



[ECE573]: Advanced Embedded Logic Design  
Winter 2025

## Project Report 2

### DESIGN DEVELOPMENT OF IEEE 802.11A ARCHITECTURE ON ZC706

Instructor: Dr. Sumit J Darak

Mentor: Jai Mangal

**Ahilan R**

MT24170

*ahilan24170@iiitd.ac.in*

**Ippili Saravana Kumar**

MT24181

*ippili24181@iiitd.ac.in*

**Ashin VA**

MT24177

*ashin24177@iiitd.ac.in*

**Mintu Kumar**

2022296

*mintu22296@iiitd.ac.in*

**Shreemann JH**

MT24206

*shreemann24206@iiitd.ac.in*

May 5, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Splitter IPs</b>	<b>2</b>
<b>3</b>	<b>RRC Filter</b>	<b>3</b>
3.1	Non Optimized . . . . .	5
3.2	Optimized . . . . .	6
3.3	Memory Mapped Interfaced . . . . .	8
3.4	Word Length Optimized . . . . .	9
<b>4</b>	<b>Packet Detection Selection</b>	<b>10</b>
4.1	Non Optimized . . . . .	13
4.2	Optimized . . . . .	14
4.3	Word Length Optimized . . . . .	15
<b>5</b>	<b>Coarse CFO</b>	<b>18</b>
5.1	Non Optimized . . . . .	19
5.2	Optimized . . . . .	20
5.3	Word Length Optimized . . . . .	21
<b>6</b>	<b>Fine CFO</b>	<b>23</b>
6.1	Non Optimized . . . . .	24
6.2	Optimized . . . . .	25
6.3	Word Length Optimized . . . . .	25
<b>7</b>	<b>FFT</b>	<b>28</b>
7.1	Non Optimized . . . . .	30
7.2	Optimized . . . . .	31
7.3	Word Length Optimized . . . . .	33
<b>8</b>	<b>Channel Estimation</b>	<b>38</b>
8.1	Part 1 . . . . .	38
8.1.1	Non Optimized . . . . .	39
8.1.2	Optimized . . . . .	40
8.2	Part 2 . . . . .	41
8.2.1	Unoptimized . . . . .	43
8.2.2	Optimized . . . . .	44
8.2.3	Word Length Optimized . . . . .	45
<b>9</b>	<b>One Tap Equalizer</b>	<b>47</b>
9.1	Part 1 . . . . .	47
9.1.1	Non Optimized . . . . .	49
9.1.2	Optimized . . . . .	50
9.2	Part 2 . . . . .	51
9.3	Handling Division by Zero in OneTap Part 2 IP . . . . .	53

9.3.1	Unoptimized	54
9.3.2	Optimized	55
9.3.3	Word Length Optimized	56
<b>10</b>	<b>Results and Comparison</b>	<b>58</b>
10.1	Non Optimized	58
10.2	Optimized	58
10.3	Memory Mapped	59
10.4	Interrupt	60
10.5	Word Length Optimization	61
10.6	Result Graphs	62
10.6.1	Resource Utilization of Receiver Part	63
	<b>APPENDIX A JTAG Outputs</b>	<b>64</b>
	<b>APPENDIX B Output Screenshots</b>	<b>78</b>

# 1 Introduction

In the first part we implemented the entire transceiver in the PS, that is, modulation, transmitter, receiver and demodulation as well. The details of the implementations can be found in the Project Report 1. In the second part of the project we implement the whole Receiver part in the PL with various configurations, like optimizations & memory mapped. The block diagram of the overall flow can be seen in the below figure.

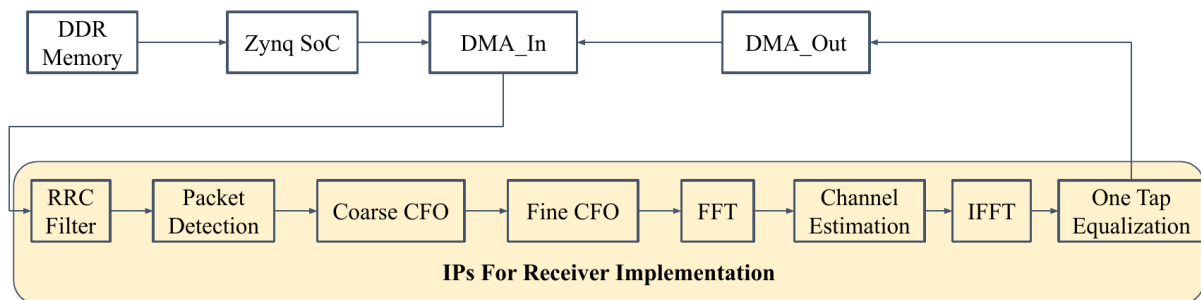


Figure 1: OFDM Block Diagram

The memory mapped version of the OFDM can be implemented in the following manner. Only the input part of RRC Filter and the output of the One Tap Equalizer is memory mapped interfaced, others are stream interfaced as earlier.

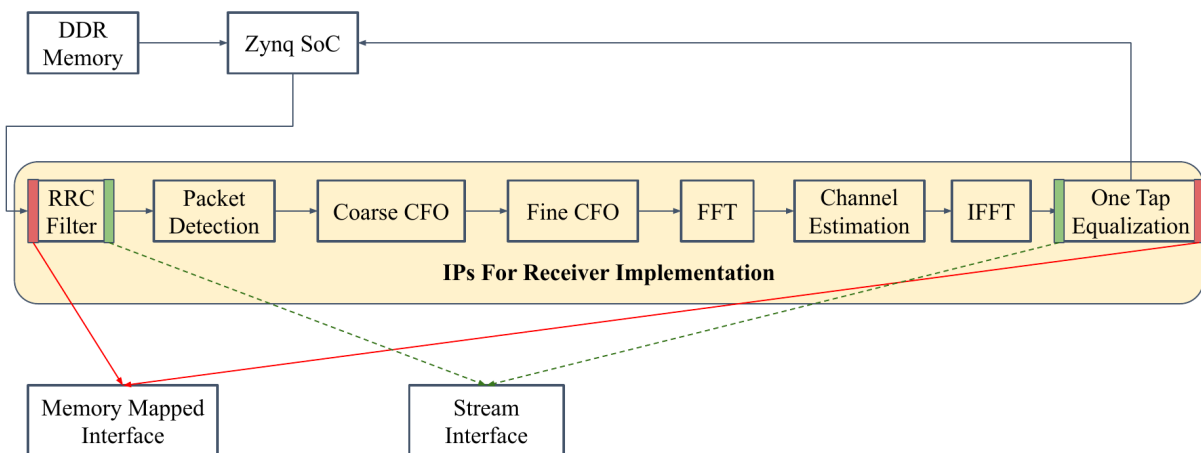


Figure 2: Memory Mapped OFDM Block Diagram

## 2 Splitter IPs

### 1. Rx\_Signal\_Splitter

- Used to receive the Rx\_Filtered\_Signal\_Combined and split into the Rx\_Signal (viz required for both RRC Filter & **Packet\_Detection**) and Rx\_Filtered\_Signal (viz passed on to **Packet\_Detection**).

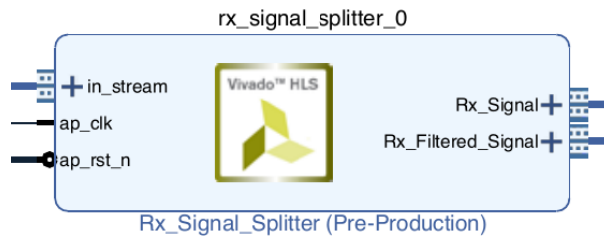


Figure 3: Rx\_Signal\_Splitter

### 2. Rx\_Frame\_Splitter

- Receives rx\_frame\_after\_fine and sends two copies of it to **Channel\_Estimation\_p1** for Long Preamble creation and **OneTap\_p1** for extracting Rx\_Payload\_no\_CP

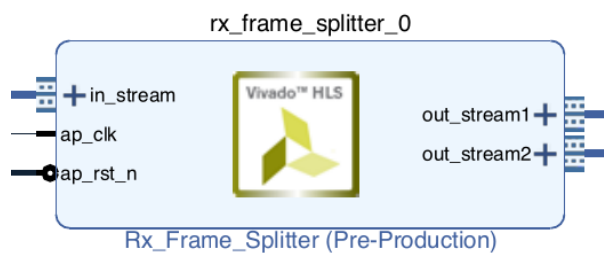


Figure 4: Rx\_Frame\_Splitter

### 3 RRC Filter

```
#ifndef solutionMM706
void RRC_MM706(std::complex<float> *Rx_Signal, hls::stream<axis_data>
↳ &Rx_Filtered_Signal_Combined){
    //#pragma HLS INTERFACE ap_ctrl_none port=return
    //#pragma HLS INTERFACE axis register both port=Rx_Signal
    #pragma HLS INTERFACE s_axilite port=return bundle=CTRL
    #pragma HLS INTERFACE axis register both port=Rx_Filtered_Signal_Combined
    #pragma HLS INTERFACE ap_ctrl_hs port=return
    #pragma HLS INTERFACE m_axi depth=3000 port=Rx_Signal offset=slave

    axis_data local_read, local_write;
    std::complex<float> Rx_Signal_Input[RX_SIGNAL_LEN];
    std::complex<float> Rx_Filtered_Signal_Output[RX_FILTERED_SIGNAL_LEN];

    // Read input stream into the local array
    // loop_read:
    // for(int i = 0; i < RX_SIGNAL_LEN; i++){
    //     local_read = Rx_Signal.read();
    //     Rx_Signal_Input[i] = local_read.data;
    // }
    memcpy(Rx_Signal_Input, (std::complex<float>*)Rx_Signal,
↳ RX_SIGNAL_LEN*sizeof(std::complex<float>));

    // Operation Starts
    std::complex<float> inp_sig_padded_rx[PADDED_LEN];
    // Initialize output array to zero
loop_init_output:
    for (int i = 0; i < RX_FILTERED_SIGNAL_LEN; i++) {
#pragma HLS PIPELINE II=1
        Rx_Filtered_Signal_Output[i] = 0;
    }

    // Fill the padded array with zeros then Rx_signal values and again zeros
loop_init_padded1:
    for (int i = 0; i < RRC_FILTER_RX_LENGTH - 1; i++) {
#pragma HLS PIPELINE II=1
        inp_sig_padded_rx[i] = 0;
    }
loop_copy_input:
    for (int i = 0; i < RX_SIGNAL_LEN; i++) {
#pragma HLS PIPELINE II=1
        inp_sig_padded_rx[RRC_FILTER_RX_LENGTH - 1 + i] = Rx_Signal_Input[i];
    }
loop_init_padded2:
    for (int i = 0; i < RRC_FILTER_RX_LENGTH - 1; i++) {
#pragma HLS PIPELINE II=1
        inp_sig_padded_rx[RRC_FILTER_RX_LENGTH - 1 + RX_SIGNAL_LEN + i] = 0;
    }

    // Perform convolution between rcc coeff rx and padded signal rx
loop_correlation:
    for (int n = 0; n < RX_FILTERED_SIGNAL_LEN; n++) {
#pragma HLS PIPELINE II=1
```

```

        for (int k = 0; k < RRC_FILTER_RX_LENGTH; k++) {
            Rx_Filtered_Signal_Output[n] += rrc_filter_rx[k] * inp_sig_padded_rx[n +
            ↪ k];
        }
    }

    // First: Original Rx_Signal
loop_output_input:
    for(int i = 0; i < RX_SIGNAL_LEN; i++){
#pragma HLS PIPELINE II=1
        local_write.data = Rx_Signal_Input[i];
        local_write.last = 0; // Not last yet
        Rx_Filtered_Signal_Combined.write(local_write);
    }
    // Then: Filtered signal
loop_output_filtered:
    for(int i = 0; i < RX_FILTERED_SIGNAL_LEN; i++){
#pragma HLS PIPELINE II=1
        local_write.data = Rx_Filtered_Signal_Output[i];
        local_write.last = (i == RX_FILTERED_SIGNAL_LEN - 1) ? 1 : 0; // Last
        ↪ element!
        Rx_Filtered_Signal_Combined.write(local_write);
    }
}
#endif

```

For Word Length Optimization:

```

#ifdef solutionWLOpt
typedef ap_fixed<24,2> fixed24_t;
typedef std::complex<fixed24_t> complex_fixed_t;

struct axis_data_fixed{
    complex_fixed_t data;
    ap_uint<1> last;
};

const ap_fixed<24,2> rrc_filter_rx_fixed[21] = {
    -0.000454720514876, 0.003536895555750, -0.007145608090912, 0.007579061905178,
    0.002143682427274, -0.010610686667250, 0.030011553981831, -0.053053433336248,
    -0.075028884954579, 0.409168714634052, 0.803738600397980, 0.409168714634052,
    -0.075028884954579, -0.053053433336248, 0.030011553981831, -0.010610686667250,
    0.002143682427274, 0.007579061905178, -0.007145608090912, 0.003536895555750,
    -0.000454720514876
};
#endif

