



XAPP1004 (v1.0) March 14, 2008

Single-Event Upset Mitigation Design Flow for Xilinx FPGA PowerPC Systems

Authors: Greg Miller, Carl Carmichael, and Gary Swift

Summary

Orbital, space-based, and extra-terrestrial applications are susceptible to the effects of high energy charged particles. Single-event upsets (SEUs) can alter the logic state of any static memory element (latch, flip-flop, or RAM cell) including the components of an embedded hard processor. These upsets are unavoidable but correctable for the logic around the processor in FPGA configuration memory.

This application note describes mitigation techniques and corresponding design flow when using a Xilinx FPGA with an embedded processor (specifically the PowerPC[®] 405 found in the Virtex[™]-4 FX family) in high-radiation environments. This example contains a block RAM scrubber example for block RAM blocks attached to the processor local bus (PLB) used for code execution. Since this technique cannot triplicate the PowerPC 405 (PPC405), the surrounding logic is mitigated as much as practically possible. Therefore, the user must determine if the system mitigation is sufficient for the target environment.

Note: It is essential for the reader to have a basic understanding of the Xilinx tool flow using the Xilinx Platform Studio (XPS), triple-module-redundancy (TMR) techniques, the Xilinx TMRTool, and ISE[™] software. An in-depth understanding of [Ref 1] is also essential. In addition, an understanding of VHDL design and practice is recommended.

Tools needed to complete the example design flow

The design flow described in this application note is based upon the following toolset:

- Xilinx ISE software (8.2i SP3)
<http://www.xilinx.com/ise>
- Xilinx XPS (8.2i SP2)
<http://www.xilinx.com/edk>
- Xilinx TMRTool (8.2i) - With V4 Support
<http://www.xilinx.com/tmrtool>
- Plan-Ahead for viewing the netlists to ensure the design is mitigated as expected. This tool is optional, but highly recommended.
www.xilinx.com/planahead
- ModelSim for simulating the system design, if desired. Processor simulation is not covered in this application note. Non-processor block RAM scrubber simulation is covered in [Ref 2].
- ChipScope[™] analyzer (8.2i) for observing the block RAM scrubbing engine on the PLB bus. Use of this tool is also optional.

Note: The revisions listed are the versions used to build this application. Changes to the flow and/or design may be needed when newer versions of the tools become available.

Reference Design Location

The reference design files are available on the Xilinx FTP site at:

http://www.xilinx.com/support/documentation/application_notes/xapp1004.zip.

Introduction

In-orbit, space-based, and extra-terrestrial applications must consider the effects of high-energy charged particles (radiation) can have on electronic components. In particular, SEUs alter the logic state of any static memory element (latch, flip-flop, or RAM cell). Since the user-programmed functionality of an FPGA depends on the data stored in millions of configuration latches within the device, an SEU in the configuration memory array may have adverse effects on the expected functionality. In addition, upsets in user block RAM can accumulate and must be monitored and corrected. Simple triplication of the block RAM might not be enough. A scrubbing macro is needed to allow the FPGA to self correct the upsets that accumulate in block RAM. Refer to [Ref 2], [Ref 3] and [Ref 4] for details.

This application note describes the tool flow required to successfully design, implement and mitigate a single-processor PowerPC design in a Virtex-4 FX device. The user must test the system to determine if these techniques are sufficient for a system in application's target orbit. The block RAM scrubber macro represented here exploits the dual-port nature of the Xilinx block RAMs and uses the same basic structure and technique as used in [Ref 2]. The modified example here replaces block RAMs used on the PLB.

Example Processor System Overview

Embedded PowerPC 405 description

The PowerPC 405 embedded processor core is a reduced instruction set computer (RISC), available in both the Virtex-2 Pro and Virtex-4 FX families. This core is a 64-bit architecture with a 32-bit subset. As shown in Figure 1, the PPC405 contains a 5-stage pipeline, a virtual-memory-management unit supporting multiple pages, separate instruction-cache and data-cache units, debug support including a JTAG interface and three timers.

The PPC405 interfaces to the IBM processor local bus (PLB) for peripheral control. The PLB structure and peripherals use the FPGA fabric resources such as slices, block RAMs and routing.

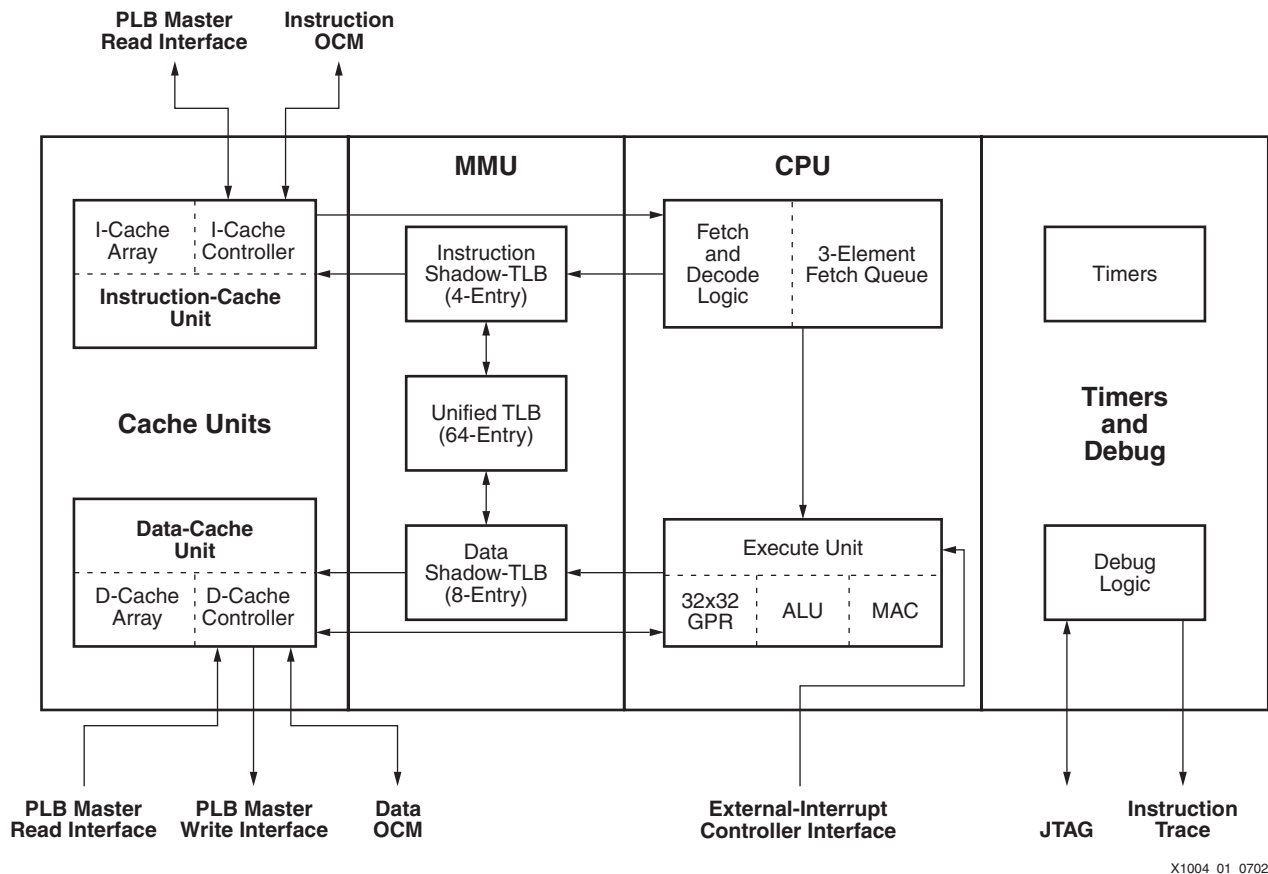


Figure 1: PowerPC 405 Organization

Example Test System Design

Figure 2 illustrates an example PPC405-based system used for this application note along with its connections to the internal memory (internal block RAM) through the PLB. For simplicity, the flow illustrated in this example design does not use external RAM.

The device-under-test (DUT) I/Os are used as a data transfer bus between the DUT and monitoring service FPGAs. For the purposes of this application note, the general-purpose I/O (GPIO) can be anything the user desires and are included as a commonly used peripheral example.

Note: This application does not use digital clock managers (DCMs) because they are not easily mitigated and add extra complexity to the system.

The architecture presented in Figure 2 is a common, simple PPC system, except the block RAM is connected via the PLB. Alternatively the block RAM could connect via the on-chip memory (OCM) interface. This application note demonstrates the use of the PLB block RAM and not the OCM RAM (not shown in Figure 2). The block RAM used for code execution hangs off of the PLB in this example. The PLB block RAM is used in this application note and the block RAM scrubber macro replaces this PLB block RAM (see “Block RAM Correction Algorithm Overview for the PLB Bus”).

The program code in this example resides in the internal block RAM memory using the PLB block RAM controller. The code is pre-initialized in the block RAM each time the DUT is configured.

The PPC405 and system clock speed is variable depending of the system and TMR limitations.

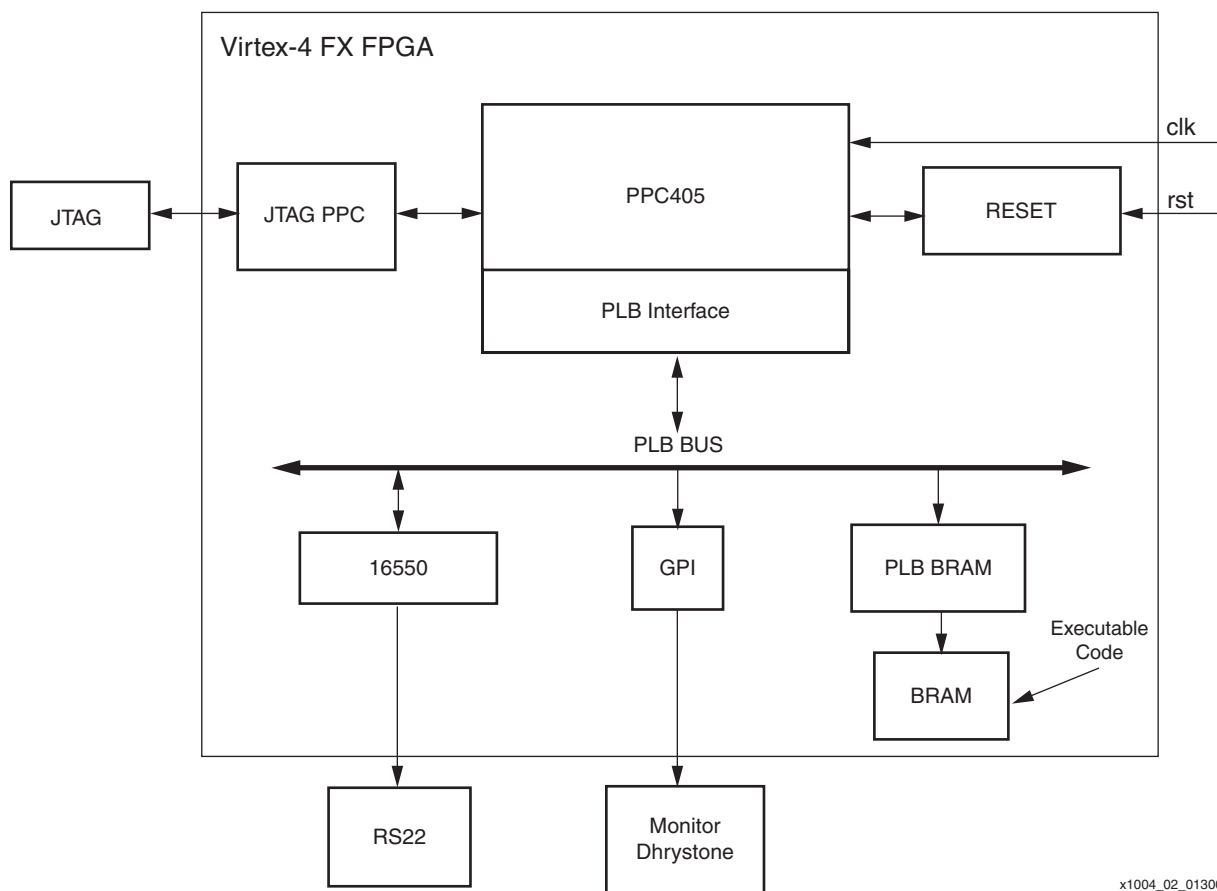


Figure 2: XAPP1004 PPC System Block Diagram

x1004_02_013008

Block RAM Correction Algorithm Overview for the PLB Bus

The configuration scrubbing process does not correct errors in block RAM. One method for scrubbing block RAM is to replace the RAM primitives with macros triplicating the block RAMs and incorporating scrubbing logic (see [Ref 2]).

