Aalto University
School of Electrical Engineering

ELEC-E3540 Digital Microelectronics II
Writing synthesizable VHDL

Enrico Roverato
enrico.roverato@aalto.fi

12.03.2018

ELE Department

Logic synthesis

- **Logic synthesis** = the process of turning a behavioral model of a digital circuit (i.e. VHDL) into a design implementation in terms of logic gates (AND, OR, …)
- The circuit manufacturer provides the library of “standard cells”
  - each standard cell implements an elementary logic function
  - each cell is usually available in a multitude of driving strengths
- Synthesis is a highly automated task!

What is synthesizable VHDL?

- During the 6 exercises, you are learning the basics of VHDL coding and simulation
- However, not all VHDL constructs can be understood by the synthesis tool
- In addition, there are some “good practices” that will produce a better synthesis outcome

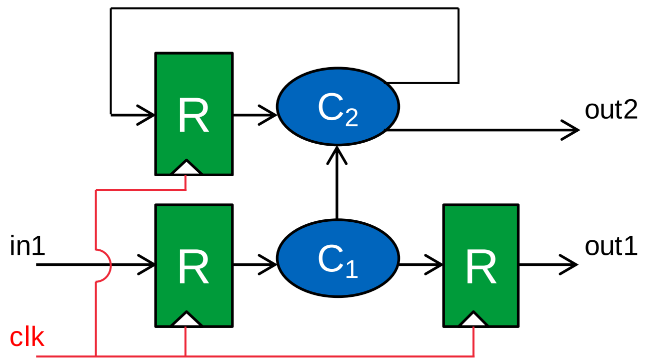
RTL coding style

RTL coding style

- RTL (Register Transfer Level) is the standard design abstraction used to model synchronous digital circuits with hardware description languages (Verilog, VHDL)

- **Learning the RTL coding style is <u>mandatory</u> in order to pass this course!**
  - exercises 4-6 and the final assignment will **not** be accepted, unless they follow the RTL style
  - even though everything seems to work!

RTL coding style



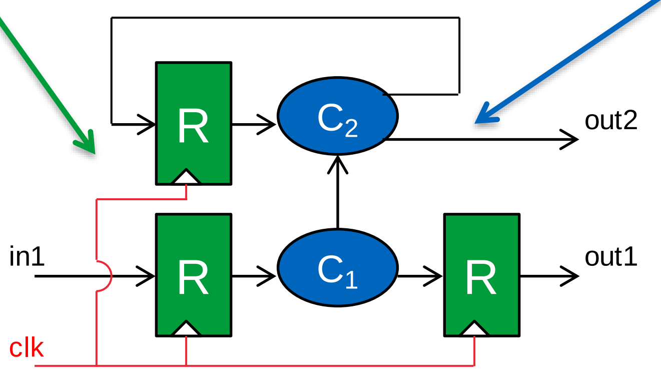
- RTL = describe a digital design in terms of
  - registers (memory elements) and
  - the flow/transformation of data between them (combinational logic)

- Only the registers have memory
- Only the registers are triggered by the clock signal
- Combinational logic only calculates outputs when inputs change
- Implementation resembles state machine

RTL coding style

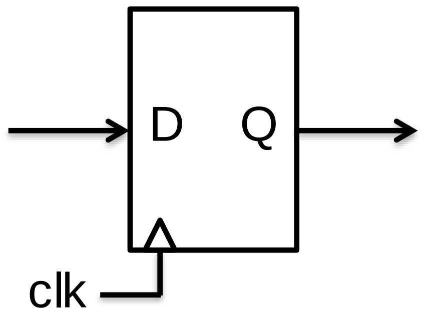
- In simple words, RTL is all about this:

REGISTERS vs EVERYTHING ELSE



How to model a register

- Simple edge-triggered D flip-flop



- With a process:

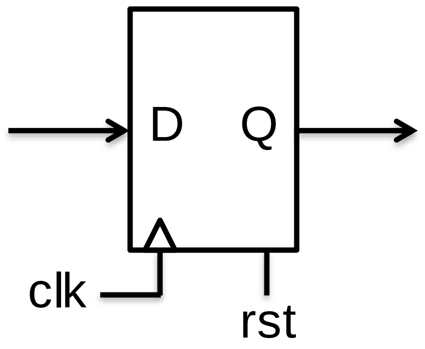
```
DFF: process(clk)
begin
   if rising_edge(clk) then
      Q <= D;
   end if;
end process;
```

