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

By

Ayman Helal

Kareem Ahmed Thabet

Mostafa Mahmoud Abdelkader

Nader Atef

Rahma Aly Mahmoud

Sondos Shabana

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

Ayman Helal

Kareem Ahmed Thabet

Mostafa Mahmoud Abdelkader

Nader Atef

Rahma Aly Mahmoud

Sondos Shabana

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
Giza, Egypt
July 2023

Acknowledgments

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

Table of Contents

List of Tables

List of Figures

List of Symbols and Abbreviations.

NR	New Radio
3GPP	Third Generation Partnership Project
IoT	Internet of things
VR	Virtual Reality
PHY	Physical Layer
MIB	Master Information Block
PBCH	Physical Broadcast Channel
SSB	Synchronization Signal Block
PSS	Primary Synchronization Signal
SSS	Secondary Synchronization Signal
UE	User Equipment
CRC	Cyclic Redundancy Check
AHB	Advanced High-performance Bus
APB	Advanced peripheral Bus
SC	Successive Cancellation
SCL	Successive Cancellation List
RS	Reliability Sequence
PM	Path Metric
LLR	Log-Likelihood Ratio
PE	Processing Element
BRAM	Block RAM
PSN	Partial Sum Network
Sgn	Sign
PS	Part of Stage index number
FFS	First Set bit
i	Decoded bit index
CU	Control Unit
DU	Decision Unit
SM	Sign and Magnitude
DFF	D FlipFlop
SR	Shift Register
LFSR	Linear Feedback Shift Register
FPGA	Field Programmable Gate Array
BD	Blind Decoder
HW BD	Hardware Blind Decoder

Abstract

The 5G technology aims at achieving an enhanced mobile broadband, ultra-reliable and low latency communications, and massive machine-type communications. To achieve these goals, 3GPP has introduced a unified network architecture, with a new physical layer design, namely the New Radio (NR), that supports very high carrier frequencies (mmWaves), large frequency bandwidths, and new techniques such as massive multiple-input and multiple-output (MIMO), and beamforming.

The NR modem includes many basic building blocks such as the carrier scanning, cell selection, physical channels decoding, measurements, and more. In this project, although we consider a system on chip solution, we focus on only one building block within the NR modem. For simplicity, we will treat this building block as a complete independent processor despite that fact that it is one gear in the whole NR system. This processor is basically the full decoding chain for the Physical Broadcast Channel (PBCH) that carries the important Master Information Block (MIB).

The PBCH Decoding processor is a purely digital processor that is customized to decode the PBCH channel within the NR modem. Form the overall system perspective and the microprocessor point-of-view, this processor is employed as a HW accelerator that receives a command from the microprocessor through its register interface. The command is mainly to decode the PBCH channel. However, this command should be associated with various system parameters (such as the cell ID, the time stamp, … etc) that enables the PBCH Decoding processor to decode the PBCH. In return, the PBCH Decoding processor would inform the microprocessor that the processing is complete through an interrupt system and/or a register polling mechanism. The PBCH Decoding processor would save the results in some internal registers that are accessible through the register interface. The results are mainly the MIB payload if the PBCH is decoded successfully, and an indicator whether the PBCH is decoded successfully or not.

The PBCH Decoding processor implements the typical receiver chain for the PBCH detection. This processor has some mandatory building blocks including the FFT, channel estimation and equalization, demodulation process, and the channel decoding stage. One important feature about this processor is its ability to control the various building blocks within the blind decode trials.

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 can connect billions of devices that share information in real time while providing stable and reliable connectivity. It introduces a massive shift in applications that require secure and reliable real time connectivity such as the Internet of Things (IoT), virtual reality (VR), and many more [1].

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, 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 broadcasted 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 3 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 conveys 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 part in the 5G network connection and is said to be the highest-priority data block in the 5G network and is defined as the first information element in a message [2].

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 are treating it for simplicity as an independent processor. This processor represents the full decoding chain of the MIB payload found in the PBCH. It consists of 3 main subsystems: FFT subsystem, post-FFT subsystem and the Decoder Subsystem.

1.2 Organization of the Thesis

In this thesis, our main focus is the decoding process (Polar decoders) used in NR technology including the cyclic redundancy check (CRC) operation.

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 a literature survey of polar codes and polar decoders in specific the successive cancellation (SC) and successive cancellation list (SCL) decoders. Then, introduces the 2 implemented decoders model in detail and discusses their operation including CRC operation in addition to some hardware architectures found in the literature. It also presents the work done to select the appropriate decoder and the fixed-point analysis operations.

Chapters 4:8 discuss the implemented hardware architecture and the steps followed to obtain an optimum design. Chapter 9 introduces the system integration including the blind decoding process and how the processor operates to successfully decode the MIB payload. Finally, the conclusion and the possible future work are stated in Chapters 11:12.

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, Fig. 2.1.

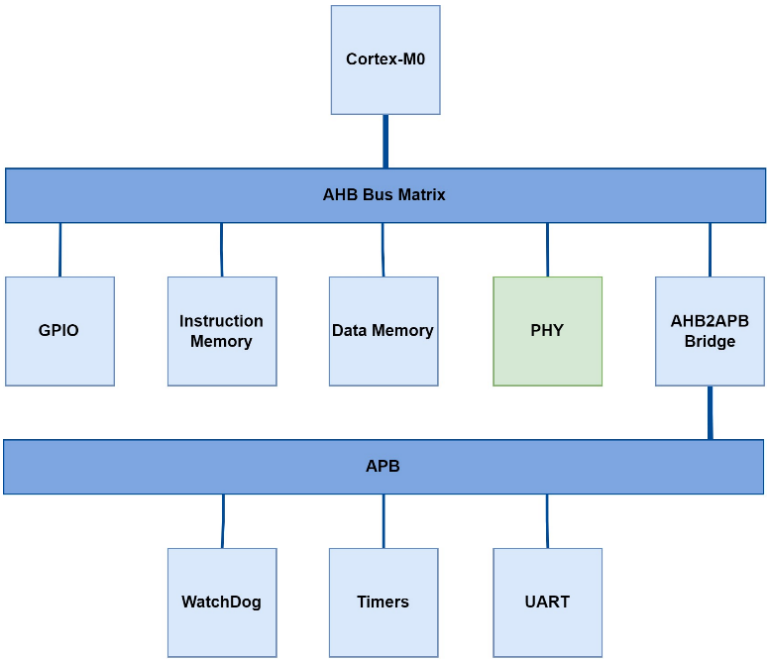


Figure 2.1 - Full System Block Diagram

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 Fig. 2.2. We discuss each of the main blocks of this diagram in the following sub-sections.

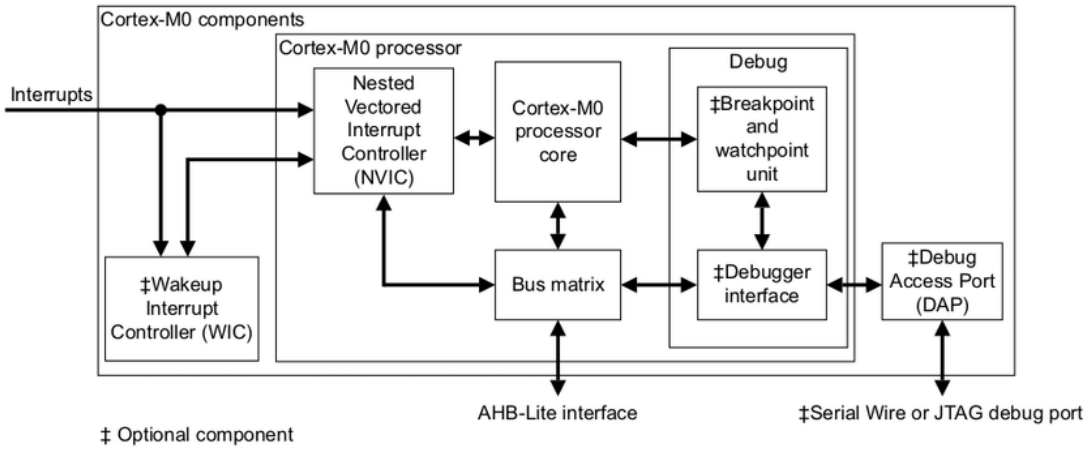


Figure 2.2 - Cortex-M0 Block Diagram

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 (2-1).

Table 2.1: 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*

2.2.2.1 Interruput Set-enable Register

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

Table 2.2: 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.

2.2.2.2 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 (2-3).

Table 2.3: 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.

2.2.2.3 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 (2-4).

Table 2.4: 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.

2.2.2.4 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 (2-5). This means the number of registers is implementation-defined and corresponds to the number of implemented interrupts. These registers are only word accessible.

Table 2.5: 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

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		Reserved	
6			
5			
4			0x10
3	-13	HardFault	
			0x0C
2	-14	NMI	
			0x08
1		Reset	
			0x04
		Initial SP value	
			0x00

The vector table contains the reset value of the stack pointer, and the start addresses, also called exception vectors, for all exception handlers. Table (2-6) 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.

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 (2-7)) 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 Fig. 2.3, 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 2.6: 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

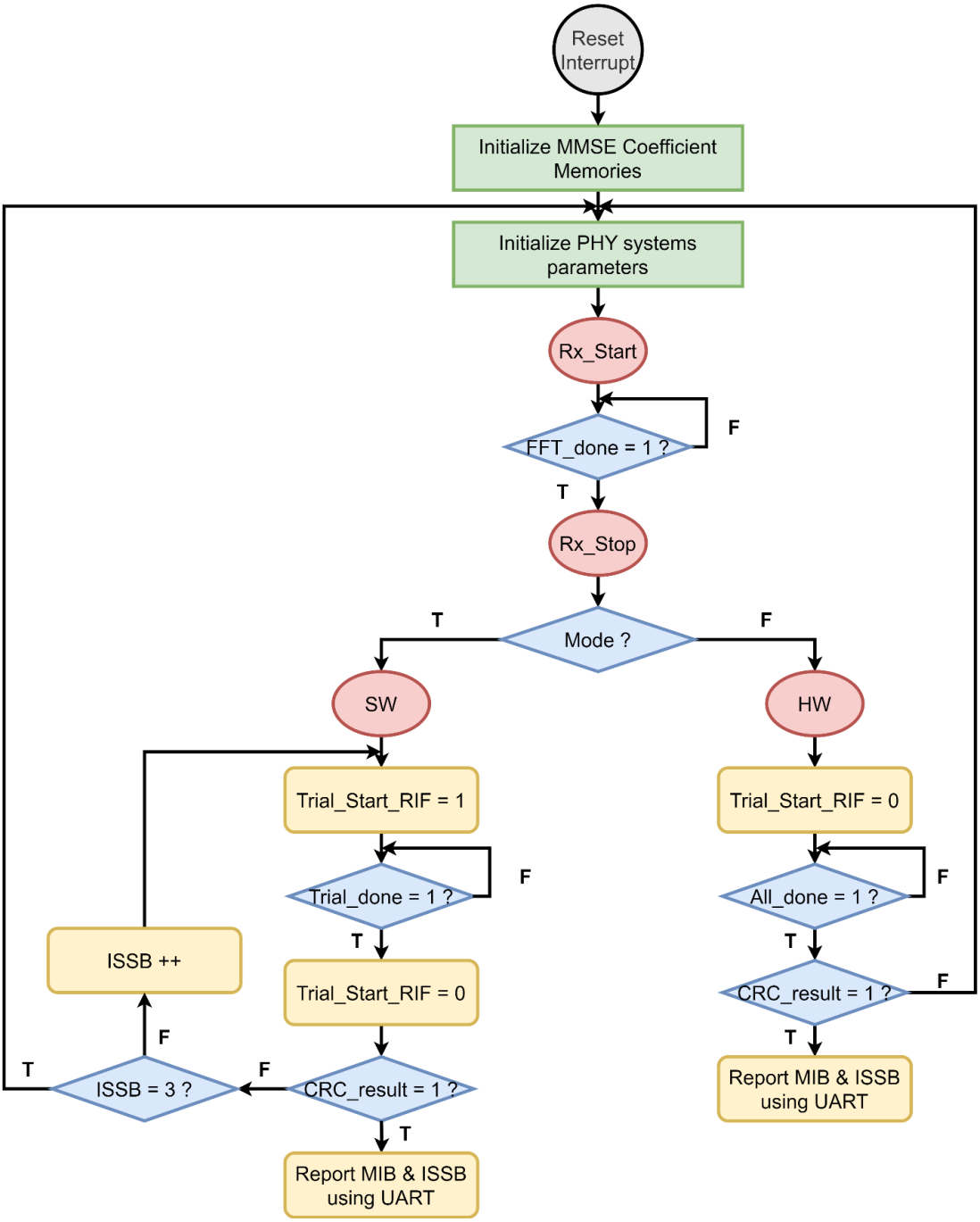


Figure 2.3 - Flow Chart of PHY Program

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 index is 0, the iteration fails as shown in Fig. 2.4.

- When the iSSB index is 1, the iteration is passed as shown in Fig. 2.5, since this is the value of iSSB index 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 2.5 - 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 2.4 - 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 1.

A section of the block diagram of the bus matrix is presented in Fig.2.6, where the input and output signals are shown, and the description of the main signals is shown in table (2).

Table 2.7: 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.

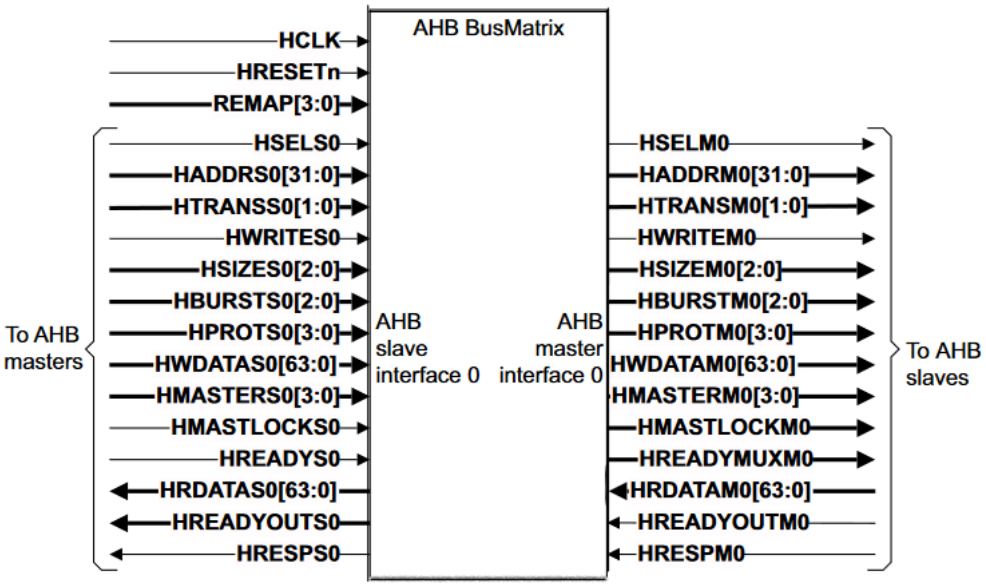


Figure 2.6 - Part of the AHB Bus matrix block diagram

Table 2.8: 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, Fig. 2.7

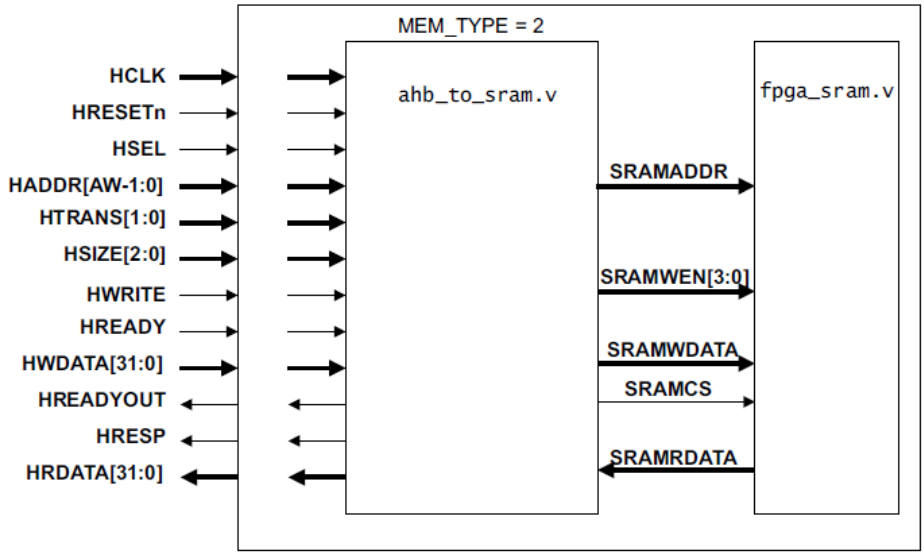


Figure 2.7 - Memories Block diagram showing the interface with the bus matrix

- 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.

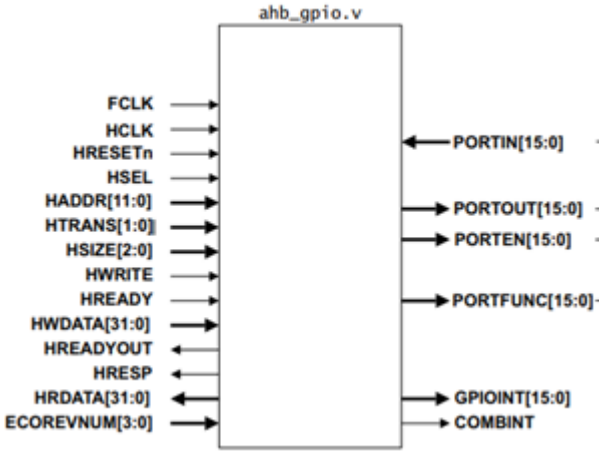


Figure 2.8 - GPIO interface

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 through out a multiplexing network for each bit as shown in Fig. 2.9. 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.

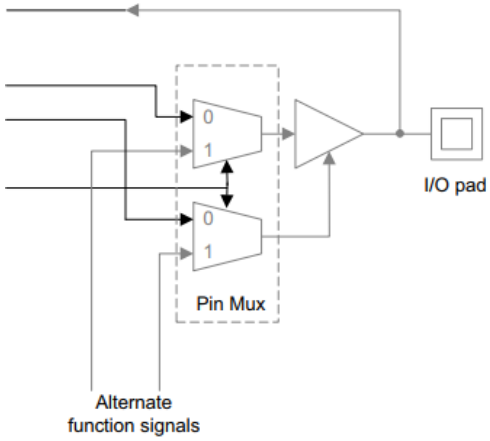


Figure 2.9 - GPIO Alt. function

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, Fig. 2.10:

- 3 main building Subsystems: FFT subsystem, post-FFT subsystem and the decoder subsystem.
- Controller: it implements the blind decoding algorithm (discussed in chapter 9).
- Register interface: an interface to the AHB bus matrix that connects the PHY to the rest of the system (discussed in chapter 9).
- Memories.

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

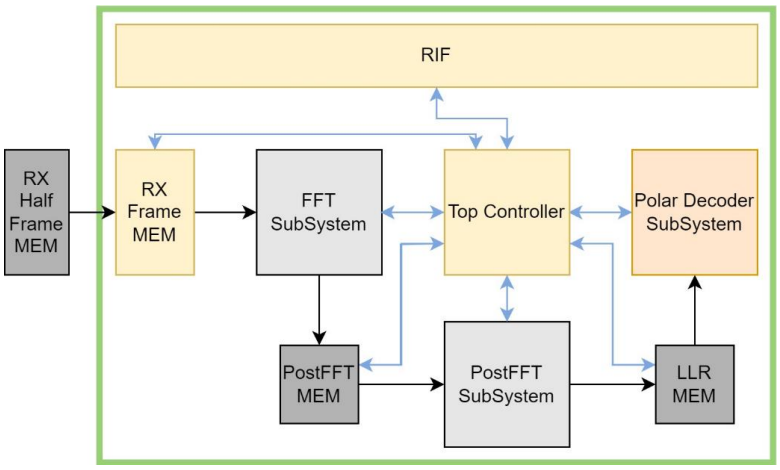


Figure 2.10 - PHY Block Diagram

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 Fig. 2.11 where table explains some of the important output signals.

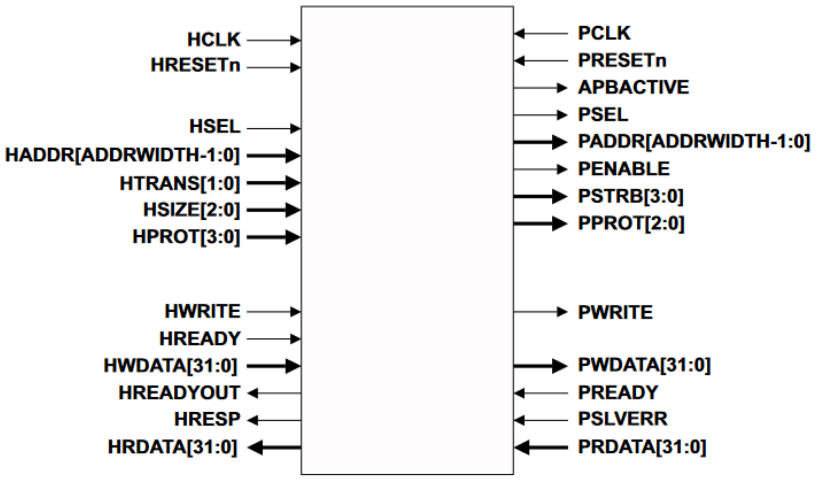


Figure 2.11 - AHB to APB bridge block diagram

Table 2.9: 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.

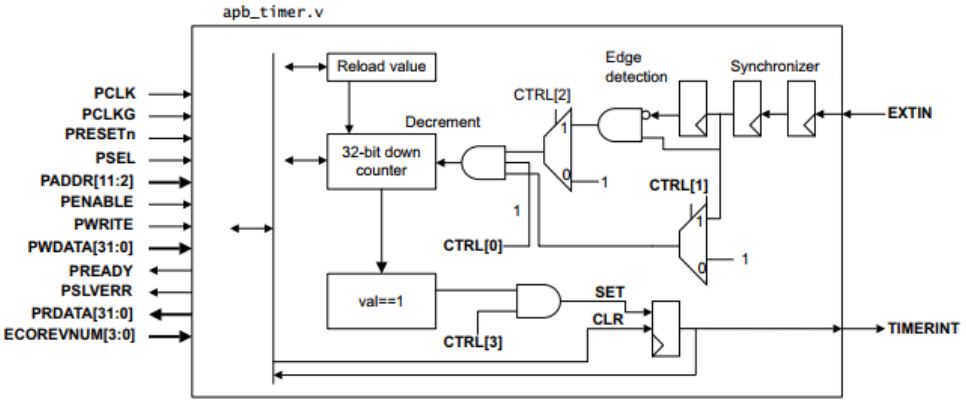


Figure 2.12 - APB Timer

2.5.1.1 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.

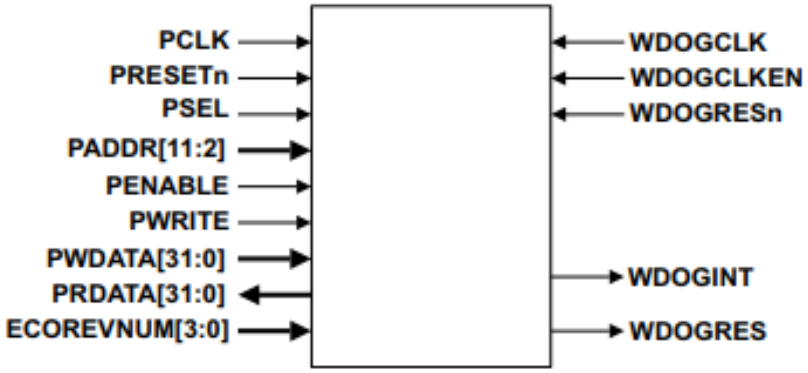


Figure 2.13 - APB Watchdog

2.5.2.1 Access WDT Peripheral

1. Enable APB 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.

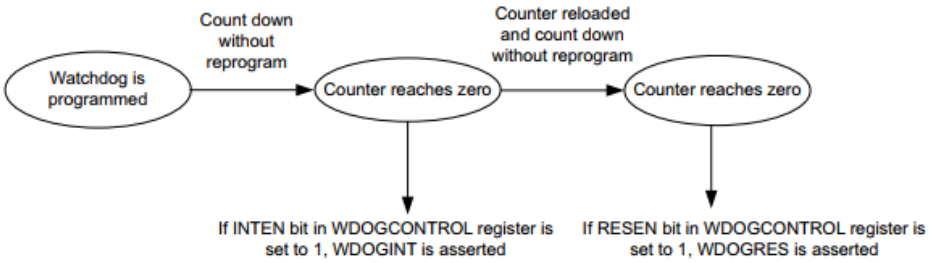


Figure 2.14 - Watchdog timer flow diagram

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 Fig. 2.15.

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 2.155

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.

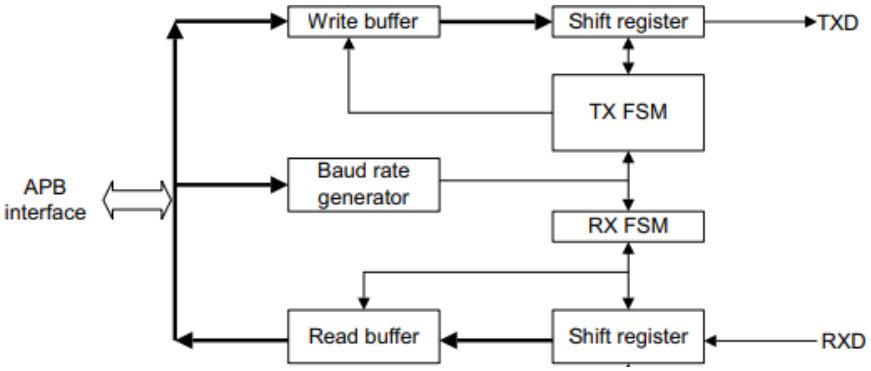


Figure 2.15 - APB UART

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.

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 Fig. 2.16.

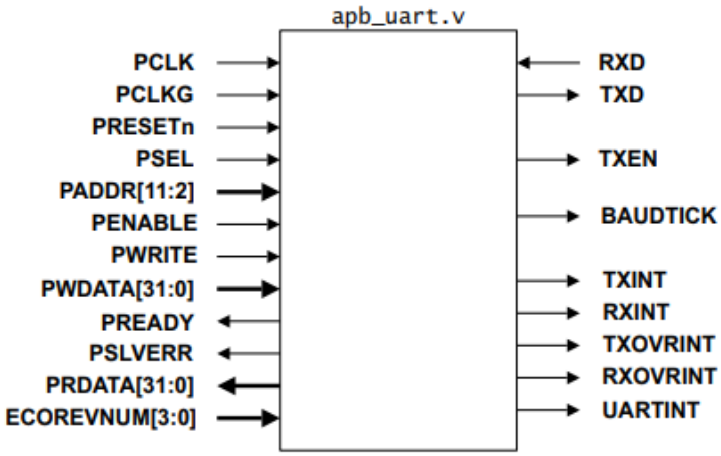


Figure 2.16 - UART block

Chapter 3: Polar Decoder Modeling.

3.1 Introduction

In this chapter, we will discuss the implemented polar decoder models, but first we will briefly talk about polar codes history then explain the basic operations of the polar encoder and the CRC operation using illustrative examples. Then, we will go into the details of the implemented algorithms while comparing their performance with an existing paper. Finally, we will explain the fixed-point analysis done on the selected model and the trials done to achieve the optimum word length.

3.2 Literature Survey

Polar codes, introduced by Erdal Arıkan in 2009 [3], are a class of error-correcting codes that have gained significant attention in recent years due to their remarkable performance and low encoding and decoding complexity. These codes have found widespread applications in various communication systems, including the fifth-generation New Radio (5G NR) technology. This literature survey explores the fundamental concepts of polar codes, polar decoders, and their role in 5G NR technology, specifically the Physical Broadcast Channel (PBCH).

Polar codes have been shown to have the capacity achieving property. They can achieve the capacity of any discrete memoryless channel with a small error probability. This makes them a strong candidate for use in modern communication systems, especially in the 5G NR standard.

The code construction is based on multiple recursive concatenations of a short kernel code which transforms the physical channel into virtual outer channels of varying capacity. Polar codes have modest encoding and decoding complexity, which renders them attractive for many applications.

One of the main advantages of polar codes is their simplicity. Unlike turbo codes or low-density parity-check (LDPC) codes, polar codes do not require any iterative

decoding process. Instead, they can be decoded using a simple successive cancellation (SC) decoder having linear complexity.

There are several decoding techniques for polar codes, including:

- Successive Cancellation (SC) Decoding: This is the original decoding algorithm proposed by Arıkan for polar codes. It is a low-complexity algorithm that uses a recursive structure to cancel out the effect of previously decoded bits. The main disadvantage of this method is its error floor, which means that its performance degrades at low signal-to-noise ratios (SNRs).
- Successive Cancellation List (SCL) Decoding: This is an extension of SC decoding that keeps a list of the most likely paths and uses a voting scheme to select the final decision. It offers better performance than SC decoding, especially at low SNRs, but at the cost of increased complexity [4].

Polar codes are chosen for the control channel of the enhanced mobile broadband scenario (eMBB) in the 5G standardization process of the 3GPP. For downlink control information, polar codes are concatenated with distributed cyclic redundancy check (CRC), whose bits are obtained by interleaving the bits between the CRC encoder and the polar encoder.

In recent years, many researchers have focused on improving the performance of polar codes by using various channel coding techniques such as concatenation, puncturing, and shortening.

3.3 Preliminaries

In this section, we describe the fundamental concepts necessary to understand the operation of polar decoders such as the CRC operation, the general encoding process and the encoding specs used in 5G standard for PBCH.

3.3.1. Polar Encoder

Polar encoder is a type of block code that takes K input bits and encodes them into N bits word, where N > K, Fig. 3.1. K can be of any integer value, but N must be an even number, why? We will see that soon.

Figure 3.1 - Polar Block Code

The idea of Polar encoder is that it takes the K bits and adds (N-K) frozen bits to them to make the total number of bits equal to N bits. Then by performing some operations on the N bits, we can obtain the final encoded word.

The first step of the encoding process is putting the frozen bits and data bits at certain indices dependent on reliability sequence (RS). The reliability sequence (RS) is a sequence that contains the reliability of each subchannel, these reliabilities are computed offline, and the ordered sequence is stored for a maximum code length. [5]

The RS is defined in the 3GPP standard and is used to know which channel to transmit data and frozen bits on. Hence, it contains the indices of the channels ordered from the worst channels (Least reliable) to the best ones. After selecting the N first channels in the sequence, we put the data bits at the indices of the best K channels and put the frozen bits at the remaining channels, where the frozen bits value is zero. In the following paragraphs we will present an example to further illustrate the polar encoder operation. [5]

3.3.1.1. Encoding Process using Tree Diagrams

Assume that the data that we want to encode is (1011) which means that K = 4, and we want to encode it into 8 bits code word then N = 8, this means that the number of the frozen bits is 4 (N-K). Now, we want to determine our reliability sequence to know where to put our data and frozen bits. From Fig.3.2, we can see a section of the RS provided by the 3GPP, by selecting the first 8 channels which are (1,2,3,4,5,6,7,8), we obtain their order as follows: Channels order = [1 2 3 5 4 6 7 8] [6].

```
Reliability_Sequence = [1 2 3 5 9 17 33 4 6 65 10 7 18 11 19 129 13 34 66 21 257 35 25 ..
                        37 8 130 67 513 12 41 69 131 20 14 49 15 73 258 22 133 36 259 ...
```

Figure 3.2 - Part of the reliability sequence provided by the 3GPP

From this, we can conclude that channels (4,6,7,8), the last 4 channels, are the best and they are reserved for the data bits, while the remining 4 channels are filled with zeros (frozen bits) [6].

This means that the code word will consist of N = [F F F D F D D D] where F means frozen and D means data. Hence, N = [0 0 0 1 0 0 1 1], after that we start doing some calculations to get the final code word. To illustrate these calculations, we explain it using tree diagram and to do so, we follow the rules presented in Fig. 3.3.

Where v_y^x: v means vector, x means the number of elements in the vector and y means the index of the vector. The steps to obtain the encoded codeword are explained by following the rules in the previous figure:

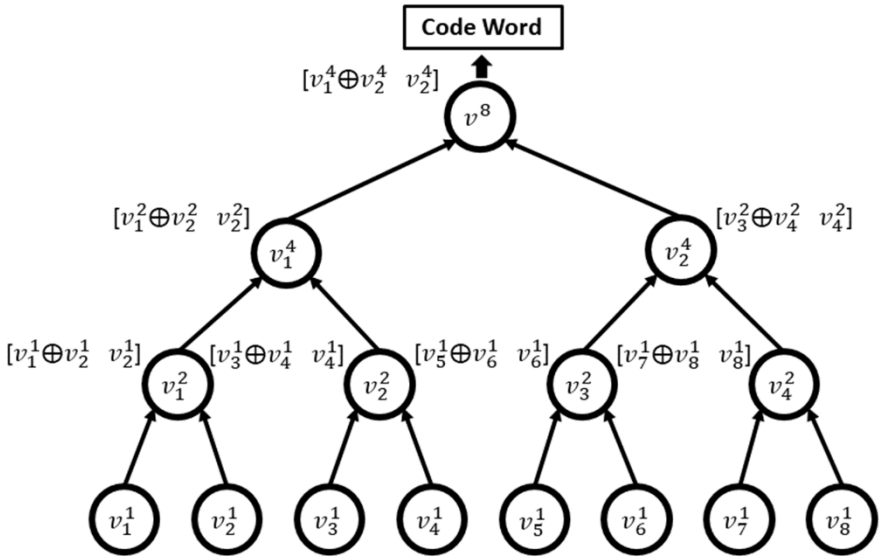


Figure 3.3 - Encoding Tree diagram containing the encoding rules.

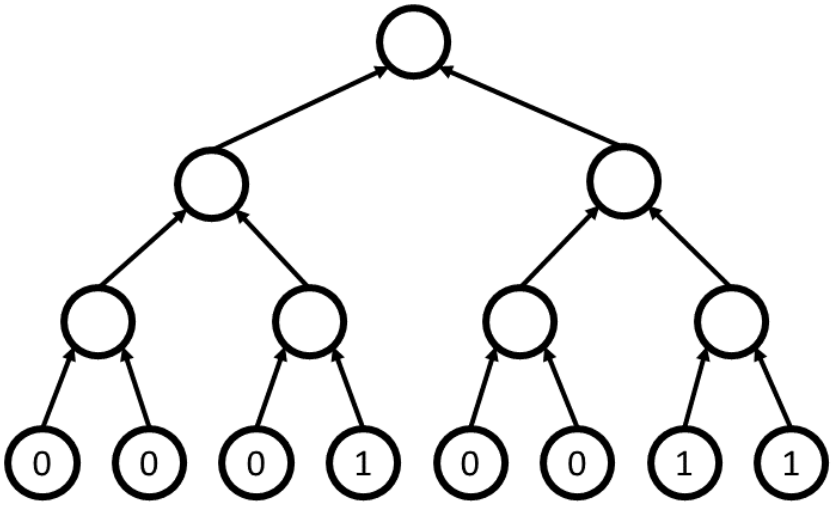


Figure 3.4 - Step 1 of the encoding process

Step 1: Put elements of vector N in the leaves of the tree diagram, Fig. 3.4.

Step 2: Propagate through the tree diagram from the leaves to the root, then calculate the parent nodes of each leaf node. Each node, from the parent nodes, consists of 2 bits, the 1st bit is calculated by XORing the 2 child nodes and the 2nd bit is the right child.

Then $v_1^2 = [0 \oplus 0, 0] = [0,0]$, $v_2^2 = [0 \oplus 1,1] = [1,1]$ and so on. So, the leaf nodes' parents are updated, Fig. 3.5.

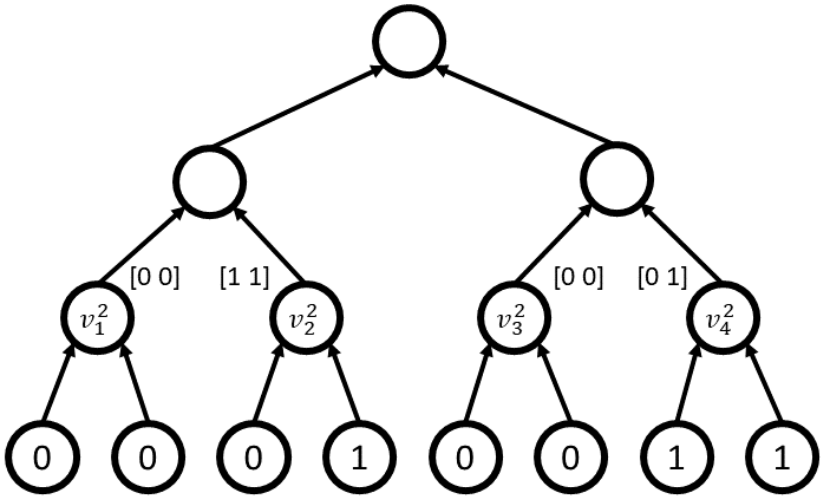


Figure 3.5 - Step 2 of the encoding process

Step 3: Repeat step 2 but with a larger size of child and parent nodes vectors. Half of the parent vector is obtained by XORing the 2 child nodes vectors, and the other half equals to the right child vector.

Then $v_1^4 = \left[[0\ 0] \oplus [1\ 1], [1\ 1]\right] = \left[[1\ 1], [1\ 1]\right] = [1,1,1,1]$ and so on. So, the parent nodes are updated, Fig. 3.6.

