ELEC-H-409
Th02: Concurrent and sequential statements

Dragomir Milojevic
Université libre de Bruxelles

Today

1. Signal assignments & simulation
2. Example of signal assignments
3. Concurrent assignments – processes
4. Assignments & simulation
5. Conditional statements
6. VHDL trickiness example
7. Practical examples

1. Signal assignments & simulation

Signal assignment 1/3

- Assignment can be seen as a "piece of wire" that is used to connect two or more pins: one source & one (**or more**) sinks pins
  - If there is more than one sink pin, it means that the net has a FanOut (FO) > 1
    - What are the electrical implications?
- Signal assignment appears **within** the architecture specification of a module, after `begin` keyword and **inside** or **outside** a **process** statement (we saw this):

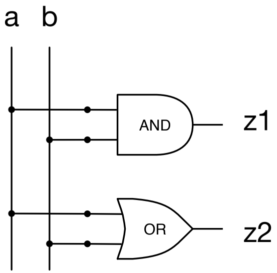
```
    a <= b;
```

- Assignment statements define new driver for the assigned signal
  - In the above assignment, `b` is a new driver for the signal `a`
  - Driver cold be input or some internal signal generated by some logic
  - Electrically speaking, what is driver and what we need to make sure for the real physical circuit?

Signal assignment 2/3

- VHDL module can contain arbitrary number of assignments
- Since any digital logic circuit is concurrent by nature, all of these assignments should occur simultaneously, that is in parallel
- In the example below, both logic functions `z1` and `z2` are evaluated in parallel
  - ▹ If any of the inputs change, the output will appear after some delay, a sum of signal propagation delay in the wires, plus the switching delay of the logic gates (or immediately if we neglect the delays)

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity test is port (
   a, b, c : in std_logic;
   z1, z2  : out std_logic
);end test;

architecture syn of test is
   begin
      z1 <= a and b;
      z2 <= a or b;
end syn;
```



Signal assignment 3/3

- Depending on what we are doing, the assignment `a <= b` will have two VERY different meanings:
  - ▷ Synthesis – this is during actual circuit generation
    - ◦ Assignment will mean that `b` is **connected** to `a`; This is why assignments could be seen as pieces of wire connecting two or more ends
  - ▷ Simulation – this is during logic circuit verification and using some SW simulation tool
    - ◦ Assignment means that the value of `b` is **assigned** to the value of `a` (in fact during simulation `a`, `b` are vectors, but more on this later on)
- During circuit (model) simulation the assignment is not done **instantaneously**, when the VHDL statement is issued, but at the end of a time window in which the current VHDL module is being simulated
- The above is often MISUNDERSTOOD, because of the software background people get before starting to learn HDLs

Signal assignment – the good way

- Multiple assignments are (of course) allowed, but only if they target **different outputs**; and if this is the case, they are then performed in **concurrent** fashion (i.e. in parallel)
- Since assignments are concurrent, the order in which assignments are written in VHDL model is irrelevant, models below are equivalent:

```
architecture syn of block is
   begin
    c <= a ;
    z <= t ;
    d <= b ;
end synthesis1;
```

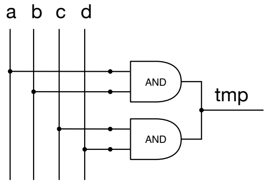
```
architecture syn of block is
   begin
    d <= b ;
    c <= a ;
    z <= t ;
end synthesis1;
```

- This is because all assignments are “executed” at the “same time”
  - ▹ Attention we use word *execution*: logic circuit is about gates, wires & flops and assignment/wiring, and not a computer & SW!
  - ▹ In this context *execution* refers to **simulation** where all signal assignments are processed in the same simulation cycle
  - ▹ They appear as concurrent even if they are processed in different **computer** clock cycles!

Signal assignment – the wrong way 1/3

- Concurrent assignment statements are used for modeling & synthesis of digital logic circuits with multiple outputs
- Several concurrent assignments **could** be applied to the **same** signal in an architecture
- So multiple drivers CAN and will be created for that signal (signal `tmp` in the model below)
- The following model is acceptable from syntax point of view:

```vhdl
... -- some definitions before

architecture syn of block is
   begin
    tmp <= a and b ;
    tmp <= c and d ;
    -- Two statements target same output
end syn;
```



- But the above model is VERY WRONG and you should understand why – one of (few!) very important things in VHDL

Signal assignment – the wrong way 2/3

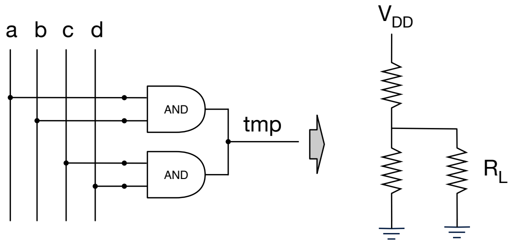
Simulation level

- In the above model and during simulation there will be two drivers (sources) for one signal (sink); question is what will be the value of the output?
- When multiple drivers target the same output (or local) signal a resolution function needs to be given so that the simulation tool can decide on the value of the output
- Two options for the resolution function:
  - ▷ Use a default resolution function
    - ◦ For `std_logic` the resolution function could point to undefined value
    - ◦ So, if you simulate the previous model you will not see '0' or '1' !!!
    - ◦ Undefined signals are pointed out by simulation tools, e.g. Model/QuestSim will use a red wave to indicate this
  - ▷ Use custom resolution function written by the designer (i.e. you)

Signal assignment – the wrong way 3/3

Circuit level

- The outputs of two AND gates are shorted !!!
- Voltage at the output will depend on the internal resistance values of transistors used to build gates



- These have some real life values, and typically the circuits with transistors do have extra resistors to limit excessive currents and avoid device destruction (e.g. the short above)
- So, in the real circuit the output value will be set to something between TRUE & FALSE depending on the voltage divider made by the above resistors
- Such output could be used by the next logic stage, to produce some totally wrong output !!!

Signal assignment within a process

- Multiple assignments (drivers) to the same output are little bit different when used within a `process` (until now we have looked into combinatorial assignments, so assignments outside the process)
- Because in a VHDL `process` statements are processed sequentially by the simulator, only the last assignment will be taken into account during simulation

