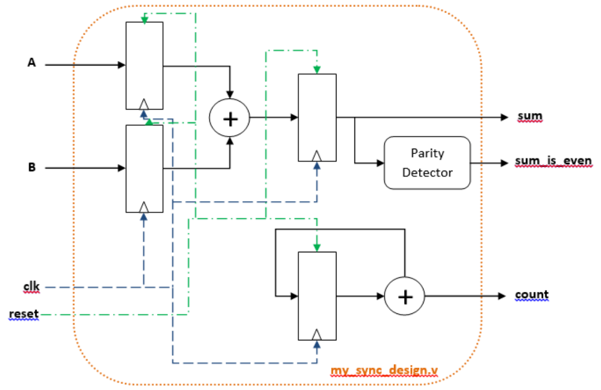
ELEC 2531

Lab 2

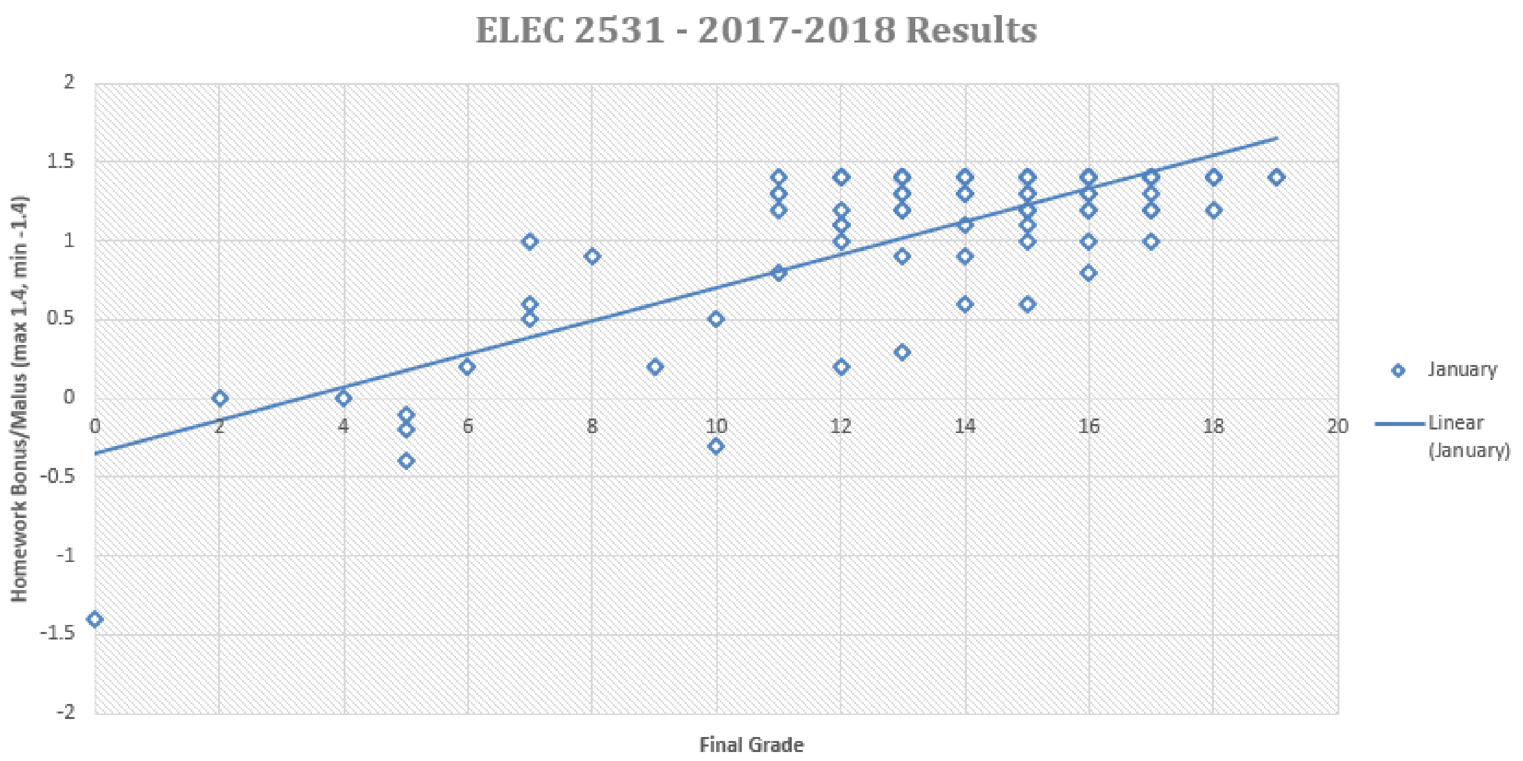
Introduction to **synchronous** circuits:

- **Design** using the (System)Verilog language
- **Synthesis** and **Place & Route** using Quartus
- **Simulation** using ModelSim

-> Case study: a custom summer & parity detector

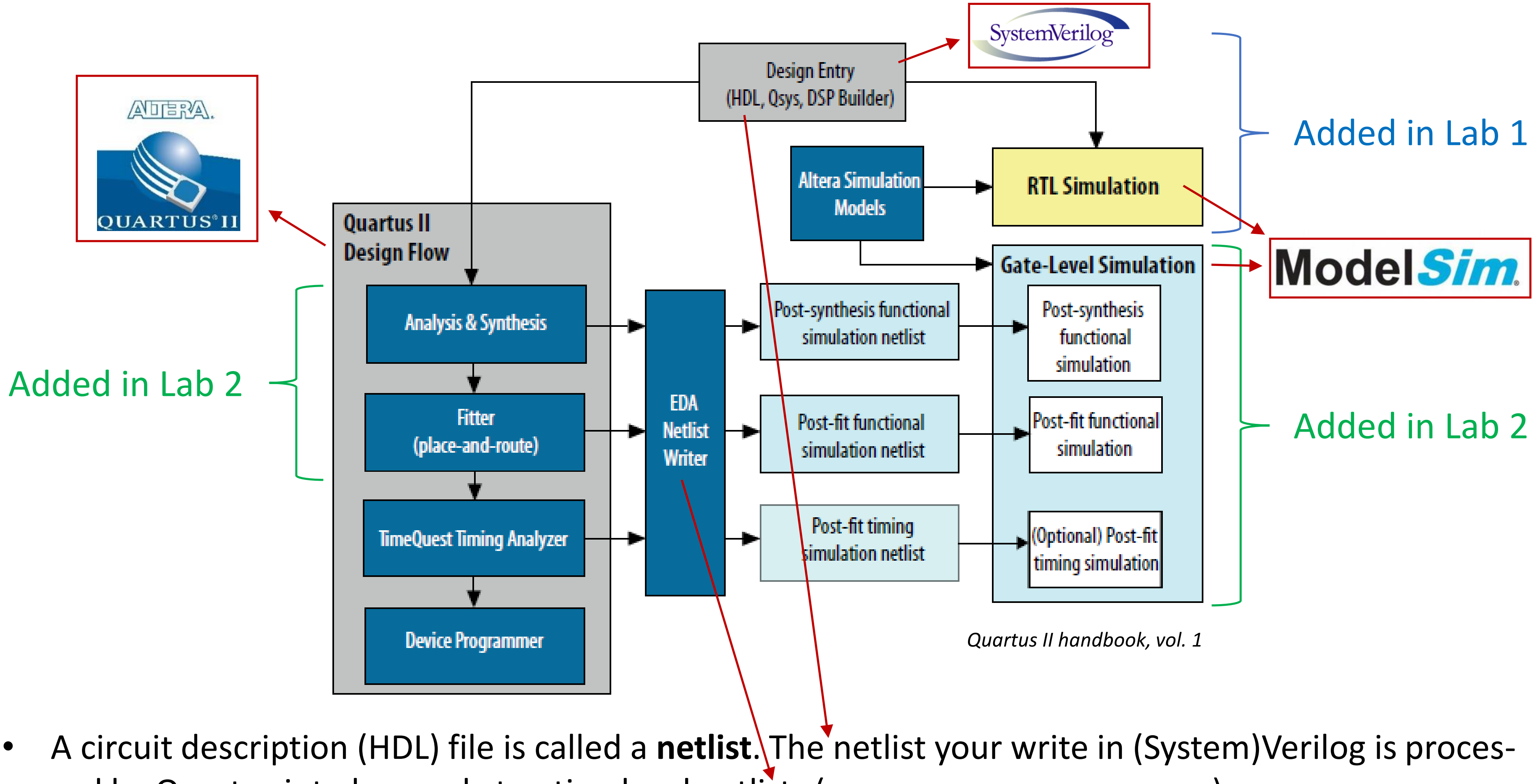


- **Let’s insist on a few things from last week:**
  - *« **Contact**: as much as possible through the student Moodle forum. Contact us by mail only for personnal matters ! »*
  - *« Do all this without delays and the evaluations will be a complete piece of cake. Do the opposite and it might turn to be highly complex… »*



Electronic Design Automation (EDA) Flow for FPGA

- We use a programming language - (System)Verilog - to **describe** **(HDL)** a circuit (our design entry) that we can simulate at different levels (ModelSim) and use to program a physical device (our FPGA, using Quartus).

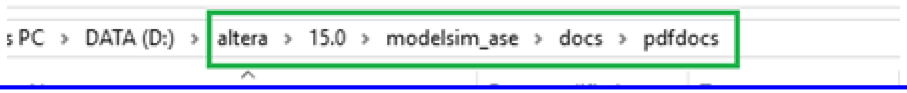


SystemVerilog
Design Entry (HDL, Qsys, DSP Builder)
Added in Lab 1
ALTERA
QUARTUS II
Altera Simulation Models
RTL Simulation
ModelSim
Quartus II Design Flow
Gate-Level Simulation
Added in Lab 2
Analysis & Synthesis
EDA Netlist Writer
Post-synthesis functional simulation netlist
Post-synthesis functional simulation
Fitter (place-and-route)
Post-fit functional simulation netlist
Post-fit functional simulation
Added in Lab 2
TimeQuest Timing Analyzer
Post-fit timing simulation netlist
(Optional) Post-fit timing simulation
Device Programmer

Quartus II handbook, vol. 1

- A circuit description (HDL) file is called a **netlist**. The netlist your write in (System)Verilog is processed by Quartus into lower abstraction level netlists (`Synthesis + Place&Route`).

Documentation: where and why ?

- We use Altera's Quartus suite 18.0 (ModelSim + Quartus) Lite Edition
- (System)Verilog: Asic-World, Verilog 2001 Ref Guide or System Verilog Ref Guide
- ModelSim: 

