SOFTWARE DEFINED RADIO (SDR)-RF IMPLEMENTATION WITH THE DIGITAL IMPLEMENTATION

By

Chaymaa Ossama Mohamed

Heba Magdy El-Gohary

Heba Mahmoud Yassin

Khadija Khalid Ali

Mahetab Ossama Mohamed

Yara Hossam El-Deen Mahmoud

Under the Supervision of

Associate Prof. Hassan Mostafa

Eng. Sherif Hosney

A Graduation Project Report Submitted to

The Faculty of Engineering at Cairo University

In Partial Fulfillment of the Requirements for

The Degree of Bachelor of Science in

Electronics and Communications Engineering

JULY 15, 2018

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

CAIRO, EGYPT

Table of Contents:

Table of Figures:

List of Symbols and Abbreviations

1G	first generation
2G	second-generation
3GPP	3rd generation partnership project
ACP	Accelerator Coherency Port
ADC	Analog to Digital Converter
ADT	Application Development Tool
AMP	Asymmetric Multi-Processing
AMBA	Advanced Microcontroller Bus Architecture
AP SoC	All Programmable SoC
API	Application Programmable Interface
ARM	Advanced RISC Machines
ASIC	Application Specific Integrated Circuit
AXI	Advanced Extensible Interface
BIF	Boot Image Format
BPSK	Binary Phase Shift Keying
BSP	Board support package
CLBs	Configurable Logic Blocks
CP	cyclic prefix
CPU	Central Processing Unit

CR	Cognitive Radio
CRC	Cyclic Redundancy Check
DAC	Digital to Analog Converter
DCM	Digital Clock Management
DCP	Design Check Point
DDR	Double Data Rate
Deb	Debian
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
DPC	Dirty Paper Coding
DPCCH	Dedicated Physical Control Channel
DPCCH2	Dedicated Physical Control Channel 2
DPDCH	Dedicated Physical Data Channel
DSP	Digital Signal Processing
DTB	Device Tree Blob
DTS	Device Tree Source
DUT	Device Under Test
EDGE	Enhanced Data rates for GSM Evolution
E-DPCCH	E-DCH Dedicated Physical Control Channel
E-DPDCH	E-DCH Dedicated Physical Data Channel
EGLIBC	Embedded GNU C Library

eSDK	Extensible Software Development Kit
ETSI	European Telecommunications Standards Institute
FBI	Feed-Back Information
FCC	Federal Communications Commission
FF	Flip Flops
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite state machine
FSBL	First Stage Boot Loader
GCC	GNU Compiler Collection
GP	General Purpose
GPP	General Purpose Processor
GPRS	General Packet Radio Services
GRC	GNU Radio Companion
GSM	Global System for Mobile
GUI	Graphical User Interface
HDF	Hardware Description File
HDL	Hardware Description Language
HF	High Performance

HS-DPCCH	Dedicated Control Channel with HS-DSCH transmission
HSPA	High Speed Packet Access
IA	Intelligent Antenna
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
ICD	Interrupt Control Distributer
IDE	Integrated Development Environments
IEEE	Institute of Electrical and Electronics Engineers
IFFT	Inverse Fast Fourier Transform
IP	Intellectual Property
IPK	Itsy Package
ISI	Inter Symbol Interference
JTAG	Joint Test Action Group
LNA	Low Noise Amplifier
LTE	Long Term Evolution
LUTs	Look up Tables
MAC	Medium Access Control
MFIFO	Multi-channel First In First Out
MM2S	Memory Mapped to Streaming
MMU	Memory Management Unit

MPDUs	MAC Protocol Data Units
MSCS	Multi-Standard Communication System
NBPSC	Number of Coded Bits Per Subcarrier
OCM	On Chip Memory
OE	Open Embedded
OFDM	Orthogonal Frequency Division Multiplexing
PA	Power Amplifiers
PCAP	Processor Configuration Access Port
PCCC	Parallel Concatenated Convolutional Code
PDR	Partial Dynamic Reconfiguration
PHY	Physical Layer
PL	Programmable Logic
PLB	Programmable Logic Blocks
PLCP	Physical Layer Convergence Protocol
PLL	Phase Locked Loop
PMD	Physical Medium Dependent
PPDU	PLCP Protocol Data Unit
PPI	Private Peripheral Interrupt
PRC	Partial Reconfiguration Controller
PS	Processing System
PSDU	Physical layer Service Data Unit

QAM	Quadrature Amplitude Modulation
QA	Quality Assurance
QEMU	Quick Emulator
QoS	Quality of Service
QPSK	Quad Phase Shift Key
RF	Radio Frequency
RP	Reconfigurable Partition
RPM	Red Hat Package Manager
S2MM	Streaming to Memory Mapped
SCU	Snoop Control Unit
SC-FDMA	Single-Carrier Frequency Division Multiple Access
SCM	Source Code Management
S-DPCCH	Secondary Dedicated Physical Control Channel
SD-Card	Secure Digital Card
SDK	Software Development Kit
SDR	Software Defined Radio
SF	Spreading Factor
SMP	Symmetric Multi-Processing
SPDX	Software Package Data Exchange
SPI	Serial Peripheral Interface

SoC	System on Chip
TFCI	Transport-Format Combination Indicator
TPC	Transmit Power Control
TrCH	Transport Channel
TTI	Transmission Time Interval
UART	Universal Asynchronous Receiver-Transmitter
UHD	USRP Hardware Driver
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VIO	Virtual Input Output
WCDMA	Wide Code Division Multiple Access
WLAN	Wireless Local Area Network
XSCT	Xilinx Software Command Line Tool

Acknowledgement

This year's work wouldn't have come to fruition if it were not for, after God, some pretty amazing people whom we got know this year. Without their help and support we wouldn't have been where we are today.

First, we would like to thank Dr. Hassan Mostafa for administering our work, providing us with the idea, the necessary guidance, the kits and a suitable working environment. We would also like to thank him for training us throughout the year for this day and keeping up with our work and encouraging team work and the collaboration between the teams to achieve the best results possible.

Second, we would like to thank Eng. Sherif Hosny for his guidance and time throughout the year and always responding to our questions regarding the last years' work. Also, Eng. Mostafa Gamal for meeting with us whenever possible and explaining the last year's work to us and putting us on the start of the road that took us where we are today.

Third, Eng. Ossama Ryad who was a great help with the USRP. Thank you for your help, your time and your patience in teaching us.

Fourth, we would like to thank Eng. Mohammed Osama for his immense help on the last leg of this journey and spending all his free time and vacation time with us trying to get the best results possible. Thank you for your time, effort and patience.

Last but not least, we would like to thank the previous years teams who have worked on this project and provided the great base on which we built our project.

Abstract

This year's part of the Software Defined Radio (SDR) graduation project discusses the implementation of communication chains via real channel for multi-standards on a partially reconfigurable heterogeneous platform as ZYNQ board and USRPs. It depends on the ability of the ZYNQ board to control the USRP board to use it as its RF plate form for transmitting and receiving.

Separating the transmitter and receiver blocks is needed in order to operate them on two different ZYNQ boards with two separate USRPs, so that real channel communication may take place. On each ZYNQ board, the two processors of the ZYNQ board are used independently to run two different applications on different platforms. One processor is used to run a bare metal application that runs on the FPGA which represents a single static standard -without DPR-. The other processor is used to run a Linux image which is used to control the USRP, by installing the UHD image.

During the project, experience has been gained in HDL, C/C++, GNU Radio, and Embedded-Linux.

The future work in this project will be integrating the Dynamic Partial Reconfiguration (DPR) of the previous year's project into this year's project using separate ZYNQ boards and real channel communication.

Chapter 1: Introduction

In the last few years, the number of users has increased significantly; a way was needed to handle the communication among them. Different types of standards have been implemented to compromise between area, power, quality and the large number of users. However, since those standards are not used simultaneously, every standard has its own transceiver which causes a big waste in area, power and correspondingly battery life. To solve those problems, SDR was proposed as a solution.

SDR is a way to implement the hardware using mainly software and simple hardware resources in order to decrease the hardware and use area in an efficient way. It is a way to define the physical layer functions in software.

In this thesis we are going to discuss the interface between the FPGA and USRP to have a real channel between transmitter and receiver, testing different communication standard behavior through the wireless channel.

1.1. Organization of the thesis

This dissertation will go as follows; First, there is a brief introduction to the project, its purpose, the concepts it follows and its progress over the years. Second, this year's project is outlined and an introduction to the work needed to execute it is written. Third, the hardware and software tools and kits needed are introduced in detail, their benefits discussed and the purpose behind them elaborated. Fourth, a brief introduction to the main tool, ZYNQ board, its uses and main functions. Fifth, the integration between the USRP and ZYNQ board is discussed in detail and the steps taken to interface between them are explained. Sixth comes the chapter on how to operate two processors of the same board independently in order to operate both bare-metal codes and OS on the same board without interfering with each other, thus achieving the purpose of chapter five. Seventh, the conclusion of our work and the results we've achieved is depicted in this chapter along with the future work this project will need, in order to reach the market as a final working project that will save a lot of power, money and time.

1.2. Software Defined Radio (SDR)

1.2.1. Introduction

The communication standards are being developed and upgraded to satisfy the speed and the time to handle connectivity among the users, whose number is increasing with time. Consequently, different communication standards have been developed, but the radio frequency spectrum is not utilized in an efficient way [1, 2], as the communication bands are not being used simultaneously. The research in the radio spectrum utilization leads to two approaches to solve this problem, the first approach is the Intelligent Antenna (IA), which is an antenna array technology that uses spatial beam forming and signal processing algorithms to cancel interference and reuse of the space resources [3]. IA depends on the Dirty Paper Coding (DPC) technique. The second approach is the Cognitive Radio (CR), which dynamically configures the user terminals, to utilize the radio spectrum that is not used, depending on the available wireless channels detected without interfering with the other users. In other words, CR is considered a way of managing the radio spectrum in an efficient way and it can be developed using the SDR technique [4, 5].

Generally, in a Multi-Standard Communication System (MSCS) two major problems exist, the utilization of the radio spectrum pointed to in the previous paragraph and utilization of hardware. As each standard has its own transceiver this leads to high cost, large area, high power consumption and low battery life. At the same time, the development of the central base stations and the users' devices change tremendously to adapt to the new technologies and support the old ones. Developing hardware, upgrading and redistributing cost money and effort. These two major problems, unutilized radio spectrum and waste in hardware, initiated the research for finding new ways of reusing (reconfiguring) the same set of hardware to operate the old and new technologies. The utilization of the radio resources and the physical hardware resources can be done by off-loading the data transmitted between the different communication systems, and at the same time reconfiguring the hardware resources or reordering them to switch from one standard to another. SDR is a type of radio system implementation using software, which is used to form different waveforms. These waveforms allow the system to switch among different communication standards.

The motivation of SDR came from the existence of some physical layer blocks, which have the same functionality in different communication systems like (GSM, UMTS, LTE, etc.…). Note that, these standards are not used at the same time which allows their hardware resources and radio spectrum resources to be shared among the standards and hence, be used in a more efficient way. Also, the switching among the different waveforms should be dynamic, more or less in real time. The Partial Dynamic Reconfiguration (PDR) is a technique used in the (FPGA), which allows hardware real time reconfigurable computing. By using the PDR's capability of dynamically changing and partially configured system files, real time SDR system can be implemented.

1.2.2. Communication System

Figure 1-1 shows the main blocks in a modern communication system. It is composed of a Digital Signal Processing (DSP) unit, digital and analog converters (DAC, ADC), RF front end and antenna.

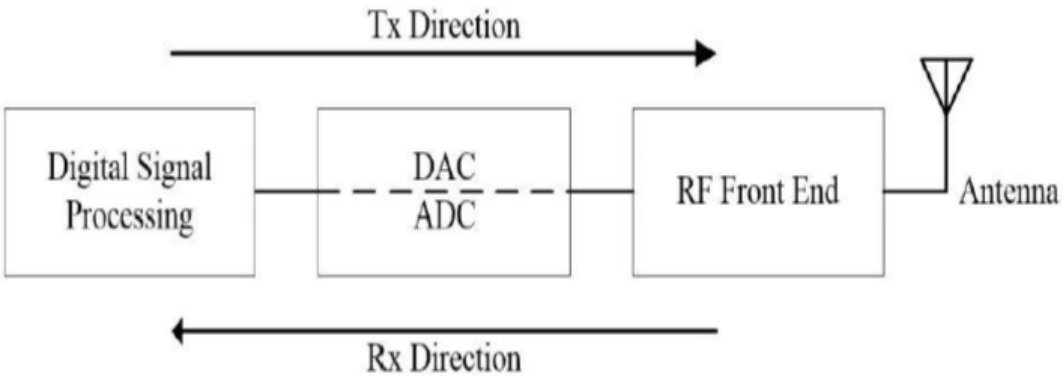

Figure 1-1 Simple communication system

High data rates can be achieved by processing the communication signals digitally using software, which is easier to develop, distribute and upgrade, the digital transceivers penetrate the traditional analog transceivers by pushing the digital and analog converters towards the antenna and pulling the communication systems more to software design on a given hardware.

This work will concentrate on the DSP block, however a brief description for each block is presented as follows:

- **Digital Signal Processing Block**: In the transmitter, this block is responsible for signal adaptation to be sent over a channel. Signal adaptation includes encryption, error correction coding schemes, modulation ...etc. Whereas, in the receiver this block is responsible for extracting the original information sent by the transmitter, by reconstructing the signal using demodulation, decoding and decryption. This block increases the flexibility of the radio development.
- **DAC/ADC Blocks:** Analog and digital converters used to transfer the signal between the analog domain and digital domain. Using ADC, the received signal is digitalized to be processed digitally using the DSP block. The digital representation depends on the sampling rate, which may lead to some information loss. On the other hand, the DAC can reconstruct the signal to nearly the original one.
- **RF Front End Block:** It is the classical block that contains the Low Noise Amplifier (LNA), filters and the Power Amplifier (PA). This block is the most challenging block in the SDR development.
- **Antenna:** Generally, the antenna is a passive device used to capture the electromagnetic waves from the surrounding media and converts it to an electrical signal. The antenna design complexity varies from a single antenna to multiple antenna arrays. The smart antenna is an antenna array that uses the signal processing algorithms to locate the direction of signal arrival and the reconfigurable antenna is capable of changing its frequency for adaptable systems.

1.2.3. SDR Definition

The daily usage of communication standards is increasing. Phone calls, accessing the internet, sharing data and controlling devices are examples of modern communication usage. The devices using these standards vary in shape, functionality and the way of usage like cellphones, wireless routers, smart chips, smart metering ...etc. Although it is not easy to invent a generic device that can do everything, but it is achievable by using an adaptable communication device that can manage the communication between them all.

This adaptation is easier to be done through software defined modules, where these modules can change functionality by using the software.

SDR is a way to implement the hardware using mainly software and simple hardware resources in order to decrease hardware and use area in an efficient way. It is a way to define the physical layer functions in software.

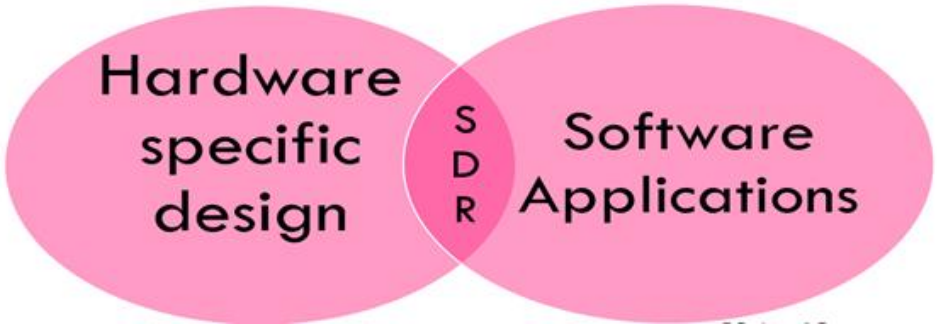

Figure 1-2 SDR approach

1.2.4. SDR Advantages

There are many benefits for using SDR that it can be used in different industries and applications, hereby listing some of these advantages:

1. **Adaptability**

   The waveform is adapted to handle different scenarios for Radio. For example, if the Radio has low battery the waveform will be low powered or if it's downloading a file the waveform will have a high throughput and so on...

2. **Interoperability**

   As some radios are incompatible with each other, SDR handles the communication between them.

3. **Frequency Reuse (cognitive Radio)**

   If there is an assigned band for a user and it is unused for a certain time, SDR can use it (borrow it) to increase its available band.

4. **Updating and Remote Upgrading**

   As we use the same hardware for different standards and the only change is in the software, we can easily update those standards and need no extra or new hardware or resources, so there is no need to go back to lab

5. **Less Area and cost**

   The hardware is decreased. Hence, area, power and cost are decreased.

6. Ability to receive and transmit various modulation methods using a common set of hardware
7. The ability to alter functionality by downloading and running new software at will.
8. The possibility of adaptively choosing an operating frequency and a mode best suited for prevailing conditions.
9. Elimination of analog hardware and its cost, resulting in simplification of radio architectures and improved performance.

1.2.5. SDR Implementation

SDR can be implemented on different hardware platforms such as General Purpose Processor (GPP), Digital Signal Processor (DSP), and Field Programmable Gate Array (FPGA). FPGA is suitable for high rate applications that have to be low in power and resources. Also, the high flexibility of the FPGA achieved by Dynamic Partial Reconfiguration (DPR) allows it to be used in hardware implementation of SDR as explained later.

1.3. Field Programmable Gate Array (FPGA)

FPGA is a programmable device that can be configured by the user for any desired application. It consists of the following blocks as shown in the Figure 1-3:

1. **Configurable Logic Blocks (CLBs):** include Look Up Tables (LUTs) and registers to implement combinational and sequential logic.
2. **Dedicated blocks:** such as DSP and RAM blocks.

3. **Input and output blocks:** are special logic blocks for external connectivity.
4. **Routing:** connects inputs and outputs to CLBs or generally any source to any destination by connecting wires through switching matrices.
5. **Clocking resources:** like digital clock manger (DCM). It is used to eliminate clock skew and to synthesize the desired clock frequency or shift phase.

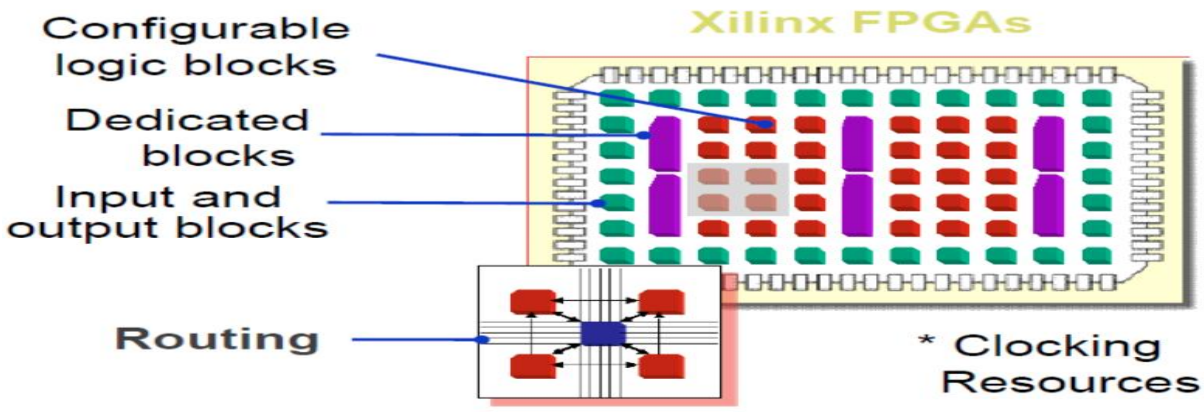

Figure 1-3 FPGA blocks

1.3.1. FPGA configuration

FPGA is a volatile device, which means that its contents are erased once the power is turned off or interrupted so we need to configure it each time we need to use it.

FPGA is considered a two-layered device as shown in the following Figure 1-4:

1. Logic layer contains the logic to be configured as the desired application.
2. Configuration memory layer contains the configuration file used to configure the logic layer.

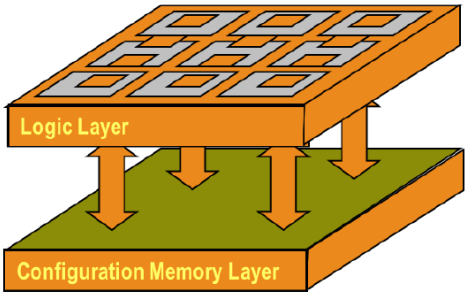

Figure 1-4 FPGA layers

1.3.2. Types of configuration

1.3.2.1. Full/Fixed Reconfiguration

As shown in Figure 1-5, a configuration file is downloaded to configure the whole chip. FPGA has to stop working during downloading the new configuration file so, it takes long time to reconfigure the chip each time.

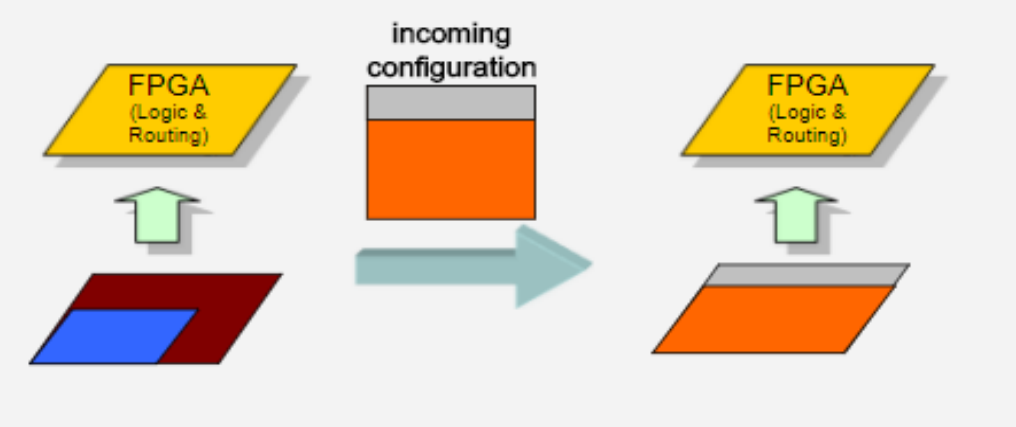

Figure 1-5 Full reconfiguration

1.3.2.2. Static partial reconfiguration

As shown in Figure 1-6, an initial full bit file with complete configuration is downloaded for the whole chip then to reconfigure a part of the FPGA a partial bit file is downloaded while suspending the work of the FPGA. In this type the reconfiguration overhead time is reduced compared with the previous type.

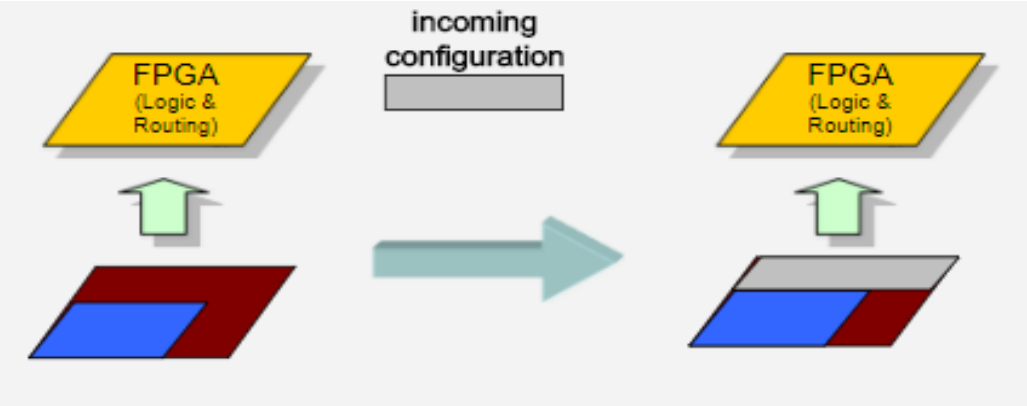

Figure 1-6 Static partial reconfiguration

1.3.2.3. Dynamic partial reconfiguration (DPR)

Like the previous type, an initial full bit file with complete configuration is downloaded at first, as shown in Figure 1-7. The main difference is that a part of the FPGA can be reconfigured at run time by loading a partial bit file to the configuration memory while the FPGA continues its normal operation except for this part so this decreases the reconfiguration time. As shown in the following figure, the FPGA is divided to static and dynamic parts. The static part is configured from the initial full bit file and remains fixed while the dynamic modules in the dynamic part is reconfigured at run time by partial bit files.

In our case, we are concerned with the Dynamic Partial Reconfiguration of the FPGA.

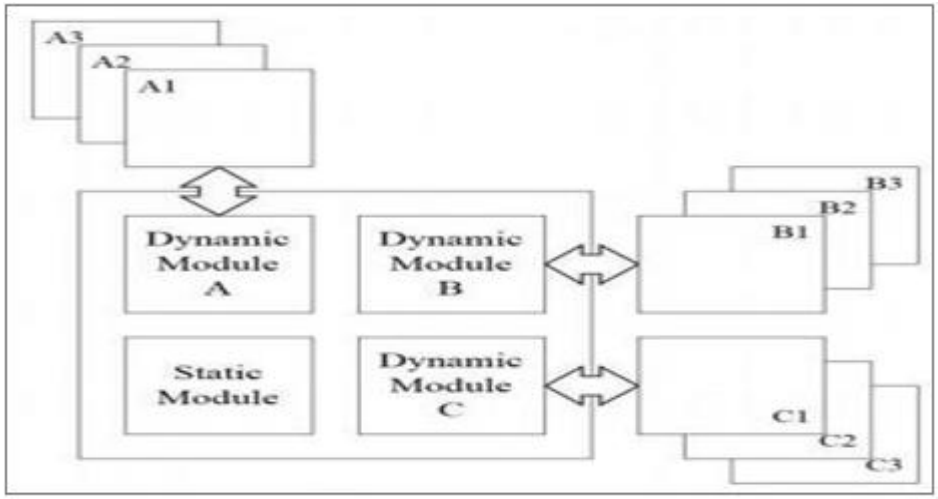

Figure 1-7 Dynamic partial reconfiguration

1.4. Dynamic Partial Reconfiguration (DPR)

DPR is very helpful in implementing multi-standard SDR system on a single chip as it allows runtime reconfiguration of a previously chosen partition on the FPGA to be reconfigured with partial bit-stream files.

1.4.1. Advantages of DPR

1. **Resource Utilization**

   In static reconfiguration, for different standards to work simultaneously each standard must have its own resources but using DPR allows having only one set of resources that serves all standards in turn.

2. **Upgradeability**

   All standards have different versions and updates that either make the system better or deals with a definite issue. DPR allows the download of the new update without any change in hardware as long as the resources reserved could support the new update. An updated partial bit-stream file can be downloaded to replace the older version.

3. **Saving power**

   As there's only one chain working at a time and due to the lower utilization of resources, power is decreased considerably.

4. **Saving money**

   Due to the decreased utilization of resources and the power saved, money is also saved.

   To load one of the partial bit-stream files stored into the reconfigurable partition we must use access ports such as JTAG, PCAP… etc. and different DPR modes.

1.4.2. DPR Modes

XILINX offers two different modes for implementing DPR in order to transfer the bitstream file into the configuration memory, where each mode has different techniques (Only common and familiar techniques are discussed not all of them), as shown in Figure 1-8,.

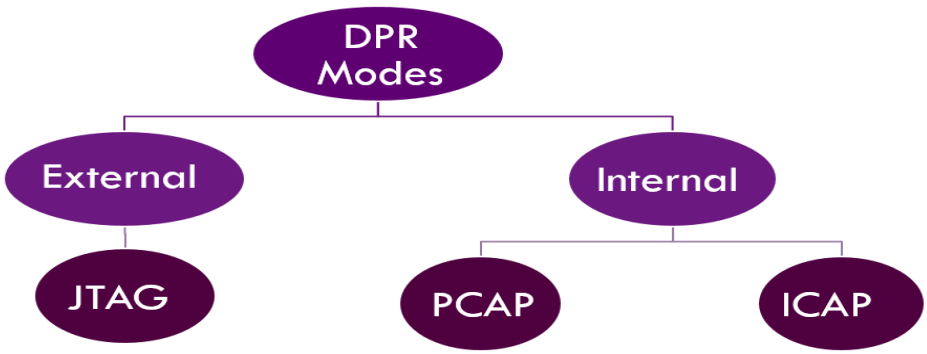

Figure 1-8 DPR Modes

1.4.2.1. External Mode:

Partial bit-stream are loaded through external source JTAG - Joint Test Action Group-. It has high reconfiguration time and a lot of time overhead due to headers, checksum and the transfer of the data from source of JTAG to the reconfigurable memory which takes about 150 msec.

The max theoretical bandwidth is approximately 66 Mbps and it transfers 1-bit at a time (i.e. serial) which leads to an actual BW of 8.25 Mbps.

1.4.2.2. Internal Mode:

Reconfiguration takes place through an already implemented access port to the configuration memory. The ARM processor is responsible for the control signal which activates the MUX in PL to either be ICAP or PCAP. A quick system overview is shown in Figure 1-9.

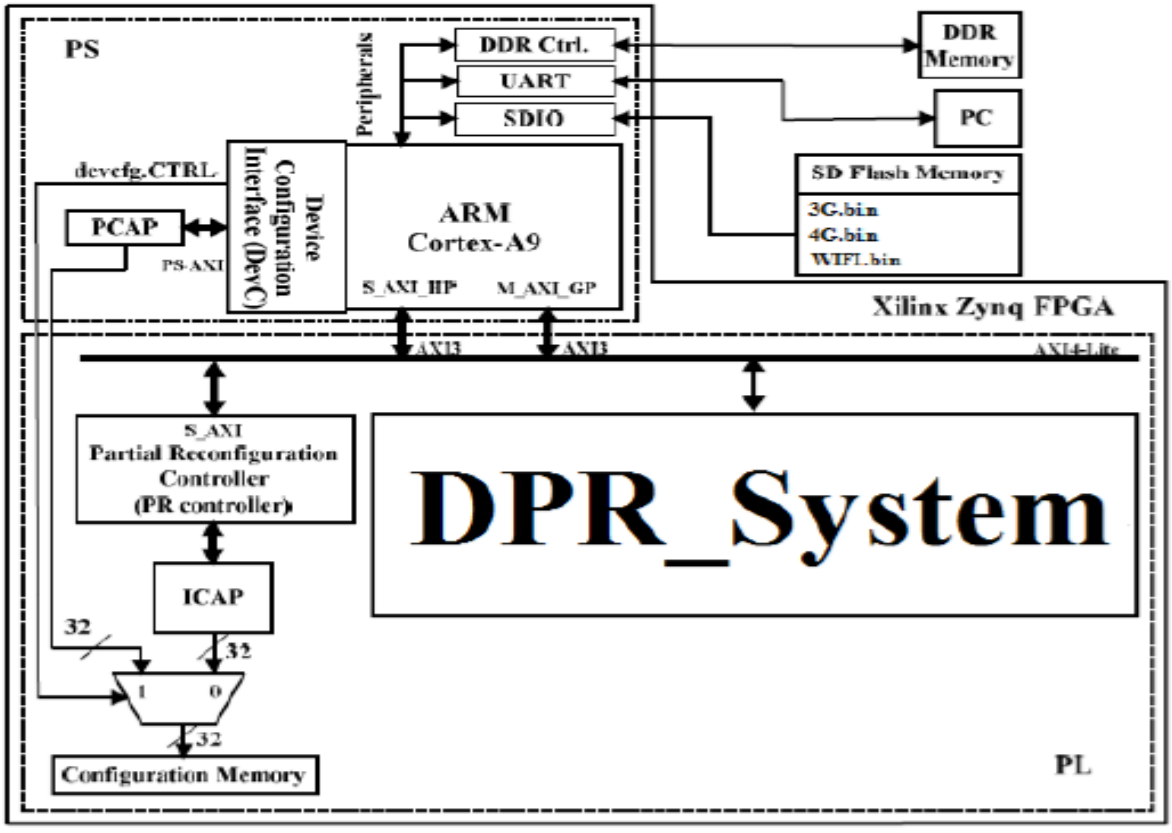

Figure 1-9 PS & PL Configuration

a. Processor Configuration Access Port (PCAP)

Located in processing system (PS), it is controlled directly by the ARM cortex 9_0 processor. It has no memory of its own so it must retrieve 32-bits by 32-bits – bus size – of the partial bit-stream file stored in the DDR memory.

It takes a higher reconfiguration time than ICAP as it is limited by the speed and size of the bus and requesting to take the bus in the first place as it has no memory.

Theoretically it has a BW of 3.2 Gbps or 400 MB/sec but its actual BW is around 145 MB/sec.

b. Internal Configuration Access Port (ICAP)

It is located in the programmable logic (PL) partition and it is controlled through controllers. Those controllers have memories inside them which are small. They can hold parts of the partial bit-stream file in order to limit the use of the bus to certain periods of time during which the controller is the master of the bus. As the ICAP transfers the stored data in controller to reconfigurable memory the controller retrieves another segment simultaneously, decreasing the overall overhead on the reconfiguration time.

It has a theoretical BW of 400MB/sec or 3.2 Gbps but its actual BW depends on the type of controller used.

As complexity increases so does bandwidth.

Types of controllers:

1) AXI-HWICAP:

This is a simple IP of a simple controller composed of an asynchronous Read & Write FIFOs, control registers and an FSM along with ICAP used for reconfiguration as shown in Figure 1-10. It has low resource utilization and low average power but it also has low throughput.

