# Using Vivado and the DSDB FPGA

Instructor: Alex Underwood/James E. Stine, Jr.
TAs: Jacob Pease/Thomas Kidd

This document will explain how to start using Xilinx Vivado to get Lab 2 working on the National Instrument (NI) Digital System Development Board's (DSDB's) FPGA [1]. The steps listed here will also work for future labs with some small caveats related to connection of devices on the DSDB in which implementation on the FPGA will also be required.

## 1. DSDB FPGA : Zynq XC7Z020-1CLG484C

The DSDB has a plethora of devices on it including the main chip we will use this semester, the AMD (formerly Xilinx) Zynq FPGA. The AMD Zynq XC7Z020-1CLG484C is extremely powerful sporting 650 MHz dual-core Cortex-A9 processors, DDR3 memory controller with 8 DMA channels, high-bandwidth peripheral controllers: 1G Ethernet, SDIO, and many others. More information on the DSDB can be found here: `https://reference.digilentinc.com/dsdb/rm` and on Canvas.

All Zynq devices have the same basic architecture, and all of them contain, as the basis of the processing system, a dual-core ARM Cortex-A9 processor. This is a 'hard' processor (PS) and it exists as a dedicated and optimized silicon element on the device. The second principal part of the Zynq architecture is the programmable logic (PL). This is based on the Artix®-7 and Kintex®-7 FPGA fabric. The PL is predominantly composed of general purpose FPGA logic fabric, which is composed of slices and Configurable Logic Blocks (CLBs), as well as Input/Output Blocks (IOBs) for interfacing (i.e., these are all AMD-specific terms). As explained in class, the CLBs are small, regular groupings of logic elements that are laid out in a two-dimensional array on the PL, and connected to other similar resources via programmable interconnects. Each CLB is positioned next to a switch matrix and contains two logic slices. Although today's FPGAs contain more devices other than CLBS (e.g., Block RAM), we will primarily use the CLBs for our designs. These collection of devices on the FPGAs are typically called System on Chip or SoC devices.

## 2. FPGA Implementation

To begin working with the Zynq platform, you will need to use the Vivado Electronic Design Automation (EDA) toolset from AMD. Before attempting implementation, it is absolutely essential that your design functions correctly in Questa/ModelSim simulation. Proceeding without a fully validated simulation greatly increases the likelihood of failure.

Do not move forward with the implementation steps until your ModelSim simulations are fully passing to the best of your knowledge. Additionally, a strong grasp of verification using a SystemVerilog testbench is critical for ensuring a functional FPGA implementation. Debugging fundamental design flaws in Vivado and on the FPGA is significantly more challenging compared to detecting and resolving them within ModelSim simulations.

Vivado is a powerful EDA tool available in the ENDV 350 laboratories. While it can be installed on personal machines, doing so may not be necessary unless you wish to explore its capabilities further.

### 2.1 Vivado System Design

Modern system design emphasizes design reuse and rapid development, as reducing time to market is crucial in many applications. Tools that accelerate the development process—without compromising verification integrity or design quality—offer clear advantages.

The Vivado design flow is built around these principles, leveraging pre-verified Intellectual Property (IP) blocks for efficient system integration. Unlike traditional design methodologies that require building systems from the ground up, Vivado prioritizes IP reuse from its built-in libraries, third-party vendors, and user-defined components. This approach shifts the primary focus toward system integration rather than low-level hardware design, streamlining development without sacrificing flexibility.

That said, custom logic can still be integrated into the design, which is precisely what we will be doing with our Zynq FPGA. However, we will also leverage Vivado's IP catalog to ensure correct functionality and seamless integration with the DSDB (Digital System Development Board). This hybrid approach allows for both efficiency and customization, ensuring a robust and functional design.

## 3. Setting Up the Vivado Project for the First Time

There is a `lab2` project that contains a pre-made Vivado project in the repository. There is also a `setup.tcl` script in the `scripts` directory that we will use to quickly set up the project so you can see your RISC-V single cycle processor in action. The default program that the project is set to load into your processor is the `memfile.dat` program, but you need to change that later [1].

1. Download the repository from GitHub and your Vivado project should be inside the `lab2/lab2` subdirectory. Place the `setup.tcl` file from the `scripts` subdirectory of your lab2 repository on the desktop. Place the Lab 2 project folder wherever is convenient for you.

2. Place your `riscv_single.sv` file on the desktop that you wish to implement on the FPGA.

   **It is important that both the `setup.tcl` and `riscv_single.sv` files are on the desktop, as the `setup.tcl` script will look for files on the desktop.** That is, they exist in a different location (i.e., the Desktop) than your `lab2` Vivado project.

3. Open the Vivado project by first opening Vivado and then clicking Open Project under Quick Start (i.e., Figure 1). The Vivado project should be available in your lab2 repository directory. Navigate to the Lab 2 project folder you extracted from you Canvas zip file and select the Lab 2 project file inside that folder. You may need to click **yes** to allow the project to be updated your version of Vivado too. AMD's Vivado utilizes a lot of memory space, so it may take some time to open.