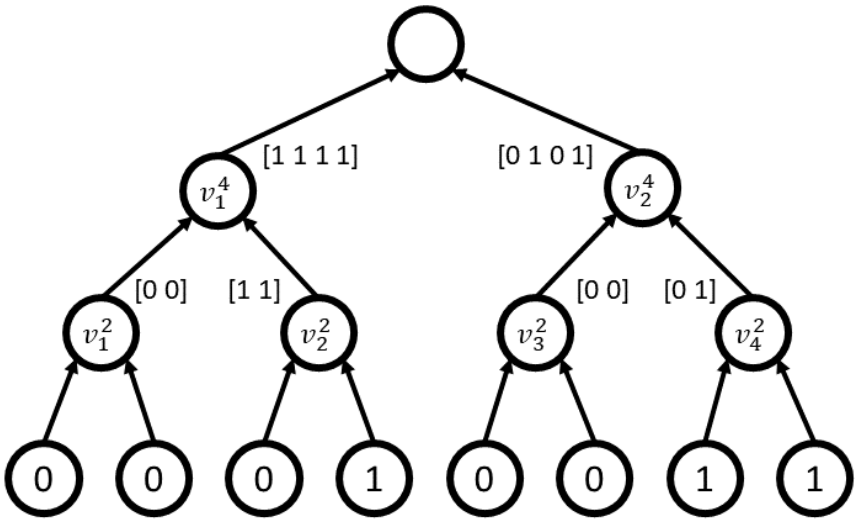


Figure 3.6: Step 3 of the encoding process.

Step 4: Similar to step 3 but with vectors double the size of those in step 3. In this step, we reach the root node, which is the final step to get the final code word that we send through the channel.

Then $v^8 = \left[[1\ 1\ 1\ 1] \oplus [0\ 1\ 0\ 1], [0\ 1\ 0\ 1]\right] = \left[[1\ 0\ 1\ 0], [0\ 1\ 01]\right] = [1,0,1,0,0,1,0,1]$ and this is the final code word, Fig. 3.7.

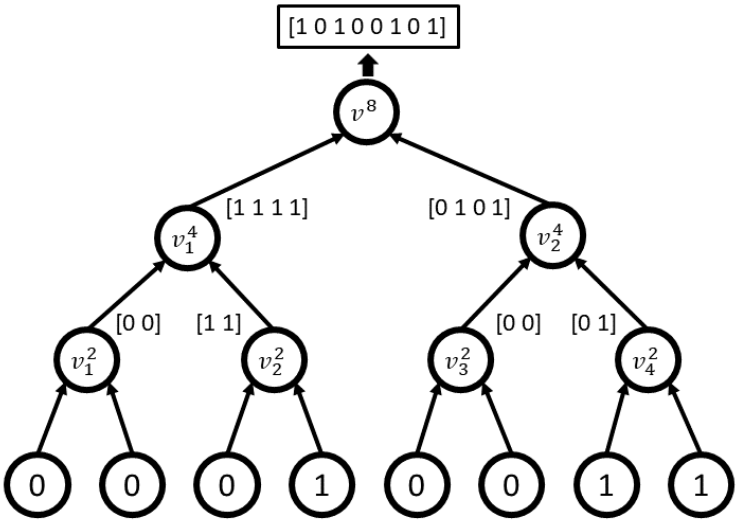


Figure 3.7 - Step 4 of the encoding process

After encoding the data, we can transmit it through the channel. The encoding process is done to increase the error correcting capabilities and decrease the probability of error caused by the channel to our data while transmitting it [6].

From this example, we can infer that the depth of the tree is calculated by $\log_2 N$, and in the given example, the depth of the tree is $\log_2 8 = 3$ (excluding the root). Now, we can understand why N must be even, because to apply these tree diagram rules, we need the number of leaf nodes to be even, it can't be odd.

This method is used to illustrate how the polar encoder works and is not the main encoding process nor the method that we used to implement the polar encoder algorithm. Hence, we will explain how the main method works [6].

3.3.1.2. Polar Encoder Main Method

This method uses matrices multiplication, Kronecker product and a generator matrix G to encode the K data bits into N bits code word. The idea is to generate an NxN permutation matrix, by doing Kronecker product to G matrix (n-1) times, where n = $\log_2 N$. By multiplying the vector N in the previous example of tree diagram with the permutation matrix, we can get the final code word [7].

First, we prepare the N vector as we've done in the previous example, N = [0 0 0 1 0 0 1 1]. Second, we prepare the permutation matrix P and by assuming the same assumptions in the previous example, then n = 3. By doing Kronecker product 2 times to G matrix, we can get P matrix.

$$G = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \qquad P = G \otimes 2 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^{\otimes 2}$$

Kronecker product is done by extending the G matrix from the size 2x2 to 4x4 then to 8x8. At each extension, the matrix rows and columns are extended by 2. This is done by copying the 2x2 G matrix at the location of the elements that are equal to 1 and putting a 2x2 zero matrix at the location of the elements that are equal to 0 [7].

$$M = G \otimes G = G \otimes \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} G & 0 \\ G & G \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

To obtain P, repeat the last step again, but this time, do the Kronecker product between M and G.

$$P = G \otimes M = G \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} G & 0 & 0 & 0 \\ G & G & 0 & 0 \\ G & 0 & G & 0 \\ G & G & G & G \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The last step is to multiply N vector with P matrix to get the final code word, as the following:

$$code\ word = NXP = (0\ 0\ 0\ 1\ 0\ 0\ 1\ 1)_{1x8} X \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}_{8x8}$$

$$= (1\ 0\ 1\ 0\ 0\ 1\ 0\ 1)$$

We can also get the code word by binary adding the 4th, 7th and 8th rows in the P matrix. These rows correspond to the one's elements in the N vector.

This is the main method of Polar encoder which was presented by Arikan and we used it to implement the Polar encoder model, since dealing with matrices is much easier than dealing with trees that need recursion in their implementation. The explained Polar encoder algorithm is shown in **Algorithm 1**. [7]

Algorithm 1: Polar encoder function algorithm

```
1  Function Code_Word = Polar_ENC_func(K,N,Data)
2    calculate n
3    initialize G matrix
4    Extract RS and determine data and frozen indices
5    initialize U vector with zeros and j = 0 // where U vector length is N and j is a
       data bits counter
     // U vector is like N vector that we talked about in the previous examples
6    for i = 1 to N do    // in this for loop, we prepare the U vector
```

```
7       if i is from Data_indices then
8           j = j + 1
9           U(i) = Data(j)
        end
    end
10  initialize P matrix with G
11  for i = 1 to n-1 do    // in this for loop, we prepare the P matrix
12      P = kron(P,G)      // do Kronecker product between G and P matrices then
        update the P matrix
    end
13  Code_Word = mod(U*P,2)  //multiplying U and P, gives a vector with decimal
    numbers
     // so, to convert it to binary, take modulus of 2 of the resulted vector
End
```

3.3.2. Cyclic Redundany Check (CRC)

Cyclic redundancy check (CRC) is an error detecting code commonly used in digital communications and storage devices to detect accidental errors to digital data. CRC plays a critical part in ensuring data integrity and reliability by identifying unintentional modifications or faults that could happen during transmission or storage. It enables the receiver to confirm the accuracy of the data received by adding a checksum to the data.

This method creates a different checksum by using polynomial division-based mathematical procedures. The sender computes the CRC checksum and appends it to the data during transmission, and the receiver recalculates the checksum after receiving the data. The data is regarded as accurate and proper if the calculated and received checksums agree.

3.3.2.1. CRC at transmitter side

CRC generation at transmitter side is as follows:

First: determine length of divisor which is L + 1 bits, Where L is number of CRC bits. The CRC length (L) in our system is equal to 24 bits and the divisor expressed as polynomial expression equals to $x^{24} + x^{23} + x^{21} + x^{20} + x^{17} + x^{15} + x^{13} + x^{12} + x^8 + x^4 + x^2 + x + 1$ according to 3GPP standard [8].

Second: append L zero bits to the transmitted message (MIB) whose length is K bits, where K equals to 32 bits according to 3GPP standard, So A length is K + L zero bits as shown in Fig. 3.8.

Third: perform a binary division operation, this operation is illustrated in **Algorithm** 2.

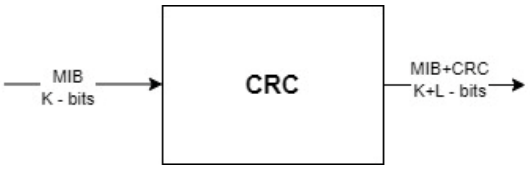


Figure 3.8 - CRC block at Tx side

After performing binary division operation, we get a reminder and a quotient. The reminder is the CRC bits, and its size is equal to L bits, and they are appended to MIB payload. Finally output of CRC block is a message whose length is K + L bits.

Algorithm 2: CRC generation at transmitter side

```
1  Data: Message // input K – bits data
2  Data: Divisor // L + 1 bits Divisor
3  Data: CRC_output // K + L output data bits
4  Function CRC_func (Message, K, L, Divisor)
5      M_CRC = concatenate (Message, zeros(1,L)) // append L zero bits to
       Message
6      Intermediate_data = M_CRC
7      quotient = [ ] // empty vector
8      for i = 1 to K do
9          MS_bit = intermediate_data(1) // Most significant bit
10         quotient = concatenate (quotient, MS_bit)
11         if MS_bit = 1 then
12             Intermediate_data = xor (intermediate_data,Divisor)
13             if i ≠ K then
14                 intermediate_data = concatenate
                   (intermediate_data(2:end),M_CRC(i+L))
15             end
16         else
17             intermediate_data = xor (intermediate_data,zeros(1,L))
18             if i ≠ K then
19                 intermediate_data = concatenate
                   (intermediate_data(2:end),M_CRC(i+L))
20             end
21         end
22     end
23     CRC_bits = intermediate_data(2:end)
24     CRC_output = concatenate (Message,CRC_bits) // append CRC bits to
       Message
25 end
```

3.3.2.2. CRC at Receiver Side

At the receiver side, CRC plays a crucial role in ensuring the integrity of data transmission. Upon receiving the data, the receiver performs the same binary division operation calculation as at the transmitter side using the same predetermined polynomial expression.

The received decoded K data bits along with the appended CRC bits, is divided by the generator polynomial as shown in Fig.3.9. This operation is illustrated in **Algorithm 3**. If the resulting remainder is zero, it indicates that the data has been transmitted without any errors.

However, if a non-zero remainder is obtained, it signifies the presence of errors during transmission. In such cases, the receiver can request retransmission of the data to ensure accuracy. By employing CRC at the receiver side, data integrity can be effectively validated, enhancing the reliability of the communication system.

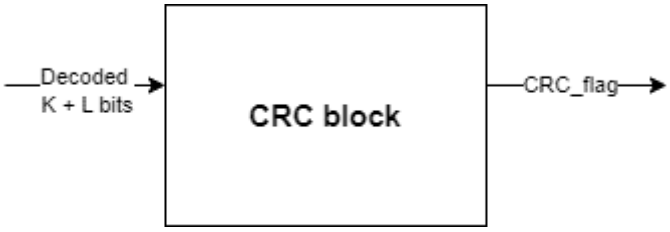


Figure 3.9 - CRC Block at Rx Side

Algorithm 3: CRC check at receiver side

```
1  Data: Message // input K + L – bits data
2  Data: Divisor // L + 1 bits Divisor
3  Data: flag // flag indicates if there is an error
4  Function CRCcheck_func (Message, K, L, Divisor)
5      M_CRC = Message
6      Intermediate_data = M_CRC
7      quotient = [ ] // empty vector
8      for i = 1 to K do
9          MS_bit = intermediate_data(1) // Most significant bit
10         quotient = concatenate (quotient, MS_bit)
11         if MS_bit = 1 then
12             Intermediate_data = xor (intermediate_data,Divisor)
13             if i ≠ K then
14                 intermediate_data = concatenate
                   (intermediate_data(2:end),M_CRC(i+L))
15             end
16         else
17             intermediate_data = xor (intermediate_data,zeros(1,L))
18             if i ≠ K then
19                 intermediate_data = concatenate
                   (intermediate_data(2:end),M_CRC(i+L))
20             end
21         end
22     end
23     CRC_bits = intermediate_data(2:end)
24     Check if all CRC bits equal zero
25 end
```

3.4 Successive Cancellation (SC) Decoder

As discussed before, Arıkan constructed the first class of error correcting codes that provably achieve the capacity of any symmetric binary-input discrete memoryless channel (B-DMC) with efficient encoding and decoding algorithms based on channel polarization.

He proposed a low-complexity successive cancellation (SC) decoder and proved that the block-error probability of polar codes under SC decoding vanishes as their block-length increases.

3.4.1 Channel Splitting

Consider a Bit Erasure Channel (BEC), as shown in Fig.3.9, that has an erasure probability equals to ϵ and channel capacity C equals to $1 - \epsilon$. [4]

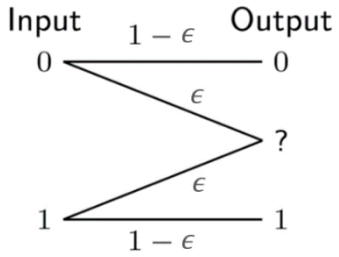


Figure 3.10: Binary Erasure Channel

After combining independent B – DMC channels and splitting into N synthetic channels $W_N$with varying reliabilities where N is channel block length equals to 512. As shown in Fig. 3.11, We split the combined channel into 2 synthetic channels $W_2^{(-)}(y_0, y_1|u_0)$, $W_2^{(+)}(y_0, y_1, u_0|u_1)$. $W_2^{(-)}$ has y_0, y_1output of its channel and can decode u_0 . While $W_2^{(+)}$has $y_0, y_1$and u_0, so we can decode u_1 by the knowledge of u_0. These 2 channels are theoretical channels not physical constructs. They are mainly defined by the order of how to decode the bits [4].

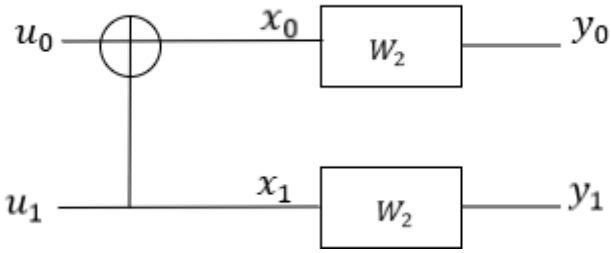


Figure 3.11 - Two synthetic channels

To calculate the bit erasure probability for the 2 channels. First, the $W_2^{(-)}$ channel has 2 inputs y_0, y_1 and has input equal to u_0. So, to decode this channel, we will go to decide what is the u_0 transmitted by knowing $\widehat{u_0}$. All the possible combinations for the $W_2^{(-)}$ channel are shown as in Fig.3.12 and all the possible combinations for the $W_2^{(+)}$ channel are shown as in Fig.3.13 [4].

In BEC, any value XORed with erasure gives erasure. In $W_2^{(-)}$, we can decode correctly only in the first combination, while others give incorrect decision. So, the bit erasure probability for $W_2^{(-)}$ equals to $p_b = 1 - (1 - \epsilon)^2 = 2\epsilon - \epsilon^2 \geq \epsilon$. While in $W_2^{(+)}$, we have 3 outputs and u_0 always received correctly, so this leads to one incorrect case lead to an erasure. The bit erasure probability for $W_2^{(+)}$ equals to $p_b = \epsilon^2 \leq \epsilon$. [4]

$$Avg.\,Bit\ Erasure\ Probability = \frac{1}{2}(2\epsilon - \epsilon + \epsilon^2) = \epsilon$$

The Average bit erasure probability and capacity are preserved. The bit erasure channel of the first channel is greater than erasure probability, while the second channel has less erasure probability that's why its takes symbol $W_2^{(+)}$ as it better and more reliable channel than $W_2^{(-)}$, less reliable channel [4].

$W_2^{(-)}$:

y_0	y_1	$\hat{u}_0$
$u_0 \oplus u_1$	u_1	u_0
?	u_1	?
$u_0 \oplus u_1$	?	?
?	?	?

Figure 3.12 - $W_2^{(-)}$ channel outputs

$W_2^{(+)}$:

y_0	y_1	u_0	$\hat{u}_1$
$u_0 \oplus u_1$	u_1	u_0	u_1
?	u_1	u_0	u_1
$u_0 \oplus u_1$	?	u_0	u_1
?	?	u_0	?

Figure 3.13 - $W_2^{(+)}$ channel outputs

3.4.2 Decoder Core Description

As mentioned before, The SC is a sequential decoder. It decodes bit by bit, for example we decode the channel $W_2^{(-)}$ Immediately as its output are ready y_0, y_1. But for the channel $W_2^{(+)}$, we cannot decode it immediately as we are waiting for the estimated bit of the previous channel $\hat{u}_0$. So, we cannot decode the bit index i until all the previous bits from$0\ to\ i-1$ are estimated and decoded.

The decoder receives from the channel N LLRs (log likelihood ratio) which can be calculated by the following equation $LLR_i = \log\left(\frac{\mathrm{p}(\mathrm{y_i}|x_i = 0)}{\mathrm{p}(\mathrm{y_i}|x_i = 1)}\right)$ where i is the bit index from 0 to N-1 [4].

The SC Decoder takes these channel LLRs and perform some processing on these inputs till it reaches the estimation of the bits; as the SC Decoder is soft decision. The Decoder operation can be represented using a tree diagram representation, as shown in Fig.3.14, showing how the processing on the LLRs take place until reach the leaf nodes where final decision of the bits takes place.

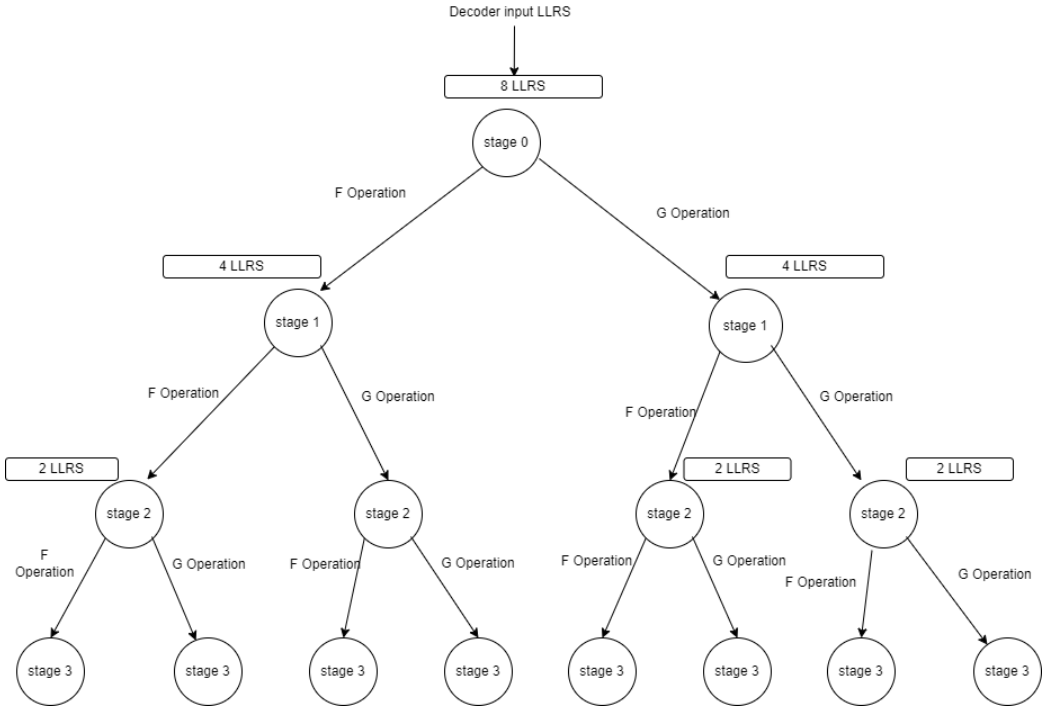


Figure 3.14 - SC tree diagram

3.4.2.1 Basic Building Blocks N = 2

Considering N = 2, after receiving 2 LLRs from the channel. First, we go to the left child node do a f operation as shown in Fig.3.15 The f operation is expressed as in equation 3.1 [4].

$$f(LLR_1, LLR_2) = \ tanh^{-1}(2 * tanh\frac{LLR_1}{2} * tanh\frac{LLR_2}{2}) \qquad (3.1)$$

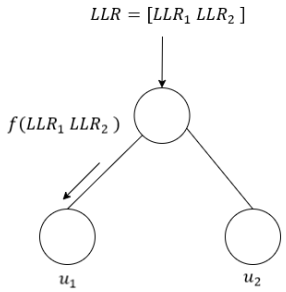


Figure 3.15: f operation on left child node N = 2

The previous f function which involves exponentiations and logarithms. For a hardware implementation of the SC decoder this function is hard to be implemented so, the f function replaced by another approximation hardware easy implementation such an approximation is called the “min-sum approximation” of the decoder as in equation 3.2 [9].

$$f(LLR_1, LLR_2) = sign(LLR_1) * sign(\ LLR_2) * min(|LLR_1|, |LLR_2|) \qquad (3.2)$$

The f function produce a LLR value depend on its sign we decide the value of the bit u_1 [4].

$$u_1 = \begin{cases} 1 \ \ if \ LLR_i < 0 \ and \ u_1 \in A \\ 0 \ \ if \ LLR_i > 0 \ and \ u_1 \in A \\ 0 \ \ if \ u_1 \notin A \end{cases} \tag{3.3}$$

Where A is set contain information bit.

Second, after estimating value of u_1, we return to the upper node with the estimated bit $\hat{u}_1$ and go to the right child node and do a g operation as shown in Fig. 3.16. The g operation is expressed as in equation 3.4. The g function produce a LLR value depend on its sign we decide the value of the bit u_2 [9].

$$g(LLR_1, LLR_2, \hat{u}_1) = (-1)^{\hat{u}_1} * LLR_1 + LLR_2 \tag{3.4}$$

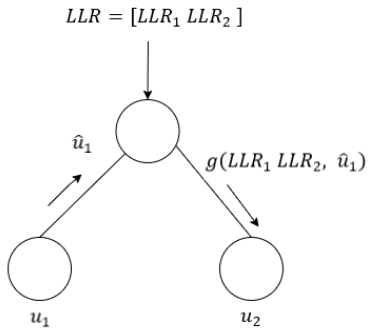


Figure 3.16 - g operation on right node N = 2

After estimating the second bit, we are done estimating all the bits but there is a final step which isn't meaningful in case N = 2. As the root node have the hard decisions from its child nodes, now the root node can decide the return back operation which is expressed in equation 3.5 and shown in Fig.3.17 [9].

$$\hat{u} = [\hat{u}_1 + \ \hat{u}_2 \ \ \hat{u}_2] \tag{3.5}$$

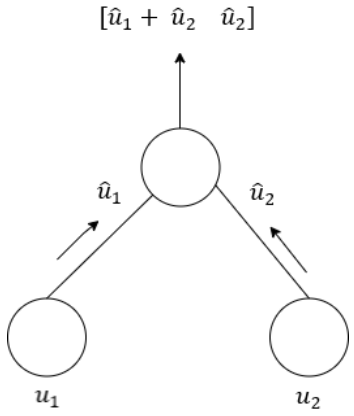


Figure 3.17 - Return back operation N = 2

3.4.2.2 Operation at any interior node

The above section is for decoding block length N = 2, so let's discuss the operation at any interior node and any block length N. The interior node here is any node in the tree except for the leaf nodes [9].

First, any interior nodes can receive several LLRs, for example if the number of LLR received at a node is M. The parent will send this M LLRs to the left child node to do the f operation. The result of the f operation is M/2 LLRs as f operation inputs are $f(LLR_{1:\frac{M}{2}}, LLR_{\frac{M}{2}+1:M})$, so it takes the 2 corresponding LLRs in each vector and do the minsum approximation as shown in Fig. 3.18 [9].

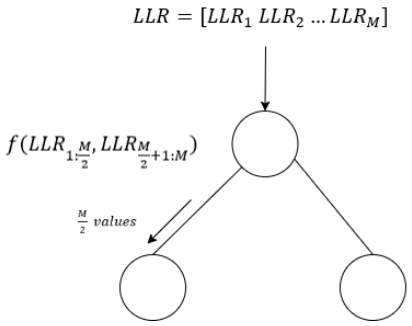


Figure 3.18 - Left node f operation

Second, the parent node takes the decision bits from the left node whose number are M/2 bits and pass them with M LLRs to the right node to perform g operation. The result of g operation is M/2 LLRs as shown in Fig. 3.19 [9].

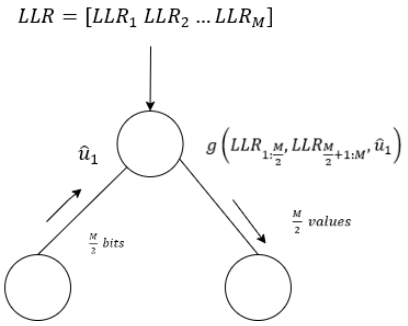


Figure 3.19 - Right child g operation

Third, after evaluating g operation and estimating right decision bit whose number are M/2 bits. The parent node evaluates the return back bits by XOR the left and right nodes and pass the right node bits as it is as shown in Fig. 3.20 [9].

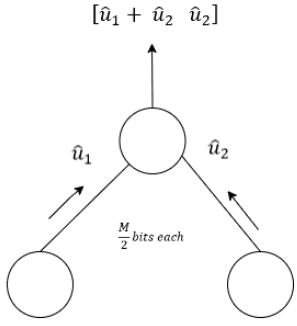


Figure 3.20 - Return back operation

3.4.2.3 Operation on a leaf node

The decision of the bit occurs at the leaf node, and it depends on the sign of the LLR entering the leaf node [4].

$$u_i = \begin{cases} 1 \;\; if \; LLR_i < 0 \; and \; u_i \in A \\ 0 \;\; if \; LLR_i > 0 \; and \; u_i \in A \\ 0 \; if \; u_i \notin A \end{cases} \tag{3.6}$$

Where A is set contain information bit. If u_i is a frozen bit always decide this bit equal zero.

3.4.2.4 Sequence of operation

1. Start at the root node.
2. Traverse tree and at each non-leaf node perform the following:
   I. Go to the left child node and perform f operation.
   II. Go to the right child node when the left decisions received and perform g operation.
   III. Perform the return back operation when the right decisions received and go to parent node.
3. If it is a leaf node, take the decision and return to the parent node.

As shown in Fig. 3.21 illustrating the sequence of operation [9].

3.4.3 Proposed Algorithm

In tree diagram representation, we have total N-LLRs at each stage(depth) starting from root node to leaf nodes. We have number of stages(depth) equals to n ,where $n = \log_2 N$. We will store all the LLRs in LLR matrix of dimensions $(n + 1, N)$ as we have $n + 1$ stages and each stage store N LLRs as shown in Fig. 3.22.

At each stage (depth) at any node we have 2^{n-d+1} input LLRs to this node at stage(depth) $d \; where \; d \; is \; from \; 0 \; to \; n$.To get the LLRs of node i at depth d equals to the following equation $LLR\left(d, 2^{n-d}\; i + 1 : 2^{n-d}\; (i + 1)\right)$ [9].

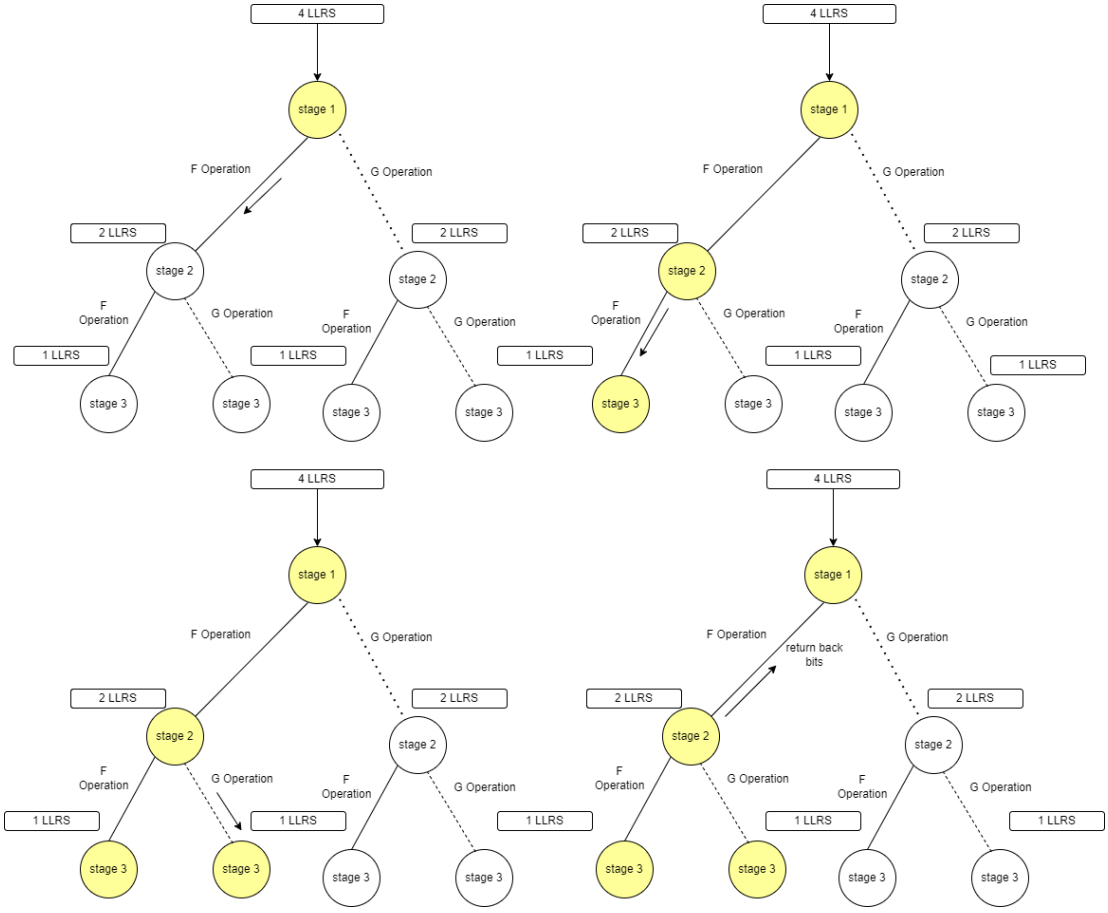


Figure 3.21 - Sequence of operation

Any interior node in the tree has one of the 4 states of operation: 0 means idle (inactive) node, 1 means done f operation, 2 means done g operation, 3 means done return back operation. The state of each node is saved inside a vector called state node vector whose dimension equals to $(1,2^{(n+1)}-1)$. The value state of node i at depth d in this vector is calculated by the formula $(2^d - 1) + i + 1$ [9].

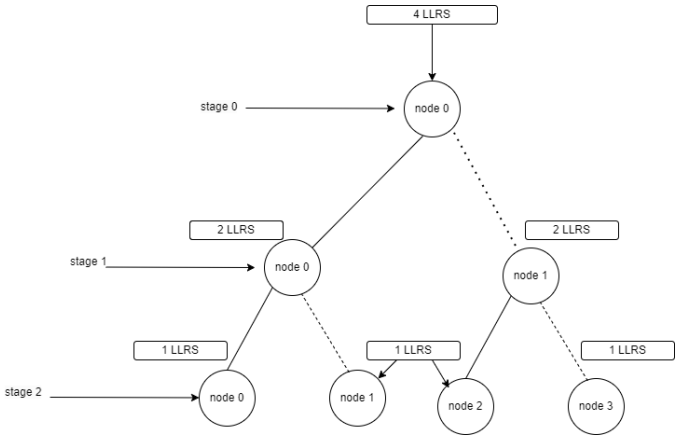


Figure 3.22 - Tree Representation

A node i at depth d to know its left child node is calculated by $2 * i$ at depth $d + 1$. It has a right child is calculated by $(2 * i) + 1$ at depth $d + 1$. The parent node is calculated by $floor(\frac{i}{2})$ at depth $d - 1$. These equations are applied only for non-leaf node or non-root node [9].

The algorithm sequence is as follow check first if the decoder reaches level n. If it reaches leaf node level, it checks if it reaches final bit whose index is N. If it reaches the final bit, it terminates the loop of operation of the decoder, if not return to parent node. If this node isn't a leaf check first the state of this node and determine the input LLRs for this node and do f or g operation if its state is 1 or 2 and go to its child. If its state is 3 do return back operation and return to parent node. More details are illustrated in the **Algorithm 4**.

Algorithm 4: SC Decoder [9]

```
1  Data K // number of Data bits + CRC bits
2  Data N // number of bits in codeword
3  Data r // received channel N LLRs
4  Data LLR // 2D buffer (n + 1, N) to store the LLRs
5  Data state nodes // 1D buffer (1,2^(n+1) − 1) to store the state of each node
6  Data estimated_U // 2D buffer (n + 1, N) to store the return back bits
7  Function SC_decoder(N,K,r)
8      LLR (1,:) = r // store the channel LLR in the first row in LLR matrix
9      Level = 0, node = 0  //start from root node
10     finish_flag = 0 // initialize finish flag = 0
11     while finish_flag = 0 then
12         node_position = (2^level − 1) + node + 1
13         if level = n then
14             Estimated_U(n + 1,node+1) = h(N, K,node,LLR(n + 1,node+1))
               //decisions of leaf nodes
15             if node = N − 1 then
16                 finish_flag = 1
17             else
18                 // move to parent node
19             end
20         else
21             Check_state_node() // check state of each node
22         end
23     end
24     received_code_word = estimated_U(n + 1,:)
25 end
```

Algorithm 5: Decisions of leaf nodes

```
Function h(N, K,node,LLR(n + 1,node+1))
    for i to size(frozen_bits) then
        if node + 1 = frozen_bits(1,i) then
            flag = 1
        end
    end
    if flag = 1 then
        result = 0
    else
        if LLR > 0 then
            result = 0
        else
            result = 1
        end
    end
end
```

Algorithm 6: Check state node

```
Function Check_state_node()
    if state_node(node_position) = 0 then
        L = LLR(level, 2^(n−level) node + 1: 2^(n−level) (node + 1))
        a = L(1: 2^(n−level)/2), b = L(2^(n−level)/2 + 1: end)
        LLR(level, 2^(n−level) node + 1: 2^(n−level) (node + 1)) = f(a, b) // go to left child then perform f operation
        state_node(node_position) = 1 // move node to next state
    else if state_node(node_position) = 1 then
        L = LLR(level, 2^(n−level) node + 1: 2^(n−level) (node + 1))
        a = L(1: 2^(n−level)/2), b = L(2^(n−level)/2 + 1: end)
        U = estimated_U(level + 1, 2^(n−level) node + 1: 2^(n−level) (node + 1)) // %go to left child then get the decision bits
        LLR(level, 2^(n−level) node + 1: 2^(n−level) (node + 1)) // go to right child and perform g operation
        state_node(node_position) = 2 // move node to next state
    else if state_node(node_position) = 2 then
        U_left = estimated_U (level + 1, 2^(n−level) node + 1: 2^(n−level) (node + 1)) // incoming decisions from left child
        U_right = estimated_U (level + 1, 2^(n−level) node + 1: 2^(n−level) (node + 1)) // incoming decisions from right child
        estimated_U(level + 1, 2^(n−level) node + 1: 2^(n−level) (node + 1))= [bitxor(U_left,U_right),U_right] // return back
        // go to parent node
    end
end
```

3.4.1 SC Decoder Limitations

The SC Decoder have major limitations as its complexity is $O(NlogN)$ and it is challenging to achieve a high throughput and low latency due to limited parallelism as we only one active node at a time. The error correcting performance of SC isn't competitive at moderate block length and can be bad as the decision node can't be revisited [4].

3.5 Successive Cancellation List (SCL) Decoder

SCL decoder is like SC decoder but with more error correcting capability. Its idea is to proceed in L parallel paths instead of one as in SC and calculate a path matric (PM) for each path as we are proceeding in them. The PM is used to select one of the paths at the end [4].

This means that we have L decoded codewords at the end of the decoding operation. The selected codeword is the one that has the minimum path metric value. Hence, the path metric is like a penalty that we pay while proceeding in the paths, so we need to select the path with minimum penalty.

But as Thomas Sowell said "There are no solutions. There are only tradeoffs". The increased error correcting capability of the SCL comes with an increase in latency and the number used hardware resources, and a decrease in throughput. The operation of the SCL decoder will be explained in the following paragraphs.

3.5.1 SCL decoder Operation

The decoder inputs are N log-likelihood ratio (LLR) values and the number of paths that we can proceed through L. To explain the SCL algorithm in a clearer way, we will use an example with a tree diagram visualization. Let's assume that we have 8 LLR values, L equals 2 and the PM is initialized with zero [4].

$$LLR = [u_0\ u_1\ u_2\ u_3\ u_4\ u_5\ u_6\ u_7] = [F_0\ F_1\ F_2\ D_0\ F_3\ D_1\ D_2\ D_3]$$

Where according to the reliability sequence F_i is the frozen bit index and D_i is the data bit index.

The first step is to check whether the index of the LLR value is that of a frozen bit or not. If it is frozen, estimate the bit as zero as illustrated in Fig. 3.23 and check the following condition to update the PM value:

- **if** LLR $>$ 0 t**hen** // LLR is a positive number
  DM $=$ 0 // DM represents the penalty that we pay if the estimated bit is wrong
  // when DM = 0 then this means that the estimation is right
- **else**
  DM $=$ |LLR| // when DM $\neq$ 0 then this means that the estimation is wrong
- $\mathbf{PM_{new} = PM_{old} + DM}$

If the index of the LLR value indicates data index then the main path is split into 2 paths, one estimates the data bit as 0 and the other estimates it as 1, Fig. 3.23. The PM is calculated for the 2 paths, and we proceed in both paths in parallel since L equals 2 in this example, even if the PM of one of them is lower than the other.