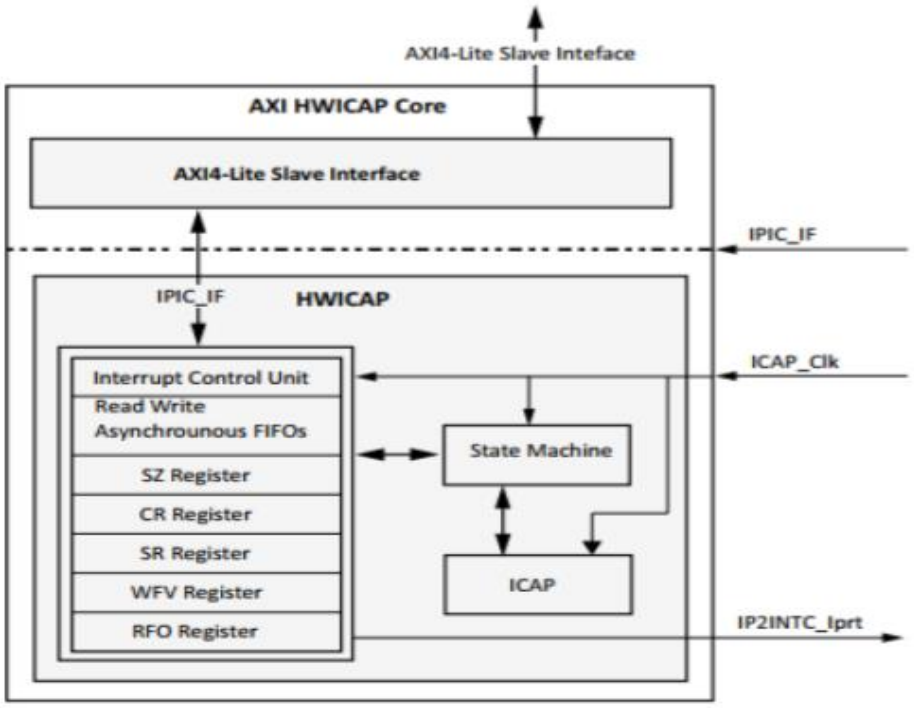

Figure 1-10 AXI-HWICAP Core

2) PRC:

More complex than AXI-HWICAP, but it has higher throughput and achieves larger BW. It uses virtual sockets which represent reconfigurable partitions and logic blocks to isolate reconfigurable partition from static region during reconfiguration.

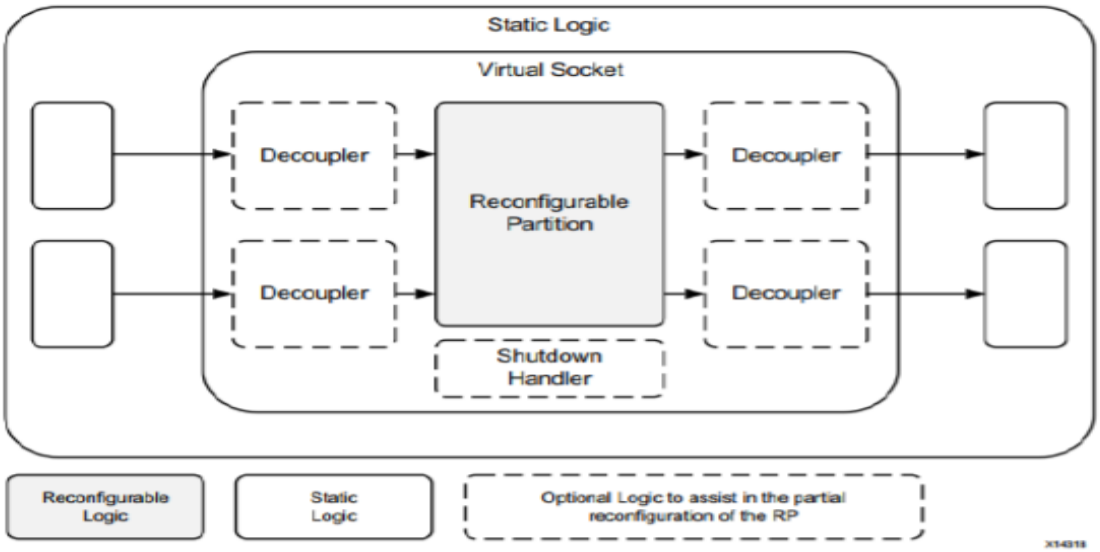

Figure 1-11 AXI-HWICAP Core

1.4.3. DPR Flow

XILINX DPR flow will be introduced through implementing a multi-standard SDR system (2G, 3G, 4G, Wi-Fi, and Bluetooth) on two reconfigurable partitions (TX, RX) using HW-ICAP IP core with the help of Vivado and SDK tools.

First, we have the SDR systems at the same time as shown in Figure 1-12, but that wastes hardware resources as previously discussed, so the system will be modified by removing (2G, 3G, 4G, Wi-Fi, Bluetooth) blocks and putting only one block which will be reconfigurable with the reconfigurable modules (RMs) as shown in Figure 1-13, where each RM represents one of the five chains that would be loaded into the chosen floor-planned reconfigurable partition on the FPGA

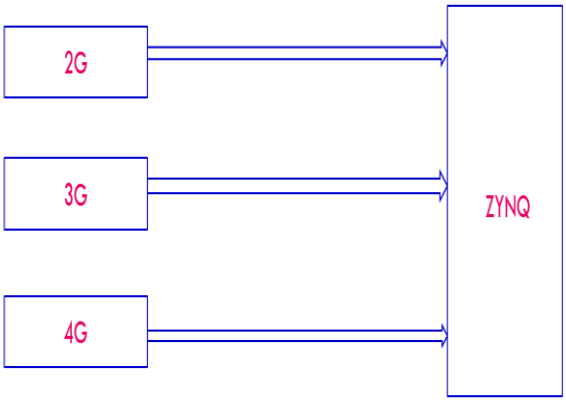

Figure 1-12 Before DPR

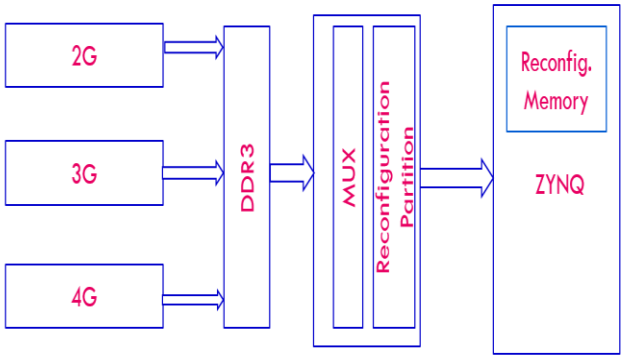

Figure 1-13 After DPR

The Flow Steps:

The flow chart shown in Figure 1-14, shows the flow steps of the DPR flow where from step 1 to step 6 are performed using Vivado, while Step No. 7 SDK is used to run our software "C" code on the processor.

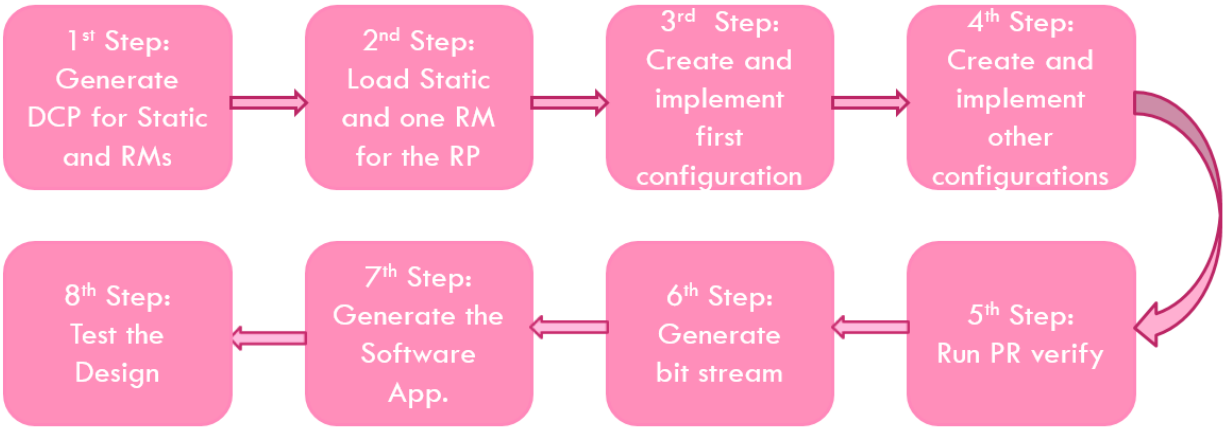

Figure 1-14 DPR flow steps

Step 1: Generate DCP for Static and RMs

1. Read HDL codes for each RM
2. Prepare a Black Box Top Module and prepare each RM to have the same I/O ports

→Black Box Module is a module where you only define the input and output ports of the module without performing any logic. This module will be used in the static design (Reconfigurable Partition shown in Figure 1-9). This Black Box Module will be modified later in the following steps with the RMs (2G, 3G, 4G, Wi-Fi, and Bluetooth). The top module of each RM must have the same module name and same input and output ports of the Black Box Module.

3. Synthesize the static and Reconfigurable Modules separately.

→In this step the static design of the DPR system is synthesized with the black box, Each RM (Standard Chain) is synthesized in a separate project

4. Generate the DCP for each synthesized RM.

Step 2: Load Static and one RM for the RP

1. Open the static DCP to do the floor-planning of the Reconfigurable Partition with resources which can cover the resources needed by each RM. In this case we have only one reconfigurable partition.

2. Make sure the Static block cover all the resource required by each RM. In our case the maximum resources required goes to the 4G RM, so if 4G RM fit into the Static block it would be guaranteed that other RMs will fit.

3. Reset after reconfigure RM

→Partial Reconfiguration solutions from Xilinx have required a manual reset action from the user to ensure all newly reconfigured logic begins in a known state (ensure that no one standard in one column at the memory, that is stagnant from the last reconfiguration, overlaps with any other standard that is newly configured).

4. Snapping Mode

→This command is to ensure that each column achieve minimum size of LUT (400)

5. Write Checkpoint for this step

Step 3: Create and implement first configuration

→In this step we will Implement a complete design (static and one Reconfigurable Module per Reconfigurable Partition) in context.

1. Read one of the RMs DCP generated in Step 1 in the Black Box cell. We start with 4G just to make sure that static block covers all the resources needed by 4G RM.

2. Implement Design using the Three Tcl command "opt_design, place_design, route_design"

3. Write Full design Checkpoint of the implemented design, as it will be used in the generation of bitstreams in Step 6.

4. Remove Reconfigurable Modules from this design and save a static-only design checkpoint.

5. Write the DCP as this DCP represents the fully implemented and routed static design that would be used later in the implementation of other RMs.

Step 4: Create and implement other configurations

→Repeat Step 3 until all Reconfigurable Modules are implemented

Step 5: Run PR_verify

→Run a verification utility (pr_verify) on all configurations to verify that the static implementation and interfaces between static block and RMs are compatible.

Step 6: Generate bit-stream

1. Read Checkpoint of each RM from DDR memory
2. Create Full bit-stream (.bit) for each RM

→ .bit using JTAG

3. Create Partial bit-stream (.bin) for each RM

→ .bin using SD Card

Step 7: Generate the Software App

1. Export Hardware
2. Launch SDK
3. Run C code

Step 8: Test the Design

Place the board in the SD boot mode. Copy the (BOOT.bin) file on the SD Card. Copy the partial bin files generated in the bitstreams directory on the SD card, and place the SD card in the board then test.

1.5. Progress of the previous years

1.5.1. Internship summer 2014

This was the first attempt in implementing SDR by using DPR. Most of the chain blocks of the three different chains (3G, Wi-Fi and LTE) were eliminated to make it simpler to implement.

The chosen blocks were only implemented using VHDL & they were as follows:

- In 3G: Convolutional Encoder, Rate Half & Rate Third.
- In Wi-Fi: Convolutional Encoder, Rate Half
- In LTE: Convolutional Encoder, Rate Third.

1.5.2. Graduation Project 2015

In that year, the objective was to design, simulate, and implement DPR system for SDR on FPGAs which was met by investigating and modeling on two different steps.

The first step by implementing PDR system for convolutional encoders used in different communication standards 3G, LTE and WIFI (completing the internship work). Where the convolutional encoders initially not exist on the chip but stored in external memory and loaded on demand.

This PDR design for the convolutional encoder was compared to conventional convolutional encoder system, where all encoders existed on the same chip. They were compared with respect to area, power, latency and memory. The results showed that PDR implementation consumes less power and area when compared to the normal design, whereas the normal design had less memory and latency.

The second step was to implement ideal communication chains for 3G, LTE and WIFI using PDR technique where swapping occurs among different blocks for implemented encoders, modulation, FFT and DFT used in these standards. This produces a reconfigurable system that can adapt different communication standards. Using PDR shows an improvement in area and power consumption with fewer extra memory and latency when compared to the normal static implementation.

These designs of both steps were implemented on Xilinx FPGA kit XUPV5-LX110T.

1.5.3. Graduation Project 2016

In that year, the progress continued & the following results were achieved.

- HDL and MATLAB implementation of 3G full transmitter and some of receiver blocks.
- HDL and MATLAB implementation of WI-FI full transmitter.
- HDL and MATLAB implementation of some of LTE transmitter blocks.
- Building a test framework to Verify of HDL implementation.
- Implementation of the three chains on the FPGA (Virtex 5).

- Generating and proving the concept of multiple RPs by implementing it on a simple example.
- Debugging the FPGA results using Chipscope.
- Building a system on chip (SOC) with input and output files.
- Reducing the total area and resources needed for implementation of the three standards.
- Reducing the total power of the system as they eliminated the static and sleep mode power consumed by the idle chains.
- Reducing reconfiguration overhead by reconfigure each internal block of the chain after finishing its function. This is a kind of pipelining as there wasn't any need to wait until all frame data was generated to reconfigure each internal block of the chain.

1.5.4. Graduation project 2017

That year is an experience of both hardware and software skills. The verification of results had been done to make sure of the success of the work.

The final results of that year's work:

- Optimization of 3G, LTE, and Wi-Fi transmitter HDL codes.
- HDL and MATLAB implementations of 3G, LTE, and Wi-Fi Receiver.
- HDL and MATLAB implementation of 2G transmitter and receiver.
- HDL and MATLAB implementation of Bluetooth transmitter and receiver.
- Building testing environment on FPGA to test all the implemented chains.
- Building DPR system using two different controllers of ICAP (HWICAP & PRC).

Chapter 2: Separation

2.1. Transmitter and Receiver separation

The transmitter and receiver of all standards were originally constructed so that, the output data of the transmitter is wired directly to the input data of the receiver with simulated attenuation and noise added to the signal, to try to imitate real channel noise and attenuation. In order to send the data via real channel, the transmitter and receiver of the standards had to be separated into two different designs with different codes, block diagrams and separate kits.

2.2. Overview

One standard was separated for testing purposes into transmitter and receiver with independent codes. The first challenge in the separation was to put that one standard into a separate code as this project will not apply the DPR concept, yet.

In this part we will talk about the steps taken in order to separate the transmitter and receiver, following the sequence of the data flow through the blocks.

For each transmitter/ receiver to act independently on a ZYNQ board it must have controlled access to the AXI interface and the processor, an input interface, a DMA, FIFO, data splitter and standard based block -which has the rate specified for the standard and the modulated data-.

The following part will be an overview of the whole system and its stages. Each stage will be discussed in detail later on in this chapter.

For the transmitter, the first block is the AXI-DMA which can control the memory access and the interface between memory mapped and stream type data. The input data is then entered through the input interface to adjust its clock to suit the system clock, which then passes the data on to the chosen test standard block. The output of said block is the modulated data, which has two parts; imaginary and real. Both parts are entered into the FIFO block for temporary storage.

The imaginary and real parts are combined together to form a single stream, which will then be stored in the SD card as a text file.

As for the receiver, the only change is an added block called data splitter. As mentioned before the imaginary and real parts of the data are concatenated into a single stream in the FIFO block of the transmitter, so in the receiver this procedure has to be reversed.

This is done using the data splitter block which comes after the input interface. It takes the received data and re-splits it into imaginary and real parts which are then entered into the chosen test standard block. Demodulation occurs in this block and then the sequence of the stages continues as before, in the transmitter.

Each of the blocks mentioned in the previous paragraph will be discussed in detail in the following parts of this chapter.

2.3. Direct Memory Access (DMA)

A DMA engine allows you to transfer data from one part of your system to another. The simplest usage of a DMA would be to transfer data from one part of the memory to another, however a DMA engine can be used to transfer data from any data producer IP block (e.g. an ADC) to a memory, or from a memory to any data consumer IP block (e.g. a DAC).

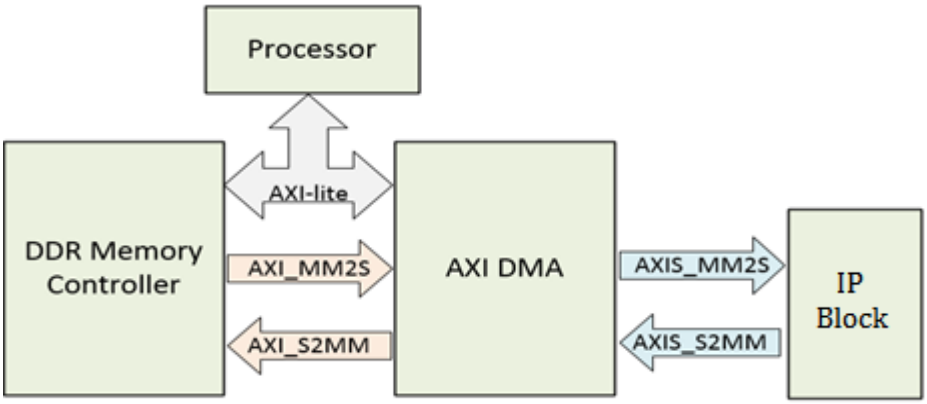

Figure 2-1 Illustration of AXI DMA use

2.3.1. The DMA functions

The AXI Direct Memory Access (AXI DMA) IP core provides high-bandwidth direct memory access between the AXI4 memory mapped and AXI4-Stream IP interfaces. All ZYNQ ports between PS and PL are memory mapped AXI interfaces, which needs a complicated control circuits to deal with it.

The AXI DMA can transfer high-burst data between PL and PS, by converting the data sent via the AXI4 interfaces to be sent via the AXI-stream interfaces, which is much easier to deal with.

As shown in Figure 2-1, the processor and DDR memory controller are contained within the Zynq PS. The AXI DMA and IP block are implemented in the Zynq PL. The AXI-lite bus allows the processor to communicate with the AXI DMA to setup, initiate and monitor data transfers. The AXI_MM2S and AXI_S2MM are memory-mapped AXI4 buses and provide the DMA access to the DDR memory. The AXIS_MM2S and AXIS_S2MM are AXI4-streaming buses, which source and sink a continuous stream of data, without addresses.

Its optional scatter/gather capabilities also offload data movement tasks from the Central Processing Unit (CPU) in processor-based systems. Initialization, status, and management registers are accessed through an AXI4-Lite slave interface. Figure 2-2 illustrates the functional composition of the core. [6,7].

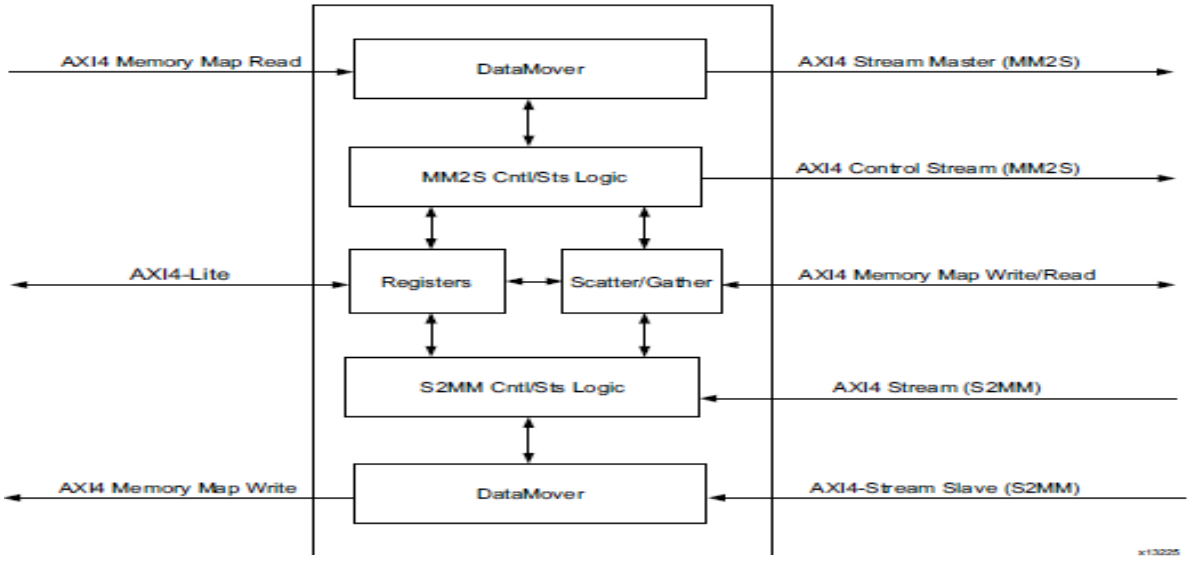

Figure 2-2 DMA block diagram

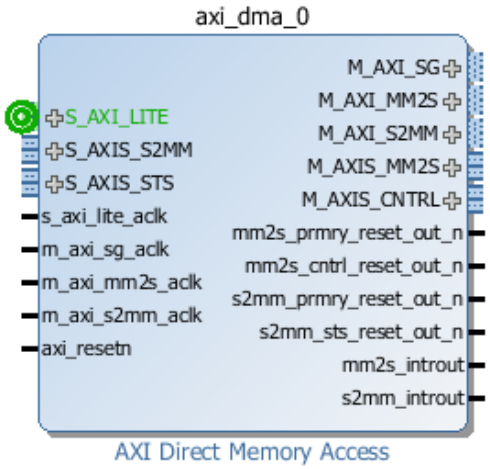

Figure 2-3 AXI DMA IP block

Figure 2-3 shows the IP block of the DMA with all its input and output signals. Primary high-speed DMA data movement between system memory and stream target is through the AXI4 Read Master to AXI4 memory-mapped to stream (MM2S) Master, and AXI stream to memory-mapped (S2MM) Slave to AXI4 Write Master. The Function of all the signals are listed in the two following tables in Figure 2-4 and Figure 2-5. [6,8]

Signal Name	Interface	Signal Type	Init Status	Description
s_axi_lite_aclk	Clock	I		AXI4-Lite Clock.
m_axi_sg_aclk	Clock	I		AXI DMA Scatter Gather Clock
m_axi_mm2s_aclk	Clock	I		AXI DMA MM2S Primary Clock
m_axi_s2mm_aclk	Clock	I		AXI DMA S2MM Primary Clock
axi_resetn	Reset	I		AXI DMA Reset. Active-Low reset. When asserted Low, resets entire AXI DMA core. Must be synchronous to s_axi_lite_aclk.
mm2s_introut	Interrupt	O	0	Interrupt Out for Memory Map to Stream Channel.
s2mm_introut	Interrupt	O	0	Interrupt Out for Stream to Memory Map Channel.
axi_dma_tstvec	NA	O	0	Debug signals for internal use.
AXI4-Lite Interface Signals				
s_axi_lite_*	S_AXI_LITE	Input/ Output		See Appendix A of the *AXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.
MM2S Memory Map Read Interface Signals				
m_axi_mm2s_*	M_AXI_MM2S	Input/ Output		See Appendix A of the *AAXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.

Figure 2-4 AXI DMA signals' functions

Signal Name	Interface	Signal Type	Init Status	Description
MM2S Master Stream Interface Signals				
mm2s_prmry_reset_out_n	M_AXIS_MM2S	O	1	Primary MM2S Reset Out. Active-Low reset.
m_axis_mm2s_*	M_AXIS_MM2S	Input/ Output		See Appendix A of the *AXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.
MM2S Master Control Stream Interface Signals				
mm2s_cntrl_reset_out_n	M_AXIS_CNTRL	O	1	Control Reset Out. Active-Low reset.
m_axis_mm2s_cntrl_*	M_AXIS_CNTRL	Input/ Output		See Appendix A of the *AXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.
S2MM Memory Map Write Interface Signals				
m_axi_s2mm_*	M_AXI_S2MM	Input/ Output		See Appendix A of the *AXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.
S2MM Slave Stream Interface Signals				
s2mm_prmry_reset_out_n	S_AXIS_S2MM	O	1	Primary S2MM Reset Out. Active-Low reset.
s_axis_s2mm_*	S_AXIS_S2MM	I	Input/ Output	See Appendix A of the *AXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.
S2MM Slave Status Stream Interface Signals				
s2mm_sts_reset_out_n	S_AXIS_STS	O	1	AXI Status Stream (STS) Reset Output. Active-Low reset.
s_axis_s2mm_sts_*	S_AXIS_STS	Input/ Output		See Appendix A of the *AXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.
Scatter Gather Memory Map Read Interface Signals				
m_axi_sg_*	M_AXI_SG	Input/ Output		See Appendix A of the *AXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.
Scatter Gather Memory Map Write Interface Signals				
m_axi_sg*	M_AXI_SG	Input/ Output		See Appendix A of the *AXI Reference Guide* (UG1037) [Ref 2] for the AXI4 signal.

Figure 2-5 cntd' DMA signals' functions

The program code for the DMA engine is written by software into a region of system memory that is accessed by the controller using its AXI master interface. The DMA engine instruction set includes instructions for DMA transfers and management instructions to control the system.

2.3.2. DMAC

The DMA controller (DMAC) uses a 64-bit AXI master interface operating at the CPU_2x clock rate to perform DMA data transfers to/from system memories and PL peripherals.

The DMAC is able to move large amounts of data without processor intervention. The source and destination memory can be anywhere in the system (PS or PL). The memory map for the DMAC includes DDR, OCM, linear addressed Quad-SPI read memory, SMC memory and PL peripherals or memory attached to an M_GP_AXI interface.

The transfers are controlled by the DMA instruction execution engine. The DMA engine runs on a small instruction set that provides a flexible method of specifying DMA transfers. This method provides greater flexibility than the capabilities of DMA controller methods.

The controller contains a multi-channel FIFO (MFIFO) to store data during the DMA transfers. The program code running on the DMA engine processor views the MFIFO as containing a set of variable-depth parallel FIFOs for DMA read and write transactions. The program code must manage the MFIFO so that the total depth of all of the DMA FIFOs does not exceed the 1,024-byte MFIFO.

The controller can be configured with up to eight DMA channels. Each channel corresponds to a thread running on the DMA engine's processor. When a DMA thread executes a load or store instruction, the DMA Engine pushes the memory request to the relevant read or write queue. See Figure 2-6 for illustration. [8]

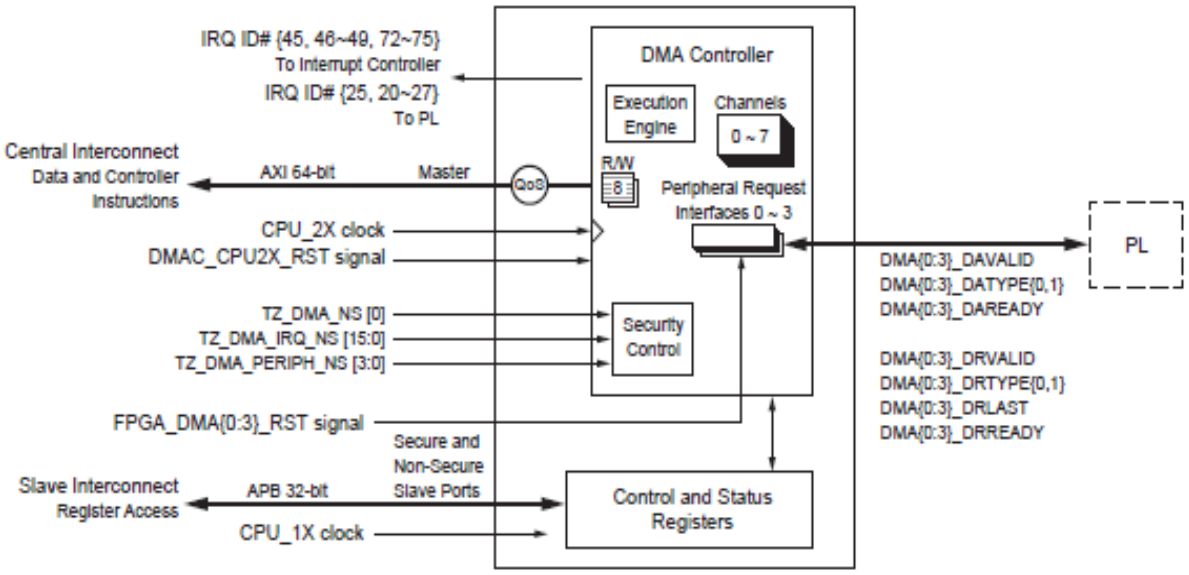

Figure 2-6 DMAC system viewpoint

2.4. Input Interface

In multi-clock systems, the system may have different clock than input or output clocks such as in our case, where the system clock is different than the input rate of the system and the output rate of the system, as shown in Figure 2-7.

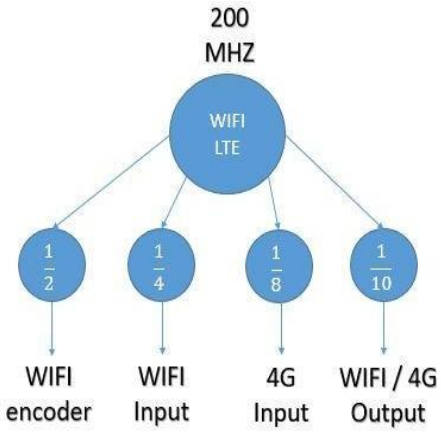

Figure 2-7 Difference problem of the system clock from the system I/P & O/P rate

The traditional design is to set a DMA with a clock for input and another for output, and a third clock for the DUT itself [3]. This may cause some issues:

a) The ARM is limited to generate 4 clocks only. Consider inserting another DUT to the system with another 3 clocks, the ARM cannot generate 7 clocks.
b) As the number of clocks increases, the Vivado synthesize time increases.
c) The clock routes in the FPGA floorplan are limited. As the clocks increases, the higher the possibility for time violation to occur.

So, the testing environment is to overcome these three issues by keeping the data (whether input or output) fixed for some clock cycles and by using the AXI-stream signals: Tready, Tvalid, Tlast.

The input interface controls the flow of data using the “Tready” signal. The Tready signal is an input to the DMA to say that the DUT is ready to receive signals. For example, if we want to get input data at a clock 8 times less than the system clock, we simply set the Tready to be LOW for 7 clock cycles and HIGH for only 1 clock cycle.

Figure 2-8 illustrates the waveform for testing the input interface in case of a rate of 4 and data length of 10.

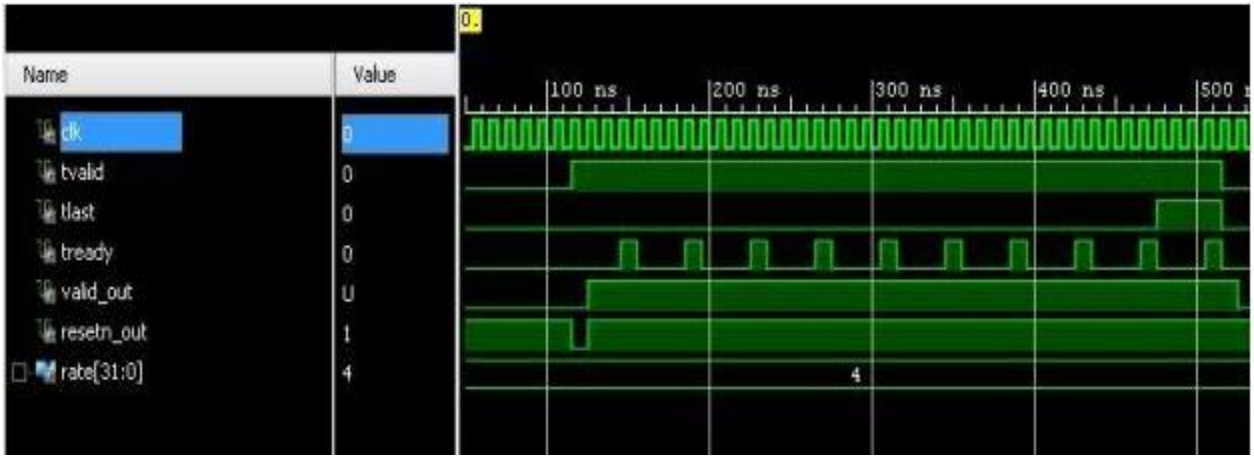

Figure 2-8 the cycles illustrating the example

As shown in Figure 2-9, the Tready is controlled through the tready counter, while the rate itself is controlled by the ARM processor through AXI [3].

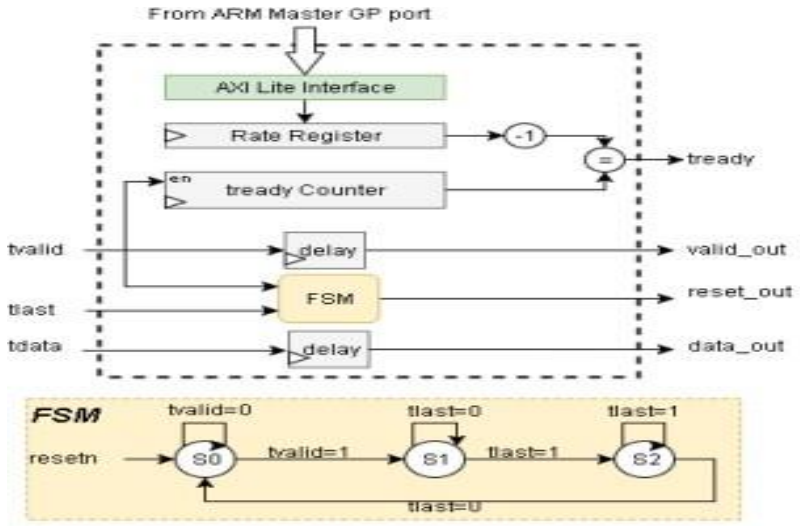

Figure 2-9 Input Interface hardware

The input interface's other function is re-setting the DUT. The idea simply is to reset the DUT at the beginning of transmitting input data, or in other words, when the "Tvalid" signal comes to HIGH. An FSM for that is shown in Figure 2-9. The "reset_out" signal is active when the state is S1. The FSM takes one clock cycle to produce the output, which is why we used the delay elements to delay the data stream to the DUT.

