Hardware Modeling [VU] (191.011)
– WS24 –
Behavioral Modeling

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

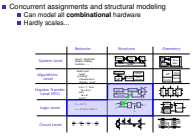
In this lecture we will discuss synthesizable processes, which can be used to describe hardware using sequential statements.

Hardware Modeling [VU] (191.011)
– WS24 –
Behavioral Modeling

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

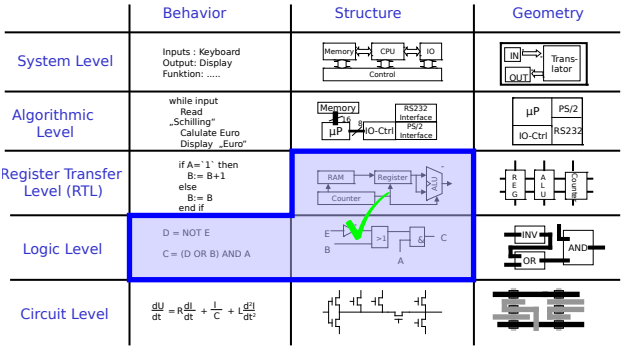
Modified: 2025-03-20, 14:05 (830283c)

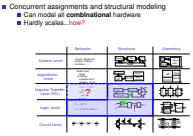


In previous lectures we have discussed how hardware can be described by using concurrent signal assignments and structural modeling. This combination already allows us to describe arbitrary combinational hardware. However, it hardly scales. Think about arithmetic logic units in CPUs which are used to perform computations. Although it is only combination, it can become quite complex and while we *could* describe it with the means we discussed so far, it would not be a very pleasant experience. The reason being that we are not really describing our hardware programmatically on the register-transfer-level as we would ideally do. Obviously, we need a method for modeling circuits with a complex behavior.

Introduction

- Concurrent assignments and structural modeling
  - Can model all **combinational** hardware
  - Hardly scales...

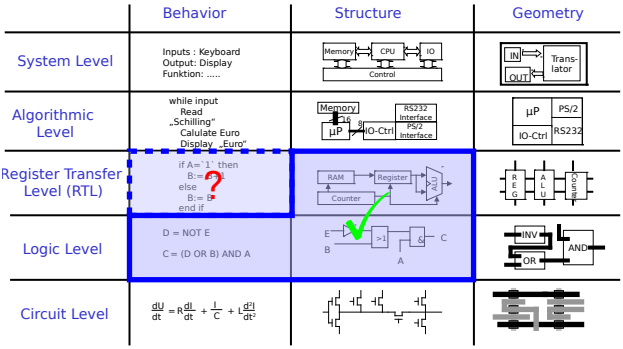


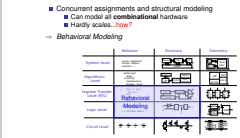


It would be nice if we could make use of the control flow statements we introduced in the lecture about VHDL basics Furthermore, it is desirable that we can encapsulate parts of a complex design in distinct structures, without the need to create a new entity and architecture and to instantiate it. Additionally, as we as humans try to break complex behavior down in sequences of simpler behavior, we would desire a means to describe our circuits sequentially.

Introduction

- Concurrent assignments and structural modeling
  - Can model all **combinational** hardware
  - Hardly scales...how?



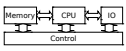
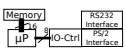


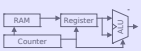
This is where behavioral modeling comes in.

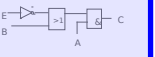
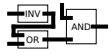
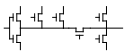
Introduction

- Concurrent assignments and structural modeling
  - Can model all **combinational** hardware
  - Hardly scales...how?

⇒ *Behavioral Modeling*

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling" Calulate Euro Display „Euro"		

Register Transfer Level (RTL)	if A='1' then
	B:= B+1
	else
	B:= B-1
	end if;
			

Logic Level	C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R\frac{dI}{dt} + \frac{I}{C} + L\frac{d^2I}{dt^2}$		

Revolves around processes
Must be synthesizable

In a nutshell, behavioral modeling allows us to use synthesizable processes to describe our hardware.

Behavioral Modeling

- Revolves around processes
  - Must be synthesizable

Revolves around processes
Must be synthesizable
“single-use entity and architecture”

Up to some extent, we can picture such synthesizable processes to be an inline entity and architecture. That is, we can split our model into functionally loosely connected submodules in a much more concise manner than by creating distinct entities and instantiating them.

Behavioral Modeling

- Revolves around processes
  - Must be synthesizable
  - “single-use entity and architecture”

- Revolves around processes
  - Must be synthesizable
  - "single-use entity and architecture"
  - Control flow statements and *variables*
  - *Sequential* description

We can also use control flow statements inside processes, thus giving us access to `if`, `case` and even `loop` statements. Furthermore, we can use variables and describe our circuits in a somewhat sequential manner. We will discuss both in detail during this lecture.

Behavioral Modeling

- Revolves around processes
  - Must be synthesizable
  - "single-use entity and architecture"
  - Control flow statements and *variables*
  - *Sequential* description

- Revolves around processes
  - Must be synthesizable
  - "single-use entity and architecture"
  - Control flow statements and *variables*
  - *Sequential* description
- **Complements** struct. modeling and concurrent assignments

However, it must be stressed that behavioral modeling is not an alternative to structural modeling and concurrent assignments but rather a powerful complement. Typically, you would use all three methods throughout a design.

Behavioral Modeling

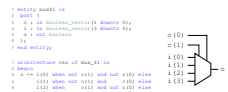
- Revolves around processes
  - Must be synthesizable
  - "single-use entity and architecture"
  - Control flow statements and *variables*
  - *Sequential* description
- **Complements** struct. modeling and concurrent assignments

- Revolves around processes
  - Must be synthesizable
  - "single-use entity and architecture"
  - Control flow statements and *variables*
  - *Sequential* description
- **Complements** struct. modeling and concurrent assignments
- Ubiquitous in synchronous designs

Furthermore, as we will see in chapter 3, behavioral modeling is ubiquitous in synchronous designs as it allows for very concise and maintainable description of such circuits. We will now look at a first example of how a behavioral model of a circuit looks like and then elaborate on its details.

Behavioral Modeling

- Revolves around processes
  - Must be synthesizable
  - "single-use entity and architecture"
  - Control flow statements and *variables*
  - *Sequential* description
- **Complements** struct. modeling and concurrent assignments
- Ubiquitous in synchronous designs

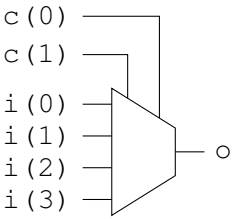


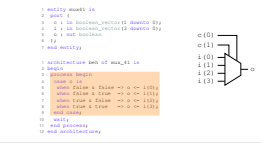
Recall the implementation of a simple 4:1 multiplexer that you saw in the lecture about entities and architectures. The slide shows the respective implementation based on concurrent signal assignments. Let us now apply our already gathered knowledge and rewrite this implementation as a process. In a first attempt we could make use of the `case` statement to describe the select logic of the multiplexer in a more comprehensible manner.

Example: Multiplexer


```
1 entity mux41 is
2  port (
3   c : in boolean_vector(1 downto 0);
4   i : in boolean_vector(3 downto 0);
5   o : out boolean
6  );
7 end entity;
```

```
1 architecture csa of mux_41 is
2 begin
3  o <= i(0) when not c(1) and not c(0) else
4       i(1) when not c(1) and     c(0) else
5       i(2) when     c(1) and not c(0) else
6       i(3) when     c(1) and     c(0);
7 end architecture;
```



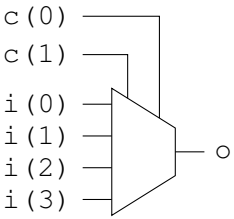


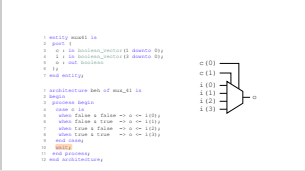
This is shown on the slide. We use a case statement to determine the output based on the control signal `c` and complete the process with a `wait` statement. So far so good, right?

