

ECE 540 Getting Started with Hardware Development

Setting up a Vivado Project for MIPSfpga

Original guide produced by Imagination Technologies,
Inc.

(<http://imgtec.com>)

Modified for ECE 540 by:

Yiwei Li, Srivatsa Yogendra, Roy Kravitz,
Melih Erdogan and Dan Hammerstrom

Note: Some of the window formats and Button
positions have changed, but you should be able to
make the appropriate translation.

MIPSfpga
by Imagination

Setting up a Vivado Project

Introduction

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 2018.2.

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 **hardware/rtl_up** folder to the working directory you plan to use for this course.

The Verilog files in the **hardware/rtl_up** directory describe the MIPSfpga system and are the design source files for the Vivado project you are about to create.

Step 1. Create Vivado project

Start Vivado. Open a new project by choosing **File → New Project** (see [Figure 1](#), same for 2018.2).

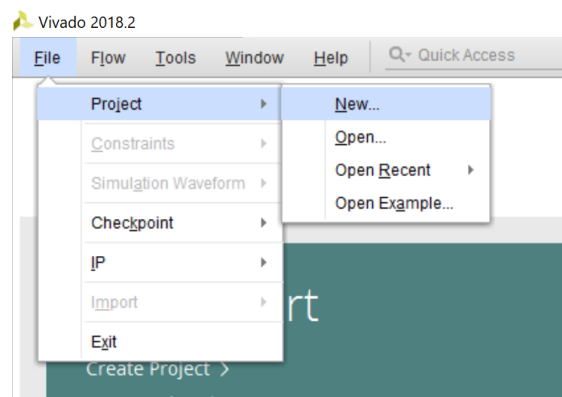


Figure 1. Create new Vivado project

Click **Next**. Browse to the project folder (you can select any directory you want to save your project) and place the new project (**Project1** or a project name of your choosing), in that folder, as shown in [Figure 2](#). Click the Create Project subdirectory box and click Next, as shown below.

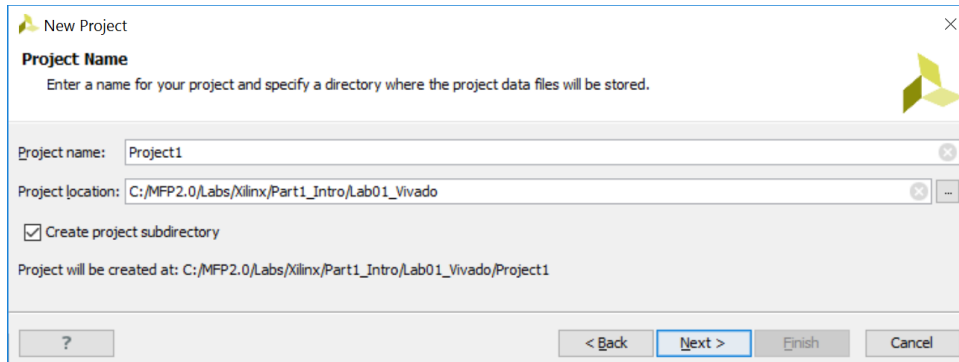


Figure 2. Create Vivado project directory

In the next window, leave **RTL Project** selected and click **Next**. Now in the **Add Sources** window, click on **Add Directories** ([Figure 3](#)).

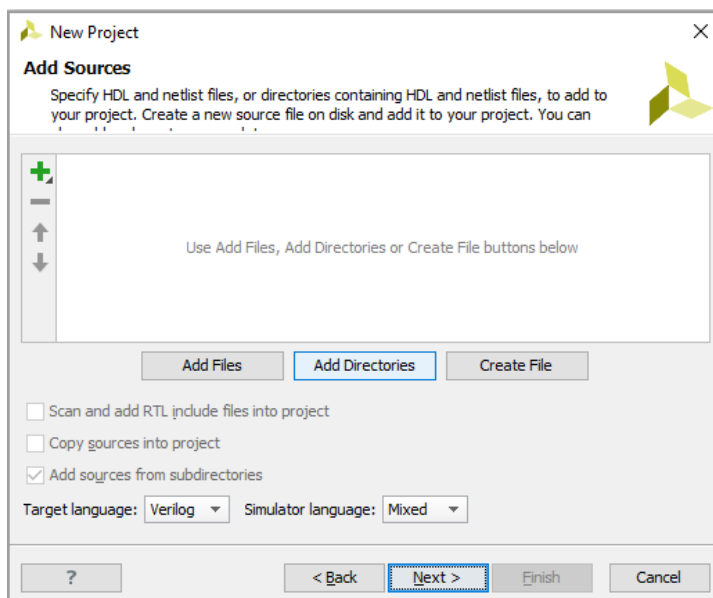


Figure 3. Add Directories of Verilog files

Browse to your copy of the **rtl_up** directory. Select the **core**, **system**, **boards\nexys4_dds**, and **testbench** directories, as shown in [Figure 4](#).

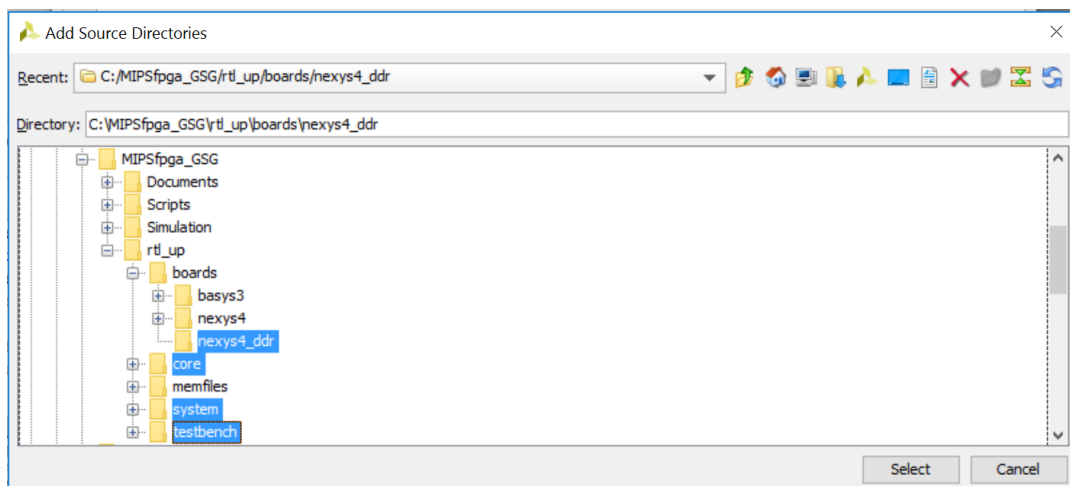


Figure 4. Adding Verilog files to project

In the Add Sources window, make sure the **Copy the sources into Project** box is **not** selected (see Figure 5). The project should refer to the Verilog (.v) and Verilog header (.vh) files located in your **rtl_up** directory – do **not** make a local copy of the files in the Vivado project. Also make sure the Add sources from subdirectories box is selected, and click **Next**.

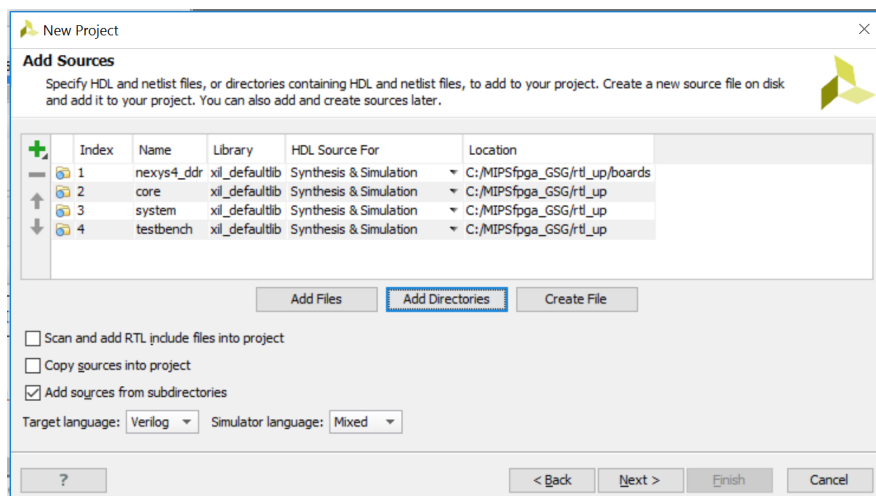


Figure 5. Adding Sources – do not copy sources into project

In the **Add Constraints (optional)** window, click on **Add Files**. Browse to **<your path>\rtl_up\boards\nexys4_ddr** directory, and select the **mfp_nexys4_ddr.xdc** file and click **Next** (see Figure 6). This constraints file maps the Verilog ports to pins on the FPGA and specifies the clock characteristics and other timing constraints.

