PYNQ Radio – Final Report

Team Members: Harveen Kaur, Rajat Gupta, Rohit Kulkarni, Vishwesh Rege

1 ABSTRACT:

General purpose processors (commonly called CPUs) are employed in a number of applications from smartphones to even cars. However, these processors can only provide limited performance and consume a lot of power. Thus, recently there has been significant interest in employing alternative platforms for applications that require very high performance and energy efficiency. One such platform is Field Programmable Gate Array (FPGA), which allows us to build dedicated hardware for a particular application. FPGAs are tailored for a particular application and can do work parallelly, leading to much better efficiency. Our project aims to take one such heavy duty application (FM radio receiver) and "accelerate" it using a combination of general purpose processor and FPGA on the same chip. This chip is part of a board called the PYNQ board and thus, we call our project PYNQ Radio.

2 INTRODUCTION:

Our project entails taking a computationally intensive software application, identifying the compute-heavy part, and "offload" that part to dedicated hardware (FPGA), in order to make the resulting application much more efficient in terms of computation and energy. The final implementation has the lighter part running in software and the compute-intensive part running on dedicated hardware, i.e. the application would run on a heterogenous platform.

One such heterogenous platform is the Xilinx Zynq [1] - which integrates a general-purpose processor (ARM CPU) with an FPGA from Xilinx. FPGAs or Field Programmable Gate Arrays can be thought of a bunch of hardware circuits that are "programmable", i.e. we can define the behavior of these circuits using programming languages such as Verilog or C/C++. Once the behavior is defined, the FPGA may be "programmed". After programming the FPGA, it exclusively executes the application it was programmed for. Thus, there is a lot more scope for parallelism and efficient design (tailored to the application) while using an FPGA. This leads to significant performance and energy benefits, and is known as "hardware acceleration".

In recent years, a number of applications have been accelerated using FPGAs. Due to their performance and energy efficiency, FPGAs can be a great addition to embedded applications. However, they have a steep learning curve, especially for software developers. PYNQ [2], or Python for Zynq, is an open source project from Xilinx which aims at overcoming this limitation. PYNQ is designed to abstract the low-level details of hardware programming, and instead present a relatively intuitive programming model for embedded systems designers and software developers looking to explore hardware acceleration. The PYNQ open source community is actively trying to create a library of hardware blocks that may be easily integrated with software applications. As mentioned before, PYNQ is based on the heterogenous Zynq platform by Xilinx, which contains an ARM processor and a Xilinx FPGA on the same chip.

In our project, we decided to use the PYNQ board, since it is a new platform and has a growing community of hardware developers contributing to make hardware acceleration more accessible. Thus, we used the PYNQ platform to accelerate a computationally intensive application. PYNQ is built around the Python programming language and provides constructs to abstract details of FPGA programming, by wrapping them in special objects called "Overlays". Thus, the goal of our project can be thought of as developing one such "Overlay" and using it in an embedded application.

The application we chose to accelerate is based on signal processing. We wanted to choose an open source application, since we aimed at open sourcing our final hardware implementation as well, and contribute to the PYNQ developer community. A perfect candidate for our requirements was GNU Radio [3], which is a set of software blocks written in C/C++. These blocks can be connected together to implement various digital signal processing application in software. Once such application is software defined radio, which entails doing radio related processing in software. Recently, software defined radio has become quite popular, and thus, we chose this as our application. To limit the scope of our project, we chose one particular area of software defined radio - an FM receiver. We implemented a full FM receiver on the PYNQ board - with the computationally intensive part (FM demodulator) as an "Overlay" (i.e. in FPGA) and the other parts (resampling, filtering, volume control etc.) in software.

Below is a block diagram of all the modules required to implement an FM receiver (in software), and indicates the module that we accelerated in hardware by creating an Overlay:

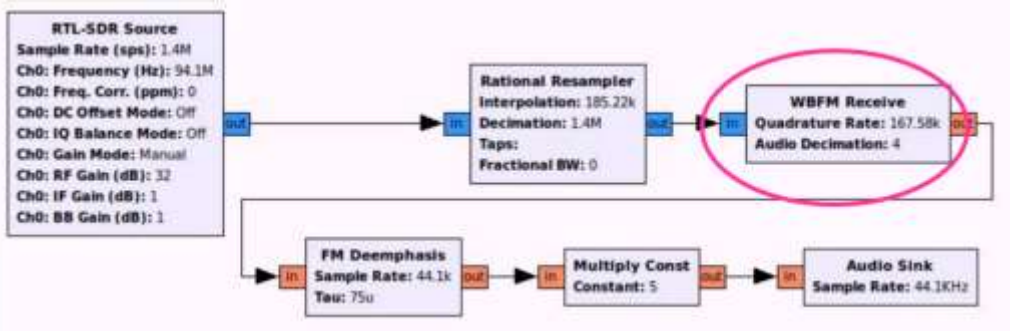


Fig 1: Block diagram for FM receiver functionality in GNU Radio. The encircled block – WBFM receiver – would be offloaded in hardware.

For FPGA programming, we used the Xilinx Vivado suite, which is a set of High Level Synthesis (HLS) tools that allow hardware programming using high level languages like C/C++, and design tools to integrate hardware blocks together and generate a “bitstream” which can be used to program the FPGA. The software portion running on the ARM processor on the PYNQ board is in Python. Communication between the ARM processor and the FPGA is through Direct Memory Access (DMA) via the AXI interface, which provides efficient copying of blocks of data between the CPU memory and the FPGA. We created a custom IP block for FM demodulation (called WBFM_accel) using Vivado HLS and connected it with DMA blocks for read and write data transfer using Vivado Design tools. Then, we used the Python library provided in the PYNQ code base to access the DMA and carry out data transfer between the CPU and FPGA. Finally, we created a custom block in GNU Radio which encapsulated the DMA transfers, such that the interface to the FPGA could be easily included as part of a GNU Radio flow graph.

Our final demonstration consisted of a comparison between a complete software implementation (baseline implementation) of the FM receiver, and the hardware accelerated counterpart (optimized implementation). It could be clearly seen that offloading the FM demodulation functionality to the FPGA reduced the CPU utilization significantly.

The major contributions of this work are:

1. Creating an Overlay for FM block in GNU Radio for the Pynq board. To the best of our knowledge, this is the first work that has ported a GNU Radio FM block to the Pynq board.
2. Creating a custom block in GNU Radio for data transfer between the CPU and FPGA through DMA.
3. Creating an "assignment" like step-by-step detailed document to replicate our work. This is accompanied by code that can be used as a base to recreate our project by following the steps. These are present at: https://github.com/harveenk/PynqRadio

Note: The 'technical details' section of this project is covered as addendum in this report, where we have created an assignment for students to try out more applications such as the FM receiver.

3 MILESTONES

Drawing from the high level goals mentioned in Section 1.4.1, we had initially developed the following milestones.

Each heading is a high level milestone, followed by bullet points which describe the low level milestones which must be achieved to complete the high level milestone.

