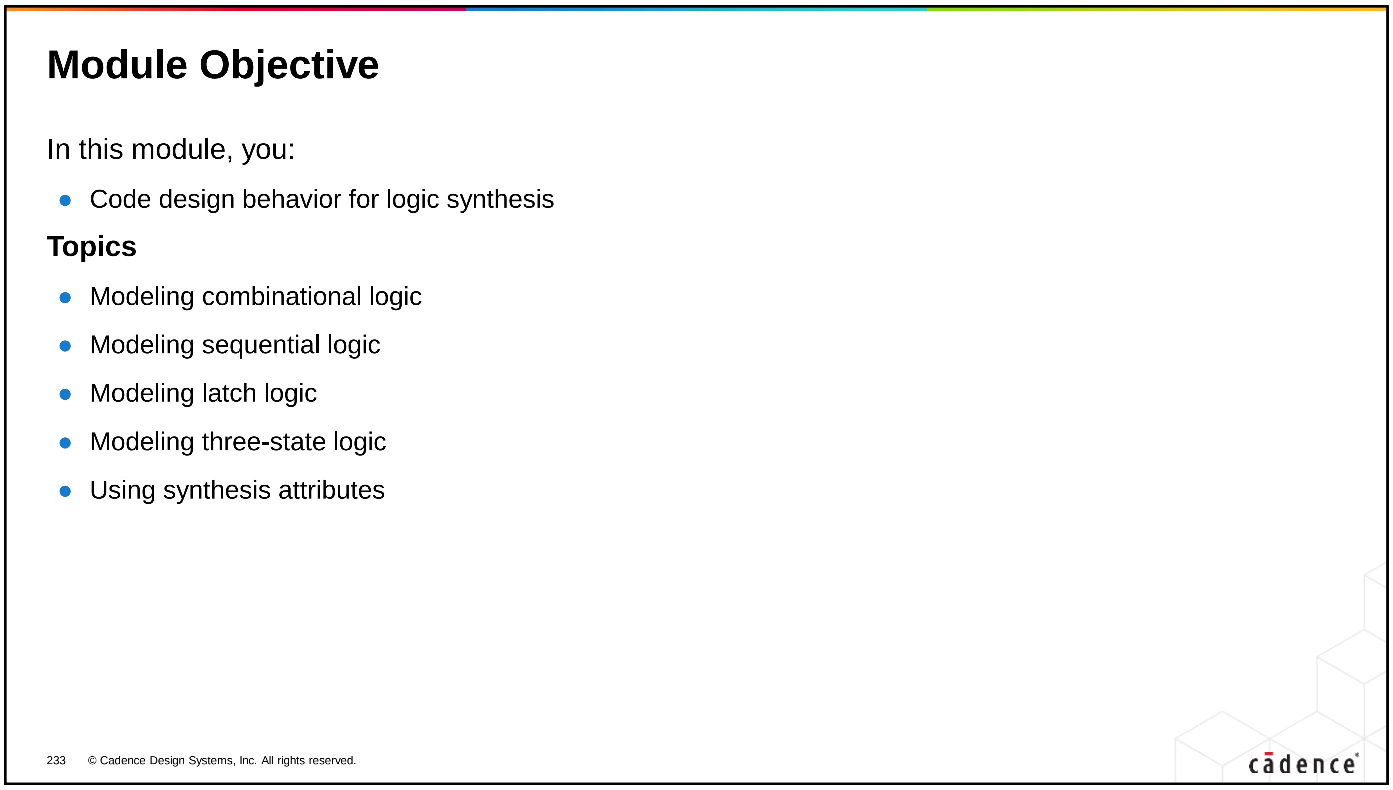
Module 13

Coding RTL for Synthesis

cādence®

This module intends to provide you with the information you need to code your RTL design in compliance with the IEEE Std 1364.1-2002 for Verilog RTL synthesis.



Module Objective

In this module, you:

- Code design behavior for logic synthesis

Topics

- Modeling combinational logic
- Modeling sequential logic
- Modeling latch logic
- Modeling three-state logic
- Using synthesis attributes

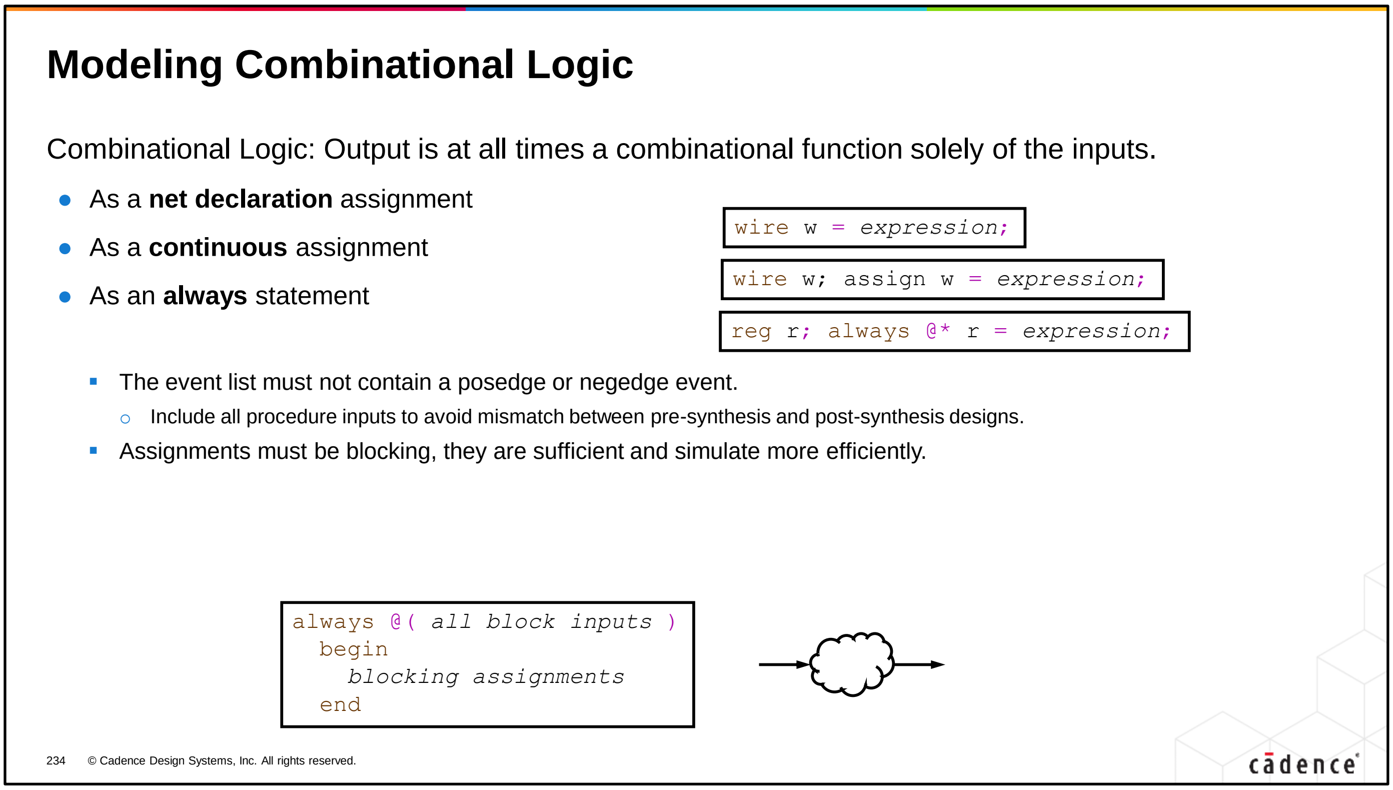
233 © Cadence Design Systems, Inc. All rights reserved.

cadence

Your objective is to code design behavior for logic synthesis.

To do that, you need to know about:

- Modeling combinational logic;
- Modeling sequential logic;
- Modeling latch logic;
- Modeling three-state logic; and
- Using synthesis attributes.



Modeling Combinational Logic

Combinational Logic: Output is at all times a combinational function solely of the inputs.

- As a **net declaration** assignment
- As a **continuous** assignment
- As an **always** statement

```
wire w = expression;
```

```
wire w; assign w = expression;
```

```
reg r; always @* r = expression;
```

  - The event list must not contain a posedge or negedge event.
    - Include all procedure inputs to avoid mismatch between pre-synthesis and post-synthesis designs.
  - Assignments must be blocking, they are sufficient and simulate more efficiently.

```
always @( all block inputs )
  begin
    blocking assignments
  end
```

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

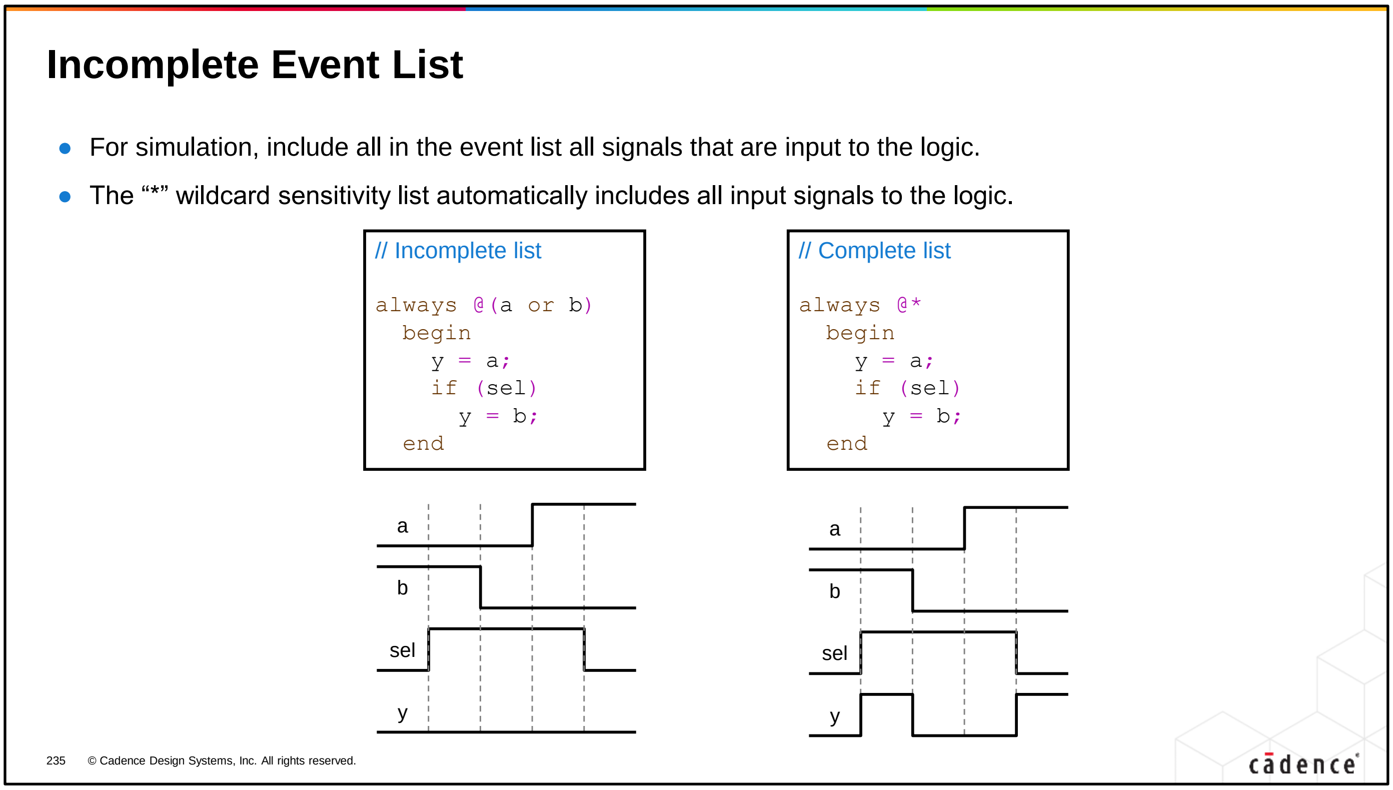
cādence

You can model combinational logic with a net declaration assignment or a continuous assignment or an always statement.

