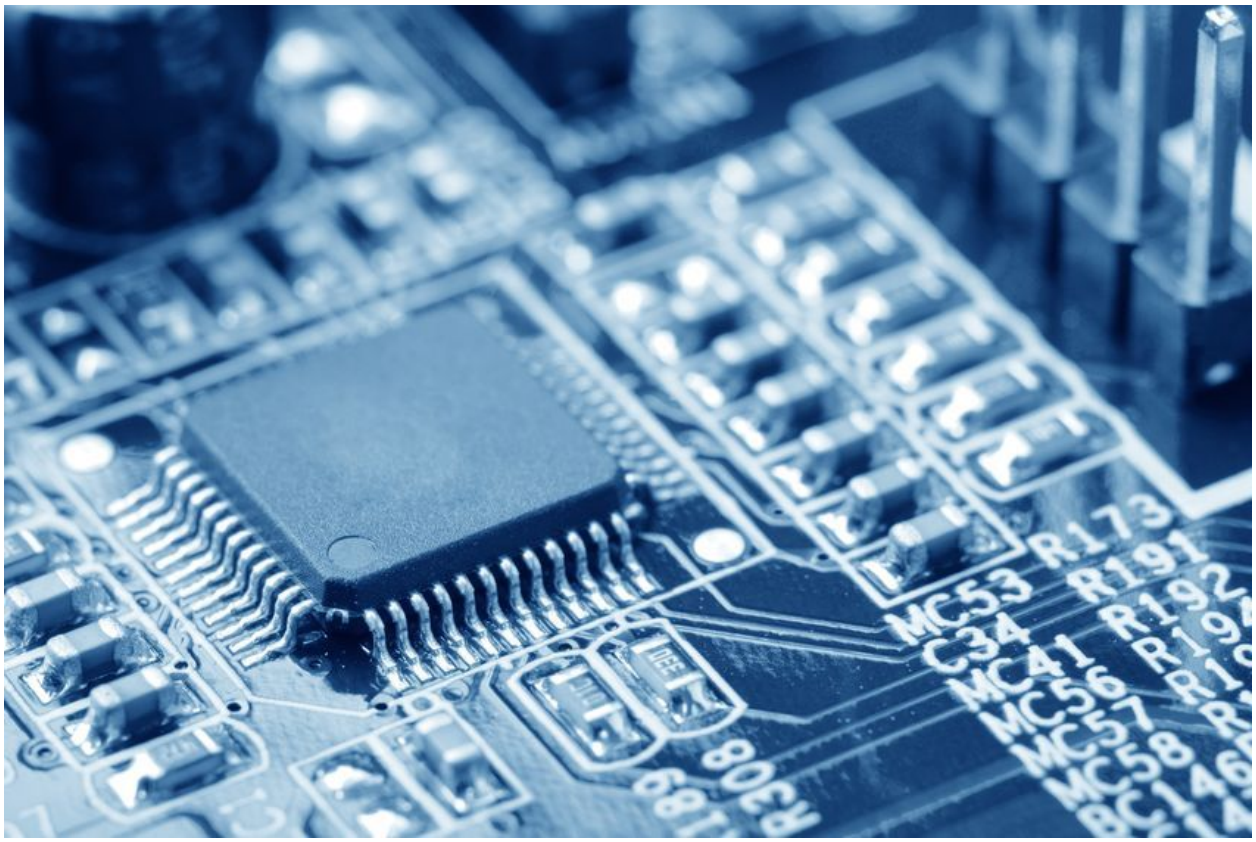


From MCU to FPGA

Last update: 07.03.2018

Ibrahim KAMAL
IKALOGIC S.A.S.
19 Rue Columbia
87000 Limoges
FRANCE

WARNING

Table of contents

Intro

Before wasting too much of your precious time I'll reply - straight forward - to those 3 questions:

- What to expect from this document?
- What not to expect?
- Who is this article addressed to?

What to expect from this document

This e-book will help you doing the transition from the world of embedded programming (normal sequential programming like C, Javascript or basic) into the world of FPGA and VHDL programming (VHDL is a language used to program FPGA). It's a journey I've been through some years ago, and although there are many articles out there, the subject is so hard to grasp sometimes that I thought one more article on the subject wouldn't harm. This document only talks about things encountered and tested with Xilinx Spartan family of FPGAs using VHDL programming language, but the principles remain the same for other product families or other manufacturers.

What not to expect

This document is absolutely not an exhaustive guide for the full VHDL syntax, nor is it a document written by an FPGA expert. I have learned on the job, but after years of practice and thousands of products in the market, I feel confident enough to help others get into that fantastic world of FPGAs. Also, this document does not handle Verilog programming language (which is another language used for FPGA). I highly encourage you to find a VHDL programming reference (out of many references available for free on the internet) to complete this e-book.

Who is this article addressed to

This articles can be a good starting point to anyone who has been working (or playing around!) with Microcontrollers, Arduino, RaspberyPi, and other embedded system and wishes to get access to a whole new level of processing power.

Ready?

Before getting into serious stuff, let me present to you FPGA's, the way I understand them. As their name implies - Field Programmable Gate Arrays - are chips with a huge amount of logic gates that can be customized to o do various functions. Imagine a circuit you build with many logic gates like flip-flops, OR, AND, XOR, Multiplexers or Decoders and many, many, MANY wires? Well, that's an FPGA, with the difference that it takes a few dozen square millimeters of

surface on your circuit board. You feel that this gives endless possibilities, right? Well, don't get too excited! FPGAs are indeed powerful beasts, but the more you try to push the limits of what they can do, the more you have to deal with very obvious limitations like the number of used gates to build your circuit, or the propagation delay between the gates. Luckily, it'll take some time until you start hitting the limits of your FPGA chip.

Document organization

I'll start by walking you through a lot of principles that you need to be familiar with in order to understand the rest of the document. I know you're eager to quickly jump to the "hello world" example that you expect to find in C-like programming languages, but please, take the time to read those couple of chapters first, they will really help you to move faster later. After the basics are laid out, I'll guide you through a very simple "led based" example, and show how step by step it can be done with a Xilinx FPGA and VHDL code.

The big picture

The following paragraph may be the most important part of this document, so don't hesitate to read it more than once until you're sure you totally grasped the idea.

In software programming, you write code that is "executed", instruction after instruction, by a microcontroller or microprocessor. I know, there is multithreading, but we all know that multithreading does not exist, It's an illusion, a processor is never really doing two things at the same time (I hope you're not just discovering this. If so, sorry to disappoint you!). So, you're used to writing code, and if syntax is good, it compiles into a binary file (executable) that is made of instructions, and that's pretty much the whole thing (in an oversimplified way). When you write code for FPGA (using the VHDL language), well, you're not writing code that will execute in the chip, you're not! You're writing code that describes a circuit. Remember that big circuit with many many wires and logic gates we were talking about? Well, when you're coding VHDL, you're writing code that ultimately describes a (usually) complex hardware circuit. As a matter of fact: the "HDL" part of VHDL stands for Hardware Description Language. Now although sometimes you'll find that VHDL lets you describe the operation of your circuit rather than describe your circuit (and we're thankful for that), it's important to always keep in mind that at some point in the "life cycle" of your code, it will all be converted to a very real electrical circuit, with very real gates and (almost) real wiring between them. Those gates and wiring have physical limitations like any other digital circuit, that need to be taken into consideration.