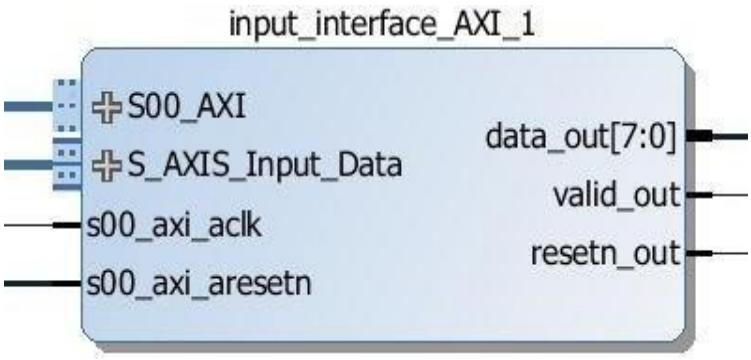

Figure 2-10 Input interface IP block

The input interface IP block illustrated in Figure 2-10, has seven signals divided into four input signals and three output signals, which will be discussed in detail. First, the S00_AXI input signal, which is a set of communication signals between the system and the input interface. Second, the S_AXIS_Input_Data input signal, which is the data coming from DDR memory and going to the connected system, remember that due to the different speeds of the connected system, the system and the memory, the input interface was needed. Third, s00_axi_aclk input signal, which is the input interface clock. Fourth, s00_axi_aresetn input signal, which is a reset signal for the input interface block. Then comes the output signals, data_out, which is the output data from the DDR memory to the connected system at a time specified by the system. The valid_out output signals specifies the time for the data_out to go out.The resetn_out signal resets the connected system.

2.5. Data splitter

The data splitter is a custom IP block constructed for the purpose of splitting the data arriving at the receiver before entering it into the standard's IP block.

The IP block shown in Figure 2-11, has Data_in signal, which is a 32-bit input signal from the input interface. The valid_in signal determines when the IP block can take the data available on the bus. When valid_in is high, this means that the data on the bus is complete and is intended for the data splitter block. After executing its sole function of splitting the input data, the block gives out three output signals, which are Data_out_real, Data_out_imag and valid out. The Data_out signals both real and imaginary are 12-bit signals that represent the split data before concatenation. Valid_out signal is the one that signifies that the output coming out of the block is correct and can be received by the next block, which is the standard block.

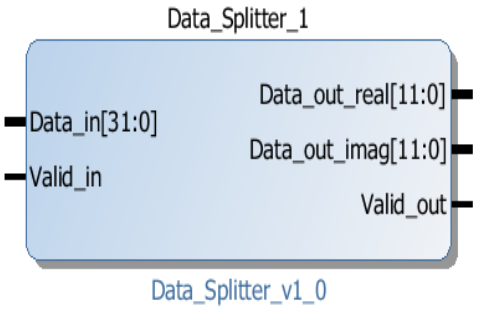

Figure 2-11 Data splitter IP block

2.6. WIFI standard IP block

The chosen standard for testing was the WIFI 802.11a. As it is standard procedure for any new technology or application to test it on the WIFI standard, then applying it on all other standards as the WIFI is easy to implement and works on a free band.

2.6.1. WIFI transmitter

The transmitter of the WIFI consists of several blocks as shown in Figure 2-12. These blocks are the ones that do the modulation and produce the output data which is then transmitted through air by using USRPs.

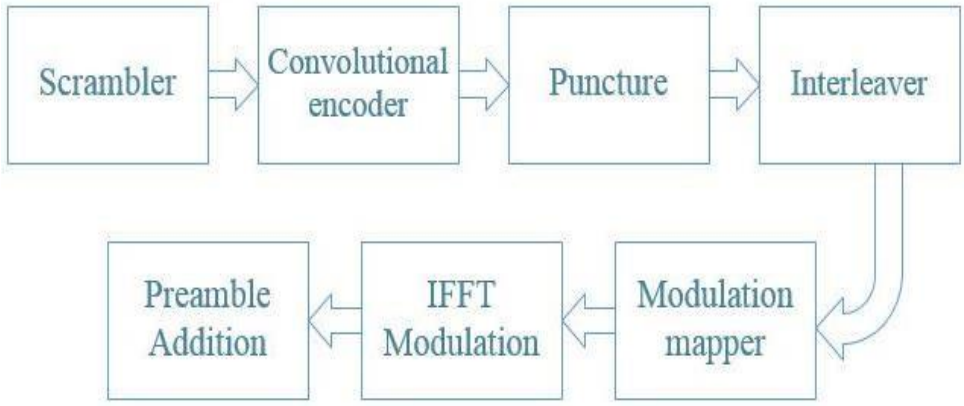

Figure 2-12 WIFI transmitter functional blocks

The previous blocks will now be discussed so as to get an overall understanding of the blocks' functions and uses.

2.6.1.1. Data Scrambler

The Scrambler is used to randomize the service, PSDU, pad and data patterns to prevent long sequences of 1s or 0s to keep synchronization. The frame synchronous scrambler uses the generator polynomial S(x) as follows:

$$S(x) = x7 + x4 + 1$$

The generator polynomial S(x) can be represented as shown in Figure 2-13.

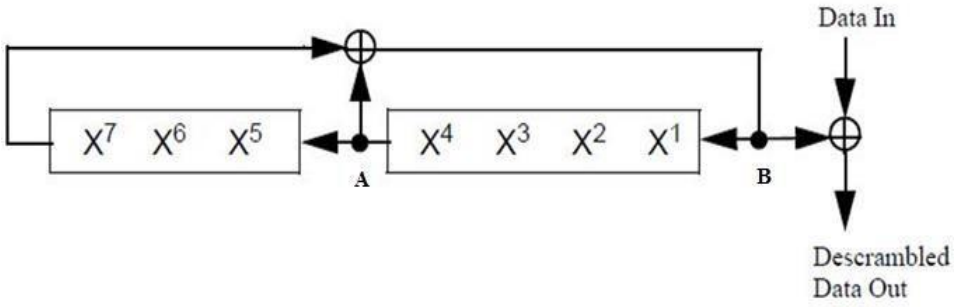

Figure 2-13 Data scrambler block diagram

According to the initial state the scrambler will generate 127-bit sequence then it will return to its initial state. The same scrambler is used to scramble the transmitted data and descramble the received data. The SERVICE filed of the data packet contains 16 bits, as shown in Figure 2-14, which are used to initialize the data scrambler. The seven LSBs of the SERVICE field are all set to zero prior to scrambling, as the receiver uses the first seven bits of the service field to determine the initial state of the scrambler.

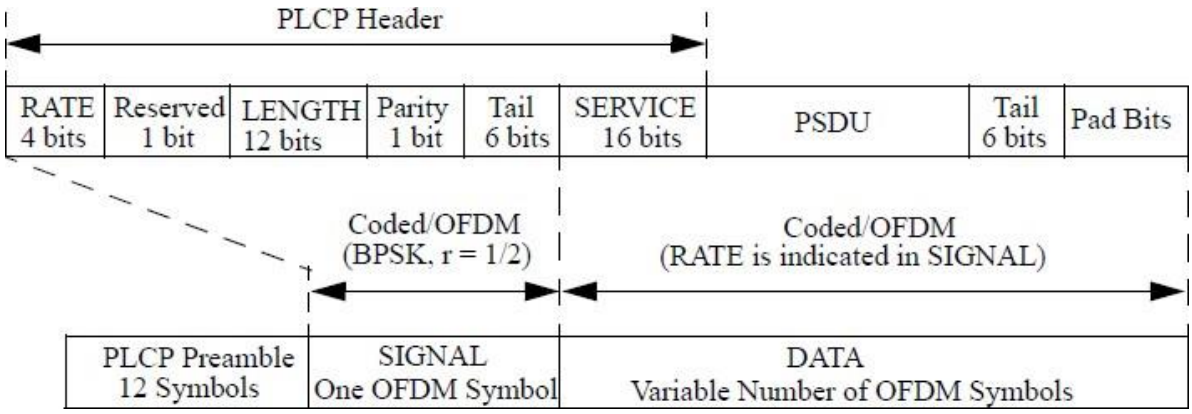

Figure 2-14 PPDU frame format

2.6.1.2. Convolutional Encoder

The DATA field, composed of SERVICE, PSDU, tail, and pad parts, shown in Figure 2-14, shall be coded with a convolutional encoder of coding rate R = 1/2, 2/3, or 3/4, corresponding to the desired data rate. The convolutional encoder shall use the industry-standard generator polynomials, g0 = 1338 and g1 = 1718, of rate R = 1/2, as shown in Figure 2-15. The bit denoted as "A" shall be output from the encoder before the bit denoted as "B." Higher rates are derived from it by employing "puncturing" [7].

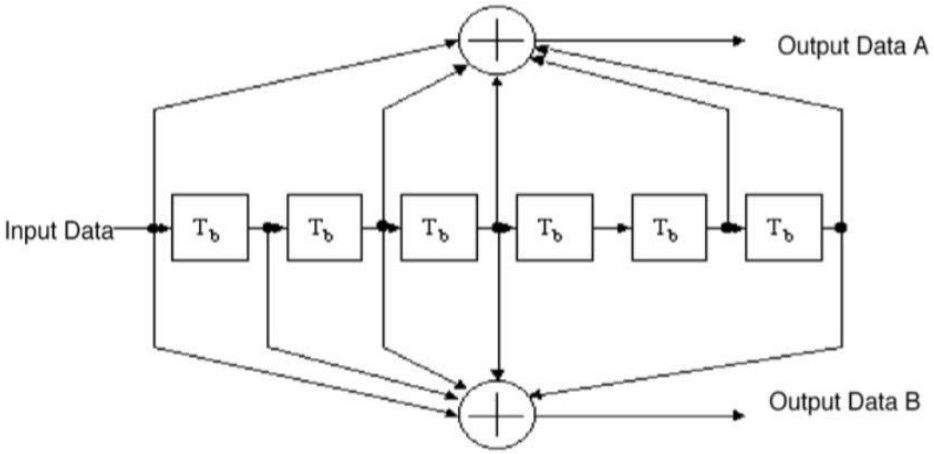

Figure 2-15 Convolutional Encoder

2.6.1.3. Puncturing

If the system could only change the data rate by adjusting the constellation size, and not the code rate, a very large number of different rates would be difficult to achieve as the number of constellations and the number of points in the largest constellation would grow very quickly. Another solution would be to implement several different convolutional encoders with different rates and change both the convolutional code rate and constellation. However, this approach has problems in the receiver that would have to implement several different decoders for all the codes used. Puncturing is a very useful technique to generate additional rates from a single convolutional code.

The basic idea behind puncturing is to not transmit some of the output bits from the convolutional encoder, thus increasing the rate of the code and inserting a dummy zero metric into the convolutional decoder on the receive side in place of the omitted bits, hence

only one encoder/decoder pair is needed to generate several different code rates. The puncture pattern is specified by the Puncture vector parameter in the mask. The puncture vector is a binary column vector. A 1 indicates that the bit in the corresponding position of the input vector is sent to the output vector, while a 0 indicates that the bit is removed. There are two types of punctures in WI-FI standard: (2/3) and (3/4) according to the data rate.

2.6.1.4. Interleaver

All encoded data bits shall be interleaved by a block Interleaver with a block size corresponding to the number of bits in a single OFDM symbol. The Interleaver is defined by a two-step permutation. The first permutation ensures that adjacent coded bits are mapped onto nonadjacent subcarriers. The second ensures that adjacent coded bits are mapped alternately onto less and more significant bits of the constellation and thereby, long runs of low reliability (LSB) bits are avoided.

2.6.1.5. Modulation Mapper

Modulation is the process by which information (e.g. bit stream) is transformed into sinusoidal waveform. A sinusoidal wave has three features those can be changed _phase, frequency and amplitude_ according to the given information and to the used modulation technique. In 802.11a Phase Shift Keying (BPSK, QPSK) and Quadrature Amplitude Modulation (16-QAM, 64-QAM) modulation techniques are used according to the desired data rate as described in the following equation: d = (I + j Q) * Kmod where Kmod is the normalization factor and is used in to achieve the same average power for all mappings.

It depends on the base modulation mode where for BPSK, Kmod = 1, for QPSK, Kmod = $1/\sqrt{2}$, for 16-QAM, Kmod = $1/\sqrt{10}$, for 64-QAM, Kmod = $1/\sqrt{40}$.

Every modulation mode has a modulation specified in the standard as shown in Figure 2-16.

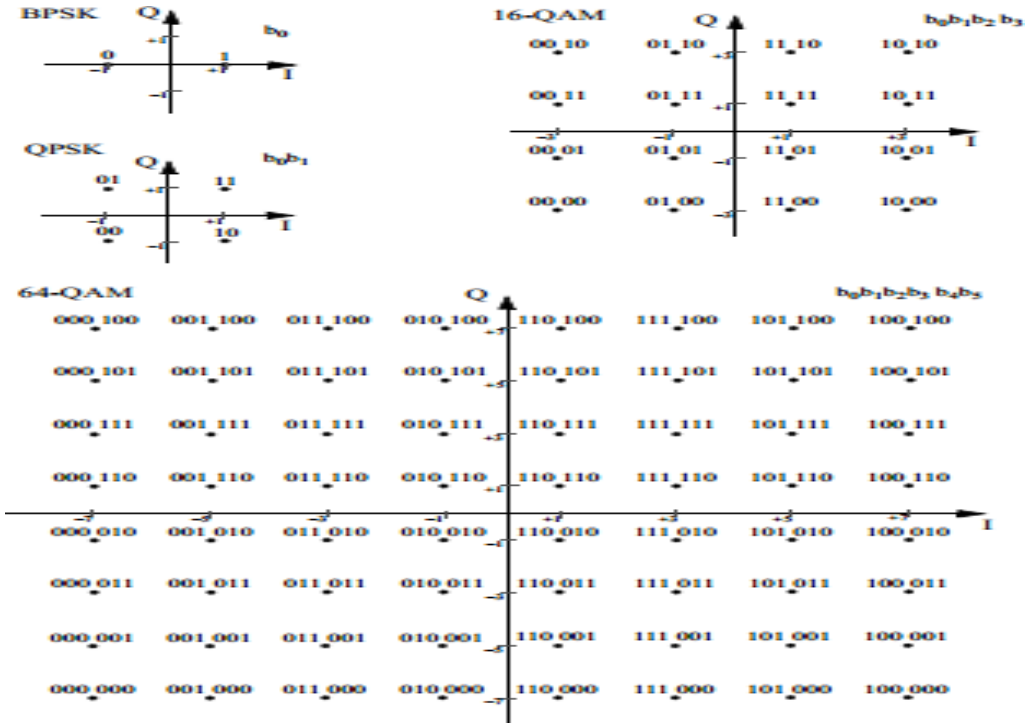

Figure 2-16 Modulation constellations for BPSK, QPSK, 16-QAM, and 64-QAM

2.6.1.6. IFFT Modulation

WIFI uses orthogonal frequency division multiplexing for modulation, An OFDM signal consists of a number of closely spaced modulated carriers as shown in Figure 2-17 , those carriers are orthogonal so the receiver could demodulate them, OFDM systems are very sensitive to frequency offset and ISI because any error in the received signal affects all carriers and all data so a guard interval is used between OFDM symbols, In this guard signal we insert a cyclic prefix of the symbol to compensate for any synchronization problems with in the receiver [10].

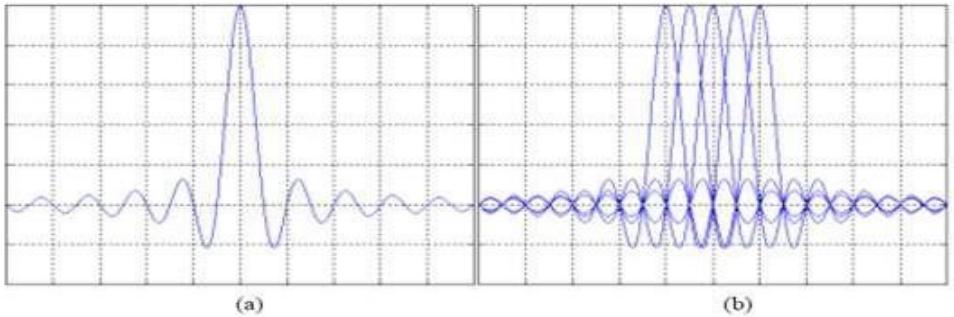

Figure 2-17 (a) Spectrum of a single subcarrier of the OFDM signal,

(b) Spectrum of the OFDM signal

The important parameters for the OFDM modulation system are the number of subcarriers used within the bandwidth, the cyclic prefix and where to insert pilot signals. Inverse fast Fourier transform is used for the modulation operation, as specified by the IEEE 802.11a, 64-point IFFT is used with symbol duration of 4 us in the 20 MHz operation of the standard. The symbol time consists of a 3.2 μs symbol and 0.8 μs for the cyclic prefix, the timing of the OFDM frame is as shown in Figure 2-18 [10].

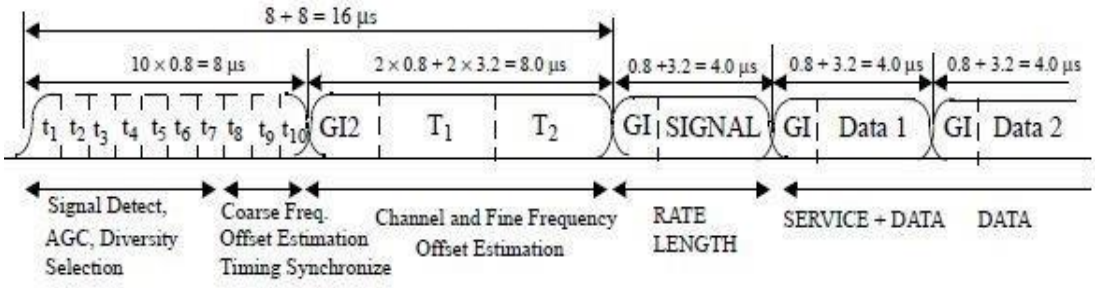

Figure 2-18 OFDM training structure

The single OFDM symbol contains 48 data symbols from the mapper, contains 4 pilot symbols, 11 null symbol and null input at DC, this mapping is shown in the below function where k is the logical subcarrier number and M(k) is the frequency offset index, the frequency offset index mapping to the IFFT inputs is shown in Figure 2-19 [10].

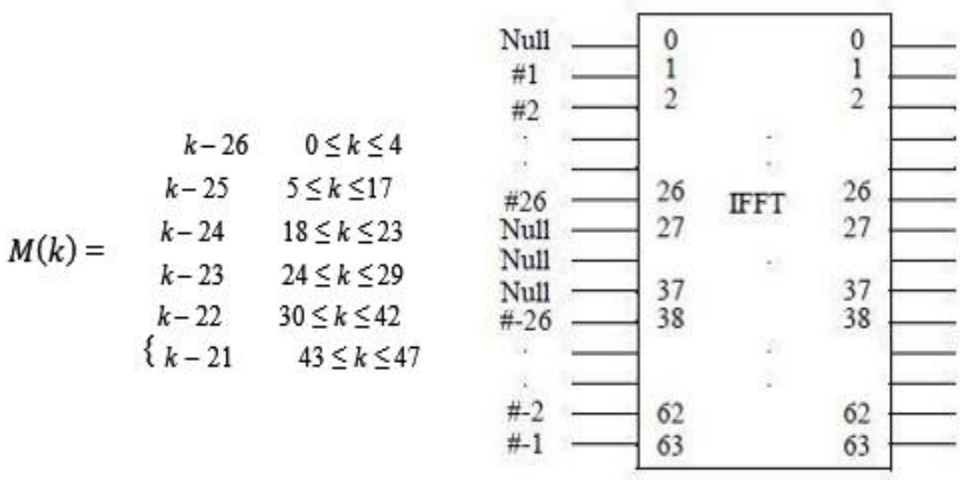

Figure 2-19 frequency offset index function & inputs and outputs of the IFFT

Pilots are inserted at subcarriers -21, -7, 7, 21. So, the final mapping of the 64 subcarrier is as shown in Figure 2-20 [10].

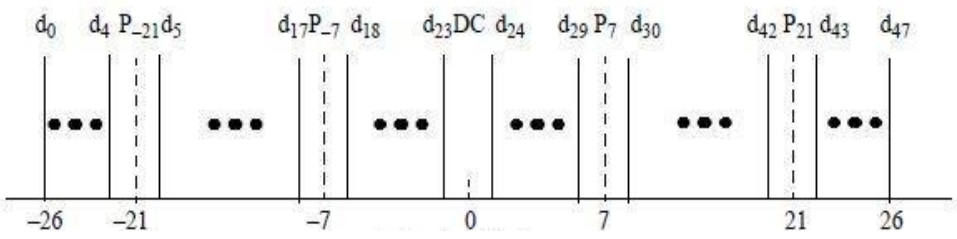

Figure 2-20 Final 64 sub-carrier mapping

The hardware circuit implementation needs an IFFT circuit, we used the Xilinx LogiCORE IP Fast Fourier Transform v7.1, and the IP has many options we used the pipelined streaming I/O to ensure continuous output to comply with the standard requirements.

2.6.1.7. Preamble

In WI-FI 802.11a, The PLCP Preamble field is used for synchronization. It consists of 10 short symbols and two long symbols that are shown in Figure 2-18. The timings described in this sub-clause and shown in Figure 2-18 are for 20 MHz channel spacing. They are doubled for half-clocked (i.e., 10 MHz) channel spacing and are quadrupled for quarter-clocked (i.e., 5 MHz) channel spacing.

Figure 2-18 shows the OFDM training structure (PLCP preamble), where t1 to t10 denotes short training symbols and T1 and T2 denote long training symbols. The total training length is 16μs. The dashed boundaries in the figure denote repetitions due to the periodicity of the inverse Fourier transform. The PLCP preamble shall be transmitted using an OFDM modulated fixed waveform.

2.6.2. WIFI receiver

The main target of the receiver is to retrieve the same data send before transmitter. The receiver consists of the blocks shown in Figure2-21. The same procedure is performed in the receiver blocks as in the transmitter blocks where in the following sub- sections, each block of the chain is explained in more details, illustrating its basic idea, showing its interfaces, connections, inputs & outputs & presenting its LUT utilization.

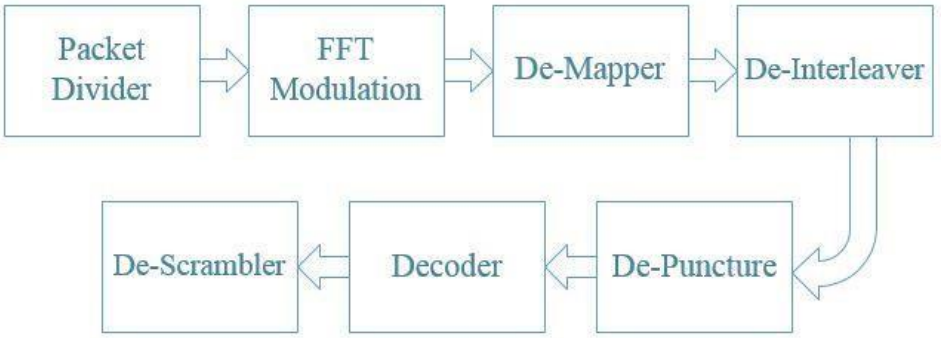

Figure 2-21 WIFI receiver full chain blocks

2.6.2.1. Packet Divider

Packet divider is the first block of the receiver, which receives the real and imaginary data of the channel which came in the form of 12 bits divided to 9 bits representing the fraction part and 3 bits representing the real part. The main target of the block is to receive these data symbols, store them, remove preamble from them and deliver the rest to the next block (FFT Modulation).

2.6.2.2. FFT modulation

As specified in section 2.5.1.6., WIFI uses orthogonal frequency division multiplexing for modulation where the hardware circuit implementation needs an FFT circuit, we used the Xilinx LogiCORE IP Fast Fourier Transform v7.1 as in the transmitter.

The first challenge is to make the data received from the Packet Divider block ready to enter the FFT core without the cyclic prefix, and to control the pipelining process such that the latency reduces as much as possible, so an FFT controller is needed to control the whole process.

The FFT controller contains the FFT core and four RAMs with size 64 x 12 where two RAMs are used for the real part & the other two RAMs are used for the imaginary part and concerning the RAM size, the FFT core receives 64 inputs with length of 12 bits. The RAM purpose is to store the input data without cyclic prefix and maintain the process of pipelining as when a RAM is getting an input, the other one delivers its data to the FFT core. This process is a continuous one till a last symbol flag is raised from the packet divider which is an indication that there is no more data to process on. While the reading and

writing processes of the RAMs are being on, the FFT core is producing its output to the next block (De-Mapper).

2.6.2.3. De-Mapper

It receives the real and imaginary data from the FFT Modulation block which came in the form of 12 bits divide to 9 bits represent the fraction part and 3 bits represent the real part. The main target of the block is to receive these data symbols, specify the decision region and convert these symbols to a stream of bits.

As specified in section 2.5.1.5., 802.11a Phase Shift Keying (BPSK, QPSK) and Quadrature Amplitude Modulation (16-QAM, 64-QAM) modulation techniques are used where the decision regions are shown in Figure 2-22. It's to be noted the .35-364-QAM modulation technique isn't included.

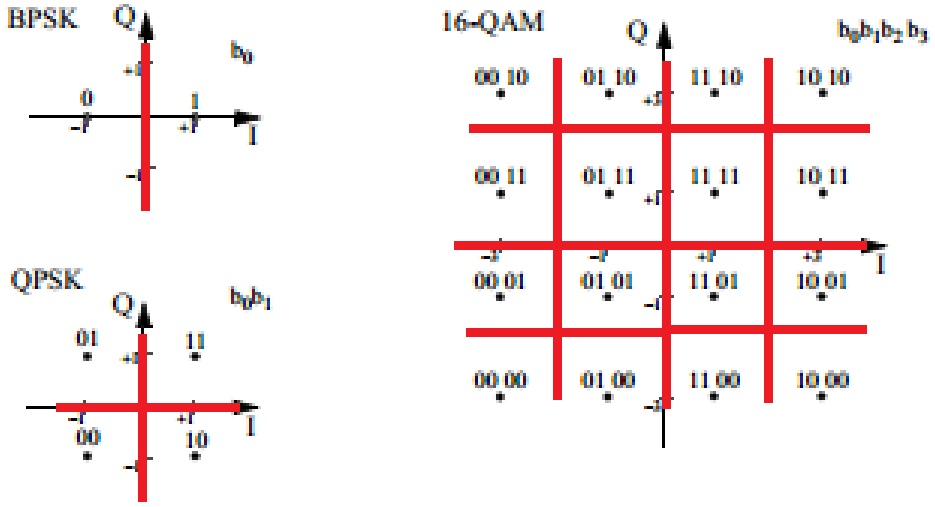

Figure 2-22 Decision regions in the de-mapper

The block design is a bit different than the usual De-Mapper design where it contains three main blocks. Thus, a controller is required to organize the signaling flow.

The first block is a stack controller which contains four stacks where two are used for real part and the other two for imaginary part to maintain the pipelining process (De-mapping while the FFT core is working). The reason for using stack structure not RAM is to cancel the transmitter effect where in the Mapper, the input data was reversed (stored up-down then read down-up).

That's why a stack is used with size 48 x 12 as the output of the FFT core is in the form of 64 data blocks and 18 sample aren't needed in the De-Mapper which are 4 pilots and 14 nulls.

The second block is the main De-Mapper block which decode the symbols into bits through the decision regions according to the modulation scheme used in the transmitter.

The third block is a large RAM used to store the whole data as the next block (De-Interleaver) needs all the frame data to be ready to work properly.

Three clocks are being used in the de-mapping process to maintain the pipelining process:

1. The normal FFT clock (system clock divided by 10) for BPSK de-mapping.
2. The output clock (system clock divided by 4) to speed up the de-mapping process in case of the modulation scheme QPSK where the De-Mapper needs more time than in the case of BPSK to decode the symbols correctly.
3. The decoder clock which is the fastest clock in the system (system clock divided by 2) to speed up the de-mapping process in case of 16 QAM.

2.6.2.4. De-Interleaver

The de-Interleaver, which performs the inverse relation to the Interleaver, is also defined by two permutations. Here the index of the original received bit before the first permutation shall be denoted by (j); (d) shall be the index after the first and before the second permutation; and (e) shall be the index after the second permutation, just prior to delivering the coded bits to the convolutional (Viterbi) decoder (if de-puncture isn't used).

The first permutation is defined by the rule:

$$\mathrm{d} = \mathrm{s} * \mathrm{Floor}\left(\frac{\mathrm{j}}{\mathrm{s}}\right) + \left(j + Floor\left(16 * \frac{j}{N_{CBPS}}\right)\right) mod\ s \qquad j = 0,1 \ldots, N_{CBPS} - 1$$

The second permutation is defined by the rule:

$$\mathrm{e} = 16 * \mathrm{d} - (N_{CBPS} - 1) * Floor\left(16 * \frac{d}{N_{CBPS}}\right) \qquad d = 0,1 \ldots, N_{CBPS} - 1$$

These permutations represent the inverse equations to the permutation equations in the Interleaver of the transmitter.

The value of s is determined by the number of coded bits per subcarrier, $NBPSC$, according to:

$$s = \max\left(\frac{N_{BPSC}}{2}, 1\right)$$

2.6.2.5. De-Puncture

De-Puncture is the reverse block of puncture. De-Puncture adds dummy bits in the position of removed bits by puncture. The positions of removed bits are determined in the standard in the puncture vector which is a binary column vector as explained in section 2.5.1.3. Figure 2-23 shows the procedure of puncture and de- puncture of rate 3/4. Figure 2-24 shows the procedure of puncture and de-puncture of rate 2/3.

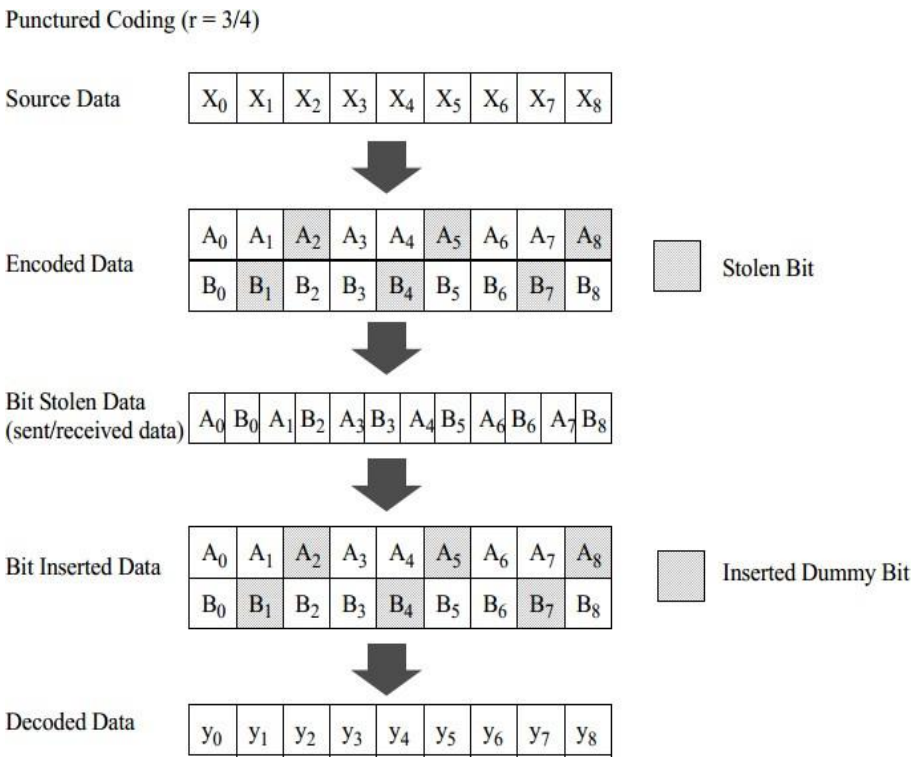

Figure 2-23 De-puncture 3/4 rate procedure

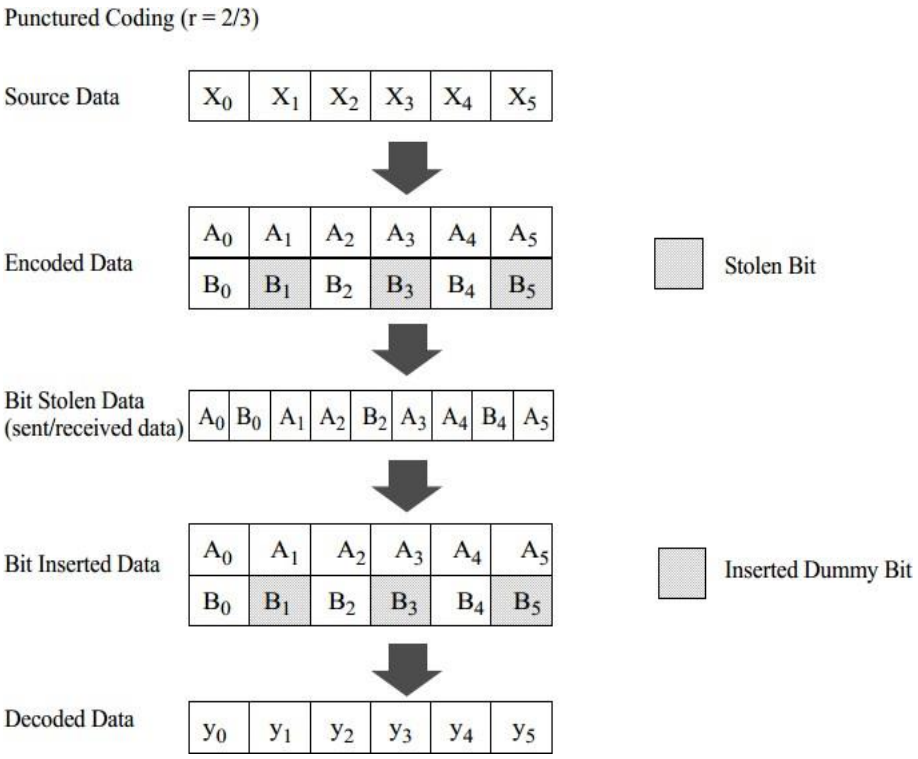

Figure 2-24 De-puncture 2/3 rate procedure

2.6.2.6. Viterbi decoder

Viterbi Decoder is the reverse block of the convolutional encoder. The block design is the same as that described in section 2.3.5. The only difference is that the used convolutional encoder has K=7, so that the Viterbi decoder here which decreases the number of the states to 64 instead of 256. Also, in this design there is no tail bit removing.

2.6.2.7. De-Scrambler

The block design is the same as that described in section 2.5.1.1., where the receiver uses the first seven bits of the service field to determine the initial state of the scrambler.

2.7. First In First Out memory (FIFO)

2.7.1. Introduction

FIFOs are essentially memory buffers used to temporarily store data until another process is ready to read it.

As their name suggests, the first byte written into a FIFO will be the first one to appear in the output. Typically, FIFOs are used in communication systems, to transfer data between two modules, running at different clocks.

