Module 15

Avoiding Simulation Mismatches

This module addresses the common and perplexing problem of the post-synthesis simulation results that do not match the pre-synthesis simulation results.

Module Objective

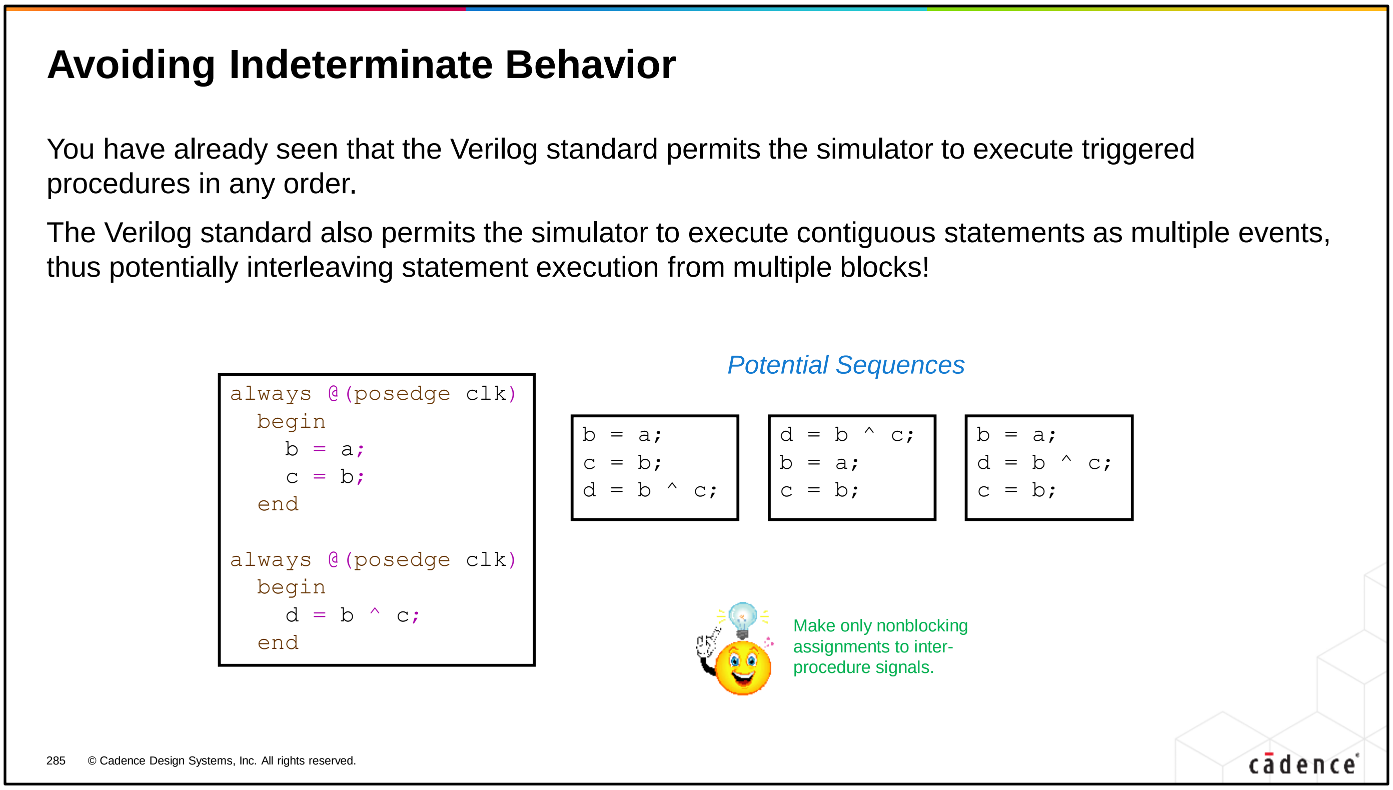
In this module, you:

- Avoid the common causes of mismatches between RTL simulation and post-synthesis netlist simulation

Topics

- Non-deterministic behavior
- Synthesis attributes
- Conditional compilation
- Incomplete sensitivity list
- Temporary variables
- Asynchronous set and reset
- RTL vs. Synthesis Models
- Incomplete Assignments
- Comparing unknown values
- Case statements: **casex** and **casez**
- Variable declaration assignment
- Delay controls

This page does not contain notes.



Avoiding Indeterminate Behavior

You have already seen that the Verilog standard permits the simulator to execute triggered procedures in any order.

The Verilog standard also permits the simulator to execute contiguous statements as multiple events, thus potentially interleaving statement execution from multiple blocks!

```
always @(posedge clk)
  begin
    b = a;
    c = b;
  end

always @(posedge clk)
  begin
    d = b ^ c;
  end
```

Potential Sequences

```
b = a;
c = b;
d = b ^ c;
```

```
d = b ^ c;
b = a;
c = b;
```

```
b = a;
d = b ^ c;
c = b;
```

Make only nonblocking assignments to inter-procedure signals.

