Xport 2.0 Custom Configuration Tutorial

Version 2.3, June 29, 2004

Rich LeGrand (rich@charmedlabs.com)

Summary

This application note/tutorial explains how obtain and install the Xilinx WebPACK or ISE software. Once installed, we write and synthesize a custom logic configuration for the Xport 2.0 called "RedGreen1". Sample C++ code is also written and executed from the GBA to test the RedGreen1 logic configuration.

Introduction

It is likely that you may want to use the Xport in a way that no one has thought of before. In which case you will probably want to create your own custom logic configuration. If so, we have good news – it's probably much easier than you think. Synthesizing logic these days is as easy as programming once you pick up a few simple concepts. Personally, I've spent considerable time in the past soldering together circuits with dozens of chips that often took days to design, prototype and debug. Looking back, it is clear that I wasted lots of time – if I had used FPGA logic instead, my designs would have been much quicker to prototype, smaller, faster, simpler to modify and more robust.

The ability to synthesize custom logic can dramatically change the way an engineer solves problems. PWM channels, UARTs and other features that are often implemented with software, can perform much better and require zero CPU overhead when implemented with custom logic. In other words, you are free to make the classic hardware-software tradeoff decisions that were previously only available to companies with large design budgets.

This tutorial will step through a simple custom logic example for the Xport 2.0 using the Xilinx ISE or WebPACK 5.2i, which is available free through the Xilinx website. It is assumed that you have programming experience, but not necessarily with HDLs (hardware description languages). We will be using Verilog, and if you have never seen Verilog code, you should still be able to follow this tutorial. If you are experienced with HDLs you can still benefit by learning the design flow of a custom Xport logic configuration.

Verilog or VHDL?

We tend to think that Verilog is easier to learn than VHDL because of its C-like syntax and smaller intended scope. For these reasons the HDL code in this tutorial is written in Verilog. If you are familiar with VHDL, you should have little trouble converting the example code into equivalent VHDL code.

However, we do not recommend using "schematic capture" for designing logic configurations. While it may be more intuitive for some designers, it can take much more time to input the design, and it usually forces undesired constraints on your design. HDLs tend to be much more expressive and flexible.

Downloading and Installing the Xilinx WebPACK

The WebPACK is a full-featured FPGA design and synthesis package that is available for free at:

http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?sGlobalNavPick=PRODUCTS&sSecondaryNavPick=Design+Tools&key=DS-ISE-WEBPACK

From this page you must fill out the registration before proceeding to the download page. When downloading, choose the "Xilinx WebInstall", which installs the complete package and applies the latest patches automatically. During the installation setup, you will be asked to select which modules you would like installed. Only the "FPGA Design Environment" module is required, and the following two modules are recommended:

- Xpower and Chip Viewer Tools
- Documentation

The CPLD tools are not necessary nor are the parallel port drivers**. It is strongly recommended that you DO NOT install the MultiLINX drivers**. These are not needed for the Xport, and we have found that they can adversely affect your computer's USB operation.

At the time of this writing, the latest WebPACK package is version 6.2i.

The "RedGreen1" Configuration

For illustrative purposes we will create a very simple configuration called "RedGreen1" which will allow a program running from the GBA to control the red and green LEDs found on the Xport 2.0. After creating the configuration we will write a test program in C++ that illuminates the red and green LEDs when the "A" and "B" buttons on the GBA are depressed. The complete RedGreen1 example with source code and precompiled binaries is located in $XPORTDIR/examples/redgreen1. But for illustrative purposes, we will create a new project from scratch and copy source from the redgreen1 directory when necessary.

You Will Need

- ✓ PC with Xport software installed (Release 2.0.23 or later)
- ✓ Xilinx WebPACK software
- ✓ Xport 2.0, GBA, parallel port interface and programming cable

Creating and Synthesizing RedGreen

Start Xilinx WebPACK by running the "Project Navigator". The shortcut should be installed on your desktop or it can be found in the Start menu (Start→Programs→Xilinx ISE). When the navigator comes up, create a new project by selecting "New Project" from the "File" pulldown menu. **Figure 1** shows the expected "New Project" dialog.

