# Introducing the Spartan 3E and VHDL

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| 0.1 Draft copy | 12-Apr-2012 | | MAF |

# Contents

# Chapter 1

# Introduction

Hi! I'm Mike Field (aka hamster@snap.net.nz). I want to help hackers take the plunge, purchase an FPGA development board and get their first projects up and running - but not starting at the "System on a Chip" level, but really understanding the low level detail.

I grew up in the 80s, when the 8-bit computer scene was happening, and on the back of my VIC 20 was an edge connector where you could attach *stuff*. Armed with vero-board and a soldering iron I made a few interfaces, but my designs soon got larger than my pocket money could support.

On my way to becoming a professional programmer I toyed with digital logic and what was then called microelectronics - designing with simple logic chips on a solder-less breadboard, and spent many evenings with graph paper sketching out and trying designs - some up to the scale of small CPUs.

In the late 90s and early 2000s micro-controllers with flash memory came on the scene, and I returned to playing with them as a hobby - they were cheap, relatively powerful and very accessible. But in the back of my mind was the graph paper designs of my late teenage years - I wanted to design the CPU, not just use it.

One day while reading Slashdot I stumbled onto FPGAs I was hooked.

I'm hoping that this book will inspire a person with far better ideas than me to create something really cool. Hopefully you are that person!

## 1.1 And what are FPGAs?

Field Programmable Gate Arrays are in essence a chip full of digital logic (and other bits and pieces) where the connections between the components has not't been decided upon at time of manufacture. Software tools are used to generate "configuration files" that contain the connections and initial values of all the components, which can then be downloaded to the FPGA.

The distinguishing feature from other technology is that (usually) the designs are completely *soft*. If power is removed you have a blank FPGA that can then be programmed with a different design (although most FPGAs can automatically download the design from a FLASH ROM chip, giving design persistance).

FPGAs first came to market in the late 80s. Initially they were seen as a very large PALs (Programmable Logic Arrays). During the early 90s their increasing size and flexibility allowed them to be used in networking and telecommunications applications. FPGAs allow the separation between the hardware design and the logic design they allowed vendors to quickly engineer solutions without the expense and time required to commission Application Specific Integrated Circuits (ASICs).

During the late 90s FPGAs started being used everywhere, replacing ASICs or enabling the use of advanced algorithms in consumer and industrial products - for example the system monitoring the data center where I work is based around an Xilinx FPGA, and Xilinx FPGAs are used inside some HP storage subsystems I use.

In the 2000s educational institutes began to pick up on using FPGAs in their digital design courses, and vendors were keen to supply them with development boards knowing that familiarity with the technology would help develop their markets. These boards (and their design software) are now available to the hobbyist community, and for what it used to cost of a solder-less breadboard, power supply and a few ICs you can have the equivalent of hundreds of thousands discrete logic chips to play with.

What to calculate MD5 check-sums in hardware? Sure! Want to implement an arcade game off the schematics? Why not! Design your own CPU? You can do that too - actually you can design a complete computer if you really want. With one of these development boards you have more resources than a corporation could muster 20 years ago!

# Chapter 2

# Why learn to use FPGAs?

For electronics and micro-controller buffs, the programmable logic on an FPGA is the next step closer to "real hardware", and the interfacing possibilities are endless - with the right FPGA you can talk to pretty much anything (DVI, HDMI, Fibre Channel, LVDS, PCIe, LVCMOS, LVTTL).

Unlike when working with chips and wires the designing, prototyping and debugging of designs is very fast. In the past designing and building complex designs used graph paper, discrete logic chips, breadboards and jumper wires making it a slow and tedious process. But not with FPGAs, after updating your design all you have to do is press the "implement" button.

As well as being fast use, designing and prototyping hardware is cheap - a mid-range laptop and an FPGA development board is all you need for designs that would cost tens of thousands of dollars to build with discrete logic. For hobbyists, the best part is that when you have finished with one design, you can just reuse your development board for your next project!

## 2.1 What will you learn?

When you reach the end of this book you should have achieved the following:

- A working knowledge of a subset of VHDL, enough to complete most projects
- Familiarity with the ISIM simulator, and have used it to debug an issues or two
- Used all the major components on a Spartan-3E FPGA
- You will have also used nearly all the interfaces on your chosen development board.
- Transferred data to a project on a FPGA over the USB host port, which is an often overlooked in other books!
- You may have even built a few custom interfaces that are not on the board

These skills will get you well on the way to implementing your own projects such as:

- Servo and motor drivers and sensor interfaces for robotics
- Digital Signal Processing for audio, RF or video work
- Interfacing with the any of the hundreds of low cost sensors that are now available, such as accelerometers and gyroscopes
- You could even consider building your own CPU

## 2.2 A note to software-coder types

If you are a coder, then your mind is presently wired to think about the flow of instructions running through a processor, running in a memory space. It really hurts to escape this mindset but please persist - you will need to escape it if you are to make the most of FPGAs.

Implementing efficient designs requires very different thinking. In general memory access is cheap, branches are inexpensive, parallelization is easy, serialization is hard. You need to be acutely aware of timings at design time rather than profiling code after implementation looking for hot spots.

When you get confused, the things to remember are:

- You are not writing a program.
- You are designing something akin to a (Very Very Very Very)VLIW CPU that only has a single instruction.
- It has the current state vector (stored in flip-flops and registers)
- It uses the current state and any inputs to compute the next state
- When the clock 'ticks\' it then atomically stores this new state into the state vector.

And that is it - there are no loops (well, not in the sense you think of them now), no "do this then do that", there is no "flow" through the code - it is all concurrent. There is pretty much only one unit of time - a single tick of a clock signal. It can be quite freaky at times!

The good thing is that as a result of this mind shift you will start thinking more 'super-scalar\', and the mental paradigm that you write your code in a closer match to the underlying hardware. You may find yourself changing a few small habits that will improve the quality of your code.

## 2.3 Size of projects that are possible to implement in an FPGA

A very low end FPGA (e.g. the Spartan 3E - 100 is equivalent to approximately 100,000 logic gates - or 25,000 TTL logic chips. The largest FPGA in the same product range has 16 times as many logic elements with an 'equivalent gate count\' of 1,600,000 gates.

The easiest way to visualize this is to think in terms of solder-less breadboards. A 40mm x 90mm breadboard can comfortably fit three 7400 series TTL chips and associated wiring, making the small FPGA equivalent to a 4.0m x 7.2m array of breadboards, each populated with three TTL logic chips. Large FPGA equivalent to nearly a basketball court full of breadboards!

Having this much logic allows you to implement pretty much any design you can envisage, and makes you appreciate the job that the design software does for you.

## 2.4 Why VHDL? Why not Verilog

Today there are two dominant Hardware Description Languages in use - Verlog and VHDL. VHDL is based on ADA, is strongly typed and very verbose. Verilog is more like C - loosely typed with lots of punctuation. Both languages are pretty much equally expressive - anything you can do one language you can also do in the other - and can even be mixed and matched in the same project with far more fluidity than mixing languages in software design.

For me, I find that the explicitness of VHDL makes it more explainable. I'm also from outside of the USA - where VHDL seems to be the de facto standard for research and industry. I find Verilog code a bit like Perl. It is quick to write but it feels very "fast an loose" - sometimes it is very hard to tell what the heck is going on.

But for the size of projects in this course use whatever language you like - it is more about the underlying concept than the code.

You won't need it at the moment, but if you want to learn the other 90% of the VHDL language that is not covered here, snag yourself a copy of "Free Range VHDL". It is available at a web browser near you, at http://www.freerangefactory.org/

# Chapter 3

# Prerequisite skills

Here is the skills that I think are required for somebody wanting to learn to program FPGAs. None of these are essential, but I will assume that you have them during this book, and won't bother delving into them. If you are particularly weak in any areas then be prepared to learn!

- Programming ability in a low level language (e.g. *C* or assembler)

Do you know what a byte is, how many bits it contains and what values it can hold? What does a bit shift two bits to the left do? What happens when you treat an unsigned number as signed? Do you have any idea of the ASCII code for *'Z'*?

- Familiarity with the basic boolean operations

If you can draw the truth tables for AND, OR, NOT, NOR, NAND and XOR then you have all the skills needed - this isn't like the old days when you needed to simplify logic equations yourself, that is what computers are for. If you can draw a Karnaugh map and converted it into a logic equation then you are most pobably overqualified in this area!

- An understanding of number representations and binary math

If you can't add binary numbers without a calculator you will struggle. If you can divide or multiply in binary using a pen and paper you will be fine.

Throughout this book I only use number systems - binary and decimal. VHDL understands hexadecimal constants, but I don't often use it as you are unable to tell if x"3F" is 8 bits, 7 bits or 6 bits in size - but "0111111" is 7 bits, no questions asked.

An inate sense of the size of numbers in binary will help you avoid problems. Being able to answer questions like "How many bits do I need to count to one million?" off the top of your head will be a big advantage. If you can't do this, print out a table of the powers of two and stick it on the wall. Yes, I'm being serious - when you spend hours trying to work out why your comparison of a 10 bit counter against 1523 is always false you will kick yourself.

- An understanding of circuit schematics used in digital designs helps

You will be gettin really close to the hardware, so although ir is not essential for using FPGAs the ability to look at the board's schematics and seeing how the hardware works is very helpful. As all the development board schematics are available it comes in handy when tracking down what external connections are used for on the FPGA.

- Microcontroller development experience

A little bit of microcontroller development experience is useful, but not essential. If you have played around in the embedded space you will have some familarity with the sorts of problems you will encounter. You will also be familar with how to debug without the help of high level debugging tools, and will be able to pick up the simulator much quicker. Yes, using an Arduino counts as microcontroller development.

## 3.1 Other resources you will need

- A modest PC is all you need.

A PC equivalent to a current entry-level laptop running either Windows XP, Windows 7 or Linux (Dual core CPU with 2GB RAM, 20GB free disk) is all you need. We are only using small FPGAs, so nothing high-end is required.

- Internet access is a must

You must have a broadband connection with an internet plan that enables you to download the multi-GB design software. It will also be helpful for downloading product documentation and seeking help.

- Money, or a friend with a FPGA development board to lend

Around US$79 + p&p will get you a modest FPGA development board. Borrowing one is even cheaper, but unless you are really good at sharing don't go halves with a friend in buying a board - they are small enough to carry around with your laptop bag, allowing you to try things out when insperation strikes or on a rainy lunchtime.

Now, with all that out of the way, it is on to the interesting stuff!

# Chapter 4

# Choosing your development board

At the moment I have six development boards, they are all different - one is little more than an FPGA on a PCB, another has DRAM, ROM and a large breadboard attached. One thing I have learnt is the more *stuff* there is on board to experiment with the more you will want to use it, and the more value you will get from it.

To keep up front cost down I have selected two of the least expensive development boards as the reference this book, the Papilio One with the LogicStart MegaWing, and the Digilent Basys2. Although they both feature a Xilinx Spartan 3E FPGA thet have very different design philosophies.

I really like the Gadget Factory boards, and Jack has been a big help supplying me with prototypes and designing the LogicStart MegaWing specifically for this book. If you are going to acquire a board and have no reason to go either way, get the Papilio One.

All of the Papilio tools open source and in GitHub, and the boards are really well engineered - I have been about to generate full HD VGA signals off of a Papilio board, where as I can't get a stable 640x480 signal from the Basys2.

But when all is said an done, both boards are great, and you won't be disappointed with either.

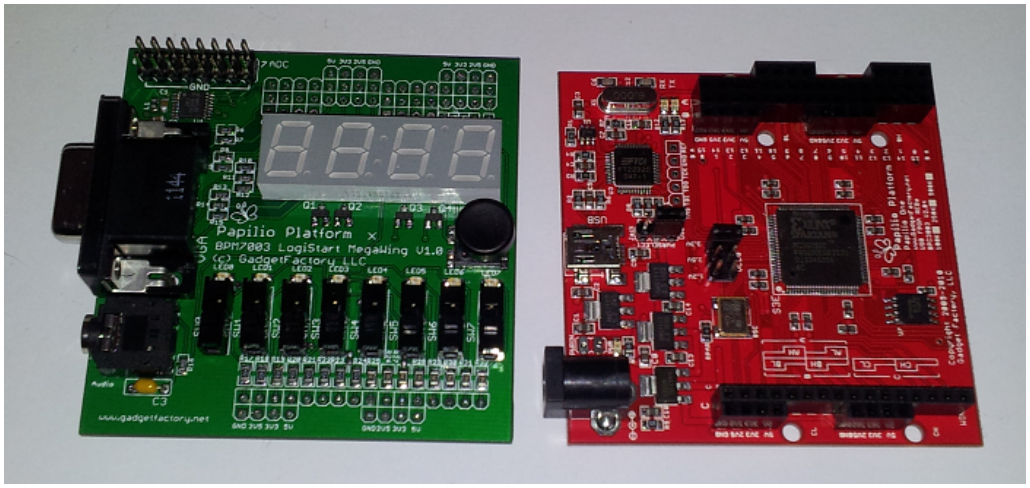
## 4.1 Why did I choose Xilinx FPGA, why not *brand X*?

There are currently two big players in the FPGA market - Xilinx and Altera. Each vendor provides their own EDA tools and although they are talking the same language they are quite different (a bit like Eclipse vs Visual Studio). I had to pick one, and Xilinx's tool set is the most approachable.

But if you are bold, you could work though this material with a different Vendor's development board, but it will be challenging at times. It will be very much like following a Visual C tutorial when running Eclipse. Consider using that Vendor's quick-start material for the first couple of projects then jump back in a few chapters on.

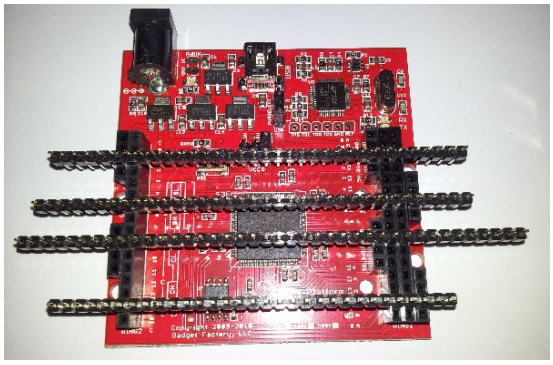
The one place where you will really struggle is with using the simulator. The configuration and setup of the simulation tools is very different between vendors, with the Altera solution being tricky to set up.

## 4.2 Papilio One + LogicStart MegaWing




Designed by Gadget Factory, the Papilio One board is squarely aimed at an somebody who has previous experience with basic electronics, who owns a soldering iron and quite possibly has a few Arduino micro-controllers kicking around. The Papilio One board holds the FPGA, a small serial ROM, a USB programming interface and the required power supplies. It provides direct access to 48 general purpose pins on the FPGA through six eight bit *wing* connectors. Originally envisaged as an Arduino/FPGA hybrid it is now used for projects such as software defined radio, emulating classic arcade games and as low cost way to experiment with FPGAs.

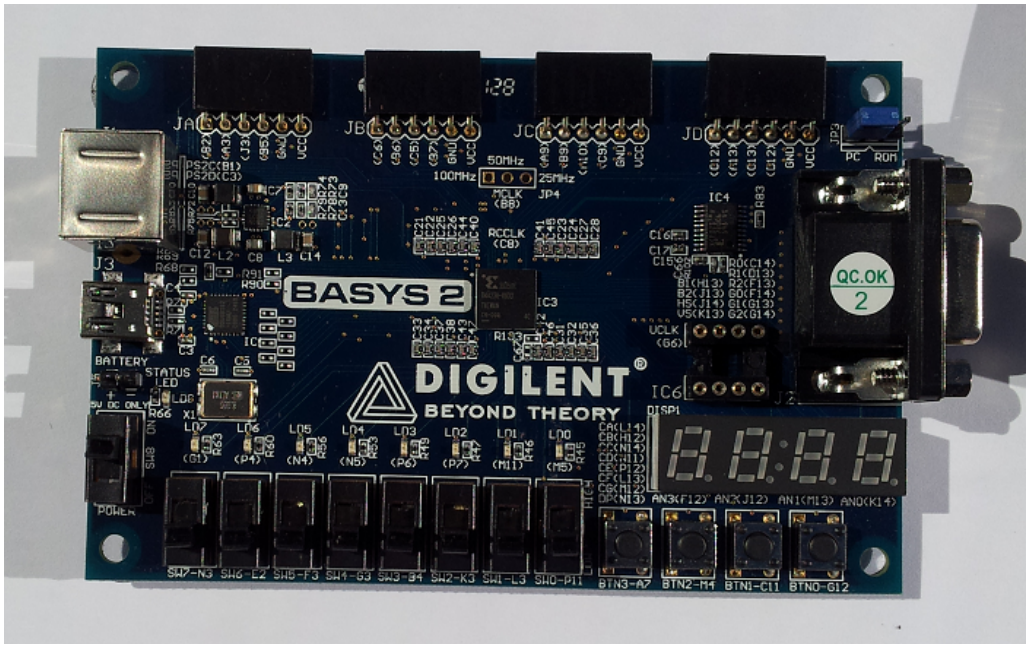
The Papilio One board ships with the headers used for attaching the wings but they are not installed. This gives added flexibility to embed the board in your own projects and most probably saves a little on cost as it simplifies packaging. When you solder on the headers getting them aligned can be a challenge. An easy way is to use some snap off pin header strips to form a jig, which will hold everything nice and square while you solder the headers in place.



The LogicStart MegaWing has been designed especially for people starting out with FPGAs, converting the Papilio into something that matches the features and accessibility of the FPGA boards developed for the education market. When mated with a Papilio One it adds few nice extras that will appeal to the Papilio One's target market such as a small joystick, 8 channels of digital-to-analogue conversion, and an 3.5mm audio socket.

Once you have outgrown the LogicStart MegaWing Gadget Factory offer a range of other Wings that can be attached to the Papilio One to customize it to your future project's requirements, and it has an active community of helpful users on the forums hosted by Gadget Factory.

## 4.3 Digilent Basys2




Digilent Inc partners with Xilinx and designs and markets the a range of FPGA development boards. The Basys2 is their entry level board, targeted at the education market with phrases like "Build digital circuits for less than the price of a textbook!" in their marketing. They offer pretty sharp Academic and US Student discounts, but there are quite a few hoops to jump through qualify.

As the Basys2 has been used as the platform for course materials and text books Digilent have far less freedom to add new features to the board - for example, it still has a PS/2 port when no current PC ships with a PS/2 keyboard. Digilent also have an equivalent of the Gadget Factory's *Wing* system called *PMOD*.

The Basys2 comes in DVD sized plastic case with foam padding, and includes a USB cable. One feature of the Basys2 that is either a help or a hindrance is that all externally available signals from the FPGA have resisters in series, aiding with ESD protection and preventing damage from abuse in the classroom environment but sometimes can cause interfacing issues.

In my view the biggest flaw in the board is that Digilent have opted not to use a crystal to generate a stable on-board clock possibly to save cost (or maybe to remain compatible with the original Basys). The jitter present in its cheaper clock prevents makes the Basys2 unsuitable for generating higher frequency signals - for example the VGA output is unusable for anything serious and most LCD monitors are unable to even sync with it! There is a socket for a second clock signal, although the required part is expensive and hard to source.

---

**Note**
The part number in the BASYS2 reference manual is wrong - order something like SGR-8002DC-PCC-ND from Digikey to provide a stable clock, should you require one.

---

As I'm in New Zealand I ordered mine from their ANZ distributor - Black Box Consulting. They normally have everything in stock, so not only is it quicker than ordering from Digilent, the international shipping direct from Digilent is really, really, really expensive. Why is it you can get a Papilio FPGA board from Seeed Studios for $49.99 incl postage, but it costs US$36.66 to ship a $99 order from Digilent? Black Box Consulting charged me a reasonable AU$10.00 for shipping.

## 4.4 A quick comparison

| | **Papilio One + LogicStart** | **Digilent Basys-250** |
|---|---|---|
| FPGA | Spartan 3E | Spartan 3E |

| | **Papilio One + LogicStart** | **Digilent Basys-250** |
|---|---|---|
| Effective Gate Count | 250,000 or 500,000 | 100,000 or 250,000 |
| Programming interface | USB | USB |
| Configuration ROM | Yes | Yes |
| VGA Connector and colour depth | Yes, 8 bit | Yes, 8 bit |
| Four digit, Seven Segment display | Yes, slightly bigger | Yes |
| Host communication interface | RS232 over USB | 8 bit parallel (EPP) |
| Maximum host transfer rate | 300kB/s | 11MB/s |
| LEDs | 8 | 8 |
| Slide switches | 8 | 8 |
| Push buttons | 1 (on joystick) | 4, in a row |
| Mini-joystick | Yes | No |
| PS/2 port | No | Yes |
| On-board clock | 32MHz, stable | 25/50/100MHz, jittery |
| Carry case | No | Yes |
| Analogue to digital converter | eight 12 bit channels | No |
| Audio output | Yes, mono (to both L+R) | No |
| Additional power connector | Barrel jack | Two pin header |
| ESD protection on all connectors | No | Yes |
| Size | Smaller, thicker | Larger |
| Open design | Yes | No |
| USB cable supplied | No | Yes |
| Add-on modules available | Yes (remove LogicStart) | Yes |
| Maximum user I/O pins | 48 (remove LogicStart) | 12 + 2 on PS/2 |
| Voltages available to add-ons | 2.5V 3.3V, 5V | 3.3V |
| Soldering required | Yes, to attach headers | No |
| Designed to work with Arduino S/W | Yes | No |
| Has geek factor? | Yes, very underground | No, used in colleges |

# Chapter 5

# Installing the EDA tools

The first step in using your FPGA is to install the tools required to implement your designs - these are collectively called "Electronic Design Automation" (EDA) tools, but you can just as easily think of them as the VHDL IDE and compiler.

## 5.1 Acquiring the EDA software tools

- The Xilinx design tools available for download from http://www.xilinx.com/support/download/index.htm. Be warned - it is a very, very big download. You want the package called "Full Installer for Windows" or "Full Installer for Linux" - one of the options is to install only the "cut down" WebPack version.
- As Xilinx ship the Windows software in a "UNIX" tar.gz file, on Windows you will need something like "7-zip" to extract the installer software. It can be obtained from http://www.7-zip.org/download.html
- During the install you will need to register with Xilinx for a license file. The software will help you acquire during the install. The install process is very good at walking you through this, so don't fret over it.
- You will need the development board specific tools from the vendor to allow you to download designs to the board - for the Papilio you will Papilio Loader from http://papilio.cc/index.php?n=Papilio.Download and for the Basys2 you will need Adept 2 from http://www.digilentinc.com.

If you don't have a FPGA development board you are still able to work your way through the modules, but it is not the same without seeing the design run in actual hardware.

## 5.2 Setting up the software

Unpack and install all the software - the downloading hurts far more than the installing! If working on Linux search the web to find any missing dependencies - I have only used the Altera tools on Linux and it was quite a challenge to get everything working correctly.

If you have Basys2 board you can play with the preloaded self-test application that is loaded into your board - the Papilio One is shipped "empty".

## 5.3 Hints for Linux users

From Andrei Barbu:

*'Digilent's GUI tool seems to segfault quite a bit, at least under Gentoo. The workaround is to use their command line tool which is nicer anyway since it can be scripted.*'

```
djtgcfg prog -d Basys2 --index 0 --file module2.bit <<< "Y"
```

# Chapter 6

# Your first project

Getting the first design to work is always the hardest part. In this chapter we will ’virtually\’ wire two switches up to two LEDs.

On finishing this chapter you will have:

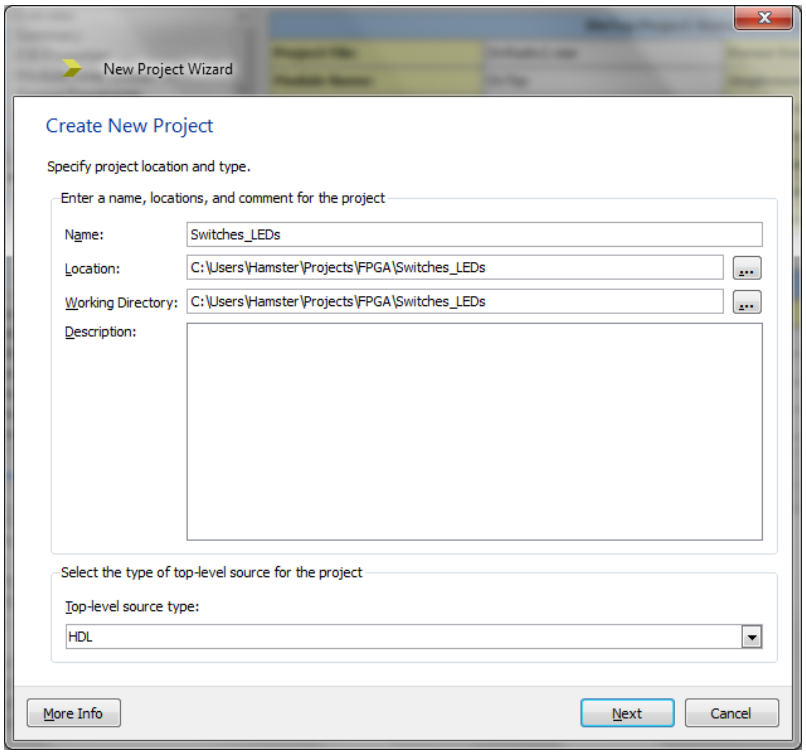
- Created a new project
- Created a new VHDL module
- Entered basic code
- Implement the design
- Set what I/O pins will be connected to which internal signals
- Implement the design again
- Used the hardware programming tool for your board
- Tested the design in hardware

Wow! That is a lot of learning for one sitting!

## 6.1 Step 1 - Create a new Project

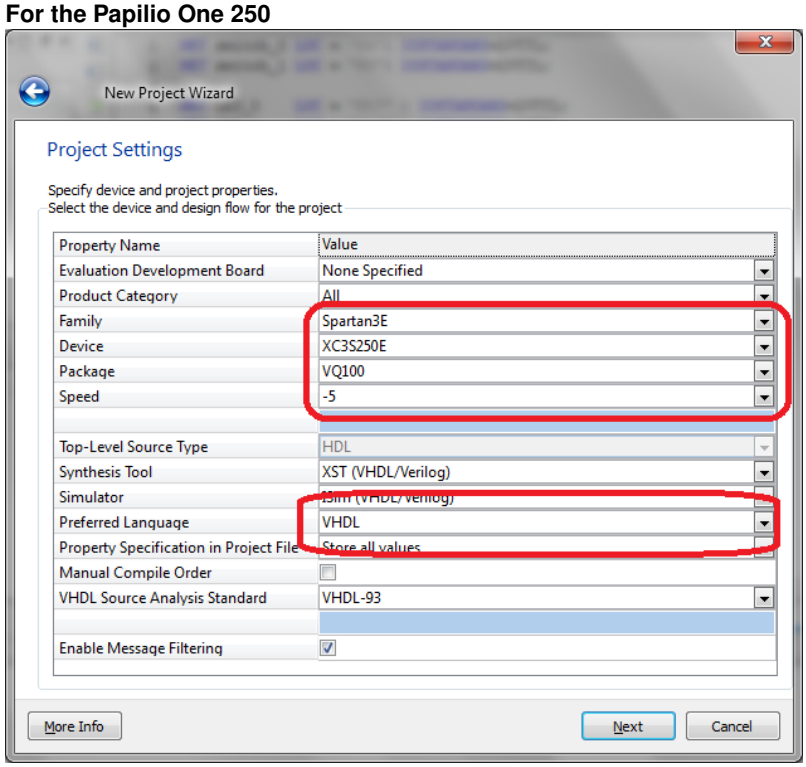
This is pretty much a "follow your nose" task, but you must get the settings for the target device must exactly match the device you are using.

- Click on "Xilinx ISE Design Suite 13.3" Studio Icon
- From the "File" menu, choose "New Project"
- Name the project "Switches_LEDs", and click on "Next".

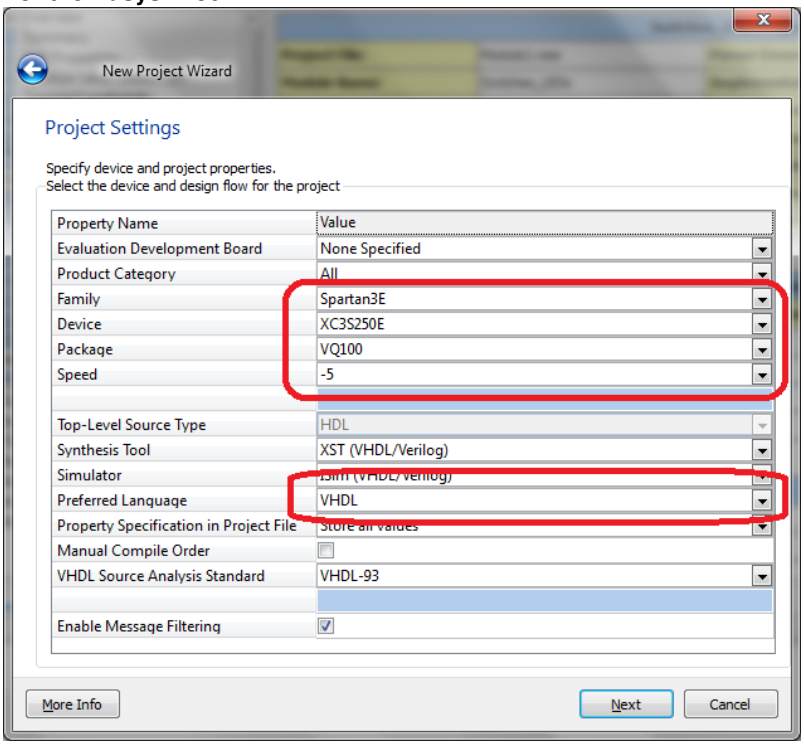



- This is the screen where you say what FPGA device you are using. Choose the following settings to tell the design tools what chip you are using (I'm using the 250,000 gate count - if you are using a different one then select xc3s100e or xc3s500e), then press the "Next" button.

**For the Papilio One 250**



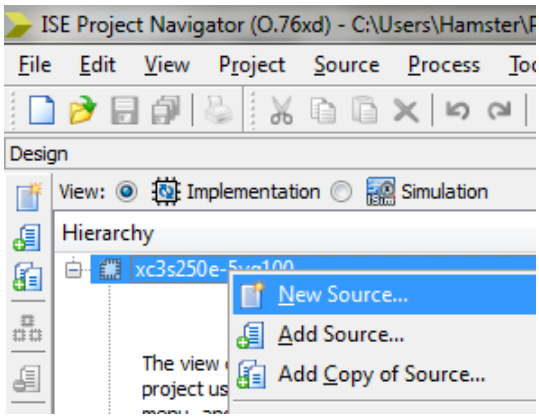
**For the Basys2 250**



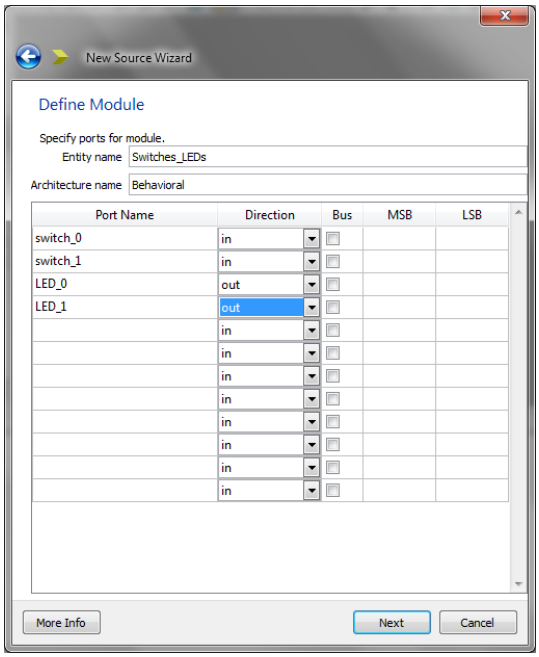

- Click on the "Finish" button to create and open the new project

## 6.2 Step 2 - Create a new VHDL Module

- Right-click on the design window, on the FPGA device, and choose "New Source"




- Highlight "VHDL module" and in the file name enter "Switches_LEDs", then press the "Next" button.
- This dialog box allows you to define what connections the module has. We need four connections - two the the switches and two for the LEDs:




- Click the "Next" button, then "Finish" to create the module and open it in the editor. To make things clearer, delete any line that starts with "--" - they are just comments that do not influence the design.

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

entity Switches_LEDs is
  Port ( switch_0 : in  STD_LOGIC;
         switch_1 : in  STD_LOGIC;
         LED_0 : out  STD_LOGIC;
         LED_1 : out  STD_LOGIC);
end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
begin

end Behavioral;
```

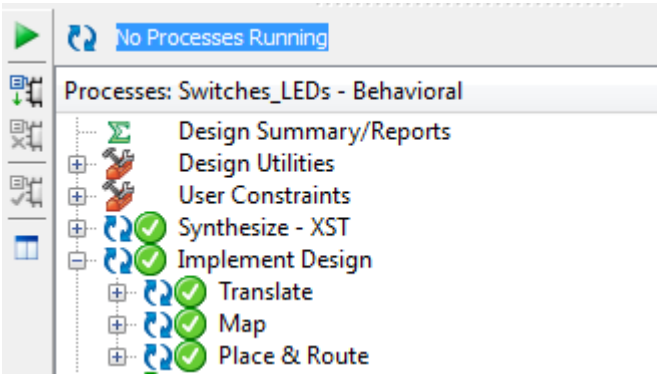
As you can see, it has created the definition for an entity called Switches_LEDs, with two inputs and two outputs - STD_LOGIC is used to indicate what values these inputs and outputs can have.

The architecture section is where you describe how the internal logic of the module actually works. For this project we use the "assignment" operator ("⇐") to assign the LEDs the values of the switches:

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

entity Switches_LEDs is
  Port ( switch_0 : in  STD_LOGIC;
         switch_1 : in  STD_LOGIC;
         LED_0    : out STD_LOGIC;
         LED_1    : out STD_LOGIC);
  end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
begin
  LED_0 <= switch_0;
  LED_1 <= switch_1;
end Behavioral;
```

If you press the green "play" arrow in the middle left of the design window the project should start building.