That process of converting your VHDL code into a circuit is called "synthesis" (and the closest thing to that process in the C programming world would be called "compiling"). We'll get into the specific of that process later.

FPGA's are versatile

An important difference you need to grasp, is that FPGAs are much more versatile than MCUs or MPUs. For instance, in MCU you've got GPIO pins, multiplexed with some other peripheral features, like timers, communication ports, PWM generators, etc. FPGAs have IO pins, and those can be configured to do almost anything. Although some "special" pins are usually dedicated for getting a clock signal inside the FPGA, there is no absolute obligation to use them. (Of course, FPGAs have some special functions pins used for JTAG programming). It's up to you to "build" the whole architecture of your system from the ground up. Decide the different modules that will be implemented (like timers, UART interface or I2C interfaces), and actually "code" those modules. This may seem like a tedious task, and frankly it is. But it is also opportunity to build the modules you need exactly like you need them. Do you need a 24 bit, ultra precise PWM signal generator? No problems! Do you need to have 20 of those modules? Well, you just need to add exactly 2 lines of code to instantiate (replicate) a block as many times as you wish.

Things can get much further: You can even build a fully functional MCU inside your FPGA. This lets you combine the flexibility of microcontrollers with the versatility of FPGAs, but it's a quite advanced feature and I wouldn't recommend this for a beginner.

One very important advantage of FPGA: You can fully test your design in simulation. Simulation can go very far, taking into account propagation delays inside actual fabrics of the FPGA, allowing you to test and validate complex designs in a comfortable test environment. Other techniques exist, similar to JTAG debug for MCUs. For xilinx it's called ChipScope if you wish to further explore this later. We won't be addressing ChipScope or equivalent in system debugging solutions in the e-book.

There are some limits to this versatility though. As stated before, high speed clock signals may need to be routed to very specific pins to help the design tools you're using (like Xilinx's ISE) to optimize the design and routing of the signals. Also, you'll notice that GPIOs are grouped in what we call "banks". A bank - or a group of pins - share the same power supply and hence operate at the same voltage. For example, you can't have 1.8V and 3.3V logic on the same bank. However, there is nothing wrong in having a 1.8V bank, and 3.3V bank on the same chip. Finally, you need to understand that although you have the right to ask for any I/O signal to be routed to any I/O pins, there is no guarantee that it will be physically possible (you'll discover why later). It's not uncommon for designers to need to change I/O placement to overcome some physical limitations of the kind.

Anatomy of a simple FPGA design

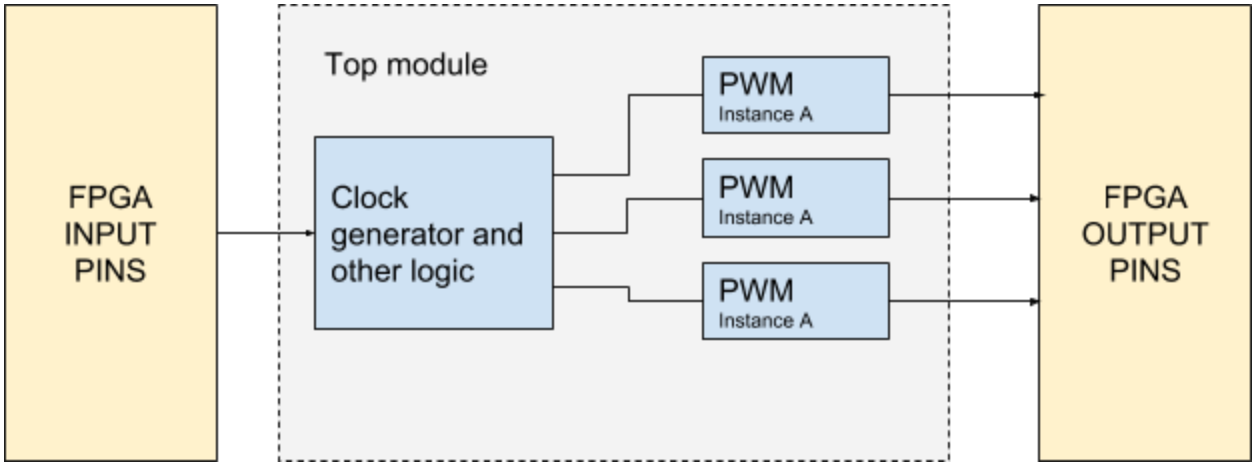
In C programming, you have two main types of files: c files and header files. (there is also a make file, a project file, or some other files of the sort). In an FPGA design, there are really, really a lot of files that get generated in dozens (if not hundreds) of folders. Thankfully, you usually don't need to do anything with those, so let's concentrate on the 4 essential components.

Modules

The most important part are modules. It's equivalent to the *.c file in your beloved C programming world; it's a plain text file that has the extension "vhd", "vhdl", or "v". Since we'll be focusing on the VHDL language in this e-book, from now on, let's stick to the *.vhd extension. A module is like a piece of circuit (or the code describing this circuit to be more precise). It has a section that describes the ports of that circuit (inputs and outputs) and then, there is the actual description of that circuit, including all gates and wiring.

A module can contain many other modules, that we call "instances", but only one module in your design can be raised to the "top" level: it's called the "top module", and the port of that module directly connects to the pins of the FPGA.

Let's imagine a project that generates 3 pwm signals. There are endless ways to code that module, of course, but an efficient approach would be to build only a single "PWM generator" module, instantiate it 3 times in a top module that "glues" everything together. Something like the graph below:



We'll get into the details of how to actually write the VHDL code for that architecture later.

Packages

Those are equivalent to headers in C files. It allows you to define data types, signals buses and pieces of code that can be reused many times in a project (or even in other projects). Although you may complete a whole project from end to end without needing packages, know that they exist. Actually there is no obligation to use them, but at some point of your evolution in the world of FPGAs, you'll feel the need to better organize your code and make it easier to maintain. When that time comes, google about VHDL package.

User Constraints file

This plain text file lets you define the placement of different I/O pins. I mean the real physical position of the pins in the package of the FPGA chip. It also lets you define the logic family to be used for each pin. Finally, it lets you add what we call "timing constraints": an indication of the speed at which some signals are clocked. Timing constraints are very important for a design, and they let the tools that route your FPGA design try to meet those constraints, and alert you in case it can't. Unless your design is so simple that it doesn't include a single clock going faster than 1M Hz, timing constraints are mandatory.

Test benches

A test bench is another plain text file that, as the name implies, allows a user to test a module. A test bench is yet another VHDL file, that describes a circuit that would connect to your FPGA in your final application. Let's take an example: you're building a board that needs to store data to an SRAM memory. In order to test the module that communicates with the SRAM, you can build a test bench that includes an emulated SRAM. You would usually grab the datasheet and try to build a test bench that mimics actual timing characteristics of the SRAM as closely as possible, the idea being to test and validate your module in the most realistic conditions.