The main difference between the macro discussed in [Ref 2] and the one used in this example design is that the wrapper for the single block RAM scrubber macro is modified for use with the PLB bus, otherwise, the solutions are identical in operation.

Figure 3 illustrates the block diagram of the example system detailing where the block RAM scrubber engine sits relative to the system. Resident in this block RAM is the CPU code accessed via the PLB block RAM controller.

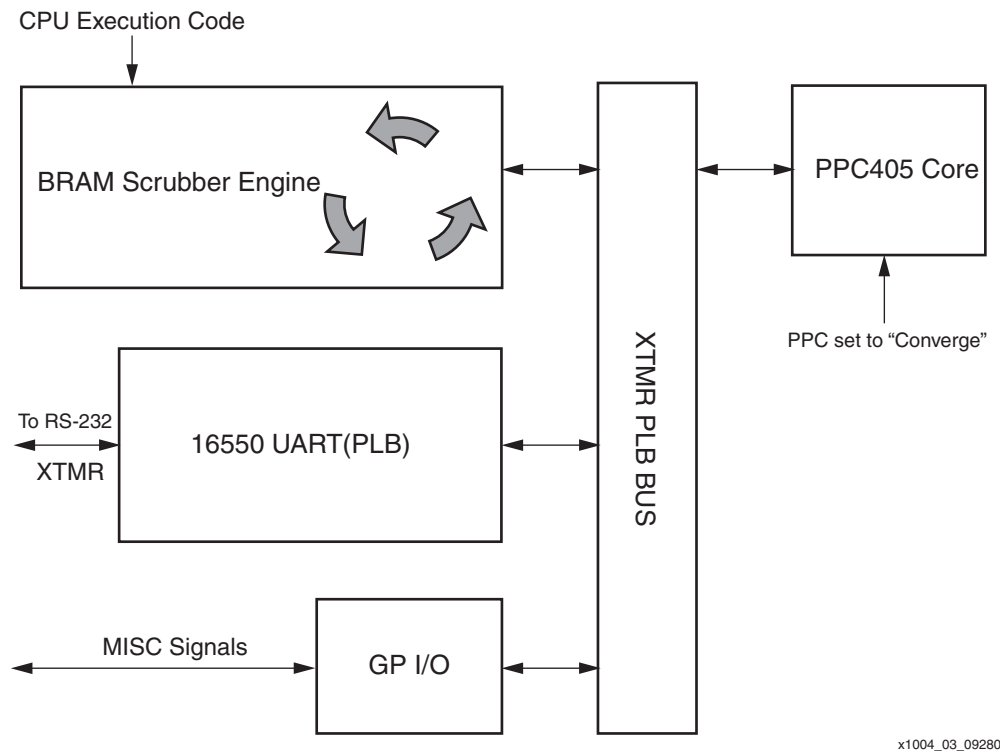
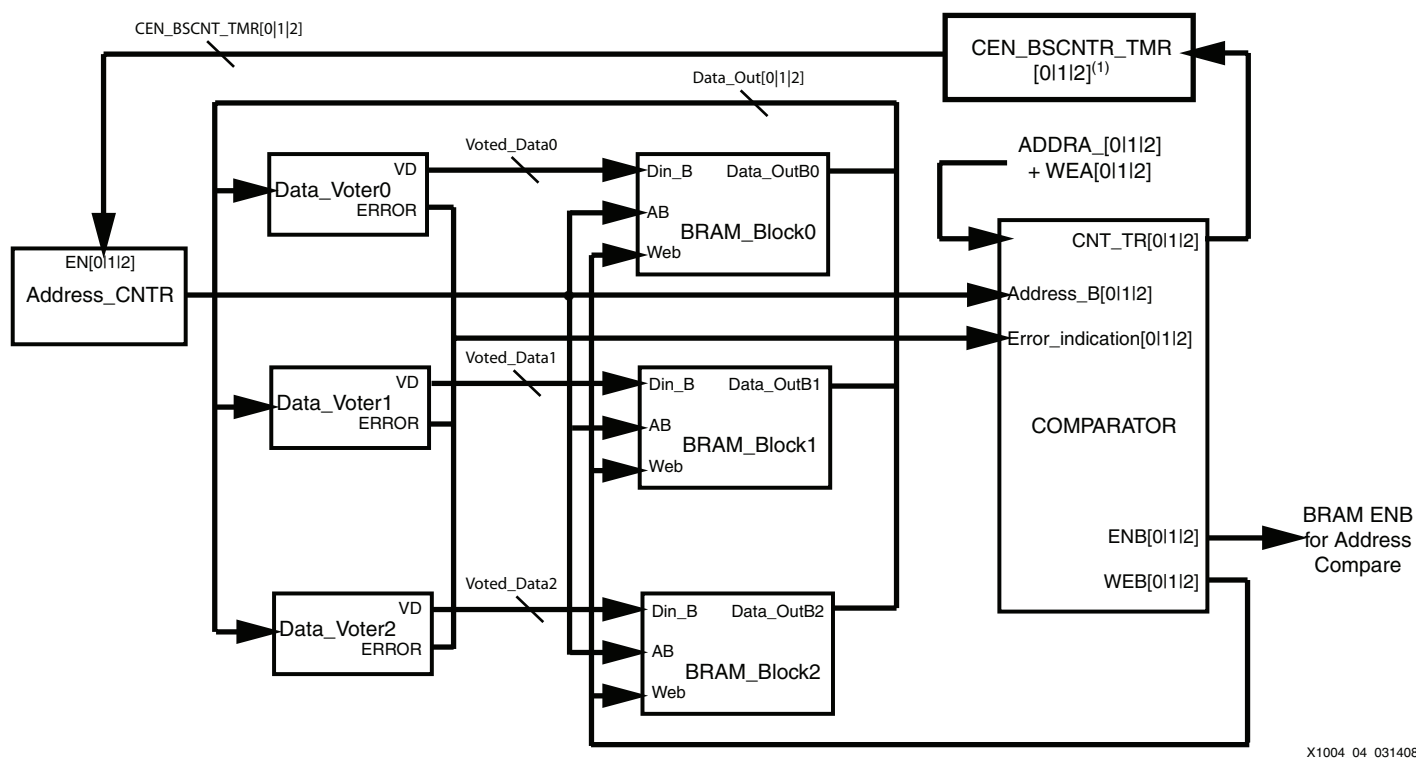


Figure 3: PLB BUS BRAM Scrubber Macro

The block RAM self-correction algorithm is based on a hand-triplicated macro that replaces a group of single-port block RAM primitives. The size and dimensions of the block RAM group depend on the HDL code implementation. In the mitigation scheme detailed here, not every FPGA block RAM primitive has an individual scrubber as many block RAM primitives can be replaced with a single scrubber macro. Multiple BRAM scrubbers may be used in a design.

The block RAM scrubber (Figure 4) functions as follows:

1. An address count is presented at address port B of the block RAMs.
2. Data at address B is presented on port Data_outB.
3. Data is then voted and presented as corrected data on Din_B.
4. If an error is detected, WEB (write enable B) is activated and corrected data is written into Din_B Data port Input B. The address counter for all three domains is paused.
5. After correction, the counter is restarted.
6. The address counter is incremented and the process is repeated.

**Notes:**

1. [0]1|2 indicates triplication.

Figure 4: Block RAM Scrubber Block Diagram

Mitigation Process

Basic mitigation of a PowerPC system includes triplication of the surrounding logic and converging into the PowerPC, since the PowerPC is a hard core and cannot be triplicated. For more information on the TMRTTool converge setting, see [Ref 1]. The goal is to mitigate as much of the system as practically possible. Ideally, everything in the FPGA fabric around the PowerPC is triplicated and converged into the PowerPC hard macro.

The next sections provide details on the exact steps of the design flow required to build the system.

Recommended Design Flow Overview

1. Create the non-mitigated hardware design using the XPS software (ISE software and EDK)
2. Create and compile the software to be executed on this platform. (EDK)
3. Modify and synthesize `system_stub.vhd` (or other top-level file) as needed for the application's particular design needs.
4. Simulate or test the non-mitigated design (ISE software or hardware).
5. Build the block RAM scrubber replacement macro (.ds file) and the counter netlist.
6. Import the synthesized netlists for the system into the Xilinx TMRTTool and set the specific options for the user design.
7. Triplicate the embedded system using the Xilinx TMRTTool.
8. Create triplicated versions of critical user files such as .bmm and .ucf.
9. Create a new ISE project and import the triplicated netlist (in EDIF format) along with the other dependant files.
10. Place and route the completed file, generating a .bit file to configure the FPGA.

Design Flow Details and Step-by-Step Instructions

The following sections contain detailed instructions on creating the mitigated system. The completed design, located in the [xapp1004.zip](#) file, is provided for reference.

This design flow uses an organized directory structure (Table 1). The user may use any structure desired, but it is highly recommended that the individual tool projects are organized into separate directories, allowing projects to be easily re-run at any point during the implementation phase. All directories are relative to where the user unpacked the example design.

Table 1: Recommended Directory Structure

Directory	Comments
/installed_example_dir/	
/Doc/	Contains <code>readme.txt</code> (describes test)
/ISE/	For ISE designs, pre-TMR. This directory can also include 'export to ISE from EDK designs'
/8.2iSP3	Version and service pack
/design_name1	Top or sub designs as needed
/design_name2	Top or sub designs as needed
/EDK	For EDK designs, pre-TMR. This can also include design software code
/8.2iSP2	Version and service pack
/design_name1	Top or sub designs as needed
/design_name2	Top or sub designs as needed
/ISE_TMR	For ISE designs, post TMR
/8.2iSP3	Version and service pack
/design_name1	Top or sub designs as needed
/design_name2	Top or sub designs as needed
/Sim	Third-party simulation project directory, pre-TMR
/golden_designname1	Golden simulation <code>.wlf</code> generated separately for test comparisons
/MSSE_6.1b	Simulator type and version, for example, ModelSim SE Version 6.1b
/design_name1	Design simulation MS project
/Sim_TMR	Third-party simulation project directory, post-TMR
/golden_designname1	Golden simulation <code>.wlf</code> generated separately for test comparisons
/MSSE_6.1b	Simulator type and version,
/design_name1	Design simulation MS project
/source	All the source code for the project, separated by type
/coregen	Core files created by CORE Generator™ software
/Macros	Macros from TMRTool (<code>.ds</code> files)
/Netlists	<code>.edf</code> or other netlists
/TB	Testbench VHDL files
/VHDL	All pertinent VHDL source (non-testbench, synthesizable) code
/Synthesis	Third-party synthesis projects
/synplify_pro_8.4	Tool and version
/design_name1	Design name synthesis project

Table 1: Recommended Directory Structure (Cont'd)

Directory	Comments
/TMR_Tool	Contains the TMRTool projects
/TMR_2.1.55	TMRTool version
/design_name1	Top or sub designs as needed
/design_name2	Top or sub designs as needed

Notes:

1. Some of the directory names listed above, reflect the software version numbers used in this example. These version numbers can change in the future.
2. This design can be targeted to standard development boards (such as the ML405). In this case, the I/Os and clocks must be left untriplicated. However, for complete tripligation of the I/Os, a custom board is needed. An example is the XRTC (Xilinx Radiation Test Consortium) board. For more information on proper tripligation techniques, see [\[Ref 1\]](#).
3. For the purposes of this guide, the ML405 is used because it is more readily accessible than the XRTC board.

