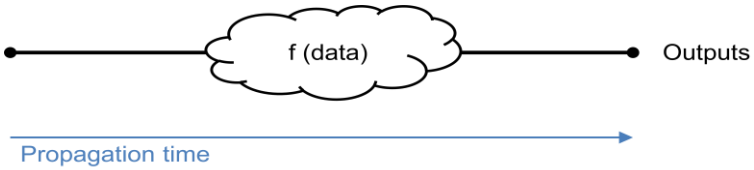
Introduction to synchronous designs PART A

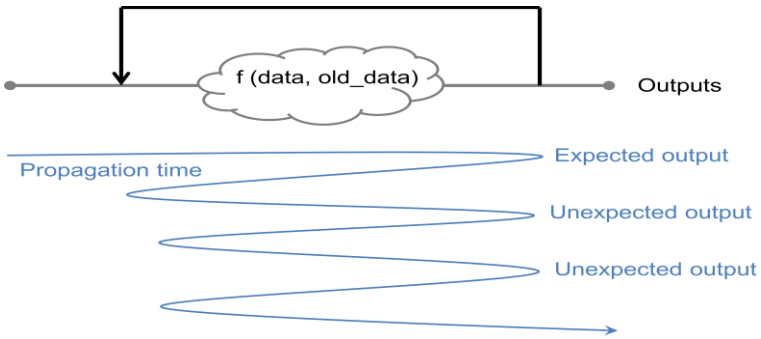
In the previous lab session you learnt about **combinational circuit designs**. While cute, friendly and easy to use, these circuits are limited to direct operations on the inputs without feedback: `out = f(in)`. This makes the design of more complex operations (such as counters) almost impossible. So, in this session we will introduce **synchronous circuit designs** (aka "sequential logic"), which are able to perform functions of their inputs *and* current state, `out = f(in, state)` through the use of a clock signal and registers.

So, this is what we had previously:



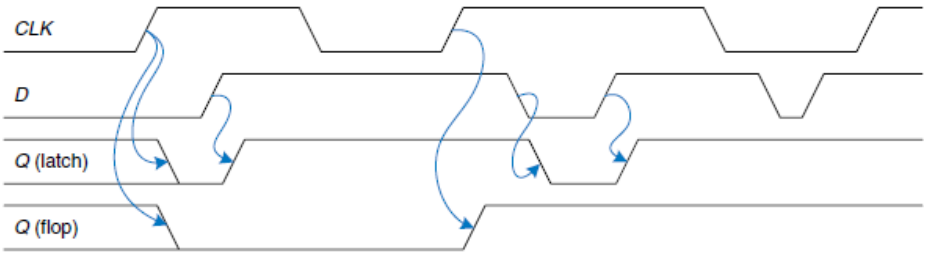
Input data is transformed through a combinational block 'f', yielding a specific output after a certain signal propagation delay.

Say we now want to implement a simple counter doing "a = a + 1", how can this be done? A naive implementation would give this:



There is a problem here: in the initial cycle the result will be correct (the feedback has not yet been able to affect the outcome), but very quickly afterwards the results will become... interesting. Building a circuit based on such unpredictable behaviour is tricky business indeed. You can already red print in your digital electronics mind toolbox that **you should never design circuits with combinational loops**.

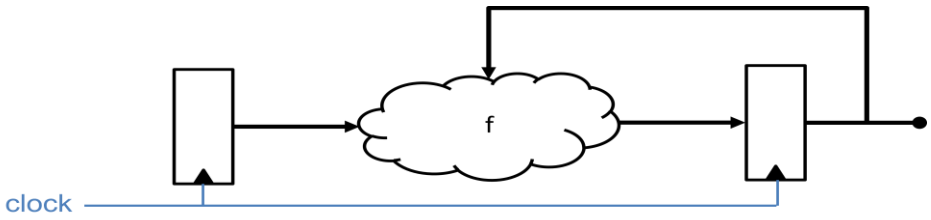
So instead of building the circuit that way, sections of the circuit are temporally isolated through the use of a "register" block which 'captures' its input at the clock edge, and outputs the captured value. Let's illustrate this with a figure (3.15) from the book:



The clock (**CLK**) is a varying signal[1] from 0 (low level) to 1 (high level, rising edge) and 1 to 0 (falling edge). As you can see, these edges are represented as slanting bars: the point is to highlight that they always take a certain of amount of time.

A register (aka "D flip-flop") is set to be sensitive to one of these edges (here the rising one) and hence modifies its internal value only during this edge by capturing its inputs (here **D**). The rest of the clock period, the register simply outputs (**Q (flop)**) its stored value regardless any input variation (unlike the latch (Q (latch)) which is level-sensitive, but never mind for today's lab).

Therefore, with registers, we are safe to apply a feedback without corrupting the result. Our simple circuit above thus becomes:



In this way, the output is always correct since the feedback loop is temporally isolated. At this point, if the register operation is still vague, please read and fully understand sections 3.2.3/.4/.6 of your book.

