SOFTWARE DEFINED RADIO IMPLEMENTATION USING PARTIAL DYNAMIC RECONFIGURATION

By

Mohamed Abbas Abd El-hameed

Mohamed Adel Attia EL-Hady

Mohamed Fawzy Mohamed

Mohamed Nafea Mohamed

Omar Sherif Salama

Yahia Khaled Ebn EL-Walid

Ziad Ibrahim Abdelati

Under supervision of

Associate Prof. Yasmine A.H. Fahmy

Associate Prof. Hassan Mostafa

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

Faculty of Engineering, Cairo University

Giza, Egypt

July 2016

Table of Contents

List of Tables viii

List of Figures x

List of Symbols and Abbreviations xviii

Acknowledgment xx

Abstract xxi

Motivation xxii

Chapter 1: Introduction 1

Chapter 2: Survey 3

Chapter 3: SDR & PDR Concepts 5

3.1 SDR by Definition 5

3.2 Why SDR? 5

3.2.1 Field Programmable Gate Arrays 6

3.2.2 What is inside the FPGA? 7

3.3 FPGA Configuration 8

3.3.1 Configuration Definition 8

3.3.2 Types of Configuration 9

3.3.3 What is Partial Reconfiguration (Problems to solve)? 10

3.4 Partial Reconfiguration of FPGAs 10

3.4.1 Partial Reconfiguration Enables 11

3.4.2 Brief History of Partial Reconfiguration 12

3.4.3 Partial Reconfiguration Analogy 13

3.4.4 System Flexibility: Communication Hub 13

3.4.5 Size and Cost Reduction: Time Multiplexing 13

3.4.6 Power Reduction Techniques with PR 13

3.4.7 Styles of Partial Regions 15

3.4.8 Full bit file V.s Partial bit file 16

3.4.9 Types of Partial Reconfiguration 17
3.4.10 Benefits of Partial Reconfiguration 17
3.4.11 Requirements of PDR 18
3.4.12 Partition Pins 18
Chapter 4: 3G transmitter and receiver implementation 21
4.1 Frame structure 22
4.2 3G Transmitter 24
4.2.1 CRC (Cyclic Redundancy Check) attachment 25
4.2.2 Segmentation 26
4.2.3 Encoder 31
4.2.4 Code block concatenation 34
4.2.5 Radio frame equalization 36
4.2.6 First interleaving 38
4.2.7 Radio Frame Segmentation 43
4.2.8 Second interleaving 44
4.2.9 Interleaving Block 47
4.2.10 Spreading and Scrambling 48
4.2.11 Modulation 51
4.3 3G Receiver 53
4.3.1 Channel modeling 53
4.3.2 Demapper 54
4.3.3 Despreading block 56
4.3.4 Deconcationation 58
4.3.5 Deinterleaver 61
4.3.6 Desegmentation Block 62
4.3.7 CRC (Cyclic Redundancy Check) check 64
Chapter 5: Wi-Fi Standard Transmitter and Receiver 67

5.1 Overview 67

5.2 Standard History 67

5.3 Frequency and Data Rates 67

5.4 Physical Layer of 802.11a 68

5.5 PPDU frame structure 68

5.5.1 SIGNAL field 69

5.5.2 RATE field 70

5.5.3 PLCP LENGTH field 70

5.5.4 Parity (P), Reserved (R), and SIGNAL TAIL fields 70

5.5.5 DATA field 70

5.5.6 SERVICE field 71

5.5.7 PPDU TAIL field 71

5.5.8 Pad bits (PAD) 71

5.5.9 PLCP preamble: 72

5.5.10 Frame Summary points 72

5.6 802.11a Transmitter PHY Block Diagram 72

5.6.1 Scrambler 73

5.6.2 Convolutional Encoder 76

5.6.3 Puncture 77

5.6.4 Interleaver 81

5.6.5 Modulation Mapper 85

5.6.6 IFFT Modulation 88

5.6.7 Preamble 97

5.7 802.11a Receiver PHY Block Diagram 99

5.7.1 DeMapper 99

5.7.2 DeInterleaver 101

5.7.3 Depuncture 103

5.7.4 Convolutional Decoder: 106

5.7.5 Descrambler 115

Chapter 6: LTE transmitter implementation 117

6.1 Scrambler 122

6.2 Cyclic redundancy check (CRC) 122

6.3 Code block segmentation and code block CRC attachment 122

6.4 Turbo encoder 126

6.4.1 Trellis termination for turbo encoder 127

6.4.2 Turbo Code Internal Interleaver 128

6.5 Rate matching for turbo coded transport channels 132

6.5.1 Sub-block interleaver 132

6.5.2 Bit collection 135

6.5.3 Bit selection 136

6.6 Code block concatenation 139

6.7 Divider 141

6.7.1 Sequential Divider 141

Combinational Divider 141

6.8 Modulation 142

6.9 IFFT 144

Chapter 7: Verification Methodology 149

7.1 Functional Verification 149

7.2 IFFT testing 151

7.3 Post-Synthesis Simulation 152

7.4 Fixed Point Simulation 153

Chapter 8: Hardware implementation 155

8.1 FPGA 155

8.1.1 Softcore and Hardcore processors 156

8.1.2 Xilinx Virtex-5 157
8.1.3 MicroBlaze softcore processor 159
8.2 Hardware System Hierarchy 160
8.3 Partial dynamic reconfiguration flow 163
8.3.1 Generation of netlist files (XST Synthesis) 164
8.3.2 Embedded Development Kit (EDK) 167
8.3.3 Xilinx Platform Studio 168
8.3.4 Software Development Kit 179
8.3.5 Plan-Ahead 184
8.3.6 Generation of static system bootable ace files 197
8.3.7 Summary of the hardware implementation flow 197
Chapter 9: Testing 201
9.1 Communication Methodology 201
9.2 User Logic Code 201
9.1 SDK C Code 204
9.1.1 Sysace_read 204
9.1.2 bus2rp 206
9.1.3 rp2bus 207
9.1.4 Sysace_write 207
Chapter 10: Multiple Reconfigurable Partitions (RPs) 209
Chapter 11: Results 215
Chapter 12: Conclusion and Recommendations 217
Conclusion: 217
Recommendations 217
12.1 3G 217
12.2 LTE 217
12.3 WIFI 219

12.4 Others Improvements 219

References: 220

List of Tables

Table 4.1: Types of CRC--- 25

Table 4.2: pins description of CRC-- 26

Table 4.3: Interface Signal Decleration -- 30

Table 4.4: pins description of encoder -- 33

Table 4.5: Code block concatenation block signals declaration------------------------ 35

Table 4.6: number of frames-- 37

Table 4.7-Frame Equalization Pin description --- 37

Table 4.8: comparison between with and without interleaving----------------------- 38

Table 4.9-Inter-column permutation for first interleaver ------------------------------ 39

Table 4.10-First Interleaver Pin description-- 41

Table 4.11-Radio Frame Segmentation Pin description -------------------------------- 44

Table 4.12-inter-column permutation for second interleaver ------------------------- 45

Table 4.13 : Second Interleaver pin description -- 45

Table 4.14: pins description of spreading block --- 51

Table 4.15: BPSK mapping--- 52

Table 4.16: pin description of mapper module--- 52

Table 4.17: Ports description of noise-- 54

Table 4.18 : Pins description of demapper -- 55

Table 4.19: pins description of despreading-- 57

Table 4.20: how to calculate z--- 58

Table 4.21 : Description of deconationation block -------------------------------------- 60

Table 4.22: The relationship between tti and nmber of frames. ---------------------- 61

Table 4.23: columns arrangement in first deinterleaver. ------------------------------ 62

Table 4.24: Desegmentation block signals declaration --------------------------------- 64

Table 4.25: Equations of CRC check--- 64

Table 4.26: Pins description of DeCRC-- 65

Table 5.1: Contents of the SIGNAL field -- 70

Table 5.2: Scrambler Signals Declaration --- 75

Table 5.3: Convolution Encoder Signals Declaration----------------------------------- 77

Table 5.4: Data Rates and Puncture types --- 79

Table 5.5: Puncture Pin description --- 80

Table 5.6: Interleaver block signals declaration --- 83

Table 5.7: Modulation-dependent parameters -- 85
Table 5.8: Normalization factor for all modulation modes.-------------------------- 85
Table 5.9: Pin description of wifi mapper. -- 87
Table 5.10: top_ofdm_wifi pin description-- 96
Table 5.11 : Frequency domain representation of the short sequences. ------------ 99
Table 5.12: Pins description of the demapper--------------------------------------- 101
Table 5.13: Output Distance in case of rate 1/2------------------------------------- 113
Table 5.14: Output Distance in case of rate 1/3------------------------------------- 114
Table 6.1-K Values -- 124
Table 6.2-Segmentation state description -- 125
Table 6.3 : Inputs and outputs of turbo encoder. ----------------------------------- 131
Table 6.4:Inter-column permutation pattern for sub-block interleaver. ----------- 134
Table 6.5:Interleaver block signals declaration ------------------------------------- 134
Table 6.6:Bit collection block signals declaration---------------------------------- 136
Table 6.7:Bit selection block signals declaration ----------------------------------- 138
Table 6.8: Rate Matching block signals declaration -------------------------------- 140
Table 6.9: BPSK mapping -- 142
Table 6.10: QPSK mapping --- 142
Table 6.11: 16-QAM mapping --- 143
Table 6.12: Pin discription od LTE mapper module -------------------------------- 143
Table 6.13: Basic transmission schemes [5] -- 146
Table 10.1-multiple RPs-Expected outputs -- 214

List of Figures

Figure 3.1: FPGA and ASIC total cost V.s production volume 7
Figure 3.2:FPGA Internal structure 8
Figure 3.3:FPGA Layers 9
Figure 3.4: FPGA configuration types 10
Figure 3.5: Reconfigurable FPGA Structure 12
Figure 3.6: PR analogy 13
Figure 3.7: FPGA as communication Hub 14
Figure 3.8: FPGA time multiplexing 14
Figure 3.9: Partial regions styles 16
Figure 3.10: Partition pins 19
Figure 4.1: Frame structure 22
Figure 4.2: DPCCH Field 23
Figure 4.3: Bit patterns of TPC 23
Figure 4.4: Pilots for uplink DPCCH 24
Figure 4.5: Transmitter blocks 24
Figure 4.6: CRC as shift register 25
Figure 4.7: Top controlled CRC 26
Figure 4.8: Internal module of CRC 27
Figure 4.9: Segmentation Interface 29
Figure 4.10: Internal Design for Segmentation 29
Figure 4.11: flag filler explanation use 30
Figure 4.12: Output from the segmentation block 31
Figure 4.13: clk and clk_fast period 32
Figure 4.14: Rate 1/2 convolutional encoder 32
Figure 4.15: Rate 1/3 Convolutional encoder 32
Figure 4.16: Schematic 33
Figure 4.17: Internal block diagram 34
Figure 4.18: Code block concatenation block interface. 35
Figure 4.19: Code block concatenation block simulation. 36
Figure 4.20: Radio Frame Equalization interface 37
Figure 4.21: Equalizer timing diagram example 38
Figure 4.22: Steps of Interleaving 39

Figure 4.23: Interleaving Example......40
Figure 4.24: First Interleaver interface......40
Figure 4.25: First Interleaver Block Diagram......41
Figure 4.26: First interleaver timing diagram......42
Figure 4.27: Radio Frame Segmentation interface......43
Figure 4.28: Radio Frame Segmentation Block Diagram......44
Figure 4.29: Radio Frame Segmentation timing diagram......44
Figure 4.30: Second Interleaver interface......46
Figure 4.31: Second Interleaver Block Diagram......46
Figure 4.32: Second interleaver timing diagram 1......47
Figure 4.33: Second interleaver - timing diagram 2......47
Figure 4.34: Interleaving Block Diagram......48
Figure 4.35: Interleaving Block Timing Diagram......48
Figure 4.36: waveform of spreading block......49
Figure 4.37: Code-tree for generation of Orthogonal Spreading Factor codes......50
Figure 4.38: block diagram of spreading......50
Figure 4.39: Top controlled spreading......50
Figure 4.40: Internal structure of spreading block......51
Figure 4.41: Mapper interface......52
Figure 4.42: Detailed block of mapper......53
Figure 4.43: receiver blocks......53
Figure 4.44: noise top block......54
Figure 4.45: constellation of bpsk......55
Figure 4.46: Wavefom of demapper......55
Figure 4.47: Top demapper 3g......56
Figure 4.48: top despreading and descrambling......57
Figure 4.49: Top deconcationation......60
Figure 4.50: Internal design of deconcationation......61
Figure 4.51 : Block diagram of deinterleaver.......61
Figure 4.52: top desegmentation......63
Figure 4.53 : Top CRC check......65
Figure 4.54: Internal structure of CRC......65
Figure 5.1: PPDU frame format......69
Figure 5.2: SIGNAL field bit assignment......70

Figure 5.3: SERVICE field bit assignment 71
Figure 5.4: Wi-Fi Tx Block Diagram 72
Figure 5.5: Data Scrambler 73
Figure 5.6: Top Controlled Scrambler Interface 75
Figure 5.7: Internal Signals for Scrambler 75
Figure 5.8: Convolutional Encoder (K=7) 76
Figure 5.9: Schematic of the convolutional encoder 77
Figure 5.10: Internal block diagram of the convolutional encoder 78
Figure 5.11: Puncturing example 1 79
Figure 5.12: Puncture vector 79
Figure 5.13: Puncture interface 80
Figure 5.14: Puncture block diagram 81
Figure 5.15: Puncture 3/4 timing diagram - write enable 81
Figure 5.16: Puncture 3/4 timing diagram - clock ratio 81
Figure 5.17: Interleaver block interface 83
Figure 5.18: Interleaver block simulation 84
Figure 5.19: Writing the input data in the RAM 84
Figure 5.20: Reading the data from the RAM 84
Figure 5.21: Modulation constellations for BPSK, QPSK, 16-QAM, and 64-QAM. 86
Figure 5.22: Wi-Fi mapper interface. 87
Figure 5.23:Detailed block diagram of Wi-Fi mapper module. 88
Figure 5.24:Spectrum of a single subcarrier of the OFDM signal (a), Spectrum of the OFDM signal (b) 88
Figure 5.25: OFDM training structure 89
Figure 5.26: inputs and outputs of the IFFT [8] 89
Figure 5.27: Final 64 subcarrier mapping [8] 90
Figure 5.28: pipelined streaming I/O [11] 91
Figure 5.29: IFFT block interface [11] 94
Figure 5.30: FFT timing for applying data 95
Figure 5.31: top_ofdm_wifi interface 95
Figure 5.32: top_ofdm_wifi hierarchy 96
Figure 5.33: Waveform of preample and IFFT outputs 96
Figure 5.34: OFDM training structure. 97

Figure 5.35: Interface of Wi-Fi preamble generator 98
Figure 5.36: Wi-Fi Rx Block Diagram 99
Figure 5.37: decision regions of WIFI demapper 100
Figure 5.38: Top Demapper 100
Figure 5.39: Deinterleaver block interface..102
Figure 5.40: Deinterleaver block simulation...103
Figure 5.41: Writing the input data in the RAM...103
Figure 5.42: Reading the data from the RAM...103
Figure 5.43 : The interface of depuncture block diagram....................................104
Figure 5.44 : internal block of depuncture..104
Figure 5.45: Depuncture 3/4 rate procedure 105
Figure 5.46 : Depuncture 2/3 rate procedure 105
Figure 5.47: Trellis diagram of Convolutional Encoder 106
Figure 5.48: Block Diagram of Viterbi decoder 107
Figure 5.49: Viterbi decoder algorithm flow chart 108
Figure 5.50: Trellis diagram for error free decoding 109
Figure 5.51: Trellis diagram for error decoding...110
Figure 5.52: Example of 8 state Trellis diagram...111
Figure 5.53-AcsSegment timing digram 112
Figure 5.54: BMU Unit block diagram 112
Figure 5.55-Branch ID Values generation 113
Figure 5.56-Branch metric timing diagram...114
Figure 5.57-ACSU timing diagram...114
Figure 5.58-Metric Memory Unit timing diagram 115
Figure 6.1 : First LTE frame structure. 118
Figure 6.2 : Resource elements in LTE...119
Figure 6.3 : Refrence signals positions in LTE frame. 120
Figure 6.4 : Second LTE frame structure..120
Figure 6.5 : LTE frame structure, TDD , Type-2...121
Figure 6.6 : Full LTE transmitter block diagram...121
Figure 6.7 : Simplified LTE transmitter block diagram...122
Figure 6.8: Segmentation State Diagram 126
Figure 6.9. Structure of rate 1/3 turbo encoder 128
Figure 6.10:Schematic of turbo encoder 129

Figure 6.11. Turbo code internal interleaver parameters 130
Figure 6.12. Internal block diagram 131
Figure 6.13: Rate matching for turbo coded transport channels. 132
Figure 6.14: Interleaver block interface. 134
Figure 6.15:Bit collection block interface. 136
Figure 6.16: Bit selection block interface. 138
Figure 6.17: Rate matching block diagram. 139
Figure 6.18:Rate matching block interface 140
Figure 20: Ceil divider simulation. 141
Figure 21: Floor divider simulation 141
Figure 6.19: LTE mapper interface 143
Figure 6.20: Detailed block diagram of LTE mapper module 144
Figure 6.21: Transmitter block diagram [4] 144
Figure 6.22: top_ofdm_wifi interface 146
Figure 6.23: DFT interface [6] 147
Figure 6.24: Encoding of size parameter [6] 147
Figure 7.1-functional verification procedure 150
Figure 7.2 : testbenches generated files 150
Figure 7.3-compariosn file 151
Figure 7.4: IFFT testing framework process 152
Figure 7.5: MATLAB testbench output for a WIFI chain 153
Figure 7.6 : Results of 16-QAM integer part fixation..154
Figure 7.7 : Results of 16-QAM fraction part fixation. 154
Figure 8.1: FPGA internal structure 156
Figure 8.2: Softcore and Hardcore processor 157
Figure 8.3: Xilinx FPGA 158
Figure 8.4: CLB routing matrix in Virtex-5 159
Figure 8.5: MicroBlaze block diagram 160
Figure 8.6: Embedded system on chip 161
Figure 8.7: Embedded system on chip detailed block diagram 162
Figure 8.8: Peripheral template for partitions 162
Figure 8.9: Peripheral slave registers 163
Figure 8.10: Partial dynamic reconfiguration flow 164
Figure 8.11: ISE create project 164

Figure 8.12: ISE kit specs 165
Figure 8.13: ISE add source 165
Figure 8.14: ISE synthesis 166
Figure 8.15: Synthesis properties 166
Figure 8.16: Run Synthesis 167
Figure 8.17: Resulting Netlist file 167
Figure 8.18: XPS main window 168
Figure 8.19: XPS new project wizard 169
Figure 8.20: Base system builder wizard 169
Figure 8.21: BSB board choice 170
Figure 8.22: BSB single processor system 170
Figure 8.23: BSB Micro-Blaze specs 171
Figure 8.24: BSB peripheral wizard 171
Figure 8.25: BSB UART specs 172
Figure 8.26: finish BSB wizard 172
Figure 8.27: Working Space for XPS 173
Figure 8.28: XPS hardware menu 173
Figure 8.29: Create or Import peripheral wizard 174
Figure 8.30: Create peripheral wizard save location 174
Figure 8.31: Peripheral name 175
Figure 8.32: Bus choice 175
Figure 8.33: IP interface 176
Figure 8.34: Slave reg choice 176
Figure 8.35: XPS rescan 177
Figure 8.36: IP catalog 177
Figure 8.37: Peripheral Bus Connection 178
Figure 8.38: Peripheral Address initiation 178
Figure 8.39: Console display for missing port connection error 178
Figure 8.40: Connecting the ICAP_Clk port to the kit clock generator 179
Figure 8.41: Generating Netlist File 179
Figure 8.42: Exporting the XPS file to the SDK for microprocessor configuration 180
Figure 8.43: Bitstream and BMM generation 180
Figure 8.44: SDK Launcher 180

Figure 8.45: creating board support package 181
Figure 8.46: Defining Project name 181
Figure 8.47: Board Support Package Supported Libraries 182
Figure 8.48: Configuration for library xilfatfs 182
Figure 8.49: Creating Application Project 182
Figure 8.50: Defining Project name 183
Figure 8.51: Import Recources 184
Figure 8.52: Generating Linker Script 184
Figure 8.53: Enable Partial Reconfiguration 185
Figure 8.54: Selecting netlist files 185
Figure 8.55: Selecting Constrain file 186
Figure 8.56: Netlist Analysis 186
Figure 8.57: Set Reconfigurable Partition 187
Figure 8.58: Reconfigurable Partition Initiation 187
Figure 8.59: Adding Black Box Module 187
Figure 8.60: Adding CRC_4G Module 188
Figure 8.61: Adding Netlist file 188
Figure 8.62: Set Pblock Size 189
Figure 8.63: Floor Planning 190
Figure 8.64: Design Rule Checking 190
Figure 8.65: Define Rules to Check 191
Figure 8.66: Creating new Strategy 192
Figure 8.67: Selecting BMM File 192
Figure 8.68: Defining Configuration 193
Figure 8.69: Selecting Strategy for Configuration 193
Figure 8.70: Selecting Partial Modules 194
Figure 8.71: Activation Reconfigurable Module 194
Figure 8.72: Launch Runs 194
Figure 8.73: Create new Runs 195
Figure 8.74: Configuration Run 195
Figure 8.75: Specify Partition 196
Figure 8.76: Verify Configuration 196
Figure 8.77: Generating BitStream 197
Figure 8.78: Hardware implementation block diagram 198

Figure 9.1: flow charts and slave registers structure....202
Figure 9.2: writing processor data into slave registers....203
Figure 9.3: reading data from slave registers to processor....203
Figure 9.4: Slave registers array code....203
Figure 9.5: Slave registers array processor read....204
Figure 9.6: encoder instantiation and library definition....204
Figure 35: User defined functions diagram....205
Figure 36: Header of Sysace_read function....207
Figure 37: Header of bus2rp function....208
Figure 38: Header of rp2bus function....208
Figure 39: Header of Sysace_write function....208
Figure 10.1-multiple RPs-system block diagram....210
Figure 10.2-Standards Blocks....210
Figure 10.3-multiple RPs-Demo project....211
Figure 10.4-multiple RPs -user logic files....212
Figure 10.5-multiple RPs- math files....212
Figure 10.6-multiple RPs-mpd files....212
Figure 10.7-multiple RPs-ports connection....213
Figure 10.8-multiple RPs-graphical design view....213
Figure 10.9-multiple RPs-Plan Ahead partitions....213
Figure 10.10: multiple RPs-Plan Ahead configurations....214
Figure 10.11-multiple RPs-generated outputs....214
Figure 12.1: Block Diagram for MIMO....218
Figure 12.2: OFDMA Subcarriers Signals....218

List of Symbols and Abbreviations

16QAM	16Quadrature Amplitude Modulation.
2G	Second Mobile Generation.
3G	Third Mobile Generation.
3GPP	Third Generation Partnership Project.
64 QAM	64Quadrature Amplitude Modulation.
BER	Bit Error Rate.
BPSK	Binary phase shift keying.
CDMA	Code division multiple access.
CRC	Cyclic Redundancy Check.
DCH	Dedicated Channel.
DL	Downlink.
DPCCH	Dedicated Physical Control Channel.
DPCH	Dedicated Physical Channel.
DPDCH	Dedicated Physical Data Channel.
DS-CDMA	Direct-Sequence Code Division Multiple Access.
DTX	Discontinuous Transmission.
EDK	Embedded Development Kit.
FEC	Forward Error Correction.
FPGA	Field Programming Gate Array.
FDD	Frequency Division Duplex.
GSM	Global system for mobile communications.
ICAP	Internal Configuration Access Port.
ISE	Integrated Synthesis Environment.
JTAG	Joint Test Action Group.
MAC	Medium Access Control.
MIMO	Multiple Input Multiple Output.
OFDM	Orthogonal Frequency Division Multiplexing.
OVSF	Orthogonal Variable Spreading Factor.
PBS	Partial Bit Stream.
PLB	Programmable Logic Block.
PhCH	Physical Channel.
PRACH	Physical Random Access Channel.

RP	Reconfigurable Partition.
RM	Reconfigurable Module.
RX	Receiver.
SDK	Software Development Kit.
SF	Spreading Factor
SFN	System Frame Number.
SIR	Signal-to-Interference Ratio.
SNR	Signal to Noise Ratio.
TFCI	Transport Format Combination Indicator.
TrCH	Transport Channel.
TTI	Transmission Time Interval.
TX	Transmitter.
UE	User equipment.
UTRA	UMTS terrestrial radio access.
XPS	Xilinx Platform Studio.
XST	Xilinx Synthesis Technology.

Acknowledgment

This dissertation does not only hold the results of the year work, but also reflects the relationships with many generous and stimulating people. This is the time where we have the opportunity to present our appreciation to all of them.

First, to our advisors, Dr. Hassan Mostafa and Dr. Yasmine Fahmy, we would like to express our sincere gratefulness and appreciation for their excellent guidance, caring, patience, and immense help in planning and executing the work in a timely manner. Their great personality and creativity provided us with an excellent atmosphere for work, while their technical insight and experience helped us a lot in our research. Their support at the time of crisis will always be remembered.

Of course, we will never find words enough to express the gratitude and appreciation that we owe to our families. Their tender love and support have always been the cementing force for building what we achieved. The all-round support rendered by them provided the much needed stimulant to sail through the phases of stress and strain.

We would like also to thank Eng. Ahmed Sadek, who was always there for any research questions, providing precious advice and sharing his time and experience.

Special thanks and words of appreciation are due to our friends from BUE, Hamza, Mostafa and Osama. They made our assimilation of the project idea in the research station a lot smoother than we thought it would be. They supported us a lot and we learned much from our numerous technical discussions. They provided us with the emotional support we much needed at first.

Finally, Thanks are due to previous GP members. Their theses were supportive, informative and guided us in many critical issues.

Abstract

This thesis discusses the implementation of chains for multi-standards communication (3G, LTE, and WIFI) on a dynamically and partially reconfigurable heterogeneous platform FPGA VIRTEX5. Implementation results highlight the benefit of considering an FPGA platform like (VIRTEX 5) that supports efficiently intensive computation components. The implementation of the desired chains for multi-standards communication proves the availability of Partial Dynamic Reconfiguration technology to support efficiently Software Defined Radio.

This project aims to implement the transmitter and receiver chains for the three standards (Wi-Fi, 3G and LTE). Then reconfigure the FPGA by the desired chain on the fly without the need for resetting. This technique depends on the new technology **Partial Dynamic Reconfiguration (PDR)** which is introduced by XILINX. The new technology is expected to save area, power and cost of communication devices and increases the speed of switching and reconfiguring the FPGA. During the project, experience is gained in HDL & MATLAB modelling of the transmitter and receiver blocks of the three standards, building a system on chip that consists of: Micro-blaze processor IP, ICAP IP and system Ace IP to enable partial configuration and other peripherals. Thus enable the communication with PC while testing the reconfiguration on separate blocks and finally testing the reconfiguration of the entire standards chains.

Motivation

Communication system design is typically a highly complex process. The telecommunications industry is technologically dynamic, with new technologies and enhancement of existing technologies constantly evolving. The implementation of Radio Communication systems could have two ways: Hardware specific design which has the advantage of minimum usage of resources and best performance and software defined radio where the hardware components are implemented as software on PC or an embedded system which has the advantage of being flexible to any changes or updates.

Reconfigurable hardware platforms like FPGAs opened a way to have a solution that combines advantages of both ways by using hardware specific design that could be reconfigured or programmed by software control.

The idea of our project aims to even improve this solution by using the partial dynamic reconfiguration technology.

Partial dynamic reconfiguration allows us to divide the FPGA to partitions that could be reconfigured on the fly without need to reset the whole FPGA. Complete separate chains could be reduced to a few partitions that could be reconfigured to fit the required communication standard.

Chapter 1: Introduction

In this thesis we are going to prove the concept of the availability for using partial dynamic reconfiguration in implementing software defined radio. The thesis flow will go as following.

Chapter 2 introduces a survey about the technology then in Chapter 3 a brief summary of software defined radio followed by a partial dynamic reconfiguration concept. Then an overview of FPGA construction is covered illustrating targeted applications and advantages. Coexistence with communication standards discussed as viewed in other patents and papers as a step towards implementing the standards chains.

Throughout Chapter 4, 5 and 6 we will go deep in the implemented standards' transmitter and receiver (3G, LTE and Wi-Fi). Those chapters introduce the architectures of the standards chains (Receiver and transmitter), implementation of the HDL codes, illustrating the challenges, mentioning each block implementation and any modification done. Each implementation is done giving the all data rates combinations, where complete system design is performed, obtaining different blocks' specifications and expected non-idealities.

Chapter 7 covers functional verification of the chain blocks using MATLAB codes and Perl scripts, IFFT test and fixed point simulation.

Chapter 8 discusses system on chip concepts, each chain implementation using one RP, switching between different chains and kit steps implementing Microblaze bus.

Chapter 9 covers the FPGA testing environment, I/O files and methods used to interface with PC.

Chapter 10 introduces new approach that's deal with reconfigurable partition rather than one partition and discusses the testing environment for this new technique.

Chapter 11 concludes our achievements and results through a year full of team work, enthusiasm, hard work and research.

Chapter 12 proposes the potentials expected in the upcoming years and improvements that are the next step for the projects fellows.

Chapter 2: Survey

Software-Defined Radio (SDR) is a rapidly evolving technology that is receiving enormous recognition and generating widespread interest in the telecommunication industry. Over the last few years, analog radio systems are being replaced by digital radio systems for various radio applications in military, civilian and commercial spaces. In addition to this, programmable hardware modules are increasingly being used in digital radio systems at different functional levels. SDR technology aims to take advantage of these programmable hardware modules to build open-architecture based radio system software.

SDR technology facilitates implementation of some of the functional modules in a radio system. This helps in building reconfigurable software radio systems where dynamic selection of parameters for each of the above-mentioned functional modules is possible. A complete hardware based radio system has limited utility since parameters for each of the functional modules are fixed. A radio system built using SDR technology extends the utility of the system for a wide range of applications that use different techniques.

Commercial wireless communication industry is currently facing problems due to constant evolution of protocol standards (2G, 3G, and 4G), existence of incompatible wireless network technologies in different countries inhibiting deployment of problems in rolling-out new services and features due to wide-spread presence of legacy subscriber handsets.

SDR technology promises to solve these problems by implementing the radio functionality as software modules running on a generic hardware platform. Further, multiple software modules implementing different standards can be present in the radio system. The system can take up different personalities depending on the software module being used. Also, the software modules that implement new services and features can be downloaded over-the-air onto the handsets. This kind of flexibility offered by SDR systems helps in dealing with problems due to differing standards and issues related to deployment of new services and features.

Current market drivers such as future-proof equipment, seamless integration of new services, multi-mode equipment and over-the-air feature insertion in commercial wireless networking industry have resulted in widespread interest in SDR technology. The technology can be used to implement wireless network infrastructure equipment as well as wireless handsets, wireless modems and other end user devices. However, factors like higher power consumption, increased complexity of software and possibly higher initial cost of equipment regarding to the benefits offered by the technology should be carefully considered before using SDR technology to build a radio system.

Chapter 3: SDR & PDR Concepts

3.1 SDR by Definition

Historically, radios have been designed to process a specific waveform. Single-function, application-specific radios that operate in a known, fixed environment are easy to optimize for performance, size, and power consumption. At first glance most radios appear to be single function a first-generation cellular phone sends your voice, while a WiFi base station connects you to the Internet. Upon closer inspection, both of these devices are actually quite flexible and support different waveforms. Clearly a software-defined radio's main characteristic is its ability to support different waveforms. The definition from wireless innovation forum (formerly SDR forum) states: A software-defined radio is a radio in which some or all of the physical layer functions are software defined. [1]

3.2 Why SDR?

It takes time for a new technology to evolve from the lab to the field. Since SDR is relatively new, it is not yet clear where it can be applied. Some of the most significant advantages and applications are summarized below.

- Interoperability. An SDR can seamlessly communicate with multiple incompatible radios or act as a bridge between them. Interoperability was a primary reason for the US military's interest in, and funding of, SDR for the past 30 years. Different branches of the military and law enforcement use dozens of incompatible radios, hindering communication during joint operations. A single multi-channel and multi-standard SDR can act as a translator for all the different radios.

- Efficient use of resources under varying conditions. An SDR can adapt the waveform to maximize a key metric. For example, a low-power waveform can be selected if the radio is running low on battery. A high-throughput waveform can be selected to quickly download a file. By choosing the appropriate waveform for every scenario, the radios can provide a better user experience.

- Opportunistic frequency reuse (cognitive radio) An SDR can take advantage of underutilized spectrum. If the owner of the spectrum is not using it, an SDR can ‘borrow’ the spectrum until the owner comes back. This technique has the potential to dramatically increase the amount of available spectrum.

- Reduced obsolescence (future-proofing). An SDR can be upgraded in the field to support the latest communications standards. This capability is especially important to radios with long life cycles such as those in military and aerospace applications. For example, a new cellular standard can be rolled out by remotely loading new software into an SDR base station, saving the cost of new hardware and the installation labor.

- Lower cost. An SDR can be adapted for use in multiple markets and for multiple applications. Economies of scale come into play to reduce the cost of each device. For example, the same radio can be sold to cell phone and automobile manufacturers. Just as significantly, the cost of maintenance and training is reduced.

- Research and development. An SDR can be used to implement many different waveforms for real-time performance analysis. Large trade-space studies can be conducted much faster (and often with higher fidelity) than through simulations.

3.2.1 Field Programmable Gate Arrays

A Field Programmable Gate Arrays (FPGA) is a pre-manufactured silicon device with high flexibility and capability to be configured to realize different applications developed by a designer. They are programmed using Hardware Description Language (HDL) like VHDL or Verilog. So, it is naturally different from an Application Specific Integrated Circuit (ASIC), which is a circuit designed for a specific application with no reconfiguration capabilities.

An ASIC not only lacks the configurability feature, but also requires a long design cycle, and high start-up engineering cost compared to an FPGA. On the other side, FPGAs trade the extra area, power consumption, and delay for its unique feature. “Typically, FPGAs occupy larger area and dissipate more switching power than ASIC standard cells by factors of 20-30x and 10x, respectively.

From marketing prospective, FPGAs are used for small volume products need to be sold faster, where ASICs are used for large volume products, but the non-recurring engineering cost in the ASICs make their cost as a function of production volume in a flatter way than the FPGA. As shown in **Error! Reference source not found.**, the more the technology advances in the scaling factor, the wider the range of using FPGAs in production.

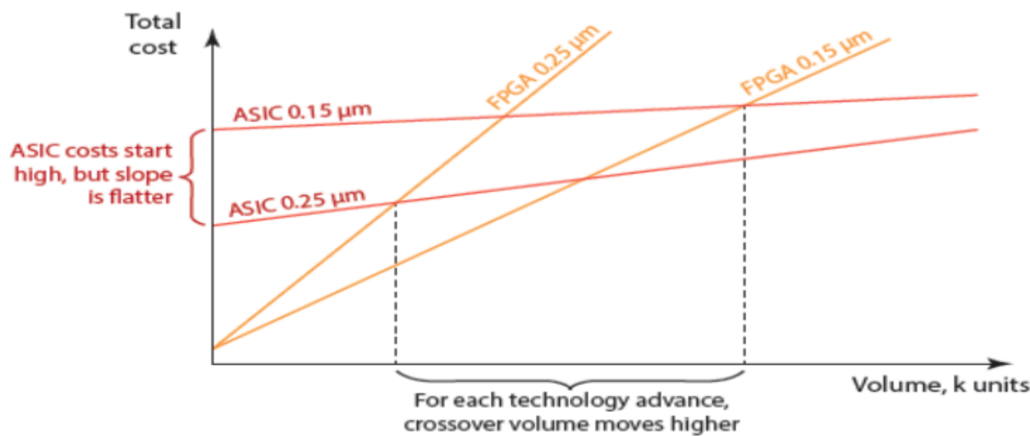


Figure 3.1: FPGA and ASIC total cost V.s production volume

3.2.2 What is inside the FPGA?

1. **Configurable Logic Blocks (CLBs):** includes Registers and Look-Up Tables (LUTs). They are the building blocks of the FPGA. Each Xilinx Virtex 5 device contains arrays of CLBs, each Virtex 5 CLB has two slices, and each slice has four LUTs and four Flip Flops. Combinatorial logic is implemented using LUTs, they can implement any 6-input combinatorial function of the user choice, with a cost of delay. Noting that, complexity of combinatorial function does not matter as long as it depends on six inputs or less. Flip Flops can be programmed to be latch, SR, JK, or D Flip Flops. Also, one carry chain is available per slice for arithmetic purpose; it helps to secure fast propagation of carry bit to nearby cells, which means it improves the speed. Moreover, it saves LUTs.
2. **Dedicated Blocks:** Like DSPs, which acts as an arithmetic logic unit, RAM blocks, PCIe core.

3. **Input/output Blocks:** with programmable standard functionality, like LVCMOS, LVPECL, and PCI. In fact, each bank can support several standards as long as they share the same reference voltage, or output voltage.
4. **Routing:** a combination of programmable and dedicated routing lines, use switching matrices connect lines from any source to any destination. Constrains can be applied.
5. **Clocking Resources:** like Phase Locked Loop (PLL) which removes clock errors, and Digital Clock Management (DCM). The dedicated clock trees balance the skew and minimize the delay. Thirty-two separate clock networks are available in Virtex 5 FPGA. As shown in Figure 3.2.

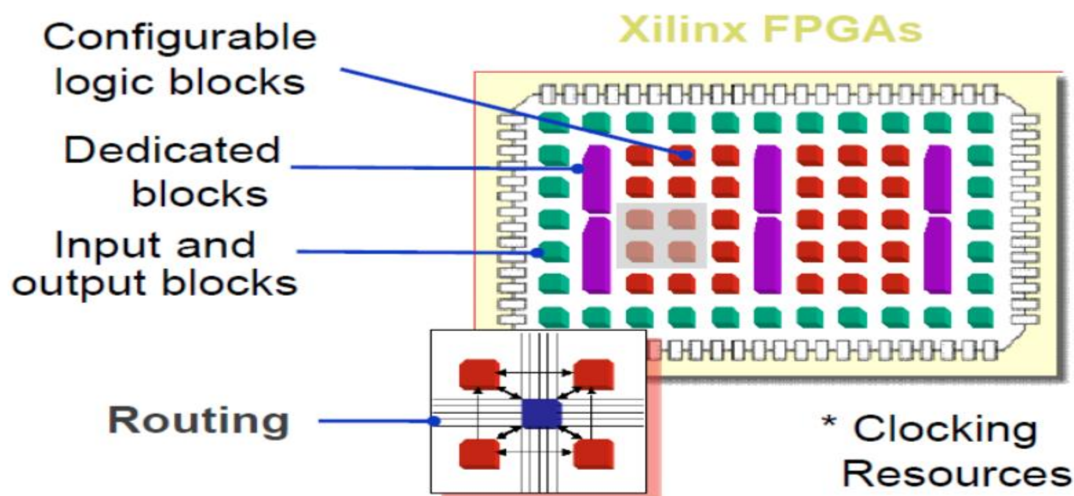


Figure 3.2:FPGA Internal structure

3.3 FPGA Configuration

3.3.1 Configuration Definition

Using a preliminary definition, a configuration is a complete FPGA design. That means, everything on the chip is specified either to do a function, or nothing at all. One can view the FPGA is a two-layered device, consists of a configuration memory layer, and a logic layer Figure 3.3. The configuration, or the complete design, stored on the configuration memory layer, will control the logic on the other layer.

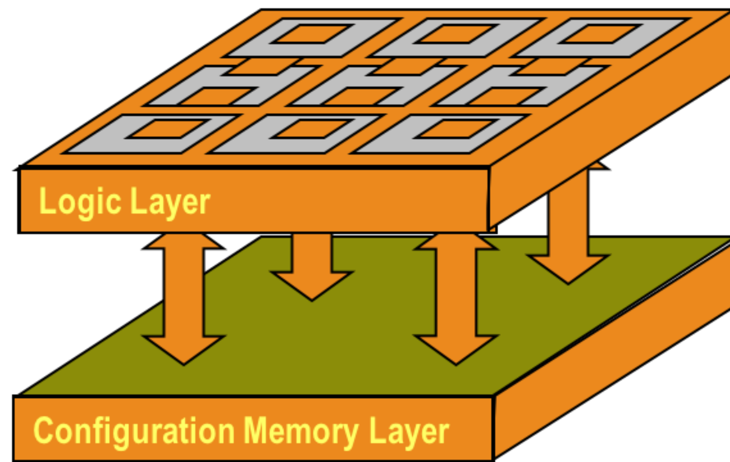


Figure 3.3:FPGA Layers

3.3.2 Types of Configuration

There are four types of configuration of FPGAs shown in Figure 3.4

1. Fixed Configuration: where data is loaded from a memory at power-on, then the configuration will remain fixed until the end of the FPGA cycle. This type lacks efficiency, since all the possible functions needed to be done by the FPGA must be specified in the configuration file from the beginning. On the other side, the space and resources of the FPGA are limited. That adds complexity to the design.

2. Reconfiguration: An initial full bit file contains a complete configuration is loaded into the device at power-on. Then different full bit files with other complete configurations can be loaded on the FPGA during its duty cycle, but the configuration memory must be erased first. This is a good step but it is not enough, as seen in figure 4, there is large overhead time in the reconfiguration phase.

3. **Partial Reconfiguration:** Initial full bit file with a complete configuration is loaded into the device at power-on. Whenever something to be altered, all computations will stop, then a partial bit file concerned with the modification in the original complete configuration is loaded. This time the reconfiguration overhead time is reduced compared the previous type. In applications where FPGAs are used as communication hub, they must be active all the time to retain active links, so partial reconfiguration is not enough, as the computations stop during loading the partial bit file.

4. **Dynamic Partial Reconfiguration (DPR):** Unlike the partial reconfiguration, while the configuration layer on the FPGA is being modified, the logical layer continues its normal operation, except for the circuit subjected to modification. This reconfiguration overhead is limited to the circuit.

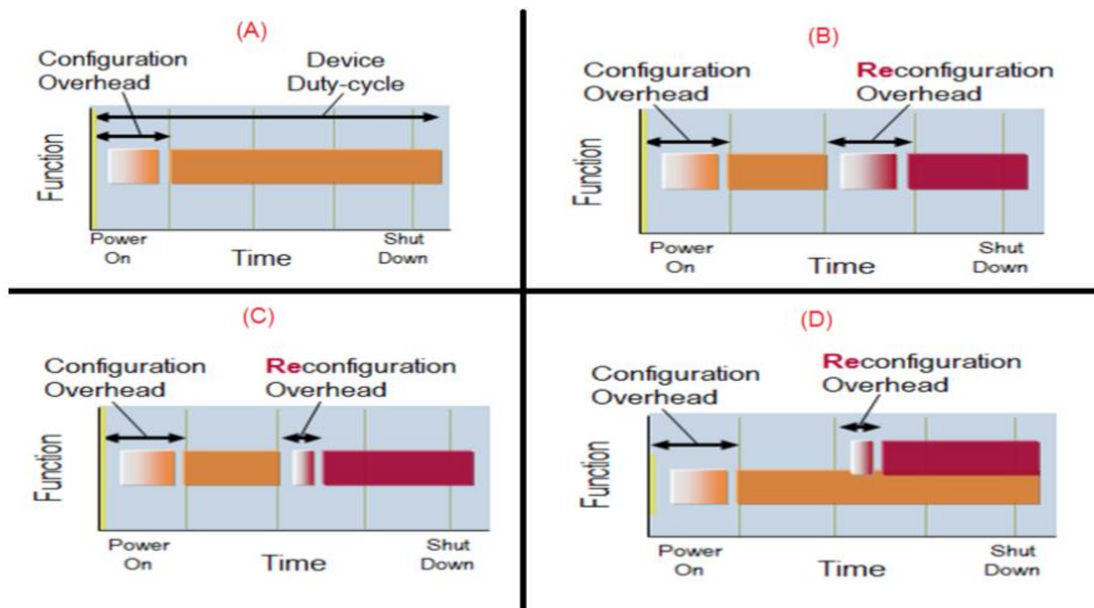


Figure 3.4: FPGA configuration types

3.3.3 What is Partial Reconfiguration (Problems to solve)?

- Applications need to be able handle a wide variety of functions.
  - Supporting many at once can use a great deal of space.
- FPGA and board space is limited.
  - Multi-chip solutions require extra area, cost and power.
- FPGA can be a communications hub, must remain active.
  - Cannot reconfigure due to established links.
- PCIe enumeration time is increasingly difficult to meet with larger devices.
  - Challenge increases significantly in Virtex-5 and Virtex-6.

3.4 Partial Reconfiguration of FPGAs

As systems become more complex and designers are asked to do more with less, FPGA adaptability has become a critical asset. While FPGAs have always provided the flexibility to do on-site device reprogramming, today’s tougher cost, board space, and power consumption constraints demand even more efficient design strategies.

Partial reconfiguration extends the inherent flexibility of the FPGA by allowing specific regions of the FPGA to be reprogrammed with new functionality while applications continue to run in the remainder of the device Figure 3.5. Partial reconfiguration addresses three fundamental needs by enabling the designer to: Reduce cost and/or board space, change a design in the field, Reduce power consumption. [2]

The two most prevalent user problems addressed by partial reconfiguration are: Fitting more logic into an existing device and fitting a design into a smaller, less expensive device. Historically, designers have spent a lot, trying new implementation switches, reworking code, and re-engineering solutions to squeeze them into the smallest possible FPGA. Partial reconfiguration enables these designers to reduce the size of their designs by dynamically time-multiplexing portions of the available hardware resources. The ability to load functions on an as-needed basis also reduces the amount of idle logic, thereby saving additional space. One of the applications of this strategy is the use of partial reconfiguration within a software defined radio (SDR) system, where the user uploads a new waveform on demand to establish communication with a new channel. Any number of waveforms can be supported by a single hardware platform, requiring only unique partial bit streams to be available for these waveforms. Established links to other channels are not disrupted by the update to another channel due to the on-the-fly characteristics of partial reconfiguration.

3.4.1 Partial Reconfiguration Enables

- System Flexibility
- Perform more functions while maintaining communication links
- Size and Cost Reduction
- Time-multiplex the hardware to require a smaller FPGA
- Power Reduction
- Shut down power-hungry tasks when not needed

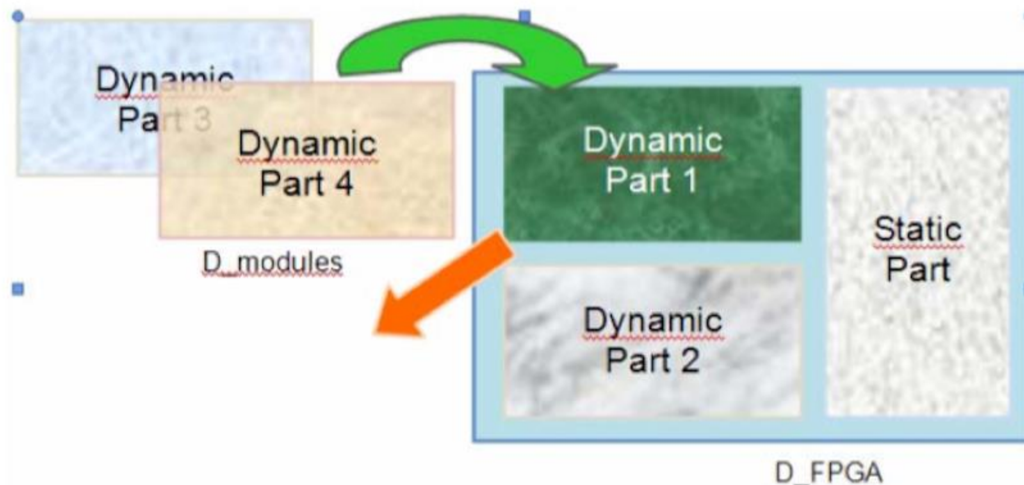


Figure 3.5: Reconfigurable FPGA Structure

3.4.2 Brief History of Partial Reconfiguration

- The ability to partially reconfigure Xilinx FPGAs has been evolving for years
  – JBits and the XC6200 family are early examples
  – These solutions were experimental and very restrictive
  – They were used by academics
- The Difference Design approach
  – Quiet cumbersome
- The Modular Design approach appeared in 2005
  – Flow was complex and the silicon still had significant limitations
  – The PR Lounge, using 9.2.04i software, phased out in 2QCY10
- More robust, mainstream solution was needed
  – Release 11 was a new dawn in the support of Partial Reconfiguration
  – Tools leverage mature technology (Partitions, PlanAhead)
  – Flow is more user-friendly, less complex
- Software has been limiting factor preventing wide adoption
  – Early efforts were difficult and labor intensive
  – Modular Design provided first real flow, but was convoluted
  – Software was not included in mainstream tools
  – Partitions now permit a more mainstream approach
- However, new flow is not “push-button”
  – User needs to follow specific rules and requirements in design and flow
  – Restrictions will reduce performance and efficiency

3.4.3 Partial Reconfiguration Analogy

A FPGAs PR region is similar to the stack in a microprocessor. The FPGA can have many PR regions so it can be much more powerful than a microprocessor for applications switching Figure 3.6**Error! Reference source not found.**.

3.4.4 System Flexibility: Communication Hub

The FPGA can be a communications hub and must remain active. It cannot perform full reconfiguration due to established links Figure 3.7.

3.4.5 Size and Cost Reduction: Time Multiplexing

Applications need to be able handle a variety of functions; supporting many at once can use a great deal of space. The library of functions use case covers a wide number of applications. The time-based multiplexing of functions reduces device size requirement **Error! Reference source not found.**.

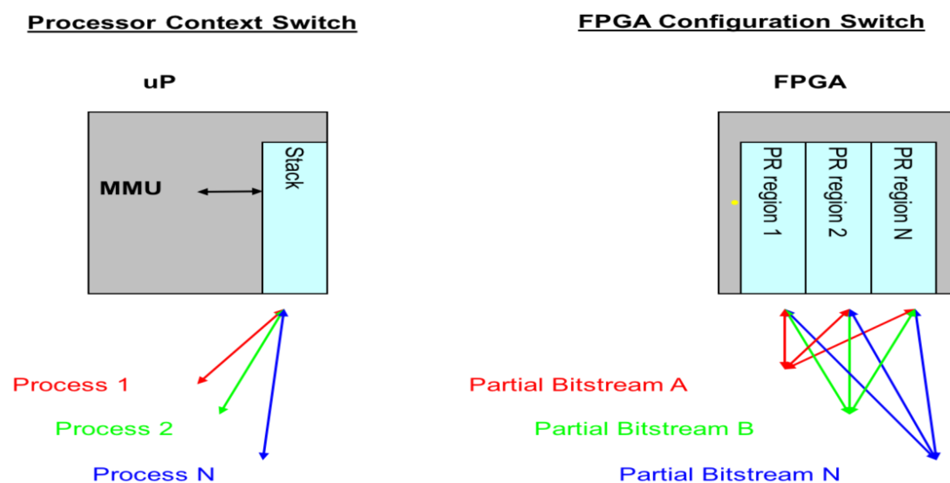


Figure 3.6: PR analogy

3.4.6 Power Reduction Techniques with PR

- **Board space and resources are limited**
  - Multi-chip solutions consume extra area, cost, and power

- **Many techniques can be employed to reduce power**
  - Swap out high-power functions for low-power functions when maximum performance is not required
  - Swap out black boxes for inactive regions
  - Swap high-power I/O standards for lower-power I/O when specific characteristics are not needed
  - Time-multiplexing functions will reduce power by reducing amount of configured logic

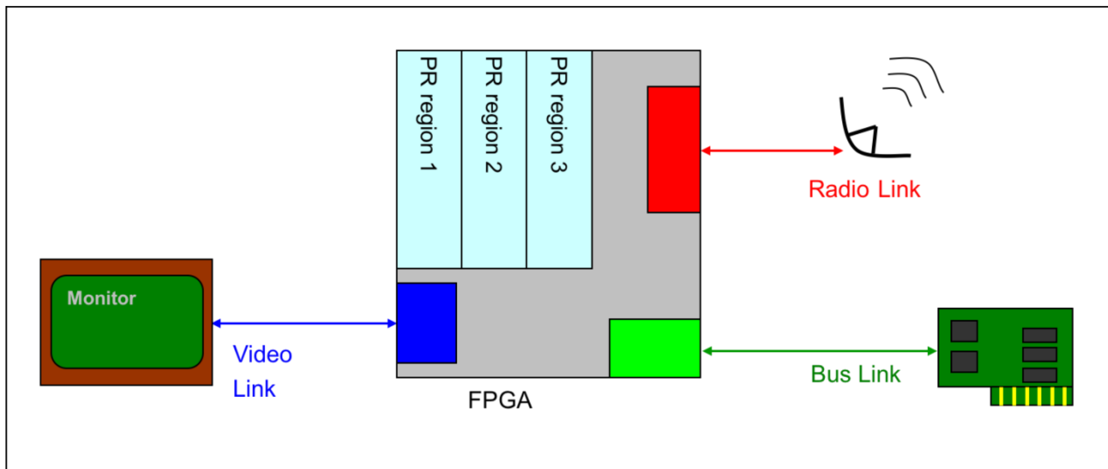


Figure 3.7: FPGA as communication Hub

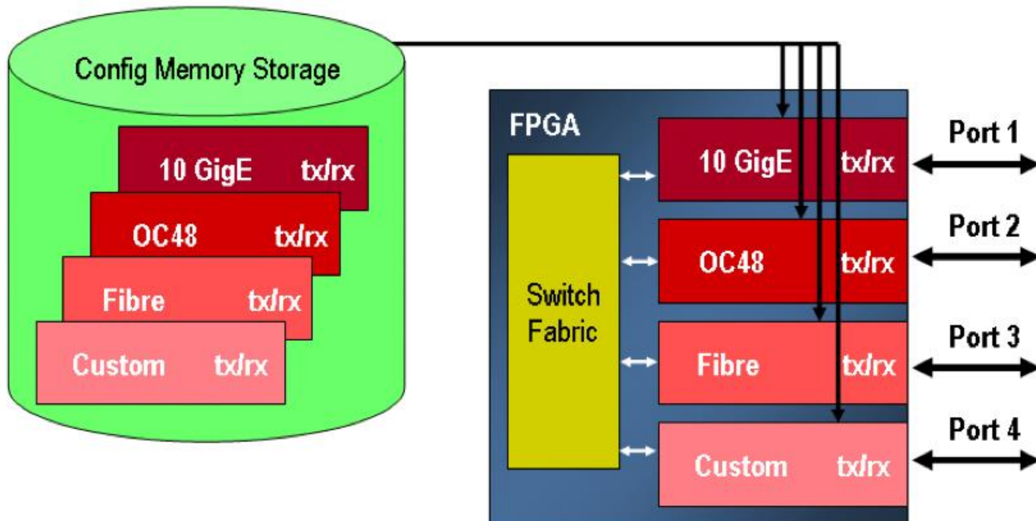


Figure 3.8: FPGA time multiplexing

Breaking the logic layer (Partial/Static Regions, Reconfigurable Partitions/Modules)

Having a partial reconfiguration design means that, there are some parts of the design that will be changed according to its various configurations during run-time, and other parts that will not be changed, either due to design, or because it cannot be reconfigured. The first type, reconfigurable areas, are named **partial region**. The other type that will not be reconfigured is named the **static region**. Nearly everything on an FPGA can be reconfigured: LUTs, Flip Flops, block RAMs, distributed RAMs, shift registers, DSP blocks, and IO components. But there are some parts that cannot be reconfigured at all, like: clock modifying blocks, global clock buffers, device feature blocks like ICAP and STARTUP.

A partial region defined by the user to make its logic reconfigurable is also called **reconfigurable partition**, on the other hand, in a single design/configuration, the portion of the design occupies this reconfigurable partition is named **reconfigurable module**. Each reconfigurable partition may have multiple reconfiguration modules, as a reconfigurable module can be replaced in run-time by another reconfigurable module occupying the same reconfigurable partition.

As seen in figure 5, the grey block can be seen as static region, they will not experience any changes in run-time. On the other side, colored blocks are the reconfigurable partitions, each partition represents a LED with a certain color, but on the same partition of a specific color, red for example, multiple modules can be applied, each module represent blinking speed of its LED. So red is a reconfigurable partition, with three modules of three speeds can be applied in run-time.

3.4.7 Styles of Partial Regions

There are three different possible configuration styles Figure 3.9 in which partial regions can be arranged into:

1. **Island Style:** It is the easiest possible style. Using it, only one reconfigurable module can be hosted exclusively per island. A system can support more than one island within its logic.

a. **Single island style,** is the case when modules are tied to their specific planned islands.

b. **Multiple island style** is the case when modules are free to be relocated to different islands within the system.

This approach is suitable with a system having few modules are being swapped. On the other side it suffers from internal fragmentation, which is the result of having more than one module sharing the same partition. One of these modules will require more resources than the other, so the partition is initially planned to contain all those resources needed by the most complex module. When the simpler module is used, not all the resources will be used, and they cannot be shared outside this partition/island.

2. **Tiling (Slot Style and Grid Style):** A slot style is one dimensional tiling, while the grid is two dimensional tiling. Tiling a reconfigurable region allows multiple modules to be hosted simultaneously within it; each module will occupy the number of tiles based on the required resources to be used. So, this improves the internal fragmentation. On the other hand, it makes it more complex to communicate with reconfigurable modules. Also, a more complex type of fragmentation exists, the external fragmentation, because the partial region is not homogeneous, various elements exists in different places within the FPGA according to its design (like ROMs, DSPs), so tiling the region will consider this.

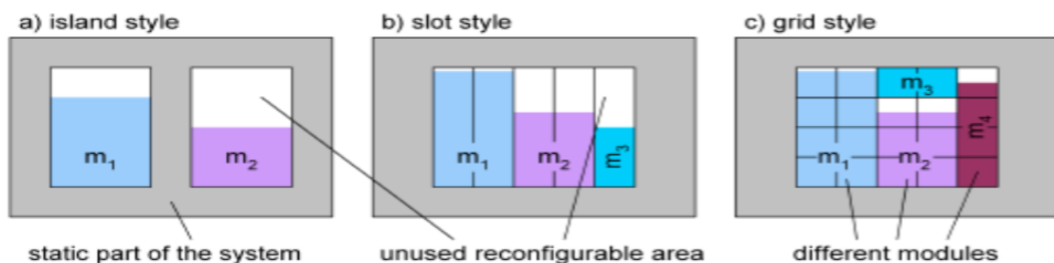


Figure 3.9: Partial regions styles

3.4.8 Full bit file V.s Partial bit file

A **full bit file** contains the data of a complete design/configuration. It contains all the necessary information to reset the FPGA device, configure it with a complete design, and verify a bit file is not corrupted.

A **partial bit file** contains a partial design configuration; it has no header, only the address of the target region and its corresponding partial data. The partial bit file may have many errors, like the address information, the data information.

There is no error detection built-in mechanism; a corrupt partial bit file can damage the FPGA if left in operation. So, system with high probability of its partial bit being corrupted, like those which are sent over radio, should implement a check circuit on the FPGA before loading the partial bit file received.

3.4.9 Types of Partial Reconfiguration

There are two mechanisms to be followed for performing partial reconfiguration, noting that the first one is used in PlanAhead:

1. **Modular Based:** All the components of the design are implemented separate from each other, and then the complete bit stream is the sum of all the partial bit streams of these components. This mechanism is suitable for large changes in the functionality of the structure of a design.
2. **Difference Based:** All the possible configurations of the design are specified, with one full bit file. The partial bit files are generated from the differences of two configurations of the design. It is suitable for small changes, giving a high switching speed between the versions of the design.

3.4.10 Benefits of Partial Reconfiguration

1. Reducing the size and the cost of the FPGA: By time multiplexing the hardware, more logic can be fit into the same area, hence bigger designs can be fit into smaller less expensive devices.
2. Reduce power consumption: Smaller and simpler designs consume less power.
   Moreover, developers can implement more than mode for the same design, so that the user will be able to choose between high performance/low power mode, and low performance/high power mode. Also, swapping between high power IO standards and low power IO standards will be possible.
3. Increase deployed system flexibility: Changing a design in the field becomes easier; the modified function is placed and routed in context with the already verified remainder of the design, then this partial file will be delivered to the system in field. These can be applied without shutting down system.

4. Improving FPGA fault tolerance.
5. Accelerating configurable computing.
6. A variety of applications are now possible using PR and PDR.

3.4.11 Requirements of PDR

To implement a run-time reconfigurable system, the following requirements must be satisfied:

1. Partial modules have to use exclusive resources, not shared with any other part of the system.
2. Routing has to be constrained to specific wires for crossing the module border for each signal bit of the module entity.
3. Activate all clock trees that are used by partial modules.
4. Constraint the timing.

The first requirement is called Area Group Constrain. In some cases, the different configuration modules, associated with a reconfigurable partition, use different resources.

The area group constraint of this reconfigurable partition must contain all the resources needed by all its reconfigurable modules. It is applied practically in the phase of floor planning in PlanAhead, or by writing directly into the constraint file. Activating all the clock trees, and constraining the timing form the Time Constrains, and they are related to the design also are done in PlanAhead by writing into the constraint file.

3.4.12 Partition Pins

Partition pins are the interfaces between the static and the reconfigurable logic. Unlike the other requirements that are done using the constraint file, there is no constraint to bind a signal in a top level design that is responsible for the communication with a partial module to a specific wire that crosses the partial-to-static border. To solve this, there are many macros have been developed using vendors like Xilinx in their tools Figure 3.10:

1. **Bus Macros:** Consists of a LUT in the static part and a LUT in the partial region, the signal then will be bounded to the internal macro wires. This costs two LUTs per signal to wire binding. Moreover, this has an effect of additional latency.
2. **Proxy Logic:** LUT acting like an anchor will be placed in the partial region. Then, differentially, the interfaces to the partial region at the static region will be routed to those anchors. Also the partial modules will be implemented incrementally from the routed static system and those anchors without modifying any static routing. This leads to preserving the initial static routing, hence satisfying binding the signal to certain wires. On the other hand, the routing will be different for each reconfigurable partition. Making it impossible to optimize island styles to be multiple island style.
3. **A new approach:** This approach Defines tunnels through the partial region by blocking other possible routes, so this drilled tunnel, is the only possible path for a signal. By this, the router is forced to bind a signal to this tunnel.

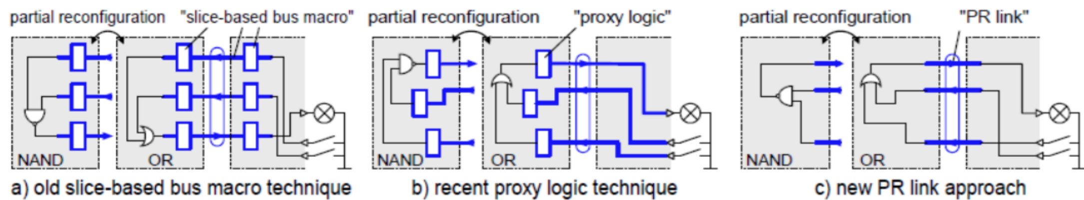


Figure 3.10: Partition pins

Chapter 4: 3G transmitter and receiver implementation

By the late 1990s, the very success of GSM (global system for mobile) was again raising questions about the future need for yet more spectrum. The GSM community was initially focused on developing GSM's circuit and packet switched data services. It is limited to maximum data rates of less than 50 kbps and neither can support video telephony. There was an obvious potential evolution towards a wider bandwidth CDMA system. The aim was to develop a radio system capable of supporting up to 2 Mbps data rates. The global WCDMA(wide code division multiple access) specification activities were combined into a third generation partnership project (3GPP) that aimed *to create the first set of specifications by the end of 1999, called Release 99.*

The early WCDMA networks offered some benefits for the end users including data rate up to 384 kbps in uplink and in downlink and simultaneous voice and data.

HSDPA (High Speed Downlink Packet Access) brought a few major changes to the radio networks: the architecture became flatter with packet scheduling and retransmissions. The peak bit rates increased from 0.384 Mbps initially to 1.8–3.6 Mbps and later to 7.2–14.4 Mbps.

Suddenly, wide area networks were able to offer data rates similar to low end ADSL (Asymmetric Digital Subscriber Line) and were also able to push the cost per bit down. So that offering hundreds of megabytes or even gigabytes of data per month became feasible. The high efficiency also allowed changes to the pricing model.

The HSPA (High Speed Packet Access) network efficiency has improved considerably especially with Ethernet-based transport and compact new base stations with simple installation, low power consumption and fast capacity expansion. HSPA evolution also includes a number of features that can enhance the spectral efficiency. Quality of Service (QoS) differentiation is utilized to control excessive network usage to keep users happy also during the busy hours. HSPA evolution includes features that cut down the power consumption considerably and also improve the efficiency of small packet transmission in the HSPA radio networks**.**

4.1 Frame structure

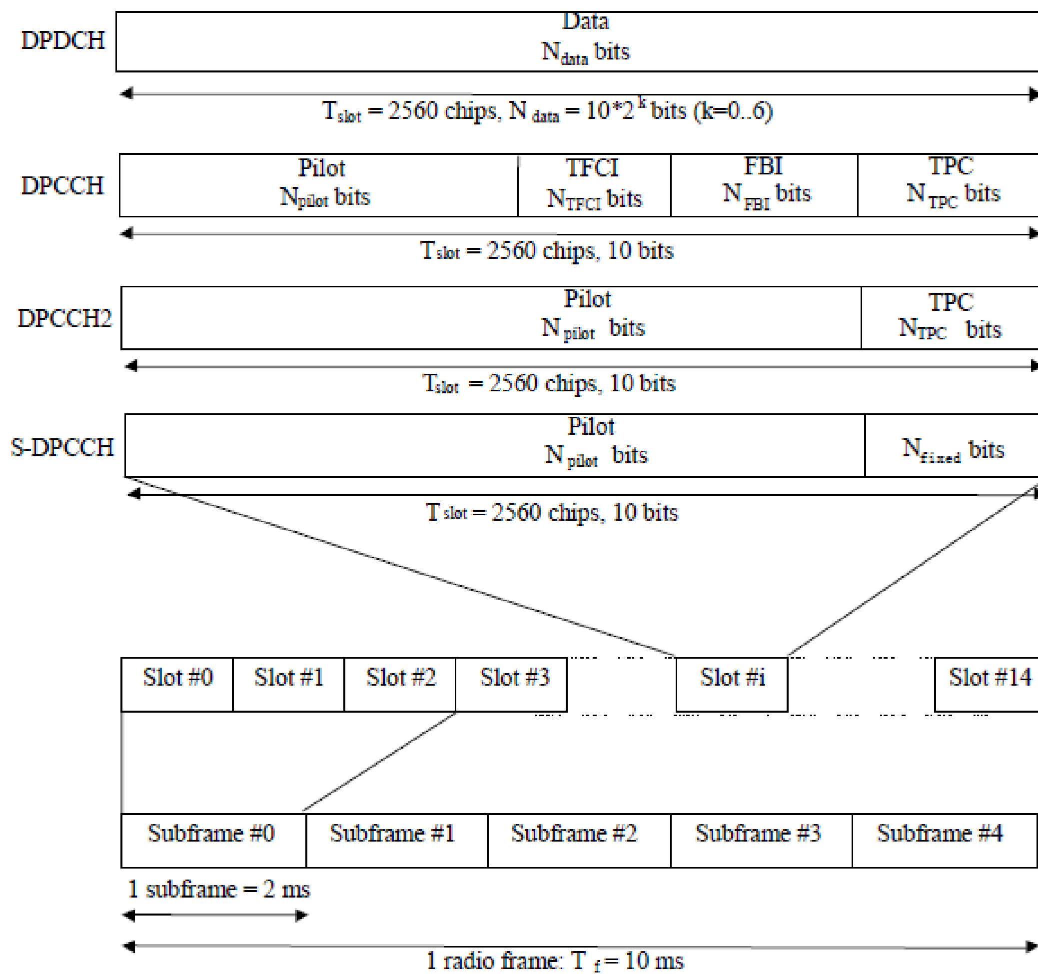


Figure 4.1: Frame structure

There are seven types of uplink dedicated physical channels, the uplink Dedicated Physical Data Channel (uplink DPDCH), the uplink Dedicated Physical Control Channel (uplink DPCCH), the uplink Secondary Dedicated Physical Control Channel (uplink S-DPCCH), the uplink Dedicated Physical Control Channel 2 (uplink DPCCH2), the uplink E-DCH Dedicated Physical Data Channel (uplink E-DPDCH), the uplink E-DCH Dedicated Physical Control Channel (uplink E-DPCCH) and the uplink Dedicated Control Channel associated with HS-DSCH transmission (uplink HS-DPCCH) [3].

Each frame consists of 5 sub-frames; each sub-frame consists of 3 slots so the frame consists of 15 slots. The Length of the frame corresponds to 38400 chips. Chip Rate is 3.84 Mcps so Frame Length is 10ms as shown in Figure 4.1.

The length of a Slot of DPDCH (Dedicated Physical Data Channel) corresponds to 2560 chips. $N_{data}(number\ of\ data\ bits\ in\ the\ slot) = 10 * 2^k\ bits$. The parameter k is related to the spreading factor SF where$SF = 256/2^K$. In uplink: SF range is from 256 down to 4 so k range is from 6 down to 0 so N_{data} range is from 10 to 640 bits.

Slot of DPCCH (Dedicated Physical Control Channel): fixed SF of 256 and contains 10 bits. DPCCH has four fields: Pilot, TFCI (transport-format combination indicator), FBI (feedback information), and TPC (transmit power-control), size of each field is not fixed and defined in the table shown in Figure 4.2 [4].

Slot Form at #i	Channel Bit Rate (kbps)	Channel Symbol Rate (ksps)	SF	Bits/ Frame	Bits/ Slot	N_{pilot}	N_{TPC}	N_{TFCI}	N_{FBI}	Transmitted slots per radio frame
0	15	15	256	150	10	6	2	2	0	15
0A	15	15	256	150	10	5	2	3	0	10-14
0B	15	15	256	150	10	4	2	4	0	8-9
1	15	15	256	150	10	8	2	0	0	8-15
2	15	15	256	150	10	5	2	2	1	15
2A	15	15	256	150	10	4	2	3	1	10-14
2B	15	15	256	150	10	3	2	4	1	8-9
3	15	15	256	150	10	7	2	0	1	8-15
4	15	15	256	150	10	6	4	0	0	8-15
5	15	15	256	150	10	6	2	2*	0	8-15

Figure 4.2: DPCCH Field

Pilot used for channel estimation and Frame Synchronization. Pilot Bit Patterns depend on number of pilot bits and slot number and are defined in this table shown in *Figure* 4.4

TFCI used for bit rate control, channel decoding, interleaving parameters for every DPDCH frame. FBI used for transmission diversity in the DL. TPC used for inner loop power control commands. TPC Bit Patterns are defined in Figure 4.3

TPC Bit Pattern			Transmitter power control command
$N_{TPC} = 2$	$N_{TPC} = 4$	$N_{TPC} = 8$	
11	1111	11111111	1
00	0000	00000000	0

Figure 4.3: Bit patterns of TPC

	$N_{pilot} = 3$			$N_{pilot} = 4$				$N_{pilot} = 5$					$N_{pilot} = 6$					
Bit #	0	1	2	0	1	2	3	0	1	2	3	4	0	1	2	3	4	5
Slot #0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0
1	0	0	1	1	0	0	1	0	0	1	1	0	1	0	0	1	1	0
2	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1
3	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
4	1	0	1	1	1	0	1	1	0	1	0	1	1	1	0	1	0	1
5	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0
6	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0	0
7	1	0	1	1	1	0	1	1	0	1	0	0	1	1	0	1	0	0
8	0	1	1	1	0	1	1	0	1	1	1	0	1	0	1	1	1	0
9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1
11	1	0	1	1	1	0	1	1	0	1	1	1	1	1	0	1	1	1
12	1	0	1	1	1	0	1	1	0	1	0	0	1	1	0	1	0	0
13	0	0	1	1	0	0	1	0	0	1	1	1	1	0	0	1	1	1
14	0	0	1	1	0	0	1	0	0	1	1	1	1	0	0	1	1	1

	$N_{pilot} = 7$							$N_{pilot} = 8$							
Bit #	0	1	2	3	4	5	6	0	1	2	3	4	5	6	7
Slot #0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0	1	1	0	1	0	1	1	1	0
2	1	0	1	1	0	1	1	1	0	1	1	1	0	1	1
3	1	0	0	1	0	0	1	1	0	1	0	1	0	1	0
4	1	1	0	1	0	1	1	1	1	1	0	1	0	1	1
5	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0
6	1	1	1	1	0	0	1	1	1	1	1	1	0	1	0
7	1	1	0	1	0	0	1	1	1	1	0	1	0	1	0
8	1	0	1	1	1	0	1	1	0	1	1	1	1	1	0
9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10	1	0	1	1	0	1	1	1	0	1	1	1	0	1	1
11	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1
12	1	1	0	1	0	0	1	1	1	1	0	1	0	1	0
13	1	0	0	1	1	1	1	1	0	1	0	1	1	1	1
14	1	0	0	1	1	1	1	1	0	1	0	1	1	1	1

Figure 4.4: Pilots for uplink DPCCH

4.2 3G Transmitter

Transmitter of 3G consists of several blocks as shown in Figure 4.5

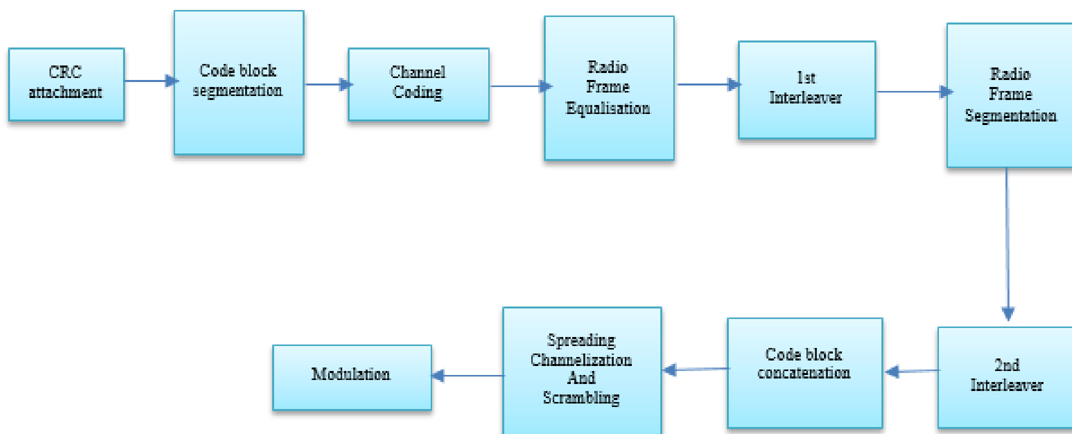


Figure 4.5: Transmitter blocks

4.2.1 CRC (Cyclic Redundancy Check) attachment

CRC process is provided on transport blocks for error detection in which the entire block is used to calculate the CRC parity bits for each transport block. Instead of adding just one bit to a block of data, several bits are added. The size of the CRC is 24, 16, 12, 8 or 0 bits and it is signaled from higher layers -depending on the channel- what CRC size that should be used [5].

