FPGAs!? Now What?

*Learning FPGA Design
with the XuLA Board*
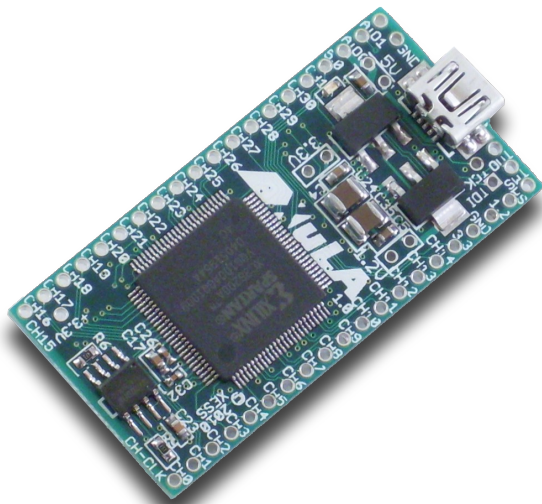
Dave Vandenbout, Ph.D.
XESS Corporation

FPGAs!? Now What?
TUT001 (V1.0) February 28, 2013

The following table shows the revision history for this document.

Date	Version	Revision
06/24/2011	1.0	Initial release of Chapters 1-4.
07/10/2011	1.0	Added Chapter 5.
08/29/2011	1.0	Added Chapter 6.
03/01/2012	1.0	Added Chapter 7.
09/04/2012	1.0	Changed "chose" to "choose" on page 29.
12/30/2012	1.0	Changed link to XSTOOLs setup file.
12/30/2012	1.0	Added note to use 32-bit Python and avoid the 64-bit version.
01/24/2013	1.0	Revised Python scripts to use the new XsTools package.
02/22/2013	1.0	Added Chapter 8.

Table of Contents

Preface

There are numerous requests in Internet forums that go something like this:

"I am new to using FPGAs. What are they? How do I start? Is there a tutorial and some cheap/free tools I can use to learn more?"

The short answer is "Yes".

The long answer is this book. It will briefly describe FPGAs and then show you how to apply them to your problems using a low-cost board and some free software. My discussion will be oriented towards using Xilinx FPGAs, but most of what I'll say is applicable to other brands of FPGAs.

But first, a little history...

Writing Tutorials Sucks Donkeys!

Yeah, I mean it and I have the experience to back it up.

I've written a lot of documentation over the past twenty years. It's in the nature of my business. I need to write manuals and application notes for my FPGA board customers. I need to generate content for my web pages. I need to write documentation for *myself* just so I can remember what I've done and where I left off.

But the documentation I hate writing most of all is *tutorials*. With a tutorial, you can't assume your reader knows much of anything. You have to build from the ground up. That requires a lot of text, figures, pictures, design examples, etc. And you have to polish and proofread it more thoroughly to make sure its meaning is clear because, if it isn't, someone is going to read it the wrong way and give you a bunch of static about it.

Allow me to take you on a trip down Memory Lane and recount some of the tutorials I've written:

FPGA Workout - 236 pages: I wrote this book back in 1994. It showed how to build electronics using the Intel FLEXlogic FPGAs. (You didn't know Intel built FPGAs? Seems that nobody else did either - they exited the FPGA business around 1995.) I self-published this and had 2000 of them printed. I sold 1500, gave 300 away (one to every EE department in the USA), and watched the last 200 get ground under a bulldozer's tracks in the Wake County landfill (I just couldn't stand having them around anymore).

The Practical Xilinx Designer Lab Book - 300 pages: I wrote this for Prentice Hall back in 1997. They were publishing a student package in cooperation with Xilinx that included this book along with the Foundation FPGA design software. It was a good product at the time. My book covered how to build electronics using Xilinx FPGAs. It

was a real bear working with Prentice Hall after self-publishing my first book. I created and proofread all the text and figures, then Prentice Hall *re-entered all that material using Framemaker* and I had to proofread it all again to catch any transcription errors. (To their credit, there weren't very many. Then again, I wasn't looking *that* hard.)

myCSoC - 213 pages: I wrote this e-book back in 2000. It discussed how to construct electronic systems around the Triscend TE505 chip. (The TE505 combined an 8051 microcontroller with an FPGA on a single chip.) I distributed this book as PDFs from my website and used it as a sales tool for the myCSoC Kit . That came to an end when Triscend went out of business around 2003 and Xilinx gobbled up their dessicated remains. (I still have five-hundred unused TE505 chips from that venture. Anybody want them? Anybody?)

The Practical Xilinx Designer Lab Book Version 1.5 - 450 pages: It wasn't enough to do this once; I went ahead and did a revision! And I added an extra 150 pages of material on - can you guess? - designing electronics using FPGAs. I tried to make a deal with Prentice Hall wherein I would write the print revision for free, and they would grant me the electronic rights to distribute it on the web. This was 1999 and they just couldn't get their heads around this, but they knew they wanted no part of it. Instead, they just paid me $15,000 for the re-write and told me to drop all the crazy talk. After the book was written, they packaged it with some CDROMs which - instead of containing the Xilinx software - contained test data for an anesthesiology text book! Yeesh, the customer support calls I got! Looking back on it, I realized that all the whipped cream in the world would never make this turd taste any better.

Pragmatic Logic Design With Xilinx Foundation 2.1i - 394 pages: This was what I originally wanted to do for Prentice Hall. An online book with a separate PDF for each chapter. Plenty of room to add as much text and as many figures as were needed because it didn't have to fit into a fixed amount of paper. Easily edited and changed to correct any errors that were found. I got five chapters of this text finished. Then Xilinx released a version of their free WebPACK tools that targeted FPGAs (instead of just CPLDs) and the market for the not-free Foundation software vanished. So did any incentive I had to finish this book.

Introduction to WebPACK X.Y - from 78 pages up to 130 pages: I wrote this tutorial to show how to use the free Xilinx ISE WebPACK software, which had a markedly different user interface than the Foundation software. I wrote a version of the tutorial for WebPACK 1.5 and then 3.1 (these only supported Xilinx CPLDs), then two re-writes for 4.1 (one for CPLDs and another for FPGAs), another re-write for 5.2 (FPGAs only, this time), two re-writes for 6.1 (one targeting the XSA-100 board, and another for the XSB-300E board), one more minor re-write for 6.3 (because of the introduction of the new XSA-3S1000 board), another re-write for 8.1, and a final re-write for ISE WebPACK 10. Xilinx is up to ISE 13 now and, thankfully, it still operates pretty much the same as version 10 because I haven't had the energy to re-write the tutorial. It still serves as a pretty good introduction to the Xilinx software.

All told, that's over two-thousand pages of tutorials. That's a lot, regardless of whether it's mostly roses or shit. What I hated most about doing the tutorials was the boring sameness of it. Each looks like a rehash of the previous one. So I cringe when I think about writing another (hence my three year hiatus from re-doing the WebPACK tutorial).

Here I Go Again

But it looks like I *am* writing another one. The XuLA board needs a tutorial; it's just too different from any of my other boards and the people who might use it probably have no experience with FPGAs. So they need something to help them get started. But, in order to get *me* started, I'm going to write this tutorial a bit differently so it won't be so boring. Here are the guidelines I set for myself:

I won't re-invent the wheel. There's no reason to re-write everything about digital logic design just so I can introduce you to FPGAs – the web is full of good information. I'll add links to good stuff I find to cut my workload and to produce a better text in less time.

I won't fence it in. What I really mean is: *steal this book!.* I'm placing the source directory for it on Github. You can copy it, modify it, sell it - I don't care. *I want you to do this!* Please take it, add your own stuff and make it better. All I ask is that you respect the terms of the Creative Commons license I applied to this work. Basically, that means don't remove the names/affiliations of anyone else who worked on this, and provide others with the same rights to it that you got. (But check the actual license to get the fine points.)

I won't let others fence it in. I'm using FOSS like LibreOffice and Inkscape to write this book. That removes some barriers for anyone who wants to work on it because they won't have to pay for the tools. And it keeps the source files in non-proprietary formats.

I won't be a prisoner of perfection. I've often held onto things too long before releasing them, trying to remove every error I can find. That's not the case with this book – you'll find spelling and grammar errors, subject/verb disagreements, inconsistent formatting, hand-drawn figures and lots of other stuff. But this won't matter to someone who really needs the material. (For those who *do* care about the finer points, you can tell me what errors you find and I'll get around to fixing them, or you can get the source and fix them for me.)

I'll use the right medium for the right message. Text and figures are great for a tutorial so you can read along while you actually do it. Screencasts and videos are almost always more difficult to follow, especially all the diddling-about with typing text and clicking icons that goes on with FPGA design tools. But there is a place for these, particularly when demonstrating the actual operation of some circuitry.

What This Is and *Is Not*

This book discusses how to use the Xilinx ISE WebPACK software to build FPGA designs for the XuLA FPGA board. Along the way, you'll see:

- How to start an FPGA project.
- How to target a design to a particular type of FPGA.
- How to describe a logic circuit using VHDL and/or schematics.
- How to detect and fix VHDL syntactical errors.
- How to synthesize a netlist from a circuit description.
- How to simulate a circuit.
- How to implement the netlist for an FPGA.
- How to check device utilization and timing for an FPGA.
- How to generate a bitstream for an FPGA.
- How to download a bitstream into an FPGA.
- How to test the programmed FPGA.
- How to detect and fix errors.

I'll also delve into things like:

- How to build hierarchical designs.
- How to build state machines.
- How to build mixed-mode designs using VHDL, Verilog and schematics.

- How to use Digital Clock Managers.

- How to use block RAMs.

- How to use multipliers.

- How to use IP and soft cores.

- How to use external components like SDRAM, VGA, audio, ADC, DAC.

That said, here are some of the things this book will *not* teach you (I may be wrong here; this book isn't even *written* yet, plus who knows what others may add):

- It will not teach you how to use VHDL. There are plenty of good VHDL textbooks already.

- It will not teach you how to choose the best type of FPGA for your particular design. I'm oriented toward using the FPGA on the XuLA board.

- It will not show you every feature of the ISE software and discuss how to set every option and property. This software already has a good help system and extensive manuals, so those should suffice.

In short, this book will get you started using the XILINX ISE software for doing FPGA designs with the XuLA board. After you finish, you should be able to move on to more advanced topics.

Really Important Stuff!

- Everything for this book is on Github! That includes all the FPGA design examples. I won't be making any zipped archives since they would get out-of-date as changes and corrections are committed to Github. So go to https://github.com/xesscorp/FpgasNowWhat if you need anything related to this book.

- All the FPGA design examples in this book are targeted to the XuLA-200 board. You can still use them if you own a XuLA-50 board – just change the target FPGA in the project from xc3s200a-4vq100 to xc3s50a-4vq100.

C.1 *"I know nothing about FPGAs. Now what?"*

What Is Programmable Logic?

In the beginning (OK, actually in the 60's) there were discrete logic chips. Systems were built from lots of individual ANDs, ORs, flip-flops, etc. with a spaghetti-like maze of wiring between them. It was difficult to modify such a system after you built it. Jeez, after a week or two it was difficult to remember what each of the chips was for!
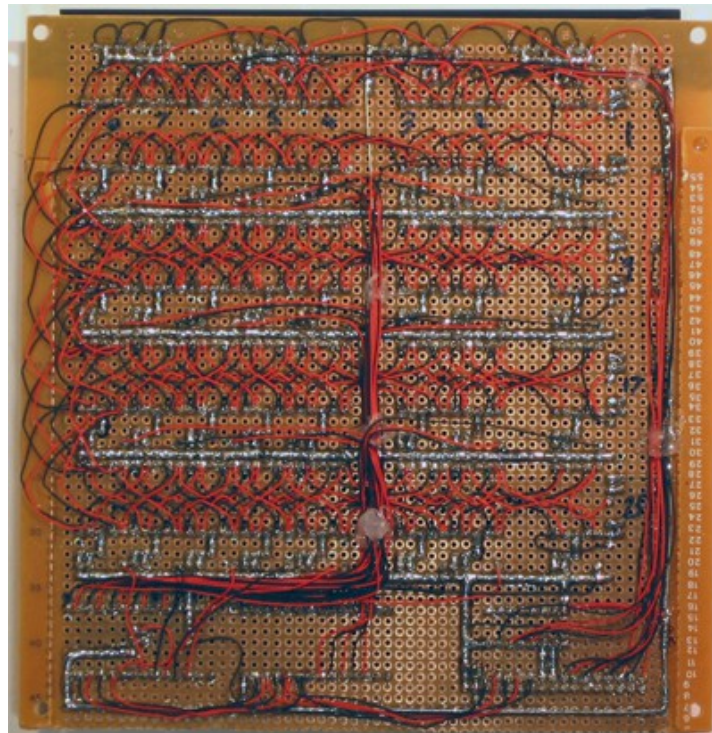

Spaghetti wiring (courtesy of www.franksworkshop.com.au).

Manufacturing a system like this took a lot of time because each design change required that the wiring be redone which usually meant building a new printed circuit board. The chip makers addressed this problem by placing an unconnected array of AND-OR gates in a single chip called a *programmable logic device* (PLD). The PLD contained an array of fuses that could be blown open or left closed to connect various inputs to each AND gate. You could program a PLD with a set of Boolean sum-of-product equations to perform the logic functions needed in your system. You could change the function of a design by removing the PLDs, blowing a new fuse pattern into them, and then placing them back into the circuit board. This reduced the need to change the actual wiring of the printed circuit boards which held them.
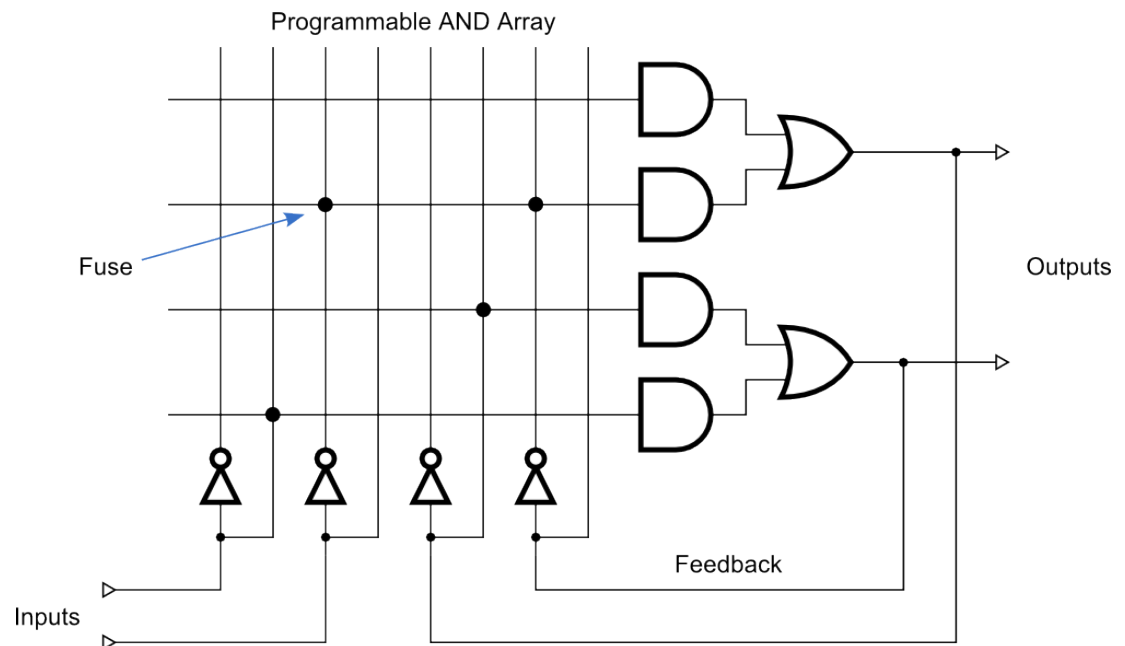
Simple PLDs could only handle up to 10–20 logic equations, so you couldn't fit a very large logic design into just one of them. You had to figure out how to break your larger designs apart and fit them into a set of PLDs. This was time-consuming and meant you had to interconnect the PLDs with wires. The wires were a big no-no because eventually you would make some design change that couldn't be handled just by reprogramming the PLDs and then you would have to build a new circuit board. The chip makers came to the rescue again by building much larger programmable chips called *complex programmable logic devices* (CPLDs) and *field-programmable gate arrays* (FPGAs). With these, you could essentially get a complete system onto a single chip.

A CPLD contains a bunch of PLD blocks like the one shown above, but their inputs and outputs are connected together by a *global interconnection matrix*. So a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed.
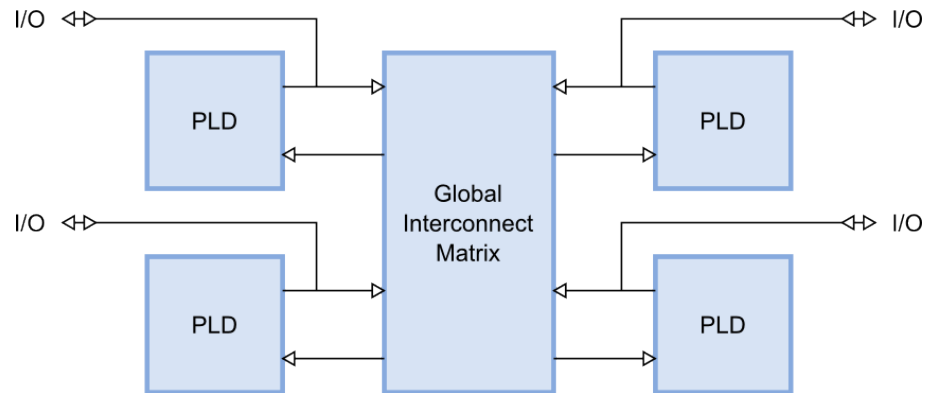
An FPGA takes a different approach. It has a bunch of simple, *configurable logic blocks* (CLBs) interspersed within a *switching matrix* that can rearrange the interconnections between the them. Each logic block is individually programmed to perform a logic function (such as AND, OR, XOR, etc.) and then the switches are programmed to connect the blocks so that the complete logic functions are implemented.
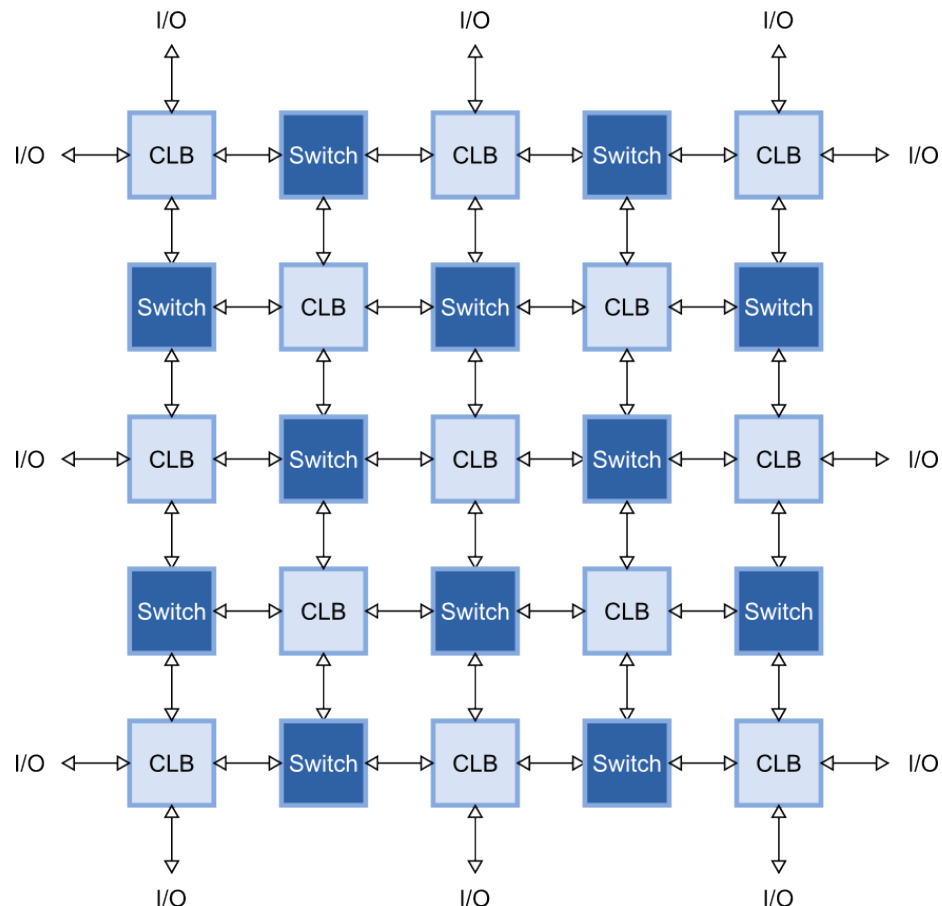
CPLD and FPGA manufacturers use a variety of methods to make the connections between logic blocks. Some make chips with *fuses* or *anti-fuses* that are programmed by

passing a large current through them. These types of CPLDs and FPGAs are *one-time programmable* (OTP) because you can't rewire them internally once the fuses are blown.

Other manufacturers make the connections using pass transistors that are opened or closed by storing a charge on their gate electrodes using a high-voltage pulse. This type of programmable device resembles an EPROM or EEPROM: you can erase it and then place it in a special programmer socket and reprogram it. That's fine unless you have the CPLD or FPGA soldered into a circuit board.

Finally, some manufacturers use static RAM or Flash bits to control the pass transistors for each interconnection. By loading each bit with a 1 or a 0, you can control whether the switch is closed or opened and, therefore, whether two logic elements are connected or not. CPLDs and FPGAs built using RAM/Flash switches can be reprogrammed without removing them from the circuit board. They are often said to be *in-circuit reconfigurable* or *in-circuit programmable*. They are different, however: a Flash-based FPGA always has its programming intact whether it is powered or not, but a RAM-based FPGA will be erased and stop functioning if it ever loses power. Even with this drawback, most FPGAs are RAM-based because the RAMs can be built on the chip using the same manufacturing process as for the CLBs and switches (that makes the chips cheaper to produce). The manufacturers also provide a separate, Flash-based chip that stores the programming switch settings externally and loads it into the FPGA whenever power is applied.

For the rest of this book, we'll concentrate on using RAM-based FPGAs from Xilinx. I'll introduce a little terminology because it will be important later on.

Within the CLBs of an FPGA are *look-up tables* (LUTs) that perform the actual logic operations. These are usually built as small RAMs that are loaded with a truth table. For example, an eight-bit RAM can perform a three-input AND operation if it is loaded as follows:
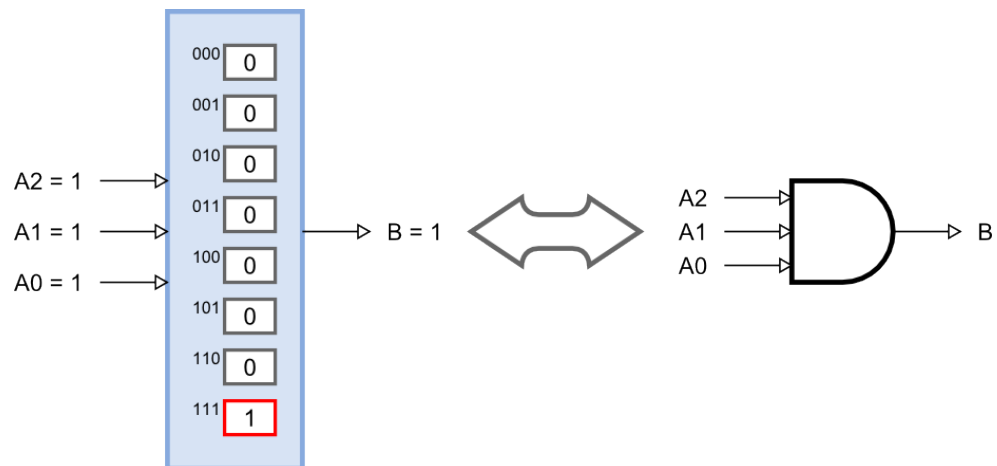
If the inputs for the AND gate are applied to the address pins of the RAM, then the output of the RAM will only be 1 when location 7 = 111 is selected. This is exactly how a three-input AND gate would operate. By setting the contents of the RAM in the right way, you can perform *any* logic function of three binary inputs. (Modern FPGAs use LUTs with four to six inputs.)

The output of a LUT can be used directly as an input to another LUT to make more complicated logic functions. Or it can be stored in a flip-flop for later use (there's usually at least one register for each LUT).

The switching matrix routes all the signals between the LUTs and registers and also to the *I/O blocks* (IOBs) around the perimeter of the FPGA chip. The IOBs connect selected internal signals of the FPGA to the outside world and contain special analog circuitry to

adjust the input and output voltage levels, insert small delays, and handle single-ended or differential signaling.

Over time, other types of more specialized circuit blocks have found their way into FPGAs: blocks of RAM, multipliers and DSP blocks, delay-locked and phase-locked loops, gigabit serial transceivers, and even complete microprocessors. But, at least when you're starting off, your main concern will be the LUTs and registers. These are the equivalent of the program and data memories in a processor: the values in the LUTs and flip-flops determine how your logic circuit functions, and your design must fit into the available number of LUTs and flip-flops within your target FPGA.

Also, like a microprocessor, you need a way to write programs for your FPGA. That's the subject of the next chapter.
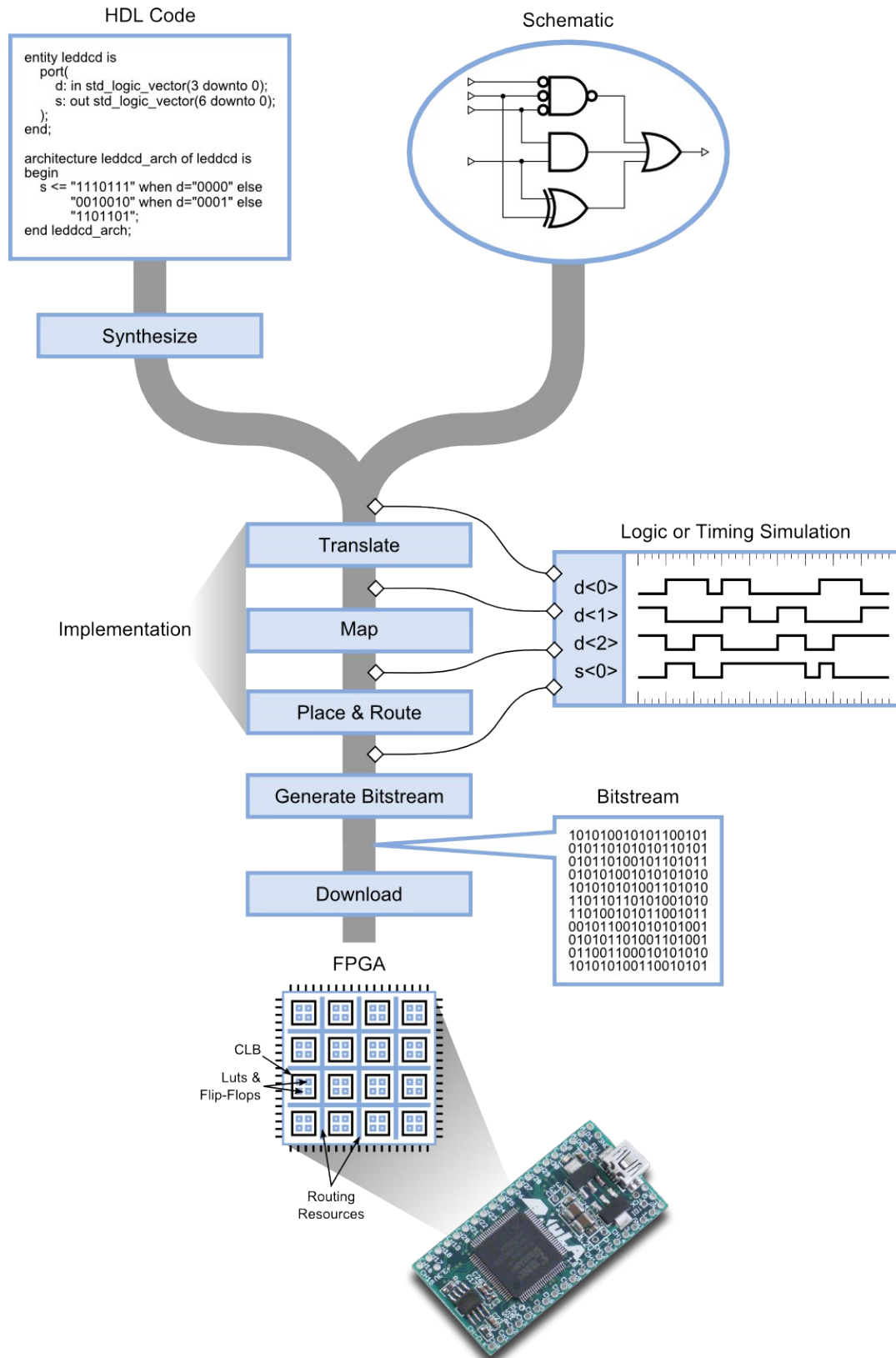
C.2 *"I have no compiler. Now what?"*

The Compilation Process: From Schematic to Bitstream

As you can imagine, figuring out which bits to set in the LUTs and the switching matrix in order to create a logic circuit is quite a chore. Not many people would want to delve into the details of this (probably the same ones who like to program in assembly). That's why the FPGA manufacturers provide development software that compiles a high-level description of a logic design and outputs a *bitstream* which sets the FPGA configuration bits appropriately.

The process the development software goes through to compile a design into a bitstream is depicted in the figure on the next page. Here's what happens:

1. You enter a description of your logic circuit using a *hardware description language* (HDL) such as VHDL or Verilog. Or you can also draw your design using a schematic editor. Or you can use a combination of the two.
2. A *logic synthesizer* transforms the HDL into a *netlist*. The netlist is just a description of the various logic gates in your design and how they are interconnected. (A schematic is already pretty close to a netlist, so it doesn't need as much work done on it as the HDL code.)
3. The *implementation phase* employs three different tools. A *translator* merges together one or more netlists along with any *design constraints* (don't worry about those now). This is fed to a *mapper* that combines gates in the netlist into groups that will fit efficiently into the LUTs of the FPGA. The gate groupings are sent to the *place & route tool* that assigns them to LUTs at various locations in the FPGA and then determines how to connect them together using the routing resources (wires) in the switching matrix. This part takes the most time as finding a placement that can be routed efficiently requires a lot of computation.
4. A *bitstream generator* takes the output of the implementation phase, combines it with a few other configuration settings, and outputs a binary bitstream. This bitstream (which, depending upon the size of the FPGA, can be many megabits in length) contains the truth-tables that will be loaded into the RAM of every LUT and the connection settings for the wiring matrix that will connect them.
5. At this point, a bitstream is just a bunch of 1s and 0s in a file on the computer. The *downloader* will transfer this file into a physical FPGA chip. In most cases, this chip is already mounted on a circuit board where it waits for the bitstream that will make it perform its intended function.

HDL Code

```
entity leddcd is
    port(
        d: in std_logic_vector(3 downto 0);
        s: out std_logic_vector(6 downto 0);
    );
end;

architecture leddcd_arch of leddcd is
begin
    s <= "1110111" when d="0000" else
         "0010010" when d="0001" else
         "1101101";
end leddcd_arch;
```

Schematic

Synthesize

Translate

Implementation

Map

Place & Route

Generate Bitstream

Download

Logic or Timing Simulation

d<0>
d<1>
d<2>
s<0>

Bitstream

```
1010100101011001011
0101101010101101011
0101101001011010111
0101010010101010100
1010101010011010101
1101101101010010101
1101001010110010111
0010110010101010010
0101011001011010010
0110011000101010101
1010101001100101011
```

FPGA

CLB

Luts &
Flip-Flops

Routing
Resources

After downloading, you can apply patterns of input signals (or *test vectors*) to the I/O pins of the FPGA to check the operation of your design. This is often an effective way to do testing, but sometimes it's difficult to set up a particular combination of signals and look deeply into the internals of your design to see why it may not be performing as you intended.

If that's the case, you can use a *simulator* to try out test examples and see how your circuit reacts before you actually load it into an FPGA. There are several places in the design flow where you can perform a simulation. If you capture the netlist before it enters the implementation tools, you can run a *logic simulation* that only tests the logical operations of the gates in your circuit. Such simulations are good for debugging the basic functions of your design.

As you tap the design flow at points further downstream, the netlist is augmented with more and more information about how your design will be placed into the FPGA. This allows you to perform a more realistic *timing simulation* that incorporates the effects of gate and wiring delays on the operation of your circuit. This is useful for detecting errors caused when signals arrive too quickly or slowly at their destinations. Because of this extra level of detail, timing simulations take longer to run than a pure logic simulation.

At this point you know the steps in the compilation process, but what you really want to know is how to get a compiler. That comes next.
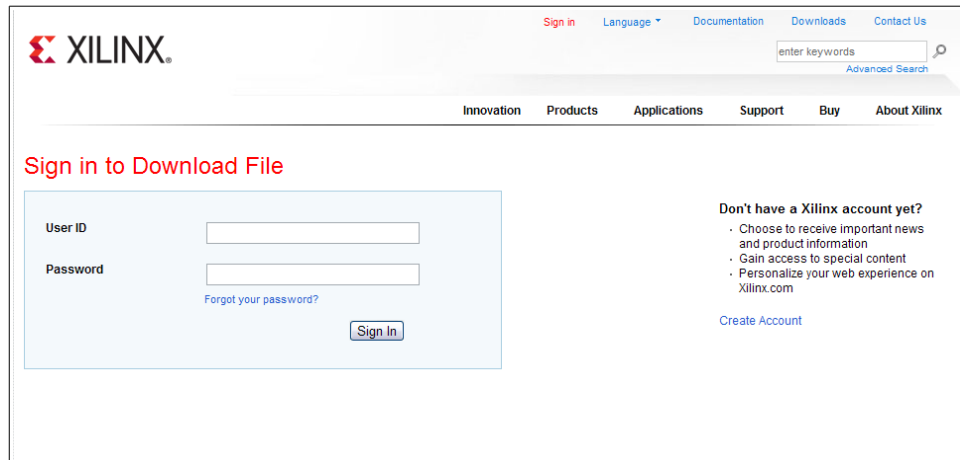
Getting and Installing Xilinx ISE WebPACK

Xilinx develops and sells their ISE FPGA tools, and they also distribute a free version called WebPACK. WebPACK won't generate bitstreams for the *really* large FPGAs and it lacks some special-function design tools, but it will perform all the functions I discussed in the previous section for FPGAs containing up to 75,000 LUTs with hundreds of I/O pins. That will be more than sufficient for this tutorial.

To get WebPACK, just do a search for "xilinx webpack download" (use Google, Bing, whatever). If you click on the first link you get, it will probably take you to the page shown below. The latest version of the software should be highlighted along the left side (it's currently 13.1). Click on the Full Installer link for whatever operating system you're using (either Windows or Linux, currently).

Before the download begins, you'll be presented with a sign-in screen. If you don't already have a Xilinx account, you'll have to create one. Then sign in using your ID and password.

Then you'll have to go through a screen that verifies all your contact information. Click on the **Next** button at the bottom and your download will begin. Now be prepared to wait an hour or two as the file is about 4 GB.

After the archive is downloaded, you can unpack it (that'll take a ten minutes) and run the installer. You'll go through a set of installation screens. Make sure you choose the WebPACK edition unless you want to pay money for one of the others or are willing to accept a time-limited license (usually 30 days).
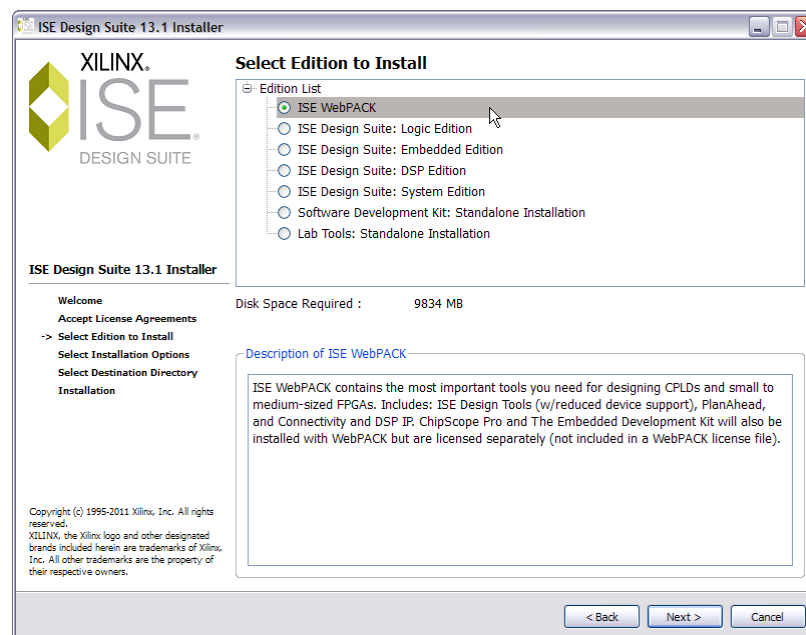
Also make sure to select the option to get a license key. (Yes, you will need a key even though WebPACK is free. The WebPACK license is not time-limited, so it will never expire.)
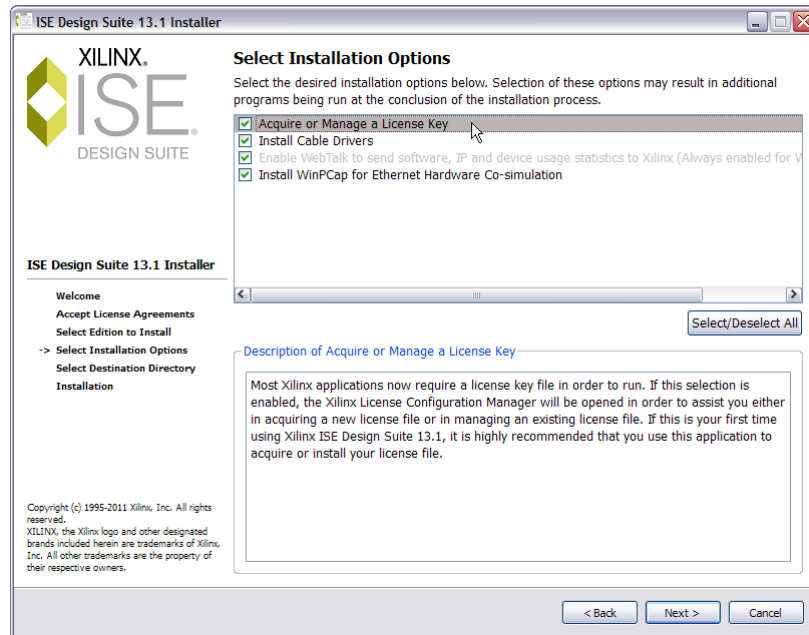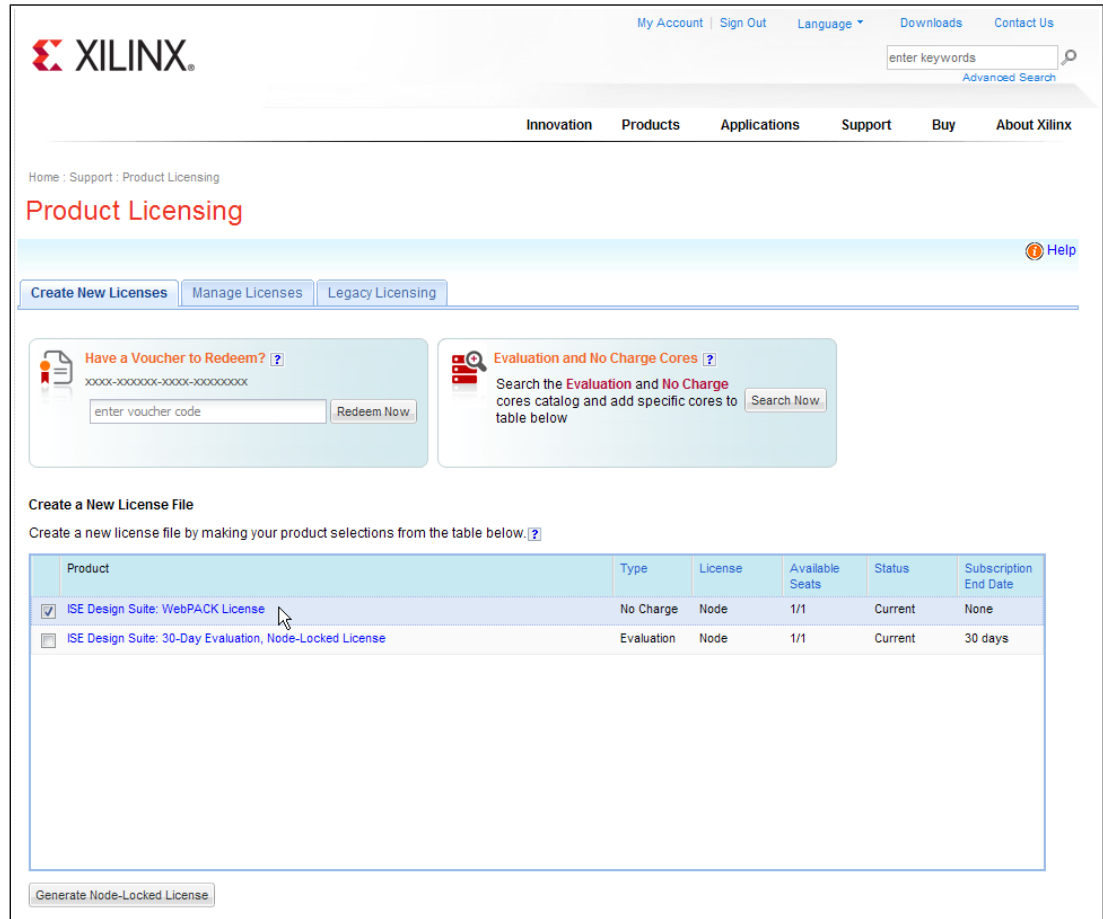
When the installation is almost done (that takes about 45 minutes), you'll see this screen appear. Select the WebPACK license as the one you want to get and click **Next**.
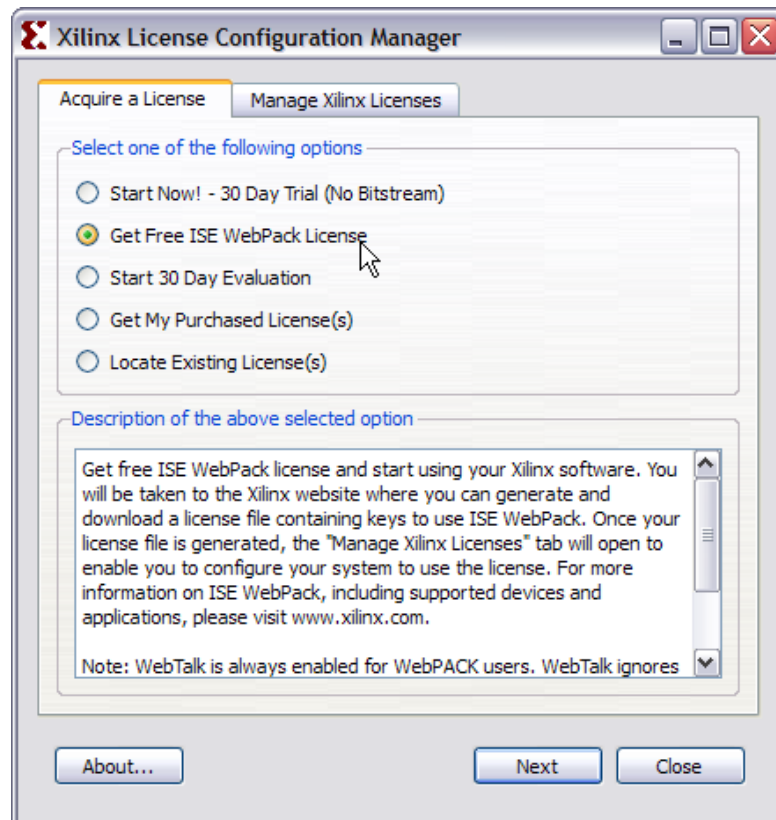
Then you'll get a screen showing the information that will be associated with your license.
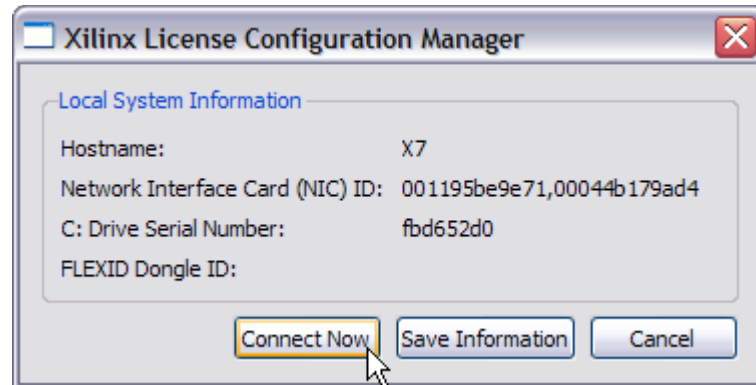
Click the **Connect Now** button to start the license generation process. Once you do, your browser will open a window to Xilinx and you'll have to login using your account information (again). Then you'll have to verify your contact information (again). Finally, you'll get to this screen. Select the WebPACK checkbox and click on the **Generate Node Locked License button**.

Next you'll see a couple of confirmation screens like the following:

After clicking the **Next** button in the final confirmation screen, you'll get an acknowledgment that your license has been emailed to you.

Once you've received the email, save the attached license file somewhere (but remember where you put it!). Then go back to the **Xilinx License Configuration Manager** screen and click on the **Manage Xilinx Licenses** tab. Enter the location of your license file into the **XILINXD_LICENSE_FILE** field and click on the **Set** button.
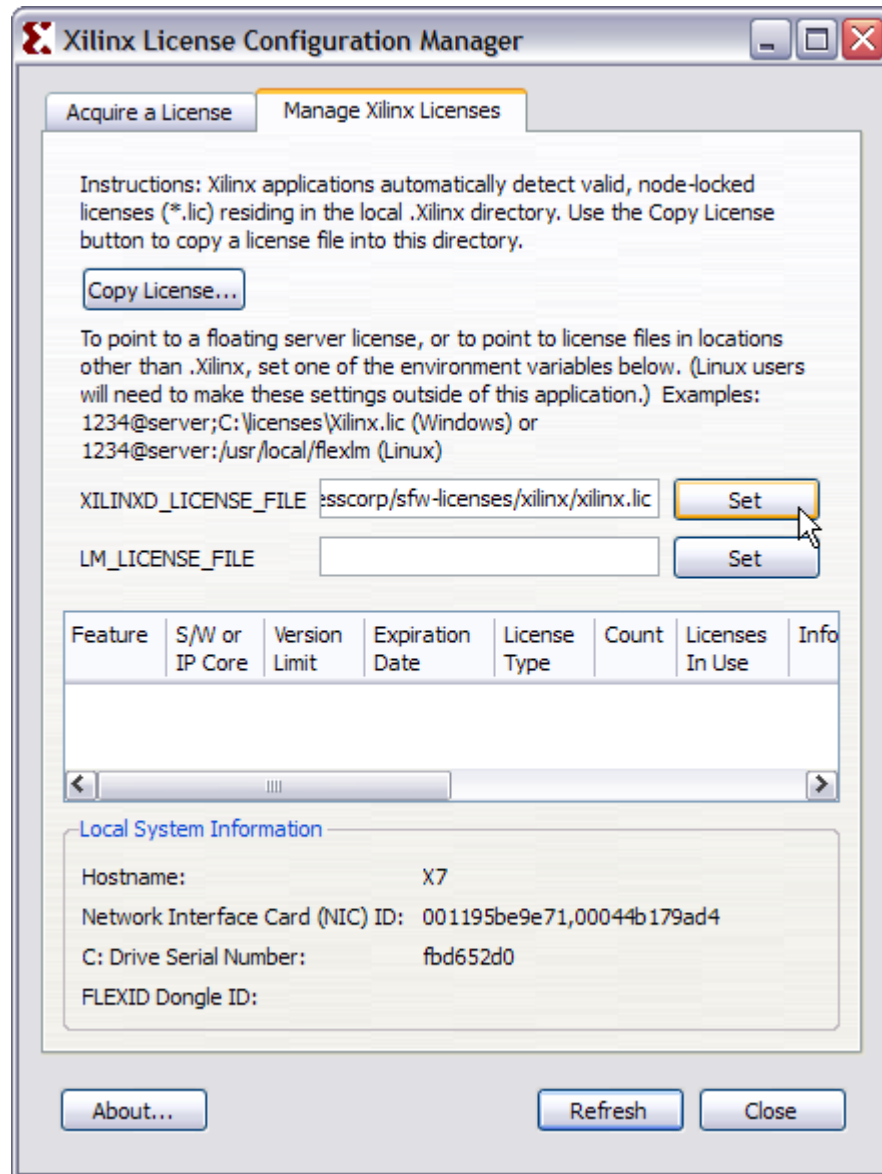
Once you click on the **Set** button, a list of the enabled features in the WebPACK software should appear. You're done setting up your license, so click on the **Close** button.
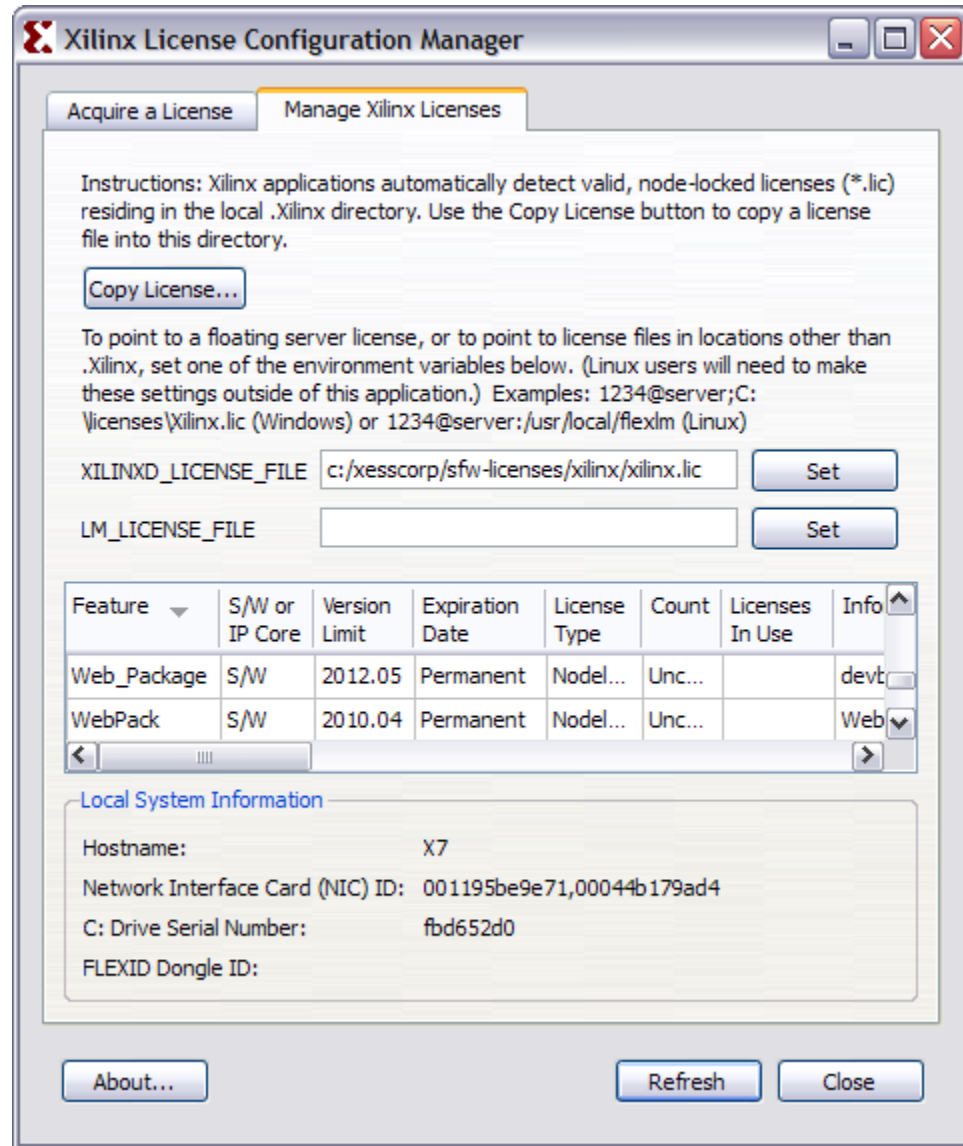
You can also click on the **Finish** button in the final window of the WebPACK installer. (You probably forgot all about that by now.)
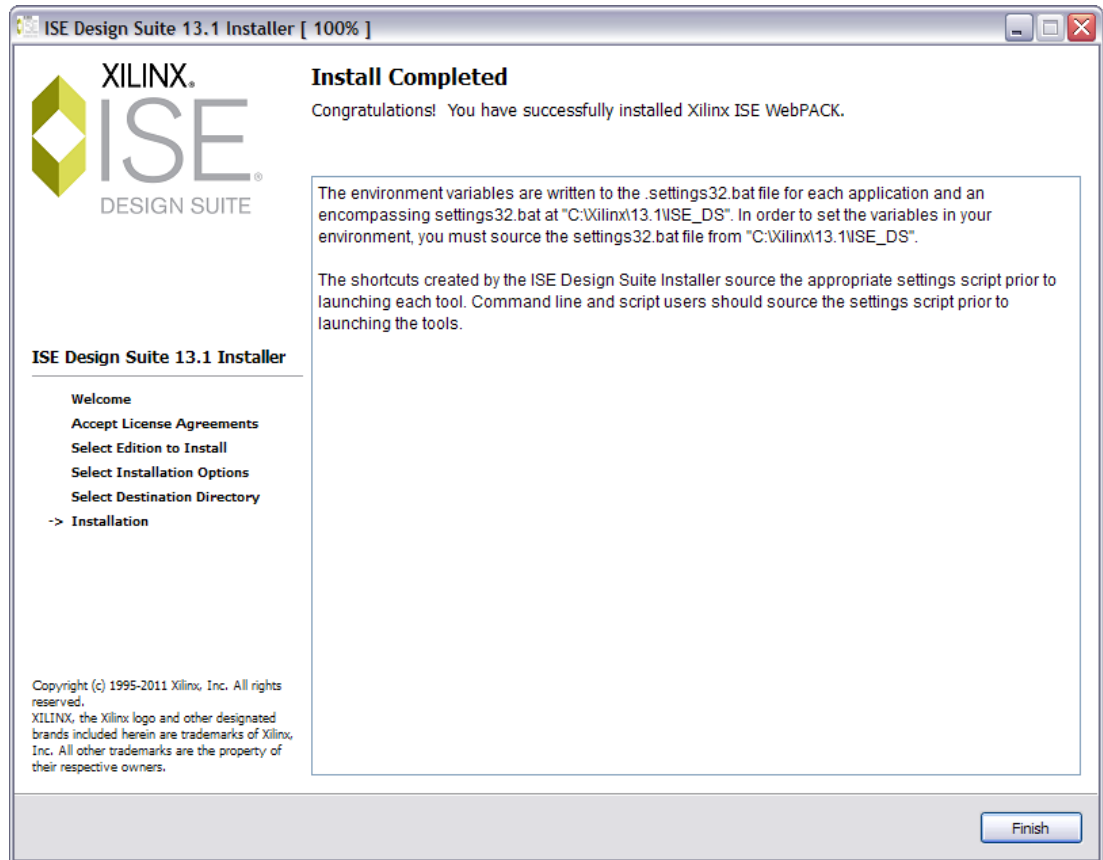
The WebPACK installation is done! You've got your party dress on, now it's time to dance!

