ELEC-H-409
Th03: Signals, variables, simulation and advanced statements

Dragomir Milojevic
Université libre de Bruxelles

Today

1. Signals & variables
2. Components
3. Test-benches
4. More data types
5. Loops
6. Generating complex repetitive structures
7. Practical examples

1. Signals & variables

Signals and their limitations

- Signal values assignments are **queued** [1] and **resolved** during the simulation of the process "execution"; this is why it is allowed to do multiple assignments to the same signal, but we know what this mean at circuit & simulation levels
- New values are not assigned to the signals when the assignment statement "executes", i.e. when it is placed in the VHDL, but only **after the process execution suspends**
- The "execution" above refers to **simulation** and not synthesized circuit
- So, signals hold the **last value** that was assigned & hence cannot store intermediate results (assignments within a process)
- Here the value of `A` is updated only later, not at line 1; computation of `C` will assume old, non-updated value of `A` whatever it was and not `D`

```
1 A <= D;
2 C <= A or B;
```

[1] This is why we said that during simulation a signal is represented as a vector

What signals can't do ...

- This means that event driven simulation will assume the following:

```
1 A+ <= D-;
2 C+ <= A- or B-;
```

  where $^+$ indicate the future values, and $^-$ the past values
  - ▷ Left side is the future and the right side is the past
  - ▷ All the future values of signals are updated simultaneously
  - ▷ This ensures the consistency of the inputs: imagine that between lines 1. and 2. the input `D` changes, we would refer with same symbol to two different values!
- So, the question then is how to model the following:

```
1 A = D;
2 C = A or B;
```

  where in line `2` we want `C` to take the new value of `A`, so `D`, as assigned in line `1` just like with computers; in another words we want the assignment to immediately take place as the statement `1` is encountered

Signals & Variables

- Such explicit **immediate** assignments can be modeled in VHDL using variables as opposed to signals
- Variables **can be used to store intermediate results**
- Previous example of computation can be then realised **as in a sequential computer**, where we are used to the following:

```
1 A = D;
2 C = A or B;   -- A is D because A=D has been executed by
3               -- the CPU BEFORE this expression
4               -- has been executed
```

- Variables can be used only in the sequential domain of the VHDL language, so:
  - inside processes and subprograms
  - they cannot be declared or used directly in an architecture
  - outside the process declared variable can not be seen
- Variables are local to that process!

Signals & Variables in VHDL

So, there is a HUGE difference between :

Signals – they change their value **after** the new value has been assigned to a given identifier; this method enables simultaneous assignments of multiples values at the same time, i.e. multiple concurrent circuits that compute things in parallel

Notation: **<=**

Variables – they change their value "**instantaneously**", like we are used to do in standard programming languages and computers; when the assignment takes place, the value is immediately taken (this is of course during simulation)

Notation: **:=**

The difference between the two is far from being subtile!

Computations using **signals**

```
1  architecture sign of example is
2  signal r : integer :=0;
3  signal s1: integer :=1;
4  signal s2: integer :=2;
5  signal s3: integer :=3;
6  begin
7    process
8      begin
9        s1 <= s2;
10       s2 <= s1 + s3;
11       s3 <= s2;
12       r  <= s1 + s2 + s3;
13       end process;
14 end sign;
```

```
1  architecture sign of example is
2  signal r : integer :=0;
3  signal s1: integer :=1;
4  signal s2: integer :=2;
5  signal s3: integer :=3;
6  begin
7     process
8        begin
9              s1+ <= s2-;
10             s2+ <= s1- + s3-;
11             s3+ <= s2-;
12             r+  <= s1- + s2- + s3-;
13       end process;
14 end sign;
```

Computation is **simultaneous**, all signals are updated at the same time after some Δ-delay; we compute the "future" values using arguments from the "past", so:

$$\begin{aligned} s1 &= s2 = 2 \\ s2 &= s1 + s3 = 1 + 3 = 4 \end{aligned}$$

$$\begin{aligned} s3 &= s2 = 2 \\ r &= s1 + s2 + s3 = 1 + 2 + 3 = \mathbf{6} \end{aligned}$$

Result: 6 – let's keep this in mind

Computations using **variables**

```
1  architecture var of example is
2    signal r: integer := 0;
3    begin
4      process
5        variable v1: integer :=1;
6        variable v2: integer :=2;
7        variable v3: integer :=3;
8          begin
9            v1 := v2;
10           v2 := v1 + v3;
11           v3 := v2;
12           r <= v1 + v2 + v3;
13     end process;
14 end var;
```

```
1  architecture var of example is
2    signal r: integer := 0;
3    begin
4      process
5        variable v1: integer :=1;
6        variable v2: integer :=2;
7        variable v3: integer :=3;
8          begin
9            v1+ := v2-
10           v2+ := v1+ + v3-
11           v3+ := v2+;
12           r <= v1+ + v2+ + v3+;
13     end process;
14 end var;
```

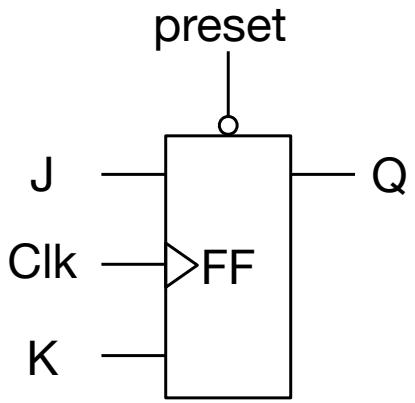
We assign the same initial values as in signals; but computation is **sequential**, update is immediate, the final result of computation is completely different (as well as intermediate values)!

$$v1 = v2 = 2 \qquad v3 = v2 = 5$$
$$v2 = v1 + v3 = 2 + 3 = 5 \qquad r = v1 + v2 + v3 = 2 + 5 + 5 = 12$$

Result: 12 (6 in case of signals !!!)

Initial values: synthesis vs. simulation

- Initial values assigned to signals & variables in the models on the previous slide are used for simulation only!
- During synthesis **any initial assignments will be ignored**
  - This is obvious, can you explain?
- If we need to assign a specific value of a signal during initialization of the actual hardware, we have to do it differently
- We need memory elements with preset, so that some value can be initialized after global system reset



- Preset value needs to be explicitly specified in the VHDL model

VHDL model of JK with preset

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity JK_flip_flop is port (
4     clk,J,K,prs,clr : in std_logic;
5     Q               : out std_logic;
6     Qnot            : out std_logic);
7   end JK_flip_flop;
8  architecture JKFF_arch of JK_flip_flop is
9  signal nxt_state,prv_state: std_logic;
10 signal input: std_logic_vector(1 downto 0);
11 begin
12    input <= J & K;                                    -- Concatenate two bits
13    process(clk, prs, clr) is
14       begin
15          if (clr='1') then                           -- This is simple clear
16             nxt_state <= '0';
17             elsif (prs='1') then                     -- This is where preset
18                nxt_state <= '1';                     -- sets the value of FF
19                elsif (clk'event and clk='1') then
20                   case (input) is
21                    when "10" => nxt_state <= '1';
22                    when "01" => nxt_state <= '0';
23                    when "00" => nxt_state <= prv_state;
24                    when "11" => nxt_state <= not prv_state;
25                    when others => null;
26                   end case;
27          end if;
28    end process;
29 Q <= nxt_state;
30 Qnot <= not nxt_state;
31 prv_state <= nxt_state;
32 end JKFF_arch;
```

Another example to discuss – Full-adder

- Computes the sum of three bits, two from operands & 1 cary bit