Based on the difference between the speeds, the size of FIFO has to be set properly. The more the speed difference, the bigger the FIFO should be. The clock and the rate of the blocks differ depending on the implemented standard. A common example is a high-speed communications channel that writes a burst of data into a FIFO and then a slower communications channel that read the data as need to send it at a slower rate.

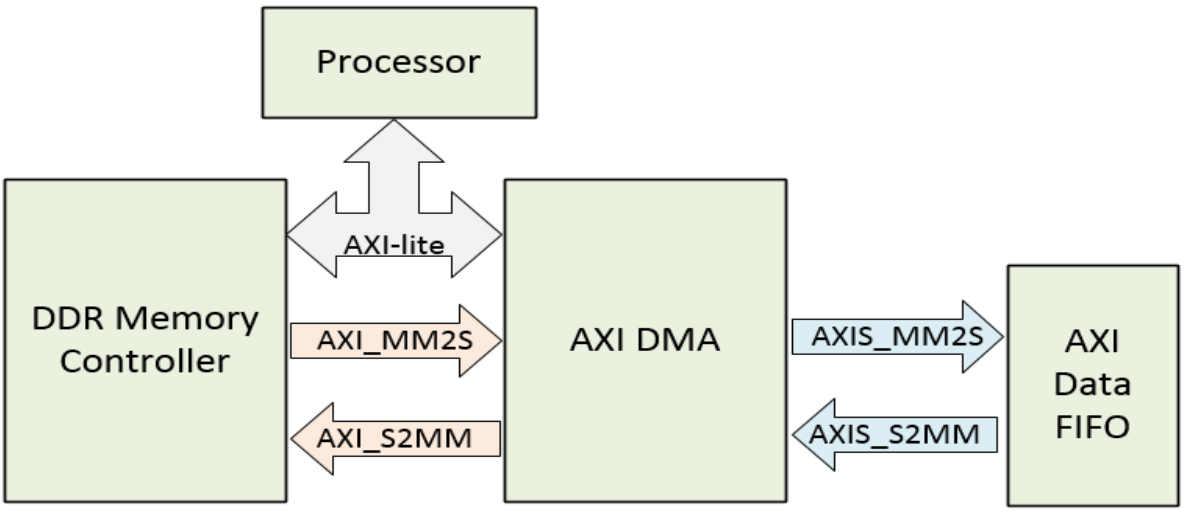

Figure 2-25 Simple Architecture of using FIFO

The preceding Figure 2-25, shows a simple architecture as an example in the use of FIFO IP block, where the FIFO is used by DMA to create loopback of reading and writing back into memory what's been read in the order of reading.

2.7.2. FIFO's rule

After the input data is modulated, the modulated data is then required to be written on the SD card, so that the USRP can read the data and send it. Hence, a way was needed to take the modulated data and write into a text file, thus the FIFO was implemented.

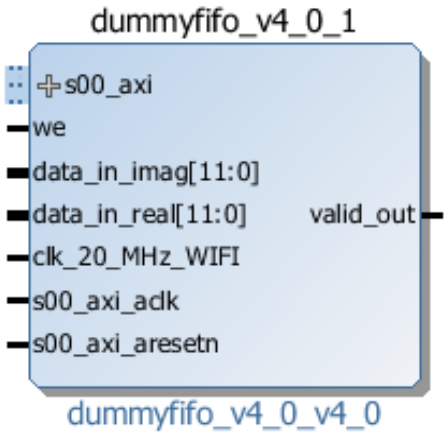

Figure 2-26 Dummy FIFO

The project's FIFO is called dummy FIFO v4 as this is the fourth version, the working version, of the FIFO. It has the interface signal with the AXI4-lite (s00_axi), a write enable signal (WE) to allow writing in the FIFO, two data input signals one for the real part of the modulated signal and one for the imaginary part (data_in_imag & data_in_real), a clock signal which is the same clock as the implemented standard (clk_20_MHz_WIFI), which is WIFI standard in this case, another clock signal for the interface with the AXI (s00_axi_aclk), and a reset signal which is a signal that is connected to all IP blocks wired to the AXI (s00_axi_aresetn).

In order to understand, we will start the flow from the beginning. The input data –which is stored as a text file on the SD card- is transferred to the DDR memory. By using AXI4-Lite DMA and AXI4-Lite interface the data can be accessed and used from the DDR memory. The data is then passed through the input interface with the appropriate size and clock rate to the IP block of the standard to modulate the data. The output data is then passed to the FIFO which is a simple first in first out memory that stores all the data coming out of the IP block of the standard in each cycle. After all data is modulated and stored in the FIFO, it can be written in a file by using a simple C code that accesses the FIFO and outputs the data on a print screen or writes it in a file on the SD card depending on the commands in the code.

2.8. AXI interface

AXI Interconnect core connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. AXI is part of ARM AMBA, a family of micro controller buses first introduced in 1996. [11]

2.8.1. AXI4 types

AXI4 which the 4th version of the AXI interface has three types, which are:

- AXI4—for high-performance memory-mapped requirements.
- AXI4-Lite—for simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream—for high-speed streaming data.

Both AXI4 and AXI4-Lite interfaces consist of five different channels:

- Read Address Channel
- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

2.8.2. AXI4-Lite

The AXI4-Lite Interface is a memory mapped interface. In memory mapped AXI (AXI3, AXI4, and AXI4-Lite), all transactions involve the concept of a target address within a system memory space and data to be transferred. Memory mapped systems often provide a more homogeneous way to view the system, because the IPs operate around a defined memory map.

AXI4-Lite is more advanced than AXI Stream as the data can move in both directions, read and write, between the masters and slaves simultaneously. That's why the channel is considered to be full-duplex or bidirectional channel. The limit in AXI4 is a burst transaction of up to 256 data transfers. AXI4-Lite allows only one data transfer per transaction by using single beat read and write; which means only one memory position can be read or written per request [11]. The Interface data widths for AXI4-Lite is 32 bits, same as the address width.

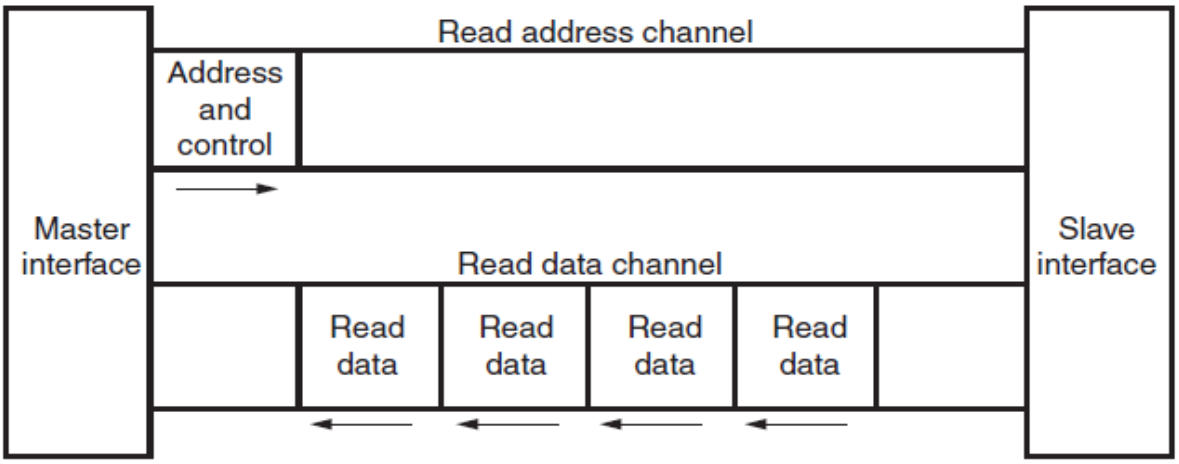

Figure 2-27 Channel Architecture of Reads

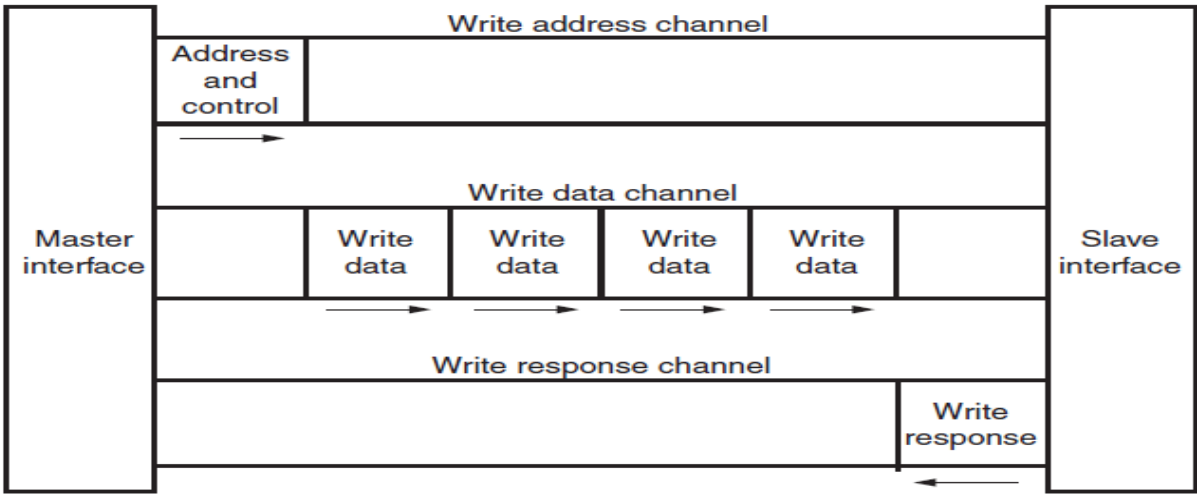

Figure 2-28 Channel Architecture of Writes

As shown in the preceding two figures, the five different channels guarantee simultaneous communication between master and slave [12].

Figure 2-29 shows a top view of the system and how the AXI interconnects the slaves and the masters in an efficient, bidirectional, simultaneous way.

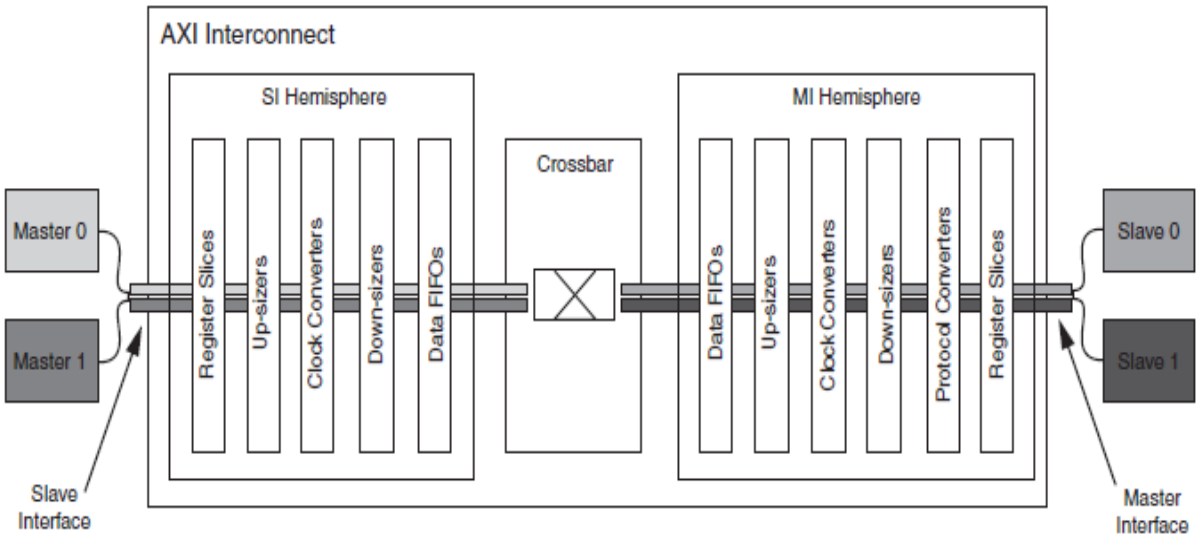

Figure 2-29 Top level AXI interconnect

Chapter 3: Interfacing

USRP and GNU radio Interface

3.1. USRP

3.1.1. USRP Hardware

The universal software radio peripheral (USRP) has become a popular platform for hardware-based research and test bed validations conducted by universities in the software defined radio (SDR) and cognitive radio (CR) fields. With the recently released version of National Instruments (NI) LabVIEW, the USRP now offers a scalable, simpler, and easier to use combined platform. [13]

The USRP Software Defined Radio Device is a tunable transceiver for prototyping wireless communication systems. It offers frequency ranges up to 6 GHz with up to 56 MHz of instantaneous bandwidth. [14]

The USRP can simultaneously receive and transmit on two antennas in real time. All sampling clocks and local oscillators are fully coherent, thus allowing you to create MIMO (multiple input, multiple output) systems. [15]

The USRP is a data acquisition board containing several distinct sections. The analog interface portion contains four analog-to-digital converters (ADC) and four digital-to-analog convertors (DAC). The ADC's operate at 64 million samples per second (Msps) and the DAC's operate at 128 Msps. Since the USB bus operates at a maximum rate of 480 million bits per second (Mbps), the FPGA must reduce the sample rate in the receive path and increase the sample rate in the transmit path to match the sample rates between the high-speed data converter and the lower speeds supported by the USB connections. [16]

So, to sum up, the entire USRP design is open source, including schematics, firmware, drivers, and even the FPGA and daughterboard designs.

When combined with the open source GNU Radio software, you get a completely open software radio system enabling host-based signal processing on commodity platforms. No software or licenses need to be purchased. It provides a complete development environment to create your own radios. While most often used with GNU Radio software, the USRP is flexible enough to accommodate other options. Some users have created their own SDR environments for the USRP, while others have integrated the USRP into the LabView and Matlab/Simulink environments. [15]

3.1.2. USRP Benefits [3]

1. **Low cost, flexible platform**

   It provides a low-cost development platform for testing Software Defined Radio (SDR) concepts. It provides several functions: digitization of the input signal, digital tuning within the IF band, and sample rate reduction before sending the digitized baseband data to the computing platform via the USB interface. It provides the opposite processing functions for the transmit path.

2. **Large community of developers**
3. Close coupling with the GNU Radio software radio framework forms a flexible and powerful platform

3.1.3. USRP Hardware Driver (UHD)

The USRP hardware driver (UHD) is the device driver provided by Ettus Research for use with the USRP product family. [17]

The UHD is a user-space library that runs on a general-purpose processor (GPP) and communicates with and controls all of the USRP device family. [18]

USRPs are transceivers, meaning that they can both transmit and receive RF signals. UHD provides the necessary control used to transport user waveform samples to and from USRP hardware, as well as control various parameters (e.g. sampling rate, center frequency, gains ...etc.) of the radio. UHD GPP driver and firmware code is written in C/C++ while the code developed for the FPGA (Field Programmable Gate Array) is written in Verilog.

There is a C/C++ API that can interface to other software frameworks, as in the case of GNU Radio, or a user can simply build custom signal processing applications directly on top of the UHD C/C++ API.

Figure 3-1 illustrates this concept.

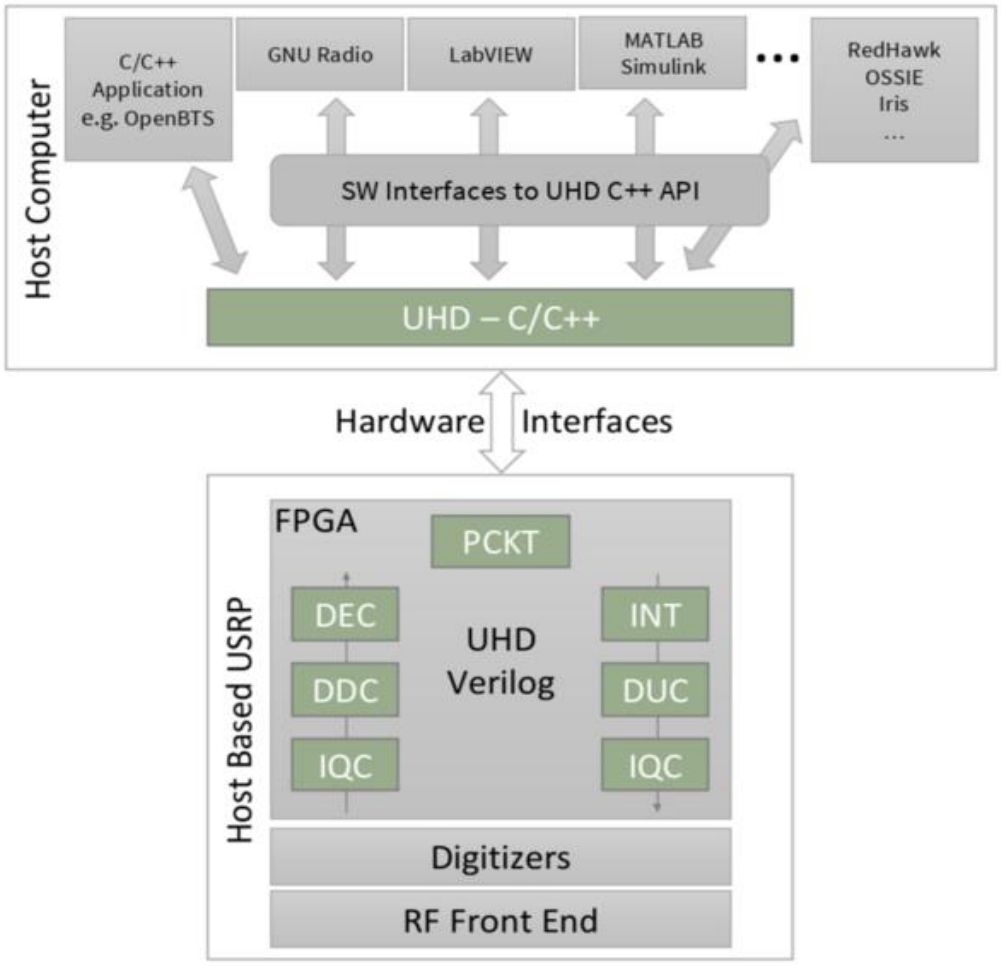

Figure 3-1 UHD Components

USRP's RF front ends may support a certain frequency step size that does not meet all or many of a user's requirements. For this reason, UHD includes Digital Up-Conversion (DUC) and Digital Down-Conversion (DDC) DSP blocks in the FPGA for fine tuning the RF frequency.

This allows users to:

- Have a sub-Hz RF frequency step size
- Mitigate the DC problem that exists on Direct Conversion (Zero IF) hardware.
- Fast tune inside the available bandwidth [18]

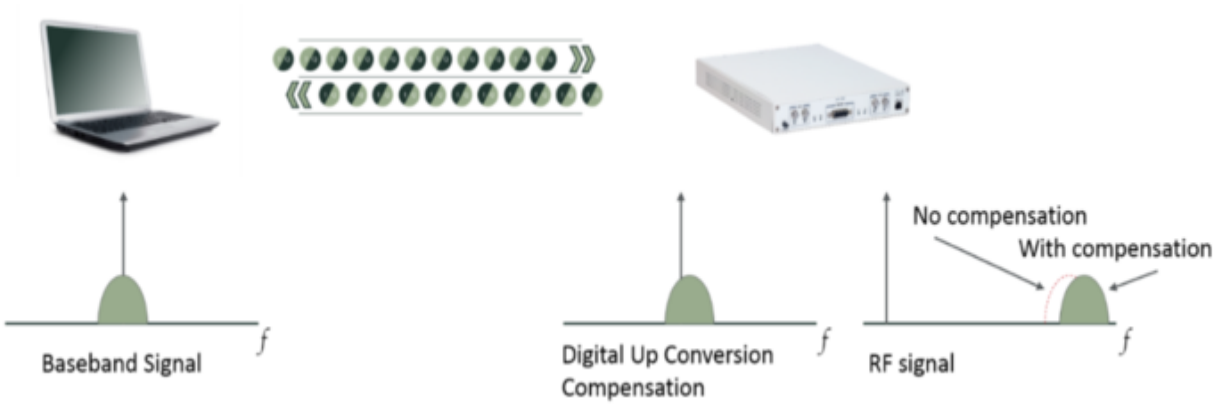

Figure 3-2 Center Frequency fine tuning

3.1.4. USRP used in our project

- B200 USRP:

Figure 3-3 B200 USRP

- It is a Universal Software Radio Peripheral, developed by Ettus Research LLC. The USRP product family is intended to be a comparatively inexpensive hardware platform for software radio.

- The USRP B200 shown in Figure 3-3, provides a fully integrated, single board, Universal Software Radio Peripheral platform with continuous frequency coverage from 70 MHz –6 GHz.
- It enables experimentation with a wide range of signals including cellular, Wi Fi, and more.
- It is designed for low-cost experimentation, it combines a fully integrated direct conversion transceiver providing up to 56MHz of real-time bandwidth.
- It consists of three components shown in Figure 3-4, which contains the USRP block diagram. The first component is an open and reprogrammable Spartan6 FPGA contains the UHD software that allows to immediately begin developing with GNU Radio, the UHD is connected to a fast and convenient bus powered super speed USB 3.0 connectivity.
- The second component is the Analog Device AD9364 RFIC that contains ADC/DAC with 12-bit flexible rate. The third component is the Temperature Controlled Crystal Oscillator (TCXO) that gives high accuracy 10 MHz frequency.

➢ Check data sheet from ref [18]

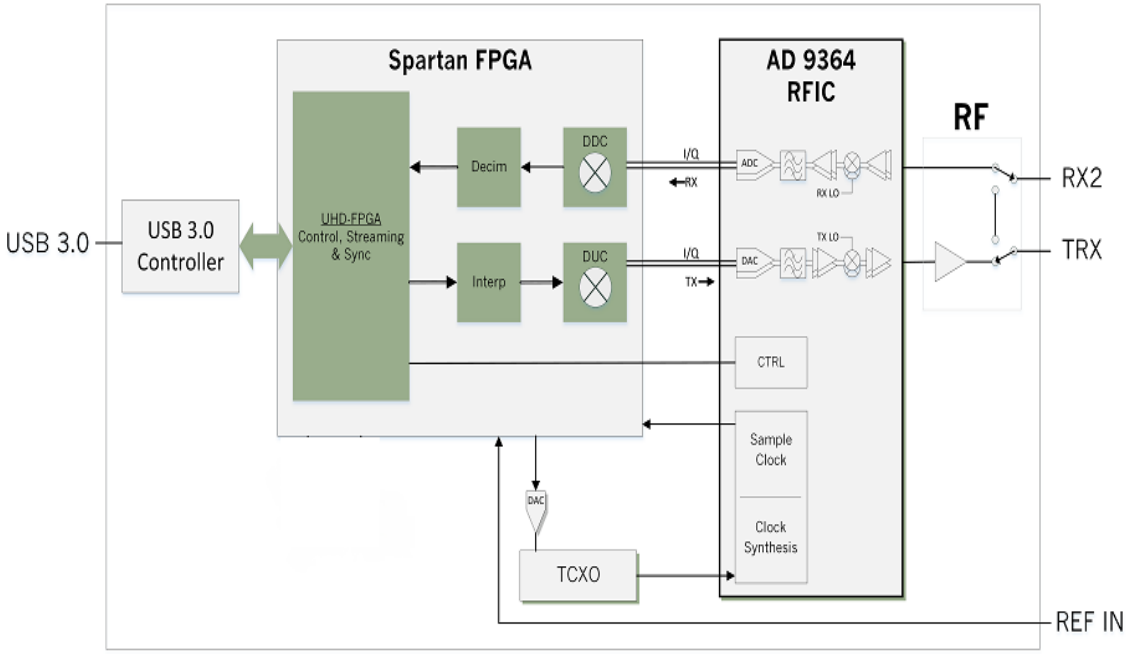

Figure 3-4 The USRP block diagram

3.2. GNU Radio

3.2.1. Introduction

In SDR there are certain stages in receiving and transmission chain, where signal is digitized, and using software techniques, computation is done on digital radio signal. The main aim of this SDR is to convert almost all hardware system problem into digital domain problems, so that it can be easily modified and problems can be solved easily. In general, SDR consist of Antennas, an ADC and subsystem defined in software domain.

Thus, there are certain conditions to be followed to implement software define radios which are as follows:

1. Antennas, which are used for any specific system, should be capable of handling all radio signal of interest, that are to be operated.
2. ADC and DAC must be designed to have sampling rate must be greater than twice the frequency of the signal of interest.
3. The unit, which executes tasks for processing, should have enough processing power to process the signal of interest.

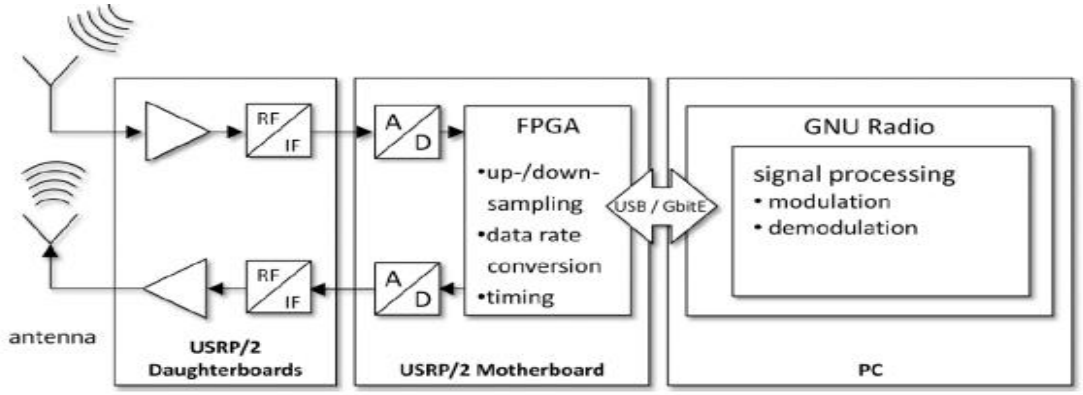

Figure 3-5 Software defined Radio Block Diagram

3.2.2. Definition

GNU Radio is a free & open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a

simulation-like environment. It is widely used in hobbyist, academic and commercial environments to support both wireless communications research and real-world radio systems. [19]

GNU Radio is a framework that enables users to design, simulate, and deploy highly capable real-world radio systems. It is a highly modular, "flow graph"-oriented framework that comes with a comprehensive library of processing blocks that can be readily combined to make complex signal processing applications.

GNU Radio has been used for a huge array of real-world radio applications, including audio processing, mobile communications, tracking satellites, radar systems, GSM networks, Digital Radio, and much more - all in computer software.

GNU Radio is licensed under the GNU General Public License (GPL) version 3 or later. All of the code is copyright of the Free Software Foundation.

It is, by itself, not a solution to talk to any specific hardware. Nor does it provide out-of-the-box applications for specific radio communications standards (e.g., 802.11, ZigBee, LTE, etc.,), but it can be, and has been, used to develop implementations of basically any band-limited communication standard.

3.2.3. What exactly does GNU Radio do?

GNU Radio performs all the signal processing. You can use it to write applications to receive data out of digital streams or to push data into digital streams, which is then transmitted using hardware. GNU Radio has filters, channel codes, synchronization elements, equalizers, demodulators, vocoders, decoders, and many other elements. In the GNU Radio jargon, we call these elements *blocks*, which are typically found in radio systems. More importantly, it includes a method of connecting these blocks and then managing how data is passed from one block to another. Extending GNU Radio is also quite easy; if you find a specific block that is missing, you can quickly create and add it.

Since GNU Radio is software, it can only handle digital data. Usually, complex baseband samples are the input data type for receivers and the output data type for transmitters.

Analog hardware is then used to shift the signal to the desired center frequency. That requirement aside, any data type can be passed from one block to another - be it bits, bytes, vectors, bursts or more complex data types.

GNU Radio applications are primarily written using the Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ using processor floating point extensions, where available. Thus, the developer is able to implement real-time, high-throughput radio systems in a simple-to-use, rapid-application-development environment.

There are ways to use GNU Radio without being able to code. First, there's the GNU Radio Companion (GRC), a Graphical User Interface (GUI) similar to Simulink, as shown in Figure 3-6. It allows you to create signal processing applications by drag-and-drop. Also, GNU Radio comes with a set of ready to-use tools and utility programs. These serve to manage the most basic operations, such as recording RF signals and performing spectrum analysis.

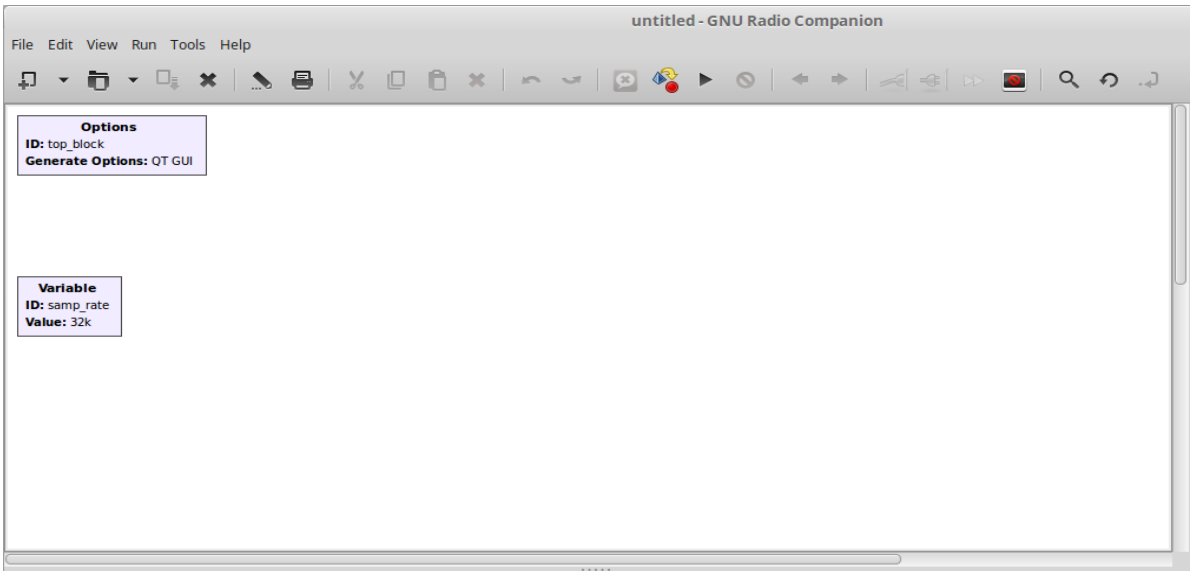

Figure 3-6 GNU Radio graphical user interface

The existing blocks in GNU Radio cover various applications from simple mathematical operations, modulators/demodulators; channel coding blocks, voice codec and others. Special classes of blocks are the input/output blocks.

The real time interface can be created by the most known UHD blocks and Audio blocks. UHD blocks are created to use the USRP to send/get signals to/from wireless medium. Thus, the audio blocks send/get the signal from the sound card.

If you want to extend GNU Radio (i.e., add new functionality), the a code for this functionality must be written. For creating applications that are too complex for the GNU Radio Companion, Python is the easiest way to go. For performance-critical code, you should write C++ code.

3.2.4. GNU Radio Live SDR Environment

The GNU Radio Live SDR Environment is a bootable Ubuntu Linux DVD or USB drive image, with GNU Radio as third-party software pre-installed. It is designed for quick and easy testing and experimentation with GNU Radio without having to make any permanent modifications to a PC or laptop. It does not, however, provide for permanent installation.

It is supplied as an ISO image to be downloaded and burned onto a recordable DVD disc or copied to a USB flash drive using a utility such as the Ubuntu Startup Disk Creator (Ubuntu Linux OS) or Unetbootin (Windows, MacOS, Linux). Creating a USB drive from the image provides much faster booting and operation and allows making changes and storing files. Finally, the ISO image may be booted within a virtual environment such as VirtualBox, QEMU/KVM, VMware, or Parallels.

The GNU Radio software source code, as well as the source code to other installed software, is installed in **/home/ubuntu/src/,** which may be browsed from the filesystem explorer or from the command line.

The GNU Radio Companion application is installed as **grc** on the system path, and may be run from any directory, or may be accessed directly from the desktop by clicking on the icon.

GNU Radio example applications are installed in **/home/ubuntu/examples** and may be run by navigating to one of the example directories and executing the python scripts using the

syntax **./foo.py**, where **foo.py** is the name of the example program, or using GRC to load and execute the GRC-based examples.

3.2.5. Installing GRC

Ref [20] explains all the steps to get a working installation of GNU Radio.

3.2.6. Using GRC

3.2.6.1. GRC Architecture

The block diagram shows transmission and reception of file source via GNU radio. In this system Host Computer is any normal Laptop / CPU with GNU Radio installed over it or running with the help of live USB environment. For interfacing of USB, system requires USB 3.0 port with USB high speed data cable. Ettus B200 board has built-in FPGA, ADC/DAC, RF Front end and RX and TX terminals blocks.

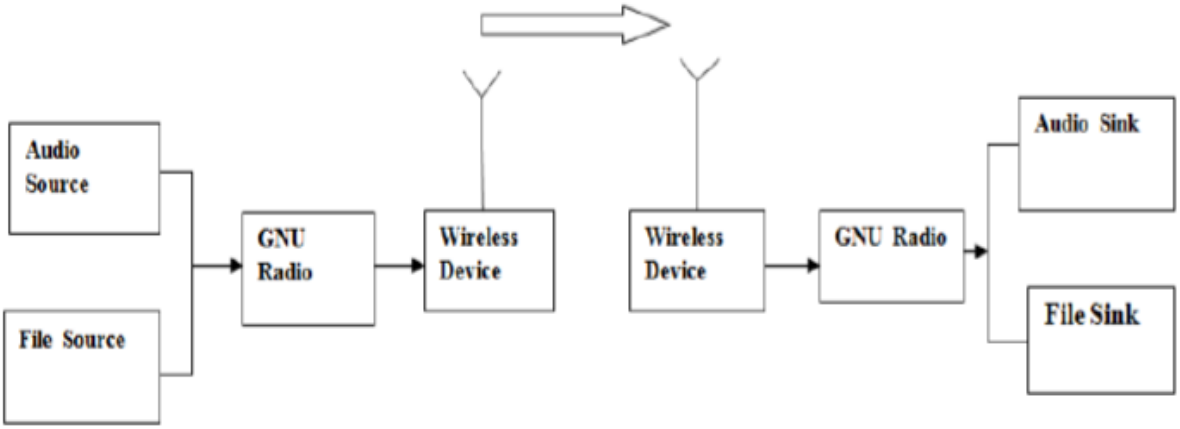

Figure 3-7 GRC Architecture for Transmitter and Receiver

3.2.6.2. Graphical signal processing development

Digital signal processing (DSP) is where GNU Radio shines; this is what it was originally made for. GRC is a Simulink-like graphical tool to design signal processing flow graphs. If you're comfortable dealing with FIR filters, digital modulators and other DSP concepts, using GRC should be simple and straightforward for you.