If your code has been entered successfully the project will build without any errors, and the design Window will now look like this:

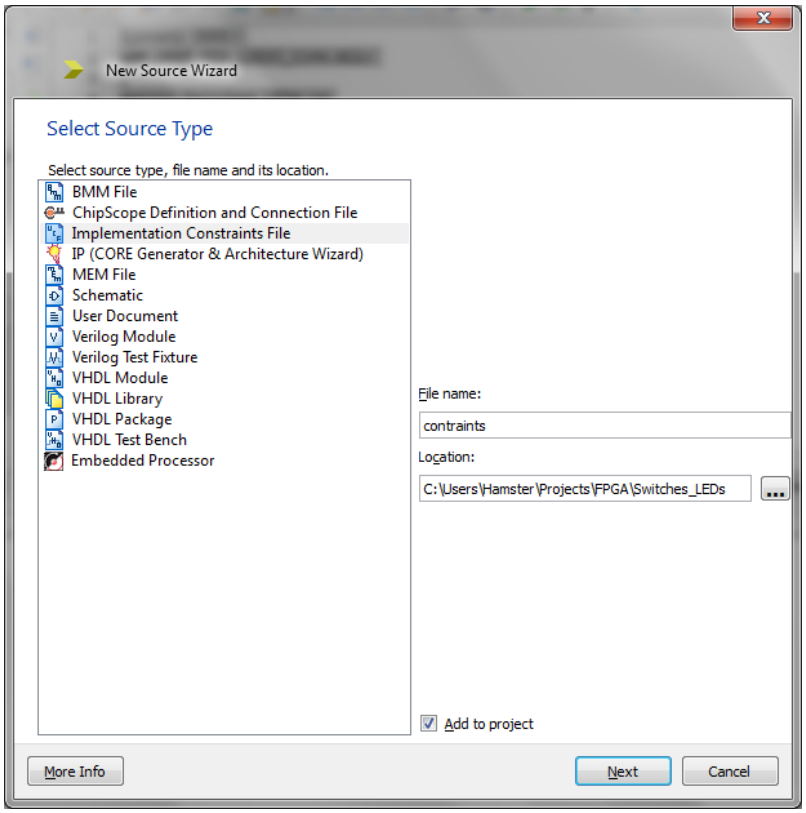



Great! You've built your first design! There is only one problem, and that is we haven't told the design tools which pin to connect these signals to.

## 6.3 Step 3 - Creating constraints

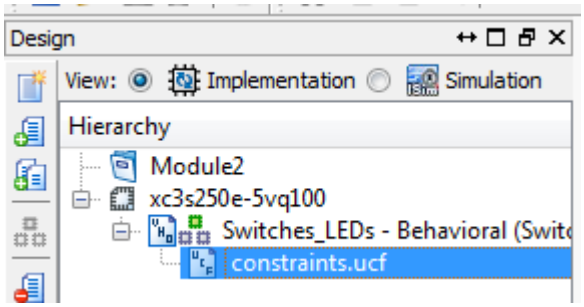
To tell the tools which physical pins should be connected to the VHDL inputs and outputs we need an "Implementation Constraints File". Here's how you add one:

- From the "Project Menu" choose "New Source"
- Select "Implementation Constraints File" and call it "constraints":




- Click "Next" and "Finished".

- In the design windows, a small "+" will appear by the Switches_LEDs module. Click that to show the new file:



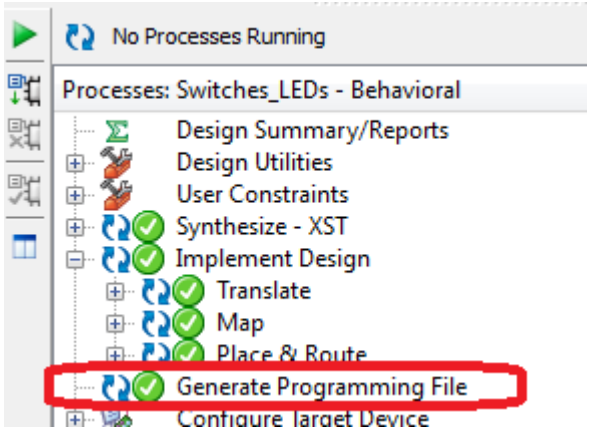

- Double click "constraints.ucf" to open it in the editor window.
- Add the following lines, which assign locations to the four wires, and instructs the tools to create a design that uses "Low Voltage Transistor Transistor Logic" signal levels:

```
# Constraints for Papilio One
NET switch_1 LOC = "P3"  | IOSTANDARD=LVTTL;
NET switch_0 LOC = "P4"  | IOSTANDARD=LVTTL;
NET LED_1    LOC = "P16" | IOSTANDARD=LVTTL;
NET LED_0    LOC = "P17" | IOSTANDARD=LVTTL;
```

```
# Constraints for Papilio Basys2
NET switch_1 LOC = "L3"  | IOSTANDARD=LVTTL;
NET switch_0 LOC = "P11" | IOSTANDARD=LVTTL;
NET LED_1    LOC = "M11" | IOSTANDARD=LVTTL;
NET LED_0    LOC = "M5"  | IOSTANDARD=LVTTL;
```

Save the changes to this file, and then once again click on the Green arrow to build the design.

If that is successful, double click on "Generate Programming file":

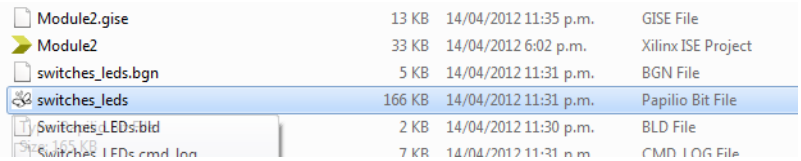



You will now have a *.bit* file in the project directory that can be used to program the FPGA!

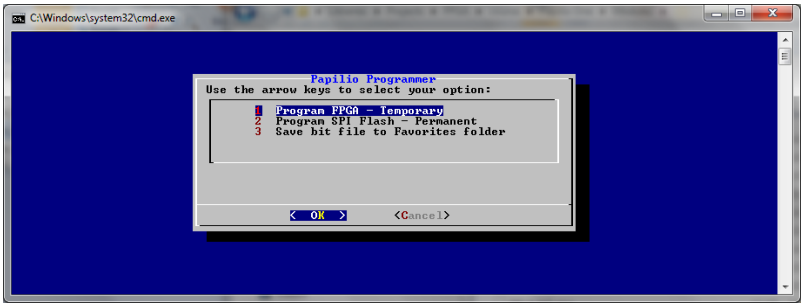
## 6.4 Step 4 - Downloading the design into the device

**For the Papilio One:**

- Connect your board to the USB port
- In Windows Explorer Navigate to the project directory and find the "Papilio Plus Bit File"



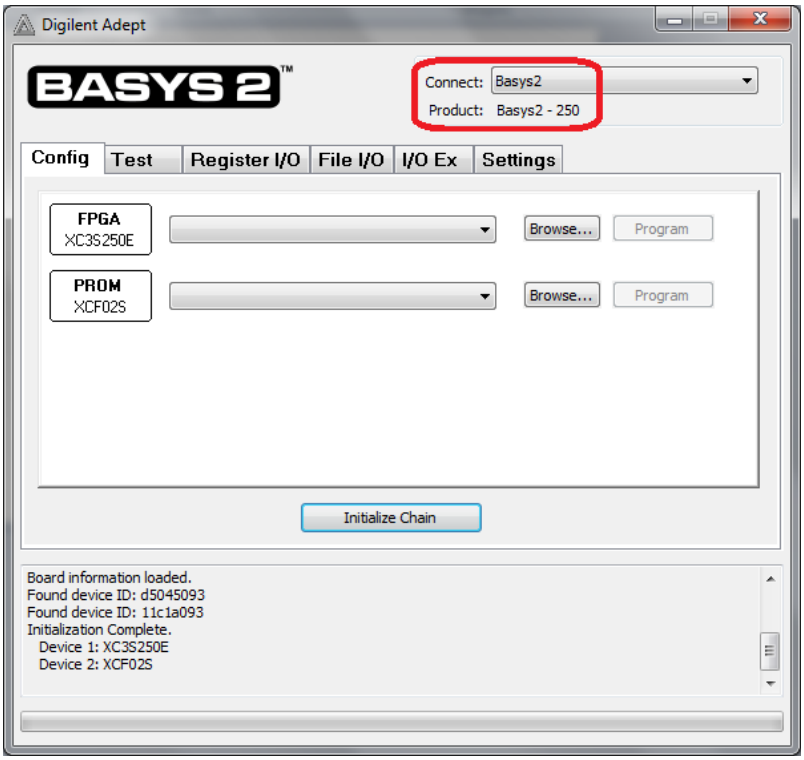

- Double-click on the file. It will bring up the following Window:



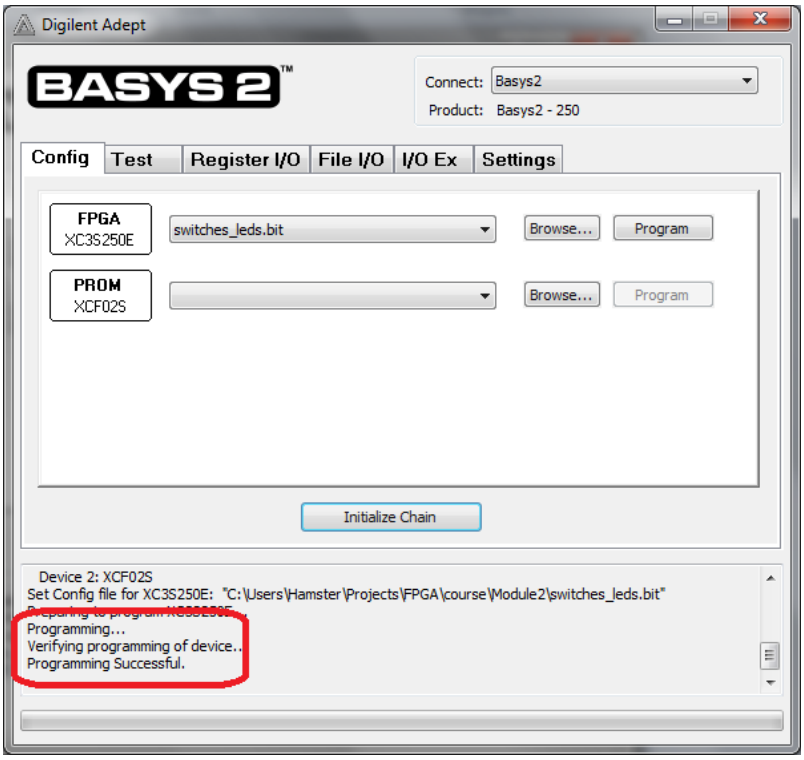

- Just press enter
- The design will be downloaded, and then the board will be configured with your design

**For the Basys2:**

- Connect your board to the USB port
- Launch Digilent's Adept software
- If the device isn't automatically detected, click on the "Device manager" and add it to the device table.




- Use the browse button to search for your project's .bit file
- Press the program button, and ignore the warnings about the JTAG clock
- The design will be downloaded, and then the board will be configured with your design



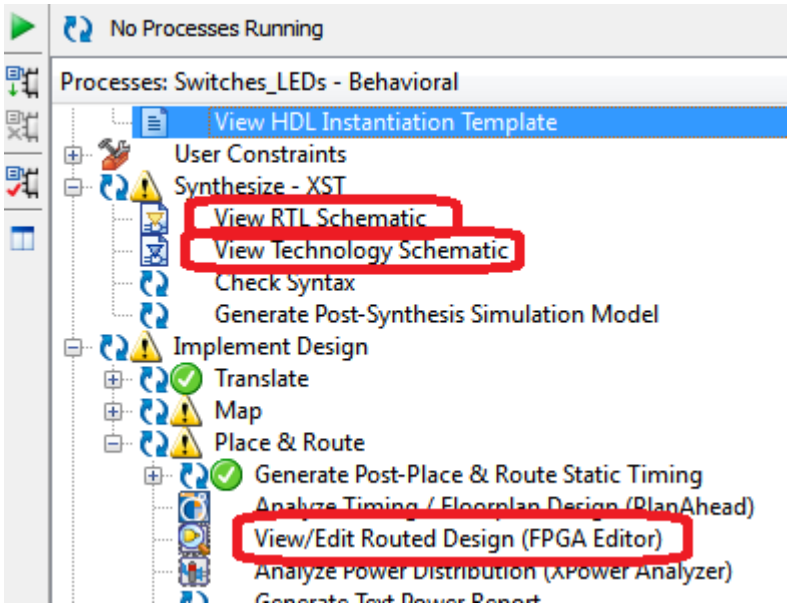
As you move the two rightmost slide switches the two rightmost LEDs should turn off and on. Well done!

## 6.5 Viewing how your design has been implemented

I find it interesting to see what the software tools make of my designs.

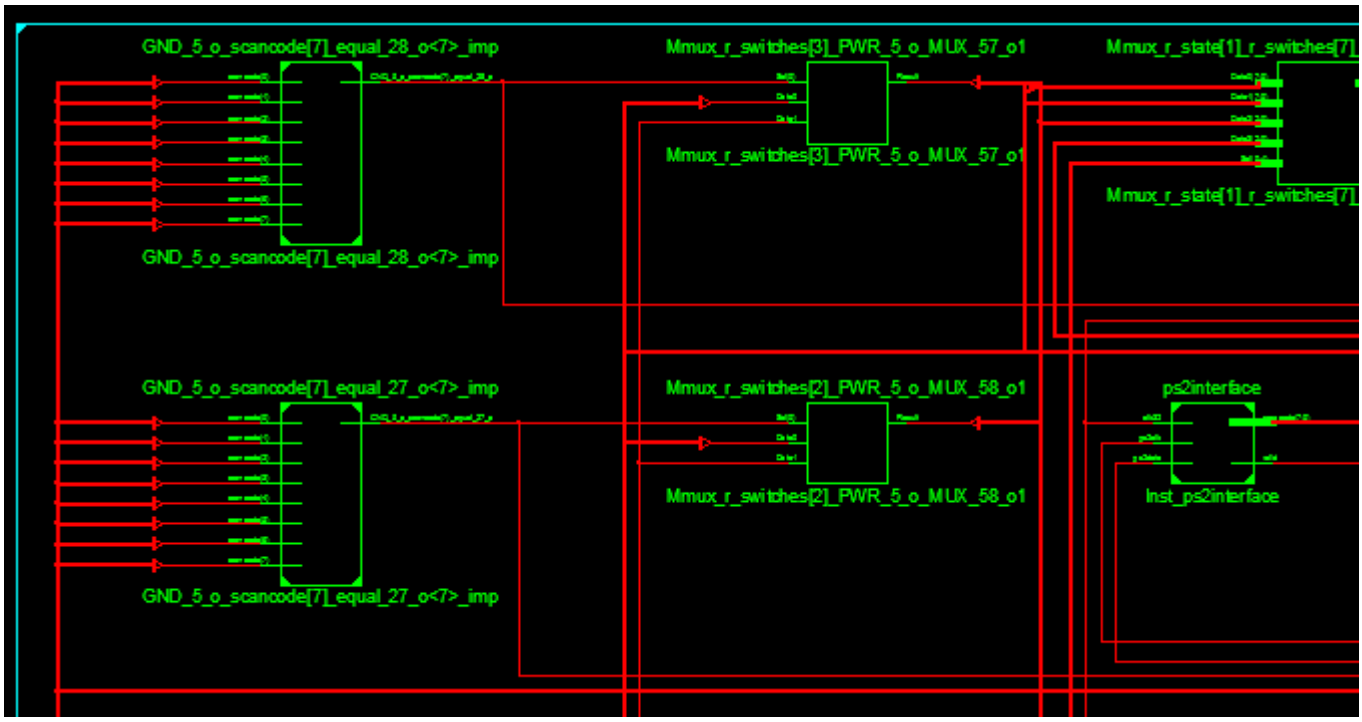
If you are keen you are able to view how your design is implemented within the FPGA at three different levels - Register Transfer, Technology and the Routed Design.

You can find the options to view buried away in the process tree:

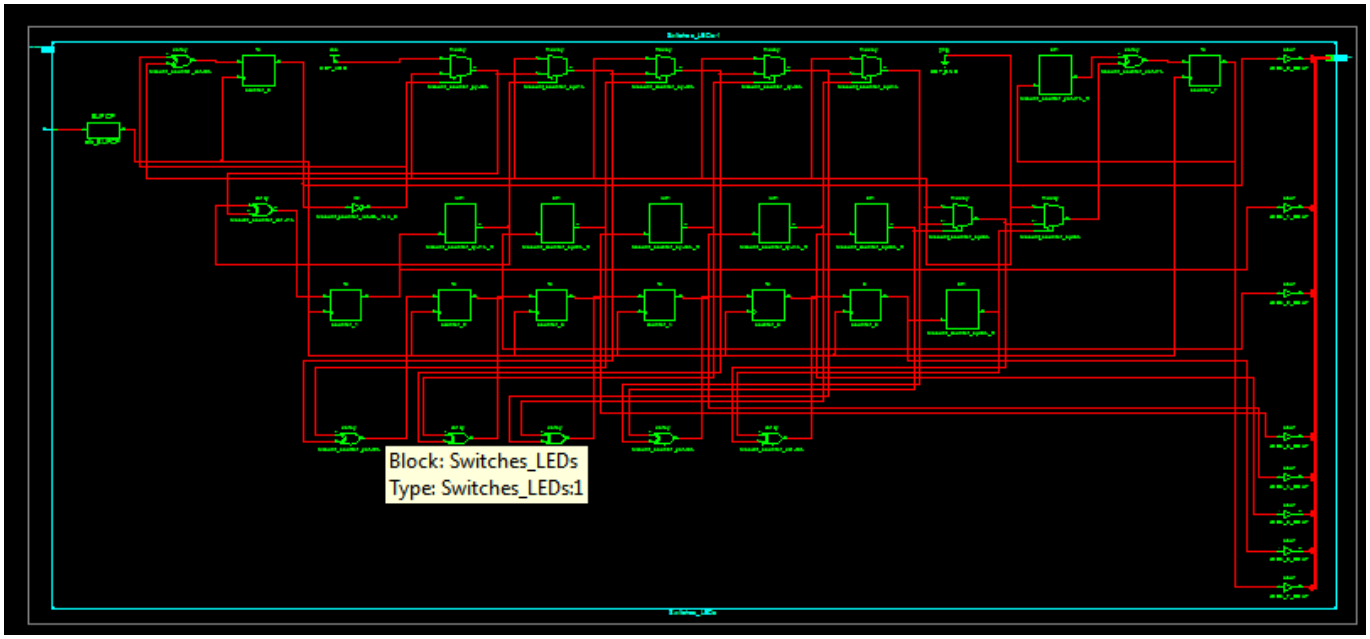



Here's a few screen shots from some other designs, and various different designs:

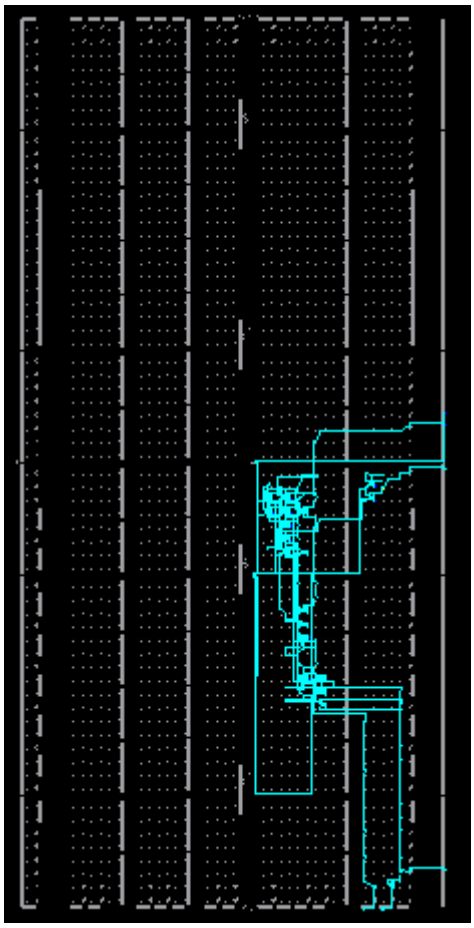
- The Register Transfer Level (*RTL*) schematic, which shows how all your design-level components are connected:




- The Technology Schematic, which shows how the individual components within the FPGA are connected:




- The Routed Design, which shows the physical locations and interconnects that are used on the FPGA chip:



# Chapter 7

# Binary operations

In this chapter we will experiment with binary operations in VHDL, using the switches for input and LEDs for output.

This project is much easier than Module 2, as you will only be changing one or two lines in your existing design, but there are a few short challenges with a lot more thinking and problem solving.

On finishing this chapter you should have re-familiarized yourself with the binary operators, learnt a bit about the VHDL STD_LOGIC data type, combined one or more signals to generate an output and then test the design in hardware.

During this process you will be building familiarity with the design tools and you might even have a small insight into the fundamentals of digital logic

## 7.1 The STD_LOGIC data type

As previously mentioned, a STD_LOGIC signal is analogous to a wire carrying a single bit of data.

Confusingly it is a lot more. Each STD_LOGIC signal can take on one of 9 different values at any one time!

But for most designs only three are used:

| **Value** | **Meaning** |
|---|---|
| *0* | Logical *Low* |
| *1* | Logical *High* |
| *Z* | High impedance (only used on bidirectional signals) |

It is important to note that these values are not numbers - in the source code they are enclosed in a single quote (').

The *Z* state does not actually happen inside the FPGA these days. Apart from on the input/output pins no tri-state logic exists within an FPGA - it is all mapped to multiplexers by the EDA tools. Because of this using tri-state logic for internal buses is not recommended as it makes the synthesis software has to do more work.

Of the other values there are "Weak high" (*H*), "Weak low" (*L*) that are mostly in interfacing and the "Uninitialized" (*U*), "Weak Uninitialized" (*W*), "Forcing unknown" (*X*), and "don't care" (-) that are only seen when simulating a design.

## 7.2 The basic Boolean operations

In most situations you will encounter four Boolean operations in VHDL:

| **Operation** | **Result** |
|---|---|
| NOT x | Result is *0* when x is *1*, otherwise *1* |
| x AND y | Result is *1* when both x and y are *1*, otherwise *0* |

| Operation | Result |
|---|---|
| x OR y | Result is *1* when either x is *1* or y is *1*, otherwise *0* |
| x XOR y | Result is *1* when only one of either x or y are *1*, otherwise *0* |

Two other less comon binary operations are supported:

| Operation | Equivalent expression |
|---|---|
| x NAND y | NOT(x AND y) |
| x NOR y | NOT(x OR y) |

## 7.3 Using these operators in VHDL

Using these operators is pretty simple. If you open up the project from Module 2 and make the changes to lines 13 and 14 as follows:

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

entity Switches_LEDs is
  Port ( switch_0 : in  STD_LOGIC;
         switch_1 : in  STD_LOGIC;
         LED_0 : out  STD_LOGIC;
         LED_1 : out  STD_LOGIC);
end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
begin
 LED_0 <= switch_0 AND switch_1;
 LED_1 <= switch_0 OR switch_1;
end Behavioral;
```

Using the instructions in Module 2 you will now be able to build this project and download it to your FPGA.

## 7.4 Project

- Compare *'switch_0 NAND switch_1'* with *'NOT(switch_0 AND switch_1)'*
- Compare *'switch_0 NAND switch_1'* with *'NOT(switch_0) OR NOT(switch_1)'*
- Compare *'NOT(switch_0)'* with *'switch_0 XOR switch_1'*, when switch 1 is on.

## 7.5 Challenges

- Can you make a design that will only light a LED when switch_0 is off and switch_1 is on?
- Can you make the AND operator out of only OR and NOT operations?
- Can you make the OR operator out of only AND and NOT operations?
- Can you make the OR operator out of only NOR operations? (hint, you can use the input values more than once)
- Experiment with XOR. Can you make an equivalent function out of only AND, OR and NOT? Can you make and AND or OR out of only XORs?

## 7.6 Further thinking

- Are any of the Boolean operations "fundamental"?, (meaning that all other operations can be built from them). This ability to implement any logic function using a set of generic components is core to how FPGAs implement your designs.
- For the operations that are not fundamental, what special ingredient are they missing?

# Chapter 8

# Using signal buses

You have now worked with signals one bit at a time, but that soon becomes tedious. What if you have thirty two bits of data to work with? There must be a much simpler way... and there is.

## 8.1 Using STD_LOGIC_VECTORs

In VHDL you can create signals that have more than one element in them (a bit like arrays in other languages).

The most common of these complex signal is a STD_LOGIC_VECTOR, which is conceptually a bundle of wires. Unlike most languages where usually one end of the range is implicitly defined, in VHDL you have to be explicit about the high and low element in the array, using *'(x downto y)*' - note that *'x*' is usually greater than or equal to *'y*'!

Unlike arrays in languages such as *'C*' you can perform operations on all the elements at once. Here is our switches and LEDs project re-coded to use buses that are two bits wide:

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

entity Switches_LEDs is
  Port ( switches : in  STD_LOGIC_VECTOR(1 downto 0);
         LEDs     : out STD_LOGIC_VECTOR(1 downto 0));
end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
begin
  LEDs <= switches;
end Behavioral;
```

If desired can address individual bits in a bus:

```
LEDs(0) <= switches(0);
LEDs(1) <= switches(1);
```

You can concatenate signals into a bus using the *'&*' operator. This code sample swaps the bits around so switch 0 lights LED 1 and switch 1 lights LED 0:

```
LEDs <= switches(0) & switches(1);
```

The important thing to remember is that like binary numbers the higher number bits are "to the left" of lower numbered bits - usually the opposite way that you think of an array.

Oh, and much like character arrays in *C* the other tricky bit is that constant expressions for STD_LOGIC_VECTOR use double quotes ("), instead of single quotes (' } that is used for individual elements:

This will work:

```
LEDs <= "10";
```

But this will throw an error:

```
LEDs <= '10';
```

To use the buses you will need to change your constraints file as follows:

```
# Constraints for Papilio One
NET switches(1) LOC = "P3"  | IOSTANDARD=LVCMOS25;
NET switches(0) LOC = "P4"  | IOSTANDARD=LVCMOS25;
NET LEDs(1)     LOC = "P16" | IOSTANDARD=LVCMOS25;
NET LEDs(0)     LOC = "P17" | IOSTANDARD=LVCMOS25;
```

```
# Constraints for the Basys2
NET switches(1) LOC = "L3"  | IOSTANDARD=LVTTL;
NET switches(0) LOC = "P11" | IOSTANDARD=LVTTL;
NET LEDs(1)     LOC = "M11" | IOSTANDARD=LVTTL;
NET LEDs(0)     LOC = "M5"  | IOSTANDARD=LVTTL;
```

## 8.2 Project - More LEDs and switches

- Modify your project to use buses.
- Extend the width of the buses to 8 bits ("(7 downto 0)"), and add the additional constraints for LEDs 2 through 7, and switches 2 through 7 to the *.ucf* file. To save you trolling through the documentation, here are the signal locations:

```
# Constraints for Papilio One
NET LEDs(7) LOC = "P5"  | IOSTANDARD=LVTTL;
NET LEDs(6) LOC = "P9"  | IOSTANDARD=LVTTL;
NET LEDs(5) LOC = "P10" | IOSTANDARD=LVTTL;
NET LEDs(4) LOC = "P11" | IOSTANDARD=LVTTL;
NET LEDs(3) LOC = "P12" | IOSTANDARD=LVTTL;
NET LEDs(2) LOC = "P15" | IOSTANDARD=LVTTL;
NET LEDs(1) LOC = "P16" | IOSTANDARD=LVTTL;
NET LEDs(0) LOC = "P17" | IOSTANDARD=LVTTL;

NET switches(7) LOC = "P91" | IOSTANDARD=LVTTL;
NET switches(6) LOC = "P92" | IOSTANDARD=LVTTL;
NET switches(5) LOC = "P94" | IOSTANDARD=LVTTL;
NET switches(4) LOC = "P95" | IOSTANDARD=LVTTL;
NET switches(3) LOC = "P98" | IOSTANDARD=LVTTL;
NET switches(2) LOC = "P2"  | IOSTANDARD=LVTTL;
NET switches(1) LOC = "P3"  | IOSTANDARD=LVTTL;
NET switches(0) LOC = "P4"  | IOSTANDARD=LVTTL;
```

```
# Constraints for the Basys2
NET LEDs(7) LOC = "G1"  | IOSTANDARD=LVTTL;
NET LEDs(6) LOC = "P4"  | IOSTANDARD=LVTTL;
NET LEDs(5) LOC = "N4"  | IOSTANDARD=LVTTL;
NET LEDs(4) LOC = "N5"  | IOSTANDARD=LVTTL;
NET LEDs(3) LOC = "P6"  | IOSTANDARD=LVTTL;
NET LEDs(2) LOC = "P7"  | IOSTANDARD=LVTTL;
NET LEDs(1) LOC = "M11" | IOSTANDARD=LVTTL;
NET LEDs(0) LOC = "M5"  | IOSTANDARD=LVTTL;

NET switches(7) LOC = "N3"  | IOSTANDARD=LVTTL;
```

```
NET switches(6) LOC = "E2"  | IOSTANDARD=LVTTL;
NET switches(5) LOC = "F3"  | IOSTANDARD=LVTTL;
NET switches(4) LOC = "G3"  | IOSTANDARD=LVTTL;
NET switches(3) LOC = "B4"  | IOSTANDARD=LVTTL;
NET switches(2) LOC = "K3"  | IOSTANDARD=LVTTL;
NET switches(1) LOC = "L3"  | IOSTANDARD=LVTTL;
NET switches(0) LOC = "P11" | IOSTANDARD=LVTTL;
```

Test that it works as expected

- Change the project to wire switches 0 through 3 to LEDs 4 through 7, and switches 4 through 7 to LEDs 0 through 3
- The AND, OR, NOT and related operators also work on buses. Change the project so that LEDs 0 through 3 show ANDing of switches 0 through 3 with switches 4 through 7, and LEDs 4 through 7 show ORing of switches 0 through 3 with switches 4 through 7

# Chapter 9

# Addition and subtraction, the hard way

Now that we have a firm handle on Boolean operators and signal buses we are in a position to implement one of computing's basic function - adding two numbers together.

## 9.1 Binary addition using Boolean operators

You don't have to, but as a learning exercise it is worthwhile to implement binary addition using simple operators.

To do so we have to make use of our first equivalent of a *local variable* - a signal that is used only within the Switches_LEDs entity.

We will need four of these local signals. To declare them the definition of the signal is added between the "architecture" and "begin" lines:

```
...
architecture Behavioral of Switches_LEDs is
  signal x      : STD_LOGIC_VECTOR(3 downto 0);
  signal y      : STD_LOGIC_VECTOR(3 downto 0);
  signal carry  : STD_LOGIC_VECTOR(3 downto 0);
  signal result : STD_LOGIC_VECTOR(4 downto 0);
begin
...
```

The size of *'result*' may look a little odd, but we are going to add two four-bit numbers on the switches, which gives a five bit result (as 15+15 = 30). Let's wire the LEDs up to the "result" signal, and set the unused LEDs to *0*.

```
LEDs <= "000" & result;
```

And we will also assign the values of X and Y to be the first and second group of four switches

```
 x <= switches(3 downto 0);
 y <= switches(7 downto 4);
```

Here is the code for adding the first bit:

```
result(0) <= x(0) XOR y(0);
carry(0)  <= x(0) AND y(0);
```

Not too hard. This a half adder - it's called this as it doesn not have a *carry in* only a *carry out*.

Now here's the second bit, which is a *full-adder* - it is a lot more complex as it has to deal with an x bit, a y bit and the carry bit from adding bit zero:

```
  result(1) <= x(1) XOR y(1) XOR carry(0);
  carry(1)  <= (x(1) AND y(1)) OR (carry(0) AND X(1)) OR (carry(0) AND Y(1));
```

It is a lot easier to understand the carry expression if you think of it as "are any two bits set?".

