Using Blocking and Nonblocking Assignments

Module 7

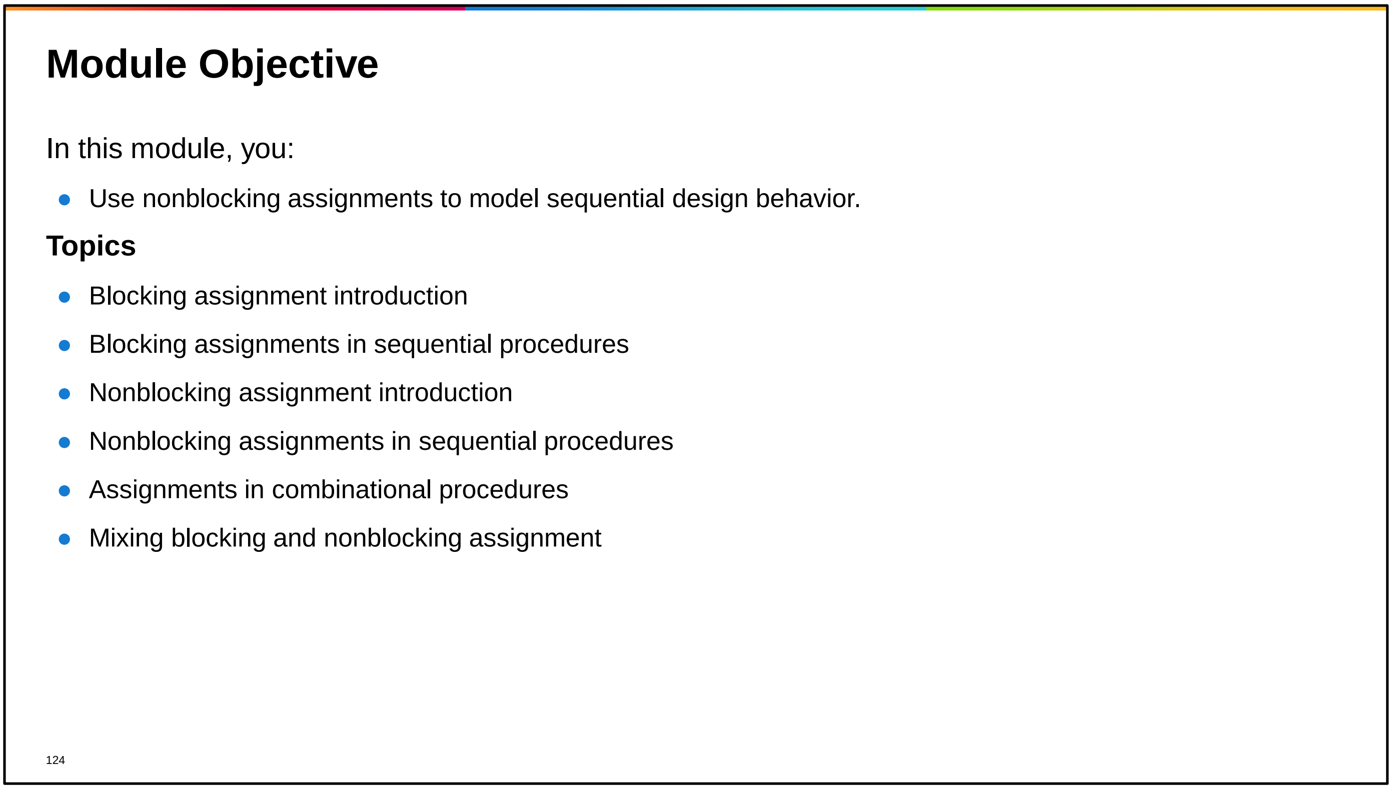
Revision 1.0

Version 1.0

Estimated Time:

- **Lecture**
- **Lab**

This module reviews blocking and nonblocking assignments, and then examines how to use them in procedural blocks meant to represent combinational logic or sequential logic, and lastly explores issues with statement order and mixing blocking and nonblocking assignments in the same procedural block.



Module Objective

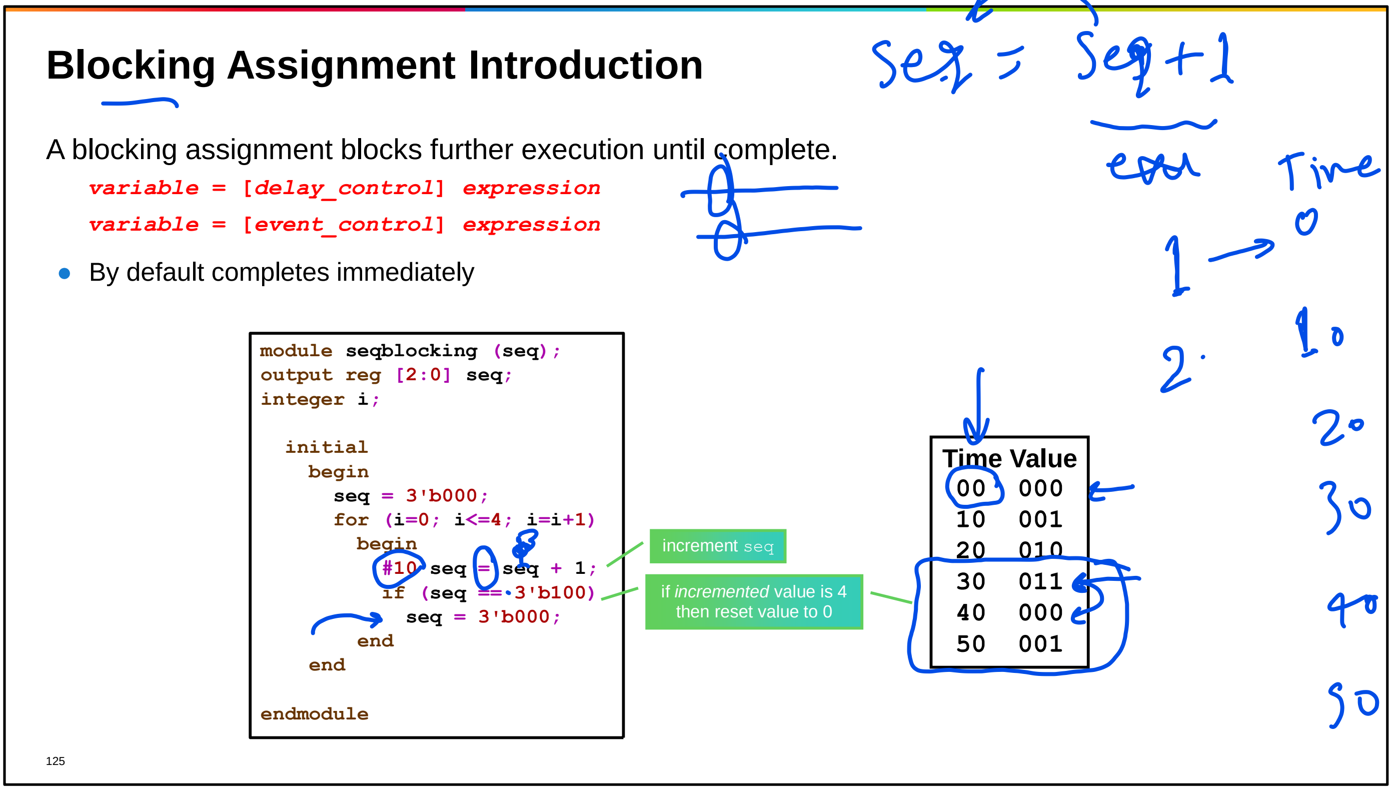
In this module, you:

- Use nonblocking assignments to model sequential design behavior.