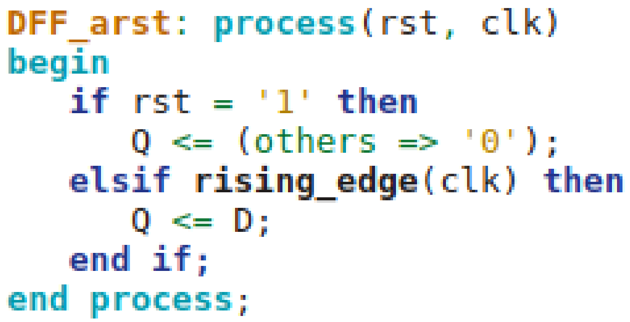
- Without a process (*concurrent signal assignment*):

```
Q <= D when rising_edge(clk);
```

How to model a register

- D flip-flop with *asynchronous* reset

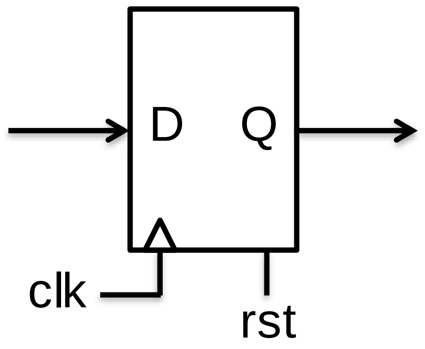


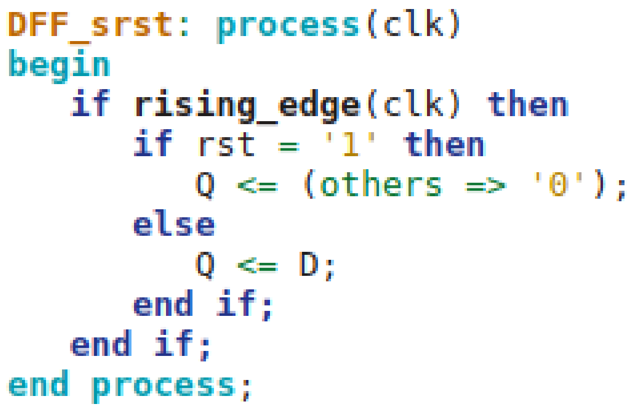


```
DFF_arst: process(rst, clk)
begin
   if rst = '1' then
      Q <= (others => '0');
   elsif rising_edge(clk) then
      Q <= D;
   end if;
end process;
```

How to model a register

- D flip-flop with *synchronous* reset

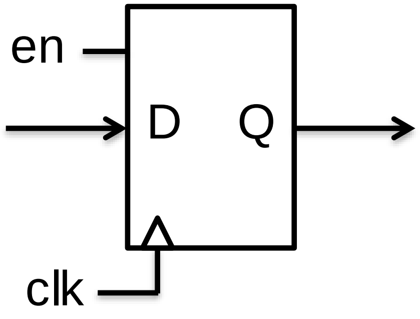




```
DFF_srst: process(clk)
begin
   if rising_edge(clk) then
      if rst = '1' then
         Q <= (others => '0');
      else
         Q <= D;
      end if;
   end if;
end process;
```

How to model a register

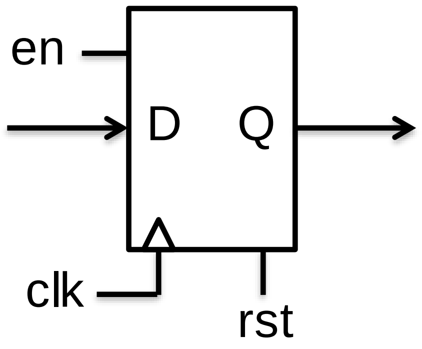
- D flip-flop with synchronous write enable



```
DFF_en: process(clk)
begin
    if rising_edge(clk) then
        if en = '1' then
            Q <= D;
        end if;
    end if;
end process;
```