```
1 architecture syn of test is
2    begin
3     process (a,b,c)
4     d1 <= a and b ;  -- this assignment addresses output d1
5     d2 <= b and c ;
6     d1 <= a and c ;  -- this one too !!! previous will be overridden
7 end syn;
```

- When simulating the above VHDL, the assignment on `d1` at line 4 will have no effect! Only the assignment at line 6 will do something
  - ▹ Note that what happens here is TOTALLY different from the SW, where the above sequence of statements would make a perfect sense; This is not SW, but HW!

Signal assignments: combinatorial and process

- If concurrent (combinatorial) assignments target same output, the value will be undefined during simulation, and at circuit level it will be something unpredictable

```vhdl
architecture syn of block is
   begin
    d1 <= a and b ;
    d2 <= b and c ;
    d1 <= a and c ;
end syn;
```

- If concurrent (sequential) assignments (within a `process`) target same output, only the last assignment will be taken into account

```vhdl
architecture syn of block is
   begin
    process (a,b,c)
    d1 <= a and b ;
    d2 <= b and c ;
    d1 <= a and c ;
end syn;
```

- You should always have this in mind!

Signal assignment – general form

- So far we have assumed no timing information during assignment operations, which was good enough for functional simulation when we are interested in Boolean correctness only
- VHDL allows definition of explicit delays that can be added to the model and will be processed by the simulator to enable timing simulation of a circuit
- There are two types of delays associated to assignment statement:
  - ▷ Inertial delays (default implementation):

```
a <= b after time_expression;                    -- inertial delay
```

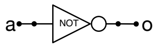
  - ▷ Transport delays:

```
a <= transport b after time_expression;          -- transport delay
```

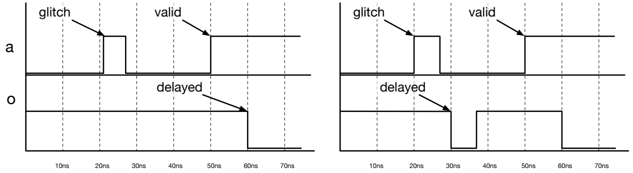
  - ▷ In the above `time_expression` is specified using absolute time (e.g. `5ns`); during simulation this value is **relative** to some reference

Inertial vs. transport delay

- Inertial delays model the fact that logic signals **must** have a stable value for a certain amount of time; any change in the signal that is shorter than that amount of time will be considered as a glitch – a non-desired short impulse
- Transport delays model the fact that both wires and gates have a certain delay, that needs to be taken into account before observing the output