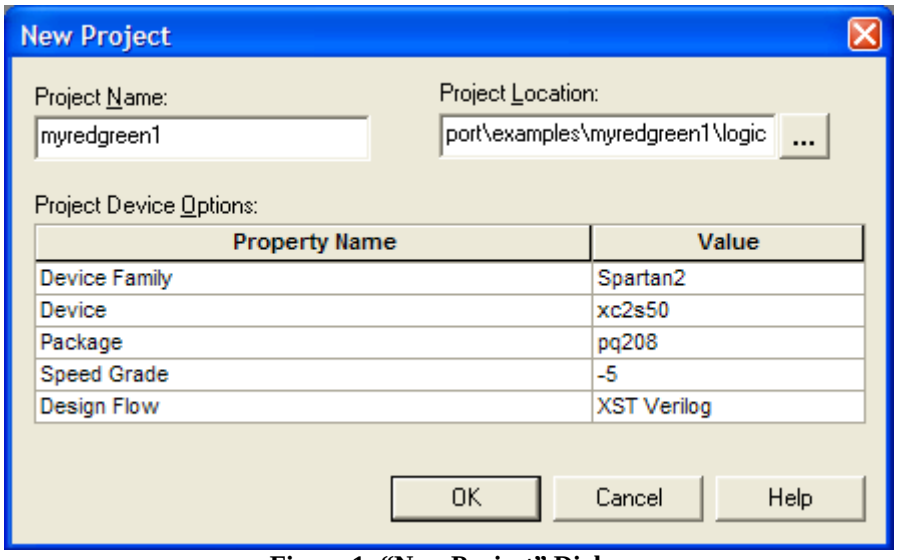

Figure 1: "New Project" Dialog

Type in "myredgreen1", for example, in the "Project Name" box. In the "Project Location" box, a good place is the Xport "examples" directory (e.g. C:\Xport\examples\myredgreen1\logic). Note, creating a separate "logic" subdirectory separates the C++ code (which we will write) from the HDL code and associated project files. The "New Project" dialog also has properties that need to be selected. Go ahead and select them as follows:

- ✓ Device Family: Spartan2

- ✓ Device: xc2s50
- ✓ Package: pq208
- ✓ Speed Grade: -5
- ✓ Design Flow: XST Verilog

and click "OK". **If you are using an Xport with 150K gate FPGA, then choose the xc2s150 device.**

Instead of typing the Verilog code by hand, we are going to be a little lazy and copy the source file from the existing RedGreen1 example directory. That is, copy the redgreen1.v file from $XPORTDIR/examples/redgreen1/logic to your new redgreen1 project directory. Now add redgreen1.v to your new project by selecting "Add Source" from the "Source" pulldown menu. Select the redgreen1.v file that you just copied from the dialog and click "Open".

After the source file is added, your "Sources in Project" pane should resemble **Figure 2**.

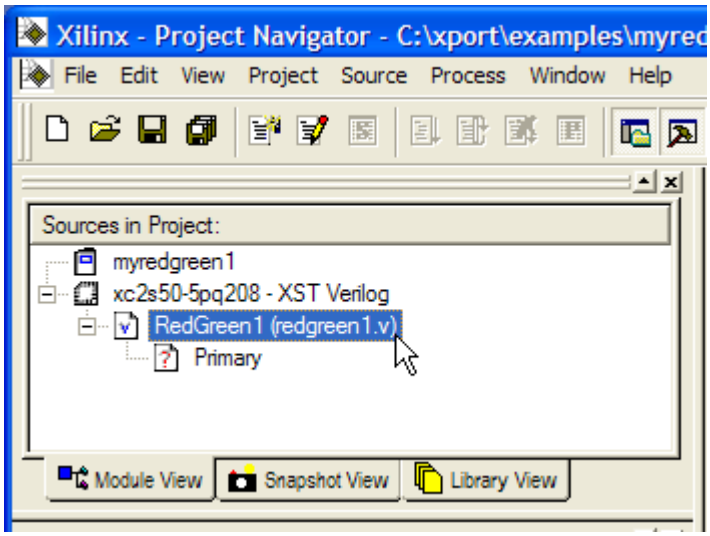

Figure 2: "Sources in Project" Pane

You can go ahead and view redgreen1.v by double-clicking on redgreen1.v (shown highlighted in **Figure 2**). If you have never seen Verilog code before, you will notice that the syntax is vaguely C-like. And you can probably guess, based on the length of the file, that whatever redgreen1.v does, it isn't very complex. And your hypothesis would be correct. But to gain some insight, we need to go over the contents of this file.

