Hardware implementation of 5G Transmitter

A Project Report

submitted by

MAGANURU SIVAPRASAD

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2018

THESIS CERTIFICATE

This is to certify that the thesis titled **Hardware implementation of 5G Transmitter**, submitted by **M.Siva Prasad**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Radha Krishna Ganti
Project Guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Dr. Nitin Chandrachoodan
Project Guide
Associate Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 8th May 2018

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude towards several people who enabled me to reach this far with their timely guidance, support and motivation.
First and foremost, I offer my earnest gratitude to my guide, Dr.Radha Krishna Ganti whose knowledge and dedication has inspired me to work efficiently on the project and we thank him for motivating me, allowing me freedom and flexibility while working on the project.
I would like to thank my co-guide Dr.Nitin Chandrachoodan who have guided me throughout the project.

ABSTRACT

KEYWORDS: Vivado HLS, Vivado IP Integrator, Xilinx SDK.

The current trend in digital design is to accelerate the design and developmental cycles without compromising on verification. One of the key factor underlying the concept is to raise the abstraction layer from the traditional RTL level which are time consuming,error prone and difficult to debug. HLS tools are an attractive proposition for the rapid prototyping of the systems and bridges the gap between development times and time to market. HLS tools automatically transforms the algorithms written in C, C++ to RTL implementations.

In this thesis we are presenting a case study of using Vivado HLS tool and Xilinx Design Suites including Vivado IP Integrator and Xilinx SDK for the design and implementation of a 5G transmitter. In order to improve the performance of the hardware various optimizations like data flow pipelining, loop pipelining, array partitioning are used to convert the C code to RTL implementations. The challenge was to achieve the required performance through minimized iteration interval with less additional hardware overhead. The hardware generated is successfully tested in Zynq Ultrascale evaluation board

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

RTL	Register Transfer Level
HLS	High Level Synthesis
IP	Intellectual Property
FPGA	Field Programmable Gate Array
SDK	Software Development Kit
HDL	Hardware description language
EDA	Electronic Design Automation
VHDL	VHSIC Hardware Description Language
IDE	Integrated Development Environment
GNU	GNUâĂŹs Not Unix
GCC	GNU Compiler Collection
JTAG	Joint Test Action Group
BSP	Board Support Package
GNU	GNUâĂŹs Not Unix

CHAPTER 1

Introduction

1.1 Introduction

The move to take new products, which are characterized by highly complex designs,to market as fast as possible made the EDA community to device strategies that reduces the design and development cycles without compromising on verification. The strategies adopted for design acceleration are (i) design reuse and (ii) making higher abstraction level of the design.The strategies to reduce design and development cycles in turn implies reduction in developmental costs.

Xilinx Vivado HLS is a tool for synthesizing digital hardware directly from a description of the system either in C/C++ or SystemC.This eliminates the need of designing the hardware in HDL languages like VHDL/Verilog. This high level design doesnâĂŹt fix the hardware architecture as is done by HDL languages. The most important feature of the HLS tools are the designed functionality and its hardware implementations are kept separate and this provides great flexibility to the design community as various architectures can be explored and arrived at an optimum design.

Xilinx SDK is a part of Vivado IDE which provides a platform for creating fully functional software applications. The SDK includes GNU based compiler tool chain (GCC compiler, GDB debugger), JTAG debugger, flash programmer, drivers for Xilinx IPs and standalone BSPs and libraries for application specific functions. All of these features are accessible from within the Eclipse based IDE.

1.2 Motivation

Current research in the field of wireless communication is tending towards increasing the data rates with increased spectral usage efficiency. This is often accompanied by trying out various algorithms and optimization of these algorithms which the situation demands. Sometimes the demand is to build customized communication systems which put forth specific requirements of the usage of specific algorithms.

1.2.1 Pros and Cons of HLS

There are a wide range of HLS tools available which are differ by their ease of use and quality of the hardware derived.Some of the challenges that HLS tools must overcome(Donald.G.Bailey (2015)) are listed below.

- Software algorithms are sequential in nature whereas hardware operates concurrently.HLS must map the sequential algorithm onto concurrent hardware
- Due to the sequential nature of software execution timing is implicit whereas hardware deals with timing constraints by controlling and synchronizing operations at the clock cycle level.
- Software supports fixed word lengths say 8,16,32 or 64 whereas hardware supports arbitrary precision data types like 5,11,15 etc depends on the computation being performed.
- Software supports dynamic memory allocation whereas hardware doesnâĂŹt support, as local variables are stored in registers with distributed address spaces.
- Data transfer between various modules is through shared memory in case of softwareâĂŹs whereas hardware depends upon the use of FIFOâĂŹs , stream interfaces and associated handshaking signals for flow control.

Synthesis of an algorithm using HLS tools generally perform the following steps data-flow analysis is being carried out as part of its algorithm synthesis to determine the type of operations that need to be performed; then resource allocation is done to determine how the resources on the hardware can be made use of in building the hardware followed by resource binding to determine the type of operations to be done by which hardware resource and finally scheduling which determines at what instant of time each operations will be performed to obtain the desired functionality.

Some of the key benefits in using HLS are listed below:

- Portability of the source code to any hardware which rarely involves restructuring of the code.
- HLS tools during its algorithm synthesis will analyze the structure of the code (loops, branches etc) to automatically extract and build the control path (FSM) whereas traditional RTL design requires explicit coding of the control path, which for complex designs it will be a herculean task.

- Data dependencies and the sequence of operation are analyzed by HLS tools to exploit parallelism which can be pipelined further to achieve the desired timing constraints.
- A pipelined architecture can be inferred from loops which involves less data dependencies between successive iterations.
- Iterations in a loop can be made parallel by loop unrolling which divides the loops to multiple hardware.

Design space exploration is accomplished through a combination of source code optimization and synthesis directives. Finding out the optimum design is generally a trade off between the speed and resources. Software profiling tools in HLS helps to identify processing bottlenecks, which enables to put more efforts on areas where it can potentially achieve the greatest gains. Design explorations at early stage can be achieved through HLS tools as it provides reasonably accurate estimate of resources without going for synthesizing the resulting RTL. In contrast,design exploration at RTL coding level will generally require considerable amount of time as it involves re-coding to change both the data and control paths and its error prone too. In HLS as the verification takes place at a higher level, simulations or verificationâĂŹs required for the generated designs are faster. But there is a necessity to validate the final design at the RTL level to ensure that the algorithm transformations are correct and HLS automatically generates RTL test benches from high level verification test benches.

Some of the key limitations in using HLS are listed below:

- Code restructuring is a must in realizing the hardware in order to improve performance.
- Treating HDL like a software leads to inefficient use of resources.
- Algorithms are based on pointers, whereas hardware implementations rely on arrays and array references which makes HLS finding it difficult to map to hardware and hence at many times a restructuring of the code is required.
- Recursive algorithms are very difficult to translate to hardware using HLS.
- Verification failure at the RTL level will be very difficult to analyze as the RTL code generated through HLS lacks human readability.
- Best algorithms for software realizations are not preferably the best suited for hardware implementations.