It's worth noting that a test bench will never be actually synthesized into real gates. That means that you can use a line code that says:

```
wait for 15 ns
```

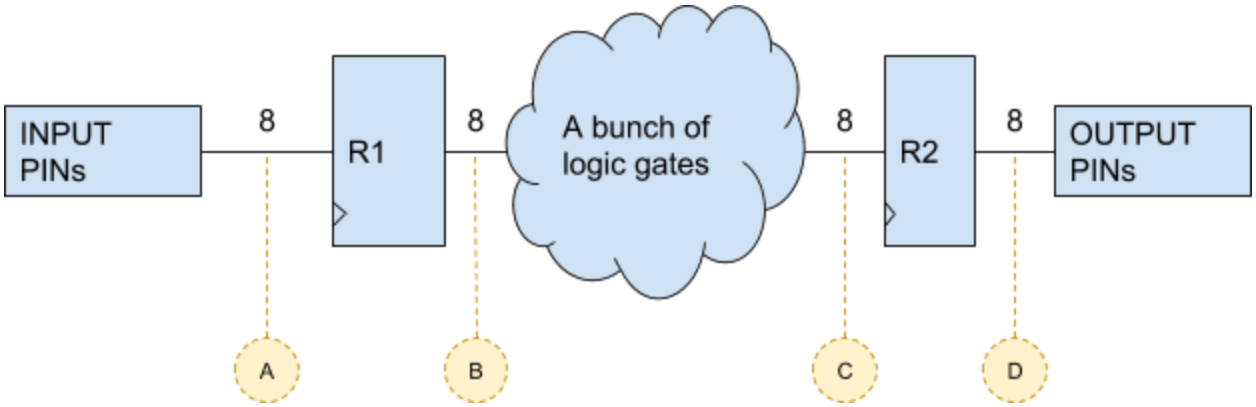
Which is something you can't do in a VHDL code that needs to describe actual logic gates. At least not that easily: to create a 15ns delays, you would need to have a clock that feeds a counter that counts elapsed time. Again, don't worry about the code for now: we'll get into actual implementation very soon!

What is RTL and how it works

RTL is one of the most important terms you'll often hear about in the world of FPGA: it stands for Register Transfer Level. RTL is the most efficient way of building high speed FPGA architectures. Actually, most of the times, it's the only practical way of getting things done in an FPGA!

Before getting into specifics, let's first talk about clocks. In an FPGA, clock is paramount. Compared to a microcontroller, where a clock is not something you worry much about, in an FPGA, the clocking is something you'll need to constantly monitor all along the way. Clock is what gets information moving in your FPGA, from the input, to logic blocks that "process" those inputs, to output pins.

The clock in an FPGA is used to clock "registers". Registers are simply a bunch of flip-flops in parallel, sharing the same clock source. Now let's look at the diagram below, this is an RTL representation that you may find in many articles out there, and it's important to be able to read it. You'll notice that there are two 8 bit registers R1 and R2, and also, you'll notice that clock signals are not drawn: This is not a mistake, but we usually assume that all registers in a design share the same clock (sharing the same clock makes things drastically simpler on the long run).



Now, let's analyze this diagram a little further, here is what it does which each clock cycle.

Clock Cycle	Register 1	Register 2
1	A is to transfered to B	
2		C is transferred to D

So, on the first clock cycle, the logic level on the pins of the FPGA is transferred from point A to point B, and nothing will happen at point B until the next clock cycle. The "bunch of logic gates" will process the data that appeared at point B to create the result at point C. What the bunch of

logic gates does is of no importance for the moment. What's important, is that the result "C" need to be ready before the next clock cycle. The time it takes for the result "C" to be constructed from the input "B" depends on the number of logic gates the signals need to go through, and the propagation delay in each gate. On the second clock cycle, the data at "C" is transferred to D and appears at the output pins (after some propagation delay).

But wait! We talked about what happens to register R2 on the second clock cycle, but what about register R1? Well, on the second clock cycle, brand new signals were "clocked-in", so the actual table showing the transfer of signals would look like this:

Clock Cycle	Register 1	Register 2
1	A_1 is to transfered to B_1	
2	A_2 is to transfered to B_2	C_1 is transferred to D_1
3	A_3 is to transfered to B_3	C_2 is transferred to D_2

If you understood what happened here, well, congratulations, you understood one of the most important concepts in modern digital systems: "pipelining". I won't focus too much on this, but it's worth explaining in a few words: You'll notice that to get data from the input all the way to the output, you needed 2 clock cycles, right? But after two clock cycles have elapsed, you'll get a new processed data byte on each clock cycle. That's a pipelined system with a latency of "2". Pipelining is very powerful, because it allows a design to "break" a huge logic circuit into smaller "bunches of logic gates", where signals can propagate fast enough. At the end, you may get a design that can run at very high clock rates. Even if you need to wait for 10 or 20 clock cycles until the very first result appears at the output, after this first latency period, you'll get a new valid output on each new clock cycle. This is what makes it possible to build FPGA designs that process information at speeds that go beyond 200 MHz. I advise you to search the internet for "FPGA pipelined RTL" for many tutorials out there when you feel you're ready to dig this further.

Notes about clocks

Before ending this chapter, I'd like to talk about clocks, again, and particularly two things: Clock edges and clock domains.

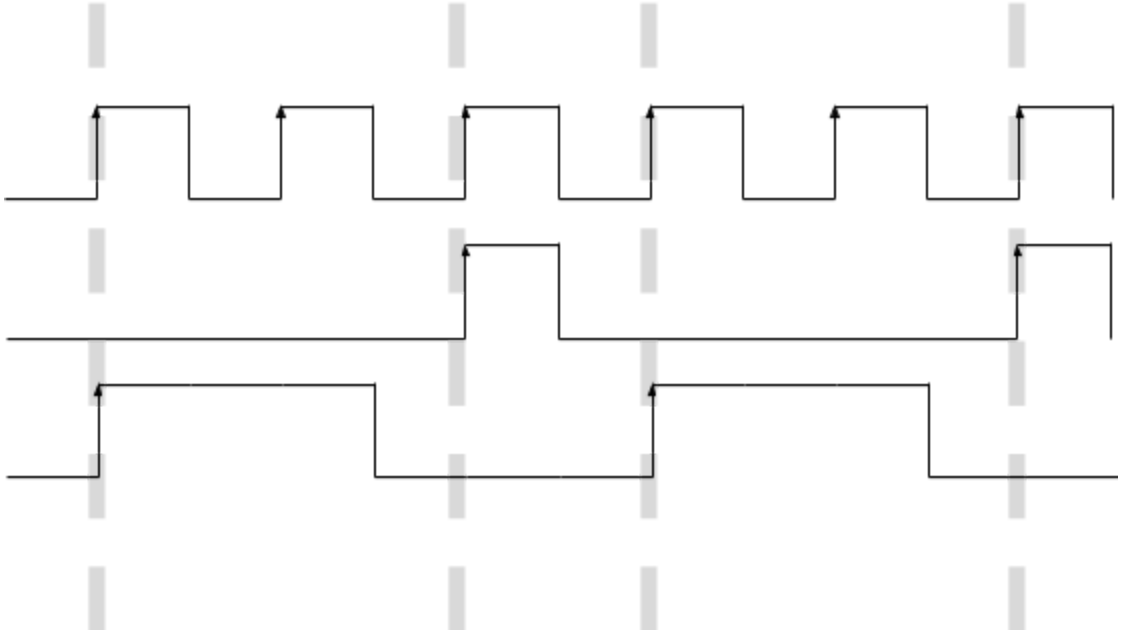
In VHDL - that translates later info logic circuits - you can't have registers that are clocked on both edges of a clock cycle. You can either synchronize your system with rising edges or falling edges, but never both. (This is not completely true, there are some exceptions to this, like DDR interface - double data rate - but it's quite sophisticated).