The first part of the file declares what we call the "Secondary" module. The Secondary is the application-specific code that implements the functionality that we need for our application. Here, we are interested in turning some LEDs on and off, so our code is relatively small and simple. The Secondary module declaration simply defines the lists the ports we will be using. These ports correspond directly to physical I/O pins on the Xport FPGA.

```
module RedGreen1(CartData, CartAddr, FData, FAddr, CartCs, CartRd, CartWr, CartIReq, FCe, FOe, FWe,
   PA, PB, ClkInA, ClkInB, GreenLED, RedLED, RCs, Clk);
```

The next section instantiates what we call the "Primary" module.

```
 Primary InstPrimary(.CartData(CartData), .CartAddr(CartAddr),
      .CartCs(CartCs), .CartRd(CartRd), .CartWr(CartWr), .CartIReq(CartIReq),
      .FData(FData), .FAddr(FAddr), .FCe(FCe), .FOe(FOe), .FWe(FWe),
      .Addr(Addr), .Rd(Rd), .Wr(Wr), .SecDataRd(DataRd), .Clk(Clk));
```

The Primary contains all of the "primary" (i.e. necessary) functions that most logic configurations will probably need. Inside the Primary, for example, is code that demultiplexes the GBA cartridge bus into useable address and data signals. The Primary that we will be using also contains logic for Cport communication, synchronous power-up reset, configuration identification,

and LED control (we'll pretend this doesn't exist for now!) Since our example is simple, we only need the GBA cartridge demultiplexing contained in the Primary.

The Primary is included with the include statement at the top of redgreen1.v. This source file can be found in $XPORTDIR/logic/src/primary.v, although we will not cover it in detail here. One of ideas behind the Primary module is that you are free to customize it based on the functionality you consider to be "primary" for your application(s). But regardless of whether it is our primary or your own, you need to instantiate some form of Primary or the GBA software will not be able to execute because of the lack of cartridge demultiplexing.

Note, the primary instantiation uses dot notation to pass signals. This prevents signals from being accidentally misassigned because of the possibility that they were passed in the wrong order. The demultiplexed cartridge bus signals that the Primary provides (and that we will be using) are detailed below in **Table 1**:

Table 1: Primary bus signals

Signal	Direction*	Description
Addr	Output	Demultiplexed 24-bit address from the GBA cartridge port. The cartridge port is mapped from GBA location 0x8000000 through 0x9ffffff (32Mbytes) Accesses that occur outside this area are not presented to the cartridge and not seen by the FPGA logic. Each address corresponds to a 16-bit memory location, which is why there are only 24 address bits (instead of 25) to span the 32Mbyte cartridge address space.
Wr	Output	Asserted HIGH when GBA is writing within 0x8000000 through 0x9ffffff address space.
CartData	Output	16-bit data from the GBA. CartData is valid while Wr signal is high.
Rd	Output	Asserted HIGH when GBA is reading within 0x8000000 through 0x9ffffff address space.
SecDataRd	Input	16-bit data to the GBA. Data presented to this port should be valid while Rd signal is high

*Direction is with respect to the Primary module. For example, Addr is an *output* from Primary.

After the Primary instantiation, the rest of the redgreen1.v file is dedicated to application-specific code. In order to control the LEDs from our C++ code, we need to create a register (LEDReg) in the cartridge address space. Each bit in this register will correspond to the state of an LED (1 or 0, on or off). Since there are only two LEDs (red and green), we only need 2 bits for our register. We first create the signal LEDEn, which we use to map a register location within the cartridge address space.

```
assign LEDEn = Addr[23:8]==16'hffe2;
```

Here, we have chosen 0xffe200 (the two zeros are implied since we are only looking at the upper 16 address bits). To calculate the actual address used by the GBA software, take this value, multiply it by 2 and add 0x8000000 to get 0x9ffc400. This address will be used by our C++ example software to control the LEDs.

