



## Module 6

# Making Procedural Statements

**cā**dence®

This module describes the Verilog procedural programming statements.

## Module Objective

In this module, you:

- Describe design behavior procedurally

### Topics

- Procedural blocks review
- Making procedural assignments
- Making conditional statements
- Making case statements
- Making loop statements

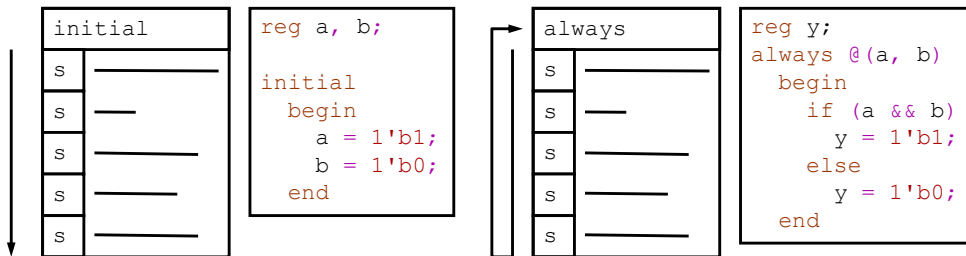


Your objective is to appropriately and effectively procedurally describe design behavior. To do that, you need to know generally about procedural blocks and procedural statements, and more specifically about the branching procedural statements. The Verilog behavioral modeling constructs are very similar to statements of the C programming language. They differ in subtle ways that can sometimes be irksome.

## Describing Module Behavior

The **initial** construct starts execution at the start of simulation, and when complete, terminates the process.

The **always** construct starts execution at the start of simulation, and when complete, executes again.



Event simulation relies upon processes. A process is an object that reacts to input events and generates output events. The continuous assignment that you have already seen is a process – the simulator reacts to its input transitions to calculate a new output value and transitions the net to that new value.

Verilog has other kinds of processes – two most obvious to the user are the **initial** construct and the **always** construct, frequently referred to as the initial block and the always block. The **initial** and **always** keywords are not statements themselves – they apply to their following statement and make it execute as a process. Their following statement is usually a statement block, e.g., statements between the **begin** and **end** keywords, but does not have to be.

- The simulator executes each **initial** process at the start of simulation, and upon completing its execution, terminates the process.
- The simulator executes each **always** process at the start of simulation, and upon completing its execution, executes it again.


## Synchronizing Module Behaviors

- Procedures “block” upon encountering an event control.
- Procedures “unblock” upon occurrence of any of the listed events.
- An event control can reference:
  - A single event identifier.
  - An event expression.
  - The “ ” wildcard operator.

### \*Examples

```
@a
@(a)
@(a or b)
@(a, b)
@*
@(*)
@(posedge clk)
@(posedge clk or negedge rstn)
```

Event  
Expression



```
always @( a or b or sel )
  if (sel == 1)
    op = b;
  else
    op = a;
```

If an *always* construct has no construct to block its execution, then when the simulator started executing it at the start of simulation, execution would continue in a tight loop forever and the simulator would appear to hang.

Verilog provides procedural timing controls for “stepping” execution of procedural blocks. The most common of these is the event 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. Both are shown here. 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.

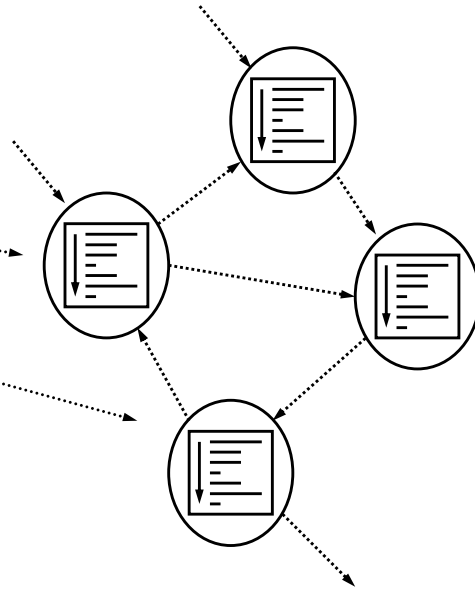
A timing control is not itself a statement, but precedes a statement, and in the case of an event control, blocks execution of that statement and subsequent sequential statements until one of those events occurs.

```
event_control ::=
  @ event_identifier
| @ ( event_expression )
| @*
| @ (*)

event_expression ::=
  expression
| hierarchical_identifier
| posedge expression
| negedge expression
| event_expression or event_expression
| event_expression , event_expression
```