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

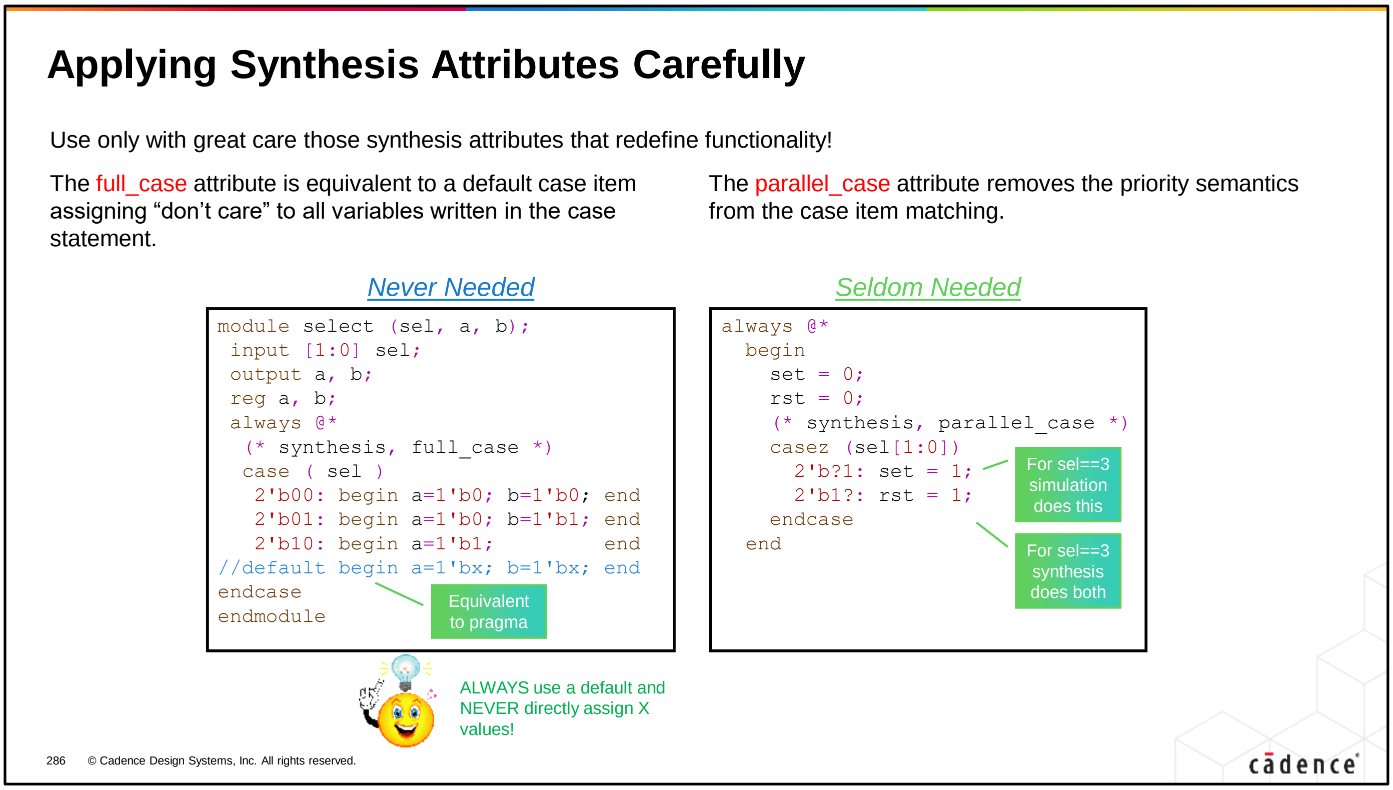
cadence

You have already seen that the Verilog standard permits the simulator to execute triggered procedures in any order.

The Verilog standard also permits the simulator to execute contiguous statements as multiple events, thus potentially interleaving statement execution from multiple blocks!

You can greatly reduce indeterminacy by making only nonblocking assignments to storage variables that communicate between procedures.

“At any time while evaluating a behavioral statement, the simulator may suspend execution and place the partially completed event as a pending active event on the event queue. The effect of this is to allow the interleaving of process execution. Note that the order of interleaved execution is nondeterministic and not under control of the user.” – IEEE Std. 1364-2001 Section 5.4.2 Nondeterminism



Applying Synthesis Attributes Carefully

Use only with great care those synthesis attributes that redefine functionality!

The full_case attribute is equivalent to a default case item assigning "don't care" to all variables written in the case statement.

The parallel_case attribute removes the priority semantics from the case item matching.

Never Needed

```
module select (sel, a, b);
 input [1:0] sel;
 output a, b;
 reg a, b;
 always @*
  (* synthesis, full_case *)
  case ( sel )
   2'b00: begin a=1'b0; b=1'b0; end
   2'b01: begin a=1'b0; b=1'b1; end
   2'b10: begin a=1'b1;         end
//default begin a=1'bx; b=1'bx; end
endcase
endmodule
```

Equivalent to pragma

Seldom Needed

```
always @*
  begin
    set = 0;
    rst = 0;
    (* synthesis, parallel_case *)
    casez (sel[1:0])
      2'b?1: set = 1;
      2'b1?: rst = 1;
    endcase
  end
```