When using an always statement, the single event list must not contain a posedge or negedge event. The event list does not otherwise 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 (@*).

You must not in an always statement assign a variable using both a blocking assignment (=) and a nonblocking assignment (<=). The blocking assignment is sufficient for a description of combinational logic and simulates more efficiently than the nonblocking assignment.

“Combinational logic shall be modeled using a continuous assignment or a net declaration assignment or an always statement. ... A variable assigned in an always statement shall not be assigned using both a blocking assignment (=) and a nonblocking assignment (<=) in the same always statement. ... The event list for a combinational logic model shall not contain the reserved words posedge or negedge.” – IEEE Std 1364.-2002 Section 5.1 Modeling combinational logic

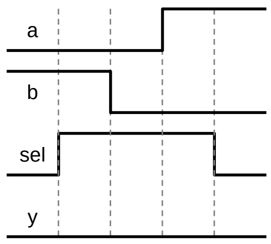


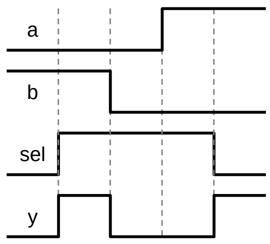
Incomplete Event List

- For simulation, include all in the event list all signals that are input to the logic.
- The “*” wildcard sensitivity list automatically includes all input signals to the logic.

```
// 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
```





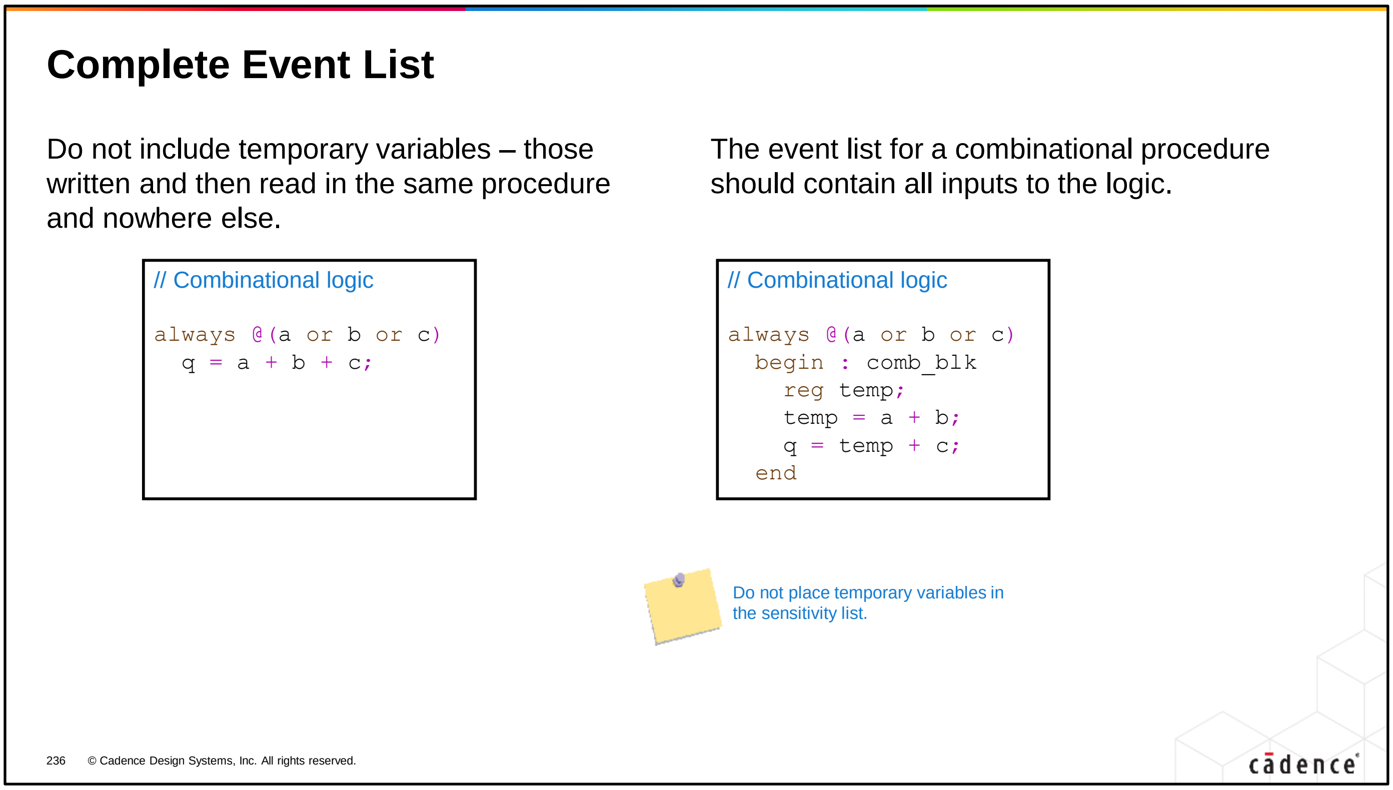
235 © Cadence Design Systems, Inc. All rights reserved.

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 affect 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 effect.
- 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



Complete Event List

Do not include temporary variables – those written and then read in the same procedure and nowhere else.

```
// Combinational logic

always @(a or b or c)
  q = a + b + c;
```

The event list for a combinational procedure should contain all inputs to the logic.

```
// Combinational logic

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

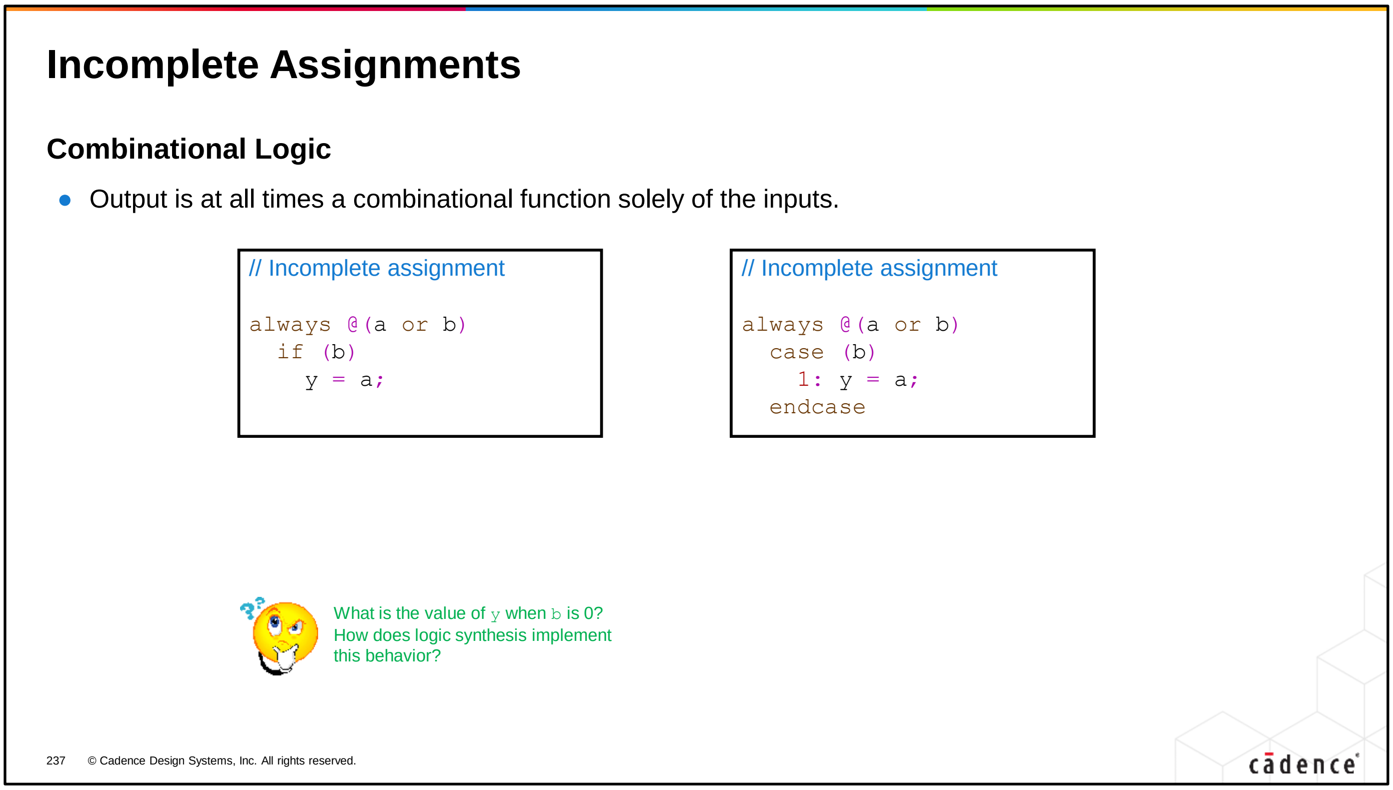
Do not place temporary variables in the sensitivity list.

The Verilog language lets you place any signals in the sensitivity list and to freely mix blocking and nonblocking assignments anywhere in the procedure. A simulation tool simply executes the procedure as you wrote it, using simulation semantics. However, to generate RTL simulation results consistent with those of the postsynthesis netlist, some guidelines exist:

- All edges of all signals input to combinational logic must be present in the sensitivity list. This is due to the rule that any changes on any inputs to the logic must have the opportunity to immediately affect the output. Do not place temporary variables in the sensitivity list. A temporary variable is one that the procedure writes only before it reads and that no other procedure uses. It is not an input to the logic.
- Do not mix blocking and nonblocking assignments to the same variable. Even better, you should use only blocking assignments within procedures that represent purely combinational logic. This is a recommendation to obtain higher simulation performance, as nonblocking assignments are not necessary and simulate more slowly.

The synthesis tool requires code that unambiguously states your design intentions. Code meant for the synthesis tool may use only a subset of the Verilog constructs and coding styles.

- For synthesis, it is an absolute requirement that you do not mix blocking and nonblocking assignments to the same variable!
- 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.



Incomplete Assignments

Combinational Logic

- Output is at all times a combinational function solely of the inputs.