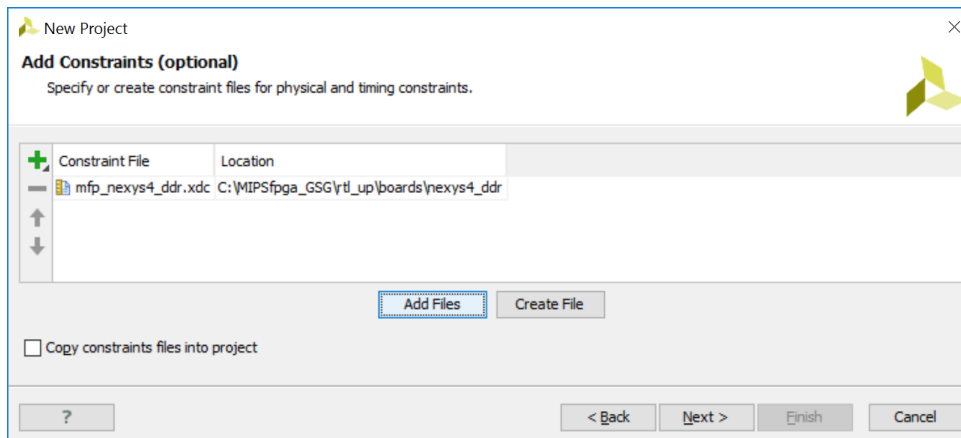


Figure 6. Add Xilinx Design Constraints (.xdc) file to Vivado project

Click on the **Copy constraints files into project** box (see Figure 7) and click **Next**.

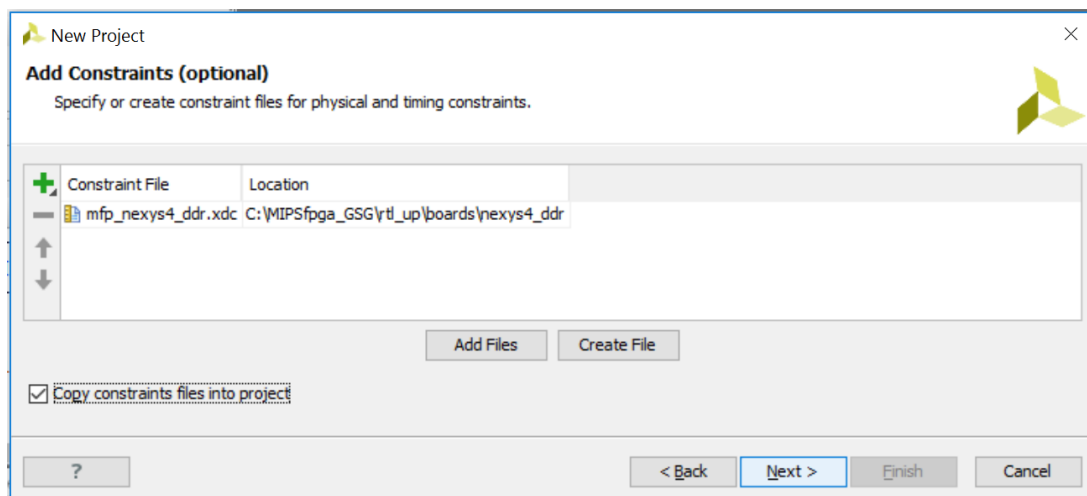


Figure 7. Copy .xdc file into Vivado project

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: **xc7a100tcs324-1**, as shown in Figure 8. Select the part, as shown, and click **Next**.

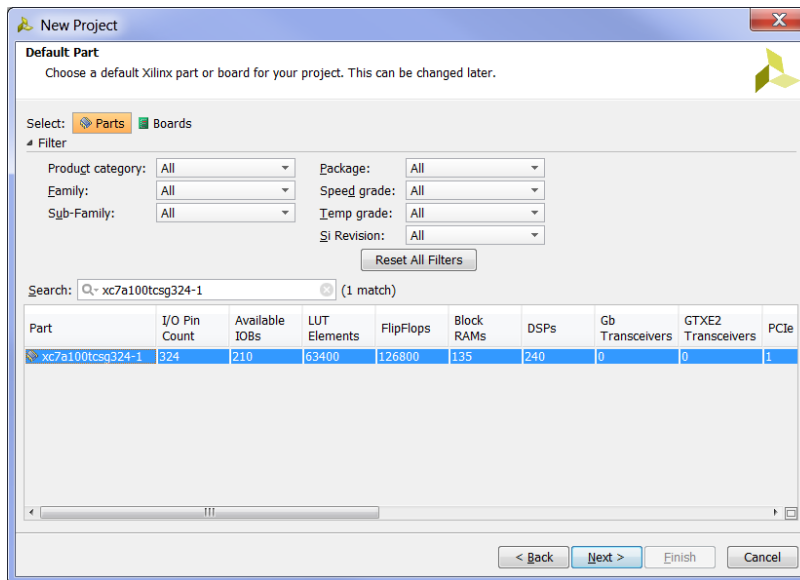


Figure 8. Selecting the Artix-7 FPGA

"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 9](#)).

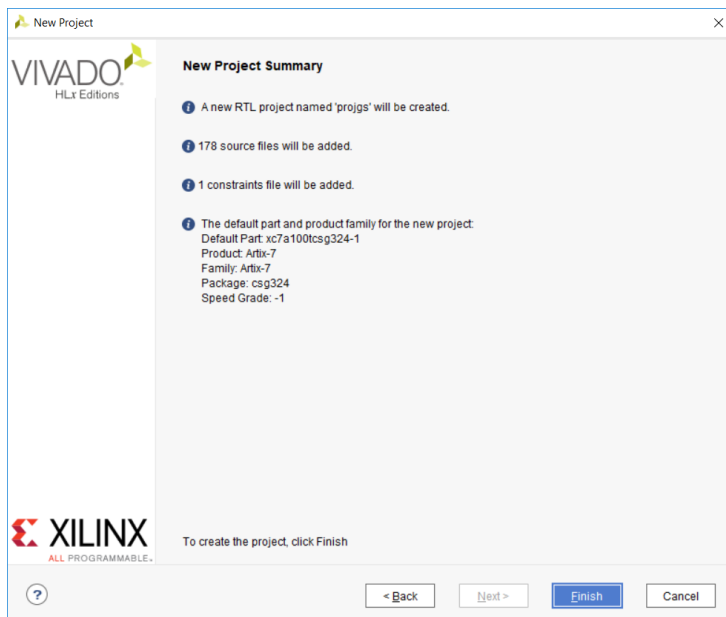


Figure 9. New Project Summary window

After the project initializes, notice the modules listed in the Project Manager Sources window. This lists the hierarchy (modules and sub-modules) of the Project. The **mfp_nexys4_ddr** module

should be bold, which indicates that it is the top-level module (see Figure 10). If it is not highlighted, you can right-click on that module and select **Set as Top** in the pull-down menu. This will set that module as the top-level module to synthesize, compile, and download to the Nexys4 DDR board.

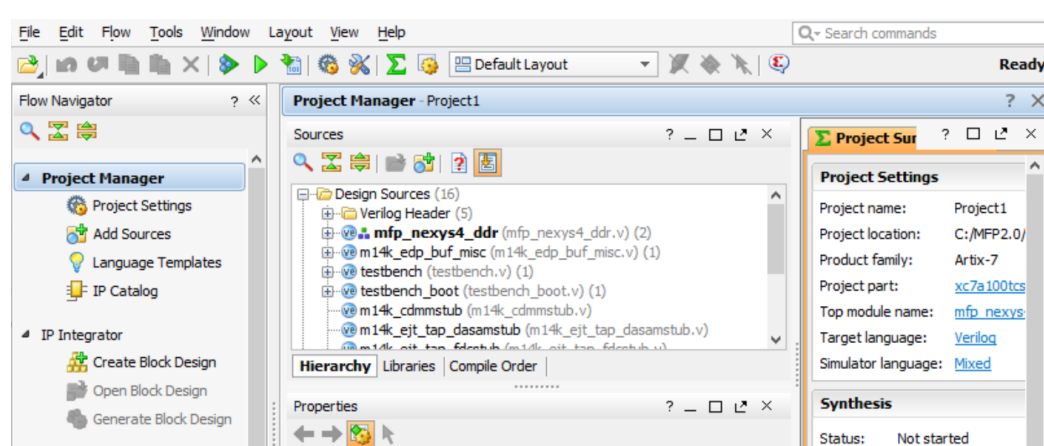


Figure 10. **mfp_nexys4_ddr** as top-level module for synthesis, implementation, and bitstream generation

Step 2. Add a Clock Block to create the 50 MHz MIPSfpga Clock

Now that you've created the project, the next step is to add a PLL (or mixed-mode clock manager – MMCM) 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 11.