```
1  entity FullAdder is port(
2     a    : IN std_logic;
3     b    : IN std_logic;
4     cin  : IN std_logic;
5     d    : IN std_logic;
6     s    : OUT std_logic ;
7     cout : OUT std_logic
8  );end FullAdder;
9
10 architecture arch of add_1 is begin
11   process (a, b, cin)
12     variable s1, s2, c1, c2: bit;
13       begin
14         s1 := a xor b;
15         c1 := a and b;
16         s2 := s1 xor cin;      -- because s1, s2 is variable this will work !
17         c2 := s1 and cin;
18         s <= s2;               -- discuss this
19         cout <= c1 or c2;
20 end process;
21 end functional;
```

- This is the right model of the full-adder circuit behaviour
- Propose an alternative implementation (using signals) and discuss the circuit difference and trade-offs

Signals vs. variables 1/2

- Declaration
  - Signals cannot be declared inside a process or a subprogram
  - Variables must be declared inside a process or a subprogram
- Update process
  - Signals update the signal driver, new value of the signal is updated when the process is suspended
  - Variables are updated immediately when the variable assignment statement is executed
- Communication
  - Signals communicate among concurrent statements
  - Ports declared in the entity are signals; as said, signals are like wires
  - Variables are used as temporary values in a function description; useful to factor out common parts of complex equations to reduce mathematical calculations during simulation

Signals vs. variables 2/2

- Right-hand side
  - Signals on the right-hand side are not a single value, rather a sequence of waveform elements with associated time expressions
  - Variables assignment statement is an expression, there is no associated time, only one value is stored
  - Look at the following:

```
1 a <= b; -- even if we don't put time,
2 a <= a; -- this will not work
3 -- but with variables something like:
4 a := f(a); -- where f is some function, will work!
5            -- a on the right side is an old value
6            -- a on the left side is the future value
```

- Variables are **cheaper to implement** in VHDL simulation since the evaluation of drivers is not needed (single memory location); but the difference will be seen only on big designs
- Variables also require **less memory** (during simulation); once again with current DRAM capacities you will not see the impact with small designs

Complex assignments: `case` statement

- Selects for "execution" one from several **alternative** statement sequences based on the value of an **expression**:

```
1  case expression is
2    when val =>                              -- this is when expression=val
3      statement_sequence
4    when val1 | val2 | ... | valn =>
5      statement_sequence
6    when val3 to val4 =>
7      statement_sequence
8    when val5 to val6 | val7 to val8 =>
9    statement_sequence
10   ...
11   when others =>                           -- default value
12 statement_sequence                         -- don't forget that there are
13 end case;                                  -- other things than 0 and 1
```

- Unlike `if`, the expression doesn't need to be Boolean: we can use signals, variables or expressions of any discrete type (an enumeration or an integer type) or a character array type (including `bit_vector` and `std_logic_vector`)
- Used when there are a large number of possible assignments

Example of `case` statement

- XOR gate with 2 input variables (looks like a truth table)
- Note the construction of the input vector from two different bits
- Bellow `&` is the concatenation operator
  - ▷ What does it do?

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity xor2 is port (
5    a, b : in std_logic;
6    x    : out std_logic
7  );end xor2;
8
9  architecture arch_xor2 of xor2 is begin
10   process (a, b)
11     variable temp: std_logic_vector (1 downto 0);
12       begin
13         temp := a & b;                -- Concatenation of 2 bits
14         case temp is
15            when "00" => x <= '0';
16            when "01" => x <= '1';
17            when "10" => x <= '1';
18            when "11" => x <= '0';
19            when others => x <= '0';   -- This one is there for sake of completeness
20         end case;                     -- Could it be possible that ab has something
21   end process;                        -- else as a value ?
22 end arch_xor2;
```

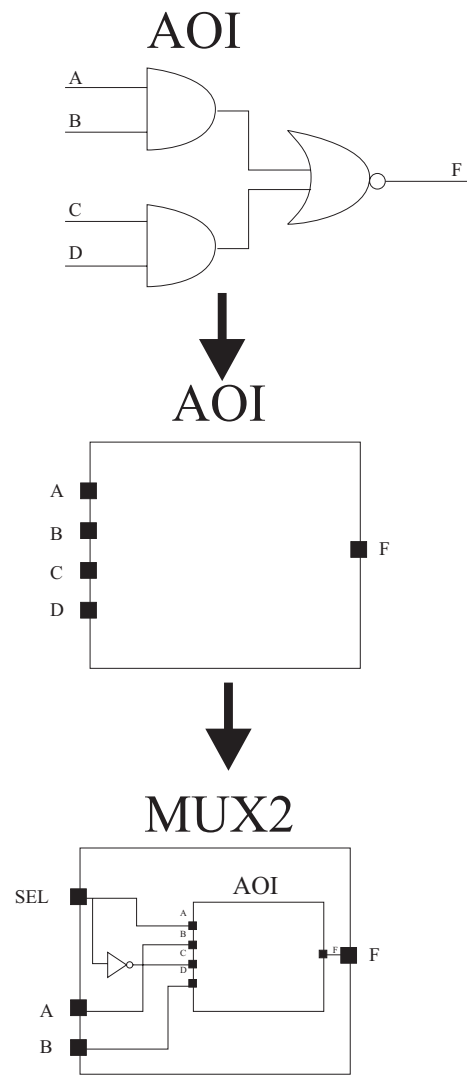
2. Components

Motivation

- Good practice is to write one module per VHDL file (how to "cut" a big system into modules is an art of its own)
- Question then is how to practically enable more complex designs in VHDL?
- Use good old "divide and conquer" through components:
  - Simplifies description of a circuit with multiple instances of the same module (component re-use within the same design)
  - Allows component re-use across different designs, enables portability
  - Enable circuit structuring, i.e. design hierarchy
  - Improve human design readability
  - Improve run-time (you need to re-synthesize only what is changed)
- Analogy with SW: pre-processor macros where the given text code is replaced at compile-time
- Module instance is **created at design (synthesis) time**, not at run-time, like the functions in compiled SW programs

Components – how it works?

- We first define AOI (And-Or-Invert) component
- AOI becomes a "black box" with embedded functionality & interface that defines the way it should be connected with other components
- Interface defines ports to the outside world (logical connections at block boundary)
- AOI is instantiated inside another module (one, or more instances can be now created)
- Appropriate connections are made to connect input/output ports of the component with internal signals or inputs/outputs
- We speak about top-level (`MUX2`) & bottom-level modules (`AOI`), this define system hierarchy



Example of usage

- Circuit `MUX2` is composed of INV and AOI – a module on its own (1.)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity aoi is port (                                    -- 1. functionality
4    a,b,c,d : in std_logic;
5    f       : out std_logic
6  );end aoi;
7  architecture v1 of aoi is
8  begin
9   f<=not((a and b)or(c and d));
10 end v1;                                                 -- assume the same for an INV
```

- In the new module components are first declared (2.)
- New instance of the component and the connexions (3.)

