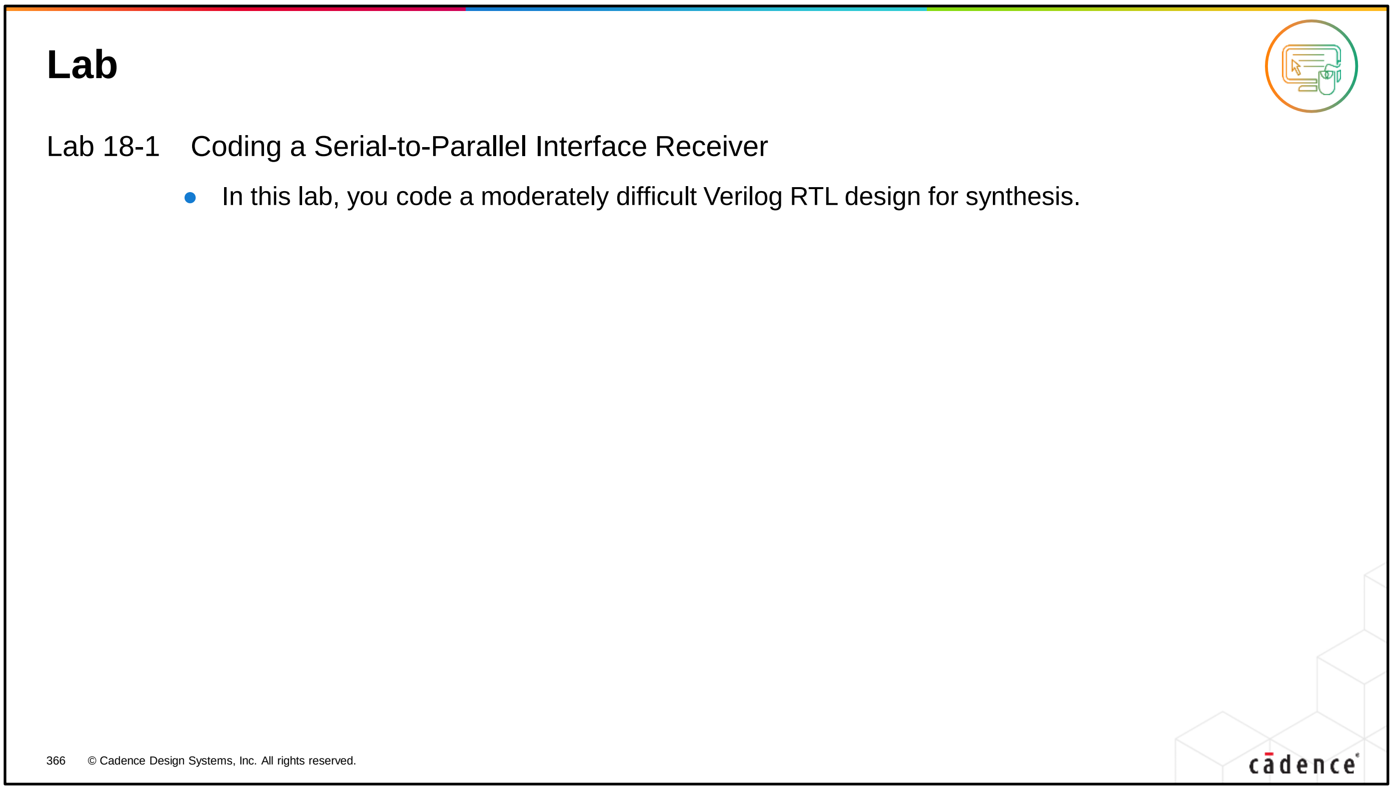
1. What are module parameters?
2. What are local parameters?
3. How do you override the value of a module parameter?
4. Why do we size literals used in arithmetic operations?

This page does not contain notes.

Module Review Solutions

1. What are module parameters?
   - Module parameters are constants such as width and depth that you can override on a per-instance basis.
2. What are local parameters?
   - Local parameters are constants that you cannot override.
3. How do you override the value of a module parameter?
   - You override module parameter values as you instantiate the module by providing a list of parameter values.
4. Why do we size literals used in arithmetic operations?
   - Unsized literals are 32-bit values. A logic synthesis tool may initially assume that you need for example a 33-bit adder when adding an unsized literal to a much smaller value.

This page does not contain notes.



Lab

Lab 18-1 Coding a Serial-to-Parallel Interface Receiver

- In this lab, you code a moderately difficult Verilog RTL design for synthesis.