1.2.2 Design Flow Comparison between RTL and HLS

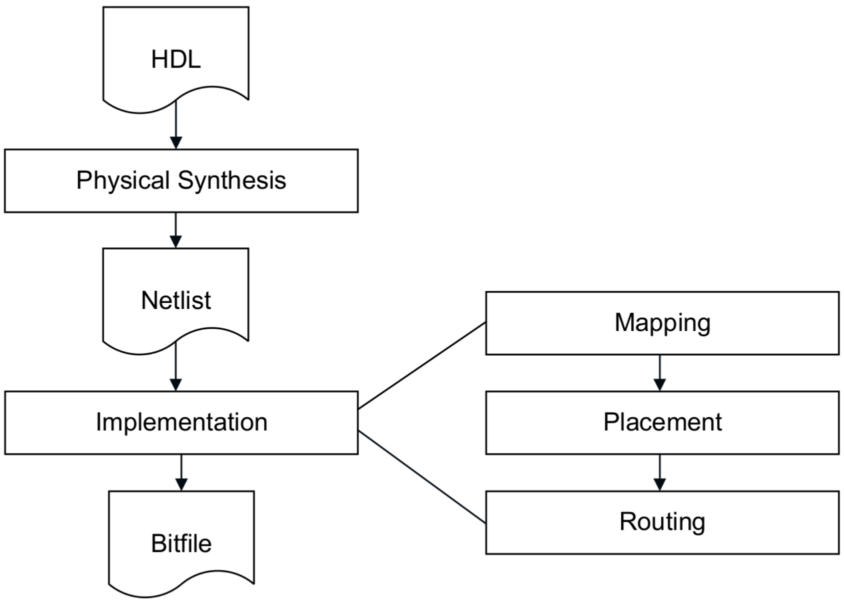


Figure 1.1: RTL Design Flow

The traditional RTL design flow is shown in Figure 1.1. The design entry comprises of files written in the HDL such as Verilog or VHDL. The design is described by RTL because any synchronous digital hardware circuit can be represented using Huffman model representation of an FSM wherein the data flows between hardware registers and the combinational circuits does the operation on these data.The functionality written in HDLâĂŹs are verified by using test benches which are also written in HDL. The test bench provides inputs to the design and also receives outputs from the design. This process is called Behavioural Simulation where the functionality of the design is verified. The physical synthesis converts this description in hardware language into netlist which has gates, registers and wires. The physical synthesis which consist of technology mapping, placement and routing adds informationâĂŹs such as gate delays, wire length, location of gates to produce the final bit file. The bit file can be used to program the FPGA to perform required digital functionality.

HDL level description of a hardware consumes a major chunk of the time as it is written at a lower level of abstraction than algorithmic level. RTLâĂŹs are written at a level of multiplexers,flip-flops etc.. In addition HDL is a concurrent programming language which makes them difficult to understand and code. HLS takes advantage of the situation through ease of programming thereby increasing productivity and ease of understanding.

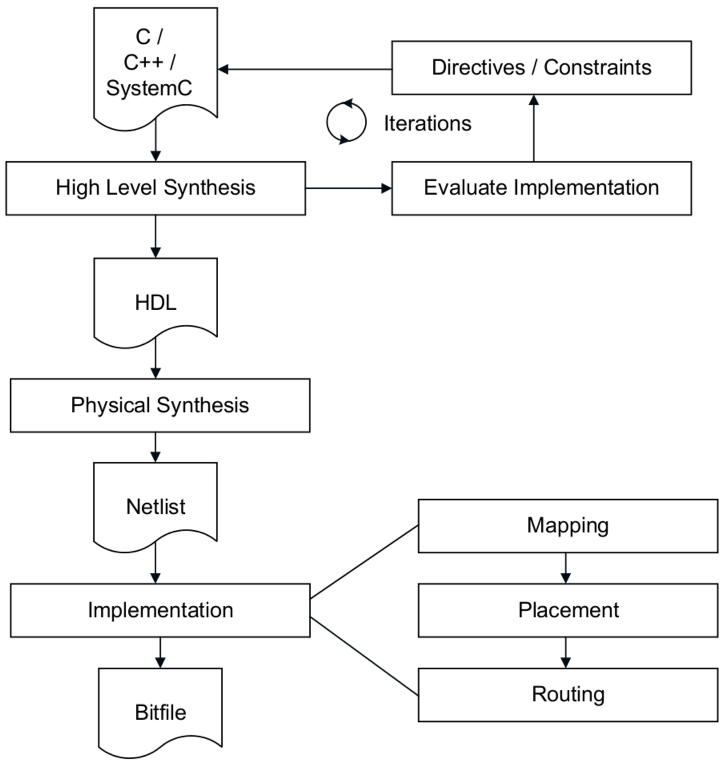


Figure 1.2: HLS Design Flow

HLS based design flow is shown in Figure 1.2. The design entry is in C/C++ or SystemC which allows the designer to describe an algorithm in a sequential manner.The algorithm written in high level languages are verified through a test bench which is also written in the same language and the process of functional verification is called C Simulation. After C Simulation produced the required result, the HLS synthesis tool analyzes and process the C based code with user specified constraints and directives and converts the C/C++ description into RTL which is called the C Synthesis or High Level Synthesis âĂŞ the rest of the flow is similar to the traditional RTL flow.The design space exploration happens through the incremental changes provided by the user in the form of directives and hence different architectures are evaluated in due course.

Once the equivalent RTL model is produced, it can be further verified against the original C/C++/SystemC code via the process of C/RTL Co-Simulation.This process re-uses the original, C-based testbench to supply inputs to the RTL version generated by HLS, and check the outputs it produces against expected values.Importantly, this saves the effort of generating a new RTL test bench. The advantages of this flow are higher productivity, flexibility of design and architectural exploration which also helps to identify performance bottlenecks and area requirements at an early stage of design.The simulation at HLS based flow is much faster compared to the RTL level as the former allows to do so in C/C++ level. Error correction and design re-use at HLS based design is more easier and hence cost effective than RTL based design.

Once the design has been validated, and the implementation iterated to the point of achieving the intended design goals, it will be intended for integration into a larger system.This can be achieved directly through the use of packaging the outputs produced by Vivado HLS by means of Export RTL. Packaging to an IP helps to introduce HLS designs easily into other Xilinx tools, namely IP Integrator within Vivado IDE, XPS(for the ISE design flow), and System Generator. Once the design has been integrated and implemented with HLS generated IPs and specific IPs provided by Xilinx like softcore processors and other peripherals, the whole hardware system can be controlled through application specific softwareâĂŹs written on top of it through Xilinx SDK. The application in the form of elf (executable and linkable format) can be made to run on softcore processors and the system can be made to interface with external world.

Keeping this background in mind, the motivation for the present work is to use HLS,Vivado IP integrator and Xilinx SDK for the implementation of a specific wireless system.

CHAPTER 2

Background

The details present in this chapter contains information that are part of ug902,ug904.But are reproduced here for easy reference.

2.1 Introduction to Vivado HLS

This section will cover topics including the specification of data types and its implications for synthesis, the creation of port and block-level interfaces, aspects of algorithm synthesis and the use of directives and constraints to influence the solutions produced by HLS.

2.1.1 Data Types

The specification of data type affects the resource utilization, timing performance, and power consumption. Under-specifying the word length compromises accuracy, while over-specifying leads to increased resources, inflated power consumption, and a sub optimal maximum clock frequency. Vivado HLS supports both the native C/C++ data types plus the arbitrary precision data types for integer (for C/C++) and fixed point (for C++ only). It also supports the complex data type by including the header file "complex.h".