## Interactions Between Behaviors

```
module design (  
    list_of_port_declarations  
);  
    net_declaration(s)  
    reg_declaration(s)  
  
    always @*  
    begin  
        statement(s)  
    end  
  
    always @(posedge clk)  
    begin  
        statement(s)  
    end  
  
    assign  
        net_assignment(s)  
  
endmodule
```



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



A non-trivial design usually contains several modules, and a module at the RT level typically contains procedural blocks:

- Each sequential procedural block executes its statements sequentially like a conventional programming language;
- Multiple blocks execute concurrently, like hardware; and
- Multiple blocks communicate with each other using events, nets and variables.

This capability to have many procedural blocks communicating concurrently with each other is the basic model which Verilog uses to describe hardware.

## Making Procedural Assignments

- Procedural assignments are assignments made inside procedures.
- You make procedural assignments only to variables.

```
module fulladder (  
    input wire a, b, cin,  
    output reg [1:0] out  
);  
  
    reg sum, carry;  
  
    always @(a, b, cin)  
    begin  
        sum  = a ^ b ^ cin;  
        carry = (a & b) | cin & (a ^ b);  
        out  = {carry, sum};  
    end  
  
endmodule
```



The *initial* construct and the *always* construct are frequently referred to as **procedural blocks**, though as you have seen, they can apply to an atomic statement. Statements within a *procedural block* are procedural statements, and most of your procedural statements will be assignments.

- Recall that a continuous assignment is to a net and is its own process, so only exists *outside* any procedural blocks.
- A procedural assignment is to a variable and is not its own process, so only exists *inside* a procedural block.
- The right operand of either assignment can contain constants, variables and nets, as on the right side only their values are used.

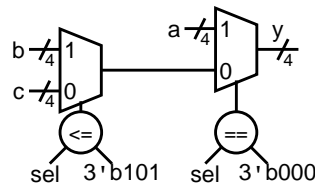
## Making Conditional Statements

- **if** is a conditional statement.
- **if** tests the conditions in sequence.
  - Condition is Boolean expression.
- Conditions can overlap.
- Executes statement associated with first known true condition.
- Statement can be null “;”.

```
module if_example (
    input      [3:0] a, b, c,
    input      [2:0] sel,
    output reg [3:0] y
);

always @(a or b or c or sel)
    if (sel == 3'b000)
        y = a;
    else if (sel <= 3'b101)
        y = b;
    else
        y = c;

endmodule
```



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

cadence

### **if** statement syntax:

```
if (expression) statement
[ { else if (expression) statement } ]
[ else statement ]
```

The conditional statement is a two-way branch. If the conditional expression evaluates to a known non-zero value then the first branch executes and otherwise the second branch executes if it exists. The conditional expression can be multiple bits.

As with other programming languages, in nested conditional statements, every **else** keyword is associated with the nearest previous **if** keyword that does not already have an **else** associated with it. If this association displeases you, then you can force association by using **begin-end** sequential blocks, or you can just toss in the **else** keyword where needed with a null following statement.

In this example, execution of the procedural block blocks at the event control until a transition occurs on at least one of the module inputs. When a transition occurs, the conditional statement executes. It evaluates the *sel* signal. If the value is exactly 3'b000, then it executes the first branch, which is an assignment of *a* to *y*. If the value is anything else, it executes the second branch, which is another conditional statement. This conditional statement again evaluates the *sel* signal. If the value is less than or equal to 3'b101, then it executes the first branch. If the value is anything else, it executes the second branch. Note that such a chain of conditional statements implies priority. If the *sel* signal is exactly 3'b000, then the second test for less than 3'b101 is not made, even though the value would match that second test if it were made.