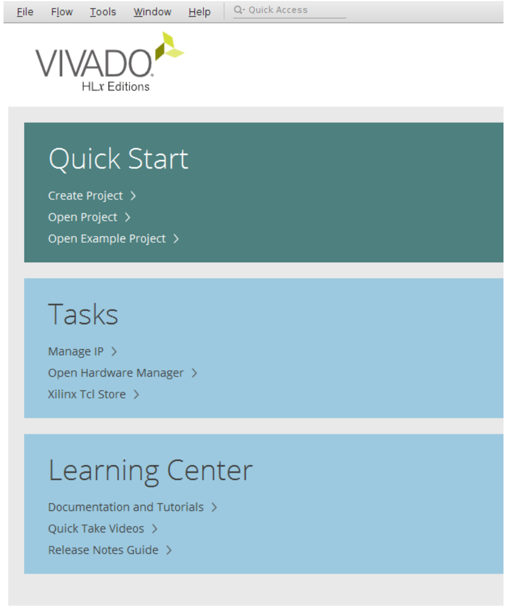
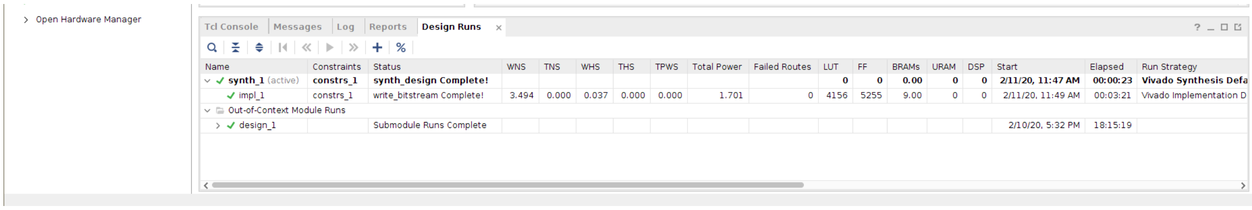


Figure 1: Quick Start and Open Project

4. Along the bottom of the screen, there is a collection of tabs for various status views. Select Tcl Console, which is where we will run our script (i.e., Figure 2).

[1] I am calling the output of our compiling process that generates the `.memfile` the `memfile.dat`. They are all ASCII files and can called whatever you want. Apologies in advance for any ambiguity.



| Name | Constraints | Status | WNS | TNS | WHS | THS | TPWS | Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP | Start | Elapsed | Run Strategy |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **✓ synth_1** (active) | **constrs_1** | **synth_design Complete!** | | | | | | | | **0** | **0** | **0.00** | **0** | **0** | **2/11/20, 11:47 AM** | **00:00:23** | **Vivado Synthesis Defa** |
| ✓ impl_1 | constrs_1 | write_bitstream Complete! | 3.494 | 0.000 | 0.037 | 0.000 | 0.000 | 1.701 | 0 | 4156 | 5255 | 9.00 | 0 | 0 | 2/11/20, 11:49 AM | 00:03:21 | Vivado Implementation D |
| Out-of-Context Module Runs | | | | | | | | | | | | | | | | | |
| ✓ design_1 | | Submodule Runs Complete | | | | | | | | | | | | | 2/10/20, 5:32 PM | 18:15:19 | |

Figure 2: Bottom Panel with Tcl Console, Messages, etc.

The Tcl Console behaves much like a terminal or command prompt in which you can run commands through keyboard entry instead of using the mouse and GUI. This allows us to script some of the more complex components involved in setting up a project and skip to the more important parts for this class including testing and profiling your design.

Just like a terminal, the Tcl Console runs inside a specific directory, but you can change that directory using the `cd` command. First, use the command `pwd` to Print your Working Directory (where you currently are). Using that information, Navigate to your desktop by typing `cd ..` to ascend a folder level and `cd <name of folder>` to descend into that folder. Your goal here is to navigate to the desktop where we put the `setup.tcl` and `riscv_single.sv` files.

5. Once on the desktop, run the following command in the Tcl Console:

   ```
   source setup.tcl
   ```

   `source` is the command used in Tcl to run a script stored in a file. Note: you only need to do this once. If you start over or find a bug, you will need to restart this process at Step 1. This script will do the following things:

   - Search your desktop for the `top_module.sv` file and copy that file into the project directory.
   - Refresh the project index, causing the modules that connect to the RISC-V core to see the newly added file and update their references.
   - Clean up the project to clear away any lingering errors or warnings regarding Block Generation, Synthesis, or Implementation.
   - Beginning running the project toolchain which will start with Block Generation in which each top level module in the project will be synthesized to check for usability. Then, Synthesis will convert the entire design and all surrounding components into a netlist that represents real hardware. Following that, Implementation will convert the hardware netlist into a design the FPGA can understand. Finally, Generate Bitstream will create a file that can be uploaded to the FPGA and allow the FPGA to start using your design.

   All of these steps are automatic when you run the script and will chain into each other so long as no errors occur during any steps. **Again, this is why you do not want to start using the FPGA until your simulation works - errors here take longer to find and are often more complicated to solve than if you had found them in your simulations in Questa/ModelSim.** These steps might take some time to run (approximately 5 minutes) - you can view their progress in the Design Runs tab on Figure 2

   If you run into errors or issues here, go to the Messages tab in Figure 2 where messages, warnings, and errors are stored and see what happened, or delve even deeper in the Log tab where the output from the toolchain is directly stored.