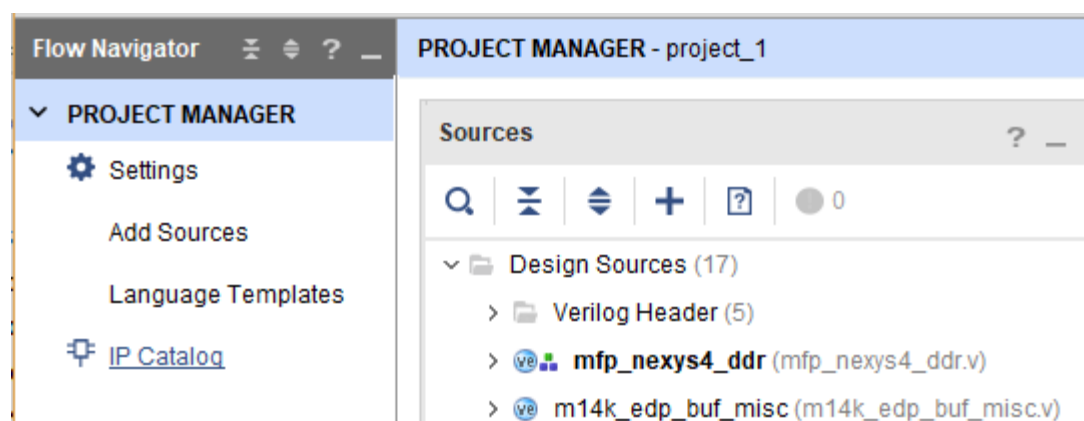


Figure 11. IP Catalog

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 12.

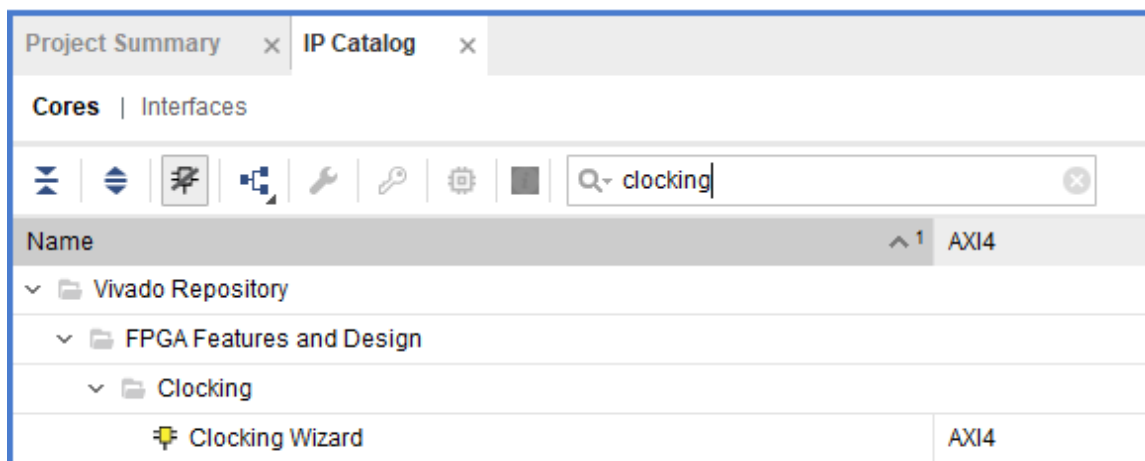


Figure 12. Clocking Wizard

The Clocking Wizard window will pop up, as shown in Figure 13. You can select either MMCM or PLL (shown). Leave the Input Clock information as the default (100 MHz), as shown in Figure 13.

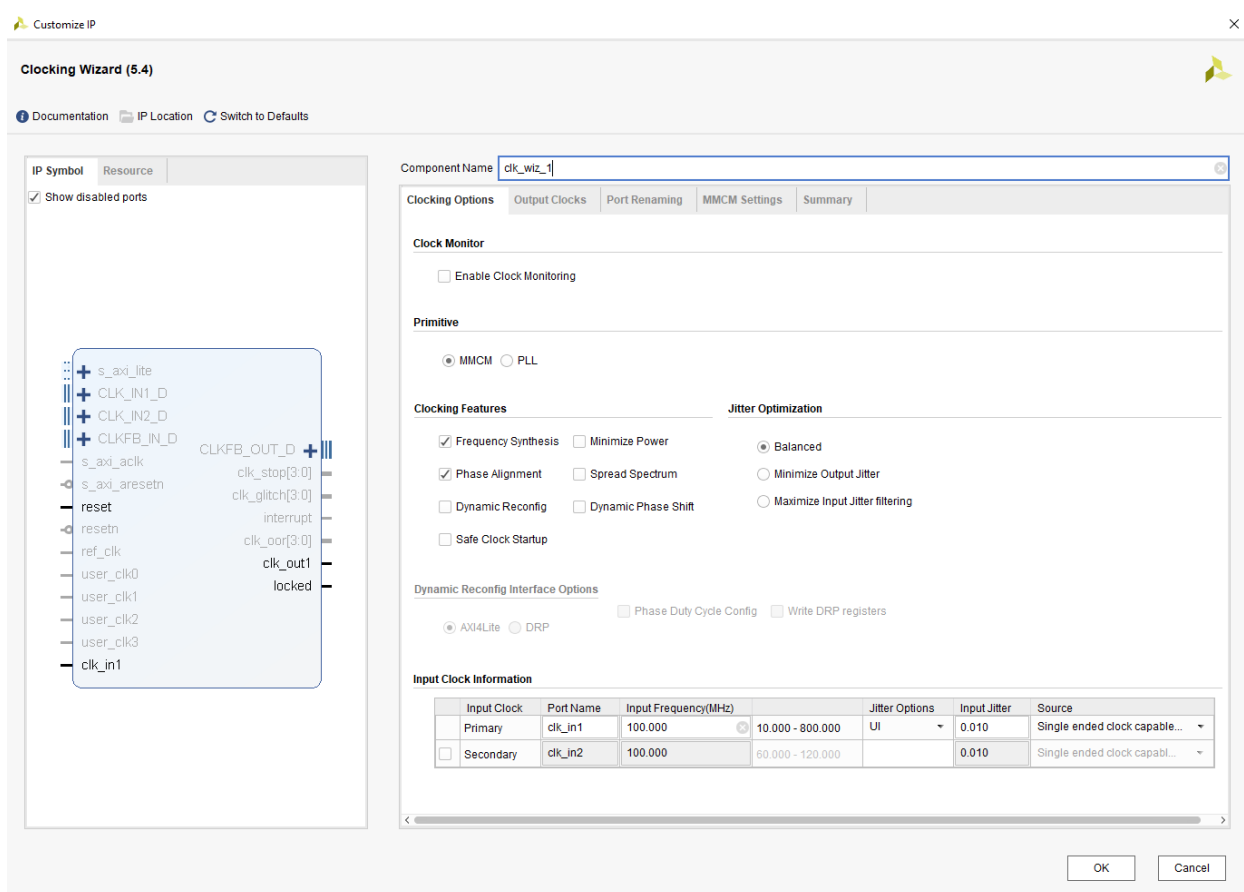


Figure 13. Clocking Wizard window

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 14](#).

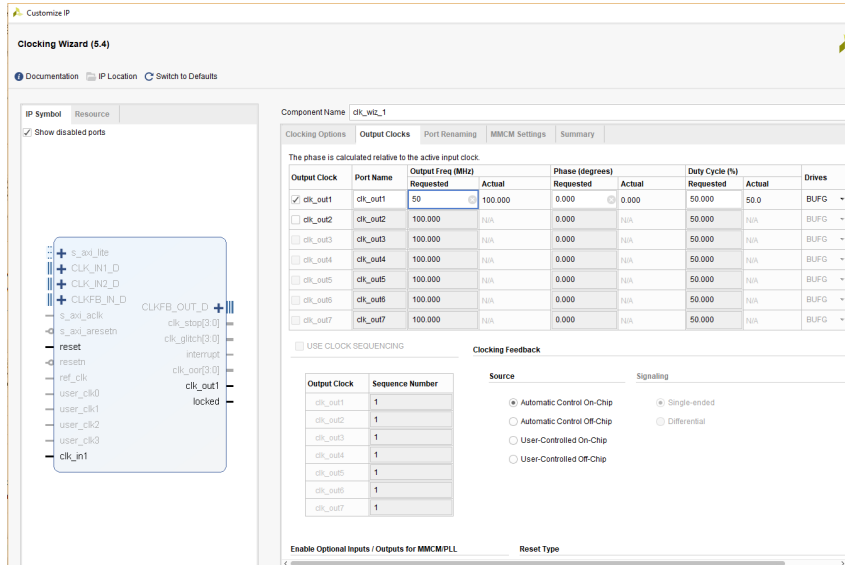


Figure 14. Select output clock frequency

Scroll down in the same tab (Output Clocks) and **deselect** reset and locked, as shown in [Figure 15](#). Then click **OK** to complete the creation of the PLL (or MMCM).