So here's the code up till now:

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

entity Switches_LEDs is
    Port ( switches : in  STD_LOGIC_VECTOR(7 downto 0);
           LEDs     : out STD_LOGIC_VECTOR(7 downto 0)
         );
end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
    signal x      : STD_LOGIC_VECTOR(3 downto 0);
    signal y      : STD_LOGIC_VECTOR(3 downto 0);
    signal carry  : STD_LOGIC_VECTOR(3 downto 0);
    signal result : STD_LOGIC_VECTOR(4 downto 0);
begin
   LEDs <= "000" & result;
   x <= switches(3 downto 0);
   y <= switches(7 downto 4);

   result(0) <= x(0) XOR y(0);
   carry(0)  <= x(0) AND y(0);

   result(1) <= x(1) XOR y(1) XOR carry(0);
   carry(1)  <= (x(1) AND y(1)) OR (carry(0) AND X(1)) OR (carry(0) AND Y(1));

end Behavioral;
```

## 9.2 Project - Adding four bits

- Based on the code so far, extend this to add all four bite (note - result(4) will be the value of carry(3)). Test the design for a few values.
- How many combination to you have to test to fully verify that your design works properly?
- Change the ordering of the statements in the code. Does it matter which order they are written in? Why would this be?

## 9.3 And now a better way to add (and subtract) numbers

Though interesting, this is a hard way to add numbers. VHDL includes standard libraries that make things a lot easier for you - STD_LOGIC_UNSIGNED allows you to treat your STD_LOGIC_VECTORS as if they are unsigned binary numbers.

To allow this, you need to add the following line to your code, just below the other "use" statement:

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

You can then just code the addition as:

```
result(4 downto 0) <= x + y;
```

Not requiring the ten lines or so of code.

The result of adding vectors will be only as long as the longest vector being added. In most cases this is one bit short that what is required to express the full range of results (and a warning like "Width mismatch. <result> has a width of 5 bits but assigned expression is 4-bit wide."). To ensure that I do get the full result I usually force at least one vector to be the length of the desired result:

```
result(4 downto 0) <= ('0' & x) + y;
```

## 9.4 Project - Adding four bit numbers

- Change your code to use unsigned addition. Test if it works.
- Try it both with and without adding an extra bit to the length of the *x* signal. Does it work as expected?

## 9.5 Challenges

- Implement binary subtraction using AND, OR, XOR and NOT operators
- Implement addition using only NOR or only NAND gates
- Design a project to add "00000001" to the value on the switches and display it on the LEDs. If you use the full adder code you will be able to simplify the logic down to a familiar pattern...

# Chapter 10

# Using a clock signal

Up to now all our project has been pure combinatorial logic - the output signals of the circuit is just a function of its inputs, and it has no internal state information (i.e. no storage elements are used). As you may have discovered the order of the statements made no difference to the design - all assignments were active all the time. Now that we can perform addition we should be able to make a project that implements a counter. But one thing is missing - the ability to track the passage of time.

This is the big difference between designing in VHDL and programming. In a program there is the "thread of execution" and it's associated state information - the program counter, the stack pointer, the contents of registers and so on. In VHDL there isn't.

What is needed is somewhere to store the values of signals and some way of synchronizing when these stored values should change. To progress our designs further need flip-flops and a clock signal.

## 10.1 Flip-flops

A flip-flop stores one bit of information, and this stored bit is updated when it's "Clock Enable" signal is asserted, and the desired transition occurs on the *clock* signal - either from *1* to *0* (a falling edge) or from *0* to *1* (a rising edge).

## 10.2 Clock signals

A clock signal is an input that has regular transitions from high to low, and is therefore very useful for keeping all the parts of a design in sync. The Papilio One has a clock signal running at 32,000,000 cycles per second (32MHz), where as the Basys2 board has a clock signal that can run at either 25,000,000, 50,000,000 or 100,000,000 cycles per second (10MHz, 50MHz or 100MHz). This is not as big a difference between boards as it sounds, because later on we will see how the FPGA can be used generate other frequencies from this reference clock.

This chapter's projects will be based around a binary counter running at 32MHz (so when a BASYS2 is used it will run about 50% quicker). As 32,000,000 is quite a big number (a million times faster than what human eyes can see) we need some way to slow it down. The easiest way is to create a 28 bit counter, and only show the top eight bits on the LEDs.

But first we need to know about VHDL processes and the "IF" statement.

## 10.3 VHDL Processes

From a designer's point of view a VHDL process is a block of statements that runs sequentially, and are triggered when any of the signals that it is *'sensitive*' to changes value.

Here is a process called *my_process* that is sensitive to two signals (input1 and input2):

```
my_process: process (input1, input2)
  begin
    output1 <= input1 and input2;
  end process;
```

**Note**

- Any event (change of value) on the the signals listed in the sensitivity list is what triggers the process.
- For purely combinatorial logic it should be all *inputs*, and none of the signals assigned within the process should be in its sensitivity list (or it will be evaluated multiple times during simulation)
- For a *clocked* processes the sensitivity list should be the clock signal and all asynchronous signals (i.e usually the clock signal and maybe an async reset).
- If you don't follow these rules you will get lots of odd behaviors in simulations as your process will be triggered when you don't expect, or fail to trigger at all. When you try to implement the design in hardware it will fail to work anything like it did in simulation.

The usefulness of processes is that they allow you to use sequential statements, the most useful of which is the *IF* statement.

## 10.4 IF statements

VHDL has an "if" statement, much like any other language. This syntax is:

```
  if [condition] then
    [statements]
  end if;
```

and

```
  if [condition] then
    [statements]
  else
    [statements]
  end if;
```

Remember that "if" statements can only be used inside processes block - if you attempt to use them outside of a process you will get compilation errors. Also remember that there is a *;* following the "end if" statement!

VHDL supports all the normal comparisons you would expect - just be aware that "not equals" is "/=" - very strange!

**Coding style tip**
If you are used to C, you might be tempted to use something like the following to implement a counter:

```
if(counter < counts-1)
  counter++;
else
  counter=0;
```

This is bad form as it is a comparison of value, and not a test for equality. The tools might implement this using "math" function, rather than a logic function. Due to the time that 'carries\' take (about 0.1ns per bit) this may lower your design's performance and increase resource usage.
If you can ensure that the value of "counter" stays between "0" and "counts-1" then it is far better to use the VHDL equivalent of the following:

```
if(counter == counts-1)
  counter=0;
else
  counter++;
```

This is because test for equality are much quicker - as no carry chain is needed

## 10.5 Detecting the rising edge of a clock

Any of the normal test (equality, inequality...) used in programming can be used to test values against each other. If we use these tests on our signals we usually end up generating combinatorial logic. For example:

```
select_switch: process(switch(0), switch(1), switch(2))
  begin
   if switch(0) = '1' then
     result <= switch(1);
   else
     result <= switch(2);
   end if;
end process;
```

This could equally implemented with the following concurrent (always active) statement:

```
result <= (switch(1) and switch(0)) or (switch(2) and not(switch(0)));
```

But we can also look for a signal's transition as the condition that triggers something to happen. The easiest way to do this is to use the "rising_edge" function:

```
if rising_edge(clock_signal) then
   result <= switch(1);
end if;
```

Another common way that you might see is to test the "event attribute" of the clock signal, which evaluates to true if this signal is the one that triggered the process to be evaluated, and then also check that the clock signal is *1*. Together these tests can detect the rising edge of the clock:

```
if clock_signal'event and clock_signal = '1' then
   [statements]
end if;
```

Although common in older text books, the use of "clock_signal'event and clock_signal = '1'" is now discouraged. It assumes that the clock signal was a *0* before the event was triggered, and can cause problems during simulation.

## 10.6 Declaring storage elements

Storage elements are declared just like a local signal. It is how you use them that implicitly makes them act as storage. If a signal only gets assigned during a clock transition it will be implemented using flip-flops:

```
...
architecture behavioural of counter
  signal counter : STD_LOGIC_VECTOR(7 downto 0);
begin

count: process(clock)
  begin
    if rising_edge(clock) then
      counter <= counter+1
    end if;
  end process;

end architecture;
...
```

The other situation that triggers a signal to be implemented as a flip-flop is when not all paths through a process assign it a value:

```
count: process(clock)
  begin
    if rising_edge(clock) then
      if switch1 = '1' then
        if switch2 = '1' then
           output_signal <= '1';
        else
           output_signal <= '0';
        end if;
      end if;
    end if;
  end process;
```

A flip-flop will be assigned to hold *output_signal* has to keep its value when switch1 changes from *1* to *0*.

And as with programming languages, it is always good practice to assign an initial value to your storage elements:

```
   signal counter : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
```

or perhaps more conveniently when working with larger signals:

```
   signal counter : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
```

It is usually safe to assume that uninitialized signals will be *zero*, however simulations will show the signal as being *'undefined'*, as will the result of any operations performed on that signal.

So here is the finished 8 bit counter:

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

entity Switches_LEDs is
   Port ( switches : in  STD_LOGIC_VECTOR(7 downto 0);
          LEDs     : out STD_LOGIC_VECTOR(7 downto 0);
          clk      : in STD_LOGIC
        );
end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
```

```
    signal counter : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
begin

clk_proc: process(clk)
  begin
    if rising_edge(clk) then
       counter <= counter+1;
    end if;
  end process;

end Behavioral;
```

**Warning**
Currently the project doesn't use the *LEDs* output - if you build using this code as-is the counter won't have any useful effect and will be optimized out, leaving you with an an empty design!

## 10.7 Project - Binary up counter

- Using the above template, extend the project to use a 30 bit counter ("29 downto 0"), displaying the top 8 bits on the LEDs. Remember to add a new constraint attaching that forces the *clk* signal to the correct pin for board's "clock" signal.

```
# Constraints for Papilio One
NET "clk" LOC = "P89" | IOSTANDARD = LVCMOS25 ;
```

```
# Constraints for the Basys2
NET "Clk" LOC = "B8";
NET "Clk" CLOCK_DEDICATED_ROUTE = FALSE;
```

## 10.8 Project - Binary down counter

- Change the project to count down

## 10.9 Project - Binary up/down counter

- Use one of the switches to indicate the direction to count.

## 10.10 Challenge

- Change the project to count accurately in (binary) seconds
- Change the project to time how long it takes to turn a switch on and off - you will need a second switch to reset the counter too!

# Chapter 11

# Assessing the speed of a design

So now we have a design that knows the passing of time, but how quick can we clock it? How can we find what limits the performance of the design? And if required, how do we make things faster?

## 11.1 The problem of timing closure

When working with FPGAs half the problem is getting a working design. The other half of the problem is getting the design to work fast enough! A lot of effort and trial and error can go into finding an solution that meets your design's timing requirements, and sometimes the best solution is not the obvious one.

It may be possible to change your FPGA for a faster grade part or use vendor specific macros to improve the performance of a design, but often there are significant gains to be made in improving your design that do not incur additional cost or limit your design to one architecture.

Even if your original design easily meets your requirements it is a good idea to look at the critical path and try to improve upon it. Not only is timing closure one of these problems where the more you practice the better you get, but usually a faster design will have quicker build times and fewer timing issues to resolve asto the design tools will have more *slack* when implementing a design.

## 11.2 This chapters's scenario

Imagine we are designing a project that needs to capture the timing and duration of a pulse to with better than 10ns accuracy, and these pulses occur within an interval of four seconds.

To give some design margin this calls for a 250MHz clock for the counter - giving at most 4ns of uncertainty around the start and end of the pulse, and a worst case of 8ns of uncertainty around the width of each pulse. Due to the timings of up to 4 seconds (1,000,000,000 ticks) a 30 bit counter is required.

So the goals of this design are simple - make a 30 bit counter that runs at 250MHz that can be used to time-stamp events.

## 11.3 So how fast can a design run?

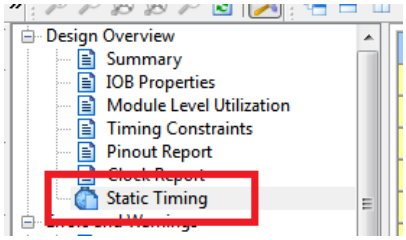
The answer is not what you most probably expect, rather than "You can clock this FPGA at X MHz" it is "as fast the chosen FPGA and your design allows".

Lets have a quick look at the two halves of this answer…

## 11.4 How the choice of FPGA changes speed

CPUs, memory and a lot of logic chips, FPGAs come in different speed grades. CPUs are rated in clock speeds of GHz or MHz, and memory is rated in clock speeds or access times, but for FPGAs there are no "simple numbers" - they come in speed grades. A different speed grade indicates that the devices performance is guaranteed to meet or exceed a set of modeling parameters, and it is these modeling parameters allows the design tools to calculate the performance limits of your design.

Once the design has been compiled and mapped to the FPGA components the tools calculate every path from input/output pins and flip-flops and totals the delay every step of the way (much like finding the critical path in project management software). This is then used to generate a report that can be reviewed from the "Static Timing" section of the "Design Summary Report" window:




The most useful number is usually right down the bottom:

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |    4.053|         |         |         |
---------------+---------+---------+---------+---------+
```

As this design has a minimum clock of 4.053 nanoseconds it can be clocked at up to 246MHz and still be within the FPGA's timing limits.

This is not fast enough in this scenario. Perhaps I could choose to use a faster grade FPGA. The gains for using a faster, more expensive chip are only minimal - for example going from a Spartan 3E -4 grade to -5 part increases a sample design's maximum speed by 13%.

## 11.5 How design decissions determine speed

Each flip-flop's input and output acts as a start or finish line for race. When a clock signal ticks the flip-flops assumes a new value, and the updated signal comes out of the flip-flops and starts rippling out through connected logic cells until all logic signals are stable. At that point we are almost ready for all the flip-flops to update their internal signals to values. For a design to work correctly the updated signal has to arrive at the flip-flop with at least enough time to ensure that when the clock ticks again the signal will be reliably captured.

So the three major components of this 'race\' are:

- Routing time - the time it takes to "charge the wires" that route signals between different logic cells. As you can well imagine the drive strength of the source signal, the length of these wires and the number of gates connected to the wires ("fan-out") dictates how much current is required to accurately transfer signals across the FPGA, and therefore the routing time.
- Logic time - this is the time it takes for a logic cell to react to a change of input and generate their new output values
- Setup time - the time required to ensure that the destination of a signal will accurately capture a changed value on the next clock transition.

As a sweeping generalization, the more complex work that is carried out each clock cycle the greater number of logic blocks will be in the critical path and the slower the design will run. As expected, the more you reduce the complexity of your design the quicker your design will run.

In some cases you can also use components from your FPGA vendor's library of standard building blocks. These building blocks will usually have more efficient implementations as they will leverage architecture-specific features within logic blocks (such as fast carry chains). The *cost* for using these features is that your design becomes architecture dependent and will need to be re-engineered if you move to a different FPGA.

You would think that a basic component like a 30 bit counter would be hard to improve on, but even in this simple design gains of 20% can be achieved!

## 11.6 Can it be made to run faster without changing the design?

Like using an optimizing compiler, giving the EDA tools a hint of how fast you need the design to run may improve things. When the EDA tools map the design to the FPGA they can be asked to take timing into account - causing them to attempt different placements for components on the FPGA until a timing constraint is met. The "Static Timing Report" will detail any errors (errors are where the amount of "slack" time between a signal's source and destination is less than zero - a negative "slack" means "not enough time").

## 11.7 The quick way to do this

The simple way to add a constraint is to include it in the Implementation constraints file by including these two lines:

```
NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 4 ns HIGH 50%;
```

The time of 4 nanoseconds ("4 ns") gives a constraint of a 250MHz to aim for.

---

**Note**
Usually you will use the actual clock period of the design (31.25ns for the Papilio One or 20ns for the Basys2 running at 50MHz)

---

## 11.8 The long way to do this

Timing contraints can also be entered using the GUI tools. You first have to successfully compile your project, so the tools can deduce what clocks are present. Then, in the process window, open the "Timing Constraints" tool:

module7/m7s2.png

You will then be presented with the list of all unconstrained clocks:

module7/m7s3.png

Double click on the unconstrained clock, and you will be presented with this dialogue box:

module7/m7s4.png

Fill it in appropriately then close off all the timing constraint related windows (forcing the constraint to be saved).

You will now need to rebuild the project with this new constraint in place.

## 11.9 What happens if the project can not meet the constraint?

If the design is unable to meet the timing requirements details of the failing paths will be reported in the Static Timing report:

```
...
Timing constraint: TS_clk = PERIOD TIMEGRP "clk" 4 ns HIGH 50%;
  465 paths analyzed, 73 endpoints analyzed, 1 failing endpoint
  1 timing error detected. (1 setup error, 0 hold errors, 0 component switching limit erro
  Minimum period is   4.053ns.
 --------------------------------------------------------------------------------


Paths for end point counter_29 (SLICE_X53Y78.CIN), 28 paths
--------------------------------------------------------------------------------
Slack (setup path):     -0.053ns (requirement - (data path - clock path skew + uncertainty
  Source:               counter_0 (FF)
  Destination:          counter_29 (FF)
  Requirement:          4.000ns
  Data Path Delay:      4.053ns (Levels of Logic = 15)
  Clock Path Skew:      0.000ns
  Source Clock:         clk_BUFGP rising at 0.000ns
  Destination Clock:    clk_BUFGP rising at 4.000ns
  Clock Uncertainty:    0.000ns

   Maximum Data Path: counter_0 to counter_29
     Location             Delay type         Delay(ns)  Physical Resource
                                                        Logical Resource(s)
     -------------------------------------------------  -------------------
     SLICE_X53Y64.XQ      Tcko                  0.514   counter<0>
                                                        counter_0
     SLICE_X53Y64.F4      net (fanout=1)        0.317   counter<0>
     SLICE_X53Y64.COUT    Topcyf                1.011   counter<0>
                                                        Mcount_counter_lut<0>_INV_0
                                                        Mcount_counter_cy<0>
                                                        Mcount_counter_cy<1>
     SLICE_X53Y65.CIN     net (fanout=1)        0.000   Mcount_counter_cy<1>
     SLICE_X53Y65.COUT    Tbyp                  0.103   counter<2>
                                                        Mcount_counter_cy<2>
                                                        Mcount_counter_cy<3>
     SLICE_X53Y66.CIN     net (fanout=1)        0.000   Mcount_counter_cy<3>
     SLICE_X53Y66.COUT    Tbyp                  0.103   counter<4>
                                                        Mcount_counter_cy<4>
                                                        Mcount_counter_cy<5>
     SLICE_X53Y67.CIN     net (fanout=1)        0.000   Mcount_counter_cy<5>
     SLICE_X53Y67.COUT    Tbyp                  0.103   counter<6>
                                                        Mcount_counter_cy<6>
                                                        Mcount_counter_cy<7>
     SLICE_X53Y68.CIN     net (fanout=1)        0.000   Mcount_counter_cy<7>
     SLICE_X53Y68.COUT    Tbyp                  0.103   counter<8>
                                                        Mcount_counter_cy<8>
                                                        Mcount_counter_cy<9>
     SLICE_X53Y69.CIN     net (fanout=1)        0.000   Mcount_counter_cy<9>
     SLICE_X53Y69.COUT    Tbyp                  0.103   counter<10>
                                                        Mcount_counter_cy<10>
                                                        Mcount_counter_cy<11>
     SLICE_X53Y70.CIN     net (fanout=1)        0.000   Mcount_counter_cy<11>
```

```
    SLICE_X53Y70.COUT    Tbyp                  0.103   counter<12>
                                                       Mcount_counter_cy<12>
                                                       Mcount_counter_cy<13>
    SLICE_X53Y71.CIN     net (fanout=1)        0.000   Mcount_counter_cy<13>
    SLICE_X53Y71.COUT    Tbyp                  0.103   counter<14>
                                                       Mcount_counter_cy<14>
                                                       Mcount_counter_cy<15>
    SLICE_X53Y72.CIN     net (fanout=1)        0.000   Mcount_counter_cy<15>
    SLICE_X53Y72.COUT    Tbyp                  0.103   counter<16>
                                                       Mcount_counter_cy<16>
                                                       Mcount_counter_cy<17>
    SLICE_X53Y73.CIN     net (fanout=1)        0.000   Mcount_counter_cy<17>
    SLICE_X53Y73.COUT    Tbyp                  0.103   counter<18>
                                                       Mcount_counter_cy<18>
                                                       Mcount_counter_cy<19>
    SLICE_X53Y74.CIN     net (fanout=1)        0.000   Mcount_counter_cy<19>
    SLICE_X53Y74.COUT    Tbyp                  0.103   counter<20>
                                                       Mcount_counter_cy<20>
                                                       Mcount_counter_cy<21>
    SLICE_X53Y75.CIN     net (fanout=1)        0.000   Mcount_counter_cy<21>
    SLICE_X53Y75.COUT    Tbyp                  0.103   counter<22>
                                                       Mcount_counter_cy<22>
                                                       Mcount_counter_cy<23>
    SLICE_X53Y76.CIN     net (fanout=1)        0.000   Mcount_counter_cy<23>
    SLICE_X53Y76.COUT    Tbyp                  0.103   counter<24>
                                                       Mcount_counter_cy<24>
                                                       Mcount_counter_cy<25>
    SLICE_X53Y77.CIN     net (fanout=1)        0.000   Mcount_counter_cy<25>
    SLICE_X53Y77.COUT    Tbyp                  0.103   counter<26>
                                                       Mcount_counter_cy<26>
                                                       Mcount_counter_cy<27>
    SLICE_X53Y78.CIN     net (fanout=1)        0.000   Mcount_counter_cy<27>
    SLICE_X53Y78.CLK     Tcinck                0.872   counter<28>
                                                       Mcount_counter_cy<28>
                                                       Mcount_counter_xor<29>
                                                       counter_29
    ----------------------------------------------  ---------------------------
    Total                                      4.053ns (3.736ns logic, 0.317ns route)
                                                       (92.2% logic, 7.8% route)
...
```

Because this is such a simple design minimal automatic improvement was possible, but it allows you to see where the crunch is - it is the time taken for the signal to propagate from "counter(0)" through to "counter(29)".

Also worthy of note is that each step along the way takes 0.103ns, indicating that there is a fundamental relationship between the size of numbers you manipulate and a design's speed - a 32 bit counter would take 0.206ns longer to compute the result.

Interestingly enough, given that 3.736ns of the time is incurred by logic and only 0.317ns is incurred by routing it would be easy to assume that a 30 bit counter can not be implemented to run faster than 267MHz in this FPGA - any faster and there would not be enough time for the logic to do its magic. You would be wrong - the final design in this chapter runs at 298MHz!

## 11.10 So how can something as simple as a counter be improved?

Here's the existing design - it is a small and easy to understand design. It also allows easy verification by checking that the LEDs count at the same speed whenever changes are made:

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

entity Switches_LEDs is
   Port ( switches : in  STD_LOGIC_VECTOR(7 downto 0);
          LEDs     : out STD_LOGIC_VECTOR(7 downto 0);
          clk      : in STD_LOGIC
        );
end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
   signal counter : STD_LOGIC_VECTOR(29 downto 0) := (others => '0');
begin

 LEDs <= counter(29 downto 22);

clk_proc: process(clk, counter)
  begin
     if rising_edge(clk) then
        counter <= counter+1;
     end if;
  end process;
end Behavioral;
```

As mentioned earlier, the problem is the of length "counter" - at 30 bits long it will take at least 30*0.103ns = 3.09ns to increment. How about splitting the counter into two 15 bit counters? will it be any faster? Lets see - Here's the updated design:

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

entity Switches_LEDs is
    Port ( switches : in  STD_LOGIC_VECTOR(7 downto 0);
           LEDs     : out STD_LOGIC_VECTOR(7 downto 0);
           clk      : in STD_LOGIC
         );
end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
    signal counter : STD_LOGIC_VECTOR(29 downto 0) := (others => '0');
    signal incHighNext : STD_LOGIC := '0';
begin

  LEDs <= counter(29 downto 22);

clk_proc: process(clk, counter)
  begin
    if rising_edge(clk) then
       counter(29 downto 15) <= counter(29 downto 15)+incHighNext;

       if counter(14 downto 0) = "111111111111110" then
          incHighNext <= '1';
       else
          incHighNext <= '0';
       end if;

       counter(14 downto 0) <= counter(14 downto 0)+1;
    end if;
  end process;
end Behavioral;
```

Here's the updated timing report:

All values displayed in nanoseconds (ns)

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |    3.537|         |         |         |
---------------+---------+---------+---------+---------+

Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 284 paths, 0 nets, and 82 connections

Design statistics:
  Minimum period:   3.537ns{1}   (Maximum frequency: 282.725MHz)
```

By using a little more logic the design has gone from 246MHz to 282MHz - about 14% faster.

Why does this work? By exploiting that we know in advance (which is when there will be a 'carry\' from bit 14 to bit 15), and storing that in a handy flip-flop (*incHighNext*) we have split the critical path across two clock cycles.

## 11.11 Project - More speed!

- See what the maximum speed the design using the 30 bit counter is for your FPGA board
- Try changing the speed grade of the FPGA and see what difference that makes to the timing. (To do this, in the hierarchy window right-click on xc3c100e-4cp132 and choose 'properties\' - just remember to set it back!)
- Add a timing constraint and try again
- See what the maximum speed the design using the 15+15 split counter is for your FPGA board.

## 11.12 Challenges

- Is the 15+15 split counter optimal compared against a 14+16 split or 16+14 split? If not, why not?
- Can you increase the maximum clock speed for the project even further?
- What is the largest counter you can make that runs at 100MHz?

## 11.13 An even better design

Sometimes designs just need a different way at looking at things. The code below performs faster still - why is this?

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

entity Switches_LEDs is
  Port ( switches : in  STD_LOGIC_VECTOR(7 downto 0);
         LEDs     : out STD_LOGIC_VECTOR(7 downto 0);
         clk      : in STD_LOGIC
       );
end Switches_LEDs;

architecture Behavioral of Switches_LEDs is
  signal counter : STD_LOGIC_VECTOR(29 downto 0) := (others => '0');
  signal incHighNext : STD_LOGIC := '0';
begin

 LEDs <= counter(29 downto 22);

clk_proc: process(clk, counter)
  begin
    if rising_edge(clk) then
       counter(29 downto 15) <= counter(29 downto 15)+incHighNext;

       incHighNext <= not counter(0)  and counter(1)  and counter(2)  and counter(3)
                      and counter(4)  and counter(5)  and counter(6)  and counter(7)
                      and counter(8)  and counter(9)  and counter(10) and counter(11)
                      and counter(12) and counter(13) and counter (14);

       counter(14 downto 0) <= counter(14 downto 0)+1;
    end if;
  end process;
end Behavioral;
```

## 11.14 Random thoughts on timing

- If there is a chance that performance will be an issue, consider setting a metric for the "levels of logic" you have in your design, and regularly review your design's static timing during the project. This is a very simple way to ensure that you will not end up with one or two very long chains of logic that limits design performance requiring significant re-work to resolve.
- Vendors will tell you that "floorplanning" (the high-level planning of the the placement of logic in a design) will allow projects to make significant improvement in achieving timing closure, but only routing delays can be reduced by controlling the placement. This is much like saying that an optimizing compiler can fix code performance issues.
- The best way to solve tricky timing closure issues is to avoid the timing issues in the first place with clean simple designs.
- It is usually possible to identify the critical within a project in advance, allowing the feasibility of a design to be assessed early on in a project
- Design all projects with speed in mind, even if the design requirements do not dictate it. A design that can run very fast is also very efficient on power when running at lower clock speeds, and it is great practice.
- "Mapping" options can have a big difference on the final design performance. In general settings that reduce size of a design make the design slow (due to high fan-outs and merging of flip-flops).
- The same design will usually run faster on a larger FPGA due to greater freedom in the place and route process. Likewise downsizing an existing design into a FPGA that is 'just big enough\' will lower performance.

# Chapter 12

# Using the ISIM simulator

Now that we have a design that changes millions of times a second testing becomes hard. In this module we will use the ISIM simulator - a tool that allows you to 'run\' the logical design and see how it behaves as it is poked and prodded with external signals.

## 12.1 What is simulation?

When you debug software you are actually running the code on the processor, with all the access to the system resources such as the OS, memory, communications and file systems. Unlike debugging software, simulating a FPGA project doesn't run it on the actual hardware - the closest equivalent you may have experience with the simulation of a micro-controller in MPLAB or WinAVR.

Although no FPGA hardware is involved simulation is very powerful - it is very much like having the most powerful of logic analyser at your fingertips. The downside is that if your idea of how an external device works isn't accurate you will not be able to spot the problems.

The initially confusing bit about simulation is that it requires another VHDL module to drive the input signals and receives the outputs from your design - a module that is called a "test bench". They are pretty easy to spot - the ENTITY declaration has no "IN" or "OUT" signals, just something like this:

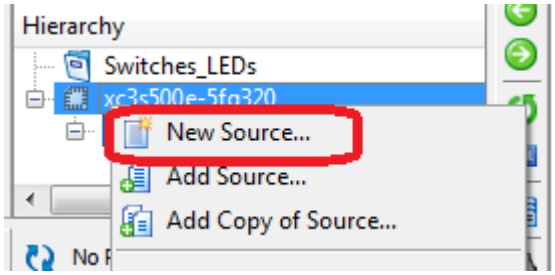
```
ENTITY TestBench IS
END TestBench;
```

In effect the test bench is a module ties up all the lose ends of your design so that the simulator can run it.

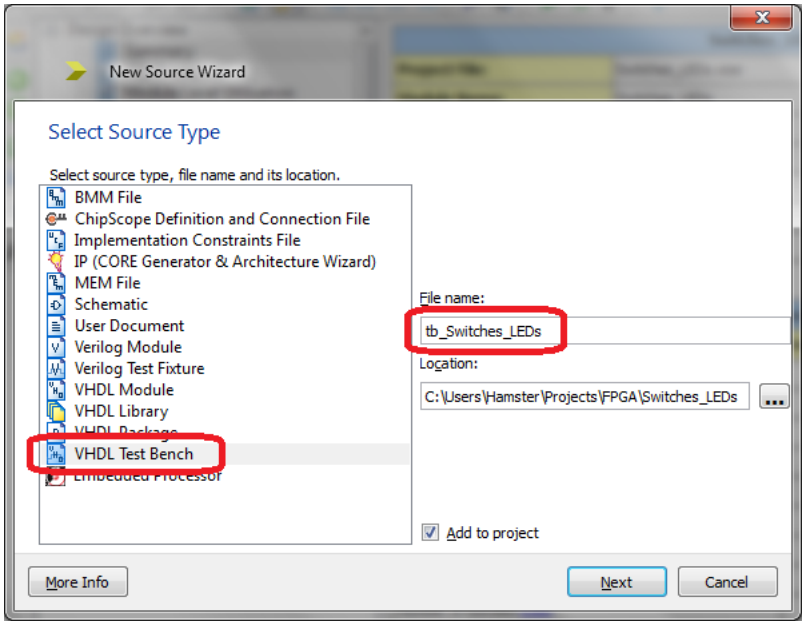
## 12.2 Creating a test bench module

Here is how to create a test bench using the wizard in WebPack.

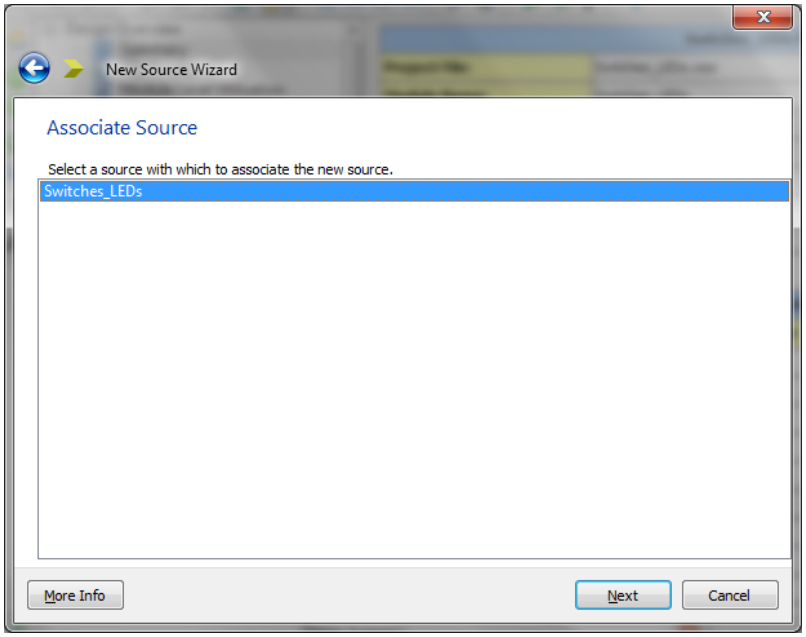
Right-click on the top level of the hierarchy and select to add a new source module into the project:



Select the "VHDL Test Bench" and assign it a name (I just add *tb_* to the name of the component being tested), then click *Next*:




You will then need to select which component of the design you wish to test and then click *Next*:




A summary screen will be presented - review the details and then click "Finish'.

## 12.3 Break-down of a Testbench module

Here is the resulting VHDL with most of the comments removed, to reduce its size:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY tb_Switches_LEDs IS
END tb_Switches_LEDs;

ARCHITECTURE behavior OF tb_Switches_LEDs IS
```

```
    COMPONENT Switches_LEDs
    PORT(
         switches : IN  std_logic_vector(7 downto 0);
         LEDs : OUT  std_logic_vector(7 downto 0);
         clk : IN  std_logic
        );
    END COMPONENT;

   --Inputs
   signal switches     : std_logic_vector(7 downto 0) := (others => '0');
   signal clk          : std_logic := '0';

    --Outputs
   signal LEDs         : std_logic_vector(7 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: Switches_LEDs PORT MAP (
          switches => switches,
          LEDs => LEDs,
          clk => clk
        );

   -- Clock process definitions
   clk_process :process
   begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      wait for 100 ns;
      wait for clk_period*10;
      wait;
   end process;
END;
```

This has a few more language structures that have not been seen so far. First is a component declaration, which defines the project that is being tested - much like a C function prototype:

```
    COMPONENT Switches_LEDs
    PORT(
         switches : IN  std_logic_vector(7 downto 0);
         LEDs : OUT  std_logic_vector(7 downto 0);
         clk : IN  std_logic
        );
    END COMPONENT;
```

There is a "constant" declaration, which is of a "time" data type - this data type is exclusively used in simulation. If your design has a timing constraint the value here is usually set correctly, but it pays to check:

```
   constant clk_period : time := 20 ns;
```

The next stanza is creating an instance of the Switches_LEDs component, and attaching its signals to the signals within the test bench:

```
uut: Switches_LEDs PORT MAP (
       switches => switches,
       LEDs => LEDs,
       clk => clk
     );
```

And finally two processes that contain "wait" statements that controls the timing of signals within the simulation:

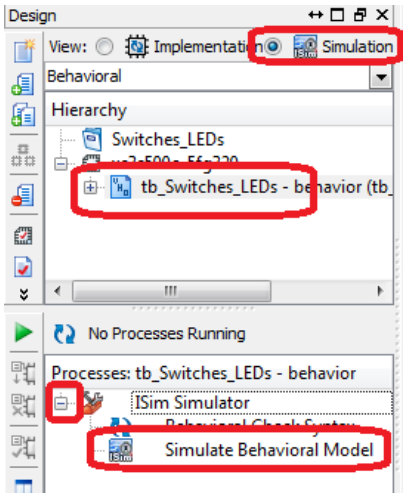
```
clk_process :process
begin
   clk <= '0';
   wait for clk_period/2;
   clk <= '1';
   wait for clk_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
   wait for 100 ns;
   wait for clk_period*10;
   wait;
end process;
```

The first process (*clk_process*) defines the clock signal - which will stay *0* for ten (simulated) nanoseconds, then flip to *1* for ten nanoseconds - giving a 20ns (50MHz) clock. The seconds process (*stim_proc*) is where you add statements to change the inputs of the unit under test - for example you could use "switches ⇐ "11111111" to simulate the switches being turned on. When initially created all inputs (other than the clock signal) are set to *0*.