Reference Files Included in the Test Design

Table 2 lists the files included with the test design.

Table 2: Included Files

File	Description
plb_bram_if_cntlr_1_bram_wrapper_tmr.vhd	The block RAM scrubber connecting to the PLB block RAM as a scrubber macro replacement. This module is already tripligated and includes the scrubbing engine.
bram_scrubber_pkg.vhd	The main package file for the block RAM scrubber. This file is edited to properly size the block RAM macro to the target design
BSCNTR.vhd	The address counter for port B. This module must be tripligated before using.
bram_scrubber_data_voter.vhd	The data voter for port B of the block RAMs.
bram_scrubber_bram_inst_bitw.vhd	Instantiation of the block RAMs needed in the block RAM scrubber engine in: <pre>plb_bram_if_cntlr_1_bram_wrapper_tmr</pre>
system.vhd	The top-level system file generated by EDK. This file may be modified to accommodate the system for particular changes at a high level. For instance, the user could remove the DCM at this level and replace it with a clock enable.

Design Flow Details

Step 1. Create and Compile the Software for Execution on this Platform (EDK)

The first step in the design flow is to build and verify the software to be run on the PPC405. The software for this example design runs a Dhrystone test with some minor adjustments to how the LEDs are driven to demonstrate the tripligation process later in the application note. This source code is typically written in C or in-line assembly language and is processed with the EDK tool suite to produce an `.elf` file for loading into block RAM for code execution.

XPS provides a wizard interface called the Base System Builder (BSB) to configure and build a system from scratch. BSB helps in obtaining the initial system which is then modified for the example system.

Note: These instructions assume the user is familiar with the Xilinx EDK tool suite. In addition, this step of the process is just for setup and is recorded here for reference. It is recommended for the user to familiarize himself/herself with this design, but can be skipped if the user wants to go directly to the mitigation process flow. The completed software is included with the source files.

1. Create the example XPS project:

Start XPS and select **Base System Builder Wizard** from the initial popup dialog. Choose a directory for this project:

```
/installed_example_dir/EDK/8.2iSP2/system.xmp
```

Click **OK**. When prompted with "C:...\system.xmp file exists. Overwrite?" click **Yes**. Then choose **I would like to create a new design**, and click **Next**.

Note: In the following screens, unless otherwise noted, click **Next** or **Alt + N** to continue.

2. Choose ML405 as the target board:

- ◆ Board Vendor: Xilinx
- ◆ Board Name: Virtex 4 ML405 Evaluation Platform
- ◆ Board Revision: 1

3. Select PowerPC as the processor.

4. Select clocking, PPC405 processor engine, and on-chip memory (OCM) options:

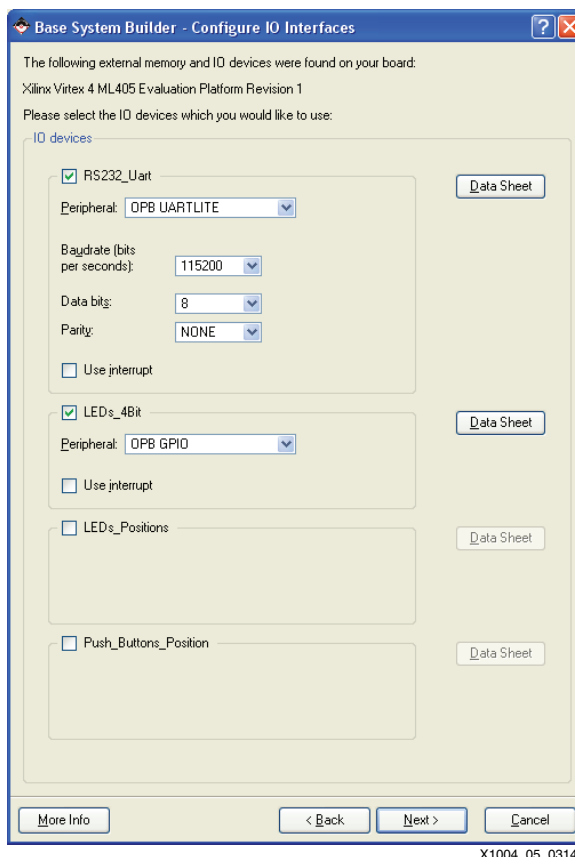
- ◆ Reference Clock Frequency = Processor Clock Frequency = Bus Clock Frequency = 100 MHz
- ◆ PowerPC 405 Processor Engine: FPGA JTAG
- ◆ Cache: Disabled (this setting is just enabled or disabled in the software)
- ◆ Accept the default of no OCM.

5. I/O devices:

Under "IO Devices" (Figure 5), deselect all options except:

- ◆ RS232_Uart (OPB_UARTLITE, 115200, 8, NONE)
- ◆ LEDs_4Bit

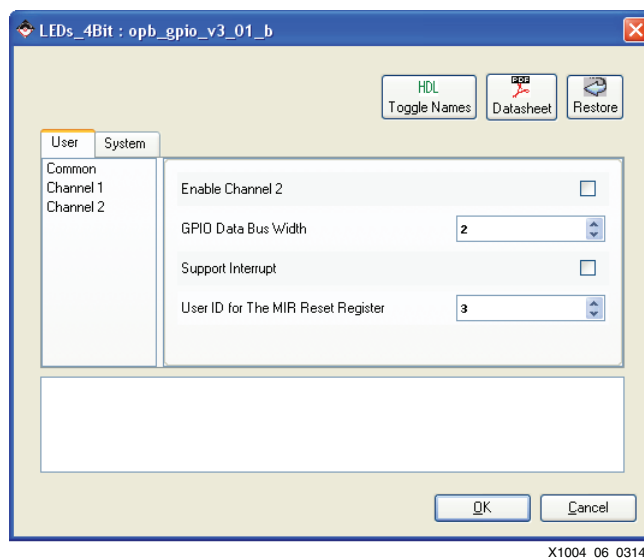
Deselect all other peripherals on the next two pages.



X1004_05_031408

Figure 5: Configure IO Interfaces Dialog Window

6. Select the memory size and build the system.
For the Peripheral PLB_BRAM_IF_CNTRLR select 32KB for the memory size. Click **Finish** or hit **Enter** to build the system. Allow the software to build the memory and peripheral example tests. The memory test is not unusable for this design, but it is a good reference.
7. Generate the project:
Click **Next** until the **Generate** button appears. Click **Generate** followed by **Finish**.
8. Start XPS:
Click **Start using XPS**.
9. Change the number of LEDs:
From the system assembly view, change the system such that it uses two-bit LEDs rather than the original four-bit LEDs (Figure 6). This operation changes several EDK files and the .ucf (see [Ref 5] for details).



X1004_06_031408

Figure 6: opb_GPIO Parameters

10. Rename the LEDs_4bit to LEDs_2bit (optional).
Rename the I/Os and signal names in the .mhs file to indicate two bits. Change the port vector size to two bits [1:0].

```
PORT fpga_0_LEDs_2Bit_GPIO_IO_pin = fpga_0_LEDs_2Bit_GPIO_IO, DIR = IO,
VEC = [0:1]
```

```
BEGIN opb_gpio
PARAMETER INSTANCE = LEDs_2Bit
PARAMETER HW_VER = 3.01.b
PARAMETER C_GPIO_WIDTH = 2
PARAMETER C_IS_DUAL = 0
PARAMETER C_IS_BIDIR = 1
PARAMETER C_ALL_INPUTS = 0
PARAMETER C_BASEADDR = 0x40000000
PARAMETER C_HIGHADDR = 0x4000ffff
BUS_INTERFACE SOPB = opb
PORT GPIO_IO = fpga_0_LEDs_2Bit_GPIO_IO
END
```

11. Modify the .mss file as follows:

```
BEGIN DRIVER
PARAMETER DRIVER_NAME = gpio
PARAMETER DRIVER_VER = 2.11.a
PARAMETER HW_INSTANCE = LEDs_2Bit
END
```

12. Edit the `.ucf` file to accommodate only two bits. The `.ucf` file can be edited while within XPS. Refer to XPS documentation for details:

The `.ucf` file must be changed to accommodate only two bits (by changing LEDs_4Bit o LEDs_2Bit) and allow one of the LEDs to point to DS13/pin G12 (by changing LOC=B10 to LOC=G12). This change can be made directly in EDK and provides for visual verification of triplicated I/Os by observing the LEDs.

Comment out the higher LED bits 2 and 3 that refer to the unused LED bits. Keep the lower bits 0 and 1 as shown in the `.ucf` snippet below.

```
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<0> LOC=A10;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<0> IOSTANDARD = LVCMOS25;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<0> PULLUP;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<0> SLEW = SLOW;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<0> DRIVE = 2;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<0> TIG;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<1> LOC=G12; #was B10
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<1> IOSTANDARD = LVCMOS25;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<1> PULLUP;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<1> SLEW = SLOW;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<1> DRIVE = 2;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin<1> TIG;
```

Comment out constraints for bits 2 and 3 below the constraints shown above.

13. Test the TestApp_Peripheral design (optional).

Connect a null-modem serial cable to the ML405 board. Set the serial monitor (such as Hyperterminal) to 115200 baud (see [Ref 5], and [Ref 6]).

Note: If running the TestApp_Peripheral design, LEDs_4bit was renamed to LEDs_2bit. As a result, some changes to the C code in the main call might be needed, for example:

```
status = GpioOutputExample(XPAR_LEDS_2BIT_DEVICE_ID, 4);
```

Build the entire design and download the test design to the on-board FX20 device (see [Ref 5], and [Ref 6]).

If the design is properly configured, the LEDs DS15, and DS13 light alternately (DS13 is next to the GPIO_SW_W button).

The last three steps are optional, but recommended. These steps create a new software platform for use with an open-source Dhrystone design — a much more interesting design for triplication. This design is just for testing and may generate numbers that cannot be printed by `xil_printf`. The change to `xil_printf` was made to allow the program to fit into the block RAM size allocated.

Note: The two LEDs indicate Dhrystone cycles.

14. Create a new software project:

Select **Add a New Software Project** from the software menu. Name the project Dhrystone and select block RAM initialization.

15. Add source code:

Right-click on sources to add the source code for the Dhrystone design. Go to the `working directory/source/processor_code/Dhrystone` and select the following source files to add to the project:

Source Files:

```
config.c
dhry_2.c
xtime_1.c
dhry_1_nocache.c
```

Headers:

```
dhry.h
xtime_1.h
```

Note: This code has been modified from the original version for demonstration purposes and to fit into the memory footprint available. This design is not intended to be an effective benchmark, but to obtain a reference point.

16. Compile, build, and run the design:

Use EDK to directly compile, map, place and route, then to initialize the bitstream and download the design to the board. Monitor the Dhrystone output using a terminal program such as HyperTerminal set to 115000 baud.

The LEDs indicate if the processor is inside or outside the Dhrystone process. DS13 indicates the processor is in the Dhrystone process, and DS15 (GPIO_LED) indicates it is outside the process (the latter should briefly blink indicating the number of Dhrystone iterations completed).

The next steps begin the actual mitigation flow process.

Step 2. Modify and Synthesize the Top-Level File

The user must modify the top-level design wrapper to accommodate the special requirements for the triplicated design. For example, in this PPC design, the DCM must be removed and the appropriate signals connected. DCMs are hard to mitigate, and for purposes of this example design, are removed. In addition, a clock disable is added to allow for smoke testing.

First, isolate and organize the project files from EDK:

1. Copy the .vhd file generated from EDK:

```
project_install_dir/EDK/8.2iSP2/hdl/system.vhd
```

to:

```
project_install_dir/source/VHDL/system.vhd
```

2. Copy the .ucf file from:

```
project_install_dir/EDK/8.2iSP2/data/system.ucf
```

to

```
project_install_dir/source/constraints/system.ucf.
```

Make another copy of the `system.ucf` in the same directory, and rename it `system_tmr.ucf` for later use.