Topics

- Blocking assignment introduction
- Blocking assignments in sequential procedures
- Nonblocking assignment introduction
- Nonblocking assignments in sequential procedures
- Assignments in combinational procedures
- Mixing blocking and nonblocking assignment

Your objective is to correctly choose between blocking and nonblocking assignments. To do that, you need to know more details about using blocking and nonblocking assignments in combinational and sequential procedures.



Blocking Assignment Introduction

A blocking assignment blocks further execution until complete.

```
variable = [delay_control] expression
variable = [event_control] expression
```

- By default completes immediately

```
module seqblocking (seq);
output reg [2:0] seq;
integer i;

  initial
    begin
      seq = 3'b000;
      for (i=0; i<=4; i=i+1)
        begin
          #10 seq = seq + 1;
          if (seq == 3'b100)
            seq = 3'b000;
        end
    end

endmodule
```

increment seq

if *incremented* value is 4 then reset value to 0

Time	Value
00	000
10	001
20	010
30	011
40	000
50	001

ass

seq = seq + 1

eval

Time

1 → 0

2.

10

20

30

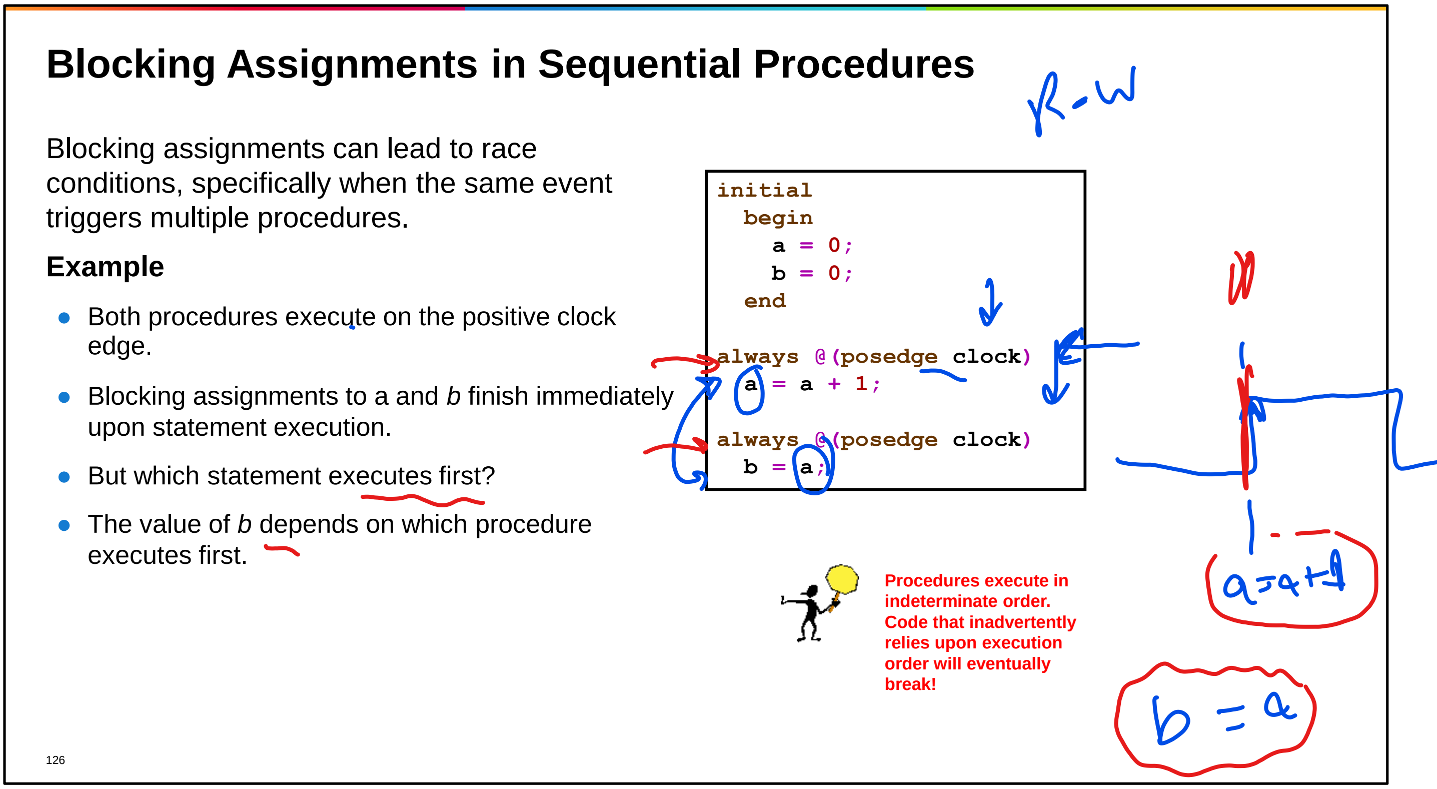
40

50

125

The simulator executes a blocking assignment by evaluating the right-side expression, retaining its value, and blocking further execution of the block until it updates the left-side variable. If the assignment includes an intra-assignment timing control, then the simulator updates the variable after the timing control expires, and then continues block execution. If the assignment does not include an intra-assignment timing control, then the simulator updates the variable immediately.

In this example, the for loop increments the seq signal every 10 time units. The update occurs before the next statement executes. The next statement tests the new value, and if the new value is 4, immediately resets it to 0. The value 4 has no duration, so does not appear in the sequence of monitored values.



Blocking Assignments in Sequential Procedures

R-W

Blocking assignments can lead to race conditions, specifically when the same event triggers multiple procedures.

Example

- Both procedures execute on the positive clock edge.
- Blocking assignments to a and *b* finish immediately upon statement execution.
- But which statement executes first?
- The value of *b* depends on which procedure executes first.

```
initial
  begin
    a = 0;
    b = 0;
  end

always @(posedge clock)
  a = a + 1;

always @(posedge clock)
  b = a;
```

Procedures execute in indeterminate order. Code that inadvertently relies upon execution order will eventually break!

