

Teaching Materials

Introduction

MIPSfpga provides the RTL source code of the MIPS microAptiv UP core for implementation on an FPGA, together with teaching materials. The MIPS microAptiv UP core is a member of the same microcontroller family found in many embedded devices, including the popular [PIC32MZ](#) microcontroller from Microchip and Samsung's new [Artik1](#).

The teaching materials will show you how to use this core as part of a Computer Architecture course, paving the way for your students to explore how a commercial pipelined processor core works inside and to use this core in their projects, in effect creating their own SoC designs.

With its long heritage and excellent documentation, MIPS is the preferred choice of RISC architecture for many teachers around the world. But in the past, to demonstrate key concepts, teachers had to settle for creating partial 'MIPS-like' cores or using unofficial copies of dubious heritage. Not now! MIPSfpga is the real 'industrial' RTL, non-obfuscated, and available freely for academic use.

Structure

The MIPSfpga teaching materials consist of three parts:

The **Getting Started Package** contains a detailed guide that begins with a brief introduction to the MIPSfpga core included in the package. It gives a brief overview of how to setup the core for simulation or putting it on to an FPGA, as well as programming the processor. Guides on software installation are also given, along with detailed references about the core and its ISA – Instruction Set Architecture. All users need this package first.

MIPSfpga Fundamentals. In here you will find slides with accompanying lab scripts, illustrated using the Nexys 4 DDR and DE2-115 platforms. With this you will be taken from building the core, to programming in both c and assembly, with exercises to complete along the way. You then move on to adding a range of peripherals to the core to enable a greater level of interaction. The final example takes you through porting MIPSfpga to other FPGA boards such as Basys 3.

MIPSfpga SOC. The Advanced package enables you to run Buildroot Linux on MIPSfpga specifically using the Nexys4 DDR platform. The microAptiv core is packaged as an IP block usable by Vivado IP Integrator. As a result, AXI based IP blocks from Xilinx can easily be interfaced with the MIPS core. These are used to create an example SoC, such as a design with a UART and Ethernet, running under Linux, on MIPSfpga. A custom AXI GPIO block along with an example Linux driver is also provided. There is extensive documentation included. Collectively these provide an excellent basis for a SoC course that is highly relevant to the needs of the chip design industry, although the

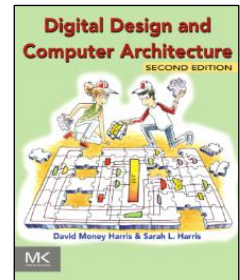
level of complexity makes this a postgrad class. PhD students and Postdocs will also find this material very useful for advanced projects.

Target Courses & Projects (Education Level)

- Digital Design & Microarchitectures (BSc)
- Computer Architecture, Advanced Computer Architecture (BSc, MSc)
- SoC design (MSc)
- Design Verification (MSc)
- Embedded Systems projects (BSc, MSc)
- Processor Architecture: modifications, enhancements, optimisation...(MSc, PhD)

The Authors

The course materials were developed by David Harris and Sarah Harris, co-authors of the popular textbook [Digital Design and Computer Architecture](#) which provides a uniquely relevant accompaniment to MIPSfpga.



Complementary Materials

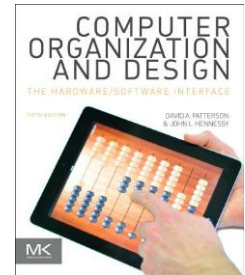
- The textbook 'Computer Organisation and Design' by David Patterson and John L. Hennessy remains the 'bible' for these activities, and provides further depth to Harris & Harris in a MIPSfpga-based course.

Other relevant textbooks are referenced here:

<http://community.imgtec.com/university/resources/books/?subject=mips-architecture>

Access the microAptiv core in silicon through boards such as [Digilent's 'WiFire'](#) incorporating Microchip's [PIC32MZ](#) MCU.

- Videos of the workshop given by Sarah Harris and Parimal Patel of Xilinx will be posted online in September 2015 here: <http://community.imgtec.com/university/video-gallery/>



Required Tools

Hardware

- Host PC: Windows 64 bit
- Digilent [Basys 3](#) or [Nexys 4 DDR](#), with Xilinx Artix FPGA
- Porting to other boards has been shown: Zed board, Nexys 3, Nexys 4 (not DDR), DE0-CV and SP605
- JTAG Probe: [SEED Studio MIPS Bus Blaster](#) including 14 to 6 pin adaptor (for Digilent boards)

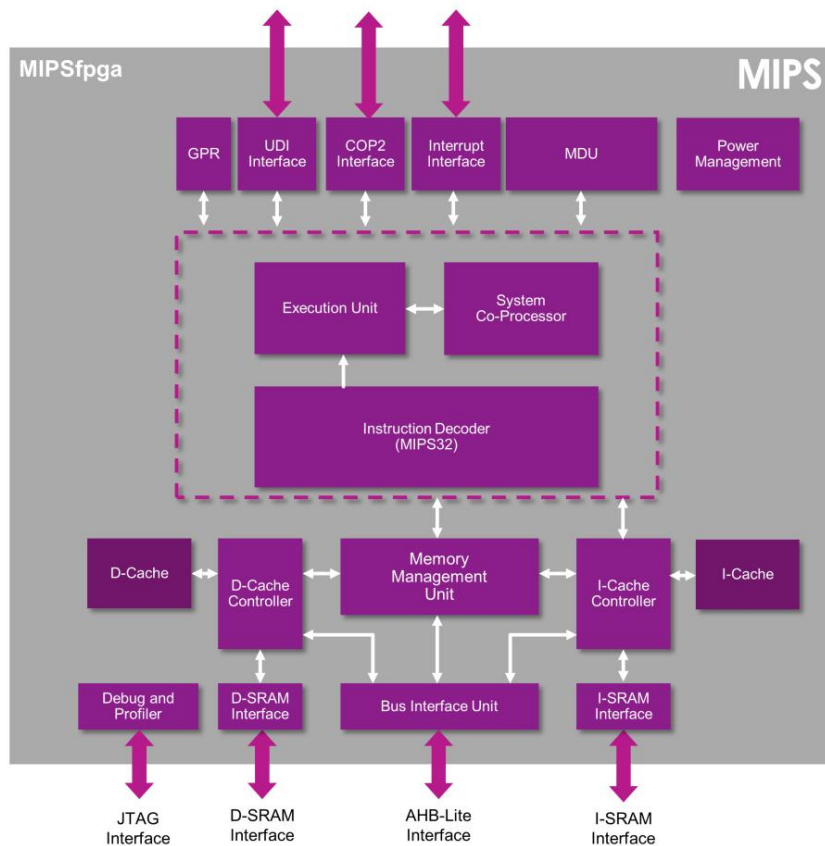
Software

- [Codescape MIPS SDK Essentials](#) (included with Getting Started Package)

- [Vivado \(Xilinx\) Web Pack edition](#)
- [Open OCD](#) (included with Getting Started Package)
- Mentor Graphics ModelSim ([Student](#) or [Full editions](#)) or Xilinx Xsim

Core Structure

The core is approximately 40K gates.



Languages available for Getting Started & Fundamentals

- English
- Simplified Chinese
- Japanese
- Russian
- Spanish

Support

- The MIPS insider forum [here](#) has a thread specifically for technical questions about MIPSfpga
- For curriculum and other discussions, there is the IUP (Imagination University Programme) forum [here](#)

Partners

We have worked closely with Xilinx and Digilent who have given wonderful support to this large and complex project.



Details on their University Programmes are here:
<http://www.xilinx.com/support/university.html>
<https://learn.digilentinc.com/list>

User Licenses

- For the **MIPS core**:
The agreement is part of the Getting Started Package download process, and acceptance is required before the download request can be submitted.
The End User Licence Agreement (EULA) allows the use of the MIPS core on FPGA platforms for the academic purposes of teaching, student projects and research. It allows teachers to distribute the core to students in classes, and it allows for the core to be modified. It does not allow the core to be put into silicon. Furthermore, if the core is modified and the user wishes to patent these modifications, the licence requires that this is negotiated with Imagination first.
The EULA is written in plain English, and a copy of the EULA is part of the Getting Started package for future reference.
- For the **Teaching Materials**:
The agreement is part of the Fundamentals and Advanced download process.
The End User Licence Agreement (EULA) explains that the materials are for Educational and Non-Commercial use, which means that companies or trainers who wish to use the materials for paid-for training, must seek Imagination's prior permission. Distribution of the materials to your Students is expressly allowed. The agreement allows extracts of the material to be used in derived teaching materials as long as Imagination's copyright is acknowledged, but publication in textbooks needs prior permission (which is usually given). No warranty is provided as to the effectiveness of the materials. The EULA is written in plain English, and a copy of the EULA is included in the materials package for future reference.

Plans

We are always pleased to hear about your needs for Teaching Materials. Our focus over the next year will be on workshops to promote use of MIPSfpga, releasing the Advanced package, debugging the existing materials, and listening to your feedback through the MIPSfpga thread on the forums. In addition, we are working on an 'MPW' route for researchers who would like to implement MIPS in silicon.

What we do after that will be determined primarily by you!

Press Release & Blogs

Free and Open Access to a Modern MIPS CPU

<http://imgtec.com/news/press-release/imagination-revolutionizes-cpu-architecture-education-with-free-and-open-access-to-a-modern-mips-cpu-3/>

MIPSfpga programme opens up the MIPS architecture to universities worldwide

<http://blog.imgtec.com/mips-processors/mipsfpga-opens-up-the-mips-architecture-to-universities-worldwide>

How to join the IUP and access these materials

1. Click 'Register' or 'Join IUP' on the landing page: www.imgtec.com/university
2. Complete the first section: 'the Community Registration'
3. Tick the box marked 'Join Imagination University Programme' and completes the additional information
4. A verification email will be sent to your inbox for activation.
(Please also check your spam mailbox because occasionally the mail will get filtered)
5. To download teaching materials, visit the IUP page -Resources
<http://community.imgtec.com/university/resources/>
6. Request the package(s) you want, accept the Licence Agreement, and give some details about how you plan to use the materials.
7. We then receive a request to approve the download, and normally action this within 48 hours. Once approved, you will receive an e-mail saying you can now make the download.

NOTE: Requests may be rejected for the following reasons

- The registration details are incomplete
- There are few or no details of intended use
- The requester appears to be a commercial company or a competitor

Please feel free circulate this information to anyone who might be interested and keep an eye on our webpages for further information such as workshops and updated packages.

MIPSfpga

by Imagination

Lab 1

Setting up a Vivado Project for MIPSfpga



These materials produced in association with Imagination.
Join our University community for more resources.

community.imgtec.com/university

MIPSfpga Lab 1: Setting up a Vivado Project

1. Introduction

This is the first in a series of laboratory exercises acquainting you with system-on-chip design using Imagination's MIPSfpga platform. In this lab you will learn to set up a Vivado project for simulating, synthesizing, and downloading the MIPSfpga system onto Digilent's Nexys4 DDR FPGA board. As you make changes to the MIPSfpga system in the future, you can follow these steps to compile, simulate, synthesize, download, and test your changes.

The instructions in this lab use Vivado 2015.1. Instructions for later versions of Vivado are similar, if not exactly the same.

2. Setting up a Vivado Project

In this section we walk through the steps of (1) creating a project for the MIPSfpga system, (2) simulating the project, (3) compiling the project, and (4) downloading the MIPSfpga system onto the Nexys4 DDR board.

Before setting up the Vivado project, make a copy of the MIPSfpga system by copying the **rtl_up** folder in the MIPSfpga directory (provided with the MIPSfpga Getting Started materials) to **MIPSfpga_Fundamentals\rtl_up**.

The Verilog files in the **MIPSfpga_Fundamentals\rtl_up** directory describe the MIPSfpga system and are the design source files for the Vivado project you are about to create. In later MIPSfpga Fundamentals labs, you will extend the functionality of the MIPSfpga system by both modifying and adding Verilog files to the MIPSfpga_Fundamentals\rtl_up folder.

Step 1. Create Vivado project

Start Vivado. Open a new project by choosing **File** → **New Project** (see [Figure 1](#)).

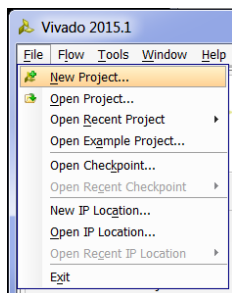


Figure 1. Create new Vivado project

Click **Next**. Browse to the MIPSfpga_Fundamentals\Xilinx\Lab01_Vivado folder and place the new project, **Project1**, in that folder, as shown in [Figure 2](#). Click the Create Project subdirectory box and click next, as shown below.

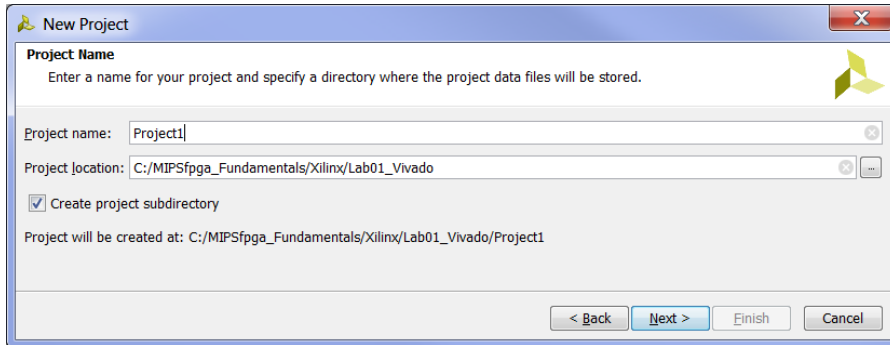



Figure 2. Create Vivado project directory

In the next window, leave **RTL Project** selected and click **Next**.

Now in the **Add Sources** window, click on the green plus sign  and **Add Files...** Browse to the **MIPSfpga_Fundamentals\rtl_up** directory. Select all of the files (ctrl-a or click, shift-click), as shown in [Figure 3](#), and click **OK**.

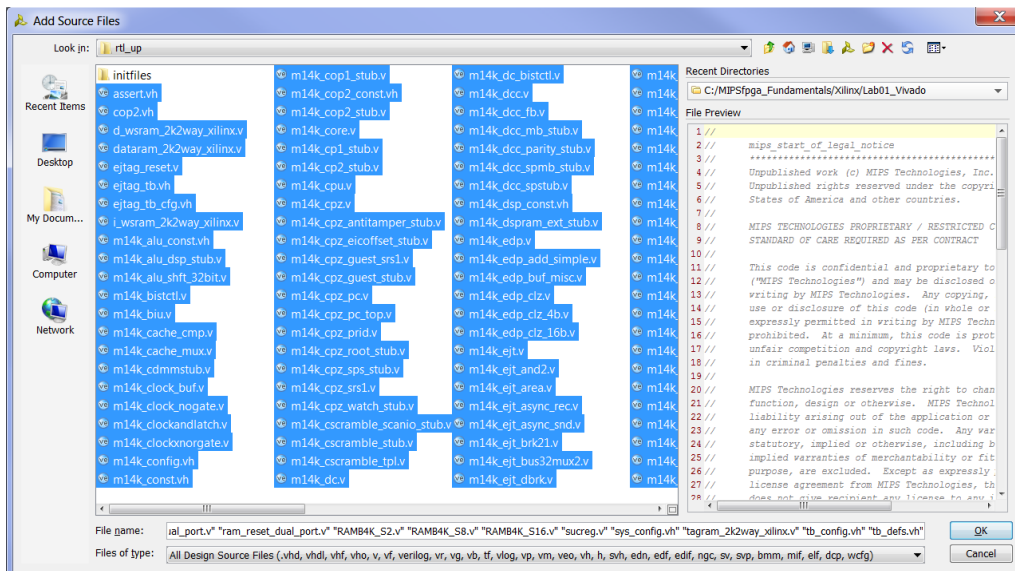


Figure 3. Adding Verilog files to project

The project should refer to the Verilog (.v) and Verilog header (.vh) files located in the MIPSfpga_Fundamentals\rtl_up directory – you do **not** want to make a local copy of the files in

the Vivado project. So, in the next window, make sure the Copy the sources into Project box is **not** selected (see [Figure 4](#)) and click **Next**.

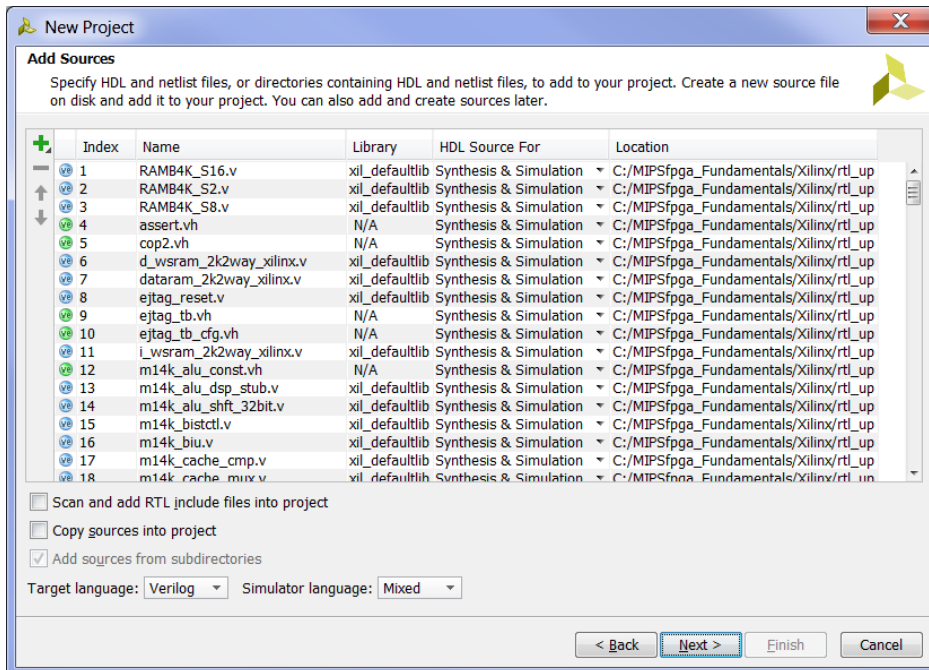



Figure 4. Adding Sources – do not copy sources into project

You will not add any IP, so click **Next** in the Add Existing IP (optional) window.

In the Add Constraints (optional) window, click on  and **Add Files...** Browse to the MIPSfpga_Fundamentals\Xilinx\Lab01_Vivado directory. Select **mipsfpga_nexys4_ddr.xdc** and click **OK** (see [Figure 5](#)). This constraints file maps the Verilog signals to pins on the FPGA and describes timing constraints.

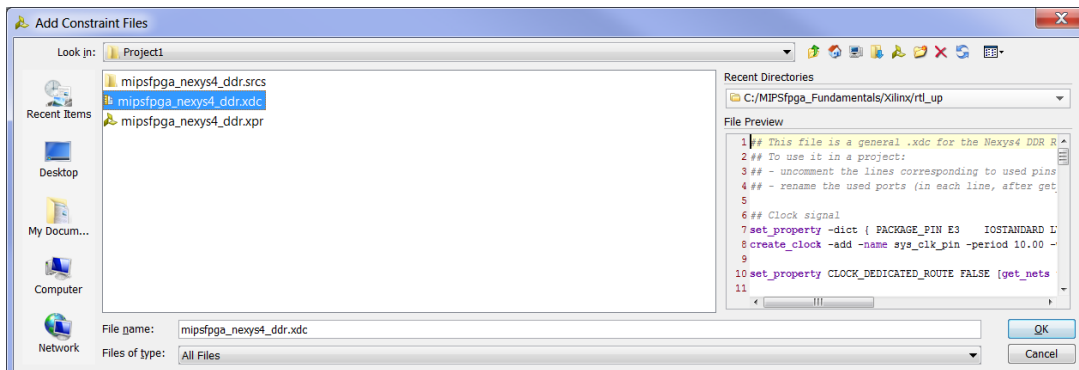


Figure 5. Add Xilinx Design Constraints (.xdc) file to Vivado project

Click on the Copy constraints files into project box (see [Figure 6](#)) and click Next.