```
o<= not a after 10ns

o<= transport
    not a after 10ns
```



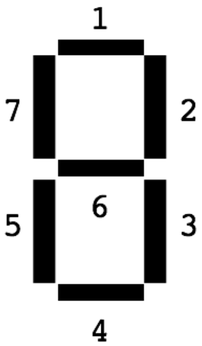
Inertial **Transport**

- Previously we have used simple assignment statements with no delay specifications; this is the most common use of them, and this is what you should use; simulation tools will anyhow assume certain amount of default delay called delta-delay (more on it in Section 4)

2. Example of signal assignments

Seven segment display

- Design digital system that should display one decimal digit at a time (from 0 to 9) on a 7-segment display
- Input of the circuit is a single binary encoded decimal digit
- Output(s), when set to 1, turn one (or more) segments (LEDs) on, off otherwise

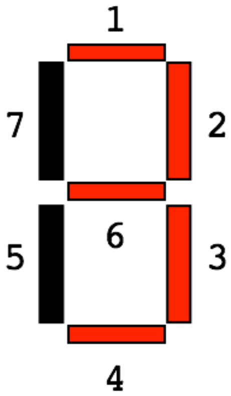


- Few questions before going any further:
  - ▷ How many bits are used as input in this case, and how do you compute this?
  - ▷ How many bits are necessary to describe the system output & how do you figure out this one?
  - ▷ What type of circuit this is (combinatorial or sequential)?

Truth table(s) and logic functions

For number 3 segments 5 and 7 are turned off (they are set to 0), all the others are on

Decimal	Binary	f1	f2	f3	f4	f5	f6	f7
0	0000	1	1	1	1	1	0	1
1	0001	0	1	1	0	0	0	0
2	0010	1	1	0	1	1	1	0
3	0011	1	1	1	1	0	1	0
4	0100	0	1	1	0	0	1	1
5	0101	1	0	1	1	0	1	1
6	0110	1	0	1	1	1	1	1
7	0111	1	1	1	0	0	0	0
8	1000	1	1	1	1	1	1	1
9	1001	1	1	1	1	0	1	1
10-15		-	-	-	-	-	-	-



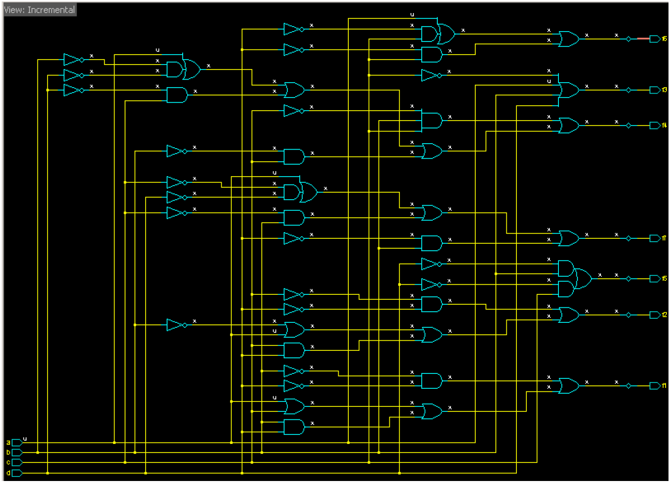
What happens for inputs going from 10 to 15?

1st implementation

- With 7 truth tables, with four input variables each, we can manually derive logic functions and encode them as a combinatorial circuit
- Logic functions can be (manually) optimized using one of the known methods (K-Maps, Quine-McCluskey), or not!
- Why we could leave logic functions non-optimized?

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity seven_seg is port(
a, b, c, d                  : IN  std_logic;
f1, f2, f3, f4, f5, f6, f7   : OUT std_logic;
);end seven_seg;

architecture synthesis1 of seven_seg is
begin
  f1<= a or c or (b and d) or (not b and not d);
  f2<= a or (not b)  or (c and d) or (not c and not d);
  f3<= a or (not c) or b or d;
  f4<= a or (not b and not d) or (c and not d) or (not b and c) or (b and not c and c);
  f5<= (b and not d) or (c and not d);
  f6<= a or (c and not d) or (c and not d);
  f7<= a or (not c and not d) or (b and not c) or (b and not d);
end synthesis1;
```

Resulting circuit



This approach is not ideal, why?

2nd (much better!) implementation

Using **`with`** `identifier` **`select`** for assignment and encode the truth table in VHDL directly

- Logic function synthesis & optimization will be performed automatically
- This type of optimization will be done very well: today synthesis & optimization software tools are GOOD
- But do not generalize this! Poor VHDL model will not result in a good circuit, no matter how good the synthesis tool is

```vhdl
library ieee;
library ieee;
use ieee.std_logic_1164.all;

entity seven_seg is port(
 dataIn : in std_logic_vector(3 downto 0);
 segs : out std_logic_vector(7 downto 0)
);end seven_seg;

architecture synthesis1 of seven_seg is
 begin
  with datain select
   segs <=
     "10000001" when "0000", -- 0
     "11001111" when "0001", -- 1
     "10010010" when "0010", -- 2
     "10000110" when "0011", -- 3
     "11001100" when "0100", -- 4
     "10100100" when "0101", -- 5
     "10100000" when "0110", -- 6
     "10001111" when "0111", -- 7
     "10000000" when "1000", -- 8
     "10000100" when "1001", -- 9
     "11111111" when others;
end synthesis1;
```

Comparing two models

- While both circuits have the same functionality, they could be implemented differently during circuit synthesis process
- First description is close to gates, while second description looks like a multiplexer or a Look-Up-Table ...
- How the final circuit will be implemented depends on the synthesis tool and target technology, eg.:
  - ▹ Combinatorial circuit as set of gates (ASIC)
  - ▹ Combinatorial circuit as set of truth tables (FPGA)
  - ▹ Mux circuit if such standard cell primitive exists in your technology
- Different physical circuits also mean different area, performance and power characteristics
- For now do not bother much with that, make your circuits functionally correct first & optimise later!

3. Concurrent assignments – processes

Concurrent statements (complete picture)

Different types of concurrent statements:

- Concurrent assignments – we have just spoke about these:
  `a <= b;`
  and we will see this in even more details in the next lecture
- Assert – used to display messages during simulation

```
assert not (R = '1' and S = '1')
report "Both signals R and S have value '1'"
severity error;
```

  - ▹ Above statements will not be synthesized as such, only used to monitor the simulation process
  - ▹ In the model above `report` will print something
  - ▹ Where will you see the printed message?
- Component instantiation – how to manage complex designs by creating design hierarchy (more on this also later on)
- Process – we have seen the basics, but let's go into more details

What is really `process`?

- A process is a sequence of VHDL statements **"executed"** in the **order specified by the VHDL source**
- It is said that the statements are **"executed sequentially"**
- You should recall what we have said in the previous chapter and understand well what does this REALLY means (this can be confusing, I do agree...)
- This sequentiality has nothing to do with sequential circuits (although very remotely this is true, since any computer is a sequential logic circuit)
- Statements are not "executed" per se, notion of the execution refers to the execution of the computer program that performs circuit simulation (here we design HW, remember?)
- Simulation and synthesis do appear as two different things
- Designers want a match between the two