```

### 3.1 Non Optimized

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.256	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
1186788	1186788	1186788	1186788	none

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	2999	2999	1	–	–	3000	no
Loop 2	3019	3019	1	–	–	3020	no
loop_read	3000	3000	1	–	–	3000	no
Loop 4	3039	3039	1	–	–	3040	no
loop_init_output	3020	3020	1	–	–	3020	no
loop_init_padded1	20	20	1	–	–	20	no
loop_copy_input	60000	60000	2	–	–	30000	no
loop_init_padded2	20	20	1	–	–	20	no
loop_correlation	1147600	1147600	380	–	–	3020	no
loop_correlation.1	378	378	18	–	–	21	no
loop_output_input	9000	9000	3	–	–	3000	no
loop_output_filtered	9060	9060	3	–	–	3020	no

(c) Loop Summary

Table 1: Performance Metrics for RRC (Unoptimized)

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	227	-
FIFO	-	-	-	-	-
Instance	-	8	491	1032	-
Memory	48	-	32	11	0
Multiplexer	-	-	-	924	-
Register	-	-	807	-	-
<b>Total</b>	48	8	1330	2194	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	17	3	1	4	0

Table 2: Utilization Estimates Summary for RRC (Unoptimized)



## 3.2 Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	9.064	1.25	57,461	57,461	57,461	57,461	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	2,999	2,999	1	—	—	3,000	no
Loop 2	3,019	3,019	1	—	—	3,020	no
loop_read	3,000	3,000	1	1	1	3,000	yes
Loop 4	3,039	3,039	1	—	—	3,040	no
loop_init_output	3,020	3,020	1	1	1	3,020	yes
loop_init_padded1	20	20	1	1	1	20	yes
loop_copy_input	3,000	3,000	2	1	1	3,000	yes
loop_init_padded2	20	20	1	1	1	20	yes
loop_correlation	33,303	33,303	95	11	1	3,020	yes
loop_output_input	3,001	3,001	3	1	1	3,000	yes
loop_output_filtered	3,021	3,021	3	1	1	3,020	yes

(c) Loop Summary

Table 3: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	469	-
FIFO	-	-	-	-	-
Instance	-	40	3418	5804	-
Memory	-	48	0	0	0
Multiplexer	-	-	-	228	-
Register	0	-	6264	1056	-
<b>Total</b>	0	88	9682	7557	0
<b>Available</b>	1090	900	437200	218600	0
<b>Utilization (%)</b>	0	10	2	3	0

Table 4: Resource Utilization Summary

- The optimized design of the RRC filter demonstrates significant performance improvements, particularly in the `loop_correlation`:
  - Latency dropped from **1,186,788** to **57,461** cycles.
  - Initiation Interval (II) reduced from **380** to just **11**, enabling much faster loop iteration starts.

- Other loops also show notable enhancements:
  - `loop_copy_input`, `loop_output_input`, and `loop_output_filtered` each achieved an II of **1**, allowing faster and more efficient execution.
- These improvements were attained through:
  - Effective pipelining.
  - Loop optimizations.
- Trade-offs observed:
  - Increased LUT usage (from **91** to **469**), which is a reasonable cost for the performance gains.
- Key challenges in the correlation loop:
  - Dependency requires each iteration over `k` to wait for the previous one to complete.
  - Solutions include rewriting accumulation using techniques like:
    - \* Partial sums.
    - \* Tree accumulation.
  - Memory access limitations: `inp_sig_padded_rx[n + k]` reads from an array—if multiple reads occur in close cycles without memory partitioning, II remains constrained.

### 3.3 Memory Mapped Interfaced

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
lap_clk	10.00	8.750	1.25	936,117	936,117	936,117	936,117	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	2,999	2,999	1	—	—	3,000	no
Loop 2	3,019	3,019	1	—	—	3,020	no
memory.Rx_Signal_Input... M_real.addr.1. .Rx_Signal...M_image	6,608	6,608	11	21	1	3,000	yes
Loop 4	3,039	3,039	1	—	—	3,040	no
loop_init_output	3,020	3,020	1	—	—	3,020	no
loop_init_padded1	20	20	1	—	—	20	no
loop_copy_input	6,000	6,000	2	—	—	3,000	no
loop_init_padded2	20	20	1	—	—	20	no
loop_correlation	893,920	893,920	296	—	—	3,020	no
+ loop_correlation.1	294	294	14	—	—	21	no
loop_output_input	9,060	9,060	3	—	—	3,000	no
loop_output_filtered	9,060	9,060	3	—	—	3,020	no

(c) Loop Summary

Table 5: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	299	-
FIFO	-	-	-	-	-
Instance	2	16	1590	2836	-
Memory	48	-	32	11	0
Multiplexer	-	-	-	140	-
Register	0	-	1069	64	-
<b>Total</b>	50	16	2691	3350	0
<b>Available</b>	1090	900	437200	218600	0
<b>Utilization (%)</b>	4	1	0	1	0

Table 6: Resource Utilization Summary

### 3.4 Word Length Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
lap_clk	10.00	8.599	1.25	48,326	48,326	48,326	48,326	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
loop_read	3,002	3,002	3,002	4	1	3,000	yes
loop_init_output	3,020	3,020	3,020	1	1	3,020	yes
loop_init_padded1	20	20	20	1	1	20	yes
loop_copy_input	3,000	3,000	3,000	2	1	3,000	yes
loop_init_padded2	20	20	20	1	1	20	yes
loop_correlation	33,222	33,222	33,222	14	11	3,020	yes
loop_output_input	3,003	3,003	3,003	5	1	3,000	yes
loop_output_filtered	3,023	3,023	3,023	5	1	3,020	yes

(c) Loop Summary

Table 7: Performance Metrics

Name	BRAM 18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	160	4563	-
FIFO	-	-	-	-	-
Instance	-	8	200	446	-
Memory	36	-	0	0	0
Multiplexer	-	-	-	167	-
Register	2	-	2761	161	-
<b>Total</b>	38	8	3121	5337	0
<b>Available</b>	1090	900	437200	218600	0
<b>Utilization (%)</b>	3	0	0	2	0

Table 8: Resource Utilization Summary

## 4 Packet Detection Selection

```
#ifndef solution706
void compute_normalized_correlation_HW_ZC706(const std::complex<float> Rx_signal[],
↪ int len_Rx,
  int delay_param, int window_length,
  float corr_out[], int out_length) {

    std::complex<float> sum_corr = std::complex<float>(0.0f, 0.0f);
    float sum_peak = 0.0;
    int n, k;

    // Compute the first window
    loop_first_window: for (k = 0; k < window_length; k++) {
        // #pragma HLS PIPELINE II=1
        std::complex<float> sample1 = Rx_signal[k];
        std::complex<float> sample2 = Rx_signal[k + delay_param];
        float real_part = sample1.real() * sample2.real() - sample1.imag() *
↪ sample2.imag();
        float imag_part = sample1.real() * sample2.imag() + sample1.imag() *
↪ sample2.real();
        sum_corr += std::complex<float>(real_part, imag_part);
        float sample2_abs_squared = sample2.real() * sample2.real() + sample2.imag()
↪ * sample2.imag();
        sum_peak += sample2_abs_squared;
    }

    if (sum_peak != 0){
        float sum_corr_magnitude_squared = sum_corr.real() * sum_corr.real() +
↪ sum_corr.imag() * sum_corr.imag();
        corr_out[0] = sum_corr_magnitude_squared / (sum_peak * sum_peak);
    }
    else {
        corr_out[0] = 0.0;
    }

    // Slide the window and update sums incrementally
    loop_slide_window: for (n = 1; n < out_length; n++) {
        // #pragma HLS PIPELINE II=1
        // Remove the contribution of the element that is leaving the window
        std::complex<float> old_sample1 = Rx_signal[n - 1];
        std::complex<float> old_sample2 = Rx_signal[n - 1 + delay_param];
        float real_part_old = old_sample1.real() * old_sample2.real() -
↪ old_sample1.imag() * old_sample2.imag();
        float imag_part_old = old_sample1.real() * old_sample2.imag() +
↪ old_sample1.imag() * old_sample2.real();
        sum_corr -= std::complex<float>(real_part_old, imag_part_old);
        float old_sample2_abs_squared = old_sample2.real() * old_sample2.real() +
↪ old_sample2.imag() * old_sample2.imag();
        sum_peak -= old_sample2_abs_squared;

        // Add the contribution of the new element entering the window
        std::complex<float> new_sample1 = Rx_signal[n + window_length - 1];
```

```

std::complex<float> new_sample2 = Rx_signal[n + window_length - 1 +
↳ delay_param];
sum_corr += new_sample1 * new_sample2;
float new_sample2_abs_squared = new_sample2.real() * new_sample2.real() +
↳ new_sample2.imag() * new_sample2.imag();
sum_peak += new_sample2_abs_squared;

// Compute normalized correlation for the current window
if (sum_peak != 0){
    float sum_corr_magnitude_squared = sum_corr.real() * sum_corr.real() +
↳ sum_corr.imag() * sum_corr.imag();
    corr_out[n] = sum_corr_magnitude_squared / (sum_peak * sum_peak);
}
else {
    corr_out[n] = 0.0;
}
}
}

int packetSelection_HW_ZC706(const float corr_out[]) {

    int candidate_indices[OUT_LENGTH];
    int candidate_count = 0;
    int last_candidate = -300; // Initialize so the first valid index always
↳ qualifies

    // Collect candidate indices where corr_out > PACKET_THRESHOLD and gap > 300
    loop_collect_candidates: for (int i = 0; i < OUT_LENGTH; i++) {
        // #pragma HLS PIPELINE II=1
        if (corr_out[i] > PACKET_THRESHOLD) {
            // Accept this index if it is separated from the last candidate by more
↳ than 300
            if (i - last_candidate > 300) {
                candidate_indices[candidate_count++] = i;
                last_candidate = i;
                if (candidate_count >= OUT_LENGTH)
                    break;
            }
        }
    }

    // Loop over collected candidates and check if candidate+THRESHOLD_LENGTH
↳ exceeds threshold
    loop_check_candidates: for (int i = 0; i < candidate_count; i++) {
        #pragma HLS LOOP_TRIPCOUNT min=1 max=10
        int candidate = candidate_indices[i];
        int check_index = candidate + THRESHOLD_LENGTH;
        if (check_index < OUT_LENGTH && corr_out[check_index] > PACKET_THRESHOLD) {
            // Adjust candidate by LENGTH_RRC_RX (assumed to be 10, hence +10+1 =>
↳ +11)
            return candidate + 11;
        }
    }
    return -1;
}
}

```

```

void Packet_Detection_ZC706(hls::stream<axis_data> &Rx_Signal,
↪ hls::stream<axis_data> &Rx_Filtered_Signal, hls::stream<axis_data> &Rx_Frame) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis register both port=Rx_Frame
#pragma HLS INTERFACE axis register both port=Rx_Filtered_Signal
#pragma HLS INTERFACE axis register both port=Rx_Signal

    axis_data local_read, local_write;

    std::complex<float> Rx_Signal_Input[RX_SIGNAL_LEN];
    #pragma HLS ARRAY_PARTITION variable=Rx_Signal_Input cyclic factor=4 dim=1
    std::complex<float> Rx_Filtered_Signal_Input[RX_FILTERED_SIGNAL_LEN];
    #pragma HLS ARRAY_PARTITION variable=Rx_Filtered_Signal_Input cyclic factor=4
    ↪ dim=1
    std::complex<float> Rx_Frame_Output[FRAME_LEN];
    #pragma HLS ARRAY_PARTITION variable=Rx_Frame_Output cyclic factor=4 dim=1

    // Read input from stream into Rx_signal_input
    loop_input: for (int i = 0; i < RX_SIGNAL_LEN; i++) {
        #pragma HLS PIPELINE II=1
        local_read = Rx_Signal.read();
        Rx_Signal_Input[i] = local_read.data;
    }

    // Read input from stream into Rx_signal_filtered_input
    loop_input_filtered: for (int i = 0; i < RX_FILTERED_SIGNAL_LEN; i++) {
        #pragma HLS PIPELINE II=1
        local_read = Rx_Filtered_Signal.read();
        Rx_Filtered_Signal_Input[i] = local_read.data;
    }

    //Run Normalized correlation
    float corr_out[OUT_LENGTH];
    #pragma HLS ARRAY_PARTITION variable=corr_out cyclic factor=4 dim=1
    compute_normalized_correlation_HW_ZC706(Rx_Signal_Input, LEN_RX, DELAY_PARAM,
    ↪ WINDOW_LENGTH, corr_out, OUT_LENGTH);

    //*****Packet
    ↪ Selection*****//
    // Run packet selection.
    int packet_idx = packetSelection_HW_ZC706(corr_out);

    if (packet_idx != -1){
        printf("Packet found!\n");
        printf("Packet start index: %d\n", packet_idx);
    }
    else
        printf("No valid packet detected.\n");

    ↪ //*****DOWN_SAMPLING*****
    // Calculate end index
    loop_downsampling: for (int j = 0; j < FRAME_TX_LEN; j++) {
        #pragma HLS PIPELINE II=1