CRCs are typically implemented in hardware as a linear feedback shift register as shown in Figure 4.6 and its equations are shown in Table 4.1.

Table 4.1: Types of CRC

CRC Mode	**Equation**
CRC24	gCRC24(D) = D24 + D23 + D6 + D5 + D + 1
CRC16	gCRC16(D) = D16 + D12 + D5 + 1
CRC12	gCRC12(D) = D12 + D11 + D3 + D2 + D + 1
CRC8	gCRC8(D) = D8 + D7 + D4 + D3 + D + 1

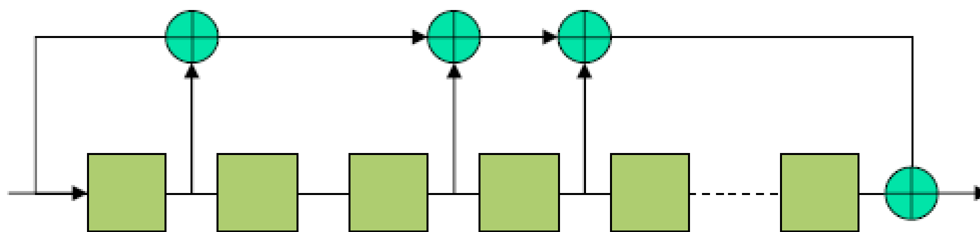


Figure 4.6: CRC as shift register

After modeling the CRC block and other blocks of the chain in HDL we face a problem that rates differ from a block to another so we have to control the time of handshaking between blocks to ensure proper data transfer. In addition, we have to unify the interface of the blocks to realize the concept of the PDR.

So we design a top controlled module consists of a controller, a FIFO (first input first output) and the designed block as shown in Figure 4.7. Fifo store the input bit stream and controller to control the fifo and the designed block.

In this module we add two signals enable and finished to control the handshake. Enable comes from the next block to indicate that it is ready to receive data.

Finished is sent to the previous block to indicate that the current block has finished its function. So the top controlled module of CRC will be as shown in Figure 4.8.

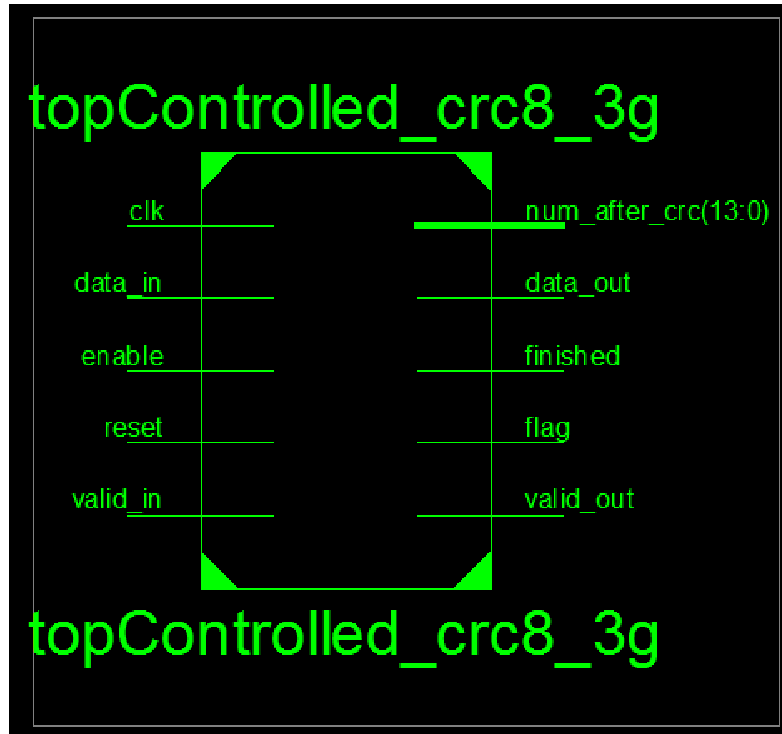


Figure 4.7: Top controlled CRC

The pins description of the controlled module is shown in Table 4.2

4.2.2 Segmentation

Segmentation of the bit sequence from transport block concatenation is performed if $X_i > Z$. The code blocks after segmentation are of the same size. The number of code blocks on TrCH 'i' is denoted by C_i. If the number of bits input to the segmentation, X_i, is not a multiple of Ci, filler bits are added to the beginning of the first block. If turbo coding is selected and Xi < 40, filler bits are added to the beginning of the code [5]

Table 4.2: pins description of CRC

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data
Data_out	The output data of the block
Valid_out	The signal indicates that current data_out is valid data
Finished	The signal indicated that the CRC is ready for the new frame
Enable	The signal indicates that the next block is ready to have data
Num_after_crc	The signal indicates that total number of bits after crc
Flag	The signal indicates that the num_after_crc is valid

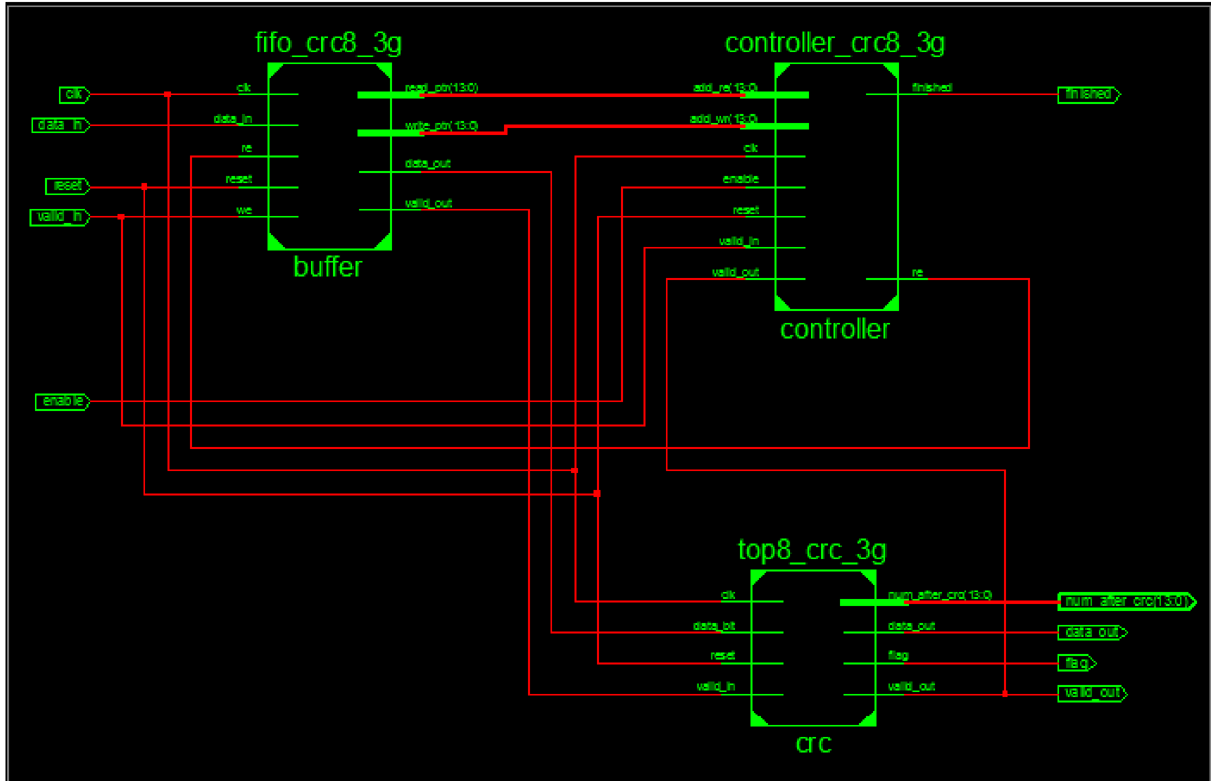


Figure 4.8: Internal module of CRC

block [6] [4]. The filler bits are transmitted and they are always set to 0. The maximum code block sizes are:

Convolutional coding: $Z = 504$;

Turbo coding: $Z = 5114$.

The bits output from code block segmentation, for $C_i \neq 0$, are denoted by o_{ir1}, o_{ir2}, o_{ir3}… o_{irki} where i is the TrCH number, r is the code block number, and K_i is the number of bits per code block.

Number of code blocks:

$C_i = [X_i/Z]$

Number of bits in each code block (applicable for $C_i \neq 0$ only):

```
if Xi < 40 and Turbo coding is used, then
Ki = 40
else
Ki = [Xi / Ci]
end if
Number of filler bits: Yi = CiKi - Xi
for k = 1 to Yi                                   --Insertion of filler bits
Oi1k=0
end for
for k = Yi+1 to Ki
Oik=Xi,(K-Yi)
end for
r = 2                                             -- Segmentation
while r ≤ Ci
for k = 1 to Ki
Oirk=Xi, (k+(r-1)-Ki-Yi)
end for
r = r+1
end while
```

Concerning the HDL implementation the Figure 4.9 shows the interface of the Segmentation

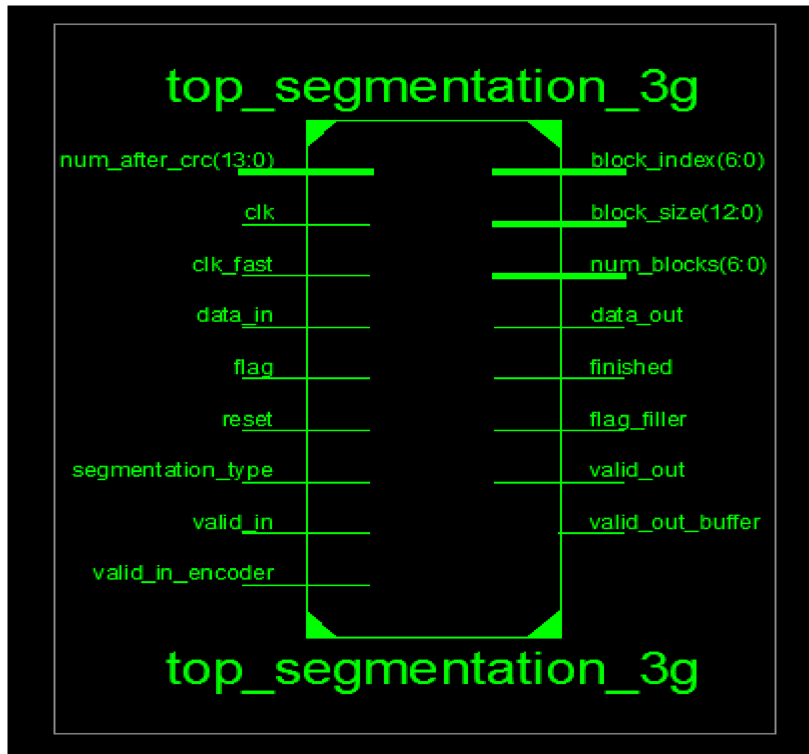


Figure 4.9: Segmentation Interface

The internal block diagram is shown in Figure 4.10

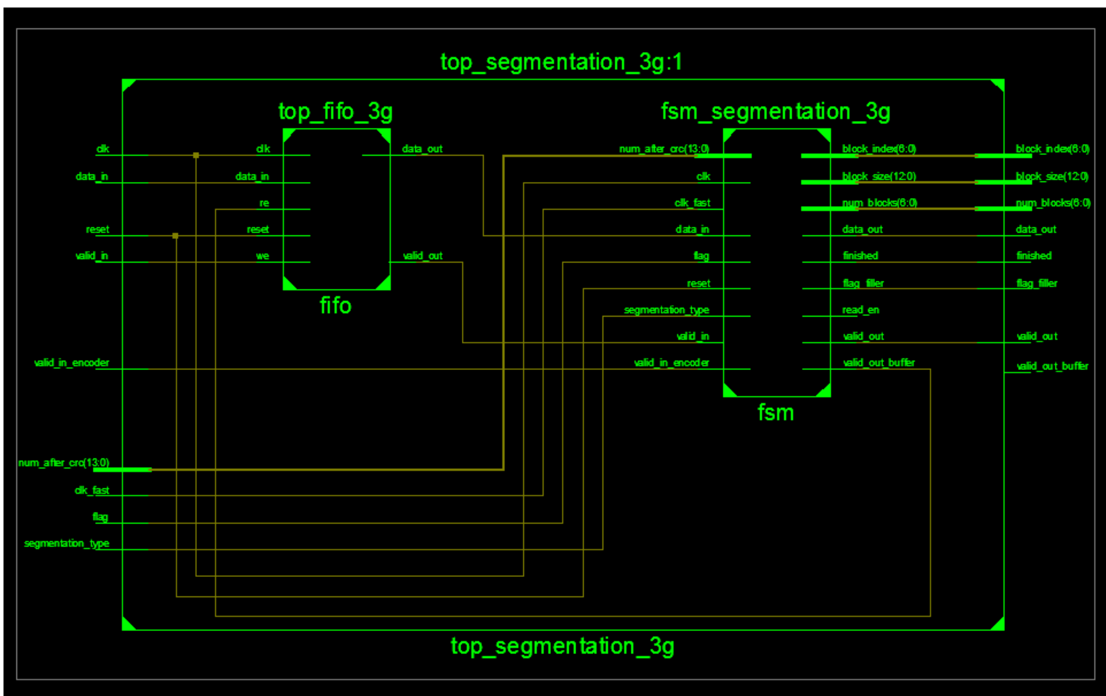


Figure 4.10: Internal Design for Segmentation

Table 4.3: Interface Signal Decleration

PIN	Description
num_after_crc	This is input signal from CRC that indicates the total number of data bits plus concatenated CRC bits
clk_fast	Connected to the clk_spreading signal to increase the speed for the division required to generate the number of blocks produced
Data_in	The input bits
Flag	Input signal from the CRC that indicates that the num_after_crc is ready to be read for the segmentation block
Segmentation_Type	To differentiate between Convolutional Encoder “0” or Turbo Encoder “1”
valid_in	This signal indicates that current data_in is valid data
valid_encoder	This signal indicates that the next block is ready to have data
Block_index	Output signal that indicates the index for the block being transmitted to the encoder
Block_size	Number of bits included in each block after performing the segmentation process.
Num_Blocks	Total number of blocks output from the segmentation process
Data_out	The output bits
finished	The signal indicated that the mapper is ready for the new frame
Flag_filler	Output signal used for the encoder such that it does not read the extra zero filler bit that remains on the bus while moving from state to another inside the code. Consequently, this reserve that valid_out signal to remain always one within the data block as shown in Figure 4.11
valid_out	This signal indicates that current data_out is valid data

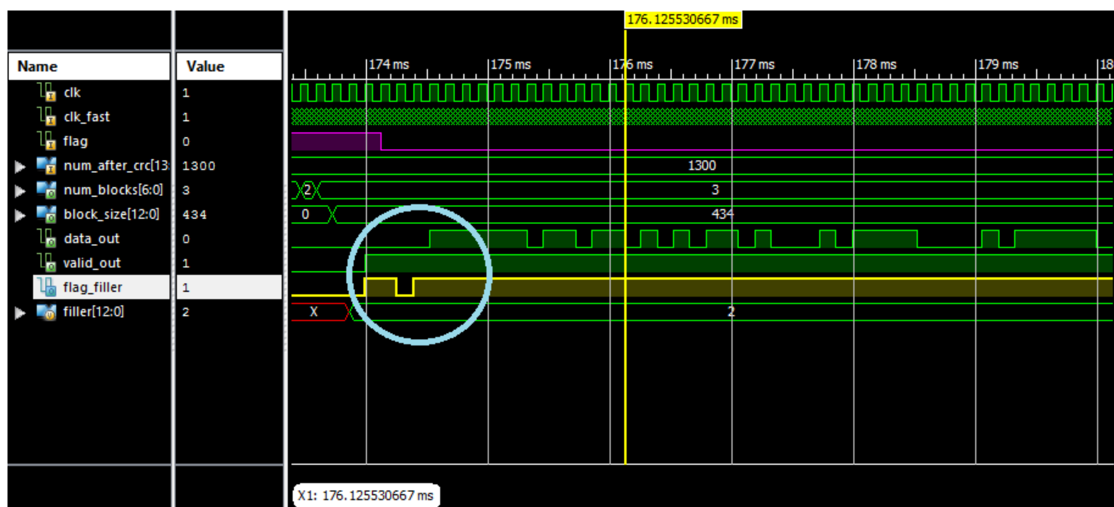


Figure 4.11: flag filler explanation use

The output segments and the valid_out that is input to the encoder to intiate the processing for each segment is as shown in Figure 4.12

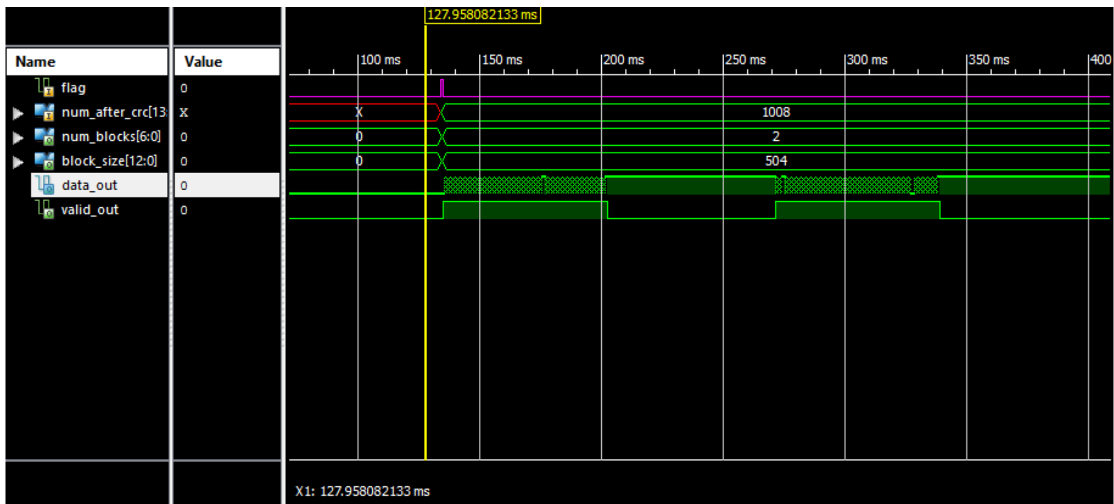


Figure 4.12: Output from the segmentation block

One of the challenging in the segmentation is the division used to generate either the num_blocks or block_size that will be used by the successing blocks. For the num_blocks, it is a result for dividing the num_after_crc by the constant value for either the convolutional or turbo decoder. Since this division is by a constant and large value, therefore we use the concept of subtracting in division and by making use for clk_spreading to maximize the speed for division as shown in Figure 4.13. Moreover, during the time of division segmentation block is still waiting to data to be exit from the CRC. On the other hand, we use a combinational synthesizable divider to get the block_size since if we make use for the subtraction concept, number of clock cycles will be wasted since the denominator is very small relatively to the numerator. Also, data were already stored in the FIFO for the segmentation and so any waste of cycles will lead in slowing down the speed for data transmission within the chain

4.2.3 Encoder

Convolutional codes with constraint length 9 and coding rates 1/3 and 1/2 are defined. The configuration of the convolutional coder is presented in Figure 4.14, Figure 4.15.

Output from the rate 1/3 convolutional coder shall be done in the order output0, output1, output2, output0, output1, output 2, output 0, ..., output 2. Output from the rate 1/2 convolutional coder shall be done in the order output 0, output 1, output 0, output 1, output 0... output 1 [5].

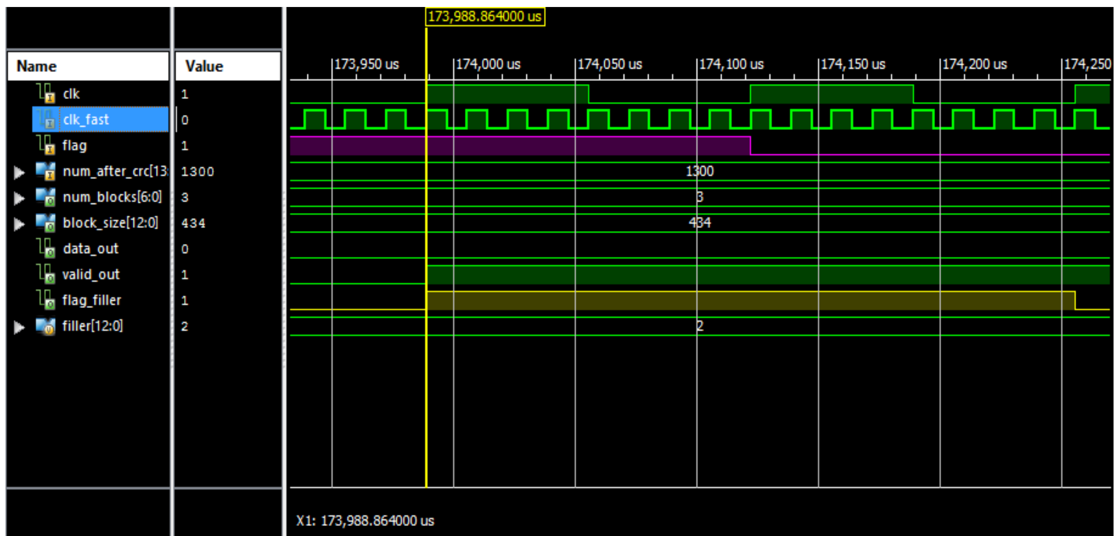


Figure 4.13: clk and clk_fast period

8 tail bits with binary value 0 shall be added to the end of the code block before encoding. The initial value of the shift register of the coder shall be "all 0" when starting to encode the input bits.

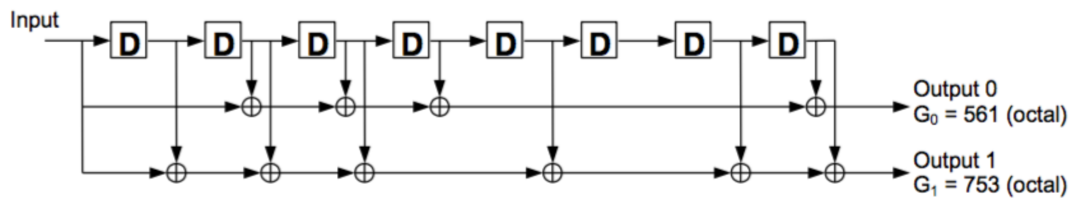


Figure 4.14: Rate 1/2 convolutional encoder

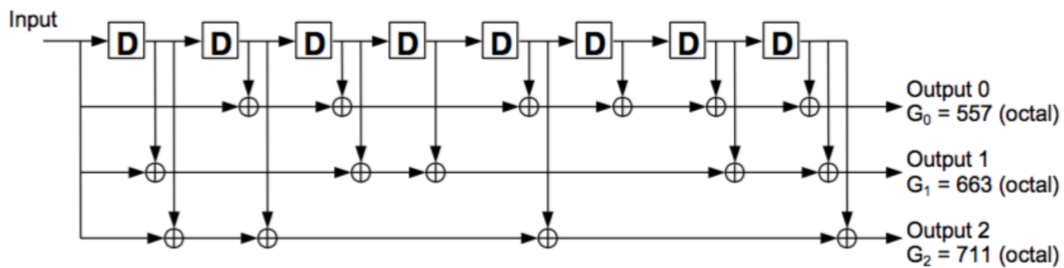


Figure 4.15: Rate 1/3 Convolutional encoder

The top module of the 3g convolutional encoder rate half is shown in Figure 4.16. The pins description of the top controlled is shown in Table 4.4.

Comment: The signal C is just through the block, which is required for the concatenation block.

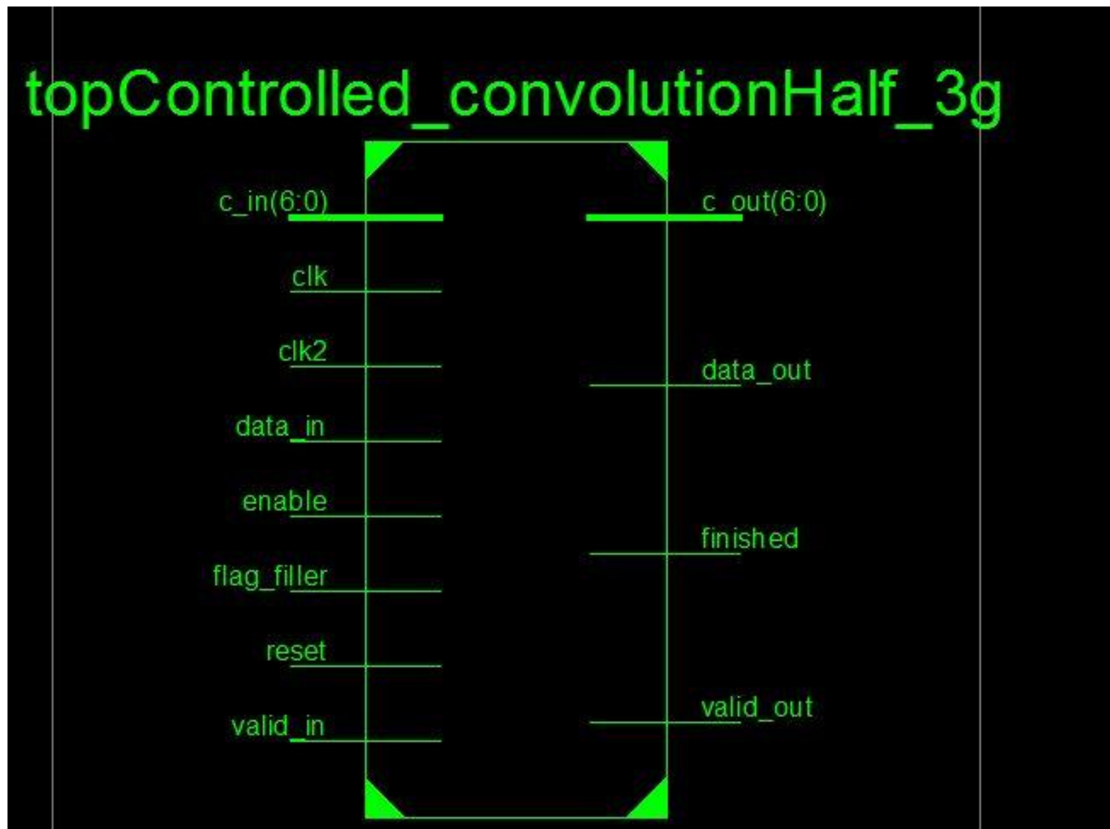


Figure 4.16: Schematic

Table 4.4: pins description of encoder

PIN	Description
C_in(6:0)	Number of code blocks from the segmentation
Clk	Clock of the all encoder blocks
Clk2	Clock of the serial output
Data_in	Data in for the convolutional encoder
Enable	Working enable for the encoder
Flag_filler	Flag from segmentation
Reset	Reset encoder registers by inserting Zeros
Valid_in	Valid in to consider the input
C_out(6:0)	Indicates the number of cade blocks
Data_out	Encoder input
Finished	Signal indicates the block is ready for the new frame
Valid_out	Valid out signal to the next block

The termination problem solved by adding a counter that starts when the valid_in signal goes low after it was high; the counter ends after eight counts thus the termination done.

Detailed block diagram for the blocks that shown internal construction of the convolutional encoder in Figure 4.17.

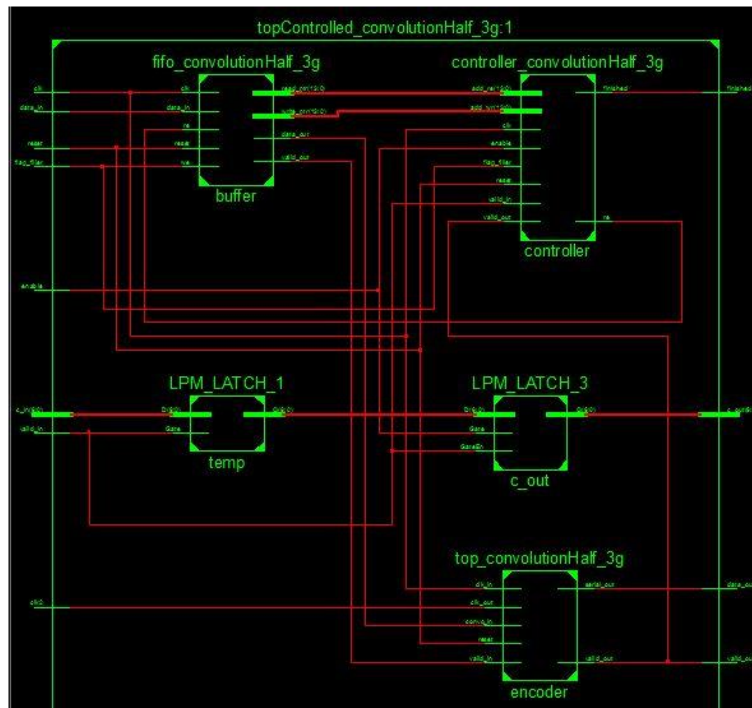


Figure 4.17: Internal block diagram

4.2.4 Code block concatenation

The input bit sequence for the code block concatenation block are the sequences e_{rk}, for r = 0,…..,C-1 and k = 0,…..,E_r-1. The output bit sequence from the code block concatenation block is the sequence f_k for k = 0,….,G-1.. The code block concatenation consists of sequentially concatenating the rate matching outputs for the different code blocks. Therefore,

Set k = 0 and r = 0

while r < C

Set j = 0

while j < E_r

$f_k = e_{rj}$ k = k +1 j = j +1

end while

r = r +1 end while

The block interface is as shown in Figure 4.18, the signals declaration and description is as shown in Table 4.5 and the block simulation is as shown in Figure 4.19.

There is no difference between the implementation of the code block concatenation in 4G and 3G, they are exactly the same [5].

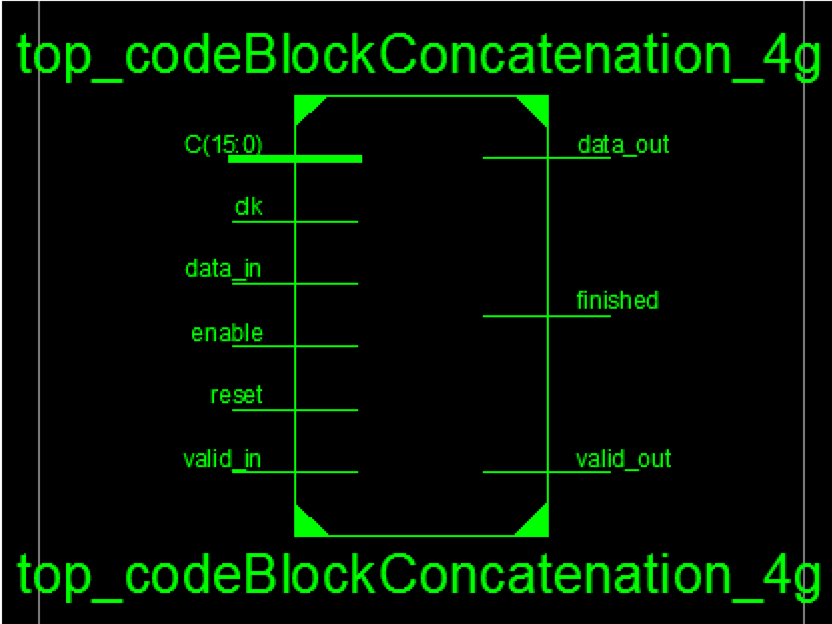


Figure 4.18: Code block concatenation block interface.

Table 4.5: Code block concatenation block signals declaration

PIN	PIN Type	Description
C	IN	Total number of code blocks (segmentation section)
Enable	IN	This signal indicates that the next block is ready to have data
valid_in	IN	This signal indicates that current data_in is valid data
data_in	OUT	The input bits
finished	OUT	This signal indicates that the interleaver is ready to have a new frame
valid_out	OUT	This signal indicates that current data_out is valid data
data_out	OUT	The output bits

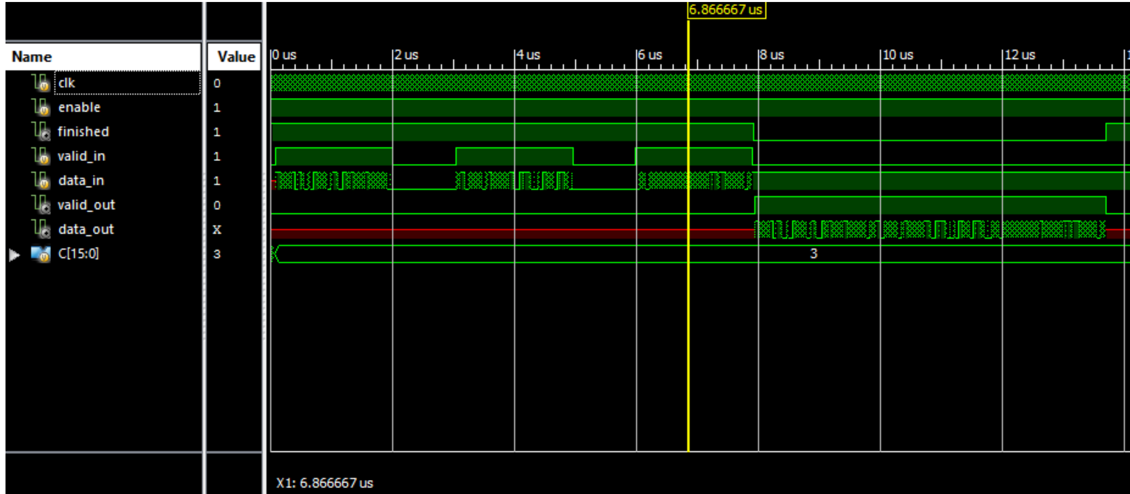


Figure 4.19: Code block concatenation block simulation.

4.2.5 Radio frame equalization

To ensure data can be divided into equal-sized blocks, padding bits are concatenated at the end of the set of coded bits, and can be either logical 0s or 1s [5].

For example, if a TrCH with a 80ms TTI (8 Frames) have 70 bits after channel coding; two padding bits would be added to give a total of 72 bits which can later be split into eight sets (radio frames) of nine bits each.

Radio frame size equalization is only performed in the UL.

The input bit sequence to the radio frame size equalization is denoted by:$e_{i1}, e_{i2}, e_{i3}, \dots \dots, e_{iE_i}$ where (i) is TrCH number and E_i the number of bits. The output bit sequence is denoted by:$t_{i1}, t_{i2}, t_{i3}, \dots, t_{iT_i}$ where T_i is the number of bits. The output bit sequence is derived as follows:

$$t_{iK} = e_{iK} \quad for\ k = 1 \dots \dots . E_i$$

$$t_{i1}{=}0 \qquad for\ k = E_i + 1 \dots \dots . . T_i$$

$$T_i = F_i * N_i$$

$$N_i = \lceil \frac{E_i}{F_i} \rceil$$

F_i is the number of segments (frames) and it depends on TTI as shown in Table 4.6

N_i is the number of bits per segment after size equalization.

Table 4.6: number of frames

TTI	F_i
10ms	1
20ms	2
40ms	4
80ms	8

Concerning the HDL implementation, Figure 4.20 shows the interface. The pins description is in the following Table 4.7. The implementation depends on counter counts input bits from zero till number of frames and repeats again , for example TTI=40 and number of bits equal 1001,the counter will end at value 1 so equalizer will pad 3 zeros to reach value 4 (number of frames) as shown in Figure 4.21.

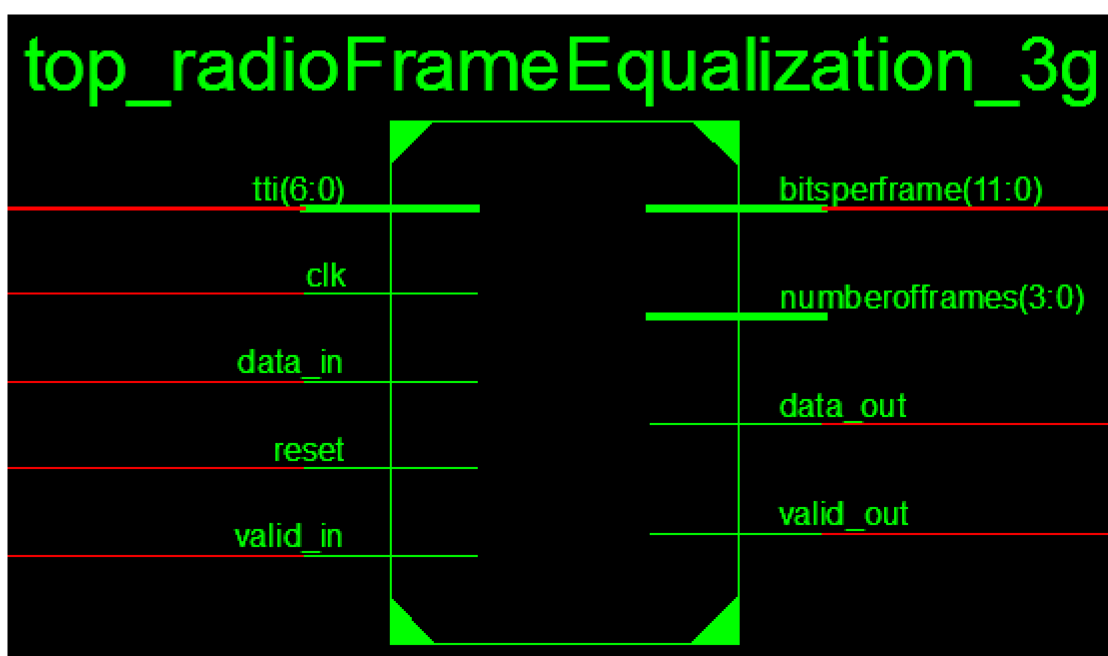


Figure 4.20: Radio Frame Equalization interface

Table 4.7-Frame Equalization Pin description

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data
TTI	Transmission Time Interval possible values are : 10 , 20 , 40 and 80
bitsperframe	Number of bits per frame N_i
numberofframes	Number of frames F_i
finished	The signal indicated that the block is ready for the new frame
enable	The signal indicates that the next block is ready to have data

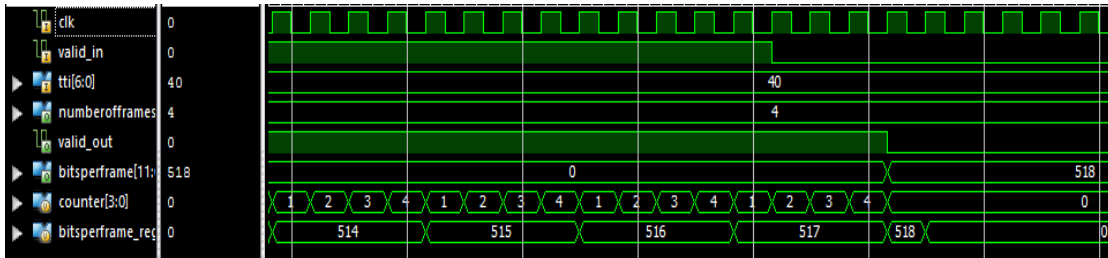


Figure 4.21: Equalizer timing diagram example

4.2.6 First interleaving

Interleaving is a way to re-arrange data in a non-contiguous way to make it stand burst errors. These types of errors can destroy many bits in a row and make it hard to recover using FEC coding, since these expects the errors to be more uniformly distributed. This method is popular because it is a less complex and cheaper way to handle burst errors than directly increasing the power of the error correction scheme and interleaving cause increasing the performance of decoding as shown in Table 4.8 [5].

The main disadvantage of using interleaving techniques is that increases latency because the entire interleaved block must be received before the packets can be decoded. Interleaving period equals to TTI (Transmission Time Interval) which determines then number of columns in the interleaving matrix (10, 20, 40, 80ms => 1, 2, 4, 8 columns).

Table 4.8: comparison between with and without interleaving

Without interleaving	**With interleaving**
Transmitted Bits : b0 b1 b2 b3 b4 b5 b6 b7 b8 Received Bits : b0 b1 b2 b3 x x x b7 b8 (x indicates to error in bit)	Transmitted Bits : b0 b1 b2 b3 b4 b5 b6 b7 b8 Interleaved Bits : b0 b3 b6 b1 b4 b7 b2 b5 b8 Received Bits : b0 b3 b6 b1 x x x b5 b8 Deinterleaved Bits: b0 b1 x b4 b5 b6 x s b8
Burst errors hard to recover	Distributed errors easy to recover

First interleaver is Inter-frame interleaving where all frames are interleaved together Three steps for interleaving as shown in Figure 4.22.

- Write input bits into the matrix row by row
- Perform inter-column permutation based on pre-defined patterns (according to the TTI)
- Read output bits from the matrix column by column

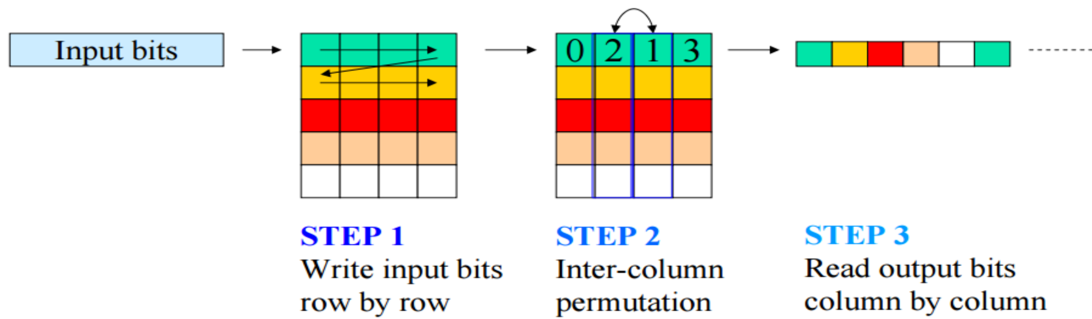


Figure 4.22: Steps of Interleaving

Inter-column permutation patterns are defined in Table 4.9.

Table 4.9-Inter-column permutation for first interleaver

TTI	Number of columns	Inter-column permutation patterns
10ms	1	<0>
20ms	2	<0,1>
40ms	4	<0,2,1,3>
80ms	8	<0,4,2,6,1,5,3,7>

The following Figure 4.23 shows an example of first interleaver where TTI=40ms and 16 bits. Number of columns equal 4 so number of rows equal 4 so first writing bits in the interleaving matrix row by row then Perform inter-column permutation <0,2,1,3> then read from matrix column by columns.

Concerning the HDL implementation, Figure 4.24 shows the interface of first interleaver

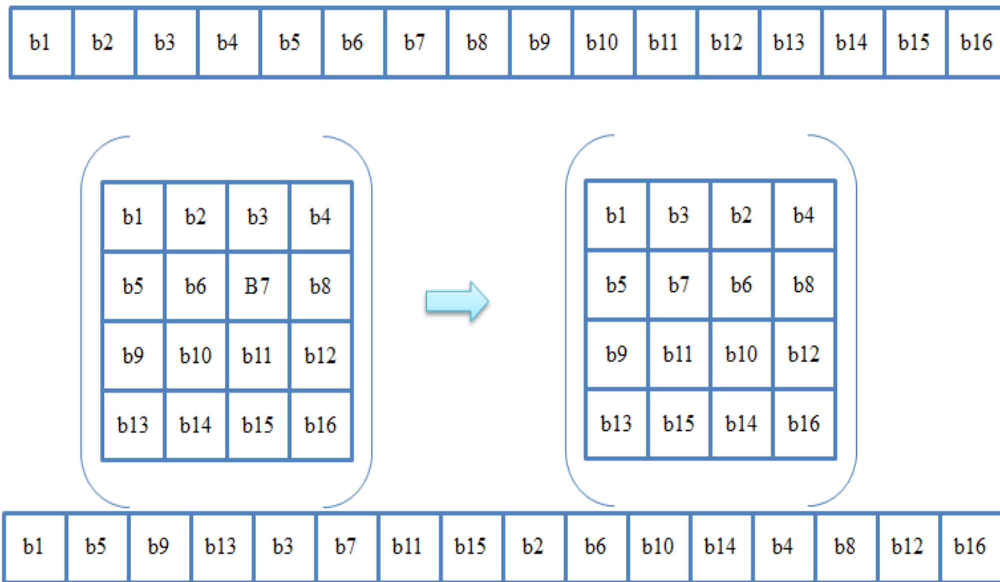


Figure 4.23: Interleaving Example

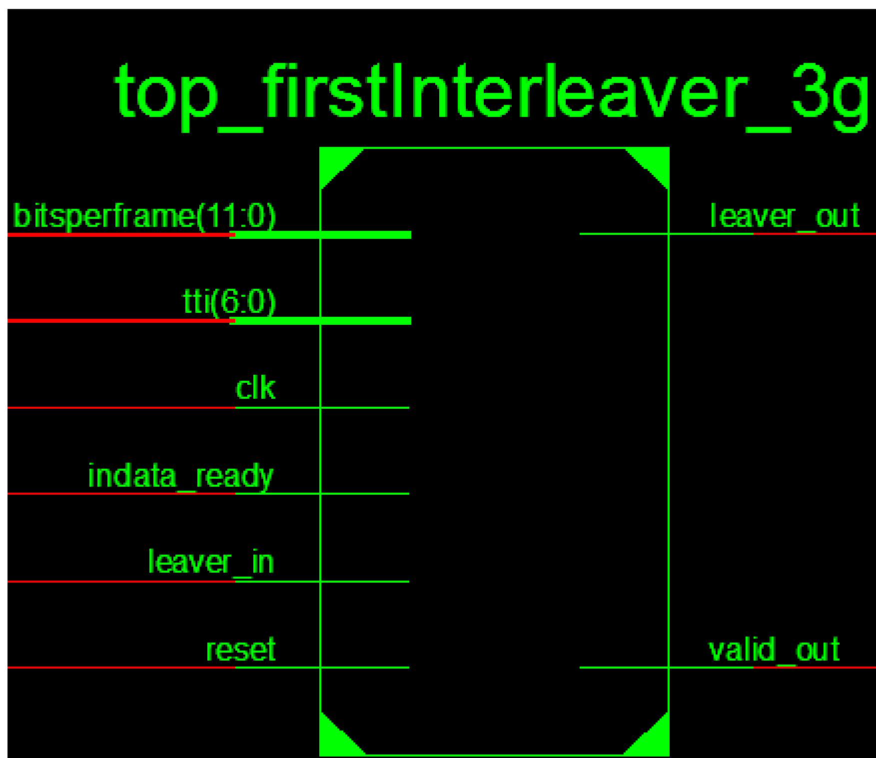


Figure 4.24: First Interleaver interface

The internal block diagram is shown in Figure 4.25

Table 4.10-First Interleaver Pin description

PIN	**Description**
Leaver_in	The input bits
Indata_ready	The signal indicates that current data_in is valid data
TTI	Transmission Time Interval possible values are : 10 , 20 , 40 and 80
Bitsperframe	Number of bits per frame N_i

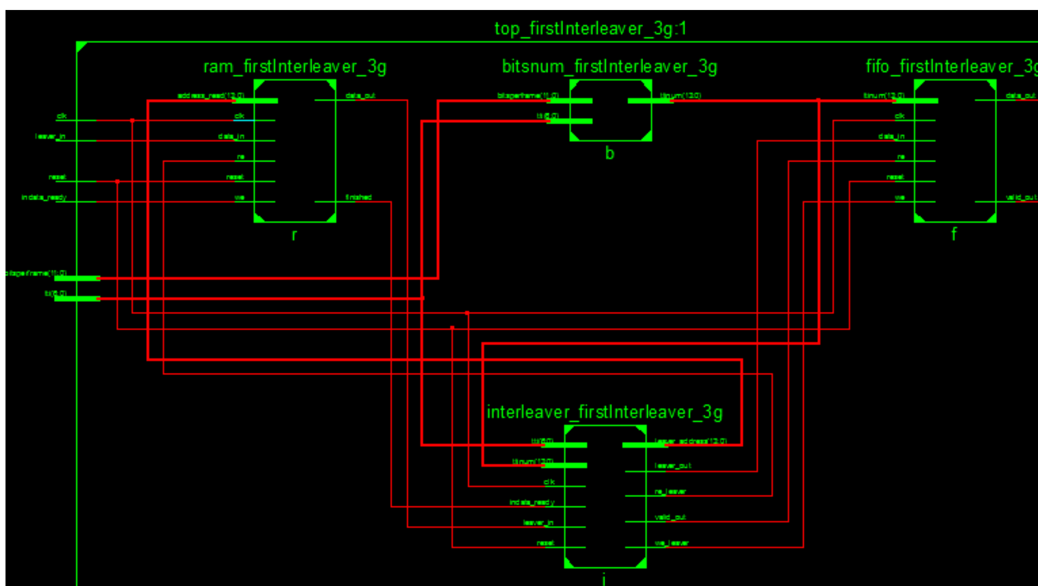


Figure 4.25: First Interleaver Block Diagram

The implementation depends on a ram which we write in it in-of order and read from it out-of-order depending TTI.

For TTI=80ms, there are 8 columns and the matric after permutation shown in following table

b_0	b_4	b_2	b_6	b_1	b_5	b_3	b_7
b_8	b_{12}	b_{10}	b_{14}	b_9	b_{13}	b_{11}	b_{15}
b_{16}	b_{20}	b_{18}	b_{22}	b_{17}	b_{21}	b_{19}	b_{23}

So the address start with 0 (first column in columns permutation) and increment by 8 (number of frames) till end and then 4 (second column in columns permutation) and increment by 8 (number of frames) till end and so on.

For TTI=40ms, there are 4 columns and the matric after permutation shown in following table

b_0	b_2	b_1	b_3
b_4	b_6	b_5	b_7
b_8	b_{10}	b_9	b_{11}

So the address start with 0 (first column in columns permutation) and increment by 4 (number of frames) till end and then 2 (second column in columns permutation) and increment by 4 (number of frames) till end and so on.

For TTI=20ms, there are 2 columns and the matric after permutation shown in following table

b_0	b_1
b_2	b_3
b_4	b_5

So the address start with 0 (first column in columns permutation) and increment by 2 (number of frames) till end and then 1 (second column in columns permutation) and increment by 2 (number of frames) till end and so on.

For TTI=10ms, reading from ram in-of-order.

Figure 4.26 shows the timing diagram for example TTI=40ms, address_read starts with 0 and increments by 4 till 2068 then be 2.

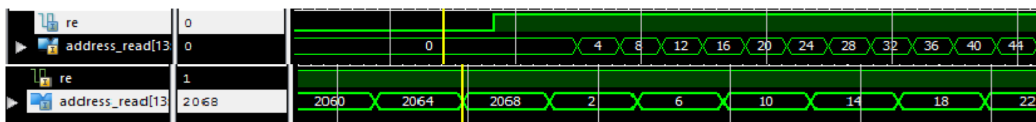


Figure 4.26: First interleaver timing diagram

4.2.7 Radio Frame Segmentation

When the transmission time interval is longer than 10ms, the input bit sequence is segmented and mapped onto consecutive F_i radio frames [5].

In our 80ms TTI example above, with 72 bits after radio frame equalization, the first nine interleaved bits will be transmitted in the first radio frame, the next nine bits in the second radio frame, and so on over all eight frames of the TTI. The input bit sequence is denoted by: $x_{i1}, x_{i2}, x_{i3}, \dots \dots, x_{iX_i}$ where i is the TrCH number and X_i is the number bits. The F_i output bit sequences per TTI are denoted by: y_{i,n_i1} , y_{i,n_i2} , y_{i,n_i3} ………, y_{i,n_iY_i} where n_i is the segment number and Y_i is the number of bits per radio frame for TrCH i. The output sequences are defined as follows

$$y_{i,n_ik} = x_{i,((n_i-1).Y_i)+k} \quad ,k = 1 \dots \dots . Y_i$$

Where $Yi = (Xi / Fi)$ is the number of bits per segment

Concerning the HDL implementation, Figure 4.27 shows the interface

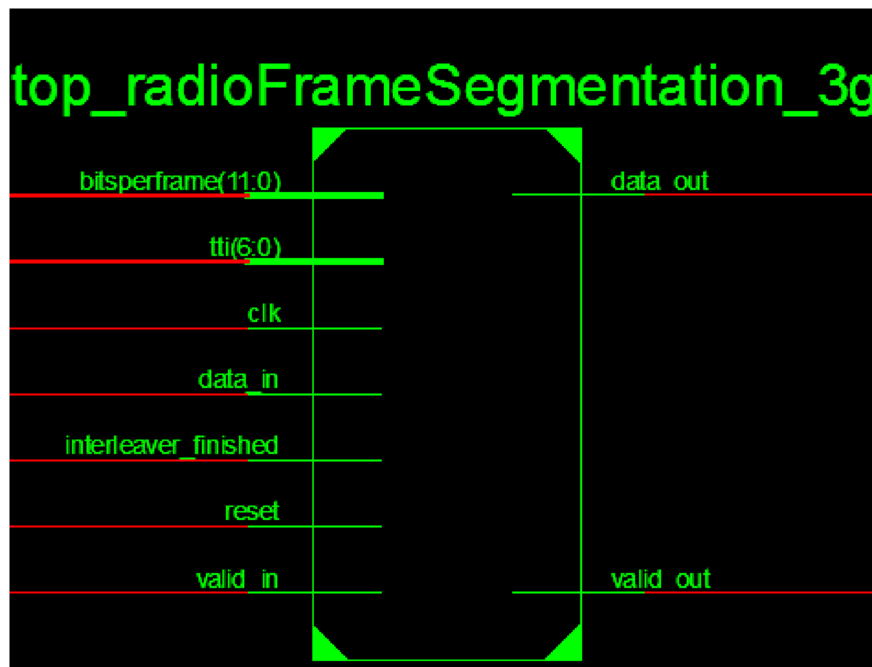


Figure 4.27: Radio Frame Segmentation interface

The pins description is in the following Table 4.11.

Table 4.11-Radio Frame Segmentation Pin description

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data
TTI	Transmission Time Interval possible values are : 10 , 20 , 40 and 80
bitsperframe	Number of bits per frame N_i
Interleaver_finished	The signal indicated that the second interleaver is ready for the new frame

The internal block diagram is shown in Figure 4.28

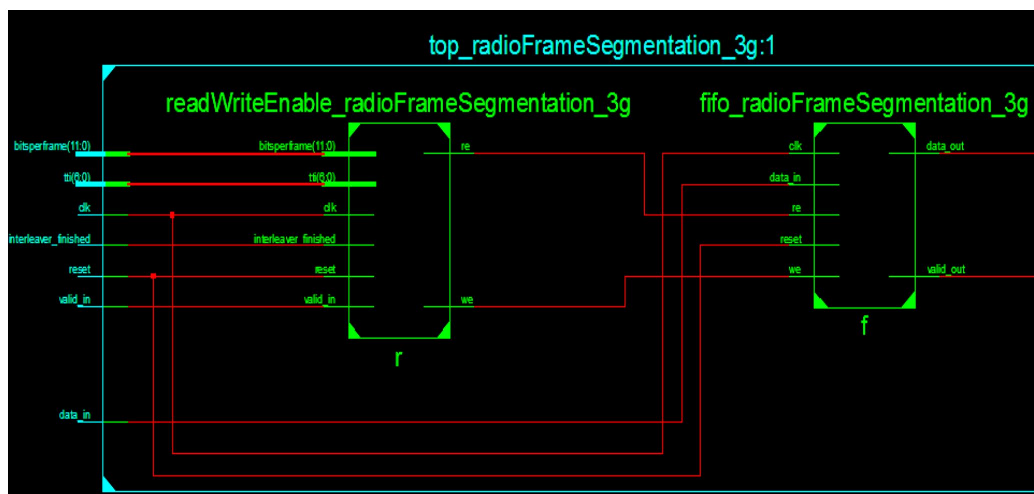


Figure 4.28: Radio Frame Segmentation Block Diagram

Segmentation outputs the first block and waits signal from second interleaver that indicates the interleaving is finished and it is ready for new block as shown in Figure 4.29.

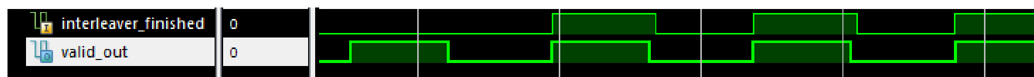


Figure 4.29: Radio Frame Segmentation timing diagram

4.2.8 Second interleaving

Second interleaving is Intra-frame interleaving which means interleaving is done frame by frame so interleaving period is 10ms.

Number of columns of the interleaving matrix is equal 30. The columns of the matrix are numbered 0, 1, 2… 29 from left to right.

The number of rows of the matrix is determined by finding minimum integer R such that: *U* <= R *30 where U is the number of bits in one radio frame and The rows of rectangular matrix are numbered 0, 1, 2, …, R2 - 1 from top to bottom.

Writing the input bit sequence into the R*C matrix row by row starting with bit in column 0 of row 0 and if R * C> U, the dummy bits are padded .These dummy bits are pruned away from the output of the matrix after the inter-column permutation.

Performing the inter-column permutation for the matrix based on the pattern that is shown in Table 4.12 [5].

Table 4.12-inter-column permutation for second interleaver

Number of columns C2	**Inter-column permutation pattern**
30	<0, 20, 10, 5, 15, 25, 3, 13, 23, 8, 18, 28, 1, 11, 21,6, 16, 26, 4, 14, 24, 19, 9, 29, 12, 2, 7, 22, 27, 17>

The output of the block interleaver is the bit sequence read out column by column and pruned by deleting dummy bits that were padded to the input of the matrix before the inter-column permutation.

Concerning the HDL implementation, Figure 4.30 shows the interface.

The pins description is in the following Table 4.13.

The internal block diagram is shown in Figure 4.31.

Table 4.13 : Second Interleaver pin description

PIN	**Description**
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data
TTI	Transmission Time Interval possible values are : 10 , 20 , 40 and 80
bitsperframe	Number of bits per frame N_i
finished	The signal indicated that the block is ready for the new frame
enable	The signal indicates that the next block is ready to have data

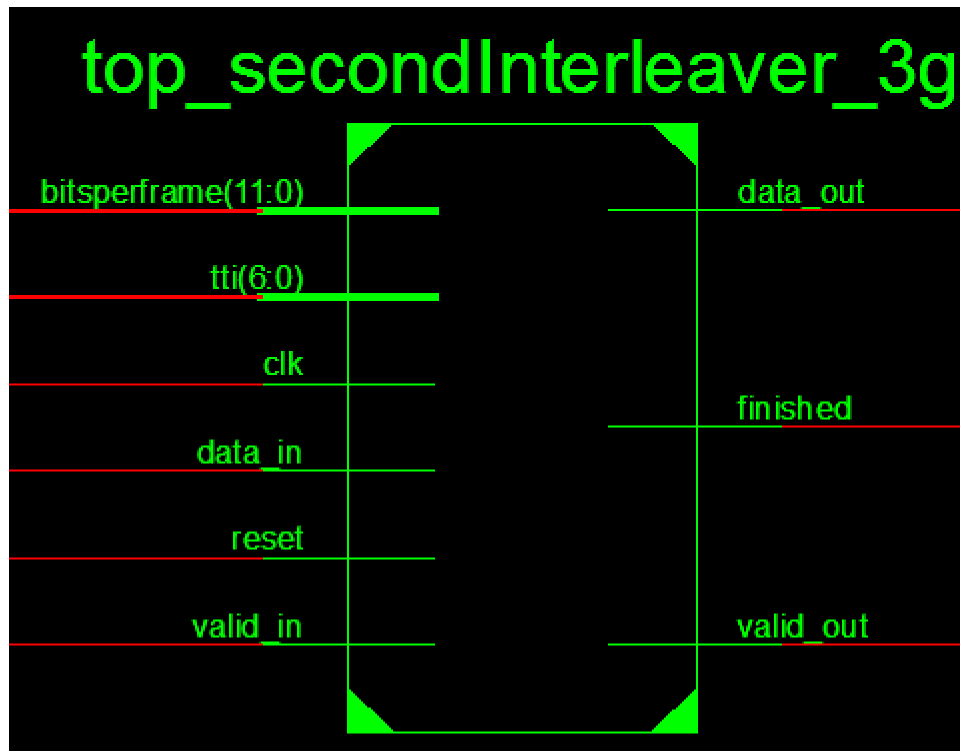


Figure 4.30: Second Interleaver interface

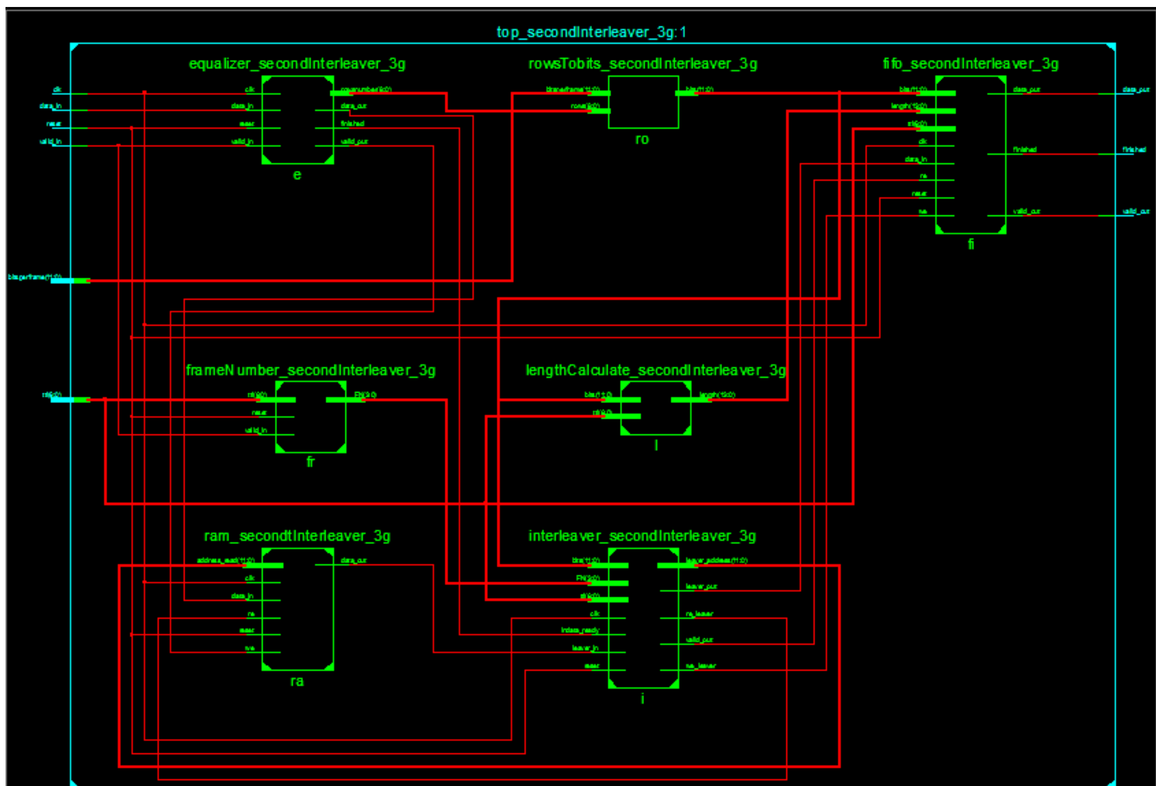


Figure 4.31: Second Interleaver Block Diagram

Equalizer adds the dummy bits and calculates number of rows, for example if the number of bits equal 518 so number of rows equal 18 and equalizer pads bits to be 540 bits as shown in Figure 4.32 .

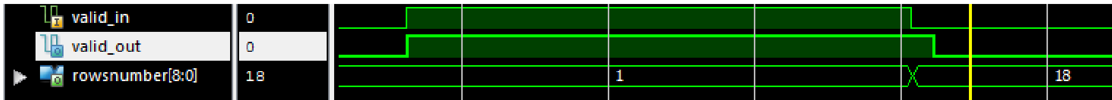


Figure 4.32: Second interleaver timing diagram 1

The implementation depends on a ram which we write in it in-of order and read from it out-of-order, there are 30 columns and the matrix after permutation shown in following table

b_0	b_{20}	b_{10}	b_5	b_{15}	b_{25}	b_3	b_{13}	… … … … … ..	b_7	b_{22}	b_{27}	b_{17}
b_{30}	b_{50}	b_{40}	b_{35}	b_{45}	b_{55}	b_{33}	b_{15}	… … … … … ..	b_{37}	b_{52}	b_{57}	b_{47}
b_{60}	b_{80}	b_{70}	b_{65}	b_{75}	b_{85}	b_{63}	b_{23}	… … … … … ..	b_{67}	b_{82}	b_{87}	b_{77}

So the address start with 0 (first column in columns permutation) and increment by 30 (number of columns) till end and then 20 (second column in columns permutation) and increment by 30 (number of frames) till end and so on.

Figure 4.33 shows the timing diagram for example address_read starts with 0 and increments by 30 till end then be 20.

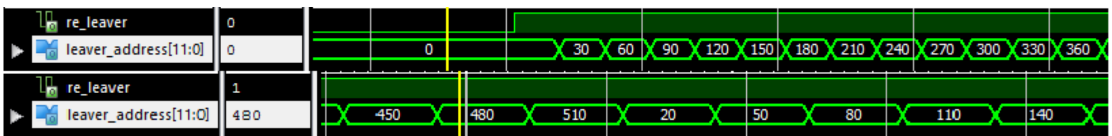


Figure 4.33: Second interleaver - timing diagram 2

4.2.9 Interleaving Block

Figure 4.34 shows the block diagram of overall interleaving block containing: Radio Frame Equalizer, First Interleaver, Radio Frame Segmentation and Second Segmentation.

Figure 4.35 shows timing diagram of overall interleaving block for two blocks of Data. First has TTI=80 and 1116 bits and second has TTI=40 and 723 bits Radio frame equalizer pads the first block with 4 bits to be equally sized blocked with140 bits per frame and 8 frames and pads the second block with 1 bit to be equally sized blocked with 181 bits per frame and 4 frames then first interleaver is inter interleaving which interleave all frames together but second interleaver is intra interleaving which interleave frame by frame.

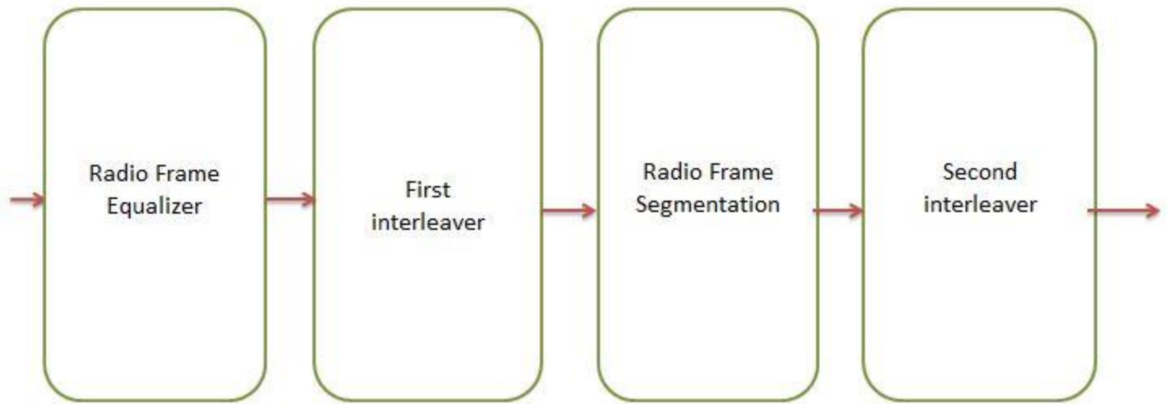


Figure 4.34: Interleaving Block Diagram

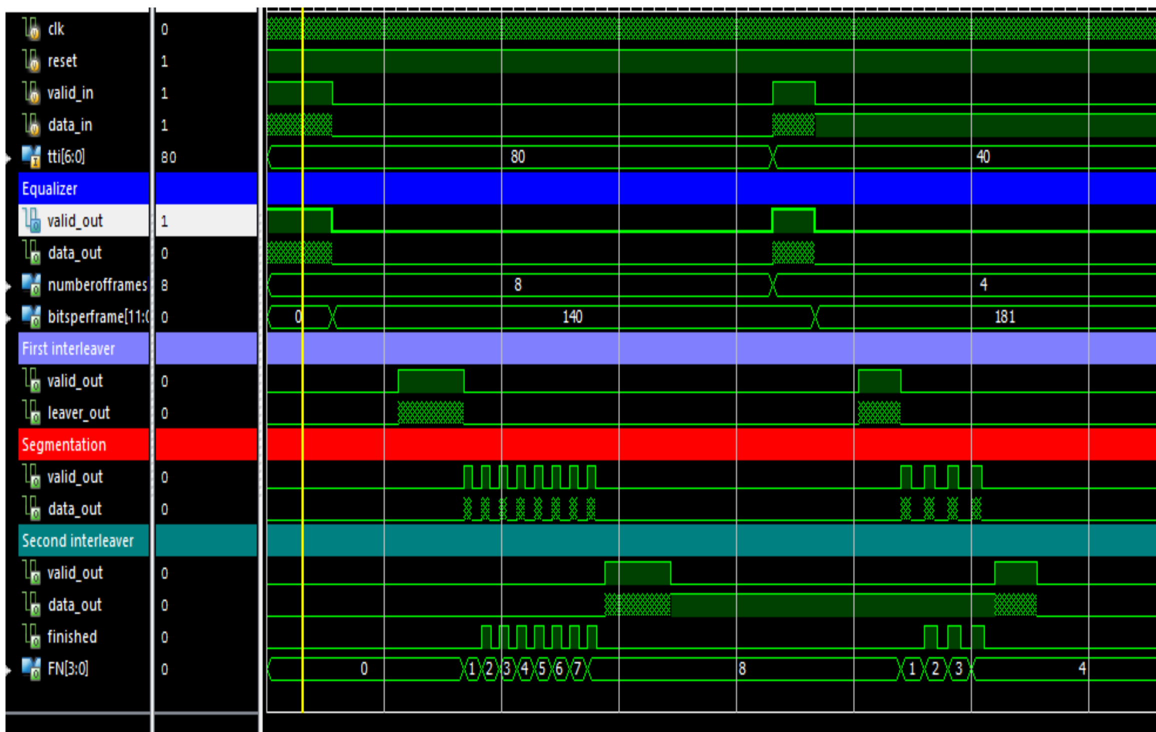


Figure 4.35: Interleaving Block Timing Diagram

4.2.10 Spreading and Scrambling

Spreading is applied to the physical channels. It consists of two operations

- Channelization operation: increase the bandwidth of the signal using fully orthogonal codes called channelization codes to not interfere with each other. Every data is transformed into number of chips. The number of chips per data symbol is called the Spreading Factor (SF). The channelization codes are picked from the code tree as shown in Figure 4.37 [7].

- In our project we transmit only one DPDCH (Dedicated Physical Data Channel) DPDCH1 shall be spread by $C_{SF,K}$ where SF is the spreading factor of DPDCH1 and k= SF / 4. We found that the generated code of channelization consists of a periodically repeated sequence of (1, 1,-1, 1).

The data out of the spreading block will be raise by ratio SF over the data input bits and this is shown in Figure 4.36

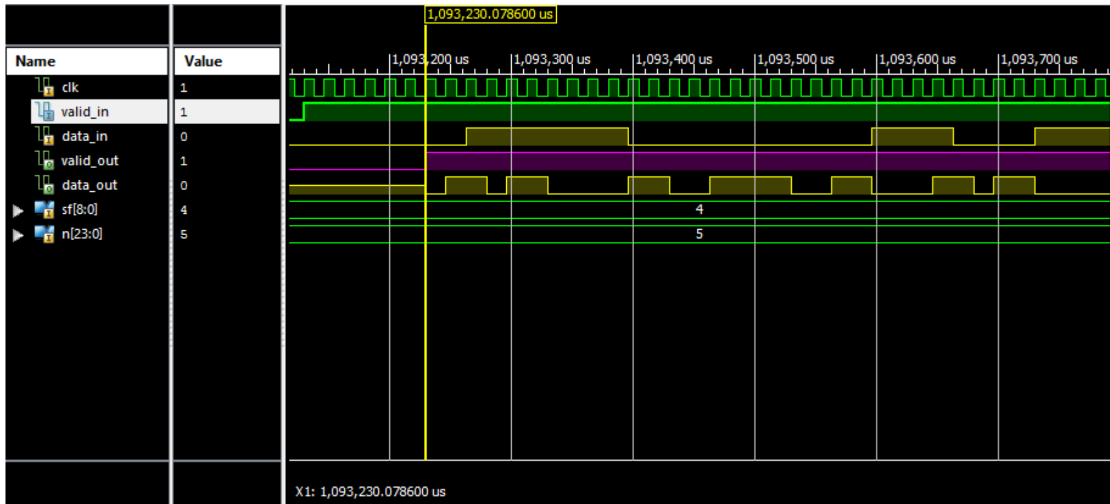


Figure 4.36: waveform of spreading block

- Scrambling operation: Scrambling code is applied to the spread signal and it doesn’t affect the signal bandwidth. The scrambling code can be a long code (a Gold code with 38400chips) or a short code (256 chips) the long code is used if the BS uses a Rake receiver and the short code is used if multiuser detector and interference cancellation receivers are used in BS. In our project we use long code.

- The data after scrambler will be unique so we can separate between different users in uplink and the receiver can retrieve the original data by using the same scrambler sequence. The overall block diagram is shown in Figure 4.38.
- The controlled module of the overall block is shown in Figure 4.39.