Example: Multiplexer


```
1  entity mux41 is
2   port (
3    c : in boolean_vector(1 downto 0);
4    i : in boolean_vector(3 downto 0);
5    o : out boolean
6   );
7  end entity;
```

```
1  architecture beh of mux_41 is
2  begin
3   process begin
4    case c is
5     when false & false => o <= i(0);
6     when false & true  => o <= i(1);
7     when true & false  => o <= i(2);
8     when true & true   => o <= i(3);
9    end case;
10    wait;
11   end process;
12 end architecture;
```



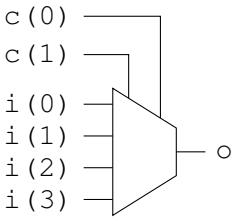


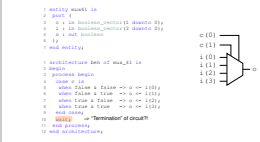
When we initially introduced processes, we did not really motivate the `wait` statement that we always put at the end of their bodies. We just stated that it is required for the process to terminate.

Example: Multiplexer


```
1 entity mux41 is
2  port (
3   c : in boolean_vector(1 downto 0);
4   i : in boolean_vector(3 downto 0);
5   o : out boolean
6  );
7 end entity;
```

```
1 architecture beh of mux_41 is
2 begin
3  process begin
4   case c is
5    when false & false => o <= i(0);
6    when false & true  => o <= i(1);
7    when true & false  => o <= i(2);
8    when true & true   => o <= i(3);
9   end case;
10  wait;
11 end process;
12 end architecture;
```



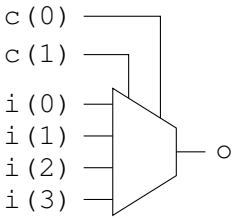


However, while this might be sensible for a sequential piece of code executed during simulation, it hardly makes sense for describing a circuit. After all, exactly what hardware behavior is this supposed to model?

Example: Multiplexer


```
1  entity mux41 is
2   port (
3    c : in boolean_vector(1 downto 0);
4    i : in boolean_vector(3 downto 0);
5    o : out boolean
6   );
7  end entity;
```

```
1  architecture beh of mux_41 is
2  begin
3   process begin
4    case c is
5     when false & false => o <= i(0);
6     when false & true  => o <= i(1);
7     when true & false  => o <= i(2);
8     when true & true   => o <= i(3);
9    end case;
10   wait;      ⇒ "Termination" of circuit?!
11  end process;
12 end architecture;
```



"Termination" of circuit not sensible
Stays active as long as powered

While termination is something a program might do, a circuit will always be active unless its power is cut. This is why a process with the simple `wait` statement we used so far is *not synthesizable* nd should therefore not be used to describe hardware.

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered

"Termination" of circuit not sensible
Stays active as long as powered
Instead: Model circuit as "sequential routine" for input changes

If we think about our multiplexer, it simply maps its inputs `i` and `c` to its output `o`. Hence, whenever an input changes the circuit's output might change as well. In behavioral modeling we capture this as a sequential routine that is applied whenever an input changes. However, note that this is just an abstraction for describing a concurrent circuit! In reality there is neither a routine nor a sequential execution of our model.

wait on Statement

Beh. Mod.
Introduction
Sensitivity
wait on
Sensitivity List
all keyword
Process Simulation
Variables
Remarks

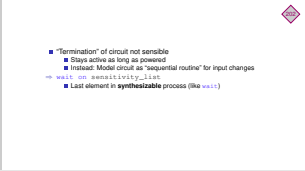
- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

The way we can capture this *sensitivity* o certain signals is using the `wait on` instead of the `wait` statement. This statement contains a, so-called, *sensitivity list* f signals to which the respective process is sensitive. Note that we refer to a process being sensitive to a signal when it reads it somewhere inside its body.

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`



In our code we simply replace the single `wait` statement we had at the end processes before by a fitting `wait on` statement.

wait on Statement

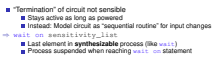
HWMod
WS24

Beh. Mod.
Introduction
Sensitivity
wait on
Sensitivity List
all keyword
Process Simulation
Variables
Remarks

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`

  - Last element in **synthesizable** process (like `wait`)



Thinking about a circuit, we can picture this statement to a check whether all inputs, and thus also the outputs, of the described circuit are stable. Only when an input changes, the output might change, which is modeled by the statements inside the process above the wait statement. In a simulation, this is achieved by letting the `wait on` statement suspend the process.

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`

  - Last element in **synthesizable** process (like `wait`)
  - Process suspended when reaching `wait on` statement

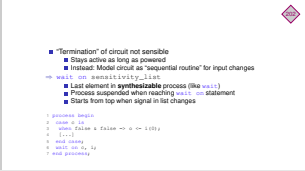
The process is only woken up again when a signal on the sensitivity list changes. Once this happens, the process is executed again from top to bottom. We can thus sort of think about a process as the mentioned routine for reacting to input changes.

wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`

- Last element in **synthesizable** process (like `wait`)
- Process suspended when reaching `wait on` statement
- Starts from top when signal in list changes



Let us have a look at the multiplexer process from the previous slide, with this change in-place. The process only drives the signal `o`, depending on the signals `i` and `c`. We can think of these two signals as being the inputs of the process. Therefore, our process is sensitive to changes of these two signals.

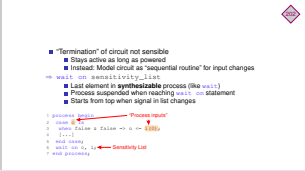
wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`

  - Last element in **synthesizable** process (like `wait`)
  - Process suspended when reaching `wait on` statement
  - Starts from top when signal in list changes

```
1 process begin
2  case c is
3   when false & false => o <= i(0);
4    [...]
5  end case;
6  wait on c, i;
7 end process;
```



wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

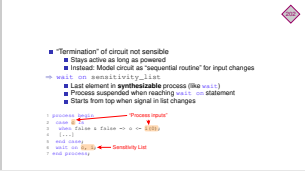
⇒ `wait on sensitivity_list`

  - Last element in **synthesizable** process (like `wait`)
  - Process suspended when reaching `wait on` statement
  - Starts from top when signal in list changes

```
1 process begin
2   case c is
3     when false & false => o <= i(0);
4     [...]
5   end case;
6   wait on c, i;
7 end process;
```

"Process inputs"

Sensitivity List



wait on Statement

- "Termination" of circuit not sensible
  - Stays active as long as powered
  - Instead: Model circuit as "sequential routine" for input changes

⇒ `wait on sensitivity_list`

  - Last element in **synthesizable** process (like `wait`)
  - Process suspended when reaching `wait on` statement
  - Starts from top when signal in list changes

```
1 process begin
2  case c is
3   when false & false => o <= i(0);
4   [...]
5  end case;
6  wait on c, i;
7 end process;
```

"Process inputs"

Sensitivity List

With the sensitivity list describing to which signals the circuit modelled by a process is sensitive to, its completeness is of course paramount. We will now demonstrate this at the hand of an example.

Example: Half-adder

Completeness of sensitivity list matters!

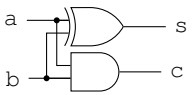
Completeness of sensitivity list matters!