a=a+1

b = a

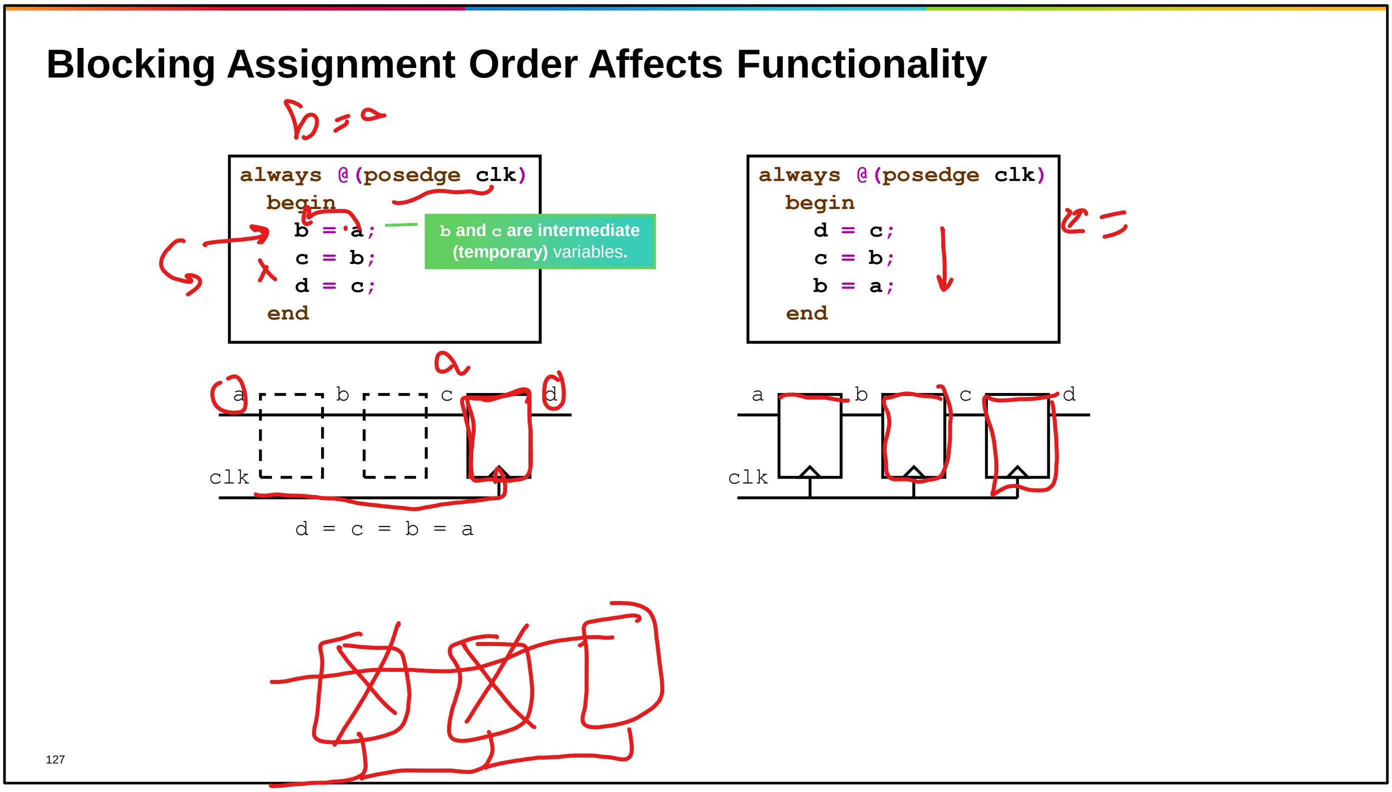
126

Blocking assignments can lead to race conditions specifically when the same event triggers multiple procedures, that then execute in a nondeterministic order, such that a procedure can read a variable that another procedure may or may not have already written.

In this example, both procedural blocks unblock on the positive clock edge. The simulator can resume block execution in any order. The assignment statements use the blocking operator, so a race exists between the assignments to a and to b.

- If the simulator first executes the upper block, it updates a to get the incremented value and then executes the lower block where it updates b to get the new a value.
- If the simulator first executes the lower block, it updates b to get the not yet changed a value and then executes the upper block where it updates a to get the incremented value.

The final b value does depend upon procedure execution order.



Blocking Assignment Order Affects Functionality

b = a

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

b and c are intermediate (temporary) variables.

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

c =

d = c = b = a

127

The left illustration uses blocking assignments:

- It assigns a to b and immediately updates the b value;
- Then it assigns b to c and immediately updates the c value; and
- Finally, it assigns c to d and immediately updates the d value.

As each variable is written before it is read, all the variables have the value of a. The b and c variables will not exist in a hardware implementation, as the d variable provides the same value.

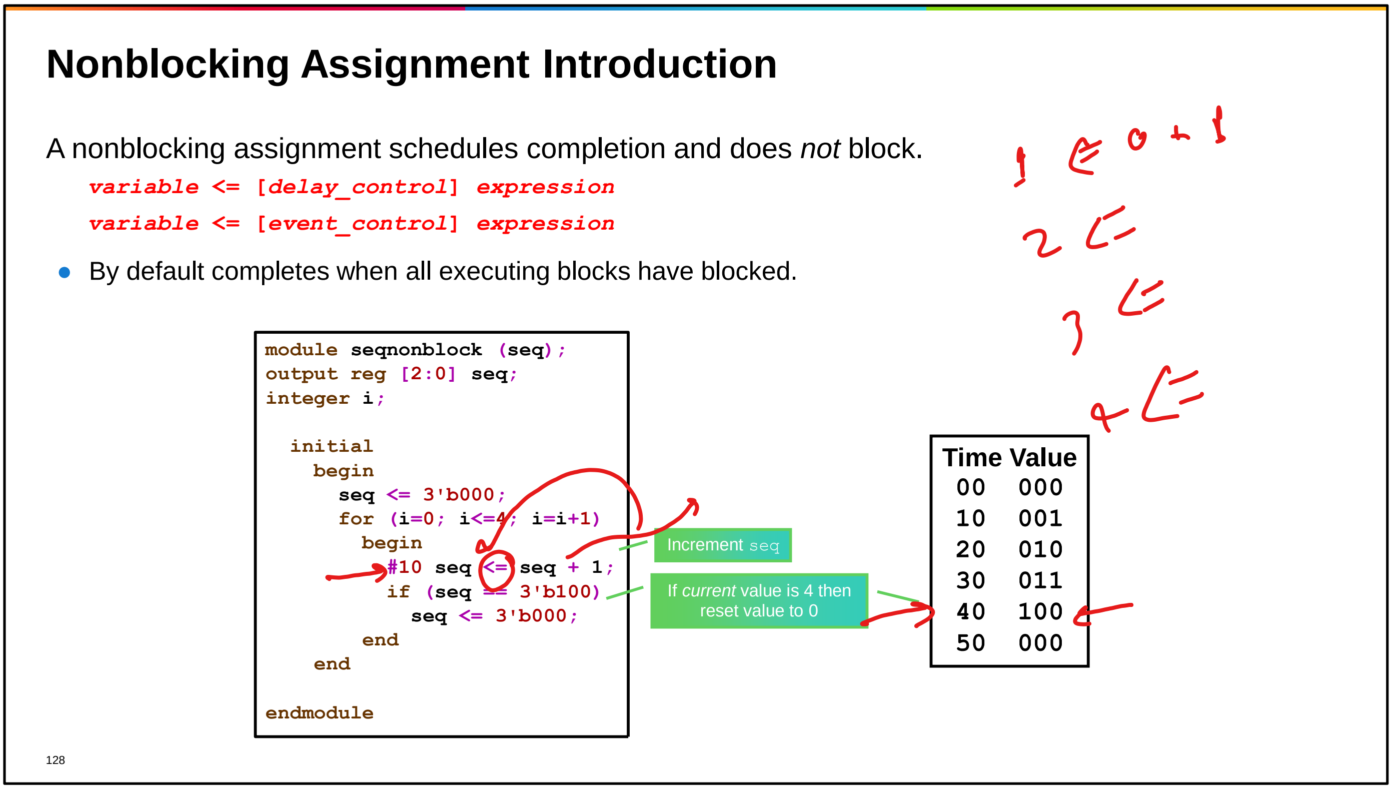
The right illustration also uses blocking assignments, but it reorders the assignments:

- It assigns c to d and immediately updates the d value;
- Then it assigns b to c and immediately updates the c value; and
- Finally, it assigns a to b and immediately updates the b value.