```
1  entity mux2 is port (                                   -- you need to include libs
2    sel, a, b     : in std_logic;
3    f             : out std_logic
4  ); end mux2;
5  architecture structure of mux2 is
6  component inv                                           -- 2. components definitions
7    port (a: in std_logic; f: out std_logic);
8  end component;
9  component aoi
10   port (a, b, c, d: in std_logic; f : out std_logic);
11 end component;
12 signal selb: std_logic;
13 begin
14    g1: inv port map (sel, selb);                        -- 3. instances and wiring
15    g2: aoi port map (sel, a, selb, b, f);
16 end;
```

Signal association schemes

Positional – internal signal have to follow the order of the signals defined by the entity definition and the component declaration

```
1  architecture struct of inc is
2
3  signal x,y,s,c : bit;
4
5  -- we need to declare component
6  component halfadd
7    port(a,b : in bit;
8         sum, carry : out bit);
9  end component;
10
11 begin
12 u1: halfadd port map (x,y,s,c);
13
14 -- other statements
15 end struct;
```

Named association – no need to respect order, the connections of local signals to ports are explicitly specified

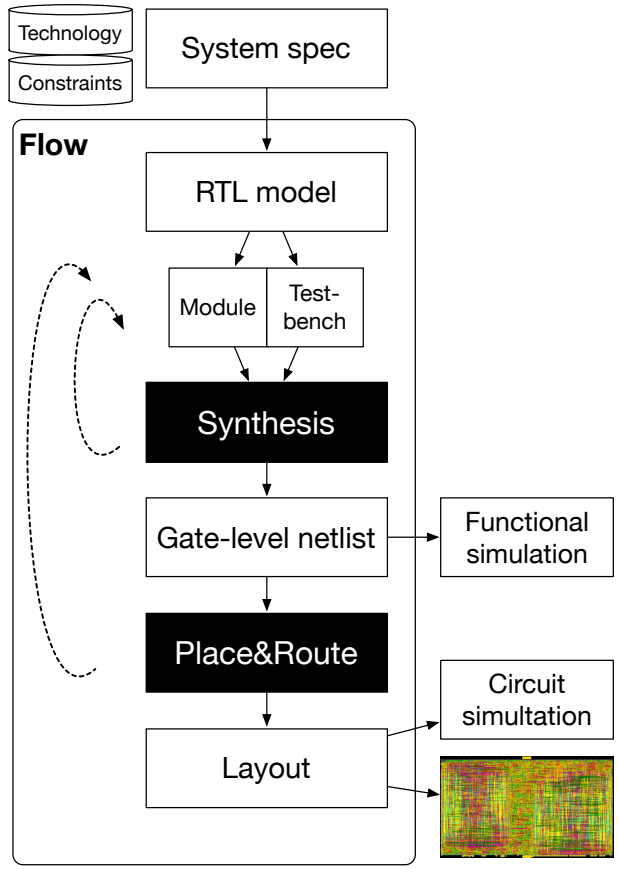
```
1  architecture struct of inc is
2  signal x,y,s,c : bit;
3
4  component halfadd
5    port(a,b : in bit;
6         sum, carry : out bit);
7  end component;
8
9  begin
10   u1: halfadd port map
11     (b     => y,
12      a     => x,
13      sum   => s,
14      carry => c);
15 end struct;
```

NO difference of the approach chosen on the final circuit! However most of designers adopt named associations (more clear and using right editor easier to write despite more characters to type)

3. Test-benches

Circuit testing

- Design flow is about circuit modeling and **model transformation** using computer programs – EDA tools
- After each step in the flow we need to make sure that the input we provide is functional, and that the transformed model we obtain is correct
- Functional simulation – checks Boolean functionality and makes rough estimates on area, timing & power
- Circuit simulation – models post placement & route are used for final check, called sign-off: circuit can be implemented and/or manufactured



Design flow and two levels of simulation

Functional simulation

- Insures that the RTL is functionally correct: synthesis engine "understood" your VHDL model and have produced the right circuit
  - ▷ Does this mean that the circuit will operate if implemented as such?
- If the circuit is not functionally correct, there is no point in going further in the flow that would typically take more time than synthesis
- In the above, test means analyze the **output(s)** for a given set of known **inputs** and check their validity
  - ▷ Is it possible to make exhaustive (i.e. complete) tests and guarantee operational correctness in all cases ?
  - ▷ Test will include creation of module instance, definition of input vectors & running the simulation for a certain amount of time
  - ▷ This is done using test benches – a VHDL module with instance of a module to test and inputs assignments
- After synthesis we can get high-level estimates on actual performance, power and area; these estimates are in general optimistic, and will degrade after placement & route

Circuit simulation

- Actual circuit properties (power, performance, area) are derived only after full implementation (place & route)
- This is because during these operations, tools will try to perform design optimizations to reach imposed constraints about the performance, area and power
- To do so the tools will transform the netlist, so the logic circuit, but will obviously keep the functionality unchanged
- Since there is no optimal solution (one that minimizes all the parameters), EDA tools are typically configured to aim for the following objectives:
  - Maximize performance – this usually compromises area & power, or
  - Minimize power – typically achieve at the expense of performance
- The above objectives are mutually exclusive, the reason why you have today two very distinct classes of digital systems: high-performance or low-power

Circuit simulation

After full physical implementation, accurate IC properties are derived and will take into account Process, power supply (Voltage) and Temperature variations (PVT)

- Timing – to extract minimum clock period (or maximum operating frequency) for given operating conditions range (e.g. temperature)
- Power – for a given logic activity, often derived from actual logic simulation to extract activity of each gate and wire
- Thermal – derived from power simulation and cooling solution adopted (simple convection, passive cooling, forced cooling etc.)
  - ▷ Heat has to be managed to avoid early tear down or device failure
- Mechanical
  - ▷ Yes, ICs "move" due to different coefficients of thermal expansion; as IC heats up & cool down, the silicon expands & shrinks and could yield mechanical failures of components
- Electromigration – high current densities can cause displacement of the conductor material at atomic scale, resulting in wire breaks & IC failure

Functional simulation & test-benches

- Test-benches (TBs) are VHDL modules that don't have IOs
  - Can you explain why?
- TBs instantiate the Unit Under the Test (acronym `UUT`) and some internal signals, typically one per input of `UUT` and they assign values to these internal signals (i.e. inputs):
  - Assignments, expressions, periodic signals (e.g. clock), but also data from external files etc.
- A good practice is to create one (or more) test-bench(es) for each module in the design and name it accordingly for easy identification
  - Typically designers use something like: `tb_top_level_module`, where `top_level_module` is the name of the module to test (and files are named accordingly, e.g. `tb_top_level_module.vhdl`)
- Once the circuit and test-bench are simulated, designer gathers outputs in different forms: waveforms (this is what you will do most of the time), memory outputs, written data files etc.
- These are then checked manually or automatically for validity

Example: two modules to test

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity myor is port(
 5    x: in std_logic;
 6    y: in std_logic;
 7    f: out std_logic
 8  );
 9  end myor;
10
11  architecture myarch of myor is
12  begin
13    f <= x or y;
14  end myarch;
15
16  library ieee;                    -- The above models are written just to have
17  use ieee.std_logic_1164.all;     -- something, there is no need to create modules
18                                   -- for individual gates synthesis tools know them
19  entity myand is port(
20    x: in std_logic;
21    y: in std_logic;
22    f: out std_logic
23  );
24  end myand;
25
26  architecture myarch of myand is
27  begin
28    f <= x and y;
29  end myarch;
```