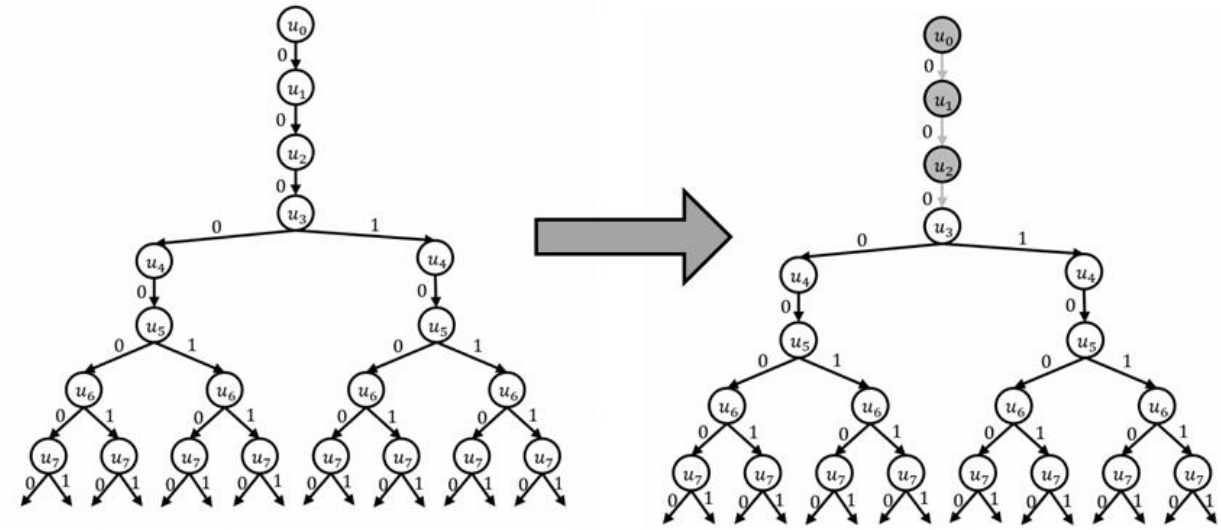


Figure 3.23 - Decoding Tree Steps

The PM is updated using the following steps:

Estimate the data bit as 0	Estimate the data bit as 1
− **if** LLR $>$ 0 t**hen** DM $=$ 0 − **else** DM $=$ |LLR| − $\mathbf{PM_{new}(path1) = PM_{old} + DM}$	− **if** LLR $>$ 0 t**hen** DM $=$ |LLR| − **else** DM $=$ 0 − $\mathbf{PM_{new}(path2) = PM_{old} + DM}$

If there is an LLR value that comes at a frozen index after a data index then no splitting occurs as illustrated in Fig. 3.24 and the penalty DM of the estimation of the bit for this LLR value is added on the existing paths, so in our example, the DM is added to the 2 paths.

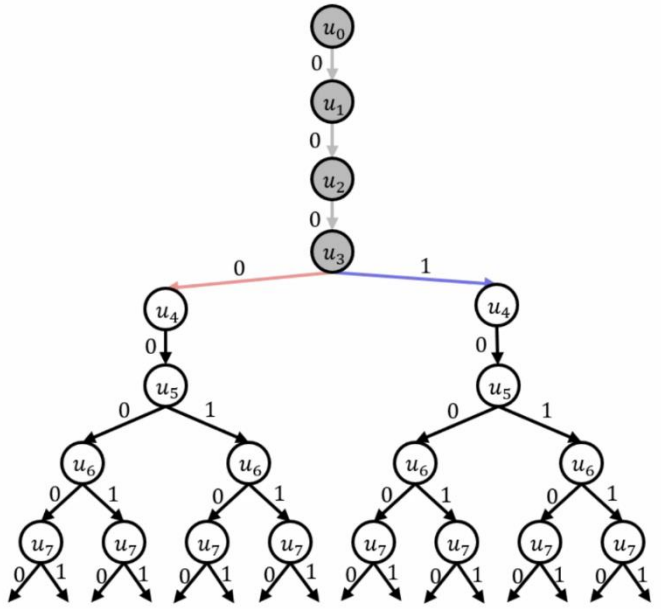


Figure 3.24 - Decoding Tree Steps

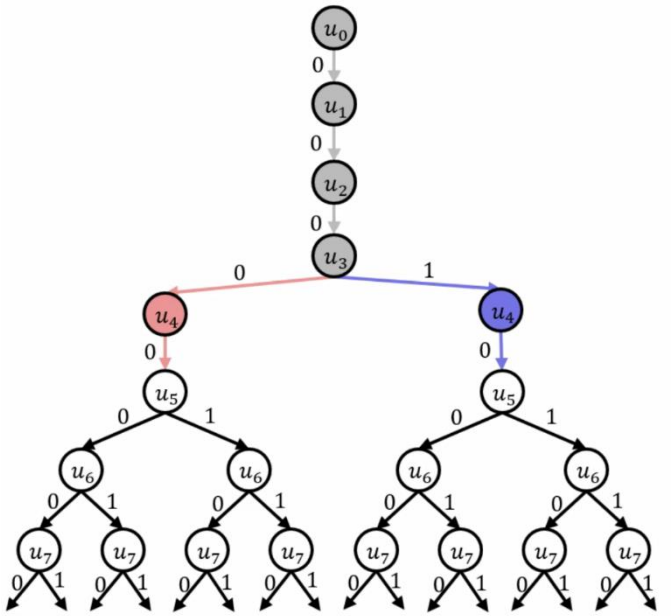


Figure 3.25 - Decoding Tree Steps

When a new LLR value at a data index comes, a new splitting occurs for the 2 paths, this means that each path is split into 2 new paths to estimate the new data. Now, the problem is that we have a total of 4 paths, and we can only proceed in 2 paths due to the limitation imposed by the hardware resources. In this case, we sort the 4 paths ascendingly according to PM then select the best 2 paths having the least PM to proceed through as illustrated in Fig. 3.25.

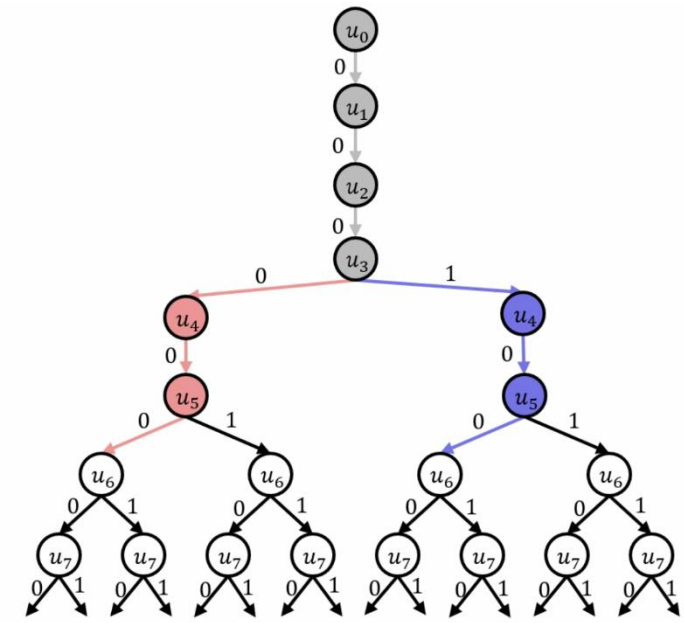


Figure 3.26 - Decoding Tree Steps

We repeat the last step every time we get an LLR value at a data index. Note that if the 2 chosen paths from the 4 new paths are from the same main path, we discard the old path side, and continue in the 2 new paths as illustrated in Fig. 3.26, where we can see that the right main path is discarded, and the left main path is split into 2 paths that becomes the new main paths. We proceed in the same way as illustrated in the previous steps, Fig. 3.27.

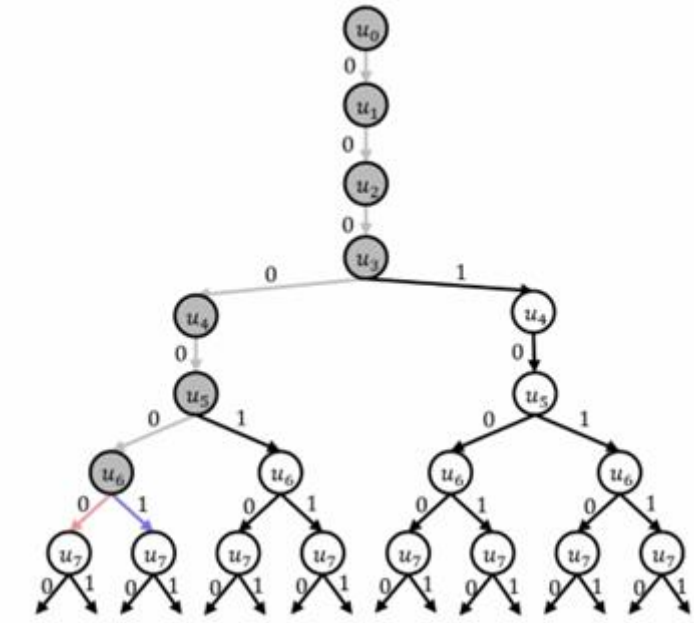


Figure 3.27 - Decoding Tree Steps

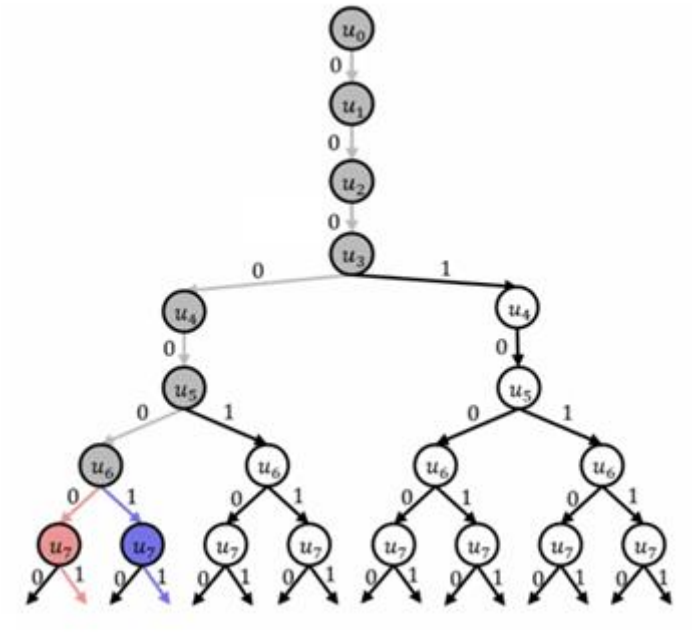


Figure 3.28 - Decoding Tree Steps

The decoder path updating algorithm is summarized in **Algorithm 7**.

Algorithm 7: update paths function algorithm

```
1   Function update_paths()
2     if bit_index is frozen_bit_index then // frozen bit
3         no_splitting and estimated_bit = 0
4         calculate DM and PM_new
5     else  // data bit
6         if number_of_paths != L then
7             split each path into 2 paths, one for estimated_bit = 0 and the other
              for estimated_bit = 1
8             calculate DM and PM_new for each path
9         else
10            split each path into 2 paths, one for estimated_bit = 0 and the other
              for estimated_bit = 1
11            sort the 2L paths ascendingly according to PM
12            choose L paths from them // as max number of paths is L
13            calculate DM and PM_new for each path
          end
      end
      end
```

The full SCL decoder algorithm is shown in algorithms 8 to 11.

Algorithm 8: SCL decoding algorithm [10]

```
Data: L // number of paths
Data: LLR // 2D buffer (L, 2N-1) to store the LLRs.
Data: BITS // 2D buffer (L, N) to store the bits.
1   Function SCL_decoder(N, o_LLR, o_BITS)
```

```
2   N_1/2 = N/2
3   if N > 1 then // not a leaf node
4     for j = 1 to L do // loop over the paths
5       for i = 1 to N_1/2 do // apply the f (Left) function
6         LLR (j, o_LLR+N+i) = Left(LLR(j, o_LLR+i),LLR(j, o_LLR+N_1/2+i))
        end
      end
7     SCL_decoder(N_1/2, o_LLR+N, o_BITS)
8     for j = 1 to L do
9       for i = 1 to N_1/2 do // apply the g (Right) function
10        LLR (j, o_LLR+N+i) = Right (LLR(j, o_LLR+i),LLR(j,
          o_LLR+N_1/2+i),BITS(j, o_BITS+i))
        end
      end
11    SCL_decoder(N_1/2, o_LLR+N, o_BITS+N_1/2)
12    for j = 1 to L do
13      for i = 1 to N_1/2 do // update the partial sums or return to the parent node
14        BITS(j, o_BITS+i: N_1/2:o_BITS+N_1/2+i) = Return_Back(BITS(j,
          o_BITS+i),BITS(j, o_BITS+N_1/2+i))
        end
      end
15  else // a leaf node
16    update_paths() // update, create and delete paths
    end
  end
17 SCL_decoder(N,0,0) //launch the decoder
18 select_best_path()
```

Algorithm 9: Left function algorithm

```
1   Function α = Left(a,b)
2   if a*b > 0 then
3     sign = 1
4   else
5     sign = -1
    end
6   α = sign * min (|a|, |b|)
    end
```

Algorithm 10: Right function algorithm

```
1   Function β = Right(a,b,u)
2     β = (1 - (2*u)) * a + b
    end
```

Algorithm11: Return_Back function algorithm

```
1 Function return_bits
=Return_Back(ua,ub)
2     return_bits = [ua ⊕ ub , ub]
  end
```

3.6 Results

In this section, we will display the simulation results of SC and SCL decoders implementations using MATLAB. We will present and discuss the following: Polar decoder models verification, 3GPP specifications and selecting the appropriate list size (L).

3.6.1 Polar Decoder Model Verification

After implementing the SC and SCL algorithms of the Polar decoder, we verified their operation by comparing the obtained Frame Error Rate (FER) curves with the curves found in the paper [11].

The Polar decoder specifications for the curves in the paper are N=1024, R=1/2, K=512, with no rate matching and CRC was not used, where R is the rate, and it is the ratio between the input and the output of the decoder.

After applying these specifications into our decoder implementations, we obtained the curve in Fig. 3.29, where the solid lines are the paper (reference) curves, and the dashed lines are our simulation results, and as shown, our results are very close to the paper curves which means that our implementation is working correctly.

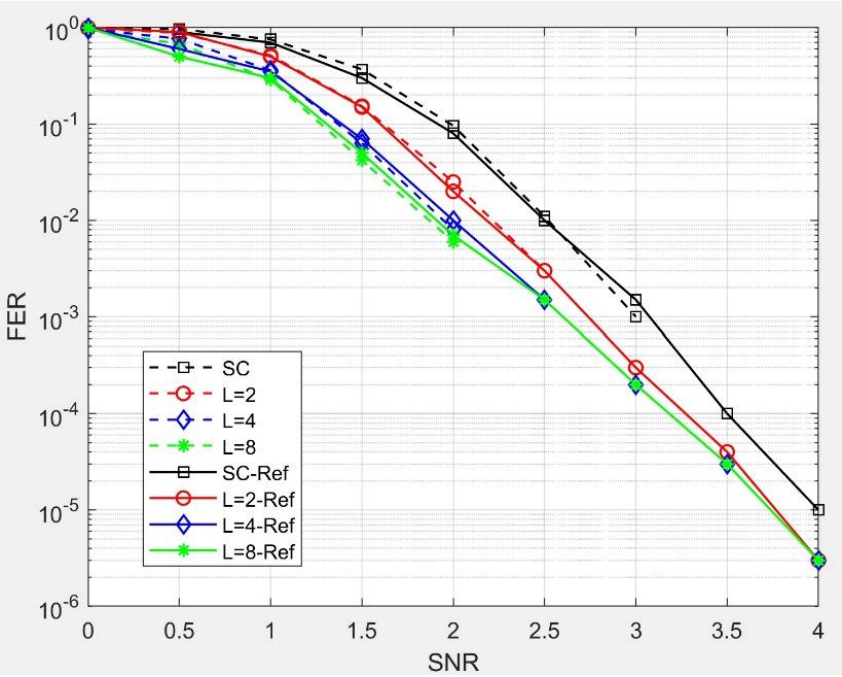


Figure 3.29 - SC and SCL decoders comparison with paper

3.6.2 3GPP Encoding/Decoding Specs

Before turning this model into hardware, we need to determine the encoding and decoding specs set by the 3GPP so that the hardware designer designs the decoder according to these specifications since any difference in the specifications gives a different performance and results.

We followed the 3GPP standard specifications [8] for using the polar decoder in the Physical Broadcast Channel (PBCH) and these specifications are listed in table (3.1).

Table 3.1 - 3GPP Specs for PBCH

N	512
A	32
E	864
Rate Matching Type	Repetition
CRC-24	$x^{24} + x^{23} + x^{21} + x^{20} + x^{17} + x^{15} + x^{13} + x^{12} + x^{8} + x^{4} + x^{2} + x + 1$

Where N is the code word length, A is the number of data bits, E is the final codeword length after using rate matching. There are several types of rate matching and in PBCH, the 3GPP standard has specified its type as repetition which means that we take the first (E-N) bits in the N code word and concatenate them at the end.

CRC (Cyclic Redundancy Check) adds 24 bits at the end of the data to help in detecting the errors. The input of the Polar encoder is (A + CRC) and its output is N, and the input of the rate matching is N, and its output is E, and all of that is illustrated in Fig. 3.30.

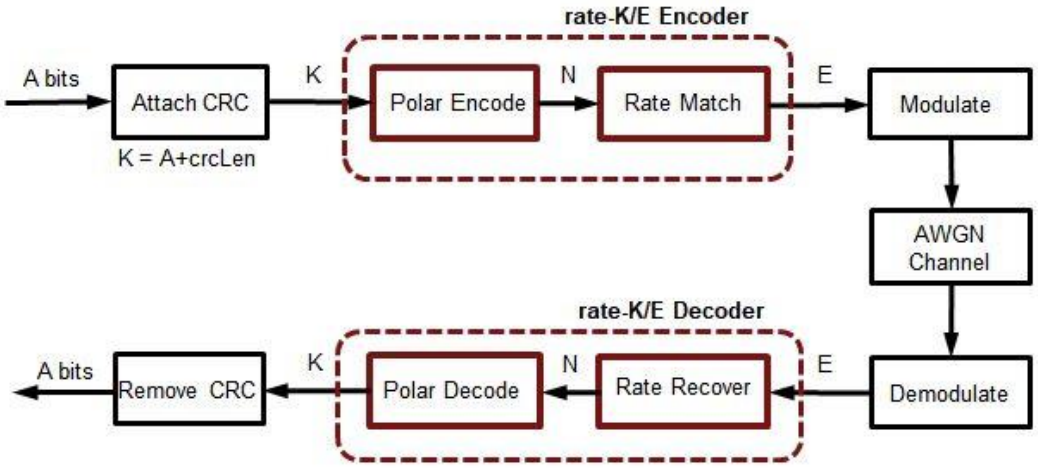


Figure 3.30 - TX-RX chain starting from channel encoder and ending at channel decoder.

3.6.3 Selecting the appropriate list length (L)

L represents the number of hardware resources used to implement the SCL decoder. Increasing the L means increasing the number of used resources, while increasing the error correcting capability. From hardware point of view using a very large L is not optimum, but using a very low L decreases the error correcting capability while optimizing hardware.

We simulated our implementation with different L values as shown in Fig. 3.32, and as L increases, the FER (frame error rate) curve decreases which means less errors happened. To choose which L to continue the design with, we compared these curves

with the reference point of 1 RX provided by the 3GPP. Below this point means that we met the specification and above it means that we didn't meet the specification.

This point tells us about the maximum accepted error at specific SNR. In the 3GPP standard, we found the reference point of 2 RX but not the 1 RX. Usually, 1 RX reference point has the same maximum accepted error as in 2 RX but at higher SNR (SNR 2 RX + 2 dB).

After adding the reference point of 1 RX to the graph, Fig. 3.31, we found that all the curves meet the specification as they are below the reference point at the specific SNR. We found that L = 2 or higher is accepted. Even though the difference between L = 4 and L = 2 is too small, we selected L = 4 to add more safety margin since the simulations are using AWGN channel which has lower noise effect compared to fading channel.

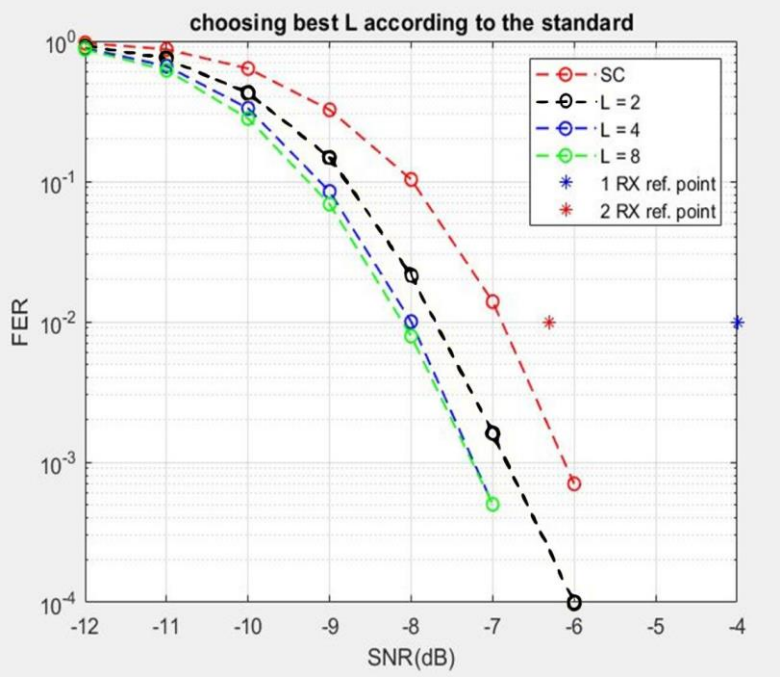


Figure 3.31 - Selecting the appropriate L.

We conclude that our implementation for the SC and SCL decoder is verified, and it works correctly. Then the decoder specs are determined from the 3GPP standard specs for PBCH, and we completed with SCL decoder as its better than SC in error correcting capability. Finally, the appropriate L for SCL decoder is chosen by taking into consideration the simulation channel type which is AWGN [12].

3.7 Fixed-Point Analysis

In this section, we will discuss the last step in the specification selection process. Before converting an algorithm into hardware, we need to determine the width of the signals, and answer the question of how many bits we need to represent the signals of our algorithm in binary with acceptable quantization error.

We simulate our signals on MATLAB which represents them in double. To represent the signals in fixed point, Fig. 3.32, we need to lose some precision which leads to losing some accuracy, causing quantization error. First, we need to determine the following:

- The signal is signed or unsigned (always positive). If signed, we assign 1 bit for the sign value, otherwise we discard the sign bit.
- The number of bits the integer and fraction parts take. If the signal's dynamic range is small and its values lie between -1 and 1 then no integer part is needed, and most of the bits are assigned to the fraction part and vice versa, if the dynamic range is high, the integer part is assigned most of the bits.

Figure 3.32 - Fixed-Point Representation

Since the channel noise is random, this makes the received signal dynamic range random. To solve this problem, we simulated the behavior of the AGC (automatic gain controller) block at the starting of the receiver chain to normalize the received signal (multiply the received signal by gain to make its values lie between -1 and 1).

The signal normalization removes the need for the integer part, so the received signal has sign bit and fraction part only. But the internal signals have an integer part due to the addition operation done on them which makes their values become bigger. We can conclude that the decoder's input has sign bit and fraction part but no integer part, the internal signals have sign bit and fraction and integer parts, and there is no need to quantize the output.

Now, we need to understand how to determine the number of bits for the integer (I) and fraction (Q) parts. For the integer part, we monitored the internal signals to obtain the maximum of integer part that they reach. After determining this number, we represent it in number of bits equals to $\lceil \log_2 \text{integer}_{\max} \rceil$.

For the fraction part, we sweep Q through the range 4 to 8 and compare the resulting FER curves with the curve that resulted from the signals when its double (the floating curve), and the result is shown in Fig. 3.33.

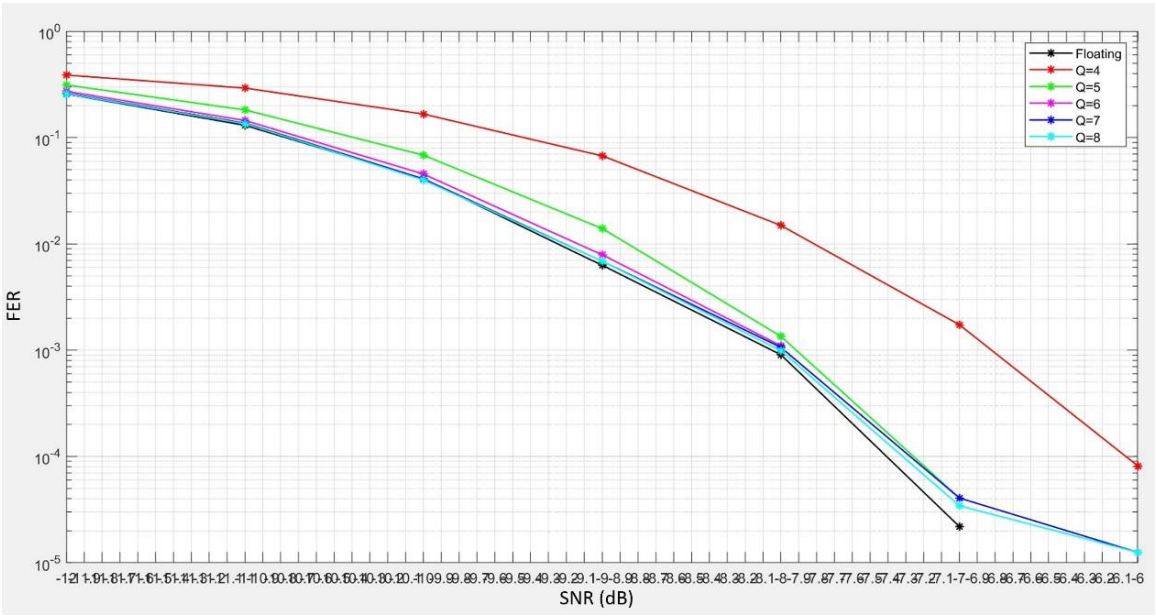


Figure 3.33: Comparing the resulting curves from quantization with the floating one. The X-axis range is -12: -6 with step 0.1 dB.

In Fig. 3.33, we can notice that as Q increases, the resulting curve becomes closer to the floating curve due to the increase of accuracy (less quantization noise). But this leads to more bits which means using more hardware resources (not optimized for hardware).

We can determine the suitable value for Q from the maximum accepted error. The maximum accepted error is 0.1 dB SNR which means that at the same FER value, the difference between the corresponding SNR in the floating curve and the quantized curve is 0.1 dB. After applying this spec on Fig. 3.33, we found that the minimum value of Q that meets the requirement is 6 and this is clearer in Fig. 3.34.

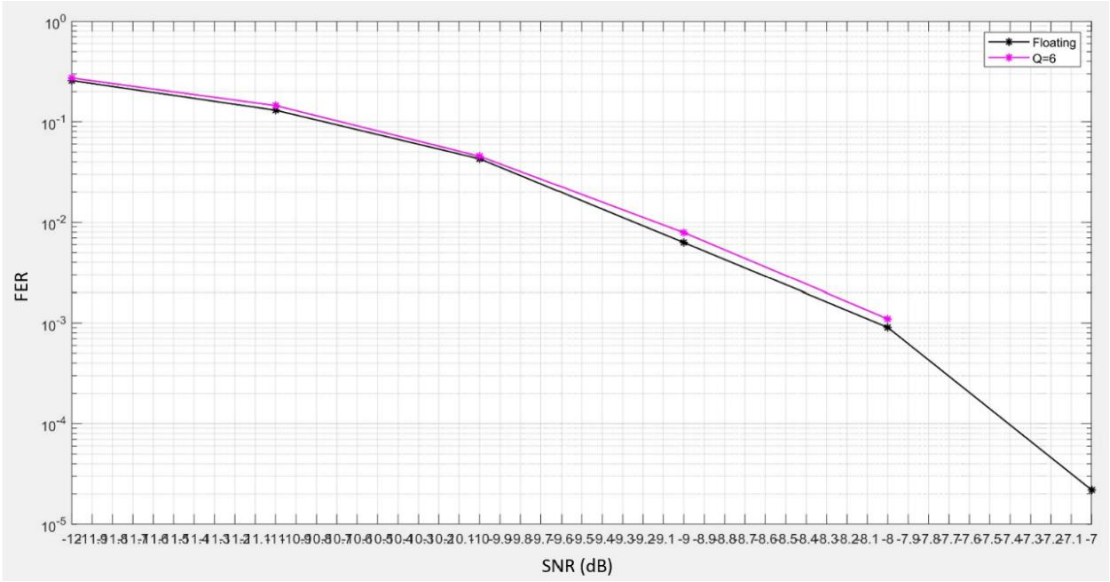


Figure 3.34 - The quantized curve at Q = 6 with the floating curve to show that the difference between the 2 curves is 0.1 dB approximately. The X-axis range is -12:-7 with step 0.1 dB.

IN_LLR	Sign(1 bit)	Int(0 bits)	.	Fraction (6 bits)
INT_LLR	Sign(1 bit)	Int(7 bits)	.	Fraction (6 bits)
PM	Sign(0 bit)	Int(7 bits)	.	Fraction (6 bits)

Figure 3.35 - Summary of the results of fixed-point analysis first trial

We can summarize the results of fixed-point analysis as shown in Fig. 3.35, where IN_LLR is the input signal and INT_LLR and PM are the internal signals. As we can see, the internal signals width is very high which leads to a large internal memory size. All of that happened due to the high dynamic range of these signals, so to achieve lower width, we need to decrease it.

From the decoder specs N equals 512, this means that we have 9 levels in the tree diagram. By doing statistical analysis on the variation of the dynamic range of the INT_LLR signal at each level, we found that at level 4 the dynamic range is high compared to the remaining levels. To control the variation in the dynamic range, we need to decrease the integer part of this signal while maintaining an acceptable quantization error. So, we decided that 1 bit for integer part is enough for the INT_LLR.

To fit the INT_LLR signal into 1 bit integer, we divided the integer part of the whole internal signal at the 4^{th} level by 2 when the maximum and minimum values exceed the range that can be tolerated by 1 integer bit. At the other levels, when the signal exceeds the upper and lower limits of 1 bit, we just saturate or truncate the signal.

Saturation is done on all the levels, but why we don't saturate the other levels like the 4^{th} level after dividing by 2? Because quantization error caused from saturation of the other levels is small compared to that in 4^{th} level, so first we need to decrease the signal values and then saturate them to decrease this error.

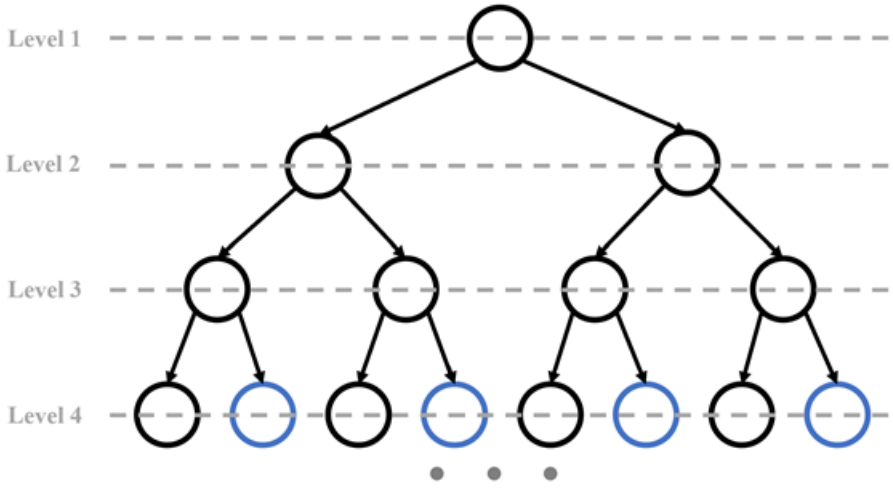


Figure 3.36: The first 4 levels of the total 9 levels of the tree diagram. The division by 2 happens at the 4th level, mainly at the right node.

As shown in Fig. 3.36, the division of the 4^{th} level mainly happens at the right node. This is due to the addition operation that happens at the right node that causes the signal value to increase while in the left one, only the sign changes.

But to decrease the dynamic range of PM, we subtract the signal by 1. This subtraction happens at any level as soon as the signal maximum integer value exceeds the maximum threshold. After the subtraction, we also saturate to be sure that all the signal values fit in the range of the 1 bit which is specified for the integer part.

We treat the PM signal differently because its nature is different from INT_LLR signal. As soon as any path reaches the maximum limit of the 1-bit integer, any truncation after that doesn't create problems as this means that this path is from the worst ones then the probability of choosing it is very low. The subtraction leads to the appearance of negative values in PM; hence the PM becomes a signed signal and not unsigned as before since it consists of adding the magnitudes of the LLR signals.

To summarize what we have reached till now, the integer part of the INT_LLR and PM signals becomes 1 bit, and the PM becomes signed. But the fraction part value increases from the previously obtained value. After sweeping the value of Q, we got the results shown in Fig. 3.37, which tells us the same results as in Fig. 3.37. The difference here is that the curve that contains the minimum Q which meets the requirement of maximum accepted error is at Q = 7 not 6 as before and this becomes clearer in Fig. 3.38.

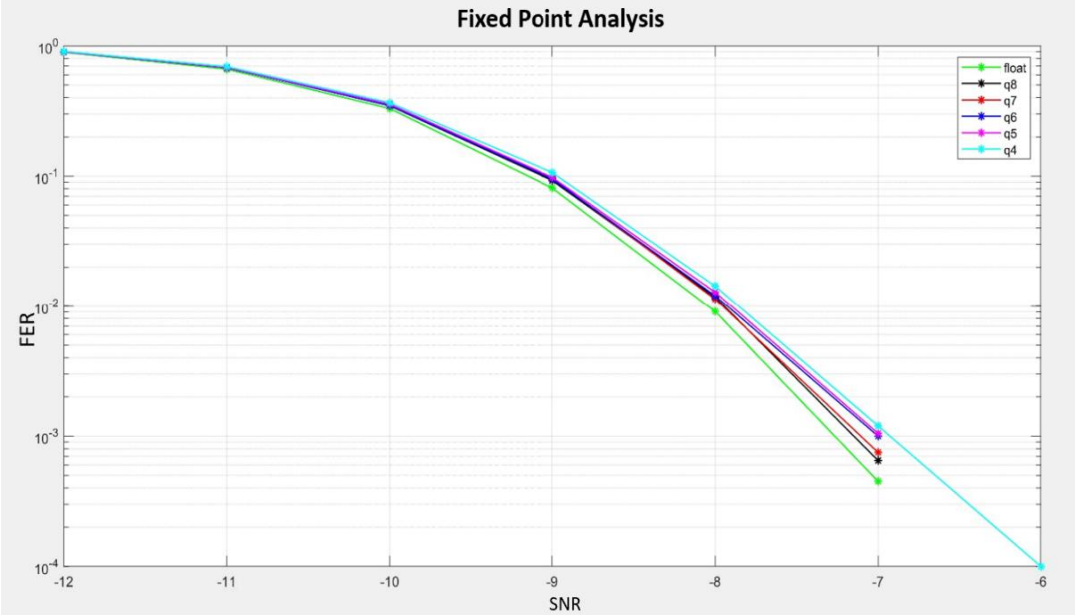


Figure 3.37 - Comparing the resulting curves from quantization with the floating one after decreasing the dynamic range of the internal signals.

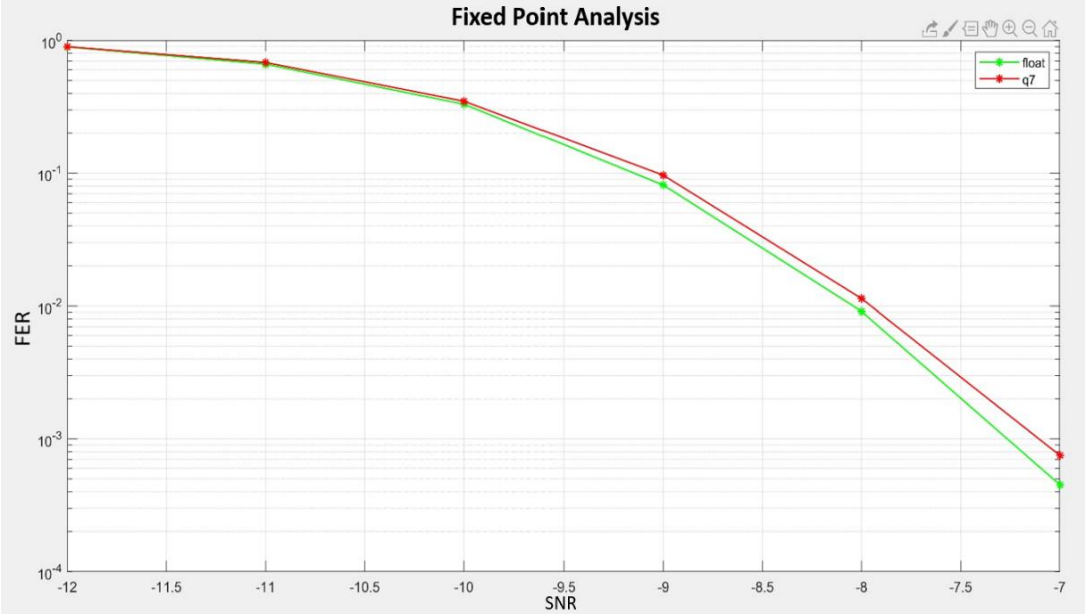


Figure 3.38 - The quantized curve at Q = 7 with the floating curve to show that the difference between the 2 curves is 0.1 dB approximately

We can conclude that decreasing the dynamic range of the signals decreases the integer part and increases the fraction part but not with the same ratio. The 7-bit integer part is substituted with 1 bit integer part and only one additional bit is added to the fraction part, Fig. 3.39. This shows the power of decreasing the dynamic range of a signal and its effect on reducing the signals width representation which in turn reduces the used hardware especially of the internal memories size.