Process: usage

- Look at the example on the right: 3 processes are concurrently "executed" within the same HW module
- The same goes for any other processes defined in other modules
- Logical grouping of processes in one module is the designers choice
- All processes within one module have access to all inputs of the module and all temporary signals; this is something like global/local variables in SW, but let's avoid direct comparisons
- During simulation, different processes, even if they are declared in the same module, will appear as a separate event driven piece of SW entity

```
library ieee;
use ieee.std_logic_1164.all;

entity myBlackBox is port(
    -- IO definition
);end myBlackBox;

architecture syn of
      myBlackBox is
begin

 p_1 : process(...)
  begin
  -- sequential machine 1
 end process;

 p_2 : process(...)
  begin
  -- sequential machine 2
 end process;

 p_3 : process(...)
  begin
  -- sequential machine 3
 end process;
end syn;
```

Process declaration structure

- Process may appear anywhere in the architecture body (but after `begin` keyword associated with the architecture definition)
- General form:

```
[name:] process [(sensitivity_list)]
[type_declarations]
[constant_declarations]
[variable_declarations]
[subprogram_declarations]
begin
...
  sequential_statements
...
end process [name];
```

- May contain optional name
  - ▹ Process name is not used for synthesis of the logic circuit
    - ◦ What the circuit will do with it?
  - ▹ Process names are useful during circuit simulation; when we have multiple processes instantiated within the same VHDL module, we can easily trace them down
  - ▹ Suggestion: always name your processes

Process and sensitivity list

- Process can contain optional sensitivity list – list of signals to which the process is said to be sensitive
- Some synthesis tools ignore sensitivity lists, i.e. the same circuit description with different sensitivity list will still produce the same circuit (complete vs. partial or non-existent sensitivity list)
- In the example below both processes, if synthesized, could lead to the same gate-level netlist: simple three input AND gate

```vhdl
proc1: process (a, b, c)
   begin
      x <= a and b and c;
end process proc1;
```

```vhdl
proc2: process (a, b)
   begin
      x <= a and b and c;
end process proc2;
```

- It the simulation tool takes the sensitivity list into account, the right hand model will not trigger on change of `c`, while the model on left will do so!
- This means we have a difference between the simulation and the circuit !!! This is not wanted ...

4. Assignments & simulation

Logic circuit simulation problem

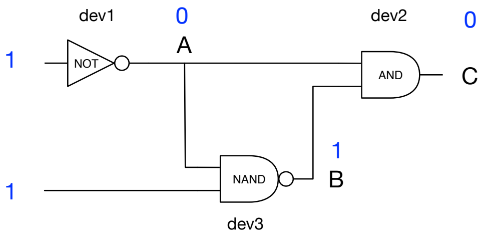
- Digital circuits are concurrent by nature, every implemented logic function has the life of it's own, independent from the others (they work in parallel)
- We want to simulate operation of the digital logic circuit using a computer that is sequential by nature, i.e. it executes instructions in a given order, one after the other in a sequence of time steps, clock cycle after the clock cycle
- Fundamental question to ask is:

  How to simulate concurrent behavior of a digital logic circuit, with a sequential computer?

- This is not a trivial problem and can be found in many different places, basically whenever computers need to interface with the real world and where the concurrency is vital
- Also do not be tempted to think that today, as we leave in a many-core world, the problem is solved; even if you would run the simulation on a cluster of computers with many CPUs and cores per CPU

Computer simulation problem illustration 1/3

- Designer wants to simulate the circuit below with a computer:

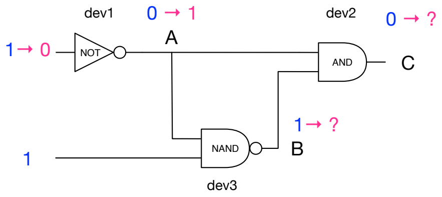


  - ▷ The circuit is bogus, can you say why?

- Imagine that the input at the instance `dev1` of the inverter `INV` changes from '1' to '0'
- Logic value at point `A` will change and should propagate through instances `dev3` (`NAND2`) AND `dev2` (`AND2`) to reach `B` and finally the output `C`

Computer simulation problem illustration 2/3

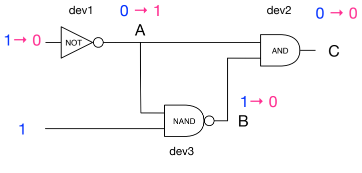
- Since the computer is sequential, say **only one gate could be evaluated at a time** (CPU executes 1 instruction per cycle)
- This would mean that in our example we should pick one of the two instances to process first (`dev2` or `dev3`)
- **General problem** – For a circuit of an arbitrary size we need to decide on the order in which the gates will be evaluated
  - ▹ Figuring out the order in our example is simple for a human, one can follow natural data (signal) flow from left to right (first `dev3` and then `dev2`) – but computers do not "see" this!



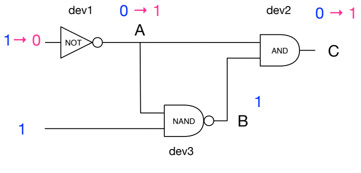
The order of the simulation is crucial, wrong order → wrong results!

Computer simulation problem illustration 3/3

- First `NAND2` (`dev3`) is evaluated before `AND2` (`dev2`), so:
  - ▷ `A: 0 → 1, B: 1 → 0, C:0 → 0`
  - ▷ In another words `B` is evaluated first `1 → 0`
  - ▷ `C` get's the right value on input, all is ok



dev3 first



dev2 first

- But then assume `AND2` (`dev2`) before `NAND2` (`dev3`), so:
  - ▷ `A: 0 → 1, C: 0 → 1` `C` is not the same !!!
  - ▷ This is because we have `A=1` and B^{-}`=1`, this `B` being the old value; but then if `B: 1→ 0` triggered, `C: 0 → 0` is now right
  - ▷ To get right `C` we need re-evaluate `B` a second time
- Who should be responsible to decide when a gate needs a re-evaluation in a multi-million gate design?