```

```

    int i = packet_idx + j * OVERSAMPLINGFACTOR;
    Rx_Frame_Output[j] = Rx_Filtered_Signal_Input[i];
}

//Write back into stream output for Rx_Frame
loop_output: for(int i = 0; i < FRAME_LEN; i++) {
    #pragma HLS PIPELINE II=1
    local_write.data = Rx_Frame_Output[i];
    if(i == FRAME_LEN-1)
        local_write.last = 1;
    else
        local_write.last = 0;
    Rx_Frame.write(local_write);
}
}
#endif

```

## 4.1 Non Optimized

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.082	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline
min	max	min	max	Type
77,456	145,606	77,456	145,606	none

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline
		min	max	min	max	Type
grp_compute_normalized_c_fu_520	compute_normalized_c	59,524	115,630	59,524	115,630	none
grp_packetSelection_HW_1_fu_527	packetSelection_HW_1	3,005	15,049	3,005	15,049	none

(c) Instance Details

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	2,999	2,999	1	—	—	3,000	no
Loop 2	3,019	3,019	1	—	—	3,020	no
Loop 3	479	479	1	—	—	480	no
loop_input	3,000	3,000	1	—	—	3,000	no
loop_input_filtered	3,020	3,020	1	—	—	3,020	no
loop_downsampling	960	960	2	—	—	480	no
loop_output	1,440	1,440	3	—	—	480	no

(d) Loop Summary

Table 9: Performance Metrics



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	—	—	—	—	—
Expression	—	—	0	82	—
FIFO	—	—	—	—	—
Instance	3	10	2632	3306	0
Memory	42	—	0	0	0
Multiplexer	—	—	—	137	—
Register	—	—	513	—	—
<b>Total</b>	45	10	3145	3525	0
<b>Available</b>	1090	900	437200	218600	0
<b>Utilization (%)</b>	4	1	~0	1	0

Table 10: Resource Utilization Summary

## 4.2 Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	9.478	1.25	72,960	135,014	72,960	135,014	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline Type
		min	max	min	max	
grp_compute_normalized_c_fu_933	compute_normalized_c	59,460	112,614	59,460	112,614	none
grp_packetSelection_HW_Z_fu_949	packetSelection_HW_Z	8	8,908	8	8,908	none

(c) Instance Details

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	2,999	2,999	1	—	—	3,000	no
Loop 2	3,019	3,019	1	—	—	3,020	no
Loop 3	479	479	1	—	—	480	no
loop_input	3,000	3,000	1	1	1	3,000	yes
loop_input_filtered	3,020	3,020	1	1	1	3,020	yes
loop_downsampling	480	480	2	1	1	480	yes
loop_output	481	481	3	1	1	480	yes

(d) Loop Summary

Table 11: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	—	—	—	—	—
Expression	—	—	0	159	—
FIFO	—	—	—	—	—
Instance	3	30	4243	6717	0
Memory	48	—	0	0	0
Multiplexer	—	—	—	303	—
Register	—	—	535	—	—
<b>Total</b>	51	30	4778	7179	0
<b>Available</b>	1090	900	437200	218600	0
<b>Utilization (%)</b>	4	3	1	3	0

Table 12: Resource Utilization Summary

### 4.3 Word Length Optimized

```
#ifdef solutionWLOpt
typedef ap_fixed<22,2> fixed24_t;
typedef std::complex<fixed24_t> complex_fixed_t;
#endif
```

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.612	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
57,514	110,709	57,514	110,709	none

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline Type
		min	max	min	max	
grp_compute_normalized_c_fu_780	compute_normalized_c	50,508	94,803	50,508	94,803	none
grp_packetselection_HW_z_fu_796	packetselection_HW_z	8	8,908	8	8,908	none

(c) Instance Details

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
loop_input	3,002	3,002	4	1	1	3,000	yes
loop_input_filtered	3,022	3,022	4	1	1	3,020	yes
loop_downsampling	480	480	2	1	1	480	yes
loop_output	483	483	5	1	1	480	yes

(d) Loop Summary

Table 13: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	—	—	—	—	—
Expression	—	—	80	3423	—
FIFO	—	—	—	—	—
Instance	3	27	5187	14085	—
Memory	48	—	0	0	—
Multiplexer	—	—	—	240	—
Register	0	—	2102	192	—
<b>Total</b>	51	27	7369	17940	—
<b>Available</b>	1090	900	437200	218600	—
<b>Utilization (%)</b>	4	3	1	8	—

Table 14: Resource Utilization Summary

## Packet Detection IP Overview

The **Packet Detection IP** primarily consists of two functional blocks:

- **Normalization (Correlation)**
- **Packet Selection Algorithm**

These blocks were already highly optimized in their initial implementation. As a result, applying further pipelining within these blocks did not yield substantial improvements. This limitation is likely attributed to the presence of *loop-carried dependencies* and *complex control logic*, which restrict the effectiveness of pipelining techniques.

## Top-Level Pipelining Strategy

Due to the constraints within the core algorithm blocks, pipelining was selectively applied at the **top-level function**. The optimization targeted the following loops:

- `loop_input`: Reading input data
- `loop_input_filtered`: Processing filtered input
- `loop_downsampling`: Performing downsampling
- `loop_output`: Writing output data

These loops exhibit a more linear and dataflow-oriented behavior, making them suitable candidates for pipelining.

## Timing and Performance Results

After applying pipelining optimizations:

- The estimated clock period increased slightly from 9.082 ns to 9.478 ns.
- Despite this increase, the design still comfortably met the 10.00 ns target clock period.

This indicates that the optimization had no adverse effect on timing closure and overall performance remained robust.

## Latency and Initiation Interval Improvements

- `loop_input` and `loop_input_filtered` retained their original latency values but achieved an **Initiation Interval (II)** of 1, allowing a new iteration to begin every clock cycle.
- `loop_downsampling`: Latency reduced from 960 cycles to 480 cycles.
- `loop_output`: Latency reduced from 1440 cycles to 481 cycles.

These reductions in latency are primarily due to the removal of idle cycles and the increased ability to initiate loop iterations without delay.

## 5 Coarse CFO

```
void coarseCFO_opt(hls::stream<axis_data> &rx_frame,
                  hls::stream<axis_data> &rx_frame_after_coarse)
{
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register both port=rx_frame
    #pragma HLS INTERFACE axis register both port=rx_frame_after_coarse

    axis_data local_read, local_write;
    std::complex<float> rx_frame_array[rx_frame_len];
    std::complex<float> rx_frame_coarse_array[rx_frame_len];

    // Read entire frame from input stream with pipelining
    loop_read_frame: for(int i = 0; i < rx_frame_len; i++) {

        local_read = rx_frame.read();
        rx_frame_array[i] = local_read.data;
    }

    float sum_real = 0.0f, sum_imag = 0.0f;

    // Calculate the complex conjugate product for coarse CFO estimation with
    ↪ pipelining
    loop_coarse_prod: for (int i = 0; i < SHORT_PREAMBLE_SLOT_LENGTH; i++) {
        #pragma HLS PIPELINE
        int idx1 = SHORT_PREAMBLE_SLOT_LENGTH * 5 + i;
        int idx2 = SHORT_PREAMBLE_SLOT_LENGTH * 6 + i;

        // Staggered complex multiplication to reduce DSP usage
        float a = rx_frame_array[idx1].real();
        float b = rx_frame_array[idx1].imag();
        float c = rx_frame_array[idx2].real();
        float d = rx_frame_array[idx2].imag();

        // Serialized computation: (ac + bd) + i(bc - ad)
        float ac = a * c;
        float bd = b * d;
        sum_real += ac + bd;

        float bc = b * c;
        float ad = a * d;
        sum_imag += bc - ad;
    }

    // Estimate the coarse frequency offset
    float phase = atan2f(sum_imag, sum_real);
    float freq_coarse_est = (-1.0f / (2.0f * M_PI * SHORT_PREAMBLE_SLOT_LENGTH *
    ↪ ts_sec)) * phase;
    float var = (-2.0f * M_PI * freq_coarse_est * ts_sec);

    // Apply the coarse frequency offset to the received frame with pipelining
    loop_coarse_correction: for (int n = 0; n < rx_frame_len; n++) {
        #pragma HLS PIPELINE
        float angle = var * n;
    }
}
```

```

// Using CORDIC-based sin/cos approximation might be better here for
↪ hardware
std::complex<float> correction = std::complex<float>(cosf(angle),
↪ sinf(angle));
rx_frame_coarse_array[n] = rx_frame_array[n] * correction;
}

// Output the coarse-corrected frame with pipelining
loop_write_frame: for(int i = 0; i < rx_frame_len; i++) {
    #pragma HLS PIPELINE
    local_write.data = rx_frame_coarse_array[i];
    local_write.last = (i == rx_frame_len - 1) ? 1 : 0;
    rx_frame_after_coarse.write(local_write);
}
}

```

## 5.1 Non Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	8.665	1.25	23,314	25,379	23,314	25,379	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
Loop 2	479	479	1	—	—	480	no
loop_read_frame	480	480	1	—	—	480	no
loop_coarse_prod	256	256	16	—	—	16	no
loop_coarse_correction	20,160	22,080	42-46	—	—	480	no
loop_write_frame	1,440	1,440	3	—	—	480	no

(c) Loop Summary

Table 15: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	236	-
FIFO	-	-	-	-	-
Instance	4	44	6638	12838	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	818	-
Register	-	-	895	-	-
<b>Total</b>	10	44	7533	13892	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	3	20	7	26	0

Table 16: Resource Utilization Summary

## 5.2 Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	8.552	1.25	2,550	2,695	2,550	2,695	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
Loop 2	479	479	1	—	—	480	no
loop_read_frame	480	480	1	—	—	480	no
loop_coarse_prod	94	94	20	5	1	16	yes
loop_coarse_correction	514	514	36	1	1	480	yes
loop_write_frame	481	481	3	1	1	480	yes

(c) Loop Summary

Table 17: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	—	—	—	—	—
Expression	—	—	0	250	—
FIFO	—	—	—	—	—
Instance	4	47	8,015	13,205	—
Memory	6	—	0	0	0
Multiplexer	—	—	—	716	—
Register	0	—	1,383	144	—
<b>Total</b>	10	47	9,398	14,315	0
<b>Available</b>	280	220	106,400	53,200	0
<b>Utilization (%)</b>	3	21	8	26	0

Table 18: FPGA Resource Utilization Summary

### 5.3 Word Length Optimized

```
// Optimized fixed-point types
typedef ap_fixed<10, 2> fixed_input_t; // 10-bit total (4 integer)
typedef ap_fixed<18, 5> fixed_accum_t; // 18-bit accumulator
typedef ap_fixed<14, 2> fixed_angle_t; // 14-bit angle representation
```

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.662	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
2,465	2,564	2,465	2,564	none

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline Type
		min	max	min	max	
grp_atan2_cordic_float_s_fu_471	atan2_cordic_float_s	1	100	1	100	none
grp_generic_sincos_fu_479	generic_sincos	11	11	1	1	function

(c) Instance Details

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
Loop 2	479	479	1	—	—	480	no
loop_read_frame	480	480	1	—	—	480	no
loop_coarse_prod	19	19	5	1	1	16	yes
loop_coarse_correction	505	505	27	1	1	480	yes
loop_write_frame	481	481	3	1	1	480	yes

(d) Loop Summary

Table 19: Performance Metrics



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	–	–	–	–	–
Expression	–	0	80	3,436	–
FIFO	–	–	–	–	–
Instance	4	33	5,725	10,439	–
Memory	6	–	0	0	0
Multiplexer	–	–	–	124	–
Register	0	–	1,940	128	–

Table 20: FPGA Resource Utilization Summary

## Pipelining Optimization Analysis

The pipelined implementation shows a significant decrease in execution time. Only HLS pipelining was applied in this version; array partitioning techniques were not utilized. The loops `loop_coarse_prod` and `loop_coarse_correction` exhibited considerably high latencies in the non-pipelined version. Therefore, the primary optimization goal was to reduce their latency through pipelining.

Pipelining was also applied to the `loop_write_frame`, which resulted in a small improvement in latency. However, pipelining was not applied to `loop_read_frame`, as its iteration latency was already 1 and could not be further improved.

According to the synthesis report:

- The latency of `loop_coarse_correction` dropped dramatically from 20,160 to 514 cycles, with an achieved initiation interval (II) of 1. This means the loop can accept new inputs every clock cycle.
- For `loop_coarse_prod`, the latency reduced from 256 to 94 cycles. However, the achieved II is 5 due to loop-carried dependencies in the following expressions: `sum_real += ac + bd;`  
`sum_imag += bc - ad;` These dependencies rely on results from previous iterations, preventing the achievement of an II of 1.