For the arbitrary precision fixed point data type W denotes the width of the word length and I denotes the bits specified for integer portion. So W âĹŠ I decides the bits for fractional representation. Q is a string which specifies the quantization mode and O gives the overflow mode. N specifies the number of bits in overflow wrap mode. The strings Q, O and N are optional. Vivado HLS also supports floating point data types and operations, as long as these maps to available Xilinx technology core libraries.

Language	Data Type	Description	Header
C	intN	N bit precision signed integer	#include "ap_cint.h"
	uintN	N bit precision unsigned integer	
C++	ap_int<N>	N bit precision signed integer	#include "ap_int.h"
	ap_uint<N>	N bit precision un signed integer	

Table 2.1: Arbitrary precision data types for C/C++

Language	Data Type	Description	Header
C++	ap_fixed<W,I,Q,O,N>	signed fixed point number of I integer bits and W-I fractional bits	#include "ap_fixed.h"
	ap_ufixed<W,I,Q,O,N>	unsigned fixed point number of I integer bits and W-I fractional bits	

Table 2.2: Arbitrary precision fixed point data types for C++

For the arbitrary precision fixed point data type W denotes the width of the word length and I denotes the bits specified for integer portion. So $W - I$ decides the bits for fractional representation. Q is a string which specifies the quantization mode and O gives the overflow mode. N specifies the number of bits in overflow wrap mode. The strings Q, O and N are optional. Vivado HLS also supports floating point data types and operations, as long as these maps to available Xilinx technology core libraries.

2.1.2 Interface Synthesis and Functions

The input arguments and return value of the designed top-level C/C++ function are synthesized into RTL data ports, each with an associated protocol. The RTL data ports can be either input, output or in-out bidirectional ports depending on whether data is read, write or both read and write into the ports respectively. Array arguments in top functions will translate to off chip RAMs . Arrays can be partitioned to increase the speed of access and by default are mapped to single port RAMs. The ports and protocol form a port interface. Port interfaces are used to communicate with other subsystems like the processor in the system. In addition to the port interfaces , block-level protocols and associated ports are used to coordinate the exchanges of data between subsystems.

2.1.3 Area/Resource

By default, Vivado HLS minimizes area, which implies time-sharing of hardware. This generally leads to increased latency and reduced throughput. Latency is defined as the number of clock cycles between applying an input, and achieving the corresponding output. Latency can be viewed at different levels of hierarchy and in the context of loops, latency refers to the completion of all iterations of the loop and the term iteration latency is used when referring to a single iteration. The total latency is equal to the iteration latency, multiplied by the number of iterations of the loop (known as trip count). Latency can also be specified as a design constraint by the user, and the Vivado HLS tool optimizes the design wherever possible to meet the requirement.The Iteration Interval (II) is the number of clock cycles that separate the acceptance of inputs to the Vivado HLS design. Without applying directives, the initiation interval will be one cycle more than the latency, because the default behaviour of Vivado HLS is to optimize for area, resulting in a serial design. However the use of pipeline directive can reduce the iteration interval to much less than the latency of the design. This results in an increase in the area of the design, so there is a trade-off involved. Initiation interval corresponds directly to throughput. Throughput expresses the rate at which data can be passed through the system. The best possible initiation interval is 1, meaning that new input samples can be accepted on every clock cycle, in which case the throughput is equivalent to the clock rate. Higher levels of throughput can be achieved through use of partial loop unrolling, or by replicating a synthesized function.

2.1.4 Pipelining

In HLS, pipelining refers to the partitioning into sub stages of an arbitrary set of dependent operations. The objective for pipelining is to enable parallel processing, and thereby increase the throughput supported by the design. Pipelining can be applied as a directive in Vivado HLS, at the level of functions and loops.

2.1.5 Loops

Loops are basic constructs of any algorithm and expresses operations that are repetitive in nature. Several loop optimization's can be made using directives, enabling architectural variation exploration with almost no change required in the software code.The various optimization's performed in loops are

Loop Unrolling

It means that the hardware inferred from the loop body is created N times. Practically it can be less than N, depending on whether data dependency or memory operations are present in the design. Advantage is throughput increase and disadvantage is the increase in hardware resource.

Partially Unrolled

This generally is a trade-off between rolled and unrolled architecture.

Merging of Loops

This directive can be applied to loops which are occurring one after the other in code and are having the same trip count. Advantages adds in creating the control path of the design as it reduces the number of states in the FSM derived.

Loop Flattening

It applies to nested loops where the overhead in additional clock cycles associated with entering and exiting the inner loops can be avoided which results in improved latency and throughput.

2.1.6 Arrays

As arrays represent storage they are synthesized into memories. The memories inferred are mapped to physical resources on the FPGA as Block RAM, or distributed RAM. Various directives are used to map these memories to physical memory resources. Some of the optimization's on arrays are discussed below.

Resource

It maps an array to a specific memory resource.

Array Map

It maps several small arrays to be combined into a single, larger array. Mapping can be either horizontal (arrays are concatenated to form an array with more elements), or vertical (array elements are combined, resulting in an array with longer words).

Array Partition

It maps the subdivision of a large array into a set of smaller ones. It increase the rate at which memory transactions can take place. In the extreme case, array partitioning will subdivide an array into individual register elements.

Array Reshape

This allows an array with many elements, each with short words, to be reshaped into an array with fewer, longer words.This directive reduces the number of required memory accesses.

Stream

If the array element access is sequential not random, then this directive can be made use of. It reduces the number of ports generated.

2.1.7 Exporting from Vivado HLS

Designs can be exported from Vivado HLS to permit easy integration of Vivado HLS IP with other development tools. IPs are exported from HLS to the IP-XACT format, which allows the module to be integrated into a Vivado IP Integrator design. This results in a zip folder residing in the ip sub-folder under impl folder of the respective solution, and represents the IP catalogue package. It also contains API's required to use the IP in Xilinx SDK environment. This format allows easy sharing and distribution of IP across all platforms.

2.2 Introduction to Vivado IP Integrator

The Vivado IP integrator tool allows to create complex subsystem designs by instantiating and interconnecting IP cores from the Vivado IP Catalogue which contains Xilinx IP's , third party IP's and user IP's. Using the repository manager provided by the tool, we can add user developed IP's especially in Vivado HLS to the current project or can be added permanently to the user IP portion of the Vivado IP catalogue. Various design flows of the FPGA are provided as separate functions on the Vivado IP integrator tool. Even the RTL level designs can also be packaged to an IP using the tool and it will provide an easy and straight forward approach to the incremental building of a complex

design. The tool provides different options to optimize the foot prints of the IP's implementation on FPGA which improves operating frequency, performance, or area and can be specified such that a suitable balance between these three metrics is achieved. This is done very easily in Vivado using a configuration wizard.
In a processor integrated system, the IP's are interfaced with each other generally on an AXI bus which are connected through an AXI interconnect. AXI interconnect can be considered as a switch in computer networks. Currently the standard is AXI4 and there are three variants of bus protocols namely
AXI4
It is provided for memory-mapped IP's, and having the highest performance with an address is supplied followed by a data burst transfer of up to 256 data words. **AXI4-Lite**
It is a simplified link supporting only single data transfer per connection (no bursts). It is also memory-mapped. In this case an address and a single data word are transferred.
AXI4-Stream
This data streaming is provided for non memory mapped IP's and supports high-speed streaming data with burst transfers of unrestricted size.