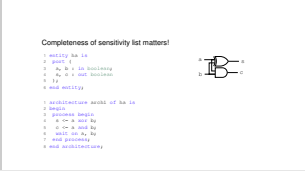
Consider the half-adder shown on the slide, featuring two inputs `a` and `b`, and two outputs `s` and `c`. The respective entity does therefore contain exactly these four ports. The circuit itself is quite simple with each output being generated from the two inputs via a single gate.

Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```



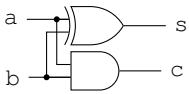


We can write a behavioral model of the half-adder as shown on the slide, consisting of a single process that drives the two outputs.

Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```



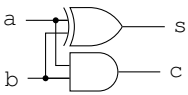

```
1 architecture arch1 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a, b;
7  end process;
8 end architecture;
```

Since the behavior modeled by this process is sensitive to both `a` and `b`, the process ends with a `wait on` statement sensitive to these two signals.

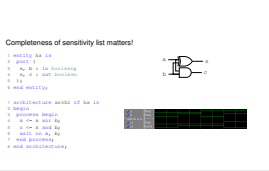
Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```




```
1 architecture arch1 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a, b;
7  end process;
8 end architecture;
```

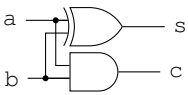


Consider the result of simulating this implementation, where we called the outputs `c1` and `s1` for later comparison. We can observe a sequence of inputs and the respective sequence of outputs generated by our circuit. For the simple half adder, with only four possible combinations of inputs, we can immediately observe that our implementation is correct.

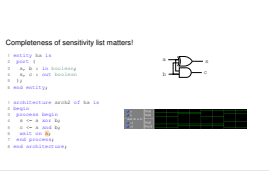
Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```




```
1 architecture arch1 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a, b;
7  end process;
8 end architecture;
```

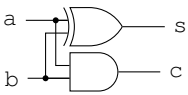


However, what if we forgot a signal, in the example on the slide `b`, when writing the sensitivity list?

Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```



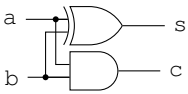

```
1 architecture arch2 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a;
7  end process;
8 end architecture;
```

If we simulate this modified implementation using the same input sequence as before, and look at the output traces `c2` and `s2`, we can observe that this implementation is **not** correct when simulated! Of course, this deviation from the desired circuit has its origin in the fact that our model is not sensitive to changes of the input `b`, although its output might change due a change of `b` only.

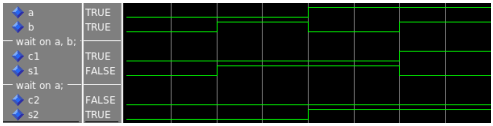
Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```




```
1 architecture arch2 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a;
7  end process;
8 end architecture;
```

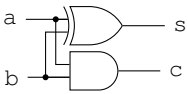


In conclusion, always keep in mind to what signals your processes should be sensitive to and write your sensitivity accordingly.

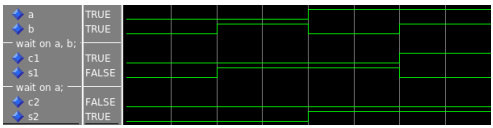
Example: Half-adder

Completeness of sensitivity list matters!

```
1 entity ha is
2  port (
3   a, b : in boolean;
4   s, c : out boolean
5  );
6 end entity;
```




```
1 architecture arch2 of ha is
2 begin
3  process begin
4   s <= a xor b;
5   c <= a and b;
6   wait on a;
7  end process;
8 end architecture;
```



On the previous slides we have seen how we can replace the `wait` statement at the end of a process by a `wait on` statement. And while this can model the behavior we actually want, there exists an equivalent alternative to the `wait on` that is usually preferred. We will now look at this alternative.

Sensitivity List

HWMod
WS24

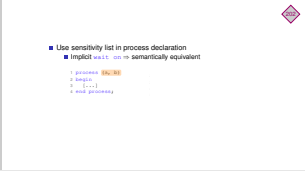
Beh. Mod.
Introduction
Sensitivity
wait on
Sensitivity List
all keyword
Process Simulation
Variables
Remarks

Use sensitivity list in process declaration
Implicit wait on ⇒ semantically equivalent

Instead of writing an explicit `wait on` statement with a sensitivity list as the last element of a process, we can also define the sensitivity list as part of the process declaration itself.

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent



The code snippet shown on the slide demonstrates this for the half-adder implementation you just saw.

Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

This implicitly expresses the respective `wait on` statement implicitly, resulting in sensitivity lists in process declarations and dedicated `wait on` statements being equivalent to each other.

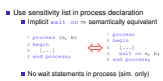
Sensitivity List

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⇔

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```



However, we want to point out that there is a consequence of using a sensitivity list in a process declaration and the explicit `wait on`. In particular, the standard states that processes with a sensitivity list must not contain any explicit wait statements.

Sensitivity List

HWMod
WS24

Beh. Mod.
Introduction
Sensitivity
wait on
Sensitivity List
all keyword
Process Simulation
Variables
Remarks

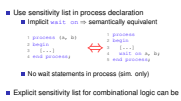
- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⇔

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)



However, before that let us briefly discuss some properties of sensitivity lists.

Sensitivity List

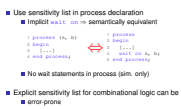
- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⇔

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)
- Explicit sensitivity list for combinational logic can be



Think about a circuit with many inputs that is so complex that you *have* o split it into many multiple processes in order to keep your code readable and maintainable. Obviously, creating sensitivity lists for all of them can be quite error-prone, especially if there are many sensitivities.

Sensitivity List

HWMod
WS24

Beh. Mod.
Introduction
Sensitivity
wait on
Sensitivity List
all keyword
Process Simulation
Variables
Remarks

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

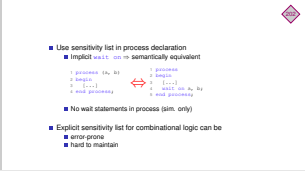
```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⇔

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)

- Explicit sensitivity list for combinational logic can be
  - error-prone



Furthermore, the maintainability of sensitivity lists is also not particularly good. Whenever you rename a signal, you have to rename it in all sensitivity lists as well.

Sensitivity List

HWMod
WS24

Beh. Mod.
Introduction
Sensitivity
wait on
Sensitivity List
all keyword
Process Simulation
Variables
Remarks

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

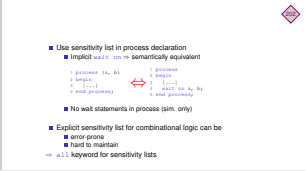
```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⇔

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)

- Explicit sensitivity list for combinational logic can be
  - error-prone
  - hard to maintain



As a remedy the VHDL-2008 standard introduced the `all` keyword for sensitivity lists. We will discuss it on the next slide.

Sensitivity List

HWMod
WS24

Beh. Mod.
Introduction
Sensitivity
wait on
Sensitivity List
all keyword
Process Simulation
Variables
Remarks

- Use sensitivity list in process declaration
  - Implicit `wait on` ⇒ semantically equivalent

```
1 process (a, b)
2 begin
3   [...]
4 end process;
```

⇔

```
1 process
2 begin
3   [...]
4   wait on a, b;
5 end process;
```

  - No wait statements in process (sim. only)

- Explicit sensitivity list for combinational logic can be
  - error-prone
  - hard to maintain

⇒ `all` keyword for sensitivity lists

Sensitivity list is constructed at compile-time
Consider all statements inside the process
Apply rules to determine sensitive signals

By using the `all` keyword, the tools determine all signal sensitivities automatically. This is done by recursively considering all statements in a process and applying the rules defined in the standard to determine the sensitivities of all statements. The resulting sensitivity list is then the union of all these sensitivities.