On Linux systems, GRC is invoked by calling the GNU radio-companion command. If your installation was fine, GRC will pop up in its own window. On the right-hand side, you can find all the available blocks (good news: adding new blocks is not terribly difficult!), which can be dragged into the main window and connected by clicking the edges. GRC has its own wiki page.

Two examples to learn using GNU radio with USRP are explained below.

Example 1: Design and Implementation of BPSK Transmitter & Receiver Using SDR

The blocks used in a bit transmitter flow graph of BPSK are shown in Figure 3-8. The block Vector Source sends a vector specified by the user. When executing this project, the vector is composed by 0s and 1s; the block repeats the vector whenever it reaches the end. The Vector Source output is connected to a Packet Encoder which is responsible for encoding the data. The packet encoder operates in a way such that the receiver can find the beginning of the transmitted data and be able to decode it - refer to section 2.5. for further explanation. After the encoder, the encoded data is sent to PSK Mod which is responsible to modulate the information using PSK [21]. The value of the samples in PSK Mod output is modified (multiplied by a constant) by multiplying it with Const block, such that the power of modulated signal can be changed. In the transmitter, the signal is sent to UHD: USRP Sink block which is responsible for the interaction with the USRP.

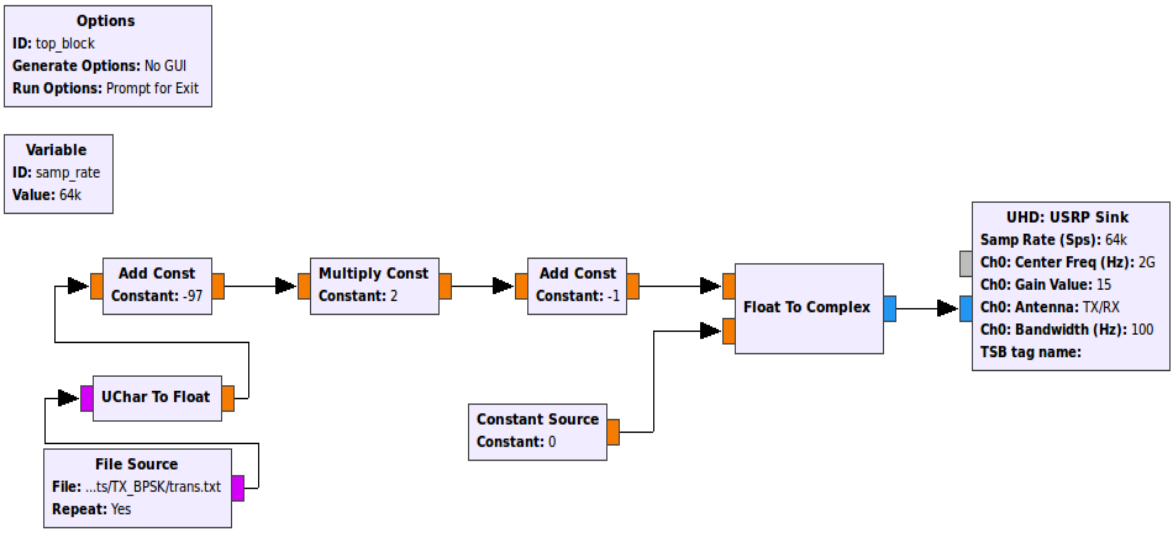

Figure 3-8 GNU Radio BPSK transmitter flow graph

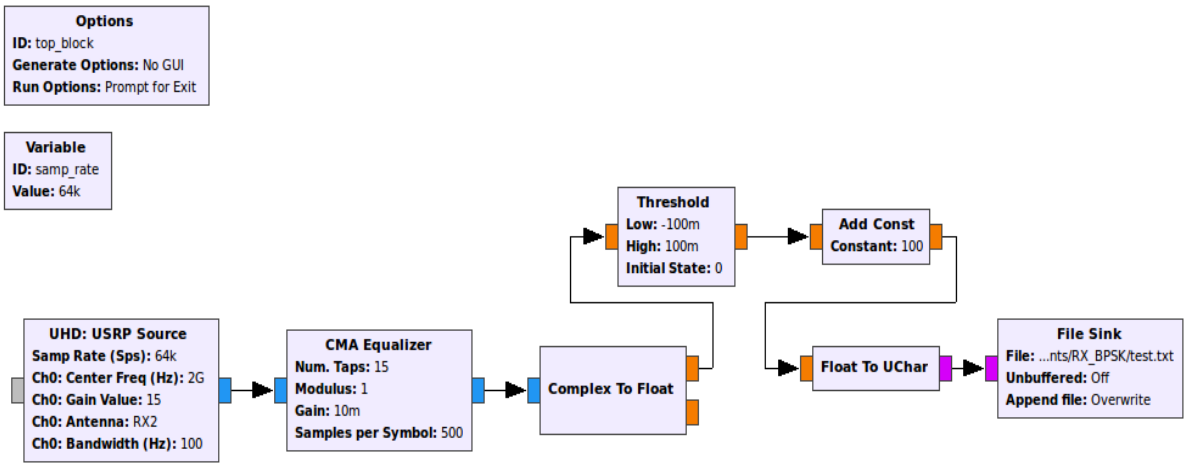

Figure 3-9 GNU Radio BPSK receiver flow graph

Figure 3-9 shows the blocks used in reception of wireless BPSK signals. The UHD: USRP Source block abstracts all the hardware in reception and its outputs are the samples of the received signal in baseband. In this block, like the UHD: USRP Sink, it is possible to define several parameters of the hardware. The UHD: USRP Source output is connected to BPSK Demodulator block that demodulates the BPSK signal, recovering the encoded data. After the demodulation, the encoded data is sent to Packet Decoder block which decodes the data and outputs the bits (the information sent by Vector Source). Once the information is recovered, the Char to float block converts the byte into float, so that the information can be used by other blocks [22].

Example 2: Design and Implementation of FM Receiver

As shown in Figure 3-10 we are trying to receive FM channels using USRP that is why our input source is going to be USRP block with center frequency of 106.7MHz which can be adjusted with the help of **Text Box**. Here, the sampling rate is 4MHz and the other properties are:

- Gain Value: It is managed by RF gain slider with min=0 and max=30.
- Antenna: Connected to TX/RX in daughter board.

Now the output of the USRP block is going to the low pass filter. Then, the filter passes the signal whose frequency is below the cutoff frequency (here it is 100 kHz) and it deducts the signal with higher frequency.

The filtered signal is then passed on to the wide band FM block. This block demodulates the wide band frequency signal from the data stream. So, at the output we get the original data stream which was sent by the sender.

Output of the WBFM block is then sent to the Rational Re-sampler. There, the data stream is decimated or interpolated according to the application and its desired rate. For example, the Audio Sink needs lower frequency and USRP block need higher, then this block converts the frequency by doing some interpolation or decimation as per the requirements. Finally, Audio sink block is used to listen the sound of the received channel.

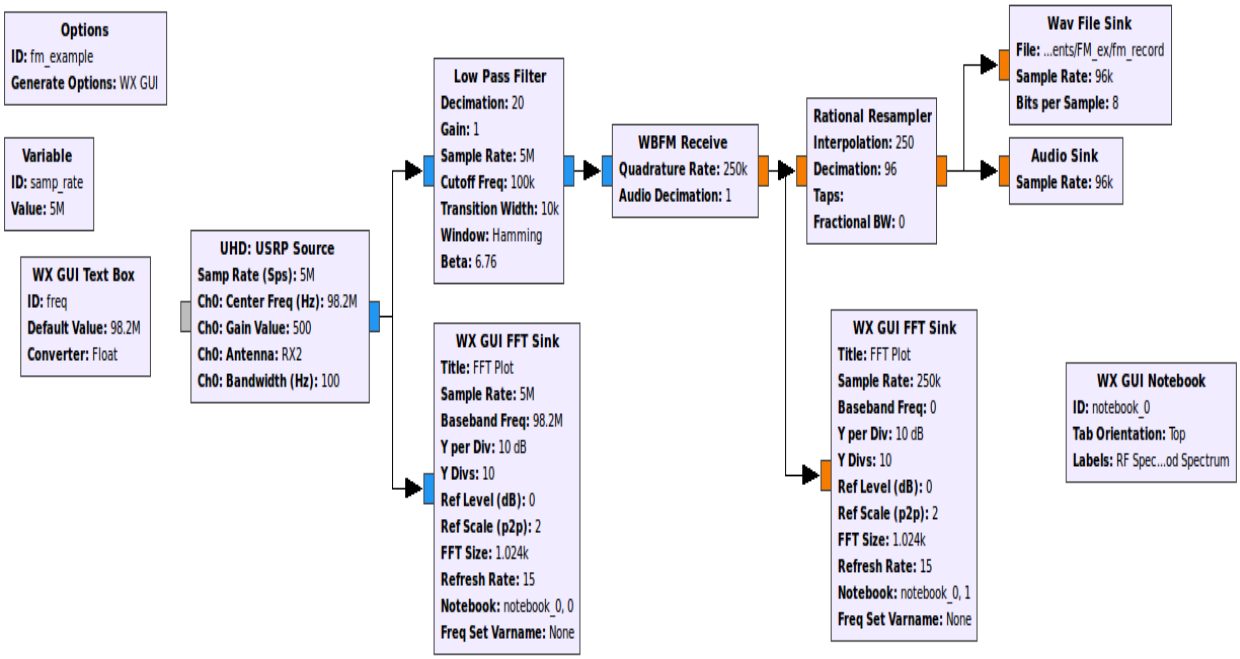

Figure 3-10 Modified FM Receiver

3.2.6.3. Using Python to write powerful signal processing and radio applications

Sometimes GRC cannot provide all the flexibility required for your application. Anything that can be clicked together in GRC can also be written in Python, and while it is more of an effort to code everything yourself, it also provides you with the entire power and functionality of Python and its libraries, such as SciPy or NumPy for Python-centric processing of your signals or your favorite widget library to create any GUI you wish.

3.2.6.4. The C++ domain: Extending GNU Radio

GNU Radio is extremely powerful and includes many kinds of signal processing blocks. However, if you're developing something particular, chances are high that sooner or later you'll be running into some component which is lacking; be it a specific channel code, a segmentation algorithm or whatever. In this case, you will want to write your own blocks to add them into GNU Radio. This is usually done in C++, in order to keep GNU Radio as fast as it is.

3.2.7. Example used in our project

In our project, the modulated file, which comes from the Transmitter block of the ZYNQ board, is processed by GNU radio and transmitted using a USRP. There is a USRP receiver node which receives the signal and GNU radio re-produces the transmitted modulated file. Then, the received modulated file is passed through the Receiver block of the ZYNQ board. The demodulated file is then checked to make sure that the output demodulated file is the same as the input file to the Transmitter block.

3.2.7.1. Basic block diagram

Two USRPs are used, one for transmitting and the other for receiving. The source file is Gaussian Minimum Shift Keying (GMSK) modulated and transmitted using the USRP transmitter. The modulator is implemented on GNU Radio. The transmitted signal is received by the receiving USRP and then demodulated by GNU Radio and played back. Figure 3-11 shows the block diagram.

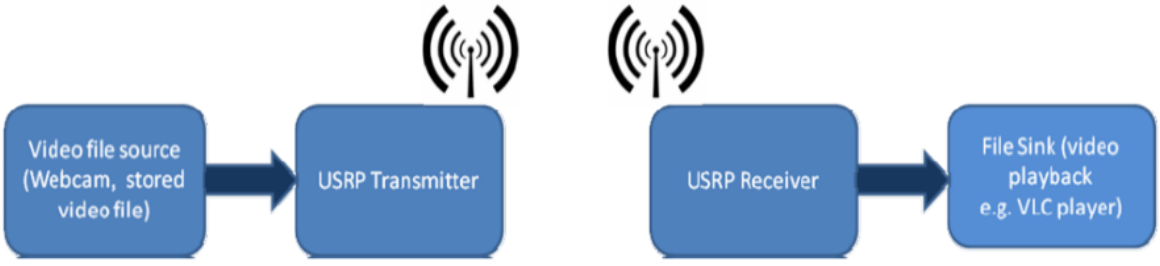

Figure 3-11 Block diagram

3.2.7.2. Experiment set up and plan

As shown in Figure 3-12 and Figure 3-13, the receiver and transmitter side are implemented on each USRP and again a modulated file shall be transmitted and received.

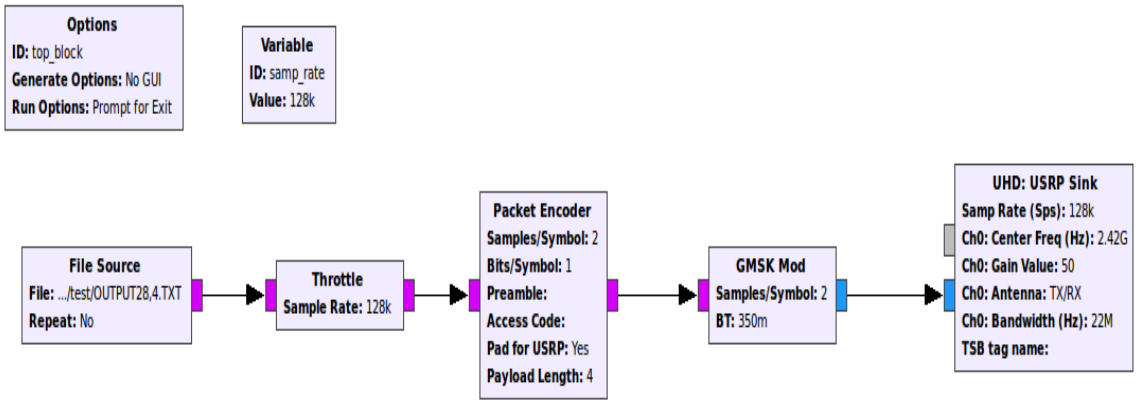

Figure 3-12 GNU Radio transmitter flow graph

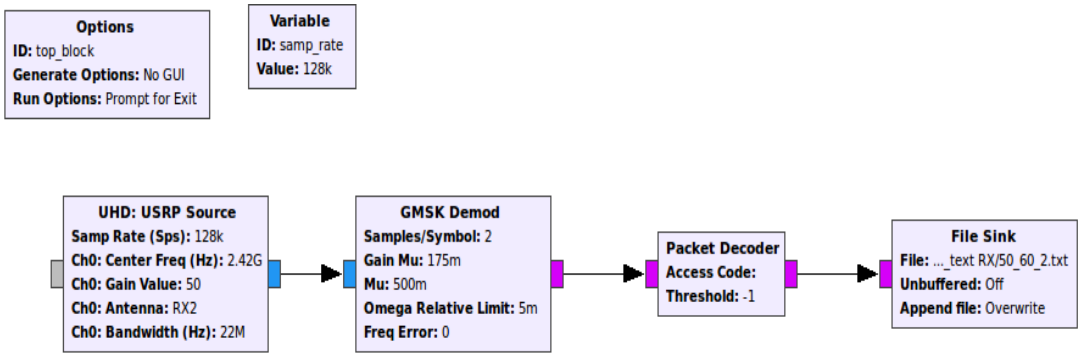

Figure 3-13 GNU Radio Receiver flow graph

The maximum distance, within which the USRP's can communicate, can be explored by changing the gain and bandwidth values. The distance between the antennas of the USRPs plays an important role as there could be distortion in the file as the distance increases due to the loss of packets or erroneous packets. The distortion and bit error rate can be anticipated by using channel estimation techniques. Introducing a suitable error correction scheme can correct the data frames received.

Chapter 4: Zynq ZC702 Evaluation Board

4.1. ZYNQ 7000 Family Overview

The Zynq®-7000 family is based on the Xilinx® All Programmable SoC (AP SoC) architecture. These products integrate a feature-rich dual or single-core ARM® Cortex™-A9 MPCore™ based processing system (PS) and Xilinx programmable logic (PL) in a single device, built on a state-of-the-art, high-performance, low-power (HPL), 28 nm, and high-k metal gate (HKMG) process technology. The ARM Cortex-A9 MPCore CPUs are the heart of the PS which also includes on-chip memory, external memory interfaces, and a rich set of I/O peripherals.

The Zynq-7000 family offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use typically associated with ASIC and ASSPs. The range of devices in the Zynq-7000 AP SoC family enables designers to target cost-sensitive as well as high-performance applications from a single platform using industry-standard tools. While each device in the Zynq-7000 family contains the same PS, the PL and I/O resources vary between the devices. As a result, the Zynq-7000 AP SoC devices are able to serve a wide range of applications.

Figure 4-1 illustrates the functional blocks of the Zynq-7000 AP SoC. The PS and the PL are on separate power domains, enabling the user of these devices to power down the PL for power management if required.

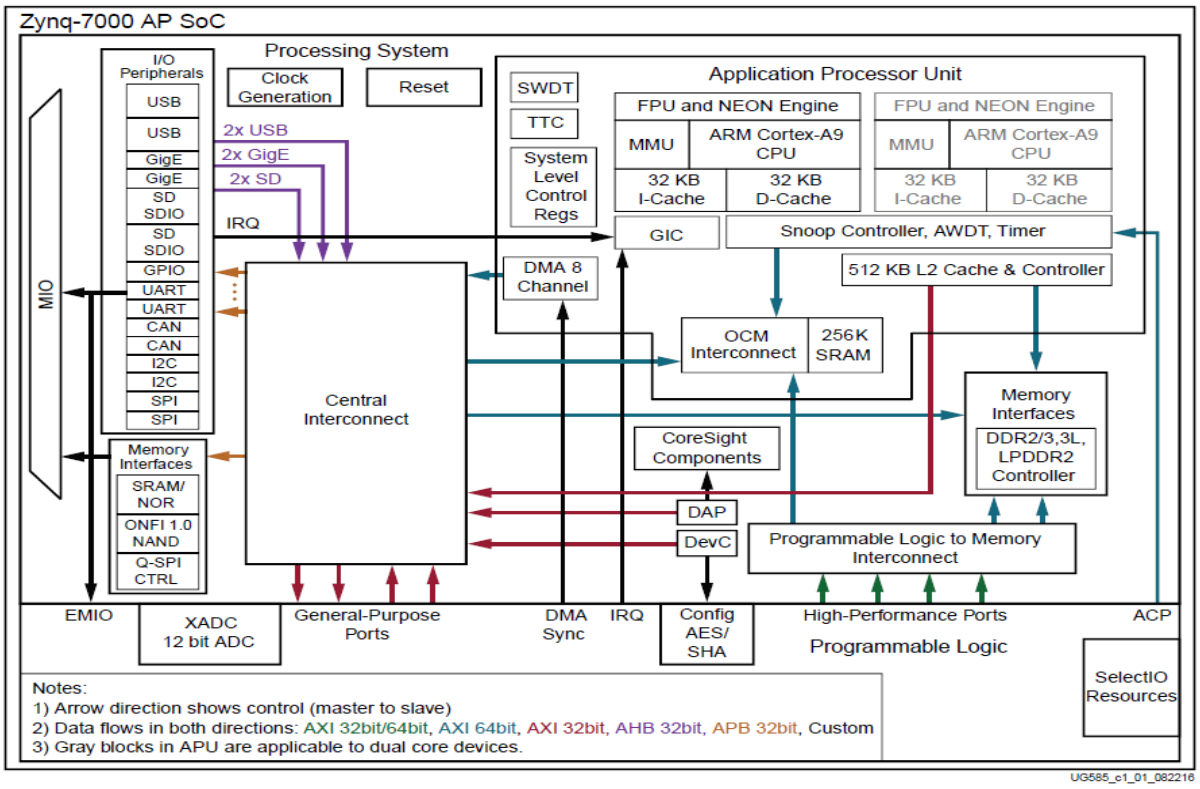

Figure 4-1 Zynq-7000 AP SoC Block Diagram

4.2. Introduction to ZC702

The ZC702 evaluation board for the XC7Z020 All Programmable SoC (AP SoC) provides a hardware environment for developing and evaluating designs targeting the Zynq® XC7Z020-1CLG484C device. The ZC702 board provides features common to many embedded processing systems, including DDR3 component memory, a tri-mode Ethernet PHY, general purpose I/O, and two UART interfaces. Other features can be supported using VITA-57 FPGA mezzanine cards (FMC) attached to either of two low pin count (LPC) FMC connectors.

The ZC702 board block diagram is shown in Figure 4-2.

The PS integrates two ARM Cortex™-A9 MP Core™ application processors, AMBA interconnect, internal memories, external memory interfaces, and peripherals including USB, Ethernet, SPI, SD/SDIO, I2C, CAN, UART, and GPIO. The PS runs independently of the PL and boots at power-up or reset.

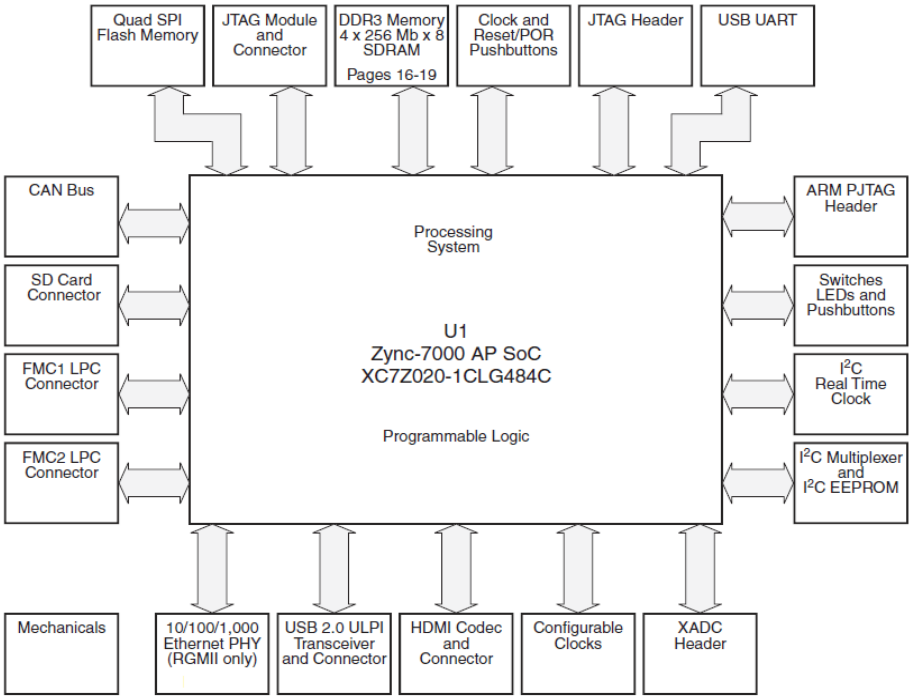

Figure 4-2 ZC702 Board Block Diagram

4.3. Look-Up Table (LUT)

The function generators in 7 series FPGAs are implemented as six-input look-up tables (LUTs). There are six independent inputs (A inputs - A1 to A6) and two independent outputs (O5 and O6) for each of the four function generators in a slice (A, B, C, and D). The function generators can implement:

- Any arbitrarily defined six-input Boolean function
- Two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs
- Two arbitrarily defined Boolean functions of 3 and 2 inputs or less

A six-input function uses:

- A1-A6 inputs
- O6 output

Two five-input or less functions use:

- A1–A5 inputs
- A6 driven High

- O5 and O6 outputs

4.4. CLB Overview

The 7 series configurable logic block (CLB) provides advanced, high-performance FPGA logic:

- Real 6-input look-up table (LUT) technology
- Dual LUT5 (5-input LUT) option
- Distributed Memory and Shift Register Logic capability
- Dedicated high-speed carry logic for arithmetic functions
- Wide multiplexers for efficient utilization

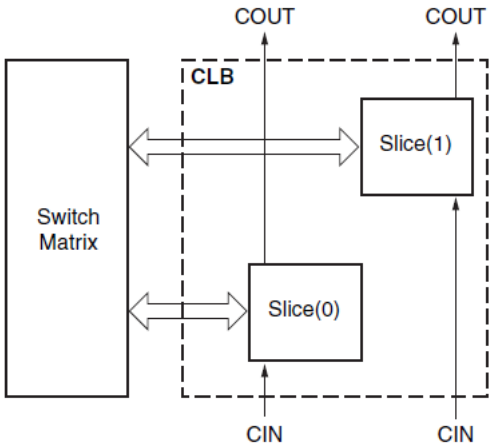

Figure 4-3 Arrangement of Slices within the CLB

CLBs are the main logic resources for implementing sequential as well as combinatorial circuits. Each CLB element is connected to a switch matrix for access to the general routing matrix as shown in Figure 4-3. A CLB element contains a pair of slices.

The LUTs in 7 series FPGAs can be configured as either a 6-input LUT with one output, or as two 5-input LUTs with separate outputs but common addresses or logic inputs. Each 5-input LUT output can optionally be registered in a flip-flop. Four such 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB. Four flip-flops per slice (one per LUT) can optionally be configured as latches. In that case, the remaining four flip-flops in that slice must remain unused.

The most important resources of the board are listed in Figure 4-4.

FPGA Components	Total Available
LUTs	53,200
I/Os	200
FPGA Logic Memory	
RAMB36E1	140
RAMB18E1	280
Slice registers	106,400

Figure 4-4 ZYNQ board important resources

4.5. Vivado Design Suite Overview

The Vivado® Design Suite is designed to improve productivity. This tool suite is architected to increase the overall productivity for designing, integrating, and implementing systems using the Xilinx® UltraScale™ and 7 series devices, Zynq® UltraScale+™ MPSoC device, and Zynq®-7000 SoC. Xilinx devices are now much larger and come with a variety of new technology, including stacked silicon interconnect (SSI) technology, up to 28 gigabyte (GB) high speed I/O interfaces, hardened microprocessors and peripherals, analog mixed signal, and more. These larger and more complex devices create multidimensional design challenges, when handled incorrectly, that can prevent the achievement of faster time-to-market and increased productivity. With the Vivado Design Suite, you can accelerate design implementation with place and route tools that analytically optimize for multiple and concurrent design metrics, such as timing, congestion, total wire length, utilization and power. The Vivado Design Suite provides you with design analysis capabilities at each design stage. This allows for design and tool setting modifications earlier in the design processes where they have less overall schedule impact, thus reducing design iterations and accelerating productivity.

4.6. SDK Overview

The Xilinx® Software Development Kit (SDK) provides an environment for creating software platforms and applications targeted for Xilinx embedded processors. SDK works with hardware designs created with Vivado®. SDK is based on the Eclipse open source standard. SDK features include:

- Feature-rich C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic Makefile generation
- Error navigation
- Well-integrated environment for seamless debugging and profiling of embedded targets
- Source code version control.

Chapter 5: Linux image

USRP and ZYNQ board interface using Linux image

In order to interface between USRP and ZYNQ board, the UHD ,explained in the chapter 3, is needed. UHD provides the necessary control used to transport user waveform samples to and from USRP hardware as well as control various parameters (e.g. sampling rate, center frequency, gains, etc.) of the radio. UHD can be installed on Linux, Windows, or a Mac. The used OS in this project is Linux. The B, N and X series of the USRPs can send and receive samples from a host computer as illustrated in 5-1.

In this project, ZYNQ board is used instead of the host computer. Thus, Linux image is needed on the SD card of the ZYNQ to boot from it. Also, the UHD and GNU Radio are to be installed on this image. Linux image is created by using Yocto project with Xilinx tools as explained in this chapter.

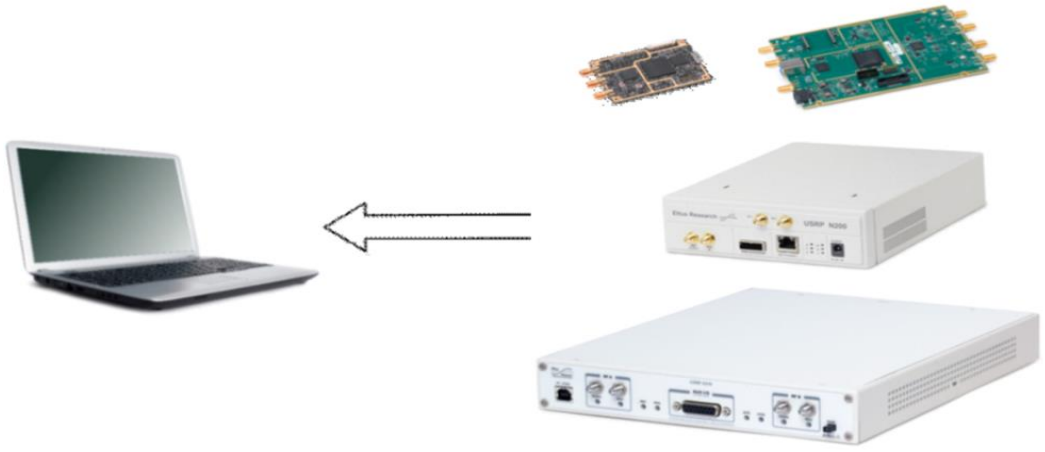

Figure 5-1 Interfacing between PC and USRP

5.1. Xilinx ZYNQ Linux kernel

Many pieces come together to boot Linux successfully on ZYNQ. The ZYNQ boot process begins with running code inside the Boot ROM. The boot ROM manages the early boot process by selecting the boot medium and quickly loading the First Stage Boot Loader (FSBL). The FSBL is created by Xilinx tools using information from the hardware project.

The Xilinx ZYNQ Linux kernel is based on the Linux kernel from kernel.org together with Xilinx additions (BSP and drivers). It is typically updated to stay close to the latest version from kernel.org on a regular basis. In general, the Xilinx Linux kernel for ZYNQ follows normal ARM Linux processes for building and running.

5.1.1. Xilinx ZYNQ Linux Support

Xilinx ZYNQ Linux is based on open source software (the kernel from kernel.org). Xilinx provides support for Xilinx specific parts of the Linux kernel (drivers and BSP). Xilinx also supports Linux through the Embedded Linux forum on [23] and mailing lists.

5.1.2. Using a Pre-Built Image/Release

Xilinx provides pre-built Linux releases which can be used in place of building the kernel and creating a boot image. The image is provided in [24] and for the 2015.2 version [25].

5.1.3. Kernel Details

5.1.3.1. The Board Support Package (BSP)

The primary code for the platform is in arch/arm/mach-ZYNQ directory of the kernel tree. The BSP contains some drivers and utilizes some existing drivers from arch/arm.

5.1.3.2. Device Tree

Device tree is a process by which the Linux kernel initializes itself based on the hardware platform. Device tree allows a single kernel image to run on multiple hardware platforms. A device tree file, named *.dts, is a text file that describes the hardware platform.

It is compiled into a device tree blob, *.dtb, which is loaded into memory before the Linux kernel is started. The Linux kernel then uses that device tree blob to initialize itself at runtime.

The process to create a device tree source (.dts) file and to compile a device tree blob (.dtb) from the DTS is described in Build Device Tree Blob. This describes the hardware which is readable by an operating system like Linux so that it doesn't need to hard code details of the machine.

Linux uses the Device Tree basically for platform identification, runtime configuration like bootargs and the device node population.

5.1.3.3. Device tree basics

Each driver or a module in the device tree is defined by the node and all its properties are defined under that node. Based on the driver it can have child nodes or parent node. For example, a device connected by SPI bus will have SPI bus controller as its parent node and that device will be one of the child nodes of SPI node. Root node is the parent for all the nodes.

Under the root node typically consists of:

1) CPUs node information

2) Memory information

3) Chosen can have configuration data like the kernel parameters string and the location of an initial image.

4) Aliases

5) Nodes which define the buses information

5.2. Yocto Project

The Yocto Project is an open-source collaboration project whose focus is developers of embedded Linux systems. Among other things, the Yocto Project uses a build host based on the OpenEmbedded (OE) project, which uses the BitBake tool, to construct complete Linux images. Two major components of the Yocto Project are maintained in conjunction with the OpenEmbedded project: BitBake, the build engine, and OpenEmbedded-Core, the core set of

recipes used to run the build process. The BitBake and OE components are combined together to form a reference build host, historically known as Poky.

The Yocto Project's industry-standard open source tools are used to create a customized Linux operating system for an embedded device and to boot the operating system in a virtual machine using QEMU. The Yocto Project, a Linux Foundation-sponsored open source project funded by major hardware companies and operating systems vendors, provides industry-class tools, methods, and metadata for building Linux systems.

The Yocto Project through the OpenEmbedded build system provides an open source development environment targeting the ARM, MIPS, PowerPC, and x86 architectures for a variety of platforms including x86-64 and emulated ones. Components from the Yocto Project can be used to design, develop, build, debug, simulate, and test the complete software stack using Linux, the X Window System, GTK+ frameworks, and Qt frameworks.

5.2.1. Introducing the Yocto Project

As a collaboration project, sometimes called an "umbrella" project, the Yocto Project incorporates many different disparate pieces of the development process. These pieces are referred to as projects within the overall Yocto Project and they include build tools, build instruction metadata called recipes, libraries, utilities, and graphical user interfaces (GUIs).

5.2.2. The OpenEmbedded Build System Workflow

The OpenEmbedded Build System is the build system used by the Yocto Project. This project is the upstream, generic, embedded distribution from which the Yocto Project derives its build system (Poky) and to which it contributes. At the heart of the build system is BitBake, the task executor.