In terms of resource utilization, the usage of DSP and BRAM units remained the same as in the non-pipelined version. However, there was a slight increase in the utilization of LUTs and flip-flops (FFs).

Additionally, the SDK output confirms a substantial performance gain, with the execution time dropping drastically from 223.17 ms to 20.11 ms.

## 6 Fine CFO

```
void fineCFO_opt(hls::stream<axis_data> &rx_frame_after_coarse,
                hls::stream<axis_data> &rx_frame_after_fine)
{
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register both port=rx_frame_after_coarse
    #pragma HLS INTERFACE axis register both port=rx_frame_after_fine

    axis_data local_read, local_write;
    std::complex<float> rx_frame_array[rx_frame_len];
    std::complex<float> rx_frame_fine_array[rx_frame_len];

    // Read entire frame from input stream
    loop_read_frame: for(int i = 0; i < rx_frame_len; i++) {
        local_read = rx_frame_after_coarse.read();
        rx_frame_array[i] = local_read.data;
    }

    float sum_real = 0.0f, sum_imag = 0.0f;
    const int corr_len = SHORT_PREAMBLE_SLOT_LENGTH * 4;

    // Calculate the complex conjugate product for fine CFO estimation
    std::complex<float> prod_consq_frame_fine = 0.0;
    loop_fine_prod: for (int i = 0; i < SHORT_PREAMBLE_SLOT_LENGTH * 4; i++) {
        #pragma HLS PIPELINE
        int idx1 = SHORT_PREAMBLE_SLOT_LENGTH * 12 + i;
        int idx2 = SHORT_PREAMBLE_SLOT_LENGTH * 16 + i;

        // Staggered complex multiplication to reduce DSP usage
        float a = rx_frame_array[idx1].real();
        float b = rx_frame_array[idx1].imag();
        float c = rx_frame_array[idx2].real();
        float d = rx_frame_array[idx2].imag();

        // Serialized computation: (ac + bd) + i(bc - ad)
        float ac = a * c;
        float bd = b * d;
        sum_real += ac + bd;

        float bc = b * c;
        float ad = a * d;
        sum_imag += bc - ad;
    }

    // Estimate the fine frequency offset
    float phase = atan2f(sum_imag, sum_real);
    float freq_fine_est = (-1.0 / (2.0 * M_PI * 64 * ts_sec)) * phase;
    float var = (-2.0 * M_PI * freq_fine_est * ts_sec);

    // Apply the fine frequency offset to the received frame
    loop_fine_correction: for (int n = 0; n < rx_frame_len; n++) {
        #pragma HLS PIPELINE
        float angle = var * n;
    }
}
```

```

std::complex<float> correction = std::complex<float>(cosf(angle),
↪  sinf(angle));
rx_frame_fine_array[n] = rx_frame_array[n] * correction;
}

// Output the fine-corrected frame
loop_write_frame: for(int i = 0; i < rx_frame_len; i++) {
    #pragma HLS PIPELINE
    local_write.data = rx_frame_fine_array[i];
    local_write.last = (i == rx_frame_len - 1) ? 1 : 0;
    rx_frame_after_fine.write(local_write);
}
}

```

## 6.1 Non Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	8.665	1.25	24,095	26,160	24,095	26,160	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
Loop 2	479	479	1	—	—	480	no
loop_read_frame	480	480	1	—	—	480	no
loop_fine_prod	1,024	1,024	16	—	—	64	no
loop_fine_correction	20,160	22,080	42-46	—	—	480	no
loop_write_frame	1,440	1,440	3	—	—	480	no

(c) Loop Summary

Table 21: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	236	-
FIFO	-	-	-	-	-
Instance	4	44	6638	12838	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	818	-
Register	-	-	895	-	-
<b>Total</b>	10	44	7533	13892	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	3	20	7	26	0

Table 22: Resource Utilization Summary

## 6.2 Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	8.552	1.25	2,803	2,948	2,803	2,948	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
Loop 2	479	479	1	—	—	480	no
loop_read_frame	480	480	1	—	—	480	no
loop_fine_prod	334	334	20	5	1	64	yes
loop_fine_correction	514	514	36	1	1	480	yes
loop_write_frame	481	481	3	1	1	480	yes

(c) Loop Summary

Table 23: Performance Metrics for IP3

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	243	-
FIFO	-	-	-	-	-
Instance	4	58	8550	14198	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	796	-
Register	0	-	1528	144	-
<b>Total</b>	10	58	10088	15381	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	3	26	9	28	0

Table 24: Resource Utilization Summary

## 6.3 Word Length Optimized

```
// Optimized fixed-point types - adjusted for better accuracy
typedef ap_fixed<12, 3> fixed_input_t; // Increased to 12-bit (5 integer)
typedef ap_fixed<20, 6> fixed_accum_t; // Increased to 20-bit accumulator
typedef ap_fixed<16, 3> fixed_angle_t; // Increased to 16-bit angle
```

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	8.621	1.25	2,531	2,676	2,531	2,676	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline Type
		min	max	min	max	
grp_atan2_cordic_float_s_fu_519	atan2_cordic_float_s	1	146	1	146	none
grp_generic_sincos_fu_527	generic_sincos	15	15	1	1	function

(c) Instance Details

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
Loop 2	479	479	1	—	—	480	no
loop_read_frame	480	480	1	—	—	480	no
loop_fine_prod	71	71	9	1	1	64	yes
loop_fine_correction	514	514	36	1	1	480	yes
loop_write_frame	481	481	3	1	1	480	yes

(d) Loop Summary

Table 25: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	—	4	—	—	—
Expression	—	—	80	9,280	—
FIFO	—	—	—	—	—
Instance	4	37	6,599	11,402	—
Memory	6	—	0	0	0
Multiplexer	—	—	—	603	—
Register	0	—	3,983	400	—
<b>Total</b>	10	41	10,662	21,685	0
<b>Available</b>	280	220	106,400	53,200	0
<b>Utilization (%)</b>	3	18	10	40	0

Table 26: FPGA Resource Utilization Summary

## Fine CFO Optimization Analysis

The working of the fine Carrier Frequency Offset (CFO) correction is structurally similar to the coarse CFO correction, with the key difference being the loop bounds in the `loop_fine_prod`. This loop processes more data, leading to higher latency in the fine CFO module compared to its coarse counterpart.

To address this, similar pipelining strategies were applied to the fine CFO loops. Specifically, HLS pipelining was introduced in the `loop_fine_prod` and `loop_fine_correction` loops,

which are computationally intensive and contribute significantly to the overall latency. Additionally, pipelining was also applied to the `loop_write_frame`, consistent with the coarse CFO implementation.

As a result of these optimizations, the synthesis report demonstrates a substantial reduction in loop latencies, analogous to those observed in the coarse CFO pipeline. Notably, the initiation interval (II) for the critical loops approaches 1 in some cases, indicating high throughput due to the effective pipelining.

In terms of resource utilization, the design remains efficient: there is no additional usage of DSPs or BRAMs, while only a minor increase is observed in the usage of flip-flops (FFs) and Look-Up Tables (LUTs).

## 7 FFT

```
#ifndef solution2_opt
// This function reorders the input to get the output in the normal order
void InputReorder_opt(const std::complex<float> dataIn[FFT_Size],
    ↪ std::complex<float> dataOut[FFT_Size]) {
    InputReorder_LOOP:
    for (int i = 0; i < FFT_Size; i++) {
        #pragma HLS PIPELINE II=1
        dataOut[i] = dataIn[input_reorder[i]];
    }
}

// For FFT of size FFT_Size, the number of butterfly stages are log2(FFT_Size)
// For 64-point FFT, there are six butterfly stages (2^6 = 64)
void FFTStages_opt(std::complex<float> FFT_input[FFT_Size], std::complex<float>
    ↪ FFT_output[FFT_Size]) {
    std::complex<float> stage1_out[FFT_Size], stage2_out[FFT_Size],
    ↪ stage3_out[FFT_Size];
    std::complex<float> stage4_out[FFT_Size], stage5_out[FFT_Size];

    // Stage 1 - Distance between butterflies: 2
    STAGE1_LOOP:
    for (int i = 0; i < FFT_Size; i+=2) {
        #pragma HLS PIPELINE II=1
        stage1_out[i] = FFT_input[i] + FFT_input[i+1];
        stage1_out[i+1] = FFT_input[i] - FFT_input[i+1];
    }

    // Stage 2 - Distance between butterflies: 4
    STAGE2_LOOP:
    for (int i = 0; i < FFT_Size; i+=4) {
        #pragma HLS PIPELINE II=1
        for (int j = 0; j < 2; ++j) {
            stage2_out[i+j] = stage1_out[i+j] +
            ↪ twiddle_factors[(j*FFT_Size/4)]*stage1_out[i+j+2];
            stage2_out[i+j+2] = stage1_out[i+j] -
            ↪ twiddle_factors[(j*FFT_Size/4)]*stage1_out[i+j+2];
        }
    }

    // Stage 3 - Distance between butterflies: 8
    STAGE3_LOOP:
    for (int i = 0; i < FFT_Size; i+=8) {
        #pragma HLS PIPELINE II=1
        for (int j = 0; j < 4; ++j) {
            stage3_out[i+j] = stage2_out[i+j] +
            ↪ twiddle_factors[(j*FFT_Size/8)]*stage2_out[i+j+4];
            stage3_out[i+j+4] = stage2_out[i+j] -
            ↪ twiddle_factors[(j*FFT_Size/8)]*stage2_out[i+j+4];
        }
    }

    // Stage 4 - Distance between butterflies: 16
    STAGE4_LOOP:
```

```

    for (int i = 0; i < FFT_Size; i+=16) {
#pragma HLS PIPELINE II=1
        for (int j = 0; j < 8; ++j) {
            stage4_out[i+j] = stage3_out[i+j] +
                ↪ twiddle_factors[(j*FFT_Size/16)]*stage3_out[i+j+8];
            stage4_out[i+j+8] = stage3_out[i+j] -
                ↪ twiddle_factors[(j*FFT_Size/16)]*stage3_out[i+j+8];
        }
    }

    // Stage 5 - Distance between butterflies: 32
    STAGE5_LOOP:
    for (int i = 0; i < FFT_Size; i+=32) {
#pragma HLS PIPELINE II=1
        for (int j = 0; j < 16; ++j) {
            stage5_out[i+j] = stage4_out[i+j] +
                ↪ twiddle_factors[(j*FFT_Size/32)]*stage4_out[i+j+16];
            stage5_out[i+j+16] = stage4_out[i+j] -
                ↪ twiddle_factors[(j*FFT_Size/32)]*stage4_out[i+j+16];
        }
    }

    // Stage 6 (Final Stage) - Distance between butterflies: 64
    STAGE6_LOOP:
    for (int i = 0; i < FFT_Size/2; i++) {
#pragma HLS PIPELINE II=1
        FFT_output[i] = stage5_out[i] +
            ↪ twiddle_factors[(i*FFT_Size/64)]*stage5_out[i+32];
        FFT_output[i+32] = stage5_out[i] -
            ↪ twiddle_factors[(i*FFT_Size/64)]*stage5_out[i+32];
    }
}

void do_fft_opt(const std::complex<float> fft_input[FFT_Size], std::complex<float>
    ↪ fft_output[FFT_Size]) {
#pragma HLS INLINE off
    std::complex<float> reordered_input[FFT_Size];
    // First, reorder the input
    InputReorder_opt(fft_input, reordered_input);
    // Then, perform the FFT butterfly stages
    FFTStages_opt(reordered_input, fft_output);
}

void fft_top_opt(hls::stream<fft_axis_data> &in_stream, hls::stream<fft_axis_data>
    ↪ &out_stream) {
#pragma HLS INTERFACE axis register both port=out_stream
#pragma HLS INTERFACE axis register both port=in_stream
#pragma HLS INTERFACE ap_ctrl_none port=return
    std::complex<float> input_local[FFT_Size];
    std::complex<float> output_local[FFT_Size];

    // Read input stream
    INPUT_STREAM_LOOP:
    for (int i = 0; i < FFT_Size; i++) {
        fft_axis_data input_data = in_stream.read();

```



```

        input_local[i] = input_data.data;
    }

    do_fft_opt(input_local, output_local);

    // Write output stream
    OUTPUT_STREAM_LOOP:
    for (int i = 0; i < FFT_Size; i++) {
#pragma HLS PIPELINE II=1
        fft_axis_data output_data;
        output_data.data = output_local[i];
        output_data.last = (i == FFT_Size - 1) ? 1 : 0;
        out_stream.write(output_data);
    }
}
#endif // fft_top_opt

```

## 7.1 Non Optimized

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.427	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline
min	max	min	max	Type
4134	4134	4134	4134	none

