MASTER INFORMATION BLOCK (MIB) DECODING PROCESSOR FOR 5G NR TECHNOLOGY

By

Abdelmonem Ahmed Abdelmonem

Ahmed Sameh Abdel-Sabour

Ahmed Sherif Sayed

Bassant Atef Gad-elkareem

Mona Mansour Amin

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 2023

MASTER INFORMATION BLOCK (MIB) DECODING PROCESSOR FOR 5G NR TECHNOLOGY

By

Abdelmonem Ahmed Abdelmonem

Ahmed Sameh Abdel-Sabour

Ahmed Sherif Sayed

Bassant Atef Gad-elkareem

Mona Mansour Amin

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

Under the Supervision of

Dr. Hassan Mostafa

Assistant Professor

Electronics and Communications Engineering Department

Faculty of Engineering, Cairo University

Faculty of Engineering, Cairo University
Giza, Egypt
July 2023

Acknowledgments

We would like to sincerely thank our supervisor, Dr. Hassan Mostafa, and STMicroelectronics’ engineers, especially Dr Abdelmohsen Ali, for their continuous support and guidance throughout our work.

Table of Contents

List of Tables

List of Figures

Abbreviations

Third Generation Partnership Project	3GPP
Analog to Digital Converter	ADC
Automatic Gain Controller	AGC
Advanced High-performance Bus	AHB
Advanced Peripheral Bus	APB
Bit Error Rate	BER
Block Length	BL
Cyclic Prefix	CP
Cyclic Redundancy Check	CRC
Discrete Fourier Transform	DFT
Decimation in Frequency	DIF
De-Modulation Reference Signal	DMRS
Down Link	DL
Fast Fourier Transform	FFT
First In First Out	FIFO
Frequency Range	FR
Half Frame	HF
Internet of things	IoT
Log-Likelihood Ratio	LLR
Master Information Block	MIB
Multiple Input Multiple Output	MIMO
Multiplexer	MUX
New Radio	NR
Orthogonal Frequency Division Multiplexing	OFDM
Peak to Average Power Ratio	PAPR
Path Metric	PM
Physical Broadcast Channel	PBCH
Physical Layer	PHY
Primary synchronization Signal	PSS
Quadrature Phase Shift Keying	QPSK
Resource Block	RB
Single path Delay Feedback	SDF
Signal to Quantization Noise Ratio	SQNR
Secondary Synchronization Signal	SSS
Signal-to-Noise Ratio	SNR
Synchronization Signal Block	SSB
User Equipment	UE
Up Link	UL
Virtual Reality	VR

Abstract

The 5th Generation Wireless Technology known as New Radio or NR is being developed by 3GPP (3rd Generation Partnership Project) which aims to address scenarios from Mobile Broadband to highly reliable communication with very low latency. The ability to operate in higher frequency bands, achieve uplink and downlink high data speeds in Gbps, and use advanced antenna systems such Massive MI-MO are some of the key advancements in 5G. The physical layer and other higher layers are also a part of the 5G NR radio air interface.

The technical specifications for NR published by the 3GPP with a focus on the physical layer offer ways to state-of-the-art realization and implementation of physical channels for both uplink and downlink. Synchronization signal (SS) and physical broadcast channel (PBCH) make up SS/PBCH block (SSB) in NR. To establish a connection between the UE and the eNB, cell search and identification are carried out using SSB.

The report aims at a detailed description on PBCH design, transmission, and reception subject to a time varying wireless channel. The project has two phases which are the MATLAB modeling for PBCH DL Transmitter and Receiver, Digital hardware design implementation for PBCH DL receiver.

PBCH transmitter is designed based on the technical specifications by 3GPP for 5G NR. For receiver channel estimate, PBCH data is generated and loaded with the Demodulation reference signal (DMRS). Since 5G NR uses OFDM for both uplink and downlink transmissions, the combined data thus produced is mapped on sub-carriers and transformed to time domain frames using the Inverse Fourier transform (IFFT). The time frames generated are convoluted with a time varying channel. The environment's noise and attenuation are represented by AWGN noise. FFT is used to convert the distorted and attenuated signal at the receiver to frequency domain. Channel estimation is performed using DMRS. The channel equalization equalizes the time varying channel effect on the symbols. Finally, the PBCH receiver's performance at specific SNR values is analyzed.

Chapter 1: Introduction

1.1 Motivation

5G New Radio (NR) is the latest global wireless network protocol developed by the Third Generation Partnership Project (3GPP) that provides faster and better mobile services compared to previous generations. 5G is capable of connecting billions of devices that share information in real time while providing stable and reliable connectivity. It introduces a huge shift in applications that require secure and reliable real time connectivity, such as the internet of things (IoT), virtual reality (VR), and many more.

To support the evolution from 4G to 5G and the different functionalities offered by the 5G network, the 3GPP defined a large set of protocols for transmitting user data and control information across all network layers.

In the physical (PHY) layer (Layer 1), the 3GPP defines a new signaling block, the Master Information Block (MIB). It contains the critical system parameters needed for radio resource management, channel quality reports, and higher layers' information. This block is broadcast on the physical broadcast channel (PBCH), where the PBCH resources are mapped with synchronization signals in a special segment of the resource grid called Synchronization Signal Block (SSB).

The SSB consists of three signals: Primary synchronization signal (PSS), Secondary synchronization signal (SSS) and Physical broadcast channel (PBCH). PSS and SSS are responsible for time domain synchronization while the PBCH payload contains the MIB. Hence, one of the PHY layer's main procedures is the downlink cell's synchronization which consists of time synchronization and MIB decoding.

MIB enables the synchronization of the user equipment (UE) with the network, it's used to convey the UE and network entities' parameters and capabilities, such as the carrier frequency, bandwidth, modulation, coding rate, and access control parameters of 5G NR. The UE must detect the MIB during the initial cell attach procedure. Hence, it plays a key role in the 5G network connection, is said to be the highest-priority data block in the 5G network, and is defined as the first information element in a message.

Multiple SSBs are periodically transmitted through the channel in a single Synchronization Signal (SS) burst. Each SSB within the burst contains the same MIB payload and is transmitted with a unique index, the SSB index, which corresponds to a specific beam. The goal of our processor is to successfully identify the SSB index and decode the MIB.

In this thesis, we are focusing on one building block within the NR modem and treating it for simplicity as an independent processor. This processor represents the full decoding chain of the MIB payload found in the PBCH and consists of three main subsystems: FFT subsystem, post-FFT subsystem and decoder Subsystem.

1.2 Organization of the Thesis

In this thesis our main focus is the modeling and the hardware implementation of the FFT and the PBCH processing Chain.

The rest of this thesis is organized as follows. Chapter 2 details the full system on chip specs and components starting from the used core, the Advanced High-performance Bus (AHB) address map and down to the used Advances Peripheral Bus (APB) slaves.

Chapter 3 provides the modeling of both FFT and PBCH processing chain to achieve the required system performance.

Chapter 4 introduces the FFT Subsystem. It illustrates the selection of the suitable architecture and optimization of it. Then shows hardware implementation of the FFT.

Chapter 5 discusses the implemented hardware architecture of the PBCH processing chain and how the testing process was implemented using the MATLAB reference model.

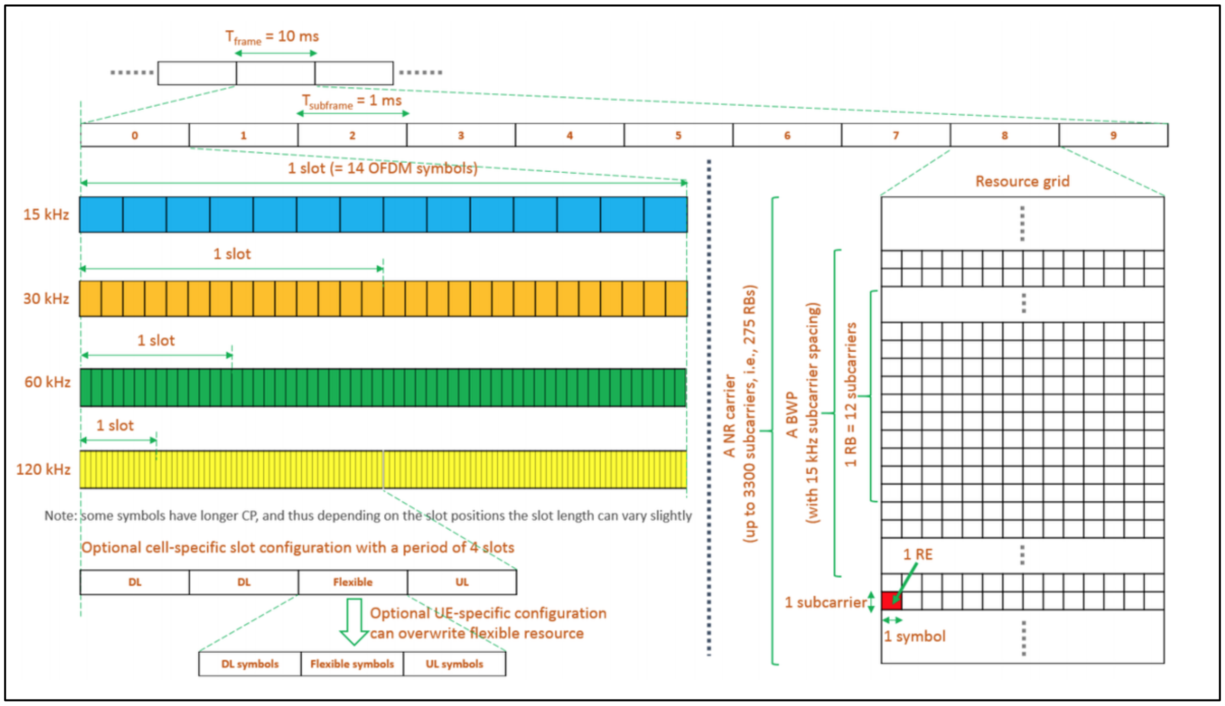
Chapter 6 introduces the integration of the PHY and the blind decoding process and how the processor operates to successfully decode the MIB payload.

1.3 Introduction to NR system

For any wireless access technology, the radio waveform plays a vital role in aspects of bandwidth and complexity for implementation. 5G NR has the requirements of wide bandwidth, very low complexity, and support for multiple antenna systems (MIMO). Therefore, 3GPP has adopted Orthogonal Frequency Division Multiplexing (OFDM) with a cyclic prefix for UL as well as DL.

There are two frequency ranges supported by NR:

- FR-1: called as sub-6 GHz band ranging from 450MHz to 6GHz
- FR-2: called as millimeter wave ranging from 24GHz to 52GHz



T_frame = 10 ms
T_subframe = 1 ms
0
1
2
3
4
5
6
7
8
9
1 slot (= 14 OFDM symbols)
15 kHz
1 slot
30 kHz
1 slot
60 kHz
1 slot
120 kHz
Note: some symbols have longer CP, and thus depending on the slot positions the slot length can vary slightly
Optional cell-specific slot configuration with a period of 4 slots
DL
DL
Flexible
UL
Optional UE-specific configuration can overwrite flexible resource
DL symbols
Flexible symbols
UL symbols
A NR carrier (up to 3300 subcarriers, i.e., 275 RBs)
A BWP (with 15 kHz subcarrier spacing)
1 RB = 12 subcarriers
Resource grid
1 RE
1 subcarrier
1 symbol

Figure 1: Frame Structure for 5G NR

OFDM numerology is scalable in NR to support a wide spectrum and diverse scenarios. The sub-carrier spacing is flexible and can be scaled from 15 kHz as in LTE to 2^{μ} * 15 kHz. The FR ranges determine the size of the cell. Lower frequencies result in larger cells for FR1, hence sub-carrier spacings of 15 kHz and 30 kHz are appropriate. For data and SSB channels at higher frequencies in FR2, the spacings employed are 60, 120, and 240 kHz. The cell size can be smaller, and delay spreads can be shorter too. As a result, the available spacing is adequate.

μ	SCS	No. of slots per subframe = 2^{μ}	No. of slots per radio frame = 10 * 2^{μ}	slot duration (ms)
0	15 kHz	1	10	1
1	30 kHz	2	20	0.5
2	60 kHz	4	40	0.25
3	120 kHz	8	80	0.125
4	240 kHz	16	160	0.0625

A frame in NR has a duration of 10 ms, as shown in Fig. 1. It consists of 10 sub-frames, each of 1 ms duration. This structure is common to both LTE and NR. Each sub-frame has slots based on the numerology, as shown in the above table.

$$\text{Slots/subframe} = 2^{\mu}$$

Each slot has 14 OFDM symbols, forming a typical small unit of transmission for NR to schedule. So, every frame consists of:

$$\text{Symbols/frame} = 2^{\mu} * 14 * 10$$

For example, if μ=2, then we have

- 1 Frame = 10 sub-frames.
- 1 sub-frame = 2^{μ} slots = 2 slots.
- 1 slot = 14 OFDM symbols.

Therefore, 1 frame = 2*10*14 = 280 OFDM symbols. Very low latency and minimum interference with other signals are achieved with such a short slot transmission. These slots are in the time domain but the data on sub-carriers is mapped in the frequency domain called as Resource Blocks (RBs). Resource blocks comprises of 12 consecutive sub-carriers piled up in the frequency domain as shown in the resource grid section in Fig. 1.

SS/PBCH block assists UE in performing initial cell search, by which UE acquires time and frequency synchronization with a cell and detects the physical layer cell ID of that cell. Each SSB includes Primary Synchronization Signal (PSS), Secondary Synchronization Signal (SSS) and PBCH data. PSS and SSS shall jointly convey the physical layer cell ID. The Master Information Block (MIB) is broadcast over the Physical Broadcast Channel (PBCH). DMRS support to decode MIB is provided, as shown in Fig. 2.

The number of SSBs (L) broadcasted within a half frame by gNB depends on the carrier frequency (fc). There are multiple periodicities supported for SS bursts by NR viz. 5, 10, 20, 40, 80 and 160 ms.

If	$fc < 3$ GHz: L=4
If	3 GHz $< fc < 6$ GHz: L=8
Else if	$fc > 6$ GHz: L=64

SSB burst enables transmission of each such block in different beams, which are received by UE and used to manage beams of other channels following. UE scans all the SSBs in a burst to get the SSB index. The measurement from all the blocks received helps to decide the best beam for UE to receive other channels, as shown in Fig. 3. In NR, unlike LTE, SSBs are flexibly placed, i.e., the position can be configured based on the numerology selected. A default burst period is 20 ms, but UE assumes the period to be 5 ms if the burst set is not available. After successful detection of SSB, UE is equipped with a cell ID and synchronized in time and frequency with gNB [7].

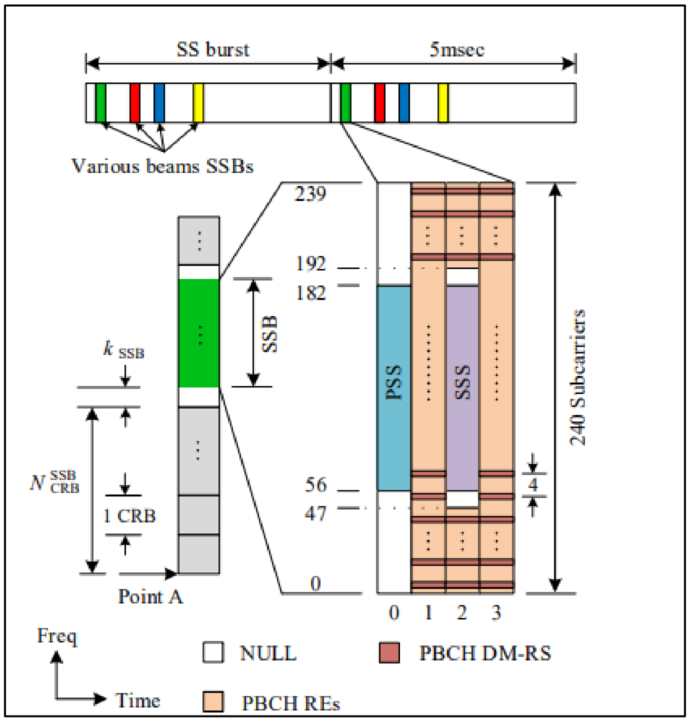


Figure 2: SSB Scheduling and Mapping

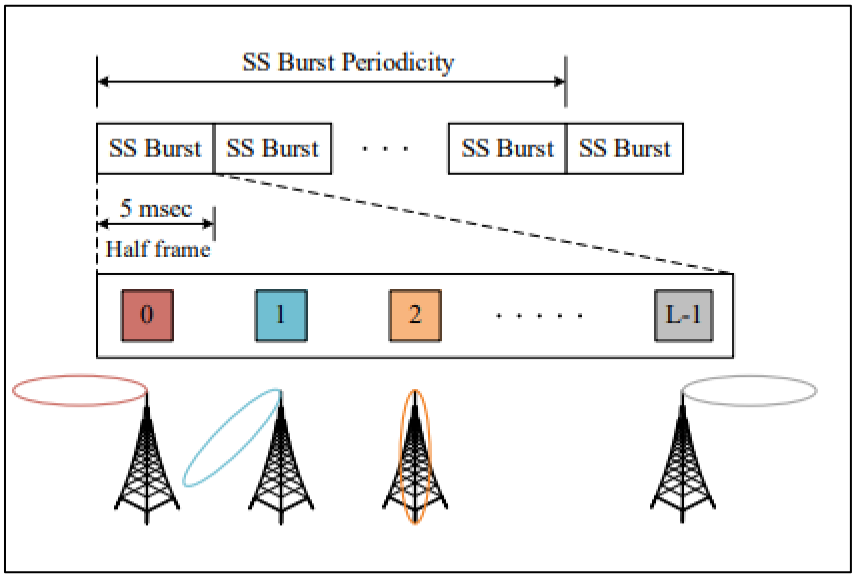
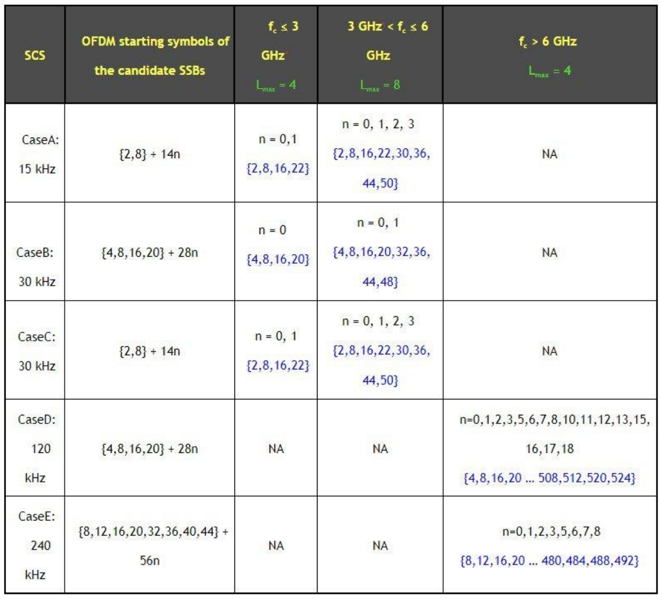


Figure 3: Beam Sweeping for various SSBs

A frame in NR is divided into two half frames of 5 ms duration each, HF0 and HF1. UE undergoing the cell search assumes a periodicity of 20 ms. The SS/PBCH block is spread over 4 OFDM symbols in the HF0 of the NR frame. Each of the synchronization signals, PSS and SS occupy 127 sub-carriers of the 1st and 3rd OFDM symbols of the SSB, respectively. PBCH occupies a total of 432 sub-carriers, and DMRS for PBCH is mapped in 144 sub-carriers over the 2nd, 3rd, and 4th OFDM symbols, as shown in Fig. 2.

240 sub-carriers are allocated to each OFDM symbol. Hence, the 2nd OFDM symbol in SSB is PBCH+DMRS, and the 3rd OFDM symbol in SSB is PBCH+SSS+DMRS. Here the SSS is the central part with 127 sub-carriers, and there are 48 sub-carriers of PBCH on either side of the SSS. There are some unused carriers in-between for guard, which are filled as nulls. For a half frame, the number of SSBs and index for the 1st OFDM symbol of the block are determined by the numerology assumed for the transmission.

The first symbol index of each possible SSB in the half frame is determined as shown in the following table and Fig. 4.



SCS	OFDM starting symbols of the candidate SSBs	$f_c \leq 3$ GHz $L_{max} = 4$	3 GHz $< f_c \leq 6$ GHz $L_{max} = 8$	$f_c > 6$ GHz $L_{max} = 4$
CaseA: 15 kHz	{2,8} + 14n	n = 0,1 {2,8,16,22}	n = 0, 1, 2, 3 {2,8,16,22,30,36, 44,50}	NA
CaseB: 30 kHz	{4,8,16,20} + 28n	n = 0 {4,8,16,20}	n = 0, 1 {4,8,16,20,32,36, 44,48}	NA
CaseC: 30 kHz	{2,8} + 14n	n = 0, 1 {2,8,16,22}	n = 0, 1, 2, 3 {2,8,16,22,30,36, 44,50}	NA
CaseD: 120 kHz	{4,8,16,20} + 28n	NA	NA	n=0,1,2,3,5,6,7,8,10,11,12,13,15, 16,17,18 {4,8,16,20 ... 508,512,520,524}
CaseE: 240 kHz	{8,12,16,20,32,36,40,44} + 56n	NA	NA	n=0,1,2,3,5,6,7,8 {8,12,16,20 ... 480,484,488,492}

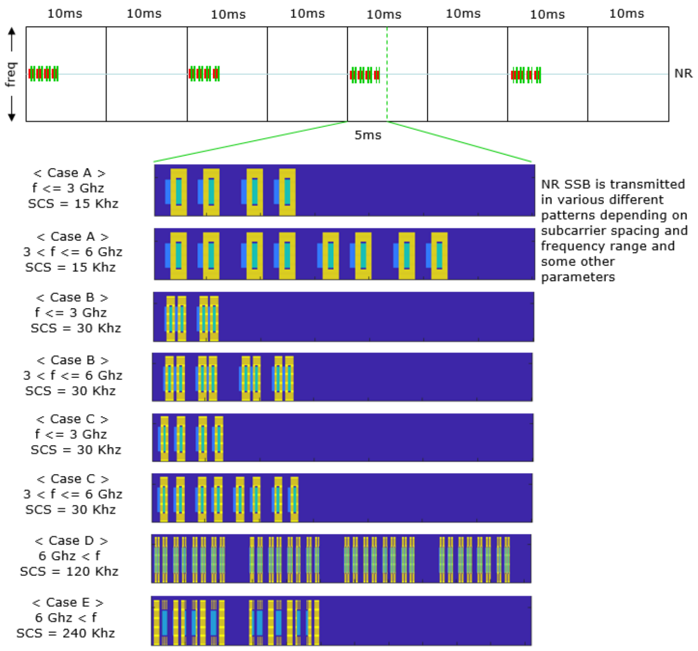

10ms
10ms
10ms
10ms
10ms
10ms
10ms
10ms
freq
NR
5ms
< Case A > f <= 3 Ghz SCS = 15 Khz
< Case A > 3 < f <= 6 Ghz SCS = 15 Khz
< Case B > f <= 3 Ghz SCS = 30 Khz
< Case B > 3 < f <= 6 Ghz SCS = 30 Khz
< Case C > f <= 3 Ghz SCS = 30 Khz
< Case C > 3 < f <= 6 Ghz SCS = 30 Khz
< Case D > 6 Ghz < f SCS = 120 Khz
< Case E > 6 Ghz < f SCS = 240 Khz
NR SSB is transmitted in various different patterns depending on subcarrier spacing and frequency range and some other parameters

Figure 4: Half Frame Structure

Chapter 2: System on Chip (SoC) Integration

2.1 Introduction

In this chapter, we will discuss the full system components, starting from the core (Cortex-M0) to the APB slaves. The Core is connected to multiple AHB slaves through an AHB Bus matrix that uses a unique memory map.

In our system, we had the following AHB slaves: Instruction and Data memories, General purpose input output (GPIO), the AHB to APB bridge and the PHY that contains the implementation of the MIB decoding chain that will be discussed in the upcoming chapters, figure (5).

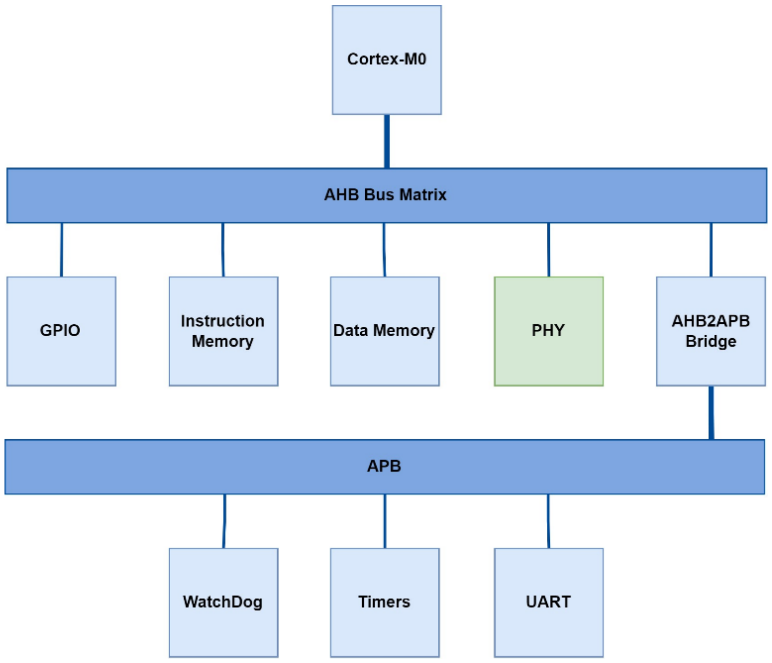


Figure 5: Full System Block Diagram

The AHB to APB bridge connects the Bus matrix to the following slaves: Timer, Watchdog and the UART. These slaves and their usage will be explained in the following sections.

2.2 System Core (Cortex-M0)

Could you imagine moving your limps without your brain? Of course not, so as a system, the system has many peripherals, but they want to talk with each other but how when each one has its own signals, standard and sequence? This is the processor mission. In this section, we talk about the kind of processor that we use to control our system which is cortex M0.

This processor is one of the smallest arm processors available. It has an exceptionally small silicon area, low power, and low cost. It is a 32-bit RISC ARM processor core licensed by ARM limited. The ultra-low gate count of the processor enables its deployment in analog and mixed devices. The block diagram of

Cortex-M0 is shown in figure (6). We discuss each of the main blocks of this diagram in the following sub-sections.

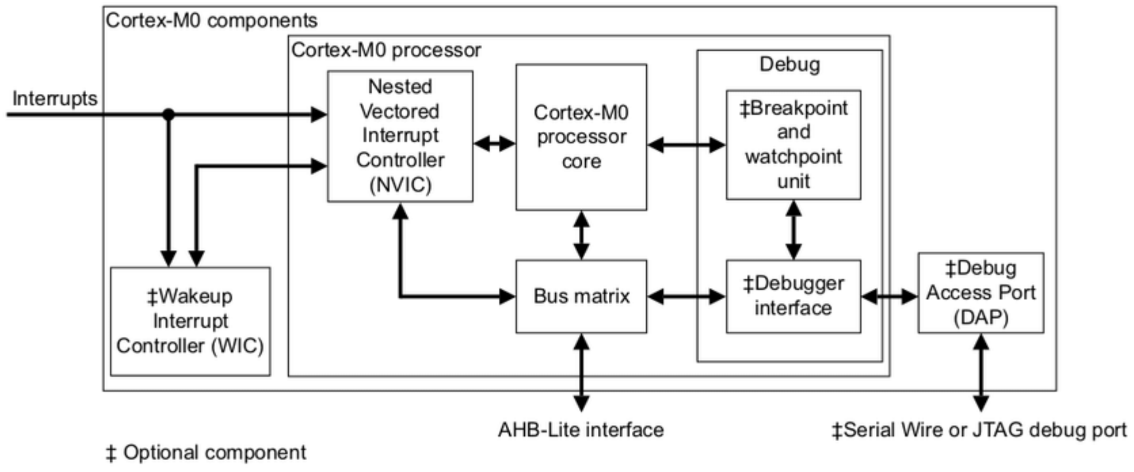


Figure 6: Cortex-M0 Block Diagram

2.2.1 Wakeup Interrupt Controller

The device might include a Wakeup Interrupt Controller (WIC), an optional peripheral that can detect an interrupt and wake the processor from deep sleep mode. The WIC is enabled only when the DEEPSLEEP bit in the SCR is set to 1. The WIC is not programmable and does not have any registers or user interface. It operates entirely from hardware signals.

When the WIC is enabled and the processor enters deep sleep mode, the power management unit in the system can power down most of the Cortex-M0 processor. This has the side effect of stopping the SysTick timer. When the WIC receives an interrupt, it takes several clock cycles to wake up the processor and restore its state before it can process the interrupt. This means interrupt latency is increased in deep sleep mode.

2.2.2 Nested Vector Interrupt Controller (NVIC)

This section describes the NVIC and the registers it uses. The NVIC supports:

- An implementation-defined number of interrupts, in the range 1-32.
- A programmable priority level of 0-192 in steps of 64 for each interrupt. A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority.
- Level and pulse detection of interrupt signals.
- Interrupt tail-chaining.
- An external NMI.

The processor automatically stacks its state on exception entry and unstacks this state on exception exit, with no instruction overhead. This provides low latency exception handling. The hardware implementation of the NVIC registers is shown in table 1.

Table 1: NVIC Registers

Address	Name	Type	Reset value	Description
0xE000E100	ISER	RW	0x00000000	*Interrupt Set-enable Register*
0xE000E180	ICER	RW	0x00000000	*Interrupt Clear-enable Register*
0xE000E200	ISPR	RW	0x00000000	*Interrupt Set-pending Register*
0xE000E280	ICPR	RW	0x00000000	*Interrupt Clear-pending Register*
0xE000E400-0xE000E41C	IPR0-7	RW	0x00000000	*Interrupt Priority Registers*

Interruput Set-enable Register

The ISER enables interrupts and shows the interrupts that are enabled. The bit assignments are shown in table 2.

Table 2: ISER Register

Bits	Name	Function
[31:0]	SETENA	Interrupt set-enabled bits. Write: 0 = no effect, 1 = enable interrupt. Read: 0 = interrupt disabled, 1 = interrupt enabled

If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

Interrupt Set-pending Register

The ISPR forces interrupts into the pending state and shows the interrupts that are pending. The bit assignments are shown in table 3.

Table 3: ISPR Register.

Bits	Name	Function
[31:0]	SETPEND	Interrupt set-pending bits. Write: 0 = no effect, 1 = changes interrupt state to pending. Read: 0 = interrupt is not pending, 1 = interrupt is pending

Writing 1 to the ISPR bit corresponding to:
- An interrupt that is pending has no effect.
- A disabled interrupt sets the state of that interrupt to pending.

Interrupt Clear-pending Register

The ICPR removes the pending state from interrupts and shows the interrupts that are pending. The bit assignments are shown in table 4.

Table 4: ICPR Register

Bits	Name	Function
[31:0]	CLRPEND	Interrupt clear-pending bits. Write: 0 = no effect, 1 = removes pending state interrupt. Read: 0 = interrupt is not pending, 1 = interrupt is pending

Writing 1 to an ICPR bit does not affect the active state of the corresponding interrupt.

Interrupt Priority Register

The interrupt priority registers provide an 8-bit priority field for each interrupt, and each register holds four priority fields is shown in table 5. This means the number of registers is implementation-defined and corresponds to the number of implemented interrupts. These registers are only word accessible.

Table 5: IPR Registers

Bits	Name	Function
[31:24]	Priority, byte offset 3	Each priority field holds a priority value, 0-192. The lower the value, the greater the priority of the corresponding interrupt. The processor implements only bits [7:6] of each field, bits [5:0] read as zero and ignore writes. This means writing 255 to a priority register saves value 192 to the register.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

The following steps show how to find the IPR number and byte offset for an interrupt *M*:

- The corresponding IPR number, *N*, is given by *N* = *N* DIV 4.
- The byte offset of the required Priority field in this register is *M* MOD 4, where:
  - byte offset 0 refers to register bits [7:0].
  - byte offset 1 refers to register bits [15:8].
  - byte offset 2 refers to register bits [23:16].
  - byte offset 3 refers to register bits [31:24].

2.2.3 Debug Access Port (DAP)

The processor has a low gate count Debug Access Port (DAP). This provides a Serial Wire or JTAG debug-port and connects to the processor slave port to provide full system-level debug access. The DAP enables communication between the core and the device pins during debug.

The Debug Access Port enables the following:

- Halting, resuming, and single stepping of program execution.
- Access to processor core registers and special registers.
- On-the-fly memory access.
- Data watchpoints.
- HW/SW breakpoints.
- PC sampling for basic profiling.

2.2.4 Vector Table

The vector table contains the reset value of the stack pointer, and the start addresses, also called exception vectors, for all exception handlers. Table 6 shows the order of the exception vectors in the vector table. The least-significant bit of each vector must be 1, indicating that the exception handler is written in Thumb mode. The vector table is fixed at address 0x00000000.

Table 6: vector Table

Exception number	IRQ number	Vector	Offset
16 + n	n	IRQn	0x40+4n
.		.	.
.		.	.
.		.	.
18	2	IRQ2	0x48
17	1	IRQ1	0x44
16	0	IRQ0	0x40
15	-1	SysTick, if implemented	0x3C
14	-2	PendSV	0x38
13 12		Reserved	
11	-5	SVCall	0x2C
10 9 8 7 6 5 4		Reserved	0x10
3	-13	HardFault	0x0C
2	-14	NMI	0x08
1		Reset	0x04
		Initial SP value	0x00

2.2.5 PHY Application on Cortex-M0

The algorithm the cortex working with is as follows:

- First, the cortex is reset. Then the PC is loaded with address 0x00000000
- The processor reads the value from 0x00000000 location to MSP.
- Then the processor reads the address of the reset handler from location 0x00000004
- Then it jumps to reset handler and start executing the instructions
- The main application (as illustrated in table 7) is included in the reset handler.
- The interrupt TTI_INT is enabled using the NVIC_EnableIRQ() function. So that when it arrives, it is served by the processor then it returns to the main application again.

The flow chart in figure (7), implements the firmware of the PHY which decodes the MIB and checks its correctness then transmits the MIB and its succeeded ISSB through UART IP to a monitor to print them.

Table 7: PHY Program Steps

1			Fill MMSE coefficient memories.
2			Transmit PHY subsystems parameters
3			- Assert Rx_Start_RIF signal. (Start FFT subsystem operation) - De-assert Rx_Stop_RIF signal.
4			If FFT_done is asserted: - De-assert Rx_Start_RIF signal. - Assert Rx_Stop_RIF signal.
	SW		Assert the trial_start_rif signal so that the blind decoding starts (Start Post-FFT subsystem operation).
	HW		De-assert trial_start_rif.
5	SW		If trial_done is asserted: - De-assert trial_start_rif. - check CRC_result value.
	HW		If All_done is asserted, check CRC_result value.
6			If CRC_result equals 1: The correct iSSB was found and the MIB was decoded successfully. Report the correct iSSB and MIB payload to the processor, then go to step 2.
7			If CRC_result equals 0
	SW	iSSB < 3	Transmit new iSSB value. Assert trial_start_rif Go to step 5
		iSSB = 3	Failed to decode MIB. Go to step 2
	HW		Go to step 2

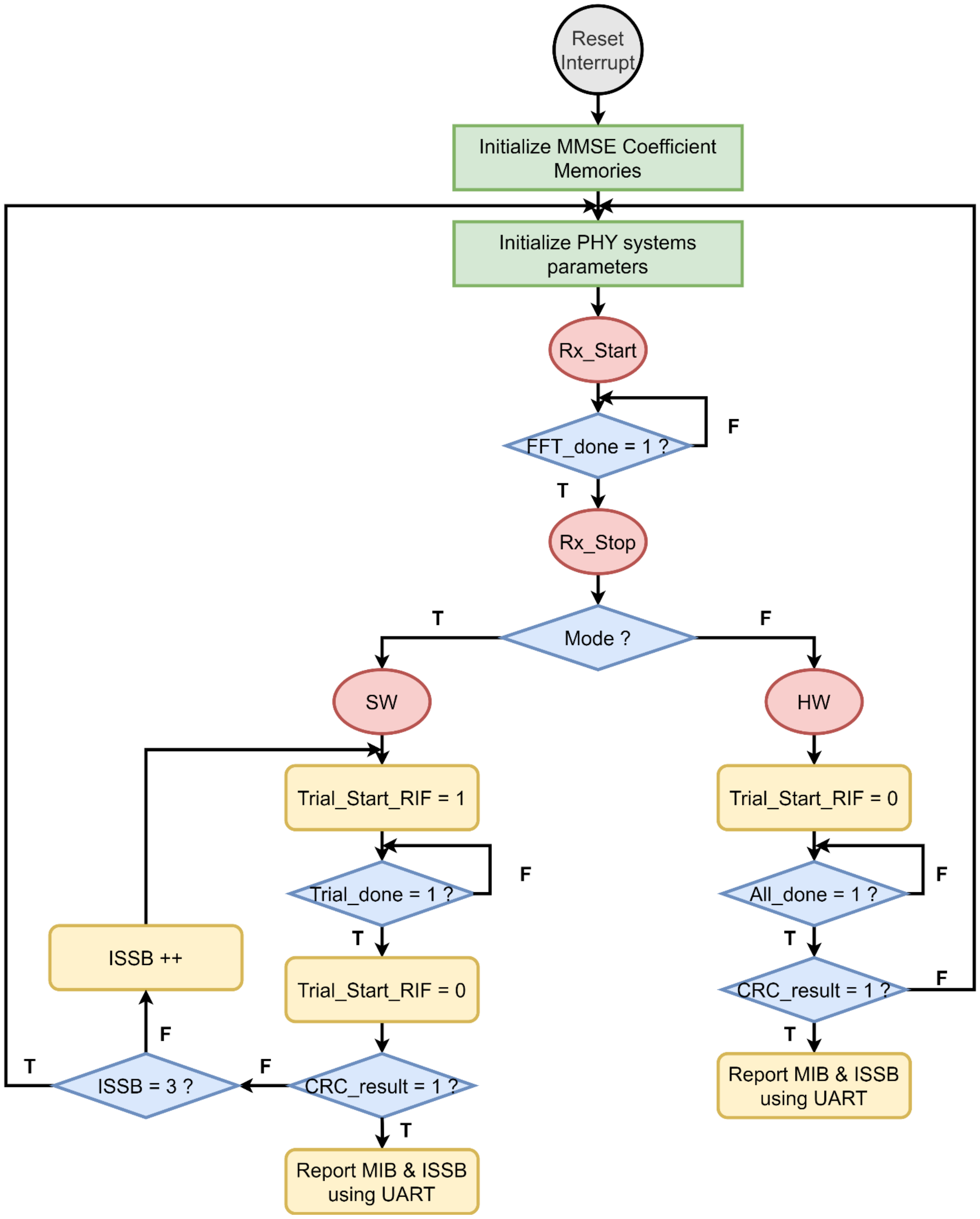
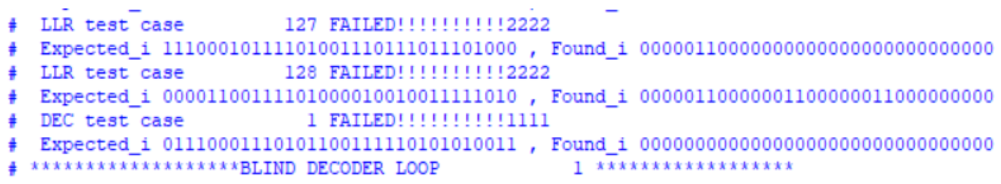


Figure 7: Flow Chart of PHY Program

2.2.6 Results

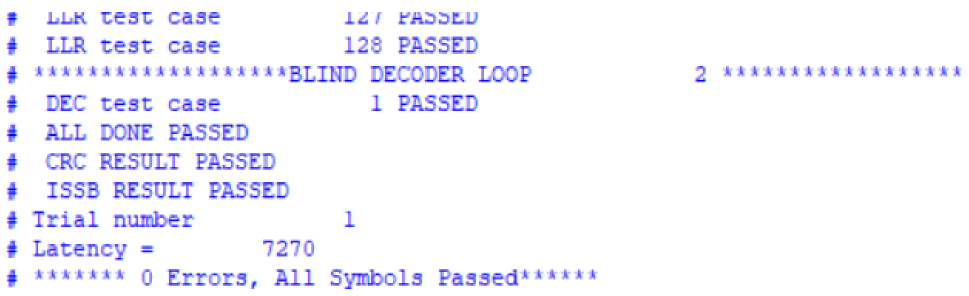
The PHY application is implemented in C language. We have loaded the code in the instruction memory, sent the required parameters to the PHY and started simulating the system on ModelSim, and we have tested output then compared it with the data from the reference model (which is implemented on MATLAB). In the chosen test case, the iSSB is 1. So, we have the following results:

- When the iSSB is 0, the iteration fails as shown in figure (9).
- When the iSSB is 1, the iteration is passed as shown in figure (8), since this is the value of iSSB the processor sent.



```
# LLR test case        127 FAILED!!!!!!!!!!2222
# Expected_i 11100010111101001110111011101000 , Found_i 00000110000000000000000000000000
# LLR test case        128 FAILED!!!!!!!!!!2222
# Expected_i 00001100111101000010010011111010 , Found_i 00000110000001100000011000000000
# DEC test case          1 FAILED!!!!!!!!!!!1111
# Expected_i 01110001110101100111110101010011 , Found_i 00000000000000000000000000000000
# ******************BLIND DECODER LOOP           1 *****************
```

Figure 9: Failed iteration when ISSB is zero



```
#  LLR test case         127 PASSED
#  LLR test case         128 PASSED
# ******************BLIND DECODER LOOP           2 *****************
#  DEC test case           1 PASSED
#  ALL DONE PASSED
#  CRC RESULT PASSED
#  ISSB RESULT PASSED
# Trial number           1
# Latency =        7270
# ******* 0 Errors, All Symbols Passed******
```

Figure 8: The passed iteration when ISSB is 1

Hint: All inputs of the cortex must take a value (you must not leave an input floating). For the outputs, connect what you need and leave the rest floating.

2.3 AHB Bus Matrix and Slaves

The AHB bus is a widely used bus protocol in the ARM cortex-M architecture. It connects the various components with the system-on-chip (SoC) design, enabling data transfer while maintaining ease of use. Each slave is set a certain address range in the cortex memory map.

The memory map for the system was set according to the specified ranges outlined in the Cortex design manual. We generated the bus matrix using XML file where we specified the address range for each slave as indicated in table (8).

Table 8: Memory Map set for cortex-M0.

Slave number	Slave Name		Start address	End address	Size
0	Instruction Memory		0x0000_0000	0x000F_FFFF	1 M
	Reserved		0x0010_0000	0x1FFF_FFFF	511 M
1	Data Memory		0x2000_0000	0x200F_FFFF	1 M
	Reserved		0x2010_0000	0x3FFF_FFFF	511 M
2	Bridge	UART	0x4000_0000	0x4000_0FFF	4 K

		Watchdog	0x4000_1000	0x4000_1FFF	4 K
		Timer	0x4000_2000	0x4000_2FFF	4 K
	Reserved		0x4000_3000	0x4000_FFFF	52 K
3	GPIO		0x4001_0000	0x4001_0FFF	4 K
4	PHY		0x4001_1000	0x4001_5FFF	20 K

Whenever the processor needs to access a certain AHB slave, it writes its specified address from the memory map.