Logic simulation with computers – the solution

Implement in SW & execute on a general purpose computer event based simulation:

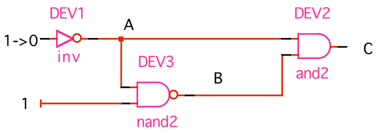
- To each component of the HW model (e.g. gate, process) we attach a separate piece of executing software
- Each piece of software is evaluated (**triggered**) if its input changes; any input change is seen as an event and will be processed accordingly by the simulator (hence the name)
- After evaluation of a single functionality (e.g. gate), different outputs generated by this gate (SW) are stored & processed to eventually become events to trigger other pieces of software (i.e. connected components)
- Nobody knows in advance how many of these evaluations will be needed!
- Their number will depend on the circuit structure and input data sets; to allow this simulation time steps need to be **"flexible"**

Delta cycles

- To allow this flexible *"rubber time"* we use the notion of delta-delay (Δ-delay)
- Δ-delay – *infinitesimal time interval in which future assignments are done*
- Note that this delay does not have anything to do with the actual circuit and actual time, it is just a way to discretize time internally in the simulator and enable event base simulation with as many steps as needed to produce the right simulation value
- Because time is discrete in digital systems we also speak of delta-cycle (Δ-cycle)
- Actual sequential circuit clock cycle will be "decomposed" in a necessary number of Δ-cycles to complete the simulation; the number of cycles will depend how many events will be generated, i.e. circuit and input data vector

Event based simulation with Δ-cycles

- Change on inputs triggers simulation relative to INV, producing the output change scheduled for next Δ-cycle
- Inverter process is suspended since no events on the inputs, so we move to next Δ-cycle
- During the next Δ-cycle either dev2 or dev3 can execute
- If dev3 scheduled 1st (NAND), next Δ-cycle updates the value of C
- If dev2 scheduled 1st, dev3 process re-triggers dev2 in the next Δ-cycle since there is an event on B, illustrated in the table below:



AND' – first evaluation

AND" – second evaluation

Time	Δcycle	Event
0ns	1	in: $1 \rightarrow 0$
		eval INV
	2	A: $0 \rightarrow 1$
		eval NAND, AND'
	3	B: $1 \rightarrow 0$
		C: $0 \rightarrow 1$
		eval AND"
	4	C: $1 \rightarrow 0$
1ns		

Simulation & real-circuit

- In real circuits the order of "execution" will exist too, and `dev2` and `dev3` could evaluate in any order
- The difference is that in the real circuit after some time of electrical transient things should (& will) settle to the right value
- So, the values will bounce here and there, due to propagation delays in gates & wires, and this is why **we always have to wait for certain time** to get the output stabilized to the right value
- This is true for all digital circuits including combinatorial ones, despite the fact that they are not subject to **race conditions**
- Event based simulation is a trick that allows us to model & simulate real-circuits, concurrent by nature, with **sequential computers** (no matter how parallel they are ...)
  - ▹ Think! Current integrated circuits can have tens of millions of gates, so tens of millions of *"separate pieces of executing software"* that need to be processed ... so there are orders of magnitude of difference!

Circuits and simulation – computer view

- We know that computers are sequential in nature (in a sense that they execute one instruction after another); to simulate digital circuits we need some kind of mechanism to emulate concurrent systems – event based simulation and Δ-cycles
- Question is how do we implement this in SW?
- HW model of a gate, process etc. will be translated in some kind of SW process or thread; What is process and thread in an OS?
- When simulating HW we do not use typical SW process and threads since their control is not flexible; rather any HW simulator will implement it's own notion of HW process/thread
- These HW processes will be controlled by the simulation kernel, (so not OS), and when we say "process is executed", we refer to execution of the **SW model by the simulation kernel running on the top of an OS**
  - ▹ Simulation kernel is responsible of launching, suspending, and scheduling of all processes that are describing our HW system

Processes & event based simulation

- The SW view of HW process **once launched, never ends**: like if it was executed in an infinite loop!
- Process will definitively stop only when the simulation ends
- However in order to serve all processes that describe our system, every process is suspended for execution at some point in time, so that the CPU on which the simulator runs can share his time among all the other processes in the system (our CPU can do only one thing at a time)
- For a process that has a sensitivity list, process will be automatically suspended for execution when the last statement in that process has been processed by the simulator
- Execution of this process will resume only if and when a signal that is described in the sensitivity list changes value (this will create an event, why we call this event driven simulation); if not, the process remains suspended

Different signal assignments

- In case of concurrent assignment statements, the change of any signal that appears on the right-hand side of the assignment symbol activates the assignment execution, without explicitly specifying a sensitivity list
- In the case of sequential assignment statements, i.e. processes, signal assignment is determined:
  - ▷ by the change of a signal in their sensitivity lists or
  - ▷ by encountering a `wait` statement (we are going to see this next)

  → wait for an event
- Activation of an assignment statement is independent of activation of other concurrent statements within the architecture
- As opposed to **concurrent** assignments, if a process contains several **sequential** assignments to the same signal, only the last assignment will be effective!