- ✓ **RUN THE SOFTWARE FLOW FOR GNU RADIO (WITHOUT OFFLOADING) ON PYNQ BOARD**
  - Familiarization with the board and RTL-SDR antenna
  - Installing required components
  - Understanding GNU Radio flow, make sure FM receiver is working completely in software
  - Demonstrable output: FM receiver running in software on PYNQ Board

- ✓ **OFFLOAD A SIMPLE BLOCK TO THE FPGA**
  - Familiarization with the Vivado flow
  - Create a simple block and testbench using HLS
  - Gain experience programming and testing in HLS, solve any roadblocks early on
  - Demonstrable output: Simulation of the block in Vivado/output on FPGA

- ✓ **INTERFACING ARM PROCESSOR WITH FPGA FOR A SIMPLE BLOCK**
  - Understand interfacing the processor and FPGA in Python on PYNQ platform
  - Solve any roadblocks early on
  - Demonstrable output: Simple application demonstrating interaction between processor and FPGA, e.g. playing audio

- ✓ **PROGRAMMING THE BLOCKS TO BE OFFLOADED TO FPGA**
  - Finish programming all required blocks (Volk, atan, FIR, IIR)
  - Simulate blocks and verify correctness using Testbenches

- This would be our major milestone
- Demonstrable output: Simulation of the blocks in Vivado

✓ **INTERFACING GNU RADIO SOFTWARE COMPONENTS WITH BLOCKS IN FPGA**
- Ensuring seamless connectivity between the software blocks (Rational Resampler, FM De-emphais) and hardware components (WBFM receiver, Audio)
- Demonstrable output: Complete FM receiver functionality and documentation

✓ **DOCUMENTATION**
- Compile a step by step manual with instructions for replicating our work
- Create the video and update website

The major change that we made to these milestones was to directly start programming the blocks to be offloaded to the FPGA (Milestone 4) instead of first trying to offload a simple block and interface it. This was done due to the realization that a major component of this project was interfacing the WBFM block with the ARM CPU and we anticipated early on that it would take a significant chunk of the time to get right. Thus, in order to be able to devote sufficient time to the Interfacing part (Milestone 5), the slightly redundant milestones (2 and 3) were removed. Instead, we focused on Programming the blocks, Testing them and Interfacing.

4 CONCLUSION

In this project, we built an FM receiver using GNU Radio software running on the PYNQ board. The PYNQ board consists of an ARM processor and a Xilinx FPGA, which can communicate with each other. We were successful in completing 2 implementations: A baseline implementation--where the entire processing involved in FM reception is carried out by the CPU with the help of existing blocks in GNURadio—and an optimized implementation, where FPGA does the compute intensive portion of FM reception and works with the CPU to generate the desired output. To encourage students to implement applications such as an FM receiver on the PYNQ board, we have created an assignment, where we list down all the steps involved in the implementation of an FM receiver and also provide partial solutions to challenging parts of the implementation. We believe that this document along with the assignment will benefit the student community!

5 REFERENCES

[1] Xilinx Zynq: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[2] Pynq : http://www.pynq.io/

[3] GNU Radio: https://www.gnuradio.org/

ADDENDUM:
PYNQ-Z1 BOARD ASSIGNMENT

PROBLEM STATEMENT

In this assignment, we will be implementing an FM receiver in GNURadio software. GNURadio software provides various signal processing blocks for applications such as software defined radio. GNURadio allows users to create their own custom blocks to implement a functionality specific to the user's application.

For this assignment, we will be doing the FM demodulation part of FM reception on FPGA by creating our own IP. We will then write a custom block in GNURadio that internally sends the data to the FPGA, and receives the processed data from FPGA. The entire flow of GNURadio runs on the CPU of Pynq board, by default.

The first section of this assignment talks about the initial setup required for the Pynq board to get things up and running. Section 2 is our baseline implementation. In this implementation, we execute the entire flow for FM receiver in the CPU, with the help of existing blocks in GNURadio. Section 3, 4 and 5 explain how FPGA can be programmed to carry out FM demodulation. These sections cover writing high level C code in Vivado HLS software to downloading the bitstream onto the FPGA. The last section is our optimized implementation of FM receiver. In this implementation, FPGA does the heavy lifting in terms of computation involved and works with CPU to perform the functionality of FM reception.

1. INITIAL SETUP

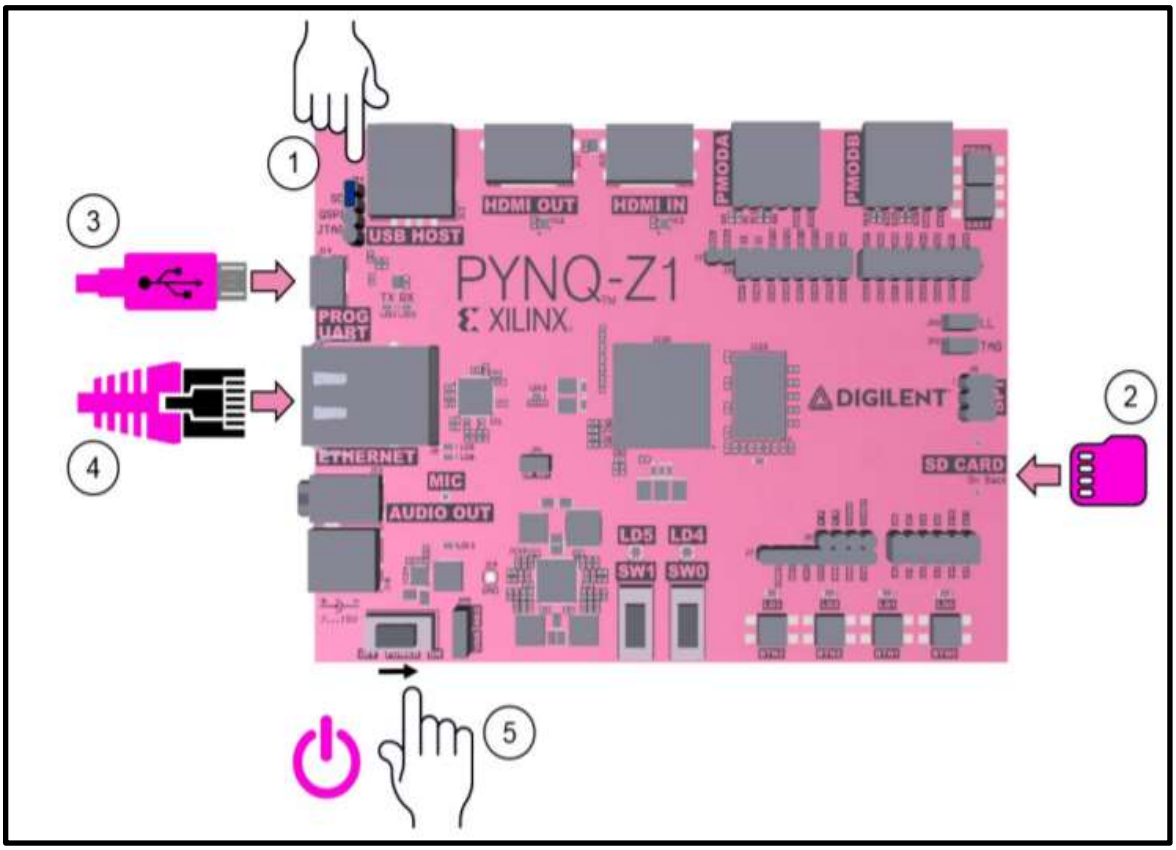


Figure 1: PYNQ Board from Xilinx

1. Set the boot jumper (labelled JP4 on the board) to the SD position by placing the jumper over the top two pins of JP4 as shown in the image. (This sets the board to boot from the Micro-SD card)
2. To power the PYNQ-Z1 board from the micro USB cable, set the power jumper (JP5) to the USB position by placing the jumper over the top two pins of JP5 as shown in the image. (You can also power the board from an external 12V power regulator by setting the jumper to REG.)
3. Insert the Micro SD card loaded with the PYNQ-Z1 image into the board. (The Micro SD slot is underneath the board)
4. Connect the USB cable to your PC/Laptop, and to the PROG/UART (J14) on the board
5. Power on the board. Make sure that all LEDs (LD0-3) are lit up. This means the board has completed booting up.
6. Connect the Ethernet cable into your board and see the steps below for connecting to a computer or network.

1.1 ETHERNET CONNECTION TO THE BOARD

Ethernet port of the PYNQ-Z1 Ethernet can be connected in the following ways:

- To a router or switch on the same network as your computer
- Directly to an Ethernet port on your computer

1.1.1 CONNECT TO A NETWORK

If you connect to a network with a DHCP server, your board will automatically get an IP address.

1. Connect the board to an Ethernet port on the router/switch (Also, connect your laptop to the same router).
2. You can determine the IP address that is assigned to the board by accessing your router's configuration page (usually found at 192.168.1.1). It would have a table of IP addresses assigned to each of the devices connected to the router.
3. Use a terminal (Linux) to ssh to this IP address. You can also use a software such as PuTTy (if using Windows).
4. Alternatively, browsing to http://pynq:9090 will open the Jupyter environment. You can access a terminal from here.