   If you need to make modifications to your files, you will need to do so in Vivado now because **the script will copy the file on your desktop into the project directory**. Modifications made to the file on your desktop will not be reflected in the project. Go to Open Block Design on the left panel and then select the Sources tab in the upper left window (i.e., Figure 3). You'll have to do a bit of digging through the source tree to get to your RISC-V core because this project has many components in the background to make things run, expand the design_1_wrapper, design_1, top_0, design_1_top_0_0, and finally inst : top to reveal your RISC-V core called `riscvsingle:riscvsingle`. Double-click that entry to open the file. Make your changes and save the file.

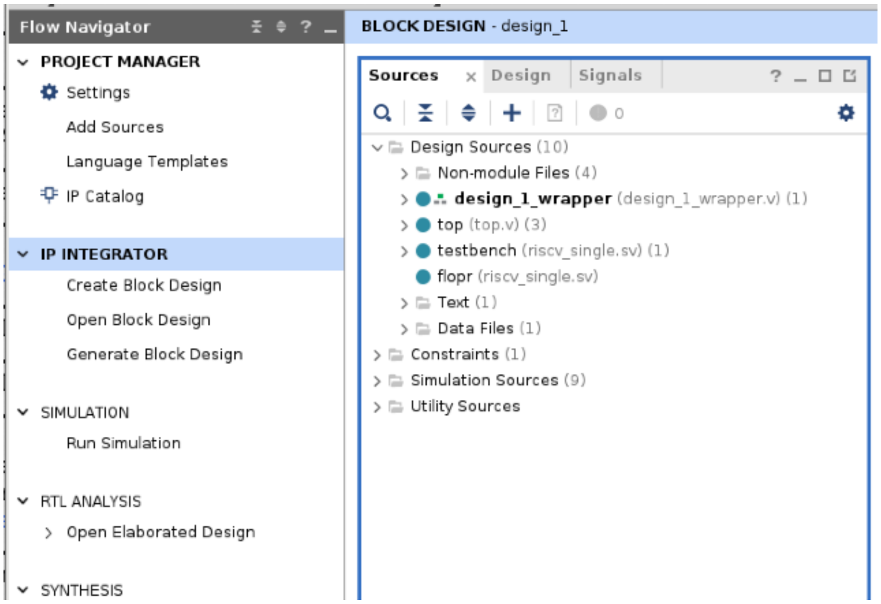


Figure 3: Open Block Design and Sources Window

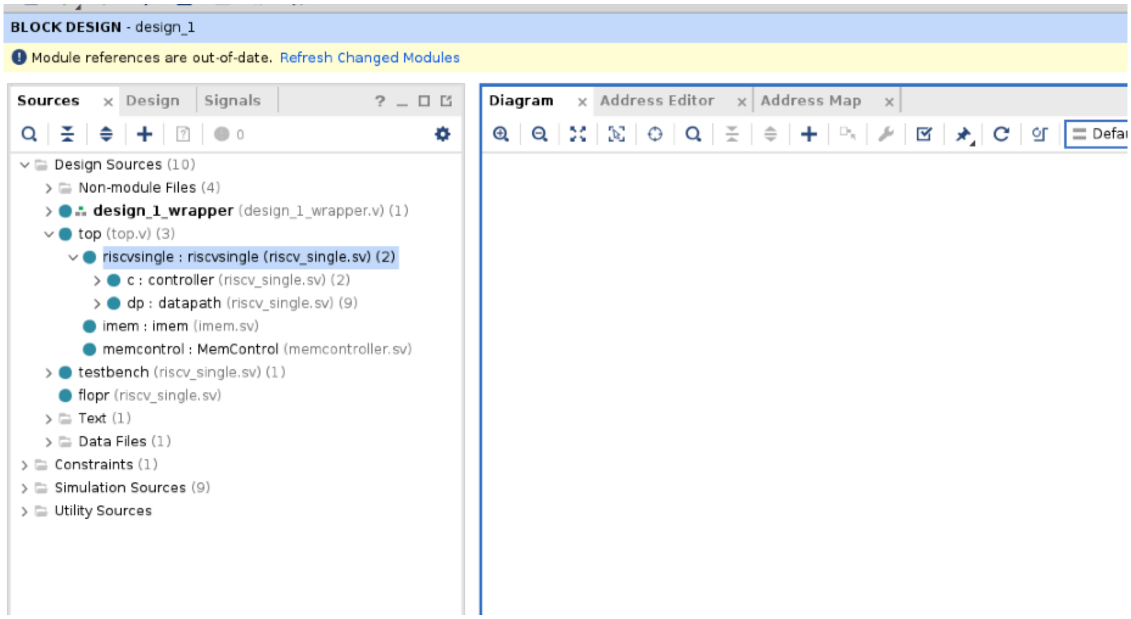
Anytime you make a change to the file and save it, a message will appear along the top of the screen that the Module References are Out-of-Date (i.e., Figure 4). Like what the script was doing, you've changed the module, but the other modules that connect to the RISC-V core haven't had a chance to refresh themselves and update their references. Clicking Refresh Changed Modules on that message will update the project index and refresh all references. You can then run the toolchain just like the script did by clicking Generate Bitstream on the bottom-left panel (i.e., Figure 5). A window will pop up letting you know that Synthesis is out-of-date and that will need to run first - answer Yes so all required steps will run first. Then, the Launch configuration window will open where you can set details about how the toolchain will run. Leave these at the defaults and click OK. Go to the Design Runs tab in Figure 2 to see each step's progress while waiting for them to complete (approximately 5 minutes).

