element14.com

AVNET MiniZed Dev Board - Review

25-32 minutes

INTRODUCTION

This is an incredible development package for a mere $89. This kit has everything you need to fully explore both embedded linux development and FPGA logic development. The kit comes with a 1 year license for Vivado System Suite. With it’s built in JTAG debugger it can instantly be used for both FPGA hardware and software development. The MiniZed development board comes preprogrammed with PetaLinux (a branch of Yochto) so you can instantly use it in an

embedded Linux application. The MiniZed development board also comes equipped with WIFI and Bluetooth 4.1. There is a micro-USB port that is used for both a 115.2 Kbps serial link and JTAG. There is also a USB-OTG connector that you can instantly use for an externally Linux mounted drive. The two PMOD connectors (2x6x .1in spacing) provide interface capabilities that has become popular for adding peripherals such as radios. The Arduino compatible interface also allows one to use the large supply of off the shelf Arduino Shields that you may already have in your collection *(NOTE: You will want to make sure that you select Arduino Shields that do not expose the FPGA pins to 5VDC only 3.3VDC maximum).*

UNBOXING

The AVNET® MiniZed(TM) Design Kit

provides the following items (see photo below):

- MiniZed Development Board
- Voucher for SDSoC License (from Xilinx)
- Micro USB Cable
- Quick Start Instructions Card
- Safety Instructions Pamphlet

When the box is first opened there is a small instruction sheet that brings you to the AVNET site and there is a large selection of documentation for the MiniZed

including the Getting Started Guide that is referenced in the instruction sheet.

PLUGGING INTO HOST THROUGH USB CONNECTOR

The host machine used (i.e. Desktop PC) is a **Linux Ubuntu-Studio 16.04** (six processor AMD).

The terminal emulator program used is “putty” with the following parameters:

- speed=115200,
- data bits=8,
- stop_bits=1,

(which has been the standard UART communication for most development boards).

The supplied micro-USB cable was connected to the USB/UART/JTAG connector and the board powered up;

Then the command “dmseg” was typed on the host machine, and the response is as shown in the screenshot below:

This indicated that the drivers are probably working correctly for the FT2232H, which is a common USB to serial adapter (UART).

This gives the mapping to the serial line that will be entered into “putty”

- Serial line= /dev/ttyUSB1” *(Note: the /dev/ directory is pre-pended)*

  The JTAG appears to be registered correctly and will be tested later in Vivado, the IDE that is supplied by Xilinx (and the MiniZed board kit comes with a license file for registering your

tools).

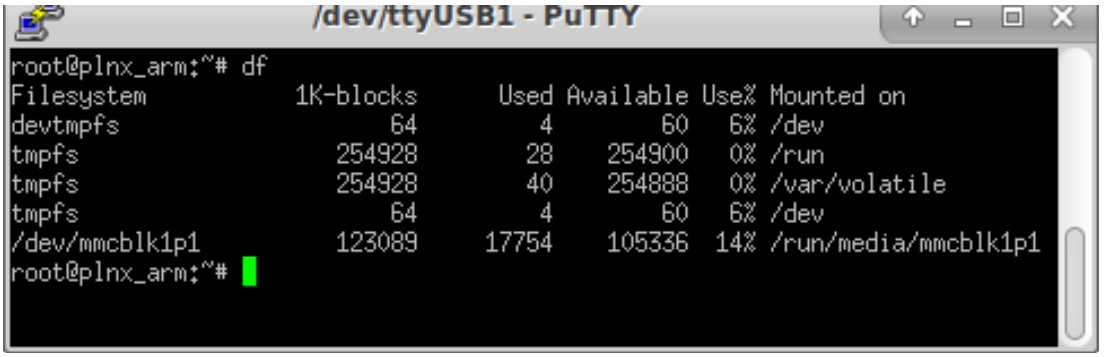
"putty", an open source terminal emulator program, was fired up with the configuration described above: serial line=/dev/ttyUSB1.

Note, the serial line may come up differently on your host so with the Linux host you will want to use dmesg when you plug in the board through USB.

A guess for the login was userid=root and passwd=root and it worked.

The command “df” was used to see how much memory was available and it looks like there is 123M available for user use.

Note: The data sheet indicates that there is more memory than appears here.

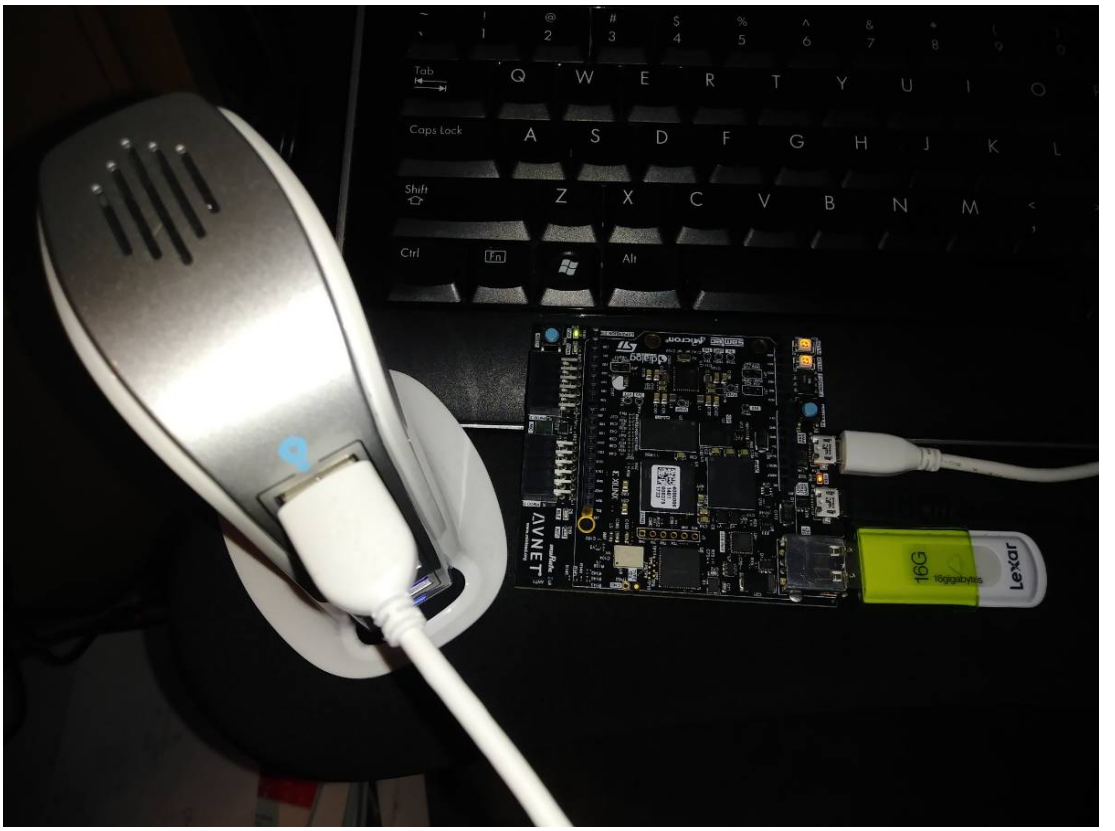


EXTERNAL USB MEMORY:

External USB memory did not function until external power was attached using an additional cable and USB power station as shown, the white USB micro cable is connected to an external 5V USB powering station. The USB memory stick is shown in the lower RHS.

It wasn’t clear from the steps in the documentation if there was a need to change a switch setting. After looking through the manual decided to take the chance! That was all that was necessary to get things to work per the quickstart guide. However, it did not seem necessary to format the USB drive to FAT32, a USB drive that was formatted to ext4 worked

fine. That makes sense because it is using petalinux.



WIFI:

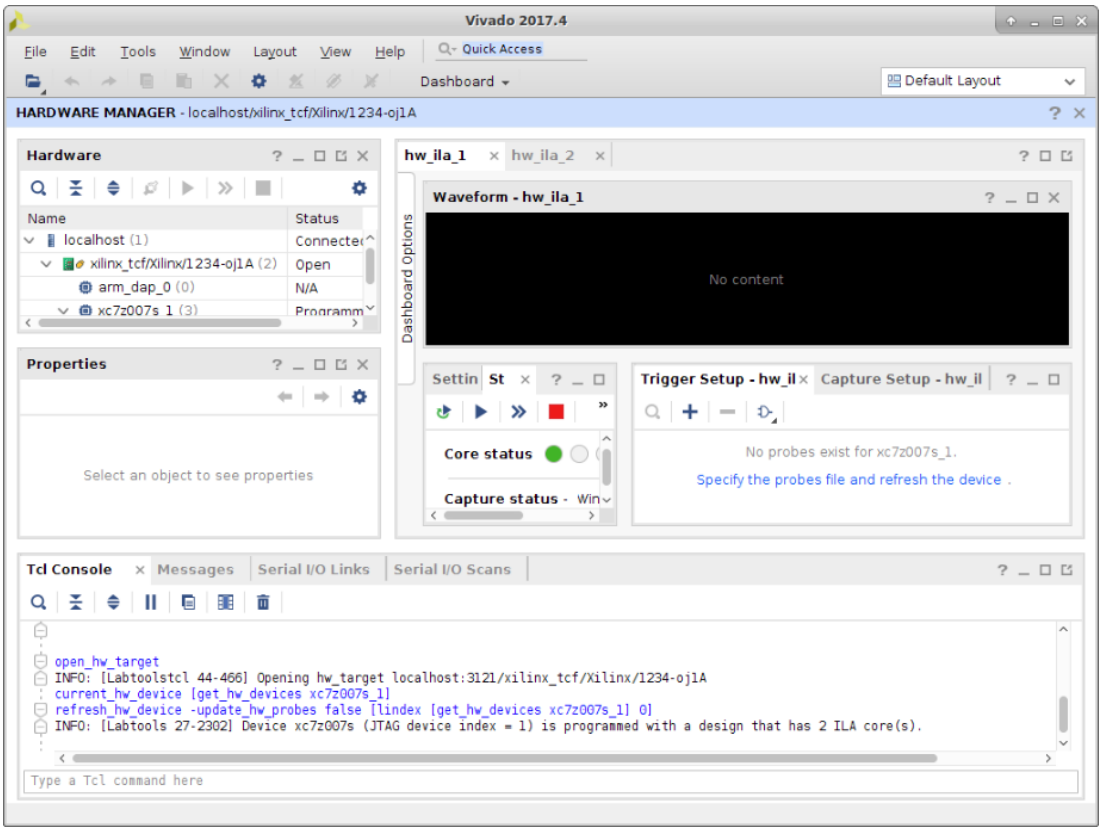
Setting up the WIFI did not appear to work per the instructions, causing me to do quite a bit of research into setting up a WPA2 connection using the configuration parameters in the wpa_supplicant.conf file that was given in the instruction but it looks like a reboot was all that

was needed. The command “reboot” was used to restart the arm processor.

TESTING JTAG USING VIVADO:

Vivado was already installed on the linux host machine, so it was opened, the hardware tab was selected and the auto-connect was used and it was able to immediately find the MiniZed board.

For those familiar with the FPGA tools of yesterday, getting to this point took some doing so it is amazing that out of the box, the FPGA interface and hardware is able to be found.



VIVADO SETUP:

At this point in the Road-test it is not clear how the processor and logic elements fit together, it was thought that an online course (or several) would be used as a starting point for **Road Testing.** An online course called "**Embedded System Design with Xilinx ZYNQ FPGA and Vivado**" seemed like a pretty good fit. The course was intended for the ZedBoard so it is likely it will not map directly into this board, so

that means there will be more learning to figure why things don't work per course. I bought several different FPGA training courses as they went on sale in anticipation of building up my FPGA and FPGA tool skills and over time will follow through with the courses. *NOTE: There are routinely sales on the courses and I paid $10 for the course, lately the courses have gone on sale for $11.99.*

Course Link: https://www.udemy.com/embedded-system-design-with-xilinx-zynq-fpga-and-vivado/learn/v4/overview.

The "**Embedded System Design with Xilinx ZYNQ FPGA and Vivado"** course discussed loading the Vivado tools, and installing the license. It would be helpful if this is the first time of installing Vivado, but it is pretty straightforward to install Vivado. However, because the **Vivado installations are so large**, and updating, between versions

requires even more space, it might be prudent to install Vivado on a hard drive and not a solid state drive (SSD). There may be several good reasons why you would want to have different versions of Vivado -- if you release a project using a previous version perhaps is one.