366 © Cadence Design Systems, Inc. All rights reserved.

cādence

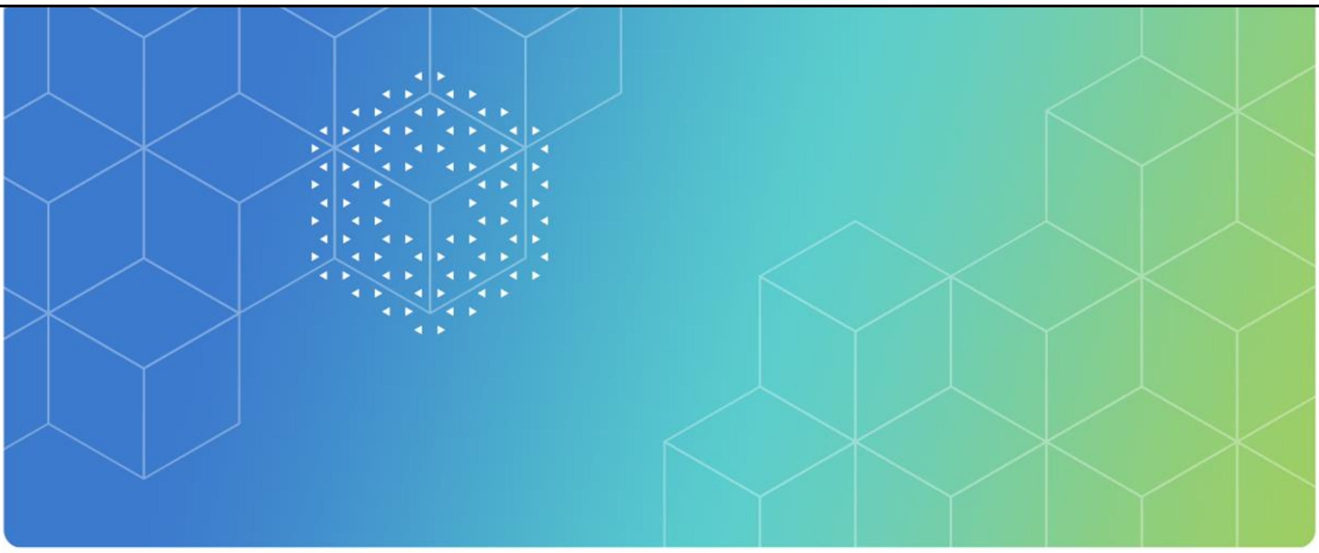
Your objective for this lab is to resolve a deadlock between two devices competing for two resources.

You will frequently encounter the situation where the test environment waits an undetermined time for the system being tested to respond to some stimulus. The system may malfunction such that it does not ever respond. The test must anticipate this failure mode and report it.

The lab model is an arbiter that arbitrates between two users of a resource. The arbiter prioritizes the requests for the resource. The lab does not model the resource.

The test environment is two instances of the arbiter and two instances of the device making the requests. The device at random intervals requires one or both resources. It requests the first resource, and upon being granted the first resource, sometimes also requests a second resource. Due to the random nature of the requests, the simulation will invariably eventually come to a point at which both devices have one resource and cannot continue until they have the second resource – hence both become “deadlocked”.

For this lab, you modify the requesting device so that after a reasonable amount of time waiting for the second resource, it cancels both current requests.



Module 19

Using Verification Constructs

This page does not contain notes.

Module Objective

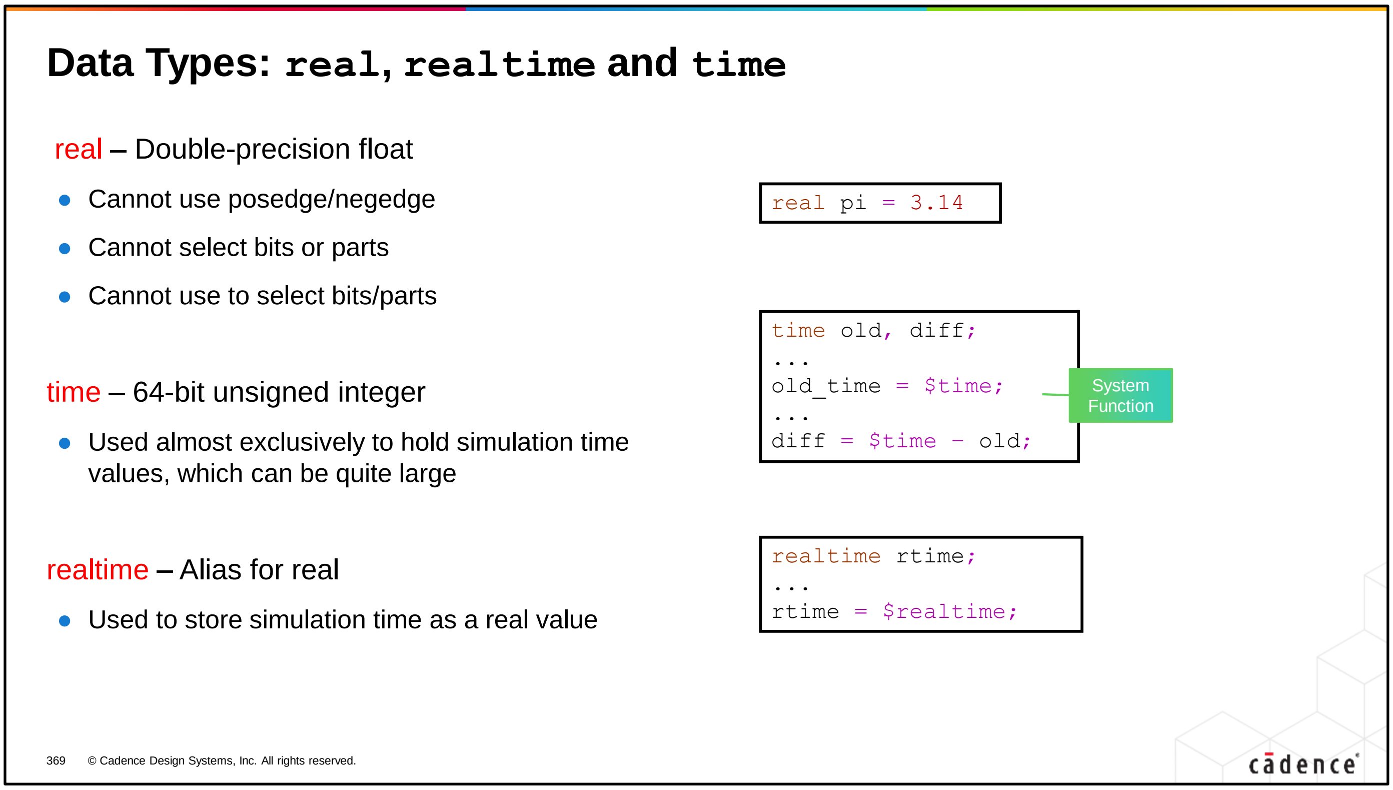
In this module, you:

- Choose and use Verilog verification constructs effectively

Topics

- Data types: real, realtime, time
- Case equality operators:
  ===, !==
- Procedural continuous assignments: force, release
- Looping statements:
  forever, repeat, while
- Named events: event, ->
- Level-sensitive event control: wait
- Parallel blocks: fork, join
- Initial construct: initial
- Disabling tasks and named blocks: disable

This page does not contain notes.



Data Types: `real`, `realtime` and `time`

real – Double-precision float

- Cannot use posedge/negedge
- Cannot select bits or parts
- Cannot use to select bits/parts

```
real pi = 3.14
```

time – 64-bit unsigned integer

- Used almost exclusively to hold simulation time values, which can be quite large

```
time old, diff;
...
old_time = $time;
...
diff = $time - old;
```

System Function

realtime – Alias for real

- Used to store simulation time as a real value

```
realtime rtime;
...
rtime = $realtime;
```

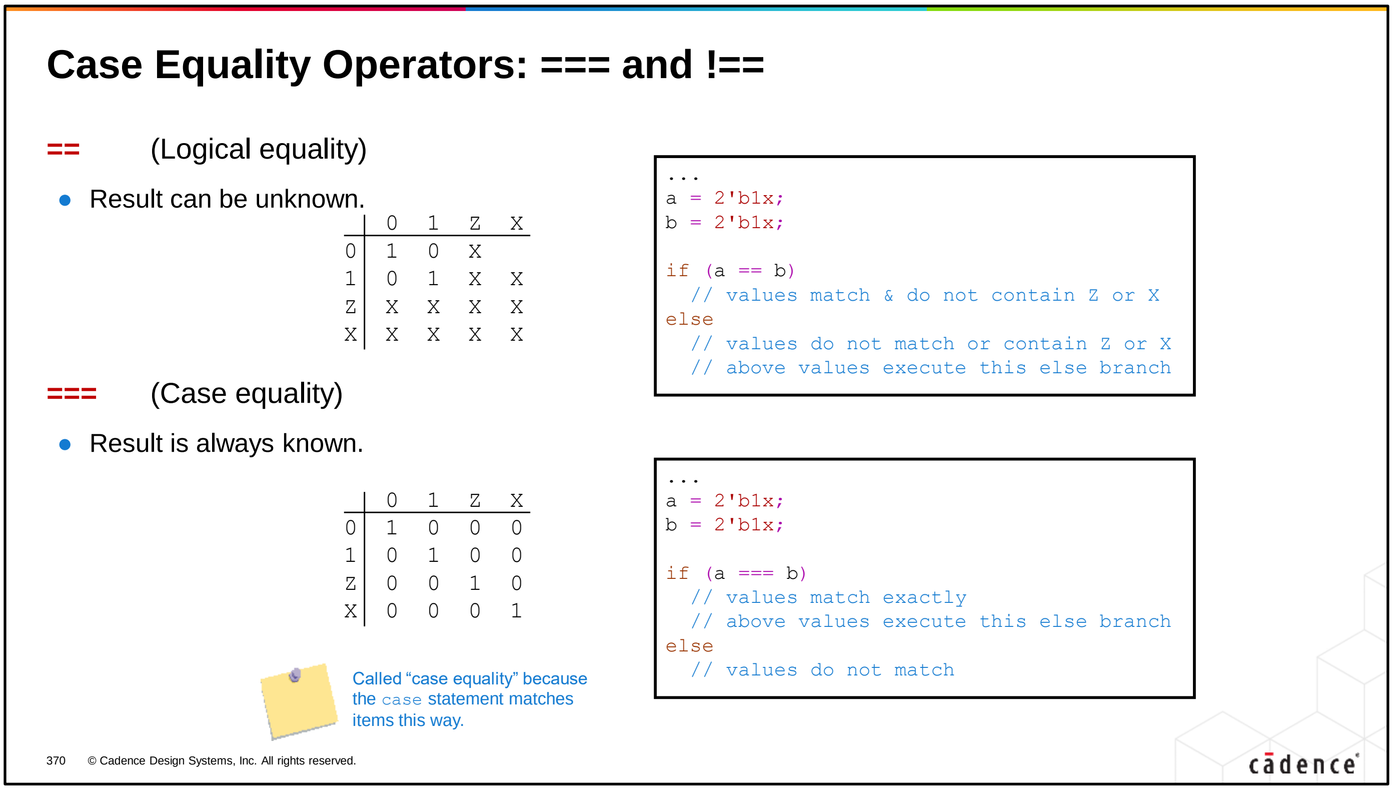
369 © Cadence Design Systems, Inc. All rights reserved.