A piece of advice: stick to one and only clock edge polarity for your whole design. Mixing rising and falling edge is not for the faint of hearts, and is actually against almost every rule that exist around FPGA designs!

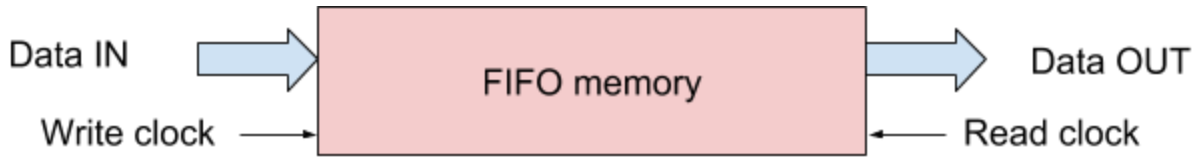
The next subject about clock is “clock domains”. If you have only 1 clock frequency for your whole system, then you have 1 clock domain. If you have two clocks that are not in sync, then you have two clock domains. I won’t say that it’s bad to have different clock domains, because this is common practice, but you need to know that extra care should be taken when mixing two different clocks in the same design. Wanna know what may go wrong with different clocks mixing up? Search the internet for “clock metastability”. There are plenty of nice articles about it, but for now, you just need to remember that it’s bad and you want to avoid it!

Two methods exist to avoid problems that may arise from mixing clocks. The first one is ensuring all clocks are synchronized, that is, ensure that the active edge of the clock (rising or falling) happen at the same time.

Below is an example of 3 clocks that don’t have the same frequency, but that have their active edges perfectly aligned. In that situation, no metastability problems may occur.



Now, if you can’t guarantee edge alignment, which is often the case, the magic trick is to use FIFO elements. A FIFO is a memory architecture whose role is often to glue two clock domains.



FIFO memory is a very limited and precious resource in an FPGA. In case of xilinx devices and development tools, a wizard is used to generate a FIFO module that can be instantiated in your design. Actually it's called an IP Core, but from a usage point of view, it's just like a VHDL module, except that you can't see the source - you just know that it works and does the job.

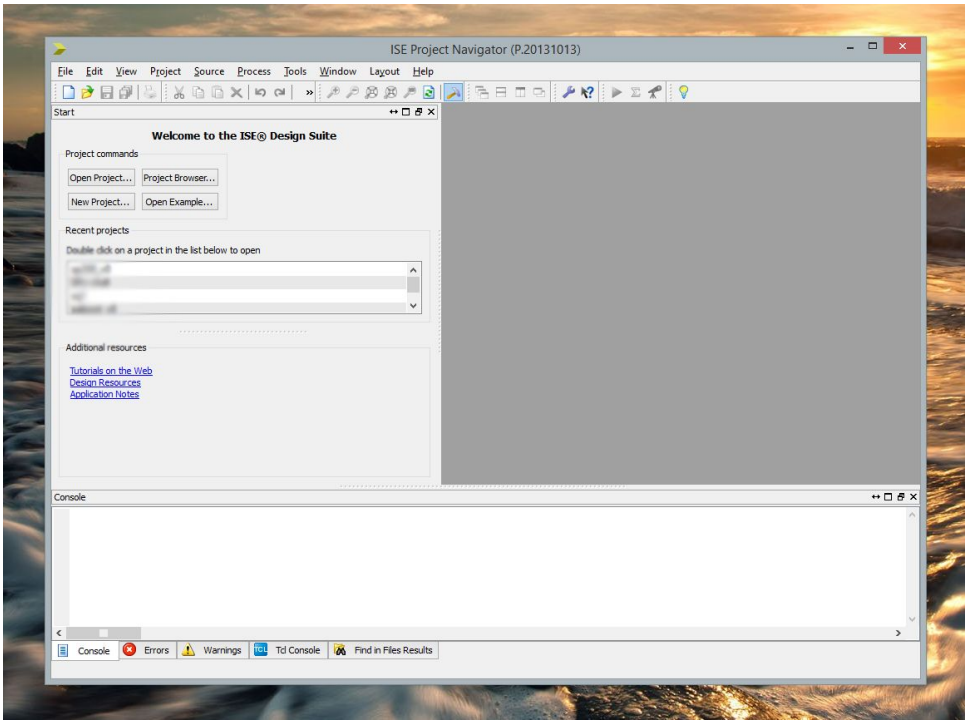
Setting up xilinx tools

We'll be focusing on Spartan FPGA devices, hence we'll focus on the tools that are compatible with this devices family. It's not the most recent family of FPGAs from Xilinx, but it's a good starting point because it's highly documented (both on official and unofficial channels), and by far one of the most famous FPGA families out there. It's also one of the less expansive.

I am not gonna get into the step by step installation process, but I'll just add a checklist to guide you in your setup:

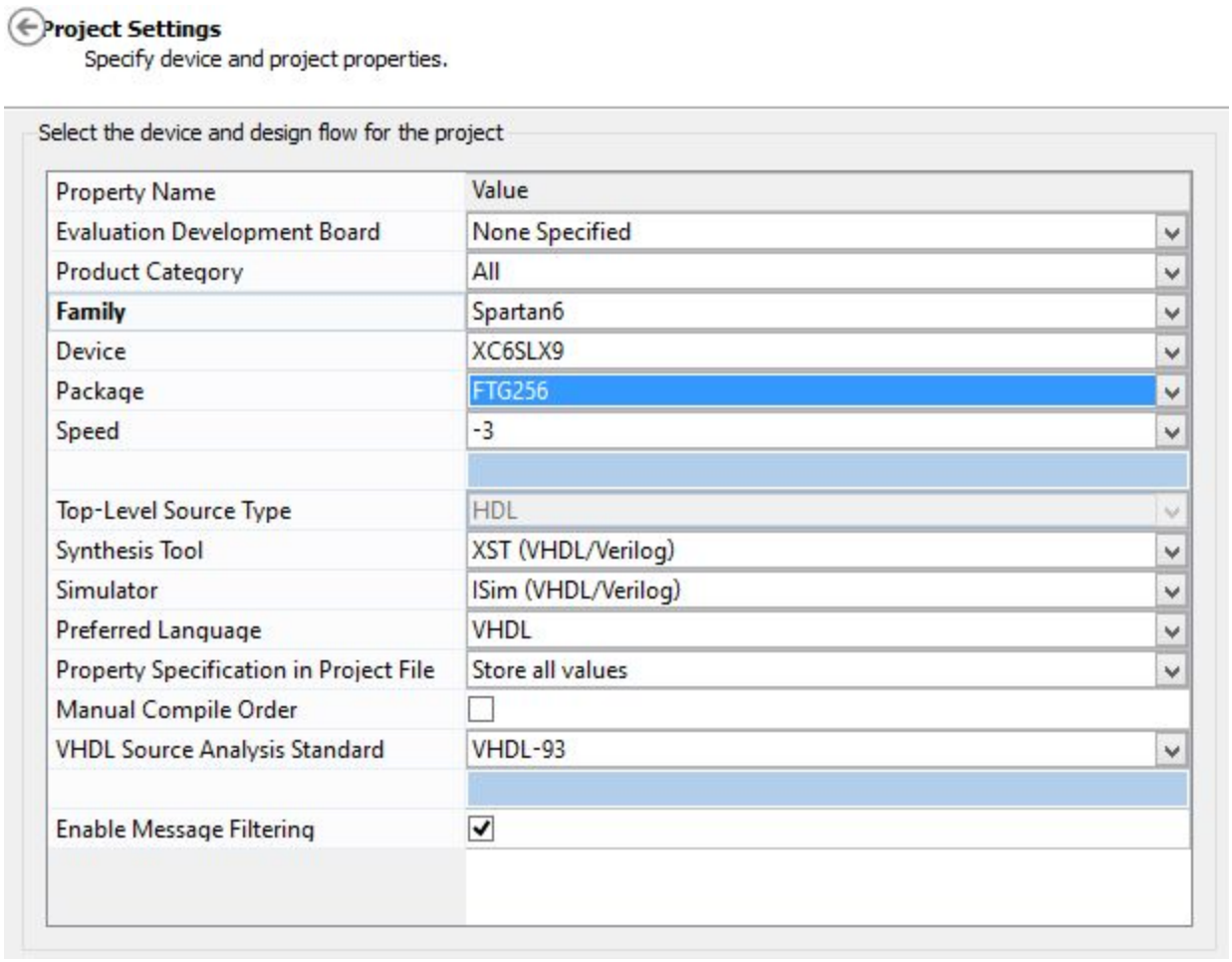
- ❏ Connect to Xilinx website and create an account
- ❏ Download the latest version of ISE software (should be V14.7). Careful, we're talking about several gigabytes.
- ❏ Setup ISE on your computer. If you can, avoid Windows 8, it was never really supported by Xilinx. (I was able to run ISE on windows 8, but the process was tedious).
- ❏ In the setup process, you'll be asked for a license: follow the steps to create a WebPack license, it's a very functional license that'll allow to get quite far until you start hitting its limits.
- ❏ Ensure that ISE does run on your computer and that licenses are correctly detected.

When you get a window like the one below, you're ready for the next step!


ISE Project Navigator (P.20131013)
File Edit View Project Source Process Tools Window Layout Help
Start
Welcome to the ISE® Design Suite
Project commands
Open Project...
Project Browser...
New Project...
Open Example...
Recent projects
Double click on a project in the list below to open
Additional resources
Tutorials on the Web
Design Resources
Application Notes
Console
Console
Errors
Warnings
Tcl Console
Find in Files Results

Creating a new project

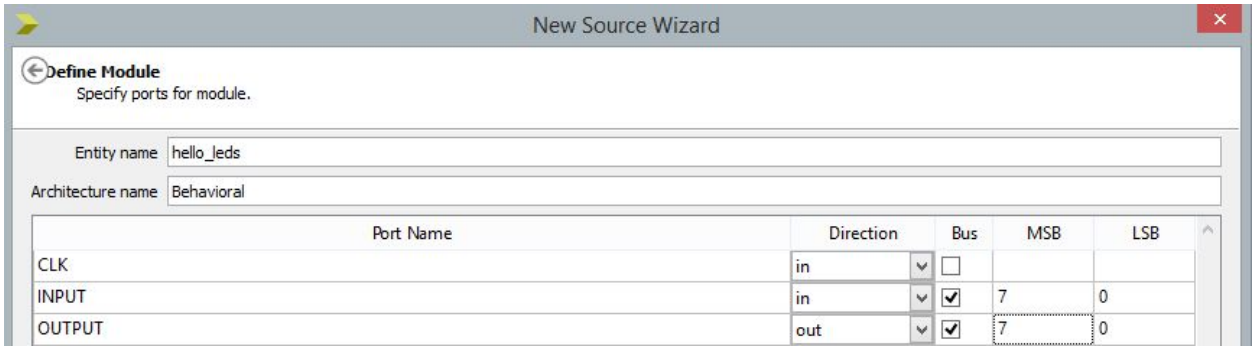
Ok, now let's get into more practical stuff. Just with like any IDE, ISE works with "projects". Go ahead and create a new project (by clicking on File > New project). After choosing a name and a location for your project, you'll be asked tougher questions, like the exact product family, device name, package, speed, etc. Honestly, it's not very critical at this stage, if you select everything just like the picture below, you'll be fine, but if you already know which chip you'll be using at the end, go ahead and select it.



I'll let you wander around through the interface and add a new VHDL module that we'll call "hello_leds.vhd". This module will have one clock input, 8 data inputs and 8 data outputs. We're gonna build a (useless) module that takes an 8-bit number, adds 50 to is, inverts the results, subtracts 20, then inverts the result again, then finally adds 10 to output the result as an 8-bit binary code on some LEDs. So, to sum up, the output would be equal to:

$$OUT = \left[\left((IN + 50)'\right) - 20\right]' + 10$$

That example is not more or less useful than the classical “Hello World” example you find in computer programming, the only aim here is to build a “playground” to test different approaches.



Your new VHDL module wizard should look like the picture above. Next step, a *.vhd file will be created. This is your module. This will automatically be your “top” module since there is only one module in your design. You can edit the code with any text file editor. Notepad++ does VHDL syntax highlighting (but there are many other solutions).

So you should end up with a file that looks like this:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity hello_leds is
    Port ( CLK : in  STD_LOGIC;
           INPUT : in  STD_LOGIC_VECTOR (7 downto 0);
           OUTPUT : out  STD_LOGIC_VECTOR (7 downto 0));
end hello_leds;

architecture Behavioral of hello_leds is

begin


end Behavioral;
```

As stated before, this is not an exhaustive VHDL tutorial, but rather a quick and minimalistic migration guide, so, here is a quick heads up:

- You'll notice that commented lines start with "--"
- The first block of code includes IEEE library and specifies which package to use from this library.
- Then comes a part that describes the in/out ports of your design. The word "STD_LOGIC_VECTOR" means a bus of bits. The part "(7 downto 0)" means it's an 8 bits bus.

If you didn't notice, that's still a rather inert piece of code. Let's write some more lines to perform the actual transfer function of that module, which is the following:

$$OUT = \left[(IN + 50)' - 20\right]' + 10$$

Here is the new code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity hello_leds is
    Port ( CLK : in  STD_LOGIC;
           INPUT : in  STD_LOGIC_VECTOR (7 downto 0);
           OUTPUT : out  STD_LOGIC_VECTOR (7 downto 0));
end hello_leds;

architecture Behavioral of hello_leds is

begin

     my_process : process(CLK)

     begin
          if (CLK'event and CLK = '1') then
               OUTPUT <= std_logic_vector(NOT(NOT(unsigned(INPUT) +
                                          50) - 20) + 10);
          end if;
     end process;
end Behavioral;
```

Before going any further, please note that this is the worst ever implementation of the required function (although it is theoretically correct). Nonetheless, the specific implementation will be

helpful to describe what you should *not* do, and why this design is flawed. But, for now, let's try to understand the lines that were added (and that are in bold above).