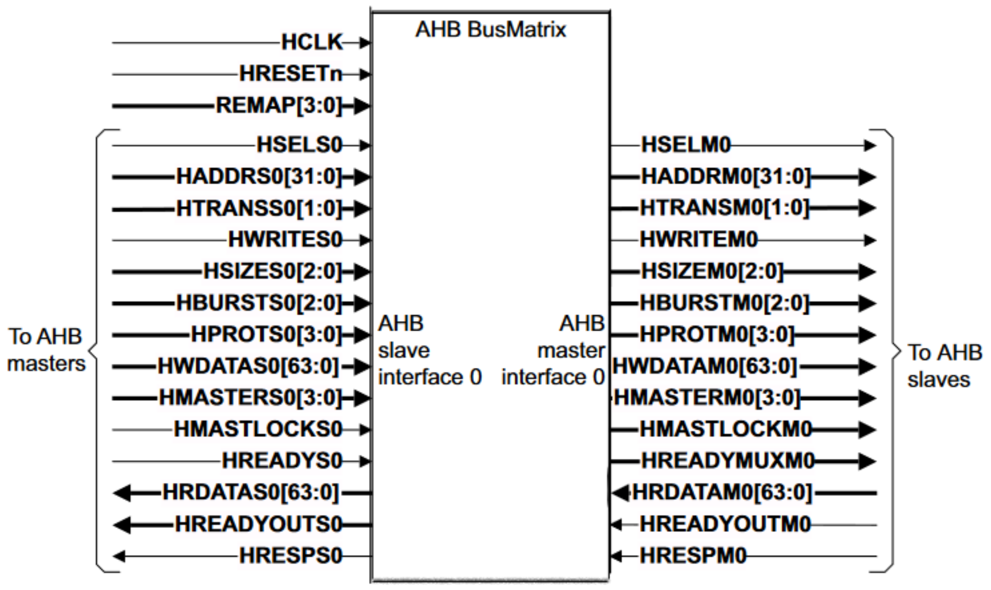


Figure 10: Part of the AHB Bus matrix block diagram

A section of the block diagram of the bus matrix is presented in figure (10), where the input and output signals are shown, and the description of the main signals is shown in table (9).

Table 9: Signal Description of some of the important AHB signals

Signal	Description
HCLK	System clock, Logic is triggered on the rising edge of the clock.
HRESETn	Activate LOW asynchronous reset.
HADDR	Address from AHB
HSEL	When enabled means a specific slave is selected
HSIZE	Indicate the size of transfer either word or half word or Byte
HWRITE	When enabled indicates a write transfer otherwise a read transfer occurs
HWDATA/ HRDATA	Data transferred from/to bus matrix
HTRANS	Indicates transfer type (IDLE, BUSY, NONSEQ, SEQ)

2.3.1 Instruction and Data Memories

The memories are connected to the AHB Bus matrix through a special interface AHB to SRAM interface. This interface translates the incoming AHB signals into signals understood by the SRAM module, figure (11).

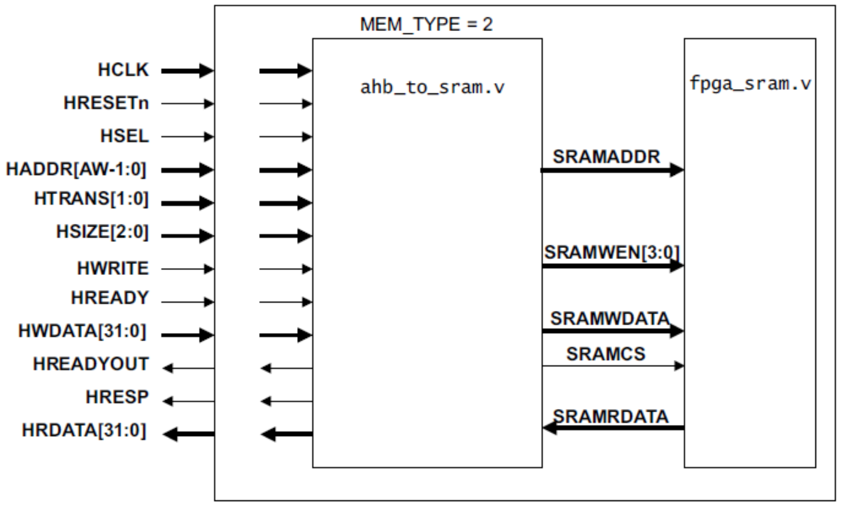


Figure 11: Memories Block diagram showing the interface with the bus matrix

- Instruction Memory: contains the instruction needed for the system to function.
- Data Memory: contains the data needed for each subsystem in the PHY to operate correctly.

2.3.2 GPIO

GPIO is an essential component in any SoC integration. It is a general purpose I/O interface unit of 16 bits with some properties such as programmable interrupts and alternate functions.

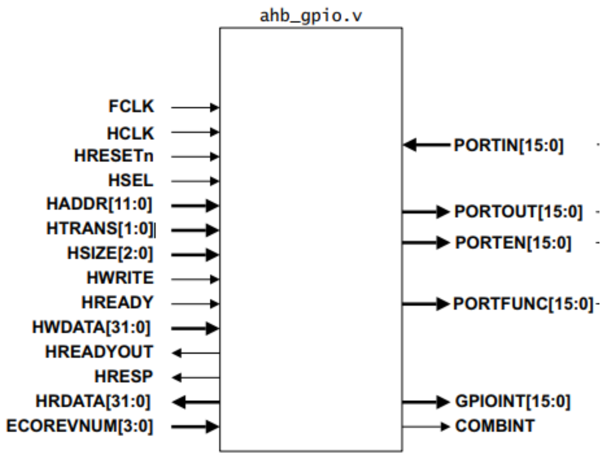


Figure 12: GPIO interface

Interrupt generation feature can be programmed through three registers which are interrupt enable, interrupt polarity and interrupt type, each register has separate set and clear addresses. Each bit of the I/O pins can be configured through these three registers. Interrupt polarity can be set to high or low while interrupt type can be set to level or edge triggered. When an interrupt is triggered, its corresponding bit in

INSTATUS register and GPIOINT are asserted. To de-assert these two bits and clear the interrupts, one has to be written inside INTCLEAR register. During interrupt generation, three cycle latency is introduced, two or input synchronization and another cycle for registering the interrupt status.

Each pin in the GPIO can be used as an I/O pin or an alternate function such as timer or UART or any other supported feature, this is done throughout a multiplexing network for each bit as shown in Figure (13). This alternate function feature is enabled by default for all GPIO pins and can be disabled by writing one inside the alternative function clear register.

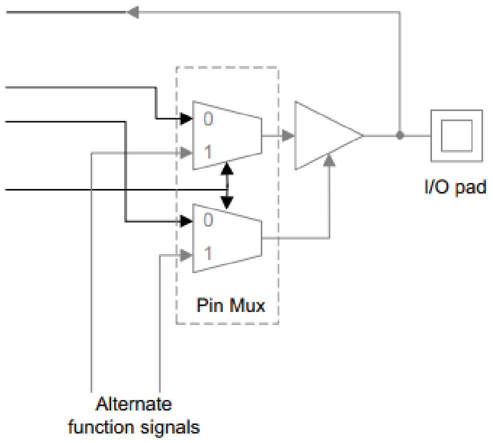


Figure 13: GPIO Alt. function

Masked access is another feature which allows reading from or writing to individual bits or multiple bits, this avoids read-modify-write operations which are not thread safe.

The GPIO slave was synthesized for FPGA and it was found that its frequency upper limit is 602 MHZ.

2.3.3 PHY

It contains the main part of our project, which is the physical (PHY) layer implementation. It consists of the following blocks, figure (14):

- 3 main building Subsystems: FFT subsystem, post-FFT subsystem and the decoder subsystem.
- Controller: it implements the blind decoding algorithm (discussed in chapter 5).
- Register interface: an interface to the AHB bus matrix that connects the PHY to the rest of the system (discussed in chapter 5).
- Memories.

We will go into further details into the implementation of the PHY in the following chapters.

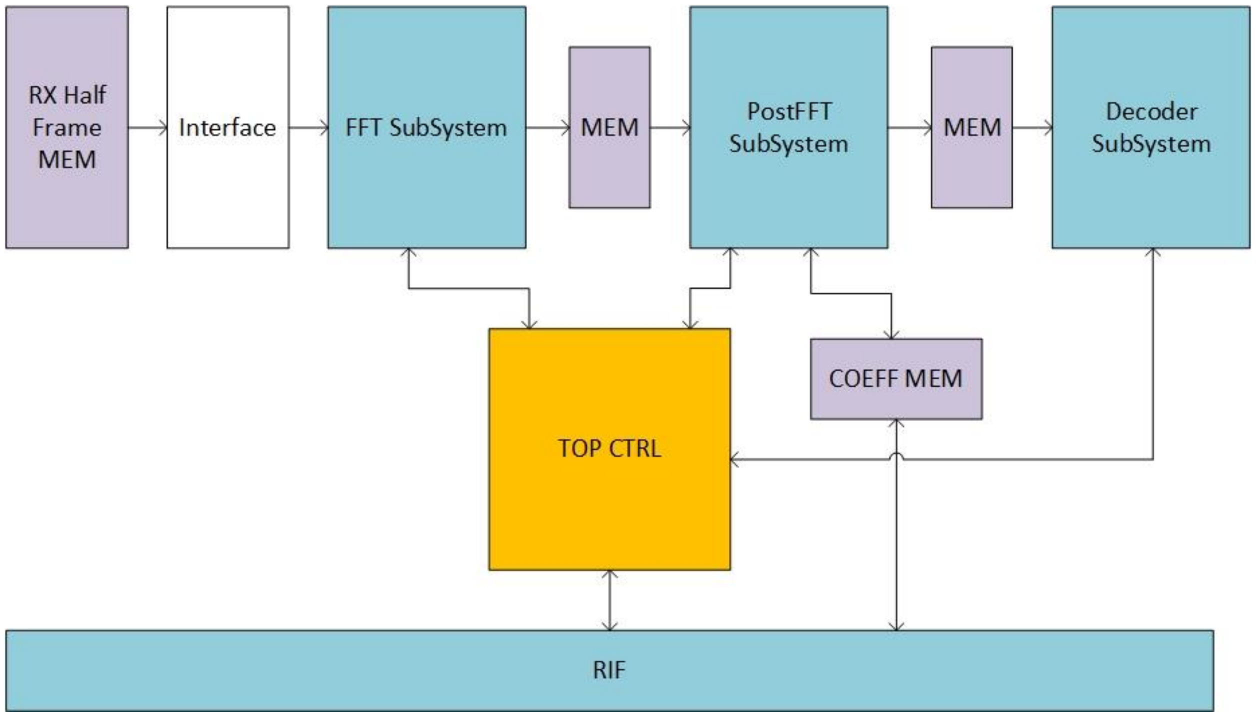


Figure 14: PHY Block Diagram

2.4 APB Subsystem

The Advanced High-performance Bus (AHB) to Advanced Peripheral Bus (APB) bridge is used in system-on-chip (SoC) designs to connect the AHB bus, which is typically a high-performance bus, to the APB bus, which is typically a lower-performance bus. Here are a few reasons why an AHB to APB bridge is used:

- Bus Compatibility: In a complex SoC design, different modules or peripherals may have different bus interfaces. The AHB bus is commonly used as the main interconnect for high-performance components such as GPIO and SRAMs, while the APB bus is used for lower-performance peripherals such as Timers and watchdogs. By using an AHB to APB bridge, it allows these different bus interfaces to communicate with each other seamlessly.
- Performance Optimization: The AHB bus is designed to provide high-performance data transfers between different modules or peripherals within the SoC. On the other hand, the APB bus operates at a lower clock frequency and is more suited for connecting slower peripherals that don't require high bandwidth. The AHB to APB bridge allows for the efficient transfer of data between the high-performance AHB bus and the lower-performance APB bus, optimizing the overall system performance.
- Performance Optimization: The AHB bus is designed to provide high-performance data transfers between different modules or peripherals within the SoC. On the other hand, the APB bus operates at a lower clock frequency and is more suited for connecting slower peripherals that don't require high bandwidth. The AHB to APB bridge allows for the efficient transfer of data between the high-performance AHB bus and the lower-performance APB bus, optimizing the overall system performance.

- Power Management: The AHB bus consumes more power compared to the APB bus due to its higher clock frequency and increased bandwidth. By using an AHB to APB bridge, it is possible to selectively enable or disable specific peripherals or modules connected to the APB bus, thus providing power management capabilities. This allows the system to conserve power by only activating the necessary peripherals when needed.
- System Integration: SoCs often consist of multiple IP (intellectual property) blocks or subsystems that may have different bus protocols. The AHB to APB bridge acts as a protocol converter, enabling seamless integration of these IP blocks into the overall SoC design. It provides a standardized interface for communication between different subsystems, regardless of their individual bus protocols.

Overall, the AHB to APB bridge plays a crucial role in enabling communication, optimizing performance, facilitating power management, and integrating different subsystems within a system-on-chip design. The block diagram of the APB bridge is presented in figure (15) where table explains some of the important output signals.

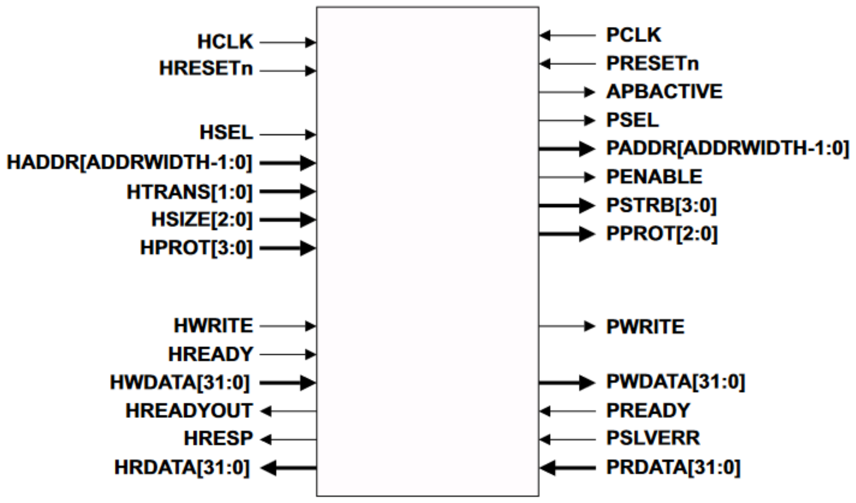


Figure 15: AHB to APB bridge block diagram

Table 10: Some of the important signals coming from the APB Bridge

Signal	Description
PCLK	System clock, Logic is triggered on the rising edge of the clock.
PRESETn	Activate LOW asynchronous reset.
PADDR	LSB of AHB address [15:0]
PSEL	When enabled means a specific slave is selected
PWRITE	When enabled indicates a write transfer otherwise a read transfer occurs
PWDATA/ PRDATA	Data transferred from/to bridge

2.5 APB Slaves

2.5.1 Timer

The APB timer is a 32 bit down-counter which generates an interrupt request signal, TIMERINT, when the counter reaches 0. The interrupt request is held until it is cleared by writing to the INTCLEAR Register. If the APB timer count reaches 0, and at the same time, the software clears a previous interrupt status, then the interrupt status is set to 1.

The timer peripheral contains a separate clock pin PCLKG for the APB register read or write logic that permits the clock to peripheral register logic to stop when there is no APB activity. You can turn-off the gated peripheral bus clock for register access PCLKG when there is no APB access which has same frequency and synchronous PCLK.

The timer can use external input signal EXTIN as a timer enable through zero to one transition of this signal.

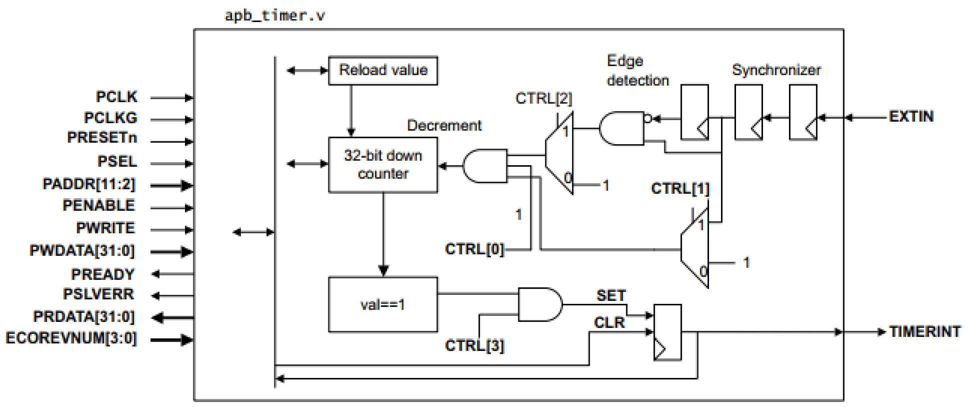


Figure 16: APB Timer

Access Timer Peripheral

1. To enable the timer peripheral interrupt, you should access the timer **CTRL** register through the following steps:
   - Set PADDR to address of **CTRL** register = 0x000.
   - Set PDATA = 32'd9 to set timer interrupt enable and global enable of module.
   - Set PSEL = 1 and PWRITE = 1.
2. To reload the counter of the timer with a given value you should access the timer **RELOAD** register through the following steps:
   - Set PADDR to address of **RELOAD** register = 0x008.
   - Set PDATA to the number you want.
   - Set PSEL = 1 and PWRITE = 1.
3. To clear the timer interrupt you should access the timer **INTCLEAR** register and set this register through the following steps:
   - Set PADDR to address of **INTCLEAR** register = 0x00c.
   - Set PDATA = 1
   - Set PSEL = 1 and PWRITE = 1.

2.5.2 Watchdog timer

The Watchdog module peripheral is a 32-bit down counter that is initialized from the Reload Register. The counter decrements by one on each positive clock edge of **WDOGCLK** when the clock enables **WDOGCLKEN** is HIGH. When the counter reaches zero an interrupt is generated. On the next enabled **WDOGCLK** clock edge the counter is reloaded from the reload Register and the countdown sequence continues. If the interrupt is not cleared by the time that the counter next reaches zero, then the Watchdog module asserts the reset signal **WDOGRES,** and the counter is stopped. This signal causes the system to be rested.

WDOGCLK can be equal to or be a sub-multiple of the **PCLK** frequency. However, the positive edges of **WDOGCLK** and **PCLK** must be synchronous and balanced.

The Watchdog module interrupt and reset generation can be enabled or disabled through the Control Register **WdogControl**. When the interrupt generation is disabled then the counter is stopped. When the interrupt is re-enabled then the counter starts from the value programmed in **WdogLoad** and not from the last count value.

The Watchdog counter only decrements on a rising edge of **WDOGCLK** when **WDOGCLKEN** is HIGH. The relationship between **WDOGCLK** and **PCLK** must observe the following constraints:

- The rising edges of **WDOGCLK** must be synchronous and balanced with a rising edge of **PCLK.**
- The **WDOGCLK** frequency cannot be greater than the **PCLK** Frequency.

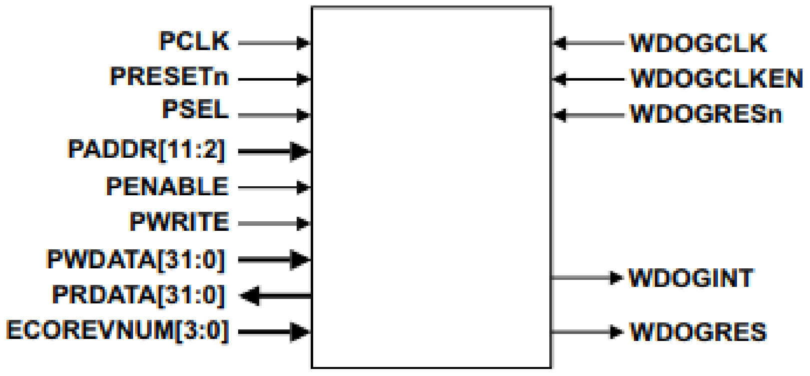
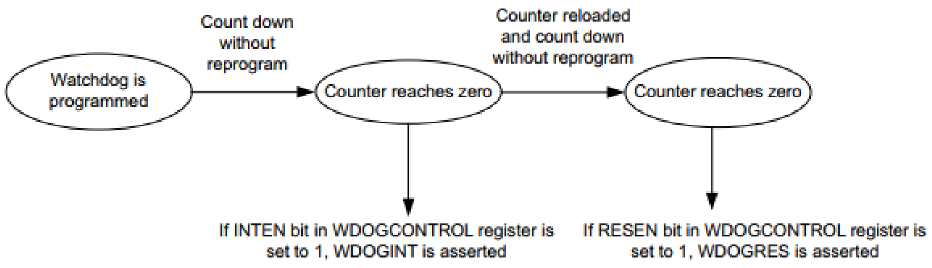


Figure 17: ABP Watchdog

<u>Access Timer Peripheral</u>

1. Enable ABP to access WDT registers by unlocking its registers through accessing **Lock WDT** register. Writing a value of 0x1ACCE551 to the register enables write accesses to all the other registers. Writing any other value disables the write accesses to all registers except the Lock Register. To access this register
   - Set PADDR to address of **WDOGLOCK** register = 0xC00.
   - Set PDATA = 0x1ACCE551.
   - Set PSEL = 1 and PWRITE = 1.
2. Enable **INTEN** and **RESEN** bits in **WDOGCONTROL** control register to enable **WDOGINT** and **WDOGRES** signals through the following procedures:
   - Set PADDR to address of **WDOGCONTROL** register = 0x008.
   - Set PDATA = 32’d2.
   - Set PSEL = 1 and PWRITE = 1.
3. To Load Watchdog with a value you should access the timer **WDOGLOAD** register and set this register with the value you want to load through the following steps:
   - Set PADDR to address of **WDOGLOAD** register = 0x000.
   - Set PDATA to the number you want.
   - Set PSEL = 1 and PWRITE = 1.

4. To clear interrupt in Watchdog peripheral, you should access **WDOGINTCLR** register and write in it any number through the following steps:
   - Set PADDR to address of **WDOGINTCLR** register = 0x00C.
   - Set PDATA to the number you want.
   - Set PSEL = 1 and PWRITE = 1.



Watchdog is programmed
Count down without reprogram
Counter reaches zero
Counter reloaded and count down without reprogram
Counter reaches zero
If INTEN bit in WDOGCONTROL register is set to 1, WDOGINT is asserted
If RESEN bit in WDOGCONTROL register is set to 1, WDOGRES is asserted

Figure 18: Watchdog timer flow diagram

2.5.3 APB UART.

The design of the APB UART supports 8 bits communications without parity, and it supports a one bit start and one bit stop of the transmitting and receiving, which means that the total width of the character frame is 10 bits.

The design has a baud divider buffer to make the baud rate configurable to make the design suitable for most simple embedded applications, we can calculate the baud rate using the baud divider value which stored in the baud divider register according to the following equation.

$$BaudRate = \frac{Clock\ freq\ of\ the\ system}{Baud\ Divider\ value} \tag{2.1}$$

The baud divider value represents approximately the number of cycles at which one bit can be transmitted or received.

The baud rate is used to calculate the number of clock cycles at which one character can be transmitted or received, we can calculate the number of the clock cycles at which a character can be transmitted or received according to the following equation.

$$num\ of\ clock\ cycles = \frac{clock\ freq\ of\ the\ system\ \times total\ width\ of\ the\ frame}{BaudRate} \tag{2.2}$$

The UART at the transmitting mode stores the data comes from the APB interface in a buffer called write buffer, then the write buffer passes the data to the transmitter shift register to convert the parallel bus of data to a serial stream of data to be transmitted and asserts the TX interrupt flag. As shown in Figure 19.

A New character can be stored to the write buffer while the shift register is sending out a character, and when the write buffer is full the TX overrun interrupt flag is asserted.

The UART at the receiving mode the UART asserts the RX interrupt flag, then passes the serial stream of the received data through a bit synchronizer to synchronize the received data with the clock of the system, then the synchronizer passes it to the receiver shift register to convert the serial stream of data to a parallel bus of data, then the receiver shift register stores the received parallel data in a buffer called read buffer, then the data is forwarded to the APB interface. As shown in Figure 19

The shift register can receive the next character while the data in the read buffer is waiting for the APB interface to read it, and when the read buffer is full the RX overrun interrupt flag is asserted.

We have two configuration registers the control register which called CTRL and the baud divider register which called the BAUDDIV these registers can be configured by the processor according to the running application.

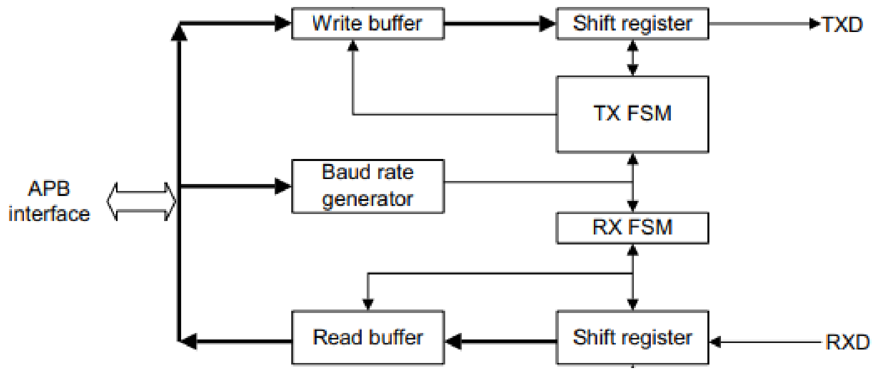


Figure 19: APB UART

The APB UART supports a high-speed test mode, which is useful for simulation during SoC or ASIC development. When CTRL [6] is set to 1, the serial data is transmitted at one bit per clock cycle. This enables you to send text messages in a much shorter simulation time. If required, you can remove this feature for silicon products to reduce the gate count. You can do this by removing bit 6 of the control register CTRL.

After doing synthesis to the design, we found that the maximum operating clock frequency of the system is 246 MHZ, and the signals PCLK and PCLKG must be equal as shown in the following figure.

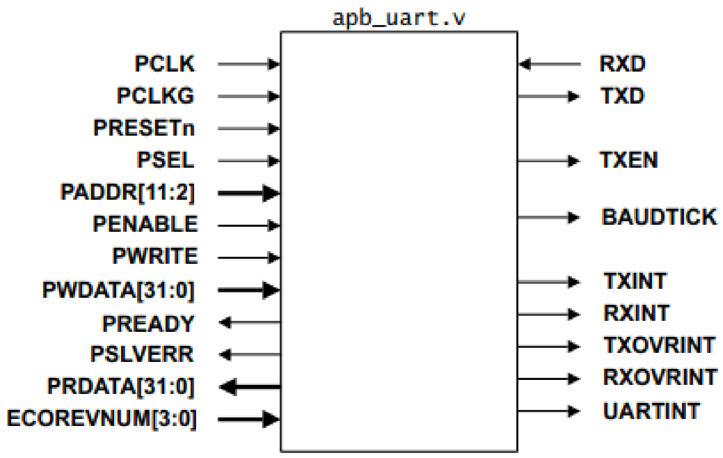


Figure 20: UART Block

Chapter 3: The Uncoded Model for PBCH Signal Processing

3.1. Transmitter Model for PBCH Signal Processing

The PBCH and DMRS signal Generation and Mapping at NR DL Transmitter is shown in Fig. 21.

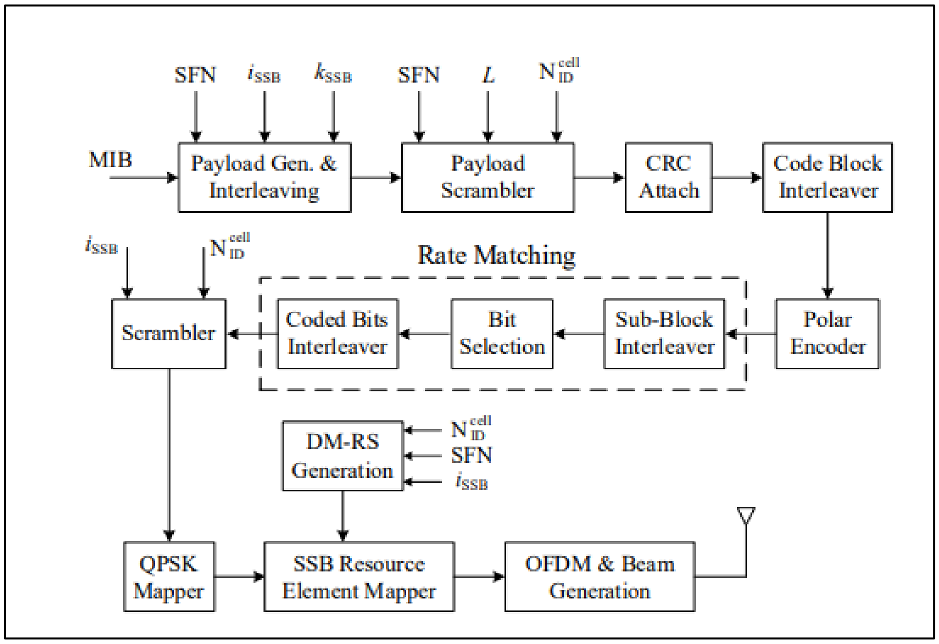


Figure 21: PBCH Processing and Mapping at NR DL Transmitter

3.1.1 Rate Matching

The encoded 512-bit code word (512-bit code word is the maximum word length supported by the NR polar encoder) is the input to the rate matching block. Rate matching is applied to the code word to obtain the number of bits that would fill the PBCH's available resource elements after the QPSK mapping. The rate matching consists of a subblock interleaver, bit selection, and a coded bits interleaver. The sub-block interleaver is a conventional block interleaver with 32-bit depth. For PBCH processing, the bit selection implements circular bit repetition so that the 512-bit code word is extended to 864 bits. The coded-bits interleaver is also a block interleaver, but it is a kind of square interleaver whose depth is a function of the number of bits to be interleaved.

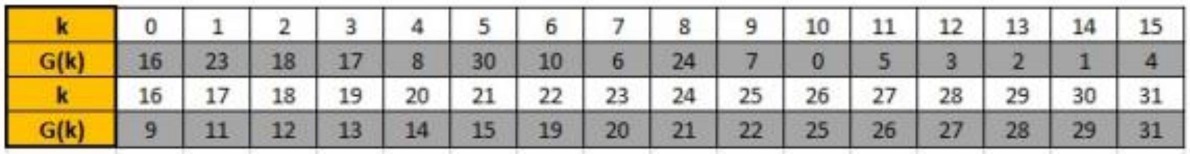
For sub-block interleaving, the bit sequence from channel coder output d_0, d_1, d_2,…,d_{N-1} (N=512) is divided into 32 sub-blocks as follows:
For n = 0 to N-1

$$i = \lfloor 32n/Nc \rfloor \qquad (3.1.1)$$

$$\mathrm{J}(n) = P(i)x(N/32) + mod(n, N/32) \qquad (3.1.2)$$

$$yn = d_{J(n)} \qquad (3.1.3)$$

Where the sub-block inter-leaver pattern P(i) is given in the following table.



k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
G(k)	16	23	18	17	8	30	10	6	24	7	0	5	3	2	1	4
k	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
G(k)	9	11	12	13	14	15	19	20	21	22	25	26	27	28	29	31

For bit-selection, the bit sequence y_0, y_1, ...y_{N-1} is loaded in a circular buffer of length N. If E is the rate to be matched (E=864), then the rate matching sequence e_1, e_2, e_3, ..., e_{E-1} is obtained as follows:

```
if E >= N —— repetition of bits by padding
   for k = 0 to E - 1
   ek = $y_{mod(k,N)}$
   end for
else
   if K/E <= 7/16 —— puncturing if the rate is lower than channel coding output
      for k = 0 to E -1
      $e_k = y_{k+N-E}$
      end for
   else —— shortening
      for k = 0 to E - 1
      $e_k = y_k$
      end for
   end if
end if
```

Finally the sequence e_1, e_2, e_3, ..., e_{E-1} is interleaved into bit sequence f_1, f_2, f_3, ..., f_{E-1} to get the final rate matched bits of PBCH.

3.1.2 Scrambler

Before modulating the final payload bit sequence f_1, f_2, f_3, ..., f_{E-1}, it is scrambled according to,

$$fi' = \big(fi + c(i + vMpn)\big)mod2 \qquad (3.1.4)$$

Where c(i) is computed as follows:

$i = 0,1,2,3.\dots.Mpn, \quad Mpn = 864$

$$c(i) = \big(x1(i + Nc) + x2(i + Nc)\big)mod2 \qquad (3.1.5)$$

$$x1(i + 31) = (x1(i + 3) + x1(i))mod2 \qquad (3.1.6)$$

$$x2(i + 31) = (x2(i + 3) + x2(i + 2) + x2(i + 1) + x2(i))mod2 \qquad (3.1.7)$$

Where $Nc = 1600$,

$x1(0) = 1, x1(i) = 0\, for\ all\ other\ i,$

$cinit = \sum_{i=0}^{i=31} x2(i). 2^i$,

And is initialized with $cinit = N_{ID}^{cell}$ at the start of each SSB.

Where:

- For L=4, v is the last 2 LSBs of the SSB index.

- For L=8 or 64, v is the 3 LSBs of the SSB index.

3.1.3 QPSK Mapping

The scrambled bit sequence f̃1, f̃2, f̃3, ..., f̃E-1 is modulated by QPSK scheme into a block of complex-valued QPSK symbols carrying PBCH data, $d_{PBCH}(0)$, $d_{PBCH}(1)$, $d_{PBCH}(2)$….., $d_{PBCH}(M_{symbol}$ - 1). Here M_{symbol} = 864/2 = 432.

3.1.4 DMRS Signal Generation

The demodulation reference signal (DMRS) is used to decode the PBCH signal at the receiver by helping in estimating the channel and noise values. DMRS symbols are QPSK modulated and generated from a PN sequence generator, as discussed in section 3.1.2, with $C\text{-}init$. initialized as follows:

$$C_{init} = 2^{11}(\bar{i}_{SSB})(\lfloor N_{ID}^{cell}/4 \rfloor + 1) + 2^{6}(\bar{i}_{SSB} + 1) + (N_{ID}^{cell}\, mod4) \qquad (3.1.8)$$

$$fi' = \left(fi + c\left(i + vM_{pn}\right)\right) mod2 \qquad (3.1.9)$$

Where c(i) is computed as follows:

$i = 0, 1, 2, 3.\ldots.M_{pn}, \quad M_{pn} = 144$

$$c(i) = \left(x1(i + N_c) + x2(i + N_c)\right) mod2 \qquad (3.1.10)$$

$$x1(i + 31) = (x1(i + 3) + x1(i)) mod2 \qquad (3.1.11)$$

$$x2(i + 31) = (x2(i + 3) + x2(i + 2) + x2(i + 1) + x2(i)) mod2 \qquad (3.1.12)$$

Where $N_c = 1600$,
$x1(0) = 1, x1(i) = 0\, for\ all\ other\ i,$
$C_{init} = \sum_{i=0}^{i=31} x2(i).2^{i}$

Where:

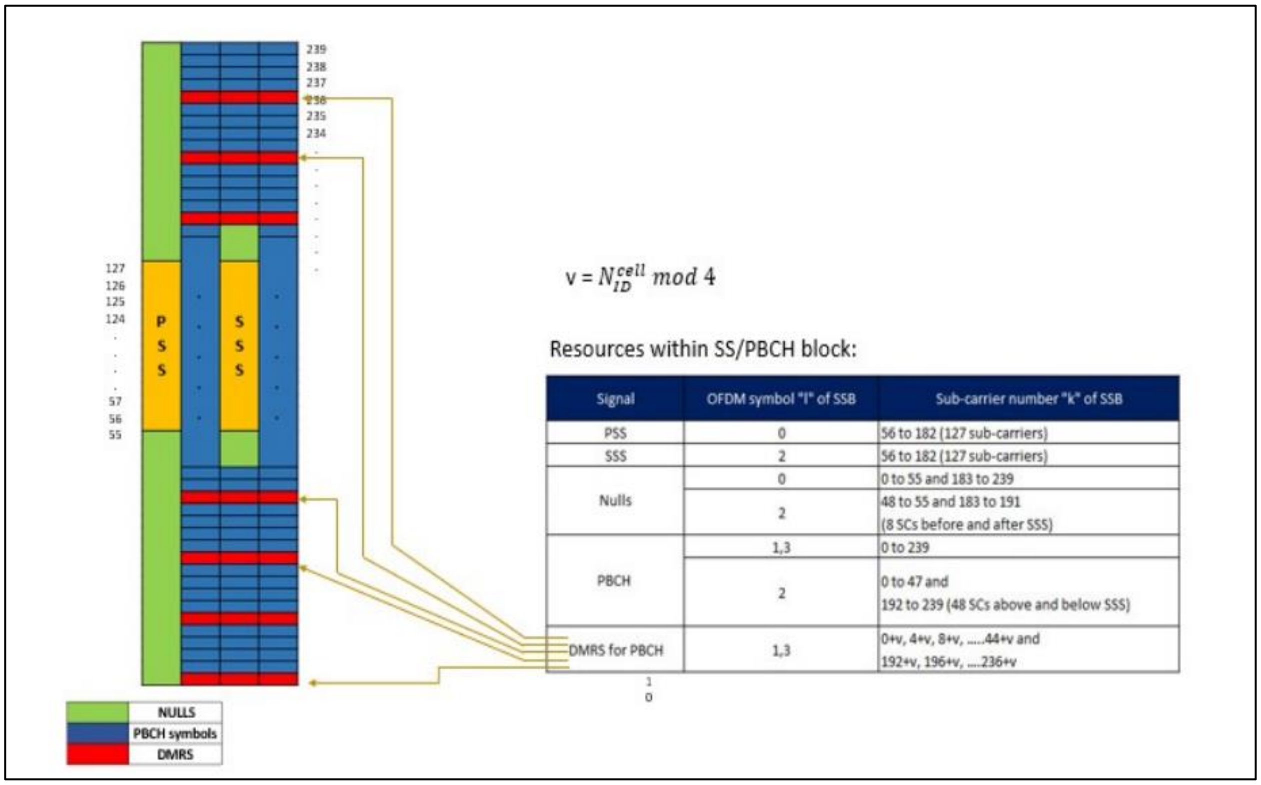
- For L=4, $\bar{i}_{SSB} = i_{SSB} + 4N_{hf}$
- For L=8 or 64 , $\bar{i}_{SSB} = i_{SSB}$

and N_{hf} is the half-frame number in which the PBCH is present.

3.1.5 SSB Resource Element Mapping

These 432 QPSK-modulated symbols, along with 144 DMRS symbols, are mapped to the 576 sub-carriers, which are the resource elements on the available bandwidth part for PBCH transmission.

An SS/PBCH block is composed of 4 OFDM symbols in the time domain, numbered from 0 to 3, in which PSS, SSS, and PBCH, along with the DMRS symbols, are mapped on the resource elements, viz. sub-carriers. An SSB consists of 240 contiguous sub-carriers numbered from 0 to 239, as shown in Fig. 22. We have 432 symbols of PBCH and 144 symbols of DMRS to be mapped in the available bandwidth part for SSB. The resource mapping per OFDM symbol is given in the following table, where k and i are frequency and time index, respectively.



$v = N_{ID}^{cell} \ mod \ 4$

Resources within SS/PBCH block:

Signal	OFDM symbol "l" of SSB	Sub-carrier number "k" of SSB
PSS	0	56 to 182 (127 sub-carriers)
SSS	2	56 to 182 (127 sub-carriers)
Nulls	0	0 to 55 and 183 to 239
	2	48 to 55 and 183 to 191 (8 SCs before and after SSS)
PBCH	1,3	0 to 239
	2	0 to 47 and 192 to 239 (48 SCs above and below SSS)
DMRS for PBCH	1,3	0+v, 4+v, 8+v, ...44+v and 192+v, 196+v, ...236+v

Figure 22: Resources within SSB

Location shift of DMRS by N_{ID}^{cell} value,

The position of DMRS in SSB shifts vertically according to the value of N_{ID}^{cell} as shown below:

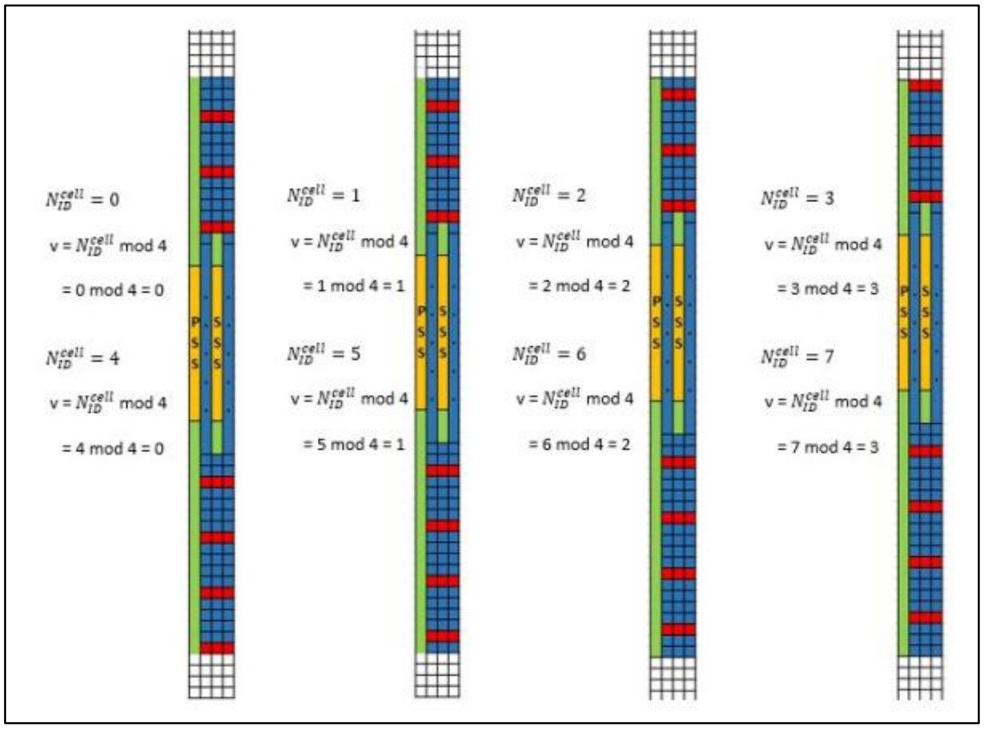


Figure 23: Shift in DMRS due to N_{ID}^{cell}

The UE assumes the sequence of symbols dPSS(0), dPSS(1),..., dPSS(126) constituting the PSS and dSSS(0), dSSS(1),..., dSSS(126) constituting the SSS are mapped to resources (k, l) in increasing order of k, where k and l are frequency and time indices, respectively.

The UE assumes the sequence of complex-valued symbols dPBCH(0), dPBCH(1),..., and dPBCH(431) carrying the PBCH data are mapped to resources (k, l), excluding the positions for DMRS mentioned in the table in Fig. 22. Similarly, the symbols for DMRS are mapped to the given resource locations.

