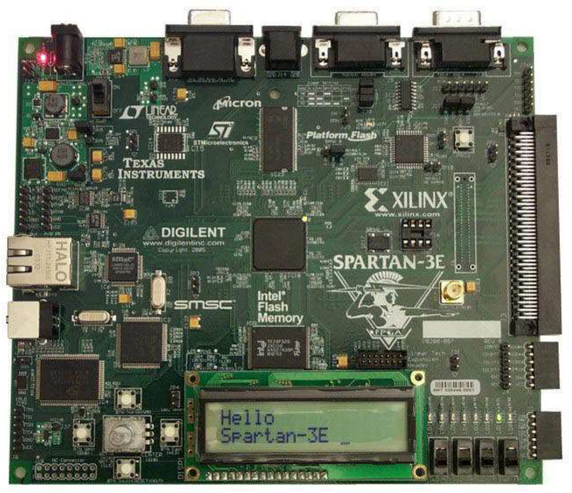
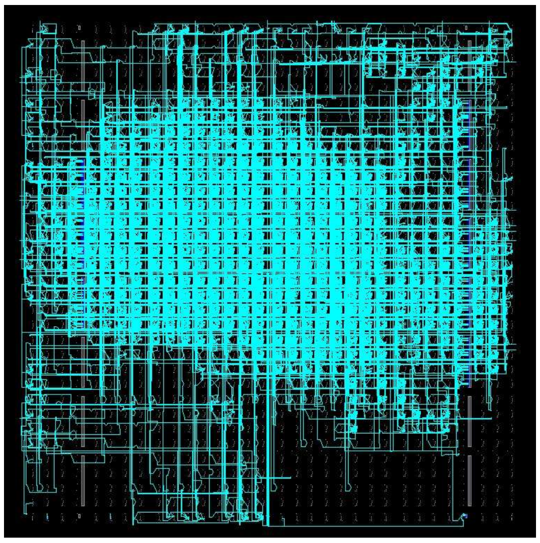
# GETTING STARTED WITH

# VHDL





A simple VHDL tutorial based on Xilinx ISE 10.1

Written by Erik Næss

07/05/2009

## 1 Introduction

Welcome to the world of VHDL and FPGAs. This tutorial will guide you through the creation of a very simple project that will make some LED's on your development board blink. The board used in this tutorial is the XilinX Spartan-3E, but if you've got a different one, that's probably not a problem. The only difference should be the location of the pins that are used.

VHDL is not at all like any other programming languages – actually it's not a programming language at all. VHDL is short for "Very High Speed Integrated Circuit Hardware Description Language". This means that when writing VHDL, one should think not so much in terms of variables, but more in terms of physical databuses and components.

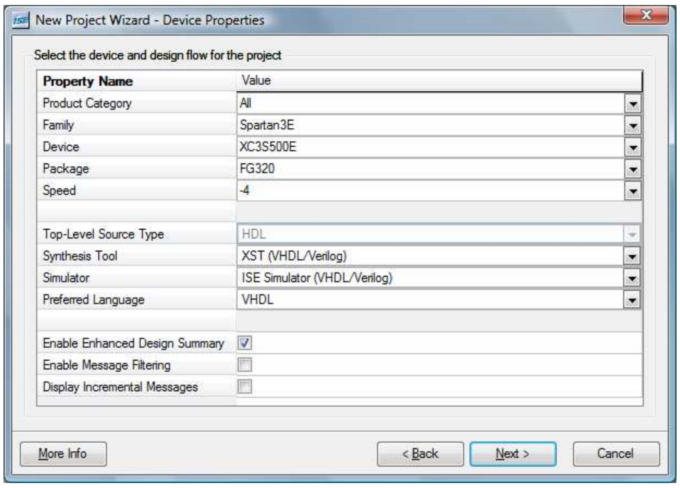
## 2 Starting a new project

Start by launching Xilinx. You should see four different windows/areas that contain nothing.

- The biggest window is the workspace. This is where you edit your files.
- The upper left window is where you chose which files to work with from the project.
- The window below the upper left window is where you choose to perform some action on the selected file, such as simulate or synthesize it. Synthesizing is FPGAs equivalent to compiling.
- The bottom window is where you get feedback from the program, such as warnings and errors.

Choose File – New Project. Call it "leds", and choose "HDL" for top-level source type. A folder will be created called "leds". This will contain all the project files. Push next.

Here we must choose the platform we are using. Fill in whatever matches your setup. If you're not sure, look in the documentation that follows the development board. Make sure that VHDL is the preferred language, and that ISE simulator is the chosen simulator. Push next.


New Project Wizard - Device Properties
Select the device and design flow for the project
More Info
< Back
Next >
Cancel

| Property Name | Value |
|---|---|
| Product Category | All |
| Family | Spartan3E |
| Device | XC3S500E |
| Package | FG320 |
| Speed | -4 |
| Top-Level Source Type | HDL |
| Synthesis Tool | XST (VHDL/Verilog) |
| Simulator | ISE Simulator (VHDL/Verilog) |
| Preferred Language | VHDL |
| Enable Enhanced Design Summary | ✓ |
| Enable Message Filtering | ☐ |
| Display Incremental Messages | ☐ |

Now you are asked if you want to add any sources. Just skip shis step and all the next ones by pressing next a couple of times and then finish.