Using wait to control process

- Instead of a sensitivity list process can contain $\boxed{\texttt{wait}}$ statement
- This is an explicit order for simulator to:
  - ▷ suspend the execution of a process currently in execution
  - ▷ and specifies a condition that will resume the suspended process

```vhdl
architecture synthesis of myBlackBox is
   begin
      proc3: process
         begin
            x <= a and b and c;
            wait on a, b, c;     -> condition
      end process proc3;
end synthesis;
```

- Sensitivity list and wait statements are **mutually exclusive**
- We will see later which one you should use where ...
- In resumed process, statements are executed until another wait is encountered, thus multiple waits are possible in a single process

Different flavors of `wait` 1/3

More generally `wait` statement has three different forms:

1. `wait on` – processes is suspended until some signal change; this is equivalent to the use of a sensitivity list in the process statement
2. `wait for` – processes is resumed after some user specified amount of simulation time (`time_expression`); this is not some absolute time, but relative to some timing reference defined elsewhere!
3. `wait until` – process is resumed when the logical condition turns true due to a **change** of any signal listed in the condition
   →If none of the signals in that expression changes, the process will not be activated, even if the conditional expression is true

```
wait on sensitivity_list;                -- 1) on signal change
   wait on a ,b, c;                         -- example
wait for time_expression;                -- 2) explicit time
   wait for 50ns                            -- example
wait until conditional_expresion;        -- 3) until
   wait until Enable = '1';                 -- example
```

Different flavors of `wait` 2/3

Condition for `wait until` can have various forms:

```
wait until signal = value;                          -- 1.
wait until signal'event AND signal = value;         -- 2.
wait until not signal'stable AND signal = value;    -- 3.
```

1. `signal` is the name of a signal, and `value` is the value tested
2. Uses Boolean expression (here `AND`)
   - ▷ `'event` is a predefined signal attribute that is TRUE if and only if there is an event on `signal` in current Δ cycle
   - ▷ If the signal is of type bit, then if the value tested is '1', the statement will wait for the rising edge of the signal (you can imagine why this is important)
   - ▷ If the value tested is '0', the statement will wait for the falling edge of the signal
3. `'stable` is predefined signal attribute to control the amount of time in which we allow events to be considered as such

Different flavors of `wait` 3/3

- Statement `wait` has two **VERY** different forms
- First form is the one with some kind of a **condition**; this is what we have seen so far (`on`, `for`, `until`)
- Second form is to use `wait` **without any condition**!
- If there is no condition (so simple `wait;`) this is equivalent to `wait until true`
- As you could guess, such `wait` statement suspends a process forever and will never resume!
- In simulation of normal digital circuits this behavior is not right
  $\rightarrow$ Can you explain why?
- However `wait` without any condition is widely used in test-benches; these are VHDL modules written to test circuits (and not to implement them!) during simulation

Synchronous operation rapide

- The `wait until` statement is used to model, simulate & implement **synchronous** operation of digital circuits
- Usually, the signal tested is some kind of a periodic signal: typically clock (we need to be clear about why do we need this; Can you explain?)
- For example, `wait until` can be used as:

```
wait until clk = '1'; (level triggered)
wait until clk'event and clk = '1'; (edge triggered)
wait until not clk'stable and clk = '1';
```

- Note the difference between level-triggered and edge-triggered clock; in these lectures (and almost always) you will use edge-triggered clock
- For models that are to be synthesized (and not only simulated), the `wait until` statement must be the first statement in the process

Combinatorial or sequential processes

- When used to model combinational logic for synthesis, a process may contain only one `wait` statement
- If a process contains a `wait` statement, it cannot contain a sensitivity list (we have seen this)
- Thus the following two models are the same:

```vhdl
entity myBlackBox is port(
... -- IO definition
);end myBlackBox;

architecture synthesis of myBlackBox is

begin
   proc3: process (a, b, c)
      begin
         x <= a and b and c;
   end process proc3;

end synthesis;
```

```vhdl
entity myBlackBox is port(
... -- IO definition
);end myBlackBox;

architecture synthesis of myBlackBox is

begin
   proc3: process
      begin
         x <= a and b and c;
         wait on a, b, c;
   end process proc3;
end synthesis;
```

When synthesized both will yield combinatorial circuits!!!

5. Conditional statements

Definition <=> MUX in combinatorial circuits

- We have already used this, but let's look at it more in depth
- The `if` statement selects one or more statement sequences for "execution" (you should know why I put quotes), based on the evaluation of a condition corresponding to that sequence:

```
[name:] process [(sensitivity_list)]
begin
  if condition then
    statement_sequence
      [elsif condition then statement_sequence...]   -- this is optional
         [else statement_sequence]                   -- this is optional too
end if;
```

- Each condition is a Boolean expression evaluated to TRUE or FALSE (conditions may be more complex Boolean expressions)
- More than one `elsif` clauses may be present (with some kind of condition), but a single `else` clause may exist
- Note that the `if` needs to be defined within a `process` and all signals used in `condition` must be declared in the sensitivity list of the process that contains an `if` statement

How does it work? = python

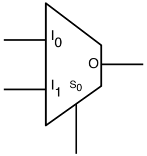
- First, the condition after the `if` keyword is evaluated, and if it evaluates TRUE, the corresponding statement are “executed”
- If it evaluates FALSE and the `elsif` clause is present, the condition after this clause is evaluated
- If `elsif` condition is TRUE, corresponding statements are executed
- Otherwise, if there are other `elsif` clauses, the evaluation of their conditions continues
- If none of evaluated conditions is TRUE, statements from `else` branch are executed, if this clause is present
  - ▷ If there is the else statement we say that `if` is complete

```
process (a, b)
  begin
    if a = b then result <= 0;
      elsif a < b then result <= -1;
        else
          result <= 1; -- else path is covered, IMPORTANT !!!
    end if;
end process;
```