A word to the wise, based on my experience, is before starting this Road-test decide where you are going to put your files long term. This is particularly important if you are using a Linux installation. On my machine by the time I started this Road-test there was 7 different versions of Vivado installed (there is a new version released each quarter).

A housecleaning for me was in order for two reasons, first, I no longer used old Vivado versions and second between versions I changed locations of where I installed Vivado. **Changing directory location between versions caused me a lot of problems with**

the tools in particular with the *Document Navigator*.

My current solution for Linux required a soft link (ln -s) to the /opt directory on my SSD to a directory on my scratch hard drive (a drive located on my local host machine) . The /opt directory is the default directory where Vivado installation program wants to install to. Many other tools want to be installed to the /opt directory so link the entire /opt directory to an area in a scratch drive is probably a good idea unless you can afford terabytes of SSD space. There are also a couple of other **work directory** locations to sort out for your work. You will want to specify a path to a directory that gets backed up and **check-in** your work, to a revision control system, so if you break something you can **revert** back to a working copy. There is not a good reason to backup the tools as they can be reinstalled, and you

will be upgrading tools quarterly anyhow.

On my Linux host machine I use a hard drive labeled "**scratch**". The path to the directory used for the Xilinx files on my host is: /media /$USER/scratch/work/xilinx.

I use the path /media/$USER/scratch /workspace/xilinx for the "SDK" work. Vivado gives you the option of leaving the files in the Xilinx project file but at some point you will want to keep the software development separate from the hardware directories. *NOTE: If you have used eclipse (a software development IDE) for other uses, you may recall that ".....workspace" is the standard path that is used for work.*

On my system I'm using SVN as a version control system, so I can revert back to different versions that I check-in. It can be checked in to a local or remote SVN server. Having a local hard drive, on your host machine that is used

for code development and intermediate files really speeds things up compared to pulling your work in from an external drive, but using an external drive for backup is prudent.

Board Definition File:

One of the first things you will want to do is to install the **board definition files** (bdf) for the MiniZed into Vivado. At the time of the writing the board files for the MiniZed were not built into the Vivado distribution (2017Q4) thus I manually added.

The MiniZed board definition files that are used with Vivado are located here: https://github.com/Avnet/bdf

One should refer to “Install Avnet Board Definition Files in Vivado 2017.1” Document. And you can also get the yochto files (linux distro) at https://github.com/Avnet/petalinux. The instructions for where to install the boards

is given in the readme file in the github link. For my installation it is /opt/Xilinx/Vivado /2017.4/data/boards/

The first project in the **Embedded System Design with Xilinx ZYNQ FPGA and Vivado** course didn't exactly map to the resources that are on the MiniZed and my goal was to get through the steps of the FPGA process so I decided to create a very simple project that exercised a couple of available resources on the MiniZed board.

Here are my list of steps for getting through the FPGA tools:

1. Coming up with requirements.
2. IO mapping pins of the Zynq 7Z007S SoC to external hardware (includes on board hardware).
3. Writing HDL code for an application (verilog was chosen)

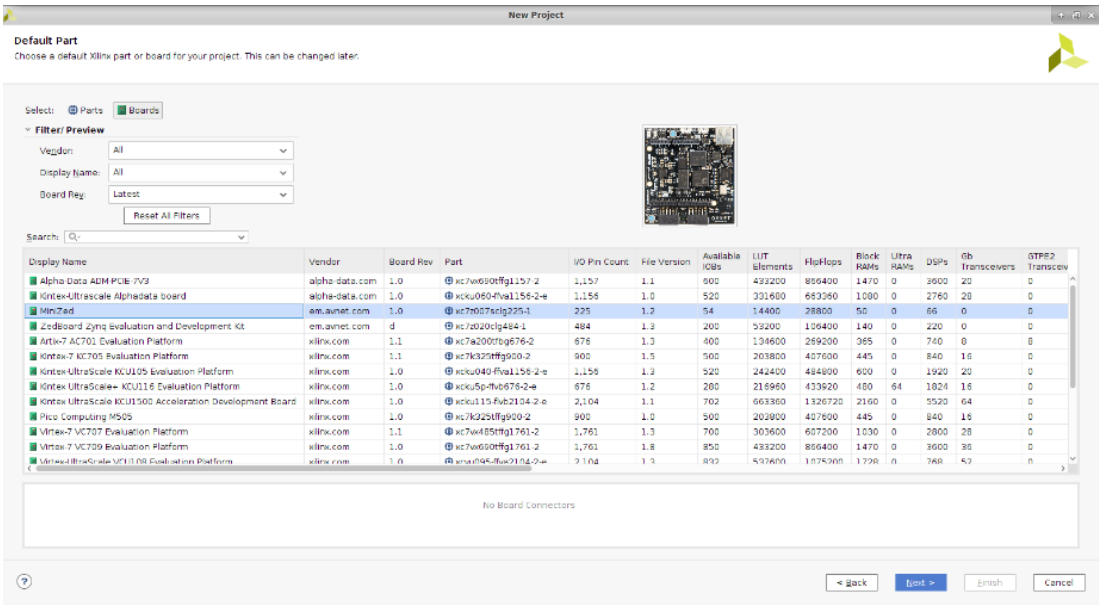
4. Clean compile of the HDL code (synthesis step)
5. Implementation (no timing constraints used for this simple example)
6. Writing a test bench (skipped in this exercise)
7. Building a bitstream
8. Loading the hardware
9. Testing the hardware
10. Restoring the stock image

1. **Requirements**

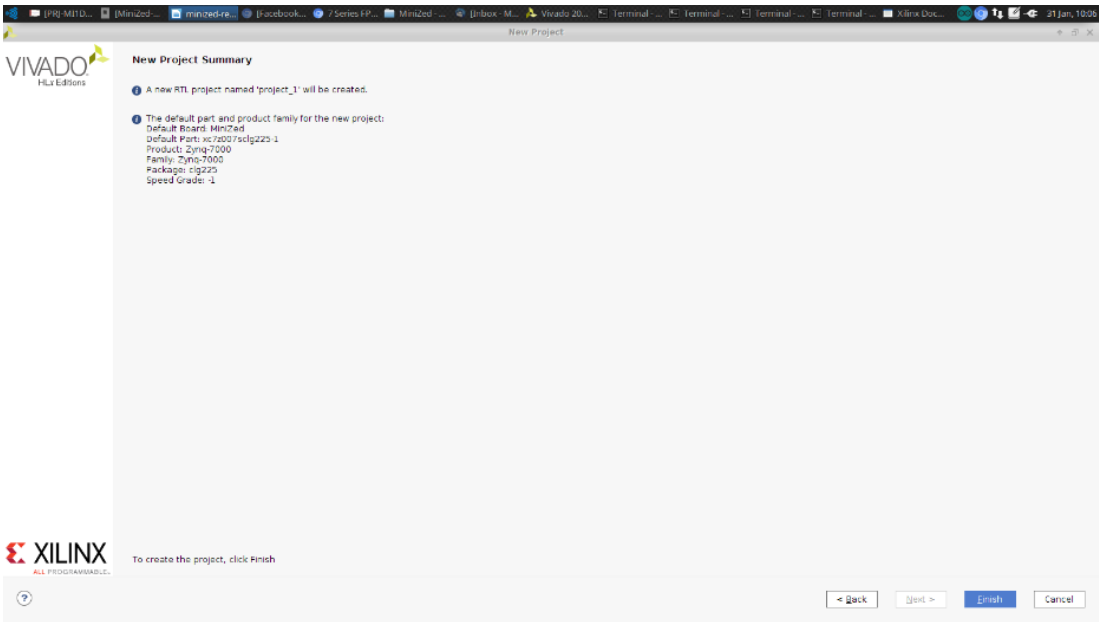
1. Use all onboard hardware
2. When the user switch is slid into position A light the Green LED and extinguish the Red LED.
3. When the user switch is slid into position B light the Red LED and extinguish the Green LED.
4. use verilog

2. **IO mapping**

1. The first time through this process, the board definition file (bdf) for the MiniZed needs to be imported. This was discussed above. Once this has been installed, use the Green Quick Start→CreateProject as shown below. Existing projects will show up on the RHS, if you have any.
2. Hit Next as shown below to create a new project.
3. Fill in the project name – the first time through the project name will default to project_1 and will increment if previous projects using project_x have already been used. I entered LEDSW as my project name, then hit Next.
4. From the Project Type" step select "RTL Project". Note: IO planning has already been completed by AVNET and we will use the constraints file that is provided with the MiniZed board in a step further below.

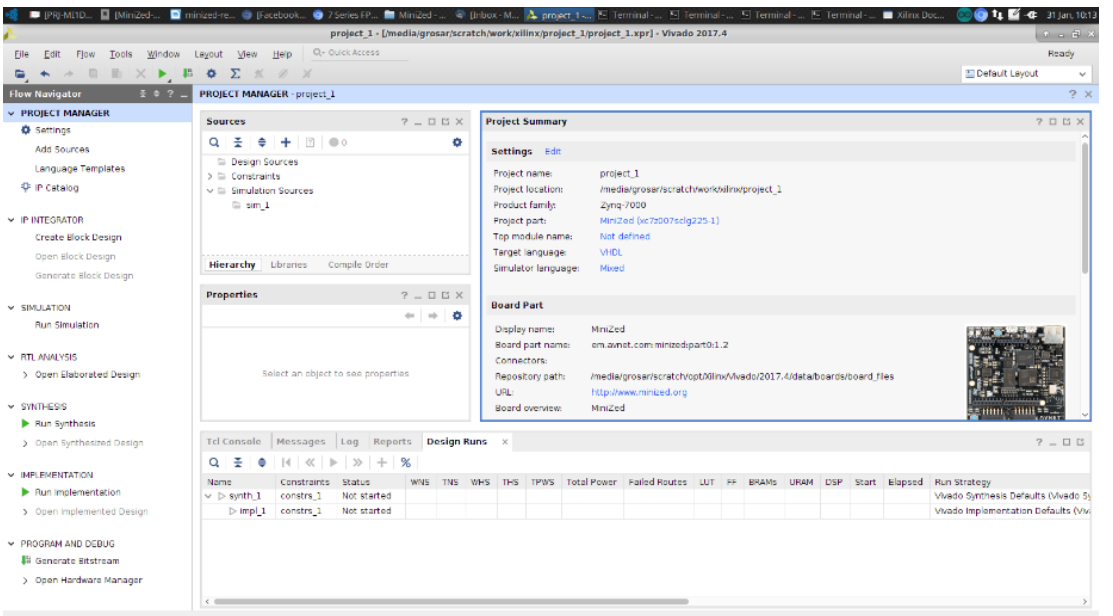


e. The “board definition file” that has previously been added, will have the MiniZed added to the database. Select it as shown highlighted below and press Next. A summary will be displayed giving you an opportunity to terminate before creating a project. If everything looks correct hit Finish.

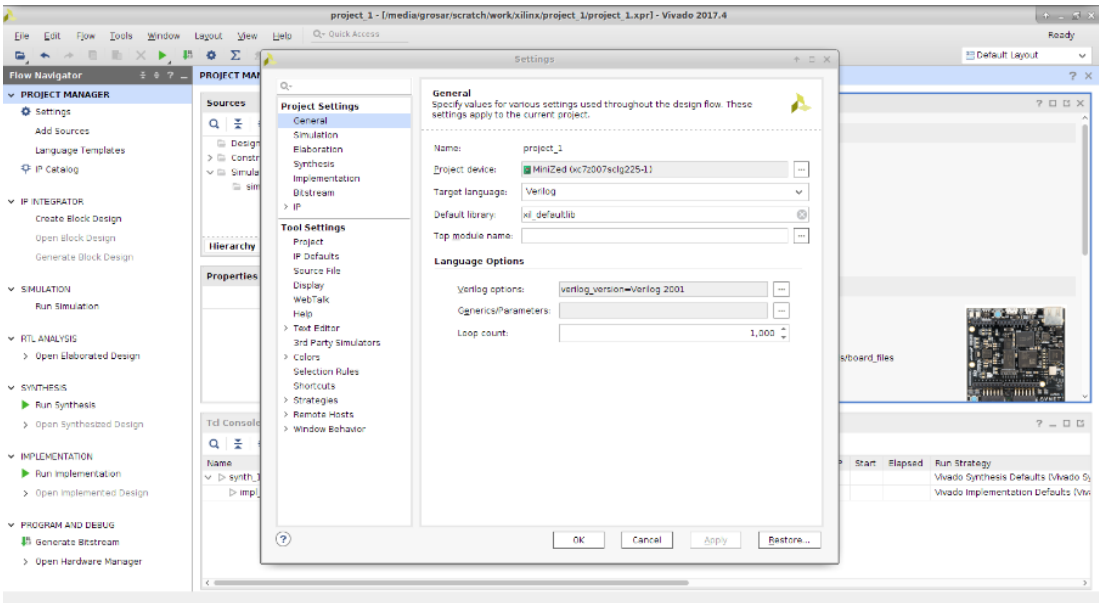


f. After several seconds you will see a screen that should look like this:

NOTE If you look in the project summary you may see the "Target Language" listed as VHDL in blue.