IN_LLR	Sign(1 bit)	Int(0 bits)	.	Fraction (7 bits)
INT_LLR	Sign(1 bit)	Int(1 bits)	.	Fraction (7 bits)
PM	Sign(1 bit)	Int(1 bits)	.	Fraction (7 bits)

Figure 3.39 - Summary of the results of the fixed-point analysis after decreasing the dynamic range of the internal signals.

The algorithms of reducing the dynamic range of the internal signals INT_LLR and PM are mentioned in **Algorithm 12 and 13** respectively. **Algorithm 12** is inserted after the loop of right function in SCL decoding algorithm (Algorithm 7 in SCL decoder algorithm section), and **Algorithm 13** is inserted at the end of the update path's function algorithm (Algorithm 6 in SCL decoder algorithm, section).

Algorithm 12: Decreasing dynamic range of INT_LLR algorithm

```
1  if we are in 4th level then
2  |  MAX_condition = integer (max (LLR (j, :)) > MAX_value   // MAX_value is
   |  the upper bound of 1 bit as
   |                                                          // integer part.
3  |  MIN_condition = integer (min (LLR (j, :)) < MIN_value   // MIN_value is the
   |  lower bound of 1 bit as
   |                                                          // integer part.
```

```
4    if MAX_condition or MIN_condition then
5        LLR (j, :) = LLR (j, :) / 2
6        LLR (j, :) = Quantize (LLR (j, :), signed, 1 bit integer, 7 bits fraction)
7    else
8        LLR (j, :) = Quantize (LLR (j, :), signed, 1 bit integer, 7 bits fraction)
     end
9  else
10   LLR (j, :) = Quantize (LLR (j, :), signed, 1 bit integer, 7 bits fraction)
   end
```

Algorithm 13: Decreasing dynamic range of PM algorithm

```
1   Condition = integer (max (PM) > MAX_value  // MAX_value is the upper bound
of 1 bit as integer part.
2   if Condition then
3    PM = PM - 1
4    PM = Quantize (PM, signed, 1 bit integer, 7 bits fraction)
   end
```

3.8 Conclusion

Now, we have completed the fixed-point analysis and our algorithm is ready with its determined specifications for hardware implementation.

Chapter 4: Hardware Literature Survey.

4.1 Introduction.

In the past few years, many researches proposed many papers on their work in the field of polar codes. Our scope in this thesis is to implement the hardware architecture of the successive cancellation list decoder for polar codes.

We are going to focus on different architectures for the successive cancellation decoder and analyze their merits and demerits to decide which one is suitable for implementing the list decoder.

4.2 Pipelined Tree Architecture.

4.2.1 Methodology.

The architecture divides the decoding operation into number of stages which

can be calculated from the code length (N) using the formula

$$l = \log_2 N \tag{4.1}$$

There are 2^l operations in each stage divided into 2 types, the first one is the F function which takes 2 inputs (LLRs) and calculates output LLR. Whereas the g function takes 3 inputs (2 LLRs and a partial sum) and calculates output LLR.

4.2.2 Processing Element Architecture.

The pipelined architecture PE has 2 function blocks Figure 4.1, each calculates one type of operation only either F or G function which corresponds left and right nodes in the decoding algorithm [13], Where the F function is calculated through the following equation

$$F(L_1, L_2) = sign\ (L_1) * sign(L_2) * \min(|\ L_1|, |L_2|) \tag{4.2}$$

While the G function is calculated through the following equation

$$G(L_1, L_2) = L_1 * (-1)^u + L_2 \tag{4.3}$$

The number of processing elements (P) used in each stage is dedicated for that stage only and cannot be used in any other stage.

$$P = 2^l \tag{4.4}$$

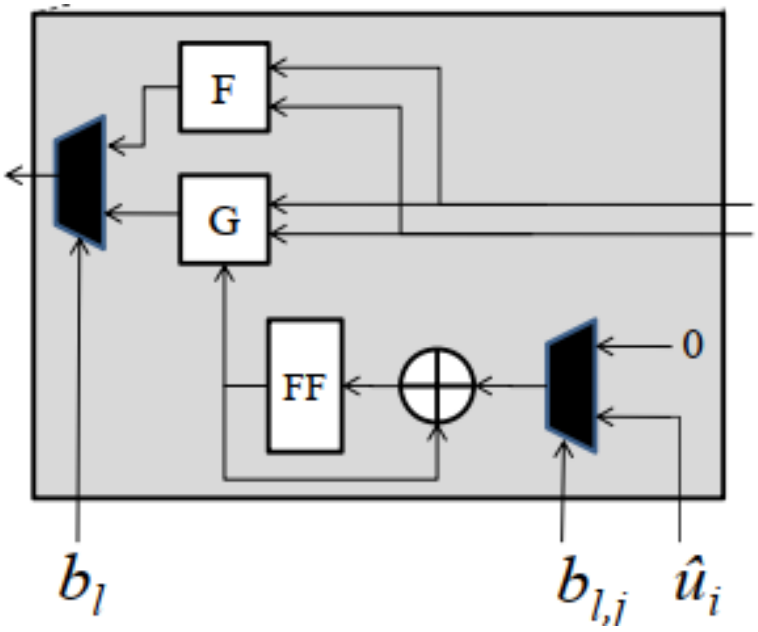


Figure 4.1- PE Pipelined architecture

4.2.3 Advantages.

Compared to other SC architectures [14], Pipelined architecture has higher throughput and higher frequency due to its pipelined nature that results in breaking the critical path of the data path.

Easier HW implementation as there are dedicated PEs for each stage therefor there is no need for a complicated multiplexing network so as to avoid high routing congestion.

4.2.4 Disadvantages.

From the disadvantages of the pipelined architecture is its large area compared to other architecture as there are number of PEs dedicated for each stage so there is no resource sharing, also there are 64 PEs (for N=512) which are used only twice throughout the whole decoding operation in the highest decoding tree node $l = \log_2(N) - 1$.

Another area inefficiency source is having 2 function blocks in each PE however, we only need either of them in each clock cycle and the other one will be idle until the next operation as shown in Table 4.1 in addition to the used registers due to the pipelined nature of each decoder as shown in Figure 4.2.

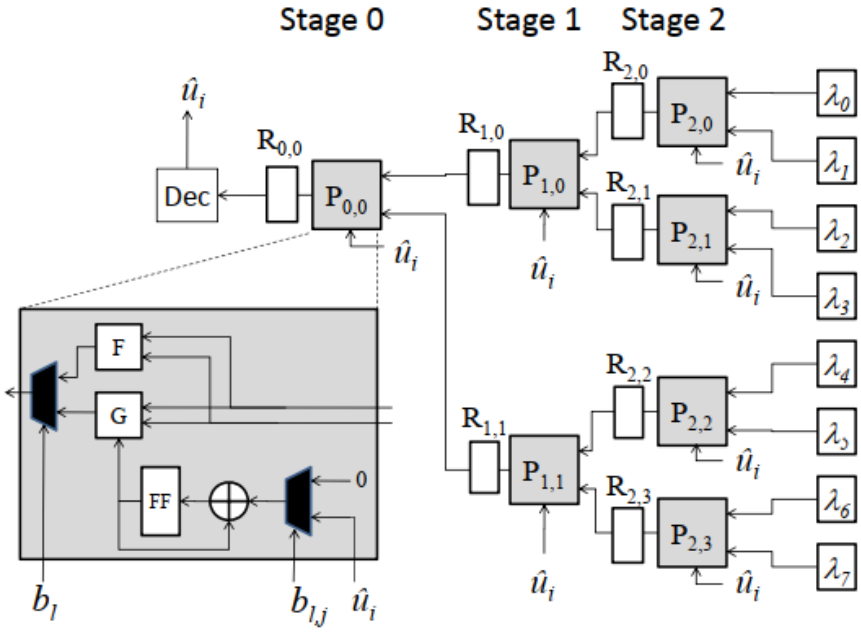


Figure 4.2-Pipelined architecture data path for N=8

Table 4.1-Pipelined architecture scheduling

CC	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S1	F							G						
S2		F			G				F			G		
S3			F	G		F	G			F	G		F	G
Ui			U0	U1		U2	U3			U4	U5		U6	U7

4.3 Line Architecture.

4.3.1 Methodology.

In order to reduce the number of PEs used in the pipelined architecture, the line architecture was introduced while maintaining the same throughput using N/2 PEs only by merging some of the PEs used in the pipelined architecture which makes this architecture simpler than the other [13] despite the extra multiplexing logic required to route the data throughout the line as shown in Figure 4.3.

The name “Line” comes from the fact that the PEs are arranged in a line while the used registers retain a tree structure emulated by a multiplexing network connecting them.

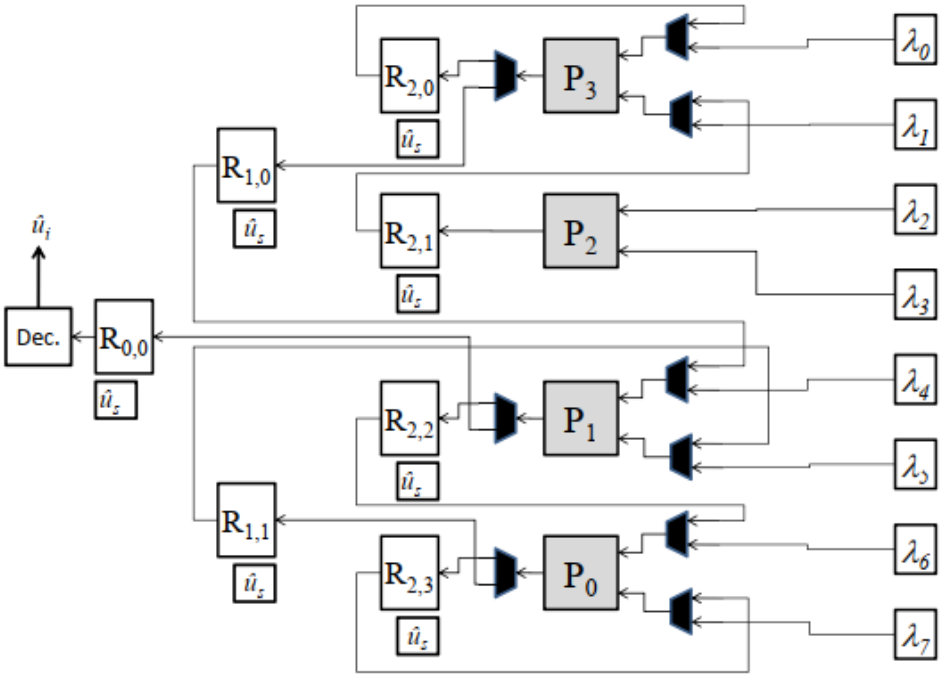


Figure 4.3- Line Architecture for N=8

4.3.2 Processing Element Architecture.

Unlike the pipelined architecture PE Figure 4.1, the line architecture PE has only one function block to perform both the F and G functions using the same equations described in the pipelined architecture (4.2),(4.3) allowing resource sharing which introduce area optimization.

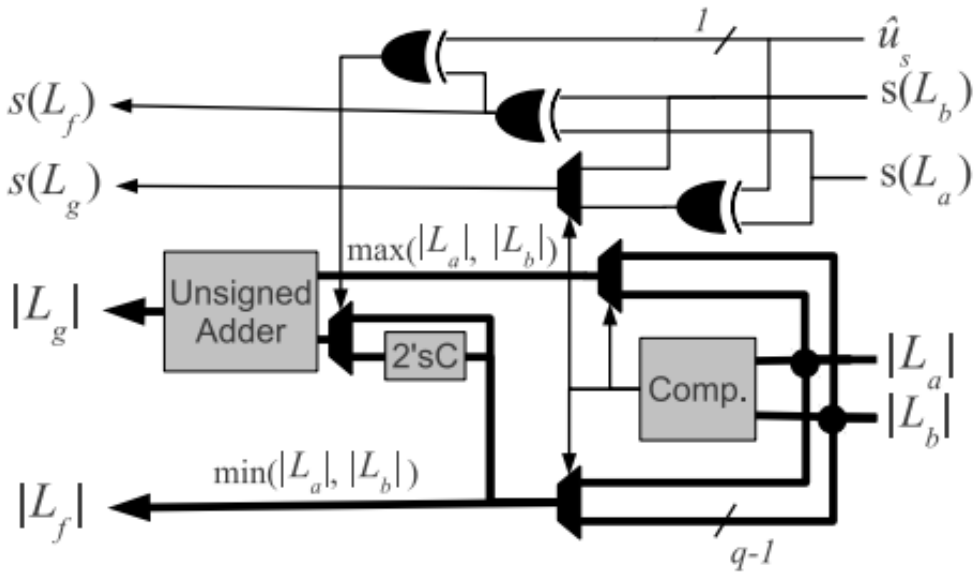


Figure 4.4 - Line architecture PE

4.3.3 Advantages.

Compared to the pipelined architecture, the line architecture has more area efficiency in terms of the used PEs due to reducing the number of processing elements while maintaining the same throughput in addition to using only one function block inside each PE.

The PE architecture modifications over the pipelined architecture PE in terms of their number and internal structure (function blocks) makes the line architecture simpler.

4.3.4 Disadvantages.

The storage of the internal LLRs that are needed throughout the decoding process are implemented using registers and multiplexing and de-multiplexing networks which add more area. It was reported that for large code length (N) required polar codes synthesis results cannot be implemented on most FPGAs [14].

Compared to the semi-parallel architecture, the line architecture uses a huge number of PEs which can be reduced to N/4 instead of using N/2 while maintaining nearly the same latency (2 cycles more) [14].

4.4 Semi-parallel Architecture.

4.4.1 Methodology.

The semi-parallel architecture introduces the usage of dual port read RAM to store the internal LLRs required through the decoding process. Moreover, the used number of PEs are $\frac{N}{4}$ which is considered a good modification over the line architecture which uses $\frac{N}{2}$ processing elements.

One of the aims of this architecture is to complete the whole operation of the PE in a single cycle, but that would require simultaneous read and write from the memory in some cases hence, they came up with a solution for this problem which is bypassing the previous output to be used in the current LLR calculation [14] as it can be seen in Figure 4.5.

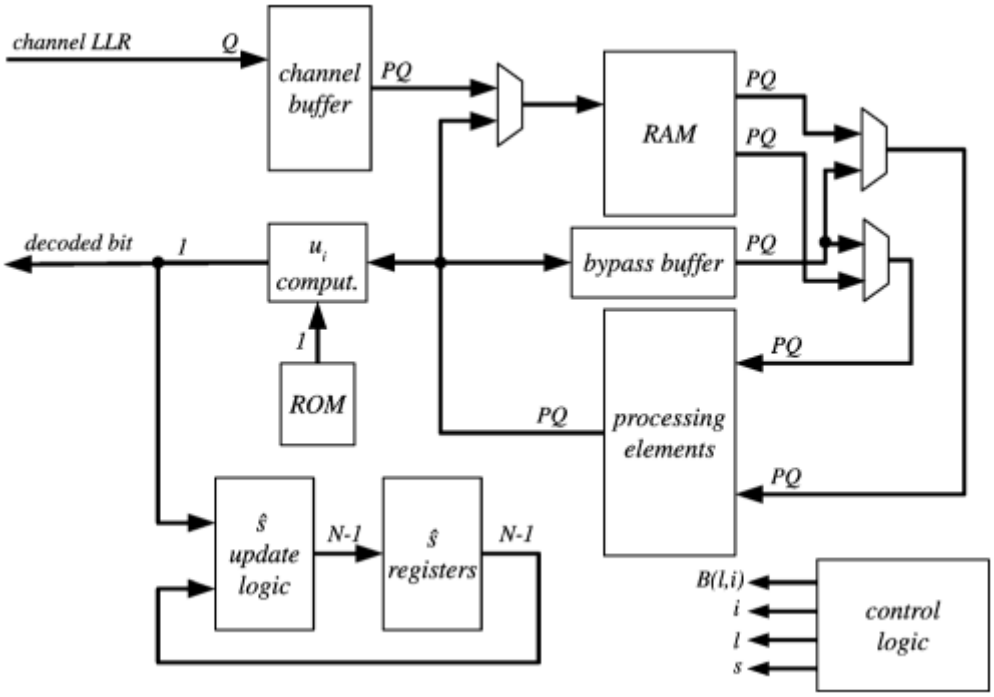


Figure 4.5 - Semi-parallel architecture

4.4.2 Processing Element Architecture.

Extra multiplexers are added as a modification over the line architecture PE to choose between the 2 functions it can perform using a function select signal. The sign and magnitude implementation of the LLRs saves 20% of the total area during the synthesis compared to the 2's complement implementation [14].

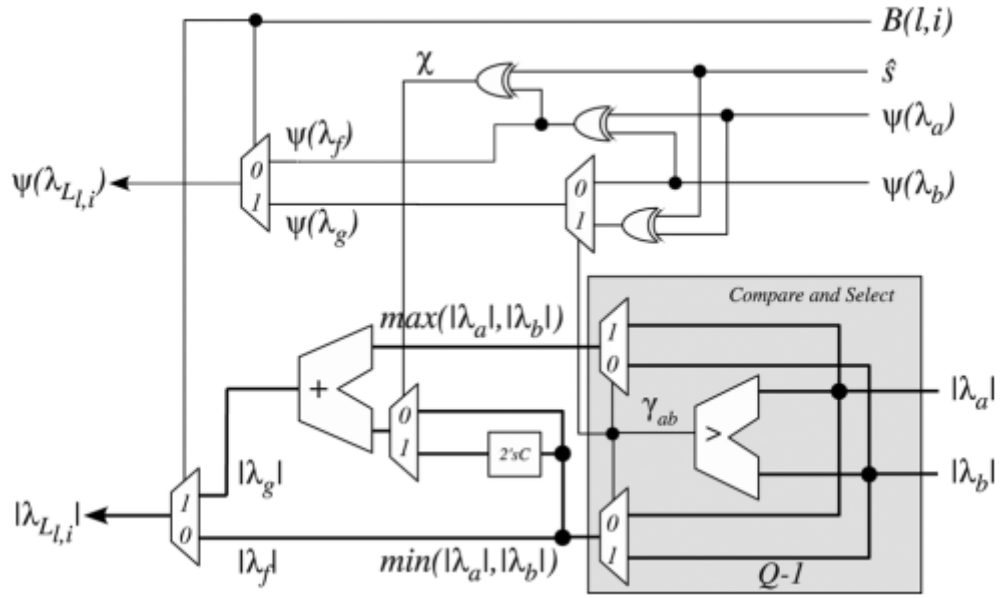


Figure 4.6 - Semi-parallel architecture PE

4.4.3 Advantages.

50% reduction in the used number of PEs over the line architecture. In addition to introducing the dual port read RAM which supports large code length (N) required by the polar codes during synthesis with only 2 clock cycles extra latency compared to the line architecture which can be ignored [14].

4.4.4 Disadvantages.

$\frac{N}{4}$ Processing elements is still a huge number to be used with the fact that we will use all of them in the higher decoding tree stage only and then most of them will be idle 50% of the decoding operation.

The used dual port read RAM introduces area inefficiency and cannot be found in most FPGAs. In addition, it introduces routing problems in the ASIC flow.

4.5 Summary.

There is no perfect design, some designs focus on optimizing the number of PEs without any considerations for the memory organization used, some design used dual port read memory which introduce area overhead in addition to bypass buffer while some others use register banks which is not feasible for large code length (N).

Synthesis results discussed in the literature, that the memory organization is the main contributor for the area. Therefore, in the next chapters we are going introduce some memory organizations which are more area efficient than the used organizations in the literature architectures without severely affecting the throughput.

Table 4.2 - Comparison between the 3 architectures for N=512

POC	Pipelined	Line	Semi-parallel
Number of PEs	N-1 = 511	N/2 = 256	N/4 = 128
Memory organization	Register banks	Register banks	Dual port read RAM
Latency	2N-4 = 1020	2N-4 = 1020	2N-2 = 1022

Chapter 5: Hardware Design Specifications.

5.1 Memory Specifications.

According to the Target FPGA (ZYNQ ULTRASCALE+), the maximum word length is 72 bits in order to infer BRAMs without using the logic fabric of the FPGA as a storage element.

As mentioned before, Dual port RAM is not preferable when targeting FPGAs hence, we are going to use single port RAM as a constraint in order to optimize the area as no other design in the literature targeted area optimization.

5.2 Clock Frequency Specifications.

Maximum clock frequency is 61.44 MHZ since our sampling frequency $f_{sample} = 256 * 15K = 3.84\ MHZ$, then the clock frequency must be a multiple of f_{sample}. Hence, it was decided to be 16 multiples of the sampling frequency so that the FFT block has an extra cycle between every 2 samples to process then our working clock frequency becomes $F = 16 * 3.84\ MHZ = 61.44 MHZ$.

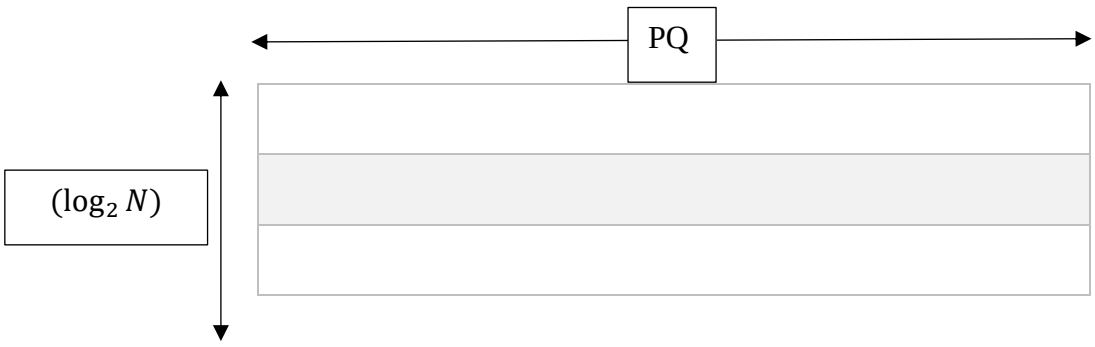
According to NR 5G, there 14 symbols in one slot, since in our case the sub carrier spacing is 15 KHZ therefore slot time is 1ms, then with clock frequency 61.44 MHZ we have 61440 clock cycles in 1ms. The decoding process is allowed to consume 50% of the total latency of the chain.

Chapter 6: Hardware Design Iterations.

6.1 First design iteration.

During the first design iteration, we saw that the outputs from the LLR memory have to be delivered to all the PEs at the same time, So the memory word size should be PQ so that all the PEs take their inputs at the same time, where P is the number of processing elements needed while Q is the total size of LLR as a number of bits. Using a single port RAM introduced a delay of 4N clock cycles.

As for the number of locations, it will be $(\log_2 N) - 1$ each of size PQ.



For N = 512, P = N/4 = 128, Q = 6 memory size will be 8 code words each of size 128Q.

Stages' LLR output were arranged as follows, Where the input to each stage is the output of its previous stage.

0	Channel LLR
1	Channel LLR
2	Channel LLR
3	Channel LLR
4	L8
5	L8
6	L7
7	L6:L0

As the input is serial of size Q, A buffer was needed to accumulate the channel LLR to form a word and then write it into the memory.
Indexing of the memory depend on the stage number during reading and writing.

Table 6.1 - Iteration 1 memory indexing

Stage (Layer) Number	Size	Number of Words	Word no.
Channel LLR	512 Q	4	0,1,2,3
L8	256 Q	2	4,5
L7	128 Q	1	6
L6	64 Q	Less than 1	7 [0:63]
L5	32 Q	Less than 1	7 [64:95]
L4	16 Q	Less than 1	7 [96:111]
L3	8 Q	Less than 1	7 [112:119]
L2	4 Q	Less than 1	7 [120:123]
L1	2 Q	Less than 1	7 [124:125]
L0	1 Q	Less than 1	7 [126]

6.2 Second design iteration.

We used number of processing elements P = N/16 = 32 as there were many PEs in the idle state during the first iteration in the lower decoding stage (0) so reduced them to N/16 to optimize in the area.

A new memory organization was introduced as we used one memory for each decoding stage as it can be seen in Figure 6.1, each word contains 5 LLRs each of size 14 bits then the memory word length was 70 bits which meets the constraint of the 72 bits word length.

That design iteration was a good one according to the area, but it had 2 major problems. The first one was the latency which increased from 4N = 2048 to 2200 which is considered a performance degradation. The second problem was memory misalignment, it happened when we needed to read LLRs which were not in the same memory word.

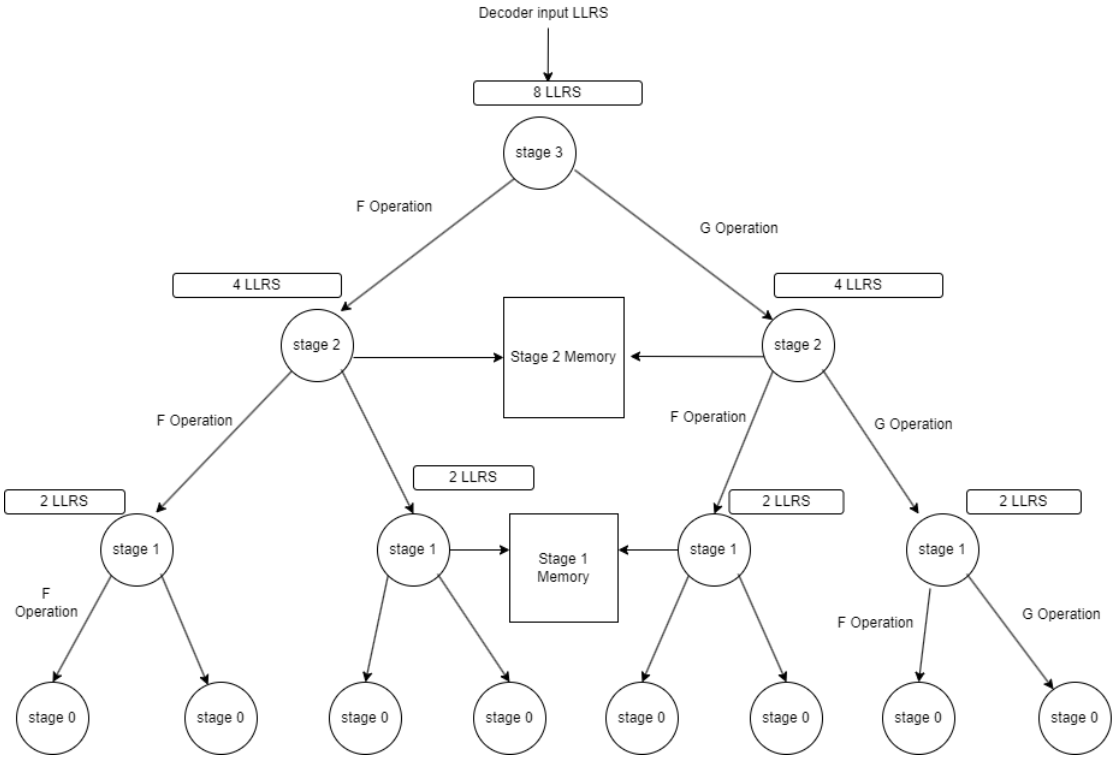


Figure 6.1 - One memory for each stage

6.3 Third design iteration.

The number of processing elements used in this design iteration was P = N/64. The memories used were single port memory for each stage as shown in Figure 6.1 with P LLRs in each word then the number of bits equals to PQ which met the memory word length constraint for channel LLRs but it did not meet the constraint for internal LLRs. The latency increased to 2560 clock cycles which was another performance degrading yet it was acceptable compared to the maximum latency of the decoding process.

6.4 Fourth design iteration.

The number of used processing elements was 5 PEs to further optimize the area as most of the PEs were in the idle in the last decoding stage (stage 0) so it was advisable to remove the unused PEs.

Only one memory was used for each stage as shown in Figure 6.1. Each memory word contains 5 LLRs, therefore the word length became equal to PQ = 70 bits which led to meeting the word length constraint for internal LLRs in this design iteration.

Due to area optimization in the number of PEs and memory organization, the latency increased to 3000 clock cycle which was a huge performance degradation compared to the other design iterations mentioned.

6.5 Fifth design iteration.

4 PEs were used in this design iteration to further optimize the area for the reasons mentioned before. As for the memory organization, two single port RAM were used for each stage starting from the channel LLR memory down to the last stage memory as shown in Figure 6.3 Each memory word contains 5 LLRs with a total word size of PQ = 70 bits.

Latency of this iteration was a first step to a huge success as it was reduced to 2400 clock cycles instead of 3000 clock cycles in the last iteration.

This iteration suffered from complexity in controlling the signals of the data paths due to memory misalignment that happened when it was necessary to read LLRs from two different memory words due to the mismatch between the number of LLRs in one word and the number of the PEs used in that stage as shown in Figure 6.2, hence adding latency for staling everything until the input LLRs for each PE is ready.

We were forced to use only 4 LLRs in each word instead of 5 to avoid the mismatch mentioned earlier which came in favor of the latency as it was reduced to 1570 clock cycles for the whole decoding process which is considered an intermediate latency among the latency mentioned earlier in the previous design iterations and the idle latency of the semi-parallel decoder [14] which consumes huge area compared to the area consumed in this iteration.

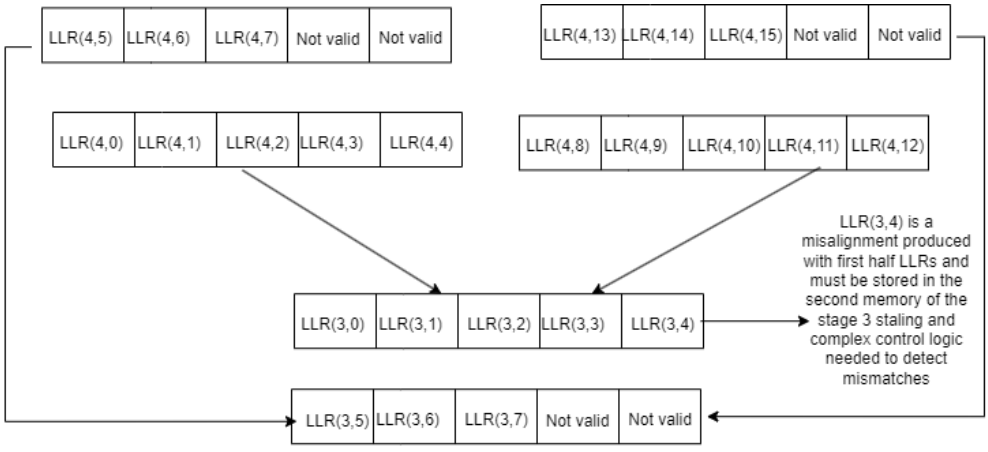


Figure 6.2 - Memory misalignment

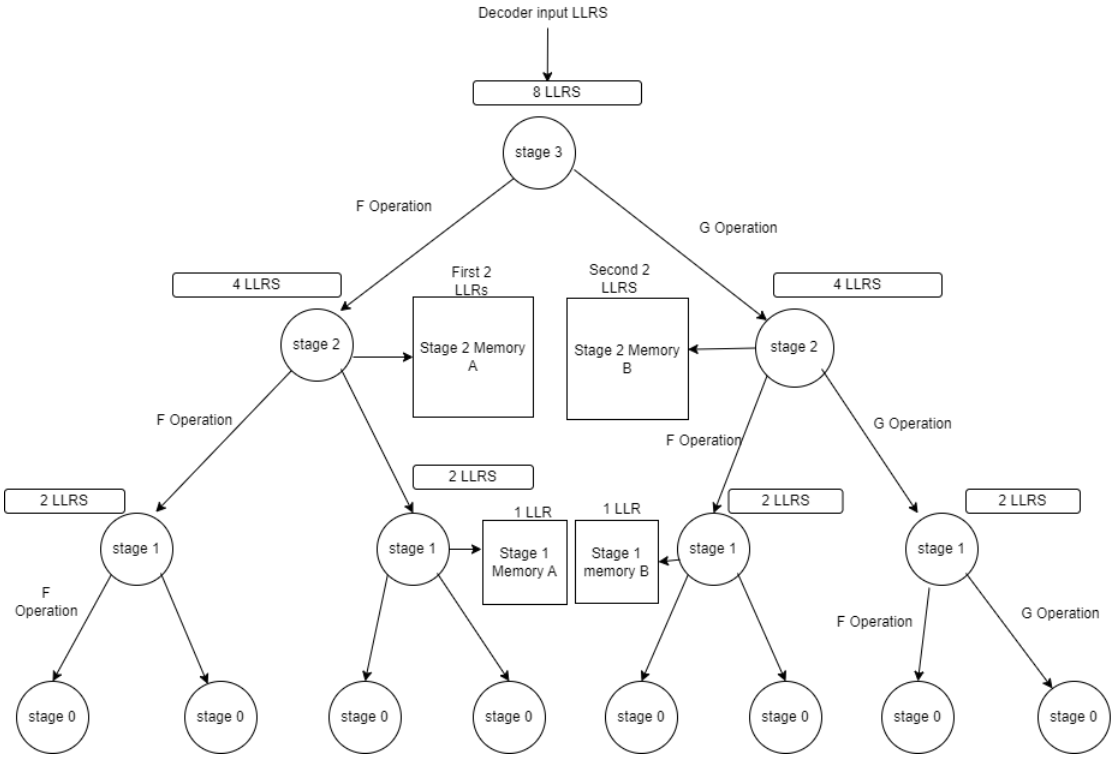


Figure 6.3 - Two memories for each stage.

6.6 Sixth design iteration.

A new fixed-point analysis was introduced in this design iteration Figure 6.4, the channel LLRs are now expressed in 8 bits instead of 7 bits in the previous iterations while internal LLRs are now expressed in 9 bits instead of 14. This reduction in the number of bits gave this iteration a huge advantage over the previous iterations as it decreased the memory word length from 56 to 36 which led to area reduction while keeping the number of processing elements and the memory organization Figure 6.3 the same as before.

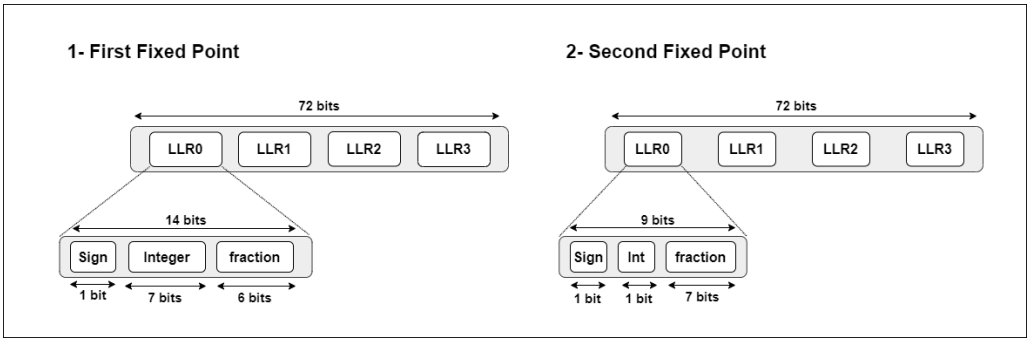


Figure 6.4 - First Vs second fixed point

Chapter 7: SC Proposed Architecture.

7.1 Introduction.

Our aim in this design is to reduce the area as much as possible without severely affecting the latency. This goal was achieved by many methods, the first of which is using four PEs only each of them contains only one function block to perform both F and G functions and choose between them by a function select signal.

The second method is using block RAM instead of register banks which are not feasible with large code length and significantly affects the area and using two memories for each stage to decrease the latency.

The third and final method is to avoid using dual port read memory since its size is nearly double the single port memory in addition to the fact that dual port memories are not available as a BRAM in most FPGAs.

7.2 SC Interface.

SC wrapper as shown in Figure 7.1 contains many input and output ports for interfacing with the outside world. The ports are as follows Table 7.1 - SC port mapping.

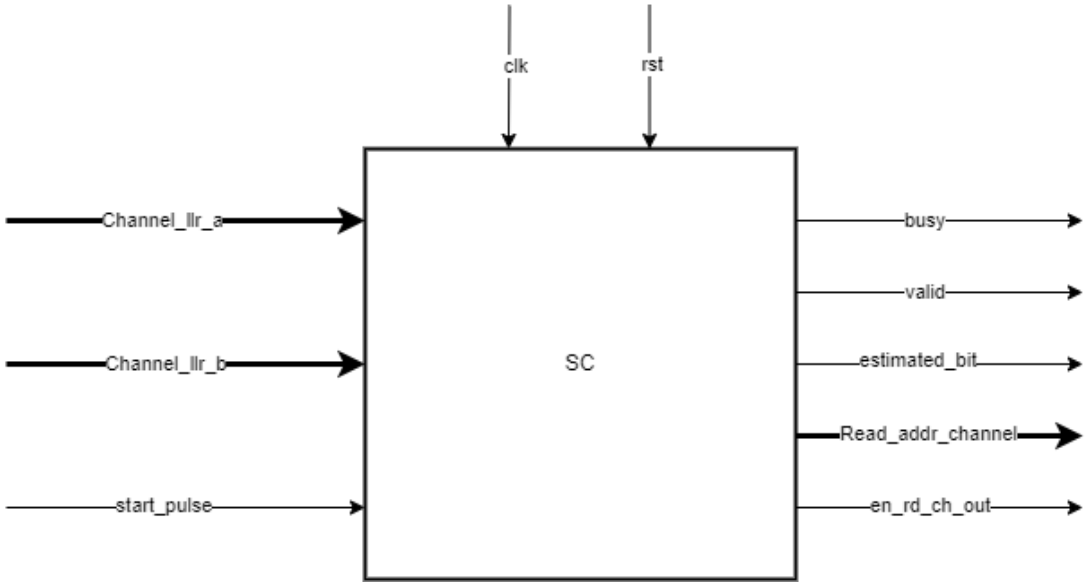


Figure 7.1 - SC interface

Start pulse port function is to trigger the SC decoder to start the decoding operation by reading the channel LLRs from the external memory that is split into two memories which connects the decoder with the preceding block (post FFT).