For sel==3 simulation does this

For sel==3 synthesis does both

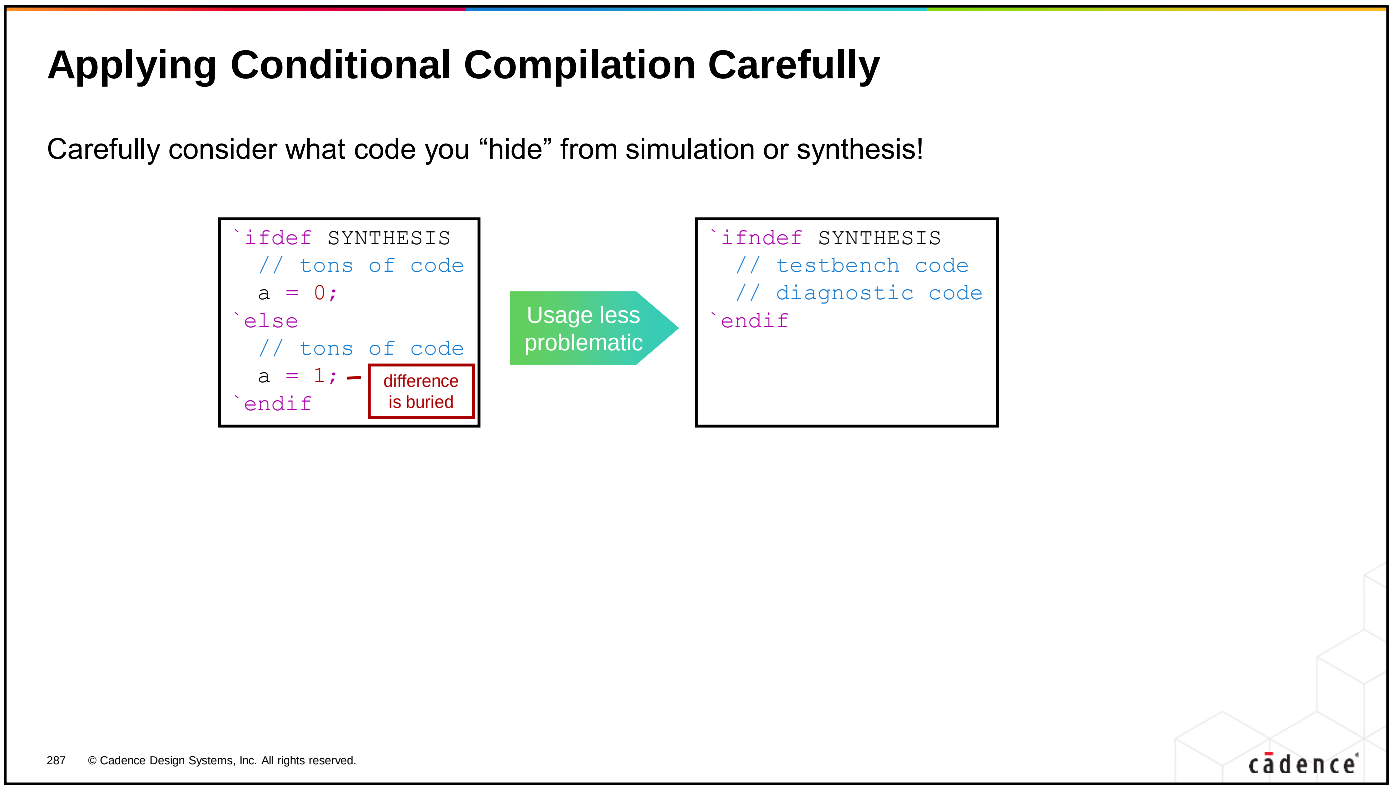
ALWAYS use a default and NEVER directly assign X values!

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

cādence

The full_case attribute is equivalent to a default case item assigning "don't care" to all variables written in the case statement. Simulation, of course, knows nothing about any synthesis attributes. For this example, when the select value is 3 or unknown, the output values are unknown for simulation and some undetermined binary value for synthesis.

The parallel_case attribute directs synthesis to not build a priority structure for testing case match items. Simulation, of course, knows nothing about any synthesis attributes. For this example, synthesis builds hardware assuming that if the select value is 3 then you really want both the set and the reset to occur simultaneously, while simulation, as usual, executes only the first matching statement.



Applying Conditional Compilation Carefully

Carefully consider what code you “hide” from simulation or synthesis!

```
`ifdef SYNTHESIS
  // tons of code
  a = 0;
`else
  // tons of code
  a = 1;
`endif
```

difference is buried
Usage less problematic

```
`ifndef SYNTHESIS
  // testbench code
  // diagnostic code
`endif
```

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

Carefully consider what code you hide from simulation or synthesis. You can reasonably safely hide from synthesis testbench-related code that you embed in an RTL module. Much less valid reason exists to hide design code from simulation. Complex, frequent and large conditionally compiled regions especially promote inadvertent coding errors that RTL simulation does not encounter.

Completing the Combinational Logic Sensitivity List

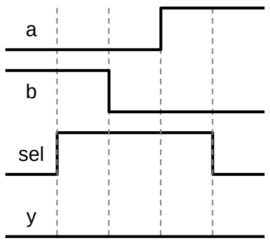
For simulation, include in the event list all signals that are input to the logic. For your convenience, you can use the "*" wildcard.

