What is a process?

A process block has the form

```
process (SENSITIVITY) is
begin
  -- one or more statements
end process;
```

It defines a simple but very powerful behavior: when a signal in the SENSITIVITY list changes, the statements inside the process block are executed.

Let's make up an example with three input signals, `a`, `b`, and `strobe`, and one output `y`.

```
process (strobe) is
begin
  y <= a and b;
end process;
```

If the assignment `y <= a and b` were placed outside the process, it would happen continuously. Any time either `a` or `b` changed, the output would be recomputed. But with the statement inside the process, `y` is only recomputed when `strobe` changes. This is a pretty weird circuit — and it's not one we'd ever want to build — but it illustrates the key idea of a process. We'll discuss the real power of this behavior when we get to sequential circuits.

You'll see many references say that a `process` defines a group of sequential statements within the larger concurrent architecture. While this is technically true, it's not a very helpful way to think about the process block. Rather than unpack this confusing definition, we'll examine the three main ways that process blocks are used, and develop intuition for how each one works.

All the signals

What would happen if you put all of the signals in the sensitivity list? For example:

```
process (a, b) is
begin
  y <= a and b;
end process;
```

Now, `y` will be recomputed every time `a` or `b` changes — just as if the statement were outside the process.

Why would you ever want to do this?

Several constructs, notably `if`, `case` and `for`[1] can only be used inside a process block. While you could always write these with basic operators and `when/else` constructs, sometimes it's just more convenient to define a logic function with `if`, `case`, and `for`.

VHDL-2008 provides a nicer way to specify this behavior, using the keyword `all`. The example below shows a 2-bit binary decoder which takes a 2-bit binary input and sets one of four output wires high.

```
process (all) is
begin
  case input is
    when "00" => y <= "0001"
    when "01" => y <= "0010"
    when "10" => y <= "0100"
    when "11" => y <= "1000"
```

[1] Yes, VHDL has 'for' loops. Don't get exited — they don't behave anything like the loops in a programming language.

```
    when others => y <= "XXXX" -- Invalid input, set output to be undefined
  end case;
end process;
```

We could write the same thing without the process statement:

```
y <= "0001" when input = "00" else
     "0010" when input = "01" else
     "0010" when input = "10" else
     "0010" when input = "11" else
     "XXXX"
```

Depending on your tastes, you may find the `case` version cleaner or easier to understand.

No signals

If we do not put any signals in the sensitivity list, then the VHDL simulator runs it immediately. This is useful for creating a testbench, which generates signals to drive the module being tested (usually called the "device under test", or DUT), and does not have any inputs of its own.

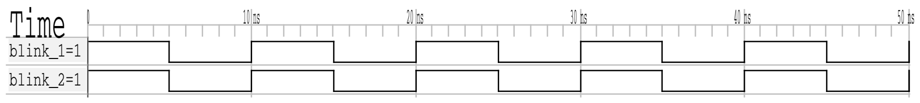
```
signal blink_1 : std_logic;
signal blink_2 : std_logic;

process begin
  blink_1 <= '1';
  blink_2 <= '1';
  wait for 5 ns;

  blink_1 <= '0';
  blink_2 <= '0';
  wait for 5 ns;
end process;
```

This example has two major differences from the previous ones: it assigns the same signal multiple times, and it uses a `wait` statement. This is the sequential behavior of the `process` block coming out. The simulator will perform operations from top to bottom, much like you would expect in a programming language. When it reaches a `wait` statement, the simulator steps the simulation clock forward by the specified amount and continues executing.

If we plot the waveforms from this testbench, we get the following:



Two things are worth discussiong. First, why does it go forever? Once a process finishes executing, it is ready to run again. Since there is no sensitivity list, it just restarts as soon as it completes. If you don't want this to happen, then just put a `wait` statement at the bottom of your process, with no duration specified. This will cause it to wait there forever and not repeat.

Second, it appears that both signals change at the same time, even though `blink_1` was first. If you could zoom in on the waveform, you would see that they really do change at the same instant, and not simply very close to one another. Why is this? As far as the simulation is concerned, no time passes until a wait statement occurs, so the signals change at exactly the same time. If we were to make this into a circuit, you would again see that the signals operate concurrently and change at the same time.

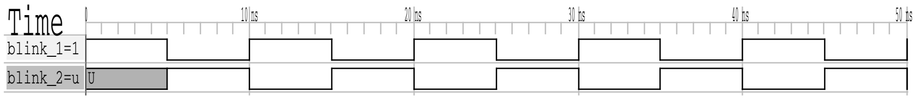
Now for an example that may be a bit discomforting:

```
process begin
  blink_1 <= '1';
  blink_2 <= blink_1;
  wait for 5 ns;

  blink_1 <= '0';
  blink_2 <= blink_1;
  wait for 5 ns;
end process;
```

If your brain is in software mode, you would expect this to be identical to the example above. First `blink_1` gets set to 1, and then this value is copied to `blink_2`, causing `blink_1` and `blink_2` to cycle in sync just like before. But if your brain is in software mode, you would be wrong.

What really happens is shown in the resulting waveform below:



The reason for this unexpected behavior is that the signals don't take on their new values until some simulation time passes, so in the first assignment `blink_2 <= blink_1`, `blink_1` still has its old value of `0`. This may seem like a very annoying bug, but it's actually a feature. Think about what would happen if we turned this into a circuit which was triggered by some event. At the instant the trigger event occurs, `blink_1` would start to change to `1`. But at that same instant (remember, there's no time delay between the first two statements), `blink_2` will take on the current value of `blink_1`. If this still seems weird, be patient and it will make more sense when we start building sequential logic.

Hopefully now it makes sense why `process` blocks are described as groups of sequential statements, and why that's not a helpful definition. They do contain sequential statements, but not in the way you would expect from software.

Synthesizeable and non-synthesizeable constructs

This brings us to one of the most important concepts in VHDL: synthesizeability.

VHDL is designed to do all kinds of things: it can describe the behavior of physical circuits, it can be used to create testbenches and simulate designs, and it can be used as source code for creating digital circuits.

Any language feature can be used in a testbench, but only a subset of the language can actually be turned into real circuit hardware. This is known as the "synthesizeable" subset of the language.

For example, the `report` statement causes a message to be printed to the simulator log. You can't synthesize this, because there's no simulator log in the finished circuit. There's no `assert` either; you could build hardware that monitors the assertion condition, but there's no way to quit the circuit like you would quit a simulation.

In the same way, you can't synthesize a `wait` statement. While a `wait` is useful in a testbench or for modeling existing circuitry (such as a 74-series logic chip with known propagation delay), there's no way to just insert a `wait` into a circuit. Many circuits do need to keep track of time, but there are other ways to do that.

In addition, the `process` block itself can be used to describe all sorts of things which are not physically possible to build. For now, stick to `process(all)` for combinational logic and `process` (with no sensitivity list) for testbenches, and you'll stay out of trouble.