Note: A `system_tmr.ucf` file is included in the reference design archive for this application note.

3. Copy the .bmm file from:

```
project_install_dir/EDK/8.2iSP2/implementation/system.bmm
```

to:

```
project_install_dir/source/constraints/system.bmm.
```

Make another copy of `system.bmm` in the same directory and name it `system_tmr.bmm`.

Note: A `system_tmr.bmm` file is included in the reference design archive for this application note.

Next, using ISE software (XST), synthesize the top-level HLD code to produce an `.ngc` netlist. This netlist is used as the top level for TMRTool along with the system netlists (`.ngo`).

4. Launch ISE software and create a new ISE project in the

```
project_install_dir/ISE/8.2i_SP3
```

, and name the design Dhrystone.

5. Click Next until "Add existing source" appears. Add the file `system.vhd` from the directory `project_install_dir/source/VHDL/`. Be sure to uncheck **Copy to Project**.

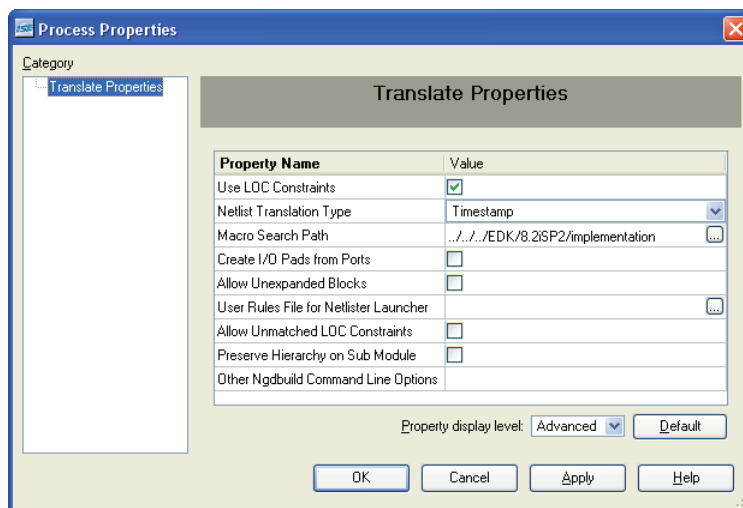
6. Select the device on the ML405 board (XC4VFX20-FF672-10C). The top-level source is HDL.

7. Add the existing file `system.ucf` from the directory

```
project_install_dir/source/constraints/
```

. Be sure to uncheck **Copy to Project**. Click **Next** and **Finish** to complete.

8. While still in the ISE software, expand "Implement Design". Right-click on **Translate** and add `../../../../EDK/8.2iSP2/implementation` to the macro search path (Figure 7, page 12). Click **OK**.



X1004_07_031408

Figure 7: Translate Macro Path

- Add two more files (the `.bmm` and `.elf`) to the ISE project by right-clicking on the device `xc4fx20-10ff672` and selecting **Add source**. Add the file `system.bmm` from the directory `project_install_dir/source/constraints/` and the file `executable.elf` from the directory `project_install_dir/EDK/8.2iSP2/Dhrystone/`.

Note: A script could be created to write out a `.bit` file instead of adding `system.bmm` and `executable.elf` to the ISE project, with the block RAMs being loaded using `data2mem`. See [\[Ref 5\]](#) for details.

- (Optional) Before modifying the `system.vhd` and `system.ucf` files, the user can compile the design, completing the rest of the process through BitGen and programming the device. The design should operate as described in [step 16, page 11](#).
- The `system.vhd` file must be modified to remove the DCM and add a signal for button control of the clock. Double-click on the file `system.vhd` inside the ISE project and make the following changes:

Note: If `system.vhd` is corrupted, remember a copy is stored in the EDK directory under `implementation`. Simply re-copy and start over.

Add one line at the beginning of the file under `entity system` to enable the control of the clock (on/off), and name the pin `clk_cntrl_pin`.

```
entity system is
  port (
    clk_cntrl_pin : in std_logic;
    fpga_0_RS232_Uart_RX_pin : in std_logic;
    fpga_0_RS232_Uart_TX_pin : out std_logic;
    fpga_0_LEDs_2Bit_GPIO_IO_pin : inout std_logic_vector(0 to 1);
    sys_clk_pin : in std_logic;
    sys_rst_pin : in std_logic
  );
end system;
```

Add the following component near the DCM component around line 664:

```
component BUFGCE
  port (
    I : in std_logic;
    CE : in std_logic;
    O : out std_logic
  );
end component;
```

Comment out the `dcm_0 : dcm_0_wrapper` module and replace with a `BUFGCE` to feed the system clock and control the clock from the pin added above (around line 1430).

```
-- dcm_0 : dcm_0_wrapper
-- port map (
-- RST => net_gnd0,
-- CLKIN => dcm_clk_s,
-- CLKFB => sys_clk_s,
-- PSEN => net_gnd0,
-- PSINCDEC => net_gnd0,
-- PSCLK => net_gnd0,
-- DSSEN => net_gnd0,
-- CLK0 => sys_clk_s,
-- CLK90 => open,
-- CLK180 => open,
-- CLK270 => open,
-- CLKDV => open,
-- CLK2X => open,
-- CLK2X180 => open,
-- CLKFX => open,
-- CLKFX180 => open,
-- STATUS => open,
-- LOCKED => dcm_0_lock,
-- PSDONE => open
-- );

-----

BUFG_MAIN_CLK : BUFGCE
port map
(
I => dcm_clk_s,
CE => not clk_cntrl_pin,
O => sys_clk_s
);
```

Add the following lines to `system.ucf` in

`project_install_dir/source/constraints/` by clicking on the `.ucf` or by double-clicking on **Edit Constraints (Text)** in the Processes window:

```
#####
NET "clk_cntrl_pin" LOC = K8;
NET "clk_cntrl_pin" TIG;
#####
```

Force an enable on the `dcm_locked` signal on the reset controller by completing the following changes (around line 1077).

```
reset_block : reset_block_wrapper
port map (
Slowest_sync_clk => sys_clk_s,
Ext_Reset_In => sys_rst_s,
Aux_Reset_In => net_gnd0,
Core_Reset_Req => C405RSTCORERESETREQ,
Chip_Reset_Req => C405RSTCHIPRESETREQ,
System_Reset_Req => C405RSTSYSRESETREQ,
-- Dcm_locked => dcm_0_lock,
Dcm_locked => '1',
Rstc405resetcore => RSTC405RESETCORE,
Rstc405resetchip => RSTC405RESETCHIP,
Rstc405resetsys => RSTC405RESETSYS,
Bus_Struct_Reset => sys_bus_reset(0 to 0),
Peripheral_Reset => open
```

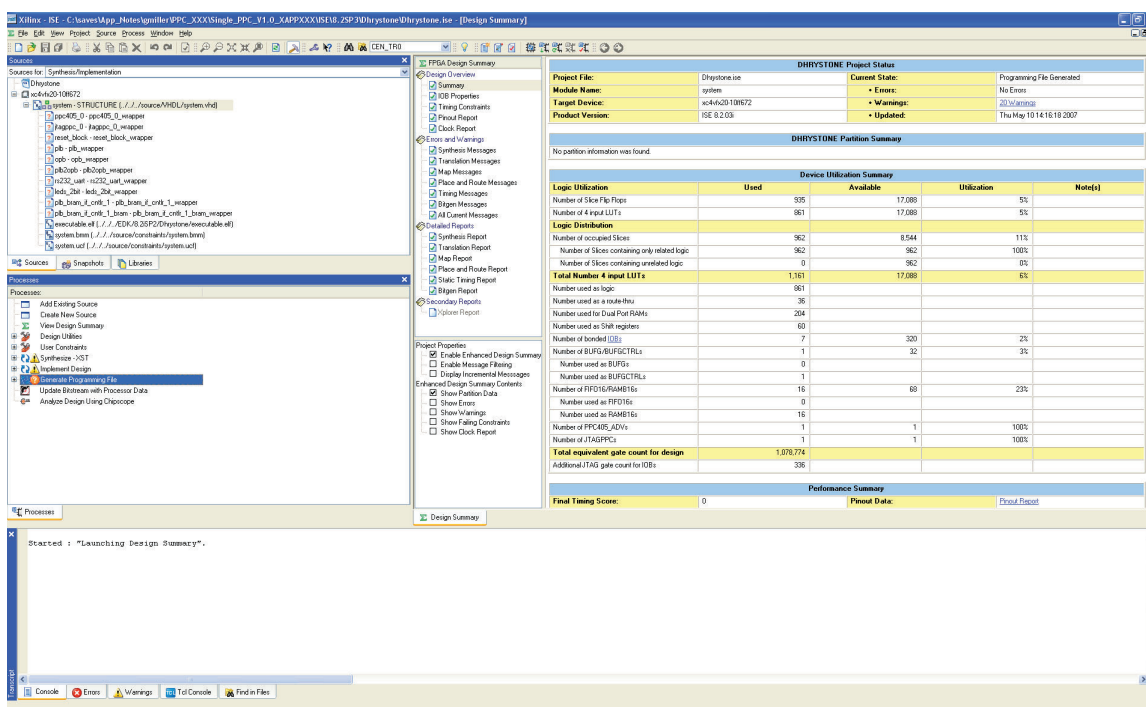
Step 3. Test and/or Simulate the Non-Mitigated Design in Hardware (ISE software/EDK plus Hardware)

After the hardware system is built using XPS, and the processor code written and tested with the EDK flow, a test of the non-mitigated version in hardware is essential to ensure the changes to the system files are correct.

After the changes are made in “[Step 2. Modify and Synthesize the Top-Level File](#),” recompile and re-build the .bit file using the ISE flow and the Dhrystone design built in “[Step 1. Create and Compile the Software for Execution on this Platform \(EDK\)](#).” The entire flow can be run by double-clicking **Generate Programming File**. Reprogram the device to ensure it functions as before.

The design modifications in “[Step 2. Modify and Synthesize the Top-Level File](#)” allow the clock to be disabled by pressing the GPIO_SW_W (sw7) button. Press the button to verify that the design halts and is restored when the button is released.

Figure 8 shows the ISE flow for non-mitigated ISE project Dhrystone generated by EDK from “[Step 2. Modify and Synthesize the Top-Level File](#).”



X1004_08_031408

Figure 8: ISE Non-Mitigated System

Step 4. Build the Block RAM Scrubber Replacement Macro (.as File) and the Counter Netlist

A block RAM scrubber is needed to refresh the PLB-controlled block RAM containing executable code to prevent the data from becoming stale. After the parameters are set in the VHDL code (bram_scrubber_pkg.vhd), the block RAM replacement scrubber ports match those for the block RAM created during the system creation. For this design, a setting of data_width=64/ addr_width=12, we_num = 8 and scrub_delay = 1 are used.

Note: When the block RAM scrubber wrapper plb_bram_if_cntlr_1_bram_wrapper_tmr is used to replace plb_bram_if_cntlr_1_bram_wrapper (PLB block RAM controller), port B is not used. Omitting the use of port B is the key to the use of the block RAM scrubber.