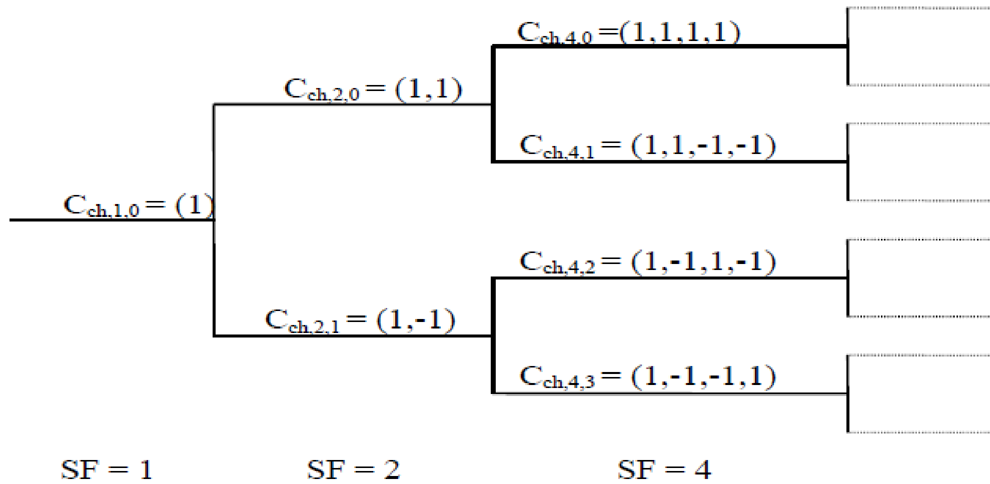


Figure 4.37: Code-tree for generation of Orthogonal Spreading Factor codes

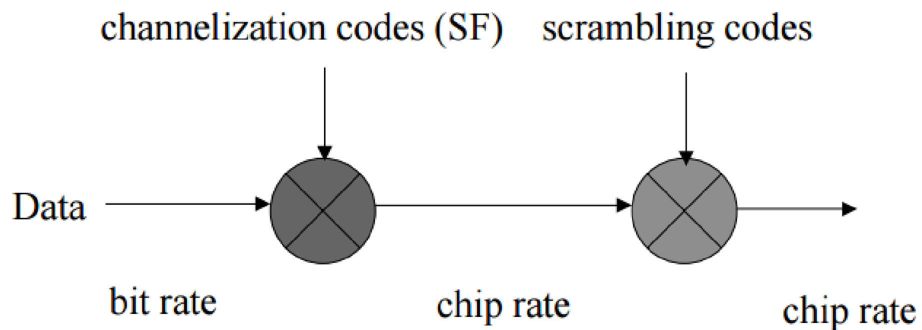


Figure 4.38: block diagram of spreading

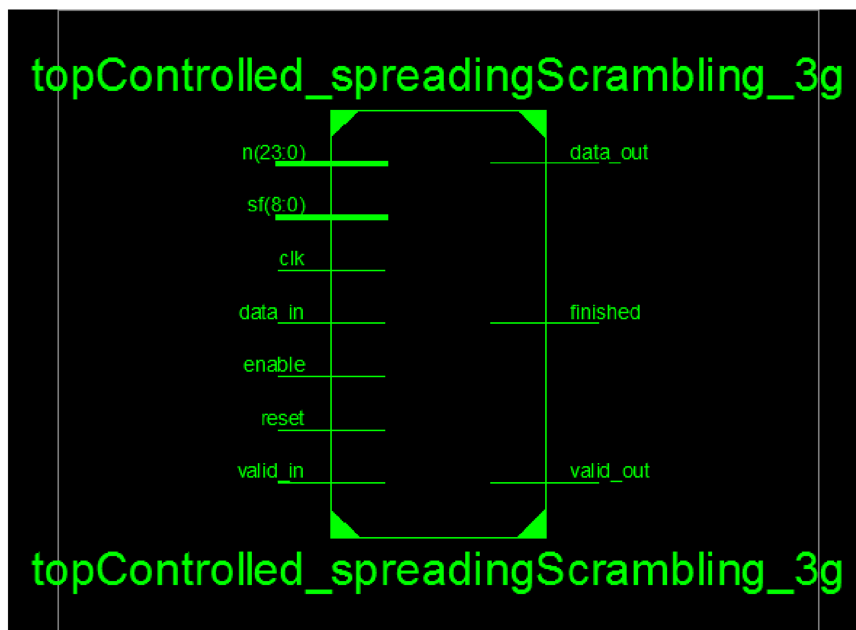


Figure 4.39: Top controlled spreading

The pins description of the controlled spreading is shown in Table 4.14

Table 4.14: pins description of spreading block

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates the current data_in is valid data
Data_out	The output data of the block
Valid_out	The signal indicates that current data_out is valid data
Finished	The signal indicates the spreading is ready for the new frame
Enable	The signal indicates the next block is ready to have data
SF	The signal indicates the number of chips per data symbol
N	The signal indicates scrambling sequence number

And the internal structure of the spreading block is shown in Figure 4.40.

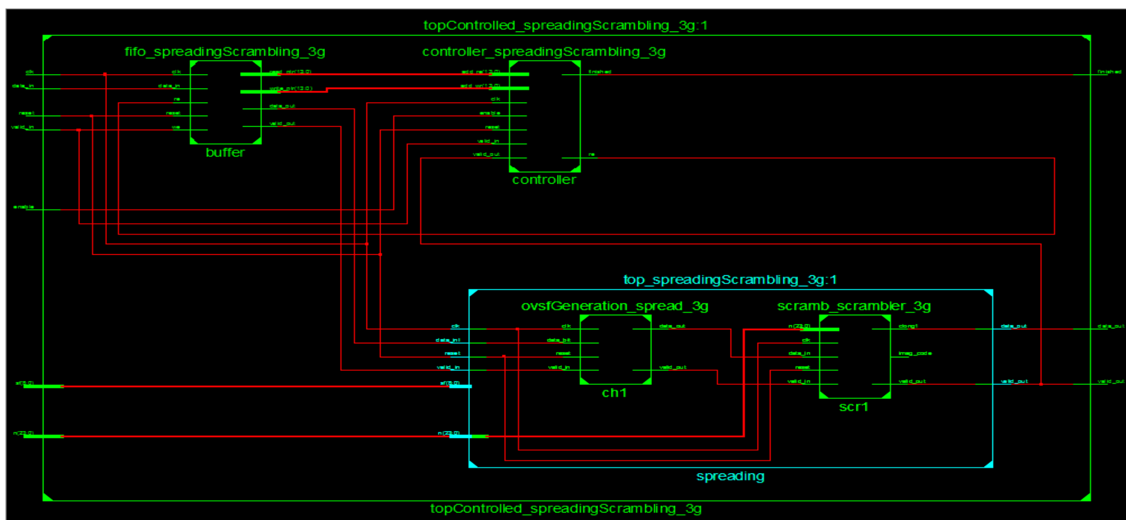


Figure 4.40: Internal structure of spreading block

4.2.11 Modulation

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 3G standard Phase Shift Keying (BPSK) modulation technique is used according to the desired data rate. The bits are mapped to complex-valued modulation symbol d=(I + j Q). In BPSK, a single bit is mapped to a complex-valued modulation symbol according to Table 4.15 [7].

Table 4.15: BPSK mapping

$b(i)$	I	Q
0	$1/\sqrt{2}$	$1/\sqrt{2}$
1	$-1/\sqrt{2}$	$-1/\sqrt{2}$

Concerning the HDL implementation, Figure 4.41 shows the interface of the mapper

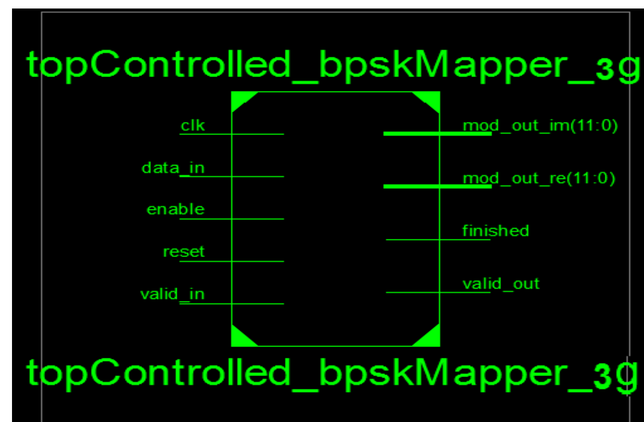


Figure 4.41: Mapper interface

As shown in the figure every symbol is represented in 12 bits – this number is determined through a simulation will be discussed later- . The pins description is in Table 4.16.

It consists of fifo to store the input bit stream, controller to control the fifo and the mapper module (top_mod_wifi) which consists of mapper module that mapps bits to the corresponding symbol following the constellation.

Table 4.16: pin description of mapper module

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data
Mod_out_Re	The modulated real part of the input
Mod_out_im	The modulated imaginary part of the input
Finished	The signal indicated that the mapper is ready for the new frame
Enable	The signal indicates that the next block is ready to have data

The internal block diagram is shown in Figure 4.42.

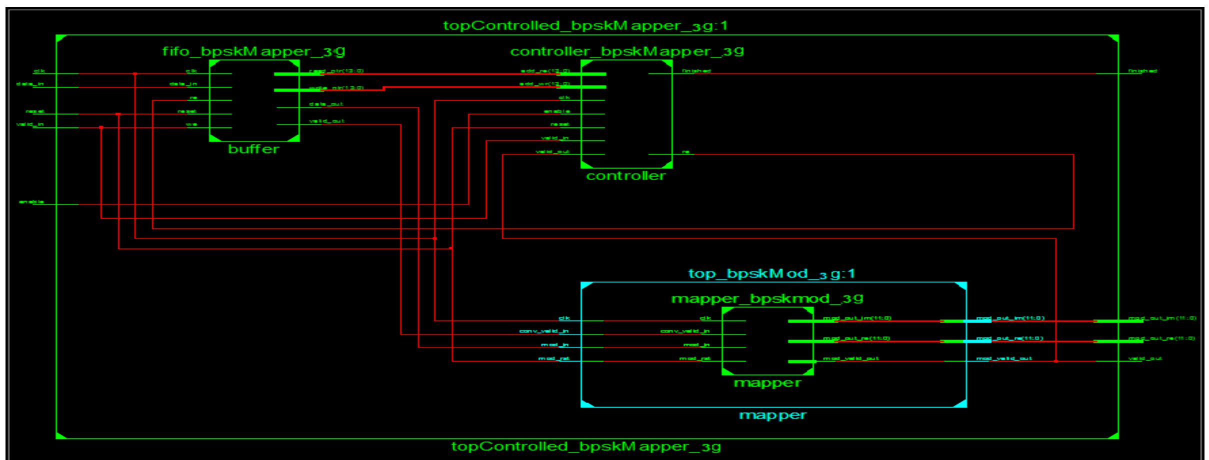


Figure 4.42: Detailed block of mapper

4.3 3G Receiver

The main target of the receiver is to retrieve the same data send before transmitter so it consists of the blocks shown in Figure 4.43

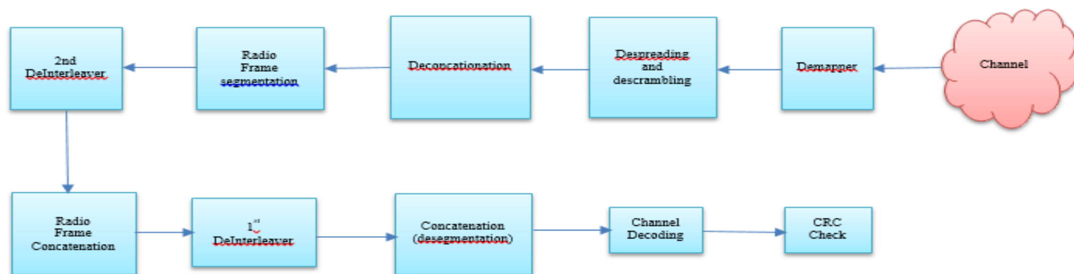


Figure 4.43: receiver blocks

4.3.1 Channel modeling

First of all we want to model the random noise of the channel as so we have to make a synthesizable HDL code which adds a random noise to the real and imaginary outputs of the transmitter. This noise depends on the SNR (signal to noise ratio) of the channel. Here we face a problem that random operation is not synthesizable

We can't implement a VHDL code to model the noise. Also we can't fix the output noise to a number of bits. Fixation advantages and process will be discussed later. In addition, random operation generate different outputs for different runs so we can't test the function and output of the block by comparing it with MATLAB code output as we made for all other blocks.

To solve all this problems we make a general MATLAB code which generate the noise and make a fixation to this output then open a .v file and print the sentences of Verilog code eg, fprintf(fid,'module top_noise(\n');. Then we assign the fixed output noise to an array of 12 bits noise of Verilog code. We open also a .m and print the code of MATLAB so the generated MATLAB and Verilog code will contain the same noise and we can compare the output. In addition, the Verilog code is now fully synthesizable and the top of it is shown in Figure 4.44 .

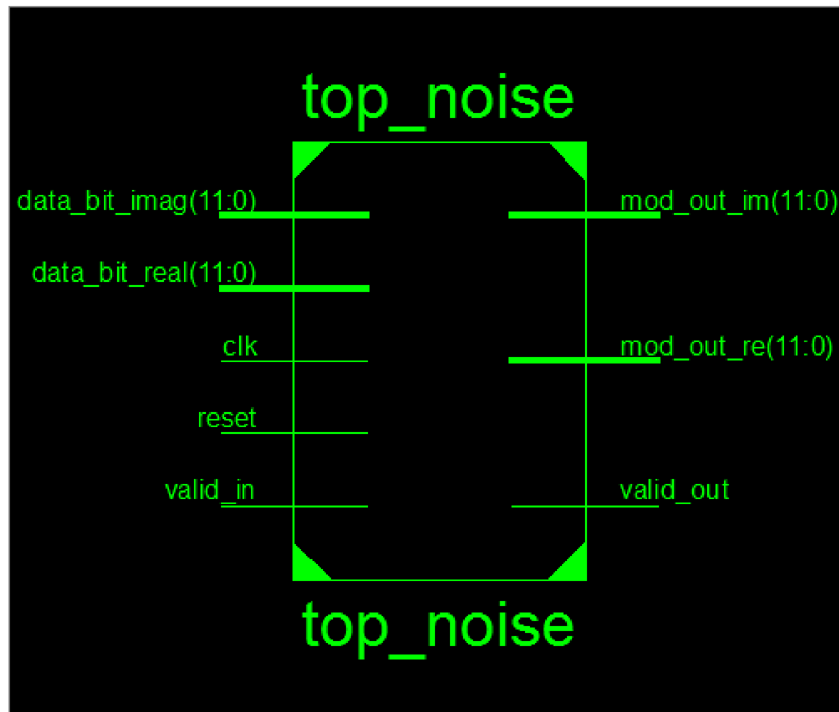


Figure 4.44: noise top block

Table 4.17: Ports description of noise

PIN	Description
Data_bit_imag	The imaginary part of the input bits
Data_bit_real	The real part of the input bits
Valid_in	The signal indicates that current data bits are valid data
Mod_out_Re	The modulated real part of the input
Mod_out_im	The modulated imaginary part of the input
Valid_out	The signal indicates that current data_out is valid data

4.3.2 Demapper

It is the first block of the receiver that will receive the real and imaginary data of the channel 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.

In 3g we have only a BPSK mapper with a constellation not on the axis as shown in Figure 4.45 . So the equation of the decision region will be $y = -x$

$b(i)$	I	Q
0	$1/\sqrt{2}$	$1/\sqrt{2}$
1	$-1/\sqrt{2}$	$-1/\sqrt{2}$

Figure 4.45: constellation of bpsk

If $y > -x$ the output will be 0 and if $y < -x$ the output will be 1

As the data can be positive or negative we have to make the two’s complement of the imaginary and compare real data with the two’s complement of the imaginary the output data shown in Figure 4.46

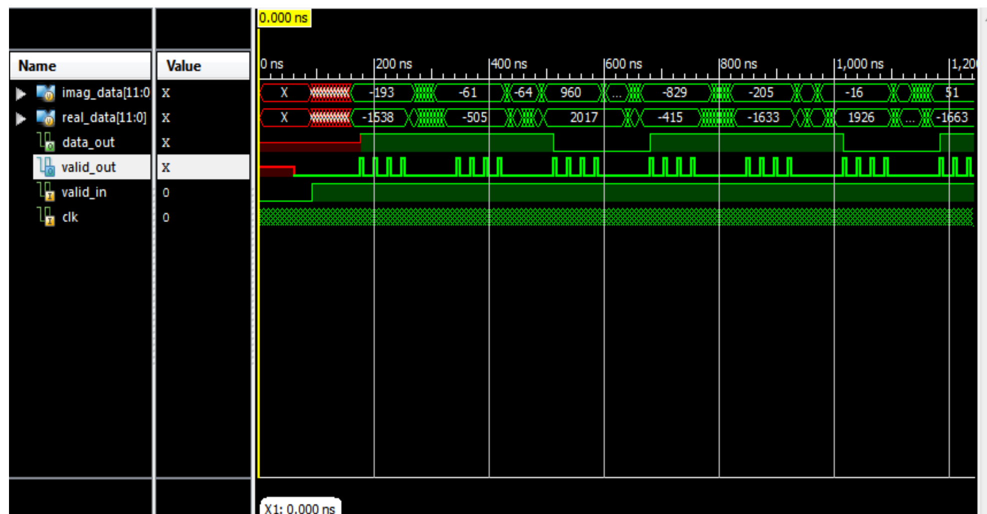


Figure 4.46: Wavefom of demapper

The pins description of the top demapper are shown in Table 4.18

Table 4.18 : Pins description of demapper

PIN	Description
Data_bit_imag	The imaginary part of the input bits
Data_bit_real	The real part of the input bits
Valid_in	The signal indicates that current data bit real and imaginary are valid data
Data_out	The output data in the form of stream of bits

Valid_out	The signal indicates that current data_out is valid data
read	The signal indicates that the block is ready to read data of the next symbol

The top demapper is shown in Figure 4.47

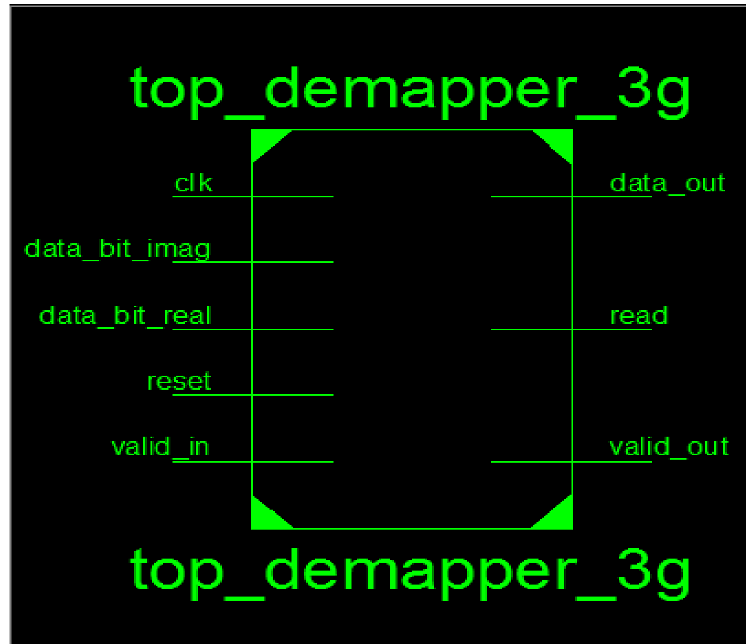


Figure 4.47: Top demapper 3g

4.3.3 Despreading block

Despreading and descrambling block consists of two operations

- Descrambling operation: one of the advantages of the scrambling codes that if we multiply the data with the scrambling data square we retrieve the same data.

  So in the descrambling process we multiply the data out from the demapper with the same scrambling code of the transmitter by using the same scrambling sequence number (n).

  The scrambling code can be a long code (a Gold code with 38400chips) or a short code (256 chips) the long code is used if the BS uses a Rake receiver and the short code is used if multiuser detector and interference cancellation receivers are used in BS. In our project we use long code.

- Despreading operation: In the despreading process we multiply the data out from the descrambler with a periodically repeated sequence of (1, 1,-1, 1) which is the same as the spreading code we repeat these sequence with a number equal k where k= SF/4 this is because we transmit only one DPDCH (Dedicated Physical Data Channel).

  Then we integrate the data by increasing a signed register count when the output of multiplying with the spreading code is one and decreasing count when the output of multiplying with the spreading code is zero. So after we receive bits equal to SF we decide if the output will be 1 or 0 and we store this value in a data out register to be out while calculating the count and decide what the next bit is.

The top of the despreading is shown in Figure 4.48

The pins description of the top despreading are shown in Table 4.19

Table 4.19: pins description of despreading

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates the current data_in is valid data
Data_out	The output data of the block
Valid_out	The signal indicates that current data_out is valid data
SF	The signal indicates the number of chips per data symbol
n	The signal indicates scrambling sequence number

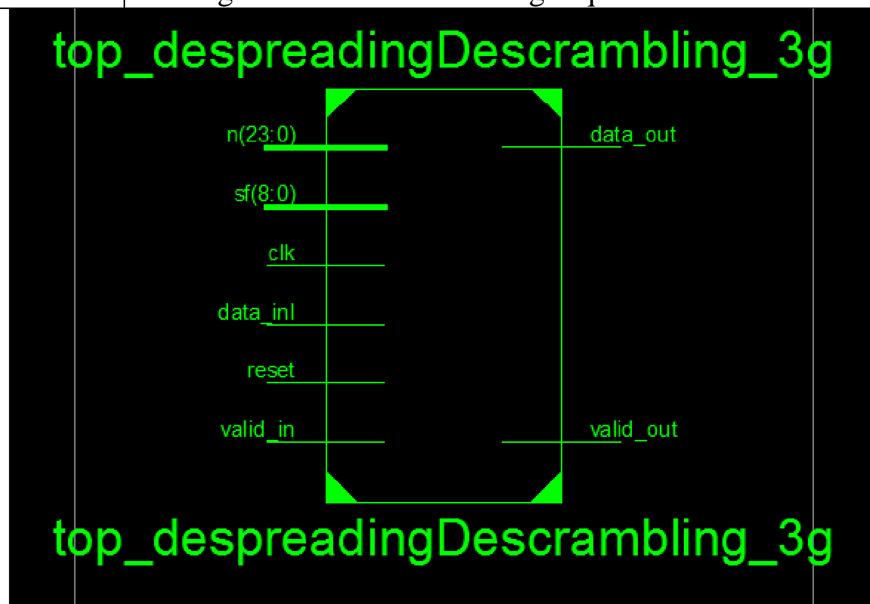


Figure 4.48: top despreading and descrambling

4.3.4 Deconcationation

Deconcationation of the bit sequence is performed if Xi>Z. This concept is the same as the segmentation block but with different z values .The code blocks after deconcationation are of the same size. The number of code blocks on TrCH 'i' is denoted by Ci. If the number of bits input to the deconcationation, Xi, is not a multiple of Ci, filler bits are added to the beginning of the first block. If turbo coding is selected and Xi < 40, filler bits are added to the beginning of the code block. The filler bits are transmitted and they are always set to 0.

To retrieve the same data before transmitter and as the data from segmentation is multiplied by the encoder rate so we calculate Z according to Table 4.20

Table 4.20: how to calculate z

Z	Description
504*2=1008	Convolutional coding and coding rate = ½
5114*2=10228	Turbo coding and coding rate = ½
504*3=1582	Convolutional coding and coding rate = 1/3
5114*3=15342	Turbo coding and coding rate = 1/3

The bits output from code block segmentation, for Ci ≠ 0, are denoted by oir1, oir2, oir3… oirki where i is the TrCH number, r is the code block number, and Ki is the number of bits per code block.

Number of code blocks: Ci = [Xi/Z]

Number of bits in each code block (applicable for Ci ≠ 0 only):

if Xi < 40 and Turbo coding is used, then

Ki = 40

else

Ki = [Xi / Ci]

end if

Number of filler bits: Yi = CiKi - Xi

```
for k = 1 to Yi                         --Insertion of filler bits

Oi1k=0

end for

for k = Yi+1 to Ki

Oik=Xi,(K-Yi)

end for

r = 2                                   -- Segmentation

while r ≤ Ci

for k = 1 to Ki

Oirk=Xi, (k+(r-1)-Ki-Yi)

end for

r = r+1

end while
```

Concerning the HDL implementation the shows the interface of the Deconcationation shown in Figure 4.49

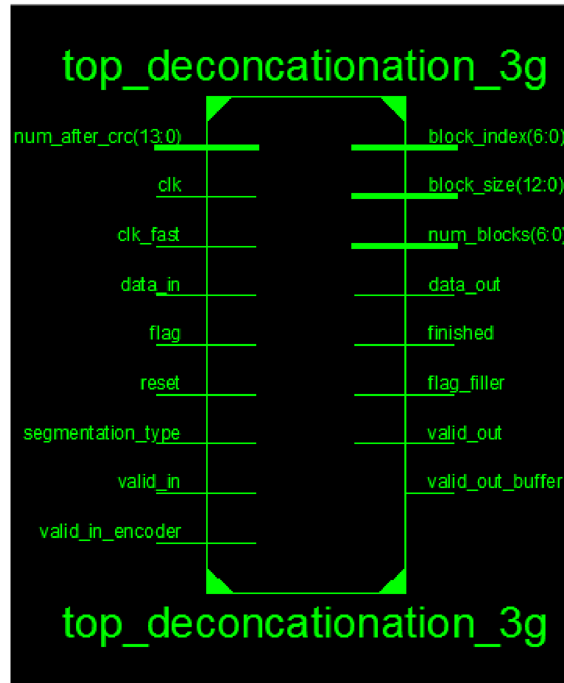


Figure 4.49: Top deconcationation

Description of pins of the block are described in Table 4.21

Table 4.21 : Description of deconationation block

PIN	Description
num_after_crc	This is input signal from despreading that indicates the total number of data bits out from the dispreading block
clk_fast	This is faster clock signal to increase the speed for the division required to generate the number of blocks produced
flag	This is an input signal from the despreading that indicates that the num_after_crc is ready to be read for the deconationation
Segmentation_Type	To differentiate between Convolutional Encoder "0" or Turbo Encoder "1"
valid_in	This signal indicates that current data_in is valid data
valid_encoder	This signal indicates that the next block is ready to have data
Block_index	Output signal that indicates the index for the block being transmitted to the deinterleaver block
Block_size	Number of bits included in each block after performing the segmentation process
Num_Blocks	Total number of blocks output from the deconcationation process
finished	The signal indicated that the deinterleaver is ready for the new frame
Flag_filler	Output signal used for the encoder such that it does not read the extra zero filler bit that remains on the bus while moving from state to another inside the code. Consequently, this reserve that valid_out signal to remain always one within the data block
valid_out	This signal indicates that current data_out is valid data

The internal block of the deconcatination is shown in

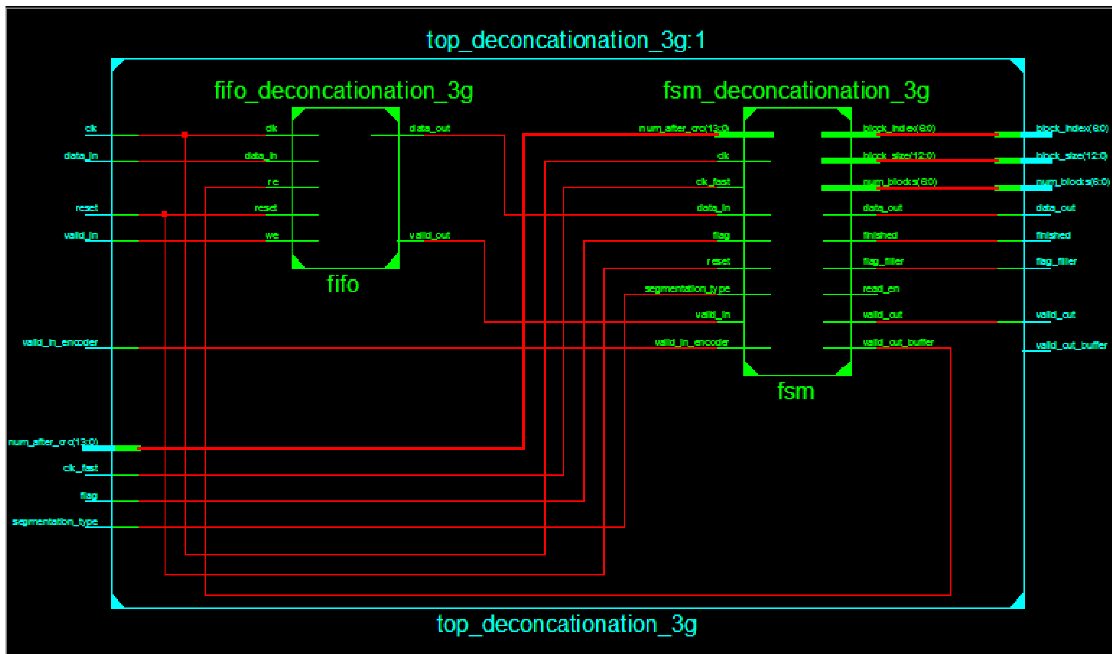


Figure 4.50: Internal design of deconcatination

4.3.5 Deinterleaver

Deinterleaver is the block which re-arranges the received bits to repeal the impact of the interleaver. The Deinterleaver's block diagram is shown Figure 4.51.

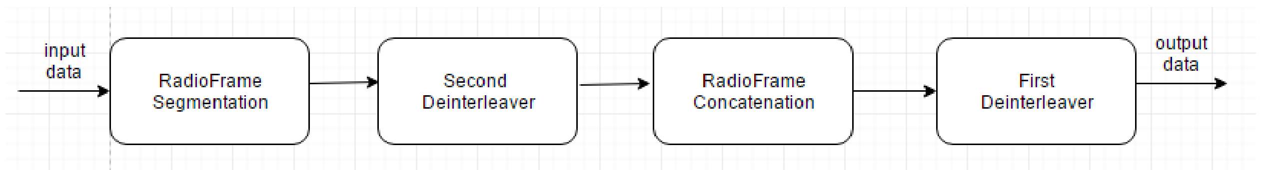


Figure 4.51 : Block diagram of deinterleaver.

Radio frame segmentation separates different frames depending on the value of "tti".

Table 4.22 shows the relationship between tti and number of frames.

Table 4.22: The relationship between tti and nmber of frames.

Tti	Number of frames
10	1
20	2
40	4
80	8

Then, every frame enters the second deinterleaver to be re-arranged. we add dummy bits to make the data multiple of 30 . Then, data with dummy bits is interleaved again using second interleaver. Finally, columns of the memory saving the interleaved data with dummy bits are arranged in the following arrangement

{Col(0),col(12),col(25),col(6),col(18),col(3),col(15),col(26),col(9),col(22),col(2),col(13),col(24),col(7),col(19),col(4),col(16),col(29),col(10),col(21),col(1),col(14),col(27),col(8),col(20),col(5),col(17),ocl(28),col(11),col(23)}

Dummy bits are thrown out, and only data bits are getting out to the Radio frame concatenation.

Radio frame concatenation adds up the bits from different frames according to tti to be fed to first deinterleaver , as first deinterleaver is interframe deinterleaver.

In first Deinterleaver, number of columns is equal to number of frames. Data is written row by row, and then column permutation is done according to Table 4.23 . Finally data is read column by column.

Table 4.23: columns arrangement in first deinterleaver.

Tti	Permutation
10	Col(1)
20	Col(1), Col(2)
40	Col(1),col(3),col(2),col(4)
80	Col(1),col(5),col(3),col(7),col(2),col(6),col(4),col(8)

All these blocks are implemented in MATLAB and tested by entering random input vector to the interleaver then to the deinterleaver and comparing the output bits to the original bits and it works properly.

4.3.6 Desegmentation Block

The desegmentation is the same as the concatenation block .The input bit sequence for the desegmentation block are the sequences e_{rk}, for r = 0,…..,C-1 and k = 0,…..,E_r-1. The output bit sequence from the code block desegmentation block is the sequence f_k for k = 0,….,G-1.

The desegmentation consists of sequentially concatenating the rate matching outputs for the different code blocks. Therefore,

Set k = 0 and r = 0

while r < C

Set j = 0

while j < E_r

$f_k = e_{rj}$

k = k +1

j = j +1

end while

r = r +1

end while

The desegmentation block interface is as shown in Figure 4.52 and the signals declaration and description is as shown in Table 4.5.

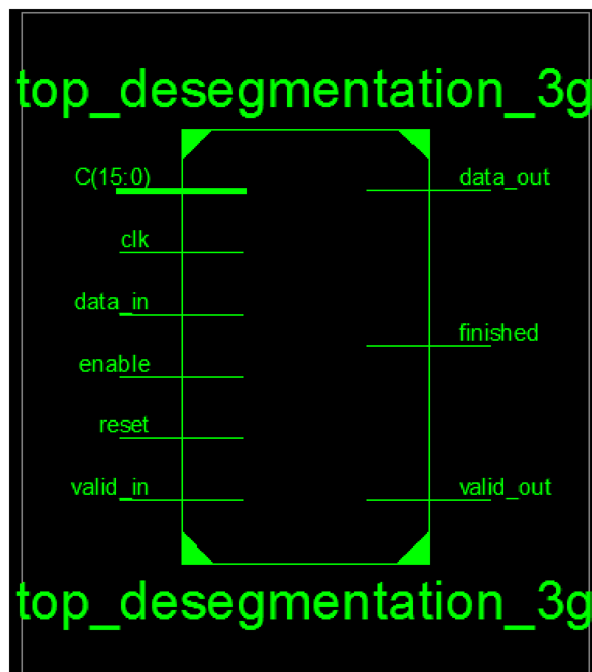


Figure 4.52: top desegmentation

Table 4.24: Desegmentation block signals declaration

PIN	Description
C	Total number of code blocks (segmentation section)
enable	This signal indicates that the next block is ready to have data
valid_in	This signal indicates that current data_in is valid data
data_in	The input bits
finished	This signal indicates that the interleaver is ready to have a new frame
valid_out	This signal indicates that current data_out is valid data
data_out	The output bits

4.3.7 CRC (Cyclic Redundancy Check) check

CRC check process is provided for error check in which the entire received block is used to calculate the CRC parity bits for each received block.

We receive the total number of bits and subtract the CRC bits number from it and generate CRC parity bits by equations shown in Table 4.25 for only $total\ number\ of\ bits - CRC\ bits$.

Finally, we compare these generated bits with the last bits received and decide out if these data was right or wrong. The data is wrong if there is any mismatch in the comparison.

Table 4.25: Equations of CRC check

CRC Mode	Equation
CRC24	gCRC24(D) = D24 + D23 + D6 + D5 + D + 1
CRC16	gCRC16(D) = D16 + D12 + D5 + 1
CRC12	gCRC12(D) = D12 + D11 + D3 + D2 + D + 1
CRC8	gCRC8(D) = D8 + D7 + D4 + D3 + D + 1

The top block of the CRC check is shown in Figure 4.53

Pins description is showed in Table 4.26

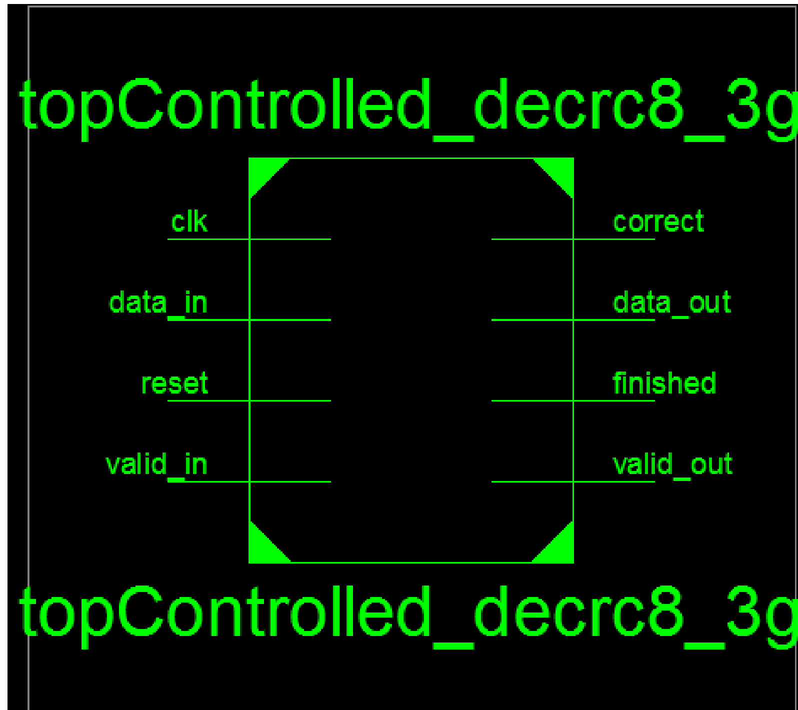


Figure 4.53 : Top CRC check

Table 4.26: Pins description of DeCRC

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data
Data_out	The output data of the block
Valid_out	The signal indicates that current data_out is valid data
Finished	The signal indicated that the CRC is ready for the new frame
Correct	The signal indicates if the data received is correct or not

And the internal structure is shown in Figure 4.54.

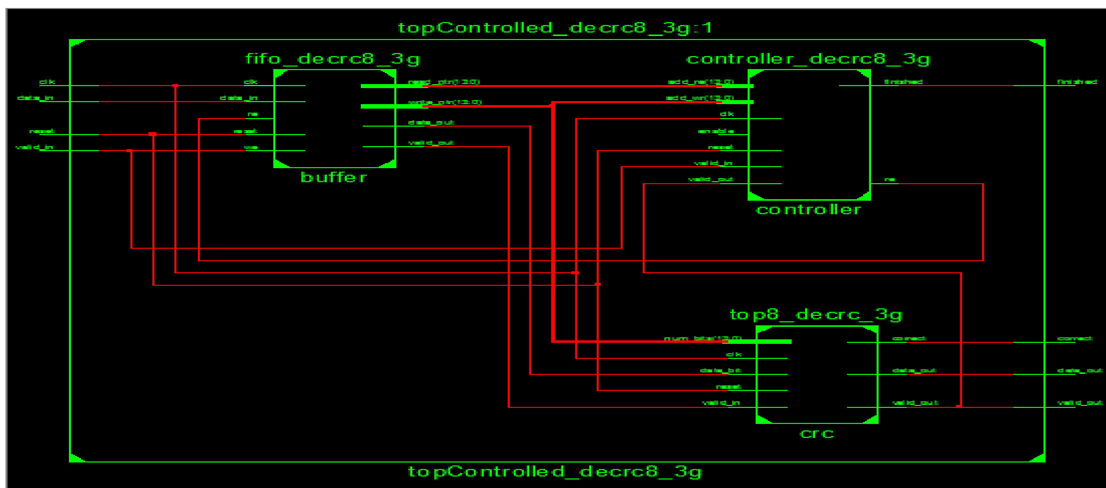


Figure 4.54: Internal structure of CRC

Chapter 5: Wi-Fi Standard Transmitter and Receiver

A brief introduction to the 802.11 and WLAN technology follows

5.1 Overview

WLAN technology and the WLAN industry date back to the mid-1980s when the Federal Communications Commission (FCC) first made the RF spectrum available to industry. During the 1980s and early 1990s, growth was relatively slow. Today, however, WLAN technology is experiencing tremendous growth. The key reason for this growth is the increased bandwidth made possible by the IEEE 802.11 standard [8].

5.2 Standard History

The IEEE initiated the 802.11 project in 1990 with a scope "to develop a Medium Access Control (MAC) and Physical Layer (PHY) specification for wireless connectivity for fixed, portable, and moving stations within an area." In 1997, IEEE first approved the 802.11 international interoperability standards. In 1999, the IEEE ratified the 802.11a and the 802.11b wireless networking communication standards. The goal was to create a standards-based technology that could span multiple physical encoding types, frequencies, and applications. The 802.11a standard uses orthogonal frequency division multiplexing (OFDM) to reduce interference. This technology uses the 5 GHz frequency spectrum and can process data at up to 54 Mbps [8].

5.3 Frequency and Data Rates

The IEEE 802.11a standard is the most widely adopted member of the 802.11 WLAN families. It operates in the licensed 5 GHz band using OFDM technology. The popular 802.11b standard operates in the unlicensed 2.4 GHz-2.5 GHz Industrial, Scientific, and Medical (ISM) frequency band using a direct sequence spread-spectrum technology. The ISM band has become popular for wireless communications because it is available worldwide.

5.4 **Physical Layer of 802.11a**

IEEE 802.11 standard specifies a 2.4 GHz operating frequency with data rates of 1 and 2 Mbps using either Direct Sequence Spread Spectrum (DSSS) or Frequency Hopping Spread Spectrum (FHSS). The IEEE 802.11a standard specifies an OFDM physical layer (PHY) that splits an information signal across 52 separate subcarriers to provide transmission of data at a rate of 6, 9, 12, 18, 24, 36, 48, or 54 Mbps. In the 802.11a IEEE standard the 6, 12, and 24 Mbps data rates are mandatory. Four of the subcarriers are pilot subcarriers that the system uses as a reference to disregard frequency or phase shifts of the signal during transmission.

In the 802.11a standard, a pseudo binary sequence is sent through the pilot subchannels to prevent the generation of spectral lines. In the 802.11a, the remaining 48 subcarriers provide separate wireless pathways for sending the information in a parallel fashion. The resulting subcarrier frequency spacing in the IEEE 802.11a standard is 0.3125 MHz (for a 20 MHz bandwidth with 64 possible subcarrier frequency slots).

Also in the 802.11a standard, the primary purpose of the OFDM PHY is to transmit Media Access Control (MAC) Protocol Data Units (MPDUs) as directed by the 802.11 MAC layer. The OFDM PHY of the 802.11a standard is divided into two elements: the Physical Layer Convergence Protocol (PLCP) and the Physical Medium Dependent (PMD) sublayers [8].

5.5 **PPDU frame structure**

The PHY Sub-layer Service Data Units (PSDU) of the 802.11a is converted to a PLCP Protocol Data Unit (PPDU). The PSDU of the 802.11a is provided with a PLCP preamble and header to create the PPDU.

Figure 5.1 shows the format for the PPDU including the OFDM PLCP preamble, OFDM PLCP header, PSDU, Tail bits, and Pad bits. The PLCP header contains the following fields: RATE, a reserved bit, LENGTH, an even parity bit, 6 Tail bits and the SERVICE field. In terms of modulation, the LENGTH, RATE, reserved bit, and parity bit (with 6 zero tail bits appended) constitute a separate single OFDM symbol, denoted SIGNAL, which is transmitted with the most robust combination of BPSK

modulation and a coding rate of R = 1/2. The SERVICE field of the PLCP header and the PSDU (with 6 zero tail bits and pad bits appended), denoted as DATA, are transmitted at the data rate described in the RATE field and may constitute multiple OFDM symbols. The tail bits in the SIGNAL symbol enable decoding of the RATE and LENGTH fields immediately after the reception of the tail bits. The RATE and LENGTH fields are required for decoding the DATA part of the packet [8].

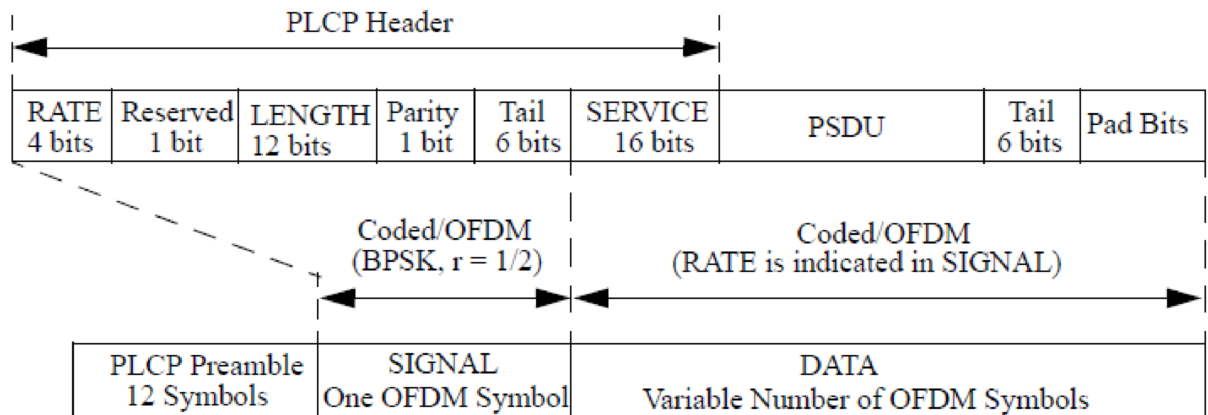


Figure 5.1: PPDU frame format

5.5.1 SIGNAL field

The OFDM training symbols shall be followed by the SIGNAL field, which contains the RATE and the LENGTH fields of the TXVECTOR (PSDU). The RATE field conveys information about the type of modulation and the coding rate as used in the rest of the packet. The encoding of the SIGNAL single OFDM symbol shall be performed with BPSK modulation of the subcarriers and using convolutional coding at R = 1/2.

The encoding procedure, which includes convolutional encoding, interleaving, modulation mapping processes, pilot insertion, and OFDM modulation, follows the steps that used for transmission of data with BPSK-OFDM modulated at coding rate 1/2. The contents of the SIGNAL field are not scrambled.

The SIGNAL field shall be composed of 24 bits, as illustrated in Figure 5.2. The four bits 0 to 3 (R1-R4) shall encode the RATE. Bit 4 shall be reserved for future use. Bits 5–16 shall encode the LENGTH field of the TXVECTOR, with the LSB being transmitted first (the length of the PSDU, this length represent the number of octets in the PSDU). A continuation is a parity bit and 6 tail bits. The tail bits are set to "zeros" to facilitate a reliable and timely detection of the RATE and LENGTH fields.

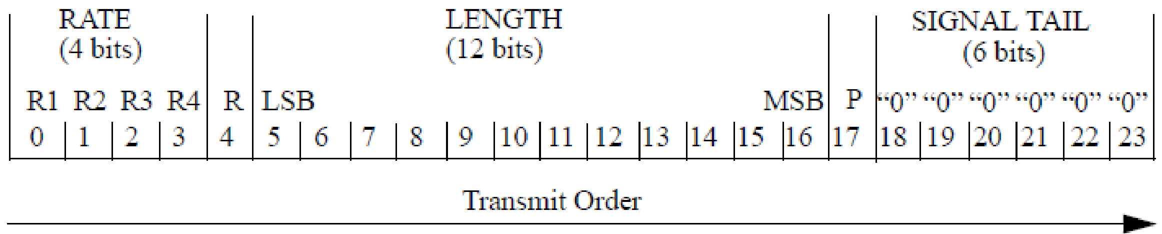


Figure 5.2: SIGNAL field bit assignment

5.5.2 RATE field

The bits R1–R4 shall be set, dependent on RATE, according to the values in Table 1.

Table 5.1: Contents of the SIGNAL field

R1-R4	Rate (Mb/s) (20 MHz channel spacing)	Rate (Mb/s) (10 MHz channel spacing)	Rate (Mb/s) (5 MHz channel spacing)
1101	6	3	1.5
1111	9	4.5	2.25
0101	12	6	3
0111	18	9	4.5
1001	24	12	6
1011	36	18	9
0001	48	24	12
0011	54	27	13.5

5.5.3 PLCP LENGTH field

The PLCP LENGTH field shall be an unsigned 12-bit integer that indicates the number of octets in the PSDU that the MAC is currently requesting the PHY to transmit. This value is used by the PHY to determine the number of octet transfers that will occur between the MAC and the PHY after receiving a request to start transmission.

5.5.4 Parity (P), Reserved (R), and SIGNAL TAIL fields

Fourth bit is reserved. It shall be set to 0 on transmit and ignored on receive. The seventh (17th) bit shall be a positive parity (even parity) bit for bits 0–16. The bits 18–23 constitute the SIGNAL TAIL field, and all 6 bits shall be set to 0.

5.5.5 DATA field

The DATA field contains the SERVICE field, the PSDU, the TAIL bits, and the PAD bits, if needed .All bits in the DATA field are scrambled.

This field contains the PSDU. The first 16 bits (7 null bits used for the scrambler initialization and 9 null bits reserved for future use) for the SERVICE field. A continuation is the PSDU. A continuation is a 6 tail bits and pad bits.

The tail bits containing 0s are appended to the PPDU to ensure that the convolutional encoder is brought back to zero state and the pad bits are used as guards for the PPDU frame.

5.5.6 SERVICE field

The IEEE 802.11 SERVICE field has 16 bits, which shall be denoted as bits 0–15. The bit 0 shall be transmitted first in time. The bits from 0–6 of the SERVICE field, which are transmitted first, are set to 0s and are used to synchronize the descrambler in the receiver. The remaining 9 bits (7–15) of the SERVICE field shall be reserved for future use. All reserved bits shall be set to 0. Refer to Figure 5.3.

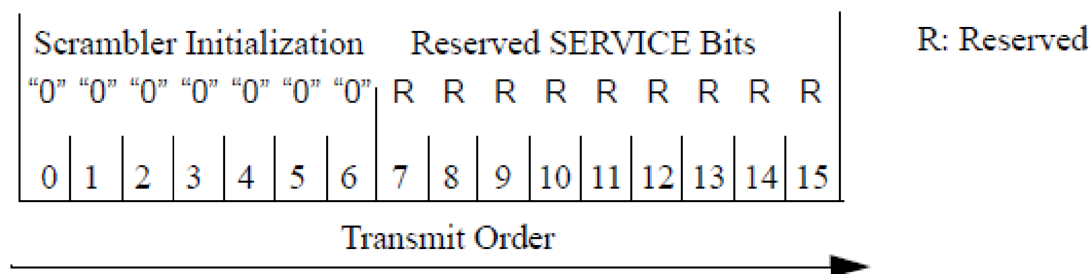


Figure 5.3: SERVICE field bit assignment

5.5.7 PPDU TAIL field

The PPDU TAIL field shall be six bits of 0, which are required to return the convolutional encoder to the zero state. This procedure improves the error probability of the convolutional decoder, which relies on future bits when decoding and which may be not be available past the end of the message. The PLCP tail bit field shall be produced by replacing six scrambled zero bits following the message end with six non-scrambled zero bits.

5.5.8 Pad bits (PAD)

The number of bits in the DATA field shall be a multiple of N_{CBPS}, the number of coded bits in an OFDM symbol (48, 96, 192, or 288 bits). To achieve that, the length of the message is extended so that it becomes a multiple of N_{DBPS}, the number of data bits per OFDM symbol.

At least 6 bits are appended to the message, in order to accommodate the TAIL bits. The number of OFDM symbols, N_{SYM}; the number of bits in the DATA field, N_{DATA}; and the number of pad bits, N_{PADS}, are computed from the length of the PSDU (LENGTH) as follows:

$$N_{SYM} = \text{Ceiling}((16 + 8 * \text{LENGTH} + 6)/N_{DBPS}$$

$$N_{DATA} = N_{SYM} * N_{DBPS}$$

$$N_{PAD} = N_{DATA} - (16 + 8 * LENGTH + 6)$$

5.5.9 PLCP preamble:

This field is used to acquire the incoming OFDM signal and train and synchronize the demodulator. The PLCP preamble is BPSK-OFDM modulated at 6 Mbps using convolutional encoding rate R=1/2.

5.5.10 Frame Summary points

- The PLCP header field is produced from the RATE, LENGTH, and SERVICE fields of the TXVECTOR by filling the appropriate bit fields.
- The PPDU SIGNAL field is the PLCP Header but without the SERVICE field.
- The contents of the SIGNAL field and the 6 tail bits in the DATA field are not scrambled but follow the same steps for convolutional encoding, interleaving, BPSK modulation, pilot insertion, IFFT, and pre-pending a GI.

5.6 802.11a Transmitter PHY Block Diagram

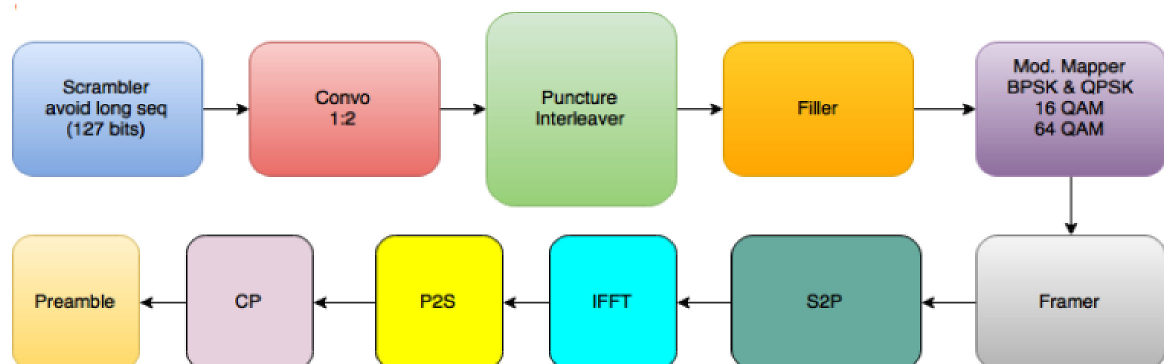


Figure 5.4: Wi-Fi Tx Block Diagram

5.6.1 Scrambler

Scrambler is used to randomize the service, PSDU, pad and data patterns to prevent long sequences of 1s or 0s to keep synchronization. The contents of the SIGNAL field and the 6 tail bits in the DATA field are not scrambled [8] [9]. The frame synchronous scrambler uses the generator polynomial S(x) as follows:

$$S(x) = x^7 + x^4 + 1$$

This generator polynomial S(x) can be represented as shown in Figure 5.5

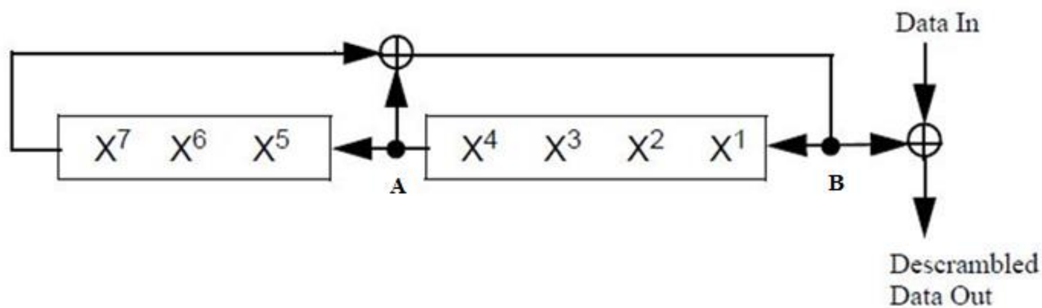
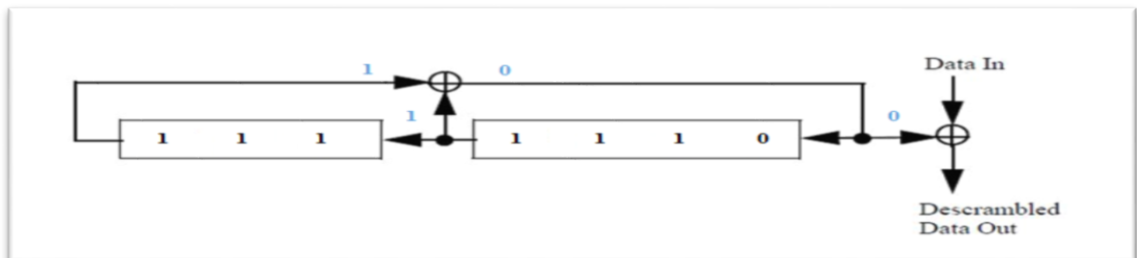
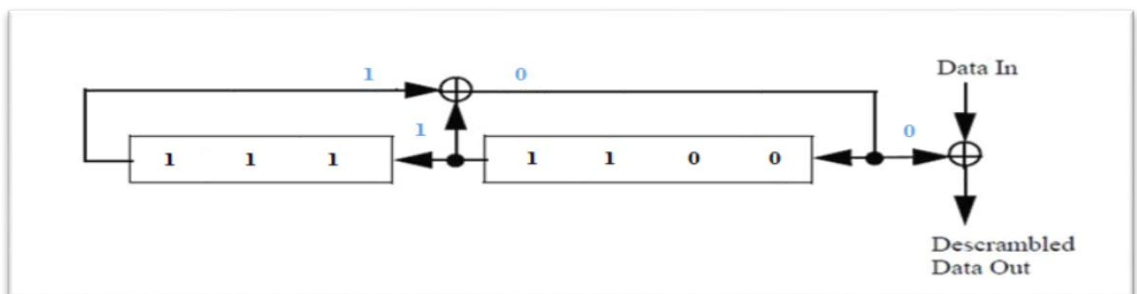


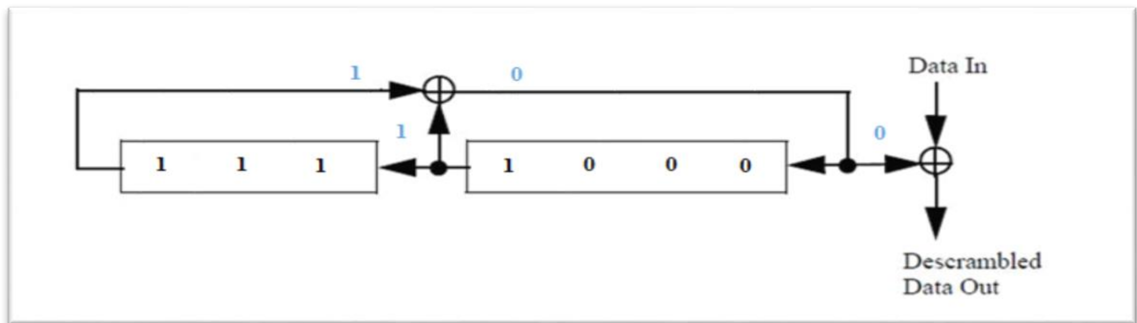
Figure 5.5: Data Scrambler

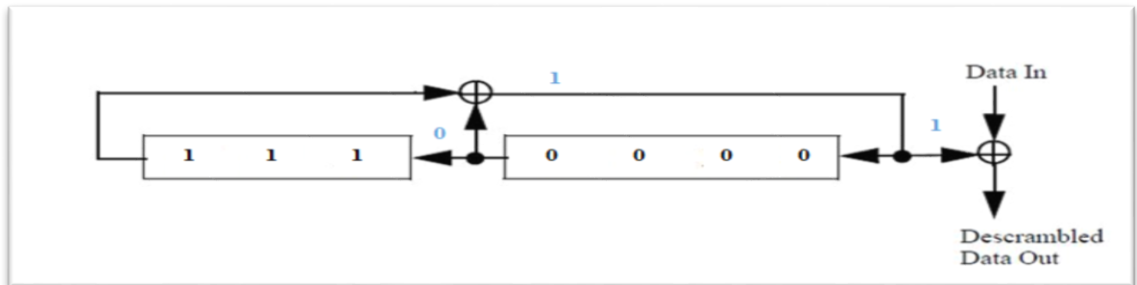
According to the initial state the scrambler will generate 127 bit sequence then it will return to its initial state.

For example: Assume that the initial state of the scrambler is all ones (1111111) but it is not a fixed initial.









And so on…..

We can say that all the bits transmitted by the 802.11a PMD in the data portion are scrambled using a frame synchronous 127 bits sequence generator. Because there is a sequence of bits generated at node B that shown in Figure 5.5 and this sequence is repeated after 127bit.

In the previous example (all ones initial state) the 127-bit sequence generated repeatedly by the scrambler is 00001110 11110010 11001001 00000010 00100110 00101110 10110110 00001100 11010100 11100111 10110100 00101010 11111010 01010001 10111000 1111111 and this sequence change when the initial state change. The same scrambler is used to scramble the transmitted data and descramble the received data

The seven LSBs of the SERVICE field will be set to all zeros prior to scrambling to enable estimation of the initial state of the scrambler in the receiver. The contents of the SIGNAL field of the 802.11a are not scrambled.

Regarding the HDL implementation Figure 5.6 shows the interface of the Scrambler and signals declaration and definition is as shown in Table 5.2

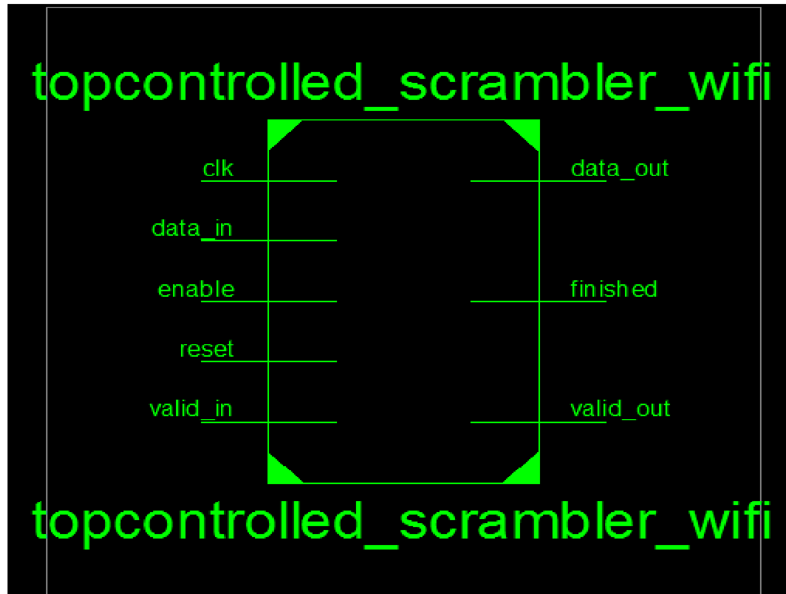


Figure 5.6: Top Controlled Scrambler Interface

Table 5.2: Scrambler Signals Declaration

PIN	Description
Data_in	The input bits
Valid_in	This signal indicates that current data_in is valid data
enable	This signal indicates that the next block is ready to have data
finished	This signal indicates that the scrambler is ready to have a new frame
Data_Out	The output bits
Valid_out	This signal indicates that current data_out is valid data

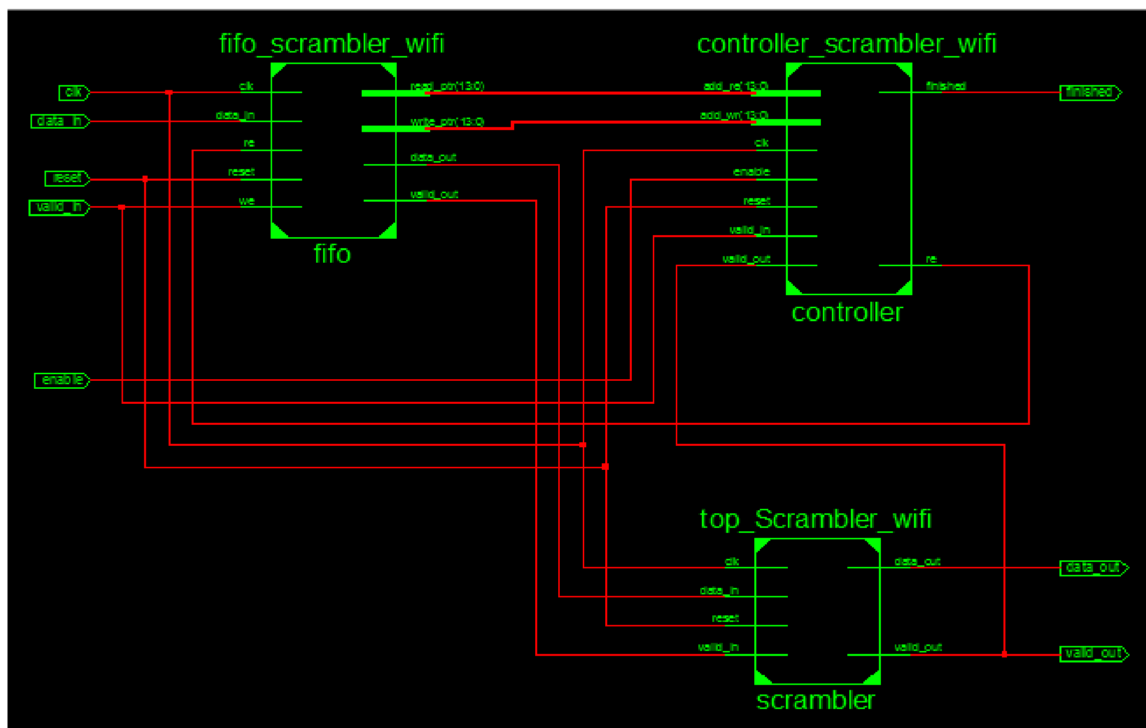


Figure 5.7: Internal Signals for Scrambler

5.6.2 Convolutional Encoder

The DATA field, composed of SERVICE, PSDU, tail, and pad parts, 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, $g_0 = 133_8$ and $g_1 = 171_8$, of rate R = 1/2, as shown in Figure 5.8. 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.” Puncturing is a procedure for omitting some of the encoded bits in the transmitter (thus reducing the number of transmitted bits and increasing the coding rate) and inserting a dummy “zero” metric into the convolutional decoder on the receive side in place of the omitted bits. The encoder is followed by parallel to serial block to transmit the encoded bits to the puncture [10].

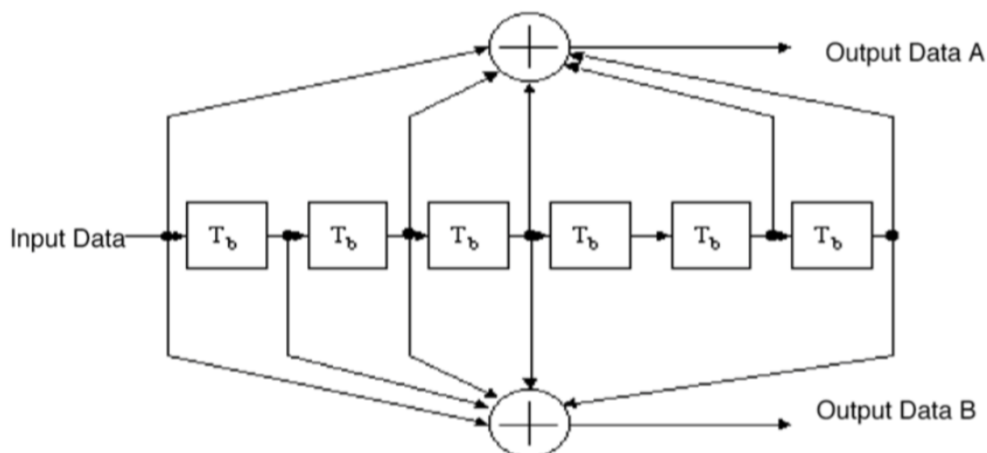


Figure 5.8: Convolutional Encoder (K=7)

Schematic (shown in the following Figure 5.9 the top module of the 3g convolutional encoder rate half)

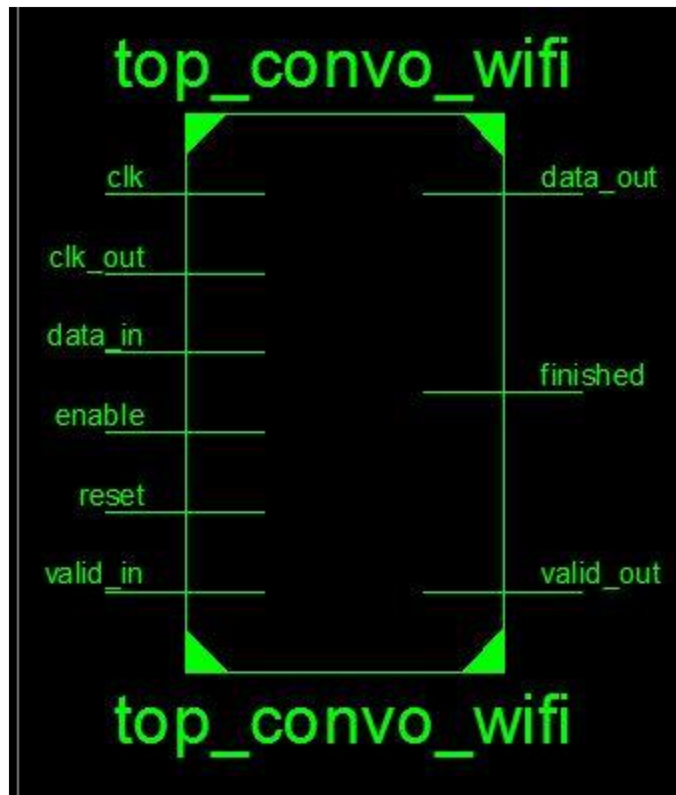


Figure 5.9: Schematic of the convolutional encoder

Table 5.3: Convolution Encoder Signals Declaration

PIN	Description
Clk_out	Clock of the serial output
Data_in	Data in for the convolutional encoder
Enable	Working enable for the encoder
Reset	Reset encoder registers by inserting Zeros
Valid_in	Valid in to consider the input
Data_out	Encoder input
Finished	Signal indicates the block is ready for the new frame
Valid_out	Valid out signal to the next block

Detailed a block diagram for the blocks that shown internal construction of the WIFI convolutional encoder in Figure 5.10.

5.6.3 Puncture

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.

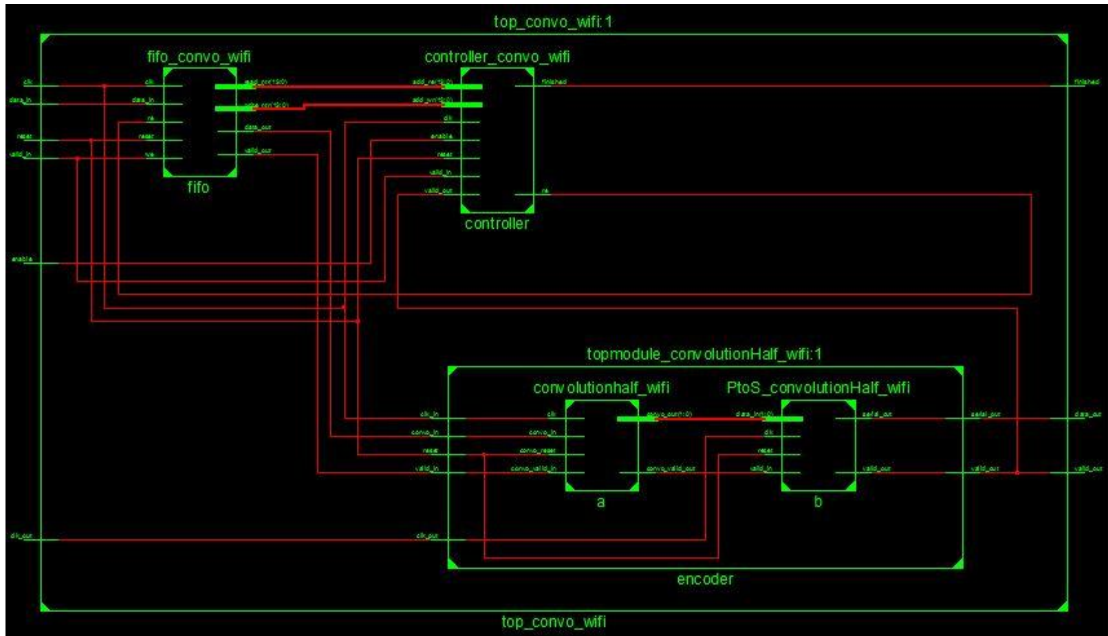


Figure 5.10: Internal block diagram of the convolutional encoder

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. Puncturing was first discovering by Cain, Clark, and Geist, and subsequently the technique was improved by Hagenauer.

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 [8].

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.

If we used 1/2 convolutional encoder and we have 6 data bits, we will send 12 bits on the channel, puncturing will remove some of the bits and adds some data bits from the next frame as shown in Figure 5.11

Puncture input : 1 0 0 1 1 0 1 0 1 1 1 0

Puncture vector: 1 1 0 1 1 0 1 1 0 1 1 0

Puncture output: 1 0 x 1 1 x 1 0 x 1 1 x

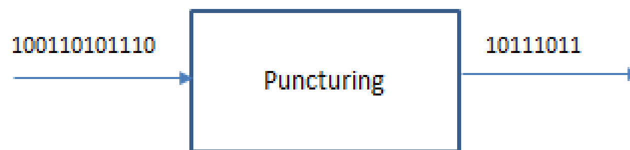


Figure 5.11: Puncturing example 1

There are two types of punctures in wifi standard: (2/3) and (3/4) according to the data rate as shown in Table 5.4.

Table 5.4: Data Rates and Puncture types

Rate (Mbps)	Code rate	Rate (Mbps)	Code rate
6	½	24	½
9	¾	36	¾
12	½	48	$^2/_3$
18	¾	54	¾

In case of total rate (½) there is no puncture and in case of total rate ($^2/_3$) the rate of puncture should be $\frac{4}{3}$ so $\frac{1}{2}*\frac{4}{3}=\frac{2}{3}$ $={}^2/_3$ but in case of total rate (¾) the rate of puncture should be $\frac{6}{4}$ so $\frac{1}{2}*\frac{6}{4}=\frac{3}{4}$. Puncture vector for different rates are show in Figure 5.12

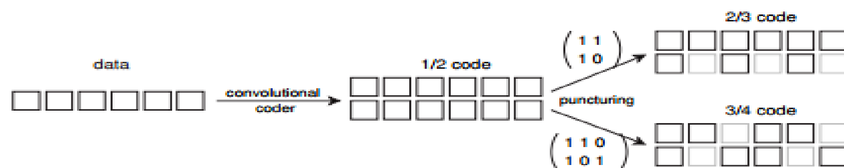


Figure 5.12: Puncture vector

Concerning the HDL implementation, Figure 5.13 shows the interface

The pins description is in the following Table 5.5.

The internal block diagram is shown in Figure 5.14.

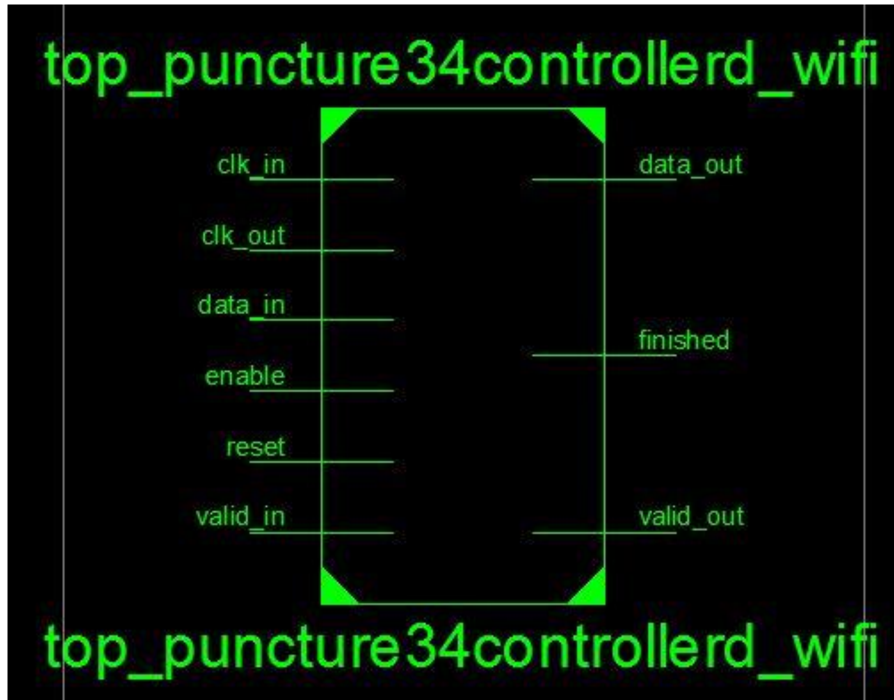


Figure 5.13: Puncture interface

Table 5.5: Puncture Pin description

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data
Clk_in	Clk of data_in
Clk_out	Clk of data_out
Finished	The signal indicated that the block is ready for the new frame
Enable	The signal indicates that the next block is ready to have data

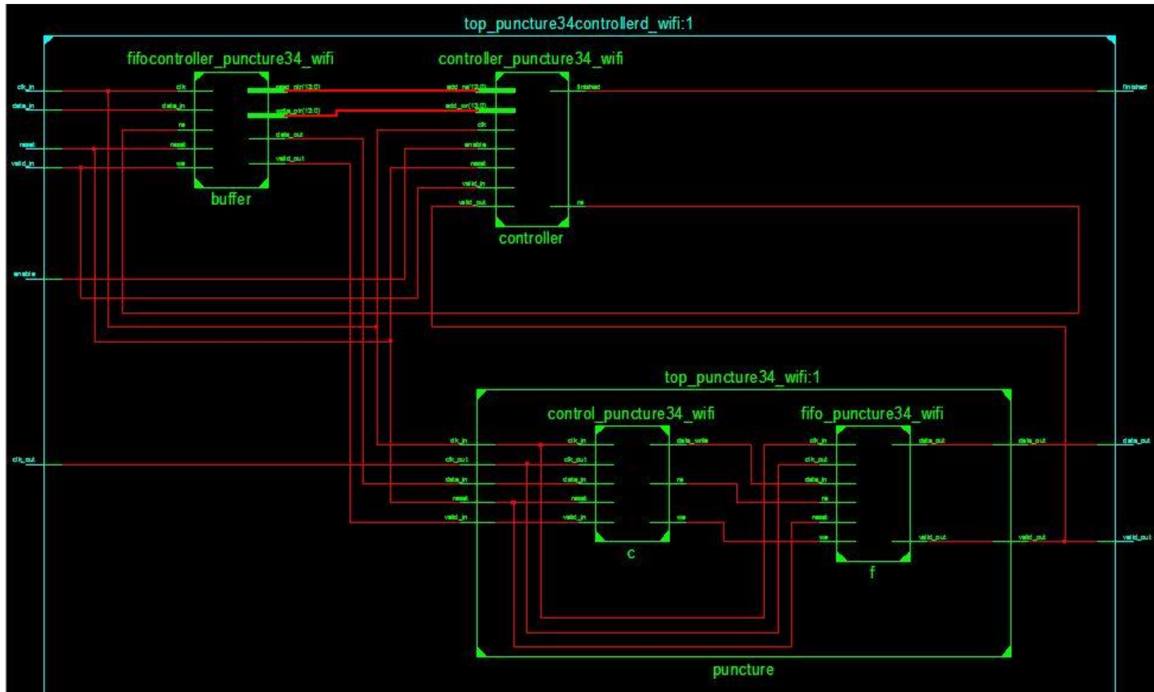


Figure 5.14: Puncture block diagram

Implementation of puncture depend on fifo which we control the write enable of it according to puncture vector and the signal field (first 48 coded bits) are not punctured then read from fifo by order , Figure 5.15 shows the timing diagram of write enable for puncture ¾.