_C.3_ _"I have a synthesizer._ _Now what?"_

The "Hello, World" of FPGAs: the LED Blinker

Now that your WebPACK tools are installed, it's time to do your first FPGA design: an LED blinker. This blinker will take a 12 MHz square-wave clock signal (you'll see why in the next chapter) and slow it down so an LED will turn on-and-off about once per second (i.e., 1 Hz).
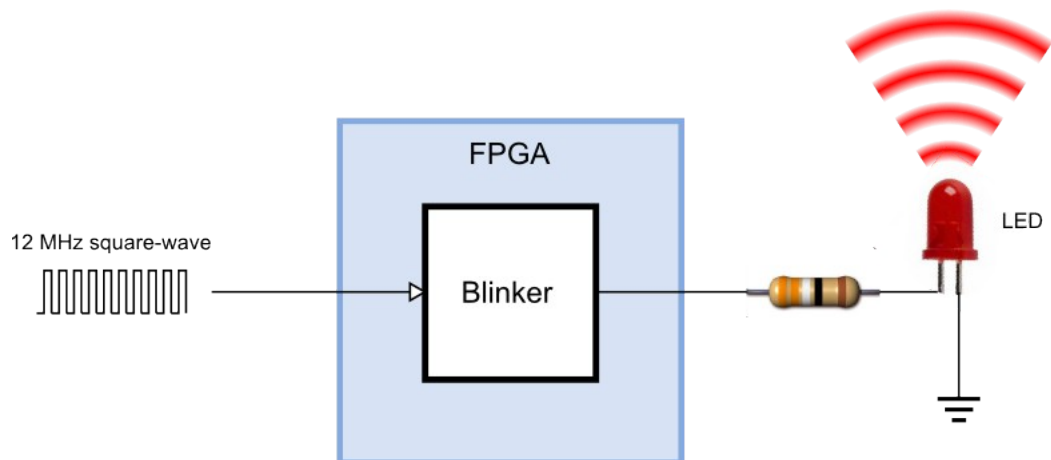
Starting a Design in WebPACK

You start WebPACK by double-clicking the **ISE 13.1** icon on the desktop. This will bring up an **ISE Project Navigator** window like the following one.
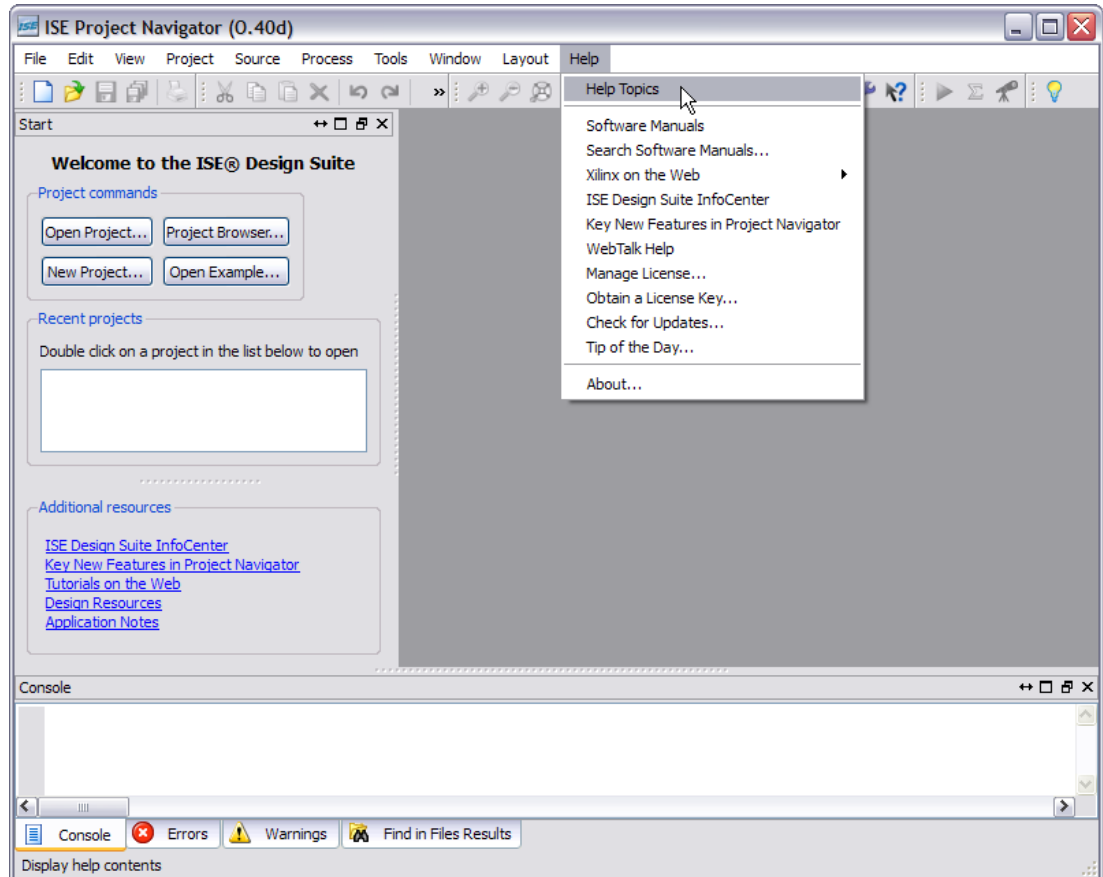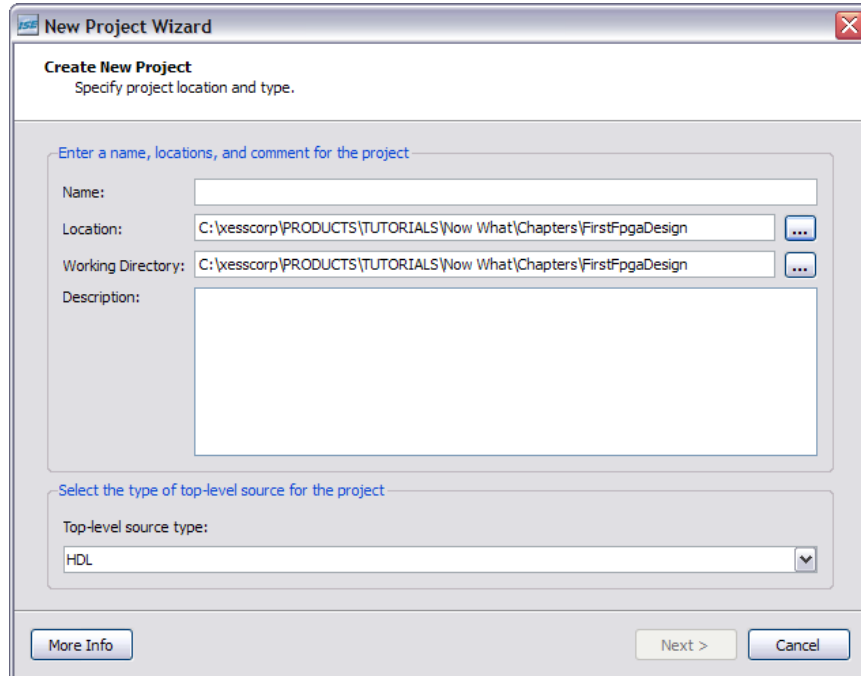
If this is your first time using Navigator, it's a good idea to click on the **Help** button and bring up the **Project Navigator Overview**. Read this to get an idea of what all the panes in the Navigator window are used for.
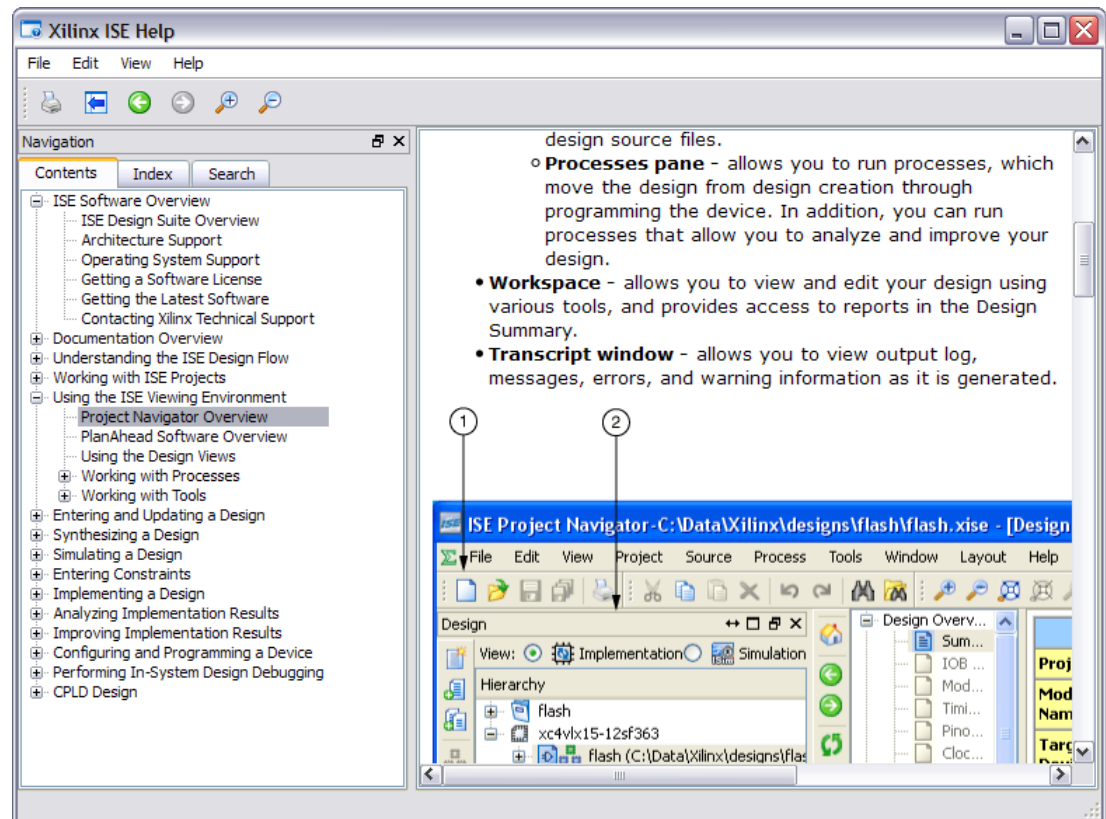
To start your LED blinker design, click on the **New Project...** button in the Navigator window.
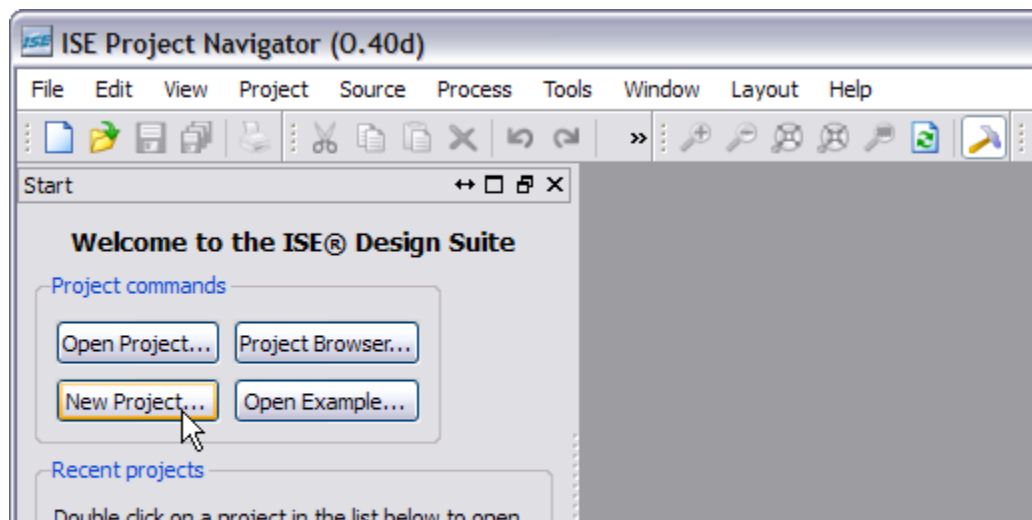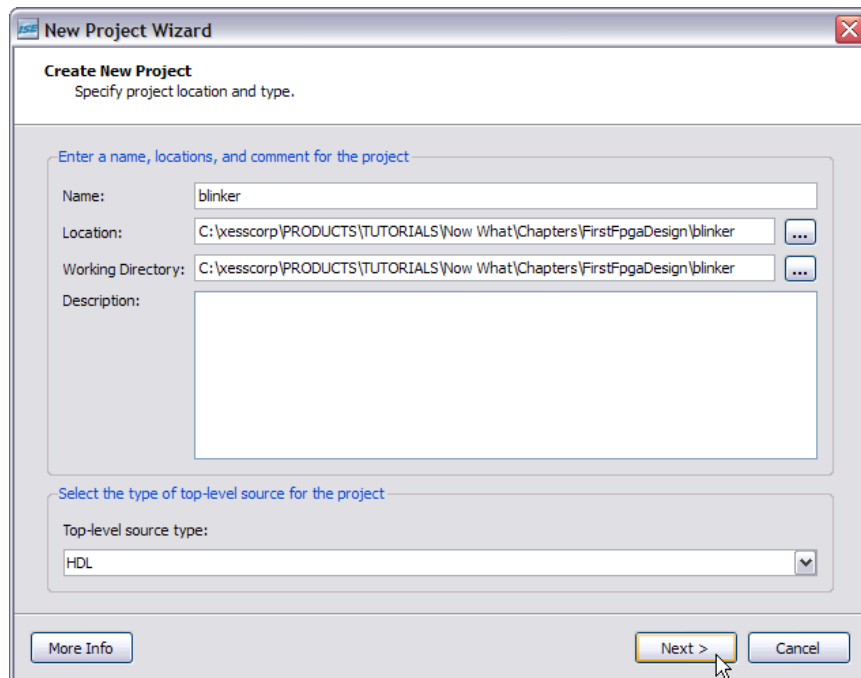
This will bring up the **New Project Wizard**. The first thing to do is select a location for your project. (The screenshot below shows my choice; yours, of course, will be different.)

Then type in a name for the project. I've chosen "blinker". (Notice how the project name is added to the end of the location you set previously. A folder with this name will be created there.) Then click on the **Next** button.

The next screen that appears is used to select the type of FPGA you want to use for your design (i.e., the *target*). Set the **Family**, **Device**, **Package** and **Speed** fields as shown below. (This sets up your design for the particular FPGA used on the XuLA board.) Also, change any other fields that don't match the screenshot. Then click the **Next** button

Now you'll get a summary of the information you've entered about your blinker design. Click on the **Finish** button and your new project will appear in the Navigator window.
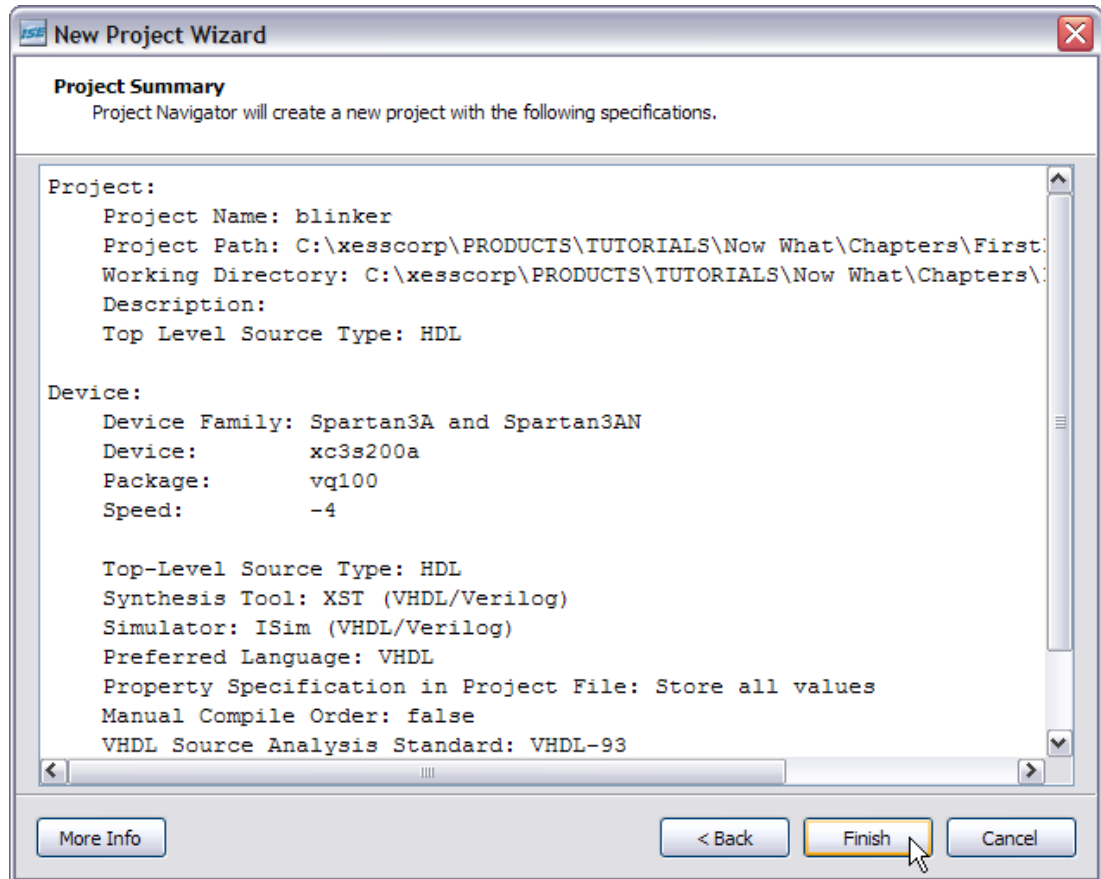
The Actual Blinker Design (in VHDL!)
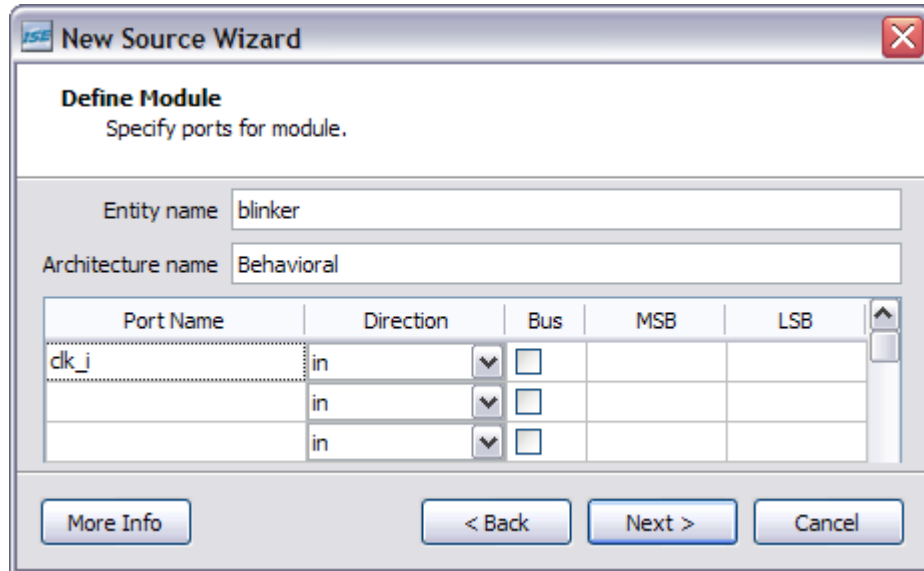
At this point, your project exists, but it doesn't do anything. It's time to enter the *source* information that describes how the blinker operates. In the Navigator window, right-click on the FPGA identifier (xc3s200a-4vq100 in this design) in the **Hierarchy** pane and select **New Source...** from the drop-down menu.
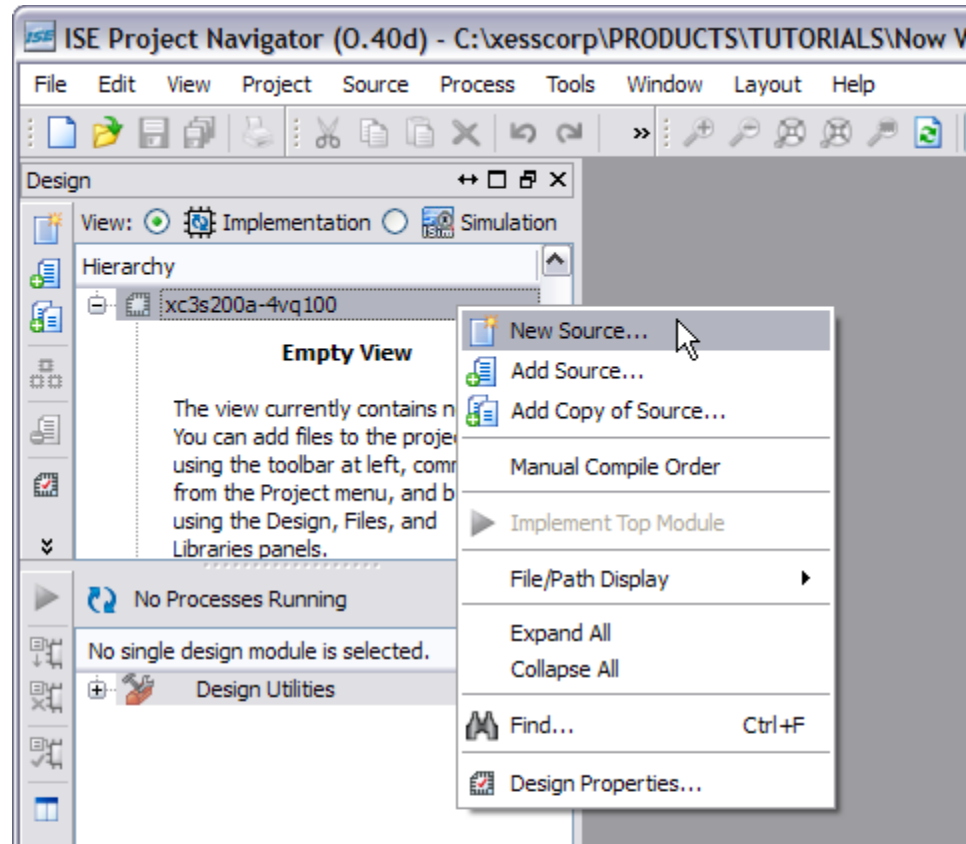
The **New Source Wizard** window appears. Here, you can select how you want to describe your design. For example, you could create the blinker from logic gates using schematics. Or maybe there is already a pre-built blinker circuit provided as an *intellectual property* (IP) core (don't bet on it!). But for this design, I will choose to use VHDL. VHDL (or Verilog, they're really quite similar) lets you describe the operations of logic designs using a high-level language. So, select **VHDL Module** from the list of source types. Then type a name for the VHDL source file. There will only be one for this simple design, so you can just name it "blinker". Then click on the **Next** button

FPGAs!? Now What?
TUT001 (V1.0) Feb 28, 2013

www.xess.com

30
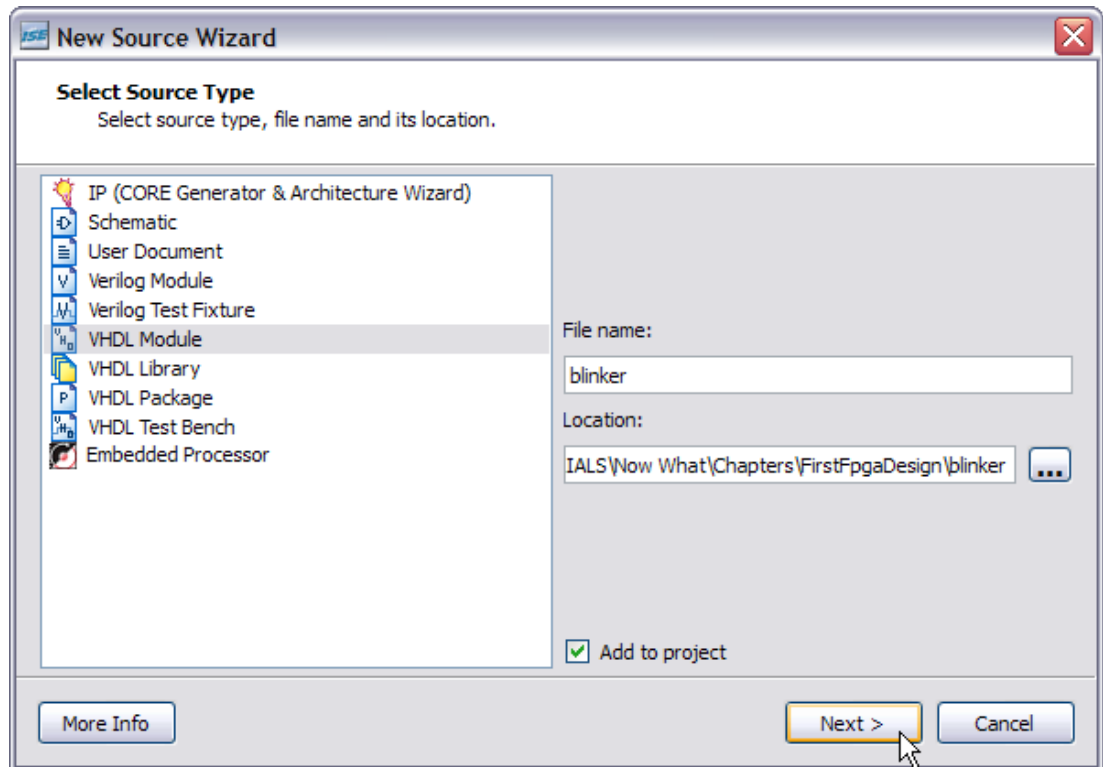
The next window that appears lets you define the input and output ports for the blinker design. Type "clk_i" into the first **Port Name** field. This will be the input for the 12 MHz clock signal. (The "_i" suffix on the port name is not necessary. I just use this notation to indicate which ports are inputs. That can help if I'm trying to figure out how an old design works if I haven't seen it for a few years.)
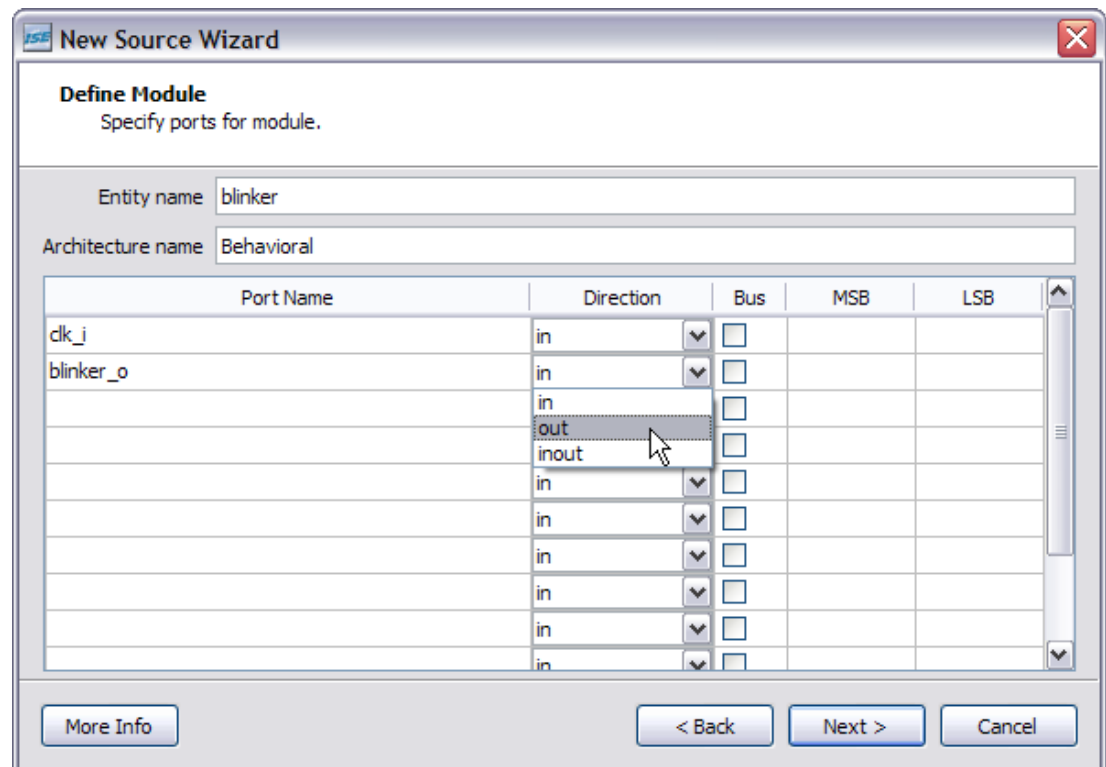
Then you can add the output that drives the LED. Name this "blinker_o" and set it to be an output using the **Direction** drop-down list as shown below.

That's all the inputs and outputs you need. Click on the **Next** button and you'll get a summary of your VHDL module. No surprises there! Click on the **Finish** button to complete the module.
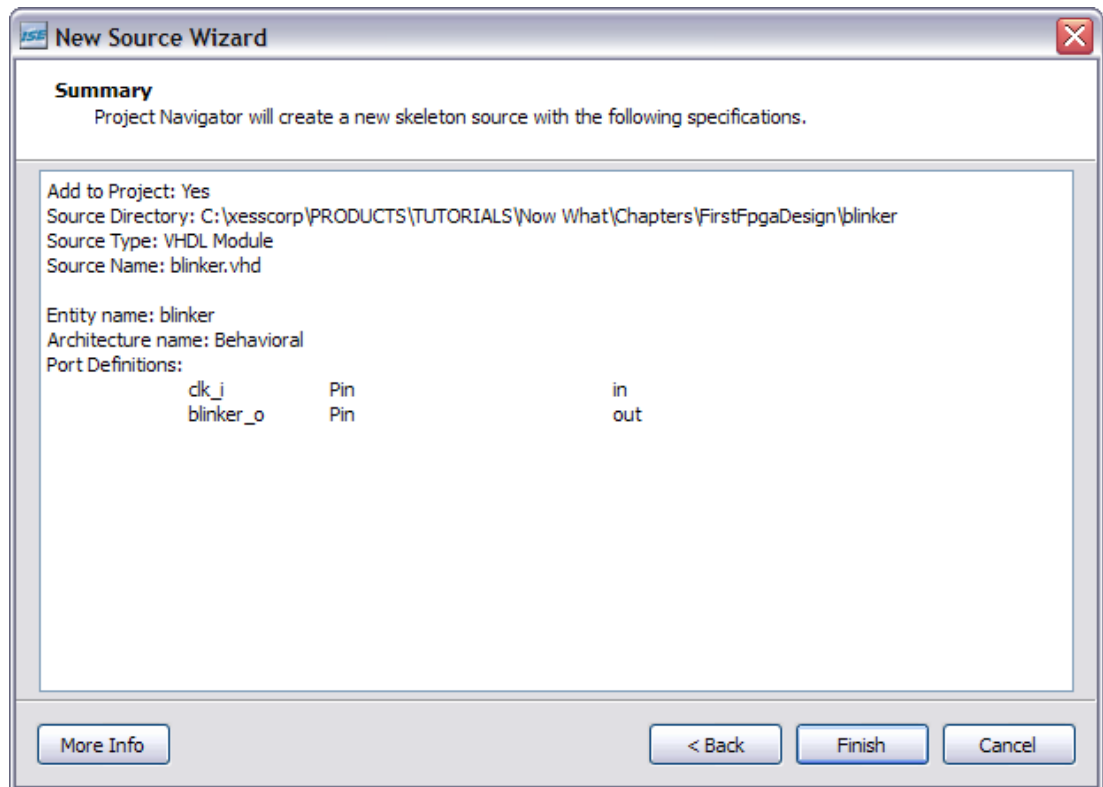
Now you should see the contents of your blinker VHDL module in the Navigator workspace pane to the right. (If you don't see it, click on the **blinker.vhd** tab along the bottom of the workspace.)
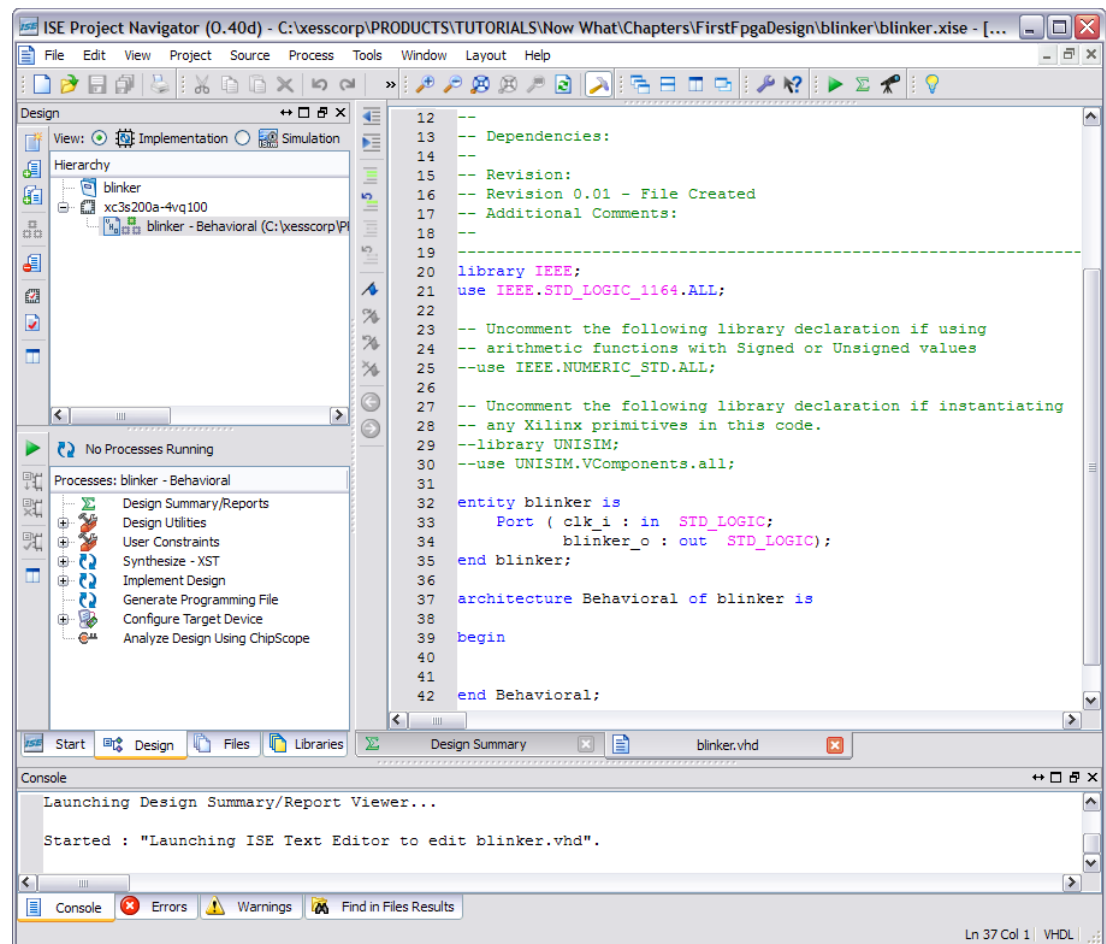
Most of the blinker.vhd file is comments, but there are a few spots you should look at:

Lines 20-21: These two lines open the STD_LOGIC_1164 *package* of the IEEE library. This library includes a bunch of useful functions and definitions and you'll see it included in most VHDL designs.

Lines 32-35: This is the *entity* section of the blinker where all the inputs and outputs are declared. This is like the skin of the module which defines how everyone on the outside will see the blinker.

Lines 37-42: This is the architecture section of the blinker where all the functions are defined. This portion is the guts of the design where the inputs are taken in, worked on, and then the result is passed out through the outputs. Notice, however, that the architecture section is conspicuously empty – the blinker will do nothing at this point!

So it's pretty clear that you need to put something into the architecture section to make the blinker work. A counter is a common component for reducing a 12 MHz clock down to a 1 Hz waveform. Why? Because a binary counter increments whenever the clock pulses and it goes through a binary sequence such as 000, 001, 010, 011, 100, 101, 110, 111, … If you look, you'll see that each bit of the counter pulses at half the frequency of the previous bit. (Look here for a better illustration of how binary counters work.) So if you build a counter with a enough bits, you can divide the 12 MHz clock downto 1 Hz.

But how many counter bits do you need? Well, a one-bit counter would divide the clock by

two down to 6MHz. A two-bit counter would divide it by four downto 3 MHz. An N-bit counter would divide by 2^N. So what value of N would divide by 12,000,000? It turns out you can't get exactly 12,000,000, but you can get 8,388,608 if you use $N=23$, or you can get 16,777,216 with $N=24$. For no particularly good reason, let's use $N=23$.

Here's a more detailed view of the blinker design given what we just covered:
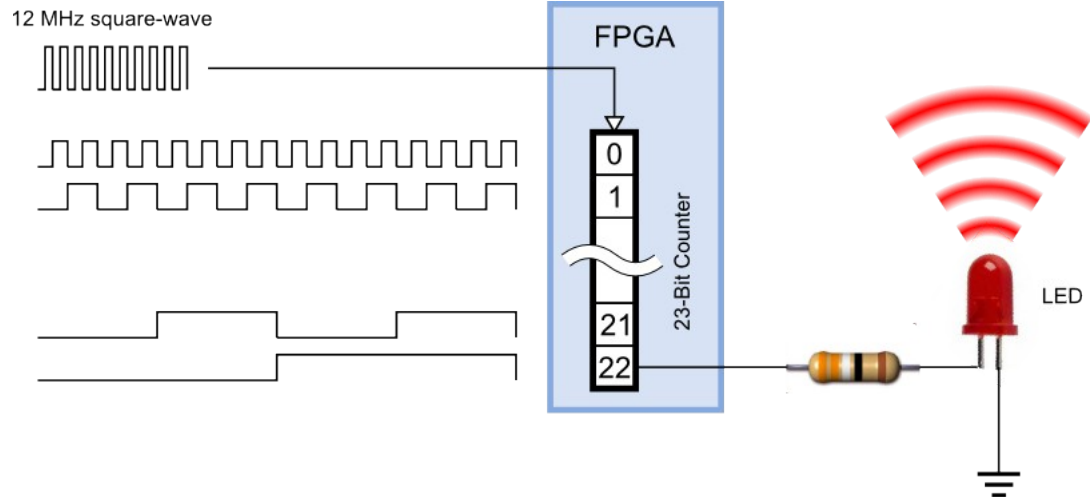
The 23-bit counter is incremented on the rising edge of the 12 MHz input clock. Bit 0 of the counter outputs a 6 MHz signal. (Why do we count the counter's bits starting from zero? BECAUSE THAT'S HOW ENGINEERS DO THINGS!!) Bit 1 outputs a 3 MHz signal. By the time we reach the final bit of the counter (bit 22), the signal is pulsing at a rate of 12,000,000 / 8,388,608 = 1.43 Hz. This bit is output from the FPGA and is connected to the LED. So the LED flashes once every 0.7 seconds.

Now all you have to do is describe this 23-bit counter in VHDL. That's not too hard. Here it is:

```vhdl
36
37   architecture Behavioral of blinker is
38   signal cnt_r : std_logic_vector(22 downto 0);
39   begin
40
41   process(clk_i) is
42   begin
43     if rising_edge(clk_i) then
44       cnt_r <= cnt_r + 1;
45     end if;
46   end process;
47
48   blinker_o <= cnt_r(22);
49
50   end Behavioral;
51
52
```

Design Summary (Synthesized)   blinker.vhd

Here's what's going on in the architecture section:

Line 38: This is the declaration for the register (`cnt_r`) that holds the 23-bit count

value. The value is represented as a vector of 23 binary digits with indexes from 22 down to 0. (I used the "_r" suffix to remind myself that this is a register.)

Lines 41-46: This is a process that is triggered every time the clock input changes its value, either on the rising edge (i.e., from 0 ⇨1) or the falling edge (1 ⇨0). Within the process is an `if` statement that increments the value in the count register only on the rising edges of the clock.

Line 48: The most-significant bit of the count register is attached to the blinker output. That's all there is to it. Now let's see if it passes through the synthesizer.

Synthesizing the Blinker (or Not)

To run the synthesizer, left-click on the **blinker** icon in the Hierarchy pane. Once you do that, you should see a list possible operations appear in the **Process** pane. Double-click the **Synthesize – XST** process icon to start the VHDL synthesizer.
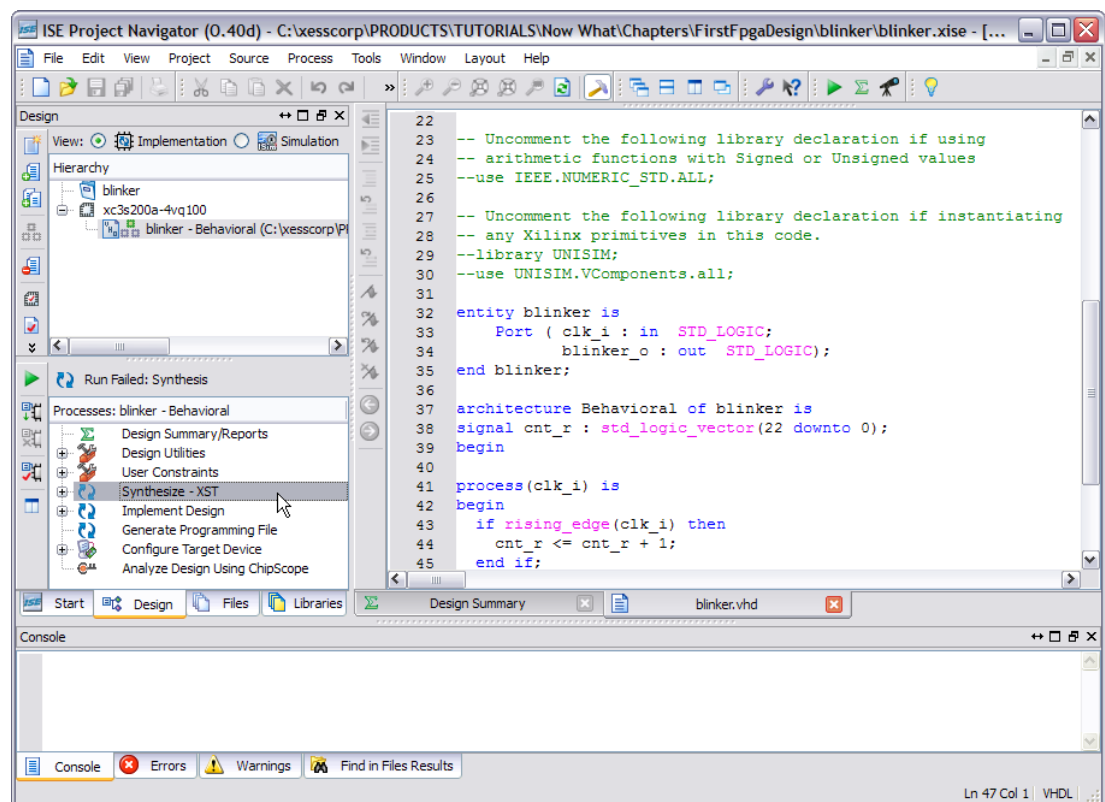
Within a few seconds you should see the results of the synthesis, and it isn't good. In the bottom portion of Navigator, you'll see this:
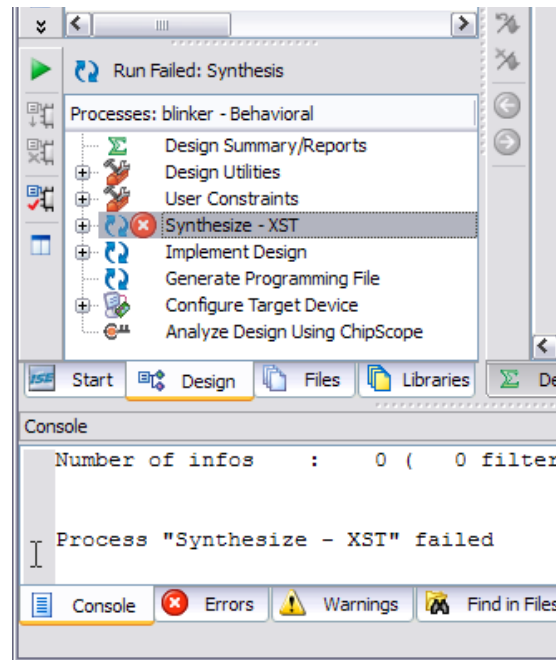
That red "X" is never a good sign. In this particular instance, it means the synthesizer has run into an error. As to what the error is, click on the **Errors** tab of the Transcript window to get the details.
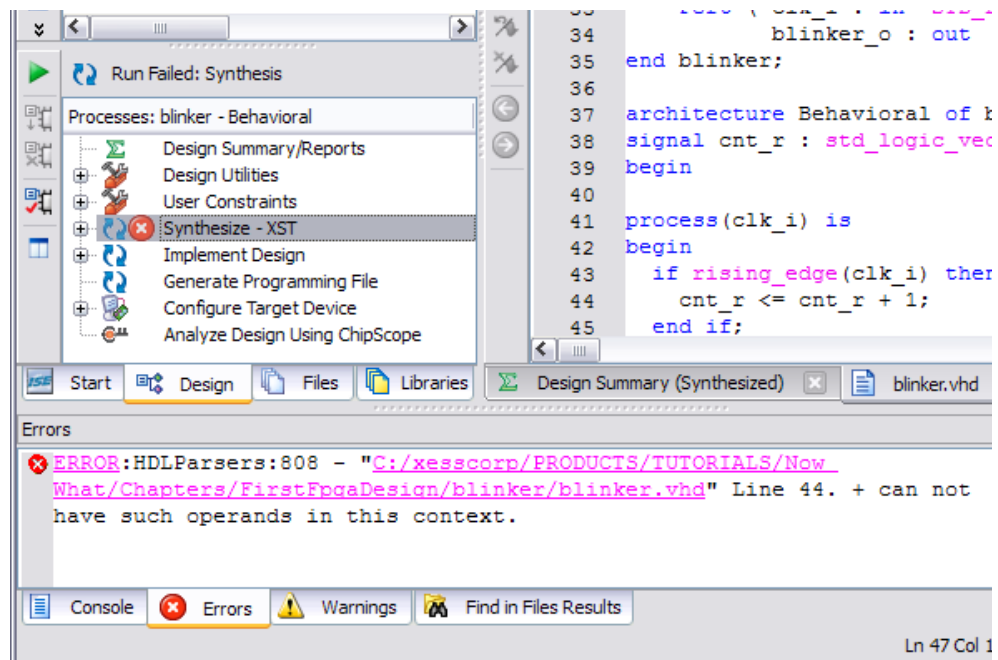
The synthesizer is telling you that the "+" operator cannot process its operands. In other words, there is no operation defined for adding an integer to a `std_logic_vector`. That's not a big problem; you just have to include the package that defines this particular addition operator. It turns out it's defined in the STD_LOGIC_UNSIGNED package, so just add what's shown on line 22 to the VHDL like so:

```
16   -- Revision 0.01 - File Created
17   -- Additional Comments:
18   --
19   ----------------------------------------------
20   library IEEE;
21   use IEEE.STD_LOGIC_1164.ALL;
22   use IEEE.STD_LOGIC_UNSIGNED.ALL;
23
24   -- Uncomment the following library declaration
25   -- arithmetic functions with Signed or Unsigned
26   --use IEEE.NUMERIC_STD.ALL;
27
28   -- Uncomment the following library declaration
29   -- any Xilinx primitives in this code.
30   --library UNISIM;
31   --use UNISIM.VComponents.all;
32
33   entity blinker is
34       Port ( clk_i : in  STD_LOGIC;
35              blinker_o : out  STD_LOGIC);
36   end blinker;
37
38   architecture Behavioral of blinker is
39   signal cnt_r : std_logic_vector(22 downto 0);
```

Design Summary (out of date)   blinker.vhd

Save the file and double-click the **Synthesize – XST** process icon again. Now you should see the synthesis completed successfully.

```
No Processes Running

Processes: blinker - Behavioral
    Σ     Design Summary/Reports
    +     Design Utilities
    +     User Constraints
    +  ✓  Synthesize - XST
    +     Implement Design
          Generate Programming File
    +     Configure Target Device
          Analyze Design Using ChipScope

  Start   Design   Files   Libraries   Design

Console
```
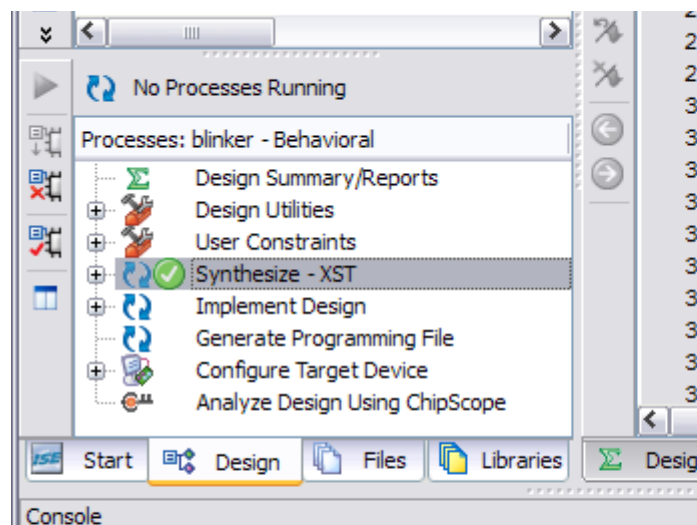
Now you have a netlist synthesized from your VHDL file. But does it work?