Complete `if` statements

- Conditional statements provide "multiple paths" depending on the conditions that may be TRUE or FALSE and that are mutually exclusive (Boolean tests can't be both at the same time)
- In a complete `if` statement assignments for signals are defined for all possible outcomes of the Boolean condition → verif que le else assign aussi
- Example: model below will result after synthesis in a multiplexer (a mux), so nothing to do with branching in SW that could be used during simulation to model mux behavior
  - ▷ Make sure you can understand the difference

```
entity mux is port (
   a, b, s : in standard_logic;
         c : out standard_logic
);end mux;
architecture functional of mux is
  begin
    process (a, b, s)
       begin
         if s = '1' then c <= a;
           else
             c <= b;
         end if;
    end process;
end functional;
```



Incomplete `if` statement 1/2 → NOT GOOD

- Previously we have defined all possible outcomes of the conditional `if` statement, i.e. there are no uncovered branches
  - ▹ And we got a mux, a combinatorial circuit
- What happens if we do not cover all possible outcomes of an `if`?
- Syntax allows it, but circuit synthesis will infer storage element to keep the previous signal value; “storage element” = latch!!!

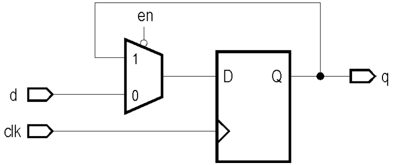
```vhdl
-- Both processes: all paths covered
process (a, b, enable)
begin
   z <= a; -- default value
   if enable = '1' then z <= b;
   end if;
end process;

process (a, b, enable)
begin
   if enable = '1' then z <= b;
      else
         z <= a;
   end if;
end process;
```

```vhdl
-- This is very much different
-- since below
-- enable = '0' is not covered

process (a, b, enable)
   begin
      if enable = '1' then
         z <= b;
      end if;
end process;
-- if enable is 1 at first and then
-- enable becomes 0, b needs to
-- be stored to keep
-- the previous value
-- Possible with memory only
```

Incomplete `if` statement 2/2: circuit level

- Imagine a process with an incomplete `if` statement
- Latch is inferred to store the value when the condition is FALSE (i.e. when `en` is not TRUE)
- Maybe designer wanted to infer a latch here ...
- Or maybe not! → plan for latches!
- If you did not planned to have a latch here, your timing may be different from what you expect!
  - ▷ This output will be delayed and whatever it will drive afterwards

```vhdl
library ieee;
architecture arch of my is
begin
  process
    begin
      wait until clk = '1';
        if en = '1' then
          q <= d;
        end if;
  end process;
end arch;
```



Attention, things may not be as you think

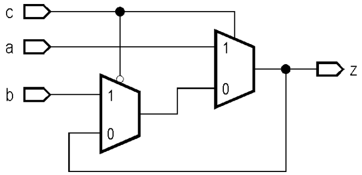
- Another example that will generate asynchronous feedback in a design
- Below, **different** signals are being assigned a value in each branch

```
wait on a, b, c;
if c = '1' then
  z <= a;
else
  y <= b;
end if;
end process;
```

- You end up with two latches, one for `z` and one for `y`

- In this example redundant test (`elseif`) will infer two muxes:

```
process
begin
 wait on a, b, c;
 if c = '1' then
  z <= a;
   elsif c = '0' then
    z <= b;
 end if;
end process;
```



Impact of inferred latches

- How does latch/FF insertion impacts my design?
- Design will use more **area**!
  - ▹ One can argue that one extra latch doesn't make any significant difference for bigger designs, but this is not always true; if repeated throughout the design we might end up using considerable amount of logic resources (and we want to use once that we really need)
- More FFs will also burn more **power**!
  - ▹ This is something we always want to minimize
- But much worse! If not understood and controlled well, extra latches will introduce supplementary latency cycles on that particular path; this will make your design less **performant**
- We will come back to this later; for now just make sure that when you use an `if` statement you always cover all possible outcomes ... unless you really want to infer a latch or a FF

6. VHDL trickiness example

Clock signal transition detection

- So far we have used the following to detect the rising edge:

```vhdl
process(clk)
   begin
      if (clk'event and clk='1') then ...
      end if;
end process;
```

- Another option is `rising_edge(clk)`, a predefined function:

```vhdl
process(clk)
   begin
      if (rising_edge(clk)) then ...
      end if;
end process;
```

- During synthesis both methods will most likely produce the same circuit (for example a flip-flop)
- But during simulation there could be a difference!
- This is a very good example of how tricky VHDL could be

Why two methods for clock transition detection?