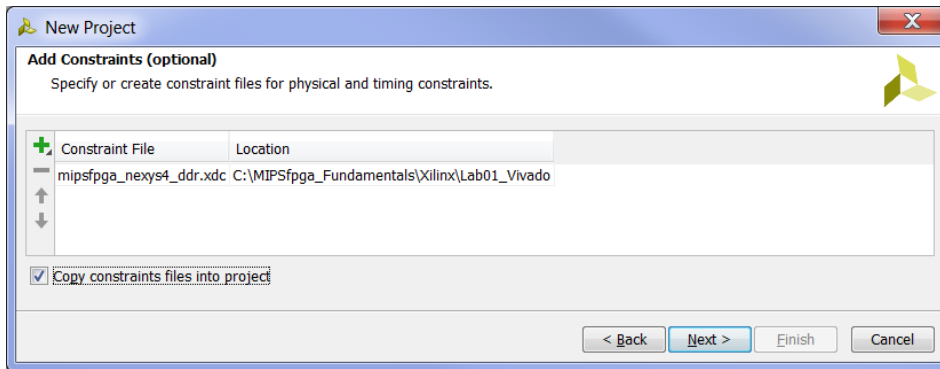


Figure 6. Copy .xdc file into Vivado project

Now you will choose the Artix-7 FPGA that is on the Nexys4 DDR board as the target. Type (or copy-paste) the following into the search box: **xc7a100tcsg324-1**, as shown in Figure 7. Select the part, as shown, and click **Next**.

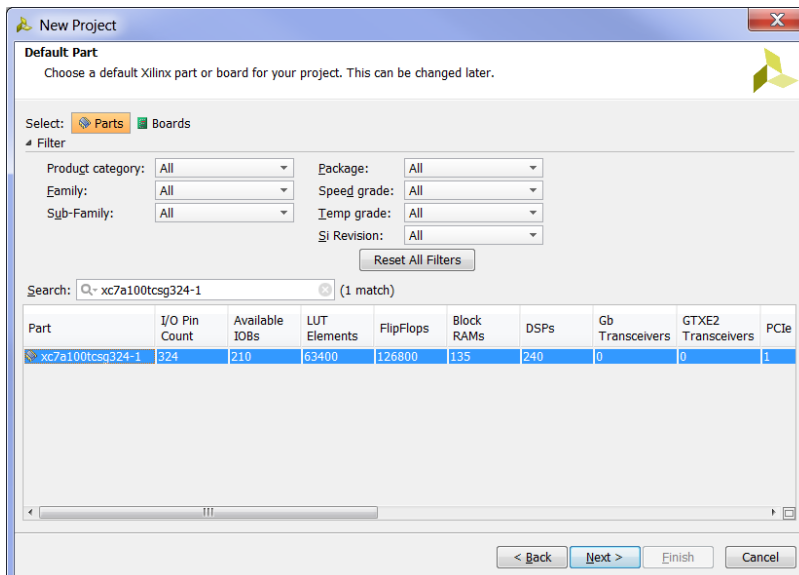


Figure 7. Selecting the Artix-7 FPGA

"xc7a" indicates that it is an Artix-7 FPGA. "100t" says that it has about 100k Logic Cells. "csg324" indicates a "chip scale ball grid array (BGA)" package with 324 pins, and "-1" is the speed grade.

Now click **Finish** in the New Project Summary window (see Figure 8).

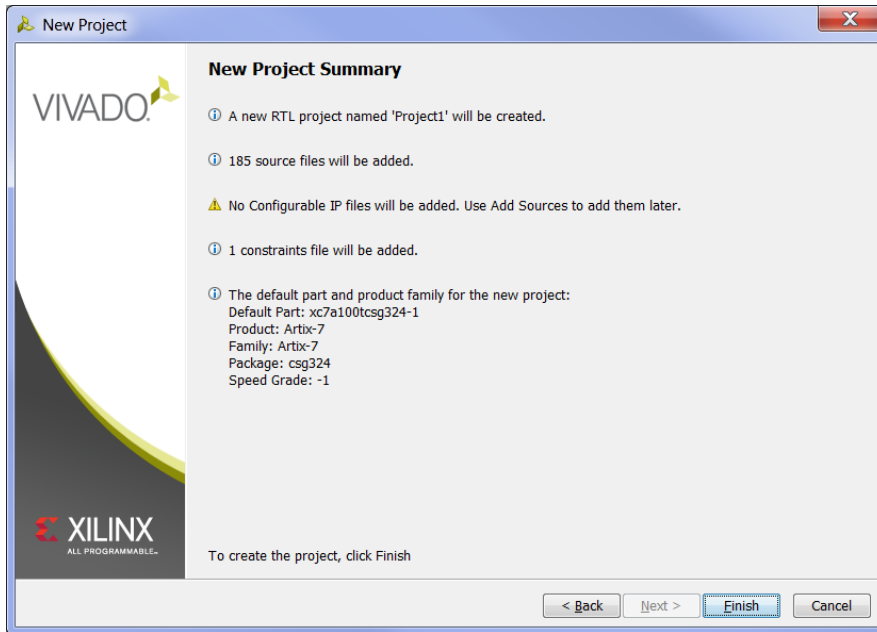


Figure 8. New Project Summary window

After the project initializes, go to the Project Manager window, right-click on `mipsfpga_nexys4_dds`, and select **Set as Top** in the pull-down menu, as shown in Figure 9. This will set that module as the top-level module to synthesize, compile, and download to the Nexys4 DDR board.

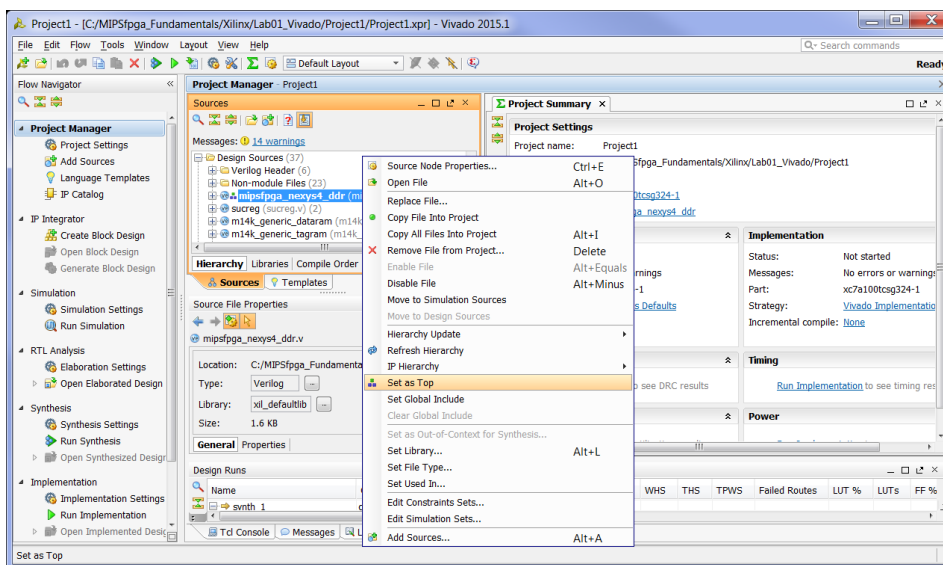


Figure 9. Setting the top-level module for synthesis, implementation, and bitstream generation

The last step of creating the project is to add a PLL that reduces the on-board 100 MHz clock to 50 MHz to meet timing constraints. To create the PLL, click on **IP Catalog** under Project Manager in the Flow Navigator, as shown in Figure 10.

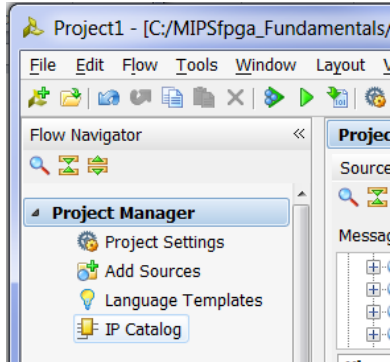


Figure 10. IP Catalog

Now, in the IP Catalog tab of the Project Manager pane, expand **FPGA Features and Design**, and then expand **Clocking**. Double-click on **Clocking Wizard**, as shown in Figure 11.

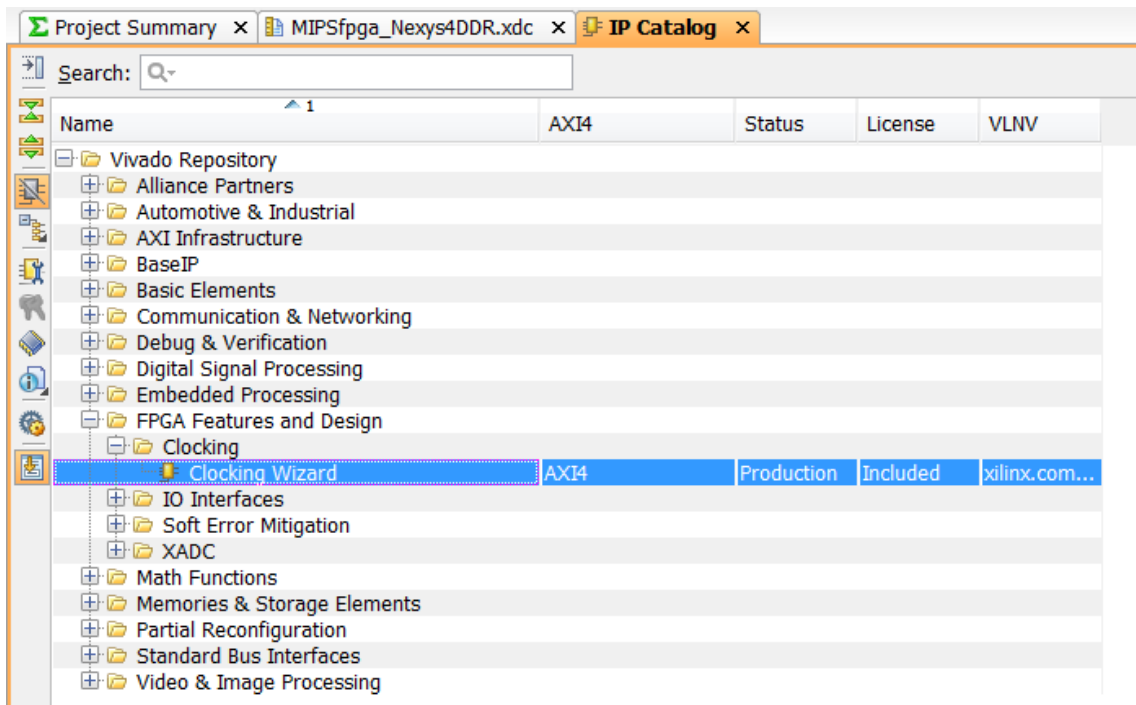


Figure 11. Clocking Wizard

The Clocking Wizard window will pop up, as shown in Figure 12. Select **PLL**, as shown in Figure 12. Leave the Input Clock information as the default (100 MHz).

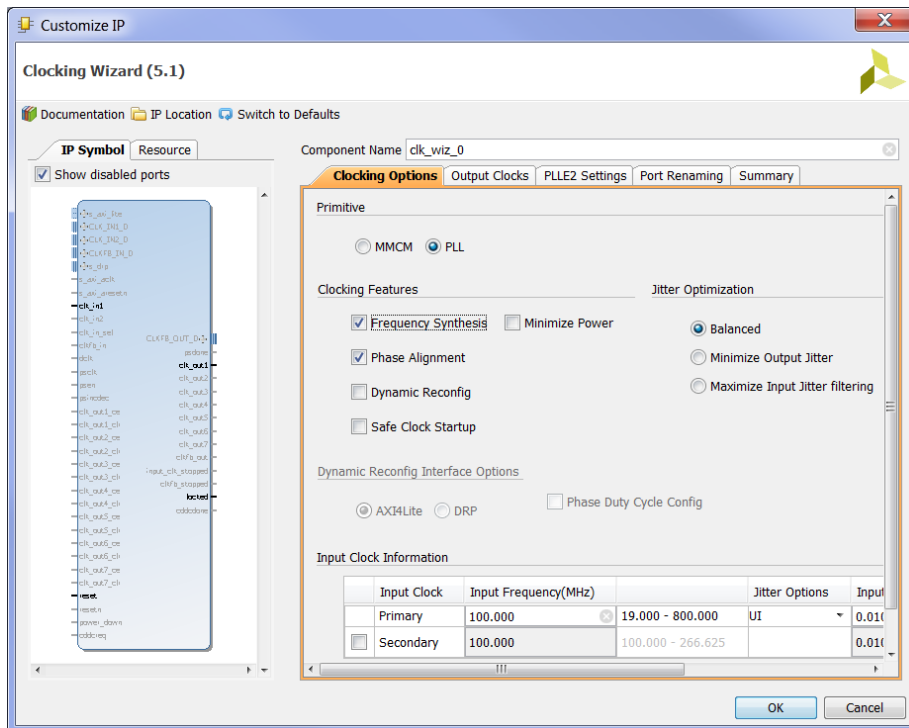


Figure 12. Clocking Wizard window

Now click on the **Output Clocks** tab, and type in **50** as the output frequency in the Output Freq (MHz) Requested box for clk_out1, as shown in Figure 13.

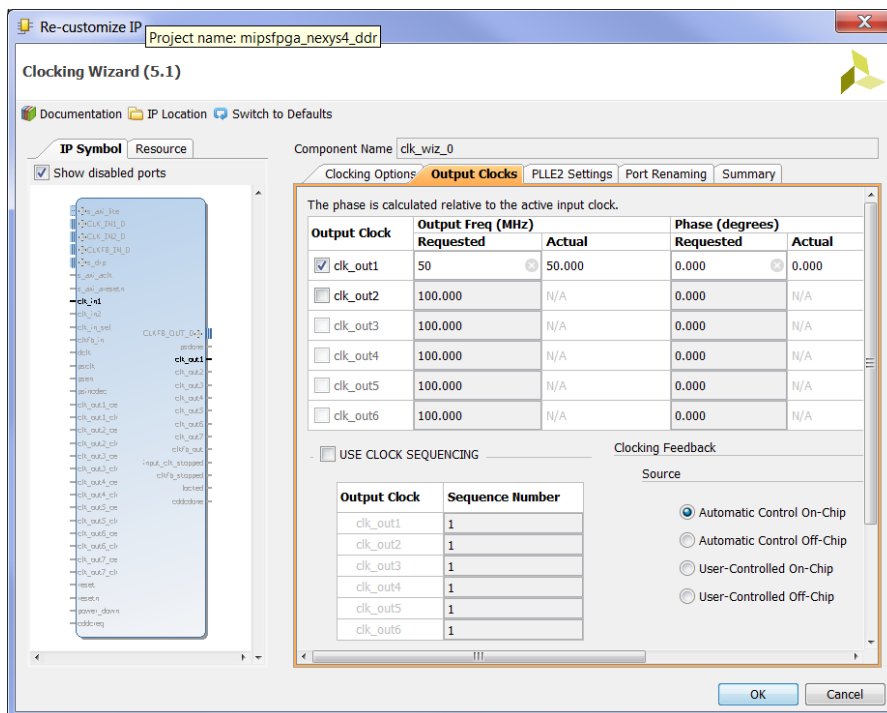


Figure 13. Select output clock frequency

Scroll down in the same tab (Output Clocks) and **deselect** reset and locked, as shown in Figure 14. Then click **OK** to complete the creation of the PLL.

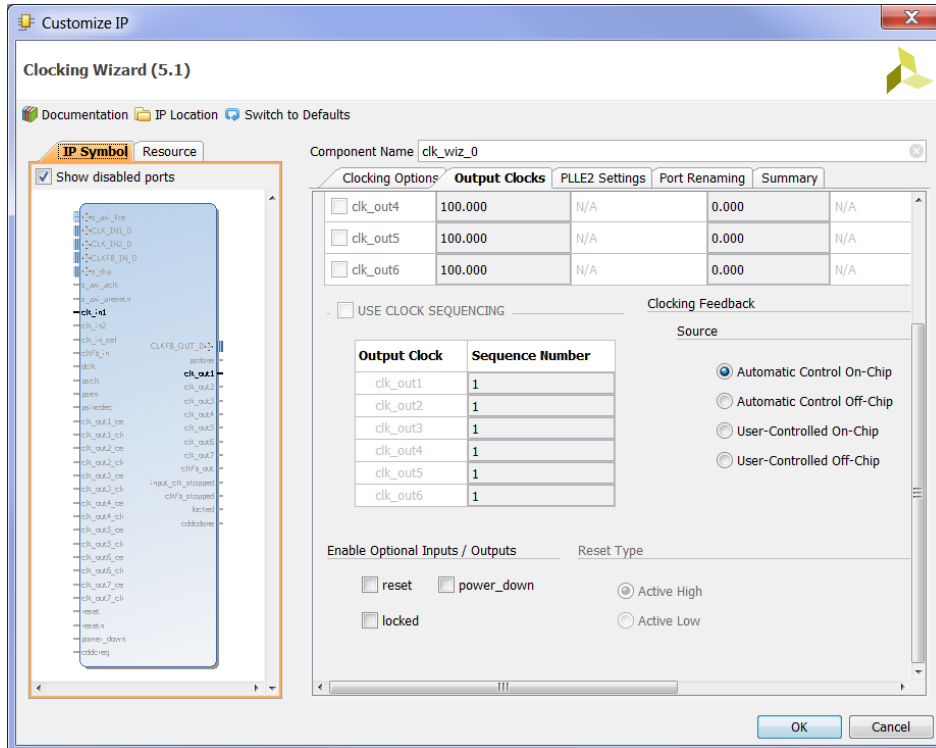


Figure 14. Deselect reset and locked

A pop-up window will prompt you to "Generate Output Products", as shown in Figure 15. Click **Generate**.

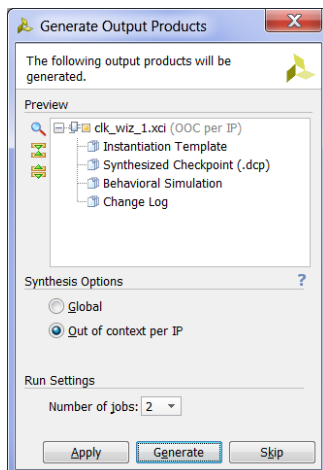


Figure 15. Generate PLL

A window will pop up that says "Out-of-context module run was launched for generating output products," as shown in [Figure 16](#). Click **OK**.



Figure 16. Out-of-context generation of PLL

Step 2. Simulating MIPSfpga

Now you are ready to simulate the MIPSfpga system. You will use Vivado's built-in simulator called XSIM. We already added the testbench.v file when we created the project, and now we will make it the top-level module for simulation. In the Project Manager panel, scroll down to **Simulation Sources** and expand it and the **sim_1** folder.

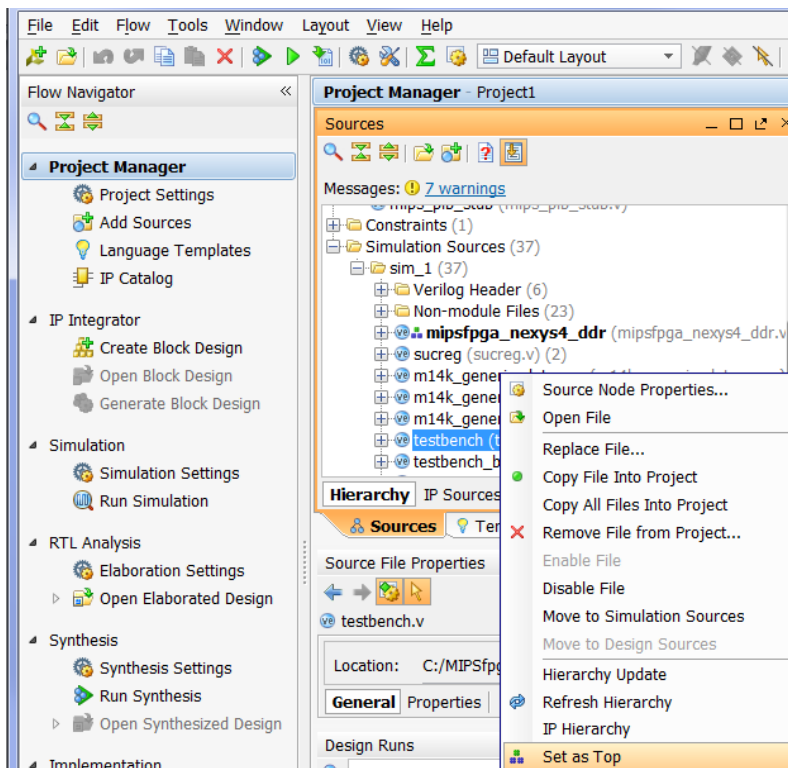



Figure 17. Setting the top-level module for simulation

Right-click on testbench.v and **set it as the top-level module** for simulation, as shown in [Figure 17](#). Notice that the testbench entry is now bold.