The default hostname is pynq. If you think that there are other boards on the network, you can check if the pynq hostname is already in use before connecting a new board. One way to check this is by pinging pynq from a command prompt (as shown below):

```
> ping pynq
```

1.1.2 CONNECT DIRECTLY TO YOUR COMPUTER

You will need to have an Ethernet port available on your computer, and you will need to have permissions to configure your network interface. With a direct connection, you will be able to work on the PYNQ board, but unless you can bridge the Ethernet connection to the board to an Internet connection on your computer, your board will not have Internet access. You will be unable to update or load new packages without Internet access. Listed below are the steps to establish a direct connection with the board to your laptop. For Windows:

1. Configure your computer with a Static IP.
    a. Go to Control Panel → Network and Sharing Center → View Network Connections.
    b. Right click on Ethernet → Internet Protocol Version 4. Your final screen should like this:

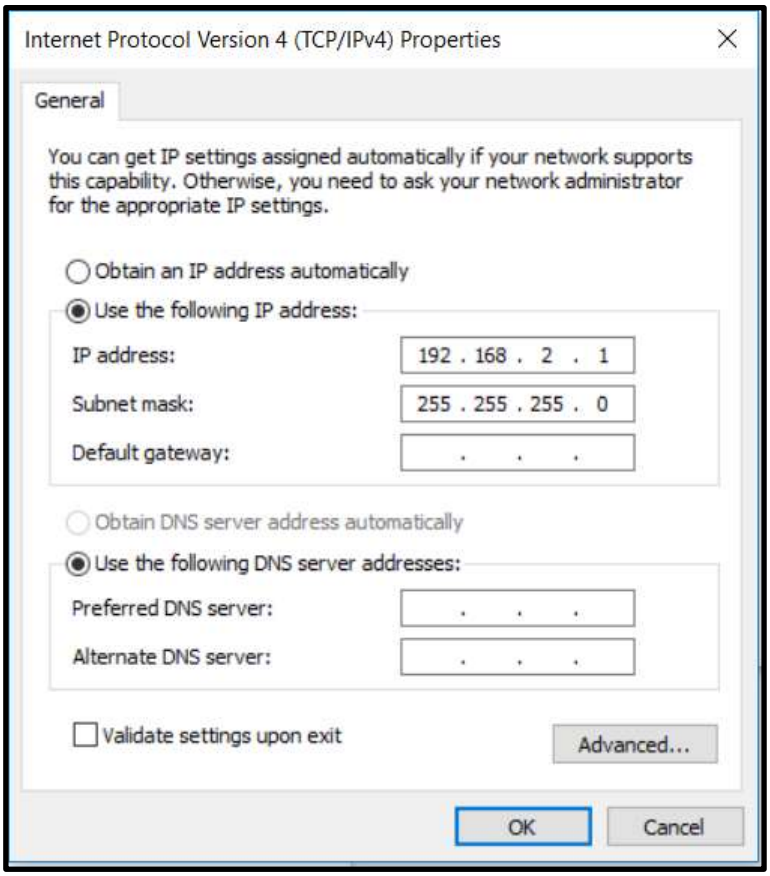


Figure 2: Ethernet properties

c. Click OK to exit from the window.

2. Connect directly to your computer's Ethernet port.
3. Browse to http://192.168.2.99:9090.
4. Alternatively, you can ssh to 192.168.2.99 (Linux) or use a software such as PuTTy to open a terminal.

1.2 CONNECT TO JUPYTER

1. Open a web browser and go to http://pynq:9090 (network) http://192.168.2.99:9090 (direct connection)
2. The Jupyter username is ‘xilinx’ and the password is also ‘xilinx’

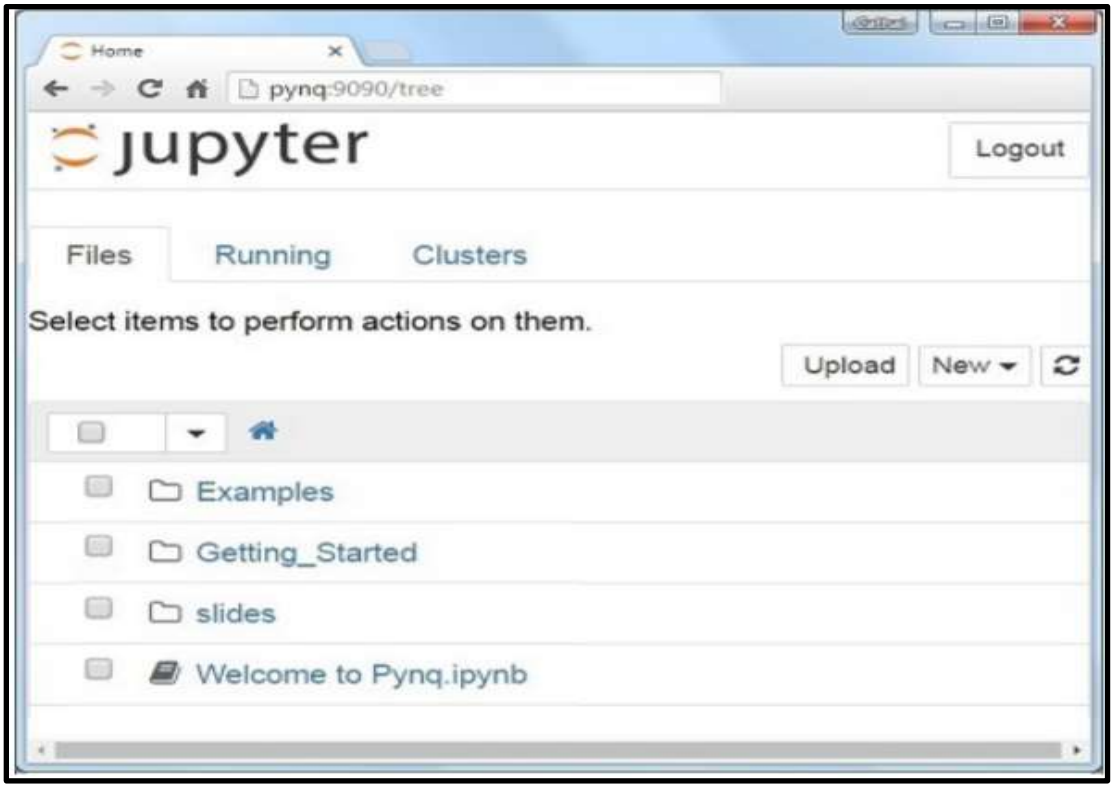


Figure 3: Jupyter environment

3. The default hostname is pynq and the default static IP address is 192.168.2.99. If you changed the hostname or static IP of the board, you will need to change the address you browse to. The first time you connect, it may take a few seconds for your computer to resolve the hostname/IP address.

1.3 CHANGE YOUR HOSTNAME

If you are on a network where other pynq boards may be connected, you should change your hostname immediately.

In the Jupyter portal home area, select **New >> terminal**. This will open a terminal inside the browser as root. Next enter and execute the following command. (Note that you should replace NEW_HOST_NAME with the hostname you want for your board.)

```
sudo /home/xilinx/scripts/hostname.sh NEW_HOST_NAME
```

```
root@pynq:/home/xilinx# ls
                          pynq  REVISION  scripts
root@pynq:/home/xilinx# cd scripts
root@pynq:/home/xilinx/scripts# ls
boot.py  hostname.sh  start_pl_server.py  stop_pl_server.py  update_pynq.sh
root@pynq:/home/xilinx/scripts# ./hostname.sh pynq_cmc
The board needs a restart to update hostname
Please manually reboot board:
sudo shutdown -r now

root@pynq:/home/xilinx/scripts# shutdown -r now
```

Follow the instructions to reboot the board. Note that as you are logged in as root, sudo is not required, but if you are logged in as Xilinx, sudo must be added to these commands.

```
sudo shutdown -r now
```

Figure 4: Change Hostname

2. RUN THE SOFTWARE FLOW FOR GNU RADIO (WITHOUT OFFLOADING) ON PYNQ BOARD

The board is assigned an IP address 192.168.2.99 by default. Use PuTTY to login to the on-board Linux using SSH.