The OpenEmbedded build system uses a workflow to accomplish image and SDK generation. Figure 5-2 overviews that workflow:

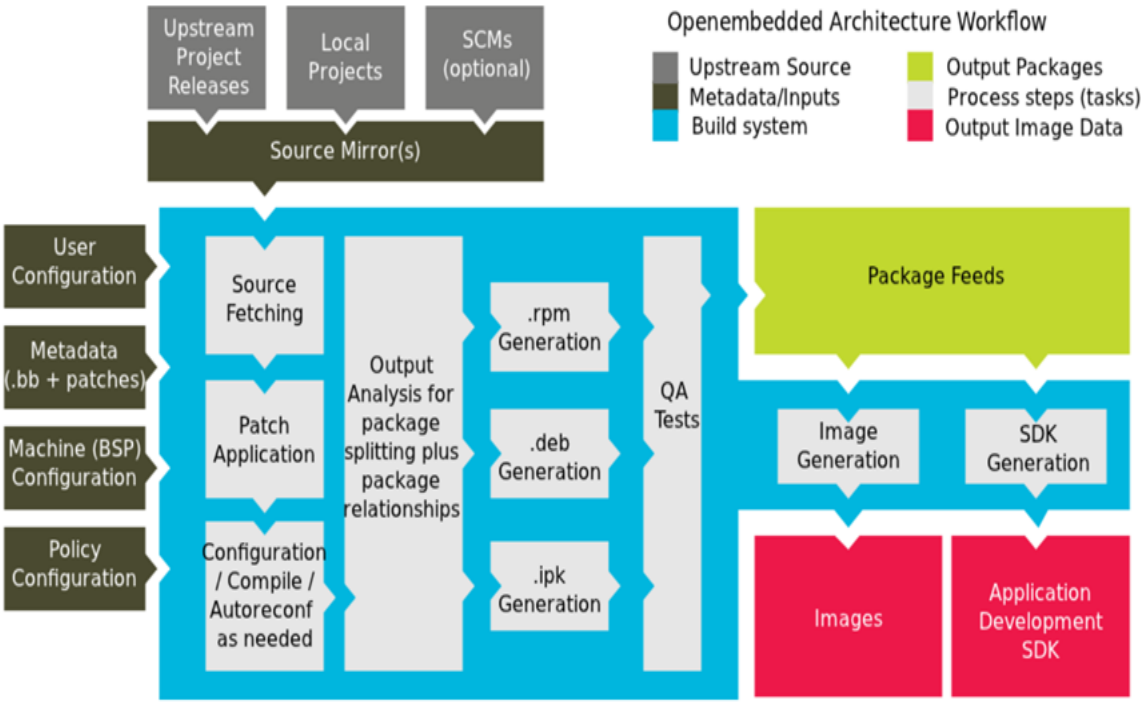

Figure 5-2 OpenEmbedded Build System Workflow

Following is a brief summary of the workflow:

1. Developers specify architecture, policies, patches and configuration details.
2. The build system fetches and downloads the source code from the specified location. The build system supports standard methods such as tarballs or source code repositories systems such as Git.
3. Once source code is downloaded, the build system extracts the sources into a local work area where patches are applied and common steps for configuring and compiling the software are run.
4. The build system then installs the software into a temporary staging area where the binary package format you select (DEB, RPM, or IPK) is used to roll up the software.
5. Different QA and sanity checks run throughout entire build process.
6. After the binaries are created, the build system generates a binary package feed that is used to create the final root file image.
7. The build system generates the file system image and a customized Extensible SDK (eSDSK) for application development in parallel.

In general, the build's workflow consists of several functional areas:

- **User Configuration:**
  metadata used to control the build process.
- **Metadata Layers**:
  Various layers that provide software, machine, and distro metadata.
- **Source Files:**
  Upstream releases, local projects, and SCMs.
- **Build System:**
  Processes under the control of BitBake. This block expands on how BitBake fetches source, applies patches, completes compilation, analyzes output for package generation, creates and tests packages, generates images, and generates cross-development tools.
- **Package Feeds:**
  Directories containing output packages (RPM, DEB or IPK), which are subsequently used in the construction of an image or Software Development Kit (SDK), produced by the build system. These feeds can also be copied and shared using a web server or other means to facilitate extending or updating existing images on devices at runtime if runtime package management is enabled.
- **Images:**
  Images produced by the workflow.
- **Application Development SDK:**
  Cross-development tools that are produced along with an image or separately with BitBake.

5.2.3. BitBake

BitBake is a core component of the Yocto Project and is used by the OpenEmbedded building system to build an embedded Linux system through configuration files collectively called metadata. BitBake is co-maintained by the Yocto Project and the OpenEmbedded project.

BitBake is used as a build tool, primarily by the OpenEmbedded and the Yocto project, to build Linux distributions. In short, BitBake is a building engine that works through recipes written in a specific format in order to perform sets of tasks, as shown in Figure 5-3.

BitBake reads recipes and follows them by fetching packages, building them, and incorporating the results into bootable images. A guideline for how to create the smallest possible project and extend it step by step and explain how BitBake works is provided here [26].

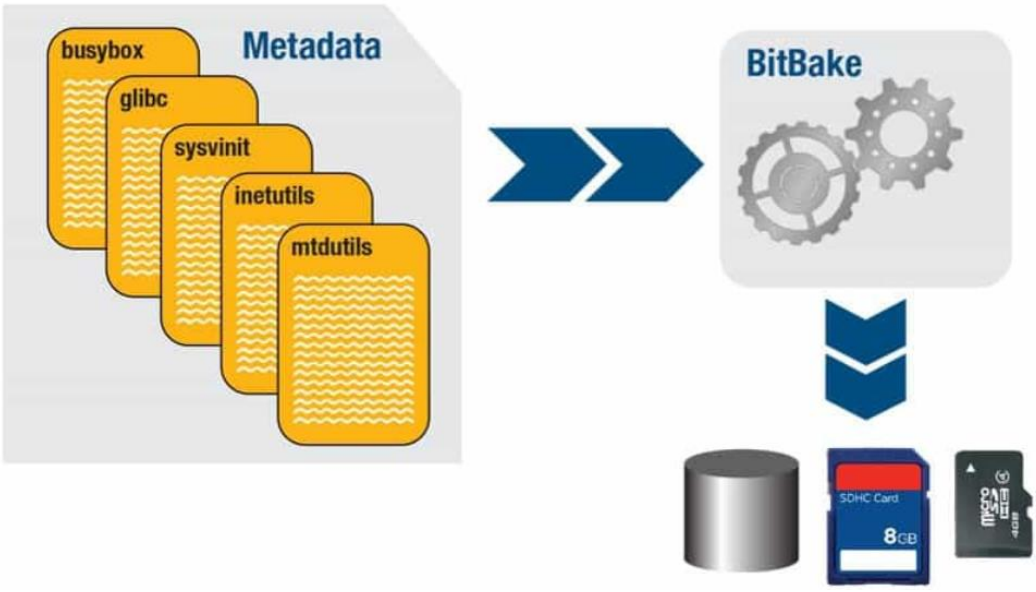

Figure 5-3 processing on metadata in BitBake

5.2.4. OpenEmbedded-Core

OpenEmbedded-Core (OE-Core) is a common layer of metadata (i.e. recipes, classes, and associated files) used by OpenEmbedded-derived systems, which includes the Yocto Project. The Yocto Project and the OpenEmbedded Project both maintain the OpenEmbedded-Core.

Historically, the Yocto Project integrated the OE-Core metadata throughout the Yocto Project source repository reference system (Poky). After Yocto Project Version 1.0, the Yocto Project and OpenEmbedded agreed to work together and share a common core set of metadata (OE-Core), which contained much of the functionality previously found in Poky. This collaboration achieved a long-standing OpenEmbedded objective for having a more tightly controlled and quality-assured core. The results also fit well with the Yocto Project objective of achieving a smaller number of fully featured tools as compared to many different ones.

Sharing a core set of metadata results in Poky as an integration layer on top of OE-Core. The Yocto Project combines various components such as BitBake, OE-Core, script “glue”, and documentation for its build system.

5.2.5. Poky

Poky is a reference build system for the Yocto Project. It includes BitBake, OpenEmbedded-Core, a board support package (BSP), and any other packages or layers incorporated into the build. The name Poky also refers to the default Linux distribution resulting from using the reference build system, which can be extremely minimal (core-image-minimal) or a full Linux system with a GUI (core-image-sato).

The Poky build system is considered a reference system for the entire project-a working example of the process in action. When the Yocto Project is downloaded, an instance of those tools, utilities, libraries, tool chain, and metadata are actually downloaded, which can be used to build the default system, as described here. That reference system and the reference distribution it creates are both named Poky. This can be used as a starting point to create a distribution, which of course can be named anything.

One item that all build systems requires is a tool chain: a compiler, assembler, linker, and other binary utilities necessary for creating binary executable files for a given architecture. Poky uses the GNU Compiler Collection (GCC), but other tool chains can be specified as well. Poky uses a technique known as cross-compilation: using a tool chain on one architecture to build binary executable files for a different architecture (for example, building an ARM distribution on an x86-based system). Developers often use cross-compilation in embedded systems development to take advantage of the host system's higher performance. Figure 5-4 illustrates what generally comprises Poky:

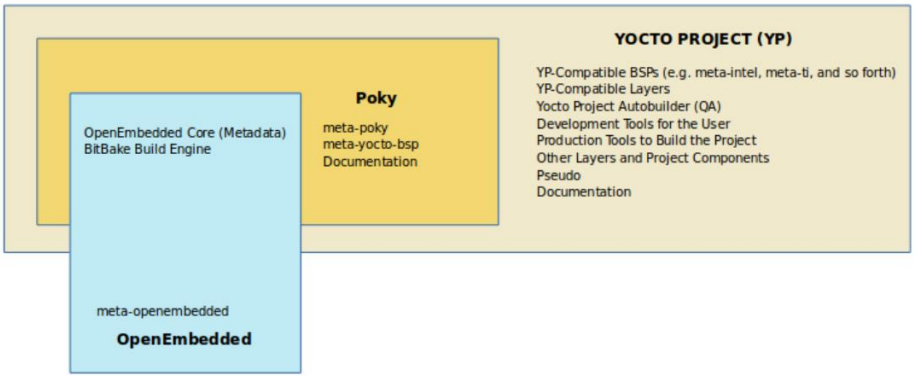

Figure 5-4 Yocto project's components

- BitBake is a task executor and scheduler that is the heart of the OpenEmbedded build system.
- meta-poky, which is Poky-specific metadata.
- meta-yocto-bsp, which are Yocto Project-specific Board Support Packages (BSPs).
- OpenEmbedded-Core (OE-Core) metadata, which includes shared configurations, global variable definitions, shared classes, packaging, and recipes. Classes define the encapsulation and inheritance of build logic. Recipes are the logical units of software and images to be built.
- Documentation, which contains the Yocto Project source files used to make the set of user manuals.

5.2.6. Metadata set

The metadata set is arranged in layers, such that each layer can provide separate functionality to the layers beneath it. The base layer is OpenEmbedded-Core, or oe-core, which provides recipes, classes, and associated functions that are common and necessary for all builds. Then builds can be customized by adding new layers on top of oe-core.

OpenEmbedded-Core is co-maintained by the Yocto Project and the OpenEmbedded project. One layer that separates the Yocto Project from OpenEmbedded is the meta-yocto layer, which provides the Poky distribution configuration and a core set of reference BSPs.

The OpenEmbedded project itself is a separate open source project with (largely) interchangeable recipes and similar goals to the Yocto Project, but different governance and scope.

5.2.7. Board support packages

A BSP contains the essential packages and drivers necessary for building Linux for a specific board or architecture. These are often maintained by the hardware manufacturers who make the boards. BSPs are the interface between the Linux operating system and the hardware that runs it. It is also possible to create BSPs for virtual machines.

5.2.8. Customizing the Build for Specific Hardware

Customizing the build for specific hardware is done by adding hardware layers into the Yocto Project development environment.

In general, layers are repositories that contain related sets of instructions and configurations that tell the Yocto Project what to do. Isolating related metadata into functionally specific layers facilitates modular development and makes it easier to reuse the layer metadata. By convention, layer names start with the string "meta-".

There are hundreds of meta-layers from the Yocto Project, OpenEmbedded, communities, and companies that should be manually cloned inside the project source directory to be used. These layers are provided at [27] and [28].

In general, three types of layer input exist:

- **Metadata (.bb + Patches):**

  Software layers containing user-supplied recipe files, patches, and append files. A good example of a software layer might be the meta-qt5 layer from the OpenEmbedded Layer Index. This layer is for version 5.0 of the popular Qt cross-platform application development framework for desktop, embedded and mobile. For example the meta-sdr layer which contains Software Defined Radio (SDR) related recipes.
- **Machine BSP Configuration:**

  Board Support Package (BSP) layers providing machine-specific configurations. This type of information is specific to particular target architecture. A good example of a BSP layer from the Poky Reference Distribution is the meta-yocto-bsp layer. Another example is meta-Xilinx layer which contains Xilinx hardware support.
- **Policy Configuration Distribution:**

  providing top-level or general policies for the images or SDKs being built for a particular distribution. For example, in the Poky Reference Distribution the distro layer is the meta-poky layer. Within the distro layer is a conf/distro directory that contains distro configuration files (e.g. poky.conf that contain many policy configurations for the Poky distribution).

The following are examples of layers used for supporting ZYNQ and USRP.

5.2.8.1. Meta-Xilinx

Support for Xilinx architectures (Zynq, ZynqMP and MicroBlaze) is available in Yocto/OE provided by either the OpenEmbedded Core or for additional and more complete support the meta-Xilinx layer. The meta-Xilinx layer also provides a number of BSPs for common boards which use Xilinx devices such as ZYNQ board used in this project.

Xilinx provides device and board information for the ZYNQ SoC for Yocto through the repository meta-Xilinx. This includes board information for the ZC702 Evaluation Kit. Meta-Xilinx provides Official support for Xilinx MicroBlaze and ZYNQ architectures as well as evaluation boards.

Boards Supported by this layer are:

- Xilinx ZC702 (ZYNQ)
- Xilinx KC705 Embedded TRD (MicroBlaze)
- Avnet/Digilent ZedBoard (ZYNQ)

5.2.8.2. Meta-Xilinx-Tools

Meta-Xilinx-Tools layer is a new layer available from the v2016.3 release. Meta-Xilinx-tools layer is a layer to support all bare metal components from Xilinx. This layer provides support for using Xilinx tools on supported architectures MicroBlaze, Zynq and ZynqMP. This layer depends on Xilinx SDK to be installed. This layer depends on XSCT being installed in the path. XSCT path has to be defined in local.conf

Meta-Xilinx-tools recipes depends on HDF to be provided. HDF_BASE can be set to git: or file:. HDF_PATH will be git repository or the path containing HDF.
This layer can be used via dependencies while creating the required Boot.bin. Basically the goal to build FSBL or PMU, etc. will depend on the use-case and Boot.bin will indicate these dependencies. Boot.bin is created using bootgen tool from Xilinx. Executing bootgen -bif_help will provide some detailed help on BIF attributes.

BIF file is required for generating Boot.bin, BIF is partitioned into Common BIF attributes and Partition BIF attributes. Attributes of BIF need to be specified in local.conf while using xilinx-bootbin.bbclass for generating Boot.bin.

5.2.8.3. Meta-SDR

It is a layer for software for Software Defined Radio (SDR) and related technologies. It is currently in development so that it will collect up various recipes for applications as well as GNURadio, UHD and some starter image recipes [29].

5.2.9. Hob

In an effort to make embedded Linux development easier, the Yocto Project provides a few different methods for working graphically. A relatively new addition to the project is called Hob, which provides a graphical front end to BitBake and the build process. Both are under continual development, complete with community user studies.

5.2.10. Open source license compliance

Complying with open source licenses is an extremely important part of any Linux development effort. One goal of the Yocto Project is to make compliance as easy as possible. It is quite easy to use the Yocto Project tools to create manifests and even to build entire source repositories, as well as filtering the build process to exclude packages that use specific licenses. The project is working with the Linux Foundation on its Open Compliance Program in relation to the Software Package Data Exchange® (SPDX™) specification.

5.2.11. EGLIBC

Embedded GLIBC (EGLIBC) is a variant of the GNU C Library (GLIBC) that is designed to work well on embedded systems. EGLIBC's goals include reduced footprint, configurable components, and better support for cross-compilation and cross-testing. EGLIBC is under the Yocto Project umbrella but is maintained within its own governance structure.

5.2.12. Application Development Toolkit

The Application Development Toolkit (ADT) enables systems developers to provide software development kits (SDKs) for the distributions they create using the Yocto Project tools, providing applications developers a way to develop against the software stacks provided by those systems developers.

The ADT includes a cross-compiling toolchain, debugging and profiling tools, and QEMU emulation and support scripts. The ADT also includes an Eclipse plug-in for those who like to work with integrated development environments (IDEs).

5.2.13. Other tools under the Yocto Project umbrella

Several other tools under the Yocto Project banner are:

- **Autobuilder:**
  Creates continuous automated builds of the Yocto Project tools, enabling automated Quality Assurance (QA) activities.
- **Cross-Prelink:**
  Provides prelinking for cross-compilation development environments, improving performance.
- **Pseudo:**
  Emulates root access, an essential part of building a bootable final image.
- **Swabber:**
  Detects when a cross-compilation build has been contaminated by host components.
- **Build Appliance:**
  Is a virtual machine that runs Hob, enabling those who use non-Linux build hosts to see the Yocto Project process firsthand. (Note: The Yocto Project build tools are currently supported on Linux only.)

5.3. Creating Linux image

5.3.1. Prepare and Boot Hardware:

There are two main ways to boot the board using an SD Card. The way used here is via Xilinx SDK bootgen and FSBL (First Stage Boot Loader). Another possibility is to use SPL (Secondary Program Loader) which has a similar, different set of files necessary.

5.3.1.1. FSBL Method

The general steps are highlighted below:

1. Generate the device tree (if not using the evaluation board) from Vivado as in [30].
2. Build the kernel image (also provides a uImage and u-boot binary) using bitbake
3. Wrap the file system image in u-boot headers [if necessary!]
4. Use Vivado to create a boot image from the ZYNQ fsbl
5. Copy all necessary files to SD card (FAT32, single partition)

Files Required:

- u-boot.elf from bitbake (recommended!) [31] or manually compiling the u-boot source code (not recommended!) [32] .
- top.bit from the synthesized hardware project.

Creating FSBL:

FSBL is created by opening up the Xilinx SDK with the HDF (hardware description file) and BIT (bitstream file) loaded. From here, an application project is created for the Zynq FSBL from File > New > Application Project, as shown in Figure 5-5 Creating FSBLFigure 5-5.

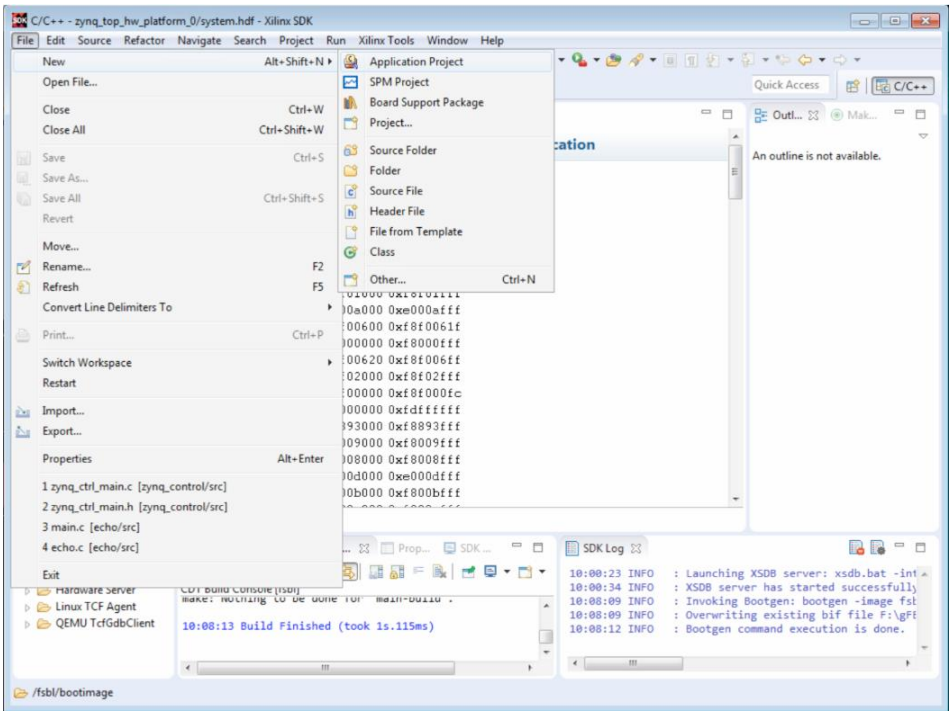

Figure 5-5 Creating FSBL

And then the ZYNQ FSBL project is selected as in Figure 5-6.

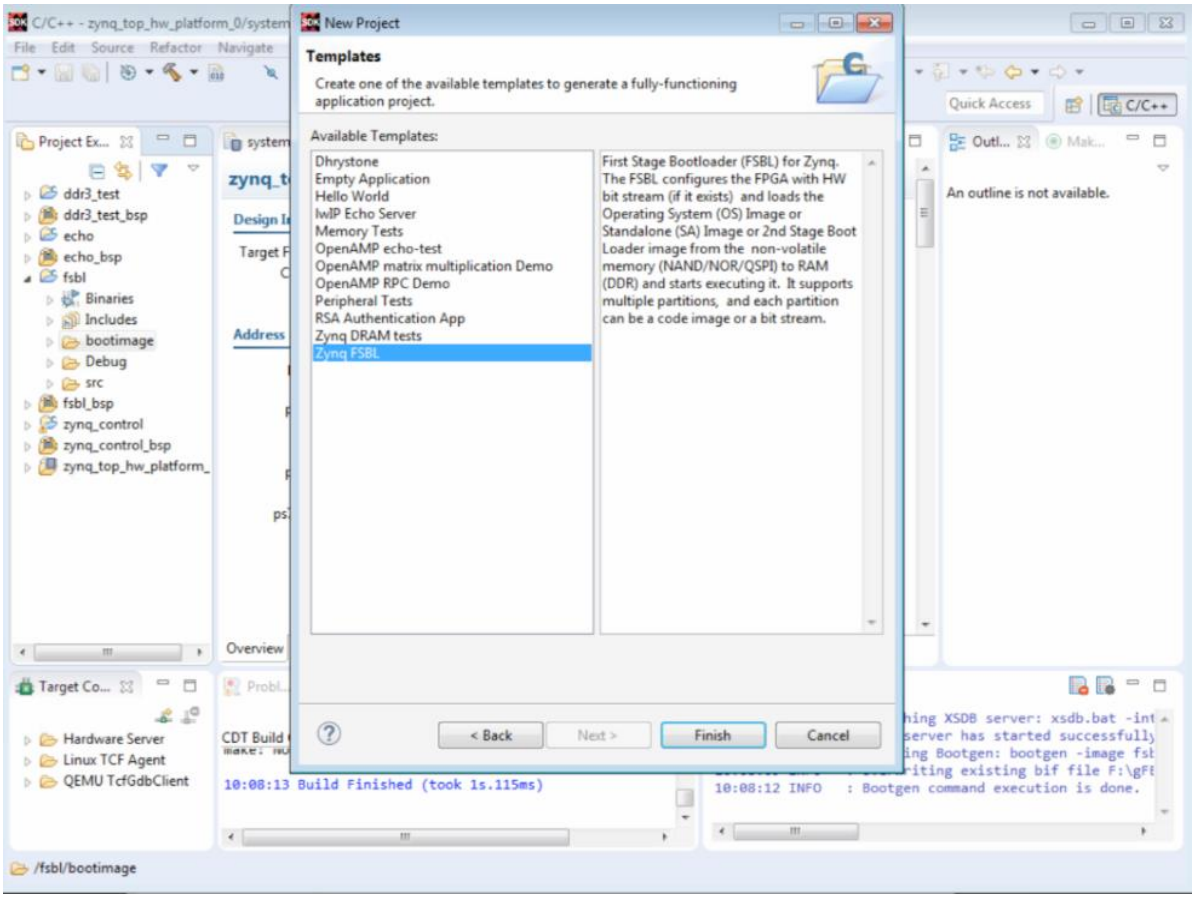

Figure 5-6 ZYNQ FSBL project

And then the SDK will automatically build the necessary files, including the fsbl.elf file and copy over the bitstream file. The fsbl.elf file and the bitstream file will most likely be found in the Debug/ folder under that project. This completes the first step.

Creating the Boot Image:

Now, the boot image BOOT.BIN is needed to be created with the necessary files loaded in the correct order. From the Xilinx SDK, Xilinx Tools > Create Boot Image, which brings up a dialog as in Figure 5-7.

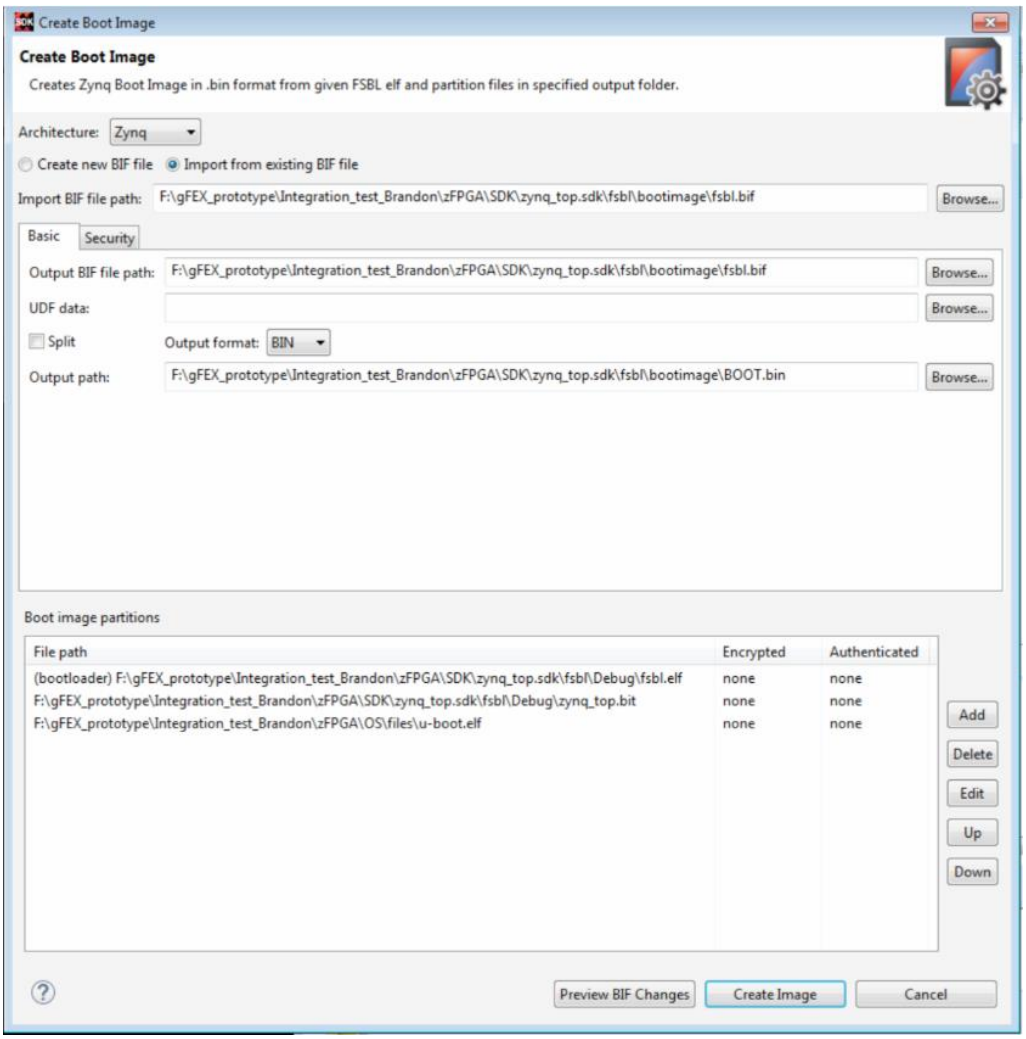

Figure 5-7 Create boot image dialogue

Figure 5-8 shows the files needed to be loaded in exactly this order and type.

File Type	File Name
bootloader	/path/to/sdk/fsbl_project/Debug/fsbl.elf
datafile	/path/to/sdk/fsbl_project/Debug/top.bit
datafile	/path/to/bitbake/files/u-boot.elf

Figure 5-8 Files needed in creating boot image

And then Create Image button is pressed. This will be created in the FSBL project under the bootimage/ folder. Files needed in creating boot image

Preparing the SD Card:

The SD Card should have the following files in Figure 5-9.

Originating	File Name	Description
Xilinx SDK	BOOT.BIN	Binary image containing the FSBL and U-Boot images produced by bootgen
bitbake	devicetree.dtb	Device tree binary blob used by Linux, loaded into memory by U-Boot
bitbake	uramdisk.image.gz	Ramdisk image used by Linux, loaded into memory by U-Boot
bitbake	u-boot.elf	U-Boot elf file used to create the BOOT.BIN image
bitbake	uImage	Linux kernel image, loaded into memory by U-Boot
Xilinx SDK	fsbl.elf	FSBL elf image used to create BOOT.BIN image

Figure 5-9 Files needed in the SD Card to boot Linux from

Chapter 6: Linux and Bare-Metal

Running Linux and Bare-Metal System on Both Zynq SoC Processors

6.1. Introduction

The Zynq-7000 AP SoC provides two Cortex-A9 processors that share common memory and peripherals. Asymmetric multiprocessing (AMP) is a mechanism that allows both processors to run their own operating systems or bare-metal applications with the possibility of loosely coupling those applications via shared resources. The Zynq-7000 All Programmable SoC contains two ARM Cortex-A9 processors that can be configured to concurrently run independent software stacks or executables. This chapter describes a method of starting up both processors, each running its own operating system and application, and allowing each processor to communicate with the other through shared memory. The reference design found in [49] includes the hardware and software necessary to build a reference design that runs both Cortex-A9 processors in an AMP configuration. CPU0 runs Linux and CPU1 run a bare-metal application. Care has been taken to prevent the CPUs from conflicting on shared hardware resources. This chapter also describes how to create a bootable solution and how to debug both CPUs.

6.2. Reference design

In the reference design [49], each of the two Cortex-A9 processors is configured to run its own software. CPU0 is configured to run Linux and CPU1 is configured to run a bare-metal application. This design has a version that was implemented using Xilinx Vivado 2014.4 on ZC702 board. The design was modified to be able to run on Xilinx Vivado 2015.2 which is the same version used in our project.

In this AMP example, the Linux operating system running on CPU0 is the master of the system and is responsible for:

- System initialization
- Controlling CPU1's startup
- Communicating with CPU1
- Interacting with the user

The bare-metal application running on CPU1 is responsible for:

- Managing a "heart beat" that can be monitored by Linux on CPU0
- Communicating with Linux on CPU0
- Servicing interrupts from a core in the programmable logic (PL)
- Communicating interrupt events to Linux running on CPU0

The Zynq SoC processing system (PS) includes resources that are both private to each CPU and shared by both CPUs. In running the design in an AMP configuration, care must be taken to prevent both CPUs from contending for these shared resources. Refer to Zynq-7000 All Programmable SoC Technical Reference Manual [8] for further information on shared and private resources.

Examples of some of the private resources are:

- L1 cache
- Private peripheral interrupts (PPIs)
- Memory management unit (MMU)
- Private timers

Examples of some of the shared resources are:

- Interrupt control distributor (ICD)
- DDR memory
- On-chip memory (OCM)
- Global timer
- Snoop control unit (SCU) and L2 cache

In this example, CPU0 is treated as the master and controls the shared resources. If CPU1 were to require control of a shared resource, it would have to communicate the request to CPU0 and let CPU0 control the resource. To keep the complexity of this reference design to a minimum, the bare-metal application running on CPU1 has been modified to limit access to the shared resources.

OCM is used by both processors to communicate to each other. When compared to DDR memory, OCM provides very high performance and low latency access from both processors. Deterministic access is further assured by disabling cache access to the OCM from both processors.

Actions taken by this design to prevent problems with the shared resources include:

1. DDR memory: Linux has only been made aware of memory at 0x00000000 to 0x2FFFFFFF. CPU1 uses memory from 0x30000000 to 0x3FFFFFFF for its bare-metal application.
2. L2 Cache: CPU1 does not use L2 cache.
3. ICD: Interrupts from the core in PL are routed to the PPI controller for CPU1. By using the PPI, CPU1 has the freedom to service interrupts without requiring access to the ICD.
4. Timer: CPU1 uses the private timer for the heartbeat.
5. OCM: Accesses to OCM are handled very carefully by each CPU to prevent contention. In the case of the heartbeat, only CPU1 writes the location and CPU0 reads the location. When data is sent from CPU1 to CPU0, only CPU1 writes the data value, then CPU1 sends a flag by setting a different address location in OCM. CPU0 in turn detects the flag, reads the data, and then clears the flag. Only CPU1 can set the flag, and only CPU0 can clear the flag.

For demonstration purposes only, a custom embedded core included with this example design is used to provide a simple interrupt source. An output from the ChipScope analyzer Virtual Input/Output (VIO) core is connected to this core, enabling the user to generate interrupts towards the PS at their leisure. Using the Chipscope VIO core provides more control over when an interrupt occurs and therefore makes it easier to measure the latency of interrupts. In a real-world design, however, this core would not exist and instead, the interrupt would be sourced by a truly functional piece of logic in the PL such as a direct memory access (DMA) engine.

6.2.1. Hardware

The PL contains a custom, embedded core connected to a synchronous output of a ChipScope analyzer VIO core as shown in Figure 6-1. The VIO core provides a mechanism for a user to interact with hardware from ChipScope analyzer.

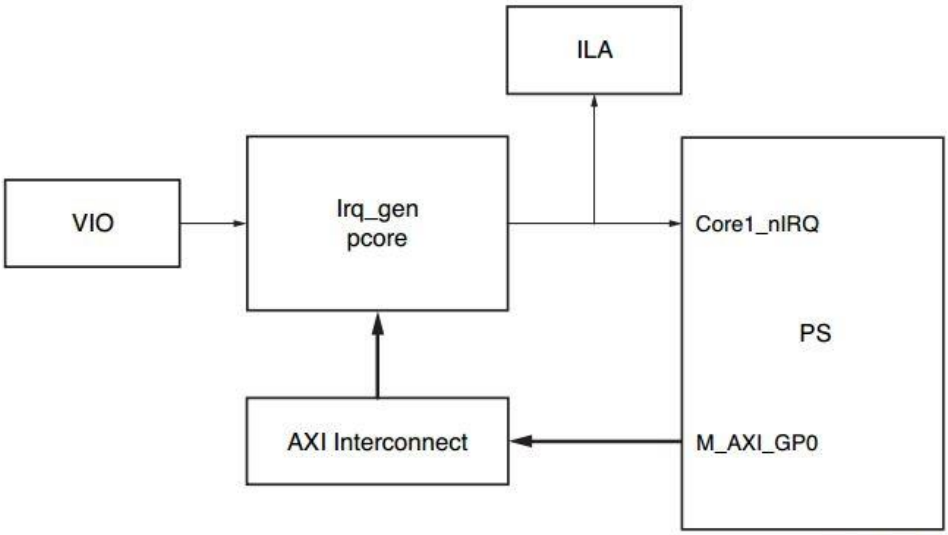

Figure 6-1 PL block diagram

In this design, when the VIO generates a pulse, the custom core forwards an interrupt to the PS Core1_nIRQ pin. The core is also connected to the PS master general purpose port (M_AXI_GP0) through an AXI Interconnect, allowing both CPU0 and CPU1 access to the control register within the core. CPU1 accesses the control register to clear the interrupt request (IRQ) during the interrupt service routine. CPU0 can optionally use the control register to create an interrupt towards CPU1. The Core1_nIRQ pin connects directly to CPU1's PPI block so there is no need to modify the configuration of the shared ICD. A ChipScope analyzer AXI monitor core is also included and allows the user to measure the latency of the IRQ being serviced.