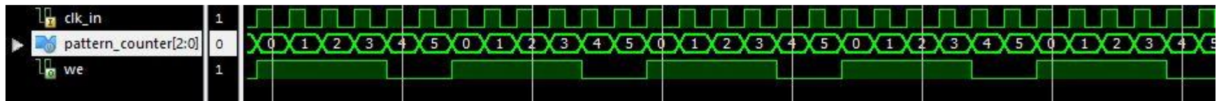


Figure 5.15: Puncture 3/4 timing diagram - write enable

The ratio between input clock and output clock depends on the rate of puncture; Figure 5.16 shows the timing diagram of write enable for puncture ¾.

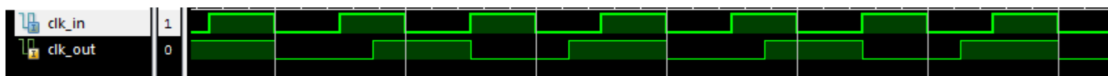


Figure 5.16: Puncture 3/4 timing diagram - clock ratio

5.6.4 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, N_{CBPS}. 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.

The index of the coded bit before the first permutation shall be denoted by k; i shall be the index after the first and before the second permutation; and j shall be the index after the second permutation, just prior to modulation mapping [8] [9]. The first permutation is defined by the rule:

$$i = \left(\frac{N_{CBPS}}{16}\right) * (k \; mod \; 16) + Floor\left(\frac{k}{16}\right) \qquad k = 0,1,\ldots\ldots, N_{CBPS} - 1$$

The function Floor (.) denotes the largest integer not exceeding the parameter.

The second permutation is defined by the rule:

$$j = s * Floor\left(\frac{i}{s}\right) + \left(i + \; N_{CBPS} - Floor\left(16 * \frac{i}{N_{CBPS}}\right)\right) mod \; s$$

$$i = 0,1,\ldots\ldots, N_{CBPS} - 1$$

The value of s is determined by the number of coded bits per subcarrier, N_{BPSC}, according to:

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

The deinterleaver, which performs the inverse relation, is also defined by two permutations. Interleaver block simulation

The interleaver block interface is as shown in Figure 5.17, the signals declaration and description is as shown in Table 5.6 and the block simulation is as shwon in Figure 5.18.

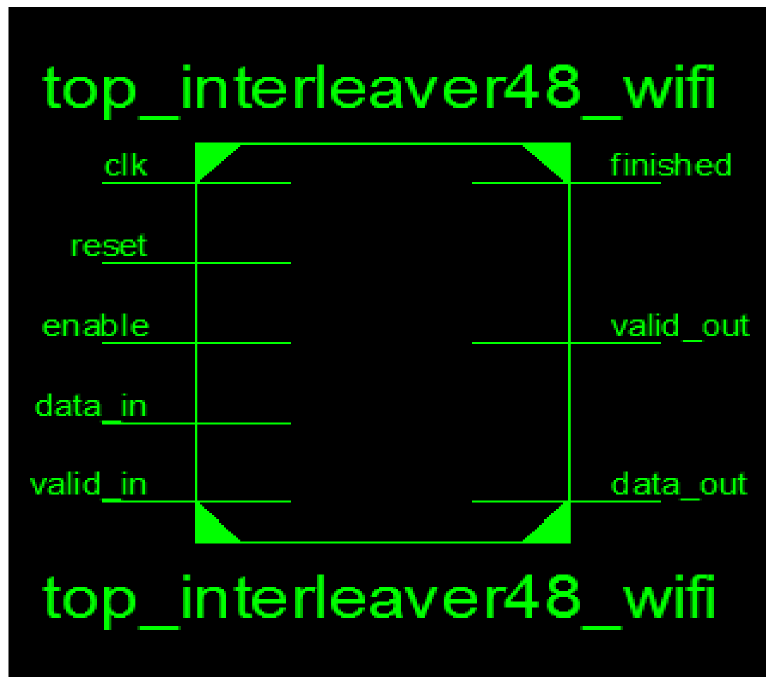


Figure 5.17: Interleaver block interface

Table 5.6: Interleaver block signals declaration

PIN	Description
enable	This signal indicates that the next block is ready to have data
valid_in	This signal indicates that current data_in is valid data
data_in	The input bits
finished	This signal indicates that the interleaver is ready to have a new frame
valid_out	This signal indicates that current data_out is valid data
data_out	The output bits

The implementation of this block is based on a RAM, this RAM used to store all the input data to the interleaver block as shown in Figure 5.19 then we read from this RAM but out of order according to the index that calculated from the interleaving equations as shown in Figure 5.20

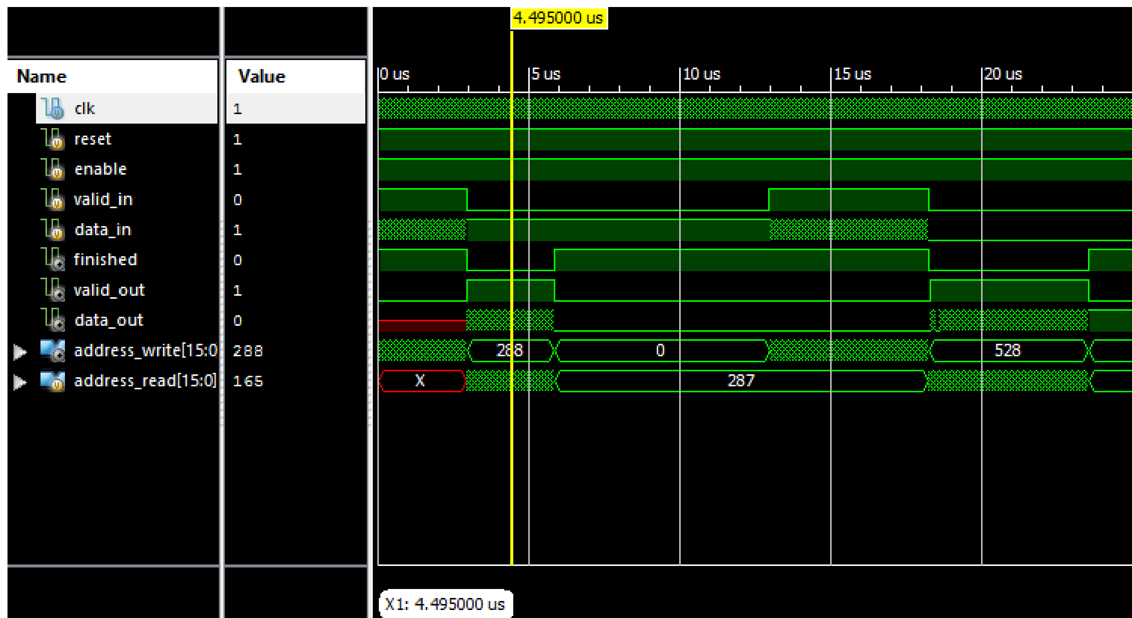


Figure 5.18: Interleaver block simulation

Figure 5.19: Writing the input data in the RAM

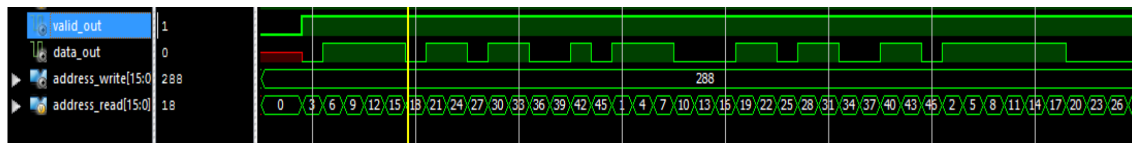


Figure 5.20: Reading the data from the RAM

The interleaving equation is different according to N_{CBPS} and N_{BPSC} values, these values are as shown in Table 5.7 So finally we have four interleavers:

$N_{BPSC} = 1$, $N_{CBPS} = 48$

$N_{BPSC} = 2$, $N_{CBPS} = 96$

$N_{BPSC} = 4$, $N_{CBPS} = 192$

$N_{BPSC} = 6$, $N_{CBPS} = 288$

Table 5.7: Modulation-dependent parameters

Modulation	Coding rate (R)	Coded bits per subcarrier (N_{BPSC})	Coded bits per OFDM symbol (N_{CBPS})	Data bits per OFDM symbol (N_{DBPS})	Data rate (Mb/s) (20 MHz channel spacing)	Data rate (Mb/s) (10 MHz channel spacing)	Data rate (Mb/s) (5 MHz channel spacing)
BPSK	1/2	1	48	24	6	3	1.5
BPSK	3/4	1	48	36	9	4.5	2.25
QPSK	1/2	2	96	48	12	6	3
QPSK	3/4	2	96	72	18	9	4.5
16-QAM	1/2	4	192	96	24	12	6
16-QAM	3/4	4	192	144	36	18	9
64-QAM	2/3	6	288	192	48	24	12
64-QAM	3/4	6	288	216	54	27	13.5

5.6.5 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 [8].

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: $\mathbf{d=(I + j\ Q)* K_{mod}}$ where K_{mod} is the normalization factor and is used in to achieve the same average power for all mappings. It depends on the base modulation mode as shown in Table 5.8.

Table 5.8: Normalization factor for all modulation modes.

Modulation	Kmod
BPSK	1
QPSK	$1/\sqrt{2}$
16-QAM	$1/\sqrt{10}$
64-QAM	$1/\sqrt{40}$

Every modulation mode has a modulation specified in the standard as shown in Figure 5.21.

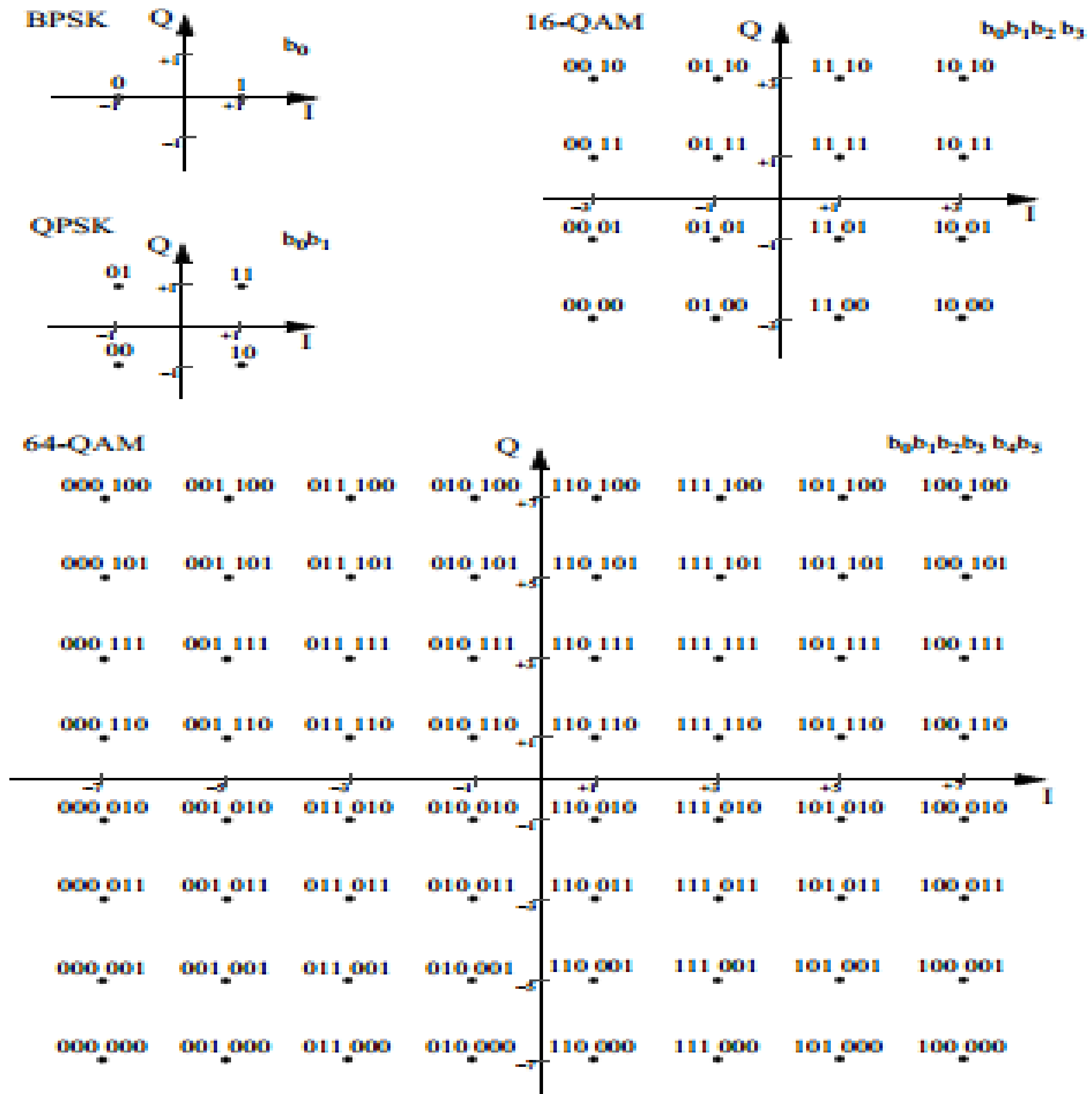


Figure 5.21: Modulation constellations for BPSK, QPSK, 16-QAM, and 64-QAM.

Concerning the HDL implementation, Figure 5.22 shows the interface of the mapper.

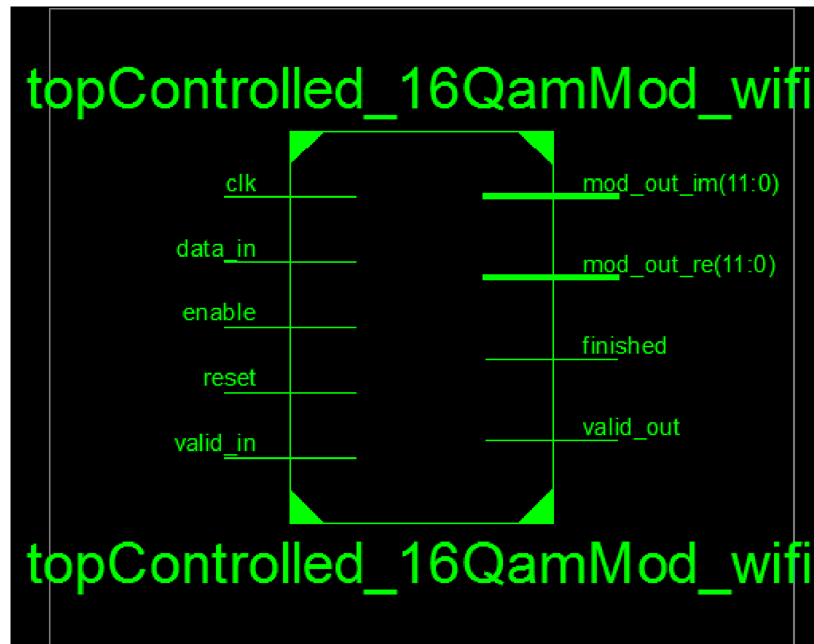


Figure 5.22: Wi-Fi mapper interface.

As shown in the figure every symbol is represented in 12 bits – this number is determined through a simulation will be discussed later- the pins description is in Table 5.9.

Table 5.9: Pin description of wifi mapper.

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data
Mod_out_Re	The modulated real part of the input
Mod_out_im	The modulated imaginary part of the input
finished	The signal indicated that the mapper is ready for the new frame
enable	The signal indicates that the next block is ready to have data

The internal block diagram is shown in Figure 5.23**Error! Reference source not found.**.

It consists of fifo to store the input bit stream, controller to control the fifo and the mapper module (top_mod_wifi) which consists of serial to parallel inverter and the mapper module that mapps bits to the corresponding symbol following the constellation.

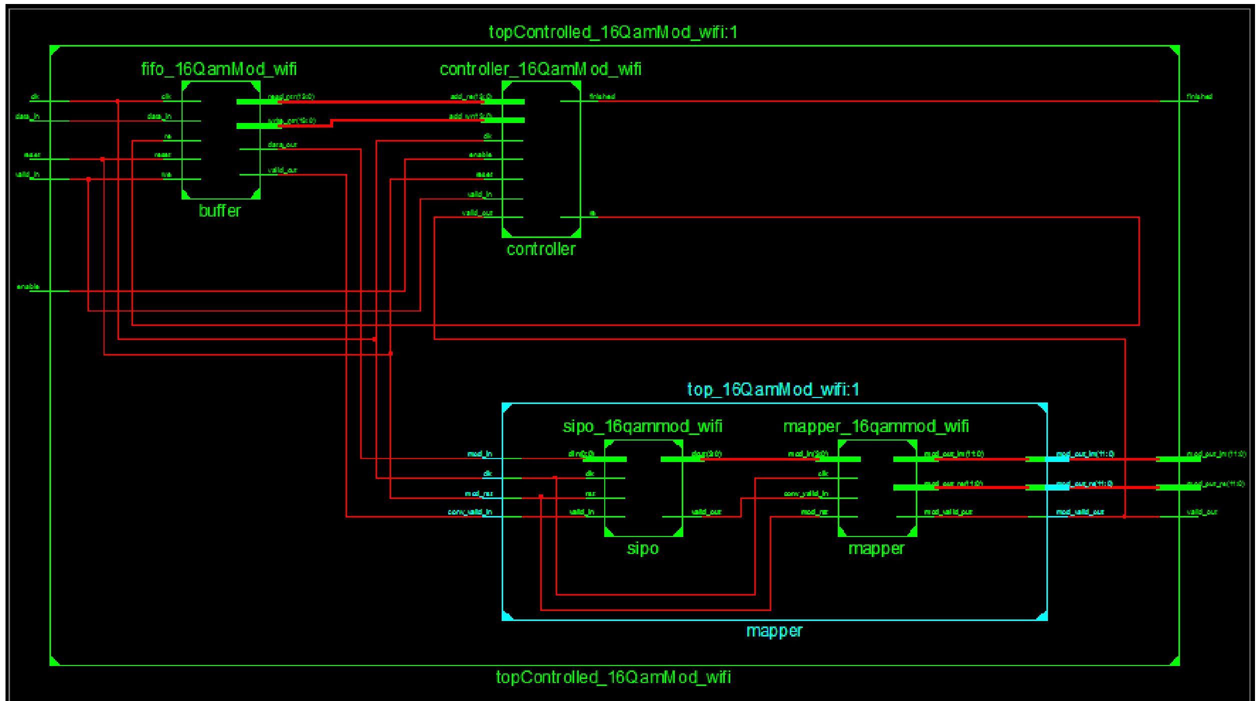


Figure 5.23:Detailed block diagram of Wi-Fi mapper module.

5.6.6 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 5.24 [9], 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.

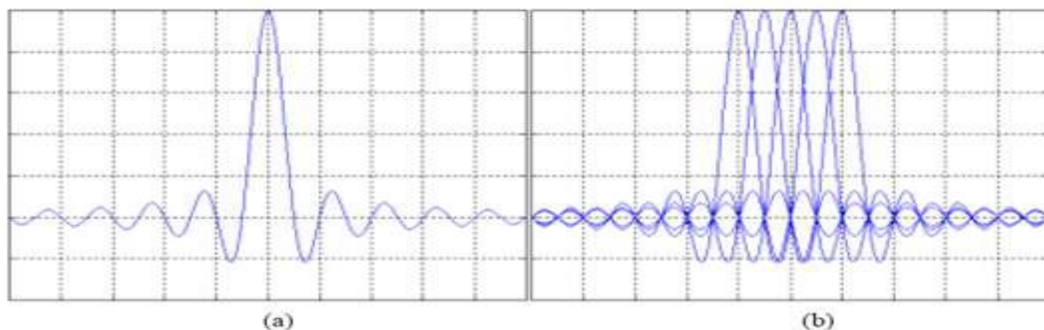


Figure 5.24:Spectrum of a single subcarrier of the OFDM signal (a), Spectrum of the OFDM signal (b)

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 802.11-2012, 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 us symbol and 0.8 us for the cyclic prefix, the timing of the OFDM frame is as shown in Figure 5.25. [8]

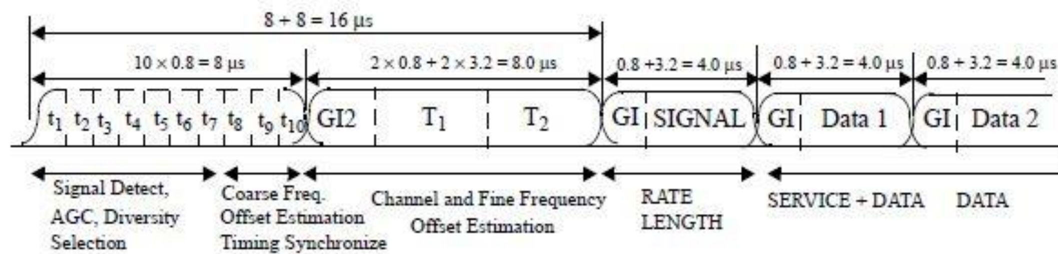


Figure 5.25: OFDM training structure

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 5.26. [8]

$$M(k) = \begin{cases} k - 26 & 0 \le k \le 4 \\ k - 25 & 5 \le k \le 17 \\ k - 24 & 18 \le k \le 23 \\ k - 23 & 24 \le k \le 29 \\ k - 22 & 30 \le k \le 42 \\ k - 21 & 43 \le k \le 47 \end{cases}$$

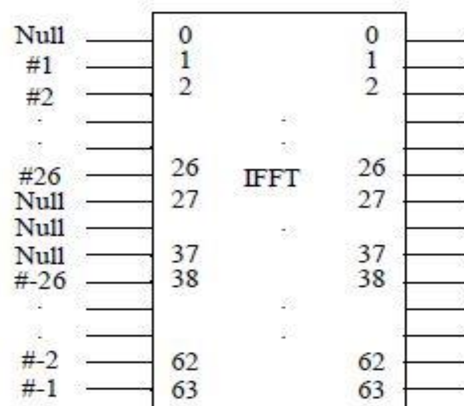


Figure 5.26: inputs and outputs of the IFFT [8]

The contribution of the pilot subcarriers for the nth OFDM symbol is produced by inverse Fourier transform of sequence P, given by

$P_{-26,26}$= {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, –1, 0, 0, 0, 0, 0} [8]

The polarity of the pilot subcarriers is controlled by the sequence, pn, which is a cyclic extension of the 127 elements sequence and is given by

$p_{0..126v}$ = {1,1,1,1,–1,–1,–1,1,–1,–1,–1,–1,1,1,–1,1, –1,–1,1,1, –1,1,1,–1, 1,1,1,1,1,1,–1,1,1,1,–1,1, 1,–1,–1,1, 1,1,–1,1,–1,–1,–1,1, –1,1,–1,–1, 1,–1,–1,1, 1,1,1,1, –1,–1,1,1,–1,–1,1,–1,1,–1,1,1,–1,–1,–1,1, 1,–1,–1,–1, –1,1,–1,–1, 1,–1,1,1, 1,1,–1,1,–1,1,–1,1,–1,–1,–1,–1,–1,1,–1,1,1,–1,1,–1, 1,1,1,–1,–1,1,–1,–1,–1,1,1,1,–1,–1,–1,–1, –1,–1,–1} [8]

Pilots are inserted at subcarriers -21, -7, 7, 21.

The final mapping of the 64 subcarrier is as shown in Figure 5.27.

Figure 5.27: Final 64 subcarrier mapping [8]

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.

The Pipelined, Streaming I/O solution pipelines several Radix-2 butterfly processing engines to offer continuous data processing. Each processing engine has its own memory banks to store the input and intermediate data Figure 5.28. The core has the ability to simultaneously perform transform calculations on the current frame of data, load input data for the next frame of data, and unload the results of the previous frame of data. The user can continuously stream in data and, after the calculation

latency, can continuously unload the results. If preferred, this design can also calculate one frame by itself or frames with gaps in between.

In the scaled fixed-point mode, the data is scaled after every pair of Radix-2 stages. The block floating-point mode may use significantly more resources than the scaled mode, as it must maintain extra bits of precision to allow dynamic scaling without impacting performance. Therefore, if the input data is well understood and is unlikely to exhibit large amplitude fluctuation, using scaled arithmetic (with a suitable scaling schedule to avoid overflow in the known worst case) is sufficient, and resources may be saved.

The input data is presented in natural order. The unloaded output data can either be in bit reversed order or in natural order. When natural order output data is selected, additional memory resource is utilized.

This architecture covers point sizes from 8 to 65536. The user has flexibility to select the number of stages to use block RAM for data and phase factor storage. The remaining stages use distributed memory. [11]

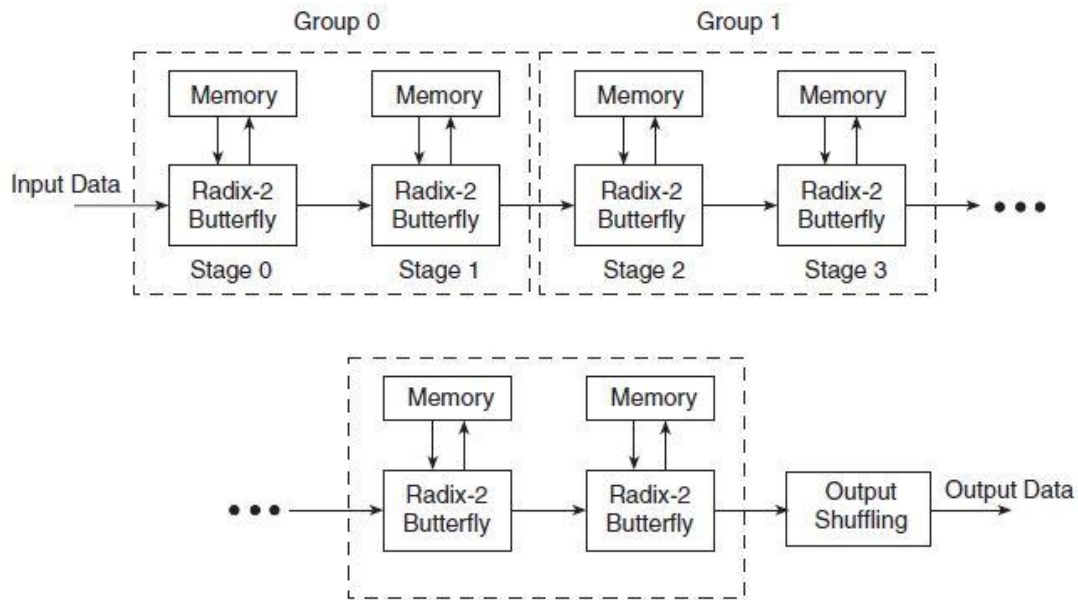


Figure 5.28: pipelined streaming I/O [11]

The IFFT block interface is as shown Figure 5.29, The XK_RE, XK_IM are the input symbols to the IFFT, XK_INDEX is the index of the symbol that should be input to the IFFT now for example in this case, it will be from 0 to 63. XN_RE, XN_IM and XN_INDEX are the same as XK but for output.

CP_LEN is the required cyclic prefix length in our case it will be fixed to 16 symbols. CP_LEN_WE is the control signal to reconfigure CP_LEN. FWD_INV is the configuration bit that defines whether the FFT block will be FFT or IFFT. FWD_INV_WE is the control signal to reconfigure FWD_INV.

Asserting START starts the data loading phase, which immediately flows into the transform calculation phase and then the data unloading phase. Pulsing START once allows the transform calculation for a single frame. Pulsing START every N clock cycles allows continuous data processing. Alternatively, holding START High also allows continuous data processing. START is ignored except when the core can begin loading a new frame, that is, when no data is being loaded, or the last value in the data frame is being loaded. If no NFFT_WE, FWD_INV_WE, or SCALE_SCH_WE were asserted before the initial START, then the defaults are used. This architecture can also support extended intervals between frames. Simply assert START at any time to begin data loading. After the data frame is loaded, the core proceeds to calculate the transform and then output the results. Figure 10 shows the timing of entire frames.

It does not show the small skews between signals which occur at the start and end of frames. [11]

5.6.6.1 Applying Data

Data is applied in a contiguous burst. The point at which data input should start relative to the START pulse is determined by the Input Data Timing parameter set in the GUI.

If "No offset" was selected for the Input Data Timing parameter, the input data (XN_RE, XN_IM) corresponding to the given XN_INDEX should arrive on the same cycle as the XN_INDEX it matches. The first data sample should therefore be applied as soon as RFD goes High, such that the first sample pair is read into the core on the first transition of XN_INDEX, If "3 clock cycle offset" was selected for the Input Data Timing parameter, the input data (XN_RE, XN_IM) corresponding to the given XN_INDEX should arrive three clock cycles later than the XN_INDEX it matches. In this way, XN_INDEX can be used to address external memory or a frame buffer storing the input data. RFD remains High with XN_INDEX during the loading phase and so indicates that data may be input. [11]

5.6.6.2 Data Processing and Data Output

BUSY goes High while the core is calculating the transform. DONE goes High when calculation is complete. EDONE goes High one cycle before that, that is, during the last cycle of the calculation phase. The cycle in which DONE goes High, the core begins unloading. During the unloading phase, while valid output results are present on XK_RE/XK_IM, DV (Data Valid) is High. During unloading, XK_INDEX corresponds to the XK_RE/XK_IM being presented. If cyclic prefix insertion is used, the cyclic prefix is unloaded first. CPV goes High to indicate that the cyclic prefix is being unloaded, and XK_INDEX counts from (point size) - (cyclic prefix length) up to (point size) – 1, After the cyclic prefix has been unloaded, or if the cyclic prefix length is zero, or if cyclic prefix insertion is not used, the whole frame of output data is unloaded. CPV goes Low (if present) and XK_INDEX counts from 0 up to (point size) - 1. [11]

5.6.6.3 Cyclic Prefix Considerations

If cyclic prefix insertion is used, more samples are unloaded from the core than are loaded. Therefore, the core cannot continuously stream frames, but must insert a gap of (cyclic prefix length) clock cycles in between each frame of input data to accommodate the additional clock cycles required to unload the cyclic prefix. This is indicated by the Ready For Start (RFS) pin. RFS goes High when the core is ready for the START pin to be asserted to begin loading the next frame of data. START is ignored except when RFS is High. RFS remains low for (cyclic prefix length) clock cycles after RFD has gone Low, to allow for unloading the cyclic prefix. [11]

A detailed waveform of the timing control is shown in Figure 5.30.

Our final implementation top module is the top_ofdm_wifi whose interface is shown in Figure 5.31, the top_ofdm_wifi contains top_preample_wifi explained above and top_IFFT_controller that we will explain later, the pin diagram for the top_ofdm_wifi is shown in Figure 5.31 and the internal hierarchy of the top_ofdm_wifi is shown in Figure 5.32 and in Table 5.10.

To generate correct output the IFFT_controller start the preamble (preamble_st) module when the mapper is ready to send data, the preamble output lasts for 16 us and the IFFT first patch of outputs will have about 12 us latency so the preamble sends a signal (enable_iff) for IFFT after 4us of its operation so the IFFT will start processing

and its output will start after the preamble, to guarantee the continuity of output as shown in Figure 5.33.

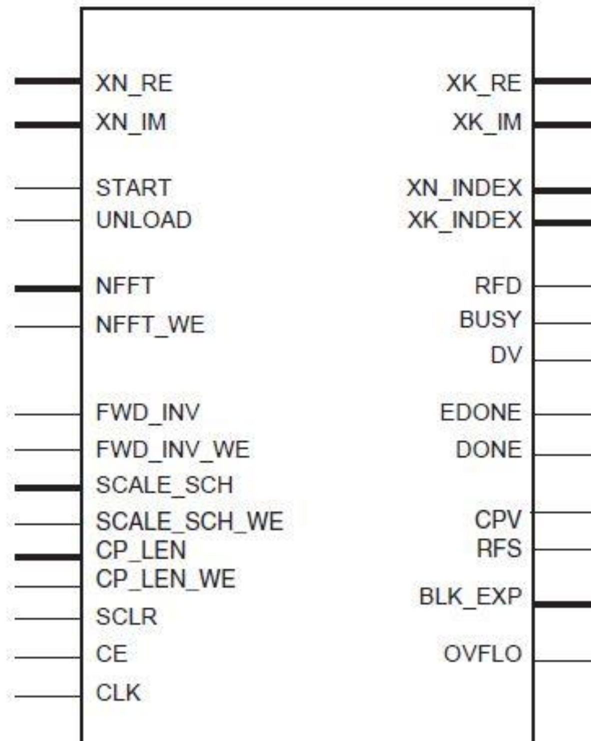


Figure 5.29: IFFT block interface [11]

The IFFT_controller consists of two ram of size 48 and the LogiCORE FFT IP, The two ram sizes are used to arrange the symbols according to the mapping function explained before. To ensure correct operation of the FFT streaming block we must have an input ready whenever the FFT block requests input. The FFT streaming block takes 48 symbols then takes no input for 16 clock cycles then takes input again. So we used two memories to buffer input so whenever one memory is inputting to the FFT block the other one has the next input ready for the next patch.

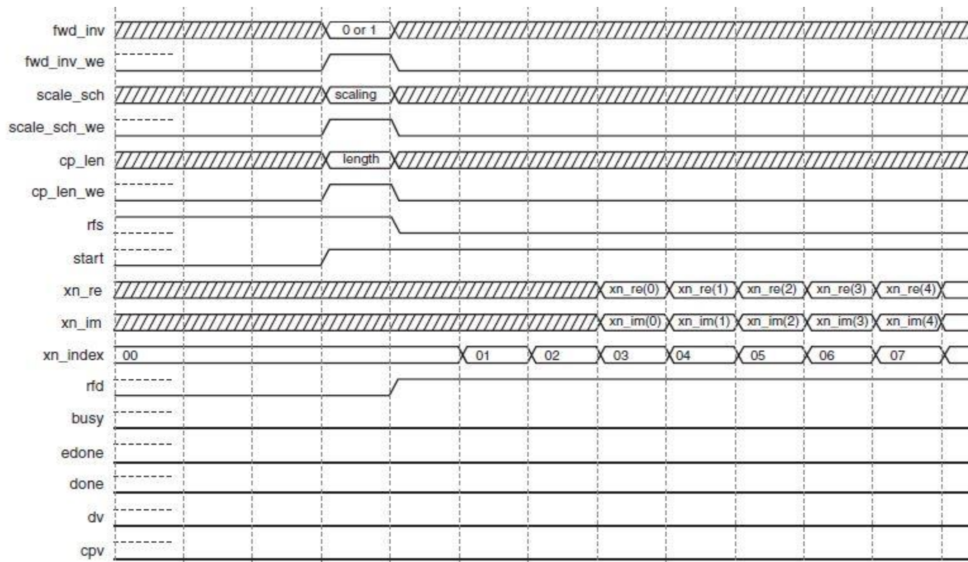


Figure 5.30: FFT timing for applying data

Figure 5.31: top_ofdm_wifi interface

Table 5.10: top_ofdm_wifi pin description

Pin	Description
Data_in_im	Imaginary input data
Data_in_re	Real input data
Clk	IFFT clk (20 MHz)
Clk_fast	System fast clk
enable	Not used here
Last_sym	Mapper marking to the IFFT that these patch of symbols are the last patch to be processed
Mapper_ready	Marks that mapper started to input symbols to its pipeline
Reset	Reset the IFFT_controller
Valid_in	Marks that the mapper output signals are valid
Data_out_im	Output imaginary part data
Data_out_re	Output real part data
Finished	Signals the mapper that the IFFT is ready to receive data
Valid_out	Signals the IFFT output now are valid

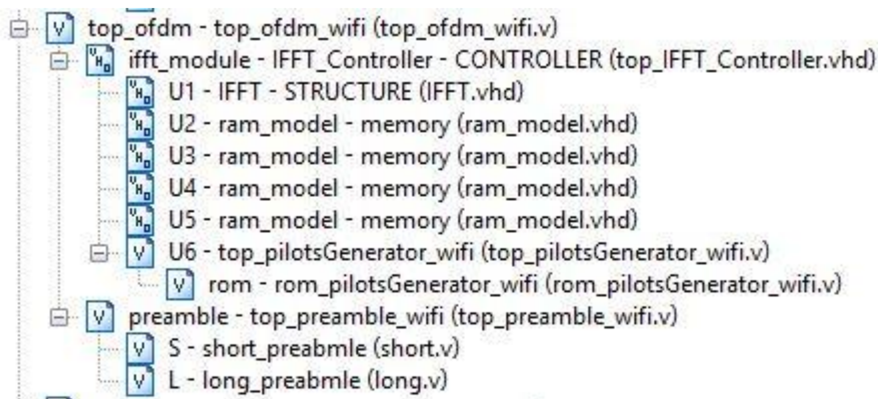


Figure 5.32: top_ofdm_wifi hierarchy

Figure 5.33: Waveform of preample and IFFT outputs

5.6.7 Preamble

In Wifi 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 5.34. The timings described in this subclause and shown in Figure 5.34 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 5.34: OFDM training structure.

Figure 5.34 shows the OFDM training structure (PLCP preamble), where t1 to t10 denotes short training symbols and T1and T2 denote long training symbols. The PLCP preamble is followed by the SIGNAL field and DATA.

The total training length is 16μs. The dashed boundaries in the figure denote repetitions due to the periodicity of the inverse Fourier transform.

A short OFDM training symbol consists of 12 subcarriers, which are modulated by the elements of the sequence, given by

$$S_{-26,\,26} = \sqrt{(13/6)} \times \{0, 0, 1+j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 0, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0\}$$

A long OFDM training symbol consists of 53 subcarriers (including the value 0 at dc), which are modulated by the elements of the sequence L, given by

$$L_{-26,\,26} = \{1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 0, 1, -1, -1, 1, 1, -1, 1, -1, 1, -1, -1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, -1, 1, 1, 1, 1\}$$

The PLCP preamble shall be transmitted using an OFDM modulated fixed waveform. The IEEE 802.11 SIGNAL field, BPSK-OFDM modulated with coding rate 1/2, shall indicate the modulation and coding rate that shall be used to transmit the MPDU.

Concerning HDL modeling, and to save performing IFFT for constant values each frame, we have implemented a lookup table for the time domain representation of the sequences.

Table 5.11Table 5.11 shows the time domain representation of the short sequence.

There is another table for the long preamble. As implementing such lookup table will be a very hard work, we have used a MATLAB script that reads this table, re-arrange the data in a suitable format and generate a Verilog code for the preamble generator.

The interface of preamble generator block is shown in Figure 5.35

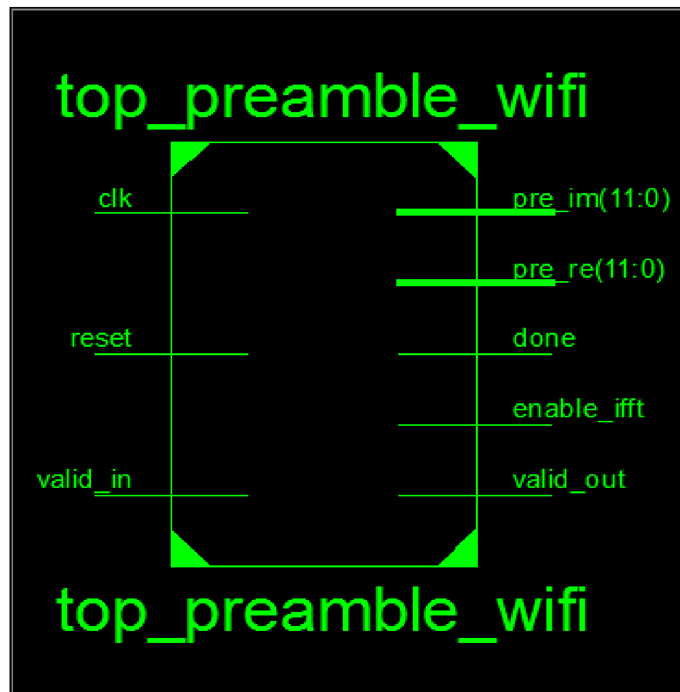


Figure 5.35: Interface of Wi-Fi preamble generator

Table 5.11 : Frequency domain representation of the short sequences

##	Re	Im	##	Re	Im	##	Re	Im	##	Re	Im
100	0.092	0.000	101	0.143	–0.013	102	–0.013	–0.079	103	–0.132	0.002
104	0.046	0.046	105	0.002	–0.132	106	–0.079	–0.013	107	–0.013	0.143
108	0.000	0.092	109	–0.013	0.143	110	–0.079	–0.013	111	0.002	–0.132
112	0.046	0.046	113	–0.132	0.002	114	–0.013	–0.079	115	0.143	–0.013
116	0.092	0.000	117	0.143	–0.013	118	–0.013	–0.079	119	–0.132	0.002
120	0.046	0.046	121	0.002	–0.132	122	–0.079	–0.013	123	–0.013	0.143
124	0.000	0.092	125	–0.013	0.143	126	–0.079	–0.013	127	0.002	–0.132
128	0.046	0.046	129	–0.132	0.002	130	–0.013	–0.079	131	0.143	–0.013
132	0.092	0.000	133	0.143	–0.013	134	–0.013	–0.079	135	–0.132	0.002
136	0.046	0.046	137	0.002	–0.132	138	–0.079	–0.013	139	–0.013	0.143
140	0.000	0.092	141	–0.013	0.143	142	–0.079	–0.013	143	0.002	–0.132
144	0.046	0.046	145	–0.132	0.002	146	–0.013	–0.079	147	0.143	–0.013
148	0.092	0.000	149	0.143	–0.013	150	–0.013	–0.079	151	–0.132	0.002
152	0.046	0.046	153	0.002	–0.132	154	–0.079	–0.013	155	–0.013	0.143
156	0.000	0.092	157	–0.013	0.143	158	–0.079	–0.013	159	0.002	–0.132
160	0.023	0.023									

5.7 802.11a Receiver PHY Block Diagram

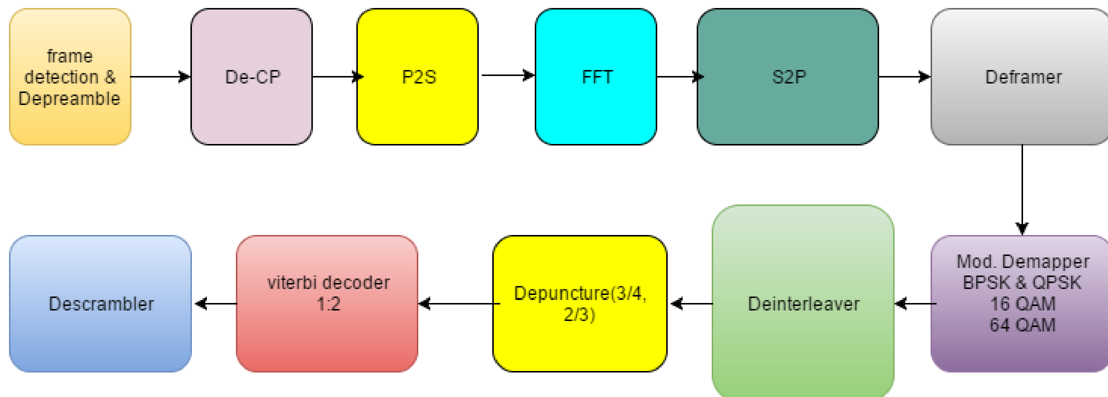


Figure 5.36: Wi-Fi Rx Block Diagram

5.7.1 DeMapper

It receive the real and imaginary data of the channel 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.

In WI-FI we have a Phase Shift Keying (BPSK, QPSK) and Quadrature Amplitude Modulation (16-QAM) and the decision regions are shown in Figure 5.37

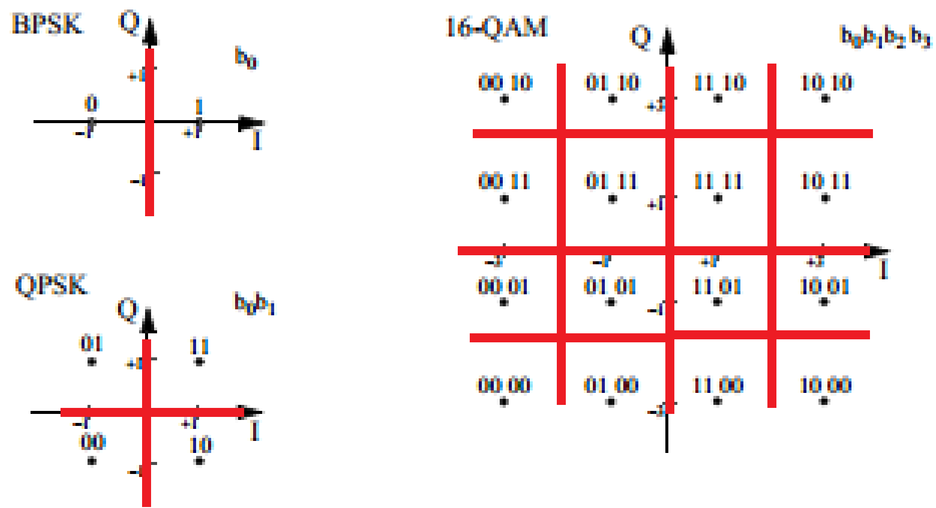


Figure 5.37: decision regions of WIFI demapper

The top Demapper is shown in Figure 5.38

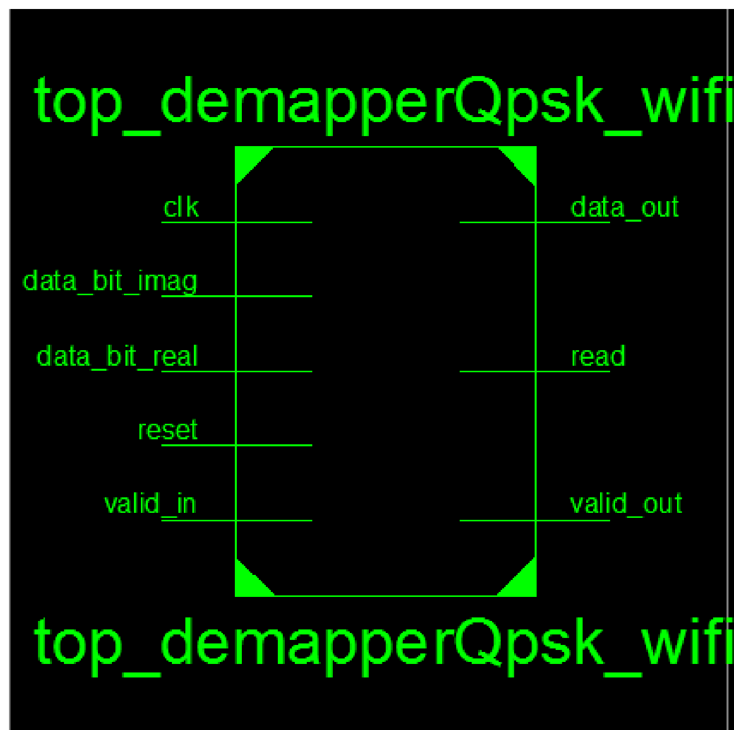


Figure 5.38: Top Demapper

The pins description of the top Demapper are shown in Table 5.12

Table 5.12: Pins description of the demapper

PIN	Description
Data_bit_imag	The imaginary part of the input bits
Data_bit_real	The real part of the input bits
Valid_in	The signal indicates that current data bit real and imaginary are valid data
Data_out	The output data in the form of stream of bits
Valid_out	The signal indicates that current data_out is valid data
read	The signal indicates that the block is ready to read data of the next symbol

5.7.2 DeInterleaver

The deinterleaver, which performs the inverse relation, 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.

The first permutation is defined by the rule:

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

This permutation represents the inverse equation of the second permutation equation in the interleaver of the transmitter (Section 5.6.4).

The second permutation is defined by the rule:

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

This permutation represents the inverse equation of the first permutation equation in the interleaver of the transmitter (Section 5.6.4).

The value of s is determined by the number of coded bits per subcarrier, N_{BPSC}, according to:

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

The block interface is as shown in Figure 5.17, the signals declaration and description is as shown in Table 5.6 and the block simulation is as shwon in Figure 5.18.

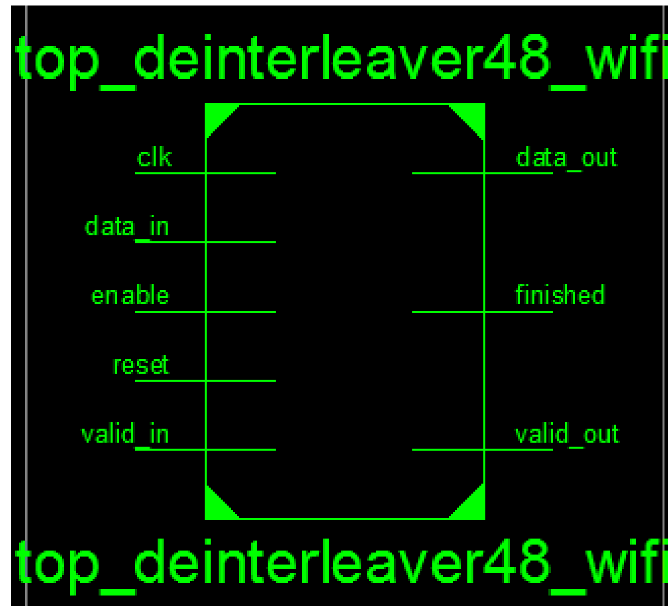


Figure 5.39: Deinterleaver block interface.

PIN	PIN TYPE	Description
enable	IN	This signal indicates that the next block is ready to have data
valid_in	IN	This signal indicates that current data_in is valid data
data_in	IN	The input bits
finished	OUT	This signal indicates that the interleaver is ready to have a new frame
valid_out	OUT	This signal indicates that current data_out is valid data
data_out	OUT	The output bits

The implementation of this block is very close to the implementation of the interleaver block because the implementation of this block is also based on a RAM, this RAM used to store all the input data to the deinterleaver block as shown in Figure 5.41 then we read from this RAM but out of order according to the index that calculated from the deinterleaving equations as shown in Figure 5.42 .

The deinterleaving equation is different according to N_{CBPS} and N_{BPSC} values, these values is as shown in Table 5.7.

So finally we have four deinterleavers:

$N_{BPSC} = 1$, $N_{CBPS} = 48$

$N_{BPSC} = 2 \, , N_{CBPS} = 96$

$N_{BPSC} = 4 \, , N_{CBPS} = 192$

$N_{BPSC} = 6 \, , N_{CBPS} = 288$

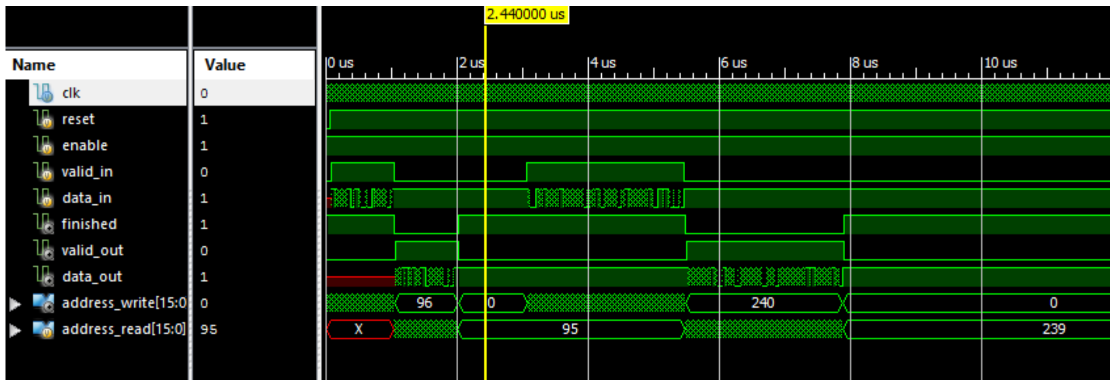


Figure 5.40: Deinterleaver block simulation.

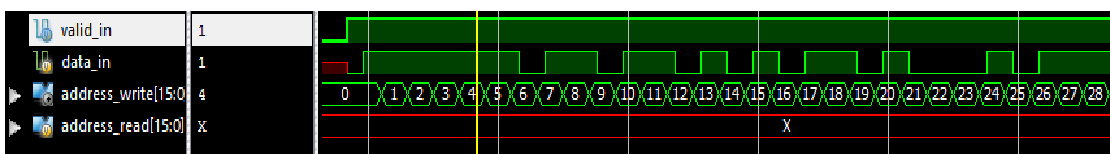


Figure 5.41: Writing the input data in the RAM.

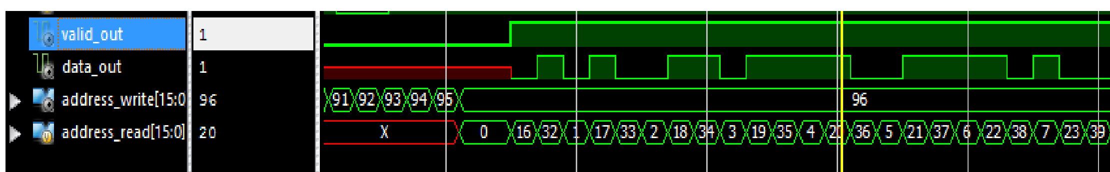


Figure 5.42: Reading the data from the RAM.

5.7.3 Depuncture

Depuncture is the reverse block of puncture. Depuncture 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. 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. Figure 5.45 shows the procedure of puncture and depuncture of rate ¾. Figure 5.46 and Figure 5.45 shows the procedure of puncture and depuncture of rate 2/3.

The interface of depuncture block is shown in Figure 5.43.

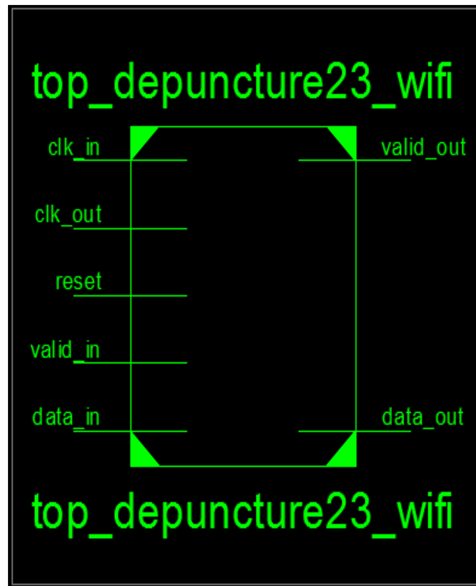


Figure 5.43 : The interface of depuncture block diagram.

The internal block diagram of depuncture is shown in Figure 5.44.

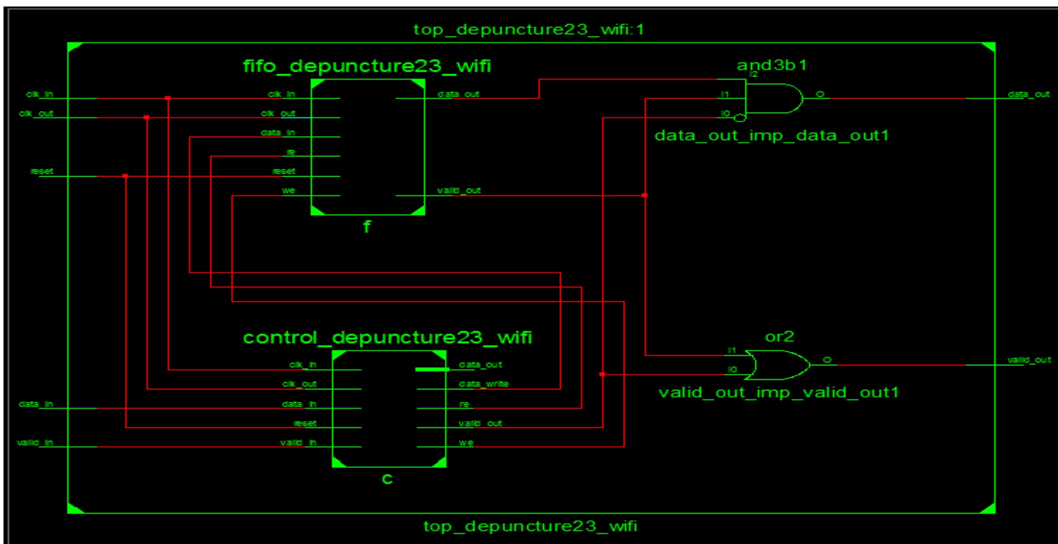


Figure 5.44 : internal block of depuncture.

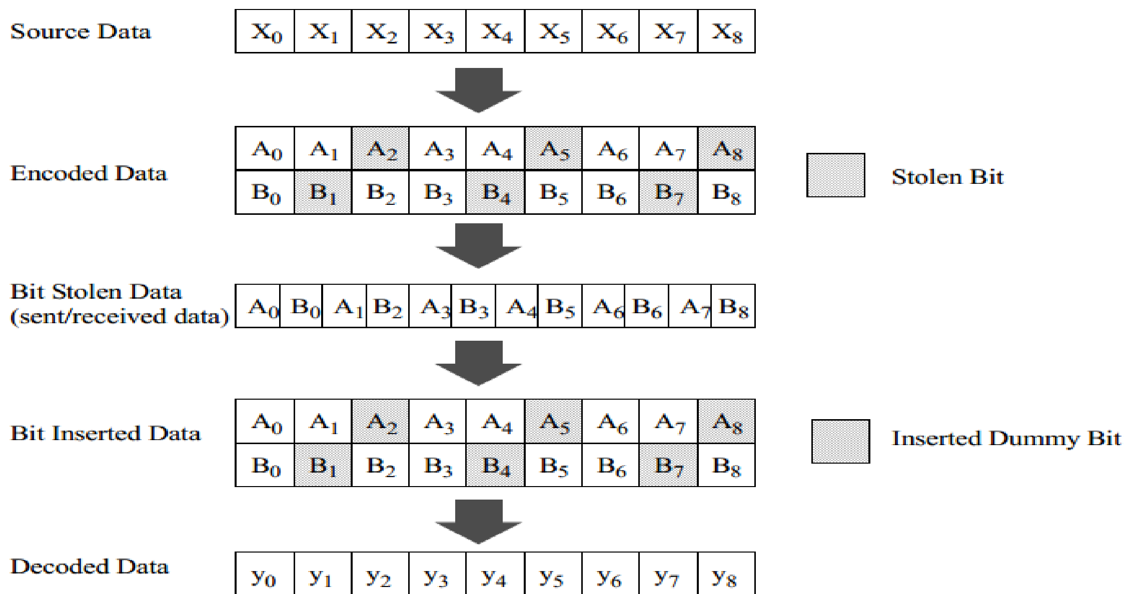


Figure 5.45: Depuncture 3/4 rate procedure

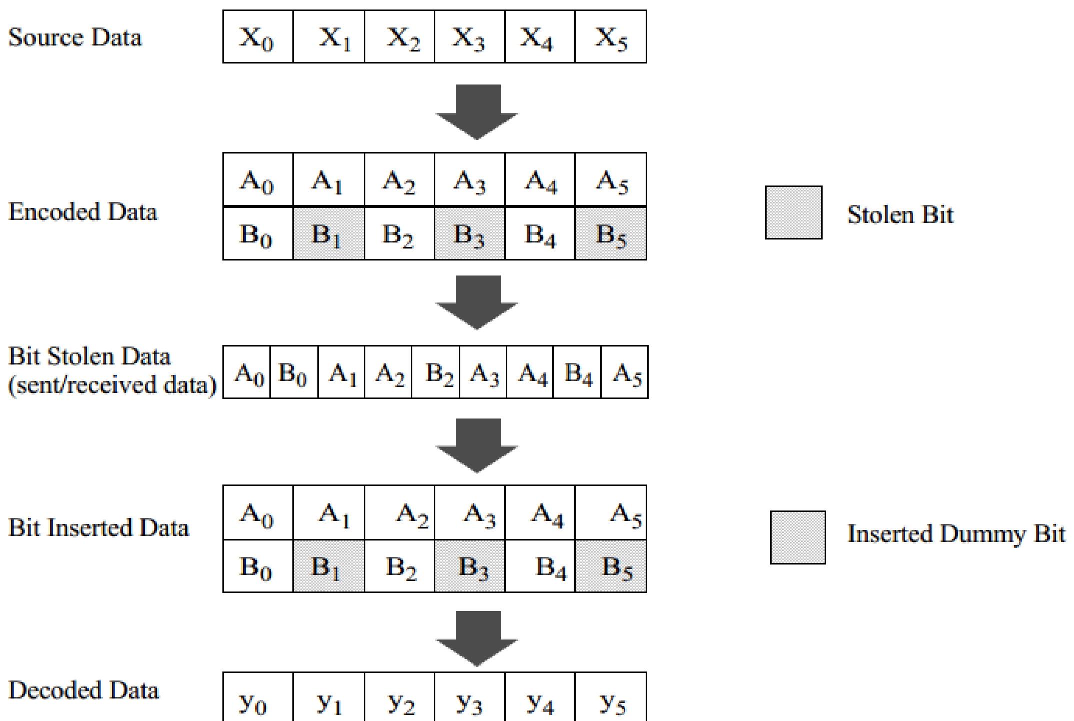


Figure 5.46 : Depuncture 2/3 rate procedure

5.7.4 Convolutional Decoder:

5.7.4.1 Decoding of Convolutional Encoded Data using Viterbi Algorithm (VA).

The trellis in Figure 5.47 shows the transitions possible for example encoder for sequence of input bits 1 0 1 1 1 0 0.

The VA works by using the received version of the encoded bit sequence to find the most likely path through this trellis representing the state machine of the encoder. Once this most likely path through the trellis is known, the data bits which would have caused the encoder to follow this path can be implied and these bits are the output from the VA.

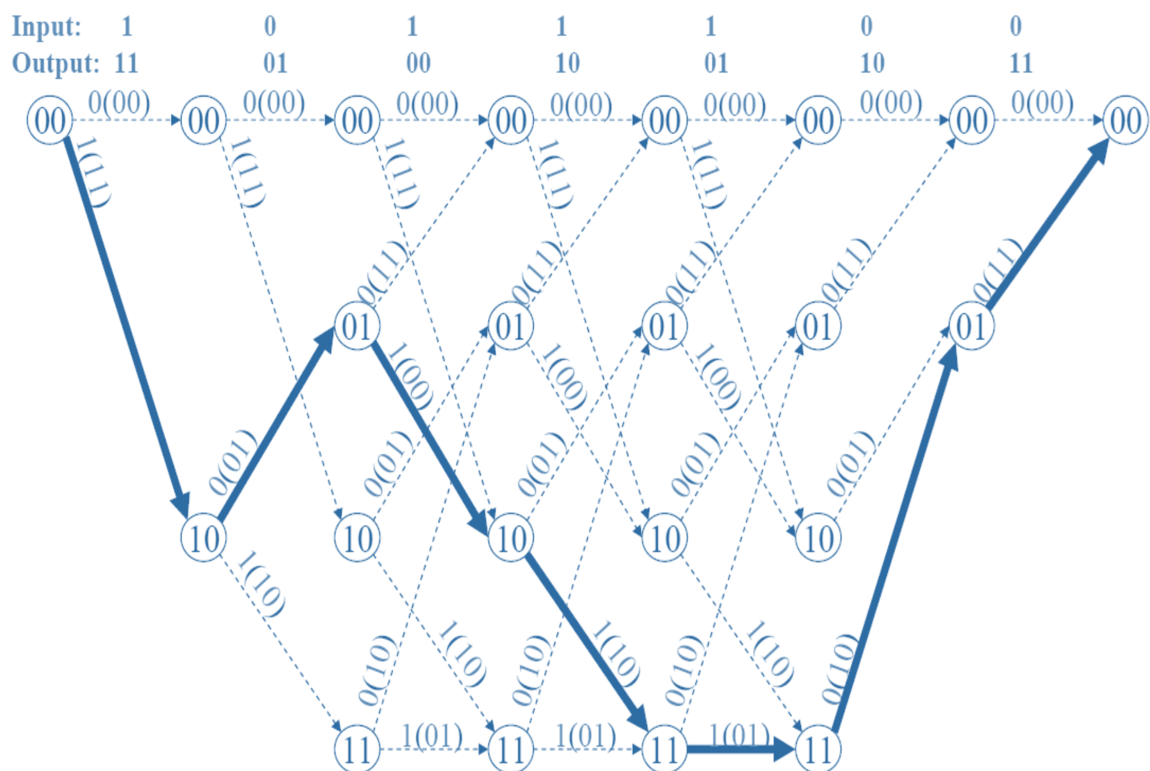


Figure 5.47: Trellis diagram of Convolutional Encoder

Viterbi algorithm is called optimum algorithm since it minimizes the probability of error. The main drawback of These Viterbi Decoders is that they are very expensive in terms of chip area.

The basic units of Viterbi decoder as shown in Figure 5.48 are:

- Branch metric unit (BMU) where received data symbols are compared to the ideal outputs of the encoder from the transmitter to compute branch metrics which is Hamming distance or the Euclidean distance. The hamming distance is the number of bits not matching the possibility and the Euclidean distance is the point distance between the possibilities and the received data, which is obtained using the point distance formula. Hamming distance is selected as it is easy to implement on hardware.
- Add–compare–select unit (ACSU) which selects the survivor paths for each trellis state, also finds the minimum path metric of the survivor paths.
- Survivor memory unit (SMU).
- Metric Memory Unit (MMU).
- Trace Back Unit (TBU) is responsible for selecting the output based on the minimum path metric.

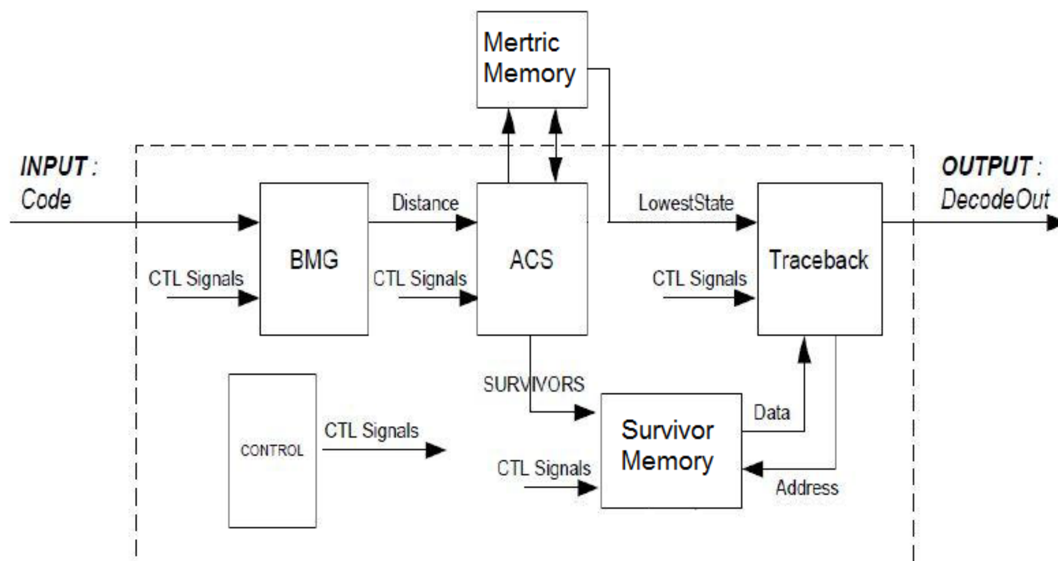


Figure 5.48: Block Diagram of Viterbi decoder

Viterbi algorithm can be explained briefly with the following three steps as shown in Figure 5.49.

1. Get one input code word (2 bits or 3 bits corresponding to the coding rate).

2. Calculating the branch metric

3. Reading the previous path metric for all the states from the Metric Memory.

4. Add the branch metric to the path metric for the old state.

5. Compare the sums for paths arriving at the new state (there are only two such paths
incoming).

6. Select the path with the smallest value which is called the survivor path. If both path Metrics are equal then any one is chosen.

7. Writing the survivor path in survivor memory unit to be used in the trace back process.

8. Writing the new path metric in metric memory unit

9. When the sliding window reaches its end then begins the trace back process

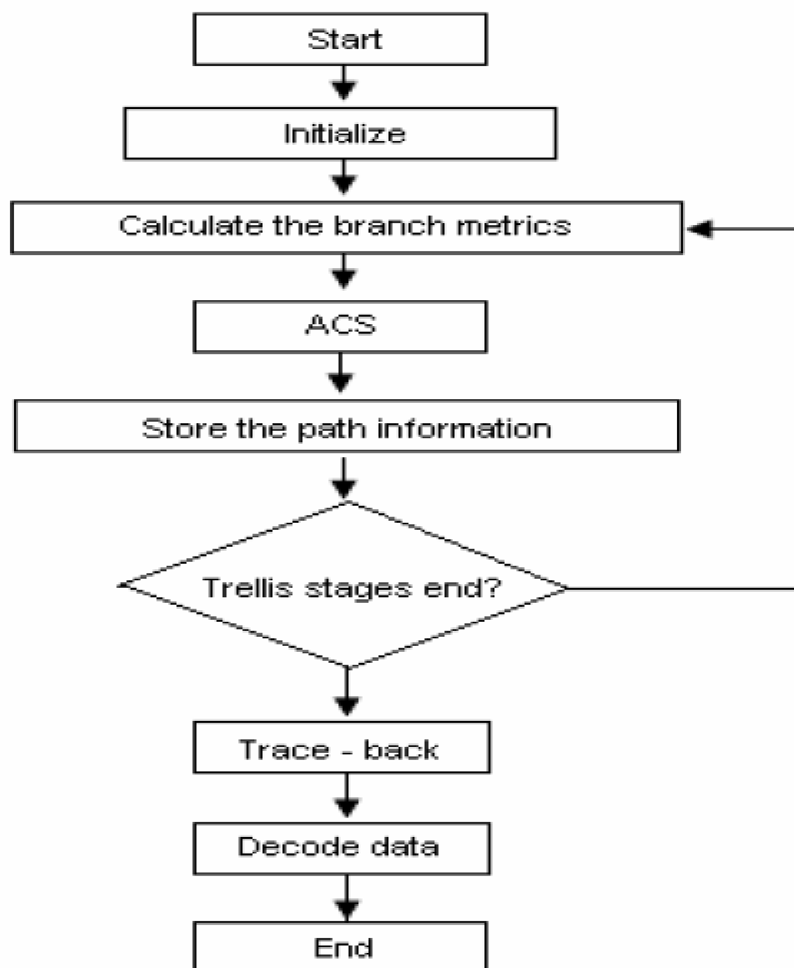


Figure 5.49: Viterbi decoder algorithm flow chart

The following example shown in Figure 5.50 describes the process of Conventional Decoding

Consider the received sequence as 11 10 00 01 01 11 which is error free and should be decoded.

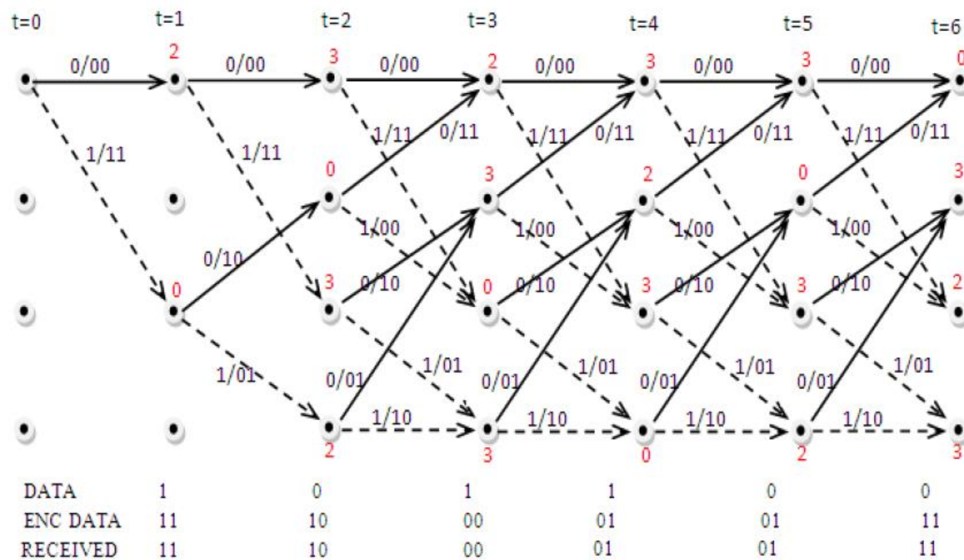


Figure 5.50: Trellis diagram for error free decoding

Given the message data 101100 the encoded output is 11 10 00 01 01 11 which is received error free at the receiver. After completing the first two steps explained above the path metrics to reach each state in the trellis is obtained which is shown in red color just above the states. After calculation of the path metrics the survivor unit traces back the optimum path which will always start from state zero.

The following example shown in Figure 5.51 describes the process of Conventional Decoding in case of errors.

Consider the output is: 11 01 00 10 01 10 11 and received sequence is 11 11 00 10 01 11 11.

5.7.4.2 Design Specification:

Code Rate = ½, Constraint Length (K) = 7 which denotes the length of the Convolutional encoder, The amount of branch metric is 2 ^ K= 128 branches, The amount of state metric is 2 ^ (K-1) = 64 states, window size equals 64 stages, and

Hard Decision which means that the demodulator is quantized to two levels: zero and one.

