WSI PROJECT

Reducing Complexity of CD-NOMA Decoding Algorithm Using Hardware Optimizations

Prateek Kumar 2021181

Sagar Keim 2019196

Abstract

One of the major challenges of the fifth generation (5G) of mobile networks is the increasing number of connected users and devices compared to 4G. The target connection density requirements have reached **a million devices per square kilometer**. This is difficult to satisfy with orthogonal multiple access. Code-domain non-orthogonal multiple access (NOMA) schemes are an evolved variant of code division multiple access (CDMA). CD-NOMA can provide the users access to a set of finite system resources, simultaneously and efficiently. The challenge with CD-NOMA techniques is their high time complexity and energy requirements. We propose to reduce this complexity by performing hardware optimizations to the decoding algorithm and running it on an FPGA. Our work will be to profile the algorithm to find the bottlenecks and then offload the bottleneck task on the FPGA to improve the overall performance of the algorithm.

Introduction

The next generation cellular systems primarily focus on providing extremely high data rates of above 100 Gbps with an end-to-end delay of less than 1ms while supporting e-health, smart cities, self-driving cars, etc. This explosive growth of communication applications leads to congested spectrum which is the major challenge to satisfy the target connection density requirements.

Multiple access is a core wireless communication technique that enables multiple users to share limited resources effectively. Traditional systems use orthogonal multiple access (OMA) schemes like TDMA, FDMA, CDMA, and OFDMA, where users are assigned distinct, non-overlapping resources. **However, OMA limits the number of simultaneously served users to the available orthogonal resources.**

Recently, non-orthogonal multiple access (NOMA) has gained attention for offering massive connectivity, reduced latency, and higher spectral efficiency by superimposing users on shared resources. Sparse code multiple access (SCMA), introduced in 2013, is a code-domain NOMA scheme combining QAM mapping and sparse sequence spreading, enabling efficient resource usage and improved performance.

Literature review

On Fixed-Point Implementation of Log-MPA for SCMA Signals (Liu et al., 2016)

This paper addresses the fixed-point implementation of the log-domain message passing algorithm (Log-MPA) for sparse code multiple access (SCMA) signals. It compares the complexity and performance of Log-MPA with traditional MPA and demonstrates its feasibility using FPGA implementations.

Performance Characterization of an SCMA Decoder (Alizadeh et al., 2024)

Study examines the error performance and hardware complexity of an SCMA decoder based on the message passing algorithm (MPA). The focus is on reducing latency and complexity through parallelization techniques and the use of high-level synthesis tools like VIVADO HLS.

Efficient Hardware Architecture of Deterministic MPA Decoder for SCMA (Yang et al., 2016)

This paper proposes a deterministic message-passing algorithm (DMPA) for SCMA and designs a stage-level folded hardware architecture to enhance speed and hardware efficiency.

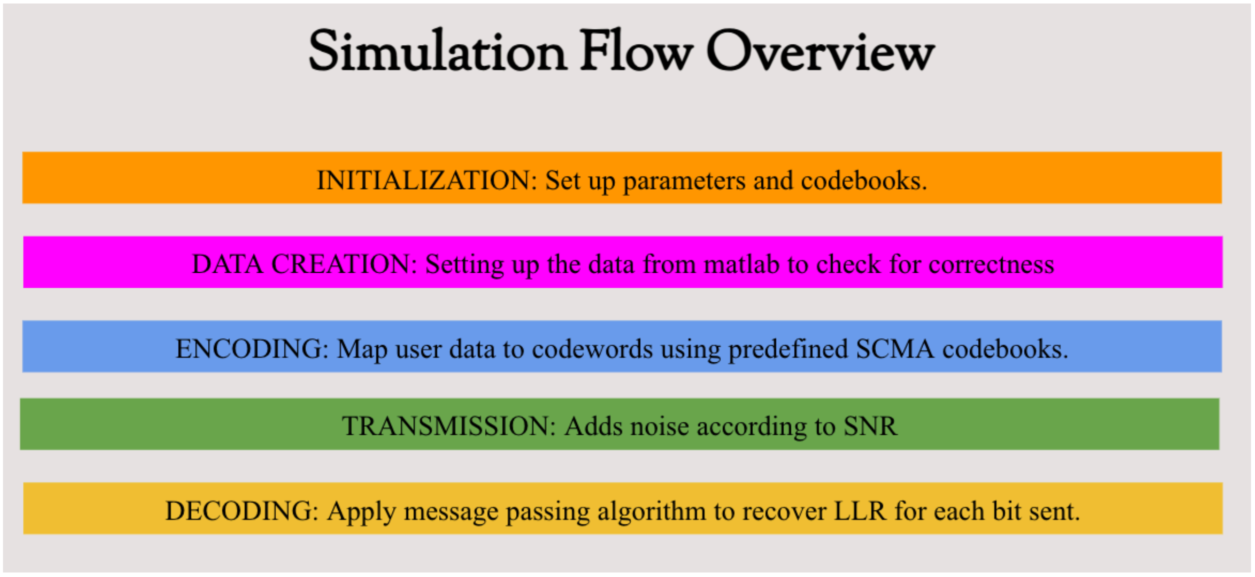
Toward High-Performance Implementation of 5G SCMA Algorithms (Ghaffari et al., 2019)

This research explores various techniques to improve SCMA decoding, focusing on message-passing algorithms (MPA) such as Log-MPA. It evaluates the effects of SIMD extensions and parallel computing strategies on improving decoder performance.

Our work

- Understood SCMA system model including encoding and decoding using Message Passing Algorithm.
- Went through various academic articles to understand the previous work done and find benchmarks for our project.
- Coded and optimized the SCMA simulation in C and Hardware, and ran the simulations on following specifications :-

# User's = **6**
# Codeword = **4**
# Channel Resource Elements = **4**
# RE shared by each user = **2**
# Users overlapped per RE = **3**
# Frames = **1** for multiple SNR values.


Simulation Flow Overview
INITIALIZATION: Set up parameters and codebooks.
DATA CREATION: Setting up the data from matlab to check for correctness
ENCODING: Map user data to codewords using predefined SCMA codebooks.
TRANSMISSION: Adds noise according to SNR
DECODING: Apply message passing algorithm to recover LLR for each bit sent.