cādence

A real variable holds a double-precision floating-point value. The simulator converts real values to integral values by rounding. Some operators you cannot use with real variables or values. For example, you cannot determine the posedge or negedge of a real variable, and you cannot select a bit or part of a real variable or real value and you cannot use a real variable or real value to select a bit or part of a variable or value.

A time variable holds a 64-unsigned integer value. People typically use it to hold simulation time values, perhaps for diagnostic and debugging purposes, but you can use it for other purposes as well.

The realtime type is an alias for the real type. People typically use it to hold simulation time as a double-precision floating-point value. Calling it a realtime variable helps them remember what they are using it for.



Case Equality Operators: === and !==

== (Logical equality)

- Result can be unknown.

	0	1	Z	X
0	1	0	X	
1	0	1	X	X
Z	X	X	X	X
X	X	X	X	X

```
...
a = 2'b1x;
b = 2'b1x;

if (a == b)
  // values match & do not contain Z or X
else
  // values do not match or contain Z or X
  // above values execute this else branch
```

=== (Case equality)

- Result is always known.

	0	1	Z	X
0	1	0	0	0
1	0	1	0	0
Z	0	0	1	0
X	0	0	0	1

```
...
a = 2'b1x;
b = 2'b1x;

if (a === b)
  // values match exactly
  // above values execute this else branch
else
  // values do not match
```

Called "case equality" because the `case` statement matches items this way.

cādence

You recall that the equality operator returns a 1-bit result representing the truth of the comparison. If the equality operator cannot determine the truth of the comparison, it returns the unknown value.

The case equality operator, so called because it performs the comparison as the case statement matches item expressions, performs a definitive match for bit positions that contain high-impedance or unknown values. The case equality operator always returns a 0 or 1 result – the result is never unknown.

You will find the case equality operator to be very useful in a testbench. Let us suppose that you have a task that uses an if statement testing an expression that utilizes the logical inequality operator to compare the DUT response with the expected response. If the DUT response contains any nonbinary values, the operator returns the unknown value, so fails to execute the subsequent statement that would presumably alert you to the miscompare. In this situation, the case inequality operator does exactly what you need.

```
if ( response != expected )
  begin // not executed if response is unknown!
```

Procedural Continuous Assignments: `force` and `release`

force – Forces a net or variable

- Single whole net or variable
- Bit or part of vector net
- Concatenation of above
- Not bit or part of vector variable
- Not array word

release – Releases a force

- Net returns to assigned value
- Variable retains forced value until next assignment

```
initial // QUICKDIRTYPATCH
 force rts = rst;
```

