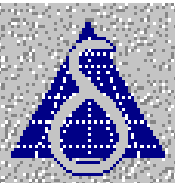


Lesson 4

Processes

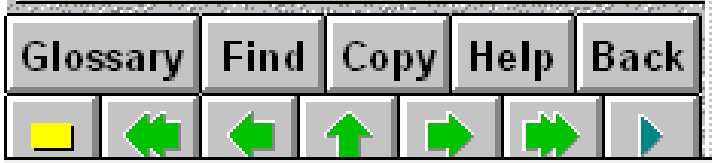
We will now look at the main way to describe behaviour in VHDL. In the next two lessons you will learn about VHDL processes, and how they can be used to simulate and synthesize your hardware.

The Process

Let's look now at the process, which is one of the most important constructs in VHDL.

The process is used to describe the behaviour of a part of a system without getting into the details of the implementation, so can be used to describe hardware at high levels of abstraction, or even to describe test vectors and simulation environments.

```
process
   ...
begin
   ...
end process;
```

The Process

The process is a concurrent statement which is written inside an architecture.

All processes execute in parallel with respect to all other processes.

Each process describes a separate piece of hardware, or an autonomous, concurrent task. Think of each process as a microcontroller executing its own program.

```
architecture V3 of AOI is
  signal AB, CD, O: STD_LOGIC;
begin
  process (A, B, C, D)
  begin
    AB <= A and B after 2 NS;
    CD <= C and D after 2 NS;
  end process;

  process (AB, CD)
  begin
    O <= AB or CD after 2 NS;
  end process;

  process (O)
  begin
    F <= not O after 1 NS;
  end process;
end V2;
```

The Process

Inside a process, VHDL looks and behaves like software! The statements inside a process execute in sequential order, from top to bottom.

Many of the statements inside the process take their syntax directly from the **ada** programming language. These sequential statements are written between begin and **end process;**

```
process (SEL, A, B, C)
begin
   if SEL = '1' then
      OP <= A and MASK;
   else
      OP <= B;
   end if;
end process;
```

The Process

Optionally, a process can be given a label as shown. Optionally, the label can be repeated at the end of the process, as shown for process P2. But if the label is repeated, you must use exactly the same name as the label at the top of the process!

```
P1: process (SEL, A, B, C)
begin
  if SEL = '1' then
    OP <= A and MASK;
  else
    OP <= B;
  end if;
end process;


P2: process (SEL, A, B, C)
begin
  if SEL = '1' then
    OP <= A and MASK;
  else
    OP <= B;
  end if;
end process P2;
```

Sensitivity Lists

The process comes in two flavours.

The first has a sensitivity list, which is a list of signals given in parenthesis at the top of the process, as shown. The process wakes up when an event occurs on one of the signals in the sensitivity list, then executes once from top to bottom.

```
process (SEL, A, B, C)
begin
  if SEL = '1' then
    OP <= A and MASK;
  else
    OP <= B;
  end if;
end process;
```

Sensitivity Lists

Were an event to occur on the signal MASK, this process would **not** wake up, because MASK is not in the sensitivity list.

```
process (SEL, A, B, C)
begin
  if SEL = '1' then
    OP <= A and MASK;
  else
    OP <= B;
  end if;
end process;
```

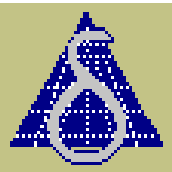
Event on MASK

Sensitivity Lists

Were an event to occur on the signal C, the process would wake up, even though the process does not make use of the value of C. So, the sensitivity list is the set of signals that the process is "sensitive to", and **not** the inputs to the process.

```
process (SEL, A, B, C)
begin
   if SEL = '1' then
      OP <= A and MASK;
   else
      OP <= B;
   end if;
end process;
```

Event on C



Sensitivity Lists

When a process is used to synthesize combinational logic, it is very important that the sensitivity list is **complete**, i.e. all the inputs to the logic must be in the sensitivity list. Otherwise, the process will not recalculate the value of the outputs when the missing inputs change value.

```
process (SEL, A, B, C)
begin
   if SEL = '1' then
      OP <= A and MASK;
   else
      OP <= B;
   end if;
end process;
```

Input missing from sensitivity list means big problems for synthesis!

Wait Statements

The other flavour of process does not have a sensitivity list at the top, but instead contains one or more wait statements.