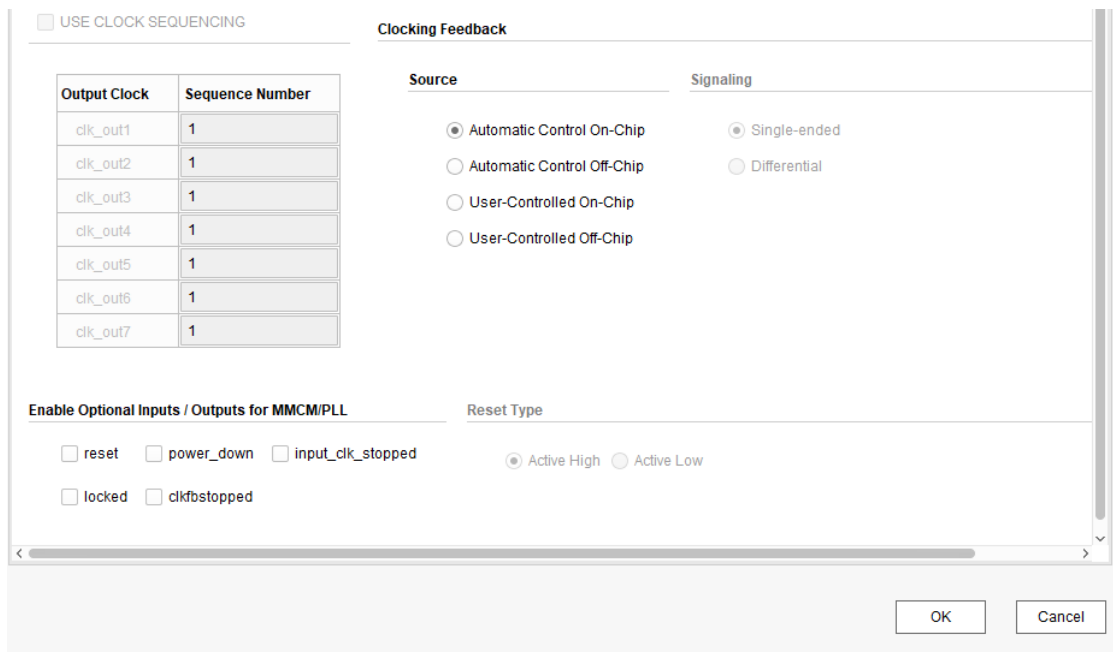


Figure 15. Deselect reset and locked

You then will see a “Create Directory” window, click **OK**.

A pop-up window will prompt you to "Generate Output Products", as shown in [Figure 16](#). Click **Generate**.

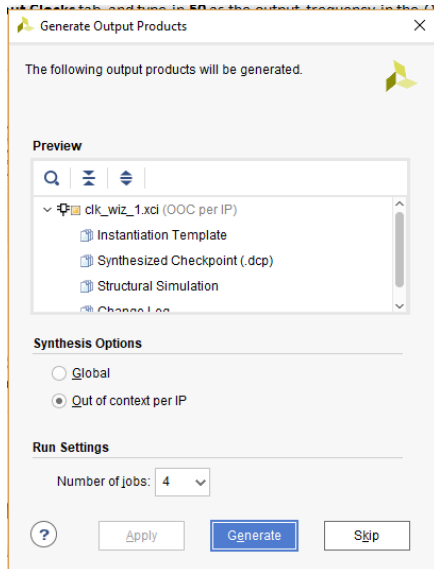


Figure 16. Generate PLL

A window will pop up that says "Out-of-context module run was launched for generating output products," as shown in [Figure 17](#). Click **OK**.

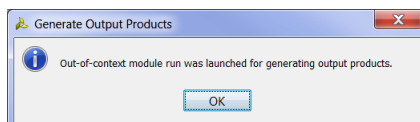


Figure 17. Out-of-context generation of PLL

Step 3. 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, Sources window pane, scroll down to **Simulation Sources** and expand it and the **sim_1** folder. Right-click on testbench.v and **set it as the top-level module** for simulation, as shown in [Figure 18](#). Notice that the testbench entry is now bold and has been moved to the top of the hierarchy.

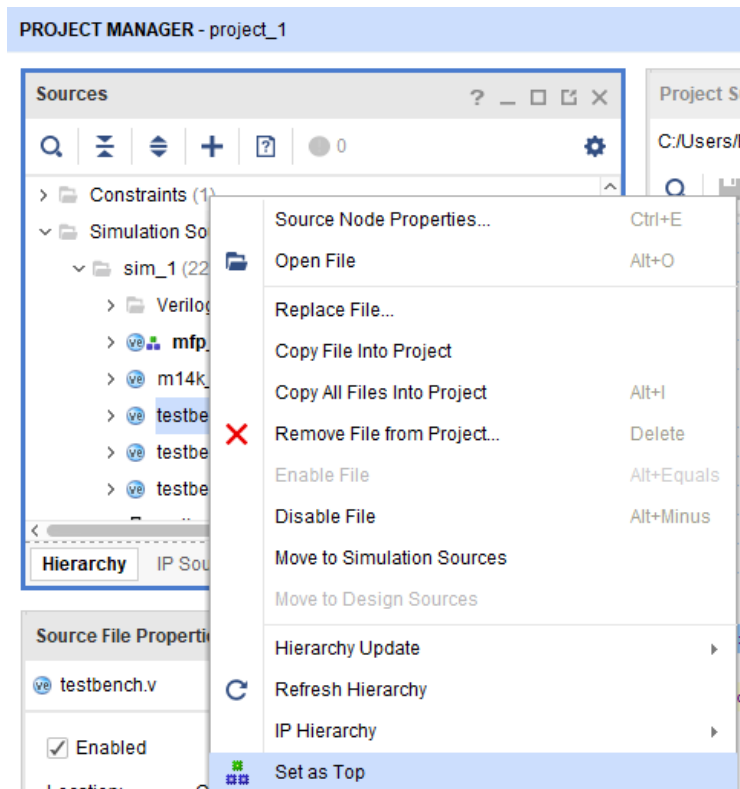


Figure 18. Setting the top-level module for simulation

Now add the memory files that define the program. Click on **Add Sources** in the Flow Navigator window on the left, select **Add or create simulation sources** option (see [Figure 19](#)), and then click **Next**.

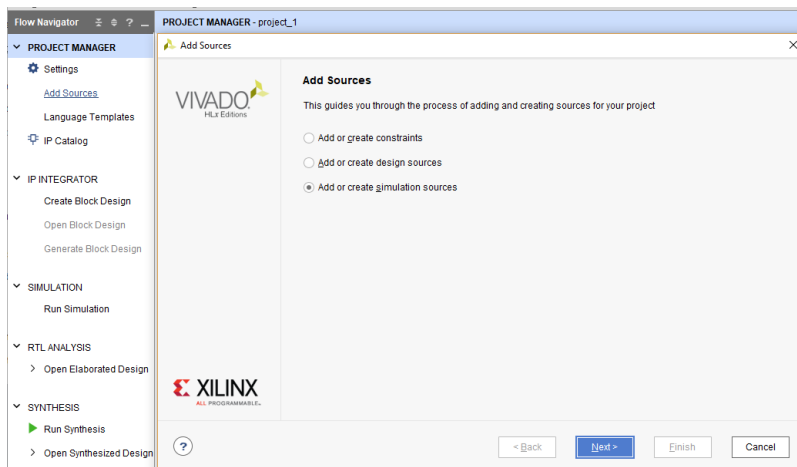


Figure 19. Add simulation sources

Click on **Add Files**, select **All Files** in the *Files of type* filter, and browse to **<your path>\rtl_up\memfiles\1_IncrementLEDs**, and select (shift-click) all of the .txt files: ram_b0.txt, ram_b1.txt, ram_b2.txt, and ram_b3.txt, and click **OK** (see [Figure 20](#)).