As the variables are read before they are written, all can all have different values. All the variables will exist in a hardware implementation.

This is an example of position-dependent code. The statement order affects the functionality. You need to be aware of this issue, but it is not “bad” programming as such, because it is actually quite useful when deliberately done.

For this illustration, statement order would not affect functionality if the assignments were nonblocking.



Nonblocking Assignment Introduction

A nonblocking assignment schedules completion and does *not* block.

```
variable <= [delay_control] expression
variable <= [event_control] expression
```

- By default completes when all executing blocks have blocked.

```
module seqnonblock (seq);
output reg [2:0] seq;
integer i;

  initial
    begin
      seq <= 3'b000;
      for (i=0; i<=4; i=i+1)
        begin
          #10 seq <= seq + 1;
          if (seq == 3'b100)
            seq <= 3'b000;
        end
    end

endmodule
```

Increment seq

If *current* value is 4 then reset value to 0

Time	Value
00	000
10	001
20	010
30	011
40	100
50	000

1 ⇐ 0 + 1

2 <=

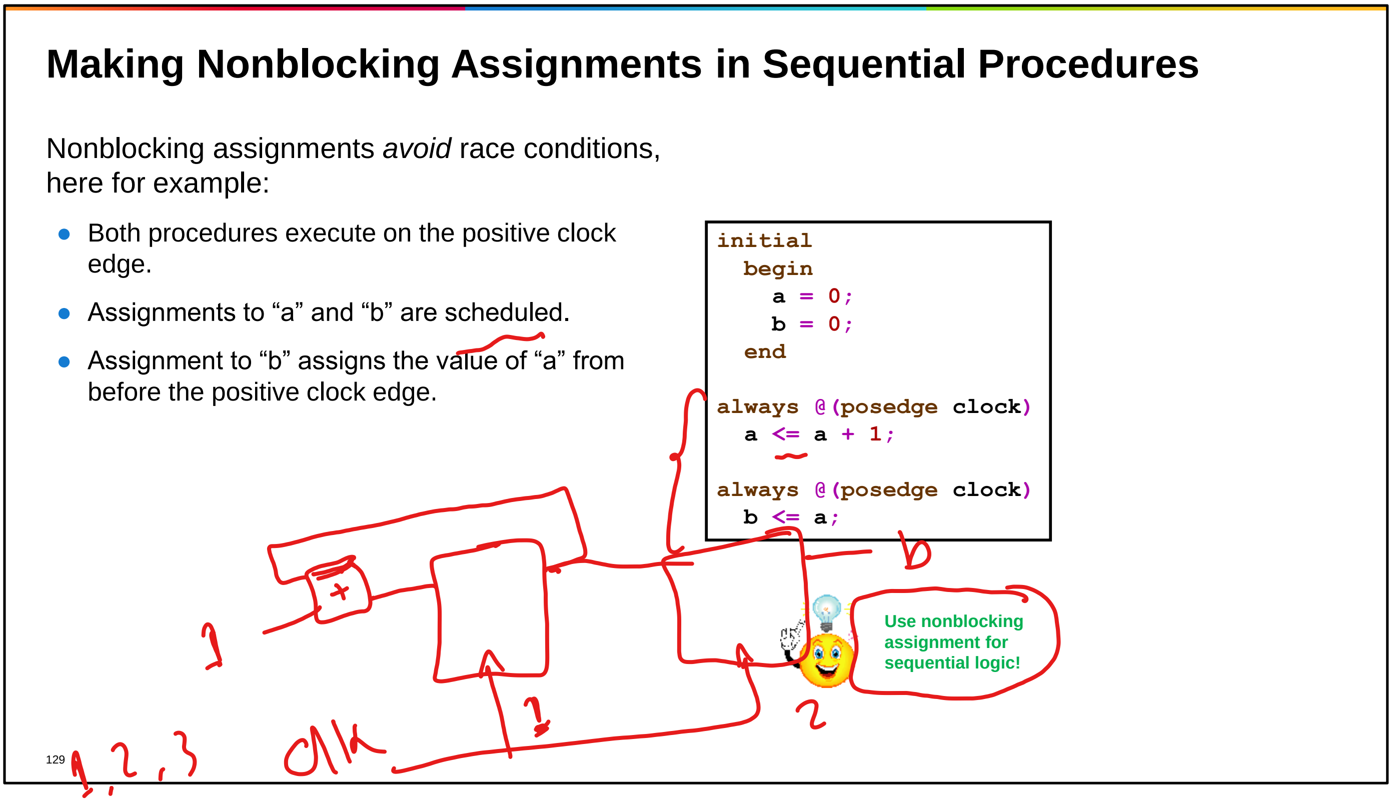
3 <=

4 <=

128

The simulator executes a nonblocking assignment by evaluating the right-side expression, retaining its value, scheduling the update, and continuing to execute the block until it encounters a blocking construct. The simulator schedules the variable update for a point in the simulation where all currently active blocks have executed up to the point where they are all blocked. This ensures that any other active block that reads the variable reads the old value and not the new value.

In this example, the *for* loop increments the *seq* signal every 10 time units. The update does not immediately occur. The next statement tests the old value, and if the old value is 4, resets it to 0. The value 4 has a duration of 10 time units, so *does* appear in the sequence of monitored values.



Making Nonblocking Assignments in Sequential Procedures

Nonblocking assignments *avoid* race conditions, here for example:

- Both procedures execute on the positive clock edge.
- Assignments to “a” and “b” are scheduled.
- Assignment to “b” assigns the value of “a” from before the positive clock edge.

```
initial
  begin
    a = 0;
    b = 0;
  end

always @(posedge clock)
  a <= a + 1;

always @(posedge clock)
  b <= a;
```