<u>Objective : Is to compute the below estimate using SPA (Sum Product Algorithm)</u>

$$\hat{\mathbf{m}}_j = \underset{\mathbf{m}_j \in \mathbb{A}_j}{\operatorname{argmax}} \sum_{\sim \mathbf{m}_j} \left(P(\mathbf{X}) \prod_{k=1}^{K} f(y_k|\mathbf{X}) \right) \quad \text{for } j = 1, \cdots, J.$$

Step 1 : To calculate the LLR (Log Likelihood Ratio) of each FN (Function Node = Resource Element). The log of the below expression shows the LLR at FN (l).

$$f(y_l|\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, N_0) = \exp\left(\frac{-1}{N_0} ||y_l - (h_{l1} C_{l,1}(\mathbf{m}_1) + h_{l2} C_{l,2}(\mathbf{m}_2) + h_{l3} C_{l,3}(\mathbf{m}_3))||^2 \right)$$
$$\text{for } \mathbf{m}_1 \in \mathbb{A}_1, \mathbf{m}_2 \in \mathbb{A}_2, \mathbf{m}_3 \in \mathbb{A}_3.$$

Step 2 : Passing message between each node in Factor Graph. The process is repeated until there’s no change found in the belief at each VN (Variable Node).

Step 3 : To compute the bit LLR of b_i^{th} bit at v VN (Variable Node)

$$LLR_{b_i}^{v} = \log \frac{P(b_i = 0)}{P(b_i = 1)}, \\ = \log \frac{\sum_{\mathbf{m}_v | b_i = 0} I_v(\mathbf{m}_v)}{\sum_{\mathbf{m}_v | b_i = 1} I_v(\mathbf{m}_v)}.$$

If LLR < 0 implies ‘1’ is decoded otherwise ‘0’ is decoded

C Code Design

```
for # Frames
    for # k(Resource Element)
        Likelihood Function(k) = y(k)[encoded message] -
.                         CB(Codebook) x h(Fading Coefficient)
    for # Iterations
        for $m_1$ for $m_2$ for $m_3$    Update Igv(m1)
        for $m_2$ for $m_1$ for $m_3$    Update Igv(m2)
        for $m_3$ for $m_1$ for $m_2$    Update Igv(m3) {Igv = Function to Variable Node Matrix}
        Update Ivg($k_1$)
        Update Ivg($k_2$) {Ivg = Variable to Function Node Matrix}
        Compute LLR of 6 Users x 2 bits each
```

Hardware Code Design

The hardware design for this project focuses on accelerating the Sparse Code Multiple Access (SCMA) decoder using the Logarithmic Message Passing Algorithm (Log-MPA). The goal of this design is to optimize the computational efficiency and energy consumption of SCMA decoding, which is a core component in next-generation wireless communication systems. The implementation is realized using High-Level Synthesis (HLS) tools provided by Vivado HLS, leveraging hardware parallelism to achieve real-time decoding performance.

The hardware implementation of the SCMA decoder is structured into modular blocks, each targeting specific computational stages of the Log-MPA algorithm. These include matrix operations, logarithmic and exponential computations, and message-passing routines. The design workflow involves the following stages:

1. **Codebook Representation**: The SCMA system utilizes predefined codebooks for encoding and decoding. Real and imaginary components of the codebooks are stored as

constant arrays to ensure efficient access during computations. These values are represented in double-precision floating-point format for accuracy.

2. **Input/Output Interface**: The design employs AXI-Stream interfaces to handle input and output data streams. Input data includes received signals, channel coefficients, and noise power, while the output consists of Log-Likelihood Ratios (LLRs) that determine the most likely transmitted symbols.

    - Input Data Handling: A dedicated read_stream function extracts the input signals and reconstructs complex numbers from real and imaginary components. This includes:
        - **Received signals (y)**: Complex received symbols over orthogonal resources.
        - Channel Coefficients (h): Channel gain for each resource-user pair.
        - Noise Power (N0): Required for SNR-based calculations.
    - Output Data Handling: A write_stream function formats the computed LLRs into AXI-Stream compliant packets and transmits them sequentially.

3. **Core Computational Blocks**: The Log-MPA algorithm's key computations were divided into reusable hardware functions optimized for performance and resource utilization:
    - **Log-Sum-Exp Function**: A numerically stable implementation of the log-sum-exp operation ensures efficient summation of probabilities during decoding.
    - Complex Logarithmic and Exponential Functions: Functions for computing the logarithm and exponential of complex numbers are implemented using the hls::math library, optimized for hardware synthesis.
    - **Message Passing**: Message computation and updates leverage parallel processing, taking advantage of the sparsity of the SCMA graph structure. This reduces unnecessary computations.

4. Parallelism and Pipelining: Hardware-level parallelism and pipelining were extensively utilized to achieve low latency. The critical computational paths in the message-passing loop were identified and pipelined using Vivado HLS directives (#pragma HLS). Unrolling and loop flattening were applied selectively to balance throughput and resource utilization.

Implementation and Optimization

The SCMA decoder design was implemented using a top-down approach in Vivado HLS. The following optimizations were applied to improve performance:

- **Memory Optimization**:
    - The codebooks (CB_real and CB_imag) were stored as read-only constant arrays in on-chip memory to minimize memory access latency.
    - Intermediate values, such as channel-wise message contributions, were stored in local variables to reduce off-chip memory transactions.
- **Arithmetic Operations**:
    - Hardware-optimized versions of logarithm, exponential, and trigonometric functions (hls::log, hls::exp, hls::atan2, etc.) were used to ensure numerical accuracy with minimal area overhead.
- **Efficient Data Access**:
    - Multidimensional arrays were designed to minimize data dependency issues. Parallel computation of different users' codewords was facilitated by separating memory regions.