Testing the Logic

Before you charge off and run the implementation tools so you can blast your design into an FPGA to see if it works, it might be wise to take a little time to do a logic simulation. (Now I admit I always run the implementation tools and blast my designs into an FPGA to see if it works before even considering running a simulation, but you shouldn't emulate my bad habits.)

In order to simulate your design, you need something to exercise its inputs and watch its outputs. For your blinker design, you need an external wrapper that will apply a signal to the clock input and watch the output to see what it does. This wrapper is called a *test bench*.
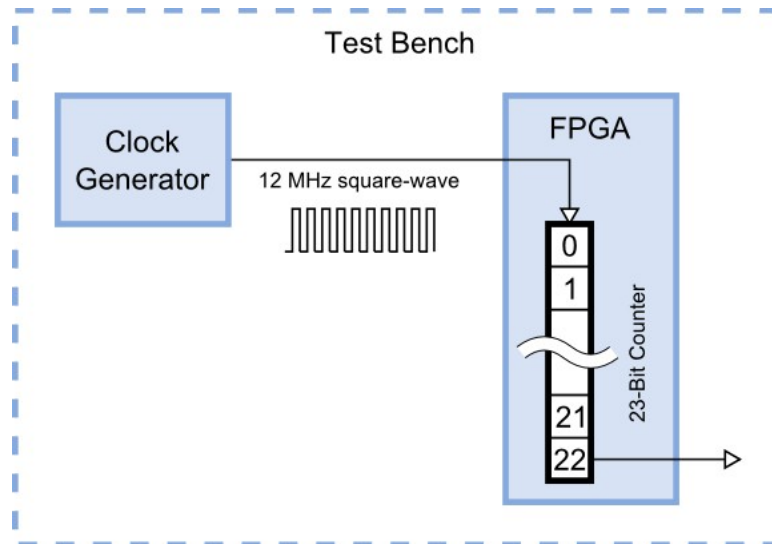
So you need to add a test bench to your project. To do this, right-click on the FPGA identifier in the **Hierarchy** pane and select **New Source…** in the pop-up menu. In the New Source Wizard window, select the VHDL Test Bench as the source type and name it "blinkerTestBench". Then click on the **Next** button.
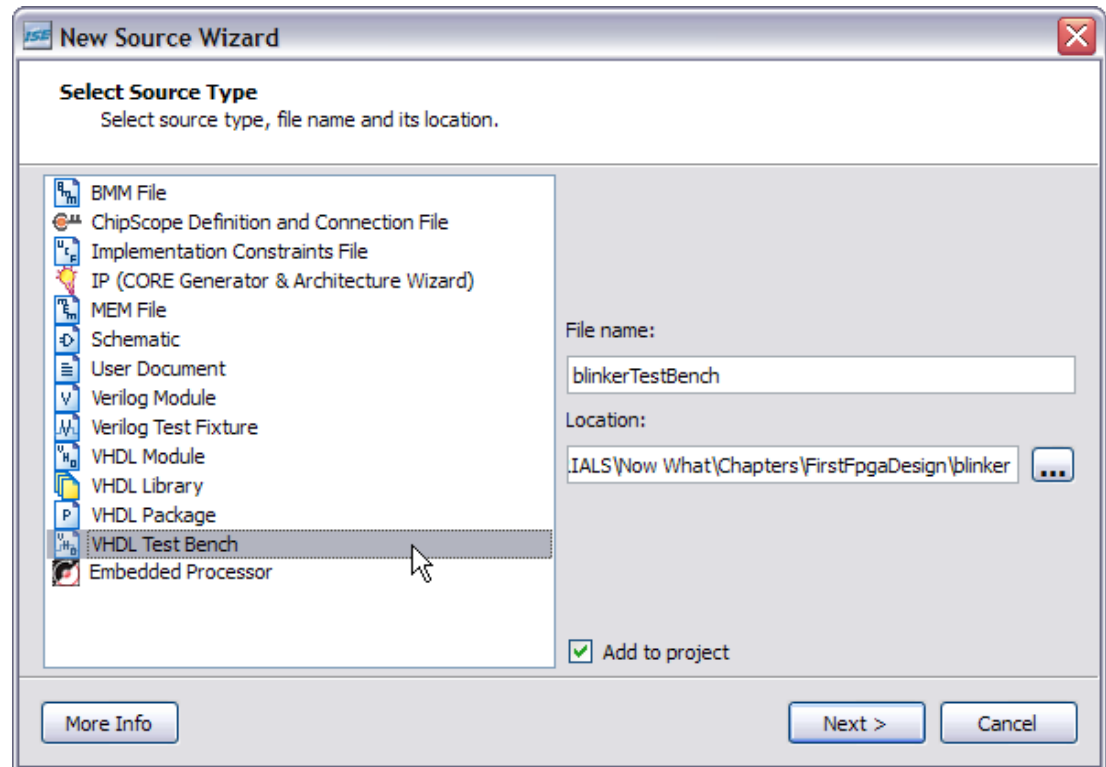
The next window that appears asks you to select a source file to associate with your test bench. Essentially, this is asking for the module to wrap the test bench around. There's only one module in your design (blinker), so that's your only choice. Click **Next** and move on.

You'll get a summary screen about the choices you've made. Again, no surprises! Click on the **Finish** button to complete the addition of the test bench module.
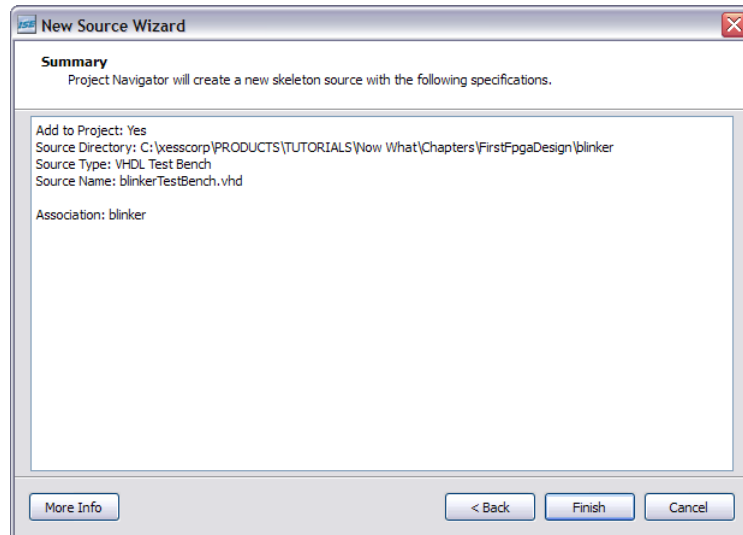
When you return to the Navigator screen, take a look at your **Hierarchy** pane. Where's the test bench you just added? It's not actually missing, it just doesn't show up in the **Implementation** view of your project. In order to see the test bench, you have to select **Simulation** in the **View** pane like so:
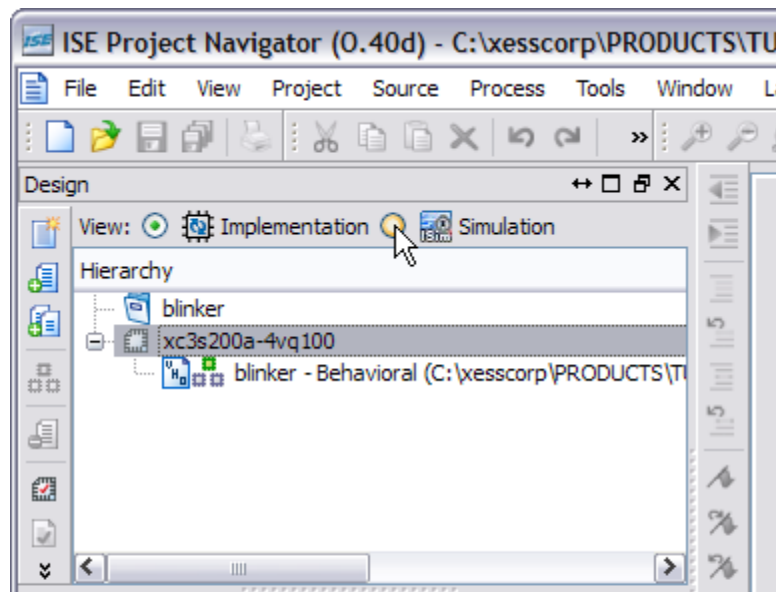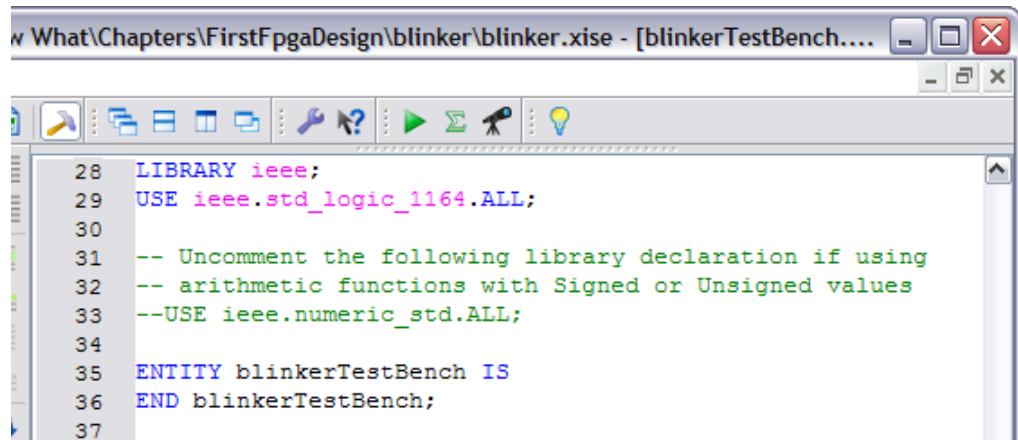
Once you select the Simulation view, the blinkerTestBench file shows up in the **Hierarchy** pane. The important point to note here is that the test bench module is not something that will ever be implemented and downloaded to the FPGA. In fact, it might be difficult or impossible to do so because the test bench might contain language constructs that are difficult or impossible to build in the FPGA (like accessing a file on the host computer to store test results). The test bench serves only to help in exercising the actual modules that will end up in the FPGA. Speaking of which, where is the blinker module?
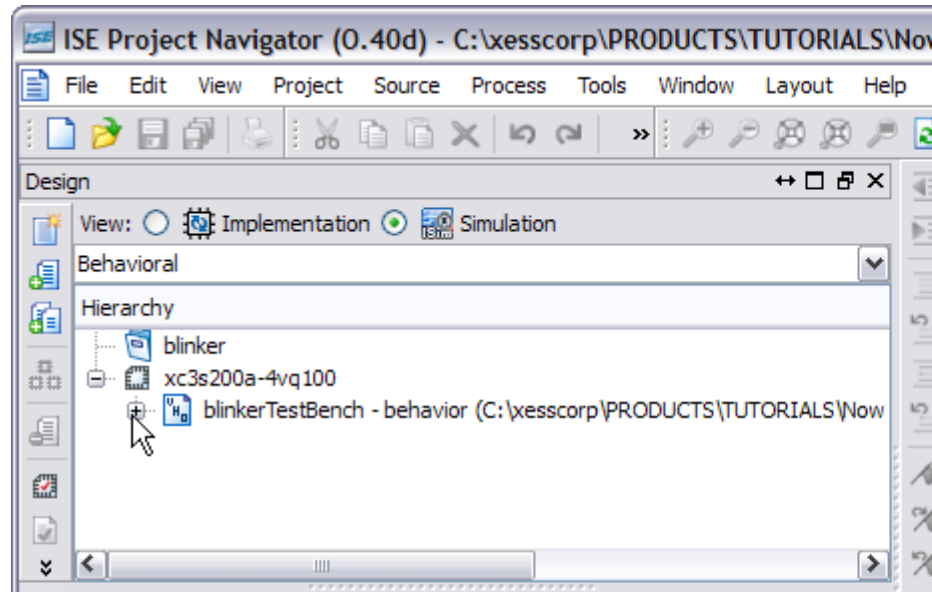
By clicking on the "+" sign next to the blinkerTestBench icon, you can see the blinker module is displayed as a submodule contained in the blinkerTestBench module. The blinker module is labeled "uut" in the test bench. This stands for "unit under test" to indicate the blinker module is what is being tested.
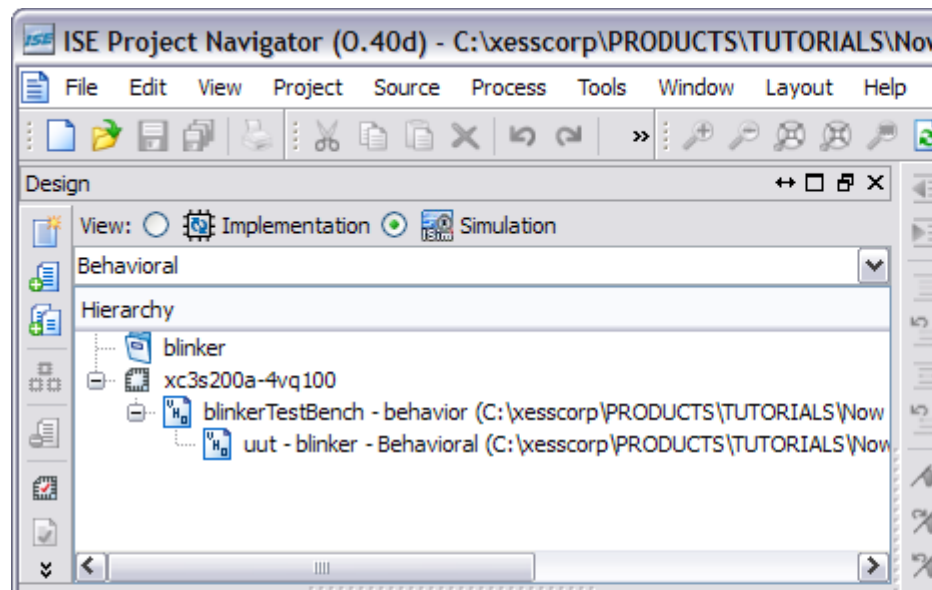
Now you'll want to double-click the blinkerTestBench file so you can add the VHDL code for exercising the blinker module. (It turns out you won't have to do much.) In the initial portion of the test bench file, you'll see the inclusion of the IEEE library (lines 28-29) and the entity declaration for the blinkerTestBench module. Notice that the entity section lists no inputs or outputs. That's because the test bench completely wraps the UUT and no inputs or outputs pierce the wrapper – everything happens inside the test bench.

The architecture section (see the following figure) instantiates the blinker module as the UUT, connects to its inputs and outputs, and then drives the inputs. Here are the details:

Lines 42-47: This is a component declaration so the rest of the code knows what the interface to the blinker module looks like. Hence, it looks very similar to the blinker module entity section.

Lines 50-54: Signals internal to the test bench are declared that will be connected to the input and output ports of the blinker module.

Line 57: The period of the clock input is defined with the default value of 10 ns. Since you want to simulate the blinker with an input clock of 12 MHz, this constant should be changed to 83.3 ns.

Lines 62-65: The blinker module is instantiated here. Its input and output ports are connected to the similarly-named signals declared previously.

Lines 68-74: The clock signal is generated in by this process. The clock signal is forced to 0 and the process waits for half of the clock period. Then the clock is raised to 1 and the process waits for the remaining half of the clock period. Then the entire sequence repeats forever.

Lines 78-88: Test vectors for other inputs to the UUT can be generated in this process. Since the blinker module only has a clock input, there's nothing for this process to do. So leave it unchanged.

```
38    ARCHITECTURE behavior OF blinkerTestBench IS
39
40        -- Component Declaration for the Unit Under Test (UUT)
41
42        COMPONENT blinker
43        PORT(
44             clk_i : IN   std_logic;
45             blinker_o : OUT  std_logic
46           );
47        END COMPONENT;
48
49
50      --Inputs
51      signal clk_i : std_logic := '0';
52
53      --Outputs
54      signal blinker_o : std_logic;
55
56      -- Clock period definitions
57      constant clk_i_period : time := 10 ns;
58
59   BEGIN
60
61      -- Instantiate the Unit Under Test (UUT)
62      uut: blinker PORT MAP (
63             clk_i => clk_i,
64             blinker_o => blinker_o
65           );
66
67      -- Clock process definitions
68      clk_i_process :process
69      begin
70         clk_i <= '0';
71         wait for clk_i_period/2;
72         clk_i <= '1';
73         wait for clk_i_period/2;
74      end process;
75
76
77      -- Stimulus process
78      stim_proc: process
79      begin
80         -- hold reset state for 100 ns.
81         wait for 100 ns;
82
83         wait for clk_i_period*10;
84
85         -- insert stimulus here
86
87         wait;
88      end process;
89
90   END;
```

Design Summary (Synthesized)   blinker.vhd   blinkerTestBench.vhd

The test bench looks like it's ready, so it's time to run a simulation. Click on the blinkerTestBench entry in the Hierarchy pane, and then double-click the Simulate Behavioral Model in the Processes pane. (There's no need to run the Behavioral Check Syntax since only the clock period was changed.)
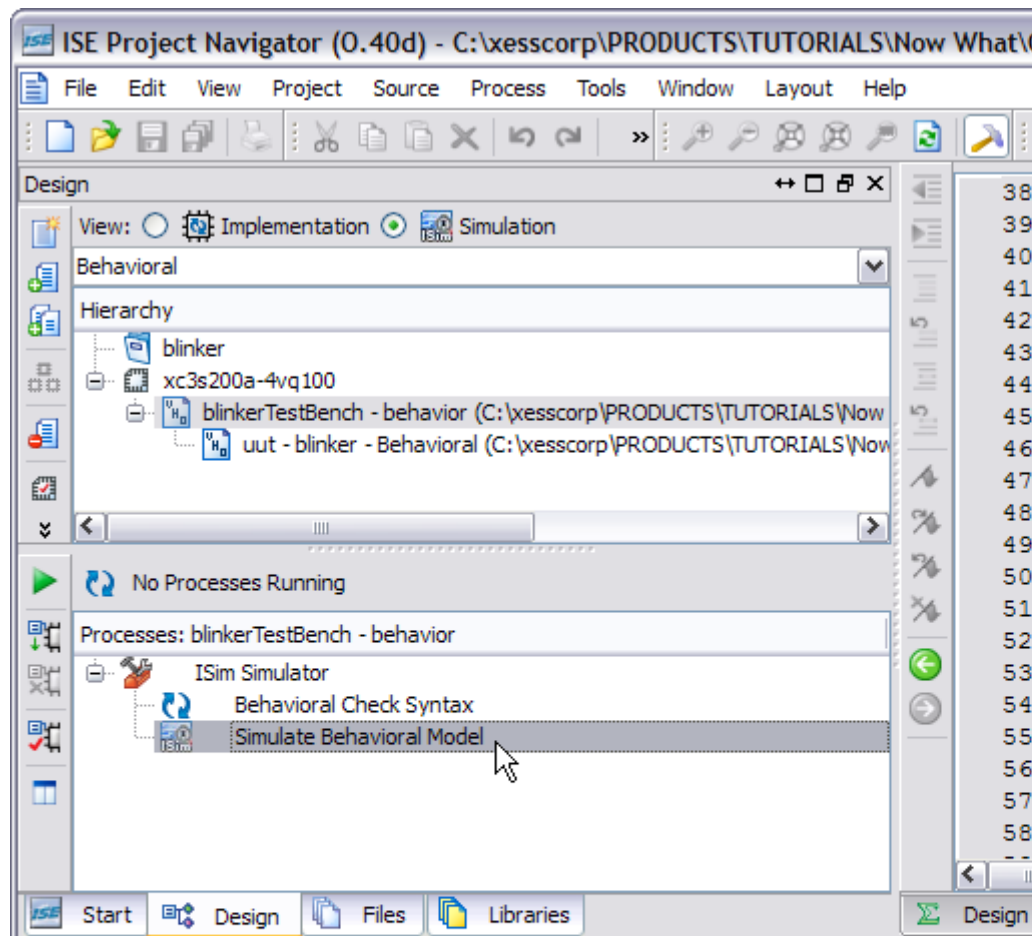
The ISim window will appear and display the results of a 1 μs simulation run. (Click on the **View ⇨ Zoom ⇨ To Full View** menu item to make the entire waveform visible.)
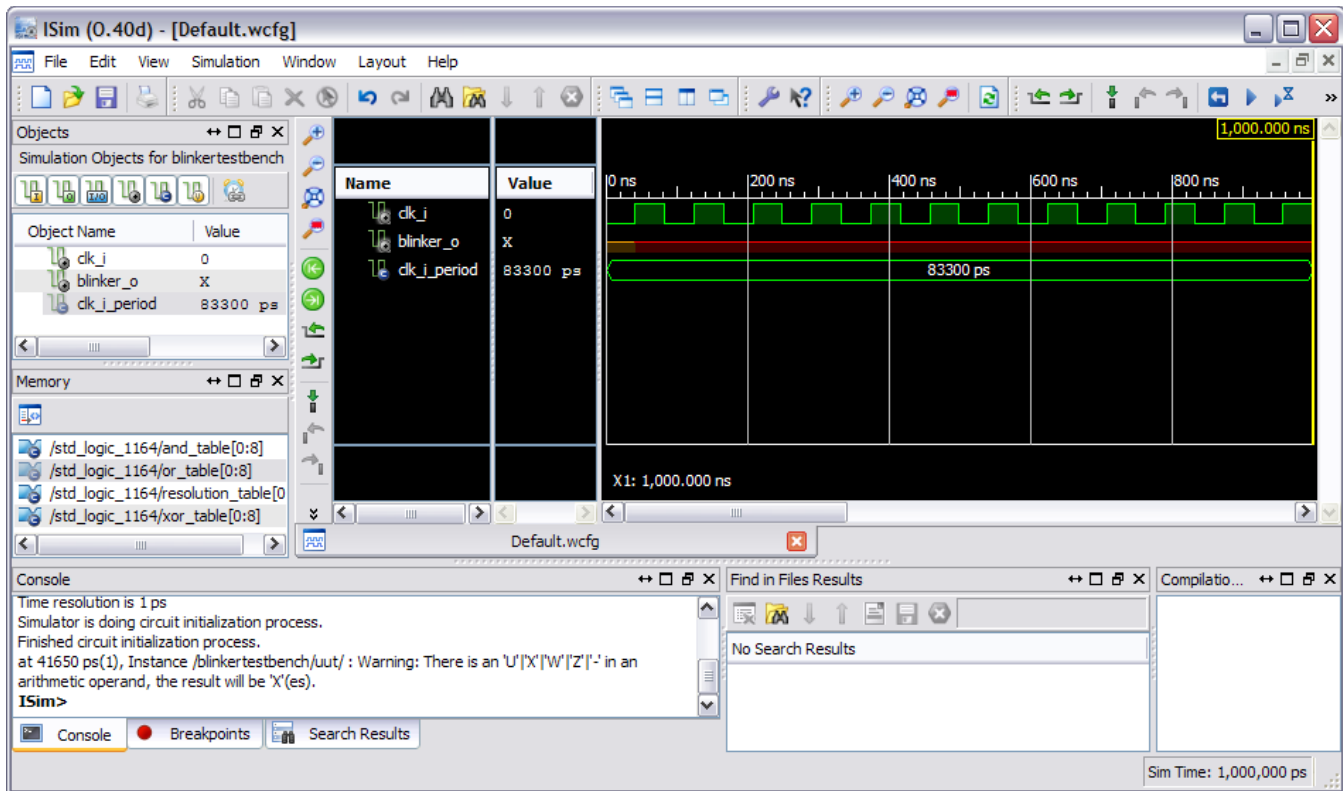
Although the simulation only ran for a microsecond, there's already something wrong about it. The blinker_o output remains in an unknown state (signified by an "X") even after the clock signal has pulsed twelve times. It should have gone to one or zero after the first clock pulse.

This is a symptom of an initialization problem. In this case, the counter register is not initialized to a known value at the start of the simulation. Since the counter value is unknown, the simulator can't determine the next value it should contain when a rising clock edge occurs. So all the counter bits stay in an unknown state, including the counter bit attached to the blinker_o output.

How do you fix this? Well, the *right* way would be to build some reset circuitry in the blinker module that sets the counter register to zero when a reset input is triggered. Then the test bench could be modified to trigger the blinker module reset at the start of the simulation. But that's a lot of work. A simpler solution, but not recommended in general, is to initialize the counter register value when it is declared. To do this, double-click the blinker module in the Hierarchy pane of the Navigator window. This will open the blinker module. Then change the cnt_r declaration on line 39 to:

```
signal cnt_r : std_logic_vector(22 downto 0) := (others=>'0');
```

This will initialize all the counter register bits to zero. Save the blinker.vhd file. Then re-run the simulation by clicking on the **Simulation ⇨ Relaunch** menu item. Now the blinker_o output takes on a value of zero at the start of the simulation.
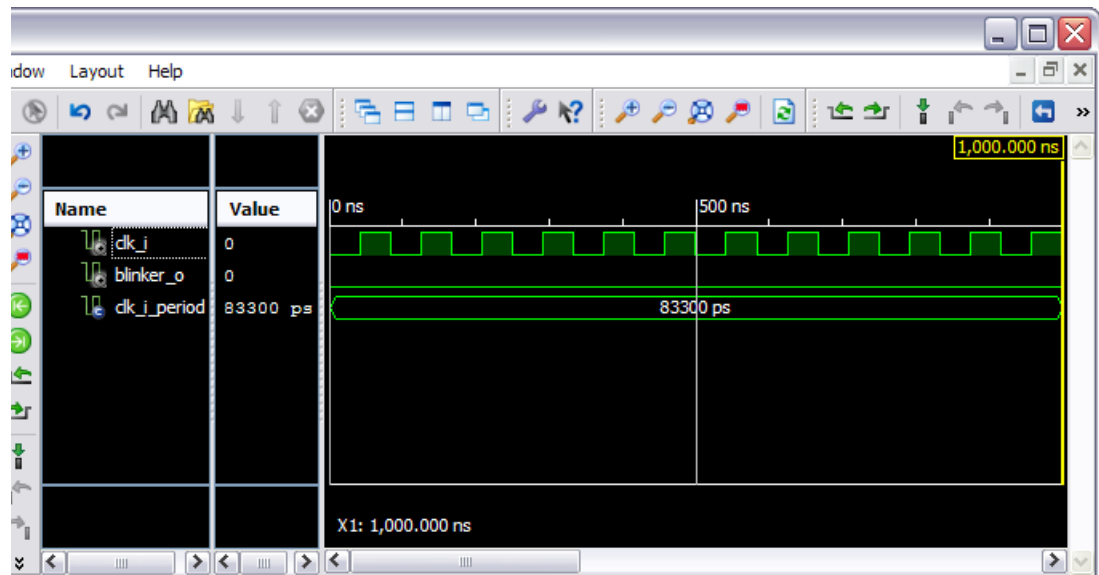
But you still don't know if the blinker output is going to do anything (except sit at zero) because the simulation duration is too small. The blinker output should pulse every 0.7 s, so a simulation time of 2.8 s will be enough to see four pulses. Enter "2.8s" into the simulation run-time field on the ISim toolbar and then click on the Run button.
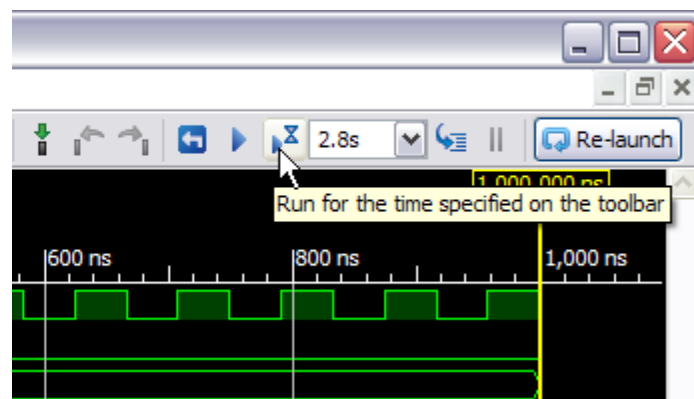
After a minute or two (because computing 2.8 s of simulation time takes a lot of work), the results of the simulation will appear. Use the **View ⇨ Zoom ⇨ To Full View** menu item to display the entire simulation. You can see that the blinker_o output pulses four times during the 2.8 s simulation, giving a period of 0.7 s. So the blinker seems to be working as expected.
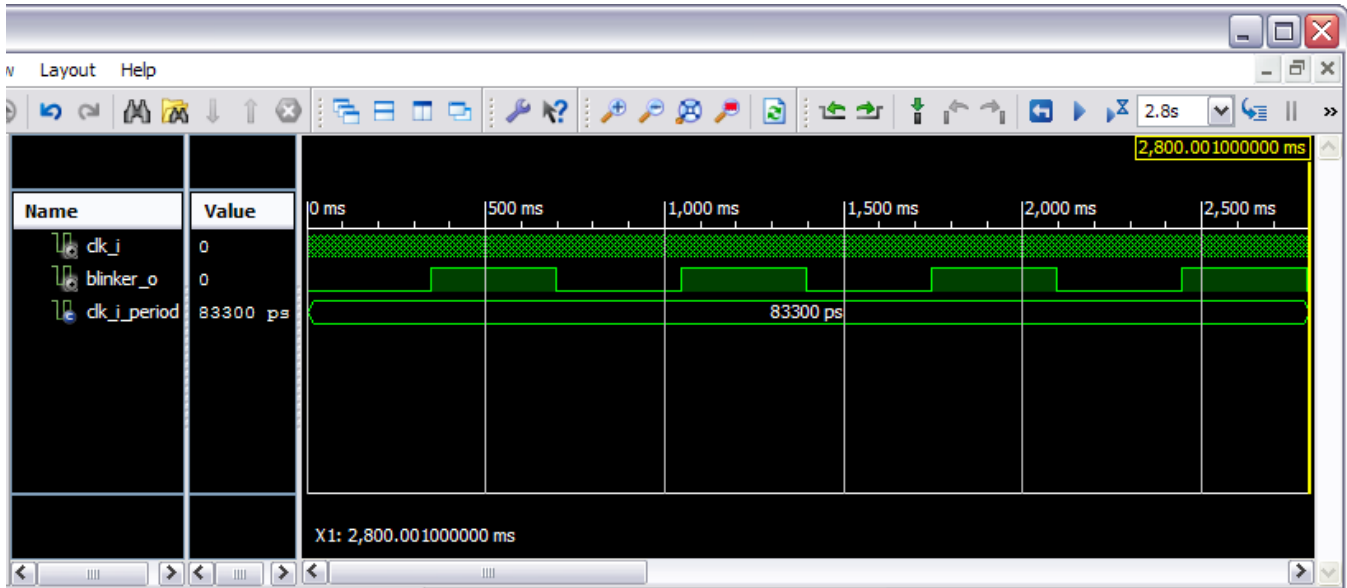
This chapter has been all software. Even the blinker, when it runs, is executing in software. That's not the reason you're interested in FPGAs, is it? No! - you want to build circuits that run in the real world. And that's what you'll do in the next chapter.

C.4 *"I have a netlist.*
Now what?"

Physical Hardware – the XuLA Board

OK, you've designed the blinker in VHDL. You've synthesized it. You've simulated it and it seems to work. But, up to now, it's all been *virtual*. Now it's time to get *physical*.

To get the blinker design into the real world, you'll need an FPGA chip. But the FPGA chip isn't much good by itself because its usually in a surface-mount package that can't be plugged directly into a breadboard. And it needs some type of interface to download the bitstream from the host (where WebPACK is running). And it would be nice to have some ancillary circuitry around the FPGA for providing clock signals, etc. For these reasons, you'll probably need an *FPGA development board*. For this book, I've chosen the XuLA board, but you can use other boards for the same purpose.

Here's a picture of the front and back of the XuLA board. The FPGA is the big chip on the front below the "XuLA" logo. (No, it is *not* the quarter.)
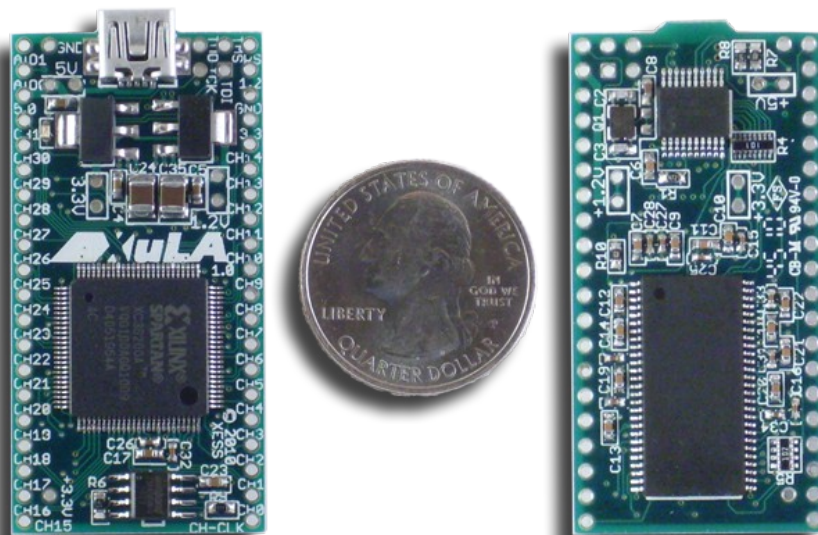
Here's a block diagram of what's on the XuLA board. You can safely ignore the most of the board's circuitry for now. I've highlighted the following parts that will be important for the blinker design:

FPGA: Of course the FPGA is important! I shouldn't need to explain why that is at this point. The FPGA is the Xilinx XC3S200A in a 100-pin QFP package (xc3s200a-4vq100), just like the one you selected when you started the blinker project in the last chapter.

Clock input: The XuLA board provides a 12 MHz square-wave clock to the FPGA. That will be the clock you'll use as the input to the blinker design.

Prototyping header: Some pins on the FPGA chip are connected to header pins around the outside of the XuLA board. These header pins are a convenient way to access signals from the FPGA when the XuLA board is inserted into a standard breadboard. You'll output the blinker_o signal on one of the FPGA pins that goes to this header and connect an external LED to it.

Finally, here's how the pins of the FPGA are assigned to the XuLA prototyping header. The FPGA pin numbers begin with a "p" so, for example, pin p50 of the FPGA is connected to the CHAN3 pin of the prototyping header.

Assigning I/O Signals to FPGA Pins

Now that you know a little bit about the hardware that will be running your blinker design, you can proceed to the implementation phase. Open the ISE Project Navigator and return to the blinker project. Then move your project from simulation mode to implementation mode by clicking on the **Implementation** button in the **Views** pane.
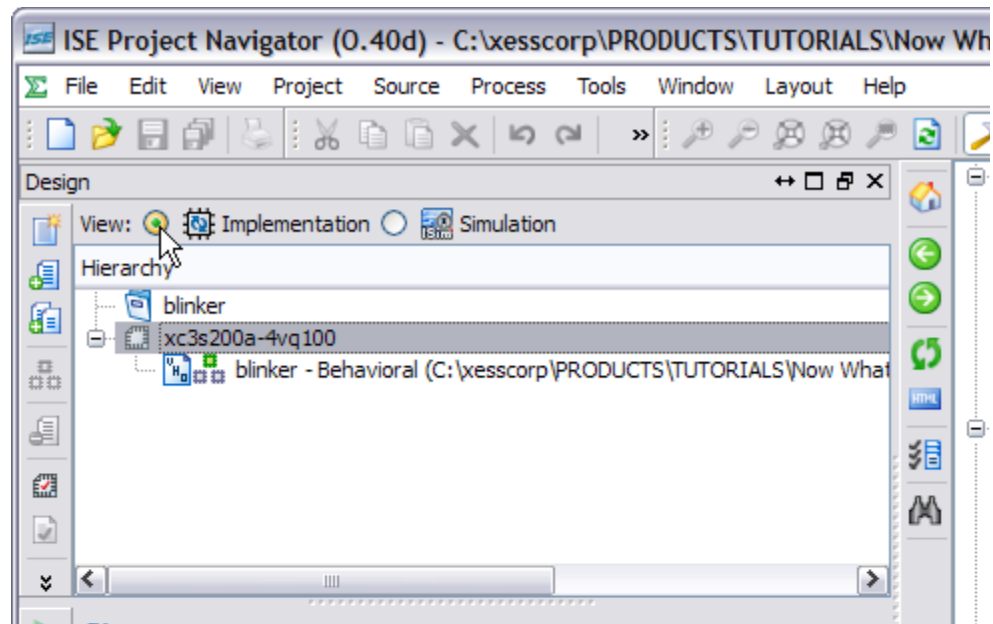
Before you can pass the synthesized netlist through the implementation tools, you'll need to tell them what pins of the FPGA will be used for the input and output ports of your blinker. Otherwise, the tools will just assign them to any pins it deems fit for the purpose. An *Implementation Constraints File* (sometimes called a *User Constraints File* or UCF) is used to assign I/O signals to the FPGA pins.

To add a constraints file to your project, right-click in the **Hierarchy** pane and select **New Source…** from the pop-up menu. In the New Source Wizard window, select **Implementation Constraints File** as the source type and give it the name "blinker". (You don't have to give it the same name as the project, but I usually do.) Then click on the **Next** button.
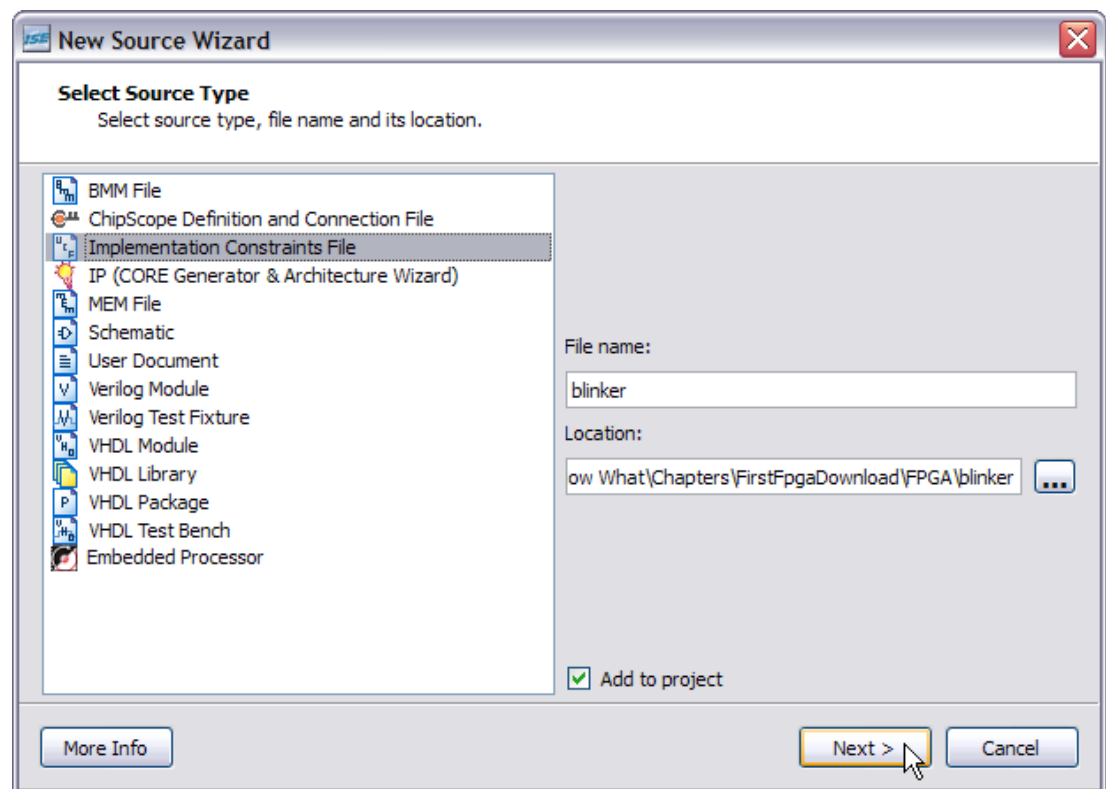
Another summary window will appear. Ho hum. Click on the **Finish** button and your constraints file will appear in the Hierarchy pane. Double-click on the constraints file and it will open in the workspace area to the right.
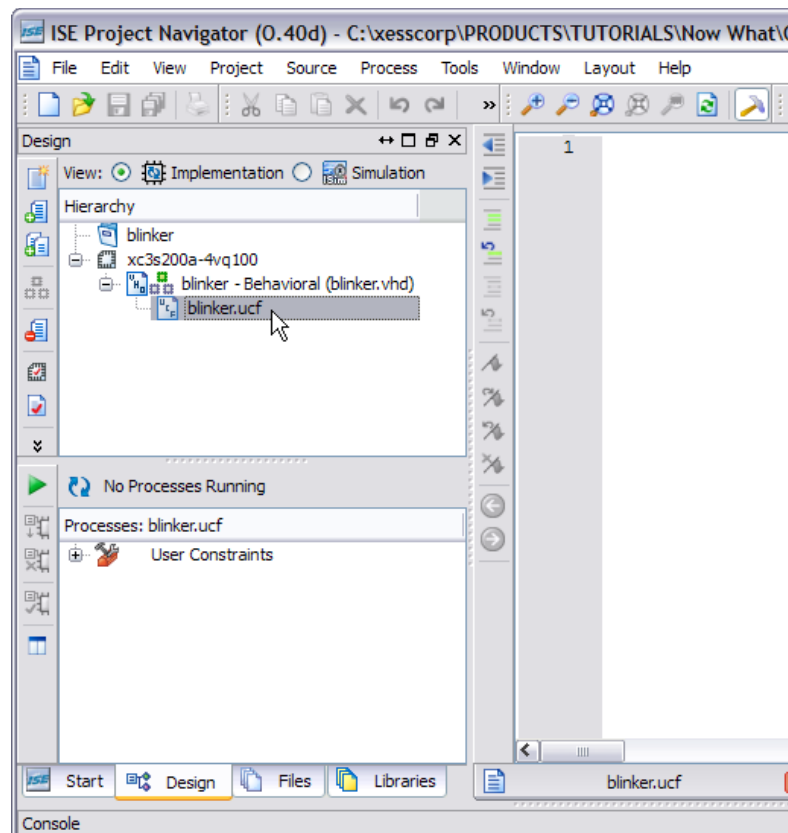
Type the following two lines into the blinker.ucf file and then save the file.


```
1   net clk_i     loc=p43;  # 12 MHz input clock.
2   net blinker_o loc=p44;  # Blinker output to LED.
```
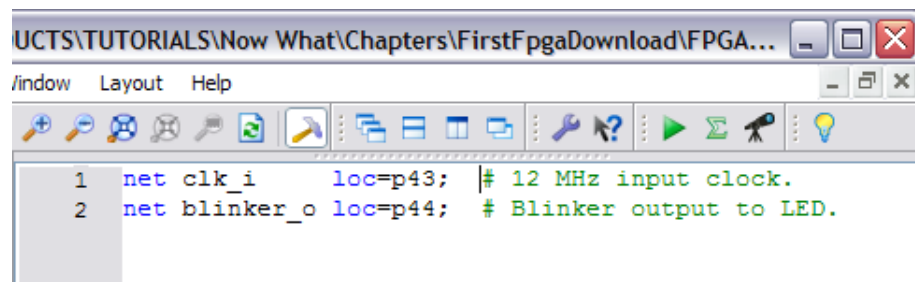
The first line assigns the `clk_i` net to pin location `p43` of the FPGA. This is the pin that is driven by the 12 MHz clock on the XuLA board. (I know because I read the XuLA manual.) The implementation tools will look through the blinker netlist and see that `clk_i` is an input net, and then they will configure the IOB for `p43` as an input buffer.

Similarly, the second line assigns output net blinker_o to pin location p44 whose IOB is then configured as an output driver. Why `p44`? Because that is one of the FPGA pins connected to the XuLA prototyping header. Pin `p44` exits the XuLA board on the lower-right corner pin labeled "CHAN-CLK".

That's all you need to do before running the implementation tools.