How to model a register

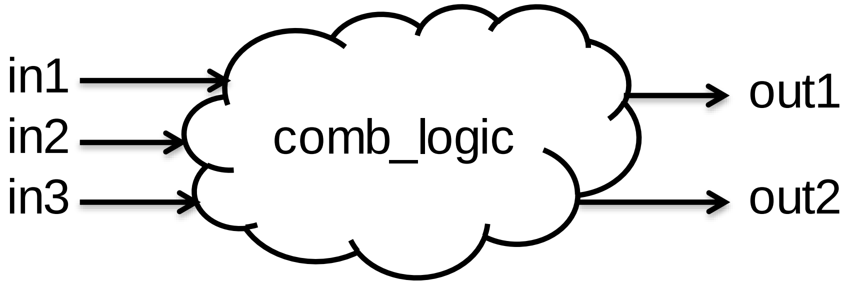
- The basic constructs shown in the previous slides can be combined to build more complex registers
- Example: D flip-flop with asynchronous reset and synchronous write enable



```
DFF_arst_en: process(rst, clk)
begin
   if rst = '1' then
      Q <= (others => '0');
   elsif rising_edge(clk) then
      if en = '1' then
         Q <= D;
      end if;
   end if;
end process;
```

How to model combinational logic

- Typically with a process which is sensitive to **all input signals** to the logic block
- In VHDL-2008, reserved word “all” can be used to replace a complete sensitivity list



```
comb_logic: process(in1, in2, in3)
    -- optional declarations
begin
    -- your VHDL code
end process;
```

```
comb_logic: process(all)
    -- optional declarations
begin
    -- your VHDL code
end process;
```

How to model combinational logic

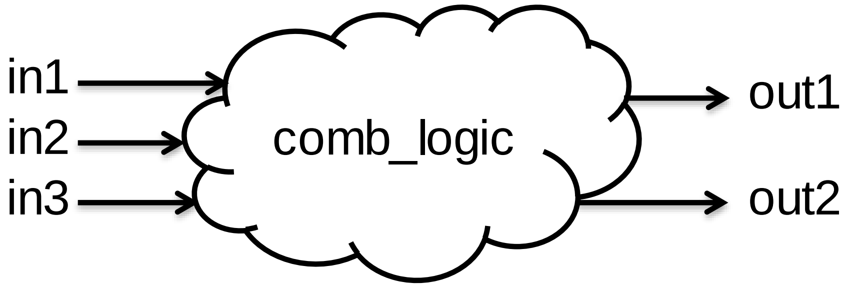
- IMPORTANT: **all outputs** of a combinational logic block must be assigned some **unique** value, for **every** value of the inputs!

- Otherwise the circuit has *memory* → it is no longer purely combinational

- Violation of this rule will result in “latch inferred” warnings during the elaboration phase of logic synthesis

How to model combinational logic

- Example 1: output not assigned under certain conditions

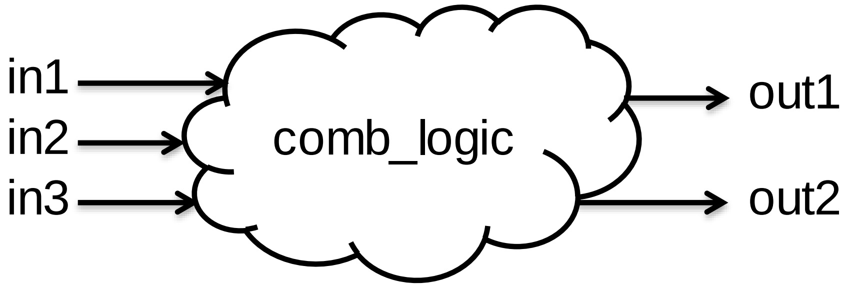


```
comb_logic_wrong: process(all)
begin
    if in1 = '1' then
        out1 <= in2 and in3;
        out2 <= in2 or in3;
    else
        out1 <= in2 xor in3;
    end if;
end process;
```

out2 not assigned when in1 = ‘0’