6. Once Bitstream Generation is complete, a window will pop up asking what you would like to do next (i.e., Figure 5). Select Open Hardware Manager and click OK. If you accidentally close this window or select something else, you can easily get to the hardware manager using the Open Hardware Manager button below Generate Bitstream in Figure 5.

7. If you have not already done so, now is the time to connect your DSDB to the computer and power it on, as the next step will involve opening a connection to it in Vivado so you can upload your design to the FPGA.

   Remove the prototyping board from the Elvis III at your desk by pulling the board horizontally towards you. Connect the DSDB by lining up the gold fingers along the top of the board with the socket on the top of the Elvis III. Make sure the board is laying flat before pushing the board into the socket - the holes in the PCB along the bottom should slide into the plastic notches along the bottom of the Elvis III. Connect the DSDB to the computer at your station using the microUSB to USB cable included in it's box.

   *Yes, even though the DSDB is connected to the Elvis III, which is itself connected to the computer already, you still need to connect the DSDB to the computer with the USB cable in order to interact with the FPGA through Vivado.*


BLOCK DESIGN - design_1

Module references are out-of-date. Refresh Changed Modules

Sources
Design
Signals
Diagram
Address Editor
Address Map
Defau
Design Sources (10)
Non-module Files (4)
design_1_wrapper (design_1_wrapper.v) (1)
top (top.v) (3)
riscvsingle : riscvsingle (riscv_single.sv) (2)
c : controller (riscv_single.sv) (2)
dp : datapath (riscv_single.sv) (9)
imem : imem (imem.sv)
memcontrol : MemControl (memcontroller.sv)
testbench (riscv_single.sv) (1)
flopr (riscv_single.sv)
Text (1)
Data Files (1)
Constraints (1)
Simulation Sources (9)
Utility Sources

Figure 4: Refresh Changed Modules Banner

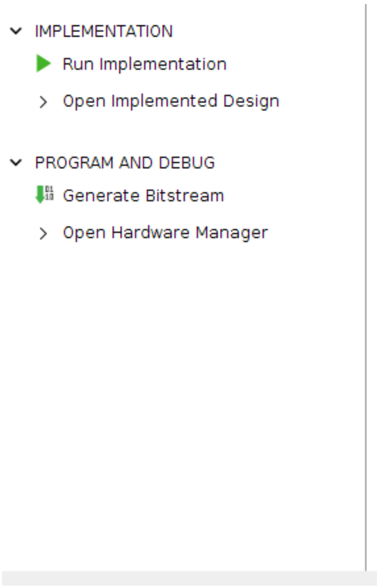
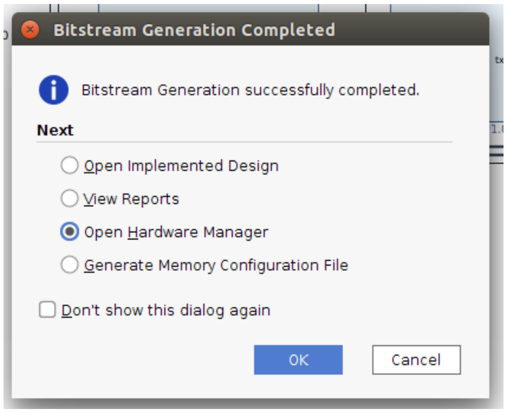


Figure 5: Generate Bitstream under Program and Debug

**The DSDB will be powered by the connector along the top of the board through the Elvis III - you do not need to power it with another power source. Do not connect a power supply to the barrel connector on the top right of the DSDB.**

Power on the Elvis III using the power switch along the top-back of the device. Then, power on the expansion board with the button on the upper left of the Elvis III. Finally, make sure the boot switch `SW8` on the DSDB is set to `QSPI`, and then power on the DSDB with the power switch on the upper right of the board. Now your DSDB is powered on and connected to your system. You can check to make sure you have the correct boot switch setting by flipping the switches `SW0-SW7` on the board, and their corresponding LEDs `LD0-LD7` should light up.



Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

○ Open Implemented Design

○ View Reports

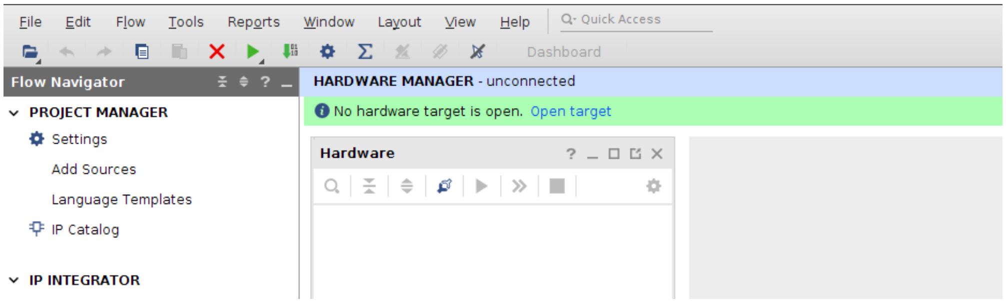
● Open Hardware Manager

○ Generate Memory Configuration File

□ Don't show this dialog again

Figure 6: Bitstream Complete, Open Hardware Manager

8. With the Hardware Manager open in Vivado and your DSDB connected to the Elvis III and the computer as well as powered on, click Open Target along the top banner that alerts you that no hardware target is open, and then click Auto Connect (i.e., Figure 7). This will automatically search the system for any powered-on devices that can interface with Vivado and connect to them for easy programming.

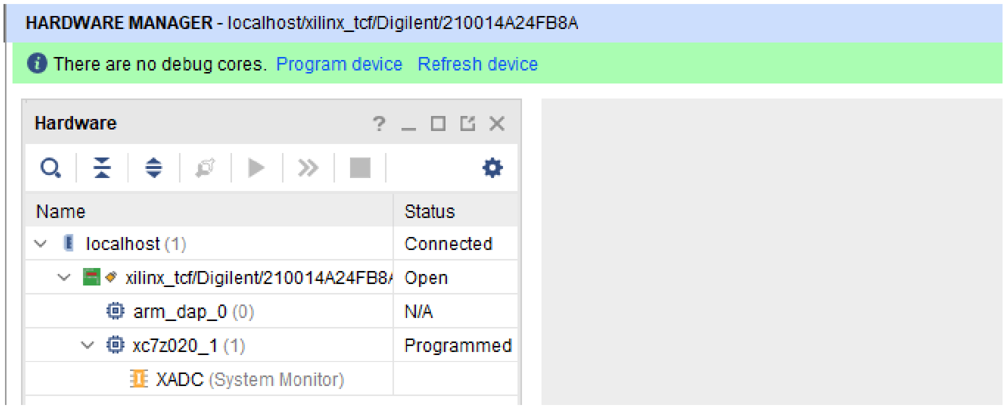


HARDWARE MANAGER - unconnected

No hardware target is open. Open target

Figure 7: Open Hardware Target

9. Once connected, the auto connect banner along the top will change to tell you there are no debug cores connected. This isn't an issue - our debug core is built-in to the design so it will connect once you program the device. Click Program device (i.e., Figure 8).



HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210014A24FB8A

There are no debug cores. Program device Refresh device

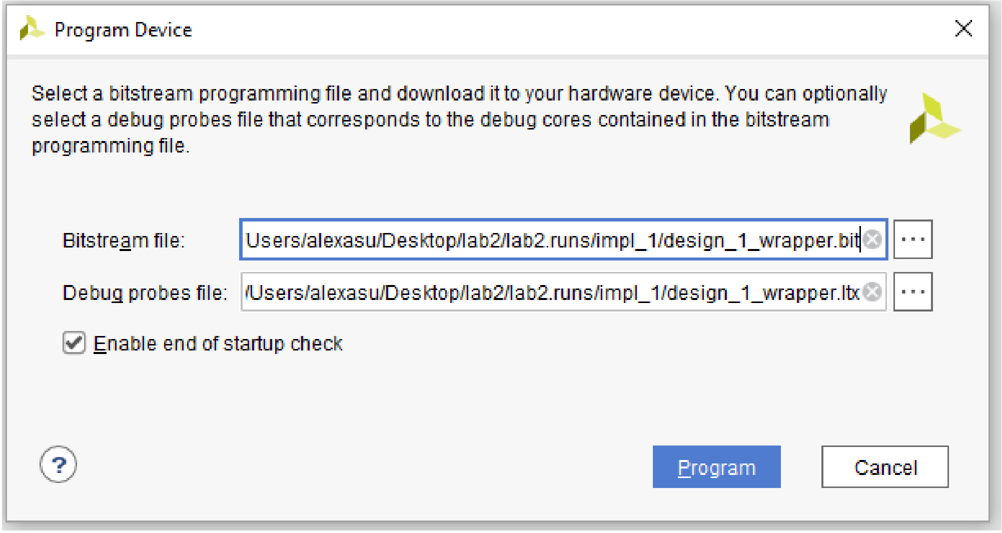
Hardware

| Name | Status |
|---|---|
| localhost (1) | Connected |
| xilinx_tcf/Digilent/210014A24FB8/ | Open |
| arm_dap_0 (0) | N/A |
| xc7z020_1 (1) | Programmed |
| XADC (System Monitor) | |

Figure 8: Program Device once Connected

**Starting at this stage and until specified later, you need to press and hold the bottom-right button `BTN0` on the DSDB. This button is bound to the reset signal in the processor and will keep the processor in a reset state once it has been programmed. We want to keep the processor in reset so it doesn't begin running until we've finished prepping the debug tool that will let us observe the system as it runs, so continue holding the button down while programming the device and prepping the debug window.**

After clicking Program device, a window will open in which you get to select the bitstream and debug profile to upload to the FPGA (i.e., Figure 9). These fields will be auto-populated, so (with `BTN0` held down), you can click program FPGA. The FPGA will load your design and the Hardware Manager window will change to look like a signal analyzer similar to ModelSim.



Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.

Bitstream file: Users/alexasu/Desktop/lab2/lab2.runs/impl_1/design_1_wrapper.bit

Debug probes file: /Users/alexasu/Desktop/lab2/lab2.runs/impl_1/design_1_wrapper.ltx

☑ Enable end of startup check

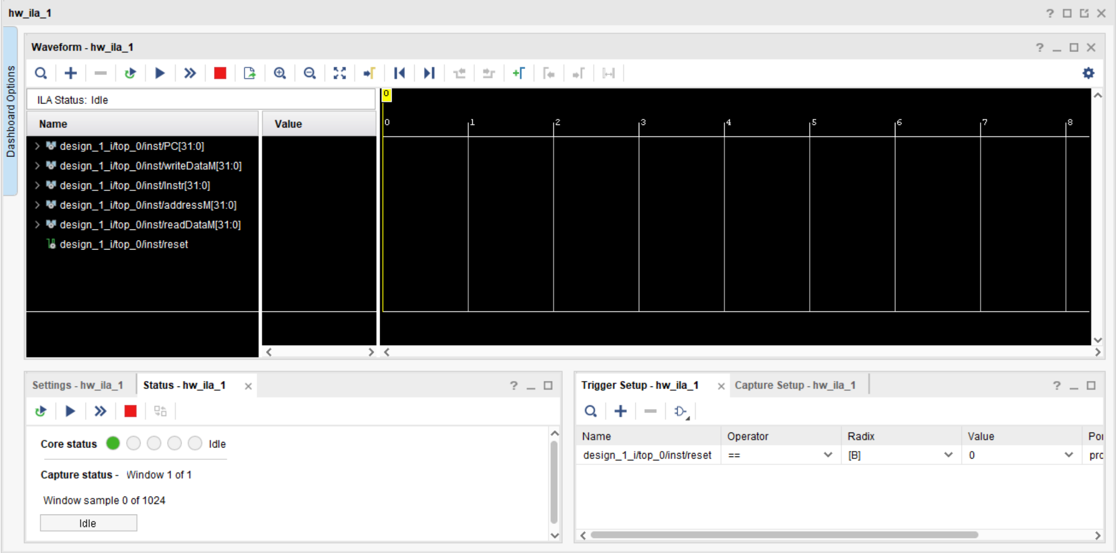
Figure 9: Select BitStream and Debug Image to Program

10. With the FPGA programmed and `BTN0` still held down, press the play button along the top panel of the signal analyzer view that has appeared in the Hardware Manager window to arm a debug trigger that has been pre-applied (i.e., Figure 10).

    Unlike simulations which can just generate the next value and save that on the computer, the FPGA is a real system with a real circuit on-board, meaning that capturing the data inside is a little trickier. To capture this data, a debug core has been pre-added to this project that observes some of the data that flows into and out of your RISC-V core. But, because these data measurements take place on the FPGA and then the data is sent back to the computer, only so much data can be saved at a time. This is where debug triggers come in - we can set the debug core to only save data when a certain event takes place such as a specific signal reaching a specified value or even a Boolean function of multiple signals. We have pre-made a trigger for this project to detect when the reset signal is logic low (which, because you are still holding down `BTN0`, is currently logic high). Pressing the play button arms the trigger so the debug core starts actively cycling through data, waiting to save the current data it has as soon as the trigger is true.

    Once the play button has been pressed and the trigger is armed, release `BTN0` which will complete the trigger requirements and cause the debug core to start taking data. This will simultaneously bring the RISC-V core out of the reset state and it will begin executing the program it was loaded with (the default is `memfile.dat`).

    The signal analyzer window will populate with data - 512 data points per signal before the reset signal went low and 512 data points per signal after the reset signal went low. You can use the values in the signal analyzer window to view view the performance of your system by matching the value of your



ILA Status: Idle

- design_1_i/top_0/inst/PC[31:0]
- design_1_i/top_0/inst/writeDataM[31:0]
- design_1_i/top_0/inst/Instr[31:0]
- design_1_i/top_0/inst/addressM[31:0]
- design_1_i/top_0/inst/readDataM[31:0]
- design_1_i/top_0/inst/reset

Core status ● ○ ○ ○ ○ Idle

Capture status - Window 1 of 1

Window sample 0 of 1024

Idle

| Name | Operator | Radix | Value | Poi |
|---|---|---|---|---|
| design_1_i/top_0/inst/reset | == | [B] | 0 | prc |

Figure 10: Signal Analyzer and Debug Trigger Menus

RISC-V core's FPGA with the instruction it should be executing at that point and see as values pass by on the memory and address busses.