Busy signal function is to indicate that the decoder has not finished the decoding operation yet so that the decoder cannot be interrupted also the external channel LLR memories cannot be modified until the decoder finishes its operation.

Estimated bit port to store the current decoded bit in the external estimated bits memory to be available for the CRC check block to determine if the received bits are correct or they had been corrupted during the receiving operation.

Valid port to determine whether the estimated bit is a frozen bit or a data bit to be stored in the external memory, where the frozen bits are redundant bits added to the payload to improve the bit error rate.

Channel LLR a and b are two input ports coming from the two channel LLR memories where they are the outputs of the post FFT block.

Table 7.1 - SC port mapping.

Port Name	**Direction**	**Width**	**Parameter**	**Active edge**
channel_llr_a	input	28 bits	WORD_LENGTH_CH	N/A
channel_llr_b	input	28 bits	WORD_LENGTH_CH	N/A
start_pulse	input	1 bit	N/A	high pulse
clk	input	1 bit	N/A	N/A
rst	input	1 bit	N/A	low
busy	output	1 bit	N/A	high
valid	output	1 bit	N/A	high pulse
estimated_bit	output	1 bit	N/A	N/A
Read_addr_channel	output	6 bits	$\log_2(\log_2(N)-1)+2$	N/A
en_rd_ch_out	output	1 bit	N/A	high

7.3 SC Top-level.

As shown in Figure 7.2 the SC module contains many sub-modules each perform a different task in the decoding operation staring from reading the channel LLR memories until the decoding process is done by writing the estimated bits in the external estimated bits memory.

The main sub-modules are:

- Control Unit.
- PE.
- PSN.
- Decision Unit.
- Memory Bank.

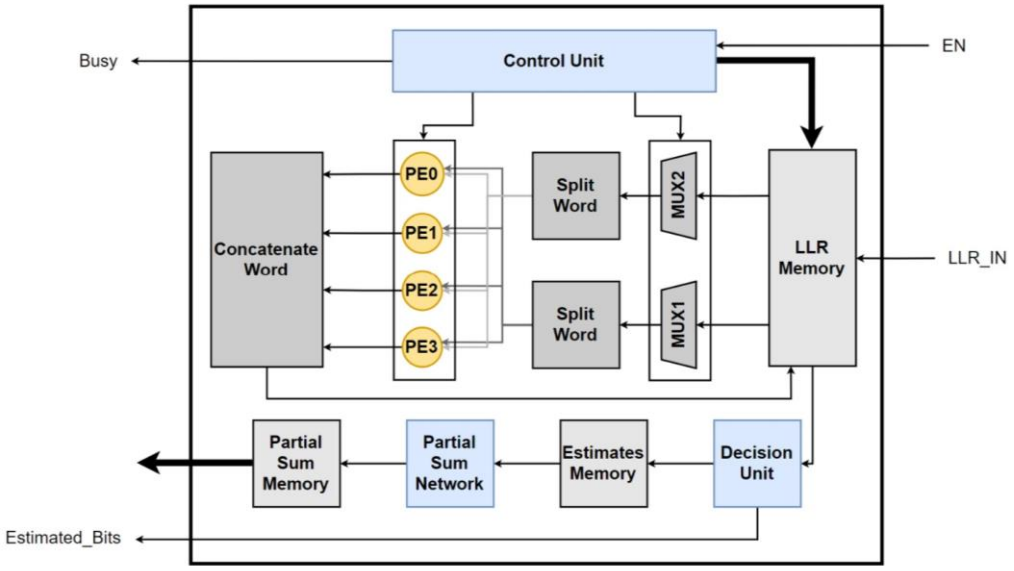


Figure 7.2 - SC top level

7.4 SC Operation Overview.

The operation starts when a start pulse triggers the decoder. Staring with reading the channel LLRs which are sign extended from channel LLR word length to internal LLR word length then use them as an input to the PE to perform either F or G function and output a new LLR to be saved in stage 8 memories, this operation is repeated until we finish the channel LLRs. Each LLR we read from the internal LLR memories or channel LLR memories is fed to the PE through a multiplexing network to multiplex between the memory banks of our decoding stages.

As mentioned before, each word read from the memory contains P LLRs then we need to split them so that each PE takes one of these P as one of its inputs. This is done by a sub-module called split word which takes one word of $P * LLR\ word\ length$ bits as an input and outputs P LLRs each of word length bits, where P is the number of PEs.

Then the operation continues by concatenating the PEs outputs to form a new internal LLR word of size $P * LLR\ word\ length$ to be stored in internal LLR memories of the current stage.

Down to stage 0 where we have to decide whether the bit is '1' or '0', the final LLR of stage 0 is fed to a decision unit which decides on the final LLR sign bit as follows:

$$estimated\ bit = \begin{cases} 1, if\ Sgn\ (LLR) = 1 \\ 0, if\ Sgn\ (LLR) = 0 \end{cases} \quad (7.1)$$

This estimated bit is stored in the estimated bits memory and is fed to the PSN to perform its function and gives out the partial sum which is needed in the next G function as shown in equation (4.3).

All the mentioned modules should be enabled in their proper time, this is done by the control unit which is also responsible for memory address generation and determining current stage number and current decoded bit index. It also decides the function select signal for the PE to perform the intended function for that part of stage, in addition to the selection lines of the multiplexing network mentioned earlier.

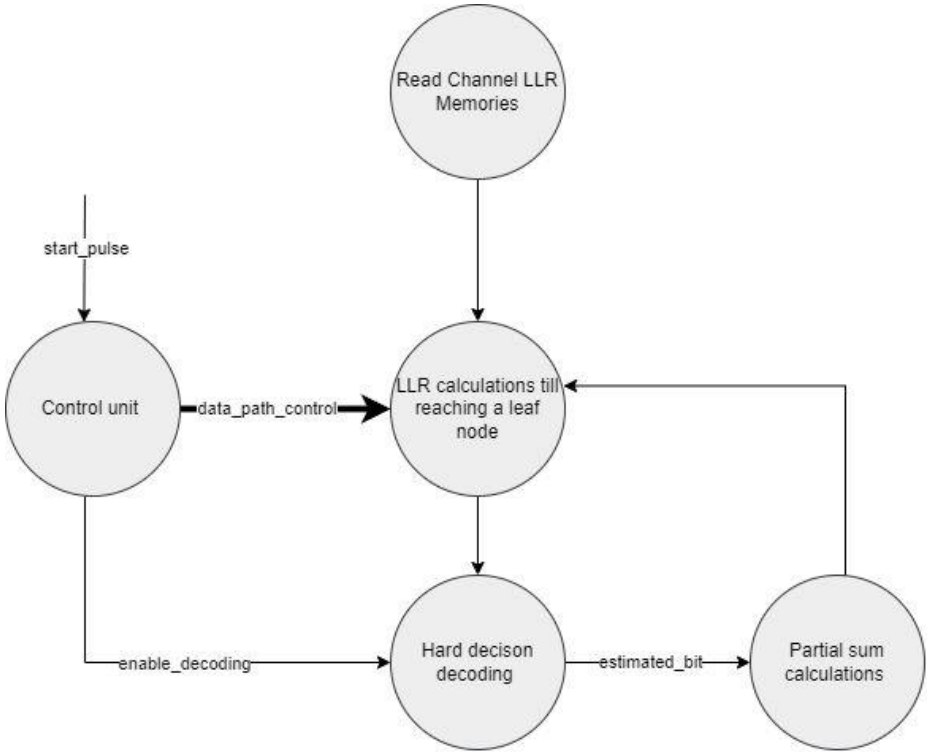


Figure 7.3 - SC Operation.

7.5 Control Unit Sub-Module.

7.5.1 Module Description.

As it can be seen in Figure 7.4 ,the controlling operation of the SC decoder can be expressed in controlling three counters each of which for a specific task to be done in the decoding operation. It starts its operation when the decoder is triggered by an external start pulse, then it continues until all decoded bits done.

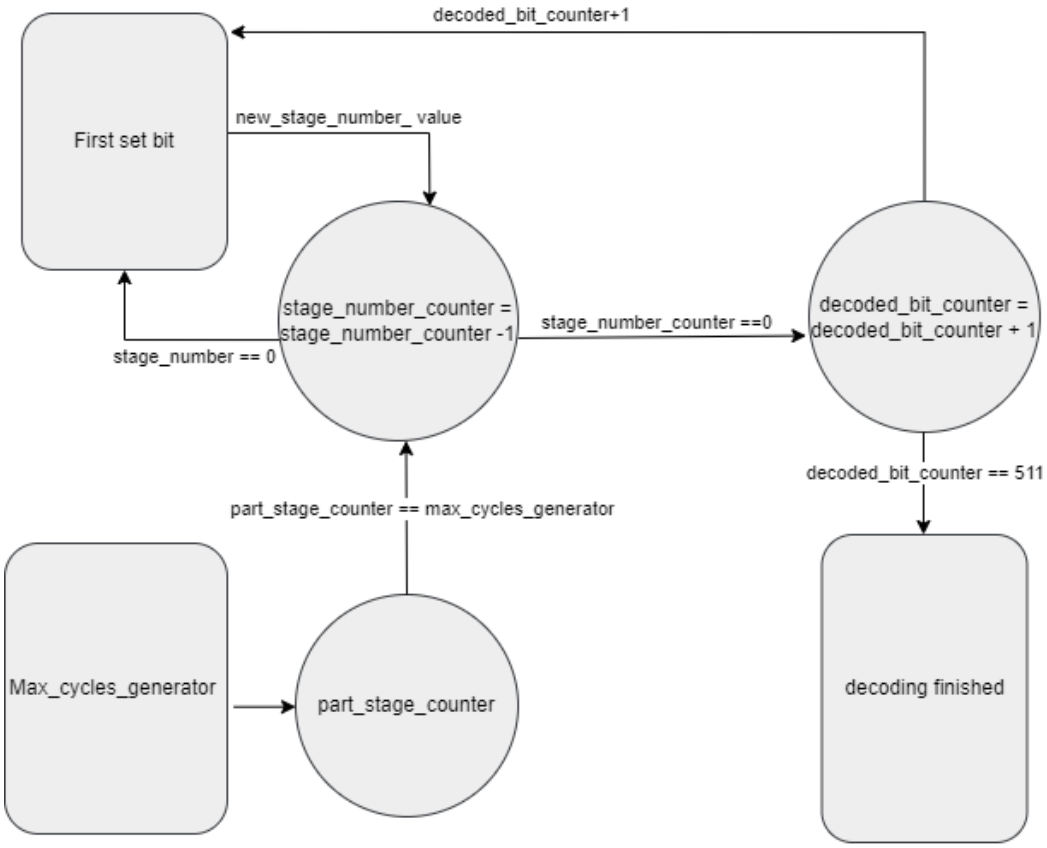


Figure 7.4 - Control Unit Operation

The first counter is an incremental counter for the part of stage index, used to count the number of cycles inside each stage depending on the number of processing elements. It starts from zero until it reaches the number of needed cycles in that stage, where each clock cycle includes memory read, PE operation and memory write.

$$PS = \frac{2^l}{P} \; where \; l \; is \; the \; stage \; number \; and \; P \; is \; the \; number \; of \; PEs \qquad (7.2)$$

The second counter is a decremental counter for the current decoding stage index. It starts from stage 8 which is the highest node in the decoding tree down to stage zero which is an estimate node for the current decoded bit. After that it goes back to a higher node depending on a function called FFS as shown in equation (7.3) which decides the index of the first leading one in the next decoded bit index. Hence the stage index is loaded with the return value of the FFS function.

$$FFS(i_{m-1}i_{m-2} \ldots . . i_0) = \begin{cases} \min(m) : i_m = 1 & , if \; i > 0 \\ m - 1 & , if \; i = 0 \end{cases} \qquad (7.3)$$

$$where \; i \; is \; the \; decoded \; bit \; index \; and \; m \; is \; its \; width$$

The third counter is an incremental counter for the decoded bit index, its value changes when the stage index reaches zero, to indicate reaching a leaf node. Upon

reaching stage zero, the decision unit is activated to take a decision on the final LLR to estimate whether the bit is ‘0’ or ‘1’.

The control unit is also responsible for address generation for both read and write operations depending on the stage number as it can be seen in …, in addition to the enable signals of those memories. Moreover, it is also responsible for choosing which memory is feeding the PE with LLRs by controlling the selection lines of the multiplexing network.

7.5.2 Port Mapping.

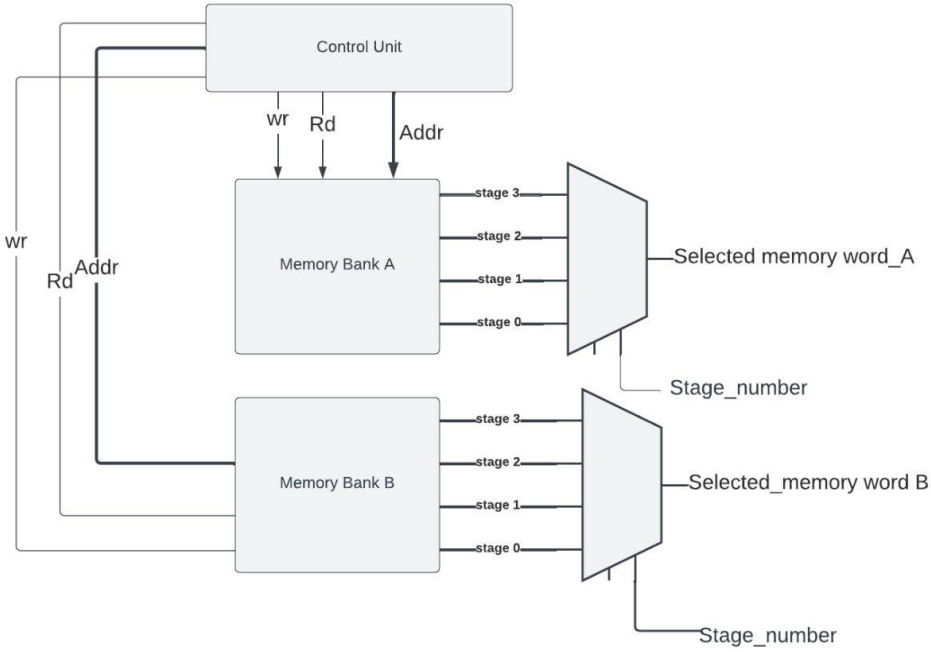


Figure 7.5 - CU Multiplexing for N=8.

Table 7.2 - CU port mapping

Port Name	Direction	Width	Parameter
clk	Input	1	N/A
rst	Input	1	N/A
start	Input	1	N/A
busy	output	1	N/A
en_Wr_8_a	output	1	N/A
en_Wr_7_a	output	1	N/A
en_Wr_6_a	output	1	N/A
en_Wr_5_a	output	1	N/A
en_Wr_4_a	output	1	N/A
en_Wr_3_a	output	1	N/A
en_Wr_2_a	output	1	N/A
en_Wr_1_a	output	1	N/A
en_Wr_0_a	output	1	N/A

en_Wr_8_b	output	1	N/A
en_Wr_7_b	output	1	N/A
en_Wr_6_b	output	1	N/A
en_Wr_5_b	output	1	N/A
en_Wr_4_b	output	1	N/A
en_Wr_3_b	output	1	N/A
en_Wr_2_b	output	1	N/A
en_Wr_1_b	output	1	N/A
en_Wr_0_b	output	1	N/A
en_rd_ch	output	1	N/A
en_rd_7	output	1	N/A
en_rd_6	output	1	N/A
en_rd_5	output	1	N/A
en_rd_4	output	1	N/A
en_rd_3	output	1	N/A
en_rd_2	output	1	N/A
en_rd_1	output	1	N/A
enable_psn	output	1	N/A
address_psn	output	7 bits	N/A
stage_index	output	4 bits	$\log_2(\log_2(N)-1)$
part_stage_index	output	6 bits	N/A
dec_bit_index	output	8 bits	$\log_2(N)-1$

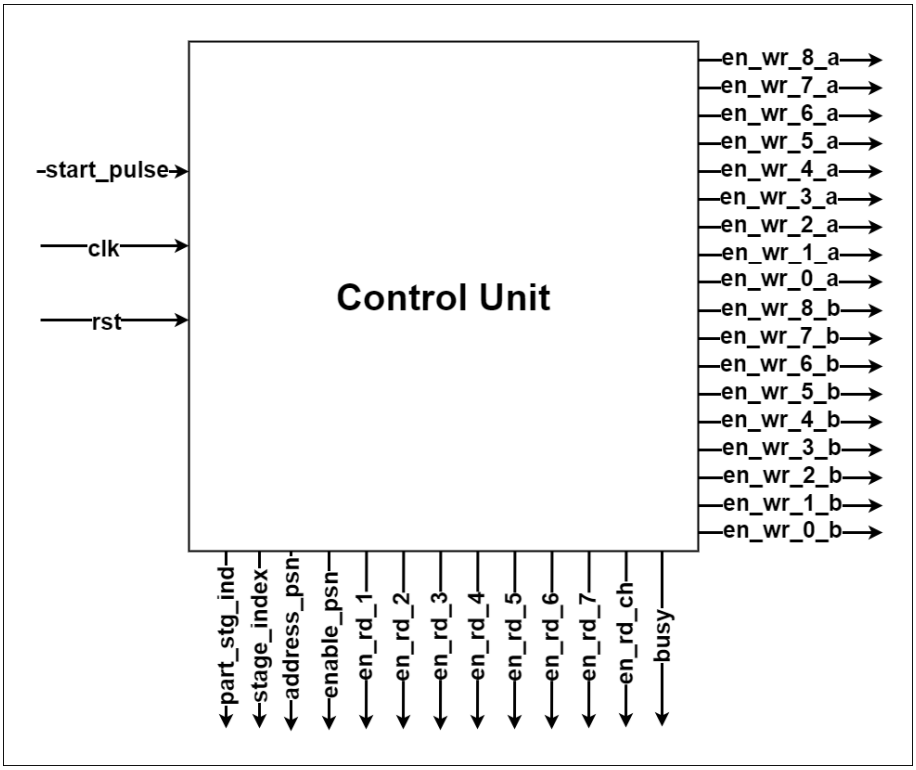


Figure 7.6 - CU port mapping

7.6 Decision Unit Sub-Module.

7.6.1 Module Description.

Error! Reference source not found. illustrates that this module is the estimated bit decision making module based on the final LLR of stage zero which is the input to this module. The output of this module is the estimated bit which is either ‘0’ or ‘1’ as demonstrated in equation (7.1). Hence, the estimated bit is stored in the estimated bit memory and used to calculate the next partial sum in the PSN sub-module.

This module includes a ROM which stores frozen bits indices, during the estimation operation, we compare the decoded bit index with its corresponding index in that ROM, if that index contains a value equals to 1 then this is a data bit and is decoded according to equation (7.1), on the other hand, if it contains a value equals to 0, then this bit is decoded to zero automatically since that was an indication that it is a frozen bit.

The decision unit outputs a valid flag indicating that the currently decoded bit is a data bit not a frozen bit. This signal is an output from the SC decoder as mention in Table 7.1 to the external decodes bit memory.

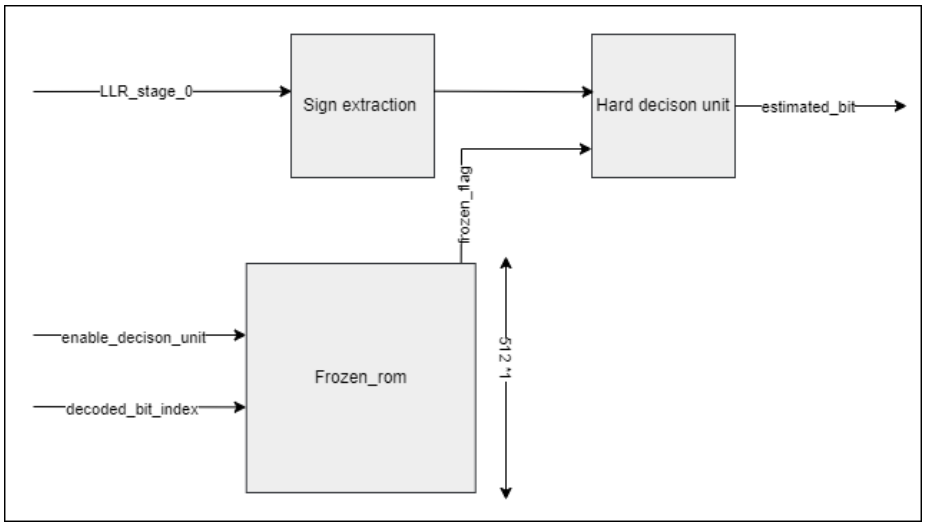


Figure 7.7 - DU operation

7.6.2 Port Mapping

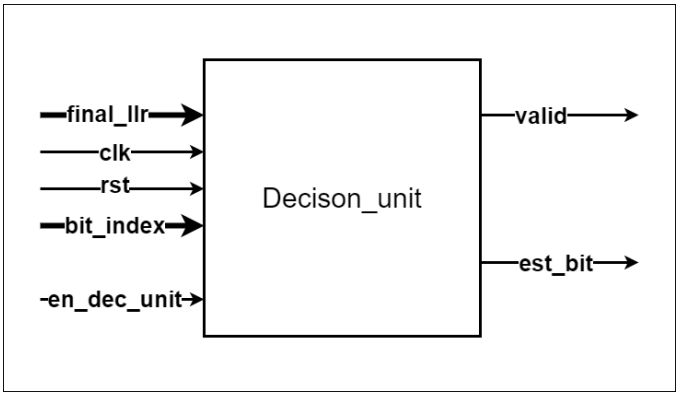


Figure 7.8 - DU port mapping

Table 7.3 - DU port mapping

Port Name	Direction	Width	Parameter
clk	Input	1	N/A
rst	Input	1	N/A
dec_bit_index	Input	8 bits	$\log_2(N) - 1$
en_dec_unit	Input	1	N/A
valid	Output	1	N/A
est_bit	Output	1	N/A

7.7 Processing Element Sub-Module.

7.7.1 Module Description.

The PE sub-module (Figure 4.6) is considered an internal core inside the SC decoder, it is in charge of the LLR calculations. PE deals with the LLRs in sign and magnitude format, since that format produces synthesis results 20% better than that of two's complement format as it was reported in [14].

It performs both F and G operations according to equations (4.2) and (4.3) where the sign and magnitude are driven by a multiplexer with function select as its selection line.

$$\psi(\lambda_{L_{l,i}}) = \begin{cases} \psi(\lambda_f)\ when\ B(L,i) = 0 \\ \psi(\lambda_g)\ otherwise \end{cases} \tag{5.4}$$

$$\left|\lambda_{L_{l,i}}\right| = \begin{cases} |\lambda_f|\ when\ B(L,i) = 0 \\ |\lambda_g| otherwise \end{cases} \tag{5.5}$$

Where the F function takes the 2 input LLRs only as it can be seen in Figure 7.10 while the G function takes the 2 input LLRs and a partial sum input as it can be seen in Figure 7.9.

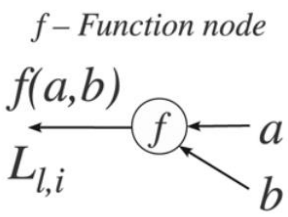


Figure 7.10 - F function node

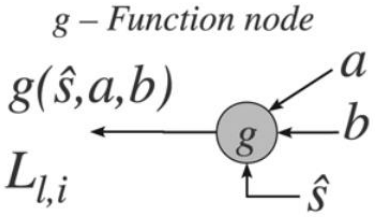


Figure 7.9 - G function node

These operations are performed using a single XOR gate and a comparator in addition to a SM adder/subtractor needed to perform G operation. The relationship between the magnitude of the input LLRs is represented by equation (5.6), this parameter γ_{ab} is used as a multiplexer select to choose between the maximum and the minimum of the magnitude of the 2 input LLRs.

$$\gamma_{ab} = \begin{cases} 1 \; if \; |\lambda_a| > |\lambda_b| \\ 0, otherwise \end{cases} \tag{5.6}$$

In case of performing F function, we choose the minimum magnitude of the 2 LLRs, where the output sign is computed by performing an XOR operation on the signs of the 2 LLRs, however in case of G function we choose the maximum magnitude of the 2 LLRs where the output sign is computed by performing an XOR operation on the signs of the 2 LLRs in addition to the partial sum coming from the PSN.

We can summarize the PE calculations for both F and G functions using the following Boolean equations

$$\psi(\lambda_g) = \overline{\gamma_{ab}}.\psi(\lambda_b) + \gamma_{ab}.(\hat{s} \oplus \psi(\lambda_a)) \tag{5.7}$$

$$|\lambda_g| = max(|\lambda_a|, |\lambda_b|) + (-1)^X min(|\lambda_a|, |\lambda_b|) \tag{5.8}$$

$$X = \hat{s} \oplus \psi(\lambda_a) \oplus \psi(\lambda_b) \tag{5.9}$$

$$\psi(\lambda_f) = \psi(\lambda_b) \oplus \psi(\lambda_a) \tag{5.10}$$

$$|\lambda_f| = \min(|\lambda_a|, |\lambda_b|) \tag{5.11}$$

7.7.2 Port Mapping.

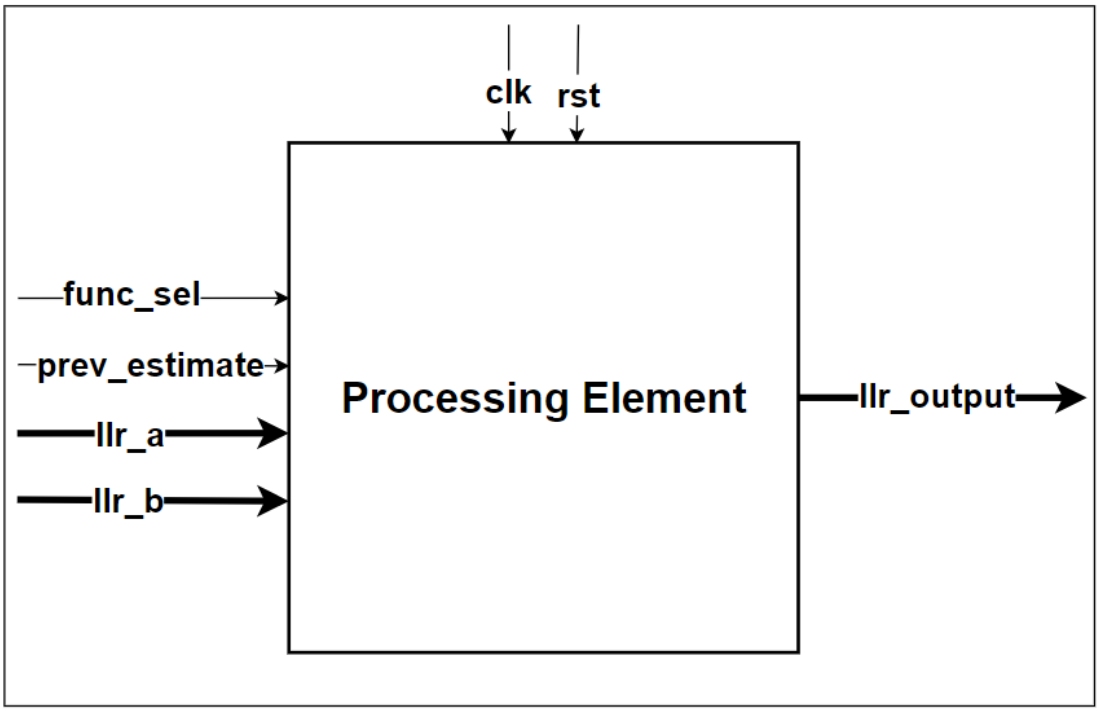


Figure 7.11 - PE port mapping

As shown in Figure 7.11 Where previous estimate is the output of the PSN calculated from the previous estimated bit.

Table 7.4 - PE port mapping

Port Name	Direction	Width	Parameter
Func_sel	Input	1 bit	N/A
Previous_estimate	Input	1 bit	N/A
llr_a	Input	14 bits	WORD_LENGTH
llr_b	Input	14 bits	WORD_LENGTH
llr_output	Output	14 bits	WORD_LENGTH

7.7.3 Fixed Point Modifications.

For further area optimization, the word length size is reduced for the internal LLRs from 14 bits to 9 bits so the size of the internal memories is reduced in addition to the data bus width.

It was required to detect overflow of the LLRs so as to prevent them from exceeding the maximum number that can be expressed in 9 bits. That was performed by a sub-module named overflow check which asserts a flag when it detects an overflow in LLRs of stages 5,6,7 or 8 as the quantization error in these mentioned stages cannot be bearable. That flag is input to another sub-module named divider which is used to divide the input LLRs to the PE by 2 then we perform quantization operation on the divider output to ensure that the input LLRs to the PE is saturated if they still exceeding the maximum number to be expressed in 9 bits. Any other stage would require quantization only.

7.8 Partial Sum Network.

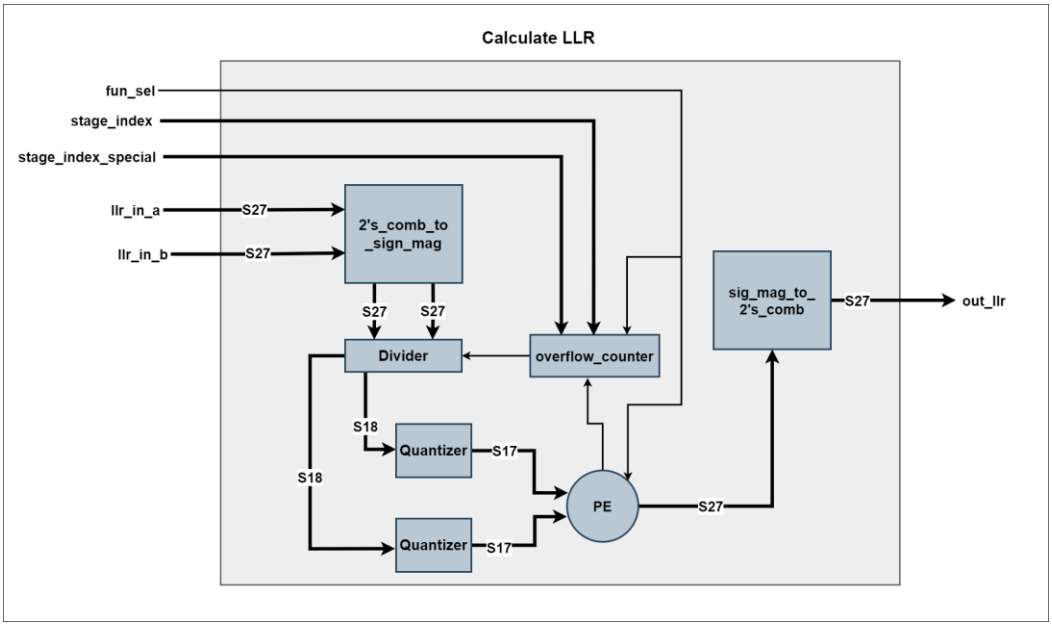


Figure 7.12 - PE after fixed point modifications

7.8.1 Module Description.

As previously mentioned, The SC decoder consists of 3 main components which are the control unit, PSN, PE, memory banks. The PSN unit is responsible for calculating the partial sums required by the PEs to calculate the G function as it takes 3 inputs 2 LLRs and 1 partial sum as shown in Figure 7.9.

It was reported in [14] that in the synthesis result of the semi-parallel decoder, the memory banks take about 75% of the total decoder area and the rest is occupied by the PSN, in addition to the critical path of the SC decoder is in the PSN which will impact the maximum frequency as it decreases as N increases.

There are $\frac{N}{2} * \log_2(N)$ partial sums to be calculated throughout the decoding process. When a new bit is decoded, the PSN should update all the partial sums that includes this bit as shown in Figure 7.13. For example, when $\widehat{u_4}$ is decoded, the partial sums $S_{4,0}\ and\ S_{4,1}$ should be updated while the partial sums that do not contain the last decoded bit should keep their values unchanged.

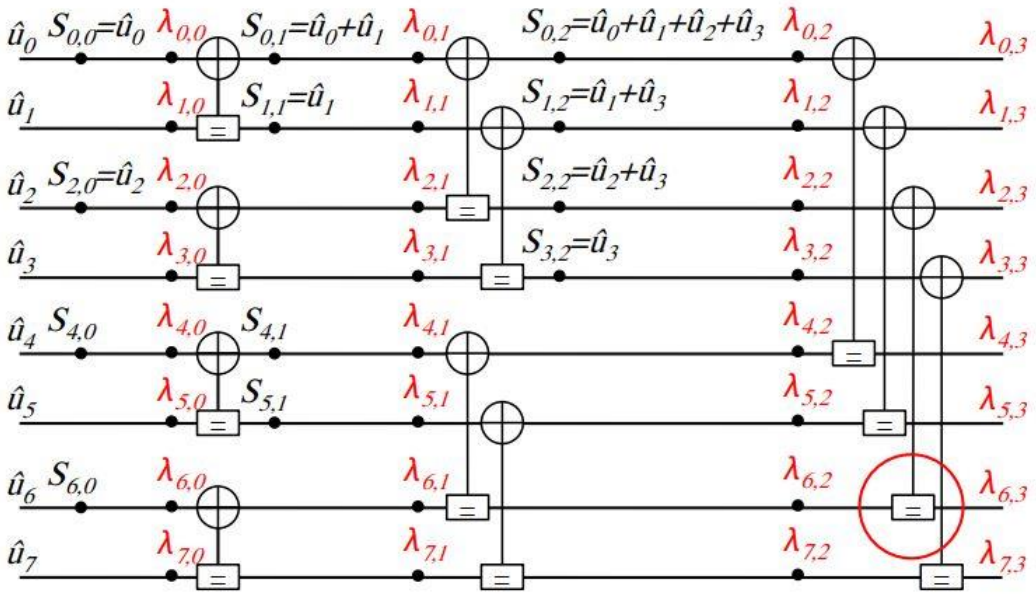


Figure 7.13 - Partial sums calculations

It was clarified in [14] that the PSN required storage was reduced from $\frac{N}{2} * \log_2(N)$ to $N-1$, in addition to introducing the indicator function which was defined to specify which DFF should be updated with the current decoded bit. The indicator function was implemented as a combinational logic and its hardware complexity increases linearly with N so it would add area overhead for large code length. Also, it would make the critical path worse.

A new technique was used for implementing the PSN based on the idea of LFSR to replace the indicator function PSN of the semi-parallel decoder. For the best of our knowledge, this technique was only tested with the line decoder [15] and was not integrated with the semi-parallel decoder. This technique (SR-PSN) has a decreased critical path and provide a better performance than the indicator function.

It was stated in [15] that a SR-PSN of size $\frac{N}{2}$ was implemented by $\frac{N}{2}$ DFFs, $\frac{N}{2}$ AND gates, $\frac{N}{2}$ XOR gates and a matrix generation unit as shown in Figure 7.14. Whenever a new bit is decoded, the SR-PSN is activated in addition to the matrix generation unit and the partial sums are updated according to the equation (5.12).

$$\begin{cases} R_0 \leftarrow \widehat{u_i}\ AND\ C_{i,0} \\ R_k \leftarrow R_{k-1}\ XOR\ \left(\widehat{u_i}\ AND\ C_{i,0}\right), if\ k > 0 \end{cases} \tag{5.12}$$

The main contributor in this SR-PSN architecture is the matrix generation unit

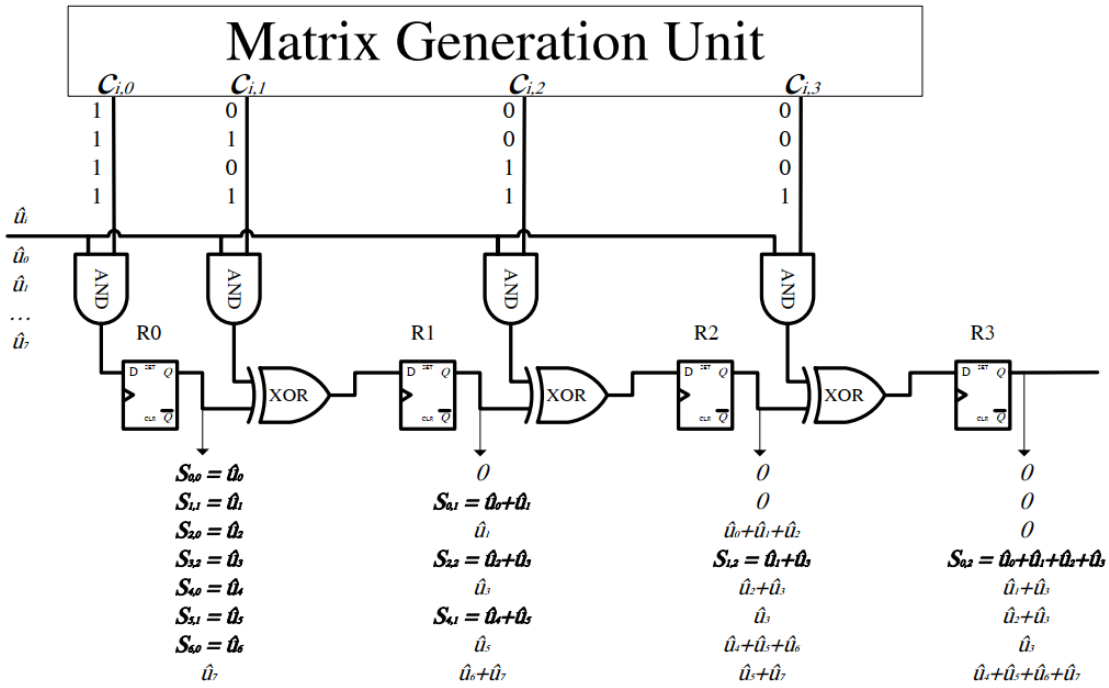


Figure 7.14 - SR-PSN internal structure

which is responsible for the generation of the Kronecker matrix which was used in the encoding process so the same matrix has to be used in the decoding operation.