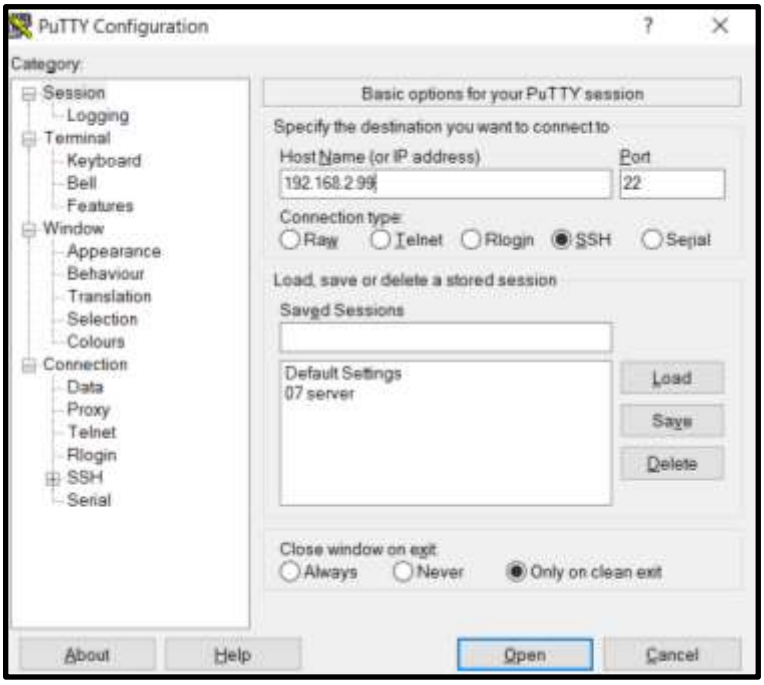


Figure 5: PuTTy Login

The default username and password is **xilinx**.

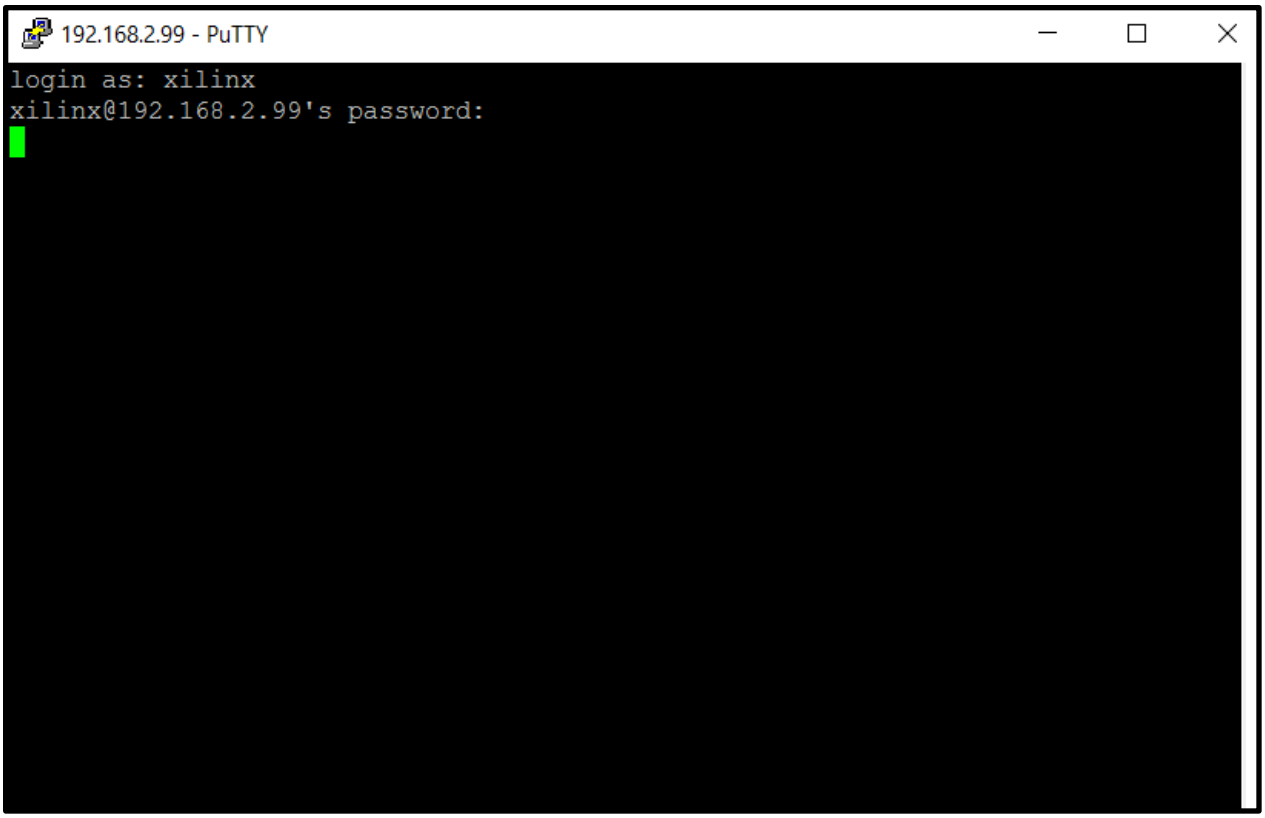


Figure 6: PuTTy Login - 2

2.1 INSTALLING REQUIRED COMPONENTS

1. To install GNURadio, use the following command:

```
sudo apt-get install gnuradio
```

2. To install rtl-sdr (antenna), use the following command:

```
sudo apt-get install gr-osmosdr
```

You can check this link for more details: http://osmocom.org/projects/sdr/wiki/rtl-sdr

3. To test if the drivers for RTL SDR have been installed correctly, connect the antenna to the USB port of the board and run :

```
sudo rtl_test
```

If rtl_test gives an error saying:

```
Kernel driver is active, or device is claimed by second instance of
librtlsdr.
```

Then this means another program is using the device so we are not able to use it. We need to blacklist the kernel from accessing the device.

> Go to /etc/modprobe.d
> Create a file "blacklist-rtl.conf" with sudo permissions
> Add the following lines in the file:

blacklist dvb_usb_rtl28xxu
blacklist rtl2832
blacklist rtl2830

> Save file and exit. Restart your laptop once so that if the kernel is still using the device, it will stop doing so.

4. After restarting, run

```
sudo rtl_test -t
```

5. To test the FM functionality, run:

```
rtl_fm -f 96.3e6 -M wbfm -s 200000 -r 48000 - | aplay -r 48k -f S16_LE
```

You should be able to hear the radio playing now!

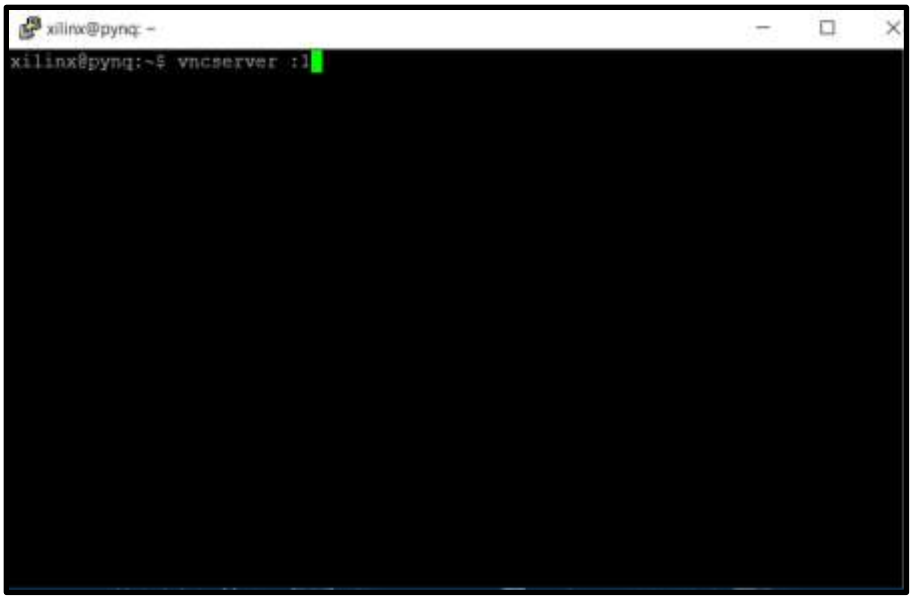
To be able to view the GNU Radio GUI, we would require a VNC server. For this, we will install a VNC server software on the Pynq board and a client software on our laptop.