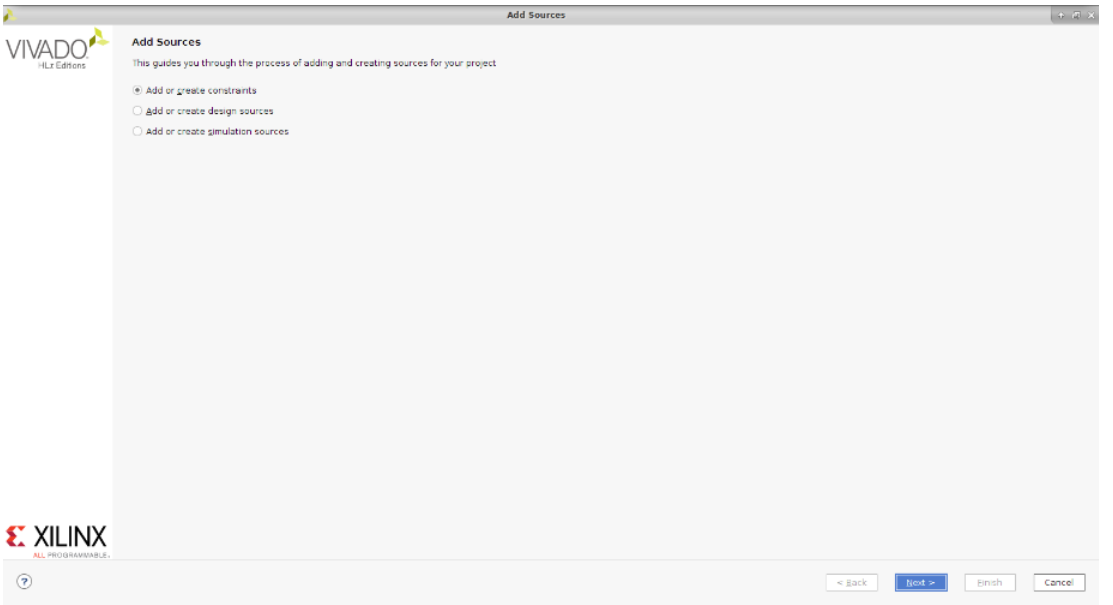


g. Once “VHDL” is pressed, a popup window appears, where the “Target Language” will be changed to verilog as shown below, by using the pull-down menu and selecting verilog. Once selected hit OK.

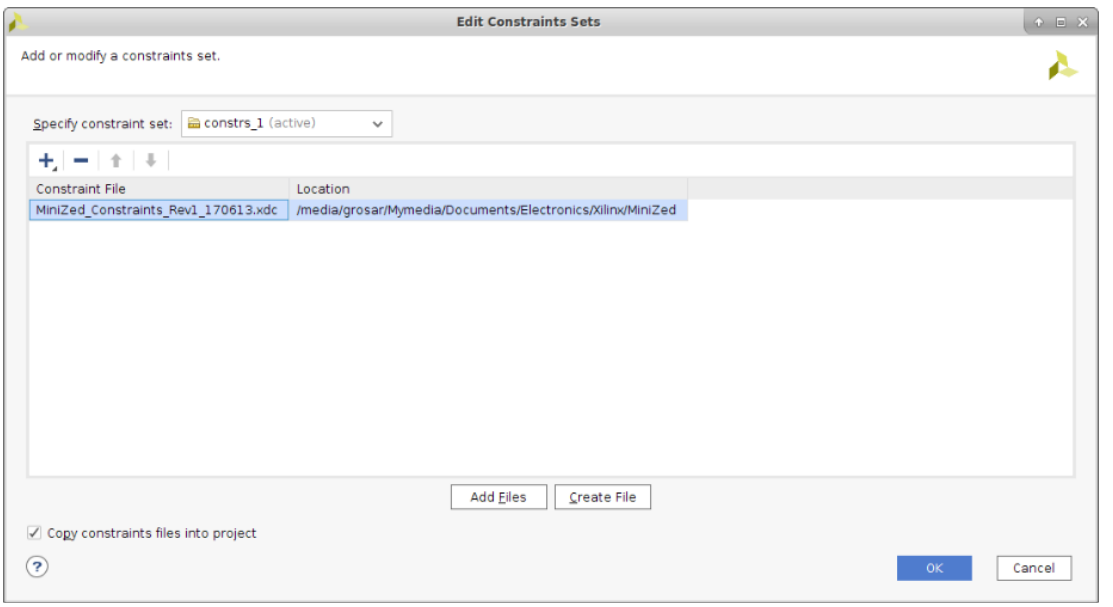


h. In the “Sources” sub window unhide constr_1 and using the right mouse button use add sources.

i. The window below pops up. “Select Add or create constraints” and hit Next.



j. Using the + button add the Constraint file that was in the files downloaded from the AVNET site. The file is MiniZed_constraints_Rev1_170613.xdc



k. On lines 80-86 of the constraints,

MiniZed_constraints_Rev1_170613.xdc, the following pins for an LED and switch relating to the on board hardware are defined and can now be referenced in the verilog code.

constraints_Rev1_170613 lines 80-86

```
# Bank 35
# Note that the LEDs and switch are shared with ARDUINO_A[3:5]. Therefore
# these next three pin locations get repeated. Depending on which features
# you use, one or the other constraints should be commented out.
set_property PACKAGE_PIN E13 [get_ports {PL_LED_G       }];  # "E13.ARDUINO_A3"
set_property PACKAGE_PIN E12 [get_ports {PL_LED_R       }];  # "E12.ARDUINO_A4"
set_property PACKAGE_PIN E11 [get_ports {PL_SW          }];  # "E11.ARDUINO_A5
```

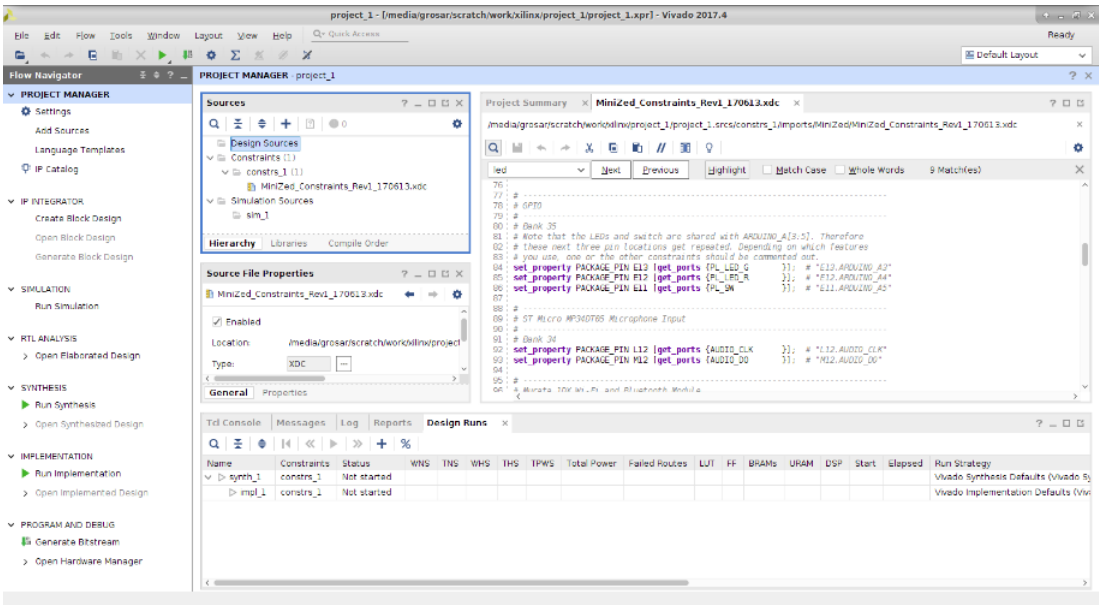
l. On lines 164-165 of the constraints, MiniZed_constraints_Rev1_170613.xdc, the I/O type is specified. The pins on Bank 35 are mapped to 3.3VDC using the LVCMOS33 standard.

constraints_Rev1_170613 lines 164-165
Set the bank voltage for IO Bank 35 to 3.3V set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 35]];

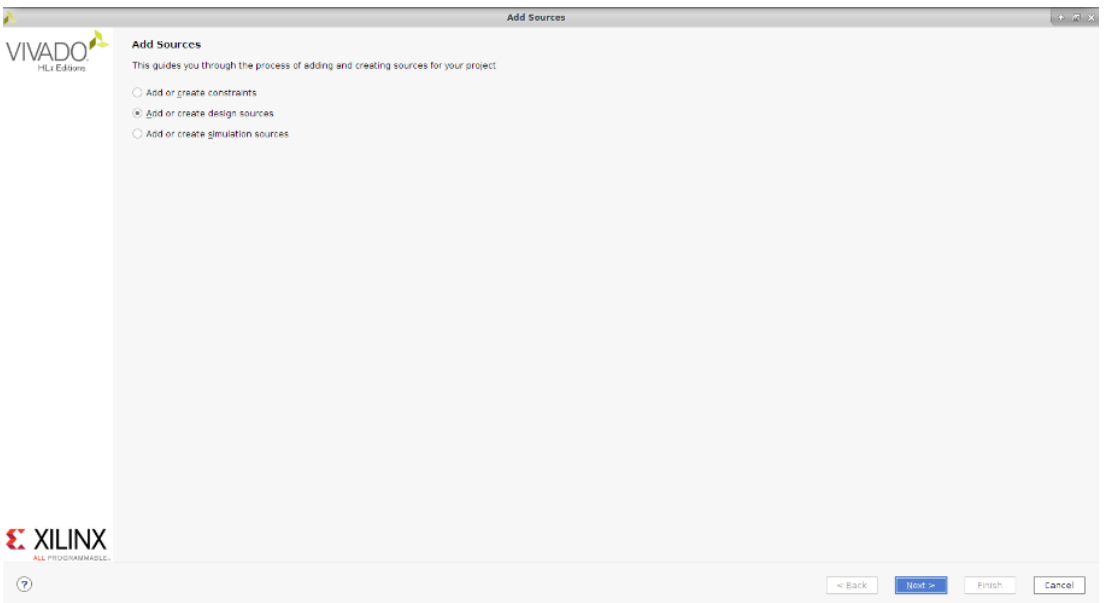
m. This completes the process of setting up the project and the constraint file, MiniZed_constraints_Rev1_170613.xdc. We are no ready to add a verilog file.