How to model combinational logic

- Example 1: output not assigned under certain conditions

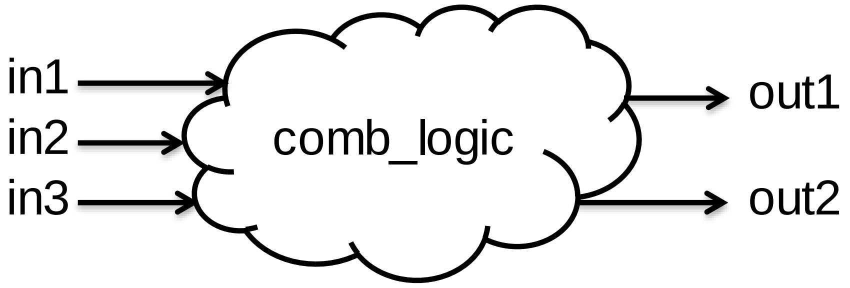


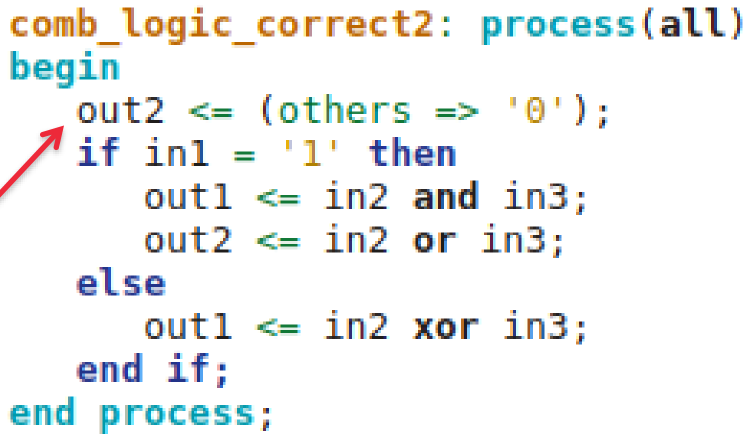
```
comb_logic_correct1: process(all)
begin
    if in1 = '1' then
        out1 <= in2 and in3;
        out2 <= in2 or in3;
    else
        out1 <= in2 xor in3;
        out2 <= (others => '0');
    end if;
end process;
```

Solution #1:
assign a value for out2 in all cases

How to model combinational logic

- Example 1: output not assigned under certain conditions



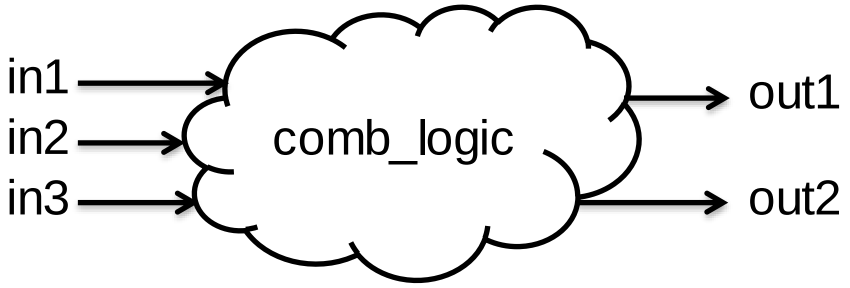


```
comb_logic_correct2: process(all)
begin
    out2 <= (others => '0');
    if in1 = '1' then
        out1 <= in2 and in3;
        out2 <= in2 or in3;
    else
        out1 <= in2 xor in3;
    end if;
end process;
```

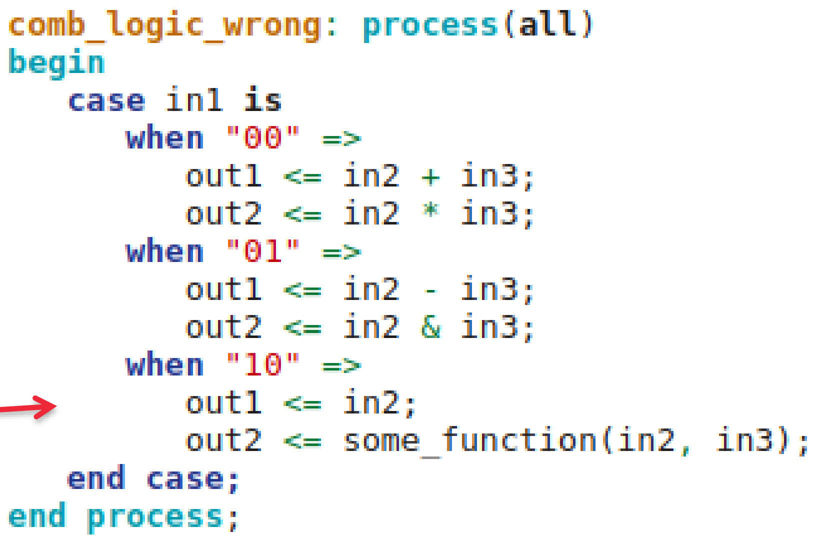
Solution #2:
initialize out2 before the if-statement