```
// Incomplete assignment

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

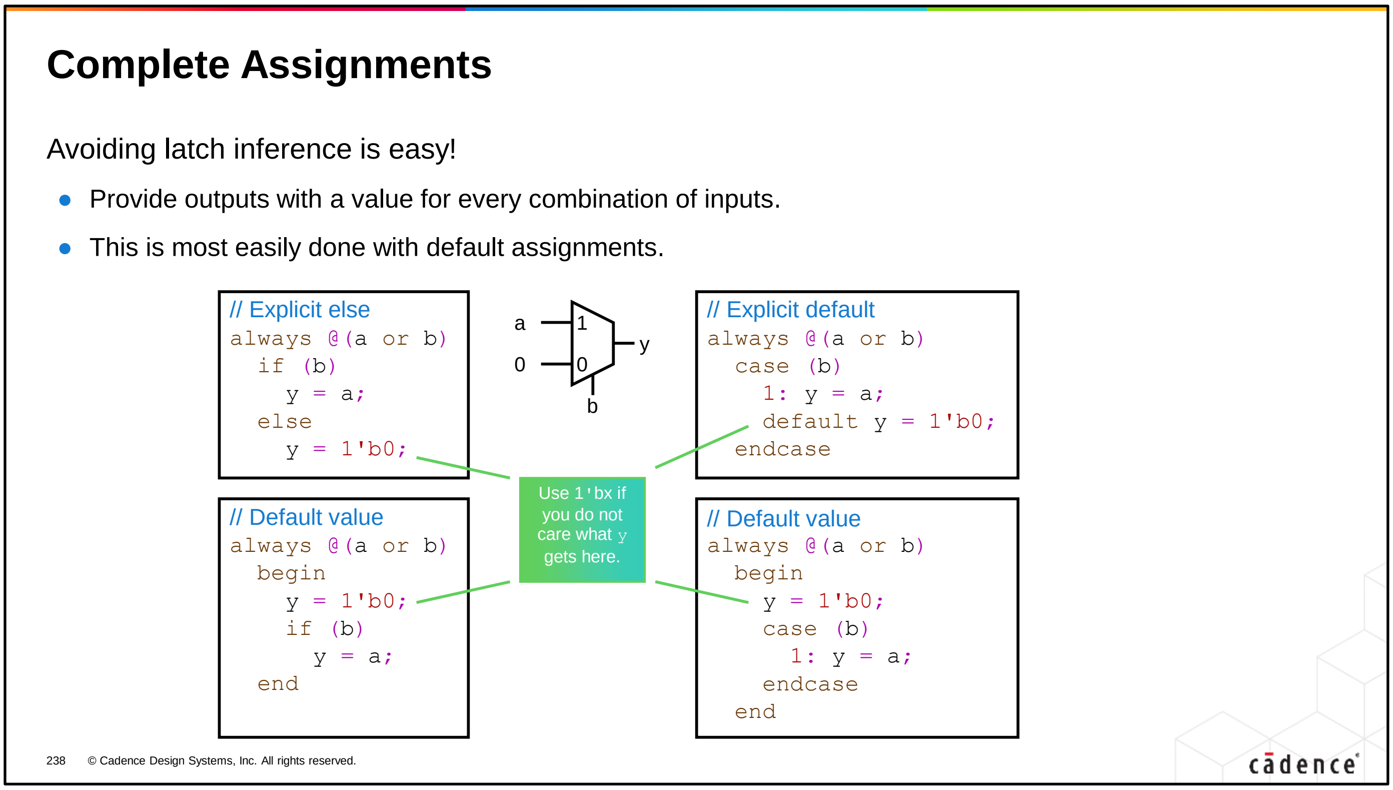
```
// Incomplete assignment

always @(a or b)
  case (b)
    1: y = a;
  endcase
```

What is the value of y when b is 0?
How does logic synthesis implement this behavior?

If an execution path through a combinational procedure exists that does not update the value of some output, then that output variable must retain its previous value. The synthesis tool infers a latch to implement this behavior. Latch inference is almost always not intended, and you can easily avoid it.

This example fails to update the y output variable when the b input is not 1.

How would you modify this code to ensure inference of purely combinational logic?

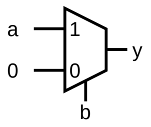


Complete Assignments

Avoiding latch inference is easy!

- Provide outputs with a value for every combination of inputs.
- This is most easily done with default assignments.

```
// Explicit else
always @(a or b)
  if (b)
    y = a;
  else
    y = 1'b0;
```



```
// Explicit default
always @(a or b)
  case (b)
    1: y = a;
    default y = 1'b0;
  endcase
```

Use 1'bx if you do not care what y gets here.

```
// Default value
always @(a or b)
  begin
    y = 1'b0;
    if (b)
      y = a;
  end
```

```
// Default value
always @(a or b)
  begin
    y = 1'b0;
    case (b)
      1: y = a;
    endcase
  end
```

Here are two methods you can use to prevent latch inference:

- You can for an if statement use an explicit else clause and for a case statement an explicit default match item; or
- You can provide default values for all procedure outputs at the start of the procedure.

Which is the best technique in a real design?

If you have a procedure with a complex set of conditional assignments, you can easily miss an assignment for one or more of these branches. Making default assignments at the start of the procedure ensures that all procedure outputs have an assignment.

“The value x may be used as a primary on the RHS of an assignment to indicate a don’t care value for synthesis.” – IEEE Std. 1364.1-2002 Section 5.5 Support for values x and z

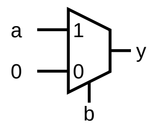
Continuous Assignments

Continuous assignments drive values onto nets.

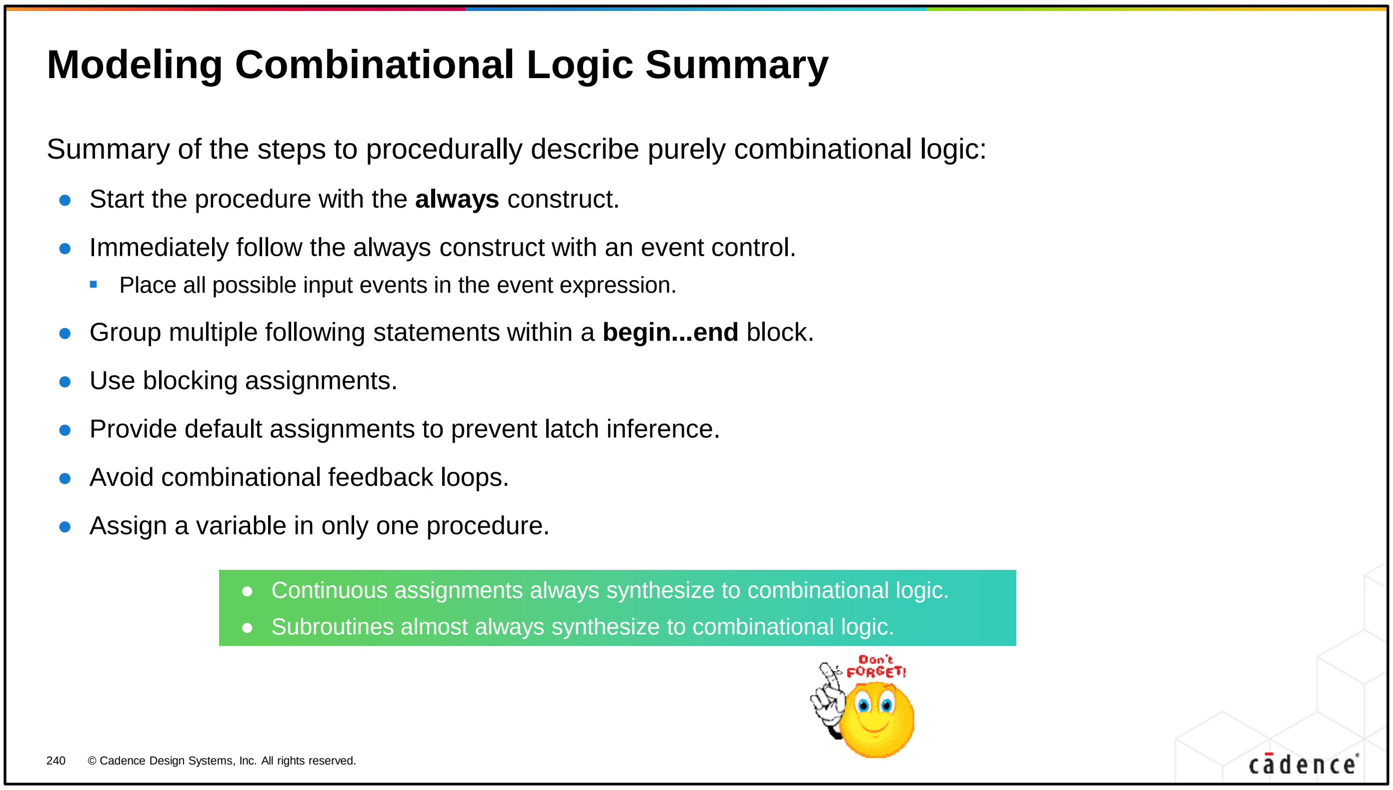
Continuous assignments always represent combinational logic.

- Impossible to not assign a value for some input combination.
- Output is always a combinational function of current inputs.

```
assign y = b ? a : 1'b0;
```



Continuous assignments and net declaration assignments drive values onto nets. These assignments always represent combinational logic, as it is syntactically impossible to not assign a value for some input combination, thus the output is always a combinational function of the current inputs.



Modeling Combinational Logic Summary

Summary of the steps to procedurally describe purely combinational logic:

- Start the procedure with the **always** construct.
- Immediately follow the always construct with an event control.
  - Place all possible input events in the event expression.
- Group multiple following statements within a **begin...end** block.
- Use blocking assignments.
- Provide default assignments to prevent latch inference.
- Avoid combinational feedback loops.
- Assign a variable in only one procedure.

- Continuous assignments always synthesize to combinational logic.
- Subroutines almost always synthesize to combinational logic.

Here is a summary of the steps to procedurally describe purely combinational logic:

- Second level
- Start the procedure with the always construct and immediately follow the always construct with an event control, placing all possible input events in the event expression.
- Group multiple following statements within a begin...end block. You can omit the begin and end keywords if you have only a single statement.
- Make blocking assignments. You can equally validly make nonblocking assignments, but they simulate less efficiently and provide no additional benefit. In any case, do not mix blocking and nonblocking assignments to the same variable and do not make assignments to a variable from multiple procedures.
- Provide default assignments at the start of the procedure to prevent latch inference.
- Avoid combinational feedback loops.

Remember that:

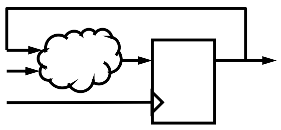
- Continuous assignments always synthesize to combinational logic.
- Functions almost always synthesize to combinational logic.

Modeling Sequential Logic

Sequential Logic

- Outputs are sampled in registers on a clock edge, thus storage is required.
- As an always statement:
  - The event list must contain only posedge and negedge events.
    - One represents the active clock edge and others represent asynchronous set and reset.
  - Model set/reset behaviors in an if statement early branches and normal behavior in the later branches.
  - Make nonblocking assignments to storage variables.