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline
		min	max	min	max	
grp_FFTStages_fu_337	FFTStages	3490	3490	3490	3490	none

(c) Instance Details

Loop Name	Latency		Iteration	Initiation Interval		Trip	Pipelined
	min	max		achieved	target		
Loop 1	63	63	1	—	—	64	no
Loop 2	63	63	1	—	—	64	no
INPUT_STREAM_LOOP	64	64	1	—	—	64	no
Loop 4	63	63	1	—	—	64	no
InputReorder_LOOP	192	192	3	—	—	64	no
OUTPUT_STREAM_LOOP	192	192	3	—	—	64	no

(d) Loop Summary

Table 27: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	—	—	—	—	—
Expression	—	—	0	207	—
FIFO	—	—	—	—	—
Instance	14	20	2493	4865	0
Memory	10	—	6	6	0
Multiplexer	—	—	—	575	—
Register	—	—	346	—	—
<b>Total</b>	24	20	2845	5653	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	8	9	2	10	0

Table 28: Resource Utilization Summary

## 7.2 Optimized

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.883	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
1009	1009	1009	1009	none

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline Type
		min	max	min	max	
grp_do_fft_opt_fu_242	do_fft_opt	748	748	748	748	none

(c) Instance Details

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	63	63	1	—	—	64	no
Loop 2	63	63	1	—	—	64	no
INPUT_STREAM_LOOP	64	64	1	—	—	64	no
OUTPUT_STREAM_LOOP	65	65	3	1	1	64	yes

(d) Loop Summary

Table 29: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	165	-
FIFO	-	-	-	-	-
Instance	26	24	10174	8864	0
Memory	6	-	0	0	0
Multiplexer	-	-	-	470	-
Register	-	-	312	-	-
<b>Total</b>	32	24	10486	9499	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	11	10	9	17	0

Table 30: Resource Utilization Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	63	63	1	—	—	64	no
Loop 2	63	63	1	—	—	64	no
Loop 3	63	63	1	—	—	64	no
Loop 4	63	63	1	—	—	64	no
Loop 5	63	63	1	—	—	64	no
STAGE1_LOOP	39	39	9	1	1	32	yes
STAGE2_LOOP	49	49	20	2	1	16	yes
STAGE3_LOOP	49	49	22	4	1	8	yes
STAGE4_LOOP	49	49	26	8	1	4	yes
STAGE5_LOOP	49	49	34	16	1	2	yes
STAGE6_LOOP	49	49	19	1	1	32	yes

Table 31: Loop Summary for FFT Stages

Reducing the number of bits used for calculations (called word length optimization) made the FFT design simpler and faster. Originally, the system used 32-bit numbers (16 bits for whole numbers, 16 bits for decimals). By shortening this to fewer bits (like 16 total bits), basic math operations (addition, multiplication) became less complex. Smaller numbers require fewer logic gates and wires, which directly saved resources like LUTs (used for logic) and FFs (used for storage), cutting their usage by 20-43%. This also sped up the design because simpler operations take less time to compute.

The system also started using specialized hardware blocks (DSP slices) more efficiently. These blocks are built for fast math and replaced slower, generic logic. This freed up resources further and reduced latency. For example, critical loops in stages like Stage 2 became 4x faster because the shorter bit-width simplified the logic and allowed better pipelining (like an assembly line for calculations).

Memory usage improved too. Smaller numbers mean smaller buffers for storing intermediate results. While the total block memory (BRAM) usage increased slightly, the design could access

data faster in parallel, improving overall performance.

Finally, these optimizations collectively reduced the total time to finish the FFT (latency) by 15% (from 1,009 to 853 cycles). By balancing precision and hardware efficiency, the system became faster, smaller, and cheaper to implement on an FPGA.

### 7.3 Word Length Optimized

```
/*----- base fixed-point type -----*/
typedef ap_fixed<32,16, AP_TRN, AP_WRAP> data_t;

/*----- tiny synthesizable complex type -----*/
struct cmplx_t {
    data_t re, im;
    cmplx_t() : re(0), im(0) {}
    cmplx_t(data_t r, data_t i = 0) : re(r), im(i) {}
    data_t real() const { return re; }
    data_t imag() const { return im; }

    /* C++98-safe operator overloads (use ctor, no {}-init) */
    friend cmplx_t operator+(const cmplx_t& a, const cmplx_t& b) {
        return cmplx_t(a.re + b.re, a.im + b.im);
    }
    friend cmplx_t operator-(const cmplx_t& a, const cmplx_t& b) {
        return cmplx_t(a.re - b.re, a.im - b.im);
    }
    friend cmplx_t operator*(const cmplx_t& a, const cmplx_t& b) {
        data_t r = a.re*b.re - a.im*b.im;
        data_t i = a.re*b.im + a.im*b.re;
        return cmplx_t(r, i);
    }
};

/*----- AXI-Stream word used by fft_top -----*/
struct fft_axis_data {
    cmplx_t data;
    ap_uint<1> last;
};

/*----- bit-reverse permutation -----*/
static const ap_uint<6> input_reorder[FFT_SIZE] = {
    0,32,16,48, 8,40,24,56, 4,36,20,52,12,44,28,60,
    2,34,18,50,10,42,26,58, 6,38,22,54,14,46,30,62,
    1,33,17,49, 9,41,25,57, 5,37,21,53,13,45,29,61,
    3,35,19,51,11,43,27,59, 7,39,23,55,15,47,31,63
};

/*----- twiddle-factor ROM -----*/
#define C(r,i) cmplx_t(data_t(r), data_t(i))
static const cmplx_t twiddle_factors[FFT_SIZE/2] = {
    C( 1.000000, 0.000000), C( 0.995185, -0.098017), C( 0.980785, -0.195090),
    C( 0.956940, -0.290285), C( 0.923880, -0.382683), C( 0.881921, -0.471397),
    C( 0.831470, -0.555570), C( 0.773010, -0.634393), C( 0.707107, -0.707107),
    C( 0.634393, -0.773010), C( 0.555570, -0.831470), C( 0.471397, -0.881921),
```

```

C( 0.382683, -0.923880), C( 0.290285, -0.956940), C( 0.195090, -0.980785),
C( 0.098017, -0.995185), C( 0.000000, -1.000000), C(-0.098017, -0.995185),
C(-0.195090, -0.980785), C(-0.290285, -0.956940), C(-0.382683, -0.923880),
C(-0.471397, -0.881921), C(-0.555570, -0.831470), C(-0.634393, -0.773010),
C(-0.707107, -0.707107), C(-0.773010, -0.634393), C(-0.831470, -0.555570),
C(-0.881921, -0.471397), C(-0.923880, -0.382683), C(-0.956940, -0.290285),
C(-0.980785, -0.195090), C(-0.995185, -0.098017)
};
#undef C

```

## Fixed-Point Data Type

The core of the word length configuration is the `ap_fixed<32,16>` type.

- **Format:** Q16.16 fixed-point (32-bit total).
  - **16-bit integer part:** Handles the dynamic range of FFT values.
  - **16-bit fractional part:** Provides precision down to  $2^{-16} \approx 1.5 \times 10^{-5}$ .
- **Rounding Mode:** `AP_TRN` (truncation toward negative infinity).
- **Overflow Mode:** `AP_WRAP` (wrap-around on overflow).

## Complex Number Representation

The `cmplx_t` struct encapsulates fixed-point arithmetic for complex numbers.

- **Arithmetic Rules:**
  - **Addition/Subtraction:** Directly uses `data_t` operations; no precision loss.
  - **Multiplication:**  $32b \times 32b \rightarrow 64b$  intermediate for normal. But it'll be truncated to 32b.
  - The 64-bit intermediate result is truncated to 32 bits, introducing quantization error.

## Twiddle Factor Quantization

Twiddle factors (complex exponentials) are precomputed and stored in Q16.16 format.

- **Precision Impact:**
  - Stored as 32-bit fixed-point values to match `data_t`.
  - Quantization errors from truncating floating-point values to Q16.16 are minimized by precomputation.

## Butterfly Stages and Word Length

The FFT stages (`FFTStages_wl`) perform arithmetic using the Q16.16 format:

- **Stage Operations:**

- **Addition/Subtraction:** No intermediate precision loss.
- **Multiplication with Twiddle Factors:**
  - \*  $32b \times 32b \rightarrow 64b$  product, truncated to 32b (Q16.16).
  - \* Accumulated errors are bounded by the fractional precision (16 bits).

## Trade-Offs

- **Accuracy vs. Resources:**

- **32-bit Width:** Balances precision (16 fractional bits) with FPGA resource usage (DSPs, LUTs).
- Smaller widths (e.g., 24b) reduce resources but risk overflow or larger errors.

- **Overflow Handling:**

- AP\_WRAP ensures no hardware exceptions but may cause unexpected results if inputs exceed  $\pm 32768$ .

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.921	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline
min	max	min	max	Type
853	853	853	853	none

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	63	63	1	—	—	64	no
Loop 2	63	63	1	—	—	64	no
READ_LOOP	64	64	1	1	1	64	yes
Loop 4	63	63	1	—	—	64	no
InputReorder_wl_LOOP	65	65	3	1	1	64	yes
Loop 6	63	63	1	—	—	64	no
Loop 7	63	63	1	—	—	64	no
Loop 8	63	63	1	—	—	64	no
Loop 9	63	63	1	—	—	64	no
Loop 10	63	63	1	—	—	64	no
Stage1_LOOP	33	33	3	1	1	32	yes
Stage2_LOOP	19	19	5	1	1	16	yes
Stage3_LOOP	20	20	7	2	1	8	yes
Stage4_LOOP	11	11	6	2	1	4	yes
Stage5_LOOP	11	11	8	4	1	2	yes
Stage6_LOOP	35	35	5	1	1	32	yes
WRITE_LOOP	65	65	3	1	1	64	yes

(c) Loop Summary

Table 32: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	565	-
FIFO	-	-	-	-	-
Instance	-	32	1028	1540	-
Memory	62	-	425	72	0
Multiplexer	-	-	-	5014	-
Register	0	-	4537	384	-
<b>Total</b>	62	32	5990	7575	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	22	14	5	14	0

Table 33: Resource Utilization Summary

Table 34: Performance Comparison of FFT Implementations

Metric	Non-Optimized	Optimized	Word Length Optimized
Clock Period (ns)	8.427	9.883	9.921
Total Latency (cycles)	4134	1009	853
Execution Time (ns)	~34,841	~10,007	~8,463
BRAM Utilization (%)	8	11	22
DSP Utilization (%)	9	10	14
FF Utilization (%)	2	9	5
LUT Utilization (%)	10	17	14
Pipelined Loops	0	7	9

## Key Observations

### 1. Performance:

- **Latency Reduction:** Word Length Optimized achieves the **lowest latency** (853 cycles), improving by **79%** over Non-Optimized and **15%** over Optimized
- **Execution Time:** Word Length Optimized is **4.1× faster** than Non-Optimized and **1.2× faster** than Optimized despite longer clock period

### 2. Resource Trade-offs:

- **BRAM/DSP:** Word Length Optimized uses **2.75× more BRAM** and **40% more DSPs** than Non-Optimized
- **FF/LUT:** Optimized uses most FFs (9%) and LUTs (17%), while Word Length Optimized reduces FF usage (5%) using BRAM/DSP

### 3. Pipeline Efficiency:

- **Non-Optimized:** No pipelined loops (all sequential)
- **Optimized:** 7 pipelined loops (e.g., FFT stages & output stream)

- **Word Length Optimized:** 9 pipelined loops (e.g., read/write, reordering, FFT stages)



## 8 Channel Estimation

### 8.1 Part 1

```
#ifndef solution2_opt
void channelEstimation_p1_opt(hls::stream<axis_data> &instream,
                             hls::stream<axis_data> &outstream_1,
                             hls::stream<axis_data> &outstream_2) {
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register both port=outstream_2
    #pragma HLS INTERFACE axis register both port=outstream_1
    #pragma HLS INTERFACE axis register both port=instream

    // rx_frame_after_fine:
    std::complex<float> rx_frame_after_fine[RX_FRAME_LEN];
    // Long_preamble_1:
    std::complex<float> Long_preamble_1[N_FFT];
    // Long_preamble_2:
    std::complex<float> Long_preamble_2[N_FFT];

    // Read input stream
    INPUT_STREAM_READ:
    for (int i = 0; i < RX_FRAME_LEN; i++) {
        axis_data input_data = instream.read();
        rx_frame_after_fine[i] = input_data.data;
    }

    LONG_PREAMBLE_EXTRACT:
    for (int i = 0; i < N_FFT; i++) {
        #pragma HLS UNROLL
        Long_preamble_1[i] = rx_frame_after_fine[SHORT_PREAMBLE_SLOT_LENGTH * 12 + i];
        Long_preamble_2[i] = rx_frame_after_fine[SHORT_PREAMBLE_SLOT_LENGTH * 16 + i];
    }

    // Write output stream

    OP_STREAM_LOOP_LP1:
    for (int i = 0; i < N_FFT; i++) {
        #pragma HLS PIPELINE II=1
        axis_data output_data1;
        output_data1.data = Long_preamble_1[i];
        output_data1.last = (i == N_FFT - 1) ? 1 : 0;
        outstream_1.write(output_data1);
    }

    OP_STREAM_LOOP_LP2:
    for (int i = 0; i < N_FFT; i++) {
        #pragma HLS PIPELINE II=1
        axis_data output_data2;
        output_data2.data = Long_preamble_2[i];
        output_data2.last = (i == N_FFT - 1) ? 1 : 0;
        outstream_2.write(output_data2);
    }
}
```

```
#endif // solution2_opt
```