The process starts to execute at time 0, then suspends whenever it executes a wait statement.

Should the process reach the end, it would start executing again from the top.

```
Stimulus: process
begin
   Reset <= '0';
   wait for 50 NS;
   Reset <= '1';
   wait;
end process;
```

Wait Statements

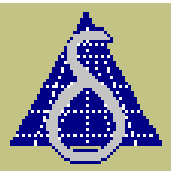
The wait for statement suspends the process for a given amount of time. During the time the process is suspended, other processes could be executing.

This kind of process is useful for generating test vectors, but not for describing hardware at the Register Transfer Level.

```
Stimulus: process
begin
   Reset <= '0';
   wait for 50 NS;
   Reset <= '1';
   wait;
end process;
```

Time = 0

Time = 50 NS



Wait Statements

The wait statement without any parameters means **wait forever.** In other words, stop the process at this point.

Without the **wait** at the end, the process would immediately start executing again from the top, and would become an infinite loop. So, the **wait** is very important in this case. Don't forget it!

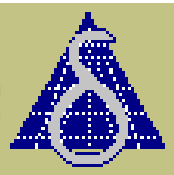
```
Stimulus: process
begin
   Reset <= '0';
   wait for 50 NS;
   Reset <= '1';
   wait;
end process;
```


Wait Statements

This style of process is usually the best way to generate test vectors within a VHDL test bench. In practice, the test bench processes get much more complex than this, but the basic principle is still the same:

1) Apply some vectors
2) Wait for some time
3) Apply some more vectors
4) Wait for some more time
5) And so on!

```
TestVectors: process
begin
   A <= "0000";
   B <= "0000";
   wait for 10 NS;
   A <= "1111";
   wait for 10 NS;
   B <= "1111";
   wait for 10 NS;
   A <= "0101";
   B <= "1010";
   wait;
end process;
```



Signal Assignments

Now let's look at signal assignments in more detail.

Signal assignments are used to create events on signals, and thus allow processes to communicate.

We've seen signal assignments before as concurrent statements. Signal assignments inside a process are sequential statements; they execute in order from top to bottom.