3. Writing HDL code for an application

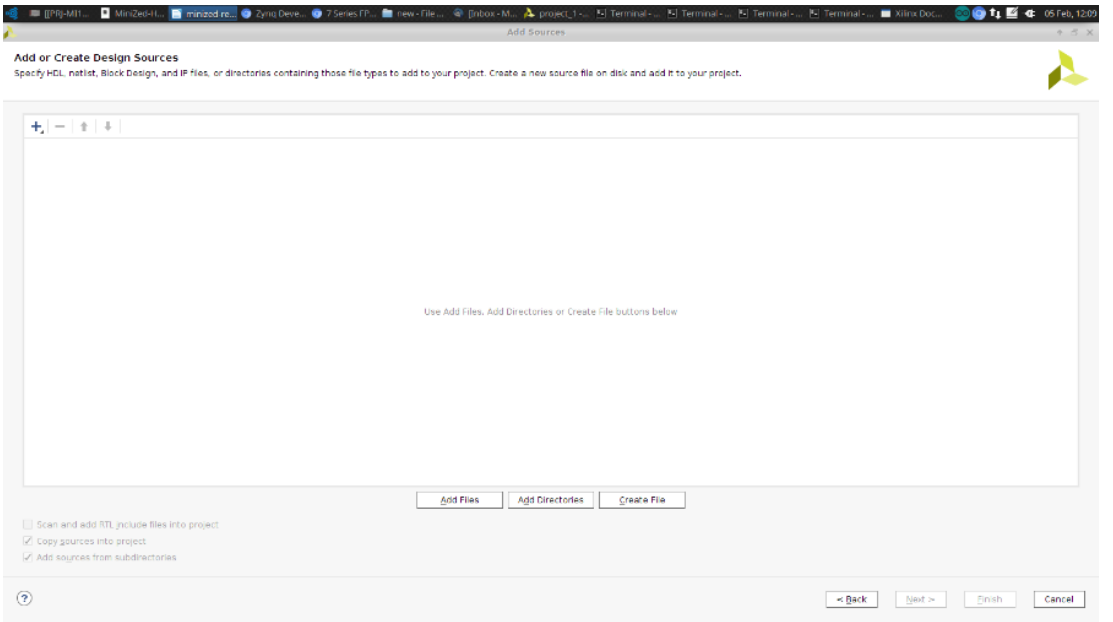
1. As shown in the screenshot below, select the Design Source, and using the right mouse button, select “Add Sources”



b. You will get a popup window, select “add or create design source” and hit Next.

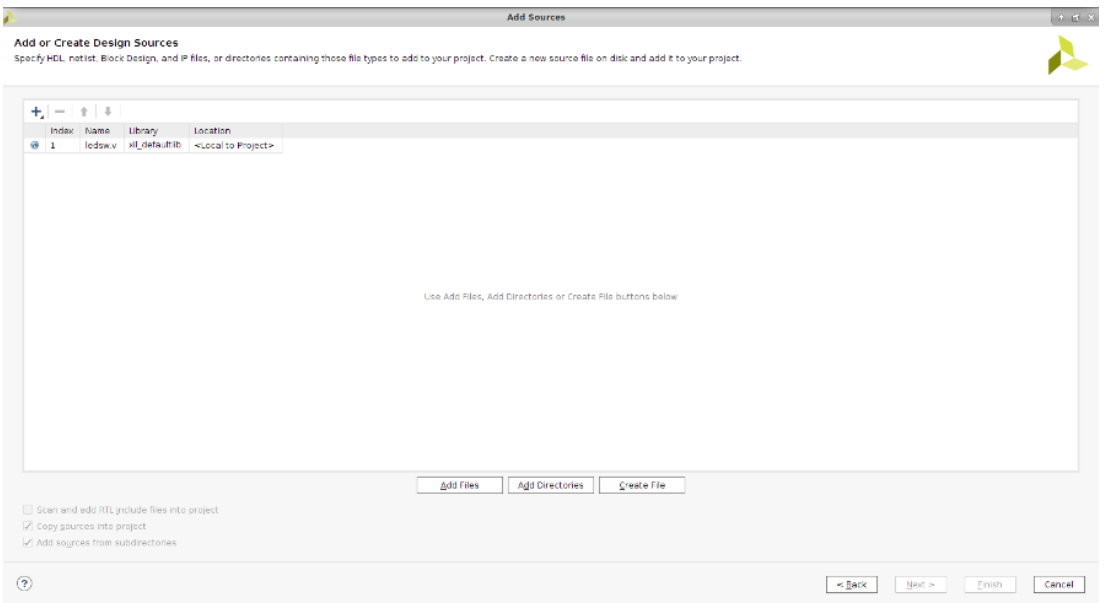


c. Select the “Create File” button and another popup window appears.

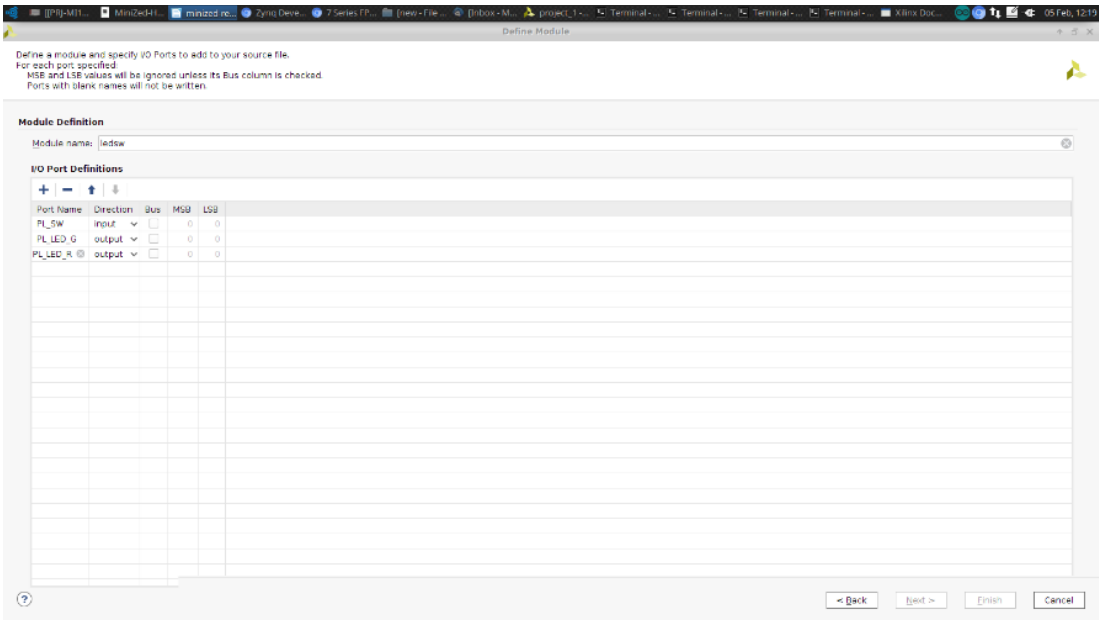


d. Type in the filename, in this instance “ledsw.v” was entered into the “Create Source File” form and hit OK.

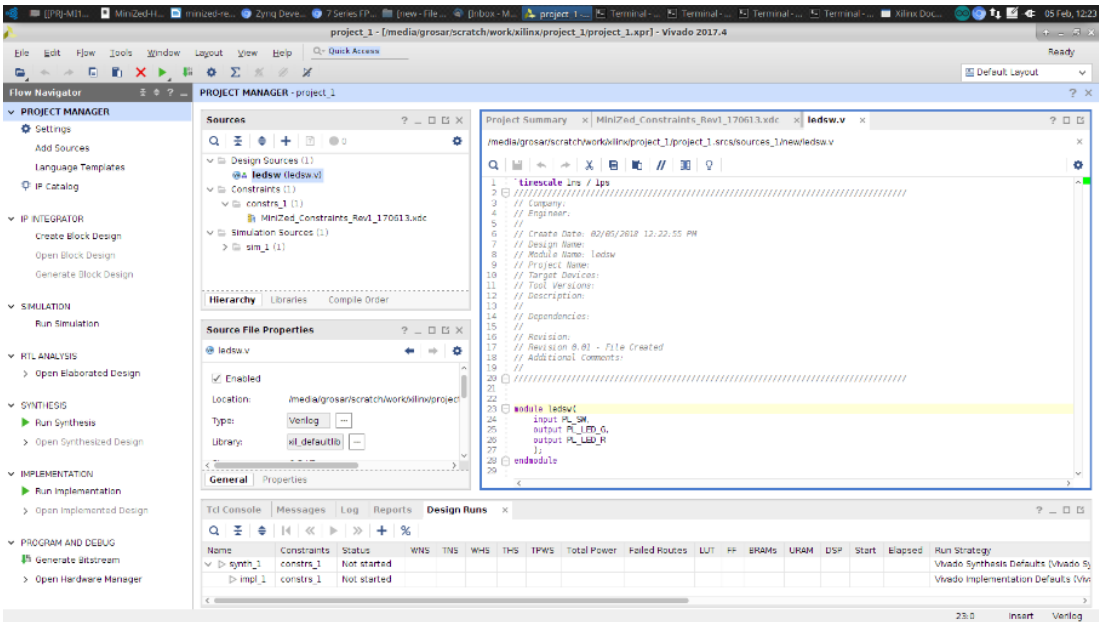
e. Hit the “Finish” button as shown in the screenshot below:



f. Type the name PL_SW, PL_LED_G and PL_LED_R into the Port Name rows and in the direction box select input for PL_SW and output for PL_LED_G and PL_LED_R as shown below, then hit the “Finish” button.



g. When complete you can look at the file that was created automatically:



h. The following code can be added to the file ledsw.v before the endmodule statement:

ledsw.v
assign PL_LED_G = PL_SW; assign PL_LED_R = ~PL_SW;

See the updated file below. In this case the green LED will be on when in switch position A and the red LED will be on when the switch is on position B. This completes the “Writing HDL code for an application” section

4. Clean compile of the HDL code (

synthesis step)

1. In the previous screenshot, press the green “Run” button and select “Run Synthesis”. In the Linux version there are options to run it on other machines or batch servers (called LSF), the default of running on this machine will be used. Press OK (default selection). In the right hand corner you should see that the synthesis is in progress. Upon successful completion you will see the following:

   NOTE: One of the reasons such a simple beginning project was chosen is that it is much easier to start with a very simple project and incrementally add to it than trying to figure out the very terse user messages.

5. Implementation

1. Press the OK and the implementation process will start. Hit the “run on local hosts” button this pops up. This will take the synthesized netlist,

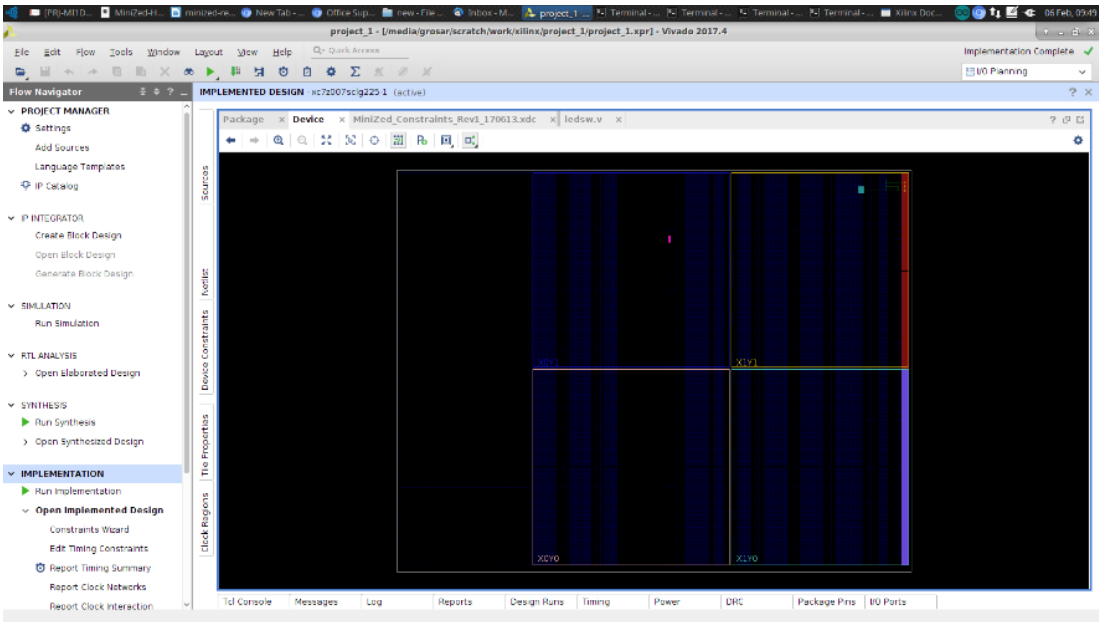
which is an ideal netlist and map it to the IC.

NOTE: On a more complex design you may want to add timing constraints to 'guarantee' that the design will meet your timing requirements. Most timing requirements are based on getting logic to the input of a flip flop with sufficient set up and hold times. This timing may need to be adjusted by logic gate delays and routing capacitance. On a complex IC design or FPGA the timing constraints can be very tricky and require many iterations. If your design is relatively slow compared to the capabilities of the process it can be quite simple, from ignoring timing constraints to creating a single one line timing constraint. Because this design contains no flip flops (combinatorial design only) we did not add any constraints.

b. When the implementation is complete, select the "Open Implemented Design"

selection and hit OK.