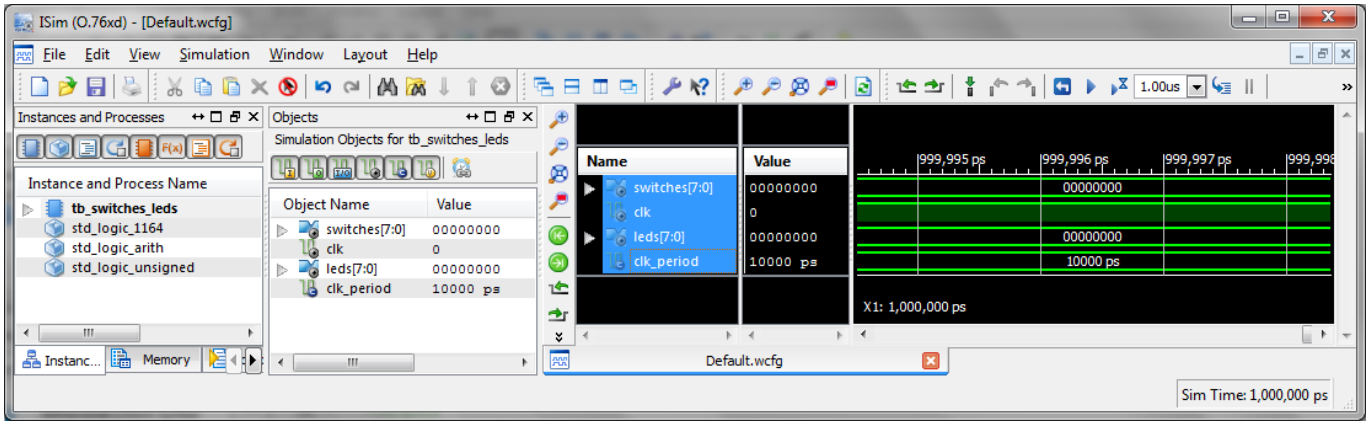
**Warning**
The "wait for [time period]" can not be realized inside an FPGA, so it is only useful inside simulations. If you use this statement outside of a test bench your design it will simulate perfectly well but you will not be able to implement your design in the FPGA.

## 12.4 Starting the simulation

From top to bottom, switch to "Simulation" view, select the desired test bench (you can have more than one), expand the "Processes" tree, and then double click on "Simulate behavioral model" - as a quirk, if you have just finished a simulation you may need to right-click on this and choose "Run all".



The simulation will be compiled, and then the simulator tool is launched. On start-up the simulator will simulate the first microsecond:

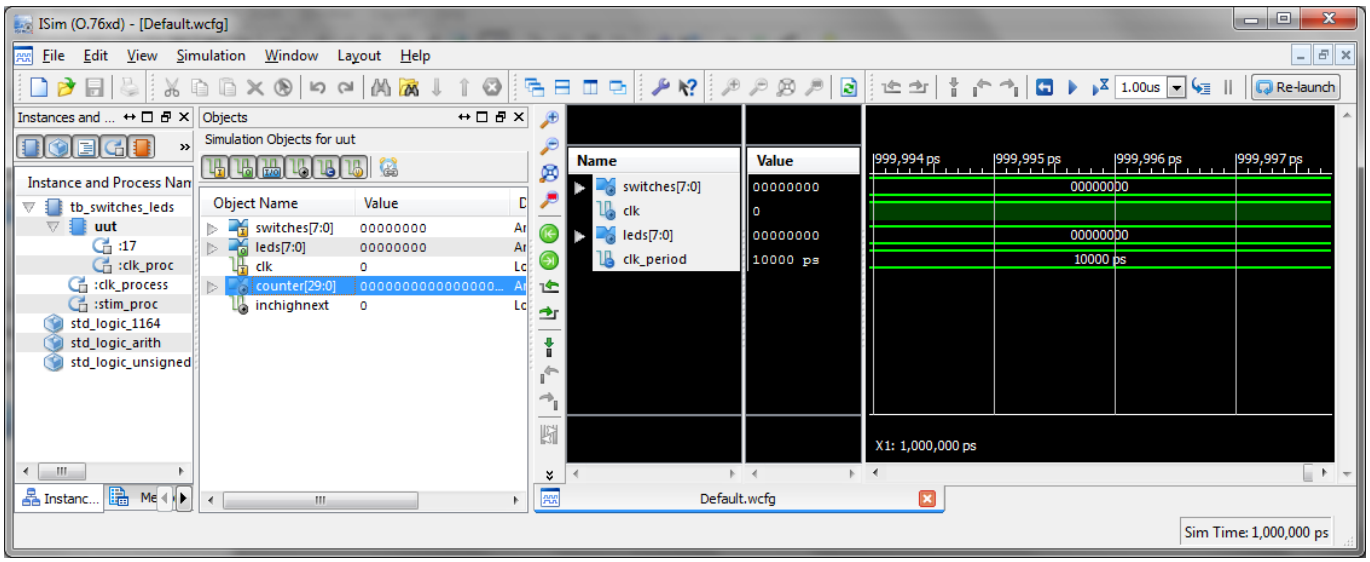



## 12.5 Using the simulator

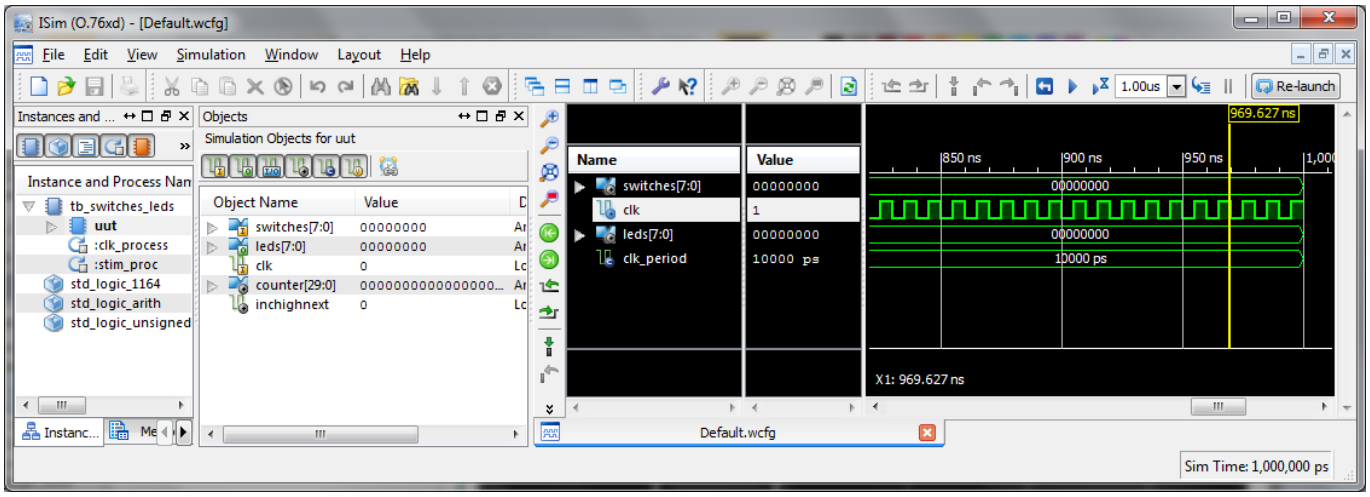
From left to right you have the following panes:

- Instances and processes - the design hierarchy being simulated
- Objects - what signals are in the selected instance
- Waveform window - A list of signals being recorded, and a graphical display of their values over time.

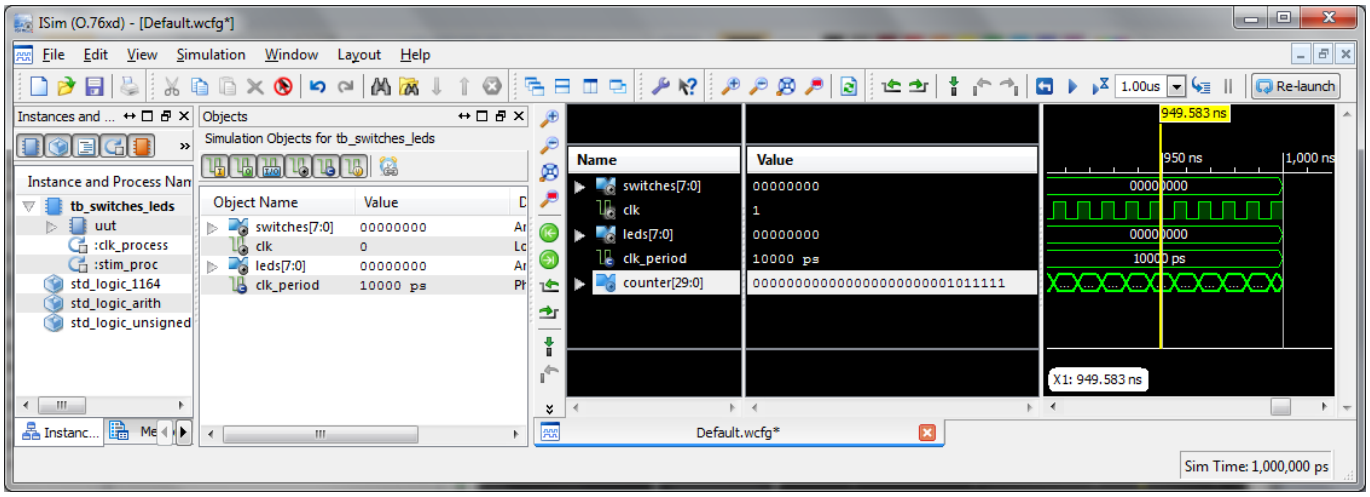
Expand the tb_switches_leds instance in the 'Instances and processes\' pane, and click on the "uut". In the "Objects" pane you will then see all the signals in your design:




Also, the default timescale is very small - 10 or so picoseconds. You can click the "zoom out" on the toolbar until you can see the clock signal ticking away:




As desired you can drag a signal from the "Objects" pane into the Waveform window, but as the signal has not yet been recorded you will need to click the "Reset" and then "Run for specified time" to get values displayed in the window. In this screen-shot I have dragged "counter[29:0]" from UUT into the waveform window and rerun the simulation:




As you drag and click on the Waveform pane the value of that signal at that time is shown - unlike when debugging code, in ISIM you can trace backwards in time!

## 12.6 Project

- Make some part of the design dependent on the state of one of the switches. Simulate the design after adding assignments to change the switch signal in the stimulus process.
- Right-click on some of the signals in the waveform window and explore the "radix" and cursor options.
- Click and drag over the waveform window to measure the duration of a signal from transition to transition
- Click on the triangle to the left of a bus's name. What happens?

## 12.7 Points to ponder

- Does the simulation take into account the propagation delays inside the internal logic of the FPGA?
- If a signal changes at exactly the same time that clock signal's rising edge, what happens?

# Chapter 13

# Using more than one module in a design

Up to now the designs have consisted of only one entity. Just like in software, there quickly comes a time when putting every in one source file is no longer practical. There is also the need to separate designs into functional units that can be designed and tested independently of each other, before they are integrated into the one design.

In VHDL speak, these are called modules.

## 13.1 Using more than one source module in a design

VHDL achieves this though "architectures", "components", "entities" and "instances" - we have already breezed over all of this.

- The "entity" statement defines the *inside* view of a module's interface

```
entity mymodule is
   Port ( input1 : in  STD_LOGIC_VECTOR (3 downto 0);
          output1 : out  STD_LOGIC_VECTOR (3 downto 0));
end mymodule;
```

This is at the top of the defining module, following the "use" statements.

- The "architecture" defines how a component works - it contains all the internal signals and sub-components, and all the internal logic:

```
architecture Behavioral of mymodule is
begin
  output1 <= input1;
end Behavioral;
```

This is usually the bulk of the module, and appears after the entity statement.

- The "component" statement defines the 'external\' connections of the module, and appears in the module that uses the component:

```
COMPONENT mymodule
PORT(
  input1 : IN std_logic_vector(3 downto 0);
  output1 : OUT std_logic_vector(3 downto 0));
END COMPONENT;
```

Component declarations appear in the same area of the code as the signal declarations.

- The "instance" statement describes the connections of the component inside the containing module - it is this that actually triggers the component to be included in the final design:

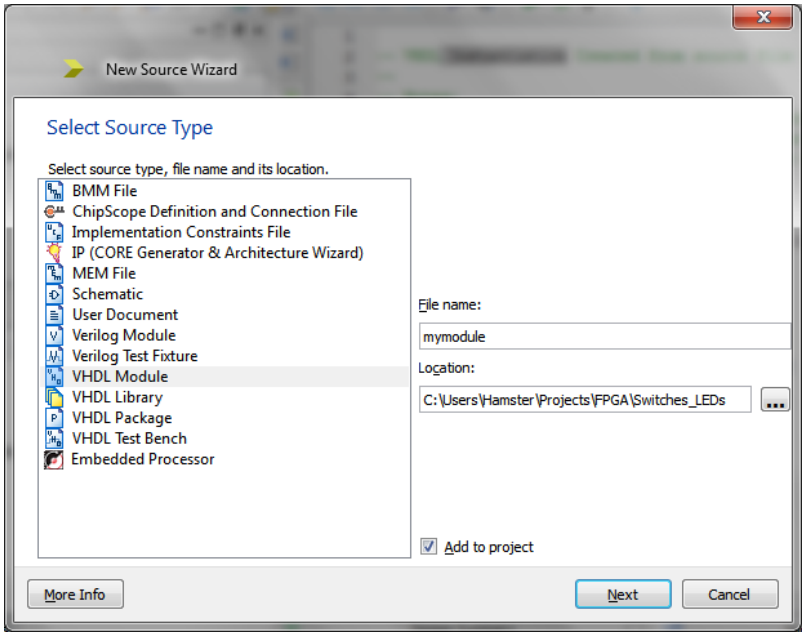
```
Inst_mymodule: mymodule PORT MAP(
   input1 =>  input_signal1,
   output1 => output_signal1
);
```

These can be intermingled with the assignment statements and processes, but not contained within a process block. One source of frustration for me is that when signals are mapped they can not be operated on (e.g. *input_a* ⇒ *signal_a* is valid but *input_a* ⇒ *NOT(signal_a)* is not). All inputs should have a value, but if you don't want to use an output, you can map it to the keyword "open" (e.g. "output1 ⇒ open").

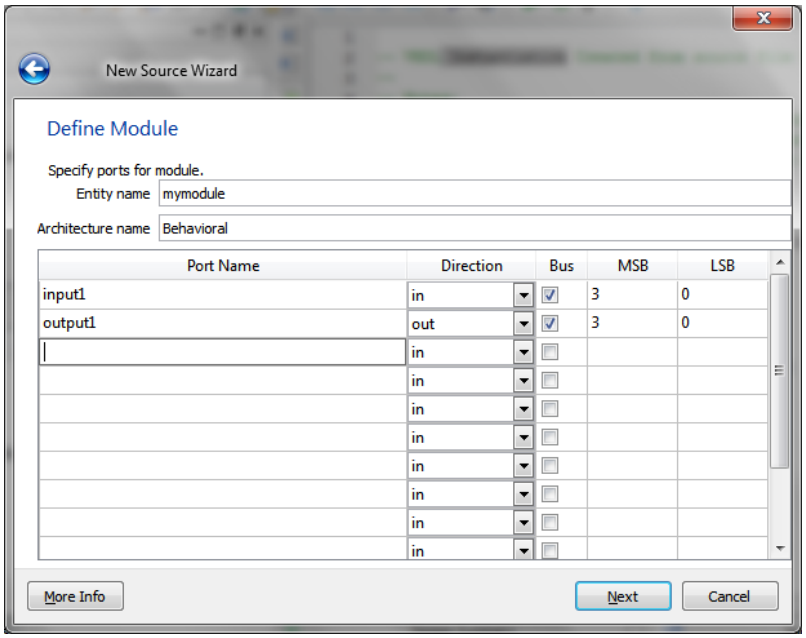
## 13.2 Creating a module using the wizard

The easy way to do create a new module is using the "New Source…" wizard.

On the first screen give the module a name:

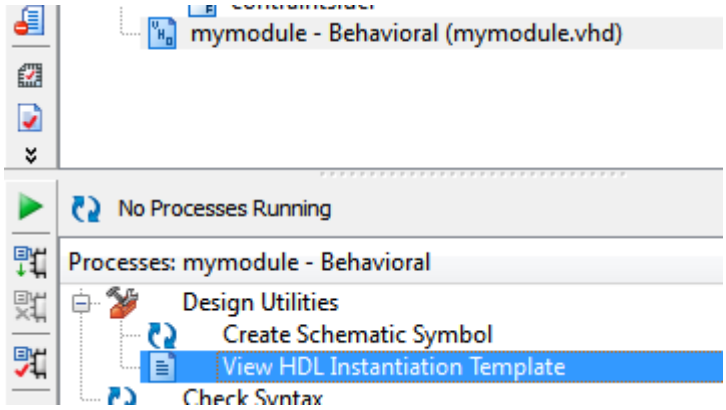



Then define the interface - do not worry if you are not 100% sure of the signals, you can change them directly in the source afterwards:




You are then presented with a summary screen, and click finish.

Once you have a new module you can highlight it and under "Design Utilities" you can run the "View HDL instantiation Template" process to get a template that you can cut and paste as needed:




It will look something like this:

```
COMPONENT mymodule
PORT(
    input1 : IN std_logic_vector(3 downto 0);
    output1 : OUT std_logic_vector(3 downto 0)
);
END COMPONENT;

Inst_mymodule: mymodule PORT MAP(
   input1 => ,
   output1 =>
);
```

In most large designs the very top level module ends up containing very little logic and resembles a big wiring loom - with a lot of instances of smaller components and the signals that interconnect them.

## 13.3 Project

- Create a new module - a 30 bit counter called "counter30", with the following external signals:

  - clk : in STD_LOGIC
  - enable : in STD_LOGIC
  - count : out STD_LOGIC_VECTOR(29 downto 0)

The internal design is up to you, but project 6.1 will be pretty close

- View the *Instantiation template* for your component. Copy the component declaration into your switches_leds.vhd source
- In switches_leds create an instances of counter30.
  - Connect the counter's *count* output to a bus called count1
  - Connect the "enable" signal to switch(0)
  - Connect the the clock.
  - Connect the top four bits of count1 to LEDs(3 downto 0). Remember to add a signal definition for *count1*.
- Implement the design and test that it works as expected - switch 0 should enable the counter driving the lower four LEDs. It is usual to get a lot of warnings about unused signals that will be timed from the design. This is expected as we are only using the top four bits of the counters.
- Create a second instance of counter30 in the switches_leds vhd source.
  - have it's *count* output connected to a bus called *count2*
  - connect the "enable" signal to switch(1).
  - connect the top four bits of *count2* to LEDs(7 downto 4)
- Check that this too works as expected

# Chapter 14

# A better display than LEDs

Now is a good time to cover a little more VHDL, and use it to efficiently implement a design that controls the seven segment display.

## 14.1 The VHDL *case* statement

Much like "switch()" in C, VHDL has the CASE statement that allows you to choose different between multiple different paths through your code based on the value of a signal. Although it is largely functionally equivalent to nested 'IF's it is far easier to write, and is implemented more efficiently within the FPGA.

It looks much like this:

```
CASE input(2 downto 0) IS
  WHEN "000" =>
     output1 <= '1';
     output2 <= '1';
  WHEN "001" =>
     output1 <= '0';
     output2 <= '1';
  WHEN "110" =>
     output1 <= '1';
     output2 <= '0';
  WHEN OTHERS =>
     output1 <= '0';
     output2 <= '0';
END CASE;
```

It differs from most similar constructs in programming languages in that all possible cases must be covered, so it pays to remember that STD_LOGIC signal can have other states than just '1\' or '0\' - most designers choose to use the 'least harmful\' actions on an unexpected value. Like an "IF" statement, "CASE" can only be used inside a process - and remember to include the signals being tested in the process's sensitivity list when a "CASE" is used outside of an "IF RISING_EDGE(clk) THEN" block.

Excellent practice for using the CASE statement is driving the seven segment display - you can use it twice. One CASE statement decodes which segments to light, and a second CASE statement selects which digit is active at any time.

## 14.2 Project - Displaying digits

These projects are a lot of work and might take a couple of sittings, but you will build up a great understanding of the seven segment displays. If you are feeling confident combine a few of the steps and race through.

- Add "anodes(3 downto 0)" and "sevenseg(6 downto 0)" as outputs to your top level design, and then add the following constraints to your ucf file:

```
# Constraints for Papilio One
NET "anodes<0>"   LOC="P18";
NET "anodes<1>"   LOC="P26";
NET "anodes<2>"   LOC="P60";
NET "anodes<3>"   LOC="P67";

NET "segments<6>" LOC="P62";
NET "segments<5>" LOC="P35";
NET "segments<4>" LOC="P33";
NET "segments<3>" LOC="P53";
NET "segments<2>" LOC="P40";
NET "segments<1>" LOC="P65";
NET "segments<0>" LOC="P57";
NET "dp"          LOC="P23";
```

```
# Constraints for the Basys2
NET "sevenseg<0>" LOC = "L14";
NET "sevenseg<1>" LOC = "H12";
NET "sevenseg<2>" LOC = "N14";
NET "sevenseg<3>" LOC = "N11";
NET "sevenseg<4>" LOC = "P12";
NET "sevenseg<5>" LOC = "L13";
NET "sevenseg<6>" LOC = "M12";
NET "dp" LOC = "N13";

NET "anodes<3>" LOC = "K14";
NET "anodes<2>" LOC = "M13";
NET "anodes<1>" LOC = "J12";
NET "anodes<0>" LOC = "F12";
```

- In your top level design, connect the output to *sevenseg* and *dp* to the switches. within the design set anodes to "1110" then build the design. (The anodes are "active low", so this value should enable only the rightmost digit of the sevenseg displays).
- Work out and document the the switch patterns required to give the digits 0 through 9, and the letters A through F.
- Build a *CASE* statement to decode the binary of switches(3 downto 0) and display it on the first seven segment display - remember that at least switches(3 downto 0) has to be included in the sensitivity list of the process acting on them, as there is no clock being used.

## 14.3 Multiplexing digits

If each digit is displayed quick succession the eye can be fooled into seeing all four displays as being lit at the same time. As we have four digits we can use to two bits of a suitably sized counter to select which is to be lit. If if the design switches digits too fast it will not give them enough time to light up, and too slow will call flickering. Something around 200Hz to 1kHz seems to work best.

| counter bits | value for anodes | values for sevenseg() |
|---|---|---|
| 00 | 1110 | Digit 0 |
| 01 | 1101 | Digit 1 |
| 10 | 1011 | Digit 2 |
| 11 | 0111 | Digit 3 |

You can either decide to decode the four digits in each option of the CASE statement (using nested CASE statements), or maybe

create a signal "thisdigit : STD_LOGIC_VECTOR(3 downto 0)" with the digit to be decoded within the case, and then just decode that signal.

## 14.4 Project - Using the Seven segments

- Update your project to multiplex all four display and showing the values of switches(3 downto 0) on all digits
- Update your project to multiplex all four display and showing the values of switches(3 downto 0) on digit 0 and 1, and the value of switches(7 downto 4) on digit 2 and 3
- Update your project show the highest 16 bits of a counter over all four digits.
- Create a new module that can display four digits on the seven segment display. This will be useful for any project you design that uses the sevenseg displays. Its interface signals should something like:

```
clk      : in  std_logic
digit0   : in  std_logic_vector(3 downto 0)
digit1   : in  std_logic_vector(3 downto 0)
digit2   : in  std_logic_vector(3 downto 0)
digit3   : in  std_logic_vector(3 downto 0)
anodes   : out std_logic_vector(3 downto 0)
sevenseg : out std_logic_vector(6 downto 0)
dp       : out std_logic
```

## 14.5 Challenges

- Can you make the display count only in decimal rather than hexadecimal?
- Can you make the display count in minutes and seconds?

# Chapter 15

# Using the FPGA's internal RAM

As well as the resources required for implementing digital logic, FPGAs also have a small amount of RAM built in. This RAM is very useful and can meet the entire RAM needs of many projects.

Each Vendor's RAM blocks have differing capabilities and is configured differently, so it makes sense to use this as a way of introducing the IP Core Generator.

This project is very "GUI" based - unlike the last module it is very much a walk through.

## 15.1 What is Block RAM? What can it do?

On the Spartan 3E each RAM block has 18 kilobits of RAM, and can be presented to the system in different widths. Eighteen kilobits is an odd size, but it is designed that way to allow for either parity or ECC bits to be stored.

The most common configuration I've used is 2048 words of 8 bits, but it can be configured as one of either 16k x 1bit, 8k x 2 bits, 4k x 4 bits, 2k x 8 bits, 2k x 9 bits, 1k x 16 bits, 1k x 18 bits, 512 x 32 bits, 512 x 36 bits or 256 x 72 bits.

The blocks are especially useful as they are dual-port - there are two independent address, read and write ports that simplifies many designs (such as building FIFOs).

See http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf for complete documentation and some very cunning uses for BRAM.

## 15.2 Using the Core Generator with BRAM

Using the Core generator makes build BRAM components very simple - and if required it also transparently constructs larger memories out of multiple primitives. In the project you will use the core generator as creating them directly in VHDL is quite cumbersome and complex.

## 15.3 Preparing the project

- Create a new project - I called mine "flashylights".
- Add a module which has the clock signal as the only input and the eight LEDs as the output

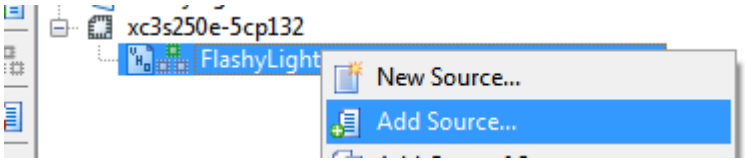
You should get a module that looks like this:

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

entity FlashyLights is
    Port ( clk : in  STD_LOGIC;
           LEDs : out  STD_LOGIC_VECTOR (7 downto 0));
end FlashyLights;