Click on **Add Sources** in the Flow Navigator window on the left, select **Add or create simulation sources** option, and then click **Next**. Click on  and **Add Files...**, select **All Files** in the *Files of type* filter, browse to the ram_reset_init.txt file in MIPSfpga_Fundamentals\rtl_up\initfiles\1_IncrementLEDs, select it, and click **OK**. Leave the Copy sources into project box unselected, but leave the Include all design sources for simulation box checked, as shown in [Figure 18](#). Then click **Finish**.

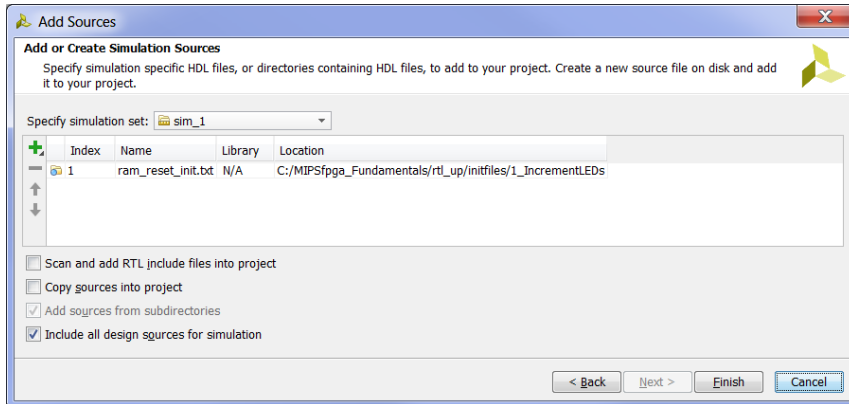


Figure 18. Adding simulation source

The text file contains the instructions that will be loaded into MIPSfpga's memory. As you recall from the MIPSfpga Getting Started Guide, this program, as shown in [Figure 19](#) for your convenience, writes incremented values to memory address 0xbf800000. Also recall from the MIPSfpga Getting Started Guide that the LEDs on the Nexys4 DDR board are mapped to memory address 0xbf800000. So the program writes incremented values to the LEDs.

<pre>// C code unsigned int val = 1; volatile unsigned int* dest; dest = 0xbf800000; while (1) { *dest = val; val = val + 1; }</pre>	<pre># MIPS assembly code # \$9 = val, \$8 = mem address 0xbf800000 addiu \$9, \$0, 1 # val = 1 lui \$8, 0xbf80 # \$8=0xbf800000 L1: sw \$9, 0(\$8) # mem[0xbf800000] = val addiu \$9, \$9, 1 # val = val+1 beqz \$0, L1 # branch to L1 nop # branch delay slot</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 19. IncrementLEDs program

The equivalent machine code for the IncrementLEDs program is given in [Figure 20](#).

Machine Code	Instruction Address	Assembly Code
24090001	// bfc00000:	addiu \$9, \$0, 1 # val = 1
3c08bf80	// bfc00004:	lui \$8, 0xbf80 # \$8=0xbf800000
ad090000	// bfc00008:	L1: sw \$9, 0(\$8) # mem[0xbf800000] = val
25290001	// bfc0000c:	addiu \$9, \$9, 1 # val = val+1
1000ffffd	// bfc00010:	beqz \$0, L1 # branch to L1
00000000	// bfc00014:	nop # branch delay slot

Figure 20. MIPS machine code

Expand the hierarchy under *Simulation Sources* and observe that *ram_reset_init.txt* is added in a separate sub-folder called *Text* and it contains the machine code (see Figure 21). Now you are ready to run the simulation of the MIPSfpga system running that program.

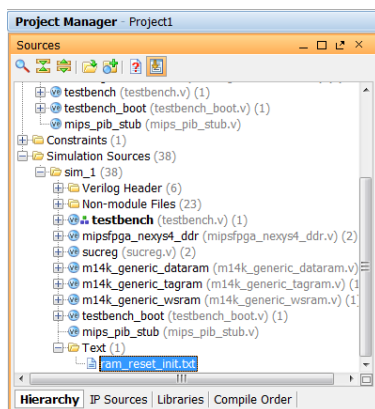


Figure 21. Text file as simulation source

Click on **Simulation Settings** in the Flow Navigator window on the left, as shown in Figure 22.

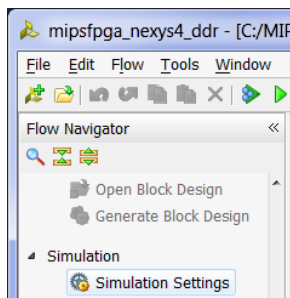


Figure 22. Simulation Settings

The simulation settings window will show up, as shown in Figure 23. Make sure the **Generate scripts only** is **not** selected. Click on the **Simulation** tab and set **the simulation run time to 2000 ns**. Click **OK**.

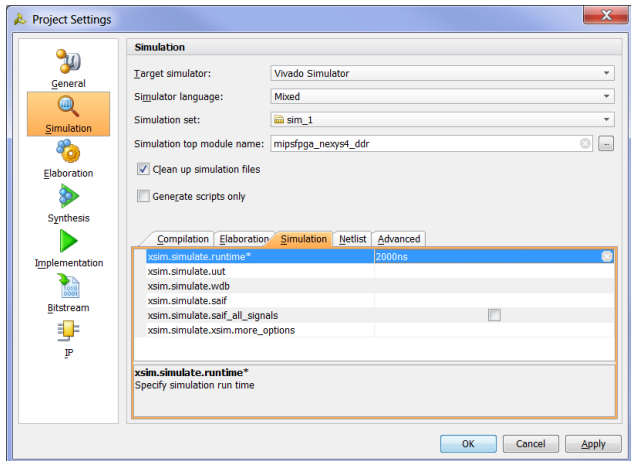


Figure 23. Change simulation run time

Click on **Run Simulation** → **Run behavioral simulation** in the Flow Navigator window, as shown in Figure 24.

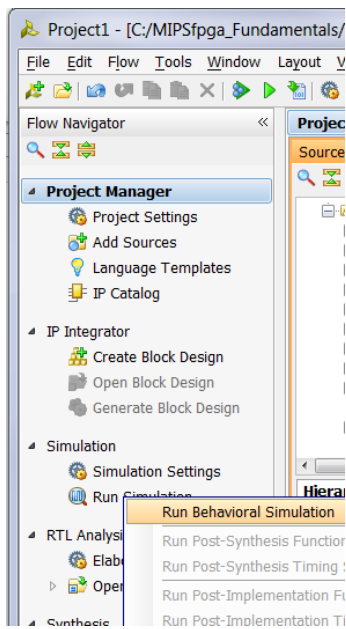



Figure 24. Run simulation

The testbench and lower-level modules will compile, the simulation window will open, and the simulation results will be displayed, as shown in Figure 25. You will see the top-level signals being displayed. Click on the Zoom Fit button ().

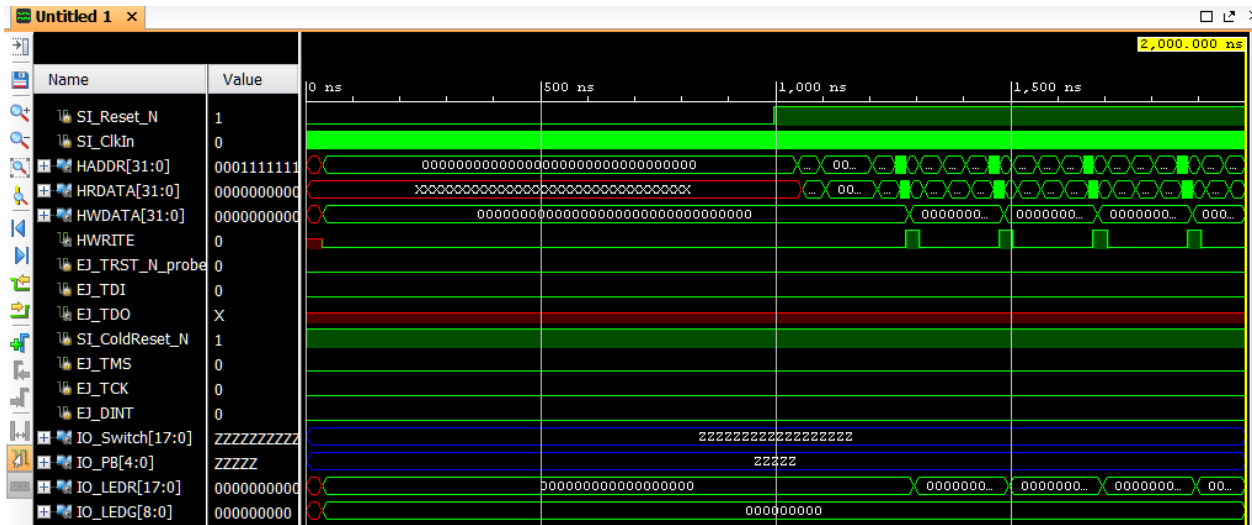


Figure 25. Simulation results showing top-level signals

Select all of the signals in the waveform window and then right-click and select **Radix** → **Hexadecimal**. Use shift-click and ctrl-click to select multiple signals at a time.

Delete all of the EJTAG signals and some of the I/O signals: EJ_TRST_N_probe, EJ_TDI, EJ_TDO, SI_ColdReset_N, EJ_TMS, EJ_TCK, EJ_DINT, IO_Switch, IO_PB, and IO_LEDG. Do **not** delete IO_LEDR. Right-click on a signal (or group of signals) and select Delete. (Or simply select the signals and press the Delete key.) Again, you can also select multiple signals using shift-click and ctrl-click.

The waveform window will now resemble Figure 26.

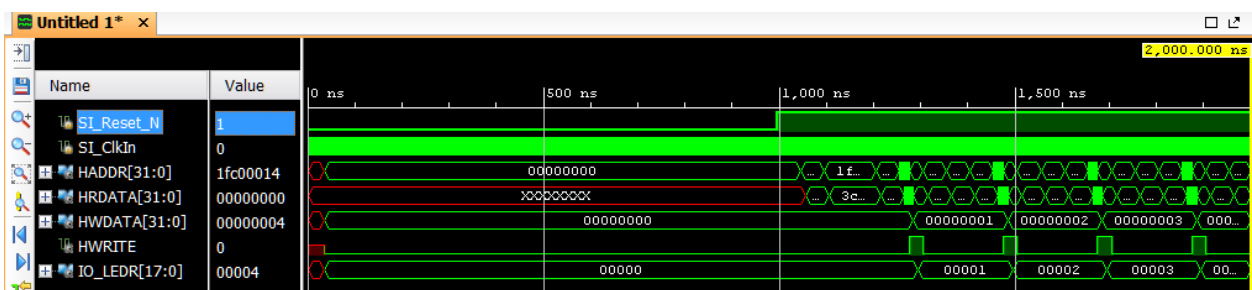


Figure 26. Keeping only the desired top-level signals

You can float the waveform window by clicking on the float button (☐) and then maximize it by clicking on the full size button (☐). Click on the Zoom Fit button to see the waveform completely. You can also use Zoom In (🔍), Zoom Out (🔍), and Zoom to Cursor (📍) buttons to view a desired section of the waveform.

You can view signals from lower-level modules by adding them to the waveform. For example, as shown in Figure 27, expand the **testbench** hierarchy to **testbench** → **sys** → **mipsfpga_ahb** → **mipsfpga_ahb_ram_reset** to see the **ram_reset_dual_port** entry in the **Scopes** window. Click on the **ram_reset_dual_port** entry to see the corresponding signals in the **Objects** window.

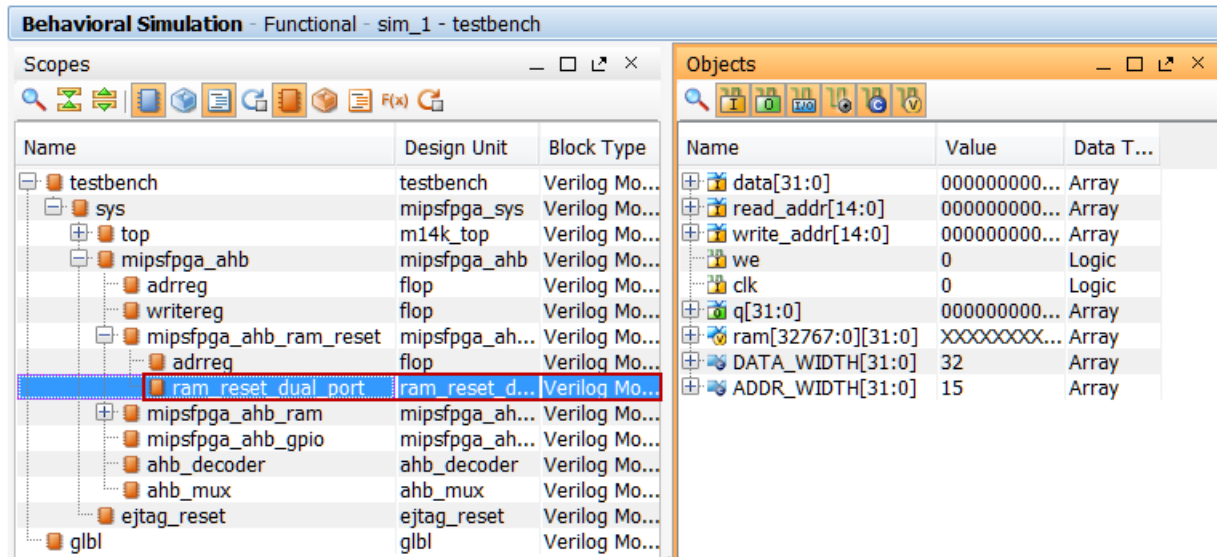
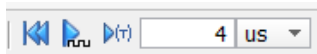


Figure 27. Accessing the signals of lower-level modules

In the waveform window, right-click in the signals area below the last signal, and select **New Divider**. The New Divider dialog box will appear. Type **Reset RAM Memory** in the field and press **Return**.

Select all of the objects in the **Objects** window, right-click and select **Add to Wave Window** and observe that the signals are added to the Waveform window. You can change the radix of the added signals to Hexadecimal as before. In the tool buttons bar, change the run time to 4 us



Now click on the Restart button , and then click on the Run for

<time> button  to reset and run the simulation for 4 us. You will see the output as shown in

Figure 28.

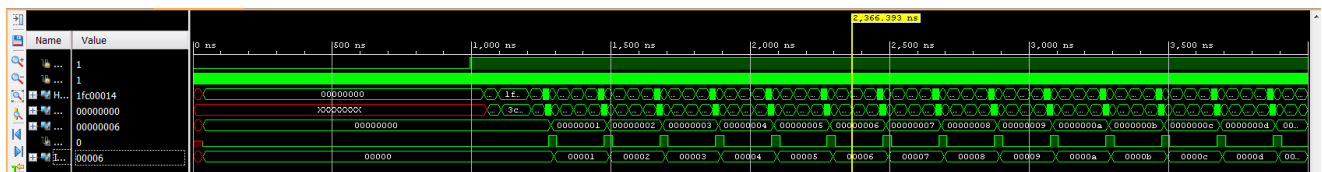



Figure 28. Simulation result showing lower-level module signals

At first the processor is reset: the low-asserted reset signal **SI_Reset_N** is low. Just after reset (when **SI_Reset_N** transitions from 0 to 1), you can view the waveform as it fetches each

instruction starting with instruction address 0x1fc00000. This address shows up on HADDR and the instruction read from memory appears on the HRDATA bus one cycle later. Recall that virtual address 0xbfc00000 translates to physical address 0x1fc00000. The instructions are executed in sequence until the branch is taken at 0xbfc00010. The code then continuously repeats from 0xbfc00008 – 0xbfc00014. Recall from the MIPSfpga Getting Started Guide, that until the caches are initialized by the boot code, each instruction takes 5 cycles. Also view how incremented values are written on the HWDATA signal as the code executes. HWDATA is the data being written to memory or, in this case, memory-mapped I/O. IO_LEDR displays the incremented value because it is memory-mapped to address 0xbf800000. The IO_LEDR signals are connected to the pins that drive the Nexys4 DDR's LEDs.

After you are finished viewing the waveform, you can close the simulator by selecting **File** → **Close Simulation**. A pop-up window will appear asking if you want to save the waveform. You could select Save but for now, click **Discard**.

Step 3. Compiling MIPSfpga

Now you are ready to compile the MIPSfpga system and create a bitfile that you can download onto the Artix-7 FPGA on the Nexys4 DDR board. Click on the **Generate Bitstream** button  at the top of the window. The bitstream is a file that configures the FPGA to be the MIPSfpga system, as defined by the Verilog files. This file is also referred to as the *bitfile*.

A window may pop up saying:

```
There are no implementation results available. OK to launch  
synthesis and implementation?...
```

Click **Yes**. Now wait for synthesis, placement, routing, and bitstream generation to complete. This typically takes around 10-20 minutes, depending on your computer speed.

After the bitstream has been generated, you will see the Bitstream Generation Completed pop-up window, as shown in [Figure 29](#).

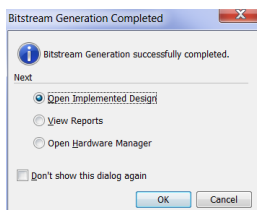


Figure 29. Bitstream Generation Completed pop-up window

Viewing the implemented design is optional, but gives some insight into the timing and layout of the MIPSfpga core. (If you don't want to view it, select Open Hardware Manager and click OK. Then continue with Step 4 below.) To view the implemented design, leave **Open Implemented Design** selected, and click **OK**. This will take a few minutes. Then the Implemented Design window will open, as shown in Figure 30.

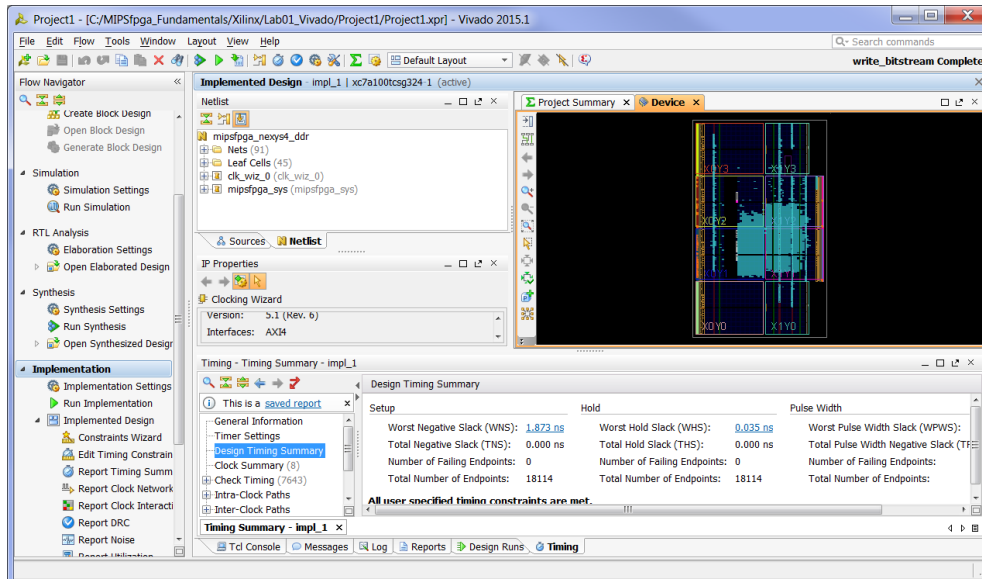


Figure 30. Implemented Design

Notice the **Design Timing Summary** pane at the bottom. Most importantly, it says that **All user specified timing constraints are met**. The Worst Negative Slack (**WNS**) is 1.273 ns and the Worst Hold Slack (**WHS**) is 0.035 ns, so there are no timing violations. The slack values for your project are likely slightly different. Each time Vivado places and routes the design, a different configuration results.