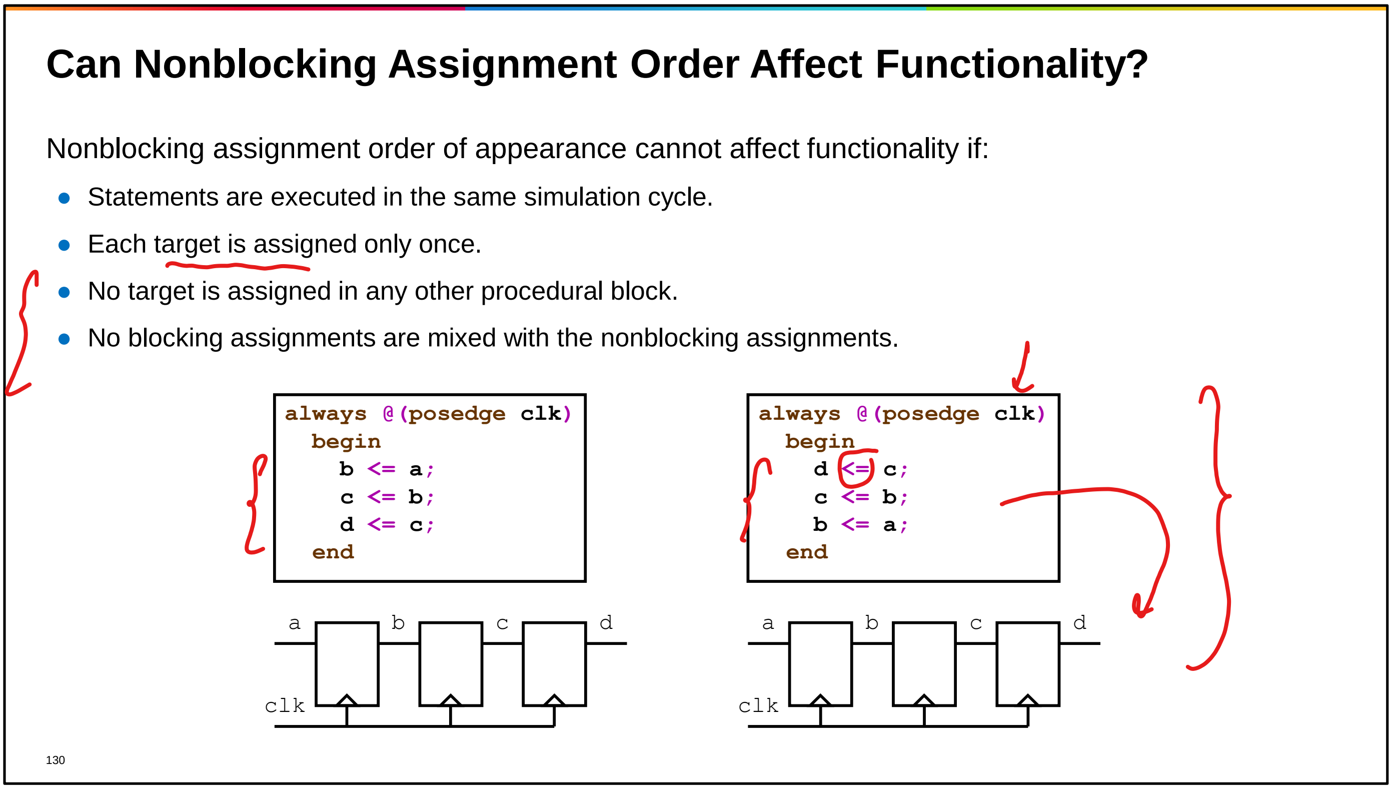
Use nonblocking assignment for sequential logic!

129

In this example, both procedural blocks unblock on the positive clock edge. The simulator can resume block execution in any order. The assignment statements use the nonblocking operator, so no race condition exists.

- If the simulator first executes the upper block, it schedules an a variable update to get the incremented value and the a variable value does not yet change. The simulator then executes the lower block where it schedules a b variable update to get the old unchanged a variable value.
- If the simulator first executes the lower block, it schedules a b variable update to get the not yet changed a variable value. The simulator then executes the upper block, where it schedules an a variable update to get the incremented value.
- After executing these and all other triggered blocks to the point where they block, the simulator completes the nonblocking assignments to update the variable values.

The final b variable value does NOT depend upon procedure execution order.



Can Nonblocking Assignment Order Affect Functionality?

Nonblocking assignment order of appearance cannot affect functionality if:

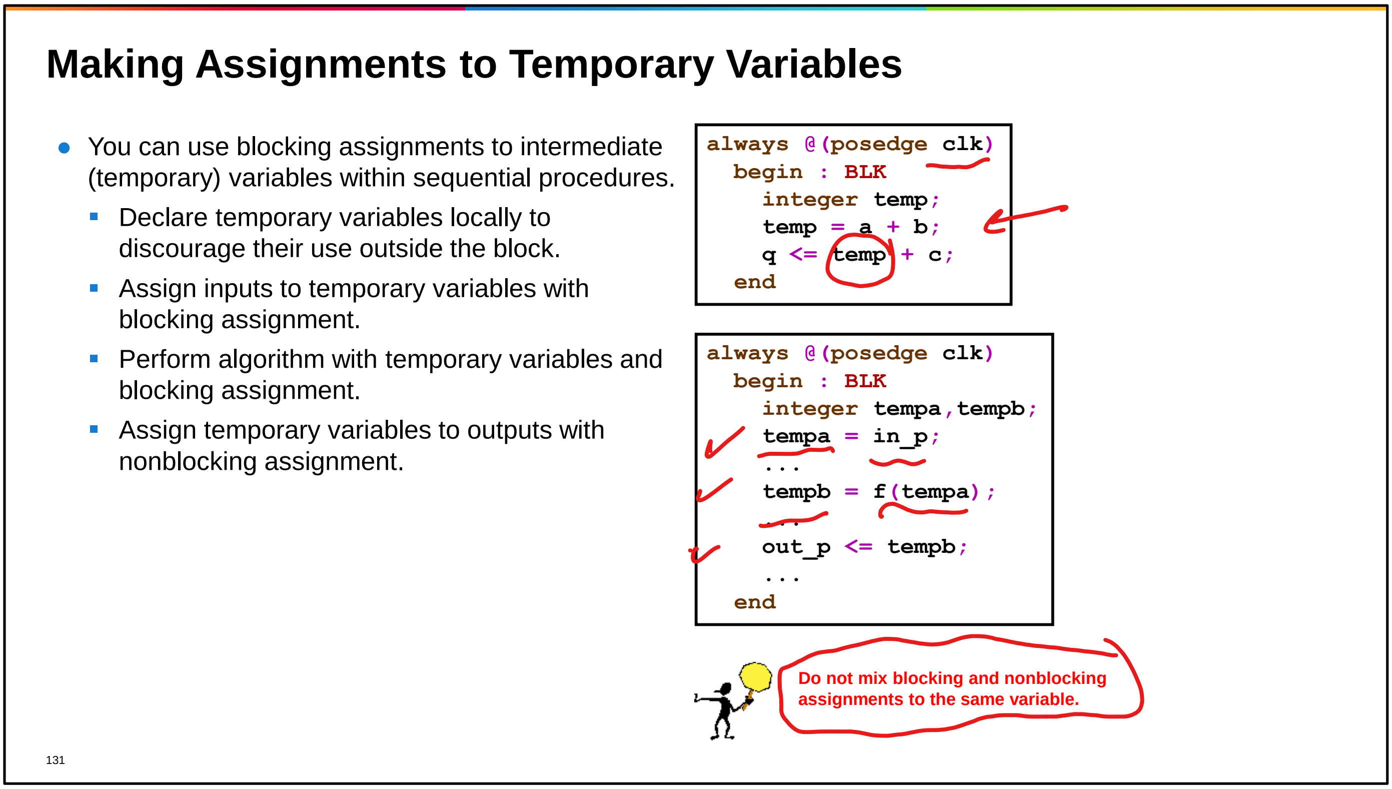
- Statements are executed in the same simulation cycle.
- Each target is assigned only once.
- No target is assigned in any other procedural block.
- No blocking assignments are mixed with the nonblocking assignments.

