Module 8

Using Continuous and Procedural Assignments

cadence

This module differentiates in more detail between continuous statements and procedural statements.

Module Objective

In this module, you:

- Appropriately choose between continuous and procedural assignments

Topics

- Continuous assignments review
- Multiple continuous assignments to a single net
- Procedural assignments review
- Multiple procedural assignments to a single variable
- Understanding the simulation cycle
- Conditional operator revisit
- Feedback loops
- Generate Statements

cadence

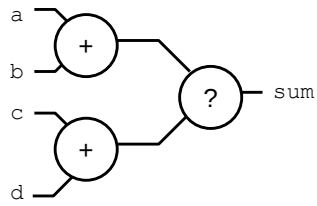
Your objective is to appropriately select between continuous and procedural statements. To do that, you need to know in more detail how these two different kinds of assignments work.

Continuous Assignments Review

- Only outside procedural blocks
- Continuously drive nets
- Order of declaration does not affect functionality

```
wire [8:0] sum;
assign sum = a + b;
assign sum = c + d;
```

Can declare in either order



A continuous assignment is its own process – the simulator automatically updates the driven value and resolved net value when any of the inputs transition.

You can place a continuous assignment in a module anywhere after you declare the primaries that are its inputs. For tools compliant with the Verilog 2005 standard, you do not need to declare the target. The target of a continuous assignment is implicitly declared as a scalar net of the default net type, usually a wire. Reliance upon implicit declarations is generally considered poor practice.

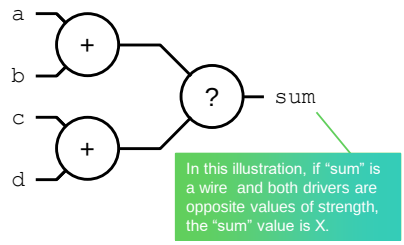
The simulator updates continuous assignments in any simulation cycle in which their inputs transition. No order is implied among multiple continuous assignments updated in the same simulation cycle. Relative position in the source code has no effect.

Multiple Continuous Assignments

Multiple continuous assignments to a single net are “wired together”.

- The wire, wand and wor net types all resolve value conflicts differently.

```
wire [8:0] sum;
assign sum = a + b;
assign sum = c + d;
```



wire

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

wand

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

wor

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

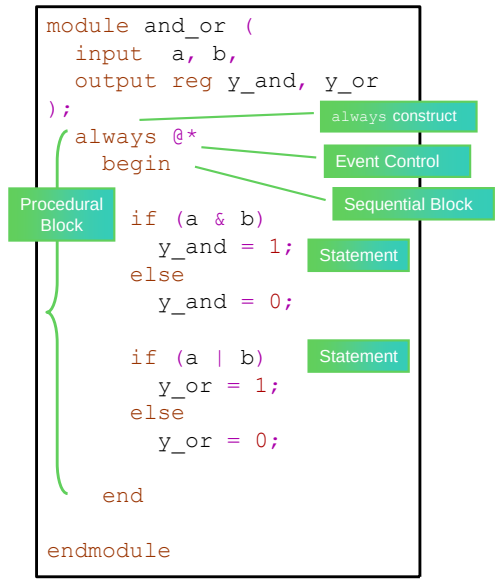
cadence

Except for the Verilog-2005 unresolved wire (**uwire**) type, the simulator resolves the value of a net driven by multiple drivers. The **uwire** type does not accept multiple drivers and reports an error.

- If one driver is stronger than the others, the strong driver “wins”. You can, for example, pull up a wire and still drive it to the 0, 1 and unknown (x) values. When you drive it to the high-impedance (z) value, the pullup takes over to maintain a pull-strength 1 on the wire. *Appendix B: Modeling with Verilog Primitives and UDPs* examines drive strengths in detail.
- If more than one driver is equally stronger than the remaining drivers, the simulator resolves the value of the net. The resolution function depends upon the net type and is different for nets that do and do not represent wired logic.
  - A **wire** net will attain the 0 or 1 value when all such strong drivers drive either a high-impedance value or the same 0 or 1 value and none of them drive an unknown value.
  - A wired-AND (**wand**) net will attain the 0 value when any such strong drivers drive a 0 value.
  - A wired-OR (**wor**) net will attain the 1 value when any such strong drivers drive a 1 value.