## Conditional Statement Syntax

```
if (expression)
    statement
```

**if** branch executes if condition is known true.

```
if (expression)
    statement
else
    statement
```

**if** branch executes if condition is known true.  
**else** branch executes if condition is false or unknown.

```
if (expression)
    statement
else if (expression)
    statement
else
    statement
```

**if** branch executes if condition is known true.  
**else if** branch executes for alternative condition known true.  
**else** branch executes if no condition known true.



The 1st illustration omits the second branch.

The 2nd illustration includes the second branch.

The 3rd illustration is not a new syntax. Unlike some languages, Verilog does not have an *elsif* keyword, so this is simply chaining two conditional statements.

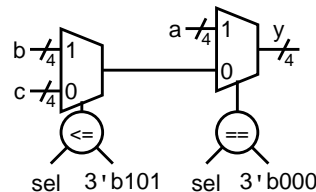


## Making Case Statements: case

- The **case** statement is a multiway prioritized branching statement.
- **case** item match expressions can overlap.
- Compares item match expressions to case expression in the order that they appear.
  - Uses case equality comparison
    - Bitwise 0,1,z,x comparison
  - Executes statement associated with first match
  - Executes optional default item if no other item matches
    - In this example, inputs containing z or x

```
module case_example (
  input  [3:0] a, b, c,
  input  [2:0] sel,
  output reg [3:0] y
);

always @(a or b or c or sel)
  case (sel)
    0          : y = a;
    1,2,3,4,5 : y = b;
    6,7       : y = c;
    default   : y = 'bx;
  endcase
endmodule
```



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

cadence

### case statement syntax:

```
case ( expression )
  expression { , expression } : statement
  {expression { , expression } : statement }
  [ default [:] statement ]
endcase
```

The **case** statement is a multiway prioritized branching statement. The case statement evaluates an expression and then evaluates a sequence of match expressions and executes the statement associated with only the first matching expression. The case statement considers high-impedance (z) and unknown (x) values and matches expressions that have a high-impedance or unknown bit in the same position in both expressions. You can comma-separate multiple match expressions associated with the same following statement. The following statement is often a statement block but does not have to be. You can optionally provide a **default** match item to be matched when no other match expression matches. It is common to place the default match item at the case statement end, but you can place it anywhere. You can have at most one default match item in each **case** statement.

In this example, execution of the procedural block is blocked at the event control until a transition occurs on at least one of the module inputs. When a transition occurs, the **case** statement executes. It evaluates the “sel” signal. If the value is 0 then it executes the first branch. If the value is between 1 and 5 then it executes the second branch. If the value is between 6 and 7 then it executes the third branch. If the value is anything else, it executes the default branch, which sets the output unknown. As an unknown value does not infer any particular hardware, the hardware has no implementation of this default branch.

## Making Case Statements: `casex`

- Treats Z, X and ? characters as “don’t care” bit positions in either:
  - case expression (`sel`)
  - case item expression `3'b0XX`
- Lets you group match values for more concise description
  - Encoders, decoders, etc.
  - Not very useful in testbenches
  - Not a preferred method usually
- `casex` treats X in case expression as meaningful in uninitialized logic
  - Hides initialization problems
  - Not recommended

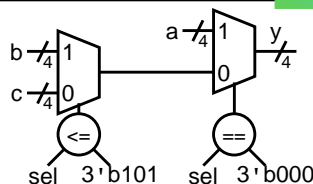


Can't match unknown or high impedance values coming out of DUT, so is less useful in testbenches.

```
module casex_example (
  input    [3:0] a, b, c,
  input    [2:0] sel,
  output reg [3:0] y
);

always @(a or b or c or sel)
  casex (sel)
    3'b000      : y = a;
    3'b0??, 3'b?0? : y = b;
    3'b11X      : y = c;
    default     : y = 'bx;
  endcase
endmodule
```

Using ? is more readable than using X or Z



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

cadence

The ***casex*** statement is a variation of the *case* statement that lets you specify bit positions it should not compare.