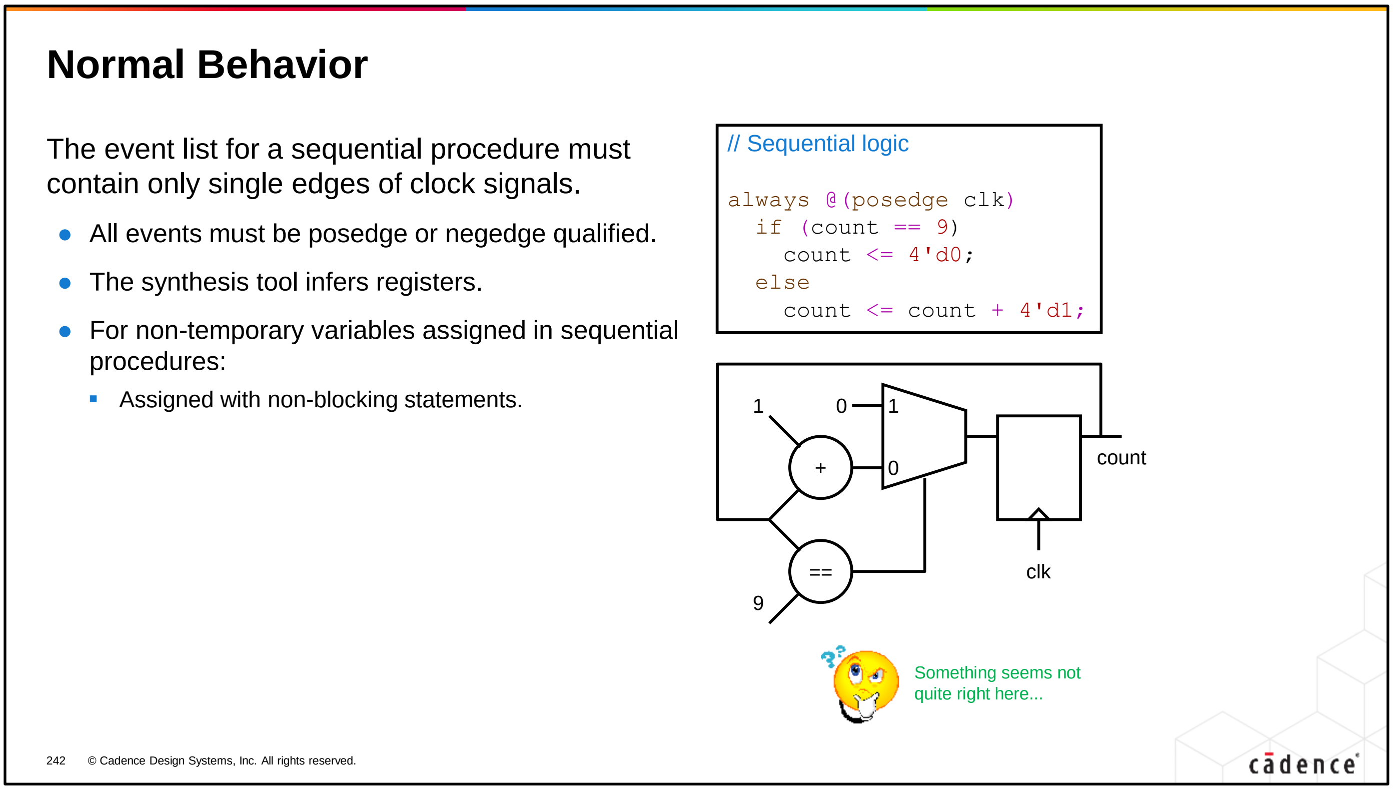
```
always @( clock edges )
  begin
    nonblocking assignments
  end
```



You model sequential logic using an always statement that has one or more posedge or negedge events in exactly one event list. Exactly one of those events represents the active clock edge that stores the value. Any additional posedge or negedge events represent asynchronous set and reset behaviors. The event list must not contain level-sensitive events.

You model the asynchronous set and reset behaviors by using an if statement. In the if statement, you model the asynchronous behaviors in one or more conditional branches. The unconditional last else branch models the normal sequential behavior.

To avoid simulation clock/data races, you should make only nonblocking assignments to variables that represent storage. You make blocking assignments to temporary variables. Temporary variables are those written and then read in the same procedure and nowhere else.

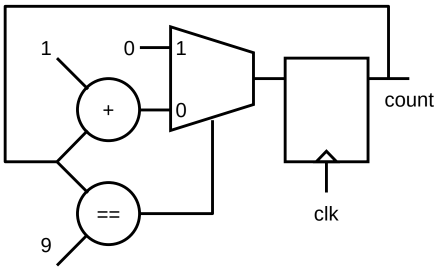


Normal Behavior

The event list for a sequential procedure must contain only single edges of clock signals.

- All events must be posedge or negedge qualified.
- The synthesis tool infers registers.
- For non-temporary variables assigned in sequential procedures:
  - Assigned with non-blocking statements.

```
// Sequential logic

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



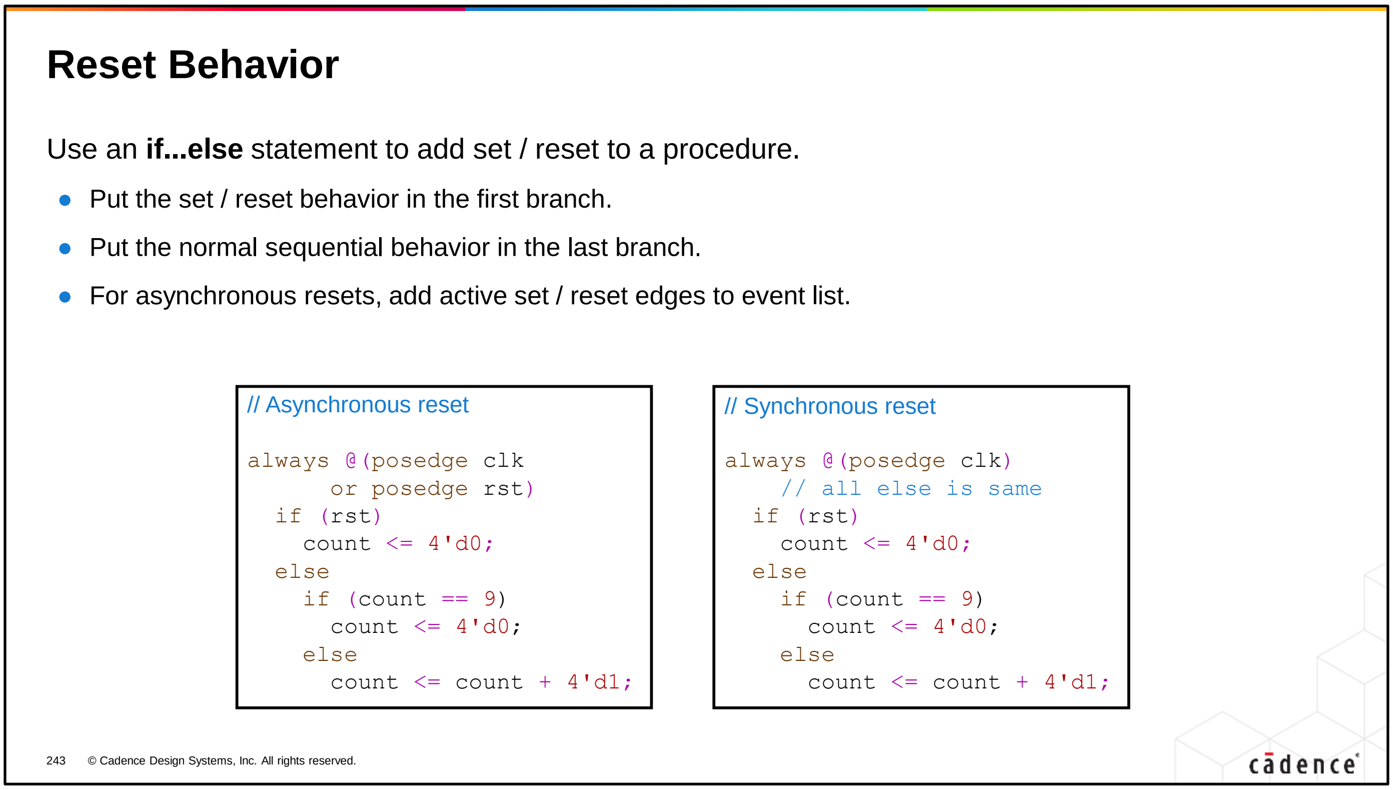
Something seems not quite right here...

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

Synthesis tools recognize sequential procedures by looking for a particular code template – in this case, a procedure with an event list containing only edge-qualified signals. Although some synthesis tools may support other coding styles for sequential procedures, your adherence to this standard produces code that you can port between all synthesis tools compliant with the standard.

This example has only the positive edge of the clock in its event list. All storage inferred for non-temporary variables that this procedure writes will have a rising active clock edge. The if statement describes the combinational logic calculating the new “count” value that is stored on the next rising clock edge.

Note that this example lacks a reset to initialize the count value!



Reset Behavior

Use an **if...else** statement to add set / reset to a procedure.

- Put the set / reset behavior in the first branch.
- Put the normal sequential behavior in the last branch.
- For asynchronous resets, add active set / reset edges to event list.

```
// Asynchronous reset

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

```
// Synchronous reset

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

You can most easily provide set and reset behaviors by using an if statement. This is a requirement for asynchronous set and reset behaviors and strongly recommended for synchronous set and reset behaviors. The synthesis tool will treat synchronous set and reset behaviors you model outside an if statement as just that much more combinational logic.

In the if statement, place the set and reset behaviors in their order of priority in the first conditional branches, and place the normal synchronous behavior in the unconditional last else branch. Do not make assignments to the storage variable outside of the if statement.

For synchronous set and reset, trigger the procedure on only the clock edge.

For asynchronous set and reset, trigger the procedure also on the active set and reset edge(s).

Code synchronously reset, asynchronously reset and not reset registers in separate procedures.

Sequential Procedure Templates

Code must follow these templates:

- Procedure starts with an always construct.
- Followed by one event control containing only edge-qualified signals.
  - Clock
  - Asynchronous set or reset

```
always @(posedge clk)
  begin
    // normal
    // sequential
    // behavior
  end
```

```
always @(posedge clk)
  if (!rst)
    begin
      // synchronous
      // reset
      // behavior
    end
  else
    begin
      // normal
      // sequential
      // behavior
    end
```