3.1.6 Frame generation and IFFT

The L number of SS/PBCH blocks thus obtained are allocated in a grid of subcarriers for every OFDM symbol. Here the total sub-carrier per OFDM symbol is 240, so the L number of 240x4 grid blocks of SS/PBCH data is mapped in frames with 240 sub-carriers per OFDM symbol. Then zero padding is applied to make the total number of subcarriers per OFDM symbol = 256. Inverse Fast Fourier Transform (256 IFFT) is applied to each symbol in the frame to convert them to the time domain. After that, the cyclic prefix is added to each symbol, and the CP can be 18 or 20 samples. The CP is 18, and for every 7 OFDM symbols, the CP is 20. For example, at SCS of 15 kHz, we have 10 subframes in the frame, and each subframe consists of 14 OFDM symbols. We then have 140 OFDM symbols in the frame, so 120 OFDM symbols have a CP of 18 and 20 OFDM symbols have a CP of 20. To determine the sampling rate in this case, $\frac{(256+18)*120+(256+20)*20}{time\ of\ frame(10mse)} = 3.84\ MHz.$

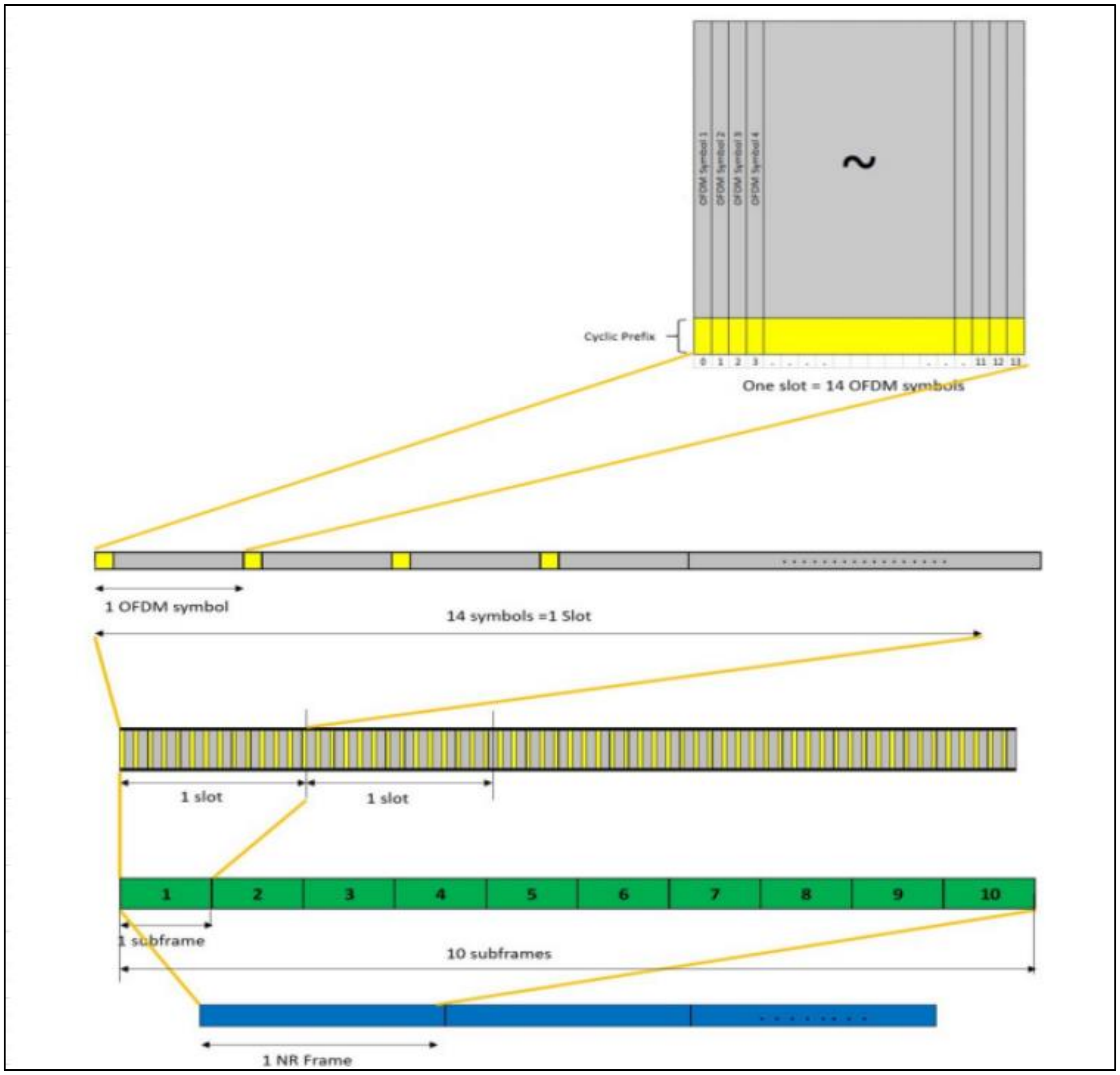


Figure 24: Frame Generation using OFDM

3.1.7 Transmission of the Frame

The PBCH baseband equivalent system model for the air interface includes transmitted signal x(t), wireless channel h(t-t0), AWGN and received PBCH signal y(t) depicted as follows:

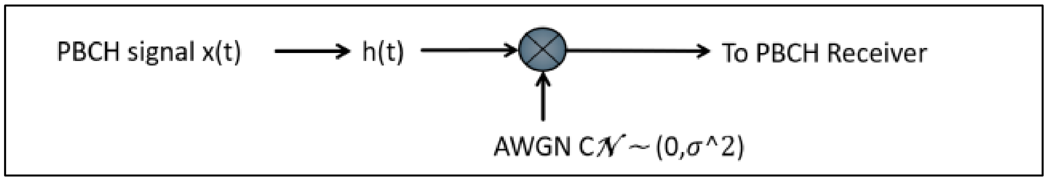


Figure 25: PBCH communication model in Baseband

The PBCH signal is the NR frame generated in the time domain, as shown above in Fig.25. Now the signal experiences attenuation, time delays, and phase delays in multiple paths, along with some random environmental noise added to it, before reaching the receiver antenna. The signal that simulates the complete attenuation and delay in multipath is the wireless time varying channel h(t-t0). The random environmental noise is characterized as Additive White Gaussian Noise. The wireless channel gets convoluted with the PBCH signal, and AWGN being additive is added to obtain the signal at receiver as y(t). y(t) is a delay and attenuated version of x(t) affected in both time and frequency.

3.2. Receiver Model for PBCH Signal Processing

The PBCH DL Receiver Process Flow is shown in Fig 26.

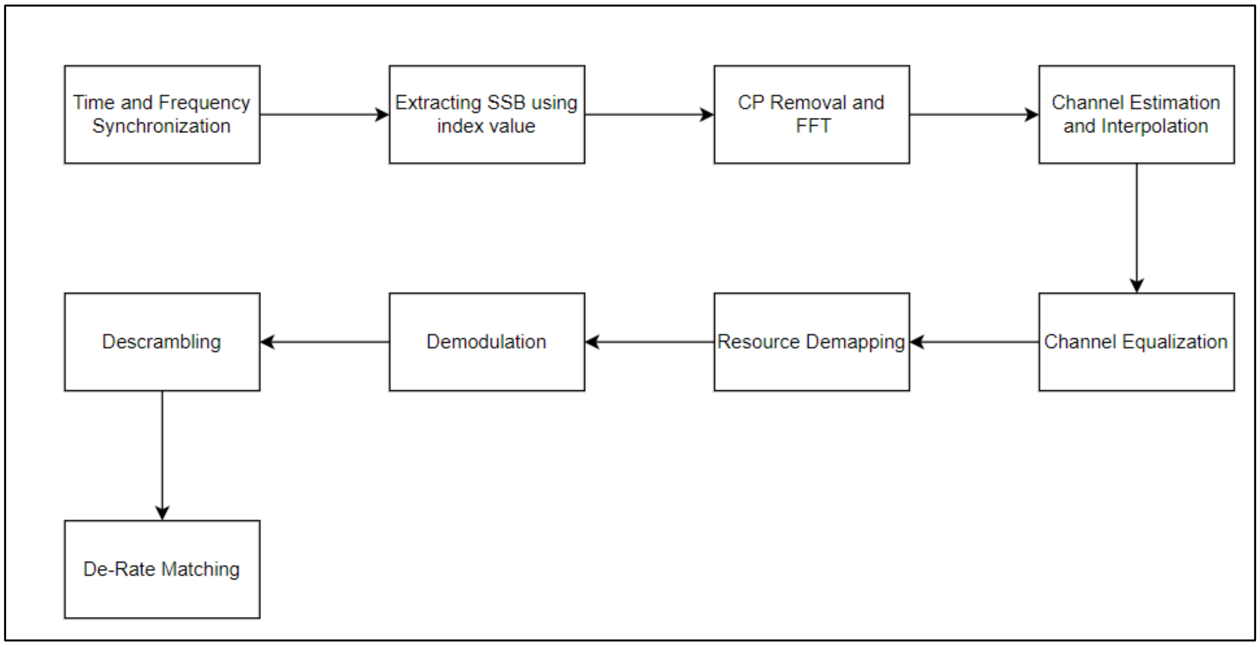

Time and Frequency Synchronization
Extracting SSB using index value
CP Removal and FFT
Channel Estimation and Interpolation
Channel Equalization
Resource Demapping
Demodulation
Descrambling
De-Rate Matching

Figure 26: PBCH Receiver Process Flow

3.2.1 Time and Frequency Synchronization

We assume perfect time and frequency synchronization. which means that the OFDM symbols indices of the SSB, fc, μ and N_{ID}^{cell} are known to the receiver.

3.2.2 Extracting SSB

As we assume that we know fc and μ, so we also know the index of each symbol in the SSB. We can extract the four OFDM symbols of the SSB from the frame. If suppose, μ =0 and fc = 2Ghz, case A is satisfied and the first index for SS/PBCH block is ”2”. This index marks the third OFDM symbol in the frame, as described in section 1.3. Fig 4.

3.2.3 CP Removal and FFT

Now we have the four symbols of the SSB, and we will remove the CP to ensure that each OFDM symbol consists of 256 subcarriers. Then a 256-point FFT is applied to each symbol of the SSB.

CP Removal

In OFDM systems, ISI occurs at the receiver due to the delay spread of the multipath channel. ISI stands for inter-symbol interference. In order to avoid ISI, a guard interval is inserted between two OFDM symbols. This guard interval is referred to as the cyclic prefix (CP). This is kept greater than the delay spread of the channel to avoid ISI.
OFDM systems require the orthogonality of carriers for correct demodulation at the receiver. When multipath channels are involved, the Orthogonality of carriers is lost, which can be restored by a cyclic prefix.
Cyclic prefix refers to the prefixing of a symbol with a repetition of the end. It is used to eliminate ISI by maintaining an integer number of cycles in the symbol duration. The receiver is typically configured to discard the cyclic prefix samples.
At the transmitter, cyclic prefix samples are copied from the end of the symbol to the beginning of the symbol. At the receiver, these samples are discarded from the symbol before going through the FFT subsystem. Cyclic prefix removal is implemented through a packet timer block that will be discussed in detail in the hardware design chapter.
The following figures show the addition and removal of cyclic prefixes to maintain an integer number of cycles in the symbol duration.

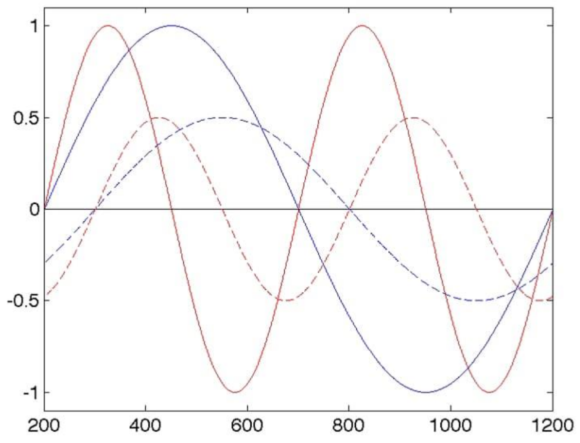


Figure 27: symbol after adding cyclic prefix

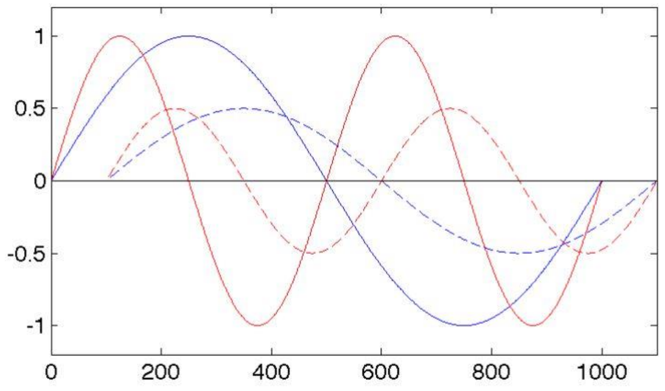


Figure 28: symbol without cyclic prefix

FFT modelling

Floating point model:

In 5G NR, the SSB (Synchronization signal block) is spread over 240 subcarriers, as shown in the figure.

However, in FFT, the selected number of input samples, or the block length BL, is always an integer power to the base 2.

Hence, a 256-point FFT (block length of 256) was chosen, and the extra subcarriers can be zero-padded [7].

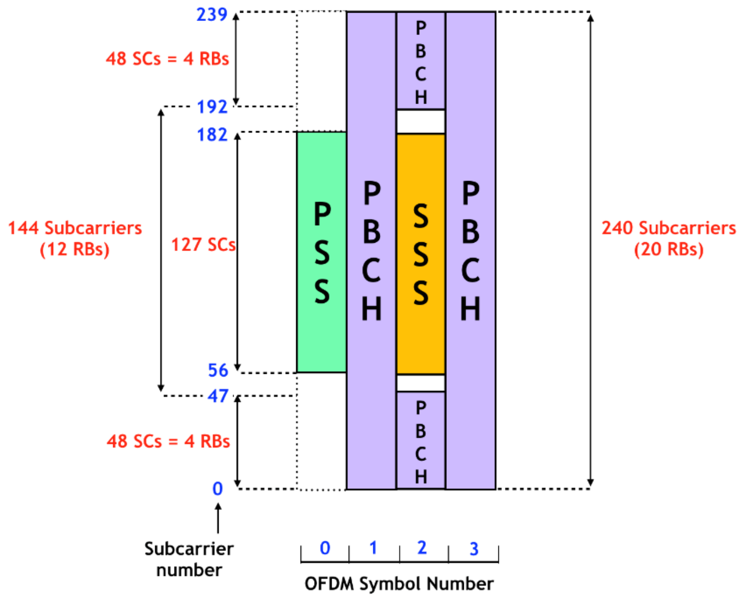


Figure 29: SSB block in NR

Radix 4 Butterfly takes four inputs, one from each quarter input of samples per group, as shown in the figure. For example, for the first butterfly operation in stage 1, the four inputs of the butterfly are samples x [1], x [65], x [129], and x [193]. After each butterfly operation, the outputs of the butterfly are multiplied by the twiddle factor exponentials. These twiddle factor exponentials follow different sequences in each stage, for example, in stage 1, the first output of each butterfly is multiplied by a twiddle factor of a power of zero. Second outputs are multiplied by the twiddle factors of power n, where n ranges from 0 to 63. Third outputs are multiplied by twiddle factors of power 2n, where n ranges from 0 to 63. Fourth outputs are multiplied by twiddle factors of power 3n, where n ranges from 0 to 63.

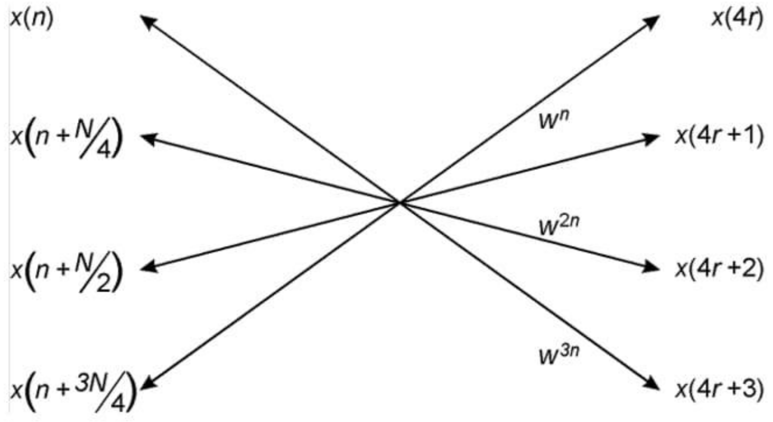


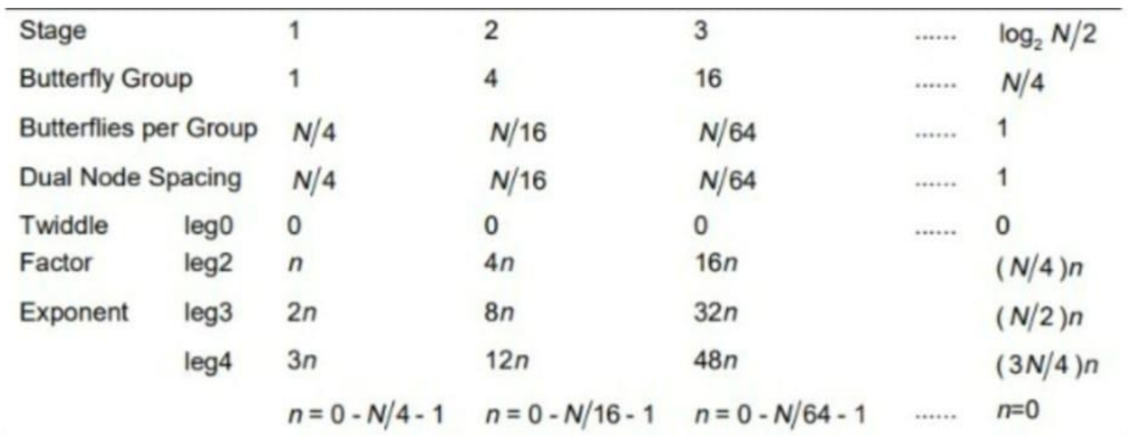
Figure 30: radix 4 butterfly

In the Matlab model, the radix-4 butterfly equations were modeled using the following matrix:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}$$

Where the four required input samples are multiplied by this matrix and then multiplied by twiddle factors depending on leg number and butterfly group. Outputs are then reordered to be ready for the next stage of the FFT.

Characteristics of the 256-point radix-4 FFT as butterfly groups per stage and twiddle factor exponentials sequence are shown in the next figure.



Stage		1	2	3	......	$\log_2 N/2$
Butterfly Group		1	4	16	......	$N/4$
Butterflies per Group		$N/4$	$N/16$	$N/64$	......	1
Dual Node Spacing		$N/4$	$N/16$	$N/64$	......	1
Twiddle	leg0	0	0	0	......	0
Factor	leg2	n	$4n$	$16n$		$(N/4)n$
Exponent	leg3	$2n$	$8n$	$32n$		$(N/2)n$
	leg4	$3n$	$12n$	$48n$		$(3N/4)n$
		$n = 0 - N/4 - 1$	$n = 0 - N/16 - 1$	$n = 0 - N/64 - 1$	......	$n=0$

Figure 31: characteristics of radix-4 FFT

Input signal power adjustments

As our input signal is an OFDM signal with a high PAPR, we modeled our complex input signal using the Randn() Matlab built-in function, which initially generates a random gaussian signal with power = 0 dB.

We assumed PAPR = 15 dB, and we scaled down the input power to -15 dBs, assuming an input power of -15 dBs adjusted by AGC. allowing a PAPR of like 15 dBs (0 to -30), as shown in the figure below.

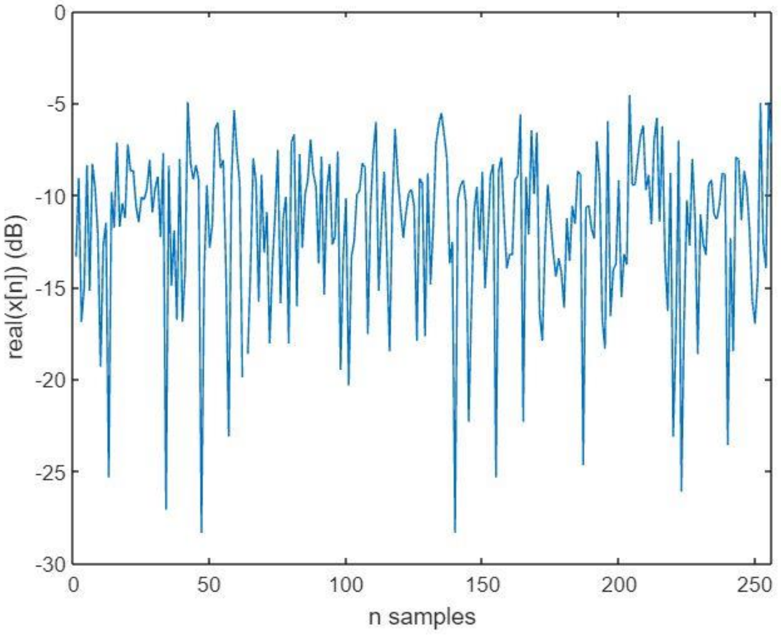


Figure 32: input signal PAPR

The AGC block at analog RF is responsible for input gain adjustment. We multiplied the input signal by 0.178 to scale the power to -15 dB to model the effect of AGC.

We assumed our input bit width is 12 bits, and as the dynamic range of 1 bit is equal to 6 dB, our input dynamic range is equal to 12*6 = 72 dB, so we have approximately 2 bits to cover the high values of the spikes in the input signal. The following figure shows The PAPR of the input signal.

To get the dynamic range of 1 bit:

Dynamic range = 20 log $[\frac{largest\ amp}{smallest\ amp}]$ = 20 log $[\frac{2^{n-1}}{2^{-1}}]$ = 20 log 2^n = 20*n* log(2)

Dynamic range = 6.02 * n

Where n = the number of bits

Fixed point model:

After designing a floating point 256 point Radix-4 FFT model and checking it using Matlab's FFT built-in function, A fixed-point model is used to quantize the floating-point numbers.

When designing a fixed-point model, there is a compromise between area and performance. The advantage of using a fixed-point number representation is that it reduces the growth of bits after each addition or multiplication operation to optimize hardware. However, excessive reduction deteriorates the performance as fewer bits are used to represent the value, leading to less accuracy, hence, our goal is to get a fixed point representation with the least possible number of bits and an acceptable performance.

We measure performance using SQNR, which is the ratio between floating point signals and the error between floating point and quantized signal [10].

For acceptable performance, the error should not exceed the value of thermal noise that already exists in our system. Hence, SQNR should be more than 40 dB in the worst case.

SQNR was calculated using the following formula:

$$\text{SQNR} = \frac{power\ of\ input}{power\ of\ error} = \frac{\Sigma_k |x_{floating}(k)|^2}{\Sigma_k (|x_{floating}(k) - x_{fixed}(k)|)^2}$$

Sign Bit
Integer
Fraction

Figure 33: fixed point representation

A fixed point representation is shown in the figure, a sign bit to represent 2's complement format, bits for the integer part, and bits for the fractional part. A Matlab built-in function (Fi(input,word length,fractional part)) was used for the fixed point, which quantizes using round-up and saturation operations.

Input bit width:

We assumed a 12-bit 2's complement format IQ interface for the input signal, and as the input power is scaled down to -15 dB, all the input values are within the range of -1: 1. Hence, no integer part is needed to represent the input therefore, the input signal's fixed point representation is one bit for sign and 11 bits for the fractional part (S11).

Twiddle factors bit width:

For twiddle factor exponentials, according to trials, we found that 9 bits is sufficient to represent twiddle factors with acceptable performance. All the twiddle factors are within the range of -1 to 1. Therefore, no integer part is needed to represent the twiddle factors, then the twiddle factors fixed point representation is one bit for sign and eight bits for fractional part (S8).

Output bit width selection:

Now we want to determine the minimum output bit width that achieves acceptable performance.

For our fixed point model to consider the worst case, we calculated the minimum SQNR of 100 trials and checked if it gave an acceptable performance as our input is a random signal that changes every trial, leading to a change in power.

Input was also scaled down and checked through the range between -10 dB and -20 dB so that if any errors occurred in AGC and the value of gain changed, the system could still work with acceptable performance within this range.

Table 11: Minimum SQNR of 100 trials for different input power and bit width

Final Output number of bits	**Input power**	**SQNR**
13	-20 dB	44.0852
13	-15 dB	44.5652
12	-20 dB	42.7101
12	-15 dB	43.8279
11	-20 dB	39.6944
11	-15 dB	42.2623

From the previous figure, we can see that we started with a larger number of bits having SQNR greater than 40 dB and kept decreasing the output number of bits until the minimum SQNR became less than 40 dB, which happened when the output number of bits was 11 and the input power was -20 dB.Hence, our selected output number of bits was 12 bits, as it achieved acceptable performance even in the worst case of input power of -20 dB and a minimum of 100 trials.

The following graph also shows that SQNR kept decreasing with decreasing bit width until SQNR became less than 40 dB at an output of 11 bits when input power was -20 dB.

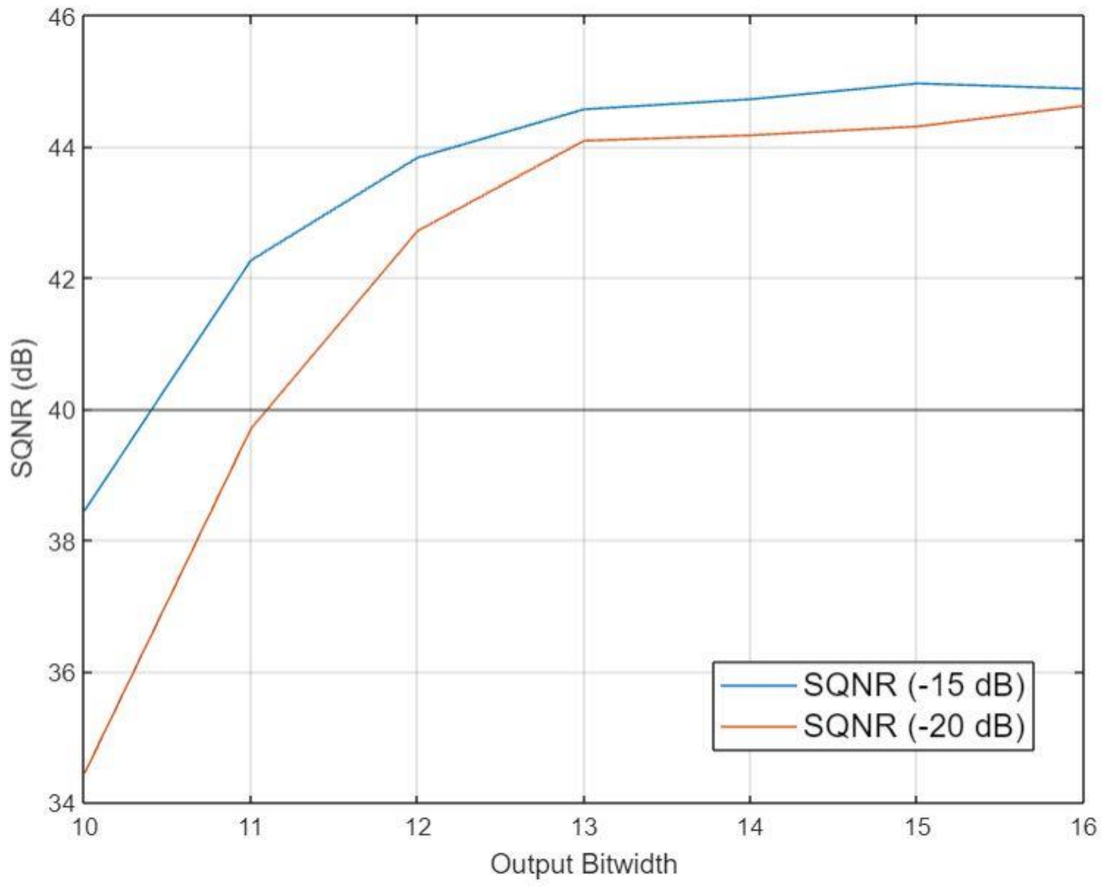


Figure 34: SQNR plotted with number of bits

For outputs of every stage, as observed, 12 bits are sufficient to represent outputs with acceptable performance. All the outputs are within the range of -1 to 1. Hence, no integer part is needed to represent the outputs hence, the output fixed point representation is one bit for sign and 11 bits for fraction (S11).

Decreasing error:

As mentioned before, a 256-point radix-4 FFT has four stages.

In radix-4, the butterfly has 4 inputs that are independent random variables, every input has a variance of σ^2 where variance represents the value of error. The variance of output after adding four inputs becomes $4\sigma^2$ which means that the variance increased. By dividing the output by any value, the variance value decreases. For example, the mean value of the output is equal to $\frac{\sigma 2}{4}$.

Therefore, we need to choose a value to divide the output by after each stage, this value should be an integer power to the base 2 so that we can implement it easily in hardware using shift operations with no extra hardware. Hence, we divide the output of each stage by 2. This is equivalent to shifting the decimal point to the left once, taking a bit from the integer part to the fraction part.

By dividing the output of every stage by 2, it is equivalent to dividing the final output by $\sqrt{N} = \sqrt{256} = 16$.

If we had chosen to divide by 4 (or more), it would have decreased the variance more than dividing by 2. On the other hand, it would have increased the sensitivity of the fractional part, making truncation of bits highly affect SQNR negatively. For example, if we have an integer part, truncation from it would have a higher effect than the fractional part. So when we divide, the integer part will be shifted, and truncation will have a higher effect on SQNR.

Results

After the floating-point model of a 256-point Radix-4 FFT was done and checked using Matlab's FFT built-in function, A fixed-point model was done to optimize in hardware with an acceptable performance of a minimum SQNR equal to 40 dB. This fixed point is S11 for input, S8 for twiddle factor exponentials, and S11 for output. Test vectors were also generated from Matlab for each internal point after each stage of the model to verify our hardware design using test benches. After our model is ready, the next step is to start building the hardware design for a 256-point radix-4 FFT.

3.2.4 Channel Estimation and interpolation

Now we have 4 symbols, but we want only 3 symbols, as the first symbol is the PSS, and we are assuming perfect synchronization so we will continue the process with 3 OFDM symbols.

Let us say that the received PBCH data is YPBCH and the received DMRS data is YDMRS. The demodulation reference signal (DMRS) is used to estimate the pilot values or the channel at the receiver. The YDMRS of length 144 symbols is obtained from the DMRS positions mentioned in Section 3.1.5, Fig. 23. Now, DMRS signal generation at the transmitter end requires the N_{ID}^{cell} and i_{SSB} which are known to the receiver at this stage from synchronization block, but we are assuming perfect synchronization as we aren't modeling the synchronization block. Hence, using N_{ID}^{cell} and i_{SSB} , the DMRS is re-generated at the receiver. Let's call this X_{DMRS}.

We have modelled two modes for channel estimation:
- Least Square Estimator (LS) for the DMRS (pilots), then linear interpolation to get the PBCH (data).
- LS for the DMRS (pilots), Minimum Mean Square Error Estimator (MMSE) to get the PBCH (data).

LSE with linear interpolation:
The received signal is obtained as:

$$Y = XH + N \tag{3.2.1}$$

Y_{DMRS} is the attenuated and delayed version of X_{DMRS} affected due to the Wireless channel model. The channel estimate $\hat{H}_{DMRS}$ is obtained as:

$$\hat{H}_{DMRS} = \frac{Y_{DMRS}}{X_{DMRS}} \tag{3.2.2}$$

Now the channel experienced by the pilots (DMRS) is known but that for the PBCH is unknown. The channel values for the remaining positions of the block are obtained by simple interpolation technique (Linear Interpolation).

Linear Interpolation algorithm, two adjacent pilot subcarriers are used to determine the channel response for data subcarriers in between the pilot signals.
The equation is as below:

$$\hat{H}(k) = \hat{H}_{DMRS}(kN + l) = \frac{\hat{H}_{DMRS}(k + 1) - \hat{H}_{DMRS}(k)}{N} + \hat{H}_{DMRS}(k) \tag{3.2.3}$$

Here k = 0, 1, 2, ..., N and N=4

MMSE channel estimation:

$$Y = XH + Z$$

$$X = diag\{X(0), X(1), \dots, X(N - 1)\}$$

$$H = [H(0)\, H(1) \dots H(N - 1)\,]^T$$

$$Y = [Y(0)\, Y(1) \dots Y(N - 1)]^T$$

$$Z = [Z(0)\, Z(1) \ldots Z(N-1)]^T$$

Where $X[k]$ denotes a pilot tone at the kth subcarrier, with $E\{X[k]\} = 0$ and $var\{X[k]\} = \sigma_x^2$, k=0,1,2,....N-1. H is the channel vector and Z is a noise vector with $E\{Z[k]\} = 0$ and $var\{Z[k]\} = \sigma_z^2$.

Consider the LS estimation $H_{LS} = X^{-1}Y$. Using the weight matrix W, define

$$H_{MMSE} = WH_{LS} \tag{3.2.4}$$

$$W = R_{HP}R_{PP}^{-1} \tag{3.2.5}$$

R_{PP} is the autocorrelation matrix of H_{LS} given as:

$$R_{PP} = E\{HH^H\} + \frac{\sigma_z^2}{\sigma_x^2} I \tag{3.2.6}$$

R_{HP} is the cross correlation between the true channel vector and the temporary channel estimate vector in frequency domain.

$$H_{MMSE} = R_{HP}R_{PP}^{-1}H_{LS}$$

$$H_{MMSE} = R_{HP}\left(R_{HH} + \frac{\sigma_z^2}{\sigma_x^2} I\right)^{-1} H_{LS} \tag{3.2.7}$$

As shown in 3.2.7, although the MMSE calculates the channel estimates, it needs the channel estimates at the DMRS positions (H_{LS}) as input. This is somewhat confusing. You can think about it as it needs the H_{LS} at the DMRS positions to build a trend of the channel at this time instant for different subchannels, then it produces the accurate estimates for all subchannels including the DMRS, which means the input estimates are different from the output, and this proves that the MMSE is not an interpolation method but a real estimator.

To simplify this equation, we assumed a uniform distribution for channel correlation, then the frequency domain will be a sinc function.

So, $R_{HP} = sinc(K * \frac{nTaps}{Nfft})$, $R_{HH} = sinc\left(K * \frac{nTaps}{\frac{Nfft}{4}}\right) + \frac{I}{SNR}$

Which: K is a vector to sum all sinc functions applied at each LS,
nTaps is the number of channel taps in time domain,
Nfft is the FFT size.

Note: $nTaps = 1$ for AWGN, otherwise $nTaps = \frac{Sampling\ Time}{Maximum\ Delay\ Spread}$

3.2.5 Channel Equalization

Before equalization, we average the $\hat{H}_{240x3}$ to $\hat{H}_{240x1}$ to ensure the accuracy of the estimation. Channel Equalization is performed by using Maximum Ratio combining (MRC) we have,

$$Y = HX + N \tag{3.2.8}$$

Y: a 240×3 matrix of complex symbols of received SSB,
Ĥ: a 240×1 matrix of channel estimates.
The estimate of each OFDM symbol in the SSB data is given by:

$$\hat{X} = conj(\hat{H}) * Y \qquad (3.2.9)$$

3.2.6 Resource De-mapping

The $\hat{X}$ obtained is in the form of a 240×3 grid with unwanted pilot estimates. The resources are de-mapped as per the positions mentioned in Section 3.1.5, Fig. 22 to obtain a 432x1 vector of noisy complex values of the PBCH estimate.

3.2.7 De-modulation

The 432 complex values are demodulated to obtain 864 length noisy values of PBCH. The demodulation is QPSK as the PBCH is QPSK modulated at the transmitter.

3.2.8 De-Scrambling

The 864 values are LLR values, which are then descrambled using the same PN sequence generated at the transmitter end. However, as the values are no longer binary but LLR, the sequence is multiplied instead of XOR.

3.2.9 De-Rate Matching

The 864 LLRs are de-rate matched to 512 in three stages: de-interleaving, bit de-selection, and de-subblock interleaving. The bit de-selection process involves puncturing the last 864-512 = 352 values from the sequence and averaging them with the first 352 values of the original sequence to get a new sequence of 512 values. This is done to avoid the loss of received data and reverse the algorithm performed at the transmitter end. Similarly, the de-sub block interleaver uses the same table of interleaving patterns to rearrange the blocks of 32 values in their original positions to finally obtain the 512 values for the decoder.

3.2.10 Simulation Results

Floating-Point BER curves

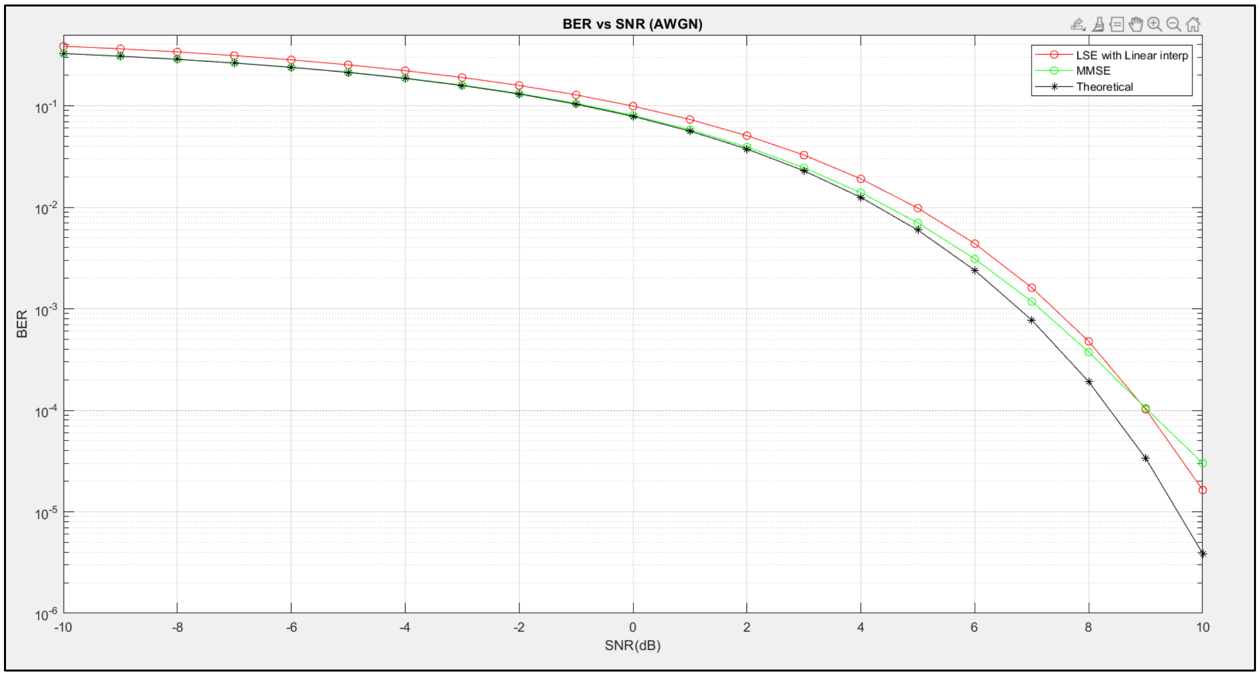


Figure 35: Linear Interpolation vs MMSE (Floating-Point)

As shown in Fig. 35, The curve of MMSE is lower than the curve of LSE with linear interpolation, which means that MMSE has a lower BER than LSE with linear interpolation at the same SNR, so MMSE has better performance than LSE with linear interpolation as expected because the MMSE channel estimator knows the length of the channel in time (number of channel taps) and the statistics of the channel (R_{HP}, R_{PP}). At very high SNR (SNR = 9 dB), the LSE curve intersects MMSE curve because the noise becomes too small and can be negligible. So, the LSE becomes better than the MMSE as it assumes that $H_{LS} = \frac{Y}{X}$, which is the case at very high SNR.

<u>Fixed-Point vs Floating-Point BER curves</u>

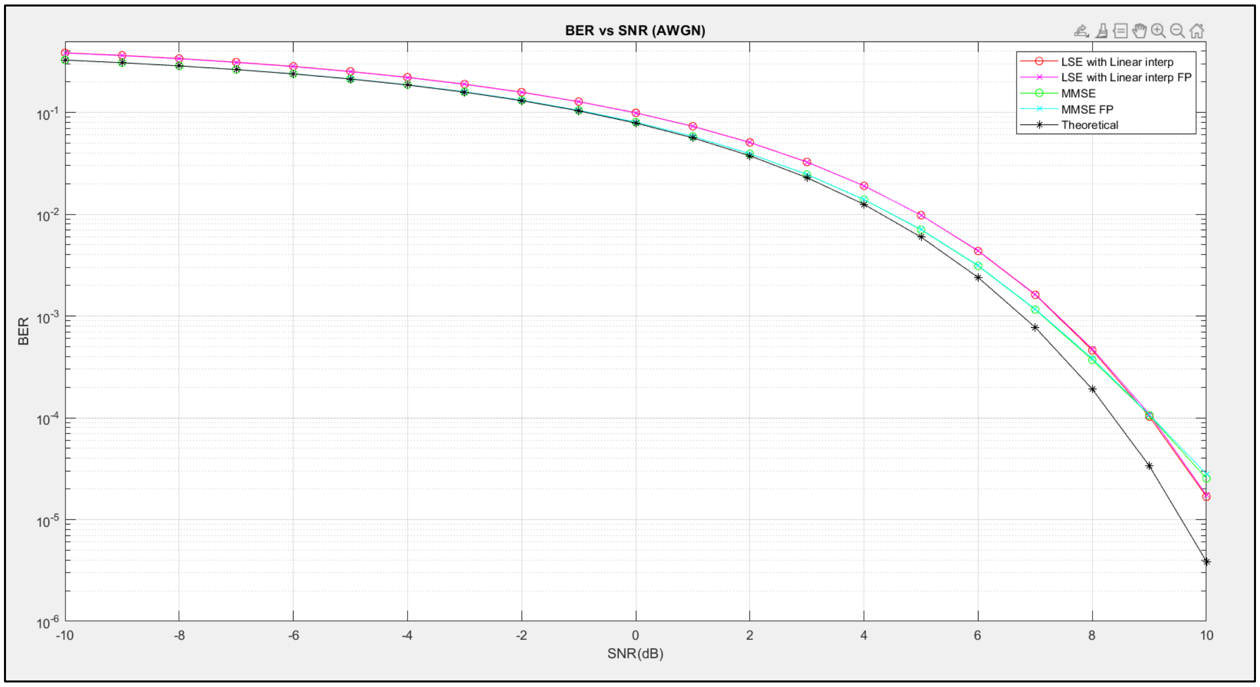


Figure 36: Linear Interpolation vs MMSE (Floating-Point and Fixed-Point)

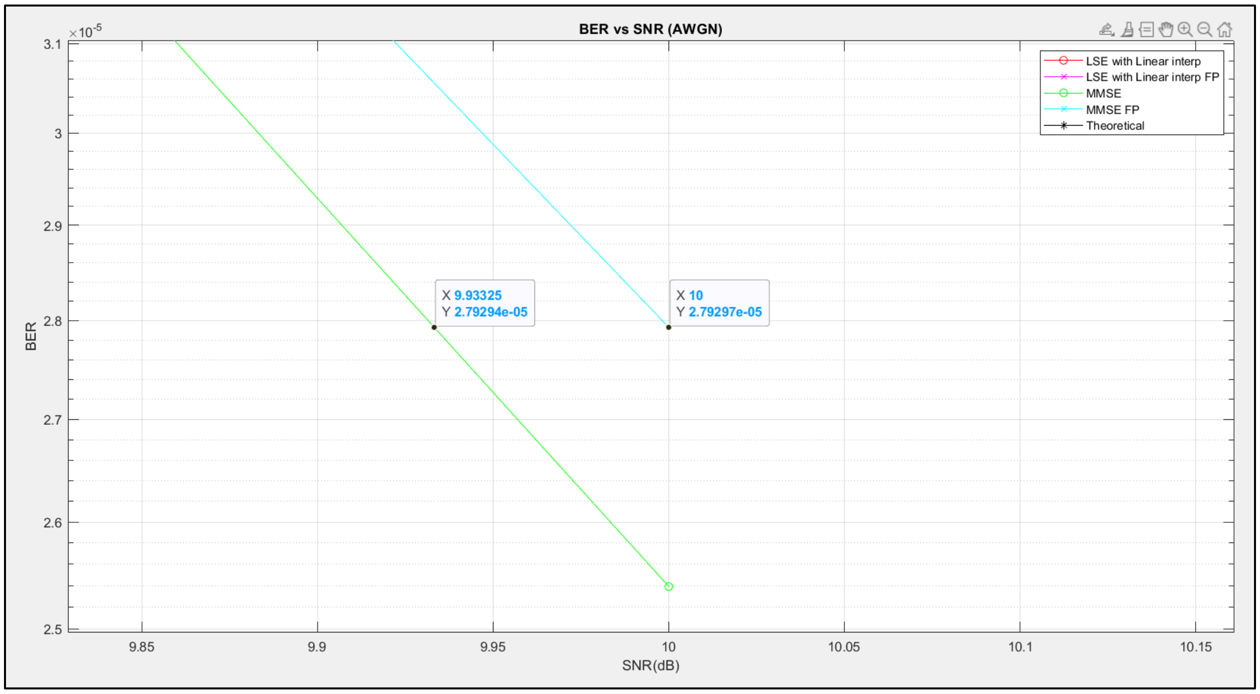


Figure 37: Quantization error due to Fixed Point in MMSE

As shown in Fig. 36, the fixed-point curve is slightly higher than the floating-point curve, which means the fixed-point curve has a higher BER at the same SNR. In other words, the difference between the two curves is less than 0.1 dB at the same BER, which is an acceptable error due to the fixed-point analysis. The goal is to achieve a maximum quantization error of 0.1 dB at very high SNR as shown in Fig. 37. This has been achieved at a fixed-point length of 8 bits (S0.7).