SCMA Decoder Flow

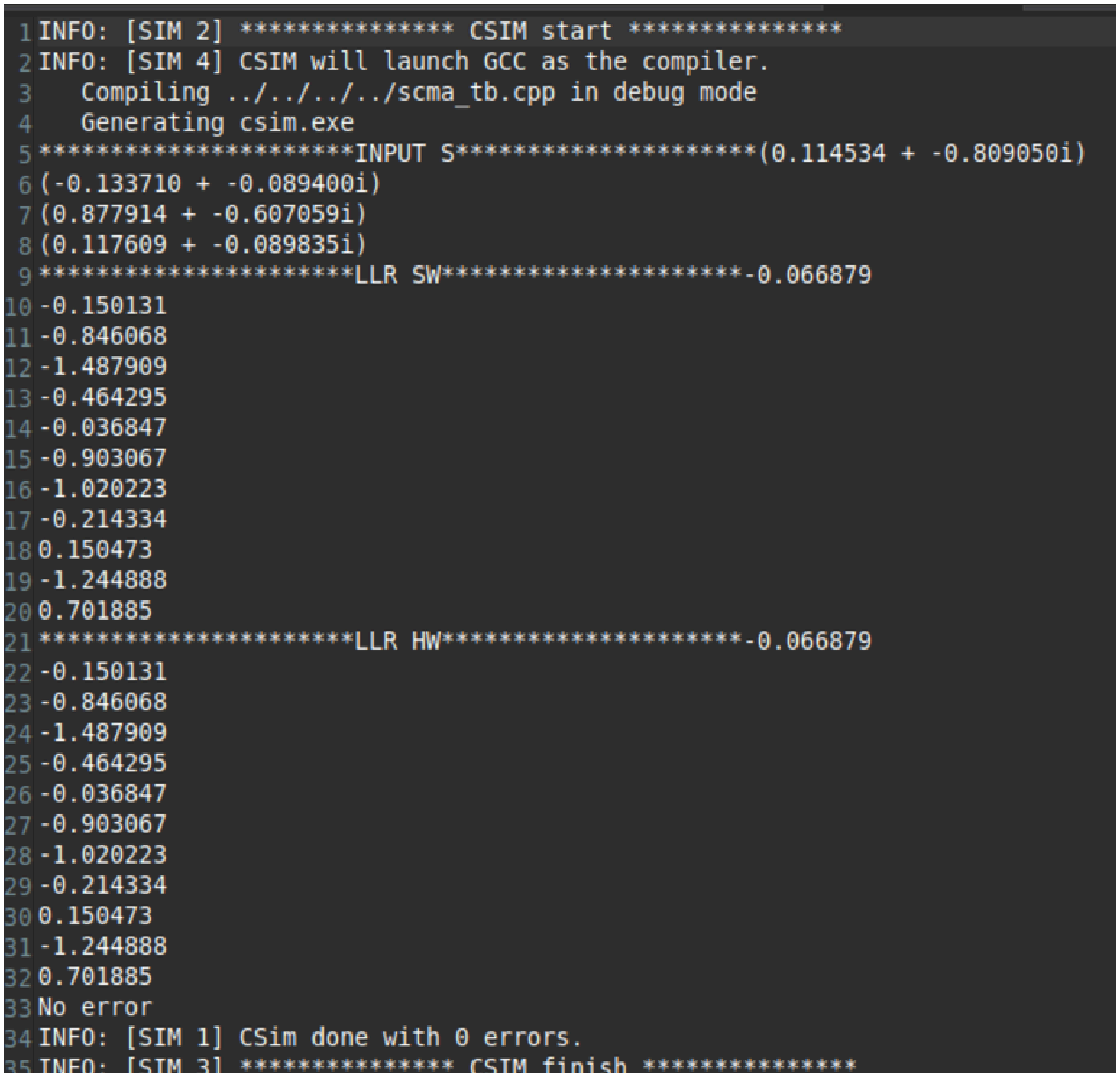
1. **Input Parsing**:
    - The received signals, channel coefficients, and noise power are read from the input stream and formatted into matrices and complex arrays.
2. **Message Initialization**:
    - Initial messages between layers and resources are computed using predefined priors.
3. **Iterative Log-MPA Decoding**:
    - The Log-MPA decoding process iteratively computes and updates messages between layers and resources, utilizing the SCMA graph's sparsity to reduce computational complexity.
    - The core computation involves calculating the likelihood of each codeword and propagating these probabilities through the factor graph.
4. **LLR Calculation**:
    - After the final iteration, the LLR values are computed for each transmitted bit. These values are packed into AXI-Stream packets for transmission.

Hardware Validation and Testing

The functionality of the hardware design was validated through co-simulation in Vivado HLS. Testbench data included randomly generated SCMA signals and channel coefficients, verified against a software-based reference implementation. The design was further synthesized for FPGA implementation, and performance metrics such as latency, and resource utilization were measured.

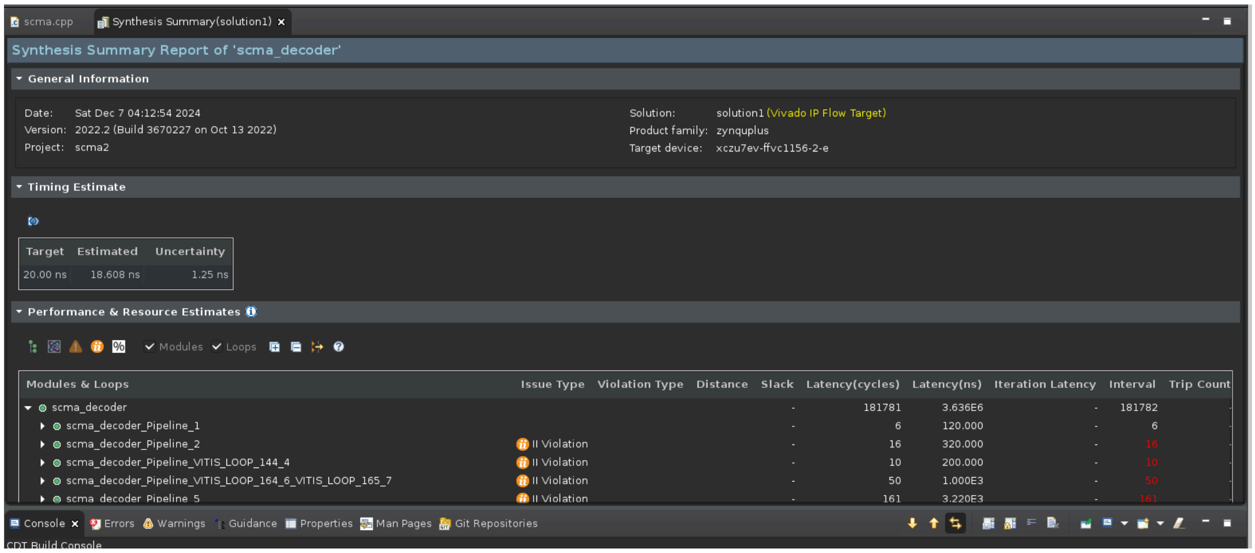
Vitis HLS Results:

C Simulation result:



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../scma_tb.cpp in debug mode
   Generating csim.exe
**********************INPUT S*********************(0.114534 + -0.809050i)
(-0.133710 + -0.089400i)
(0.877914 + -0.607059i)
(0.117609 + -0.089835i)
**********************LLR SW********************-0.066879
-0.150131
-0.846068
-1.487909
-0.464295
-0.036847
-0.903067
-1.020223
-0.214334
0.150473
-1.244888
0.701885
**********************LLR HW********************-0.066879
-0.150131
-0.846068
-1.487909
-0.464295
-0.036847
-0.903067
-1.020223
-0.214334
0.150473
-1.244888
0.701885
No error
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Synthesis result:



Synthesis Summary Report of 'scma_decoder'

General Information

Date: Sat Dec 7 04:12:54 2024
Version: 2022.2 (Build 3670227 on Oct 13 2022)
Project: scma2
Solution: solution1 (Vivado IP Flow Target)
Product family: zynquplus
Target device: xczu7ev-ffvc1156-2-e

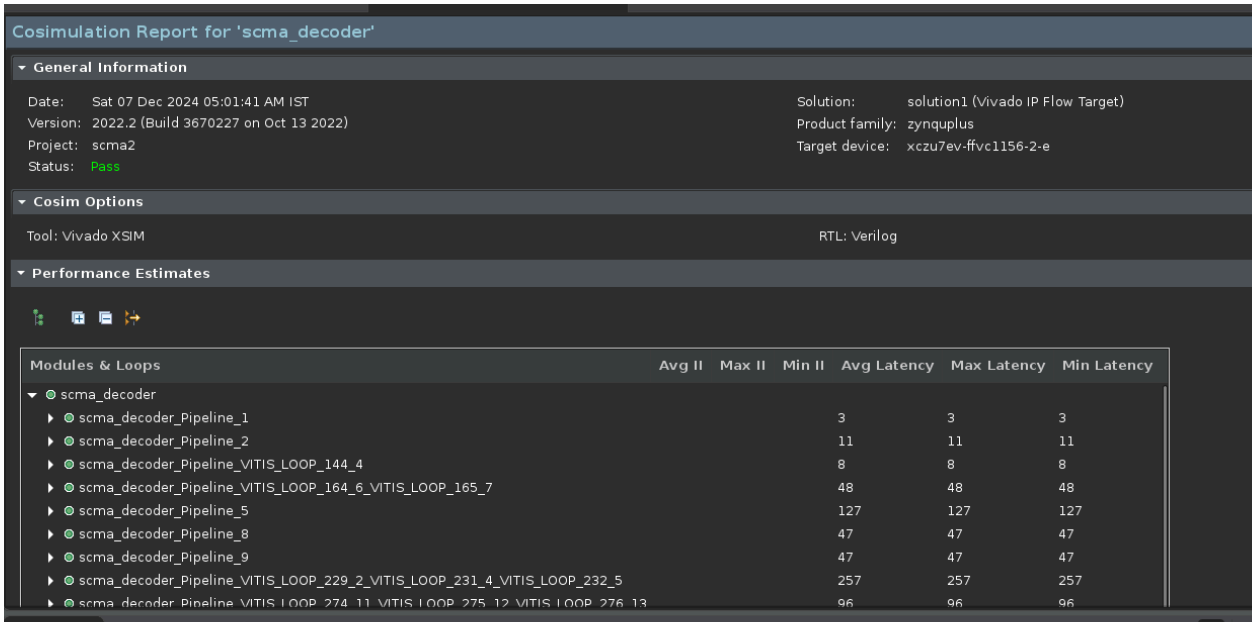
Timing Estimate

Target	Estimated	Uncertainty
20.00 ns	18.608 ns	1.25 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count
scma_decoder				-	181781	3.636E6	-	181782	-
scma_decoder_Pipeline_1				-	6	120.000	-	6	-
scma_decoder_Pipeline_2	II Violation			-	16	320.000	-	16	-
scma_decoder_Pipeline_VITIS_LOOP_144_4	II Violation			-	10	200.000	-	10	-
scma_decoder_Pipeline_VITIS_LOOP_164_6_VITIS_LOOP_165_7	II Violation			-	50	1.000E3	-	50	-
scma_decoder_Pipeline_5	II Violation			-	161	3.220E3	-	161	-

C RTL Cosimulation result:



Cosimulation Report for 'scma_decoder'

General Information

Date: Sat 07 Dec 2024 05:01:41 AM IST
Version: 2022.2 (Build 3670227 on Oct 13 2022)
Project: scma2
Status: Pass
Solution: solution1 (Vivado IP Flow Target)
Product family: zynquplus
Target device: xczu7ev-ffvc1156-2-e

Cosim Options

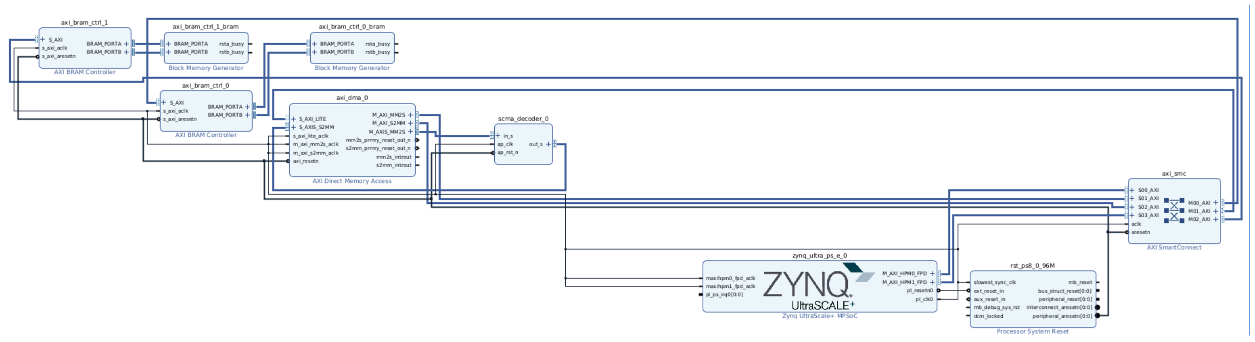
Tool: Vivado XSIM
RTL: Verilog

Performance Estimates

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
scma_decoder						
scma_decoder_Pipeline_1				3	3	3
scma_decoder_Pipeline_2				11	11	11
scma_decoder_Pipeline_VITIS_LOOP_144_4				8	8	8
scma_decoder_Pipeline_VITIS_LOOP_164_6_VITIS_LOOP_165_7				48	48	48
scma_decoder_Pipeline_5				127	127	127
scma_decoder_Pipeline_8				47	47	47
scma_decoder_Pipeline_9				47	47	47
scma_decoder_Pipeline_VITIS_LOOP_229_2_VITIS_LOOP_231_4_VITIS_LOOP_232_5				257	257	257
scma_decoder_Pipeline_VITIS_LOOP_274_11_VITIS_LOOP_275_12_VITIS_LOOP_276_13				96	96	96

Next the IP is exported and then is sent to Vivado to generate the block design and then finally generate the bitstream for implementing on the hardware.