Next, we handle write operations to our LED register (LEDReg).

```
always @(negedge Wr)
   begin
   if (LEDEn)
      LEDReg <= CartData[1:0];
   end
```

Here we wait until the trailing edge of the Wr signal and then latch the GBA data onto LEDReg, but only if the address is valid (LEDEn is asserted).

Reading the state of LEDReg is not absolutely necessary (we are only interested in controlling the LEDs), but it's a nice feature to have. To handle read operations to LEDReg, we simply need to present the contents of LEDReg to DataRd when LEDEn is asserted. Note, DataRd gets passed to the Primary as the SecDataRd signal (see **Table 1**.)

```
always @(LEDEn or LEDReg)
   begin
   if (LEDEn)
      DataRd = {14'b00000000000000, LEDReg[1:0]};
   else
      DataRd = 16'hxxxx;
```

```
    end
```

Here, we have implemented the beginnings of multiplexer logic for DataRd. Although multiplexing one signal (LEDReg) onto DataRd does not result in multiplexer logic being instantiated, it does allow future signals to be multiplexed easily. Note that the multiplexer is not dependent on the Rd signal, which you might expect. The Primary module takes care of the Rd signal synchronization for you – the Secondary (your own code) only needs to implement multiplexer logic.

Lastly, we assign the LED signals to the corresponding states of LEDReg. Since the LED signals are low-going (low=on), we invert the register bits.

```
assign RedLED = ~LEDReg[0];
assign GreenLED = ~LEDReg[1];
```

Synthesizing

We are now ready to synthesize our design, but before we do so, we must tell the Xilinx synthesizer where to find the Primary code (primary.v). This requires that we change the property preferences to advanced. From the "Edit" pulldown menu in the title bar, select "Preferences". From the "Preferences" dialog, select the "Processes" tab and "Advanced" for the "Property Display Level" as shown in **Figure 3**. The Xilinx ISE will remember your advanced property preferences so you will not have to select this in the future.

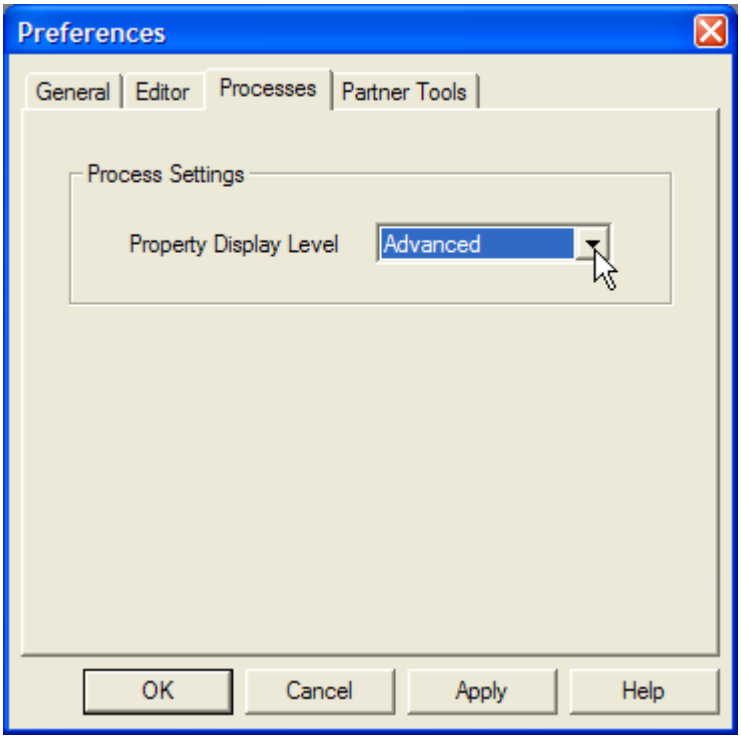

Figure 3: Advanced property preferences

Next, from the "Processes for Current Source" pane (shown in **Figure 4**) right-click on "Synthesize" and select "Properties..." to bring up the synthesis "Process Properties" dialog as shown in **Figure 5**.