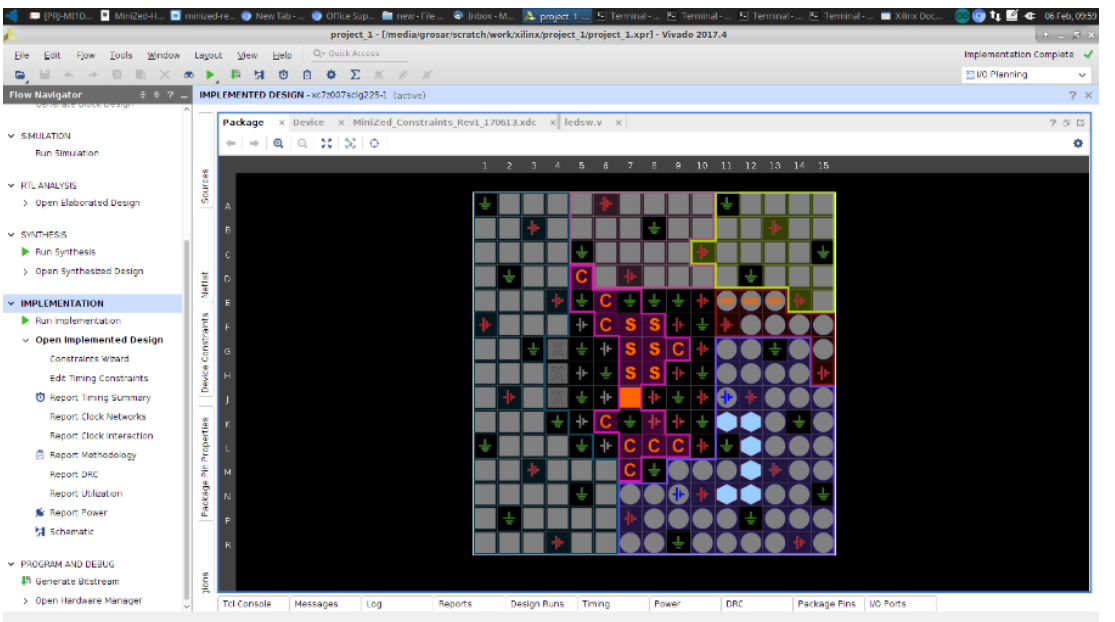
c. You can view the FPGA resources used. In the “Device” tab, and you can zoom in and review the small amount of logic resources being used.



NOTE: This design does not include any of the resources for the ARM processor and subsystem.

d. In the package view if you examine E11 E12 and E13 you will see that the usage matches the constraints that were given:

constraints
set_property PACKAGE_PIN E13 [get_ports {PL_LED_G }]; # "E13.ARDUINO_A3" set_property PACKAGE_PIN E12 [get_ports {PL_LED_R }]; # "E12.ARDUINO_A4" set_property PACKAGE_PIN E11 [get_ports {PL_SW }]; # "E11.ARDUINO_A5"



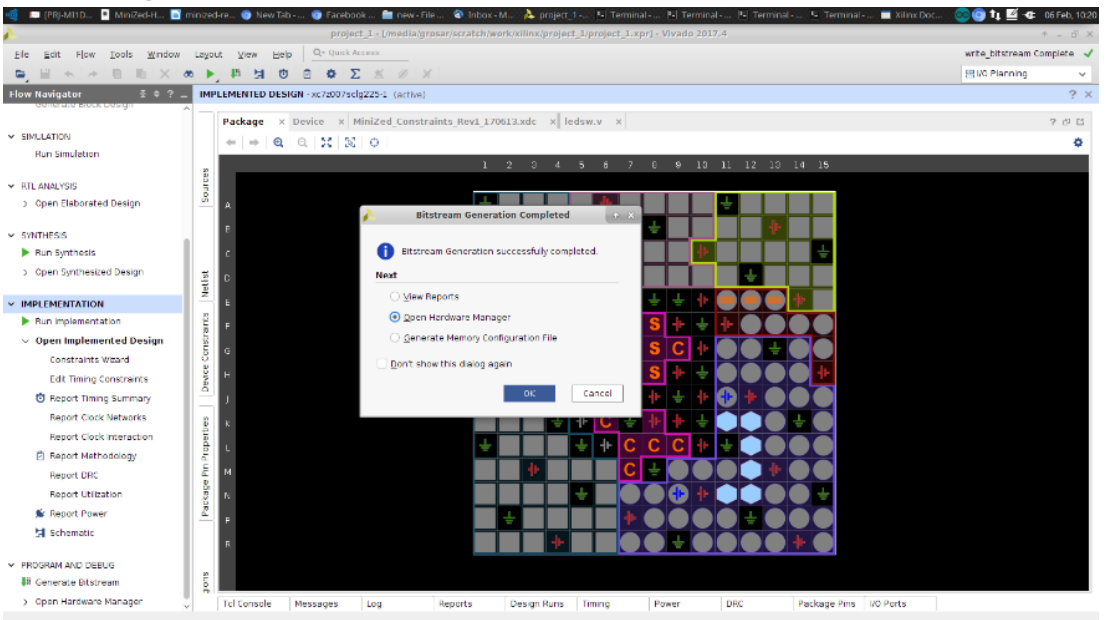
6. Writing a test bench (skipped in this exercise – see next simple project)

7. Building a bitstream

1. On the left hand side of the screenshot above

hit the “Generate Bitstream” and when the popup occurs, select the local machine as before.

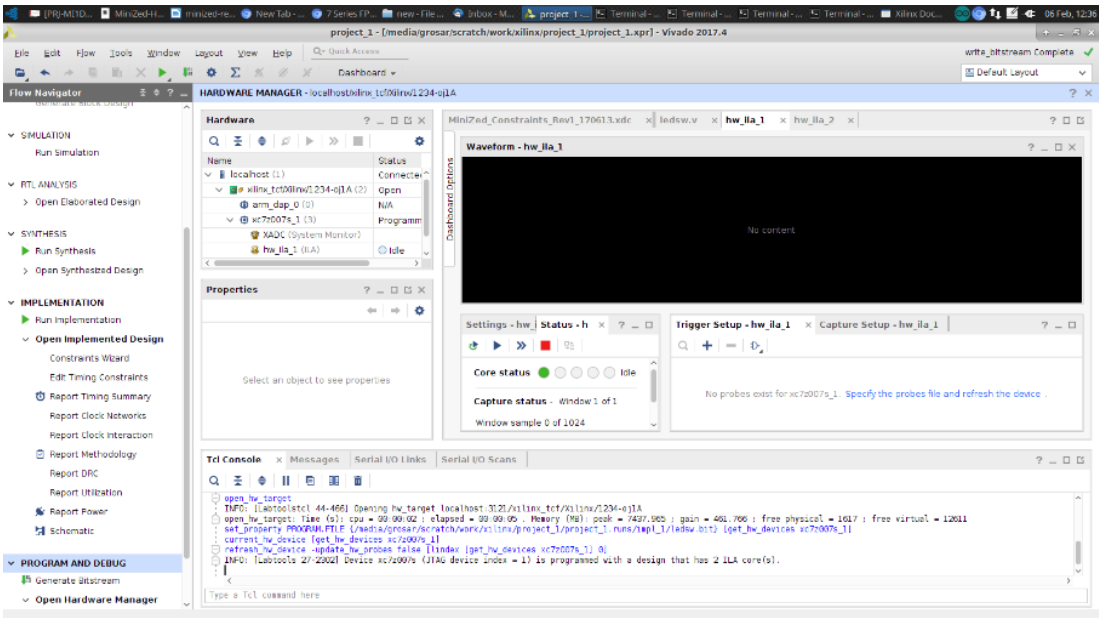
2. When completed the “Bitstream Generation successfully completed” dialog message pops up, plug your board into the USB port of your host computer and select the “Open Hardware Manager” button and hit OK.



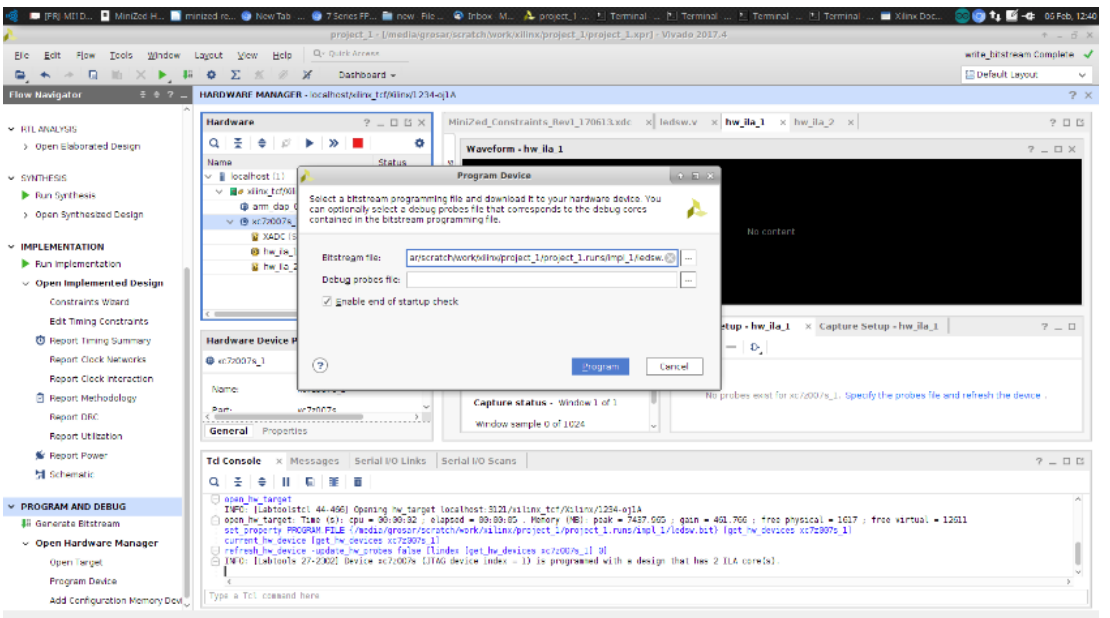
3. Now hit open target and Autoconnect in pulldown menu

4. This screen indicates the JTAG interface has connected to the MiniZed board and lists all of

the subsystems Vivado can communicate through JTAG.

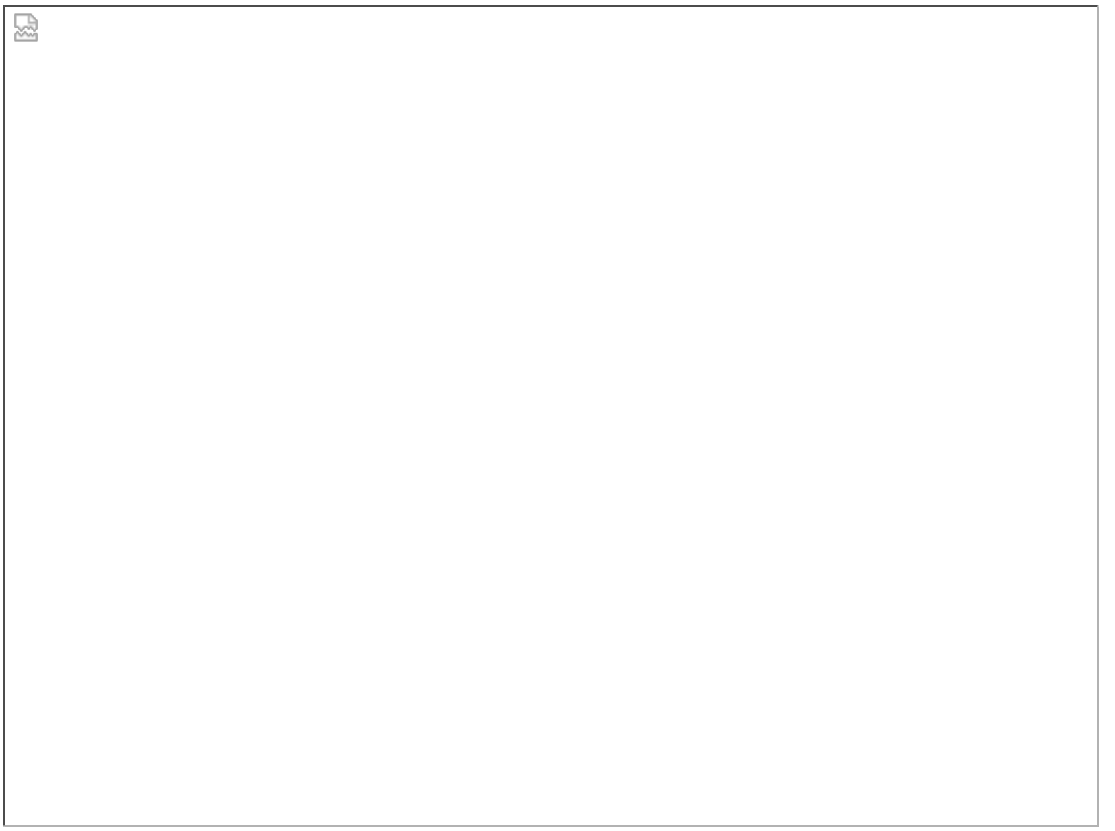


e. Hit program and the FPGA is loaded with the simple test program.