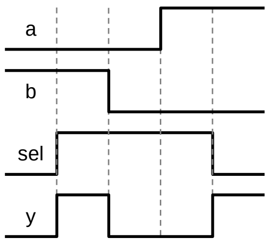
```
// Incomplete list

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

```
// Complete list

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



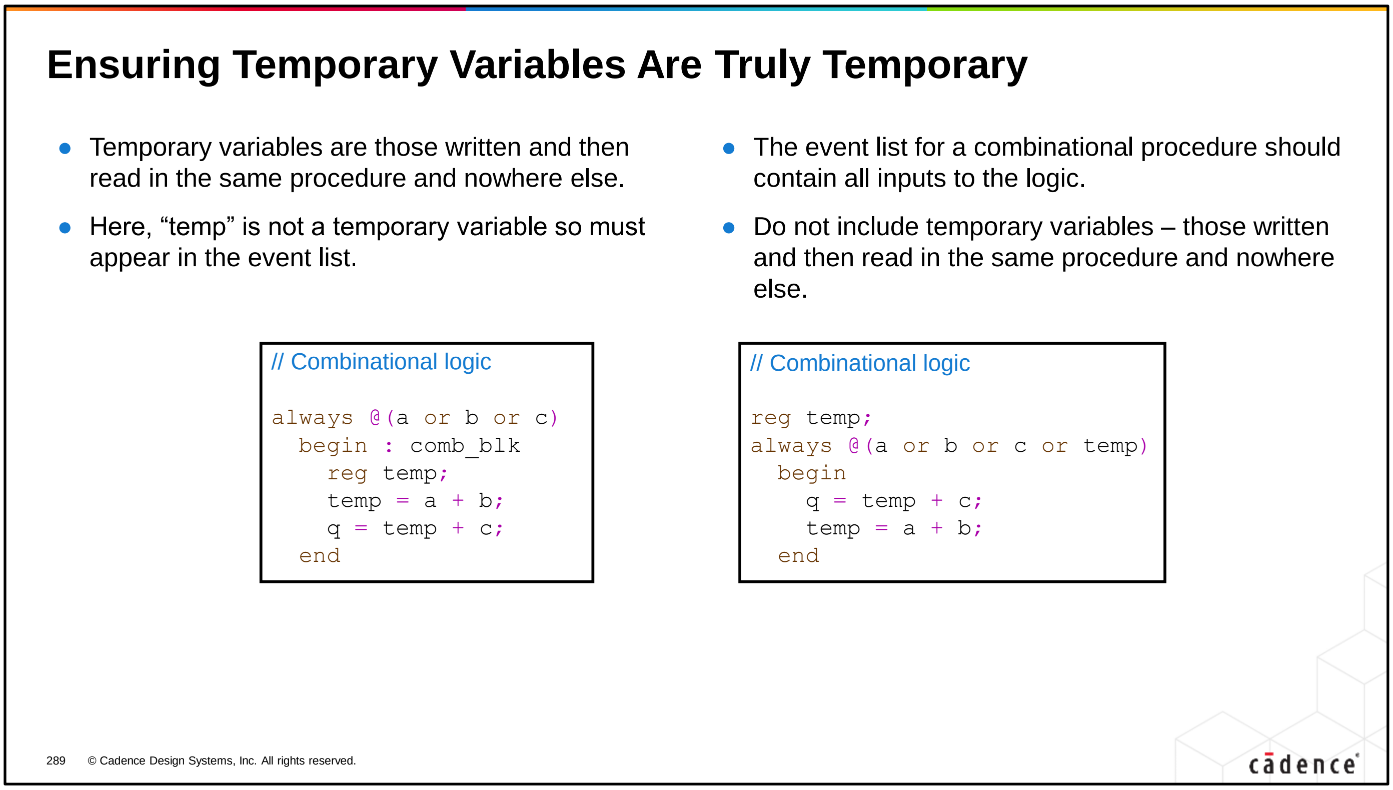


The event list does not affect the synthesis result, but to avoid incorrect RTL simulation results, you should include in the event list all inputs to the procedure. The easiest way to ensure this is to use the Verilog 2001 wildcard event control (@*).

This example illustrates the effect of an incomplete sensitivity list. If the event list omits the sel signal, the procedure executes upon transitions of only the a and b inputs – transitions of the sel signal have no affect.

Debugging problems caused by an incomplete sensitivity list is difficult, so you might want to develop the habit of simply always using the wildcard event control for all combinational procedures.

"The event list does not affect the synthesized netlist." – IEEE Std. 1364.1-2002 5.1 Modeling combinational logic



Ensuring Temporary Variables Are Truly Temporary

- Temporary variables are those written and then read in the same procedure and nowhere else.
- Here, “temp” is not a temporary variable so must appear in the event list.

```
// Combinational logic

always @(a or b or c)
  begin : comb_blk
    reg temp;
    temp = a + b;
    q = temp + c;
  end
```

- The event list for a combinational procedure should contain all inputs to the logic.
- Do not include temporary variables – those written and then read in the same procedure and nowhere else.

```
// Combinational logic

reg temp;
always @(a or b or c or temp)
  begin
    q = temp + c;
    temp = a + b;
  end
```

The synthesis standard states that the sensitivity list shall not affect the generation of combinational logic. That means that if the synthesis tool recognizes the block as combinational logic, it will proceed as if you had included all inputs to the logic in the sensitivity list. The synthesis tool may or may not warn you about missing inputs. The generated gates will simulate correctly, but very likely differently than the incorrect RTL simulation.