Doing the Implementation

After all this work, the actual implementation is going to be rather anticlimactic. To begin, highlight the blinker module in the **Hierarchy** pane. Then, double-click **Implement Design** in the **Processes** pane.
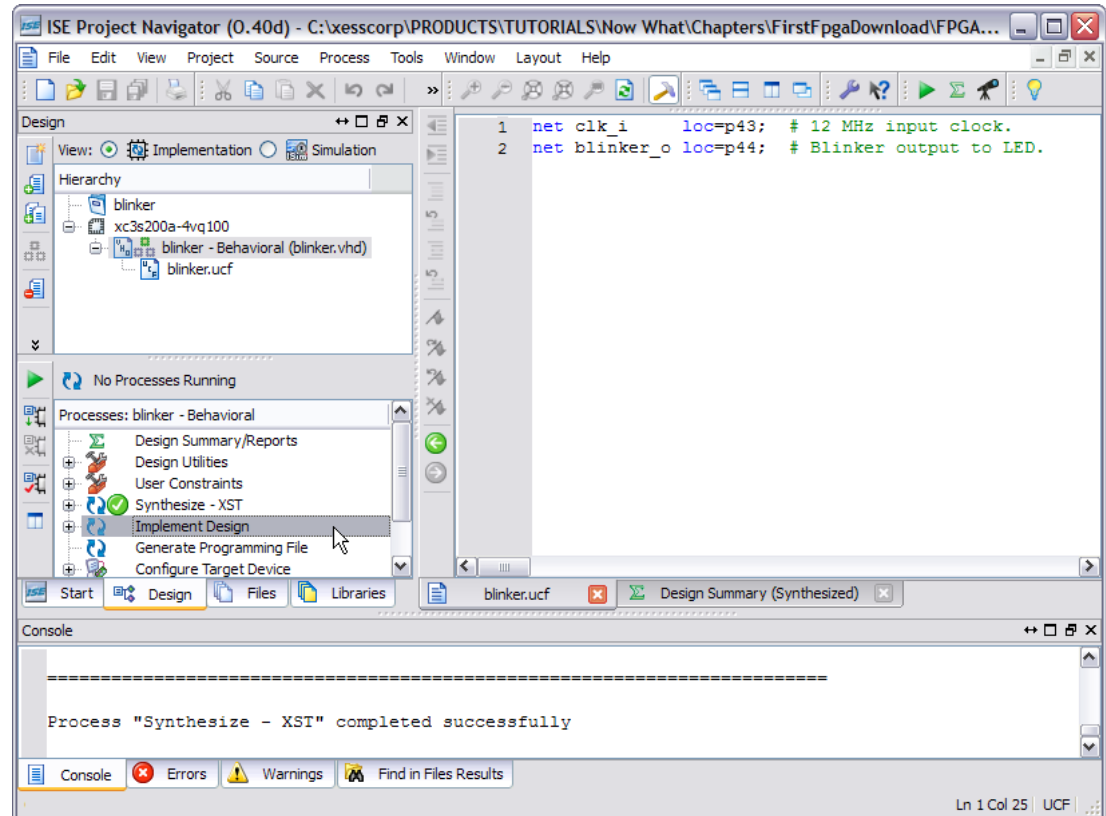
You'll see messages scrolling through the **Console** tab in the **Transcript** window. In less than a minute (depending upon the speed of your PC), the translation, mapping and place & route phases will be done and a green check-mark will appear. You have successfully implemented your blinker! Now you have to get it into a form that you can download into the FPGA.
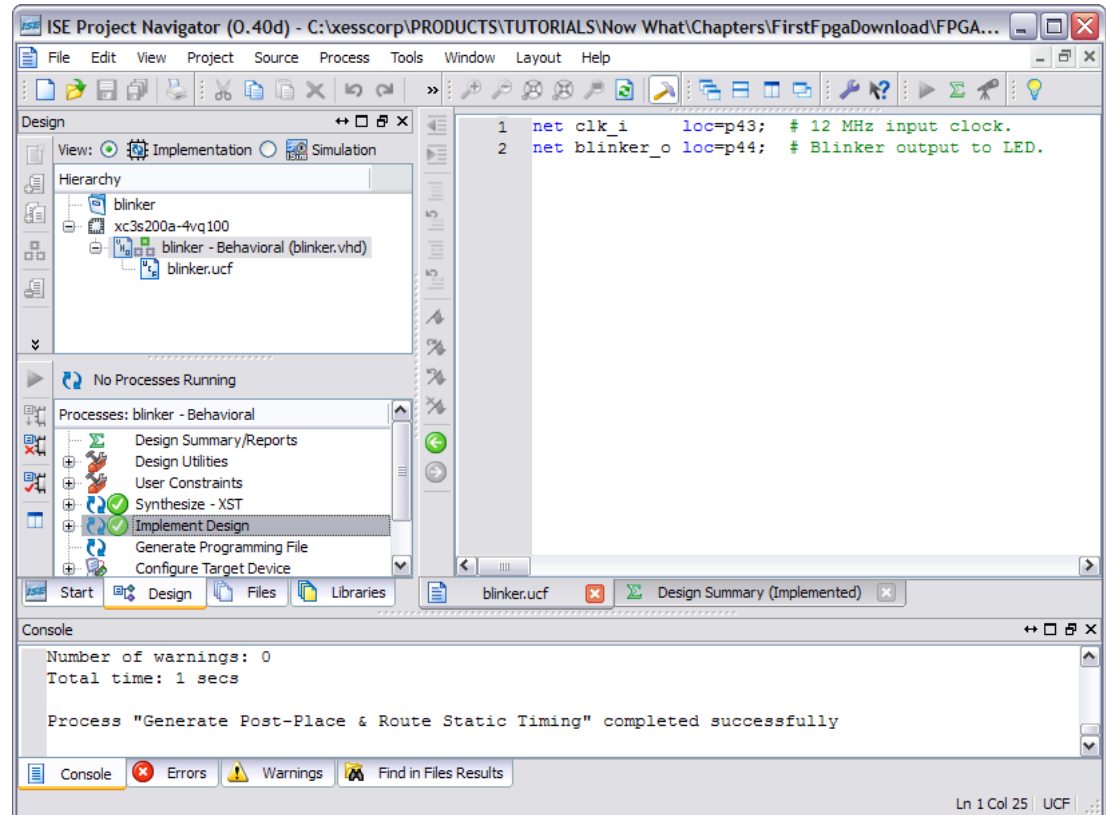
Preparing the Bitstream

To generate the bitstream, all you have to do is double-click **Generate Programming File** in the Processes pane. *But don't do that yet!*

First, you have to change some configuration settings to make the bitstream compatible with the XuLA board. To do this, right-click on **Generate Programming File** and select **Process Properties...** from the pop-up menu.
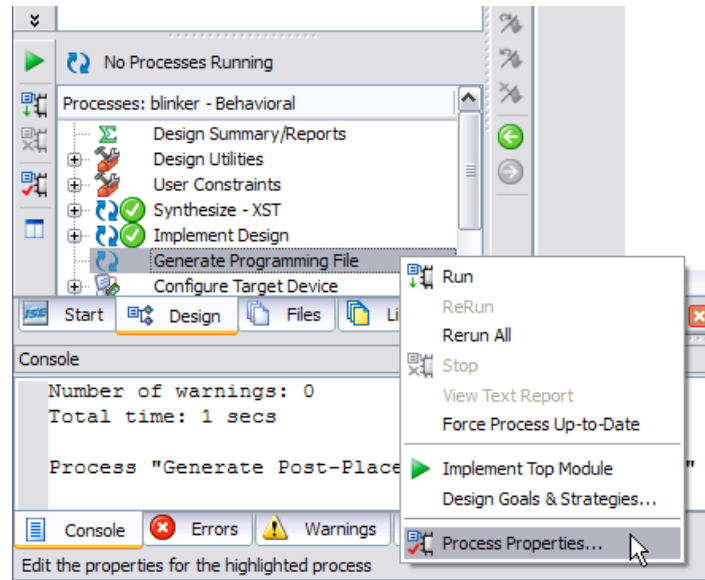
In the **Process Properties** window that appears, click on the **Configuration Options** category.
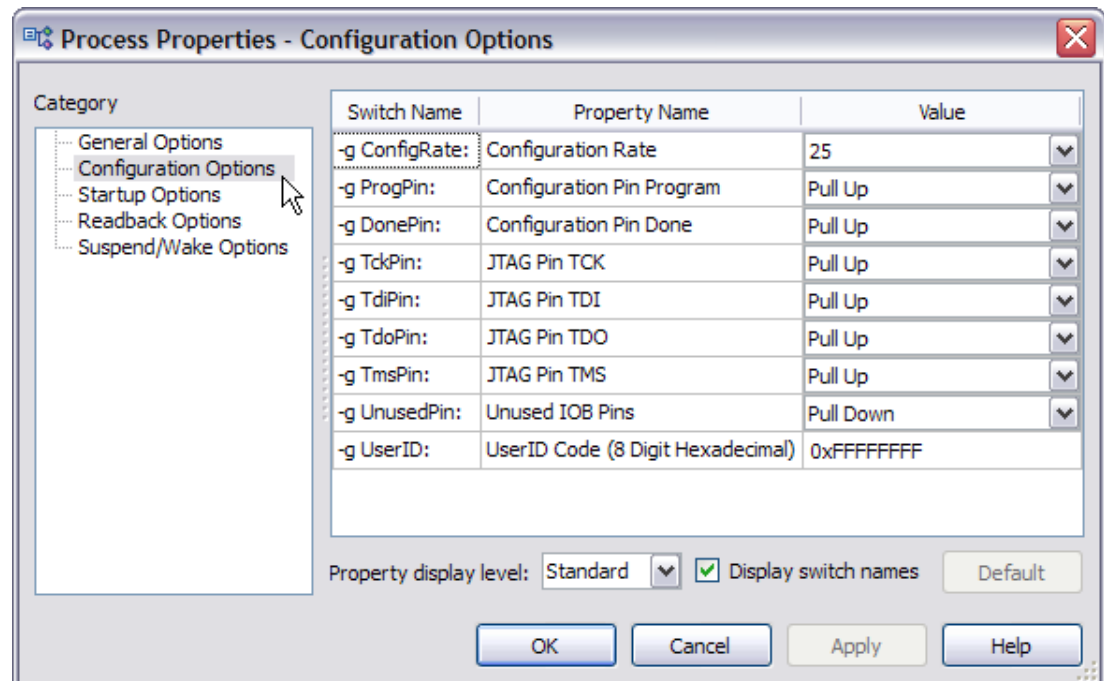
Then, click on the **Unused IOB Pins** drop-down menu and select **Float** from the list of options. This will cause all FPGA I/O pins that are not specifically used in your design to float in a high-impedance state.
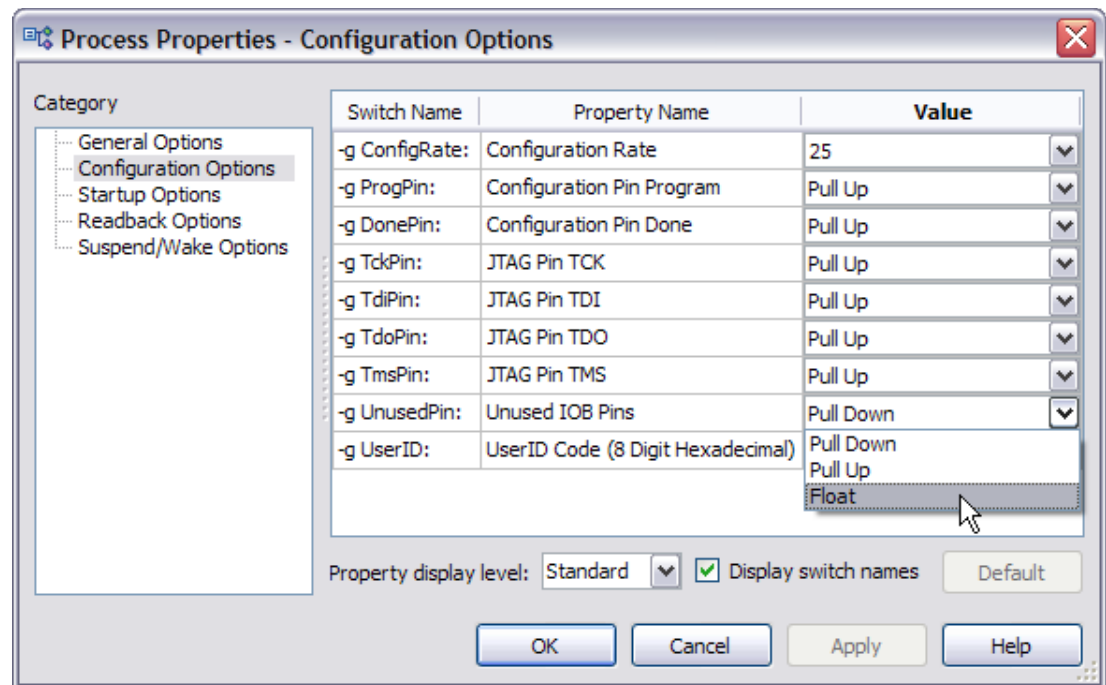
Now click on the **Startup Options** category and change the **FPGA Start-Up Clock** to **JTAG clock**. The FPGA on the XuLA board is programmed through the JTAG port and the clock that kickstarts your design right after downloading also comes through there. If you select any other option, your design will just sit in the FPGA and not do anything because it hasn't gotten a startup clock. This is always a good thing to check if you have a design that acts like that.
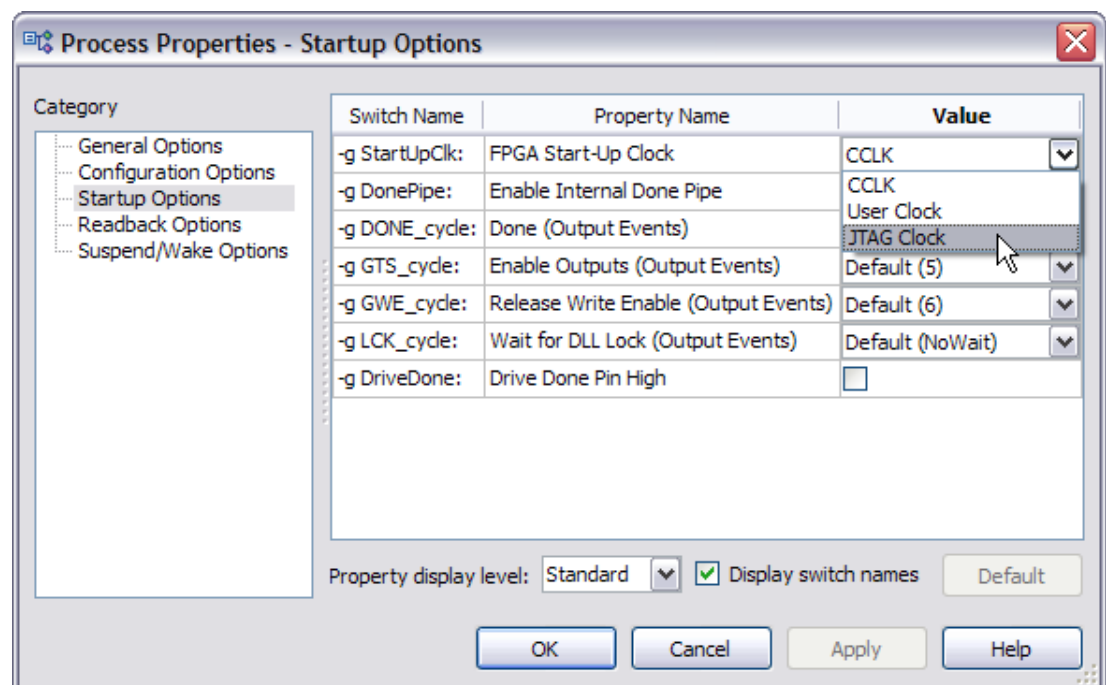
After setting these two options, click on the **OK** button and return to Navigator. Now you can double-click **Generate Programming File** to build the bitstream for the blinker project.
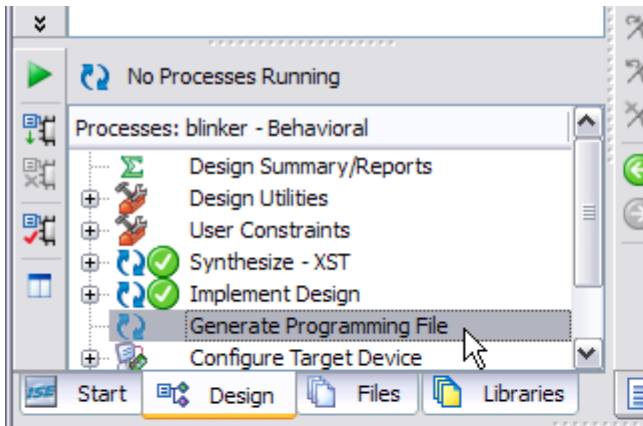
If the bitstream is generated successfully, you should be rewarded with the sight of three green check-marks!
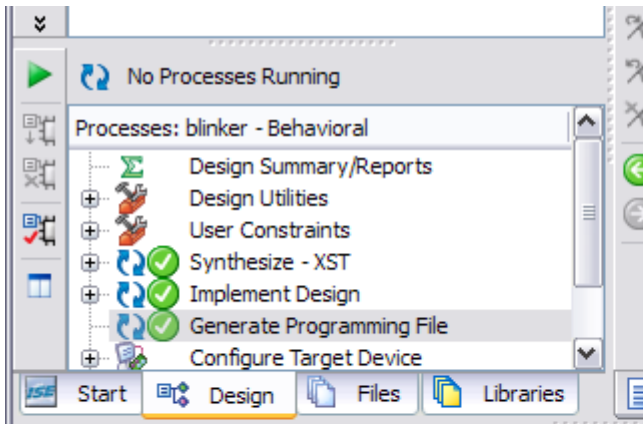
Now that you have the blinker project bitstream, we will leave Navigator and use a XuLA-specific tool to download the bitstream.

Installing the XSTOOLs Utilities

To download bitstreams to the XuLA board, you'll need to install the XSTOOLs utilities. You can download them here. After you download the installer, just double-click it and accept the default installation settings. You'll download the blinker bitstream to the XuLA board using the GXSLOAD utility that gets installed.

Connecting the XuLA Board

Before you download the bitstream, you'll need to hook a resistor and LED to the XuLA Board. This is relatively easy. First, insert a XuLA board into a breadboard.
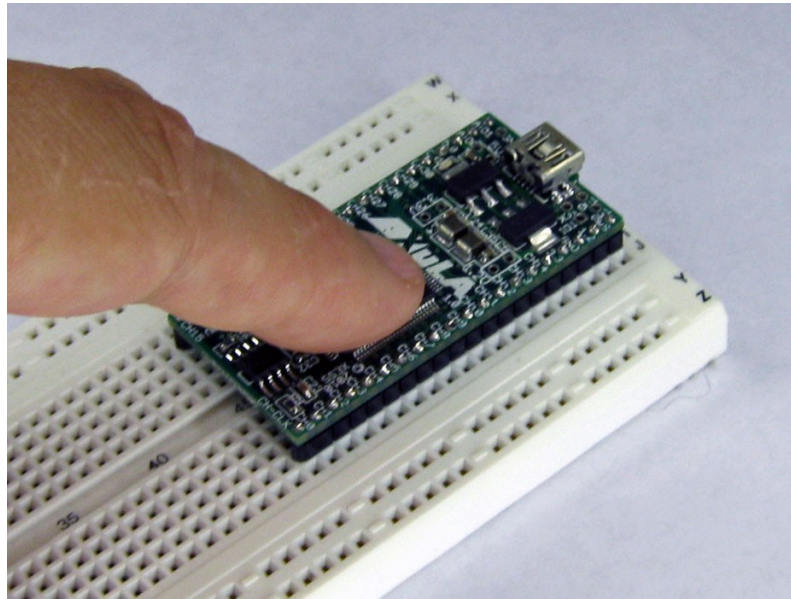
Then connect the ground connection of the XuLA board to the ground bus of the breadboard.
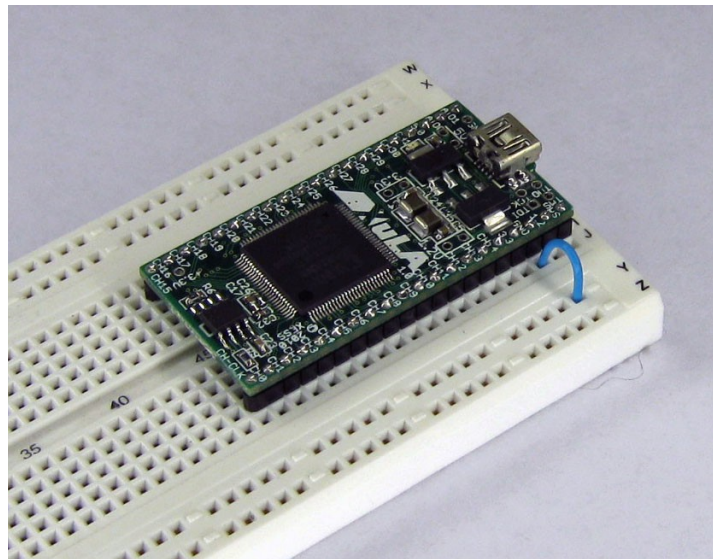
Add a resistor from the prototyping header pin that carries the blinker_o signal. (The resistor value can be anywhere in the range of 100Ω to 500Ω.)
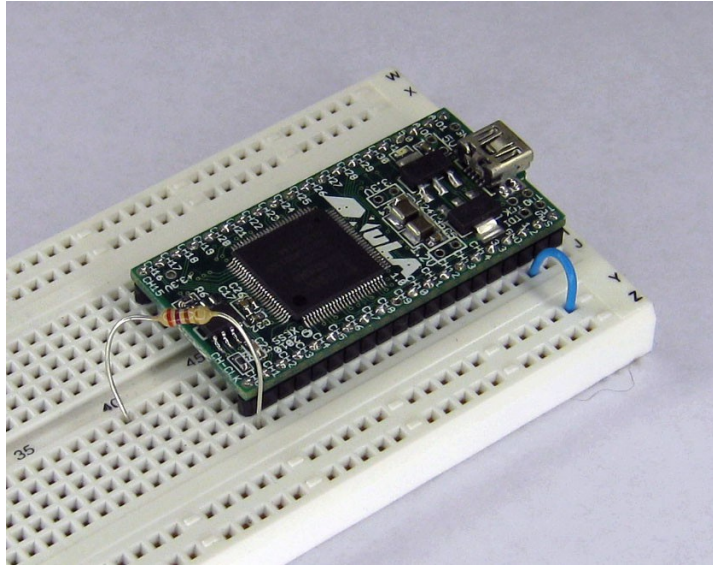
Insert an LED with the anode (the rounded side of the case) connected to the resistor and the cathode (the flat side of the case) plugged into an empty breadboard row. (I apologize – the leads on my diode were too short and you can't actually see where they plug into the breadboard.)
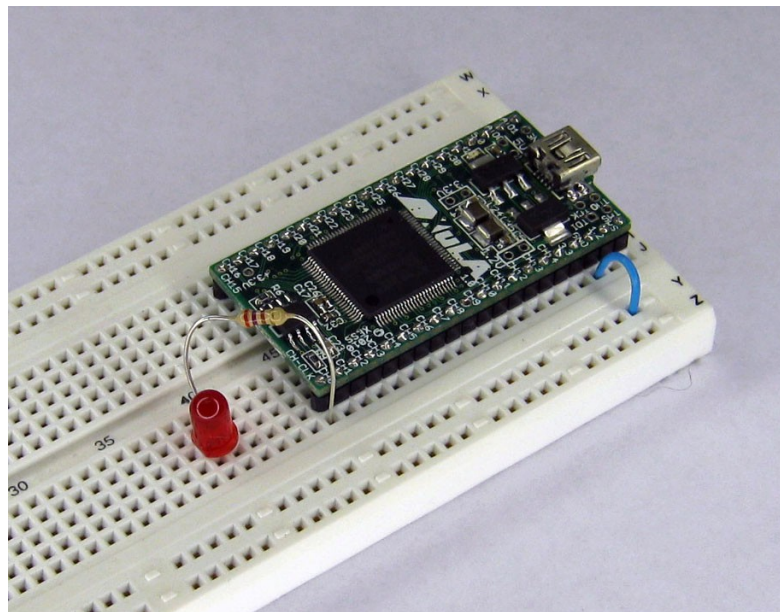
Then use a jumper wire to connect the LED's cathode to the ground bus.
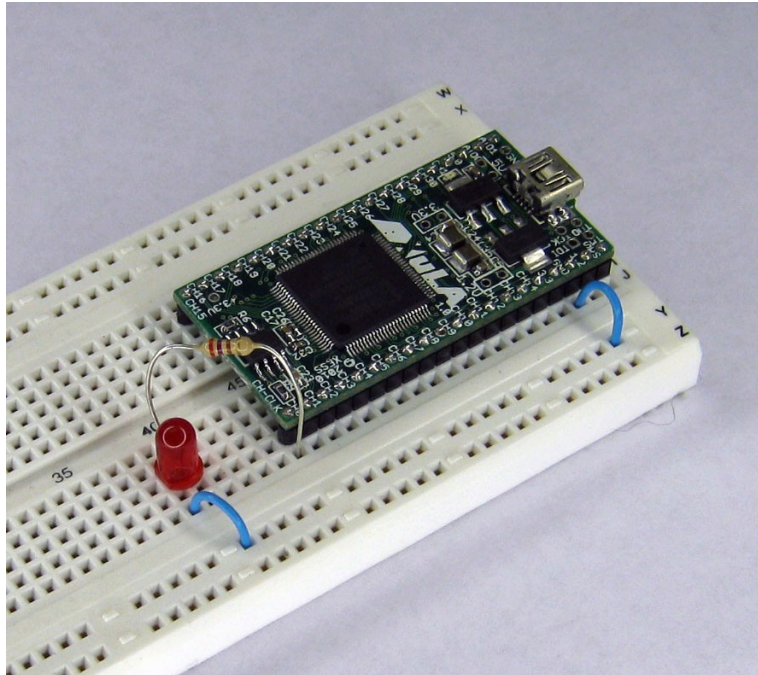
Finally, connect a mini-USB cable from the XuLA board to the PC that you installed the XSTOOLs utilities on.
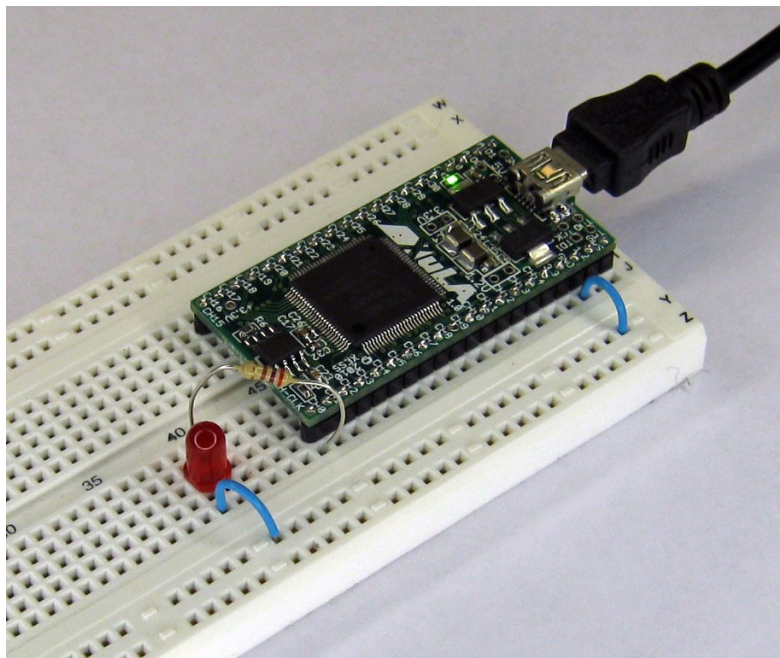
That's it! Time to dance.

Downloading the Blinker Bitstream

To begin the downloading process, double-click the GXSLOAD icon. This will bring up the following window:
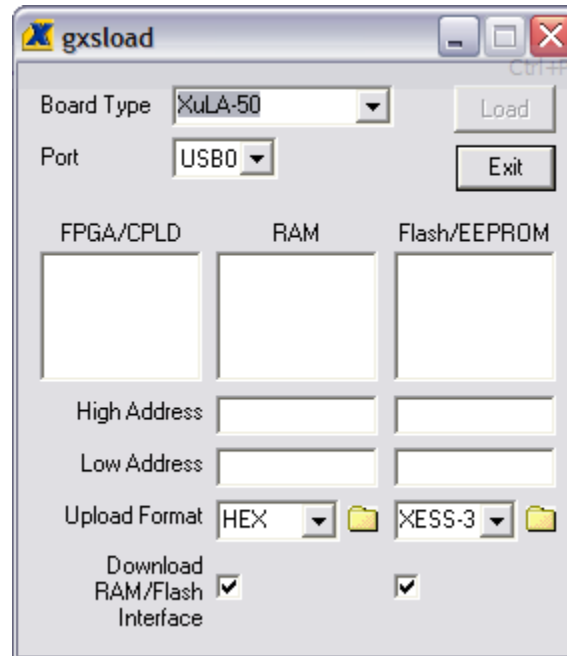
Select the **Board Type** using the drop-down list. Since the blinker project was targeted at the XC3S200A, then you must be using a XuLA-200 board. (If you were using a XuLA-50 board, then your target FPGA would have been the XC3S50A.)
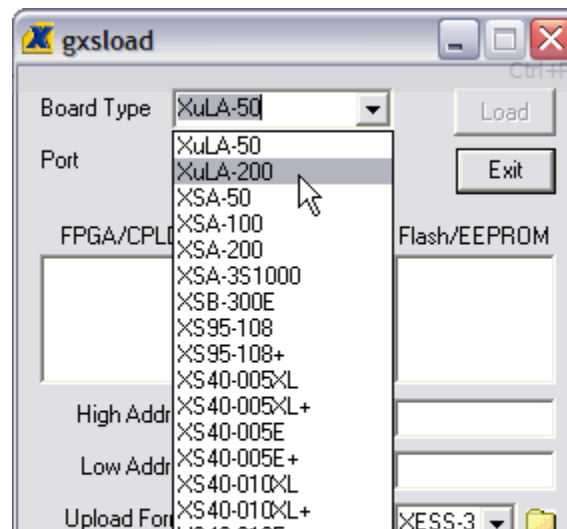
As for the **Port**, it should be set to **USB0** (if you only have one XuLA board connected to the PC).

Next, open the folder where you stored your blinker project. Locate the blinker.bit bitstream file and drag-and-drop it into the **FPGA/CPLD** pane of the **gxsload** window.
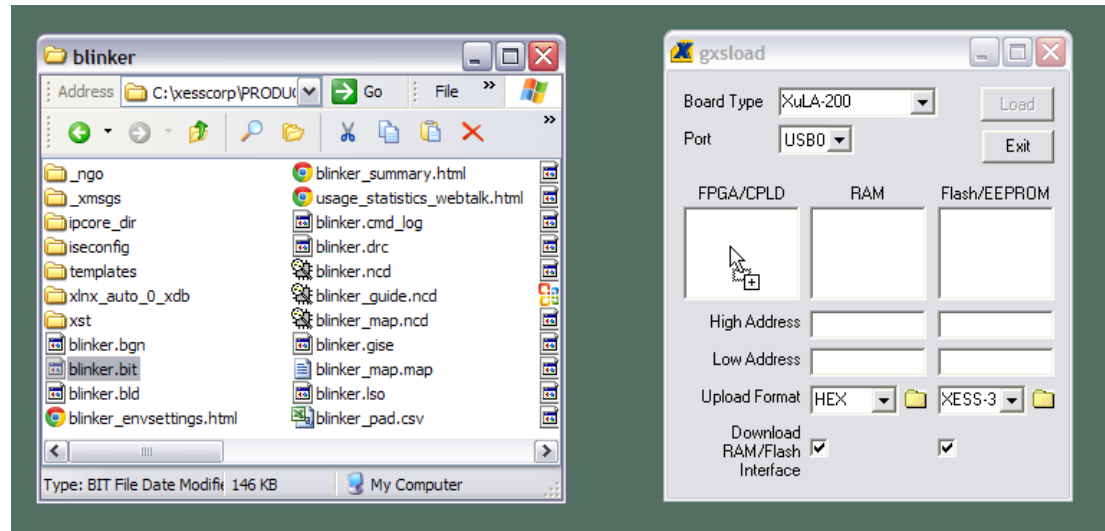
Once the bitstream file is dropped into the **FPGA/CPLD** pane, the **Load** button becomes active (i.e., it's no longer grayed-out). Just click on the **Load** button and the blinker bitstream will download from the PC into the FPGA on the XuLA board.
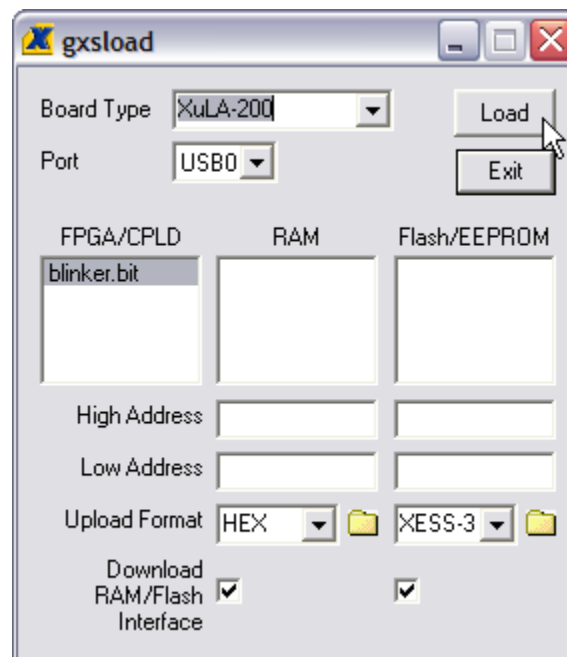
When the download begins, the green LED on the XuLA board will flash as the bitstream enters the FPGA. This LED will stop flashing once the complete bitstream has entered the FPGA. Then the blinker circuit will take control of the FPGA and it will start to blink the other LED you inserted into the breadboard. This video shows the downloading and operation of the blinker. If you time the flashing of the LED, you'll see it is pulsing once every 0.7 s as we hoped.

This completes your first design. You coded it, synthesized it, simulated it, implemented it, generated it, built it, downloaded it and ran it. You are officially an FPGA designer. A beginner, true, but an FPGA designer none the less.

C.5 *"Only 12 MHz! Now what?"*

Speed Envy

In the last chapter, some of you were probably saying: "A 12 MHz clock? That's all!? My Arduino runs faster than that. My *grandmother* runs faster than that!" (Maybe your grandmother, but not mine, both of whom died long ago. If you see them up and running around, then something's *really* gone wrong.)

One of the unfortunate consequences of the popularization of computers has been people's fixation on clock speed, with more being better. But fear not that your Arduino-toting compatriots will laugh at you, because the FPGA on the XuLA board has a feature that will let it run at over 300 MHz.

The Digital Clock Manager

Modern Xilinx FPGAs contain *Digital Clock Managers* (or DCMs) that serve many functions, one of which is multiplying the frequency of a clock. A *Digital Frequency Synthesizer* (DFS) within the DCM can multiply an incoming clock signal by any integer in the range 2..32 and then divide it by any factor in the range 1..32. Here's an example of generating an 18 MHz clock:
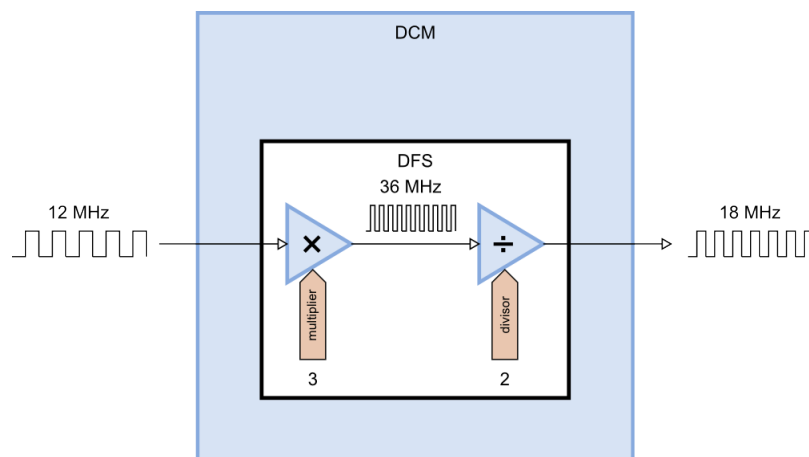
With a 12 MHz input clock and a multiplier of 32, you can theoretically generate a clock up to 384 MHz, but the practical electrical limit for a Spartan-3A device is between 320 to 350 MHz (depending upon the speed grade of the FPGA).

Adding a DFS to the Blinker

Now, how am I going to demonstrate the operation of a DFS to you if you don't have an oscilloscope to view the incoming and outgoing clock signals? Simple! We can take the blinker design from the previous chapter, add a DFS to multiply the input clock, and then see if the blinker pulses faster.

So the first thing to do is to add the DFS to the blinker VHDL file. But how do you describe a DFS in VHDL? After all, it's not something simple like a NAND gate or a counter.

Well, Xilinx has made it easy for you by creating a *device primitive* for the DFS. (It's called a primitive because it can't be decomposed into a collection of simpler components. Perhaps *atom* would have been a better name.) You can instantiate the DFS device primitive into your VHDL source and not have to worry about how it is actually built.

But you need to know what the inputs and outputs of the DFS primitive are in order to use it. To get this information, click on the **Edit ⇨ Language Templates...** menu item. This will make the Language Templates tab appear in the workspace pane like so:
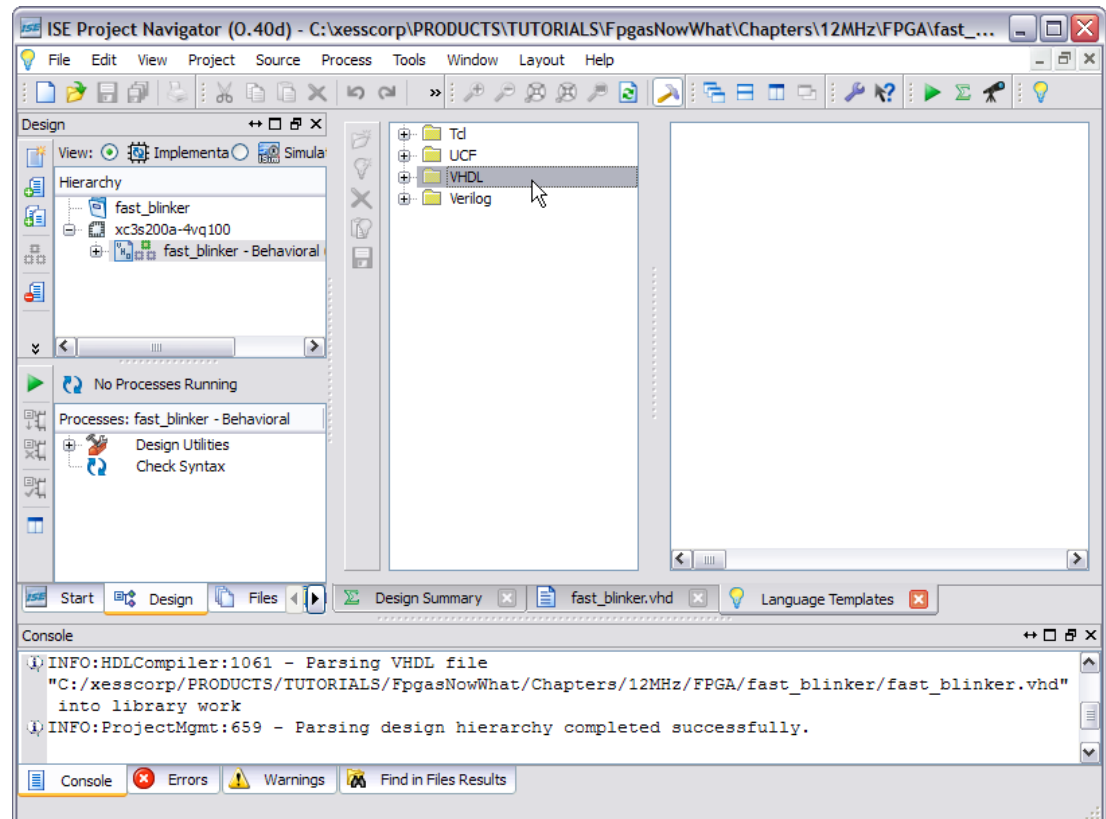
To find the DFS primitive, click on the **VHDL** folder and expand it until you see the DCM item appear. Then select the DCM item and a VHDL template for instantiating this primitive will appear in the rightmost pane.
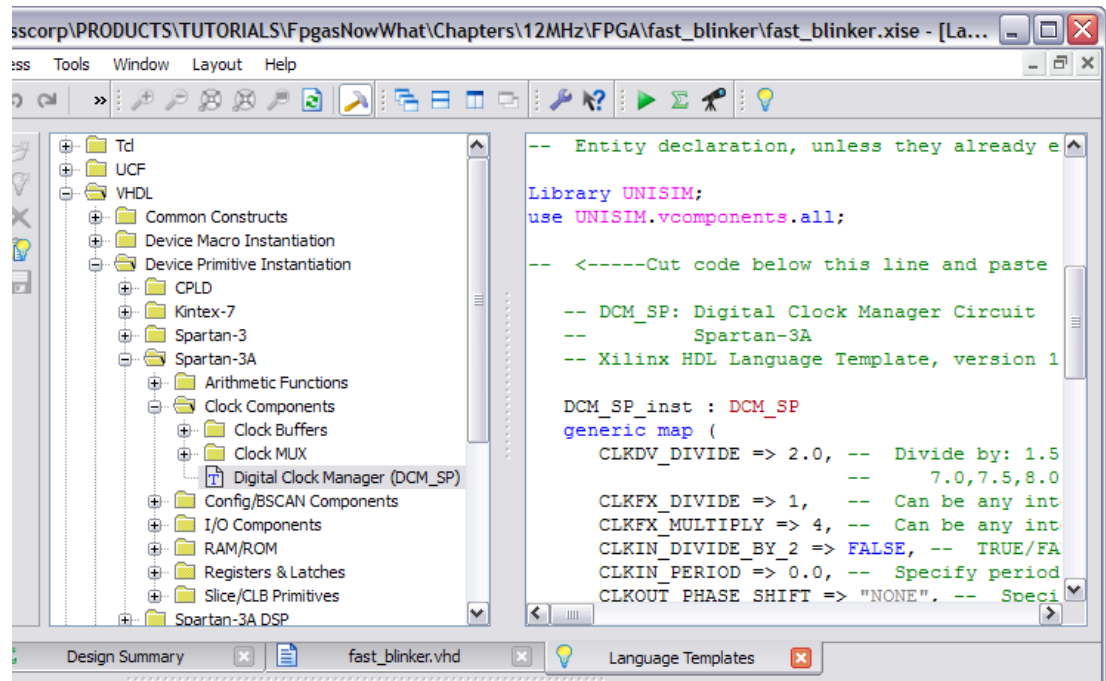
The VHDL template is intimidating! There are eighteen inputs and outputs and twelve generic parameters for tuning the operation of the DCM primitive. But we can ignore most of them and accept their default settings. The only things we care about for this application are the reset (`RST`), input clock (`CLK_IN`), the DFS output clock (`CLK_FX`), and the DFS multiplier (`CLKFX_MULTIPLY`) and divider (`CLKFX_DIVIDE`).

After instantiating the DCM and doing some trimming and editing, the blinker VHDL looks like what's shown on the following page. Here's what changed:

Lines 26-27: Since the DCM primitive is being used, these lines need to be uncommented to bring in the Xilinx primitive libraries.

Line 35: A new signal, `clk_fast`, was added to carry the clock generated by the DFS.

Lines 39-48: The DCM is instantiated here. The DFS divider and multiplier are set to one and four, respectively, on lines 41 and 42. The DFS output is connected to the `clk_fast` signal on line 45, and the input clock is connected to the DFS clock input on line 46. The reset input is tied to an inactive level on line 47.

Lines 50-55: The clock for the counter was changed from `clk_in` to `clk_fast`.

```
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_UNSIGNED.ALL;
23
24  -- Uncomment the following library declaration if instantiating
25  -- any Xilinx primitives in this code.
26  library UNISIM;
27  use UNISIM.VComponents.all;
28
29  entity fast_blinker is
30      Port ( clk_i : in  STD_LOGIC;
31             blinker_o : out  STD_LOGIC);
32  end fast_blinker;
33
34  architecture Behavioral of fast_blinker is
35  signal clk_fast : std_logic;
36  signal cnt_r : std_logic_vector(22 downto 0) := (others=>'0');
37  begin
38
39  DCM_SP_inst : DCM_SP
40     generic map (
41        CLKFX_DIVIDE   => 1, --  Can be any interger from 1 to 32
42        CLKFX_MULTIPLY => 4  --  Can be any integer from 1 to 32
43     )
44     port map (
45        CLKFX => clk_fast,  -- DCM CLK synthesis out (M/D)
46        CLKIN => clk_i,     -- Clock input (from IBUFG, BUFG or DCM)
47        RST   => '0'        -- No reset
48     );
49
50  process(clk_fast) is
51  begin
52    if rising_edge(clk_fast) then
53      cnt_r <= cnt_r + 1;
54    end if;
55  end process;
56
57  blinker_o <= cnt_r(22);
58
59  end Behavioral;
```

After the VHDL file is finished, you can run the synthesizer to see if there are any problems. (There aren't, at least when I did it).

Does It Work?

That's always the question, especially when you are applying a chip feature that you haven't used before. So let's run a simulation to find out. First, add a new test bench to your project:
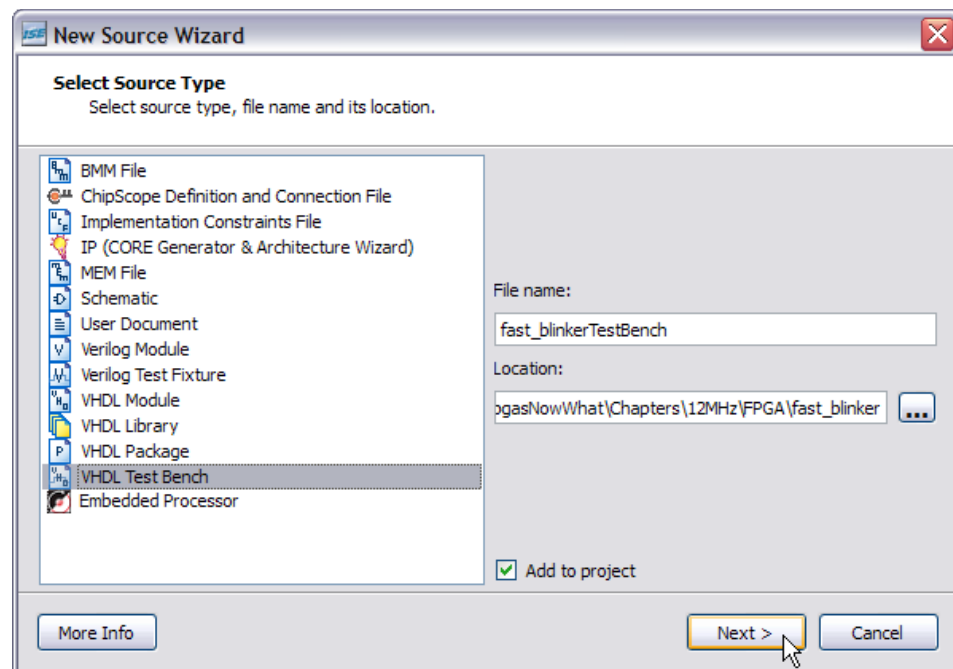
Then go into the test bench VHDL file and change the clock period to match the 12 MHz clock period:

Then switch to the **Simulation** view of your project and click on the **Simulate Behavioral Model** item in the **Processes** pane. When the **ISim** window appears, enter `2.8s` in the simulation duration field and then click on the run button.
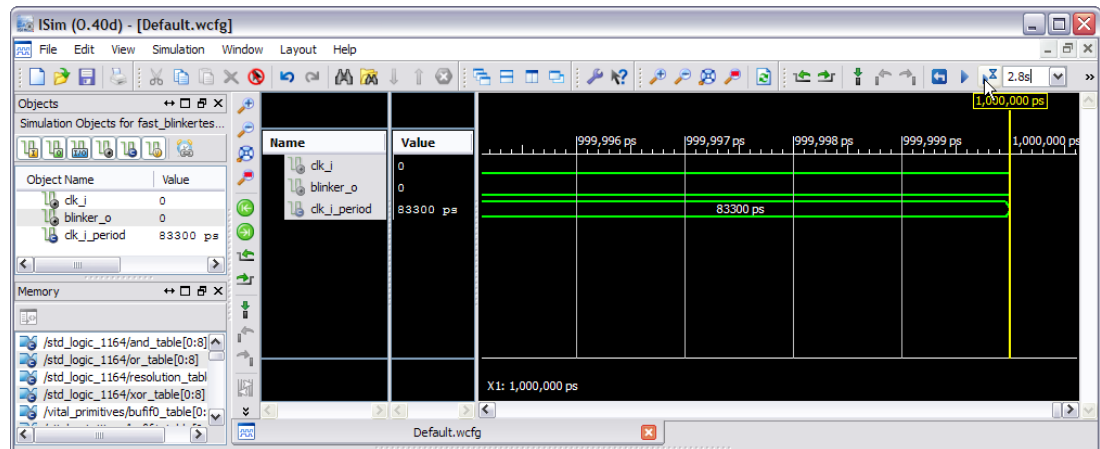
Then be prepared to wait … for … a … very … long … time. On my PC the simulation took about 30 minutes, probably because simulating the DFS is computationally demanding. After the simulation finishes, compress the view so the entire 2.8 s time interval is visible and count the number of blinker pulses. You should see sixteen pulses. Remember in the previous blinker design, there were only four pulses in the 2.8 s window. So it appears – at least in the simulation – that the DFS has multiplied the 12 MHz clock by a factor of four.
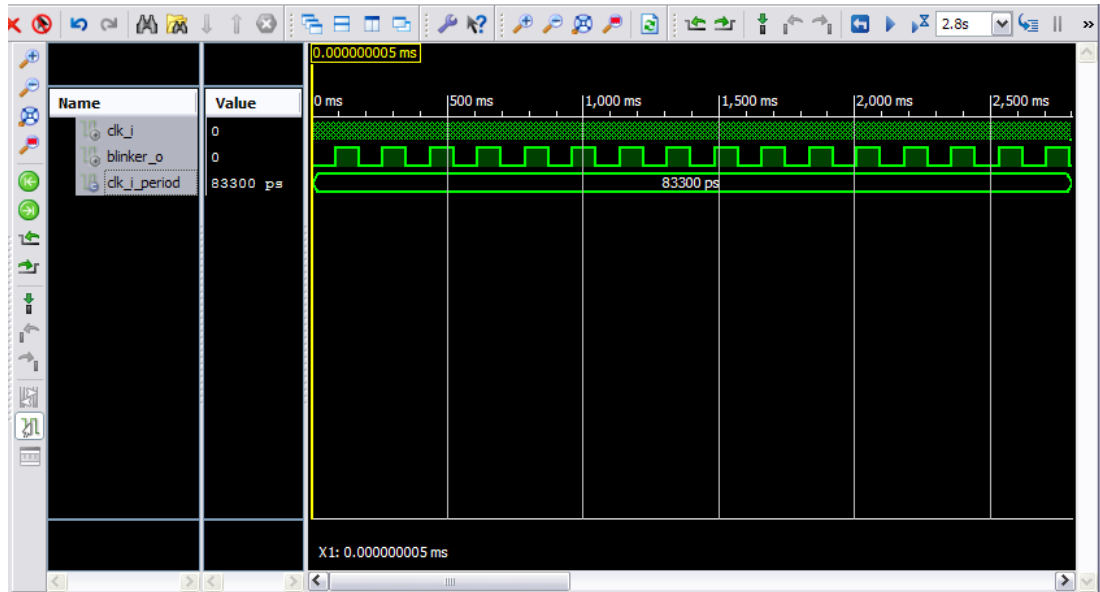
But Does It *Really* Work?

That's always the *real* question: Will it work in the FPGA? Well, it's easy enough to find out. Here are the high points for how to do that:

1. Connect an LED and resistor to the XuLA board just as you did for the previous blinker project. Then connect the XuLA board to a USB port on your PC.

2. Add a constraints file to your project that's identical to the one you used for the

previous blinker project since the LED and clock pin assignments haven't changed.

3. Right-click the **Generate Programming File** process item and select **Process Properties...** from the pop-up menu. Then select the options to float any unused I/O pins after configuration and to use the JTAG clock as the start-up clock.

4. Double-click the **Generate Programming File** process. This will run the synthesis and implementation tools in sequence and then create a downloadable bitstream.

5. Start the GXSLOAD tool and drag-and-drop the fast_blinker.bit file into the **FPGA** pane. Then click the **Load** button to download the bitstream into the FPGA on the XuLA board.

After doing these steps, you should be rewarded by seeing the LED attached to the XuLA board blinking away at about six times per second.

C.6 *"No LEDs! No buttons! Now what?"*

Blinky Envy

By now you're probably saying to yourself: "Trying to tell what my design is doing by looking at a flashing LED really sucks! All the other FPGA boards have loads of flashing LEDs. And they have bunches of buttons and switches! My XuLA board has *nothing*!"

Now when you learned how to ride a bike, you might have started out using training wheels. These were helpful – they let you get experience with the bike without getting hurt (unless your parents *push you out in a busy street* – but that's another story). But after a while, you couldn't wait to get rid of those training wheels. Wherever you went, they just screamed out: "I'm a bike-riding noob! Come give me a wedgie!" But the worst part is they got in the way; you couldn't get any speed up because those training wheels were dragging on the ground. Luckily, they weren't welded to the bike and you could take them off. Then you could use the bike for what it was for: getting you from here to there faster and easier than being on foot.

LEDs and buttons are similar to training wheels: good when getting started, but a hindrance after that. LEDs and buttons are meant to operate at human interaction speeds, like 10 Hz or less. But FPGAs are meant for applications that run at hundreds of MHz; it's a waste to have them waiting around while you look at an LED and then push a button to tell them what to do! Plus, they're hard to use for anything but the simplest designs: do you really want to flip switches to input a three-digit hexadecimal number and then translate the pattern on a bunch of LEDs back into hex? (If you do, then I've got a PDP-8 in the basement with your name on it!)

But LEDs and buttons are different from training wheels in one important aspect: they aren't easy to take off when you're done with them. They're usually soldered to the board where they take up space (and make the board larger than it needs to be). They interfere electrically when you use the FPGA pins for other purposes (unless there are jumpers to disconnect them, taking up even more space). And they're never connected to the pin you want them on (although this is less of a problem with FPGAs since you can move your I/O signals from pin to pin as you please).

Still, there is no denying that you need some way to observe what your design is doing. Logic probes are one solution (sort of a portable LED), but limited in the amount of information they can present (just like an LED). Oscilloscopes are good for capturing signal behavior, especially at high frequencies, but most scopes have four inputs or less

and they cost a fair bit of money (around $300 for a minimal scope). Logic analyzers provide a lot of inputs that can be sampled at hundreds of MHz, but they are truly a pain to hook up to your circuit (but they are much cheaper now: as little as $50). A companion to the logic analyzer is the digital pattern generator which can drive signals into your design and which suffers from the same connection difficulties as the analyzer.

The ideal solution for observing what your application circuit is doing would have these characteristics:

- Free.
- Flexible.
- Easy to use.
- Takes up no space.
- Uses no FPGA pins.
- Lets you drive inputs into your circuit and also observe its response.
- Drives inputs into your circuit and observes its response *automatically*.
- Goes away when you no longer need it.

This sounds like a fantasy product, but you actually already have it! The FPGA itself can provide the circuitry you need to monitor and control your own application circuit. Here's the basic idea: shift a test vector into the FPGA, apply it in parallel to the inputs of your circuit, load the output response of your circuit into the shift register and shift it out of the FPGA (see the figure below).

You need three things to make this work:

1. Hardware modules are needed for the interface through the FPGA's JTAG port, the shift register, and other housekeeping details.

2. Firmware is needed for the microcontroller on the XuLA board so it can transfer the test vectors and responses between the FPGA's JTAG port and the USB port of a host PC.

3. Software is needed on the host PC to make it easy to generate, transmit, receive and display the test vectors and responses.