The first added line:

```
use IEEE.NUMERIC_STD.ALL;
```

Is just another library we need to work with signed or unsigned numbers. The second block of code is a process, a piece of code (or a part of a circuit) that is clocked by the signal "CLK". In other words all that is described between...

```
my_process : process(CLK)
```

...and

```
end process;
```

...will only "happen" when the signal "CLK" changes. Now if you look at what's inside that block, you'll find this line:

```
if (CLK'event and CLK = '1') then
```

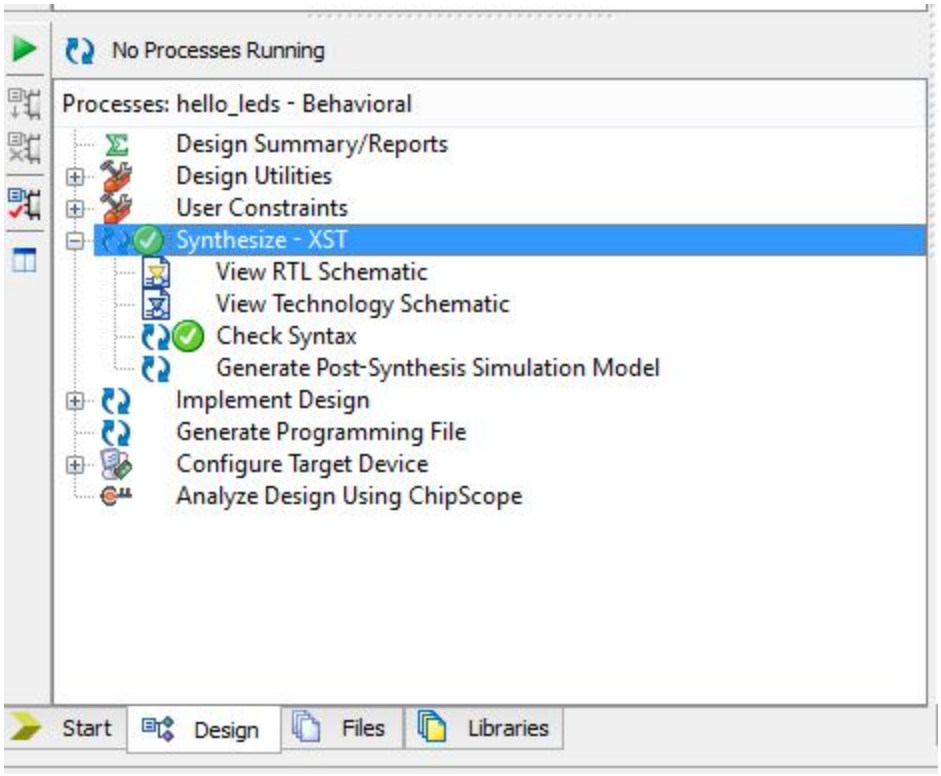
Which is an "if" block, just like the ones you regularly use in C programming. The condition here can be translated to: "If clock changed, and if after that change its value is 1". This could also very simply translate to "If a rising edge was detected on CLK". So, to recap, this line of code (which actually performs the transfer function of our module):

```
OUTPUT <= std_logic_vector(NOT(NOT(unsigned(INPUT) +
                           50) - 20) + 10);
```

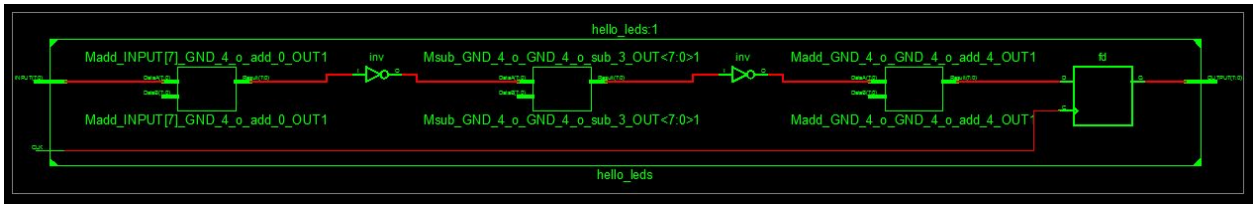
will only happen if there is a rising edge on CLK signal. I am using the word "happen" instead of "execute" because I don't want you to forget that at the end, we're describing a logic circuit, and this code will never actually be "executed" in the FPGA.

There are two new functions used in that line of code. First, **std_logic_vector()** can be used as function, to convert another signal type to **std_logic_vector**. Then, the function "**unsigned()**" is used to convert a **std_logic_vector** to the "**unsigned**" type. It's quite like type cast in C programing: We can't perform additions and subtractions to a std_logic_vector, so we need to convert it to unsigned, then, we need to convert the result back to std_logic_vector, which is the same type as "OUTPUT".

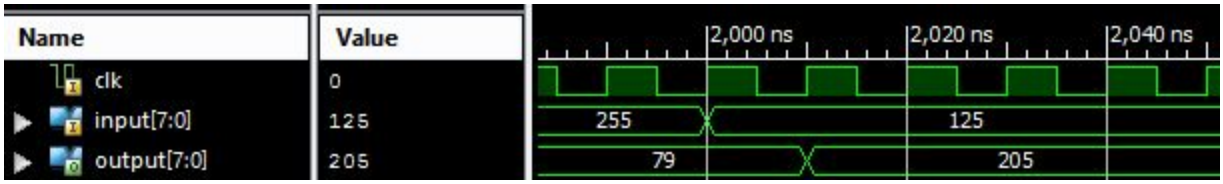
You may now go ahead and synthesize this circuit by double clicking on the synthesize line as in the following image:



After that, you can double click on view RTL schematic. I encourage you to wander around the RTL diagram and explore how your design was actually converted in a logic circuit. You should see something like this:



You can even go to the simulation view, and double click on the hello_leds module to run a simulation. I'll let you discover iSim tool on your own (you can right click on signals to apply test signals quickly). You should get a result similar to this:



This looks pretty good, you would say. And indeed, it looks good, the input is correctly

