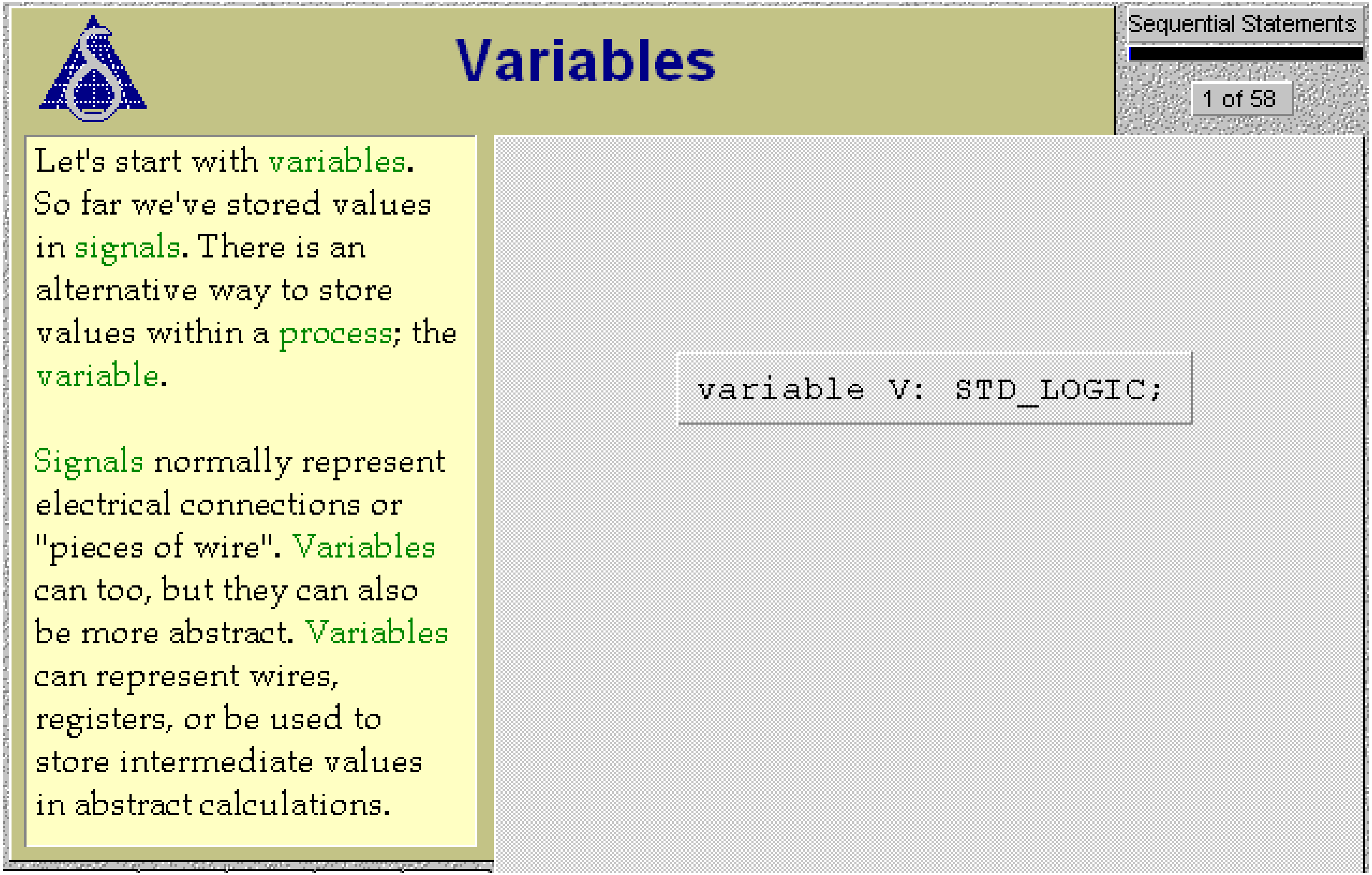
DOULOS

Lesson 5

Sequential Statements

The previous lesson looked at the structure of the process. You will now learn about the statements written inside a process. The focus will be on the behaviour and synthesis of these statements. Sequential statements have many uses, from describing hardware for synthesis to writing test vectors.

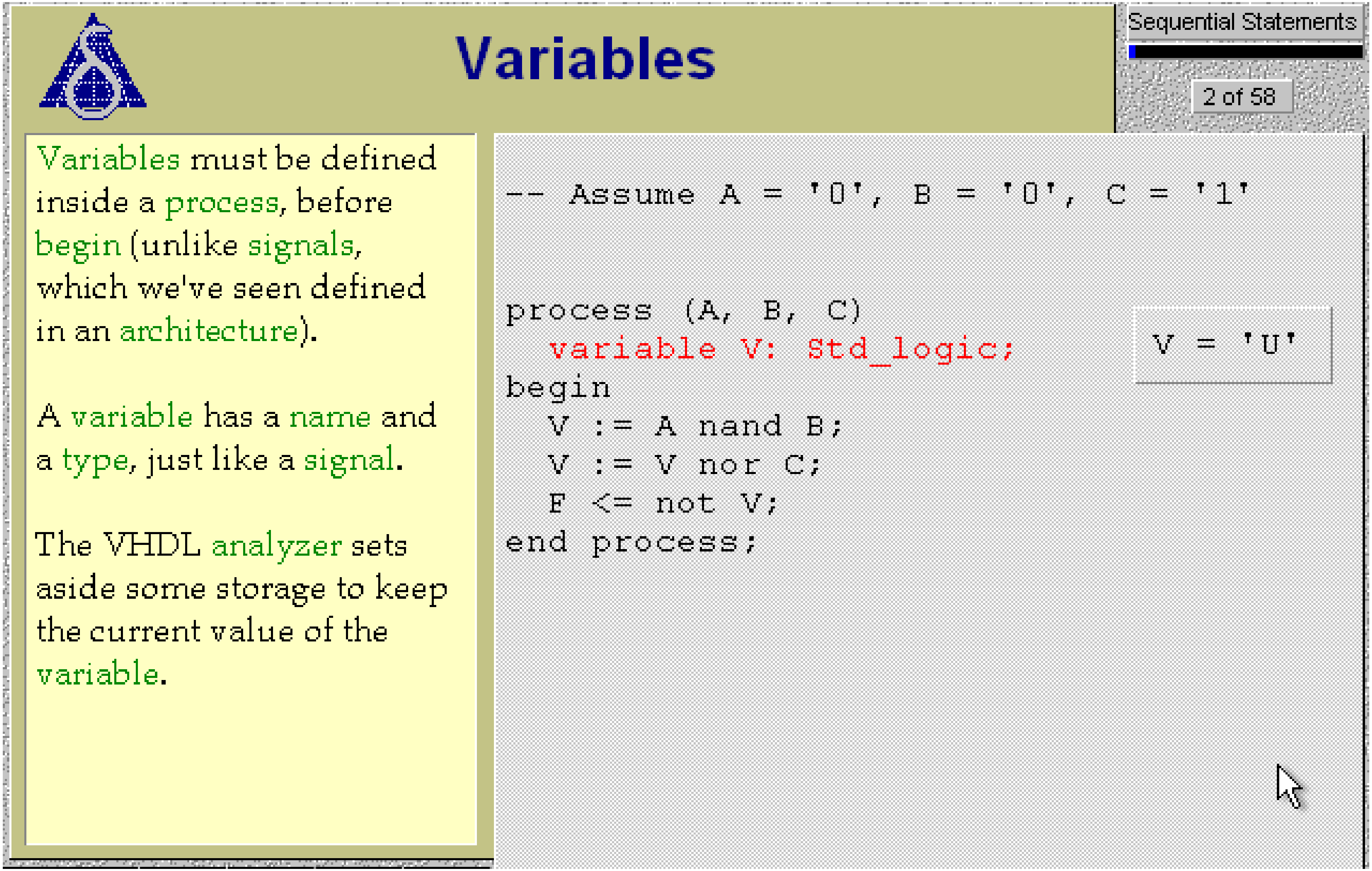


Variables

Let's start with variables. So far we've stored values in signals. There is an alternative way to store values within a process; the variable.

Signals normally represent electrical connections or "pieces of wire". Variables can too, but they can also be more abstract. Variables can represent wires, registers, or be used to store intermediate values in abstract calculations.

```
variable V: STD_LOGIC;
```



Variables

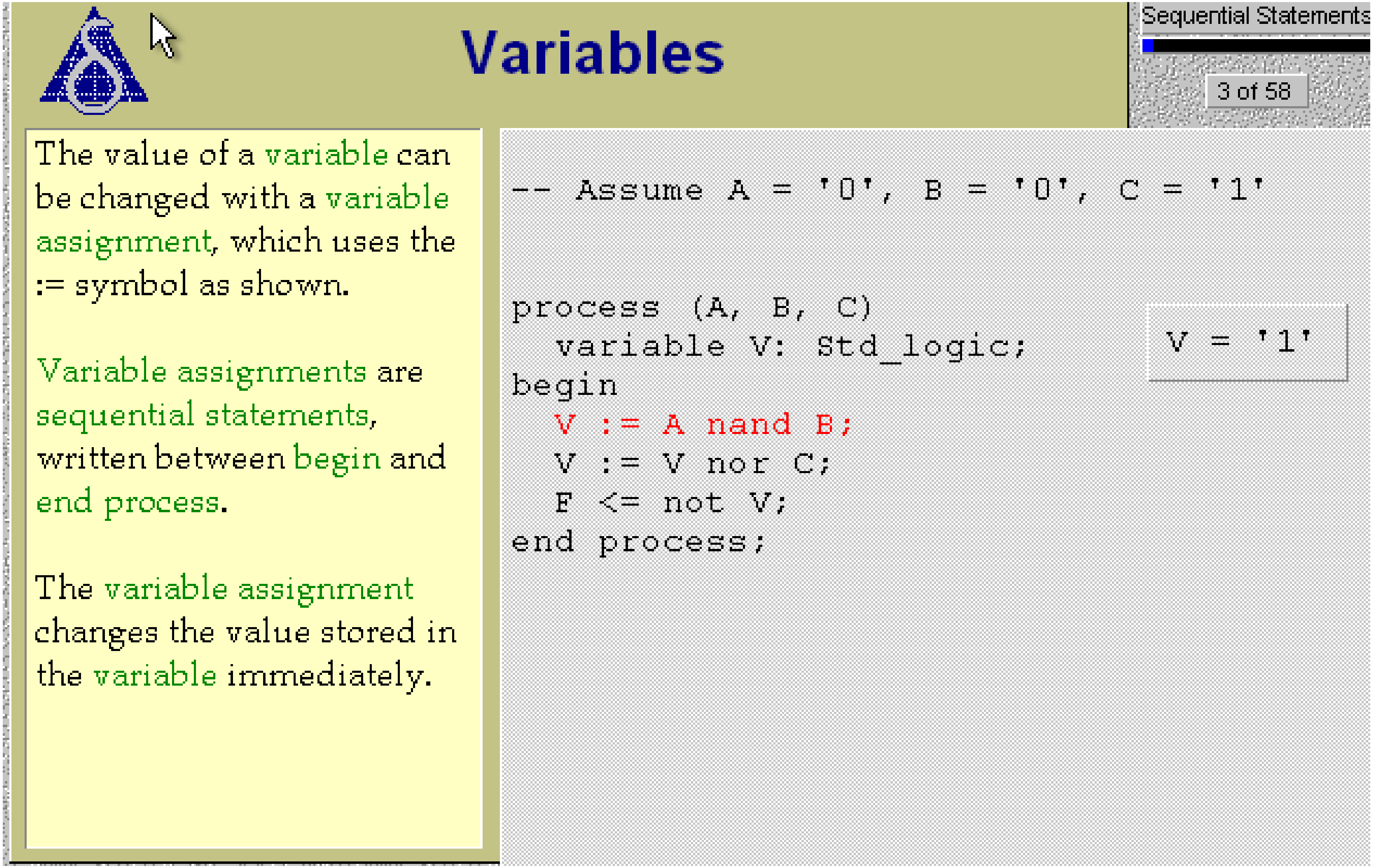
Variables must be defined inside a process, before begin (unlike signals, which we've seen defined in an architecture).

A variable has a name and a type, just like a signal.

The VHDL analyzer sets aside some storage to keep the current value of the variable.

```
-- Assume A = '0', B = '0', C = '1'


process (A, B, C)
  variable V: Std_logic;
begin
  V := A nand B;
  V := V nor C;
  F <= not V;
end process;
```

```
V = 'U'
```



Variables

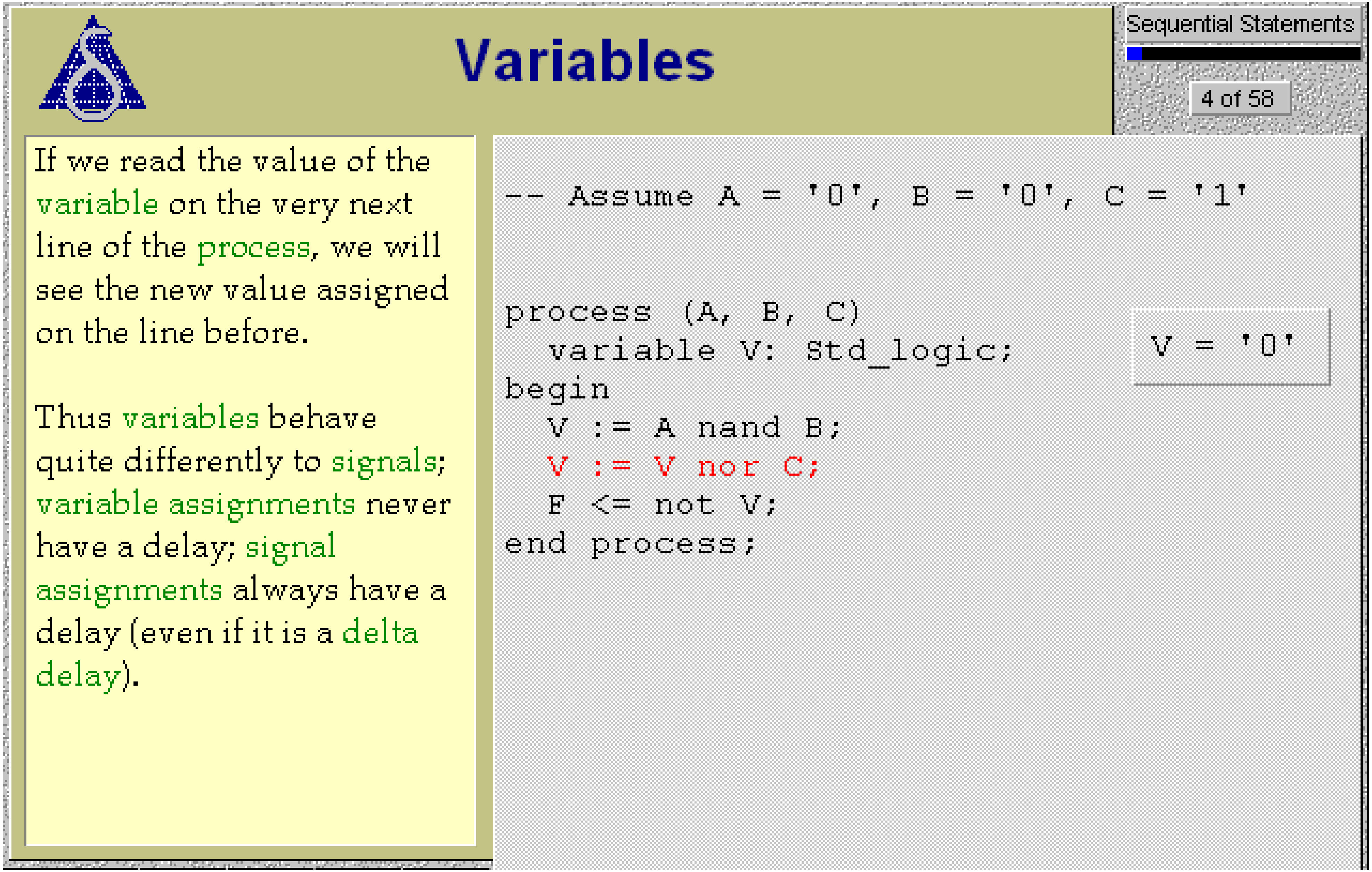
The value of a variable can be changed with a variable assignment, which uses the := symbol as shown.

Variable assignments are sequential statements, written between begin and end process.

The variable assignment changes the value stored in the variable immediately.

```
-- Assume A = '0', B = '0', C = '1'


process (A, B, C)
  variable V: Std_logic;
begin
  V := A nand B;
  V := V nor C;
  F <= not V;
end process;
```

V = '1'



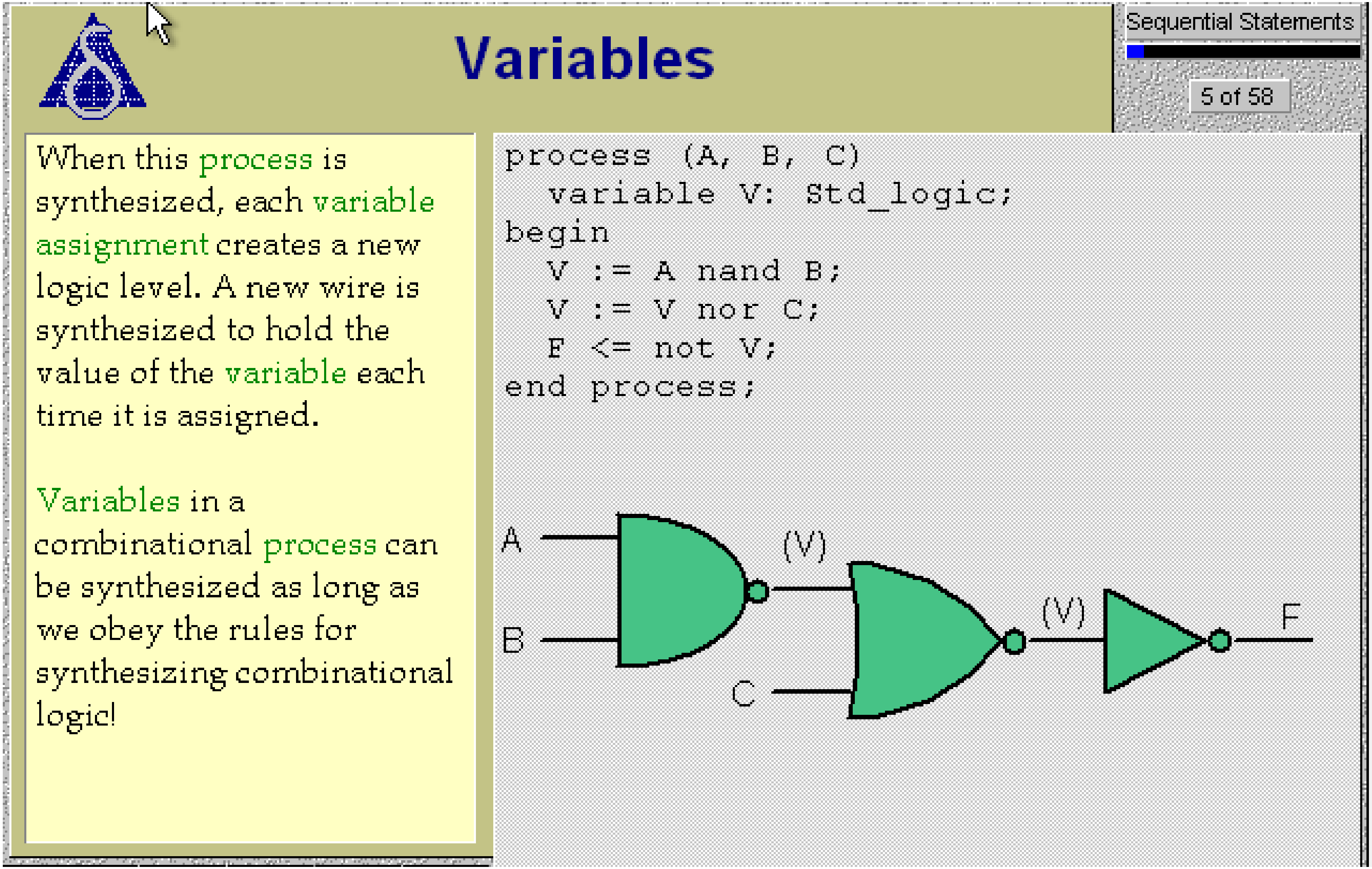
Variables

If we read the value of the variable on the very next line of the process, we will see the new value assigned on the line before.

Thus variables behave quite differently to signals; variable assignments never have a delay; signal assignments always have a delta delay (even if it is a delta delay).

```
-- Assume A = '0', B = '0', C = '1'


process (A, B, C)
  variable V: Std_logic;
begin
  V := A nand B;
  V := V nor C;
  F <= not V;
end process;
```

V = '0'

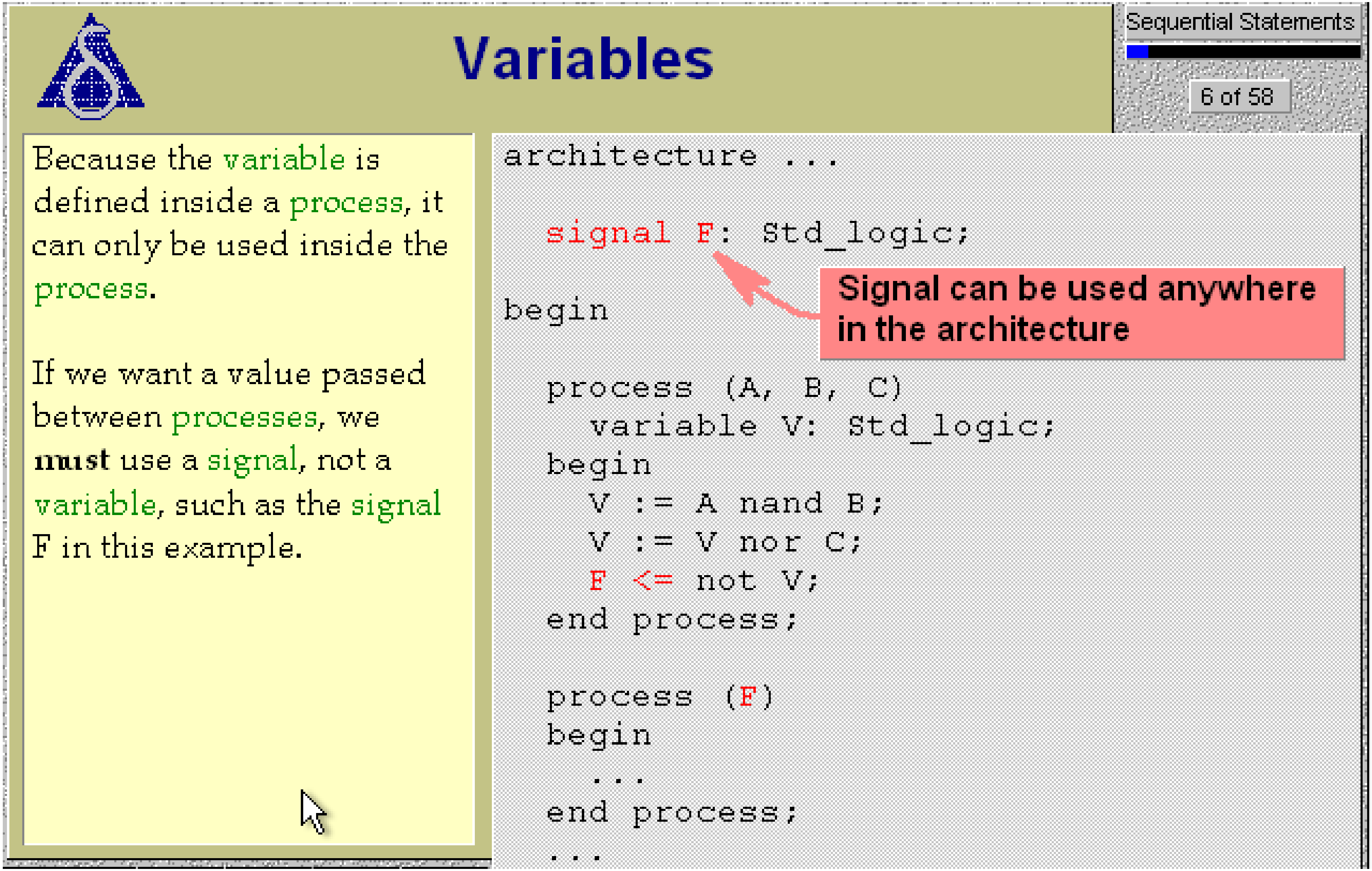


Variables

When this process is synthesized, each variable assignment creates a new logic level. A new wire is synthesized to hold the value of the variable each time it is assigned.

Variables in a combinational process can be synthesized as long as we obey the rules for synthesizing combinational logic!

```
process (A, B, C)
  variable V: Std_logic;
begin
  V := A nand B;
  V := V nor C;
  F <= not V;
end process;
```



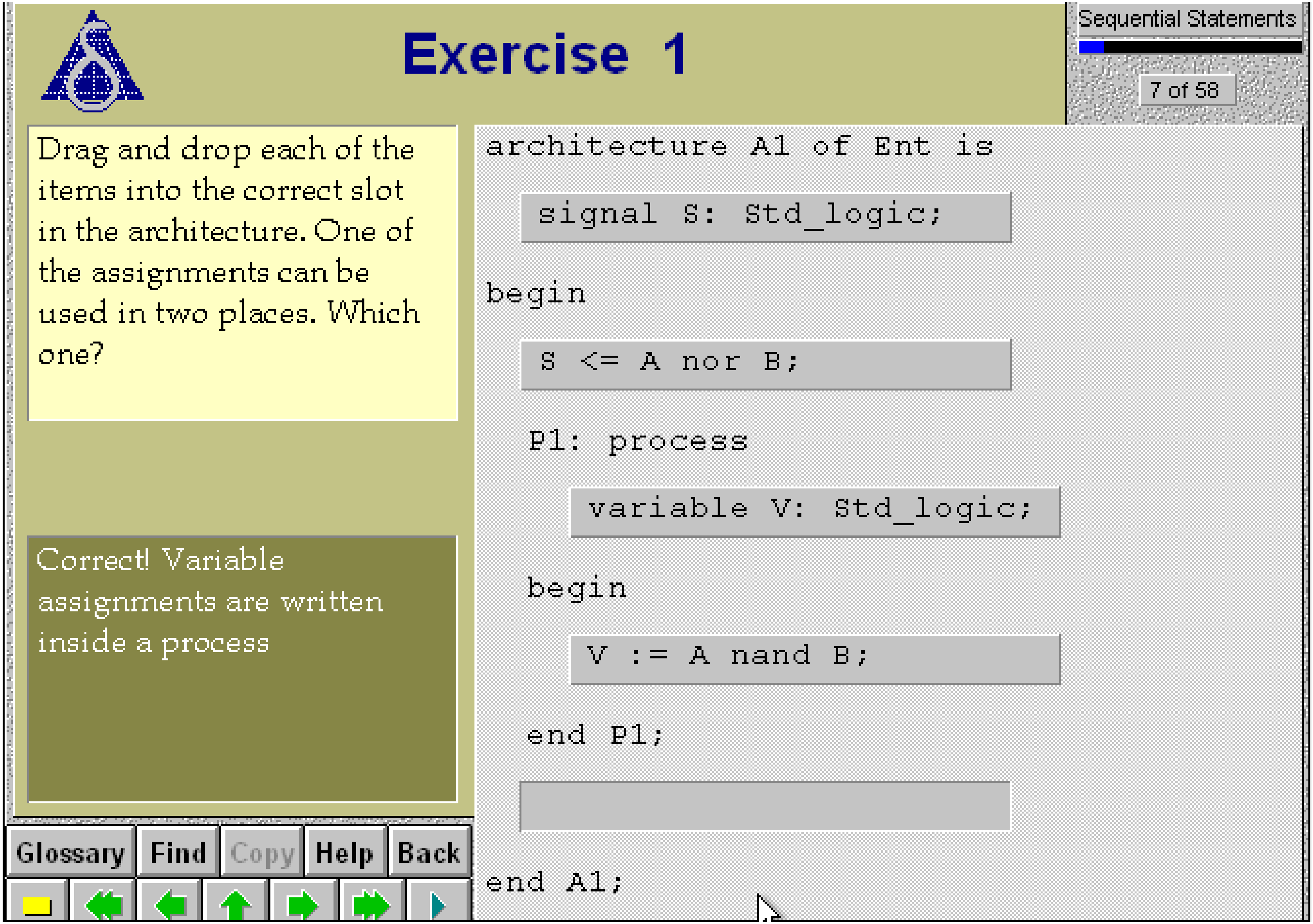
Variables

Because the variable is defined inside a process, it can only be used inside the process.