Chapter 4: Hardware Design of FFT

4.1 Introduction

The fast Fourier transform (FFT) is one of the most important algorithms in the field of digital signal processing. It is used to calculate the discrete Fourier transform (DFT) efficiently. In order to meet the high performance and real time requirements of modern applications, hardware designers have always tried to implement efficient architectures for the computation of the FFT.

FFT stands for Fast Fourier Transform. It is an algorithm used to efficiently compute the Discrete Fourier Transform (DFT) of a sequence or a signal. The DFT is a mathematical operation that transforms a time-domain signal into its frequency-domain representation.

Discrete Fourier Transform:

The Discrete Fourier Transform (DTF) can be written as follows.

$$X[k] = \sum_{n=0}^{N-1} x(n)\, e^{\frac{-2\pi jkn}{N}}$$

Where $k \;=\; 0,1,\ldots,N-1$

For each k value, we need N complex multiplications and N-1 complex additions.

$$\begin{tikzcd}[column sep=small] \text{Therefore, for N values} \arrow[r] \arrow[dr] & N * N \;=\; N^2 \text{ multiplications} \\ & N * (N-1) \;=\; N^2 - N \text{ Additions} \end{tikzcd}$$

The FFT algorithm was developed by Cooley and Tukey in the mid-1960s and revolutionized digital signal processing by significantly reducing the computational complexity of the DFT. It takes advantage of the symmetry and periodicity properties of sinusoidal functions to achieve a faster computation.

The FFT algorithm decomposes a DFT of size N into a series of smaller DFTs, exploiting the divide-and-conquer strategy. By recursively dividing the input sequence into smaller sub-sequences, the FFT algorithm reduces the computational complexity from O(N^2) (as in the naive DFT computation) to O (N log N), where N is the size of the input sequence.

Begin with normal FFT:

$X[k] = \sum_{n=0}^{N-1} x(n)\, e^{\frac{-2\pi jkn}{N}}$,k = 0,1, …., N − 1.

Now we can divide the N-point FFT into N/2 point FFTs one for the even and one for the odd indices (summation is linear, so there is no effect).

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} x_{2m}\, e^{\frac{-2\pi jk(2m)}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1}\, e^{\frac{-2\pi jk(2m+1)}{N}} \qquad ,k = 0,1 \dots ., N-1$$

We could define that: $W_N^{nk} = e^{\frac{-2\pi jkn}{N}}$

Therefore, the equation becomes:

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} x_{2m}\, W_N^{2mk} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1}\, W_N^{(2m+1)k} \qquad ,k = 0,1 \dots ., N-1$$

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} x_{2m}\, W_N^{2mk} + W_N^k \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1}\, W_N^{2mk} \qquad ,k = 0,1 \dots ., N-1$$

$X[k] = E[k] + W_N^k O[k]$, E[k]: even part O[k]: odd part

Each of E and O has N/2 samples, DFT is cyclic, thus we only calculate N/2 points each and calculate the remaining N/2 points from the first N/2 as:

- $E[k] = E\left[k + \frac{N}{2}\right]$
- $O[k] = O\left[k + \frac{N}{2}\right]$
- $W_N^{k+\frac{N}{2}} = e^{\frac{-2\pi j(k+\frac{N}{2})}{N}} = e^{\frac{-2\pi jk}{N}} . e^{\frac{-\pi jN}{N}} = - W_N^k$
- $X[k] = E[k] + W_N^k O[k] \quad ,k = 0,1, \dots ., N/2-1$
- $\quad = E\left[k - \frac{N}{2}\right] - W_N^k O\left[k - \frac{N}{2}\ \right], k = \frac{N}{2}, \dots \dots, N$

What happened to complexity so far?

- Complexity becomes $2\left(\frac{N}{2}\right)^2 + N = \frac{N^2}{2} + N$

Since we will calculate two sets of N/2 point DFTs then add additional N complexity of additions.

Projecting further, the two N/2 FFTs will be further broken into N/4 FFTs and so on leading to the logarithmic complexity.

4.2 Architecture Selection

First, we found that there are two approaches to implement the FFT hardware architecture either to implement it parallel or serial.

Where Parallel pipelined architectures such as multi-path delay feedback (MDF) architecture, multi-path delay commutator (MDC) architecture and multi-path serial commutator (MSC) architecture where multiple data streams are processed concurrently in parallel through different stages of the pipeline. Each stage operates on a separate data stream simultaneously, and the pipeline stages are designed to execute their operations independently and concurrently. This architecture maximizes parallelism, allowing for multiple operations to be performed simultaneously and increasing the overall input of the system.

While Serial pipelined architectures such as single-path delay feedback (SDF) architecture, single-path delay commutator (SDC) architecture and single-path serial commutator (SC) architecture, the data flows through a sequence of stages, and each stage processes a single data stream serially before passing it to the next stage. In this architecture, only one operation is active at a time in each pipeline stage, and the data progresses through the stages in a sequential manner.

Then we headed to serial pipelined architectures over the parallel pipelined architectures which is more suitable in our application as the nature of the signal coming from ADC is coming as a sample at a time so the data samples are coming serially so we do not need to implement our architecture parallel as it will cost us to implement an external memory to store in it our serial data samples, while in serial architectures we already have memory elements inside our architecture.

Then from serial pipelined architectures, we chose to implement the Single-path delay feedback architecture (SDF) [6].

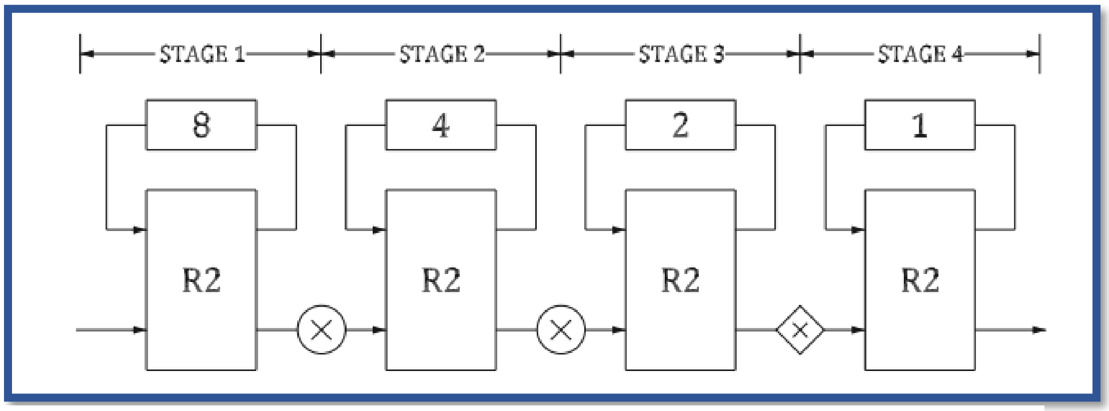


Figure 38: 16-point radix-2 SDF FFT architecture

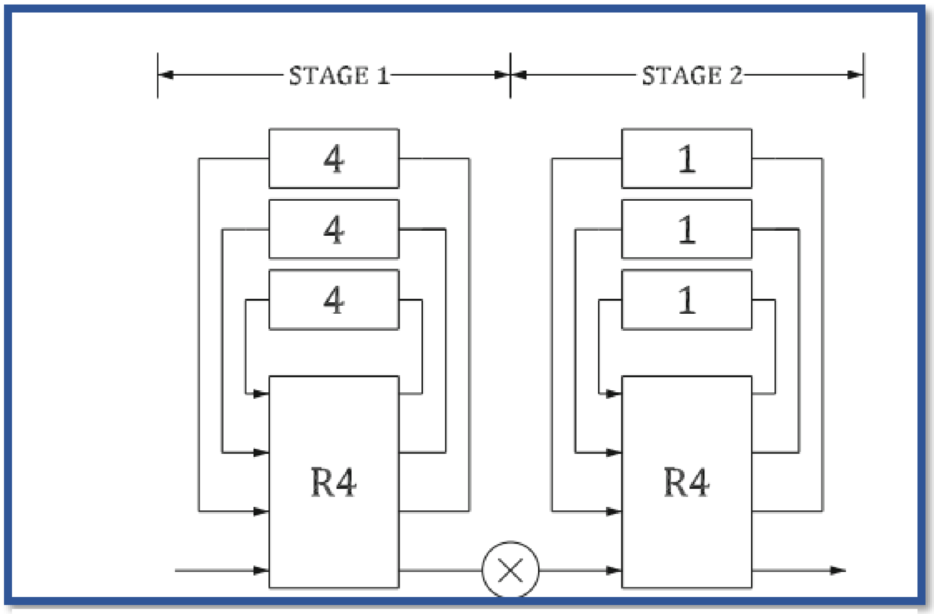


Figure 39: 16-point radix-4 SDF FFT architecture

4.2.1 Why we chose SDF architecture

- Reduce Hardware Complexity: The SDF FFT architecture simplifies the hardware design by utilizing a single feedback loop and delay elements. This leads to a reduction in the number of required hardware components compared to other FFT architectures. As a result, the implementation of the SDF FFT architecture is generally more straightforward and requires fewer resources.
- Lower Memory Requirements: The SDF FFT architecture typically requires less memory compared to other FFT architectures. It achieves this by reusing and updating the same memory locations as the input data progresses through the stages. This reduces the need for additional memory buffers and simplifies the memory management aspect of the design.
- Regular Computation Patterns: The SDF FFT architecture exhibits regular computation patterns as the data flows through the stages. This regularity allows for efficient pipelining. It facilitates the use of regular and predictable hardware resources, which simplifies the scheduling and optimization of the pipeline.
- Resource Efficiency: Due to its reduced hardware complexity and memory requirements, the SDF FFT architecture offers resource efficiency. It makes efficient use of hardware resources such as multipliers, adders, and registers so it has high utilization. This makes it particularly useful for applications with limited resources or embedded systems where resource utilization is a crucial factor.
- Simplified Control and Synchronization: The single feedback loop and regular computation patterns simplify the control and synchronization aspects of the architecture. The control logic can be designed to operate in a straightforward and predictable manner, leading to easier verification and debugging processes.

Overall, the Single-Path Delay Feedback FFT architecture offers a balance between hardware simplicity and computational efficiency. It is well-suited for applications with resource constraints, real-time processing requirements, and situations where simplicity of hardware implementation is a priority.

4.3 Radix Selection

First, we have to define or know what radix is.

In the context of the Fast Fourier Transform (FFT), the term "radix" refers to the base of the number system used in the butterfly computations. The butterfly operation is the fundamental computation unit in the FFT algorithm, where the results of previous stages are combined to compute the final Fourier transform.

During the FFT computation, the input sequence is divided into smaller subsequences, and the butterfly operation combines the results from previous stages. The radix determines the number of samples or points involved in each butterfly computation.

For example, in a radix-2 FFT, the input sequence is divided into two subsequences, and each butterfly computation involves two points. The radix-2 FFT algorithm recursively applies this process until the base case is reached, where each subsequence has only one point, Here is an example of 8-point radix-2 butterfly in figure 40.

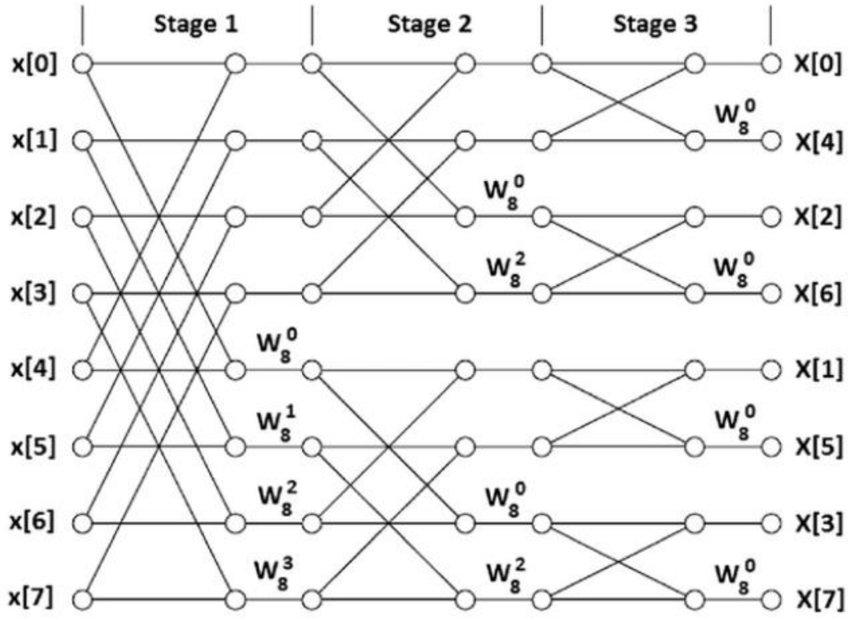


Figure 40: 8-point radix-2 FFT

Similarly, in a radix-4 FFT, the input sequence is divided into four subsequences, and each butterfly computation involves four points. The radix-4 FFT algorithm continues the process recursively until the base case of one point is reached, here is an example of 16-point radix-4 butterfly in figure 41.

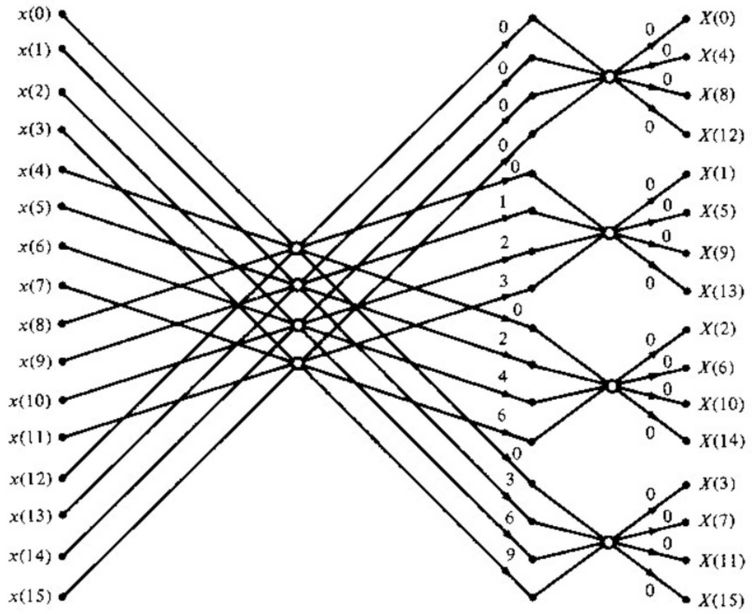


Figure 41: 16-point Radix-4 FFT

The choice of radix in the FFT algorithm impacts the number of arithmetic operations required and the structure of the computation. Different radices can result in different trade-offs between computational complexity and memory requirements.

In summary, the radix in the FFT algorithm represents the base of the number system used in the butterfly computations and determines how the input sequence is divided and combined during the computation of the Fourier transform.

Now let us look at trade-offs between some different radices in the following table and compare them due to the number of stages and butterfly complexity.

Table 12: Radix Selection trade-offs

WHERE N = 256 IN OUR CASE	NUMBER OF STAGES	BUTTERFLY COMPLEXITY
RADIX-2	$\log_2 N = \log_2 256 = 8$ stages	Simple
RADIX-4	$\log_4 256 = 4$ stages	More complex than radix-2 but still simple due to trivial multiplications
RADIX-8	In that case we can mix between radix and it will take 3 stages.	More complex
RADIX-16	$\log_{16} 256 = 2$ stages	More and more complex

We can conclude that as the number of the radix increase the complexity of the butterfly itself will increase. And as the number of the radix decrease the number of stages increase so the latency will increase.

Then we chose Radix-4 in order to compromise between number of stages and the complexity of butterfly In addition to that we were lucky to find that the butterfly complex multiplications is with j,-j, 1,-1 that can be implemented by adders and subtractors without requiring to implement a complex multiplier inside the butterfly since that they are all trivial multiplications. So let us now go through Radix-4 algorithm to verify or to proof that we will get trivial multiplications.

4.3.1 Radix-4 FFT Algorithm:

The butterfly of a radix-4 algorithm consists of four inputs and four outputs. The FFT length is 4^M, where M is the number of stages where in our case we have N = 256 point where the number of stages (M)= $\log_4 256 = 4\ stages$. A stage is half of radix-2. The radix-4 DIF FFT divides an N-point discrete Fourier transform (DFT) into four N/4 -point DFTs, then into 16 N/16 -point DFTs, and so on. In the radix-2 DIF FFT, the DFT equation is expressed as the sum of two calculations. One calculation sum for the first half and one calculation sum for the second half of the input sequence. Similarly, the radix-4 DIF fast Fourier transform (FFT) expresses the DFT equation as four summations. The following equations illustrate radix-4 decimation in frequency (DIF)[8].

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{nk}$$

$$= \sum_{n=0}^{N/4-1} x(n)\, W_N^{nk} + \sum_{n=N/4}^{2N/4-1} x(n)\, W_N^{nk} + \sum_{n=2N/4}^{3N/4-1} x(n)\, W_N^{nk} + \sum_{n=3N/4}^{N-1} x(n)\, W_N^{nk}$$

$$= \sum_{n=0}^{N/4-1} x(n)\, W_N^{nk} + \sum_{n=0}^{N/4-1} x\left(n + N/4\right) W_N^{(n+N/4)k} + \sum_{n=0}^{N/4-1} x\left(n + N/2\right) W_N^{(n+N/2)k}$$
$$+ \sum_{n=0}^{N/4-1} x\left(n + 3N/4\right) W_N^{(n+3N/4)k}$$

$$= \sum_{n=0}^{N/4-1} \left[x(n) + x\left(n + N/4\right) W_N^{(N/4)k} + x\left(n + N/2\right) W_N^{(N/2)k} + x\left(n + 3N/4\right) W_N^{(3N/4)k}\right] W_N^{nk}$$

Therefore, four twiddle factor coefficients can be expressed as follows:

- 1
- $W_N^{(N/4)k} = e^{\frac{-\pi jk}{2}} = \left[\cos\left(\pi/2\right) - j\sin\left(\pi/2\right)\right]^k = (-j)^k$
- $W_N^{(N/2)k} = e^{-\pi jk} = [\cos(\pi) - j\sin(\pi)]^k = (-1)^k$
- $W_N^{(3N/4)k} = e^{\frac{-3\pi jk}{2}} = \left[\cos\left(3\pi/2\right) - j\sin\left(3\pi/2\right)\right]^k = (j)^k$

X(k) can thus be expressed as:

$$X(k) = \sum_{n=0}^{N/4-1} \left[x(n) + x\left(n + N/4\right)(-j)^k + x\left(n + N/2\right)(-1)^k + x\left(n + 3N/4\right)(j)^k\right] W_N^{nk}$$

Then to arrive at a four-point DFT decomposition, let $W_N^4 = W_{N/4}^1$ therefore the equation can be written as four N/4 point DFTs, or

$$\text{X(4k)} = \sum_{n=0}^{N/4-1}\left[x(n) + x\left(n + N/4\right) + x\left(n + N/2\right) + x\left(n + 3N/4\right)\right] W_{N/4}^{nk} \quad (1)$$

$$\text{X(4k+1)} = \sum_{n=0}^{N/4-1}\left[x(n) - jx\left(n + N/4\right) - x\left(n + N/2\right) + jx\left(n + 3N/4\right)\right] W_N^{2n}\, W_{N/4}^{nk} \quad (2)$$

$$\text{X(4k+2)} = \sum_{n=0}^{N/4-1}\left[x(n) - x\left(n + N/4\right) + x\left(n + N/2\right) - x\left(n + 3N/4\right)\right] W_N^{2n}\, W_{N/4}^{nk} \quad (3)$$

$$\text{X(4k+3)} = \sum_{n=0}^{N/4-1}\left[x(n) + jx\left(n + N/4\right) - x\left(n + N/2\right) - jx\left(n + 3N/4\right)\right] W_N^{3n}\, W_{N/4}^{nk} \quad (4)$$

For $k\ = 0\ to\ N/4\ -1$

Note: from equations 1,2,3,4 the twiddle factors coefficients forms our matrix that will be implemented in our butterfly.

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}$$

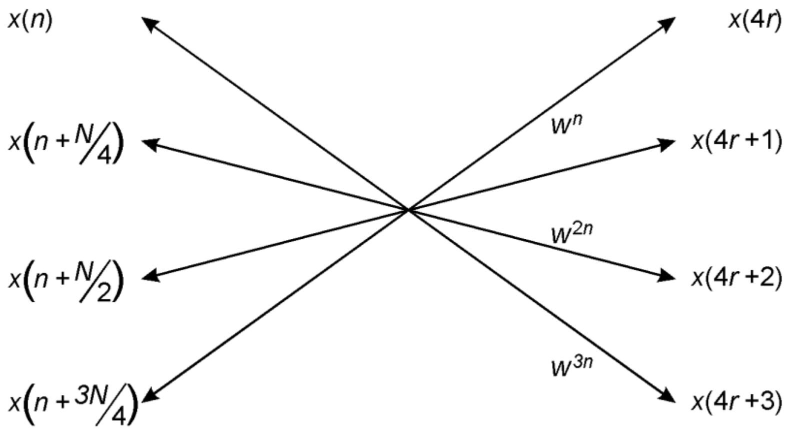


Figure 42: Radix-4 Butterfly

4.4 Hardware Architecture Optimization

SDF Architecture fully pipelined with four engines each engine is used to run one stage is shown in figure 43.

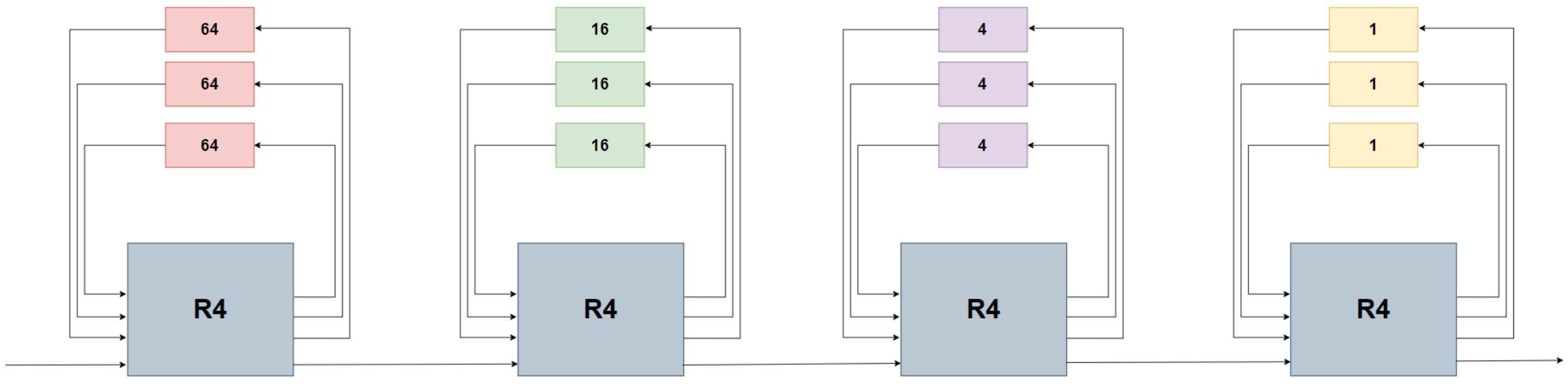


Figure 43: SDF Architecture fully pipelined

After selecting the suitable architecture (Single-path delay feedback) and Radix for our application, we thought if we could optimize more in our architecture by implementing less number of engines and processing units instead of implementing 4 engines one for each stage so we asked ourselves if we could run more than one stage on the same engine in order to reduce the area of our design. So now we have two options:

Option 1: That we run all stages on two engines each one processes two stages. SDF architecture with two engines is shown in figure 44.

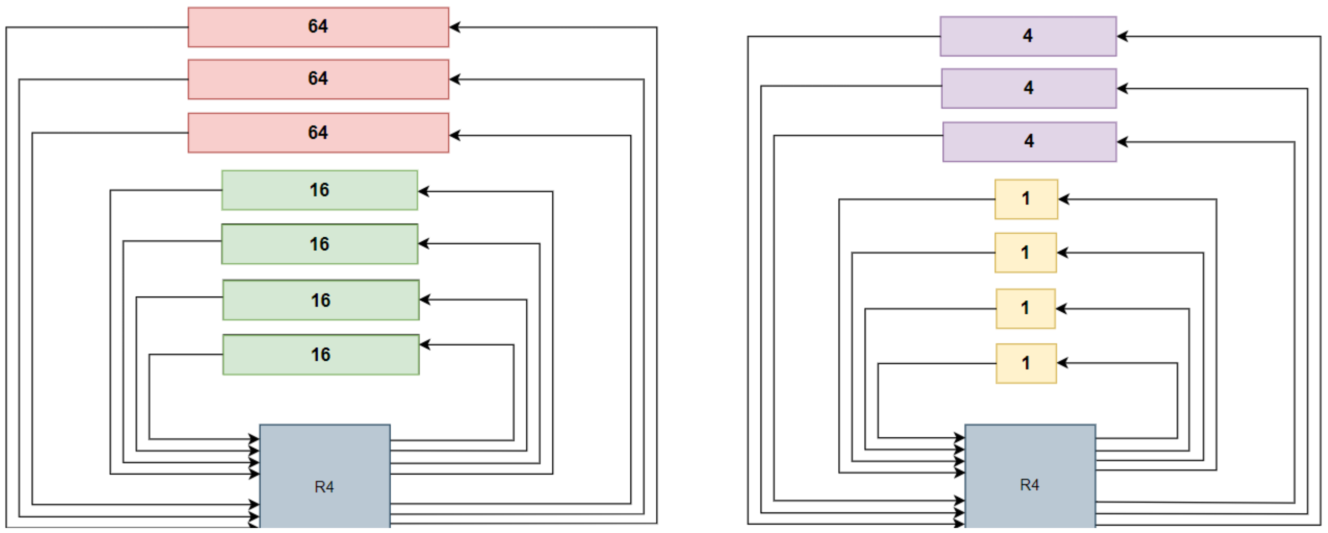


Figure 44: two engines SDF Architecture

In that architecture we reduced the area by using only two engines instead of the four engines so we reduced the area and the number of used processing units so lower area and power consumption with a little bit increase in the number of storing elements which is represented as FIFOs in our architecture.

Option 2: That we run all stages on one engine ONLY

SDF architecture with Single engine is shown in figure 45.

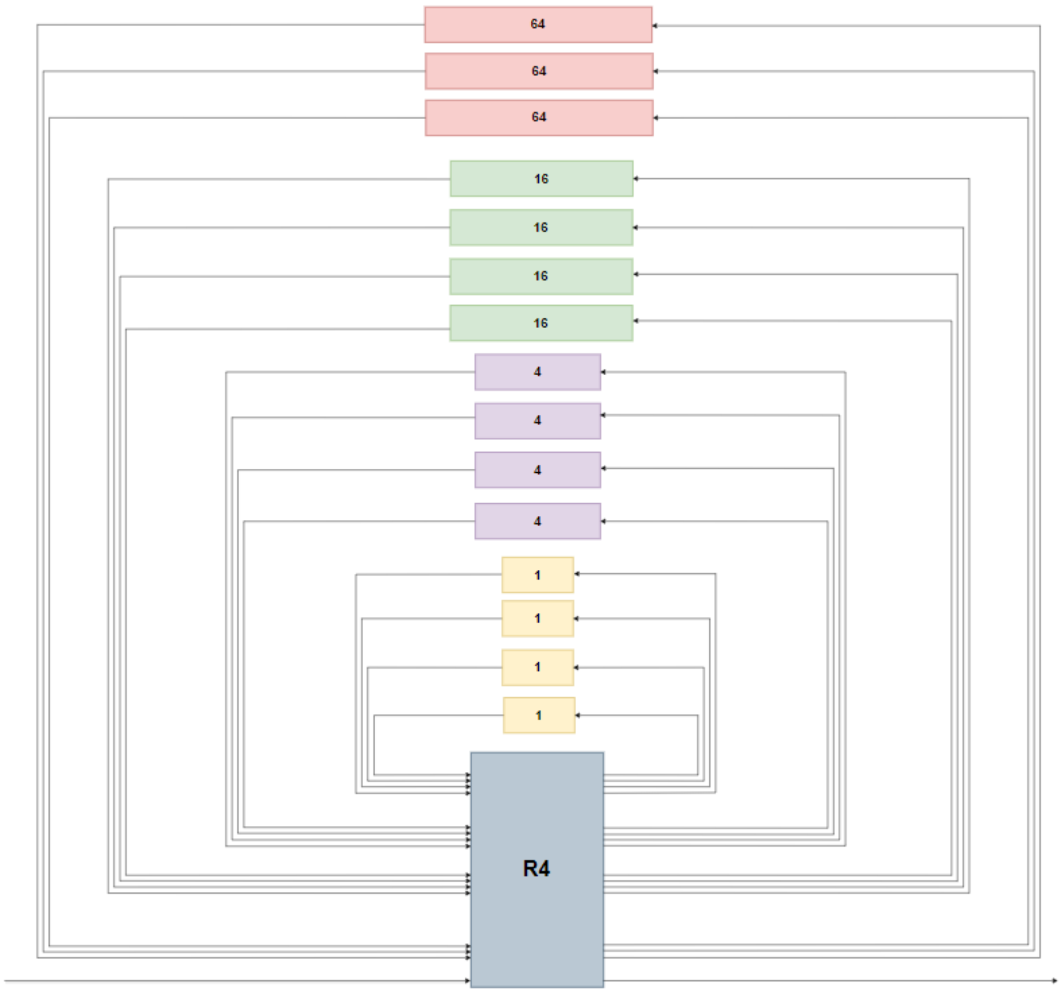


Figure 45: Single engine SDF Architecture

It is obvious that the SDF Architecture with single engine is the least one to cost area in our system so it will optimize the system area in addition to that we reduced the number of used processing units four times compared to four engines architecture and that gives us lowest power consumption if we implemented it compared to previous two architectures, but first we have to check that its latency does not exceed the maximum allowable latency of our system or application.

Therefore, the decision to choose which architecture to implement is dependent on a threshold which is the maximum allowable latency in NR technology.

In order to calculate the maximum allowable latency of our system we should first indicate our working system frequency.

4.4.1 Working Frequency indication:

- First we have 256 sample per symbol and numerology (μ) = 0 that means in NR technology that we have sub carrier spacing 15 kHz.
- Therefore our sampling frequency $(f_{sample}) = 256 * 15K = 3.84\ MHZ$, we chose that our clock frequency will be a multiple of f_{sample} so we decided to go with $16 * f_{sample}$ to be our working clock frequency $= 16 * 3.84\ MHz = 61.44\ MHz$.
- We can say now that the system will receive sample each 16 clock cycle.

Note: by working with frequency multiple of input frequency this makes design more flexible

4.4.2 Calculations to get the maximum allowable Latency:

- According to NR 5G we have 14 symbols per slot, since in our case the sub carrier spacing is 15 kHz (μ=0) therefore slot time is 1ms [7]. Therefore, with clock frequency 61.44 MHz we have 61440 clock cycles per 1ms, so the maximum latency for our architecture could be $\frac{number\ of\ clk\ cycles}{number\ of\ symbols} = \frac{61440}{14} = 4388$ clock cycles of latency.
- So, if we could achieve this latency or less with the single engine architecture therefore it will suitable for us to use it in our application as it also will optimize a lot in area.

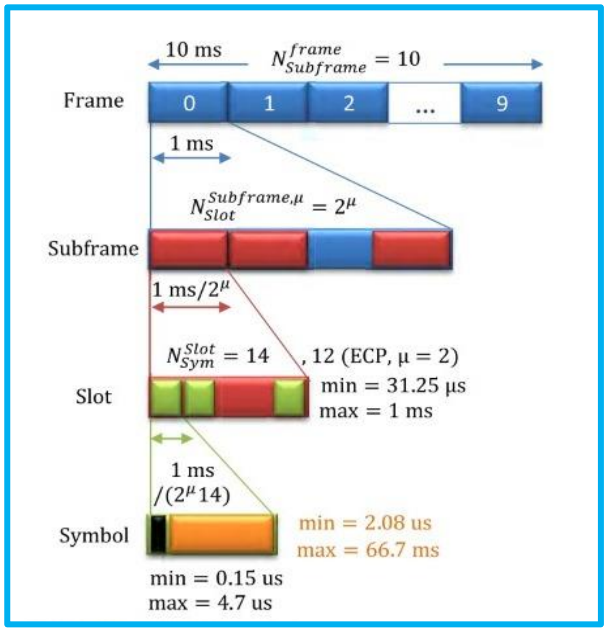


Figure 46: Frame in 5G NR technology

Before calculating the single engine architecture latency, we would like to define first what latency is.

We define latency that it is the number of clock cycles from which the architecture starts processing till we are done and execute the last sample of the output. Where latency is defined for **first message or first symbol only**

4.4.3 Single engine SDF Architecture Latency:

In order to calculate or identify architecture latency we made a timing diagram to help us to calculate it.

<u>Timing Diagram:</u>

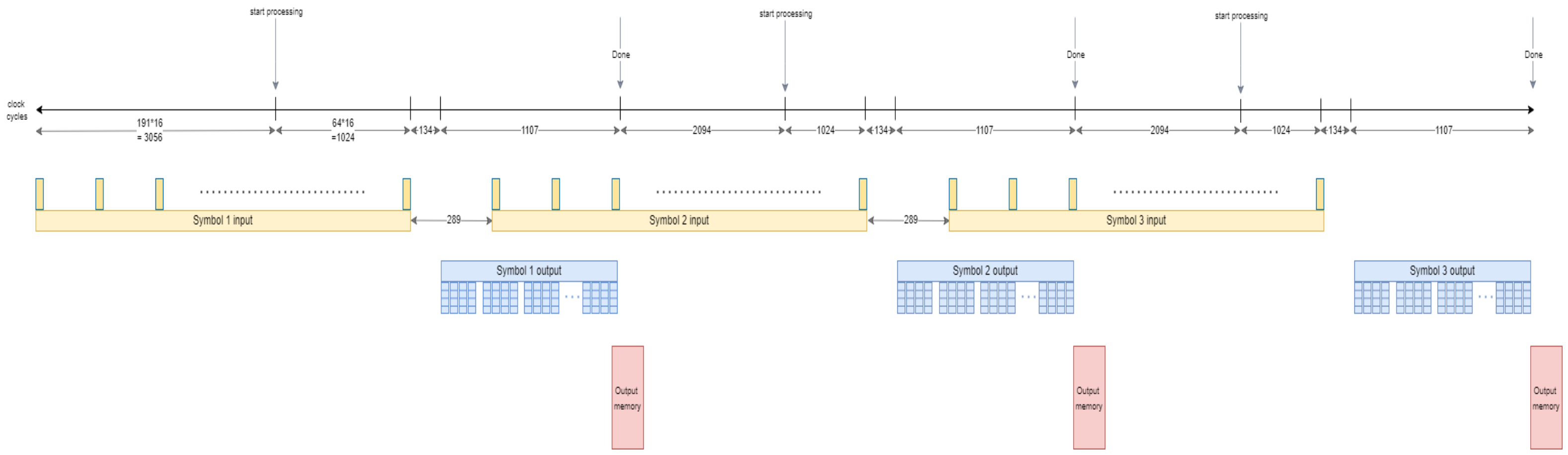


Figure 47: Single engine SDF Architecture time diagram

As shown in the timing diagram that the number of clock cycles taken by the single engine architecture from start processing point till done flag at which all samples of output are processed is 2265 clock cycles.

Note: Between input symbols there are 289 clock cycles that represent Cyclic Prefix samples since cyclic prefix time is equal 4.69 µsec in NR 5G technology[9] therefore the number of clock cycles of CP = $4.69\mu sec * 61.44MHz = 289\ clock\ cycles$

Conclusion:

It is found that single engine architecture has latency less than maximum allowable latency therefore we could implement it without any problem and achieving high optimization in area and lower power consumption.

4.5 SDF Architecture methodology

In this section we will focus on how data flow in our architecture deeply:

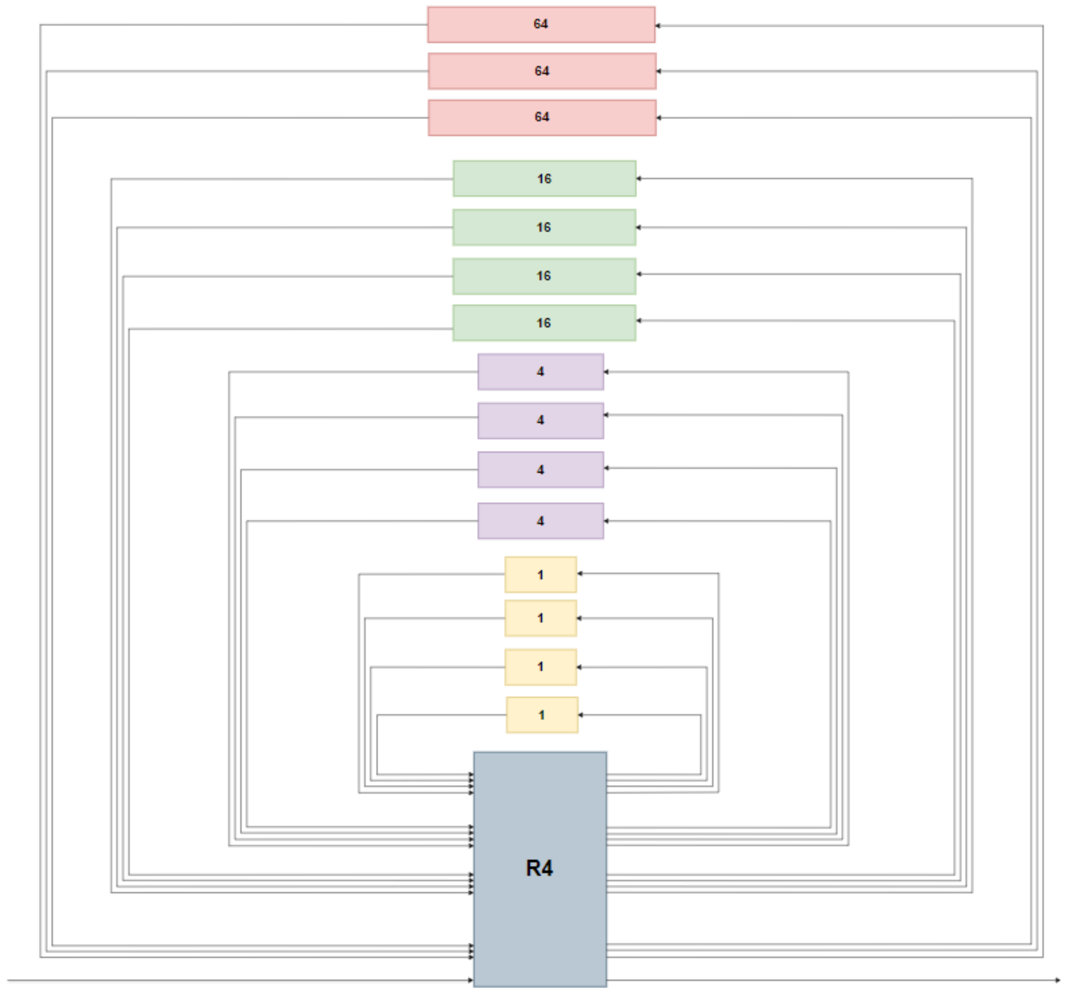


Figure 48: Single engine Architecture

First the system is settled in an idle state till a flag comes to it that indicates that the data samples are going to enter the system then our architecture starts to store three quarters of data samples of symbol as shown in timing diagram figure 47 in the three FIFOs of size 64 since it is a radix-4 architecture in order to pass to butterfly 4 samples each one is from different quarter of data samples as x[0] , x[64],x[128], x[192].

4.5.1 Storing in FIFOs methodology

How data samples are stored in FIFOs in Storing state?
The data samples are stored one by one inside the 1st FIFO until the FIFO is full where 64 samples are stored then we switch to store in the 2nd FIFO until it is full then switch to the 3rd FIFO till it becomes full as shown in figure 49.
Where all of this is controlled or directed by our main controller that will be discussed later in the RTL section.

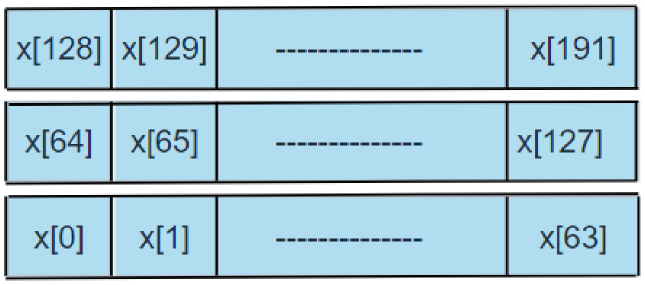


Figure 49: FIFOs of size 64

Now we could say that we stored 192 data samples and the next coming sample is x[192] so by now we could start processing our data samples where four samples will enter the first processing unit in our flow which is the butterfly where samples x[0],x[64],x[128] will be read from FIFOs of size 64 and sample x[192] will be coming from outside as input.

4.5.2 Stages sequence and iterations:

First, we have to know that each stage of our four processing stages has $\frac{256}{4} = 64\ iterations.$
So what is the sequence of these stages and their iterations?
1. We run all 64 iterations of stage 1.
2. A Loop of iterations is repeated 4 times.
    2.1) 16 iterations of stage 2.
    2.2) 4 iterations of stage 3.
    2.3) 4 iterations of stage 4 (the stage at which the output data samples are extracted).
    2.4) Repeat step 2.2 and 2.3 another three times alternately.

By applying this sequence, we will have finished processing on a whole symbol with 64 iterations per stage, as shown in figure 50.

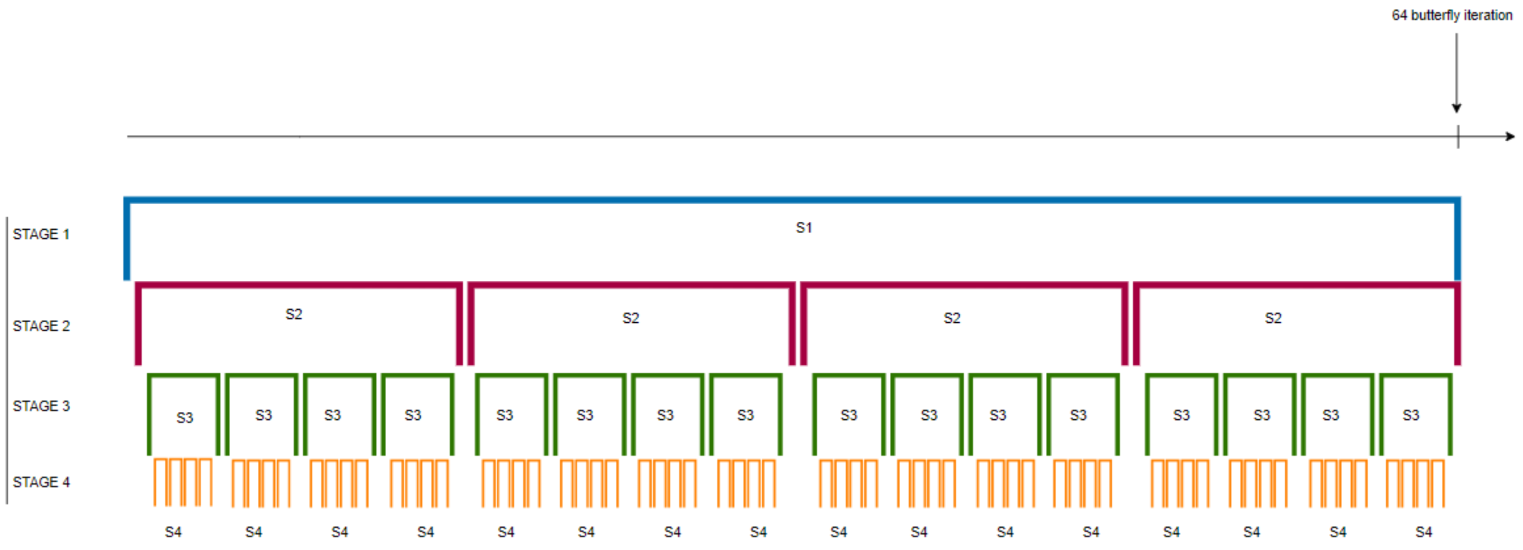


Figure 50: Diagram that shows Stages iterations

4.5.3 Processing Unit Flow

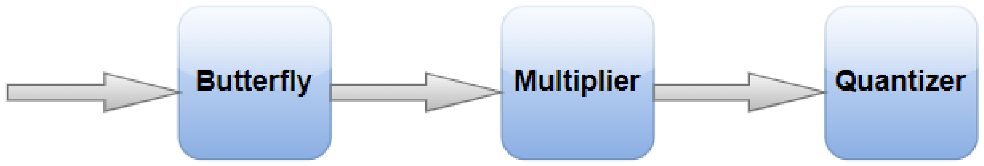


Figure 51 Processing Unit Flow

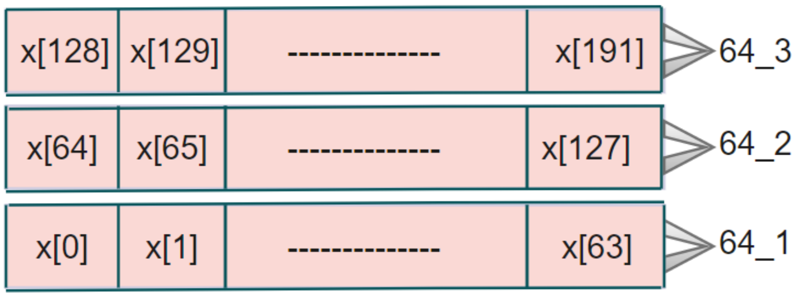
1. Data samples enter the butterfly and the matrix mentioned before is applied on them.
2. Each sample is multiplied by its twiddle factor in multiplier.
3. Each sample is quantized in quantizer.