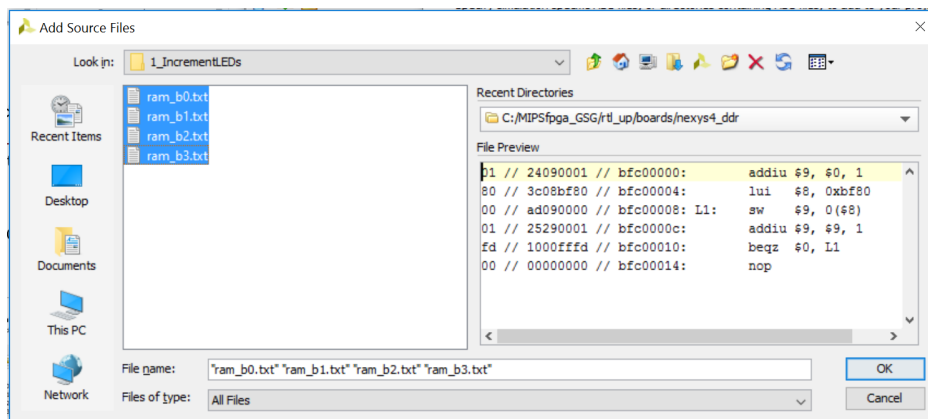


Figure 20. Selecting simulation files that define the memory contents

Leave the Copy sources into project box unselected but leave the Include all design sources for simulation box checked, as shown in [Figure 21](#). Then click **Finish**.

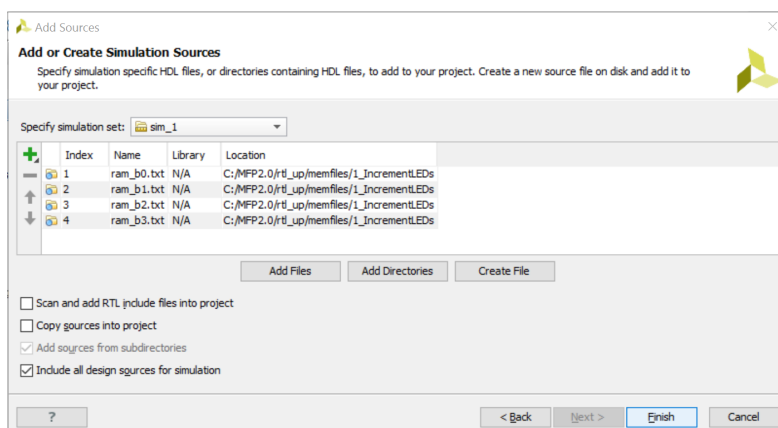


Figure 21. Adding simulation source

These text files contain the instructions that will be loaded into MIPSfpga's memory. ram_b0.txt – ram_b3.txt show the byte-wide versions of the memories. ram_b0.txt holds the least significant byte (LSB) of each instruction, ram_b1.txt the next most significant byte, and so on. This program, as shown again in [Figure 22](#) for your convenience, writes incremented values to memory address 0xbf800000. 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) {</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</pre>
---	--

<pre> *dest = val; val = val + 1; } </pre>	<pre> beqz \$0, L1 # branch to L1 nop # branch delay slot </pre>
--	--

Figure 22. Increment LEDs program

The equivalent machine code for the IncrementLEDs program is given in [Figure 23](#).

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
1000fffd	// bfc00010:	beqz \$0, L1 # branch to L1
00000000	// bfc00014:	nop # branch delay slot

Figure 23. MIPS machine code

Expand the hierarchy under *Simulation Sources* and observe that these four text files are added in a separate sub-folder called Text and it contains the machine code (see [Figure 24](#)). Now you are ready to run the simulation of the MIPSfpga system running that program.

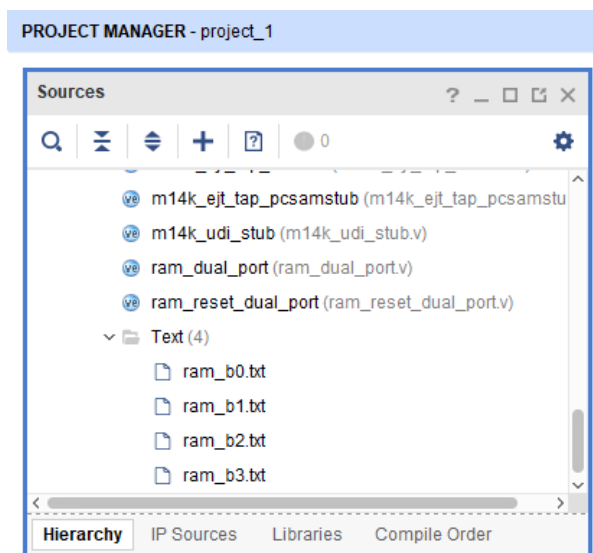


Figure 24. Text file as simulation source

Right click on SIMULATION and select Simulation Settings.

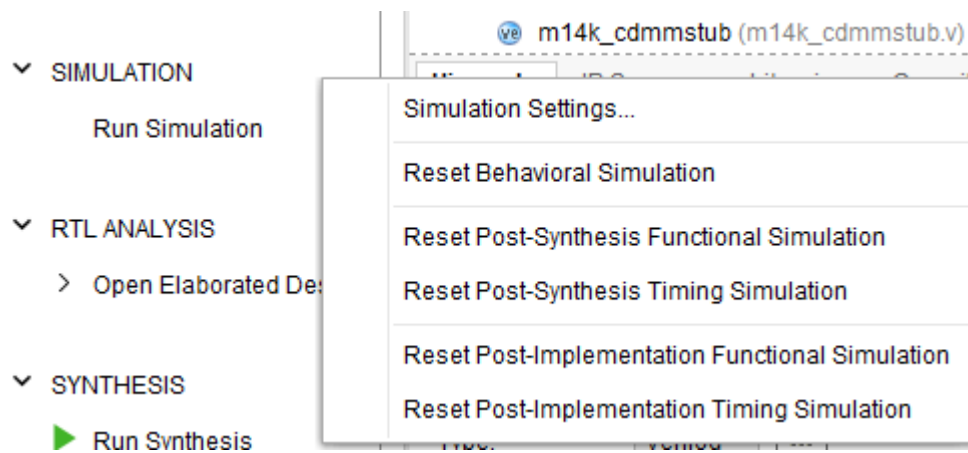


Figure 25. Simulation Settings

The simulation settings window will show up, as shown in [Figure 26](#). Click on the Simulation tab and set **the simulation run time to 2000 ns**. Click **OK**.