6.2.2. Address map

In the PL, there is a single irq_gen embedded core that contains a single control register. The register is located at BASE + 0 (0x78600000). The table in Figure 6-2 contains a description of the IRQ_GEN control register.

Bit	Access	Description
[31:1]	R/W	Unused. Value written can be read.
[0]	R/W	IRQ Asserted • 0: IRQ is not asserted towards the PS. • 1: IRQ is asserted towards the PS. If the VIO_IRQ_TICK

Figure 6-2 IREQ_GEN control register

6.2.3. Software

The software can be broken down into three sections:

- First stage boot loader (FSBL)
- Linux operating system and applications for CPU0
- Bare-metal operating system and application for CPU1

6.2.4. FSBL

The FSBL always runs on CPU0. It is the first software application that is run after power-on reset of the PS. The FSBL is responsible for programming the PL and both application executable and linkable format (ELF) files to DDR memory. After loading the applications to DDR memory, the FSBL starts executing the first application that was loaded.

The FSBL first looks for a bit file. If a bit file is found, the FSBL writes it to the PL. Next, whether or not a bit file is found, the FSBL loads one application ELF into memory and executes it. This operating sequence does not support such an AMP configuration, so the FSBL must be modified. Within this AMP example's project files, the FSBL has been modified to continue searching for files and loading them into memory until it detects a file that has a load address of 0xFFFFFFF0. Upon detection, the FSBL downloads this last file

and jumps to the executable address of the first non-bit or non-boot file found (which is the application for CPU0). For details regarding how CPU1 starts up, refer to Zynq-7000 All Programmable SoC Technical Reference Manual [8].

6.2.5. Linux

The easiest way to use Linux in an AMP configuration is to configure Linux as symmetric multiprocessing (SMP) but restrict the number of available CPUs to 1. Such an approach ensures that Linux configures the ICD and SCU correctly for a multiple CPU environment. To create the Linux kernel, U-Boot, device tree, and the root file system ramdisk, refer to chapter 4. All generated files are available as part of the project files.

To instruct Linux to use only one CPU for SMP, the bootargs in the device tree is modified to add maxcpus=1. By default, the Linux .config is already setup to use SMP on the ZC702 demonstration board.

The device tree is also modified to reduce the amount of memory available to Linux to provide untouched memory space for CPU1's application.

6.2.6. Linux Applications

Two Linux applications that run on CPU0 are provided to interact with CPU1 that is running the bare-metal application. The first application, rwmem, provides a simple memory read and write access from Linux to OCM. This rwmem application is used to peek (read) and poke (write) addresses in OCM. As specific address locations are changed, CPU1 detects the changes and interacts in a specific way. The second application, softUart, provides a UART-style communication between Linux running on CPU0 and bare-metal running on CPU1 through predefined memory locations in OCM.

After the PS powers up and the internal boot ROM completes execution, CPU1 will have been redirected to a small piece of code in OCM at 0xFFFFFE00. This piece of code is a continuous loop that waits for an event, checks address location 0xFFFFFFF0 for a non-zero value and then continues the loop. If 0xFFFFFFF0 contains a non-zero value, CPU1 will jump to the fetched address.

CPU0 (running Linux) starts CPU1 (running bare-metal) by writing the value of 0x30000000 to address 0xFFFFFFF0 using the included rwmem application. Normally, CPU0 would need to run a set event (SEV) command to wake up CPU1. Because Linux, running on CPU0, is constantly servicing interrupts (another source of events), an SEV command is not necessary. When CPU1 wakes up, it reads the value 0x30000000 from address 0xFFFFFFF0 (written using the rwmem command) and then jumps to address 0x30000000. Note that the FSBL placed CPU1's ELF at 0x30000000.

The softUart application, which is also included in the design files, is run as a background task in Linux. When running, softUart continuously monitors shared OCM memory at locations 0xFFFF9000 (COMM_TX_FLAG_OFFSET) and 0xFFFF9004 (COMM_TX_DATA_OFFSET). Whenever a 1 is present at COMM_TX_FLAG_OFFSET, softUart reads the value found at COMM_TX_DATA_OFFSET and temporarily stores the value in a string array. When a value of 0x0A (\n) is received, the string array is displayed on STDOUT. Every time softUart reads a value from COMM_TX_DATA_OFFSET, it clears the COMM_TX_FLAG_OFFSET content. This clear signals to CPU1 that another character can be sent towards the softUart application running on Linux.

6.2.7. Bare-metal application code

The reference design has CPU1 running bare-metal application code. Linux, running on CPU0, is responsible for initializing shared resources and starting up CPU1.

The bare-metal board support package (BSP) named standalone_v5_1 that is part of the SDK 15.2 install includes support for the preprocessor define constant USE_AMP. This constant prevents the BSP from re-initializing the PS SCU that has previously been initialized by CPU0. One caveat of using the USE_AMP constant is that the MMU mapping is adjusted to create an alias of memory where the physical memory located at address 0x20000000 is virtually mapped to 0x00000000. This remapping is done in the BSP file boot.s. The re-mapping is not necessary for this design. A modified version of the BSP is included in the reference design to remove the re-mapping when USE_AMP is set.

Within this AMP reference design, no Zynq UARTs are used by the bare-metal application. Instead, the application running on CPU1 contains its own outbyte() function that is used to communicate via OCM to a software UART running in a Linux application on CPU0. By adding the outbyte() function, all stdout functionality of the standalone BSP is intact, allowing functions such as xil_printf() to be used.

To prevent shared resource conflicts, the bare-metal application running on CPU1 must be careful not to access resources such as the SCU. Linux disables cache access to the OCM. However, the default standalone BSP would attempt to enable cache for OCM and therefore conflict with Linux. When used in an AMP configuration, the function XIL_SetTlbAttributes() is used in the CPU1's main() application function to disable cache on OCM. The XIL_SetTlbAttributes() function has been modified in the included source code such that it only flushes L1 cache and leaves L2 cache untouched to prevent access to the SCU where L2 cache is controlled.

If the bare-metal code running on CPU1 requires control of L2 cache, a communications channel must be created allowing the bare-metal code to request Linux to make the necessary changes to the SCU. This action is beyond the scope or requirement for this example design so care has been taken to prevent SCU access directly from the bare-metal code.

6.2.8. CPU1 applications

CPU1's application is located in memory starting at address 0x30000000. The linker script is used to set the starting address.

CPU1's application does the following:

1. Configures the MMU to disable cache for OCM accesses in the address range of 0xFFFF0000 to 0xFFFFFFFF. The address mapping of the OCM is untouched so OCM exists at addresses 0x00000000–0x0002FFFF and addresses 0xFFFF0000–0xFFFFFFFF. Only the high 64 KB of OCM is used by the design so cache is disabled on addresses 0xFFFF0000–0xFFFFFFFF.
2. Initializes the PPI interrupt controller and interrupt subsystem.

3. Increments an OCM location (COMM_VAL). This OCM location is referred to as the Heartbeat.
4. Sleeps for one second.
5. CPU1's main() function repeats step 3 and step 4 continuously.
6. As interrupts are detected, an interrupt service routine in the background clears the interrupt status of the embedded core and prints a string. The output from the print statement is redirected to use the OCM COMM_TX_FLAG_OFFSET and COMM_TX_DATA_OFFSET locations. In turn, Linux consumes the OCM data and prints the string to the Linux console.

6.2.9. Design Files

The modified reference design contains these files:

- Vivado project
- SDK source files for Linux and CPU1 applications
- Generated files including:
  - Bit file
  - All files for the SD card
  - Application ELF files for Linux and CPU1
  - BOOT.BIN build scripts
  - Modified bare-metal BSP
  - Modified FSBL
  - Modified devicetree.dts and devicetree.dtb

6.2.10. Generating Hardware

This section describes the creation of the hardware design. To implement the design and export it to SDK:

a. Create a new directory called 'design/work'
b. Open Vivado 2015.2, In the Tcl command window, write 'cd design/work'

c. Create, build, and export the hardware design to SDK by running the included script using tcl:

run the tcl command 'source ../src/scripts/create_proj_702.tcl'.

This script does the following:

- Create a new project
- Set the properties of the project such as part used and board used
- Set Vivado to use the included IP repository in src/pcores. This repository includes the custom IP that can create interrupts towards the PS using either a register or chipscope VIO
- Create the IP design using the tcl script “src/scripts/create_bd_702.tcl”. This script was originally created after creating the IP design manually, and then issuing the command 'write_bd_tcl ../src/scripts/create_bd_702.tcl'.
- Validate and save the IPI design
- Create the top level HDL wrapper and add it to the project. (Same as navigating to 'Project Manager', right clicking on design_1.bd, and selecting 'Create HDL Wrapper')
- Create bitstream. This command will recognize that the design hasn't been implemented yet and will run 'generate files' on design_1.bd, synthesis, implementation, and bitstream generation.
- Exports the hdf file to SDK and launch SDK. The hdf file contains system information, PS startup information, and bitfile information

When the script finishes running it will open SDK. In the SDK workspace a hardware platform project is created automatically.

6.2.11. Generating Applications

6.2.11.1. Configuring SDK

The modified standalone BSP files (used by the bare-metal application) are included in design/work/project_1/project_1.sdk/app_cpu1_bsp and the modified FSBL file is included in design/work/bootgen.

6.2.11.2. Creating Bare-Metal Application for CPU1

The instructions in this section create the application ELF that runs on CPU1 after the FSBL loads the applications to DDR memory. This step is slightly different than creating the application for CPU0 because CPU1 uses the customized BSP (like app_cpu1_bsp). This design prevents CPU1 from accessing shared resources such as the ICD or SCU. The application has already been compiled and is available at design\generated_files\SDK_apps\app_cpu1.elf.

1. Within SDK, create the BSP using the customized standalone BSP from the repository that was included with the design.
   a. Select File > New > board_support_package.
   b. Change the project name to app_cpu1_bsp.
   c. Change the CPU to ps7_cortexa9_1.
   d. Select the board support package OS standalone
   e. Click Finish.
   f. In the board support package settings, select overview > standalone and change both stdin and stdout to none (Figure 6-3).

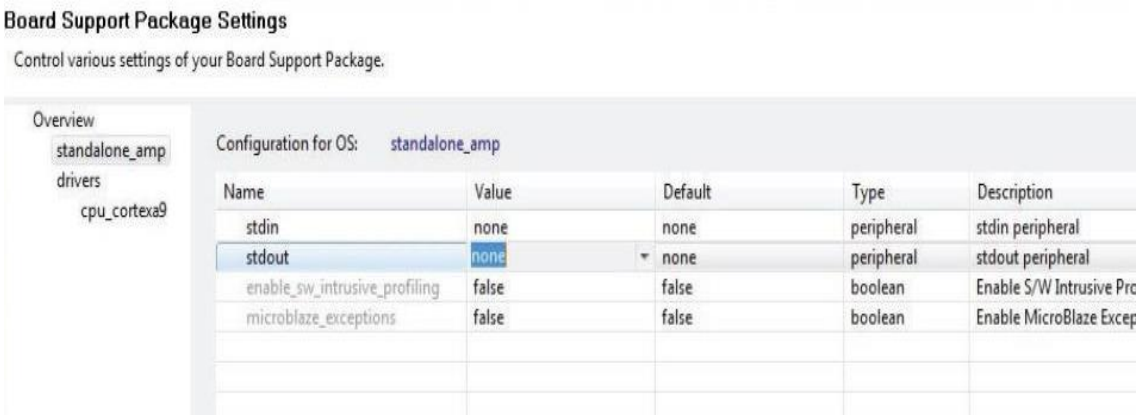

Figure 6-3 set NO stdin or stdout

g. Select Overview > drivers > ps7_cortexa9_1 and change the extra_compiler_flags value to contain '-g -DUSE_AMP=1 -DSTDOUT_REDIR=1' (Figure 6-4).

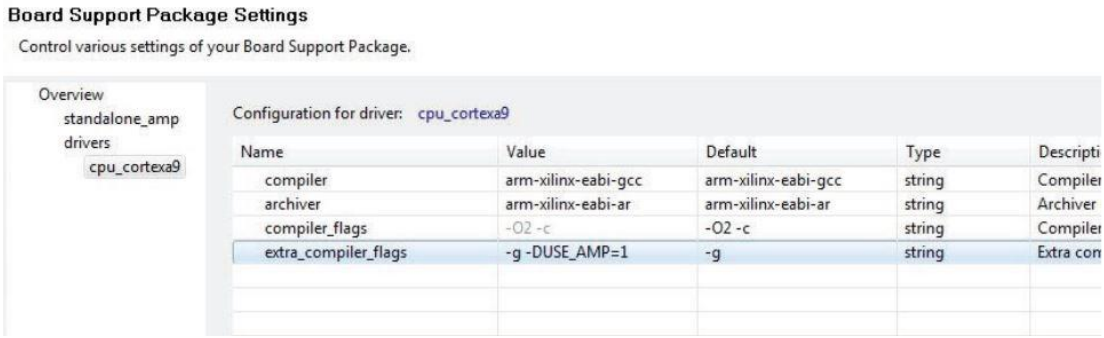

Figure 6-4 CPU1 BSP add USE_AMP

h. Select OK.

2. Create the bare-metal application that will be running on CPU1 and import the included software:
   a. Select File > new > application_project.
   b. Enter the project name app_cpu1.
   c. Change processor to ps7_cortexa9_1.
   d. Change board support package to Use existing and select app_cpu1_bsp.
   e. Click Next
   f. Choose the Empty Application template.
   g. Click Finish.
   h. In SDK's Project Explorer tab, expand app_cpu1 and right click on the src folder.
   i. Select Import.
   j. Select General > File_System.
   k. Click Next.
   l. Browse to and select the included directory design/src/apps/app_cpu1.
   m. In the left window pane, select the app_cpu1 folder but do not add a checkmark. In the right window pane, select all files.
   n. Click Finish and select yes to overwrite lscript.ld.
3. After SDK completes compiling the new application, the ELF is available at design/work/project_1/project_1.sdk/app_cpu1/Debug/app_cpu1.elf.

6.2.11.3. Creating Linux Application RWMEM

This utility provides simple read and write accesses to memory locations from the Linux console much like the mrd and mwr commands within Xilinx Microprocessor Debug (XMD).

This application has already been compiled and is available at design\generated_files\SDK_apps\rwmem.elf.

1. Create a blank Linux application:
    a. Select File > new > Application_Project.
    b. Set the Project Name to rwmem.
    c. Change the OS Platform to Linux.
    d. Click Next
    e. Select the Linux Empty Application template
    f. Click Finish.
2. Import the included rwmem source.
    a. In SDK's Project Explorer tab, expand rwmem and right click on the src folder.
    b. Select **Import**.
    c. Select General > File_System
    d. Browse to and select the included directory design\src\apps\rwmem.
    e. Click **OK**.
3. In the right window pane, select rwmem.c. Do not select anything in the left window pane. Click **Finish**.
4. After SDK completes compiling the new application, the ELF file is available at design/work/project_1/project_1.sdk/rwmem/Debug/rwmem.elf.

6.2.11.4. Creating Linux Application Soft UART

The Soft UART application runs on Linux and continuously monitors OCM to receive data from CPU1. The data is echoed to the Linux terminal. This application has already been compiled and is available at design\generated_files\SDK_apps\softUart.elf.

1. Follow the same steps as Creating Linux Application RWMEM, but name the project softUart and import the source from the included design\src\apps\softUart.
2. After SDK completes compiling the new application, the ELF file is available at design/work/project_1/project_1.sdk/softUart/Debug/softUart.elf.

6.2.11.5. Creating Linux Kernel

Refer to chapter 5 for instructions on how to compile the kernel. A copy of a pre-compiled kernel is included at AMP_exampe_image_yocto/uImage.

6.2.11.6. Creating Linux Device Tree

Refer to chapter 5 for instructions on how to compile the device tree. The device tree needs to be changed to instruct Linux SMP to only use one CPU and to decrease the amount of memory available to Linux. A copy of the modified and compiled devicetree.dtb is included at AMP_exampe_image_yocto/devicetree.dtb .

The commands used to modify the device tree are listed here:

1. Go to: <yocto_dir>/poky-jethro/meta-xilinx/conf/machine/boards/zc702/zc702-zynq7-board.dtsi
2. Modify the device tree to reduce the memory. The memory entry should be:
   memory {
   device_type = "memory";
   reg = <0x00000000 0x30000000>;
   };
3. Set the maximum number of CPUs to 1 by adding maxcpus=1 to the bootargs assignment: bootargs = "console=ttyPS0,115200 maxcpus=1 mem=768M root=/dev/ram rw earlyprintk";

6.2.11.7. Creating U-Boot

Refer to chapter 5 for instructions on how to compile U-Boot. A copy of U-Boot has already been compiled and is included at AMP_exampe_image_yocto/u-boot.elf.

6.2.11.8. Acquiring Root File System

Refer to chapter 5 for instructions on how to make the root file system. A copy of the ramdisk is included at final_rootfs\uramdisk.image.gz.

6.2.11.9. Generating Boot File

The boot file BOOT.BIN normally contains the FSBL, FPGA bit file, and the ELF for the application that runs on CPU0 (U-Boot). The design files contain a batch file, and a BootGen configuration file.

The configuration file contains the names of the files that are copied to DDR memory. The order of these files is important. For this design, the order is:

1. FSBL ELF.
2. CPU0 application.
3. CPU1 application.

A precompiled version of BOOT.BIN is available at design/work/bootgen/BOOT.BIN.

The boot file must be named BOOT.BIN.

1. Copy the included directory design\src\bootgen to design\work\bootgen. This directory includes the BootGen batch file (createBoot.bat), a BIF file (bootimage.bif), and a binary file (cpu1_bootvec.bin) that only contains the hexadecimal value 0xFFFFFF00 (swapped for little endian is 0x00, 0xFF, 0xFF, 0xFF). The FSBL recognizes this file's load address of 0xFFFFFFF0 as configured in bootimage.bif. It triggers the FSBL to stop loading ELF or bin files and start running the first ELF that was downloaded.
2. Copy the compiled FSBL ELF to design/work/bootgen/zynq_fsbl.elf
3. Copy the bit file download.bit into design\work\bootgen.
4. Copy u-boot.elf that was created from the information in chapter 4 into design\work\bootgen.
5. Copy the generated bare-metal application for CPU1 into design\work\bootgen.
6. Within SDK, select Xilinx_tools->launch_shell. A new command shell is started with an environment pointing to the SDK and Vivado tools.

7. In the command prompt, change the directory to design\work\bootgen.
8. Run the createBoot.bat file. This batch file runs bootgen and uses bootimage.bif for input. The bif is used to package the fsbl, u-boot, cpu1 app, and fpga bit file into boot.bin. This creates the boot file BOOT.BIN is created in the current directory.

6.2.12. Copying Files to SD Card

The Zynq AP SoC requires these files to be present on the SD card in order to boot Linux:

1. BOOT.BIN (contains the FSBL, BIT file, U-Boot, and application for CPU1)
2. uramdisk.image.gz (ramdisk file system extracted to memory by U-Boot)
3. devicetree.dtb (used by U-Boot and Linux for device information)
4. uImage (Linux kernel loaded and executed by U-Boot)
5. u-boot.img (bootloader)

To copy the files to the SD card:

1. devicetree.dtb: This can be the user's devicetree.dtb created from the instructions in chapter 4 or copied from the AMP_exampe_image_yocto/devicetree.dtb included in the reference design.
2. uramdisk.image.gz: This can be the user's own file created from the instructions in chapter 4 or copied from the final_rootfs/uramdisk.image.gz included in the reference design.
3. uImage: This can be the user's own file created from the instructions in chapter 4 or copied from the included AMP_exampe_image_yocto/uImage included in the reference design.
4. BOOT.BIN: This is the user's design\work\bootgen\BOOT.BIN created from the above steps.

6.2.12.1. Running the Design

To setup the hardware, follow the board setup instructions in the "TRD Demonstration Procedure" section of Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit. The hardware setup configures the ZC702 demonstration board to boot from the SD card.

The terminal program should be configured to listen to the correct COM port with a baud rate of 115200. When the design is powered up, the board boots. CPU0 then starts running U-Boot and boots Linux. When booting from the SD card, the system can take up to 18 seconds before an output appears on the UART. This UART is dependent upon a third-party driver.

During boot, the PS bootloader detects that the mode pins have been configured to boot from the SD card. In turn, the PS bootloader opens the BOOT.BIN file and searches for the block of data that has been flagged with bootloader. As seen in the bootimage.bif file, amp_fsbl.elf has this flag. The bootloader loads this file into DDR memory and starts running it. In turn, the FSBL loads the BIT file, U-Boot ELF, CPU1's ELF, and then the dummy file cpu1_bootvec.bin. At this point, the FSBL running on CPU0 jumps to the execution address of the first application that was loaded after the FSBL.

The soft UART application is started in linux with the command softUart.elf &. This command runs softUart in the background. SoftUart continues to monitor shared OCM memory at locations 0xFFFF9000(COMM_TX_FLAG_OFFSET) and 0xFFFF9004 (COMM_TX_DATA_OFFSET). Whenever a 1 is present at COMM_TX_FLAG_OFFSET, softUart reads the value found at COMM_TX_DATA_OFFSET and temporarily stores the value in a string array. When a value of 0x0A (\n) is received, the string array is displayed on STDOUT. Every time softUart reads a value from COMM_TX_DATA_OFFSET, it clears the COMM_TX_FLAG_OFFSET content, which signals to CPU1 that another character can be sent towards the softUart application running on Linux.

The second CPU is started with the command rwmem.elf 0xfffffff0 0x30000000. Location 0xFFFF8000 should start incrementing every second and can be viewed by using the command rwmem.elf 0xffff8000. At this point, the bare-metal app running on CPU1 only prints to the softUart when an interrupt is received from the PL. The PL contains a custom core that generates an interrupt from an input signal to the core or from a register within the core.

A ChipScope analyzer VIO core is connected to the input of the custom core. Thus, either software or the ChipScope analyzer VIO core can create an interrupt to CPU1. A ChipScope Integrated Logic Analyzer (ILA) core is also located in the design to monitor the IRQ signal. To force an interrupt to CPU1, the Linux command rwmem.elf 0x78600000 1 is entered as shown in Figure 6-5. Refer to the Address Map to see the register bit that was just set. By setting this bit, an interrupt was sent to CPU1 and CPU1's interrupt service routine printed "CPU1: IRQ clr 0" to the soft UART application via the OCM memory.

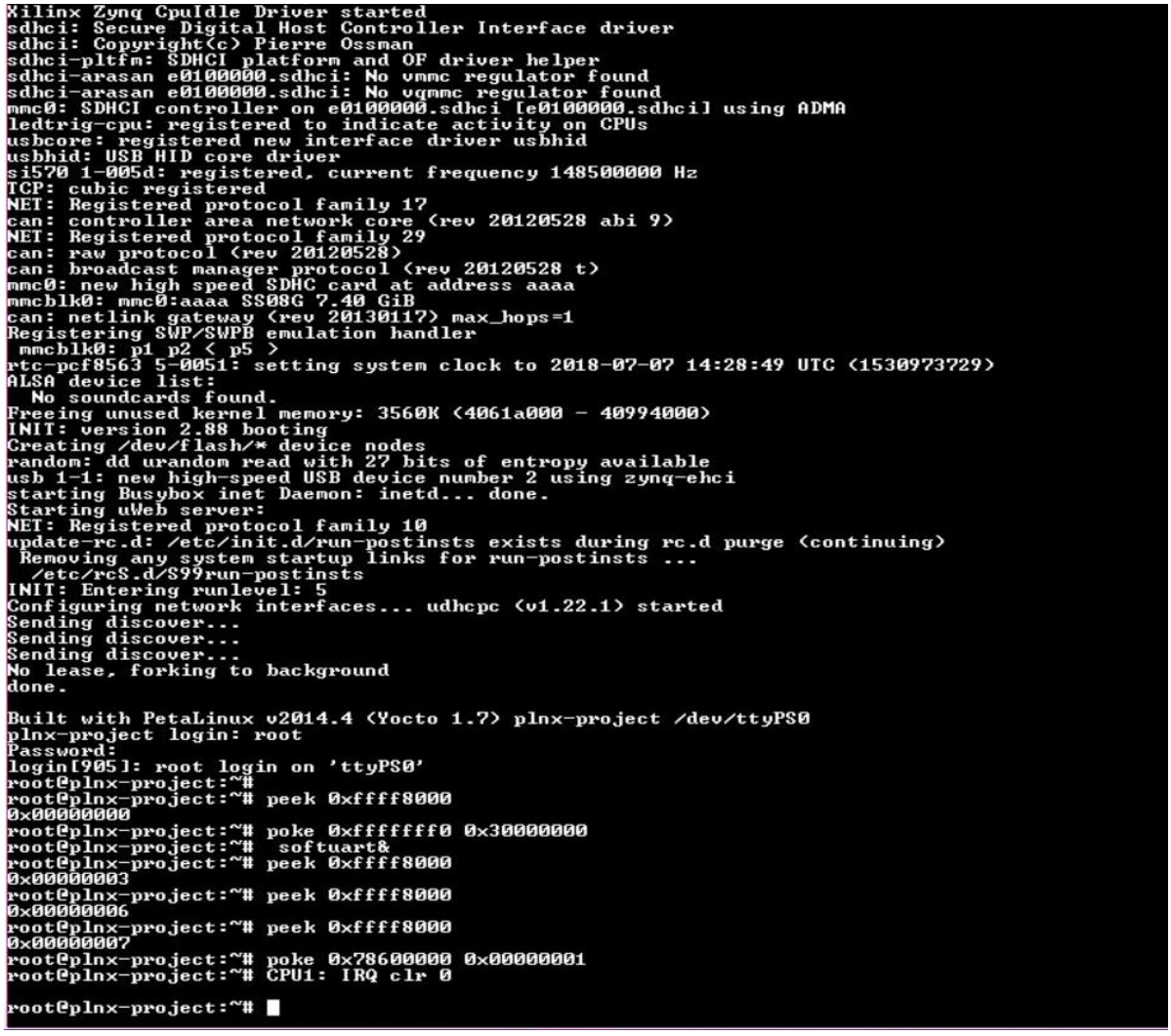

Figure 6-5 Consol Output

The bare-metal application that services the interrupt is located in DDR memory. When the first interrupt occurs, CPU1 is instructed to jump to the service routine. This jump causes the instructions located in DDR memory to be read into cache and executed. During execution, the service routine finishes by clearing the interrupt signal being generated by the embedded core. After the first IRQ occurs, the service routine is located in cache so fetches of the instructions for the routine are sourced by the cache instead of the slower, less deterministic DDR memory. The time difference between the first and later interrupt services could be reduced by moving the service routine into non-cached OCM.

6.2.13. Debugging the design

SDK can be used to connect and debug the application running on CPU1. XMD provides a command shell and GNU debugger (GDB) server that connects to the CPU via the JTAG cable. Normally, SDK automatically starts XMD in the background when starting to debug an application. For this example design, XMD is manually started to connect to CPU1. SDK is then instructed to connect to the XMD GDB server during debug.

Because FSBL was used to boot the design, there is no need to re-initialize the PS registers.

Care must be taken not to reset the full PS to not upset Linux running on CPU0.

1. Connect the platform cable to the ZC702 board. Ensure the jumper options are configured for the correct debug cable.
2. From SDK, start XMD and connect to CPU1 (Figure 6-6):
    a. In SDK, open a Xilinx command shell by selecting Xilinx_Tools > Launch_shell.
    b. In the new command shell, enter xmd.
    c. At the XMD prompt, enter the command connect arm hw -debugdevice cpunr 2.
    d. XMD should respond with the TCP port number 1234.

A GDB server is now running and listening to TCP port 1234. When XMD connects, CPU1 is halted.

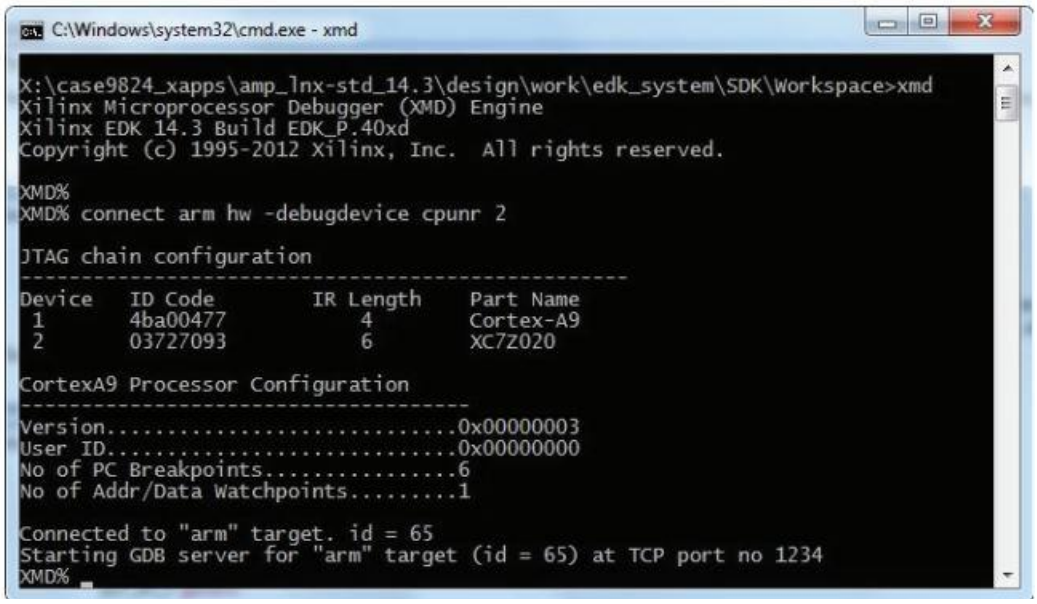

Figure 6-6 Connect XMD to CPU1

3. Start debugging CPU1 in SDK:
   a. In the SDK project explorer window, right click app_cpu1 and select debug_as >debug_configurations.
   b. Highlight Xilinx C/C++ ELF and select the New launch configuration icon at the top left.
   c. The configuration name is automatically set to app_cpu1 Debug (Figure 6-7).

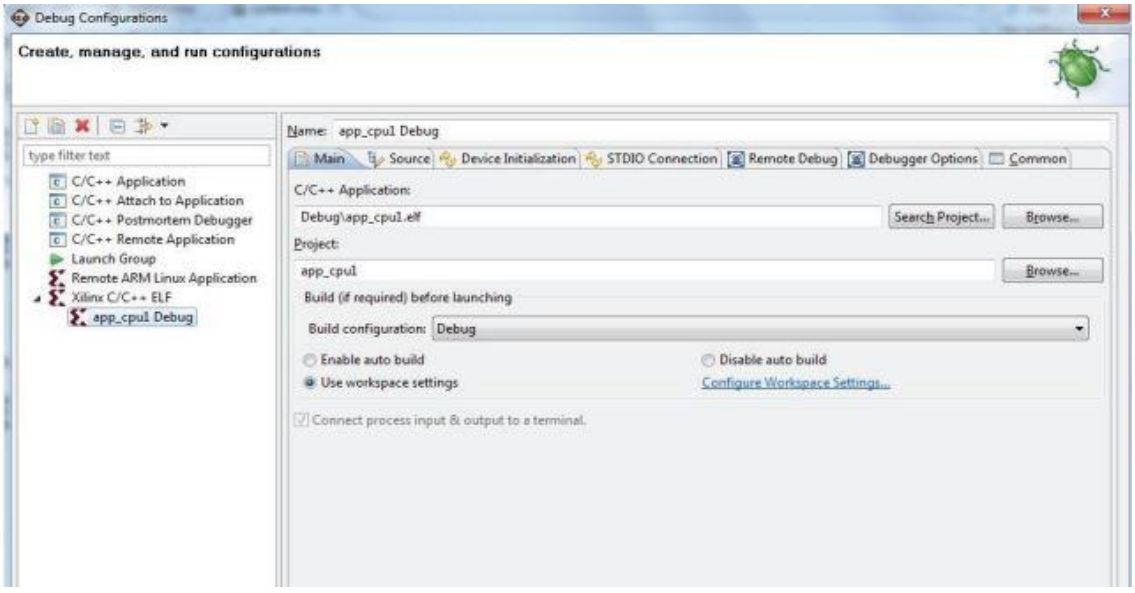

Figure 6-7 CPU1 debug configuration

d. Select the Device Initialization tab and delete the contents of the Path to initialization TCL file field. Initialization has already been done by Linux and FSBL.
e. Select the Remote Debug tab.
f. Instruct SDK to connect to the externally created GDB server by checking the box next to Connect to gdbserver on a different machine. The IP address should default to localhost and the port should be 1234 (Figure 6-8).