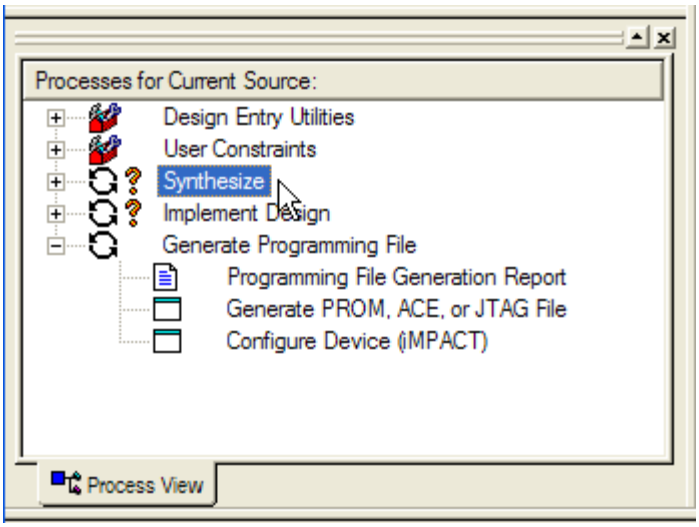

Figure 4: "Processes for the Current Source" Pane

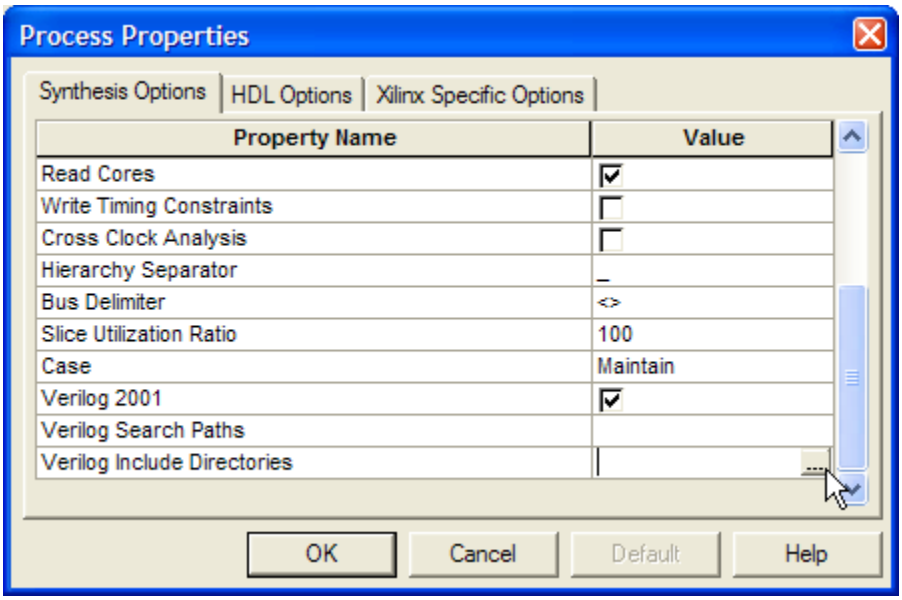

Figure 5: Synthesis "Process Properties" Dialog

Scroll down to the "Verilog Include Directories" option and click on the browse button as shown in **Figure 5**. From the file browser, browse to the logic/src directory (typically located in C:/xport/logic/src) and click "OK". Now double-click on "Synthesize" in the "Processes for the Current Source" pane (**Figure 4**). This will launch the synthesize process. When it finishes the Processes pane will resemble **Figure 6**.

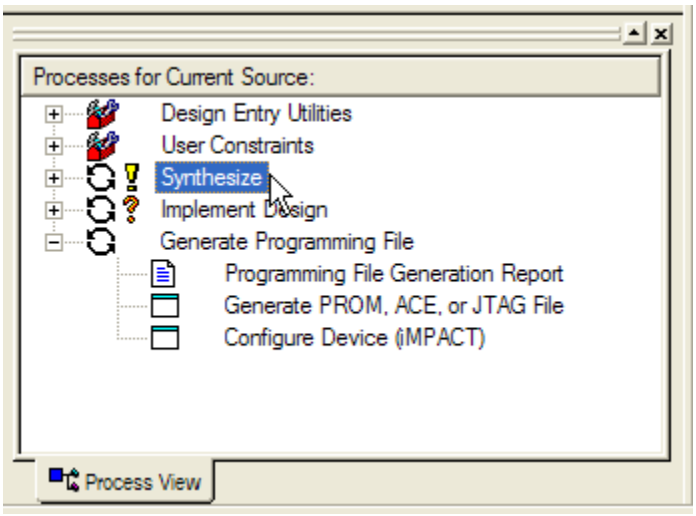

Figure 6: Synthesis Complete

The yellow "!" indicates that one or more warnings were generated. The warnings in this case can be safely ignored.