If we want a value passed between processes, we **must** use a signal, not a variable, such as the signal F in this example.

```
architecture ...

  signal F: Std_logic;

begin

  process (A, B, C)
    variable V: Std_logic;
  begin
    V := A nand B;
    V := V nor C;
    F <= not V;
  end process;

  process (F)
  begin
    ...
  end process;
  ...
```

Signal can be used anywhere in the architecture

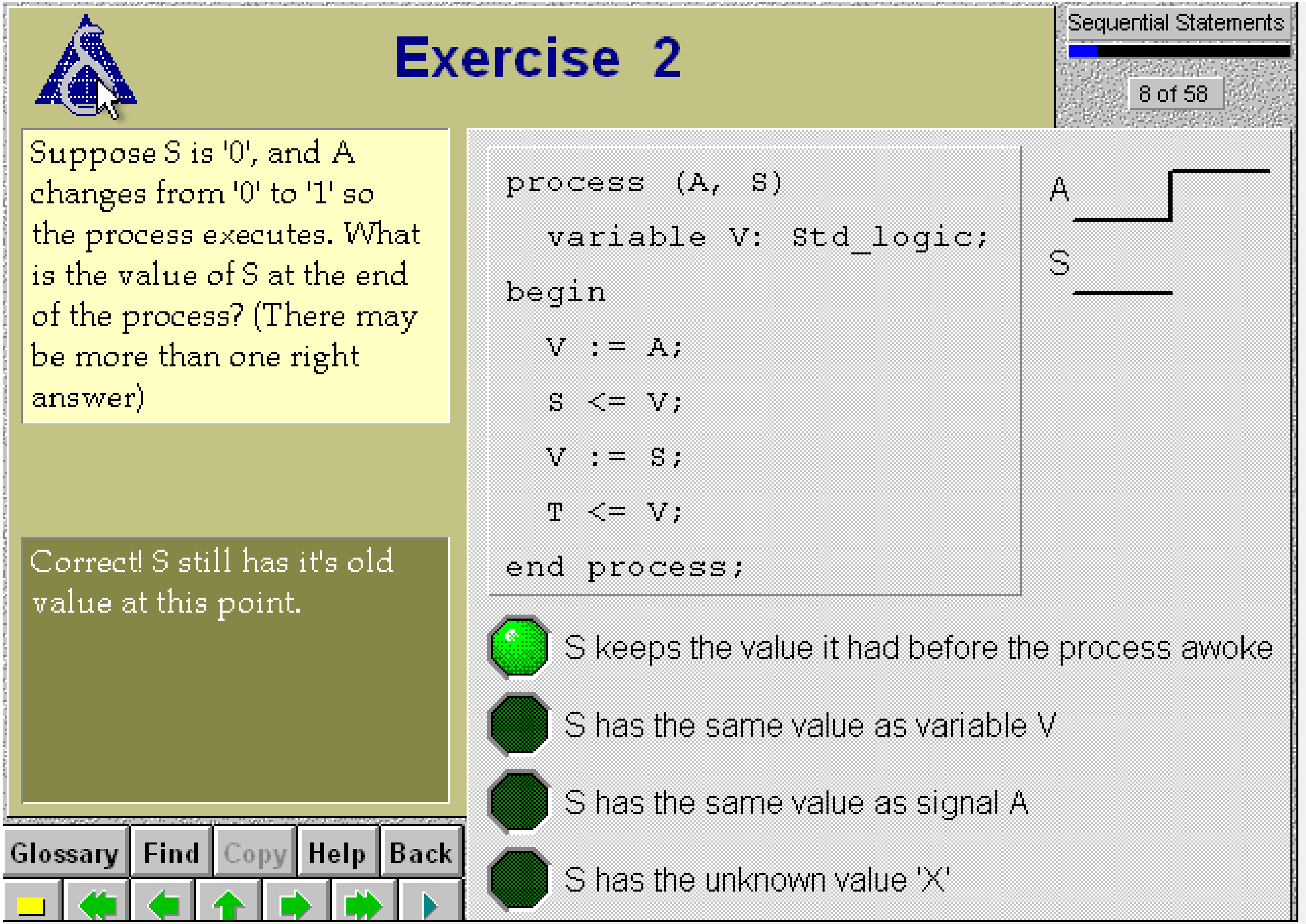


Exercise 1

Drag and drop each of the items into the correct slot in the architecture. One of the assignments can be used in two places. Which one?

Correct! Variable assignments are written inside a process

```
architecture A1 of Ent is

   signal S: Std_logic;

begin

   S <= A nor B;

   P1: process

      variable V: Std_logic;

   begin

      V := A nand B;

   end P1;

end A1;
```



Exercise 2

Suppose S is '0', and A changes from '0' to '1' so the process executes. What is the value of S at the end of the process? (There may be more than one right answer)

```
process (A, S)
  variable V: Std_logic;
begin
  V := A;
  S <= V;
  V := S;
  T <= V;
end process;
```

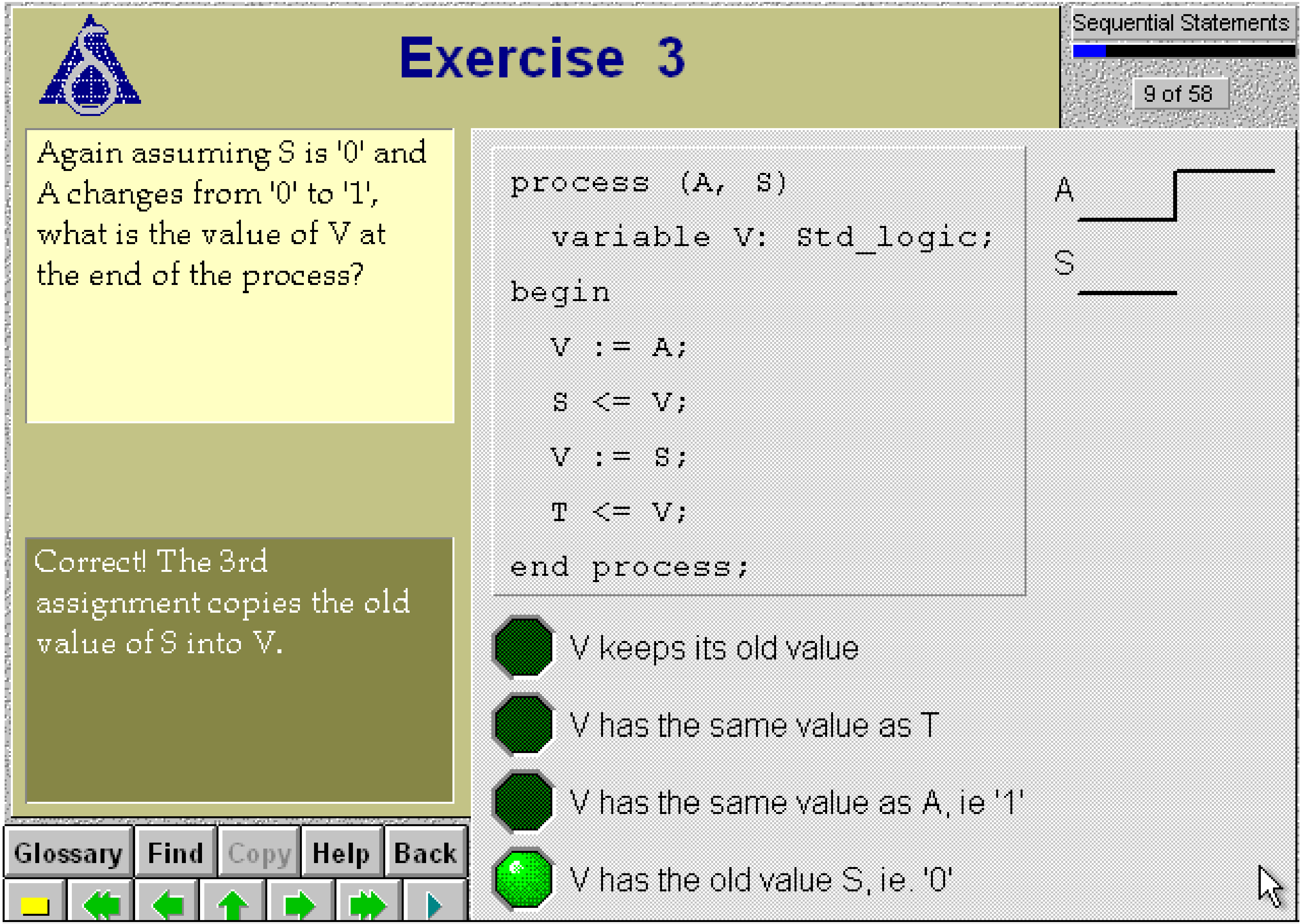
● S keeps the value it had before the process awoke

○ S has the same value as variable V

○ S has the same value as signal A

○ S has the unknown value 'X'

Correct! S still has it's old value at this point.



Exercise 3

Again assuming S is '0' and A changes from '0' to '1', what is the value of V at the end of the process?

```
process (A, S)
  variable V: Std_logic;
begin
  V := A;
  S <= V;
  V := S;
  T <= V;
end process;
```

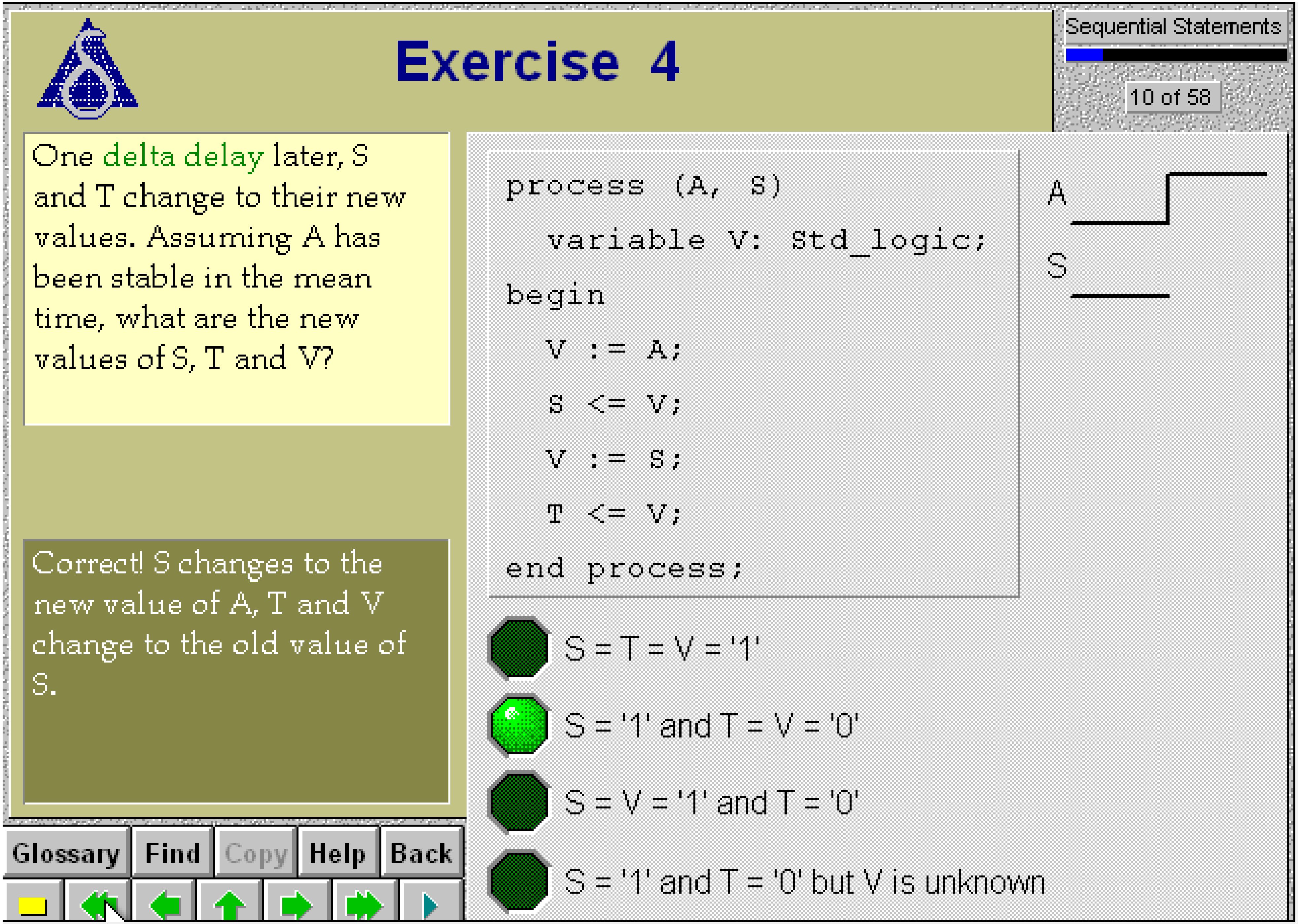
Correct! The 3rd assignment copies the old value of S into V.

○ V keeps its old value

○ V has the same value as T

○ V has the same value as A, ie '1'

● V has the old value S, ie. '0'



Exercise 4

One delta delay later, S and T change to their new values. Assuming A has been stable in the mean time, what are the new values of S, T and V?

```
process (A, S)
  variable V: Std_logic;
begin
  V := A;
  S <= V;
  V := S;
  T <= V;
end process;
```

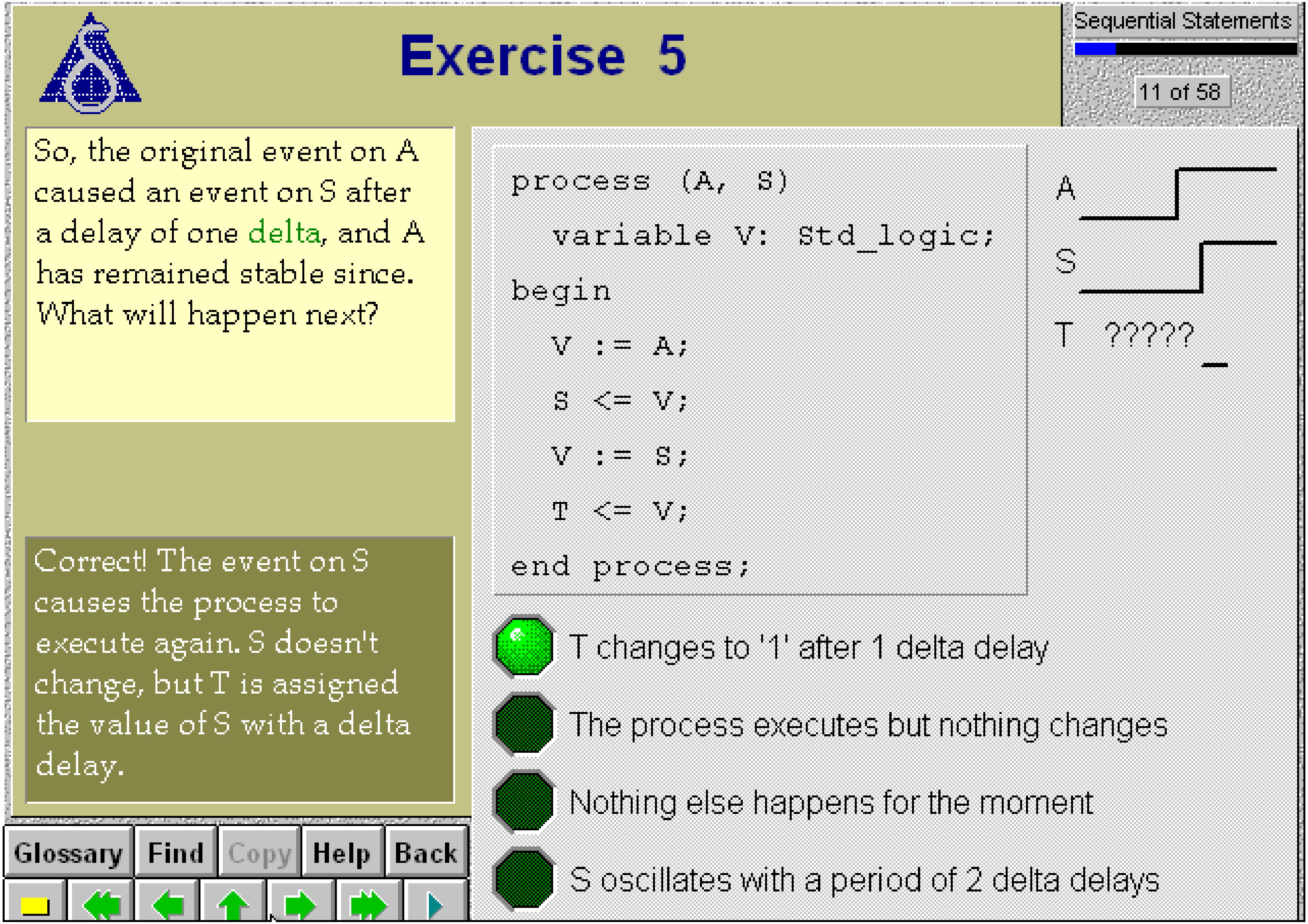
○ S = T = V = '1'

● S = '1' and T = V = '0'

○ S = V = '1' and T = '0'

○ S = '1' and T = '0' but V is unknown

Correct! S changes to the new value of A, T and V change to the old value of S.



Exercise 5

So, the original event on A caused an event on S after a delay of one delta, and A has remained stable since. What will happen next?

```
process (A, S)
  variable V: Std_logic;
begin
  V := A;
  S <= V;
  V := S;
  T <= V;
end process;
```

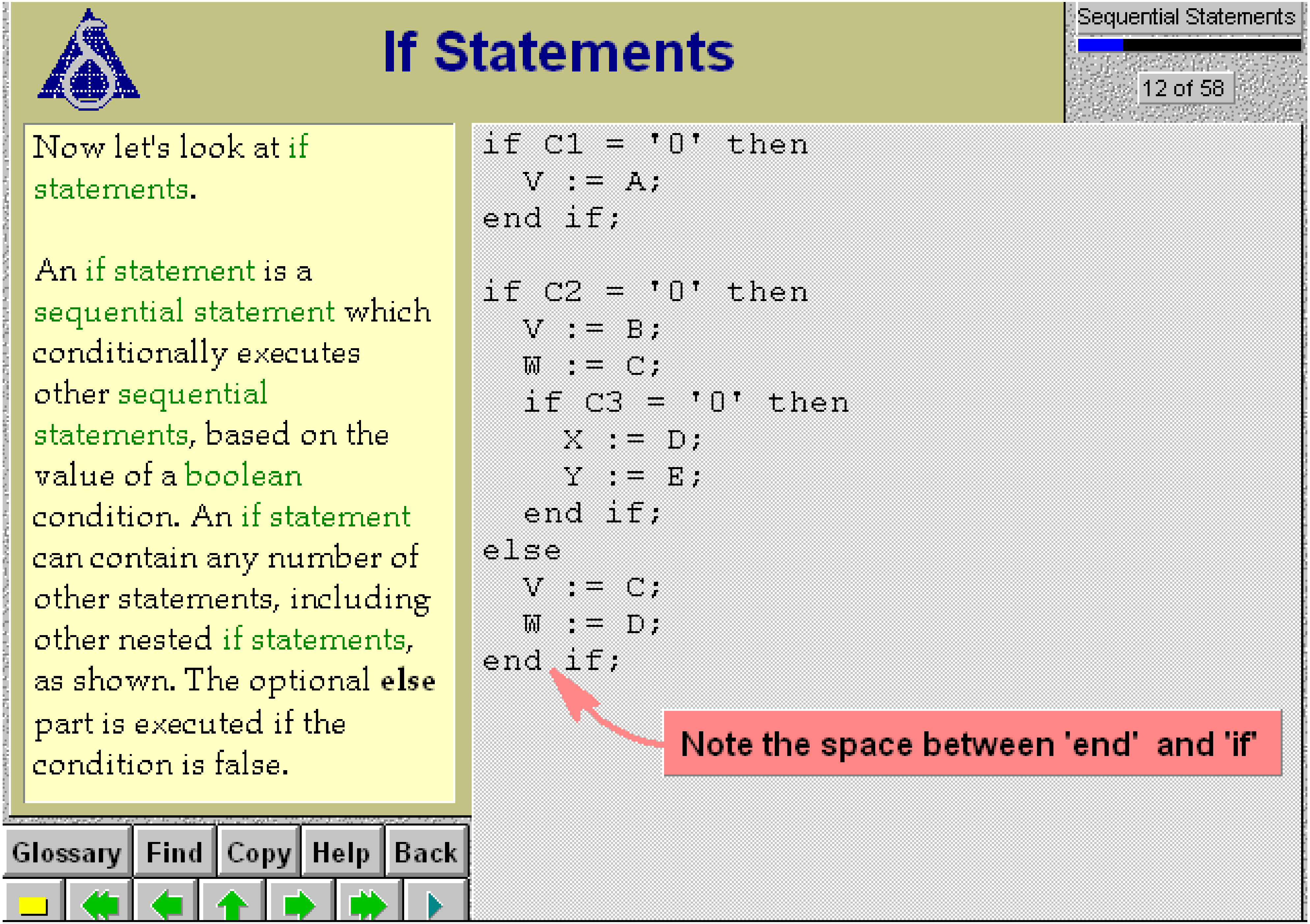
● T changes to '1' after 1 delta delay

○ The process executes but nothing changes

○ Nothing else happens for the moment

○ S oscillates with a period of 2 delta delays

Correct! The event on S causes the process to execute again. S doesn't change, but T is assigned the value of S with a delta delay.



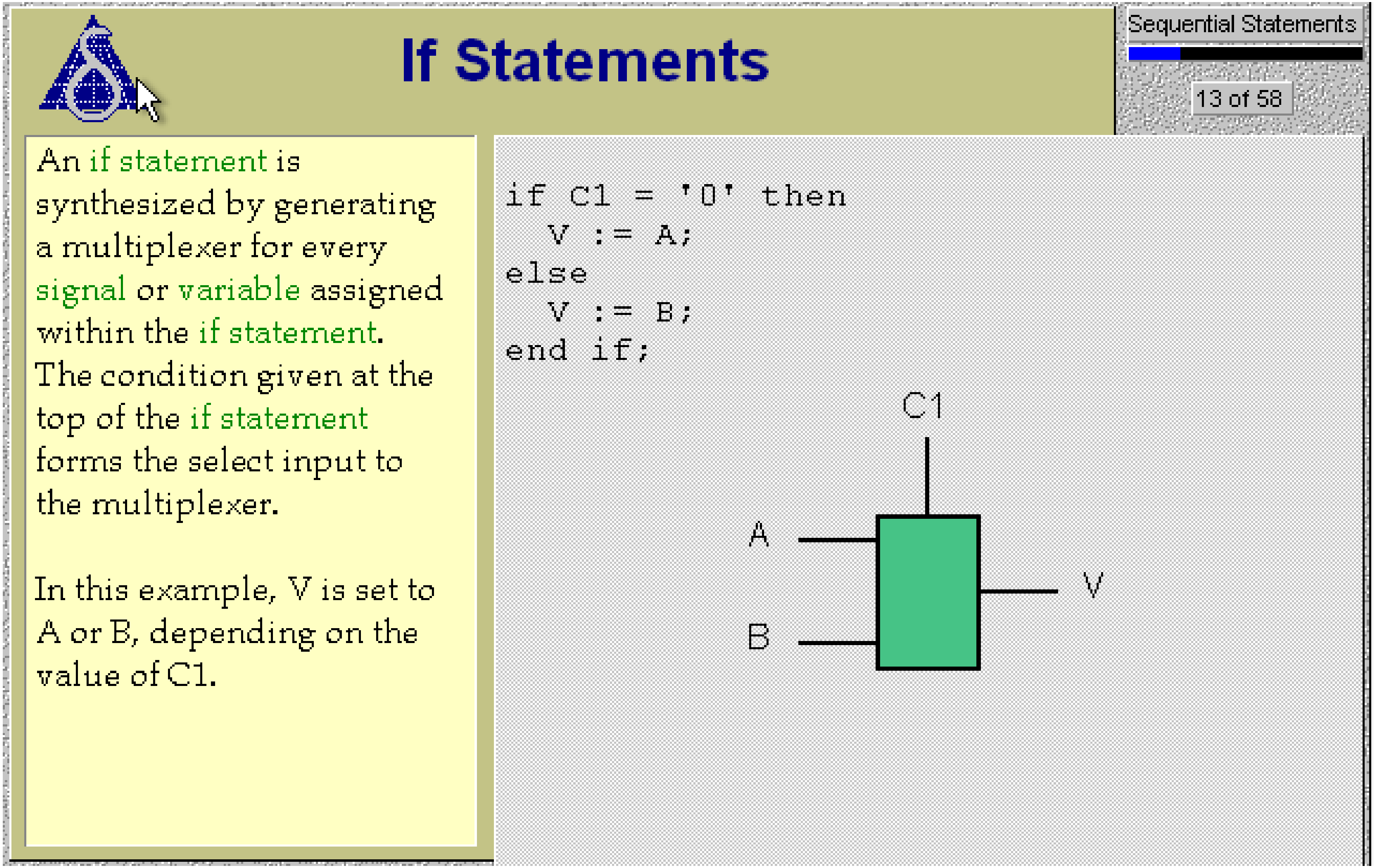
If Statements

Now let's look at if statements.

An if statement is a sequential statement which conditionally executes other sequential statements, based on the value of a boolean condition. An if statement can contain any number of other statements, including other nested if statements, as shown. The optional **else** part is executed if the condition is false.

```
if C1 = '0' then
  V := A;
end if;

if C2 = '0' then
  V := B;
  W := C;
  if C3 = '0' then
    X := D;
    Y := E;
  end if;
else
  V := C;
  W := D;
end if;
```

Note the space between 'end' and 'if'



If Statements

An if statement is synthesized by generating a multiplexer for every signal or variable assigned within the if statement. The condition given at the top of the if statement forms the select input to the multiplexer.

In this example, V is set to A or B, depending on the value of C1.

```
if C1 = '0' then
   V := A;
else
   V := B;
end if;
```

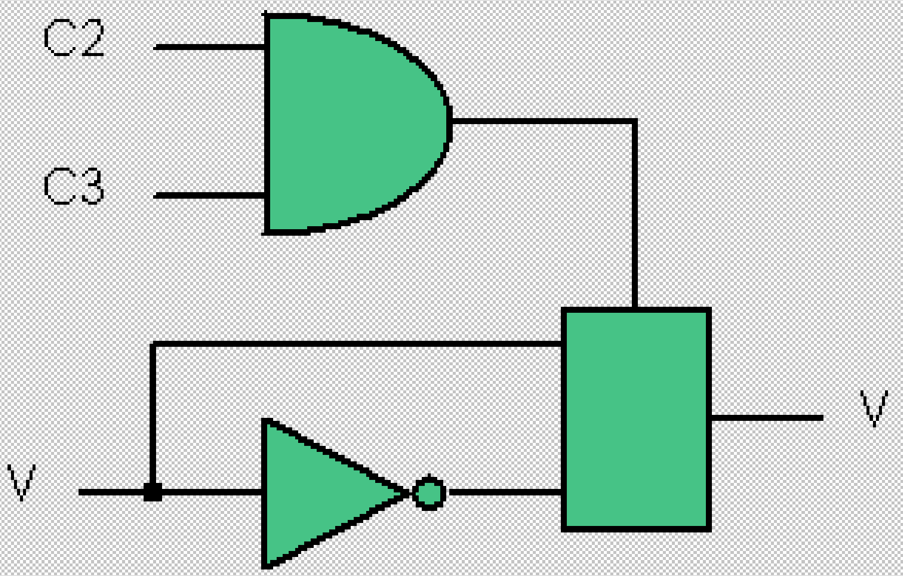
If Statements

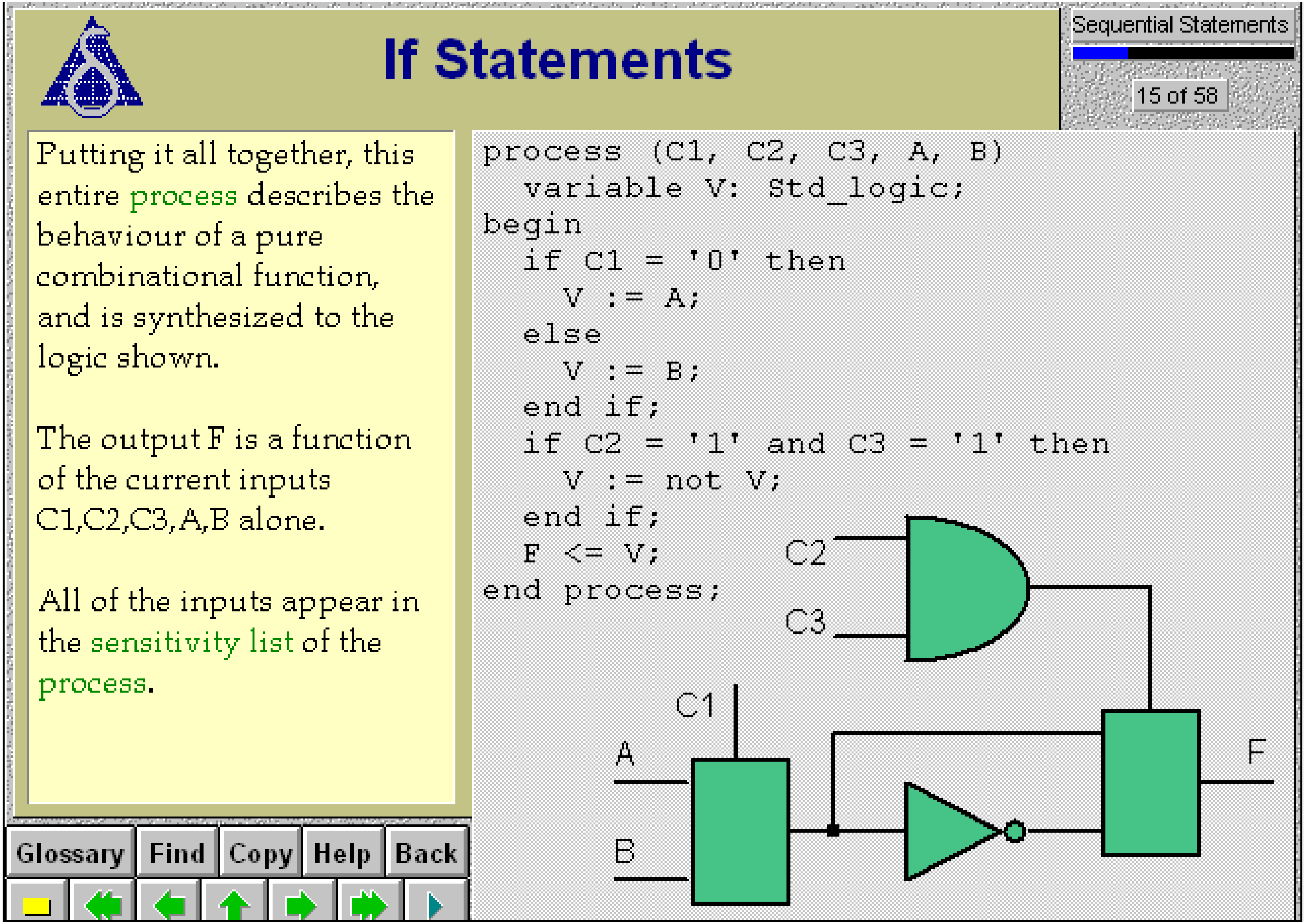
This if statement does not have an **else** part, but the assignment to V within the **if** still synthesizes to a multiplexer.

The boolean condition is synthesized to logic (an **and** gate) on the select line.