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

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

130

Here is the same illustration as previously, but utilizing nonblocking assignments. If you simply adhere to good programming practices, then the order of assignment appearance cannot affect functionality.



Making Assignments to Temporary Variables

- You can use blocking assignments to intermediate (temporary) variables within sequential procedures.
  - Declare temporary variables locally to discourage their use outside the block.
  - Assign inputs to temporary variables with blocking assignment.
  - Perform algorithm with temporary variables and blocking assignment.
  - Assign temporary variables to outputs with nonblocking assignment.

```
always @(posedge clk)
  begin : BLK
    integer temp;
    temp = a + b;
    q <= temp + c;
  end
```

```
always @(posedge clk)
  begin : BLK
    integer tempa,tempb;
    tempa = in_p;
    ...
    tempb = f(tempa);
    ...
    out_p <= tempb;
    ...
  end
```

Do not mix blocking and nonblocking assignments to the same variable.

You have previously seen that position dependent code is not “bad” programming as such, because it is actually quite useful when deliberately done.

You can use blocking assignments to strictly temporary variables within sequential procedural blocks meant to represent sequential logic.

The usage model is to use blocking assignments, which complete immediately, to break a complex expression representing combinational logic into a series of simpler assignments to temporary variables. Your final assignment uses the nonblocking operator to assign the temporary result to the variable that represents the sequential hardware.

For the temporary variables to be truly temporary, they cannot be used anywhere other than that one block. This is impossible to enforce in the Verilog language, but you can make them less likely to be used elsewhere by declaring them locally. Recall that to declare local variables you need to name the block.

Multiple Assignments and Assignment Type

You can make multiple assignments to a variable within one procedure:

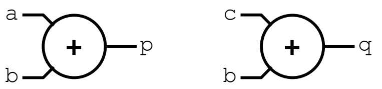
- All assignments to a variable should be the same type.
- Subsequent assignments override previous assignments.
- Watch out for unintended results!

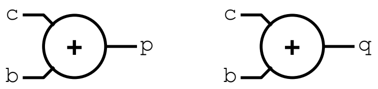
```
always @(a, b, c)
  begin
    m = a;
    n = b;
    p = m + n;
    m = c;
    q = m + n;
  end
```

m and n are temporary variables.

```
always @(a, b, c, m, n)
  begin
    m <= a;
    n <= b;
    p <= m + n;
    m <= c;
    q <= m + n;
  end
```

m and n are *not* temporary variables.



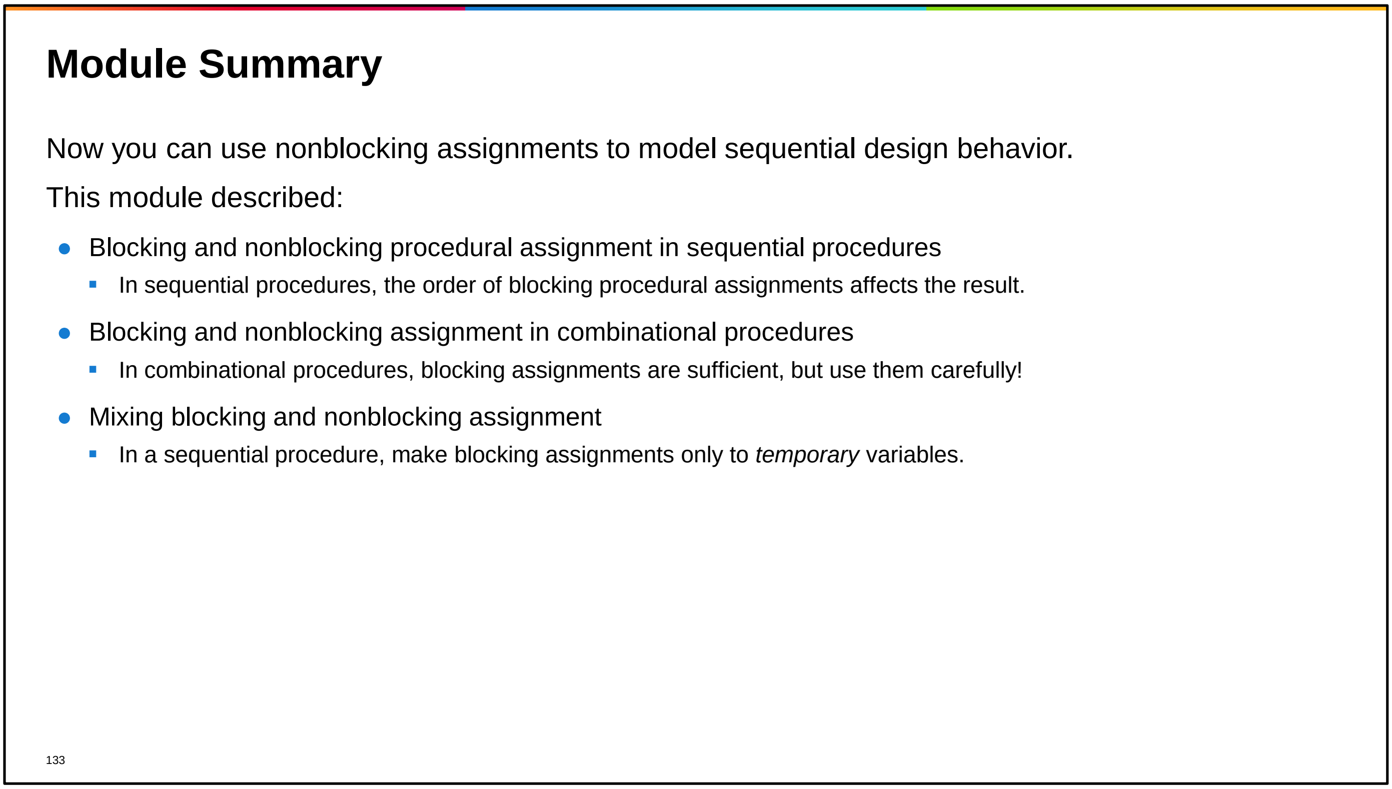


You have previously seen that statement order affects functionality.

Assignment type can also affect functionality.

For blocking assignments, the simulator immediately updates the variable before continuing block execution. New values are immediately available until overwritten. In the blocking illustration, the simulator immediately updates the assignment of *a* to *m* and calculates a new value for *p* using the *a* value of *m*. It then immediately updates the assignment of *c* to *m* and calculates a new value for *q* using the *c* value of *m*.

For nonblocking assignments, the simulator schedules the variable update. New values are available only after the update occurs. In the nonblocking illustration, the simulator schedules the assignment of *a* to *m* and immediately replaces it with a scheduled assignment of *c* to *m*. Upon updating the *m* variable it gets the value of *c* and not that of *a*. The simulator re-executes the procedural block due to the transition of *m* and *n* and calculates new values for *p* and *q* that use the *c* value of *m*.