Vivado Block design:

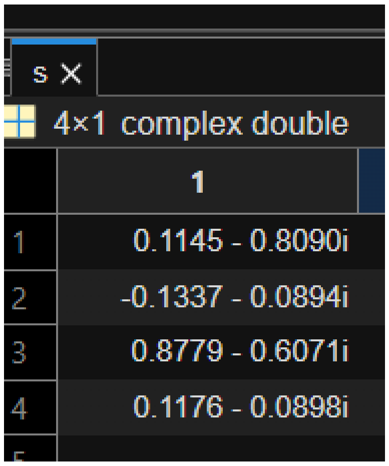


Hardware execution has also been done for the PS implementation for which an SDK has been created for the given block design and then the results have been observed on the ZCU 106 board. The results have been added in the results section below.

Results

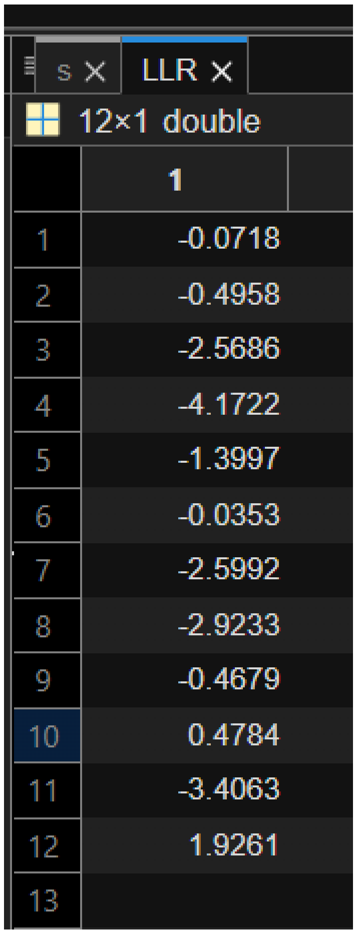
Matlab Simulation

For the encoded signal


s ×

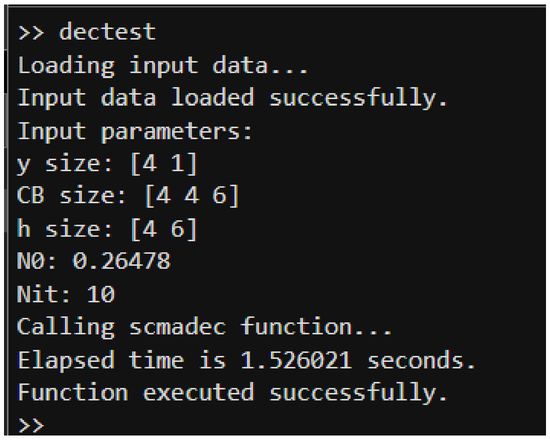
4×1 complex double

	1
1	0.1145 - 0.8090i
2	-0.1337 - 0.0894i
3	0.8779 - 0.6071i
4	0.1176 - 0.0898i



	1
1	-0.0718
2	-0.4958
3	-2.5686
4	-4.1722
5	-1.3997
6	-0.0353
7	-2.5992
8	-2.9233
9	-0.4679
10	0.4784
11	-3.4063
12	1.9261
13	

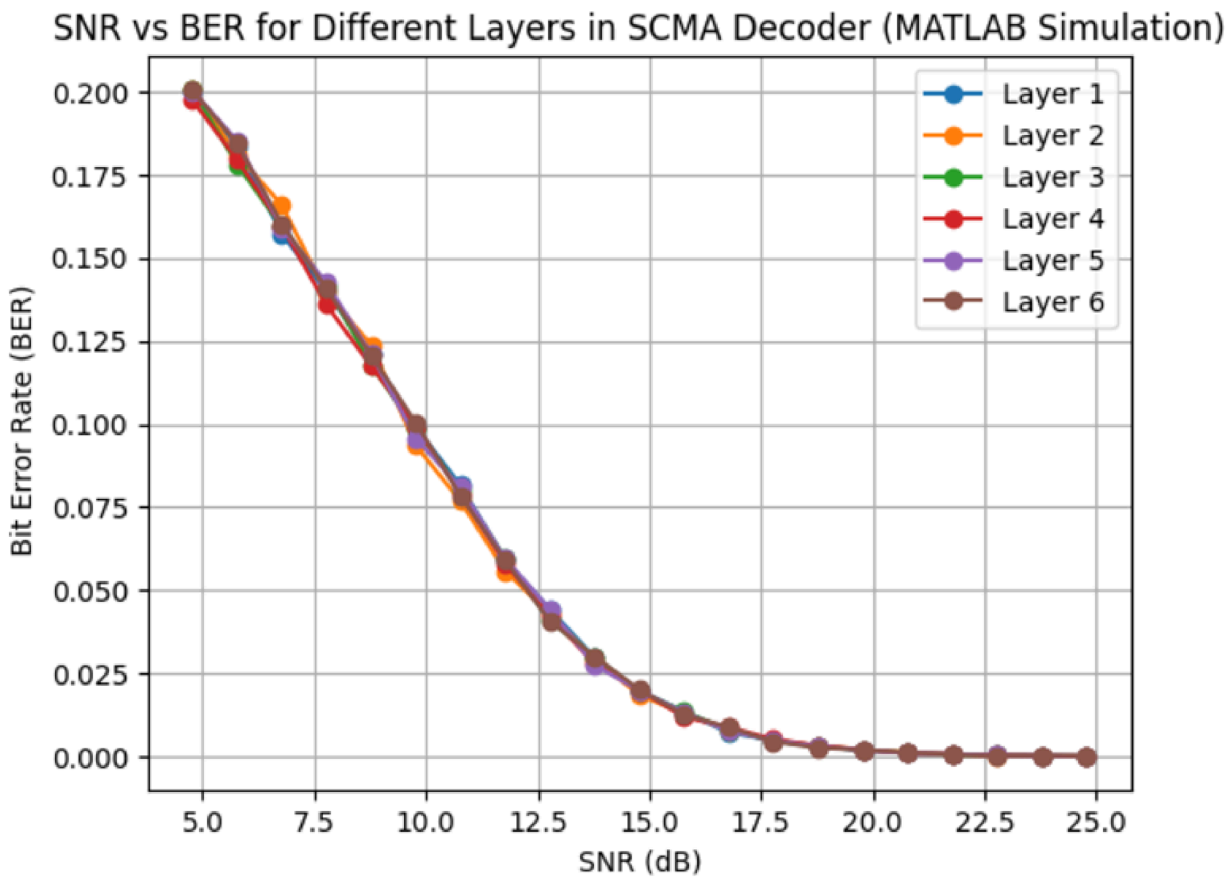
We are getting the LLRs as



```
>> dectest
Loading input data...
Input data loaded successfully.
Input parameters:
y size: [4 1]
CB size: [4 4 6]
h size: [4 6]
N0: 0.26478
Nit: 10
Calling scmadec function...
Elapsed time is 1.526021 seconds.
Function executed successfully.
>>
```