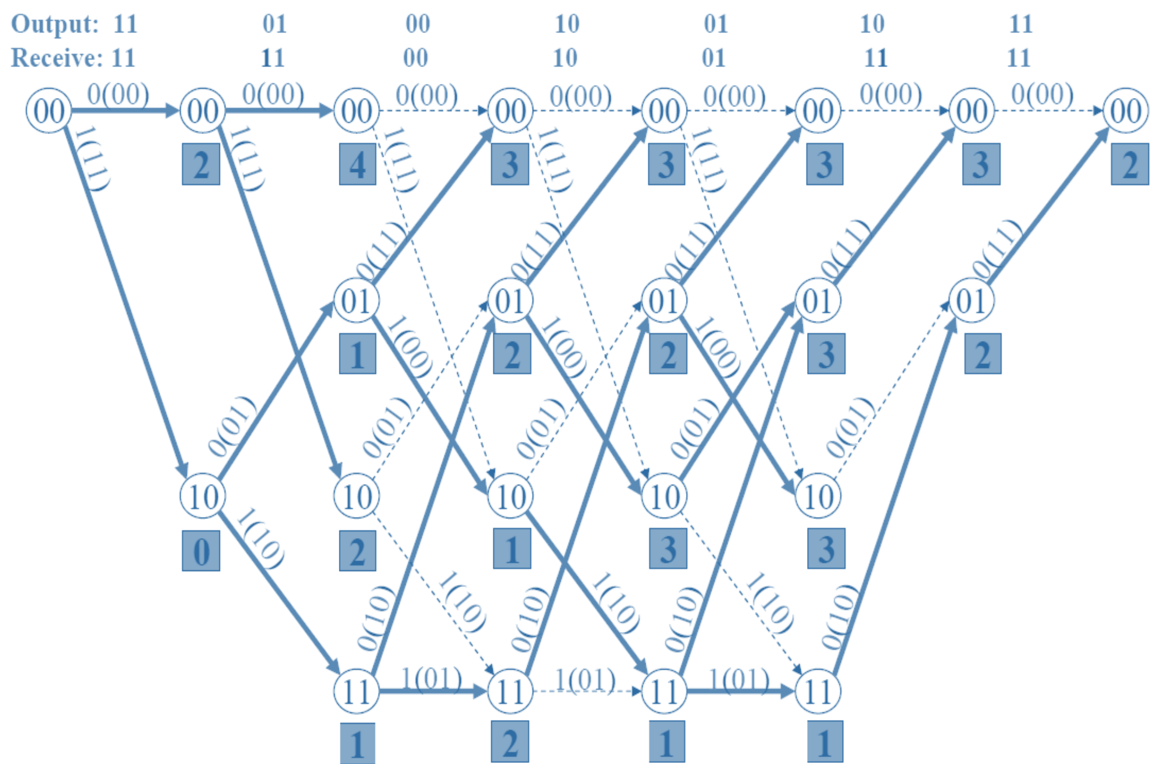


Figure 5.51: Trellis diagram for error decoding

Since we have 64 states should be estimated at each decoding instants (each input code word) but this requires 64 ACS (Add-Compare-Select) in parallel which requires more Hardware which will increase the cost so we put 4 ACSs only and this will make 4 states only are ready at a time and then we need 16 iteration to complete estimation of the 64 states as shown in Figure 5.53. 4 states are processed together and from the observations any next state can be reached through 2 previous state, so we can see the 64 states as a 16 group each group contains 4 adjacent states (for example "S0s”,"S1", "S2" and "S3" are in one group).we need first an address that indicates which group's processing is in progress and since we have 16 group so we need 4 bits to indicate the group and this is done through ACSSegment. The other important thing is that to process 4 states we have 8 previous states that can reach these next states (recall the observations) so we need another bits to indicate which previous state and branch. The next example of K=3 trellis diagram in Figure 5.52 may help us to understand this. But notice that in this example we assume the following we have only 2 ACSs and then each group contains only 2 states and hence we have 4 groups so we need 4 iterations and we need ACSSegmet to be 2 bits only.

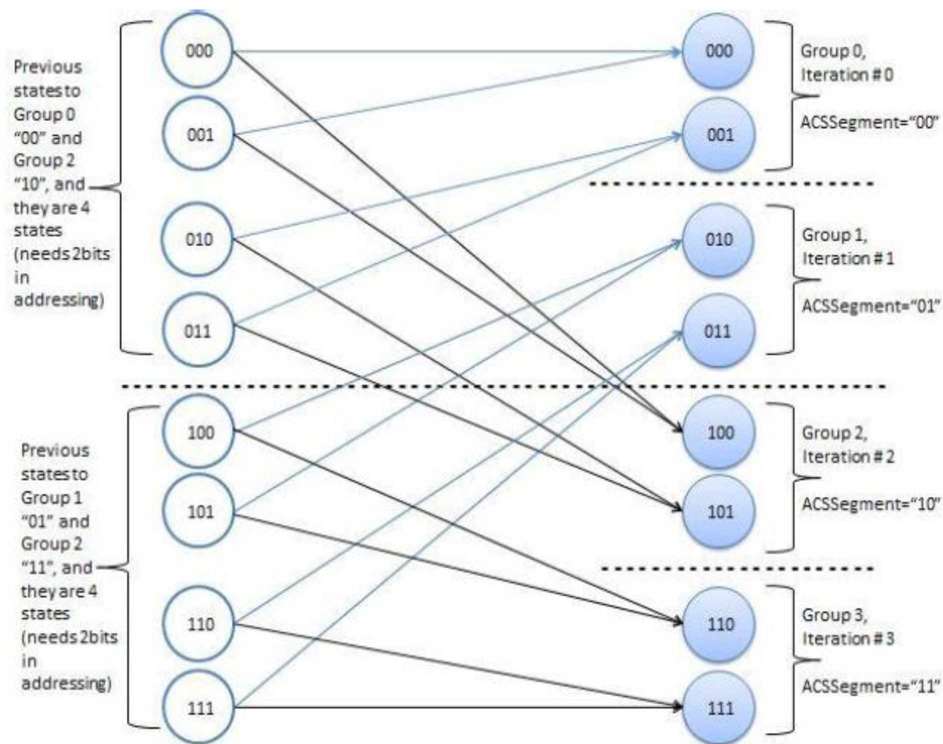


Figure 5.52: Example of 8 state Trellis diagram.

As shown in the previous figure, the previous states are in the left and the next states are in the right, the blue branches stands for "0" input bit and the black branches stand for “1” input bit .Ok let's start the methodology of addressing the next state , as said before we need in this example 4 iterations at iteration # 0 we need to process group 0 which contains states "S000" and "S001" and they can be reached from the previous states "S000","S001","S010" and "S011" and to process Group1 which contains "S10" and "S11" which can be reachesd from "S100","S101","S110" and "S111". With simple observation we can say the following to access the memory reading the path metrics of the previous states we need to 2 additional bits that changes from 00 to 11 and another most significant bit that chooses whether the first 4 previous states(S"0 00" to S"0 11") or the second 4 previous states (S"1 00" to "S1 11"). Also with simple observations we can deduce that this most significant bit(s) is/are the ACSSegment bits without the ACSSegment MSB. Here in this example ACSSegment[0] is used in addition to the 2 bits that get all the combinations. Ok now to generalize these observations in our implementation, ACSSegment are 4 bits [3:0] they are used to know which Group of next states will be processed in the iteration #(ACSSegment[3:0]) and 2 bits changes from 0 to 3 to indicate which state of the states(we will called them as State ID[1:0]), in other words if we get the new metrics

and we need to write them in the metric memory we use an address as following {ACSSegment[3:0], State ID[1:0]}, for example if we need to calculate the new metric of the state "0 000 00" that can be reached only from state"0000 00" or state"0 000 01" .

Figure 5.53-AcsSegment timing digram

5.7.4.3 Branch Metric Unit:

The BMU receives Code signals, calculating its distance with all possibility of branch metric, giving the output of Distances signal. Hard Decision which means that the demodulator is quantized to two levels: zero and one. Values generated are depending on the value of ACSSegment as shown in Figure 5.55. The Distance Calculator block computes the hard-distance of Code with the branch metric from Viterbi Encoder block.

The Block diagram of the BMU Unit is as following Figure 5.54

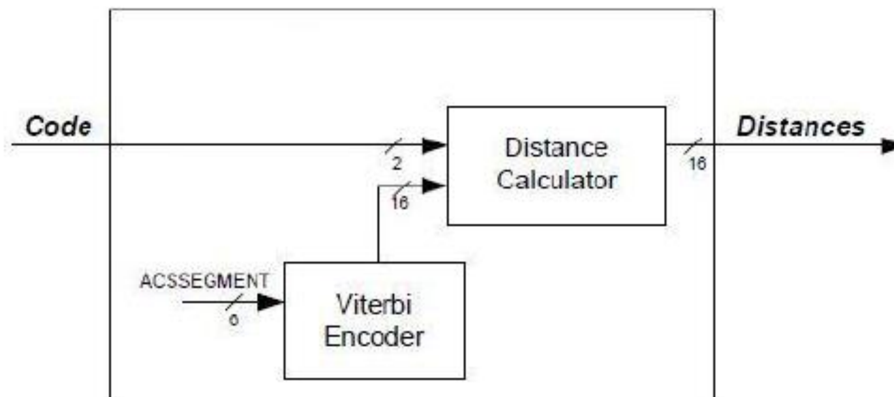


Figure 5.54: BMU Unit block diagram

The distance calculator calculate the distance for each branch which is the number of error bits between the received code word and the output of the branch and this distance is stored in 2 bits only since in case of R=1/2 , the error could be 0, 1 or 2 bits which needs only 2bits.

In the case of R=1/3 the error between the received code word (which is 3 bits in this case) and the output of the branch can be 0, 1, 2 or 3 bits which needs only 2 bits.

The calculation of the distance differs from the R=1/2 and R1/3 cases as following in Table 5.13 and Table 5.14:

Output of branch metric unit is 16 bits which is concatenation of eight branch metrics as shown in Figure 120.

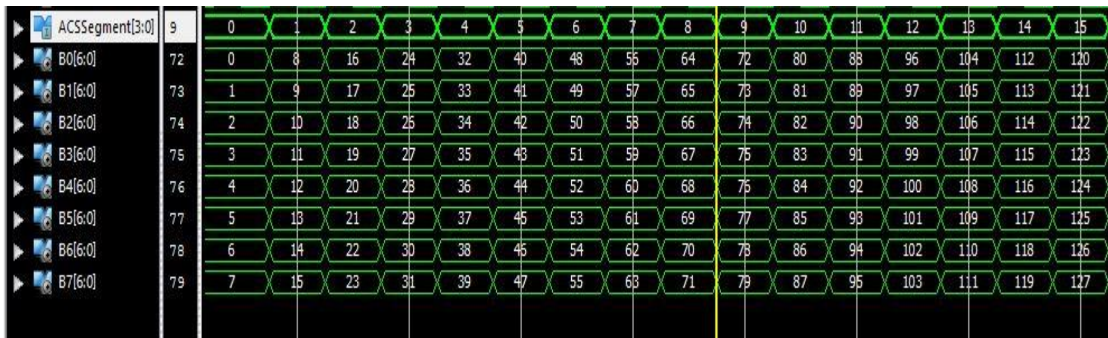


Figure 5.55-Branch ID Values generation

The distance is function of XOR output between received code and branch output

Table 5.13: Output Distance in case of rate 1/2

XOR Output[1:0]		Output Distance[1:0]	
MS	LS	Output Distance[1]	Output Distance[0]
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

For Rate ½:

Output Distance [0] = MS^LS

Output Distance [1] =MS&LS

For rate 1/3:

Output Distance [0] =MS^XS^LS.

Output Distance [1] =MS&XS+XS&LS+LS&MS.

Table 5.14: Output Distance in case of rate 1/3

XOR Output[2:0]			Output Distance[1:0]	
MS	XS	LS	Output Distance[1]	Output Distance[0]
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

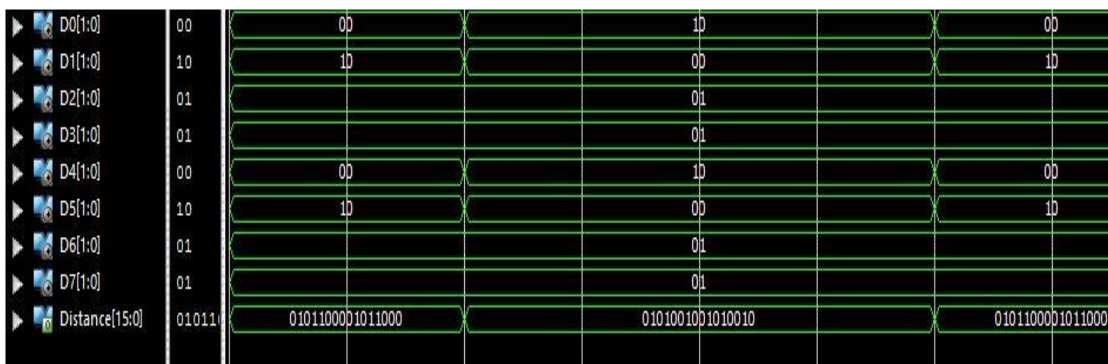


Figure 5.56-Branch metric timing diagram

5.7.4.4 Add–compare–select unit (ACSU)

It adds the path metric to the distance and compare the new path metric to choose the least metric and save it as a survivor path as shown in .

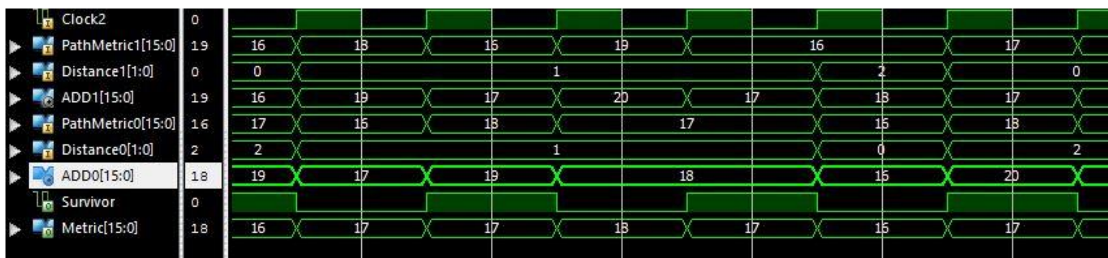


Figure 5.57-ACSU timing diagram

5.7.4.5 Metric Memory Unit:

The metric values are saved on Metric Memory. Two blocks of RAM needed as we have to know the current metric values and save the next metric values we've just calculated. Each Memory has its own index, therefore the addressing scheme using signal MMReadAddress and MMWriteAddress.

We should toggle the read from and the write in Memories between the decoding instants (every new input code word) as shown in . If MMBlockSelect = 0: Read from RAM B and Write from RAM A. MMBlockSelect = 1: Read from RAM A and Write from RAM B,if MMBlockSelect =1: the previous Path Metrics are stored in RAM A then we read from RAM A and after calculation the new Path Metrics write them to RAM B.

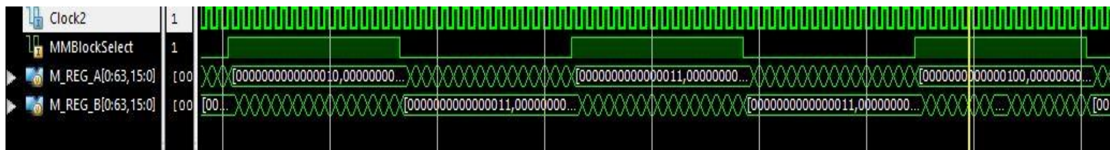


Figure 5.58-Metric Memory Unit timing diagram

5.7.5 Descrambler

The same as stated in section 5.6.1

Chapter 6: LTE transmitter implementation

LTE stands for Long Term Evolution. The technology designed and developed by 3GPP (Release 8) as air interface for cellular mobile communication systems. It is used to increase the capacity and data transfer speed of mobile telephone networks used mainly for data communication. LTE is marketed as 4G technology.

LTE uses OFDMA in the downlink and SC-FDMA in the uplink. It supports six different channel bandwidths from 1.4 to 20 MHz and both frequency- and time-division-duplex (FDD and TDD) modes. The resource allocation in LTE is as based on resource block concept defined. LTE supports various frequency bands in both TDD (Band 33 to 43) and FDD (Band 1 to 25).

Since the bands for the GSM and UMTS are implemented previously, therefore we use them bands in the LTE as it is expected that at one day the GSM will vanish and the band will be unutilized. Consequently, the LTE uses the band of the GSM.

About the duplex technique used to separate between the UL and DL. In LTE we use both the FDD and TDD technique where we can use F1 for UL and F2 for DL in case FDD or we can divide the frame in slots used for UL and DL. The frame distribution is going to be explained later.

Before moving to the frame structure let's consider the BW for the Carrier in LTE. The Carrier BW in LTE may be one of the following values (1.4, 3, 5, 10, 15, 20)MHz.

Back to the Frame structure, we have two frame structures in LTE.

The first frame structure is for FDD and the second one is for TDD. FDD is dividing the frequency into different subcarriers. The distribution of frequencies is as follow.

- 12 subcarrier per every sub-channel.
- Sub-channel BW is 180KHz
- Subcarrier BW is 15KHz
- For the first frame structure, it will be FDD. The distribution for the frame per one subcarrier is shown in Figure 6.1 .

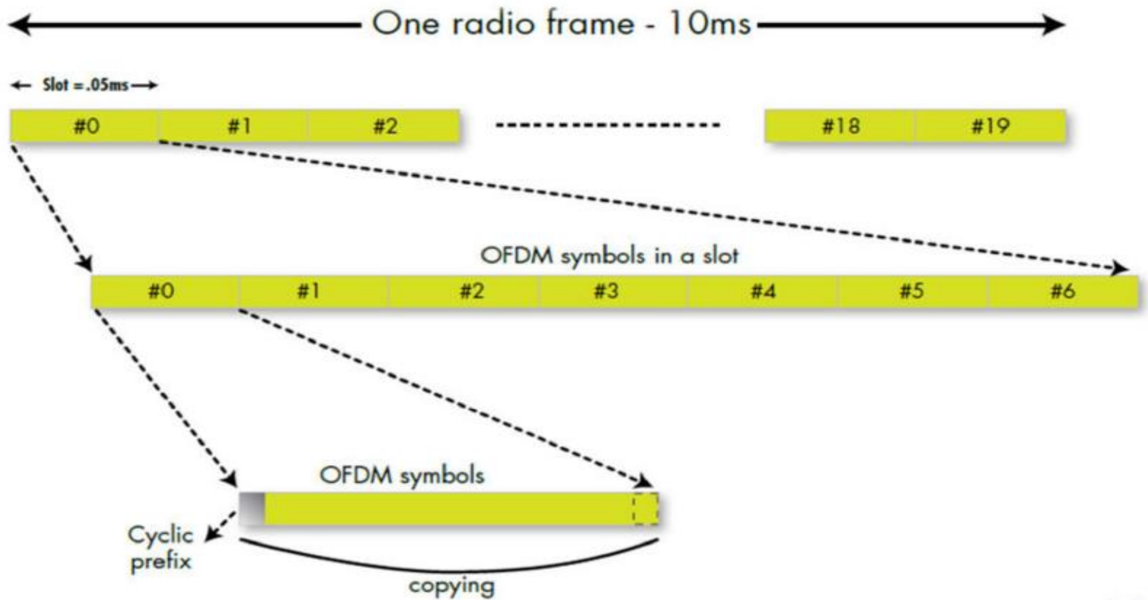


Figure 6.1 : First LTE frame structure.

The subcarrier is divided into frames. The time for one frame is 10msec. The frame is divided into sub-frame each with duration 1msec. the sub-frame is divided into two slots. The slot carries the OFDM symbol where the symbol contains both the samples plus the CP. The number of bits per sample depends on the modulation technique used; it may be 16QAM-64QAM ……etc.

Now taking a focus look for the usage of the resource in LTE. As shown in Figure 6.2the resource of the LTE are divided horizontally to 12 different frequencies represent the subcarriers of LTE. Each frequency is divided into small blocks where the small block represents a symbol and every 7 successive symbols represent a time slot.

.

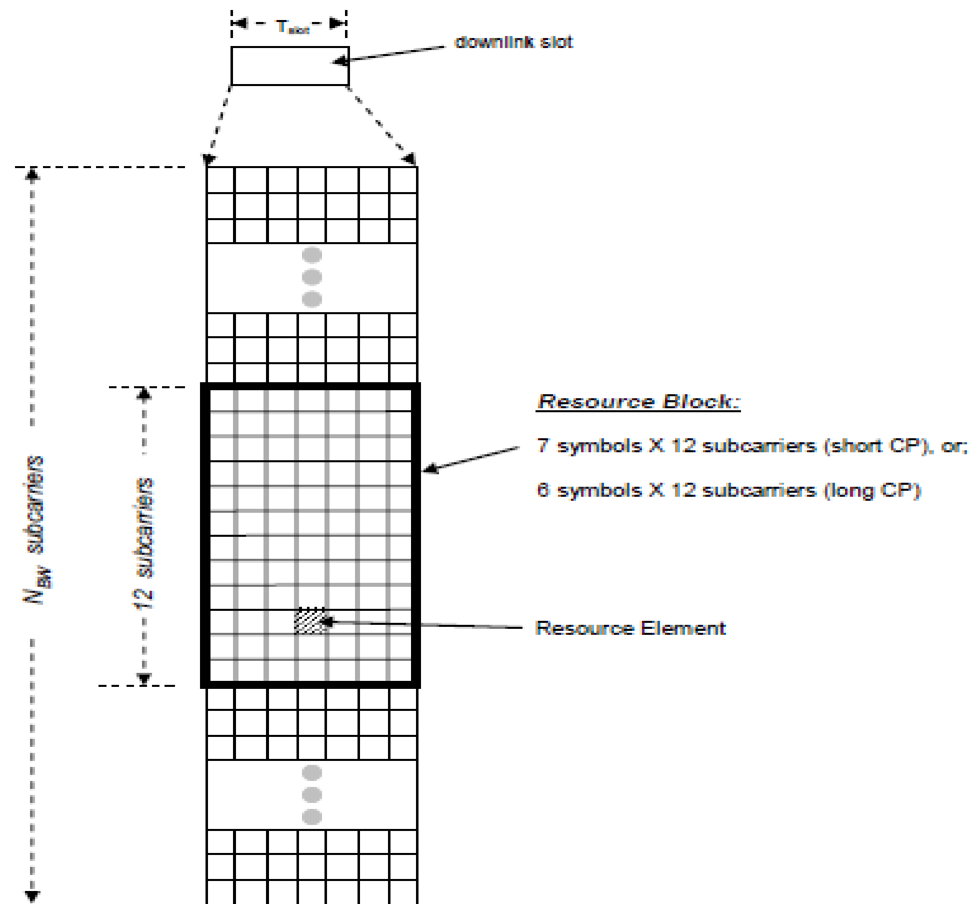


Figure 6.2 : Resource elements in LTE.

To adjust synchronization and to see the effect of channel on data being transmitted, reference signals are embedded within the data during transmission as shown in Figure 6.3. Moreover, the reference signal is used during the usage of MIMO antennas concept.

The second frame structure is TDD, shown in Figure 6.4.

As shown in Figure 6.4, time slot may be used for either the UL or DL. If the user need more speed for the downlink he takes more slots for DL rather than UL, radio frame composed of two half frames, each of 5ms duration resulting in total frame duration of about 10ms. Each radio frame will have total 10 sub-frames; each sub-frame will have 2 time slots. Sub-frame configuration is based on Uplink downlink configuration (0 to 6). Usually in all the cases, sub-frame #0 and sub-frame#5 is always used by downlink. The Special sub-frame carry DwPTS (Downlink Pilot Time Slot),GP(Guard Period) and UpPTS (Uplink Pilot Time Slot). For the 5ms DL to UL

switch point periodicity case, SS(Special subframe) exists in both the half frames. For the 10ms DL to UL switch point periodicity case, SS exists only in first half frame.

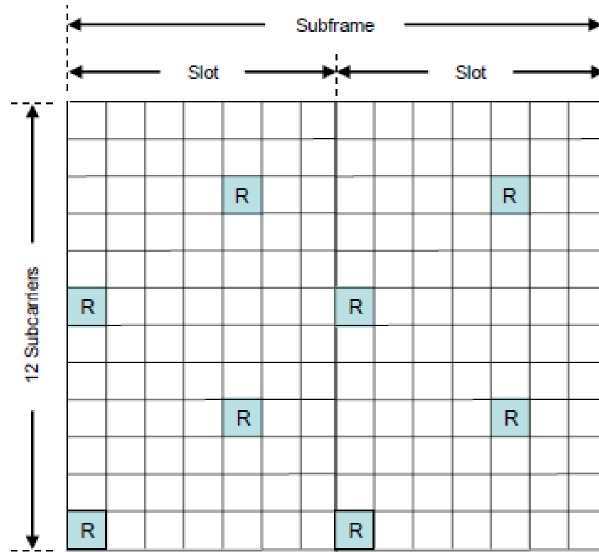


Figure 6.3 : Refrence signals positions in LTE frame.

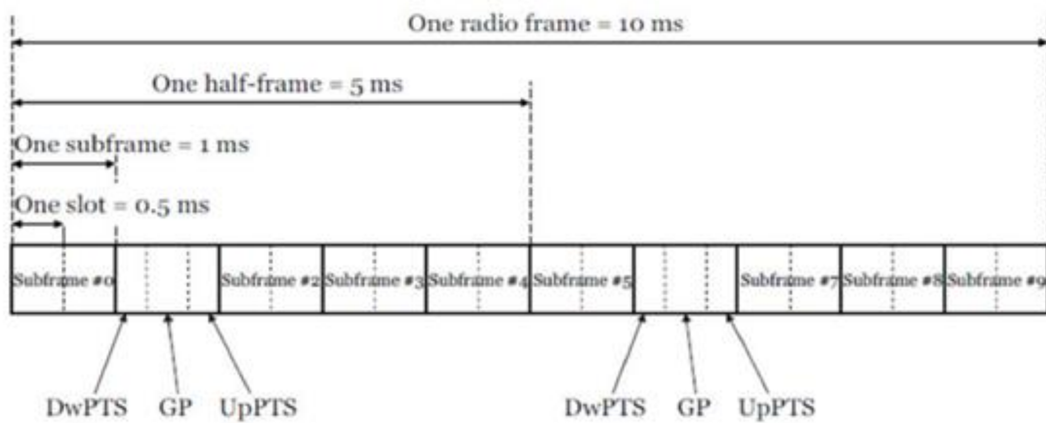


Figure 6.4 : Second LTE frame structure.

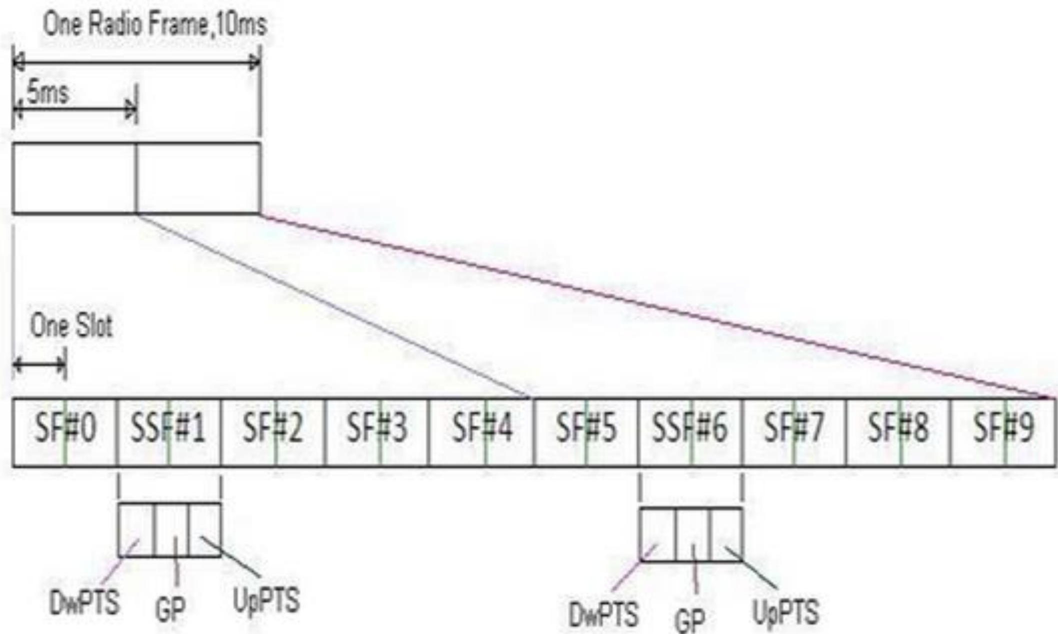


Figure 6.5 : LTE frame structure, TDD , Type-2.

LTE transmitter block diagram is shown in Figure 6.6 .

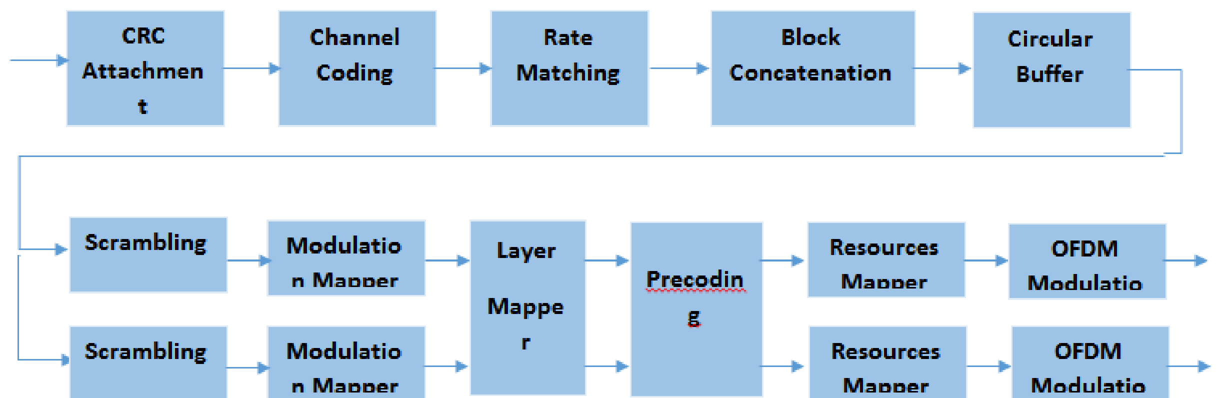


Figure 6.6 : Full LTE transmitter block diagram.

Since in our project we are working on a kit that holds only one port for antenna. Therefore, the main concept of LTE that consider in MIMO technique is not used and so the block diagram is eliminated to that shown in Figure 6.7 where we remove the (Layer Mapper and Precoding) blocks.

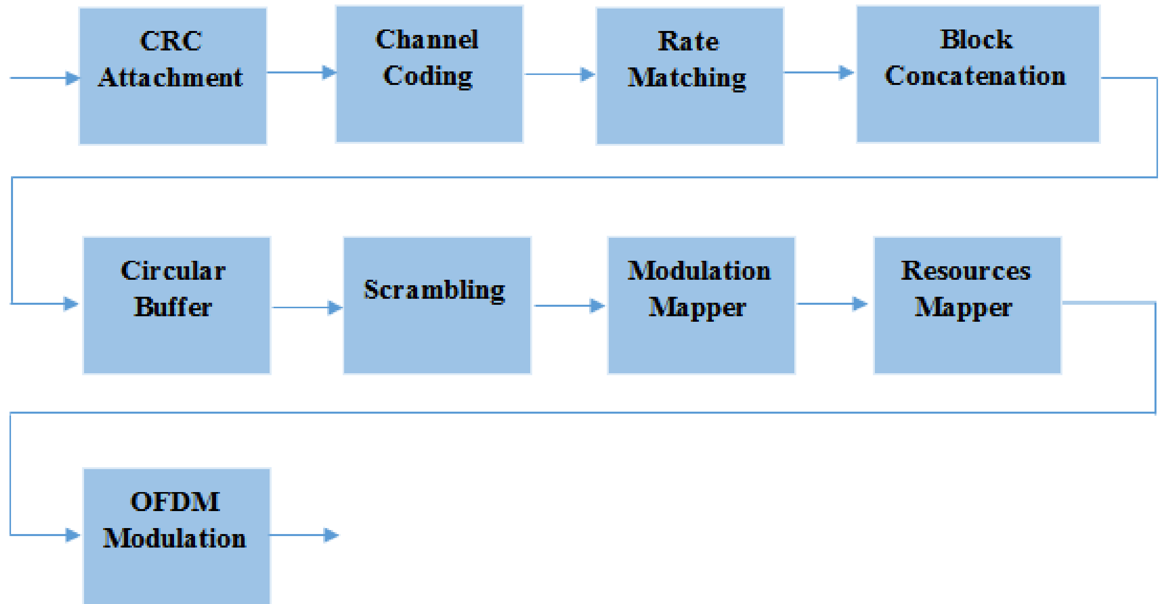


Figure 6.7 : Simplified LTE transmitter block diagram.

6.1 Scrambler

Scrambler is used to randomize the bits, prevent long sequences of 1s or 0s to keep synchronization. The scrambling sequence generator shall be initialized with

$$C_{init} = n_{RNTI} .2^{14} + q.2^{13} + floor(ns/2).2^{9} + N^{cell}$$

At the start of each subframe where n_{RNTI} corresponds to the RNTI associated with the PUSCH transmission.

6.2 Cyclic redundancy check (CRC)

The CRC block in LTE is the same as in 3G, stated in section 2.2.1 except the polynomial generator which is different than 3G. The polynomial generator equation is

$$g_{CRC24A}= D^{24} + D^{23} + D^{18} + D^{17} + D^{14} + D^{11} + D^{10} + D^{7} + D^{6} + D^{5} + D^{4} + D^{3} +D +1.$$

6.3 Code block segmentation and code block CRC attachment

The input bit sequence to the code block segmentation is denoted by:$b_0, b_1, b_2 \ldots\ldots b_{B-1}$.

If B is larger than the maximum code block size Z, segmentation of the input bit sequence is performed and an additional CRC sequence of L = 24 bits is attached to each code block.

The maximum code block size (Z) is equal 6144 and the minimum code block size (Z) is equal 40.

Total number of code blocks C is determined by the following algorithm:

if $B <= Z$

$L = 0$

Number of code blocks: $C = 1$

$B' = B$

else

$L = 24$

Number of code blocks: $C = \lceil B/(Z - L) \rceil$.

$B' = B + C.\ L$

end if

The bits output from code block segmentation, for $C != 0$, are denoted by: $C_{r0}, C_{r1}, C_{r2}, \ldots \ldots \ldots, C_{r(k_r - 1)}$ where r is the code block number and K_r is the number of bits for the code block number r.

First segmentation size: K_+ = minimum K in Table 6.1 such that $C.K > B'$

if $C = 1$

the number of code blocks with length K_+ is $C_+ = 1$, $K_- = 0$, $C_- = 0$

else if $C > 1$

Second segmentation size: K_- = maximum K in Table 6.1 such that $K < K_+$

$\Delta_K = K_+ - K_-$

Number of segments of size K_- : $C_- = \lfloor \frac{C.\,K_+ - B'}{\Delta_K} \rfloor$

Number of segments of size K_+ : $C_+ = C - C_-$

Blocks with size K_- are out first then Blocks with K_+

end if

Number of filler bits which added to the beginning of the first block: $F = C_+ . K_+ + C_- . K_- - B'$

if $B < 40$, filler bits are added to the beginning of the code block.

if $C > 1$ The sequence $C_{r0}, C_{r1}, C_{r2}, \dots\dots\dots, C_{r(k_r - L - 1)}$

is used to calculate the CRC parity bits $p_{r0}, p_{r1}, p_{r2}, \dots\dots\dots, p_{r(L-1)}$ with the generator polynomial $g_{CRC24B}(D) = 1 + D + D^5 + D^6 + D^{23} + D^{24}$.

Table 6.1-K Values

i	K	i	K	I	k
1	2	5	32	9	512
2	4	6	64	10	1024
3	8	7	128	11	2048
4	16	8	256	12	4096

This table is simplified version of full k table in standard to simplify the implementation of Segmentation and turbo encoder and also we assumed Z equal 4096 instead of 6144.

For example: if number of input data bits (B) =8000

the number of blocks (C)= $\lceil 8000 / (4096 - 24) \rceil = 2$

$B' = B + C.\,L = 8000 + 2 * 24 = 8048$

K_+ = minimum K in Table 6.1 such that $K > 4024 = 4096$

K_- = maximum K in Table 6.1 such that $K < 4096 = 2048$

$\Delta_K = 4096 - 2048 = 2048$

Number of segments of size K_- : $C_- = \left\lfloor \frac{8192 - 8048}{2048} \right\rfloor = 0$

Number of segments of size K_+ : $C_+ = 2 - 0 = 2$

$F = 2*4096 + 0*2048 - 8048 = 144$

First block (r=0)		
144 filler bits	3928 bits from b_0 to b_{3927}	24 crc bits
Second block (r=1)		
4072 bits from b_{3928} to b_{7999}		24 crc bits

Segmentation was implemented as a Finite state machine (FSM) and Figure 6.8 shows the state diagram of it and Table 6.2 shows the state description of its.

Table 6.2-Segmentation state description

State	Description
IDLE	Reset state and waiting bits from CRC
Save	Store bits in fifo and waiting number of bits from CRC
Calc	Calculate segmentation parameters like: C_+, C_-, K_-, K_+, F
Filler	Generate filler bits for output
First_oneblock	Generate rest of first block bits in case of one block (C=1 , B<Z)
First_cplus	Generate rest of first block bits in case of multi blocks (C>1) & $C_- = 0$
First_cminus	Generate rest of first block bits in case of multi blocks (C>1) & $C_- != 0$
Waiting	Waiting for the turbo encoder to be ready for next block.
Other_blocks	Generate other blocks

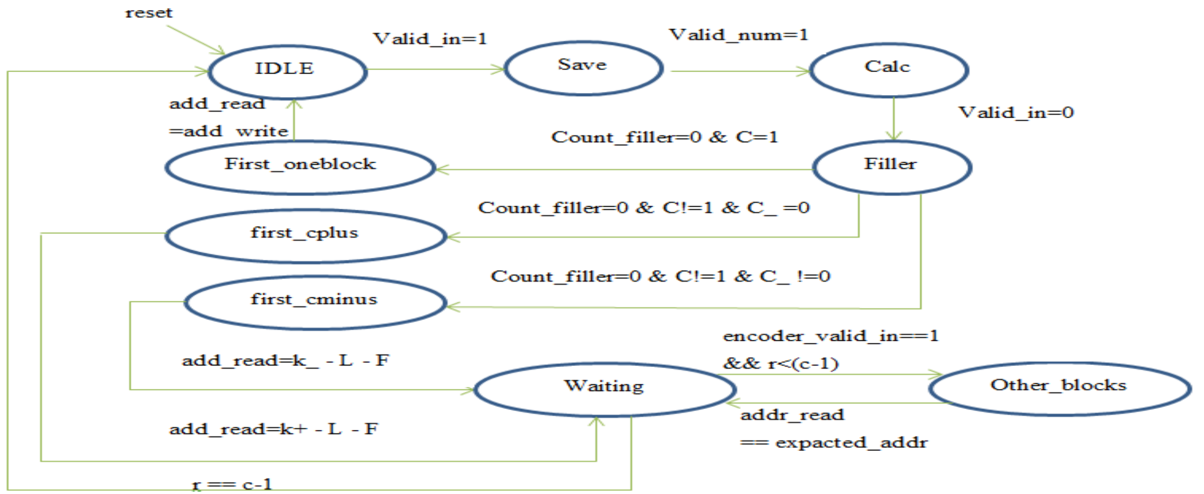


Figure 6.8: Segmentation State Diagram

6.4 Turbo encoder

The scheme of turbo encoder is a Parallel Concatenated Convolutional Code (PCCC) with two 8-state constituent encoders and one turbo code internal interleaver.

The coding rate of turbo encoder is 1/3. The structure of turbo encoder is illustrated in. The transfer function of the 8-state constituent code for the PCCC is:

$$G(D)= \left[1, \frac{g_1(d)}{g_0(d)}\right],$$

Where $g_0(D)$= 1+D^2+D^3

$g_1(D)$= 1+D+D^3

The initial value of the shift registers of the 8-state constituent encoders shall be all zeros when starting to encode the input bits.

The output from the turbo encoder is

$$d_k^{(0)} = x_k$$

$$d_k^{(1)} = z_k$$

$$d_k^{(2)} = z'_k$$

For k= 0,1,2, ..., K-1. If the code block to be encoded is the 0-th code block and the number of filler bits is greater than zero, i.e., $F > 0$, then

the encoder shall set c_k, = 0, k = 0,...,(F-1) at its input and shall set $d_k^{(0)}$= $NULL$, k = 0,...,(F-1) and $d_k^{(1)}$= $NULL$, k = 0,...,(F-1) at its output.

The bits input to the turbo encoder are denoted by c_0 , c_1 , c_2 , c_3 ,...,$c_{K\ 1}$, and the bits output from the first and second 8- state constituent encoders are denoted by z_0, z_1, z_2, z_3,….,z_{k-1} and z'_0, z'_1, z'_2, z'_3,….., z'_{K-1}respectively. The bits output from the turbo code internal interleaver are denoted byc'_0, c'_1, c'_2, c'_3,…., c'_{k-1} and these bits are to be the input to the second 8-state constituent encoder.

6.4.1 Trellis termination for turbo encoder

Trellis termination is performed by taking the tail bits from the shift register feedback after all information bits are encoded. Tail bits are padded after the encoding of information bits.

The first three tail bits shall be used to terminate the first constituent encoder (upper switch of Figure 6.9 in lower position) while the second constituent encoder is disabled. The last three tail bits shall be used to terminate the second constituent encoder (lower switch of Figure 6.9 in lower position) while the first constituent encoder is disabled.

The transmitted bits for trellis termination shall then be:

$d_k^{(0)} = x_k$, $d_{k+1}^{(0)} = z_{k+1}$, $d_{k+2}^{(0)} = x'_k$,$d_{k+3}^{(0)} = z'_{k+1}$

$d_k^{(1)} = z_k$, $d_{k+1}^{(1)} = x_{k+2}$, $d_{k+2}^{(1)} = z'_k$,$d_{k+3}^{(1)} = x'_{k+1}$

$d_k^{(2)} = x_{k+1}$, $d_{k+1}^{(2)} = z_{k+2}$, $d_{k+2}^{(2)} = x'_{k+1}$,$d_{k+3}^{(2)} = z'_{k+2}$

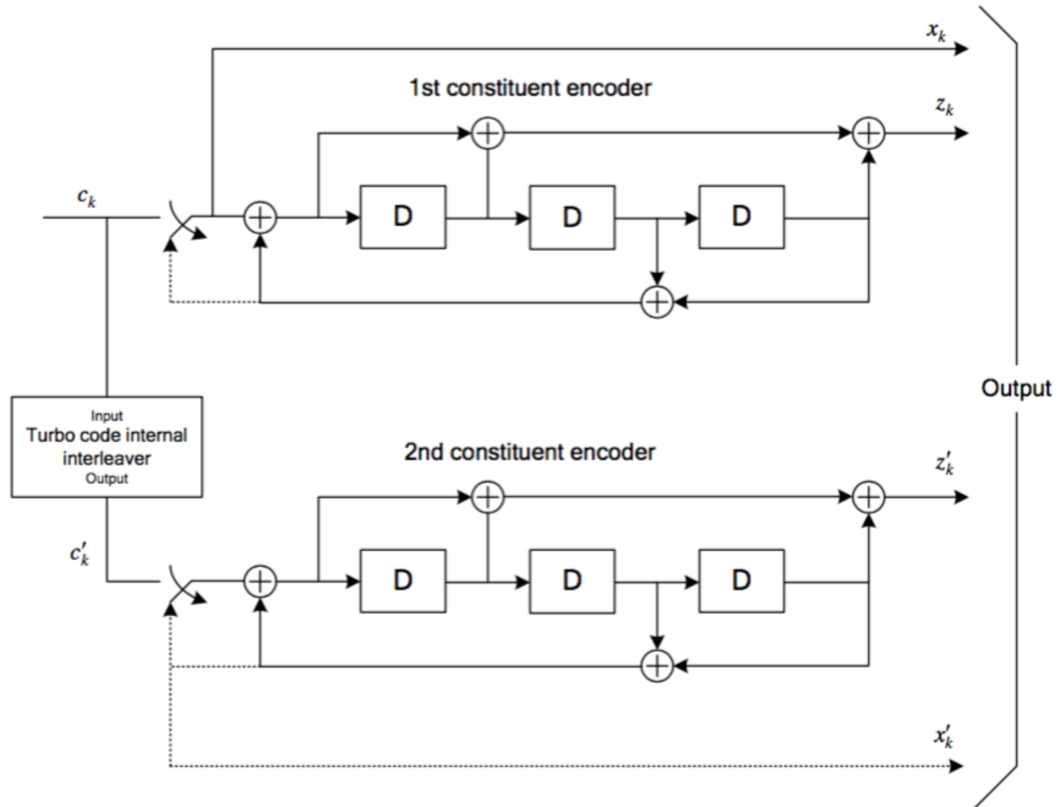


Figure 6.9. Structure of rate 1/3 turbo encoder

6.4.2 Turbo Code Internal Interleaver

The bits input to the turbo code internal interleaver are denoted by c_0 , c_1 ,..., c_K, where K is the number of input bits.

The bits output from the turbo code internal interleaver are denoted by c'_0, c'_1, c'_2, c'_3,…., c'_{k-1}.

The relationship between the input and output bits is as follows:

$c'_i = c_{\prod(i)}$, i=0, 1,..., (K-1)

where the relationship between the output index i and the input index $\prod(i)$ satisfies the following quadratic form: $\prod(i) = (f_1 . i + f_1 . i^2)$mod K.

The parameters depend on the block size K and are summarized in Figure 6.11.

N.B: We implemented the turbo encoder with even K numbers thus the division would be synthesizable.

Schematic show the top module of the turbo encoder rate third.

The input and output ports indicated below in Figure 6.10.

Detailed block diagram for the blocks that shown internal construction of the turbo encoder in Figure 6.12.

N.B: C&R signals just pass through the turbo encoder block.

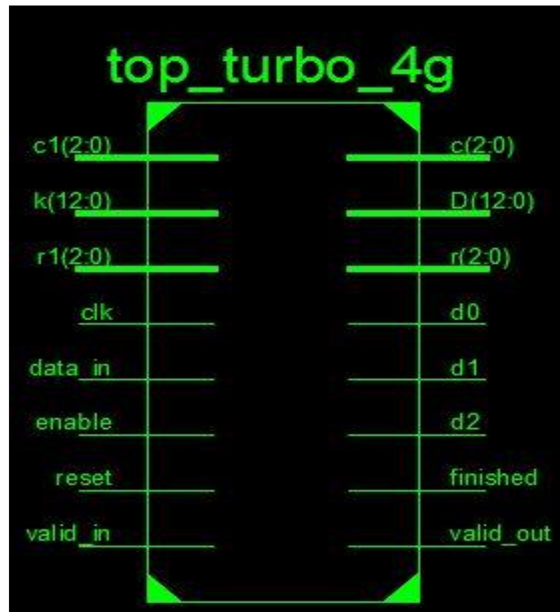


Figure 6.10:Schematic of turbo encoder

i	*K*	f_1	f_2	*i*	*K*	f_1	f_2	*i*	*K*	f_1	f_2	*i*	*K*	f_1	f_2
1	40	3	10	48	416	25	52	95	1120	67	140	142	3200	111	240
2	48	7	12	49	424	51	106	96	1152	35	72	143	3264	443	204
3	56	19	42	50	432	47	72	97	1184	19	74	144	3328	51	104
4	64	7	16	51	440	91	110	98	1216	39	76	145	3392	51	212
5	72	7	18	52	448	29	168	99	1248	19	78	146	3456	451	192
6	80	11	20	53	456	29	114	100	1280	199	240	147	3520	257	220
7	88	5	22	54	464	247	58	101	1312	21	82	148	3584	57	336
8	96	11	24	55	472	29	118	102	1344	211	252	149	3648	313	228
9	104	7	26	56	480	89	180	103	1376	21	86	150	3712	271	232
10	112	41	84	57	488	91	122	104	1408	43	88	151	3776	179	236
11	120	103	90	58	496	157	62	105	1440	149	60	152	3840	331	120
12	128	15	32	59	504	55	84	106	1472	45	92	153	3904	363	244
13	136	9	34	60	512	31	64	107	1504	49	846	154	3968	375	248
14	144	17	108	61	528	17	66	108	1536	71	48	155	4032	127	168
15	152	9	38	62	544	35	68	109	1568	13	28	156	4096	31	64
16	160	21	120	63	560	227	420	110	1600	17	80	157	4160	33	130
17	168	101	84	64	576	65	96	111	1632	25	102	158	4224	43	264
18	176	21	44	65	592	19	74	112	1664	183	104	159	4288	33	134
19	184	57	46	66	608	37	76	113	1696	55	954	160	4352	477	408
20	192	23	48	67	624	41	234	114	1728	127	96	161	4416	35	138
21	200	13	50	68	640	39	80	115	1760	27	110	162	4480	233	280
22	208	27	52	69	656	185	82	116	1792	29	112	163	4544	357	142
23	216	11	36	70	672	43	252	117	1824	29	114	164	4608	337	480
24	224	27	56	71	688	21	86	118	1856	57	116	165	4672	37	146
25	232	85	58	72	704	155	44	119	1888	45	354	166	4736	71	444
26	240	29	60	73	720	79	120	120	1920	31	120	167	4800	71	120
27	248	33	62	74	736	139	92	121	1952	59	610	168	4864	37	152
28	256	15	32	75	752	23	94	122	1984	185	124	169	4928	39	462
29	264	17	198	76	768	217	48	123	2016	113	420	170	4992	127	234
30	272	33	68	77	784	25	98	124	2048	31	64	171	5056	39	158
31	280	103	210	78	800	17	80	125	2112	17	66	172	5120	39	80
32	288	19	36	79	816	127	102	126	2176	171	136	173	5184	31	96
33	296	19	74	80	832	25	52	127	2240	209	420	174	5248	113	902
34	304	37	76	81	848	239	106	128	2304	253	216	175	5312	41	166
35	312	19	78	82	864	17	48	129	2368	367	444	176	5376	251	336
36	320	21	120	83	880	137	110	130	2432	265	456	177	5440	43	170
37	328	21	82	84	896	215	112	131	2496	181	468	178	5504	21	86
38	336	115	84	85	912	29	114	132	2560	39	80	179	5568	43	174
39	344	193	86	86	928	15	58	133	2624	27	164	180	5632	45	176
40	352	21	44	87	944	147	118	134	2688	127	504	181	5696	45	178
41	360	133	90	88	960	29	60	135	2752	143	172	182	5760	161	120
42	368	81	46	89	976	59	122	136	2816	43	88	183	5824	89	182
43	376	45	94	90	992	65	124	137	2880	29	300	184	5888	323	184
44	384	23	48	91	1008	55	84	138	2944	45	92	185	5952	47	186
45	392	243	98	92	1024	31	64	139	3008	157	188	186	6016	23	94
46	400	151	40	93	1056	17	66	140	3072	47	96	187	6080	47	190
47	408	155	102	94	1088	171	204	141	3136	13	28	188	6144	263	480

Figure 6.11. Turbo code internal interleaver parameters

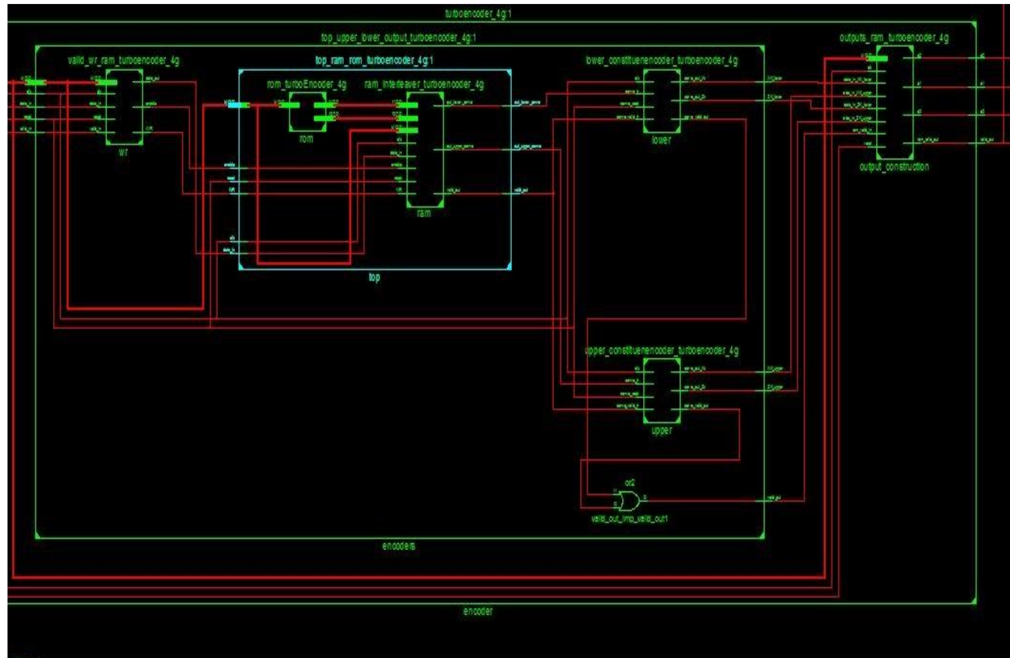


Figure 6.12. Internal block diagram

Table 6.3 : Inputs and outputs of turbo encoder.

Pin	Pin Direction	Description
C_in(2:0)	Input	Number of code blocks from the segmentation
K(12:0)	Input	Block size of data
R1(2:0)		Block index
Clk	Input	Clock of the all encoder blocks
Data_in	Input	Data in for the convolutional encoder
Enable	Input	Working enable for the encoder
Reset	Input	Reset encoder registers and interleaver by inserting zeros.
Valid_in	Input	Valid in to consider the input
C(2:0)	Output	Indicates the number of code blocks
D(12:0)	Output	Block size after turbo encoder
R(2:0)	Output	Block index
d0	Output	Turbo output(0)
d1	Output	Turbo output(1)
d2	Output	Turbo output(2)
Finished	Output	Signal indicates the block is ready for the new frame of data

Valid_out	Output	Valid out to the next block

6.5 Rate matching for turbo coded transport channels

The rate matching for turbo coded transport channels is defined per coded block and consists of interleaving the three information bit streams $d_k^{(0)}$, $d_k^{(1)}$ and $d_k^{(2)}$, followed by the collection of bits and the generation of a circular buffer as depicted in Figure 6.13.

The bit stream $d_k^{(0)}$ is interleaved according to the sub-block interleaver defined in section 6.5.1 with an output sequence defined as $v_0^{(0)}, v_1^{(0)}, v_2^{(0)}, \ldots \ldots . v_{K_\pi-1}^{(0)}$.

The bit stream $d_k^{(1)}$ is interleaved according to the sub-block interleaver defined in section 6.5.1 with an output sequence defined as $v_0^{(1)}, v_1^{(1)}, v_2^{(1)}, \ldots \ldots . v_{K_\pi-1}^{(1)}$.

The bit stream $d_k^{(2)}$ is interleaved according to the sub-block interleaver defined in section 6.5.1 with an output sequence defined as $v_0^{(2)}, v_1^{(2)}, v_2^{(2)}, \ldots \ldots . v_{K_\pi-1}^{(2)}$.

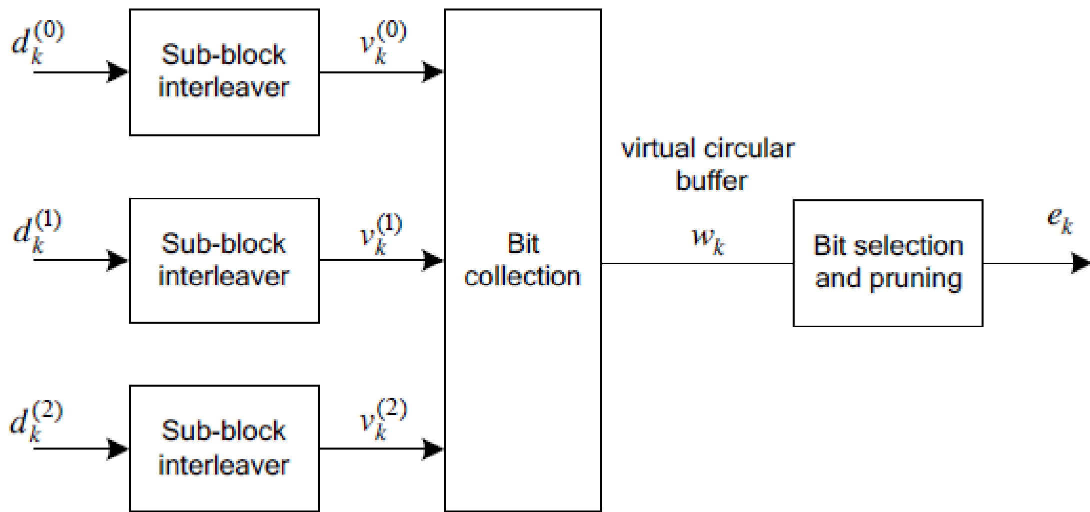


Figure 6.13: Rate matching for turbo coded transport channels.

6.5.1 Sub-block interleaver

The bits input to the block interleaver are denoted by $d_0^{(i)}, d_1^{(i)}, d_2^{(i)}, \ldots \ldots, d_{D-1}^{(i)}$,where D is the number of bits. The output bit sequence from the block interleaver is derived as follows:

1. Assign $C_{subblock}^{TC} = 32$ to be the number of columns of the matrix. The columns of the matrix are numbered $0,1,2,\ldots\ldots, C_{subblock}^{TC} - 1$ from left to right.
2. Determine the number of rows of the matrix $R_{subblock}^{TC}$, by finding minimum integer $R_{subblock}^{TC}$ such that:

$$D \leq R_{subblock}^{TC} * C_{subblock}^{TC}$$

The rows of rectangular matrix are numbered $0,1,2,\ldots\ldots, R_{subblock}^{TC} - 1$ from top to bottom.

3. $R_{subblock}^{TC} * C_{subblock}^{TC} > D$, then $N_D = R_{subblock}^{TC} * C_{subblock}^{TC} - D$ dummy bits are padded such that y_k = <NULL> for $k = 0,1,2,\ldots\ldots, N_D - 1$. Then, $y_{N_{D+k}} = d_k^{(i)}$, k = 0, 1,…, D-1, and the bit sequence y_k is written into the $R_{subblock}^{TC} * C_{subblock}^{TC}$ matrix row by row starting with bit y0 in column 0 of row 0:

$$\begin{bmatrix} y_0 & y_1 & y_2 & \cdots & y_{C_{subblock}^{TC}-1} \\ y_{C_{subblock}^{TC}} & y_{C_{subblock}^{TC}+1} & y_{C_{subblock}^{TC}+2} & \cdots & y_{2C_{subblock}^{TC}-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{(R_{subblock}^{TC}-1)\times C_{subblock}^{TC}} & y_{(R_{subblock}^{TC}-1)\times C_{subblock}^{TC}+1} & y_{(R_{subblock}^{TC}-1)\times C_{subblock}^{TC}+2} & \cdots & y_{(R_{subblock}^{TC}\times C_{subblock}^{TC}-1)} \end{bmatrix}$$

For $d_k^{(0)}$ and $d_k^{(1)}$:

4. Perform the inter-column permutation for the matrix based on the pattern P(j) that is shown in Table 6.4, where P(j) is the original column position of the j-th permuted column. After permutation of the columns, the inter-column permuted ($R_{subblock}^{TC} * C_{subblock}^{TC}$) matrix is equal to

$$\begin{bmatrix} y_{P(0)} & y_{P(1)} & y_{P(2)} & \cdots & y_{P(C_{subblock}^{TC}-1)} \\ y_{P(0)+C_{subblock}^{TC}} & y_{P(1)+C_{subblock}^{TC}} & y_{P(2)+C_{subblock}^{TC}} & \cdots & y_{P(C_{subblock}^{TC}-1)+C_{subblock}^{TC}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{P(0)+(R_{subblock}^{TC}-1)\times C_{subblock}^{TC}} & y_{P(1)+(R_{subblock}^{TC}-1)\times C_{subblock}^{TC}} & y_{P(2)+(R_{subblock}^{TC}-1)\times C_{subblock}^{TC}} & \cdots & y_{P(C_{subblock}^{TC}-1)+(R_{subblock}^{TC}-1)\times C_{subblock}^{TC}} \end{bmatrix}$$

5. The output of the block interleaver is the bit sequence read out column by column from the inter-column permuted ($R_{subblock}^{TC} * C_{subblock}^{TC}$) matrix. The bits after sub-block interleaving are denoted by $v_0^{(i)}, v_1^{(i)}, v_2^{(i)}, \ldots\ldots\ldots v_{K_\pi-1}^{(i)}$ where $v_0^{(i)}$ corresponds to $y_{p(0)}$, $v_1^{(i)}$ corresponds to $y_{p(0)+C_{subblock}^{TC}}$ ... and $k_\pi = (R_{subblock}^{TC} * C_{subblock}^{TC})$.

For $d_k^{(2)}$:

1. The output of the sub-block interleaver is denoted by $v_0^{(2)}, v_1^{(2)}, v_2^{(2)}, \ldots\ldots\ldots v_{K_\pi-1}^{(2)}$, where $v_k^{(2)} = y_{\pi(k)}$ and where

$$\pi(k) = \left(P\left(Floor\left(\frac{k}{R^{TC}_{subblock}}\right)\right) + C^{TC}_{subblock} * (k \bmod R^{TC}_{subblock}) + 1\right) \bmod k_{\pi}$$

The permutation function P is defined in Table 6.4.

Table 6.4:Inter-column permutation pattern for sub-block interleaver.

Number of columns $C^{TC}_{subblock}$	Inter-column permutation pattern $< P(0), P(1), \ldots, P(C^{TC}_{subblock}-1) >$
32	< 0, 16, 8, 24, 4, 20, 12, 28, 2, 18, 10, 26, 6, 22, 14, 30, 1, 17, 9, 25, 5, 21, 13, 29, 3, 19, 11, 27, 7, 23, 15, 31 >

The block interface is as shown in Figure 6.14 and the signals declaration and description is as shown in Table 6.5.

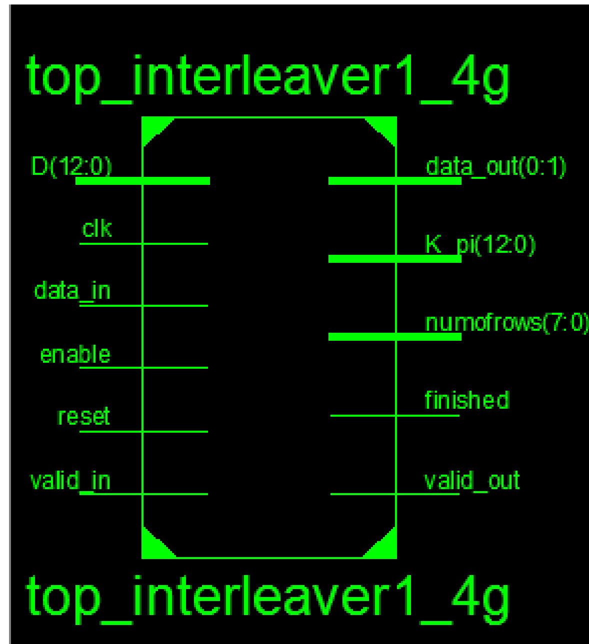


Figure 6.14: Interleaver block interface.

Table 6.5:Interleaver block signals declaration

PIN	PIN Type	Description
enable	IN	This signal indicates that the next block is ready to have data
D	IN	This signal indicates the number of bits in the code-block
valid_in	IN	This signal indicates that current data_in is valid data
data_in	IN	The input bits
K_pi	OUT	This signal indicates the number of output bits (multiple from 32)
numofrows	OUT	This signal indicates the number of rows of the matrix $R^{TC}_{subblock}$
finished	OUT	This signal indicates that the interleaver is ready to have a new frame

valid_out	OUT	This signal indicates that current data_out is valid data
data_out	OUT	The output bits

K_pi and numofrows output signals are required in the bit selection block to be used in a certain calculations so to prevent any conflict we are going to transfer these signals with each code-block through the bit collection block to be sure that each code-block at the bit selection will be associated with the correct information.

The output signal data_out can't be one bit only because we want to represent the dummy bits, so as shown in Figure 6.14 that data_out is two bits and the representation is as following:

When the output bit is 0 therefore data_out = 00

When the output bit is 1 therefore data_out = 01

When the output bit is dummy (x) therefore data_out = 10

6.5.2 Bit collection

The input bits to the bit collection block is the output bits from the three sub-block interleaver and the block output can be represented by virtual circular buffer as shown in Figure 6.13.

The circular buffer of length $K_w = 3K_\pi$ for the r-th coded block is generated as follows:

$W_k = v_k^{(0)}$ For $k = 0,1, \dots\dots K_\pi - 1$

$W_{K_\pi + 2k} = v_k^{(1)}$ For $k = 0,1, \dots\dots K_\pi - 1$

$W_{K_\pi + 2k+1} = v_k^{(2)}$ For $k = 0,1, \dots\dots K_\pi - 1$

The block interface is as shown in Figure 6.15 and the signals declaration and description is as shown in Table 6.6.

As shown in Figure 6.15 and Table 6.6 that the signals K_pi and numofrows are passed through the block without any modifications (as a buffer) because these signals

are required in the bit selection block to be used in a certain calculations as mentioned before.

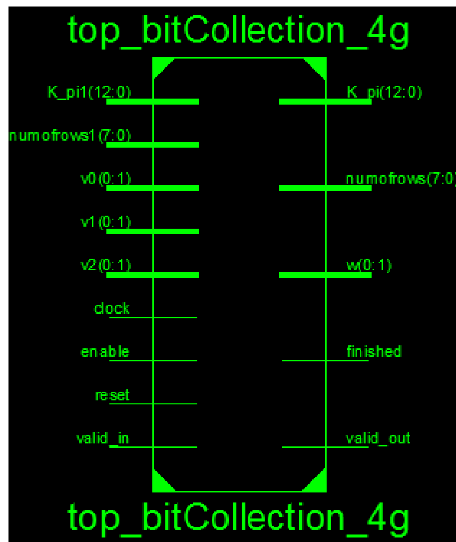


Figure 6.15:Bit collection block interface.

Table 6.6:Bit collection block signals declaration

PIN	PIN Type	Description
enable	IN	This signal indicates that the next block is ready to have data
valid_in	IN	This signal indicates that current data_in is valid data
v0,v1,v2	IN	The input bits that received from each interleaver
K_pi1	IN	This signal indicates the number of output bits from the interleaver (multiple from 32)
numofrows1	IN	This signal indicates the number of rows of the matrix $R_{subblock}^{TC}$
finished	OUT	This signal indicates that the interleaver is ready to have a new frame
valid_out	OUT	This signal indicates that current data_out is valid data
W	OUT	The output bits
K_pi	OUT	This signal indicates the number of output bits from the interleaver (multiple from 32)
numofrows	OUT	This signal indicates the number of rows of the matrix $R_{subblock}^{TC}$

6.5.3 Bit selection

It is the last block in the block diagram of the rate matching, the block has its input from the output of the bit collection as shown in Figure 6.13 and it is used to remove the dummy bits from the bit collection output according to the following calculations:

Denote the soft buffer size for the r-th code block by N_{cb} bits. For UL-SCH, MCH, SL-SCH and SL-DCH transport channels $N_{cb} = K_w$.

Define by G the total number of bits available for the transmission of one transport block.

$$G = N_{prb}.N_{rb_{sc}}.\left(N_{ul_{symb}} - 1\right).2.Q_m.N_L$$

- N_{prb} is the number of resource blocks which is assigned to UE
- $N_{rb_{sc}} = 12$
- $N_{ul_{symb}} = 7$
- Q_m is the number of bits per symbol
- N_L is the number of layers

Denoting by E the rate matching output sequence length for the r-th coded block, and rv_{idx} the redundancy version number for this transmission (rv_{idx} = 0), the rate matching output bit sequence is e_k , k = 0,1,..., E-1.

Set $G' = G/(N_L.Q_m)$ where Q_m is equal to 2 for QPSK, 4 for 16QAM, 6 for 64QAM and 8 for 256QAM and where N_L= 2 for transmit diversity.

Set $\gamma = G' \ mod\ C$, where C is the number of code blocks (segmentation section).

If $r \leq C - \gamma - 1$ Set $E = N_L.Q_m.Floor(G'/C)$

else Set $E = N_L.Q_m.ceil(G'/C)$

endif

Set $k_0 = R^{TC}_{subblock}.\left(2.ceil\left(\frac{N_{cb}}{8R^{TC}_{subblock}}\right).rv_{idx} + 2\right)$

Set $k = 0\ and\ j = 0$

while k < E

If $w_{(k_0+j)mod\ N_{cb}} \neq <NULL>$

$e_k = w_{(k_0+j)mod\ N_{cb}}$

$k = k + 1$

endif

$j = j + 1$

end while

The block interface is as shown in Figure 6.16 and the signals declaration and description is as shown in Table 6.7 .

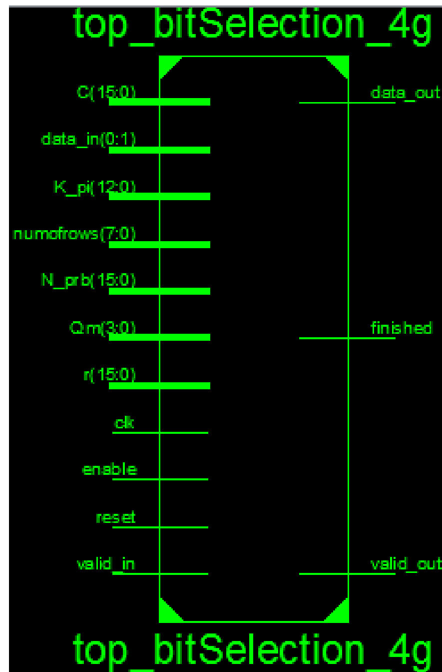


Figure 6.16: Bit selection block interface.

Table 6.7:Bit selection block signals declaration

PIN	PIN Type	Description
Enable	IN	This signal indicates that the next block is ready to have data
valid_in	IN	This signal indicates that current data_in is valid data
data_in	IN	The input bits that received from each interleaver
C	IN	This signal indicates the number of code blocks
K_pi	IN	This signal indicates the number of output bits from the interleaver (multiple from 32)
numofrows	IN	This signal indicates the number of rows of the matrix $R_{subblock}^{TC}$
N_prb	IN	This signal indicates the number of resource blocks (from MAC layer)
Qm	IN	This signal indicates the number of bits per symbol
R	IN	This signal indicates the index number of a code block
finished	OUT	This signal indicates that the interleaver is ready to have a new frame
valid_out	OUT	This signal indicates that current data_out is valid data
W	OUT	The output bits

Finally the block diagram of the rate matching block will be as shown in Figure 6.17 and the block interface will be as shown in Figure 6.18.

6.6 Code block concatenation

As stated in section 4.2.4

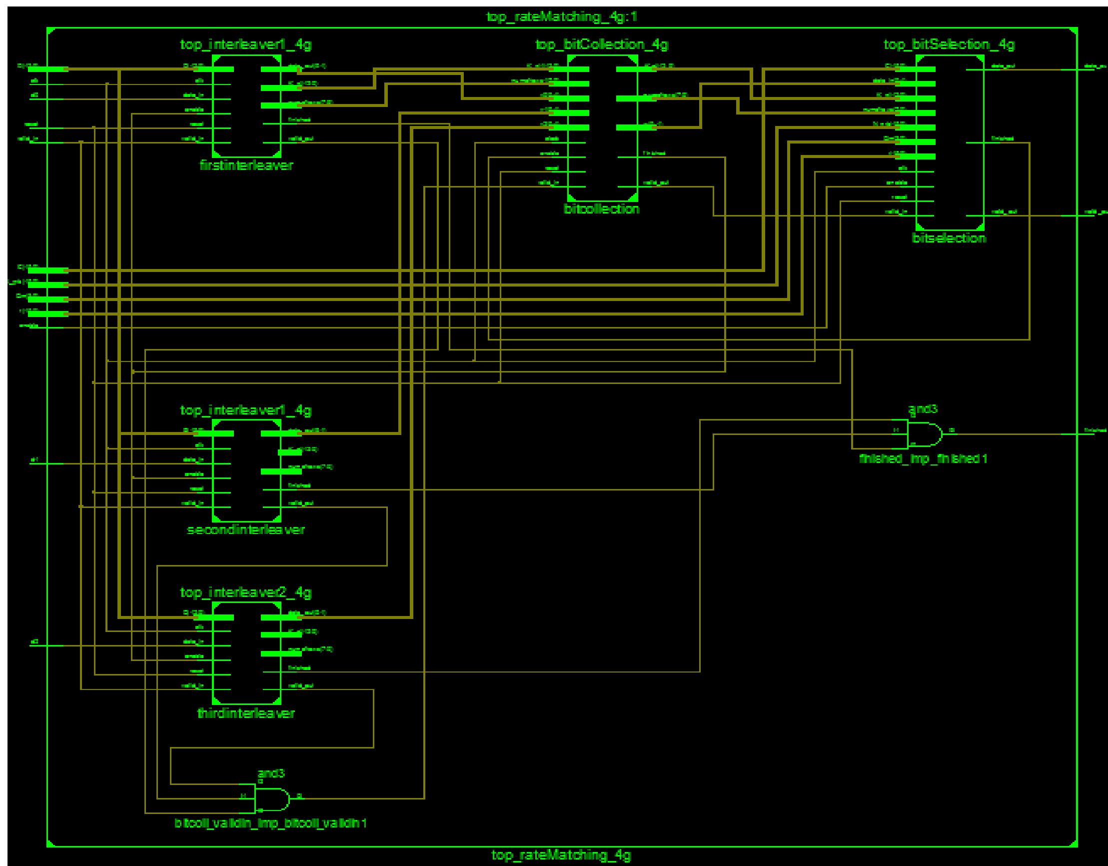


Figure 6.17: Rate matching block diagram.