2.2.1 AXI Architecture

The AXI protocol supports burst based transactions, with each containing address and control information on the address channel. Several AXI masters can be connected to several AXI slaves through an AXI interconnect. An AXI master transfers data to an AXI slave through the AXI interconnect using a write data channel (or a read data channel from slave to master). The write transactions in particular have an additional write response channel, as all data flows from master to slave and as such this is used for the slave to signal completion of a write transaction. The following figures demonstrate communication between AXI master and slave.

The Figure 2.1 shows the write channel architecture where address and control data is passed from master to slave before a burst of data is transmitted, and a write response signalled following completion.The Figure 2.2 shows a read transaction, with address and control data transmitted to the slave before a burst of read data is transmitted to the master.

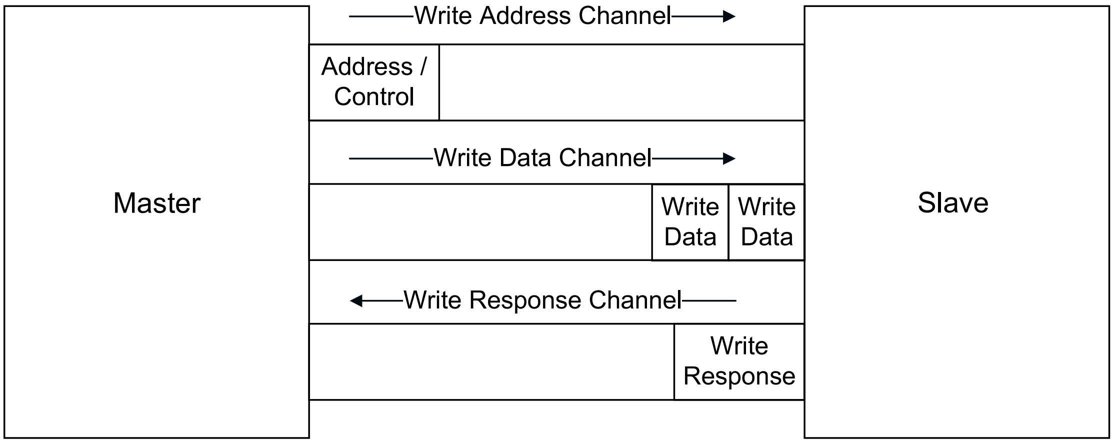


Figure 2.1: AXI4 write channel architecture

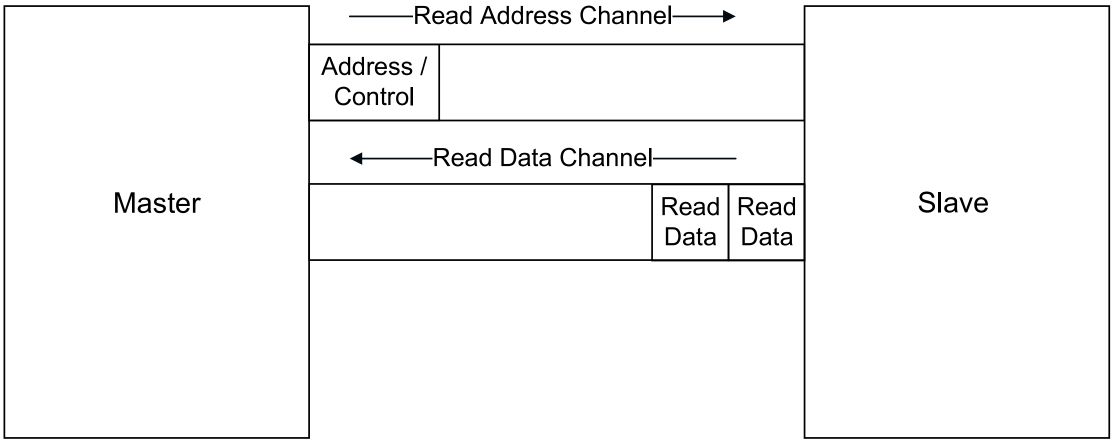


Figure 2.2: AXI4 read channel architecture

2.2.2 AXI Transactions

Write-Burst Transaction

Suppose we need to write burst of data to an address A. The master drives the slave and transaction begins with the sending of address and control information via the signal AWADDR. Following confirmation of a valid address with AWVALID, a signal is sent to confirm that the system is ready for the transaction on AWREADY. The master then sends the data blocks in the burst of data to the slave on the WDATA signal, with the final data item being indicated through the WLAST signal going high and confirming the completion of the transaction. The master also sends various control signals regarding data bursts.

Read-Burst Transaction

In case of a read-burst transaction using AXI4 for data being read from an address A, the slave is driven by the master through transmission of address and control information in signal ARADDR. ARVALID goes high signalling a valid address and the system is confirmed ready for transmission with the signal ARREADY. Data blocks are read from address A via the signal RDATA, and as before the final data block is indicated via signal RLAST. The RVALID signal kept low by the slave until read data is available.

2.3 Introduction to Xilinx SDK

Xilinx SDK provides an environment for creating software platforms and applications for Xilinx processor cores.It works with hardware designs generated with Vivado.
SDK is provided with

1. Reference Software Applications
   Applications like lwip echo server which we made use of for building this project.
2. XMD
   Xilinx Microprocessor Debugger is debug agent used to communicate with Xilinx embedded processors.
3. XSDB
   Xilinx System Debugger is a command-line interface to debug the Xilinx hw_server.
4. FPGA programmer
   Used to program the Xilinx FPGA with the bitstream.

5. Flash programmer
   Used for burning bitstream and software application images into external parallel NOR Flash devices.

6. Bootloader generator
   Used for automatically bootloading your embedded software applications from parallel Flash.

The two main terminologies used in Xilinx SDK are hardware platform and software platform.The hardware platform is the embedded hardware design that is created in Vivado and exported in the form of an HDF/XML file through the use of export hardware wizard. Once the hardware platform is identified and imported, we create the software platform. A software platform is a collection of libraries and drivers that form the lowest layer of application software stack. The software applications must run on top of a given software platform, using the provided API's. Therefore before creating and use software applications in SDK, a software platform project must be created. SDK includes the following two software platform types. They are

1. Standalone OS
   It is a simple and single-threaded environment that provides basic features like standard input/output and access to processor hardware features. In this project we extensively used this software platform.

2. Xilkernel
   A simple and lightweight kernel that provides POSIX-style services such as scheduling, threads, synchronization, message passing, and timers.

CHAPTER 3

Implementation of Transmitter

This chapter gives information about the various algorithms that were used to implement the system using Vivado HLS.

3.1 System Description

The 5G transceiver block diagram of the system is show in the Figure 3.1.