Procedural Assignments Review

- Procedural blocks start with always or initial.
- An event control almost always immediately follows the **always** keyword.
  - Blocks further statement execution until an event occurs.
- Statements within a sequential block (*begin-end*) execute sequentially.
- Assignments are only to variables – not nets.
- Multiple procedural blocks execute “concurrently”.



A procedural block is a process. It reacts to input transitions and generates output transitions.

You start a procedural block with an *always* keyword or an *initial* keyword. The *always* keyword starts a block that executes continually and the *initial* keyword starts a block that executes once. You control the execution of a procedural block by using timing controls. The most common of these is the event control, which blocks further statement execution until an event occurs.

The *always* and *initial* keywords apply to a following statement. That following statement is often a statement group between the *begin* and *end* keywords. Statements between the *begin* and *end* keywords execute sequentially. The previous statement finishes execution before the next statement starts. The Verilog for Verification section in this book examines a similar construct that encloses statements executing concurrently.

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.

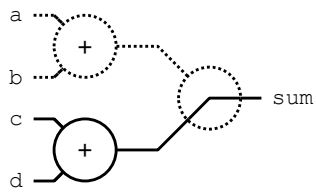
You will later see nonblocking assignments that use the same token as the less-than-or-equal-to (<=) operator. They are called nonblocking assignments because their completion is scheduled and they do not block execution of the next statement.

Multiple Procedural Assignments

Statements within a sequential block (*begin-end*) execute sequentially.

- Subsequent assignments override previous assignments.

```
reg [8:0] sum;
...
always @*
  begin
    sum = a + b;
    sum = c + d;
  end
```



Statements within a sequential block execute sequentially. Subsequent assignments override previous assignments. The latest assignment to a variable overrides all previous assignments to the variable.

This example first assigns to the variable the sum of a+b. Then without any delay it assigns to the variable the sum of c+d. The first assignment has no duration, so no other block can observe its results. No glitch will occur. Smart simulators will see this and optimize away the first assignment.

Example Multiple Procedural Assignments

- Subsequent assignments override previous assignments.
- Both these code fragments are equivalent.
- The version with the default assignment statement will be preferred for synthesis.

```
always @(a or b or sel)
  begin
    if (sel)
      y = b;
    else
      y = a;
  end
```

```
always @(a or b or sel)
  begin
    y = a;
    if (sel)
      y = b;
  end
```

144

cadence

Statements within a sequential block execute sequentially. Subsequent assignments override previous assignments. The latest assignment to a variable overrides all previous assignments to the variable.

This semantic is quite useful. You can write a code block that first provides default values for all the variables it updates, and then executes its algorithm to conditionally update some variables with various values. Upon exiting the block, you know that it has provided values for all of its variables. None has been missed regardless of what path the execution took through the block.

This illustration replaces the default branch of an *if* statement with a default assignment. The illustration is admittedly trivial, but does serve to illustrate the concept. Imagine a larger block having perhaps 50 statements and several possible paths through the statements.

Conditional Operator Revisited

- Can replace a simple combinational procedure with a continuous assignment.
- Target must be a net!
- Event control is assumed.
- May be more “readable” than procedural equivalent.

Both these modules are equivalent.

```
module procedural_if (
  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
```

```
// USING CONDITIONAL OP
y = (sel == 3'b000) ? a
  : (sel <= 3'b101) ? b
  :                   c;
```

```
module continuous_if (
  input      [3:0] a, b, c,
  input      [2:0] sel,
  output     [3:0] y
);
  assign
    y = (sel == 3'b000) ? a
      : (sel <= 3'b101) ? b
      :                   c;
endmodule
```


cadence

For assignments to a single target, a conditional operator may be more readable than a conditional statement, and with the conditional operator, you can alternatively make the assignment a continuous assignment instead of a procedural assignment. Remember that continuous assignments are to nets and procedural assignments are to variables.

Combinational Feedback Loops – Be Careful

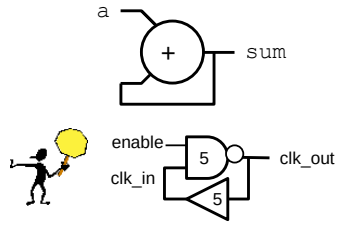
Zero-delay feedback loops may cause the simulator to appear to “lock up”.

- The process never finishes or suspends.
- The simulator never gets to do anything else.