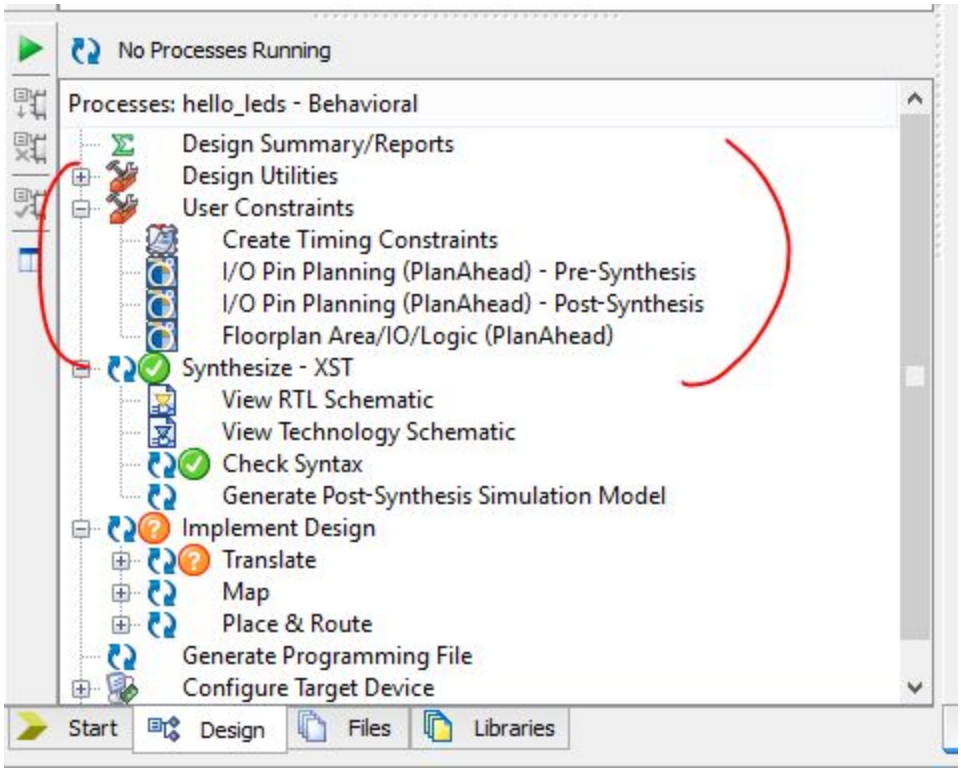
converted to the required output. But we're not done yet. We're not even close to it. You'll discover why very shortly. But for now, let's assume that everything is fine, you're up for a big surprise!

Let's move one step further along our design and add some constraints, that is, timing constraints to specify the frequency of the input clock, and location constraints to fix the actual locations of the pins on the FPGA chip.

Go ahead and switch back to the "implementation view" if you're still in "Simulation view". Now let's add some timing constraints. There are many ways to do that, and I encourage you to explore different possibilities. A very effective way is to manually add an "Implementation constraints file" to the workspace and call it "hello_leds.ucf" then copy paste this content in it:

```
NET "CLK" TNM_NET = "CLK";
TIMESPEC TS_CLK = PERIOD "CLK" 10 ns HIGH 50 %;
INST "INPUT[0]" TNM = "IN_time_group";
INST "INPUT[1]" TNM = "IN_time_group";
INST "INPUT[2]" TNM = "IN_time_group";
INST "INPUT[3]" TNM = "IN_time_group";
INST "INPUT[4]" TNM = "IN_time_group";
INST "INPUT[5]" TNM = "IN_time_group";
INST "INPUT[6]" TNM = "IN_time_group";
INST "INPUT[7]" TNM = "IN_time_group";
TIMEGRP "IN_time_group" OFFSET = IN 10 ns VALID 10 ns BEFORE "CLK" RISING;
INST "OUTPUT[0]" TNM = "OUT_time_group";
INST "OUTPUT[1]" TNM = "OUT_time_group";
INST "OUTPUT[2]" TNM = "OUT_time_group";
INST "OUTPUT[3]" TNM = "OUT_time_group";
INST "OUTPUT[4]" TNM = "OUT_time_group";
INST "OUTPUT[5]" TNM = "OUT_time_group";
INST "OUTPUT[6]" TNM = "OUT_time_group";
INST "OUTPUT[7]" TNM = "OUT_time_group";
TIMEGRP "OUT_time_group" OFFSET = OUT 10 ns AFTER "CLK";

NET "INPUT[5]" LOC = N3;
NET "INPUT[4]" LOC = R1;
NET "INPUT[3]" LOC = P2;
NET "INPUT[2]" LOC = N1;
NET "INPUT[1]" LOC = M1;
NET "INPUT[0]" LOC = R2;
NET "INPUT[7]" LOC = P1;
NET "INPUT[6]" LOC = M2;
NET "OUTPUT[1]" LOC = K1;
NET "OUTPUT[0]" LOC = K2;
NET "OUTPUT[7]" LOC = G1;
NET "OUTPUT[6]" LOC = G3;
NET "OUTPUT[5]" LOC = J1;
NET "OUTPUT[4]" LOC = H1;
NET "OUTPUT[3]" LOC = J3;
NET "OUTPUT[2]" LOC = H2;
NET "CLK" LOC = F1;
```

Don’t worry if you don’t understand all the lines of that file for now, ISE includes many tools that will generate those for you. You can find those tools under the user constraints tab as in the image below:



What’s important to note for now is that the *.ucf file you’ve just added, tells ISE where each pin should be bonded on the physical FPGA chip, and at what frequency the input clock is running. In our example, we used 100MHz (10ns period), which is something FPGA should be able to handle easily. We have also placed the clock input in a “clock dedicated site”, to achieve the best possible performance.

Now, let’s move to the next step in the design flow and click on “Implement design”. This should take a few minutes depending on the performance of your machine, and the design goes from Translate, to Map to Place and Route.

The place and route step is the most critical one. As the name implies, that’s where the design tool (ISE) tries to actually allocate actual slices of the FPGA to build the circuit you described in FPGA, then it has to route all the connections between the gates, just like you would do when routing a circuit board.

Place & Route is done? Okay, now check the project summary, you should see this:

hello_leds Project Status (03/07/2018 - 19:25:17)		
	Parser Errors:	No Errors
	Implementation State:	Placed and Routed
	• Errors:	No Errors
	• Warnings:	2 Warnings (2 new, 0 filtered)
	• Routing Results:	All Signals Completely Routed
	• Timing Constraints:	X 1 Failing Constraint
	• Final Timing Score:	7250 (Timing Report)

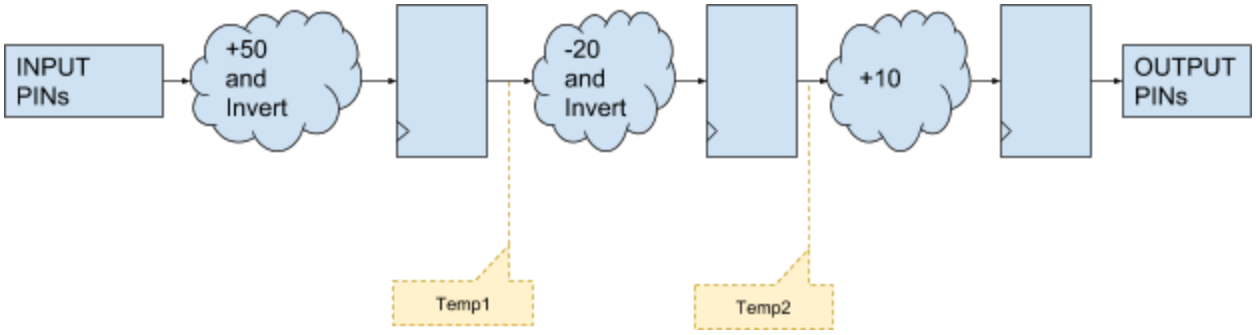
There is 1 failing constraint, one that ISE couldn't meet. You can click on the link to see exactly what's the root cause, but I can tell you without a doubt that it's caused by the poorly written VHDL code. No offense, I dragged you into this, to show exactly how you should not write VHDL code. Now, let's see how to solve this constraint problem.