  s PC › DATA (D:) › altera › 15.0 › modelsim_ase › docs › pdfdocs

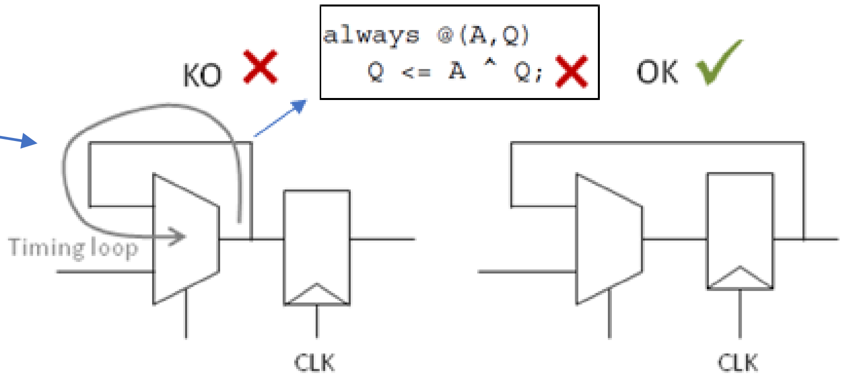
- Quartus II (V.18.0): Handbook -> interesting chapters for this course are
  - Vol. 1 (Design & Synthesis) : chap. 1 (Projects), 12 (HDL Coding), 16 (Synthesis),
  - Vol. 2 (Design Implementation & Optimization) : chap. 4 & 5 (Scripting your flow)
  - Vol. 3 (Verification) : chap. 1 (Simulating), 13 (Signal Tap II), 18 (Quartus II Programmer)
- Why and how read documentation ?
  - As your education goes on, we expect from you to be more and more able to find information by yourself in order to understand new tools, concepts or technologies.
  - You hence have to know where to find the right documentation and what it contains. Obviously these documents are prohibitively long and should not be read at once, but used as something to refer to.
  - All this is especially true in electronics ! At the end of your master you should be really used to cope with this kind of large and verbose documentation, so you should start as soon as possible.

Quick Restructuring: HDL coding good practices

- Verilog vs SystemVerilog:
  - “logic” is too generalist but “reg” can be confusing in combinational structures
    - Therefore there is no good recommendation, but try to always have in mind that you manipulate physical stuff (wires or registers) and not variables (like in C, Java, Python, etc.).
  - In SystemVerilog, use the precise “always” keyword:
    - always_comb (combinational), always_ff (flip-flop, e.g. for registers) and always_latch (latches, rare usage in this course)

- In general,

1. Avoid combinational loops



```
always @(A,Q)
  Q <= A ^ Q;
```

D. Bol, « Verilog Coding Guidelines for robust VLSI », ICTEAM, v1.2

2. Always reset your sequential circuits
   - In order to have them in a known state
   - Does not concern combinational blocks
   - Does not apply for large register arrays (e.g. register file) or memories

3. Avoid unwanted latch insertions
   - By having an “`else`” possibility in “`if`” statements
   - By having a “`default`” possibility in “`case`” statements
   - Concerns only “always_comb” structures, not “always_ff” or “always_latch” because there are already a register or latch there.

Quick restructuring: testbench

- Let’s review this lab’s testbench:
  - The testbench is another module, one level higher than the design’s top-level module, given that it has to instantiate it
  - The simulator time unit and precision are set using the « `timescale » preprocessor directive
  - Some « logic » elements have to be defined, at least for the top-level (or DUT) I/Os
    - Now, we also need 1-bit inputs for the clock and reset of the sequential blocks of our circuits
    - In testbenches, DUT’s inputs should be reg while outputs should be wires (or all “logic” in SystemVerilog)
  - A non-synthezisable « initial » structure is used to apply external stimuli to the top-level inputs
    - The testbench clock can be generated simply using a delay operator (# => non-synthesizable). Don’t forget to initialize the clock to 0 or 1
    - Given if your reset is active-high or active-low, it should be initialized to 1 or 0 and then set to 0 or 1 after a small delay and before applying any other inputs (other than the initial values)

```
`timescale 1ns/1ps

module my_wonderful_testbench();

   reg          clk, reset;
   reg  [31:0]  A, B;

   wire  [31:0] cnt, sum;
   wire         even;

   // Instantiate the DUT.
   my_sync_design dut (
      .clk(clk),.reset(reset),.A(A),.B(B),
      .sum(sum),.count(cnt),.sum_is_even(even));

   // Generate a (non-synthesizable) clock
   always  #10 clk = ~clk;

   initial begin
      clk      = 1'b0;
      reset    = 1'b1;
      A        = 32'h44;
      B        = 32'h45;
      #100;
      reset    = 1'b0;
      #20;
      A        = 32'h45;
      B        = 32'h58;
      #20;

      // (...)
   end
endmodule
```