Here is a short feedback loop deliberately generating a clock:

- For illustrative purposes only – more elegant ways exist to generate a clock.
- A continuous assignment is its own process. Whenever clk_out changes, the value is updated continuously.

```
always @(a , sum)
  sum <= sum + a;
```




```
always @*
  clk_out = #5 !(clk_in && enable);

assign #5 clk_in = clk_out;
```

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

cadence

The semantics of a Verilog process is that it reacts to its inputs and generates outputs. This is a different semantic than a procedural programming language such as C that does not autonomously react to inputs.

In continuous assignment, the simulator automatically updates the driven value and resolved net value when any of the inputs transition. For this illustration, the process is sensitive to its output, so the simulator could enter a tight loop where it does nothing but continually update the net.

You may understand this behavior better by instead making a procedural assignment to a variable within an always block and triggering the assignment with the same set of events. The assignment occurs when any of the inputs transition, so again the simulator could enter a tight loop where it does nothing but continuously update the variable.

Generate Statements in Verilog-2001

Verilog 2001 adds new reserved words: generate endgenerate genvar

- Defined within a module.
- Used to generate code dynamically within a module using conditional statements.
- Genvar is a positive integer value, used only inside a generate block.
- Generate_block_name (optional) is used to create an unique instance name for each generated item.
- Conditional (**case**, **if**) generation.
  - Instances, functions, tasks, variables, and procedural blocks.
- Iterative (**for**) generation.
  - Instances, variables, and procedural blocks (no functions or tasks).
- The keywords generate – endgenerate are optional in Verilog-2005.

cadence

Place your generated instances, functions, tasks, variables, and procedural blocks between the *generate* and *endgenerate* reserved words (you may not include parameters, ports, or specify blocks).

Declare *genvar* index variables for your generate *for* loops. You can declare them either inside or outside the *generate* statement. You can assign only integer values to them, and only within a *for* loop. These variables disappear after elaboration and are not available during simulation.

Iteratively generate statements using the *for* construct. Declare a named block within the *for* construct – this becomes the name of an array of scopes to match the iterations of the *for* construct. Inside this named block place your generated instances, variables, and procedural blocks. (You cannot define tasks and function within a generate *for* construct.)

Conditionally generate statements using *case* and *if* constructs.

Generate Statement – Conditional `if` Example

- Conditional *if* generation:
  - Instances, functions, tasks, variables, and procedural blocks
- Label not required to create generate-if scope.

```
module multiplier(a,b,product);
  parameter a_width = 8, b_width = 8;
  localparam product_width = a_width+b_width;
  input [a_width-1:0] a;
  input [b_width-1:0] b;
  output [product_width-1:0] product;
     generate
       if ((a_width < 8) || (b_width < 8)) begin: mult
           CLA_multiplier #(a_width,b_width) u1(a, b, product);
         // instantiate a CLA multiplier
       end
       else begin: mult
           WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
       end
    endgenerate
endmodule
```

Cannot be modified directly with the defparam statement or the module instance statement #

The hierarchical instance name is mult.u1

An implementation of a parameterized multiplier module

cadence

These are the examples for usage of the *generate* statement using the conditionals *if-else*, case and the iterative *for*.

A *generate-for* loop permits one or more generate items to be instantiated multiple times. The index loop variable must be a *genvar*. This example shows a parameterized gray-code to binary-code converter using a loop to generate a continuous assignment for each bit of the converter.

A generate *if-else* or case permits generate items to be conditionally instantiated based on an expression that is deterministic at the time the design is elaborated.

The *if-else* generate example shows the generation of instances of a carry-look-ahead multiplier. If the input bus widths are greater than 8 bits, then an instance of a wallace-tree multiplier is generated.

Generate Statement – Conditional `case` Example

- Conditional case generation
  - Instances, functions, tasks, variables, and procedural blocks
- Label not required to create *generate-case* scope.

```
generate
  case (WIDTH)
    1: begin: adder // 1-bit adder implementation
         adder_1bit x1(co, sum, a, b, ci);
       end
    2: begin: adder // 2-bit adder implementation
         adder_2bit x1(co, sum, a, b, ci);
       end
    default:
       begin: adder // others - CLA
         adder_cla #(WIDTH) x1(co, sum, a, b, ci);
       end
  endcase
endgenerate
```

Generate with a case to handle widths less than 3

The hierarchical instance name is adder.x1

The *case-generate* example shows the instantiation of an appropriate adder depending on the case index WIDTH given as a *genvar*.