If the condition is false, the previous value of V is fed through the multiplexer unaltered.

```
if C2 = '1' and C3 = '1' then
  V := not V;
end if;
```





If Statements

Sequential Statements

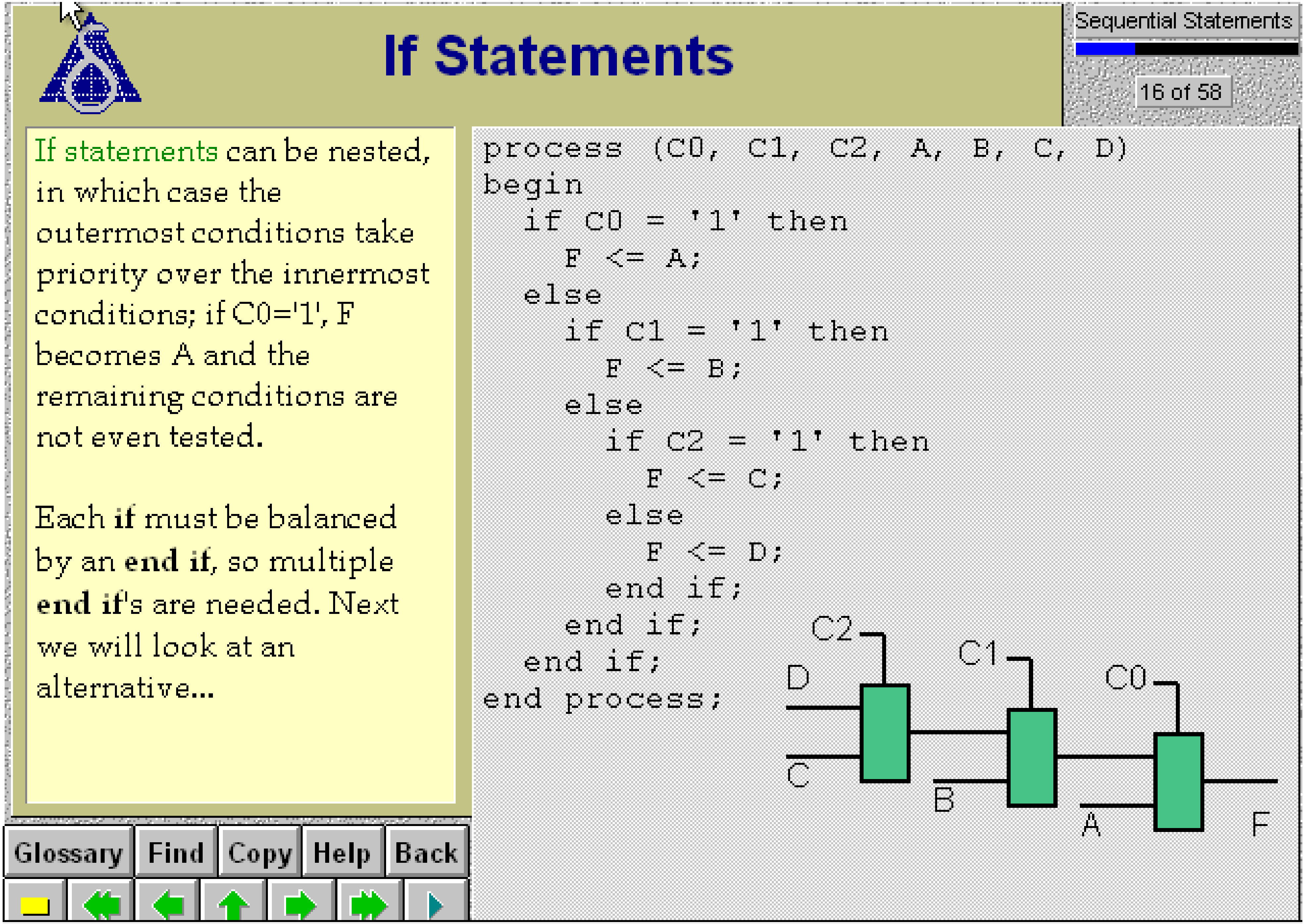
15 of 58

Putting it all together, this entire process describes the behaviour of a pure combinational function, and is synthesized to the logic shown.

The output F is a function of the current inputs C1,C2,C3,A,B alone.

All of the inputs appear in the sensitivity list of the process.

```
process (C1, C2, C3, A, B)
   variable V: Std_logic;
begin
   if C1 = '0' then
      V := A;
   else
      V := B;
   end if;
   if C2 = '1' and C3 = '1' then
      V := not V;
   end if;
   F <= V;
end process;
```

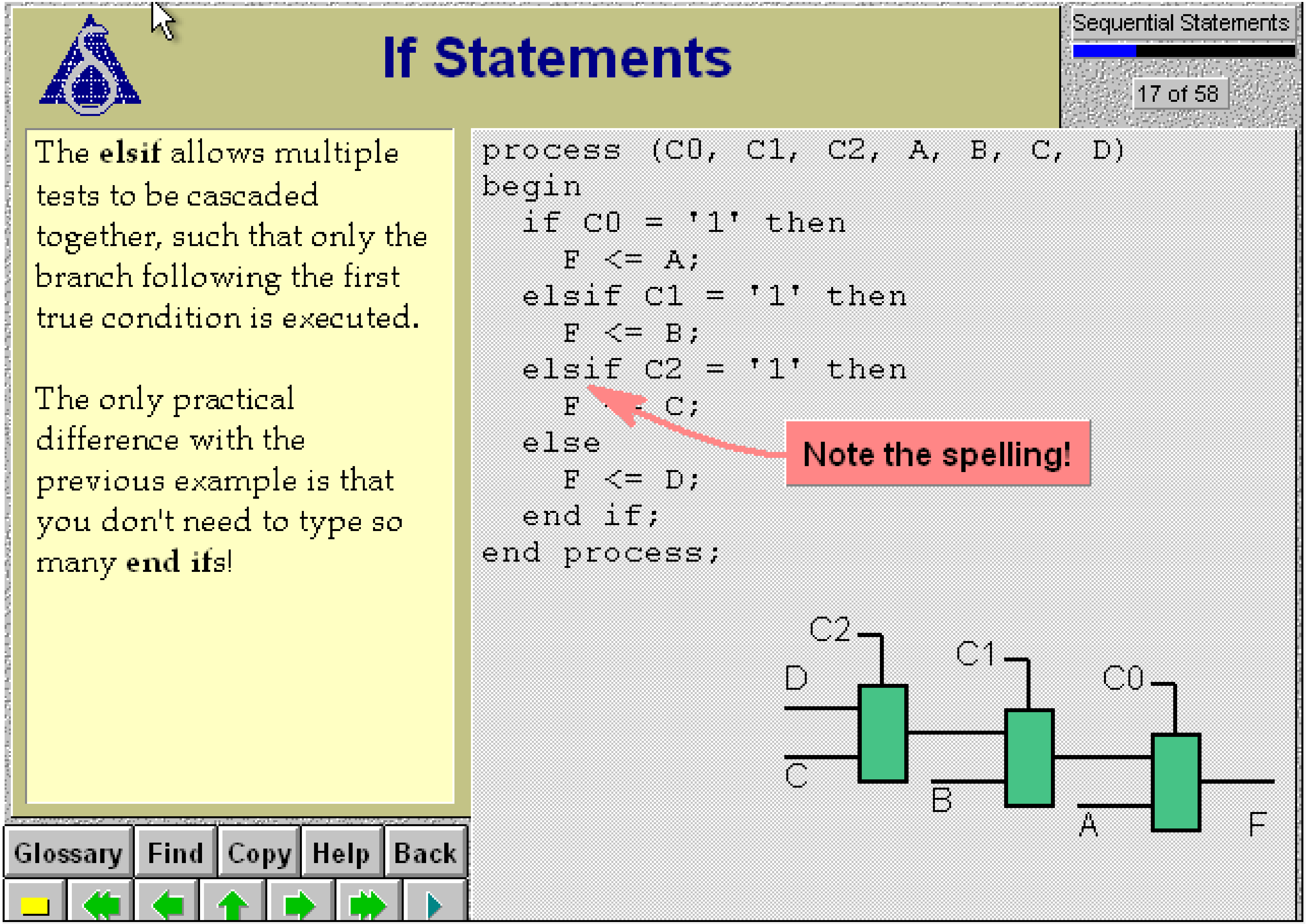


If Statements

If statements can be nested, in which case the outermost conditions take priority over the innermost conditions; if C0='1', F becomes A and the remaining conditions are not even tested.

Each **if** must be balanced by an **end if**, so multiple **end if**'s are needed. Next we will look at an alternative...

```
process (C0, C1, C2, A, B, C, D)
begin
   if C0 = '1' then
      F <= A;
   else
      if C1 = '1' then
         F <= B;
      else
         if C2 = '1' then
            F <= C;
         else
            F <= D;
         end if;
      end if;
   end if;
end process;
```



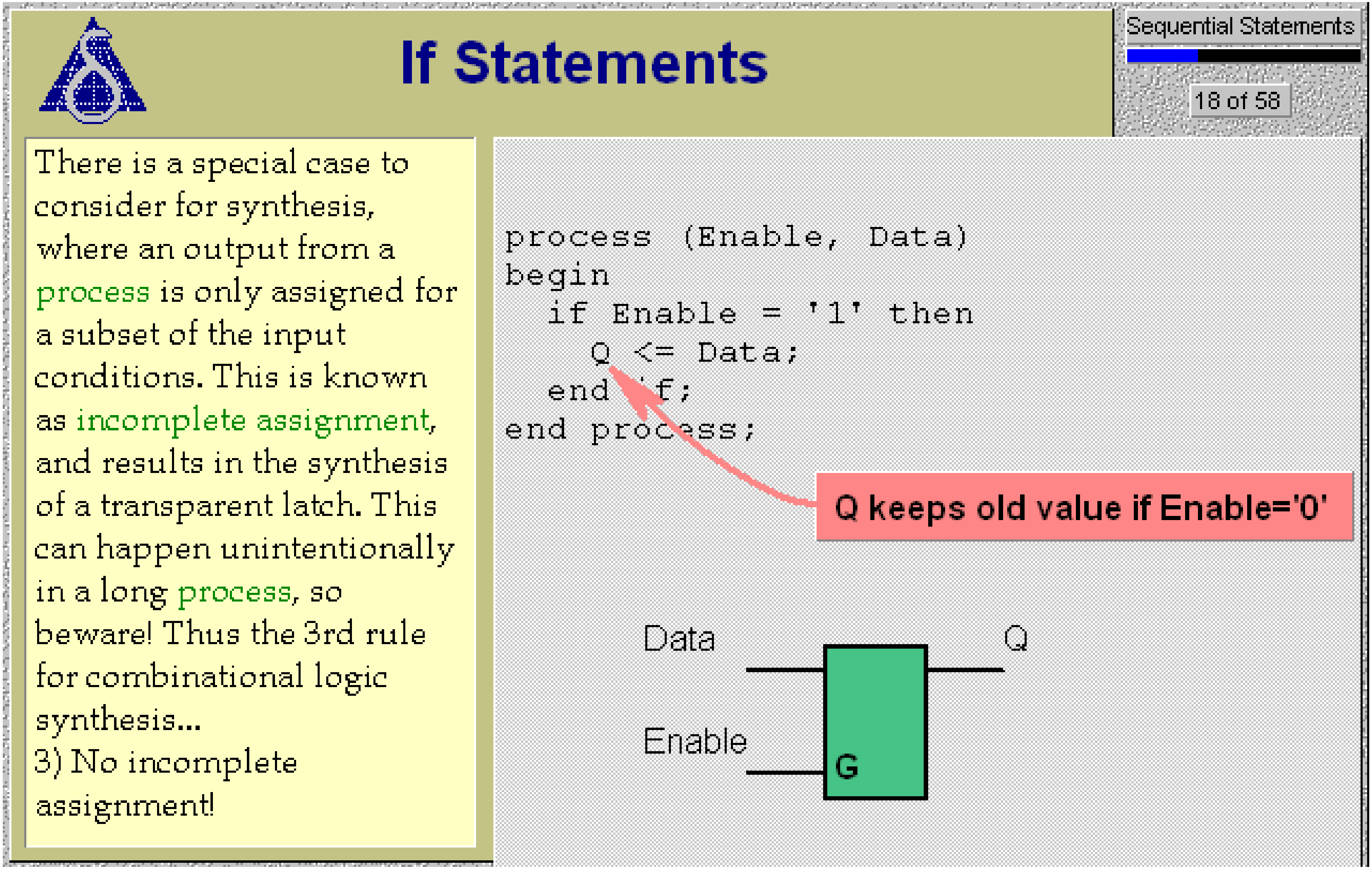
If Statements

The **elsif** allows multiple tests to be cascaded together, such that only the branch following the first true condition is executed.

The only practical difference with the previous example is that you don't need to type so many **end ifs**!

```
process (C0, C1, C2, A, B, C, D)
begin
  if C0 = '1' then
    F <= A;
  elsif C1 = '1' then
    F <= B;
  elsif C2 = '1' then
    F <= C;
  else
    F <= D;
  end if;
end process;
```

Note the spelling!



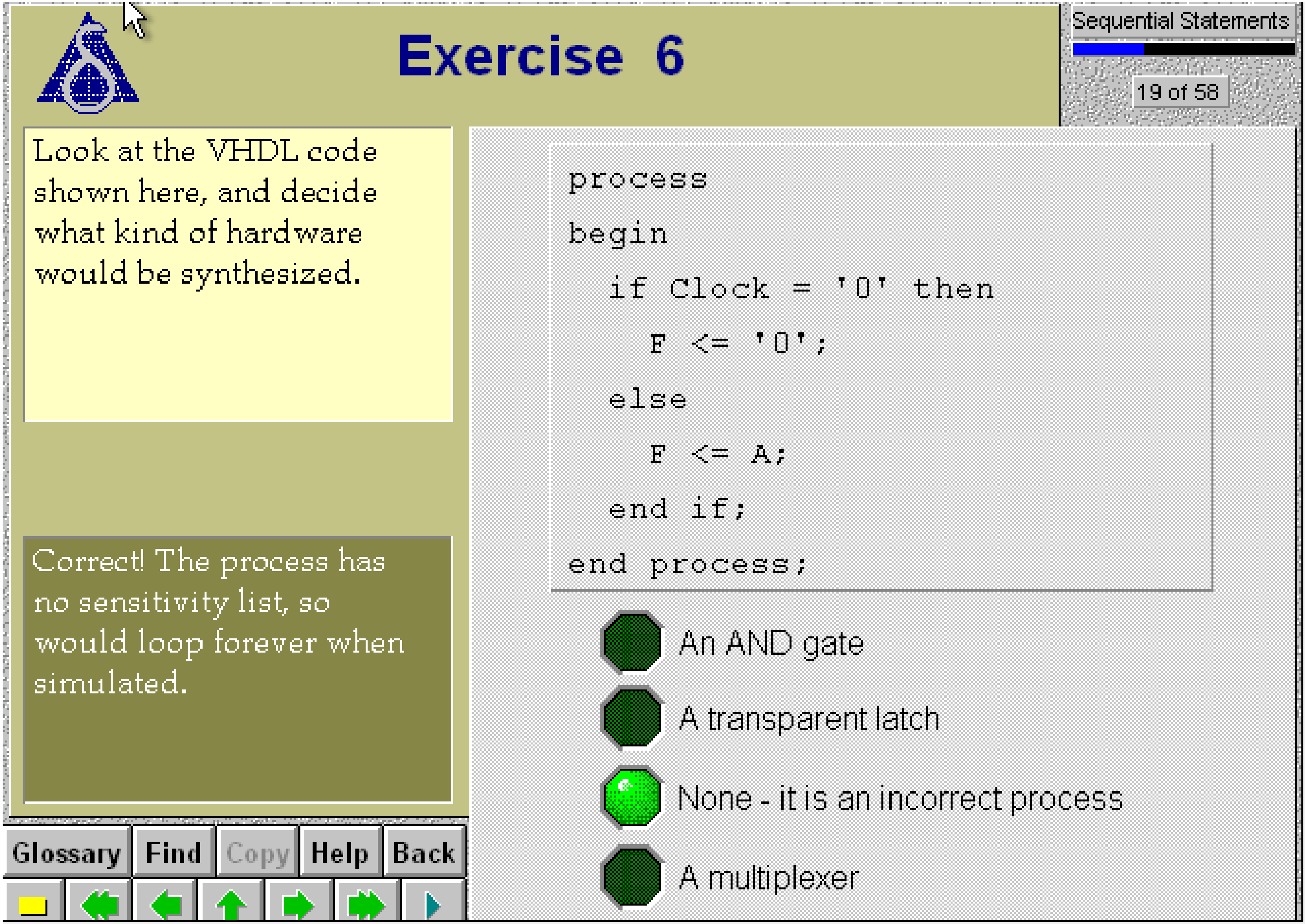
If Statements

There is a special case to consider for synthesis, where an output from a process is only assigned for a subset of the input conditions. This is known as incomplete assignment, and results in the synthesis of a transparent latch. This can happen unintentionally in a long process, so beware! Thus the 3rd rule for combinational logic synthesis...

3) No incomplete assignment!

```
process (Enable, Data)
begin
  if Enable = '1' then
    Q <= Data;
  end if;
end process;
```

Q keeps old value if Enable='0'



Exercise 6

Look at the VHDL code shown here, and decide what kind of hardware would be synthesized.

```
process
begin
  if Clock = '0' then
    F <= '0';
  else
    F <= A;
  end if;
end process;
```

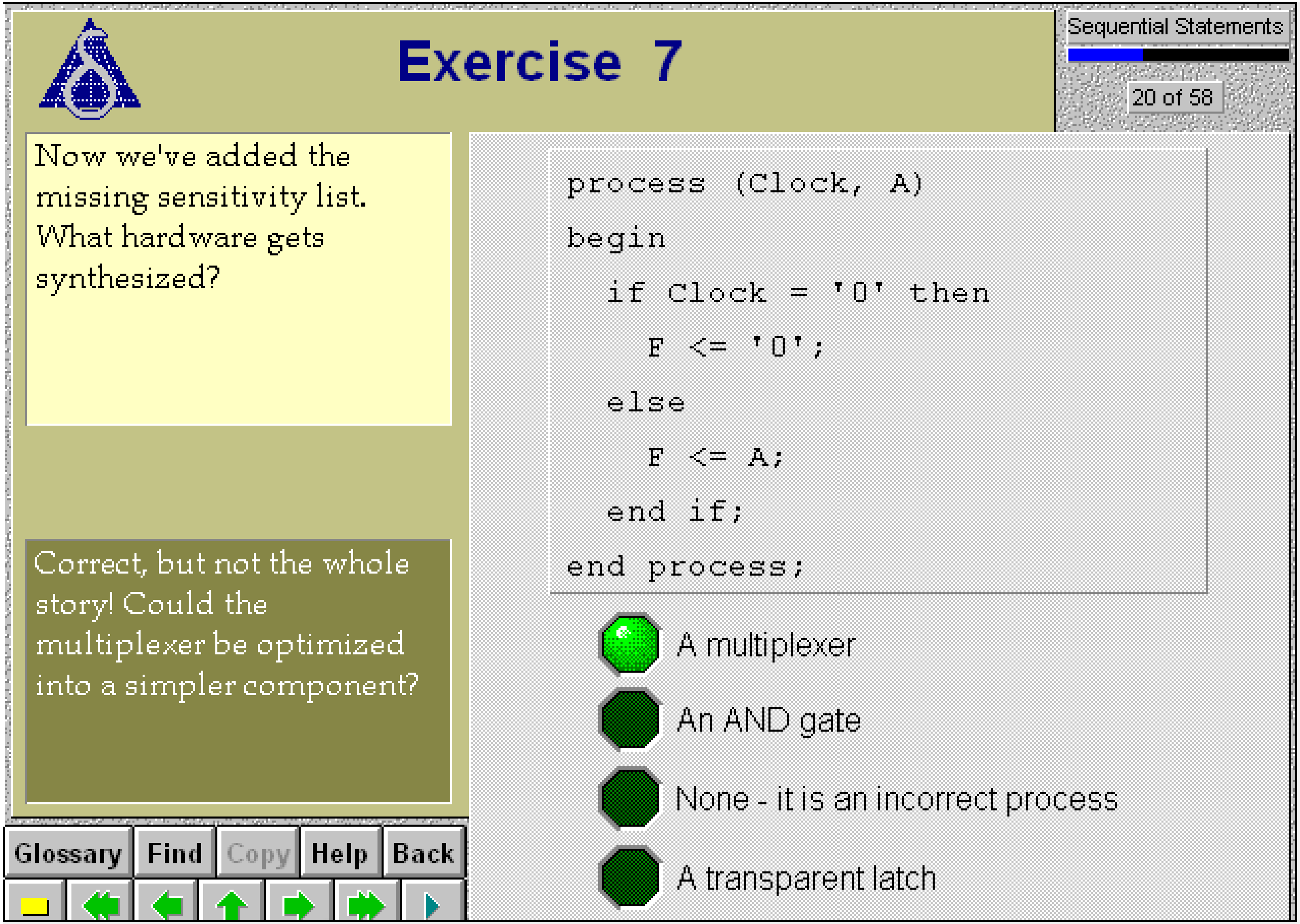
○ An AND gate

○ A transparent latch

● None - it is an incorrect process

○ A multiplexer

Correct! The process has no sensitivity list, so would loop forever when simulated.



Exercise 7

Now we've added the missing sensitivity list. What hardware gets synthesized?

Correct, but not the whole story! Could the multiplexer be optimized into a simpler component?

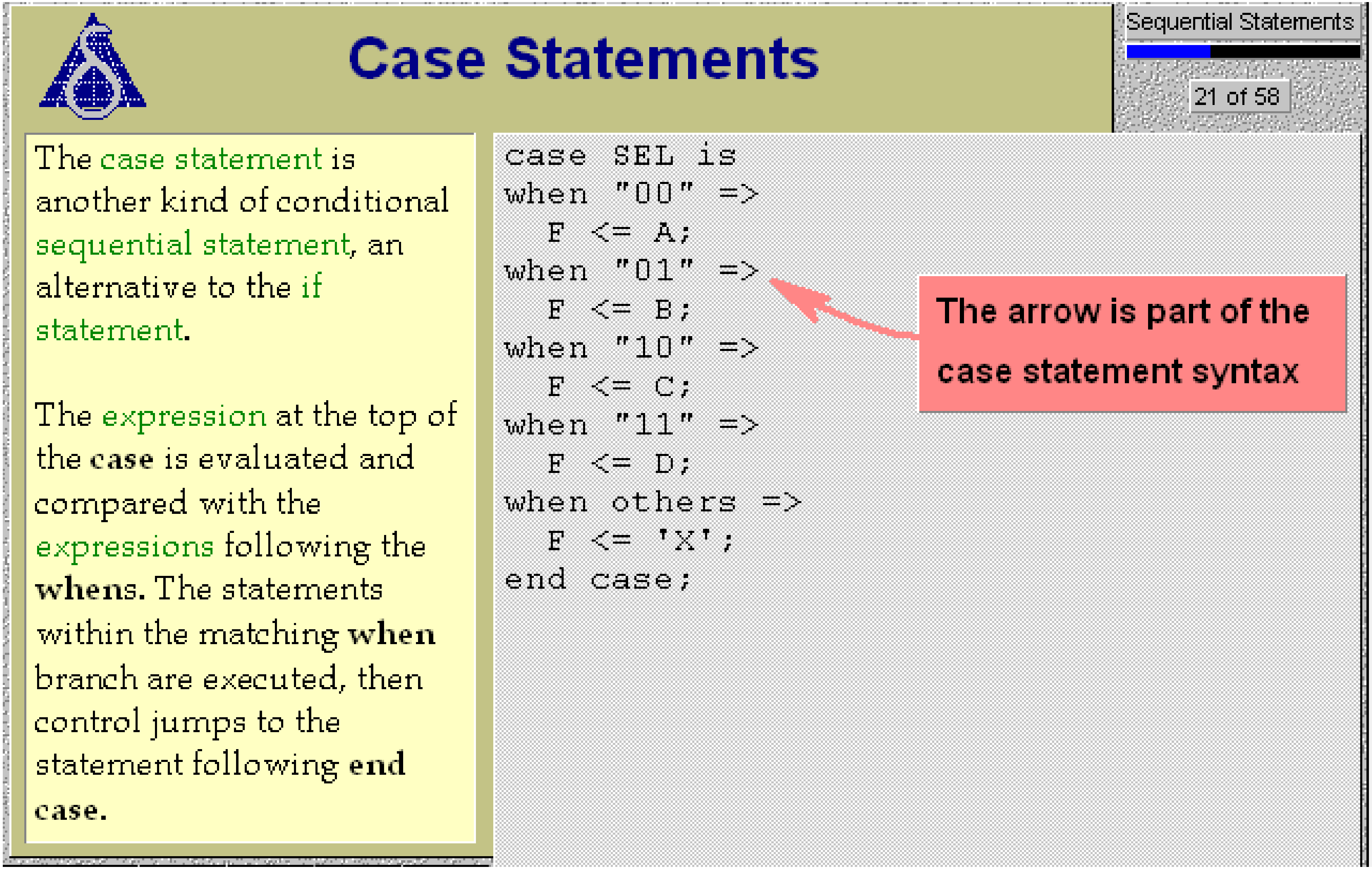
```
process (Clock, A)
begin
  if Clock = '0' then
    F <= '0';
  else
    F <= A;
  end if;
end process;
```

● A multiplexer

○ An AND gate

○ None - it is an incorrect process

○ A transparent latch



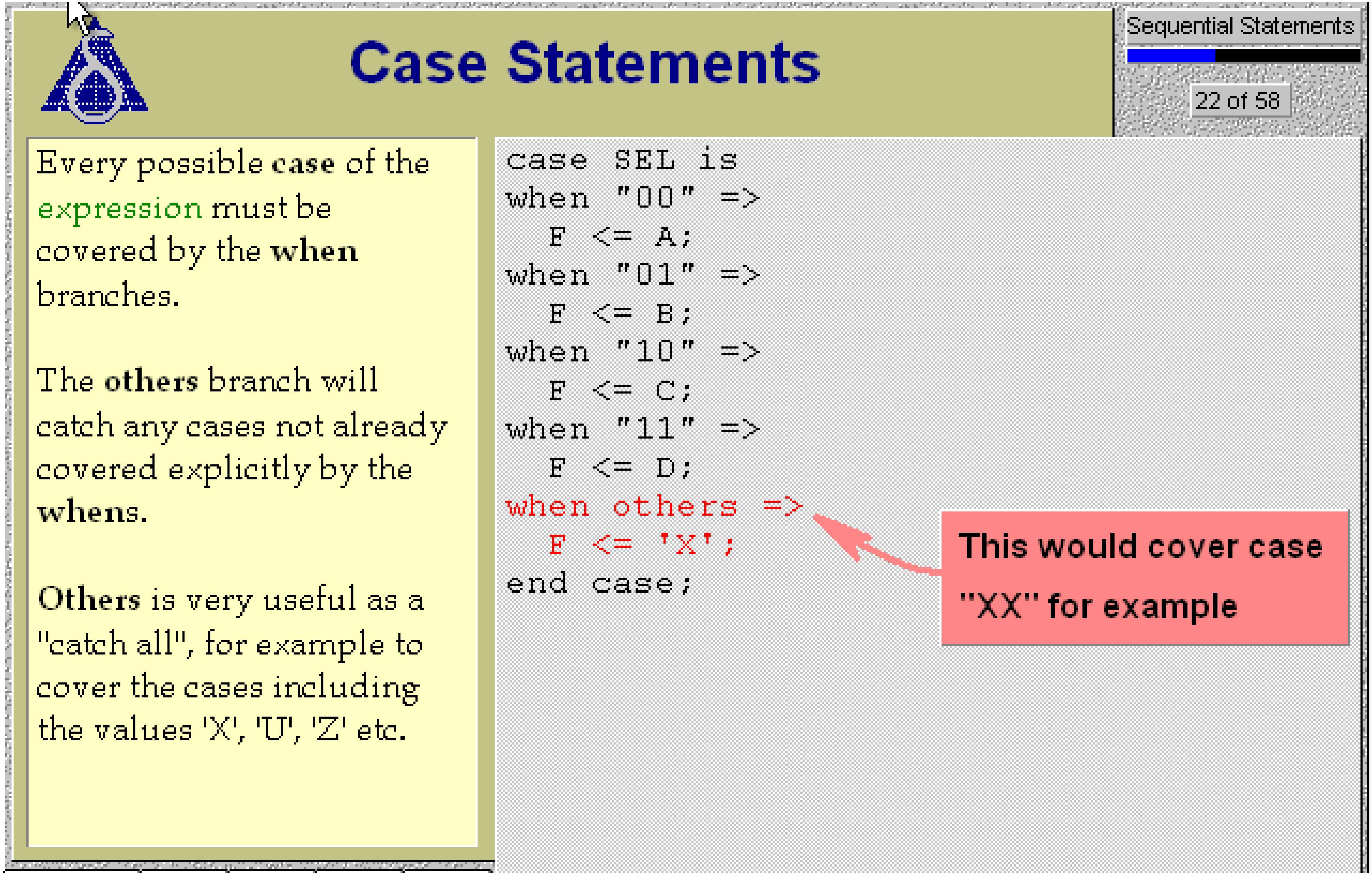
Case Statements

The case statement is another kind of conditional sequential statement, an alternative to the if statement.