8. Testing the hardware

The hardware worked correctly – a shot of the board with the switch in one of the two positions is shown for reference.



9. Restoring the stock image

Powering the device off then on again restored the image as it was loaded through JTAG and not programmed into the onboard FLASH.

VIVADO SIMULATION:

The first project in the "**Embedded System Design with Xilinx Zynq FPGA and Vivado**" did not map directly to this board. The digital I/O required 4 switches and 4 LED's. Eight digital I/O pins mapped to the Arduino's interface connector.

This was a trivial assignment, and **I didn't really want to wire this up**, I decided to use this to illustrate the **simulation aspects of Vivado that was skipped in the previous example.** Like said previously, it is always a good idea to come up with a very simple project to test things out.

This exercise will be used to show some of the simulation capabilities and at this moment I will not be wiring them up as the time could be spent better on other exercises.

The code used is shown in the screenshot below, the module blinkey() is the top level

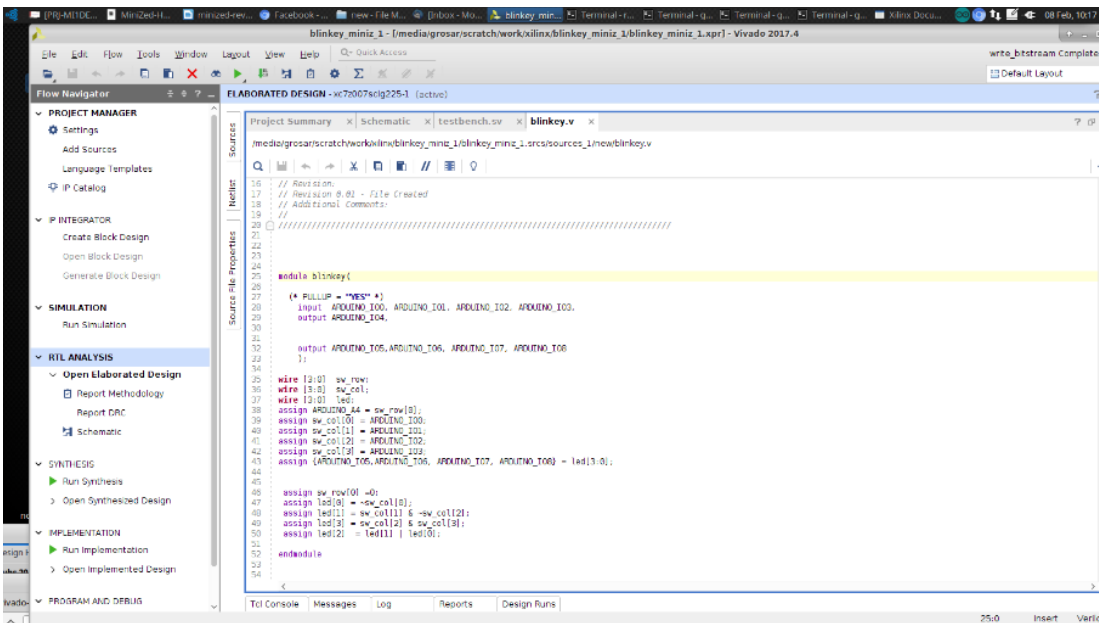
module. The input and outputs are mapped to the names given in the constraint file and notice the use of the Vivado compiler directive that sets the inputs to pullups. This can be changed in the constraints file as well but I didn't want to change the stock constraints file at this time. ARDUINO_IO0 to ARDUINO_IO8 are connected to the Arduino header pins.

```
  (* PULLUP = "YES" *)
  input ARDUINO_IO0, ARDUINO_IO1, ARDUINO_IO2, ARDUINO_IO3,
  output ARDUINO_IO4,
  output ARDUINO_IO5,ARDUINO_IO6, ARDUINO_IO7, ARDUINO_IO8
  );
```

I have a 4x4 membrane keypad that I selected for the 4 switches. The four column switch pins are pulled up with a pullup resistor and mapped to (ARDUINO_IO0 to ARDUINO_IO3)

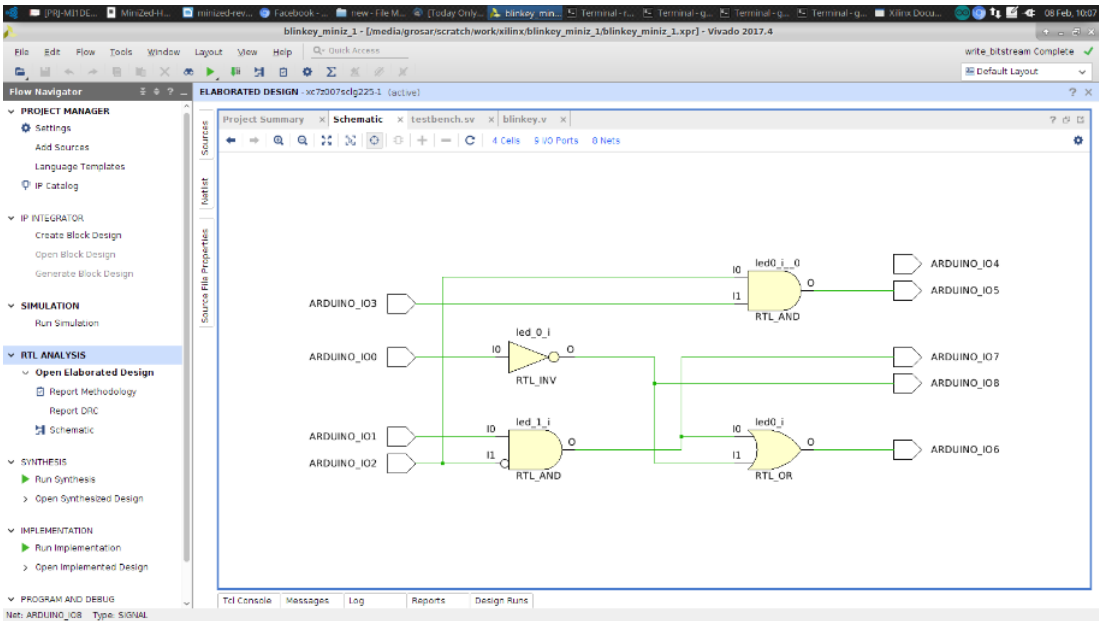
and the row switch (ARDUINO_IO4) is set to 0V so when a key is pressed it will go from a logical ‘1’ to a logical ‘0’. Four LED’s are mapped to ARDUINO_IO5 to ARDUINO_IO8.

Names are assigned to the generic pin names and the logic from the switches to the LED’s are the logic from the project assignment. The logic is combinatorial so it does not require a clock yet.



The schematic from the implementation is shown in the screenshot below. That was captured from the implementation→schematic

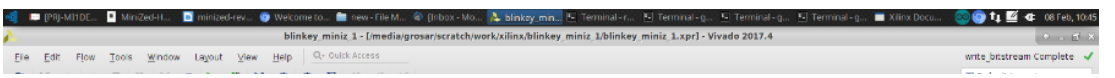
view.

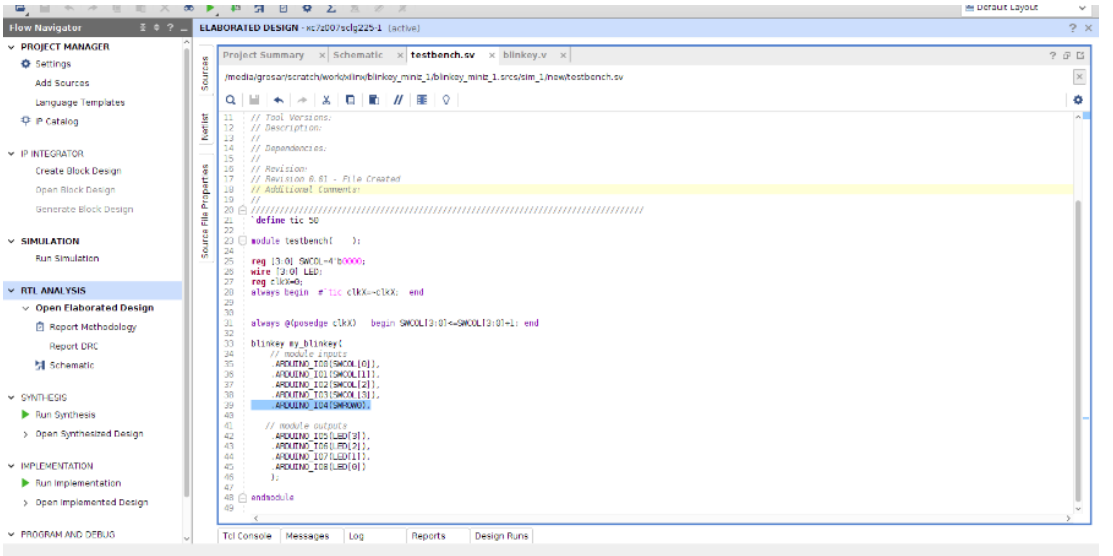


- In order to demonstrate the capabilities of the very powerful built in Vivado simulator, a simulation file called testbench.sv was created and used for the simulation source. This is a very powerful aspect of Vivado and this feature is available whether you bought the board or not.
- In this case the simulation file is a very simple behavioral model (meaning you wouldn’t want to synthesize it). Also an important point is to make sure you add your simulation sources in

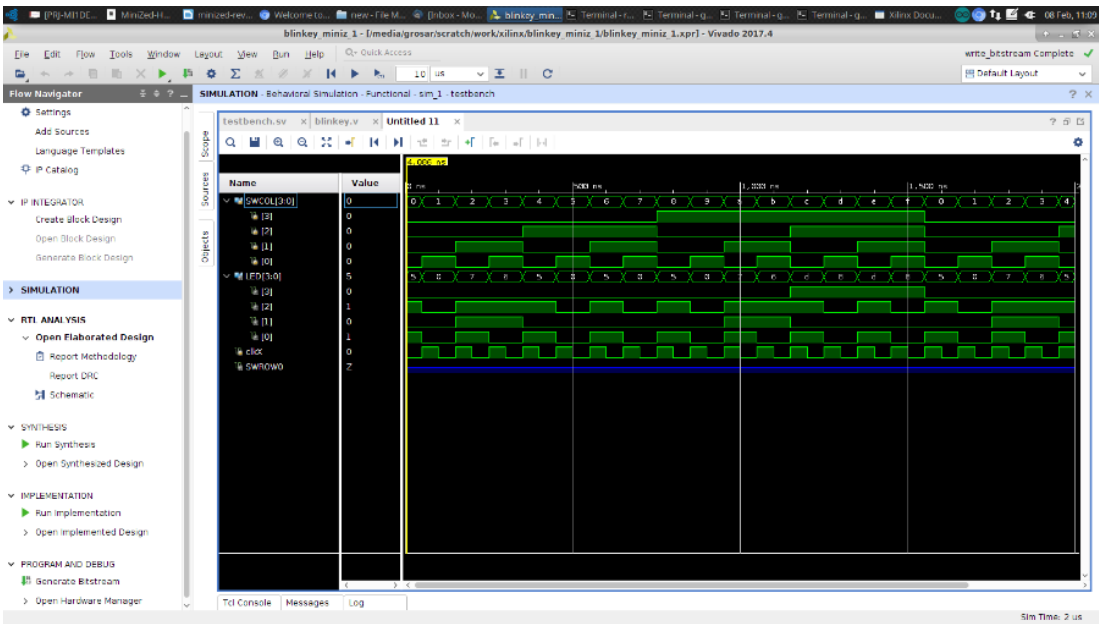
the project section called “Simulation Sources” It can cause some major debug headaches if you try and synthesize it!