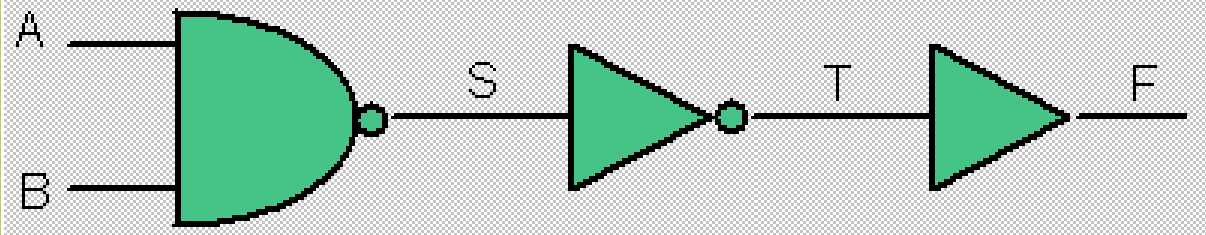
```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

Signal Assignments

The expressions on the right of the assignments describe logical or arithmetic operations.

This process would be synthesized to hardware as shown.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

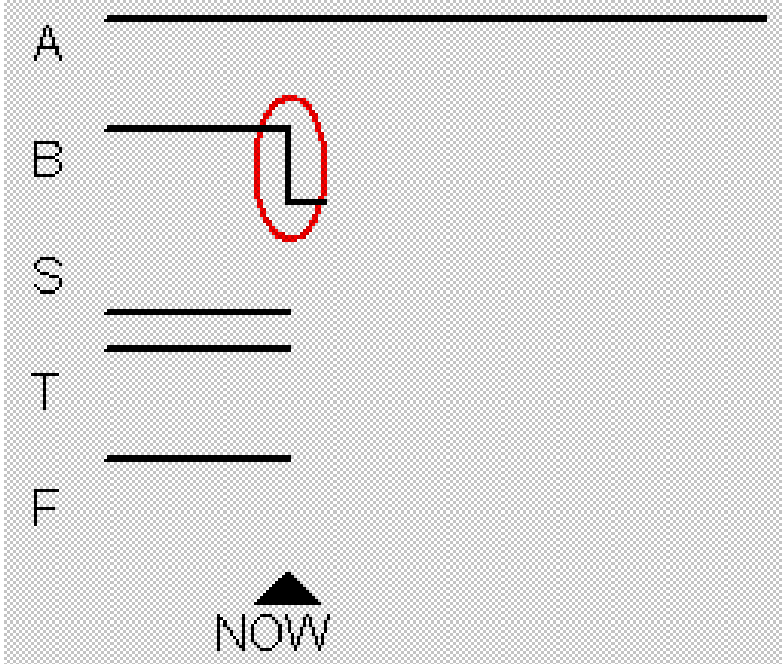


Signal Assignments

Let's follow through the execution of this process to see exactly what happens when it executes.

We'll start with an event on B. Because B is a signal in the sensitivity list of the process, the process wakes up and starts executing.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

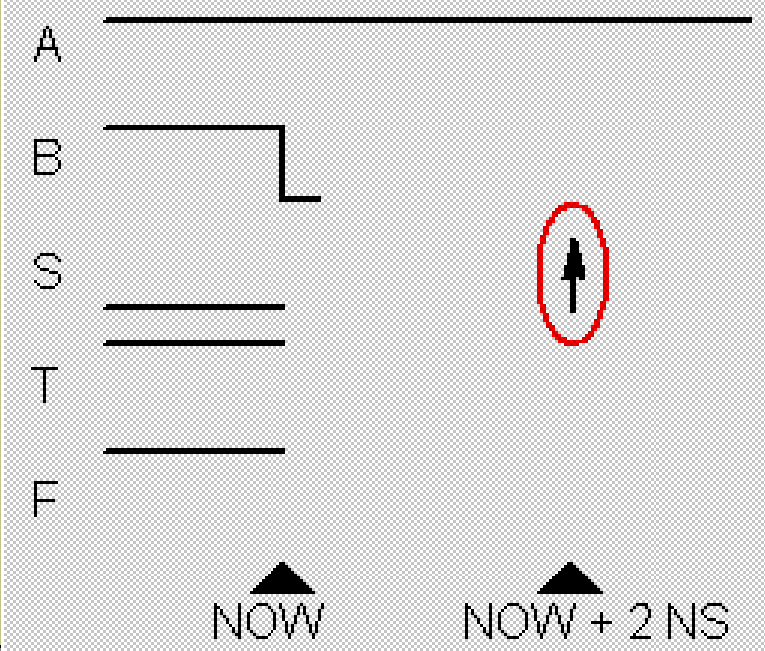


Signal Assignments

The first signal assignment is executed. This calculates A nand B then schedules an event on S to occur 2 NS after the current time.

Note that a signal assignment does not change the value of the signal; it only sets up an event to occur in the future.

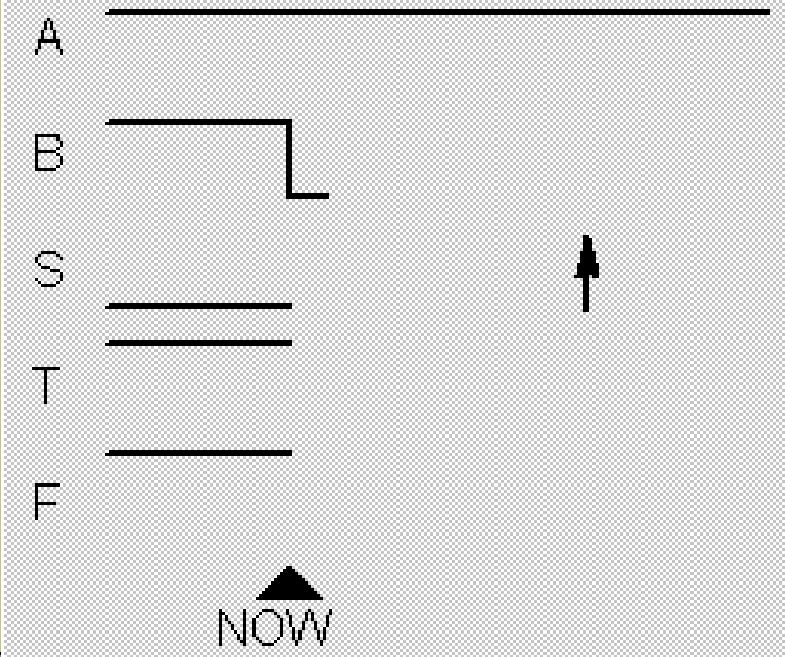
```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```