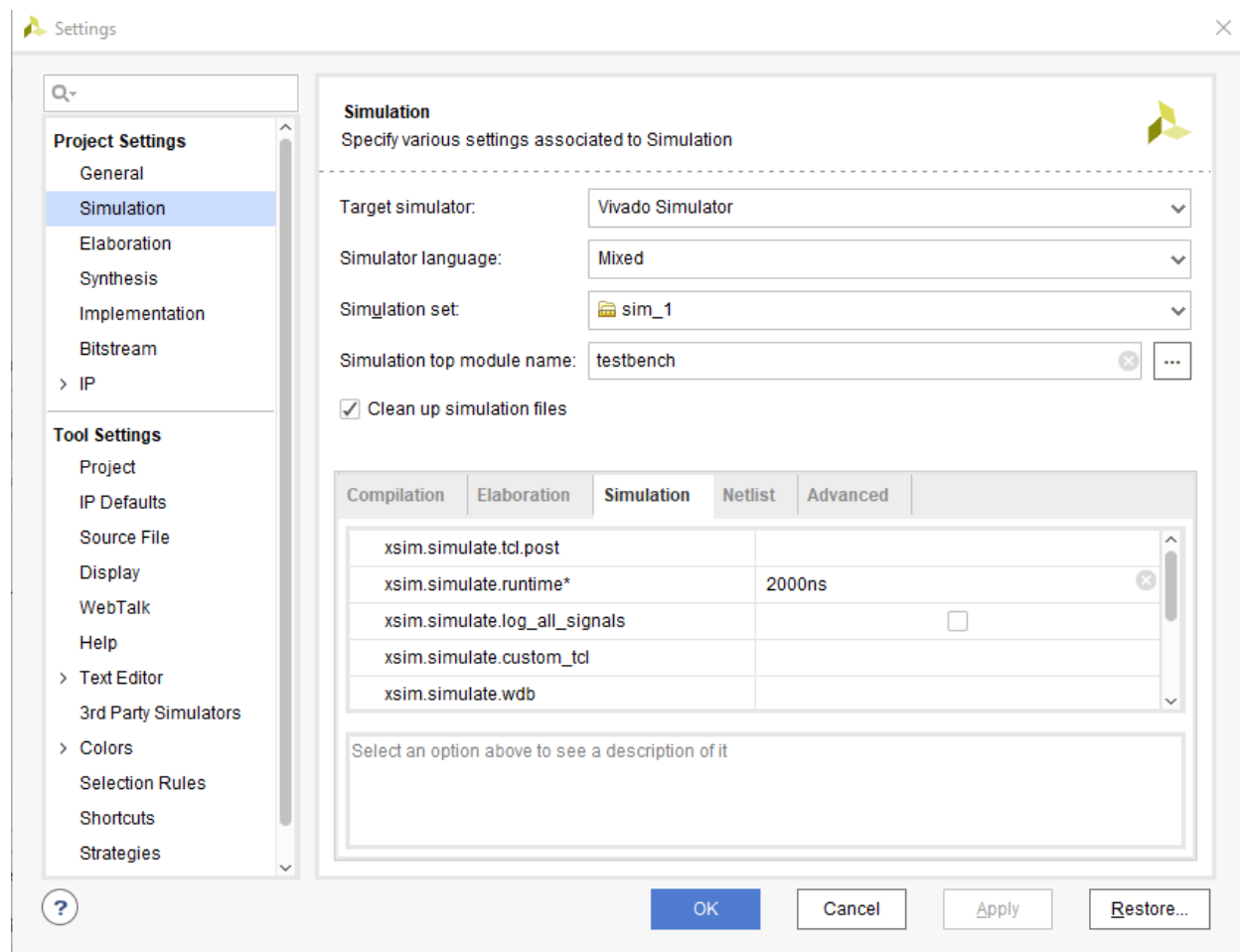


Figure 26. Change simulation run time

Click on **Run Simulation** → **Run behavioral simulation** in the Flow Navigator window, as shown in [Figure 27](#).

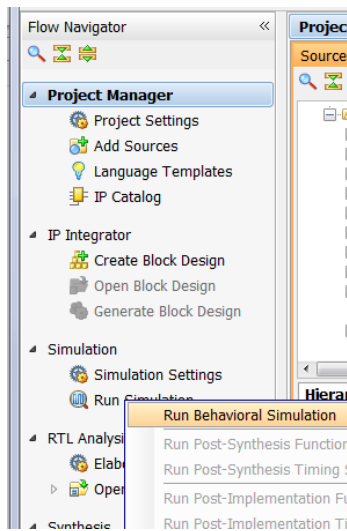



Figure 27. Run simulation

The testbench and lower-level modules will compile, the simulation window will open, and the simulation results will be displayed, as shown in [Figure 28](#). This could take several minutes.

The simulation waveform shows the top-level signals of the top-level module, in this case the testbench module. Click on the Zoom Fit button ().

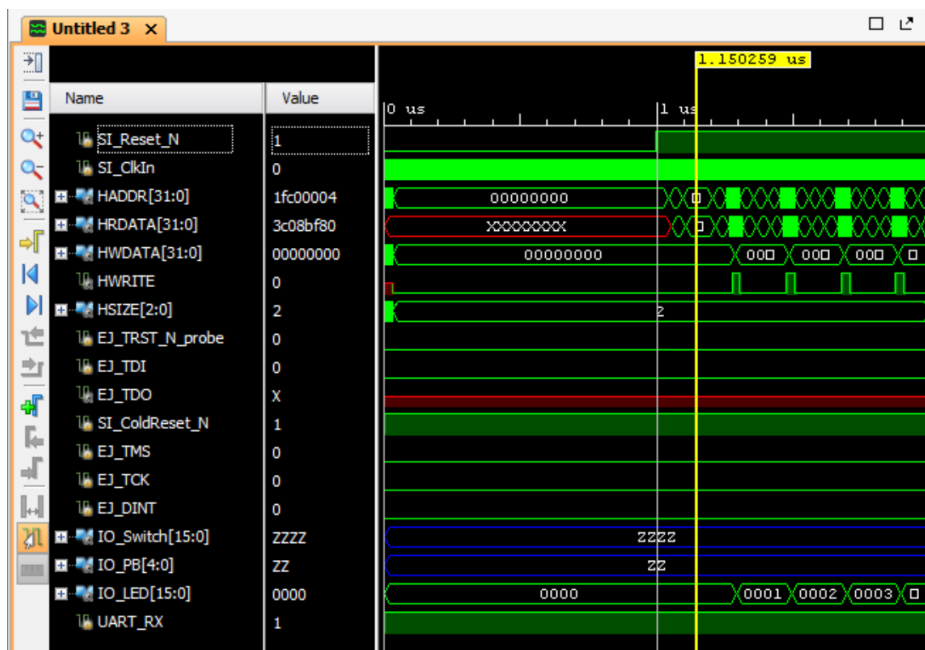


Figure 28. Simulation results showing top-level signals

The radix of the signals defaults to hexadecimal, but you could change the radix of any multi-bit signal by selecting it in the waveform window and then right-clicking and selecting **Radix** → **Binary**, or whichever representation you prefer. 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. More specifically, delete: EJ_TRST_N_probe, EJ_TDI, EJ_TDO, SI_ColdReset_N, EJ_TMS, EJ_TCK, EJ_DINT, IO_Switch, IO_PB, and UART_RX. Do **not** delete IO_LED. 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 29.

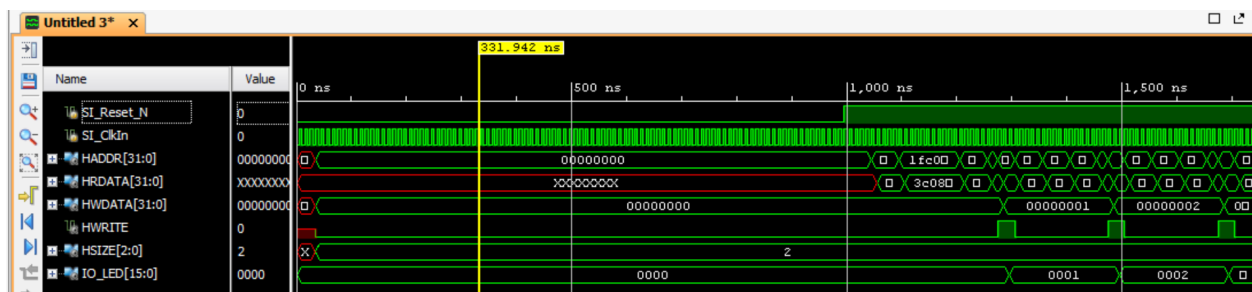
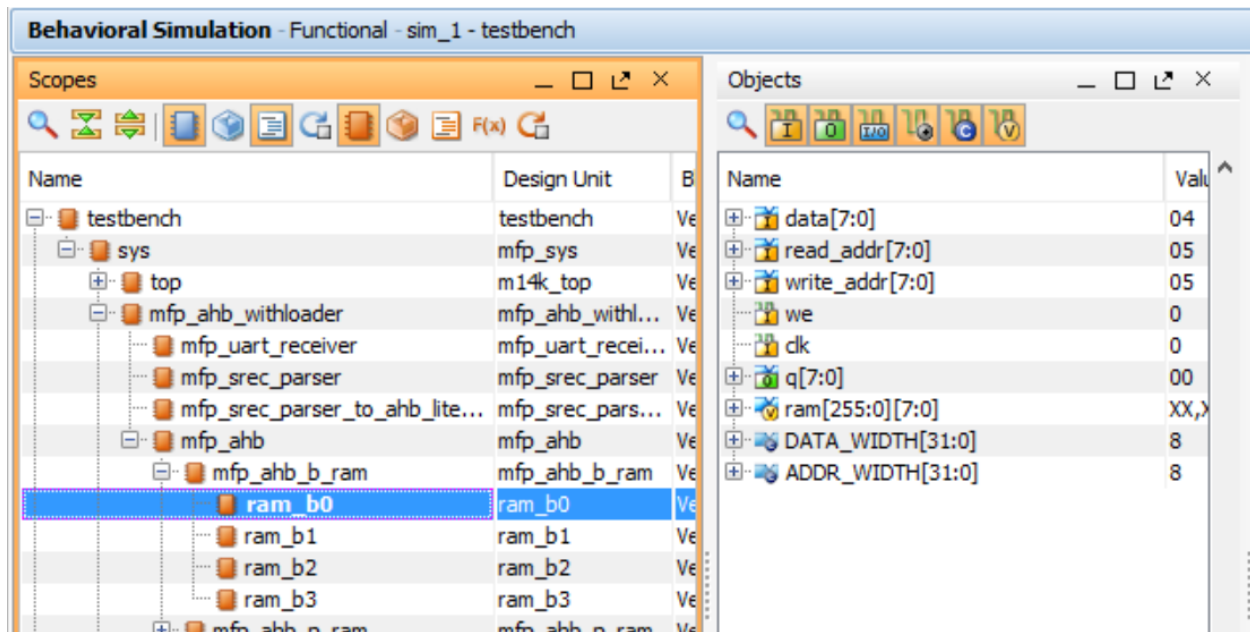


Figure 29. Keeping only the desired top-level signals

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 30, expand the **testbench** hierarchy to **testbench** → **sys** → **mfp_ahb_withloader** → **mfp_ahb** → **mfp_ahb_b_ram** → **ram_b0** to see the **ram_b0** module in the **Scopes** window. Click on the **ram_b0** module to see the corresponding signals in the **Objects** window. This module holds the least significant byte of the instructions in the boot RAM.