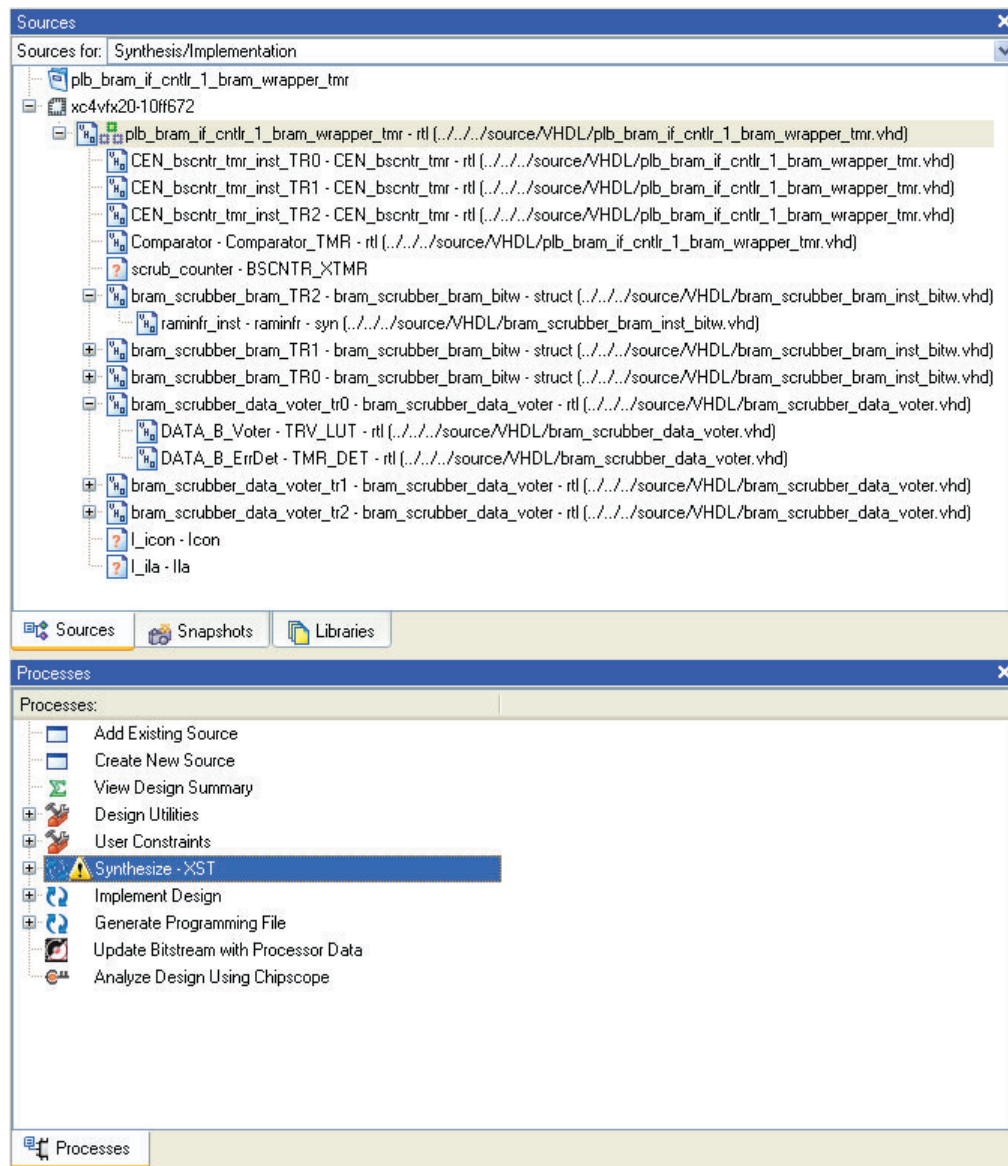
For this step, use XST to synthesize the block RAM scrubber macro replacement (top level: plb_bram_if_cntlr_1_bram_wrapper_tmr) to create the proper .ngc file to use in TMRTool.

1. Launch ISE software and create a new ISE project in the `project_install_dir/ISE/8.2i_SP3` and name the design `plb_bram_if_cntlr_1_bram_wrapper_tmr`.
2. Set up the project as in “[Step 2. Modify and Synthesize the Top-Level File,](#)” (FX20-10, etc.) and add the following files ([Figure 9](#)):

```
project_install_dir/source/VHDL/plb_bram_if_cntlr_1_bram_wrapper_tmr.vhd
project_install_dir/source/VHDL/bram_scrubber_bram_inst_bitw.vhd
project_install_dir/source/VHDL/bram_scrubber_pkg.vhd
project_install_dir/source/VHDL/bram_scrubber_data_voter.vhd
```

Caution! Make sure to set the option in ISE software to *not* copy the files to the project directory.

For this application note, the block RAM scrubber for the PLB is marked as version 8.2.3.

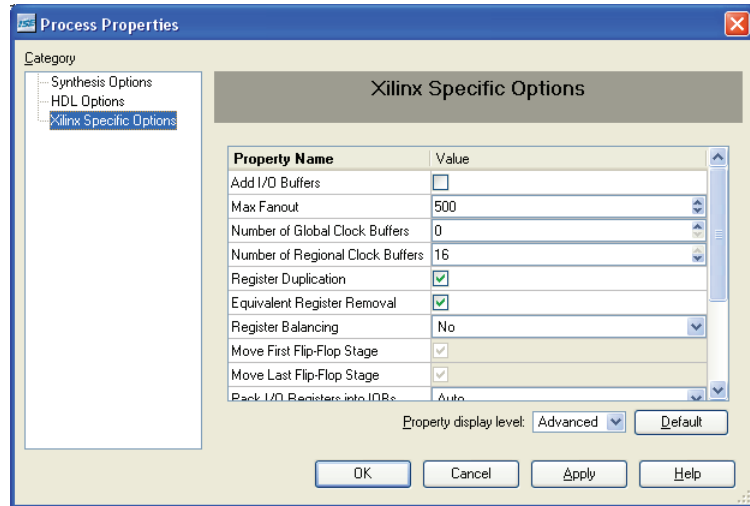


X1004_09_031408

Figure 9: The `plb_bram_if_cntlr_1_bram_wrapper_tmr` Project

3. Right-click on **Synthesize - XST** under processes, then click on **Synthesis Directives** and make the following changes (Figure 10):
 - ◆ Add I/O Buffers (uncheck).
 - ◆ Number of Global Clock Buffers should be 0.

These changes are needed because this is a sub level in the TMR process (neither I/O nor clock buffers are needed).



X1004_10_031408

Figure 10: ISE Synthesis Options

4. Click on the **libraries** tab under sources, and expand the Work source library. Double-click on the `bram_scrubber_pkg.vhd` file, and ensure that the constants under the `bram_scrubber_package` are set as follows:
 - ◆ constant `data_width` : integer := 64
 - ◆ constant `addr_width` : integer := 12;
 - ◆ constant `we_num` : integer := 8;
 - ◆ constant `BRAM_Data_Size` : integer := 4;
 - ◆ constant `scrub_delay` : integer := 1;

These constants allow for building different sizes of the block RAM scrubber. There are two places where changes are needed:

- ◆ `bram_scrubber_pkg.vhd` (as shown in the declare constants section of the VHDL file. Descriptions of the constants are contained in the VHDL file.)
- ◆ `bram_scrubber_bram_inst_bitw.vhd` (instantiation for BRAM primitive type. Changes are needed in two locations to accomplish the BRAM primitive instantiation: the component section and in the for loop. For this design, the instantiation is set to `ramb16_s4_s4` and does not need to be modified. See [Ref 2] for more details.)

To build the netlist, double-click on **Synthesize-XST**.

5. Next, import the synthesized netlist into TMRTool. Importing the `.ngc` file creates the `.ds` file. This netlist is pre-TMR since this part of the design is hand triplicated. Even though TMRTool indicates it is pre-XTMR, it is still triplicated (previously by hand). See [Ref 1] for details on macro use.
6. Open TMRTool and select **File** → **New Project**. Create the project under `project_install_dir/TMR_Tool/TMR_2.1.55/`. Add a slash after the directory name or TMRTool might create the project one level up. Name the project `plb_bram_if_cntrl_1_bram_wrapper_tmr`, then click **OK** (Figure 11).

Note: The application note design archive already contains this project. Either use the existing project or create a new one.

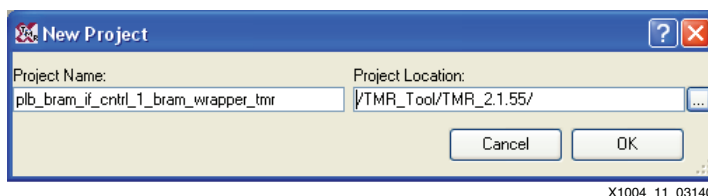


Figure 11: Creation of TMRTool BRAM Scrubber Wrapper Project

7. Add the netlist created in [step 4, page 16](#), by right-clicking in "Sources for Project" and selecting **Set Top Level Source**. Browse to:


```
project_install_dir/ISE/8.2SP3/plb_bram_if_cntrl_1_bram_wrapper_tmr
```

 and select `plb_bram_if_cntrl_1_bram_wrapper_tmr.ngc`. Click **Open**.
8. While still in TMRTool, double-click on **Import** under "Processes for Source" to create the needed macro for use later in the system triplication process. This macro replaces the PLB block RAMs with the PLB block RAM scrubber macro.

Copy the `.ds` file into the `TMRTool_Install_Dir/macrolib/USER` directory. The `.ds` custom macro is now ready for use.
9. In Windows, browse to:


```
project_install_dir/TMR_Tool/TMR_2.1.55/plb_bram_if_cntrl_1_bram_wrapper_tmr
```

 and copy the file `plb_bram_if_cntrl_1_bram_wrapper_tmr.ds` to `TMRTool_install_dir/macrolib/USER`. The PLB block RAM macro is now ready for use.

Synthesize the address counter (`BSCNTR.vhd`) and use TMRTool to triplicate it. This counter is used to continually cycle through the addresses to check block RAM integrity and needs to be triplicated.

10. Launch ISE software and create a new ISE project in the `project_install_dir/ISE/8.2i_SP2` and name the design `BSCNTR`.
11. Set up the project as in the above projects (FX20-10 etc...) and add the following files:


```
project_install_dir/source/VHDL/BSCNTR.vhd
```

```
project_install_dir/source/VHDL/bram_scrubber_pkg.vhd
```

Caution! Do not copy the files to the project directory.
12. Make the same changes to **Synthesize - XST** as in [step 3, page 16](#) ([Figure 9, page 15](#) and [Figure 10, page 16](#)). Double-click on **Synthesize-XST** to build the netlist.

Use TMRTool to triplicate the counter and produce a triplicated EDIF netlist. Make sure to tell TMRTool to produce a "Design Module". This netlist is needed when placing and routing the entire design. Copy this EDIF file to a convenient location so that it is visible to the ngdbuild process.

13. Open TMRTool and select **New Project** under the **File** directory. Create the project under `project_install_dir/TMR_Tool/TMR_2.1.55/`. Add a slash after the directory name or TMRTool may create the project one level up. Name the project `BSCNTR`, then click **OK** ([Figure 12](#)).

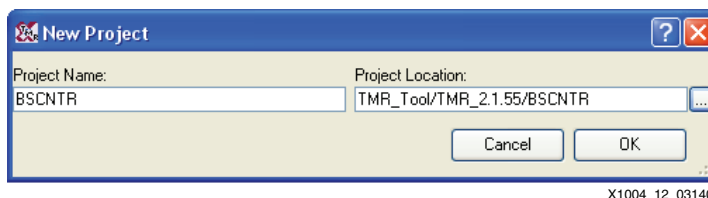
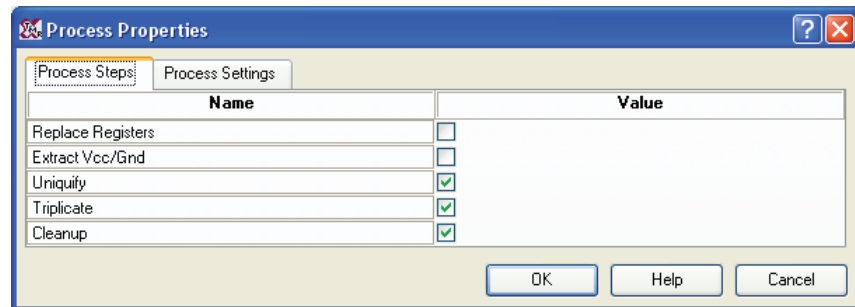


Figure 12: Creating Project BSCNTR

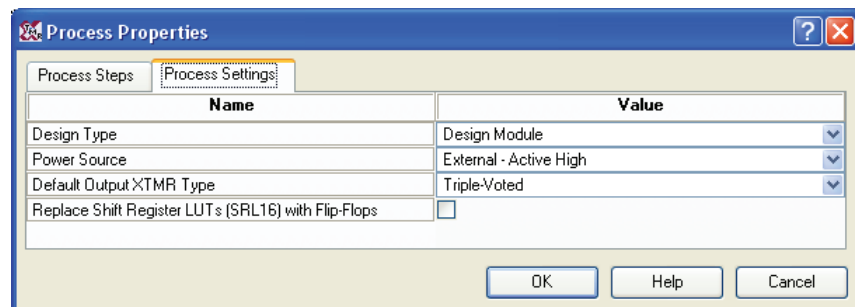
14. Add the netlist created in [step 9](#), by right-clicking in "Sources for Project" and selecting **Set Top Level Source**. Brows to `project_install_dir/ISE/8.2SP3/BSCNTR/` and select `BSCNTR.ngc`. Click **Open**.
15. Right-Click on **Implement XTMR** in "Processes for Source". Uncheck **Replace Registers** and **Extract GND/VCC** ([Figure 13](#)).