```
always @(posedge clk
      or negedge rst)
  if (!rst)
    begin
      // asynchronous
      // reset
      // behavior
    end
  else
    begin
      // normal
      // sequential
      // behavior
    end
```

Sequential procedures have the clock edge in the event list, and also the set and reset edge(s) if there is asynchronous set or reset.

Separately code your not reset, synchronously reset and asynchronously reset procedures.

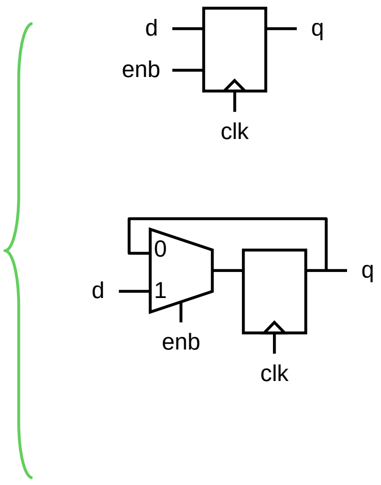
Incomplete Assignments

Sequential Logic

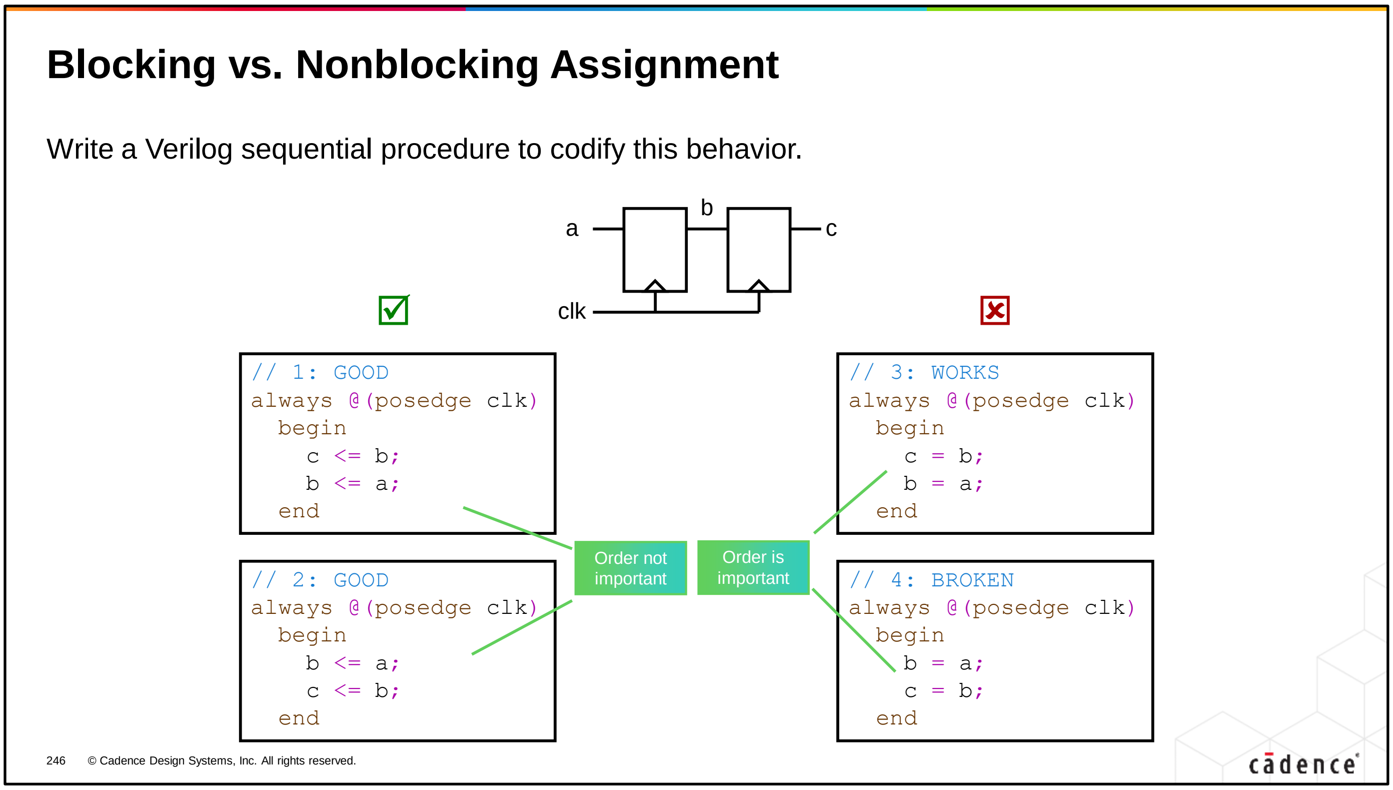
- Output is not at all times a combinational function solely of the inputs.
  - Implies some sort of storage.
- Incomplete assignment in a sequential procedure does not infer a latch.
  - Storage is already there!

```
always @(posedge clk)
  if (enb)
    q <= d;
```

Potential Implementations

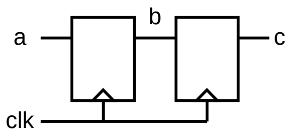


If procedure execution does not update a variable, then the variable value is not changed. In procedures representing combinational logic, this infers a latch to store the previous variable value. For procedures representing synchronous logic, the variable value is stored in the inferred register. For these procedures you do not use default assignment or else clauses to prevent latch inference.



Blocking vs. Nonblocking Assignment

Write a Verilog sequential procedure to codify this behavior.



✓

```
// 1: GOOD
always @(posedge clk)
  begin
    c <= b;
    b <= a;
  end
```

```
// 2: GOOD
always @(posedge clk)
  begin
    b <= a;
    c <= b;
  end
```

Order not important

Order is important

```
// 3: WORKS
always @(posedge clk)
  begin
    c = b;
    b = a;
  end
```

```
// 4: BROKEN
always @(posedge clk)
  begin
    b = a;
    c = b;
  end
```

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

Good coding practice demands that you make only nonblocking assignments to variables that represent storage elements. This is to avoid clock/data races in simulation and to avoid unintended logic inference in synthesis. The synthesis standard requires only that you not mix the two types of assignment to the same variable.

For examples 1 and 2, you can see that with nonblocking assignments the order of assignments is not important, as simulation schedules the actual assignments for the NBA region of the stratified event queue.

For examples 3 and 4, you can see that with blocking assignments the order of assignments is important, as simulation completes the assignments as it executes the statements. Different order generates different simulation results and different synthesis results.

- Example 3 reads variable b before writing it, so synthesis infers two registers;
- Example 4 reads variable b only after writing it. Variable b is a temporary variable. Synthesis infers one register.

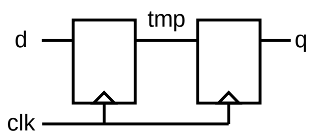
“Nonblocking procedural assignments should be used for variables that model edge-sensitive storage devices.” – IEEE Std. 1364.-2002 Section 5.2.2 Modeling edge-sensitive storage devices

Temporary Variables in Sequential Procedures

Temporary Variable

- Written first and then read (in same procedure).
- Variable is alias for expression so no register is inferred.

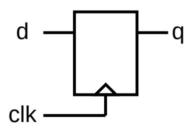
```
// 2 flop
always @(posedge clk)
  begin: dff2
    reg tmp;
    q <= tmp;
    tmp = d;
  end
```



Persistent Variable

- Read first and then written (in same procedure).
- Synthesis must infer a register to hold the value to the next read.

```
// 1 flop
always @(posedge clk)
  begin: dff1
    reg tmp;
    tmp = d;
    q <= tmp;
  end
```

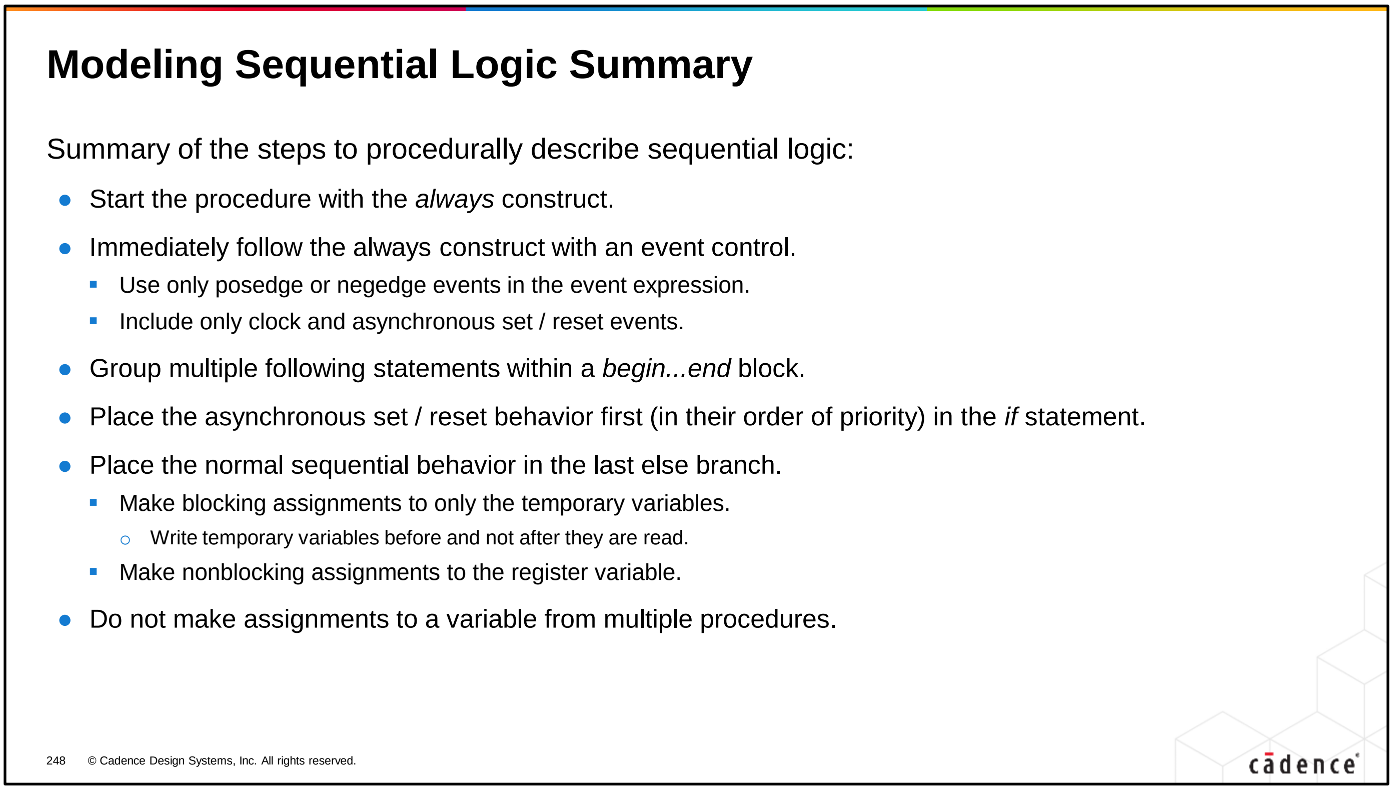


You can use a temporary variable to hold the intermediate result of a calculation before you use the result later in the procedure. You use temporary variables to break up complex expressions and combinational logic into a series of smaller steps – making complex logic easier to describe, understand and maintain.

A temporary variable is one that a procedure writes with a blocking assignment only before the procedure reads it, and no other procedure uses it. You can declare temporary variables locally to ensure that no other procedure uses it.

A typical use model would be to make blocking assignments to temporary variables and then make nonblocking assignment of the temporary variable values to other variables.

"Blocking procedural assignments may be used for variables that are temporarily assigned and used within an always statement." – IEEE Std. 1364.-2002 Section 5.2.2 Modeling edge-sensitive storage devices



Modeling Sequential Logic Summary

Summary of the steps to procedurally describe sequential logic:

- Start the procedure with the *always* construct.
- Immediately follow the always construct with an event control.
  - Use only posedge or negedge events in the event expression.
  - Include only clock and asynchronous set / reset events.
- Group multiple following statements within a *begin...end* block.
- Place the asynchronous set / reset behavior first (in their order of priority) in the *if* statement.
- Place the normal sequential behavior in the last else branch.
  - Make blocking assignments to only the temporary variables.
    - Write temporary variables before and not after they are read.
  - Make nonblocking assignments to the register variable.
- Do not make assignments to a variable from multiple procedures.

Here is a summary of the steps to procedurally describe sequential logic:

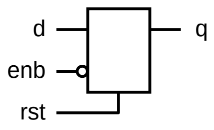
- Start the procedure with the always construct and immediately follow the always construct with an event control, placing only edge-qualified events in the event expression.
- Group multiple following statements within a begin...end block. You can omit the begin and end keywords if you have only a single statement.
- Make only blocking assignments to temporary variables and make only nonblocking assignments to variables representing storage. Write the temporary variables only before you read them in the same procedure. Do not make assignments to a variable from multiple procedures.

Modeling Latch Logic

But what if you want to infer a latch?

- A combinational block that for some combination of inputs does not provide an output value infers storage, i.e., latch.
- Make assignments for latch logic as you do for sequential logic.
  - Make blocking assignments to only the temporary variables.
    - Write temporary variables only before they are read.
  - Make nonblocking assignments to the latch variable.