The *casex* statement treats the *x*, *z*, and question mark (?) characters as *don't-care* bit values. Recall that in a literal constant the ? character is equivalent to the *z* character and that only the based literal constants can use the *x*, *z*, and ? characters.

The *casex* statement does not compare bit positions of either the case expression or the case match items that have any of these *don't-care* values.

By convention, you might want to use the ? character to indicate *don't-care* bit positions in literal constants to avoid confusion with the high-impedance and unknown values. However, the *casex* statement still considers bit positions containing high-impedance (*z*) or unknown (*x*) values in expressions of nets and variables to be positions it should not match, so a *casex* statement in a testbench cannot test high-impedance or unknown values coming from the DUT.

This example is functionally identical to the previous example, but uses *don't-care* bit positions to represent a range of matching values.

## Making Case Statements: `casez`

- Treats `z` and `?` characters as “don't care” bit positions in:
  - case expression (`sel`)
  - case item expression `3'b0ZZ`
- Performs definitive match for `x`
  - Useful in testbenches!
- `casez` is preferable to `casex`.
  - `x` in case expression is not wildcard
  - Safer for uninitialized RTL code



Can match unknown values so is more useful in testbenches.

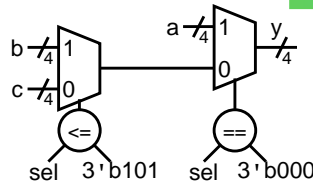


Use `?` instead of `z` – it's more intuitive and will cause less confusion!

```
module casez_example (
    input    [3:0] a, b, c,
    input    [2:0] sel,
    output reg [3:0] y
);

always @(a or b or c or sel)
    casez (sel)
        3'b000      : y = a;
        3'b0??, 3'b?0? : y = b;
        3'b11Z      : y = c;
        default     : y = 'bx;
    endcase
endmodule
```

Use `?` To avoid confusion with high impedance logic state `Z`



The ***casez*** statement is a variation of the ***casex*** statement and treats only the `z`, and question mark (`?`) characters as *don't-care* bit values and performs a definitive match for unknown (`x`) values. In a testbench, to test the existence of unknown values from the DUT, you should use the ***casez*** statement instead of the ***casex*** statement.

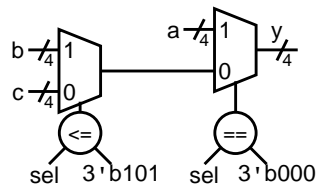
This example is functionally identical to the previous example, but uses the `z` character instead of the `x` character to represent the *don't-care* bit positions.

## Alternate Case Statement Form

- **case** item match can be an expression as well.
- Case item match expression does not have to be a constant value!
- Can use case statement to match *variable* expressions.
- Case items are prioritized according to the order.

```
module finalcase (
  input      [3:0] a, b, c,
  input      [2:0] sel,
  output reg [3:0] y
);

always @(a or b or c or d)
  case (1'b1)
    sel == 3'b000 : y = a;
    sel <= 3'b101 : y = b;
    sel >= 3'b110 : y = c;
    default      : y = 'bx;
  endcase
endmodule
```



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

cadence

### Alternate *case* statement syntax:

```
case ( expression )
  expression { , expression } : statement
  {expression { , expression } : statement }
  [ default [:] statement ]
endcase
```

Unlike most programming languages, the Verilog case item expression does not have to be constant.

This example is functionally identical to the previous example, but tests which match expression first matches the value 1, that is, which one is the first “true” expression.

```
case (1'b1)
  en_a : op = a;
  en_b : op = b;
  en_c : op = c;
endcase
```

In the above code, the case statement en\_a has a highest priority. The statement checks for en\_a first. If this is 1'b1, then the output op is a. Otherwise it checks for en\_b. If en\_b is 1'b1 then output op = b. When both en\_a and en\_b not 1'b1, then en\_c is checked. If en\_c is 1'b1 then output op = c.