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 **Boot RAM0** in the field and press **Return**.

Select all of the objects in the **Objects** window (Ctrl-a), then right-click and select **Add to Wave Window** and observe that the signals are added to the Waveform window. 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 31](#).

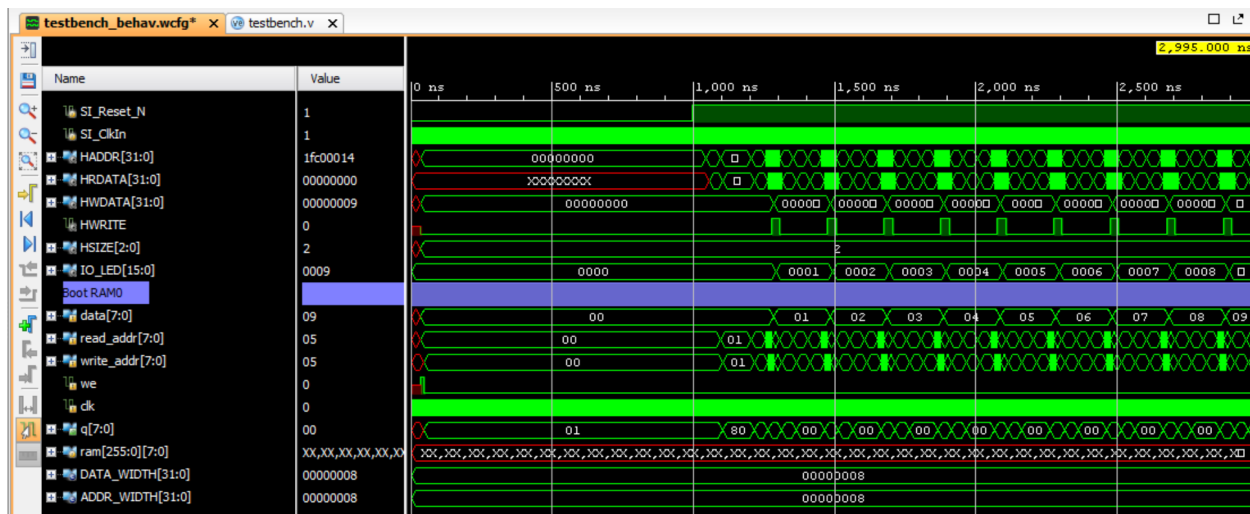



Figure 31. Simulation result showing lower-level module (ram_b0) signals

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 (physical) 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`. That because this is a simple program and contains no boot code, the caches are not initialized, so 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_LED[15:0]` displays the incremented value because it is memory-mapped to address `0xbf800000` (physical address `0x1f800000`). The `IO_LED` 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 4. Compiling MIPSfpga

Now you are ready to compile the MIPSfpga system and create a file 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. You may get a “No Implementation Results Available window”, just click **OK**. You may also then get a “Launch Runs” window, it should indicate the Default Launch Directory, click **OK**.

A 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* and has a *.bit* extension. Now wait for synthesis, placement, routing, and bitstream generation to complete. This typically takes about 10-20 minutes, depending on your computer speed. It should take about 15 minutes if you use the lab computers. Don't be alarmed by the high number warnings (almost 400 on my system) most of them are related to unconnected signals and the like. When you start making changes to existing code and creating your own code it will be imperative that you chase down the warnings, lest they cause strange and unpredictable results downstream in the process. After the bitstream has been generated, you will see the **Bitstream Generation Completed** pop-up window, as shown in [Figure 32](#).

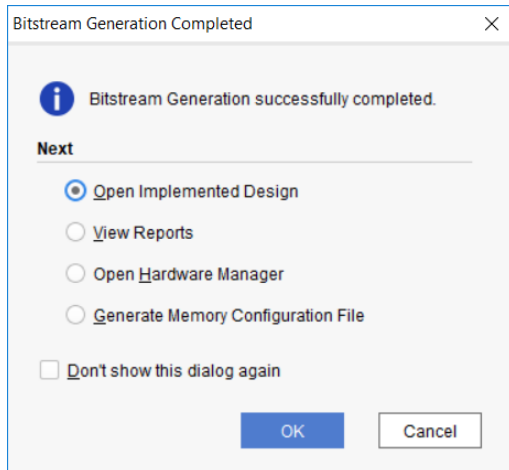


Figure 32. Bitstream Generation Completed pop-up window

Opening and viewing the implemented design is optional, but gives some insight into the timing and layout of the MIPSfpga system. (If you don't want to view it, select Open Hardware Manager and click OK. Then continue with Step 5 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 33.

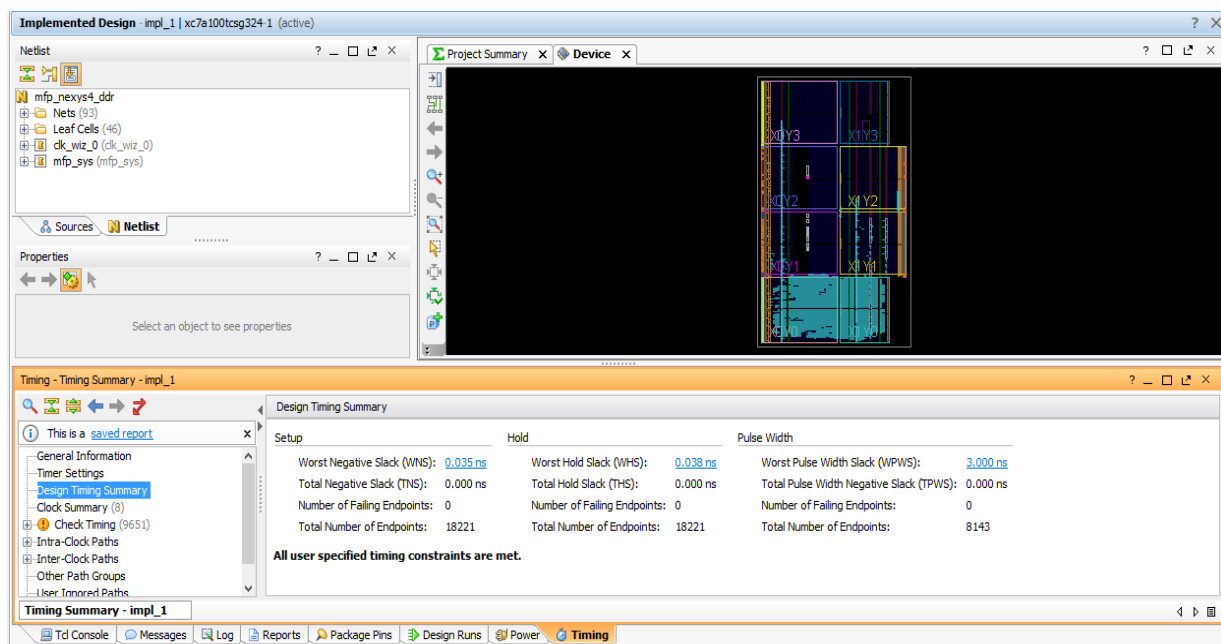


Figure 33. Implemented Design

Notice the **Design Timing Summary** pane at the bottom. Most importantly, it says that **All user specified timing constraints are met** (i.e. there were 0 Failing Endpoints) and the Worst Negative Slack (**WNS**) and the Worst Hold Slack (**WHS**) were all greater than 0ns.

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 34](#) 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 \approx 15 ns (frequency = 1/15 ns \approx 66 MHz).