Generate Statement – Iterative Example

- Iterative (for) generation
  - Instances, variables, and procedural blocks (no functions or tasks)
- Label is required to create *generate-for* scope

```
module gray2bin1 (bin, gray);
  parameter SIZE = 8; // this module is parameterizable
  output [SIZE-1:0] bin;
  input [SIZE-1:0] gray;

  genvar i;
  generate
     for (i=0; i<SIZE; i=i+1) begin:bitnum
       // required label for scope naming
     assign bin[i] = ^gray[SIZE-1:i];
     end
   endgenerate
endmodule
```

A parameterized gray-code–to–binary-code converter module using a loop to generate continuous assignments.

i refers to the implicitly defined localparam whose value in each instance of the generate block is the value of the genvar when it was elaborated.

cadence

These are the examples for usage of the *generate* statement using the conditionals *if-else*, *case* and the iterative *for*.

A *generate-for* loop permits one or more generate items to be instantiated multiple times. The index loop variable must be a *genvar*. This example shows a parameterized gray-code to binary-code converter using a loop to generate a continuous assignment for each bit of the converter.

A generate *if-else* or *case* permits generate items to be conditionally instantiated based on an expression that is deterministic at the time the design is elaborated.

The *if-else* generate example shows the generation of instances of a carry-look-ahead multiplier. If the input bus widths are greater then 8 bits, then an instance of a Wallace-tree multiplier is generated.

The case generate example shows the instantiation of an appropriate adder depending upon the case index WIDTH.

Module Summary

Now you can appropriately choose between continuous and procedural assignments.

This module described:

- Continuous assignments using **assign** that you make only to a *net* and only *outside* procedural blocks. The simulator resolves a net value due to multiple continuous assignments.
- Procedural assignments using **=** that you make only to a *variable* and only *inside* procedural blocks. The last such assignment “wins” (which makes multiple such assignments in different procedural blocks problematic!).
- The conditional operator **?:**, which is useful in both continuous and procedural assignments.
- Inadvertently coding combinational feedback loops.
- The generate statements introduction: Verilog-2001 construct.

cadence

You should now be able to appropriately select between continuous and procedural statements. This module described in more detail the difference between continuous statements and procedural statements.

Module Review

1. What is the result of multiple continuous assignments to a net?
2. What is the result of multiple procedural assignments to a variable?
3. Explain how the conditional operator contributes to the usefulness of continuous assignments.
4. What causes combinational feedback loops?

cadence

This page does not contain notes.

Module Review Solutions

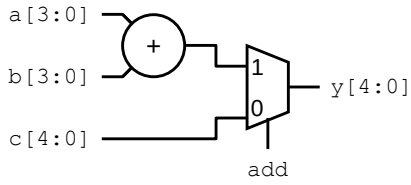
1. What is the result of multiple continuous assignments to a net?
   - Verilog resolves the value of multiple drivers of a net.
2. What is the result of multiple procedural assignments to a variable?
   - The last procedural assignment to the variable “wins”.
3. Explain how the conditional operator contributes to the usefulness of continuous assignments.
   - A continuous assignment can sometimes more intuitively describe simple combinational logic than can a procedure. With the conditional operator, you can concisely describe more complex logic.
4. What causes combinational feedback loops?
   - A combinational feedback loop occurs when a transition on the output of a process propagates to an input of the process. A combinational feedback loop is not a digital construct, so is a coding error in a digital design.

cadence

This page does not contain notes.

Module Exercise

Code the following logic using a conditional operator in a (i) continuous assignment and (ii) procedural assignment.

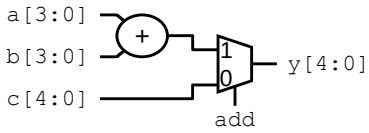


cadence

This page does not contain notes.

Module Exercise Solution

Code the following logic using a conditional operator in a (i) continuous assignment and (ii) procedural assignment.



Solution:

```
always @* y = (add == 1) ? a + b : c ;
```

```
assign y = (add == 1) ? a + b : c ;
```

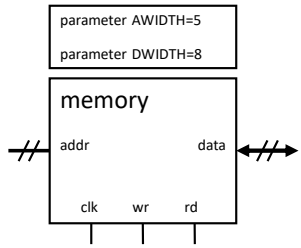

cadence

This page does not contain notes.

Lab

Lab 8-1 Modeling a Single-Bidirectional-Port Memory

- Use continuous and procedural assignments while describing a memory.



Your objective is to choose between continuous and procedural assignments.

For this lab, you use continuous and procedural assignments while describing a memory:

- You use procedural assignment for the write operation; and
- You use continuous assignment for the read operation.