As you add to or modify the MIPSfpga system in the future, check that the timing constraints are met, and if not, reduce the frequency of the PLL and/or change the timing constraints in the Xilinx Design Constraints (.xdc) file until they are. The WNS will indicate how much the cycle time needs to be increased.

For example, [Figure 31](#) shows a Project Summary page for a design that does **not** meet timing. Notice the negative values of WNS (shown in red). In this case the cycle time is too short by 4.91 ns, so add that amount (or slightly more) to the cycle time to meet timing. If the frequency of the failing design were 100 MHz (cycle time = 1/100 MHz = 10 ns), the cycle time would need to increase to at least (10 + 4.91) ns ≈ 15 ns (frequency = 1/15 ns ≈ 66 MHz).

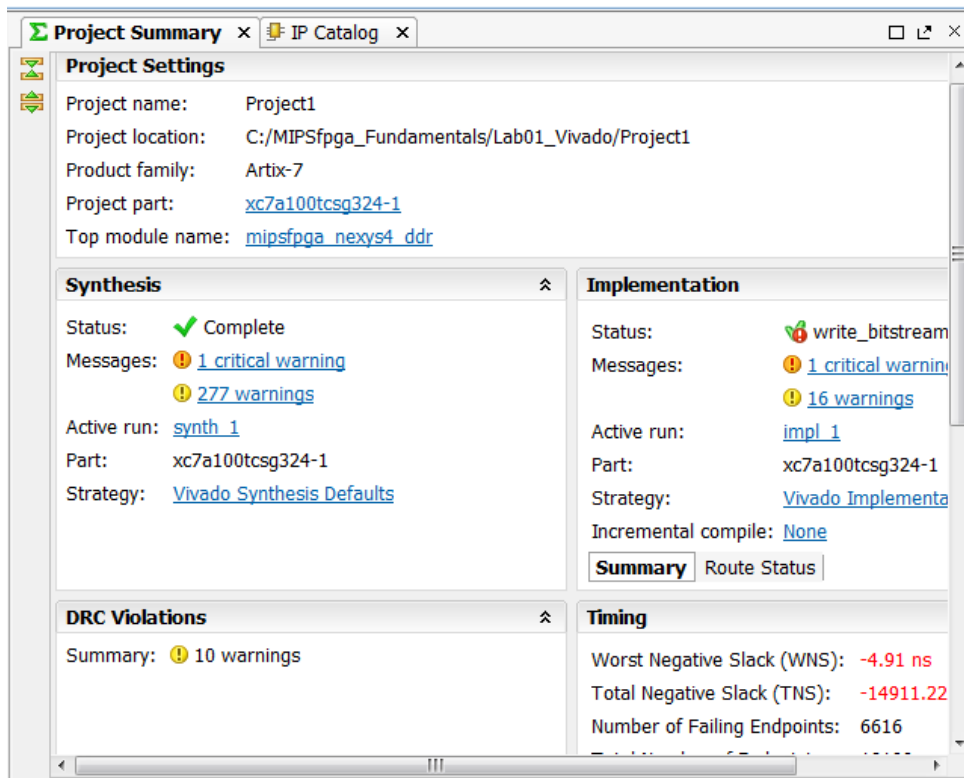


Figure 31. Design that does not meeting timing constraints

Step 4. Downloading MIPSfpga onto Nexys4 DDR FPGA Board

Now you are ready to download the compiled design onto the Nexys4 DDR FPGA board. First, connect and turn on the Nexys4 DDR board. Figure 32 shows the board and highlights the power switch and the USB port. Plug the standard end of the programming cable into your computer and the micro-USB end of the programming cable into the board, at the location labeled "USB Programmer Port" in Figure 32. Now turn the board's power switch ON. If the board is factory configured, the board will run a pre-loaded program that writes to the 7-segment displays with a snake-like pattern that repeats indefinitely. To program the board, it can be in QSPI mode (as shown in Figure 32 with the left-most Mode pins connected by a jumper) or in JTAG mode (with the two middle pins of the Mode selector connected by a jumper).

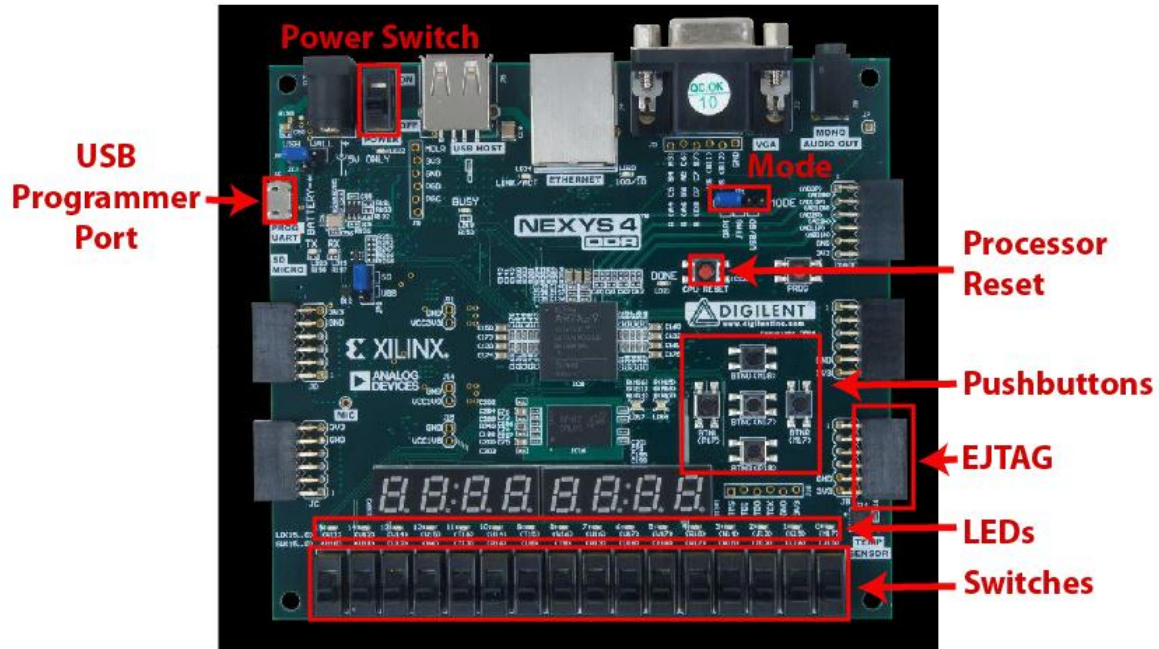


Figure 32. Nexys4 DDR board (photograph © Digilent Inc., 2015)

In the Vivado window, select **Flow** → **Open Hardware Manager**, as shown in Figure 33.

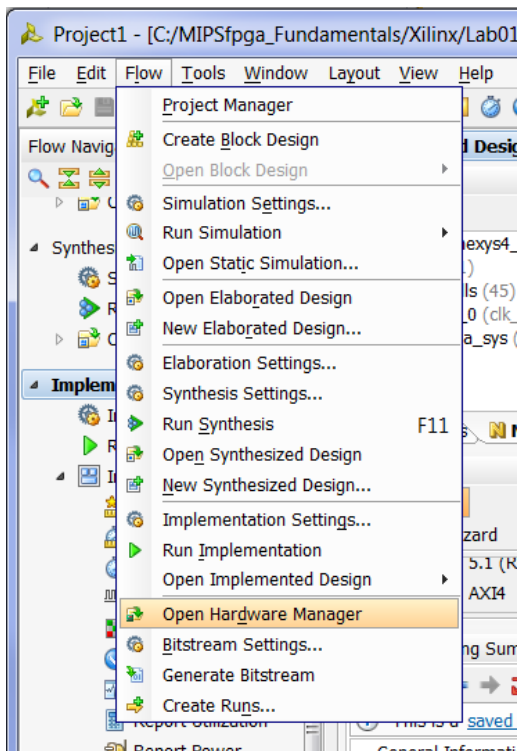


Figure 33. Open Hardware Manager

The Hardware Manager window will open. Now click on **Open Target** and choose **Auto Connect**, as shown in Figure 34.

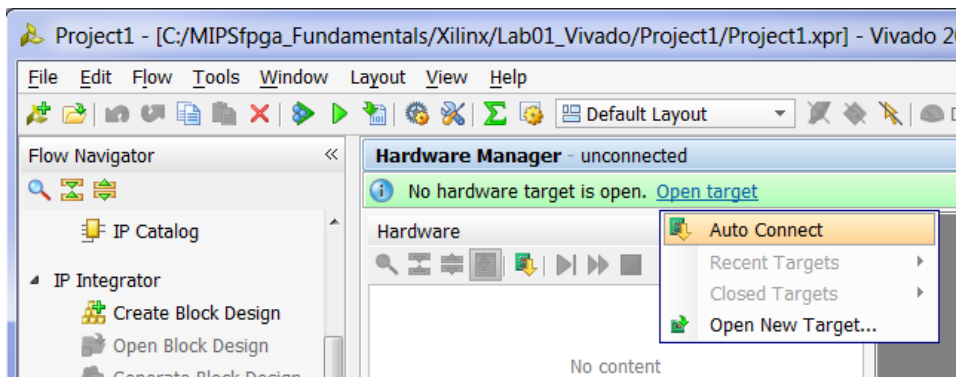


Figure 34. Hardware Manager window, Auto Connect

After you click on *Auto Connect*, Vivado takes several seconds to connect to the target FPGA on the Nexys4 DDR board. You will see the following warning, that you can ignore:

```
WARNING: [Labtools 27-3123] The debug hub core was not detected
at User Scan Chain 1 or 3. ...
```

If you see the message "No hardware target is open," two causes are most likely:

1. You forgot to plug in the Nexys4 DDR board into your computer and/or turn it on.
or
2. You need to install/reinstall the driver for the Nexys4 DDR board's USB programmer cable. Refer to instructions in the MIPSfpga Getting Started Guide if you need help reinstalling the driver.

Now click on **Program device** and select **xc7a100t_0**, as shown in Figure 35.

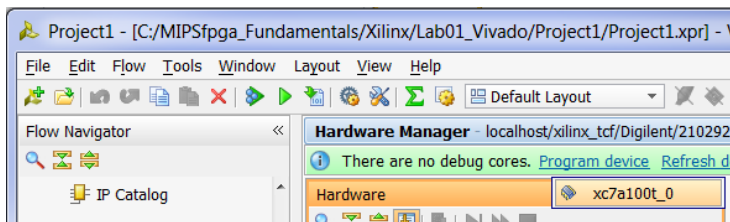


Figure 35. Selecting Program device

The Program Device window will open, as shown in Figure 36. The Bitstream file box should autopopulate, but if it does not, choose:

```
MIPSfpga_Fundamentals\Xilinx\Lab01_Vivado\Project1\Project1.runs\impl_1\mipsfpga_nexys4_ddr.bit
```

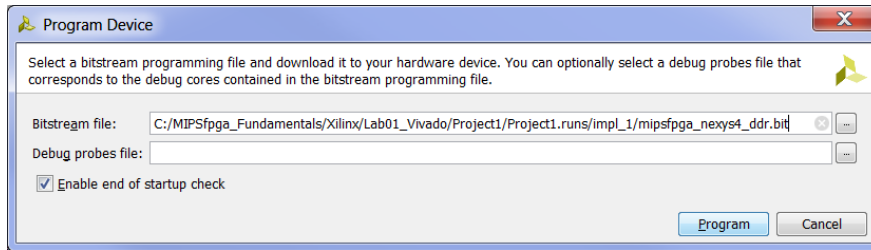


Figure 36. Program device with bitstream file

Leave the **Enable end of startup** box selected and click **Program**.

A window will pop up showing the programming progress, as shown in Figure 37. Programming the Artix-7 FPGA on the Nexys4 DDR board will take several seconds. Once it is complete, the progress window will close.

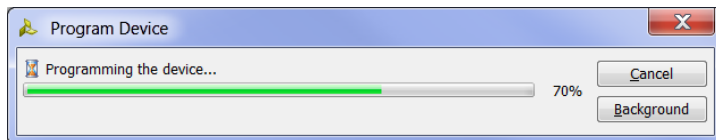


Figure 37. Program Device progress window

The MIPSfpga system is now downloaded onto the Nexys4 DDR board. Push the red reset pushbutton (labeled CPU RESET on the board, see Figure 32) to reset and start the MIPSfpga core. You will now see the LEDs display increasingly incremented values.

The MIPSfpga system is loaded with the IncrementLEDsDelay program. The machine code for this program is in the ram_rest_init.txt file located in the same directory as the Verilog files (i.e., in MIPSfpga_Fundamentals\rtl_up). The IncrementLEDsDelay program is similar to the IncrementLEDs program that you simulated earlier in this lab, but it adds a delay so that our eyes can detect the results on the LEDs. It would have been tedious to simulate thousands of cycles of delay, so we took the delay out of the program code we used for simulation (see Figure 19). The C, MIPS assembly, and machine code for IncrementLEDsDelay is shown in Figure 38 and Figure 39.

<pre>// C code unsigned int val = 1; volatile unsigned int* ledr_ptr; ledr_ptr = 0xbf800000; while (1) { *ledr_ptr = val;</pre>	<pre># MIPS assembly code # \$9 = val, \$8 = memory address 0xbf800000 addiu \$9, \$0, 1 # val = 1 lui \$8, 0xbf80 # \$8=0xbf800000 L1: sw \$9, 0(\$8) # mem[0xbf800000] = val addiu \$9, \$9, 1 # val = val+1</pre>
--------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<pre> val = val + 1; // delay } </pre>	<pre> delay: # loop 2,500,000x lui \$5, 0x026 # \$5 = 2,500,000 ori \$5, \$5, 0x25a0 add \$6, \$0, \$0 # \$6 = 0 L2: sub \$7, \$5, \$6 # \$7 = 2,500,000 - \$6 addi \$6, \$6, 1 # increment \$6 bgtz \$7, L2 # finished? nop # branch delay slot beqz \$0, L1 # branch to L1 nop # branch delay slot </pre>
----------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 38. IncrementLEDsDelay program

```

24090001 // bfc00000:      addiu $9, $0, 1
3c08bf80 // bfc00004:      lui    $8, 0xbf80
ad090000 // bfc00008: L1:  sw    $9, 0($8)
25290001 // bfc0000c:      addiu $9, $9, 1
3c050026 // bfc00010: delay: lui $5, 0x026
34a525a0 // bfc00014:      ori    $5, $5, 0x25a0
00003020 // bfc00018:      add    $6, $0, $0
00a63822 // bfc0001c: L2:  sub    $7, $5, $6
20c60001 // bfc00020:      addi   $6, $6, 1
1ce0fffd // bfc00024:      bgtz   $7, L2
00000000 // bfc00028:      nop
1000fff6 // bfc0002c:      beq    $0, $0, L1
00000000 // bfc00030:      nop

```

Figure 39. ram_reset_init.txt memory initialization file for IncrementLEDsDelay

The next labs will show how to write, compile, download, and run C and MIPS assembly programs on the MIPSfpga system.



Lab 2

C Programming



These materials produced in association with Imagination.
Join our University community for more resources.

community.imgtec.com/university

MIPSFpga Lab 2: C Programming

3. Introduction

In this lab you will learn to program the MIPSfpga processor in C. You will first complete a tutorial on writing, compiling, and downloading an example C program. Then you will write your own C program to calculate the Fibonacci numbers.

4. MIPSfpga C Tutorial

The MIPSfpga processor is programmed using Imagination's Codescape compiler tools. If you have not already, install the Codescape SDK and OpenOCD by running the installer located here:

MIPSFpga_Fundamentals\Scripts\OpenOCD-0.9.3-Installer.exe

If you used a version earlier than 1.2 of the MIPSfpga Getting Started Guide, you will need to update the Codescape/OpenOCD installation by running the installer indicated above.

Codescape supports programming in both C and assembly language. You will use C in this lab and MIPS assembly language in Lab 3.

In this tutorial, you will learn to write and compile a simple program that reads the value of the switches on the Nexys4 DDR board and flashes their values to the LEDs. You'll also learn to step through a program and debug it using the Bus Blaster probe and gdb, which is part of the Codescape tools.

Example C Program

Before writing your own program, we walk you through the steps of compiling, debugging, and running a program using some example code. Browse to this directory:

MIPSFpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches

Open the file main.c using a text editor such as Notepad or Wordpad. The main.c file contains the ReadSwitches program, as shown in Figure 40.

```
int main() {
    volatile int *IO_SWITCHES = (int*)0xbf800008;
    volatile int *IO_LEDR = (int*)0xbf800000;
    volatile unsigned int switches;

    while (1) {
```

```

    switches = *IO_SWITCHES;
    *IO_LEDR = switches;
    delay();
    *IO_LEDR = 0; // turn off LEDs
    delay();
}
return 0;
}

void delay() {
    volatile unsigned int j;

    for (j = 0; j < (1000000); j++) ; // delay
}

```

Figure 40. ReadSwitches C program

The following variable declarations make the variables `IO_SWITCHES` and `IO_LEDR` *point to* the addresses `0xbf800008` and `0xbf800000`, which are the memory-mapped I/O addresses of the switches and LEDs, respectively, on the Nexys4 DDR board.

```

volatile int *IO_SWITCHES = (int*)0xbf800008;
volatile int *IO_LEDR = (int*)0xbf800000;

```

Memory-mapped I/O was described in the MIPSfpga Getting Started Guide and will be described further in Lab 5. A read to address `0xbf800008` returns the value of the switches in the lower 16 bits (and 0's in the upper 16 bits), and a write to `0xbf800000` displays the lower 16 bits of the value on the LEDs.

The program reads the switches on the FPGA board by reading address `0xbf800008`:

```

switches = *IO_SWITCHES;

```

The program then writes that value to the LEDs by writing to address `0xbf800000`, the memory-mapped address of the LEDs.

```

*IO_LEDR = switches;

```

The program then delays some time, turns the LEDs off, then delays again before repeating.

```

delay();
*IO_LEDR = 0;
delay();

```

The variables `switches` and `j` are declared `volatile` so that they are not optimized away by the compiler.

It is critical to declare variables related to hardware **volatile** so that the compiler does not optimize them away.

Compiling, Running, and Debugging

Now compile, run, and debug the ReadSwitches example C program on the MIPSfpga core using the following steps, described in detail below.

- Step 1.** Download the MIPSfpga system onto the Nexys4 DDR board
- Step 2.** Compile the C program
- Step 3.** Load the C program onto MIPSfpga using Bus Blaster
- Step 4.** Debug the C program using `gdb` as needed

Remember, that to complete these labs you need to have installed all of the required software and drivers (Vivado, Codescape, and OpenOCD, as well as Bus Blaster and Nexys4 DDR board drivers) as described in the MIPSfpga Getting Started Guide.

Step 1. Download the MIPSfpga system to the Nexys4 DDR board

First, you will download the MIPSfpga system onto the Nexys4 DDR board. To do so, connect the Nexys4 DDR FPGA board to your computer, turn the board on, and open Vivado. Choose **Flow** → **Open Hardware Manager** from Vivado's top menu (see [Figure 41](#)).



Figure 41. Open Vivado's Hardware Manager

Click on **Open Target** → **Auto Connect** (see [Figure 42](#)). **Warning:** Sometimes Vivado crashes at this point. Simply reopen Vivado if that happens.