### 8.1.1 Non Optimized

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.508	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
1604	1604	1604	1604	none

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
Loop 2	63	63	1	—	—	64	no
Loop 3	63	63	1	—	—	64	no
INPUT_STREAM_READ	480	480	1	—	—	480	no
LONG_PREAMBLE_EXTRACT	128	128	2	—	—	64	no
OP_STREAM_LOOP_LP1	192	192	3	—	—	64	no
OP_STREAM_LOOP_LP2	192	192	3	—	—	64	no

(c) Loop Summary

Table 35: Performance Metrics for Channel Estimation (Non Optimized)

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	280	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	8	-	0	0	0
Multiplexer	-	-	-	500	-
Register	-	-	497	-	-
<b>Total</b>	8	0	497	780	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	2	0	0	1	0

Table 36: Utilization Estimates Summary for Channel Estimation (Non Optimized)

### 8.1.2 Optimized

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.508	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
1287	1287	1287	1287	none

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
Loop 2	63	63	1	—	—	64	no
Loop 3	63	63	1	—	—	64	no
INPUT_STREAM_READ	480	480	1	—	—	480	no
OP_STREAM_LOOP_LP1	65	65	3	1	1	64	yes
OP_STREAM_LOOP_LP2	65	65	3	1	1	64	yes

(c) Loop Summary

Table 37: Performance Metrics for Channel Estimation (Optimized)

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	266	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	8	-	0	0	0
Multiplexer	-	-	-	3087	-
Register	-	-	539	-	-
<b>Total</b>	8	0	539	3353	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	2	0	0	6	0

Table 38: Utilization Estimates Summary for Channel Estimation (Optimized)

- Pipeline optimizations observed:
  - OP\_STREAM\_LOOP\_LP1 and OP\_STREAM\_LOOP\_LP2 have been pipelined
    - \* Latency reduction: 127 cycles each
- Effects of unrolling LONG\_PREAMBLE\_EXTRACT:
  - Variables were not partitioned
  - Pipeline can process 2 reads per cycle
  - Resulting latency:
    - \* 64 cycles for reads

- \* +1 cycle to finish
- \* Total = 65 cycles
- Total latency calculation:
  - Original latency: 1604 cycles
  - Subtracting optimizations:
    - \* -127 (LP1 pipelining)
    - \* -127 (LP2 pipelining)
    - \* -63 (read optimization)
  - Final latency: 1287 cycles

## 8.2 Part 2

```
#ifndef solution1
// Function to perform FFT shift (reordering the frequency bins)
void fftshift_opt(std::complex<float> *X) {
#pragma HLS PIPELINE
  int mid = N_FFT / 2;
  FFTSHIFT_OPT_LOOP:
  for (int i = 0; i < mid; i++) {
    std::complex<float> temp = X[i];
    X[i] = X[i + mid];
    X[i + mid] = temp;
  }
}
#endif // fftshift_opt

#ifdef opt_Zed
void channelEstimation_p2_opt_Zed(hls::stream<axis_data> &instream_1,
    hls::stream<axis_data> &instream_2,
    hls::stream<axis_data> &outstream)
{
#pragma HLS INTERFACE axis register both port=outstream
#pragma HLS INTERFACE axis register both port=instream_2
#pragma HLS INTERFACE axis register both port=instream_1
#pragma HLS INTERFACE ap_ctrl_none port=return
  // Local Variables Declaration
  std::complex<float> LP1_after_FFT[N_FFT];
  std::complex<float> LP2_after_FFT[N_FFT];
  std::complex<float> H_est[N_FFT];

  // Read input stream - Long_preamble_1_after_FFT
  INPUT_STREAM_LP1:
  for (int i = 0; i < N_FFT; i++) {
    axis_data input_data1 = instream_1.read();
    LP1_after_FFT[i] = input_data1.data;
  }

  // Read input stream - Long_preamble_2_after_FFT
```

```

    INPUT_STREAM_LP2:
    for (int i = 0; i < N_FFT; i++) {
        axis_data input_data2 = instream_2.read();
        LP2_after_FFT[i] = input_data2.data;
    }

    fftshift_opt(LP1_after_FFT);
    fftshift_opt(LP2_after_FFT);

    //VALUES OF SLOT FREQUENCY
    std::complex<float> Long_preamble_slot_Frequency[N_FFT] = {0};

    L_k_slot:
    for (int i = 0; i < 53; i++) {
        Long_preamble_slot_Frequency[i + 6] = L_k[i];
    }

    // Perform the computation
    H_est_Comp:
    for (int i = 0; i < 64; i++) {
#pragma HLS PIPELINE II=1
        H_est[i] = 0.5f
            * (LP1_after_FFT[i] + LP2_after_FFT[i])
            * std::conj(Long_preamble_slot_Frequency[i]);
    }

    // Write output stream
    OUTPUT_STREAM_LOOP:
    for (int i = 0; i < N_FFT; i++) {
#pragma HLS PIPELINE II=1
        axis_data output_data;
        output_data.data = H_est[i];
        output_data.last = (i == N_FFT - 1) ? 1 : 0;
        ostream.write(output_data);
    }
}
#endif

```

### 8.2.1 Unoptimized

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.256	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
2310	2310	2310	2310	none

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	63	63	1	—	—	64	no
Loop 2	63	63	1	—	—	64	no
Loop 3	63	63	1	—	—	64	no
INPUT_STREAM.LP1	64	64	1	—	—	64	no
INPUT_STREAM.LP2	64	64	1	—	—	64	no
FFTSHIFT_LOOP	64	64	2	—	—	32	no
FFTSHIFT_LOOP	64	64	2	—	—	32	no
H_est_Comp	1664	1664	26	—	—	64	no
OUTPUT_STREAM.LOOP	192	192	3	—	—	64	no

(c) Loop Summary

Table 39: Performance Metrics Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	—	—	—	—	—
Expression	—	—	0	313	—
FIFO	—	—	—	—	—
Instance	—	16	982	2064	—
Memory	11	—	0	0	0
Multiplexer	—	—	—	894	—
Register	—	—	937	—	—
<b>Total</b>	11	16	1919	3271	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	3	7	1	6	0

Table 40: Resource Utilization Summary

## 8.2.2 Optimized

Clock	Target	Estimated	Uncertainty
lap_clk	10.00	7.256	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
543	543	543	543	none

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline Type
		min	max	min	max	
gpp_fftshift_opt_fu_394	fftshift_opt	63	63	64	64	function
gpp_fftshift_opt_fu_400	fftshift_opt	63	63	64	64	function

(c) Instance Details

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	63	63	1	—	—	64	no
Loop 2	63	63	1	—	—	64	no
Loop 3	63	63	1	—	—	64	no
INPUT_STREAM_LP1	64	64	1	—	—	64	no
INPUT_STREAM_LP2	64	64	1	—	—	64	no
H_est_Comp	88	88	26	1	1	64	yes
OUTPUT_STREAM_LOOP	65	65	3	1	1	64	yes

(d) Loop Summary

Table 41: Performance Metrics Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	261	-
FIFO	-	-	-	-	-
Instance	-	36	10438	8998	-
Memory	11	-	0	0	0
Multiplexer	-	-	-	1041	-
Register	0	-	1304	64	-
<b>Total</b>	11	36	11742	10364	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	3	16	11	19	0

Table 42: Resource Utilization Summary

- Pipeline optimizations performed:
  - Loop "H\_est\_Comp" pipelining:
    - \* Latency reduction: 1567 cycles

- **Output pipeline** optimization:
  - \* Additional latency reduction: 127 cycles
- **fftshift\_opt** function instances pipelining (from table):
  - \* gpp\_flishift\_opt\_fu\_394 and gpp\_flishift\_opt\_fu\_400
  - \* Pipeline Type: 64 cycles
  - \* Latency calculation:  $63 + 1 = 64$  cycles
  - \* Total reduction: 64 cycles
- **Total latency calculation:**
  - Original latency: 2310 cycles
  - Subtracting optimizations:
    - \* -1567 (H\_est\_Comp pipelining)
    - \* -127 (output pipeline)
    - \* -64 (fftshift\_opt pipelining)
  - Final latency: 543 cycles

### 8.2.3 Word Length Optimized

```
#ifndef CHEST_H
#define CHEST_H

#include <complex>
#include <stdio.h>
#include <string.h>
#ifdef solutionWLOpt
typedef ap_fixed<14,3> fixed24_t;
typedef std::complex<fixed24_t> complex_fixed_t;
#endif
```



Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.599	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline
min	max	min	max	Type
523	523	523	523	none

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
INPUT_STREAM_LP1	66	66	4	1	1	64	yes
INPUT_STREAM_LP2	66	66	4	1	1	64	yes
Loop 3	64	64	2	—	—	32	no
Loop 4	64	64	2	—	—	32	no
INIT_ARRAY	64	64	1	1	1	64	yes
L_k_slot	53	53	1	1	1	53	yes
H_est_Comp	65	65	3	1	1	64	yes
OUTPUT_STREAM_LOOP	67	67	5	1	1	64	yes

(c) Loop Summary

Table 43: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	—	2	—	—	—
Expression	—	—	80	3209	—
FIFO	—	—	—	—	—
Instance	—	0	200	286	—
Memory	4	—	69	41	0
Multiplexer	—	—	—	149	—
Register	4	—	1887	194	—
<b>Total</b>	8	2	2236	3879	0
<b>Available</b>	1090	900	437200	218600	0
<b>Utilization (%)</b>	~0	~0	~0	1	0

Table 44: Resource Utilization Summary

We can see that resources has reduced and even performance has increased slightly.

## 9 One Tap Equalizer

### 9.1 Part 1

```
#ifndef opt_new
void OneTapEqualizer_p1_opt_new(
    hls::stream<axis_data> &rx_frame_stream,
    hls::stream<axis_data> &Payload_1_NoCP_stream,
    hls::stream<axis_data> &Payload_2_NoCP_stream) {
    #pragma HLS INTERFACE axis register both port=Payload_2_NoCP_stream
    #pragma HLS INTERFACE axis register both port=Payload_1_NoCP_stream
    #pragma HLS INTERFACE axis register both port=rx_frame_stream
    #pragma HLS INTERFACE ap_ctrl_none port=return

    // Read rx_frame_after_fine from rx_frame_stream
    std::complex<float> rx_frame_after_fine[RX_FRAME_LEN];
    #pragma HLS ARRAY_PARTITION variable=rx_frame_after_fine cyclic factor=2 dim=1
    RX_FRAME_STREAM_LOOP:
    for (int i = 0; i < RX_FRAME_LEN; i++) {
        axis_data input_data = rx_frame_stream.read();
        rx_frame_after_fine[i] = input_data.data;
    }

    // Step 1: Extract RX_Payload_1_time (elements 321 to 400)
    std::complex<float> RX_Payload_1_time[80];
    #pragma HLS ARRAY_PARTITION variable=RX_Payload_1_time cyclic factor=2 dim=1
    RX_PL1_TIME_EXT:
    for (int i = 0; i < 80; i++) {
        #pragma HLS UNROLL
        RX_Payload_1_time[i] = rx_frame_after_fine[320 + i];
    }

    // Step 2: Remove cyclic prefix (first 16 elements)
    std::complex<float> RX_Payload_1_no_CP[N_FFT];
    #pragma HLS ARRAY_PARTITION variable=RX_Payload_1_no_CP cyclic factor=2 dim=1
    REMOVE_CP1:
    for (int i = 0; i < N_FFT; i++) {
        #pragma HLS UNROLL
        RX_Payload_1_no_CP[i] = RX_Payload_1_time[16 + i];
    }

    // Step 1: Extract RX_Payload_2_time (elements 401 to 480)
    std::complex<float> RX_Payload_2_time[80];
    #pragma HLS ARRAY_PARTITION variable=RX_Payload_2_time cyclic factor=2 dim=1
    RX_PL2_TIME_EXT:
    for (int i = 0; i < 80; i++) {
        #pragma HLS UNROLL
        RX_Payload_2_time[i] = rx_frame_after_fine[400 + i];
    }

    // Step 2: Remove cyclic prefix (first 16 elements)
    std::complex<float> RX_Payload_2_no_CP[N_FFT];
    #pragma HLS ARRAY_PARTITION variable=RX_Payload_2_no_CP cyclic factor=2 dim=1
    REMOVE_CP2:
    for (int i = 0; i < N_FFT; i++) {
```

```

#pragma HLS UNROLL
    RX_Payload_2_no_CP[i] = RX_Payload_2_time[16 + i];
}

// Write Payload_1_NoCP_stream from RX_Payload_1_no_CP
OP_STREAM_LOOP_LP1:
for (int i = 0; i < N_FFT; i++) {
#pragma HLS PIPELINE II=1
    axis_data output_data1;
    output_data1.data = RX_Payload_1_no_CP[i];
    output_data1.last = (i == N_FFT - 1) ? 1 : 0;
    Payload_1_NoCP_stream.write(output_data1);
}

// Write Payload_2_NoCP_stream from RX_Payload_2_no_CP
OP_STREAM_LOOP_LP2:
for (int i = 0; i < N_FFT; i++) {
#pragma HLS PIPELINE II=1
    axis_data output_data2;
    output_data2.data = RX_Payload_2_no_CP[i];
    output_data2.last = (i == N_FFT - 1) ? 1 : 0;
    Payload_2_NoCP_stream.write(output_data2);
}
}
#endif // OneTapEqualizer_p1_opt_Zed