architecture Behavioral of FlashyLights is
begin

end Behavioral;
```

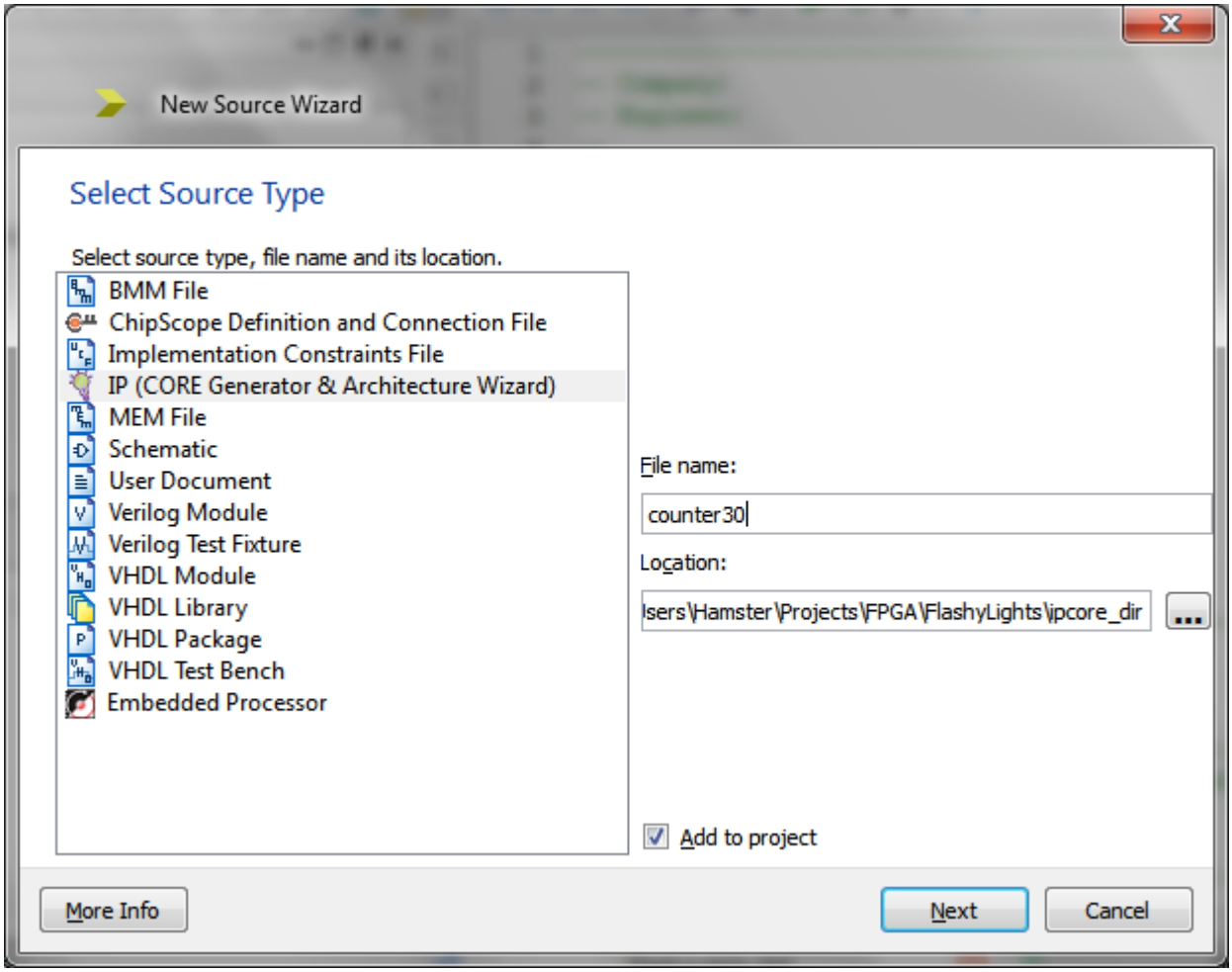
We now need to add a couple of wizard generated components.

## 15.4 Using the IP core generator

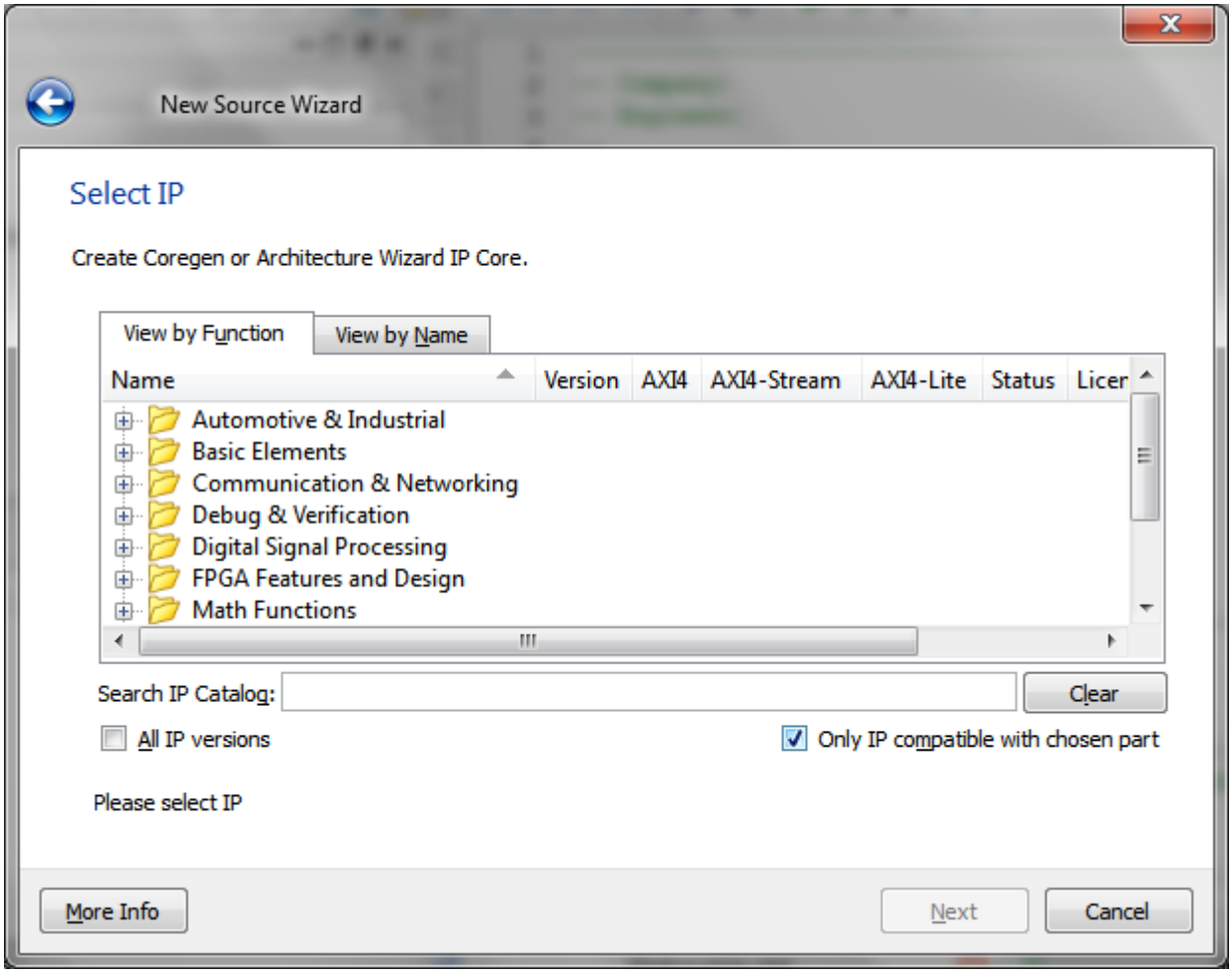
Add a new source file to the project:



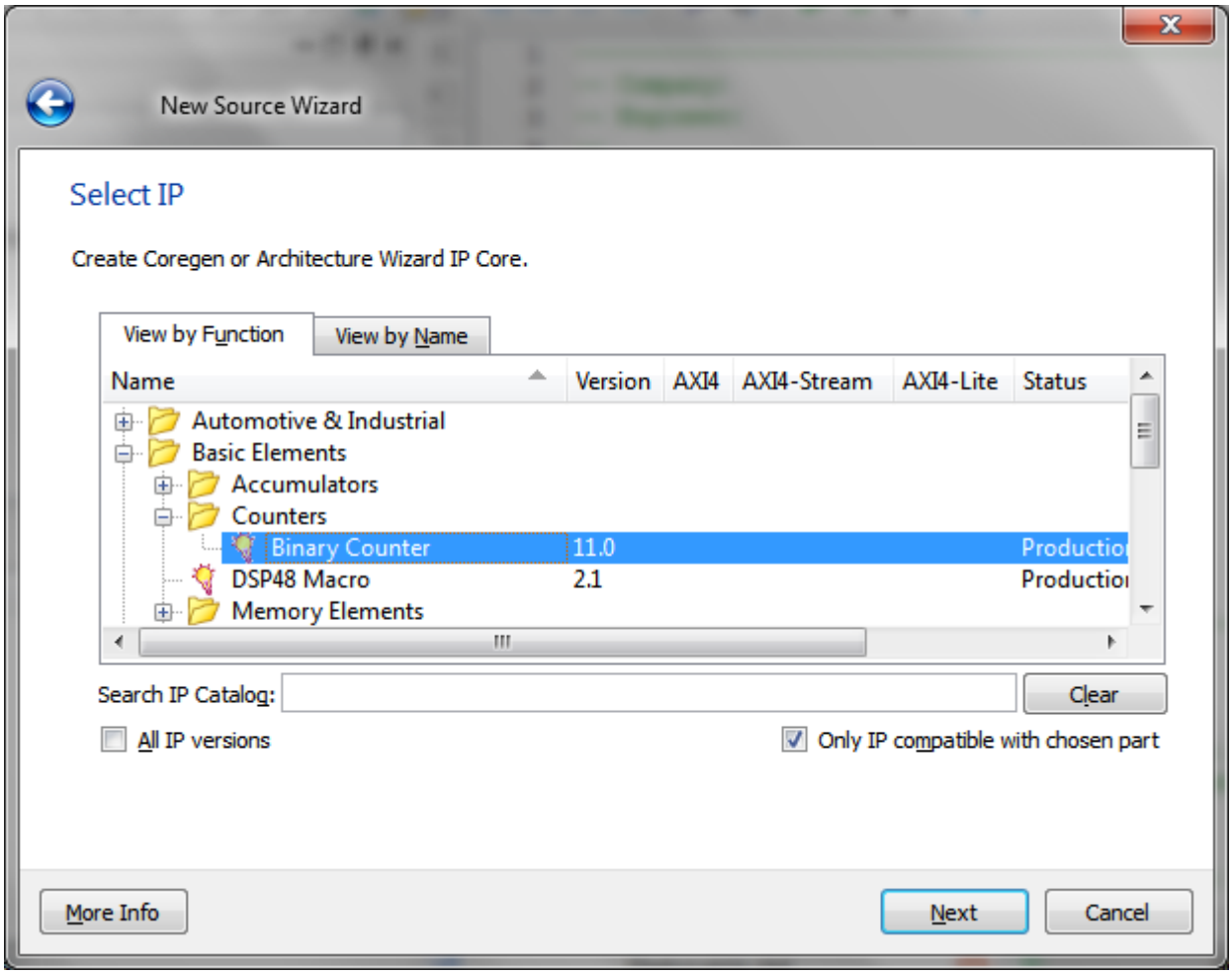

Select "IP" and call the module counter30 - it will be a 30 bit counter



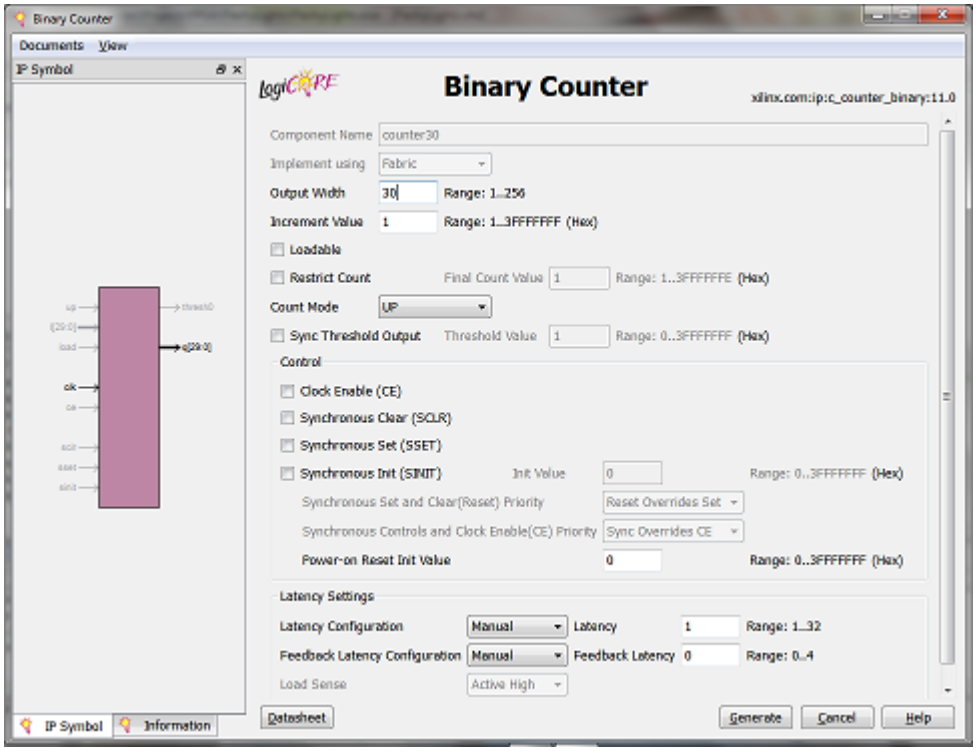
You will be presented with the "Select IP" dialogue box. Tick the "Only IP compatible with chosen part" tickbox:




Navigate down into "Basic Elements"/"Binary Counter" and click next

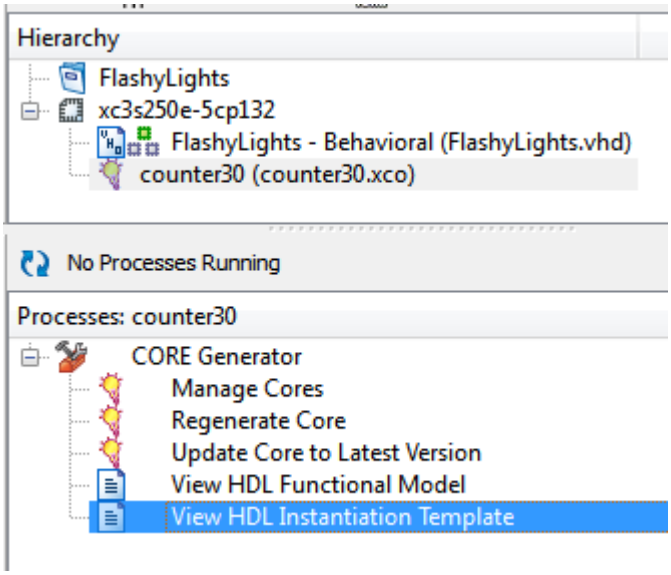



After a long delay the options for Binary Counter will appear. Set the "Output Width" to 30 - and if you want, click on the "Datasheet" button:



Then click "Generate".

In the Hierarchy window you will now have a "counter30" component. Click on it and then under the Processes tree select "View HDL Instantiation Template":

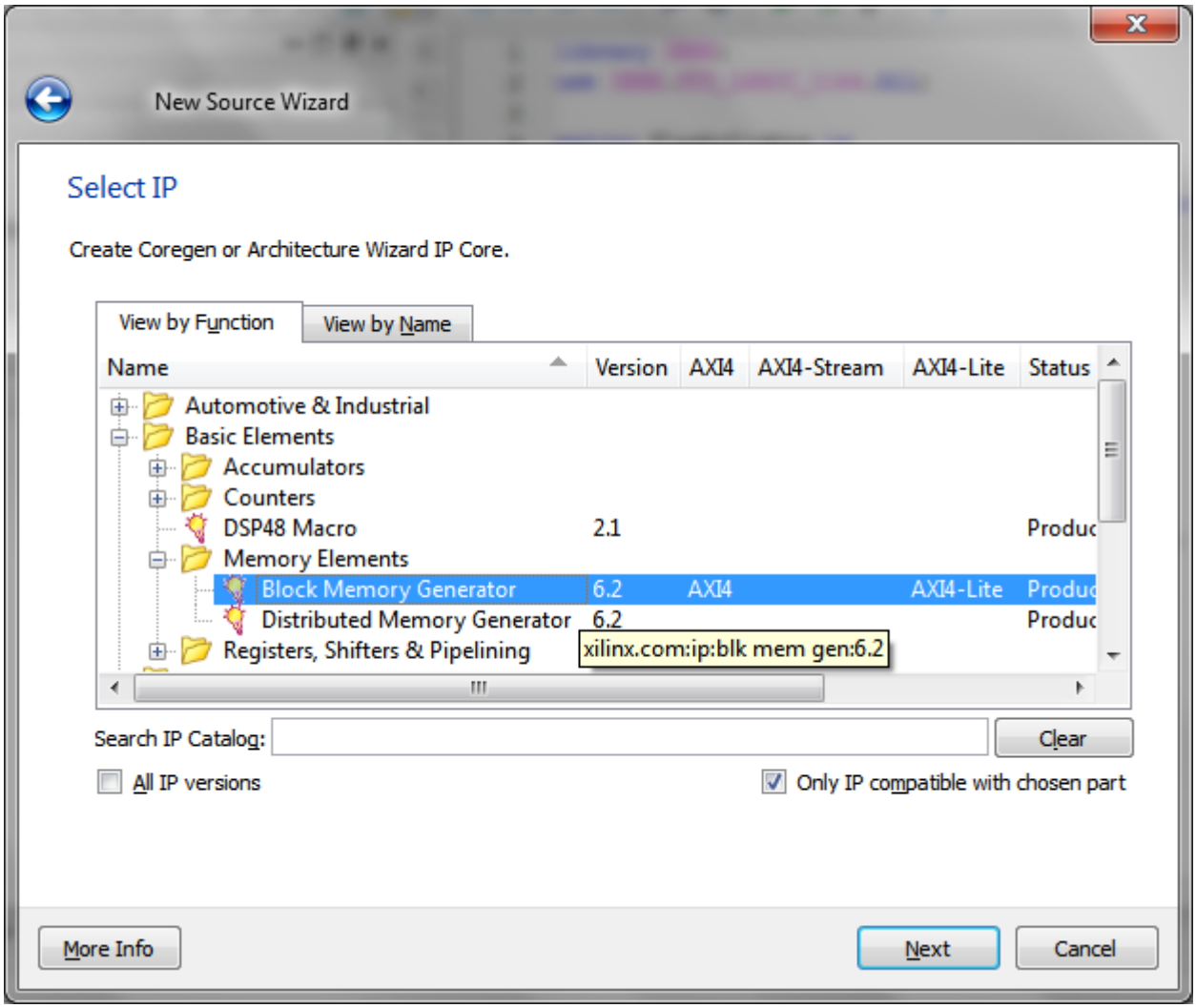



Copy and paste the userful bits into your top level project - add a signal "counter" to be connected to the output of the counter. Here's the completed source:

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

entity FlashyLights is
    Port ( clk : in  STD_LOGIC;
           LEDs : out  STD_LOGIC_VECTOR (7 downto 0));
end FlashyLights;

architecture Behavioral of FlashyLights is
  COMPONENT counter30
    PORT (
      clk : IN STD_LOGIC;
      q : OUT STD_LOGIC_VECTOR(29 DOWNTO 0)
    );
  END COMPONENT;

  signal count : STD_LOGIC_VECTOR(30 downto 0);
begin

addr_counter : counter30
  PORT MAP (
    clk => clk,
    q => count
  );

end Behavioral;
```

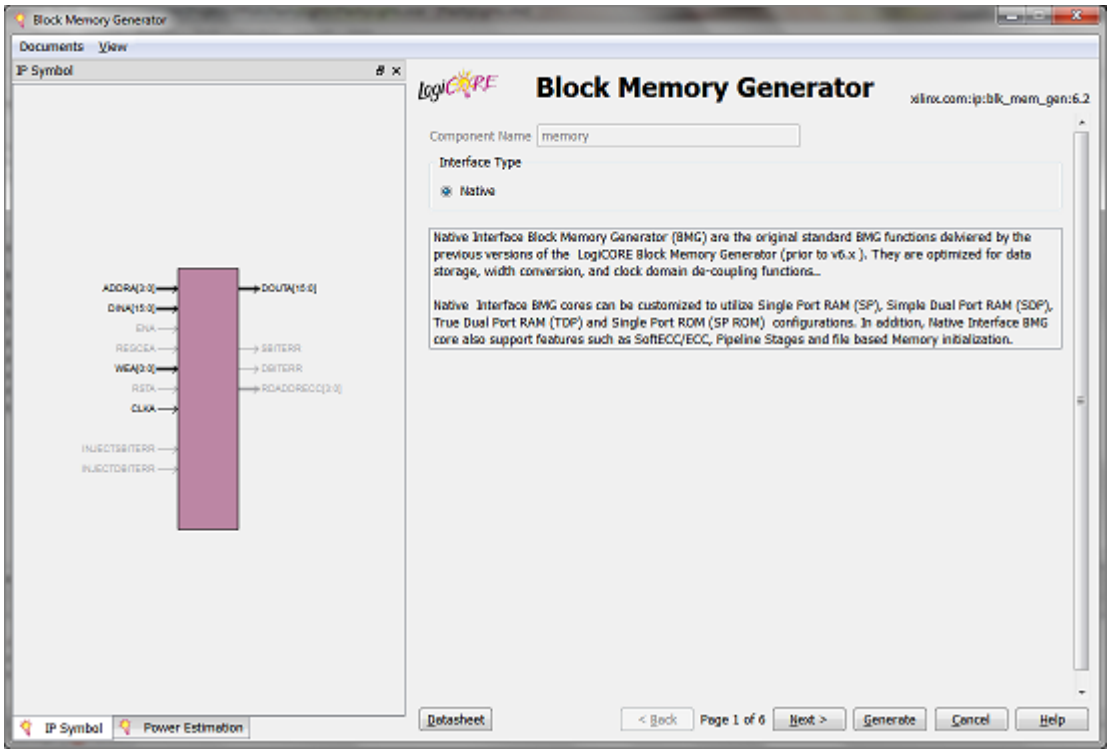
## 15.5 Adding the ROM component

Add another new IP module called "memory", but this time select the Block Memory Generator:

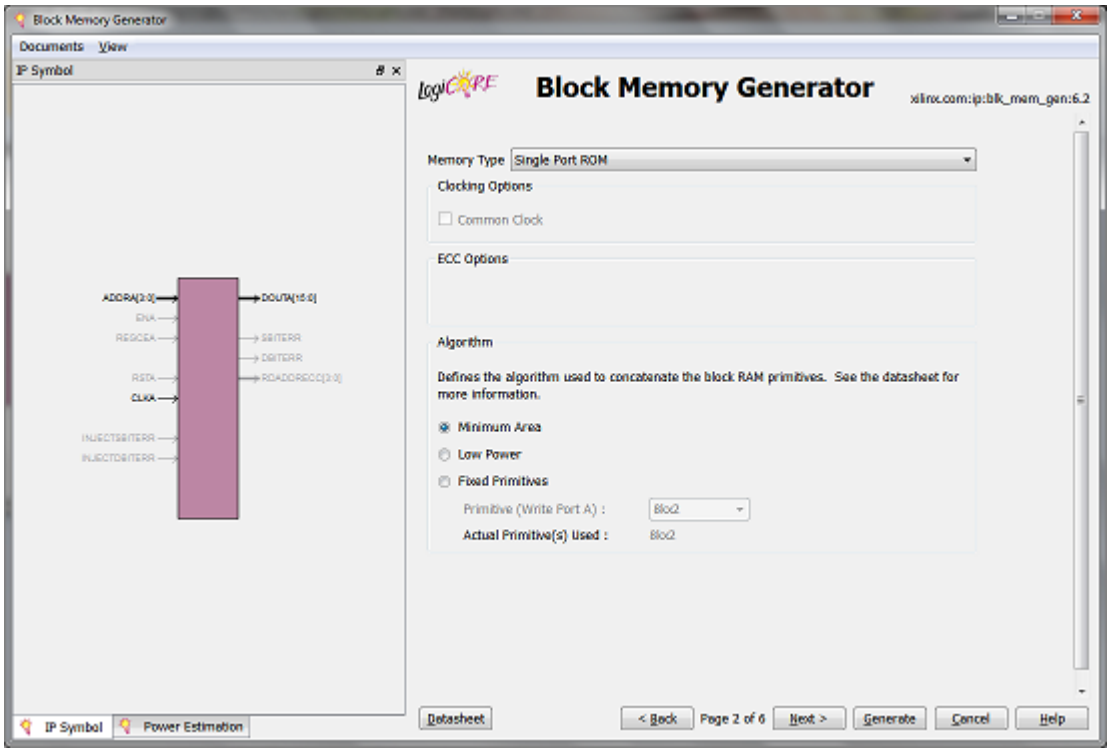



The block memory generator has 6 pages of settings - at the moment we only need to enter things on the first three.

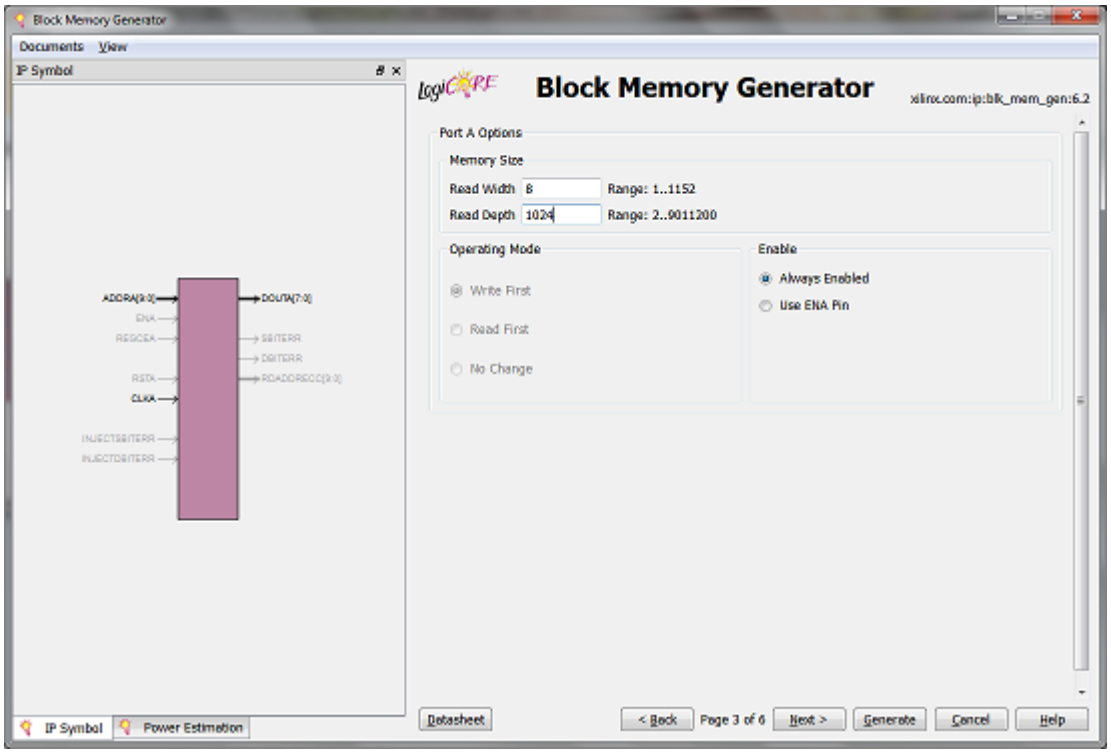
Just click "next" on the first screen:




Select that we want a single port ROM, then click "Next":

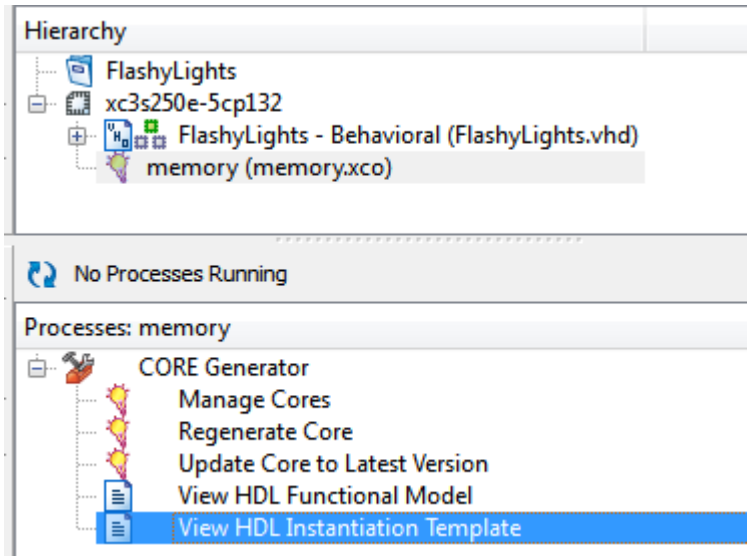



Set "Read Width" to 8 - we have eight LEDs to light. Set the "Read Depth" to 1024. Click "Next":




Don’t bother going through the rest of the screens - they don’t apply at the moment - just click "Generate"

You will now have another component, and you can view it’s instantiation template.

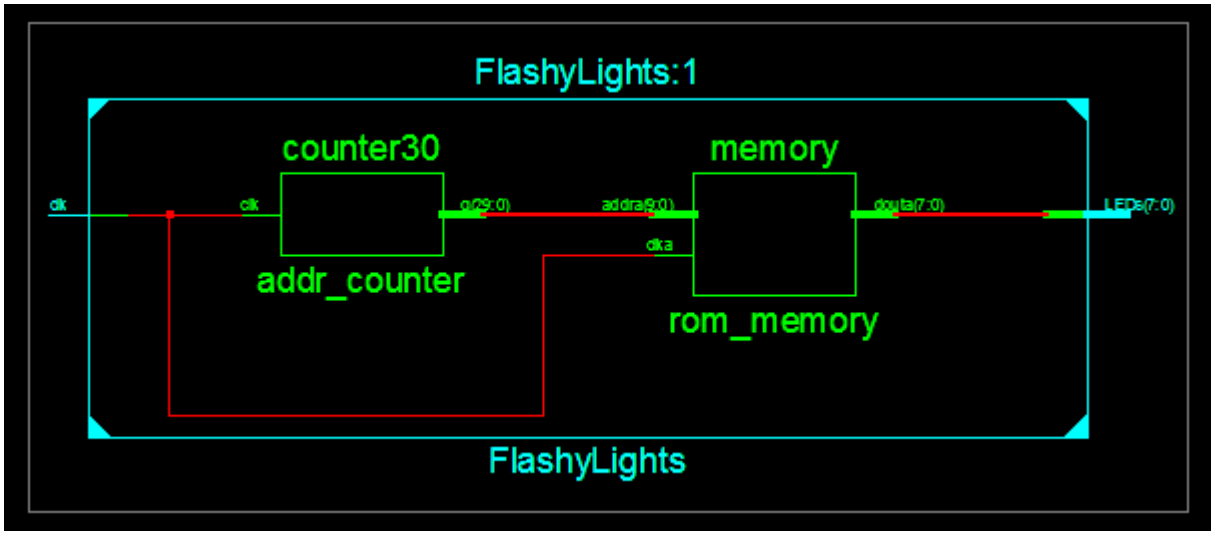



Add it to the source, connecting the top 10 bits of the counter to the ROM’s address bus (addra), and the data bus (douta) to the LEDs:

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

entity FlashyLights is
    Port ( clk : in  STD_LOGIC;
           LEDs : out  STD_LOGIC_VECTOR (7 downto 0));
end FlashyLights;

architecture Behavioral of FlashyLights is
  COMPONENT counter30
```

```
    PORT (
      clk : IN STD_LOGIC;
      q : OUT STD_LOGIC_VECTOR(29 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT memory
  PORT (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
  END COMPONENT;

  signal count : STD_LOGIC_VECTOR(29 downto 0);
begin

addr_counter : counter30
  PORT MAP (
    clk => clk,
    q => count
  );

rom_memory: memory
  PORT MAP (
    clka => clk,
    addra => count(29 downto 20),
    douta => LEDs
  );
end Behavioral;
```

Once built, you can view the RTL schematic - looks as you would expect:




## 15.6 Setting the contents of the ROM

At the moment the ROM is blank (all '0's). When the FPGA is configured the contents of the block RAM can be set to values that are predefined in the configuration bit stream.

Page 4 of the block memory generator gives you the option to set the contents of the ROM using a ".coe" file. Here's enough of the file that you will be able to write your own from scratch:

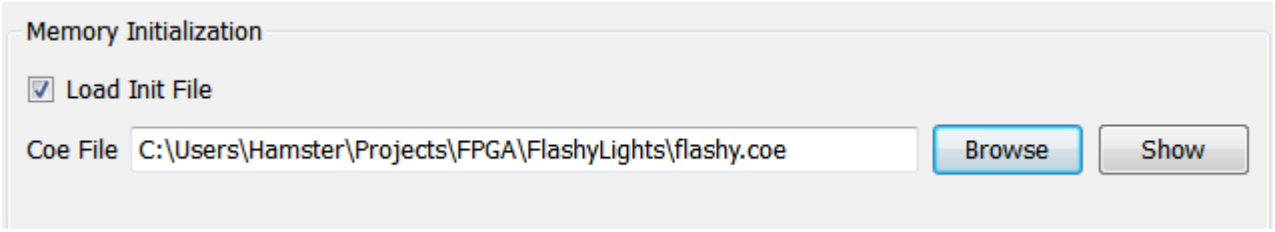
```
memory_initialization_radix=10;
memory_initialization_vector=
128,
128,
127,
127,
127,
```

Here's another, using binary (as memory_initialization_radix=2) for a memory with a data width of 15:

```
memory_initialization_radix=2;
memory_initialization_vector=
001110000000001,
010110000000010,
000010000000011,
000010000000100,
000010000000101,
000010000000110,
```

Create a sample file of 8 bit binary values - make the *1* bits zig-zag from left to right, or some other pattern - the more lines the merrier. Call it "flashy.coe".

Edit the "memory" component (just double click it in the Hierarchy tree) and skip through to Page 4. Set the initialisation file to flashy.coe.




It is always a good idea to click on the "show" button - it will give you a warning if your *.coe* file is not correct. Click the *Generate* button to update the IP module.

As an aside there are other ways to do this, allowing you to inject contents (e.g. maybe bootloader) after the *.bit* file is built. This allows you to avoid a length rebuild of a whole project just to change the initial values in a BRAM. It is also a good way to allow an end-user to customise the *.bit* file without providing access to your source code. Seach for "xilinx data2mem" in Google.

## 15.7 The finishing touches

```
NET LEDs(7) LOC = "P5"  | IOSTANDARD=LVCMOS25;
NET LEDs(6) LOC = "P9"  | IOSTANDARD=LVCMOS25;
NET LEDs(5) LOC = "P10" | IOSTANDARD=LVCMOS25;
NET LEDs(4) LOC = "P11" | IOSTANDARD=LVCMOS25;
NET LEDs(3) LOC = "P12" | IOSTANDARD=LVCMOS25;
NET LEDs(2) LOC = "P15" | IOSTANDARD=LVCMOS25;
NET LEDs(1) LOC = "P16" | IOSTANDARD=LVCMOS25;
NET LEDs(0) LOC = "P17" | IOSTANDARD=LVCMOS25;

NET "clk" LOC="P89" | IOSTANDARD=LVCMOS25 | PERIOD=31.25ns;
```

Rebuild the project, download it and watch the lights!

# Chapter 16

# Generating analogue signals

One of the nice features of FPGAs is how versitile the I/O pins are. I this chapter we will make a standard I/O pin generate an analogue signal, playing a tone using a waveform that is stored in block RAM

This module is largely based on Xilinx's AppNote xapp154.pdf

## 16.1 One bit (Delta Sigma) DAC

You are most probably familiar with Pulse Width Modulation (PWM) - It is when a signal of a constant frequency has its duty cycle modulated to generate different power levels. If a PWM signal is passed through a low pass filter you end up with an analogue voltage that is proportional to the duty cycle. PWM is used in power supplies, light dimmers and motor controllers and such.

Delta Sigma modulation is a lot like that, but without the constant frequency of PWM. It has a output that *hunts* for the desired output value. A one bit DAC has only two output values (1 or 0), and it generates the value which when included in a running average brings it closest to the desired level:

- To generate a level of 0.5 the output will be "10101010101…"
- To generate 0.25 the output will be "000100010001…."
- To generate 0.66 the output will be "110110110110110".

All of these signals average out to the desired value but have different frequencies.

## 16.2 Um, that looks really hard to do

It's not that hard at all. For this example work in decimal to make it clearer, but implementation in binary is just the same.

To make a Delta Sigma DAC with 100 output levels you need an accumulator with two decimal digits, and you use the "carry to the hundreds" as the output. Just keep adding the desired output level to the two digits and the "carry to the hundreds" will be a stream of ones and zeros that averages to the desired level.

Here's a two decimal digit DAC generating the output of 33:

| Iteration | Digits | Carry/Output |
|---|---|---|
| 0 | 50 | 0 |
| 1 | 83 | 0 |
| 2 | 16 | 1 |
| 3 | 49 | 0 |
| 4 | 82 | 0 |

| | | |
|---|---|---|
| 5 | 15 | 1 |
| 6 | 48 | 0 |
| 7 | 81 | 0 |

Pretty simple!

Of course there are a few little tricks:

- Do it quick enough so that at the highest required frequency you have enough *1's and '0* to average over
- Careful design of an analogue output filter is required for best performance
- Do not use all the DAC's range, as the spectrum of noise at either end is problematic

## 16.3 Rough back-of-the-envelope bandwidth and effective resolution calculation

If you need to produce signals at 22kHz have to use at least a 44kHz playback frequency. If the one-bit DAC runs at 25MHz there is a little of a five hundred output values (ones and zeros) to average over in 1/44000th of a second - giving you at best 9 bit resolution at that frequency.

## 16.4 Doing it in VHDL

Here is the code for an 8 bit DAC - It is pretty much a "count by *n*" counter:

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

entity dac8 is
   Port ( Clk : in  STD_LOGIC;
          Data : in  STD_LOGIC_VECTOR (7 downto 0);
          PulseStream : out  STD_LOGIC);
end dac8;

architecture Behavioral of dac8 is
   signal sum : STD_LOGIC_VECTOR (8 downto 0);
begin
   PulseStream <= sum(8);

  process (clk, sum)
  begin
     if rising_edge(Clk) then
        sum <= ("0" & sum(7 downto 0)) + ("0" &data);
     end if;
  end process;
end Behavioral;
```

## 16.5 Connecting up the headphones

On the Papilio One, just plug amplified speakers into the socket and use the following constraint. You need t use amplified speakers as the output impedance is around 3.3k ohm.