A solution would be to reduce the clock frequency until we meet all constraints. But that's barely a solution. It's a last resort when everything else fails. A real solution would be to "pipe line" the design. Instead of performing the operation in 1 clock cycle, perform it in 2 or 3 clock cycles.

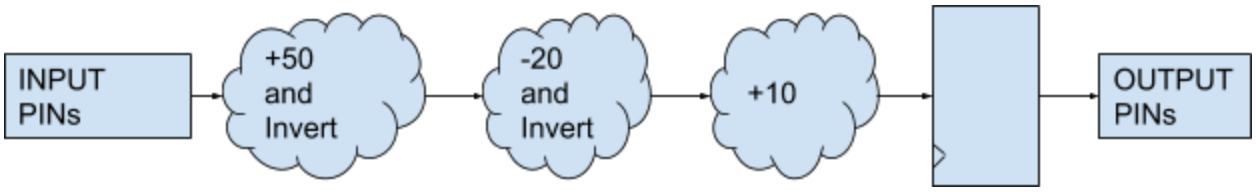
Let's see how this could be coded in VHDL:

```
entity hello_leds is
    Port ( CLK : in  STD_LOGIC;
           INPUT : in  STD_LOGIC_VECTOR (7 downto 0);
           OUTPUT : out  STD_LOGIC_VECTOR (7 downto 0));
end hello_leds;

architecture Behavioral of hello_leds is
signal temp1 : unsigned(7 downto 0);
signal temp2 : unsigned(7 downto 0);
begin

     my_process : process(CLK)

     begin

          if (CLK'event and CLK = '1') then
               temp1 <= NOT(unsigned(INPUT) + 50);
               temp2 <= NOT(temp1 - 20);
               OUTPUT <= std_logic_vector(temp2 + 10);
          end if;

     end process;

end Behavioral;
```

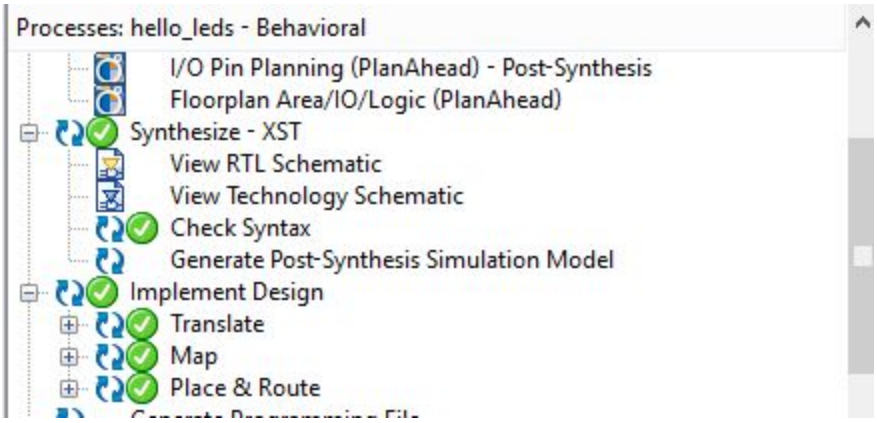
The lines that changed since the last version of the code are in **bold**. You'll notice that we have added two signals (buses) called temp1 and temp2. Also, the function carried by the module was broken into several steps. Three steps actually. The code above can be drawn as the following RTL diagram where the three steps can be clearly seen.



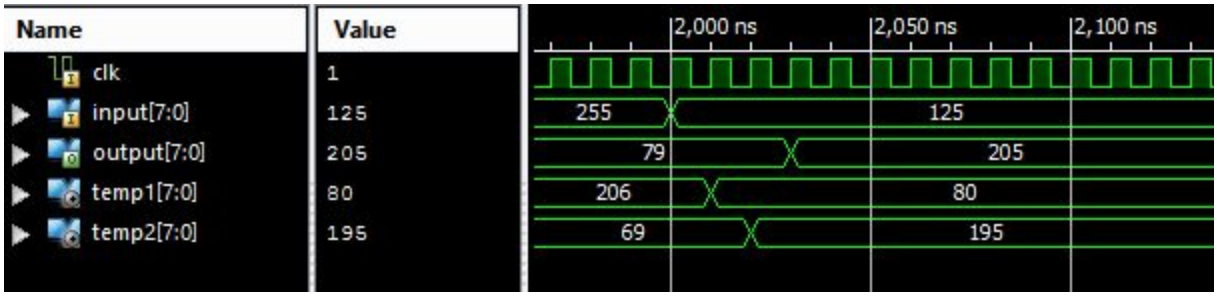
This architecture is much more efficient, as opposed to our first, flawed design:



Running the "Implement design" step again you notice a few things. First, the place and route runs successfully as seen in the image below, but also you'll notice that is runs much much faster. That's normal, because a "well designed" VHDL code can be easily implemented. On the other hand, a bad design takes a lot of effort and processing power as ISE tries millions of combinations of placement and routing that would meet the constraints that you defined.



Finally, you can run the simulation again, and notice that we get exactly the same outputs for the same inputs. You can also see that, as expected, the new code introduced some latency: It takes 3 clock cycles for the result to appear on the OUTPUT port.



Conclusion

This concludes this e-book but it's just the beginning of your journey in the world of FPGAs. Very soon, you'll have to continue searching and learning things like:

- Writing a test bench
- Understanding the difference between behavioral simulation and Post Place-And-Route simulation

I should cover those aspects in a coming ebook, and also discuss the process of drawing the schematic for your next FPGA board.

About the author

Ibrahim KAMAL, CEO at Ikalogic.
Growing up in Egypt and France gave Ibrahim a deep sense of multiculturalism. After obtaining his Bsc of Mechatronics Engineering in Egypt (2007) and master degree in France (2009), he worked as CTO of an IOT Start-up before founding Ikalogic in 2010.

Ibrahim has always been fond of self-education. In his young age, his passion to programming and electronics drove him to learn as much programming languages as he could.

By creating Ikalogic, Ibrahim wants to empower engineers around the globe with tools that unleash their creativity.

Contact the author: i.kamal@ikalogic.com