## Making Loop Statements: **while**

- **while** loop iterates and executes its following statement while the expression is known and non-zero
- 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
```

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



**while** statement syntax:

```
while ( expression )
    statement
```

The **while** loop executes its following statement while the expression is known and non-zero. The following statement is usually a statement block but does not have to be. If the expression is not known or is zero, then the following statement never executes.

The 1<sup>st</sup> illustration executes a statement block while the count is known and less than 10. The last statement of the block increments the count. Some statement before the loop presumably initializes the count, but this is not shown.

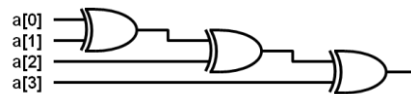
The 2<sup>nd</sup> illustration counts the number of bits in the *tempreg* that are 1. While the “tempreg” is known and non-zero, if the least significant bit of the *tempreg* is known and non-zero it increments the count. In any case, each loop iteration shifts the *tempreg* one position to the right.

## Making Loop Statements: `for`

- `for` loop requires prior declaration of any variables used in expressions.
- *Initialization* is done for the loop.
- Condition checked.
- *while* condition is known true, executes statement(s).
- *Update* expression is then executed at end of each iteration.

```
module parity (
    input [3:0] a,
    output reg odd
);

integer i;
always @*
begin
    odd = 0;
    for (i = 0; i <= 3; i = i + 1)
        odd = odd ^ a[i];
    end
endmodule
```



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

cadence

### *for* loop Syntax:

```
for ( initializer exp; condition exp;
      update exp )
    statement;
```

You can more concisely rewrite a while loop as a *for* loop that is not as complex. The *for* loop performs the initial variable assignment and then while the test expression is known and non-zero executes the following statement. The following statement is usually a statement block but does not have to be. After each iteration of its following statement, the *for* loop executes the second variable assignment. The two variable assignments are almost always to a variable that participates in the test expression, but they don't have to be.

This example calculates the parity of an input vector reg. The *for* loop initializes an index to 0 and while the index is less than or equal to 3 it exclusive-ORs the partial result with each successively indexed bit of the input, each time incrementing the index by 1.

If you think for a moment, you can think of an operator that will do this in one assignment without a loop. The same *for* loop can also be implemented using an equivalent while loop. The syntax is:

```
Initializer exp;
while ( condition exp )
    statement;
update exp;
```

## Making Loop Statements: repeat

- **repeat** loop executes the statements number of times specified in the loop.
- Must previously declare any variables used in expression
- For expressed count executes statement.
- The **repeat** loop is equivalent to a **for** loop with the index variable automatically declared, initialized and incremented.

```
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
```



**repeat** loop syntax:

```
repeat ( expression ) statement
```

You can more concisely rewrite a for loop as a *repeat* loop, where you do not need to use the index, for example, to select a bit of a vector. A *repeat* loop executes its following statement the number of times specified by its parenthesized expression.

This example multiplies two 4-bit inputs to produce an 8-bit output. Because it uses a shift multiplier implementation, it does not need a loop index, so uses a *repeat* loop instead of a *for* loop. Equivalent logic can be obtained using for loop as well. The syntax is:

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

## Making Loop Statements: `forever`

- `forever` loop executes statement forever.

