Introduction to Digital Electronics Design Part A

In an analog world, full of continuous values and quantities, **digital electronics** is an audacious attempt to bring order from chaos. With noise contaminating all measurements, made worse by parasitic effects such as temperature and materials, we chose to limit the entire spectrum of possible values to just ‘1’ and ‘0’. Analog values can be sampled at an arbitrary resolution for digital conversion, and digital values can be converted back to analog values. In between, in the digital part, magic happens.

Digital systems can be *complex*. And I mean huge, crazy big systems containing billions of transistors. Thus, one of the fundamental principles of digital electronics is finding ways to limit complexity, so that we (admittedly awesome) electrical engineers can understand the system in its entirety. This is achieved through ***abstraction***, or black boxes. At each level, we only need to know the external interface of each block in order to use it, without necessarily thinking about all the underlying details.

So instead of having to consider individual transistors, we consider a small group of them together forming a logic gate, which has a specific purpose and well-defined interface. Take for example an inverter gate; although it typically contains two transistors, we only need to consider the interface (1->0 and 0->1) to be able to use it. There are many standard gates (corresponding to basic Boolean operations), and custom gates can of course be designed to do...something special! Above that, we can define modules containing several gates that do even cleverer things, and so on up until we have an awesome system.

So, **building a digital system is as simple as connecting gates and modules together**!

Hardware Description Languages

In the days of lore, electronic circuits, regardless of their complexity, had to be designed through schematics. These were complex, hard to simulate and generally error-prone. With the introduction of **hardware description languages** (HDL), a specification was established to allow circuits to be concisely and accurately modelled, thus giving designers a way of providing not only an exact description of the circuit, but also automatic design verification tools. Verilog is one such HDL, popular in the electronics industry. It is a subset of **SystemVerilog** that will be covered in this course, which introduces a myriad of really cool testing features, as we shall see later.

Now, this is important: Verilog is a programming language, but not similar to the classical high-level language you are used to like C, Java, Matlab or Python: **Verilog is a circuit description**. The difference is subtle but crucial. In high-level programming languages, the processor typically executes a number of instructions sequentially. In Verilog, there are no “operations”, only connections between registers, gates or blocks. Therefore, in Verilog all operations are parallel and can only be made sequential by inserting registers or special structures, as we shall see in the next few weeks.

Modules

So, Verilog is based on **modules** that are **connected** to each other. A Verilog description of a circuit is contained in a module, which can instantiate basic logic gates (and, or, xor, etc.) and one or more other modules. A **module** is characterized by its ports (inputs/outputs) which allow communications with other modules or the outside world. A module has to be **defined** once (to describe its ports and internal behaviour) but can be **instantiated** at will (we add and connect an instance of the module to our design).

We can define a module like this:

```
module my_cool_module(
    input [3:0] my_first_input,
    input [7:0] my_second_input,

    output      my_output
);

    // Do stuff.

endmodule
```

So, what did we do there? We defined a module, that we call "my_cool_module", which has two inputs and one output. The first input is 4-bit wide while the second is 8-bit. Both are defined from the highest bit (3 or 7) down to 0 in order to match the standard bit-ordering convention (aka "*little-endian*"[1]) that has the LSB being bit 0. While it is possible to change the order (to "*big-endian*"), you should never do it in this course! When no bits are specified, like for "my_output", it corresponds to a 1-bit wide signal.

You can define as much inputs or outputs you want, with various sizes. Their definition order is not important, but please observe that they are separated by commas (',') except for the last one, which is followed by ');'.

Suppose that our module does something useful, and we would like to use it. Here's how to instantiate it in a higher-level module:

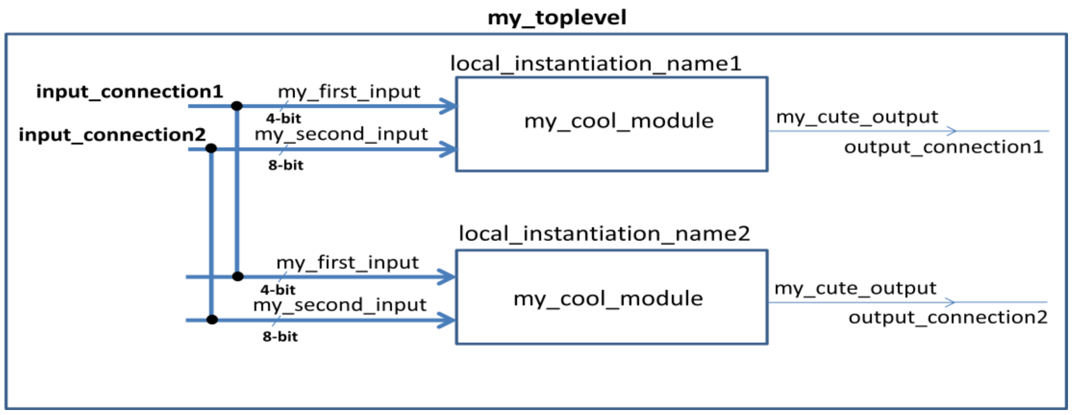
```
module my_toplevel();
    wire [3:0]  input_connection1;
    wire [7:0]  input_connection2;
    wire        output_connection1;
    wire        output_connection2;

    my_cool_module local_instantiation_name1 (
        .my_first_input     (input_connection1),
        .my_second_input    (input_connection2),
        .my_output      (output_connection1)
    );
    my_cool_module local_instantiation_name2 (
        .my_first_input     (input_connection1),
        .my_second_input    (input_connection2),
        .my_output      (output_connection2)
    );
endmodule
```

[1] See the Harris & Harris (ARM ed.) book, HDL example 4.2, page 178.

Okay, so **to instantiate a module** we need to specify what module (using the "definition" name), and then give it a local name (must be unique and is case-sensitive). Next we need to connect the input/output ports, which is done via the dot ('.') operator: use '.' followed by the port name in the target module, and connect that to the local wire in the parentheses. Please observe as well that, like for module definition, the inputs and outputs are separated by commas (',') except for the last one, which is followed by ');'. Besides, the order of the inputs and outputs doesn't matter, as long as they are clearly identified.

Simple, huh?

The notion of **levels** (and hence **hierarchy**) corresponding to "where" a module is instantiated is important, we will soon get back to it. Here is a graphical representation of the module "my_toplevel" defined above:



Verilog vs SystemVerilog

As stated previously, we will use for this course the description language SystemVerilog instead of Verilog. Hence, we should clarify some points to avoid further confusions.

Even if you would not see it at first glance, there are quite big differences between Verilog and SystemVerilog in the sense that: "*SystemVerilog includes a set of extensions to the Verilog HDL to help engineers design and verify larger and more complex designs.*" In other words, SystemVerilog was built on top of Verilog and allows much more features. Besides, there are also some syntax differences, notably:

- "**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[2] while the former is used for combinational blocks as well as to connect gates and modules. In SystemVerilog, life is made easier as these two are replaced by the single "**logic**" type. However, and especially for neophytes, this "stronger" keyword can really be a fake friend. Indeed, using "**reg**" and "**wire**"

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

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). These are a lot of new keywords and notions that we won’t use before next week. 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.

These are a small part of the differences between Verilog and SystemVerilog, but the most important to know for beginners. You will find more on this here for example. Please also note that, in this course we use SystemVerilog in a very Verilog-oriented manner, without using much of all the testbench and verification features that it contains[4]. Finally, in addition to your book, you will find many good online resources to get familiar with SystemVerilog, such as this tutorial and examples website.

Basic operators

You may be thinking that if you need to instantiate all gates individually it could take a while to get anything done! That’s why Verilog has **operators**, which are like shortcuts to standard modules that the tools automatically replace when run. There are quite a few operators around:

Operator Type	Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+	Unary plus
	-	Unary minus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	inequality
Reduction	~	Bitwise negation
	~&	nand
	|	or

[3] For a good introduction, see the Harris & Harris (ARM ed.) book, HDL example 4.17, page 194.
[4] More on this if you choose to follow LELEC 2570 next year.

	~|	nor
	^	xor
	^~	xnor
	~^	xnor
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?	conditional

Pay attention that in SystemVerilog (compared to Verilog), you cannot apply directy the combination for nand, nor and xnor, you have to use parenthesis: e.g. ~(a&b) for nand, ~(a|b) for nor, etc.

On top of that, you also get **the assignment**, which sets a wire equal to a number or another wire:

```
wire [3:0] a, b;
assign a = b;
```

Number formats

Obviously, numbers are important in Verilog, and as such they are defined in a special way. When specifying a number, you must specify their **width** (the number of bits that are used to represent that number), and their **radix** (what base the number is in).