Now that's a lot of stuff to build and test, but luckily I've already done that for you. All you have to do is apply it to your own circuits. And I'll show you an example of how to do that using our old friend – you guessed it – the LED blinker!
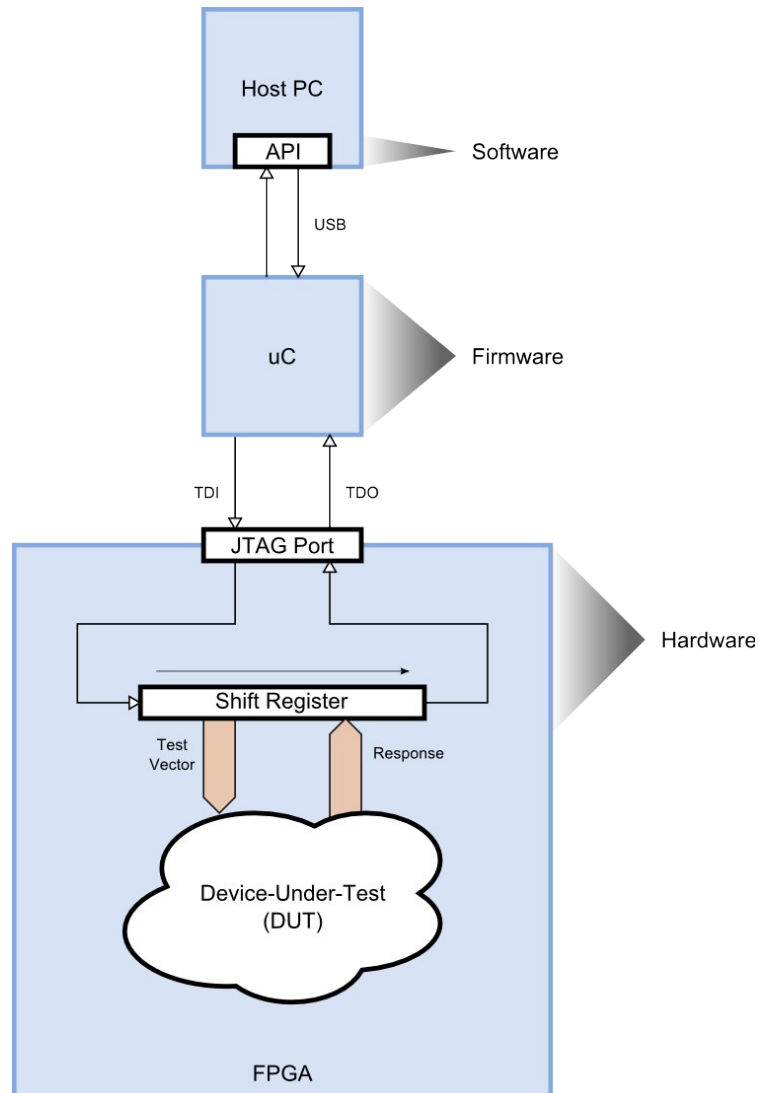
Monitoring the LED Blinker from the Host PC

Modifying the LED Blinker Project

Before diving into the VHDL code, it might be helpful to get a picture of what we're trying to do. The original LED blinker circuit from Chapter 3 is shown at the bottom of the figure below: it takes a 12 MHz clock, divides the frequency down using a counter, and attaches one bit of the counter to an LED so it flashes about once per second. The counter bit is also loaded into a shift-register (although it's not *just* a shift-register) where its value can be shifted out of the FPGA through the JTAG port and on to a PC. The PC could also shift in bits to drive inputs to the counter, but the counter has no inputs so these signals are left dangling. There are a few other signals running around in the background that aren't shown so I don't complicate the very simple technique that is being used here. Don't worry, you'll be seeing them soon.
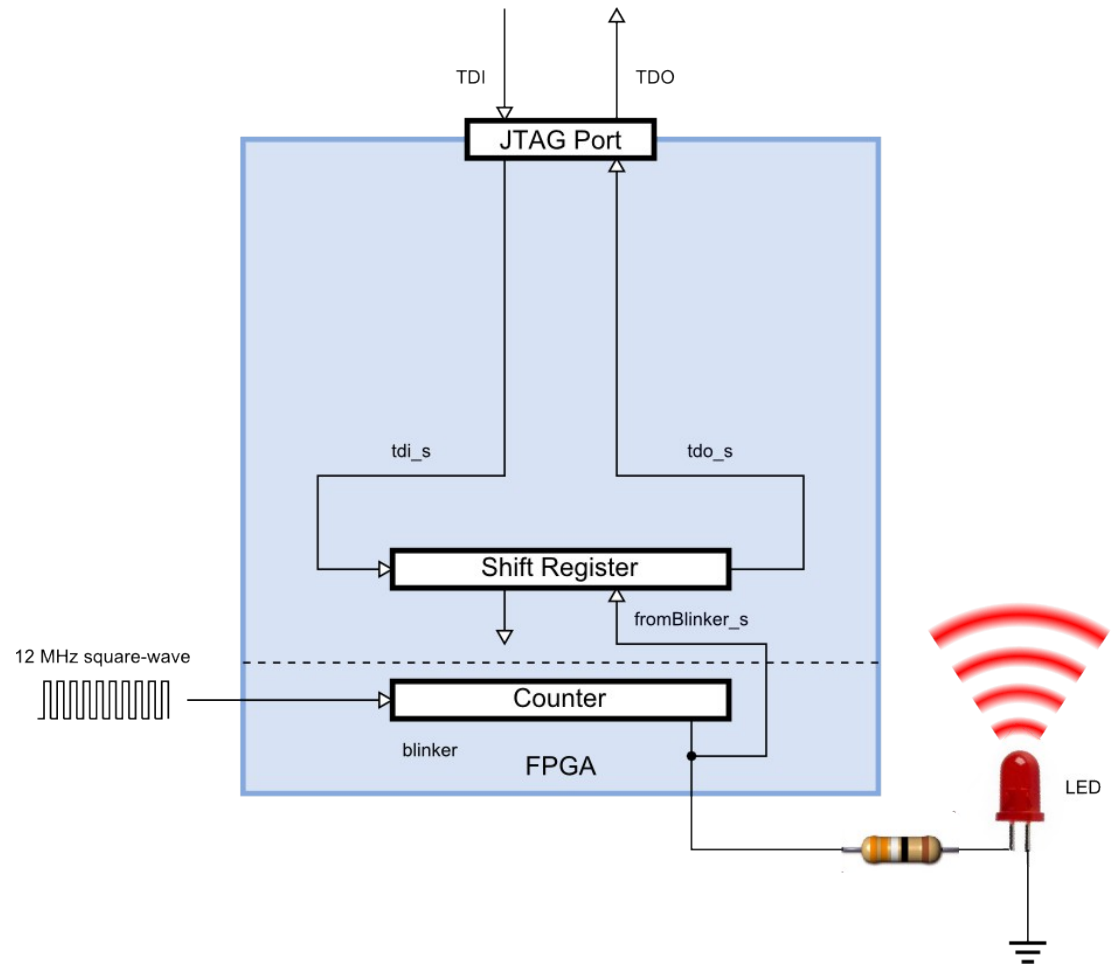
Now on to the VHDL. The list of files needed to add the JTAG interface and shift-register modules to the LED blinker project are shown below. Here's what each one does:

HostIo.vhd: This file contains the VHDL for the JTAG entry point and shift-register modules that will let the host PC talk to the blinker circuit in the FPGA.

Common.vhd: Some helpful VHDL functions that are used in the HostIo.vhd code are defined in this file.

SyncToClk.vhd: This file describes some synchronization circuitry that is used in the HostIo.vhd file, but which has no role to play in this example. You need it to keep the VHDL synthesizer from complaining about undefined modules.

pc_blinker.vhd: This is where the original LED blinker VHDL code is augmented with the modules found in HostIo.vhd.

pc_blinker.ucf: The same pin assignments for the original LED blinker are repeated in this file.
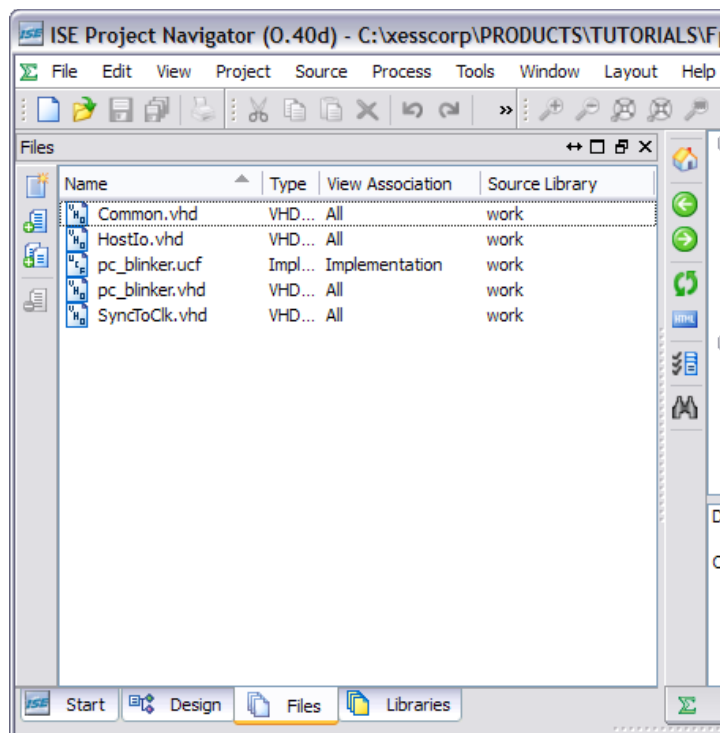
The pc_blinker module is where we want to concentrate our attention. Let's take a look inside:

The first three lines are the same as for the original LED blinker. They just import some standard packages from the IEEE library. But the fourth line is new: it imports the JTAG entry point and shift-register modules from the HostIo.vhd file. The `work` library just refers to the directory where the whole project is stored; `HostIoPckg` refers to the package of modules defined in the HostIo.vhd file; `all` just says to import every module found in the HostIoPckg (although I could have been more discriminating and just imported the modules that were needed by this design).

```
1    library IEEE;
2    use IEEE.STD_LOGIC_1164.all;
3    use IEEE.STD_LOGIC_UNSIGNED.all;
4    use work.HostIoPckg.all;  -- Package for PC <=> FPGA communications.
5
```

The next few lines declare the interface for the LED blinker. It is identical to the entity

declaration used in the original LED blinker. Even though we are going to monitor the LED blinker with the PC, the signals that pass in and out of the JTAG port do not need to be declared here.

```
 6   entity pc_blinker is
 7     port (clk_i     : in  std_logic;
 8           blinker_o : out std_logic);
 9   end entity;
10
```

The architecture section comes next. The 23-bit counter for the LED blinker is declared on line 12. Lines 14 and 15 declare two, 1-bit vectors: one for carrying bits from the PC to the blinker's inputs, and one to carry the outputs of the blinker back to the PC. (There are no inputs to the blinker, but a dummy signal is needed just to have something to connect to the shift-register module.) Even though these are both single-bit signals, they need to be declared as std_logic_vectors in order to match the declared I/O types of the modules in the HostIoPckg.

Lines 17-20 declare the signals that connect from the JTAG entry point to the shift-register. A selection signal and a clock are declared which are used to shift bits in and out of the register on the tdi_s and tdo_s signals.

```
11   architecture Behavioral of pc_blinker is
12     signal cnt_r        : std_logic_vector(22 downto 0) := (others => '0');
13     -- Connections between the shift-register module and the blinker.
14     signal toBlinker_s   : std_logic_vector(0 downto 0); -- From PC to blnkr.
15     signal fromBlinker_s : std_logic_vector(0 downto 0); -- From blnkr to PC.
16     -- Connections between JTAG entry point and the shift-register module.
17     signal inShiftDr_s   : std_logic; -- True when bits shift btwn PC & FPGA.
18     signal drck_s        : std_logic; -- Bit shift clock.
19     signal tdi_s         : std_logic; -- Bits from host PC to the blinker.
20     signal tdo_s         : std_logic; -- Bits from blinker to the host PC.
21   Begin
22
```

Next up is circuitry for the LED blinker. It's identical to what you've already seen in Chapter 3.

```
23     ----------------------------------------------------------------------
24     -- Application circuitry: the LED blinker.
25     ----------------------------------------------------------------------
26
27     -- This counter divides the input clock.
28     process(clk_i) is
29     begin
30       if rising_edge(clk_i) then
31         cnt_r <= cnt_r + 1;
32       end if;
33     end process;
34
35     blinker_o <= cnt_r(22); -- This counter bit blinks the LED.
36
```

The JTAG entry point module follows. The I/O for this module connects to the internal signals declared on lines 17-20.

```
37     ----------------------------------------------------------------------
38     -- JTAG entry point.
39     ----------------------------------------------------------------------
40
41     -- Main entry point for the JTAG signals between the PC and the FPGA.
42     UBscanToHostIo : BscanToHostIo
43       port map (
44         inShiftDr_o => inShiftDr_s,
45         drck_o      => drck_s,
```

```
46           tdi_o        => tdi_s,
47           tdo_i        => tdo_s
48           );
49
```

Finally, there's the shift-register module that grabs the output from the blinker counter and sends it back to the PC. Line 55 assigns an identifier of '1' to this module so the PC can select it for access (I know this seems unnecessary, but it will make sense later). Lines 58-61 tie this module to the JTAG entry point module. And line 67 - arguably the most important of all - connects the bit of the counter that drives the LED to the shift-register so it can be loaded and shifted back to the PC.

```
50      ----------------------------------------------------------------------
51      -- Shift-register.
52      ----------------------------------------------------------------------
53
54      -- This is the shift-register module between blinker and JTAG entry point.
55      UHostIoToBlinker : HostIoToDut
56        generic map (ID_G => "00000001") -- The identifier used by the PC.
57        port map (
58           -- Connections to the BscanToHostIo JTAG entry-point module.
59           inShiftDr_i    => inShiftDr_s,
60           drck_i         => drck_s,
61           tdi_i          => tdi_s,
62           tdo_o          => tdo_s,
63           -- Connections to the blinker.
64           vectorToDut_o   => toBlinker_s,    -- From PC to blinker (dummy sig).
65           vectorFromDut_i => fromBlinker_s  -- From blinker to PC.
66           );
67
68      fromBlinker_s <= cnt_r(22 downto 22); -- Blinker output to shift reg.
69
70   end architecture;
```

Once the pc_blinker.vhd and the other files are included in the pc_blinker project, the Hierarchy pane looks like this:
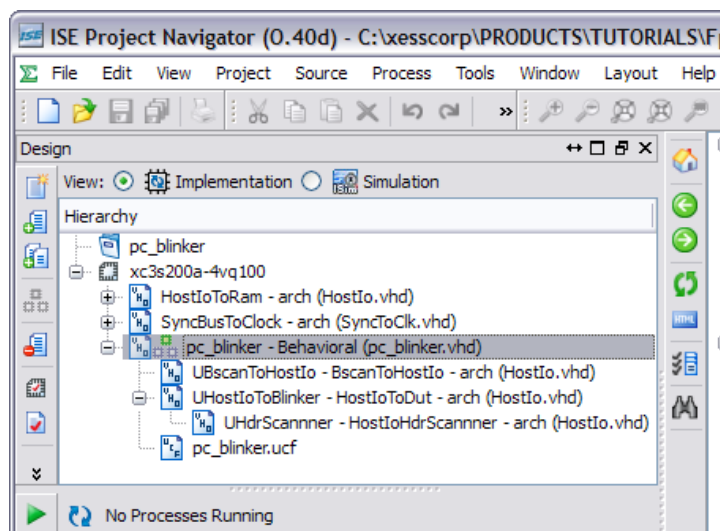
The HostIoToRam and SyncBusToClock modules aren't used in this project; they're just showing up because their code is included in the VHDL files. In order to make sure the pc_blinker design is compiled and not one of these, right-click on the pc_blinker identifier and select **Set as top module** from the pop-up menu.

FPGAs!? Now What?
TUT001 (V1.0) Feb 28, 2013

www.xess.com

78
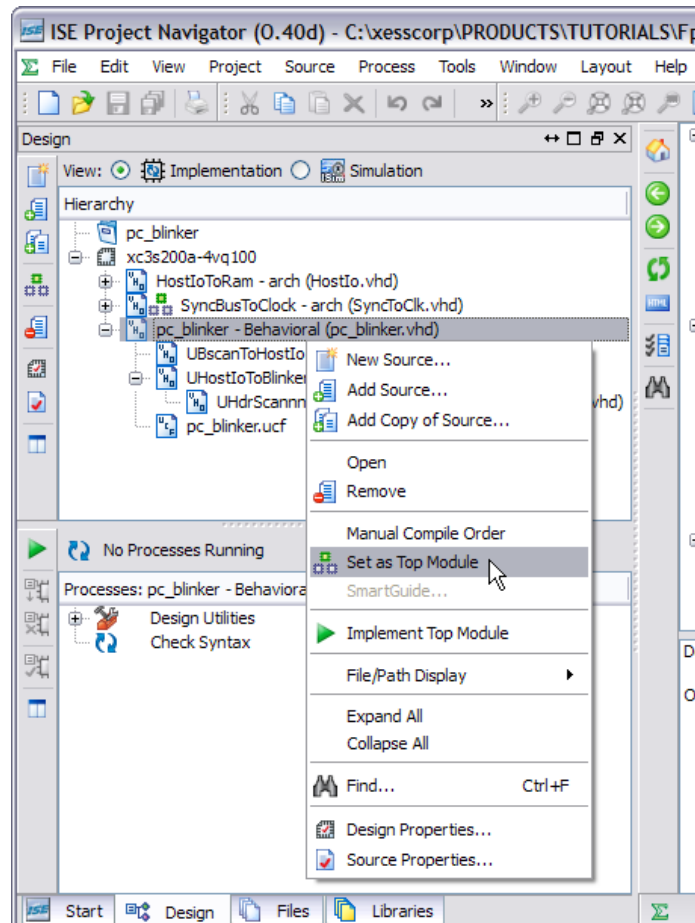
Before generating the bitstream, remember to change the configuration settings to make the it compatible with the XuLA board. Right-click on **Generate Programming File** and select **Process Properties...** from the pop-up menu.
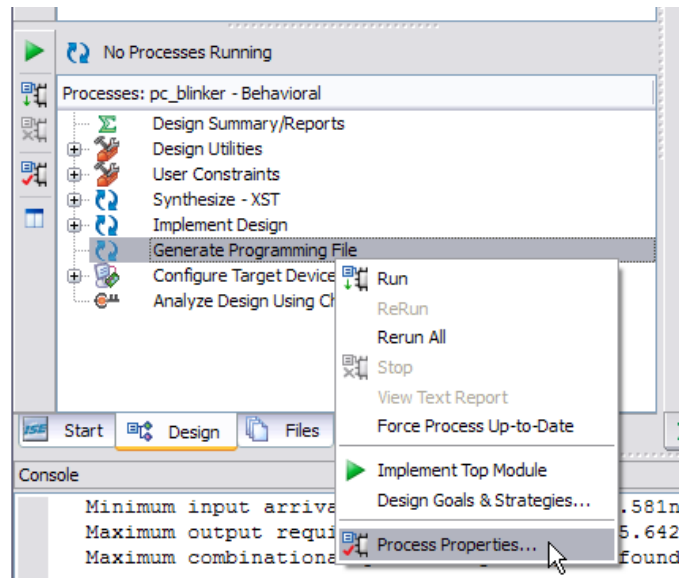
In the **Configuration Options** tab of the **Process Properties** window that appears, click on the **Unused IOB Pins** drop-down menu and select **Float** from the list of options. This will cause all FPGA I/O pins that are not specifically used in this design to float in a high-impedance state.
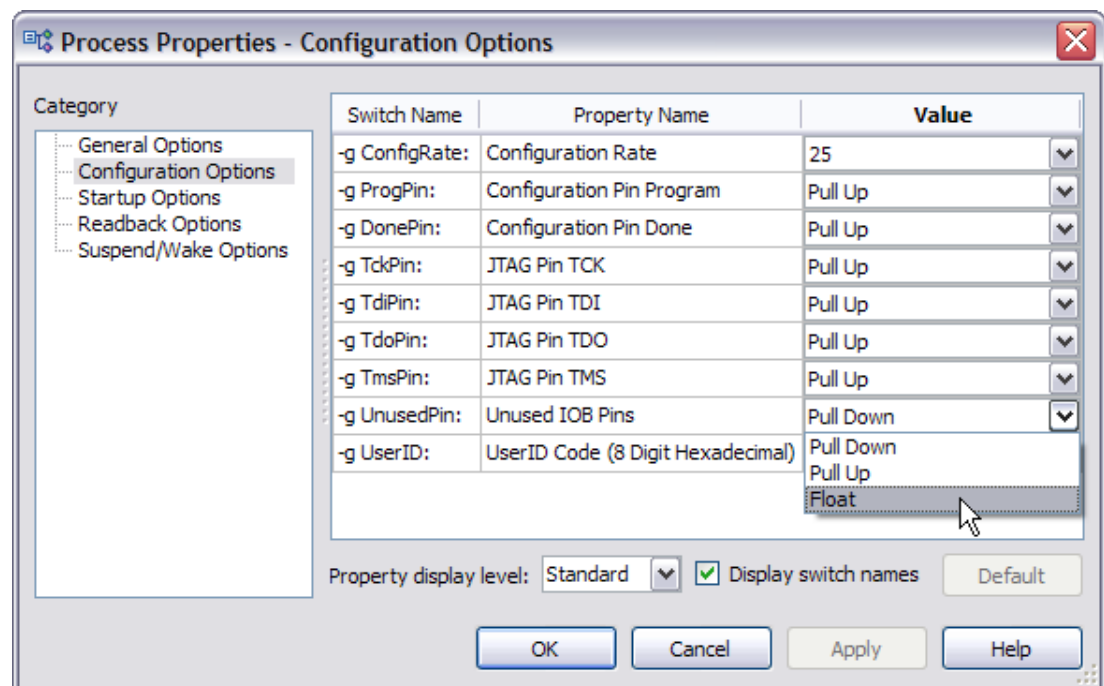
Now click on the **Startup Options** category and change the **FPGA Start-Up Clock** to **JTAG clock**.
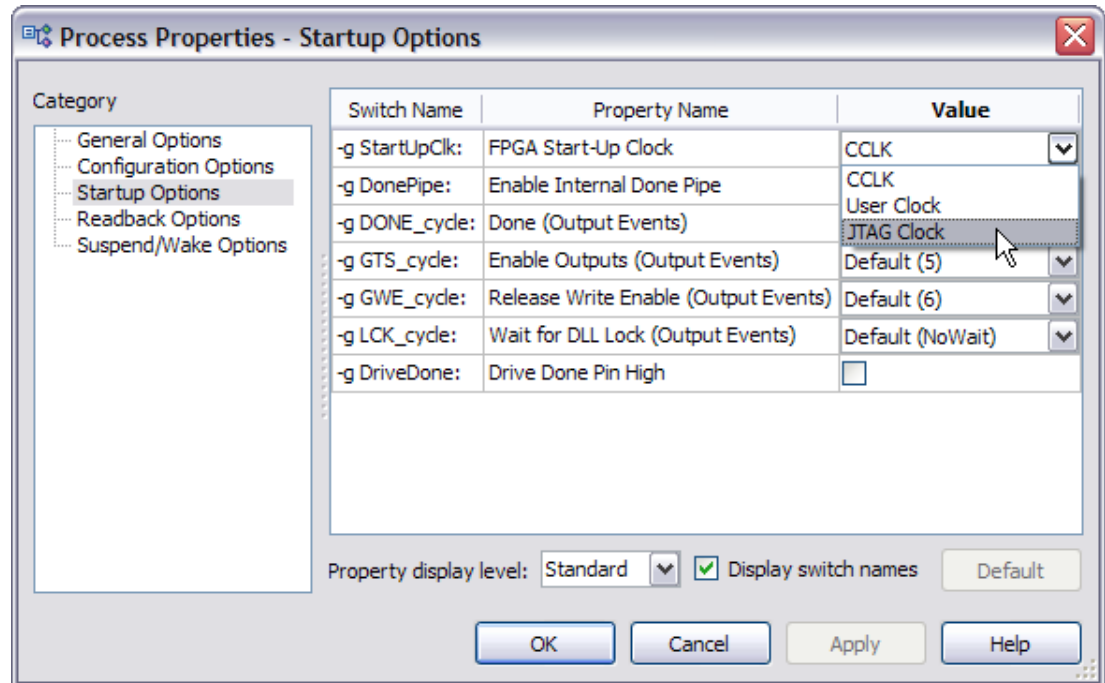
After setting these two options, click on the **OK** button and return to Navigator. Now you can double-click **Generate Programming File** to build the bitstream for the pc_blinker project.

Changing the XuLA Firmware

The XuLA board's factory-installed firmware does not support the commands needed to use the HostIoPckg modules. Therefore, you must upgrade the firmware using the **XuLA Firmware Update** command placed in the Windows **Start** menu when you installed the XSTOOLS software. If your XSTOOLs software does not have this command, you can install the latest software from the XESS website or use the firmware upgrade command in the FMW subdirectory for this project.

Once you've upgraded the XuLA firmware, you won't need to change it again. The upgrade provides all the features of the previous firmware while adding support for the HostIoPckg modules.

PC Software for Talking with the LED Blinker

The host-side software that communicates with the LED blinker is contained in a single file: pc_blinker_test.py. This is a Python script file, so you'll need a Python interpreter on your PC to run it. You can get a complete Python environment for free from Enthought or ActiveState. *(Please use the 32-bit version of Python to avoid run-time problems encountered with the 64-bit version.)*

After installing Python, you'll also need to install the XsTools package that supports communications between the XuLA board and the PC. That's easy to do using the command:

```
C:\> easy_install xstools
```

or:

```
C:\> pip install xstools
```

Here's what's in the pc_blinker_test.py file. It starts by importing all the functions and

classes needed to drive the inputs and read the outputs of a device-under-test (DUT) in the FPGA.

```
1    from xstools.xsdutio import * # Import funcs/classes for PC <=> FPGA link.
2
```

Then the program prints out a description of what it's trying to do.

```
3    print '''\n
4    ################################################################
5    # This program tests the interface between the host PC and the FPGA
6    # on the XuLA board that has been programmed to act as a blinker.
7    # You should see the state of the LED displayed on the screen
8    # flip back-and-forth between one and zero about once per second.
9    ################################################################
10   '''
```

Next, the program defines two identifiers: one for the USB port index of the XuLA board (which is usually 0 because your XuLA port is usually USB0), and another for the interface identifier of the blinker in the FPGA (as defined on line 56 of the pc_blinker.vhd file).

```
11   USB_ID = 0   # USB port index for the XuLA board connected to the host PC.
12   BLINKER_ID = 1   # This is the identifier for the blinker in the FPGA.
13
```

Now create an object that lets the host talk to the blinker in the FPGA. The USB and blinker identifiers are needed to initialize the object. You also pass in two arrays: one which defines the sizes of the input fields to the blinker, and another that defines the sizes of the output fields. (As I said before, while there are no inputs to the blinker, we still needed to define a single dummy input so we also need to recognize that dummy input when we create the blinker object here.)

```
14   # Create a blinker intfc obj with one 1-bit input and one 1-bit output.
15   blinker = XsDut(USB_ID, BLINKER_ID, [1], [1])
16
```

Finally, go into an infinite loop where the state of the counter output that drives the LED is monitored using the Read() method of the blinker object. Then print the value of the counter bit along with a RETURN character to go back to the start of the line. When this program is run, this will give the effect of the LED value flipping back-and-forth between one and zero.
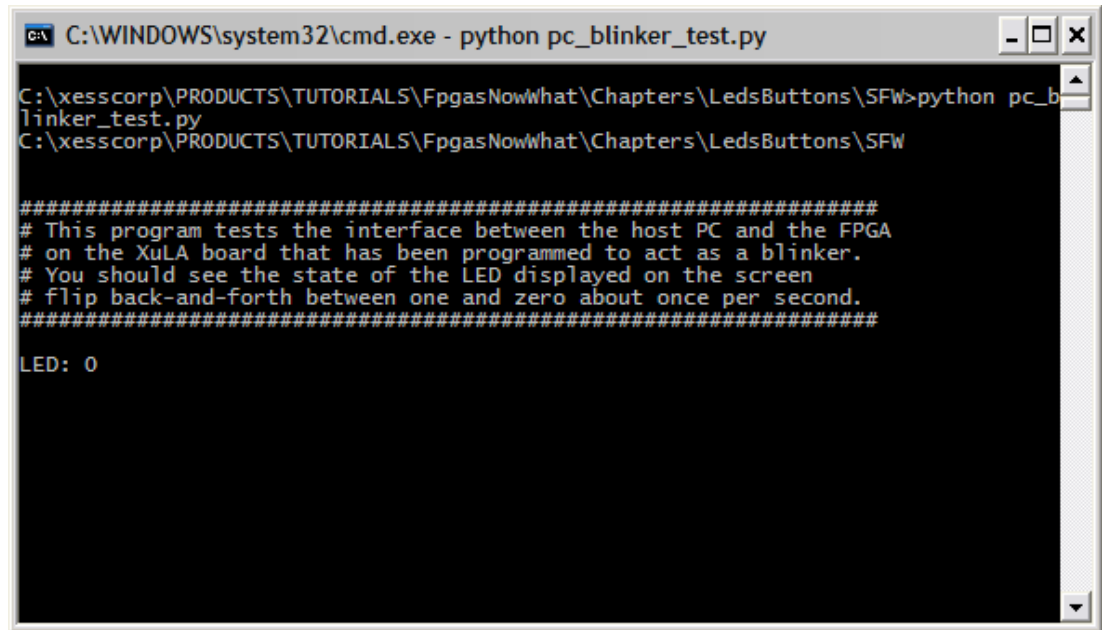
```
17   while True: # Do this forever...
18       led = blinker.Read() # Read the current state of the LED.
19       print 'LED: %1d\r' % led.unsigned, # Print the LED state and return.
```

Putting It All Together

At this point, I assume you have the pc_blinker ISE project compiled and the bitstream is ready to go, the firmware in your XuLA board has been upgraded to support the HostIoPckg modules, and you have Python installed and the pc_blinker_test.py file available. Now all you have to do is this:

1. Download the pc_blinker.bit file to the FPGA on the XuLA board using GXSLOAD.

2. Run the Python program in a command window as follows:
    ```
    C:\SFW> python pc_blinker_test.py
    ```

Then you should see the following text appear in the command window:

What I can't show here is the digit flipping back-and-forth between one and zero. But you can see it in this exciting video. Notice how the LED and the digit are synchronized with each other, showing that the PC is able to accurately display what the FPGA is doing.

Testing a Subtractor

I know you've really grown to love the LED blinker over the past few chapters, but it's time to move on to something new: a subtractor. The subtractor circuit in this section takes two eight-bit inputs (a minuend and a subtrahend) and calculates their difference.

Here's a picture of how the subtractor is hooked up to the JTAG entry point and the shift-register. It's not too different from what we did with the LED blinker except here there are two, byte-wide inputs and a byte-wide output instead of just a single output bit. That means we have more data to shift in and out of the circuit for each test, but that's not a big complication. The HostIoPckg modules will handle all that transparently.
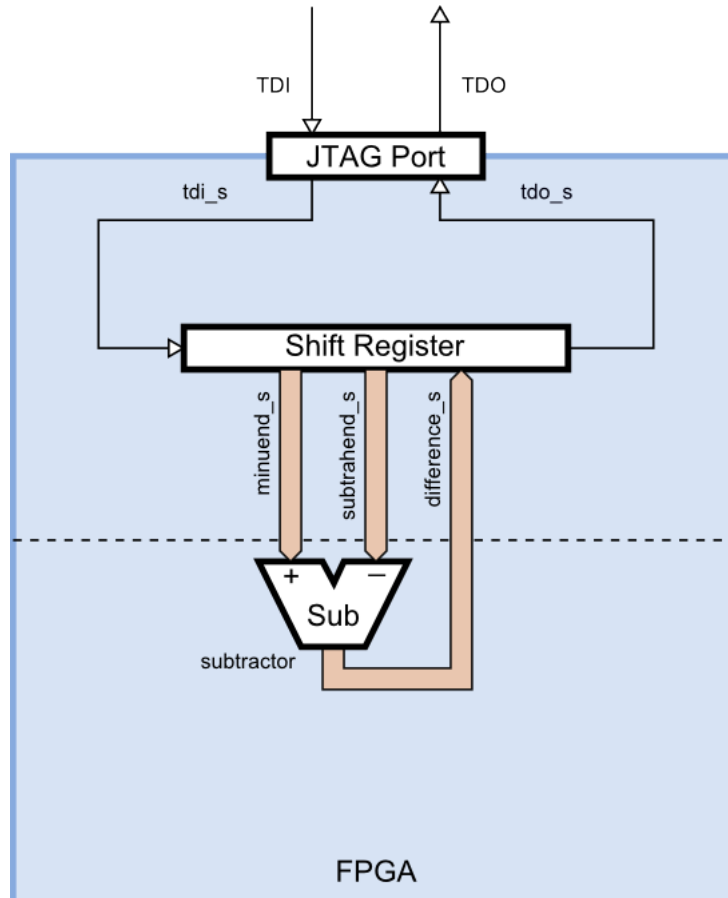
We use the same VHDL files as in the previous section, except we replace the pc_blinker.vhd file with the pc_subtractor.vhd file that contains the following code:

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.all;
3   use IEEE.STD_LOGIC_UNSIGNED.all;
4   use work.HostIoPckg.all;  -- Package for PC <=> FPGA communications.
5
```

We're not connecting the subtractor to the outside world, so there are no inputs or outputs declared in the entity section.

```
6   entity pc_subtractor is
7   end entity;
8
```

On line 11, the signal from the shift-register module to the subtractor is declared to be

sixteen bits wide in order to hold both byte-wide operands. On line 12, another byte-wide
signal is declared to carry the result from the subtractor to the shift register. Aliases to
these signals are declared on lines 13 – 15 so we can use more understandable names in
the rest of the code.

```
 9  architecture Behavioral of pc_subtractor is
10    -- Connections between the shift-register module and the subtractor.
11    signal toSub_s     : std_logic_vector(15 downto 0); -- From PC to subtrctr.
12    signal fromSub_s   : std_logic_vector(7 downto 0);  -- From subtrctr to PC.
13    alias minuend_s    is toSub_s(7 downto 0);  -- Subtrctr's 1st operand.
14    alias subtrahend_s is toSub_s(15 downto 8); -- Subtrctr's 2nd oprnd.
15    alias difference_s is fromSub_s;            -- Subtractor's output.
16    -- Connections between JTAG entry point and the shift-register module.
17    signal inShiftDr_s : std_logic; -- True when bits shift btwn PC & FPGA.
18    signal drck_s      : std_logic; -- Bit shift clock.
19    signal tdi_s       : std_logic; -- Bits from host PC to the subtractor.
20    signal tdo_s       : std_logic; -- Bits from subtractor to the host PC.
21  begin
22
```

The actual subtraction operation is defined on line 27.

```
23    -------------------------------------------------------------------------
24    -- Application circuitry: the subtractor.
25    -------------------------------------------------------------------------
26
27    difference_s <= minuend_s - subtrahend_s;
28
```

The JTAG entry point is the same as in the previous example.

```
29    -------------------------------------------------------------------------
30    -- JTAG entry point.
31    -------------------------------------------------------------------------
32
33    -- Main entry point for the JTAG signals between the PC and the FPGA.
34    UBscanToHostIo : BscanToHostIo
35      port map (
36        inShiftDr_o => inShiftDr_s,
37        drck_o      => drck_s,
38        tdi_o       => tdi_s,
39        tdo_i       => tdo_s
40        );
41
```

On line 48, the shift-register module is assigned an identifier of '4' so the PC can select it
for access. The signals from the shift-register to the minuend and subtrahend operands of
the subtractor are connected on line 56. The difference output by the subtractor is
connected to the shift-register on line 57.

```
42    -------------------------------------------------------------------------
43    -- Shift-register.
44    -------------------------------------------------------------------------
45
46    -- Shift-register module between subtractor and JTAG entry point.
47    UHostIoToSubtractor : HostIoToDut
48      generic map (ID_G => "00000100")    -- The identifier used by the PC.
49      port map (
50        -- Connections to the BscanToHostIo JTAG entry-point module.
51        inShiftDr_i     => inShiftDr_s,
52        drck_i          => drck_s,
53        tdi_i           => tdi_s,
54        tdo_o           => tdo_s,
55        -- Connections to the subtractor.
56        vectorToDut_o   => toSub_s, -- From PC to sbtrctr subtrahend & minuend.
57        vectorFromDut_i => fromSub_s  -- From subtractor difference to PC.
```

```
58            );
59
60    end architecture;
```

Here's the design hierarchy for the pc_subtractor project:
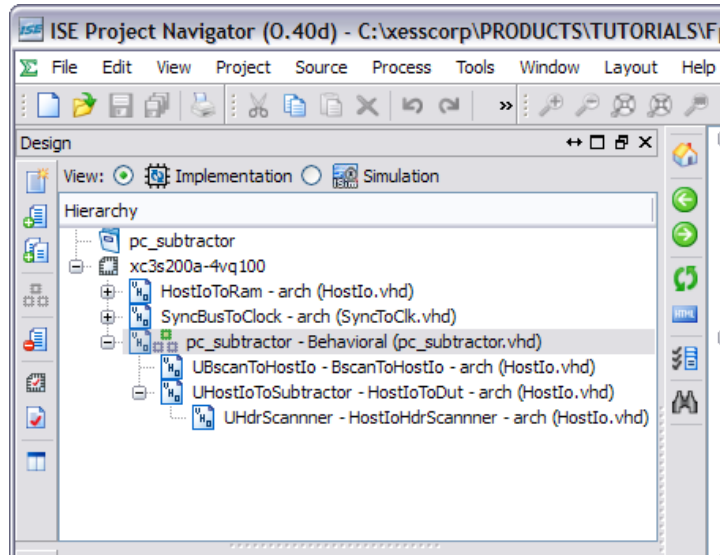
At this point, you can compile the design to create the bitstream. (As in the previous example, remember to set the pc_subtractor as the top module and to assign the JTAG clock as the start-up clock and place the unused FPGA pins in a high-impedance state.)

With the VHDL completed and compiled, it's time to move on to the microcontroller firmware. You should have already updated that to handle the HostIoPckg modules in the previous example, so there's no need to do it again.

Finally, a new host-side program is needed to drive the subtractor's inputs and monitor its output. Here's the Python source code from the pc_subtractor_test.py file. As in the previous example, it starts by importing all the functions and classes in the XsTools Python package. It also imports a random number library that is used to generate random inputs to the subtractor.

```
1     from xstools.xsdutio import * # Import funcs/classes for PC <=> FPGA link.
2     from random import *  # Import some random number generator routines.
3
4     print '''
5     ################################################################
6     # This program tests the interface between the host PC and the FPGA
7     # on the XuLA board that has been programmed to act as a subtractor.
8     ################################################################
9     '''
10
```

On line 12, the program defines an identifier to match that of the subtractor in the FPGA (as defined on line 48 of the pc_subtractor.vhd file).

```
11    USB_ID = 0  # USB port index for the XuLA board connected to the host PC.
12    SUBTRACTOR_ID = 4  # This is the identifier for the subtractor in the FPGA.
13
```

Now create an object that lets the host talk to the subtractor in the FPGA (line 15). The USB and subtractor identifiers are needed to initialize the object. You also pass in two arrays: one which declares the inputs to the subtractor to be two byte-wide fields, and another that declares the output as a single byte-wide field.

```
14   # Create a subtractor intfc obj with two 8-bit inputs and one 8-bit output.
15   subtractor = XsDut(USB_ID, SUBTRACTOR_ID, [8, 8], [8])
16
```

Finally, iterate through 100 test cases (line 18) where a random minuend and subtrahend are generated (lines 19 and 20) and sent to the subtractor using its Exec() method (line 21) which also returns their difference. Print the operands and the resulting difference (line 22). Then, do the same subtraction operation in Python and compare it to the previous result returned by the subtractor in the FPGA (line 23). Report whether the results agree or not (lines 24 – 26).

```
17   # Test the subtractor by iterating through some random inputs.
18   for i in range(0, 100):
19       minuend    = randint(0, 127)  # Get a random, positive byte...
20       subtrahend = randint(0, 127)  # And subtract this random byte from it.
21       diff = subtractor.Exec(minuend, subtrahend) # Use the subtractor in FPGA.
22       print '%3d - %3d = %4d' % (minuend, subtrahend, diff.int),
23       if diff.int == minuend - subtrahend: # Compare Python result to FPGA's.
24           print '==> CORRECT!'  # Print this if the differences match.
25       else:
26           print '==> ERROR!!!'  # Oops! Something's wrong with the subtractor.
```

Now that everything is ready, you can test the subtractor like this:

1. Download the pc_subtractor.bit file to the FPGA on the XuLA board using GXSLOAD.

2. Run the Python program in a command window as follows:
    `C:\SFW> python pc_subtractor_test.py`

Then you should see something similar to the following text appear in the command window:

```
################################################################
# This program tests the interface between the host PC and the FPGA
# on the XuLA board that has been programmed to act as a subtractor.
################################################################


 68 -  60 =    8 ==> CORRECT!
107 - 114 =   -7 ==> CORRECT!
 90 -  47 =   43 ==> CORRECT!
105 -  91 =   14 ==> CORRECT!
 71 - 114 =  -43 ==> CORRECT!
 60 - 115 =  -55 ==> CORRECT!
 19 -  80 =  -61 ==> CORRECT!
...
```

Two at Once!

You've seen how to use the HostIoPckg modules to test individual designs, but how would you control and monitor a design built from several submodules? You could just build a larger shift-register and hook everything to that of course, but then you would have to worry about screwing-up the inputs to one submodule while you were twiddling the bits of another.

A better way is to give each submodule its own shift-register and connect them all through the JTAG port. But how do you select which shift-register is connected to the JTAG port? Well, that's where the shift-register identifiers come in. Before the host-side PC accesses any shift-register, it sends an eight-bit identifier. The shift-register that has that identifier will enable itself and accept bits shifted in from the JTAG port, and it will unlock its TDO output so its contents can be shifted out of the JTAG port. All the other shift-registers will ignore the bits coming from the JTAG port and will force their TDO outputs to zero. (I told you these weren't *just* shift-registers.) Then, the TDO outputs of all the shift-registers are logically OR'ed together and sent to the JTAG port. Since all the inactive shift-registers hold their TDO outputs at zero, only the TDO output of the selected shift-register actually gets to the JTAG port and out to the host PC. An example of how the JTAG port and shift-registers are connected is shown below, using the LED blinker and the subtractor as the submodules.
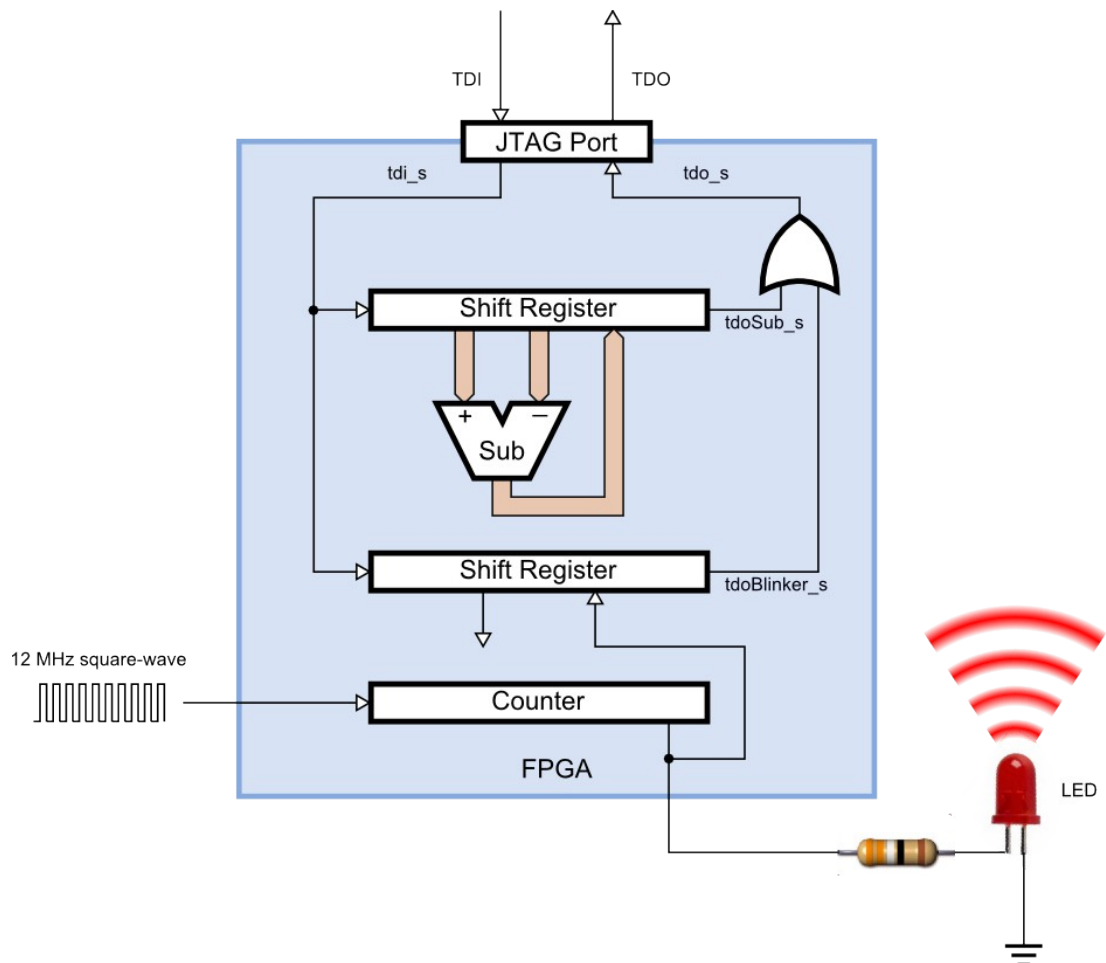
The VHDL code for the circuit shown above is stored in the pc_blink_sub.vhd file. It basically looks like the VHDL files for the two previous examples were just smashed into the same file.

```
1    library IEEE;
2    use IEEE.STD_LOGIC_1164.all;
3    use IEEE.STD_LOGIC_UNSIGNED.all;
4    use work.HostIoPckg.all;  -- Package for PC <=> FPGA communications.
5
6    entity pc_blink_sub is
7      port (clk_i     : in  std_logic;
8            blinker_o : out std_logic);
```

```
 9    end entity;
10
```

The only change in the signal declarations for the merged design is that two individual TDO signals are created, one for the LED blinker (line 27) and another for the subtractor (line 28).

```
11    architecture Behavioral of pc_blink_sub is
12      signal cnt_r         : std_logic_vector(22 downto 0) := (others => '0');
13      -- Connections between first shift-register module and the blinker.
14      signal toBlinker_s   : std_logic_vector(0 downto 0);  -- From PC to blnkr.
15      signal fromBlinker_s : std_logic_vector(0 downto 0);  -- From blnkr to PC.
16      -- Connections between second shift-register module and the subtractor.
17      signal toSub_s   : std_logic_vector(15 downto 0);  -- From PC to subtrctr.
18      signal fromSub_s : std_logic_vector(7 downto 0);   -- From subtrctr to PC.
19      alias minuend_s is toSub_s(7 downto 0);      -- Subtrctr's 1st operand.
20      alias subtrahend_s is toSub_s(15 downto 8); -- Subtrctr's 2nd oprnd.
21      alias difference_s is fromSub_s;       -- Subtractor's output.
22      -- Connections between JTAG entry point and the shift-register modules.
23      signal inShiftDr_s   : std_logic; -- True when bits shift btwn PC & FPGA.
24      signal drck_s        : std_logic; -- Bit shift clock.
25      signal tdi_s         : std_logic; -- Bits from host PC to blnkr/subtrctr.
26      signal tdo_s         : std_logic; -- Bits from blnkr/subtrctr to host PC.
27      signal tdoBlinker_s  : std_logic; -- Bits from the blinker to the host PC.
28      signal tdoSub_s      : std_logic; -- Bits from the sbtrctr to the host PC.
29    begin
30
```

The VHDL for the LED blinker and subtractor stay the same; they're just merged into the same section.

```
31      ----------------------------------------------------------------------------
32      -- Application circuitry
33      ----------------------------------------------------------------------------
34
35      -- This counter divides the input clock.
36      process(clk_i) is
37      begin
38        if rising_edge(clk_i) then
39          cnt_r <= cnt_r + 1;
40        end if;
41      end process;
42
43      blinker_o <= cnt_r(22); -- This counter bit blinks the LED.
44
45      -- This is the subtractor.
46      difference_s <= minuend_s - subtrahend_s;
47
```

The JTAG entry-point doesn't change at all. Only one entry-point is needed no matter how many shift-register modules you use.

```
48      ----------------------------------------------------------------------------
49      -- JTAG entry point.
50      ----------------------------------------------------------------------------
51
52      -- Main entry point for the JTAG signals between the PC and the FPGA.
53      UBscanToHostIo : BscanToHostIo
54        port map (
55          inShiftDr_o => inShiftDr_s,
56          drck_o      => drck_s,
57          tdi_o       => tdi_s,
58          tdo_i       => tdo_s
59          );
60
```

This is where the TDO outputs of the LED blinker and subtractor are OR'ed together. The result is passed into the JTAG entry-point above.

```
61    -- OR the bits from both shift-registers and send them back to the PC.
62    -- (Non-selected modules pull their TDO outputs low, so only bits from
63    -- the active module are transferred.)
64    tdo_s <= tdoBlinker_s or tdoSub_s;
65
```

Next, two shift-registers are created. The only change from the previous examples is that the TDO outputs of the shift-registers are connected to the individual TDO signals (lines 78 and 94) that are OR'ed together on line 64.

```
66     ---------------------------------------------------------------------------
67     -- Shift-registers.
68     ---------------------------------------------------------------------------
69
70     -- Shift-register module between blinker and JTAG entry point.
71     UHostIoToBlinker : HostIoToDut
72       generic map (ID_G => "00000001")    -- The identifier used by the PC.
73       port map (
74         -- Connections to the BscanToHostIo JTAG entry-point module.
75         inShiftDr_i     => inShiftDr_s,
76         drck_i          => drck_s,
77         tdi_i           => tdi_s,
78         tdo_o           => tdoBlinker_s, -- Serial bits from blinker output.
79         -- Connections to the blinker.
80         vectorToDut_o   => toBlinker_s,   -- From PC to blinker (dummy sig).
81         vectorFromDut_i => fromBlinker_s  -- From blinker to PC.
82         );
83
84     fromBlinker_s <= cnt_r(22 downto 22);  -- Blinker output to shift reg.
85
86     -- Shift-register module between subtractor and JTAG entry point.
87     UHostIoToSubtractor : HostIoToDut
88       generic map (ID_G => "00000100")    -- The identifier used by the PC.
89       port map (
90         -- Connections to the BscanToHostIo JTAG entry-point module.
91         inShiftDr_i     => inShiftDr_s,
92         drck_i          => drck_s,
93         tdi_i           => tdi_s,
94         tdo_o           => tdoSub_s,-- Serial bits from subtractor result.
95         -- Connections to the subtractor.
96         vectorToDut_o   => toSub_s, -- From PC to sbtrctr subtrahend & minuend.
97         vectorFromDut_i => fromSub_s -- From subtractor difference to PC.
98         );
99
100  end architecture;
```

That's all there is to it! You can compile it and download the bitstream into the XuLA board.

You don't need a new Python test program – the ones from the previous examples will work just fine. The shift-register module identifier coded into each program will select the appropriate portion of the circuit for testing. You can run them in any order and they will produce the correct result. (Just don't try to run them *simultaneously* since they will collide when trying to use the same USB link.)

So What?

At this point you might be saying: "I'm adding tons of VHDL code and writing Python programs that access DLLs that access firmware that talks to the FPGA just so I can see a blinking LED. My head hurts! All I wanted was some LEDs and some buttons! Is that too much to ask?"

I can understand those feelings, but I'll restate the advantages you get from the techniques shown in this chapter:

It's free. LEDs and buttons cost money and board space, and those things would add cost to the XuLA board. While cheaper isn't always better, it is always cheaper.

It's flexible. You can use it with multiple sub-circuits each having hundreds of inputs and outputs and perform tests that are limited only by your ability to write programs for them.

It' easy to use. OK, stop laughing! Admittedly, it's not as easy as using a single LED with the blinker project, but most of your circuits aren't going to be that simple. Even the byte-wide subtractor would require sixteen switches for the inputs and eight LEDs for the output. Would you want to build that for just a single test, or try flipping the switches to test all 65,536 different input combinations?

It takes no space. You need space to attach a bunch of switches and LEDs and even more headroom to manipulate and observe them. With these techniques, you can fit the XuLA board into some pretty tight spaces and still have controllability and observability. (It does take up space *inside* the FPGA, using about 3% of the LUTs and flip-flops in an XC3S200A.)

It uses no FPGA pins. All it needs is the JTAG port and those pins can't even by used by the application circuitry in the FPGA. Without these techniques, testing a simple 16-bit wide subtractor would require 48 I/O pins, and that's impractical with the XuLA board.

It's automated. Once you write your test program, all you have to do is run it whenever you want to perform that test. So it's faster to run the test, making it more likely you'll do so because it takes so little effort. (This is the same advantage you get from writing simulation testbenches for your designs.)

It's removable. Once you're done testing your circuit, you can remove the HostIoPckg modules and reclaim any space they consumed in the FPGA.

You'll be seeing a lot more of this technique in the coming chapters, so if I haven't convinced you, then you might as well stop reading now. For example, in a future chapter I'll use a variation on this technique to read and write RAMs and registers in the FPGA. Try *that* with LEDs and buttons!

C.7 "RAMs! Now What!?"

There are two metrics that almost everybody uses to evaluate an FPGA: how much logic can it hold, and how much memory does it contain. Up to now, I've concentrated pretty much on logic design and ignored memory. That will change in this chapter: I am going to talk about memory. A lot.

There are three types of memory you can use with your FPGA:

Distributed RAM: This is internal memory built from the FPGA's LUTs as I discussed briefly back in Chapter 1.

Block RAM: This is also internal memory, but built using one or more larger, special-purpose block RAM (BRAM) components interspersed throughout the FPGA fabric.

External RAM: Just like the name says, this is memory housed in a RAM chip outside the FPGA that's connected to the FPGA's pins.

Along with the three types of memory, there are three ways to include RAM in your VHDL design:

Inference: You can write your VHDL code in such a way that the logic synthesizer can infer your desire to use memory and will instantiate and connect the correct FPGA components for you.

Core Generator: You can use the ISE CORE Generator tool to build the memory you want from the FPGA components and then instantiate this higher-level module in your VHDL.

Instantiation: You can plunk down RAM modules and/or external memory interfaces in your VHDL and then wire them together to build the memory you want.

In the sections that follow, I'll use each of these techniques to implement various types of memory.

Inferring RAM

Inferring Distributed RAM

To show how to build RAM by inference, I'm going to build a simple application (located in the DRamSPInf project directory) that will store some numbers into a *single-port distributed RAM* and then read each RAM address, multiply the address with whatever number is stored there, and then sum all these products together. (I know this is a stupid example, but it shows the basics: how to write to RAM and how to read the data back. Plus, you get to use a hardware multiplier! Does it get any better than that!?)

The beginning section of the DRamSPInf.vhd file will look pretty familiar by now. In this example, I'm using the NUMERIC_STD library to support the arithmetic operations I'll need.