The expression at the top of the **case** is evaluated and compared with the expressions following the **whens**. The statements within the matching **when** branch are executed, then control jumps to the statement following **end case.**

```
case SEL is
when "00" =>
  F <= A;
when "01" =>
  F <= B;
when "10" =>
  F <= C;
when "11" =>
  F <= D;
when others =>
  F <= 'X';
end case;
```

The arrow is part of the case statement syntax



Case Statements

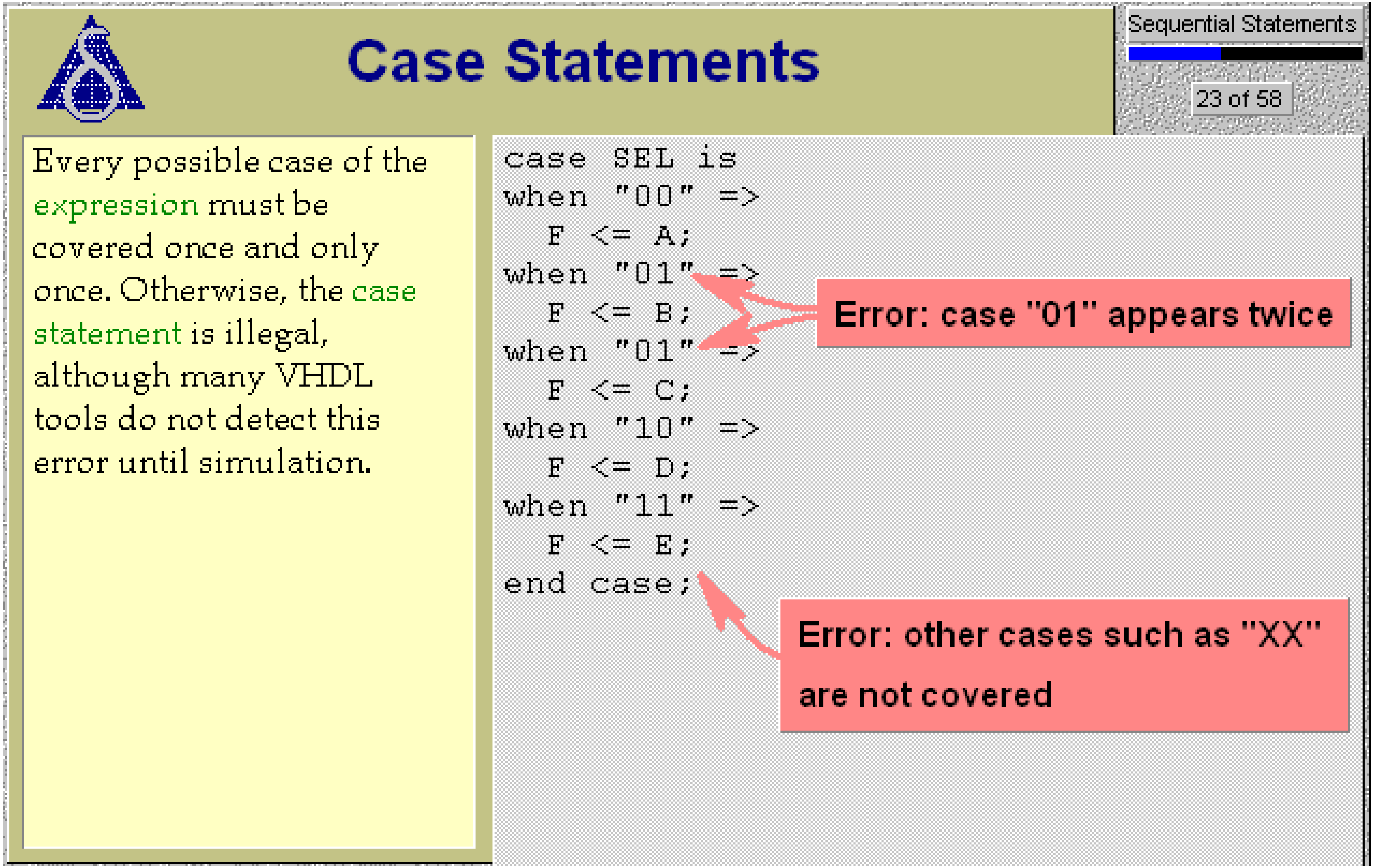
Every possible **case** of the expression must be covered by the **when** branches.

The **others** branch will catch any cases not already covered explicitly by the **whens.**

Others is very useful as a "catch all", for example to cover the cases including the values 'X', 'U', 'Z' etc.

```
case SEL is
when "00" =>
   F <= A;
when "01" =>
   F <= B;
when "10" =>
   F <= C;
when "11" =>
   F <= D;
when others =>
   F <= 'X';
end case;
```

This would cover case "XX" for example



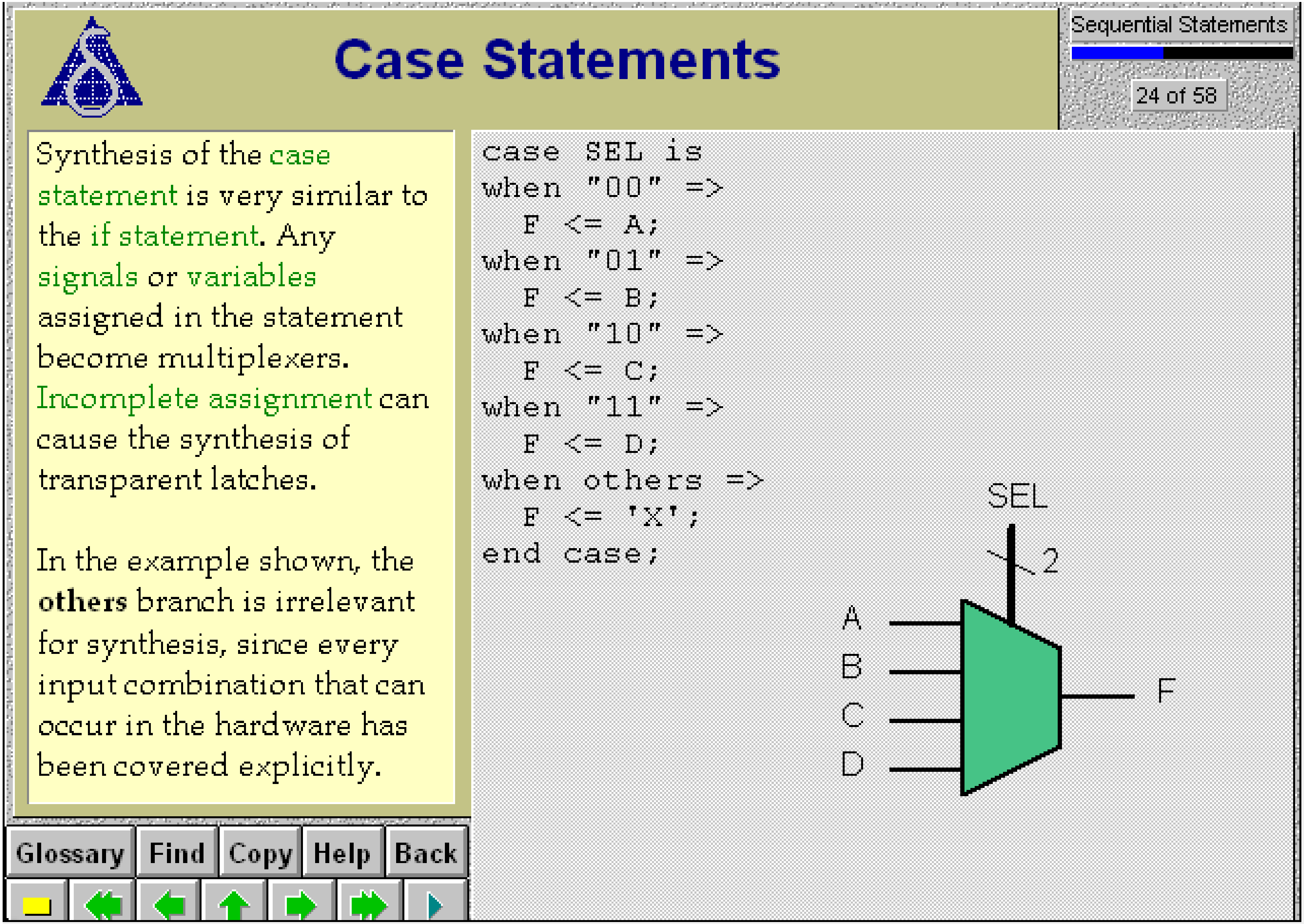
Case Statements

Every possible case of the expression must be covered once and only once. Otherwise, the case statement is illegal, although many VHDL tools do not detect this error until simulation.

```
case SEL is
when "00" =>
  F <= A;
when "01" =>
  F <= B;
when "01" =>
  F <= C;
when "10" =>
  F <= D;
when "11" =>
  F <= E;
end case;
```

Error: case "01" appears twice

Error: other cases such as "XX" are not covered

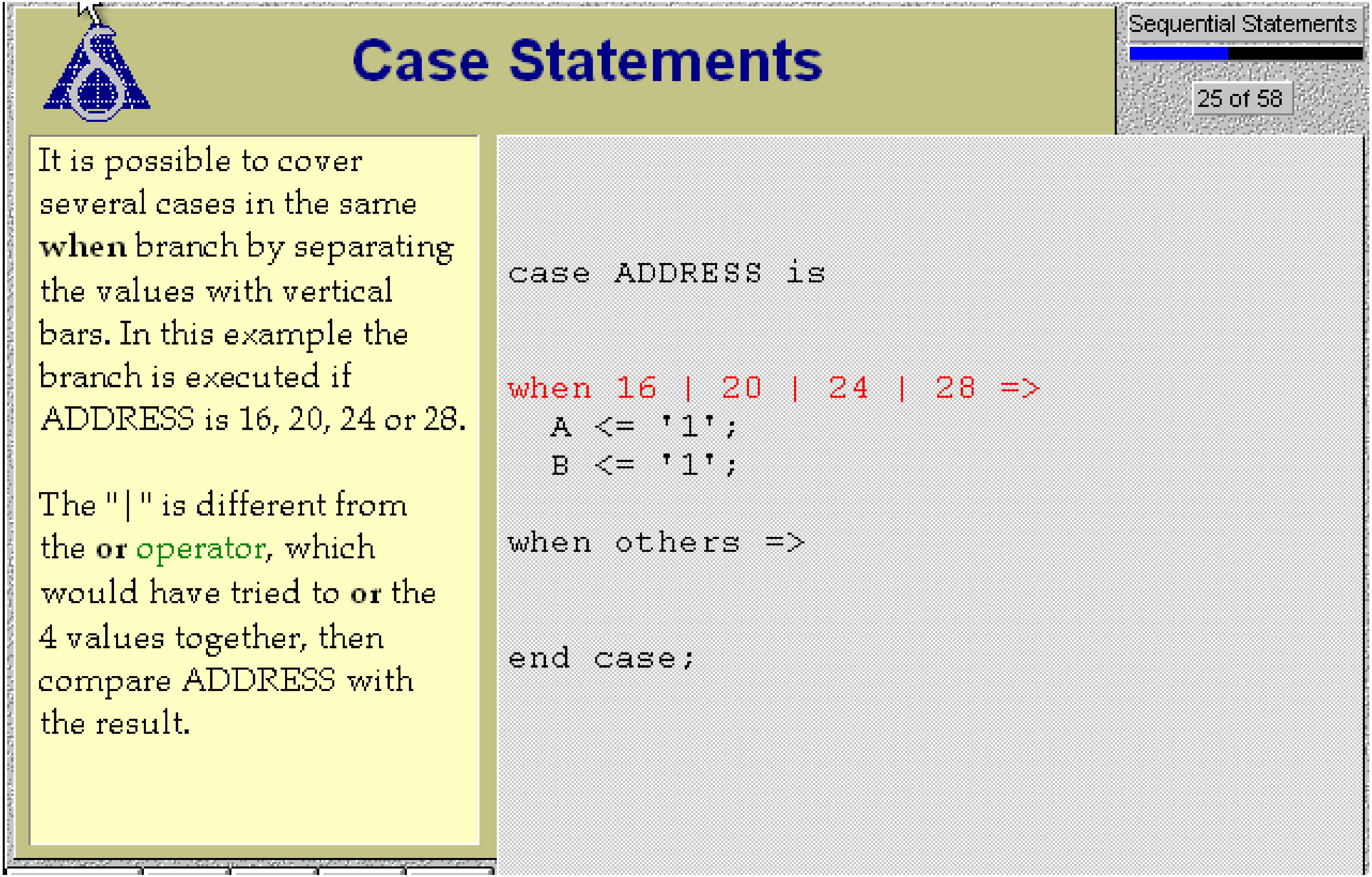


Case Statements

Synthesis of the case statement is very similar to the if statement. Any signals or variables assigned in the statement become multiplexers. Incomplete assignment can cause the synthesis of transparent latches.

In the example shown, the **others** branch is irrelevant for synthesis, since every input combination that can occur in the hardware has been covered explicitly.

```
case SEL is
when "00" =>
  F <= A;
when "01" =>
  F <= B;
when "10" =>
  F <= C;
when "11" =>
  F <= D;
when others =>
  F <= 'X';
end case;
```

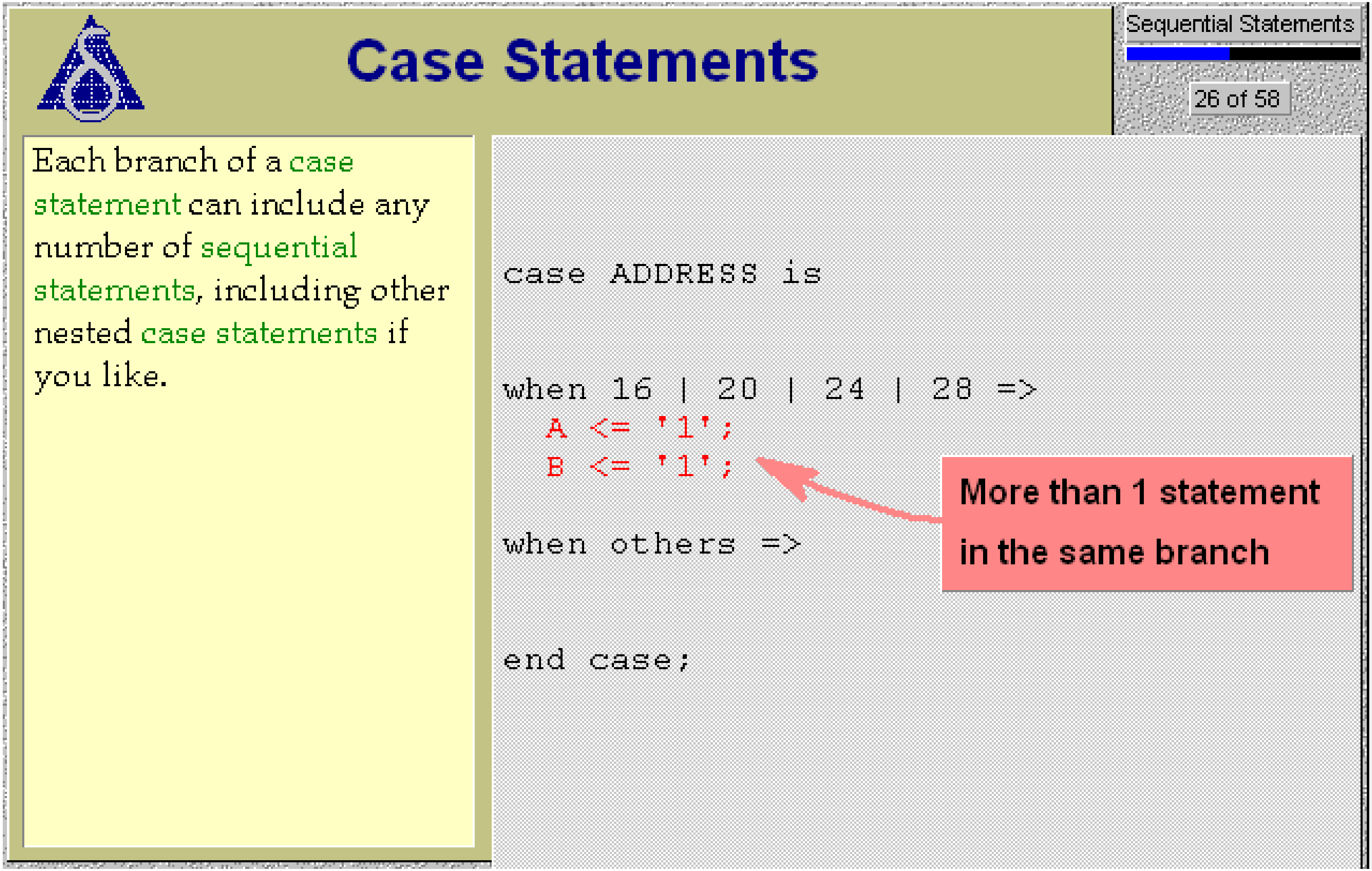


Case Statements

It is possible to cover several cases in the same **when** branch by separating the values with vertical bars. In this example the branch is executed if ADDRESS is 16, 20, 24 or 28.

The "|" is different from the **or** operator, which would have tried to **or** the 4 values together, then compare ADDRESS with the result.

```
case ADDRESS is


when 16 | 20 | 24 | 28 =>
   A <= '1';
   B <= '1';

when others =>


end case;
```

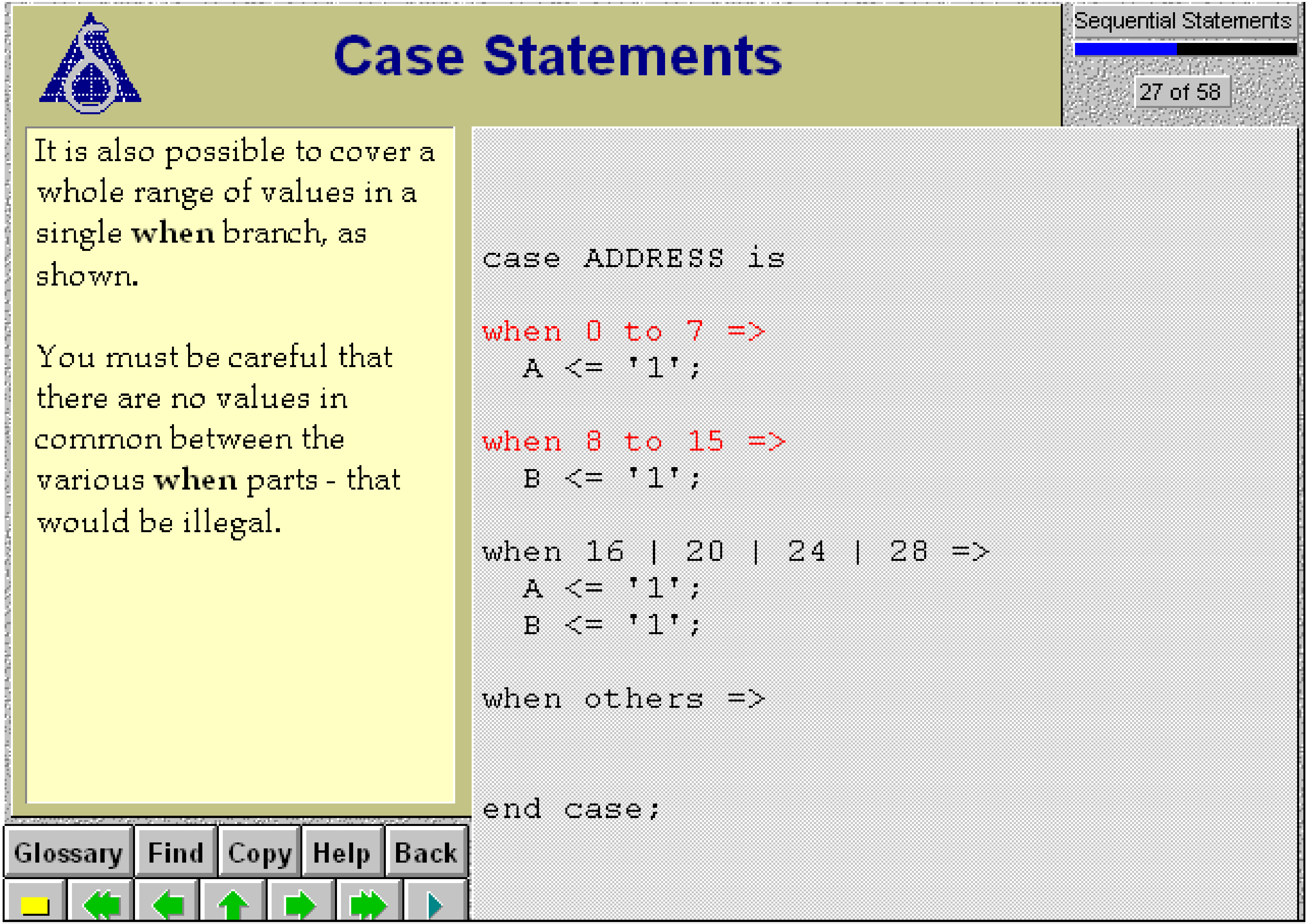


Case Statements

Each branch of a case statement can include any number of sequential statements, including other nested case statements if you like.

```
case ADDRESS is

when 16 | 20 | 24 | 28 =>
  A <= '1';
  B <= '1';

when others =>

end case;
```

More than 1 statement in the same branch

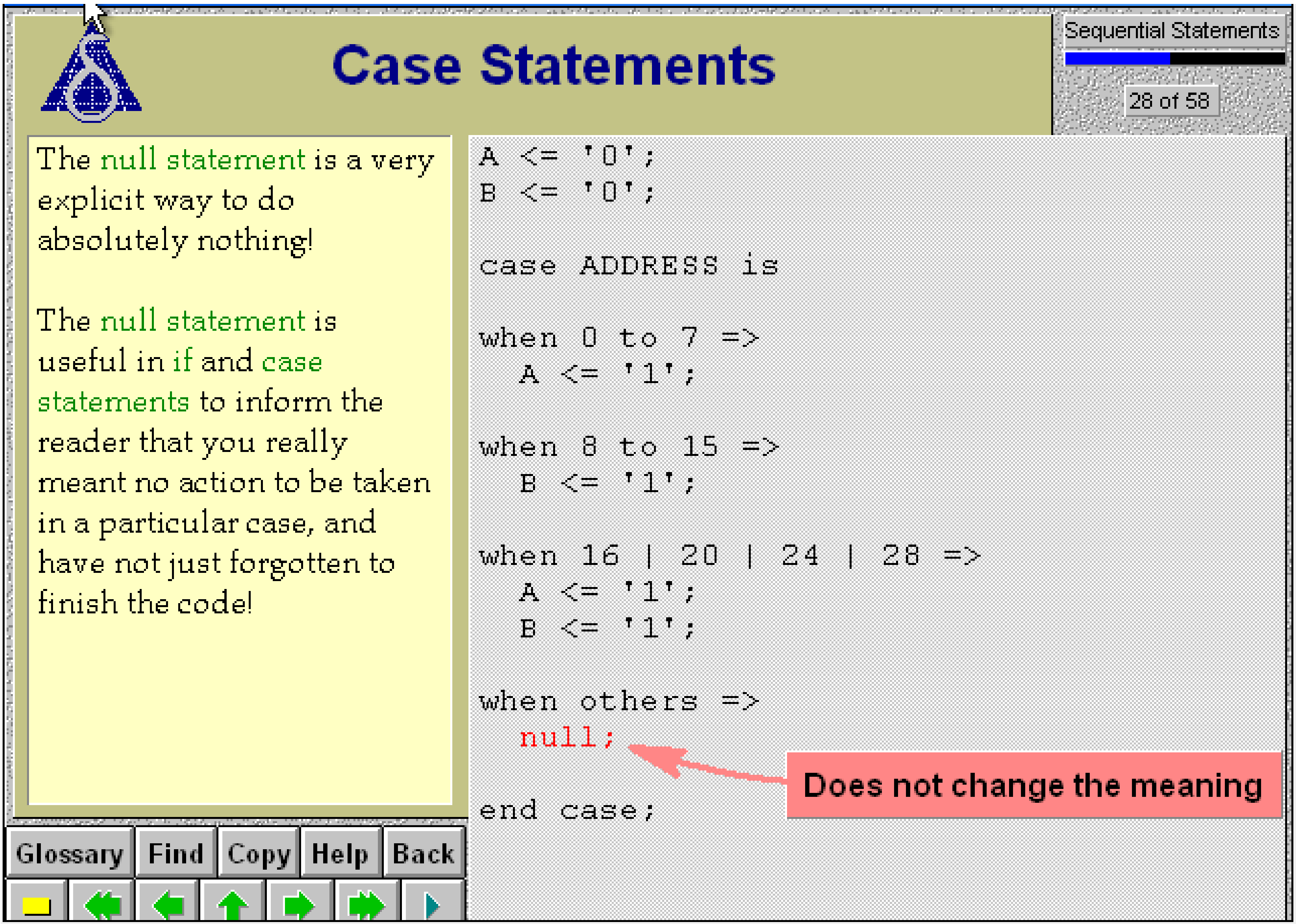


Case Statements

It is also possible to cover a whole range of values in a single **when** branch, as shown.

You must be careful that there are no values in common between the various **when** parts - that would be illegal.

```
case ADDRESS is

when 0 to 7 =>
   A <= '1';

when 8 to 15 =>
   B <= '1';

when 16 | 20 | 24 | 28 =>
   A <= '1';
   B <= '1';

when others =>


end case;
```



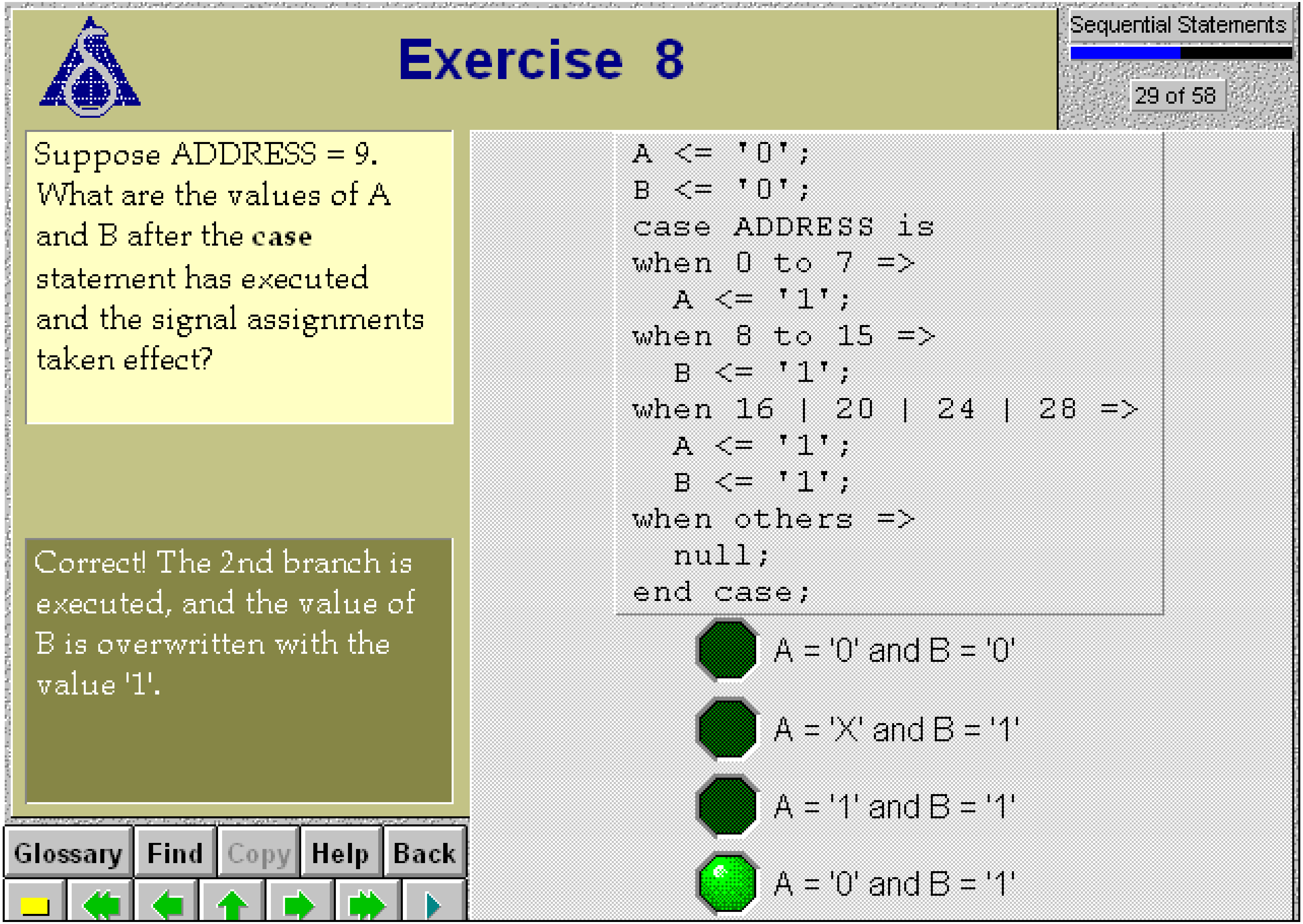
Case Statements

The null statement is a very explicit way to do absolutely nothing!

The null statement is useful in if and case statements to inform the reader that you really meant no action to be taken in a particular case, and have not just forgotten to finish the code!

```
A <= '0';
B <= '0';

case ADDRESS is

when 0 to 7 =>
   A <= '1';

when 8 to 15 =>
   B <= '1';

when 16 | 20 | 24 | 28 =>
   A <= '1';
   B <= '1';

when others =>
   null;

end case;
```

Does not change the meaning



Exercise 8

Suppose ADDRESS = 9. What are the values of A and B after the **case** statement has executed and the signal assignments taken effect?

```
A <= '0';
B <= '0';
case ADDRESS is
when 0 to 7 =>
   A <= '1';
when 8 to 15 =>
   B <= '1';
when 16 | 20 | 24 | 28 =>
   A <= '1';
   B <= '1';
when others =>
   null;
end case;
```

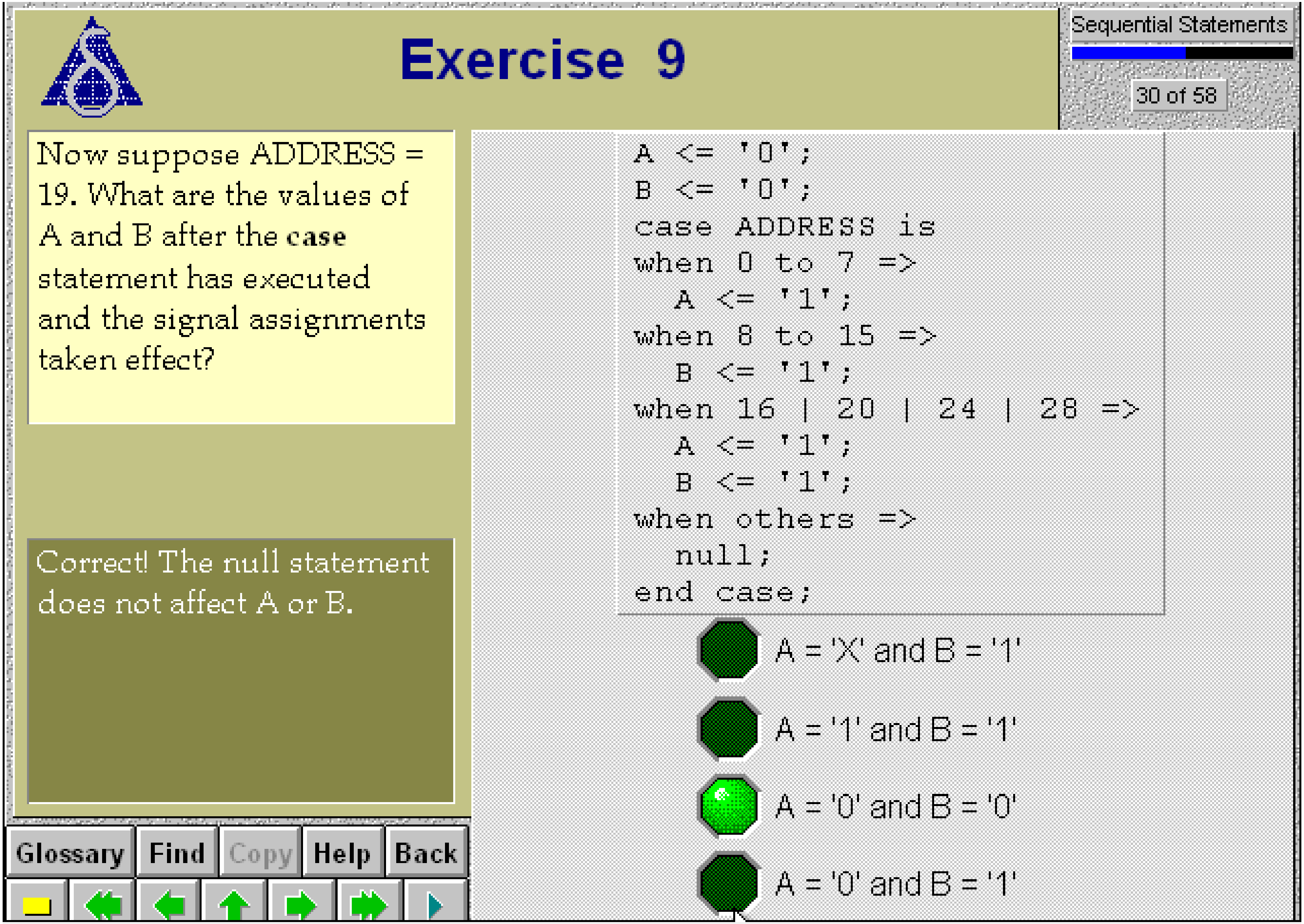
○ A = '0' and B = '0'

○ A = 'X' and B = '1'

○ A = '1' and B = '1'

● A = '0' and B = '1'

Correct! The 2nd branch is executed, and the value of B is overwritten with the value '1'.