Sensitivity Lists Using *all*

Beh. Mod.
Introduction
Sensitivity
wait on
Sensitivity List
all keyword
Process Simulation
Variables
Remarks

- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

On the slide you are shown how using this keyword would lookk like for the half-adder circuit and that this is equivalent to the sensitivity list we saw on the previous slide. However, if the tools are able to automatically determine all sensitivities, why shouldn't we always use this?

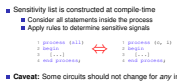
Sensitivity Lists Using *all*

- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

```
1 process (all)
2 begin
3   [...]
4 end process;
```

⇔

```
1 process (c, i)
2 begin
3   [...]
4 end process;
```



Well, there is a caveat with using the `all` keyword, as not *all* ircuits should react to *any* nput change.

Sensitivity Lists Using *all*

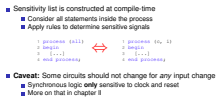
- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

```
1 process (all)
2 begin
3   [...]
4 end process;
```

⟺

```
1 process (c, i)
2 begin
3   [...]
4 end process;
```

- **Caveat:** Some circuits should not change for *any* input change



For example, synchronous circuits are only supposed to change when their clock or reset input changes. Changes of other inputs should never lead to a changed output on their own. We will elaborate further on this in the next chapter where we are concerned with synchronous logic.

Sensitivity Lists Using *all*

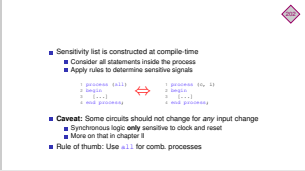
- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

```
1 process (all)
2 begin
3   [...]
4 end process;
```

⟺

```
1 process (c, i)
2 begin
3   [...]
4 end process;
```

- **Caveat:** Some circuits should not change for *any* input change
  - Synchronous logic **only** sensitive to clock and reset
  - More on that in chapter II



For now you can simply remember the rule of thumb that you can use the `all` keyword whenever you write a process that exclusively describes combinational logic.

Sensitivity Lists Using *all*

- Sensitivity list is constructed at compile-time
  - Consider all statements inside the process
  - Apply rules to determine sensitive signals

```
1 process (all)
2 begin
3   [...]
4 end process;
```

⟺

```
1 process (c, i)
2 begin
3   [...]
4 end process;
```

- **Caveat:** Some circuits should not change for *any* input change
  - Synchronous logic **only** sensitive to clock and reset
  - More on that in chapter II
- Rule of thumb: Use `all` for comb. processes

Process is a *sequential* description
Hardware is highly concurrent

If you recall the very first lecture, you might ask yourself what we mean when we state that behavioral modeling allows sequential description of circuits. After all, the hardware we model is still highly concurrent.

Process Simulation

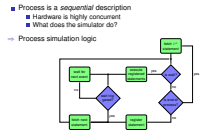
- Process is a *sequential* description
  - Hardware is highly concurrent

Process is a *sequential* description
Hardware is highly concurrent
What does the simulator do?

The thing is that while processes do actually look a lot like sequential programs, the tools do not interpret them this way. Understanding what the tools actually do, given a process, is key when describing the behavior of hardware. We will now address how processes are simulated such that they mimic the behavior to concurrent hardware, leaving the treatment of synthesis for a future lecture.

Process Simulation

- Process is a *sequential* description
  - Hardware is highly concurrent
  - What does the simulator do?

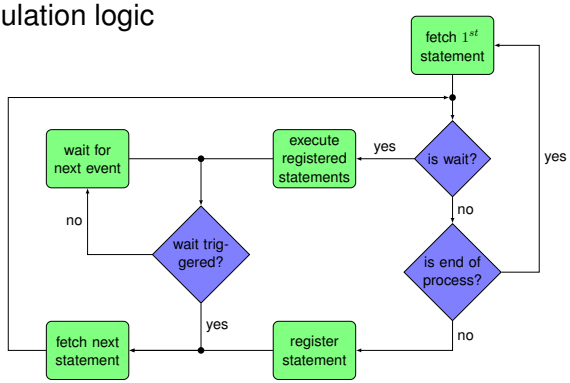


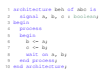
The flowchart on the slide captures the essential process simulation logic used by a simulator. Green boxes express an action taken by the simulator, blue diamonds mark decisions. At such decision nodes the out-going edges are annotated with the respective decision outcome. Since this process simulation logic is quite intricate, we will break it down using an example during the next few slides.

Process Simulation

- Process is a *sequential* description
  - Hardware is highly concurrent
  - What does the simulator do?

⇒ Process simulation logic

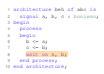




Let us consider the example architecture shown on the slide. It comprises three signals `a`, `b` and `c` and a process that drives `b` and `c`.

Example: Process Simulation

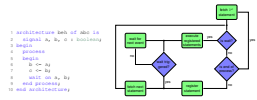

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



If you apply what you heard before, you can immediately observe that the process is sensitive to `a` and `b`.

Example: Process Simulation

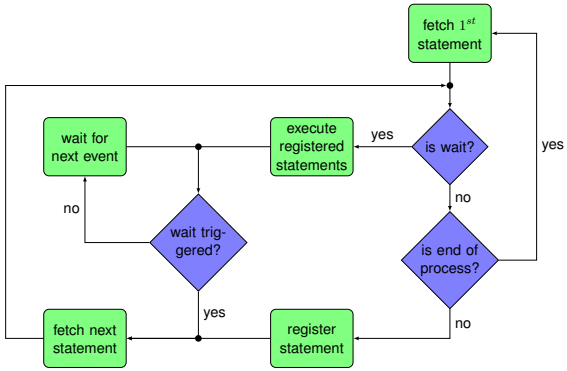

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```

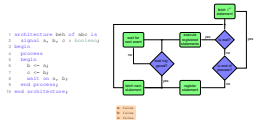


We will now go through the simulation of the architecture's process step-by-step, using the process simulation logic shown in the flow-chart.

Example: Process Simulation


```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```

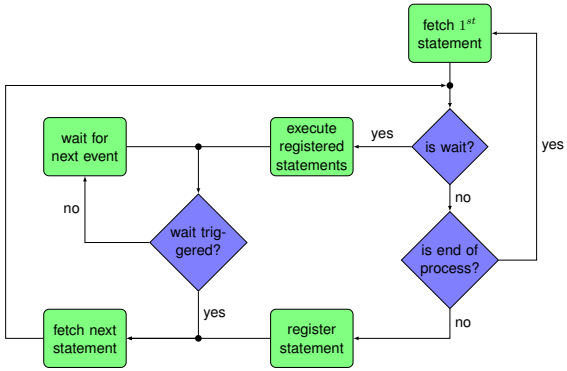




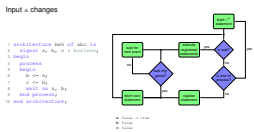
We assume that all signals are initially `false` and that the process is executed for the first time. Note that we will keep track of the current values of all signals in the highlighted area below the flow chart.

Example: Process Simulation


```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



a: `false`
b: `false`
c: `false`

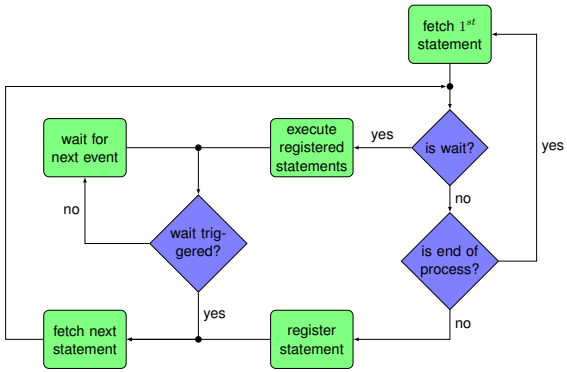


Our example starts with an assumed transition of signal `a` from `false` to `true`. The simulator knows that the process is sensitive to `a` and thus that it is to be executed.