Implementation

After synthesizing, we can target our design to the Xport FPGA (a process called implementation). But before we do so, we have to add the pin assignment/constraints file to our project. Select "Add Source" from the "Project" pulldown menu in the top toolbar and browse to the logic/src/xport.ucf file (typically located at C:/xport/logic/src/xport.ucf) select it, and click on "Open" (**Figure 7**).

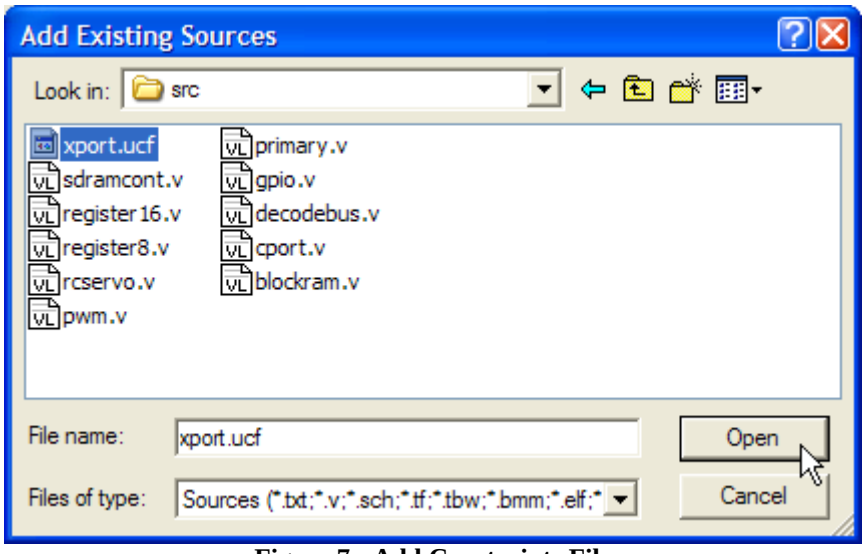

Figure 7: Add Constraints File

After adding the constraints file, it will be highlighted in the "Sources in Project" pane. Re-select the redgreen1.v file as shown in **Figure 8**. (Note that the "Processes for Current Source" pane is context sensitive based on the selected source file in the "Sources in Project" pane.)

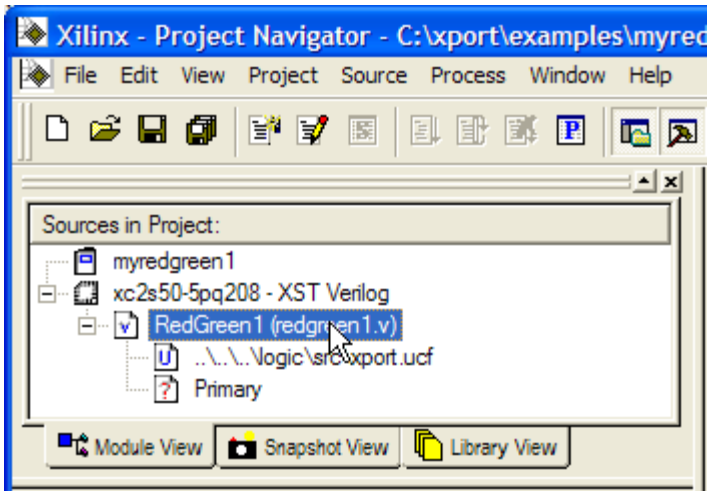

Figure 8: Relesect redgreen1.v

We also have to tell the implementation utility where to find the precompiled modules or "macros". For example, the GBA cartridge demultiplexing module is provided in macro form. Right click on "Implement Design" in the "Processes for Current Source" pane (**Figure 9**) and select "Properties..." to bring up the implementation "Process Properties" dialog.