Exercise 9

Now suppose ADDRESS = 19. What are the values of A and B after the **case** statement has executed and the signal assignments taken effect?

```
A <= '0';
B <= '0';
case ADDRESS is
when 0 to 7 =>
  A <= '1';
when 8 to 15 =>
  B <= '1';
when 16 | 20 | 24 | 28 =>
  A <= '1';
  B <= '1';
when others =>
  null;
end case;
```

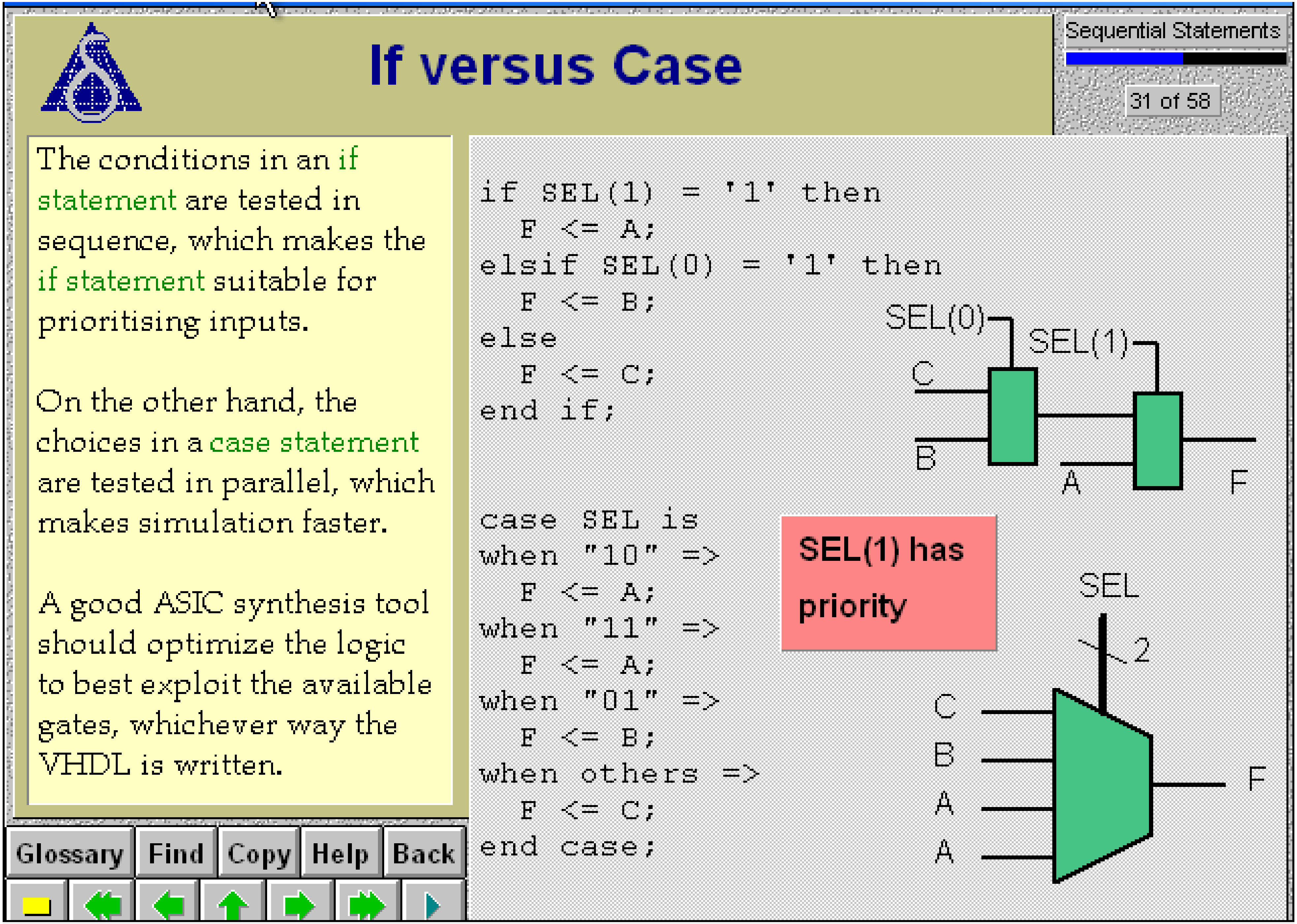
○ A = 'X' and B = '1'

○ A = '1' and B = '1'

● A = '0' and B = '0'

○ A = '0' and B = '1'

Correct! The null statement does not affect A or B.



If versus Case

The conditions in an if statement are tested in sequence, which makes the if statement suitable for prioritising inputs.

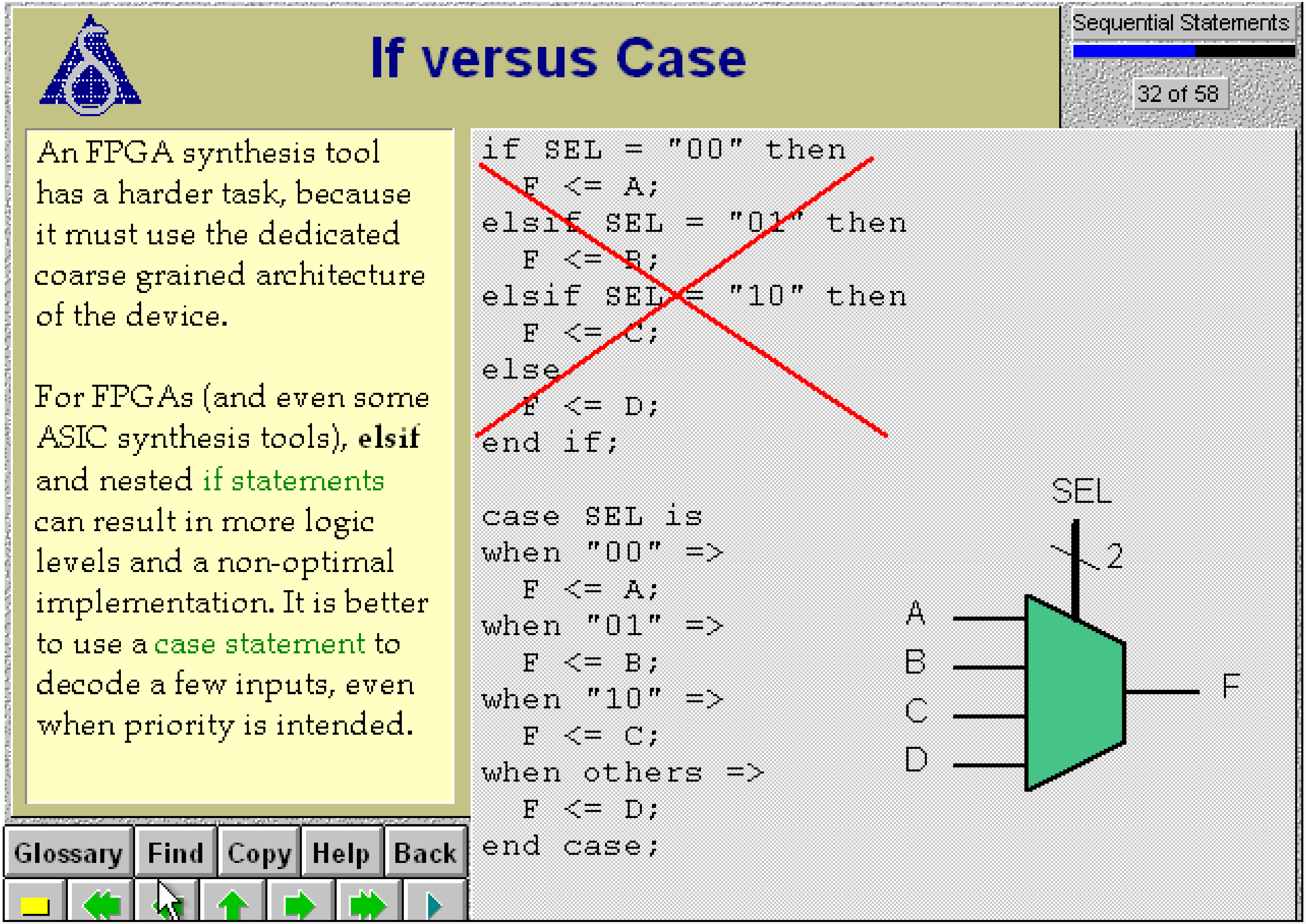
On the other hand, the choices in a case statement are tested in parallel, which makes simulation faster.

A good ASIC synthesis tool should optimize the logic to best exploit the available gates, whichever way the VHDL is written.

```
if SEL(1) = '1' then
  F <= A;
elsif SEL(0) = '1' then
  F <= B;
else
  F <= C;
end if;
```

```
case SEL is
when "10" =>
  F <= A;
when "11" =>
  F <= A;
when "01" =>
  F <= B;
when others =>
  F <= C;
end case;
```

SEL(1) has priority



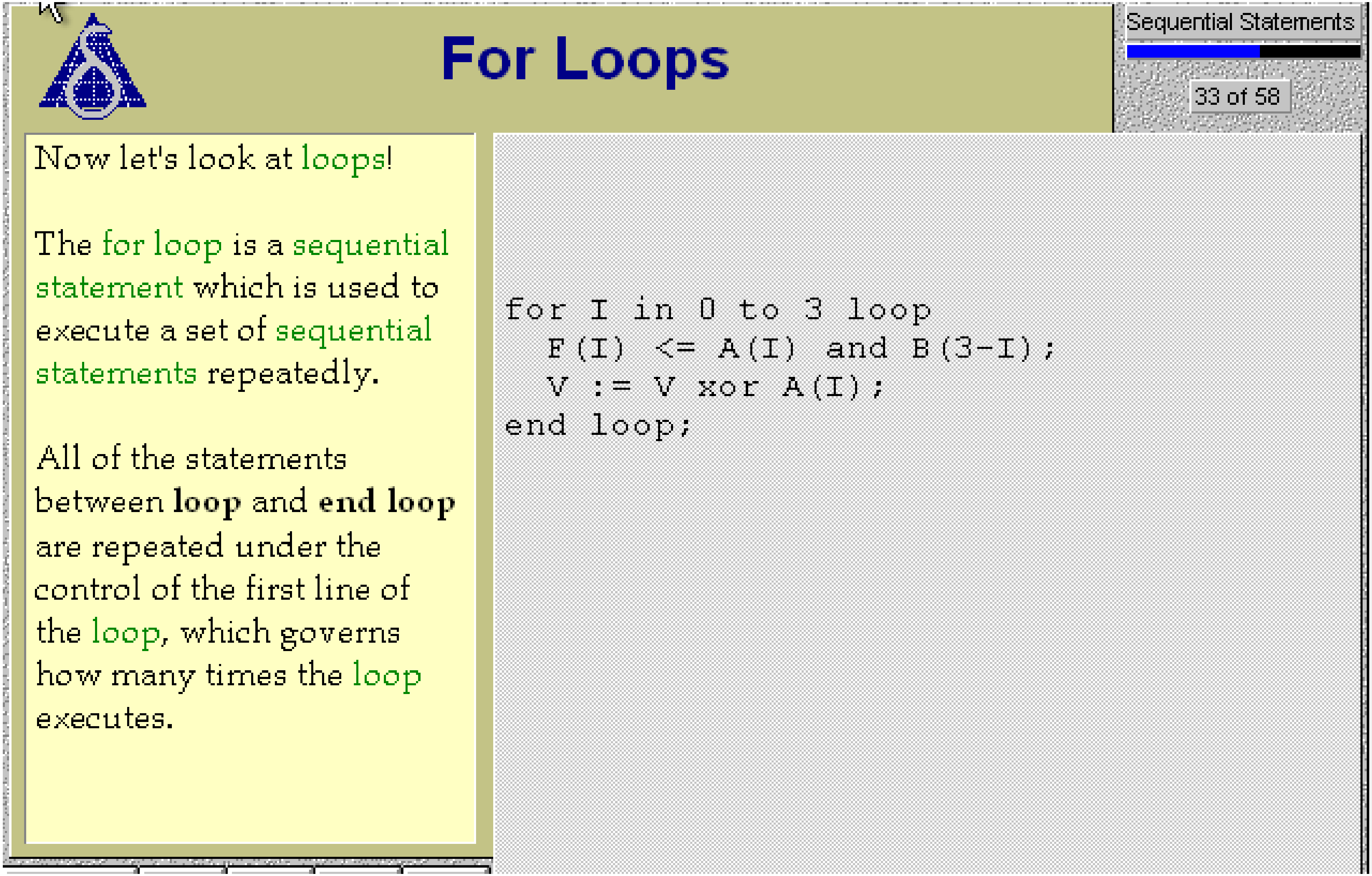
If versus Case

An FPGA synthesis tool has a harder task, because it must use the dedicated coarse grained architecture of the device.

For FPGAs (and even some ASIC synthesis tools), **elsif** and nested if statements can result in more logic levels and a non-optimal implementation. It is better to use a case statement to decode a few inputs, even when priority is intended.

```
if SEL = "00" then
   F <= A;
elsif SEL = "01" then
   F <= B;
elsif SEL = "10" then
   F <= C;
else
   F <= D;
end if;
```

```
case SEL is
when "00" =>
   F <= A;
when "01" =>
   F <= B;
when "10" =>
   F <= C;
when others =>
   F <= D;
end case;
```



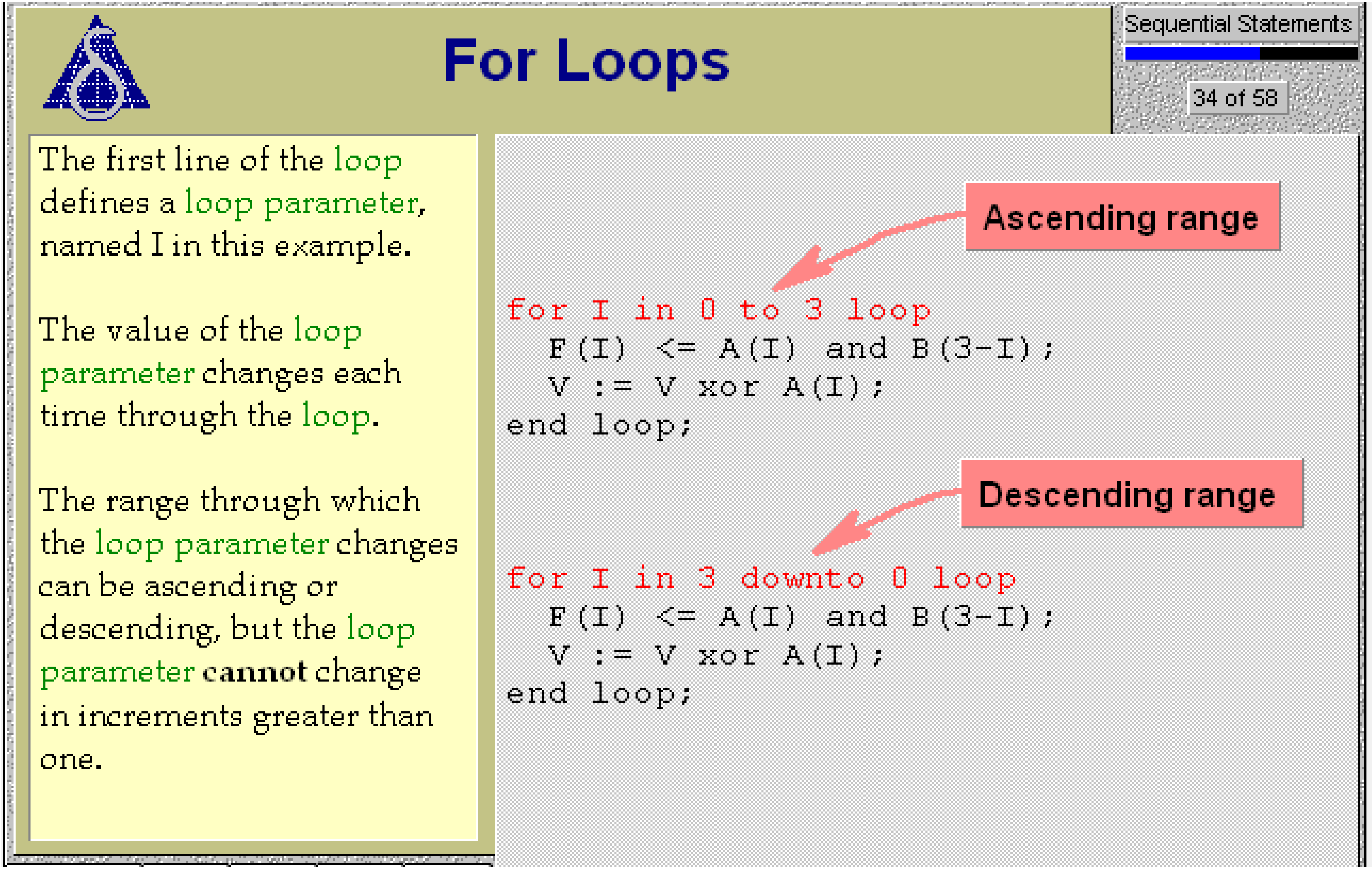
For Loops

Now let's look at loops!

The for loop is a sequential statement which is used to execute a set of sequential statements repeatedly.

All of the statements between **loop** and **end loop** are repeated under the control of the first line of the loop, which governs how many times the loop executes.

```
for I in 0 to 3 loop
  F(I) <= A(I) and B(3-I);
  V := V xor A(I);
end loop;
```



For Loops

The first line of the loop defines a loop parameter, named I in this example.

The value of the loop parameter changes each time through the loop.

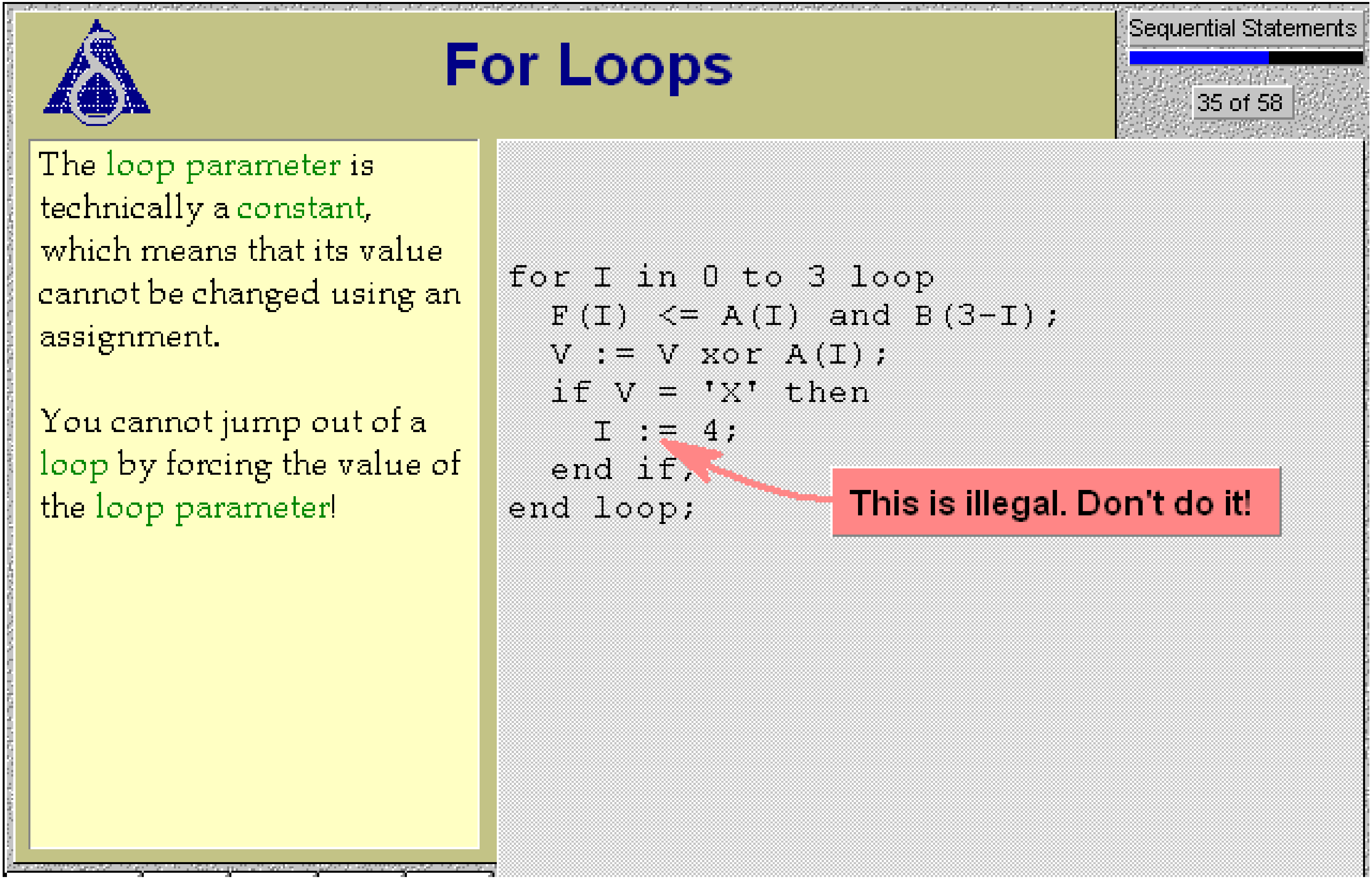
The range through which the loop parameter changes can be ascending or descending, but the loop parameter **cannot** change in increments greater than one.

Ascending range

```
for I in 0 to 3 loop
  F(I) <= A(I) and B(3-I);
  V := V xor A(I);
end loop;
```

Descending range

```
for I in 3 downto 0 loop
  F(I) <= A(I) and B(3-I);
  V := V xor A(I);
end loop;
```



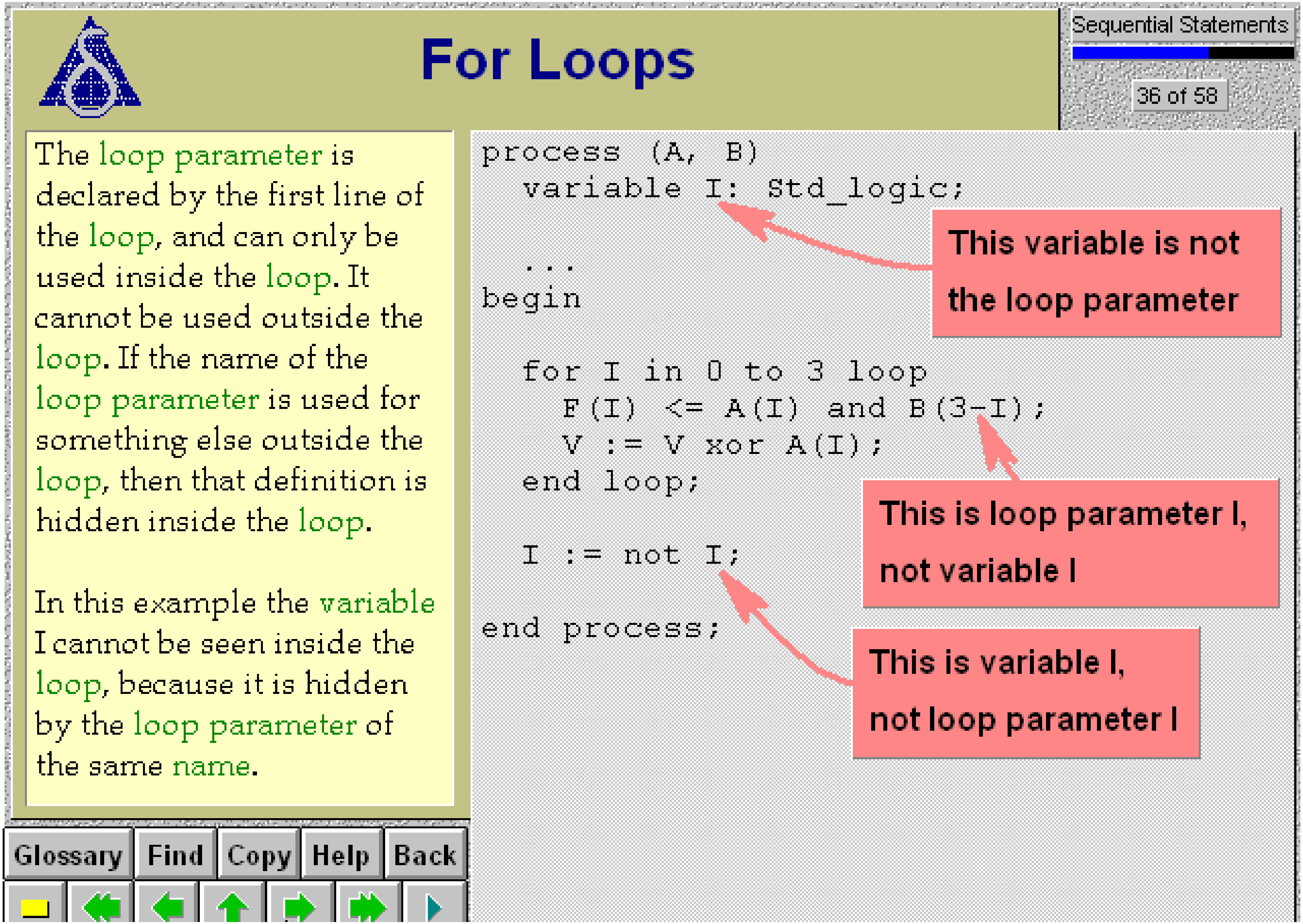
For Loops

The loop parameter is technically a constant, which means that its value cannot be changed using an assignment.

You cannot jump out of a loop by forcing the value of the loop parameter!

```
for I in 0 to 3 loop
  F(I) <= A(I) and B(3-I);
  V := V xor A(I);
  if V = 'X' then
    I := 4;
  end if;
end loop;
```

This is illegal. Don't do it!



For Loops

The loop parameter is declared by the first line of the loop, and can only be used inside the loop. It cannot be used outside the loop. If the name of the loop parameter is used for something else outside the loop, then that definition is hidden inside the loop.