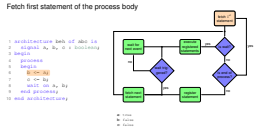
Example: Process Simulation

Input `a` changes

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `false` → `true`
b: `false`
c: `false`

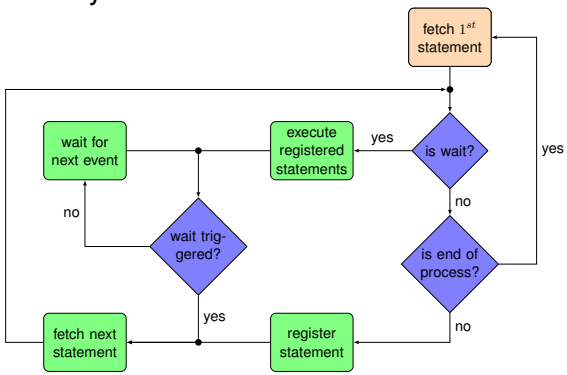


In a first step, the simulator will fetch the first statement of the process body. In our case that's the assignment of `a` to `b`. Note that we highlight the current statement, as well as the current part of the flowchart we consider.

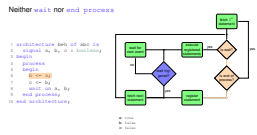
Example: Process Simulation

Fetch first statement of the process body

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

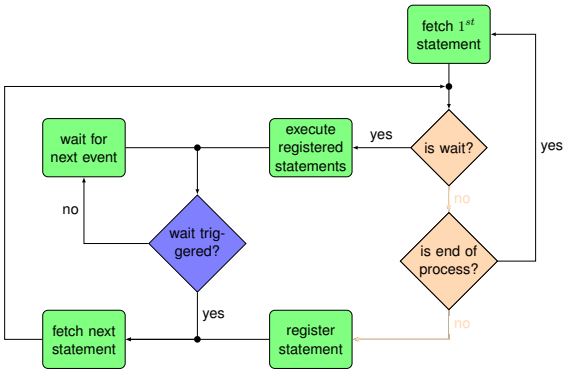


Next, based on the fetched statement the simulator has to decide how to proceed. Since the statement is a signal assignment, and thus neither a wait statement nor the end of the process, it will continue to the *register statement* tate.

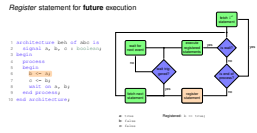
Example: Process Simulation

Neither `wait` nor `end process`

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

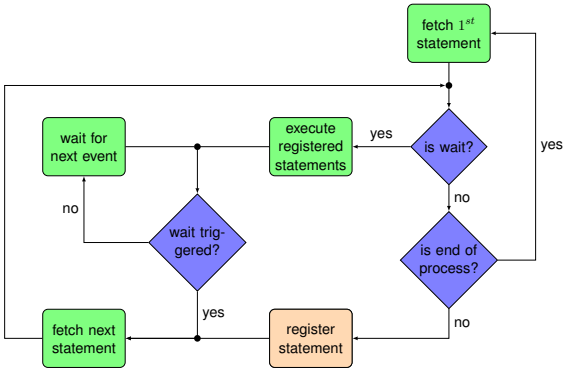


In this state, the simulator will store the fetched statement for *future* xecution. It must be stressed that the assignment does **not take place immediately**. As we will see later, this is required to mimic the concurrent execution of a process.

Example: Process Simulation

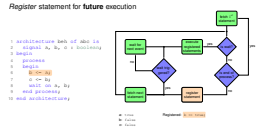
Register statement for **future** execution

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

Registered: `b <= true;`

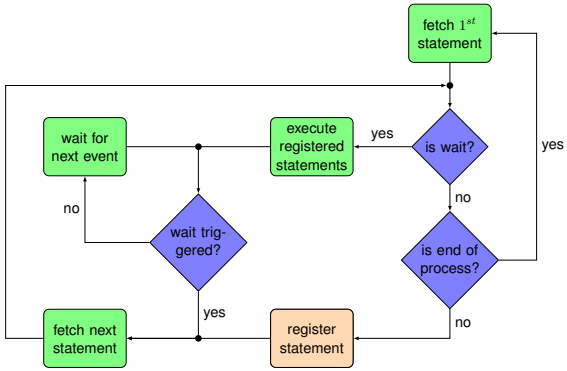


Note that we will track all registered statements below the flowchart.

Example: Process Simulation

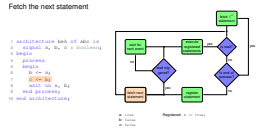
Register statement for **future** execution

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

Registered: `b <= true;`

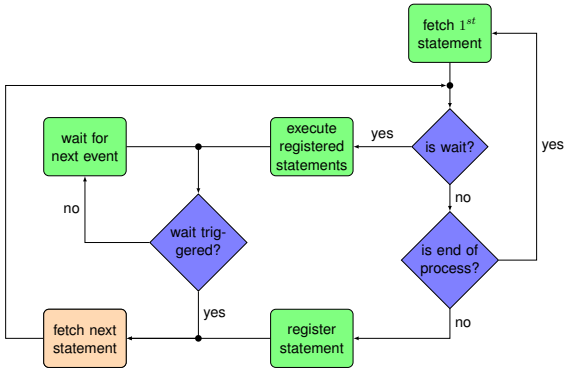


After its registration, the simulator is for now done with this statement and thus continues by fetching the next one. This is the assignment of `b` to `c`.

Example: Process Simulation

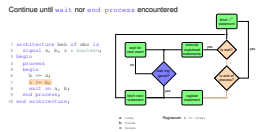
Fetch the next statement

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

Registered: `b <= true;`

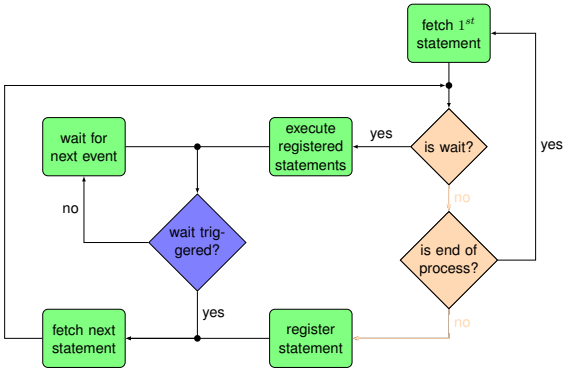


This brings us back to where we started with the first statement where the simulator will again check if the fetched statement is either a wait statement or the end of the process.

Example: Process Simulation

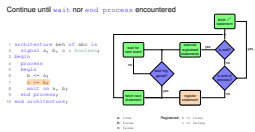
Continue until `wait` nor `end process` encountered

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

Registered: `b <= true;`

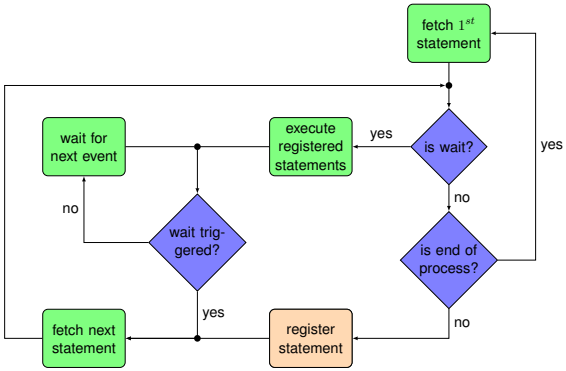


However, since the statement in line 7 is neither, the simulator once again continues by registering the statement without actually performing the assignment. Note how the value of `b` is still `false` because the previous assignment to `b` was not yet executed.