X1004_13_031408

Figure 13: TMRTool BSCNTR Process Steps

16. Within the same window, Click "Process Settings" and change the "Design Type" to **Design Module**. Uncheck **Replace SRL 16s**, and click **OK** ([Figure 14](#)).



X1004_14_031408

Figure 14: TMRTool BSCNTR Process Settings

17. Double-click **Export Design** under "Processes for Source". This process generates the triplicated netlist used in the final processing of the triplicated design.
18. Within Windows, browse to the `project_install_dir/TMR_Tool/TMR_2.1.55/BSCNTR` and find the file `BSCNTR_xtmr.edf` created in [step 17](#) for use in the final step to build the PPC triplicated system with the block RAM scrubber. Copy the file `BSCNTR_xtmr.edf` to the `project_install_dir/source/Netlists` directory so that translate can insert this netlist when building the final system.

Step 5. Import the Hardware System Top-Level Netlist and the Synthesized Netlists (TMRTool)

For this step, the entire EDK system infrastructure is imported into the TMRTool for triplication.

Note: The main system netlists can usually be found in the EDK project directory under `../implementation`.

1. Open TMRTool and select **File** → **New Project**, and create the project under `project_install_dir/TMR_Tool/TMR_2.1.55/`. Add a slash after the directory name or TMRTool may create the project one level up. Name the project `ppc_ds_tmr_bs`, and then click **OK**.
2. Add the top level netlist created in "[Step 2. Modify and Synthesize the Top-Level File,](#)" [page 11](#) by right-clicking in "Sources for Project" and selecting **Set Top Level Source**. Browse to `project_install_dir/ISE/8.2SP3/Dhrystone/`, select `system.ngc`, and click **Open**.

3. Add the sub-level netlists created by EDK in “[Step 1. Create and Compile the Software for Execution on this Platform \(EDK\)](#),” page 7 by right-clicking in “Sources for Project” and selecting **Add Source**. Browse to:

```
project_install_dir/EDK/8.2iSP2/implementation/
```

And select the following files to add to the TMRTool project ([Figure 15](#)):

```
jtagppc_0_wrapper.ngc
leds_2bit_wrapper.ngc
opb_wrapper.ngc
plb_bram_if_cntlr_1_bram_wrapper.ngc
plb_wrapper.ngc
ppc405_0_wrapper.ngc
plb2opb_wrapper.ngc
reset_block_wrapper.ngc
rs232_uart_wrapper.ngc
```

Note: The file `plb_bram_if_cntlr_1_bram_wrapper.ngc` is not included, because it is replaced with the block RAM scrubber macro.

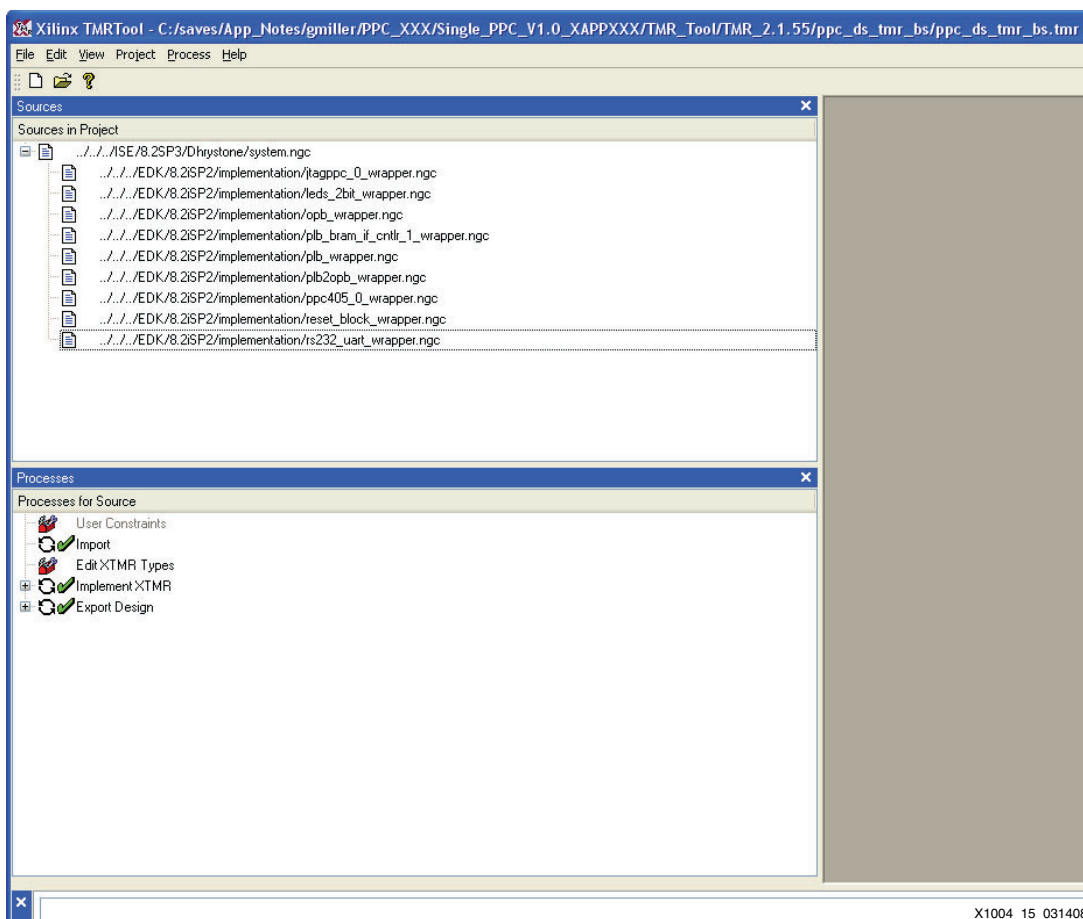


Figure 15: Project `ppc_ds_tmr_bs` in TMRTool

4. Double-click on **Edit XTMR Types** to import the design and display the Set XTMR Types on the right pane.

Set TMRTool options for the user specific system design. For example, in the Dhystone system, the PPC 405 is set to converge and the `plb_bram_if_cntlr_1_bram_wrapper` is replaced with the block RAM scrubber macro (`plb_bram_cntrl_1_bram_wrapper_tmr_pretmr`).

- Once the design is imported, make the following changes to the components under XTMR Types pull-down menu (refer to [Ref 1] for details).

Set the following to converge:

- ◆ PPC405_ADV
- ◆ JTAGPPC
- ◆ JTAGPPC_0_WRAPPER

Expand Set plb_bram_if_cntrl_1_bram_wrapper and set plb_bram_if_cntrl_1_bram_wrapper to "Custom". Then select plb_bram_if_cntrl_1_wrapper_tmr_prextmr under the user directory, and click **OK** (Figure 16).

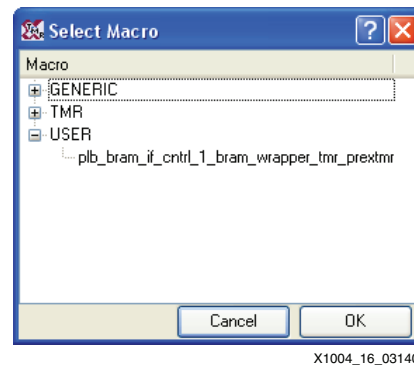


Figure 16: Custom Macro Selection

Expand IBUF and set the following to "Don't Touch":

- ◆ fpga_0_RS232_Uart_RX_pin_IBUF
- ◆ sys_rst_pin_IBUF

Expand OBUF and set the following to "Don't Touch":

- ◆ fpga_0_RS232_Uart_TX_pin_OBUF to converge

Expand IBUFG and set the following to "Don't Touch":

- ◆ sys_clk_pin_IBUFG

- Right-click on **Implement XTMR** under the "Processes for Source" window and select **Properties**. Make the following changes in the Process Steps tab (Figure 17).

- ◆ Uncheck **Replace Registers**.
- ◆ Uncheck **Extract Vcc/Gnd**.
- ◆ Leave all other options checked.

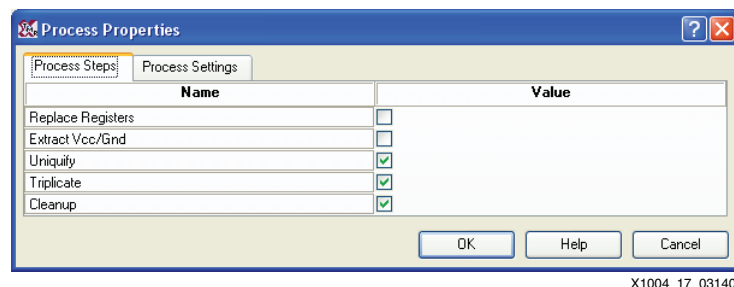
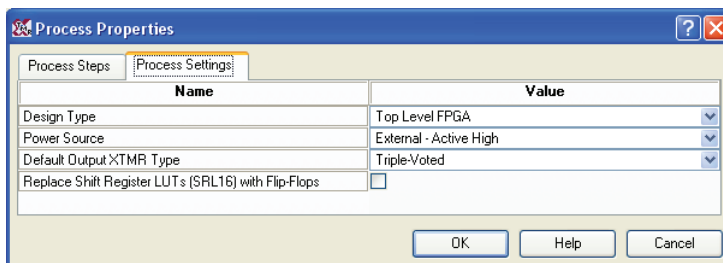


Figure 17: TMRTool Dhrystone Process Steps

Within the same window check the Process settings tab and make the following changes (Figure 18):

- ◆ Uncheck **Replace Shift Register LUT's**.
- ◆ Leave all other options as is.



X1004_18_031408

Figure 18: TMRTool Dhrystone Process Settings

Step 6. Triplicate the Embedded System Using the Xilinx TMRTool and Produce a Triplicated Netlist (TMRTool)

When all the options are set, tell TMRTool to begin triplication of the design (all logic surrounding the PPC is triplicated), and insert the block RAM scrubber macro for the PLB block RAM replacement. The resulting netlist contains the entire mitigated design except for the block RAM scrubber address counter to be used later.

1. Double-click **Export Design** under “Processes for Source”. This process can take a few minutes while it produces the triplicated system netlist for the PowerPC system.

Step 7. Create a Triplicated Version of .ucf

For creation of the triplicated .ucf file, please see [Ref 1]. The example .ucf included here for this design is created for reference only as this file is design dependant.