How to model combinational logic

- Example 2: incomplete conditional statement (don’t write these)



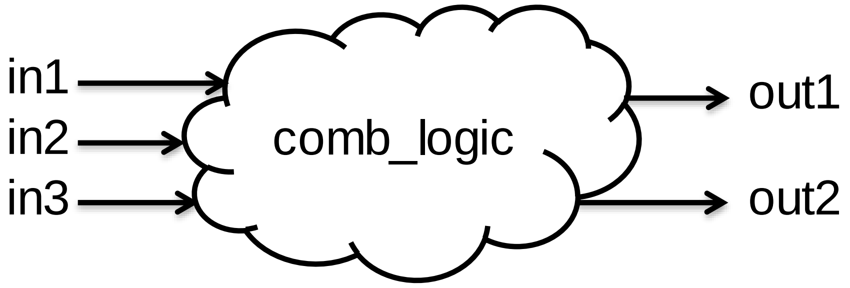
not defined what happens when in1 = “11”



```
comb_logic_wrong: process(all)
begin
   case in1 is
      when "00" =>
         out1 <= in2 + in3;
         out2 <= in2 * in3;
      when "01" =>
         out1 <= in2 - in3;
         out2 <= in2 & in3;
      when "10" =>
         out1 <= in2;
         out2 <= some_function(in2, in3);
   end case;
end process;
```

How to model combinational logic

- Example 2: incomplete conditional statement



```
comb_logic_correct: process(all)
begin
    case in1 is
        when "00" =>
            out1 <= in2 + in3;
            out2 <= in2 * in3;
        when "01" =>
            out1 <= in2 - in3;
            out2 <= in2 & in3;
        when "10" =>
            out1 <= in2;
            out2 <= some_function(in2, in3);
        when others =>
            out1 <= in2;
            out2 <= (others => '0');
    end case;
end process;
```

Solution: terminate conditional statements with else, when others to include all possible cases

Other useful hints

Read the book!

- Before starting the final assignment work, it is strongly recommended that you read:

1. P. J. Ashenden, **The designer's guide to VHDL**, *3rd ed.*
   - Chapter 21 - Design for synthesis

2. H. Bhatnagar, **Advanced ASIC chip synthesis** *(PDF in MyCourses)*
   - Chapter 5 - Partitioning and coding styles

Think “hardware”!

- When writing VHDL, have always in mind what you want the synthesis tool to implement!
- If you have no idea how your VHDL is going to be implemented, most likely the synthesis tool has no idea as well :-)
- Remember: the synthesis tool is stupid (and the static verification tool is even more stupid)

Avoid unessential features

- The simpler your VHDL code
  - the smaller the area and power consumption
  - the easier your life
  - the better your VHDL will be understood by the synthesis tool

- Ideally, your VHDL code should achieve the desired functionality with the minimum possible complexity!

Variables vs signals

- Students are usually tempted to make extensive use of variables within processes
  - Similarity with software-oriented programming languages
  - Avoid “delta cycle delay” of signal assignments.
  - Consequent assignments to same signal within one clock cycle usually do not result in the intended (programming style) behaviour.
- **However, <u>try not to abuse variables</u>**
- Reason: variables are abstract, signals are real (i.e. physical wires in your chip)
- Using too many variables
  - is in contrast with the “think hardware” guideline given earlier
  - will cause troubles, because you don’t understand how they are mapped to a physical circuit