Specifically, we are interested in the memory performance of the system, so we want to see how long the system is forced to wait for `LW` and `SW` operations to complete. `fib.s` has both of these operations, but you should test other programs later as well. Once you've found a memory operation (fairly easy to find, as they take the most time when looking at the signal analyzer view), calculate how many cycles they take by positioning the cursor (click and drag it) at the end of the operation to get the exact end time, then position the cursor at the start of the operation to get the exact start time. Subtract start time from end time and that will be your memory response time for this particular program on your RISC-V core.

Remember that load and store both take different amounts of time, so measure the response time of both. Of course, this is also a good opportunity to make sure your RISC-V core is still functional by seeing if the value read out of memory in the `memfile.dat` program matches the value that was written to memory just a couple of instructions prior. If that isn't working, then your measure of the memory's response time may not be accurate because the rest of the processor isn't working as intended.

If you re-run the trigger and press reset after you have already captured data once, the debug core will still capture data and your RISC-V core will still be functioning, but because the system repeats the same program over and over (the instruction memory in the FPGA is only as big as it needs to be to hold the program you specified to load in) the positions used in memory for the program will already hold the correct values and the values read from memory might still be on the memory read port due to the small number of `LW` instructions preventing the value on that port from being overwritten.

You can rerun the program from scratch by powering off the DSDB and then powering it back on. Vivado might throw an error because the board was disconnected without warning, but simply follow the steps to auto-connect to the board and proceed from there to run a completely fresh run again.

By now, you should have your Vivado project set up so that it contains your modified and functional `riscv_single.sv` file, the project has run through the toolchain including Block Generation, Synthesis, Implementation, and finally Bitstream Generation, and you've been able to upload the bitstream to the FPGA and see your processor function correctly on the FPGA running the `memfile.dat` program. With this, you've been able to take the data on how long each `LW` and `SW` operation should take when using real memory like that of the DSDB.

## 4. Changing the Program the RISC-V Core will Run

By default, the project from Canvas will load the `memfile.dat` program. Once you've successfully run that program on your own RISC-V core on the FPGA, you should try running some other programs and see what results you get and how they compare. The project comes with `memfile.dat` and `test_hw.s` pre-added, making it fairly easy to swap between the two programs. If you want to run your own programs, you can do that as well but the section after this one will cover how to add those files to the project so Vivado knows where to look for them.

1. Navigate to the sources view as explained in the previous section under step 5 and shown in Figure 3. You'll want to descend the source tree just like it was explained before, but this time open `imem.sv` instead of `riscv_single.sv`. Inside this file, you will find the following line:

   ```
   initial $readmemh("memfile.dat", RAM);
   ```

   This is the command that loads the program into instruction memory. Changing `"memfile.dat"` to your respective `"xxx.memfile"` will load in any program into the FPGA. Currently, the `"memfile.dat"` should be the Fibonacci program in the riscvtest directory.

2. Save the changes to the file and follow the steps from the first section of this guide again starting halfway through step 5. These steps boil down to the following things:

   - Refresh Changed Modules using the button on the top banner (i.e., Figure 4).
   - Generate Bitstream, which will also run all necessary steps prior to that in the process and Open Hardware Manager once complete.
   - Make sure your DSDB is connected to the computer, set to `QSPI` boot, and powered on.
   - Auto Connect to hardware target, hold down `BTN0` and program the FPGA.
   - Arm the trigger in the signal analyzer with the play button, then release `BTN0` to begin running the program.
   - Take data on the results of the program execution on your hardware.

By now, you should have been able to go through the toolchain in order to upload your design to the FPGA and see it in action using the debug information.

## 5. Adding your own Programs into Vivado

You can run your own programs on your RISC-V core, as well. To do this, you will need to import the `.dat` file for the program into your Vivado project, then follow the steps in the previous section on Changing the Program the RISC-V Core will run to change which program will be loaded into the instruction memory. Right now, the instruction memory is hard-coded into the FPGA within the Block RAM.