Example: associated test-bench

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity tb_and_or_ent is                          -- test-bench module definition
5  end tb_and_or_ent;                               -- No IO
6
7  architecture tb of tb_and_or_ent is
8  component myand port (                           -- component definition
9    x : in std_logic;
10   y : in std_logic;
11   f : out std_logic);
12 end component;                                   -- Same goes for myor
13
14 signal x  : std_logic := '0';
15 signal y  : std_logic := '0';
16 signal f1 : std_logic;
17 signal f2 : std_logic;
18
19 constant tbperiod : time := 100 ns;            -- clk with some time !!!
20 signal tbclock : std_logic := '0';              -- above time is relative to something
21
22 begin
23   dut_and : myand port map (x=>x,y=>y,f=>f1);   -- component instantiation
24   dut_or : myor port map (x=>x,y=>y,f=>f2);     -- and mapping of IO
25
26   tbclock <= not tbclock after tbperiod/2;
27
28   stimuli : process
29     variable cnt : integer := 0;
30       begin
31         x <= '0'; y <= '0';
32         wait for tbperiod;                       -- This input will last 1 cycle
33
34         x <= '1'; y <= '0';
35         wait for tbperiod;                       -- and one more
36   end process;
37 end tb;
```

Discussion on test-benches

Using the previous example, discuss the following:

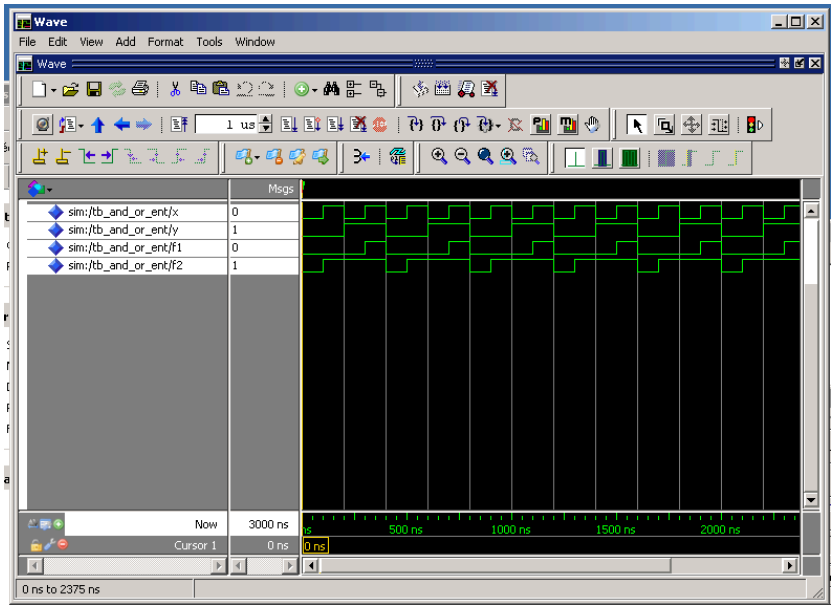
- Is it normal that the test-bench don't have interface definition?
- How long the simulation will last?
- How long the process `stimuli` will last?
- Why `cnt` has been defined as a variable and what it can be used for in this circuit?
- Does 100ns Clk period has something to do with the model written?
- Have we explored all possibilities for the inputs?
- How we could do that?
- Will you implement the TB as a circuit?
- How you would implement similar simulation in HW?

Simulation process

- RTL is first elaborated – syntax errors checks, design hierarchy is established starting from the top module, an unique copy of each sub-module instance is created (in hierarchical manner)
- Order of instance creation is important! (instance module needs to be in the library before it can be used); some synthesis tools can figure out the order automatically (e.g. ModelSim)
- During elaboration (technology) unmapped gate-level netlist is produced (generic gates with logic functionality only)
- This is commonly stored in a subdirectory to root directory of the project and called something like **`WORK`**, or similar; commonly also referred to as design library
- When you simulate something it has to be in this library **`WORK`**
- In the previous example both modules and TB are going to be found in **`WORK`**
- If you simulate the TB you should be able to generate (logic) **wave(forms)s**

Illustration – simulated waveforms

- After simulation, output is analyzed (manually or automatically) against something we are sure off – golden specification; this could be anything: hand written truth tables, computer generated files etc.



- Process of VHDL modeling, synthesis and simulation is repeated many times! Any change in VHDL means re-synthesis & re-simulation & re-validation of the results
- Simulation tools provide automation (scripts) to speed-up things, which is very useful and you should use this!

Example of simulator scripting

- Example of simulation automation script for ModelSim and QuestaSim using Tcl language (a reference scripting language in EDA industry, most tools are using it): similar to Makefiles

```
1  # ee201_gcd_tb_Part1.do
2  vlib work  -- vlib : specifies a project work library
3  vlog +acc "ee201_gcd.v" -- invokes verilog synthesis
4  vlog +acc "ee201_gcd_tb_Part1.v"
5
6  -- this will launch simulation of top instance
7  vsim -t 1ps -lib work ee201_gcd_tb_Part1 view objects
8
9  -- specifies a tool window for view
10 view wave
11
12 run 300ns -- runs for some time
```

- Above script could be sourced from the simulator shell (interactive interface with the tool)
- You do not necessarily need to learn Tcl, but some basic knowledge could help (all of the above commands are from the simulation tool)

4. More data types

VHDL data types

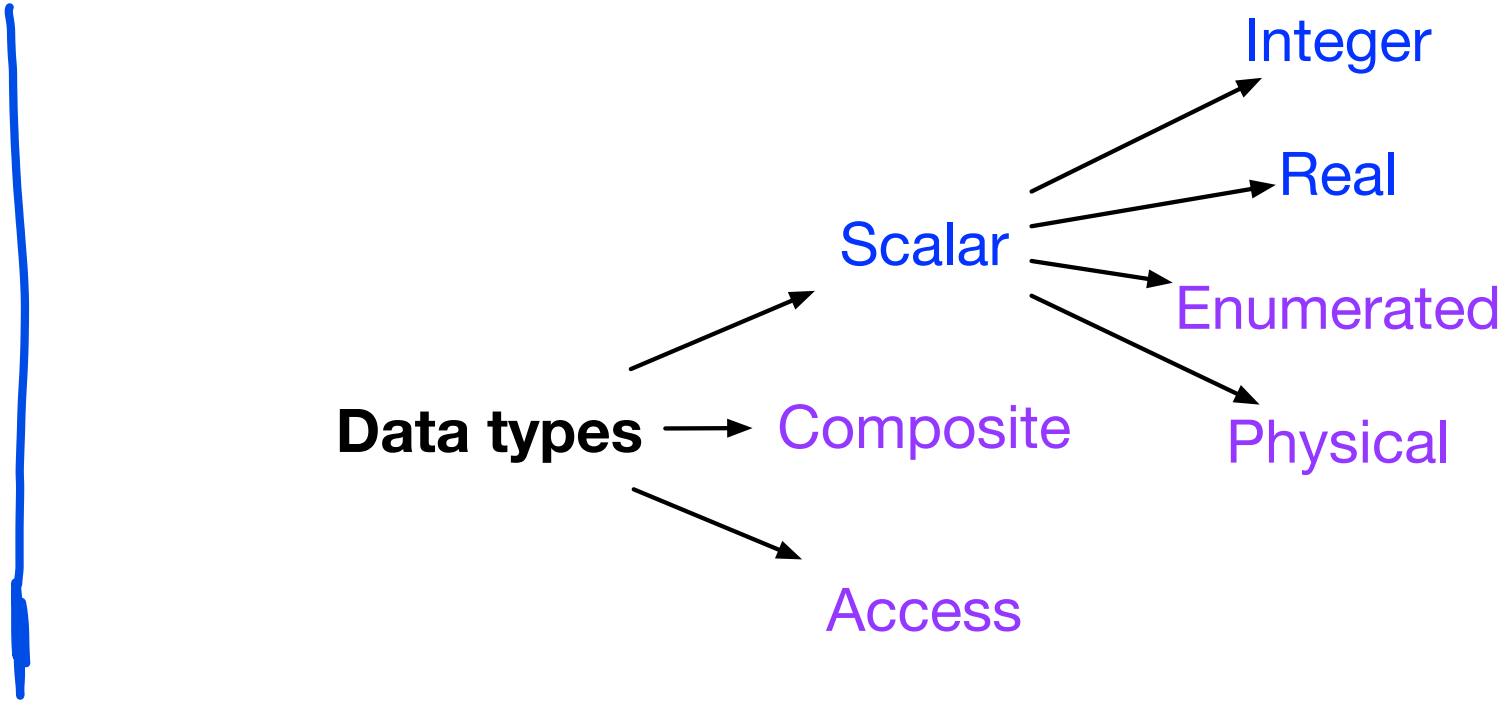
- We know that VHDL ports, signals and variables must specify their type → VHDL is a **strongly** typed language, no room for abstraction
- Data types could be further refined using subtypes
- Using a keyword `subtype` design can define the exact range for an identifier:

```
1 -- General form
2 subtype subtype_name is base_type range range_constraint;
3
4 -- Example that you know
5 subtype byte is bit_vector (7 downto 0);
```

- If you declare a sub-type the synthesis tool will infer the appropriate number of bits required to encode the object!
- This is fundamental for HW designs: we need to transform everything into wires that process binary values

All data types in one place

- All VHDL data types are summarized as follows:



- We have already discussed integer (& real) scalars, on top of which there are enumerated & physical types
- Most of the time we have been using `std_logic` which is enumerated type and `std_logic_vector` is an array of enumerated type

Enumerated data types

- Typical programming languages define enumerated data types – a data type consisting of a set of named values called **elements**, members or enumerators of the type; Identifier of that data type can take one out of enumerated elements only
- A user defined type in VHDL is always an "enumerated type"
- Most synthesis tools are able to synthesize a circuit containing enumerated types (this is particularly useful for state machines as we will see), but check before
- Syntax for defining an enumerated type:

```
1 -- general format
2 type my_type is (my_coma_separated_list);
3
4 -- From std_logic_package_1164
5 type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
6
7 -- Note that the range is not defined
8 type std_logic_vector is array ( natural range <> ) of std_logic;
```

- Since digital systems process binary data only, what does the synthesis process do with this enumerated list?

Example of use: data types & case statement

- Traffic light model: a fixed sequence of light transitions; this is clearly a state machine, I hope you see this right away
- Based on the **present state** (`color`) the system moves into the next state (`next_color`); note that there are no inputs here (this is OK)
- We first define our own enumerated data types (line 1) and then use `case` to assign future state depending on the current state:

```
1  type type_color is (red, yellow, green);      -- only these options are possible
2
3  process (color)
4    case color is
5      when red =>
6        next_color <= green;                    -- no other option
7      when yellow =>
8        next_color <= red;                      -- no other option
9      when green =>
10       next_color <= yellow;                   -- no other option
11   end case;
12 end process;
```

  - ▷ Why the model above would result in a sequential circuit?
  - ▷ Something is missing here, what?
  - ▷ How many bits are required to encode this state machine?

Physical data types

- Since VHDL is used to model and implement electrical circuits, it allows the use of objects that have a value and the unit of some physical property as well
- We have already used quantities such as time, but designer could define other ones, such as resistance and capacitance (useful when dealing with electrical properties of ICs)
- Physical type defines a basic unit for the quantity and may define other units, multiples of this unit

```
1  type time is range 0 to ...
2     units
3        fs;
4        ps = 1000 fs;
5        ns = 1000 ps;
6        us = 1000 ns;
7        ms = 1000 us;
8        sec = 1000 ms;
9        min = 60 sec;
10       hr = 60 min;
11    end units time;
```

Composite & access data types

- Composite data types are collections of scalar types, there are: arrays and records
- We have used arrays, defined as indexed collections of scalar types, where the index must be a discrete scalar type; arrays can be one- or multi- dimensional (e.g. `std_logic_vector`)
- Records are like structures in C, they allow to group different objects into a new user defined and single object
- Access types are pointers, or handles to objects such as memory very much like in C, files etc.; they allow to dynamically (during simulation) use host computer ressources
- Simple example (FYI only)

```
1 type node;
2 type pointer is access node;
3
4 type node is record
5   data : integer;
6   link : pointer;
7 end record;
```

Access to files

- Could be used to interface in SW way the simulation of HW: you could read in stimuli, or write out the simulation results
- Files have data types, identifiers (file handles), they must be opened (and closed) before the use (read and/or write)
- Example of file creation, opening and write:

```
1  architecture arch of io is begin
2  process is
3     type IntegerFileType is file of integer; -- file type
4     file data_out: IntegerFileType; -- handle for this file
5
6     variable fstatus: FILE_OPEN_STATUS;
7     variable count: integer:= 0;
8     begin
9        -- opens the file for write, you could check the status
10       file_open(fstatus, data_out, "myfile.txt", write_mode);
11
12       -- Do something with the file
13       for i in 1 to 8 loop
14          write(data_out, count);
15          count := count + 2;
16       end loop;
17       -- do not leave the file locked by OS
18       file_close(data_out);
19       wait;
20 end process;
21 end architecture arch;
```

5. Loops

Loops

- Enable repeated "execution" of the statements found in the loop body that must be declared within a `process` statement
- In HW loops mean replication of logic structures, so this is very, VERY different from SW and a source of misunderstandings
  - In SW loops mean execution of instructions sequenced in time & driven by CPU clock (1 instruction at a time)!
- Different types of loop statements:
  - `loop` – indefinite repetition of some statements
  - `while` – placed before the `loop` keyword: statements are repeated until a given **condition** (typically Boolean expression) becomes **false**
  - `for` – placed before the `loop` keyword: statements of the loop body are repeated for a **fixed** number of times (loop index, counter, ...)
- Loop `for` is the only one that will be used for actual logic synthesis, other two are used for simulation only

Controlling the loops & syntax

```
1  -- 1. simple loop
2  -- this will loop forever
3  [label:] loop
4    statement_sequence
5  end loop [label];
6  -- Is it ok to loop forever?
7
8  -- 2. while with condition
9  [label:] while condition loop
10   -- condition above is some Boolean expression
11   statement_sequence
12 end loop [label];
13
14 -- 3. for
15 -- this one is the only one that can be synthesized
16 -- synthesized instance will be named according
17 -- to the loop_label --> useful for debugging the simulation
18 loop_label: for loop_parameter in range loop
19   sequence_of_statements
20 end loop loop_label;
```

Loops usage: examples

```
1  -- 1. loop for simulation
2  process(clk)
3    variable count: integer := 0;
4      begin
5        wait until clk = '1';
6          while level = '1' loop
7            count := count + 1;
8            wait until clk = '0';
9          end loop;              -- when this will end? explain ...
10 end process;
11
12 -- 2. nested loop: ok but we need to name them
13 E1: while i < 10 loop
14   E2: while j < 20 loop
15   -- statements
16   end loop E2;
17 end loop E1;
18
19 -- 3. We don't need to declare i previously (integer always)
20 for i in 1 to 10 loop
21   i_square (i) <= i * i;
22 end loop;
```

Loops usage: examples (contd.)