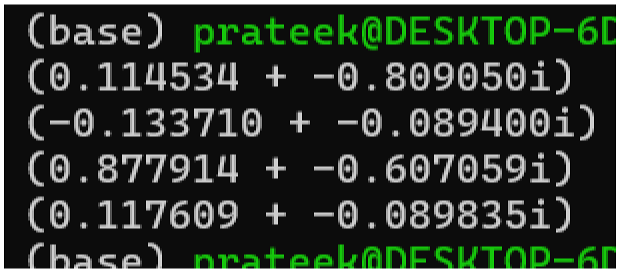
This is the Matlab execution log showing the time delay. We are getting 1526 ms latency from the Matlab implementation.

Matlab implementation results for SNR ranging from 4.771213 to 24.771213 for various users. Here layers mean various users.



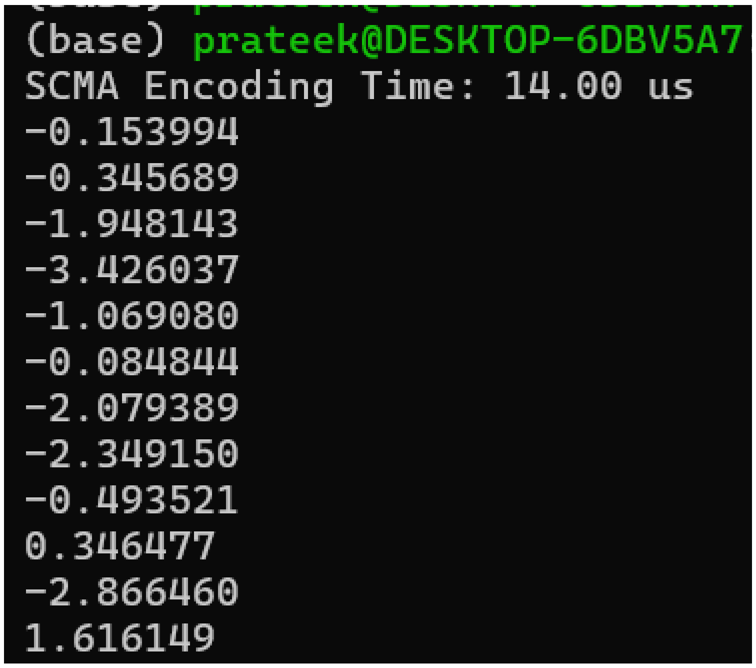
C Implementation

For the same encoded signals as those in the matlab simulation



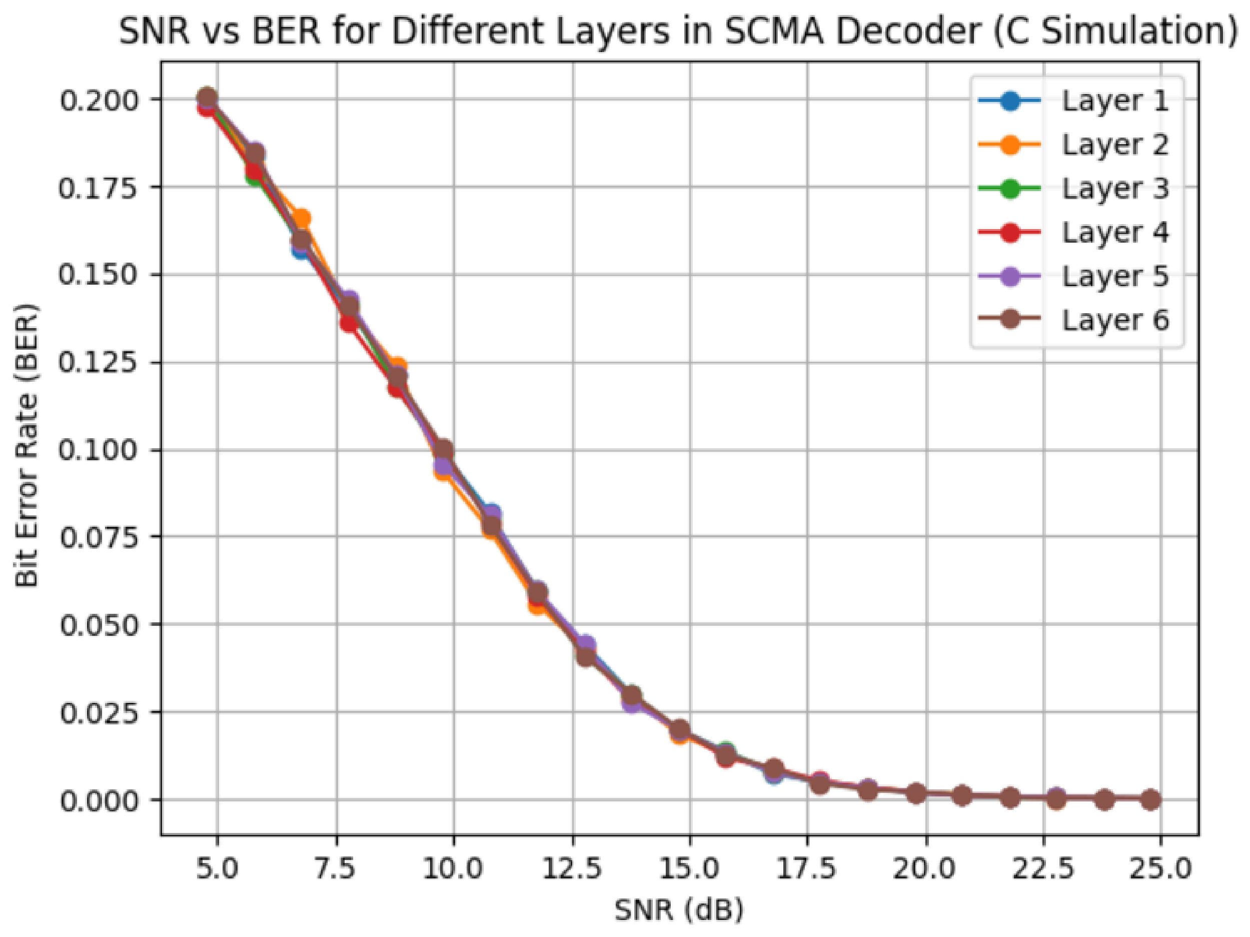
```
(base) prateek@DESKTOP-6D
(0.114534 + -0.809050i)
(-0.133710 + -0.089400i)
(0.877914 + -0.607059i)
(0.117609 + -0.089835i)
(base) prateek@DESKTOP-6D
```