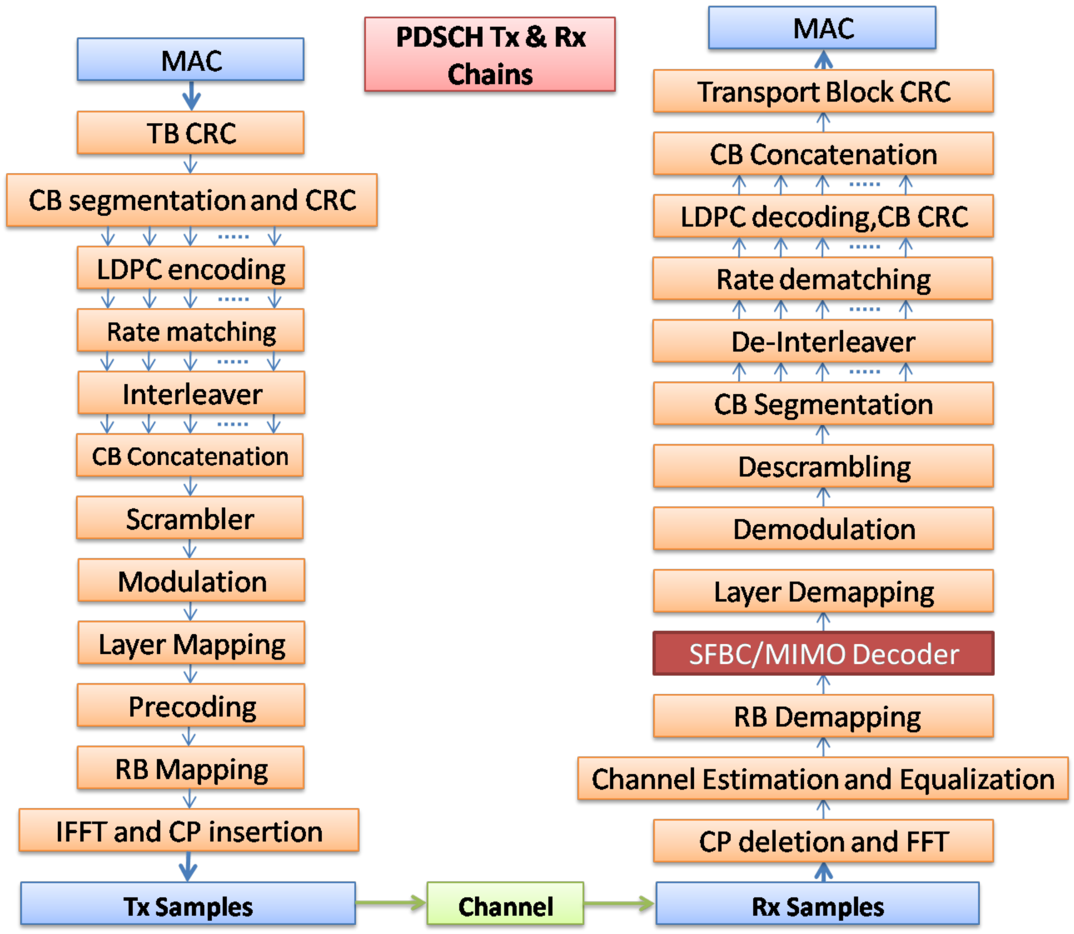


Figure 3.1: Block diagram of a 5G transceiver system

Following blocks are implemented in Transmitter:

- Modulation
- Scrambler
- Layer Mapping
- RB Mapping
- IFFT
- VRB-PRB Mapping

3.2 Modulation

The modulation mapper takes binary digits, 0 or 1, as input and produces complex-valued modulation symbols as output.All these complex valued modulation symbols are stored in the form of look up tables.

3.2.1 BPSK

In case of BPSK modulation, bit ***b(i)*** is mapped to complex valued modulation symbol ***x*** according to

$$x = \frac{1}{\sqrt{2}}\left[(1-2b(i)) + j(1-2b(i))\right]$$

3.2.2 QPSK

In case of QPSK modulation, pair of bits ***b(i),b(i+1)*** are mapped to complex valued modulation symbol ***x*** according to

$$x = \frac{1}{\sqrt{2}}\left[(1-2b(i)) + j(1-2b(i+1))\right]$$

3.2.3 16 QAM

In case of 16 QAM modulation, quadruplets of bits ***b(i),b(i+1),b(i+2),b(i+3)*** are mapped to complex valued modulation symbol ***x*** according to

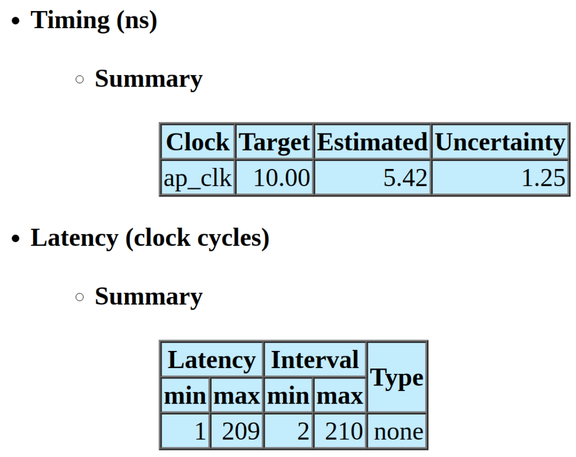
$$x = \frac{1}{\sqrt{10}}\{(1-2b(i))[2-(1-2b(i+2))] + j(1-2b(i+1))[2-(1-2b(i+3))]\}$$

3.2.4 64 QAM

In case of 64 QAM modulation, hextuplets of bits ***b(i),b(i+1),b(i+2),b(i+3),b(i+4),b(i+5)*** are mapped to complex valued modulation symbol ***x*** according to

$$x = \frac{1}{\sqrt{42}}\{(1-2b(i))[4-(1-2b(i+2))[2-(1-2b(i+4))]] + j(1-2b(i+1))[4-(1-2b(i+3))[2-(1-2b(i+5))]]$$

Following figures shows Performance and Utilization estimates of this block in HLS.



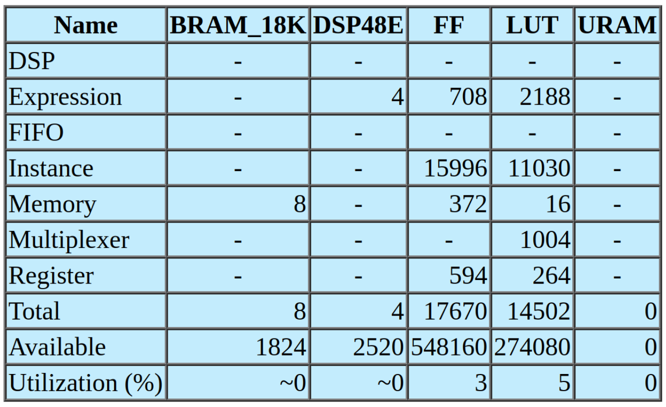
- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.42	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
1	209	2	210	none

Figure 3.2: Timing performance of Modulator



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	4	708	2188	-
FIFO	-	-	-	-	-
Instance	-	-	15996	11030	-
Memory	8	-	372	16	-
Multiplexer	-	-	-	1004	-
Register	-	-	594	264	-
Total	8	4	17670	14502	0
Available	1824	2520	548160	274080	0
Utilization (%)	~0	~0	3	5	0