Example: Process Simulation

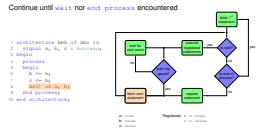
Continue until `wait` nor `end process` encountered

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

Registered: `b <= true;`
`c <= false;`

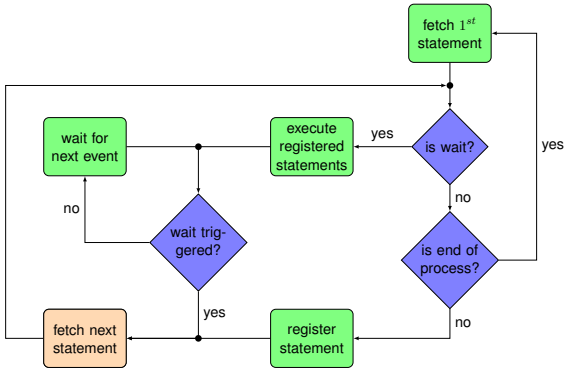


The simulator then continues by fetching the next statement, which is `wait on`.

Example: Process Simulation

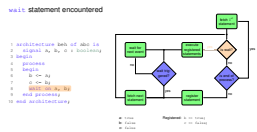
Continue until `wait` nor `end process` encountered

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

Registered: `b <= true;`
`c <= false;`

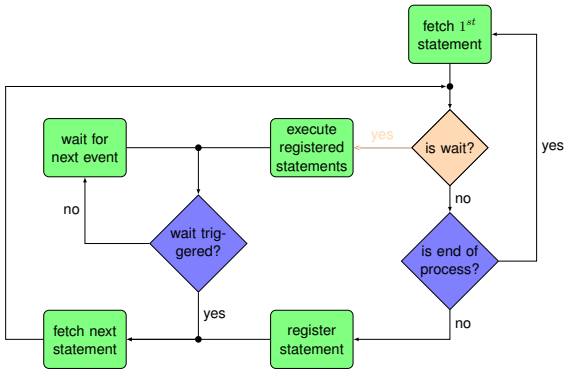


With this statement being a `wait` statement, the simulation of the process transitions to the state where the registered statements are executed.

Example: Process Simulation

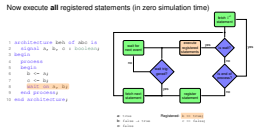
`wait` statement encountered

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `false`
c: `false`

Registered: `b <= true;`
`c <= false;`

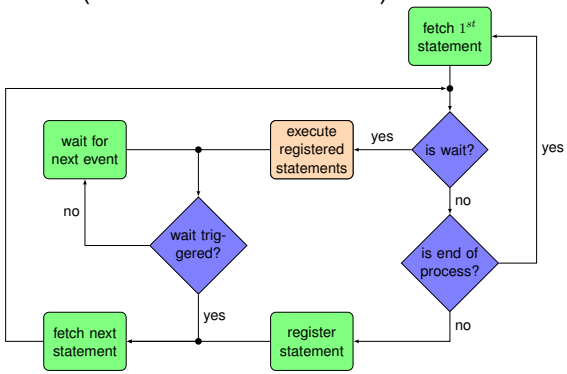


First, the registered assignment of `a` to `b` is executed. Since `a` was `true` during the registration, `b` will become `true` as well.

Example: Process Simulation

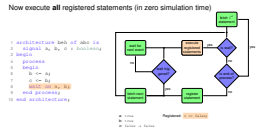
Now execute **all** registered statements (in zero simulation time)

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



a: `true`
b: `false` → `true`
c: `false`

Registered:
```
b <= true;
c <= false;
```

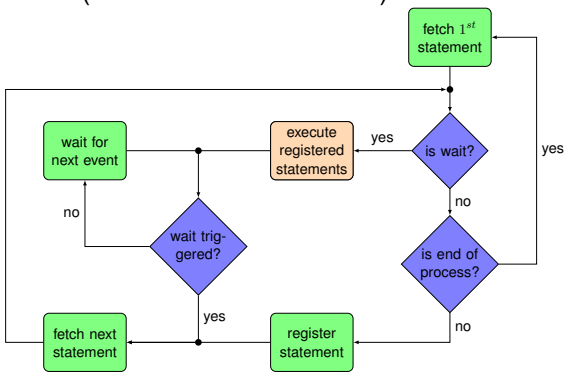


Then, the second registered statement will be executed. This is the assignment of `b` to `c`. However, as `b` was `false` when this assignment got registered, `c` keeps its value of `false`.

Example: Process Simulation

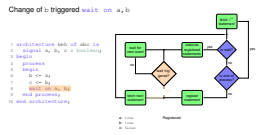
Now execute **all** registered statements (in zero simulation time)

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



a: `true`
b: `true`
c: `false` → `false`

Registered: `c <= false;`

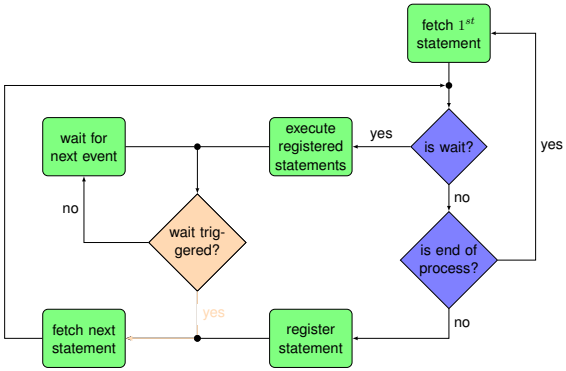


Next, the simulator checks if the execution of the registered assignments has triggered the currently fetched wait statement. In this case it is thus checked whether `b` is on the sensitivity list.

Example: Process Simulation

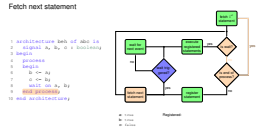
Change of `b` triggered `wait on a,b`

```
1  architecture beh of abc is
2    signal a, b, c : boolean;
3  begin
4    process
5    begin
6      b <= a;
7      c <= b;
8      wait on a, b;
9    end process;
10 end architecture;
```



a: `true`
b: `true`
c: `false`

Registered:

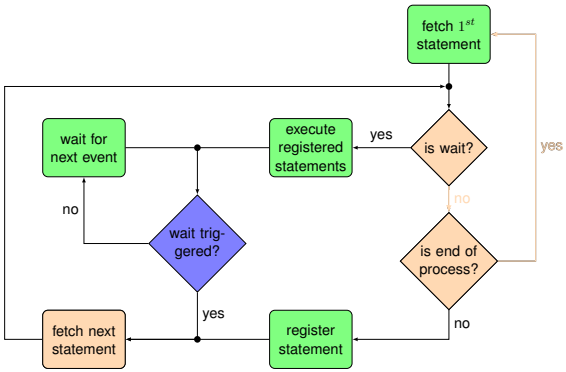


Since this is the case, the next statement is fetched and the process is simulated again with the changed values.