This is the final LLR results which are matching in the final output with the matlab results.

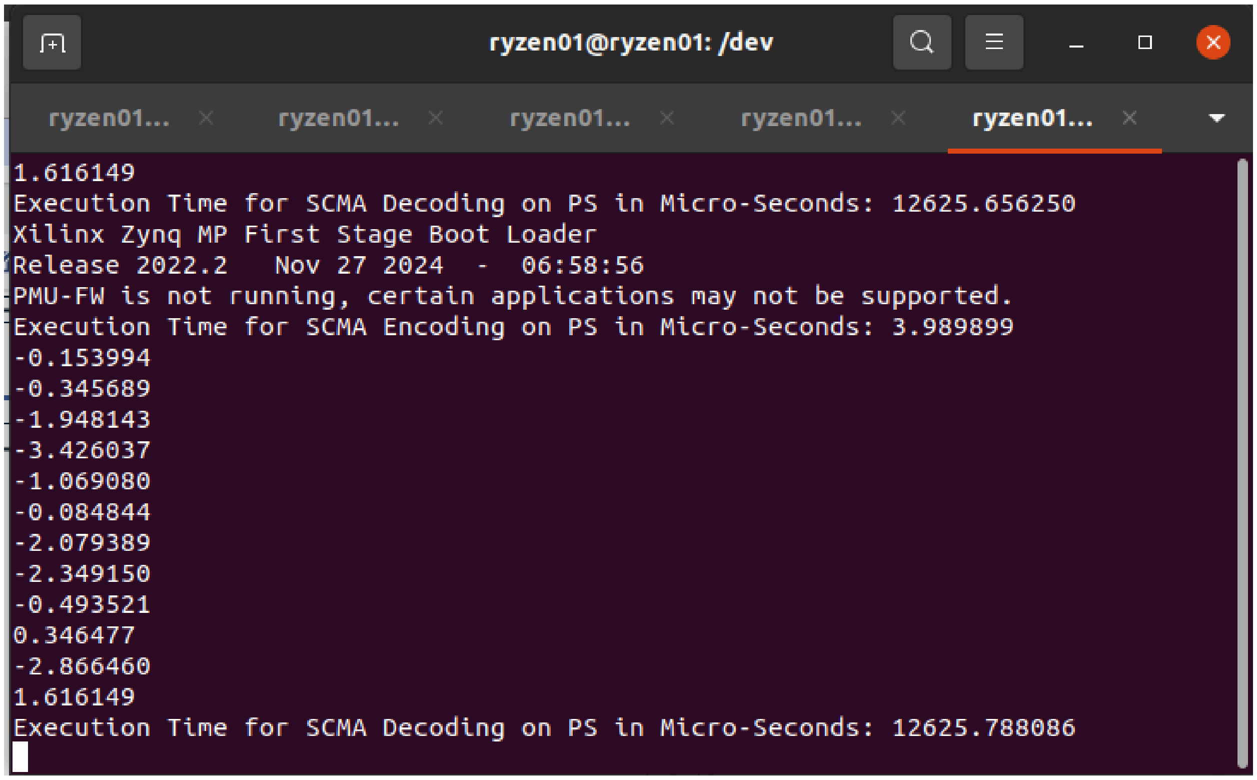


```
(base) prateek@DESKTOP-6DBV5A7
SCMA Encoding Time: 14.00 us
-0.153994
-0.345689
-1.948143
-3.426037
-1.069080
-0.084844
-2.079389
-2.349150
-0.493521
0.346477
-2.866460
1.616149
```

C implementation results for SNR ranging from 4.771213 to 24.771213 for various users. Here layers mean various users. We can observe that the graph is same as the Matlab implementation stating the correctness of the algorithm implementation.


SNR vs BER for Different Layers in SCMA Decoder (C Simulation)
Bit Error Rate (BER)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
SNR (dB)
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Layer 6