6. To install VNC server on the board, run:

```
sudo apt-get install tightvncserver
```

7. Install a VNC client such as “UltaVNC Viewer” (for Windows) on your laptop.

8. Start VNC server on the PYNQ board in order to be able to view the GUI as follows:



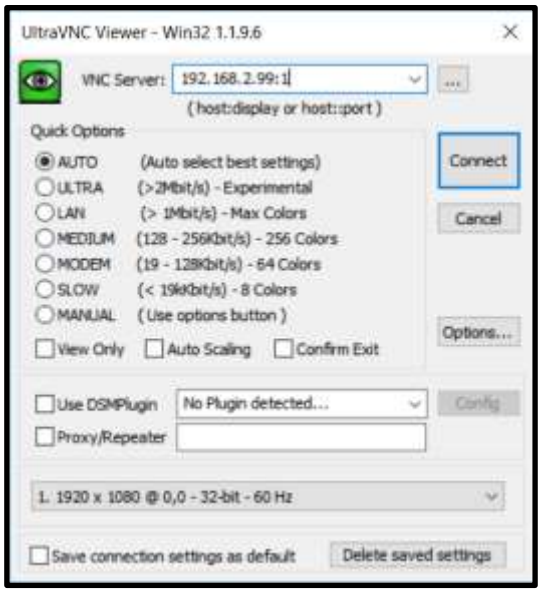


Figure 7 & 8: Starting VNC server on the board and VNC client on laptop

9. Once connected, start the GNURadio Companion from the VNC client's command line using the command **gnuradio-companion**.

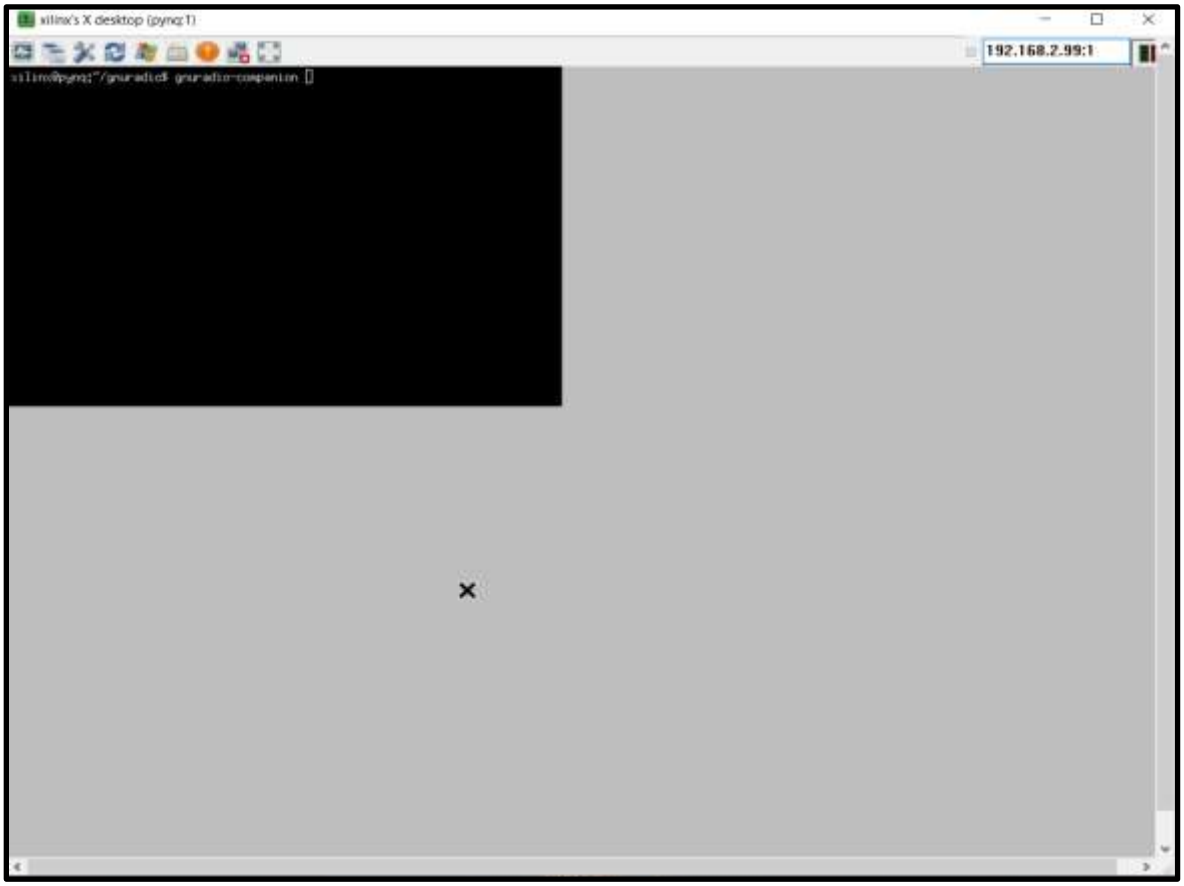


Figure 9: VNC session

You can check this link to implement FM radio on GNU radio.

https://bitbucket.org/akhodamoradiUCSD/237c_data_files/downloads/Setting%20up%20WBFM%20radio%20blocks.pdf

10. Replace the **Audio Sink** block with **Wav File Sink** block, since we do not have audio driver in the CPU of Pynq. Due to this, we will not be able to play Audio directly. We will store the output in a .wav file and then listen to the audio after transferring the file to our laptop using scp.

The final flow graph should look like this. Notice the **Wav File Sink** block at the end :