Modifying RTL Model to Match Synthesis Model

This RTL model embeds testbench constructs to correct the simulation.

```
always @(posedge clk
      or posedge set
      or posedge rst)
  if (set)
    q <= 1;
  else
    if (rst)
      q <= 0;
    else
      q <= d;
```

This RTL model stays set if "set" and "rst" are both applied and then only "set" is removed.

```
always @(posedge clk
      or posedge set
      or posedge rst)
  if (set)
    q <= 1;
  else
    if (rst)
      q <= 0;
    else
      q <= d;

`ifndef SYNTHESIS
always @(set or rst)
  if (set)
    assign q = 1;
  else
    if (rst)
      assign q = 0;
    else
      deassign q;
`endif
```

Procedural Continuous Assignment

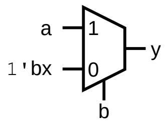
You are unlikely to model both asynchronous set and asynchronous reset, and even less likely to use them in a situation where they are both simultaneously active. If you do, though, the RTL synthesis template will not accurately model the behavior, as it does not respond to the inactive edge of the set or reset inputs.

You can work around this with embedded testbench code not meant for synthesis. You can use the procedural continuous assignment to override a normal procedural assignment. You apply a procedural continuous assignment only to a variable, unlike the normal continuous assignment you apply only to a net. While the procedural continuous assignment to a variable is in effect, the simulator ignores any normal procedural assignment to the variable, so be sure to deassign the variable when you want it to again behave normally.

Do Not Confirm Expected Unknown Values

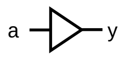
Synthesis of a “x” assignment is a “don’t care” condition that is “optimized away”.

```
always @(a or b)
  if (b)
    y = a;
  else
    y = 'bx;
```



Simulation of an “x” assignment makes the variable value unknown.

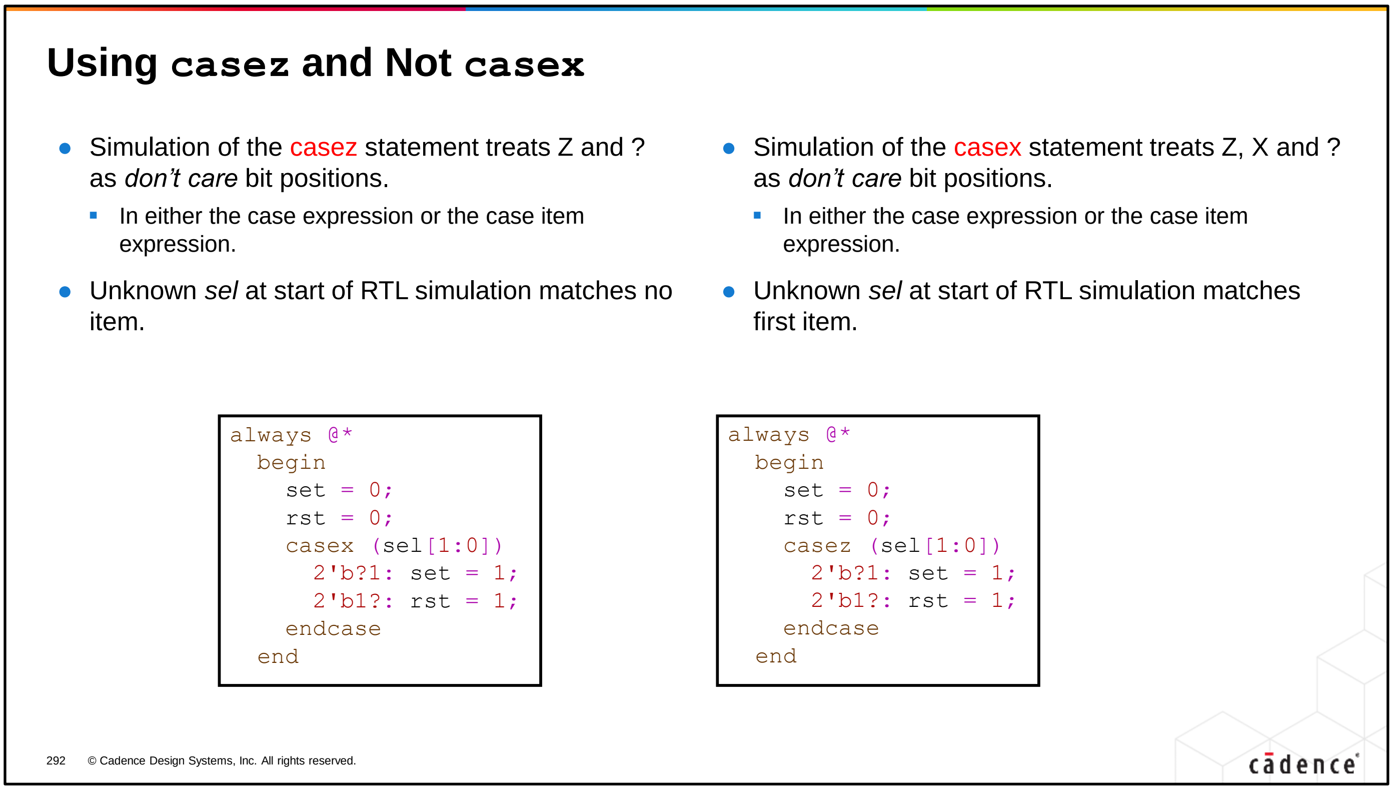
```
always @(a or b)
  if (b)
    y = a;
  else
    y = 'bx;
```



Don’t confirm x values!