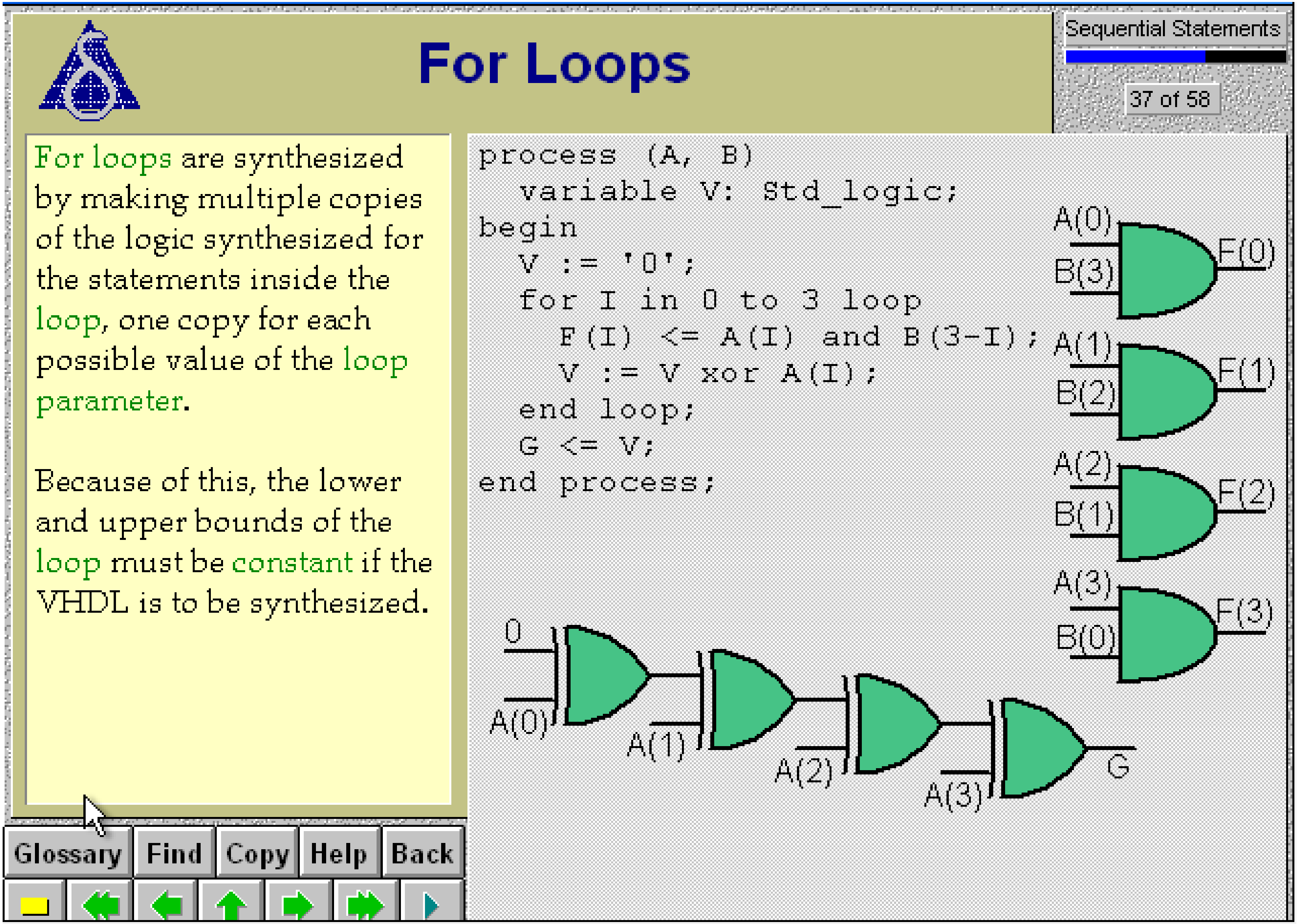
In this example the variable I cannot be seen inside the loop, because it is hidden by the loop parameter of the same name.

```
process (A, B)
  variable I: Std_logic;

  ...
begin

  for I in 0 to 3 loop
    F(I) <= A(I) and B(3-I);
    V := V xor A(I);
  end loop;

  I := not I;

end process;
```

This variable is not the loop parameter

This is loop parameter I, not variable I

This is variable I, not loop parameter I

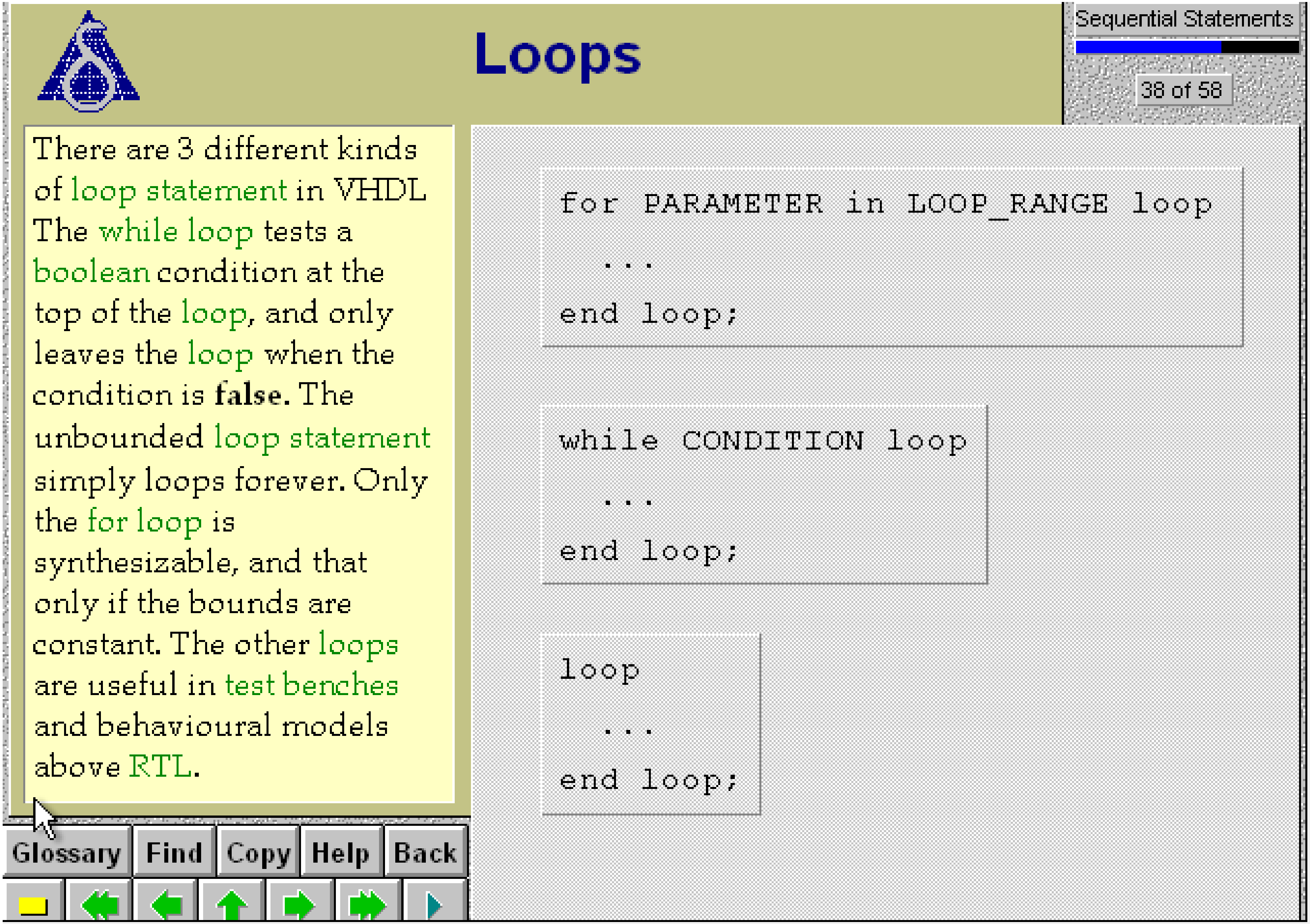


For Loops

For loops are synthesized by making multiple copies of the logic synthesized for the statements inside the loop, one copy for each possible value of the loop parameter.

Because of this, the lower and upper bounds of the loop must be constant if the VHDL is to be synthesized.

```
process (A, B)
  variable V: Std_logic;
begin
  V := '0';
  for I in 0 to 3 loop
    F(I) <= A(I) and B(3-I);
    V := V xor A(I);
  end loop;
  G <= V;
end process;
```



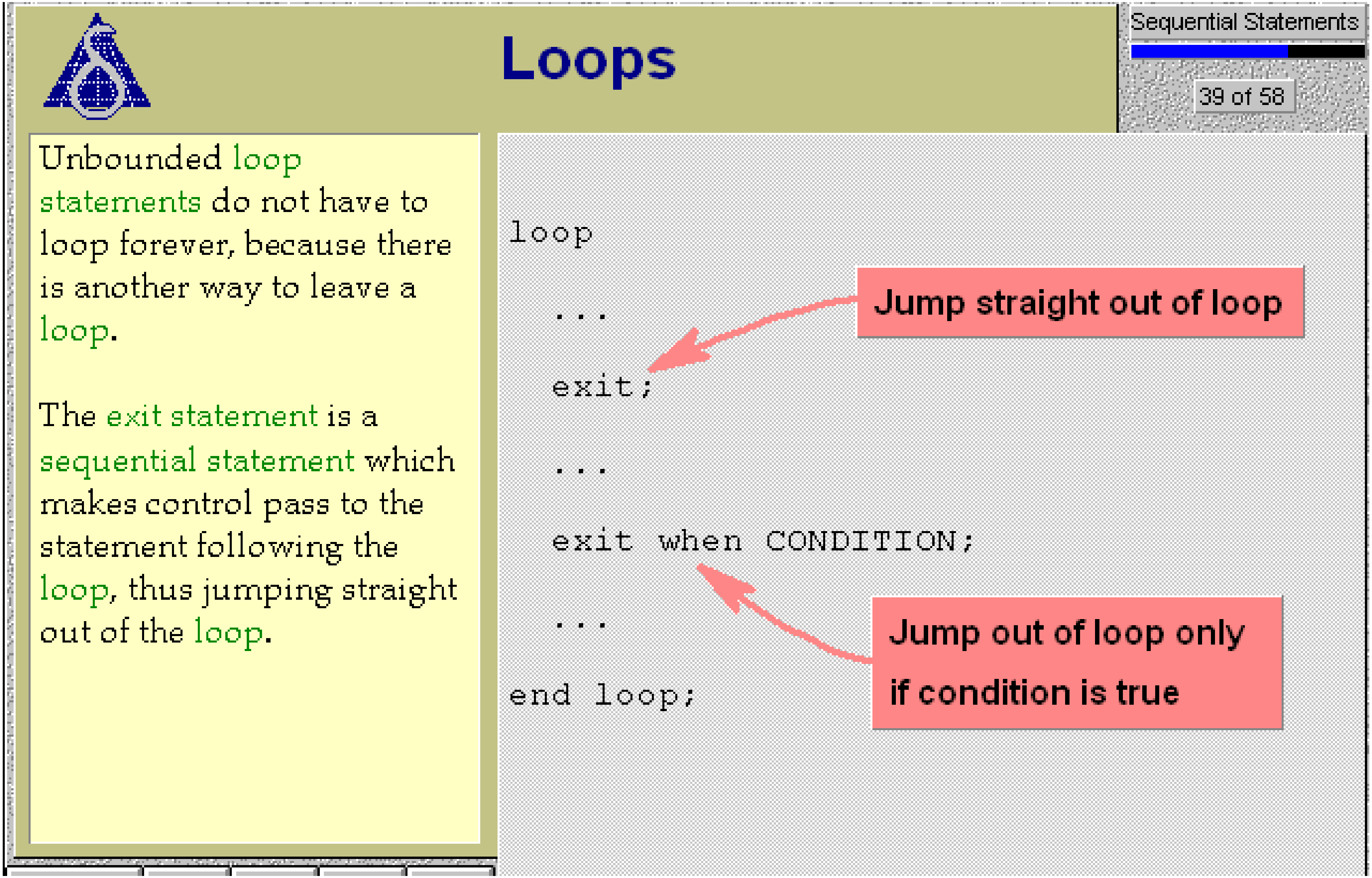
Loops

There are 3 different kinds of loop statement in VHDL The while loop tests a boolean condition at the top of the loop, and only leaves the loop when the condition is **false.** The unbounded loop statement simply loops forever. Only the for loop is synthesizable, and that only if the bounds are constant. The other loops are useful in test benches and behavioural models above RTL.

```
for PARAMETER in LOOP_RANGE loop
  ...
end loop;
```

```
while CONDITION loop
  ...
end loop;
```

```
loop
  ...
end loop;
```



Loops

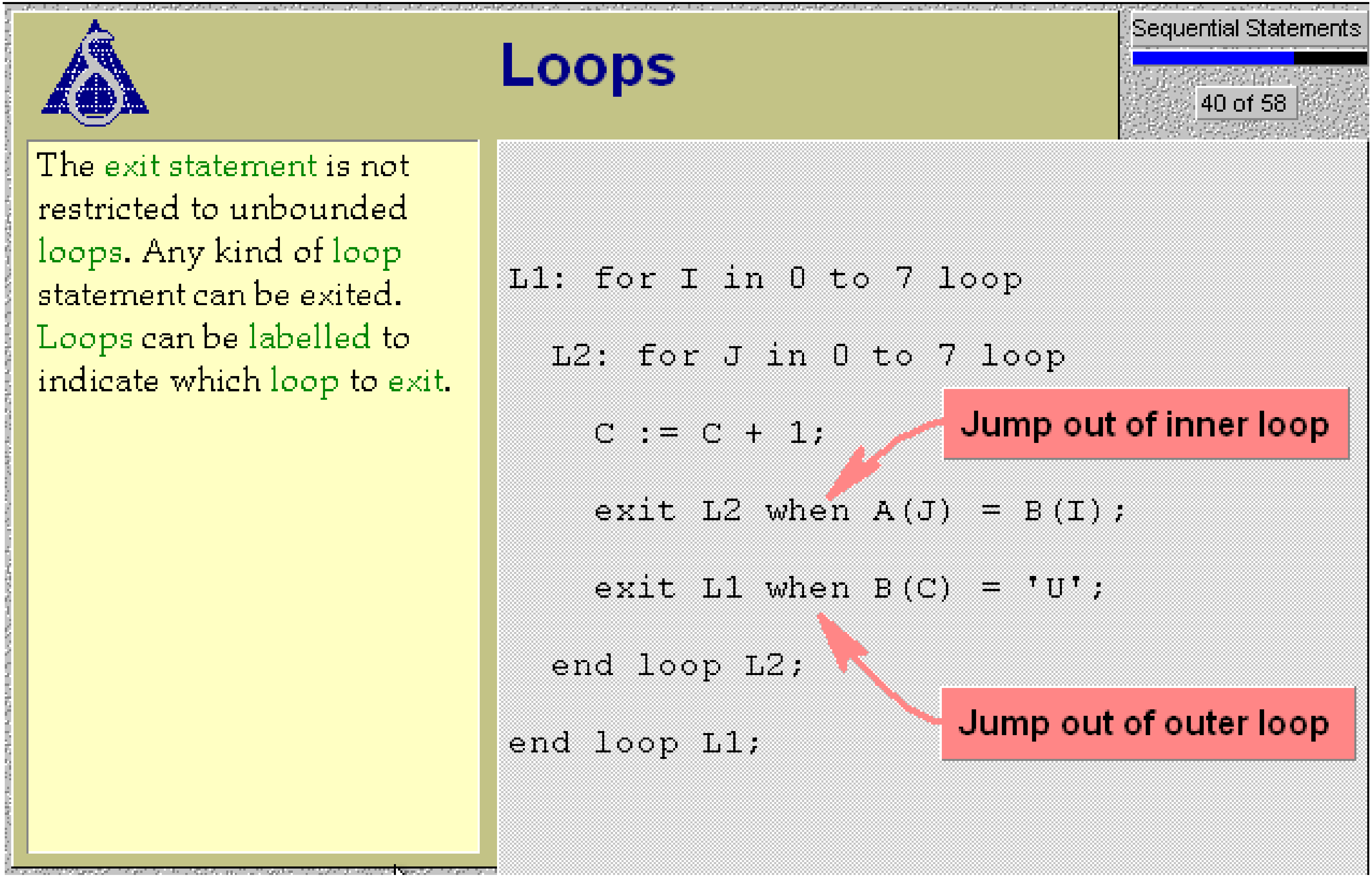
Unbounded loop statements do not have to loop forever, because there is another way to leave a loop.

The exit statement is a sequential statement which makes control pass to the statement following the loop, thus jumping straight out of the loop.

```
loop
   ...
   exit;
   ...
   exit when CONDITION;
   ...
end loop;
```

Jump straight out of loop

Jump out of loop only if condition is true



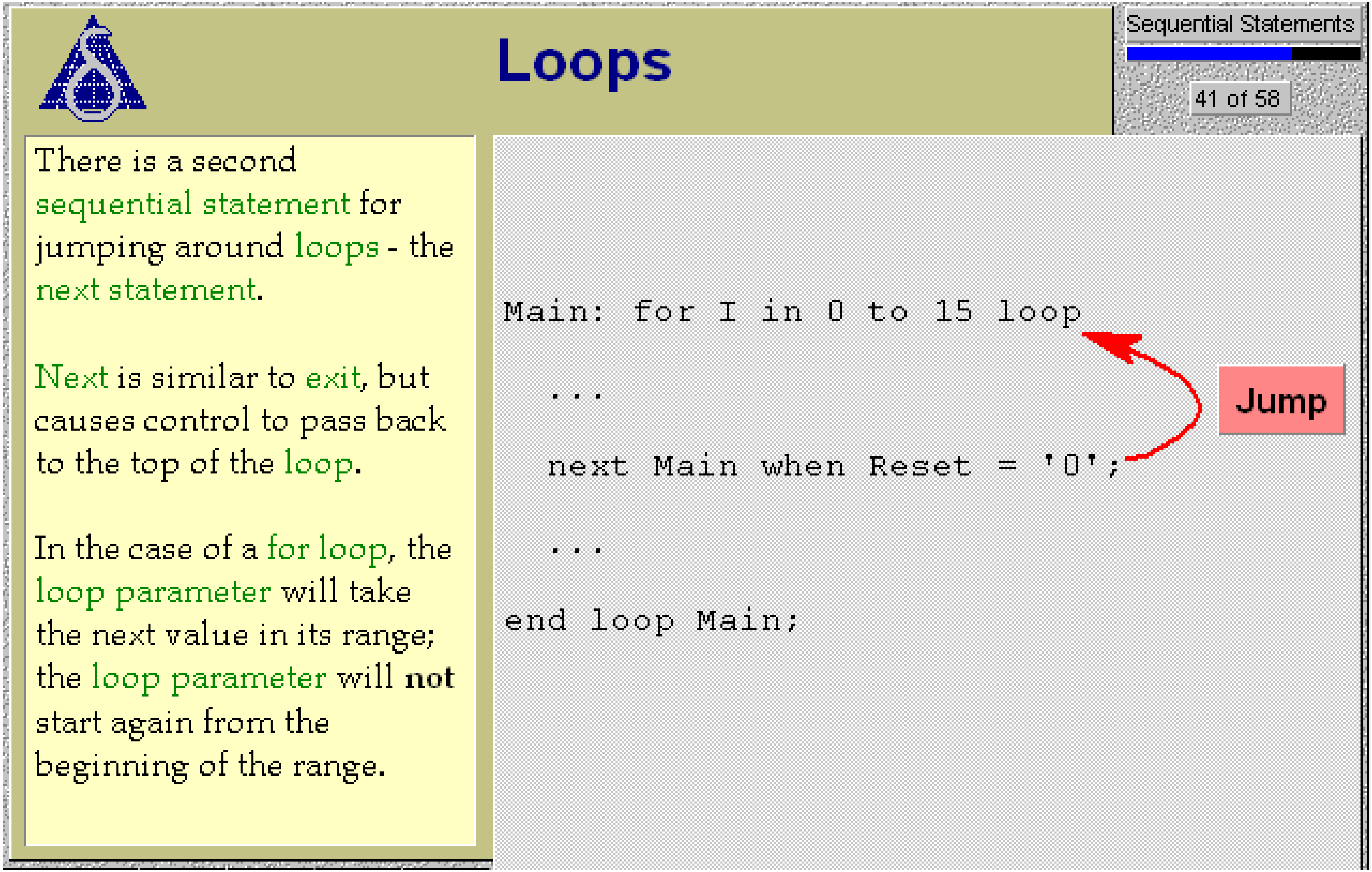
Loops

The exit statement is not restricted to unbounded loops. Any kind of loop statement can be exited. Loops can be labelled to indicate which loop to exit.

```
L1: for I in 0 to 7 loop
  L2: for J in 0 to 7 loop
    C := C + 1;
    exit L2 when A(J) = B(I);
    exit L1 when B(C) = 'U';
  end loop L2;
end loop L1;
```

Jump out of inner loop

Jump out of outer loop



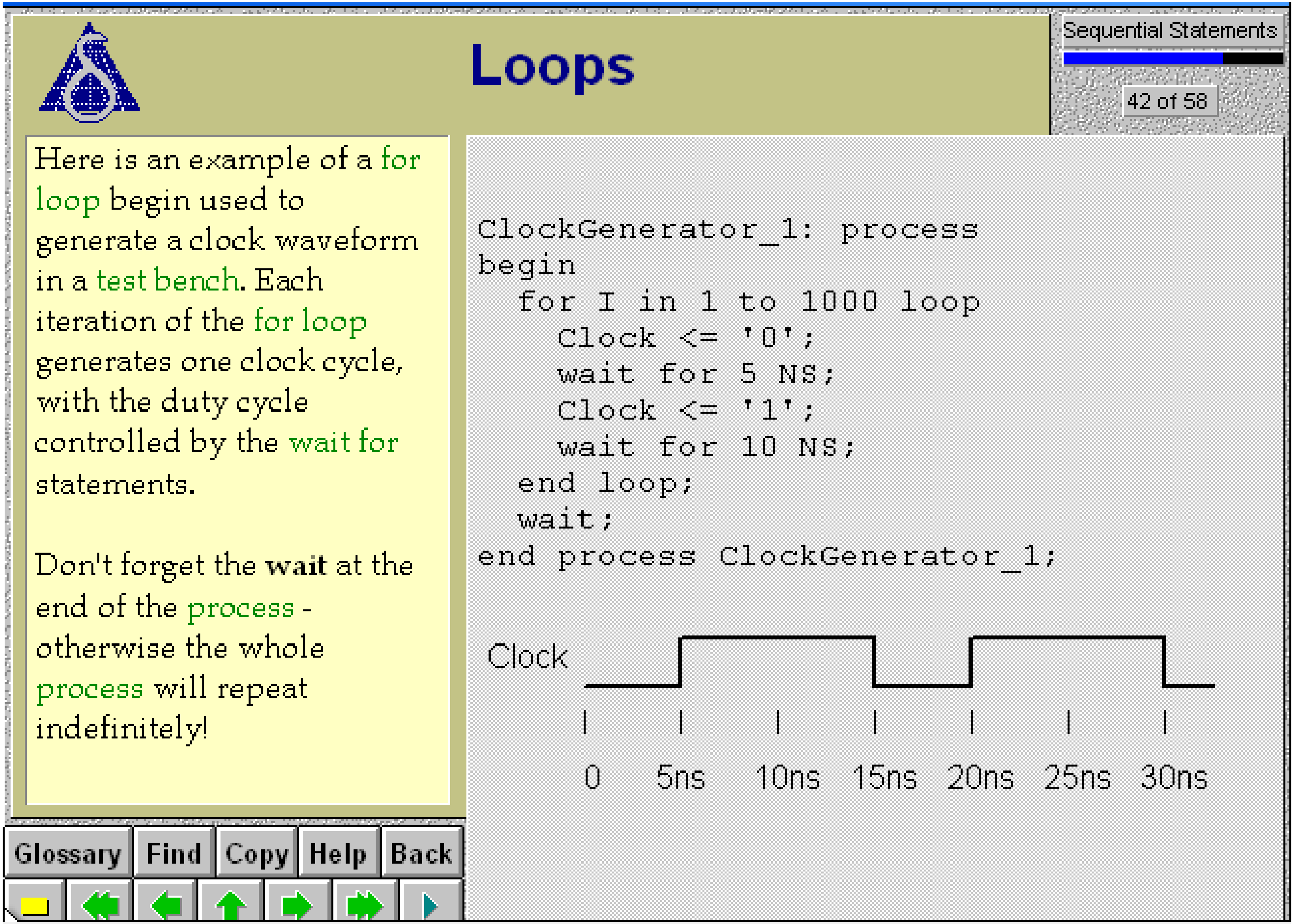
Loops

There is a second sequential statement for jumping around loops - the next statement.

Next is similar to exit, but causes control to pass back to the top of the loop.

In the case of a for loop, the loop parameter will take the next value in its range; the loop parameter will **not** start again from the beginning of the range.

```
Main: for I in 0 to 15 loop
   ...
   next Main when Reset = '0';
   ...
end loop Main;
```

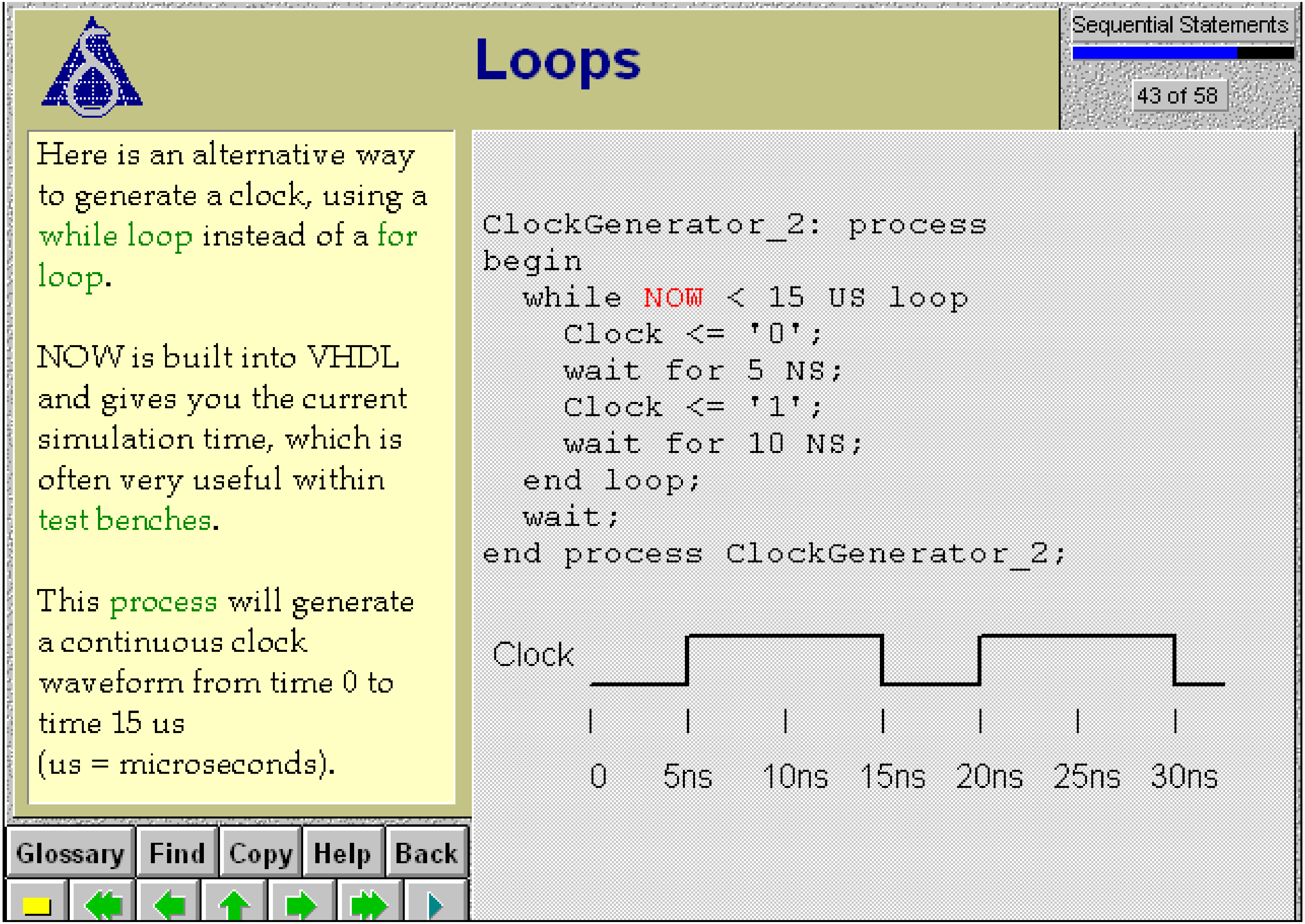


Loops

Here is an example of a for loop begin used to generate a clock waveform in a test bench. Each iteration of the for loop generates one clock cycle, with the duty cycle controlled by the wait for statements.

Don't forget the **wait** at the end of the process - otherwise the whole process will repeat indefinitely!

```
ClockGenerator_1: process
begin
  for I in 1 to 1000 loop
    Clock <= '0';
    wait for 5 NS;
    Clock <= '1';
    wait for 10 NS;
  end loop;
  wait;
end process ClockGenerator_1;
```



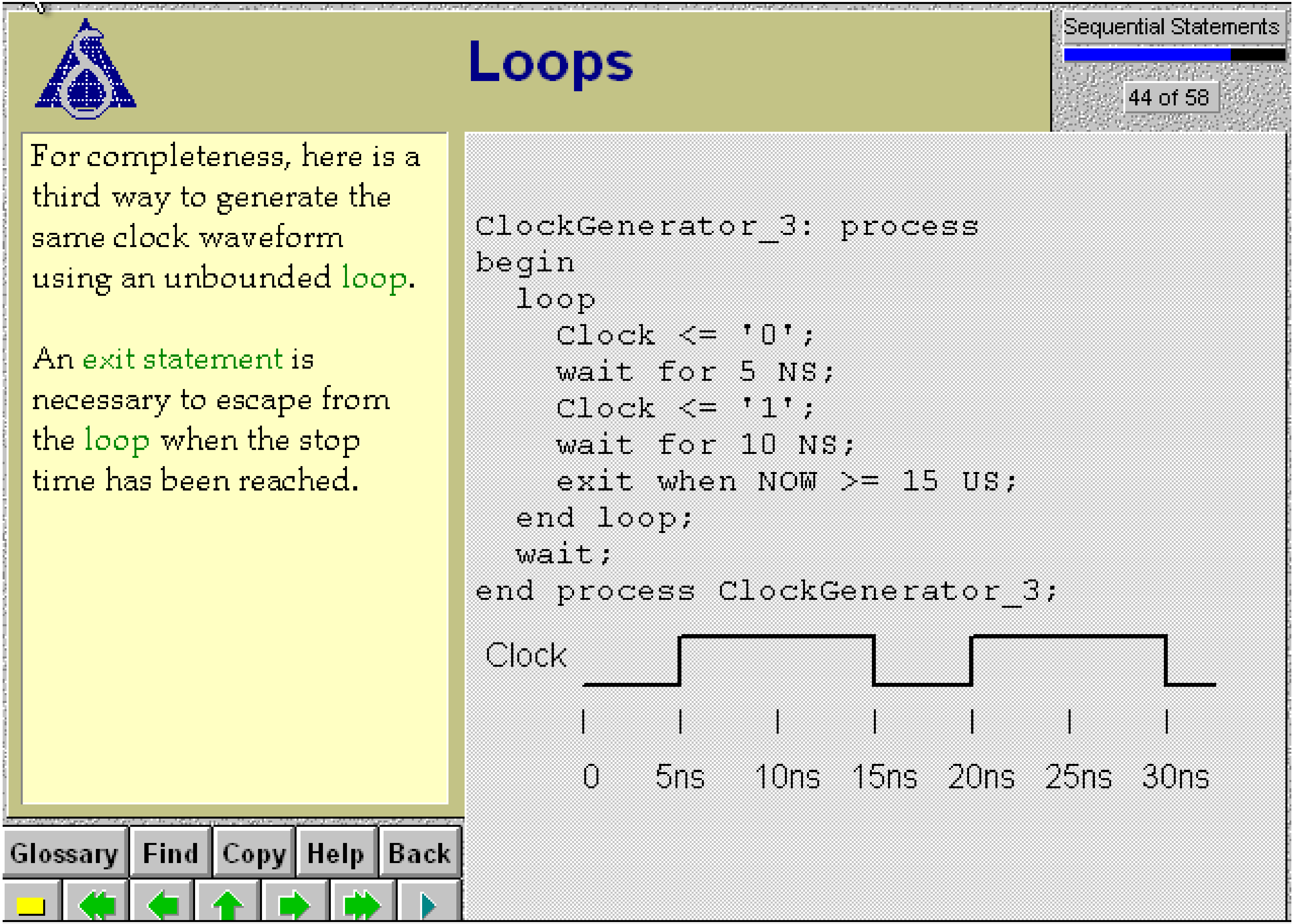
Loops

Here is an alternative way to generate a clock, using a while loop instead of a for loop.