```
always @(enb or rst or d)
  begin: latch
    reg tmp;
    tmp = d;
    if (rst)
      q <= 0;
    else
      if (enb)
        q <= tmp;
  end
```



Latch inference is almost always a mistake – be sure to document where and why you did it!

You deliberately infer a latch, if you truly want to, the same way you inadvertently infer a latch, with the exception that you remember to make nonblocking assignments to the variable that represents the storage.

A latch-based design can have higher performance than a register-based design, but can also be more difficult to correctly design and debug. Use of latches is typically infrequent and reserved for extenuating circumstances.

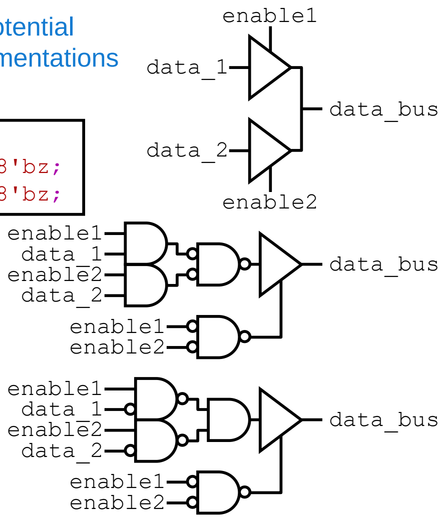
Modeling Three-State Logic

Synthesis infers three-state logic when you assign a net or variable the high-impedance value.

```
wire data_bus;
assign data_bus = enable1 ? data_1 : 8'bz;
assign data_bus = enable2 ? data_2 : 8'bz;
```

```
reg data_bus;
always @*
  begin
    if (enable1)
      data_bus = data_1;
    else
      data_bus = 8'bz;
    if (enable2)
      data_bus = data_2;
    else
      data_bus = 8'bz;
  end
```

Potential Implementations



The standard does not specify an implementation.

Ensure the enable signals are mutually exclusive!

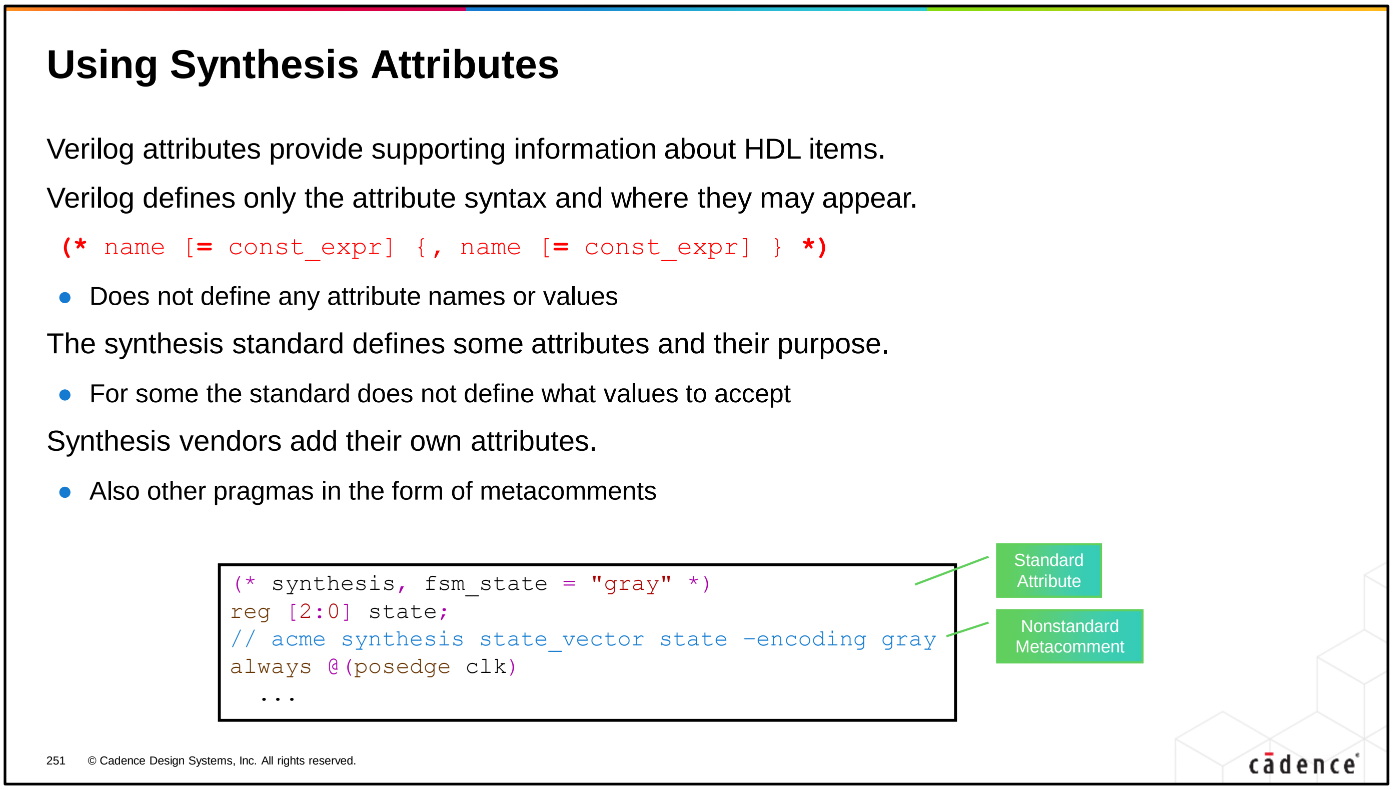
You model three-state logic by assigning the high-impedance (z) value to a net or variable. Any other assignments to that net or variable must also assign the high-impedance value. Further propagating the high-impedance value by use of net or variable assignments does not also make that downstream logic into three-state logic.

This example codes behavior representing three-state drivers in a form that synthesis can recognize. For this example:

- The “enable1” signal when high drives “data_1” onto the data bus; and
- The “enable2” signal when high drives “data_2” onto the data bus.

The synthesis standard does not address what the result should be if multiple enables are simultaneously true.

“Three-state logic shall be modeled when a variable is assigned the value z.” – IEEE Std 1364-2002 Section 5.4 Modeling three-state drivers.



Using Synthesis Attributes

Verilog attributes provide supporting information about HDL items.

Verilog defines only the attribute syntax and where they may appear.

```
(* name [= const_expr] {, name [= const_expr] } *)
```

- Does not define any attribute names or values

The synthesis standard defines some attributes and their purpose.

- For some the standard does not define what values to accept

Synthesis vendors add their own attributes.

- Also other pragmas in the form of metacomments

```
(* synthesis, fsm_state = "gray" *)
reg [2:0] state;
// acme synthesis state_vector state -encoding gray
always @(posedge clk)
  ...
```

Standard Attribute

Nonstandard Metacomment

The IEEE Std. 1364-2001 Verilog HDL defines an attribute construct as a means to provide supporting information about an HDL construct to the various tools reading the HDL. The Verilog standard defines only the attribute syntax and where they may appear, and does not define any particular attributes.

The IEEE Std. 1364.1-2002 for Verilog RTL synthesis defines some synthesis attributes and requires that if a compliant tool supports any means to provide this information then it must support provision my means of the attributes.

As synthesis existed for many years prior to the Verilog RTL synthesis standard, synthesis vendors typically do still offer their own means to provide this information as metacomments. The industry collectively refers to these attributes and metacomments as “pragmas”.

This section will describe only a few of the standard synthesis attributes.

Attributes: “...properties about objects, statements and groups of statements in the HDL source...” – IEEE Std. 1364-2001 Section 2.8 Attributes

“An attribute instance can appear in the Verilog description as a prefix attached to a declaration, a module item, a statement, or a port connection. It can appear as a suffix to an operator or a Verilog function name in an expression.” – IEEE Std. 1364-2001 Section 2.8 Attributes

“A pragma is a generic term used to define a construct with no predefined language semantics that influences how a synthesis tool should synthesize Verilog HDL code into a circuit.” – IEEE Std. 1364.1-2002 Section 6 Pragmas

“If a synthesis tool supports pragmas to control the structure of the synthesized netlist or to give direction to the synthesis tool, attributes shall be used to convey the required information.” – IEEE Std. 1364.1-2002 Section 6.1 Synthesis attributes.

Pragma “full_case”

Synthesis tools accept a pragma to complete a **case** statement.

The directive applies to the case item expressions, not their statements!