Please already note as well that using clocked circuits implies **timing consideration** (see the important section 3.5 of your book). Hence, it is not sufficient to be only aware of functional specifications of sequential circuits: you also have to take care of their timing specifications.

Doing that in Verilog

In the first, purely **combinatorial case**, there is no clock, no fuss, and the output comes as soon as it propagates through the electronics and is available. Here is an example of such a circuit:

```
module add(input [3:0] a, input [3:0] b, output [4:0] out);
    assign out = a + b;
endmodule
```

[1] Generally periodic.

This is exactly the same as:

```
module add(input [3:0] a, input [3:0] b, output reg [4:0] out);
    always_comb
        begin
                out = a + b;
        end
endmodule
```

So what happened here? The output is now typed 'reg', which means it is modified inside an always block (it is not necessarily a register, and is not in this case). Then we have an always block that acts exactly as an "assign" but can encompass several assignations with or without interdependencies.

Now, a **sequential circuit** can be implemented in a similar way:

```
module add(input clock, input reset, input [3:0] a, input [3:0] b, output reg [4:0] out);
    always_ff @ (posedge clock, posedge reset)
        begin
                if(reset)   out <= 5'b0;
                else        out <= a + b;
        end
endmodule
```

In this scenario, whenever whatever is in its sensitivity list (the parentheses after the '`@`' symbol) changes, the block is re-evaluated. All we do when that happens is to set `out` to be equal to the sum of `a` and `b`. Thus '`out`' is only set *after* a rising edge on the '`clock`' signal. And in this case, '`out`' is the output of a physical register, since an `always_ff` block is used to force implementation using a flip-flop. Also, since we are not designing a program but a circuit here, we like to have a '`reset`' signal that puts everything back into a known state at the beginning. More on sequential logic design using an HDL in your book, section 4.4[2].

Notice the use of '<=', a **non-blocking** assignment, instead of '=', which is a **blocking** assignment:

`three <= 4'd1 + 4'd2;`
`four  <= three + 4'b1;`
 	
These are done in **parallel**, so
four = "old value of *three*" + 1 = 1
(and 4 only on the next cycle).
	`three = 4'd1 + 4'd2;`
	`four  = three + 4'b1;`
	
	These are done **sequentially**, so
	four = "*three*" + 1 = 4.

To avoid further confusion, let's retake and complete some statements of the previous lab about noticeable syntax differences between Verilog and SystemVerilog:

- *"**reg**"/"**wire**" vs "**logic**": in Verilog, the main types you will use to define internal nodes are **wires** and registers (**reg**). The latter is used to build sequential blocks while the former is used for combinational blocks as well as to connect blocks and modules. In SystemVerilog, life is made easier as these two are replaced by the single "**logic**" type. However, and especially for*

[2] See the Harris & Harris (ARM ed.) book, section 4.4, page 193.

*neophytes, this "stronger" keyword can really be a fake friend. Indeed, using "**reg**" and "**wire**" could be much more beneficial for you in order to really visualize and understand the circuit you are describing in Verilog. If you choose to only use the "**logic**" type, it is then primary to always understand if you manipulate a register or a wire.*

- *"**always**" vs "**always_comb**"/"**always_ff**"/"**always_latch**": the "**always**" structure is another important feature of Verilog[3]. Here, on the other hand, SystemVerilog extends it by allowing to use "**always_comb**" (for combinational purposes), "**always_ff**" (for flip-flops) and "**always_latch**" (for latches). (…) But already note that, in SystemVerilog, it is good practice to use the precise keywords. This way, not only you will have a clearer circuit description for yourself but also, the compiler will be able to better detect building errors in these structures.*

Unlike what was said, using the "reg" type does not strictly imply a register. As stated above, whenever you use an "always" structure, your assigned node should be a "reg" type in Verilog (or a logic in SystemVerilog). But if the always is an "always_comb" (implying a combinational block), your node will obviously not be an actual register, because they are a purely sequential block.

That was a (very) quick introduction to sequential Verilog modelling, you will find more details and implementation notes in your reference textbook.

Synthesis

In the previous lab session, you performed **behavioural** simulation. Indeed, the ModelSim simulator interpreted your Verilog code directly, and displayed the results. Sadly, this cannot run on any physical device this way; you may have noticed, for instance, that it takes zero time for operations to complete, which makes no physical sense. In order to design an integrated circuit chip (ASIC – Application-specific Integrated Circuit) or to program an FPGA (Field-Programmable Gate Array), your Verilog code has to be processed in a number of steps.

For **FPGAs** (which are the circuit types considered in this course), these steps are:

1. Analysis (checks for syntax errors in your Verilog).
2. Synthesis (converts the Verilog to a netlist in the tool's internal format).
3. Mapping (transforms 'standard' gate models to physical elements found on the device).
4. Place & Route (selects which physical elements are to be used on the device, and calculates the signal routing between them).
5. Assembler (creates a file that can be used to program the FPGA).