In the real world, the unknown state does not exist. Digital design nodes are almost invariably in a binary state. Synthesis accepts assignment of the x character to indicate a don’t care value.

Assigning the unknown value can help debug your RTL design. You can, for example, initially assign an unknown value to a variable that should always get assigned some other value before it is tested. Just don’t expect the unknown value to show up in simulation of the post-synthesis netlist.



Using casez and Not casex

- Simulation of the casez statement treats Z and ? as *don't care* bit positions.
  - In either the case expression or the case item expression.
- Unknown *sel* at start of RTL simulation matches no item.

- Simulation of the casex statement treats Z, X and ? as *don't care* bit positions.
  - In either the case expression or the case item expression.
- Unknown *sel* at start of RTL simulation matches first item.

```
always @*
  begin
    set = 0;
    rst = 0;
    casex (sel[1:0])
      2'b?1: set = 1;
      2'b1?: rst = 1;
    endcase
  end
```

```
always @*
  begin
    set = 0;
    rst = 0;
    casez (sel[1:0])
      2'b?1: set = 1;
      2'b1?: rst = 1;
    endcase
  end
```

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

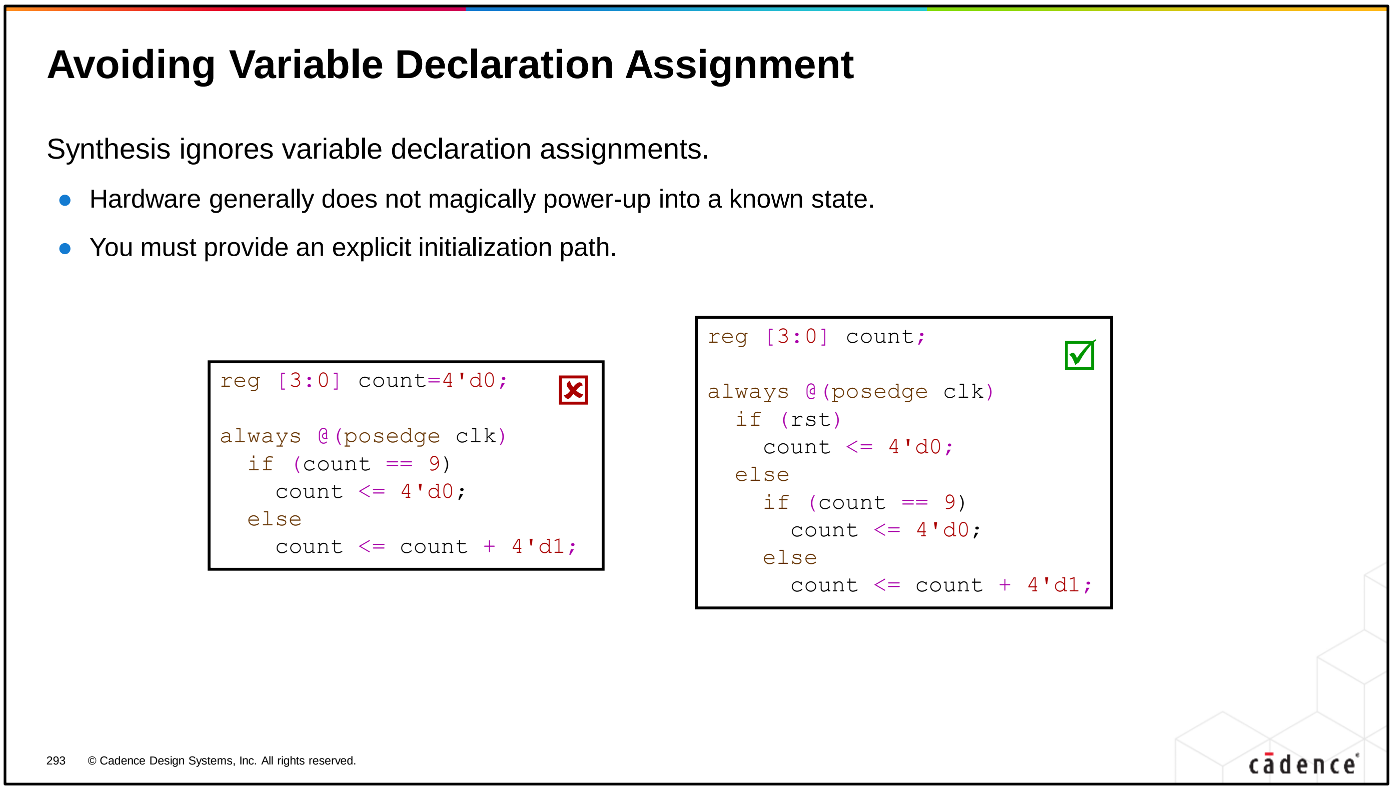
cadence

In the real world , the unknown state does not exist and the high-impedance state exists only extremely rarely. Digital design nodes are almost invariably in a binary state.