Signal Assignments

The next signal assignment then executes. But because S still has its old value, this assignment has no effect. There will not be an event scheduled on T.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

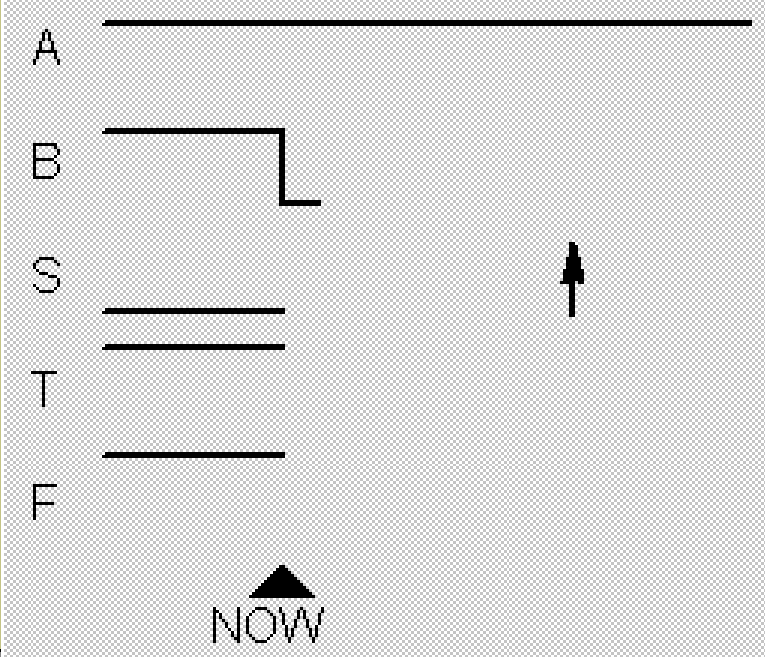


Signal Assignments

The final signal assignment then executes, but again without having any effect.

We've reached the end of the process, so it suspends and waits for an event on a signal in the sensitivity list. There is still an outstanding event scheduled to occur on S in 2 NS time.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

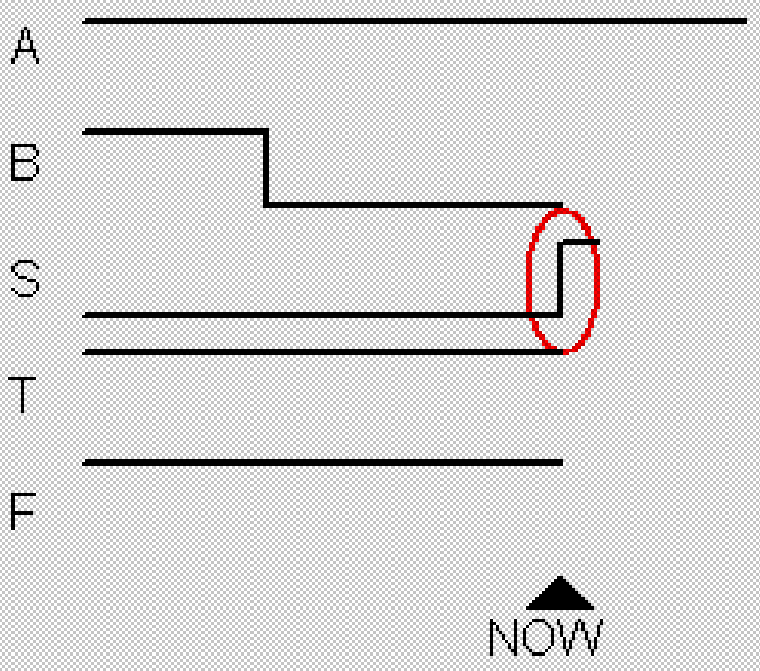


Signal Assignments

Simulation time advances, and other processes may execute.

When simulation time reaches the time of the event scheduled on S, then the event happens, and S changes value. The event causes the process to execute once more, since S is in the sensitivity list.