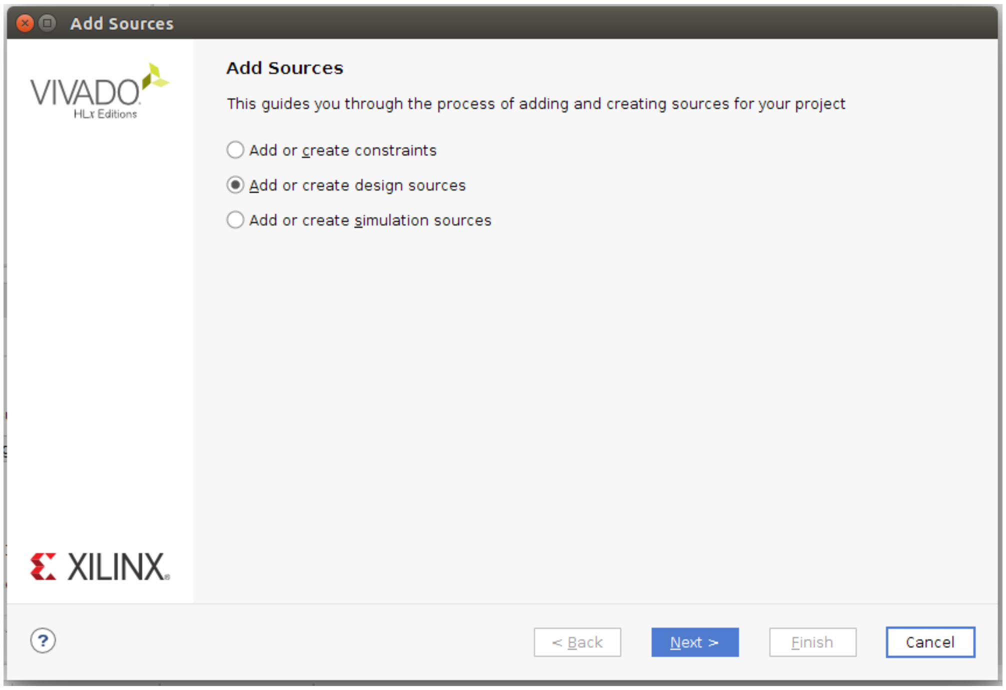
Make sure you test your hardware with `test_hw.s` in the `risvtest` subdirectory of the repository. You should be able to compare the state of the hardware compared to what you get with `spike`. To test with spike, you can type: `spike --isa=rv32i -d test_hw.elf`. To get full credit for this lab, the state of your hardware should match.

1. Click Add Sources under the Project Manager heading on the left panel (i.e., Figure 3).

2. Select Add or create design sources on the window that opens and click Next (i.e., Figure 11).

3. Select Add Files to open a file browser you can use to navigate to your `.dat` file. By default, the file browser will be looking for HDL source files, so you'll need to change file type visibility by clicking the Files of Type drop down menu and selecting All Files. Select your `.dat` file and click OK. You can select more than one `.dat` file if you'd like to add multiple files at the same time.

4. The files you selected will be listed in the Add or Create Design Sources window. Make sure the Copy sources into project box is checked and then click Finish (i.e., Figure 12).



**Add Sources**

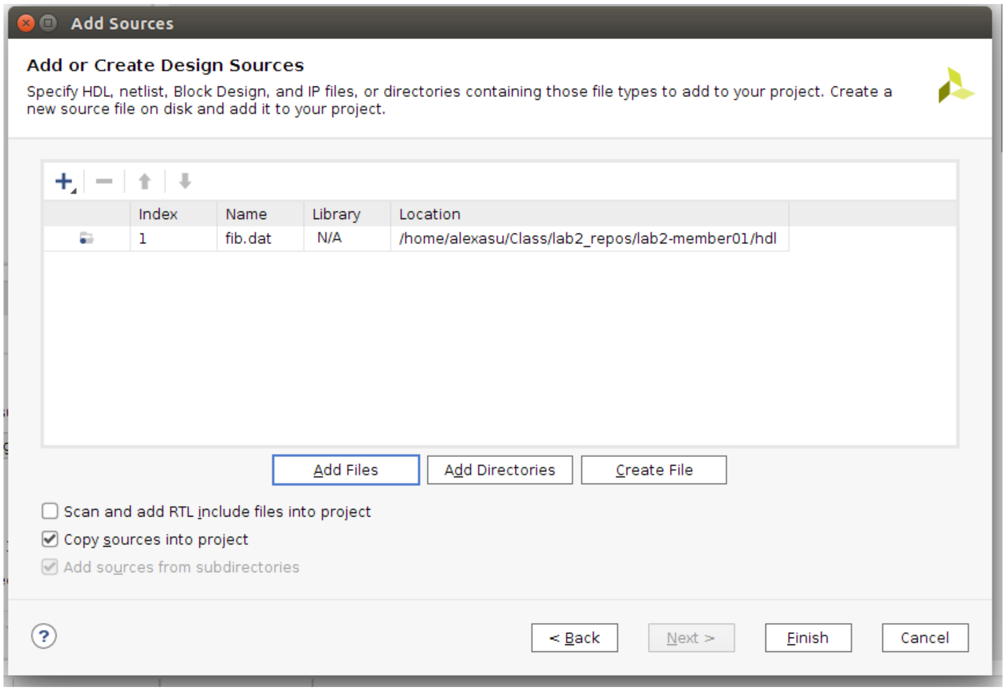
This guides you through the process of adding and creating sources for your project

○ Add or create constraints

● Add or create design sources

○ Add or create simulation sources

Figure 11: Open Hardware Target



**Add or Create Design Sources**

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

| | Index | Name | Library | Location |
|---|---|---|---|---|
| | 1 | fib.dat | N/A | /home/alexasu/Class/lab2_repos/lab2-member01/hdl |

□ Scan and add RTL include files into project

☑ Copy sources into project

☑ Add sources from subdirectories

Figure 12: Open Hardware Target

5. With the new `.dat` files added to your project, follow the steps in the previous section to change which `.dat` file will get loaded into the instruction memory and then continue through the steps to run the toolchain, upload the bitstream to the FPGA, and test your design.

# References

[1] "User Manual: NI Digital System Development Board," National Instruments, Tech. Rep. Pub ID: 376627B-01, 2018.