```
module full (
  output reg a, b,
  input [1:0] sel
);
  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; b = ....  end
    // default begin a = 1'bx; b = 1'bx; end
      endcase
    end
endmodule
```

Better Solution

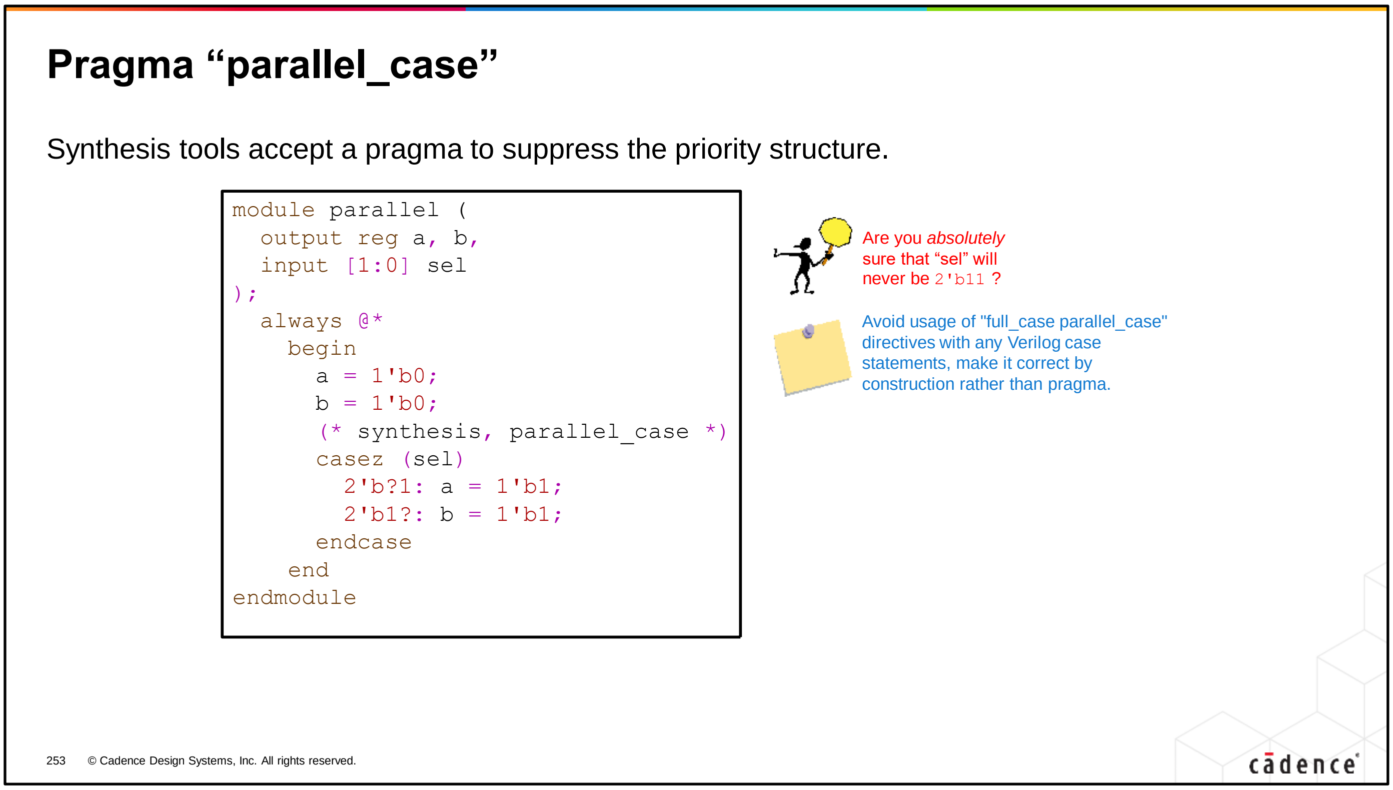
Pragma

Without default value assignment you can get a latch anyway!

Equivalent to Pragma

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

The full_case attribute directs the synthesis tool to assign don’t-care values to case statement outputs for all unspecified case choices. Synthesis can still infer a latch for outputs not assigned by all specified case items. Your use of the full_case attribute thus promotes inadvertent errors. A better solution is to explicitly provide default values for all procedure outputs.



Pragma “parallel_case”

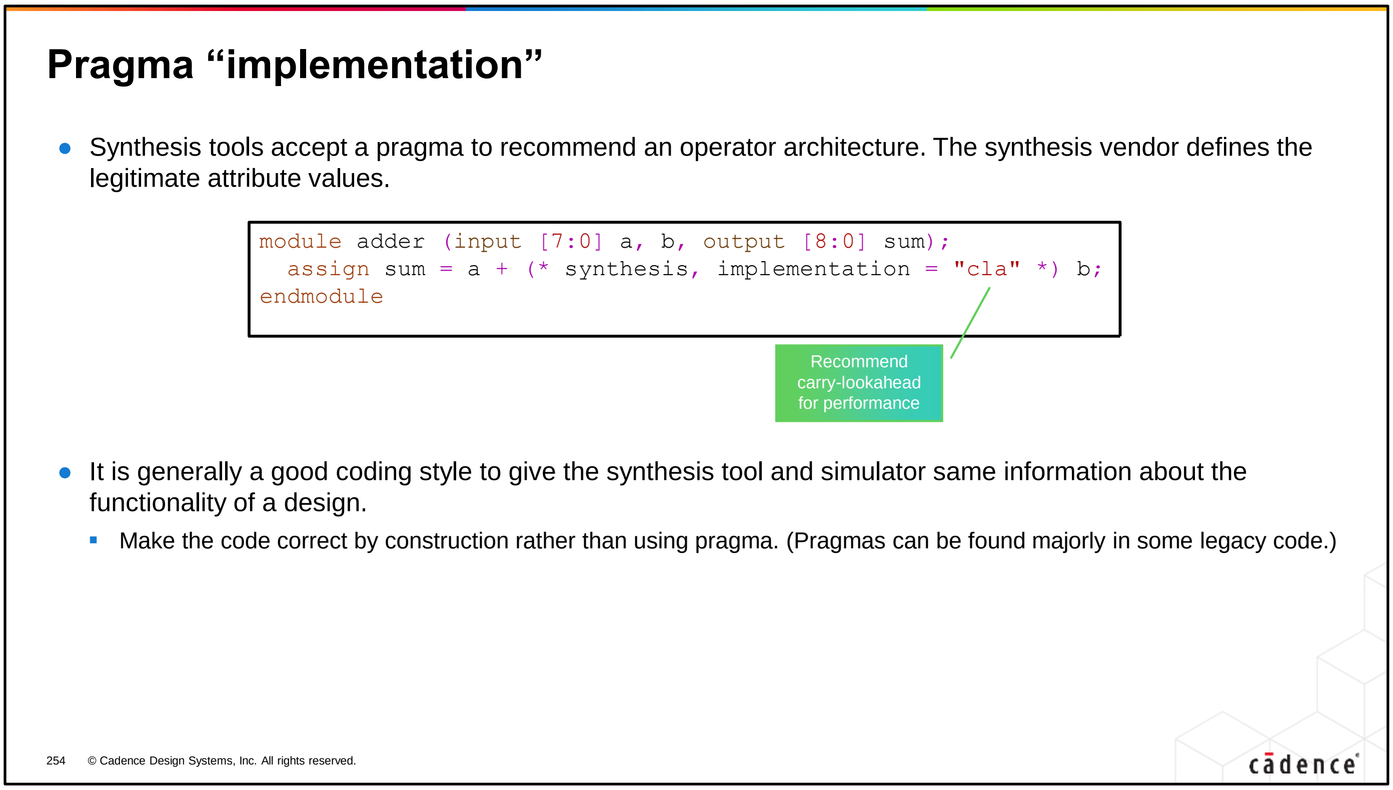
Synthesis tools accept a pragma to suppress the priority structure.

```
module parallel (
  output reg a, b,
  input [1:0] sel
);
  always @*
    begin
      a = 1'b0;
      b = 1'b0;
      (* synthesis, parallel_case *)
      casez (sel)
        2'b?1: a = 1'b1;
        2'b1?: b = 1'b1;
      endcase
    end
endmodule
```

Are you *absolutely* sure that “sel” will never be `2'b11` ?

Avoid usage of "full_case parallel_case" directives with any Verilog case statements, make it correct by construction rather than pragma.

The parallel_case attribute directs the synthesis tool to disregard the implied priority of the case items and test all case items in parallel. This is in effect a declaration that the case items are mutually exclusive in normal operation, and can produce a more optimal postsynthesis netlist. You must be very careful to not declare case items to be mutually exclusive if they are not.



Pragma “implementation”

- Synthesis tools accept a pragma to recommend an operator architecture. The synthesis vendor defines the legitimate attribute values.

```
module adder (input [7:0] a, b, output [8:0] sum);
  assign sum = a + (* synthesis, implementation = "cla" *) b;
endmodule
```

Recommend carry-lookahead for performance

- It is generally a good coding style to give the synthesis tool and simulator same information about the functionality of a design.
  - Make the code correct by construction rather than using pragma. (Pragmas can be found majorly in some legacy code.)

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

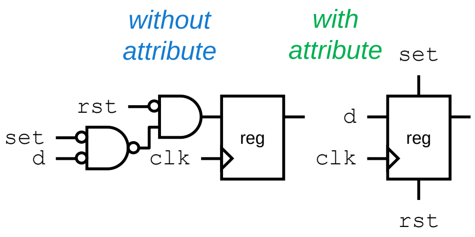
The implementation attribute recommends an architecture for operator implementation. The synthesis vendor defines the legitimate values. The synthesis tool can legally ignore the recommendation.

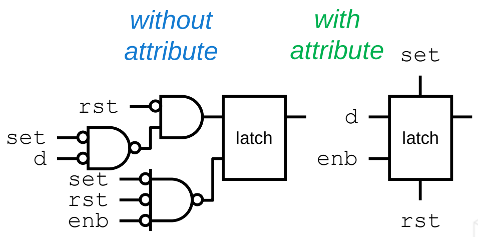
Pragma `[a]sync_set_reset`

Synthesis tools accept a pragma to cause direct connection of set / reset signals to set / reset terminals and optionally designate set / reset signals.