This part will be discussed in details later in RTL section.

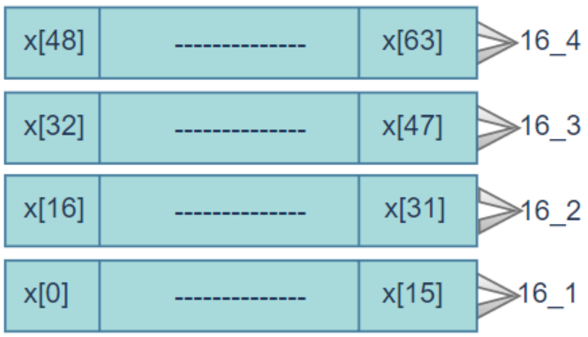
4.5.4 Data Samples Flow

First, we would like to give each FIFO a number at each group of same size FIFOs in order to facilitate the clarification of data flow.

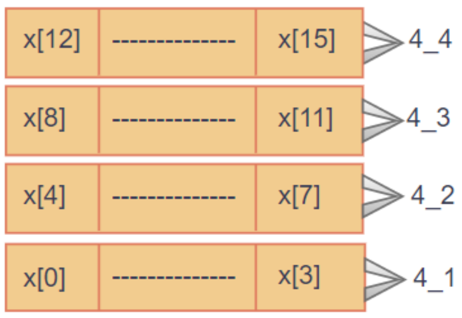
- FIFOs of size 64



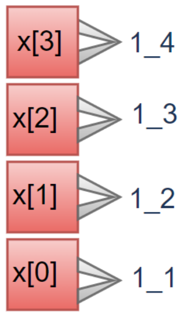
- FIFOs of size 16



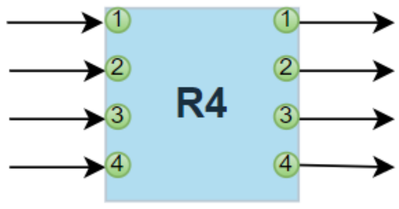
➢ FIFOs of size 4



➢ FIFOs of size 1



Now we will go through with data samples flow in each stage iteration in details:



N.B:

Processing unit inputs connections

- 1st input port is connected to fifos of number 1 that store the former or premier samples until it is full
- 2nd input port is connected to fifos of number 2 that store the next following samples until it is full
- 3rd input port is connected to fifos of number 3 that store the next following samples until it is full
- 4th input port is connected to fifos of number 4 that store the next following samples until it is full or if we are in stage 1 processing with fifos of size 64 the 4th port is connected with input data samples.

Where this is applied with all fifos with all sizes and certainly each port is connected to a MUX in order to be connected with four fifos of different sizes as it shown later in RTL part

Now let us go to our main part which is Data flow at each stage:

STAGE 1:

After our 4 samples finish our processing flow:

1. Output sample number 1 is transmitted to FIFOs of size 16 and by applying the 64 iterations of stage 1 the four FIFOs of size 16 will be filled with the same methodology as discussed before with output samples of stage 1 that are now ready to start processing at stage 2.
2. The output samples number 2,3 and 4 are fed back to FIFOs of size 64 where:
   2nd output is transmitted to FIFO of number 1 (64_1)
   3rd output is transmitted to FIFO of number 2 (64_2)
   4th output is transmitted to FIFO of number 3 (64_3)
   And this is repeated for the 64 iterations until the three FIFOs of size 64 are filled with output samples of stage 1, that going to be shifted later to FIFOs of size 16 during processing on the next stages and the shifting methodology will be discussed in details later.

STAGE 2:

Starting from this stage our architecture is able to receive data samples of a new symbol.

After our 4 samples finish our processing flow:

1. Output sample number 1 is transmitted to FIFOs of size 4 and by applying the 16 iterations of stage 2 (per loop) the four FIFOs of size 4 will be filled with the same methodology as discussed before with output samples of stage 2 that are now ready to start processing at stage 3.
2. The output samples number 2, 3 and 4 are fed back to FIFOs of size 16 where:
   2nd output is transmitted to FIFO of number 2 (16_2)
   3rd output is transmitted to FIFO of number 3 (16_3)
   4th output is transmitted to FIFO of number 4 (16_4)

   And this is repeated for the 16 iterations until the three FIFOs of size 16 are filled with output samples of stage 2.

We could observe that FIFO 16_1 is not mentioned that any samples will be fed back to it so does it will remain empty or there will be data samples shifted to it?

- Yes there will be samples shifted to it and this will be discussed in details in shifting methodology part.

STAGE 3:

After our 4 samples finish our processing flow:

1. Output sample number one is transmitted to FIFOs of size 1 and by applying the 4 iterations of stage 3(as first iteration per loop) the four FIFOs of size 1 will be filled with the same methodology as discussed before with output samples of stage 3 that are now ready to start processing at stage 4.
2. The output samples number 2, 3 and 4 are going to do feedback to FIFOs of size 4 where:
   2nd output is transmitted to FIFO of number 2 (4_2)
   3rd output is transmitted to FIFO of number 3 (4_3)
   4th output is transmitted to FIFO of number 4 (4_4)
   And this is repeated for the 4 iterations until the three FIFOs of size 4 are filled with output samples of stage 3.

We could observe that FIFO 4_1 is not mentioned that any samples will be fed back to it so does it will remain empty or there will be data samples shifted to it?

- Yes there will be samples shifted to it and this will be discussed in details in shifting methodology part.

STAGE 4:

After our 4 samples finish our processing flow now these samples are our output of the FFT operation and each sample will be extracted with its address and stored in external memory to be ready for next post FFT operations.

Note: During Stages 2, 3, 4 the architecture is able to receive the data samples of new symbol or next symbol.

4.5.5 Architecture Shifting Methodology between FIFOs

In this section we will explain shifting methodology between FIFOs in one loop only as an example since that will be repeated in other loops except the 4th loop (last loop)

N.B:

Shifting from which FIFO of size 64 depends on which loop we are in and that is clarified in the next table 13.

Table 13: Shifting source in each loop

1st loop	Shifting from number 1 of FIFOs of size 64 (64_1)
2nd loop	Shifting from number 2 of FIFOs of size 64 (64_2)
3rd loop	Shifting from number 3 of FIFOs of size 64 (64_3)

- During Stage 1:
    - There is no shifting between FIFOs.

- During Stage 2:
    - Shifting from one of FIFOs of size 64 >>> 16_1, so now 16_1 will be full ready for next iterationof stage 2 (16 shifts).

- During Stage 3:

  <u>Within 1st four iterations:</u>

    - Shifting from FIFO 16_2 >>> 4_1 (4 shifts) therefore 4_1 will be ready for next iteration of stage 3.
    - Shifting from one of FIFOs of size 64 >>>16_2 (4 shifts), therefore 16_2 now has 4 samples only ready for next iteration of stage 2.

- During Stage 4:

  <u>Before 1st iteration:</u>

    - Has no shifting between FIFOs since the four FIFOs of stage 4 will be filled by output of stage 3.

  <u>Before 2nd iteration:</u>

    - Shifting from FIFO 4_2 >>> to the 4 FIFOs of size 1. Therefore 4_2 will be empty.
    - Shifting from FIFO 16_2>>> to 4_2, (4 shifts) therefore 4_2 will be ready for next iteration of stage 3.
    - Shifting from one of FIFOs of size 64 >>>16_2, therefore 16_2 has 8 samples now ready for next iteration of stage 2.

  <u>Before 3rd iteration:</u>

    - Shifting from FIFO 4_3>>> to the 4 FIFOs of size 1. Therefore 4_3 will be empty.
    - Shifting from FIFO 16_2>>> to 4_3, (4 shifts) therefore 4_3 will be ready for next iteration of stage 3.
    - Shifting from one of FIFOs of size 64 >>> 16_2, therefore 16_2 has 12 samples now ready for next iteration of stage 2.

Before 4th iteration:

- Shifting from FIFO 4_4>>>to the 4 FIFOs of size 1. therefore 4_4 will be empty
- Shifting from FIFO 16_2>>> to 4_4, (4 shifts) (ready for next iteration of stage 3)
- Shifting from one of FIFOs of size 64 >>> 16_2, therefore 16_2 has 16 samples now (ready for next iteration of stage 2)

- During Stage 3:

Within 2nd four iterations:

- Shifting from FIFO 16_3 >>> 4_1 (4 shifts) therefore 4_1 will be ready for next iteration of stage 3.
- Shifting from one of FIFOs of size 64 >>>16_3 (4 shifts), therefore 16_3 now has 4 samples only ready for next iteration of stage 2.

- During Stage 4:

Before 1st iteration:

- Has no shifting between FIFOs since the four FIFOs of stage 4 will be filled by output of stage 3.

Before 2nd iteration:

- Shifting from FIFO 4_2 >>> to the 4 FIFOs of size 1. Therefore 4_2 will be empty.
- Shifting from FIFO 16_3>>> to 4_2, (4 shifts) therefore 4_2 will be ready for next iteration of stage 3.
- Shifting from one of FIFOs of size 64 >>> 16_3, therefore 16_3 has 8 samples now ready for next iteration of stage 2.

Before 3rd iteration:

- Shifting from FIFO 4_3>>> to the 4 FIFOs of size 1. Therefore 4_3 will be empty.
- Shifting from FIFO 16_3>>> to 4_3, (4 shifts) therefore 4_3 will be ready for next iteration of stage 3.
- Shifting from one of FIFOs of size 64 >>> 16_3, therefore 16_3 has 12 samples now ready for next iteration of stage 2.

Before 4th iteration:

- Shifting from FIFO 4_4>>> to the 4 FIFOs of size 1. therefore 4_4 will be empty
- Shifting from FIFO 16_3>>> to 4_4, (4 shifts) therefore 4_4 will be ready for next iteration of stage 3.
- Shifting from one of FIFOs of size 64 >>> 16_3 therefore 16_3 has 16 samples now (ready for next iteration of stage 2).

- During Stage 3:

  Within 3rd four iterations:

  - Shifting from FIFO 16_4 >>> 4_1 (4 shifts) therefore 4_1 will be ready for next iteration of stage 3.
  - Shifting from one of FIFOs of size 64 >>>16_4 (4 shifts), therefore 16_4 now has 4 samples only ready for next iteration of stage 2.

- During Stage 4:

  Before 1st iteration:

  - Has no shifting between FIFOs since the four FIFOs of stage 4 will be filled by output of stage 3.

  Before 2nd iteration:

  - Shifting from FIFO 4_2 >>> to the 4 FIFOs of size 1. Therefore 4_2 will be empty.
  - Shifting from FIFO 16_4>>> to 4_2, (4 shifts) therefore 4_2 will be ready for next iteration of stage 3.
  - Shifting from one of FIFOs of size 64 >>> 16_4, therefore 16_4 has 8 samples now ready for next iteration of stage 2.

  Before 3rd iteration:

  - Shifting from FIFO 4_3>>> to the 4 FIFOs of size 1. Therefore 4_3 will be empty.
  - Shifting from FIFO 16_4>>> to 4_3, (4 shifts) therefore 4_3 will be ready for next iteration of stage 3.
  - Shifting from one of FIFOs of size 64 >>> 16_4, therefore 16_4 has 12 samples now ready for next iteration of stage 2.

Before 4th iteration:

- Shifting from FIFO 4_4>>> to the 4 FIFOs of size 1. therefore 4_4 will be empty.
- Shifting from FIFO 16_4>>> to 4_4, (4 shifts) therefore 4_4 will be ready for next iteration of stage 3.
- Shifting from one of FIFOs of size 64 >>> 16_4 therefore 16_4 has 16 samples now (ready for next iteration of stage 2).

Notes:

- Now second iteration for stage 2 is ready to start processing in the second loop stages where the four FIFOs of size 16 are now shuffled and arranged with next samples that are the output of stage 1.
- The remaining four 4th iterations for stage 3 followed by iterations of stage 4 that will not have any shifting.

➔ N.B: in the 4th loop at the 16 4th iterations of stage 2 we won't need to shift from FIFOs of size 64 to FIFOs of size 16 since we have already shifted all output samples of stage 1 and FIFOs of size 64 now contain or shifted to it the new samples of next symbol.

4.6 Hardware Design & RTL

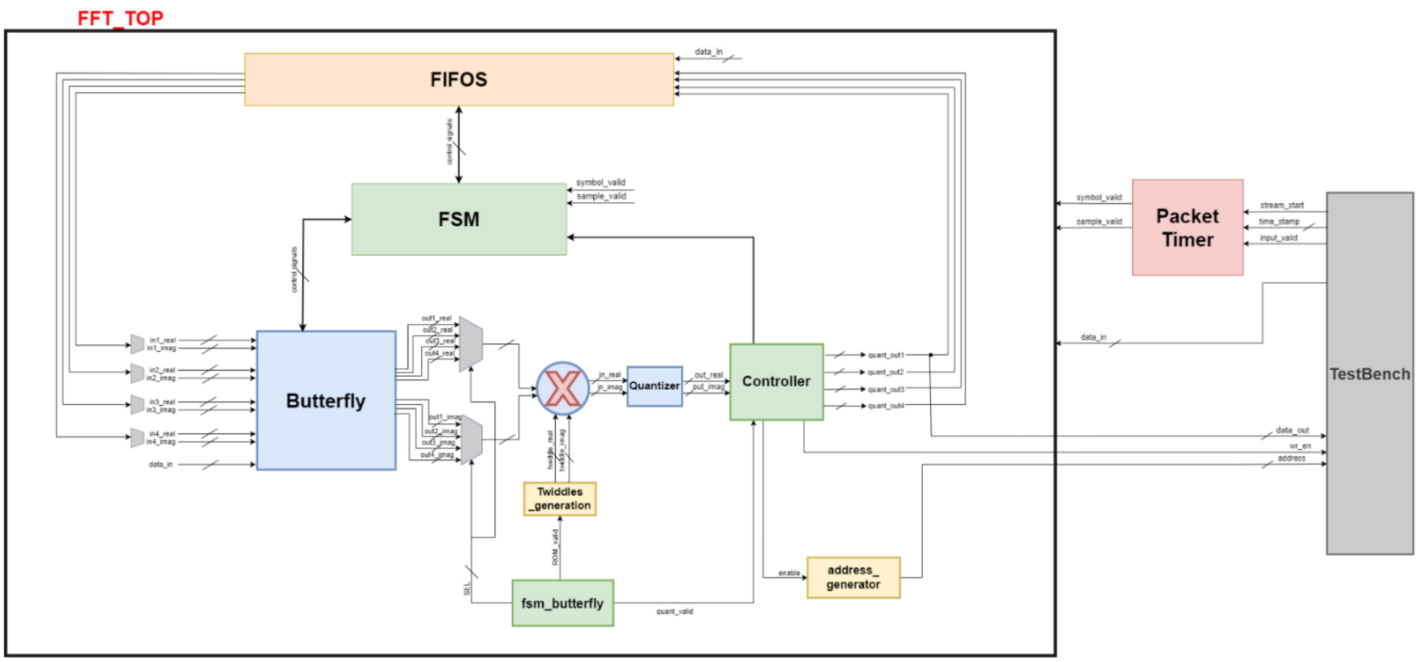


Figure 52: Block Diagram of FFT Sybsystem

4.6.1 Memory elements

A synchronous FIFO (first-in first-out) is where data values are written sequentially into a memory array using a clock signal, and the data values are sequentially read out from the memory array using the same clock signal. It's used in our architecture as the memory elements to store the input samples of the symbol and to shuffle the data between the stages.

As our architecture is radix-4, so in each iteration takes 4 inputs and gives 4 outputs. For stage 1, we take an input from each quarter of the symbol's samples. At first, we store three-quarters of the samples of a symbol in 3 FIFOs of size 64 elements. Each quarter is stored in a separate FIFO of size 64 elements. So, we can start processing when the last quarter of data has arrived.
For stage 2, the data is split into 4 quarters and then another 4 quarters so we need the data to be arranged in FIFOs of sizes 16 and so on for stage 3 and 4 as shown in figure 53. Therefore, the data is kept shifting between the stages. So, we need FIFOs of sizes 16, 4, and 1 elements.

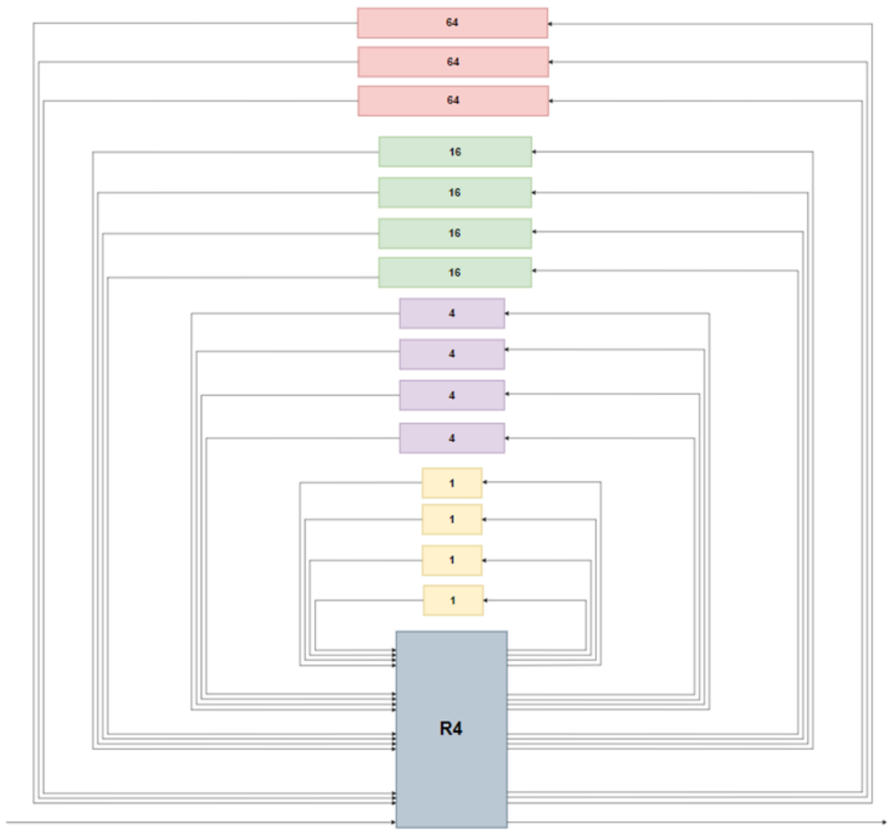


Figure 53: SDF architecture with memory elements

As shown in figure 53, there are 3 FIFOs of size 64, 4 FIFOs of size 16, 4 FIFOs of size 4, and 4 FIFOs of size 1.

There is a top module for each size of FIFOs contains multiple instantiation of the FIFO module as needed as stated above.

4.6.1.1 FIFO_64

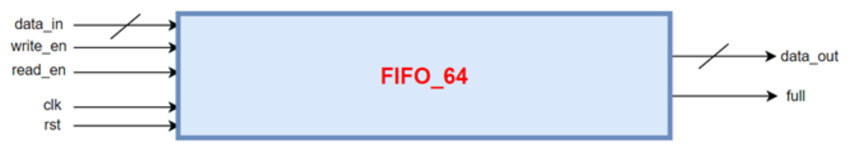


Figure 54: Block diagram of module of FIFO_64

A FIFO module contains clock, reset, write enable, read enable, and data in as inputs, data out and full flag as outputs.

The top module contains multiple instantiation of the FIFO module each is preceded by a 2x1 MUX as it receives data from 2 different sources as shown in figure 55.

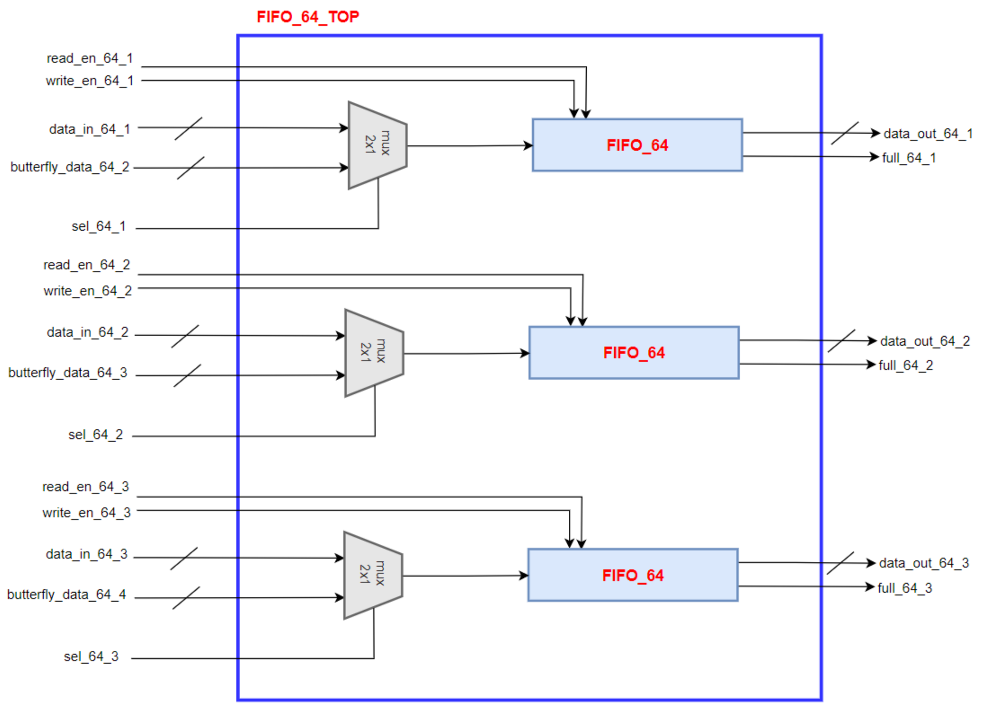


Figure 55: Block diagram of top module of FIFO_64

All ports of the top module of FIFO_64 are described in the following table 14.

Table 14: Ports description of top module of FIFO_64

Port	Direction	Width	Description
data_in_64_1	Input	24	Input data real and imaginary for first FIFO
data_in_64_2	Input	24	Input data real and imaginary for second FIFO
data_in_64_3	Input	24	Input data real and imaginary for third FIFO

Butterfly_data_64_2	Input	24	Input data coming from second output of quantizer
Butterfly_data_64_3	Input	24	Input data coming from third output of quantizer
Butterfly_data_64_4	Input	24	Input data coming from fourth output of quantizer
read_en_64_1	Input	1	Read enable signal to read from FIFO 64_1
read_en_64_2	Input	1	Read enable signal to read from FIFO 64_2
read_en_64_3	Input	1	Read enable signal to read from FIFO 64_3
write_en_64_1	Input	1	Write enable signal to write in first FIFO
write_en_64_2	Input	1	Write enable signal to write in second FIFO
write_en_64_3	Input	1	Write enable signal to write in third FIFO
sel_64_1	Input	1	Selection lines of mux of first FIFO
sel_64_2	Input	1	Selection lines of mux of second FIFO
sel_64_3	Input	1	Selection lines of mux of third FIFO
clk	Input	1	Clock pulses
rst	Input	1	Reset signal
data_out_64_1	Output	24	Output data from first FIFO
data_out_64_2	Output	24	Output data from second FIFO
data_out_64_3	Output	24	Output data from third FIFO
full_64_1	Output	1	Output signal when first FIFO is full
full_64_2	Output	1	Output signal when second FIFO is full
full_64_3	Output	1	Output signal when third FIFO is full

For the top modules of FIFO_16, FIFO_4, and FIFO_1, there are differences in the input data to each FIFO and described in the following figures (56, 57 and 58) and their ports in tables (15, 16, and 17).

4.6.1.2 FIFO_16

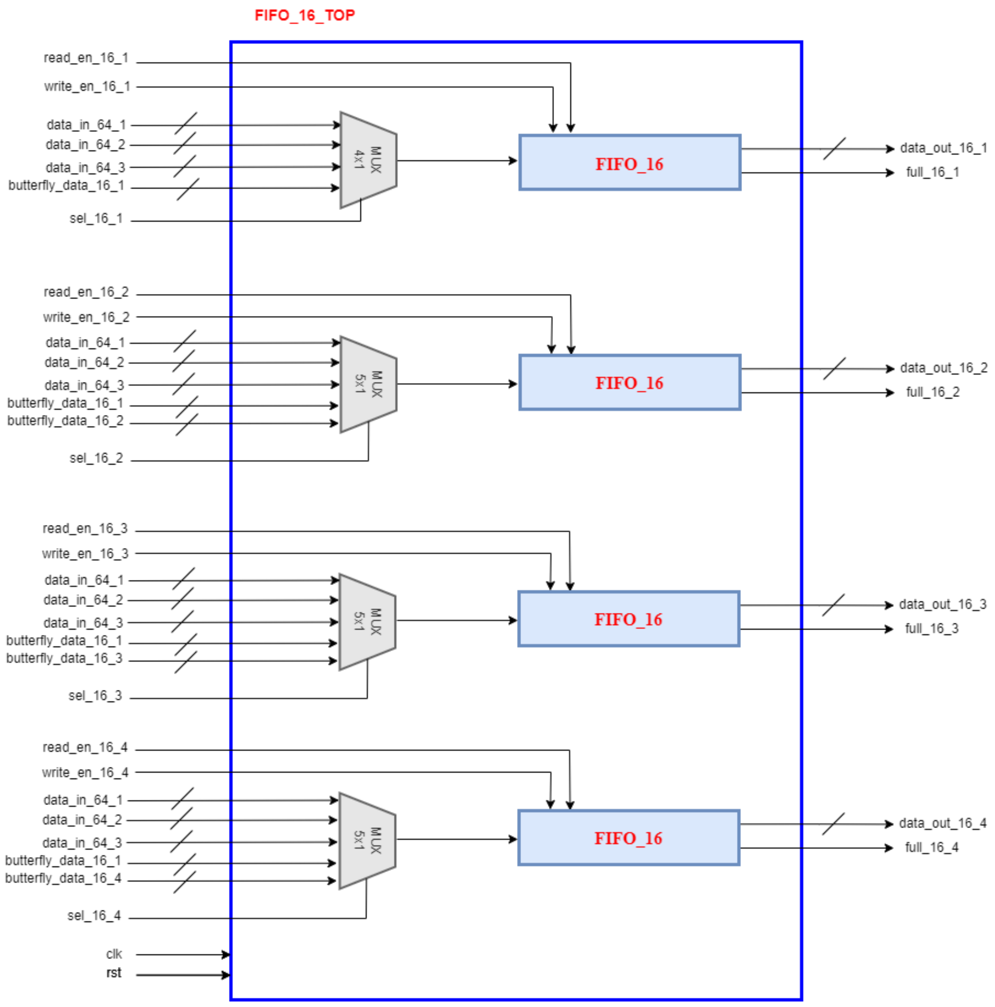


Figure 56: Block diagram of top module of FIFO_16

All ports of the top module of FIFO_16 are described in the following table 15

Table 15: Ports description of top module of FIFO_16

Port	Direction	Width	Description
data_in_64_1	Input	24	Input data real and imaginary for first FIFO
data_in_64_2	Input	24	Input data real and imaginary for second FIFO
data_in_64_3	Input	24	Input data real and imaginary for third FIFO
Butterfly_data_16_1	Input	24	Input data coming from first output of quantizer
Butterfly_data_16_2	Input	24	Input data coming from second output of quantizer
Butterfly_data_16_3	Input	24	Input data coming from third output of quantizer
Butterfly_data_16_4	Input	24	Input data coming from fourth output of quantizer
read_en_16_1	Input	1	Read enable signal to read from FIFO 16_1
read_en_16_2	Input	1	Read enable signal to read from FIFO 16_2
read_en_16_3	Input	1	Read enable signal to read from FIFO 16_3
read_en_16_4	Input	1	Read enable signal to read from FIFO 16_4
write_en_16_1	Input	1	Write enable signal to write in first FIFO
write_en_16_2	Input	1	Write enable signal to write in second FIFO
write_en_16_3	Input	1	Write enable signal to write in third FIFO
write_en_16_4	Input	1	Write enable signal to write in fourth FIFO
sel_16_1	Input	2	Selection lines of mux of first FIFO
sel_16_2	Input	3	Selection lines of mux of second FIFO
sel_16_3	Input	3	Selection lines of mux of third FIFO
sel_16_4	Input	3	Selection lines of mux of fourth FIFO
clk	Input	1	Clock pulses
rst	Input	1	Reset signal
data_out_16_1	Output	24	Output data from first FIFO
data_out_16_2	Output	24	Output data from second FIFO
data_out_16_3	Output	24	Output data from third FIFO
data_out_16_4	Output	24	Output data from fourth FIFO
full_16_1	Output	1	Output signal when first FIFO is full
full_16_2	Output	1	Output signal when second FIFO is full
full_16_3	Output	1	Output signal when third FIFO is full
full_16_4	Output	1	Output signal when fourth FIFO is full

4.6.1.3 FIFO_4

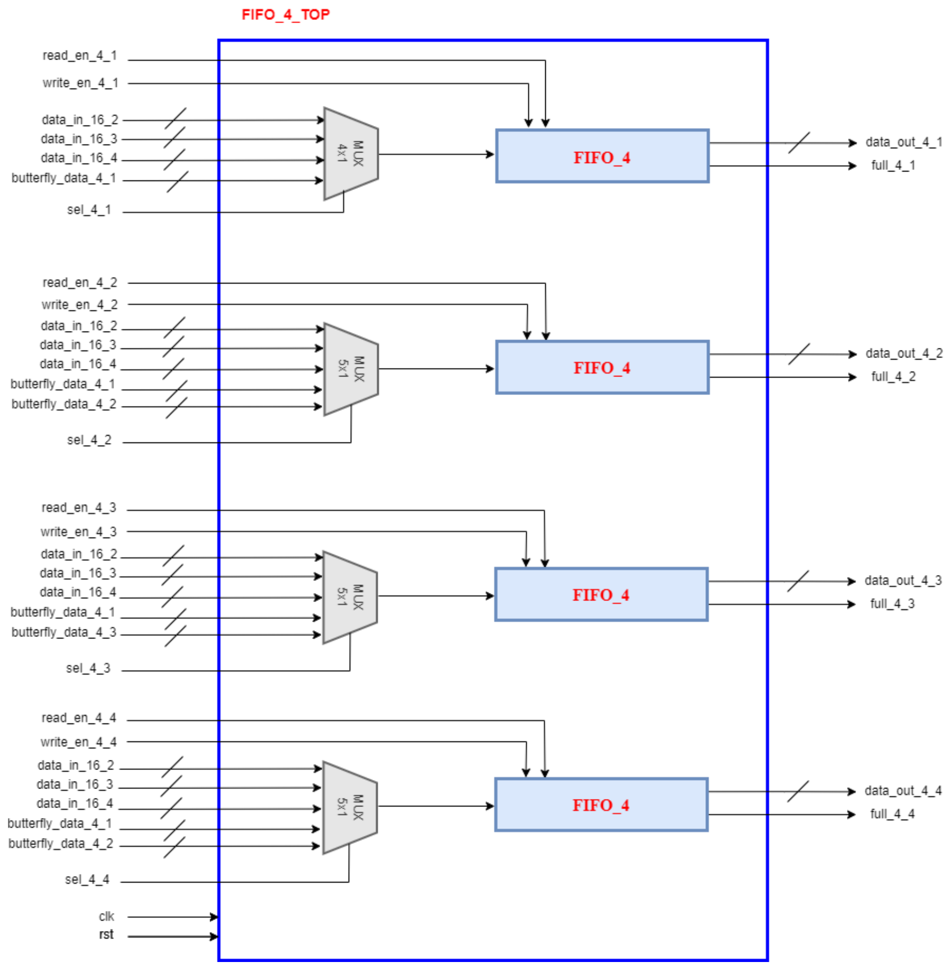


Figure 57: Block diagram of top module of FIFO_4

All ports of the top module of FIFO_4 are described in the following table 16

Table 16: Ports description of top module of FIFO_4

Port	Direction	Width	Description
data_in_16_2	Input	24	Input data real and imaginary for first FIFO from 16_2
data_in_16_3	Input	24	Input data real and imaginary for second FIFO from 16_3
data_in_16_4	Input	24	Input data real and imaginary for third FIFO from 16_4
Butterfly_data_4_1	Input	24	Input data coming from first output of quantizer
Butterfly_data_4_2	Input	24	Input data coming from second output of quantizer
Butterfly_data_4_3	Input	24	Input data coming from third output of quantizer
Butterfly_data_4_4	Input	24	Input data coming from fourth output of quantizer
read_en_4_1	Input	1	Read enable signal to read from FIFO 4_1
read_en_4_2	Input	1	Read enable signal to read from FIFO 4_2
read_en_4_3	Input	1	Read enable signal to read from FIFO 4_3
read_en_4_4	Input	1	Read enable signal to read from FIFO 4_4
write_en_4_1	Input	1	Write enable signal to write in first FIFO
write_en_4_2	Input	1	Write enable signal to write in second FIFO
write_en_4_3	Input	1	Write enable signal to write in third FIFO
write_en_4_4	Input	1	Write enable signal to write in fourth FIFO
sel_4_1	Input	2	Selection lines of mux of first FIFO
sel_4_2	Input	3	Selection lines of mux of second FIFO
sel_4_3	Input	3	Selection lines of mux of third FIFO
sel_4_4	Input	3	Selection lines of mux of fourth FIFO
clk	Input	1	Clock pulses
rst	Input	1	Reset signal
data_out_4_1	Output	24	Output data from first FIFO
data_out_4_2	Output	24	Output data from second FIFO
data_out_4_3	Output	24	Output data from third FIFO
data_out_4_4	Output	24	Output data from fourth FIFO
full_4_1	Output	1	Output signal when first FIFO is full
full_4_2	Output	1	Output signal when second FIFO is full
full_4_3	Output	1	Output signal when third FIFO is full
full_4_4	Output	1	Output signal when fourth FIFO is full

4.6.1.4 FIFO_1

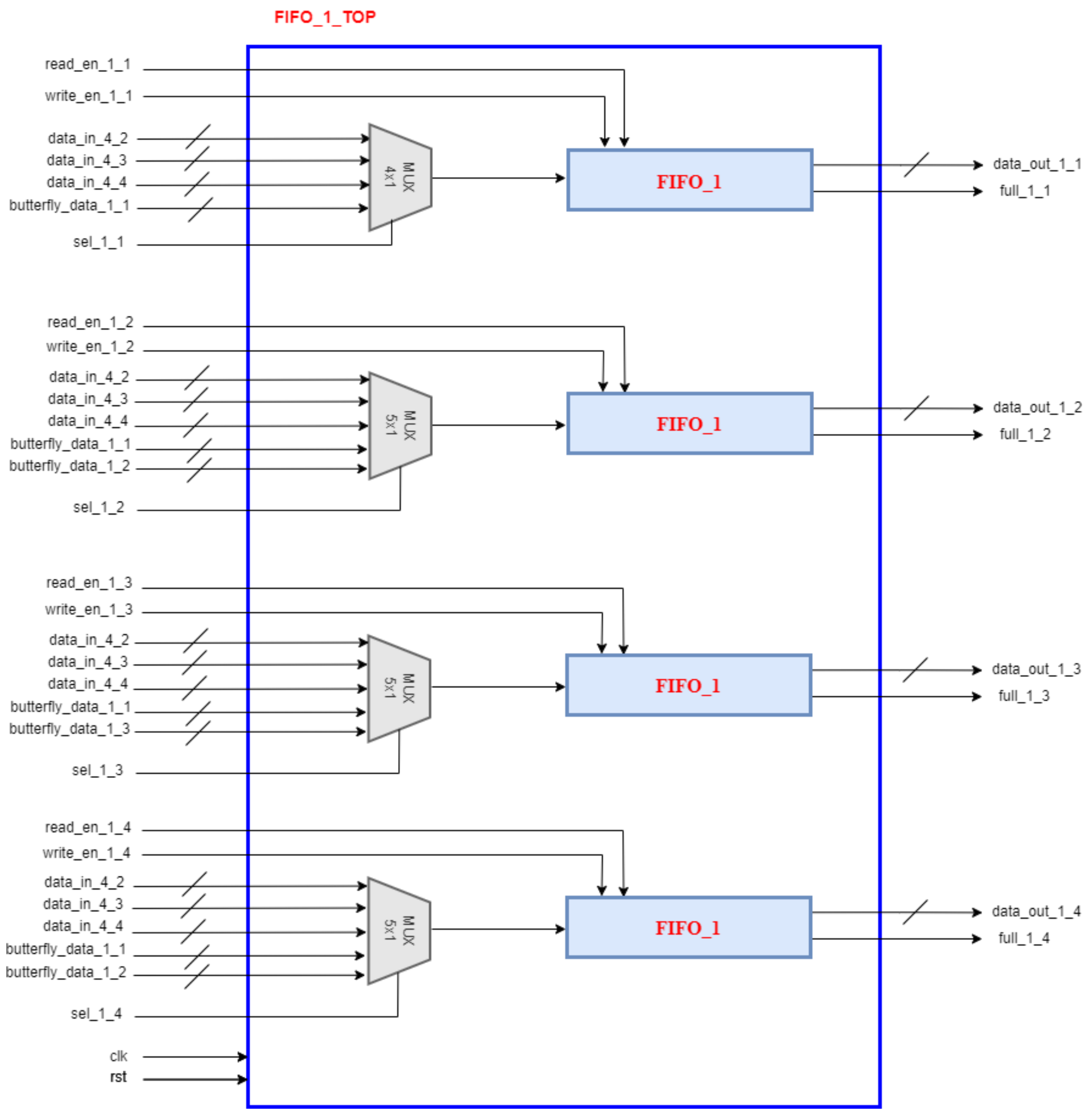


Figure 58: Block diagram of top module of FIFO_1

All ports of the top module of FIFO_1 are described in the following table 17.

Table 17: Ports description of top module of FIFO_1

Port	Direction	Width	Description
data_in_4_2	Input	24	Input data real and imaginary for first FIFO coming from 4_2
data_in_4_3	Input	24	Input data real and imaginary for second FIFO from 4_3
data_in_4_4	Input	24	Input data real and imaginary for third FIFO from 4_4
Butterfly_data_1_1	Input	24	Input data coming from first output of quantizer
Butterfly_data_1_2	Input	24	Input data coming from second output of quantizer
Butterfly_data_1_3	Input	24	Input data coming from third output of quantizer
Butterfly_data_1_4	Input	24	Input data coming from fourth output of quantizer
read_en_1_1	Input	1	Read enable signal to read from FIFO 1_1
read_en_1_2	Input	1	Read enable signal to read from FIFO 1_2
read_en_1_3	Input	1	Read enable signal to read from FIFO 1_3
read_en_1_4	Input	1	Read enable signal to read from FIFO 1_4
write_en_1_1	Input	1	Write enable signal to write in first FIFO
write_en_1_2	Input	1	Write enable signal to write in second FIFO
write_en_1_3	Input	1	Write enable signal to write in third FIFO
write_en_1_4	Input	1	Write enable signal to write in fourth FIFO
sel_1_1	Input	2	Selection lines of mux of first FIFO
sel_1_2	Input	3	Selection lines of mux of second FIFO
sel_1_3	Input	3	Selection lines of mux of third FIFO
sel_1_4	Input	3	Selection lines of mux of fourth FIFO
clk	Input	1	Clock pulses
rst	Input	1	Reset signal
data_out_1_1	Output	24	Output data from first FIFO
data_out_1_2	Output	24	Output data from second FIFO
data_out_1_3	Output	24	Output data from third FIFO
data_out_1_4	Output	24	Output data from fourth FIFO
full_1_1	Output	1	Output signal when first FIFO is full
full_1_2	Output	1	Output signal when second FIFO is full
full_1_3	Output	1	Output signal when third FIFO is full
full_1_4	Output	1	Output signal when fourth FIFO is full

4.6.2 Processing units

4.6.2.1 Butterfly

The butterfly of a radix-4 algorithm consists of four inputs and four outputs as shown in figure 59.