**Constraint for the Papilo One**

```
NET "Audio" LOC = "P41";
```

### 16.5.1 Connecting headphones to the Basys2

Unlike the Papilio One + Megawing combo the Basys2 does not have an audio output, so we need to use a PMOD port. The PMODs on the Basys2 board have four signal wires from the FPGA, a ground and a 3.3V power connection. For the JA header on the BASYS2 board the constraints are:

**Contraints for the Basys2**

```
NET "JA<0>" LOC = "B2";
NET "JA<1>" LOC = "A3";
NET "JA<2>" LOC = "J3";
NET "JA<3>" LOC = "B5";
```

**Caution**
Make sure that you don't short the power pins. Shorting out ground and power will upset your USB port and/or your FPGA board

For this project connect a set of stereo earphones between pins 0 and pins 1 and the ground. To do this I used a header strip, 3.5mm socket and a length of wire:

If you pull the unused pins out of the header strip you might just be able to hold the 3.5mm jack in place at the correct time…

The inductive nature of the headphones/earphones proves to be a pretty good low pass filter for the high frequency signals so no additional components are needed - but if you want to you can include a suitable capacitor in series to prevent average DC voltage running through them.

[INFO] The Basys2 board have a 200 ohm resister in series with the FPGA output pin. This makes the PMOD connectors somewhat protected against ESD, overvoltage and shorts. For this project it it also acts as a voltage divider reducing the DC bias and the peak to peak voltages that go through the headphones/earphones.

## 16.6 Project - Wave file generation

In the prior project we hooked a block RAM to the LEDs, and used it to flash them. We can do the same to generate an audio waveform.

- Make a COE file containing the samples for a sine wave (something like "f(n) = int((sine(n*PI()/1024)+1)*100)+128" will give you values between 28 and 228 that you can use).

- Load it into the flashylighs project and check that the lights look OK.
- To generate an audible tone we need to cycle through this somewhere around 400 times per second - so we need to use counter(15 downto 6) to address the ROM component. This should generate a tone of one cycle every 65536 clocks = 381.4Hz
- Add a 8 bit DAC to your project and connect it to the audio output. Remember to add the appropriate constraints to your project!
- Build and download the design. If you connect your headphones you should have a tone!

## 16.7 Challenges

- At the moment we can only generate one frequency. Design and try out ways to make different frequencies.
- The Spartan 3E-250 has 24K of on-chip memory. That's enough for 2 seconds of telephone quality 11kHz/8 bit audio....
- If you connect the two high address bits on RAM to switches you can have four different waveforms, each with 256 samples per cycle. e.g Square, Saw, Ramp, Sine.
- By right-shifting the samples you can control the volume - and with a *wider* DAC you can keep the least significant bits. Remember to *sign-extend* the sample when you shift it (e.g. y(8 downto 0) = x(7) & x(7 downto 0)).
- The design is quite lo-fi - very 8 bit!, Make the DAC into 16 bits, and changing the ROM to have a data width of 16 (you will also need a new *.coe* file with samples expanded out to match the range of the 16 bit values).

# Chapter 17

# Implementing Finite State Machines

Up to now the projects have been very *linear* - mostly counters that work like clockwork. Now we are going to investigate how you can get your logic to allow external signals change it's behaviour, rather than just processing the results. The technique introduced is used in many different areas of a design, such as

- Communication protocols, where data may be sent asynchronously or different data required different responses
- Scheduling of control signals in a memory controller
- Decoding and executing instructions in a CPU
- Control of simple machine
- Implementing simple user interfaces.

## 17.1 Introduction to the project

For the project we are going to build a combination lock, which works as follows:

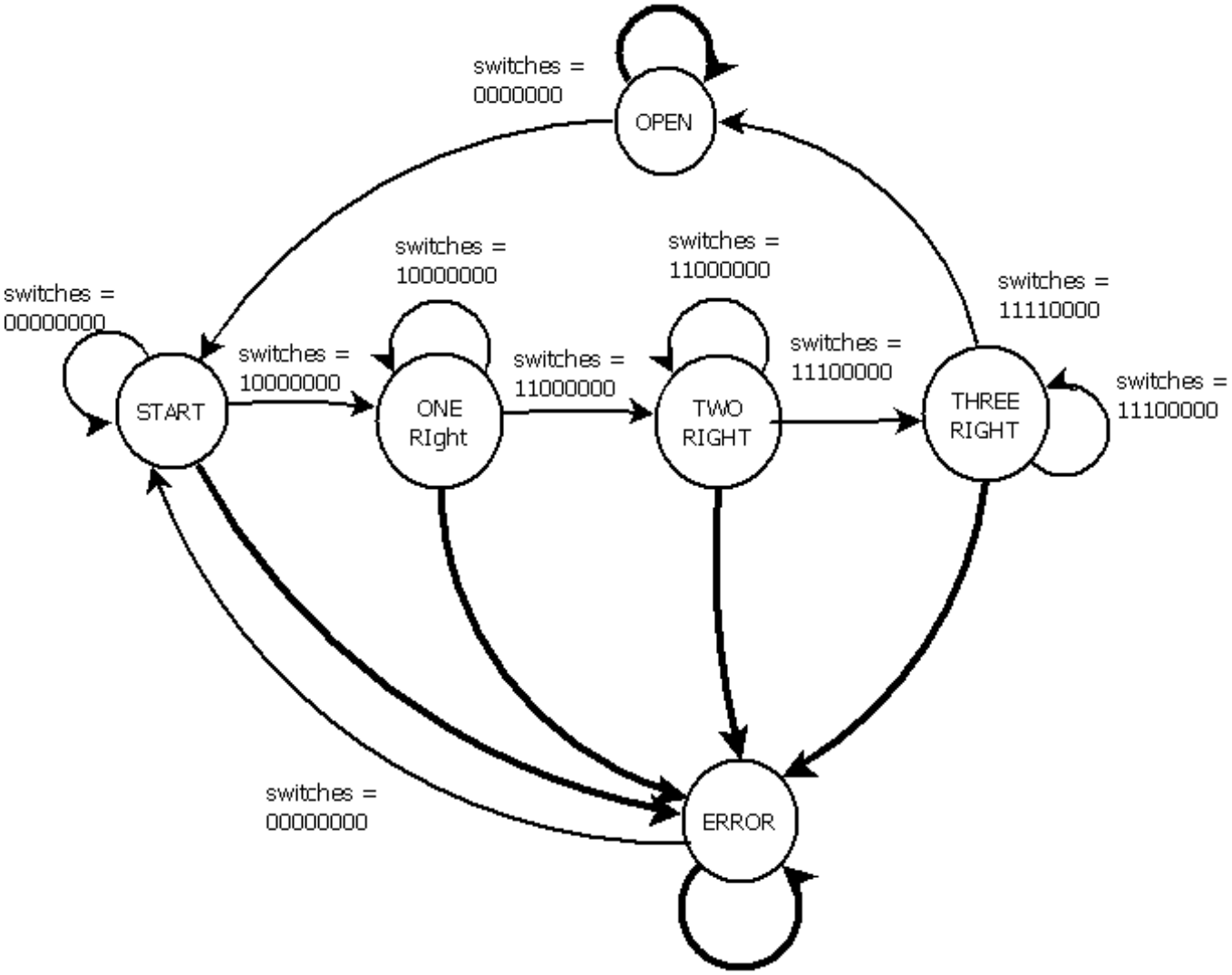
- All switches must be turned off.
- then switch 7 must be turned on
- Then switch 6 must be turned on
- then Switch 5 must be turned on
- then Switch 4 must be turned on

If this sequence is followed, all the LEDs will turn on and stay on until all switches are moved back to off.

In software this would be quite easy - using the console for user input something like this would be quite close :

```
while(1)
{
 if(getchar() = '7' && getchar() = '6' && getchar() = '5' && getchar() = '4')
 {
   LEDs = 0xFF;
   getchar();
 }
}
```

To get the same result, we need to use a finite state machine (FSM) - a directed graph of states and how the system moves between the states. There is a formalized way to document FSMs, but here's my somewhat less formal approach which works well when sketching designs on paper.




At any point in time your design is at a state indicated by a circle. On the next clock tick *"it must"* follow an arrow. All options *"must"* be mutually exclusive. I have added bold arrows to indicate the "no other arrow applies" option.

So when the system is in the "START" state the options are either:

- If switches are set to "0000000" we go to the "START" state.
- If switches are set to "1000000" we go to the "ONE RIGHT" state.
- Otherwise we go to the "ERROR" state.

Likewise, in the "ERROR" state the options are

- If switches are set to "0000000" we go to the "START" state.
- Otherwise we go to the "ERROR" state.

If I have designed it correctly the only way to get to the "OPEN" state is to move the switches through "00000000", "10000000", "11000000", "11100000", then finally to "11110000"

## 17.2 Implementing in VHDL

Implementation is relatively easy.

You can either use enumerated types (that have not been covered), but it is usually better to use constants:

```
...
constant state_error     : STD_LOGIC_VECTOR(3 downto 0) := "0000";
constant state_start     : STD_LOGIC_VECTOR(3 downto 0) := "0001";
constant state_one_right : STD_LOGIC_VECTOR(3 downto 0) := "0010";
...
signal state : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
```

If you use constants, then you can encode output signals within the states, ensuring that you get glitch-less signals. If you like, you could include this in your project to help with debugging:

```
leds(3 downto 0) <= state;
```

In your project's process it is usually easiest to code it using a CASE statement like this:

```
if rising_edge(clk) then
   case state is
      when state_error =>
         case switches is
         when "00000000" => state <= state_start;
         when others     => state <= state_error;
         end case;
      when state_start =>
         case switches is
         when "00000000" => state <= state_start;
         when "10000000" => state <= state_one_right;
         when others     => state <= state_error;
         end case;
      when state_one_right =>
         case switches is
         when "10000000" => state <= state_one_right;
         when "11000000" => state <= state_two_right;
         when others     => state <= state_error;
         end case;
      ....
      when others =>
         state <= state_error;
   end case;
end if;
```

## 17.3 Project 14.1 - Combination lock 1

- Code the above FSM to implement the combination lock. To give some feedback on success set LEDs to "11111111", and
- Test it in the simulator, using this in the test bench stimulus process

```
switches <= "00000000";
wait for 200 ns;
switches <= "10000000";
wait for 200 ns;
switches <= "11000000";
wait for 200 ns;
switches <= "11100000";
```

```
 wait for 200 ns;
 switches <= "11110000";
 wait for 1000 ns;
 switches <= "00000000";
```

- Try running it in hardware - it most probably won't work reliably - 50:50 if you are lucky.

## 17.4 The problem with switch bounce

This design will work perfectly well - as long as the switch contacts don't bounce. If they bounce the FSM will view that as an "otherwise" case and go to the error state.

Solutions are:

- Debounce the switches in hardware
- Debounce the switch signals using logic within the FPGA
- Sample the switches at intervals that should mask any bounce (e.g. 1/10th of a second)
- Update the FSM to allow for switch bounces.

The "debounce" solutions are all relatively hard, while updating the FSM will only need a few lines of code.

## 17.5 Project 14.2 - Combination lock 2

- Trace through the FSM diagram to work out why a bounce causes it to fail
- Update the FSM to ignore switch bounces
- If you wish, test it in the simulator - here is stimulus that looks like four bouncing switches:

```
switches <= "00000000";
wait for 200 ns;

switches <= "10000000";
wait for 50 ns;
switches <= "00000000"; -- bounce
wait for 50 ns;
switches <= "10000000";
wait for 300 ns;

switches <= "11000000";
wait for 50 ns;
switches <= "10000000"; -- bounce
wait for 50 ns;
switches <= "11000000";
wait for 300 ns;

switches <= "11100000";
wait for 50 ns;
switches <= "11000000"; -- bounce
wait for 50 ns;
switches <= "11100000";
wait for 300 ns;

switches <= "11110000";
```

```
wait for 50 ns;
switches <= "11100000"; -- bounce
wait for 50 ns;
switches <= "11110000";
wait for 1000 ns;

switches <= "00000000";
```

- Test it in hardware

## 17.6 Challenges

- Can you make the LEDs flash off and on for a few seconds when an error occurs?
- Can you make the board flash the LEDs in a pattern stored in BRAM when it reaches the "OPEN" state?

# Chapter 18

# Using the Digital Clock Manager

One of the other resources on the Spartan 3E FPGA is the Digital Clock Manager. These are very handy!

## 18.1 What are Digital Clock Managers?

DCMs receive an incoming clock and can do the following and more:

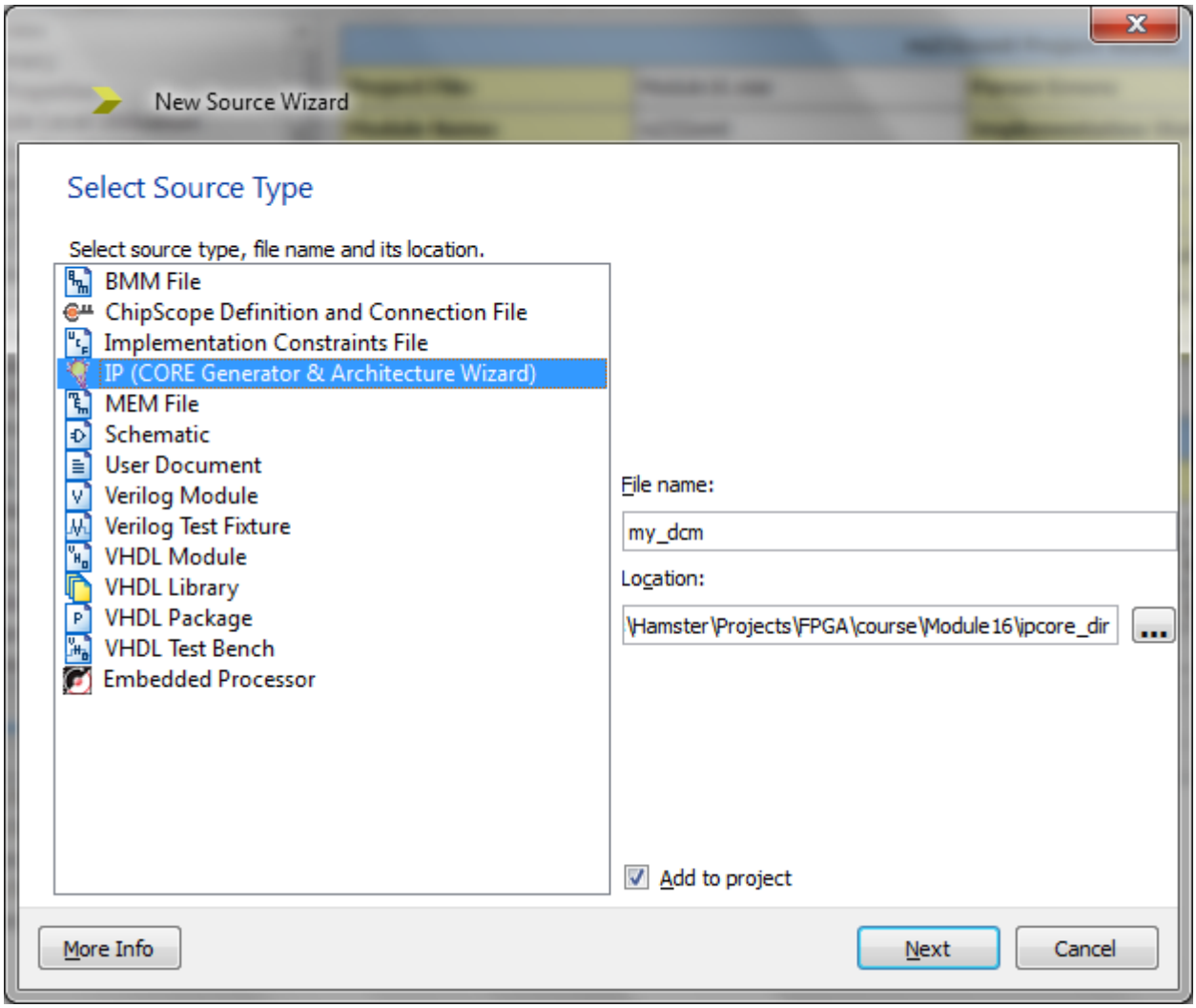
- Generate a faster or slower clock signal using an input clock as a reference
- Generate signals with a known phase shift (e.g. 90, 180 or 270 degrees out of phase)
- Correct clock duty cycles, ensuring that the high and low times are 50%.
- Phase shift the internal FPGA clock signals to compensate for internal clock distribution delays

DCMs can also be cascaded, allowing multiple clocks to be used. For example one external 50MHz clock can be used to generate 100MHz controlling memory and 25MHz for the VGA pixel clock

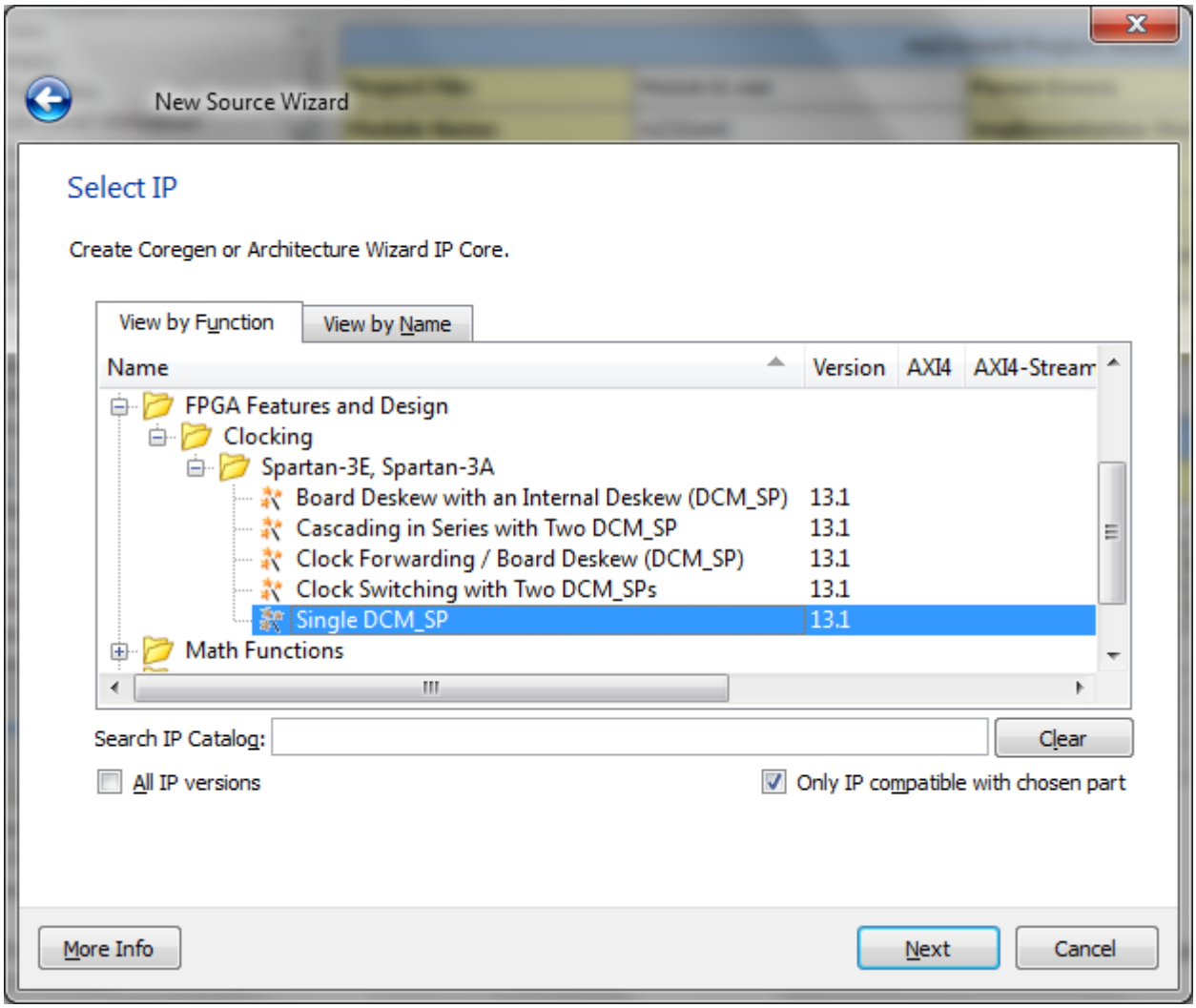
Because of this flexibility they are quite complex to use. I find using the Core Generator is the best way configure a DCM.

## 18.2 Using the Wizard

Pick any project you like, and add a "New Source", using the "IP (Core Generator. . . )" option to create a component "my_dcm":

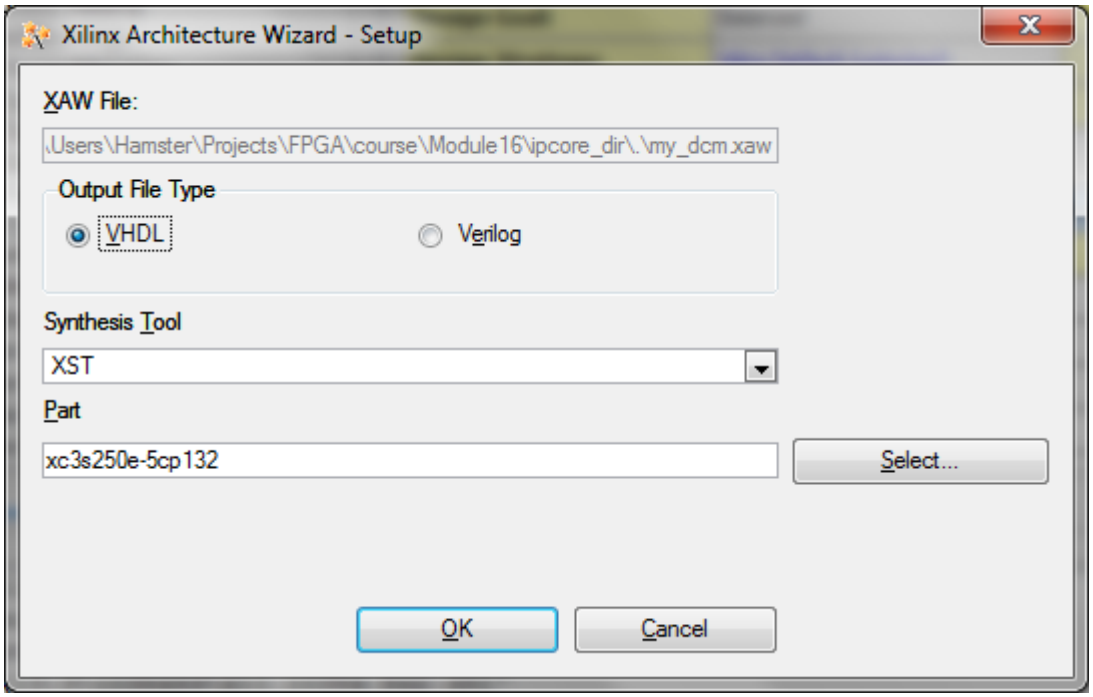



One again choose the "Only IP compatible with chosen part" option, then drill down to "Single DCM_SP":



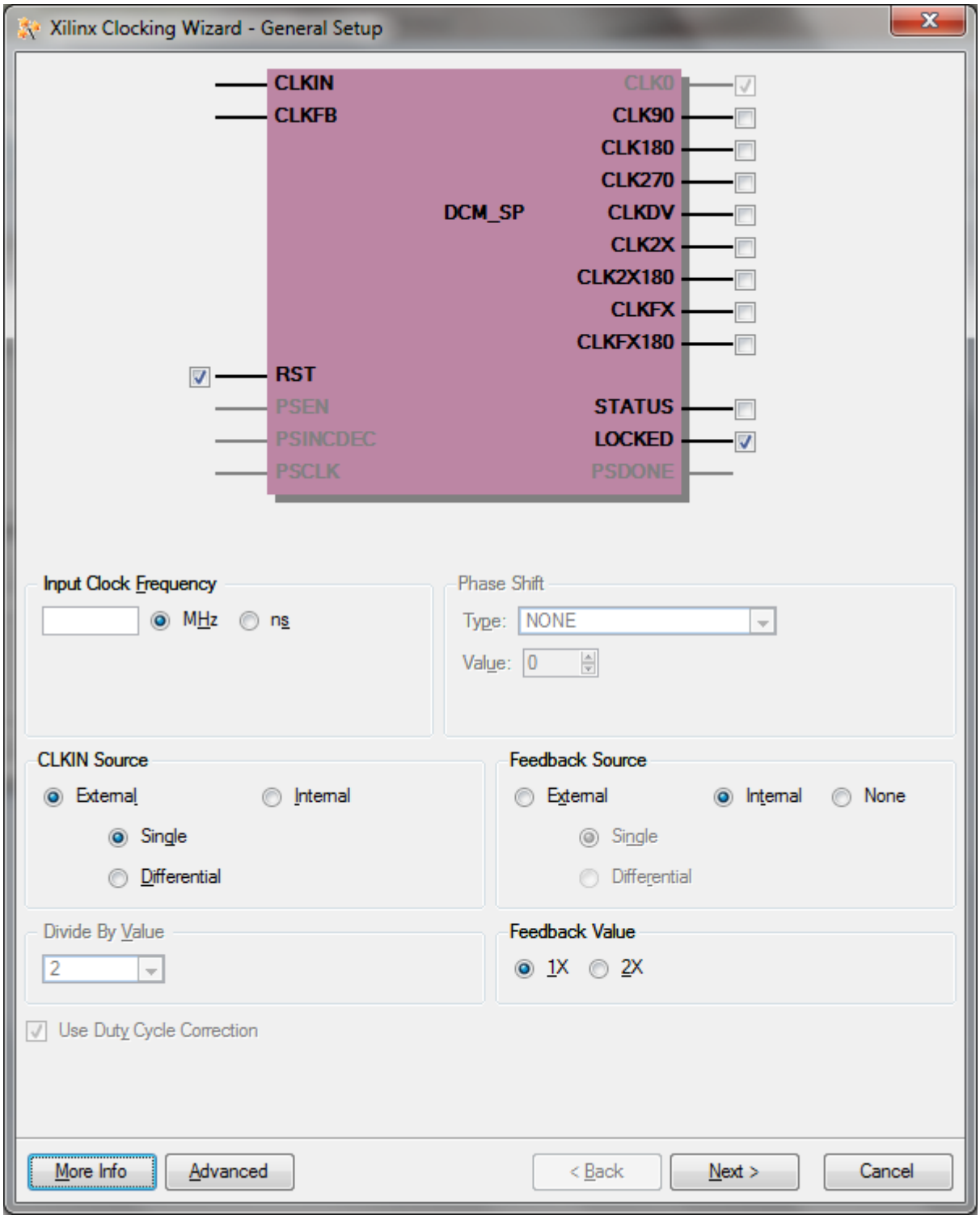

Click "Next" then "Finish" to start the Core Generator.

You will then be presented with this dialog box:

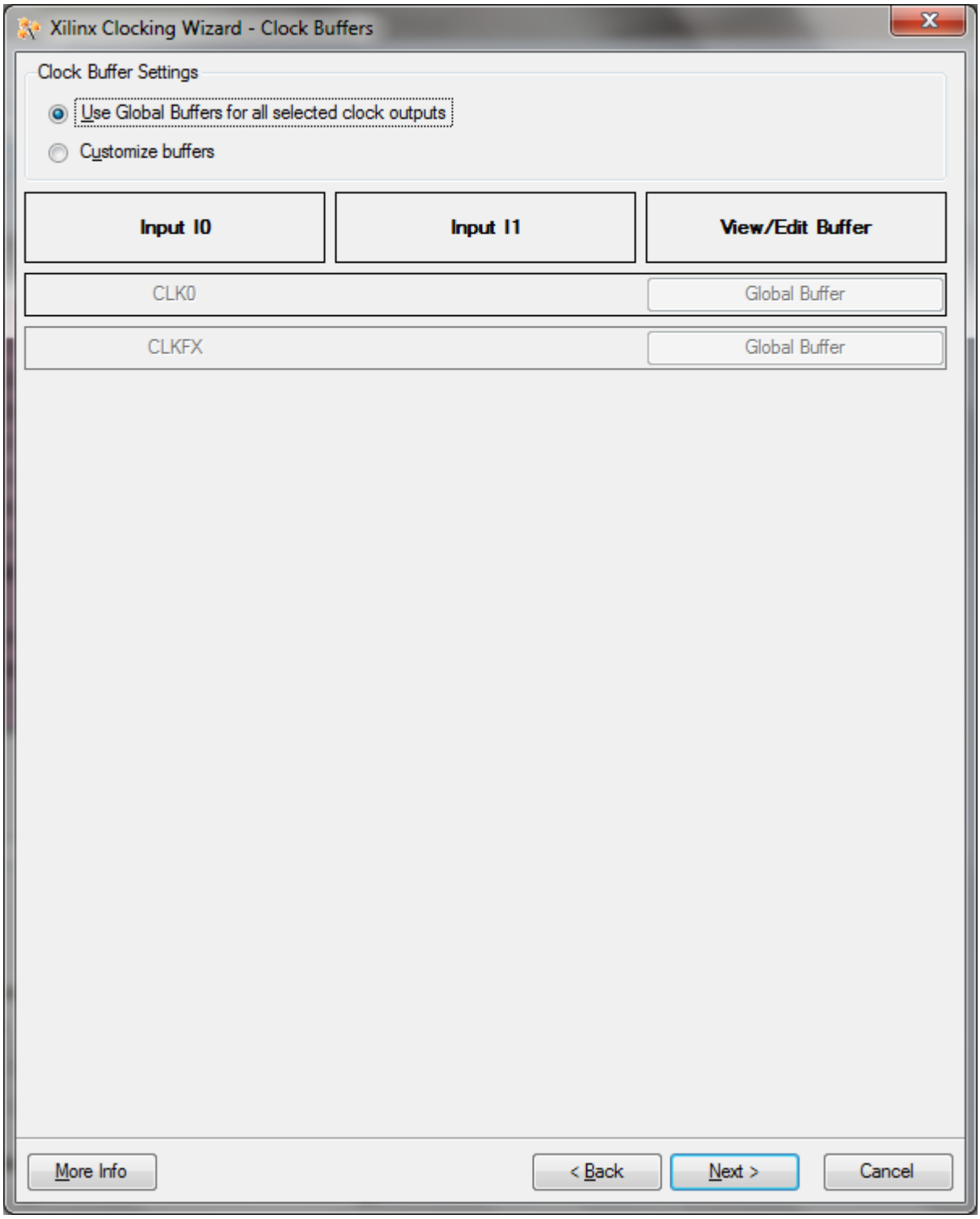



Just click "OK" to open the Clocking Wizard's General Setup dialogue box:

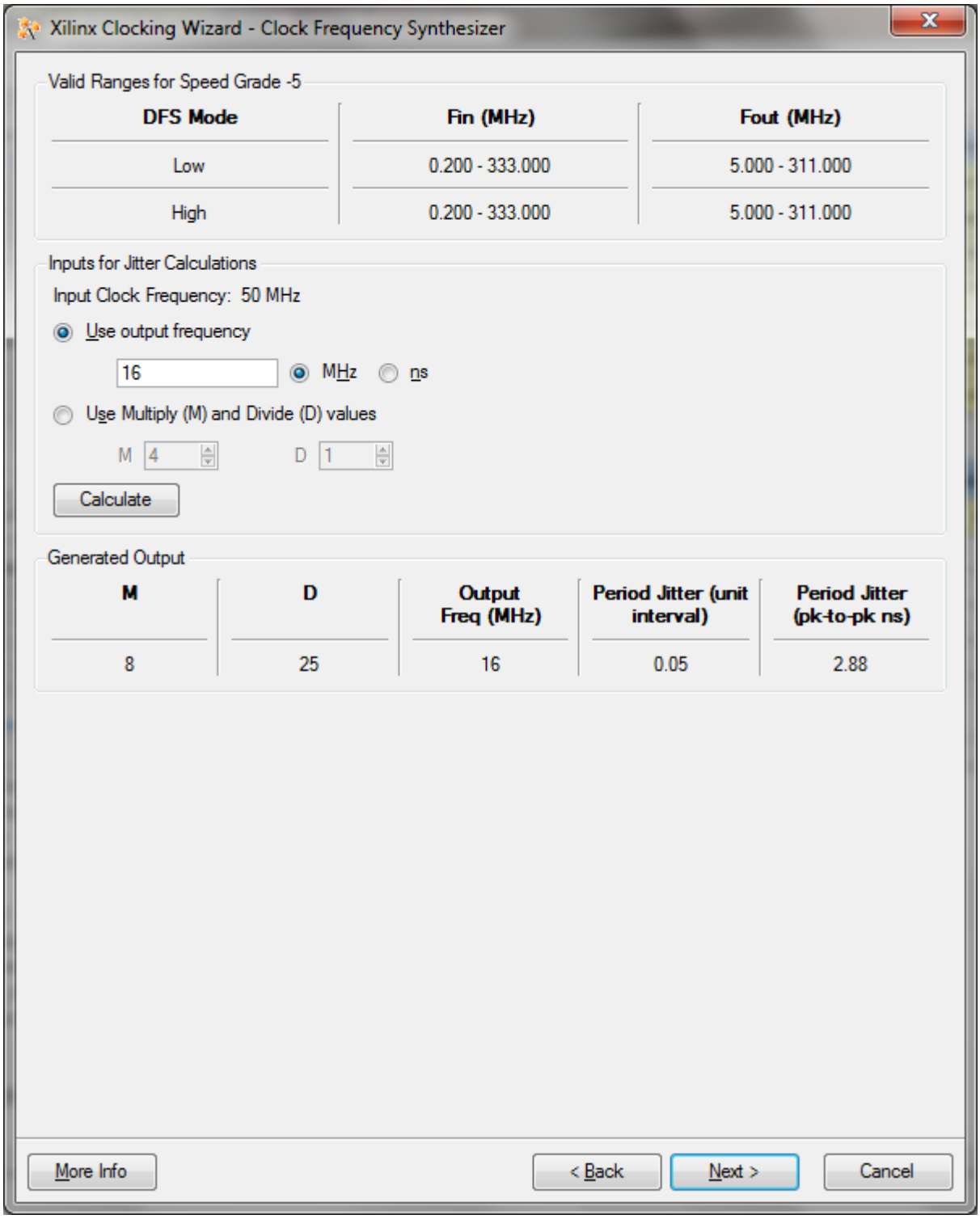
Here you can choose what signals you will use and set the input clock frequency. The most common outputs I use is the CLKFX (which is the synthesized output frequency). You may want to untick the RST (reset) signal if this is the only clock for the entire project:



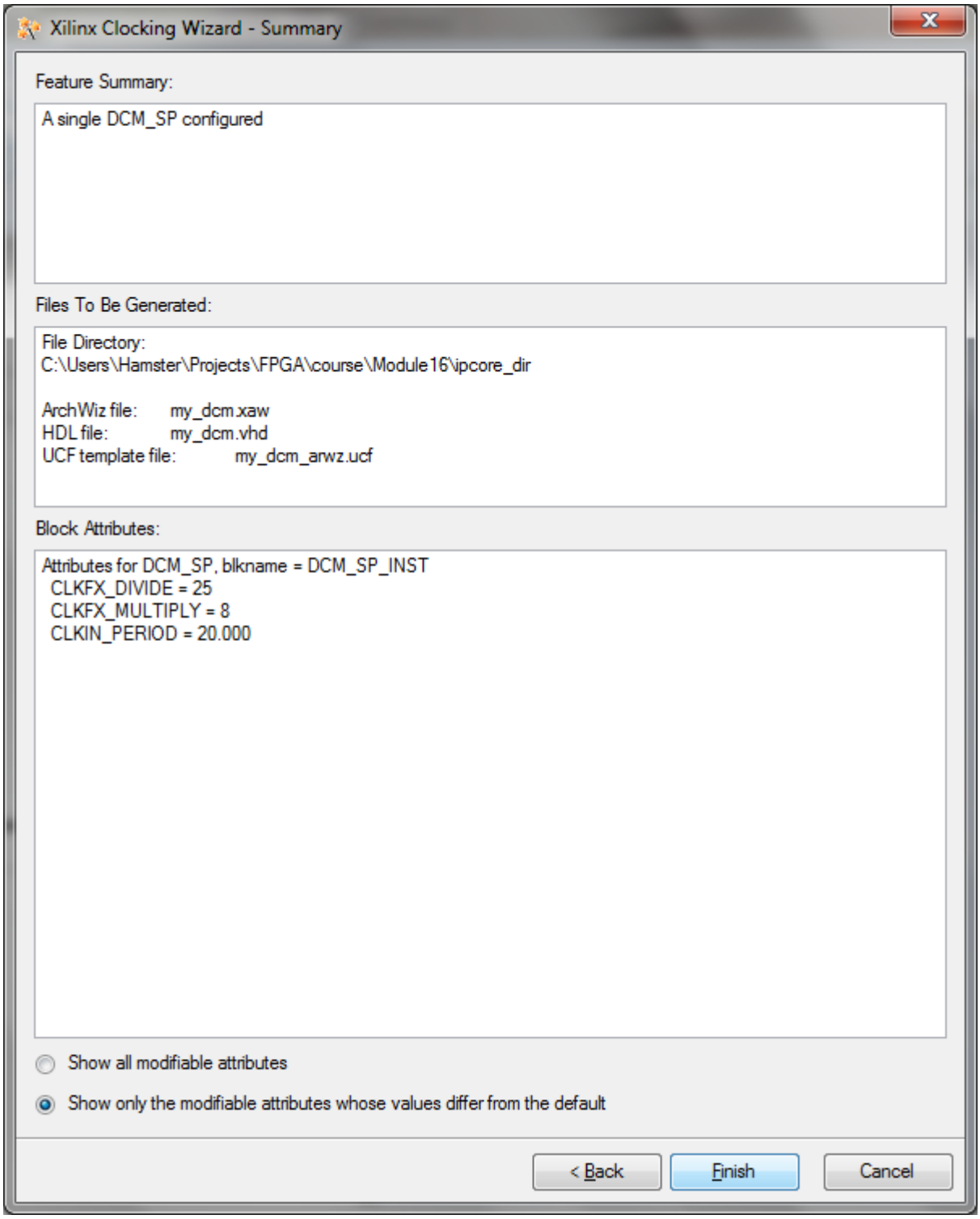

The next screen allows you to choose what clock buffers are being used. For most projects you will use "Global Buffers" - being global the clock signal is available to all logic on the FPGA:




The next screen is the interesting one - it’s where you get to set the output frequency. Input the desired frequency and press "Calculate":




You will now get the summary screen, where you can click "Finish":




Once generated you will be able to use the instantiation templates to add a "my_dcm" component to your project.