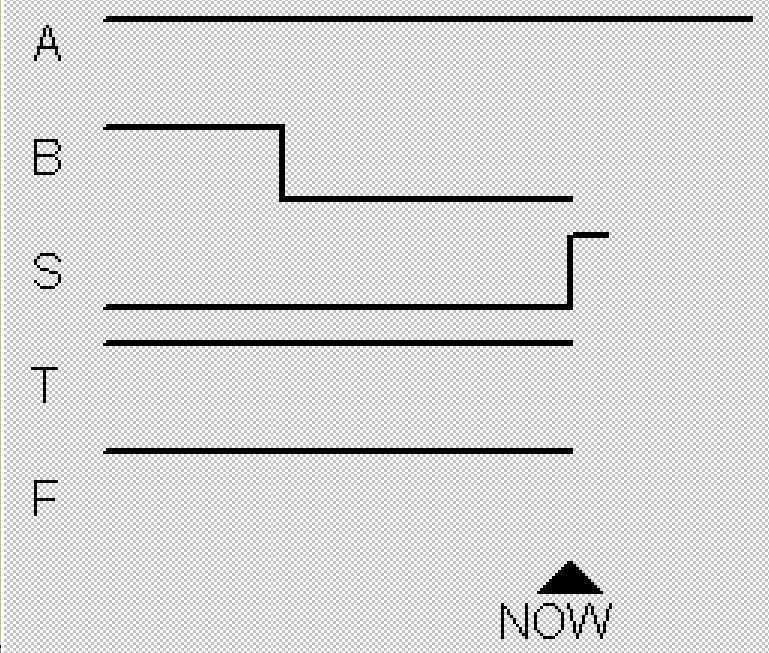
```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

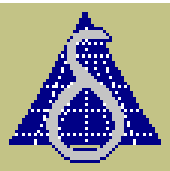


Signal Assignments

The first signal assignment within the process executes, but has no effect, since **A nand B** has the same value as S.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```





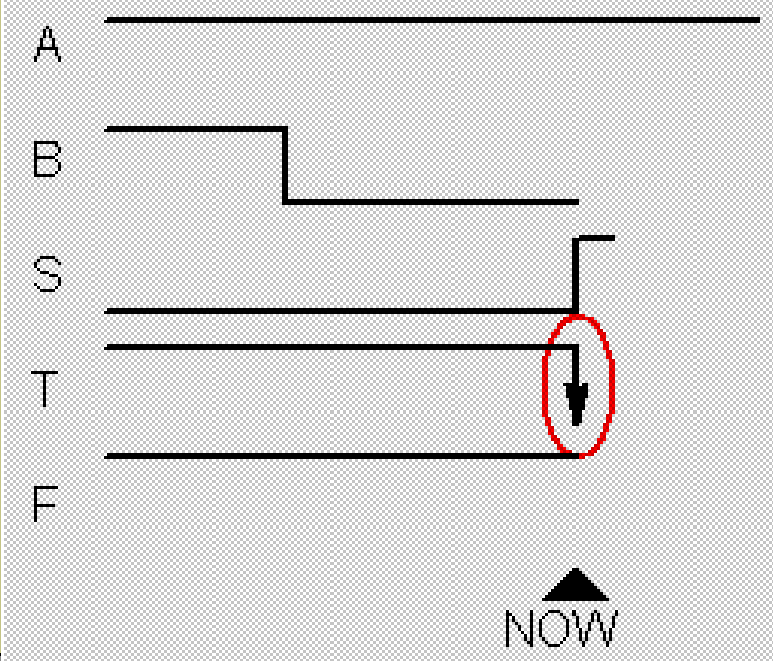
Signal Assignments

The second signal assignment now executes.

Since S has changed value, a new event will be scheduled on T with a delay of zero.

But remember that a signal assignment only sets up an event, it doesn't change the value of the signal. We'll see why that's so important next!

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

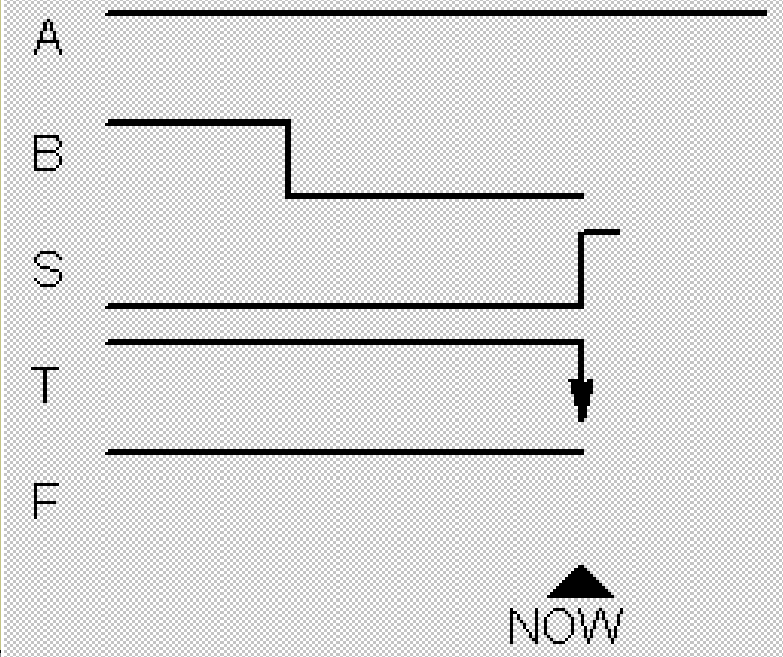


Signal Assignments

Now comes the crunch!

The final signal assignment is executed, but T still has its old value of '1', so the assignment to F has no effect. There is an event queued for T, but the event will not happen until execution of the process is complete. So we reach the end of the process, and T has not yet changed to '0'.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```



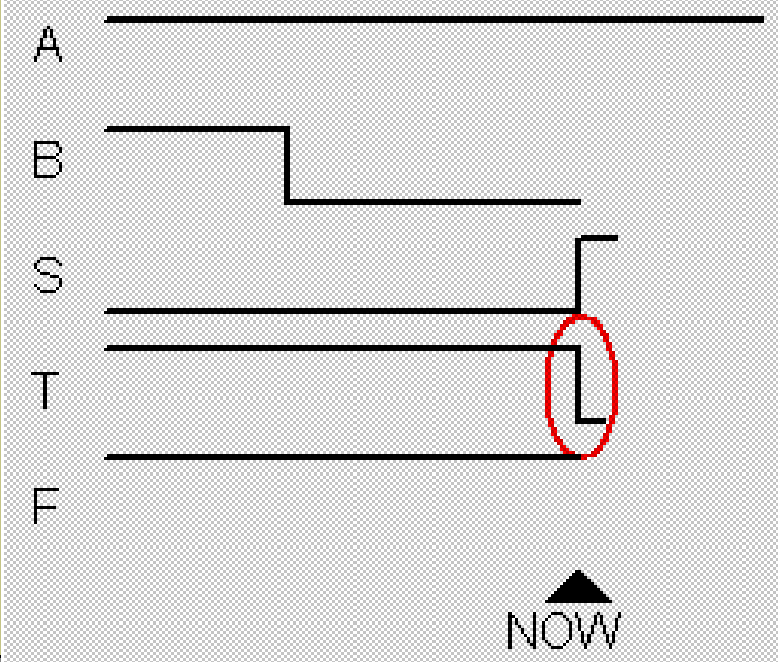
Signal Assignments

Once all the processes active at the current time have finished executing, the simulator deals with any outstanding events, and so T changes value.

The small delay associated with the event on T is known as a delta delay.

The event on T causes the process to execute for a third time.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

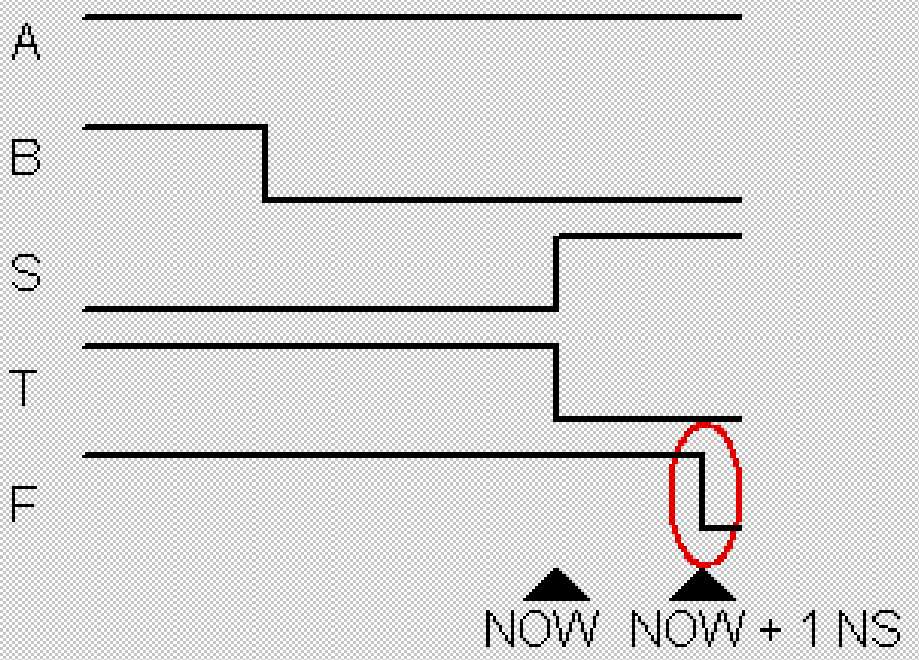


Signal Assignments

This time the assignments to S and T have no effect, but the assignment to F schedules an event 1 NS into the future.

When this event on F eventually happens, the process will **not** execute again, because F is not in the sensitivity list.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```

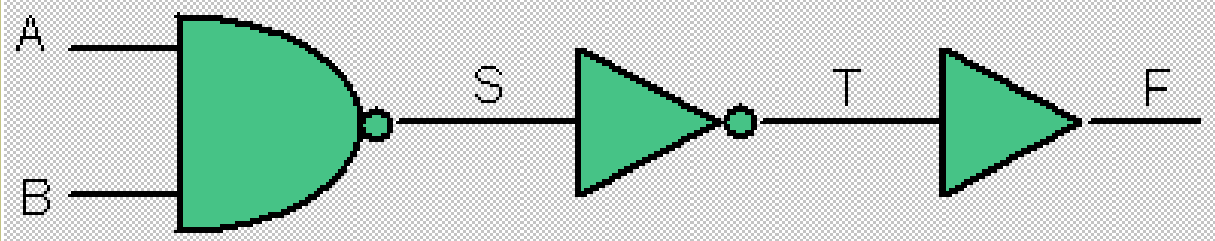


Signal Assignments

The delays have been included in this example to help you understand signal assignments.

Delays are usually **not** included when VHDL is used to synthesize hardware, because the synthesis tool would just ignore them! The actual delays of the synthesized hardware depend on the technology used and the constraints given for synthesis.