Figure 3.3: Resource utilization of Modulator

3.3 Scrambler

Block of bits ***b(0),....b(M-1)*** where M is number of bits transmitted, is scrambled prior to modulation, resulting in a block of scrambled bits ***x(0),....x(M-1)*** according to

$$x(i) = (b(i) + c(i))mod2$$

where scrambling sequence ***c(i)*** is generated using Pseudo Random Sequence Generator. The Scrambling sequence generator is initialized with

$$c_{init} = n_{RNTI}.2^{15} + q.2^{14} + n_{ID}$$

3.3.1 Pseudo Random Sequence Generator

Pseudo random sequences are defined by a length 31 Gold sequence. The output sequence c(n) is defined by

$$c(n) = (x_1(n + N_c) + x_2(n + N_c))mod2$$

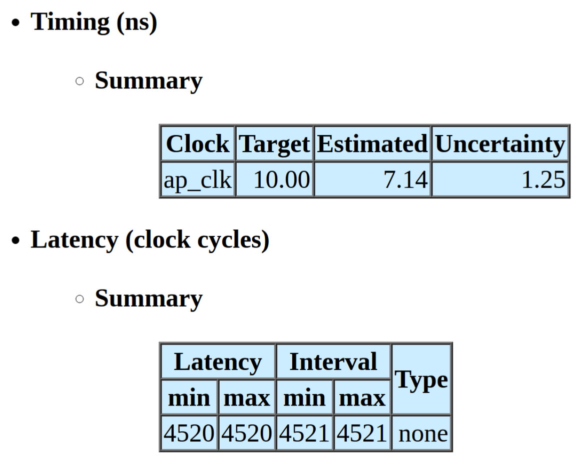
$$x_1(n + 31) = (x_1(n) + x_1(n + 3))mod2$$

$$x_2(n + 31) = (x_2(n) + x_2(n + 1) + x_2(n + 2) + x_2(n + 3))mod2$$

where $N_c = 1600$ and $x_1(n)$ is initialized with $x_1(0) = 1, x_1(n) = 0, n = 1, ...30$. The initialization of second sequence is denoted by

$$c_{init} = \sum_{i=0}^{30} x_2(i).2^i$$

Following figures shows Performance and Utilization estimates of scrambler block in HLS.



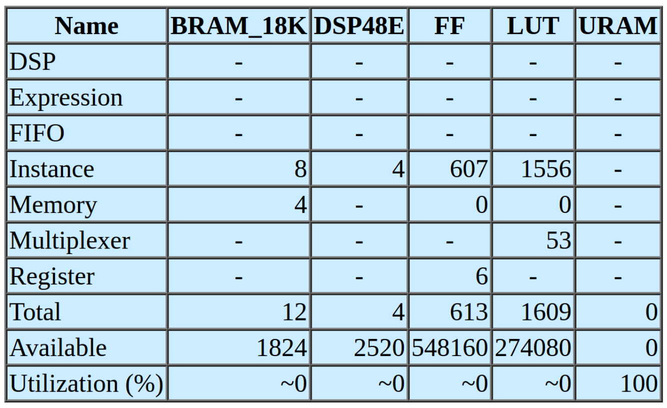
- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.14	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
4520	4520	4521	4521	none

Figure 3.4: Timing performance of Scrambler

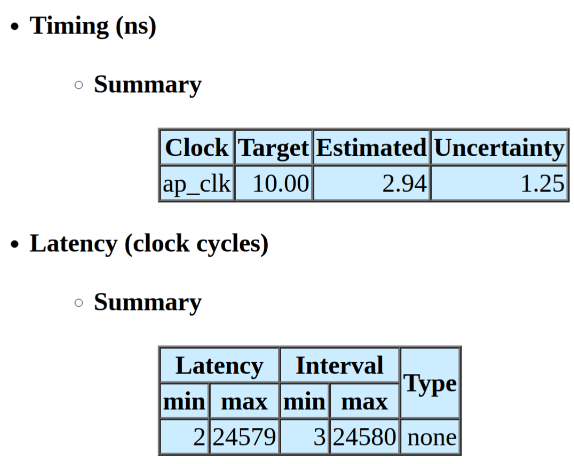


Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	8	4	607	1556	-
Memory	4	-	0	0	-
Multiplexer	-	-	-	53	-
Register	-	-	6	-	-
Total	12	4	613	1609	0
Available	1824	2520	548160	274080	0
Utilization (%)	~0	~0	~0	~0	100

Figure 3.5: Resource utilization of Scrambler

3.4 Layer Mapping

Layer Mapper splits the output from modulator into multiple layers. Following figures shows Performance and Utilization estimates of Layer Mapper block in HLS.



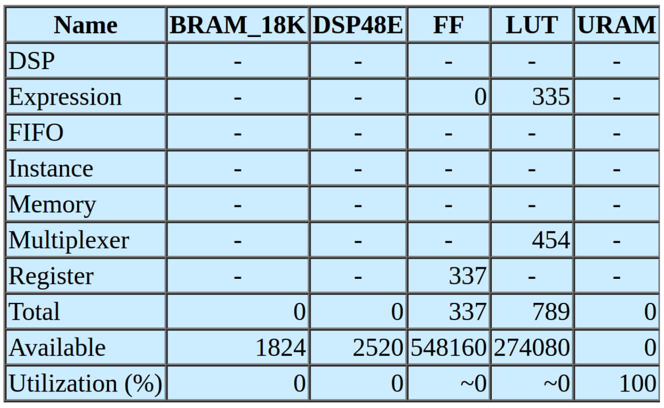
- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.94	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
2	24579	3	24580	none

Figure 3.6: Timing performance of Layer Mapper

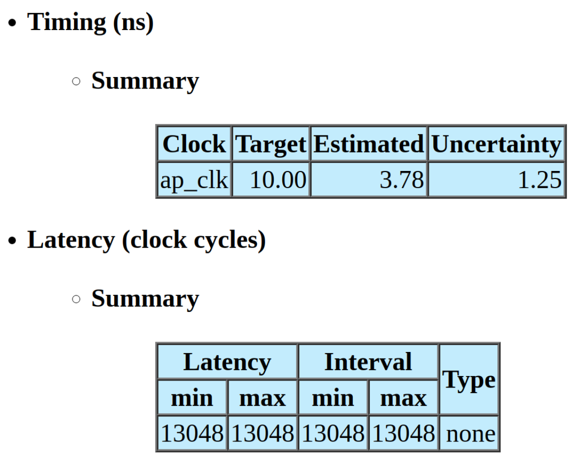


Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	335	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	454	-
Register	-	-	337	-	-
Total	0	0	337	789	0
Available	1824	2520	548160	274080	0
Utilization (%)	0	0	~0	~0	100

Figure 3.7: Resource utilization of Layer Mapper

3.5 RB Mapping

RB Mapper maps input data,pilots and control Info into 600x14 matrix respective positions provided by frame structure,which is of size 600X14. Where 0,1,2 corresponds to data,pilot,control info respectively.Input to FFT block should be in the range of [-1,1). So output of the RB mapper is scaled down by a factor of 2 because output of modulation block can take maximum value of 7/sqrt(42) in 64-QAM. Following figures shows Performance and Utilization estimates of this block in HLS.

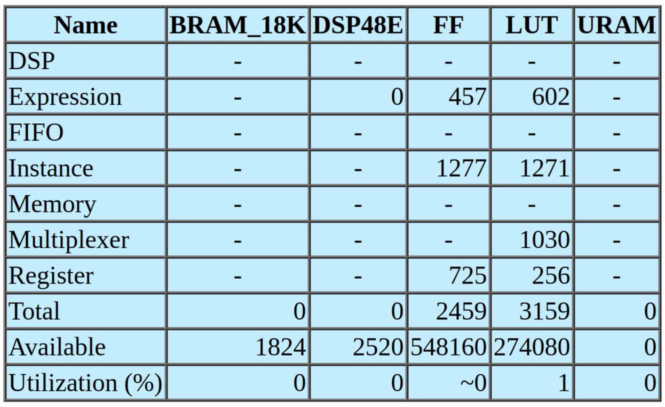

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	3.78	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
13048	13048	13048	13048	none

Figure 3.8: Timing performance of RB Mapper



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	0	457	602	-
FIFO	-	-	-	-	-
Instance	-	-	1277	1271	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	1030	-
Register	-	-	725	256	-
Total	0	0	2459	3159	0
Available	1824	2520	548160	274080	0
Utilization (%)	0	0	~0	1	0

Figure 3.9: Resource utilization of RB Mapper

3.6 IFFT

IFFT(1024 point) is implemented using Xilinx FFT IP.The Xilinx FFT IP block can be called within a C++ design using the library `hls_fft.h`. To use the FFT in C++ code:

- Include the `hls_fft.h` library in the code.
- Set the default parameters using the pre-defined struct `hls::ip_fft::params_t`.
- Define the run time configuration.
- Call the FFT function.