```
 101    --***********************************************************************
```

```
102   -- Distributed RAM, single-port, inferred.
103   --*********************************************************************
104
105   library IEEE;
106   use IEEE.STD_LOGIC_1164.all;
107   use IEEE.NUMERIC_STD.all;
108
```

The entity declaration for my application follows. The I/O is pretty simple: just an input clock to drive the logic and a 32-bit output for the final sum-of-products.

```
109   entity DRamSPInf is
110     port (
111       clk_i : in  std_logic;
112       sum_o : out std_logic_vector(31 downto 0) := (others => '0')
113       );
114   end entity;
115
```

The architecture section starts with some constants to specify the number of RAM locations (line 19) and the number of bits in each word (line 20). Lines 21 and 22 specify the beginning and ending addresses for the summation process.

```
116   architecture Behavioral of DRamSPInf is
117     constant NO          : std_logic := '0';
118     constant YES         : std_logic := '1';
119     constant RAM_SIZE_C  : natural   := 16;  -- Number of words in RAM.
120     constant RAM_WIDTH_C : natural   := 8;   -- Width of RAM words.
121     constant MIN_ADDR_C  : natural   := 1;   -- Process RAM from this address
122     constant MAX_ADDR_C  : natural   := 5;   -- ... to this address.
```

Line 23 defines a subtype of the NUMERIC_STD unsigned bit vector which has the same length as the RAM word. This subtype is used to define a type of array with the same number of elements as the RAM (line 23). Then the actual RAM is declared on line 25.

```
123     subtype RamWord_t is unsigned(RAM_WIDTH_C-1 downto 0); -- RAM word type.
124     type Ram_t is array (0 to RAM_SIZE_C-1) of RamWord_t;  -- RAM word array.
125     signal ram_r        : Ram_t;            -- RAM declaration.
```

Lines 26-29 declare the RAM's write-enable control line, the address bus, and the input and output data buses. The address bus is sized such that it can hold any possible RAM address, and the input and output data bus types are the same as that of the RAM words for a similar reason. Finally, the register that holds the summation is sized to hold the product of the largest possible address and data word (line 30). That completes the declaration section of the architecture.

```
126     signal wr_s         : std_logic;      -- Write-enable control.
127     signal addr_r       : natural range 0 to RAM_SIZE_C-1;  -- RAM address.
128     signal dataToRam_r  : RamWord_t;      -- Data to write to RAM.
129     signal dataFromRam_s : RamWord_t;     -- Data read from RAM.
130     signal sum_r        : natural range 0 to RAM_SIZE_C*(2**RAM_WIDTH_C)-1;
131   begin
132
```

The first thing I do in the body of the architecture is create the VHDL code that will infer the distributed RAM. But, can I write the VHDL any way I want and still have the synthesizer figure out that I'm trying to build a RAM? Probably not! That's why the ISE software provides a set of templates to guide me. Simply click the **Edit ⇨ Language Templates…** menu item and a **Language Templates** tab will appear. Then expand the tree structure until you reach the **Single Port, Async Read** template for the distributed RAM:
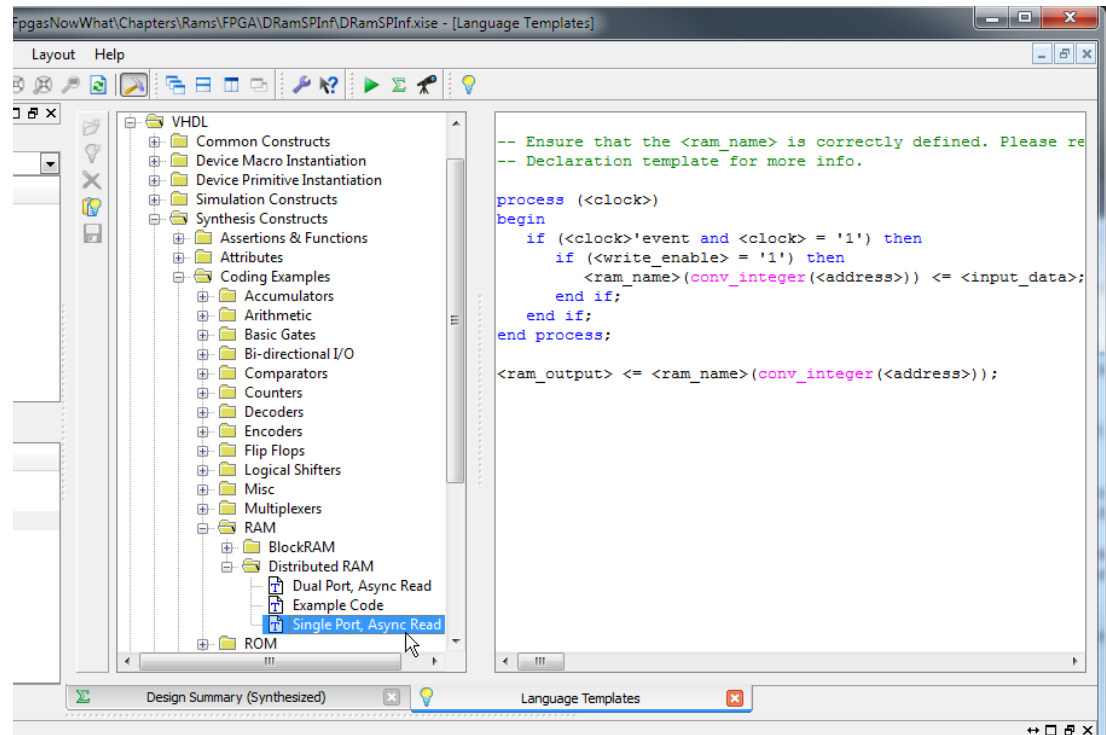
I cut-and-pasted the template code into lines 36-46 and replaced the template names with the actual signal names for my example. If a rising clock edge occurs (line 39) when the write-enable line is high (line 40), then whatever is on the input data bus is written into the RAM at the current address (line 41). So this is a *synchronous-write* RAM.

Meanwhile, the RAM output bus shows whatever data is stored at the current address (line 46). This is an *asynchronous-read* RAM since the output value will change whenever the address changes, regardless of the clock.

```
133     --*****************************************************************
134     -- RAM is inferred from this process.
135     --*****************************************************************
136     Ram_p : process (clk_i)
137     begin
138       -- Write to the RAM at the given address if the write-enable is high.
139       if rising_edge(clk_i) then
140         if wr_s = YES then
141           ram_r(addr_r) <= dataToRam_r;
142         end if;
143       end if;
144     end process;
145     -- Continually read data from whatever RAM address is present.
146     dataFromRam_s <= ram_r(addr_r);
147
```

The finite-state machine (FSM) that performs the summation of the RAM contents is shown on lines 48-88. The FSM starts off in the INIT state (line 54) and performs an operation and changes state on each rising edge of the input clock (line 56). In the INIT state (lines 58-62), the FSM sets the address and input data buses to the RAM at their initial values (0 and 1, respectively) and sets the write-control signal high to enable writing. Then it moves to the next state.

```
148    --*********************************************************************
149    -- State machine that initializes RAM and then reads RAM to compute
150    -- the sum of products of the RAM address and data.
151    --*********************************************************************
152    Fsm_p : process (clk_i)
153      type state_t is (INIT, WRITE_DATA, READ_AND_SUM_DATA, DONE);
154      variable state_v : state_t := INIT;    -- Start off in init state.
155    begin
156      if rising_edge(clk_i) then
157        case state_v is
158          when INIT =>
159            wr_s        <= YES;          -- Enable writing of RAM.
160            addr_r      <= MIN_ADDR_C;   -- Start writing data at this address.
161            dataToRam_r <= TO_UNSIGNED(1, RAM_WIDTH_C); -- Initial write value.
162            state_v     := WRITE_DATA;   -- Go to next state.
```

In the WRITE_DATA state, the FSM increments the address (line 65) and adds three to the data value to write to RAM (line 66) as long as the address has not reached its maximum (line 64). When the maximum address is reached, the write-enable is lowered (line 68), the address is reset to its beginning value (line 69), and the summation register is initialized to zero (line 70). Then the FSM moves to the next state (line 71). At this point, the RAM locations 1 through 5 should be loaded with the values 1, 4, 7, 10 and 13.

```
163          when WRITE_DATA =>
164            if addr_r < MAX_ADDR_C then  -- If haven't reach final address ...
165              addr_r      <= addr_r + 1; -- go to next address ...
166              dataToRam_r <= dataToRam_r + 3; -- and write this value.
167            else  -- Else, the final address has been written...
168              wr_s    <= NO;             -- so turn off writing, ...
169              addr_r  <= MIN_ADDR_C;     -- go back to the start, ...
170              sum_r   <= 0;              -- clear the sum-of-products, ...
171              state_v := READ_AND_SUM_DATA;  -- and go to next state.
172            end if;
```

In the READ_AND_SUM_DATA state, the FSM gets the number read from the RAM location, multiplies it by the location's address and then adds it to the summation register (line 77). It also increments the address to point to the next location (line 78). This is repeated until the maximum address is reached (line 79). At that point, the summation register has the complete sum-of-products, so the FSM moves into the DONE state (line 80) where it remains forever.

```
173          when READ_AND_SUM_DATA =>
174            if addr_r <= MAX_ADDR_C then  -- If haven't reached final address
175              -- add product of RAM address and data read
176              -- from RAM to the summation ...
177              sum_r  <= sum_r + TO_INTEGER(dataFromRam_s * addr_r);
178              addr_r <= addr_r + 1;        -- and go to the next address.
179            else  -- Else, the final address has been read ...
180              state_v := DONE;             -- so go to the next state.
181            end if;
182          when DONE =>                     -- Summation complete ...
183            null;                          -- so wait here and do nothing.
184          when others =>                   -- Erroneous state ...
```

```
185              state_v := INIT;              -- so re-run the entire process.
186          end case;
187       end if;
188    end process;
189
```

The next few lines just dump the value in the summation register onto the output bus of this module.

```
190    -- Output the sum of the RAM address-data products.
191    sum_o <= std_logic_vector(TO_UNSIGNED(sum_r, sum_o'length));
192
193  end architecture;
```

At this point, I can run the synthesizer on this design and see how the RAM is implemented by checking the synthesis report.
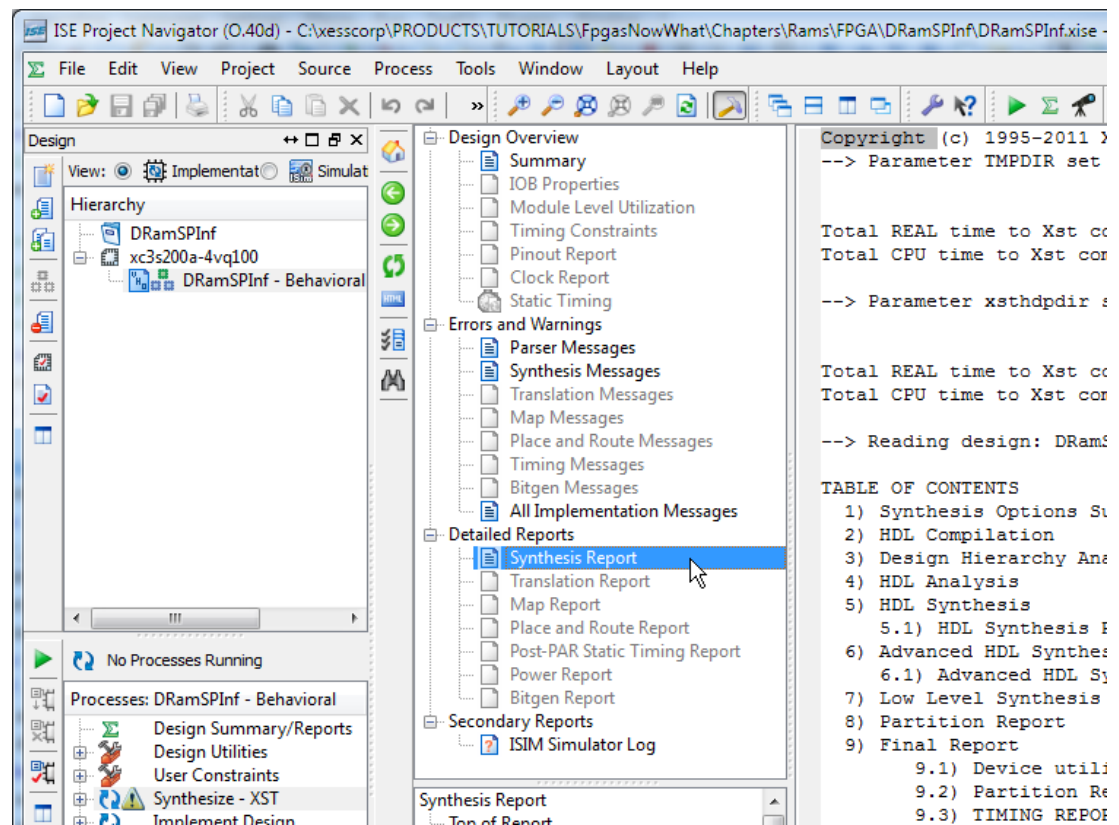
Scrolling through the report, I see the following:

```
 1   INFO:Xst:3231 - The small RAM <Mram_ram_r> will be implemented on LUTs in
     order to maximize performance and save block RAM resources. If you want to
     force its implementation on block, use option/constraint ram_style.
 2       ------------------------------------------------------------------
 3       | ram_type           | Distributed                     |       |
 4       ------------------------------------------------------------------
 5       | Port A                                                         |
 6       |     aspect ratio   | 16-word x 8-bit                 |       |
 7       |     clkA           | connected to signal <clk_i>     | rise  |
 8       |     weA            | connected to signal <wr_s>      | high  |
 9       |     addrA          | connected to signal <addr_r>    |       |
10       |     diA            | connected to signal <dataToRam_r>  |    |
11       |     doA            | connected to signal <dataFromRam_s> |   |
12       ------------------------------------------------------------------
```

Line 1 indicates the RAM was implemented using LUTs, and this is further confirmed on line 3 where it shows a distributed RAM was used. The size of the RAM (line 6) matches the number of locations and word-size that I specified in the VHDL. The connections to the clock, write-enable, address bus, and input/output data buses also match with the VHDL.

Now it's time to see if the RAM works. I added a simple VHDL test-bench file (DramSPInf_tb.vhd) to the project that just applies a clock to the FSM and RAM. Then I ran the simulator.
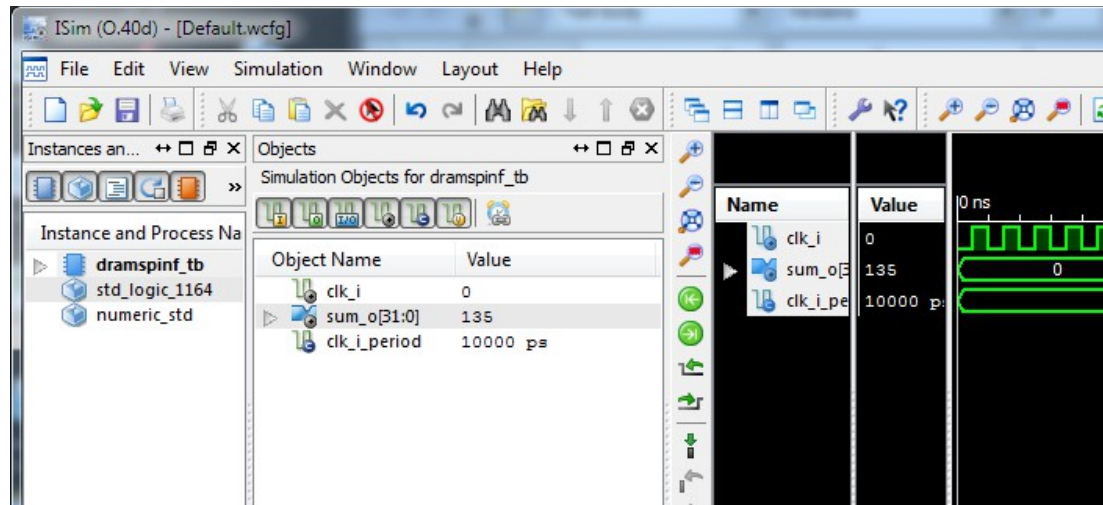
I want to see more than just the clock and the output summation, so I expand the design instances a bit and select the FSM portion.
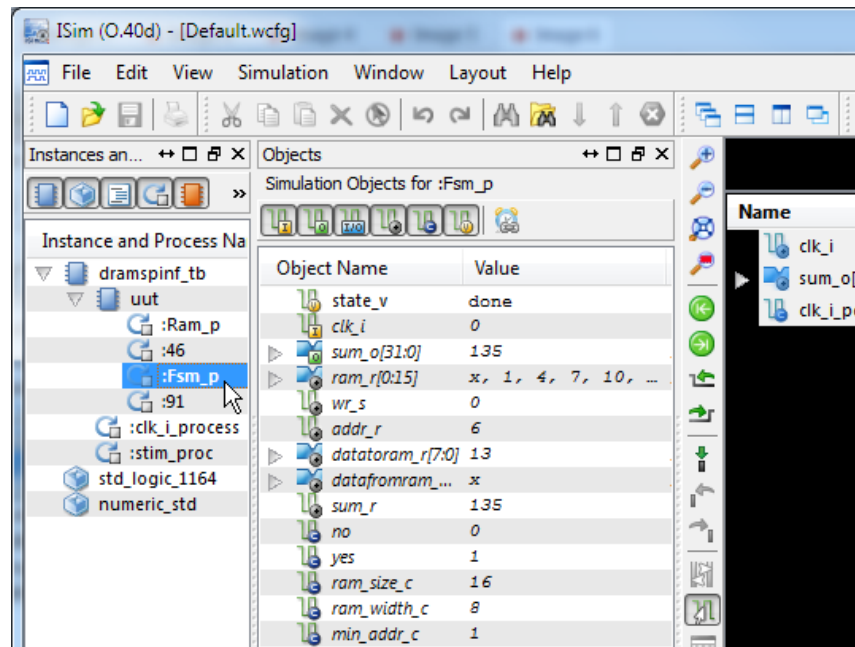
Within the list of simulation objects for the FSM, I select the RAM's write-enable, address and input/output data buses and drag them into the pane of observable signals.
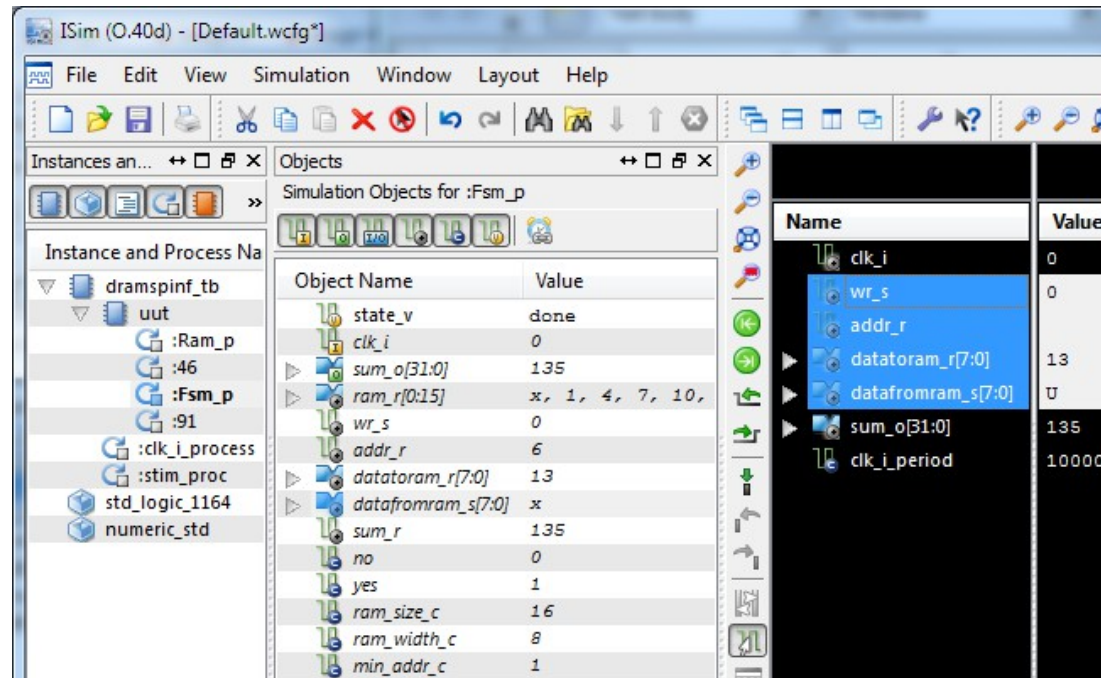
Now I can re-run the simulation and see what happens. On the first rising clock edge, the address (addr_r) and data (datatoram_r) are initialized to 1, and the write-enable is raised. Over the next five rising clock edges, the values 1, 4, 7, 10, and 13 are written to addresses 1 through 5. The write-enable goes low after the last write and then the values are read back from RAM (datafromram_s). Multiplying each value by its address gives 1x1 + 2x4 + 3x7 +4x10 + 5x13 = 135 which matches the final value on the sum output.
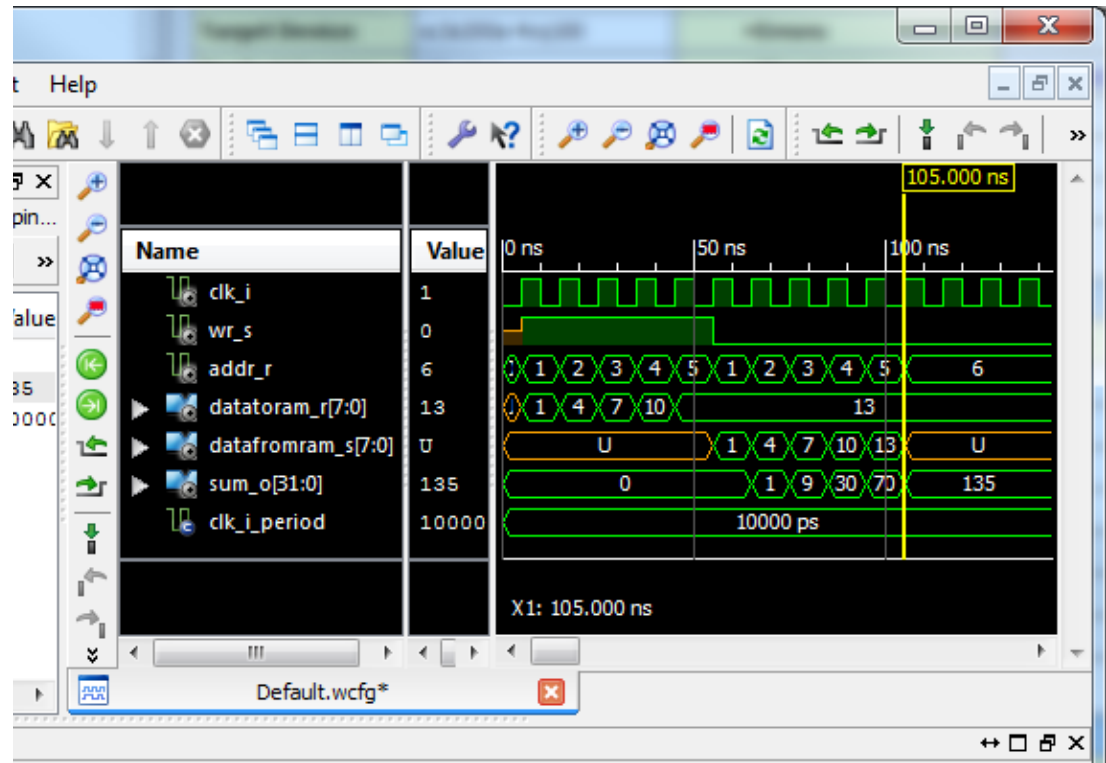
I can also view the contents of the RAM by selecting the **Memory** tab and double-clicking the ram_r memory. This shows the right data is stored at the right addresses.
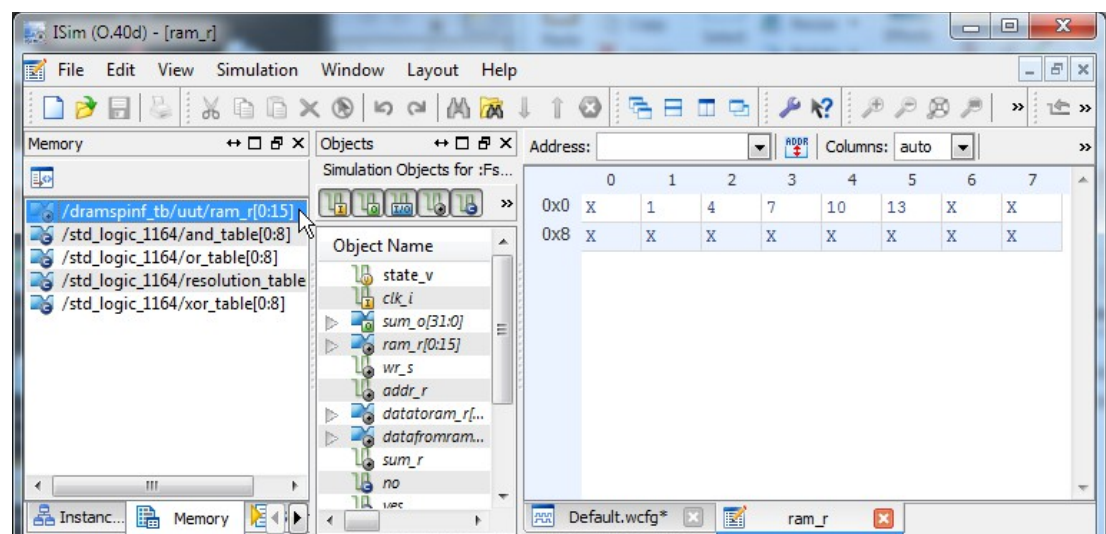
At this point, I've managed to infer a small 16x8 distributed RAM and managed to make it operate correctly (at least, in simulation). But what if I need more memory? That's when block RAMs come into play.

Inferring Block RAM

The FPGA's internal BRAMs provide a total of 18 Kb of memory in a variety of word widths (anywhere from 1 to 36 bits). I can infer BRAM the same way that I did distributed RAM: look-up the associated language template and then customize it for my application. In this case, I use the **1 Clock, 1 Write Port, 1 Read Port** BRAM.
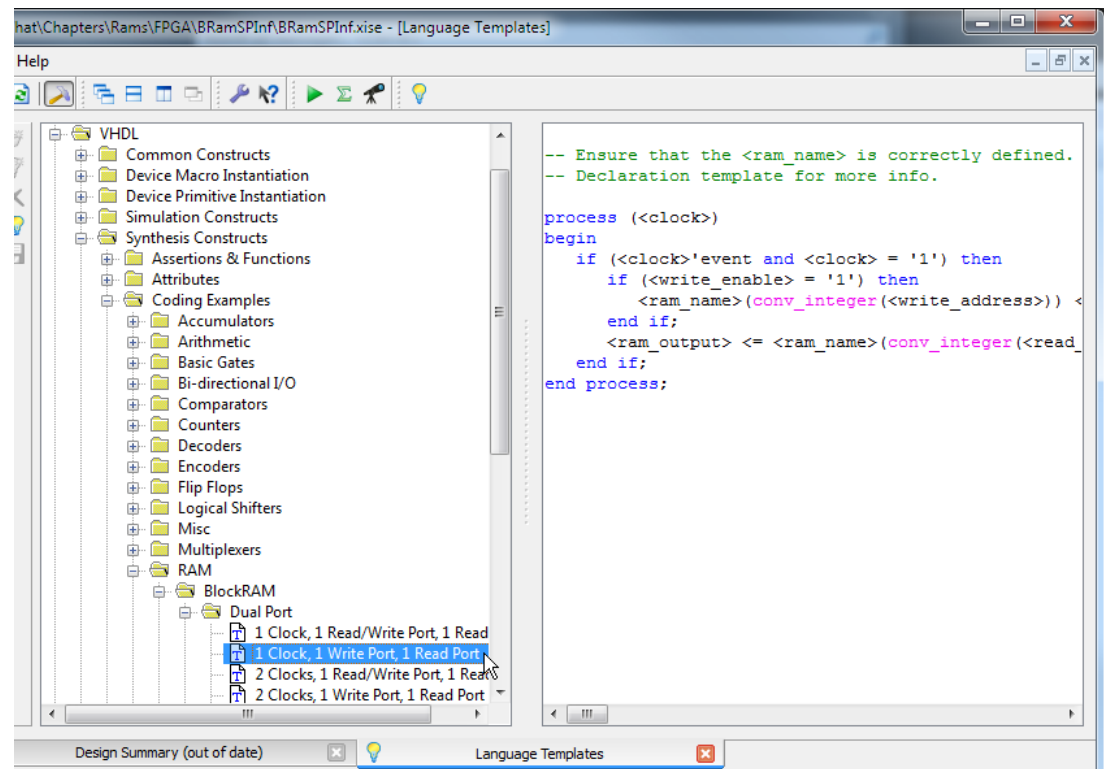
The customized BRAM code is shown on lines 36-46 of BRamSPInf.vhd and it looks very similar to the previous code for the distributed RAM with one important exception: *block RAM reads are synchronous.* The read operation has been pulled into the process block and only gets updated on the rising clock edge. So if an address is applied to a BRAM, the value stored at that address only appears after the next rising clock edge. This has implications on the FSM portion of the application.

```
36    Ram_p : process (clk_i)
37    begin
38      -- Write to the RAM at the given address if the write-enable is high.
39      if rising_edge(clk_i) then
40        if wr_s = YES then
41          ram_r(addr_r) <= dataToRam_r;
42        end if;
43        -- Synchronously read from whatever RAM address is present.
44        dataFromRam_s <= ram_r(addr_r);
45      end if;
46     end process;
```

The one-cycle delay caused by the synchronous read nature of BRAMs has implications on the FSM process. On line 78 below, the calculation has been changed to the product of the current data from the BRAM and the address sent to the BRAM in the previous cycle

(stored in the newly-added signal prevAddr_r). Also, the exit from the READ_AND_SUM_DATA state occurs when the RAM address goes one location past the maximum address in order for read of the last address to be completed (line 75).

Another issue that needs to be handled is the initial entry into the READ_AND_SUM_DATA state. At that time, there is no valid data coming from the BRAM because the read operation from the first address has not completed yet. To keep from corrupting the summation, the prevAddr_r register is cleared upon exiting from the WRITE_DATA state (line 72) so that the first multiplication (line 78) returns a zero. This is a hack that works for this particular application. A more general solution would be to place a one-cycle wait state between the WRITE_DATA and READ_AND_SUM_DATA states to allow the first BRAM read operation time to complete.

```
52    Fsm_p : process (clk_i)
53      type state_t is (INIT, WRITE_DATA, READ_AND_SUM_DATA, DONE);
54      variable state_v : state_t := INIT;    -- Start off in init state.
55    begin
56      if rising_edge(clk_i) then
57        case state_v is
58          when INIT =>
59            wr_s        <= YES;         -- Enable writing of RAM.
60            addr_r      <= MIN_ADDR_C;    -- Start writing data at this address
61            dataToRam_r <= TO_UNSIGNED(1, RAM_WIDTH_C);  -- Initial value.
62            state_v     := WRITE_DATA;    -- Go to next state.
63          when WRITE_DATA =>
64            if addr_r < MAX_ADDR_C then   -- If haven't reach final address ...
65              addr_r      <= addr_r + 1;  -- go to next address ...
66              dataToRam_r <= dataToRam_r + 3; -- and write this value.
67            else  -- Else, the final address has been written...
68              wr_s     <= NO;              -- so turn off writing, ...
69              addr_r   <= MIN_ADDR_C;      -- go back to the start, ...
70              sum_r    <= 0;               -- clear the sum-of-products, ...
71              state_v := READ_AND_SUM_DATA;  -- and go to next state.
72              prevAddr_r <= 0;             -- THIS IS A HACK!
73            end if;
74          when READ_AND_SUM_DATA =>
75            if addr_r <= MAX_ADDR_C + 1 then -- If not the final address+1 ...
76              -- add product of previous RAM address and data read
77              -- from that address to the summation ...
78              sum_r  <= sum_r + TO_INTEGER(dataFromRam_s * prevAddr_r);
79              addr_r <= addr_r + 1;        -- and go to next address.
80            else  -- Else, the final address has been read ...
81              state_v := DONE;             -- so go to the next state.
82            end if;
83            prevAddr_r <= addr_r;          -- Store current address ...
84          when DONE =>                     -- Summation complete ...
85            null;                          -- so wait here and do nothing.
86          when others =>                   -- Erroneous state ...
87            state_v := INIT;               -- so re-run the entire process.
88        end case;
89      end if;
90    end process;
```

Now if I increase RAM_SIZE_C to 256 and synthesize the design, the synthesis report shows that a block RAM is used to build the memory:

```
1   INFO:Xst:3225 - The RAM <Mram_ram_r> will be implemented as BLOCK RAM
2   ---------------------------------------------------------------------
3   | ram_type          | Block                              |         |
4   ---------------------------------------------------------------------
5   | Port A                                                            |
6   |     aspect ratio  | 256-word x 8-bit                   |         |
7   |     mode          | write-first                        |         |
8   |     clkA          | connected to signal <clk_i>        | rise    |
9   |     weA           | connected to signal <wr_s>         | high    |
10  |     addrA         | connected to signal <addr_r>       |         |
11  |     diA           | connected to signal <dataToRam_r>  |         |
12  ---------------------------------------------------------------------
13  | optimization      | speed                              |         |
14  ---------------------------------------------------------------------
15  | Port B                                                            |
16  |     aspect ratio  | 256-word x 8-bit                   |         |
17  |     mode          | write-first                        |         |
18  |     clkB          | connected to signal <clk_i>        | rise    |
19  |     addrB         | connected to internal node         |         |
20  |     doB           | connected to signal <dataFromRam_s>|         |
21  ---------------------------------------------------------------------
22  | optimization      | speed                              |         |
23  ---------------------------------------------------------------------
```

But if I return RAM_SIZE_C to 16 and re-synthesize, the memory is built with distributed RAM again *even though the VHDL specifies a synchronous read operation*. This happens because the synthesizer judges that a distributed RAM is more efficient for smaller memories, so it places a register on the outputs of the distributed RAM to implement the synchronous read behavior and uses the combination in place of a BRAM.

So the synthesizer is able to switch the method for building RAMs in the FPGA based on how much storage is needed. However, you can force the synthesizer to use a particular type of RAM by right-clicking on the **Synthesize-XST** process and selecting **Process Properties...**
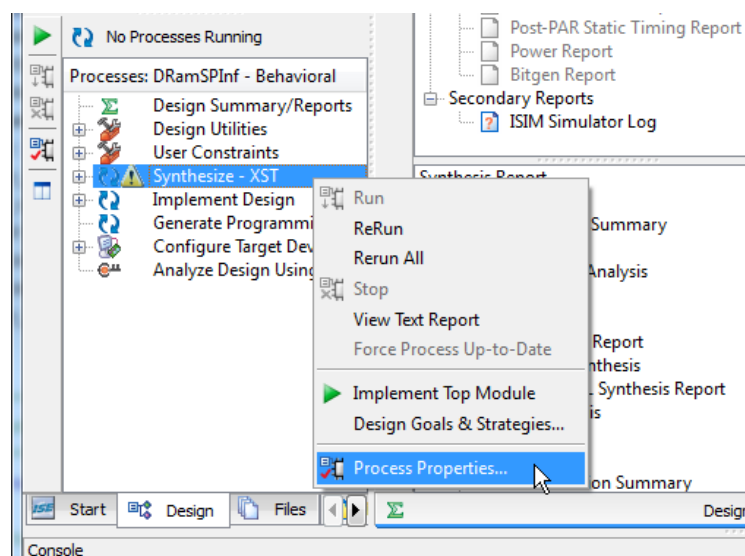
Then set the style of RAM to **Distributed** or **Block**, depending upon what you want. Note that this setting will be applied to *all* the RAMs in your design, so it's probably best to leave it set at **Auto** and let the synthesizer select the best RAM implementation on a case-by-case basis.
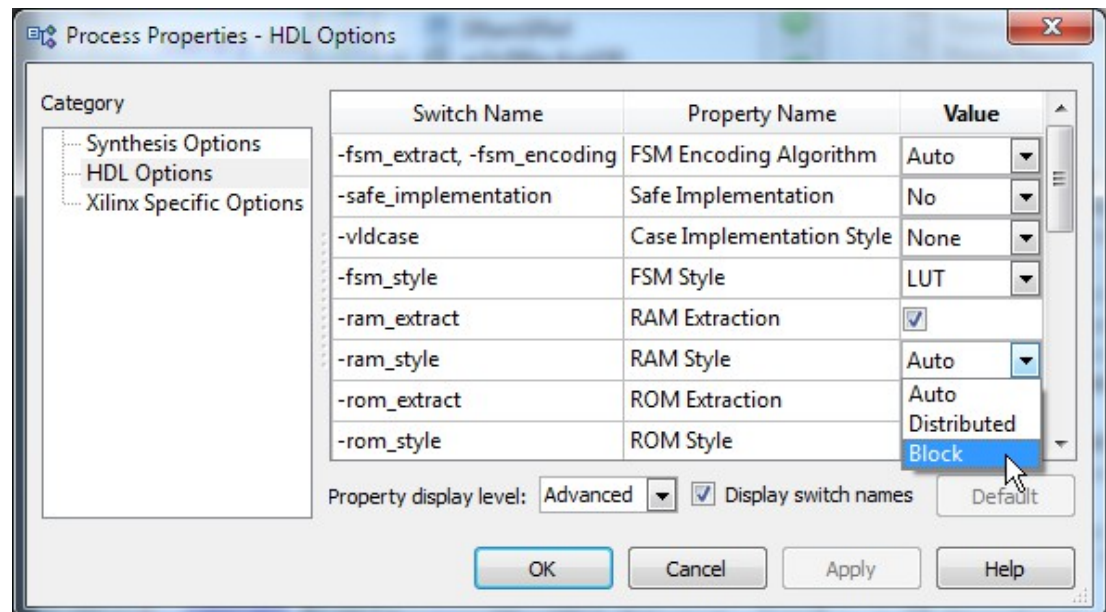
Simulating the 256 x 8 BRAM version of the application gives the following results.
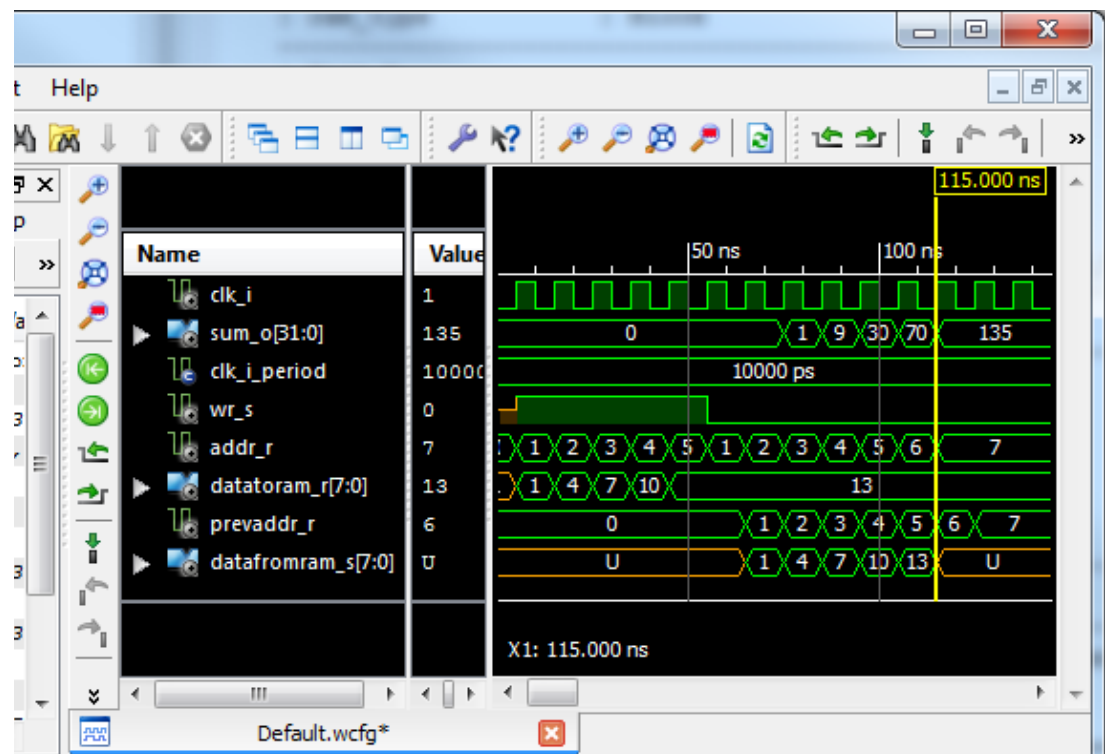
Note that the final sum is the same as for the previous design, but the result appears at 115 ns instead of 105 ns – one full clock-cycle later than for the distributed RAM version.

That's a direct result of the synchronous read nature of the BRAM.

Integrating RAM into the FSM

While I can successfully infer distributed or block RAMs using the VHDL I've already shown, requiring a separate process to describe the RAM is a bit clunky. It would be nice if this could be integrated right into the FSM process so the intent of the code would be clearer. Below is a re-write of the architecture section from the distributed RAM example that does just that (DramSPInfClear.vhd). The separate process for the RAM has been removed and lines 46 and 59 of the FSM process have been changed so that the write and read operations are now expressed as accesses directly to the RAM array.

```
35    Fsm_p : process (clk_i)
36      type state_t is (INIT, WRITE_DATA, READ_AND_SUM_DATA, DONE);
37      variable state_v : state_t := INIT;
38    begin
39      if rising_edge(clk_i) then
40        case state_v is
41          when INIT =>
42            addr_r      <= MIN_ADDR_C;
43            dataToRam_r <= TO_UNSIGNED(1, RAM_WIDTH_C);
44            state_v     := WRITE_DATA;
45          when WRITE_DATA =>
46            ram_r(addr_r) <= dataToRam_r; -- Write to RAM
47            if addr_r < MAX_ADDR_C then
48              addr_r      <= addr_r + 1;
49              dataToRam_r <= dataToRam_r + 3;
50            else
51              addr_r  <= MIN_ADDR_C;
52              sum_r   <= 0;
53              state_v := READ_AND_SUM_DATA;
54            end if;
55          when READ_AND_SUM_DATA =>
56            if addr_r <= MAX_ADDR_C then
57              -- add product of RAM address and data read
58              -- from RAM to the summation …
59              sum_r  <= sum_r + TO_INTEGER(ram_r(addr_r) * addr_r); -- Read RAM
60              addr_r <= addr_r + 1;
61            else
62              state_v := DONE;
63            end if;
64          when DONE =>
65            null;
66          when others =>
67            state_v := INIT;
68        end case;
69      end if;
70    end process;
71
72    -- Output the sum of the RAM address-data products.
73    sum_o <= std_logic_vector(TO_UNSIGNED(sum_r, sum_o'length));
```

The synthesis report indicates that the synthesizer is still able to infer the correct type of RAM from this VHDL, and the simulator shows that the calculated result is the same as that of the original dual-process VHDL.

In addition, I can even increase RAM_SIZE_C to 256 and the synthesizer will shift to using a BRAM while still maintaining the same timing behavior. This is not always guaranteed. You may write obfuscated code that hides its intent, or request read/write behavior that the synthesizer just can't translate. In those cases, you may have to revert to a more explicit style of VHDL like in the first two examples I discussed.

Inferring Multi-Port RAM

So far I've used a simple, single-port RAM with a single address bus and both input and output data buses. The FPGA also supports *dual-port RAMs* which have an additional address and output data bus, thus making it possible to read data from two locations at once. Dual-port RAMs can also be inferred from VHDL as in the inner-product application shown below.

```
1    --*************************************************************************
2    -- Distributed RAM, dual-port, inferred more clearly.
3    --*************************************************************************
4
5    library IEEE;
6    use IEEE.STD_LOGIC_1164.all;
7    use IEEE.NUMERIC_STD.all;
8
9    entity DRamDPInfClear is
10     port (
11       clk_i       : in  std_logic;
12       innerProd_o : out std_logic_vector(31 downto 0) := (others => '0')
13       );
14   end entity;
15
16   architecture Behavioral of DRamDPInfClear is
17     constant NO          : std_logic := '0';
18     constant YES         : std_logic := '1';
19     constant RAM_SIZE_C  : natural    := 16;
20     constant RAM_WIDTH_C : natural    := 8;
21     constant VEC_LEN_C   : natural    := 5;    -- Length of vectors.
22     subtype RamWord_t is unsigned(RAM_WIDTH_C-1 downto 0);
23     type Ram_t is array (0 to RAM_SIZE_C-1) of RamWord_t;
24     signal ram_r         : Ram_t;          -- RAM declaration.
25     signal dataToRam_r   : RamWord_t;      -- Data to write to RAM.
26     signal vec1Ptr_r     : natural range 0 to RAM_SIZE_C-1;  -- Ptr to 1st vec.
27     signal vec2Ptr_r     : natural range 0 to RAM_SIZE_C-1;  -- Ptr to 2nd vec.
28     signal innerProd_r   : natural range 0 to (2**RAM_WIDTH_C - 1)**2;
29   begin
30
31     --*************************************************************************
32     -- State machine that initializes RAM with two vectors and then reads
33     -- the vectors simultaneously from RAM to compute the inner-product.
34     --*************************************************************************
35     Fsm_p : process (clk_i)
36       type state_t is (INIT, WRITE_VECS, INNER_PRODUCT, DONE);
37       variable state_v : state_t := INIT;
38     begin
39       if rising_edge(clk_i) then
40         case state_v is
41           when INIT =>
42             vec1Ptr_r   <= 0;              -- Init ptr for writing data to RAM.
43             dataToRam_r <= TO_UNSIGNED(1, dataToRam_r'length); -- Init data val
44             state_v     := WRITE_VECS;
```

```
45              when WRITE_VECS =>                -- Init the 1st and 2nd vectors.
46                ram_r(vec1Ptr_r) <= dataToRam_r;  -- Write data to RAM at curr ptr.
47                if vec1Ptr_r < 2 * VEC_LEN_C - 1 then  -- Still init'ing both vecs.
48                  vec1Ptr_r   <= vec1Ptr_r + 1;  -- point at next RAM location ...
49                  dataToRam_r <= dataToRam_r + 1;  -- and load with the next value.
50                else  -- Else, finished initializing the vectors.
51                  vec1Ptr_r   <= 0;  -- Init ptr to 1st vector at start of RAM.
52                  vec2Ptr_r   <= VEC_LEN_C;  -- Init ptr to 2nd vec follow 1st vec.
53                  innerProd_r <= 0;  -- Init inner-product summation register.
54                  state_v     := INNER_PRODUCT;  -- Go to next state.
55                end if;
56              when INNER_PRODUCT =>  -- Compute inner-prod of 1st and 2nd vectors.
57                if vec1Ptr_r <= VEC_LEN_C - 1 then  -- Still proc'ing vec elems...
58                  -- Add the product of the current elements from the 1st
59                  -- and 2nd vectors using two simultaneous reads of the RAM.
60                  innerProd_r <= innerProd_r +
61                                 TO_INTEGER(ram_r(vec1Ptr_r) * ram_r(vec2Ptr_r));
62                  vec1Ptr_r   <= vec1Ptr_r + 1;  -- Inc to next element of 1st vec.
63                  vec2Ptr_r   <= vec2Ptr_r + 1;  -- Inc to next element of 2nd vec.
64                else  -- Else, all the vector elements have been processed ...
65                  state_v := DONE;             -- so go to the next state.
66                end if;
67              when DONE =>                      -- Inner-product complete ...
68                null;                           -- so wait here and do nothing.
69              when others =>                    -- Erroneous state ...
70                state_v := INIT;                -- so re-run the entire process.
71            end case;
72          end if;
73        end process;
74
75        -- Output the inner-product.
76        innerProd_o <= std_logic_vector(
77                       TO_UNSIGNED(innerProd_r, innerProd_o'length));
78
79      end architecture;
```

The basic idea of the inner-product FSM is to write a sequence of values to the RAM in the WRITE_VECS state. Then, two pointers are initialized: one pointing at the first vector at the beginning of the RAM (line 51), and another pointing at the second vector immediately following the first vector (line 52). In the INNER_PRODUCT state, the contents of the RAM addressed by each pointer are read out simultaneously, multiplied and added to the inner-product register (lines 60-61), after which the pointers are incremented (lines 62-63). The basic operations are depicted below.
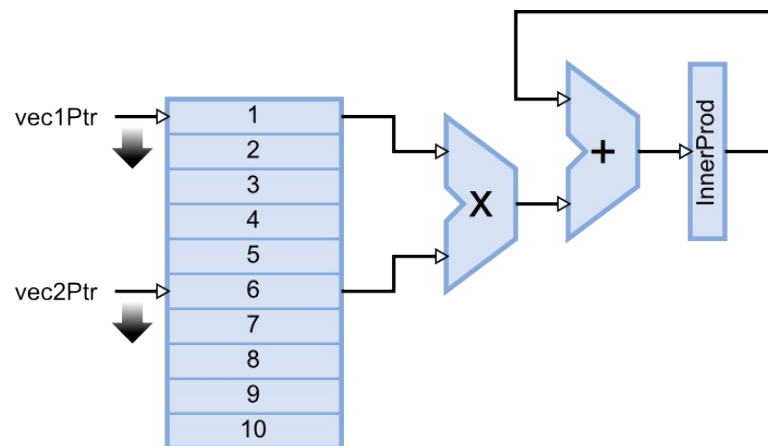
After the VHDL shown above is synthesized, the synthesis report shows a single dual-port distributed RAM has been used (line 6).