Hardware Implementation

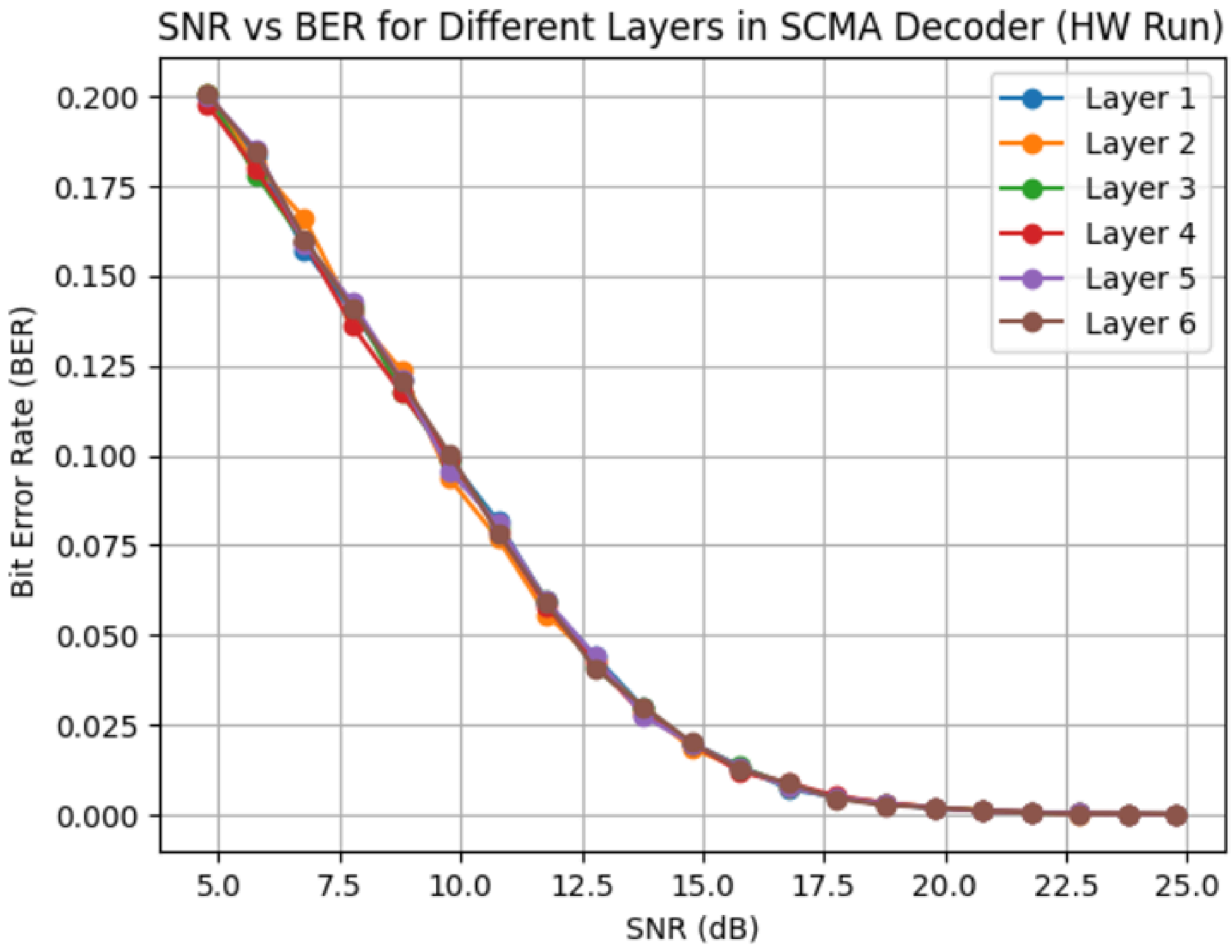


```
1.616149
Execution Time for SCMA Decoding on PS in Micro-Seconds: 12625.656250
Xilinx Zynq MP First Stage Boot Loader
Release 2022.2   Nov 27 2024  -  06:58:56
PMU-FW is not running, certain applications may not be supported.
Execution Time for SCMA Encoding on PS in Micro-Seconds: 3.989899
-0.153994
-0.345689
-1.948143
-3.426037
-1.069080
-0.084844
-2.079389
-2.349150
-0.493521
0.346477
-2.866460
1.616149
Execution Time for SCMA Decoding on PS in Micro-Seconds: 12625.788086
```

This is the execution result on the ZCU-106 FPGA board and its result was noted down.
Software implementation results:

- Execution Time for encoder: 3.98 us
- Execution time for decoder: 12.625 ms

Hardware implementation and its results are also mentioned in the table below. We can also observe that the SNR vs BER plot for the hardware implementation is the same as the Matlab and C implementation depicting the correct functionality of the algorithm on the hardware.



The SCMA decoder algorithm was implemented in two solutions for hardware acceleration using Vivado HLS: **Solution 1** (with no directives) and **Solution 2** (optimized with pipeline, and BRAM utilization). The SCMA decoder is also shown for MATLAB and C implementation for performance comparison. The following table summarizes the timing, latency, acceleration factor, and resource utilization metrics:

	Algorithm evaluation using Vivado HLS				
		Solution 1 (No Directive)	**Solution 2** (pipeline,BRAM)	**C Implementation**	**Matlab Implementation**
Timing Analysis	**Estimated Clock Period [ns]**	8.728 ns	18.608 ns	-	-
	Latency in clock cycles	659211	181781	-	-
	Latency in time [ms]	6.592ms	3.383 ms	12.626 ms	1526 ms
	Acceleratio n Factor	**231x**	**451x**	**120x**	**1x**
Resource utilization analysis	**BRAM**	56 (8%)	64 (10%)	-	-
	DSP	384 (22%)	802 (46%)	-	-
	FF	52691 (11%)	65924 (14%)	-	-
	LUT	50568 (21%)	73698 (31%)	-	-

Performance Improvement with Hardware Acceleration:

- The MATLAB implementation is the slowest, with a latency of 1,526 ms, serving as the baseline (1x acceleration).
- The C implementation provides a significant improvement, achieving a latency of 12.626 ms, corresponding to a **120x acceleration** factor compared to MATLAB.
- Hardware acceleration with Solution 1 (no directive) further reduces the latency to 6.592 ms (**231x acceleration**).
- The optimized Solution 2 achieves the best performance with a latency of 3.383 ms, corresponding to a **451x acceleration factor** compared to the MATLAB implementation.

Impact of Optimization Directives:

- Solution 2 achieves significantly reduced latency compared to Solution 1. This improvement is attributed to the use of optimization directives such as **pipelining** and **efficient BRAM utilization**, which enhance data processing speed and reduce bottlenecks.
- The estimated clock period for Solution 2 (18.608 ns) is slightly higher than that of Solution 1 (8.728 ns), but the overall reduction in latency is achieved due to reduced clock cycles.

Resource Utilization Analysis:

- Resource utilization increases in Solution 2 compared to Solution 1:
    - **BRAM usage** increases from 56 (8%) to 64 (10%), indicating efficient storage utilization for intermediate computations.
    - **DSP blocks** increase from 384 (22%) to 802 (46%), reflecting increased use of arithmetic resources for parallelized operations.
    - **Flip-flop (FF)** and **Look-Up Table (LUT)** usage also increase, with FF usage rising from 52,691 (11%) to 65,924 (14%) and LUT usage increasing from 50,568 (21%) to 73,698 (31%).
- The increased resource usage in Solution 2 is justified by the significant reduction in latency.

Trade-offs:

- While Solution 2 achieves the best performance in terms of latency and acceleration factor, it requires more hardware resources. This trade-off between resource utilization and performance is an important consideration for hardware design optimization.

Conclusion

The project demonstrates the successful optimization and hardware acceleration of the SCMA decoder algorithm. Solution 2, with pipelining, unrolling, and BRAM utilization, achieves the best latency (3.383 ms) and acceleration factor (451x) compared to MATLAB, with reasonable resource utilization. This highlights the importance of leveraging hardware optimization techniques for achieving high-performance embedded systems.