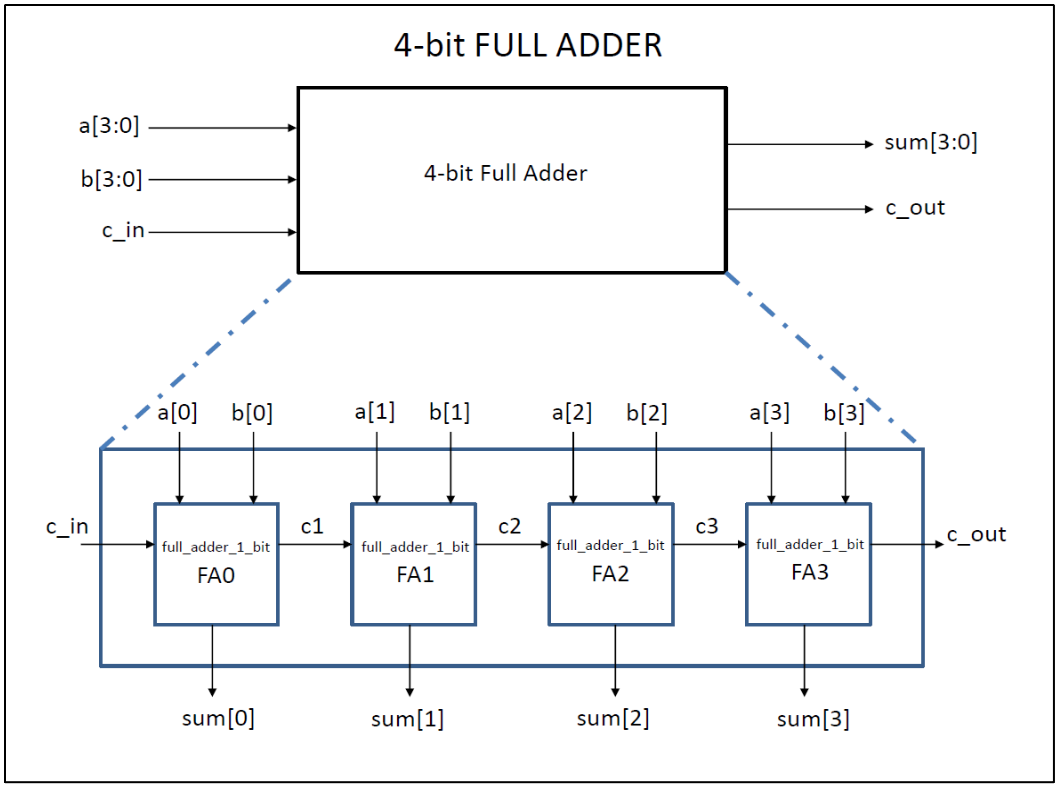
In Verilog, you must first specify the width, then a separator symbol `'`, followed by the radix, then the number itself, in the form `<width>'<radix><value>`. The radix can be **d**ecimal, **b**inary, **o**ctal, **h**exadecimal and others. Any underscore '_' in the number is ignored, and can be used for cosmetic purposes.

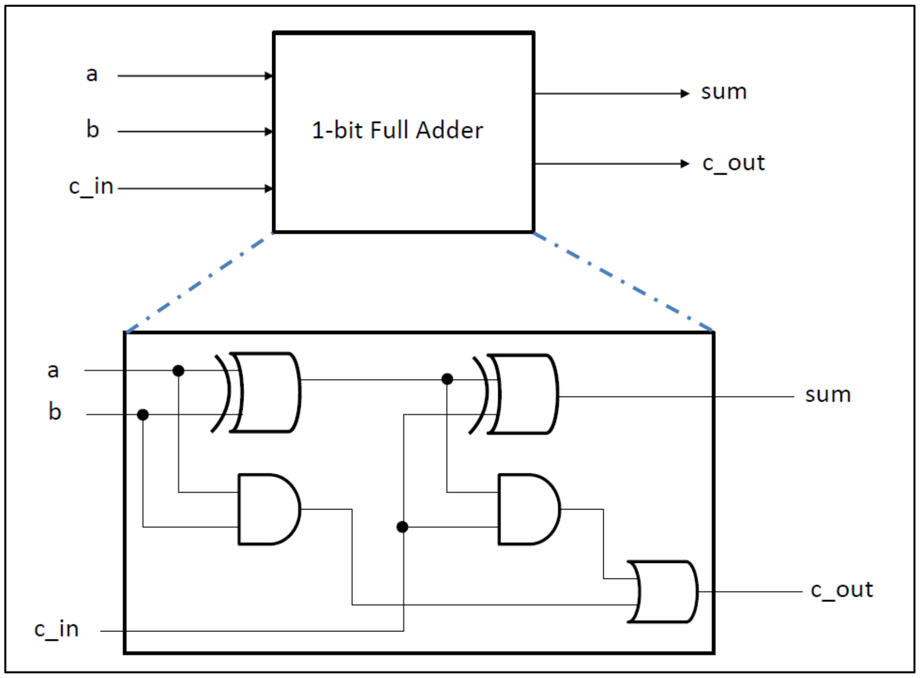
So, 4'b1001 represents the decimal number 9 and is stored on 4 bits, 10'd1023 represents a 10-bit decimal number and 32'hDEAD_BEEF is a fancy hexadecimal number.