The matrix generation unit could have been implemented using a $\frac{N}{2} * N$ ROM, which will produce a 16 KB memory for $N = 512$ that would affect the area severely. Therefore, in [15] LFSR of size $\frac{N}{2}$ as it can be seen in Figure 7.15 was used to overcome this area overhead. The LFSR is updated whenever a new estimate is produced according to the following equations.

$$\begin{cases} C_{i,0} = 1\,, 0 \le i \le N-1 \\ C_{i+1,k} = C_{i,k-1}\ XOR\ C_{i,k}\ \ , 0 \le i < N-1\ and\ 1 \le k \le \frac{N}{2} - 1 \end{cases} \tag{5.13}$$

$$\begin{cases} M_0 \leftarrow 1 \\ M_k \leftarrow M_k\ XOR\ M_{k-1}\ if\ k > 0 \end{cases} \tag{5.14}$$

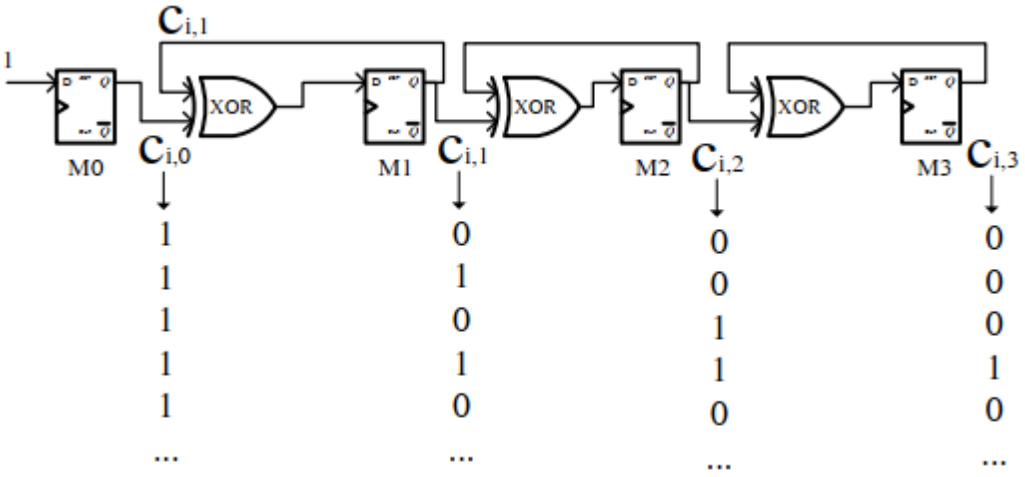


Figure 7.15 - Matrix generation unit

The SR-PSN produces $\frac{N}{2}$ partial sums and in our decoder, we only use 4 PEs, so we needed to use a memory to store them as it can be seen in Figure 7.17 and route them to the PE properly according to their indices.

To use the SR-PSN with the semi-parallel decoder, a multiplexing network is needed since the output partial sums must be ordered before they can be used as inputs to the PEs as shown in Figure 7.16.

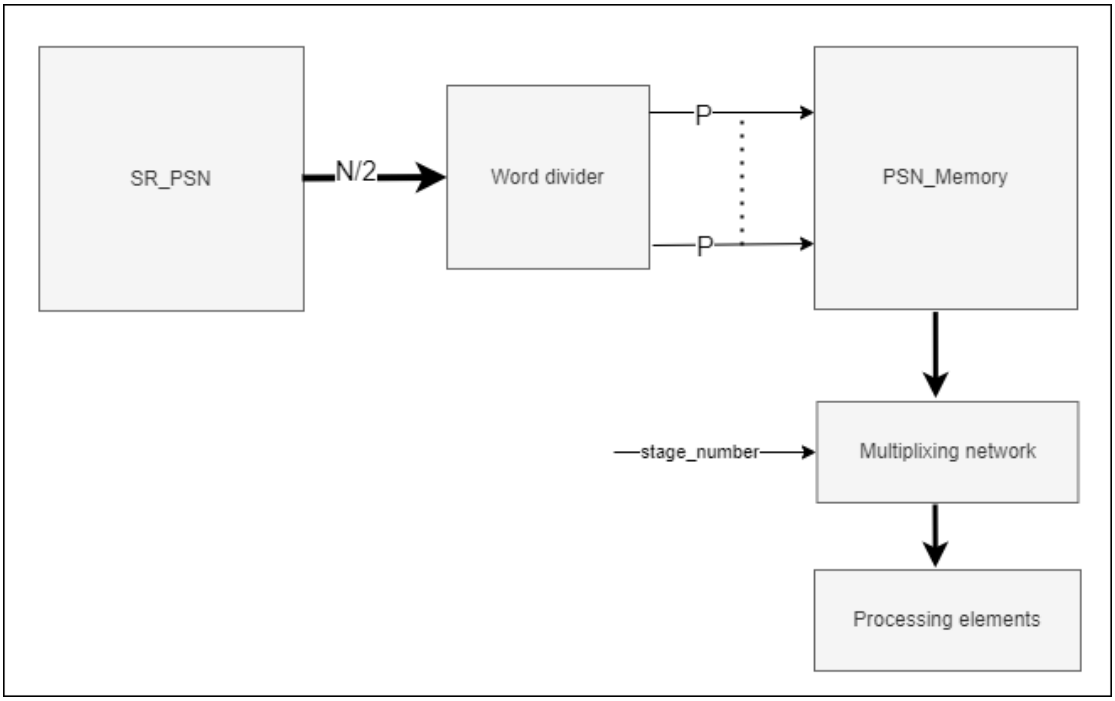


Figure 7.16 - PSN operation

To summarize, The PSN sub-module contains 3 essential modules in addition to AND and XOR gates. These essential modules are the matrix generation unit, SR and a memory to store the output partial sums to be passed to the PEs whenever needed.

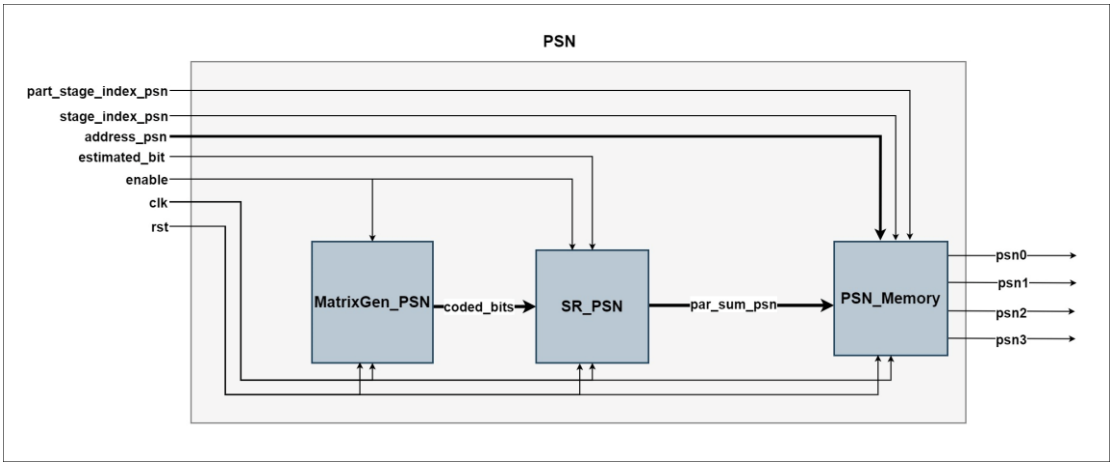


Figure 7.17 - PSN internal structure

7.8.2 Port Mapping.

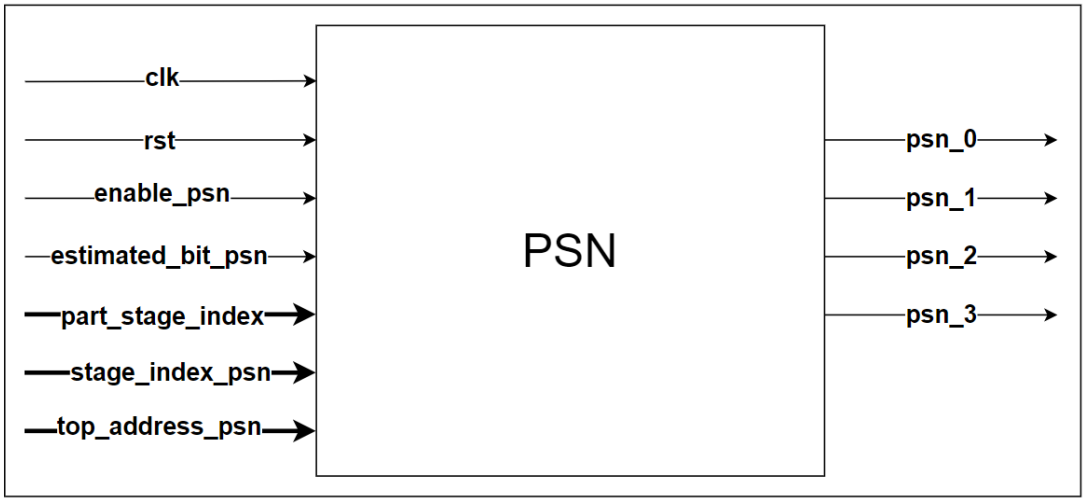


Figure 7.18 - PSN port mapping

Table 7.5 - PSN port mapping

Port	Direction	Width	Parameter
clk	Input	1 bit	N/A
rst	Input	1 bit	N/A
enable_psn	Input	1 bit	N/A
estimated_bit_psn	Input	1 bit	N/A
part_stage_index	Input	7 bits	$\log_2(\frac{2^l}{p})$
stage_index_psn	Input	4 bits	$\log_2(\log_2(N)-1)$
top_address_psn	Input	7 bits	N/A

7.9 SC Pipelined

As shown in Figure 7.19 the data path of SC decoder consists of a multiplexing network and processing element. As a conservative approach we decided to break this combinational path so we can prevent any timing violation when we reach synthesis stage. Thus, we modified the data path and accordingly we needed to schedule the read and write enables of the memories in addition to the addresses so we can synchronize the decoding operation.

As shown in Figure 7.19 , the first modification to the data path is adding a flip flop to the output of the processing element to break the pipeline of the data path, since there is a new added clock cycle latency we will need to delay the write and read enables of the memories to prevent storing or reading invalid data as a result of this change each LLR is computed and stored in 3 cycles due to the pipelining.

Another modification to the SC decoder, we will add a flip flop to delay the partial sums produced by the PSN so that they become synchronized with the processing element at the same stage and operation.

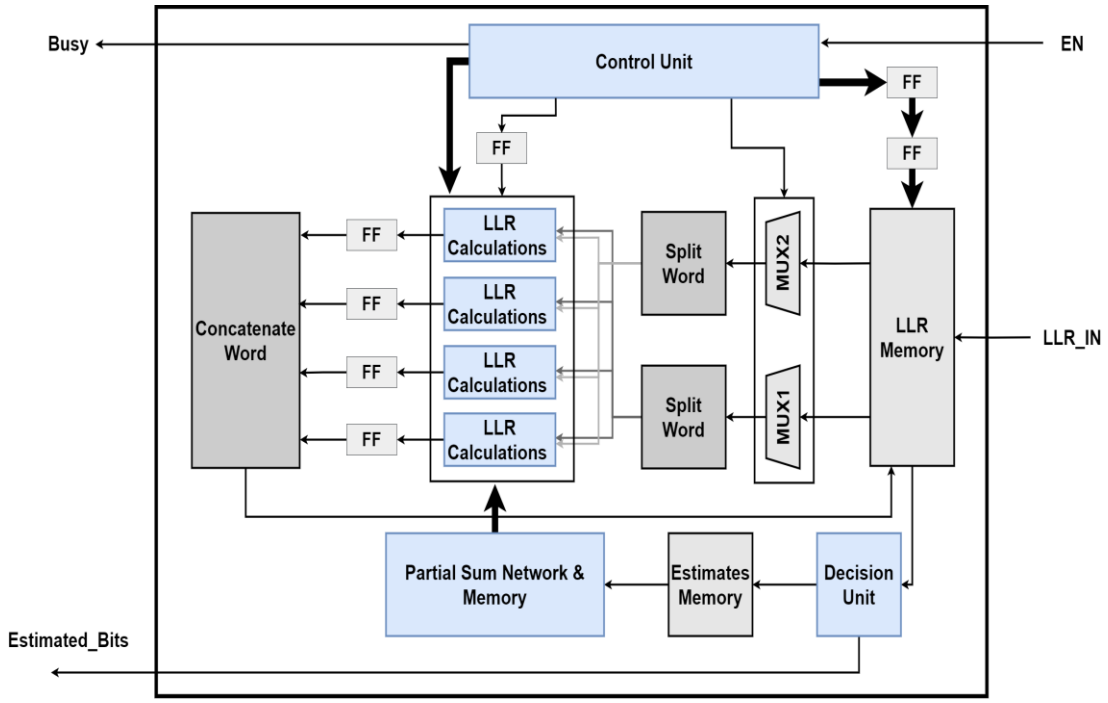


Figure 7.19 SC Pipelined Top level

The last modification to the SC decoder in the pipelined register is registering the output of any RAM in the design which was necessary to synchronize the read operation to be compatible with the pipelining modification. This change will make the memories compatible with most FPGAs as most of them doesn't have asynchronous read BRAMs. However, if the BRAM is asynchronous the FPGA can simply modify it by adding a register to the output and we will still utilize the BRAM, the synchronous BRAM cannot be modified to become asynchronous BRAM. Therefore, the synthesis engine will use the logic fabric to create a memory which is a waste of resources.

As shown in Figure 7.20 in the start of the operation of the pipelined decoder the first 3 cycles contain 3 separate operations memory read, LLR calculations, memory write. Then as shown in Figure 7.21 after passing the first 3 clock cycles the actual pipeline will start and the 3 operations will move simultaneously till the end of the decoding process therefore the throughput will increase as expected due to the pipelining modifications.

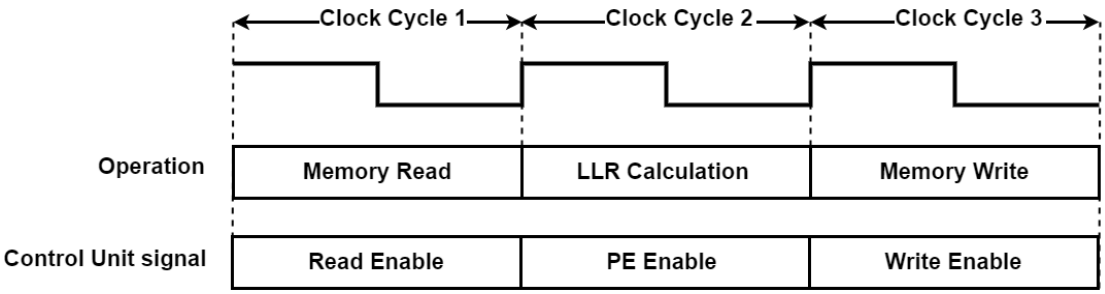


Figure 7.20 First 3 cycles in pipeline operation

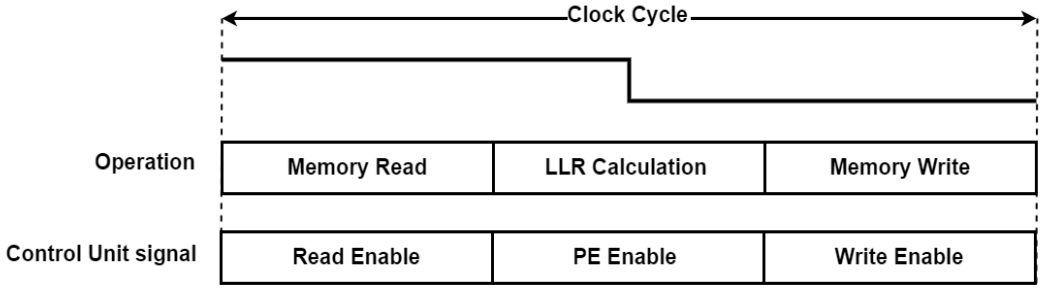


Figure 7.21 Normal pipeline operation

7.10 SC Block level verification:

To properly verify the design, that the hardware matches the MATLAB model using test cases generated from the model sweeping over the SNR range from -12 to 5 dB and we need to check the internal LLR values and the output bits from the decoding process.

We tested the design with 1500 test cases over the SNR range with no failed test cases as shown in Fig 7.22 and Fig 7.23. The verification also tested the decoder capability of decoding multiple frames.

```
# test case finished sucessfully all internal LLRs match the test case
# test bench is sucessfull completely
# All test cases are sucessfull
# ** Note: $stop    : Top_Module_tb.v(228)
#    Time: 37165 ns  Iteration: 12  Instance: /Top_Module_tb
# Break in Module Top_Module_tb at Top_Module_tb.v line 228
```

Figure 7.22 - Test bench outputs

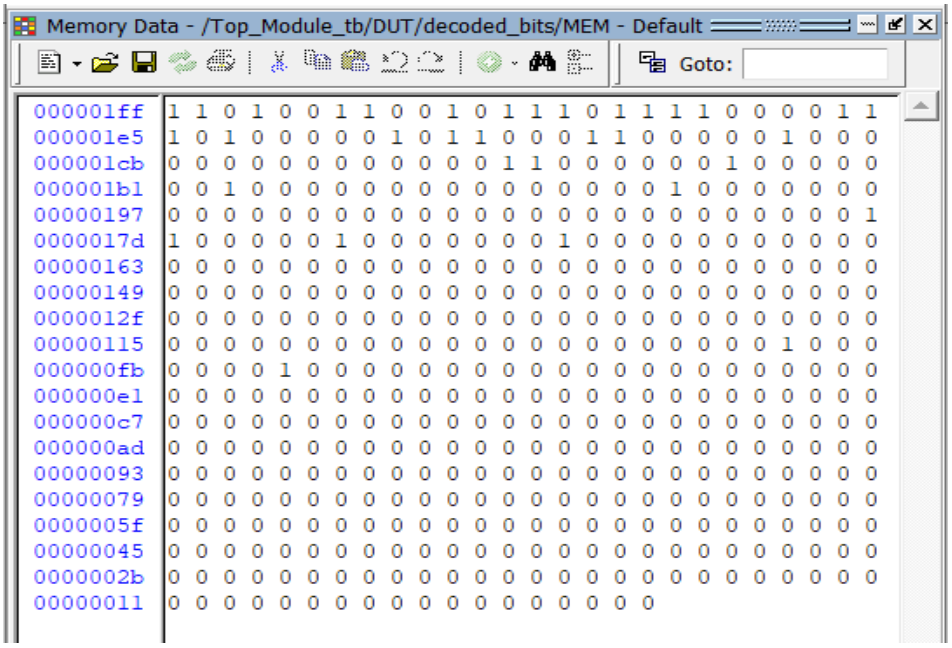


Figure 7.23 Output decoded bits

Chapter 8: SCL Proposed Architecture

8.1 Introduction.

This architecture is based on the SC decoder as a building block where four SC decoders were used. Hence during the decoding operation there were eight paths I some cases and we have to choose only four of them according to a metric called path metric.

During the design phase of this architecture, there was two approaches, the first is to modify the SC decoder to operate as a SCL decoder with pointer memory and a multiplexing network to decide which PE reads from which memory. This design approach has some major problems as it does not utilize the SC decoder as a black box, its pointer introduces extra hardware complexity and it is not easily scalable if it is required to increase the number of decoders [11].

The second approach is to use the SC decoder as a black box with a master control unit which is implemented using a FSM, in addition to some modules such as a metric sorter and another module for the copying logic to handle the operation between the decoders and a multiplexing network between these L decoders. This design approach does not introduce much hardware complexity compared to the previous one. It utilizes the SC decoder as a black box which makes this approach scalable if it is required to increase the number of paths.

The internal blocks or modules of the SC decoder were modified to match the new architecture and be suitable for copying to match the SCL algorithm. The modified blocks were the SC control unit, PSN, and the decision unit which had the greatest modification as it is responsible for the PM calculations.

The SCL is integrated with the CRC block which is in charge of detecting the payload errors after receiving the data which passes through AWGN channel which may deteriorate the data. That deterioration leads to decoding errors.

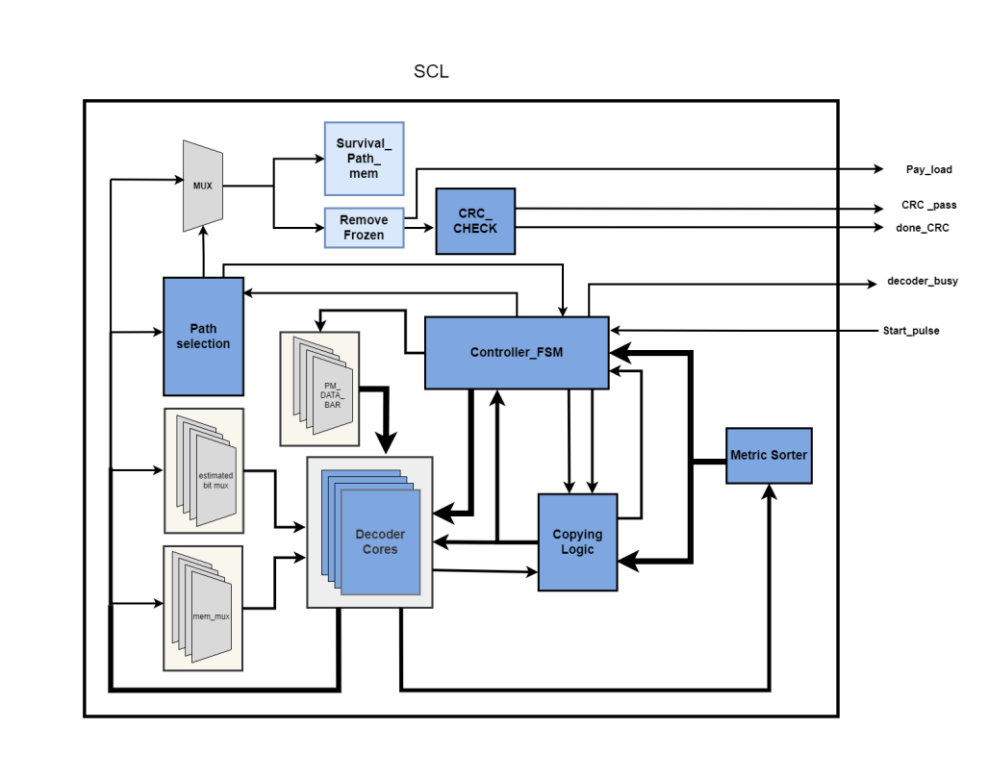


Figure 8.1 - SCL architecture

8.2 Operation Overview.

The operation starts by triggering the SCL with a start pulse. When the operation starts, all the decoders perform the same operation for the frozen bits until reaching the first data bit where the decoders split into two groups. upon reaching the second data bits, the two groups split into four groups which are split into eight groups after decoding the third data bit. Each decoder contains two paths and calculates a path metric for each one of them. These path metrics are arranged ascendingly by a module called metric sorter where only the least four path metrics are allowed to continue the decoding operation. If any decoder contains more than one of the survival paths, then copying is required from that decoder to any other decoder which does not contain any of the survival paths.

After the decoding operation is done, we have to choose which path contains the correct payload, this is done based on the PM sorting. After choosing, we have to sperate the data bits from the frozen bits and store the data bits in an external memory. Hence, the CRC is activated to indicate whether the decoded payload is received without any errors or there were errors during the data transmission in the AWGN channel.

The list decoding operation is controlled by a FSM which passes through many states including the normal operation state, copying state and CRC state.

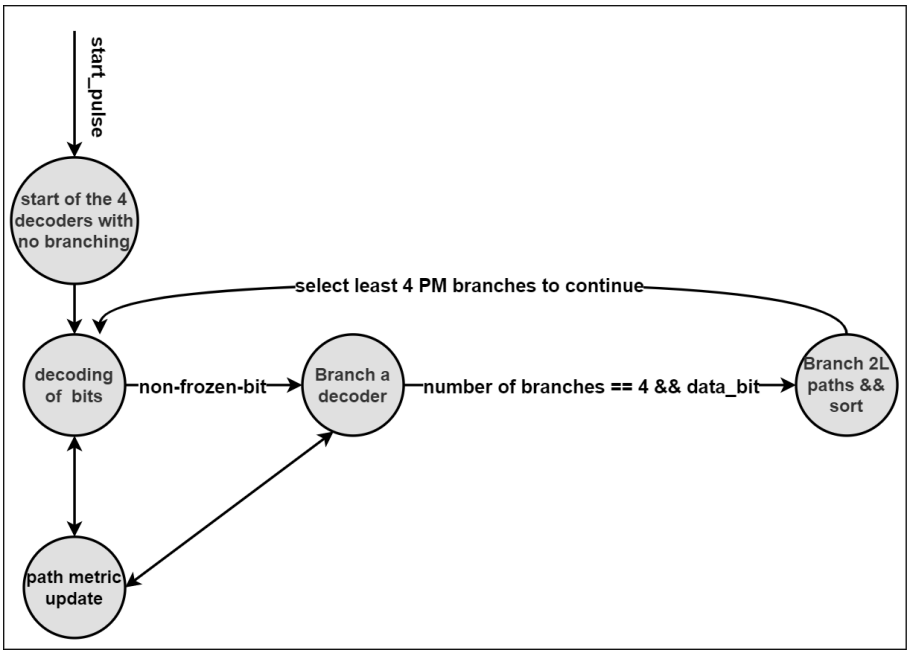


Figure 8.2 - SCL operation

8.3 Controller FSM Sub-Module.

8.3.1 Module Description.

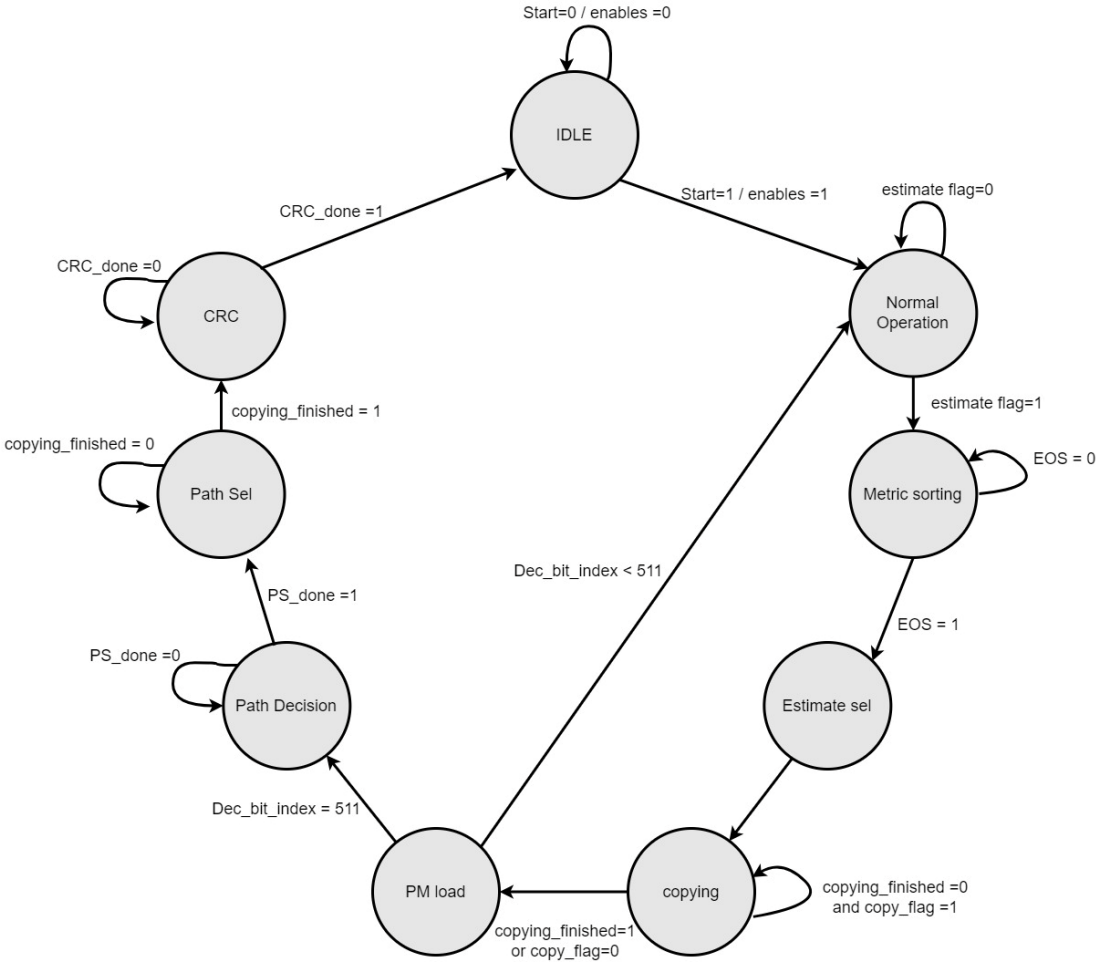


Figure 8.3 FSM of SCL Top Controller

The operation starts in the Idle state waiting for a start pulse to trigger the SCL decoder. When the SCL is triggered, we move to the normal operation state where actual decoding takes place by enabling all the SC decoders. The FSM keeps the normal operation state until the available paths become 8 paths after decoding the third data bit.

After reaching the third data bit the FSM moves to the metric sorting state. The metric sorting decides which 4 path out of the 8 available paths by sorting the PMs ascendingly and produces an end of sorting pulse to indicate the finishing of its operation to trigger the FSM again.

We then move to the estimate selection state to choose the estimated bit of each decoder since each decoder contains two paths each with a different decoded bit. One of those estimated bits inside each decoder is decided according to the decision unit while the other is its opposite. The FSM informs each decoder to go along with one of its estimated bits according to which one of its 2 paths is surviving.

The FSM keeps the estimate selection state for only one cycle and then moves to the copying logic state where a module with the same name is enabled to decide if there would be any copying from any decoder to another or not. This is done based on the valid flags vectors which is the output of the metric sorter. This valid flags vector is composed of 8 bits each one of them corresponds to one of the paths. The metric sorter puts 1 in the bits corresponding to the survival paths, then the copying logic module takes that vector and checks on the 2 bits of each decoder to decide if that decoder requires copying from/to any other decoder. If there is any copying, all the decoders are disabled and the decoding operation is paused until the 2 decoders finish copying their internal memories. During copying, the FSM has to control the selection lines of the multiplexing network between the decoders depending on the valid flags vector so that each decoder performs copying with the proper selected decoder. When copying is finished, a flag named copying finished is asserted by the decoders involved in the copying operation to indicate that they have finished so that the FSM can move to the next state.

The next state is the PM load state. Before we dive into the operation of this state there are 2 PM registers in each decoder PM_data and PM_data_bar these 2 registers should be equal during the operation until we reach stage 0 then PM_data_bar is penalized as it selects the path opposite to the decoded path based on the LLR of stage 0. During the copying stage, we load PM_data and PM_data_bar with PM_in signal copied from another decoder. However, we need to load both PM_data and PM_data_bar with the same PM value if no copying occurred so that we can continue decoding for the same path with both PMs equal. After finishing PM_load state we return to the normal operation state until we finish decoding the N bits.

After completing the decoding process, we have 4 paths with 4 different PMs we need to choose the path with the least PM so that we can extract the payload from this path to perform the CRC on this payload.

The last state in the operation of SCL is to perform CRC on the extracted payload from the path with the least PM to detect if there is any error in the decoded payload.

8.4 Port mapping.

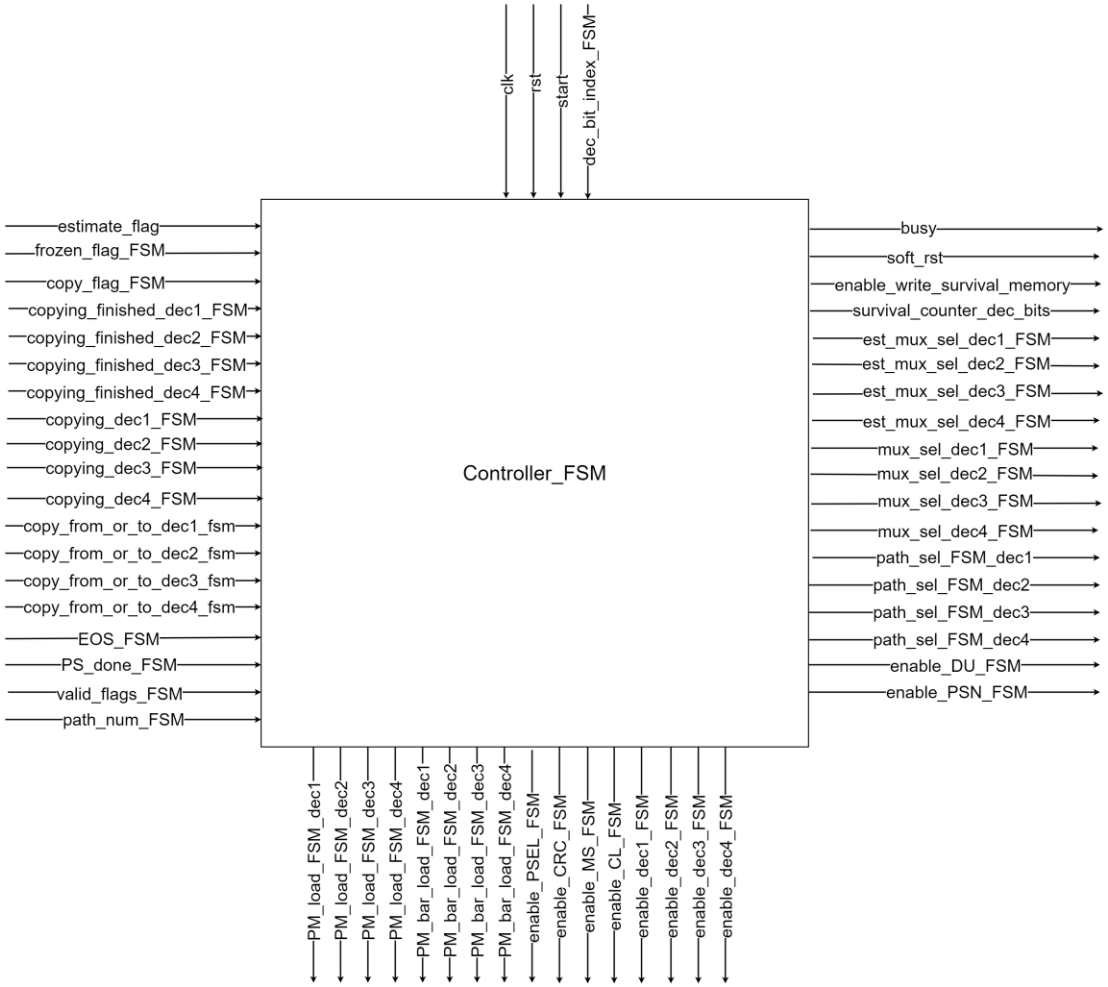


Figure 8.4 Controller FSM port mapping

Table 8.1 Controller FSM port mapping

port	direction	width	parameter
clk	input	1	N/A
rst	input	1	N/A
start	input	1	N/A
estimate_flag	input	1	N/A
frozen_flag_FSM	input	1	N/A
copy_flag_FSM	input	1	N/A
copy_finished_dec1_FSM	input	1	N/A
copy_finished_dec2_FSM	input	1	N/A
copy_finished_dec3_FSM	input	1	N/A
copy_finished_dec4_FSM	input	1	N/A
EOS_FSM	input	1	N/A

copying_dec1_FSM	input	1	N/A
copying_dec2_FSM	input	1	N/A
copying_dec3_FSM	input	1	N/A
copying_dec4_FSM	input	1	N/A
copy_from_or_to_dec1_fsm	input	1	N/A
copy_from_or_to_dec2_fsm	input	1	N/A
copy_from_or_to_dec3_fsm	input	1	N/A
copy_from_or_to_dec4_fsm	input	1	N/A
CRC_Done_FSM	input	1	N/A
ps_done_FSM	input	1	N/A
dec_bit_index_FSM	input	8	$\log_2(N) - 1$
valid_flags_FSM	input	8	N/A
path_num_FSM	input	2	N/A
PM_load_FSM_dec1	output	1	N/A
PM_load_FSM_dec2	output	1	N/A
PM_load_FSM_dec3	output	1	N/A
PM_load_FSM_dec4	output	1	N/A
PM_bar_load_FSM_dec1	output	1	N/A
PM_bar_load_FSM_dec2	output	1	N/A
PM_bar_load_FSM_dec3	output	1	N/A
PM_bar_load_FSM_dec4	output	1	N/A
enable_DU_FSM	output	1	N/A
enable_PSN_FSM	output	1	N/A
enable_dec1_FSM	output	1	N/A
enable_dec2_FSM	output	1	N/A
enable_dec3_FSM	output	1	N/A
enable_dec4_FSM	output	1	N/A
enable_CL_FSM	output	1	N/A
enable_MS_FSM	output	1	N/A
enable_CRC_FSM	output	1	N/A
enable_PSEL_FSM	output	1	N/A
path_sel_FSM_dec1	output	1	N/A
path_sel_FSM_dec2	output	1	N/A
path_sel_FSM_dec3	output	1	N/A
path_sel_FSM_dec4	output	1	N/A
mem_mux_sel_dec1_FSM	output	2	N/A
mem_mux_sel_dec2_FSM	output	2	N/A
mem_mux_sel_dec3_FSM	output	2	N/A
mem_mux_sel_dec4_FSM	output	2	N/A
est_mux_sel_dec1_FSM	output	1	N/A
est_mux_sel_dec2_FSM	output	1	N/A
est_mux_sel_dec3_FSM	output	1	N/A

est_mux_sel_dec4_FSM	output	1	N/A
survival_counter_dec_bits	output	9	N/A
enable_write_survival_memory	output	1	N/A
soft_rst	output	1	N/A
busy	output	1	N/A

8.5 Metric Sorter Sub-Module.

8.5.1 Module description

Metric sorter is one of the major blocks in the list decoder which is responsible for arranging the path metrics ascendingly to choose the least 4 path metrics as the survival paths, there are a lot of sorting algorithms used in literature. We used the bubble sorting algorithm due to its low area which matched our target to reduce the area. Moreover, the latency of the bubble sorting algorithm is lower than both pruned bitonic and the proposed sorter in [16]. However, it has higher latency than radix-2L sorter for L=4.