Module Summary

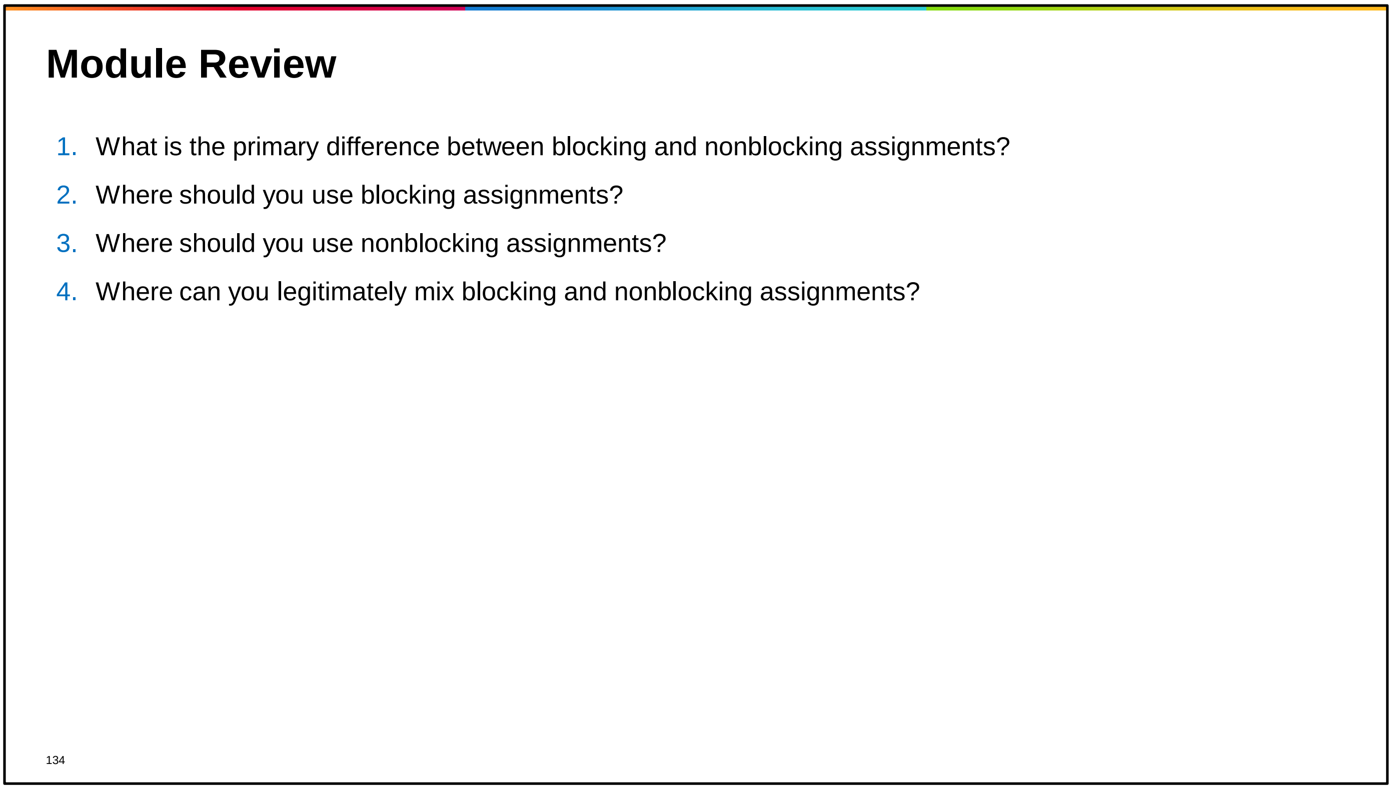
Now you can use nonblocking assignments to model sequential design behavior.

This module described:

- Blocking and nonblocking procedural assignment in sequential procedures
  - In sequential procedures, the order of blocking procedural assignments affects the result.
- Blocking and nonblocking assignment in combinational procedures
  - In combinational procedures, blocking assignments are sufficient, but use them carefully!
- Mixing blocking and nonblocking assignment
  - In a sequential procedure, make blocking assignments only to *temporary* variables.

133

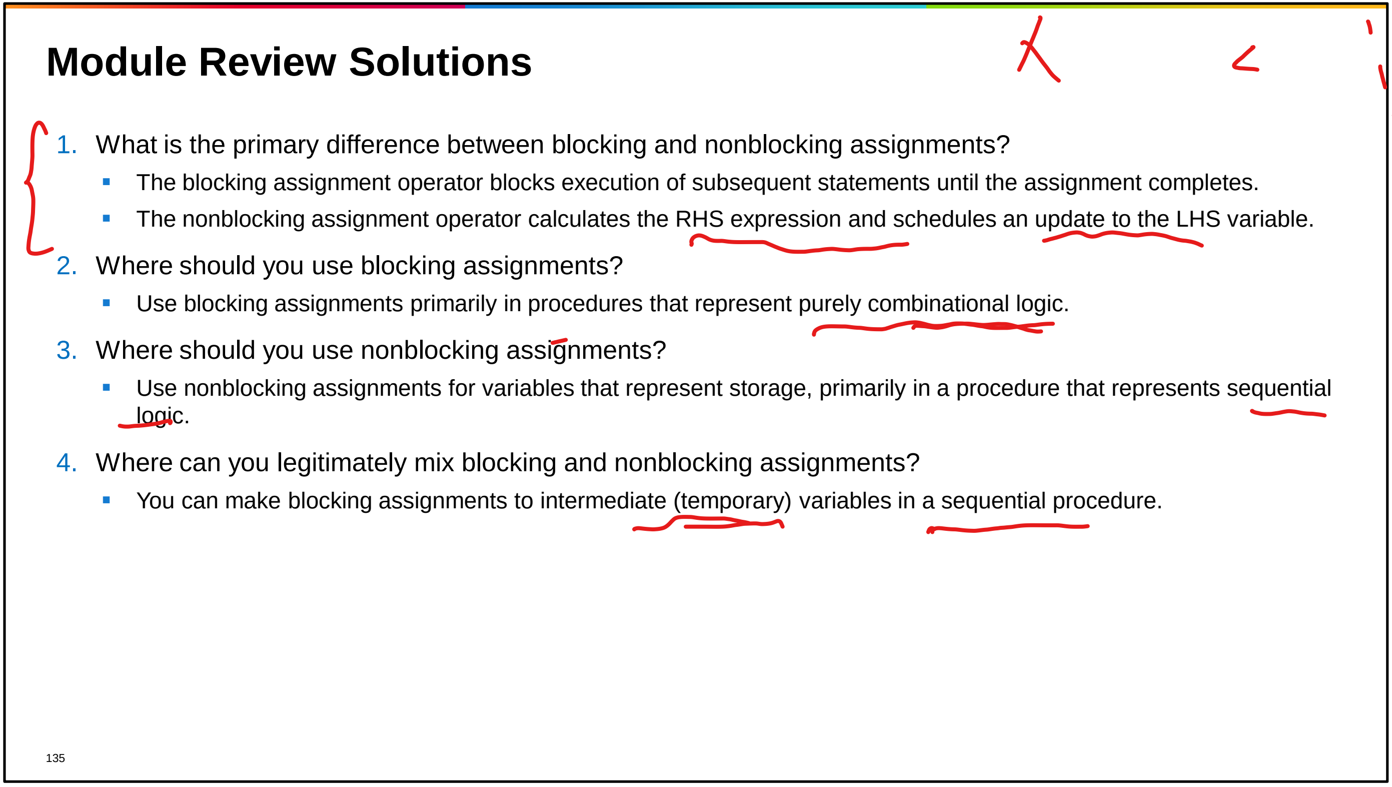
You can now correctly choose between blocking and nonblocking assignments. This module reviewed blocking and nonblocking assignments, and then examined how to use them in procedural blocks meant to represent combinational logic or sequential logic, and lastly explored issues with statement order and mixing blocking and nonblocking assignments in the same procedural block.