Synthesis accepts:

- Assignment of the x character to indicate a don't care value;
- Assignment of the z character to infer three-state logic;
- The characters "x, "z", and "?" in the casex item expression to designate bit positions to not participate in the match, that is, "don't care" bit positions;
- The characters z, and ? in the casez item expression to designate bit positions to not participate in the match, that is, don't care bit positions;

RTL simulation accepts don't care bit designations also in the case expression. At the start of RTL simulation, much of the design will be in the unknown state and very likely little or none of it in the high-impedance state. The casex statement treats an unknown sel signal as don't care for all bit positions, thus matching the first case item, whatever that might happen to be. The casez statement does a definitive match for the unknown state, which is unlikely to appear in any of the case items, thus leaving the block outputs with their default values.



Avoiding Variable Declaration Assignment

Synthesis ignores variable declaration assignments.

- Hardware generally does not magically power-up into a known state.
- You must provide an explicit initialization path.

```
reg [3:0] count=4'd0;

always @(posedge clk)
  if (count == 9)
    count <= 4'd0;
  else
    count <= count + 4'd1;
```

```
reg [3:0] count;

always @(posedge clk)
  if (rst)
    count <= 4'd0;
  else
    if (count == 9)
      count <= 4'd0;
    else
      count <= count + 4'd1;
```

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

Synthesis ignores variable declaration assignments. With the exception of some PALs, manufacturers generally do not guarantee that their components power up in any particular state, so variable declaration assignments are rarely truly a hardware construct. As a general statement, you must provide an explicit initialization path for any RTL that you mean to synthesize.

The variable declaration assignment construct is new to the Verilog-2001 update. You may find it useful in your testbench but must not use it in RTL meant for synthesis.

Remembering Delay Controls Are Not Synthesized

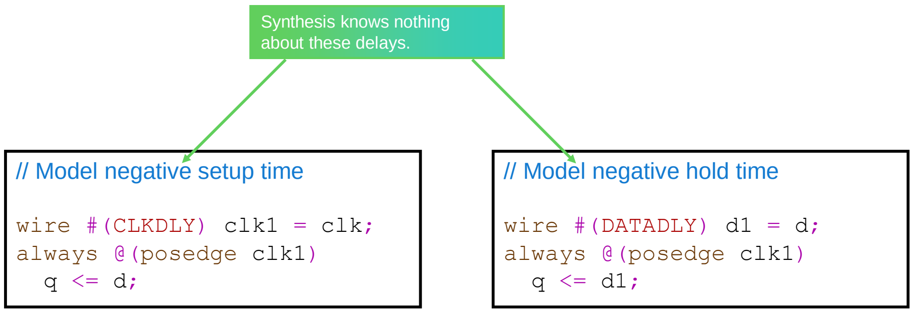
Synthesis *rejects* delay controls before the procedure event control.

Synthesis *ignores* delay controls after the procedure event control.

Synthesis *ignores* net delay and continuous assignment delay.

Time delay is *not* how you model timing for RTL synthesis.

- You specify arrival and departure times in a separate constraints file.



```
// Model negative setup time

wire #(CLKDLY) clk1 = clk;
always @(posedge clk1)
  q <= d;
```

```
// Model negative hold time

wire #(DATADLY) d1 = d;
always @(posedge clk1)
  q <= d1;
```

You can add timing delays to your RTL design to help you visualize the approximate delays of the post-synthesis netlist, but do not for even a moment imagine that this is how you specify the design timing for synthesis. Synthesis totally ignores RTL timing delays that do not violate the synthesis templates. You provide design timing information to synthesis in a separate file as constraints in the vendor’s syntax. While simulating the post-synthesis netlist, you may find that some “guesstimates” of the design timing are not as close as you would have liked.

Module Review

1. State and defend your opinion that the **full_case** attribute has an overall positive or negative impact on designer productivity.
2. State and explain the consequence of making an incomplete assignment in a subroutine.
3. State and defend your selection between the **casex** and **casez** statements for designs meant for synthesis.

This page does not contain notes.

Module Review Solutions

1. State and defend your opinion that the full_case attribute has an overall positive or negative impact on designer productivity.
   - The **full_case** attribute is equivalent to a **default** case match item that assigns “don’t-care” values to all case statement outputs. However, it less clearly states the designer’s intentions and does *not* prevent latch inference.
2. State and explain the consequence of making an incomplete assignment in a subroutine.
   - Synthesis tools infer a latch if you specify no output value for at least one combination of input values. This latch inference may be inadvertent but it is at least functionally correct. Most synthesis tools do *not* infer such latch if the incomplete assignment is in a subroutine. They instead combinationally generate an undetermined logic output value for the missing input combination.
3. State and defend your selection between the **casex** and **casez** statements for designs meant for synthesis.
   - Synthesis accepts in a **casex** match expression the “x”, “z”, and “?” characters, and in a **casez** match expression the “z”, and “?” characters to indicate bit positions that do not participate in the match. Simulation accepts these designations also in the **casex** and **casez** expression. At the start of RTL simulation, that expression is likely to be of unknown value, thus matching the *first* **casex** item and *no* **casez** item. At the start of post-synthesis simulation, that expression is likely to be of arbitrary value, thus matching *some* **casex** or **casez** item.

This page does not contain notes.