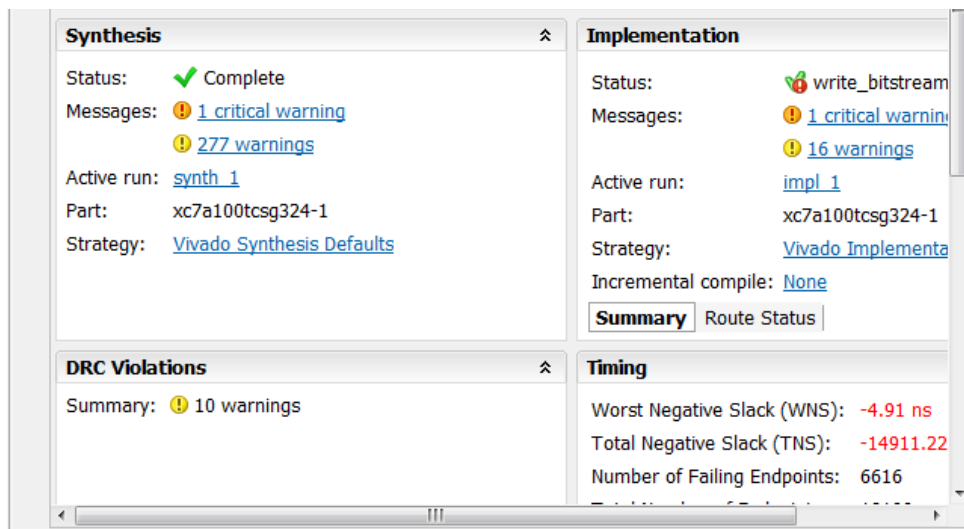


Figure 34. Design that does not meeting timing constraints

Step 5. Downloading MIPSfpga onto the 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 35 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 35. 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 display with a snake-like pattern that repeats indefinitely. To program the board, it can be in QSPI mode (as shown in [Figure 35](#) 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).

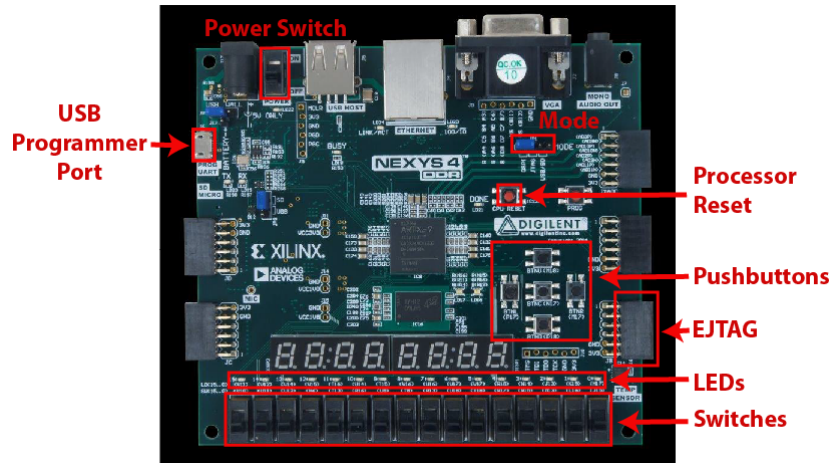


Figure 35. Nexys4 DDR board (photograph © Digilent Inc., 2015)

In the Vivado window, select **Flow** → **Open Hardware Manager**, as shown in Figure 36.

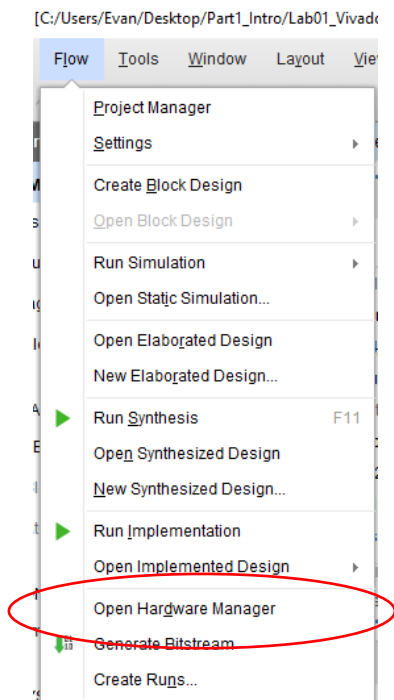


Figure 36. Open Hardware Manager

The Hardware Manager window will open. Now click on **Open Target** and choose **Auto Connect**, as shown in Figure 37.

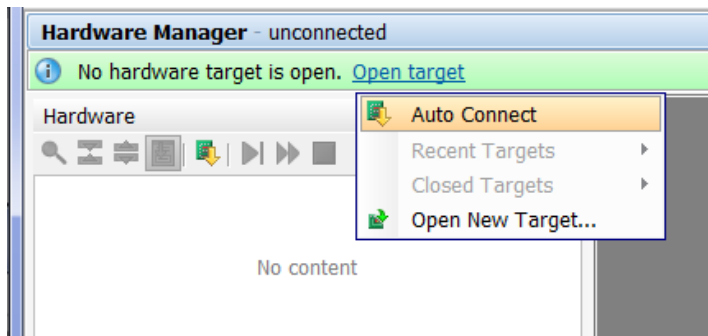


Figure 37. Hardware Manager window, Auto Connect

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.
...

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.

Now click on **Program device** and select **xc7a100t_0**, as shown in Figure 38.



Figure 38. Selecting Program device

The Program Device window will open, as shown in Figure 39. The Bitstream file box should auto populate, but if it does not, choose:

MIPSfpga_Labs\Labs\Xilinx\Part1_Intro\Lab01_Vivado\Project1\Project1.runs\impl_1\mfp_nexys4_ddr.b
it

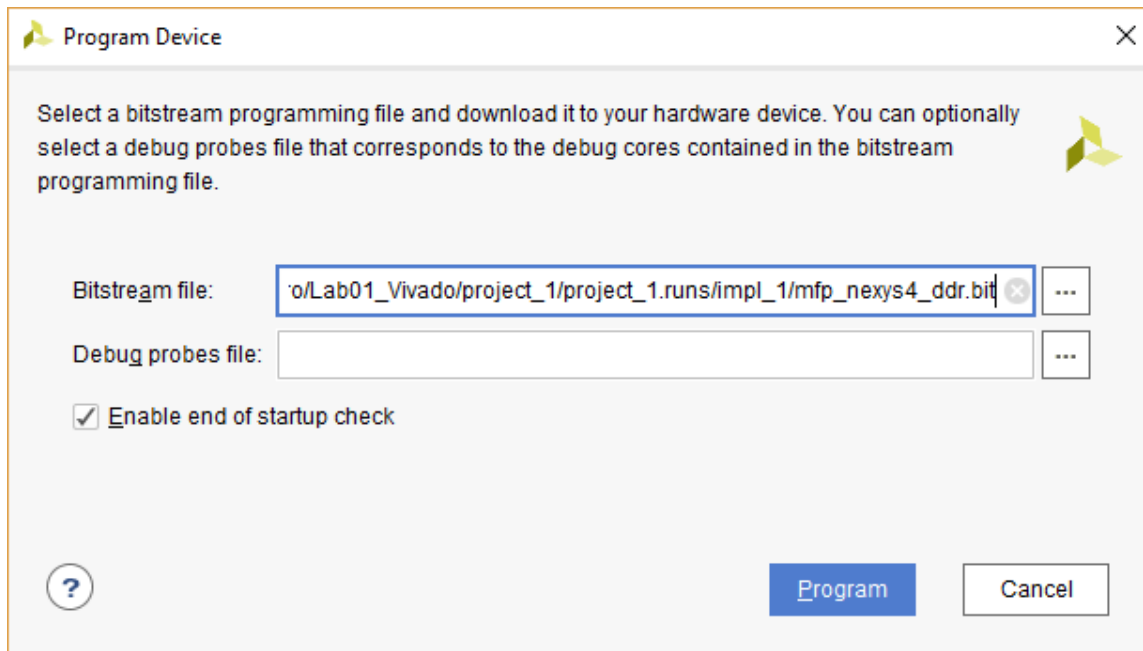


Figure 39. Program device with bitstream file

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

A window will pop up showing the programming progress, as shown in Figure 40. Programming the Artix-7 FPGA on the Nexys4 DDR board will take several seconds. Once it is complete, the progress window will close.

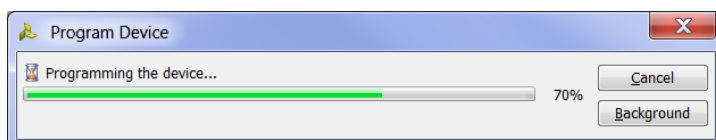


Figure 40. Program Device progress window

The MIPSfpga system is now downloaded onto the Nexys4 DDR board. Push the red reset pushbutton (labeled CPU RESET on the board, see [Figure 35](#)) 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_Labs\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 22](#)). The C, MIPS assembly, and machine code for IncrementLEDsDelay is shown in [Figure 41](#) and [Figure 42](#).

<pre>// C code unsigned int val = 1; volatile unsigned int* ledr_ptr; ledr_ptr = 0xbf800000; while (1) { *ledr_ptr = val; val = val + 1; // delay }</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 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 41. 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 42. Machine code for IncrementLEDsDelay (files actually used are ram_b0.txt, ram_b1.txt, ram_b2.txt, and ram_b3.txt – byte-wide memories)

<Finis >