```
reg clock;  
initial  
  forever  
    #10 clock = !clock ;
```

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



*forever* loop syntax:

*forever* *statement*

Verilog also provides a *forever* loop for when you want to repeat execution of a statement an essentially infinite number of times or until simulation ends. The same logic can be modeled using while loop as well. The syntax is:

```
while ( 1 )  
  statement
```



## Module Summary

Now you can appropriately and correctly describe design behavior procedurally.

In this module, you studied these topics:

- The **initial** and **always** constructs introduce procedural blocks.
- The **@** event control blocks further procedural block execution until an event occurs.
- The **=** procedural assignment that is inside a procedural block you make only to variables.
- The **if-else**, and **case/casex/casez** procedural branching statements.
- The **for**, **while**, **repeat**, and **forever** procedural looping statements.



You should now be able to appropriately and effectively procedurally describe design behavior. This module described Verilog procedural blocks and procedural programming statements.

## Module Review

1. If more than one case item expression matches the case expression, which associated statement(s) execute?
2. Can you use the **case**, **if** and **for** statements in continuous assignments?



*This page does not contain notes.*

## Module Review Solutions

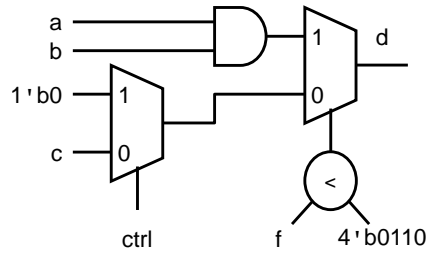
1. If more than one case item expression matches the case expression, which associated statement(s) execute?
  - The case expression is compared to the case item expressions in the order they appear and the statement associated with only the first match is executed.
2. Can you use the **case**, **if** and **for** statements in continuous assignments?
  - The **case**, **if** and **for** statements are procedural statements that cannot appear in a continuous assignment.



*This page does not contain notes.*

## Module Exercise 1

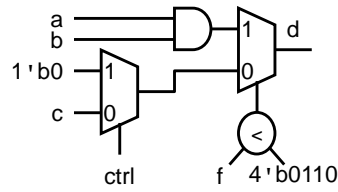
Code the following conditional logic as a procedural block:



*This page does not contain notes.*

## Module Exercise 1 Solution

Code the following conditional logic as a procedural block:



Solution:

```
always @(a or b or c or f or ctrl)
  if (f < 4'b0110)
    d = a & b;
  else if (ctrl)
    d = 1'b0;
  else
    d = c;
```



*This page does not contain notes.*

## Module Exercise 2

```
// Find at least twelve things wrong!

module twelve_wrong (w, v, y, t, b, c, s)
input  [3:0] a, b, c, s;
output [3:0] w, v, y, t;
reg    [3:0] w, v, y, t,

assign W = s;

always @(a or b or c and s)
  if (s == 0)
    v = a;
    y = b;
    t = c;
  else if (s <= 4'b0101)
  begin
    v = c;
    y = b;
    t = a;
    a = s;
  end
  else if (s == 6 or s == 7)
    v = c;
  else v == 4'b'xxxx;
endmodule
```



*This page does not contain notes.*

## Module Exercise 2 Solution

```

module twelve_wrong (w, v, y, t, b, c, s) // Append semicolon
input  [3:0] a, b, c, s;                // Include "a" in port list
output [3:0] w, v, y, t;                // Change first "," to ":"
reg    [3:0] w, v, y, t;                // Change last "," to ";"

assign W = s;                           // Change "w" to "W"
                                         // Change to procedural assignment
                                         // Change "and" to "or"
always @(a or b or c and s)
    if (s == 0)                          // Insert "begin"
        v = a;
        y = b;
        t = c;
    else if (s <= 4'b0101)                // Insert "end"
    begin
        v = c;
        y = b;
        t = a;
        a = s;                           // Change "a" to "w"
    end
    else if (s == 6 or s == 7)            // Change "or" to "||"
        v = c;
    else v == 4'b'xxxx;                  // Change "==" to "="
                                         // Remove second "'"
endmodule

```

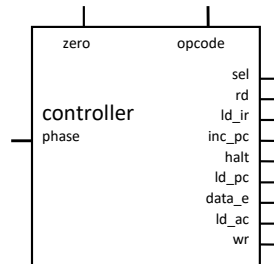
*This page does not contain notes.*

## Lab



### Lab 6-1 Modeling a Controller

- Use the Verilog **case** statement to describe a controller.



*This page does not contain notes.*