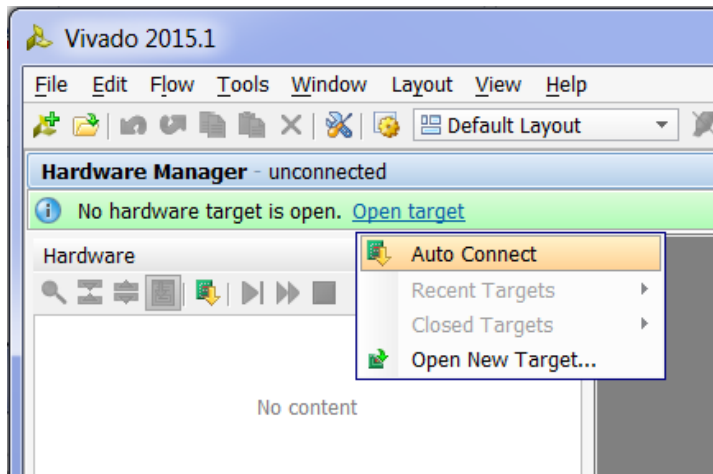


Figure 42. Autoconnect to FPGA on Nexys4 DDR board

Now click on **Program device** → **xc7a100t_0** (see Figure 43).

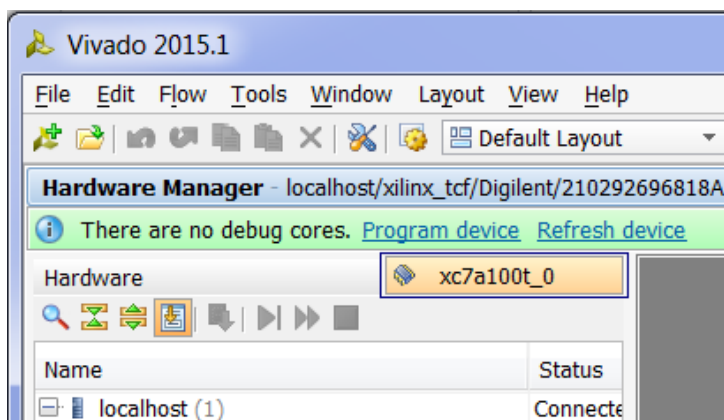



Figure 43. Program device

In the Program Device window, select the bitfile you created in Lab 1 (or the one provided at MIPSfpga_Fundamentals\Xilinx\Lab01_Vivado\mipsfpga_nexys4_ddr.bit). Then click Program. Leave the Debug probes file blank.

Click on the red CPU Reset button on the Nexys4 DDR board to reset the MIPSfpga core and begin running the pre-loaded program that displays incremented values on the LEDs.

Step 2. Compile the example program

Now compile the ReadSwitches example C program by opening a command shell. To do so, go to the **Start menu**, type in **cmd.exe**, and select  **cmd.exe**. Or you can shift-right-click on an empty space on your screen and select "Open command window here". In the command

shell, change to the MIPSfpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches directory. For example, if MIPSfpga_Fundamentals is in C:\ type:

```
cd C:\MIPSfpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches
```

Compile the example C code by typing make at the prompt in the command window:

```
make
```

This runs the Makefile, which compiles the user code (found in main.c) with the boot code (found in boot.S and the other .S files). Open and view the Makefile using a text editor such as Notepad or Wordpad. For future programs, any C program files can be placed under "CSOURCES="

```
CSOURCES= \  
main.c
```

Below is a brief description of the main parts of the Makefile. The top part of the file gives the names and locations of the compiler tools (gcc, ld, objdump, etc.). These compiler tools are provided with Codescape. They are GNU tools targeted to the MIPSfpga processor.

```
ifndef MIPS_ELF_ROOT  
$(error MIPS_ELF_ROOT must be set to point to toolkit  
installation root)  
endif  
  
CC=mips-mti-elf-gcc  
LD=mips-mti-elf-ld  
OD=mips-mti-elf-objdump  
OC=mips-mti-elf-objcopy  
SZ=mips-mti-elf-size
```

The next part of the Makefile indicates the flags to use for compiling and loading the program.

```
CFLAGS = -O1 -g -EL -c -msoft-float -march=m14kc  
LDFLAGS = -EL -msoft-float -march=m14kc -Wl,-Map=FPGA_Ram_map.txt
```

For the C flags, -O1 says to use optimization level 1. You can change this to higher optimization levels as desired, for example optimization level 2 (-O2) or 3 (-O3). It is typically a good idea to

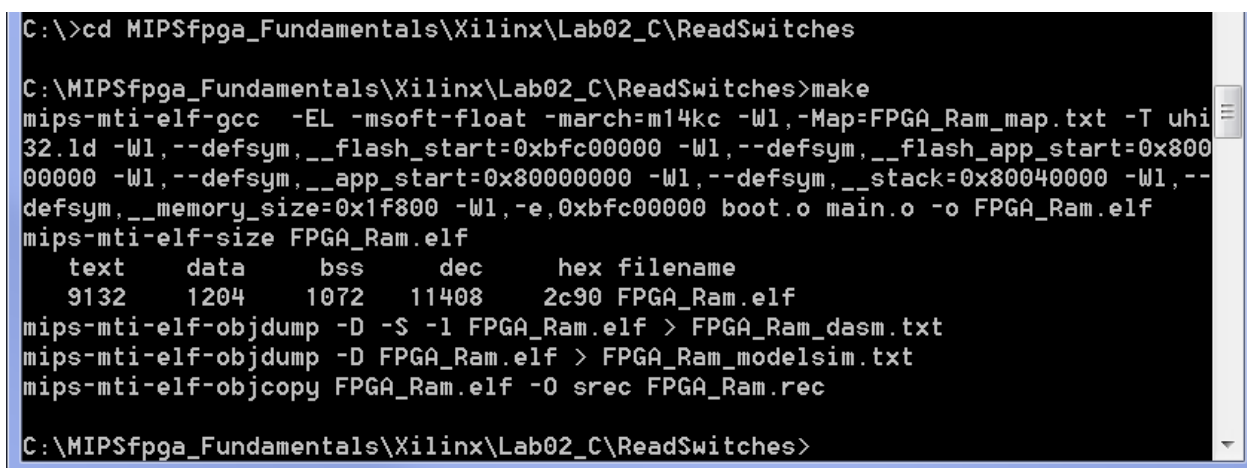
debug your code using the lowest optimization level (0 or 1). Then, once your code is working, increase the optimization level to produce faster, denser code. `-march=m14kc` indicates to target the M14K MIPS microAptiv architecture, `-msoft-float` says that there is no floating-point unit and to use software floating point routines instead.

The LDFLAGS are used when creating the ELF file, which will provide the program code as well as information about how to load the program and data into the MIPSfpga system's memory. Generally, the LDFLAGS indicate to not generate any floating point instructions (`-msoftfloat`) and to target the M14K MIPS architecture. The file indicated (`FPGA_Ram_map.txt`) describes how and where the program, boot code, and data will be loaded into memory, according to the physical memory map shown in Lab 1. The rest of the LD flags (`LDFLAGS += ...`) indicate where to place the program, boot code, stack, etc. in memory.

The remainder of the Makefile describes how to compile the program and to clean the directory (i.e., remove files created during compilation). To clean the directory of files created during compilation, type the following at the command prompt:

```
make clean
```

If you just cleaned the directory, compile the program (`main.c`) again by typing `make` at the command prompt. Notice that at the end of compilation, the Makefile outputs the size of the executable, as shown in **Error! Reference source not found.**



```
C:\>cd MIPSfpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches
C:\MIPSfpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches>make
mips-mti-elf-gcc -EL -msoft-float -march=m14kc -Wl,-Map=FPGA_Ram_map.txt -T uhi
32.ld -Wl,--defsym,__flash_start=0xbfc00000 -Wl,--defsym,__flash_app_start=0x800
00000 -Wl,--defsym,__app_start=0x80000000 -Wl,--defsym,__stack=0x80040000 -Wl,--
defsym,__memory_size=0x1f800 -Wl,-e,0xbfc00000 boot.o main.o -o FPGA_Ram.elf
mips-mti-elf-size FPGA_Ram.elf
text  data  bss  dec  hex filename
9132  1204  1072  11408  2c90 FPGA_Ram.elf
mips-mti-elf-objdump -D -S -l FPGA_Ram.elf > FPGA_Ram_dasm.txt
mips-mti-elf-objdump -D FPGA_Ram.elf > FPGA_Ram_modelsim.txt
mips-mti-elf-objcopy FPGA_Ram.elf -o src FPGA_Ram.rec
C:\MIPSfpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches>
```

Figure 44. Command shell output of Makefile

The text is **9132** bytes, the data is **1204** bytes and the bss segment (static data that should be initialized to 0) is **1072** bytes for a total of **11408 = 0x2c90** bytes. This fits easily within the MIPSfpga memory space. However, you will want to keep your eye on these numbers for larger programs to make sure they fit within MIPSfpga's physical memory.

You should now see the following files in the ReadSwitches directory:

- FPGA_Ram.elf
- FPGA_Ram_dasm.txt
- FPGA_Ram_modelsim.txt
- main.o

FPGA_Ram.elf is the main output of compilation. It is the ELF (executable and linkable format) executable that you will use to load the program into the memory of the MIPSfpga core.

FPGA_Ram_dasm.txt is a *disassembled* version of the executable. It is basically a human-readable version of the ELF file that shows instruction addresses and instructions interspersed with the line numbers of the higher-level (assembly or C) source code.

FPGA_Ram_modelsim.txt is another human-readable version of the ELF file, but it is not interspersed with the source code information. It shows the memory addresses and corresponding instructions/data, including those memory addresses that should be initialized to 0. We will use this file to create memory definition files for simulation of compiled programs using Modelsim.

main.o is the executable and linkable version of main.c.

Open FPGA_Ram_dasm.txt using a text editor to see where the boot code and user code will be placed. The top of the file shows the boot code, starting at 0x9fc00000. Recall that this virtual address maps to physical address 0x1fc000000, which is the physical address of the first instruction fetched upon reset of the MIPSfpga core.

Near the bottom of the file, you can view the user code from main.c, starting at 0x8000075c. This will map to physical addresses starting at 0x0000075c.

Step 3. Load the C program onto the MIPSfpga system using Bus Blaster

Now that the example program is compiled, you will load it onto the MIPSfpga core using the Bus Blaster probe. First, connect the Bus Blaster probe to the Nexys4 DDR board, as shown in Figure 45. Do so by connecting the two rows of 6 header pins into the small Adapter Board. Then connect the Adapter Board into the PMOD-B port of the Nexys4 DDR board, as shown in the figure. Connect one side of the ribbon cable into the small Adapter board and the other side

into the Bus Blaster probe. Connect one side of the USB cable between the Bus Blaster probe and your computer. Remember, it is best to **use the same port** for the Bus Blaster probe as the one on which you installed the drivers for it.

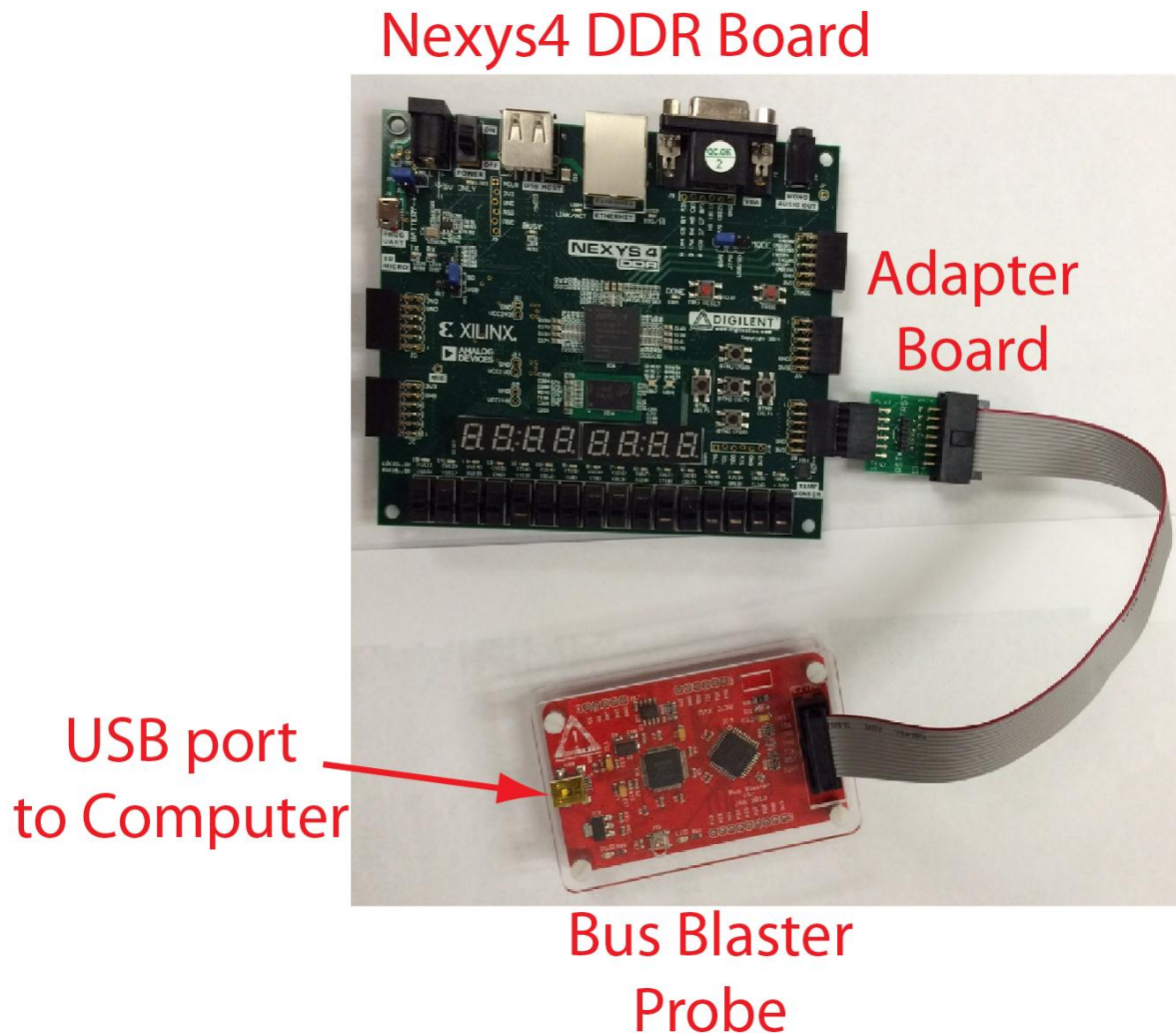


Figure 45. Nexys4 DDR board connected to the Bus Blaster probe

Now open a command shell (i.e., Start menu → cmd.exe.) In the command shell, change to the **MIPSfpga_Fundamentals\Scripts\Nexys4_DDR** directory. For example, if MIPSfpga_Fundamentals is located on the C drive, type the following at the shell prompt:

```
cd C:\MIPSfpga_Fundamentals\Scripts\Nexys4_DDR
```

Now you will run the **loadMIPSfpga.bat** script that will (1) compile the program, (2) set up a connection to the MIPSfpga core using OpenOCD, (3) download the program onto the MIPSfpga

system, and (4) allow you to use the Gnu debugger (gdb) to load and debug the program on the MIPSfpga core.

Run the loadMIPSfpga.bat batch script supplying the directory of the ReadSwitches program as the argument. For example, if MIPSfpga_Fundamentals is on the C drive, at the command font, type:

```
loadMIPSfpga.bat C:\MIPSfpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches
```

Or you could type:

```
loadMIPSfpga.bat ..\..\Xilinx\Lab02_C\ReadSwitches
```

After running the script, you will see the ReadSwitches program running on the MIPSfpga core. Recall that the ReadSwitches program repeatedly reads the value of the switches and flashes that value on the LEDs. Toggle some of the switches at the bottom of the Nexys4 DDR board and watch as the corresponding LEDs flash.

The loadMIPSfpga.bat script first opens a shell to compile the specified program using make. It then opens up two more shells to create the OpenOCD connection and run gdb. After you are finished with a program, you need to close these two windows.

Step 4. Debug the C program using gdb as needed

Although this ReadSwitches program works and requires no debugging, we will show you how to go through the process of debugging using gdb, supplied as part of the Codescape SDK.

Click on the gdb command shell that was opened by the loadMIPSfpga.bat script in the previous step. Enter the sequence of commands shown in [Table 1](#) to halt the program, set breakpoints, view variable and register values, etc.

Table 1. gdb command sequence

Command	Description
<code>monitor reset halt</code>	Reset and stop the processor. Notice the program stopped running. Shortcut: <code>mo reset halt</code>
<code>b main</code>	Set a breakpoint at the main function. (Short for: "break main".) Notice that the breakpoint is set at 0x800007b4, just after the code for the delay function and stack operations (located at addresses 0x8000075c – 0x800007b0). Note that you can set breakpoints even when the processor is running, but the breakpoints will take effect only when the processor is halted (<code>mo reset halt</code>).

b *0x800007b8	<p>Set a breakpoint at instruction address 0x800007b8. In the ReadSwitches C program, this is the load word instruction (<code>lw</code>) that reads the value of the switches (see MIPSfpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches\FPGA_Ram_d asm.txt)</p> <pre>800007b8: 8e020008 lw v0, 8(s0)</pre> <p>Note that you could also have typed: <code>b 20</code> This would set a breakpoint at line 20 of main.c</p>
b *0x800007c4	<p>Set a breakpoint at instruction address 0x800007c4. In the ReadSwitches C program, this is the store word instruction (<code>sw</code>) that writes to the LEDs (see MIPSfpga_Fundamentals\Xilinx\Lab02_C\ReadSwitches\FPGA_Ram_d asm.txt)</p> <pre>800007c4: ae020000 sw v0, 0(s0)</pre>
i b	List the breakpoints. (Short for: "info breakpoint".) At this point it will list the breakpoint at instruction addresses 0x800007b4 (main), 0x800007b8, and 0x800007c4.
c	Continue the processor execution. (Short for: "continue".) It will stop at the first breakpoint, in this case, when it gets to <code>main</code> (instruction address 0x800007b4).
x/3i \$pc	<p>Prints 3 instructions starting with the current instruction (<code>\$pc</code> is the program counter and contains the address of the current instruction).</p> <pre>800007b4: lui s0, 0xbf80 800007b8: lw v0, 8(s0) 800007bc: sw v0, 16(sp)</pre>
x/3x \$pc	Prints 3 instructions in hexadecimal, starting at the address specified.
c	Continue to the next break point, which is at 0x800007b8.
stepi	<p>Executes a single instruction. For example, now you will see the PC increment to 0x800007bc.</p> <p>Shortcut: <code>si</code></p>
si	Step one more instruction. (You can also simply press the Enter key to repeat the last gdb command.)
p switches	Now that the switches have been read, we can print the value of the variable <code>switches</code> . (Short for: "print switches".) For example, if the 3 least significant switches are 1 (i.e., in the UP position), <code>switches</code> will have the value 7.
p/x switches	Prints the value of the <code>switches</code> variable in hexadecimal.
p/x &switches	Prints the address of the <code>switches</code> variable
i r	Print the value of all registers. (Short for: "info registers".)

<code>i r v0</code>	Print the value of register <code>v0</code> only. At this point, <code>v0</code> holds the value of the FPGA board switches. This value will be written to the LEDs by the store word (<code>sw</code>) instruction at <code>0x800007c4</code> .
<code>c</code>	Continue program execution. (Short for: "continue".) Execution is now at <code>0x800007c4</code> , the store word instruction that will write the value of the switches to the LEDs.
<code>i r s0</code>	Print the value of register <code>s0</code> . <code>s0</code> currently holds the memory-mapped I/O address of the LEDs: <code>0xbf800000</code> .
<code>i r v0</code>	Print the value of register <code>v0</code> . <code>v0</code> holds the value of the switches that will get written to the LEDs.
<code>si</code>	Execute the store word instruction and watch as the LEDs are updated to the value of the switches.
<code>d 1</code>	Delete breakpoint 1 (type <code>i b</code> to list the breakpoints with their numbers). This deletes the breakpoint at the beginning of <code>main</code> .
<code>monitor reset run</code>	Reset and run the processor. This will run the processor without breakpoints, even if breakpoints have been set. Shortcut: <code>mo reset run</code>

For a list of other gdb commands, refer to the GDB User Manual available as a link on this webpage:

<http://www.gnu.org/software/gdb/documentation/>

5. MIPSfpga Exception Handler

MIPSfpga can enter the exception handler for various reasons including accessing an illegal address or attempting to execute an unknown instruction. Here we show how to write an exception handler for MIPSfpga, so that you will know when an exception occurs. Browse to `MIPSfpga_Fundamentals\Xilinx\Lab02_C\ExceptionHandler` and open `main.c`. View the `_mips_handle_exception` function, as shown below. This function, which will be called upon an exception, displays `0x8001` on the LEDs to indicate that an exception occurred. You could choose a different value to output. If this function does not exist in your code, upon an exception, the processor simply hangs.

```
void _mips_handle_exception(void* ctx, int reason) {
    volatile int *IO_LEDR = (int*)0xbf800000;

    *IO_LEDR = 0x8001; // Display 0x8001 on LEDs to indicate error state
    while (1) ;
}
```

```
}
```

To illustrate what happens when an exception occurs, the code in this program intentionally causes an exception by attempting to write to address 0 by executing the code below.

```
volatile int *test_error = (int*)0x0;  
*test_error = 56; // write to address 0 will cause an exception
```

Compile and run this code on the MIPSfpga system. At a command shell prompt, change to the MIPSfpga_Fundamentals\Scripts\Nexys4_DDR directory and type:

```
loadMIPSfpga.bat ../../Xilinx\Lab02_C\ExceptionHandler
```

You will want to include this exception handler function (`_mips_handle_exception`) in all of your code to detect when an exception occurs.