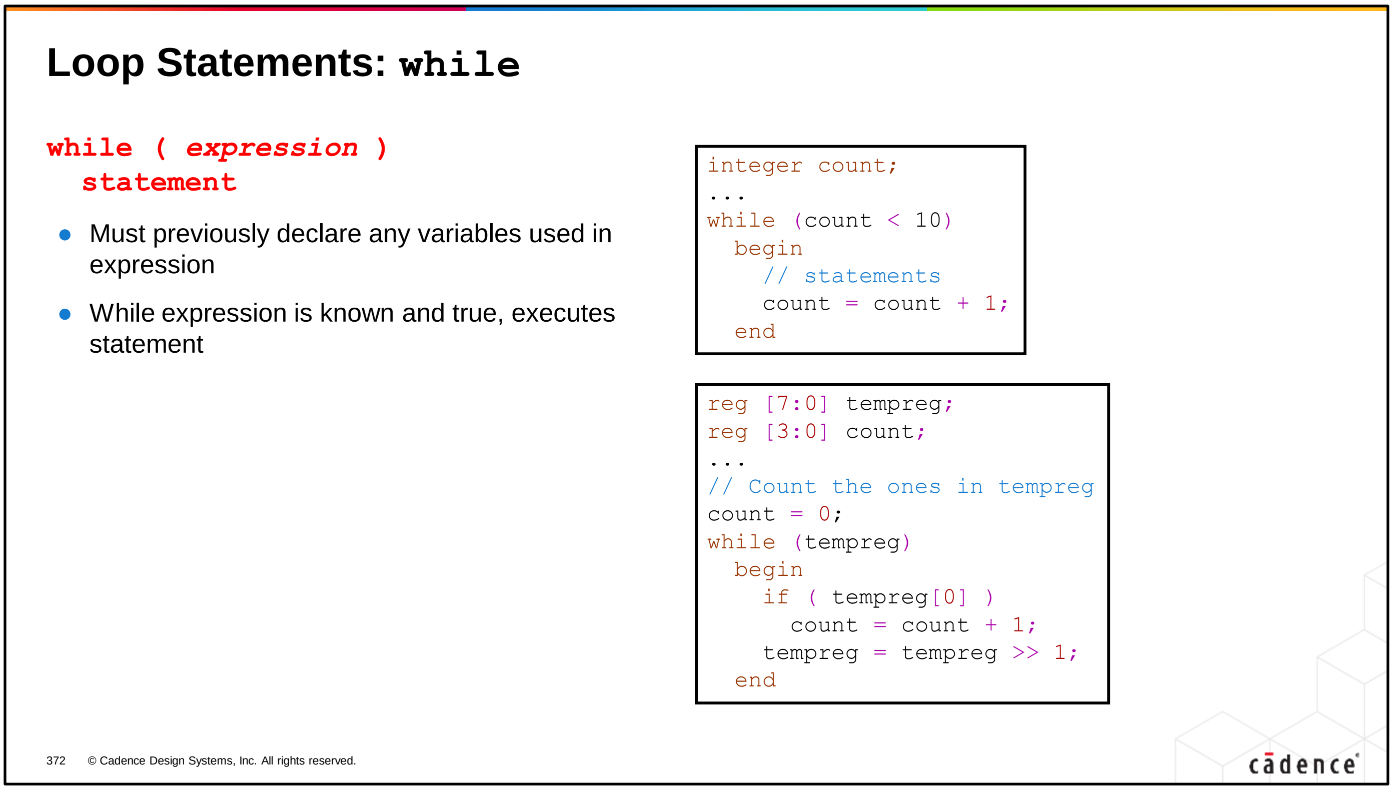
The force statement procedurally sets up a continuous assignment of an expression to a net or variable. It is equivalent to a continuous assignment, but as you do it procedurally, you can also apply it to variables. You can also apply it to a constant bit-select of a vector net, a part-select of a vector net, or a concatenation. You cannot apply it to a bit-select or a part-select of a vector variable, or to an array word. The procedural continuous assignment overrides any other assignment to the net or variable until it is released or another procedural continuous assignment is made.

The release statement releases the forced continuous assignment. A net that is subject to a continuous assignment immediately resumes that value. A variable retains the forced value until it is next procedurally assigned.

The force statement is an anachronism. Early simulators provided an interpretive user interface that accepted interactive procedural statements from the keyboard. Verification personnel used the force statement interactively as a temporary patch mechanism during their debug efforts. You would be very unlikely to see it in source code as you do here.

This example forces the rts signal to follow the rst signal. The user hypothesized that the rts signal was implicitly declared when connected to an instance port, but spelled incorrectly. The user temporarily forced it to follow the rst signal to prove the hypothesis.

To prevent such implicit net declarations, use the `default_nettype compiler directive to set the default net type to none.



Loop Statements: `while`

```
while ( expression )
   statement
```

- Must previously declare any variables used in expression
- While expression is known and true, executes statement

```
integer count;
...
while (count < 10)
  begin
    // statements
    count = count + 1;
  end
```