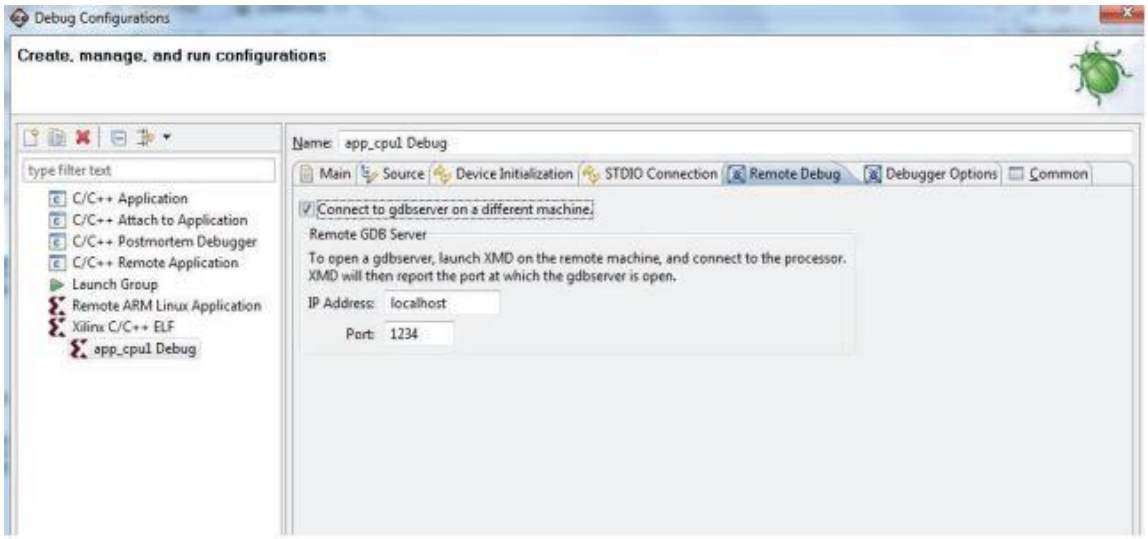

Figure 6-8 CPU1 remote debug configuration

g. Click **Apply**.
h. Click **Debug**. Click **Yes** to confirm the perspective switch.
i. The application is downloaded and then executed (the ELF download could have been disabled in the Device Initialization Tab). The application stops at a breakpoint at the first executable line in main().
j. Press **resume**, **single step**, and other buttons to continue running the application on CPU1.

6.3. WIFI transmitter and receiver system designs

In the Transmitter and Receiver Systems design, each of the two Cortex-A9 processors is also configured to run its own software as in the reference design [49]. CPU0 is configured to run Linux and CPU1 is configured to run a bare-metal application.

In those AMP designs, the Linux operating system running on CPU0 is the master of the system and is responsible for:

- System initialization
- Controlling CPU1's startup
- Communicating with CPU1
- Running the USRP applications
- Interacting with the user

The bare-metal application running on CPU1 is responsible for:

- Controlling the Transmitter and Receiver Systems in the programmable logic (PL)
- Managing a "heart beat" that can be monitored by Linux on CPU0
- Communicating with Linux on CPU0

In running the designs in AMP configuration, care has been taken to prevent both CPUs from contending for shared resources. The bare-metal application running on CPU1 has been modified to limit access to the shared resources. Actions taken by this design to prevent problems with the shared resources include:

1. DDR memory: Linux has only been made aware of memory at 0x00000000 to 0x2FFFFFFF. CPU1 uses memory from 0x30000000 to 0x3FFFFFFF for its bare-metal application.
2. L2 Cache: CPU1 application was modified so that it does not use L2 cache.
3. Global Timer: CPU1 was modified so that it does not use the global timer and instead uses the private timer for the heartbeat.

6.3.1. Hardware

The PL was not modified by any blocks. The block diagrams of the Transmitter and Receiver systems are the same as described in chapter 2.

6.3.2. Software

The software is also broken down into three sections:

- First stage boot loader (FSBL)
- Linux operating system and applications for CPU0
- Bare-metal operating system and application for CPU1

6.3.3. FSBL

Within this AMP design project files, the FSBL has been also modified by Xilinx to continue searching for files and loading them into memory until it detects a file that has a load address of 0xFFFFFFF0.

6.3.4. Linux

The Linux in this AMP configuration is configured as symmetric multiprocessing (SMP) but restrict the number of available CPUs to 1. The Linux kernel, U-Boot, device tree, and the root file system ramdisk, was created by the same instructions in chapter 4. All generated files are available as part of the project files.

The bootargs in the device tree is modified to add maxcpus=1. The device tree is also modified to reduce the amount of memory available to Linux to provide untouched memory space for CPU1's application.

6.3.5. Linux Applications

The two Linux applications that run on CPU0 in the reference design [49] are also used to interact with CPU1 that is running the bare-metal application. The first application, rwmem, which provides a simple memory read and write access from Linux to OCM. This rwmem application is used to peek (read) and poke (write) addresses in OCM. The second application, softUart, which provides a UART-style communication between Linux running on CPU0 and bare-metal running on CPU1 through predefined memory locations in OCM. The two USRP applications were also added to run on CPU0. These two applications use the UHD and GNURadio installed on the file system to transmit and receive the Wi-Fi time domain using the USRP devices connected to the ZYNQ FPGA through USB. The USRP transmitter application reads the output file of the Wi-Fi transmitter application of CPU1 and then sends it through the RF platform. The USRP receiver application receives the Wi-Fi time domain and write it in a file. This file is read by the Wi-Fi receiver application of CPU1 and the receiver produces the output Wi-Fi data bits.

CPU0 (running Linux) starts CPU1 (running bare-metal) by also writing the value of 0x30000000 to address 0xFFFFFFF0 using the included rwmem application. When CPU1 wakes up, it reads the value 0x30000000 from address 0xFFFFFFF0 (written using the rwmem command) and then jumps to address 0x30000000. Note that the FSBL placed CPU1's ELF at 0x30000000.

6.3.6. Bare-Metal Application Code

Linux, running on CPU0, is responsible for initializing shared resources and starting up CPU1. The bare-metal board support package (BSP) named standalone_v5_1 that is part of the SDK 15.2 install includes support for the preprocessor define constant USE_AMP. The BSP was also modified as the version included in the reference design to remove the re-mapping when USE_AMP is set.

Within those AMP designs, no Zynq UARTs are used by the bare-metal application. Instead, the application running on CPU1 contains its own outbyte() function that is used to communicate via OCM to a software UART running in a Linux application on CPU0.

Linux disables cache access to the OCM. However, the default standalone BSP would attempt to enable cache for OCM and therefore conflict with Linux. In this AMP configuration, the function XIL_SetTlbAttributes() is used in the CPU1's main() application function to disable cache on OCM. The XIL_SetTlbAttributes() function has been modified in the included source code such that it only flushes L1 cache and leaves L2 cache untouched to prevent access to the SCU where L2 cache is controlled.

6.3.7. CPU1 Applications

CPU1's application is located in memory starting at address 0x30000000. The linker script is used to set the starting address.

CPU1's transmitter application does the following:

1. Configures the MMU to disable cache for OCM accesses in the address range of 0xFFFF0000 to 0xFFFFFFFF. The address mapping of the OCM is untouched so OCM exists at addresses 0x00000000–0x0002FFFF and addresses 0xFFFF0000–0xFFFFFFFF. Only the high 64 KB of OCM is used by the design so cache is disabled on addresses 0xFFFF0000–0xFFFFFFFF.
2. Controls the Transmitter System in the programmable logic (PL).
3. Reads the input data to the Wi-Fi transmitter from a file and deliver it to the DMA block through the AXI bus.
4. Extracts the output data from the FIFO block through the AXI bus and write it in a file.
5. Increments an OCM location (COMM_VAL). This OCM location is referred to as the Heartbeat.
6. Sleeps for one second.
7. CPU1's main() function repeats step 5 and step 6 continuously.
8. The output from the print statement is redirected to use the OCM COMM_TX_FLAG_OFFSET and COMM_TX_DATA_OFFSET locations. In turn, Linux consumes the OCM data and prints the string to the Linux console.

CPU1's receiver application does the following:

1. Configures the MMU to disable cache for OCM accesses in the address range of 0xFFFF0000 to 0xFFFFFFFF. The address mapping of the OCM is untouched so OCM exists at addresses 0x00000000–0x0002FFFF and addresses 0xFFFF0000–0xFFFFFFFF. Only the high 64 KB of OCM is used by the design so cache is disabled on addresses 0xFFFF0000–0xFFFFFFFF.
2. Controls the Receiver System in the programmable logic (PL)
3. Reads the input data to the Wi-Fi receiver from a file and deliver it to the DMA block through the AXI bus.
4. Extracts the output data from the FIFO block through the AXI bus and write it in a file.
5. Increments an OCM location (COMM_VAL). This OCM location is referred to as the Heartbeat.
6. Sleeps for one second.

7. CPU1's main() function repeats step 5 and step 6 continuously.
8. The output from the print statement is redirected to use the OCM COMM_TX_FLAG_OFFSET and COMM_TX_DATA_OFFSET locations. In turn, Linux consumes the OCM data and prints the string to the Linux console.

6.3.8. Design Files

Each of the Wi-Fi Transmitter and Receiver Systems design contains these files:

- Vivado project
- SDK source files for Linux and CPU1 applications
- Generated files including:
  - Bit file
  - All files for the SD card
  - Application ELF files for Linux and CPU1
  - BOOT.BIN build scripts
  - Modified bare-metal BSP
  - Modified Xilinx FSBL
  - Modified devicetree.dts and devicetree.dtb

6.3.9. Generating Applications

6.3.9.1.Configuring SDK

The standalone BSP files (used by the bare-metal application) and modified FSBL files are the same files that have been included in the reference design files.

6.3.9.2.Creating Custom FSBL Application

The same instructions in 1.2.11.2 are used in generating the fsbl.elf.
A pre-compiled version is also available at:
Work_Tx_wifi\project_1_wifi_edited\project_1.sdk\bootgen\zynq_fsbl.elf.

6.3.9.3.Creating Bare-Metal Application for CPU1

The same instructions in 6.2.11.3 are used in generating tx_wifi.elf and rx_wifi.elf.
The applications have already been compiled and are available at
Work_Tx_wifi\project_1_wifi_edited\project_1.sdk\ tx_wifi\Debug\ tx_wifi.elf.
Work_Rx_wifi\Project_1_wifi\project_1.sdk\rx_wifi\Debug\rx_wifi.elf.

6.3.9.4.Creating Linux Application RWMEM

The same instructions in 6.2.11.4 are used in generating rwmem.elf.

6.3.9.5.Creating Linux Application Soft UART

The same instructions in 1.2.11.5 are used in generating softUart.elf.

6.3.10. Creating Linux Kernel

Refer to chapter 5 for instructions on how to compile the kernel. A copy of a pre-compiled kernel is included at AMP_wifiRx_GNUradio_image/uImage.

6.3.11. Creating Linux Device Tree

The same instructions in 1.2.12.1 are used in generating devicetree.dtb. A copy of a pre-compiled device tree is included at AMP_wifiRx_GNUradio_image/devicetree.dtb.

6.3.11.1. Creating U-Boot

Refer to chapter 5 for instructions on how to compile U-Boot. A copy of U-Boot has already been compiled and is included at
Work_Tx_wifi\project_1_wifi_edited\project_1.sdk\bootgen\u-boot.elf.

6.3.11.2. Acquiring Root File System

Refer to chapter 5 for instructions on how to make the root file system. A copy of the ramdisk is included at final_rootfs\uramdisk.image.gz.

6.3.11.3. Generating Boot File

The boot file BOOT.BIN contains the FSBL, FPGA bit file, and the ELF for the application that runs on CPU0 (U-Boot). The design files contain a batch file, a BootGen configuration file.

The configuration file contains the names of the files that are copied to DDR memory. The order of these files is important. For this design, the order is:

1. FSBL ELF.
2. CPU0 application.
3. CPU1 application.
4. Dummy cpu1_bootvec.bin file.

A precompiled version of BOOT.BIN is available at
AMP_wifiTx_GNUradio_image\BOOT.BIN for the transmitter and
AMP_wifiRx_GNUradio_image\BOOT.BIN for the receiver.

The same instructions in 1.2.11.9 are used in generating BOOT.BIN file.

6.3.12. Copying Files to SD Card

The Zynq AP SoC requires these files to be present on the SD card in order to boot Linux:

1. BOOT.BIN (contains the FSBL, BIT file, U-Boot, and application for CPU1)
2. uramdisk.image.gz (ramdisk file system extracted to memory by U-Boot)
3. devicetree.dtb (used by U-Boot and Linux for device information)
4. uImage (Linux kernel loaded and executed by U-Boot)
5. u-boot.img (bootloader)

The Wi-Fi Transmitter bare metal code require the input file “wifidata.txt” to be present on the SD card in order to run, and the USRP applications should also be present on the SD card.

6.3.13. Running the Design

Setup the hardware by following the board setup instructions in the “TRD Demonstration Procedure” section of Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit. The hardware setup configures the ZC702 demonstration board to boot from the SD card. The terminal program should be configured to listen to the correct COM port with a baud rate of 115200. CPU0 then starts running U-Boot and boots Linux.

The soft UART application is started in linux with the command softUart.elf &.This command runs softUart in the background. SoftUart continues to monitor shared OCM memory at locations 0xFFFF9000 (COMM_TX_FLAG_OFFSET) and 0xFFFF9004 (COMM_TX_DATA_OFFSET).

The second CPU running the Wi-Fi Transmitter bare metal code is started with the command rwmem.elf 0xfffffff0 0x30000000. Location 0xFFFF8000 should start incrementing every second and can be viewed by using the command rwmem.elf 0xffff8000. Also, the output file of the transmitter “OUTPUT.txt” will be written to the SD card.

In order to run the USRP transmitter application, the output file of the transmitter and the USRP transmitter application need to be copied from the SD card to the Linux file system. The following commands are used for mounting the SD card on Linux and copying these files:

- mount /dev/mmcblk0p1 /mnt/
- cp /mnt/OUTPUT.TXT ~
- cp /mnt/top_txblock.py ~

After copying these files, the USRP transmitter application can be executed using the command “python top_txblock.py".

For the USRP receiver application to run, the application need to be copied first from the SD card to the Linux file system then executed using the commands:

- mount /dev/mmcblk0p1 /mnt/
- cp /mnt/top_rxblock.py ~
- python top_rxblock.py

In order to run the Wi-Fi Receiver bare metal application, the output file of the USRP receiver application needs to be copied to the SD card so that the Wi-Fi Receiver bare metal code can read it. The file can be copied using the command cp ~ /mnt/OUTPUT.TXT

The second CPU running the Wi-Fi Receiver bare metal code is started with the command rwmem.elf 0xfffffff0 0x30000000. Location 0xFFFF8000 should start incrementing every second and can be viewed by using the command rwmem.elf 0xffff8000. Also the output file of the receiver "OUTRX.txt" will be written to the SD card.

Chapter 7: Conclusion & Future work

7.1. Results Summary

Our project is an experience of both hardware and software working environment. A summary of this year's work, building it on all the previous years' hard work, are as follows:

- Separation of 2G Transmitter and receiver HDL codes.
- Separation of WIFI Transmitter and receiver HDL codes.
- Modifying block design and adding new blocks.
- Controlling two USRPs using GNU radio's GUI.
- Interfacing between USRP and FPGA using Asymmetric multi-processors.
- Making a complete semi-automatic system to send and receive with WIFI standard by using two ZYNQ boards and two USRPs acting as transmitter and receiver. See Figure 7-1.

Figure 7-1 Transmitter and Receiver hardware full chain

7.2. Real-time testing results

As mentioned in chapter 3, the distance between the transmitter and receiver antennas affect greatly the correctness of the reception, especially if there is no channel estimation -which is our case-, where the location of interference and amount of gain needed for correct transmission and reception, can't be estimated. So, it's going to be a trial and error process to achieve the best results possible with the available tools.

Upon operating and testing the full chain using the WIFI standard, the BER has been calculated and the following figures will illustrate those bit error rates Vs. the distance between the antennas for different gain values.

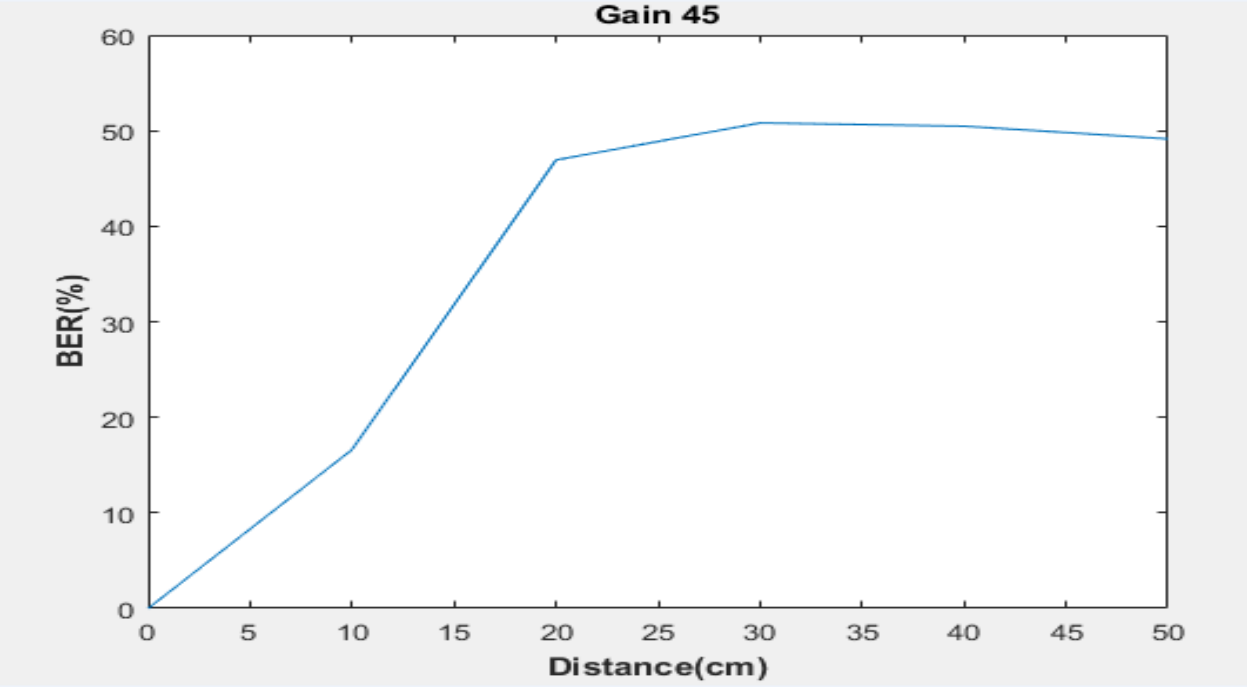

Figure 7-2 BER ratio for 45 gain and 128K sampling rate

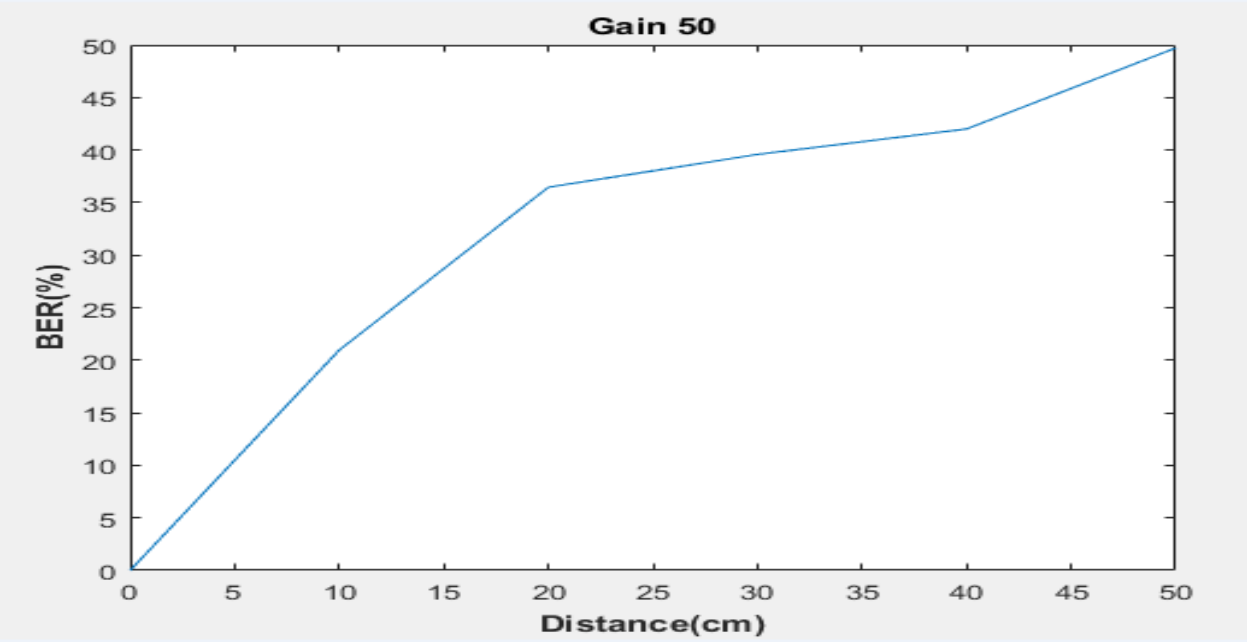

Figure 7-3 BER ratio for 50 gain and 128K sampling rate

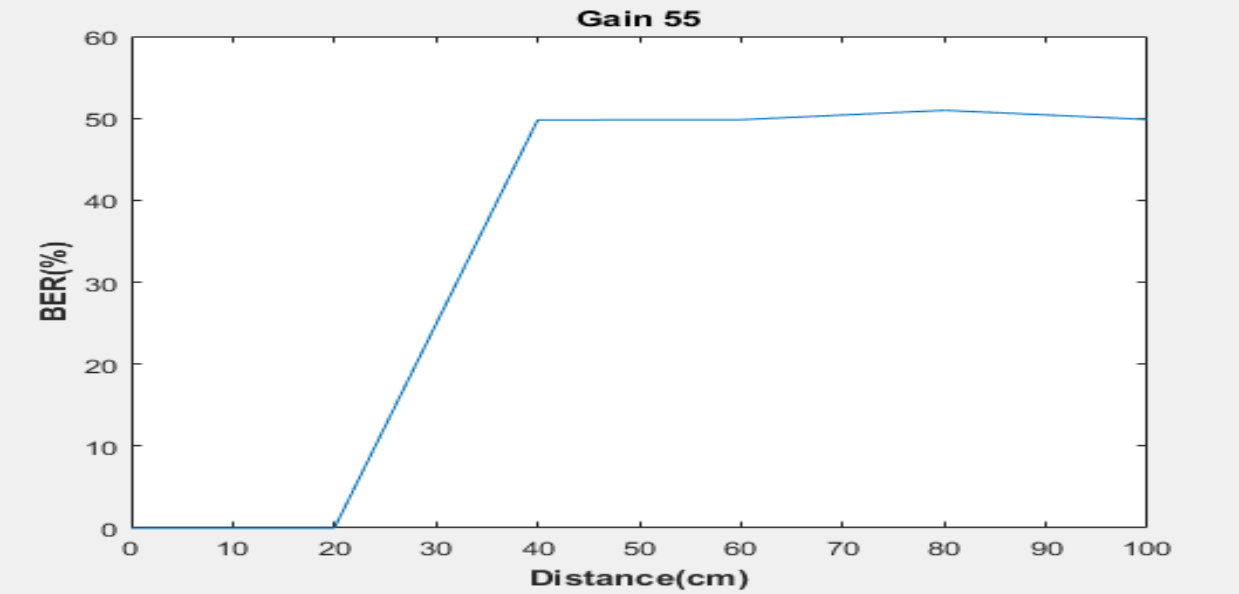

Figure 7-4 BER ratio for 55 gain and 128K sampling rate

As shown in the previous figures, as the gain increase the packets arrive at the transmitter with zero BER for small distances. By increasing the distance, the BER increases almost proportionally until it nearly reaches 50% at 50 cm which is expected to decrease after adding the channel estimation RTL codes. These results can be improved by using channel estimation and a suitable error correction scheme can correct the data frames received.

Please note that the sudden 50% BER is due to ONE missing byte of the whole packet, as shown in Figure 7-5. This missing byte disturbs the sequence and causes the whole packet to be marked as corrupt, as the system identifies this packet to be missing half of its information.

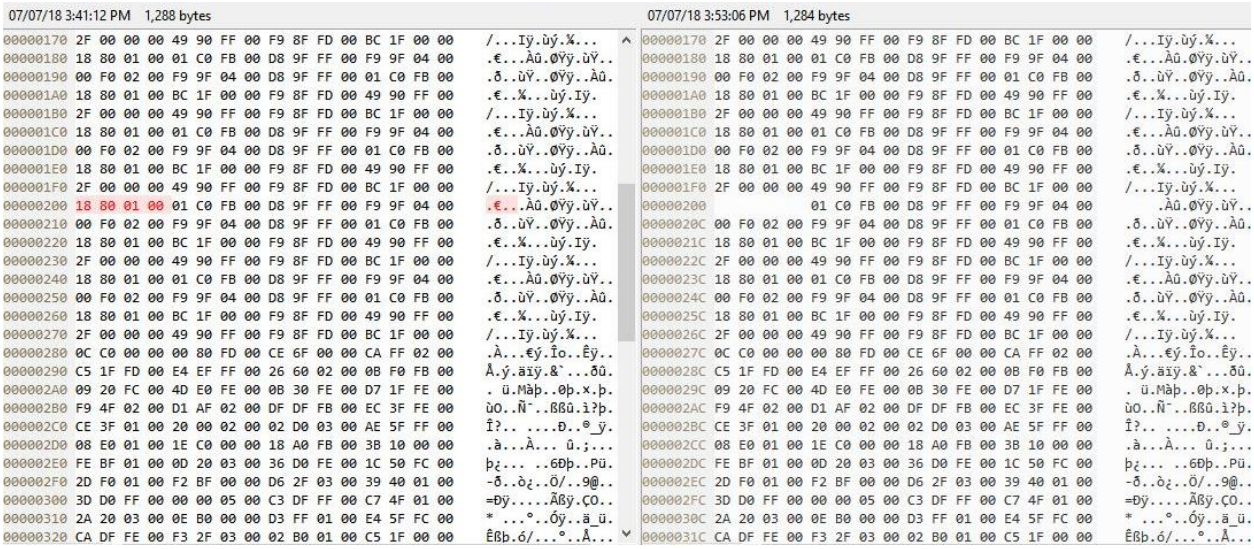

Figure 7-5 The missing byte

In figure 7-5, the missing byte is highlighted in the left-hand side and has an empty space in its place in the right-hand side of the image.

7.3. Conclusion

Our project was an integration to the previous years' work. As mentioned before, our mission was to separate the transmitter & the receiver, in order to transfer a data file through two separate ZYNQ board + USRP kits. The ZYNQ board's FPGA, which has the transmitter codes, modulates the data and then passes it on to the USRP, which is the transmitter's antenna, through air to be received by the other USRP. The other USRP acts as the receiving antenna which finally forwards the data file received on to the receiver blocks configured on the other ZYNQ board's FPGA to be demodulated. Controlling the transmitter and receiver FPGAs and USRP with Asymmetric multi-processors by running transmitter/receiver bare-metal codes on CPU1 (running codes on FPGA) and running the LINUX image which has the UHD and GNU Radio on CPU0 (control USRP). The final hardware prototype is shown below in Figure 7-6. Unfortunately, a bug in Xilinx Linux image causes the bitstream to be corrupted on running bare

metal codes on a processor and a Linux image with USB driver on another, which is necessary to connect and control the USRP. Hence, oursystem is semi-automatic not completely standalone.

Figure 7-6 Project's prototype

7.4. Future work

7.4.1. Interfacing Using Ethernet

Due to the existence of the bug in the USB driver in the Xilinx linux (2015.2) ,our version, a proposed solution is to use another series of USRPs (N-series), which has an ethernet connection. The ethernet connection would avoid this bug and increase the data rate.

7.4.2. Communication

7.4.2.1. Channel Estimation

In the previous years' work, the channel was modeled only using Matlab so the channel estimation techniques were done using Matlab codes only. Now, due to the real channel, it is needed to implement the channel estimation with RTL codes in order to decrease the BER which exceeds 50% in the WIFI standard.

7.4.2.2. Separating new systems

As the main target is to build Multi-standards communication system, the separation for transmitter and receiver codes and block design of the different communication standards (as 3G, LTE, Bluetooth) will be needed.

7.4.3. Electronics

7.4.3.1. DPR between communication standards

Applying DPR concept to the afore mentioned communication standards, so that we have an SDR system using DPR concept which will realize all DPR advantages discussed before.

7.4.3.2. Synchronization between transmitter and receiver

Synchronize between the separated transmitter and receiver systems and enhancing switching rates.

7.4.4. RF

Check that the whole system is working correctly with DPR technique in the RF band.

References

[1] I. F. Akyildiz, W.-Y. Lee, M. C. Vuran, and S. Mohanty, “NeXt generation/ dynamic spectrum access /cognitive radio wireless networks: a survey,” Computer Networks, vol. 50, pp. 2127–2159, 2006

[2] T. Yücek and H. Arslan, “A survey of spectrum sensing algorithms for cognitive radio applications,” Communications Surveys & Tutorials, IEEE, vol. 11,pp. 116–130, 2009.

[3] A. A. Bletsas, Intelligent antenna sharing in cooperative diversity wireless networks. PhD thesis, Citeseer, 2005

[4] J. Mitola III and G. Q. Maguire Jr, “Cognitive radio: making software radios more personal,” Personal Communications, IEEE, vol. 6, pp. 13–18, 1999.

[5] J. Mitola, “Cognitive Radio—An Integrated Agent Architecture for Software Defined Radio,” 2000

[6] Xilinx Inc. “AXI DMA PG021”, March 7, 2011.

[7] 3GPP, Multiplexing and channel coding (FDD) (Release 12), 2014.

[8] Xilinx Inc. “Zynq 7000 TRM UG585”, September 2015.

[9] A Testing Environment for Multi-Clock Systems on Xilinx ZynQ SoC Dummy FIFO

[10] Asmaa Rayan, Alaa Othman, Maha Abd El-Maqsoud, OFDM over optical fiber channel, Cairo: Cairo University, Faculty of Engineering, 2015.

[11] Xilinx Inc. “AXI Reference Guide UG761”, March 7, 2011.

[12] Xilinx Inc. “AXI interconnect DS768”, December 18, 2012.

[13] https://ieeexplore.ieee.org/document/6288496/

[14] http://www.ni.com/en-lb/shop/select/usrp-software-defined-radio-device

[15] https://www.upc.edu/sct/en/documents_equipament/d_174_id-459.pdf

[16] https://www.faculty.ece.vt.edu/swe/chamrad/crdocs/CRTM09_060727_USRP.pdf

[17] https://en.wikipedia.org/wiki/Universal_Software_Radio_Peripheral

[18] https://kb.ettus.com/UHD

[19] https://wiki.gnuradio.org/index.php/Main_Page

[20] https://wiki.gnuradio.org/index.php/InstallingGR

[21] Sami H.O.Salih, Mamoun M.A.Suliman, “Implementation of BPSK Modulation using SDR”, International Journal Of Scientific And Engineering Research, vol.2, Issue 5, May-2011. Pp.1-4.

[22] J.Markstember, “BPSK Modulation/Demodulation Techniques” provides lowest probability of error, microwave systems, News, pp.150-176, June 1984.

[23] http://forums.xilinx.com

[24] http://www.wiki.xilinx.com/Zynq%20Releases

[25] http://www.wiki.xilinx.com/Zynq%202015.2%20Release

[26] https://a4z.bitbucket.io/docs/BitBake/guide.html

[27] http://git.yoctoproject.org/

[28] http://layers.openembedded.org.

[29] https://layers.openembedded.org/layerindex/branch/master/layer/meta-sdr/

[30] http://www.wiki.xilinx.com/Build+Device+Tree+Blob

[31] https://github.com/kratsg/meta-l1calo/wiki/Building-and-Deploying-an-OS

[32] http://www.wiki.xilinx.com/Build+U-Boot

[33] https://github.com/jpendlum/meta-xilinx

[34] http://www.wiki.xilinx.com/Zynq%20Linux

[35] https://github.com/kratsg/meta-l1calo/wiki/Zynq-7:-Prepare-and-Boot-Hardware#fsbl-method

[36] https://www.ibm.com/developerworks/library/l-yocto-linux/index.html

[37] https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html#yocto-project-components

[38] https://www.yoctoproject.org/docs/2.5/brief-yoctoprojectqs/brief-yoctoprojectqs.html

[39] https://www.safaribooksonline.com/library/view/embedded-linux-development/9781783282333/ch10s03.html

[40] https://www.yoctoproject.org/docs/1.8/dev-manual/dev-manual.html#understanding-and-creating-layers

[41] https://www.yoctoproject.org/docs/2.0/yocto-project-qs/yocto-project-qs.html

[42] https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html#yocto-project-components

[43] https://wiki.gnuradio.org/index.php/OpenEmbedded

[44] http://www.wiki.xilinx.com/Using%20meta-xilinx-tools%20layer

[45] https://github.com/Xilinx/meta-xilinx-tools

[46] http://www.wiki.xilinx.com/Yocto

[47] https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html#yocto-project-components

[48] McDougall J. Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors. 2013.

[49] Xilinix inc. “Vivado Design Suite User Guide Partial Reconfiguration”, UG909 (v2015.2) June 24, 2015

[50] Xilinix inc. “Spartan-6 FPGA Configuration User Guide”, UG380 (v2.10) March 31, 2017

[51] Xilinix inc. “Spartan-6 Family Overview”, DS160 (v2.0) October 25, 2011

Appendix A: Using the FPGA inside the USRP

The Vivado Design Suite Release used is 2015.2. This release supports the following products:

- 7 Series devices: This release supports Partial Reconfiguration for all Virtex®-7,
- Kintex®-7, Artix®-7, and Zynq®-7000 All Programmable SoC devices.
- UltraScale™ devices: This release includes UltraScale device support for the following: KU035, KU040, KU060, KU115, VU095, VU125, VU160, and VU190 [50].

Inside the USRP B200 there is Spartan 6 XC6SLX75 FPGA but it cannot be used in our project as Spartan 6 **does not support DPR** in all Vivado releases [51],[52].

Appendix B: Alternative USRP series

a. No standalone USRP

As discussed in chapter 4, UHD is needed in order to interface between USRP and ZYNQ board, but UHD needs OS to be installed on such as Linux. So, in all USRP series there is always a need for OS which lead to the problem and its solution discussed in chapter 5.

b. The optimum USRP version for this application

As mentioned in our conclusion, that due to a Xilinx image bug we can't use USB driver with our bare-metal codes. so, the suggested solution is to use the Ethernet cable instead of the USB cable to interface between the USRP and FPGA. The Ethernet connection is available in N-series USRPs.

Appendix C: Partitioning of SD card

Partitioning the SD card was a suggested solution to be able to run LINUX OS on one part of the drive and running the bare metal codes from the other part. The LINUX OS is needed to drive the USRP and access the data files. Unfortunately, this solution doesn't work as the LINUX image part can't access the other part of SD card, so the other solution, which is discussed in asymmetric multi-processing chapter 6, was executed.