Module Review

1. What is the primary difference between blocking and nonblocking assignments?
2. Where should you use blocking assignments?
3. Where should you use nonblocking assignments?
4. Where can you legitimately mix blocking and nonblocking assignments?

This page does not contain notes.



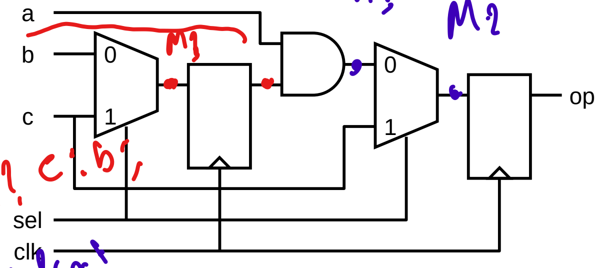
Module Review Solutions

1. What is the primary difference between blocking and nonblocking assignments?
   - The blocking assignment operator blocks execution of subsequent statements until the assignment completes.
   - The nonblocking assignment operator calculates the RHS expression and schedules an update to the LHS variable.
2. Where should you use blocking assignments?
   - Use blocking assignments primarily in procedures that represent purely combinational logic.
3. Where should you use nonblocking assignments?
   - Use nonblocking assignments for variables that represent storage, primarily in a procedure that represents sequential logic.
4. Where can you legitimately mix blocking and nonblocking assignments?
   - You can make blocking assignments to intermediate (temporary) variables in a sequential procedure.

This page does not contain notes.

Module Exercise

Code this circuit in two procedures in such a manner that execution order does not affect the result.



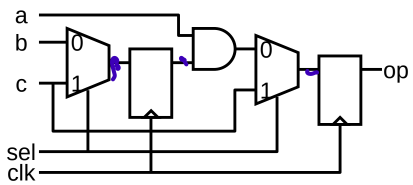
```
reg q1 ;
wire M1
assign M1 = (sel)? c : b ;
assign A1 = q1 & a ;
always @ posedge clk
  begin
    q1 <= M1
  end
```

```
assign M2 = (sel) ? c : A1
always @ posedge clk
  begin
    op <= M2 ;
  end
```

This page does not contain notes.

Module Exercise Solution

Code this circuit in two procedures in such a manner that execution order does not affect the result.



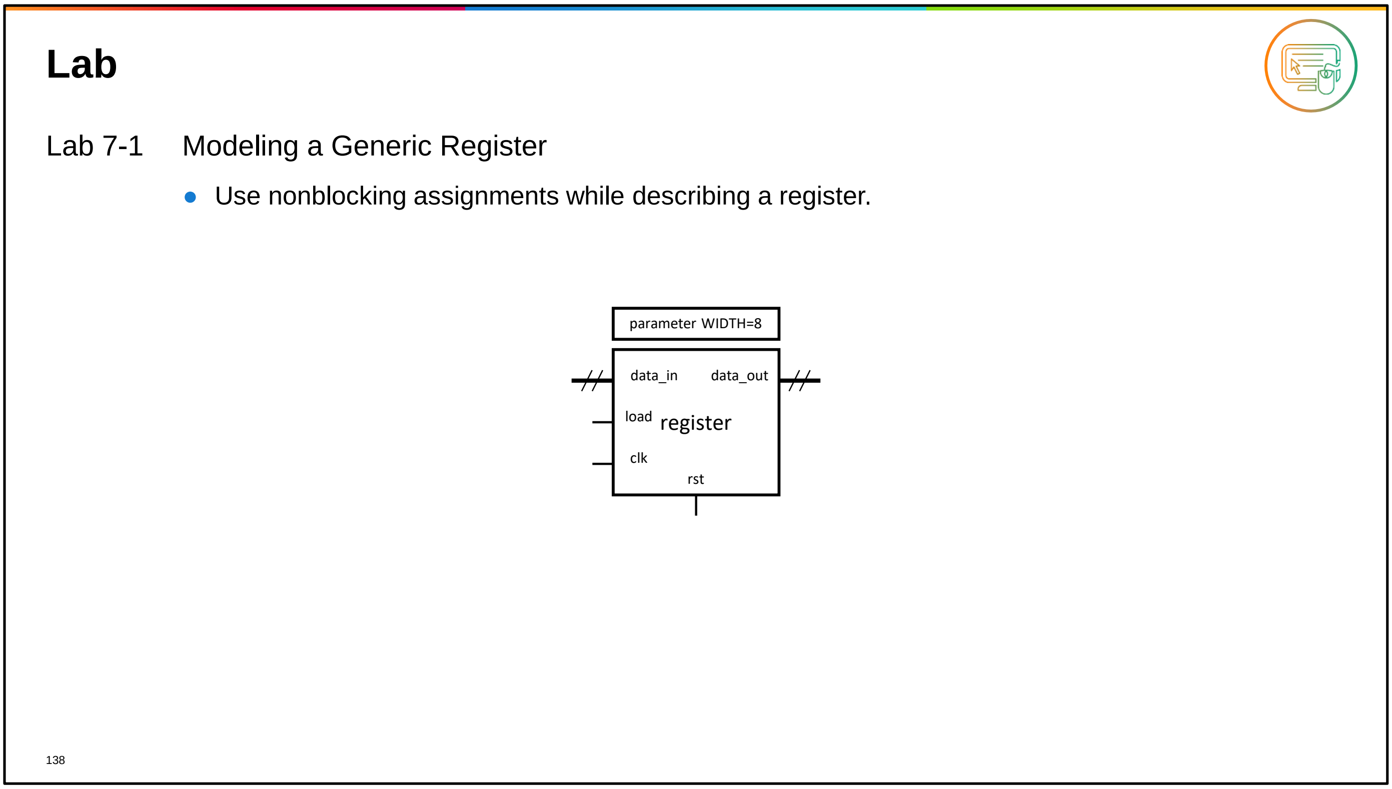
Solution:

as

```
always @(posedge clk)
  rg <= sel ? c : b ;

always @(posedge clk)
  op <= sel ? c : a & rg ;
```

This page does not contain notes.



Lab

Lab 7-1 Modeling a Generic Register

- Use nonblocking assignments while describing a register.

Your objective is to use nonblocking assignments to model sequential design behavior.

For this lab, you use nonblocking assignments while describing a register.