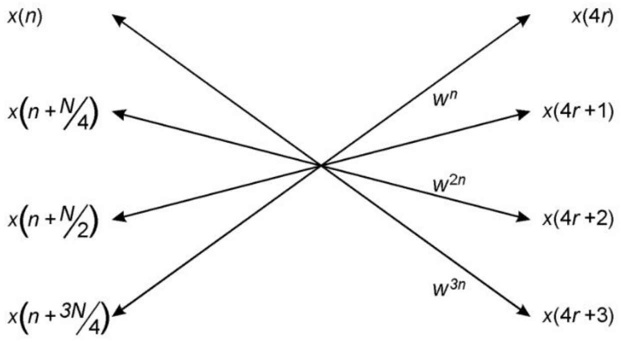


Figure 59: Radix-4 FFT butterfly

And the operation here is basically multiplying the 4 input by a matrix extracted from the equations:

$$\begin{bmatrix} x(n) \\ x\left(n+\frac{N}{4}\right) \\ x\left(n+\frac{N}{2}\right) \\ x\left(n+\frac{3N}{4}\right) \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} = \begin{bmatrix} x(4r) \\ x(4r+1) \\ x(4r+2) \\ x(4r+3) \end{bmatrix}$$

And this equation is translated and optimized to equations in the RTL as it's supposed to be as in the left, then by optimization as in the right as following:

```
assign temp1_real = in1_real +
in2_real + in3_real + in4_real;
assign temp1_imag = in1_imag +
in2_imag + in3_imag + in4_imag;

assign temp2_real = in1_real +
in2_imag - in3_real - in4_imag;
assign temp2_imag = in1_imag -
in2_real - in3_imag + in4_real;

assign temp3_real = in1_real -
in2_real + in3_real - in4_real;
assign temp3_imag = in1_imag -
in2_imag + in3_imag - in4_imag;

assign temp4_real = in1_real -
in2_imag - in3_real + in4_imag;
assign temp4_imag = in1_imag +
in2_real - in3_imag - in4_real;
```

```
assign a = in1_real + in3_real ;
assign b = in1_real - in3_real ;

assign c = in1_imag + in3_imag ;
assign d = in1_imag - in3_imag ;

assign e = in2_real + in4_real ;
assign f = in2_real - in4_real ;

assign g = in2_imag + in4_imag ;
assign h = in2_imag - in4_imag ;

assign temp1_real = a + e ;
assign temp1_imag = c + g ;

assign temp2_real = b + h ;
assign temp2_imag = d - f ;

assign temp3_real = a - e ;
assign temp3_imag = c - g ;

assign temp4_real = b - h ;
assign temp4_imag = d + f ;
```

In the hardware, it's represented in figure 60 as follows:

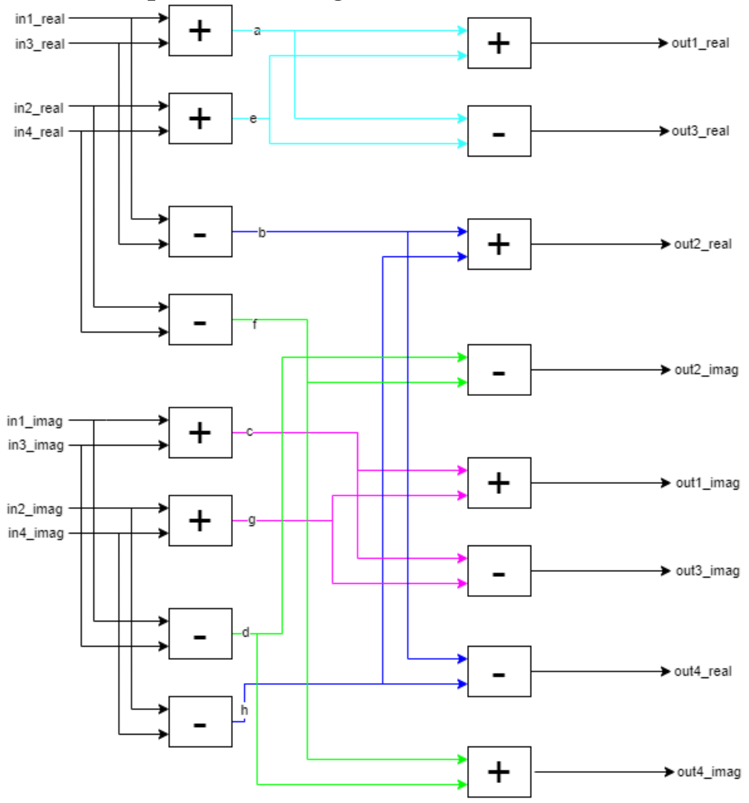


Figure 60: Design of the butterfly

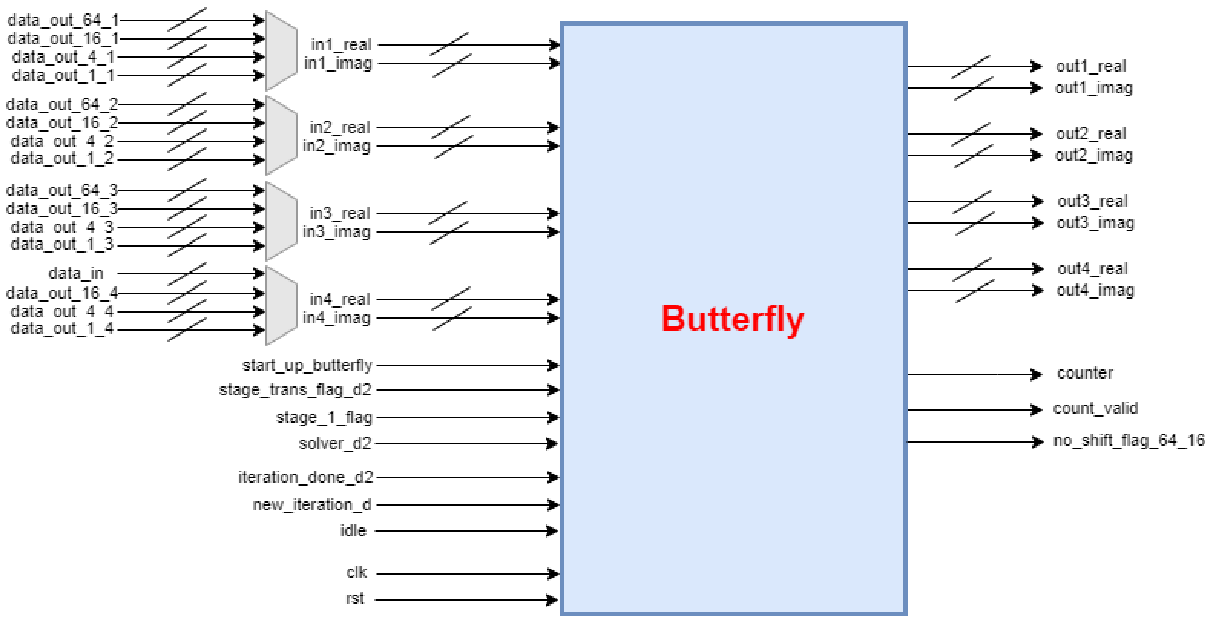


Figure 61: Block diagram of module butterfly

The ports are described in the following table 18:

Table 18: Ports description of butterfly

Port	Direction	Width	Description
clk	Input	1	Clock pulse
rst	Input	1	Reset signal
in1_real	Input	12	First input real part
in1_imag	Input	12	First input imaginary part
in2_real	Input	12	Second input real part
in2_imag	Input	12	Second input imaginary part
in3_real	Input	12	Third input real part
in3_imag	Input	12	Third input imaginary part
in4_real	Input	12	Fourth input real part
in4_imag	Input	12	Fourth input imaginary part
start_up_butterfly_flag	Input	1	Flag from FSM to enable butterfly at first time
stage_trans_flag_d2	Input	1	Flag to enable butterfly the first time at each stage 2,3,4
stage_1_flag	Input	1	Flag that we are processing in stage 1
solver_d2	Input	1	Flag used to delay stage transition in last operation
iteration_done_d2	Input	1	Delayed signal by 2 cycles from iteration_done (used to synchronize between last of any stage and start of new stage)
new_iteration_d	Input	1	Delayed Valid signal from controller to let butterfly to takeinput to do an operation
idle	input	1	Used to stop butterfly in idle state
Out1_real	Output	14	First output real part
Out1_imag	Output	14	First output imaginary part
Out2_real	Output	14	Second output real part
Out2_imag	Output	14	Second output imaginary part
Out3_real	Output	14	Third output real part
Out3_imag	Output	14	Third output imaginary part
Out4_real	Output	14	Fourth output real part
Out4_imag	Output	14	Fourth output imaginary part
counter	Output	7	Counter that counts number of butterfly iterations and will be transmitted to FSM to be a condition intransition between states
count_valid	Output	1	It is a flag that indicates that we finished processing of a symbol and the next symbol is ready to process it
no_shift_flag_64_16	Output	1	It is a flag used to stop shifting from 64 FIFOs to 16 FIFOs in last iteration of stage 2 in a symbol (last loop)

Note: the counter in this module is used to count the number of iterations executed on it. As each stage has 64 iterations, it counts stage 1 iterations from 1 → 64. Then it starts the loops of stage 2, 3, and 4. So, it counts:

- From 65 → 80 for stage 2 iterations.
- From 81 → 84 for stage 3 iterations.
- From 85 → 88 for stage 4 iterations.
- From 89 → 92 for stage 3 iterations.
- From 93 → 96 for stage 4 iterations.
- From 97 → 100 for stage 3 iterations.
- From 101 → 104 for stage 4 iterations.
- From 105 → 108 for stage 3 iterations.
- From 109 → 112 for stage 4 iterations.
- Then it is reset to the value 64 again to enter the following loop, and this loop is repeated 4 times.

Since in each loop Stage 2, Stage 3 and Stage 4 run 16 iterations each and the loop is repeated 4 times, hence each Stage ran 64 iterations.

The inputs of the butterfly module are the outputs of MUXs as they depend on which stage we are in. If we are in:

1. Stage 1, the input is coming from FIFOs_64 and the external input.
2. Stage 2, the input is coming from FIFOs_16.
3. Stage 3, the input is coming from FIFOs_4.
4. Stage 4, the input is coming from FIFOs_1.

And the selection lines are in the following table 19:

Table 19: Selection lines of MUXs of butterfly inputs

	Stage 1	**Stage 2**	**Stage 3**	**Stage 4**
	64_1	**16_1**	**4_1**	**1_1**
Sel_butterfly_1	00	01	10	11
	64_2	**16_2**	**4_2**	**1_2**
Sel_butterfly_2	00	01	10	11
	64_3	**16_3**	**4_3**	**1_3**
Sel_butterfly_3	00	01	10	11
	data_in	**16_4**	**4_4**	**1_4**
Sel_butterfly_4	00	01	10	11

4.6.2.2 Multiplier

A complex multiplier is used as a processing unit in the architecture. It takes the output of the butterfly and the twiddle factor as inputs and multiply them to produce the output. The complex multiplier is preceded by a MUXs to order the outputs of the butterfly. As the butterfly produces 4 outputs at a time and the multiplier should take each of them each clock cycle.

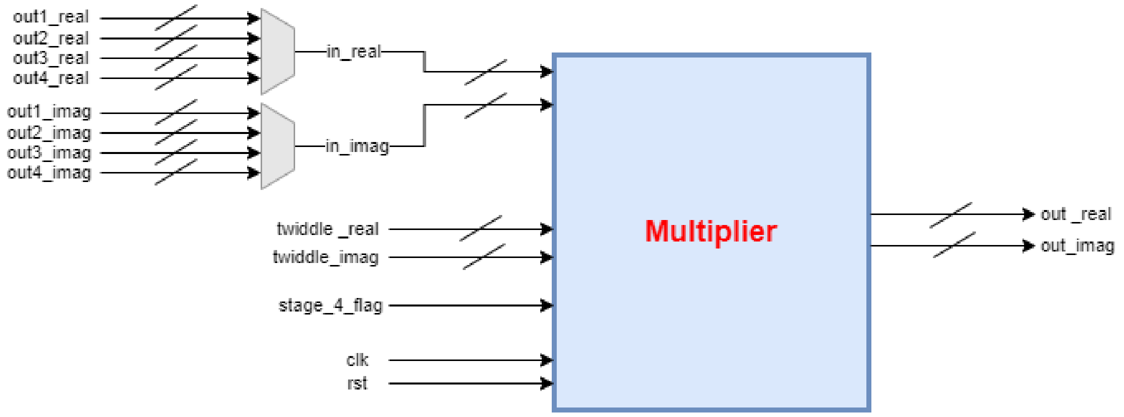


Figure 62: Block diagram of module multiplier

The complex multiplier is implemented using 3 multipliers and 5 adders as shown in figure 63

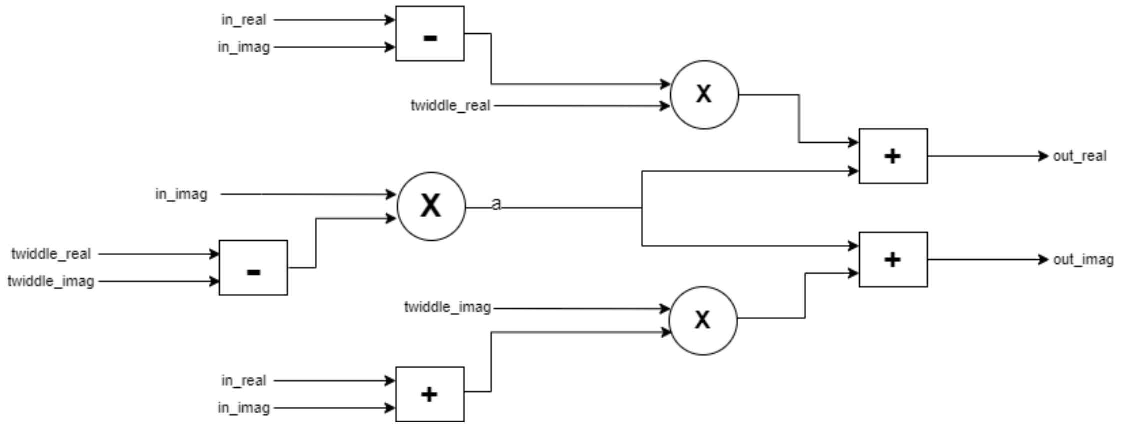


Figure 63: Design of the multiplier

The ports of this module are described in the following table 20.

Table 20: Ports description of multiplier

Port	Direction	Width	Description
clk	Input	1	Clock signal
rst	Input	1	Reset signal
in_real	Input	14	Input real part
in_imag	Input	14	Input imaginary part
twiddle_real	Input	9	Twiddle real part
twiddle_imag	Input	9	Twiddle imaginary part

Stage_4_flag	Input	1	Flag for stage 4
out_real	Output	23	Output real part
out_imag	Output	23	Output imaginary part

4.6.2.3 Quantizer

After the multiplier, the output is 23 bits, but it's needed to be fixed at 12 bits after each stage. So, the need of a block to quantize the output arises. The quantizer module is doing the same function as the function used in the MATLAB model. It rounds up the output and saturates it if it is out of the chosen range (greater or smaller than it). In our case, the output is S.11 and there are no integer bits.

- Example on rounding:
  If the input is 0000.11011010010(1)0111000,
  The quantizer throws the 3 integer bits and the least 7 bits then decide on the bit between the brackets if the number will be rounded up or not.
  If it's 1, then the output will be 0.11011010010 + 1 = 0.11011010011
  If it's 0, then the least 8 bits are truncated and the output will be 0.11011010010
- Example on saturation:
  If there's any input to the quantizer has integer bits (the value is greater than 1), it's saturated at the value 0.99999 → 0.11111111111.

Note: in stage 4, the input is 14 bits since there is no twiddle multiplication. So in this case, all we do is just rounding the input as explained above. Therefore, we use stage_4_flag as input to differentiate between the two cases.

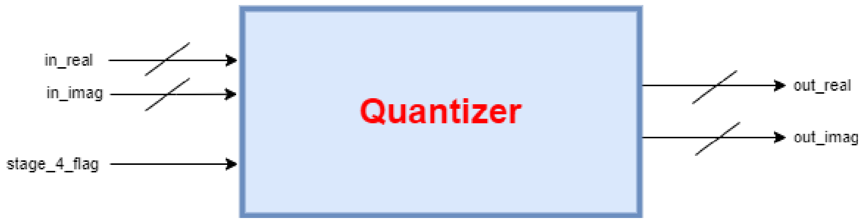


Figure 64: Block diagram of module quantizer

The ports of this module is described in the following table 21.

Table 21: Ports description of quantizer

Port	Direction	Width	Description
in_real	Input	23	Input real part
in_imag	Input	23	Input imaginary part
Stage_4_flag	Input	1	Flag for stage 4 as input length in this stage is different from other stages
out_real	Output	12	Output real part

Out_imag	Output	12	Output imaginary part

4.6.3 Control Units

4.6.3.1 FSM

It is a MEALY finite state machine which has 6 states (IDLE, STORING, STAGE1, STAGE2, STAGE3, STAGE4). It is the main control unit that sends the control signals to all blocks in the FFT subsystem. This FSM controls the write and read enables of all FIFOs, selection lines of inputs of all FIFOs, selection lines of inputs of butterfly and a flag for each stage.
It is working as follows:

- The system is in IDLE state until a symbol_valid signal is received, then a transition to STORING state is occurred.
- In STORING state, 3 quarters of data samples is stored in the 3 FIFOs of size 64 where the first 64 samples are stored in FIFO_64_1 until a full_64_1 flag is raised, then the second 64 samples are stored in FIFO_64_2 until a full_64_2 flag is raised, then the third 64 samples are stored in FIFO_64_3 until a full_64_3 flag is raised. So, a transition to STAGE1 state occurs.
- With every sample_valid signal, reading the data from FIFOs is occurred and processing of stage 1 is starting. And wait for a signal sent by the controller (iteration_done) after each iteration is done to feedback the outputs to the FIFOs. And this is repeated for the 64 iterations of stage 1. So, when the counter of the butterfly reaches 64, A transition to STAGE2 state occurs.
- As the controller sends a signal (new_iteration), read enables of FIFOs of size 16 is raised in order to start a new iteration. And wait for a signal (iteration_done) after each iteration is done to feedback the outputs to the FIFOs. As discussed in section 4.5.2, stage 2 has 16 iterations per loop. Then transition to STAGE3 is occurred when butterfly counter reaches 80.
- For STAGE3 state, the same idea for reading from and writing to the FIFOs but the difference is that the FIFOs of size 4 and 1. As discussed in section 4.5.2, stage 3 has 16 iterations per loop split into 4 groups where STAGE3 and STAGE4 alternate 4 times per loop. Then a transition to STAGE4 occurs when the butterfly counter reaches 84 or 92 or 100 or 108.
- For STAGE4 state, the same idea for reading from the FIFOs but the difference is that the FIFOs of size 1 and produce the FFT output. As discussed in section 4.5.2, stage 4 has 16 iterations per loop split into 4 groups where STAGE3 and STAGE4 alternates 4 times per loop. Then a transition to STAGE3 occurs when butterfly counter reaches 88 or 96 or 104.
- But when the counter reaches 112, it is reset to 65 as mentioned in the butterfly section 4.5.2 and transition to STAGE2 occurs and the loop is repeated.
- And after the last iteration of the loop is finished and the symbol is done, transition to STORING state occurs to store the remaining samples of the next symbol.

The communication between the FSM and the other blocks, where the inputs row is the inputs to the block from the FSM and the outputs row is the outputs from the block to the FSM. Is shown in the coming table

Table 22: Connection signals between FSM and other blocks

FIFO_64 TOP	**Inputs**	write_enable_64_1 write_enable_64_2 write_enable_64_3	read_enable_64_1 read_enable_64_2 read_enable_64_3	sel_64_1 sel_64_2 sel_64_3	—
	Outputs	full_64_1 full_64_2 full_64_3	—	—	—
FIFO_16 TOP	**Inputs**	write_enable_16_1 write_enable_16_2 write_enable_16_3 write_enable_16_4	read_enable_16_1 read_enable_16_2 read_enable_16_3 read_enable_16_4	sel_16_1 sel_16_2 sel_16_3 sel_16_4	—
	Outputs	full_16_1 full_16_2 full_16_3 full_16_4	—	—	—
FIFO_4 TOP	**Inputs**	write_enable_4_1 write_enable_4_2 write_enable_4_3 write_enable_4_4	read_enable_4_1 read_enable_4_2 read_enable_4_3 read_enable_4_4	sel_4_1 sel_4_2 sel_4_3 sel_4_4	—
	Outputs	full_4_1 full_4_2 full_4_3 full_4_4	—	—	—
FIFO_1 TOP	**Inputs**	write_enable_1_1 write_enable_1_2 write_enable_1_3 write_enable_1_4	read_enable_1_1 read_enable_1_2 read_enable_1_3 read_enable_1_4	sel_1_1 sel_1_2 sel_1_3 sel_1_4	—
	Outputs	full_1_1 full_1_2 full_1_3 full_1_4	—	—	—
quant_controller	**Inputs**	stage_4_flag	—	—	—
	Outputs	iteration_done iteration_done_d1	new_iteration new_iteration_d	solver solver_d1	solver_2
butterfly	**Inputs**	start_up_butterfly_flag	stage_transition_flag_d2	stage_1_flag	idle
	Outputs	no_shift_flag_64_16	butterfly_counter	count_valid	—
multiplier	**Inputs**	stage_4_flag	—	—	—
	Outputs	—	—	—	—
fsm_butterfly	**Inputs**	start_up_butterfly_flag	stage_transition_flag_d2	stage_1_flag stage_4_flag	idle
	Outputs	—	—	—	—
	Inputs	stage_1_flag	stage_2_flag	stage_3_flag	stage_4_flag

twiddles_generation	**Outputs**	—	—	—	—
4 MUXs for inputs of butterfly	**Inputs**	sel_butterfly_MUX1	sel_butterfly_MUX2	sel_butterfly_MUX3	sel_butterfly_MUX4
	Outputs	—	—	—	—

The state diagram of the FSM is shown in the following diagram:

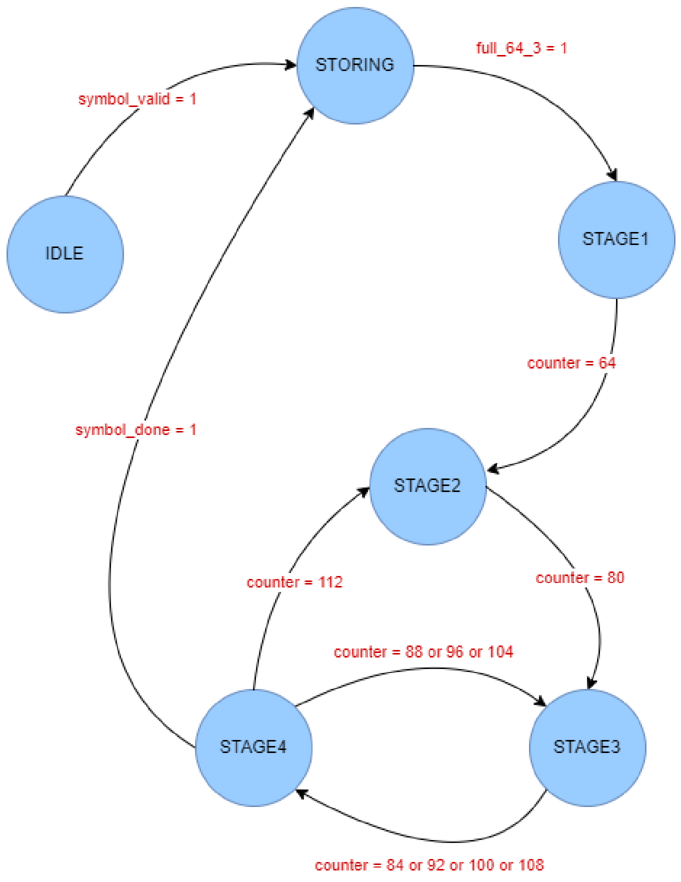


Figure 65: State diagram of FSM of butterfly

Table 23: State table of FSM

State	Description	Transition
IDLE	Idle state.	- IDLE → STORING if(symbol_valid)
STORING	Storing the input samples to the FIFOs of size 64 to be ready to start processing in stage 1.	- STORING → STAGE1 if(full_64_3)
STAGE1	Processing stage 1 iterations.	- STAGE1→ STAGE2 if(butterfly_counter == 64 && iteration_done)
STAGE2	Processing stage 2 iterations.	- STAGE2→ STAGE3 if(butterfly_counter == 80)
STAGE3	Processing stage 3 iterations.	- STAGE3→ STAGE4 if((butterfly_counter == 84 || 92 || 100 || 108) && iteration_done)
STAGE4	Processing stage 4 iterations.	- STAGE3→ STAGE4 if((butterfly_counter == 88 || 96 || 104) && iteration_done) - SAMPLE4 → IDLE if((butterfly_counter == 112) && count_valid)

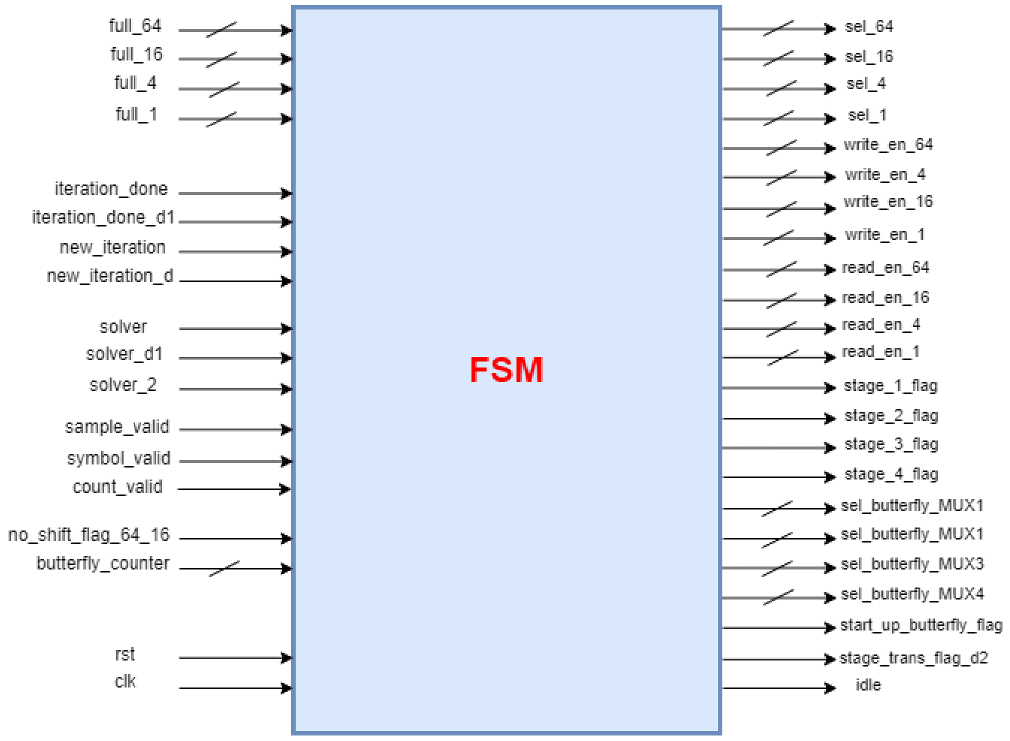


Figure 66: Block diagram of top module of FSM

Table 24: Ports description of FSM

Port	Direction	Width	Description
clk	Input	1	Clock signal

rst	Input	1	Reset signal
full_64_1 **full_64_2** **full_64_3**	Input	1	Signals coming from FIFOs_64 when they are full
full_16_1 **full_16_2** **full_16_3** **full_16_4**	Input	1	Signals coming from FIFOs_16 when they are full
full_4_1 **full_4_2** **full_4_3** **full_4_4**	Input	1	Signals coming from first fifo_4 when they are full
full_1_1 **full_1_2** **full_1_3** **full_1_4**	Input	1	Signals coming from first fifo_1 when they are full
iteration_done	Input	1	Coming from controller when data is ready to be written in FIFOs
iteration_done_d1	Input	1	Delayed version of iteration_done
new_iteration	Input	1	Coming from controller to let butterfly take input to do a new operation
new_iteration_d	Input	1	Delayed version of iteration_done
sample_valid	Input	1	A signal that comes with every input be written in 64_fifos sent by packet timers block
count_valid	Input	1	Coming from butterfly when processing on a symbol is finished.
Symbol_valid	Input	1	A signal that comes with new symbol sent by packet timers block.
butterfly_counter	Input	7	Coming from butterfly to identify how many times the butterfly was used to switch between states
no_shift_flag_64_16	Input	1	It is a flag used to stop shifting from 64 FIFOs to 16 FIFOs in last iteration of stage 2 in a symbol
solver	Input	1	Flag to ensure to write values in FIFOs before transition to another state
solver_d1	Input	1	Delayed version of solver
solver_2	Input	1	Flag to prevent reading from FIFOs at transition time between stages
sel_64_1 **sel_64_2** **sel_64_3**	Output	1	Selection lines of mux of FIFOs_64
sel_16_1 **sel_16_2** **sel_16_3** **sel_16_4**	Output	2 3 3 3	Selection lines of mux of FIFOs_16
sel_4_1 **sel_4_2** **sel_4_3** **sel_4_4**	Output	2 3 3 3	Selection lines of mux of FIFOs_4
sel_1_1	Output	2	Selection lines of mux of FIFOs_1

sel_1_2 **sel_1_3** **sel_1_4**		2 2 2	
write_en_64_1 **write_en_64_2** **write_en_64_3**	Output	1	Write enable signals to write data in FIFOs_64
write_en_16_1 **write_en_16_2** **write_en_16_3** **write_en_16_4**	Output	1	Write enable signals to write data in FIFOs_16
write_en_4_1 **write_en_4_2** **write_en_4_3** **write_en_4_4**	Output	1	Write enable signals to write data in FIFOs_4
write_en_1_1 **write_en_1_2** **write_en_1_3** **write_en_1_4**	Output	1	Write enable signals to write data in FIFOs_1
read_en_64_1 **read_en_64_2** **read_en_64_3** **read_en_64_4**	Output	1	Read enable signals to write data in FIFOs_64
read_en_16_1 **read_en_16_2** **read_en_16_3** **read_en_16_4**	Output	1	Read enable signals to write data in FIFOs_16
read_en_4_1 **read_en_4_2** **read_en_4_3** **read_en_4_4**	Output	1	Read enable signals to write data in FIFOs_4
read_en_1_1 **read_en_1_2** **read_en_1_3** **read_en_1_4**	Output	1	Read enable signals to write data in FIFOs_1
sel_butterfly_MUX1 **sel_butterfly_MUX2** **sel_butterfly_MUX3** **sel_butterfly_MUX4**	Output	2	Selection lines of mux of inputs of butterfly
start_up_butterfly_flag	Output	1	A signal given to the butterfly to start processing for the first time only
stage_1_flag	Output	1	A signal to butterfly that indicates that we are in stage 1
stage_2_flag	Output	1	A signal to butterfly that indicates that we are in stage 2
stage_3_flag	Output	1	A signal to butterfly that indicates that we are in stage 3
stage_4_flag	Output	1	A signal that indicates that we are in stage 4 to be passed in multiplier without multiplication
idle	Output	1	Used to stop butterfly in idle state
Stage_trans_flag_d2	Output	1	Used to start butterfly in the start of each state

4.6.3.2 Butterfly FSM

It's a MEALY finite state machine that controls the following in the processing units:

- Input of the multiplier
- Control the quantized samples
- Enable for ROM of twiddle factors

This FSM arranges the 4 outputs of each iteration such as they are all produced from the butterfly at one clock cycle. But they are needed to go through the multiplier each one at a time since there is only one multiplier. So, it controls the two selection lines of MUXs (1 for the real part and the other for the imaginary part) at the input of the multiplier. It also controls the enable of the ROM of twiddle factors (twiddle_generation block) in order to extract the twiddle factor of each sample. It also sends a quant_valid signal to the controller to control the quantized samples. As shown in the following diagram.

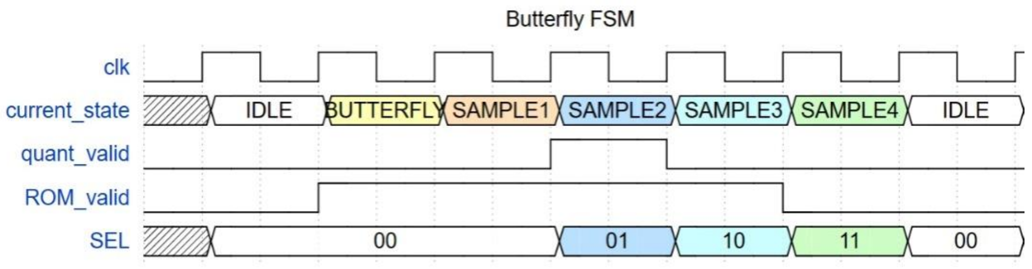


Figure 67: Timing diagram of butterfly_FSM

Where ROM_valid precedes by one clock cycle than SAMPLE1, SAMPLE2, SAMPLE3 and SAMPLE4 and it is raised in BUTTERFLY state in order to synchronize between the data sample and its twiddle factor as it takes one clock cycle to be extracted from Twiddles generation block.

Table 25: State table of fsm_butterfly

State	Description	Transition	Output

IDLE	Idle state.	If(start_up_butterfly_flag==1)
		IDLE → BUTTERFLY
			ROM_valid = 0
			quant_valid = 0
			SEL = 00

BUTTERFLY	Butterfly operation is executed.	BUTTERFLY → SAMPLE1	ROM_valid = 1
			quant_valid = 0
			SEL = 00

SAMPLE1	1st sample to go through the multiplier.	SAMPLE1 → SAMPLE2	ROM_valid = 1
			quant_valid = 0
			SEL = 00

SAMPLE2	2nd sample to go through the multiplier.	SAMPLE2 → SAMPLE3	ROM_valid = 1
			quant_valid = 1
			SEL = 01

SAMPLE3	3rd sample to go through the multiplier.	SAMPLE3 → SAMPLE4	ROM_valid = 1
			quant_valid = 0
			SEL = 10

SAMPLE4	4th sample to go through the multiplier.	SAMPLE4 → IDLE	ROM_valid = 0
			quant_valid = 0
			SEL = 11

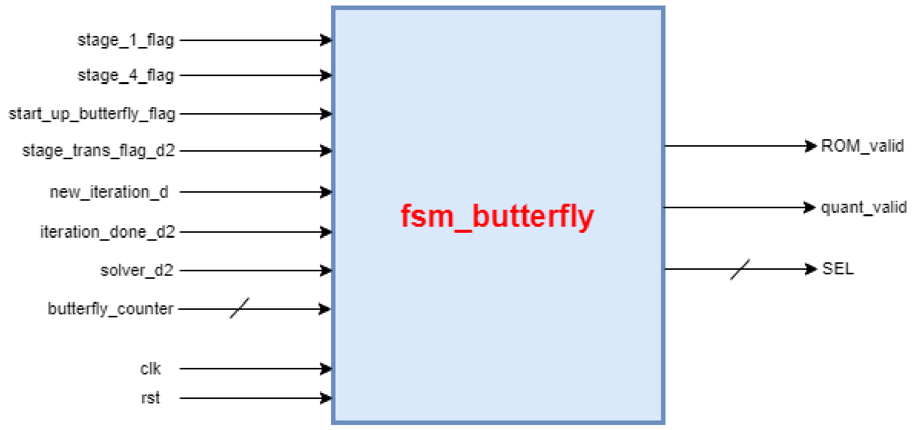


Figure 68: Block diagram of module of fsm_butterfly

The state diagram of the FSM is shown in the following diagram:

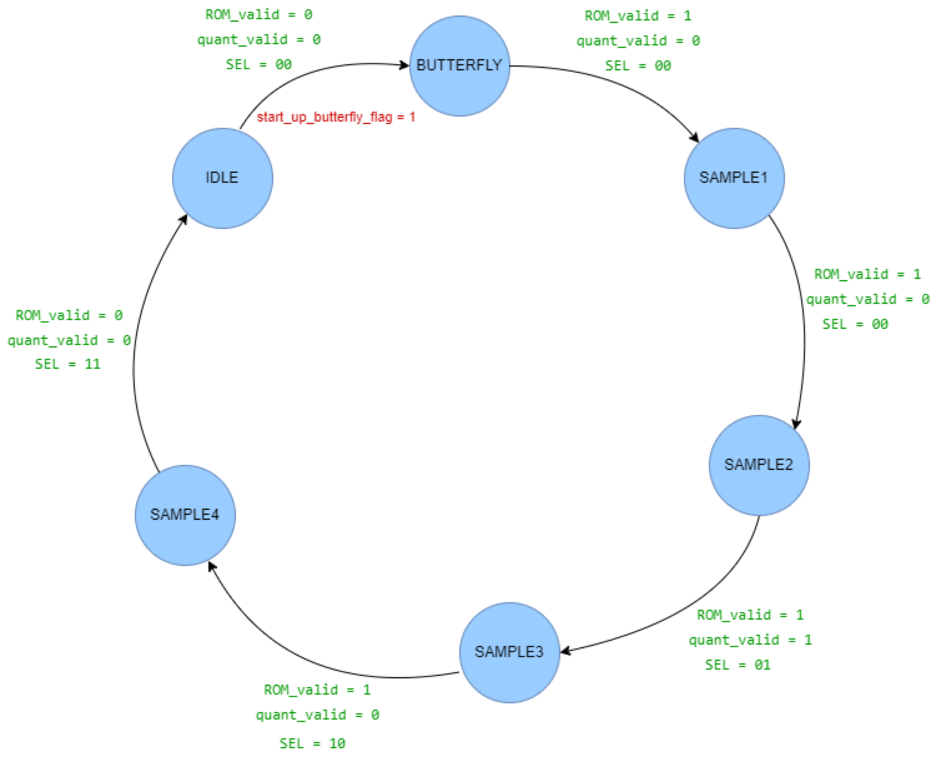


Figure 69: State diagram of FSM of butterfly

Table 26: Ports description of fsm_butterfly

Port	Direction	Width	Description
clk	Input	1	Clock signal
rst	Input	1	Reset signal
stage_1_flag	Input	1	Flag for stage 1
stage_4_flag	Input	1	Flag for stage 4
start_up_butterfly_flag	Input	1	Flag to enable the butterfly in stage 1
stage_trans_flag_d2	Input	1	Delayed version by 2 cycles of transition to enable the butterfly in the beginning of each stage
new_iteration_d	Input	1	Enable for the butterfly to start processing
iteration_done_d2	Input	1	Enable for the butterfly to start processing in the transition between the stages
solver_d2	Input	1	Used to enable the butterfly in the proper time (after transition)
idle	Input	1	Used to stop butterfly in idle state
butterfly_counter	Input	7	Butterfly counter used for transition between stages
ROM_valid	Output	1	ROM enable
quant_valid	Output	1	Quantizer enable
SEL	Output	2	Selection line for the mux (input of multiplier)

4.6.3.3 Controller

After the 4 outputs of the iteration are quantized in the quantizer module, they are fed back to the FIFOs or sent to the external memory as an output depending on which stage we are in.

- In stages 1, 2, and 3, the outputs are fed back to the FIFOs as explained in section 4.5.4. So, we need to register the 4 outputs one by one till the 4 outputs are ready then a signal (iteration_done) is sent to the main control unit to raise the write enables of the desired FIFOs.
- In stage 4, the outputs are sent to the external memory of the FFT. So, they are sent one by one and the write enable (stage_4_out) to the external memory is sent with the output.

In order to use the concept of pipelining, after the second output, a flag (new_iteration) is sent to read from the FIFOs to start a new iteration of the butterfly. This keeps pipelining between the butterfly, the multiplier and the quantizer.

And that is shown in the next time diagram.

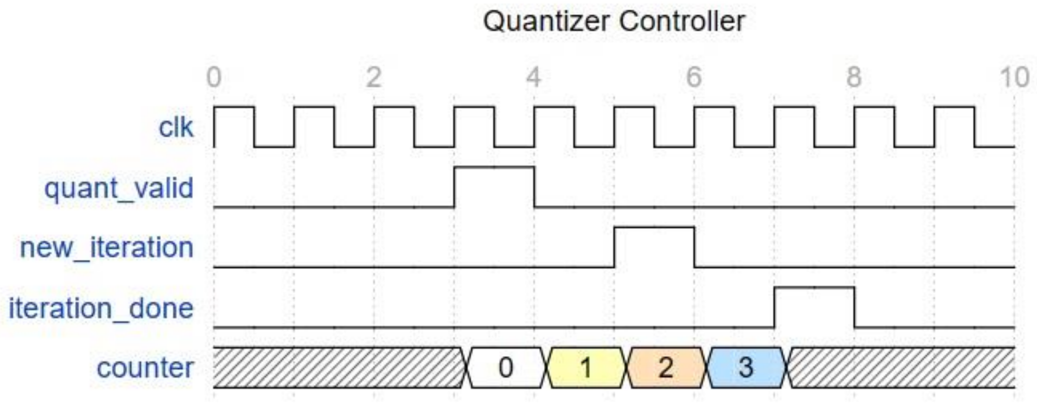


Figure 70: Timing diagram of quantizer_controller

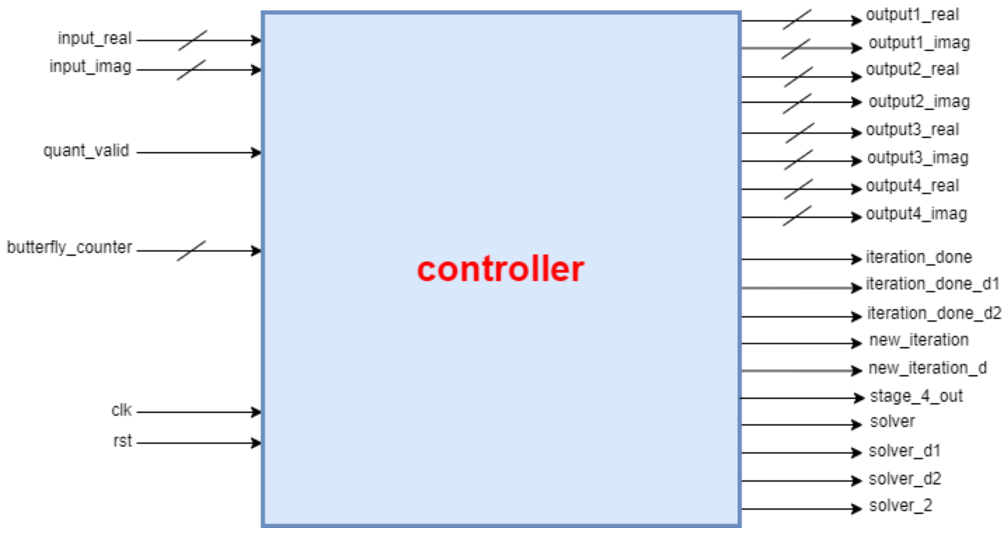


Figure 71: Block diagram of module of quantizer_controller

Table 27: Ports description of controller

Port	Direction	Width	Description
clk	Input	1	Clock pulses
rst	Input	1	Reset signal
out_real	Input	12	Input real part
out_imag	Input	12	Input imaginary part
quant_valid	Input	1	Flag from butterfly_fsm to enable quantizer
butterfly_counter	Input	7	Butterfly counter
output1_real	Output	12	1st output real part
output1_imag	Output	12	1st output imaginary part
output2_real	Output	12	2nd output real part
output2_imag	Output	12	2nd output imaginary part
output3_real	Output	12	3rd output real part
output3_imag	Output	12	3rd output imaginary part

output4_real	Output	12	4th output real part
output4_imag	Output	12	4th output imaginary part
iteration_done	Output	1	Output flag that quantizer has finished processing on 4samples and to enable write in FIFOs.
iteration_done_d1	Output	1	Delayed version of iteration_done
iteration_done_d2	Output	1	Iteration_done signal delayed by 2 clock cycles
new_iteration	Output	1	Output flag from quantizer to enable read from FIFOs (to pass new samples to butterfly)
new_iteration_d	Output	1	Delayed version of new_iteration used to enable elbutterfly to process on new samples
stage_4_out	Output	1	Signal used as the write enable to the external memory of the FFT
solver	Output	1	Flag to ensure to write values in FIFOs before transition to another state
solver_d1	Output	1	Delayed version of solver
solver_d2	Output	1	Delayed version of solver with 2 cycles
solver_2	Output	1	Flag to prevent reading from FIFOs at transition time between stages

4.6.4 Others

4.6.4.1 Twiddles Generation

Stage 1, 2 and 3, each has 256 twiddle factors. So, there are in total $3 * 256 = 768$ twiddle factors each of them represented in 18 bits, with 9 bits were assigned to the real part and 9 bits to the imaginary part. This is going to take a huge memory (huge area) if they are stored as they are. Since the twiddle factors basically consist of sine and cosine functions, the symmetry of these two functions can be used to optimize this memory.

In the case of the 256-point FFT radix-4, the twiddle factors exhibit a specific symmetry pattern. The twiddle factors can be grouped into four distinct sets, each containing 64 factors as shown in figure 72. Within each set, the factors are related by conjugation and cyclic permutations. Specifically, the twiddle factors in each set are obtained by raising a base factor to different powers.

There are only 64 twiddle factors stored in the ROM and the other twiddle factors are generated from these 64 using the symmetry. And there are 4 quadrants as shown in figure 72.

- 1st quadrant has the unique twiddle factors → W_{Q1}.
- 2nd quadrant has symmetry with 1st quadrant as any twiddle factor in this quadrant $W_{Q2} + W_{Q1} = \frac{N}{4} = \frac{256}{4} = 64$, so we can conclude that:

  $W_{Q2} = -W_{Q1}^*$

  $\rightarrow real(W_{Q2}) = -real(W_{Q1}),\ imag(W_{Q2}) = imag(W_{Q1}).$
- 3rd quadrant has symmetry with 1st quadrant as any twiddle factor in this quadrant $W_{Q3} + W_{Q1} = \frac{N}{2} = \frac{256}{2} = 128$, so we can conclude that:

  $W_{Q3} = -W_{Q1}.$

- 4th quadrant has symmetry with 1st quadrant as any twiddle factor in this quadrant $W_{Q4} + W_{Q1} = N = 256$, so we can conclude that $W_{Q4} = W_{Q1}^{*}$, although there are no twiddle factors in the 4th quadrant in our case.