```
1    Advanced HDL Synthesis Report
2
3    Macro Statistics
4    # FSMs                                              : 1
5    # RAMs                                              : 1
6     16x8-bit dual-port distributed RAM                : 1
7    # Multipliers                                       : 1
8     8x8-bit multiplier                                : 1
9    # Adders/Subtractors                                : 4
10    16-bit adder                                       : 1
11    4-bit adder                                        : 2
12    8-bit adder                                        : 1
13   # Registers                                         : 32
14    Flip-Flops                                         : 32
15   # Comparators                                       : 2
16    4-bit comparator less                             : 1
17    4-bit comparator lessequal                        : 1
```

Given the data the RAM is initialized with, the inner product should be 1x6 + 2x7 + 3x8 + 4x9 + 5x10 = 130. Running a simulation on the VHDL shows that it produces the correct result.
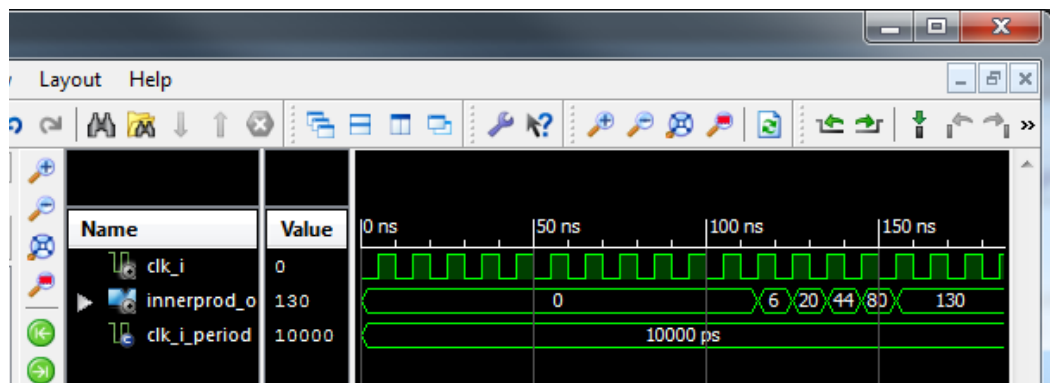
If a dual-port RAM can be synthesized, are triple- and quad-port RAMs possible? The answer is yes! Although the FPGA hardware itself is limited to building single and dual-port RAMs, the synthesizer can replicate data into multiple RAMs to provide the number of simultaneous read operations that are required. For example, here is how a triple-port RAM is realized:
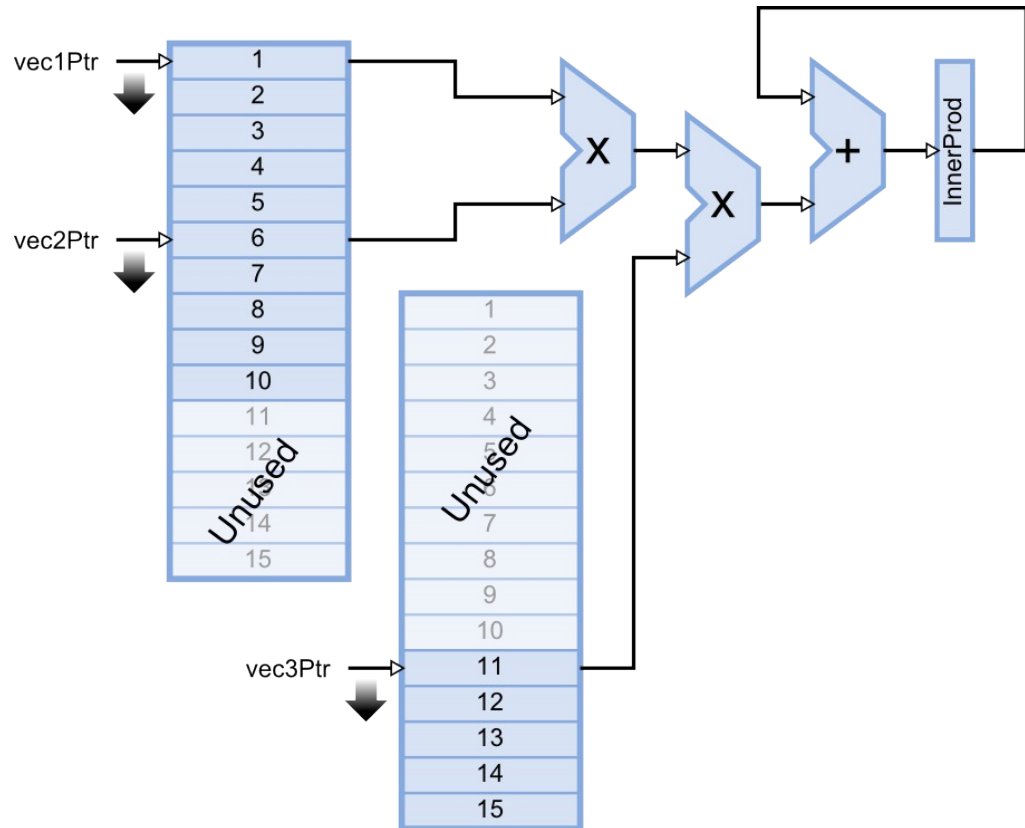
Notice that the number of RAM blocks has doubled, but half of it goes unused. That's usually what happens with digital systems: you can make them go faster by using more circuitry, or slower using less, but you seldom achieve speed and efficiency at the same time.

Generating RAM

To demonstrate the use of ISE's CORE Generator, I'm going to use it to re-build the BRAM-version of the single-port RAM example (BramSPInf). I'll begin with an empty project (BramSPCoreGen) and add a new source file for the soon-to-be-generated RAM to it.
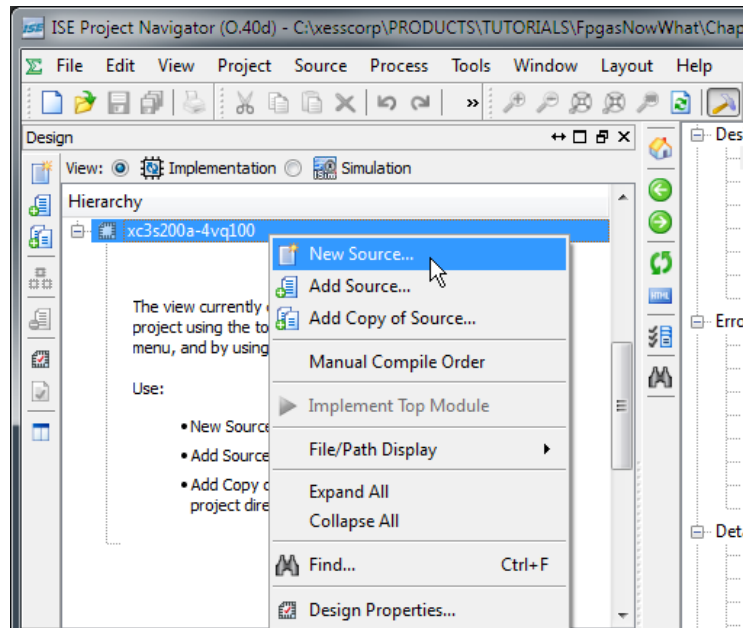
For this example, I've given the CORE Generator source file the descriptive name of
BRAM.
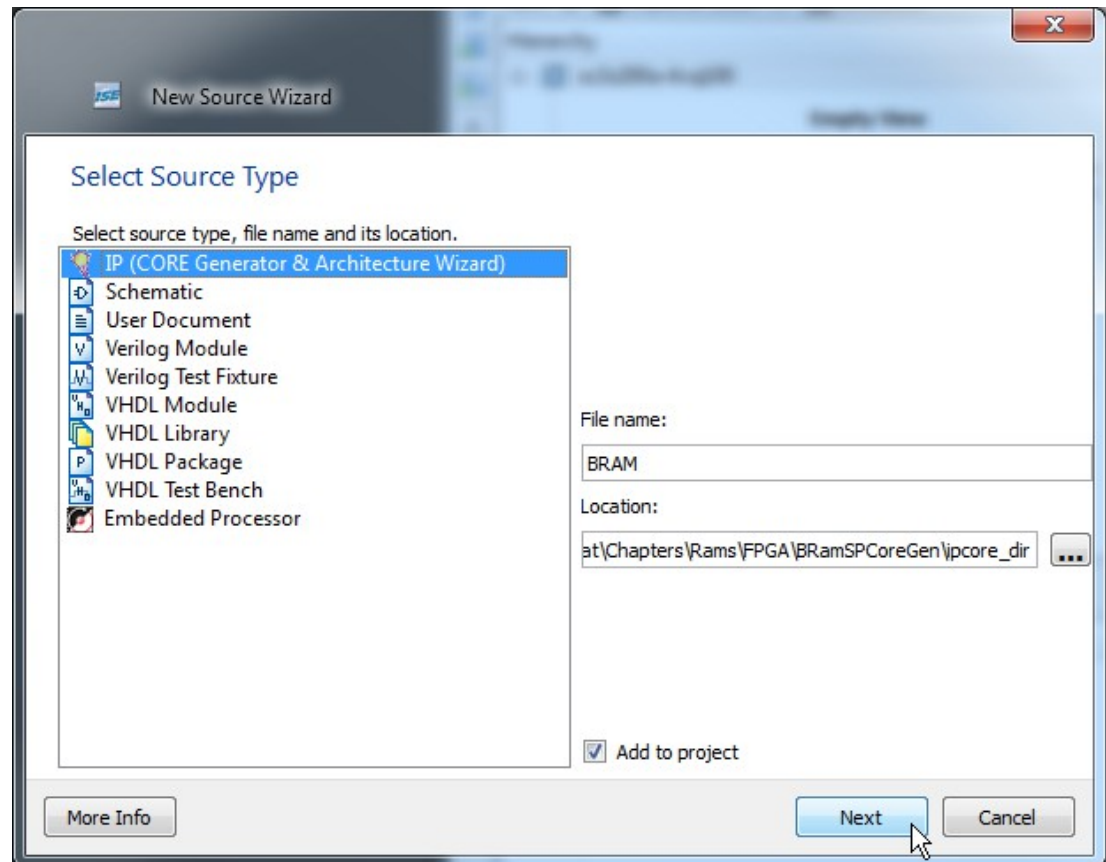
The CORE Generator gives me a choice of either block or distributed RAMs. For this example, I chose block RAM.
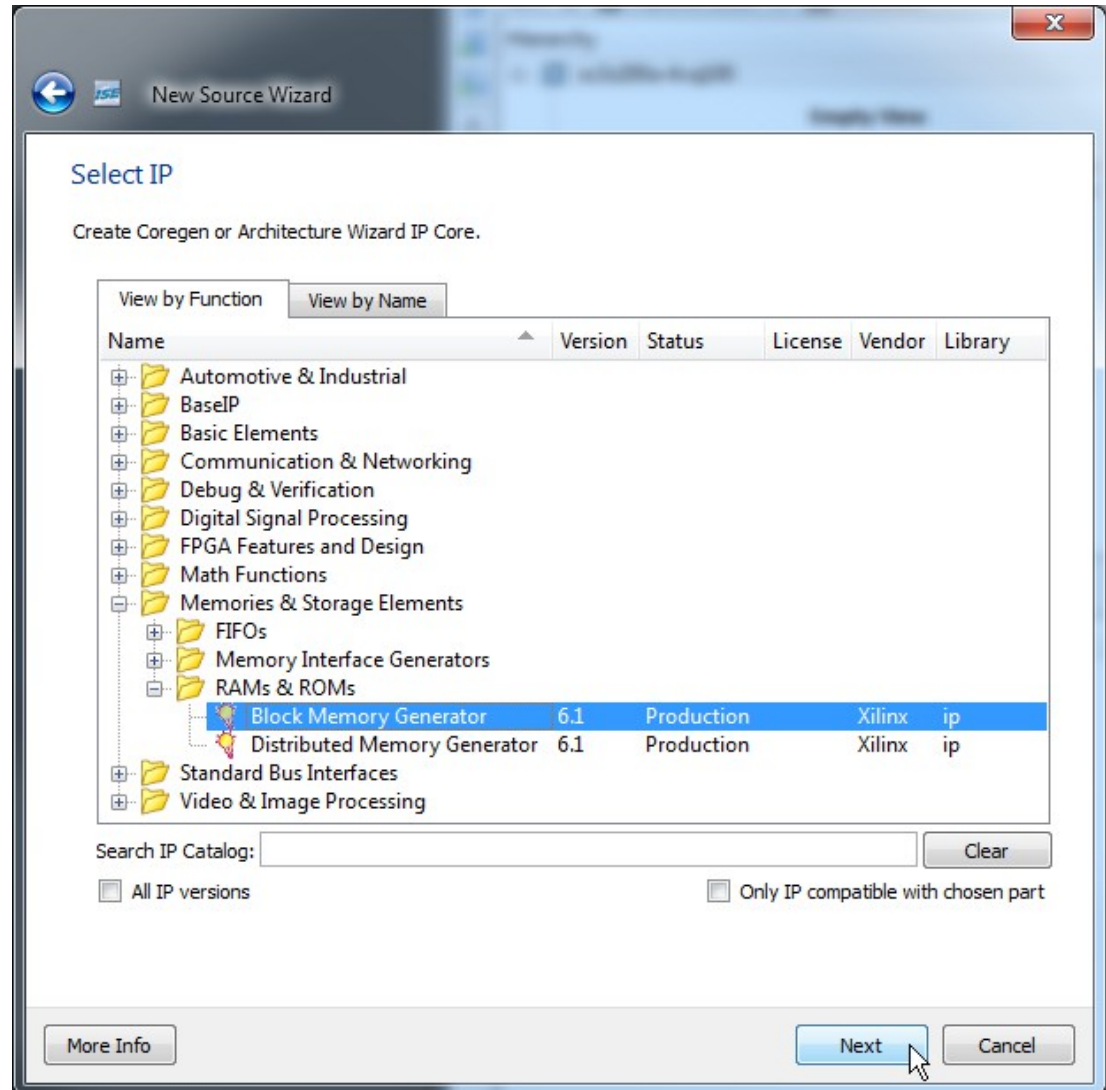
Then I just click on the **Finish** button to add the BRAM core to the project.
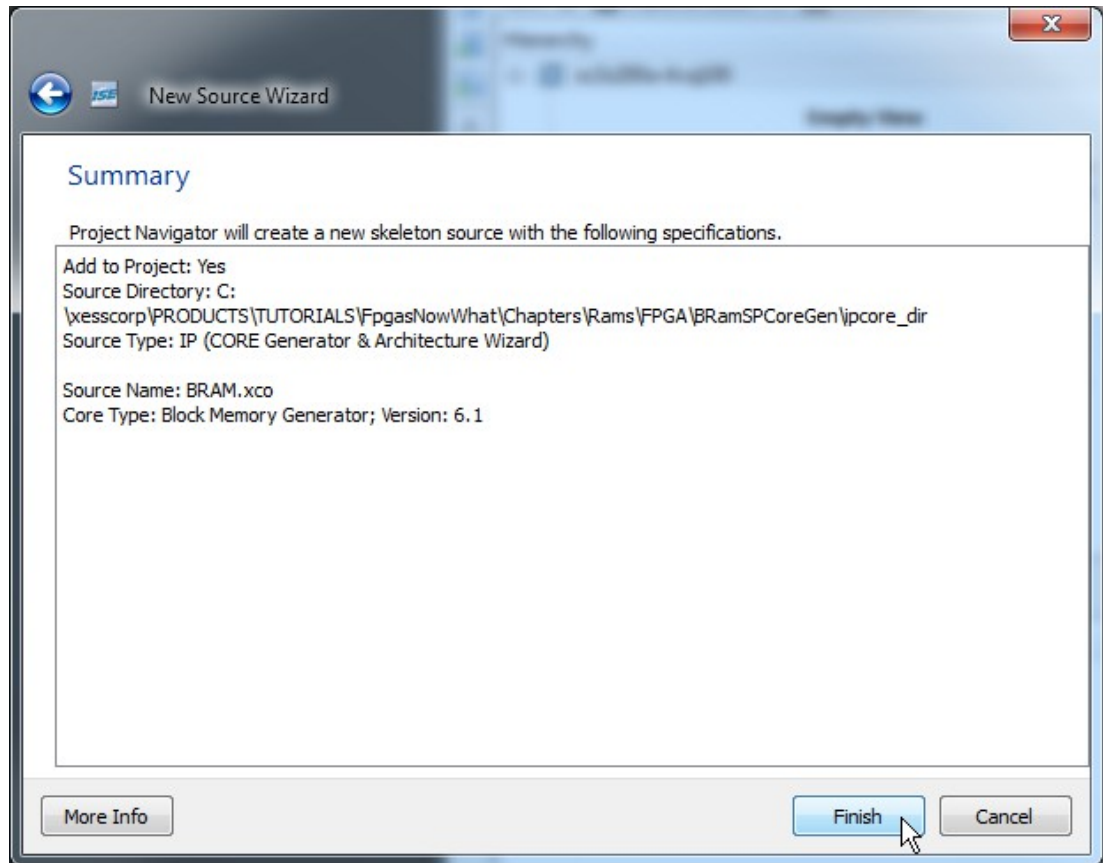
Now the real fun begins! I have to set all the configuration parameters that define how the block RAM works. There isn't much to do in the first configuration screen.
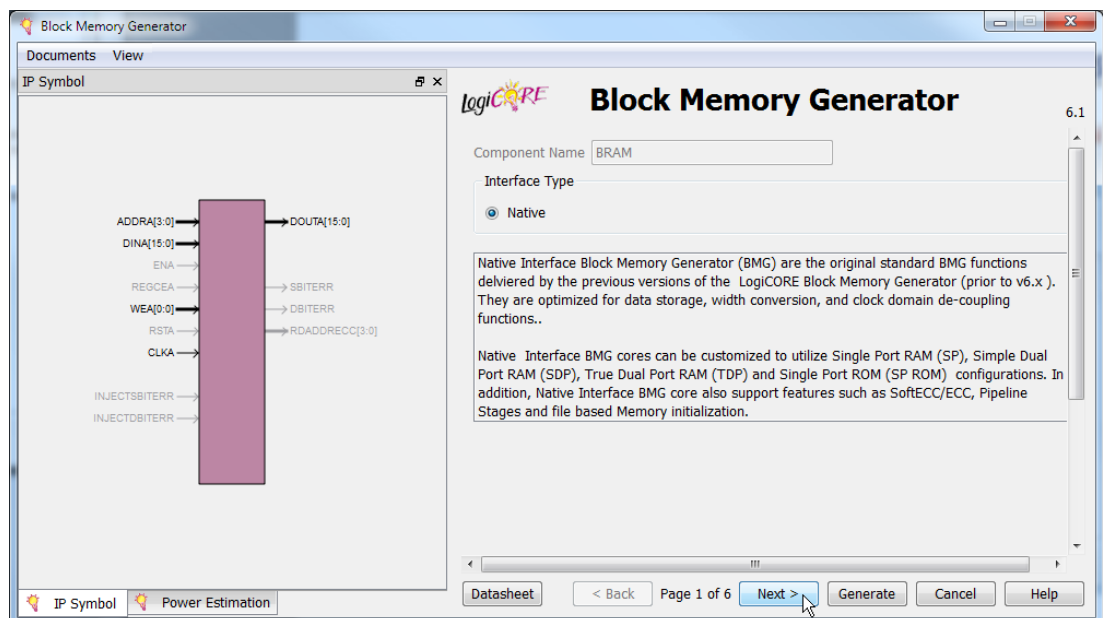
On the next screen, all I need to do is select a single-port RAM and move on.
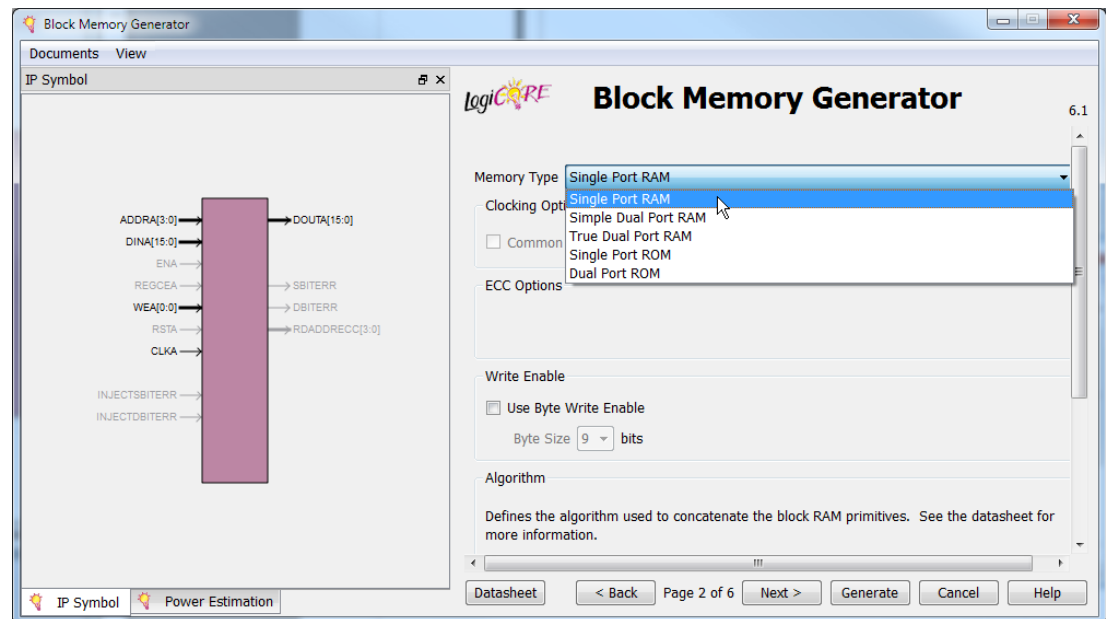
Then I set the RAM's word width (eight bits) and the number of memory words it contains (256).
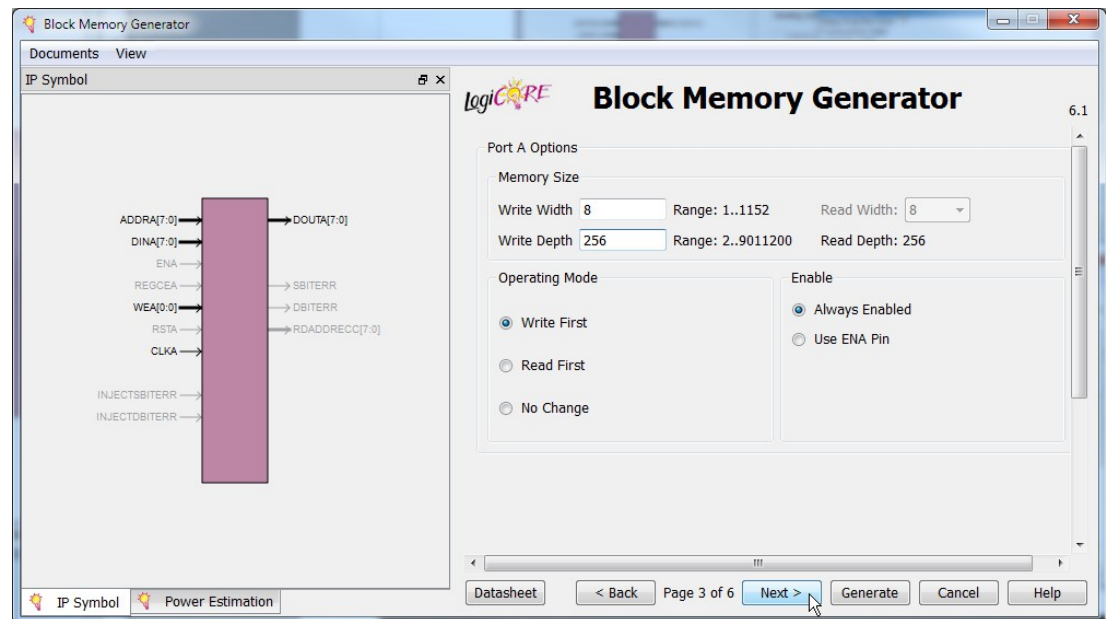
I don't want any registers on the output of the generated RAM (although it will still be a synchronous-read RAM even without them because it will be made from BRAMs) and I don't want to initialize the RAM, so I can just click **Next** on this screen.
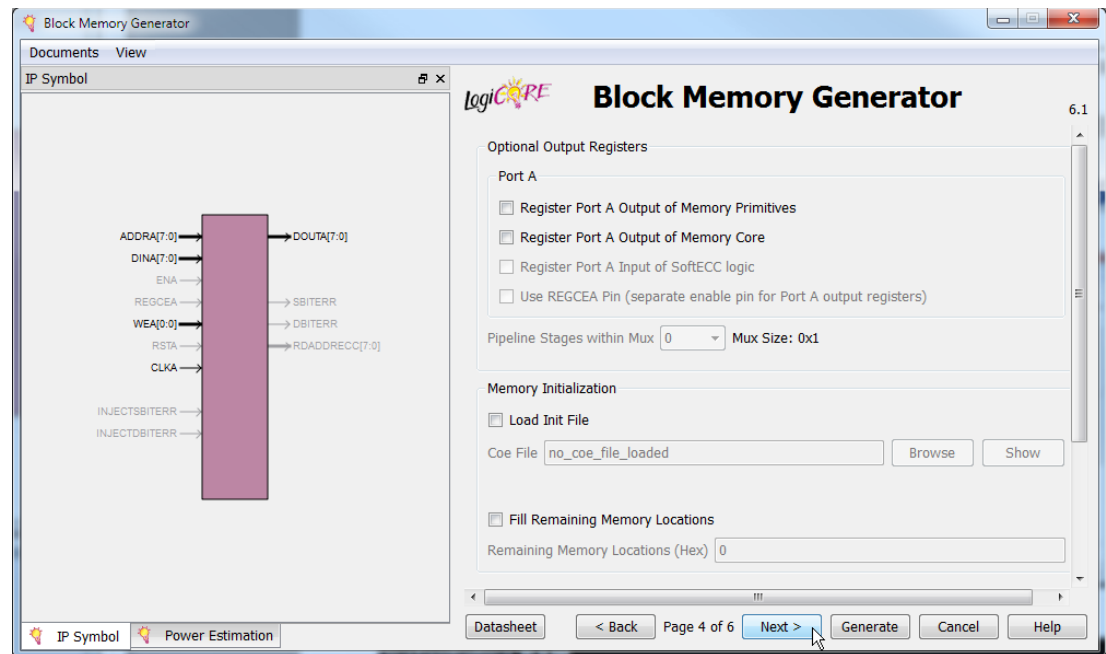
I also don't need a reset for the output of the RAM.
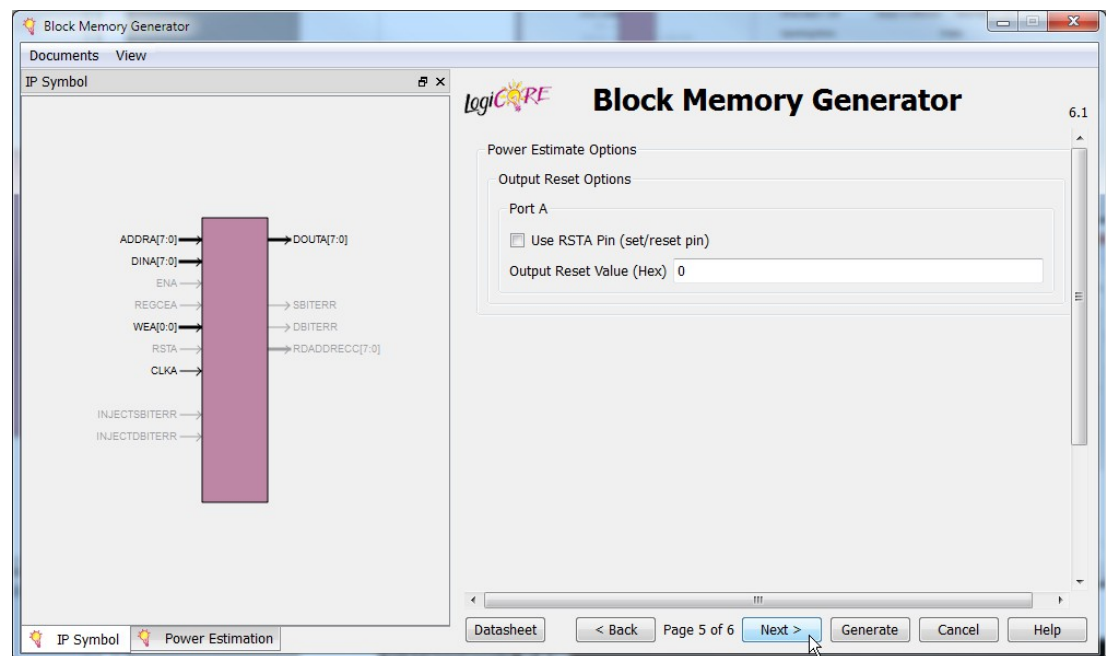
Finally, I've set all the configuration parameters and CORE Generator can build the RAM as specified.
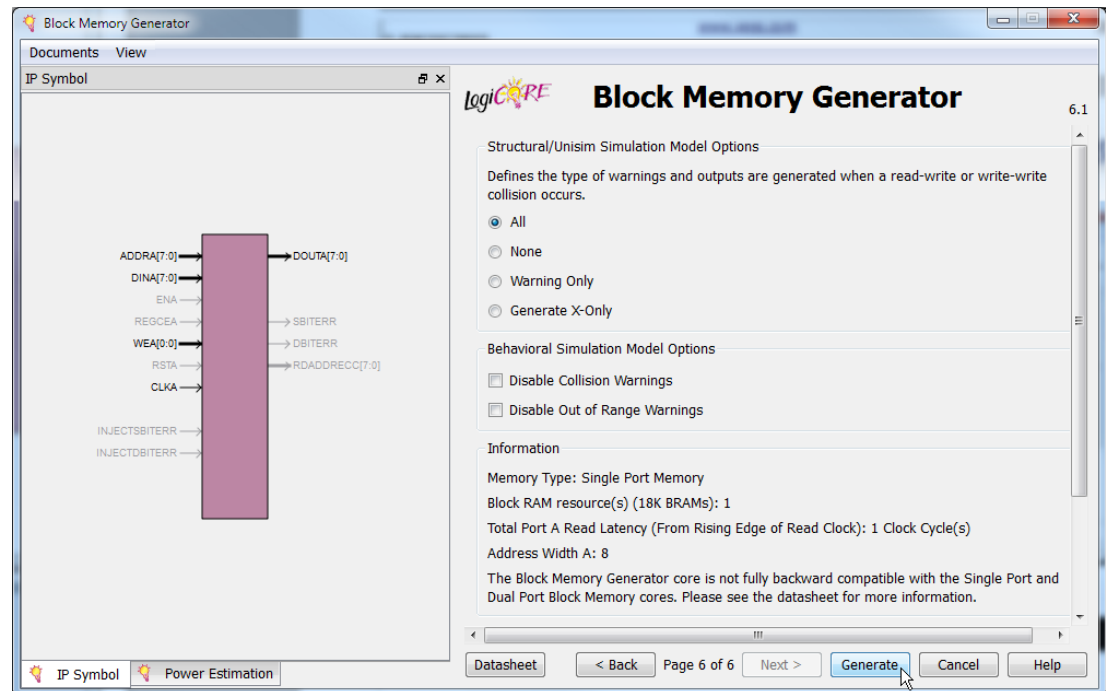
After grinding away for about ten seconds, the BRAM.xco source file gets added to the project. If I decide I need to change any of the RAM parameters, I can just double-click the BRAM.xco file and proceed through the configuration screens again while making modifications.
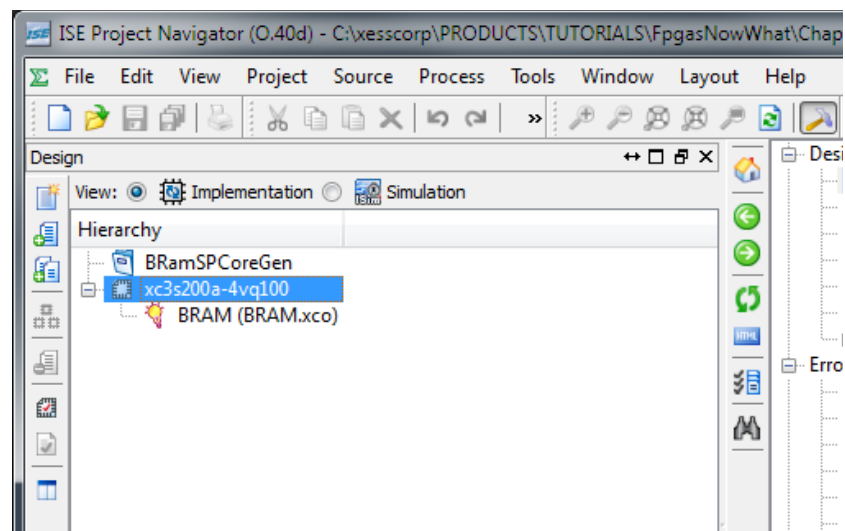
Now that I have the RAM built by the CORE Generator, how do I use it in my VHDL file? To figure that out, highlight the BRAM module and then double-click the **View VHDL Instantiation Template** in the **Process** pane.
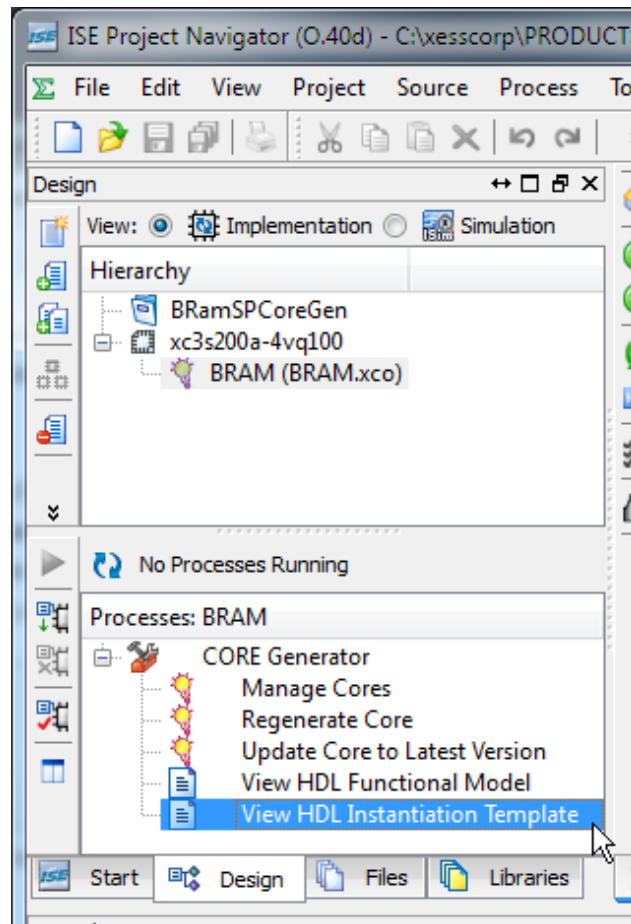
The template file that opens has two sections of interest. The first is the component declaration for the generated RAM that I'll have to include verbatim in the architecture header.

```
38  ------------- Begin Cut here for COMPONENT Declaration ------ COMP_TAG
39  COMPONENT BRAM
40    PORT (
41      clka : IN STD_LOGIC;
42      wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
43      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
44      dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
45      douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
46    );
47  END COMPONENT;
48  -- COMP_TAG_END ------ End COMPONENT Declaration ------------
```

The second section has a template for the generated RAM that I have to paste into the body of my architecture and then edit the signal names.

```
53  ------------- Begin Cut here for INSTANTIATION Template ----- INST_TAG
54  your_instance_name : BRAM
```

```
55      PORT MAP (
56        clka => clka,
57        wea => wea,
58        addra => addra,
59        dina => dina,
60        douta => douta
61      );
62   -- INST_TAG_END ------ End INSTANTIATION Template ------------
```

Now I can take the existing code from the BramSPInf.vhd file, put it in a file called BramSPCoreGen.vhd and add it to this project. Then all I need to do is edit the file a bit as follows.

The component declaration is entered as-is into the architecture header on lines 17-25.

```
16   architecture Behavioral of BRamSPCoreGen is
17     component BRAM
18       port (
19         clka  : in  std_logic;
20         wea   : in  std_logic_vector(0 downto 0);
21         addra : in  std_logic_vector(7 downto 0);
22         dina  : in  std_logic_vector(7 downto 0);
23         douta : out std_logic_vector(7 downto 0)
24         );
25     end component;
```

Next, the signal declaration for the inferred RAM isn't needed anymore, so it's been removed. I've kept the RamWord_t subtype definition (line 32), however, because I still need it for declaring the input and output buses of the RAM.

```
26       constant NO          : std_logic := '0';
27       constant YES         : std_logic := '1';
28       constant RAM_SIZE_C  : natural   := 256;  -- Number of words in RAM.
29       constant RAM_WIDTH_C : natural   := 8; -- Width of RAM words.
30       constant MIN_ADDR_C  : natural   := 1; -- Process RAM from this address ...
31       constant MAX_ADDR_C  : natural   := 5; -- ... to this address.
32       subtype RamWord_t is unsigned(RAM_WIDTH_C-1 downto 0);   -- RAM word type.
33       signal wr_s          : std_logic;      -- Write-enable control.
34       signal addr_r, prevAddr_r : natural range 0 to RAM_SIZE_C-1;-- RAM address.
35       signal dataToRam_r   : RamWord_t;      -- Data to write to RAM.
36       signal dataFromRam_s : RamWord_t;      -- Data read from RAM.
37       signal sum_r         : natural range 0 to RAM_SIZE_C * (2**RAM_WIDTH_C)-1;
```

At the end of the architecture header, I declare three new eight-bit, std_logic_vector signals for attaching to the RAM module's address and input/output data buses. This makes it a bit easier to attach the RAM (which uses std_logic_vector signals for its I/O) to the FSM (which uses natural and RamWord_t types of signals).

```
38     signal addrCoreGen_s : std_logic_vector(7 downto 0);
39     signal dataToRamCoreGen_s : std_logic_vector(7 downto 0);
40     signal dataFromRamCoreGen_s : std_logic_vector(7 downto 0);
41   begin
42
```

The first change in the architecture body is to remove the VHDL process that inferred the BRAM and replace it with an instantiation of the RAM created using the CORE Generator (lines 46-53). The clock for the RAM is connected to the master clock of the application (line 48), and the RAM write-enable from the FSM attaches directly to the single bit of the RAM's write-enable bus (line 49). But on lines 50-52, the RAM's address and data buses are routed to the intermediate signals I defined previously. Then these intermediate signals are attached to the analogous signals from the FSM (lines 56-58) after a bit of type-casting to keep the synthesizer from complaining about mismatched signal types.

```
43    --*********************************************************************
44    -- Instantiate the block RAM created by CORE Generator.
45    --*********************************************************************
46    ram_u0 : BRAM
47      port map (
48        clka   => clk_i,
49        wea(0) => wr_s,
50        addra  => addrCoreGen_s,
51        dina   => dataToRamCoreGen_s,
52        douta  => dataFromRamCoreGen_s
53        );
54
55    -- Connect the RAM signals to the signals from the FSM.
56    dataToRamCoreGen_s <= std_logic_vector(dataToRam_r);
57    dataFromRam_s <= RamWord_t(dataFromRamCoreGen_s);
58    addrCoreGen_s <= std_logic_vector(TO_UNSIGNED(addr_r,8));
```

That's all there is to it; the FSM code doesn't have to change at all since the generated-RAM behaves the same as the VHDL-inferred RAM.

The synthesis report shows that a single BRAM is used for this design (line 11):

```
1    Device utilization summary:
2    ---------------------------
3
4    Selected Device : 3s200avq100-4
5
6     Number of Slices:                     47  out of   1792    2%
7     Number of Slice Flip Flops:           44  out of   3584    1%
8     Number of 4 input LUTs:               88  out of   3584    2%
9     Number of IOs:                        33
10    Number of bonded IOBs:                33  out of     68   48%
11    Number of BRAMs:                       1  out of     16    6%
12    Number of MULT18X18SIOs:               1  out of     16    6%
13    Number of GCLKs:                       1  out of     24    4%
```

Finally, the simulation results for this design are identical to those for the inferred BRAM in both the calculated result and the timing of the signal waveforms.
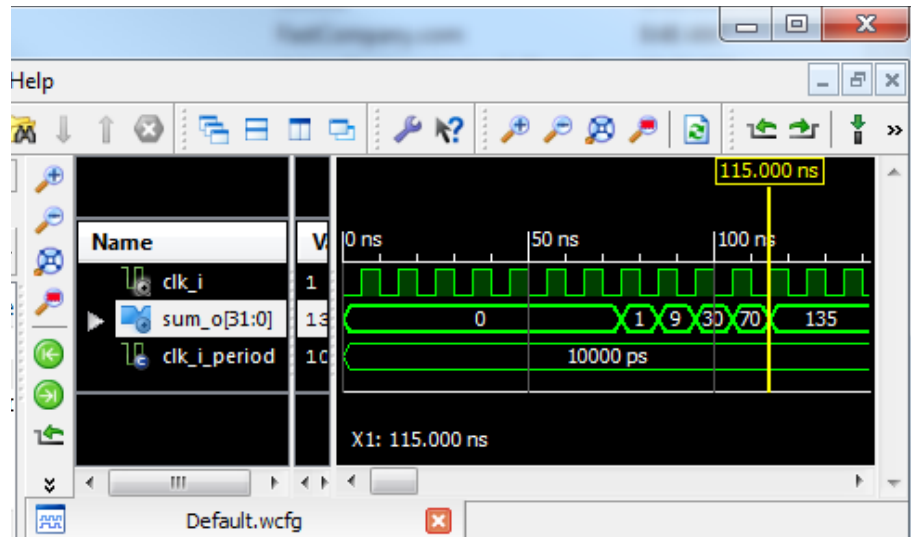
So after all that, was using CORE Generator worth it? Here are some of the disadvantages:

- CORE Generator creates modules which use std_logic_vectors for their I/O buses. This always requires some fiddling with the types to make the connections with other parts of the VHDL that are using more application-appropriate types. (Using std_logic_vector for module I/O is actually standard industry practice, so I can't really fault CORE Generator for this.)

- Changing anything about the RAM – even width or size – requires another trip through CORE Generator. And, after that, you may have to edit the component declaration and instantiation and change the widths of the connecting buses.

- CORE Generator .xco files aren't portable to the design software for non-Xilinx FPGAs.

Even with these problems, there are times I would use CORE Generator:

- If I needed to build a large and/or wide RAM that takes advantage of some of the special features of the FPGA hardware, then CORE Generator can be easier to use than trying to craft the behavior in VHDL.

- If I needed to build something really complicated (e.g., an FFT module), then CORE Generator would save me a lot of time writing and verifying the module.

Instantiating RAM

I already used instantiation in the previous example to drop the RAM built with CORE Generator into the top-level VHDL code. Now I'm going to use it again for building an interface to the SDRAM on the XuLA board.

The Winbond W9812 SDRAM chip stores 8-million sixteen-bit words – much more than the internal RAMs in the FPGA can hold. This large memory works by storing bits as charges on millions of tiny capacitors. Since the charge on a capacitor tends to leak away over time (thus losing the bit that was stored there), the SDRAM memory cells need to be refreshed every so often to restore their charge. (In essence, the SDRAM has to be continually reminded of what it remembers so it doesn't forget. Kind of like me.)

SDRAMs are internally organized into banks, rows and columns. For example, the W9812 is divided into four banks, each containing 4096 rows, with each row made up of 512 columns, and with each column holding a sixteen-bit number. To access an individual memory location requires a 23-bit address ($4 \times 4096 \times 512 = 2^{23}$). In order to cut down on the number of address pins needed on the physical chip package, a multiplexed address bus is used consisting of a two-bit bank-address bus and a twelve-bit row and column address bus (for a total of fourteen address pins). To access a memory location, the bank address is set and the row address is strobed-in which activates an internal row of that bank in the SDRAM. After the activation completes, column elements can be accessed by strobing in the column addresses without the need to re-enter the row address.

The need for periodic refreshing, address multiplexing, and row-activation makes it more complicated to use an SDRAM than the distributed or block RAMs I've already discussed. For this reason, I created a controller module that simplifies the interface so SDRAM looks more like a regular RAM with just an address and data bus. Here's a block diagram for the application that sums the products of addresses and data, but using the SDRAM on the XuLA board instead of internal RAMs. The SDRAM controller module (SdramCntl) handles breaking the address into pieces, activating the appropriate bank, row and column, and inserts the periodic refreshes needed by the SDRAM.
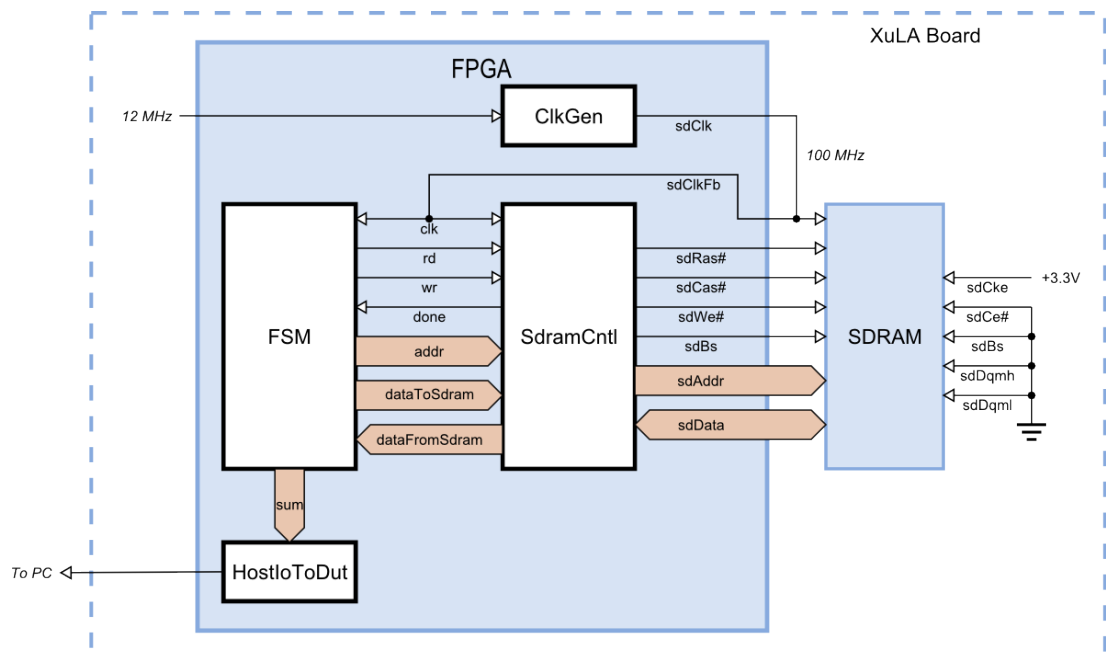
On one side of the SDRAM controller module is a host interface that connects to the FSM with the following signals:

clk: This is the master clock input. Anything the SDRAM controller connects to has to be synchronous with this clock.
rd: This active-high input initiates a read of a single word from the SDRAM. It is sampled on the rising clock edge and must be held high until the read process completes as indicated by the done signal going high. The read control must be lowered before the next rising clock edge after the done signal goes high or else another read operation will start.
wr: This active-high input initiates a write of a single word from the SDRAM. It is sampled on the rising clock edge and must be held high until the done signal indicates the write process completes. The write control must be lowered before the next rising clock edge after the done signal goes high or else another write operation will start.

done: This synchronous output signal goes high to indicate the completion of the currently active read or write operation. It remains high for a single clock cycle.

addr: The address of the SDRAM word that is to be read or written is passed through this input bus. The address value must be held stable until the done signal goes high. The two most-significant bits of addr correspond to the bank address bits of the SDRAM, the next $\log_2(N_{ROWS})$-bits of addr correspond to the row address within that bank, and the least-significant $\log_2(N_{COLS})$-bits correspond to the column address within that row.

dataToSdram: The data to be written to the SDRAM enters through this input bus. The data value must be held stable until the done signal goes high.

dataFromSdram: The data read from the SDRAM comes out on this bus. This data must be latched by the host-side logic on the rising clock edge after the done signal goes high.

The signals on the other side of the SdramCntl module attach directly to the SDRAM chip:

sdRas#: This active-low output drives the row-address strobe input of the SDRAM.

sdCas#: This active-low output drives the column-address strobe input of the SDRAM.

sdWe#: This active-low output drives the write-enable input of the SDRAM.

sdBs: This output selects one of the two banks of SDRAM. (Due to a shortage of pins on the XuLA FPGA chip, the other bank-select pin of the SDRAM had to be permanently tied to ground, thus preventing access to the other two banks.)

sdAddr: The row and column addresses for the SDRAM memory location are output on this bus.

sdData: The data word to be written to SDRAM exits the FPGA on this bus during write operations, and data from the SDRAM enters the FPGA on this bus during read operations.

sdCke: This active-high output drives the clock-enable input of the SDRAM. (Due to a shortage of pins on the XuLA FPGA chip, the clock-enable pin on the SDRAM was tied high, thus keeping the SDRAM from ever entering a quiescent state.)

sdCs#: This active-low output drives the chip-select of the SDRAM. (Due to a shortage of pins on the XuLA FPGA chip, the chip-select pin on the SDRAM was tied to ground, thus keeping the SDRAM permanently enabled.)

sdDqmh: This active-high output drives the SDRAM input that disables the drivers for the upper-half of the data bus during read operations. (Due to a shortage of pins on the XuLA FPGA chip, this pin on the SDRAM was tied low so the upper-half of the databus is always enabled.)

sdDqml: This active-high output drives the SDRAM input that disables the drivers for the lower-half of the data bus during read operations. (Due to a shortage of pins on the XuLA FPGA chip, this pin on the SDRAM was tied low so the lower-half of the databus is always enabled.)

The SDRAM controller module supports several modes of operation, but I'll use it here in its simplest and lowest-performance form to do non-pipelined reads and writes of SDRAM.

The timing waveforms for a non-pipelined read operation are shown below. These assume the read operation accesses a memory location in the currently active bank and row of the SDRAM. The sequence of actions is:

Cycle 1: The SDRAM address is applied and the read control signal is driven high. The address and read control must be held stable until the done signal goes high.

Cycle 2: The column address is output on the pins that go to the SDRAM chip and the SDRAM control signals are set to initiate a read operation.

Cycle 3: The SDRAM initiates a read of the given column address on the rising clock edge.

Cycle 4: The SDRAM waits for the data to arrive from the given address.

Cycle 5: The data from the SDRAM arrives sometime during this cycle and is guaranteed

to be stable by the end of the cycle.

Cycle 6: The data from the SDRAM is clocked into a register on the rising clock edge. The done signal goes high to signal the host-side logic that the data is available on the dataFromRam bus. The read control must be lowered before the next rising clock edge or else another read operation will be initiated.

Cycle 7: The done signal goes low again but the output data remains stable until another read operation is completed.
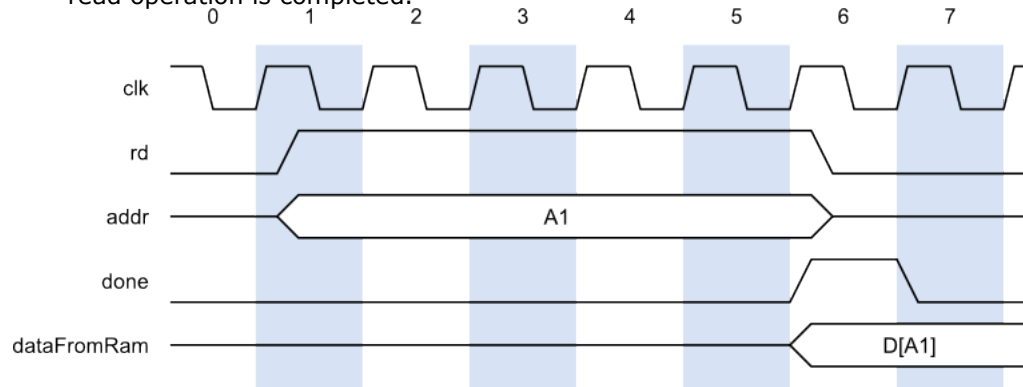
Next up are the timing waveforms for a non-pipelined write operation. This example assumes the write operation accesses a memory location in the currently active bank and row of the SDRAM. The sequence of actions is:

Cycle 1: The SDRAM address and the data to be stored there are applied and the write control signal is driven high. The address, data and write control must be held stable until the done signal goes high.

Cycle 2: The data and the column address are output on the pins that go to the SDRAM chip and the SDRAM control signals are set to initiate the write operation. The done signal goes high because the SDRAM controller is effectively done at this point since the SDRAM can complete the write operation on its own.

Cycle 3: On the rising clock edge the SDRAM latches the address and data and initiates a write operation. The output drivers on the data bus are disabled to free the SDRAM data bus.

Cycles 4 and 5: The SDRAM continues its internal operations to write the data into the given address.
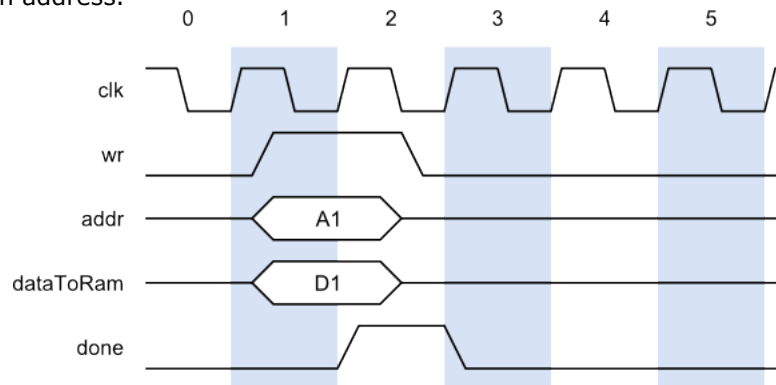
In the previous sequence of actions for a read or write operation, it was assumed the operation was initiated as soon as the appropriate control signal was asserted. There are several cases when the SDRAM controller delays the initiation of an operation:

When a row is being refreshed. In this case, SDRAM controller completes the row refresh operation. Then the SDRAM banks are precharged and the bank and row containing the given address are activated. Then the read or write operation can progress as described above.

When the given address is not in the currently active bank or row of the SDRAM.
In this case, the controller precharges the SDRAM banks and the bank and row containing the given address is activated. Then the read or write operation can progress as described above.

When a previous write operation is still in progress. For write operations, the SDRAM controller indicates the operation is complete before the SDRAM chip has actually finished storing the data. If another read or write operation is initiated before the SDRAM chip is actually done, then the SDRAM controller will delay the current operation until the SDRAM is completely finished writing the previous data. Then the current operation proceeds, including any preceding activation steps that are needed.

You don't actually have to be concerned with these delays because the SDRAM controller handles them automatically. All you have to do is initiate a read or write operation and hold your signals steady until the done signal goes high.

In adition to the FSM and SDRAM controller modules, there is a clock generator module (ClkGen) that takes the 12 MHz input clock to the FPGA and ramps it up to 100 MHz. This high-speed clock exits the FPGA and goes to the clock input of the SDRAM. It also gets fed back into the FPGA and drives the clock inputs of the FSM and SDRAM controller. This effectively synchronizes the internal operations of the FPGA with those of the SDRAM since each receives a clock signal with approximately the same amount of accumulated chip-I/O and PCB trace delays.

Finally, I include a single HostIoToDut module to transfer the value in the summation register back to the PC. Looking at this value will tell me if the FPGA and SDRAM performed correctly or not.

The top-level module of the SdramSPInst ISE project that connects all these modules together is shown below. The first order of business is to pull in the necessary VHDL packages for the clock generator, SDRAM controller and the PC-to-XuLA transfer modules (lines 8-10).