6. Fibonacci Numbers

Now you will write your own C program, compile it, and run it on MIPSfpga. Create a program that will calculate and display the first 11 Fibonacci numbers on the LEDs. Each number in the Fibonacci series is the sum of the previous two numbers. Table 2 lists the first few numbers in the series.

Table 2: Fibonacci Series

n	1	2	3	4	5	6	7	8	9	10	11	...
fib(n)	1	1	2	3	5	8	13	21	34	55	89	...
fib(n)	0x1	0x1	0x2	0x3	0x5	0x8	0xd	0x15	0x22	0x37	0x59	

We can also define the fib function for negative values of n. To be consistent with the definition of the Fibonacci series, what would the following values be?

fib(0) = _____

fib(-1) = _____

These values are useful when writing a loop to compute fib(n) for all non-negative values of n.

Make a copy of the Lab02_C/ExceptionHandler folder and rename the new folder Fibonacci. Write your program in the main.c file in that folder. Your program should compute the Fibonacci numbers for n = 1...11 and output each Fibonacci number to the LEDs. The LEDs should display the Fibonacci numbers in binary, with a delay between each number so that they are viewable.

Once you have finished writing your program, use the `make` command to compile it. If there are any errors, fix them and recompile.

After your Fibonacci program compiles without errors, load it onto the MIPSfpga core by:

1. Opening a command shell
2. Changing to the MIPSfpga_Fundamentals\Scripts\Nexys4_DDR directory
3. Typing at the command prompt:

```
loadMIPSfpga.bat C:\MIPSfpga_Fundamentals\Xilinx\Lab02_C\Fibonacci
```

Table 2 lists the Fibonacci numbers in hexadecimal to help you read the binary values on the LEDs. In later labs, you will expand the MIPSfpga hardware to enable you to use the 7-segment displays available on the Nexys4 DDR FPGA board. But for now, remember that you can also use breakpoints in `gdb` to examine the values produced by your program.

MIPSfpga

by Imagination

Lab 5

Memory-Mapped I/O: 7-Segment Displays



These materials produced in association with Imagination.
Join our University community for more resources.

community.imgtec.com/university

MIPSfpga Lab 5:

Memory-Mapped I/O: 7-Segment Displays

7. Introduction

In this lab you will learn about memory-mapped inputs and outputs (I/O) by building hardware modules to expand the capability of the MIPSfpga system so that it can write to the 7-segment displays on the Nexys4 DDR board. You will then test your new hardware by simulating a short sequence of MIPS assembly code. At the end of the lab, you will write C programs that display the value of the switches on the 7-segment displays.

8. MIPSfpga Memory-Mapped I/O

A processor uses the memory interface to interact with peripheral devices, such as the switches, LEDs, and 7-segment displays on the Nexys4 DDR FPGA board. *Memory-mapped I/O* enables a processor to write to or read from a peripheral device in the same manner that it reads or writes memory. Each peripheral device is assigned one or more memory addresses. When the processor accesses such a memory address, the peripheral device is accessed instead of memory. The MIPSfpga system uses the AHB-Lite bus to access external memory and peripherals.

AHB-Lite Bus

The AHB-Lite bus has a clock, write enable, address, and read and write data signals (HCLK, HWRITE, HADDR, HRDATA, and HWDATA), as shown in [Figure 46](#). The "H" prefix indicates that they are part of the AHB-Lite bus. Memory and peripherals are connected to this interface to receive and supply data. The MIPSfpga core sends these signals to the AHB-Lite Bus:

- **HCLK**: the 50 MHz system clock
- **HWRITE**: write enable (1 when writing, 0 when reading)
- **HADDR**: the address being read or written
- **HWDATA**: the data being written on a write

The MIPSfpga core receives the following input from the AHB-Lite bus:

- **HRDATA**: the read data produced by memory or the peripherals

The MIPSfpga system has three modules on the AHB-Lite bus: two memories (RAM0 and RAM1) and a general-purpose I/O module (GPIO). RAM0 contains the boot code and RAM1 contains the user code and data. The GPIO unit interfaces with the LEDs, switches, and pushbuttons on the Nexys4 DDR board.

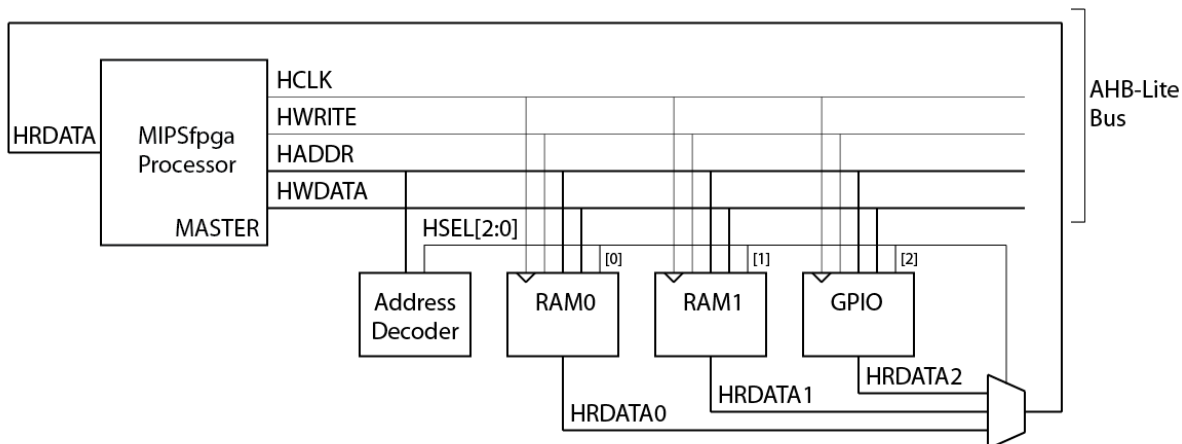


Figure 46. MIPSfpga processor with three peripheral devices

In addition to the three peripherals, the memory-mapped I/O interface requires an Address Decoder and a multiplexer. Depending on the address generated by the processor (HADDR[31:0]), the Address Decoder will enable the processor to access one of the three modules. The Address Decoder generates a select signal HSEL[2:0] that is used by the modules and by the 3:1 multiplexer.

RAM0 holds the boot code (virtual addresses 0xbf000000-0xbf01ffff = physical addresses 0x1fc00000-0x1fc1ffff). RAM1 holds the user code (virtual addresses 0x80000000-0x8003ffff = physical addresses 0x00000000-0x0003ffff). The LEDs, switches, and pushbuttons on the Nexys4 DDR board are mapped to virtual memory addresses 0xbf800000-0xbf80000c, as shown in Table 3. The processor code uses virtual memory addresses, and the AHB-Lite bus receives physical addresses. The memory management unit (MMU) on the MIPSfpga core performs this address translation.

Table 3. Memory addresses for Nexys4 DDR FPGA board

Virtual address	Physical address	Signal Name	Nexys4 DDR
0xbf80 0000	0x1f80 0000	IO_LED	LEDs
0xbf80 0008	0x1f80 0008	IO_SW	switches
0xbf80 000c	0x1f80 000c	IO_PB	U, D, L, R, C pushbuttons

The following sequence of MIPS assembly instructions writes the value 5 to the LEDs:

```

lui $8, 0xbf80      # $8 = 0xbf800000 (address of LEDs)
addi $9, $0, 5     # $9 = 5
sw $9, 0($8)

```

Recall that load-upper-immediate (*lui*) loads the 16-bit value 0xbf80 into the upper half of \$8 and clears the lower half. Upon execution of the store word instruction (*sw*), HADDR = 0x1f800000, HWRITE = 1, and HWDATA = 5. The Address Decoder detects that address

0x1f800000 belongs to the general-purpose I/O (GPIO) peripheral and asserts HSEL[2], the select signal associated with that peripheral. The GPIO module detects that HSEL[2] and HWRITE are asserted. Because the GPIO module could potentially write to multiple peripherals, the module uses the address to determine that the LEDs should be written with the value on the HWDATA bus. Specifically, a register whose output is physically connected to the LEDs is updated with the value on HWDATA. That way, the value persists until the LEDs are written again by a later instruction.

Similarly, the following sequence of code reads the value of the switches:

```
lui $8, 0xbf80      # $8 = base address of the I/O
lw  $9, 8($8)      # $9 = value of the switches
```

Upon execution of the load word instruction (`lw`), `HADDR = 0x1f800008` and `HWRITE = 0` (indicating a read). The Address Decoder detects that address `0x1f800008` belongs to the GPIO peripheral and asserts HSEL[2] (and keeps the other select signals HSEL[1] and HSEL[0] low). The GPIO module detects the address corresponding to the switches and selects to send the value of the switches to its read data output, `HRDATA2`. The select signals HSEL[2:0] control the multiplexer. Because HSEL[2] is asserted, the multiplexer sends `HRDATA2` through to `HRDATA`. The MIPSfpga processor then reads the value on `HRDATA`, as it would with a typical read from memory, and stores that value in `$9`. Thus, after the `lw` completes, `$9` contains the value of the switches.

The hardware for the MIPSfpga AHB-Lite modules is located in the `mipsfpga_ahb` module and its submodules. It is best to view this module in your Vivado project, so that the hierarchy is clear. Open the Vivado project that you created in Lab 1 (i.e., in `MIPSfpga_Fundamentals\Xilinx\Lab01_Vivado\Project1`). In the Project Manager window, expand `mipsfpga_nexys4_dds`, then `mipsfpga_sys`, then `mipsfpga_ahb` to view the `mipsfpga_ahb` hierarchy, as shown in Figure 47. Double-click on any of the modules and the Verilog file will open in the neighboring panel.

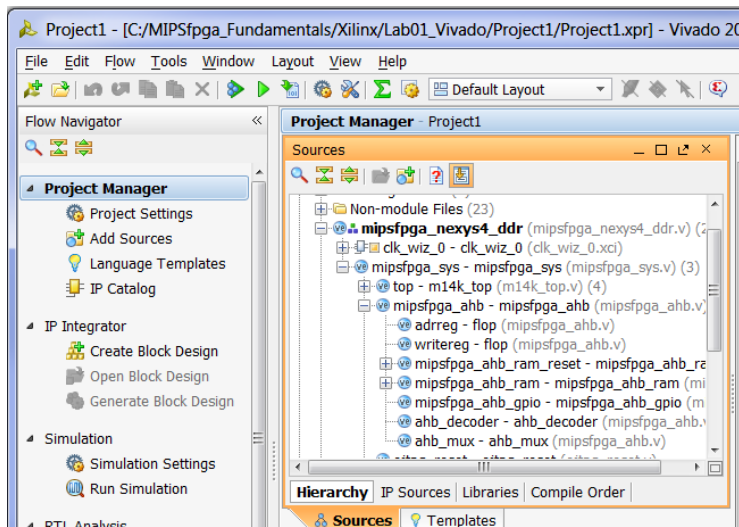


Figure 47. `mipsfpga_ahb` hierarchy shown in Vivado

For example, double-click on `mipsfpga_ahb` to view the interface signals (see [Figure 48](#)). Notice all of the AHB-Lite signals from [Figure 46](#) (`HCLK`, `HADDR`, `HWRITE`, `HWDATA`, and `HRDATA`). Additional AHB-Lite signals are also available if desired. The module also has the memory-mapped I/O signals `IO_Switch`, `IO_PB`, and `IO_LEDR` that connect to the switches, pushbuttons, and LEDs on the Nexys4 DDR board. `IO_LEDG` is not used on the Nexys4 DDR board.

```

16 (
17     input          HCLK,
18     input          HRESETn,
19     input          [ 31: 0] HADDR,
20     input          [  2: 0] HBURST,
21     input          HMASTLOCK,
22     input          [  3: 0] HPROT,
23     input          [  2: 0] HSIZE,
24     input          [  1: 0] HTRANS,
25     input          [ 31: 0] HWDATA,
26     input          HWRITE,
27     output         [ 31: 0] HRDATA,
28     output         HREADY,
29     output         HRESP,
30     input          SI_Endian,
31
32 // memory-mapped I/O
33     input          [ 17: 0] IO_Switch,
34     input          [  4: 0] IO_PB,
35     output         [ 17: 0] IO_LEDR,

```

Figure 48. `mipsfpga_ahb` interface signals

The modules instantiated within `mipsfpga_ahb` are the three peripherals, address decoder, and multiplexer shown in [Figure 46](#). The corresponding Verilog module names are given in [Table 4](#). View the Verilog code to see how the functionality described above is implemented.

Table 4. AHB-Lite Modules

Name from Figure 46	Module Name
RAM0	<code>mipsfpga_ahb_ram_reset</code>
RAM1	<code>mipsfpga_ahb_ram</code>
GPIO	<code>mipsfpga_ahb_gpio</code>
Address Decoder	<code>ahb_decoder</code>
Multiplexer (for HRDATA)	<code>ahb_mux</code>

The GPIO module (`mipsfpga_ahb_gpio`) interfaces with the general-purpose I/O on the Nexys4 DDR board. The MIPSfpga system includes access to the LEDs, switches and pushbuttons on the board. In this and the next labs, you will expand the MIPSfpga functionality to extend to other peripherals, starting with the eight 7-segment displays available on the Nexys4 DDR board.

7-Segment Displays

Digits can be represented using 7-segment displays, as shown in [Figure 49](#). Each of the seven segments is labeled a through g. The numbers 0 through F light up the segments shown in [Figure 50](#). For example, the number 0 lights up all but the middle segment, g.

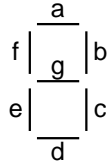


Figure 49. Seven-segment display

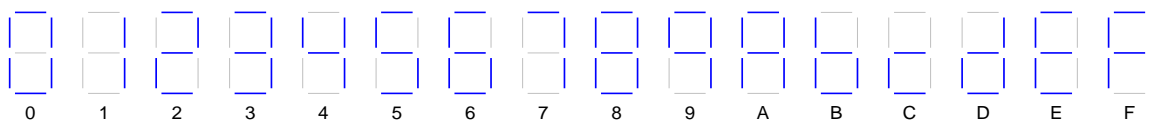


Figure 50. Seven-segment display function

Given an input number ranging from 0x0 – 0xF, we will show how to expand the MIPSfpga system to drive the 7-segment displays to show that number. Each segment of the display is low-asserted, so it turns ON when it is 0.

The truth table below ([Table 5](#)) shows the inputs (a 4-bit value from 0-15) and outputs for a 7-segment display decoder that takes in a 4-bit number and produces the value of the segments corresponding to that number. So, for example, with an input of "0", the 7-segment display decoder turns all but the middle segment (S_g) ON. Thus, the first row for Hexadecimal digit "0" shows all the segments as 0 except S_g . (Remember that the segments are low-asserted, so they are ON when they receive 0.) The digit "1" should only have S_b and S_c ON (so S_b and S_c are 0 in that row), and so forth.

Table 5. Truth table for 7-segment display decoder

Hexadecimal Digit	Inputs				Outputs							HEX	
	D ₃	D ₂	D ₁	D ₀	S _a	S _b	S _c	S _d	S _e	S _f	S _g		
0	0	0	0	0	0	0	0	0	0	0	0	1	01
1	0	0	0	1	1	0	0	1	1	1	1	1	4f
2	0	0	1	0	0	0	1	0	0	0	1	0	12
3	0	0	1	1	0	0	0	0	1	1	0	0	06
4	0	1	0	0	1	0	0	1	1	0	0	0	4c
5	0	1	0	1	0	1	0	0	1	0	0	0	24
6	0	1	1	0	0	1	0	0	0	0	0	0	20
7	0	1	1	1	0	0	0	1	1	1	1	1	0f
8	1	0	0	0	0	0	0	0	0	0	0	0	00
9	1	0	0	1	0	0	0	1	1	0	0	0	0c
A	1	0	1	0	0	0	0	1	0	0	0	0	08
B	1	0	1	1	1	1	0	0	0	0	0	0	60
C	1	1	0	0	1	1	1	0	0	1	0	0	72
D	1	1	0	1	1	0	0	0	0	1	0	0	42
E	1	1	1	0	0	1	1	0	0	0	0	0	30
F	1	1	1	1	0	1	1	1	0	0	0	0	38

Build 7-segment decoder

Write a Verilog module that describes the seven-segment display decoder in hardware. The module declaration is provided for you in:

MIPSFpga_Fundamentals\Xilinx\Lab05_7seg\VerilogFiles\mipsfpga_ahb_sevensegdec.v

The module has a 4-bit input, data[3:0], and a 7-bit output, segments[6:0], corresponding to each of the segments a-g. Test your hardware in simulation using XSIM and debug as needed. In

the next step, you will use this module to drive the 7-segment displays on the Nexys4 DDR board.

7-Segment Displays on the Nexys4 DDR board

The Nexys4 DDR board has eight 7-segment digits. All eight of the digits on the Nexys4 DDR board are connected to the same low-asserted segment pins, referred to as CA, CB, CC,...,CG. However, each digit has its own enable which is also low-asserted. Figure 51 shows the eight 7-segment displays on the Nexys4 DDR board. CA is connected to the cathode of the A segment for all eight displays, CB to the cathode of the B segment for all displays, and so forth. Each digit has an enable signal, corresponding to the respective bit of the signal AN[7:0]. AN[7:0] is connected via an inverter, to the anode of all segments for the respective digit. For example, if AN[7] is 0, digit 7 will be driven to the values on CA...CG.

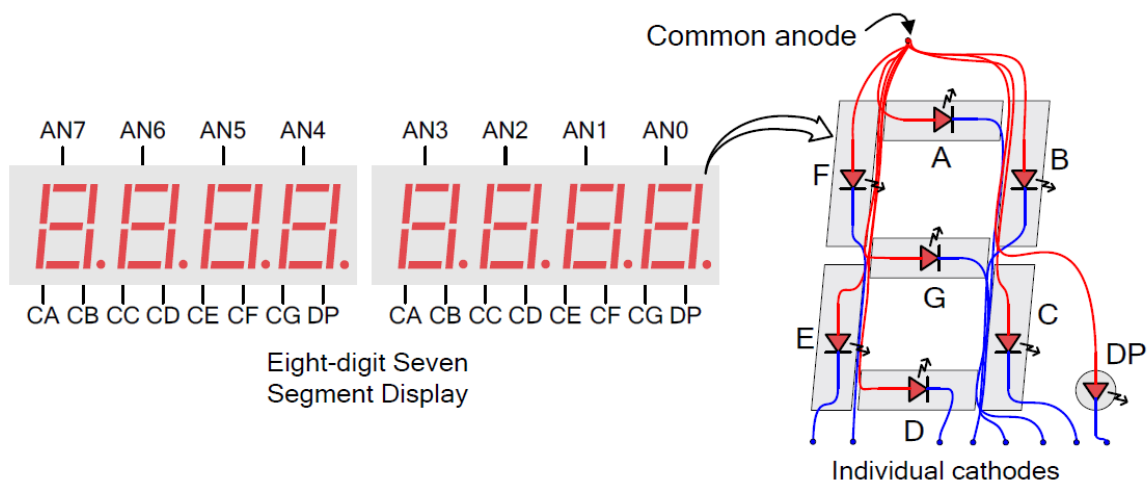


Figure 51. Eight 7-segment displays on the Nexys4 DDR board

(© Nexys4 DDR Reference Manual)

To drive each segment to a different value, the enables (AN[7:0]) and segment values (CA...CG) must be driven sequentially, at a rapid enough speed that our eyes don't detect the flicker. For example, to drive display 0 and 1 to the values 3 and 9, we drive CA...CG to display the value 3, and then drive AN[0] LOW, then we drive CA...CG to display the value 9 and drive AN[1] LOW. If we refresh each digit about every 2 ms, our eyes can't detect any flicker.