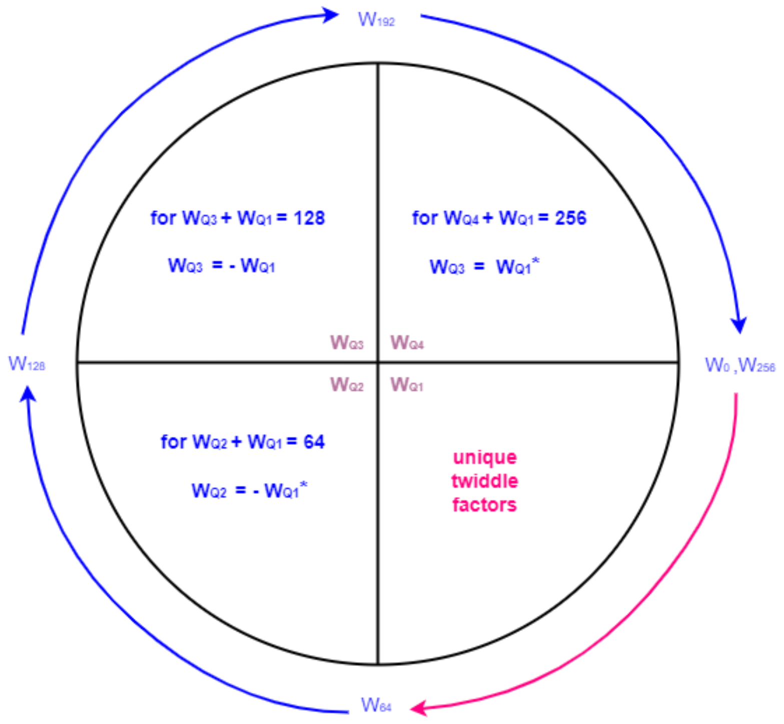


Figure 72: Twiddle factor symmetry

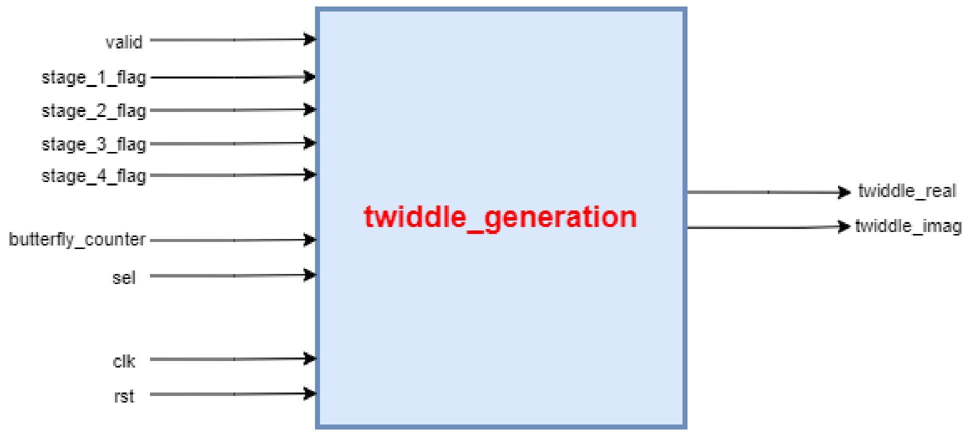


Figure 73: Block diagram of module twiddle_generation

The inputs to this module are:

- Valid signal works as enable to the ROM.
- 4 flags indicating in which stage we are.
- Selection line of the MUX that is before the multiplier to indicate in which sample of the 4 it is.

- Butterfly counter to specify in which iteration of the stage we are in.

Table 28: Ports description of twiddle_generation

Port	Direction	Width	Description
clk	Input	1	Clock signal
rst	Input	1	Reset signal
valid	Input	1	ROM enable
Stage_1_flag	Input	1	Flag for stage 1
Stage_2_flag	Input	1	Flag for stage 2
Stage_3_flag	Input	1	Flag for stage 3
Stage_4_flag	Input	1	Flag for stage 4
sel	Input	2	Signal to indicate in which sample of the four samples we are
Butterfly_counter	Input	7	Butterfly counter is used to indicate in which loop of iterations we are
twiddle_real	Outp ut	9	Twiddle real part
twiddle_imag	Outp ut	9	Twiddle imaginary part

4.6.4.2 Address generator

In this module, the address in which the output data to be written in the memory is generated. Since the output is written in the memory in a specific sequence, so this sequence is generated here.

- For the 4 outputs of each iteration , each one of them are in a quarter of the 4 quarters of locations ($0 \rightarrow 63, 64 \rightarrow 127, 128 \rightarrow 191, 192 \rightarrow 255$):
  - 1st output is at address x,
  - 2nd one is at $x + 64$,
  - 3rd one is at $x + 128$,
  - 4th one is at $x + 192$
- Let's call each 4 consequent iterations → group. Inside each group, there's an increase in the address by 16
  - 1st iteration is at address y,
  - 2nd one is at $y + 16$,
  - 3rd one is at $y + 32$,
  - 4th one is at $y + 48$
- Between each 4 consequent groups, there's an increase in the address by 4
  - 1st group is at address z,
  - 2nd one is at $z + 4$,
  - 3rd one is at $z + 8$,
  - 4th one is at $z + 12$

- Each 4 groups represent a quarter of the 256 location of memory. After each 4 groups, the address is increased by 1 and all the above points are applied.

The sequence is represented by the following flowchart in figure 74. As the address generated here is the first address of each iteration (including 4 outputs each is separated by 64).

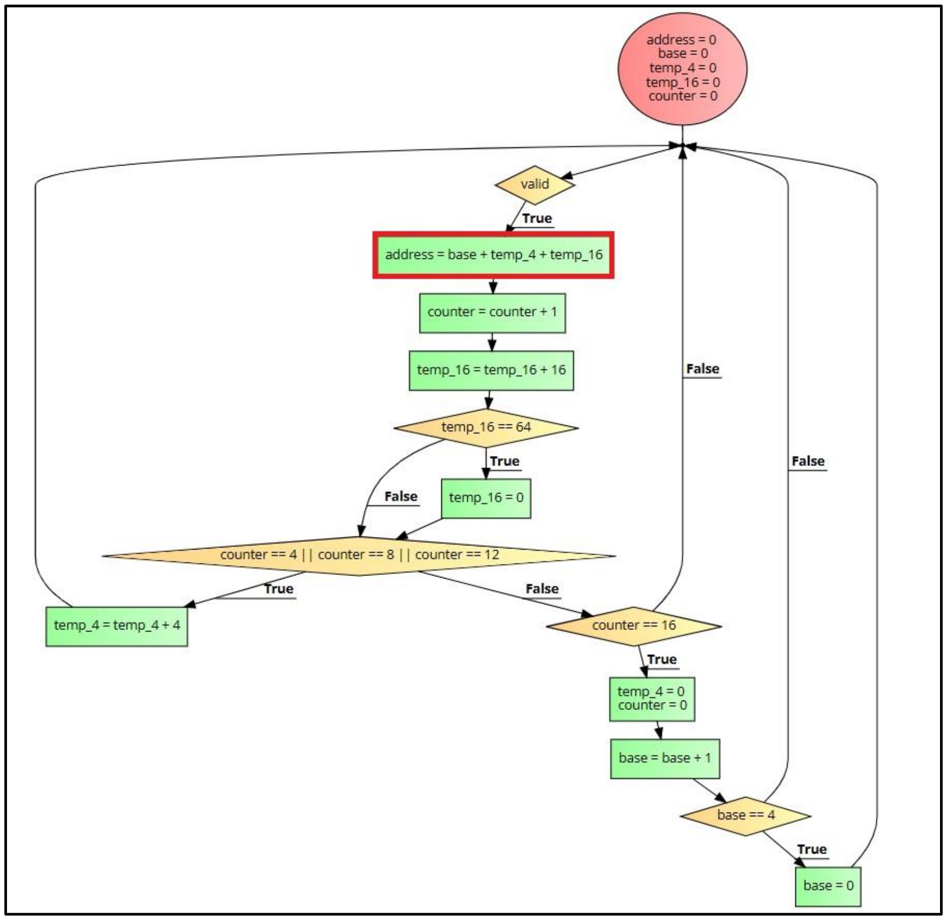


Figure 74: Flowchart represents the sequence of address

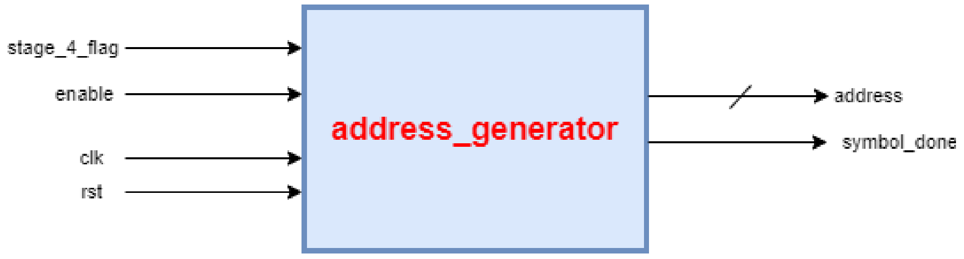


Figure 75: Block diagram of module address_generator

Table 29: Ports description of address_generator

Port	Direction	Width	Description
clk	Input	1	Clock signal
rst	Input	1	Reset signal
enable	Input	1	Enable for address (after data is ready from the quantizer)
stage_4_flag	Input	1	Flag for stage 4 (address is generated for output of stage 4 only)
symbol_done	Output	1	Flag is raised when last sample of the symbol is written in the memory
address	Output	8	Address for the output to be written in the memory

4.6.5 Packet timer

The FFT subsystem takes only the useful data samples. In order to extract only the useful data, there should be a block to filter out the data received and that is the packet timer block.

It has an input signal called stream_start indicates the start of the data stream. It has another input called time_stamp, this value indicates the position of the required SSB in the half frame. So, when these signals are received, the useful symbols location is known. The symbol received is 274 samples, including 18 samples that are cyclic prefix (where the CP duration in 5G standard is 4.69 µsec and that is equivalent to 289 clock cycles as mentioned in section 4.4.3, since the block receives a sample every 16 clock cycles. Therefore $\frac{289}{16} = 18$ samples). CP removal is done in this module to transmit only the useful 256 samples of the symbol [9].

It does the following as an output to the FFT:

- At the start of each useful symbol, it sends a symbol_valid signal to the FFT to start receiving the samples.
- It passes the input_valid signal of the useful samples of the symbol.
- It raises the TTI signal every 1ms from the start of the first useful symbol (which is transmitted to the processor as an interrupt).

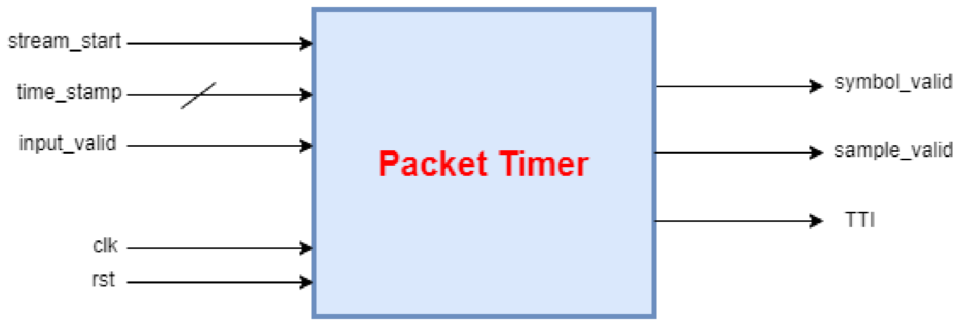


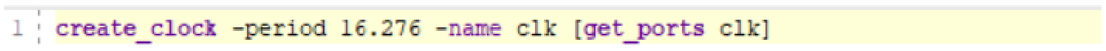
Figure 76: Block diagram of module packet_timer

Table 30: Ports description of packet_timer

Port	Direction	Width	Description
clk	Input	1	Clock signal
rst	Input	1	Reset signal
stream_start	Input	1	Signal to indicate the start of the stream and it resets the free running counter
time_stamp	Input	15	Value indicates the start of the first SSB in half frame(5ms) = first sample of PSS, it resets the primary counter
input_valid	Input	1	Valid for samples received each 16 clock cycles from the start of the free running counter
symbol_valid	Output	1	Signal indicates the start of the symbol
sample_valid	Output	1	A valid signal for the useful samples to be transmitted to the FFT
TTI	Output	1	Transmit Time Interval, a signal to be raised every 1 ms and this is the interval for 5G.

4.6.6 Synthesis

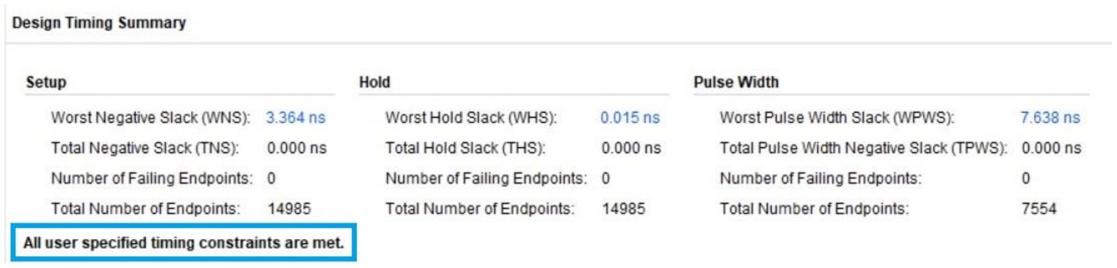
Timing analysis plays a crucial role in verifying the performance of digital designs implemented on FPGAs. Using the clock frequency 61.44 MHz, the FFT subsystem has been synthesized using FPGA part xc7a35tcpg236-1 on Vivado Xilinix. And the timing analysis reports have been generated in order to check if the timing constraints are met. The timing constraint is only the clock frequency (period is 16.276 nsec) as shown in figure 77.



```
create_clock -period 16.276 -name clk [get_ports clk]
```

Figure 77: Timing constraints of the FFT Subsystem

The worst negative slack was found to be 3.364 ns, and it's because of the critical path (which is of the complex multiplier logic path).
We conclude that there is no time violation and the system can work on clock frequency 61.44 MHz with achieving 20.7 % slack.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.364 ns	Worst Hold Slack (WHS):	0.015 ns	Worst Pulse Width Slack (WPWS):	7.638 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	14985	Total Number of Endpoints:	14985	Total Number of Endpoints:	7554

All user specified timing constraints are met.

Figure 78: Timing analysis results of the FFT Subsystem

Chapter 5: Hardware Design of Post-FFT

This block contains some mandatory building blocks, including channel estimation, channel equalization, and demodulation processes (soft de-mapper, de-scrambler, de-interleaver, and bit selection), as shown in Fig. 79.

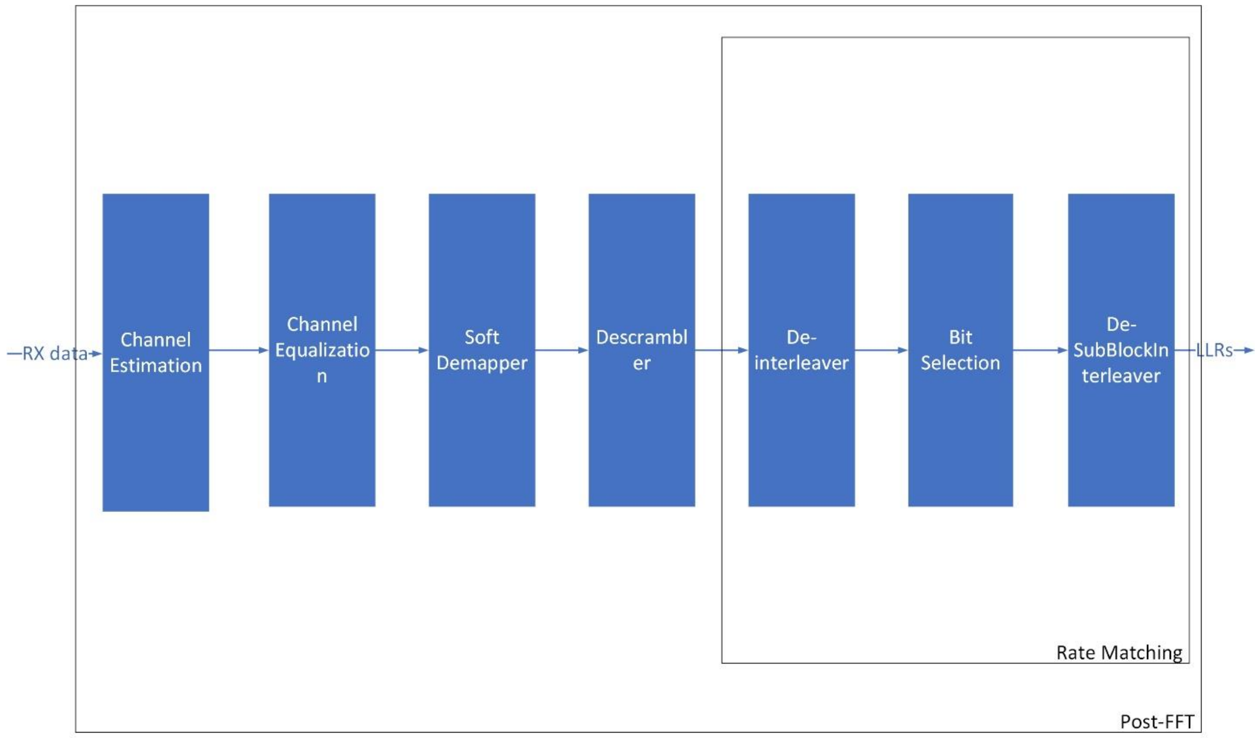


Figure 79: Post-FFT Subsystem

5.1 Channel Estimation

As discussed before, we modelled two modes of channel estimators, but as a hardware choice, we chose the model with the better performance, which is the MMSE. The MMSE needs the channel estimates at the DMRS positions as input, which means that we will use the LSE to get those estimates for the MMSE, and then the MMSE will calculate the estimates for the whole symbol (DMRS+PBCH).

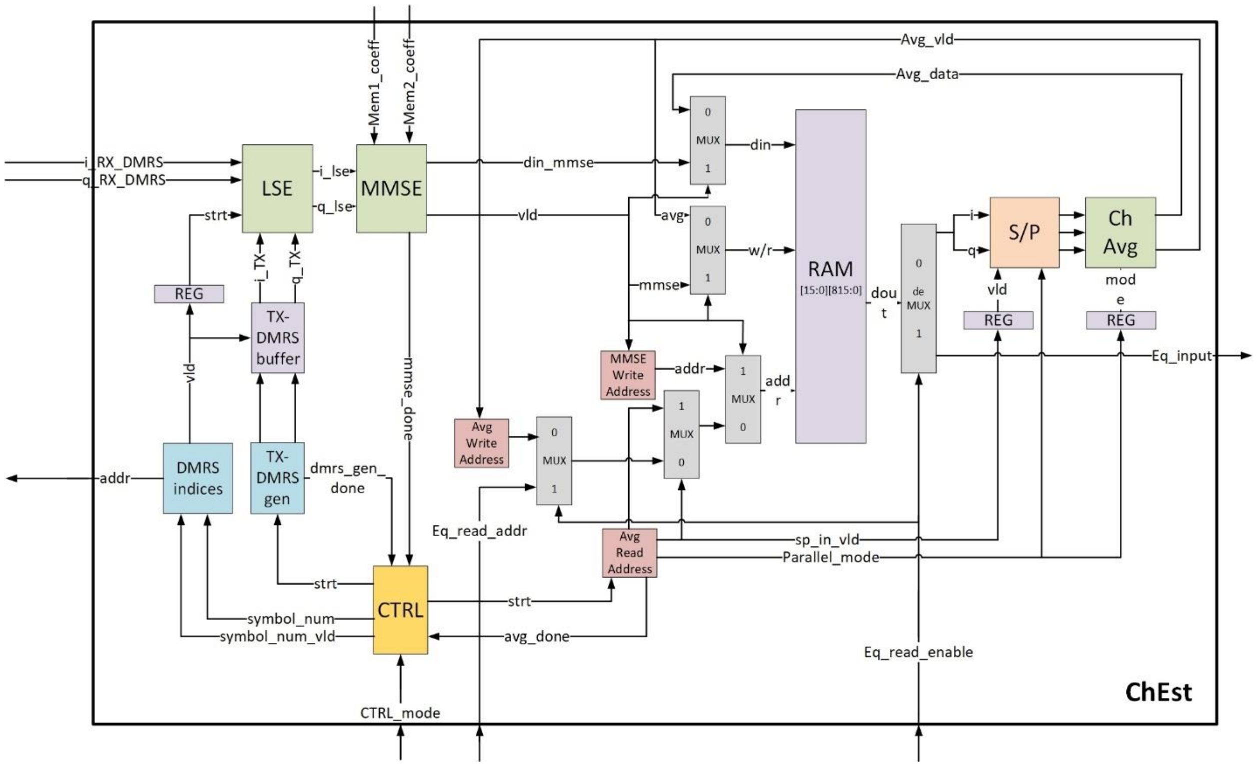


Figure 80: Channel Estimation Architecture

5.2 Tx- DMRS Generation

We said before that the LSE estimation depends on the DMRS received and the DMRS generated in the receiver (pilots). So, we want both Rx_DMRS and Tx_DMRS. For Rx_DMRS, it's an input from the FFT-subsystem memory based on the address location of the DMRS data in the FFT-subsystem memory (the address of the DMRS data will be discussed in the DMRS-indices generation section). For Tx_DMRS, it's generated using the gold sequence as discussed in Section 3.1.4.

The Tx_DMRS generation block architecture is shown in Fig. 81.

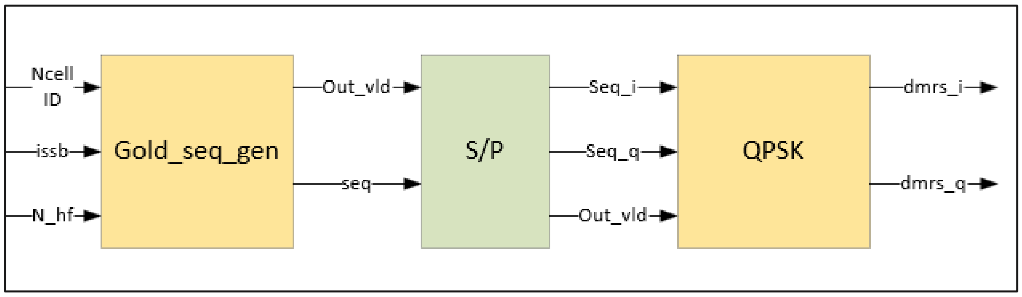


Figure 81: Tx-DMRS generation Architecture

For the gold sequence generation, it's described as having two linear feedback shift registers. Each of them is initialized with a seed value according to the standard specs as discussed in Section 3.1.4, and then the output of the two LFSRs is xored to obtain the gold sequence as shown in Fig. 82.

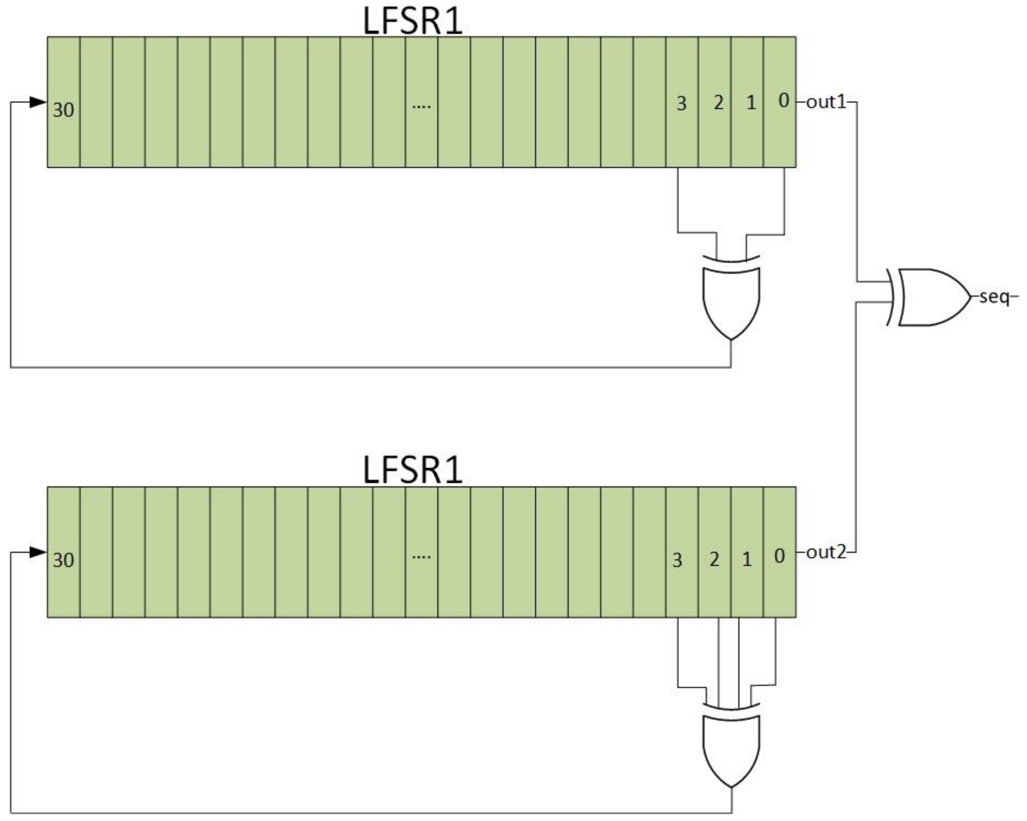


Figure 82: Gold Sequence Generation for DMRS

5.3 DMRS Indices generation

The FFT-subsystem memory stores the 3 OFDM symbols (2nd, 3rd, and 4th symbols in the SSB). To obtain the DMRS data only, we should address the DMRS locations correctly. As discussed in Section 3.1.5, the DMRS locations depend on the value of N_{ID}^{cell}. For the 2nd and 4th symbols, the first DMRS location is N_{ID}^{cell}mod 4, which is simply the least two significant bits in N_{ID}^{cell} (N_{ID}^{cell}[1:0]), then the counter starts to count by 4 to get the second DMRS location, and so on. Why did we add 8 to N_{ID}^{cell}[1:0]? Because the FFT-subsystem memory stores 256 subcarriers, not 240, the first 8 subcarriers and the last 8 subcarriers aren't PBCH or DMRS data. For the third symbol, DMRS exists only above and below SSS. We also start with address N_{ID}^{cell}[1:0] +8 and count by 4 until we reach the SSS, then jump to the first DMRS after the SSS and continue to count by 4 again. The architecture of this block is shown in Fig. 83.

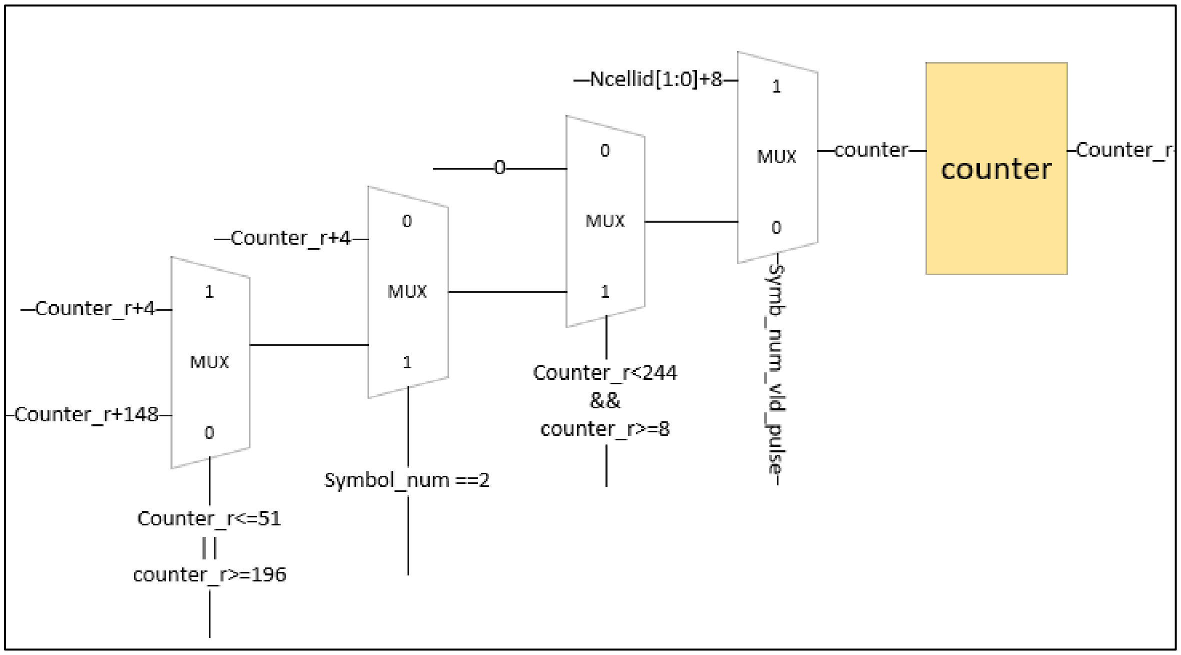


Figure 83: DMRS indices Generation

The timing diagrams shown describe how the DMRS indices are generated for the 3 symbols in the case of $N_{ID}^{cell}\ mod$ 4 = 0 ,1.

For $N_{ID}^{cell} mod\ 4 = 0$, we know that the first sample is a DMRS, then three PBCH samples, and so on. The first sample address is 8, as we said before that we're addressing 256 subcarriers, not 240 (the first and last 8 samples aren't in our concern). So, the first DMRS data is at address 8.

For the 3rd symbol in the SSB, there's SSS data in the middle, so the DMRS address jumps from 52 to 200 to skip the SSS data in the OFDM symbol.

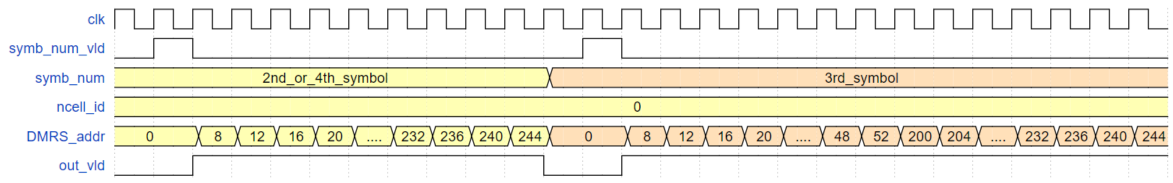


Figure 84: DMRS indices at N_{ID}^{cell} mod 4=0

For $N_{ID}^{cell}\ mod\ 4 = 1$, we know that the first sample is a PBCH, then 1 sample is a DMRS, then 3 PBCH samples, and so on. So, the first DMRS address is 9 in this case.

For the 3rd symbol in the SSB, there's SSS data in the middle so the DMRS address jumps from 53 to 201 to skip the SSS data in the OFDM symbol.

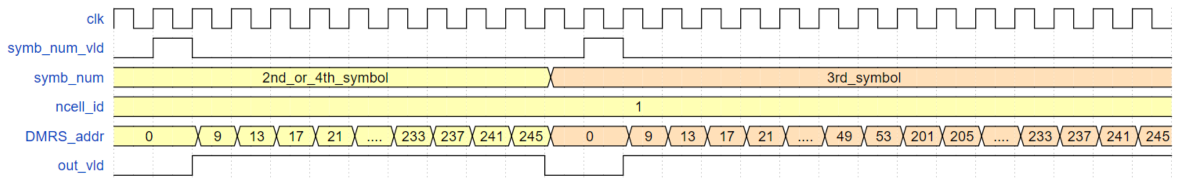


Figure 85: DMRS indices at N_{ID}^{cell} mod 4=1

The same way is applied in case of $N_{ID}^{cell}\ mod\ 4 = 2,3$.

This DMRS address generator addresses the FFT-subsystem memory to get the DMRS data received. We might have an offset to this address according to the design of the FFT-subsystem memory we address from.

5.4 Lease Squares Estimator (LSE)

$$\hat{H}_{LS} = \frac{Y_{LS}}{X_{LS}}$$

To implement this equation as hardware, we need a complex divider, which is not recommended to use, so we convert this equation to another suitable form.

$$\hat{H}_{DMRS} = \frac{Y_{DMRS}}{X_{DMRS}} * \frac{conj(X_{DMRS})}{conj(X_{DMRS})}$$

As we discussed before, the X_{DMRS} values are all QPSK values ($\pm\frac{1}{\sqrt{2}}\pm\frac{1}{\sqrt{2}}$) which means the term $X_{DMRS} * conj(X_{DMRS}) = 1$, so the LS equation can be deduced to:

$$\hat{H}_{DMRS} = Y_{DMRS} * conj(X_{DMRS})$$

Now the LS is a complex multiplication:

$$\hat{H}_{DMRS} = (Y_R + jY_I) * (X_R - jX_I)$$

We have two choices to implement this complex multiplication:
<u>First Method:</u>

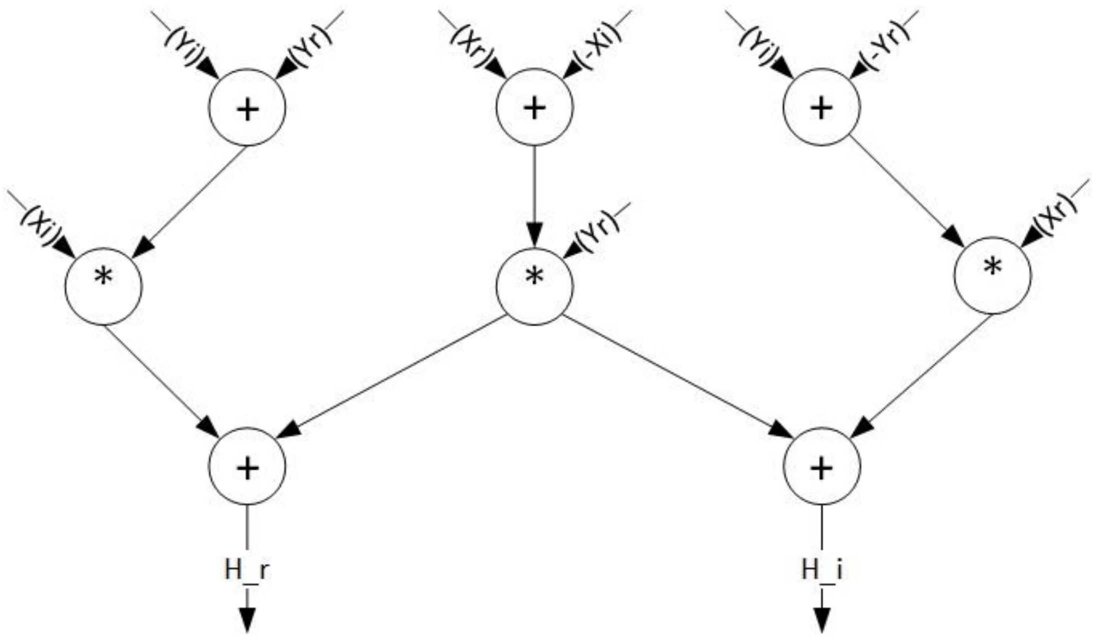
$$\hat{H}_{DMRS} = (Y_R * X_R + Y_I * X_I) + j(Y_I * X_R - Y_R * X_I)$$

This form requires 4 multipliers and 3 adders.
<u>Second Method:</u>

$$\hat{H}_{DMRS} = (Y_R * (X_R - X_I) + X_I * (Y_R + Y_I)) + j(Y_R * (X_R - X_I) + X_R * (Y_I - Y_R))$$

This form requires 3 multipliers and 5 adders.

So, there is a tradeoff between 1 multiplier vs 2 adders, we go with the second method.



5.5 Minimum Mean Square Estimator (MMSE)

$$H_{MMSE} = R_{HP}\left(R_{HH} + \frac{\sigma_z^2}{\sigma_x^2} I\right)^{-1} H_{LS}$$

$$H_{MMSE} = F(nTaps, SNR, K) * H_{LS}$$

To implement the above equation in hardware, we need to store the coefficients $F(nTaps, SNR)$ in an external memory that is filled by the software depending on the value of the SNR and the number of channel taps. Then matrix multiplication will be done.

For the second and fourth OFDM symbols in SSB:

We have the channel estimates at the DMRS subcarriers (H_{LS}), which have a size 60×1 for each OFDM symbol (2nd, 4th).

Coefficients matrix size = 240×60, H_{LS} size = 60×1. As we have 60 DMRS values. H_{MMSE} Size = (240×60) * (60×1) = 240×1, as we have 240 subcarriers in each OFDM symbol (2nd, 4th). Now we have the channel estimates at both DMRS and PBCH subcarriers for the second and fourth OFDM symbols in SSB.

For the third OFDM symbols in SSB:

This OFDM symbol has SSS in the middle of it, so we want to estimate the channel above and below the SSS. We have the channel estimates at the DMRS subcarriers H_{LS}, which has a size 12×1 above and below the SSS.

Coefficients matrix size = 48×12, H_{LS} size = 12×1. as we have 12 DMRS values above and below SSS. H_{MMSE} Size = 48×1. as we have 48 subcarriers above and below the SSS. Now we have the channel estimates at both DMRS and PBCH subcarriers for the third OFDM symbol in SSB.

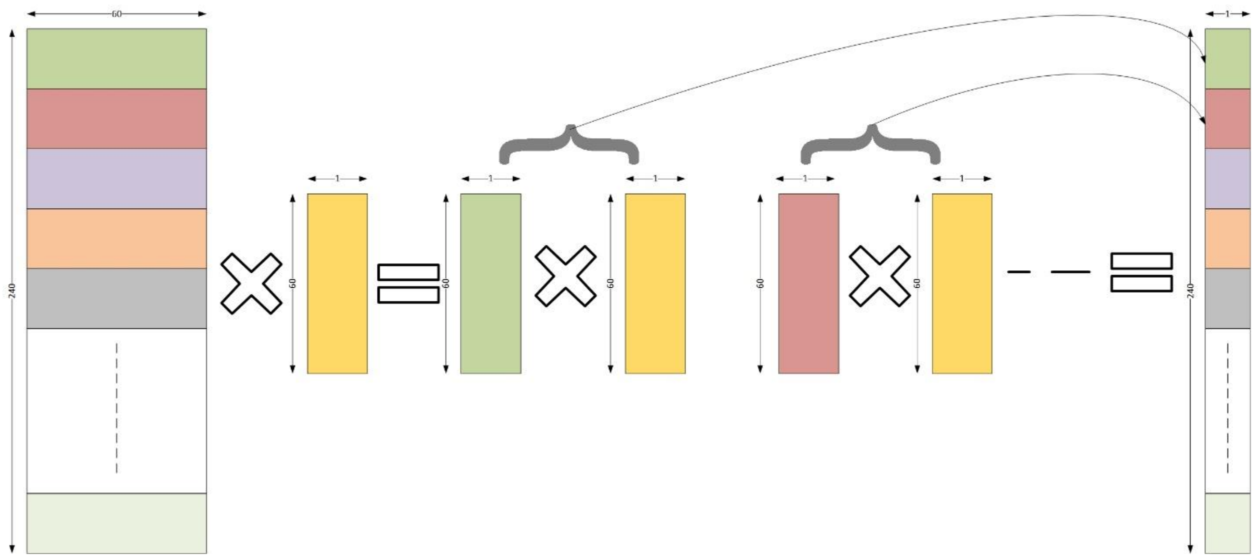


Figure 86: MMSE Matrix Multiplication

As shown above, this matrix multiplication requires many multipliers and adders. We have only 12 multipliers to perform this operation. So, we do each row-column multiplication in 5 clock cycles (5*12) as shown below. The whole multiplication requires 5*240 clock cycles.

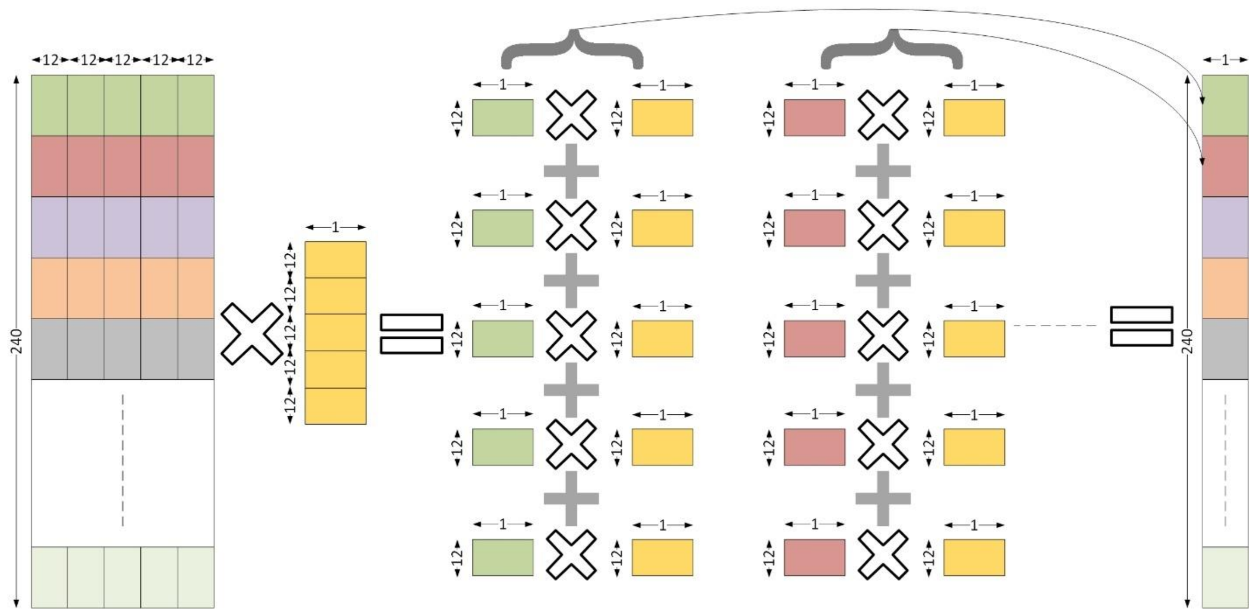


Figure 87: Detailed MMSE Matrix Multiplication

As discussed above, we want 12 coefficients in each clock cycle to perform the multiplication process. The coefficients are stored in two memories, each of which is 1248×6. This is in order to obtain 12 coefficients (6 from each memory) in each clock cycle. Why 1248? We have 240×60 coefficients (for the 2nd and 4th symbols) and 48×12 coefficients (for the 3rd symbol). So, we have a total of 14976 coefficients (240*60 + 48*12). Each memory has 14976/2 = 7488 coefficients. So, each memory is 1248*6 to obtain 6 coefficients from each memory in one clock cycle.

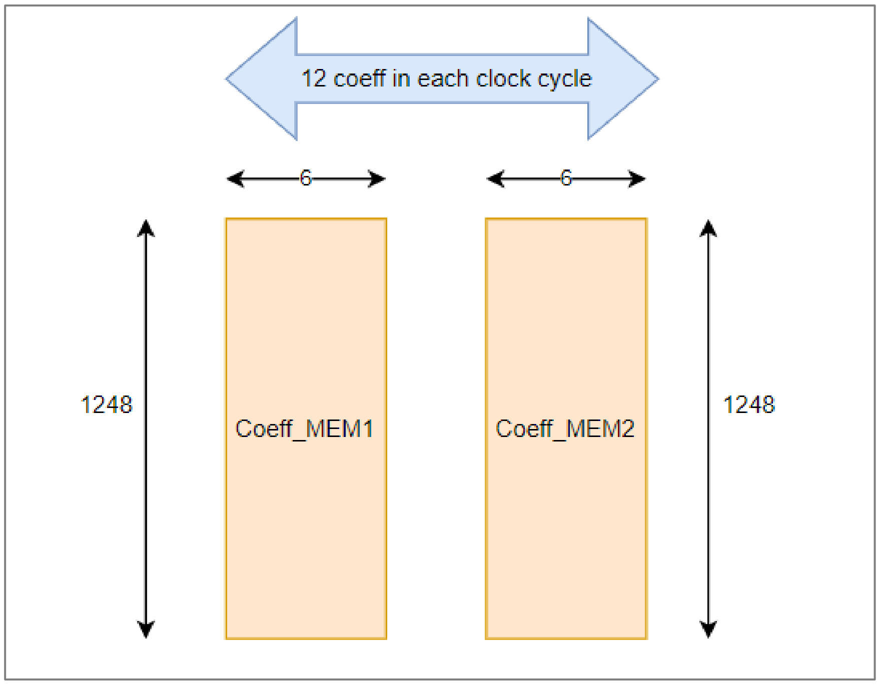


Figure 88: Coefficients Memories

5.6 Channel Average

After the estimation of the channel at the three OFDM symbols (2nd, 3rd, and 4th in the SSB), these estimates are stored in RAM. The channel estimates of the 2nd and 4th symbols are 240 subcarriers each, and the channel estimates of the 3rd symbol are 96 channel estimates only (due to the SSS region in the middle). So, the RAM stores 240 * 2 + 96 = 576 channel estimates. After that, we get the average across the time to have one-symbol channel estimation per SSB having 240 subcarriers, as this reduces the estimation noise and increases the channel estimation accuracy. The channel average block takes the channel estimates of each symbol from the RAM and gets the channel average across the three symbols. The latest 240 channel estimates, which are outputs from the channel average block, are stored in the RAM. Now the size of the RAM is 576 + 240 = 816 samples.