## 18.3 Project - Use a DCM

- Add a DCM to one of your projects (e.g. project 4.1).

**Note**
Remember to update not only the signal monitored by rising_edge(), but also the signal used on the process sensitivity list.

# Chapter 19

# Generating a VGA signal

## 19.1 Aims of module

- Generate tight tolerance signals
- Display something on a VGA monitor

Let me know if I haven't given enough directions on how to implement this module. I think that the less hand-holding given the greater the joy when your project actually displays something for the first time.

---

**Special note for Basys2 users**
The Basys2 reference manual infers that the oscillator on the board isn't too stable. Digilent recommends to use an quality aftermarket oscillator to correct this, but the reference manual has the wrong part number - you want to order a SGR-8002DC-PCC-N from Digikey (the only place that seems to have it!)
Test your board/monitor compatibility using the the board self test that is in the flash, or from the file from Digilent's web site if you suspect that this is an issue.
I have not been able to get an current 1080p HD LCD monitor to display a picture (although I've only tried two), but it works on plenty of CRTs.
A cheep fix may be adding additional load to the power supply with a 150 Ohm resistor will help - see http://www.youtube.com/-watch?v=bVee4dDwO1k
I have had no such issues with my Papilio One - I've even generated signals 1920 x 1080 @ 60Hz (145MHz).

---

## 19.2 VGA signal timing

For this demo we will be aiming at 640x480. As detailed on http://tinyvga.com/vga-timing/640x480@60Hz this required a pixel clock of 25.175MHz . 25MHz is close enough for most monitors to sync up and display a stable image, and using a DCM we can generate that frequency from either 32MHz of the Papilo's crystal or the 50MHz of the Basys2 clock generator.

## 19.3 How does the VGA interface work?

In the "good ol' days" most monitors were analogue devices, using Cathode Ray Tubes. Two signals controlled the position of the electron beam on the display.

### 19.3.1 Vertical sync (vsync)

This signal gets pulsed every 60th of a second, and took the electron beam back to the top of the screen. Once back a the top of the screen the monitor would scan the beam slowly down the screen.

In this video mode the pulse is negative pulse of 0.063555ms, every 16.6832ms.

### 19.3.2 Horizontal sync (hsync)

This signal is a pulsed every 1/31,468th of a second, and causes the electron beam to return to the left hand side of the monitor. Once there the beam scans rather more rapidly to the right hand side.

In this video mode it is a positive pulse of 3.8133068us every 31.777557us.

When properly timed, the correct hsync and vsync timings caused the electron beam to scan the whole visible area, so all that is needed is the colour signals.

### 19.3.3 The colour signals - red, green and blue

These are analogue signals which controlled the intensity of each colour, and each pixel gets 1/25,175,000th of a second.

These must only driven only for the correct portion of horizontal scan as the monitor uses the "blanking interval" to register what voltages are used for black. You have two blanking intervals - the horizontal blanking interval (either side of the hsync pulse) and the vertical blacking interval (either side of the vsync pulse.

## 19.4 Pins used to drive the VGA connector

Ten pins are used to drive the VGA connecter - the Red, Green and Blue signals use a passive D2A convertor made out of resisters

**The constraints for the Papilio board are:**

```
NET "HSYNC"    LOC = "??"  | DRIVE = 2;
NET "VSYNC"    LOC = "??"  | DRIVE = 2;
NET "Red<2>"   LOC = "??"  | DRIVE = 2;
NET "Red<1>"   LOC = "??"  | DRIVE = 2;
NET "Red<0>"   LOC = "??"  | DRIVE = 2;
NET "Green<2>" LOC = "??"  | DRIVE = 2;
NET "Green<1>" LOC = "??"  | DRIVE = 2;
NET "Green<0>" LOC = "??"  | DRIVE = 2;
NET "Blue<2>"  LOC = "??"  | DRIVE = 2;
NET "Blue<1>"  LOC = "??"  | DRIVE = 2;
```

**The constraints for the Basys2 board are:**

```
NET "HSYNC"    LOC = "J14"  | DRIVE = 2;
NET "VSYNC"    LOC = "K13"  | DRIVE = 2;
NET "Red<2>"   LOC = "F13"  | DRIVE = 2;
NET "Red<1>"   LOC = "D13"  | DRIVE = 2;
NET "Red<0>"   LOC = "C14"  | DRIVE = 2;
NET "Green<2>" LOC = "G14"  | DRIVE = 2;
NET "Green<1>" LOC = "G13"  | DRIVE = 2;
NET "Green<0>" LOC = "F14"  | DRIVE = 2;
NET "Blue<2>"  LOC = "J13"  | DRIVE = 2;
NET "Blue<1>"  LOC = "H13"  | DRIVE = 2;
```

## 19.5 Making the timings easy implement

If you multiply the hsync and vsync timings by the pixel clock you will get something close to the following numbers:

| Scanline (Horizontal) timing | Duration in pixel clocks |
|---|---|
| Visible area | 640 |
| Front porch | 16 |
| Sync pulse | 96 |
| Back porch | 48 |
| Whole line | 800 |

The horizontal blanking interval is the front porch + sync pulse + back porch = 160 pixel clocks

| Frame (vertical) timing | Duration in lines (800 pixel clocks) |
|---|---|
| Visible area | 480 |
| Front porch | 10 |
| Sync pulse | 2 |
| Back porch | 33 |
| Whole frame | 525 |

The vertical blanking interval is the front porch + sync pulse + back porch = 45 lines

## 19.6 The RGB signal

The Basys2 board can generate only 256 colours - four shades of red, 8 shades of green and 8 shades of blue. It does this using a passive D-to-A converter made up of a dozen of so resistors. There really isn't much more to say!

## 19.7 Pseudo-code implementation

Implementation of the hsync and vsync signals should be coming clear. Here it is in pseudo-code:

```
hcounter and vcounter are 10 bit counters

every 1/25,000,000th of a second
  if hcount == 799 then
    hcount = 0
    if vcount == 524 then
      vcount = 0
    else
      vcount = vcount + 1
    end if
  else
    hcount = hcount + 1
  end if

if vcount >= 490 and vcount < 491 then
  vsync = '0'
else
  vsync = '1'
end if
```

```
if hcount >= 656 and hcount < 752 then
  hsync = 0
else
  hsync = 1
end if

if hcount < 640 and vcount < 480 then
  display a colour on the RGB signals
else
  display black colour on the RGB signals
end if
```

## 19.8 Project - Displaying something on a VGA monitor

- Create a new project to drive the VGA device. It needs to accept a clk signal and generate hsync, vsync, red(2 downto 0), green(2 downto 0) and blue(2 downto 1) outputs. *"Note that it is Blue(2 downto 1) not Blue(2 downto 0)*"
- Add a implementation constraint file and add the definitions for *clk* and the 10 VGA signals.
- Implement the horizontal counter (you will need a 10 bit counter). Remember to include the unsigned library ("use IEEE.STD_LOGIC_ so you will be able to do numeric operations on STD_LOGIC_VECTOR signals.
- Run it in the simulator, and verify the pulse widths and direction.
- Implement the vertical counter (once again you will need a 10 bit counter). You can also verify this in the simulator, but as you need to simulate 16,667us to see the whole frame it can take a while!
- To generate a white image, assign '1's to all the RGB signals during the active time. Test this too in the simulator. You only want to see '1's for the first 640 pixel clocks of the first 480 lines.
- If all looks correct, plug a VGA monitor into your board. It should detect the signal and display an image.
- Rather than assigning '1's to the RGB values, experiment with assigning different bits out of hcounter and vcounter - you can make colour bars and check-board patterns.
- Look really closely at the simulation. Do the RGB values go to *1* when hcounter transitions from 799 back to 0? If not, why not?

## 19.9 A common cause of problems

It looks as though this code doesn't need to go into a "if rising_edge(clk) then..." block:

```
if hcount >= 656 and hcount < 752 then
  hsync = '0'
else
  hsync = '1'
end if

if vcount >= 490 and vcount < 491 then
  vsync = '0'
else
  vsync = '1'
end if
```

For maximum reliably it does. As the counters ripple between two values (remember, at about 0.1ns per bit) the binary value of the counters will be in transition. If the signals are not buffered in a flip-flop the hsync and vsync can contain unpredictable pulses of around 1ns wide. You won't see these in simulation, and not many of use have a 1GHz Logic Analyser or 'scope, but it is really there.

I've generated a 1440x900 signal (105MHz clock rate) and used logic to display objects on the screen. If I didn't buffer the RGB values the objects wouldn't show correctly or had fuzzy edges. Registering all the VGA signals made these problems go away, as the signals were solidly high or low for the entire clock duration.

This is only an annoyance while generating VGA signals, but if you are interfacing into other devices (e.g. SRAM) this can cause you no end of heartache. A few *implementation time* tool options are available that can alter this too, by forcing all the I/O flip-flops to be put as close to the pin as possible, instead of being buried away in the middle of the FPGA fabric. It is also possible to add an "IOB=TRUE" constraint to your UCF file to enable this behaviour on a pin by pin basis.

# Chapter 20

# Communicating with the outside world

So, after displaying something on a VGA monitor, how do we talk to a PC?

In this chapter you will build the transmit part of a serial (RS232) interface, using shift registers. On the Papilio One you can talk directly to the USB interface, but on the Basys2 you will need a USB to 3.3V Serial breakout board.

## 20.1 What is RS232?

RS232 is a very old standard originally used to interface digital systems with analogue phone lines and other data circuits. It enables relatively low speed communication between devices, and is relatively simple to implement.

If hardware handshaking is not used only three wires are needed:

| Wire | Use |
|---|---|
| GND | Signal Ground |
| TX | Transmitted data |
| RX | Received data |

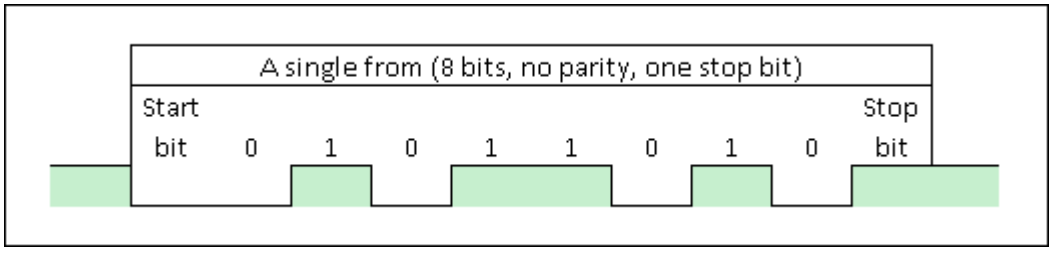
Which signal a device listens to for incoming data and which signal it actively sends data is very confusing. If the device is a "DTE" (Data Terminating Equipment) it transmits on TX and listens on RX. If the device is "Data Communicating Equipment" (e.g. a modem) it listens for data on TX and transmits on RX.

The standard speeds range from 75 baud up to 115,200 baud, with 9600 or 19200 being the most common speeds for data that is presented to people (such as on a serial console).

As well as baud speed both ends of a connection must be using the same frame parameters - the most common being one start bit, eight data bits, no parity bit and one stop bit. As the frame is ten bits long at 9600 baud you can send 960 bytes per second.

There is a whole lot more to the standard, mostly around how senders and receivers control the flow of data to ensure that data does not overrun receiving buffers. When using modern hardware at slow speeds handshaking isn't really an issue.

Here's what it looks like on the wire:



## 20.2 Generating an RS-232 signal

For this project we need a shift register (well two actually). So what does a shift register look like in VHDL?

Here is a 16 bit register that loops from bit 0 to bit 15 - a much simpler way to generate one pulse every 16 cycles than using a counter

```
...
signal shiftreg : std_logic_vector(15 downto 0) := "0000000000000001";
...
if rising_edge(clk) then
   shiftreg <= shiftreg(0) & shiftreg(15 downto 1);
end if;
```

For RS-252 we use pretty much this construct, but feed in the the idle bit value (*1*). This code will send the *Z* character once (after which the shift register is filled with '1's):

```
...
signal shiftreg : std_logic_vector(9 downto 0) := "1010110100";
...
data_out <= shiftreg(0);
...
if rising_edge(clk) then
   shiftreg <= '1' & shiftreg(9 downto 1)
end if;
```

The user data is bits 8 downto 1 is the "byte" of user data - bit 0 is the start bit, and bit 9 is the stop bit. I chose the ASCII code for *Z* as it will still be a *Z* regardless of if the least or most significant bit gets transferred first - very useful for testing!

The only problem with the code so far is that we are transmitting at the clock speed - either 32,000,000 or 50,000,000 baud! To control the rate of sending we also need a counter that allows a bit to be sent at 9600 baud - once every 3333 cycles (at 32MHz) or once every 5208 clock cycles (@50MHz):

```
...
signal shiftreg : std_logic_vector(9 downto 0) := "1010110100";
signal counter : std_logic_vector(12 downto 0) := (others => '0');
...
data_out <= shiftreg(0);
...
if rising_edge(clk) then
   if counter = 3332 then
      shiftreg <= '1' & shiftreg(9 downto 1);
      counter <= (others => '0');
   else
      counter <= counter+1;
   end if;
end if;
```

We can make it send the same data over and over again by making the shift register longer and looping the shift register's output back on its input. To do this it needs longer shift register, ensuring that we have some quiet space following the stop bit to allow the receiver to frame the data correctly:

```
...
signal shiftreg : std_logic_vector(15 downto 0) := "1111111010110100";
signal counter : std_logic_vector(12 downto 0) := (others => '0');
...
data_out <= shiftreg(0);
...
if rising_edge(clk) then
   if counter = 3332 then
      shiftreg <= shiftreg(0) & shiftreg(15 downto 1);
      counter <= (others => '0');
```

```
    else
       counter <= counter+1;
    end if;
end if;
```

(This code should be enough to enable you to test your RS232 port actually sends data as expected).

## 20.3 Sending variable data

To make this useful you really need to be able to send different data bytes. And to do this correctly you have to know when the interface is busy.

The easiest way to do this is to have a second shift register which is filled with *1's when the character is loaded into 'shiftreg* and filled with '0's as bits are transmitted. Once this second shift register is all zeros, then things are ready for the next byte to be sent:

```
...
signal busyshiftreg : std_logic_vector(9 downto 0) := (others => '0');
signal datashiftreg : std_logic_vector(9 downto 0) := (others => '1');
signal counter : std_logic_vector(12 downto 0) := (others => '0');
...
data_out <= datashiftreg(0);
busy_out <= busyshiftreg(0);
...
if rising_edge(clk) then
    if busyshiftreg(0) = '0' then
       busyshiftreg <= (others => '1');
       datashiftreg <= '1' & databyte & '0';
       counter <= (others <= '0');
    else
       if counter = 3332 then
          datashiftreg <= '1' & datashiftreg(9 downto 1);
          busyshiftreg <= '0' & busyshiftreg(9 downto 1);
          counter <= (others => '0');
       else
          counter <= counter+1;
       end if;
    end if;
end if;
```

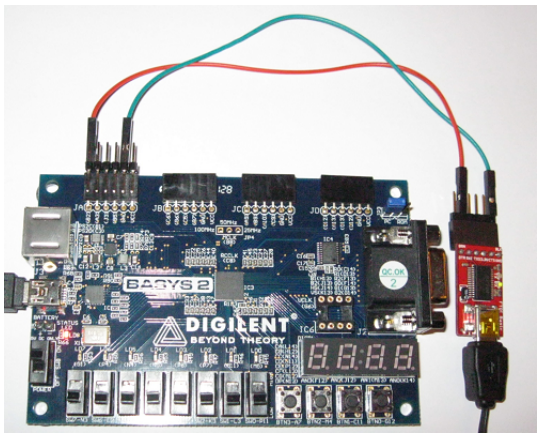
The important bit is to remember to reset *counter* when a new byte is loaded into *datashiftreg*. Failing to do this will cause the start bit to be of different lengths - the project will work correctly when streaming bytes to the host, but will sometimes get garbage for the first few bytes of a message until it recovers from the bad bit.

## 20.4 Connecting your FPGA board to a PC

**Caution**
Connecting the FPGA directly to your serial port will most likely ruin your FPGA```

Most modern PCs do not have RS232 ports, and if they do they are expecting the higher voltage levels that standard RS232 uses - the standards uses up to +/- 25V!

To connect to a PC over USB you can use something like Sparkfun's "FTDI Basic 3.3V - USB to Serial" (http://www.sparkfun.com/-products/9893) and wire jumpers. Here's my setup:




**Tip**
If you are using the Basys2 and want to talk to a true standards compliant RS-232 port, or if you want to avoid issues caused by loose wires you can use the RS-232 PMOD http://www.digilentinc.com/Products/Detail.cfm?Prod=PMOD-RS232 with your Basys2.

## 20.5 Project 16.1

- Create a project that sends *Z* over RS-232
- Create a project that sends the state of switches(3 downto 0) over RS-232
  - You could increase the length of the shift register and send multiple bytes
  - You could convert the data to ASCII and send four switches in a single byte
  - You could map the 16 possible values into 16 contiguous printable characters (A-Q perhaps)
- Change it to only send a byte when the switches change.
- Extend the project to send the state of all eight switches

## 20.6 Challenge

- What would happen if the input to the RS232 TX component was to change, and then change back to its original state in less than 1/960th of a second? Can loss of data be avoided?

# Chapter 21

# Recieving data from the outside world

In the last module we created one-way communicaiton from the FPGA to a PC. It would be really good if we could send data from the PC back to the FPGA too.

To allow this to happen we need to be able to recover the sender's clock - a process called clock recovery.

## 21.1 Problems with clock recovery and framing

Synchronising with an incoming signal is usually a hard problem to solve. But for short transfers using low bit rates (like RS232) it is pretty easy to solve by oversampling the incoming signal. Although this isn't the most efficient method it is easy to understand and implement.

If the incoming signal has a bit rate is 9600 baud your design over samples the signal at four times this speed (38400), thus ensuring that for each received bit we will have two good samples.

The next challenge is then to work out which pairs of bits are good, and where a frame starts and ends. Here's my solution.

As discussed in the last module an RS232 frame starts with a Start bit (*L*), has eight data bits and ends with a stop bit (*H*). To receive the data use a 40 bit shift register initialised to '1's, and then capture the incoming signal into the left hand end of a shift register.

After 40 samples we will have the following bits Where - is *don't care* and *??* are pairs of matching *LL* or *HH* bits (as they will have been sampled in the sample bit windows):

```
(MSB)                                   (LSB)
-HH--??--??--??--??--??--??--??--??--LL-
Stop  7   6  5   4   3   2   1  0   start
```

If we see this pattern we know have a valid frame, and can then make use of the data

The test to see if we have received a valid frame we need to check the following:

- Check that bits(38 downto 37) = *1*
- Check that bits(34) are bits(33) the same
- Check that bits(30) are bits(29) the same
- Check that bits(26) are bits(25) the same
- Check that bits(22) are bits(21) the same
- Check that bits(18) are bits(17) the same

- Check that bits(14) are bits(13) the same
- Check that bits(10) are bits(9) the same
- Check that bits(6) are bits(5) the same
- Check that bits(2 downto 1) are both *0*

If all this is true we can then capture the byte, set a signal to indicate receiving of the byte then reset the shift register back to the empty state, preventing false triggering:

```
value          <= bits(34) & bits(30) & bits(26) & bits(22) & bits(18) & bits(14) & bits ↩
    (10) & bits(6);
byte_received <= '1';
bits          <= (others => '1');
```

Wow - much easier than expected. So what is the catch?

## 21.2 Problems with this solution

The main problem with this solution is that the sending clock and receiving clock must be closely matched. A drift of 2.5% (1/40) in clocks will be enough that the sampling of the start bit and stop bit will be one sample out of step from each other - but will still work with a very *crisp* signal.

If there is a difference of 5% in timing the first and last sample will be two sample periods out of step, and will never be able to receive the data correctly.

## 21.3 Project 17.1

- Create a project that receives characters over RS232 and displays them on the LEDs or seven segment display
- Merge the code from project 16 and 17 to create your own RS232 RX/TX component.

# Chapter 22

# A high speed external interface

This chapter is only applicable to Basys2 board - The Papilio board only has a serial port. It also assumes that you are using the Windows OS - but I'm sure that only minor changes are needed for it all to work under Linux too.

## 22.1 The Digilent Parallel Interface

Digilent FPGA boards have a port of the USB interface wired to the FPGA. I've used this to transfer data at up to 11 megabytes per second. The supplied documentation is pretty terse, so here is a quick start guide.

The interface implements the long obsolete EPP protocol that was traditionally used to talk to parallel port scanners. It allows the connected device to address up to 256 8-bit registers that can be implemented within the FPGA.

These registers can either be read by the host PC one byte at a time, or a "Repeat" function can be called to read multiple bytes from the same register.

The most "make or break" shortcoming of this interface is that there is no interrupt signal going back to the host which would allow the FPGA it's attention. Unlike when using RS232 this forces the host software to poll the FPGA at regular intervals - which is not ideal for responsiveness or CPU usage.

## 22.2 Resources

- http://www.digilentinc.com/data/software/adept/dpimref%20programmers%20manual.pdf documents the FPGA side of the interface
- http://digilentinc.com/Data/Products/ADEPT/DPCUTIL%20Programmers%20%20Reference%20Manual.pdf documents the host side of interface
- http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT2 for the latest Adept SDK

## 22.3 The FPGA side of the interface

The following signals make up the interface:

| Name | Type | Description |
|---|---|---|
| DB(7 downto 0) | INOUT | Data bus |
| WRITE | IN | Write Enable (active Low) - data will be written from the host during this cycle |

| | | |
|---|---|---|
| ASTB | IN | Address strobe (active low) - data bus will be captured into the address register |
| DSTB | IN | Data Strobe (active low) - bus will be captured into the currently selected data register |
| WAIT | OUT | Asserted when FPGA read to accept data, |
| INT | OUT | Interrupt request - not used |
| RESET | IN | Reset - not used |

## 22.4 Read Transaction

The steps in a read transaction are

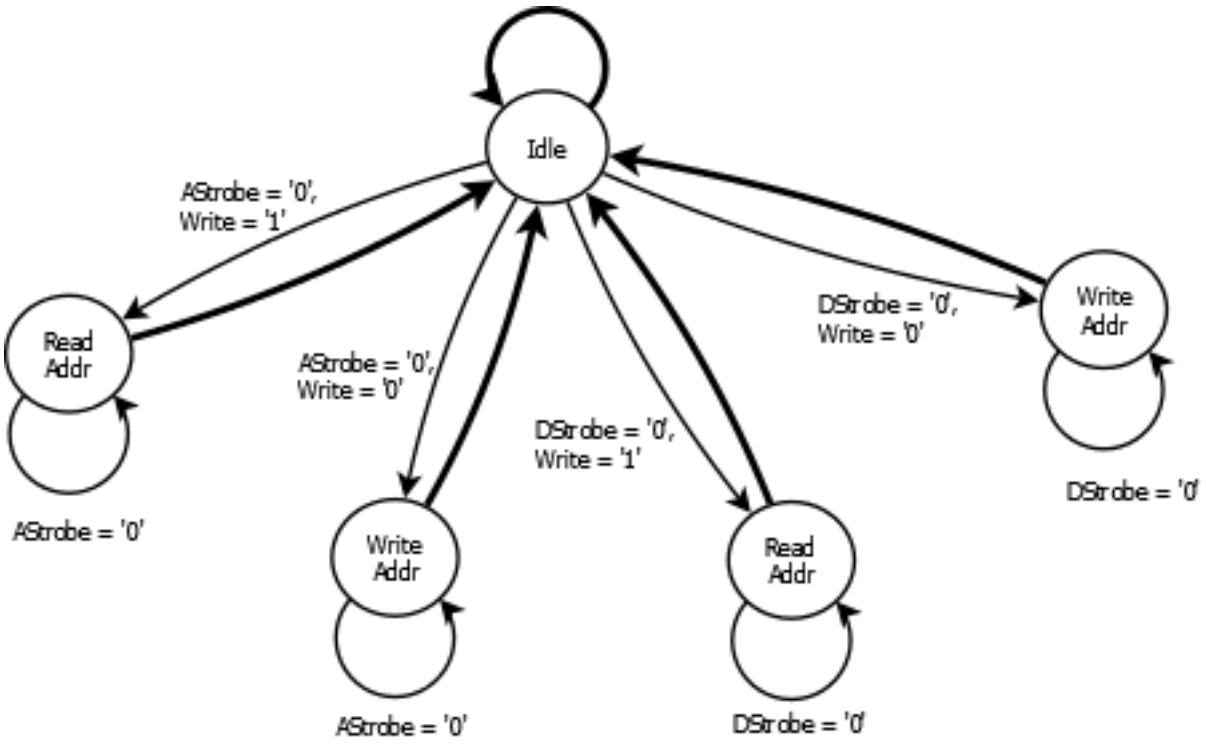
- Host lowers ASTB or DSTB to commence read of either the address register or the selected data register
- FPGA presents data on data bus
- FPGA raises WAIT indicating that the data is valid
- Host captures the data
- Host raises ASTB or DSTB
- FPGA removes the data from the data bus
- FPGA lowers WAIT to finish transaction

## 22.5 Write transaction

The steps in a write transaction are

- Host presents data on data DB()
- Host lowers Write Enable to 0
- Host lowers either ASTB or DSTB to commence write of either the address register or the selected data register.
- FPGA raises WAIT once data is captured
- Host raises ASTB or DSTB, removes data from bus and raises Write Enable
- FPGA lowers WAIT to finish transaction

## 22.6 FSM diagram




## 22.7 Constraints for the BASYS2 board

The constraints required to implement the interface are:

```
NET "EppAstb" LOC = "F2"; # Bank = 3
NET "EppDstb" LOC = "F1"; # Bank = 3
NET "EppWR"   LOC = "C2"; # Bank = 3

NET "EppWait" LOC = "D2"; # Bank = 3

NET "EppDB<0>" LOC = "N2"; # Bank = 2
NET "EppDB<1>" LOC = "M2"; # Bank = 2
NET "EppDB<2>" LOC = "M1"; # Bank = 3
NET "EppDB<3>" LOC = "L1"; # Bank = 3
NET "EppDB<4>" LOC = "L2"; # Bank = 3
NET "EppDB<5>" LOC = "H2"; # Bank = 3
NET "EppDB<6>" LOC = "H1"; # Bank = 3
NET "EppDB<7>" LOC = "H3"; # Bank = 3
```

## 22.8 VHDL for the FPGA interface

This source allows you to set the LEDs and read the switches from the PCs. It has a few VHDL features that you won't have seen up to now:

- The EppDB (EPP Data bus) is INOUT - a tristate bidirectional bus. When you assign "ZZZZZZZZ" (high impedance) to the signal it will then *read* as the input from the outside world. This is only really useful on I/O pins - within the FPGA tristate logic is implemented using multiplexers

- It uses an enumerated type to hold the FSM *state*. This is only really useful if you don't want to use bits withing the state value to drive logic (which is usually a good way to get glitch free outputs).