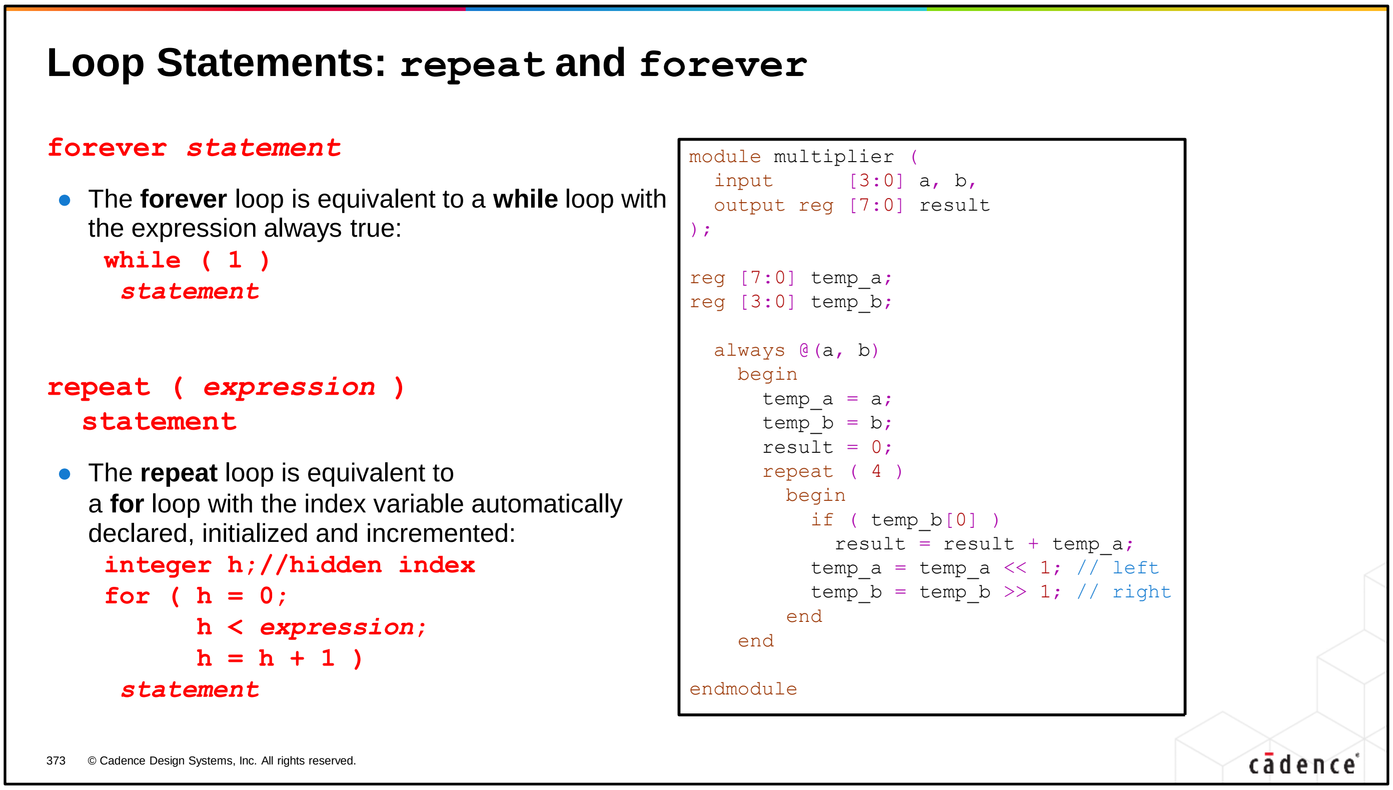
```
reg [7:0] tempreg;
reg [3:0] count;
...
// Count the ones in tempreg
count = 0;
while (tempreg)
  begin
    if ( tempreg[0] )
      count = count + 1;
    tempreg = tempreg >> 1;
  end
```

cādence

The while loop executes a statement while its expression is true.

The first example tests a counter and executes a statement while the count is less than 10.

The second example counts the number of 1 bits in a bit mask.



Loop Statements: `repeat` and `forever`

```
forever statement
```

- The **forever** loop is equivalent to a **while** loop with the expression always true:

```
while ( 1 )
 statement
```

```
repeat ( expression )
  statement
```

- The **repeat** loop is equivalent to a **for** loop with the index variable automatically declared, initialized and incremented:

```
integer h;//hidden index
for ( h = 0;
      h < expression;
      h = h + 1 )
 statement
```