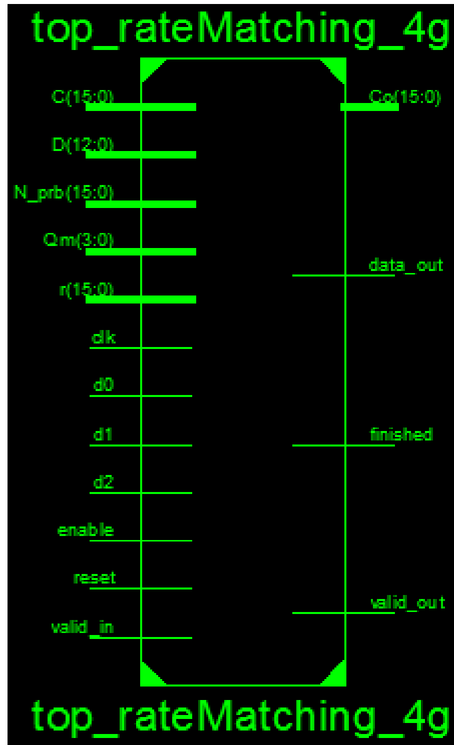


Figure 6.18:Rate matching block interface

Table 6.8: Rate Matching block signals declaration

PIN	PIN Type	Description
enable	IN	This signal indicates that the next block is ready to have data
valid_in	IN	This signal indicates that current data_in is valid data
d0,d1,d2	IN	The input bits that received from each interleaver
C	IN	This signal indicates the number of code blocks
D	IN	This signal indicates the number of bits in the code-block
K_pi	IN	This signal indicates the number of output bits from the interleaver (multiple from 32)
numofrows	IN	This signal indicates the number of rows of the matrix $R_{subblock}^{TC}$
N_prb	IN	This signal indicates the number of resource blocks (from MAC layer)
Qm	IN	This signal indicates the number of bits per symbol
r	IN	This signal indicates the index number of a code block
Co	OUT	This signal indicates the number of code blocks
finished	OUT	This signal indicates that the interleaver is ready to have a new frame
valid_out	OUT	This signal indicates that current data_out is valid data
w	OUT	The output bits

the signal C is passed through the block without any modifications (as a buffer) because this signal is required in the code block concatenation block.

6.7 Divider

As shown in the previous blocks (Rate matching, interleaver, ….etc) that there are some equations that need to be modeled to be able to write a synthesizable HDL code. Division is the most common problem.

6.7.1 Sequential Divider

The sequential divider works at the raising/falling edge of the clock, the sequential divider takes many clock cycles to generate output which complicates our RTL, and we need faster divider.

Combinational Divider

The combinational divider works without clock because its RTL is combinational so it is more better that the sequential divider because it generates output within the same clock cycle.

Ceil divider simulation is as shown in Figure 19 and Floor divider simulation is as shown in Figure 20.

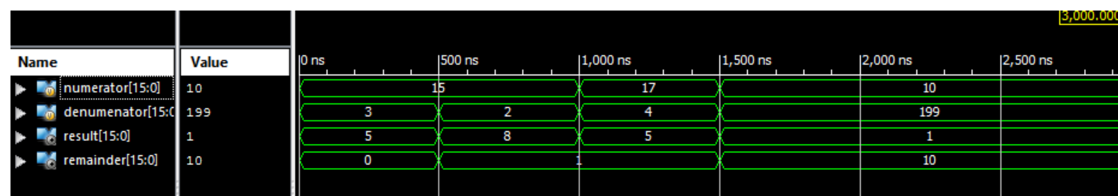


Figure 19: Ceil divider simulation.

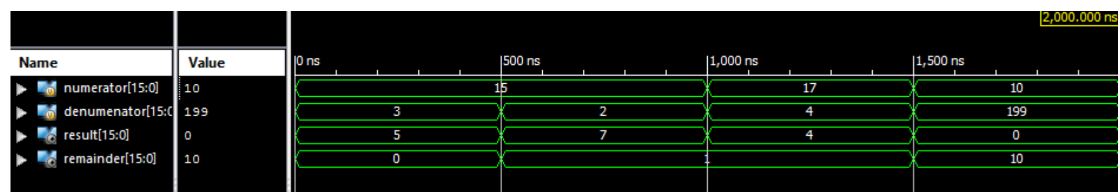


Figure 20: Floor divider simulation

6.8 Modulation

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 LTE standard Phase Shift Keying (BPSK, QPSK) and Quadrature Amplitude Modulation (16-QAM, 64-QAM, 256-QAM) modulation techniques are used according to the desired data rate. The bits are mapped to complex-valued modulation symbol d= **(I + j Q)** In BPSK, a single bit is mapped to a complex-valued modulation symbol according to Table 6.9.

Table 6.9: BPSK mapping

$b(i)$	*I*	*Q*
0	$1/\sqrt{2}$	$1/\sqrt{2}$
1	$-1/\sqrt{2}$	$-1/\sqrt{2}$

In QPSK, pairs of bits are mapped to complex-valued modulation symbols x=I+jQ according to Table 6.10.

Table 6.10: QPSK mapping

$b(i), b(i+1)$	*I*	*Q*
00	$1/\sqrt{2}$	$1/\sqrt{2}$
01	$1/\sqrt{2}$	$-1/\sqrt{2}$
10	$-1/\sqrt{2}$	$1/\sqrt{2}$
11	$-1/\sqrt{2}$	$-1/\sqrt{2}$

In 16-QAM, quadruplets of bits are mapped to complex-valued modulation symbols x=I+jQ according to Table 6.11.

Concerning the HDL implementation, Figure 6.21 shows the interface of the mapper as shown in the figure every symbol is represented in 12 bits – this number is

determined through a simulation will be discussed later- .the pins description is in Table 6.12.

The internal block diagram is shown in Figure 6.22.

It consists of fifo to store the input bit stream, controller to control the fifo and the mapper module (top_mod_wifi) which consists of serial to parallel inverter and the mapper module that mapps bits to the corresponding symbol following the constellation.

Table 6.11: 16-QAM mapping

$b(i), b(i+1), b(i+2), b(i+3)$	I	Q
0000	$1/\sqrt{10}$	$1/\sqrt{10}$
0001	$1/\sqrt{10}$	$3/\sqrt{10}$
0010	$3/\sqrt{10}$	$1/\sqrt{10}$
0011	$3/\sqrt{10}$	$3/\sqrt{10}$
0100	$1/\sqrt{10}$	$-1/\sqrt{10}$
0101	$1/\sqrt{10}$	$-3/\sqrt{10}$
0110	$3/\sqrt{10}$	$-1/\sqrt{10}$
0111	$3/\sqrt{10}$	$-3/\sqrt{10}$
1000	$-1/\sqrt{10}$	$1/\sqrt{10}$
1001	$-1/\sqrt{10}$	$3/\sqrt{10}$
1010	$-3/\sqrt{10}$	$1/\sqrt{10}$
1011	$-3/\sqrt{10}$	$3/\sqrt{10}$
1100	$-1/\sqrt{10}$	$-1/\sqrt{10}$
1101	$-1/\sqrt{10}$	$-3/\sqrt{10}$
1110	$-3/\sqrt{10}$	$-1/\sqrt{10}$
1111	$-3/\sqrt{10}$	$-3/\sqrt{10}$

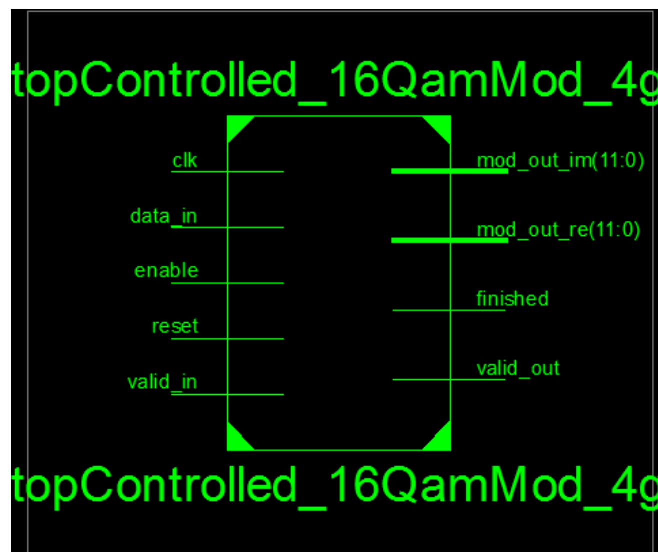


Figure 6.21: LTE mapper interface

Table 6.12: Pin discription od LTE mapper module

PIN	Description
Data_in	The input bits
Valid_in	The signal indicates that current data_in is valid data

Mod_out_Re	The modulated real part of the input
Mod_out_im	The modulated imaginary part of the input
finished	The signal indicated that the mapper is ready for the new frame
Enable	The signal indicates that the next block is ready to have data

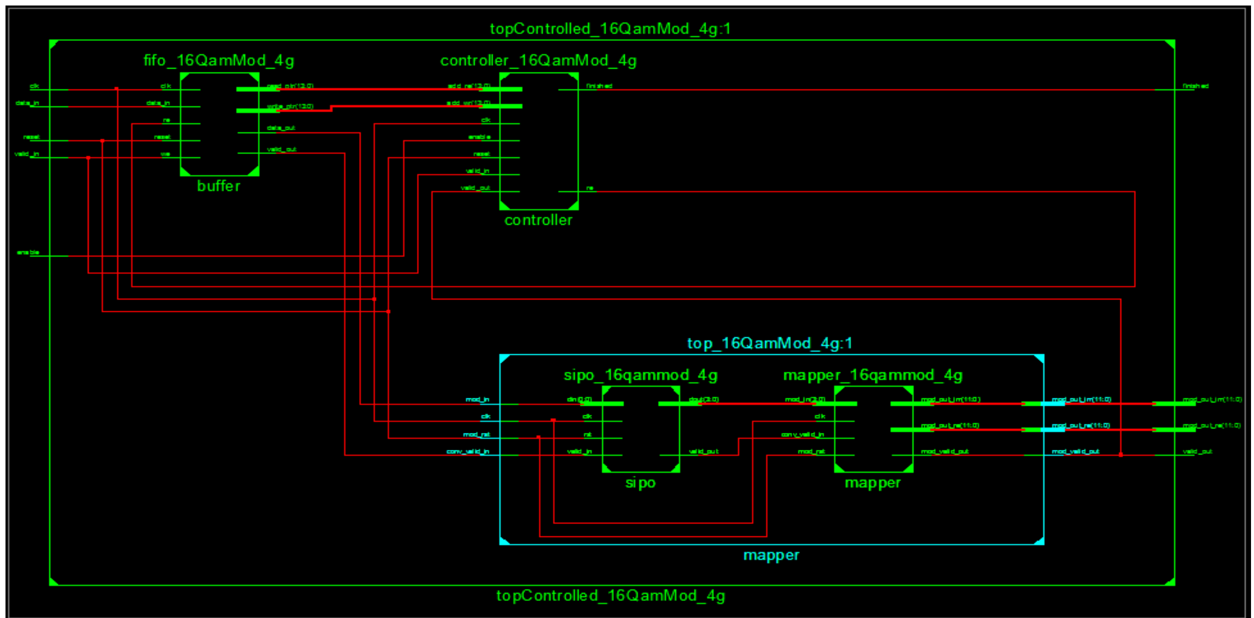


Figure 6.22: Detailed block diagram of LTE mapper module

6.9 IFFT

LTE uplink uses SC-FDMA which is a modified form of the OFDM with similar throughput performance and complexity, SC-FDMA is viewed as DFT-coded OFDM where time-domain symbols are transformed to frequency domain symbols and then go through the standard OFDM modulation, SC-FDMA has all the advantages of OFDM like robustness against multi-path signal propagation, the block diagram for the SC-FDMA is shown in Figure 6.23. [4]

Figure 6.23: Transmitter block diagram [4]

The IFFT subcarriers are grouped into sets of 12 subcarrier, each group is called a resource block.

The main advantage of SC-FDMA is the low Peak Average Power Ratio (PAPR) of the transmit signal, PAPR is a big concern for user equipment, as PAPR relates to the power amplifier efficiency as low PAPR allows the power amplifier to operate close to the saturation region resulting in high efficiency that is why SC-FDMA is the preferred technology for user terminals.

The LTE supported bandwidths are as shown in Table 6.13 [5], we chose the 128 point IFFT to facilitate our testing process, we also chose the extended cyclic prefix so for our 128 point IFFT we have 32 cyclic prefix and our sampling frequency is 1.92 MHz, the pin interface for the our top_ofdm_4g is as shown Figure 6.24, the pin description is the same as what we will explain later in WIFI the different pins are start_rb and num_of_rbs, start_rb defines the starting resource block that will be used, and num_of_rbs defines the number of resource blocks to use and the IFFT_clk is 1.92 MHz.

In our implementation we used Xilinx IP LogiCORE Discrete Fourier Transform, [6] the pin interface is shown in Figure 6.25, the DFT module transform size is reconfigurable by the pin called size, the desired transformation size is decided as the required number of resource blocks The core indicates that it is ready to accept a new frame of data by setting RFFD high. When RFFD is high, data input may be started by setting FD_IN high for one or more cycles. Data is input via XN_RE and XN_IM. It should be provided over N cycles without interruption. Data input and output are complex and in natural order. FD_OUT signals when the core starts data output and DATA_VALID signals when data on XK_RE and XK_IM is valid.

Note that FD_IN is ignored while RFFD is low, and so FD_IN can be kept high for multiple cycles. FD_IN is accepted on the first cycle that RFFD is high, if FD_IN is set permanently high, then the core will start a new frame of data input as soon as the core is ready, this arrangement provides maximum transform throughput. Alternatively, RFFD may be connected directly to FD_IN to achieve the same behavior. [6]

The first element of input data should be provided on the same cycle that the core starts to receive data, that is, the first cycle in which both FD_IN and RFFD are high, the IFFT module is later explained in WIFI.

The encoding of the size parameter that we used is shown in Figure 6.26. [6]

Table 6.13: Basic transmission schemes [5]

Transmission Bandwidth	1.4 MHz	3 MHz	5 MHz	10 MHz	15 MHz	20 MHz
Sampling frequency	1.92 MHz	3.84 MHz	7.68 MHz	15.36 MHz	23.04 MHz	30.72 MHz
FFT Size	128	256	512	1024	1536	2048
#RBs (12 subcarriers)	6	15	25	50	75	100

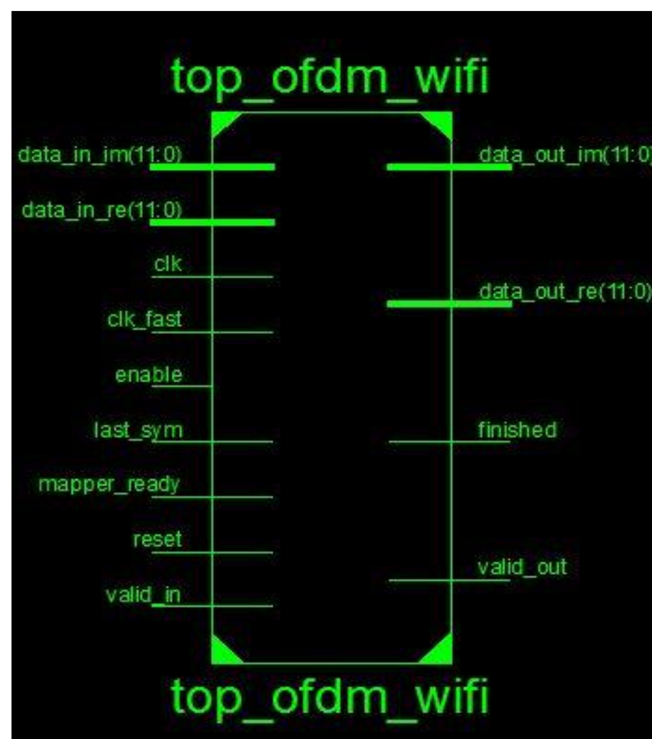


Figure 6.24: top_ofdm_wifi interface

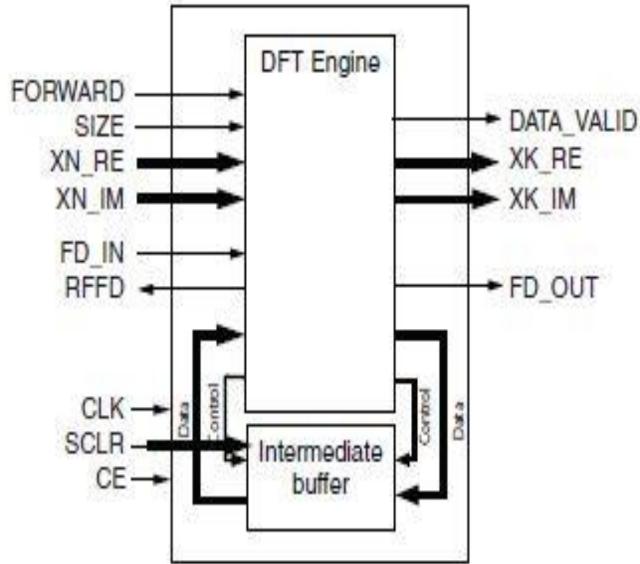


Figure 6.25: DFT interface [6]

Size (Binary)	N	M (Radix-2)	P (Radix-3)	Q (Radix-5)	Latency C_L Cycles	Period C_T Cycles	Time to Process 1200 Points, µs (at 245.76 MHz)
0	12	2	1		75	62	25.28
1	24	3	1		122	109	22.22
2	36	2	2		152	139	18.90
3	48	4	1		176	163	16.63

Figure 6.26: Encoding of size parameter [6]

Our top_ofdm_4g module contains a 48 register memory, reconfigurable DFT module and 128-point IFFT module, when the mapper is ready to send data we buffer the symbols in the memory then we set the DFT size to the required size, the we start inputting symbols to the DFT and after it finishes it writes its output to the memory then we input the symbols for the IFFT. Each LTE frames consists of 6 sub frames, the third sub frame of each frame is dedicated for demodulation reference signal, in our implementation we assumed the demodulation reference signal to be all ones, and it should be improved in later designs.

Chapter 7: Verification Methodology

7.1 Functional Verification

Functional Verification is very important to check that HDL implementations outputs are identical to expected outputs, we implemented MATLAB models for all transmitter and receiver blocks to use as a reference models.

Verification methodology is based on equivalence checking between HDL Model and MATLAB Model, we setup testing framework consists of Verilog testbench, MATLAB testbench and Perl script.

Figure 7.1 shows the functional verification procedure; first Verilog testbench generates random input bits and store it in a file and also save the output data bits from all blocks in different files, MATLAB testbench read the input file and also save the output data bits from all blocks in different files, Perl script compare these files and generate output comparison file.

To cover different cases we change number of bits and also number of successive frames to check that every block can reset its state after frame finished and process next frame correctly.

Figure 7.2 shows the different generated files from both testbenches, for example outputfile14_hdlModel means that this file contains output from HDL model of fourth block in chain and first frame.

Figure 7.3 Shows the comparison output file from Perl script, first columns represents bit number (index) starting from zero, second column represents bit value from hdl model, third column represents bit value from MATLAB model, fourth column represents result for this bit whether correct which indicates they are equal or wrong which indicates they are not equal and finally after all bits there are number of correct bits and number of wrong bits.

-

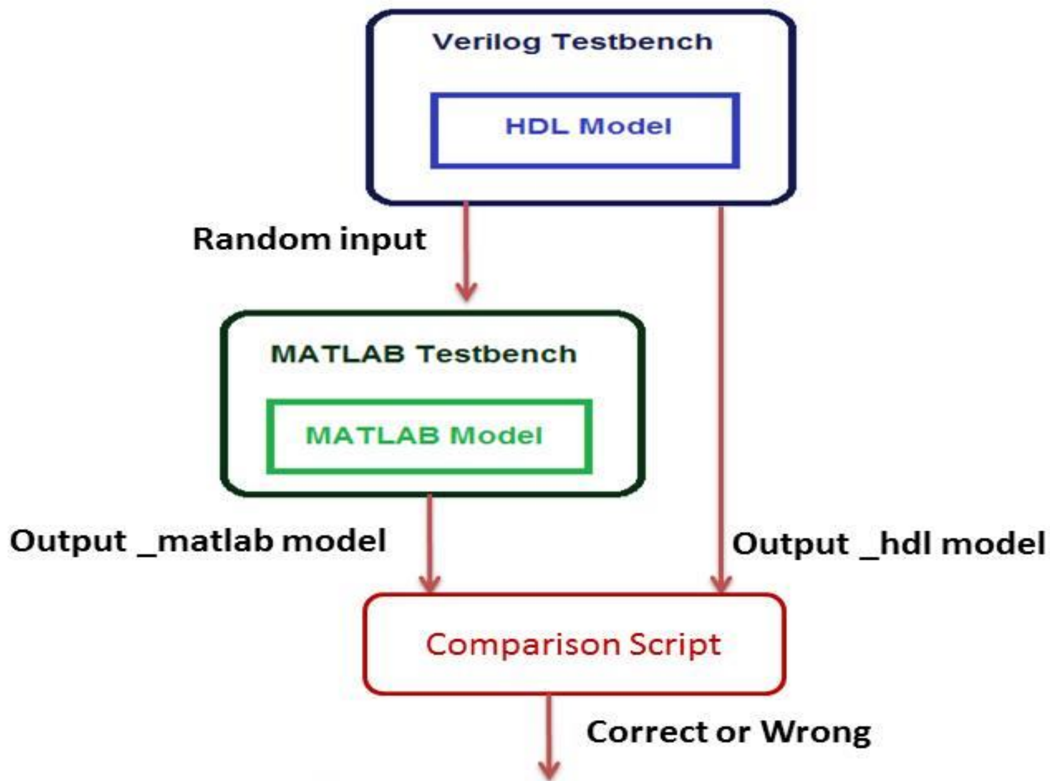


Figure 7.1-functional verification procedure

outputfile11_hdlModel.txt
outputfile11_matlabModel.txt
outputfile12_hdlModel.txt
outputfile12_matlabModel.txt
outputfile13_hdlModel.txt
outputfile13_matlabModel.txt
outputfile14_hdlModel.txt
outputfile14_matlabModel.txt
outputfile15_hdlModel.txt
outputfile15_matlabModel.txt
outputfile16_hdlModel.txt
outputfile16_matlabModel.txt
outputfile17_hdlModel.txt
outputfile17_matlabModel.txt
outputfile18_hdlModel.txt
outputfile18_matlabModel.txt

outputfile21_hdlModel.txt
outputfile21_matlabModel.txt
outputfile22_hdlModel.txt
outputfile22_matlabModel.txt
outputfile23_hdlModel.txt
outputfile23_matlabModel.txt
outputfile24_hdlModel.txt
outputfile24_matlabModel.txt
outputfile25_hdlModel.txt
outputfile25_matlabModel.txt
outputfile26_hdlModel.txt
outputfile26_matlabModel.txt
outputfile27_matlabModel.txt
outputfile28_hdlModel.txt
outputfile28_matlabModel.txt

Figure 7.2 : testbenches generated files

```
case number  verilog MATLAB comparison
 0  0   0   correct
1  1   1   correct
2  1   1   correct
3  1   1   correct
4  1   1   correct
5  1   1   correct
6  1   1   correct
7  0   0   correct
8  1   1   correct
9  1   1   correct
511  1   1   correct
512  0   0   correct
513  1   1   correct
514  0   0   correct
515  1   1   correct
correct : 516
wrong : 0
```

Figure 7.3-compariosn file

7.2 IFFT testing

We set up a small framework using MATLAB to test our IFFT results, the test block diagram is simple we use MATLAB script to generate random modulated input like the mapper output and then set it to a fixed point representation, the output file MATLAB script is used by the IFFT test bench to feed the IFFT with the test data then the IFFT output is dumped to a file, the MATLAB script performs the same operation of our IFFT module like symbol arrangement and pilot insertion then performs

the IFFT operation then the MATLAB output is converted to fixed-point representation of 12-bit then compares the two outputs and computes average error, our average error in WIFI is -73 dB, The block diagram of the testing process is shown in Figure 7.4.

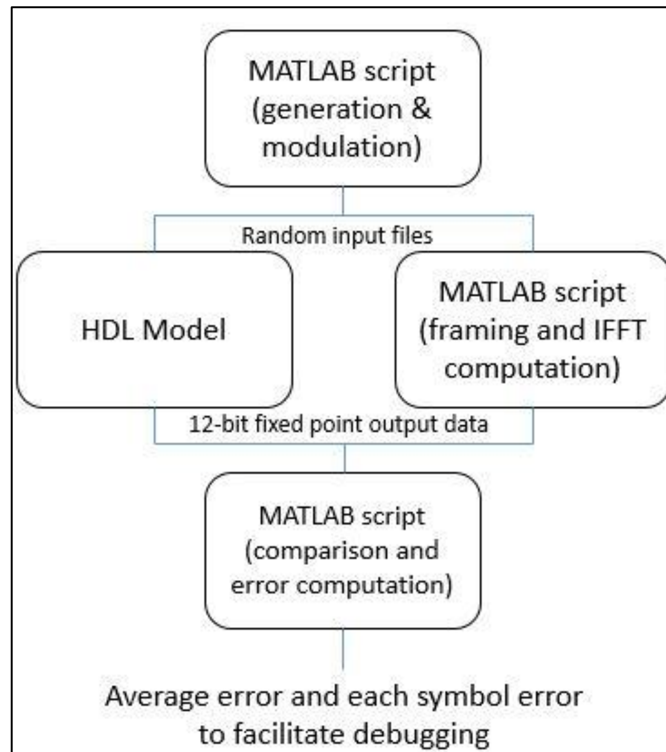


Figure 7.4: IFFT testing framework process

The framing, IFFT computation, comparison and error computation scripts are integrated in the complete chain test bench.

The framing and IFFT computation part is in LTE_mod_func.m for LTE and Wifi_mod_func.m for WIFI, the comparison and error computation parts are in the top chain test bench.. An example for the final output of the testing framework is shown Figure 7.5.

7.3 Post-Synthesis Simulation

Synthesis is transformation from RTL to gate level and generate the netlist level.

Because of bad coding style, there may be mismatch between Pre-synthesis simulation and Post –synthesis simulation so we built Post-synthesis simulation models and rerun the testing framework to check that there is no mismatch between Pre-synthesis simulation and Post–synthesis simulation.

```
Command Window
Error computations:

--------------------

   Total average error:

     Absolute Value: 0.025129
     Error in dB    : -73.674897
--------------------

Symbol Error computations:

--------------------

   Symbol 1 average error:

     Absolute Value: 0.018332
     Error in dB    : -79.981814
--------------------

   Symbol 2 average error:

     Absolute Value: 0.019898
     Error in dB    : -78.343202
--------------------

   Symbol 3 average error:

     Absolute Value: 0.020944
     Error in dB    : -77.318344
--------------------

   Symbol 4 average error:
```

Figure 7.5: MATLAB testbench output for a WIFI chain

7.4 Fixed Point Simulation

Starting from the output of the mapper to the end of any chain, we deal with soft values not bits. This lightens the way for us to determine number of digits representing each bit.

We have modeled the transmitter and receiver chain and the AWGN noise of the channel using MATLAB, calculated the BER when using floating point operations. Then we implemented MATLAB functions (dec2fix, dec2twos, fix2dec ,twos2dec) that convert numbers from decimal to binary and vice versa to be able to convert fractions and negative numbers –not only integers as in MATLAB built-in functions-. Using these functions we could calculate BER using different number of bits and compare it with the BER of the floating point to choose the number of digits representing each bit to get an acceptable BER. These results are shown in Figure 7.6, Figure 7.7. In Figure 7.6, simulation was done to determine number of bits representing the integer part. Figure 7.7, simulation was done to determine number of bits representing the fraction part. These simulation results that we use 9 bits for fraction part and 3 bits for integer part.

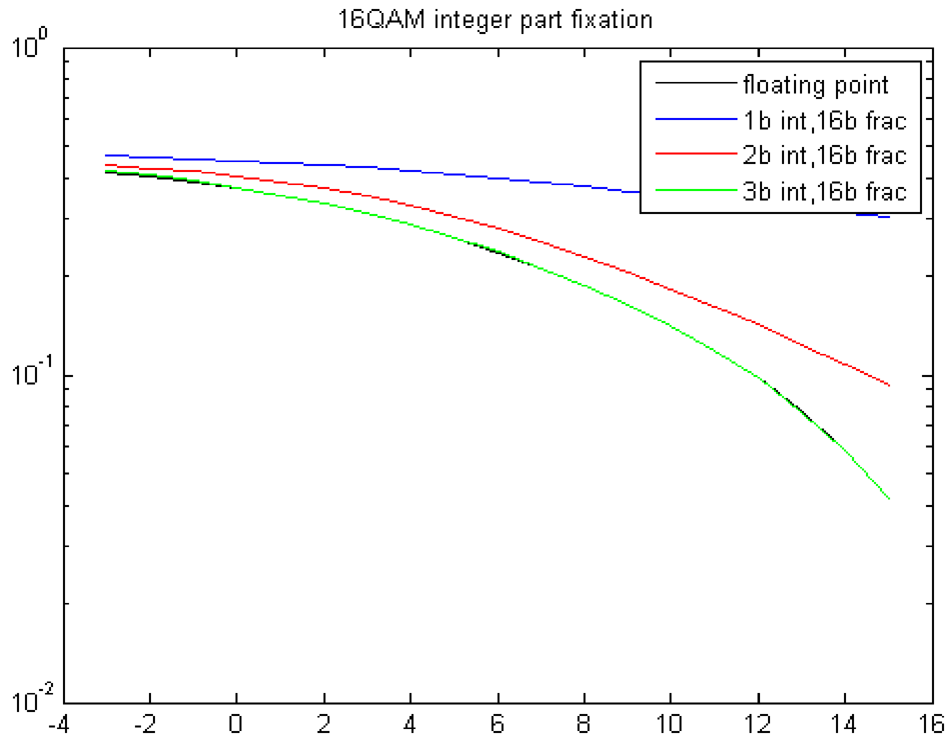


Figure 7.6 : Results of 16-QAM integer part fixation.

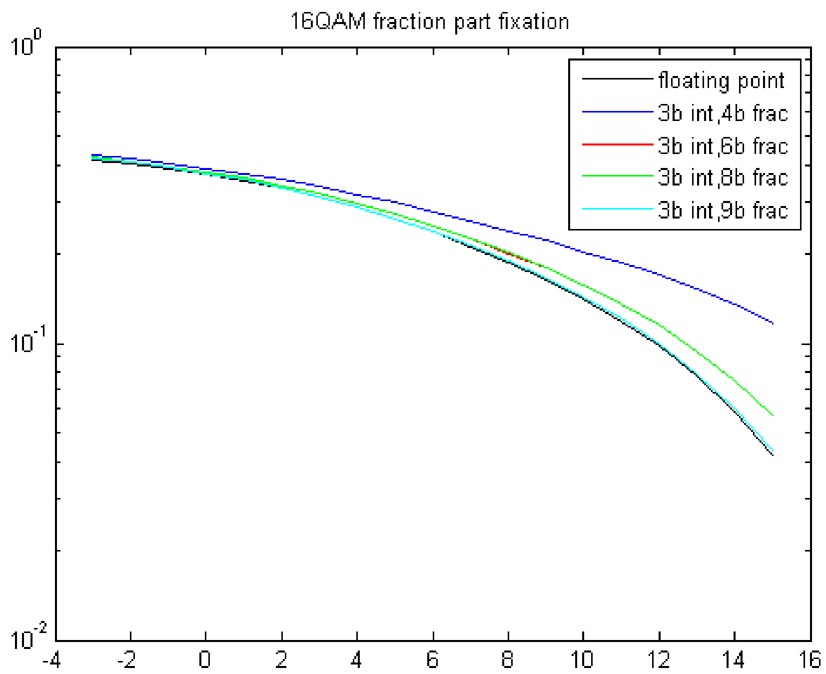


Figure 7.7 : Results of 16-QAM fraction part fixation.

Chapter 8: Hardware implementation

8.1 FPGA

The FPGA is an Integrated Circuit (IC) that is electrically programmed to execute a certain application. It initially has no functionality to operate before it is programmed. FPGA is formed from a combination of transistors that are connected in a specific way. Applying an external voltage to these transistors it will operate certain functionality. This combination of transistors called Look up Tables (LUTs). Each group of LUTs forms a Programmable Logic Blocks (PLB). These PLB blocks have been developed through many years. Recent FPGAs has different types of PLB functionality such as memory blocks that can store data for internal operations, multipliers for complex arithmetic operations, and general PLBs that is used to implement general functions from simple 2-bit adder to a complete microprocessor unit. The internal heterogeneity of FPGA PLBs is shown in Figure 8.1 . The FPGA internal routing consists of wires and programmable switches that allow the connections among the PLBs, memory blocks, multipliers and I/O ports. These connections are developed for best data routing and latency, sometimes with different characteristics varies from the shortest path to the fastest one. Also, there is a dedicated network of connections that takes care of clock distribution and reset signals for achieving low skew.

The LUT size is measured by its number of inputs such as an LUT has 3 inputs will be named as 3-LUT. The number of LUTs in the PLB may be of equal size or mixture of different sizes. There are three different major techniques used to program the FPGA LUTs: Anti-Fuse Flash, look up table and SRAM programming technologies. The advantages of the Anti-Fuse and Flash over the SRAM, they are non-volatile and occupies a small area. While the SRAM is easily reprogrammed and use the standard CMOS process technology so SRAM has become the dominant approach to program the FPGA LUTs ,but till now there is no technique that can combine the best of them all.

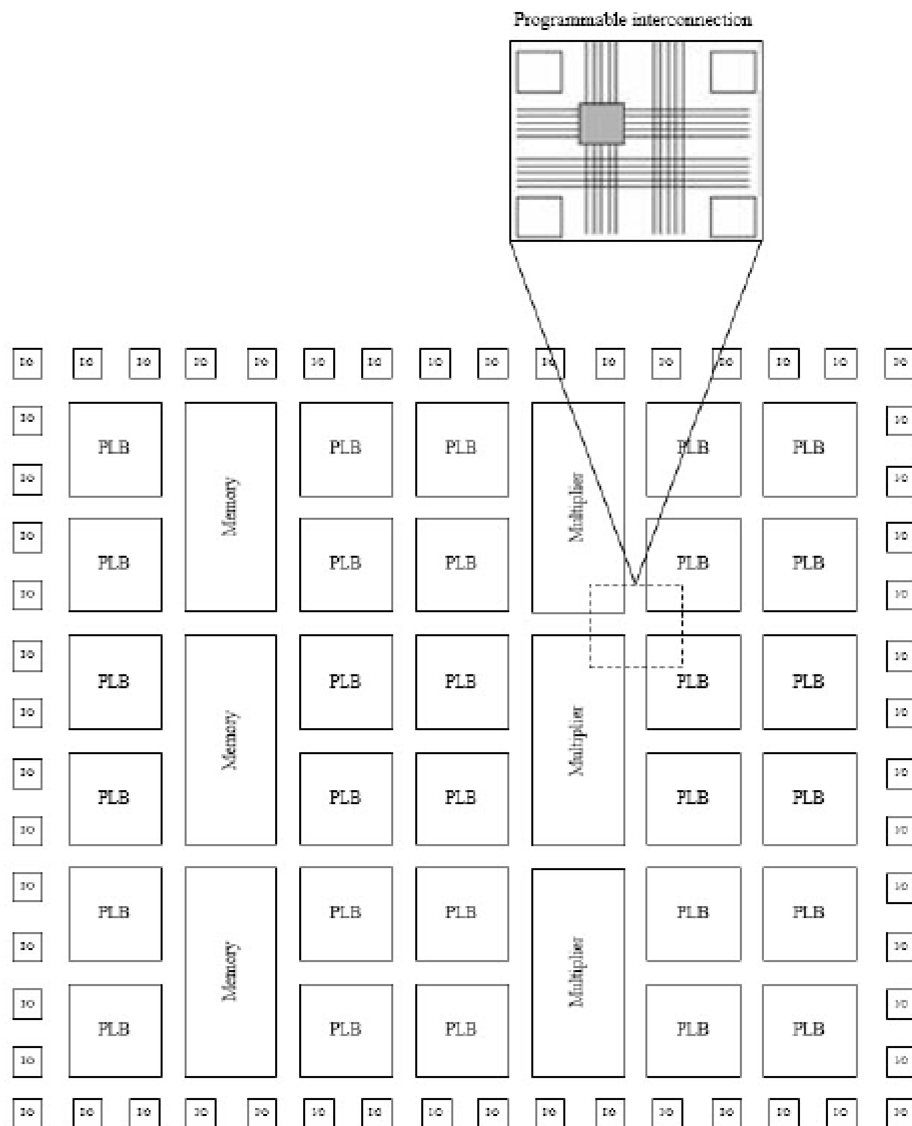


Figure 8.1: FPGA internal structure

8.1.1 Softcore and Hardcore processors

Current FPGAs has IP blocks, these IPs are standard libraries which are optimized and developed to facilitate the FPGA development. An engineer can drag and drop certain functionality instead of building the new block from scratch. IPs like accumulators, bus interfaces, encoders … etc. The microprocessors are considered one of the important IP core. There are two types of microprocessors, softcore and hardcore. The softcore processor like MicroBlaze by Xilinx is implemented using the FPGA logic gates. The hardcore processor like PowerPC by IBM is fabricated in the core of the IC of the FPGA chip and connected to FPGA fabric as shown in Figure 8.2.

The main concern of the softcore processor is its limitation in speed, around 200 MHz, also, it takes many resources on the FPGA. Where there are some advantages of using softcore processor like modifying it for specific requirements, customizing instructions and multiple core system. On the other hand, using hardcore processor can achieve higher processing speeds more than 1GHz. Hence, the hardcore processor has its own fabric in the FPGA chip it doesn't occupy resources on the FPGA fabric which allows the full usage for the FPGA. The disadvantage of the hardcore is its fixed architecture that can't be modified. Zynq series by Xilinx is a perfect example of the current SoC chips; it combines ARM dual-core or quad-core microprocessor in a processing system (PS) with Xilinx FPGA fabric as a Programmable Logic (PL).

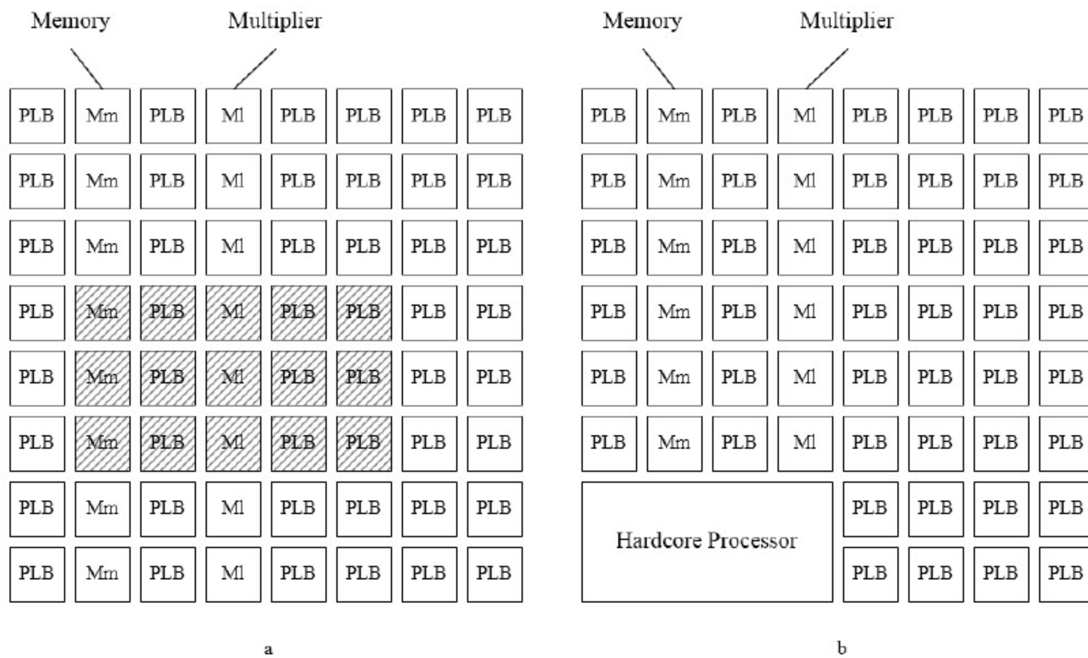


Figure 8.2: Softcore and Hardcore processor

As shown in Figure 8.2 (a) that the shaded part represent the implementation of softcore processor on the FPGA logic it acquires some of the available resources like PLBs, memory and multiplier blocks and as shown in Figure 8.2 (b) Hardcore processor fabricated beside the FPGA fabric.

8.1.2 Xilinx Virtex-5

An example to the FPGA that is generally introduced in section 8.1, Xilinx FPGA Virtex-5-XC5VLX110T that is shown in Figure 8.3. It is FPGA chip from Xilinx Virtex-5 series. Xilinx is a major FPGA vendor of market share 50%.

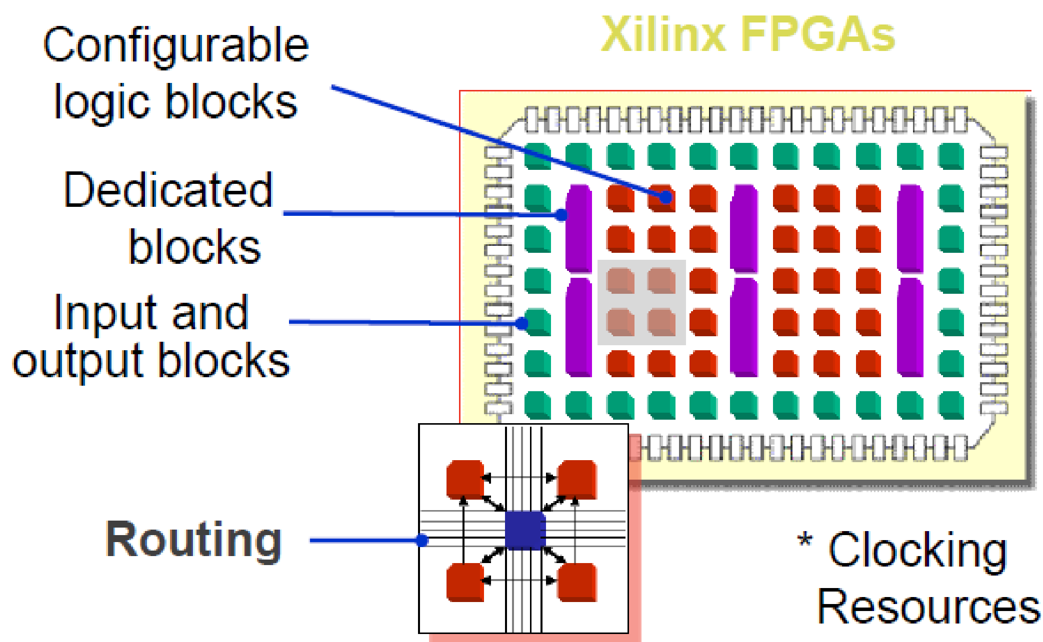


Figure 8.3: Xilinx FPGA

Configurable Logic Blocks (CLBs): The Configurable Logic Blocks (CLBs) are the main programmable logic resources in Xilinx FPGAs. The CLBs are general PLBs that is used for implementing sequential and combinational circuits on Xilinx FPGA. The XC5VLX110T has in total CLB array of 160 x 54 (Rows x Columns). In Virtex-5 series each CLB contains two slices and a switching matrix is used to switch between them as shown in figure 2.3.

The 54 CLB columns contain 108 slices, where it is an important note that, on using the DPR technique a complete CLB is taken in the constraint boundaries. In other words, you cannot split the CLB while reconfiguring the FPGA. Each slice has four 6-LUTs, four flip-flops, carry-logic and multiplexers, to provide logic, arithmetic and ROM functions. Slice heterogeneity exists in Xilinx Virtex-5 that allows more area and time optimization.

Dedicated Blocks: Like DSPs, which acts as an arithmetic logic unit, RAM blocks, PCIe core.

Input/Output Blocks: With programmable standard functionality, like LVCMOS, LVPECL, and PCI. In fact, each bank can support several standards as long as they share the same reference voltage, or output voltage.

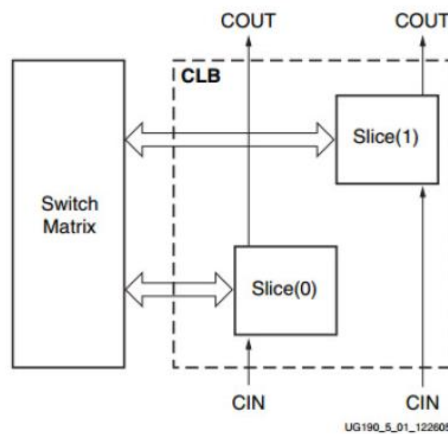


Figure 8.4: CLB routing matrix in Virtex-5

Routing: A combination of programmable and dedicated routing lines, use switching matrices connect lines from any source to any destination. Constrains can be applied.

Clocking Resources: like Phase Locked Loop (PLL) which removes clock errors, and Digital Clock Management (DCM). The dedicated clock trees balance the Skew and minimize the delay. Thirty two separate clock networks are available in Virtex 5 FPGA.

8.1.3 MicroBlaze softcore processor

MicroBlaze is an embedded softcore microprocessor. It is a reduced instruction set computer (RISC) based architecture. MicroBlaze is optimized for implementation in

Xilinx FPGAs families using a portion of the available resources on the FPGA. Figure 8.5 shows the internal construction for the MicroBlaze .

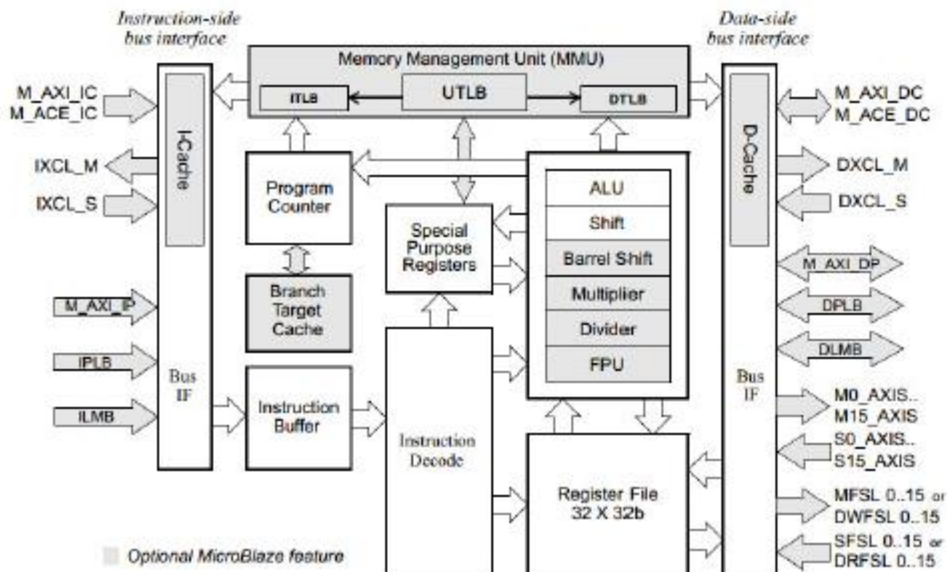


Figure 8.5: MicroBlaze block diagram

8.2 Hardware System Hierarchy

As explained before PDR has two for implementation: JTAG cable with PC control or Embedded system on Chip for control, for our project we chose the embedded system on chip for a better control environment so the FPGA chip could be completely independent and could control its switching process without external interference.

The system hierarchy is shown in Figure 8.6, The system consists of Static part that contains the microprocessor which control the PDR process and enables the ICAP module to reconfigure the specified partition with the proper bit file, we use an external compact flash memory to hold the bit files with a sys_ace interface in our system to read from it, For testing purposes only we use the board UART port, so we add to this system an additional UART module to handle communications between PC and the micro-blaze processor, For testing also we use the compact flash to hold input and output files for our system.

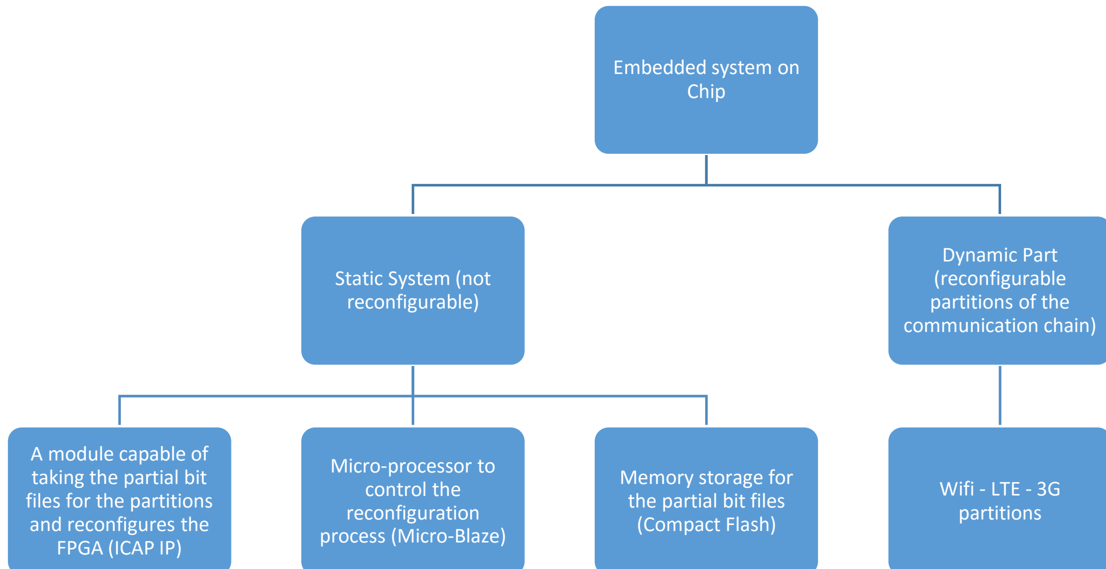


Figure 8.6: Embedded system on chip

A more detailed block diagram is shown in Figure 8.7, the micro-blaze processor uses PLB bus to communicate with its peripherals like the UART, System ACE, ICAP, and communication partitions. So the communication system partitions should be in the Green RP module, as the communication modules are not built to communicate with a Micro-processor or to be a peripheral of it so a container is needed to handle the communications with the micro-processor.

This container must have some modules to deal with the PLB bus and other controllers to communicate with the partitions, the block diagram for this container is shown in Figure 8.8, in the system used, reconfigurable partitions are only slaves so only PLB slaves are used, the communication process between the micro-blaze and the peripheral is shown in Figure 8.9, the peripheral contains a memory of slave registers, each one has a unique address, the micro-blaze processor transfers certain data with certain slave register address through the PLB bus to the PLB slave, the PLB slaves moves data and address to the user_logic module which contains the slave registers and the communication partition, the user_logic should store the data in its slave register specified by the address, later the communication peripheral should read data from the salve registers, The same happens for reading, the communication peripheral should write the data in the slave registers and then the micro-blaze processor requires to read certain address from the salve register to get data.

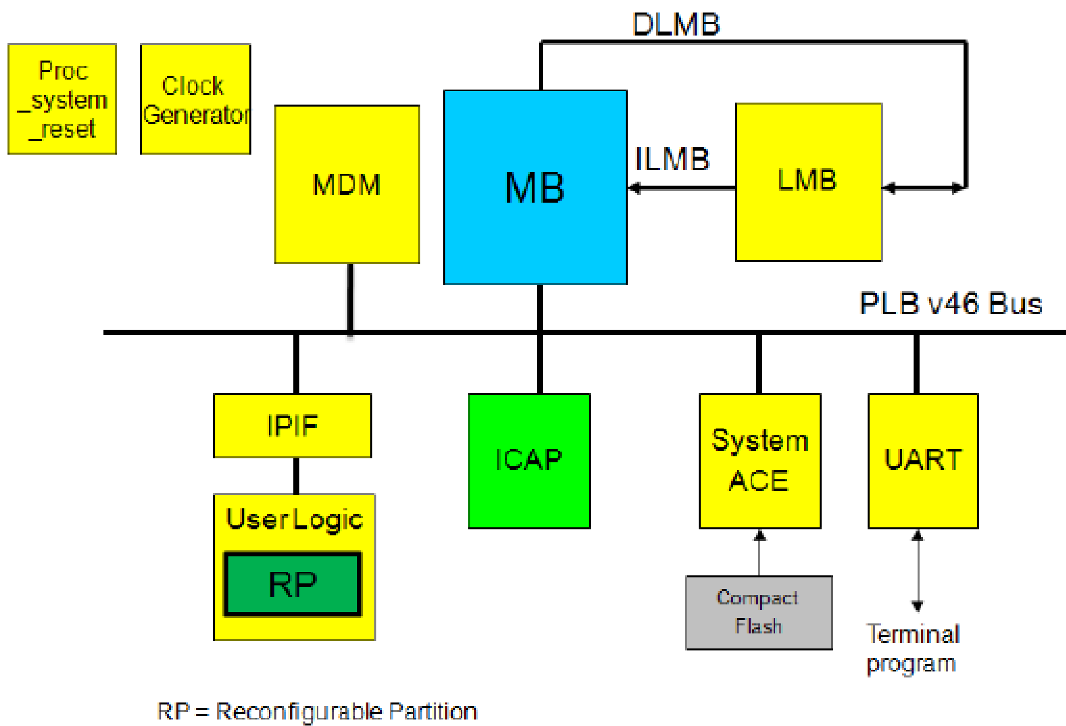


Figure 8.7: Embedded system on chip detailed block diagram

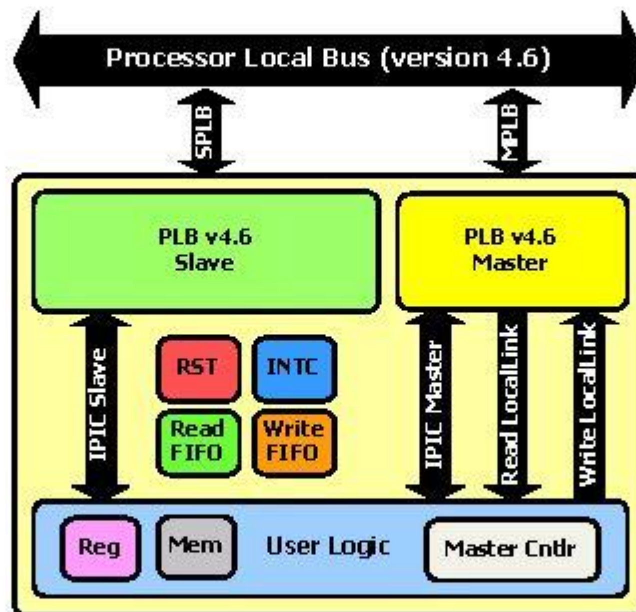


Figure 8.8: Peripheral template for partitions

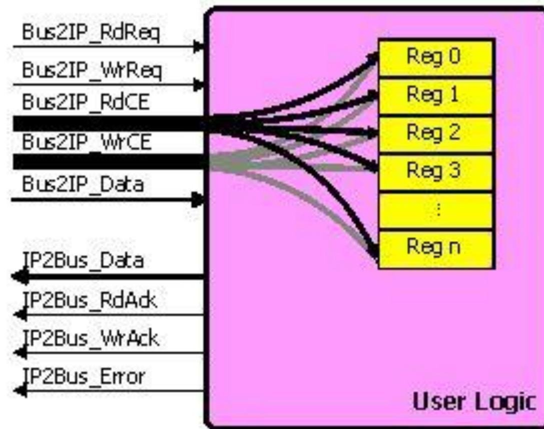


Figure 8.9: Peripheral slave registers

8.3 Partial dynamic reconfiguration flow

To develop a partial reconfiguration design using Xilinx tools we are going to use the Xilinx Platform Studio (XPS), Software Development Kit (SDK), and the Plan-Ahead™ design tool. XPS is used to create a processor hardware system that includes a lower-level module defining the Reconfigurable Partition (RP). SDK is used to create a software application that enables you to perform partial reconfiguration. XPS and SDK are part of the Embedded Design Kit (EDK), which is included in the ISE® Design Suite Embedded and System Editions.

Plan-Ahead is used to Floor-plan the design including defining a reconfigurable partition for the reconfigurable Region and Create multiple configurations and run the partial reconfiguration implementation flow to generate full and partial bit-streams.

The example that will be used to explain the flow of partial dynamic reconfiguration we are going to define one Reconfigurable Partition (RP) and two Reconfigurable Modules (RM). The two RM perform crc_4g and crc_3g functions.

The steps will be as shown in Figure 8.10 .

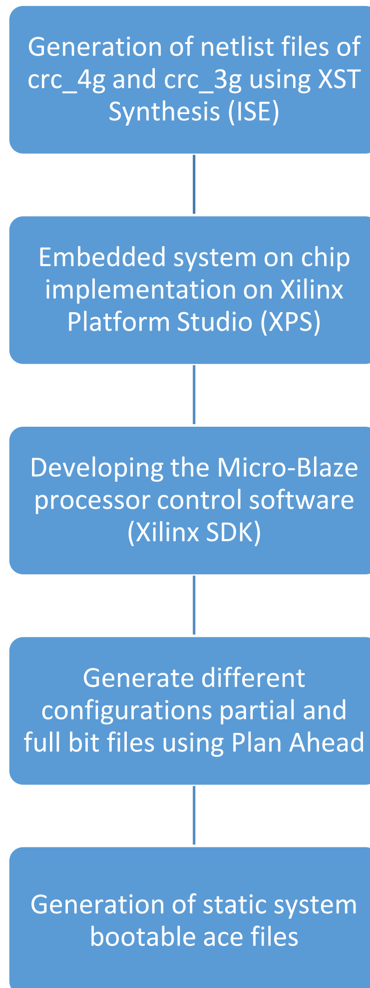


Figure 8.10: Partial dynamic reconfiguration flow

8.3.1 Generation of netlist files (XST Synthesis)

1. Create new project.

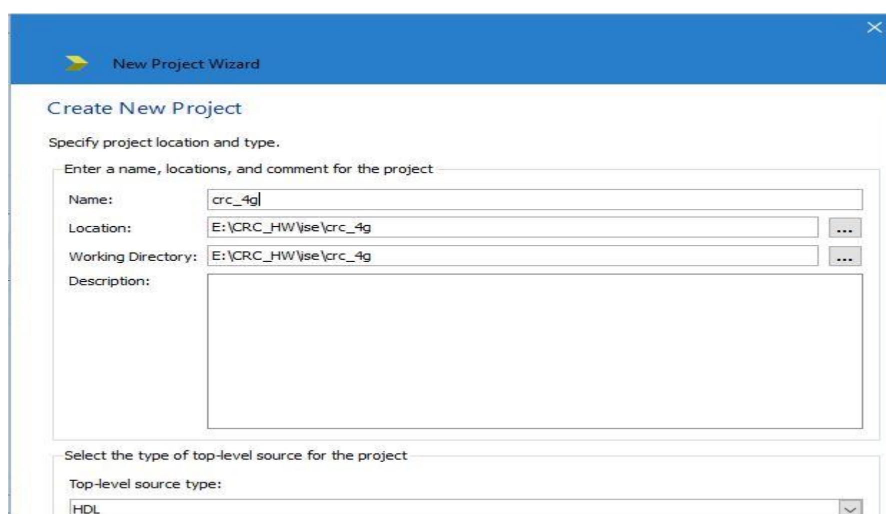


Figure 8.11: ISE create project

2. Adjust the design properties to meet the kit specifications.

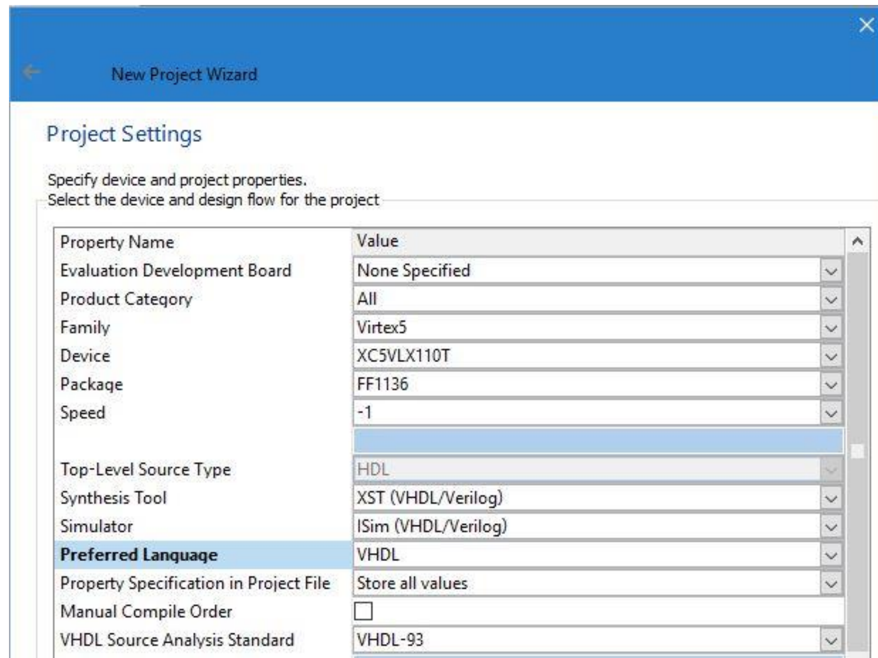


Figure 8.12: ISE kit specs

3. Add the source codes to the project.

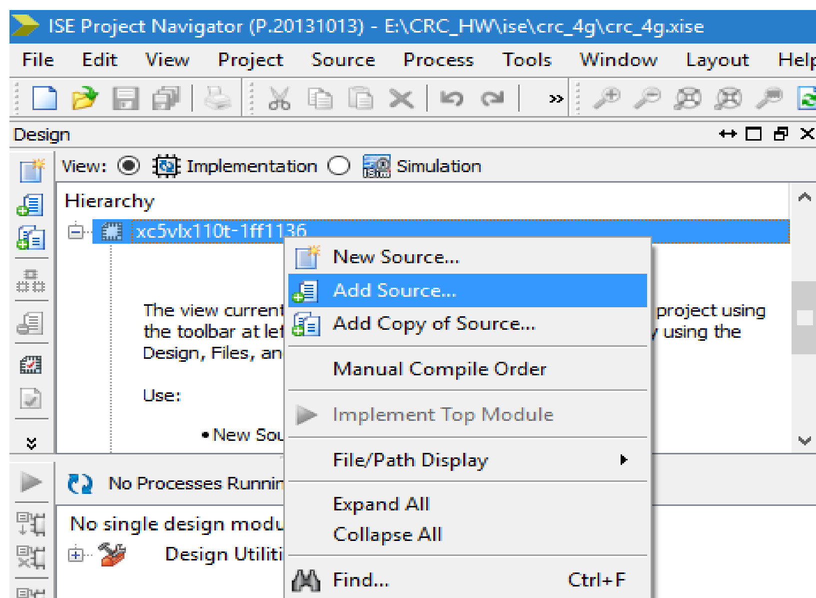


Figure 8.13: ISE add source

4. From synthesis design properties uncheck the I/O buff as shown in Figure 8.15.

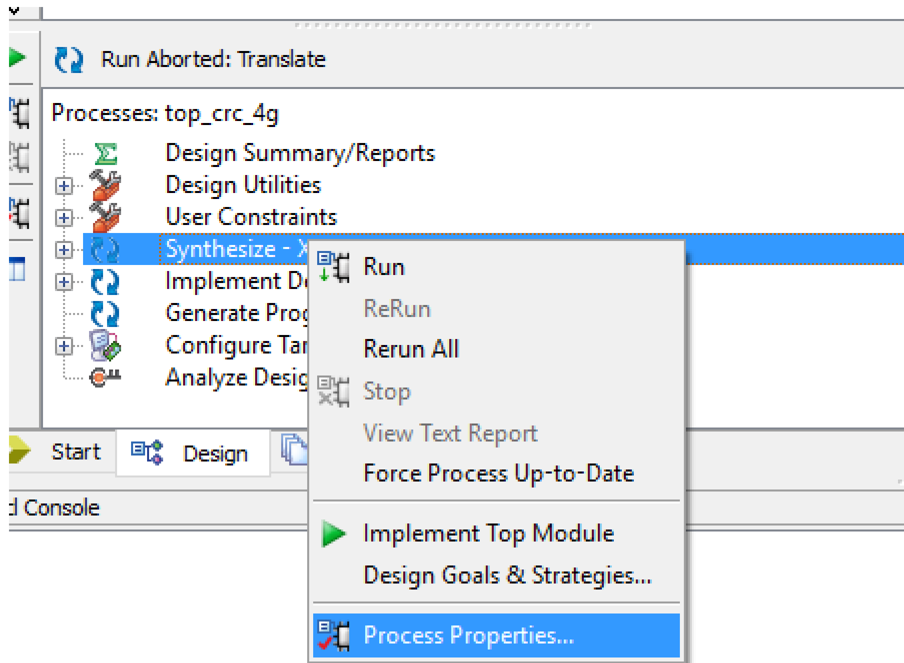


Figure 8.14: ISE synthesis

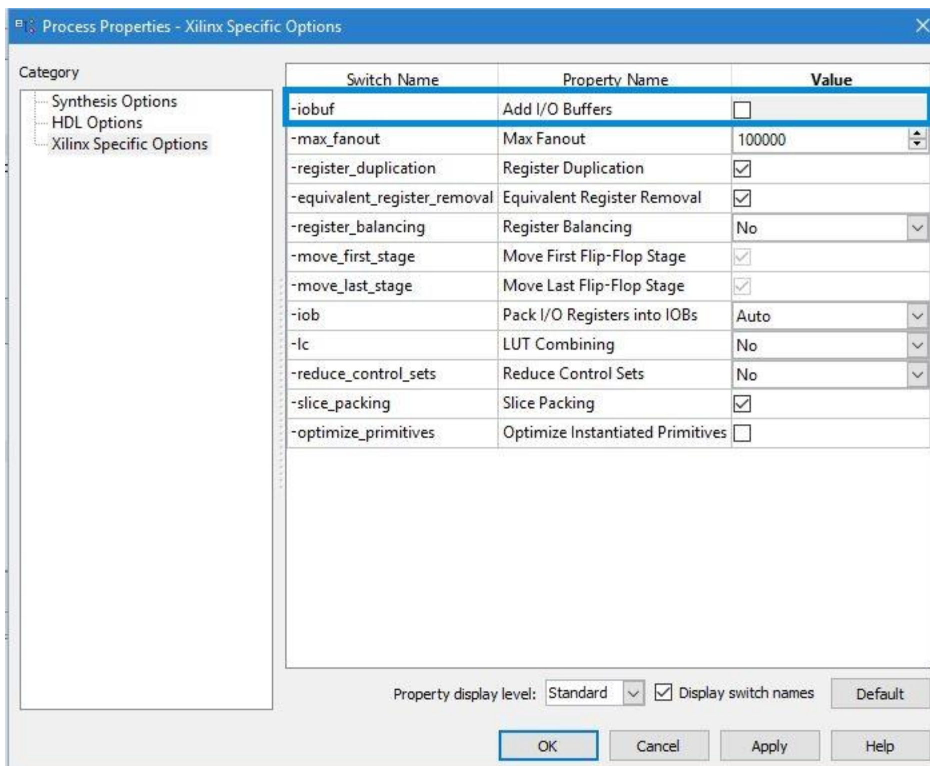


Figure 8.15: Synthesis properties

5. XST synthesis.

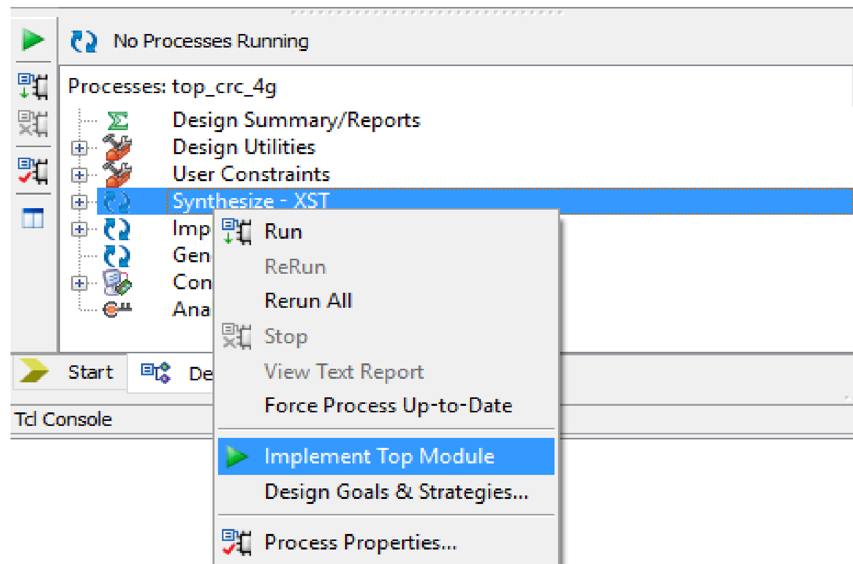


Figure 8.16: Run Synthesis

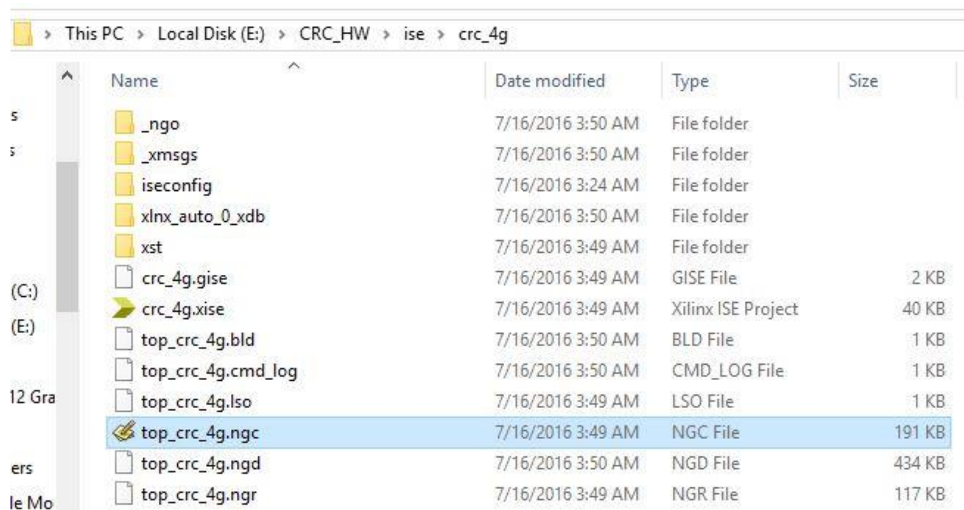


Figure 8.17: Resulting Netlist file

Get the netlist file (.ngc) from the project directory then repeat the same steps for the crc_3g function.

8.3.2 Embedded Development Kit (EDK)

- Embedded Development Kit (EDK) is a Xilinx software suite for designing complete embedded programmable systems.
- It enables the integration of both hardware and software components of an embedded system.

- It includes all the tools, documentation, and IP that you require for designing systems with embedded IBM PowerPC™ hard processor cores, and/or Xilinx MicroBlaze™ soft processor cores.
- XPS and SDK are part of the Embedded Design Kit (EDK), which is included in the ISE® Design Suite Embedded and System Editions.

8.3.3 Xilinx Platform Studio

- Includes IP cores that required for designing a complete embedded systems.

- Generates netlists and simulation models of the hardware of the embedded programmable systems.

1. Open Xilinx platform studio and create new project using base system builder.

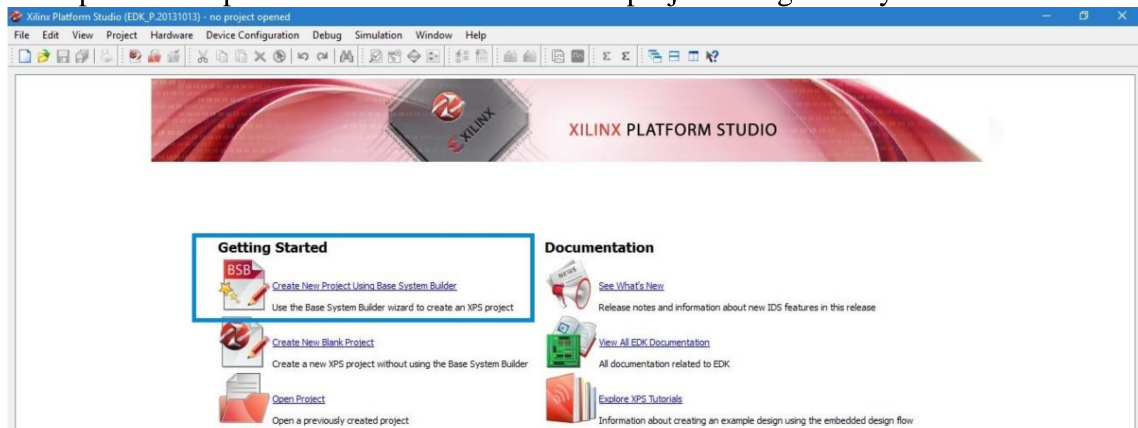


Figure 8.18: XPS main window

2. Define the project path, select the interconnect type and add the board support package path because by default the XPS doesn't contain the package of our board.