1. Copy the `system.ucf` from `project_install_dir/EDK/8.2iSP2/data` to `project_install_dir/source/constraints`. This file is the same as modified in “Step 1. Create and Compile the Software for Execution on this Platform (EDK),” page 7 for changes to the LEDs, etc. Rename this file to `system_tmr.ucf` to indicate that this is the triplicated version.
2. Change the .ucf file to the triplicated version. Modify the triplicated .ucf file `system_tmr.ucf` similar to the following:

```
# Set for CES4 chip
CONFIG STEPPING = "SCD1";

Net sys_clk_pin LOC=AB14;
Net sys_clk_pin IOSTANDARD = LVCMOS33;
Net sys_rst_pin LOC=M5;
Net sys_rst_pin PULLUP;
## System level constraints
Net sys_clk_pin TNM_NET = sys_clk_pin;
#TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 10000 ps;

Net sys_clk_s_TR0 TNM_NET = sys_clk_s_TR0;
TIMESPEC "TS_sys_clk_s_TR0" = PERIOD "sys_clk_s_TR0" 10 ns HIGH 50 %;
Net sys_clk_s_TR1 TNM_NET = sys_clk_s_TR1;
TIMESPEC "TS_sys_clk_s_TR1" = PERIOD "sys_clk_s_TR1" "TS_sys_clk_s_TR0" * 1;
Net sys_clk_s_TR2 TNM_NET = sys_clk_s_TR2;
TIMESPEC "TS_sys_clk_s_TR2" = PERIOD "sys_clk_s_TR2" "TS_sys_clk_s_TR0" * 1;
Net sys_rst_pin TIG;
NET "*C405RSTCORERESETREQ*" TPTHU = "RST_GRP";
NET "*C405RSTCHIPRESETREQ*" TPTHU = "RST_GRP";
NET "*C405RSTSYSRESETREQ*" TPTHU = "RST_GRP";
TIMESPEC "TS_RST1" = FROM CPUS THRU RST_GRP TO FFS TIG;
```

```

## IO Devices constraints

### Module RS232_Uart constraints

Net fpga_0_RS232_Uart_RX_pin LOC=T4;
Net fpga_0_RS232_Uart_RX_pin IOSTANDARD = LVCMOS33;
Net fpga_0_RS232_Uart_TX_pin LOC=T8;
Net fpga_0_RS232_Uart_TX_pin IOSTANDARD = LVCMOS33;

#####
NET "clk_cntrl_pin_TR0" LOC = K8; #NET gpio<10> LOC = K8; # W Button
NET "clk_cntrl_pin_TR0" TIG;

NET "clk_cntrl_pin_TR1" LOC = D6; #NET gpio<10> LOC = K8; # W Button
NET "clk_cntrl_pin_TR1" TIG;

NET "clk_cntrl_pin_TR2" LOC = M6; #NET gpio<10> LOC = K8; # W Button
NET "clk_cntrl_pin_TR2" TIG;
#####

### Module LEDs_4Bit constraints

Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<0> LOC=A10;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<0> IOSTANDARD = LVCMOS25;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<0> PULLUP;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<0> SLEW = SLOW;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<0> DRIVE = 2;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<0> TIG;

Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<0> LOC=B10;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<0> IOSTANDARD = LVCMOS25;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<0> PULLUP;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<0> SLEW = SLOW;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<0> DRIVE = 2;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<0> TIG;

Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<0> LOC=F13;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<0> IOSTANDARD = LVCMOS25;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<0> PULLUP;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<0> SLEW = SLOW;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<0> DRIVE = 2;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<0> TIG;

Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<1> LOC=G12; #was B10
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<1> IOSTANDARD = LVCMOS25;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<1> PULLUP;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<1> SLEW = SLOW;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<1> DRIVE = 2;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR0<1> TIG;

Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<1> LOC=E6;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<1> IOSTANDARD = LVCMOS25;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<1> PULLUP;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<1> SLEW = SLOW;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<1> DRIVE = 2;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR1<1> TIG;

Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<1> LOC=L7;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<1> IOSTANDARD = LVCMOS25;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<1> PULLUP;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<1> SLEW = SLOW;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<1> DRIVE = 2;
Net fpga_0_LEDs_2Bit_GPIO_IO_pin_TR2<1> TIG;

```

The system constraint files are now be ready for use in the final mitigated system design.

Step 8. Create a Triplicated Version of .bmm

A .bmm file is necessary if the user needs to initialize the code. The executable code (.elf file) can be loaded via the debug port using the Xilinx microprocessor debugger (XMD). However, to create a .bit file that runs without the XMD interface, it is necessary to use the .bmm file.

The creation of the triplicated .bmm file is not straightforward. The structure of this file depends on the size of the block RAM memory address range, the types of block RAMs used, whether the design is triplicated with the block RAM scrubber or not. Examples of the .bmm file are provided with the source files for reference (the .bmm (triplicated system only) and the .bmm with the block RAM scrubber).

The suggested method for creating the triplicated .bmm file is to start with a copy of the non-triplicated version, first making the triplicated version without the block RAM scrubber, followed by the final version with the block RAM scrubber. By making the triplicated version first, the user has the option to test the triplicated system for proper functionality without the block RAM scrubber. The Xilinx Tool PlanAhead can assist the user in the process of making the proper .bmm files. By importing the netlist into PlanAhead, and clicking on the PLB controller block RAMs, the hierarchical pattern is displayed. PlanAhead helps display the proper block RAM hierarchal netlist used in the creation of the triplicated .bmm file.

1. Copy the system.bmm file from:

```
project_install_dir/EDK/8.2iSP2/implementation
```

to:

```
project_install_dir/source/constraints.
```

Rename the file to system_tmr_bs.bmm (TMRed with block RAM scrubber)

2. Change the .bmm file to the triplicated version. The .bmm file in this example accommodates for the block RAM scrubber.

To create the triplicated .bmm file system_tmr_bs.bmm similar to the following:

```
ADDRESS_MAP ppc405_0 PPC405 100
  ADDRESS_SPACE plb_bram_if_cntlr_1_bram_combined_TR0 COMBINED [0xffff8000:0xffffffff]
    ADDRESS_RANGE RAMB16
      BUS_BLOCK
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[7].raminfr_inst/br[1].ramb16_s4_s4_inst [63:60];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[7].raminfr_inst/br[0].ramb16_s4_s4_inst [59:56];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[6].raminfr_inst/br[1].ramb16_s4_s4_inst [55:52];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[6].raminfr_inst/br[0].ramb16_s4_s4_inst [51:48];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[5].raminfr_inst/br[1].ramb16_s4_s4_inst [47:44];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[5].raminfr_inst/br[0].ramb16_s4_s4_inst [43:40];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[4].raminfr_inst/br[1].ramb16_s4_s4_inst [39:36];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[4].raminfr_inst/br[0].ramb16_s4_s4_inst [35:32];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[3].raminfr_inst/br[1].ramb16_s4_s4_inst [31:28];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[3].raminfr_inst/br[0].ramb16_s4_s4_inst [27:24];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[2].raminfr_inst/br[1].ramb16_s4_s4_inst [23:20];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[2].raminfr_inst/br[0].ramb16_s4_s4_inst [19:16];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[1].raminfr_inst/br[1].ramb16_s4_s4_inst [15:12];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[1].raminfr_inst/br[0].ramb16_s4_s4_inst [11:8];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[0].raminfr_inst/br[1].ramb16_s4_s4_inst [7:4];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR0/bram[0].raminfr_inst/br[0].ramb16_s4_s4_inst [3:0];
      END_BUS_BLOCK;
    END_ADDRESS_RANGE;
  END_ADDRESS_SPACE;

  ADDRESS_SPACE plb_bram_if_cntlr_1_bram_combined_TR1 COMBINED [0xffff8000:0xffffffff]
    ADDRESS_RANGE RAMB16
      BUS_BLOCK
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[7].raminfr_inst/br[1].ramb16_s4_s4_inst [63:60];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[7].raminfr_inst/br[0].ramb16_s4_s4_inst [59:56];
        plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[6].raminfr_inst/br[1].ramb16_s4_s4_inst [55:52];
```

```

plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[6].raminfr_inst/br[0].ramb16_s4_s4_inst [51:48];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[5].raminfr_inst/br[1].ramb16_s4_s4_inst [47:44];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[5].raminfr_inst/br[0].ramb16_s4_s4_inst [43:40];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[4].raminfr_inst/br[1].ramb16_s4_s4_inst [39:36];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[4].raminfr_inst/br[0].ramb16_s4_s4_inst [35:32];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[3].raminfr_inst/br[1].ramb16_s4_s4_inst [31:28];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[3].raminfr_inst/br[0].ramb16_s4_s4_inst [27:24];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[2].raminfr_inst/br[1].ramb16_s4_s4_inst [23:20];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[2].raminfr_inst/br[0].ramb16_s4_s4_inst [19:16];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[1].raminfr_inst/br[1].ramb16_s4_s4_inst [15:12];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[1].raminfr_inst/br[0].ramb16_s4_s4_inst [11:8];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[0].raminfr_inst/br[1].ramb16_s4_s4_inst [7:4];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR1/bram[0].raminfr_inst/br[0].ramb16_s4_s4_inst [3:0];
END_BUS_BLOCK;
    END_ADDRESS_RANGE;
    END_ADDRESS_SPACE;

ADDRESS_SPACE plb_bram_if_cntlr_1_bram_combined_TR2 COMBINED [0xffff8000:0xffffffff]
    ADDRESS_RANGE RAMB16
BUS_BLOCK
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[7].raminfr_inst/br[1].ramb16_s4_s4_inst [63:60];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[7].raminfr_inst/br[0].ramb16_s4_s4_inst [59:56];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[6].raminfr_inst/br[1].ramb16_s4_s4_inst [55:52];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[6].raminfr_inst/br[0].ramb16_s4_s4_inst [51:48];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[5].raminfr_inst/br[1].ramb16_s4_s4_inst [47:44];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[5].raminfr_inst/br[0].ramb16_s4_s4_inst [43:40];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[4].raminfr_inst/br[1].ramb16_s4_s4_inst [39:36];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[4].raminfr_inst/br[0].ramb16_s4_s4_inst [35:32];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[3].raminfr_inst/br[1].ramb16_s4_s4_inst [31:28];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[3].raminfr_inst/br[0].ramb16_s4_s4_inst [27:24];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[2].raminfr_inst/br[1].ramb16_s4_s4_inst [23:20];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[2].raminfr_inst/br[0].ramb16_s4_s4_inst [19:16];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[1].raminfr_inst/br[1].ramb16_s4_s4_inst [15:12];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[1].raminfr_inst/br[0].ramb16_s4_s4_inst [11:8];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[0].raminfr_inst/br[1].ramb16_s4_s4_inst [7:4];
plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[0].raminfr_inst/br[0].ramb16_s4_s4_inst [3:0];
END_BUS_BLOCK;
    END_ADDRESS_RANGE;
    END_ADDRESS_SPACE;
END_ADDRESS_MAP;

```

Step 9. Create a New ISE Project and Import the Triplicated Netlist (in EDIF Format) with the Other Dependant Files

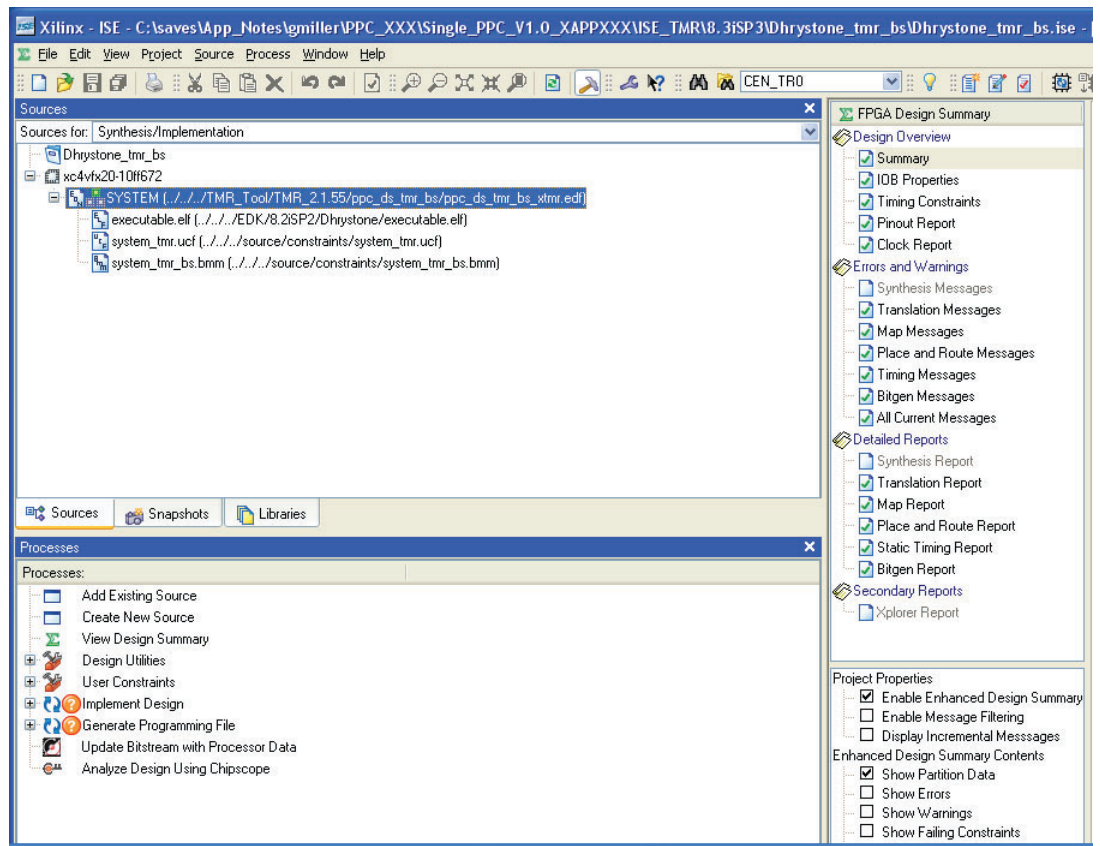
Use ISE software to make a project for the final place and route. This step involves importing the triplicated .edf netlist, triplicated .ucf, triplicated .bmm and the executable .elf file. It is also necessary to tell ngdbuild where the triplicated counter netlist resides.