The bubble sorting algorithm is one of the simplest and hardware efficient algorithms, it consists of comparators to compare between the adjacent PMs which are stored in memory then if the PM stored in the next index in the memory is greater than the current index swapping will occur till we scan the memory for $2L-1$ times then the bubble sorting algorithm is completed.

The metric sorter is activated in the metric sorting state during the list decoding process, the sorting starts when the enable signal is received from the FSM and when the sorting is completed it produces an EOS pulse.

To determine which 4 paths will continue the decoding process and signal them to the SC decoder a comparison is made inside the metric sorter between the $2L$ input path metrics and the L output path metrics then produce an $2L$ bit valid flags vector to index which paths will continue the decoding process. The valid flags register has only L ones at a time.

IMPLEMENTATION RESULTS OF 8-BIT WIDE SORTERS

	Pruned Radix-2L Sorter		Simplified Bubble Sorter		Pruned Bitonic Sorter		Proposed Sorter	
L	EGC†	Latency [ns]	EGC	Latency [ns]	EGC	Latency [ns]	EGC	Latency [ns]
2	138 (1.00)††	0.33 (1.00)	138 (1.00)	0.33 (1.00)	138 (1.00)	0.33 (1.00)	138 (1.00)	0.33 (1.00)
4	892 (1.09)	0.67 (0.51)	735 (0.90)	0.93 (0.70)	894 (1.09)	1.44 (1.09)	819 (1.00)	1.32 (1.00)
8	4593 (1.61)	1.55 (0.57)	3423 (1.20)	2.13 (0.78)	4738 (1.66)	2.84 (1.04)	2848 (1.00)	2.73 (1.00)
16	19399 (2.66)	4.03 (0.95)	13752 (1.88)	4.73 (1.12)	18998 (2.60)	4.43 (1.05)	7298 (1.00)	4.22 (1.00)
32	109887 (3.87)	10.15 (1.66)	59121 (2.08)	9.72 (1.59)	58459 (2.06)	6.53 (1.07)	28363 (1.00)	6.12 (1.00)
64	435539 (10.59)	23.43 (3.09)	202371 (4.92)	20.63 (2.72)	116872 (2.84)	8.24 (1.09)	41111 (1.00)	7.58 (1.00)

†Equivalent gate counts (EGCs) are measured by counting a two-input NAND as one.
††The numbers in parentheses are normalized with respect to the proposed sorter.

Figure 8.5 - Comparison between sorting algorithms [16]

8.5.2 Port mapping.

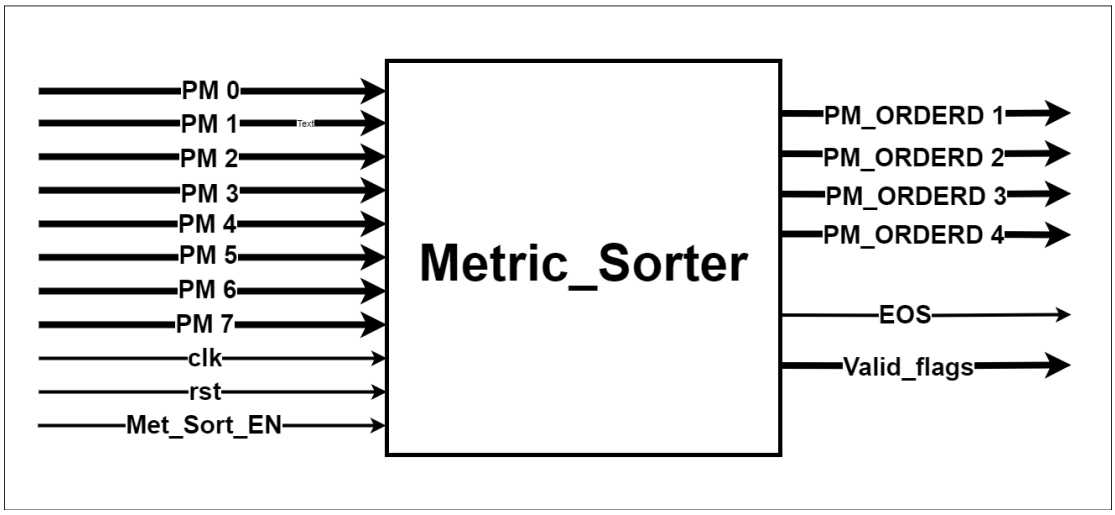


Figure 8.6 - metric sorter port mapping

Table 8.2 metric sorter port mapping

Port	Width	direction	Parameter
clk	1	input	N/A
rst	1	input	N/A
enable_met_sort	1	input	N/A
PM1	10	input	WIDTH_OF_PM
PM2	10	input	WIDTH_OF_PM
PM3	10	input	WIDTH_OF_PM
PM4	10	input	WIDTH_OF_PM
PM5	10	input	WIDTH_OF_PM
PM6	10	input	WIDTH_OF_PM
PM7	10	input	WIDTH_OF_PM
PM8	10	input	WIDTH_OF_PM
PM_ORDERD1	10	output	WIDTH_OF_PM
PM_ORDERD2	10	output	WIDTH_OF_PM
PM_ORDERD3	10	output	WIDTH_OF_PM
PM_ORDERD4	10	output	WIDTH_OF_PM
EOS	1	output	N/A
valid_flags	8	output	L_MUL_2

8.6 Copying Logic Sub-Module.

8.6.1 Module description.

This module is responsible for determining when it is required to copy a path from one decoder to another decoder this is done based on the valid flags that are produced from the metric sorter. This vector contains 2 bits for each decoder when a bit is equal to one this imply that this is a survival path, if there are 2 ones for a decoder then copying is required from this decoder to another idle decoder. On the other hand, if the decoder has 2 zeros, copying is required to this decoder. If no decoder has 2 survival paths there will be no need for copying.

Another function for the copying logic module is to configure the decoder whether it will copy its survival path including all the internal memories, the previously decoded bits and the partial sums to another decoder or from another decoder.

The copying between the 2 decoders will continue until a flag is asserted from the decoders that are involved in the copying process. This flag is named copying_finished based on a counter implemented in a newly modified control unit.

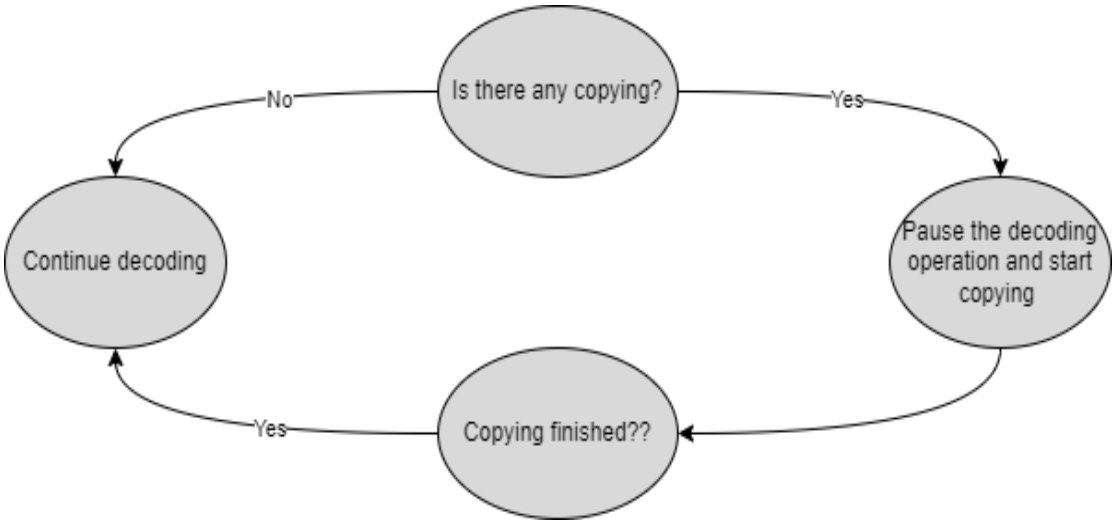


Figure 8.7 - Copying logic operation

8.6.2 Port mapping

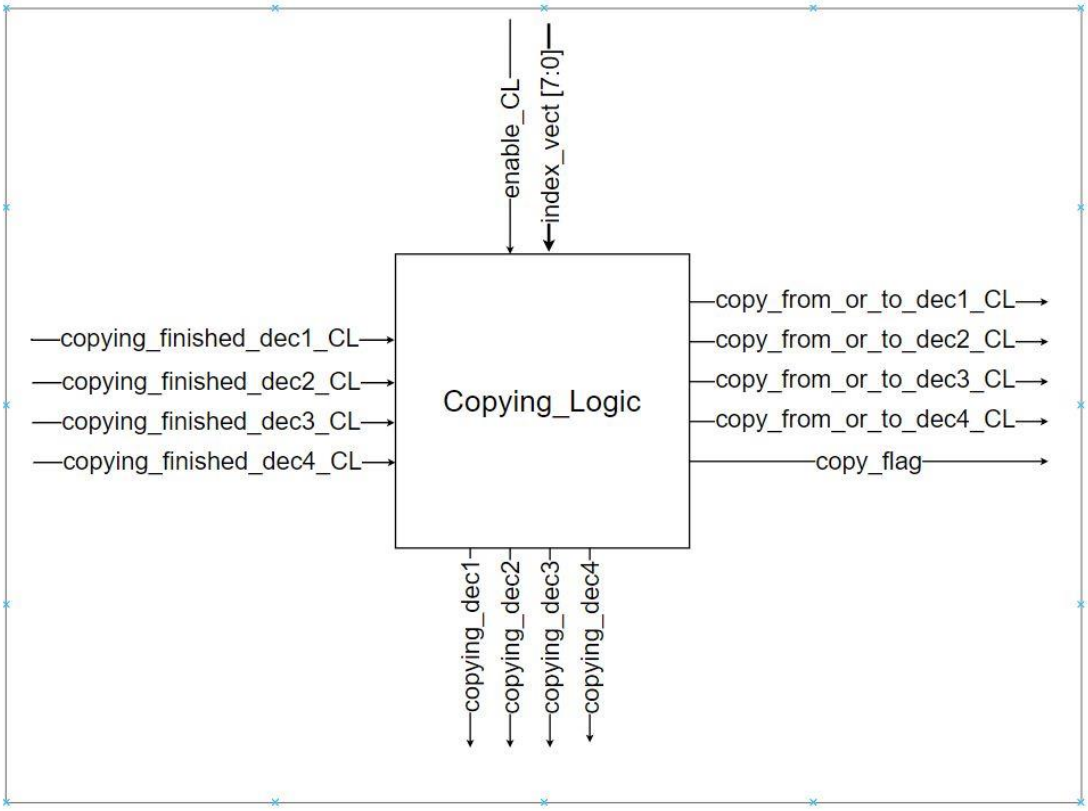


Figure 8.8 - Copying logic port mapping.

Table 8.3 Copying logic port mapping.

Port	Width	direction	Parameter
enable_CL	1	input	N/A
index_vect	8	input	L_MUL_2
copying_finished_dec1_CL	1	input	N/A
copying_finished_dec2_CL	1	input	N/A
copying_finished_dec3_CL	1	input	N/A
copying_finished_dec4_CL	1	input	N/A
copying_dec1	1	output	N/A
copying_dec2	1	output	N/A
copying_dec3	1	output	N/A
copying_dec4	1	output	N/A
copy_flag	1	output	N/A

8.7 Path Selection Sub Module.

8.7.1 Module Description.

As previously explained in the FSM operation we need to select the path with the least path metric so we can extract the pay load and perform CRC check, this path_sel module is responsible for selecting the survival path from the decoding process.

8.7.2 Port Mapping.

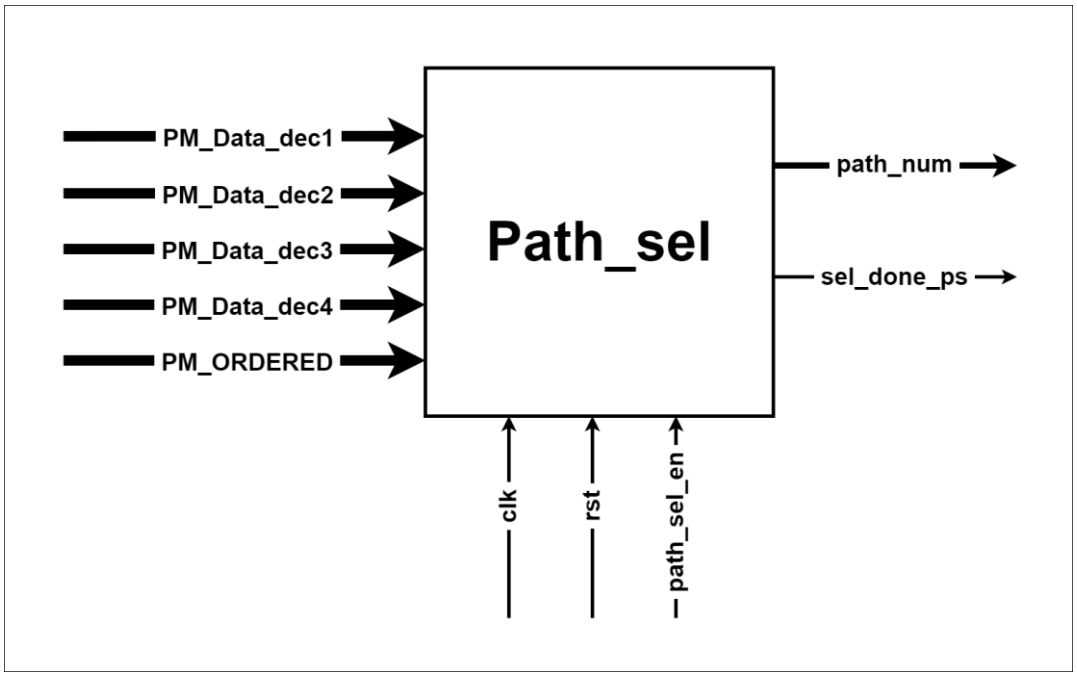


Figure 8.9 - Path_sel port mapping

Table 8.4 - Path_sel port mapping

port	width	direction	parameter
PM_Data_dec1	10	input	PM_WIDTH
PM_Data_dec2	10	input	PM_WIDTH
PM_Data_dec3	10	input	PM_WIDTH
PM_Data_dec4	10	input	PM_WIDTH
PM_ORDERD	10	input	PM_WIDTH
clk	1	input	N/A
rst	1	input	N/A
enable_PSEL	1	input	N/A
sel_done_ps	1	output	N/A
path_num	2	output	N/A

8.8 Remove Frozen Sub-Module.

8.8.1 Module Description.

After decoding the N bits, it will be required to remove the frozen bits from the N decoded bits so that we get the 56 data bits for CRC checking and to be stored in the external memory. Hence, this module was developed for that purpose, to remove the frozen bits and output the 56 data bits.

8.8.2 Port Mapping.

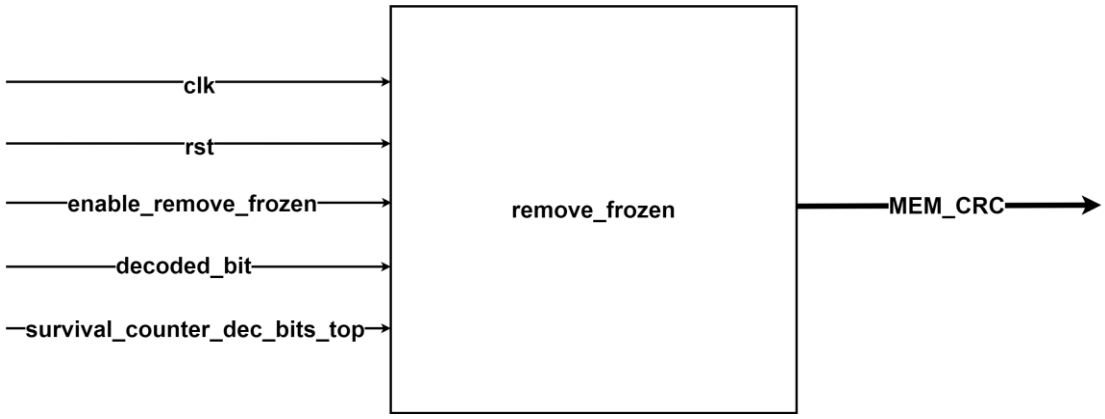


Figure 8.10 - Remove frozen port mapping

Table 8.5 - Remove frozen port mapping

Port	Width	Direction	Parameter
clk	1	input	N/A
rst	1	input	N/A
enable_remove_frozen	1	input	N/A
decoded_bit	1	input	N/A
survival_counter_dec_bits_top	9	input	N/A
MEM_CRC	56	output	N/A

8.9 CRC Check Sub-Module.

8.9.1 Module Description.

The payload can be affected by the channel which may lead to decoding errors. Hence, we need a checker to take a decision on the received payload whether it is received correctly or not. This checker is the CRC check which divides the received payload by a certain number, if the reminder is equal to zero, then the received payload has a high probability to be correctly decoded. If the reminder is not equal to zero, then the received payload is corrupted.

The CRC check is a part of the SCL decoding operation which runs after the operation is done, then it can be considered the last decoding step. This module output two signals to the top controller of the PBCH chain. The first is the CRC done signal which indicates finishing of its operation, the second is the CRC pass to indicate whether the received payload is corrupted or not.

8.9.2 Port Mapping.

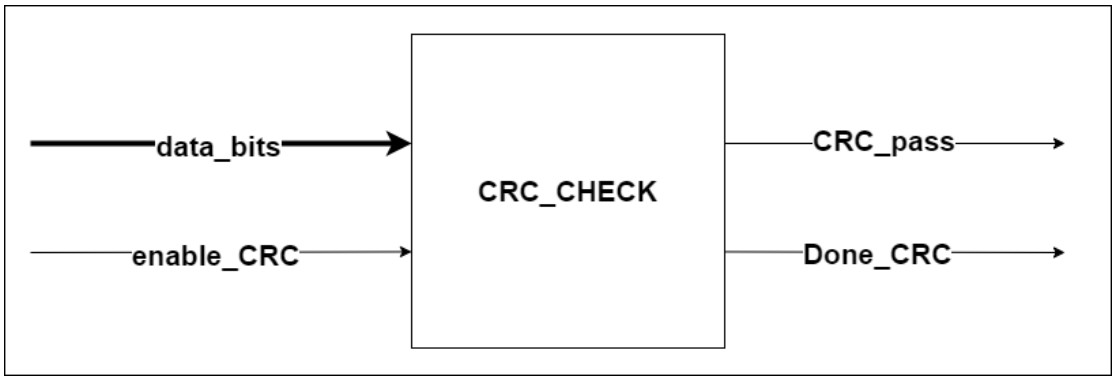


Figure 8.11 - CRC check port mapping

Table 8.6 - CRC check port mapping

Port	Width	direction	Parameter
data_bits	56	Input	N/A
enable_CRC	1	Input	N/A
CRC_pass	1	Output	N/A
Done_CRC	1	Output	N/A

8.10 PM Sorter Sub-Module.

8.10.1 Module Description.

This module is in charge of watching over the calculated PMs of the L SC decoders in all decoding stages. Knowing that an extra bit is added to each PM to detect the overflow that may happen, if any one of the least 4 PMs exceeds the maximum number that can be represented in 8 bits, the PM sorter asserts a flag indicating overflow and quantization and normalization are required.

Unlike the metric sorter, the PM sorter does not require an enable signal to perform its function, it has to be functioning all the time to detect any overflow. It was developed to allow area optimization by asserting the overflow flag which is considered an enable signal to the PM quantizer.

8.10.2 Port Mapping.

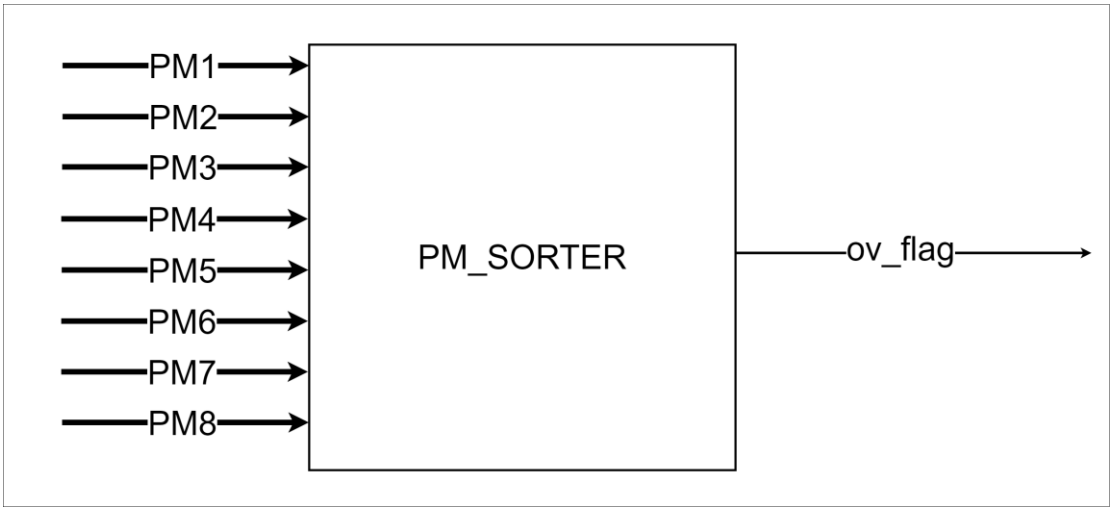


Figure 8.12 - PM sorter port mapping

Table 8.7 - PM sorter port mapping

Port	Width	Direction	Parameter
PM1	10	input	WIDTH_OF_PM
PM2	10	input	WIDTH_OF_PM
PM3	10	input	WIDTH_OF_PM
PM4	10	input	WIDTH_OF_PM
PM5	10	input	WIDTH_OF_PM
PM6	10	input	WIDTH_OF_PM
PM7	10	input	WIDTH_OF_PM
PM8	10	input	WIDTH_OF_PM
ov_flag	1	output	N/A

8.11 PM Quantizer Sub-Module.

8.11.1 Module Description.

When the PM sorter detects an overflow in the least 4 PMs, it asserts a flag. This flag is an input to the PM quantizer acting as its enable. This module subtracts 128 from the PM magnitude and then compares it with the maximum number that can be represented in 8 bits, if it exceeds that number, saturation is done so that the new PM will be equal to 255. This saturation allows area optimization as the PM registers are 9 bits, one sign bit and 8 bits for the magnitude instead of 12 bits in the previous fixed point.

8.11.2 Port Mapping.

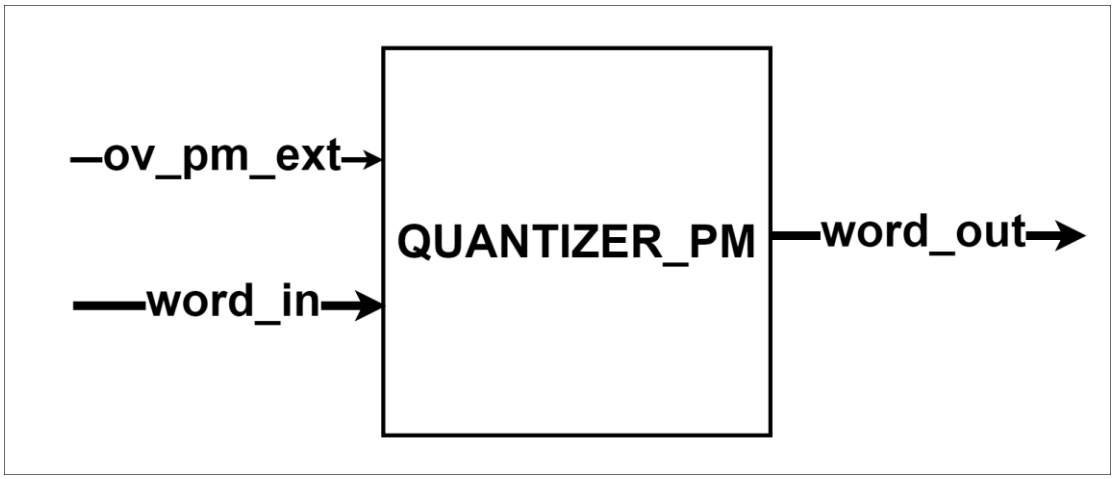


Figure 8.13 - PM quantizer port mapping

Table 8.8 - PM quantizer port mapping

Port	Width	Direction	Parameter
word_in	9 bits	Input	N/A
ov_pm_ext	1	Input	N/A
word_out	9 bits	Output	N/A

8.12 Modified SC Control Unit Sub-Module.

8.12.1 Module Description.

Some modifications were added to the SC control unit to match the new SCL architecture. Some of those modifications were for the copying between the L decoders. When copying is required, each control unit of the 2 decoders involved in the copying has to generate addresses for the internal memories of its decoder in addition to the read or write enable of each memory depending on the copying state whether it is copying from or copying to that decoder.

Some of those modifications were introduced due to the existence of a new control unit for the SCL which controls the decoding flow. The FSM informs the control units of each of the L decoders when to enable its decision unit, it also informs them when to enable their PSN.

8.12.2 Port Mapping.

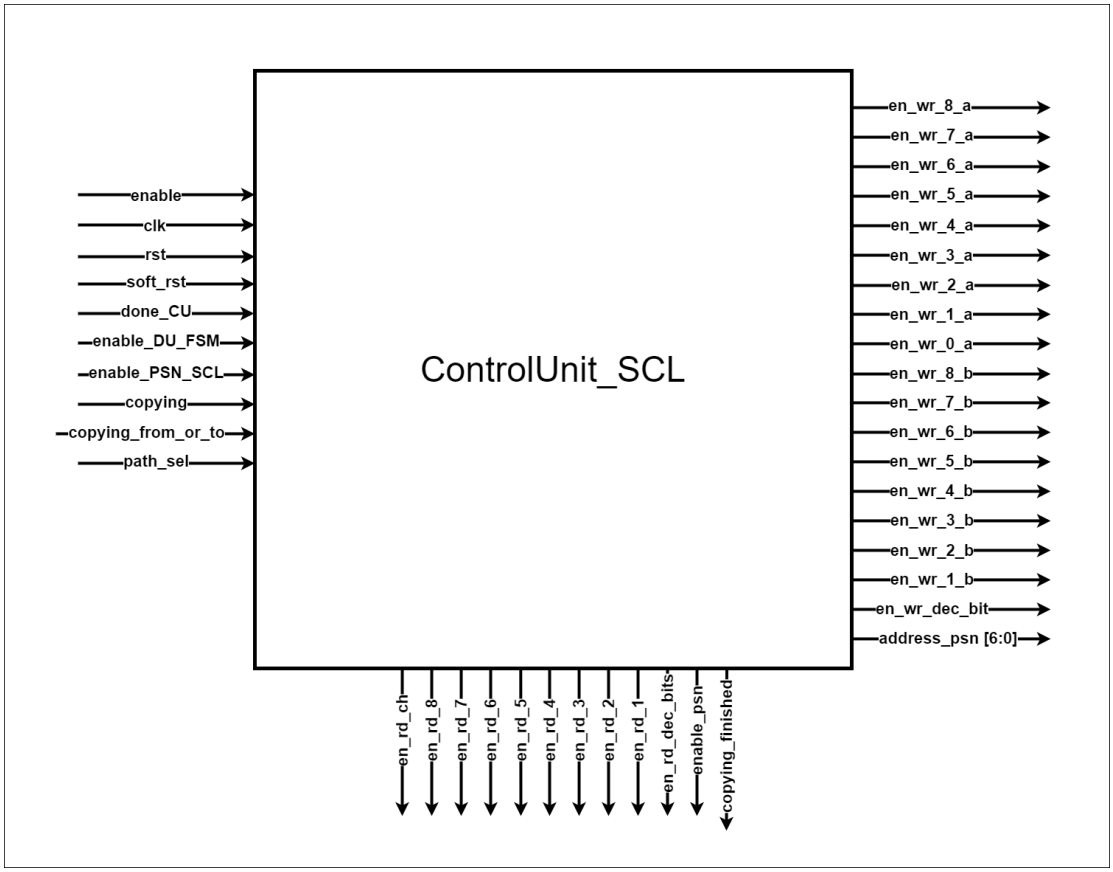


Figure 8.14 - Modified control unit port mapping

Table 8.9 - Modified CU New ports

Port name	Direction	Width	Parameter
enable	1	Input	N/A
done_cu	1	Input	N/A
soft_rst	1	Input	N/A
coping	1	Input	N/A
copy_from_or_to	1	Input	N/A
path_sel	1	Input	N/A
enable_DU_FSM	1	Input	N/A
enable_PSN_SCL	1	Input	N/A
copying_finished	1	Output	N/A

8.13 Modified Decision Unit Sub-Module.

8.13.1 Module Description.

During the SCL architecture design, some modifications had to be made to the SC decision unit. The first and the most important modification is that the decision unit

became in charge of the PM calculations. Each decoder contains two registers for PMs which corresponds to the two paths of that decoder. The first register is the PM data register which corresponds to the path of the estimated bit while the other is the PM data bar which corresponds to the path of the estimated bit bar. The estimated bit is decided according to the input LLR sign while the estimated bit bar is its opposite. While decoding a frozen bit which is a zero bit, if the estimate is equal to one, magnitude of the LLR will be added to both PM data and PM data bar, if it is estimated correctly, their values will stay the same. During estimation of a non-frozen bit, the PM data register remains unchanged while the PM data bar register is penalized as it chooses a wrong path, it is updated by adding the LLR magnitude to its previous value.

The second modification was done in the estimated bit path. Two multiplexers were added to that path, the first to choose between the estimated bit and estimated bit bar depending on the valid flags vector which comes out from the metric sorter to inform that decoder which of its two paths will survive so that the chosen bit is stored in the estimated bits memory. The second multiplexer is used during copying where we choose between the estimated bit bar of the $L-1$ decoders, the selection line of that multiplexer is derived from the FSM depending on the valid flags vector.

8.13.2 Port Mapping.

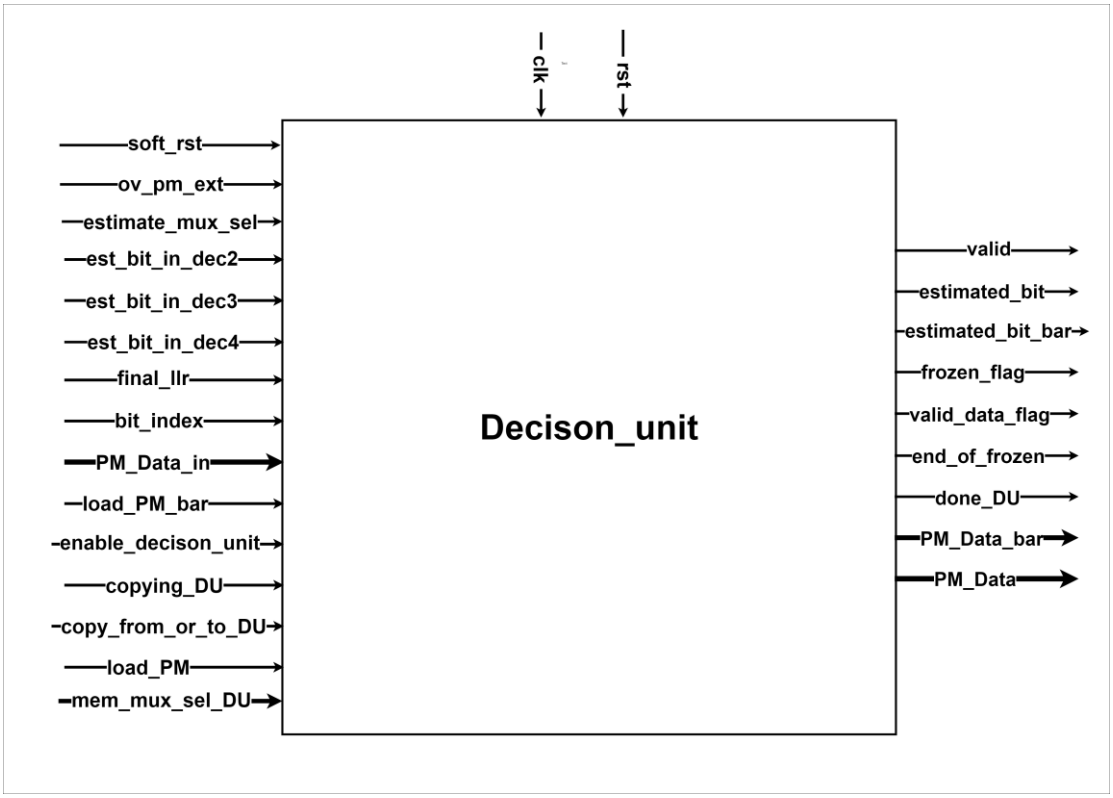


Figure 8.15 - DU New port mapping

Table 8.10 - DU New Ports

Port name	Direction	Width	Parameter
PM_Data_in	input	10	PM_WIDTH
copy_from_or_to_DU	input	1	N/A
copying_DU	input	1	N/A
estimate_mux_sel	input	1	N/A
mem_mux_sel_DU	input	2	N/A
est_bit_in_dec2	input	1	N/A
est_bit_in_dec3	input	1	N/A
est_bit_in_dec4	input	1	N/A
load_PM	input	1	N/A
load_PM_barov_pm_ext	input	1	N/A
soft_rst	input	1	N/A
estimated_bit_bar	output	1	N/A
frozen_flag	output	1	N/A
valid_data_flag	output	1	N/A
end_of_frozen	output	1	N/A
done_DU	output	1	N/A
PM_Data_bar	output	10	PM_WIDTH
PM_Data	output	10	PM_WIDTH

8.14 SCL Block Level Verification

To completely verify the functionality of the SCL decoder we need to verify that the branched paths match the MATLAB model, and the PMs completely match those in the model. In addition to, checking the payload and the CRC check. Therefore, we tested the list decoder with 1000 test cases over all the SNR range. All the test cases are successful as shown in Fig 8.16 and Fig. 8.17.

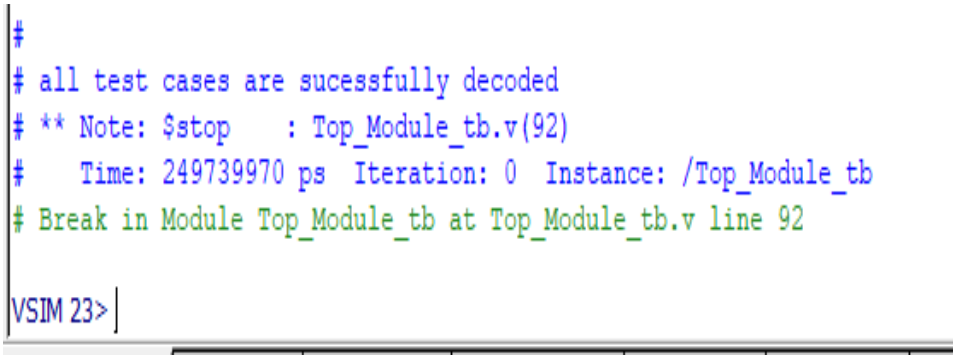


Figure 8.16 - SCL Functional verification

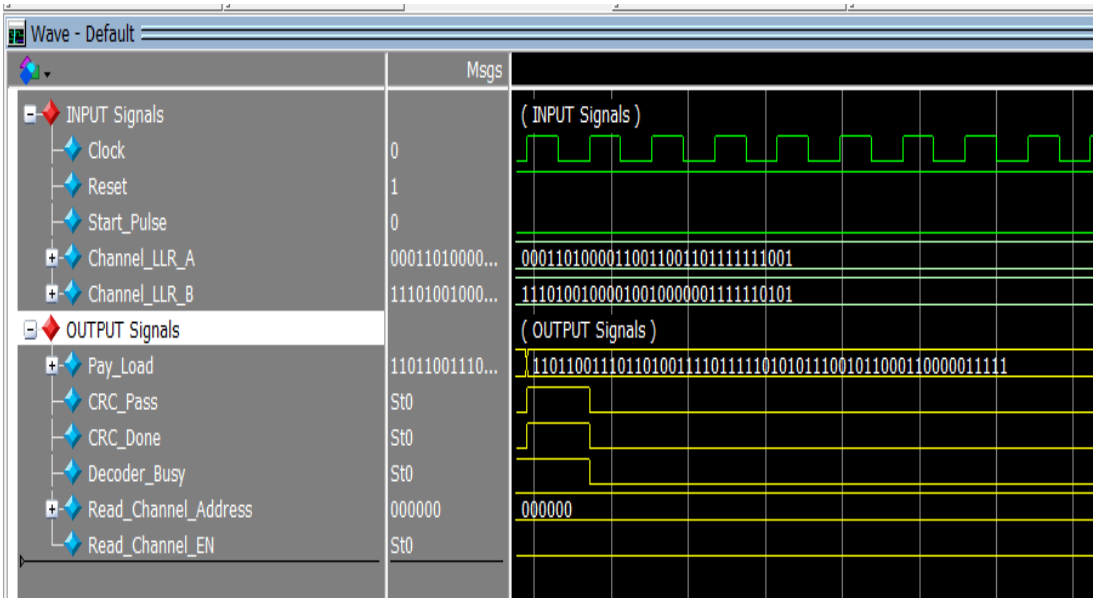


Figure 8.17 SCL Modelsim waveform

Chapter 9: PHY integration

In this chapter, we will discuss the PHY integration, and the blocks used to facilitate its integration with the bus matrix from one side, and the integration of the subsystems from the other side. We will also introduce the algorithm used to find the correct iSSB and report the MIB payload to the processor.