```
1  -- 4. Skip one iteration of the loop
2  for i in 0 to 7 loop
3     if skip = '1' then next;          -- this will skip current iteration
4        else N_BUS <= TABLE(I);
5     wait for 5 ns;
6     end if;
7  end loop;
8
9  -- 5. Loop with an exit
10 process (a)
11    variable i : integer range 0 to 4;
12    begin
13       z <= "0000";
14       i := 0;
15       loop
16          exit when i = 4;           -- this will exit the loop
17          if (a = i) then
18             z(i) <= '1';
19       end if;
20       i := i + 1;
21    end loop;
22 end process;
```

More on loop index & common misunderstandings

- In programming languages, the loop index is a **variable** like any other, that is a memory location, i.e. you can use this variable for any further computation a value may be **assigned** to the iteration count (`i` in the previous slide)
- In VHDL you can not use the loop index like in C!
- **You can not assign** a value to the iteration count or use it as an input or output parameter of a procedure
- In VHDL loop counter may be used in an expression, provided that its **value is not modified** (since loop index is not solved dynamically, which is the case in programming languages!)
- There is no need to declare index explicitly within the process (e.g. it is automatically defined after using `for`)
- If there is another variable with the same name within the process, these two will be treated as separate variables (don't do this unless you want to confuse other that will read your model)

confused by the big d** of kicks

Practical use of loops

- Whenever we need concurrent assignment using the **same** logical function on a set of bits, rather then a single bit
- Two descriptions below are equivalent: 2nd model is better than 1st, since it can be parametrized, it is much easier to write, more readable etc.

```
1  entity match_bits is port (
2    a, b: in bit_vector (7 downto 0);
3    m: out bit_vector (7 downto 0)
4  ); end match_bits;
5
6  architecture functional of match_bits is
7  begin
8     process (a, b)
9        begin
10          m(7)<= not(a(7) xor b(7));
11          m(6)<= not(a(6) xor b(6));
12          m(5)<= not(a(5) xor b(5));
13          m(4)<= not(a(4) xor b(4));
14          m(3)<= not(a(3) xor b(3));
15          m(2)<= not(a(2) xor b(2));
16          m(1)<= not(a(1) xor b(1));
17          m(0)<= not(a(0) xor b(0));
18          -- this could be long ...
19 end process;
```

```
1  entity match_bits is port (
2    a, b: in bit_vector (7 downto 0);
3    matches: out bit_vector (7 downto 0)
4  ); end match_bits;
5
6  architecture functional of match_bits is
7  begin
8    process (a, b)
9      begin
10       for i in 7 downto 0 loop
11         matches(i)<=not(a(i) xor b(i));
12       end loop;
13   end process;
14 -- much better description
15 end functional;
```

- Also, loops are cool to perform a sequence of inter-dependent computations (using variables for example)

Example of inter-dependent computations: parity

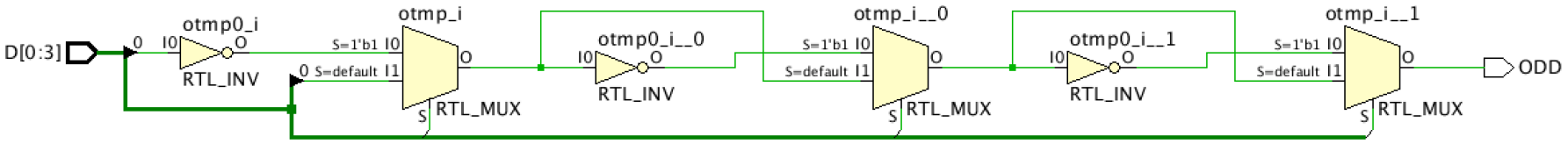
- Count the number of '1' in a vector (of a fixed size)
- If the number of '1' is odd, set the output `ODD` to 1, otherwise set it to '0'
- Idea of the circuit model:
  - Assume all bits in vector are 0
  - Sweep all the bits of the vector from LSB to MSB (this is why we use the loop)
  - For each 1 that has been detected during the sweep, invert the parity bit

Attention!!! How do you interpret the VHDL model on the right

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity parity10 is port(
5   D    : in std_logic_vector(0 to 9);
6   ODD  : out std_logic
7  );
8  end parity10;
9
10 architecture behavior of parity10 is
11 begin
12 process(D)
13 variable otmp: Boolean;
14 begin
15    otmp := false;
16    for i in 0 to 9 loop
17       if D(i) = '1' then
18          -- we need variables
19          -- to update immediately
20          -- the value of otmp
21          otmp := not otmp;
22       end if;
23    end loop;
24    if otmp then
25       ODD <= '1';
26          else
27             ODD <= '0';
28    end if;
29 end process;
30 end behavior;
```

Loop repeat things in space, not in time!!!

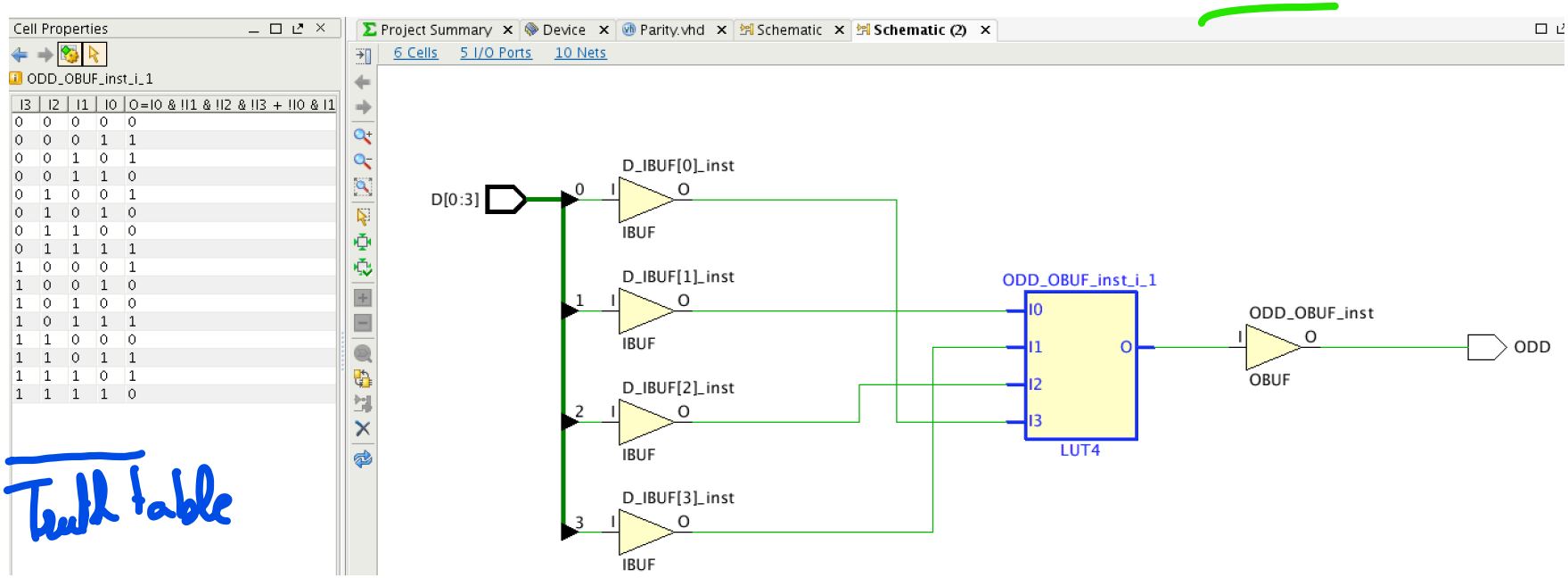
- When synthesized to "RTL schematic" (an option available in Xilinx tools) this is what you will get after elaboration:



- Explain the schematic above and make a link with the VHDL module on the previous slide?
- However the model below has issues:
  - Loop creates a serial connection of identical circuits in space, not in time (like usual computer loops)
  - If the loop index is high, the logical depth of the circuit will be high too, resulting in huge combinatorial delays
  - Such circuit could be a critical path in the design, limiting performance of the system
    - Could you suggest and improvement of the model above?

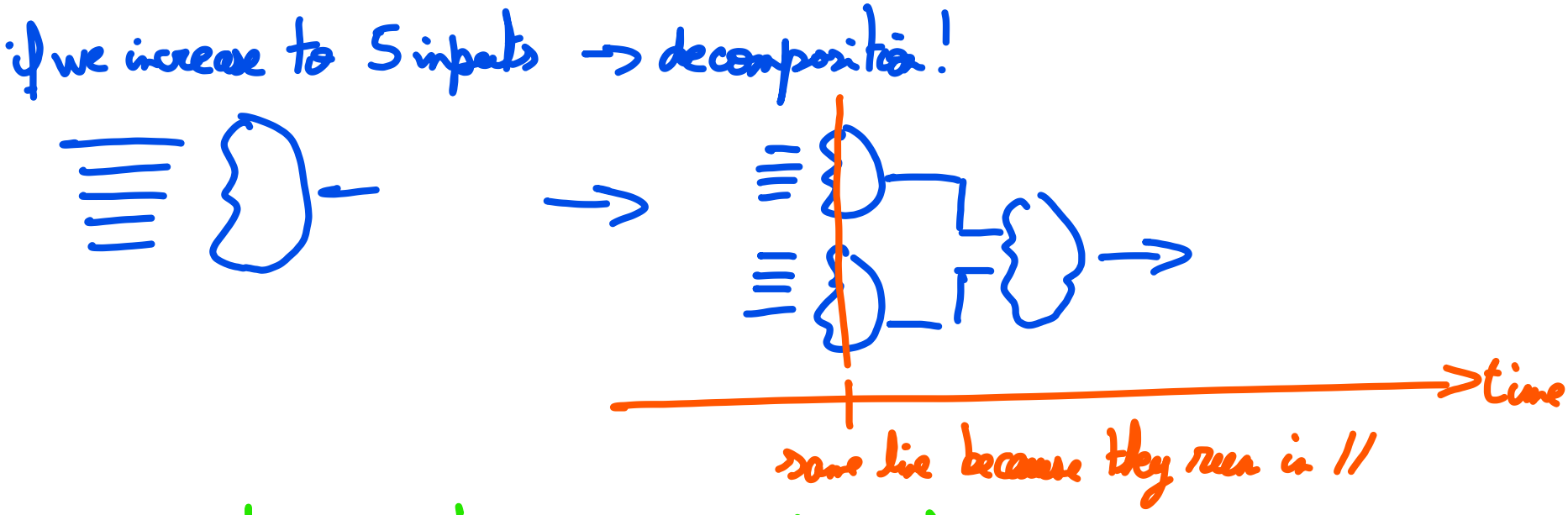
Circuit after synthesis (for an FPGA)

- This is what we get after the actual synthesis of the above model to FPGA circuit (after technology mapping): LUT = small



- Answer the following questions: → obtain a single block (LUT) in FPGA that looks at the table sends to buffer and then to output
  - Explain the difference between the two schematics?
  - What do you see on the left?
  - What would you expect for a bigger input word?

LUT = look up table



2, 3, 4 input = same timing
5 = more time

To optimize for power/area ⇒ less performance
To optimize performance ⇒ more power/area

6. Generating complex repetitive structures

Creating repetitive structures

- To instantiate "arrays of components" (1D or 2D) and duplicate HW in a **regualar way** use a loop with the `generate` statement:

  `for parameter generate`

- Generate `parameter` may be used to index array-type signals associated with component ports (we can "read" the index value); we will see this just right after this
- Exact syntax (1-D array):

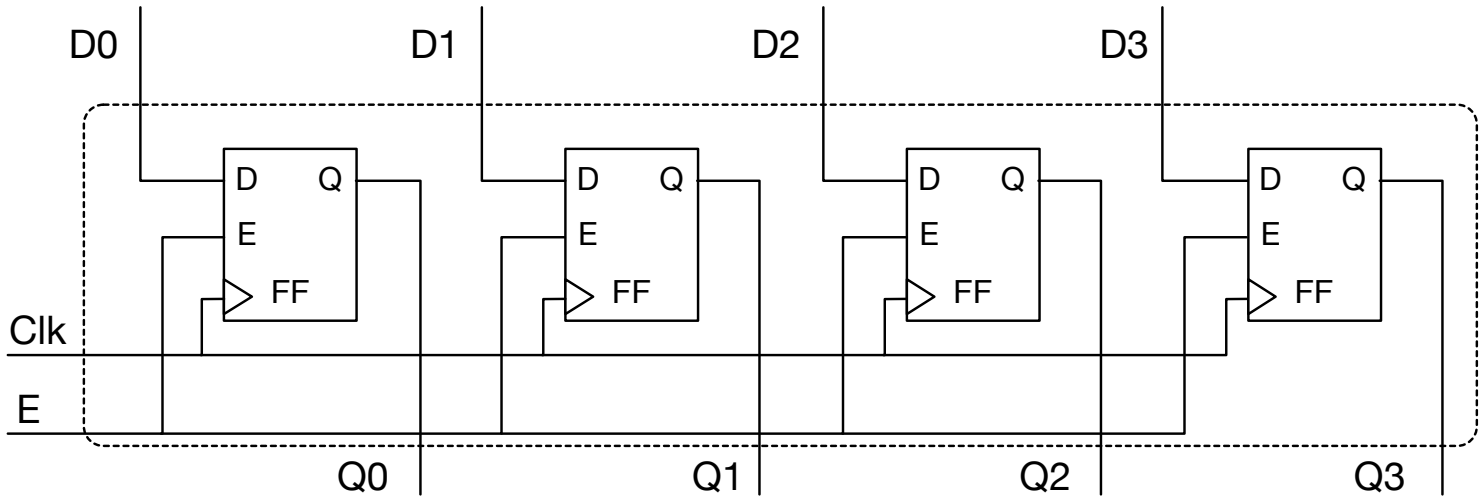
```
1 label: for parameter in range generate
2    concurrent_statements
3 end generate label;
```

- A label is **compulsory** with a generate statement
- 2D arrays are instantiated using two nested loops with two different generate parameters