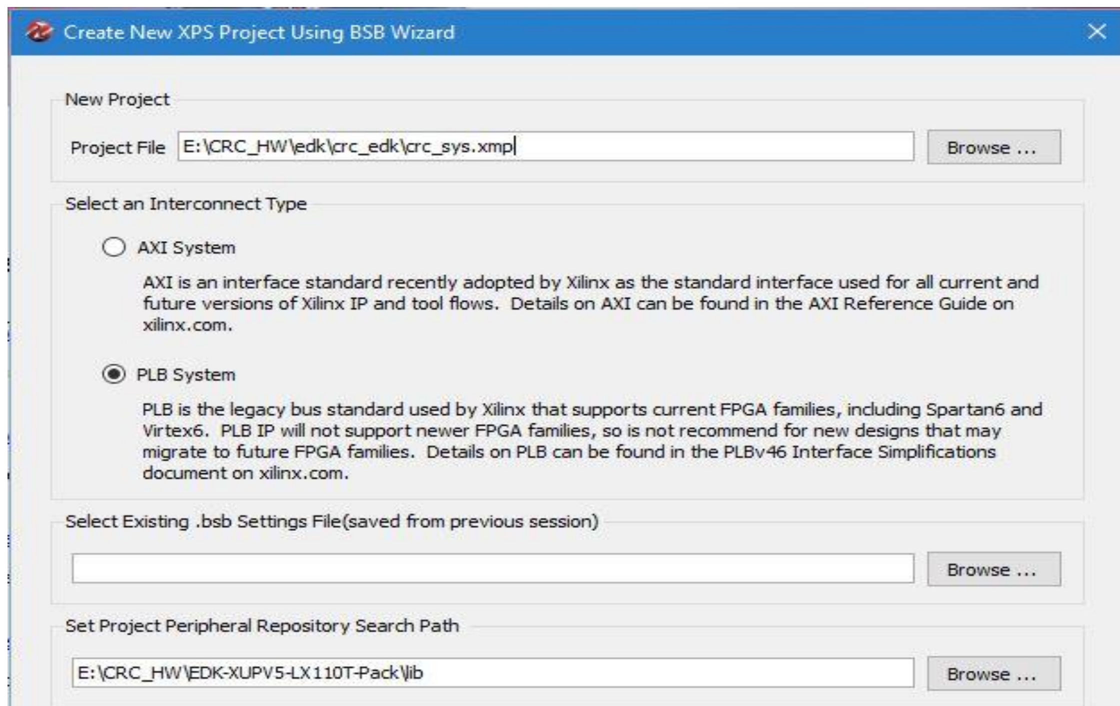


Figure 8.19: XPS new project wizard

3. Create new design.

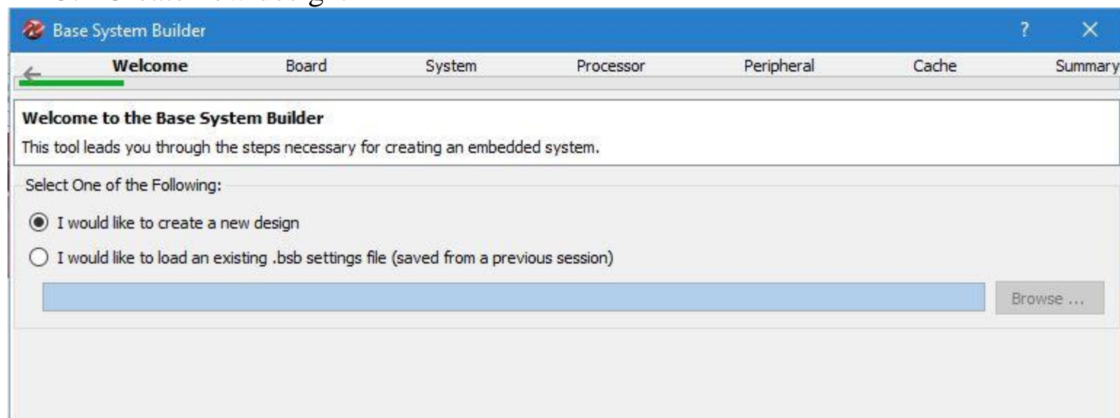


Figure 8.20: Base system builder wizard

4. Select the target development board that defined in step 2.

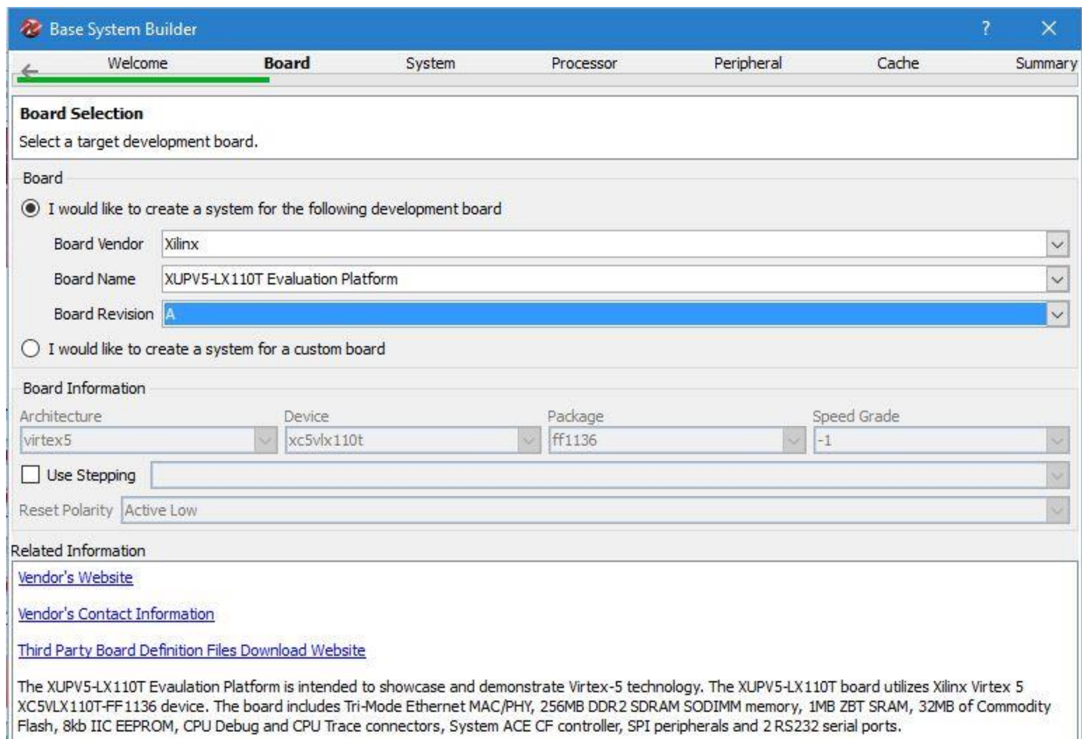


Figure 8.21: BSB board choice

5. Configure the system as a single processor system in the system configuration.

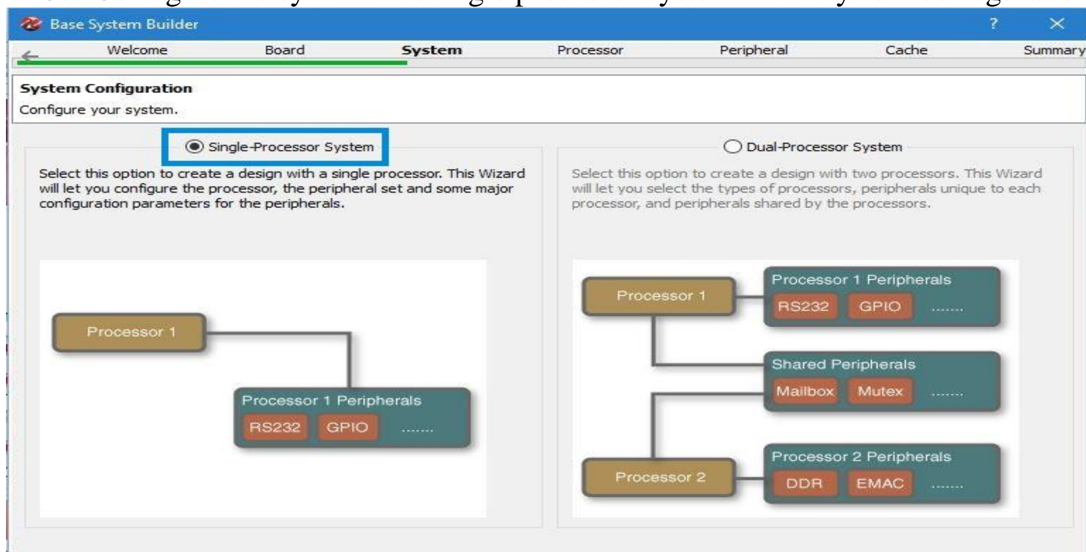


Figure 8.22: BSB single processor system

6. Configure the processor by choosing processor type as MicroBlaze, speed (design dependent) for example 100 and memory size (design dependent) for example 64k.

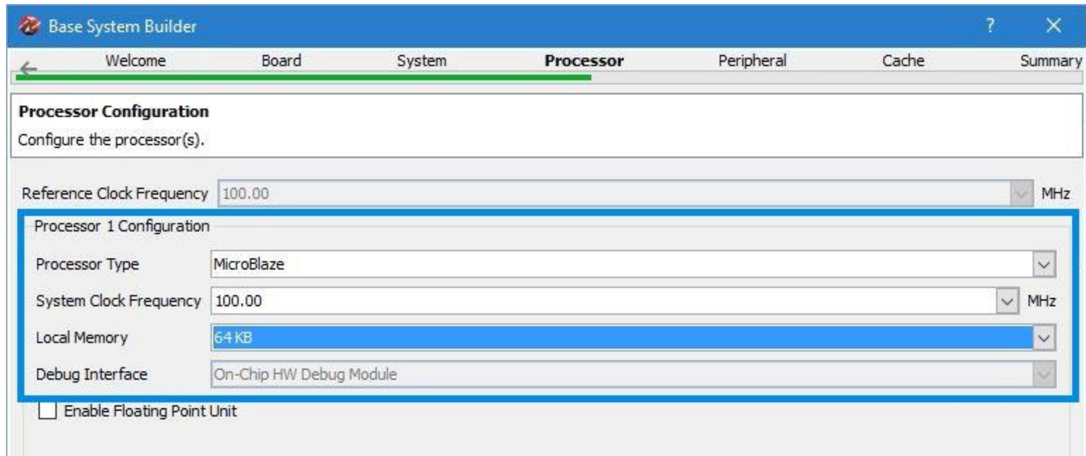


Figure 8.23: BSB Micro-Blaze specs

7. Adjust the peripheral configuration list by removing all the unrequired peripherals then adjust the UART baud rate to 115200.

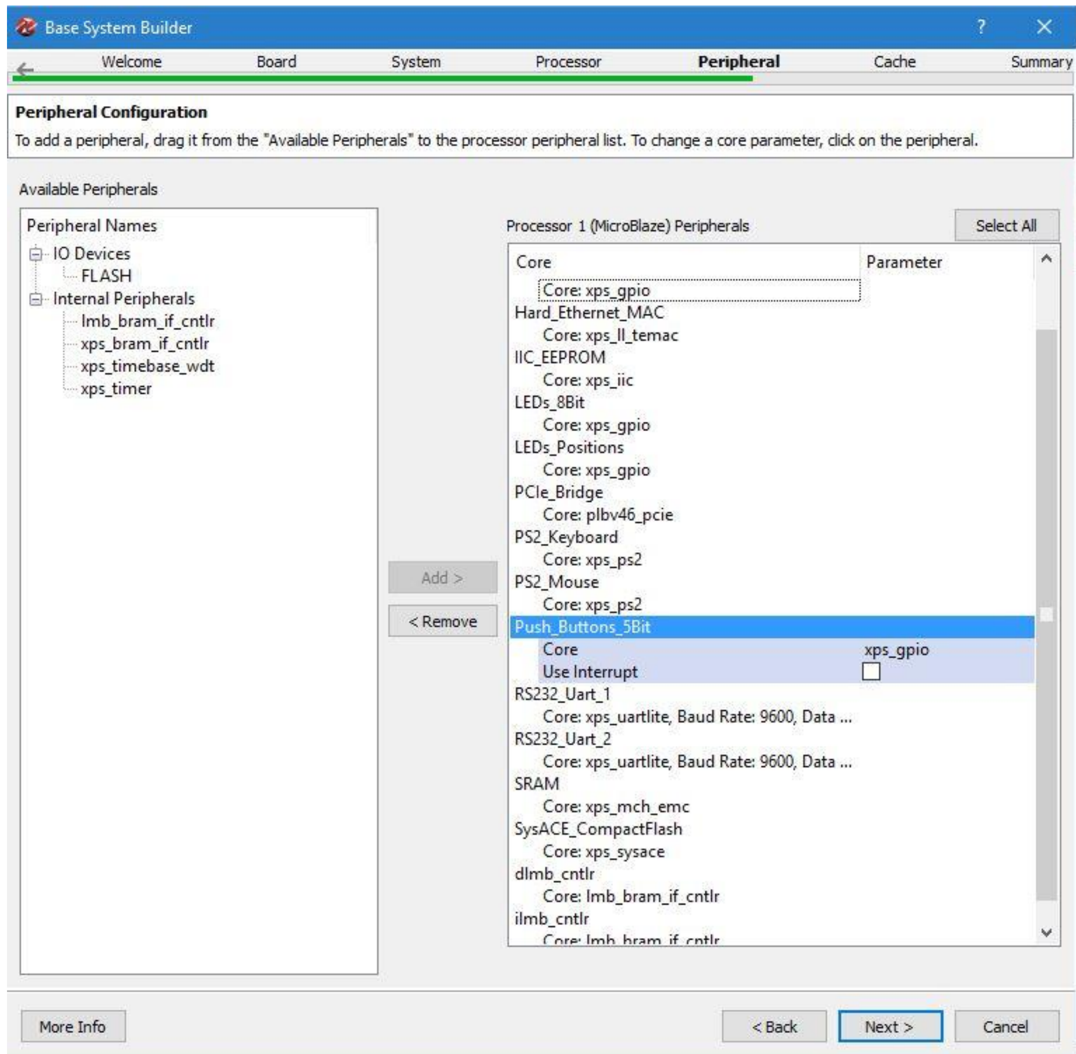


Figure 8.24: BSB peripheral wizard

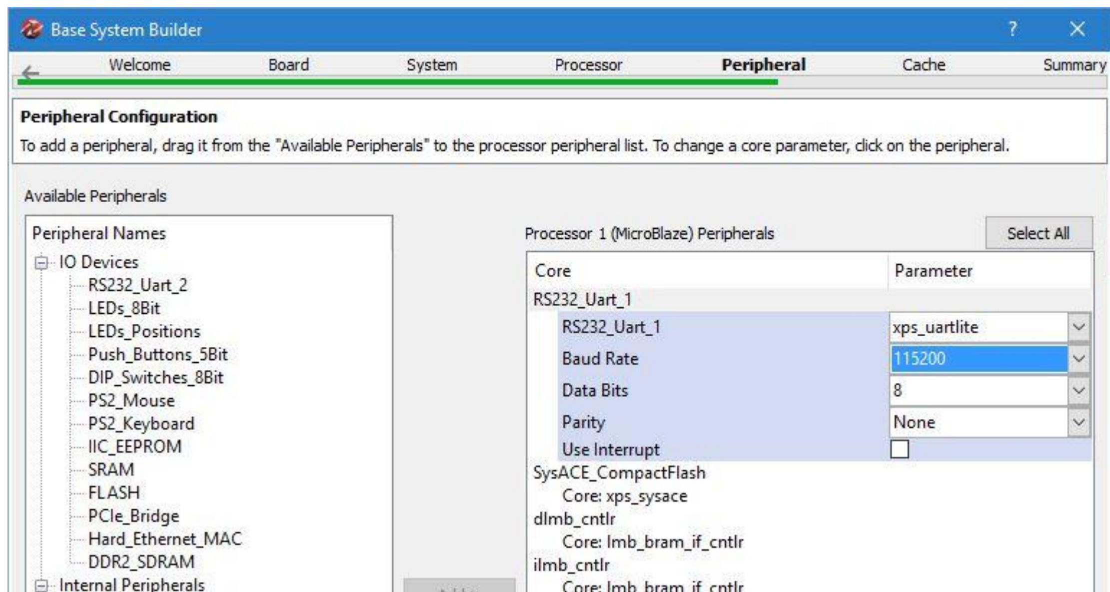


Figure 8.25: BSB UART specs

8. Advance to finish the base system builder wizard.

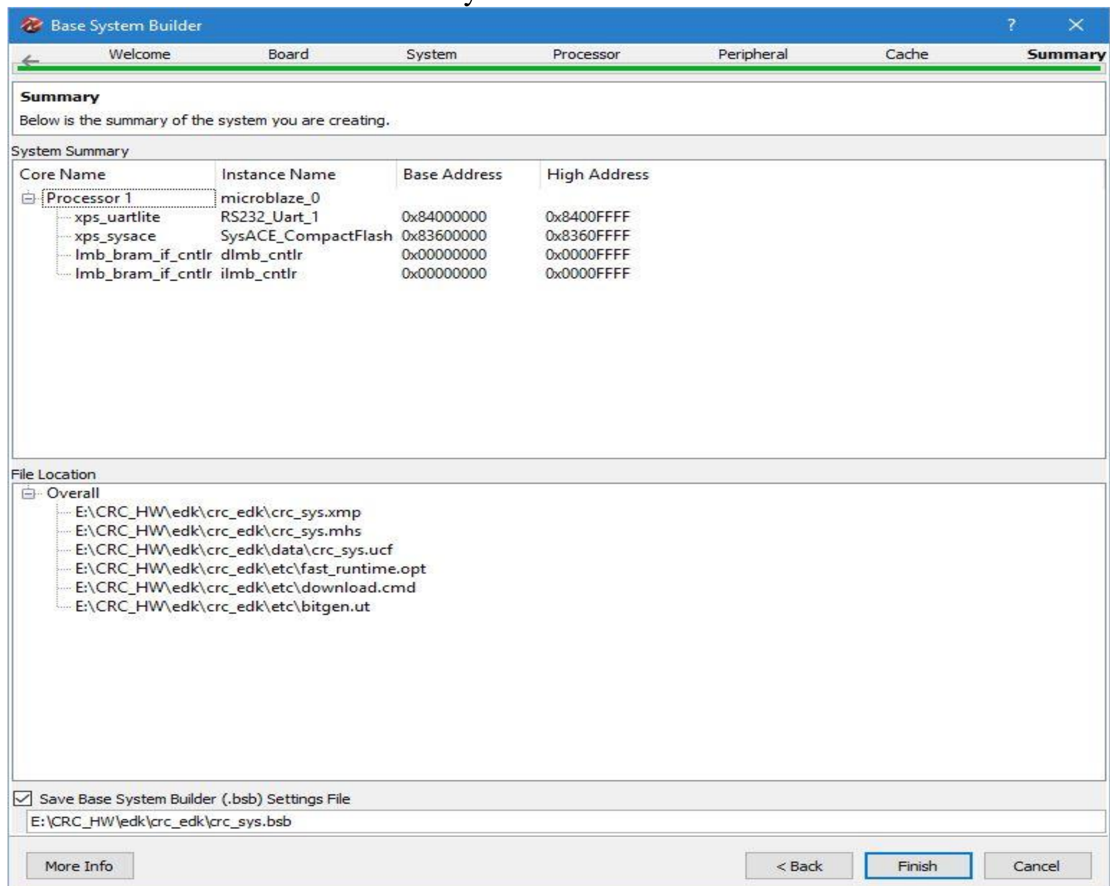


Figure 8.26: finish BSB wizard

9. Now we want to create a new peripheral that include the reconfigurable partition and has the ability to deal with PLB bus (IP core) as explained before. The steps will be as follows:

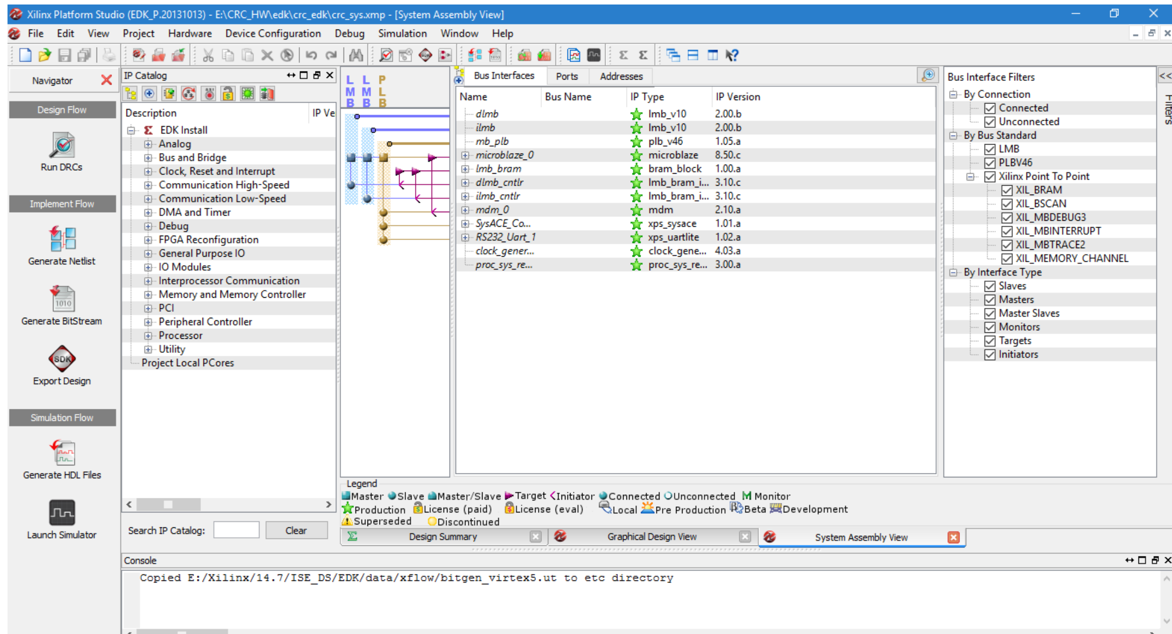


Figure 8.27: Working Space for XPS

1. From hardware select create or import peripheral.

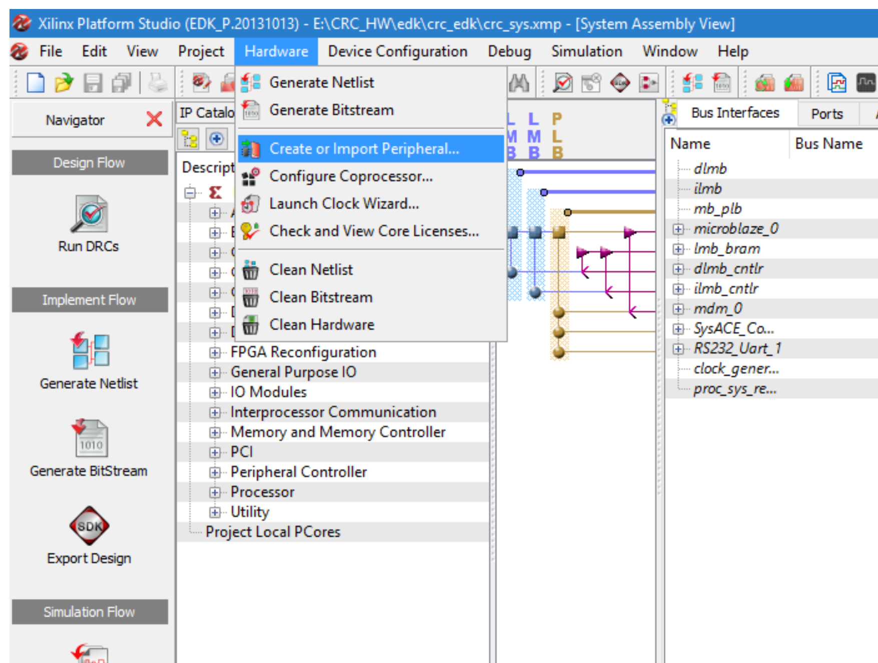


Figure 8.28: XPS hardware menu

2. Select create template for a new peripheral from the wizard.

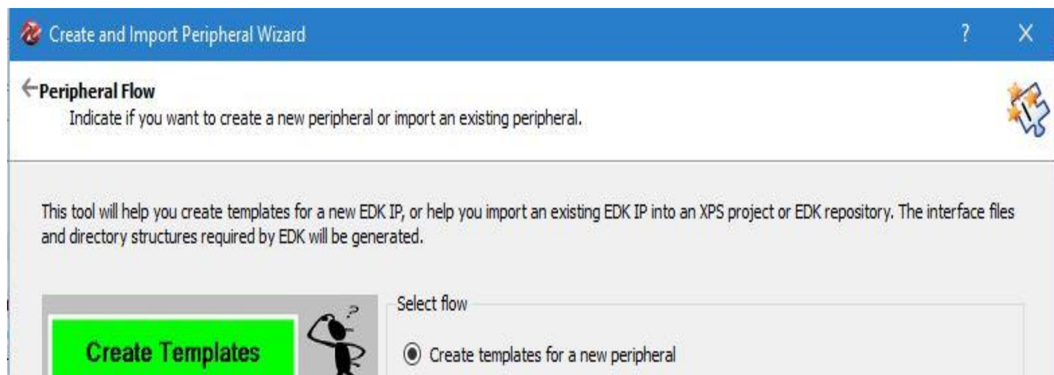


Figure 8.29: Create or Import peripheral wizard

3. Store the new peripheral to an XPS project, this will store the peripheral in the pcores folder. This folder should contain all user custom peripherals.

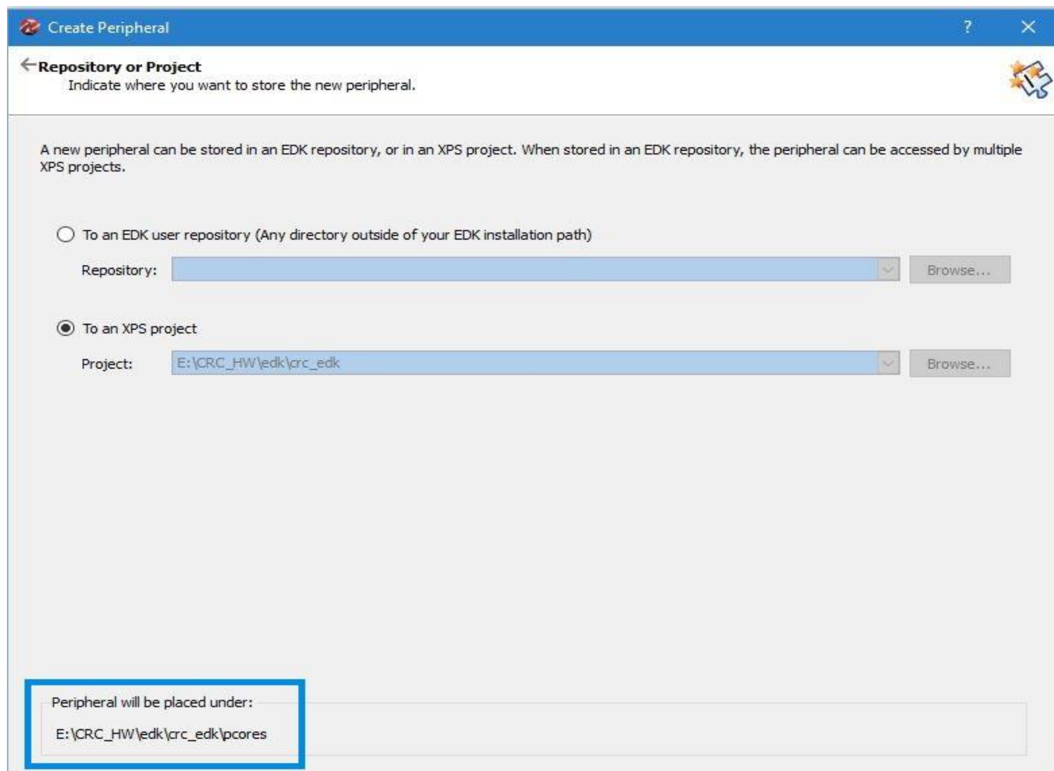


Figure 8.30: Create peripheral wizard save location

4. Enter the name of the peripheral.

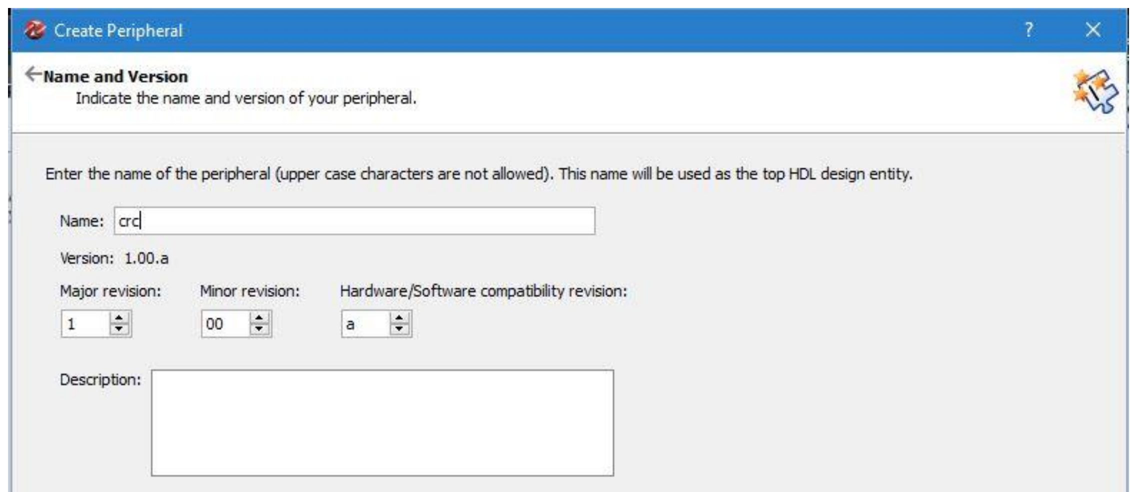


Figure 8.31: Peripheral name

5. Choose which buses will the peripheral attached to, in our case we are going to select BLP bus (Xilinx Virtex 5).

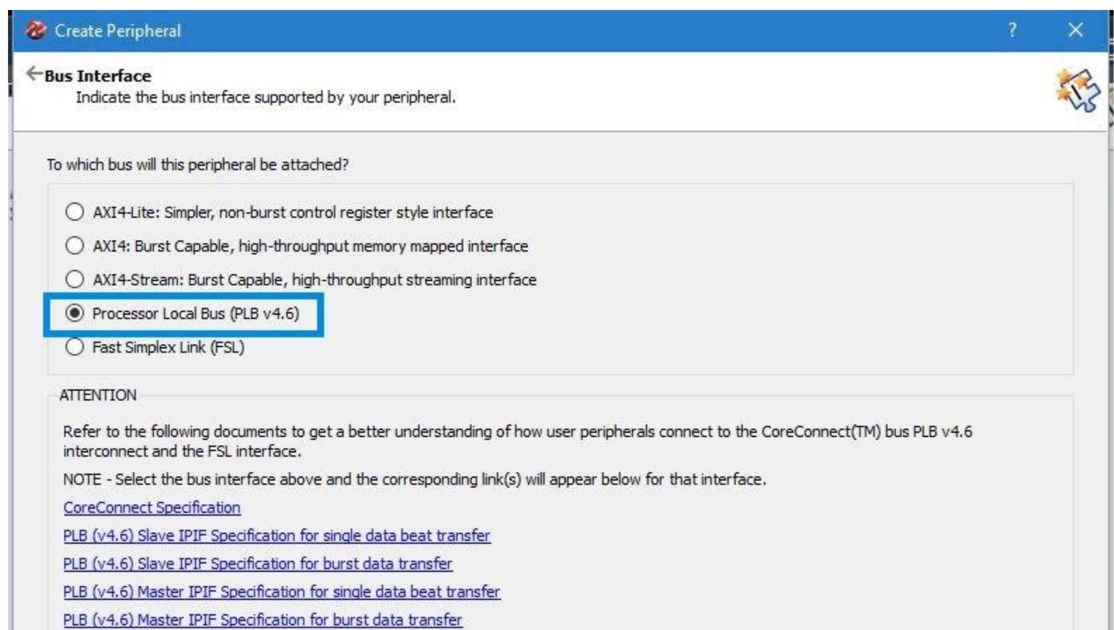


Figure 8.32: Bus choice

6. The peripheral will be connected to the PLB interconnect through corresponding PLB interface (IPLF) modules, which provide us with a quick way to implement the interface between the PLB interconnect and user logic. Besides the standard functions like address decoding provided by the slave IPIF module, the wizard tool also offers other commonly used services and configurations to simplify the implementation of the design.

According to the wizard we check the required services and configurations.

Figure 8.33: IP interface

7. Determine the number of software accessible registers.

Figure 8.34: Slave reg choice

8. Advance to finish create peripheral wizard.

10. Now we created the required peripheral template of our RP and it stored in the pcores folder. From project make rescan user repositories to import the created peripheral in the IP catalog.

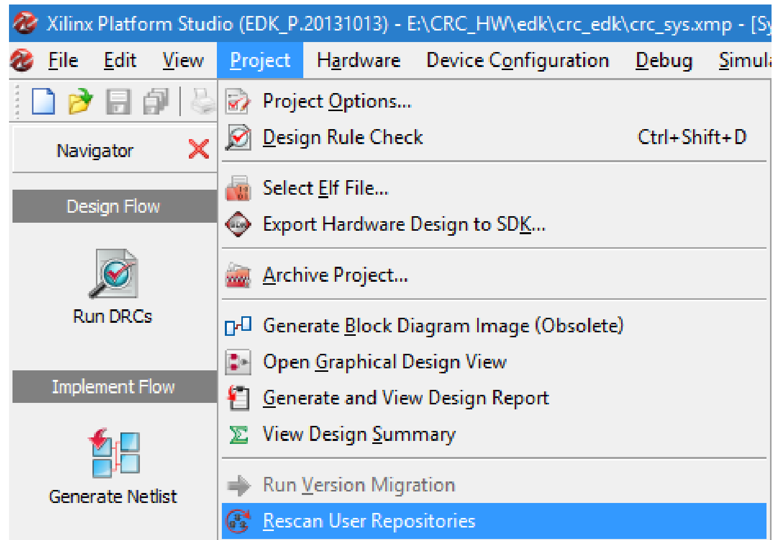


Figure 8.35: XPS rescan

11. We also need to import another peripheral that responsible for the FPGA reconfiguration but this peripheral already exists in the IP catalog.

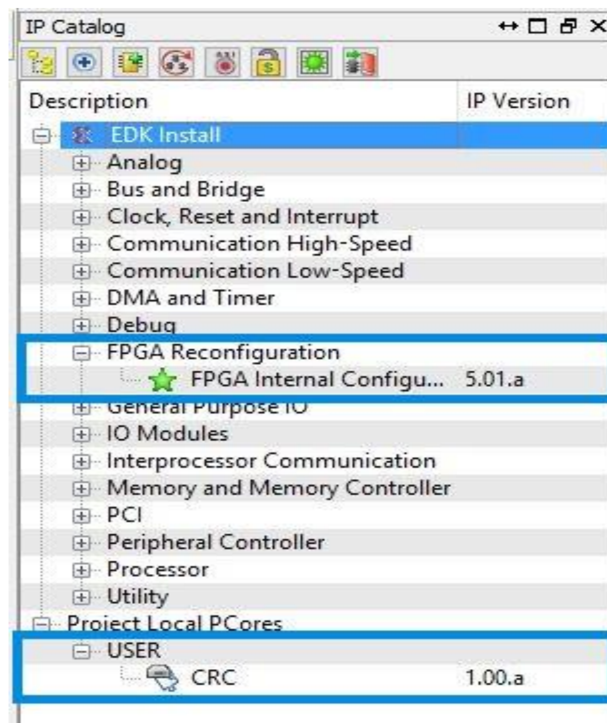


Figure 8.36: IP catalog

12. After adding the two peripheral to the bus interface they need to be connected to the PLB bus.

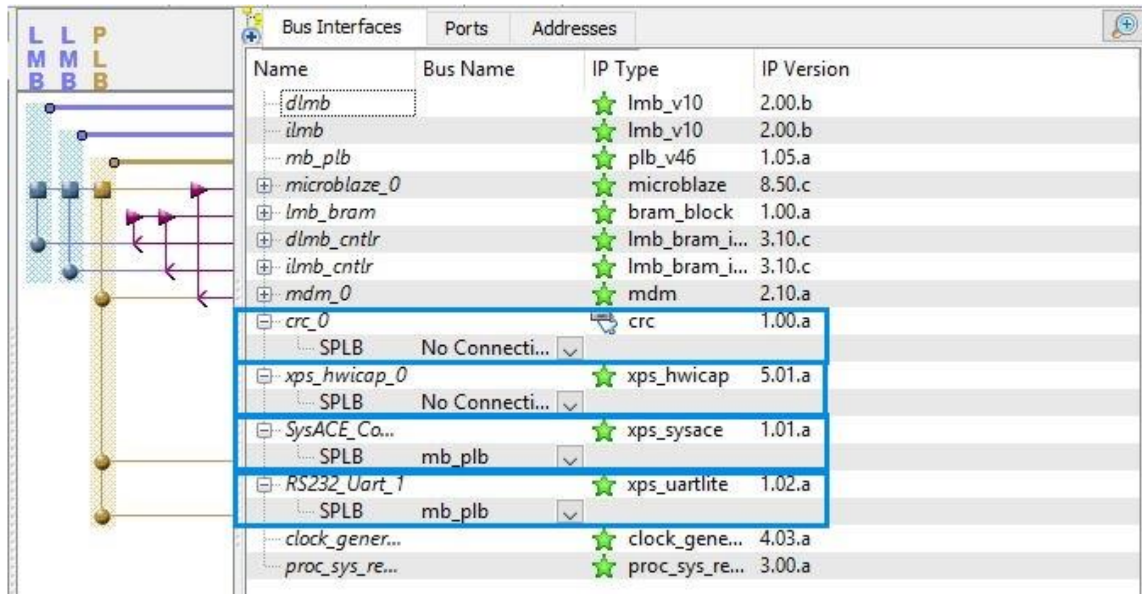


Figure 8.37: Peripheral Bus Connection

13. Generate addresses for the unmapped peripherals.

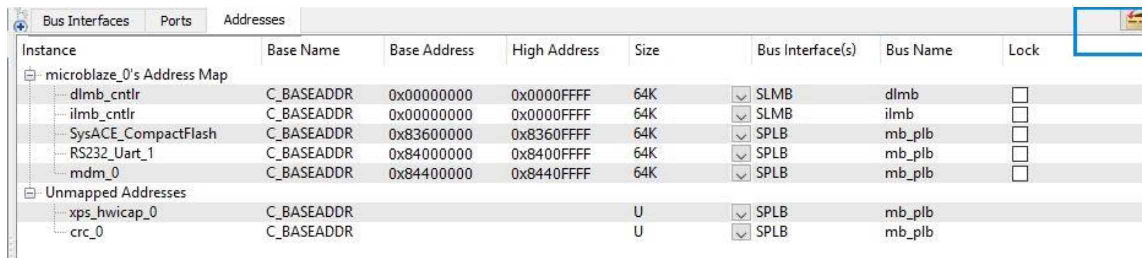


Figure 8.38: Peripheral Address initiation

14. If one of these peripherals has unconnected port we have to connect it from ports to prevent floating connections.in our case only the peripheral that responsible on the FPGA reconfiguration has unconnected port which is ICAP _Clk, we connect it to the output of the clock generator as shown.

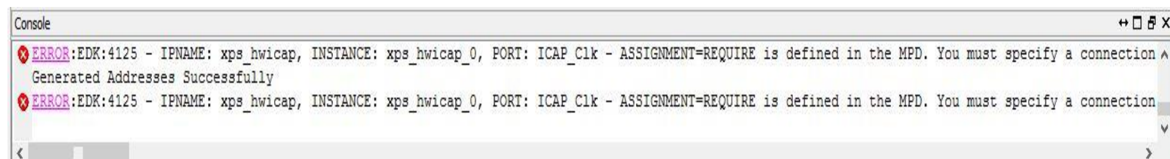


Figure 8.39: Console display for missing port connection error

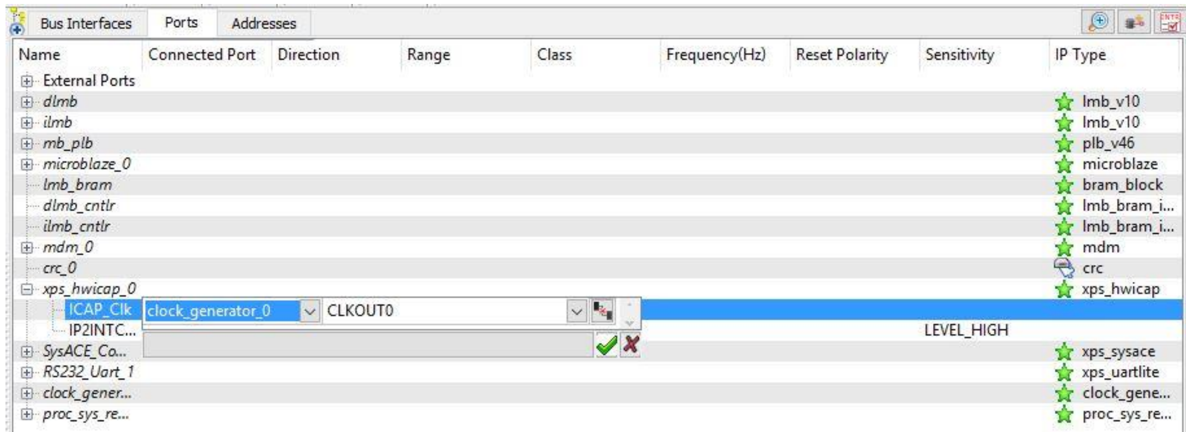


Figure 8.40: Connecting the ICAP_Clk port to the kit clock generator

15. Generate netlist.

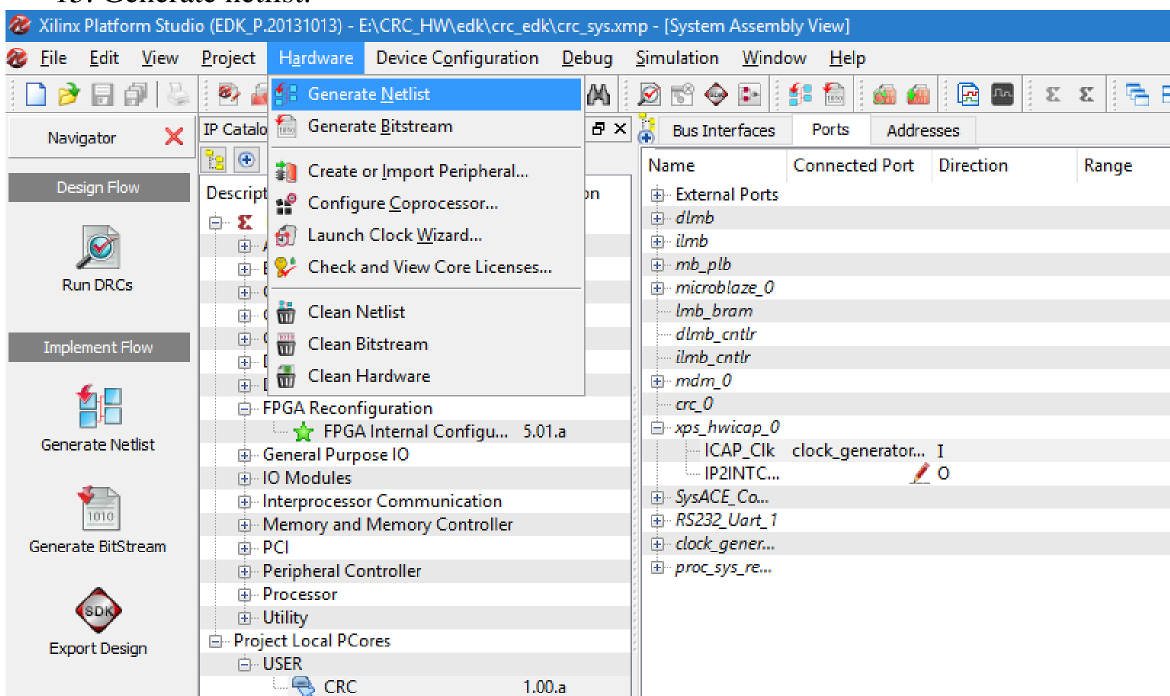


Figure 8.41: Generating Netlist File

8.3.4 Software Development Kit

- Used to perform the Software development.
- Xilinx Custom Compiler settings for PowerPC, MicroBlaze.
- Code editor, Error Navigation and debug.
- A hardware image is first generated to define the hardware platform for which the software application will be developed.

Resuming XPS flow:

1. To export to SDK from XPS, in XPS select projects > Export Hardware Design to SDK.

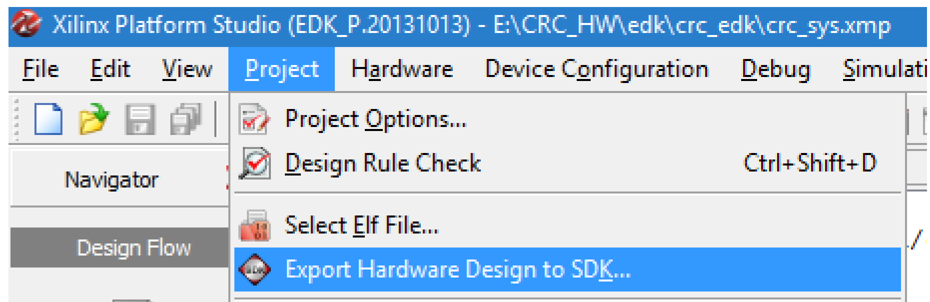


Figure 8.42: Exporting the XPS file to the SDK for microprocessor configuration

2. Uncheck include bit stream and BMM file because we haven't generated these files yet.

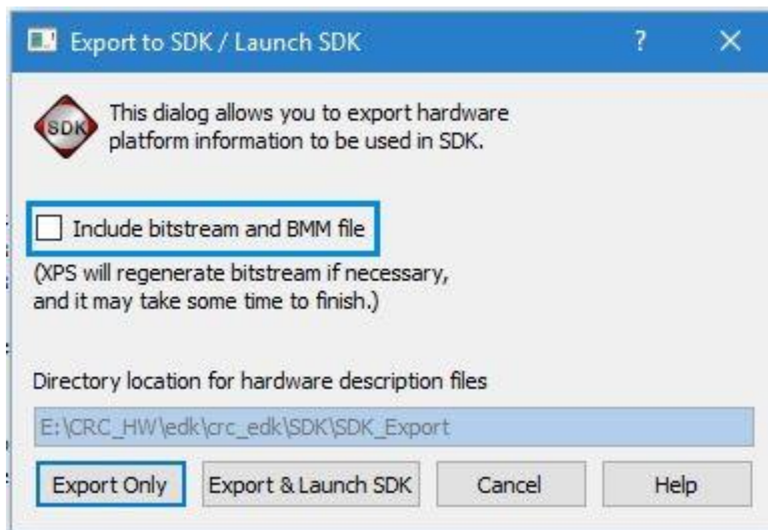


Figure 8.43: Bitstream and BMM generation

3. Browse to the <Extract_Dir>/edk/SDK/SDK_Export workspace directory then click ok to open the SDK.

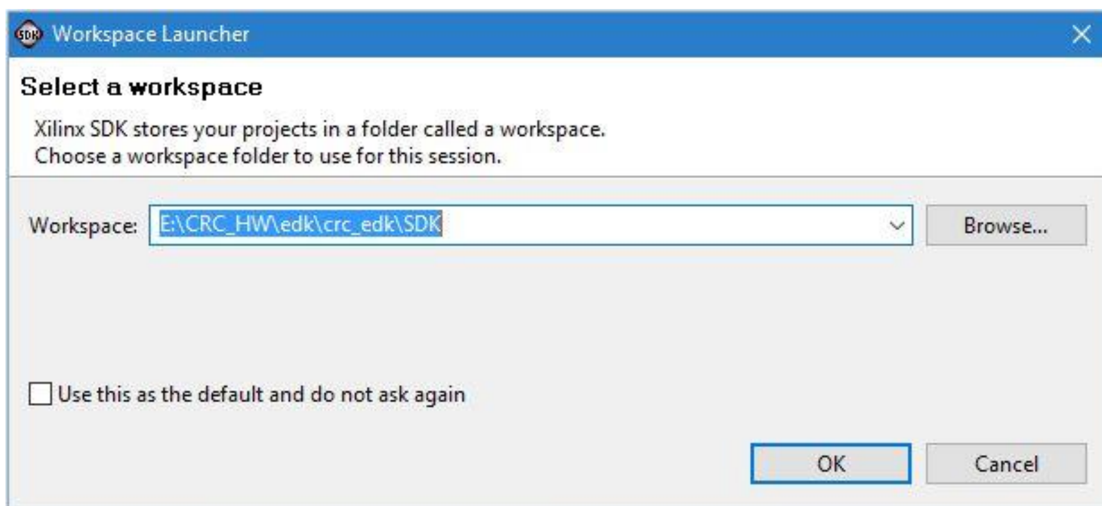


Figure 8.44: SDK Launcher

4. Create new board support package.

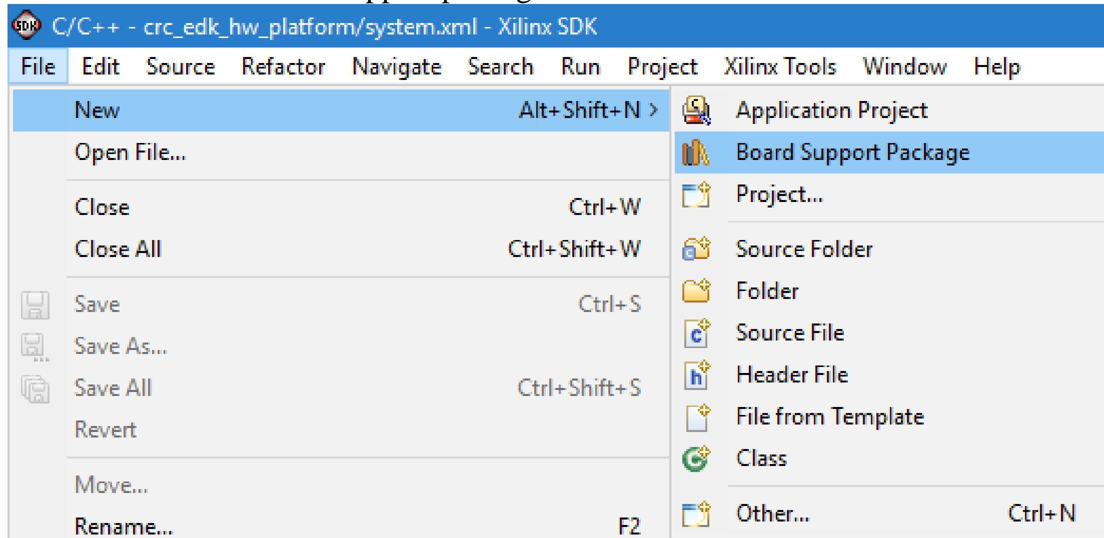


Figure 8.45: creating board support package

5. The default Project Name is standalone_bsp_0 and the OS is standalone. Then click finish to open the board support package settings window.

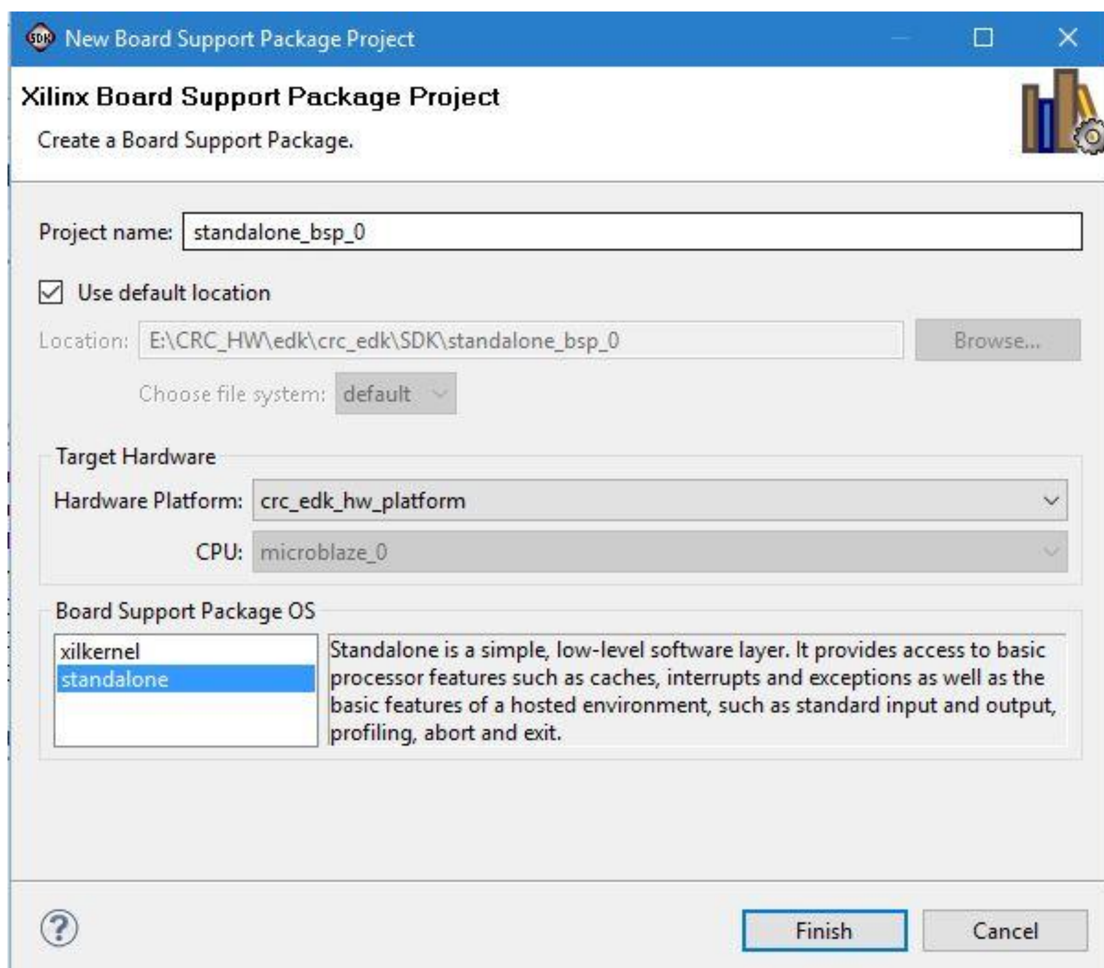


Figure 8.46: Defining Project name

6. Check the xilfatfs check box to select the FAT file system support for the Compact Flash card.

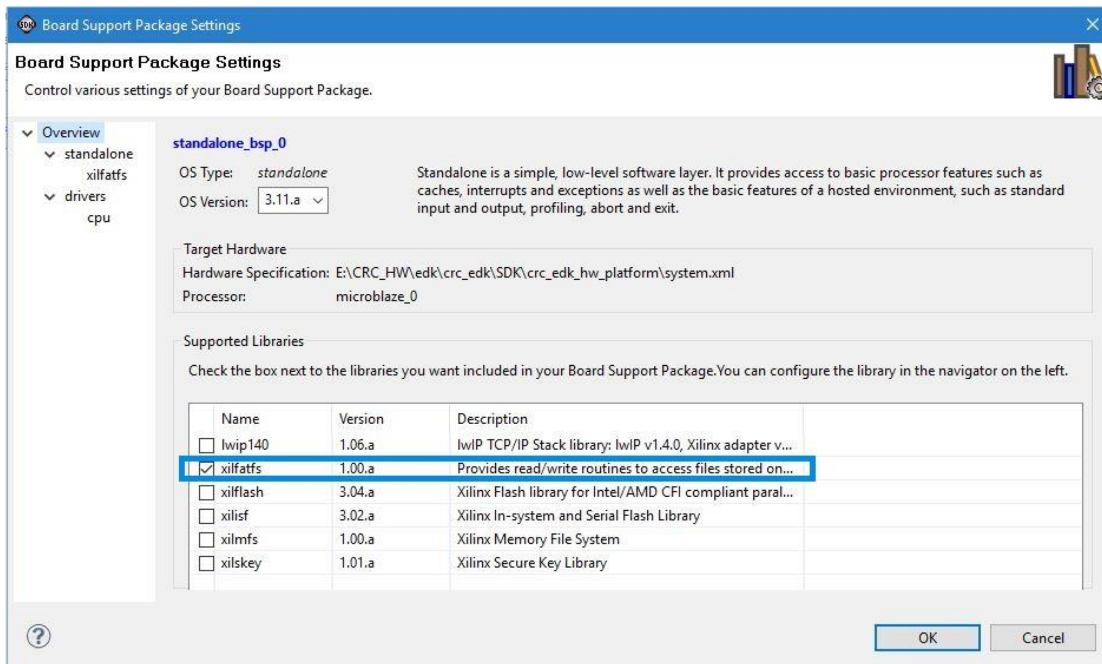


Figure 8.47: Board Support Package Supported Libraries

7. From xilfatfs, make sure that the value of CONFIG_WRITE is true to be able to write in the system ace.

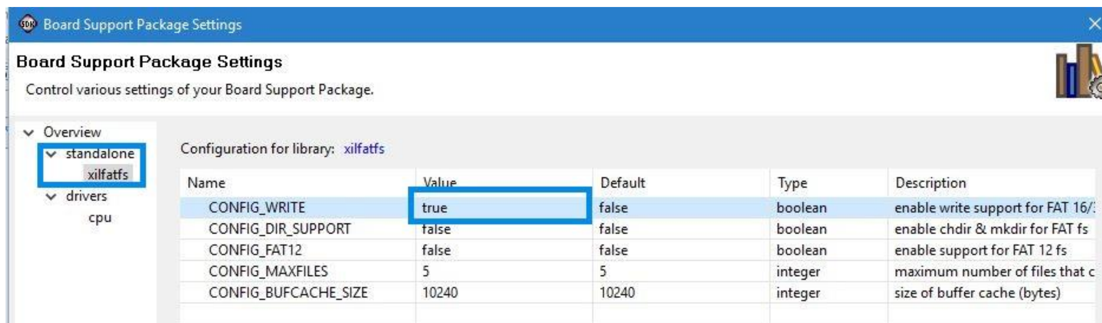


Figure 8.48: Configuration for library xilfatfs

8. Create new application project.

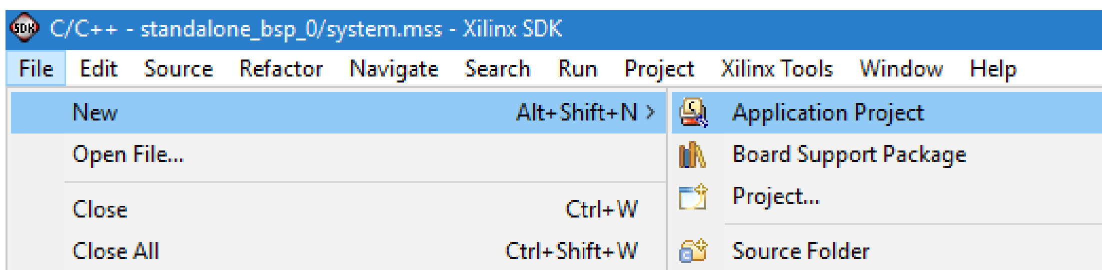


Figure 8.49: Creating Application Project

9. Type the project name and select use existing option under the Board Support Package field

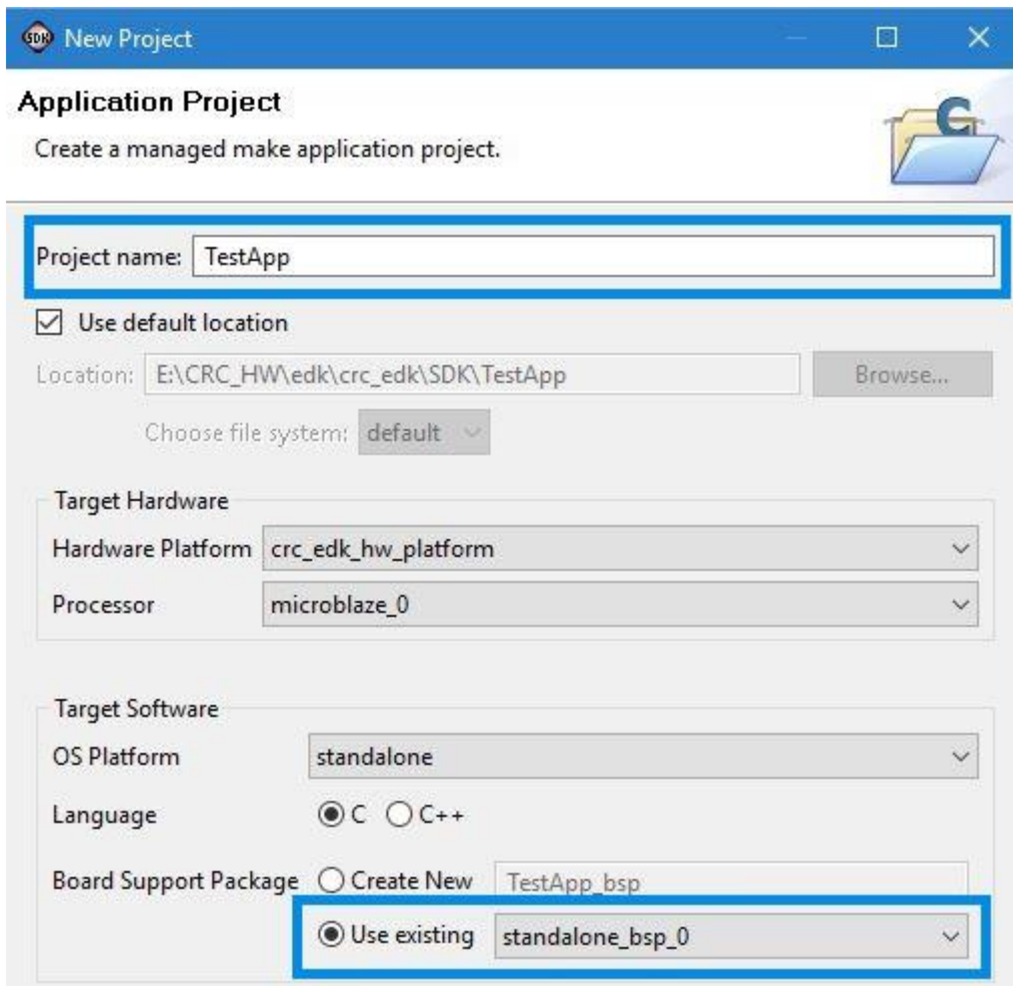


Figure 8.50: Defining Project name

10. Select empty application in the project application template pane.
11. Right click on TestApp from project explorer and select import.
12. From import choose general -> file system.
13. Browse to the <Extract_Dir>/resources/TestApp/src/ folder and Select main.c and xhwicap_parse.h then click finish.

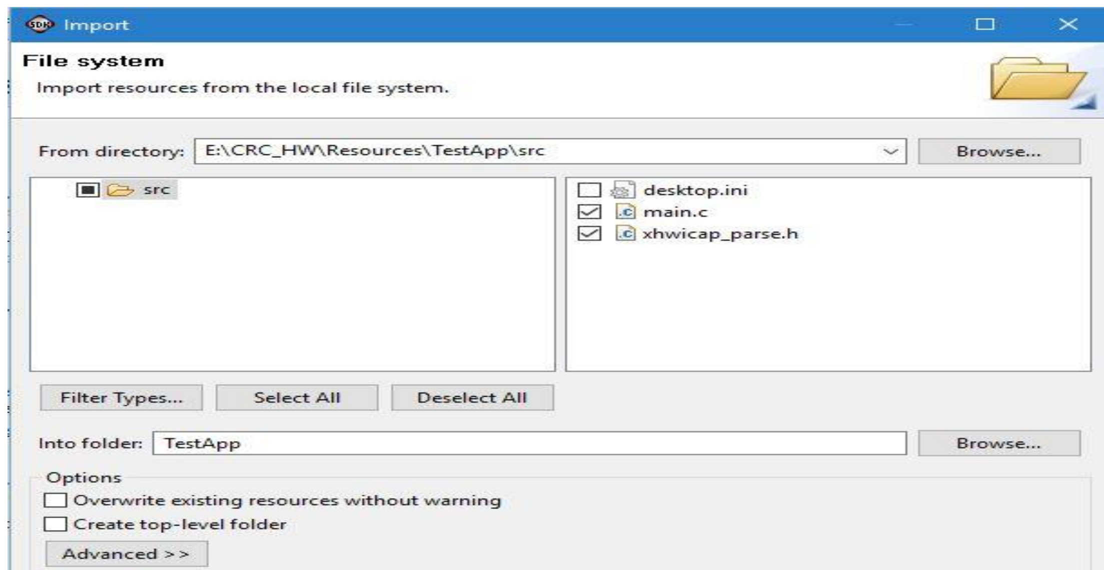


Figure 8.51: Import Recources

14. Right click on TestApp and select generate linker script.
15. Be sure that the Heap and Stack sizes are set to 2048 (0x800).
16. Click generate.

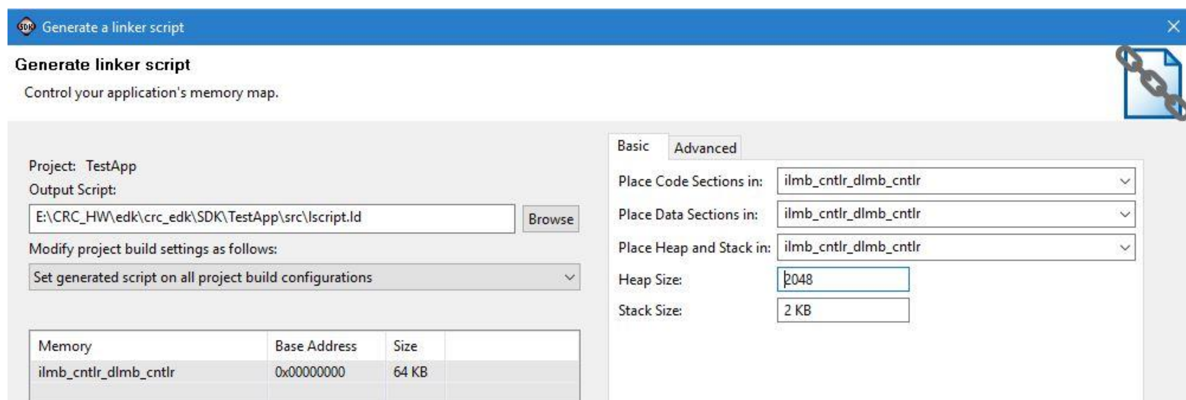


Figure 8.52: Generating Linker Script

8.3.5 Plan-Ahead

This tool is also one of Xilinx tools and it is used to make the following:

- Creating partitions and setting them as reconfigurable.
- Management of Reconfigurable Modules (RMs) and configurations.
- Creation of floorplans and AREA_GROUP RANGE constraints.
- Promote / import of static and reconfigurable logic.
- Partial Reconfiguration-specific DRCs.
- Verification of consistency among configurations.
- Bit file size estimates and resource reporting.

Resuming XPS and SDK flow:

1. Open Plan-Ahead then Create New Project.
2. Enable Partial Reconfiguration in the Post-synthesis Project.

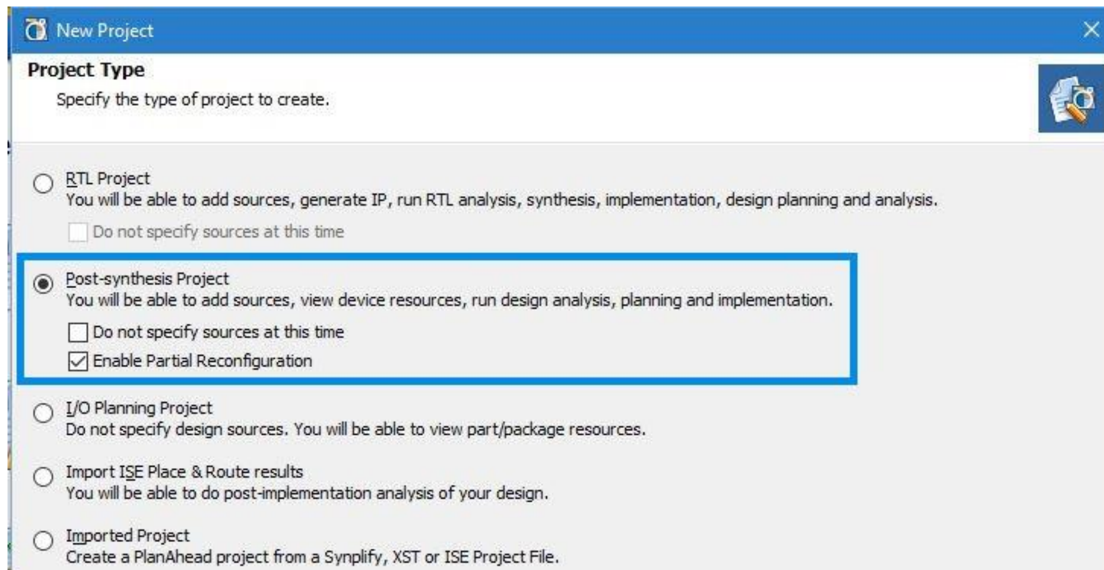


Figure 8.53: Enable Partial Reconfiguration

3. Browse to <Extract_Dir>/edk/ implementation/ and select all the netlist files (.ngc) of the system on chip that was implemented using Xilinx platform studio (XPS).

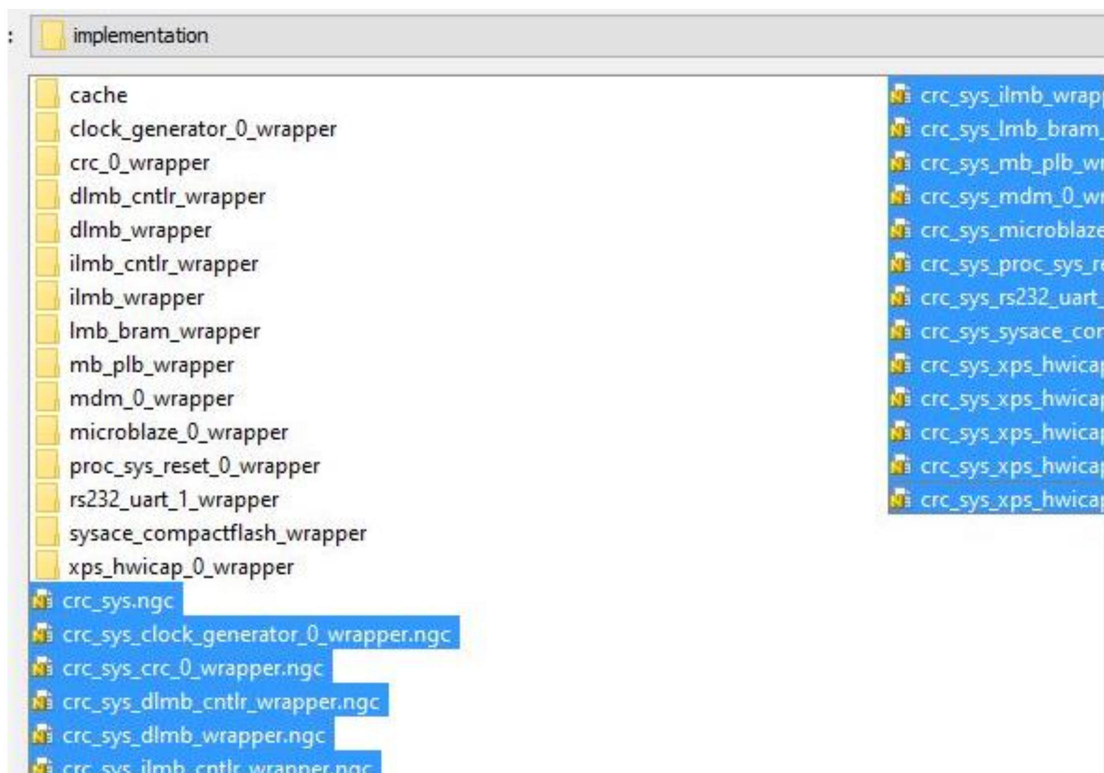


Figure 8.54: Selecting netlist files

4. Browse to <Extract_Dir>/edk/ data/ and select the UCF constrain file of the top level.

Figure 8.55: Selecting Constrain file

5. Choose Xilinx FPGA board from parts partition because the kit that we use is not defined in board's partition.
6. Click Open Synthesized Design under Netlist Analysis step of the Project Manager Flow Navigator pane to invoke the netlist files parser. This is necessary to access a lower-level module to define a reconfigurable partition.

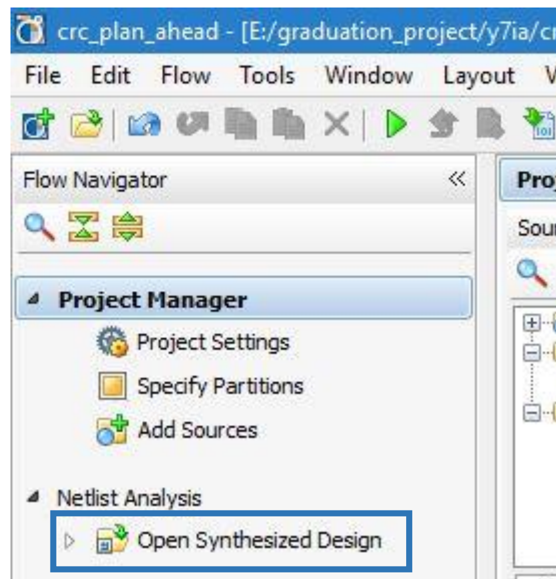


Figure 8.56: Netlist Analysis

7. Select crc_0/USER_LOGIC_I/rp_instance in the Netlist view then right-click and select Set Partition to create a reconfigurable partition.

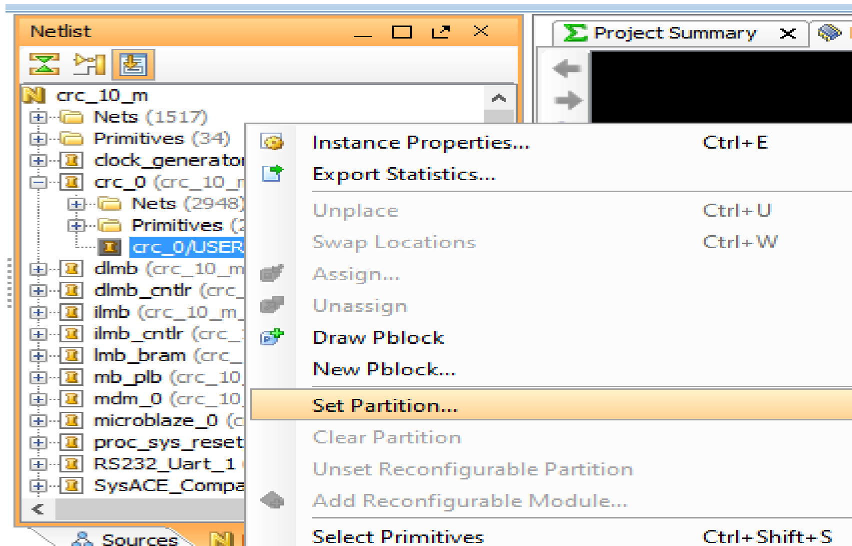


Figure 8.57: Set Reconfigurable Partition

8. Click Next twice then select Add this Reconfigurable module as a black box without a Netlist.

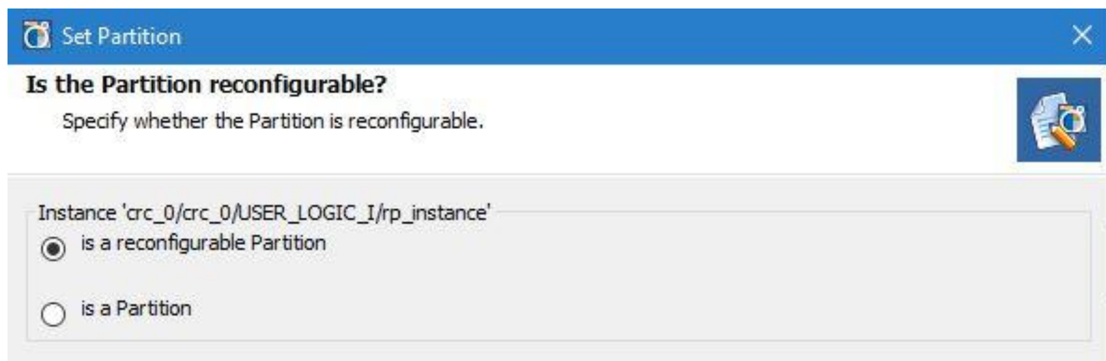


Figure 8.58: Reconfigurable Partition Initiation

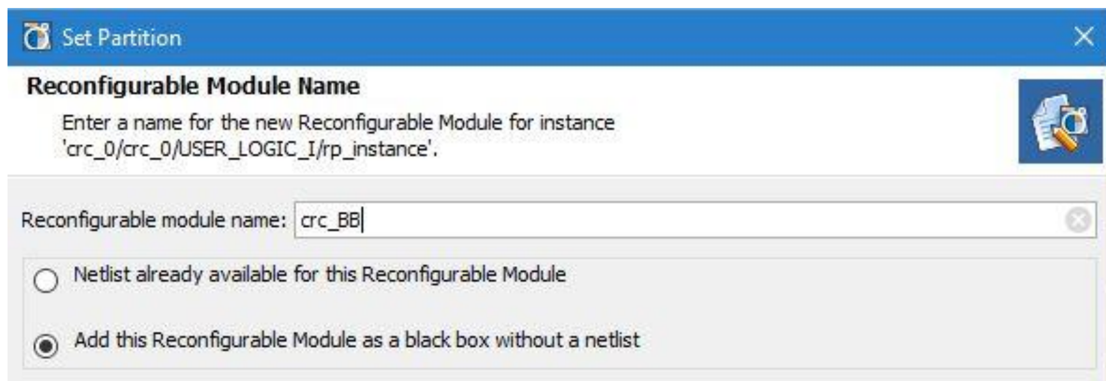


Figure 8.59: Adding Black Box Module

This design has two Reconfigurable Modules (RMs) for the Reconfigurable Partition (RP) as we explained before, now we are going to add the two modules.

a) Adding crc_4g module:

9. Select the math_0/USER_LOGIC_I/ rp_instance then right-click and select Add Reconfigurable Module.
10. Click Next then type crc_3g in the Reconfigurable Module Name field and verify that Netlist already available for this Reconfigurable Module is selected.