```
1    --**********************************************************************
2    -- SDRAM, single-port, instantiated.
3    --**********************************************************************
4
5    library IEEE;
6    use IEEE.STD_LOGIC_1164.all;
7    use IEEE.NUMERIC_STD.all;
8    use work.ClkgenPckg.all;              -- For the clock generator module.
9    use work.SdramCntlPckg.all;           -- For the SDRAM controller module.
10   use work.HostIoPckg.HostIoToDut;      -- For the FPGA<=>PC transfer link
     module.
11
```

In the entity declaration, the output for the summation has been removed since the HostIo package will be used to send it to the PC. The I/O signals for connecting the FPGA to the SDRAM have also been added.

```
12   entity SdramSPInst is
13     port (
14       fpgaClk_i : in    std_logic;  -- 12 MHz clock from external clock source.
15       sdClk_o   : out   std_logic;  -- 100 MHz clock to SDRAM.
16       sdClkFb_i : in    std_logic;  -- 100 MHz clock fed back into FPGA.
17       sdRas_bo  : out   std_logic;  -- SDRAM row address strobe.
18       sdCas_bo  : out   std_logic;  -- SDRAM column address strobe.
19       sdWe_bo   : out   std_logic;  -- SDRAM write-enable.
20       sdBs_o    : out   std_logic;  -- SDRAM bank-address.
21       sdAddr_o  : out   std_logic_vector(11 downto 0);  -- SDRAM address bus.
22       sdData_io : inout std_logic_vector(15 downto 0)   -- SDRAM data bus.
23       );
```

```
24   end entity;
25
```

In the architecture header, there's no need to add the component declarations for any of the modules because their respective VHDL packages already include this information.

Note also that several of the declared registers have been divided into two, distinct signals: one with a _r suffix that holds the current value, and another with a _x suffix that holds the signal value that will become current on the next rising clock edge. The reason for this will become clear when I discuss the FSM portion of the code.

```
26   architecture Behavioral of SdramSPInst is
27     constant NO              : std_logic := '0';
28     constant YES             : std_logic := '1';
29     constant RAM_SIZE_C      : natural   := 256; -- # of words in RAM.
30     constant RAM_WIDTH_C     : natural   := 16;  -- Width of RAM words.
31     constant MIN_ADDR_C      : natural   := 1; -- Process RAM from here
32     constant MAX_ADDR_C      : natural   := 5; -- ... to here.
33     subtype RamWord_t is unsigned(RAM_WIDTH_C-1 downto 0);  -- RAM word type.
34     signal clk_s             : std_logic;  -- Internal clock.
35     signal wr_s              : std_logic;  -- Write-enable control.
36     signal rd_s              : std_logic;  -- Read-enable control.
37     signal done_s            : std_logic;  -- R/W operation done signal.
38     signal addr_r, addr_x    : natural range 0 to RAM_SIZE_C-1; -- addr.
39     signal dataToRam_r, dataToRam_x : RamWord_t;  -- Data to write to RAM.
40     signal dataFromRam_s     : RamWord_t;  -- Data read from RAM.
41     -- Convert the busses for connection to the SDRAM controller.
42     signal addrSdram_s       : std_logic_vector(22 downto 0); -- Address.
43     signal dataToSdram_s     : std_logic_vector(sdData_io'range); -- Data.
44     signal dataFromSdram_s   : std_logic_vector(sdData_io'range);  -- Data.
45     -- FSM state.
46     type state_t is (INIT, WRITE_DATA, READ_AND_SUM_DATA, DONE); -- FSM states.
47     signal state_r, state_x          : state_t   := INIT; -- FSM state reg.
48     signal sum_r, sum_x : natural range 0 to RAM_SIZE_C * (2**RAM_WIDTH_C)-1;
49     signal sumDut_s      : std_logic_vector(15 downto 0); -- Send sum bck to PC.
50     signal nullDutOut_s : std_logic_vector(0 downto 0); -- HostIo dummy output.
51   begin
52
```

The clock generator module multiplies the 12 MHz input clock by 25/3 to create the 100 MHz clock that is output to the SDRAM. The 100 MHz clock also re-enters the FPGA and becomes the master clock for all the rest of the FPGA circuitry (line 60).

```
53   --***********************************************************************
54     -- Generate a 100 MHz clock from the 12 MHz input clock and send it out
55     -- to the SDRAM. Then feed it back in to clock the internal logic.
56     --***********************************************************************
57     Clkgen_u1 : Clkgen
58       generic map (BASE_FREQ_G => 12.0, CLK_MUL_G => 25, CLK_DIV_G => 3)
59       port map(I                => fpgaClk_i, O => sdClk_o);
60     clk_s <= sdClkFb_i;  -- SDRAM clock feeds back into the FPGA.
61
```

The SDRAM controller module is instantiated on lines 66-87. The FREQ_G generic parameter (line 68) is used to inform the module of the input clock frequency so it can accurately calculate the timing delays that the SDRAM requires. The rest of the instantiation is just the connection of I/O signals to the FSM and the external SDRAM.

```
62     --***********************************************************************
63     -- Instantiate the SDRAM controller that connects to the FSM
64     -- and interfaces to the external SDRAM chip.
65     --***********************************************************************
66     SdramCntl_u0 : SdramCntl
67       generic map(
68         FREQ_G      => 100.0,  -- Use clock freq. to calc. timing parameters.
```

```
 69           DATA_WIDTH_G => RAM_WIDTH_C  -- Width of data words.
 70           )
 71        port map(
 72          clk_i      => clk_s,
 73          -- FSM side.
 74          rd_i       => rd_s,
 75          wr_i       => wr_s,
 76          done_o     => done_s,
 77          addr_i     => addrSdram_s,
 78          data_i     => dataToSdram_s,
 79          data_o     => dataFromSdram_s,
 80          -- SDRAM side.
 81          sdRas_bo  => sdRas_bo,
 82          sdCas_bo  => sdCas_bo,
 83          sdWe_bo   => sdWe_bo,
 84          sdBs_o(0) => sdBs_o,
 85          sdAddr_o  => sdAddr_o,
 86          sdData_io => sdData_io
 87          );
 88
```

Next, the host-side address and data buses are routed to the intermediate signals the same way I did in the previous CORE Generator example.

```
 89        -- Connect the SDRAM controller signals to the FSM signals.
 90        dataToSdram_s <= std_logic_vector(dataToRam_r);
 91        dataFromRam_s <= RamWord_t(dataFromSdram_s);
 92        addrSdram_s   <= std_logic_vector(TO_UNSIGNED(addr_r, addrSdram_s'length));
 93
```

The biggest change in this example is to the FSM VHDL code. I have divided it into two, separate processes: a combinatorial process that computes the current outputs and the next state given the current inputs and state (lines 100-151), and a sequential process that updates the registers with their new values upon a rising clock edge (lines 157-165).

The combinatorial process starts with a sensitivity list. Basically, this includes every signal in the process that appears on the right-hand side of an assignment statement or in a conditional statement. If any of these signals changes value, then the entire process is re-evaluated.

```
 94        --*********************************************************************
 95        -- State machine that initializes RAM and then reads RAM to compute
 96        -- the sum of products of the RAM address and data. This section
 97        -- is combinatorial logic that sets the control bits for each state
 98        -- and determines the next state.
 99        --*********************************************************************
100        FsmComb_p : process(state_r, addr_r, dataToRam_r,
101                            sum_r, dataFromRam_s, done_s)
102        begin
```

The first thing the combinatorial process does is set the default values for various outputs and registers. These assignments will be overridden further-on in the process when a given signal needs to be changed. Note that signals with a _x suffix are always on the left-hand side of an assignment while signals with a _r suffix are on the right.

```
103        -- Disable RAM reads and writes by default.
104        rd_s        <= NO;                -- Don't write to RAM.
105        wr_s        <= NO;                -- Don't read from RAM.
106        -- Load the registers with their current values by default.
107        addr_x      <= addr_r;
108        sum_x       <= sum_r;
109        dataToRam_x <= dataToRam_r;
110        state_x     <= state_r;
```

```
111
112        case state_r is
113
```

The initialization state operates pretty much the same as always, but note how the default values of some of the signals have been overridden.

```
114          when INIT =>                        -- Initialize the FSM.
115            addr_x       <= MIN_ADDR_C;        -- Start writing data at this addr.
116            dataToRam_x <= TO_UNSIGNED(1, RAM_WIDTH_C);  -- Initial write value.
117            state_x      <= WRITE_DATA;        -- Go to next state.
118
```

When values are being written to the SDRAM, the write-enable is asserted until the done signal from the SDRAM controller goes high (line 120). Once that occurs, and before the next rising clock edge, the write-enable signal has to be disabled. This is the reason the FSM was divided into combinatorial and sequential portions. If a one-process sequential FSM like those in the previous examples was used, then the write-enable signal would not change until the next rising clock edge *after* the done signal went high, and that would be too late. With a combinatorial process, the write-enable goes low as soon as the done signal is asserted.

```
119          when WRITE_DATA =>                  -- Load RAM with values.
120            if done_s = NO then  -- While current RAM write is not complete ...
121              wr_s <= YES;                    -- keep write-enable active.
122            elsif addr_r < MAX_ADDR_C then    -- If haven't reach final address ...
123              addr_x       <= addr_r + 1;     -- go to next address ...
124              dataToRam_x <= dataToRam_r + 3; -- and write this value.
125            else                -- Else, the final address has been written ...
126              addr_x  <= MIN_ADDR_C;          -- go back to the start, ...
127              sum_x   <= 0;                   -- clear the sum-of-products, ...
128              state_x <= READ_AND_SUM_DATA;   -- and go to next state.
129            end if;
130
```

Reading values back from the SDRAM also uses the done signal to gate the read-enable signal and to accumulate to the sum-of-products only when the data becomes available (lines 134-142).

```
131          when READ_AND_SUM_DATA => -- Read RAM and sum address*data products
132            if done_s = NO then     -- While current RAM read is not complete ...
133              rd_s <= YES;          -- keep read-enable active.
134            elsif addr_r <= MAX_ADDR_C then  -- If not the final address ...
135              -- add product of previous RAM address and data read
136              -- from that address to the summation ...
137              sum_x  <= sum_r + TO_INTEGER(dataFromRam_s * addr_r);
138              addr_x <= addr_r + 1;           -- and go to next address.
139              if addr_r = MAX_ADDR_C then  -- If last addr has been read ...
140                state_x <= DONE;            -- go to the next state.
141              end if;
142            end if;
143
144          when DONE =>                        -- Summation complete ...
145            null;                            -- so wait here and do nothing.
146          when others =>                     -- Erroneous state ...
147            state_x <= INIT;                 -- so re-run the entire process.
148
149        end case;
150
151    end process;
152
```

Here's the sequential process of the FSM. On the rising clock edge, the registers get updated with the values calculated in the combinatorial process.

```
153      --*********************************************************************
```

```
154        -- Update the FSM's registers with their next values as computed by
155        -- the FSM's combinatorial section.
156        --*********************************************************************
157        FsmUpdate_p : process(clk_s)
158        begin
159          if rising_edge(clk_s) then
160            addr_r      <= addr_x;
161            dataToRam_r <= dataToRam_x;
162            state_r     <= state_x;
163            sum_r       <= sum_x;
164          end if;
165        end process;
166
```

The final thing to do is send the summation value back to the PC. Since I'm only using a single HostIoToDut module in this application, I can employ an easy-to-use version that already incorporates the BscanToHostIo module internally.

```
167        --*********************************************************************
168        -- Send the summation to the HostIoToDut module and then on to the PC.
169        --*********************************************************************
170        sumDut_s <= std_logic_vector(TO_UNSIGNED(sum_r, 16));
171        HostIoToDut_u2 : HostIoToDut
172          generic map (SIMPLE_G => true)    -- Use simple form of module.
173          port map (
174            vectorFromDut_i => sumDut_s,    -- Send sum back to PC.
175            vectorToDut_o   => nullDutOut_s -- Dummy output. Not used.
176            );
177
178    end architecture;
```

That completes the top-level module in the SdramSPInst.vhd file. Here's the SdramSPInst ISE project hierarchy:
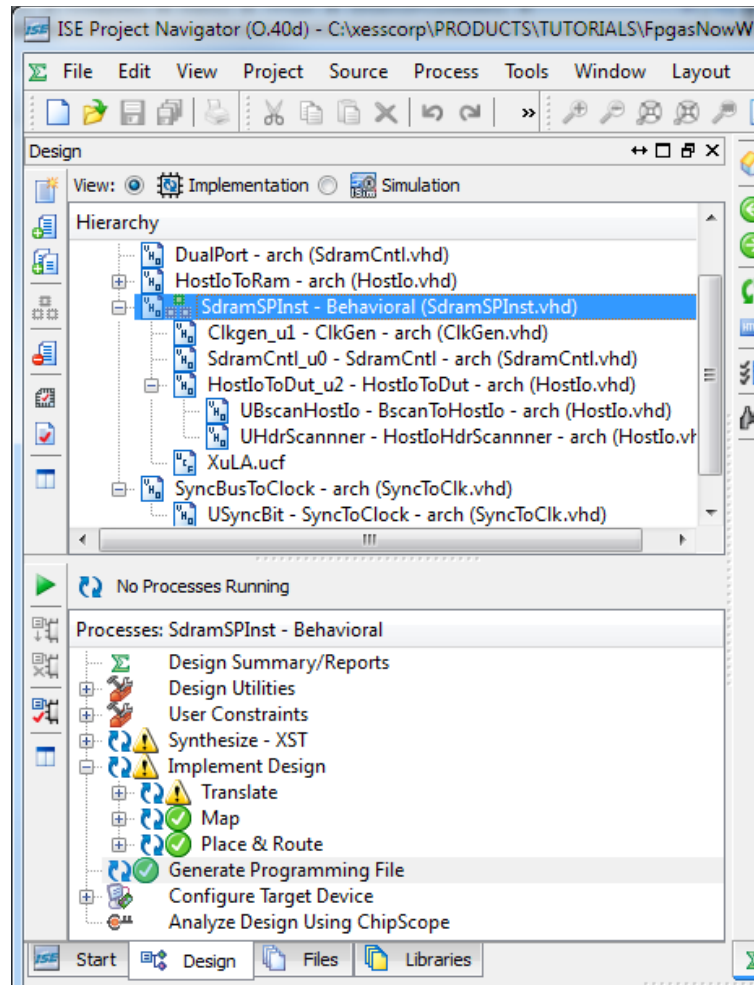
Compiling the project in the same way as all the previous examples (you should know the drill by now) generates the SdramSPInst.bit file.

Of course, to go with the FPGA bitstream, I need a Python program on the PC to grab the summation value from the XuLA board. Here's what's in the sdram_test.py file. It starts by importing all the functions and classes in the XsTools package.

```
1    from xstools.xsdutio import * # Import funcs/classes for PC <=> FPGA link.
2
```

Then the program prints out a description of what it's trying to do.

```
3    print '''\n
4    ################################################################
5    # Read the summation of the address * data products from the
6    # FSM + SDRAM application on the XuLA board.
7    ################################################################
8    '''
```

Next, the program defines two identifiers: one for the USB port index of the XuLA board (which is usually 0 because your XuLA port is usually USB0), and another for the interface identifier of the HostIoToDut module attached to the summation register in the FPGA (the identifier is 255 by default for the simple version of the module).

```
9    USB_ID = 0     # USB port index for the XuLA board connected to the host PC.
```

```
10   DUT_ID = 255  # This is the default identifier for the DUT in the FPGA.
11
```

Now create an object that lets the PC talk to the HostIoToDut module in the FPGA. The USB and DUT identifiers are needed to initialize the object. Then two arrays are needed: one which defines the size of the summation register that will be read back to the PC, and another that defines the size of the single dummy output of the HostIoToDut module. (Obviously the dummy output of the HostIoToDut module doesn't do anything; I just need at least one output to keep the VHDL synthesizer from complaining. And because the dummy output is there, I need to account for it in the Python code even though nothing is done with that, either.)

```
12   # Create an intfc obj with one 16-bit input and one 1-bit output.
13   dut = XsDut(USB_ID, DUT_ID, [16], [1])
```
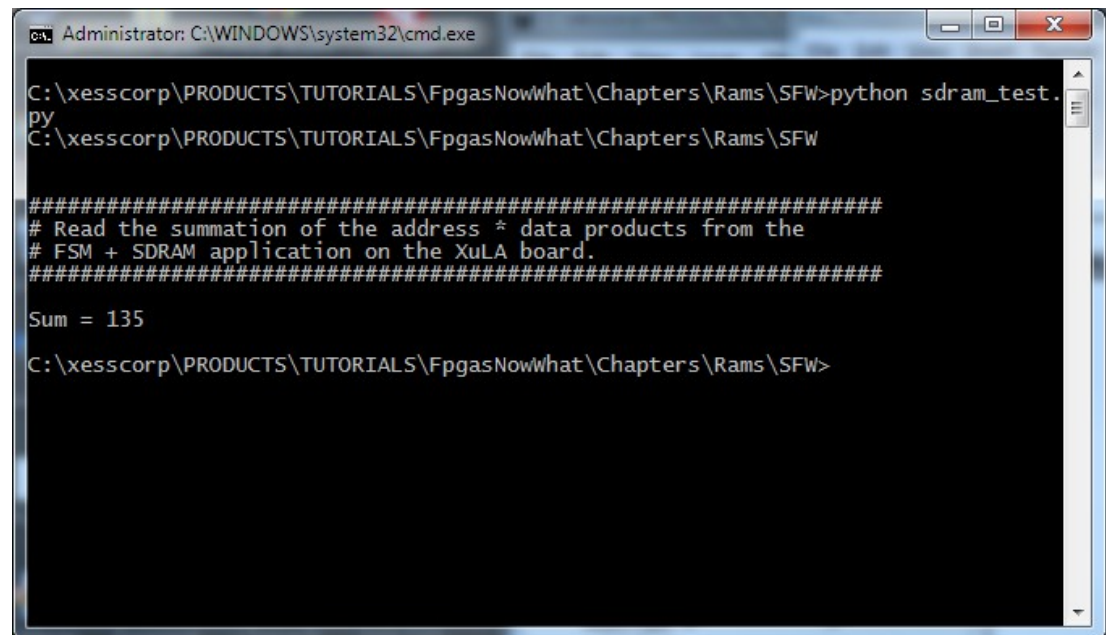
Finally, use the Read() method of the dut object to get the value in the summation register. Then print the value of the sum and exit.

```
14   sum = dut.Read(); # Read the 16-bit summation from the FPGA.
15   print 'Sum = %d\r' % sum.unsigned
```

At this point, I assume you have the SdramSPInst ISE project compiled and the bitstream is ready to go, the firmware in your XuLA board has been upgraded to support the HostIoPckg modules (you should have already done that in the previous chapter), and you have Python installed and the sdram_test.py file is available. Now all you have to do is this:

1. Download the SdramSPInst.bit file to the FPGA on the XuLA board using GXSLOAD.

2. Run the Python program in a command window as follows:
    `C:\SFW> python sdram_test.py`

Then you should see the following text appear in the command window:

The value in the sum register matches the results found in the previous examples, so the SDRAM circuitry appears to be working.

Now you might ask (if you're still there): "Why did you go to all the trouble of running this on the actual hardware instead of doing a simulation like in the other examples?" The answer to that is simple: the ISE simulator knows how to model the behavior of the distributed and block RAMs built into the FPGA itself, but it has no built-in model for the external SDRAM chip. So I just went straight to the hardware and prayed that it worked.

The problem with doing it in hardware is that if the result is incorrect, then how do you find where the problem is? Is it in the FSM? Is the SDRAM controller being operated in the correct manner? Is the problem *inside* the SDRAM controller itself? (If so, take a look in the SdramCntl.vhd file and tell me you would want to debug that! I did back in 2001 after I wrote it and believe me, it was not pretty.)

So, while throwing a "Hail Mary" with the hardware sometimes works, we really need a backup plan to debug systems that extend beyond the boundaries of the FPGA. That's what I'll show you in the next chapter.

C.8 *"Verilog! Now What!?"*

Back in the 80's, there was a TV commercial for Reeses Peanut Butter Cups that went something like this. Some dude is bebopping down the sidewalk on his skateboard eating from a jar of peanut butter. Another dork is riding down the sidewalk on a bike eating a chocolate bar. They don't see each other as they arrive at an intersection (probably because THEY ARE TOO BUSY EATING!) and a horrendous crash occurs. The first guy is lying on the ground and says "Hey, you got chocolate in my peanut butter!" The second guy retorts "Oh, yeah? You got peanut butter on my chocolate!". (Nowadays the conversation would go something like this: "I'm going to sue you 'til your eyeballs explode!"; "Yeah? Say hello to my 9mm.") Well, they each try the mixed concoction and arrive at the shared realization that chocolate and peanut butter taste pretty good together. Then they gaze into each other's eyes, join hands, and go off to start a B&B in the Napa Valley wine country.

So what does this have to do with using FPGAs? Not much, really; I just thought it was funny. But it is kind of similar to how VHDL and Verilog users view their favorite language as superior and it shouldn't be mixed with the other when, in reality, sometimes a combination of them is unavoidable and desirable. I'll demonstrate this with a simulation of the FPGA sum-of-products circuit and the external SDRAM from the last chapter.

Simulating SDRAM

As I said at the end of the last chapter, ISE knows how to simulate the internal RAMs of the FPGA, but it doesn't know jack about the behavior the external SDRAM on the XuLA board – it doesn't have those models. So to run a simulation, we need a VHDL model of the XuLA's Winbond W9812 8M x 16 SDRAM to combine with the VHDL for the FPGA circuit. Unfortunately, I've never been able to find a VHDL model of the W9812. [Winbond does supply a Verilog model of the W9812](#), but it's encrypted so it will only work with the NC-Verilog simulator. That's not much help.

So there's no VHDL model for the W9812, and there's no vanilla-Verilog version, either. Luckily, the free market works in my favor here. All memory vendors are competing for the same slots in PCs, tablets and phones, so they all make their chips interoperable to try to steal business from each other. That means we can use an SDRAM model from another manufacturer in place of the one from Winbond. In this case, Micron supplies a Verilog model for their equivalent 8M x 16 SDRAM, the MT48LC8M16A2: http://www.micron.com/parts/dram/sdram/mt48lc8m16a2p-7e?pc={428A5CC9-2A78-447E-939B-6F3A40D538C6}#sim.

After downloading the archive from Micron, I just unpack it and store the mt48lcm16a2.v Verilog file in the folder for the SdramSPInst ISE project I built in the last chapter. Then I reopen the project in ISE and select the **Add Source...** item from the pop-up menu.
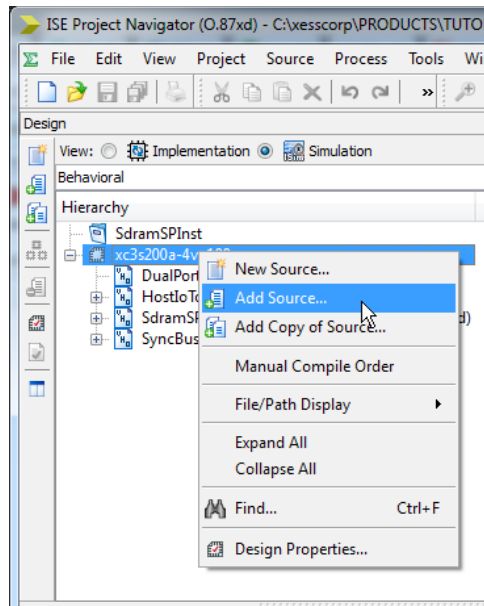

Once the dialog for selecting the source file appears, I highlight the mt48lcm16a2.v file and click **Open** which should add the Verilog file to my previously pristine VHDL project.
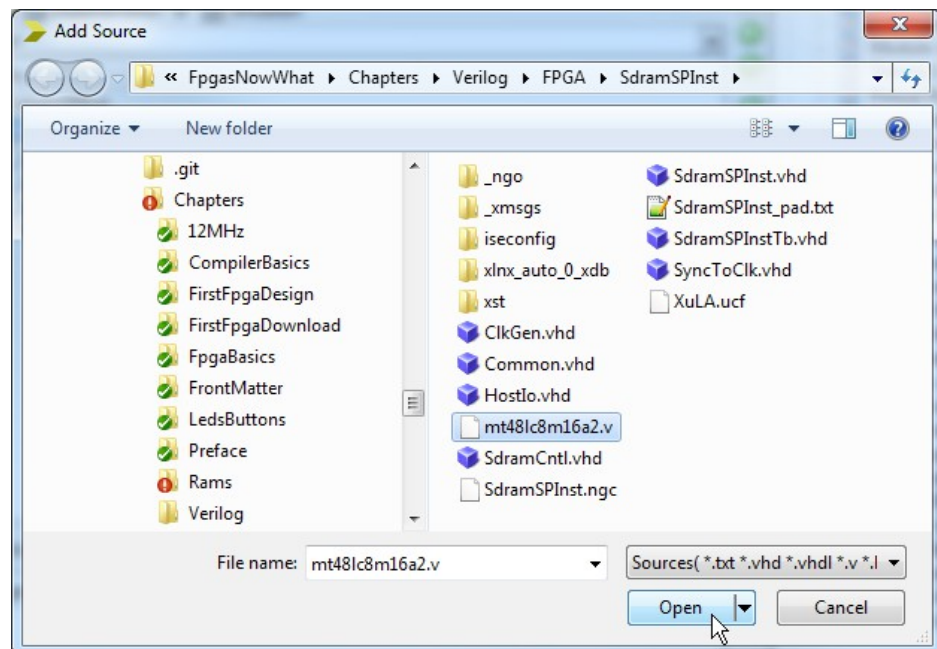
When adding the SDRAM model, I make sure to *only associate the SDRAM Verilog file with simulation processes!* That's because the SDRAM model contains stuff that's unsynthesizable or, at least, very inefficient to synthesize. I don't want ISE trying to build the SDRAM inside the FPGA if I run the implementation tools. That would only lead to tears.
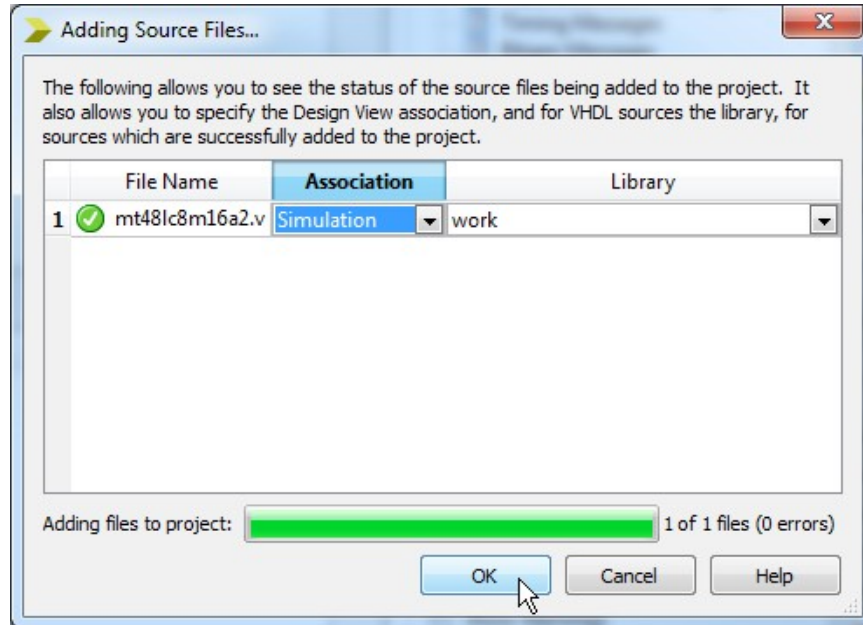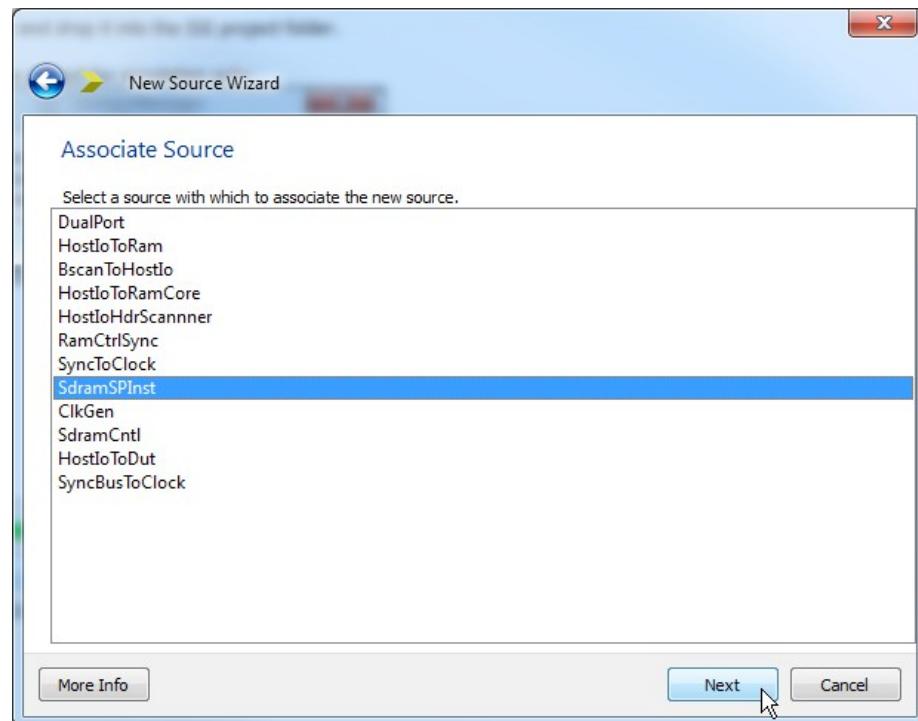

Now it's time to create a simulation test bench, so I do the whole **New Source...** thing.
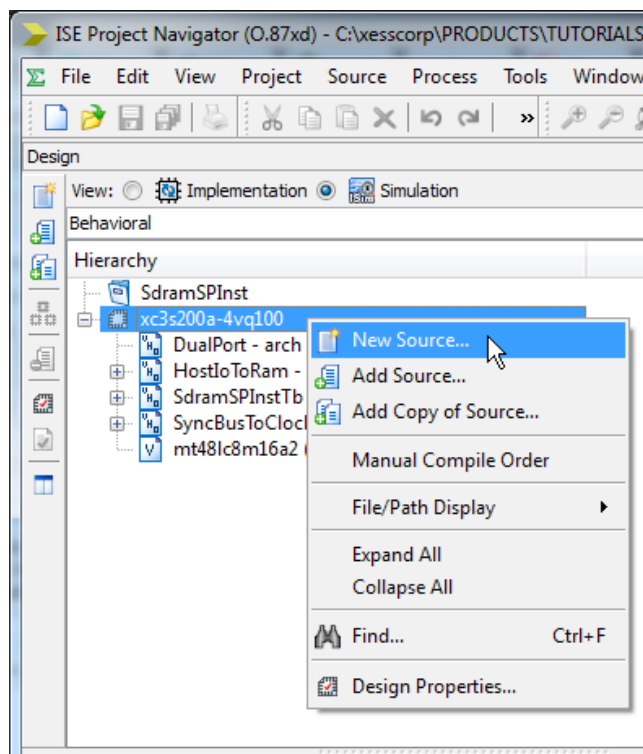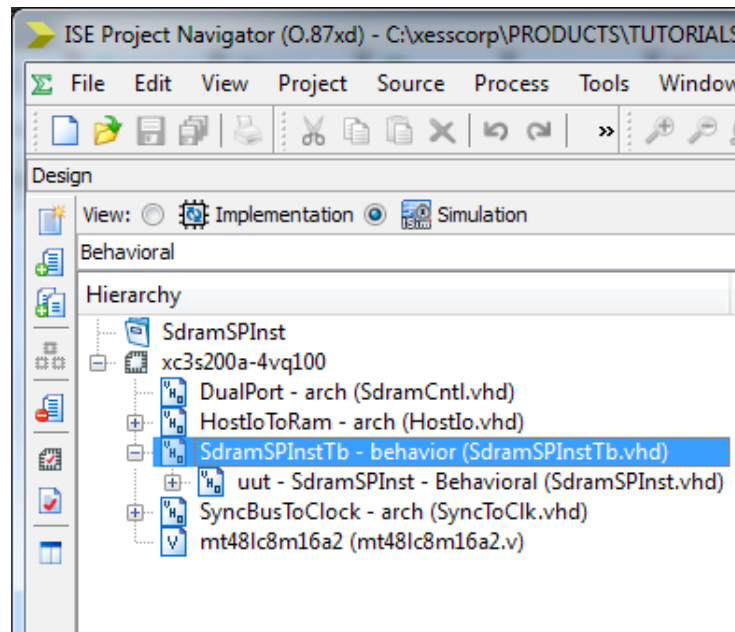
Then I add a VHDL test bench source file called SdramSPInstTb.
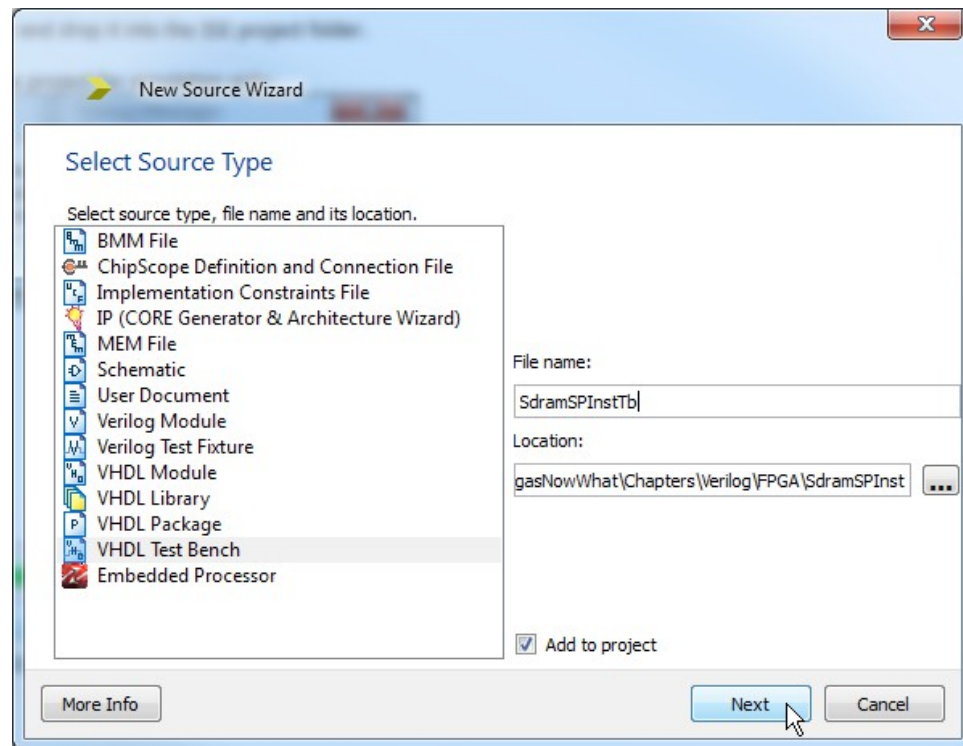
Next I associate the test bench with the SdramSPInst.vhd file containing the sum-of-products module.

Now the test bench appears at the top of the design hierarchy with the original SdramSPInst as a submodule.

Note that the SDRAM Verilog model is outside the test bench hierarchy, so my next task is to integrate that into the mix by attaching it to the sum-of-products module within the SdramSPInstTb.vhd file. In effect, the test bench file emulates the XuLA board because it describes how an FPGA programmed with a sum-of-products function is connected to an external SDRAM. But to make this connection, I have to figure out the I/O definitions for the SDRAM from the Verilog source. Here is the pertinent portion of the mt48lc8m16a2.v file:

```
42   module mt48lc8m16a2 (Dq, Addr, Ba, Clk, Cke, Cs_n, Ras_n, Cas_n, We_n, Dqm);
43
44      parameter addr_bits =      12;
45      parameter data_bits =      16;
46      parameter col_bits  =       9;
47      parameter mem_sizes = 2097151;
48
49      inout     [data_bits - 1 : 0] Dq;
50      input     [addr_bits - 1 : 0] Addr;
51      input                [1 : 0] Ba;
52      input                        Clk;
53      input                        Cke;
54      input                        Cs_n;
55      input                        Ras_n;
56      input                        Cas_n;
57      input                        We_n;
58      input                [1 : 0] Dqm;
```

Using this information, I can go into the SdramSPInstTb.vhd file and add an equivalent VHDL component declaration that uses the same module name with the same I/O names, in the same order, and with the same vector widths as follows:

```
60   component mt48lc8m16a2
61   port(
62     Dq : inout std_logic_vector(15 downto 0);
63     Addr : in std_logic_vector(11 downto 0);
64     Ba : in std_logic_vector(1 downto 0);
```

```
65      Clk : in std_logic;
66      Cke : in std_logic;
67      Cs_n : in std_logic;
68      Ras_n : in std_logic;
69      Cas_n : in std_logic;
70      We_n : in std_logic;
71      Dqm : in std_logic_vector(1 downto 0)
72      );
73   end component;
```

Then I instantiate and connect the mt48lc8m16a2 module to the SdramSPInst module like so:

```
97           -- Instantiate the Unit Under Test (UUT)
98    uut: SdramSPInst PORT MAP (
99            fpgaClk_i => fpgaClk_i, -- 12 MHz XuLA clock.
100           sdClk_o => sdClk_o, -- 100 MHz clock from DCM.
101           sdClkFb_i => sdClkFb_i, -- 100 MHz clock fed back into FPGA.
102           sdRas_bo => sdRas_bo, -- Row-address strobe.
103           sdCas_bo => sdCas_bo, -- Column-address strobe.
104           sdWe_bo => sdWe_bo, -- Write-enable.
105           sdBs_o => sdBs_o, -- Bank-select.
106           sdAddr_o => sdAddr_o, -- 12-bit address bus.
107           sdData_io => sdData_io -- 16-bit data bus.
108         );
109    sdClkFb_i <= sdClk_o; -- Feedback 100 MHz clock to FPGA.
110
111     -- Use the mt48lc8m16a2 declaration from above to instantiate
112     -- the SDRAM here and connect it to the UUT.
113     sdram: mt48lc8m16a2 port map(
114       Dq => sdData_io,   -- 16-bit data bus.
115       Addr => sdAddr_o,  -- 12-bit address bus.
116       Ba(0) => sdBs_o,   -- One bank-select pin.
117       Ba(1) => '0', -- The other is tied to GND on XuLA PCB.
118       Clk => sdClk_o,   -- 100 MHz clock.
119       Cke => '1',   -- Clock-enable tied high on XuLA PCB.
120       Cs_n => '0', -- Chip-enable tied low on XuLA PCB.
121       Ras_n => sdRas_bo, -- Row-address strobe.
122       Cas_n => sdCas_bo, -- Column-address strobe.
123       We_n => sdWe_bo, -- Write-enable.
124       Dqm(0) => '0', -- Data qualifier masks tied low ...
125       Dqm(1) => '0'  -- on the XuLA PCB.
126         );
```

Finally, I define the clock period to match the 12 MHz clock of the XuLA board:

```
93   constant fpgaClk_period : time := 83.3333 ns; -- 12 MHz XuLA clock.
```
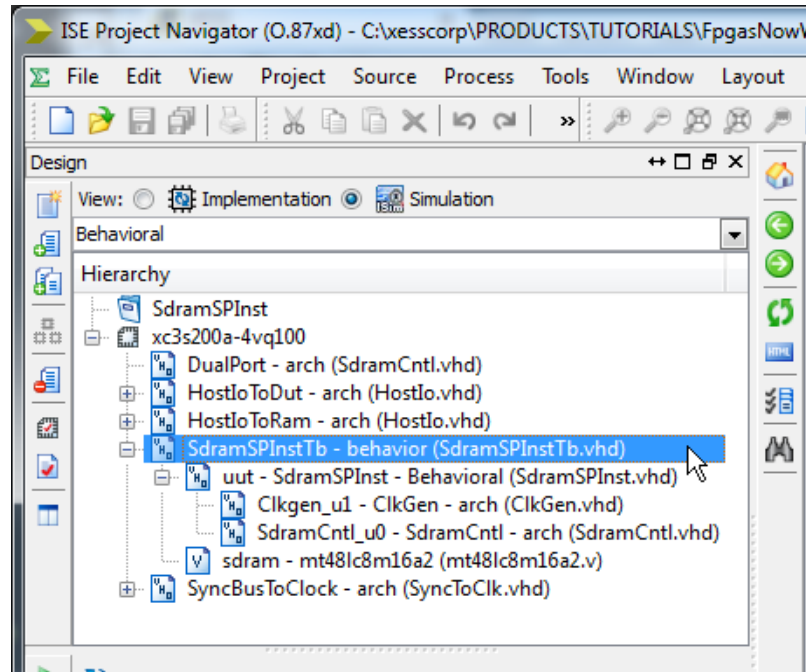
Then I place the clock signal name into the process that generates the clock:

```
130  fpgaClk_process :process
131  begin
132    fpgaClk_i <= '0';
133    wait for fpgaClk_period/2;
134    fpgaClk_i <= '1';
135    wait for fpgaClk_period/2;
136  end process;
```

After making these changes, I'm rewarded by seeing the inclusion of the Micron SDRAM as a submodule of the test bench:


Now it's time to run the simulation. I highlight the SdramSPInstTb test bench module in the **Hierarchy** pane and double-click the **Simulate Behavioral Model** process in the **Processes** pane. The **ISim** window will appear. I type 202 uS into the simulation run-time field and click on the button to run the simulation. (The reason for the long simulation run-time is that the SDRAM has a 200uS initialization delay before it can perform any read/write operations.)
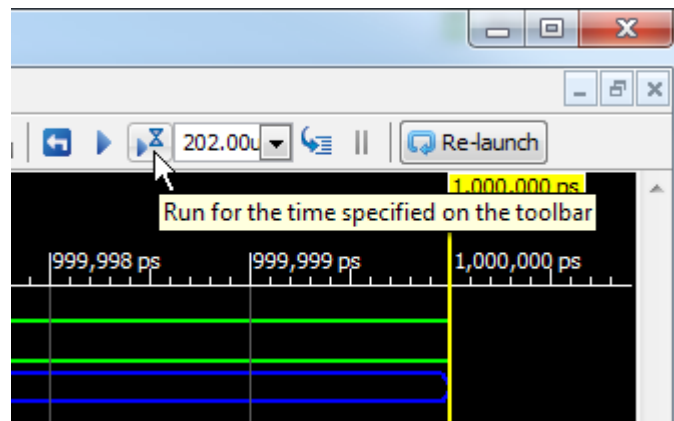

Looking at the simulation waveforms after the initialization period, you can see the five write operations as the sum-of-products circuit loads the SDRAM with data: 1 is written to address 1, 4 to address 2, 7 to address 3, and so on as in the example from the previous chapter.
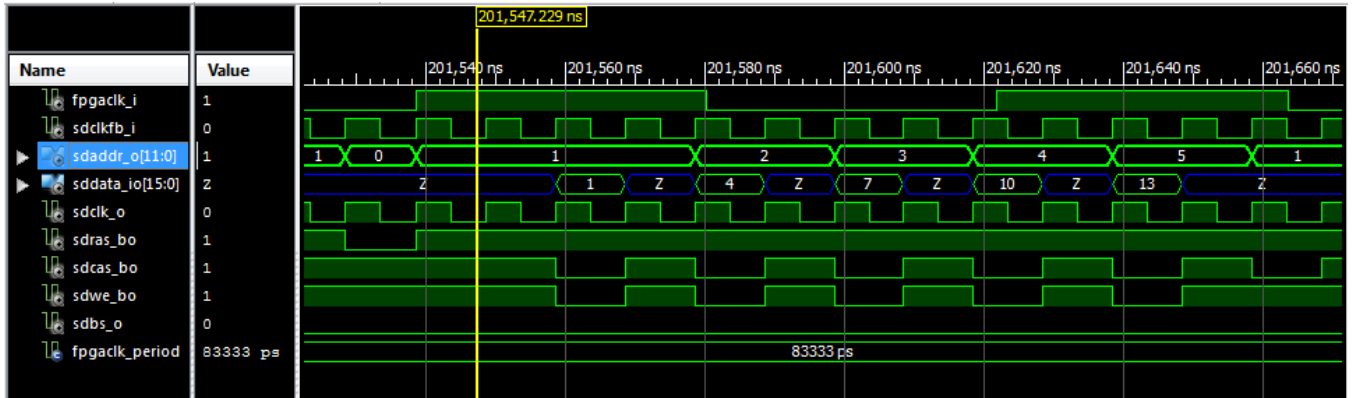
After the writes, the data is read back from the SDRAM. Notice that 1 is read from address 1, 4 from address 2, and so on. It appears the SDRAM Verilog model is working correctly for the basic read and write functions.
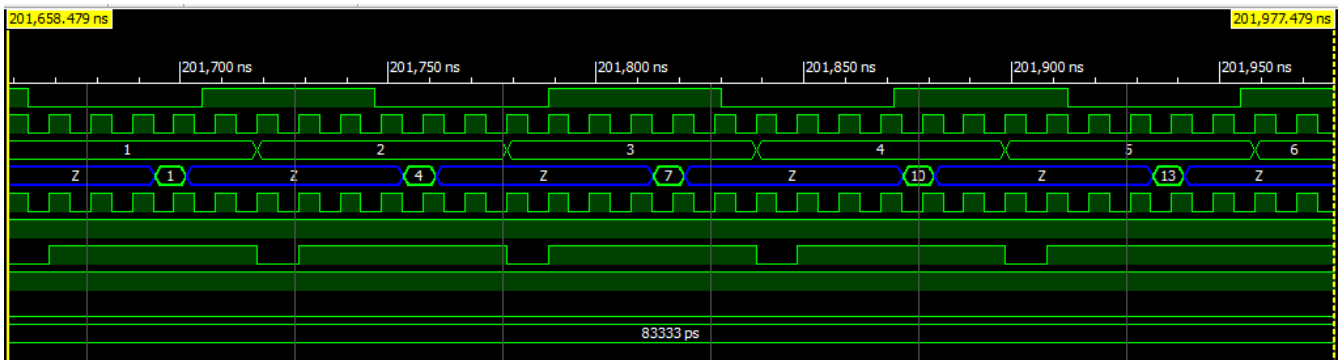
The final test of the SDRAM model is whether the computed sum-of-products value is correct. To check this, I highlight the **uut** instance in the **Instance and Process Name** pane and then look for the **sum_r** register in the **Objects** pane. There I see that the sum register contains the value of 135 at the end of the simulation. This matches the result I saw in the SDRAM example of the last chapter.
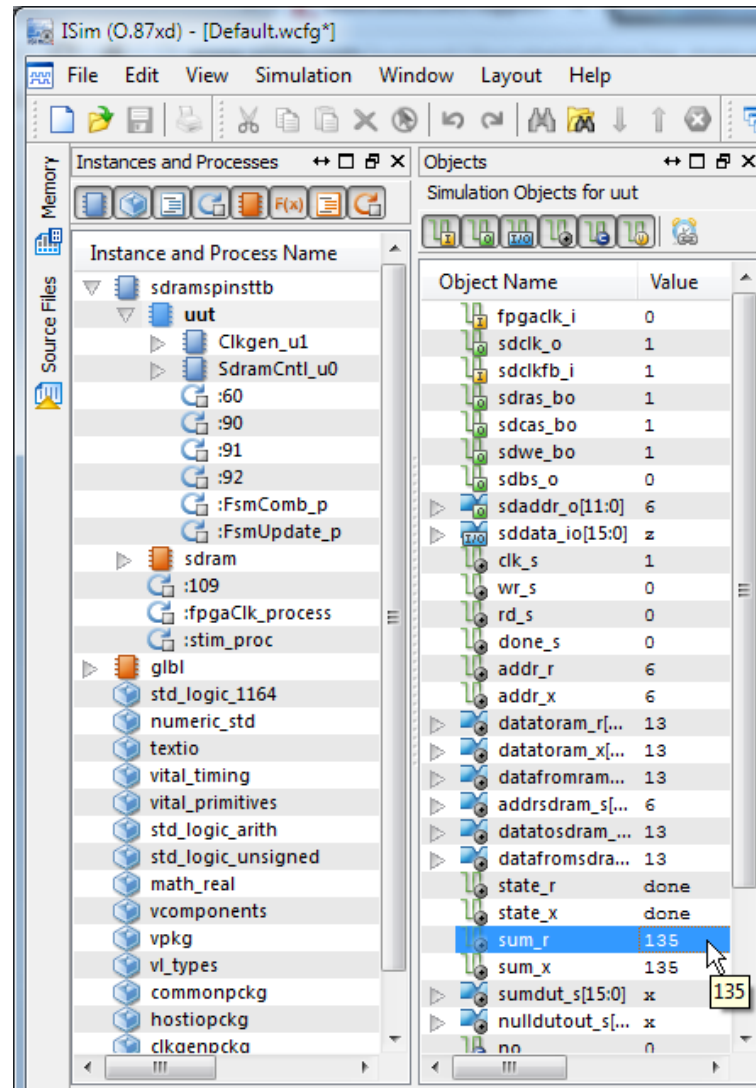
So to recap, here's what I did:

1. Found a file containing a Verilog model for some circuit.

2. Added the file to my project and associated it with simulation processes only.

3. Opened the Verilog file and found the I/O declarations for the model.

4. Created a component declaration in my VHDL code that has the same module name with the same I/O names, in the same order, and with the same vector widths.

5. Instantiated and connected the module wherever it was needed just as if it was another VHDL component.

The example I've just described was done for simulation purposes. Is there any difference if I had wanted to include a Verilog module for actual implementation in my FPGA? Not

really, except in that case I would want to *associate that module with both simulation and implementation processes*, rather than just simulation alone. Everything else would stay the same.