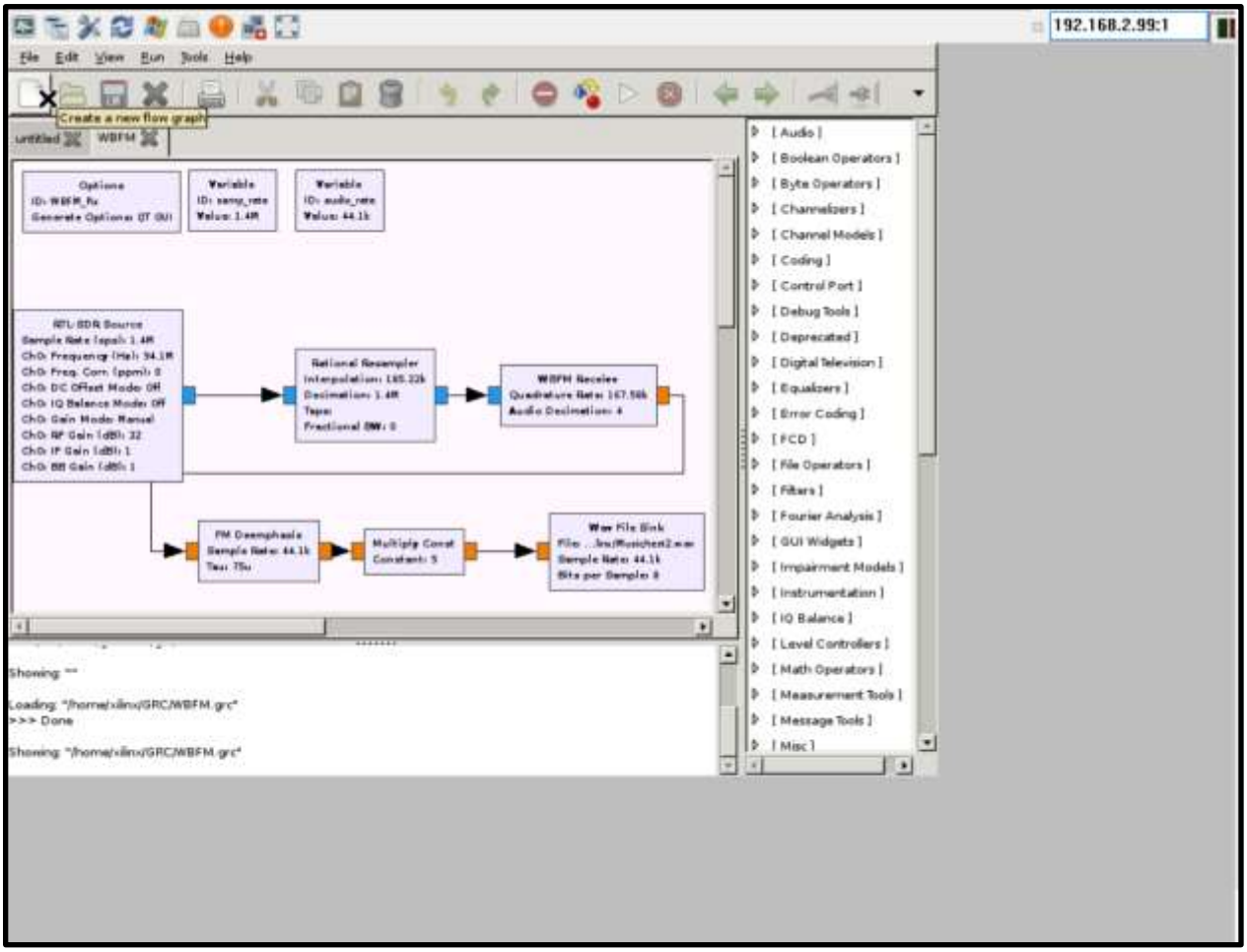


Figure 10: Flow Graph for FM receiver application

3. CODE THE BLOCK TO BE OFFLOADED (WBFM) ON VIVADO HLS

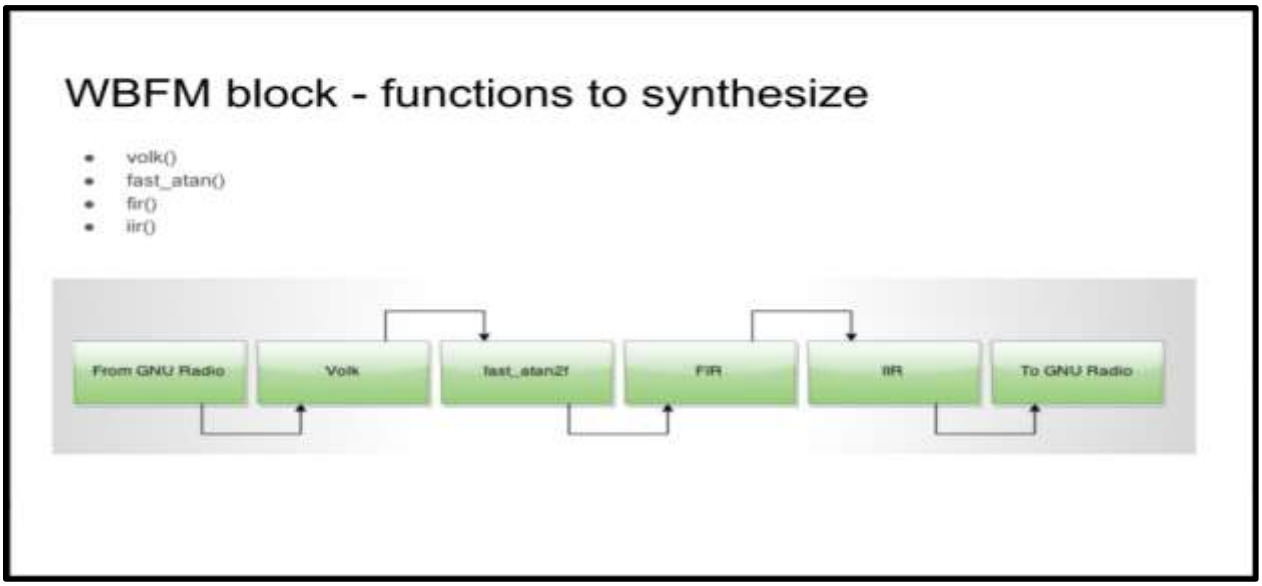


Figure 11: Sub-blocks of WBFM block

We will offload the “WBFM receive” block to the FPGA. For this, we need to achieve the FM demodulation functionality in FPGA. FM demodulation in GNU Radio is done using 4 blocks, namely, conjugate-multiply (here, called “Volk”), fast_atan2f (tan inverse), FIR (low pass filtering) and IIR filter (de-emphasis). We will need to code these blocks and join them together to create a WBFM IP block in Vivado HLS.

You can start with the skeleton code provided here: https://github.com/harveenk/PynqRadio

1. Create a new project called “WBFM_project” and code the four blocks of WBFM in Vivado HLS. Skeleton is provided in the folder “HLS”. Four blocks are: Volk, atan, FIR and IIR.
2. These blocks should pass their individual test-benches.
3. Integrate the four blocks together by entering code in the file wbfm.cpp in folder “Demo”. The final top level function should be tested against the input and output obtained separately from the WBFM Receive block in GNU Radio. The corresponding input and output are recorded using the file sink option in GNU Radio.

4. CREATING WBFM IP

Now we will synthesize the code and package it into an IP.

1. The top-level function to be synthesized is WBFM_accel, contained in file wrapped_wbfm.cpp. Purpose of this file is to connect the Vivado HLS WBFM block to the AXI DMA, we need to change the existing WBFM code with Vivado HLS, and add some additional functions for synthesis. In particular, pop_stream and push_stream are functions to extract and insert elements from/into an AXI4 interface.
2. Change the name of the top function to **WBFM_accel** in the project settings.
3. Run synthesis. Solution → Run C Synthesis → Active Solution.
4. Export RTL as an IP in IP-XACT. Go to Solution → Export RTL.