NOW is built into VHDL and gives you the current simulation time, which is often very useful within test benches.

This process will generate a continuous clock waveform from time 0 to time 15 us
(us = microseconds).

```
ClockGenerator_2: process
begin
  while NOW < 15 US loop
    Clock <= '0';
    wait for 5 NS;
    Clock <= '1';
    wait for 10 NS;
  end loop;
  wait;
end process ClockGenerator_2;
```

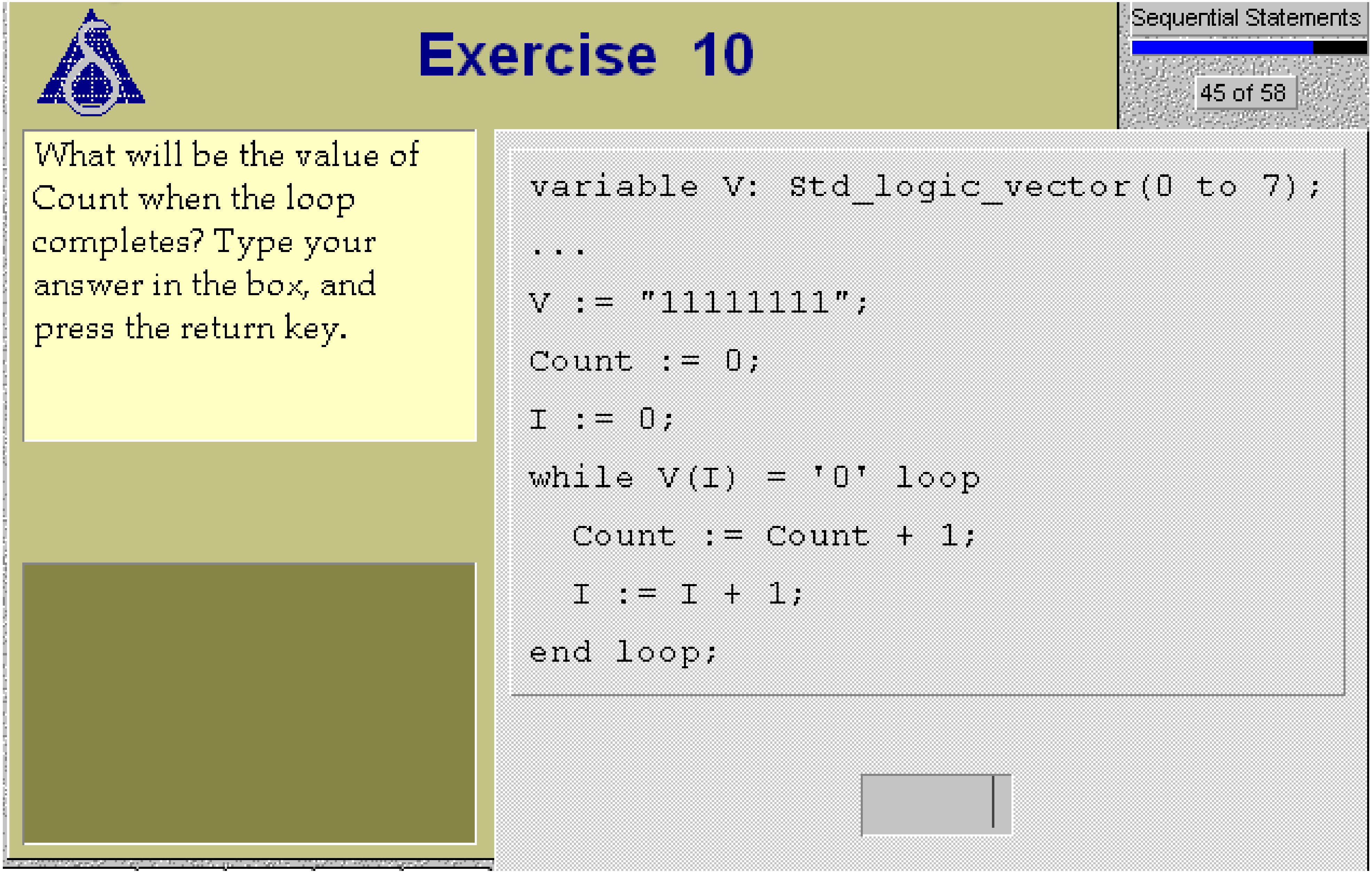


Loops

For completeness, here is a third way to generate the same clock waveform using an unbounded loop.

An exit statement is necessary to escape from the loop when the stop time has been reached.

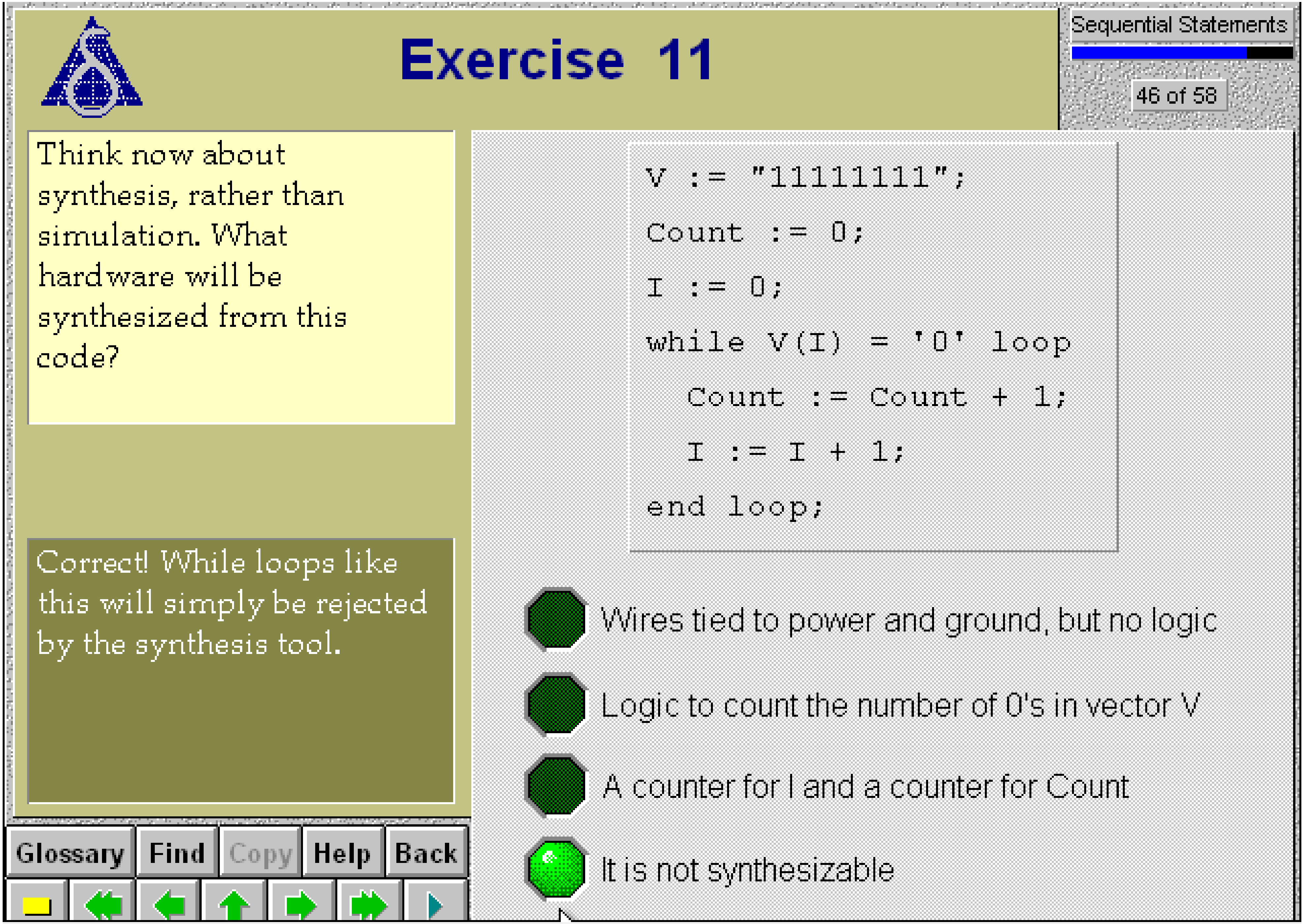
```
ClockGenerator_3: process
begin
  loop
    Clock <= '0';
    wait for 5 NS;
    Clock <= '1';
    wait for 10 NS;
    exit when NOW >= 15 US;
  end loop;
  wait;
end process ClockGenerator_3;
```



Exercise 10

What will be the value of Count when the loop completes? Type your answer in the box, and press the return key.

```
variable V: Std_logic_vector(0 to 7);
...
V := "11111111";
Count := 0;
I := 0;
while V(I) = '0' loop
  Count := Count + 1;
  I := I + 1;
end loop;
```



Exercise 11

Think now about synthesis, rather than simulation. What hardware will be synthesized from this code?

```
V := "11111111";
Count := 0;
I := 0;
while V(I) = '0' loop
  Count := Count + 1;
  I := I + 1;
end loop;
```

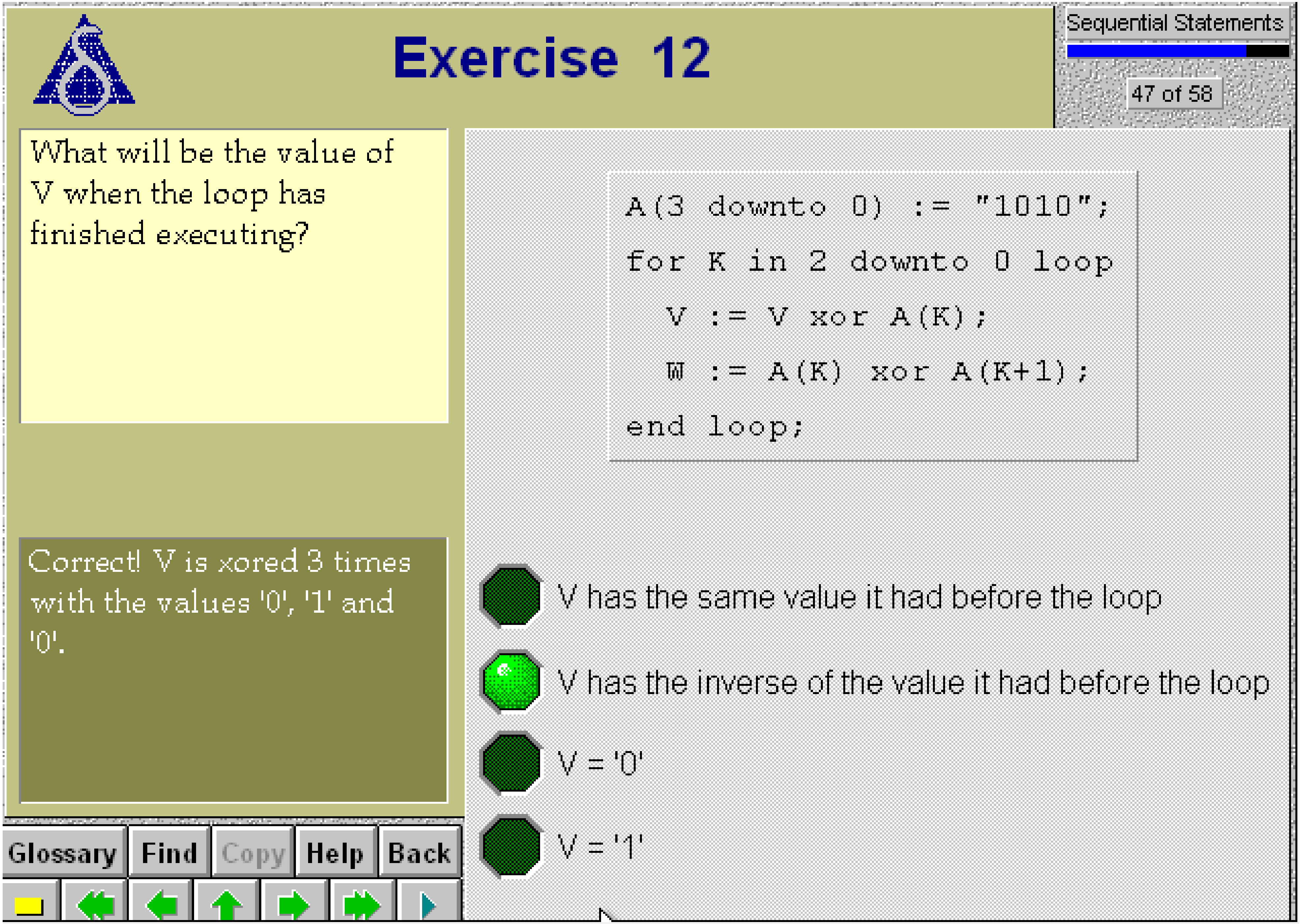
○ Wires tied to power and ground, but no logic

○ Logic to count the number of 0's in vector V

○ A counter for I and a counter for Count

● It is not synthesizable

Correct! While loops like this will simply be rejected by the synthesis tool.



Exercise 12

What will be the value of V when the loop has finished executing?

```
A(3 downto 0) := "1010";
for K in 2 downto 0 loop
   V := V xor A(K);
   W := A(K) xor A(K+1);
end loop;
```

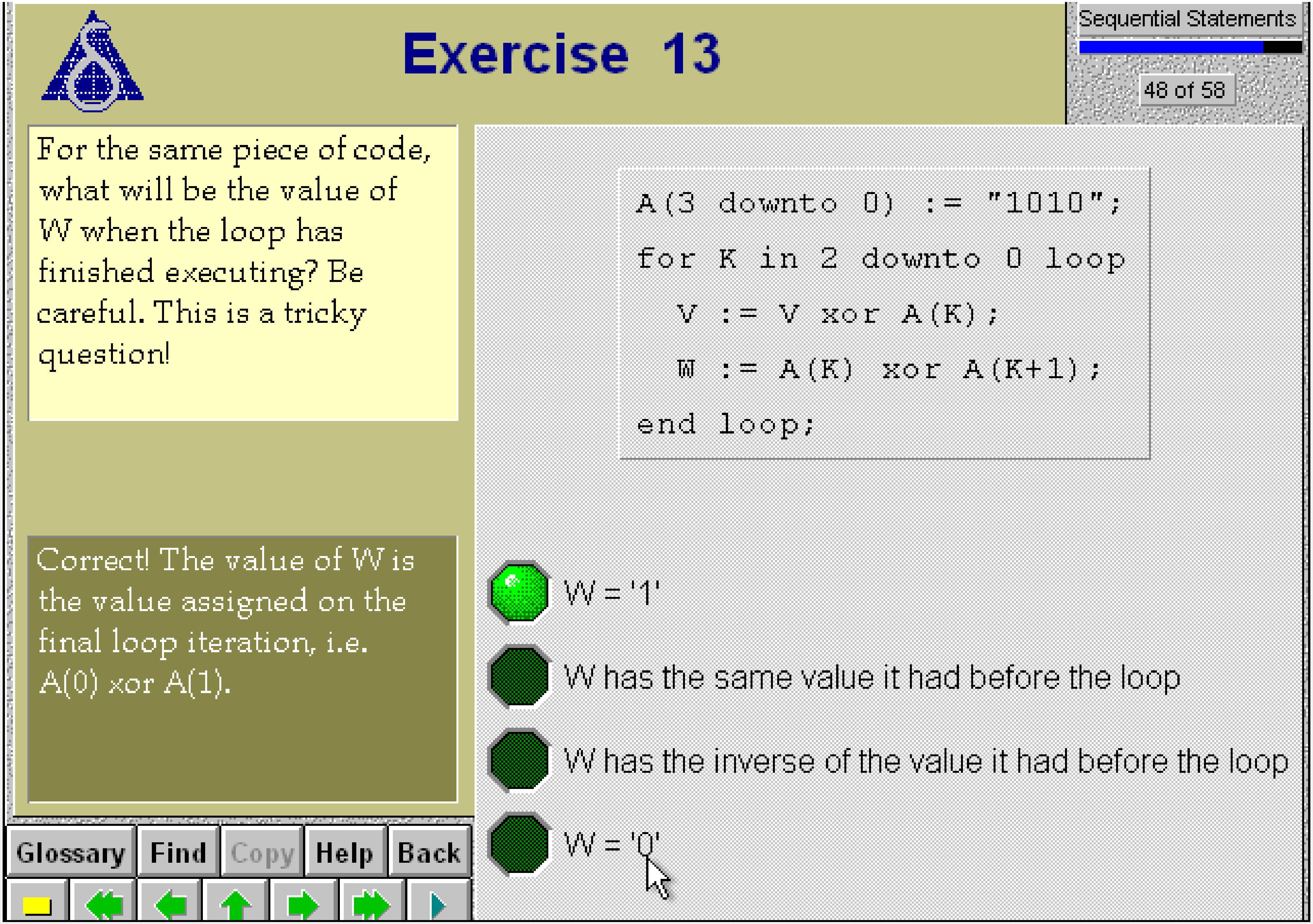
Correct! V is xored 3 times with the values '0', '1' and '0'.

○ V has the same value it had before the loop

● V has the inverse of the value it had before the loop

○ V = '0'

○ V = '1'



Exercise 13

For the same piece of code, what will be the value of W when the loop has finished executing? Be careful. This is a tricky question!

```
A(3 downto 0) := "1010";
for K in 2 downto 0 loop
  V := V xor A(K);
  W := A(K) xor A(K+1);
end loop;
```

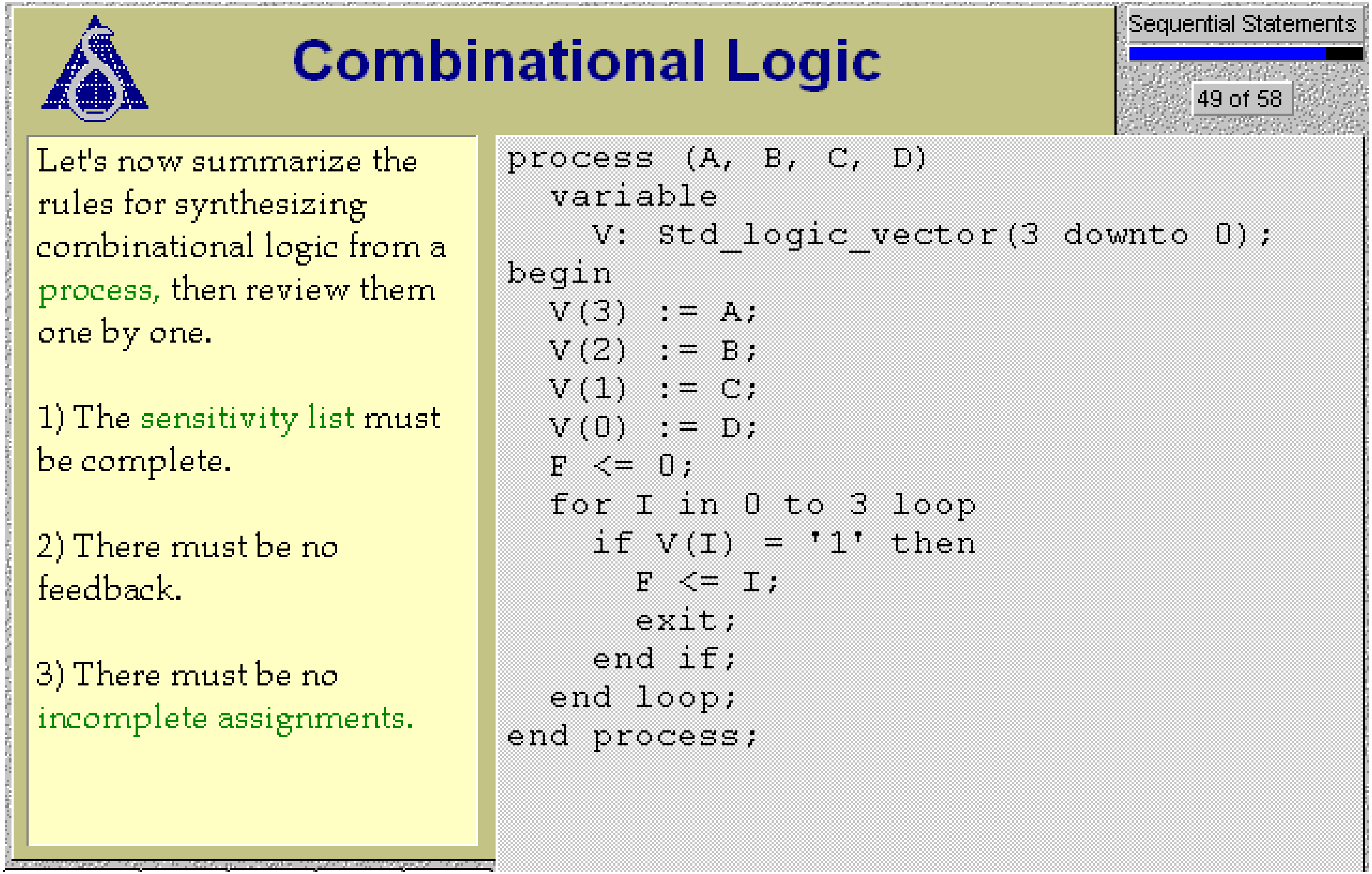
● W = '1'

○ W has the same value it had before the loop

○ W has the inverse of the value it had before the loop

○ W = '0'

Correct! The value of W is the value assigned on the final loop iteration, i.e. A(0) xor A(1).



Combinational Logic

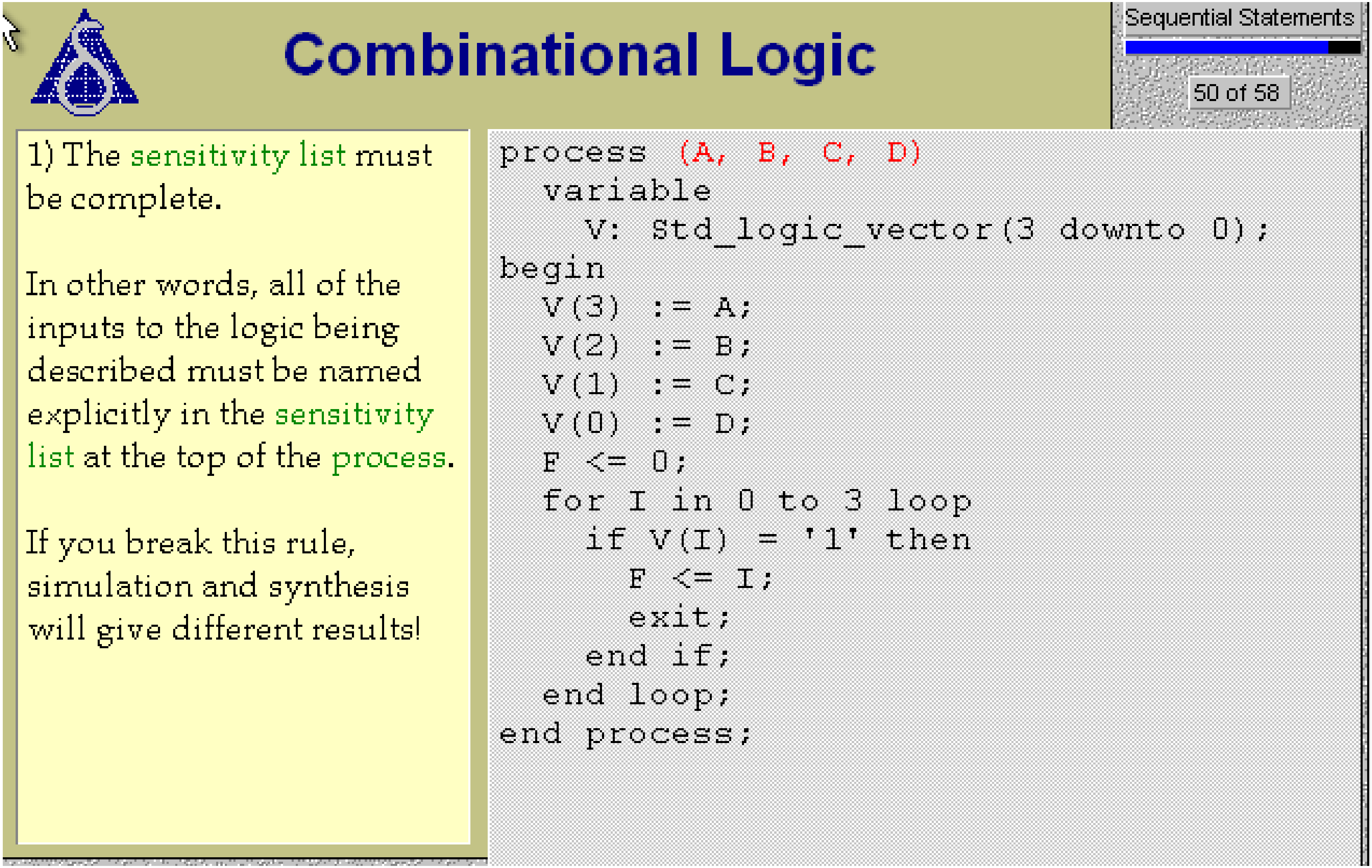
Let's now summarize the rules for synthesizing combinational logic from a process, then review them one by one.

1) The sensitivity list must be complete.

2) There must be no feedback.

3) There must be no incomplete assignments.

```
process (A, B, C, D)
  variable
    V: Std_logic_vector(3 downto 0);
begin
  V(3) := A;
  V(2) := B;
  V(1) := C;
  V(0) := D;
  F <= 0;
  for I in 0 to 3 loop
    if V(I) = '1' then
      F <= I;
      exit;
    end if;
  end loop;
end process;
```



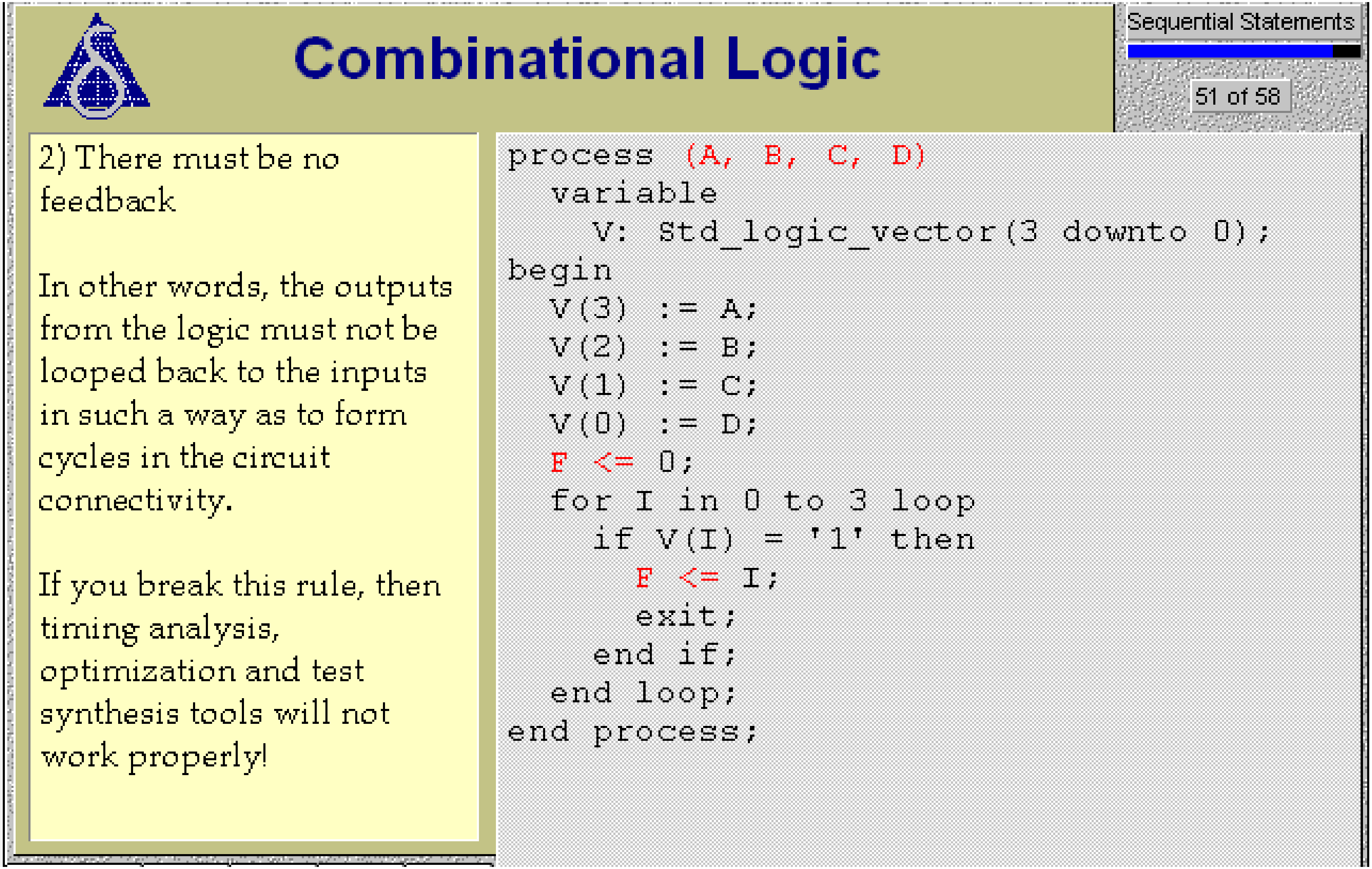
Combinational Logic

1) The sensitivity list must be complete.

In other words, all of the inputs to the logic being described must be named explicitly in the sensitivity list at the top of the process.