```
process (A, B, S, T)
begin
   S <= A nand B after 2 NS;
   T <= not S;
   F <= T after 1 NS;
end process;
```



Signal Assignments

A process like this can be used to synthesize combinational logic if certain rules are obeyed.

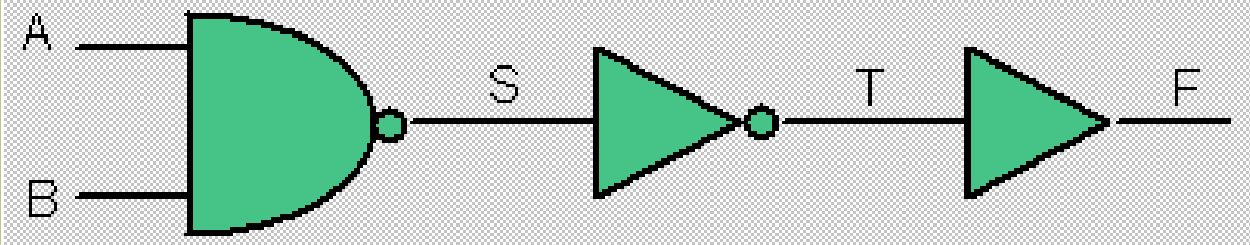
1) The sensitivity list must include all the inputs

2) The outputs must depend only on the values of the current inputs (no feedback)

The third and final rule is introduced later on.