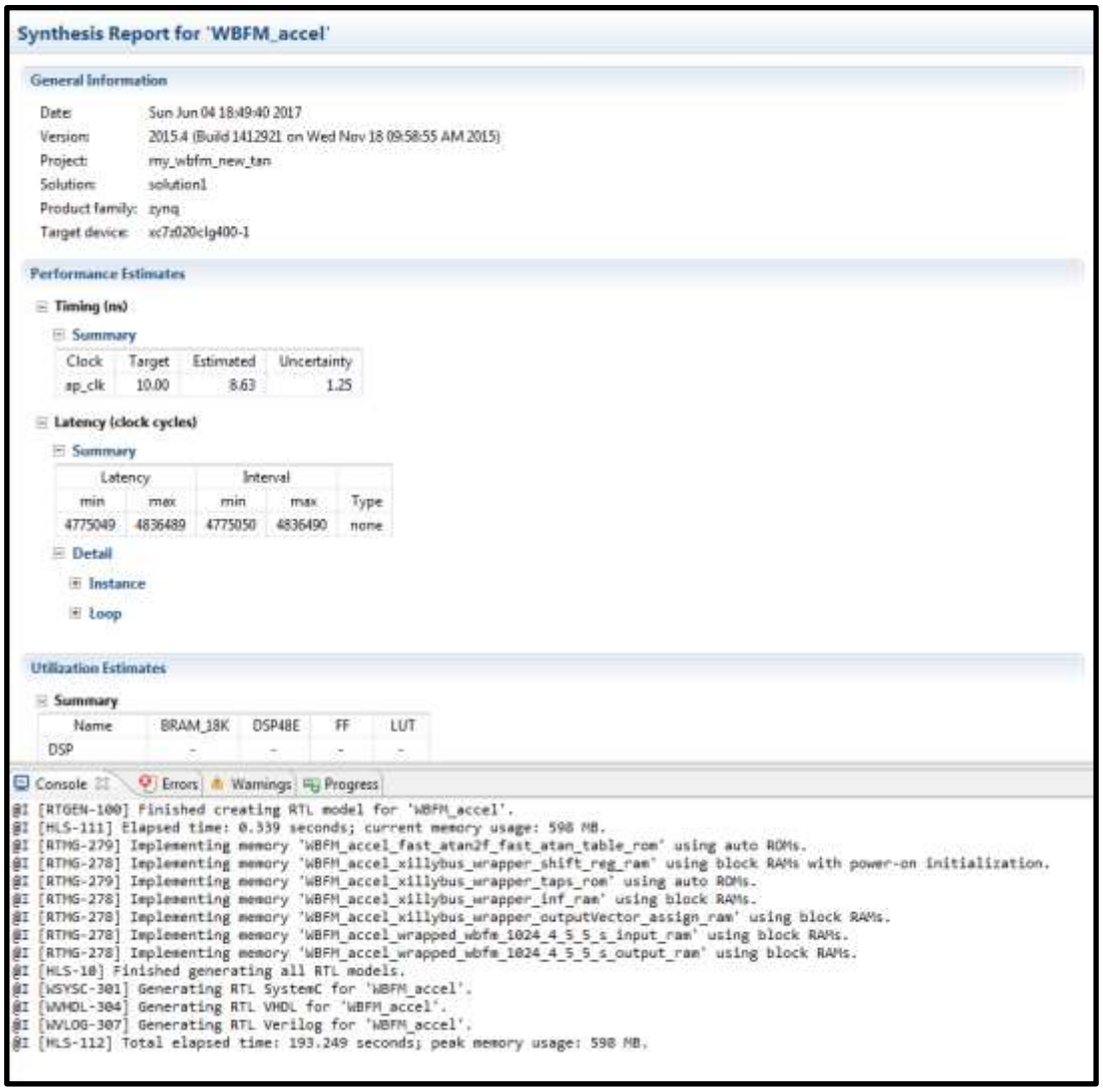


Figure 12: Generated report after successful synthesis

5. GENERATING BITSTREAM

Now, we will integrate the WBFM IP that we have created so that the ARM CPU will be able to communicate with our IP. We will use Xilinx Vivado software for this. We will create a block diagram, include our IP and DMA blocks in it and finally generate a bitstream.

Note: Use **Vivado 2016.1** to avoid compatibility issues!

1. Launch **Vivado 2016.1**. Create a new project in your working directory and select:
   - RTL Project
   - No Sources, IP, or constraints (for now)

2. In the **Default Part,** select **Board → xc7z020clg400-1**
3. Click next and then finish. Vivado GUI opens with an empty project.

Next we need to add our custom IP to the list of IPs in Vivado.

4. Click on “IP Catalog” under Project Manager in the Flow Navigator.
5. Right click in the IP Catalog pane and go to “Add Repository”
6. Add the “WBFM_project” folder. This should add the IP under “User Repository” in IP Catalog.

We will now generate a block diagram. Download the *wbfm_base.tcl* script from the Github repository.
7. Enter following command in Vivado Tcl console:

```
source <path_of_wbfm_base.tcl>
```

8.This will generate the following block diagram:

:

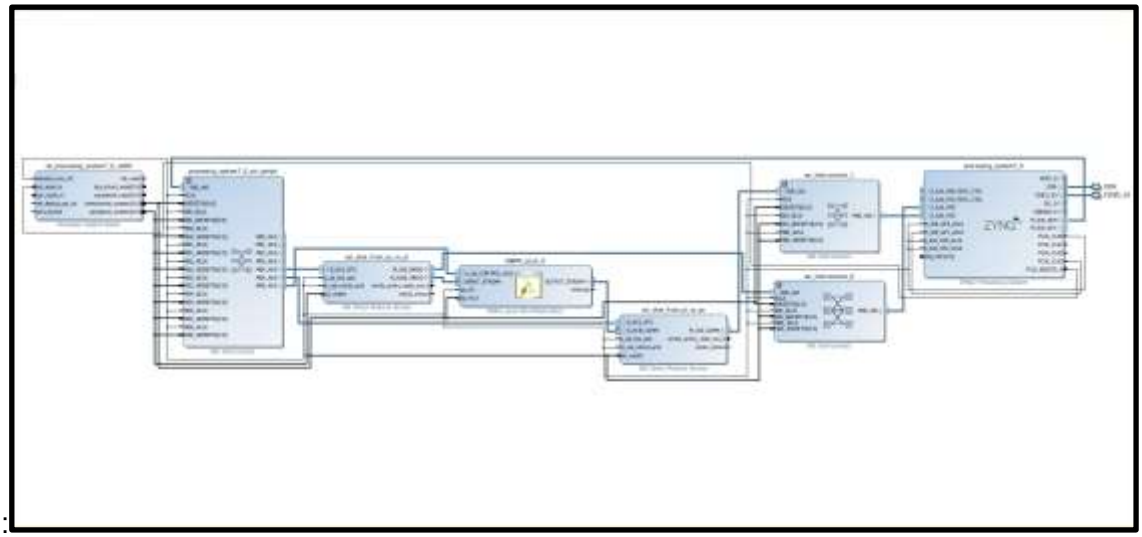


Figure 13: Block design with our custom IP

The block diagram should contain the WBFM IP and 2 AXI DMA blocks.

8. Next, we need to create a wrapper for the block diagram. This will be the top level block to be synthesized. Right click on the block diagram and select Create HDL wrapper as shown:

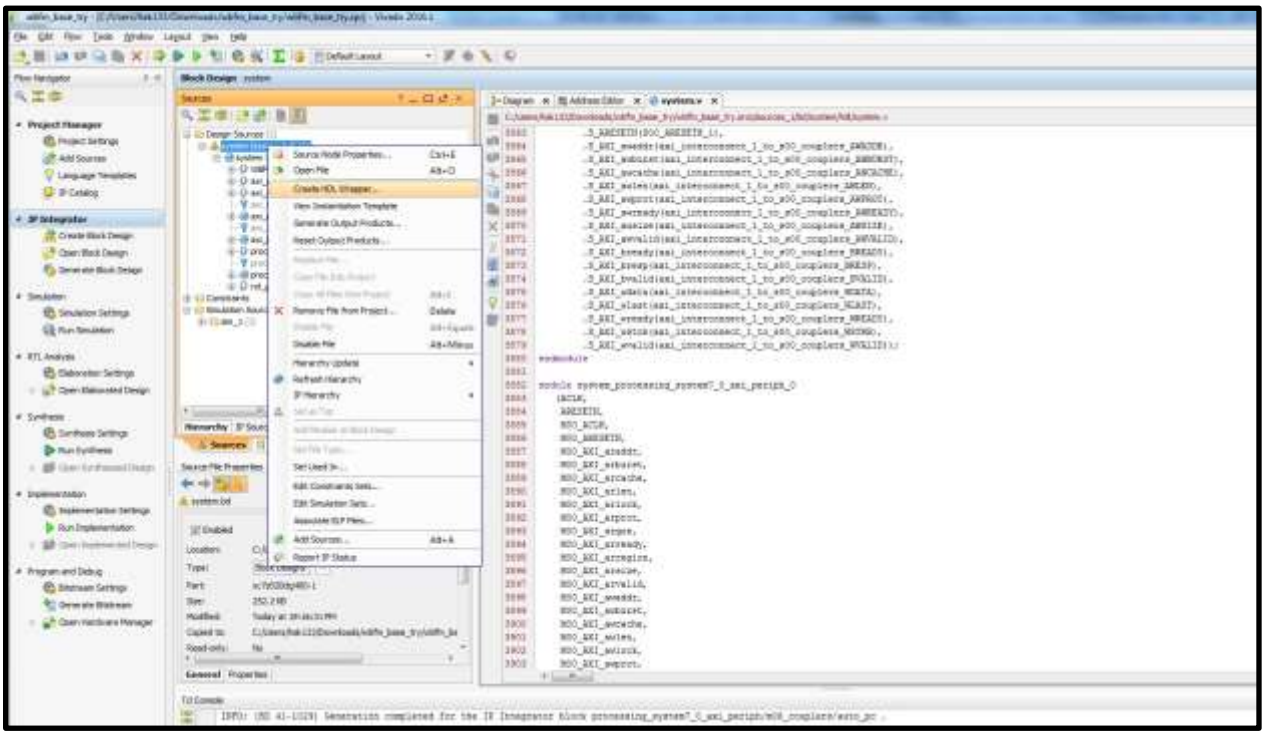


Figure 14: Create HDL Wrapper

9. Generate bitstream by going to Flow → Generate bitstream.
Note: Ensure that the name of the bitstream is the same as the tcl file. You can also generate a new tcl file by executing: `write_bd_tcl <tcl_filename>` in Vivado Tcl Console.