If you break this rule, simulation and synthesis will give different results!

```
process (A, B, C, D)
  variable
    V: Std_logic_vector(3 downto 0);
begin
  V(3) := A;
  V(2) := B;
  V(1) := C;
  V(0) := D;
  F <= 0;
  for I in 0 to 3 loop
    if V(I) = '1' then
      F <= I;
      exit;
    end if;
  end loop;
end process;
```



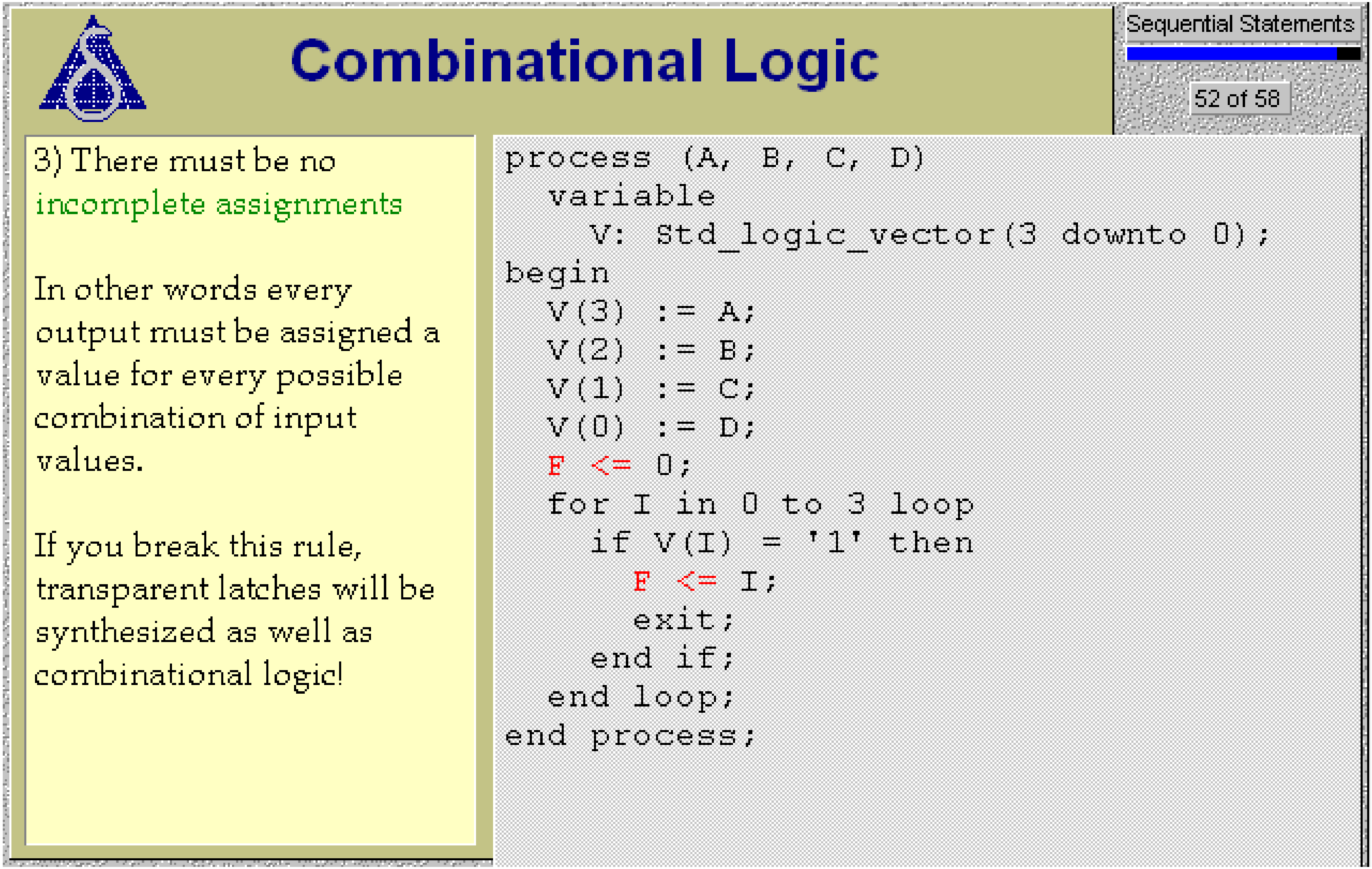
Combinational Logic

2) There must be no feedback

In other words, the outputs from the logic must not be looped back to the inputs in such a way as to form cycles in the circuit connectivity.

If you break this rule, then timing analysis, optimization and test synthesis tools will not work properly!

```
process (A, B, C, D)
  variable
    V: Std_logic_vector(3 downto 0);
begin
  V(3) := A;
  V(2) := B;
  V(1) := C;
  V(0) := D;
  F <= 0;
  for I in 0 to 3 loop
    if V(I) = '1' then
      F <= I;
      exit;
    end if;
  end loop;
end process;
```



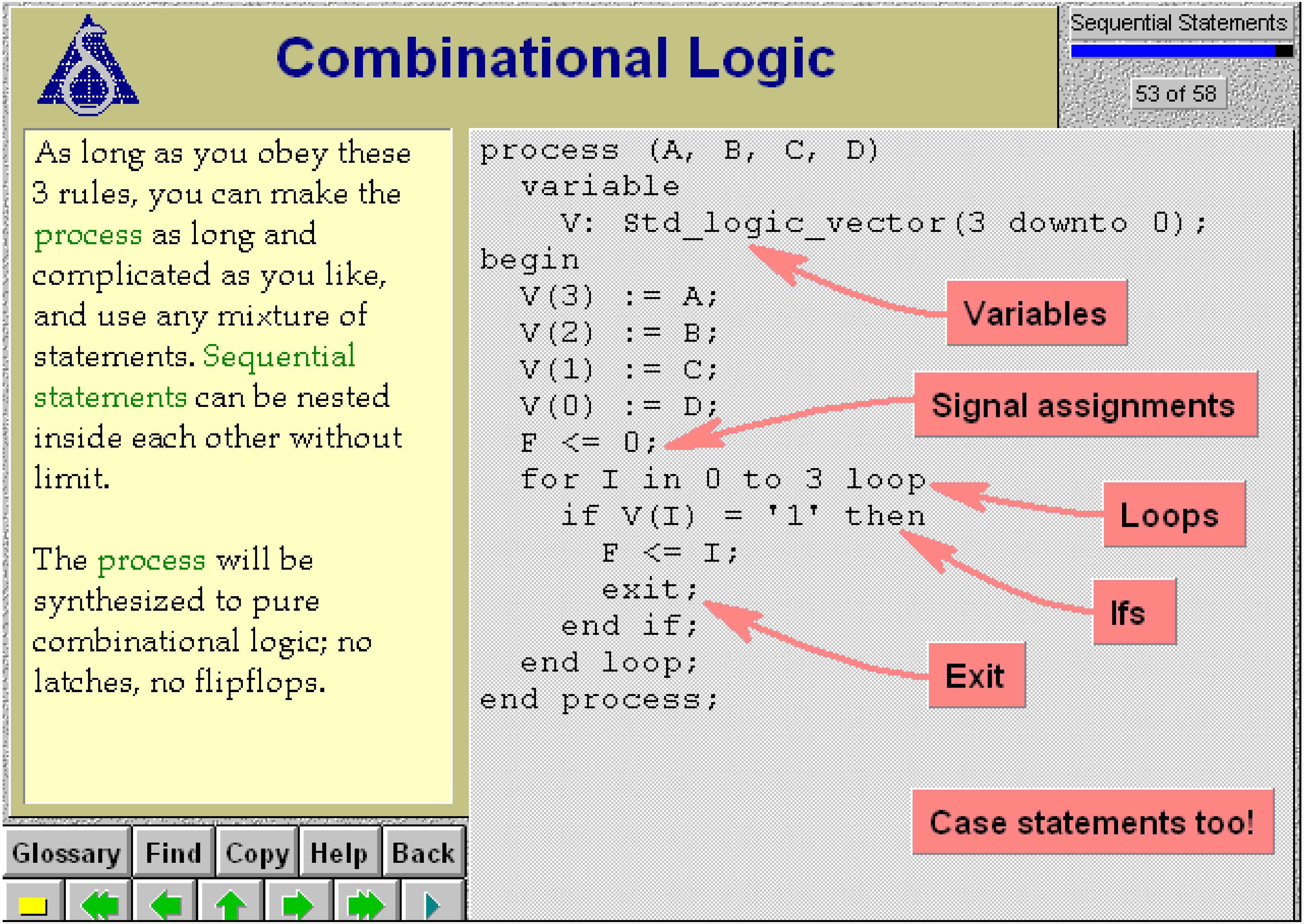
Combinational Logic

3) There must be no incomplete assignments

In other words every output must be assigned a value for every possible combination of input values.

If you break this rule, transparent latches will be synthesized as well as combinational logic!

```
process (A, B, C, D)
   variable
      V: Std_logic_vector(3 downto 0);
begin
   V(3) := A;
   V(2) := B;
   V(1) := C;
   V(0) := D;
   F <= 0;
   for I in 0 to 3 loop
      if V(I) = '1' then
         F <= I;
         exit;
      end if;
   end loop;
end process;
```

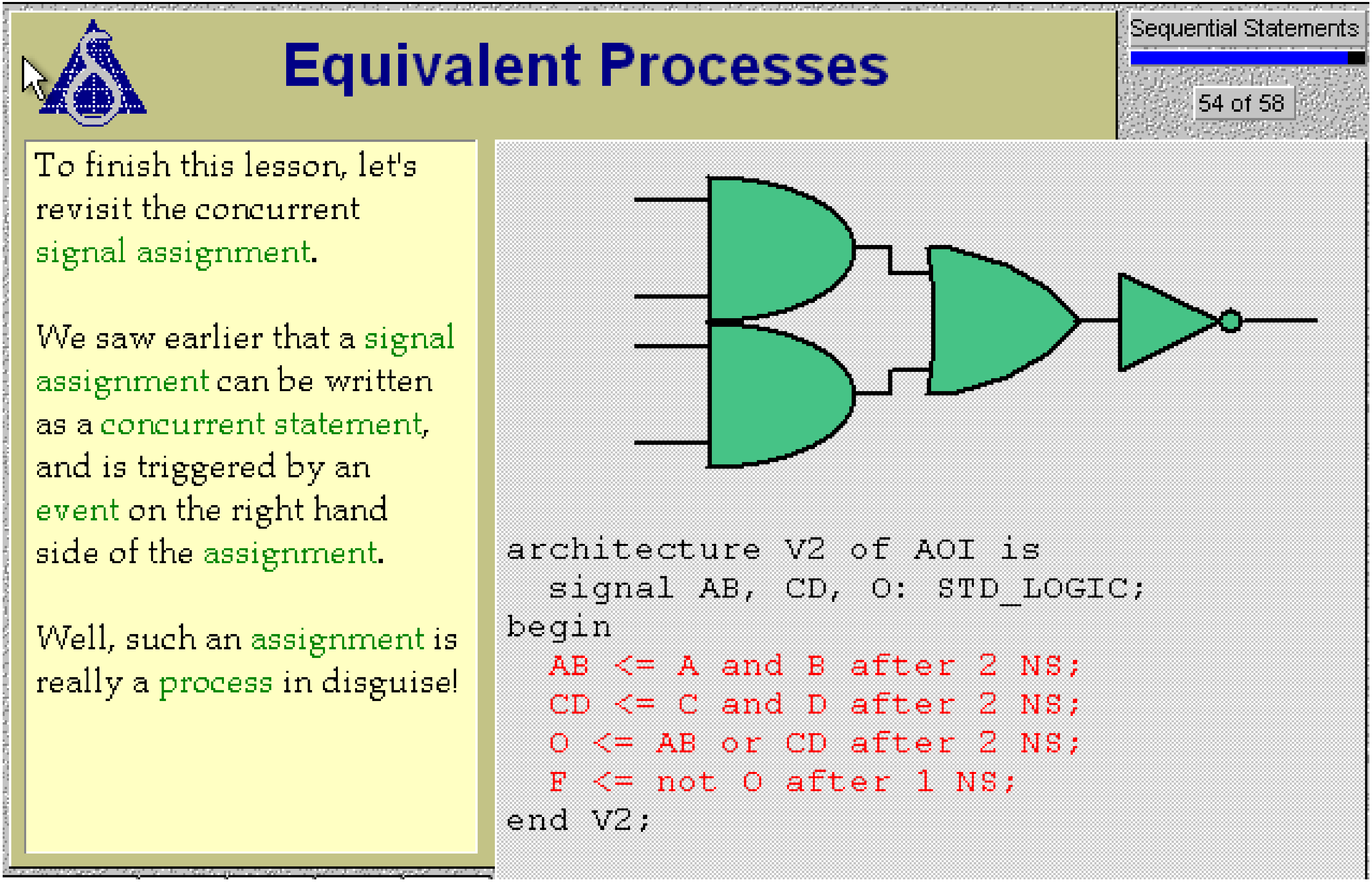


Combinational Logic

As long as you obey these 3 rules, you can make the process as long and complicated as you like, and use any mixture of statements. Sequential statements can be nested inside each other without limit.

The process will be synthesized to pure combinational logic; no latches, no flipflops.

```
process (A, B, C, D)
  variable
    V: Std_logic_vector(3 downto 0);
begin
  V(3) := A;
  V(2) := B;
  V(1) := C;
  V(0) := D;
  F <= 0;
  for I in 0 to 3 loop
    if V(I) = '1' then
      F <= I;
      exit;
    end if;
  end loop;
end process;
```



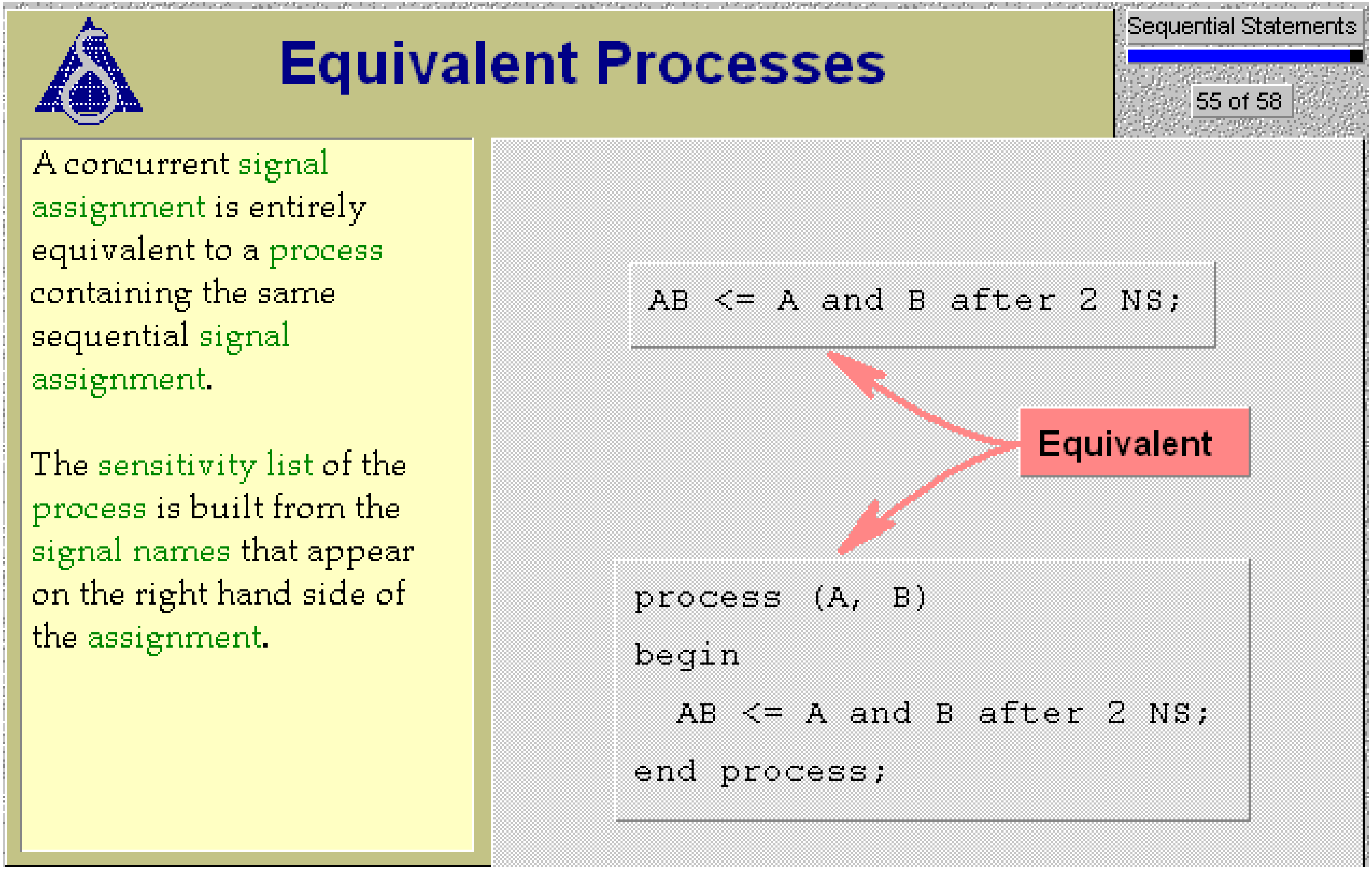
Equivalent Processes

To finish this lesson, let's revisit the concurrent signal assignment.

We saw earlier that a signal assignment can be written as a concurrent statement, and is triggered by an event on the right hand side of the assignment.

Well, such an assignment is really a process in disguise!

```
architecture V2 of AOI is
  signal AB, CD, O: STD_LOGIC;
begin
  AB <= A and B after 2 NS;
  CD <= C and D after 2 NS;
  O <= AB or CD after 2 NS;
  F <= not O after 1 NS;
end V2;
```



Equivalent Processes

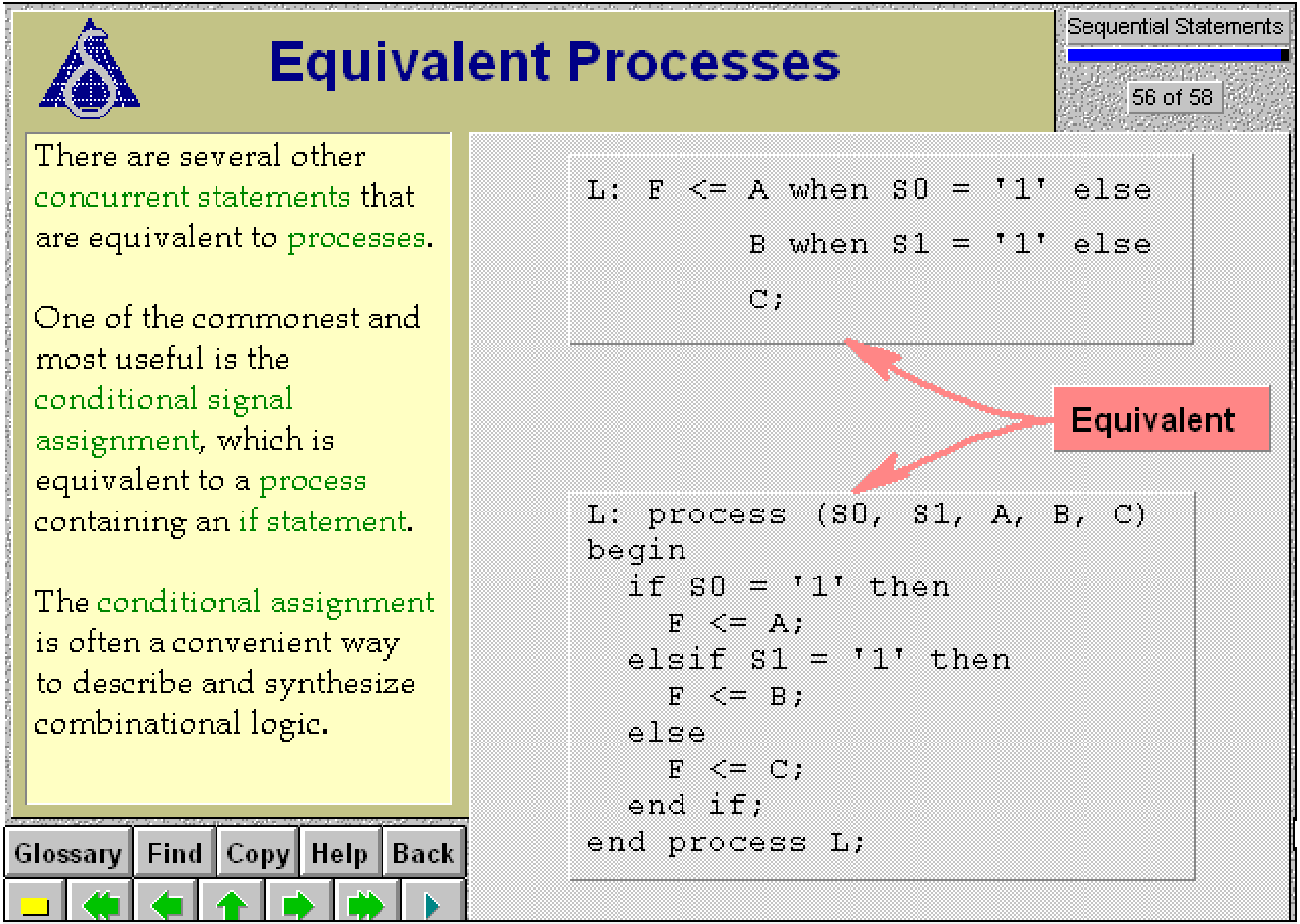
A concurrent signal assignment is entirely equivalent to a process containing the same sequential signal assignment.

The sensitivity list of the process is built from the signal names that appear on the right hand side of the assignment.

```
AB <= A and B after 2 NS;
```

Equivalent

```
process (A, B)
begin
  AB <= A and B after 2 NS;
end process;
```



Equivalent Processes

There are several other concurrent statements that are equivalent to processes.

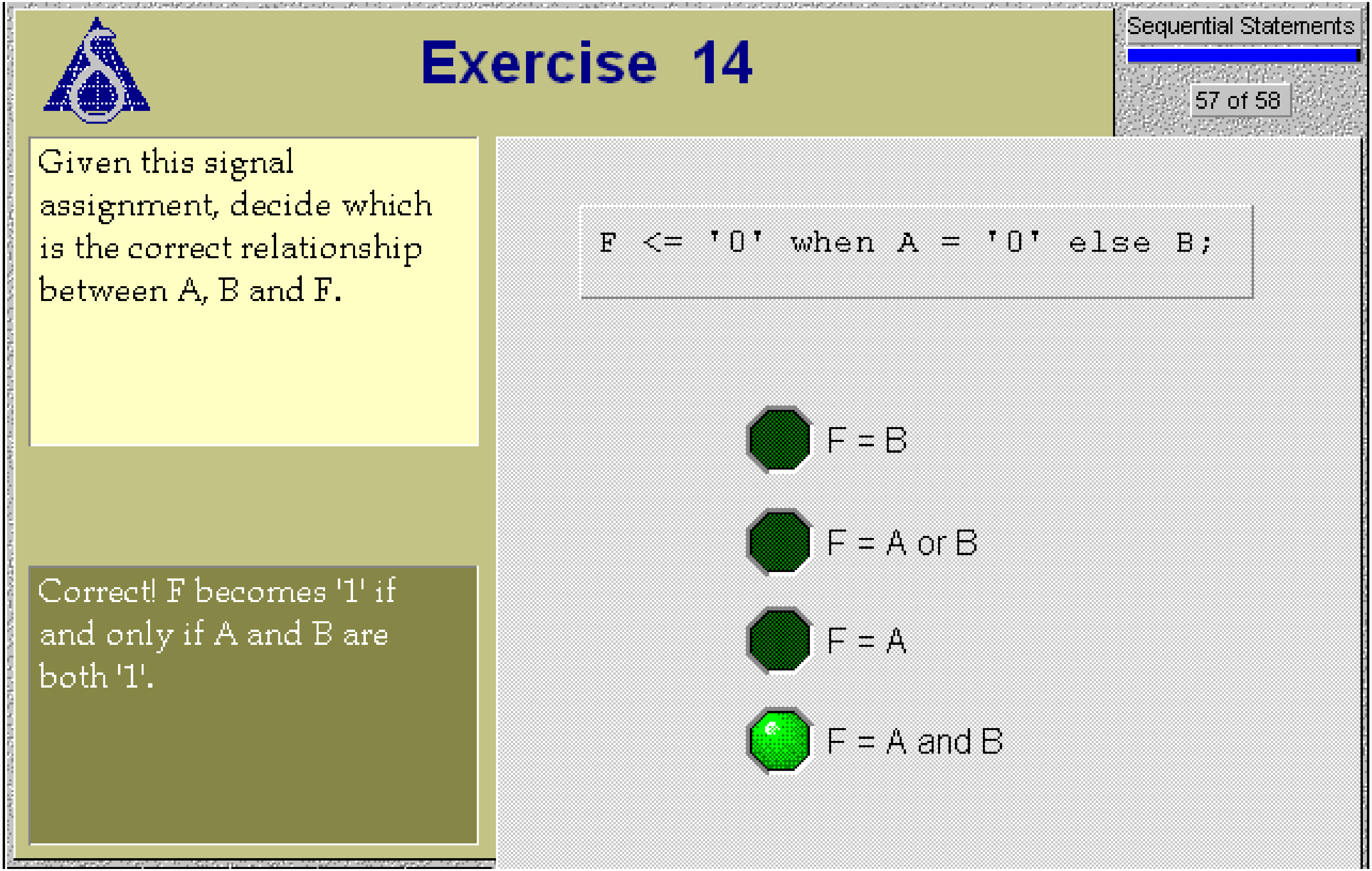
One of the commonest and most useful is the conditional signal assignment, which is equivalent to a process containing an if statement.

The conditional assignment is often a convenient way to describe and synthesize combinational logic.

```
L: F <= A when S0 = '1' else
        B when S1 = '1' else
        C;
```

Equivalent

```
L: process (S0, S1, A, B, C)
begin
  if S0 = '1' then
    F <= A;
  elsif S1 = '1' then
    F <= B;
  else
    F <= C;
  end if;
end process L;
```



Sequential Statements

57 of 58

Exercise 14

Given this signal assignment, decide which is the correct relationship between A, B and F.

```
F <= '0' when A = '0' else B;
```

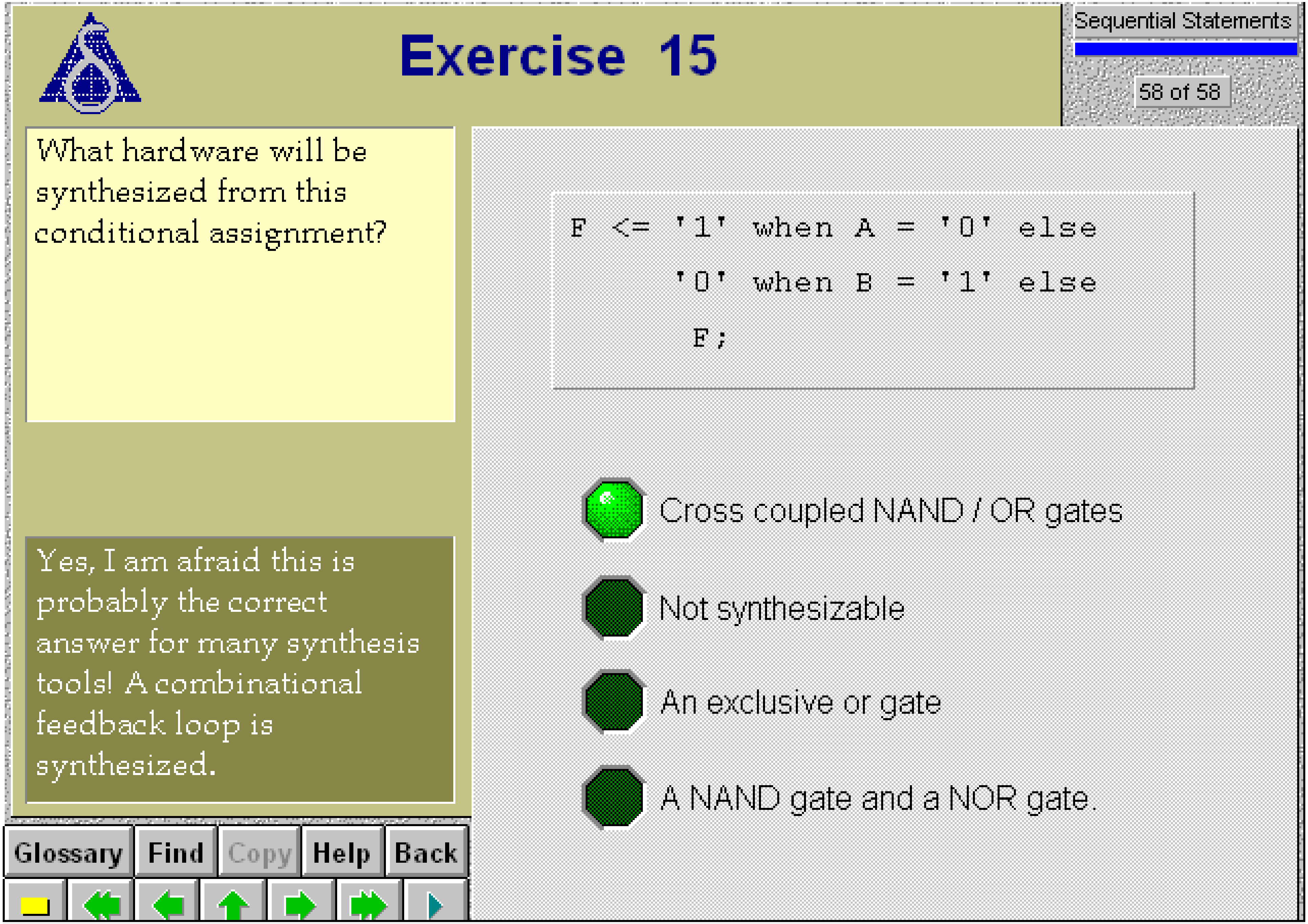
○ F = B

○ F = A or B

○ F = A

● F = A and B

Correct! F becomes '1' if and only if A and B are both '1'.



Exercise 15

What hardware will be synthesized from this conditional assignment?

```
F <= '1' when A = '0' else
     '0' when B = '1' else
      F;
```

● Cross coupled NAND / OR gates

○ Not synthesizable

○ An exclusive or gate

○ A NAND gate and a NOR gate.

Yes, I am afraid this is probably the correct answer for many synthesis tools! A combinational feedback loop is synthesized.