# 3 Creating the sources

## 3.1 Creating a top-level file

By now you should have your FPGA listed in the “Sources” window. Mark it, then choose “Add new source”. Often one may want to reuse files made in different projects etc, then one may use “Add existing source”.

In the “New Source Wizard”, choose “VHDL module” and call it “main”. This will be the top-level file for our project, which will contain all the pins used on the FPGA, global variables etc. Push next and finish.

Now this file should appear as a sub-file under your FPGA in the “sources” window. Double-click it, and the contents of the file should be displayed in the main window. As you may see, some code has already been generated. Also, any line starting with “—“ is commented away, like “//” in C.

Looking at the code, you’ll see three parts. First there are declarations, just saying that we want to include some functions from the IEEE libraries. All files must start with these, just like the “#include” statements in C.

Next there is an “entity” part, and an “architecture” part. These should also be included in all sources. The entity part tells us what in- and output variables the source has, and the architecture tells us what to do with these variables. Because this is the top-level file, the in- and output variables are the actual pins on the FPGA. Because we want to control the LEDS, we will have to add these connections. Also, we must add the clock. Modify the entity part to look like this:

```
entity main is
  Port ( clk : in std_logic;
         --LEDS
         LED_0, LED_1, LED_2, LED_3, LED_4, LED_5, LED_6, LED_7 : out std_logic
  );
end main;
```

Now we have one input pin, and 8 output pins. Std_logic means that these are single pins. Some sources may for instance take an 8-bit variable as input. This would be written as:

“var_name : in std_logic_vector(7 downto 0)” or
“var_name : in std_logic_vector(0 to 7)” etc.

The difference between these two, is which bit will be indexed when writing “var_name(0)”.

Now let’s look at the architecture part. The first thing we may want to do, is to create an internal signal. We want blinking leds, but they shouldn’t be blinking at 50 MHz. Let’s create an internal clock with a frequency of 1 Hz. Also, we want to create two sub-blocks, one to perform the clock division, and one to be the LED-driver.

Modify the architecture part to look like this:

```
architecture Behavioral of main is
        signal slow_clk : std_logic;
begin
        clk_div : entity work.clk_div(Behavioral)
                port map (clk, slow_clk);
        led_drv : entity work.led_drv(Behavioral)
                port map (slow_clk, LED_0, LED_1, LED_2, LED_3, LED_4, LED_5, LED_6, LED_7);
end Behavioral;
```

This creates the sub-blocks clk_div and led_drv that should be entities of "work.clk_div" and "work.led_drv". "work" just means that these are files in our current project. But we haven't made these yet. As you can see, these are created a bit like how you use function calls in C, except the call includes both the input and the output variables that the function needs.

If you wanted to use an identical led-driver for some other leds, you could for instance add:

```
led_drv2 : entity work.led_drv(basic)
        port map (slow_clk, LED_8, LED_9, LED_10, LED_11, LED_12, LED_13, LED_14, LED_15);
```

This way, you don't have to write the driver twice.

## 3.2 Creating the clock divisor

Now it’s time to create our entities. Make sure “main” is chosen under “sources”, and doubleclick “create new source”. Make it “VHDL module”, and call it “clk_div”. Do the same for “led_drv”.

Open the clk_div file, and make the entity like this:

```
entity clk_div is
        Port ( clk : in std_logic;
                slow_clk : inout std_logic := '0'
  );
end clk_div;
```

Here, “inout” means that the variable can also be read within the block. Kind of like a feedback function. It is set to be ‘0’ as default.

Replace the architecture part with the following:

```
architecture Behavioral of clk_div is
begin
        --This generates the 1HZ clock for the leds
        slow_clk_proc : process (clk) is
        constant clkspeed : integer := 50000000;
        variable Counter : integer range 0 to clkspeed := 0;
        begin
                 if (clk'event and clk='1') then
                        Counter := Counter + 1;
                        if (Counter = clkspeed) then
                                slow_clk <= not slow_clk;
                                Counter := 0;
                        end if;
                end if;
        end process slow_clk_proc;
end Behavioral;
```

This generates a process called “slow_clk_proc” that should be “trigged” whenever something happens on the “clk” signal. It is possible to have several signals as triggers, but “clk” usually covers it all.

One can have several processes in each block, meaning that you could easily create a “slow_clk2_proc” below this one if you wanted to. The constants that are declared in this process, are availiable to this process only. If you want variables to work in several processes, they must be declared before the “begin” statement in the architecture. This is a bit dangerous though, and should mainly be done for constants.

After the begin statement in the process, we say that something is supposed to happen when the clk signal has a rising edge. It is customary to always use the rising edge. The rest of the code should be quite self explanatory. Note that we use ":=" when updating a local variable, and "<=" when updating one of the output signals declared in the entity statement.