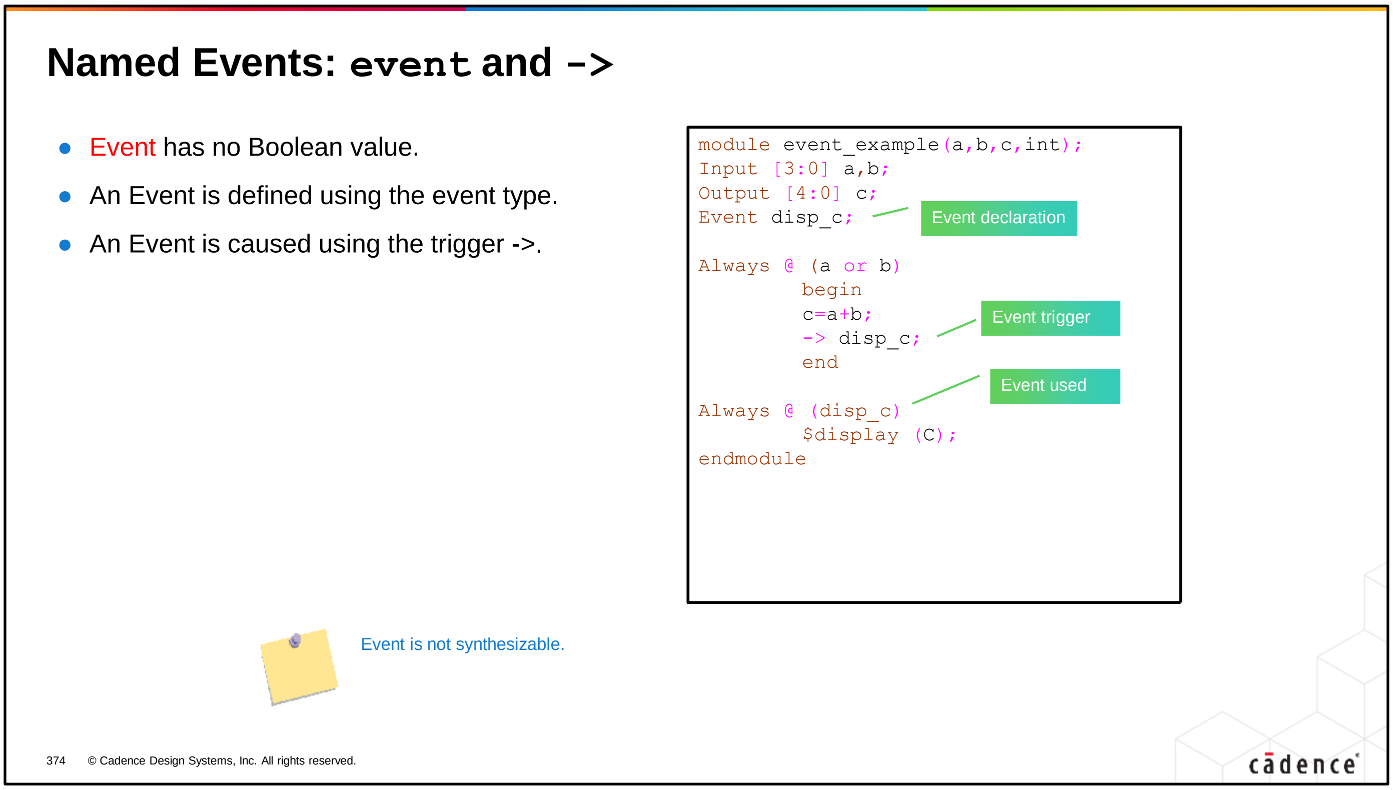
```
module multiplier (
  input      [3:0] a, b,
  output reg [7:0] result
);

reg [7:0] temp_a;
reg [3:0] temp_b;

  always @(a, b)
    begin
      temp_a = a;
      temp_b = b;
      result = 0;
      repeat ( 4 )
        begin
          if ( temp_b[0] )
            result = result + temp_a;
          temp_a = temp_a << 1; // left
          temp_b = temp_b >> 1; // right
        end
    end

endmodule
```

373 © Cadence Design Systems, Inc. All rights reserved.

The forever loop is equivalent to a while loop with its expression always true.

The repeat loop executes a statement the number of times by the value of the parenthesized expression.

This example implements a multiplication algorithm that iteratesspecified through the bit width of its inputs.



Named Events: `event` and `->`

- Event has no Boolean value.
- An Event is defined using the event type.
- An Event is caused using the trigger ->.

```
module event_example(a,b,c,int);
Input [3:0] a,b;
Output [4:0] c;
Event disp_c;

Always @ (a or b)
        begin
        c=a+b;
        -> disp_c;
        end

Always @ (disp_c)
        $display (C);
endmodule
```

Event declaration
Event trigger
Event used

Event is not synthesizable.

The event is a very efficient Verilog type because it does not have a Boolean value nor does it need to be scheduled. It is used mostly in testbenches.

To trigger procedures in a behavioral design or testbench, if you do not need to communicate a value, use a named event instead of a variable. This prevents your co-workers from having to guess whether you actually used the value anywhere. The simulator handles named events more efficiently than variables, as it does not have to consider the value.

Level-Sensitive Event Control: `wait`

```
wait ( expression )
  statement
```

- The **wait** statement waits for the expression to become true.
- It does not wait if the expression is already true.

```
module runReqAck (
 input run,ack,rst,
 output reg req
);
 always @(posedge run)
  begin
   req = 1;
   wait (ack || rst)
   req = 0;
   if (!rst)
    send_stuff;
  end
endmodule
```

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

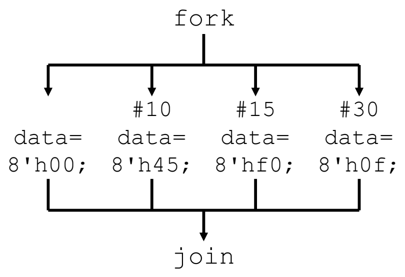
This example responds to the run positive edge by asserting the req output and waiting asynchronously for either the ack input or the rst input, upon which it deasserts the req output, and if the reset input is not true, calls the send task.