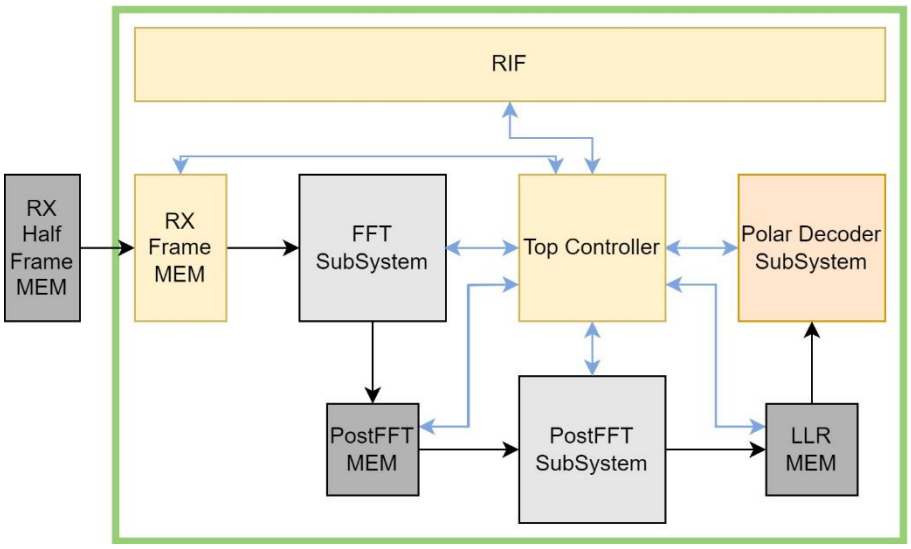


Figure 9.1 - PHY block diagram

9.1 Integrate the DL chain

9.1.1 Register Interface (RIF)

It acts as an AHB interface for the PHY by translating the incoming AHB signals into signals understood by the PHY. It also fills the memories needed by the post-FFT subsystem to operate correctly and provides the top controller with the control signals needed to start or end operations, Fig. 9.1.

The RIF consists of 2 parts:

- AHB interface: responsible of transferring the AHB bus protocol to simple register read write protocol and making sure that the data and address signals arrive at the same time to the PHY.
- Register file: It contains the address map section assigned to PHY based on which the input AHB data is written in a certain register. It also reads some flags from the PHY and transfers them to the processor when requested.

The PHY address map, presented in table (9-1), is translated such that whenever the RIF receives an address, it translates it into a chip select (enable) signal to a certain part/register of the RIF. All the registers in RIF have 32 bits width.

Table 9.1: PHY Memory map

Reg#	Register Name	Start Address	End Address	Description
1	RIF_SW_FLAGS_REG	0x4001_1000	-	Contains the 3 Flags reported to SW and the correct iSSB index Read by AHB, written by PHY
2	RIF_PARAM_REG	0x4001_1004	-	Contains the RIF Parameters Read by PHY, written by AHB
3	RIF_CONTROL_REG	0x4001_1008	-	Contains the control signals required by the top controller Read by PHY, written by AHB
4	MIB_Payload	0x4001_100C	-	Written by PHY, Read by AHB

5	RIF_MMSE_MSB_REG1	0x4001_1010	0x4001_3710	MMSE Coefficient 1
				The 32 MSB are sent to address having third LSB = 0
				Read by PHY, written by AHB
6	RIF_MMSE_MSB_REG2	0x4001_3718	0x4001_5E18	MMSE Coefficient reg 2 (MSB)
				The 32 MSB are sent to addr. having third LSB = 0
				Read by PHY, written by AHB
7	INT_CLR_REG	0x4001_5E1C	-	Clear interrupt Reg
				The LSB contain the clear bit
				Written by PHY, Read by AHB

Each of these registers contains information important to the PHY operation, where:

1. The “RIF_SW_FLAGS_REG” register contains the SW flags reported by the controller so that the processor can know which subsystem has completed its operation, whether it must issue some control signals, and to read the decoded MIB when the decoding process is complete. This register consists of the following flags, table (9-2):
    - Trial done: a single blind decoding trial is done. This signal is needed in SW mode where the processor is responsible of the blind decoding process not the HW controller.
    - All done: all the subsystems have completed their operation. When this flag is issued and the CRC check fails, it means that the decoding chain has fails to decode the MIB correctly.

- CRC Result: indicates whether the CRC check passes or fails.
- FFT done: the FFT subsystem has completed its operation and the processor can start the blind decoding process in SW mode.
- iSSB success: the correct iSSB value reported to the processor after successfully decoding the MIB payload.

Since all the registers are of 32 bits width, the remaining bits are zero padded.

Table 9.2: RIF_SW_FLAGS_REG content

RIF_SW_FLAGS_REG	0's (26 bits)	iSSB_success (2bit)	FFT_done (1 bit)	CRC_result (1bit)	ALL_done (1bit)	Trial_done (1bit)

2. The "RIF_PARAM_REG" register contains the needed parameters for the PHY subsystems to operate properly. These parameters can be updated after each full MIB decoding operation, and they are, table (9-3):
   - Scale: a value used to change the decoder subsystem's input gain in certain cases.
   - N half-frame and cell ID: needed by the post-FFT subsystem to
   - Timestamp: used by the FFT subsystem
   - iSSB: sent by the processor during SW mode only and is updated every blind decoding trial until the correct MIB is found or it equals 3 and the decoding fails.
   - Blind decoding mode: It sets the operating mode of the blind decoding process, when asserted the SW mode is activated.

Table 9.3: RIF_PARAM_REG content

RIF_PARAM_REG	Scale (3 bits)	N_HF (1 bit)	TimeStamp (15 bits)	NCellID (10 bits)	iSSB (2 bits)	Blind_dec_mode (1 bit)

3. The "RIF_CONTROL_REG" register contains the control signals needed by the controller to start operation, and these signals are, table (9-4):
   - Rx Start: Start operation signal sent to controller in both SW and HW modes. When asserted the FFT subsystem starts operating.
   - Rx Stop: Stop read new data signal sent to Rx Frame memory feeding the FFT subsystem in both SW and HW modes. It is enabled after the FFT subsystem finishes operation.
   - Trial Start: indicates the start of a new blind decode operation in SW mode only, when asserted the new iSSB value (from RIF) is used.

Table 9.4 RIF_CONTROL_REG content

RIF_CONTROL_REG	0's (29 bits)	Trial_start_RIF (1bit)	Rx_Stop_RIF (1bit)	Rx_Start_RIF (1bit)

These signals are transformed into pulses by the controller.

4. The “MIB PAYLOAD” register contains the decoded 32 bits of the MIB and is read by the processor upon successful decoding operation.
5. The “RIF_MMSE_MSB_REG1” and “RIF_MMSE_MSB_REG2” hold the 32 most significant bits of the MMSE coefficient memory used by the post-FFT subsystem.
   These memories consist of 1248 elements of 48 bits width; hence the processor sends the memory content using 2 transfers. The first data transfer contains the 32 most significant bits of the data, and it must be followed by the 16 least significant bits using consecutive addresses.
   The 32 most significant bits are sent using addresses whose third least significant bit is equal to zero, then the remaining 16 bits are sent on the following address.
   For example, address 0x4001_1010 is used to write the most significant bits of the memory in an internal register in RIF. Then, the remaining 16 bits are sent on the following address 0x4001_1014 (address [2] = 1).
   If this sequence is not followed, the coefficient values won’t be written correctly.
6. The least significant bit of the “INT_CLR_REG” is asserted by the processor to clear the interrupt signal sent by the PHY (interrupt is served).

9.1.2 RX Frame Memory

The RX Frame Memory is an interface block between the RX Half Frame Memory and the FFT subsystem. It’s functionality is to take a sample from the RX Half Frame Memory each 16 CLK cycle and store this sample and after 16 CLK cycle sends this sample to FFT with a valid pulse and take the next sample from RX Half Memory.

The RX Frame Memory contains a FSM controller which controls the operation of the RX Frame Memory as shown in Fig. 9.2.

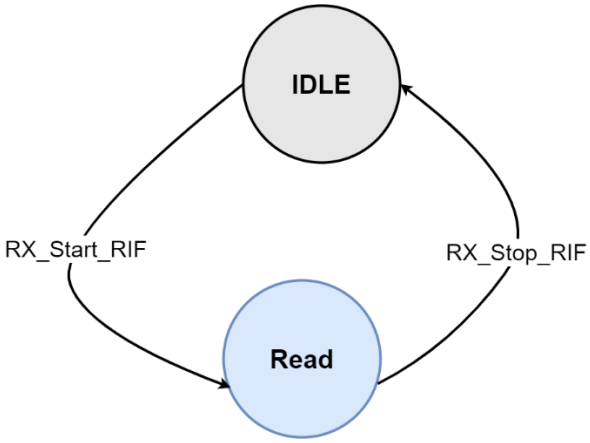


Figure 9.2 - RX Frame Memory FSM

It starts its functionality after receiving a RX start pulse which move it from IDLE state to Read state where in it the RX Frame Memory starts read from the RX Half Frame Memory. When it receives a RX Stop pulse, it moves to IDLE state again and stop its functionality. The functionality of RX Frame Memory is shown in Fig. 9.3.

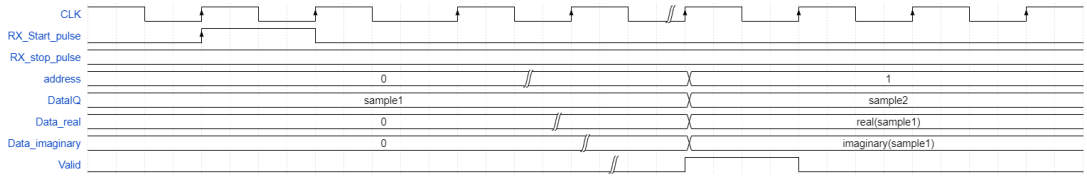


Figure 9.3 - RX Frame Memory Waveform

9.1.2.1 Port Mapping

Table 9.5: Half Frame Memory port mapping

Port	Direction	Width	Description
CLK	Input	1	Clock signal
RST_n	Input	1	Reset signal
RX_Start_pulse	Input	1	Start flag from controller to start operation
RX_Start_pulse	Input	1	Stop flag from controller to stop operation
Data_IQ	Input	24	Input sample from RX Half Frame Memory
Data_real	Output	12	Real part of the sample to FFT
Data_real	Output	12	Imaginary part of the sample to FFT
address	Output	16	Address to Half frame Memory
Valid	Output	1	Valid signal to FFT subsystem to start processing in the given input

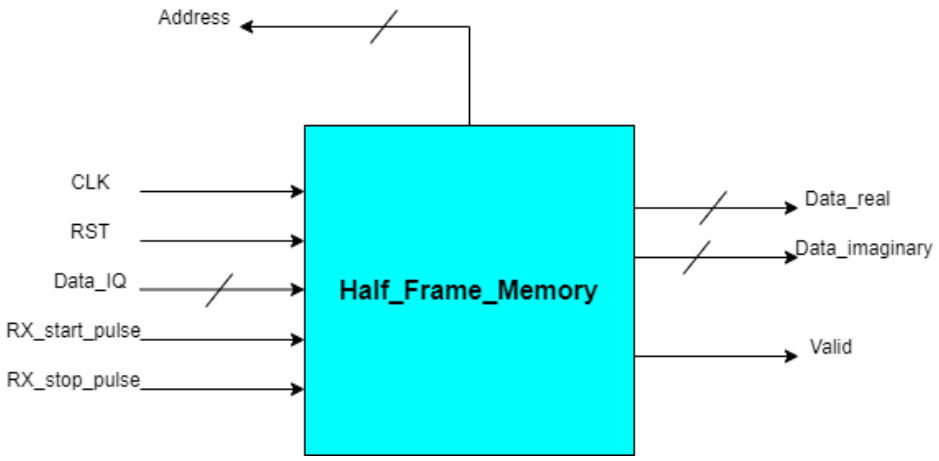


Figure 9.4 - RX Frame Memory block diagram

9.2 The DL chain controller

9.2.1 Module Overview

The Top chain controller Module is responsible for controlling all the subsystems modules throughout its output control signals. The controller takes action according to its input status signals.

The Top controller module is divided into two sub-modules controllers. A sub-controller module to control the operation of the sub-modules and a hardware Blind Decoder controller which will be discussed later. The sub-controller module is responsible for generating an interrupt signal to cortex **INT_PHY = 1** when it receives from **FFT** subsystem a **TTI** pulse equals to 1. The interrupt is cleared by the controller also when the cortex sends a **INT_PHY_CLR** signal equals to 1. The sub-controller module FSM is shown in Fig. 9.5.

Initially, the sub-controller is at **IDLE** state where all its outputs equal to zero. The controller stays in the **IDLE** state until it reads that **RX_Start** signal **in RIF** equals to 1. The controller goes to the next state which is **FFT** state, while transitioning the controller converts the **RX_Start** to a pulse signal and sends it into **RX Frame Memory** and **FFT** subsystem to start operation.

At the **FFT** state, the controller waits the **FFT** subsystem to sends a symbol done pulse three times which means that the **FFT** finishes the processing on all the symbols of the SSB block. When it receives this signal three times, it moves to next state which is **Wait_Trial_Trig.**

In **Wait_Trial_Trig**, the next state depends on decision signal from Hardware Blind Decoder Controller which will be discussed later. If the controller receives a **Start_trial = 1 from** BD controller means that CRC fails and iSSB index less than 3, so next state is **Post-FFT** and the controller here generates a start pulse to the **post-FFT** subsystem **Start_trial_PFFT = 1** to start operation. If it receives **Terminate = 1** means that CRC passes, so next state is IDLE or means that CRC fails and iSSB index more than 3, so

next state is **IDLE**, else it will remain in the same state until the controller receives a decision signal from HW-BD controller.

In Post-FFT state, the controller next state is **Polar Decoder** state and the controller here generates a start pulse to the **Decoder** subsystem **Start_trial_Dec = 1** to start operation when the controller receives **Trial_done_PFFT = 1** from **Post-FFT** subsystem else it will remain in the same state.

In Polar Decoder state, the controller next state **Finish** state when the controller receives **Trial_done_dec = 1** from **Post-FFT** subsystem else it will remain in the same state.

In **Finish** state, the controller wait a single CLK cycle until the CRC reaches the HW BD controller and return to **Wait_Trial_Trig** state and send **Trial_done** = 1 signal to alert the HW BD controller that the chain finishes a single trial.

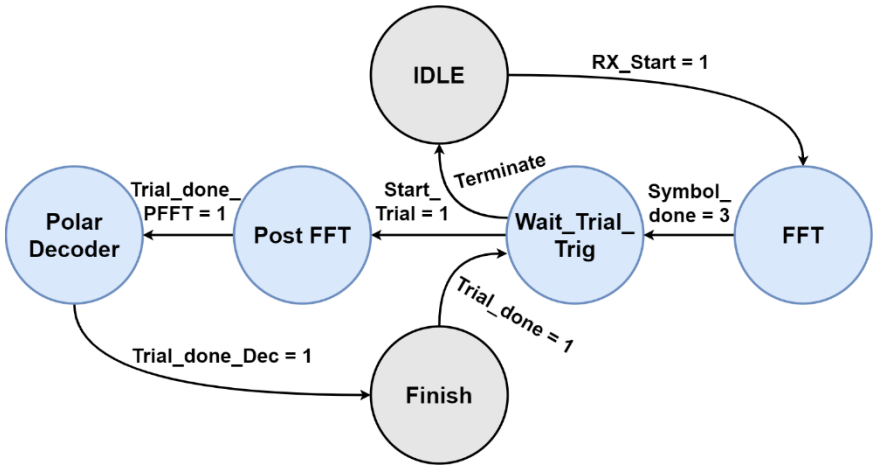


Figure 9.5 - Controller's FSM diagram

9.2.2 Port Mapping

Table 9.6: Top controller memory map

Port	Direction	Width	Description
CLK	Input	1	Clock signal
RST_n	Input	1	Reset signal
RX_Start_RIF	Input	1	Start flag from RIF to controller
Trial_start_RIF	Input	1	Flag from RIF to controller to start a new trial in case of SW BD where SW_Start_trial equals to posedge of this signal
ISSB_RIF	Input	2	iSSB index from RIF register
Blind_Dec_mode	Input	1	To choose type of Blind decoder 0 : HW 1: SW

CRC_result	Input	1	CRC Flag from Polar Decoder block
Symbol_done	Input	2	Flag from FFT must be received three times to ensure that the FFT block finished processing on the 3 symbols
Trial_done_PFFT	Input	1	Flag from the Post-FFT system
Trial_done_Dec	Input	1	Flag from the Post-FFT system
ISSB_index	Output	2	iSSB index value to Post FFT system whether it from BP(HW) or from RIF(SW)
Start_trial_PFFT	Output	1	Flag to Post FFT system to enable its operation
Start_trial_Dec	Output	1	Flag to Decoder system to enable its operation
INT_PHY_CLR	Input	1	Signal from cortex to clear interrupt signal
FFT_done	Output	1	To alert Cortex that FFT subsystem finishes through RIF
Trial_done	Output	1	To alert Cortex that a single trial finished through RIF
INT_cleared	Output	1	To clear the INT_PHY_CLR register value inside the RIF
Start_Trial_Dec	Output	1	Flag to Polar Decoder subsystem to enable its operation
Start_Trial_PFFT	Output	1	Flag to Post FFT subsystem to enable its operation
INT_PHY	Output	1	Interrupt signal to Cortex
Terminate	From BD to sub-controller	1	Flag to return controller to IDLE state due to failure or success
Start_Trial	From BD to sub-controller	1	Signal to indicate a start of new trial in the RX chain, it is output of 2x1 MUX according to type of BD mode

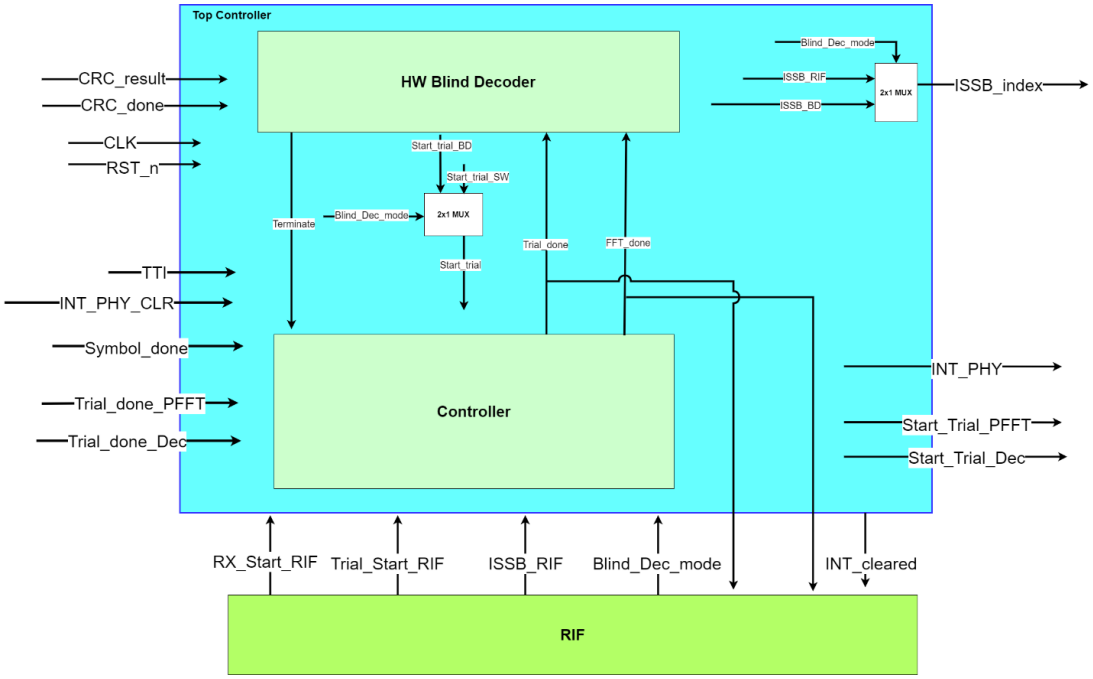


Figure 9.6: Top controller block diagram

9.3 The Blind Decoder procedure

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 evaluates 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 implementation as hardware or software.

9.3.1 Blind Decoding 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 14** [17]. 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 14: Blind decoding algorithm

Inputs: Recovered SSB OFDM grid from SSB signal
Inputs: PBCH and DMRS positions and samples

```
Blind Decoding loop
for iSSB ∈ [0, Lmax -1] then
    - Compute corresponding DMRS sequence
    - Perform channel estimation and equalization
    - Compute LLR values
    - Implement de-rate matching
    - Decode the input LLRs
    If CRC is valid then
        - Report the decoded MIB and the correct iSSB
        Break
    end
end
```

9.3.2 HW implementation

The Hardware Blind Decoder implementation is implemented as a FSM controller as shown in Figure (9-7) in the Top controller module. It is responsible for generating an **iSSB index** each time it receives from sub-controller **Trial_done** signal equals 1. It is also responsible for monitoring the CRC_result from Polar Decoder subsystem to decide this trial is a success one or a failure trial.

The Hardware Blind Decoder is initially at an **IDLE** state, where all its outputs equal to zero. When the HW-BD controller receives a **RX_start** and **Blind_Dec_mode = 0** (HW), it moves to **Wait_FFT** state.

At the **Wait_FFT** state, the HW-BD controller remains in this state until it receives from sub-controller that **FFT-done = 1** and moves to **Process_trial** state

At **Process_Trial** state, the HW-BD controller starts the **ISSB index = 0** and moves to **Check_Trial** state if **Trial_done = 1** from sub-controller. At this state the **Start_Trial** signal equals to 1.

At **Check_Trial** state, the HW-BD controller checks the value of **CRC_result** and the **ISSB index** and the decides its next state. If **CRC_result** = 0 and **iSSB index** less than 3, so next state is **Process_trial**. If **CRC_result** = 0 and **ISSB_index** more than 3, so next state is **Fail**. If **CRC_result = 1**, so next state is **Success**.

At **Fail** state, the next state is IDLE and the HW-BD controller asserts the **Terminate** signals equals to 1.

At **Success** state, the next state is IDLE and the HW-BD controller asserts the **Terminate** signals equals to 1.

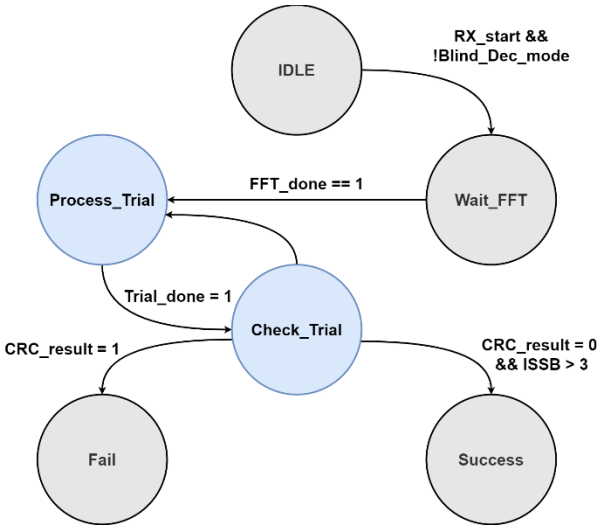


Figure 9.7:Hardware Blind Decoder FSM

9.3.3 SW implementation

The Software implementation of Blind Decoder is discussed at chapter 2 at System Core section 2.2.

Chapter 10: FPGA

10.1 Synthesis.

Before diving in the synthesis flow results, efficient RTL simulation was done to get the correct output which will be used as a reference through the FPGA flow results.

Since, the post synthesis simulation (GLS) and post implementation are exhaustive, this comparison between the results was done on a few test cases. In this Chapter we will use only one test case for comparing between them.

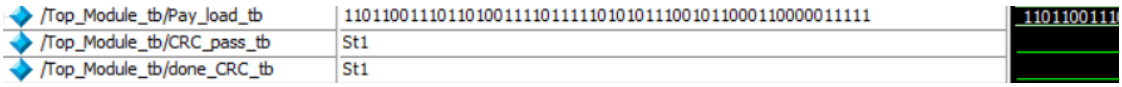


Figure 10.1 - RTL simulation

ZYNQ ULTRASCALE+ is our target FPGA, the clock frequency constraint is 61.44 MHZ.

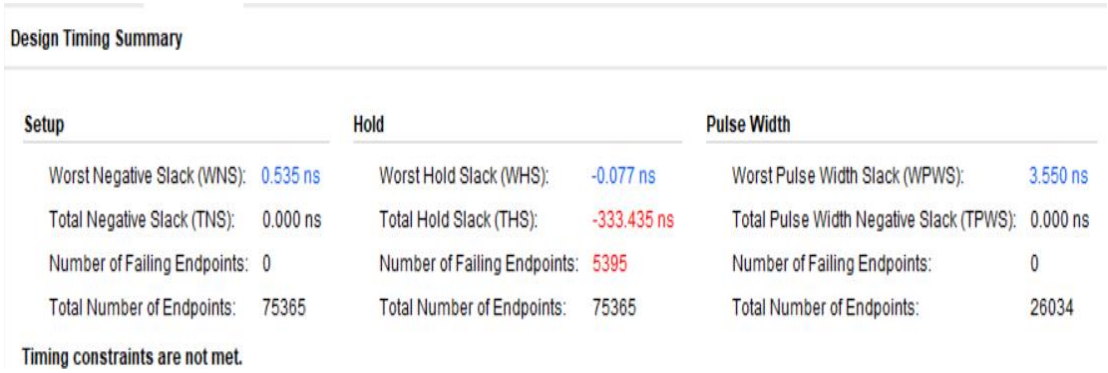


Figure 10.2 - Synthesis timing analysis.

As it can be seen in Figure 10.2, there was no setup time violation but there was hold time violations in some paths which will be fixed after implementation but the tool by adding buffers in the violating paths.

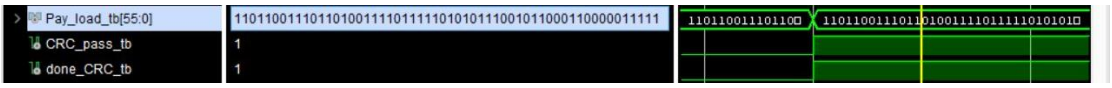


Figure 10.3 - Post synthesis simulation

As shown in Figure 10.3 and Figure 10.1, The post synthesis simulation matchs the RTL simulation which indicates a proper translation of the synthesizable RTL to a functioning gate level netlist.

10.2 Implementation.

Implementation is a step in which the gate level netlist is placed and routed on the FPGA fabric, any hold time violation can be fixed in this step as the timing between any 2 flipflops can be determined thus, the software knows how much delay is required to add buffers to fix the hold time violation.

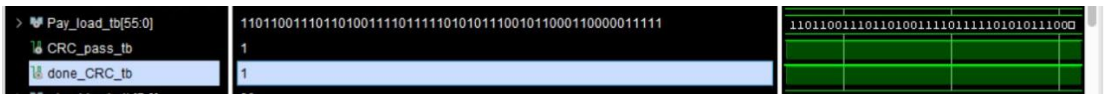


Figure 10.4 - Post implementation simulation.

As shown in Figure 10.1 and Figure 10.5, the payload of the post implementation simulation match that of the RTL simulation which indicates the correctness of the implementation.

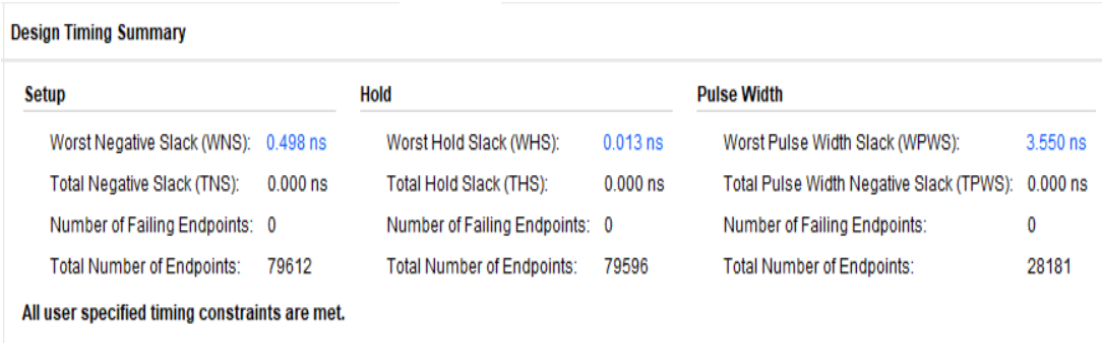


Figure 10.5 - Implementation timing analysis.

As it can be seen in the previous figure, hold time violations are fixed with a slack of 0.013 ns compared to the synthesis result in Figure 10.2.

10.3 Design Wrapper.

To prepare the design for bit stream generation we need to identify the ports of the design and synchronize them with the pins of the FPGA board, so we need a reset synchronizer and a bit synchronizer for the start pulse of the decoder.

The clock source of the ULTRASCALE+ board produces a differential clock of frequency 125 MHZ so we used the clock wizard IP to produce a single ended clock with frequency 61.44 MHZ.

We also used the memory generator block as the interface memories to the SCL decoder and they were loaded with the same test case used before to verify the functionality of the wrapper.

The ILA block is used for hardware debugging after we program the FPGA with the generated bit stream to be able to analyze the outputs.

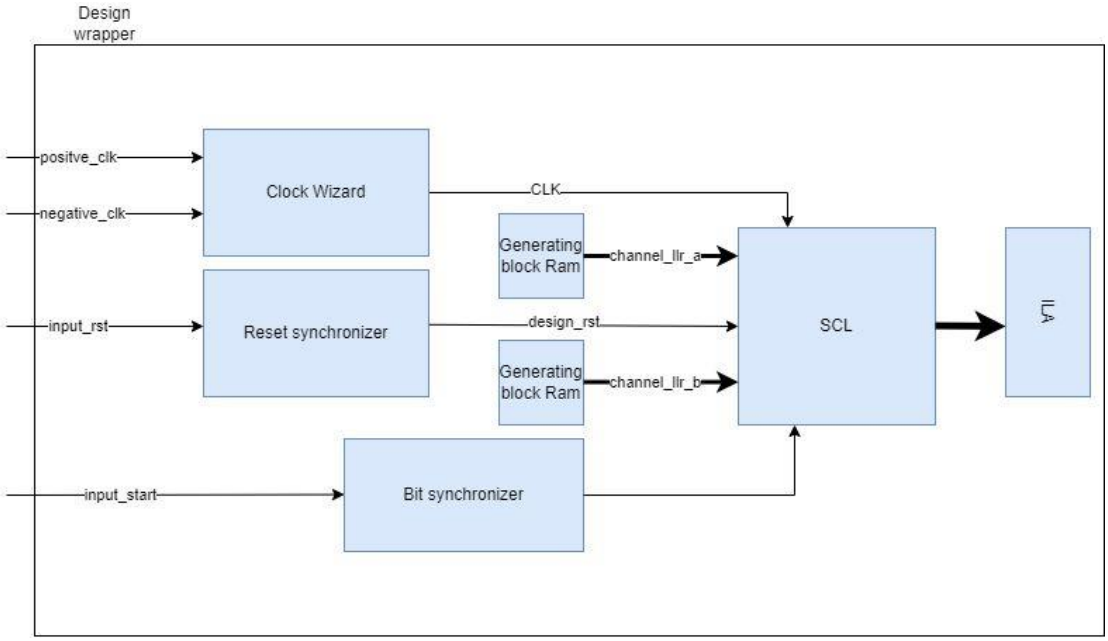


Figure 10.6 - Design block

Resource	Utilization	Available	Utilization %
LUT	21396	230400	9.29
LUTRAM	176	101760	0.17
FF	27133	460800	5.89
BRAM	28	312	8.97
IO	4	360	1.11
BUFG	1	544	0.18
MMCM	1	8	12.50

Figure 10.7 - FPGA utilization

Chapter 11: Conclusion

As a conclusion, our polar decoder hardware implementation was verified against the MATLAB model and meets all the hardware constraints (Clock frequency and target FPGA resources) with optimal area without severely affecting the latency. Our decoder was implemented on ZYNQ ULTRASCALE+ and succeeded in decoding a few test cases.

The downlink chain was integrated as slave on the APB with many IPs such as the UART IP, Timer IP and the WDT IP. The APB was connected to the AHB through a bridge. It receives its commands from the Cortex-M0 through the RIF. The PBCH decoding processor succeeded in decoding the MIB payload at the proper iSSB index and it saves the results in internal registers inside the RIF.

Chapter 12: Future Work

The decoder was tested through block level verification using some test benches and generated test cases from the MATLAB model. Hence as a future work it can tested using UVM test benches which will be more effective.

Another task to be done is to synthesize the integrated downlink chain and implement it on FPGA. In addition, it can be also tested using UVM test benches.

References

[1] Ericsson. [Online]. Available: https://www.ericsson.com/49f1c9/assets/local/5g/documents/07052021-ericsson-this-is-5g.pdf. [Accessed 10 jun 2023].

[2] Telcoma. [Online]. Available: https://telcomaglobal.com/p/mib-master-information-block-in-5g. [Accessed 10 jun 2023].

[3] E. Arikan.

[4] A. Balatsoukas-Stimming, "IEEE information theory society," 2020. [Online]. Available: https://www.itsoc.org/video/efficient-decoding-polar-codes-algorithms-and-implementations. [Accessed Dec 2022].

[5] N.-N. IITM, "youtube," 2019. [Online]. Available: https://youtu.be/9b2z6bua0xY . [Accessed feb 2023].

[6] N.-N. IITM, "youtube," 2019. [Online]. Available: https://youtu.be/1uYEq4ueOok . [Accessed feb 2023].

[7] N.-N. IITM, 2019. [Online]. Available: https://youtu.be/rB0rhQKyV34 . [Accessed feb 2023].

[8] 3GPP, "etsi," july 2018. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/138200_138299/138212/15.02.00_60/ts_138212v150200p.pdf. [Accessed feb 2023].

[9] N.-N. IITM, "youtube," [Online]. Available: https://www.youtube.com/watch?v=O3JWkvEY8Lc&list=PLyqSpQzTE6M81HJ26ZaNv0V3ROBrcv-Kc&index=33&ab_channel=NPTEL-NOCIITM. [Accessed jan 2023].

[10] A. C. C. L. &. C. J. Mathieu Léonardon, "Fast and Flexible Software Polar List Decoders".

[11] Alexios Balatsoukas-Stimming, Alexandre J. Raymond, Warren J. Gross, and Andreas Burg, "Hardware Architecture for List Successive Cancellation Decoding of Polar Codes," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 61, pp. 609-613, 2014.

[12] 3GPP, "etsi," may 2019. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/138100_138199/13810101/15.05.00_60/ts_138101v150500p.pdf. [Accessed jan 2023].

[13] Camille Leroux, Alexandre J. Raymond, Gabi Sarkis, Ido Tal, Alexander Vardy and Warren J. Gross, "Hardware architectures for successive cancellation decoding of polar codes," *2011 IEEE International Conference on Acoustics, Speech and Signal Processing,* pp. 1665-1668, 2011.

[14] Camille Leroux, Alexandre J. Raymond, Gabi Sarkis, and Warren J. Gross, "A Semi-Parallel Successive-Cancellation Decoder for Polar Codes," *"IEEE Transactions on Signal Processing",* vol. 61, pp. 289-299, 15 Jan 2013.

[15] Guillaume Berhault, Camille Leroux, Christophe Jego, Dominique Dallet, "Partial sums generation architecture for successive cancellation decoding of polar codes," *SiPS 2013 Proceedings,* pp. 407-412, 2013.

[16] B. a. Y. H. a. P. I.-C. Yong Kong, "Efficient Sorting Architecture for Successive-Cancellation-List Decoding of Polar Codes," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 63, pp. 673-677, 2016.

[17] A. d. Javel, "5G RAN : physical layer implementation and network".

[18] N.-N. IITM, 2019. [Online]. Available: https://www.youtube.com/watch?v=10cnv-vik90&list=PLyqSpQzTE6M81HJ26ZaNv0V3ROBrcv-Kc&index=36. [Accessed Nov 2022].

[19] N.-N. IITM, 2019. [Online]. Available: https://www.youtube.com/watch?v=WbC5Ux5Pjp8&list=PLyqSpQzTE6M81HJ26ZaNv0V3ROBrcv-Kc&index=37. [Accessed Jan 2023].

[20] Z. B. a. X. L. a. M. R. G. a. H. L. Kaykac Egilmez, "The Development, Operation and Performance of the 5G Polar Codes," *IEEE Communications Surveys & Tutorials,* 2020.