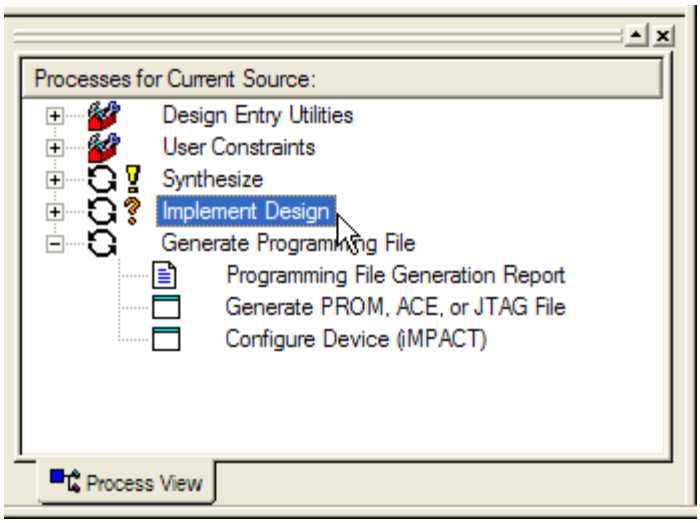

Figure 9: Right Click on "Implement Design"

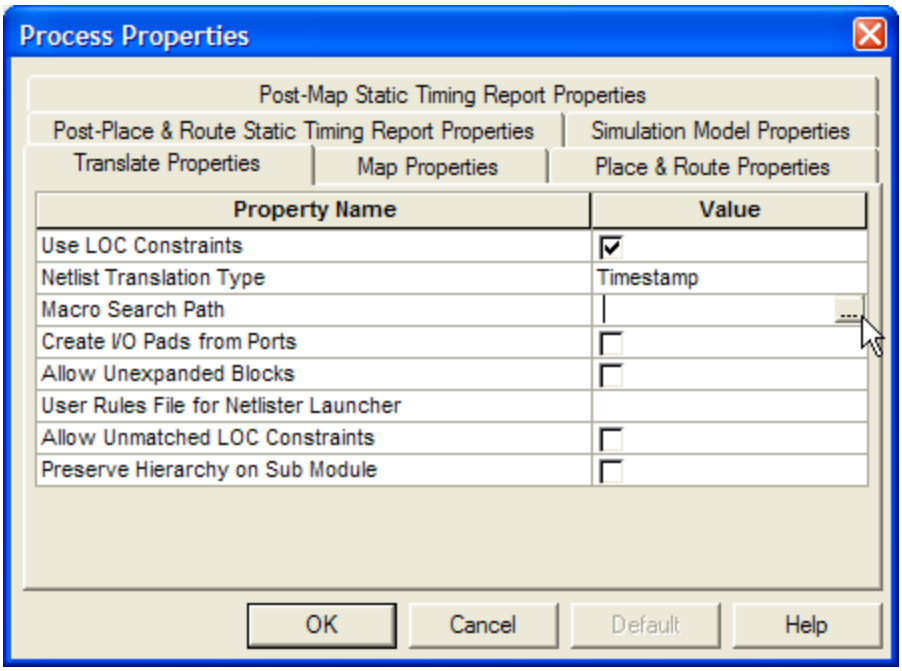

Figure 10: Implementation "Process Properties" Dialog

Find the "Macro Search Path" option and click on the browse button as shown in **Figure 10**. From the file browser, browse to the logic/lib directory (typically located in C:/xport/logic/lib) then click "OK".

Next, find the "Allow Unmatched LOC Constraints" option in the "Process Properties" dialog and click on its check box. Click on "OK" to dismiss the dialog. We are now ready to implement the design. Double click on "Implement Design" in the "Processes for the Current Source" pane to launch the implementation utility. Again, the yellow "!" indicates that one or more warnings were generated. The warnings can again be safely ignored.

All we have to do now is generate the programming file. Scroll to the bottom of the "Processes for Current Source" window and double-click on "Generate Programming File" (**Figure 11**). This will generate the redgreen1.bit file, which can be uploaded into the Xport. We're done!