Parallel Blocks: `fork` and `join`

fork…join – Concurrent statements

- Enclosed statements start executing immediately.
  - Start in any order.
- Each statement is subject to only its own timing controls.
- The statement after join executes when all forked statements complete.

```
module forkjoin_tb;
 reg [7:0] data;
 // instance of DUT
 initial
  fork
        data = 8'h00;
   #10 data = 8'h45;
   #15 data = 8'hf0;
   #30 data = 8'h0f;
  join
endmodule
```



Statements enclosed by the fork...join construct execute concurrently, that is, they all start together at the moment simulation enters the block, and thereafter each statement is subject to only its own timing controls.

The parallel block completes when all forked statements have completed.

You can use a parallel block to start multiple concurrent processes. One common use is to start one process that takes an undetermined amount of time to complete and to start a second process as a watchdog timer that issues a message and disables the first process if the first process takes too long to complete.

This example forks four stimulus statements to apply data at different offsets from their starting points. Here the offsets are ordered for readability, but any such ordering is not necessary.

More About Parallel Blocks

A parallel block can contain any procedural statement of any complexity.

- Task calls, loops, other blocks…

Issues

- You cannot predict the execution order of statements scheduled to execute at the same simulation time. This can cause race conditions.
- If any statement (e.g., forever loop) does not complete, the block never joins.

```
module concurrent_tb;
 reg [7:0] data;
 // instance of DUT
 initial
  fork
   #10 data = 8'h45;
   #20 repeat (7)
    #10 data = data + 1;
   #25 repeat (5)
    #20 data = data << 1;
   #99 data = 8'h0f;
  join
endmodule
```

A parallel block is a set of statements that the stimulator starts executing simultaneously at the moment simulation enters the block. Each of those parallel statements is then subject to only its own timing controls and can be of unlimited complexity. Each can further contain parallel and sequential subblocks.

In this example, the two repeat loops start at different times and execute concurrently. This stimulus would be very difficult to apply in a single sequential block.

`initial` and `always` Constructs

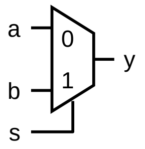
always – Executes *always*

- Starts at beginning of simulation
- Continually loops
  - Must have a timing control
- Hardware construct

initial – Executes *initially*

- Starts at beginning of simulation
- Executes exactly once
- Testbench construct

```
module mux (
 input a,b,s,
 output reg y );
 always @*
  y = s? b : a;
endmodule
```



```
module mux_tb;
 wire a,b,s,y;
 mux m(a,b,s,y);
 reg [2:0] stim=0;
 assign {a,b,s}=stim;
 initial
  #1 repeat(7)
   begin
    stim=stim+1;
    #1;
   end
endmodule
```

An always construct executes continuously. It always contains at least one timing control so that it blocks at some point to let the simulator do something else. The always construct is ideal for describing the behavior of hardware that always reacts to input transitions. If it describes hardware for synthesis, then it has only one event control, and places it immediately after the always keyword.

The initial construct executes exactly once, starting at the beginning of the simulation. It can contain any number of timing controls, including none at all. The initial construct steps through the timing controls, and when it executes its last statement, it is finished and never runs again during that simulation session. The initial construct is ideal for describing the behavior of a testbench that applies each stimulus just once.

You can also use the initial construct to describe system-level behavior that you do not want to synthesize. After executing statements to initialize various parts of the system, you can execute a forever loop in which you continually generate randomized stimulus.

This example generates stimulus within an initial construct. In a repeat loop it generates all possible binary combinations of multiplexor inputs.

Disabling Tasks and Named Blocks: `disable`

disable – Disables task or named block

- Activity associated with task or named block ceases.
- Execution continues with statement after block or task call.
- Subsequent statement execution can immediately again call task or re-enter block.

```
module busif_tb;
 ...
 always #50 clk = ~clk;
 initial
  begin: STIMULUS
   repeat(5) @(negedge clk);
   wait(!interrupt);
   cpu_driver(8'h00);
   wait(!interrupt);
   cpu_driver(8'haa);
   ...
  end
 ...
 always @(posedge interrupt)
  begin
   disable cpu_driver;
   service_interrupt;
  end
endmodule
```

The disable statement terminates activity associated with the task or named block. You can use it to work around the lack of C-like break and continue statements. You can use it to behaviorally model asynchronous activity such as interrupts.

The simulator discontinues any activity associated with the task or named block. If executing the task or named block, it resumes execution with the statement after the task call or after the named block. Implementations can choose whether to remove scheduled nonblocking assignments and whether to discontinue procedural continuous assignments, so this can be a source of differences between simulators.

Disabling a task or named block has no persistence. Subsequent statement execution can immediately again call the task and execution can immediately re-enter the named block, for example, if it is located in an always construct or loop construct.

This example repeatedly calls the cpu_driver task to apply stimulus. Upon arrival of an interrupt, this example terminates the current execution of the cpu driver to service the interrupt.