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

entity epp_interface is
  port (Clk     : in    std_logic;
        -- EPP interface
        EppAstb : in    std_logic;
        EppDstb : in    std_logic;
        EppWR   : in    std_logic;
        EppWait : out   std_logic;
        EppDB   : inout std_logic_vector(7 downto 0);

        -- Feedback
        switches: in    std_logic_vector(7 downto 0);
          leds    : out   std_logic_vector(7 downto 0)
  );
end epp_interface;

architecture Behavioral of epp_interface is
  type   epp_state is (idle, data_read, data_write, addr_read, addr_write);
  signal state      : epp_state := idle;
  signal address    : std_logic_vector(7 downto 0) := (others => '0');
  signal port0data  : std_logic_vector(7 downto 0) := (others => '0');
begin
  process(clk)
  begin

     if rising_edge(clk) then
        case state is
           when data_read  =>
              EppWait <= '1';
              case address is
                 when "00000000" =>
                    EppDB <= not port0data;
                 when "00000001" =>
                    EppDB <= switches;
                 when others =>
              end case;

              if EppDstb = '1' then
                 state <= idle;
              end if;
           when data_write =>
              EppWait <= '1';
              case address is
                 when "00000000" =>
                    port0data <= EppDB;
                 when "00000001" =>
                    leds <= EppDB;
                 when others =>
              end case;

              if EppDstb = '1' then
                 state <= idle;
              end if;

           when addr_read  =>
              EppWait <= '1';
```

```
                EppDB   <= address;
                if EppAstb = '1' then
                   state <= idle;
                end if;

             when addr_write =>
                EppWait <= '1';
                address <= eppDB;
                if EppAstb = '1' then
                   state <= idle;
                end if;

             when others =>
                EppWait  <= '0';
                EppDB <= "ZZZZZZZZ";
                if EppWr = '0' then
                   if  EppAstb = '0' then
                      state <= addr_write;
                   elsif EppDstb = '0' then
                      state <= data_write;
                   end if;
                else
                   if EppDstb = '0' then
                      state <= data_read;
                   elsif EppAstb = '0' then
                      state <= addr_read;
                   end if;
                end if;
          end case;
       end if;
    end process;
end Behavioral;
```

## 22.9 The PC side of the interface

### 22.9.1 Header files and libraries

These are in the Adept SDK, which can be downloaded from http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT2

The zip file includes all the files you need, including documentation, libraries and examples.

The following header files are needed in your C code:

- gendefs.h
- dpcdefs.h
- dpcutil.h

You will also need to add the path to the libraries into your project's linking settings.

### 22.9.2 Connecting to a device

Connecting isn't that simple, but it's not that hard either. Three functions are needed:

- DpcInit()
- DvmgGetDefaultDev()

- DvmgGetDevName()

```
        if (!DpcInit(&erc)) {
                printf("Unable to initialise\n");
                return 0;
        }

        id = DvmgGetDefaultDev(&erc);
        if (id == -1) {
                printf("No default device\n");
                goto error;
        }

        if(!DvmgGetDevName(id, device, &erc)) {
                printf("No device name\n");
                goto error;
        }
```

```
The first time you make use of the interface you may need to call one more

function only once to present a dialogue box allowing you to select which FPGA board
will be your default device:

* DvmgStartConfigureDevices()
```

## 22.10 Connecting to the EPP port of that device

One function is used to connect to the device (vs connecting to the JTAG port):

- DpcOpenData()

```
        if (!DpcOpenData(&hif, device, &erc, NULL)) {
                goto fail;
        }
```

## 22.11 Reading a port

Reading a port is achieved with either of these functions:

- DpcGetReg() - Read a single byte from a register
- DpcGetRegRepeat() - Read multiple bytes from a register

Here's an example function that opens the EPP port and reads a single register

```
  static int GetReg(unsigned char r) {
        unsigned char b;
        ERC             erc;
        HANDLE  hif;

        if (!DpcOpenData(&hif, device, &erc, NULL)) {
                goto fail;
        }

        if (!DpcGetReg(hif, r, &b, &erc, NULL)) {
```

```
                DpcCloseData(hif,&erc);
                goto fail;
        }

        erc = DpcGetFirstError(hif);
        DpcCloseData(hif, &erc);

        if (erc == ercNoError)
                return b;
 fail:
        return -1;
}
```

## 22.12 Writing to a register

Reading a port is achieved with either of these functions:

- DpcGetReg() - Read a single byte from a register
- DpcGetRegRepeat() - Read multiple bytes from a register

Here's an example function that opens the EPP port and writes to a single register

```
static int PutReg(unsigned char r, unsigned char b) {
        ERC             erc;
        HANDLE  hif;
        printf("Put %i %i\n",r,b);
        if (!DpcOpenData(&hif, device, &erc, NULL)) {
                goto fail;
        }

                if(!DpcPutReg(hif, r, b, &erc, NULL)) {
                        DpcCloseData(hif,&erc);
                        goto fail;
                }

        erc = DpcGetFirstError(hif);
        DpcCloseData(hif, &erc);

        if (erc == ercNoError)
                return 0;

fail:
        return -1;
}
```

## 22.13 Closing the EPP port

One function is used to close the EPP port:

- DpcCloseData()

```
        DpcCloseData(hif, &erc);

        if (erc == ercNoError)
                return b;
```

## 22.14 Closing the interface

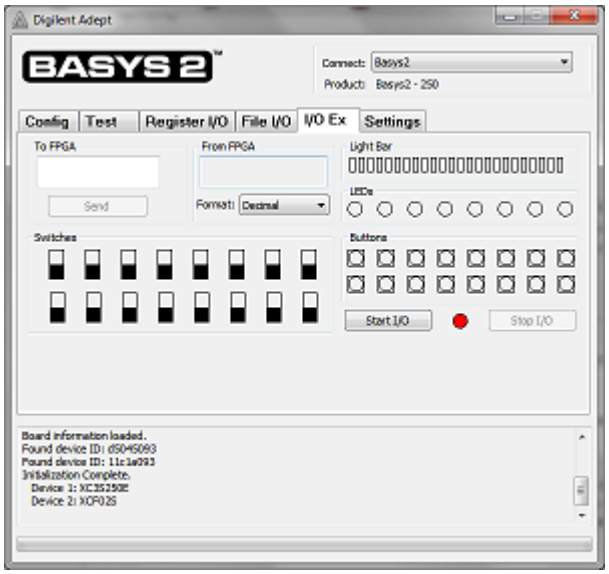
It is always good to clean up after yourself. Use the following function to do so:

- DpcTerm()

```
        DpcTerm();
```

## 22.15 Project - Using the PC end of the interface

- Download and configure your board with the "Adept I/O expansion reference design" project from http://www.digilentinc.com/-Products/Detail.cfm?Prod=BASYS2
- Check that the Adept I/O expansion tab responds to changes in the switches




- Create a C program that opens the interface and reads a single byte from registers 5 and 6 and displays the value to the screen.
- Close of Adept and check that your C program also shows the state of the switches on the Basys2.
- Expand you C program to write to the value of the switches to register 1 - this is the LEDs.

You now have the host side of bidirectional communication sorted!

## 22.16 Project - Implementing the FPGA end of the interface

- Create a new FPGA project
- Create a module that implements the EPP protocol - or use the one above if Digilent's reference design if you want.
- Connect writes of register 1 to the LEDs.
- Connect reads of register 5 or 6 to the switches
- Test that your design works just as well with your program as Digilent's reference design

# Chapter 23

# Binary Multiplication

Up to now we have managed to complete all the projects using only logical operations and addition or subtraction. But then there comes a time when you need multiplication - and this is where FPGAs really shine.

After going through the basics of binary multiplication you'll be introduced to the embedded multiplier blocks in the Spartan 3E. The XC3S250E FPGAs have twelve of these blocks, allowing you to do number crunching of well over a 2 billion 18 bit multiplications per second, allowing it to compete with a desktop CPU core.

## 23.1 Performance of binary multiplication

Binary multiplication is complex - implementing multiplication of any two *n* bit numbers approximately n*(n-1) full adders, n half adders and n*n *AND* operations.

To multiply the four bit binary numbers "abcd" by "efgh" the following operations are needed (where *&* is a binary AND):

```
+                  a&h b&h c&h d&h
+              a&g b&g c&g d&g  0
+          a&f b&f c&f d&f  0   0
+      a&e b&e c&e d&e  0   0   0
   --- --- --- --- --- --- --- ---
=  ?   ?   ?   ?   ?   ?   ?   ?
```

Multiplication also has a big implication for your designs performance - because of the *carries* required, multiplying two *n* bit numbers takes around twice as long as adding two *n* bit numbers.

It also consumes a very large amount of logic if multiplication is implemented in the generic logic blocks within the FPGA.

## 23.2 Multiplication in FPGAs

To get around this, most FPGAs include multiple multiplier blocks - a XC3S100 has four 18 bit x 18 bit multipliers, and a XC3S250 has twelve!

To improve performance multipliers also include additional registers, allowing the multiplicands and result to be registered withing the multiplier block. There are also optional registers within the multiplier that holds the partial result half way through the multiplication.

Using these internal registers greatly improves throughput performance by removing the routing delays to get the inputs to and from the multipliers, but a the cost of increased latency - measured in either time between two number being given to the multiplier and the result being available, or the number of clock cycles.

When all these internal registers are enabled the multiplier works as follows

| Clock cycle | Action |
|---|---|
| 0 | A and B inputs are latched |
| 1 | The first half of the multiplication is performed |
| 2 | The second half of the multiplication is performed |
| 3 | The result of the multiplication is available on the P output |

Multipliers can accept a new set of *A* and *B* values each clock cycle, so up to three can be *in flight* at any one time. In some cases this is useful but in other cases it can be annoying.

A useful case might be processing Red/Green/Blue video values, where each channel is separate.

An annoying case is where feedback is needed of the output value back into the input. If the math isn't in your favor you may be better off not using any registers at all - it may even be slightly faster and running at one third the clock speed will use less power.

## 23.3 What if 18x18 isn't *wide* enough?

What if you want to use bigger numbers? say 32 bits? Sure!

Just like in decimal when multiplying two two digit numbers "ab" and "cd" is calculated as "a*c*10*10 + b*c*10 + a*d*10 + b*d" the same works - just replace each of a,v,c,d with an 18 bit number, and each *10* with 2^18.

As the designer you have the choice of either:

- using four multipliers and three adders, with a best-performance latency of 5 cycles with a throughput of one pair of A and B values per clock
- using the same multiplier to calculate each of the four intermediate products, with a best-performance latency of 13 cycles (four 3-cycle multiplication plus the final addition) and with careful scheduling you can process three input pairs every 12 cycles.

## 23.4 Project - Digital Volume control

- Revisit the Audio output project
- Use the core generator to add an IP multiplier, with an 8 bit unsigned input for the volume and the other matching your BRAM's sample size
- Add a multiplier between the block BRAM and the DAC, using the value of switches as the other input.
- Use the highest output bits of the multiplier to feed the DAC
- If you get the correct *signed* / *unsigned* settings for each input of the multiplier you will now be able to control the volume with the switches

Unless you are careful you may have issues with mismatching signed and unsigned values - it pays to simulate this carefully!

- You can also implement multiplication using the generic logic ("LUT"s). If interested, you can change the IP multiplier to use LUTs instead of the dedicated multiplier blocks and compare maximum performance and resource usage.

# Chapter 24

# Using an ADC

This chapter is only applicable to Papilio One board, as the Basys2 does not include any ADC functionality - it is still a useful read as it shows how simple periperals can be to interface to.

Unlike other project so far, I've included the full code for the module, giving some sort of reference implementation that can be used to verify your own design.

## 24.1 The ADC

The ADC on the LogicStart is an 8 channel 12 bit ADC, with a serial interface compatible with the Serial Peripheral Interface Bus ("SPI") standard. The reference voltage for the ADC is 3.3V, giving a resolution of about 0.8mV.

In the official SPI bus specifies four logic signals are called:

- SCLK: serial clock (output from master);
- MOSI; SIMO: master output, slave input (output from master);
- MISO; SOMI: master input, slave output (output from slave);
- SS: slave select (active low, output from master).

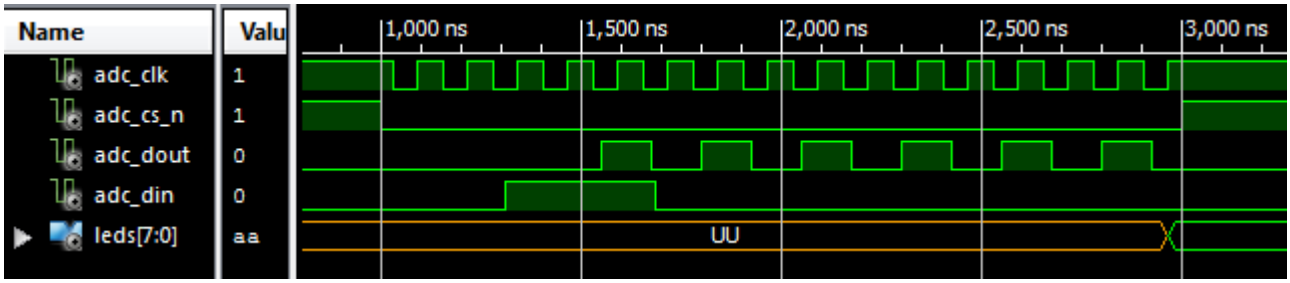
But for this design I'm following the names used in the datasheet - which are named from the perspective of the slave device:

- CS; Chip Select
- DIN; Date In
- DOUT; Data Out
- SCLK; Serial Clock

To read channel 0 of the ADC it is pretty simple

- Hold DIN low (this ensures that you read channel 0)
- Hold CS high while the ADC is idle
- Lower CS when you are ready to convert a sample
- Send 16 clock pulses of with a frequency somewhere between 8MHz and 16MHz,
- Raise CS when finished

The data bits will be available on DOUT for clocks pulses 4 through 16.




To read another diffeent channel is a little harder - you need to give the ADC the bits to select the channel *'for the next sample'* on clock pulses 2, 3 and 4. These bits are sent in MSB first order.

This sounds simple enough, but as ever the difficulty is in the details. To make this work reliably the setup and holdup times must be factored in:

- CS must go low a few ns before the SCLK line drops for the first time
- The DOUT signal transition just after the rising edge of the SCLK signal. For reliable results it needs to be sampled mid-pulse
- The DIN signal must be given enough time to be stable before the SCLK falls

I decieded that the easiest way to do this is to run a counter at the 32MHz clock of the crystal, then the gross timings for the signals are:

- the SCLK signal is generate from bit 2 of a counter running at the system clock of 32MHz
- bits 3 through 6 indicates what bit of the frame we are on
- if bit 7 or over are set, then CS is held high.
- Data is sampled when the lowest two bits are "10"

To ensure that I don't have any setup and hold time issues with the interface a shift register is used to delayed the SCLK signal by one cycle, and a second shift register is used to delay DIN by three clocks. This ensures that CS and DIN have at plenty of setup and hold time.

## 24.2 VHDL for the interface

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity AtoD is
   port
   (
      clk         : IN  std_logic;
      -- user interface
      switches    : IN  std_logic_vector(2 downto 0);
      leds        : OUT std_logic_vector(7 downto 0);
      -- Signals to the ADC
      ADC_CS_N    : OUT std_logic;
      ADC_SCLK    : OUT std_logic;
      ADC_DIN     : OUT std_logic;
      ADC_DOUT    : IN  std_logic
   );
end entity;
```

```
architecture rtl of AtoD is
  -- Counter - the lowest 6 bits are used to control signals to the ADC.
  -- The rest are used to activate the ADC when 0
  signal counter          : std_logic_vector(22 downto 0) := (others =>'0');

  -- shift registers fo delay output signals
  signal clk_shiftreg     : std_logic_vector( 1 downto 0) := (others =>'0');
  signal dataout_shiftreg : std_logic_vector( 2 downto 0) := (others =>'0');

  -- shift register to collect incoming bits
  signal datain_shiftreg  : std_logic_vector(11 downto 0) := (others =>'0');

  -- register to hold the current channel
  signal channel_hold     : std_logic_vector( 2 downto 0) := (others =>'0');

  signal adc_active         : std_logic;
begin
  -- set outoging signals
  adc_din  <= dataout_shiftreg(2);
  adc_sclk <= clk_shiftreg(1);

  with counter(22 downto 6) select adc_active <= '1' when "00000000000000000",
                                                 '0' when others;

  process (clk)
  begin
    if rising_edge(clk) then
      -- A small shift register delays the clk by one cycle (31.25ns) to ensure timings are ↩
          met.
      clk_shiftreg(1) <= clk_shiftreg(0);

      -- Including adc_cs_n in a clocked process to ensure that it is adc_cs is implemented ↩
          in a flipflop
      adc_cs_n        <= not(adc_active);

      if adc_active = '1' then
        clk_shiftreg(0) <= counter(1);
      else
        clk_shiftreg(0) <= '1';
      end if;

      -- This controls where we send out the address to the ADC (bits 2,3 and 4 of the  ↩
          stream)
      -- we use a short shift register to ensure that the ADC_DOUT transistions are delayed
      -- 31 ns or so from the clk transitions
      dataout_shiftreg(2 downto 1)  <= dataout_shiftreg(1 downto 0);
      if adc_active = '1' then
        case counter(5 downto 2) is
          when "0010" => dataout_shiftreg(0) <= channel_hold(2);
          when "0011" => dataout_shiftreg(0) <= channel_hold(1);
          when "0100" => dataout_shiftreg(0) <= channel_hold(0);
          when others => dataout_shiftreg(0) <= '0';
        end case;

        -- As counter(2) is used used to generate sclk, this test ensures that we
        -- capture bits right in the middle of the clock pulse
        if counter(5 downto 0) = "000000" then
          channel_hold <= switches;
        end if;

        if counter(1 downto 0) = "11" then
          datain_shiftreg <= datain_shiftreg(10 downto 0) & adc_dout;
```

```
        end if;

        -- When we have captured the last bit it is the time to update the output.
        if counter(5 downto 0) = "111111" then
          -- Normally you would grab "datain_shiftreg(10 downto 0) & adc_dout" for 12 bits
          LEDs           <= datain_shiftreg(10 downto 3);
        end if;
      else
        dataout_shiftreg(0) <= '0';
      end if;

      counter <= counter+1;
    end if;
  end process;
end rtl;
```

Constraints for the Papilio One board

```
The constraints required to implement the interface are:

-------------------------------------
 NET LEDs(7) LOC = "P5";
 NET LEDs(6) LOC = "P9";
 NET LEDs(5) LOC = "P10";
 NET LEDs(4) LOC = "P11";
 NET LEDs(3) LOC = "P12";
 NET LEDs(2) LOC = "P15";
 NET LEDs(1) LOC = "P16";
 NET LEDs(0) LOC = "P17";

 NET switches(2) LOC = "P2";
 NET switches(1) LOC = "P3";
 NET switches(0) LOC = "P4";

 NET ADC_CS_N LOC="P70";
 NET ADC_SCLK LOC="P86";
 NET ADC_DOUT LOC="P79";
 NET ADC_DIN  LOC="P84";

 NET "clk" LOC="P89" | IOSTANDARD=LVCMOS25 | PERIOD=31.25ns;
-------------------------------------

Project - Playing with the ADC
```

- Modify the above project to output all 12 bits, and display it on the Seven Segment display in hex.

A jumper wire with a 100Ohm resistor is useful for testing, but only test using the GND, 2.5V and 3.3V singnals - connecting the ADC to 5V will damage it! Another option is to use one of the colour channels on the VGA socket, giving you a range of sixteen test values.

- If you multiply the value received by 129/16, you have a range of 0 to 33016 - very close to 10,000*Vin). The multiplication is wasy to do in logic, but can you convert the remaing binary back to Decimal to display on the seven segment display? One easy way would be to build a Decimal counter, that counts up to the sampled value.

# Chapter 25

# Using tri-state logic

After reviewing all the learning to date I realised that I have failed to cover tri-state logic! Although common when building using individual chips it only really makes an apperance in FPGA designs when interfacing to external componenets (explaining why it was only seen when interfacing to the Basys2's bidirectional EPP port).

## 25.1 What is tri-state logic?

Put simply, tri-state logic is where a signal can be either "logic high level", "logic low level" or "not actively driven" - *1*, *0* and *Z* in VHDL. This allows the same wire / signal to be used as both an input or output, or allow multiple devices to share "share" a common bus.

The most familiar example is a RAM chip's data bus. During the read cycles the memory chip drives the data bus, and during write cycles the memory controller drives the data bus. To enable this most RAM chips have a signal called "Output Enable" ("OE") that tells the chip when to drive the bus.

On a tristate bus all devices on the bus can read the value of the bus at any time, but to avoid data corruption your design must ensure that one device should drive the bus at any time. Should two or more devices try to drive the bus to different values at the same time the data on the bus will be corrupted. If this overlap of multiple devices driving the bus lasts for only for a short time the error may not noticed, but you will get high power usage and signal integrity as the drivers are saturated.

## 25.2 How is tri-state logic used within a FPGA

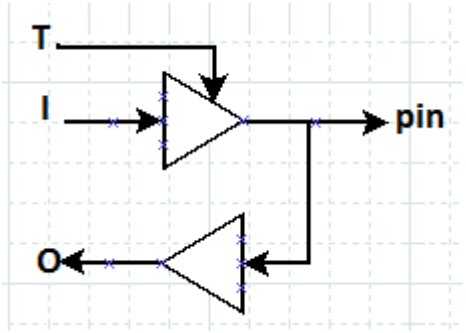
In short, for the Spartan 3E it isn't. To avoid timing and power issues the design tools ensure that any signals are only ever driven by one device.

Any internal tri-state logic within a design is mapped into a hidden "input" and "output" signals. The bus is then implemented with a multiplexer that selects the active *output* signal and then delivers that signal to all the inputs.

## 25.3 How is tri-state logic use when interfacing with a FPGA

Most general purpose I/O pins are of an FPGA are driven by a tri-state driver, and the pin is monitored by an input buffer.

When any internal tri-state is attached to an I/O pin it is implemented as three signals driving an IOBUF component:




- *T* controls the state of the tri-state driver
- *O* is the value of the pin
- *I* is the value that will be sent to the pin when *T* is asserted.

(Yes, the signal names do seem the wrong way around, but they are from the IOBUFs point of view)

## 25.4 Project - using tri-state logic

Sadly this project is Basys2 only - as on the Papilio plus the LogicStart MegaWIng uses all the I/O pins. It is possible to remove the MegaWing and connect directly to the headers on the Papilio One if you want...

- Create a new project
- Configure two of the PMOD pins. Remember to define the PMOD pins as "INOUT"!
- Have 2 LEDs show the status to the two pins on a PMOD connector,

```
led(0) <= pmod(0);
led(1) <= pmod(1);
```

- Connect two slide switches to these pins

```
  pmod(0) <= sw(0);
  pmod(1) <= sw(1);
```

- Put a 300 Ohm + resistor between the two pins (to limit the current if both pins are driven at once)
- Put a voltmeter across the resistor.
- Play around with the design
  - What is the highest voltage you can over the resistor?
  - How much power is this ( remember P=V^2/R)
- using a third slide switch decide which of the pins will be in high-Z mode. Something like:

```
process(sw)
begin
  if sw(2) = '1' then
    pmod(0) <= 'Z';
    pmod(1) <= sw(1);
  else
    pmod(0) <= sw(0);
    pmod(1) <= 'Z';
  end if;
end process;
```

- Play around with it.
  - What is the highest voltage you can get over the resistor now?
  - How much power is this?

# Chapter 26

# Closing

Sorry! All finished! Apart from the advanced feature of the I/O blocks (such as DDR2 input and outputs) you have pretty much played with all the features of the Spartan 3E.

So if you are still keen to learn more:

- have a read through the Xilinx AppNotes library. The ones on creative uses of BRAM and MULT18s is full of good ideas
- read through full Spartan 3E User Guide - it will make some sense now.
- create a system using the PicoBlaze embedded processor
- sell or gift your development board to a friend and move up to one with off-chip RAM, ROM, DACs, ADCs, Ethernet...
- try a different FPGA vendor's board - that will really make your head hurt
- Have a go at building something really nifty

If there is something I have missed or you want to say thanks send me an email - or maybe send me postcard to me at 370 Ellesmere Junction Road, Springston 7616, Canterbury, New Zealand. It will be fun to see if I actually get any cards!

# Chapter 27

# The complete Papilio One constraint file

```
##############################################################################
## BPC3003_2.03+.ucf
##
## Author: Jack Gassett
##
## Details: http://gadgetforge.gadgetfactory.net/gf/project/butterfly_one/
##
## Contains assignment and iostandard information for
## all used pins as well as timing and area constraints for Papilio One 2.03 and higher
    boards. Papilio One boards started using 32Mhz oscillators at version 2.02 and above.
##
##############################################################################

# Crystal Clock - use 32MHz onboard oscillator
NET "clk" LOC = "P89" | IOSTANDARD = LVCMOS25 | PERIOD = 31.25ns ;

# Wing1 Column A
NET "W1A<0>" LOC = "P18" ;      # LogicStart 7Seg anode(0)
NET "W1A<1>" LOC = "P23" ;      # LogicStart 7seg Decimal Point
NET "W1A<2>" LOC = "P26" ;      # LogicStart 7Seg anode(1)
NET "W1A<3>" LOC = "P33" ;      # LogicStart 7seg segment E
NET "W1A<4>" LOC = "P35" ;      # LogicStart 7seg segment F
NET "W1A<5>" LOC = "P40" ;      # LogicStart 7seg segment C
NET "W1A<6>" LOC = "P53" ;      # LogicStart 7seg segment D
NET "W1A<7>" LOC = "P57" ;      # LogicStart 7seg segment A
NET "W1A<8>" LOC = "P60" ;      # LogicStart 7seg anode(2)
NET "W1A<9>" LOC = "P62" ;      # LogicStart 7seg segment G
NET "W1A<10>" LOC = "P65" ;     # LogicStart 7seg segment B
NET "W1A<11>" LOC = "P67" ;     # LogicStart 7seg anode(3)
NET "W1A<12>" LOC = "P70" ;     # LogicStart A2D SPI_CS
NET "W1A<13>" LOC = "P79" ;     # LogicStart A2D SPI_DOUT
NET "W1A<14>" LOC = "P84" ;     # LogicStart A2D SPI_DIN
NET "W1A<15>" LOC = "P86" ;     # LogicStart A2D SPI_SCLK

# Wing1 Column B
NET "W1B<0>" LOC = "P85" ;      # LogicStart vsync
NET "W1B<1>" LOC = "P83" ;      # LogicStart hsync
NET "W1B<2>" LOC = "P78" ;      # LogicStart blue1
NET "W1B<3>" LOC = "P71" ;      # LogicStart blue2
NET "W1B<4>" LOC = "P68" ;      # LogicStart green0
NET "W1B<5>" LOC = "P66" ;      # LogicStart green1
NET "W1B<6>" LOC = "P63" ;      # LogicStart green2
NET "W1B<7>" LOC = "P61" ;      # LogicStart red0
NET "W1B<8>" LOC = "P58" ;      # LogicStart red1
```

```
NET "W1B<9>" LOC = "P54" ;      # LogicStart red2
NET "W1B<10>" LOC = "P41" ;     # LogicStart audio
NET "W1B<11>" LOC = "P36" ;     # LogicStart joystick right
NET "W1B<12>" LOC = "P34" ;     # LogicStart joystick left
NET "W1B<13>" LOC = "P32" ;     # LogicStart joystick down
NET "W1B<14>" LOC = "P25" ;     # LogicStart Joystick up
NET "W1B<15>" LOC = "P22" ;     # LogicStart Joystick Select

# Wing2 Column C
NET "W2C<0>" LOC = "P91" ;       # LogicStart Switch 7
NET "W2C<1>" LOC = "P92" ;       # LogicStart Switch 6
NET "W2C<2>" LOC = "P94" ;       # LogicStart Switch 5
NET "W2C<3>" LOC = "P95" ;       # LogicStart Switch 4
NET "W2C<4>" LOC = "P98" ;       # LogicStart Switch 3
NET "W2C<5>" LOC = "P2" ;        # LogicStart Switch 2
NET "W2C<6>" LOC = "P3" ;        # LogicStart Switch 1
NET "W2C<7>" LOC = "P4" ;        # LogicStart Switch 0
NET "W2C<8>" LOC = "P5" ;        # LogicStart LED 7
NET "W2C<9>" LOC = "P9" ;        # LogicStart LED 6
NET "W2C<10>" LOC = "P10" ;      # LogicStart LED 5
NET "W2C<11>" LOC = "P11" ;      # LogicStart LED 4
NET "W2C<12>" LOC = "P12" ;      # LogicStart LED 3
NET "W2C<13>" LOC = "P15" ;      # LogicStart LED 2
NET "W2C<14>" LOC = "P16" ;      # LogicStart LED 1
NET "W2C<15>" LOC = "P17" ;      # LogicStart LED 0

## RS232
NET "rx"  LOC = "P88" | IOSTANDARD = LVCMOS25 ;
NET "tx"  LOC = "P90" | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = SLOW ;
```

# Chapter 28

# The complete Basys2 constraint file

```
# This file is a general .ucf for Basys2 rev C board
# To use it in a project:
# - remove or comment the lines corresponding to unused pins
# - rename the used signals according to the project

# clock pin for Basys2 Board
NET "mclk" LOC = "B8"; # Bank = 0, Signal name = MCLK
NET "uclk" LOC = "M6"; # Bank = 2, Signal name = UCLK
NET "mclk" CLOCK_DEDICATED_ROUTE = FALSE;
NET "uclk" CLOCK_DEDICATED_ROUTE = FALSE;

# Pin assignment for EppCtl
# Connected to Basys2 onBoard USB controller
NET "EppAstb" LOC = "F2"; # Bank = 3
NET "EppDstb" LOC = "F1"; # Bank = 3
NET "EppWR"       LOC = "C2"; # Bank = 3

NET "EppWait" LOC = "D2"; # Bank = 3


NET "EppDB<0>" LOC = "N2"; # Bank = 2
NET "EppDB<1>" LOC = "M2"; # Bank = 2
NET "EppDB<2>" LOC = "M1"; # Bank = 3
NET "EppDB<3>" LOC = "L1"; # Bank = 3
NET "EppDB<4>" LOC = "L2"; # Bank = 3
NET "EppDB<5>" LOC = "H2"; # Bank = 3
NET "EppDB<6>" LOC = "H1"; # Bank = 3
NET "EppDB<7>" LOC = "H3"; # Bank = 3


# Pin assignment for DispCtl
# Connected to Basys2 onBoard 7seg display
NET "seg<0>" LOC = "L14"; # Bank = 1, Signal name = CA
NET "seg<1>" LOC = "H12"; # Bank = 1, Signal name = CB
NET "seg<2>" LOC = "N14"; # Bank = 1, Signal name = CC
NET "seg<3>" LOC = "N11"; # Bank = 2, Signal name = CD
NET "seg<4>" LOC = "P12"; # Bank = 2, Signal name = CE
NET "seg<5>" LOC = "L13"; # Bank = 1, Signal name = CF
NET "seg<6>" LOC = "M12"; # Bank = 1, Signal name = CG
NET "dp" LOC = "N13"; # Bank = 1, Signal name = DP

NET "an<3>" LOC = "K14"; # Bank = 1, Signal name = AN3
NET "an<2>" LOC = "M13"; # Bank = 1, Signal name = AN2
NET "an<1>" LOC = "J12"; # Bank = 1, Signal name = AN1
```

```
NET "an<0>" LOC = "F12"; # Bank = 1, Signal name = AN0

# Pin assignment for LEDs
NET "Led<7>" LOC = "G1" ; # Bank = 3, Signal name = LD7
NET "Led<6>" LOC = "P4" ; # Bank = 2, Signal name = LD6
NET "Led<5>" LOC = "N4" ;  # Bank = 2, Signal name = LD5
NET "Led<4>" LOC = "N5" ;  # Bank = 2, Signal name = LD4
NET "Led<3>" LOC = "P6" ; # Bank = 2, Signal name = LD3
NET "Led<2>" LOC = "P7" ; # Bank = 3, Signal name = LD2
NET "Led<1>" LOC = "M11" ; # Bank = 2, Signal name = LD1
NET "Led<0>" LOC = "M5" ;  # Bank = 2, Signal name = LD0

# Pin assignment for SWs
NET "sw<7>" LOC = "N3";  # Bank = 2, Signal name = SW7
NET "sw<6>" LOC = "E2";  # Bank = 3, Signal name = SW6
NET "sw<5>" LOC = "F3";  # Bank = 3, Signal name = SW5
NET "sw<4>" LOC = "G3";  # Bank = 3, Signal name = SW4
NET "sw<3>" LOC = "B4";  # Bank = 3, Signal name = SW3
NET "sw<2>" LOC = "K3";  # Bank = 3, Signal name = SW2
NET "sw<1>" LOC = "L3";  # Bank = 3, Signal name = SW1
NET "sw<0>" LOC = "P11";  # Bank = 2, Signal name = SW0

NET "btn<3>" LOC = "A7";  # Bank = 1, Signal name = BTN3
NET "btn<2>" LOC = "M4";  # Bank = 0, Signal name = BTN2
NET "btn<1>" LOC = "C11"; # Bank = 2, Signal name = BTN1
NET "btn<0>" LOC = "G12"; # Bank = 0, Signal name = BTN0

# Loop back/demo signals
# Pin assignment for PS2
NET "PS2C"    LOC = "B1"   | DRIVE = 2  | PULLUP ; # Bank = 3, Signal name = PS2C
NET "PS2D"    LOC = "C3"   | DRIVE = 2  | PULLUP ; # Bank = 3, Signal name = PS2D

# Pin assignment for VGA
NET "HSYNC"   LOC = "J14"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = HSYNC
NET "VSYNC"   LOC = "K13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = VSYNC

NET "OutRed<2>"  LOC = "F13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = RED2
NET "OutRed<1>"  LOC = "D13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = RED1
NET "OutRed<0>"  LOC = "C14"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = RED0
NET "OutGreen<2>"  LOC = "G14"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = GRN2
NET "OutGreen<1>"  LOC = "G13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = GRN1
NET "OutGreen<0>"  LOC = "F14"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = GRN0
NET "OutBlue<2>"  LOC = "J13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = BLU2
NET "OutBlue<1>"  LOC = "H13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = BLU1

# Loop Back only tested signals
NET "PIO<72>" LOC = "B2"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA1
NET "PIO<73>" LOC = "A3"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA2
NET "PIO<74>" LOC = "J3"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA3
NET "PIO<75>" LOC = "B5"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JA4

NET "PIO<76>" LOC = "C6"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB1
NET "PIO<77>" LOC = "B6"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB2
NET "PIO<78>" LOC = "C5"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB3
NET "PIO<79>" LOC = "B7"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JB4

NET "PIO<80>" LOC = "A9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC1
NET "PIO<81>" LOC = "B9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC2
NET "PIO<82>" LOC = "A10" | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC3
NET "PIO<83>" LOC = "C9"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JC4

NET "PIO<84>" LOC = "C12"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JD1
```

```
NET "PIO<85>" LOC = "A13"  | DRIVE = 2  | PULLUP ; # Bank = 2, Signal name = JD2
NET "PIO<86>" LOC = "C13"  | DRIVE = 2  | PULLUP ; # Bank = 1, Signal name = JD3
NET "PIO<87>" LOC = "D12"  | DRIVE = 2  | PULLUP ; # Bank = 2, Signal name = JD4
```