Example: Process Simulation

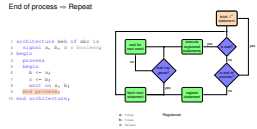
Fetch next statement

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `true`
c: `false`

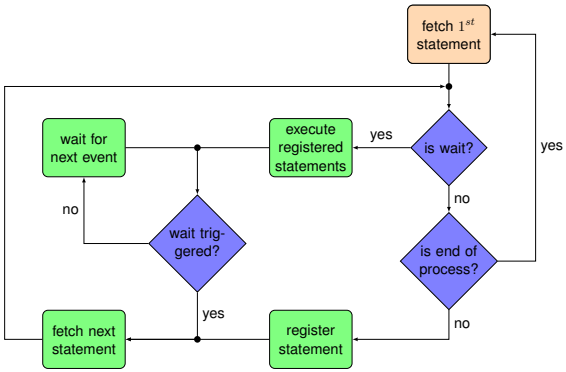
Registered:



Example: Process Simulation

End of process ⇒ Repeat

```
1 architecture beh of abc is
2   signal a, b, c : boolean;
3 begin
4   process
5   begin
6     b <= a;
7     c <= b;
8     wait on a, b;
9   end process;
10 end architecture;
```



a: `true`
b: `true`
c: `false`

Registered:

Observation I
Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.

Let us take a moment to express some observations we were able make during the example process simulation we just saw.

Observations: Process Simulation

Observation I

Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.

Observation I
Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.
Observation II
Without b in the sensitivity list in the example, c would end remaining false until the next change of a ⇒ proper sensitivity list paramount.

The first thing we could notice was that statements are not executed immediately but rather *registered* or future execution. Only when some wait statement is encountered, the registered statements get executed. Admittedly, this seems quite strange. So why is that? Well, the thing is that a process must ultimately be capable to describe a concurrent circuit. Such a circuit does not operate sequentially as suggested by the process. However, the primary problem of a sequential description are actually side effects. Not executing signal assignments mitigates such side effects, as the states of all signals throughout the process execution will only change at the simulation time and thus seemingly concurrent to another. Next, as motivated before, a proper sensitivity list is paramount. We could observe that during this example. If `b` was not on the sensitivity list the process would have become suspended after encountering the `wait on` statement for the first time, leading to a behavior that will most likely strongly deviate from the one the designer had in mind.

Observations: Process Simulation

Observation I

Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.

Observation II

Without `b` in the sensitivity list in the example, `c` would end remaining `false` until the next change of `a` ⇒ proper sensitivity list paramount.

Observation I
Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.
Observation II
Without b in the sensitivity list in the example, c would end remaining false until the next change of a ⇒ proper sensitivity list paramount.
Observation III
A process without (implicit) wait statement loops endlessly.

Finally, looking at the flowchart and knowing what every node inside it means clearly tells us that a process without a wait statement will loop endlessly. Keep that in mind, as wait statements are something beginners tend to forget.

Observations: Process Simulation

Observation I

Statements are not executed immediately, but rather gathered until a wait is encountered. Then they are all executed without consuming simulation time. This mimics a concurrent execution.

Observation II

Without `b` in the sensitivity list in the example, `c` would end remaining `false` until the next change of `a` ⇒ proper sensitivity list paramount.

Observation III

A process without (implicit) `wait` statement loops endlessly.

Signal assignments executed at `wait`

During the previous few slides we could observe that signal assignments only take an effect at wait statements. However, sometimes we would like an assignment to behave like the ones we are familiar with from typical software programming.

Variables

- Signal assignments executed at `wait`

Signal assignments executed at `wait`
⇒ Variables

This is why VHDL also comes with variables in addition to signals.

Variables

- Signal assignments executed at `wait`

⇒ Variables

Such variables can be declared in the declarative parts of processes and some other VHDL constructs, but not within architectures. As shown by the example on the slide, syntactically a variable declaration is, except for the `variable` keyword, equivalent to a signal declaration.

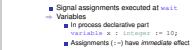
Variables

- Signal assignments executed at `wait`

⇒ Variables

  - In process declarative part

```
variable x : integer := 10;
```



As alread foreshadowed, the key difference between variables and signals is that assignments to variables happen *immediately*

Variables

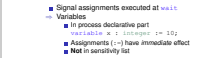
- Signal assignments executed at `wait`

⇒ Variables

- In process declarative part

```
variable x : integer := 10;
```

- Assignments (`:=`) have *immediate* effect



Furthermore, variables can never be on the sensitivity list of a process. If we recall that the elements of the sensitivity list do to some extent correspond to the inputs of the sub-circuit described by the process, this makes sense as variables are declared within a process and thus cannot act as inputs to the respective circuit.

Variables

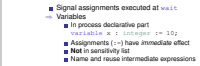
- Signal assignments executed at `wait`

⇒ Variables

- In process declarative part

```
variable x : integer := 10;
```

- Assignments (:=) have *immediate* effect
- **Not** in sensitivity list



In general, variables allows referring to intermediate expressions and results throughout a process body, thus allowing to reuse such values. Variables can thus lead to concise and maintainable code.

Variables

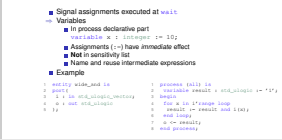
- Signal assignments executed at `wait`

⇒ Variables

- In process declarative part

```
variable x : integer := 10;
```

- Assignments (`:=`) have *immediate* effect
- **Not** in sensitivity list
- Name and reuse intermediate expressions



This is illustrated by the example of a wide AND gate which determines the bitwise AND of the elements of a vector.

Variables

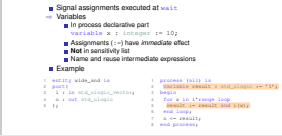
- Signal assignments executed at `wait`

⇒ Variables
  - In process declarative part
    `variable x : integer := 10;`
  - Assignments (`:=`) have *immediate* effect
  - **Not** in sensitivity list
  - Name and reuse intermediate expressions

- Example

```
1  entity wide_and is
2  port(
3   i : in std_ulogic_vector;
4   o : out std_ulogic
5  );
```

```
1  process (all) is
2   variable result : std_ulogic := '1';
3  begin
4   for x in i'range loop
5    result := result and i(x);
6   end loop;
7   o <= result;
8  end process;
```



In the example we declare a variable `result` holds the intermediate results of a loop that ANDs all elements of the given input vector.

Variables

- Signal assignments executed at `wait`

⇒ Variables
  - In process declarative part
    ```
    variable x : integer := 10;
    ```
  - Assignments (`:=`) have *immediate* effect
  - **Not** in sensitivity list
  - Name and reuse intermediate expressions

- Example

```
1  entity wide_and is
2  port(
3   i : in std_ulogic_vector;
4   o : out std_ulogic
5  );
```

```
1  process (all) is
2   variable result : std_ulogic := '1';
3  begin
4   for x in i'range loop
5    result := result and i(x);
6   end loop;
7   o <= result;
8  end process;
```

Let us now consider an example to highlight the difference between variables and signals in processes. While the example is a bit constructed, it is very illustrative.

Example: Variables vs Signals

Consider the process that combines the entity inputs `q`, `b` and `c` to drive two internal signals `w` and `z`. Note that we have ommitted the entity declaration here as it is not vital to the example. The signals `x` and `y` will be used to store intermediate values.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initially, all signals are assumed to be high, with `c` transitioning to low. The process is assumed to have been suspended before.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

As the process uses the `all` keyword in its sensitivity list, and reads `c`, the transition of this signal will lead to its execution. You can now simply apply what you learned about the semantics of signal assignments in processes to determine the values of the architecture's four signals after the process execution.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: A=B=C=W=X=Y=Z='1', C='1'→'0'

First, `a` will be assigned to `x`, which will thus be set to `'1'`. However, note that we do not explicitly keep track of registered statements, but rather let later assignments override previous ones. This will become clear shortly.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1'
2	

Next, `b` and thus `'1'` is assigned to `y`.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='1'`
2	

Then, z is set to the logical AND of x and y.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: A=B=C=W=X=Y=Z='1', C='1'→'0'

Iteration #	signal values at end
1	x='1', y='1' z='1'
2	

Now something interesting happens. `c` is assigned to `y`, overriding the previous assignment to `y`.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1'
2	

Finally, `w` is assigned the logical AND of `x` and `y`. Note that since the assignments have not yet taken place as there was no wait statement, `w` will become `'1'`. Now, since the process is sensitive to `y`, which changed its value, the process will trigger again.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	

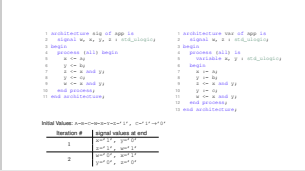
The values of the signals after this second iteration are shown on the slide. In case you have doubts about some values, you can simply determine them step-by-step as we did in the first iteration. Since none of the signals to which the process is sensitive to changed in this second iteration, the process will again be suspended.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`



Let us now move our attention to a slightly modified version of this process.