Build HDL module to drive Nexys4 DDR 7-segment displays

Now you will write a Verilog module that drives the eight 7-segment displays on the Nexys4 DDR board. The module declaration is provided in:

```
MIPSFpga_Fundamentals\Xilinx\Lab05_7seg\VerilogFiles\mipsfpga_ahb_sevensegtimer.v
```

The module receives the number to display on each of the eight digits (DISP0[3:0] – DISP7[3:0]) and a signal indicating which of the eight displays are enabled (EN[7:0]). It also receives the 50 MHz clock (clk) and a low-asserted reset signal (resetsn) as inputs. The outputs are the 7-

segment display enables (DISPENOUT[7:0]) and the values of the 7 segments, A-G (DISPOUT[6:0]). Later in the lab you will connect these outputs through to the top-level module (mipsfpga_nexys4_ddr) so that they drive the eight display enables (AN[7:0]) and the seven segment pins (CA...CG).

Your module should drive each enabled digit sequentially about every 2 ms. You will need to use your 7-segment display decoder that you wrote in the previous section. Note that you could expand the functionality of this module to include the decimal point (DP) if desired. Test your hardware in simulation using XSIM and debug as needed.

Adding Seven-Segment Display Functionality to the GPIO AHB-Lite Module

Now that you have written the hardware modules that will write the eight 7-segment displays, add functionality to the MIPSfpga system to interface with the displays. Your goal is to enable the user to write to the eight 7-segment displays using `sw`. Start by doing the following:

1. Assign memory-mapped I/O addresses to the enable signal and each of the eight digits
2. Modify the GPIO module to detect these addresses and store the written data to the associated memory-mapped I/O registers
3. Connect these registers to the mipsfpga_ahb_sevensegtimer module you just created

To make these changes, you will need to modify the following files (found in the MIPSfpga_Fundamentals\rtl_up directory):

- mipsfpga_ahb_const.vh
- mipsfpga_ahb_gpio.v

Below is some guidance for each of the above steps.

1. Assign memory-mapped I/O addresses

Assign nine addresses to the seven-segment displays, one for the enable and eight for the value of each digit, as shown in Table 6. The user will write to these addresses to set the enable and the digit values.

Table 6. Memory addresses for Nexys4 DDR FPGA 7-segment displays

Virtual address	Physical address	Signal Name	Nexys4 DDR
0xbf80 0010	0x1f80 0010	SEGEN_N[7:0]	AN[7:0]
0xbf80 0014	0x1f80 0014	SEG0_N[3:0]	Digit 0 value
0xbf80 0018	0x1f80 0018	SEG1_N[3:0]	Digit 1 value
0xbf80 001c	0x1f80 001c	SEG2_N[3:0]	Digit 2 value
0xbf80 0020	0x1f80 0020	SEG3_N[3:0]	Digit 3 value
0xbf80 0024	0x1f80 0024	SEG4_N[3:0]	Digit 4 value
0xbf80 0028	0x1f80 0028	SEG5_N[3:0]	Digit 5 value
0xbf80 002c	0x1f80 002c	SEG6_N[3:0]	Digit 6 value
0xbf80 0030	0x1f80 0030	SEG7_N[3:0]	Digit 7 value

To define these memory-mapped addresses, modify the `mipsfpga_ahb_const.vh` Verilog header file. In Vivado, open Project1. Browse to **mipsfpga_ahb_const.vh** in the Project Manager window (as shown in Figure 52), under Design Sources → Verilog Header.

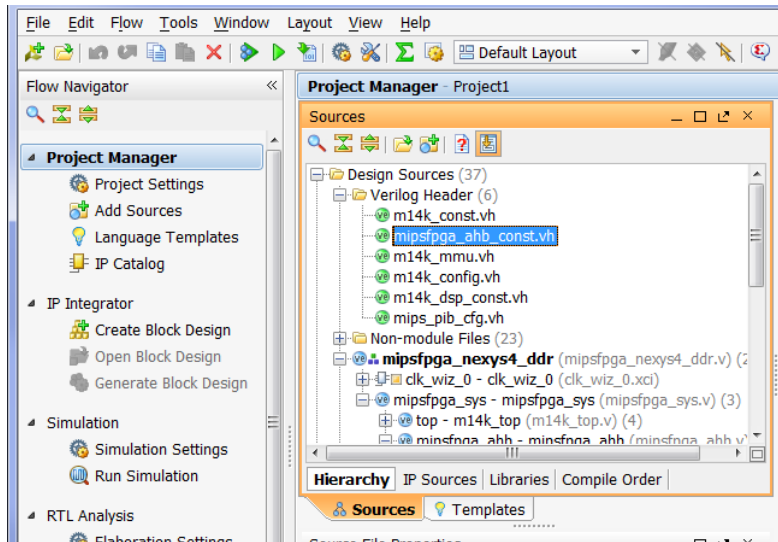


Figure 52. mipsfpga_ahb_const.vh Verilog header file

Define the new memory-mapped I/O addresses for the 7-segment displays as `H_7SEGEN_ADDR`, `H_7SEG0_ADDR`, ... `H_7SEG7_ADDR`. The Address Decoder (`ahb_decoder` module) uses the most significant bits of the address to detect which of the three AHB slaves to enable (the reset RAM, program RAM, or GPIO module). Then, once selected, the GPIO module uses the lower bits of the address to determine which of its peripherals should be written or read. Bits 5:2 of the memory-mapped I/O address are saved in another constant: `H_*_IONUM`, lower in the `mipsfpga_ahb_const.vh` file, as shown below:

```

`define H_LED_R_IONUM          (4'h0)
`define H_LED_G_IONUM          (4'h1)
`define H_SW_IONUM             (4'h2)
`define H_PB_IONUM             (4'h3)

```

For example, the switches are mapped to physical address `0x1f800008`, so bits 5:2 are `0x2` (i.e., `H_SW_IONUM` is `4'h2`).

Name the I/O numbers for the 7-segment display variables: `H_7SEGEN_IONUM`, `H_7SEG0_IONUM`, etc. For example since the address for the 7-segment enable is `0x1f800010`, `H_7SEGEN_IONUM` is `0x4`.

2. Modify the GPIO module

Now modify the GPIO module to detect the nine memory-mapped I/O addresses you just defined and write the data (`HWDATA`) to those registers when the corresponding address is detected. In Project1 in Vivado, open **mipsfpga_ahb_gpio.v**. In the module declaration, declare

and output the enable and segment signals (A-G). Name these signals `IO_7SEGEN_N[7:0]` and `IO_7SEG_N[6:0]`, respectively. In a higher-level module, these will drive the enables (`AN[7:0]`) and segment values (`CG...CA`) on the Nexys4 DDR board.

You must now create 9 registered values that hold the value of the enables and the values to display on the eight 7-segment display digits. The user will write to these registers using memory-mapped I/O. Name the registered signal that holds the enables `SEGEN_N[7:0]`. Name the registers that each hold the 4-bit values to display on the eight digit `SEGO_N[3:0]`, ..., `SEG7_N[3:0]`. Modify the GPIO module so that these registers get written when the correct address is detected.

3. Connect these registers to the `mipsfpga_ahb_sevensegtimer` module you just created

Now, within the GPIO module, instantiate and connect the `mipsfpga_ahb_sevensegtimer` module that you built earlier in this lab. You will connect its inputs to the memory-mapped I/O registers (as well as the clock and reset signals) and its outputs to the 7-segment display signals (`IO_7SEGEN_N` and `IO_7SEG_N`).

Connect the 7-Segment Display Signals to the Nexys4 DDR Board

Now you will connect the output signals from the AHB GPIO module through to drive the 7-segment displays on the Nexys4 DDR board. Feed the output signals from the GPIO module (`IO_7SEGEN_N` and `IO_7SEG_N`) up through the levels of hierarchy until they reach the Nexys4 DDR board (i.e., outputs of the `mipsfpga_nexys4_ddr` module). To do so, you will need to modify the following modules:

- `mipsfpga_ahb.v`
- `mipsfpga_sys.v`
- `mipsfpga_nexys4_ddr.v`

In the highest-level module (`mipsfpga_nexys4_ddr.v`), name the output signals that will drive the 7-segment display: `CA`, `CB`, `CC`, `CD`, `CE`, `CF`, `CG`, and `AN[7:0]`. Recall that `CA...CG` drive the segments and `AN[7:0]` drives the enables.

You will also modify the Xilinx Design Constraint (`.xdc`) file. Open this file from Project1 (in Vivado) by expanding Constraints → `constrs_1` in the Project Manager window as shown in [Figure 53](#). Double-click on the `mipsfpga_nexys4_ddr.xdc` file to open it. The XDC file assigns the signal names, `AN[7:0]` and `CA – CG`, to pins on the Artix-7 FPGA that are physically connected to the 7-segment display inputs on the Nexys4 DDR board using wire traces.

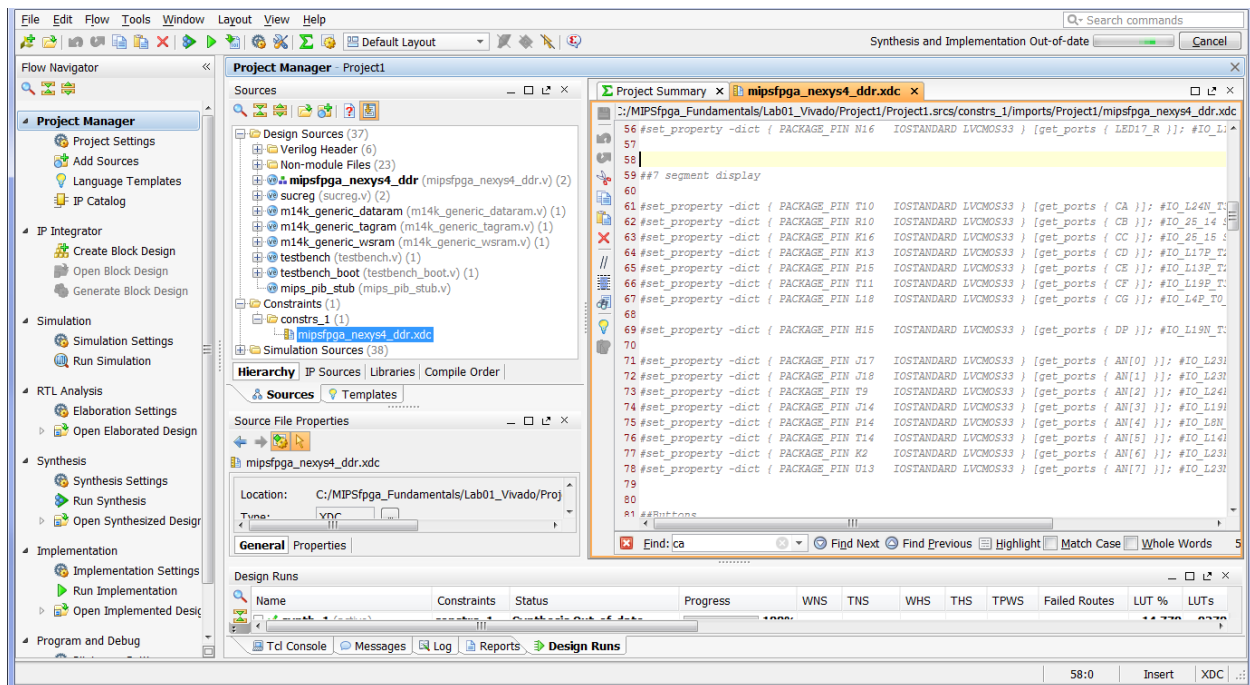


Figure 53. Opening Xilinx Design Constraint file

With the .xdc file open in Vivado, search for (ctrl-F) "segment" to find the listing of 7-segment display outputs, as shown in Figure 53. The information about which pins are connected to the 7-segment displays is already available in the file – it needs only be uncommented. Do so by deleting the # before each line you'd like to use. For example, signal CA that drives the A segment of the 7-segment displays connects to pin T10 on the FPGA by the following line:

```
set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports {
CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
```

This pin is connected via a wire trace to the segment A input of the 7-segment displays on the Nexys4 DDR board. CB should output to the R10 Artix-7 pin, and so on. The signals driving the anodes of the 7-segment displays (AN[7:0]) are also assigned Artix-7 package pins. Leave the DP signal is commented out (#) because it is not used.

Near the bottom of the constraints file, add the following output timing constraints for the 7-segment display signals:

```
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports
{AN[*]}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports
{AN[*]}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CA}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CA}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CB}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CB}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CC}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CC}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CD}]
```

```
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CD}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CE}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CE}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CF}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CF}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CG}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CG}]
```

Testing Seven-Segment Display Functionality

Now test the basic functionality of the 7-segment display hardware by writing a simple assembly program that writes to the 7-segment displays. You will then test your hardware by simulating this simple program **without the bootcode** using XSIM, Xilinx's built-in simulator.

Write a simple MIPS assembly program that enables the 7-segment displays and then writes values to each of the eight digits. Create your program in the following directory:

```
MIPSfpga_Fundamentals\Xilinx\Lab05_7seg\AssemblyExample
```

Copy the entire contents of the ReadSwitches directory from Lab 3 to the AssemblyExample directory as a starting point for your new MIPS assembly program. Modify the main.S file. Then compile the program (and debug as needed) using make.

Extract Machine Code for Simulation

Now convert the MIPS assembly instructions from this program to machine code in order to simulate the instructions on the MIPSfpga core. You can use any method you prefer to convert assembly to 32-bit machine code: by hand, using another simulator (such as QtSpim), extracting the machine code from the executable generated by Codescape, etc. We show you two ways to use Codescape to convert MIPS assembly to machine code. For either method, you must first compile the MIPS assembly code using Codescape (as described in Labs 2-4 and in Section 7.2 of the MIPSfpga Getting Started Guide). In the first method, you then manually extract the program's machine code from the executable. In the second method, you use a script to extract the machine code. For simple programs, the first method is faster.

Method 1: Manually Extract the Machine Code from the Executable

First, we show you how to extract the machine code manually from the executable produced by the Codescape compiler (gcc). Browse to your assembly program's directory:

```
MIPSfpga_Fundamentals\Xilinx\Lab05_7seg\AssemblyExample
```

After compiling the code using 'make', open **FPGA_Ram_dasm.txt** or **FPGA_Ram_modelsim.txt**, the disassembled text representation of the executable. FPGA_Ram_modelsim.txt does not include the interleaved higher-level program code, so it is simpler and easier to read but contains less information. (Note: if those files don't exist, you still need to compile the program using make.) Search for the `main` label, which is the beginning of the user program. Extract these machine instructions and put them in a text file called **ram_reset_init.txt** in the MIPSfpga_Fundamentals\Xilinx\Lab05_7seg\ directory.

We will use this file to initialize the contents of the boot RAM, which holds memory addresses starting at physical address 0x1fc00000. So, when running the simulation, the boot ram will

initialize its contents with these instructions. And upon reset, MIPSfpga will begin executing the first instruction. Note that "relocating" code in this manner only works if there are no jump instructions.

Method 2: Automatically Extract the Machine Code using a Script

We now show you how to use a script to do the same thing that you did manually above. Open a command shell and change to the MIPSfpga_Fundamentals\Scripts directory. For example if MIPSfpga_Fundamentals is on the C:\ drive:

```
cd C:\MIPSfpga_Fundamentals\Scripts
```

Now run the createMemfiles.bat batch script on the 7-segment display AssemblyExample program (located in the MIPSfpga_Fundamentals\Xilinx\Lab05_7seg\AssemblyExample folder). For example, if the MIPSfpga_Fundamentals folder is located on the C:\ drive, type:

```
createMemfiles.bat C:\MIPSfpga_Fundamentals\Xilinx\Lab05_7seg\AssemblyExample
```

This script creates files that list the instructions in the boot RAM (ram_reset_init.txt) and the program RAM (ram_program_init.txt). You can simulate the entire program (boot code plus program code) using both of these files. However, a full simulation of the boot code and the program code is unnecessary. For our purposes here, simulating only the program code suffices to test the added 7-segment display hardware. So, extract the program code only and place it in the boot RAM so that, upon reset, the program code will begin executing.

This extraction process takes about 10 minutes or more, depending on the speed of your computer. Once it completes, the prompt will return in the command window. Now browse to this folder, where the script-generated files are located:

```
MIPSfpga_Fundamentals\Xilinx\Lab05_7seg\AssemblyExample\MemoryFiles
```

The script generated four files:

- ram_reset_init.txt
- ram_program_init.txt
- ram_reset_init.mif
- ram_program_init.mif

The .txt files contain the memory contents of the reset (boot) and program RAMs. ram_reset_init.txt contains the boot code, instructions starting at physical address 0x1fc00000 (virtual address 0xbfc00000 or 0x9fc00000), the value of the PC at reset. ram_program_init.txt contains the program instructions starting at physical address 0x00000000 (virtual address 0x80000000). The .mif files are a similar representation of the same memory called *memory initialization files* used by some CAD tools.

Open FPGA_Ram_dasm.txt and ram_program_init.txt. In FPGA_Ram_dasm.txt, search for "80000000:". Code at this address is part of the exception handler, as shown below.

```
80000000: 3c1b8000    lui    k1,0x8000
80000004: 277b1430    addiu k1,k1,5168
```

```

...
80000220: 3c1b0000    lui    k1,0x0
80000224: 277b0000    addiu  k1,k1,0
...

```

In `ram_program_init.txt`, which describes the program code starting at virtual address `0x80000000` (physical address `0x0`), the top of the file lists these instructions:

```

@0
3c1b8000
277b1430
...

```

`ram_program_init.txt`, lists instructions starting at physical address `0x0` (indicated by the `@0`). MIPS uses byte-addressable memory, but the 256 KB program memory returns 32-bit words (i.e., it is organized as $2^{16} \times 32$ bits). So the memory modules discard the lower 2 bits of the address on the AHB-Lite bus, essentially dividing the address by 4. Thus physical address `0x0000005c` (virtual address `0x8000005c`) is at program RAM memory address `0x5c/4 = 0x17`.

Further down in the `ram_program_init.txt` file, notice the following instructions located at RAM1 address `0x88`.

```

@88
3c1b0000
277b0000
...

```

These correspond to the instructions starting at `0x80000220` (i.e., `0x80000000 + 0x88*4`) in `FPGA_Ram_dasm.txt`.

Now, in `ram_program_init.txt` search for "`@1d7`". These are the instructions located at `0x80000000 + (0x1d7*4) = 0x8000075c`, the starting location of the user's program code. In `FPGA_Ram_dasm.txt`, search for "`8000075c:`" to see all of the program instructions. Now, find the end of your program code in both `FPGA_Ram_dasm.txt` and `ram_program_init.txt`. In the next step, you will extract this program code (from `ram_program_init.txt`) and place it in a new `ram_reset_init.txt` file, that you will simulate.

Now you will extract the machine code version of your program and relocate it to virtual address `0xbfc00000`, so that your code (instead of the boot code) simulates upon reset. To do so, create a new file called **`ram_reset_init.txt`** that contains the program code (that you just located in `ram_program_init.txt`). Place this new file in:

```

MIPSfpga_Fundamentals\Xilinx\Module05_7seg\ram_reset_init.txt

```

Remove all of the address directives (`@1d7`, etc.). With no addresses indicated, the memory contents default to begin at address 0 relative to the start of the boot code (starting at virtual address `0xbfc00000`).

You will point the simulation to initialize the reset/boot RAM (RAM0) with the ram_reset_init.txt file you just created. Note again that we can only relocate code in this manner if there are no jump instructions.