```

### 9.1.1 Non Optimized

Clock	Target	Estimated	Uncertainty	Latency min	Latency max	Interval min	Interval max	Pipeline Type
lap_clk	10.00	5.542	1.25	2215	2215	2215	2215	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
RX_FRAME_STREAM_LOOP	480	480	1	—	—	480	no
Loop 3	79	79	1	—	—	80	no
Loop 4	160	160	2	—	—	80	no
Loop 5	63	63	1	—	—	64	no
Loop 6	128	128	2	—	—	64	no
Loop 7	79	79	1	—	—	80	no
Loop 8	160	160	2	—	—	80	no
Loop 9	63	63	1	—	—	64	no
Loop 10	128	128	2	—	—	64	no
OP_STREAM_LOOP_LP1	192	192	3	—	—	64	no
OP_STREAM_LOOP_LP2	192	192	3	—	—	64	no

(c) Loop Summary

Table 45: Performance Metrics for IP3

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	184	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	10	-	0	149	0
Multiplexer	-	-	561	-	-
Register	-	-	561	-	-
<b>Total</b>	10	0	561	333	0
<b>Available</b>	1090	900	437200	218600	0
<b>Utilization (%)</b>	~0	0	~0	~0	0

Table 46: Resource Utilization Summary

### 9.1.2 Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	3.952	1.25	1,271	1,271	1,271	1,271	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	479	479	1	—	—	480	no
RX_FRAME_STREAM_LOOP	480	480	1	—	—	480	no
Loop 3	63	63	1	—	—	64	no
Loop 4	63	63	1	—	—	64	no
OP_STREAM_LOOP_LP1	65	65	3	1	1	64	yes
OP_STREAM_LOOP_LP2	65	65	3	1	1	64	yes

(c) Loop Summary

Table 47: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	394	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	24	-	0	0	0
Multiplexer	-	-	-	4673	-
Register	-	-	4621	-	-
<b>Total</b>	24	0	4621	5067	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	8	0	4	9	0

Table 48: Resource Utilization Summary

#### • Optimal Loop Unrolling Strategy:

##### – Array Partitioning is essential for efficient unrolling:

- \* Enables parallel memory access during unrolled iterations
- \* Must balance performance gains with resource constraints
- \* Excessive partitioning can lead to:
  - Dramatic increase in LUT/FF utilization routing congestion
- \* This especially happens when it is called variable indices, which can lead the resource utilization to more than 100%

– **Streaming Input Considerations:**

- \* For streamed data inputs, recommend **cyclic(2) partitioning**:
  - Limits to 2 parallel reads/writes per cycle
  - Maintains resource utilization within acceptable limits

• **Performance Optimization Results:**

– **Pipelined Output Stream** achieves critical improvements:

- \* Initiation Interval (II) = 1 (new iteration every cycle)
- \* Latency reduction visible even with partial unrolling

– **Design Trade-offs:**

- \* Full unrolling not always necessary for performance gains
- \* Balanced approach yields:
  - 2× throughput improvement (cyclic(2))
  - Contained resource growth
  - Sustained clock frequency

## 9.2 Part 2

```
void fftshift_opt(std::complex<float> *X) {
    #pragma HLS INLINE off
    int mid = N_FFT / 2;
    FFTSHIFT_LOOP:
    for (int i = 0; i < mid; i++) {
        //#pragma HLS UNROLL factor=2
        #pragma HLS PIPELINE
        std::complex<float> temp = X[i];
        X[i] = X[i + mid];
        X[i + mid] = temp;
    }
}

#ifdef opt_Zed
void OneTapEq_p2_opt_Zed(hls::stream<axis_data> &PL1_Freq_stream,
    hls::stream<axis_data> &PL2_Freq_stream,
    hls::stream<axis_data> &H_est_stream,
    hls::stream<axis_data> &outstream) {
    #pragma HLS INTERFACE axis register both port=outstream
    #pragma HLS INTERFACE axis register both port=H_est_stream
    #pragma HLS INTERFACE axis register both port=PL2_Freq_stream
    #pragma HLS INTERFACE axis register both port=PL1_Freq_stream
    #pragma HLS INTERFACE ap_ctrl_none port=return

    // Local Variables Declaration
    std::complex<float> PL1_Freq[N_FFT];
    std::complex<float> PL2_Freq[N_FFT];
}
```

```

std::complex<float> H_est[N_FFT];
std::complex<float> PL_FreqEq_Comb[4*N_FFT];

    // Read input stream - Long_preamble_1_after_FFT
    INPUT_STREAM_PL1_Freq:
    for (int i = 0; i < N_FFT; i++) {
        axis_data input_data1 = PL1_Freq_stream.read();
        PL1_Freq[i] = input_data1.data;
    }

    // Read input stream - Long_preamble_2_after_FFT
    INPUT_STREAM_PL2_Freq:
    for (int i = 0; i < N_FFT; i++) {
        axis_data input_data2 = PL2_Freq_stream.read();
        PL2_Freq[i] = input_data2.data;
    }

    // Read input stream - Long_preamble_2_after_FFT
    INPUT_STREAM_H_est:
    for (int i = 0; i < N_FFT; i++) {
        axis_data input_data3 = H_est_stream.read();
        H_est[i] = input_data3.data;
    }

    // FFT Shift Operations
    fftshift_opt(PL1_Freq);
    fftshift_opt(PL2_Freq);

    // safe division precaution
    REPLACE_ZEROES:
    for (int i = 0; i < N_FFT; ++i) {
#pragma HLS PIPELINE
        float real = H_est[i].real();
        float imag = H_est[i].imag();

        if (real == 0.0f)
            real = THRESHOLD;
        if (imag == 0.0f)
            imag = THRESHOLD;

        H_est[i] = std::complex<float>(real, imag);
    }

    // Equalize the frequency-domain data and Insert PL freq signals
    SPLIT_PL_Freq:
    for (int i = 0; i < N_FFT; i++) {
#pragma HLS PIPELINE
        PL_FreqEq_Comb[i] = PL1_Freq[i];
        PL_FreqEq_Comb[N_FFT + i] = PL2_Freq[i];
    }
    EQUALIZE:
    for (int i = 0; i < N_FFT; i++) {
#pragma HLS PIPELINE
        PL_FreqEq_Comb[2 * N_FFT + i] = PL1_Freq[i] / H_est[i];
        PL_FreqEq_Comb[3 * N_FFT + i] = PL2_Freq[i] / H_est[i];
    }

```

```

    }

    // Write output stream
    OUTPUT_STREAM_LOOP:
    for (int i = 0; i < 4*N_FFT; i++) {
#pragma HLS PIPELINE
        axis_data output_data;
        output_data.data = PL_FreqEq_Comb[i];
        output_data.last = (i == 4*N_FFT - 1) ? 1 : 0;
        outstream.write(output_data);
    }
}
#endif

```

### 9.3 Handling Division by Zero in OneTap Part 2 IP

In the OneTap Part 2 IP block, the output values could potentially become  $-\infty$  or  $\infty$  due to division by zero, particularly when channel estimation yields zero-valued  $H_{\text{est}}$  coefficients. These infinite outputs are problematic for the AXI4-Stream interface, which cannot transmit non-finite numerical values such as  $\infty$  or NaN.

To address this issue while preserving the sign of the result (which is important for downstream processing in the processor), we implement a logic inside the OneTap Part 2 IP that replaces zero-valued  $H_{\text{est}}$  with a small threshold value  $\epsilon = 0.001$ . This ensures the division produces a finite, though very large, numerical output which retains the sign information.

- This approach prevents invalid values from being transmitted over the stream interface.
- The large-magnitude output values act as markers for these exceptional conditions.
- On the processor side, these marked outputs can be easily identified by their unusually high magnitude and appropriately handled or replaced with values from the PS (Processing System) output.

This method provides a logical and safe way to avoid undefined behavior while preserving critical sign information, enabling the processor to reverse or adjust these values effectively during post-processing.



### 9.3.1 Unoptimized

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.685	1.25

(a) Timing (ns): Summary

Latency		Interval		Pipeline Type
min	max	min	max	
3592	3592	3592	3592	none

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	63	63	1	—	—	64	no
Loop 2	63	63	1	—	—	64	no
Loop 3	63	63	1	—	—	64	no
Loop 4	255	255	1	—	—	256	no
INPUT_STREAM_PL1_Freq	64	64	1	—	—	64	no
INPUT_STREAM_PL2_Freq	64	64	1	—	—	64	no
INPUT_STREAM_H_est	64	64	1	—	—	64	no
FFTSHIFT_LOOP	64	64	2	—	—	32	no
FFTSHIFT_LOOP	64	64	2	—	—	32	no
REPLACE_ZEROES	256	256	4	—	—	64	no
EQUALIZE_N_INSERT	1792	1792	28	—	—	64	no
OUTPUT_STREAM_LOOP	768	768	3	—	—	256	no

(c) Loop Summary

Table 49: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	547	-
FIFO	-	-	-	-	-
Instance	-	40	5631	9614	-
Memory	14	-	0	0	0
Multiplexer	-	-	-	1040	-
Register	-	-	1643	-	-
<b>Total</b>	14	40	7274	11201	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	5	18	6	21	0

Table 50: Resource Utilization Summary

### 9.3.2 Optimized

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
lap_clk	10.00	8.685	1.25	1195	1195	1195	1195	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Instance	Module	Latency		Interval		Pipeline Type
		min	max	min	max	
gpp_fftshift_opt_fu_634	fftshift_opt	66	66	66	66	none
gpp_fftshift_opt_fu_640	fftshift_opt	66	66	66	66	none

(c) Instance Details

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
Loop 1	63	63	1	—	—	64	no
Loop 2	63	63	1	—	—	64	no
Loop 3	63	63	1	—	—	64	no
Loop 4	255	255	1	—	—	256	no
INPUT_STREAM_PL1_Freq	64	64	1	—	—	64	no
INPUT_STREAM_PL2_Freq	64	64	1	—	—	64	no
INPUT_STREAM_H_est	64	64	1	—	—	64	no
REPLACE_ZEROES	66	66	4	1	1	64	yes
SPLIT_PL_Freq	64	64	2	1	1	64	yes
EQUALIZE	90	90	28	1	1	64	yes
OUTPUT_STREAM_LOOP	257	257	3	1	1	256	yes

(d) Loop Summary

Table 51: Performance Metrics for IP3

Name	BRAM_18K	DSPM8E	FF	LUT	URAM
DSP	—	—	—	—	—
Expression	—	—	0	539	—
FIFO	—	—	—	—	—
Instance	—	40	5711	9872	—
Memory	16	—	0	0	0
Multiplexer	—	—	—	1151	—
Register	0	—	1896	160	—
<b>Total</b>	16	40	7607	11722	0
<b>Available</b>	280	220	106400	53200	0
<b>Utilization (%)</b>	5	18	7	22	0

Table 52: Resource Utilization Summary

As seen earlier, we've limited the use of UNROLL, since the arrays we're dealing here are big on size. It's better to use PIPELINE.