“After” statements

- Statements like after, wait for and similar are meaningless for the synthesis tool
  - don’t waste your time with those
- VHDL should describe the ideal logic behavior of your circuit
  - timing is taken into account during synthesis and P&R
- P.S. = The simulation testbench must contain wait statements! This is not a problem, as the testbench will not be synthesized

Initial values

- Initial values (e.g. in signal declarations) are ignored by the synthesis tool
  - You MUST initialize your signal with reset, or assing constants to them with proper signal assignment (not in signal declaration)
  - Not doing this is very risky.
  - Again, you can still use them in the simulation testbench

- If you want to be sure that your memory and/or registers are initialized to 0, you must include a RESET signal
  - Wither synchronous or asynchronous

Naming

- The synthesized netlist is written in VERILOG

- In your VHDL, avoid using names that are reserved words for VERILOG
  - examples: “input”, “output” can be used in VHDL, but they will cause problems in VERILOG

- As a general rule, use your common sense
  - avoid any potentially dangerous words :-)

Dealing with numbers

- Simple arithmetic operations are well supported by synthesis tools
  - package ieee.numeric_std
- Correct way to implement an addition:

```
sum <= STD_LOGIC_VECTOR(UNSIGNED(a) + UNSIGNED(b));
```

- Correct way to address a memory cell:

```
sel     <= to_integer(UNSIGNED(addr));
RAM_out <= RAM_array(sel);
```

- Don’t implement e.g. the internal adder architecture by yourself! Let the synthesis tool do this for you

A couple of words on TCL

Tool Command Language

- Tool Command Language (TCL) is the scripting language used to control all tools needed for the digiflow
  - QuestaSim, Design Compiler, Formality, Encounter, PrimeTime
- Every operation performed through the GUI corresponds to one or many TCL commands
  - the inverse is not necessarily true :-)
- Worth learning at least the very basics of the language
  - if you want to learn more, use google!

Command invocation

- A TCL script consists of several command invocations:

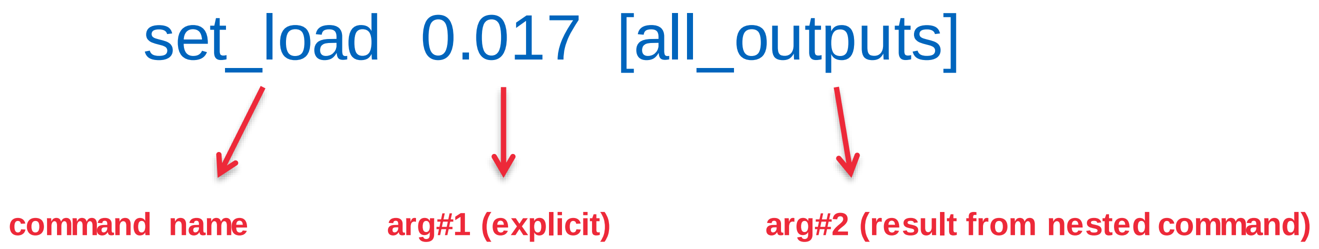
```
command1  arg11  arg12  ...  arg1N
command2  arg21  arg22  ...  arg2N
...
```

- The list of valid command names and arguments can be found in each tool's text reference manual

Command substitution

- Square brackets [ ] allow to execute commands in a nested fashion
  - the command inside brackets is executed first, and its result is used as argument to another command

- Example (Design Compiler):



```
set_load 0.017 [all_outputs]
```

Variables

- Variable declaration:

```
set varname varvalue
```

- Variable substitution (i.e. using the variable's content):

```
$varname
${varname}
```

- All variables are manipulated as strings!

- Example:

```
set  loadvalue  0.017
set_load  $loadvalue  [all_outputs]
```

Quotes

- Quotes are used to group many space-separated words into a single argument
  - for example, to create lists

- Two tipes of quotes used in TCL
  - double quotes " ": substitution **does** take place within them
  - curly braces { }: substitution does **not** take place within them

Quotes

- Example:

```
set  var1  value1
command1  "$var1  [command2]"
command1  {$var1  [command2]}
```

- Assume that executing command2 returns value2
  – the line with double quotes uses string value1 value2 as first argument of command1
  – the line with curly braces uses string $var1 [command2] as first argument of command1 (no substitution!)