```
process (A, B, S, T)
begin
   S <= A nand B;
   T <= not S;
   F <= T;
end process;
```

This process is OK!

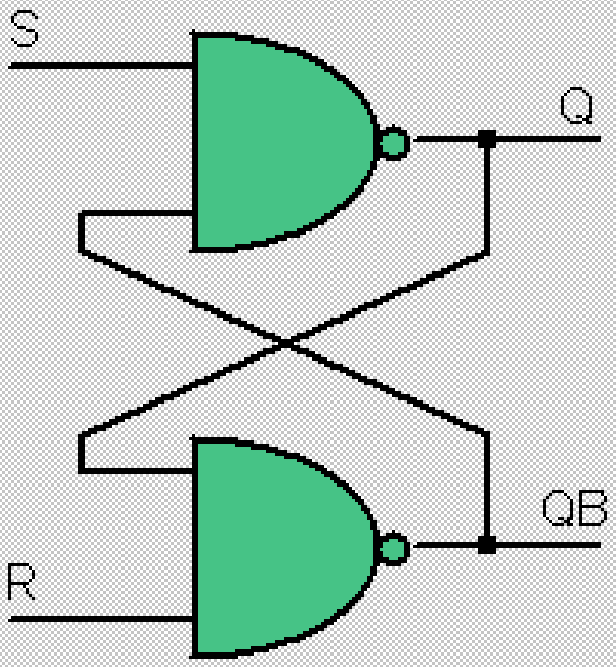


Signal Assignments

Here is an example of a process which breaks the 2nd rule of combinational logic synthesis, and does contain feedback.

Although some synthesis tools will accept this code, problems are likely to occur during optimization. The synthesizer thinks it's still dealing with a combinational circuit, and does not understand the sequential behaviour of this circuit.