- The test bench, testbench(), instantiates the blinkey() module and creates some signals.
- The value `tic is used to flip the clock clkX every 50ns.
- All possibilities of the column switch are tested by incrementing through all 16 possibilities. This is done by creating a simple 4 bit counter called SWCOL using 4 register bits.
- One other point, to change the length of the simulation, you need to change the setting, otherwise it defaults to 1000ns. That is done by right clicking on the Left Hand Side Simulation pull down, going to the Simulation setting and changing it there. This step was not obvious to me the first time I used the simulator.





- The simulator output is shown and produces the desired output. Being able to simulate using HDL is a feature that used to cost thousands of dollars (at least hundreds) and is extremely wonderful.



NOTE: I set the simulation up using system

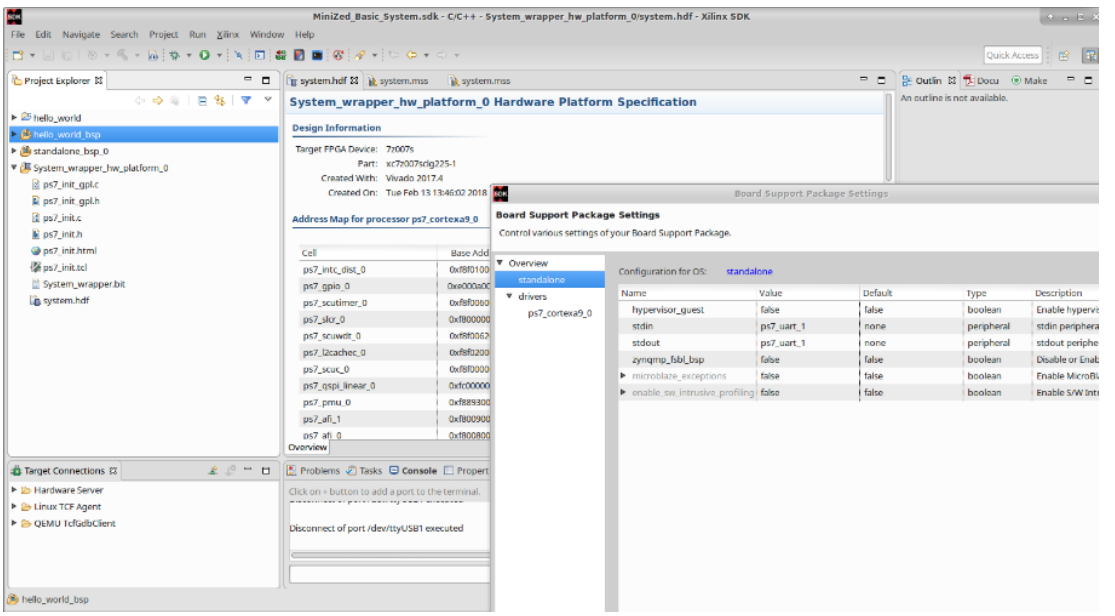
*verilog (.sv), which has many more capabilities than 'vanilla' verilog (Verilog 2001) has, albeit I did not use those special features. Using some of the features during simulation could be a good way of building up skills and determining what features are supported and what ones are not. The Xilinx application note UG900 may be referenced for both System-C and System Verilog. It is also worth mentioning that with the installation of the Xilinx Documentation Navigator you can search for many topics and this is a huge data base of informatio*n.

- Initially we looked at the PS (Processing System) by poking around in the embedded Linux and the last two exercise we looked at the PL (Programmable Logic) portion of the MiniZed, now it is time to see how we can use both PS and PL. I will just be touching on the aspects of this in the Road-Test.

- After having difficulty in jumping into Lesson Two ("Hello World") of the "**Embedded System Design with Xilinx ZYNQ FPGA and Vivado**", Udemy course, A more logical approach to getting my first program to operate through Vivado's SDk system was started. (See http://zedboard.org/support/design/18891/146). There is quite a bit of really good training prepared by AVNET for the MiniZed.
- Tutorial 01: Build a Zynq Hardware Platform: This went very well and a Zynq Platform was created. This creates the files necessary for exporting the design to the SDk. The approach was pretty simple.
- Tutorial 02: First Application - Hello World: The directions in this tutorial did not match how the SDk behaved on my version and with Linux. The tutorial wanted to show how the two tools can work independently.
- Tutorial 03: MiniZed Zynq Test Apps: Memory

Test and Peripheral Test are built, by this time I did not need to use the tutorial as it could be built exactly like the Hello World test.

- There were two takeaways with this exercise, the first takeaway, one that has already been documented in the MiniZed FAQ when googled, is that the UART needs to be configured correctly from within the SDk. **Use right mouse button on hello_world_bsp=>Board Support Package Settings and select standalone, stdin and stdout should be set as shown below.**



- The second takeaway is to **have the boot source switch set to F (for FPGA),** set this so that each time the board is reset it doesn't try and load the Linux OS. Although I was able to workaround this one by resetting from the switch in between loading code from the SDK to the MiniZed.

  NOTE: I also used Putty instead of the built in SDK terminal as the SDK one was a little quirky.

  The output to the terminal is shown below:

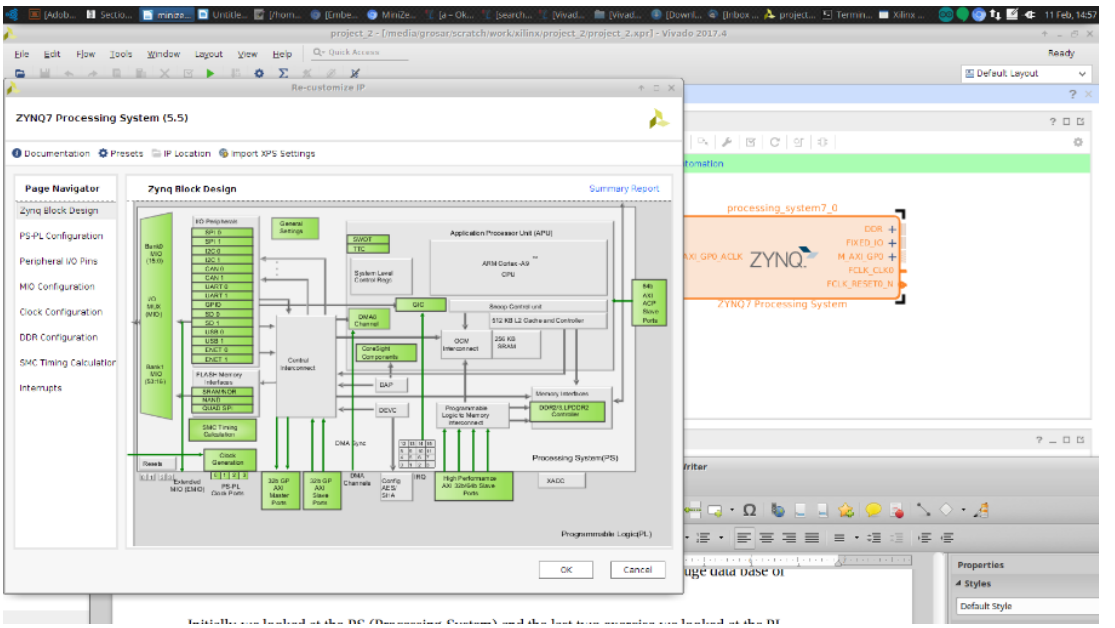
I didn't create a lot of screenshots as the Tutorials referenced above did a good job and there was no sense in repeating it here.

I had some difficulty with the Sdk and it was for a rather silly reason. With my installation with Ubuntu Linux somehow the desktop Icons were not built so I created some Icon launchers by hand and inadvertently did not create one for the Sdk. I thought that the SDx was the same tool. I spent several hours trying to figure out what I was doing wrong. Once I figured that I need to link the program /opt/Xilinx/SDK/bin/xsdk to an Icon, things worked as the AVNET Tutorials claimed. It’s funny how something like that can get you derailed for a couple of days.

AVNET Tutorials cover this also so this may be a bit of a repeat if you follow Tutorial 3 as mentioned above.

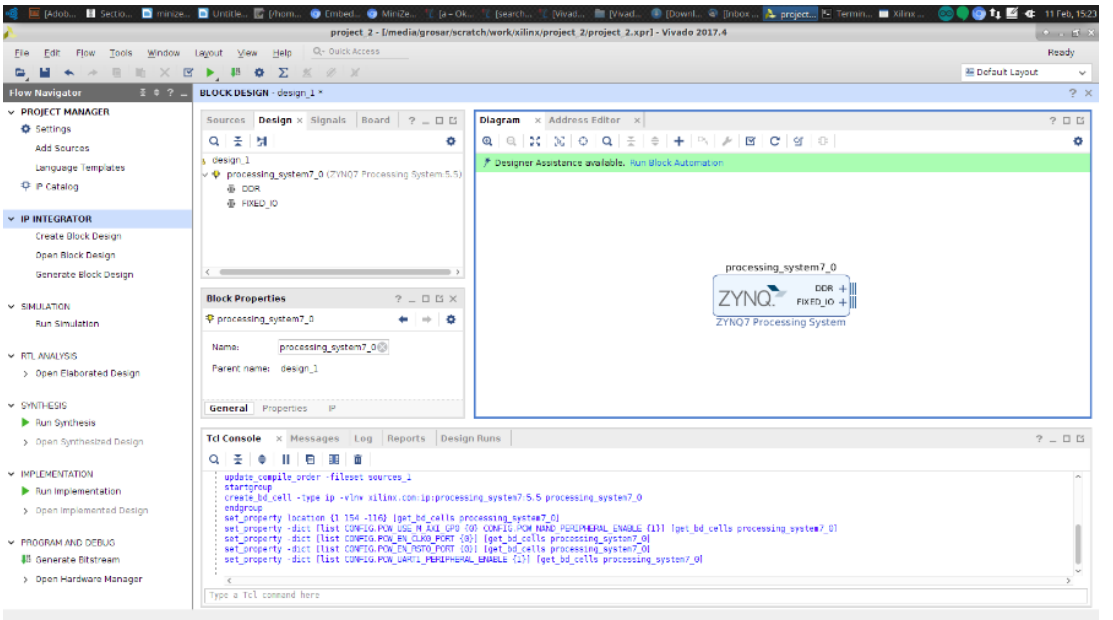
1. **Start a new RTL project**=>IP

Integrator=>Create Block Design +IP (search for ZYNQ) select it and then double click it and you will see the following block diagram as shown below:



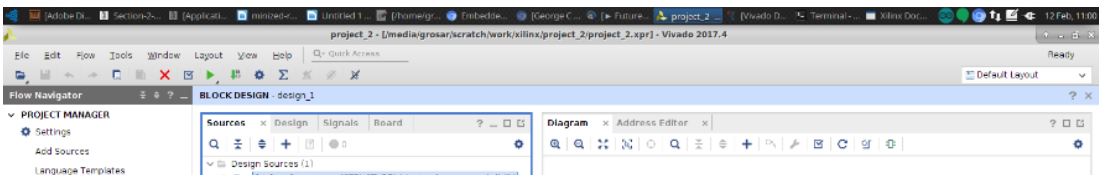
1. PS-PL Configuration=>AXI Non Secure Enablement=>GPI Master AXI Interface=>M AXI GPIO Interface=>(uncheck)
2. Clock Configuration=>PL Fabric Clock=>Fclk Clk0=>(uncheck)
3. PS-PL Configuration=>General=>Enable Clock Resets=>FCLK_RESET0_N(uncheck)
4. Peripheral and I/O Pins=> UART 1 => (check)