Module Exercise

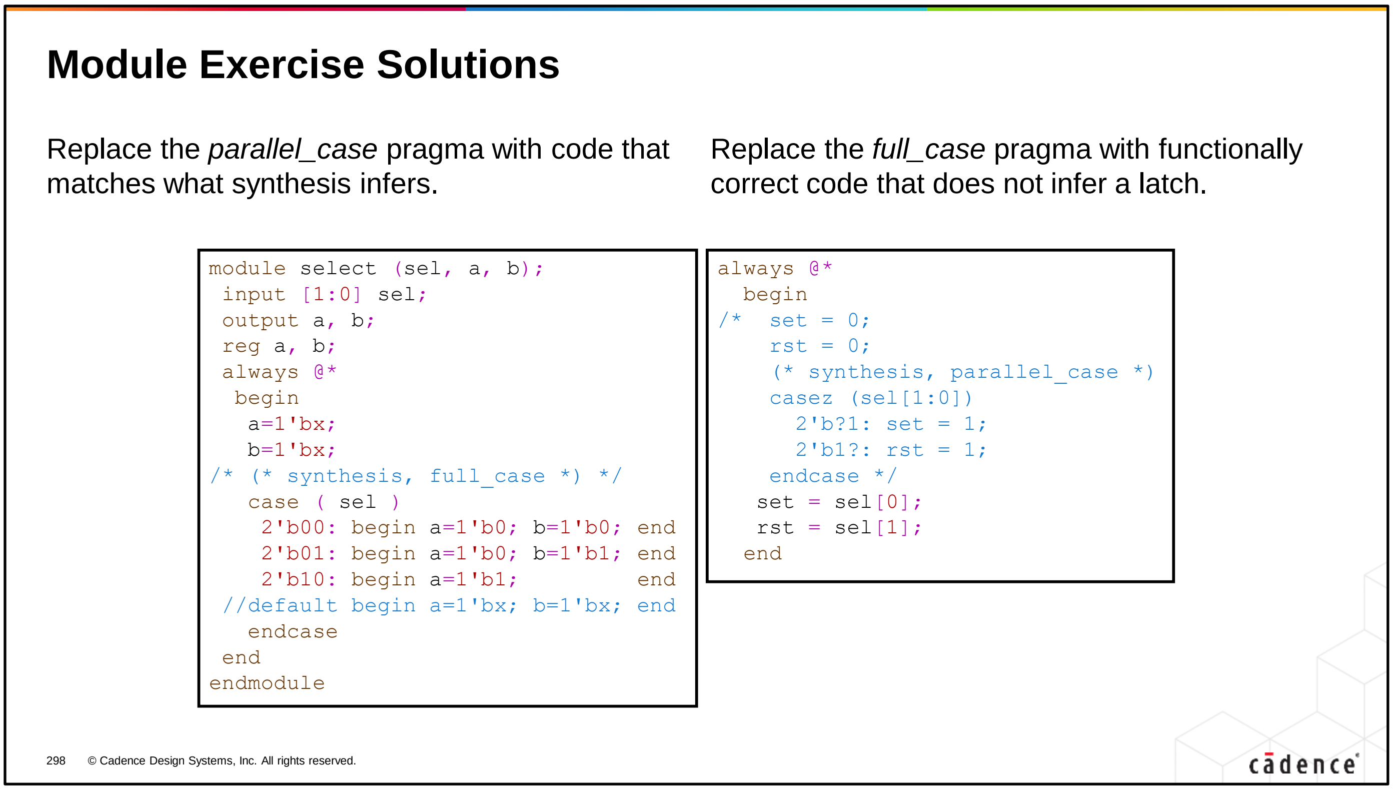
Replace the *parallel_case* pragma with code that matches what synthesis infers.

Replace the *full_case* pragma with functionally correct code that does not infer a latch.

```
module select (sel, a, b);
 input [1:0] sel;
 output a, b;
 reg a, b;
 always @*
  (* synthesis, full_case *)
  case ( sel )
   2'b00: begin a=1'b0; b=1'b0; end
   2'b01: begin a=1'b0; b=1'b1; end
   2'b10: begin a=1'b1;         end
//default begin a=1'bx; b=1'bx; end
  endcase
endmodule
```

```
always @*
  begin
    set = 0;
    rst = 0;
    (* synthesis, parallel_case *)
    casez (sel[1:0])
      2'b?1: set = 1;
      2'b1?: rst = 1;
    endcase
  end
```

This page does not contain notes.



Module Exercise Solutions

Replace the *parallel_case* pragma with code that matches what synthesis infers.

Replace the *full_case* pragma with functionally correct code that does not infer a latch.

```
module select (sel, a, b);
 input [1:0] sel;
 output a, b;
 reg a, b;
 always @*
  begin
   a=1'bx;
   b=1'bx;
/* (* synthesis, full_case *) */
   case ( sel )
    2'b00: begin a=1'b0; b=1'b0; end
    2'b01: begin a=1'b0; b=1'b1; end
    2'b10: begin a=1'b1;         end
 //default begin a=1'bx; b=1'bx; end
   endcase
 end
endmodule
```

```
always @*
  begin
/*   set = 0;
     rst = 0;
     (* synthesis, parallel_case *)
     casez (sel[1:0])
       2'b?1: set = 1;
       2'b1?: rst = 1;
     endcase */
    set = sel[0];
    rst = sel[1];
  end
```

This page does not contain notes.

Labs

There are no labs in this module.

This page does not contain notes.