1. Launch ISE software and create a new ISE project in the *project_install_dir/ISE_TMR/8.3iSP3*, and name the design *Dhrystone_tmr_bs*.
2. Select the device resident on the ML405 board (XC4VFX20-FF672-10C). The top-level source is EDIF.
3. Click **Next** until "Add existing source" appears, then add *ppc_ds_tmr_bs_xtmr.edf* from *project_install_dir/TMR_Tool/TMR_2.1.55/ppc_ds_tmr_bs* (the triplicated system netlist created from "Step 6. Triplicate the Embedded System Using the Xilinx TMRTool and Produce a Triplicated Netlist (TMRTool)," page 21. Uncheck **Copy to Project**, then click **Next** until the project is created.
- 4). Right-click the part and select **Add Source**, and select the following files (Figure 19):

```

project_install_dir/EDK/8.2iSP2/Dhrystone/executable.elf
project_install_dir/source/constraints/system_tmr.ucf
project_install_dir/source/constraints/system_tmr_bs.bmm

```

X1004_19_031408

Figure 19: Adding Source Files

- Expand **Implement Design** under “Processes”. Right-click on **Translate**, add `../../../../source/Netlists` to the macro search path, and then click **OK**.

Step 10. Place and Route to Create a .bit file to Configure the FPGA

After the final project is set up, this step runs place and route to generate the final .bit file for programming the target device. Programming the device ensures correct functionality and verifies that the final design meets the timing requirements.

- Produce a .bit file by double-clicking on **Generate Programming File**.

Note: If using this design with a configuration scrubber on another board (such as the XRTC radiation test board), be sure to select **Allow select map bit to persist**.
- Inspect the place-and-route timing report to verify timing was met.
- Run Impact and program the device as in the previous steps. Make sure the system is running by verifying that LEDs, DS14, DS12, and DS11, are blinking alternately with LEDs, DS15, DS4, and DS3. Also, using the RS-232 interface terminal as before, verify the Dhrystone output is running. Since this design is triplicated, there are six LEDs indicating Dhrystone status instead of the two as before. In addition, buttons GPIO_SW_W (SW7), GPIO_SW_C (SW6), and GPIO_SW_E (SW5) control the functionality of clock domains 0, 1, and 2 respectively.

Resource Evaluation

Table 3 one shows the system building blocks for the test design.

Table 3: Reference Design Resource Evaluation

Tested Design	PPC405	Reset Cntrl	Jtag_ppc	Plb_bus	Plb_gprio	Plb_BRAM cntrl	BRAMs	UART_Lite
Dhrystone Test	✓ converge	✓ in funcmon	✓ converge	✓ trip	✓ trip	✓ trip	✓ BR_scrubber	✓ trip

For the example design, the entire system except for the PPC was triplicated, resulting in a change in the resource utilization and timing versus a non-mitigated design.

Table 4 shows the comparison between the standard and mitigated designs in FPGA resource utilization. As for maximum frequency, a value of 100Mhz is used. This frequency setting is not the maximum possible, but was chosen for experimental and peripheral operational consistency purposes.

Table 4: Device Utilization

Tested Design	No. of FFs	No. of LUTs	No. of LUT-RAMs	No. of BRAMs	No. of BUFGCTRLs	No. of I/Os	No. of MULTs
Single Dhrystone design	935 (5%)	861 (5%)	0	16 (23%)	1 (3%)	7 (2%)	0 (0%)
Mitigated Dhrystone test design	2849 (16%)	3829 (22%)	0	48 (70%)	3 (9%)	13 (4%)	0 (0%)

Notes:

- The global clocks are triplicated for this design, but only one physical clock actually feeds the FPGA due to limitations of the ML405 board.

Smoke Testing and Verification of the Mitigated PPC System

The last phase of the design flow is to complete the smoke test of the mitigated PPC system (the example Dhrystone design used in this application note has the smoke test capability built in). A smoke test is just a simple way to test the mitigated system by disabling parts of the design to ensure the design still functions as intended. The user can simply shut off the clocks to the system using buttons on the ML405 board to accomplish this smoke test.

Verification of the block RAM scrubber can be completed in one of two ways, either:

- Load bogus data into the third copy of the block RAMs (domain 2). Then turn off the other system clock (domain 1). The system now votes on the block RAM output. If the block RAM scrubber is not running, then the system stops functioning because there is bad data arriving from domain 2 (bogus data) and from domain 1 (no clock).
- Verify the block RAM scrubber with ChipScope analyzer (refer to [Ref 9] for details). The example system contains the source code for a ChipScope analyzer inside the block RAM scrubber. The ICON and ILA cores reside in the `/Netlists` directory. Edit the `plb_bram_if_cntlr_1_bram_wrapper_tmr.vhd` file to uncomment the ICON/ILA core and signals to the triggers. Re-run the design. Evoke the ChipScope analyzer, and name the signals. Observe the block RAM scrubber in action (this is left as an exercise for the reader).

Perform the Smoke Test

- For a simple clock domain smoke test, press button SW6. This shuts off clock domain 1. Also, release SW6 and press SW5, shutting off clock domain 2.

Note: Pressing SW7 shuts off the main clock domain (feeding the PPC), and the system ceases to function. The processor stopping is the single point of failure in the mitigated single-PPC system.

- To test the block RAM scrubber, load the ISE design from “[Step 8. Create a Triplicated Version of .bmm,](#)” [page 23](#). Comment out the third block RAM in the .bmm file, causing the third copy of the block RAM to be loaded with zeros:

```
ADDRESS_MAP ppc405_0 PPC405 100
// ADDRESS_SPACE plb_bram_if_cntlr_1_bram_combined_TR2 COMBINED [0xfffff8000:0xffffffff]
// ADDRESS_RANGE RAMB16
// BUS_BLOCK
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[7].raminfr_inst/br[1].ramb16_s4_s4_inst [63:60];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[7].raminfr_inst/br[0].ramb16_s4_s4_inst [59:56];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[6].raminfr_inst/br[1].ramb16_s4_s4_inst [55:52];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[6].raminfr_inst/br[0].ramb16_s4_s4_inst [51:48];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[5].raminfr_inst/br[1].ramb16_s4_s4_inst [47:44];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[5].raminfr_inst/br[0].ramb16_s4_s4_inst [43:40];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[4].raminfr_inst/br[1].ramb16_s4_s4_inst [39:36];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[4].raminfr_inst/br[0].ramb16_s4_s4_inst [35:32];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[3].raminfr_inst/br[1].ramb16_s4_s4_inst [31:28];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[3].raminfr_inst/br[0].ramb16_s4_s4_inst [27:24];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[2].raminfr_inst/br[1].ramb16_s4_s4_inst [23:20];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[2].raminfr_inst/br[0].ramb16_s4_s4_inst [19:16];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[1].raminfr_inst/br[1].ramb16_s4_s4_inst [15:12];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[1].raminfr_inst/br[0].ramb16_s4_s4_inst [11:8];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[0].raminfr_inst/br[1].ramb16_s4_s4_inst [7:4];
// plb_bram_if_cntlr_1_bram/bram_scrubber_bram_TR2/bram[0].raminfr_inst/br[0].ramb16_s4_s4_inst [3:0];
// END_BUS_BLOCK;
// END_ADDRESS_RANGE;
// END_ADDRESS_SPACE;
```

- Rebuild the .bit file as in “[Step 9. Create a New ISE Project and Import the Triplicated Netlist \(in EDIF Format\) with the Other Dependant Files,](#)” [page 24](#) and reprogram the FPGA. The design should function normally.
- Press button (SW6) and verify the design is functioning.

If the block RAM scrubber is not working, pressing SW6 shuts down the system. In this case, having bad data in domain 2 and disabling clock 1 results in bad data from the block RAMs being voted, crashing the PPC. However, if the scrubber is running, the bad data in domain 2 is corrected. Now when clock 1 is disabled, correct data from the block RAM domains (0 and 2) is voted, and the PPC continues to function normally.

Improvements Under Consideration

There are some improvements under consideration by the Xilinx Radiation Test Consortium for addition in future releases:

- Add the ability to compensate for block RAMs other than one data-bit wide without code modifications (for example, as is done in the PLB version).
- Creating dual port RAM replacements, instead of just single port.
- Separate the A and B clocks for designs needing higher performance.

Conclusion

Xilinx FPGAs are an ideal fit for space applications in harsh environments due to the flexibility, size and reliability these robust devices provide, but like all devices containing memory elements, they are susceptible to SEUs and SETs. By using a combination of TMR and configuration scrubbing, designers can mitigate these errors.

This application note describes the process for building a mitigation scheme for an embedded hard-core PPC405 system targeted to Virtex-4 FX device. This solution is based on continuous external configuration scrubbing, functional surrounding logic triplication, and independent internal block RAM scrubbing (also triplicated) to build a robust PPC405-based system.

References

1. [UG156](#), *Xilinx TMRTool User Guide*.
2. [XAPP962](#), *Single-Event Upset Mitigation for Xilinx FPGA Block Memories*.
3. [XAPP216](#), *Correcting Single-Event Upsets through Virtex Partial Configuration*.
4. [XAPP779](#), *Correcting Single-Event Upsets in Virtex-II/-II Pro Devices through Partial Reconfiguration*.
5. [EDK user guides](#)
6. [UG210](#), *ML405 Evaluation Platform User Guide*.
7. [UG018](#), *PowerPC 405 Processor Block Reference Guide*.
8. [UG011](#), *PowerPC Processor Reference Guide*.
9. [UG029](#), *ChipScope Pro Software and Cores User Guide*.
10. [XAPP507](#), *Running the Dhrystone 2.1 Benchmark on a Virtex-II Pro PowerPC Processor*.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/14/08	1.0	Initial Xilinx release.

Notice of Disclaimer

Xilinx is disclosing this Application Note to you "AS-IS" with no warranty of any kind. This Application Note is one possible implementation of this feature, application, or standard, and is subject to change without further notice from Xilinx. You are responsible for obtaining any rights you may require in connection with your use or implementation of this Application Note. XILINX MAKES NO REPRESENTATIONS OR WARRANTIES, WHETHER EXPRESS OR IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL XILINX BE LIABLE FOR ANY LOSS OF DATA, LOST PROFITS, OR FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR INDIRECT DAMAGES ARISING FROM YOUR USE OF THIS APPLICATION NOTE.