## 3.3 Creating the LED driver

Open the led_drv file, and make the entity as follows:

```
entity led_drv is
  Port (slow_clk : in std_logic;
        LED_0, LED_1, LED_2, LED_3, LED_4, LED_5, LED_6, LED_7 : out std_logic := '0'
  );
end led_drv;
```

Make the architecture as follows:

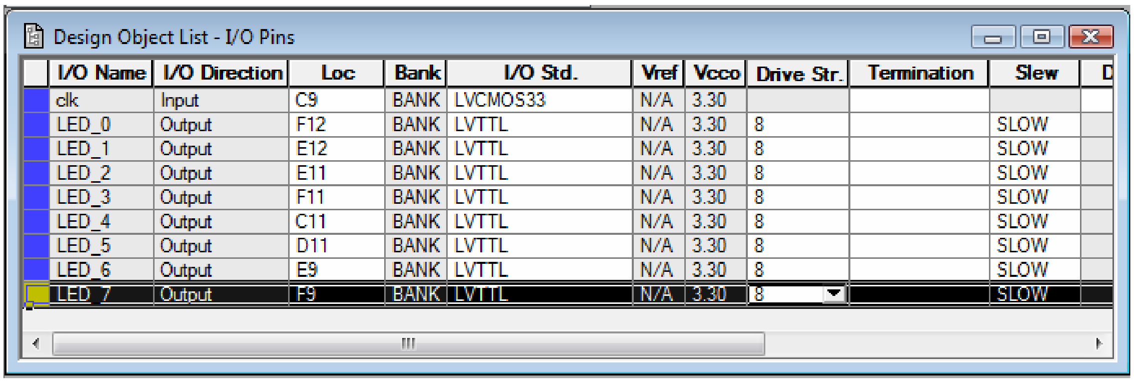
```
architecture Behavioral of led_drv is
begin
        leds_proc : process (slow_clk)
        variable counter : std_logic_vector(7 downto 0) := "00000000";
        begin
          if (slow_clk='1' and slow_clk'event) then
                        counter := counter + 1;
                        LED_0<=counter(0);
                        LED_1<=counter(1);
                        LED_2<=counter(2);
                        LED_3<=counter(3);
                        LED_4<=counter(4);
                        LED_5<=counter(5);
                        LED_6<=counter(6);
                        LED_7<=counter(7);
                end if;
        end process leds_proc;
end Behavioral;
```

Here we have created an 8-bit counter that will increase on each cycle of our slow clock. We pass each bit out to the respective LED connection. When the counter variable overflows, it will simply start on 0 again.

Mark the FPGA under "sources", expand "Design Utilities", and doubleclick "Update All Schematic Files". Choose okay if you get a pop-up. Once this is done, our two blocks should appear as sub-blocks of the main file.

## 4 Linking the physical pins and the variables

Now select “main.vhd”, and take a look in the “Processes” window. Choose “user constraints” – “Floorplan IO – Pre-Synthesis”. Choose “yes” on the popup. For all the pins, enter the correct data as specified in the development boards user manual.



Design Object List - I/O Pins

| | I/O Name | I/O Direction | Loc | Bank | I/O Std. | Vref | Vcco | Drive Str. | Termination | Slew | D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 🟦 | clk | Input | C9 | BANK | LVCMOS33 | N/A | 3.30 | | | | |
| 🟦 | LED_0 | Output | F12 | BANK | LVTTL | N/A | 3.30 | 8 | | SLOW | |
| 🟦 | LED_1 | Output | E12 | BANK | LVTTL | N/A | 3.30 | 8 | | SLOW | |
| 🟦 | LED_2 | Output | E11 | BANK | LVTTL | N/A | 3.30 | 8 | | SLOW | |
| 🟦 | LED_3 | Output | F11 | BANK | LVTTL | N/A | 3.30 | 8 | | SLOW | |
| 🟦 | LED_4 | Output | C11 | BANK | LVTTL | N/A | 3.30 | 8 | | SLOW | |
| 🟦 | LED_5 | Output | D11 | BANK | LVTTL | N/A | 3.30 | 8 | | SLOW | |
| 🟦 | LED_6 | Output | E9 | BANK | LVTTL | N/A | 3.30 | 8 | | SLOW | |
| 🟨 | LED_7 | Output | F9 | BANK | LVTTL | N/A | 3.30 | 8 | | SLOW | |

## 5 Synthesizing and transferring the code to the FPGA

Under “Configure Target Device”, rightclick “Manage Configuration Project” and select “Run”.

Everytime you have made some change to the program that you wish to try out, this has to be performed. When the program starts to grow in size, this will take longer and longer time. Ass the program gets huge, and the FPGA is almost completely filled up, this could easily take more than 20 minutes.

When done, you should get a popup asking you to choose an action. Leave it as default (Automatically connect to a cable….). Press finish.

The first time you program the FPGA you also have to specify how the “JTAG chain” is. This means what elements the programming-signal has to go through before it comes to the FPGA. Select “Operations” – “Initialize Chain”. Open “main.bit” (which contains our synthesized code), then choose bypass on any subsequent popups. Choose okay on the final popup.

Now you can right-click the green square with “main.bit” and select “program”. If everything is working, the LEDs should be counting.