Now we have a problem: the channel average block takes 3 estimates to average above and below the SSS location and 2 estimates to average at the location of the SSS, as shown in Fig. 89. This means that we want a signal that identifies whether we are averaging 2 or 3 estimates. This signal is the parallel_mode signal, which takes the value 2 or 3 that is shown in Fig. 80.

To summarize the process, the channel average block starts with the avg_start signal from the controller, then the average read address generator block starts to read the channel estimates from the RAM (which may be 2 or 3 estimates according to the parallel_mode signal) and pass them to the S/P block to pass 2 or 3 estimates to the channel average block, then the average starts to average these estimates to one estimate and write it back to the RAM in the average write address location.

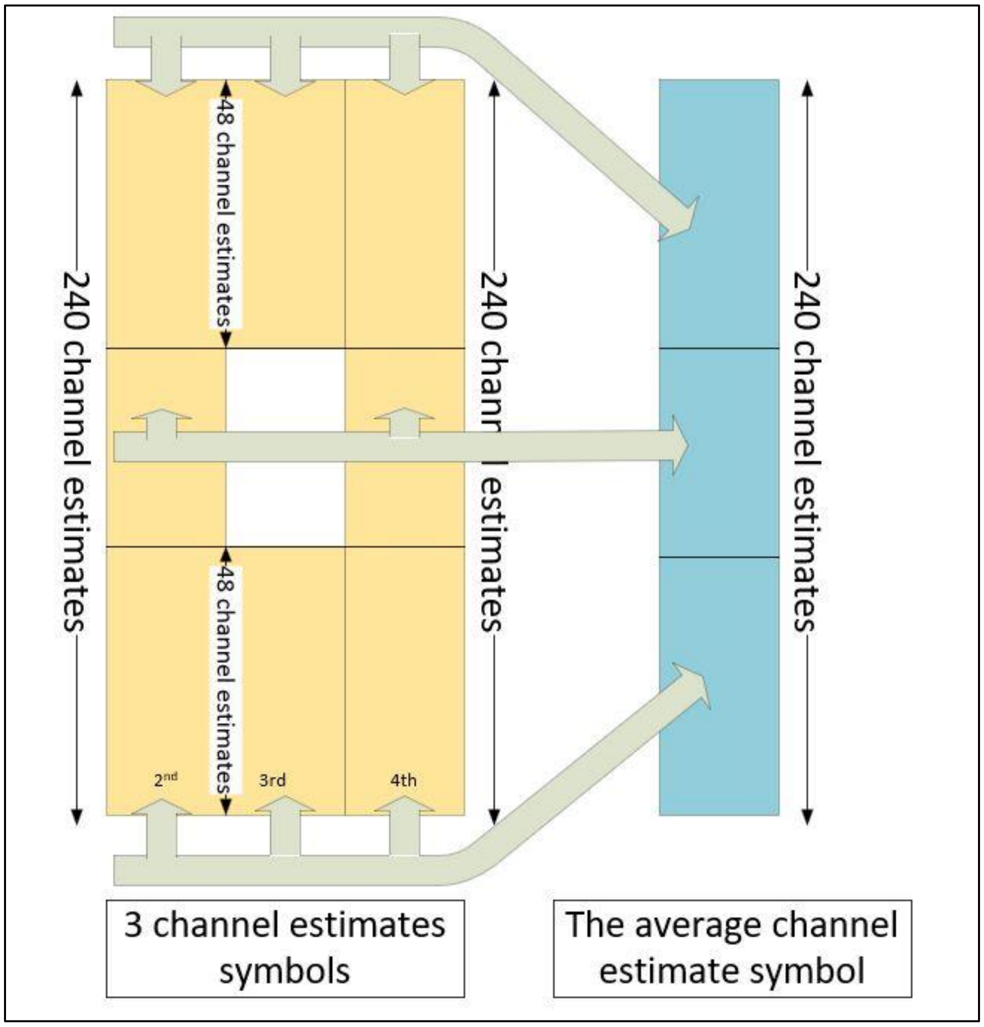


Figure 89: Channel estimates averaging

Now we have the channel estimate symbol with 240 subcarriers stored in the RAM, from which the equalizer will read and start equalization to get the estimated PBCH.

5.7 Channel Equalization

Channel Equalization is performed by using maximum ratio combining (MRC), which is simply getting the estimated PBCH data from the channel estimates by dividing them. The estimate of each OFDM symbol in the SSB data is given by:

$$\hat{X}_{PBCH} = conj(\hat{\mathrm{H}}) * Y_{PBCH}$$

So, we will do it the same as in the least squares estimator section.

The equalizer needs the received PBCH and the channel estimates at the PBCH locations. For the channel estimates at the PBCH locations, we will get them from the RAM we discussed in the Channel

Average section. For the received PBCH data, we will get it from the FFT-subsystem memory based on the address of the PBCH data.

Till now, we need the addresses of the PBCH data in the FFT-subsystem memory and in the RAM, which takes us to another block, the PBCH indices generation block.

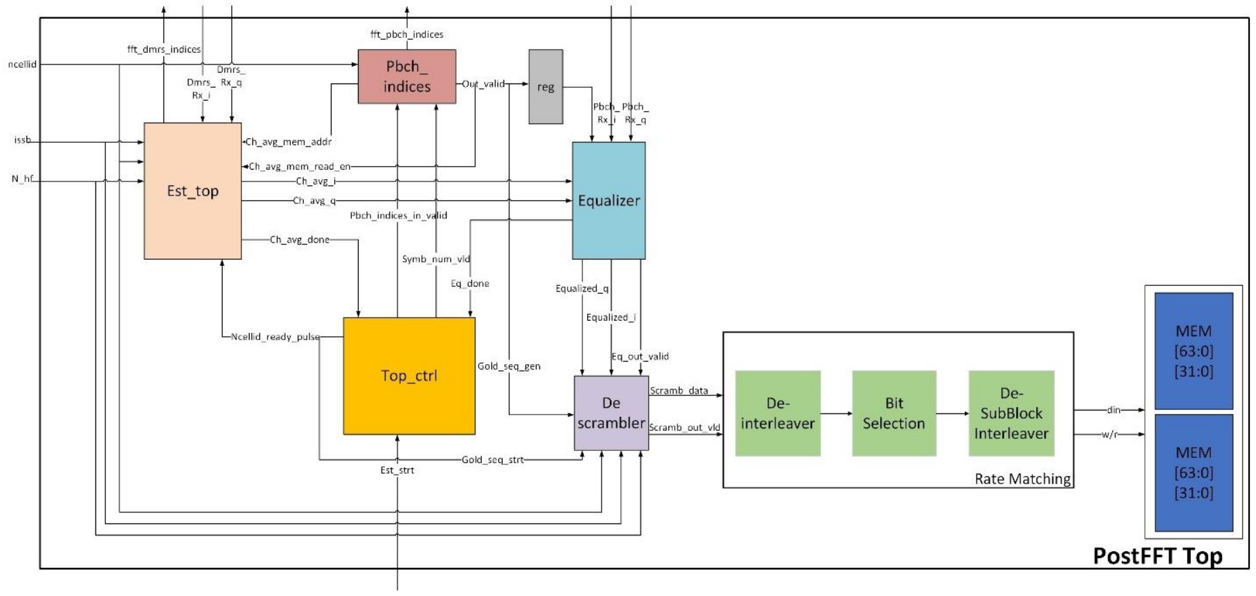


Figure 90: Post-FFT Processing Chain

5.8 PBCH indices Generation

The timing diagrams shown describe how the PBCH indices are generated for the 3 symbols in the case of N_{ID}^{cell} mod 4 = 0 ,1.

For $N_{ID}^{cell} mod\ 4 = 0$, we know that the first sample is a DMRS, then three PBCH samples, and so on. The first sample address is 8, as we said before that we're addressing 256 subcarriers, not 240 (the first and last 8 samples aren't in our concern). So, the first PBCH data is at address 9.

For the 3rd symbol in the SSB, there's SSS data in the middle, so the PBCH address jumps from 55 to 201 to skip the SSS data in the OFDM symbol.

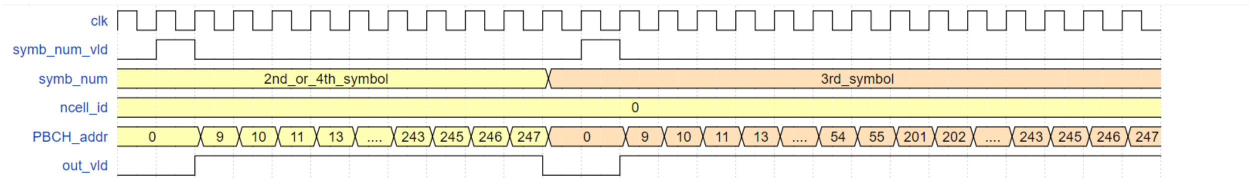


Figure 91: PBCH indices at N_{ID}^{cell} mod 4=0

For $N_{ID}^{cell}\ mod\ 4 = 1$, we know that the first sample is a PBCH, then 1 sample of DMRS, then 3 PBCH samples, and so on. So, the first PBCH address is 8 in this case.

For the 3rd symbol in the SSB, there's SSS data in the middle, so the PBCH address jumps from 55 to 200 to skip the SSS data in the OFDM symbol.

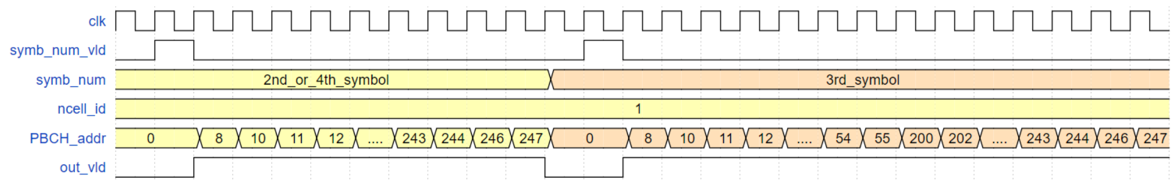


Figure 92: PBCH indices at N_{ID}^{cell} mod 4=1

The same way is applied in case of $N_{ID}^{cell}\ mod\ 4 = 2{,}3$.

This PBCH address generator addresses both the FFT-subsystem memory and average channel estimates RAM to get the PBCH data received and the channel estimates at the PBCH locations, respectively. We might have an offset to this address according to the memory design we address from.

5.9 De-Scrambler

The equalizer passes the PBCH data estimates to the de-scrambler, which is complex data. The de-scrambler works as we said before in Section 3.1.2. We have two linear feedback shift registers, and each of them is initialized with a seed value according to the standard specs. Then, a gold sequence is generated from the two linear feedback shift registers. This gold sequence is either 0 or 1. If the gold sequence output is 0, then the descrambled data will be the same as the equalized data. If the gold sequence output is 1, the descrambled data will be negative equalized data.

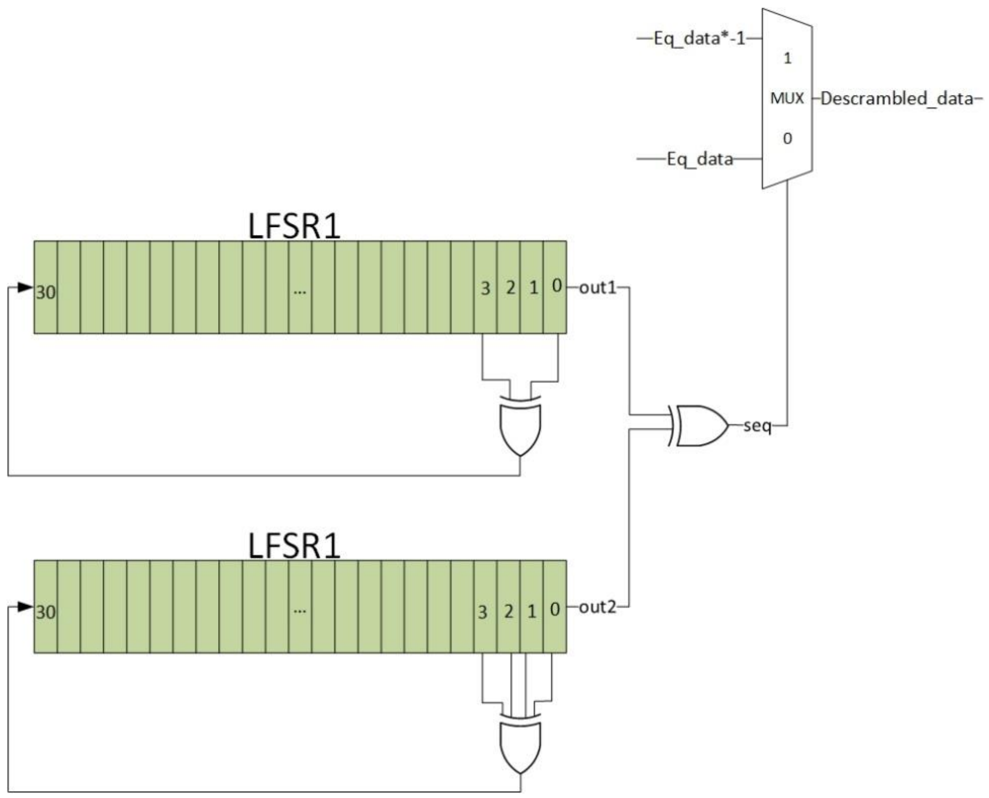


Figure 93: De-scrambler Architecture

Now we have a problem: the equalizer passes complex data (equalized_i, equalized_q) each clock cycle. This means that the descrambler takes two PBCH equalized data each clock cycle, and the descrambler gold sequence generates one seq bit each clock cycle, so we need to stop the equalizer one clock cycle in order to get the gold sequence bits for both equalized_i and equalized_q. In other words, the equalizer should work one clock cycle and stop one clock cycle while the descrambler works each clock cycle.

This approach is done using the pbch_indices_in_valid signal, which comes from the top controller, to enable the PBCH indices block for one clock cycle, disable it for one clock cycle, and so on. With this signal, the PBCH indices block won't change the address of the PBCH received data nor the address of the channel estimates at the PBCH locations unless the pbch_indices_in_valid signal is enabled, so the equalizer won't change its output data accordingly.

5.10 De-Rate Matching

De-Rate Matching consists of three blocks: de-interleaver, bit-selection, and sub-block de-interleaver. The first and third blocks re-arrange the data, but the second block selects some data and drops others. The input to the de-rate matching block is 864 LLRs, and the output is 512 LLRs. That's why some LLRs are dropped, and others change their locations. To be clearer, descrambler throughput is 1 LLR per clock cycle, and the de-rate matching block checks whether this LLR will be dropped or stored in the LLRs memory depending on the MSB of the addresses stored in the LUT. These addresses stored in the LUT are specified in the standard.

For example, the first LLR coming from the descrambler will be stored at address 0 in the first LLR memory, and the sixteenth LLR coming from the descrambler will be dropped and won't be stored. The counter counts with the descrambler output valid. This counter points to the LUT to select the address of the current LLR. If the current LLR will be dropped and won't be stored in the memory, then the MSB in the address stored in the LUT will be 1, which disables the WE of the memory.

The LLR memory size should have been 512 LLRs × 8 bits each, but the next stage, which is the polar decoder, needs this memory to be divided into two memories with a word length of 32 bits, which means that each word contains 4 LLRs. So, we have two memories with a size of 64 LLRs × 32 bits each.

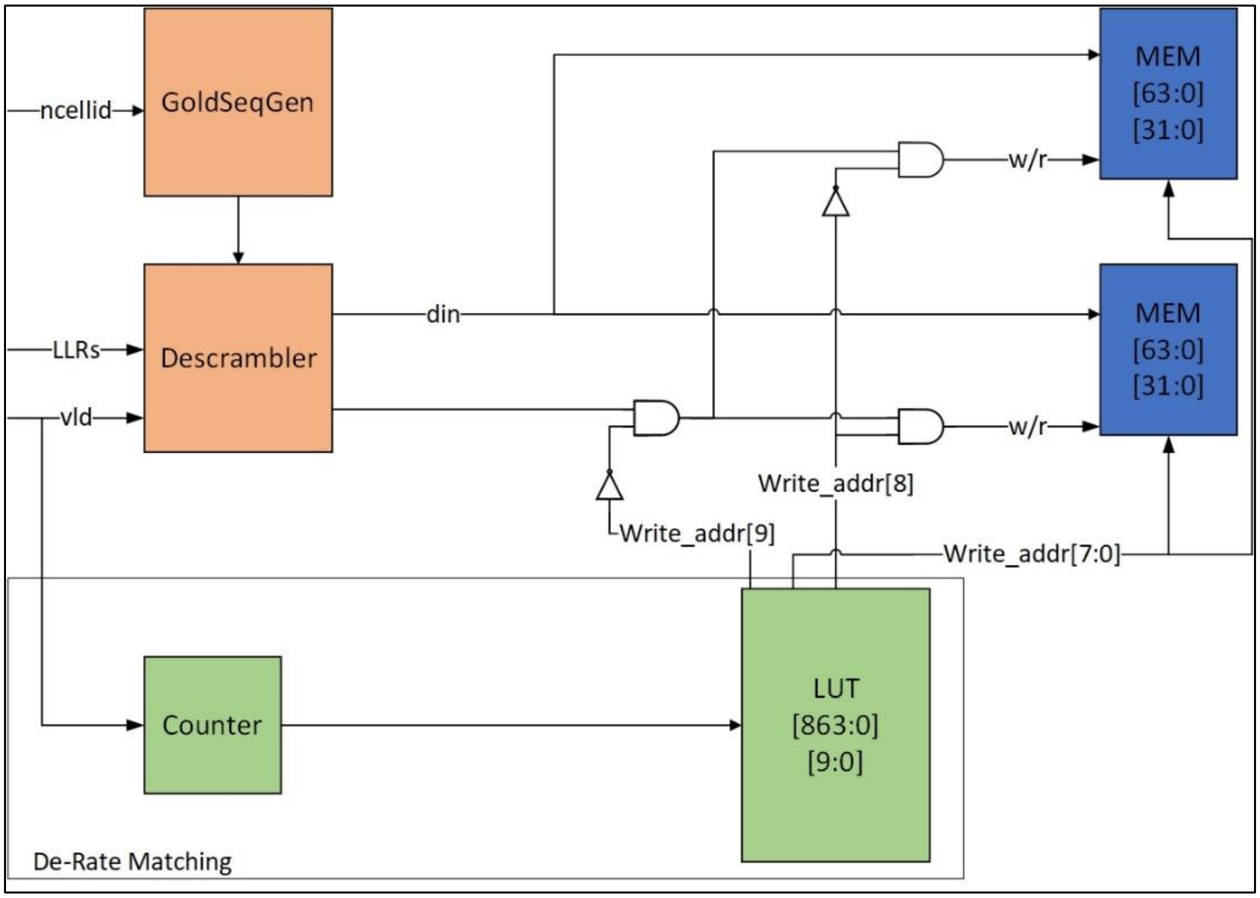


Figure 94: De-Rate Matching Architecture

5.11 Post-FFT Interfaces

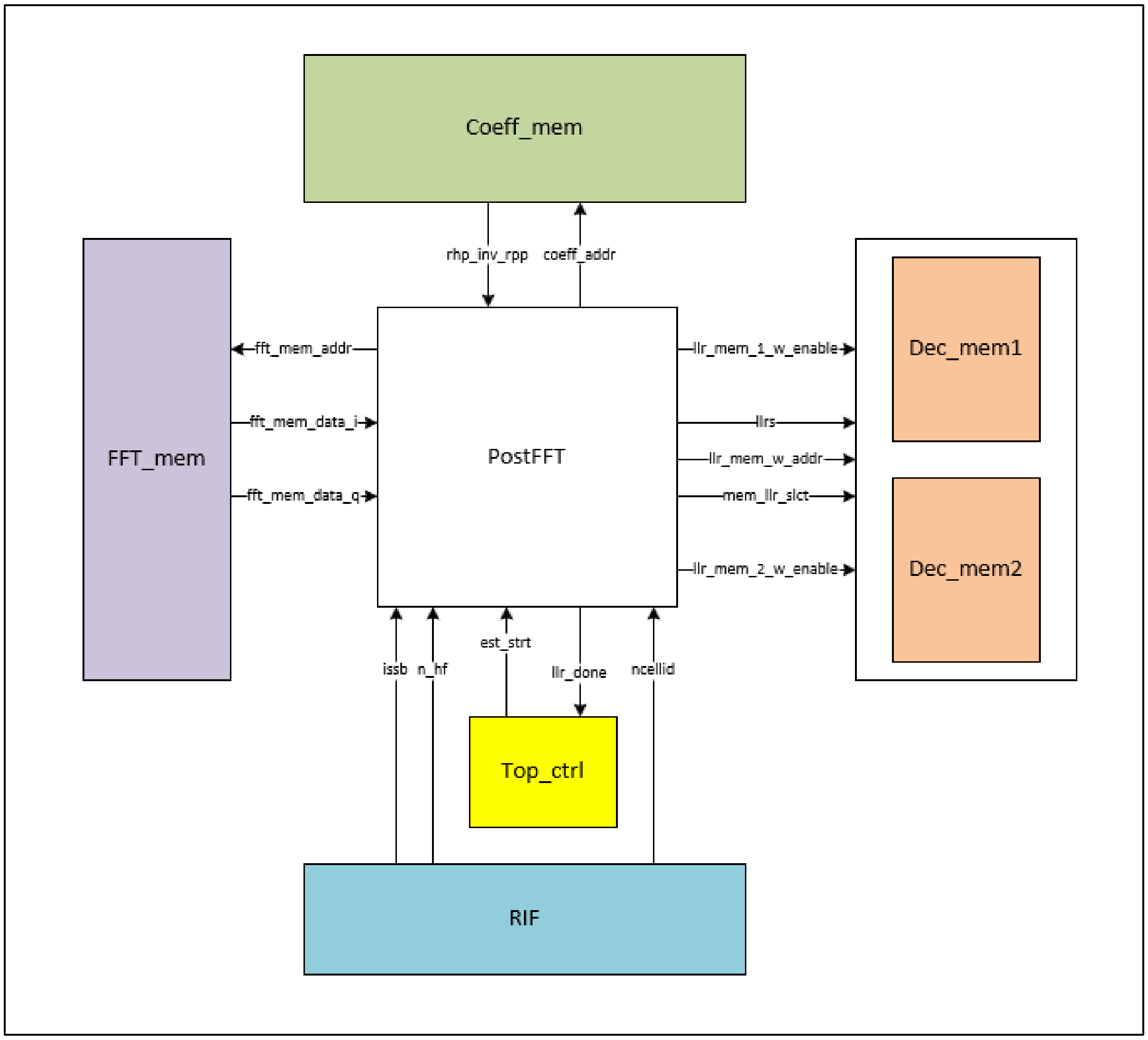


Figure 95: Post-FFT Interfaces

Ports Description is shown in the following table:

Table 31: post-FFT interfaces

Port Name	Port Direc.	Port Width	Active Low/ High	Port Description
issb	input	2	NA	The index of the SSB in the half frame. Range 0→3.
ncellid	input	10	NA	Cell identifier number. 0→1007.

n_hf	input	1	NA	0→ SSB in the first half in the frame. 1→ second half.
clk	input	1	NA	System clock = 61.44 MHz
rst	input	1	Low	Asynchronous reset.
fft_mem_addr	output	10	NA	The address of the data stored in the fft_mem. It may be DMRS or PBCH address (selected internally).
fft_mem_data_i	input	12	NA	The real part of the data is stored in fft_mem. It may be DMRS or PBCH data (selected internally).
fft_mem_data_q	input	12	NA	The imaginary part. Same as above
llrs	output	8	NA	The channel LLRs stored in the dec_mem.
llr_mem_w_addr	output	6	NA	The address of the channel LLRs that will be stored in the dec_mem.
llr_mem_1_w_enable	output	1	High	Write Enable Level of the first decoder memory. Decoder has 2 memories each of width 32 bits and depth 64 bits.
llr_mem_2_w_enable	output	1	High	Write Enable Level of the second decoder memory.
mem_llr_slct	output	2	NA	Selects which part (byte) of the word in the decoder memory will be written into.
est_strt	input	1	High	Pulse allows postFFT to start.
llr_done	output	1	High	Pulse informs top_ctrl that all LLRs are stored in the dec_mem
coeff_addr	output	11	NA	Coefficients address which are needed for MMSE calculations.
rhp_inv_rpp	input	8*12	NA	This signal is the data comes from the coeff_mem to the MMSE.

5.12 Block Level Testing for PBCH Processing Chain

Using a self-checking testbench to verify the output of an RTL design is a common and effective way to ensure that the design works correctly. By comparing the output of the RTL design with the expected output from a reference model such as MATLAB, you can determine whether the RTL design is producing the correct results.

To implement a self-checking testbench, we typically use a file-based approach, where the testbench reads in input data and the expected output data from files generated in MATLAB. These files contain the test vectors that we want to use to verify the RTL design.

In the testbench, we apply the inputs from the input file and compare the RTL output with the expected output file. If the outputs match, the test passes; if the outputs do not match, the test fails, and we would need to investigate and correct the RTL design.

It's important to ensure that the input and output files are formatted correctly, and that the RTL design and the reference model use the same input data. Additionally, we consider using multiple test vectors to ensure that the design is working correctly for a range of inputs and not just a single test case.

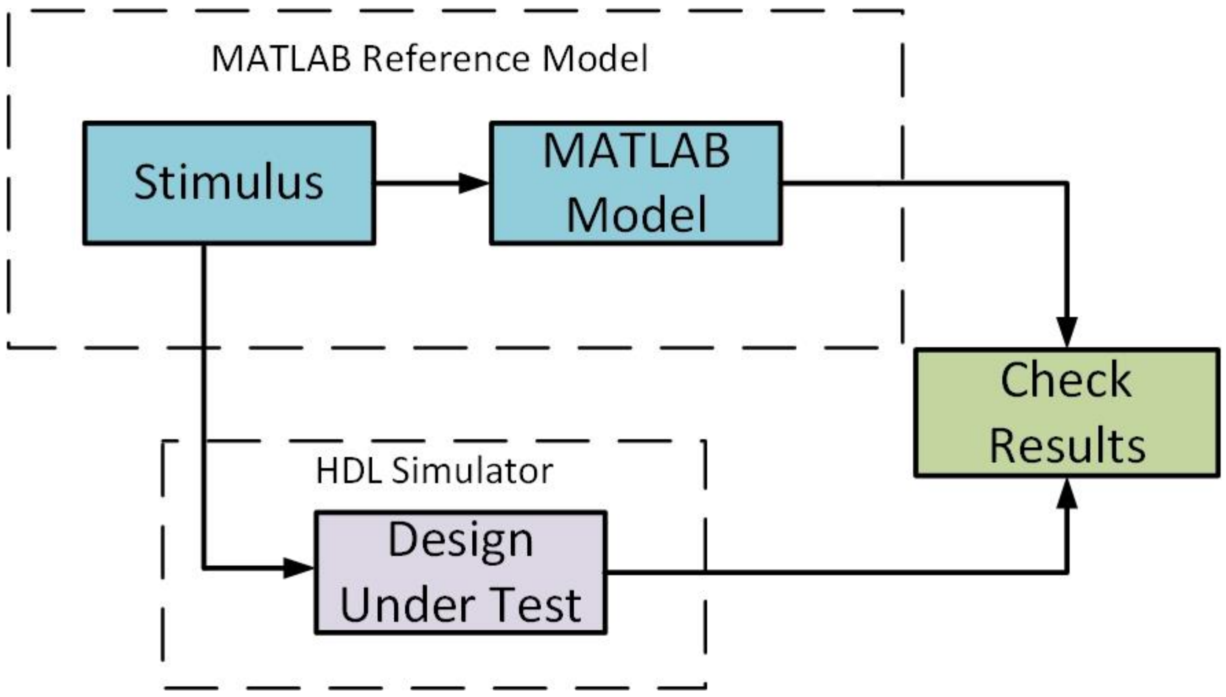
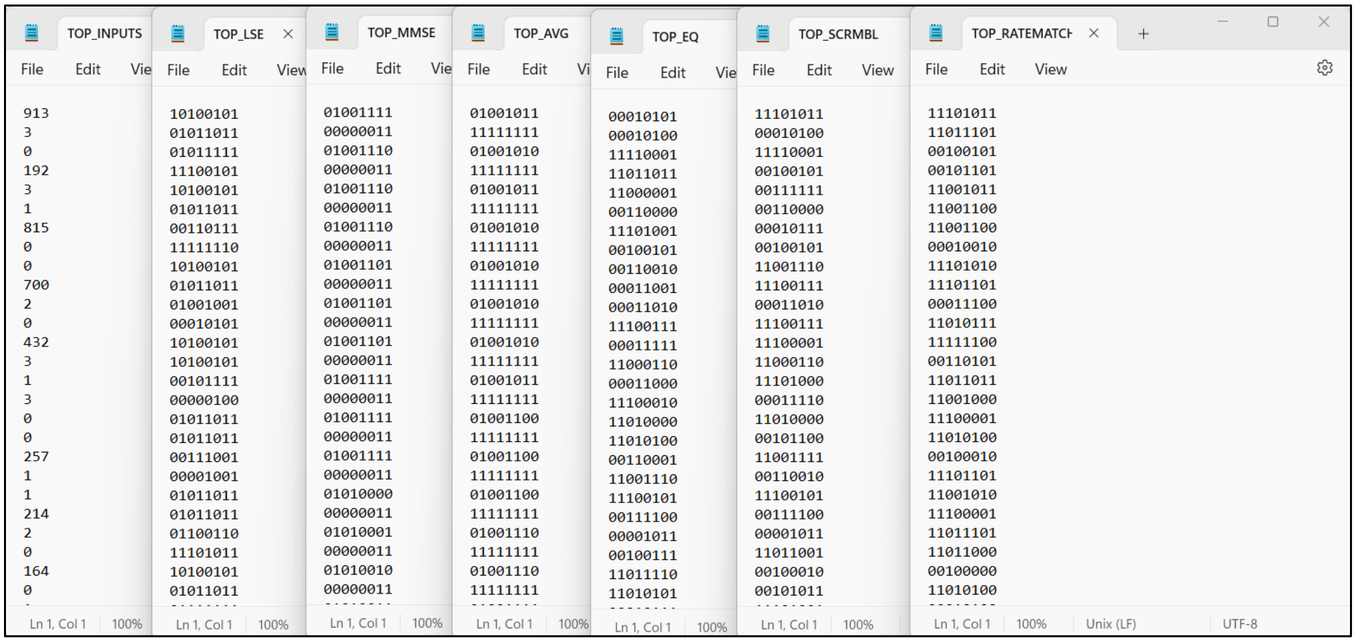


Figure 96: MATLAB vs RTL results

We used 10,000 random test cases to ensure that we covered most of the cases, and all of them passed in the RTL. The inputs to the chain are $N_{ID}^{cell}, i_{SSB}, and\ N_{hf}$. These inputs are randomized in the MATLAB reference model 10,000 times and stored in a file, as well as the outputs from each stage in the MATLAB reference model. The following figure shows an example of the files generated by MATLAB.



TOP_INPUTS

```
913
3
0
192
3
1
815
0
0
700
2
0
432
3
1
3
0
0
257
1
1
214
2
0
164
0
```

TOP_LSE

```
10100101
01011011
01011111
11100101
10100101
01011011
00110111
11111110
10100101
01011011
01001001
00010101
10100101
10100101
00101111
00000100
01011011
01011011
00111001
00001001
01011011
01011011
01100110
11101011
10100101
01011011
```

TOP_MMSE

```
01001111
00000011
01001110
00000011
01001110
00000011
01001110
00000011
01001101
00000011
01001101
00000011
01001101
00000011
01001111
00000011
01001111
00000011
01001111
00000011
01010000
00000011
01010001
00000011
01010010
00000011
```

TOP_AVG

```
01001011
11111111
01001010
11111111
01001011
11111111
01001010
11111111
01001010
11111111
01001010
11111111
01001010
11111111
01001011
11111111
01001100
11111111
01001100
11111111
01001100
11111111
01001110
11111111
01001110
11111111
```

TOP_EQ

```
00010101
00010100
11110001
11011011
11000001
00110000
11101001
00100101
00110010
00011001
00011010
11100111
00011111
11000110
00011000
11100010
11010000
11010100
00110001
11001110
11100101
00111100
00001011
00100111
11011110
11010101
```

TOP_SCRMBL

```
11101011
00010100
11110001
00100101
00111111
00110000
00010111
00100101
11001110
11100111
00011010
11100111
11100001
11000110
11101000
00011110
11010000
00101100
11001111
00110010
11100101
00111100
00001011
11011001
00100010
00101011
```

TOP_RATEMATCH

```
11101011
11011101
00100101
00101101
11001011
11001100
11001100
00010010
11101010
11101101
00011100
11010111
11111100
00110101
11011011
11001000
11100001
11010100
00100010
11101101
11001010
11100001
11011101
11011000
00100000
11010100
```

Figure 97: Test Vectors MATLAB files

Chapter 6: PHY Integration

6.1 MIB Decoding Process

The PBCH decoding processor implements the typical receiver chain for PBCH detection. This processor has some mandatory building blocks, including the FFT, post-FFT (channel estimation and equalization, demodulation process), and the channel decoding stage, as shown in Fig. 98.

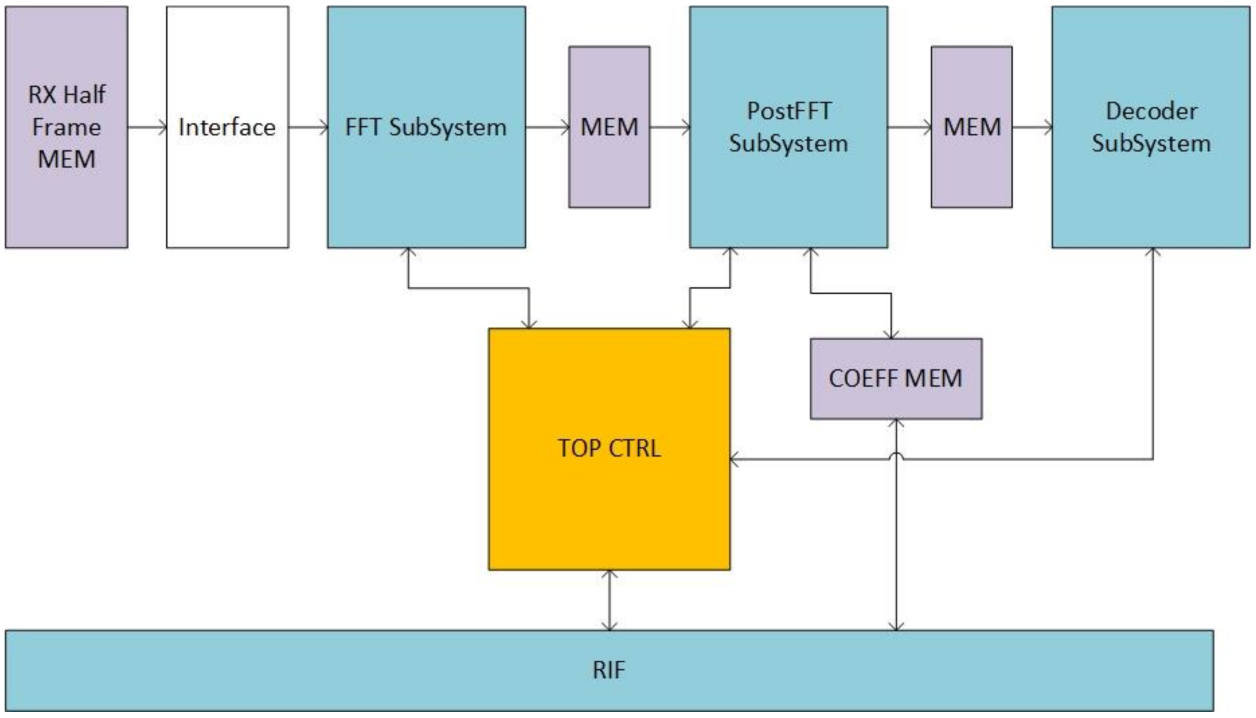


Figure 98: PBCH Decoding Processor

The input of the receiver chain should come from the RF transceiver, so we emulate the RF transceiver with a digital block that stores the received half frame data in a memory (RX Half Frame MEM) and provides the receiver chain with these data at the required time.

As discussed before, the sampling rate is 3.84 MHz and the clock frequency is 16 * 3.84 MHz, so the FFT should receive one sample per 16 clock cycles. The interface block provides the FFT with the inputs at this rate (1 sample per 16 clock cycles).

The FFT subsystem starts its operation with a start pulse that comes from the top controller, and then the outputs of the FFT subsystem are stored in a memory, which are the 256 samples of the 2nd, 3rd, and 4th OFDM symbols in the SSB. So, the memory size is $256*3=768\ samples$. Once all 768 outputs of the FFT are stored in the memory, the top controller sends a start pulse to the post-FFT subsystem. The post-FFT subsystem starts the channel estimation and equalization process using the data stored in the FFT memory as well as the MMSE coefficients that are stored in a coefficient memory that is filled by the RIF. After channel equalization, the processing chain (de-scrambler and de-rate matching) is implemented to store the final 512 LLRs in memory. Once the post-FFT subsystem stores the 512 LLRs in the memory, the top-controller sends a start pulse to the decoder subsystem. The decoder subsystem starts decoding the 512 LLRs to 56 bits (24-bit CRC and 32-bit payload). The decoder output is not only the payload but also the CRC result, which tells the top-controller whether the CRC passed or failed to decode the payload. Once the decoder finishes, the top-controller checks whether the CRC result is passed or failed, to change the i_{SSB} in case the CRC result is failed and starts the processing chain again. This is what we call "blind decoding."

6.2 Blind Decoder

In 5G systems, decoding the Master Information Block (MIB) transmitted by the base station is a critical process that enables a reliable connection between the UE and the base station. However, due to various factors such as channel impairments, multipath fading, and interference, accurately decoding the MIB can be challenging, especially when the Initial Synchronization Signal Block (ISSB) index is unknown.

The term "blind decoding" refers to the process of decoding the MIB without prior knowledge of the ISSB index, relying solely on the received signal. The blind decoding algorithm is typically implemented in the receiver of the mobile device. It operates by searching through a set of possible ISSB indices and evaluating the likelihood of each index based on the received signal characteristics.

The algorithm explores different possibilities until the correct ISSB index is found. This process involves using known synchronization signals and exploiting statistical properties of the received signal, such as signal strength, timing, and correlation.

In this section, we will explore the used blind decoding algorithm and its hardware implementation.

Blind Decoder Algorithm:

The blind decoding algorithm connects the PHY subsystems to find the correct ISSB and to decode the MIB payload; the decoding steps are shown in **Algorithm**, [5]. The decoding algorithm acts like a controller and issues the enable signals required for each of the subsystems to operate, and this will be discussed in more detail in the following section.

The decoded payload is reported whenever the CRC is valid.

Algorithm : Blind decoding algorithm

```
Inputs: Recovered SSB OFDM grid from SSB signal
Inputs: PBCH and DMRS positions and samples
1    Blind Decoding loop
2    for ISSB ∈ [0, $L_{max}$ -1] then
3        -  Compute corresponding DMRS sequence
4        -  Perform channel estimation and equalization
5        -  Compute LLR values
6        -  Implement de-rate matching
7        -  Decode the input LLRs
8        If CRC is valid then
9            -  Report the decoded MIB and the correct ISSB
10           Break
11       End
     End
```

6.3 PHY Testing

We have implemented a self-checking testbench, we typically use a file-based approach, where the testbench reads in input data and the expected output data from files generated in MATLAB. These files contain the test vectors that we want to use to verify the RTL design.

In the testbench, we apply the inputs from the input file and compare the RTL output with the expected output file. If the outputs match, the test passes; if the outputs do not match, the test fails, and we would need to investigate and correct the RTL design.

It's important to ensure that the input and output files are formatted correctly, and that the RTL design and the reference model use the same input data. Additionally, we consider using multiple test vectors to ensure that the design is working correctly for a range of inputs and not just a single test case.

The inputs to the chain are the half frame received data, timestamp, $N_{ID}^{cell}, i_{SSB}, and\ N_{hf}$. These inputs are randomized in the MATLAB reference model and stored in a file, along with the output from each stage in the MATLAB reference model. The testbench checks each stage, starting from the FFT output to the decoder output payload and the CRC pass signal that are stored in the RIF.

Chapter 7: Conclusion

In this thesis, a 256-point radix-4 FFT is modeled using MATLAB. Then implemented in RTL using the architecture Single Delay Feedback (SDF). Optimization is done by running the 4 stages on only one engine instead of 4 engines. Testing is done using testbench by applying test vectors from the MATLAB model. In addition to that, synthesis is done to check that the timing constraints are met. The post-FFT processing chain has been modeled, implemented in RTL and tested using self-checking testbench with 10000 test cases. The PHY is integrated and tested with self-checking testbench. In the System on Chip, all IPs are integrated with the PHY and the system core through the AHB bus, the PHY application is written in C code, and loaded to the instruction memory and tested.

The upcoming work is testing more test cases for the PHY and complete system verification using UVM as well as FPGA synthesis flow.

Chapter 8: References

[1] 3GPP 38.211, "Physical channels and modulation", Rel-16 Ver 16.10.0, (Accessed Feb 2023).

[2] 3GPP 38.212, "Multiplexing and channel coding", Rel-16 Ver 16.10.0, (Accessed Feb 2023).

[3] 3GPP 38.213, "Physical layer procedures for control", Rel-16 Ver 16.10.0, (Accessed Feb 2023).

[4] 3GPP 38.101, "User Equipment (UE) radio transmission and reception", Rel-16 Ver 16.12.1, (Accessed Feb 2023).

[5] "5G RAN: physical layer implementation and network slicing. Networking and Internet Architecture" Aymeric de Javel. Institut Polytechnique de Paris, 2022. English. ⟨NNT : 2022IPPAT031⟩. (Accessed 10 March 2023).

[6] "A Survey on Pipelined FFT Hardware Architectures" https://link.springer.com/article/10.1007/s11265-021-01655-1 (Accessed Feb 2023).

[7] "Synchronization Procedure in 5G NR Systems" https://ieeexplore.ieee.org/document/8675913 (Accessed Oct 2022).

[8] "Implementing the Radix-4 Decimation in Frequency (DIF) Fast Fourier Transform (FFT) Algorithm Using a TMS320C80 DSP" https://www.ti.com/lit/an/spra152/spra152.pdf (Accessed Feb 2023).

[9] 5G NEW RADIO : Designing For The Future (Ericsson Technology Review) (Accessed Mar 2023).

[10] AREA-DELAY EFFICIENT FFT ARCHITECTURE USING PARALLEL PROCESSING AND NEW MEMORY SHARING TECHNIQUE Yousri Ouerhani, Maher Jridi, Ayman Alfalou (Accessed Jan 2023).