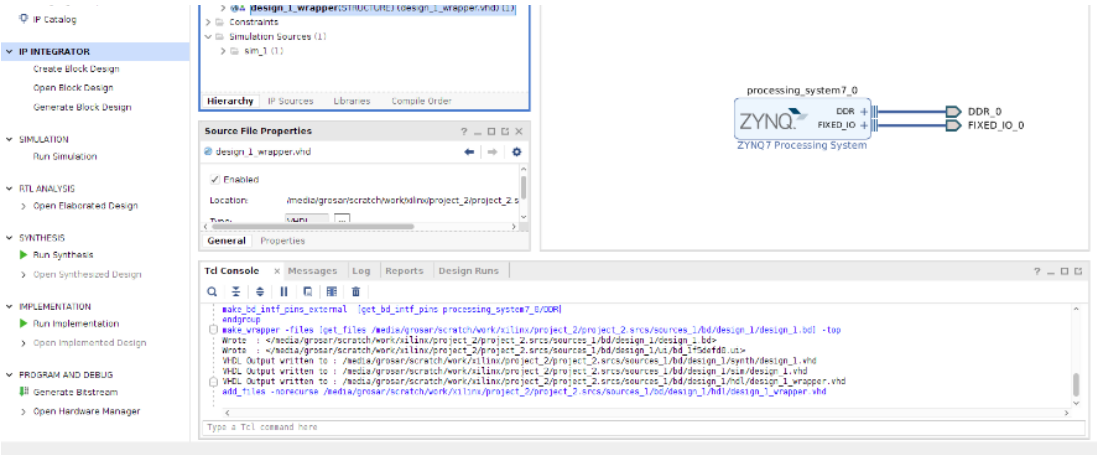
5. Hit OK and you should see the following updated ZYNQ block



2. **Create HDL Wrapper**

1. Touch the DDR port connection with the most pointer, select “Make External, repeat for FIXED_IO
2. Select Block Design=>design1=> select “Create HDL wrapper” and use the default option
3. You should see the project as shown below:

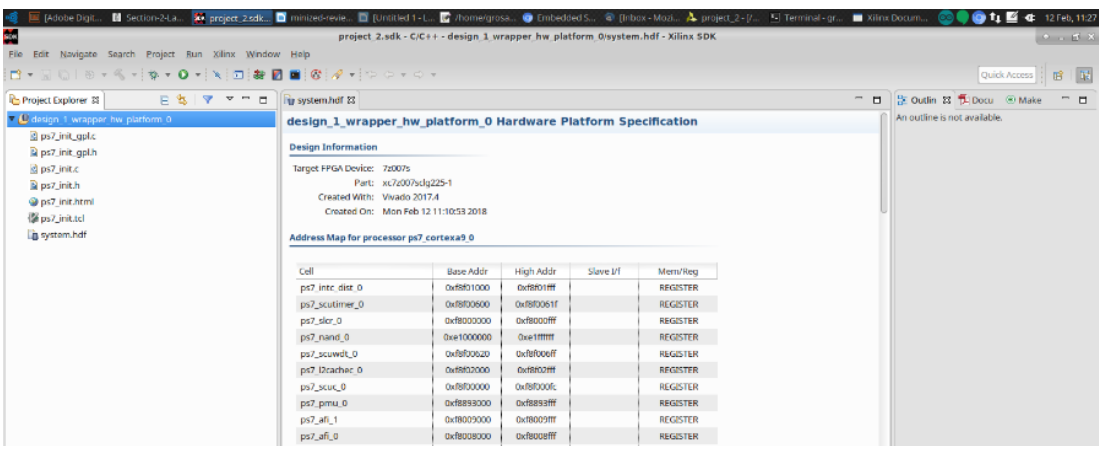


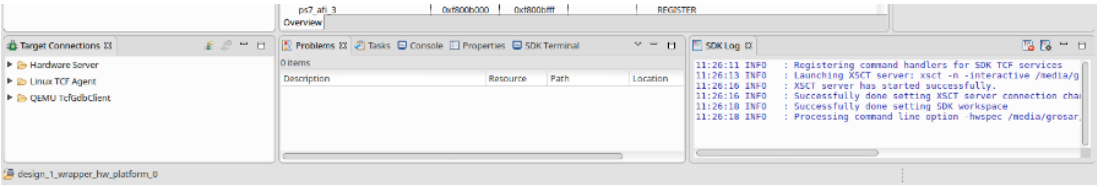


3. Generate Bitstream and start SDK

1. Press Generate Bitstream, wait a few minutes to completion
2. Export=>Export Hardware (select to local project) and hit include bitstream selection box in popup window and hit OK
3. Launch SDK use default settings

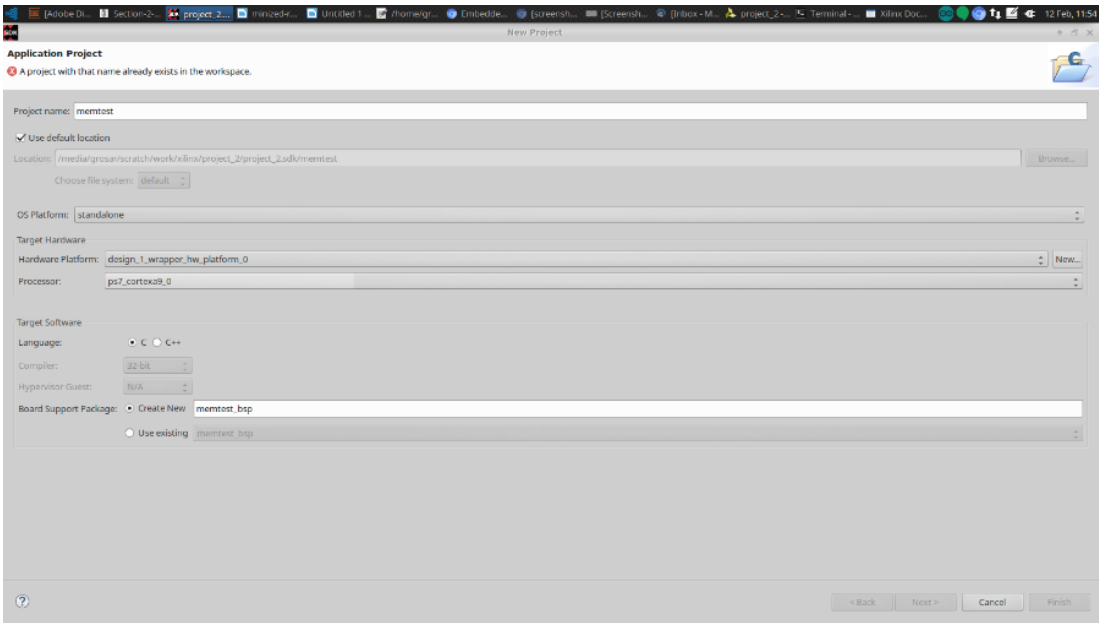
   You should see the screenshot as shown below upon the SDK opening



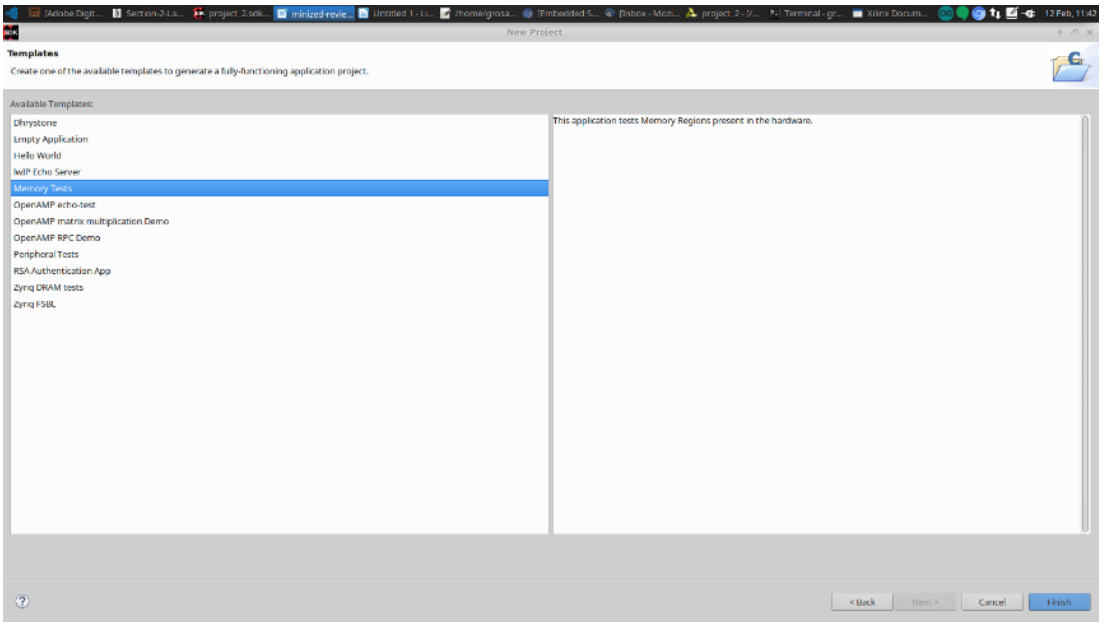


4. Application

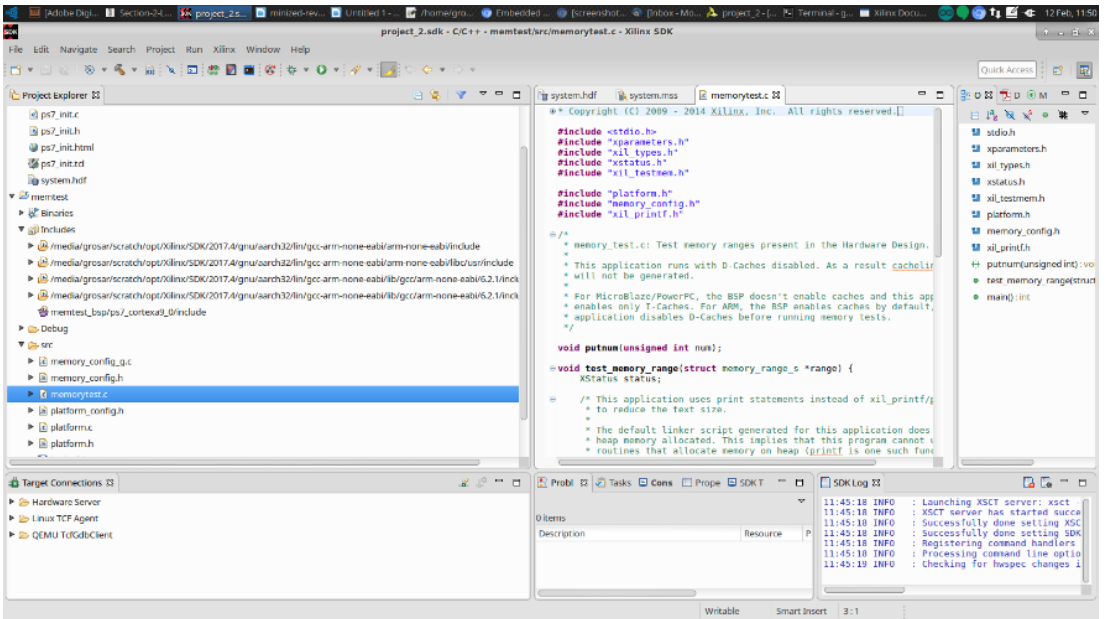
1. File=>Application Project
2. type in memtest into the popup window as shown below hit NEXT



c. Select Memory Tests (from the available templates) and hit Finish



d. Once created you will get the screen similar to this (memtst→src was expanded to show what was created)



e. Select Window=>ShowView=>Other=>Terminal=>Terminal

The memory test worked as described.

At this point there are a few items that are left on my To-Do list:

1. Build an image "from scratch" that is comparable to the "stock image"
2. Build a Yochto Image
3. Write new image to the MiniZed Board and Test
4. Replace Image with "stock image"

During the process of working on Item 1,

- I got to the point where in the "Embedded

System Design with Xilinx ZYNQ FPGA and Vivado Project" I was getting errors.

- I went to try the Zynq Vivado System Example: this first required an update to the example to bring it up to date with the current Vivado version and once I tried to Implement it using the MiniZed board and I was getting errors. At this point I plan on going back to the Tutorials and working out the rest of the details.

  For the sake of the Test-Drive I believe that this completes the test drive. The MiniZed Board is a remarkable board and has capabilities that far exceeded my expectations. That said, building an image from scratch and completing the last items on my To-Do list did not go as well as I would like.