Define the static parameters of the FFT. This includes such things as input width, number of channels, type of architecture. which do not change dynamically. The FFT library includes a parameterization struct `hls::ip_fft::params_t`, which can be used to initialize all static parameters with default values. Default values for output ordering,input width,output width and phase factor width are over ridden using a user defined struct config1 based on the pre-defined struct.

```
struct config1 :  hls::ip_fft::params_t {
  static const unsigned ordering_opt =1;
  static const unsigned input_width = 32;
  static const unsigned output_width = 32;
  static const unsigned phase_factor_width = 24;
};
```

where 1 refers to natural_order of output. set the run time configuration. The direction of the FFT (Forward or Inverse) based on the value of variable direction and also set the value of the scaling schedule.

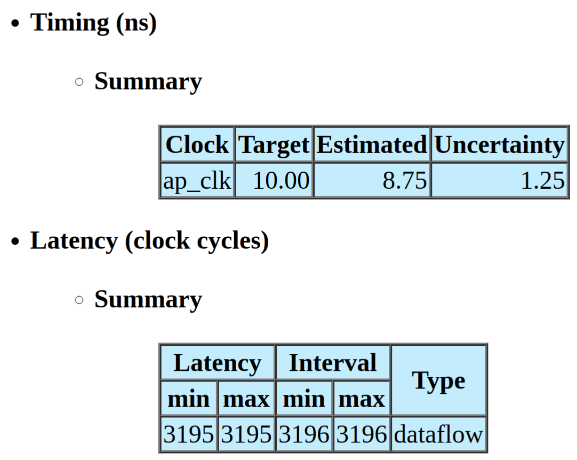
```
// direction=0 refers to ifft, direction=1 refers to fft
fft_config1.setDir(direction);
fft_config1.setSch(0x2AB);
```

Output scaling is done for both Forward and Inverse FFT. FFT function is called in the following way. The function parameters are, in order, input data, output data,output status and input configuration.

```
hls::fft<config1>(xn1,xk1,&fft_status1,&fft_config1);
```

Refer to LogiCORE IP Fast Fourier Transform Product Guide (PG109) for details on the parameters and the implication for their settings.

Performance and Utilization estimates of this block in HLS is shown below.



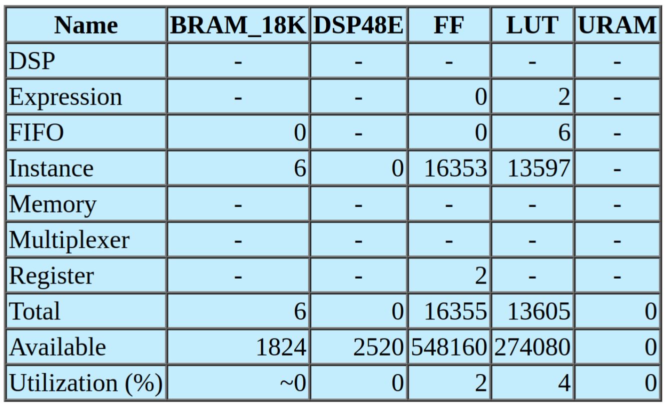
- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
3195	3195	3196	3196	dataflow

Figure 3.10: Timing performance of FFT



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	0	-	0	6	-
Instance	6	0	16353	13597	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	2	-	-
Total	6	0	16355	13605	0
Available	1824	2520	548160	274080	0
Utilization (%)	~0	0	2	4	0

Figure 3.11: Resource utilization of FFT

3.7 VRB-PRB Mapping

Virtual resource blocks are mapped to physical resource blocks according to the indicated mapping scheme, non-interleaved or interleaved mapping.

For non-interleaved VRB to PRB mapping, virtual resource block *n* is mapped to physical resource block *n*.

For interleaved VRB to PRB mapping, the mapping process is defined in terms of resource block bundles: The set of Nsize,resource blocks in bandwidth part i with starting position Nstart are divided into $Nbundle = ceil((Nsize + (Nstart)modL)/L)$ resource-block bundles in increasing order of the resource block number and bundle number where L is the bundle size provided by the higher layer parameter VRB to PRB interleaver.

- Resource block bundle 0 consists of $L - ((Nstart)modL)$.
- Resource block bundle Nbundle-1 consists of $(Nstart + Nsize)modL$ resource blocks if $(Nstart + Nsize)modL$>0 and L resource blocks otherwise.
- All other resource block bundles consists of L resource blocks.

Virtual resource blocks in the interval j ϵ {0,1,...Nbundle-1} are mapped to physical resource blocks according to

$$f(j) = rC + c$$

$$j = cR + r$$

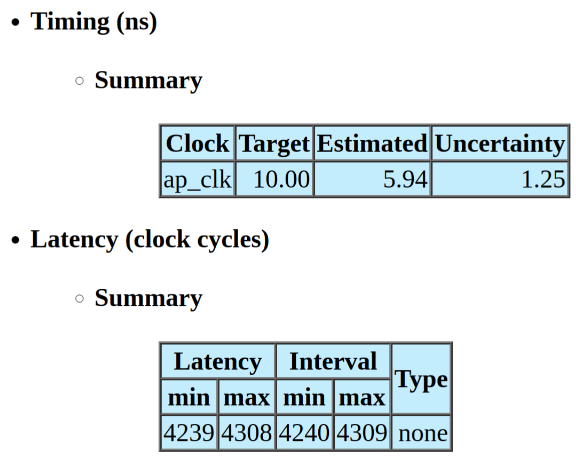
$$r = 0, 1, ...R - 1$$

$$c = 0, 1, ...C - 1$$

$$R = 2$$

$$C = floor(Nbundle/R)$$

Performance and Utilization estimates of this block in HLS is shown below.



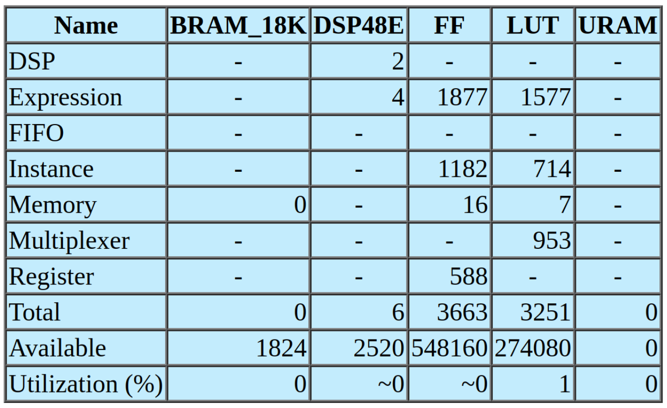
- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.94	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
4239	4308	4240	4309	none

Figure 3.12: Timing performance of VRB-PRB Mapping



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	2	-	-	-
Expression	-	4	1877	1577	-
FIFO	-	-	-	-	-
Instance	-	-	1182	714	-
Memory	0	-	16	7	-
Multiplexer	-	-	-	953	-
Register	-	-	588	-	-
Total	0	6	3663	3251	0
Available	1824	2520	548160	274080	0
Utilization (%)	0	~0	~0	1	0

Figure 3.13: Resource utilization of VRB-PRB Mapping

CHAPTER 4

Integration and Results

This chapter gives information about integration of transmitter and receiver on Zynq ultra scale board.