7. Practical examples

Example1 – 4-bit register with parallel load

- Build a 4-bit word **synchronous** register
- The register has **parallel input** and **parallel output**
- When enable (E) is on, input data (D) is copied to the FF (output Q) for all 4 bits
- Signal `enable` is common for all 4 bits
- We first need to define D Flip-flop (basic element of our register)



Example1 – 4-bit register with parallel load

19,10: reset and clock are sensitive

```
1  -- D-FF with enable and asynch rst
2  entity dff is port (
3   rst,clk,ena,d    : in std_logic;
4   q                : out std_logic
5  ); end dff;
6  architecture synthesis1 of dff is
7  begin
8  -- ignore  variations at input d
9   process (rst, clk)
10   begin
11    if (rst = '1') then q <= '0';
12     elsif (clk'event) and (clk =
         '1') then
13      if (ena = '1') then
14        q <= d;
15      end if;
16    end if;
17  end process;
18 end synthesis1;
```

asynchronous reset

Asynchronous because rst before rising edge,
to made sync -> swap lines 11 and 12

- We use a basic FF with enable
- Reset is asynchronous
- Active on rising edge

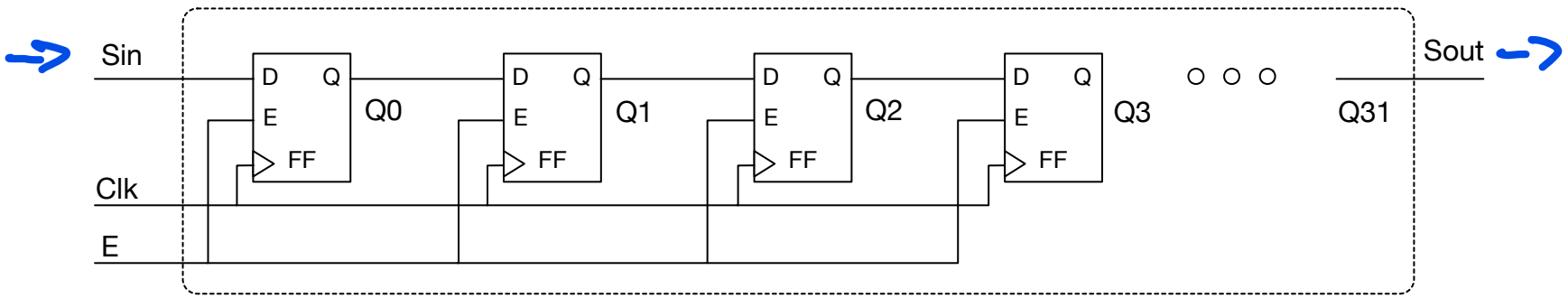
```
1  -- 4-bit register
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity reg_bank is port (
6   rst,clk,ena : in std_logic;
7   din  : in std_logic_vector (3 down 0);
8   dout : out std_logic_vector (3 down 0)
9  ); end reg_bank;
10
11 architecture gen of reg_bank is
12 -- component defined previously
13 component dff
14 port(rst,clk,ena,d : in  std_ulogic;
15        q             : out std_ulogic);
16 end component;
17
18 begin
19   gen_reg:
20     for i in 0 to 3 generate
21       regx : dff port map
22         (rst,clk,ena,din(i),dout(i));
23   end generate gen_reg;
24 end gen;
```

i is ≠ from python → i is constant

→ way easier to loop than rewrite!

Example2 – 32-bit wide shift left register

- Design 1 bit input, 1 bit output 32-bit **shift** register
- When `enable` set to 0, input is ignored, data remain stored
- When `enable` set to 1, the content of the register is shifted **from LSB to MSB** at each clock cycle
- When shifted to left, content of MSB is erased, first FF (LSB) gets the value that appears at the input of the register (all other bits shift for one place in MSB direction)
  - What does this operation do in arithmetic?
- Assume you have a basic D Flip-Flop



1st approach – 32 `port maps` of FFD

```
1  library IEEE;
2  use ieee.std_logic_1164.all;
3
4  entity D_REG32 is b (
5    Sin               : in STD_LOGIC;
6    Sout              : out STD_LOGIC
7  ); end D_REG32;
8
9  architecture STRUCTURE of D_REG32 is
10 -- need for a temporary signal that will be used
11 -- for "internal" wiring of the register
12 signal parallel_data: bit_vector(1,32);
13
14 component edge_triggered_FFD port(                -- Previously implemented DFF
15   D, clk, clr, en: in STD_LOGIC;
16   Q : out STD_LOGIC0);
17 end component edge_triggered_FFD;
18
19 begin
20 dff1: component edge_triggered_FFD
21  port map (Sin, clk, clr, en, parallel_data(1));
22
23 dff2: component edge_triggered_FFD
24  port map (parallel_data(1), clk, clr, en, parallel_data(2));
25
26 dff3: ...
27 end V1;
```

input and output

instantiate first FF (with input)

instantiate 2nd FF (with output of 1st FF...)

BORING

This is not a good idea: many cut & paste plus no scalability

2nd approach with conditional `generate`

- There is a possibility to use conditional generation such as:

```
1 dffx_others: if ((i>1) and (i<32)) generate
2   begin
3     dff: component edge_triggered_Dff
4       port map(parallel_data(i-1),clk,clr,en,parallel_data(i));
5 end generate dffx_others;
```

- Difference with a simple `generate`: all the concurrent statements that will be generated do not have to be the same!
- While this `if` statement may seem reminiscent to the `if-then-else` constructs used in SW, in VHDL it is different: for conditional generates there are no `else` or `elsif` clauses
  - ▹ Repetition is not in time like in computers (sequential machines), but in ASIC/FPGA "space" and is resolved during synthesis time, and not execution

2nd approach with conditional `generate`

```
1  -- usual definitions as in previous RTLs
2  begin
3  shift_reg: for i in 1 to 32 generate
4    begin
5    -- A. first
6    dffx_left: if i=1 generate                          -- This one connects input
7      begin
8      dff: component edge_triggered_Dff
9           port map (Sin, clk, clr, en, parallel_data(i));
10      end generate dffx_left;
11   -- B. 30 others
12     dffx_others: if ((i>1) and (i<32)) generate
13       begin
14        dff: component edge_triggered_Dff              -- These are parametrizable
15           port map (parallel_data(i-1), clk, clr, en, parallel_data(i));
16        end generate dffx_left;
17   -- C. last
18     dffx_right: if i=32 generate                      -- This one connects output
19        begin
20         dff: component edge_triggered_Dff
21           port map (parallel_data(i-1), clk, clr, en, Sout);
22     end generate dffx_right;
23 end generate shift_reg;
24 end reg32;
```

Note the split in the three parts (A. B. C.) of the VHDL module to enable IO connectivity

we forgot the assignment statement. temp <= D

Another solution is to map the Sin and Sout to //data | Even if no change of performance

Modifications to this RTL: your turn!

You can do this by hand:

- What can be done to avoid the split of the previous HW description into three different parts?
  - In another words how we could avoid conditional generate used to differentiate IO pins and internal signals?
- Add parallel output feature: at all times we have access to the 32-bit word at the output
- Add parallel input as an option, so that we can load the register either in parallel or in a serial way (you obviously need an extra control signal to do this)
- What else do we need to provide to the module description to enable this functionality?
- Could you think of an alternative way to describe a shift register?
- Add generics to enable arbitrary register size