Simulation

Now you will run the user code in simulation on the MIPSfpga system to test your new 7-segment display hardware. Before simulating, you will add your new and modified Verilog files to the project you created in Lab1.

Open the Vivado project you created in Lab 1, located in MIPSfpga_Fundamentals/Xilinx/Lab01_Vivado/Project1. Add the 7-segment display decoder and 7-segment display timer modules that you built earlier in the lab (by selecting Add Sources under the Project Manager pane in Vivado).

Now, you are ready to simulate your new hardware. In the Project Manager window, scroll down to and expand Simulation Sources and sim_1. Make sure testbench is bold, indicating that it is the top-level simulation module, as shown in Figure 54.

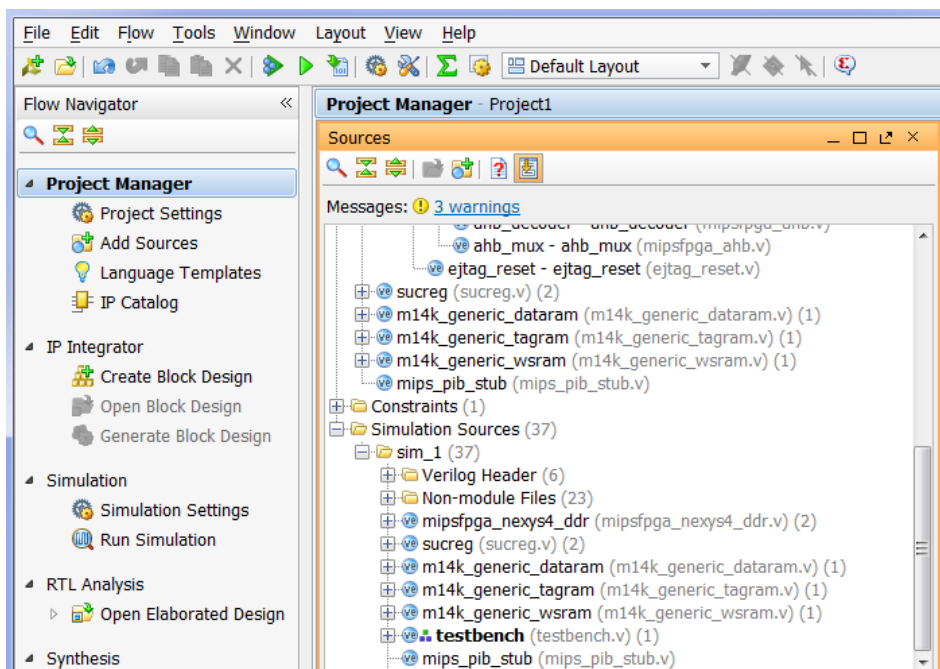


Figure 54. testbench as top-level module for simulation

Modify testbench.v to instantiate your modified MIPSfpga system (mipsfpga_sys).

If you added a memory initialization file for simulation in Lab 1, remove it by expanding Simulation Sources → sim_1 → Text and deleting any existing files (i.e., ram_reset_init.txt) as shown in Figure 55. Right-click on ram_reset_init.txt and select Remove File from Project and click OK.

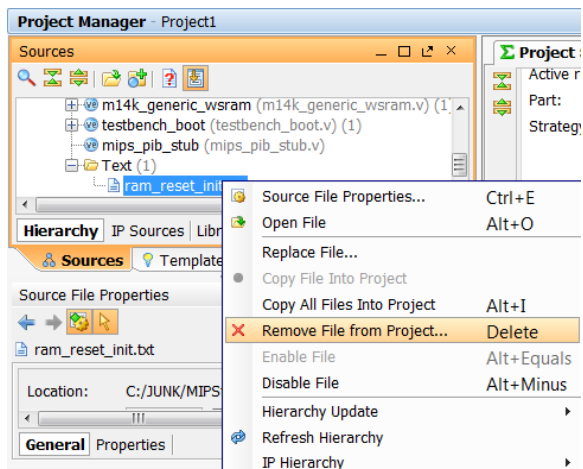


Figure 55. Remove existing memory definition file

Now add the `ram_reset_init.txt` file that you just created (located in `MIPSfpga_Fundamentals\Xilinx\Module05_7seg\`) as a simulation source. Refer to Lab 1 if you've forgotten how to do this.

Run the simulation and check that the MIPSfpga system outputs the correct values for the 7-segment display signals (`IO_7SEGEN_N` and `IO_7SEG_N`). If they don't, debug your modules. To debug, you'll likely want to view signals from lower levels in the hierarchy. Again, refer to Lab 1 instructions if you've forgotten how to do this.

As you're simulating, remember that the timing of displaying each of the 7-segment display digits is much slower than the rest of the system (~2 ms versus 20 ns cycle time).

Running the Example Assembly Program on the MIPSfpga System

Now that you have simulated and tested your added hardware to support writing values to the 7-segment displays, you are ready to run your MIPS assembly program on the MIPSfpga system in hardware.

Open a command shell and change to the following directory:

```
MIPSfpga_Fundamentals\Scripts\Nexys4_DDR
```

At the prompt, type:

```
loadMIPSfpga.bat C:\MIPSfpga_Fundamentals\Xilinx\Lab05_7seg\AssemblyExample
```

Make sure that your MIPS assembly program operates correctly in hardware. If not, you get another chance to practice your debugging skills.

Example C Program on the MIPSfpga Core

Now write a simple C program that writes to the 7-segment displays. Compile and debug your program and then run and test your C program in hardware.

9. Write a Program Using 7-segment Displays

Now you will write two new C programs to exercise the 7-segment display capabilities:

1. The first program, called **SwTo7segHex**, should write the **hexadecimal** value of the 16 switches to the 7-segment displays.
2. The second program, called **SwTo7segDec**, should write the **decimal** value of the 16 switches to the 7-segment displays.

When you are done, run, debug, and test your programs on your modified MIPSfpga system.

MIPSfpga

by Imagination

Lab 9

Porting MIPSfpga to Other FPGA Boards



These materials produced in association with Imagination.
Join our University community for more resources.

community.imgtec.com/university

MIPSfpga Lab 9: Porting MIPSfpga to Other FPGA Boards

Introduction

This lab describes how to port the MIPSfpga system onto other FPGA boards. You may choose to use a board other than the Nexys4 DDR board because of, for example, wanting a lower-cost option or having existing availability of other boards.

We will describe how to port the MIPSfpga system to Digilent's **Basys3** board and Digilent's **Nexys4** board, which is the predecessor to the Nexys4 DDR board. You may follow similar steps to port the MIPSfpga system to other boards based on Xilinx FPGAs. Table 7 gives an overview of the features of the Nexys4 and Basys3 FPGA boards, as well as the Nexys4 DDR board that we've been using in the prior labs. If desired, the prior labs could have been completed on either the Nexys4 or Basys3 FPGA boards – or on other FPGA boards as well.

Table 7. FPGA Boards

Board	Overall specifications	Web Link	Cost
Nexys4 DDR	<ul style="list-style-type: none"> • FPGA: Artix-7 (XC7A100T-CSG324) • Amount of block RAM: 607 KB • # of Logic Cells: 101k • 7-segment displays: 8 • Switches: 16 • Pushbuttons: 5 • LEDs: 16 • PMOD Connectors: 5 	http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS4DDR	\$159 (academic), \$320 (non-academic)
Nexys4	Similar to Nexys4 DDR (for example, all of the above specifications are the same)	http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS4	\$179 (academic), \$320 (non-academic)
Basys3	<ul style="list-style-type: none"> • FPGA: Artix-7 (XC7A35T-CPG236) • Amount of block RAM: 225 KB • # of Logic Cells: 33k • 7-segment displays: 4 • Switches: 16 	http://www.digilentinc.com/Products/Detail.cfm?Prod=BASYS3	\$79 (academic), \$149 (non-academic)

	<ul style="list-style-type: none"> • Pushbuttons: 5 • LEDs: 16 • PMOD Connectors: 4 		
--	-------------------------------------------------------------------------------------------------------------------------------------------	--	--

Porting MIPSfpga to Digilent's Basys3 Board

In this section we show you how to port the MIPSfpga system onto Digilent's Basys3 board, shown in Figure 56.

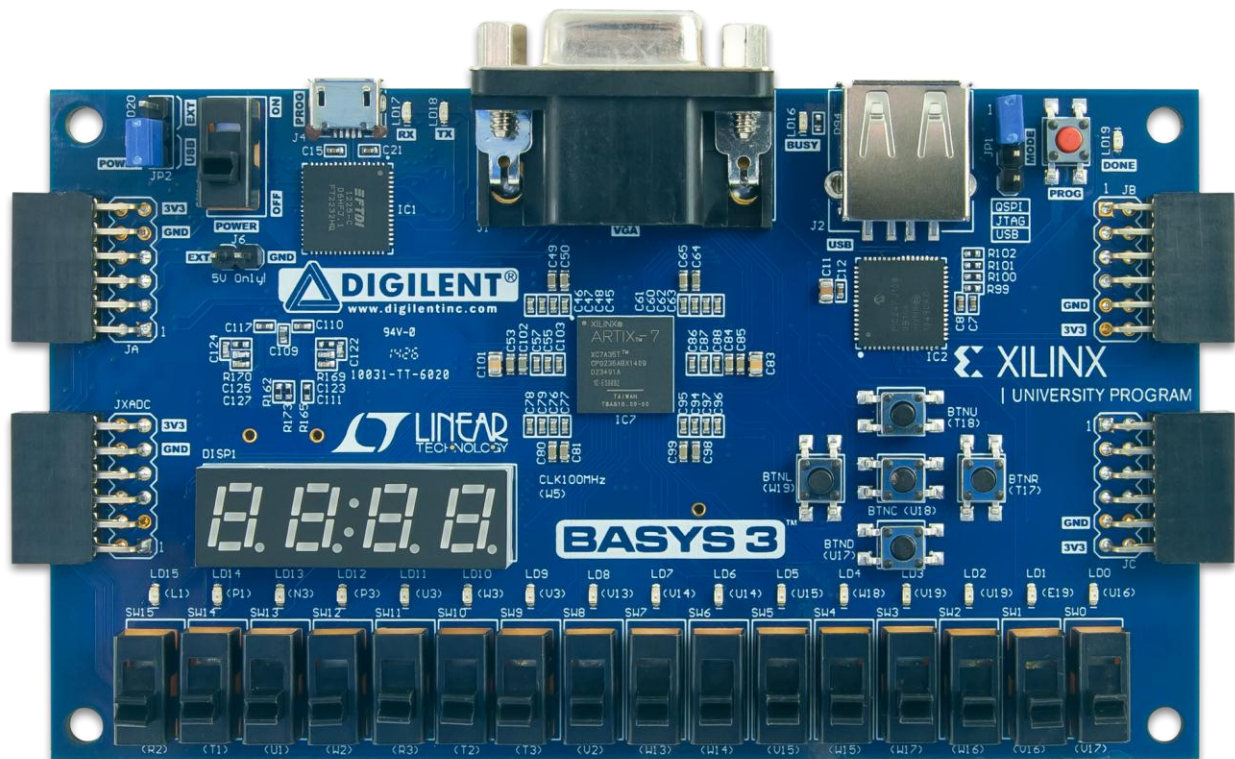


Figure 56. Digilent's Basys3 Board

As listed in Table 7, the Basys3, like the Nexys4 DDR board, is also built around Xilinx's Artix7 FPGA, however it is a smaller version of the FPGA on the Nexys4 DDR board. The Basys3's Artix7 FPGA has about a third of the logic and memory as the one on the Nexys4 DDR board (225 KB of block RAM and 33K configurable logic blocks (CLBs) on the Basys3 vs. 607 KB of block RAM and 101K CLBs on the Nexys4 DDR). The MIPSfpga hardware fits easily on either board, however, the amount of block RAM needs to be reduced in order to fit on the Basys3 board.

MIPSfpga Modifications for the Basys3 Board

To port the MIPSfpga system to the Basys3 board, we must:

Step 1. Write a *wrapper module* that maps the MIPSfpga I/O to the Basys3 board I/O

Step 2. Decrease the memory size of the MIPSfpga system to fit on the Basys3 board

Step 3. Add a constraints file that maps the board I/O to the correct FPGA pins

We describe each of these steps in detail and then show how to download the MIPSfpga system onto the Basys3 board. We also describe how to build a Vivado project and compile the MIPSfpga system for the Basys3 board.

Step 1. Basys3 Wrapper Module

First, we write a wrapper module that maps the MIPSfpga I/O to the I/O on the Basys3 board. Browse to the MIPSfpga_Fundamentals\Xilinx\Lab09_PortingMIPSfpga\Basys3 folder and open the `mipsfpga_basys3.v` file.

First, observe the `mipsfpga_basys3` module inputs and outputs. These signals are the interfaces to the Basys3 board.

```
module mipsfpga_basys3( input      clk,
                      input      btnU, btnD, btnL, btnR, btnC,
                      input  [15:0] sw,
                      output [15:0] led,
                      inout  [ 5:0] JB
);
```

`clk` is the onboard 100 MHz clock. `btnU`, `btnD`, `btnL`, `btnR`, and `btnC` are the names of the up, down, left, right, and center push buttons on the Basys3 board. We use `btnC` to reset the processor.

Input signal `sw[15:0]` are the 16 switches, `led[15:0]` are the 16 LEDs, and so on. The `JB` signals are the PMODB connector pins that connect to the EJTAG signals needed by the Bus Blaster. Notice that the JB connector is on the upper right of the Basys3 board (see [Figure 56](#)) and on the lower right of the Nexys4 DDR board.

Next, the phase-lock-loop, `clk_wiz_0`, is instantiated to create the 50 MHz MIPSfpga system clock from the onboard 100 MHz clock.

```
clk_wiz_0 clk_wiz_0(.clk_in1(clk), .clk_out1(clk_out));
```

The following lines manually connect the `clk` pin (`JB[3]`) to an input buffer (IBUF) and then to a clock buffer (BUFG) to address the problem that the `JB[3]` pin carries a clock signal but is

not connected to a clock buffer.

```
IBUF IBUF1(.O(tck_in), .I(JB[3]));  
BUFG BUFG1(.O(tck), .I(tck_in));
```

Finally, the heart of the module is to instantiate the MIPSfpga system (`mipsfpga_sys`) and connect it to the Basys3 I/O.

```
mipsfpga_sys mipsfpga_sys (  
    .SI_Reset_N(~btnC),  
    .SI_ClkIn(clk_out),  
    .HADDR(),  
    .HRDATA(),  
    .HWDATA(),  
    .HWRITE(),  
    .EJ_TRST_N_probe(JB[4]),  
    .EJ_TDI(JB[1]),  
    .EJ_TDO(JB[2]),  
    .EJ_TMS(JB[0]),  
    .EJ_TCK(tck),  
    .SI_ColdReset_N(JB[5]),  
    .EJ_DINT(1'b0),  
    .IO_Switch({2'b0, sw}),  
    .IO_PB({btnU, btnD, btnL, 1'b0, btnR}),  
    .IO_LEDR(led),  
    .IO_LEDG()  
);
```

Step 2. Decrease Memory

The Basys3 board only has 225 KB of block RAM instead of the 607 KB of the Nexys4 DDR board. Thus, the two memory blocks (128 KB for boot RAM and 256 KB for program RAM) will not fit on the Basys3 board. Luckily, the boot code can fit in 32 KB and we can limit our program needs to 64 KB. Thus, the total memory need (32 KB + 64 KB = 96 KB) fits on the Basys3 board. The remaining $225 - 96 = 129$ KB of block RAM can be used for other memory needs of the MIPSfpga system, such as caches.

We reduce the amount of memory by modifying the memory sizes declared in the Verilog header file. Again, browse to the

MIPSfpga_Fundamentals\Xilinx\Lab09_PortingMIPSfpga\Basys3 directory and open the `mipsfpga_ahb_const.vh` file. The size of the reset (boot) RAM address is 13 bits. So the boot RAM has 2^{13} 32-bit words = 2^{15} bytes = 32 KB.

```
`define H_RAM_RESET_ADDR_WIDTH      (13)
```

The size of the program RAM address is 14 bits. So the program RAM has 2^{14} 32-bit words = 2^{16} bytes = 64 KB.


```
`define H_RAM_ADDR_WIDTH          (14)
```

Step 3. Basys3 Constraints File

As the last step, we add a constraints file that maps the wrapper module's (mipsfpga_basys3) I/O signal names to the correct FPGA pins on the Basys3 FPGA board. Again, browse to the MIPSfpga_Fundamentals\Xilinx\Lab09_PortingMIPSfpga\Basys3 directory and open the mipsfpga_basys3.xdc Xilinx Design Constraints file. This file maps the FPGA pins to the inputs and outputs of the mipsfpga_basys3 wrapper module. For example, the following line maps the input `clk` to FPGA package pin W5, which is fed by the 100 MHz clock (period = 10 ns) on the Basys3 board.

```
set_property PACKAGE_PIN W5 [get_ports clk]

set_property IOSTANDARD LVCMOS33 [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports clk]
```

The following line addresses the issue that the tck pin of the EJTAG interface is not connected to an FPGA pin with a clock buffer.

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets tck_in]
```

The lines below map the switch inputs (`sw[15:0]`) to the FPGA pins that are physically wired to the switches on the Basys3 board. For example `sw[0]` is connected to the Artix7 package pin V17, `sw[1]` to pin W16, and so forth. They all use LVCMOS 3.3V signal levels.

```
## Switches
set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
set_property PACKAGE_PIN W16 [get_ports {sw[2]}]
...

```

The timing constraints for FPGA I/O are specified at the bottom of the file.

Loading the MIPSfpga System onto the Basys3 Board

A pre-compiled bitfile called `mipsfpga_basys3.bit` contains the MIPSfpga system targeted to the Basys3 board. It is provided in this directory:

```
MIPSfpga_Fundamentals\Xilinx\Lab09_PortingMIPSfpga\Basys3
```

Open Vivado and load the bitfile onto the Basys3 board. Press the center pushbutton (BTNC) on

the Basys3 board to reset the processor. You should now see the LEDs display incremented values. To use Codescape and the Bus Blaster probe to program the MIPSfpga core, connect the Bus Blaster probe to the PMOD B connector at the top right of the board (labeled JB). Follow the same instructions as provided in previous labs (see Lab 2, for example) to download and debug programs on the MIPSfpga system running on the Basys3 board.

Setting up a Vivado Project for MIPSfpga on the Basys3 Board

Use similar steps to those described in Lab 1 to set up a Vivado project targeted to the Artix7 on the Basys3 board. Before setting up the project, make a copy of the MIPSfpga_Fundamentals\rtl_up directory and name the copy rtl_up_basys3. Copy the mipsfpga_basys3.v and mipsfpga_ahb_const.vh files to the rtl_up_basys3 folder.

Now open Vivado and create a new project. Add all the files in the rtl_up_basys3 folder as source files. Add mipsfpga_basys3.xdc from the MIPSfpga_Fundamentals\Xilinx\Lab09_PortingMIPSfpga\Basys3 folder as the constraints file. You will also need to add a 50 MHz PLL (see Lab 1 for instructions). The project target device for the Basys3 board is the **xc7a35tcpg236-1** Artix7 FPGA. After the project is set up, compile, synthesize, and layout the design by generating a bitfile for the project as described in Lab 1.

MIPSfpga on the Nexys4 Board

Some universities or laboratories may have legacy Nexys4 boards, the predecessors to the Nexys4 DDR board. We provide the wrapper file and Xilinx Design Constraint file for the Nexys4 board for your convenience. They are located in this directory:

```
MIPSfpga_Fundamentals\Xilinx\Lab09_PortingMIPSfpga\Nexys4
```

The target Artix-7 FPGA is the same as the one on the Nexys4 DDR board: **xc7a100tcsg324-1**. A precompiled bitfile (mipsfpga_nexys4.bit) is also provided in that directory.

Other Xilinx FPGAs and FPGA Boards

You may follow the same methods described here to target different FPGA boards and Xilinx FPGAs. Specifically, you will need to write a wrapper module, include the board-specific Xilinx Design Constraints (.xdc) file, and possibly modify the amount of memory used by the MIPSfpga system. Some FPGAs may be too small for the MIPSfpga system to fit.