Let's finally point out that in this course and for the tools we use (ModelSim, Quartus, etc.), the signing system used is "**two's complement**". For more information on this, see the Harris & Harris (ARM ed.) book, section 1.4.6.

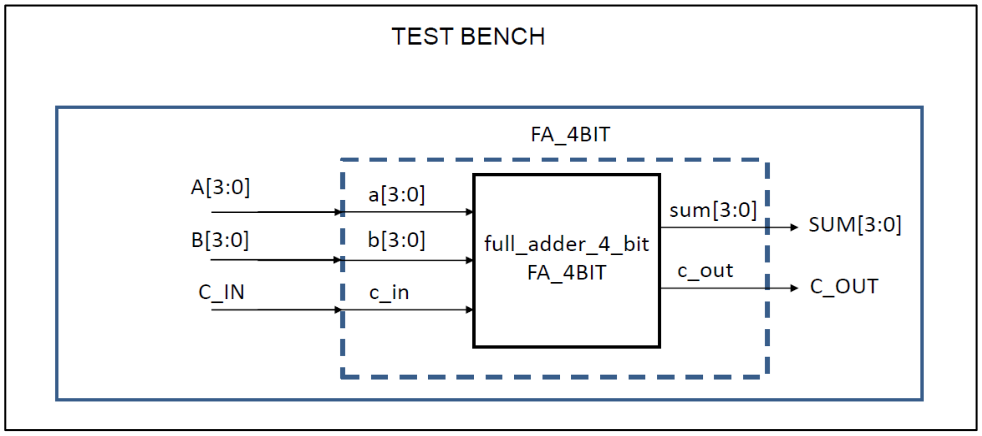
Simulation: the 4-bit full adder example

When developing something, it is nice to be able to debug it. To do so, we need a Verilog simulator. Here we will use ModelSim to simulate a 4-bit full adder, built from four 1-bit adders. Have a first look at the following block schemes and try to understand how it works (for equations and truth table see the Harris and Harris, page 240, fig. 5.3).





On top of that, we define a "testbench" file, which is not part of the circuit but will allow us to test it appropriately. It contains special instructions (e.g.: "initial") that are non-synthesizable but are understood by the simulator. More details on this during the lab session.



The "Three-Y's" rule

To end this first part, let's see how this digital circuit example follows the "Three-Y's" rule introduced by the book[5] that you should always have in mind to manage complexity.

Two of these (fundamental) concepts are indeed illustrated by the 4-bit full-adder: **modularity** in the sense that you can describe a 1-bit full-adder one time and then instantiate it several times (hence reuse it) and **hierarchy** because to implement the 4-bit full adder, you instantiate four 1-bit full adder in a same module of "higher level".

These two relates to concept of **abstraction**: one of the most important thing in digital circuits design is the idea of implementing a module at some level of **hierarchy** and using it as a black box at the higher level. In other words: as far as we have the guarantee that the 1-bit full adder is well implemented, we only need to know about its inputs and outputs to use it at a higher level of hierarchy, not how it is implemented. This doesn't mean that it doesn't matter but sometimes you will have to use IP cores (Intellectual Properties) that you cannot reverse engineer (because it is encrypted or because it would take too much time) and will have no choice but to use it as a black box. You will be confronted to that already later in this course (e.g. with the use of Quartus megafunctions).

For now, you are of course always able to see how the different modules are implemented. But, what is important is that by being **disciplined**, you will quickly be able to design more complex (and hence interesting) digital circuits. In others words, you have to define **hierarchy** between the blocks and show **regularity** in the different gates or modules you use, allowing for a high degree of **modularity**.

If you design circuits following these guidelines, another good practice is to regularly validate parts of the circuits independently and then treat them as "working black boxes" that you don't have to worry about anymore; you can just instantiate them over and over. This will save you tremendous amount of debug time, so you should really pay attention to this.

[5] Harris & Harris (ARM ed.), section 1.2.3, page 6.