6. RUNNING GNURADIO FLOW WITH CPU-FPGA INTERFACE

Firstly, we need to program the FPGA with the bitstream generated.

1. Copy the bitstream and corresponding Tcl script (Make sure they have same name) to the folder /home/xilinx/pynq/bitstream on the Pynq board using scp command.
2. Create Overlay object using the bitstream and download the bitstream to the board.

Important: In order to work with float values, we need to make a small change to the DMA driver in Pynq.

3. In the file /home/xilinx/pynq/drivers/dma.py, change Line 463 to the following:

```
return ffi.cast("float *",self.buf)
```

Next, we need to add a custom GNURadio block in order to interface FPGA and ARM processor in Pynq. First, install all the dependencies:

4. Install pip for Python 2.7 -

```
curl -O https://bootstrap.pypa.io/get-pip.py
python get-pip.py
```

5. Install cffi for Python 2.7 -

```
sudo python -m pip install cffi
```

6. Install gnuradio dev tools to be able to compile the custom block:

```
sudo apt-get install gnuradio-dev
```

7. Install dependencies required to compile the custom block (**Note**: it can take very long!)

```
> wget -O boost_1_55_0.tar.gz
http://sourceforge.net/projects/boost/files/boost/1.55.0/boost_1_55_0.
tar.gz/download
> tar xzvf boost_1_55_0.tar.gz
> cd boost_1_55_0/

> ./bootstrap.sh --prefix=/usr/local     #bootstrap setup

> ./b2                                   #build

> sudo ./b2 install                      #install
```

Next, we will to add a custom block from GNURadio GUI in order to interface FPGA and ARM processor in Pynq.

You can check the steps given the Wiki page below to generate a custom block in GNURadio for interfacing FPGA and ARM processor (using DMA transfer).

https://wiki.gnuradio.org/index.php/Guided_Tutorial_GNU_Radio_in_Python#3.2.1._Using_gr_modtool

The Python code for this block is provided in our Github repository (gr-fpga-interface folder).

Execute the following commands to make your own custom block:

8. First, we will create a new module with:

```
gr_modtool newmod fpga-interface
```

9. Next, we add our block to the existing set of blocks in GNURadio with the following command:

```
gr_modtool add -t decimator -l python
```

10. Copy grc and Python subfolders from our Github repository.

11. Create a build directory in the block's main folder and cd to this directory. Now, build your block with following commands:

```
Cmake ../
make
sudo make install
sudo ldconfig
```

12. Start VNC server on the board and open a new session with VNC client. Open GNURadio GUI. You should see a new category called fpga-interface in the block list. Check that the block is correctly generated.

13. Create the following flowgraph:

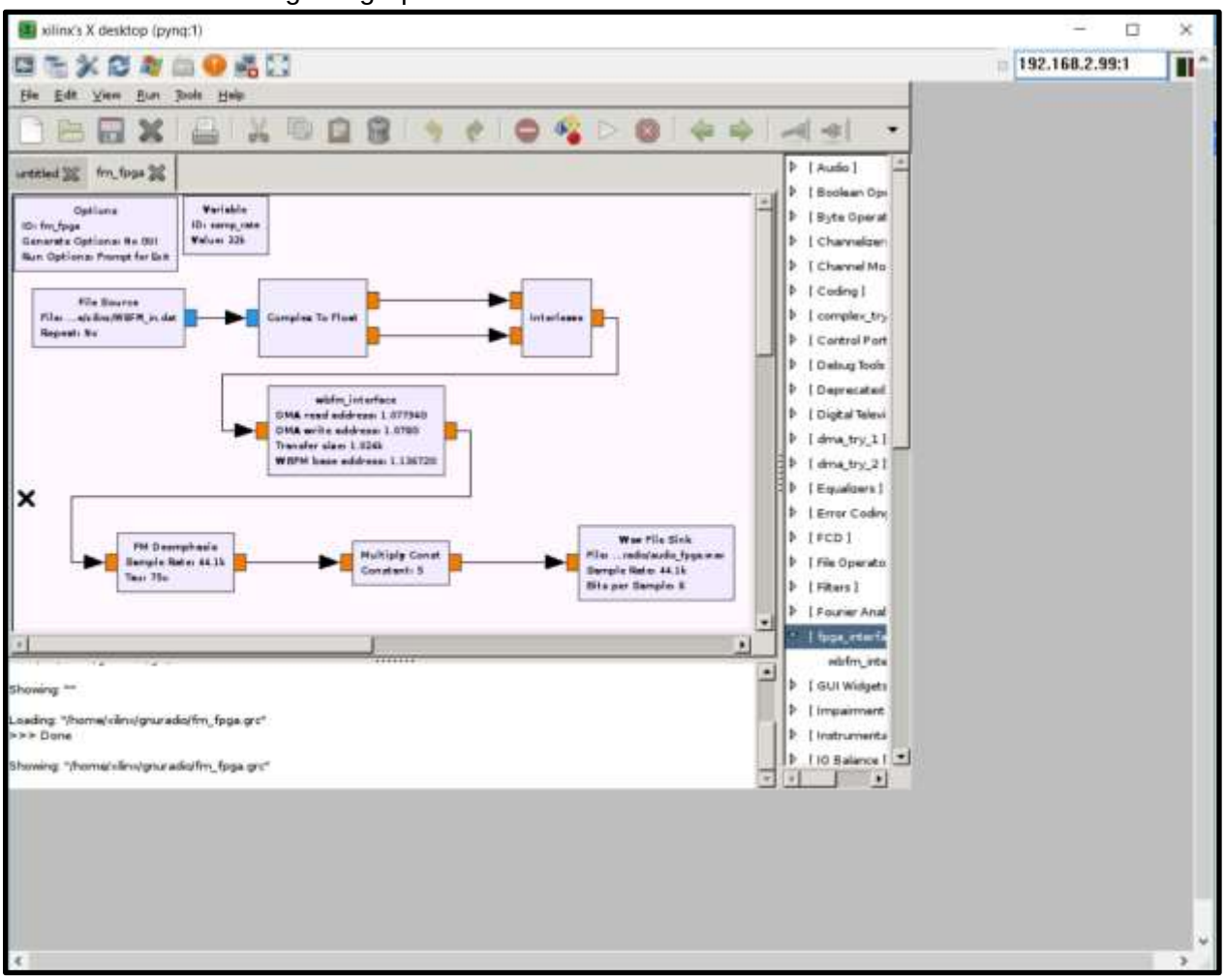


Figure 15: Flow graph with custom wbfm_interface block

14. Set the parameters of the wbfm_interface as follows:

DMA read address: 1077936128
DMA write address: 1078001664
Transfer size: 1024
WBFM base address: 1136721920

Note that the input file to this flow graph is a binary file names WBFM_in.dat. Download this file from the Github repository.

15. Finally, you can execute this flow graph from the terminal with the command below and listen to the audio generated!

```
sudo python fm_fpga.py
```