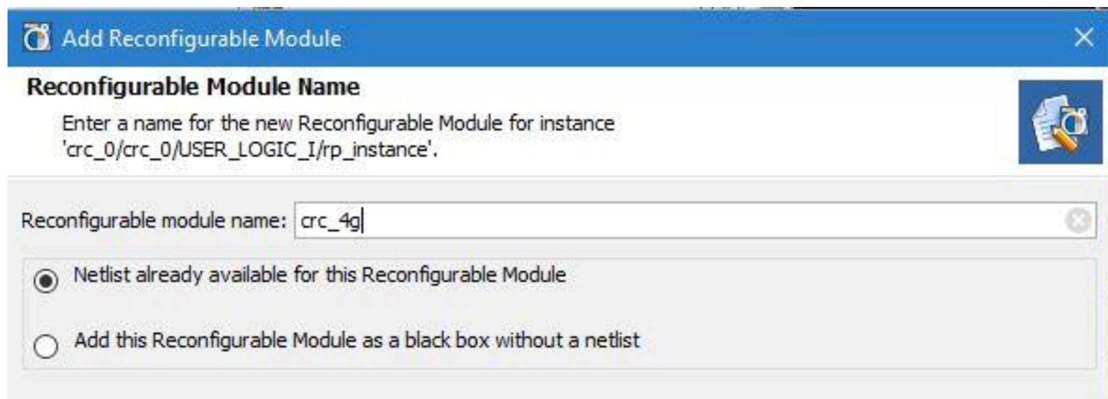


Figure 8.60: Adding CRC_4G Module

11. Click Next then browse to <Extract_Dir>/resources/CRC/crc_4g/ and select the top_crc_4g.ngc file.

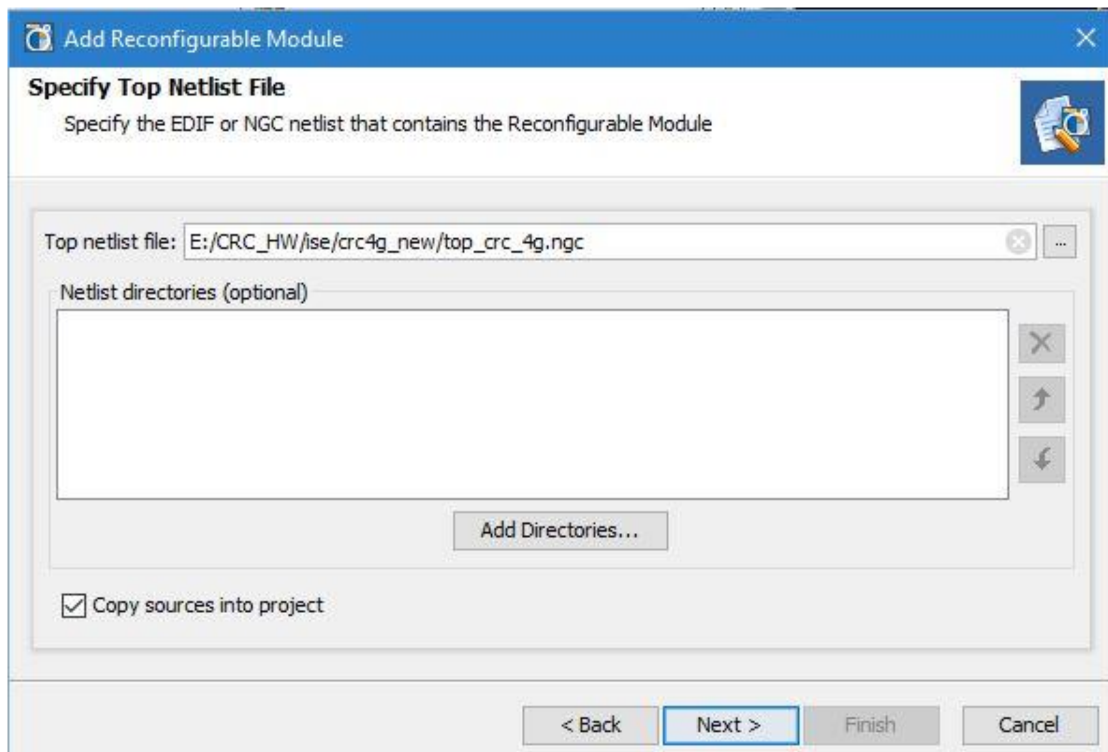


Figure 8.61: Adding Netlist file

b) Adding crc_3g module:

12. Follow the same steps, from step 9 to step 11, to add a crc_3g RM from the <Extract_Dir>/resource/CRC/crc_3g/top_crc_3g.ngc directory. Name the RM crc_4g.

Next, floorplaning for the RP region. Depending on the type and amount of resources used by each RM, the RP region must be appropriately defined so it can accommodate any RM variant.

13. Select Window > Physical Constraints,
14. Select pblock_crc_0/USER_LOGIC_I/rp_instance then Right-click and select Set Pblock Size.

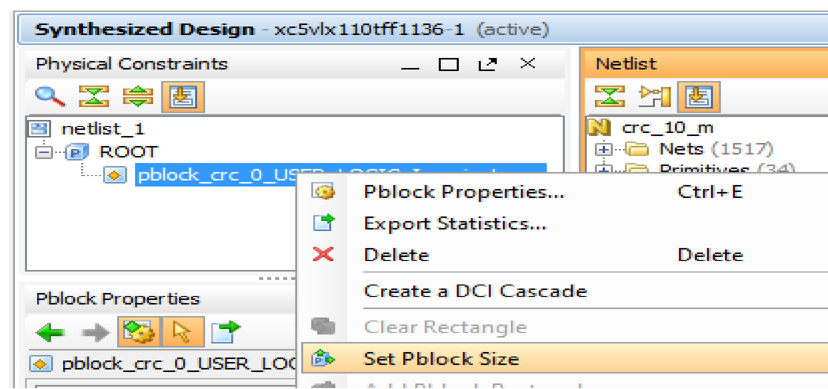


Figure 8.62: Set Pblock Size

15. Move the cursor in the Device window.
16. Click and drag the cursor to draw a box that bounds SLICE_X8Y230:SLICE_X17Y239.

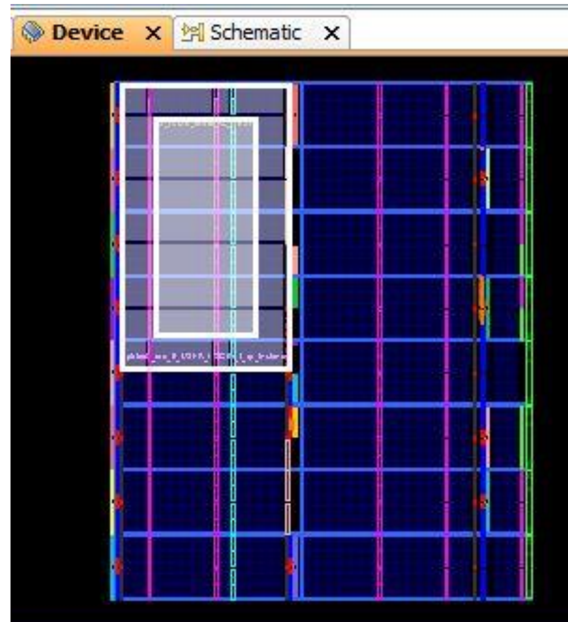


Figure 8.63: Floor Planning

Xilinx recommends that you run a Design Rule Check (DRC) in order to detect errors as soon as possible.

17. Select Tools > Report DRC then Deselect All Rules then Select Partial Reconfig.
18. Click OK to run the PR-specific design rules.

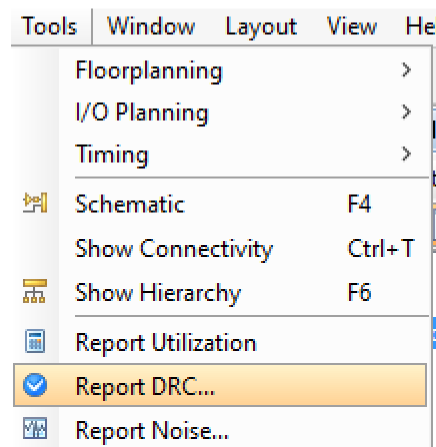


Figure 8.64: Design Rule Checking

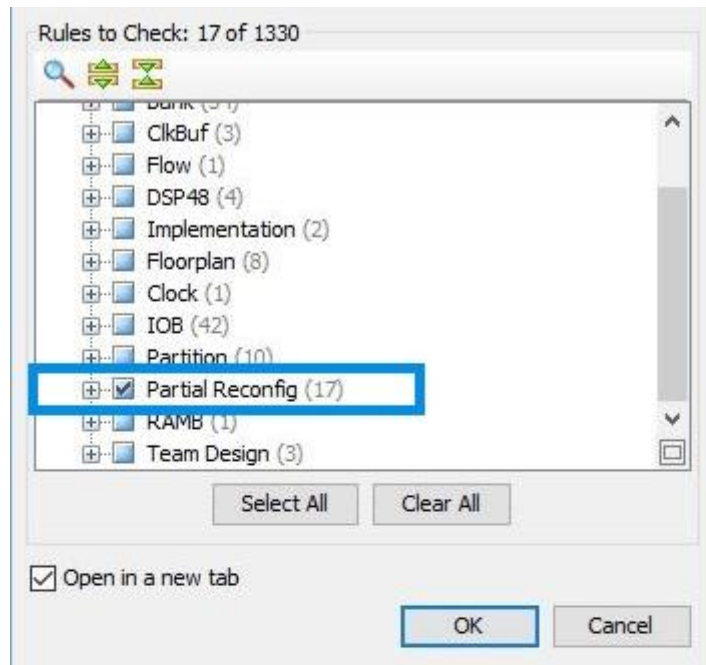


Figure 8.65: Define Rules to Check

Now you can create and implement the first configuration.

Creating a New Strategy by using the -bm option pointing to the crc_sys.bmm file for the new strategy.

19. Select Tools > Options then select Strategies in the left pane then select ISE 14 in the Flow drop-down box.
20. Under PlanAhead Strategies, select ISE Defaults and click the + button to create a new strategy then name the new strategy ISE14_BM, and set the Type to Implement.

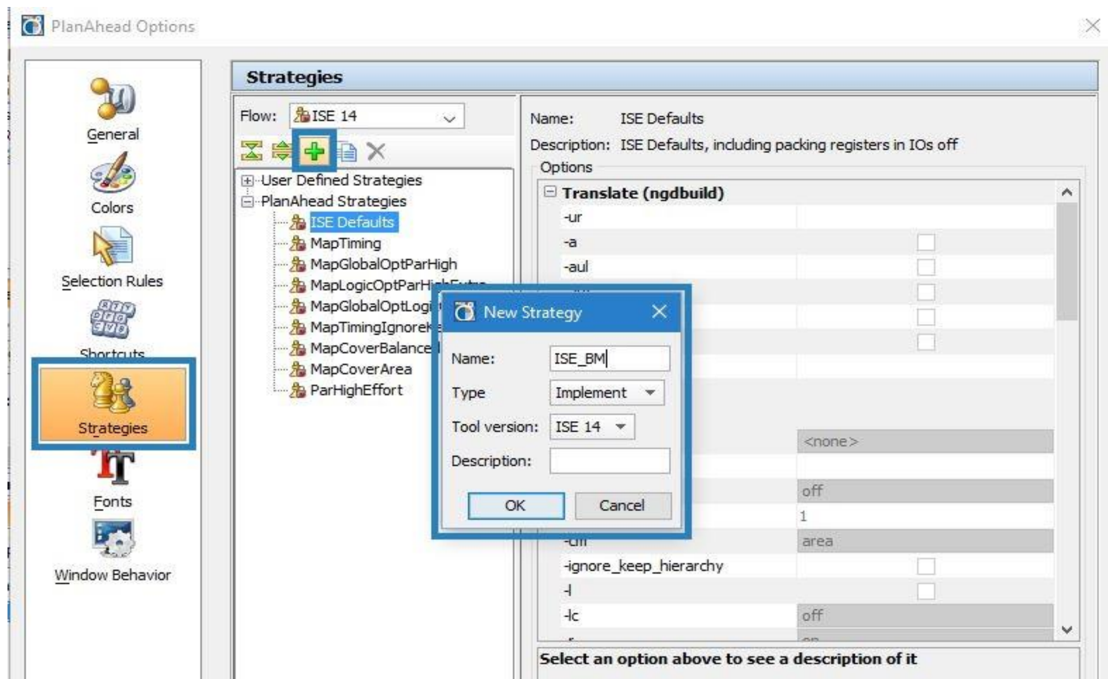


Figure 8.66: Creating new Strategy

21. Under Translate (ngdbuild), click in the More Options field and Type -bm ../../../edk/implementation/system.bmm then click ok.

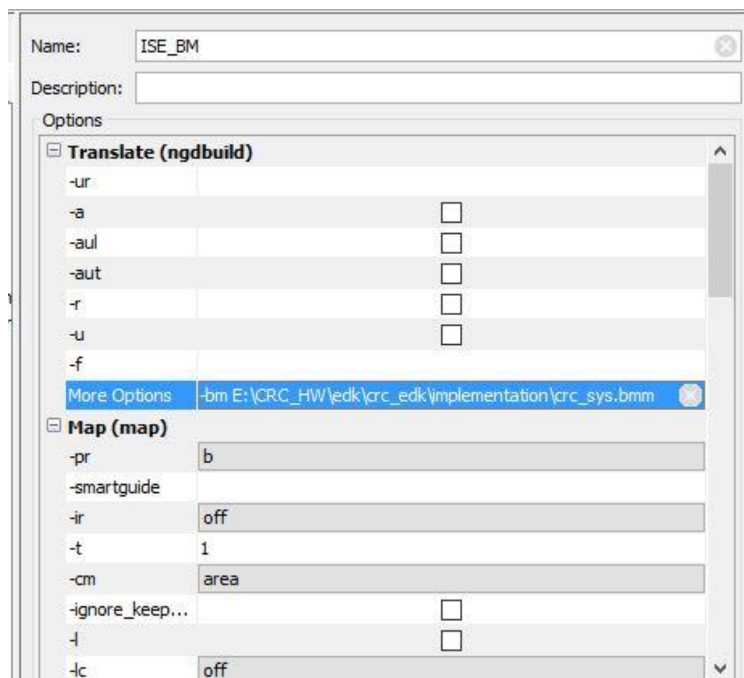


Figure 8.67: Selecting BMM File

Running the Implementation Using crc_4g as a Variant.

22. At the bottom of the PlanAhead tool user interface, select the Design Runs tab then Select the config_1 run.

23. In the Implementation Run Properties window, select the General tab. In the Name field, type crc_4g as the run name then click Apply to change the run name from config_1 to crc_4g.

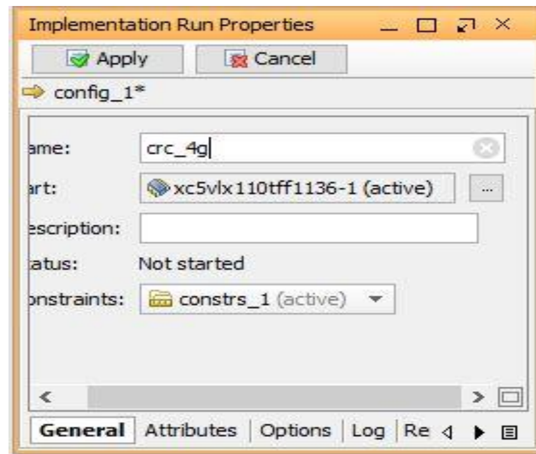


Figure 8.68: Defining Configuration

24. In the Options tab, change the Strategy to ISE14_BM and check that –bm option point to the correct directory then click Apply.

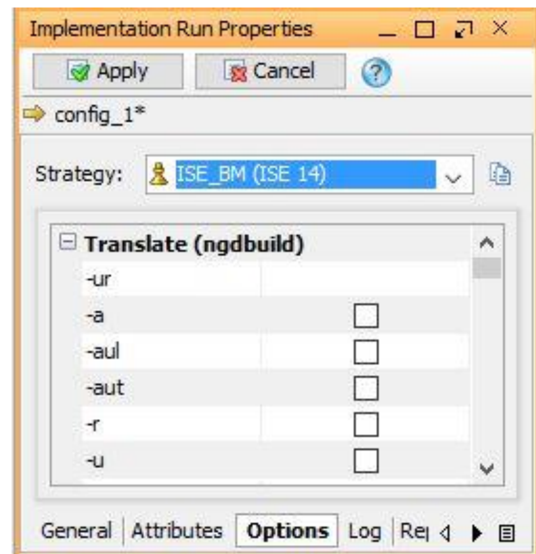


Figure 8.69: Selecting Strategy for Configuration

25. In the Partitions tab, click the Module Variant column drop-down button and select crc_4g as the variant then click apply.

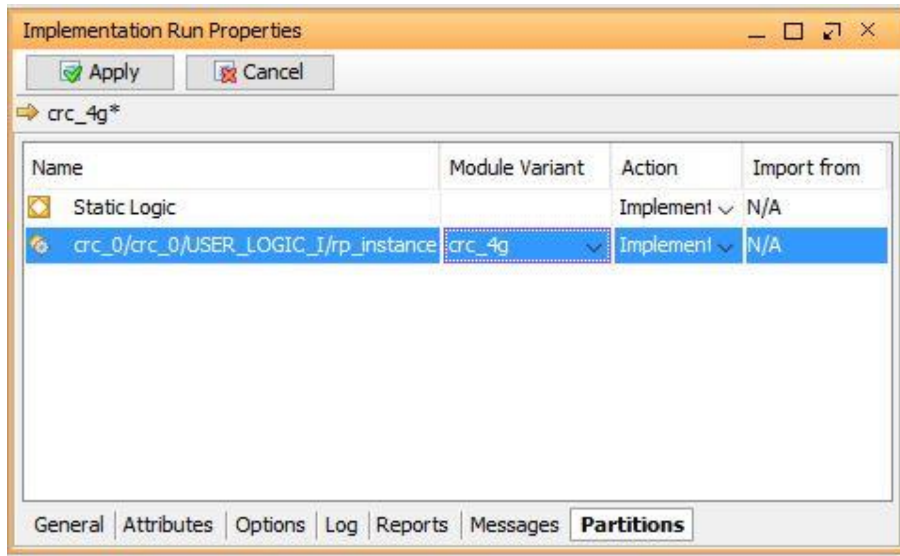


Figure 8.70: Selecting Partial Modules

26. In the Netlist window, make the corresponding module (crc_4g) active, right-clicking on its entry and selecting Set as an Active Reconfigurabe Module.

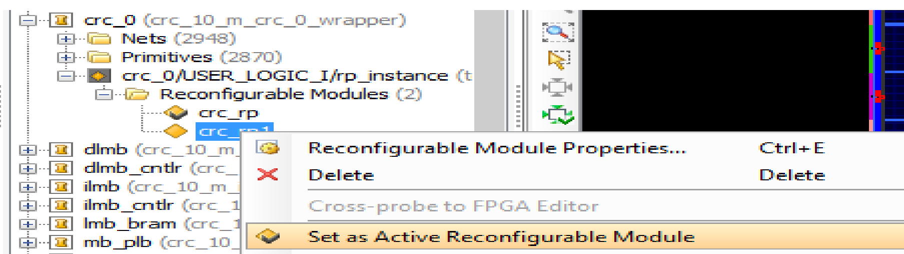


Figure 8.71: Activation Reconfigurable Module

27. In the Design Run window, select crc_4g, and right-click and select Launch Runs to run the implementation then select Launch Runs on Local Host.

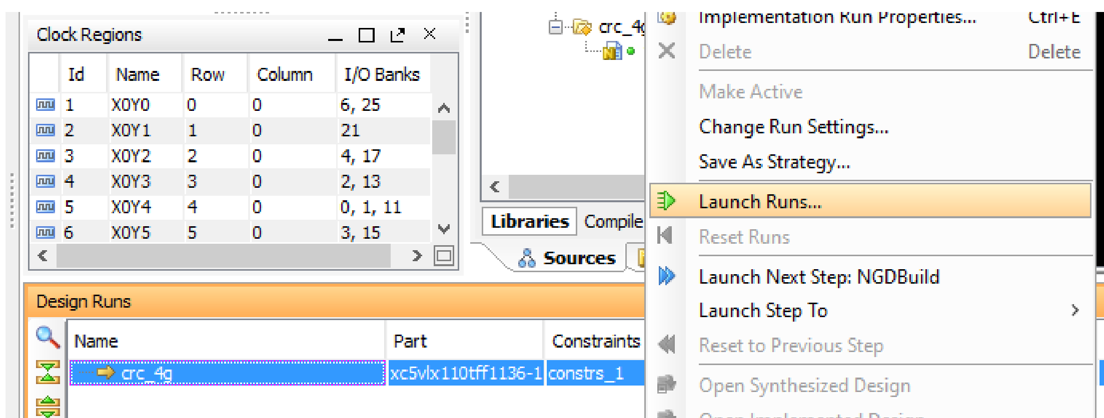


Figure 8.72: Launch Runs

Now we want to create another configuration for the crc_3g.

28. Select Flow > Create Runs. Click Next twice then in the Choose Implementation Strategies and Reconfigurable Modules page, change the name of the configuration from config_1 to crc_3g.

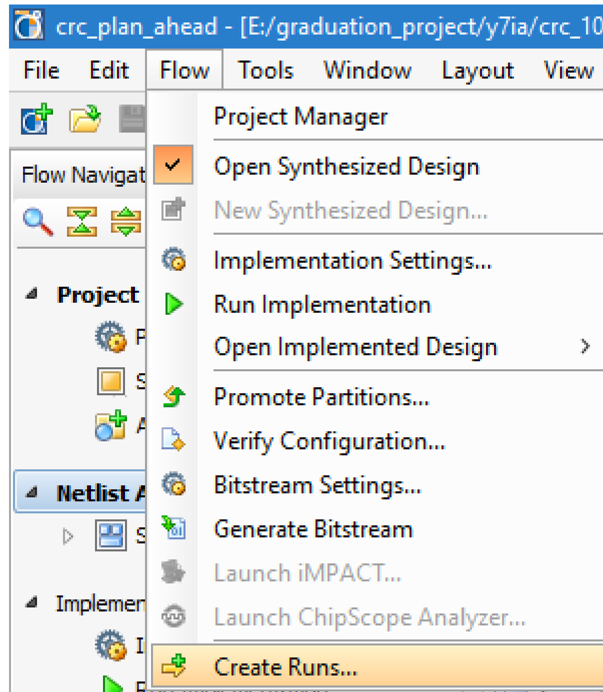


Figure 8.73: Create new Runs

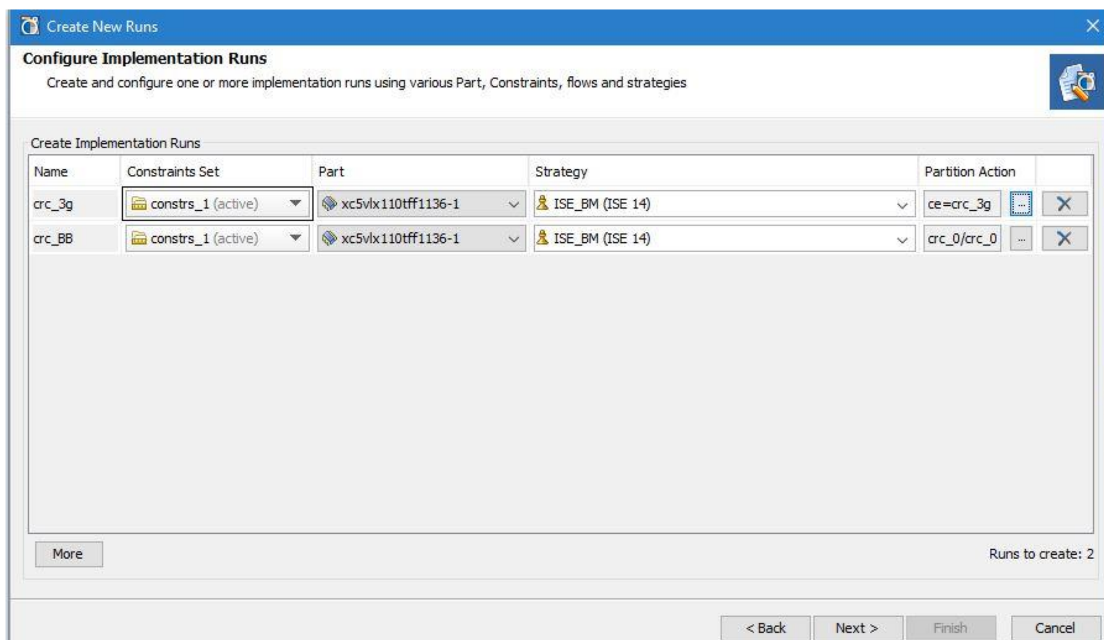


Figure 8.74: Configuration Run

29. In the crc_3g configuration row, click the Partition Action field.

30. For the rp_instance row, click the Module Variant column drop-down arrow, and select crc_3g as the variant to be implemented.

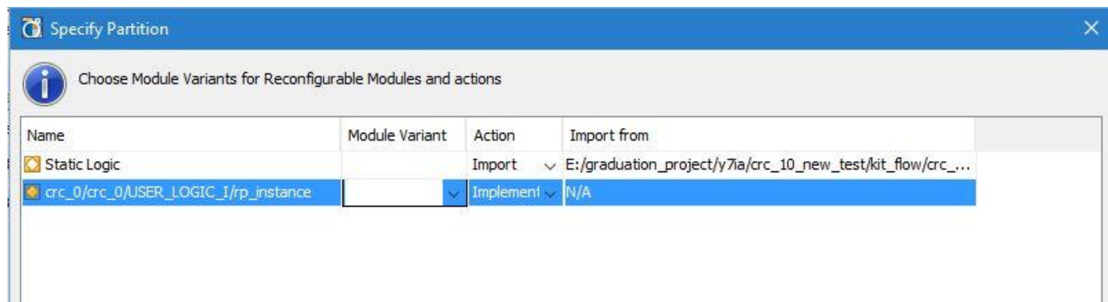


Figure 8.75: Specify Partition

31. In the Netlist window, expand the crc_0 > crc_0 > Reconfigurable Modules and rightclick on the adder, and select Set As an Active Reconfigurabe Module.
32. In the Design Run window, select crc_3g, and right-click and select Launch Runs to run the implementation then select Launch Runs on Local Host.

Repeat the same steps (from step 28 to step 32) to make a black box configuration (crc_BB).

Next, you will check to be sure that the static implementation, including interfaces to reconfigurable regions, is consistent across all configurations. To verify this, you can run the PR_Verify utility.

33. Run the PR_Verify utility to make sure that there are no errors. In the Configurations window, select any of the configurations then right-click, and select Verify Configuration.

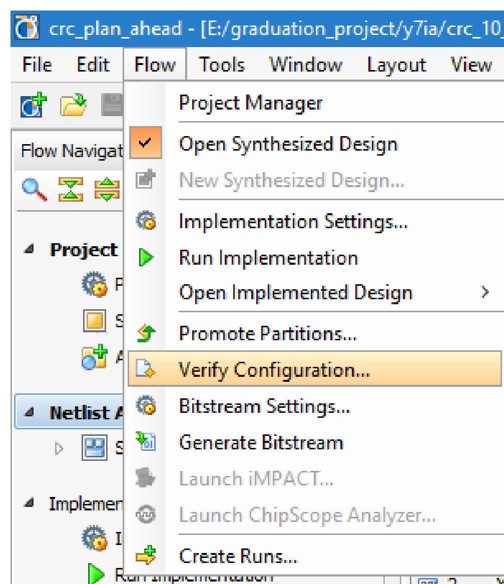


Figure 8.76: Verify Configuration

After PR_Verify validates all the configurations, you can generate full and partial bit files for the entire project.

34. Generate bitstream.

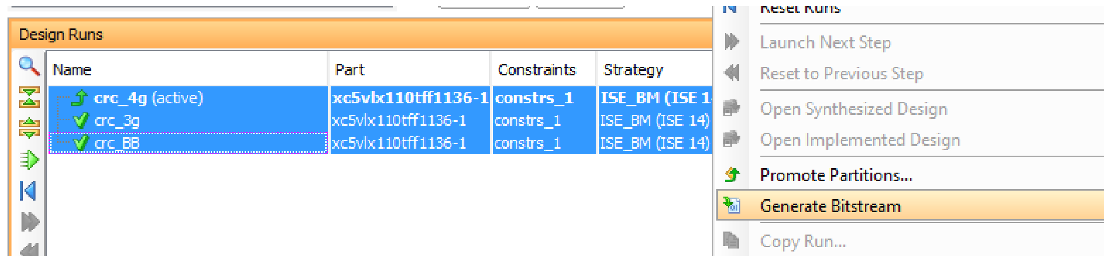


Figure 8.77: Generating BitStream

8.3.6 Generation of static system bootable ace files

For this step you need to open an EDK shell, and create both a download.bit and a crc_sys.ace file in the image/ directory. Copy the generated partial bit files, place them in the image/ directory, and name them crc_4g.bit and crc_3g.bit.

Resuming the flow:

1. Launch the ISE Design Suite command prompt from your Windows environment by selecting Start > All Programs > Xilinx Design Tools > Xilinx ISE Design Suite > Accessories > ISE Design Suite Command Prompt.
2. In the command window, go to the <Extract_Dir>/image/ directory.
3. Execute the following command to generate the download.bit file (with the software component included) from crc_4g.bit (with the hardware component) only.

```
   data2mem -bm ../edk/implementation/crc_sys_bd
   -bt ../PlanAhead/PlanAhead.runs/crc_4g/crc_4g.bit
   -bd ../edk/SDK/SDK_Export/TestApp/Debug/TestApp.elf tag
microblaze_0 -o b download.bit
```

This generates the download.bit in the image/ directory.

4. In the Bash shell, execute the following command to generate the crc_sys.ace file in the image/ directory.

```
   xmd -tcl genace.tcl -jprog -target mdm -hw download.bit -board
ml605 -ace crc_sys.ace
```

8.3.7 Summary of the hardware implementation flow

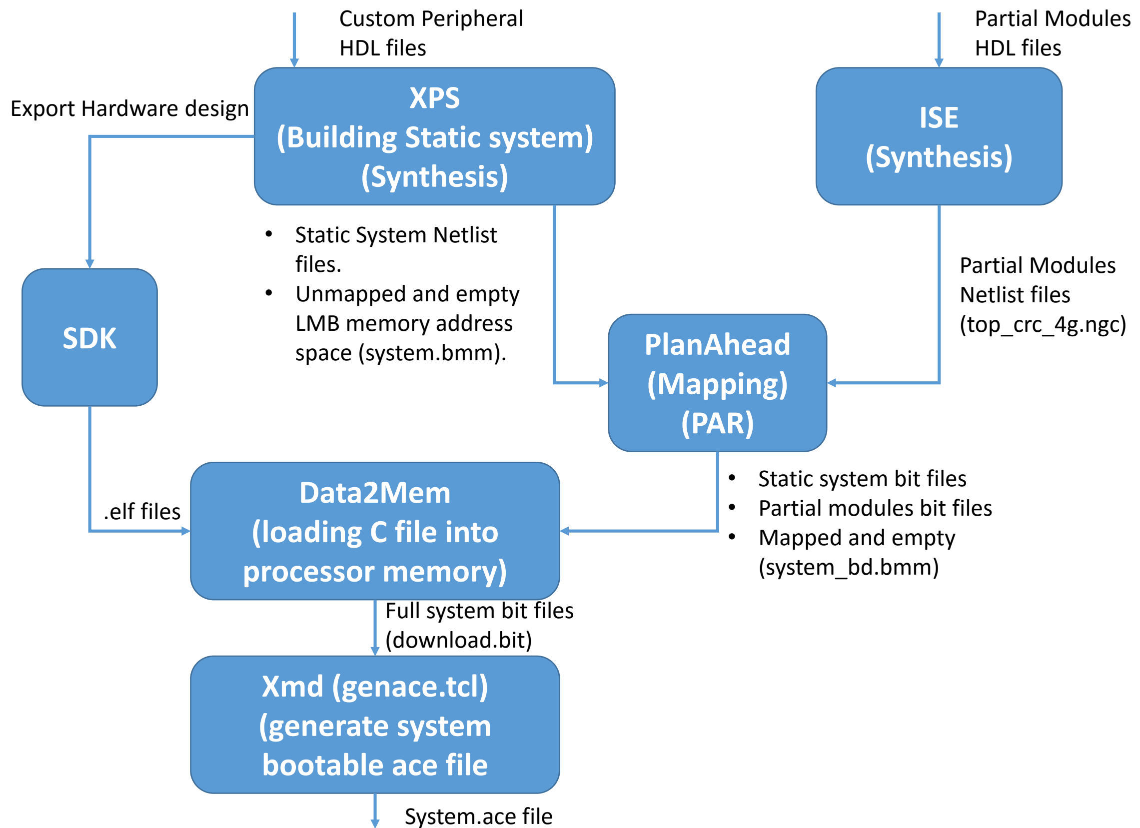


Figure 8.78: Hardware implementation block diagram

While seemingly complex when viewed in total, the system design flow simply combines the standard hardware flow used to create FPGA bitstreams and standard software flow used to create processor ELF files. In fact, unless on-chip memory resources are used to store the software image, the Embedded Developers Kit can be viewed as nothing more than an extension to the Xilinx core generation tool CoreGen.

The first step is to create the 'System Netlist' using the Embedded Developers Kit and instantiate that netlist into the design's HDL code. The hardware design is then synthesized, merged and implemented using the exact same flow as used with any other 'black box' core. While it is common to include a portion of the yet created software image inside the FPGA using block RAM.

The second step is to create the 'Board Support Package' (BSP) using the Embedded Developers Kit (EDK) and include the required drivers in the system's C code. The code is then compiled and linked with the various functions available in the BSP as is the same with any other processor system. Finally 'TestApp.elf' file (Compiled ELF file) is generated.

The third step is to create the partial dynamic reconfiguration using the PlanAhead tool as shown in **Error! Reference source not found.** this tool take from the XPS all the Netlist files (.ngc) ,UCF constrain file the ‘crc_sys.bmm’ file that contain the unmapped memory address space and take from the ISE the Netlist files of the reconfigurable module from ISE. After defining the different configurations, making floorplaning and generating the bitstream files finally the output files are, ‘crc_sys_bd.bmm’ file that contain the memory address space after mapping, ‘crc_4g.bit’ file (Compiled BIT file) created during this phase of development only contains the systems hardware description after defining the reconfigurable partition.

and ‘crc_4g_crc_0_crc_0_user_logic_i_rp_instance_crc_4g_partial.bit’ file which is the partial module bit file.

EDK provides a tool called Data2MEM which merges the appropriate sections of the ‘Compiled ELF’ file with the ‘Compiled BIT’ file. The resulting BIT file is typically created in a few seconds and can then be used to configure the FPGA. When the entire software image is stored within the FPGA, only the BIT file is needed to both configure the system and load the software image. If only portion of the software image, such as the bootstrap, is stored within the FPGA, then Data2MEM is run to create a combined BIT file and the system is once again configured/loaded as any two chip solution using the unmerged ELF sections and the combined BIT file.

So now we need to create ‘download.bit’ file (Combined BIT file) from ‘crc_sys.bmm’ file, ‘crc_4g.bit’ file (Combiled BIT file) and ‘TestApp.elf’ file (Combiled ELF file) using Data2MEM tool.

Unlike general purpose processors, the physical system can be probed using Chip-Scope modules. This capability provides a level of visibility into the operation of the system unmatched by external processors.

Chapter 9: Testing

9.1 Communication Methodology

We have 2 entities (Micro-Blaze processor & our Reconfigurable partition) that we want to establish a communication between though the PLB bus, They both run on the same clock system clock frequency (100 MHz) but the problem is that the processor code takes many cycles to write data in one register, so although they run on the same clock they are not synchronized, that's why a simple hand-shaking algorithm is used.

The Micro-blaze processor will send all data to slave registers and the RP will not read anything from the slave registers until the micro-processor sends an acknowledgement that he finished transferring data.

The same happens for data outputs, The RP starts writing its output to the slave registers and then writes an acknowledgement that it finished so the processor starts reading data.

Assuming a system of 11 slave registers the flow chart for the methodology of the micro-processor is shown in , the flow chart for the user logic is shown in, and the structure of the slave_registers is shown in Figure 9.1.

9.2 User Logic Code

As explained before user logic is the container that holds our RP and facilitate its communication with the micro-processor, the user logic code is auto-generated using the Create or Import Peripheral Wizard from Xilinx Platform Studio, The following is a quick walkthrough for this code.

The user logic code functions are:

- Communicate with PLB slave (auto generated, Not edited in the project).
- Store Micro-processor sent data to the slave register with the specified address (auto generated, Edited in the project).
- Communicate with the user IP (User code).

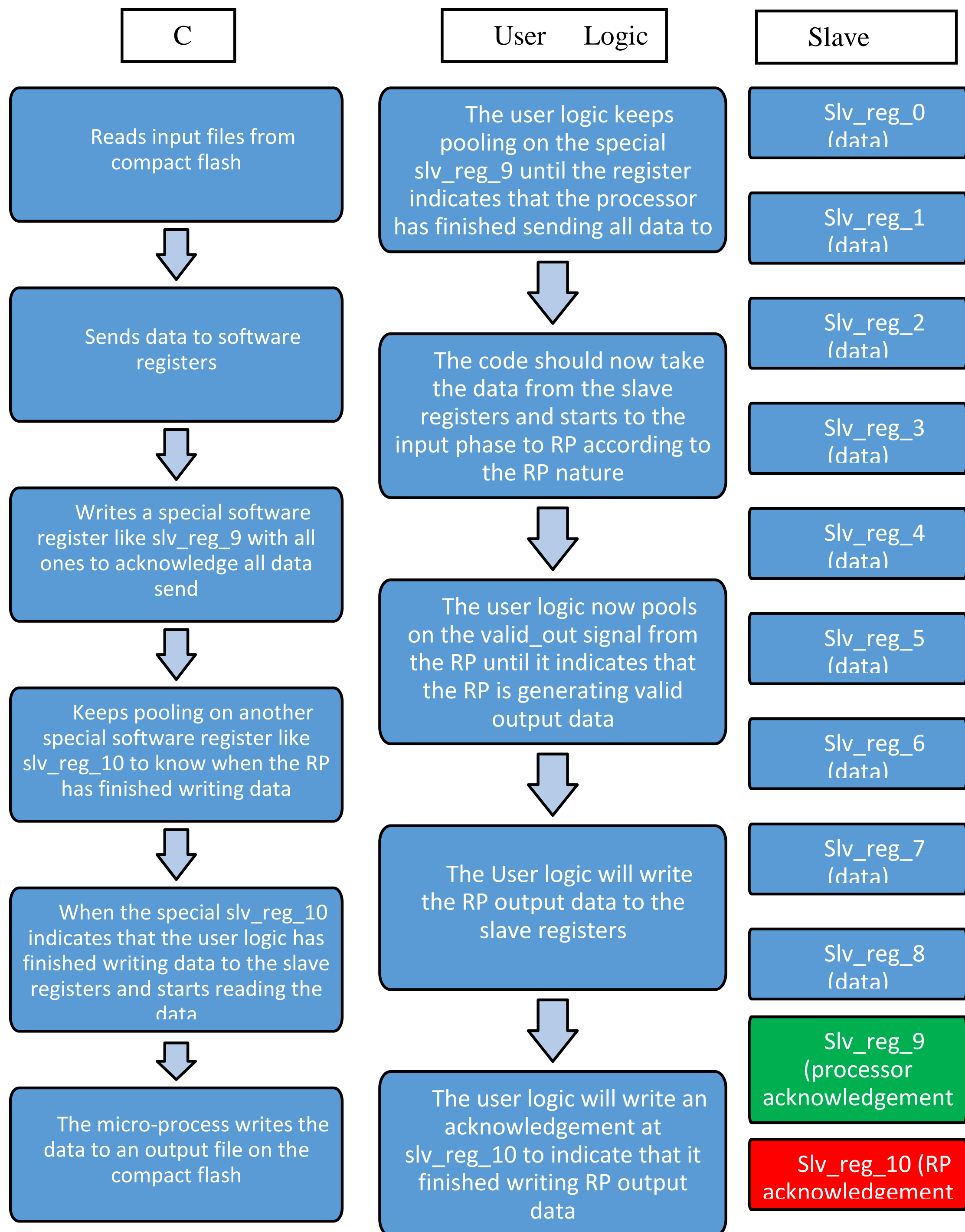


Figure 9.1: flow charts and slave registers structure

The typical auto generated user logic code contains:

- Port definitions and components (RP).
- A process block for writing processor data into slave registers as shown in Figure 9.2.
- A process block for sending salve registers data to processor when selected as shown in Figure 9.3.

```
-- implement slave model software accessible register(s)
  SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
  begin

    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Reset = '1' then
        slv_reg0 <= (others => '0');
        slv_reg1 <= (others => '0');
      else
        case slv_reg_write_sel is
          when "1000000" =>
            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg0(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
          when "0100000" =>
            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg1(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
```

Figure 9.2: writing processor data into slave registers

```
  -- implement slave model software accessible register(s) read mux
  SLAVE_REG_READ_PROC : process( slv_reg_read_sel, slv_reg0, slv_reg1, slv_reg2, slv_reg3, slv_reg4, slv_reg5, slv_reg6 ) is
  begin

    case slv_reg_read_sel is
      when "1000000" => slv_ip2bus_data <= slv_reg0;
      when "0100000" => slv_ip2bus_data <= slv_reg1;
```

Figure 9.3: reading data from slave registers to processor

Now as explained before to set a large test case many slave registers are like 128 registers or even 1024 registers, this will result in a very bad repetitive code which will be hard to develop and debug, to solve this problem the following edit was done to the user logic.

For easier large set of slave registers, the salve should instead be defined as an array of signals instead of separate signals; this will require an additional address encoder to convert the one-hot address send by the microprocessor to a normal index that could be used to access the slave register signal array.

The edits are shown in Figure 9.4, Figure 9.5.

```
 -- implement slave model software accessible register(s)
SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
  begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Reset = '1' then
    for i in 0 to (C_NUM_REG - 1) loop
        slv_reg(i) <= (others => '0');
    end loop;
      else
    if( not (reg_index_in = "10000000") ) then
            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg(to_integer(unsigned(reg_index_in)))(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
    else
        null;
    end if;
```

Figure 9.4: Slave registers array code

```
-- implement slave model software accessible register(s) read mux
SLAVE_REG_READ_PROC : process(slv_reg_read_sel,slv_reg(0),slv_reg(1),slv_reg(2),slv_reg(3),slv_reg(4),slv_reg(5),slv_reg(6),slv_reg(7)
begin
  if(not (reg_index_out = "10000000")) then
  slv_ip2bus_data <= slv_reg(to_integer(unsigned(reg_index_out)));
  else
  slv_ip2bus_data <= (others => '0');
  end if;
end process SLAVE_REG_READ_PROC;
```

Figure 9.5: Slave registers array processor read

The reg_index_in is the output of the encoder that is used to convert to one hot address from the processor to normal address, an important note is that the user logic cannot have any components unless they are added to the same library of the user_logic contrainer, that's why the encoder instantiation code is as shown in Figure 9.6. The rest of the user_logic code is used to convert the data from the slave registers to RP input.

```
library crc_v1_00_a;
use crc_v1_00_a.user_logic;
use crc_v1_00_a.encoder;
```

```
enc1: entity crc_v1_00_a.encoder
  port map (a  => slv_reg_write_sel,
        f => index_in
       );
```

Figure 9.6: encoder instantiation and library definition

9.1 SDK C Code

SDK c code is modified to be able to read input data from the input file that exist in the compact flash, transfer this input data to the system on chip (reconfigurable partition) (user logic), wait the acknowledge from the system on chip to inform the SDK c code that the output data is ready in the slave registers, read the output data from the system on chip and write it in the compact flash in the output file.

9.1.1 Sysace_read

This function is used to read input data from the input file that exist in the compact flash. The header of this function is as shown in Figure 8 , and as shown in Figure 240 that this function call another four functions:

- **sysace_fopen**

  This function is used to open the input file and its header is as following:

  ```
  void *sysace_fopen (const char *file, const char *mode)
  ```

  Parameters: file is the name of the file on the flash device. Mode is "r" or "w".

Returns: A non zero file handle on success.0 for failure.

User Defined Functions	calculation()	Sysace_read()	sysace_stdio.h • sysace_fopen() • sysace_fread() • sysace_fwrite() • sysace_fclose() stringtoint()
		bus2rp()	xil_io.h • Xil_Out32()
		rp2bus()	xil_io.h • Xil_In32
		Sysace_write()	sysace_stdio.h •sysace_fopen() •sysace_fwrite() inttostring() shuffle() stringlen()

Figure 240: User defined functions diagram

The file name should follow the Microsoft 8.3 naming standard with a file name of up to 8 characters, followed by a '.' and a 3 character extension. In this version of the library, the 3 character extension is mandatory so a sample file might be called test.txt. This function returns a file handle that has to be used for subsequent calls to read/write or close the file. If mode is "r" and the named file does not exist on the device 0 is returned.

- **sysace_fread**

This function is used to read the input file and its header is as following:

```
int sysace_fread (void *buffer, int size, int count, void *file)
```

Parameters: buffer is a pre allocated buffer that is passed in to this procedure, and is used to return the characters read from the device. Size is

restricted to 1. Count is the number of characters to be read. File is the file handle returned by sysace_fopen.

Returns: None zero number of characters actually read for success. 0 for failure.

The preallocated buffer is filled with the characters that are read from the device. The return value indicates the actual number of characters read, while count specifies the maximum number of characters to read. The buffer size must be at least count. Stream should be a valid file handle returned by a call to sysace_fopen.

- **sysace_fclose**

This function is used to close the input file and its header is as following:

```
int sysace_fclose (void *file)
```

Parameters: file is the file handle returned by sysace_fopen.

Returns: 0 on success and -1 on failure.

Closes an open file. This function also synchronizes the buffer cache to memory. If any files were written to using sysace_fwrite, then it is necessary to synchronize the data to the disk by performing sysace_fclose. If this is not performed, then the disk could possibly become corrupted.

- **stringtoint**

```
int stringtoint(char str[])
```

This function is used to convert a string to an integer. Since that the size of the memory is very limited so we can't use built in functions like atoi, itoa, strlen, pow,…..etc. so we created similar functions.

9.1.2 bus2rp

This function is used to transfer the input data to the system on chip slave registers (reconfigurable partition) (user logic). The header of this function is as shown in Figure 242.

- **Xil_Out32**

```
#define Xil_In32(Addr)  (*(volatile u32 *)(Addr))
```

Perform an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters: Addr contains the address to perform the input operation at.

Return : The value read from the specified input address.

9.1.3 rp2bus

This function is used to read the output data from the system on chip after waiting the acknowledging signal to inform the SDK c code that the output data is ready in the slave registers. The header of this function is as shown in Figure 243.

- **Xil_In32**

```
#define Xil_Out32(Addr, Value) \ (*(volatile u32 *)((Addr)) = (Value))
```

* Perform an output operation for a 32-bit memory location by writing the specified value to the specified address.

Parameters: Addr contains the address to perform the output operation at. Value contains the value to be output at the specified address.

Return : None.

9.1.4 Sysace_write

This function is used to write the output data in the output file that exist in the compact flash. The header of this function is as shown in Figure 244, and as shown in Figure 240 that this function call another four functions:

- **sysace_fopen** (as explained before)
- **sysace_fwrite** (as explained before)
- **inttostring**

  This function is used to convert an integer to a string without using built in functions duo to the memory size limitation.
- **Shuffle**

  This function is used to shuffle a string because the output string from inttostring function is reversed.
- **Stringlen**

  This function is used to find the length of a string without using built in function (strlen).

```
//-------------------------------Function to Read from System ace----------------------------------
int Sysace_read (Xuint32 * data_in ,Xuint16 * incount , Xuint8*flag1, Xuint8*flag2,Xuint8*flag3){
```

Figure 8: Header of Sysace_read function

```
//------------------------------Function to transfer input data to RP---------------------------------//
void bus2rp (Xuint32 data_in,Xuint16 * incount,Xuint8*flag2){
```

Figure 242: Header of bus2rp function

```
//----------------------------Function to transfer output data to RP---------------------------------//
Xuint8 rp2bus (Xuint32 * data_out,Xuint16 * incount,Xuint16 * outcount,Xuint8 * flag3){
```

Figure 243: Header of rp2bus function

```
//--------------------------------Function to write in System ace-------------------------------------//
int Sysace_write(SYSACE_FILE ** stream_output,Xuint32 data_out, Xuint8 * flag4,Xuint8 flag5){
```

Figure 244: Header of Sysace_write function

Chapter 10: Multiple Reconfigurable Partitions (RPs)

Till now we are able to reconfigure one RP so the process is as following

- Reconfigure the new partition (3G, 4G or WI-FI) using the PLB bus
- Handshake data between different internal blocks of this standard using the PLB bus
- After all the frame data is generated from the last internal block we reconfigure the new partition

This approach faces a lot of problems. First, we have to wait a long time until all data is generated from last block of the chain. In addition, we can't reconfigure each internal block using multiple partitions only. That's because of conflict of data on the PLB bus between several data out of different internal blocks and conflict between these data out and the reconfigurable data which is transferred using the same PLB bus.

To solve all these problems we use a new approach using multiple reconfigurable partitions with floating ports for the input and output data. By connecting the input floating port of the internal block with the previous block's output floating port we generate a new bus for the data flow far away from the PLB bus as shown in Figure 10.1. [12]

The new approach solves the problem of data conflict and adds a kind of pipelining for the flow. Now after each internal block finish its function we can reconfigure it with the next used standard internal block which decreases the overhead time greatly.

Now when the frame is finished the overhead time will be only the time of reconfiguring the last internal block of each standard.

To implement this approach on our standards chains we have to divide our chains to number of internal blocks which handshake the data and add the related blocks of the different standards in the same RP to change between them as shown in Figure 10.2 . After that we have to unify the interface of these related blocks in order to change between them properly and perform the PDR concept.

.

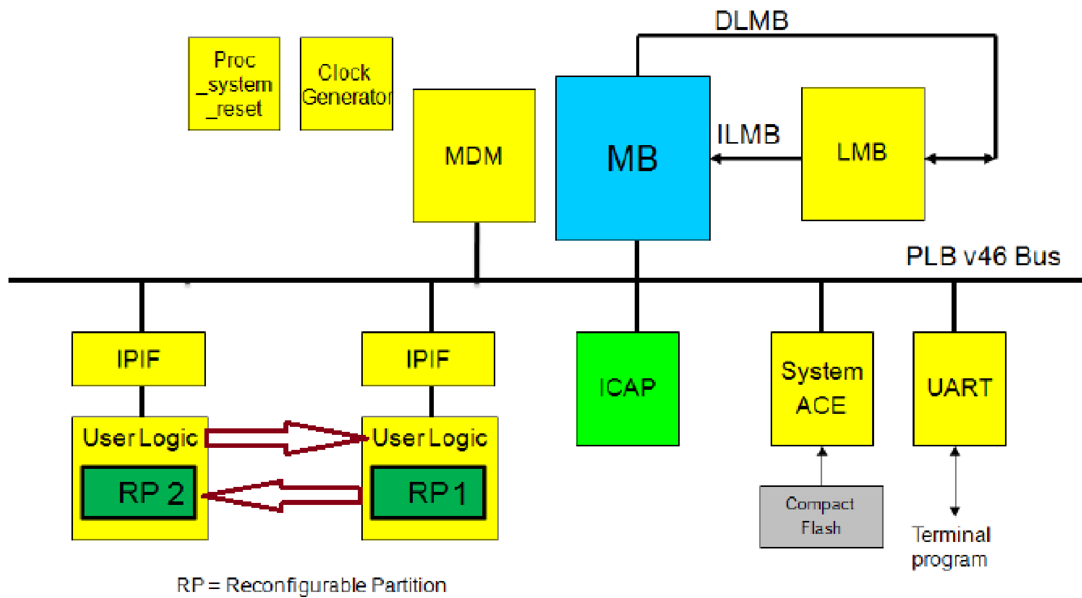


Figure 10.1-multiple RPs-system block diagram

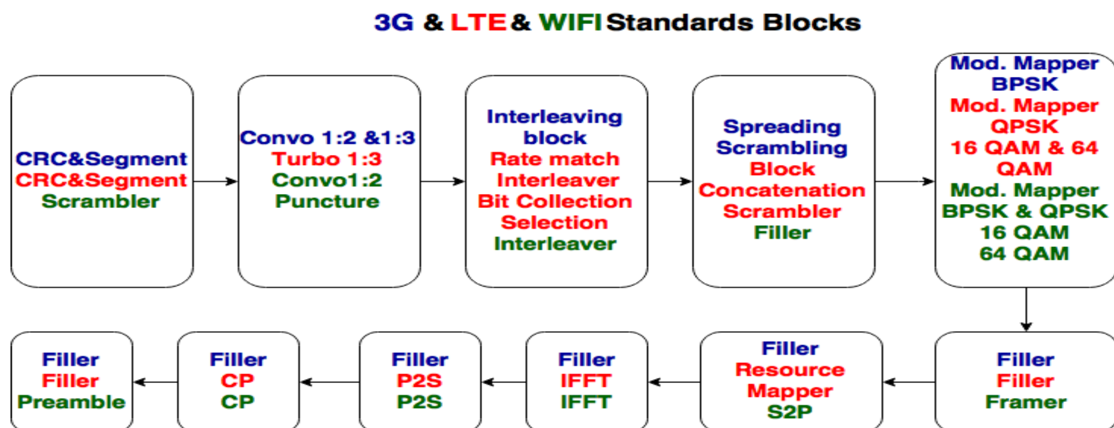


Figure 10.2-Standards Blocks

To prove the concept of multiple RPs, we made demo project to test the idea. The system contains two reconfigurable partitions, first contains two reconfigurable modules: Adder, multiplier. Second contains two reconfigurable modules: addition by 3 and addition by 4 as shown in Figure 10.3.

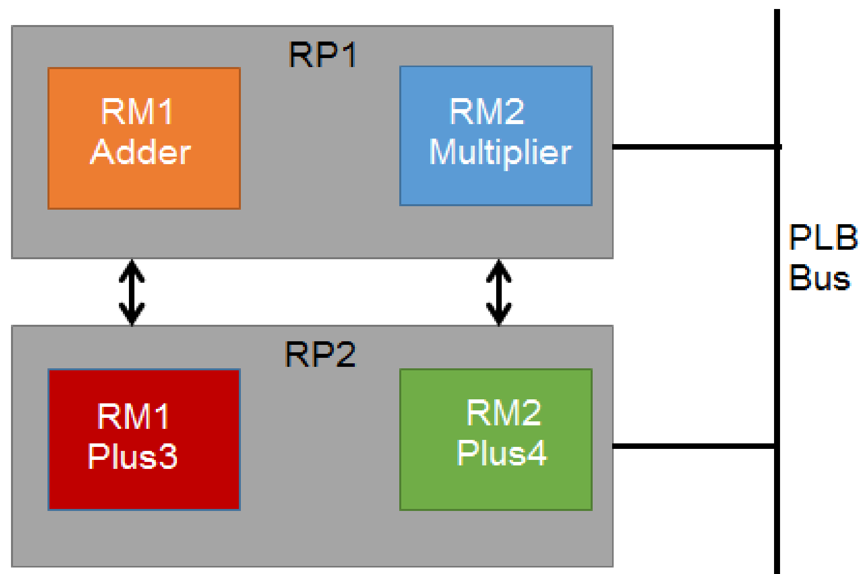


Figure 10.3-multiple RPs-Demo project

First, build the system on Xilinx platform studio (XPS). That needs modifications to user logic VHDL files as shown in Figure 10.4 and modifications to math VHDL files as shown in Figure 10.5.

In user logic1, math1 files, we added output port and connected it to output port of reconfigurable partition.

In user logic2, math2 files, we added input port and connected it to input port of reconfigurable partition.

We also adjusted mpd files to define the ports, directions of them and size of them as shown in Figure 10.6.

To connect the ports of math1, math2 together we need to edit the ports connection on Xilinx platform studio (XPS) as shown in Figure 10.7 .

Graphical Design View of Xilinx platform studio (XPS) shows the connection to verify that adjustment of files was correct as shown in Figure 10.7.

After that, we use Plan Ahead to generate bit files for different configurations. First we define the reconfigurable partitions and reconfigurable modules as shown in Figure 10.9.

```
entity user_logic1 is

port
(
  -- ADD USER PORTS BELOW THIS LINE ------------------
  --USER ports added here
  output_port  :  out std_logic_vector(0 to C_SLV_DWIDTH-1);
  -- ADD USER PORTS ABOVE THIS LINE ------------------


 --USER logic implementation added here
 rp1_instance : rp1
  port map (
     ain => slv_reg0,
     bin => slv_reg1,
     Clk => Bus2IP_Clk,
     Reset => Bus2IP_Reset,
     result => result
  );

 output_port<= result;
```

```
entity user_logic2 is

port
(
  -- ADD USER PORTS BELOW THIS LINE ------------------
  --USER ports added here
  input_port  :  in std_logic_vector(0 to C_SLV_DWIDTH-1);
  -- ADD USER PORTS ABOVE THIS LINE ------------------


 --USER logic implementation added here
 rp1_instance : rp2
  port map (
--    ain => slv_reg0,
      ain => input_port,
      bin => slv_reg1,
      Clk => Bus2IP_Clk,
      Reset => Bus2IP_Reset,
      result => result
  );
```

Figure 10.4-multiple RPs -user logic files

```
entity math1 is
 port
 (
  -- ADD USER PORTS BELOW THIS LINE ------------------
  --USER ports added here
  output_port_math : out std_logic_vector(0 to 31);
  -- ADD USER PORTS ABOVE THIS LINE ------------------

 output_port_math<=user_output_port;
 signal user_output_port :   std_logic_vector(0 to 31);
 USER_LOGIC_I : entity math1_v1_00_a.user_logic1

  port map
  (
   -- MAP USER PORTS BELOW THIS LINE ------------------
   --USER ports mapped here
   -- MAP USER PORTS ABOVE THIS LINE ------------------
   output_port    =>user_output_port,
   Bus2IP_Clk                  => ipif_Bus2IP_Clk,
   Bus2IP_Reset                => rst_Bus2IP_Reset,
   Bus2IP_Data                 => ipif_Bus2IP_Data,
   Bus2IP_BE                   => ipif_Bus2IP_BE,
   Bus2IP_RdCE                 => user_Bus2IP_RdCE,
   Bus2IP_WrCE                 => user_Bus2IP_WrCE,
   IP2Bus_Data                 => user_IP2Bus_Data,
   IP2Bus_RdAck                => user_IP2Bus_RdAck,
   IP2Bus_WrAck                => user_IP2Bus_WrAck,
   IP2Bus_Error                => user_IP2Bus_Error
  );
```

```
entity math2 is

port
(
  -- ADD USER PORTS BELOW THIS LINE ------------------
  --USER ports added here
  input_port_math : in std_logic_vector(0 to 31);
  -- ADD USER PORTS ABOVE THIS LINE ------------------

 user_input_port<=input_port_math;
 signal user_input_port :   std_logic_vector(0 to 31);
 USER_LOGIC_I : entity math2_v1_00_a.user_logic2

  port map
  (
   -- MAP USER PORTS BELOW THIS LINE ------------------
   --USER ports mapped here
   -- MAP USER PORTS ABOVE THIS LINE ------------------
   input_port    =>user_input_port,
   Bus2IP_Clk                  => ipif_Bus2IP_Clk,
   Bus2IP_Reset                => rst_Bus2IP_Reset,
   Bus2IP_Data                 => ipif_Bus2IP_Data,
   Bus2IP_BE                   => ipif_Bus2IP_BE,
   Bus2IP_RdCE                 => user_Bus2IP_RdCE,
   Bus2IP_WrCE                 => user_Bus2IP_WrCE,
   IP2Bus_Data                 => user_IP2Bus_Data,
   IP2Bus_RdAck                => user_IP2Bus_RdAck,
   IP2Bus_WrAck                => user_IP2Bus_WrAck,
   IP2Bus_Error                => user_IP2Bus_Error
  );
```

Figure 10.5-multiple RPs- math files

```
## Ports
PORT output_port_math = "output_port_math" , DIR = O, VEC = [0:31]

## Ports
PORT input_port_math = input_port_math , DIR = I, VEC = [0:31]
```

Figure 10.6-multiple RPs-mpd files

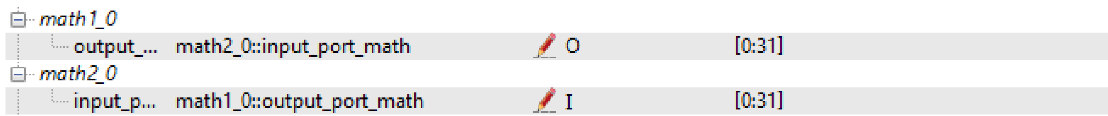


Figure 10.7-multiple RPs-ports connection

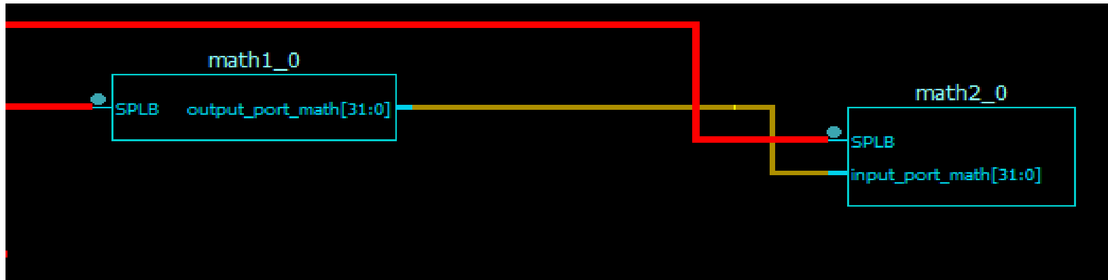


Figure 10.8-multiple RPs-graphical design view

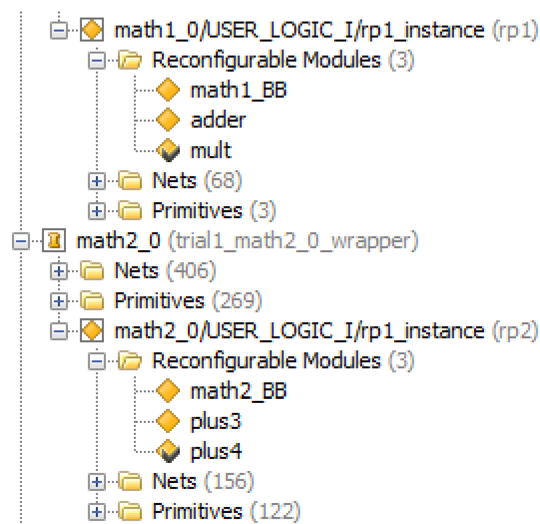


Figure 10.9-multiple RPs-Plan Ahead partitions

There are four different possible configurations: (adder+plus3), (adder+plus4), (multiplier +plus3), (multiplier+plus4) but each reconfigurable module should be used at least at one configuration so we can run only two configurations: (adder+plus3), (multiplier+plus4) as shown in Figure 10.9.

To test the system we only reconfigure one partition and check the effect on the output as shown in .the expected outputs are shown in Table 10.1 and Figure 10.11 shows the generated outputs which are identical to expected outputs.

Configuration	Module Variant
config1 (3)	
Static Logic	
math1_0/math1_0/USER_LOGIC_I/rp1_instance	adder
math2_0/math2_0/USER_LOGIC_I/rp1_instance	plus3
config2 (3)	
Static Logic	
math1_0/math1_0/USER_LOGIC_I/rp1_instance	mult
math2_0/math2_0/USER_LOGIC_I/rp1_instance	plus4

Figure 10.10: multiple RPs-Plan Ahead configurations

Table 10.1-multiple RPs-Expected outputs

First operand = 3 , second operand = 5		
	Adder	**Multiplier**
Plus3	11	18
Plus4	12	19

```
-- Entering main() --
System ACE Controller Initialized
After HWICAP LookupConfig
HWICAP Initialized
--------------------------------------------------------
      Press m or M for multiplication
      Press a or A for adder
      Press p or P for plus3
      Press r or R for plus4
      Press b or B for blanking configuration
      Press o or O to enter operands and display result
      Press q or Q to quit the demo

Performing reconfiguration for adder

Performing reconfiguration for plus3

First operand: 3
first: 3

Second operand: 5
second: 5
Result: 11

Performing reconfiguration for plus4

Result: 12

Performing reconfiguration for multiplier

Result: 19

Performing reconfiguration for plus3

Result: 18
```

Figure 10.11-multiple RPs-generated outputs

Chapter 11: Results

In this chapter we are going to conclude our achievements and results through a year full of team work, enthusiasm, hard work and research.

Our project is an experience of both hardware and software skills. And in our project we have been keen on verifying our results to make sure of the success of our work.

Here are some of the results of our work

- HDL and MATLAB implementation of 3G transmitter and receiver
- HDL and MATLAB implementation of WI-FI transmitter and receiver
- HDL and MATLAB implementation of LTE (4G) transmitter
- Building a test framework to Verify of HDL implementation
- Implementation of the three chains on the FPGA (Virtex 5)
- Generate and prove the concept of multiple RPs by implementing it on a simple example
- Prove the concept of PDR
- Debug the FPGA results using Chipscope
- Build a system on chip (SOC) with input and output files
- Reduce the total area and resources needed for implementation of the three standards. We choose the most consuming standard and its resources is the only needed resources to implement all the chains on this FPGA
- Reduce the total power of the system as we eliminate the static and sleep mode power consumed by the idle chains
- Reduce reconfiguration overhead by reconfigure each internal block of the chain after finishing its function. This is a kind of pipelining as we don't need to wait until all frame data is generated to reconfigure each internal block of the chain

Chapter 12: Conclusion and Recommendations

Conclusion:

After going through simulation and hardware implementation of PDR between different communication standards like 3G, 4G, WIFI, we can say that it is very useful to implement software defined radio in handsets using PDR. This enables us to reduce area and power used.

The motivation behind this sustainable work that Cairo University could be pioneer in this promising field and could build a system by hands and minds of ambitious engineers and researchers to be used by all industrial companies.

Recommendations

Moving towards the market and changing project's goal from just proofing concepts to implementing an applicable market project needs some improvement, These improvements also can be a suitable idea to broaden the horizon for future graduation projects. Throughout this chapter, we are going to figure out most of the enhancements that are required for the previously recalled chains as well as general developing for the project.

12.1 **3G**

Most of the blocks and specifications for the 3G standards are covered within the projects only few of blocks are needed to be added at the receiver since we assume that the transmitter and receiver are synchronized [6]. First of all, the rate matching and the turbo encoder at the transmitter are still not implemented. Also, the physical interface with the MAC layer has to be created as most of the parameters are forced during running since this part was out the project`s scope.

12.2 **LTE**

One of the main privileges for the LTE technology is boosting the rate for data transmission and reception. To have this advantage numbers of techniques are used;

one of these techniques is the modulation used for data bits that are stated in the following table. For our project, we use only the QPSK and 16QAM technique [13].

Secondly, LTE support the use of multiple antennas for sending and receiving data. These allow that more bits may be send at the same time by means of orthogonality. To implement this technique numbers of blocks are required to be added to support the MIMO. Figure 0.1 shows the block diagram for the MIMO implementation [13].

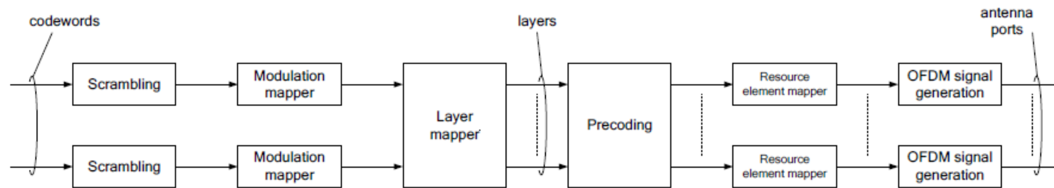


Figure 0.1: Block Diagram for MIMO

Moreover, for the OFDM block is consists of DFT followed by IFFT and the pilots insertion block. Pilots are bits assigned to specific subcarriers used for channel estimation at the receiver. Sequences of these bits are defined in standards for different versions of LTE. In this project we assume that these pilots are assigned to all ones since there is no channel estimation block at the receiver. Figure 0.2 shows the position for the pilots within the subcarriers.

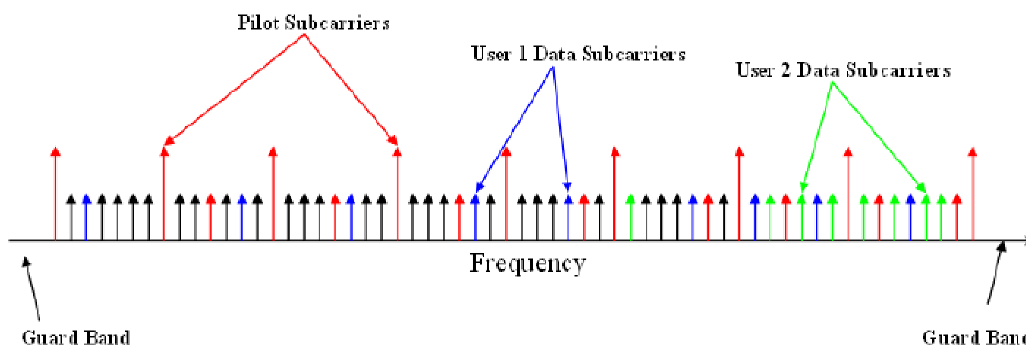


Figure 0.2: OFDMA Subcarriers Signals

Finally, some blocks are needed at the receiver for accurate reception of data such as synchronization block as well as the turbo decoder as the decoder need soft Viterbi instead of hard Viterbi (values rather than zeros and ones) [8].

12.3 WIFI

Timing synchronization block is the block required for accurate detection of the data at the receiver. Therefore, implementation of such a block is a challenging one. Timing synchronization performs two main functions packet detection and symbol timing. Packet detection is the task of finding an approximate estimate of the start of the preamble of an incoming data packet. It is the first synchronization algorithm that is performed.

The rest of the synchronization process is dependent on good packet detection performance. Also, one of drawbacks of OFDM is its sensitivity to carrier frequency offset.To solve this problem, implementation for frequency synchronization is required to be present in practical usage for Wi-Fi [9].

12.4 Others Improvements

Other standards may be implemented such as the Bluetooth standard and GSM standard. Regarding the switching algorithm used in the FPGA to switch between the standards may be improved to be more automated such without the need for user presence.

References:

[1] E. Grayver, Implementing Software Defined Radio.

[2] "wp374_Partial_Reconfig_Xilinx_FPGAs".

[3] 3GPP, Physical channels and mapping of transport channels onto physical channels (FDD) (Release 12), 2014.

[4] Ahmed Hamdy, Ahmed Khaled, Karim Mohammed, Yahia Ramadan, Yahia Zakaria, HSPA+ uplink physical layer simulation modelling and hardware impelementation, Cairo : Cairo University, Faculty of Engineering, 2011.

[5] 3GPP, Multiplexing and channel coding (FDD) (Release 12), 2014.

[6] 3GPP, 3GPP TS 36.212 - Multiplexing and channel coding (Release 12), 2015.

[7] 3GPP, Spreading and modulation (FDD) (Release 12), 2014.

[8] IEEE, IEEE 802.11-2012, Part 11: Wireless LAN Medium Access Control, 2012.

[9] Ahmed Magdy, Salah El-Din, Tarek Ibrahim, Mazen Ahmed, Mohamed El-Sayed, Yasser Samer, SDR Implementation of 802.11a receiver, Cairo: Cairo University, Faculty of Engineering, 2014.

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

[11] Xilinx, "LogiCORE IP, Fast Fourier Transform v7.1, Product specification," Xilinx, 2011.

[12] Xilinx, "Partial Reconfiguration of a Processor Tutorial," Xilinx, 2013.

[13] 3GPP, 3GPP TS 36.211 - Physical channels and modulation (Release 12), 2015.

[14] Mahmoud Hassan, Mourad Salih, Mustafa Alaa, Moustafa Mohamed, Moustafa Zaky , CU HSPA+ Phone Documentation book, Cairo: Cairo University, Faculty of Engineering, 2012.

[15] L. Z. Fuertes, OFDM PHY Layer Implementation based on the 802.11a Standard and System Analysis, 2005.

[16] 3GPP, 3GPP TS 36.213 - Physical layer procedures (Release 12), 2015.

[17] Sara M. Hassan, A. Zekry, M.A. Bayomy, G. Gomah, "Software Defined Radio Implementation of LTE Transmitter Physical Layer," *International Journal of Computer Applications,* vol. 74, no. 8, 2013.

[18] J. Zyren, "Overview of the 3GPP Long Term Evolution Physical Layer," freescale semiconductors, 2007.

[19] A. M. S. Mabrouk, SOFTWARE DEFINED RADIO USING DYNAMIC PARTIAL RECONFIGURATION, Cairo: Cairo University, Faculty of Engineering, 2016.

[20] Xilinx, LibXil FATFile System(FATfs), Xilinx, 2006.

[21] Xilinx, ML505/ML506/M507 Evaluation Platform, Xilinx, 2011.

[22] K. S. Arunlal, Dr. S. A. Hariprasad, "AN EFFICIENT VITERBI DECODER," *International Journal of Computer Science, Engineering and Applications (IJCSEA),* vol. 2, no. 1, 2012.

[23] Mr. Sandesh Y.M, Mr. Kasetty Rambabu, "Implementation of Convolution Encoder and Viterbi Decoder for Constraint Length 7 and Bit Rate 1/2," *Int. Journal of Engineering Research and Applications,* vol. 3, no. 6, 2013.

[24] M. 6. D. L. Notes, "Viterbi Decoding of Convolutional Codes," 2010.

[25] V. Stojanović, "Viterbi Algorithm -Implementation," Massachusetts Institute of Technology, 2006.

[26] IXIA, "Single Carrier FDMA in LTE," IXIA, 2009.

[27] Marius Pesavento, Willem Mulder, "LTE Tutorial part1 LTE Basics," Femto Forum Plenary, 2010.

[28] "SDR-White Paper".

[29] "http://www.wirelessinnovation.org/assets/documents/SoftwareDefinedRadio," [Online].

[30] "IJCSI International Journal of Computer Science Issues, Vol. 2, 2009".