- Difference comes from the fact that: `std_logic` or `std_logic_vector` are not just a simple binary values, contrary to `bit` that only define TRUE and FALSE
- Below table defines all possible values for `std_logic`:

`U`	uninitialized	`Z`	high impedance (tri state)
`X`	unknown[1]	`W`	weak unknown
`0`	logic 0	`L`	weak "0"
`1`	logic 1	`H`	weak "1"

[1]When multiple drivers for the same signal

- We do prefer `std_logic` to `bit` because it is closer to what happens at circuit level; `bit` is ok for more "theoretical" exploration of Boolean logic
- If `clk` is defines using `std_logic` than it could take any of the values from the table above
- In some situations this can be a problem, let's see the simulation

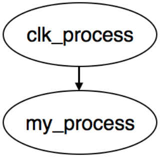
Illustrating the difference

- Let's use both in the following VHDL model:

```
my_process: process(clk)
begin
-- case 1
  if(rising_edge(clk)) then x0 <= not x0;
     - Note that we need to initialize x0 to certain value
  end if;
-- case 2
  if(clk'event and clk='1') then xr <= not xr;
     - Note that we need to initialize xr to certain value
  end if;
end process;
```

- We need another process to generate the `clk` signal:

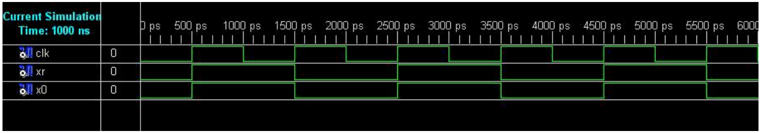
```
clk_process: process
   begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
end process;
```



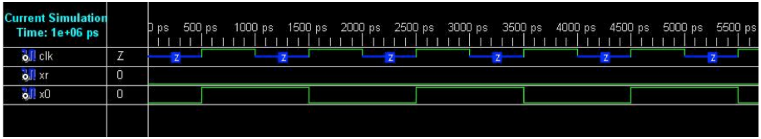
  - ▷ We use `wait for` to flip the value of the clock every half cycle!
- Note the interaction between clock & module processes through signal connectivity (`clk` is a signal shared between the two)

Clock event vs. rising edge – simulation

- Clock signal changes neatly from `0` to `1`, both descriptions (`case1` & `case2`) generate the same output:



- But if for some reason `clk` signal changes from something else, e.g. from `Z` to `1`, `case1` will not work, see waveforms below:



  `xr` value (`case1`) is wrong, it stays always at '0', since the `if` branch is never taken

- Statement `clk'event and clk = '1'` has been designed to detect ONLY valid 0 to 1 transitions!

Take away

- Wast majority of VHDL models use:

  `clk'event and clk='1'`

- But, as we saw:

  `rising_edge(clk)`

  is more safe for simulation purposes, since it covers transitions from all possible `clk` values to 1
- During these lectures (and especially labs) you can use both methods
- But you should understand the difference between the two approaches
- And yes as said VHDL modeling could be very subtile!

7. Practical examples

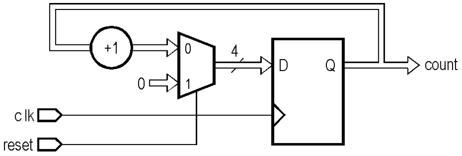
Example 1: JK Flip-flop

JK FF with synchronous reset (R), and falling edge clock:

```vhdl
library ieee;                                           --     ___________________
use ieee.std_logic_1164.all;                            -- Q | 00 | 01 | 11 | 10 | JK
                                                        --     ________________________
entity jkff is port (                                   -- 0 |  0 |  0 |  1 |  1 |
   clk, rst, j, k : in std_logic;                       --     ___________________
   q              : out std_logic                       -- 1 |  1 |  0 |  0 |  1 |
);end jkff;                                              --    ___________________

                                                    -- Do you understand the table above?


architecture synthesis1 of jkff is
begin
 process (clk)
   begin
      -- Falling edge
      if (clk'event and clk='0') then
         if (rst = '1') then q <= '0';                               -- Synch rst
            else
               if (j='0' and k='0') then q <= q;
                  elsif (j='1' and k='0') then q <= '1';             -- Use of elseif
                     elsif (j='0' and k='1') then q <= '0';
                        elsif (j='1' and k='1') then q <= not(q);
         end if;                                                     -- All paths covered
      end if;
end process;
end synthesis1;
```

Example 2: 4-bit counter

```vhdl
library ieee;
use ieee.numeric_bit.all;                             -- Why do we need this one?

entity count4 is port (
   clk, reset : in std_logic;
   count      : out std_logic_vector (3 downto 0)
);end count4;

architecture synthesis1 of count4 is
   begin
      signal z: unsigned (3 downto 0);                -- some vector of 4 bits
      process (clk, reset)
         begin
            wait until clk = '1';
               if reset = '1' then z <= "0000"; -- synch reset
                  else z <= z + "0001";               -- So this is arithmetics?
       end if;
   count <= z;
end process;
```



Example 3: Single port SRAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity single_port_RAM is port(
    clk  : in std_logic;
    we   : in std_logic;
    addr : in std_logic_vector(3 downto 0);
    din  : in std_logic_vector(7 downto 0);
    dout : out std_logic_vector(7 downto 0)
); end single_port_RAM;

architecture arch of single_port_RAM is
 -- Declare array type - why do we need this?
 type ram_type is array (2**3-1 downto 0) of std_logic_vector (7 downto 0);
 signal ram : ram_type;
 begin
  process(clk) -- Memory is synchronous
   begin
    if (clk'event and clk='1') then
      if (we='1') then
         -- convert addr std_logic_vector to integer ; why we need this?
         ram(to_integer(unsigned(addr))) <= din;   -- write data to address 'addr'
      end if;
    end if;
   end process;

  -- read data from address 'addr'
  -- convert 'addr' type to integer from std_logic_vector
  dout<=ram(to_integer(unsigned(addr)));
end arch;
```