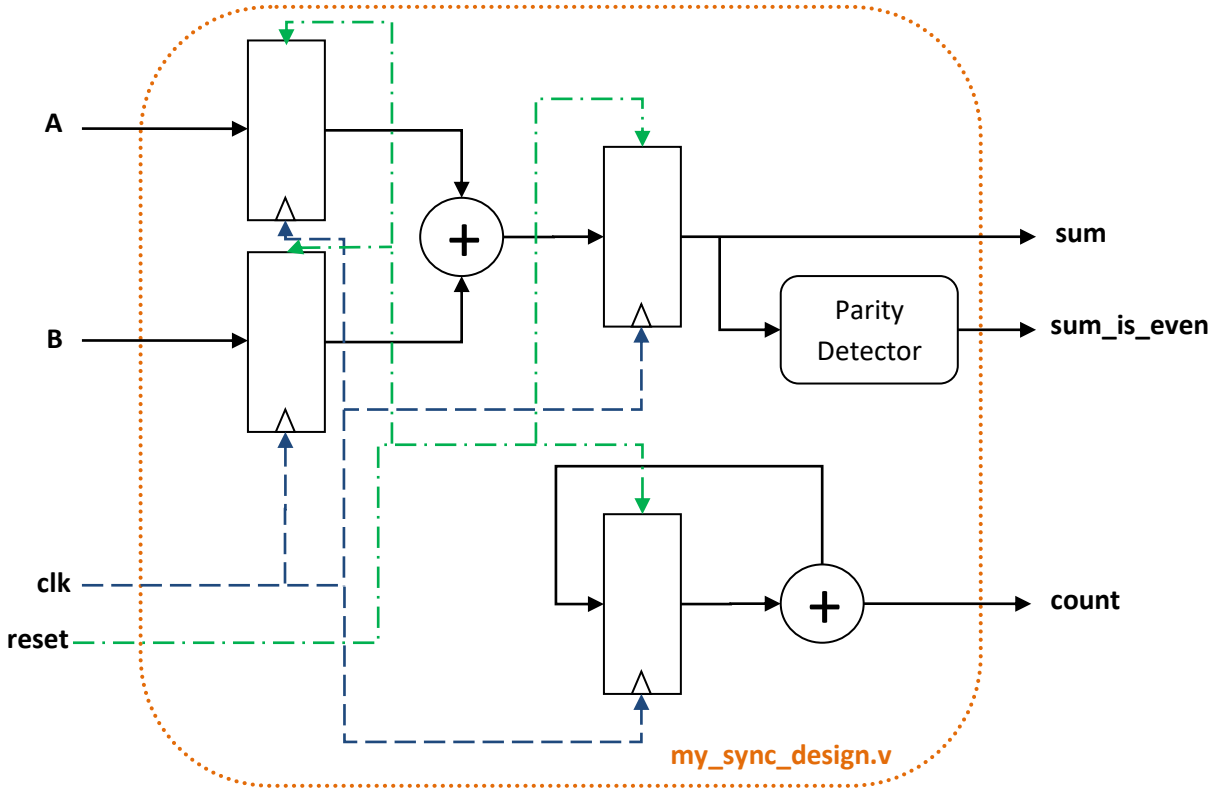
These steps are performed by a synthesis tool, **Quartus** in the case of Altera FPGAs (you should note that, in Quartus, steps 3 and 4 are combined into a single "Fitter" step). For ASIC design, see next year's ELEC2570 course ;-).

[3] For a good introduction, see the Harris & Harris (ARM ed.) book, HDL example 4.17, page 194.

Quartus synthesis tutorial

Okay, now that you have the basics, let's get started!

For this session, we have prepared a small example circuit for you. It's a simple adder, a parity detector at the output, combined with a clock counter (commonly called a timer, but hey!):



The Verilog design of this circuit can be found in "my_sync_design.sv" (see Moodle). Please take a moment to look through this file, and understand its design. The other file given is the testbench file; please take a look at it now also.

Before going further, let's make two remarks about the Quartus software:

- As stated on Moodle, **the version you must use is the 18.0 for Cyclone IV device.** Given that software edition is not the strong side of Altera, we cannot ensure the backward compatibility between the different versions (we had problems between version 13.0/1, 14.0 and 15.0). To avoid problems along the semester, use the 18.0. Besides, it should be the **Lite edition** (free). Please note as well with the Lite edition, it is likely that a 30-day trial is activated by default for the complete version; if so you should have pop-ups telling you that you have X days left, like this:

```
Info (292036): Thank you for using the Quartus II software
30-day evaluation. You have X days remaining (until Xxx
XX, 20XX) to use the Quartus II software with compilation
and simulation support.
```

Obviously **you can ignore these messages**.

- Secondly, we will learn during the lab how to run ModelSim from Quartus. Be warned that **opening several instances of ModelSim from the same Quartus project can prevent it from working well**. Therefore a good practice is to always close it if you want to re-run a simulation. Normally you should even have error messages and ModelSim should not open if an instance already is.