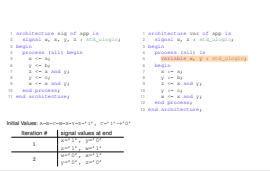
Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: A=B=C=W=X=Y=Z='1', C='1'→'0'

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'



Instead of signals for the intermediate values `x` and `y`, corresponding variables are declared and used within the process.

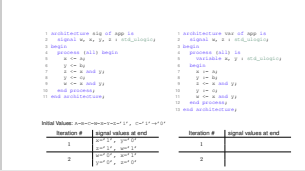
Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`



As before, let us now consider the effects the statements inside the process body have on the signals step-by-step.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: A=B=C=W=X=Y=Z='1', C='1'→'0'

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	
2	

First, the two variables `x` and `y` are assigned `a` respectively `b`. Since variable assignments take place immediately, both variables will from now on hold the value `'1'`.

Example: Variables vs Signals

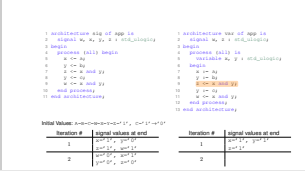

```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`

Iteration #	signal values at end
1	`x='1', y='1'`
2	



As before, `z` is now assigned the value `'1'` as both operands of the AND are high.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	x='1', y='1' z='1'
2	

Now the variable `y` is aassigned the value of `c`. Since this assignment takes place immediately, and since `c` is `'0'`, `y` will from now on be low. This is in contrast to before we `y` was `'1'` during the whole process execution since the signal assignments were only registered but did not take place until the final implicit `wait on` statement.

Example: Variables vs Signals

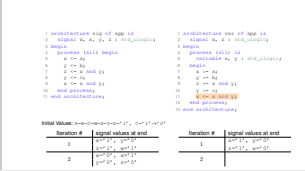

```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1'`
2	



As a result, the value of `w` ends up being `'0'` rather than `'1'` as was the case before.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='1'`
2	`w='0', x='1'` `y='0', z='0'`

Iteration #	signal values at end
1	`x='1', y='0'` `z='1', w='0'`
2	

Finally, since no signal to which the process is sensitive to changed, the process will be suspended after the first iteration.

Example: Variables vs Signals

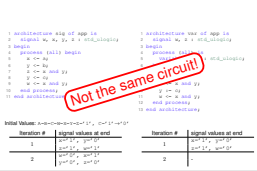

```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

Iteration #	signal values at end
1	x='1', y='0' z='1', w='0'
2	-



It should be evident by now that the two processes do **not** describe the same circuit.

Example: Variables vs Signals


```
1  architecture sig of app is
2    signal w, x, y, z : std_ulogic;
3  begin
4    process (all) begin
5      x <= a;
6      y <= b;
7      z <= x and y;
8      y <= c;
9      w <= x and y;
10   end process;
11 end architecture;
```

```
1  architecture var of app is
2    signal w, z : std_ulogic;
3  begin
4    process (all) is
5      variable x, y : std_ulogic;
6    begin
7      x := a;
8      y := b;
9      z <= x and y;
10     y := c;
11     w <= x and y;
12   end process;
13 end architecture;
```

Not the same circuit!

Initial Values: `A=B=C=W=X=Y=Z='1', C='1'→'0'`

Iteration #	signal values at end
1	x='1', y='0' z='1', w='1'
2	w='0', x='1' y='0', z='0'

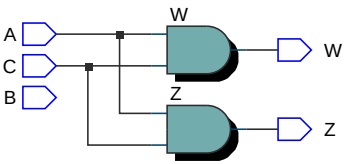
Iteration #	signal values at end
1	x='1', y='0' z='1', w='0'
2	-

To further illustrate that the two processes from the previous example do not model the same design, the two images on this slide show the resulting circuits when running the two versions through a synthesis tool.

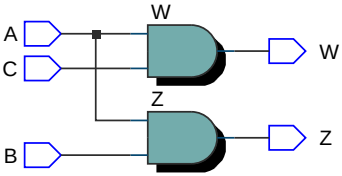
Variables vs Signals (Cont'd)

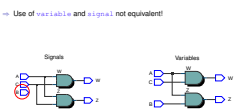
⇒ Use of `variable` and `signal` not equivalent!

Signals



Variables



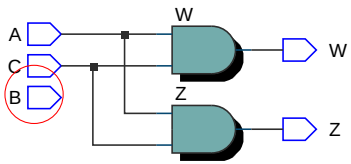


Note how the code using signals results in a circuit that does not use the input `B`.

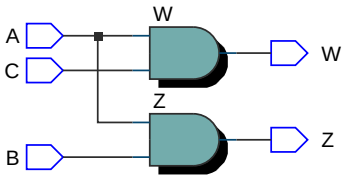
Variables vs Signals (Cont'd)

⇒ Use of `variable` and `signal` not equivalent!

Signals



Variables



Finally, we would like to end this lecture with a few remarks about behavioral modeling.

Remarks

First, for the sake of completeness, now that we know all its elements it is about time to show you how a process is declared. We did not do this so far as it is very similar to many declarations you saw already, and the sensitivity list would previously have been confusing.

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

Next, we want to stress that every concurrent signal assignment has an equivalent expression to be used inside a process.

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process

During this lecture we deliberately included some examples highlighting that, like the 4:1 multiplexer, or the half-adder which you already knew from the entity-architecture lecture. Furthermore, we also showed you a behavioral model of a wide AND-gate, which you previously encountered in the structural modeling lecture.

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate

However, note that the other direction is not true, meaning that there are circutis which can describe using behavioral modeling but **not** using concurrent signal assignments. A particularly important class of such circuits is synchronous logic which we will cover in-depth in future lectures.

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate
  - The other direction is not true (e.g., sync. logic)

Finally, although we restricted ourselves to a single process for the purpose of keeping things short and educational, you can in general use arbitrary many processes in an architecture.

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate
  - The other direction is not true (e.g., sync. logic)
- Arbitrary many processes possible

These processes are executed concurrently in the manner we saw during this lecture.

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate
  - The other direction is not true (e.g., sync. logic)
- Arbitrary many processes possible
  - Executed concurrently

However, be aware that the order in which processes are executed is not defined by the VHDL standard. Hence, you should not rely on a particular order being used by a simulator.

Remarks

- `process` declaration (simplified)

```
1 [label] : process designator [(sensitivity_list)] [is]
2   [declarative_part]
3 begin
4   [statement part]  -- process body
5 end process;
```

- Every concurrent signal has an equivalent process
  - Examples: MUX41, halfadder, wide AND-gate
  - The other direction is not true (e.g., sync. logic)
- Arbitrary many processes possible
  - Executed concurrently
  - Order of actual execution *undefined* $\Rightarrow$ do not rely on it

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

Lecture Complete!