```
Feedback: process (S, R, Q, QB)
begin
   Q  <= S nand QB;
   QB <= R nand Q;
end process;
```



Exercise 1

Suppose an event occurs on A at time 100 NS and the process executes. At what time is the end of the process reached? Type in your answer, then press the return key.

```
P1: process (A, B)
begin
   B <= A after 5 NS;
   C <= B after 5 NS;
end process P1;
```

NS

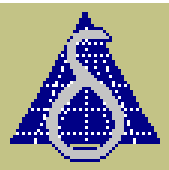
Exercise 2

In the same process, given an event on A at time 100 NS, at what time would C eventually change value?

```
P1: process (A, B)
begin
   B <= A after 5 NS;
   C <= B after 5 NS;
end process P1;
```

NS

The Answer



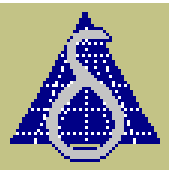
Exercise 3

We have seen that given an event on A at time 100 NS, both B and C eventually change value. But how many times does the process execute during this period?

```
P1: process (A, B)
begin
   B <= A after 5 NS;
   C <= B after 5 NS;
end process P1;
```

Correct! The 1st execution causes the event on B, the 2nd causes the event on C.

- Twice, caused by events on A and then on B
- Not at all
- Three times, because events occur on A, B and C
- Once only, when there is an event on A



Exercise 4

A different process now. At what time does this process start executing? Type your answer in the box then hit return.

```
P2: process
begin
   B <= '0';
   wait for 20 NS;
   B <= '1' after 10 NS;
   wait for 5 NS;
   C <= B after 1 NS;
   wait;
end process P2;
```

NS

Exercise 5

For the same process, at what time does the final **wait** statement execute? That is, the **wait** statement just before the **end process.**

```
P2: process
begin
   B <= '0';
   wait for 20 NS;
   B <= '1' after 10 NS;
   wait for 5 NS;
   C <= B after 1 NS;
   wait;
end process P2;
```

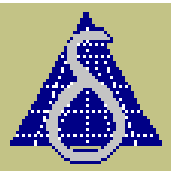
NS

Exercise 6

For the same process, at what time does B change from '0' to '1' ?

```
P2: process
begin
   B <= '0';
   wait for 20 NS;
   B <= '1' after 10 NS;
   wait for 5 NS;
   C <= B after 1 NS;
   wait;
end process P2;
```

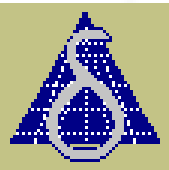
NS



Exercise 7

We've looked at the times at which things happen, but what is the value assigned to C by the bottom signal assignment?

```
P2: process
begin
   B <= '0';
   wait for 20 NS;
   B <= '1' after 10 NS;
   wait for 5 NS;
   C <= B after 1 NS;
   wait;
end process P2;
```



Exercise 8

And finally: At what time does the event on signal C occur?

```
P2: process
begin
   B <= '0';
   wait for 20 NS;
   B <= '1' after 10 NS;
   wait for 5 NS;
   C <= B after 1 NS;
   wait;
end process P2;
```

NS