```
(* synthesis, sync_set_reset *)
always @(posedge clk)
  if (rst)
    q <= 0;
  else
    if (set)
      q <= 1;
    else
      q <= d;
```

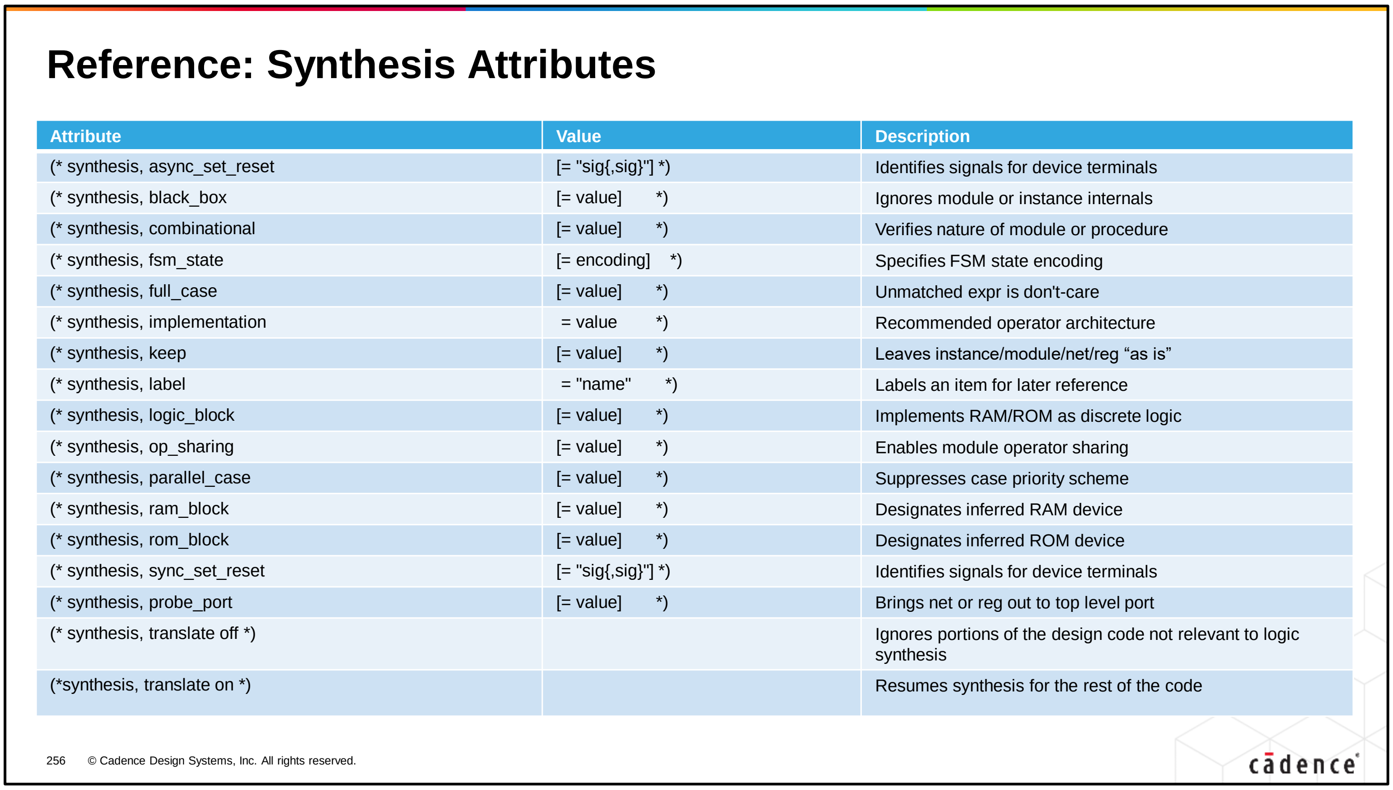
```
(* synthesis, async_set_reset *)
always @*
  if (rst)
    q <= 0;
  else
    if (set)
      q <= 1;
    else
      if (enb)
        q <= d;
```





The sync_set_reset attribute causes synthesis to directly connect synchronous set and reset signals to the device set and reset terminals, if a device with such terminals exists in the library. You can apply this attribute to a module or to a procedure. If you also designate the set or reset signals, then synthesis will not directly connect any other set or reset signals to the device set and reset terminals, but will instead route them into the data logic.

The async_set_reset attribute has exactly the same effect upon latch logic.



Reference: Synthesis Attributes

Attribute	Value	Description
(* synthesis, async_set_reset	[= "sig{,sig}"] *)	Identifies signals for device terminals
(* synthesis, black_box	[= value] *)	Ignores module or instance internals
(* synthesis, combinational	[= value] *)	Verifies nature of module or procedure
(* synthesis, fsm_state	[= encoding] *)	Specifies FSM state encoding
(* synthesis, full_case	[= value] *)	Unmatched expr is don't-care
(* synthesis, implementation	= value *)	Recommended operator architecture
(* synthesis, keep	[= value] *)	Leaves instance/module/net/reg “as is”
(* synthesis, label	= "name" *)	Labels an item for later reference
(* synthesis, logic_block	[= value] *)	Implements RAM/ROM as discrete logic
(* synthesis, op_sharing	[= value] *)	Enables module operator sharing
(* synthesis, parallel_case	[= value] *)	Suppresses case priority scheme
(* synthesis, ram_block	[= value] *)	Designates inferred RAM device
(* synthesis, rom_block	[= value] *)	Designates inferred ROM device
(* synthesis, sync_set_reset	[= "sig{,sig}"] *)	Identifies signals for device terminals
(* synthesis, probe_port	[= value] *)	Brings net or reg out to top level port
(* synthesis, translate off *)		Ignores portions of the design code not relevant to logic synthesis
(*synthesis, translate on *)		Resumes synthesis for the rest of the code

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

The async_set_reset attribute specifies that asynchronous set and reset signals of the module or procedure to which it applies shall directly connect to the asynchronous set and reset terminals of the library device if the library provides a device with such terminals. If the attribute specifies signal names, then any other set and reset signals shall not directly connect to asynchronous set and reset terminals of the device.

The black_box attribute specifies that synthesis shall ignore the internal contents of the instance or module to which it applies.

The combinational attribute specifies that synthesis shall report as an error any storage inferred by the module or procedure to which it applies.

The fsm_state attribute specifies a state encoding scheme for the register to which it applies.

The full_case attribute specifies that you don’t care what the outputs are for unspecified case choices.

The implementation attribute recommends an operator architecture.

The keep attribute specifies that synthesis shall preserve an instance, module, net or register “as is”.

The label attribute labels an item for future reference.

The logic_block attribute supports implementation of a RAM or ROM block in discrete gates.

The op_sharing attribute can disable operator sharing within a module. Operator sharing may occur anyway without the attribute. With the attribute enabled, operator sharing may still not occur.

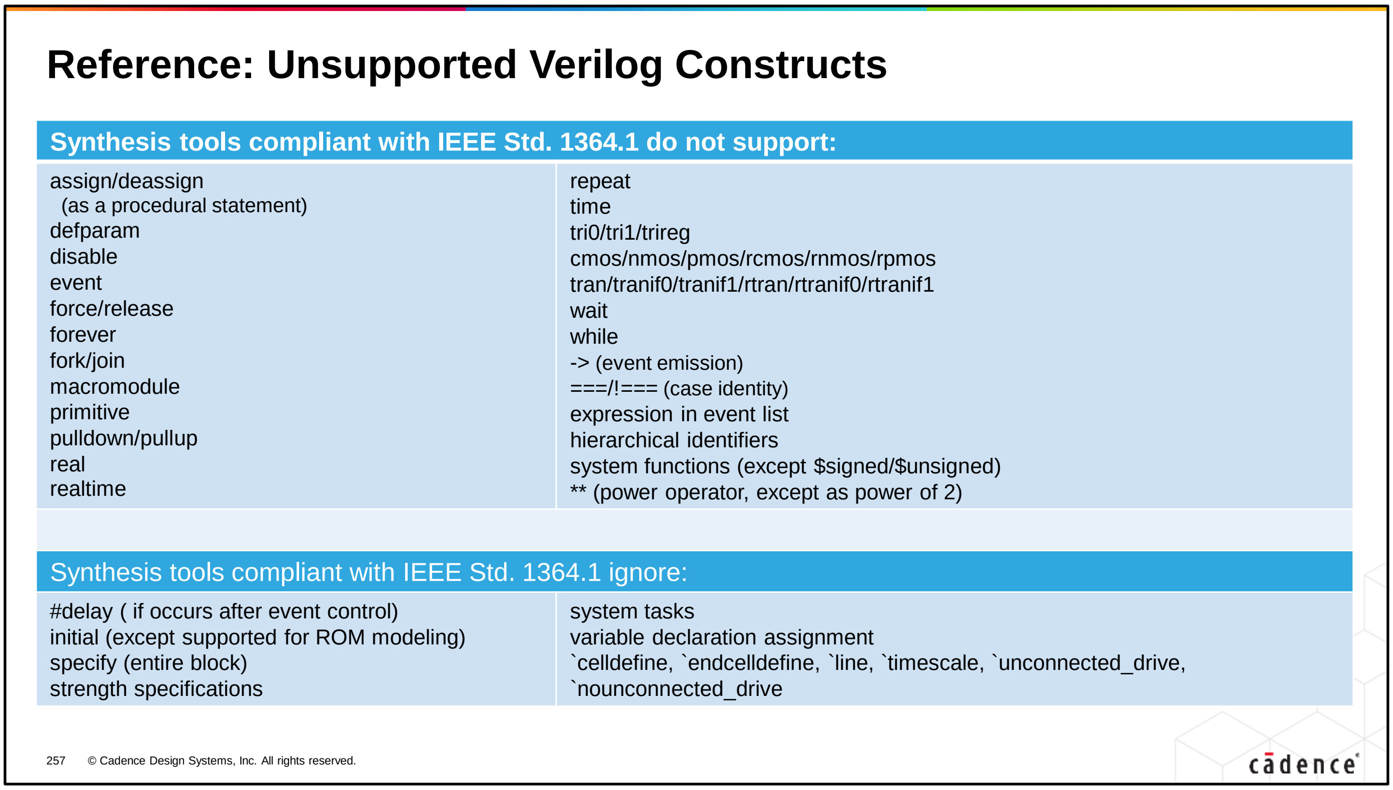
The parallel_case attribute specifies that synthesis should suppress the normal case statement priority architecture.

The ram_block attribute supports selection of an inferred RAM device.

The rom_block attribute supports selection of an inferred ROM device.

The sync_set_reset attribute specifies that synchronous set and reset signals of the module or procedure to which it applies shall directly connect to the synchronous set and reset terminals of the library device if the library provides a device with such terminals. If the attribute specifies signal names, then any other set and reset signals shall not directly connect to synchronous set and reset terminals of the device.

The probe_port attribute causes synthesis to provide a top-level output port connected to the net or register.



Reference: Unsupported Verilog Constructs

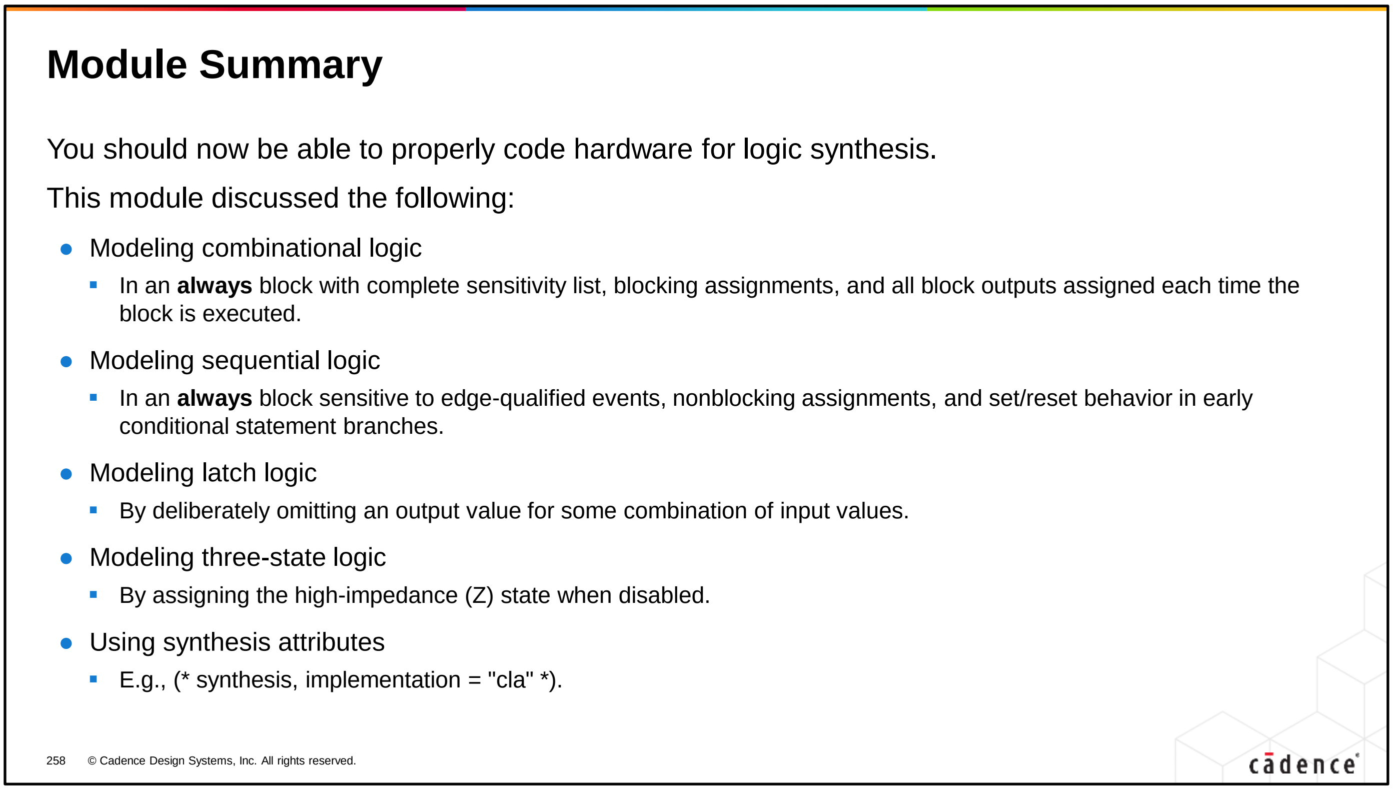
Synthesis tools compliant with IEEE Std. 1364.1 do not support:	
assign/deassign	repeat
(as a procedural statement)	time
defparam	tri0/tri1/trireg
disable	cmos/nmos/pmos/rcmos/rnmos/rpmos
event	tran/tranif0/tranif1/rtran/rtranif0/rtranif1
force/release	wait
forever	while
fork/join	-> (event emission)
macromodule	===/!=== (case identity)
primitive	expression in event list
pulldown/pullup	hierarchical identifiers
real	system functions (except $signed/$unsigned)
realtime	** (power operator, except as power of 2)

Synthesis tools compliant with IEEE Std. 1364.1 ignore:	
#delay (if occurs after event control)	system tasks
initial (except supported for ROM modeling)	variable declaration assignment
specify (entire block)	`celldefine, `endcelldefine, `line, `timescale, `unconnected_drive,
strength specifications	`nounconnected_drive

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

Here for your reference, are constructs that the synthesis tool does not support, and constructs that the synthesis tool ignores.

Synthesis supports the always keyword only with an immediately following event control (@).



Module Summary

You should now be able to properly code hardware for logic synthesis.

This module discussed the following:

- Modeling combinational logic
  - In an **always** block with complete sensitivity list, blocking assignments, and all block outputs assigned each time the block is executed.
- Modeling sequential logic
  - In an **always** block sensitive to edge-qualified events, nonblocking assignments, and set/reset behavior in early conditional statement branches.
- Modeling latch logic
  - By deliberately omitting an output value for some combination of input values.
- Modeling three-state logic
  - By assigning the high-impedance (Z) state when disabled.
- Using synthesis attributes
  - E.g., (* synthesis, implementation = "cla" *).

This module explained how to describe hardware for logic synthesis. It addressed combinational, sequential, latch, and three-state logic. It introduced synthesis attributes that you embed in the source code to influence the synthesis process.

Module Review

1. For what conditions does logic synthesis infer a latch in logic you intend to be purely combinational?
2. Explain why a synthesis tool may or may not infer a register for a blocking assignment to a variable.
3. For what Verilog construct does synthesis infer a three-state gate?
4. In what most significant way does synthesis restrict a **for** loop?

This page does not contain notes.