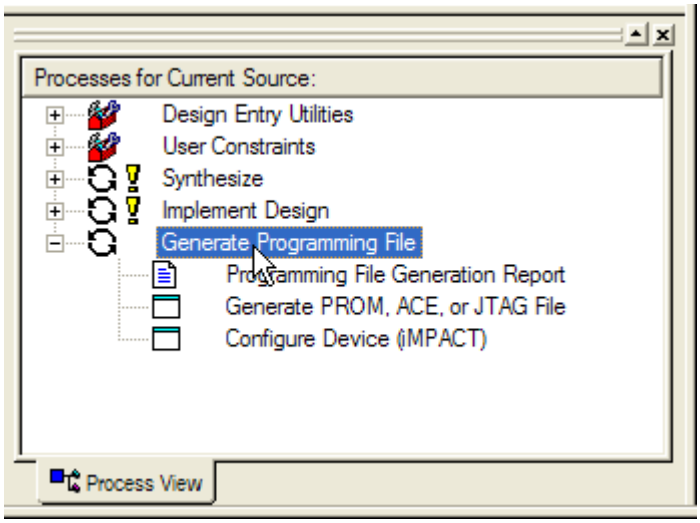

Figure 11: Generate Programming File

Testing Redgreen1

To test the RedGreen1 logic configuration we have provided C++ example software that demonstrates its exciting features. The example code can be found in $XPORTDIR/xport/examples/redgreen1. **Listing 1** provides the contents of the main source file (main.cpp.)

Listing 1: RedGreen C++ Example Software (main.cpp)

```
#include "../../include/textdisp.h"
#include "../../include/gba.h"

CTextDisp td(TDM_LCD);

#define IO_ADDR               *((volatile unsigned short *)0x9ffc400)

int main(void)
        {
        // Turn off LEDs
        IO_ADDR = 0;
        while(1)
                {
                // Write keypad value into IO address.  Use xor operation to invert logic.
                IO_ADDR = GBA_REG_P1^0x03;

                // For illustration purposes, readback what we just wrote and print results.
                if (IO_ADDR==0x01)
                        td.Printf("Red\n");
                else if (IO_ADDR==0x02)
                        td.Printf("Green\n");
                else if (IO_ADDR==0x03)
                        td.Printf("Red and green\n");
                else
                        td.Printf("\n");
                }
        }
```

The comments illustrate what the code does, which isn't too much. Essentially, it reads the GBA keypad register (GBA_REG_P1) and copies it into the memory mapped I/O register we created at 0x9ffc400. The two least significant bits of GBA_REG_P1 happen to correspond to the A and B buttons. The program then prints the states of the LEDs based on what it reads back from the register, demonstrating that we can read as well as write.

Note that since we ignored the lower address bits, we expect that if we read or write to any location between 0x9ffc400 and 0x9ffc5ff (inclusive) we should get the same results (and indeed this is the case). A complied version of the example software (redgreen.bin) is available in the RedGreen1 example directory, but if you would like to compile it yourself, use the Xport Shell and type "make" from the $XPORTDIR/xport/examples/redgreen1 directory.

Programming and Running RedGreen1

To actually see our logic configuration and example code in action, we have to program the Xport. Programming the FPGA is accomplished by running Xpcomm from the Xport Shell as follows:

xpcomm redgreen1.bit

where "redgreen1.bit" is the output bitstream (logic configuration) we created (probably located in $XPORTDIR/examples/myredgreen1/logic).

Programming the flash can be accomplished with Xpcomm as well:

xpcomm redgreen1.bin

where redgreen1.bin is the compiled binary of the example software found in $XPORTDIR/xport/examples/redgreen1.

After you have programmed the FPGA and flash, you should be able to power-cycle the GBA to test RedGreen1. After the GBA logo sequence, the screen will turn dark. You can now press the A and B buttons. Pressing A should illuminate the red LED. Pressing B should illuminate the green LED. Pressing both buttons will illuminate both LEDs. Success!

More to come...

In the near future we will extend the RedGreen1 example to RedGreen2 and RedGreen3. RedGreen2 will use the PWM module to control the duty-cycle and hence brightness of the red and green LEDs. RedGreen3 will add a frequency scaler so that the frequency as well as duty-cycle can be changed from the GBA. And that's about all you can demonstrate with two LEDs!