4.1 Method of on-board testing and results

- The first step towards validating the design has been completed in the Vivado-HLS tool through the various design flows namely C-Simulation, Synthesis, C/RTL co-simulation.
- After co-simulation status is pass export the IP from Vivado HLS to Vivado.
- In Vivado all the IP's generated from HLS and Uart IP are connected to the Zynq ultra scale processor.
- Uart baud rate is set to 115200.
- Block diagram of the complete system is shown below

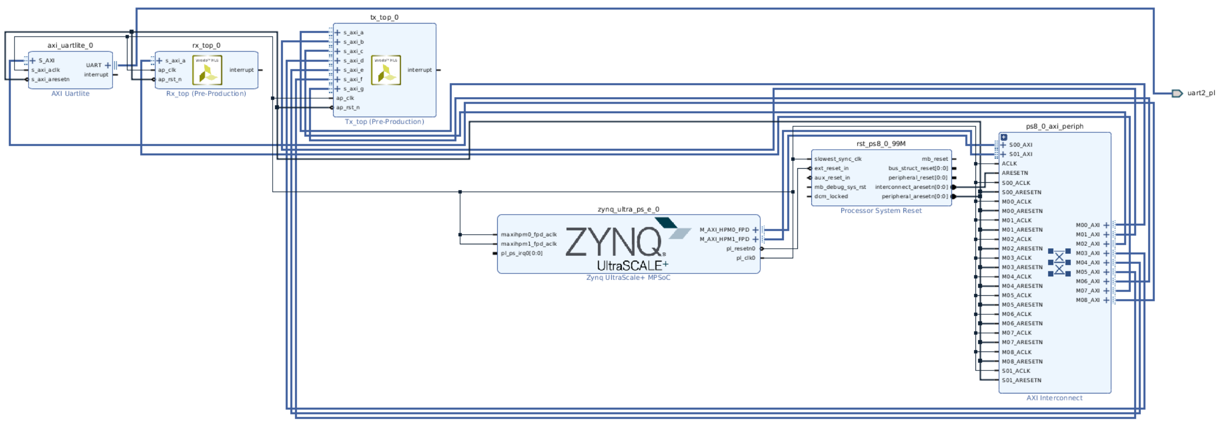
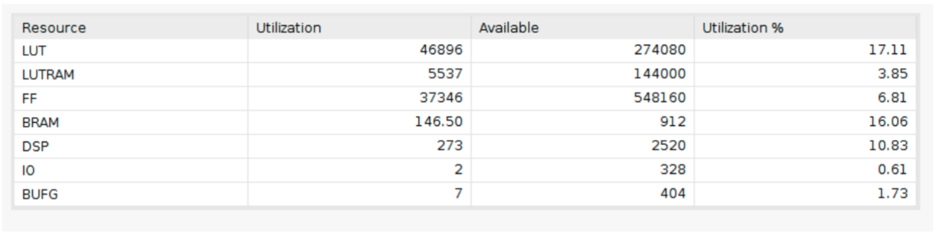


Figure 4.1: Block Diagram of Transmitter and Receiver system

- After Validating the design generate bitstream.
- Utilization estimates of complete transmitter and receiver chain is shown in the following figure



Resource	Utilization	Available	Utilization %
LUT	46896	274080	17.11
LUTRAM	5537	144000	3.85
FF	37346	548160	6.81
BRAM	146.50	912	16.06
DSP	273	2520	10.83
IO	2	328	0.61
BUFG	7	404	1.73

Figure 4.2: Utilization estimates of Transmitter and Receiver system

- Bit stream is exported to Vivado SDK.
- Input parameters of tx and rx blocks are set by using following functions
  XTx_top_Set_in_para_in_size(tx_topptr,4128);
  XTx_top_Set_in_para_modulation_type_V(tx_topptr,6);
  XTx_top_Set_rs_index1_V(tx_topptr,0);
  XTx_top_Set_nresources1_V(tx_topptr,50);
  XTx_top_Write_in_V_Words(tx_topptr,0,in1,172);
  XTx_top_Write_frame_strucure1_V_Bytes(tx_topptr,0,fr,8400);
  XRx_top_Write_channel_M_real_V_Words(rx_topptr,0,&l,4128);
  XRx_top_Write_channel_M_imag_V_Words(rx_topptr,0,&l,4128);
  int v= (1«5);
  XRx_top_Set_N0_V(rx_topptr,v);
  XRx_top_Set_in_para_in_size(rx_topptr,688);
  XRx_top_Set_in_para_modulation_type_V(rx_topptr,6);
  XRx_top_Set_rs_index1_V(rx_topptr,0);
  XRx_top_Set_nresources1_V(rx_topptr,50);
  XRx_top_Write_frame_strucure1_V_Bytes(rx_topptr,0,fr,8400);
  XRx_top_Write_inp_Words(rx_topptr,0,kk,28672);
- Output from receiver is read by using following function
  XRx_top_Read_output_V_Words(rx_topptr,0,b1,4128);

Received output from rx module is compared with rx block in HLS. And both the outputs are matched.

Iteration interval for Tx module is 13136 clock cycles. Clock cycle period is 10ns. Tx blocks input size is of 4128 bits so input bit rate is 31.425Mbps.

Appendix

Problems faced and solved during the Project :

- Based on the precision of the variable N0 in demodulation block, output precision of the demodulation block should be adjusted such that output of the demodulation block should not overflow.
- Output of FFT block in receiver chain has precision of <32,1> typecasting it to <16,2> (output of resource de mapper)in resource de mapper precision will be lost. So it is type casted to <32,2>, left shift it by 13 and then typecasted back to <16,2>.The return type of the shift left operation is the same width as the type being shifted. Refer to page no 654 in UG902 (v2017.2) June 7, 2017.
- While testing the individual blocks test for all possible cases. For example in demodulation block test for all modulation types bpsk, qpsk, 16qam, 64qam.
- Integrate the complete transmitter and receiver system inside example fft project given by Xilinx. Otherwise exporting of the IP to Vivado will create issues.
- In SDK inputs are given in the form of integers (even for fixed point data types). So give the fixed point inputs in its equivalent integer format. For example consider ap_fixed<10,5> k=1 in HLS corresponding input in SDK is 32(1 is left shifted by 5).

REFERENCES

1. Xilinx, Vivado Design Suite User Guide High-Level Synthesis (UG902). Xilinx (v2017.2) User Guide, 2017.

2. Xilinx, Vivado Design Suite User Guide Implementation (UG904). Xilinx (v2017.2) User Guide, 2017.

3. LogiCORE IP Fast Fourier Transform Product Guide (PG109).

4. Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973).

5. 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; NR; Physical channels and modulation (Release 15) (3GPP TS 38.211 V15.1.0 (2018-03)).