### 9.3.3 Word Length Optimized

```
#ifdef solutionWLOpt
typedef ap_fixed<18,5> fixed24_t;
typedef std::complex<fixed24_t> complex_fixed_t;
#endif
```

Clock	Target	Estimated	Uncertainty	Latency		Interval		Pipeline Type
				min	max	min	max	
ap_clk	10.00	9.083	1.25	1,273	1,273	1,273	1,273	none

(a) Timing (ns): Summary

(b) Latency (clock cycles): Summary

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
INPUT_STREAM_PL1_Freq	66	66	4	1	1	64	yes
INPUT_STREAM_PL2_Freq	66	66	4	1	1	64	yes
INPUT_STREAM_H_est	66	66	4	1	1	64	yes
Loop 4	64	64	2	—	—	32	no
Loop 5	64	64	2	—	—	32	no
Loop 6	512	512	8	—	—	64	no
Loop 7	163	163	38	2	1	64	yes
OUTPUT_STREAM_LOOP	259	259	5	1	1	256	yes

(c) Loop Summary

Table 53: Performance Metrics

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	10	-	-	-
Expression	-	-	160	6889	-
FIFO	-	-	-	-	-
Instance	-	0	8952	7308	-
Memory	8	-	0	0	0
Multiplexer	-	-	-	206	-
Register	6	-	4122	483	-
<b>Total</b>	14	10	13234	14886	0
<b>Available</b>	1090	900	437200	218600	0
<b>Utilization (%)</b>	1	1	3	6	0

Table 54: Resource Utilization Summary

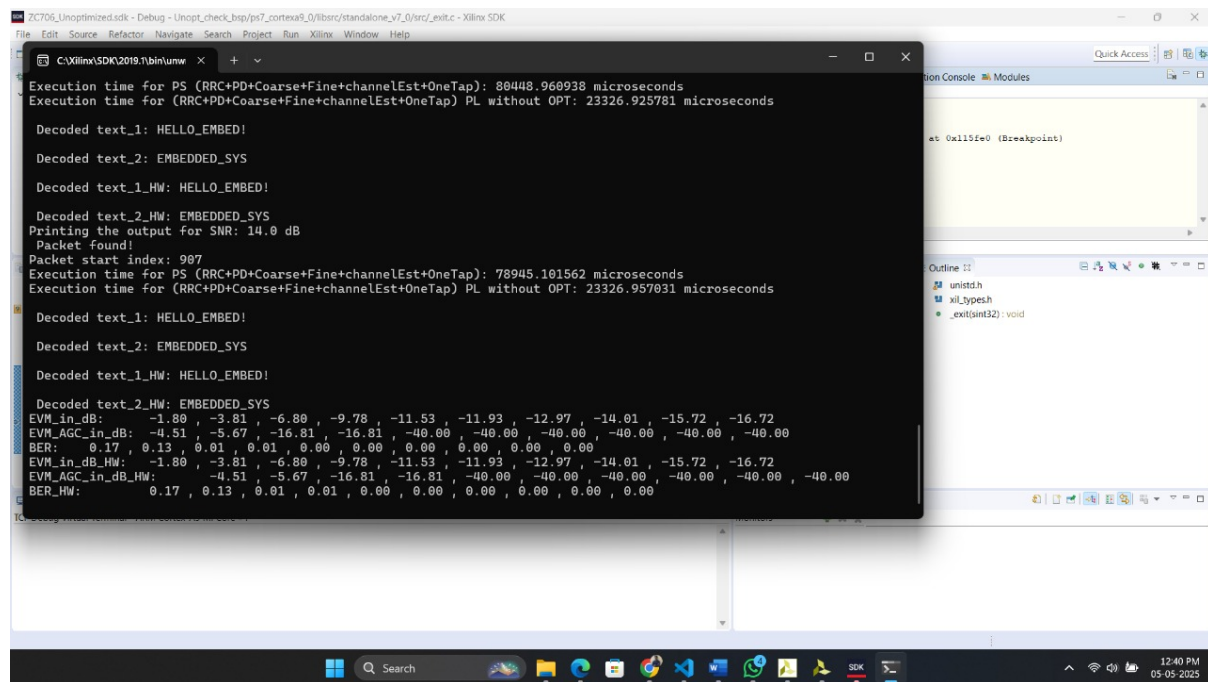
- Word-length optimization reduces DSP usage from 18% (40 units) to 1% (10 units) and LUTs from 22% to 6%, by minimizing bit-widths in arithmetic operations.

- Memory resources (BRAMs) drop from 16 to 8 blocks, demonstrating efficient data representation with optimized precision.

## 10 Results and Comparison

### 10.1 Non Optimized

Each IPs of the Receiver part has been implemented and functionality has been verified. We can see that without any optimizations, just by implementing them in PL the execution time has been reduced by five fold as expected.



```
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 88448.960938 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without OPT: 23326.925781 microseconds

Decoded text_1: HELLO_EMBED!
Decoded text_2: EMBEDDED_SYS
Decoded text_1_HW: HELLO_EMBED!
Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 14.0 dB
Packet found!
Packet start index: 997
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 78945.181562 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without OPT: 23326.957831 microseconds

Decoded text_1: HELLO_EMBED!
Decoded text_2: EMBEDDED_SYS
Decoded text_1_HW: HELLO_EMBED!
Decoded text_2_HW: EMBEDDED_SYS
EVM_in_dB: -1.80, -3.81, -6.80, -9.78, -11.53, -11.93, -12.97, -14.01, -15.72, -16.72
EVM_AGC_in_dB: -4.51, -5.67, -16.81, -16.81, -40.00, -40.00, -40.00, -40.00, -40.00, -40.00
BER: 0.17, 0.13, 0.01, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00
EVM_in_dB_HW: -1.80, -3.81, -6.80, -9.78, -11.53, -11.93, -12.97, -14.01, -15.72, -16.72
EVM_AGC_in_dB_HW: -4.51, -5.67, -16.81, -16.81, -40.00, -40.00, -40.00, -40.00, -40.00, -40.00
BER_HW: 0.17, 0.13, 0.01, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00
```

### 10.2 Optimized

- In FPGA design using High-Level Synthesis (HLS), **pragmas** serve as compiler directives that guide the translation of high-level C/C++ code into optimized hardware. Key optimization pragmas include:
  - #pragma HLS PIPELINE
    - \* Initiates new loop iterations every clock cycle, improving throughput.
  - #pragma HLS UNROLL
    - \* Replicates loop bodies for parallel execution.
  - #pragma HLS ARRAY\_PARTITION
    - \* Enables simultaneous array access by breaking memory into smaller parts.
  - #pragma HLS DATAFLOW
    - \* Facilitates task-level pipelining, allowing multiple functions to run concurrently.
  - #pragma HLS INLINE

- \* Eliminates function call overhead and enables further optimizations.
- Resource directives:
  - \* #pragma HLS RESOURCE and #pragma HLS ALLOCATION
    - Control how operations and variables are mapped to FPGA hardware (e.g., using LUTs or DSPs).
- Together, these pragmas enable designers to fine-tune performance, area, and timing to meet application-specific constraints.

Using appropriate pragmas, we've optimized the IPs as discussed earlier. We could see the boost in performance almost by **20x**.

The screenshot shows a Xilinx IDE interface. The main terminal window displays the following output:

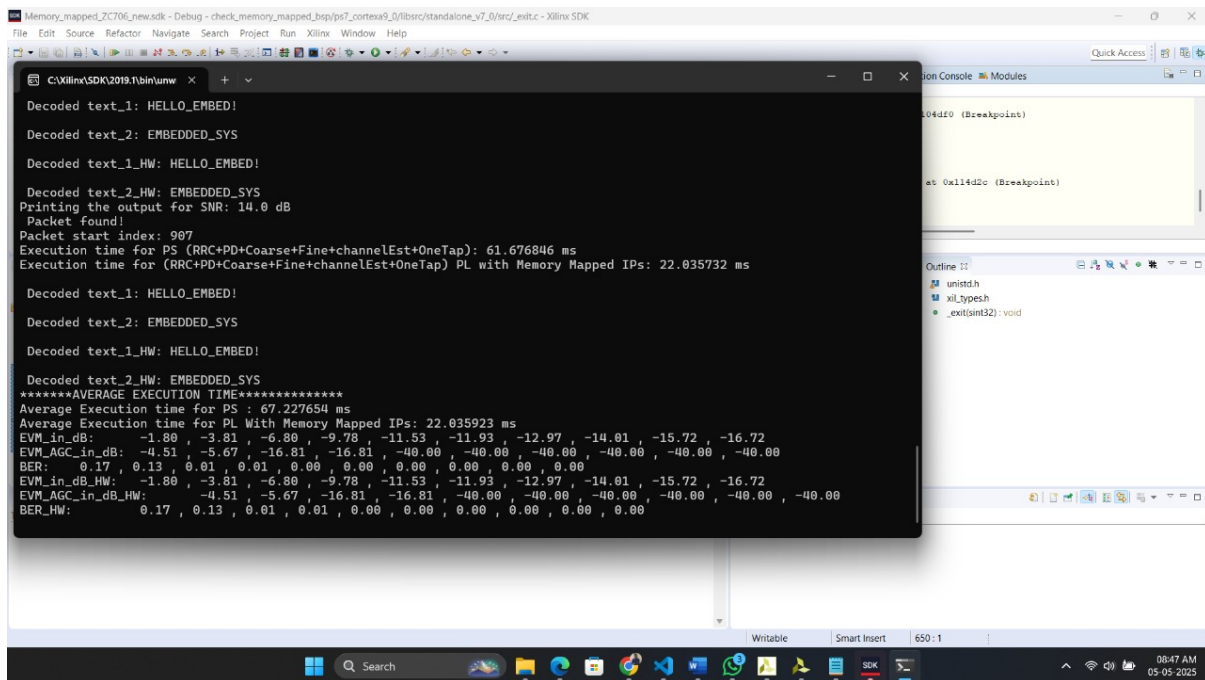
```
Decoded text_1: HELLO_EMBED!
Decoded text_2: EMBEDDED_SYS
Decoded text_1_HW: HELLO_EMBED!
Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 14.0 dB
Packet found!
Packet start index: 997
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 81.316681 ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without OPT: 3.717757 ms

Decoded text_1: HELLO_EMBED!
Decoded text_2: EMBEDDED_SYS
Decoded text_1_HW: HELLO_EMBED!
Decoded text_2_HW: EMBEDDED_SYS
*****AVERAGE EXECUTION TIME*****
Average Execution time for PS : 66.621765 ms
Average Execution time for PL With Optimization: 3.717730 ms
EVM_in_dB: -1.80, -3.81, -6.80, -9.78, -11.53, -11.93, -12.97, -14.01, -15.72, -16.72
EVM_AGC_in_dB: -4.51, -5.67, -16.81, -16.81, -40.00, -40.00, -40.00, -40.00, -40.00, -40.00
BER: 0.17, 0.13, 0.01, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00
EVM_in_dB_HW: -1.80, -3.81, -6.80, -9.78, -11.53, -11.93, -12.97, -14.01, -15.72, -16.72
EVM_AGC_in_dB_HW: -4.51, -5.67, -16.81, -16.81, -40.00, -40.00, -40.00, -40.00, -40.00, -40.00
BER_HW: 0.17, 0.13, 0.01, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00
```

The console window on the right shows breakpoints at 0x104c00 and 0x1160a0. The Outline pane shows files like unistd.h, xil\_types.h, and \_exit(sim32). The status bar at the bottom indicates 'Writable', 'Smart Insert', and '651:1'.

## 10.3 Memory Mapped

Moving on with further optimizations, proceeding without DMA we directly use Memory Mapped Interfaced IPs to reduce the polling time, the double transaction of DMA, and it's hardware resources. Since it's memory mapped with the PS (ZYNQ Processor) we got the drivers Set, Start, Done for setting the real and imaginary parts of the complex variable.



```
Decoded text_1: HELLO_EMBED!  
Decoded text_2: EMBEDDED_SYS  
Decoded text_1_HW: HELLO_EMBED!  
Decoded text_2_HW: EMBEDDED_SYS  
Printing the output for SNR: 14.0 dB  
Packet found!  
Packet start index: 907  
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 61.676846 ms  
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with Memory Mapped IPs: 22.035732 ms  
Decoded text_1: HELLO_EMBED!  
Decoded text_2: EMBEDDED_SYS  
Decoded text_1_HW: HELLO_EMBED!  
Decoded text_2_HW: EMBEDDED_SYS  
*****AVERAGE EXECUTION TIME*****  
Average Execution time for PS : 67.227654 ms  
Average Execution time for PL With Memory Mapped IPs: 22.035923 ms  
EVM_in_dB: -1.80, -3.81, -6.80, -9.78, -11.53, -11.93, -12.97, -14.01, -15.72, -16.72  
EVM_AGC_in_dB: -4.51, -5.67, -16.81, -16.81, -40.00, -40.00, -40.00, -40.00, -40.00, -40.00  
BER: 0.17, 0.13, 0.01, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00  
EVM_in_dB_HW: -1.80, -3.81, -6.80, -9.78, -11.53, -11.93, -12.97, -14.01, -15.72, -16.72  
EVM_AGC_in_dB_HW: -4.51, -5.67, -16.81, -16.81, -40.00, -40.00, -40.00, -40.00, -40.00, -40.00  
BER_HW: 0.17, 0.13, 0.01, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00
```

## 10.4 Interrupt

Now we've implemented the Receiver using interrupts rather than polling mode in the DMA. We can see that the execution time has sharply reduced.

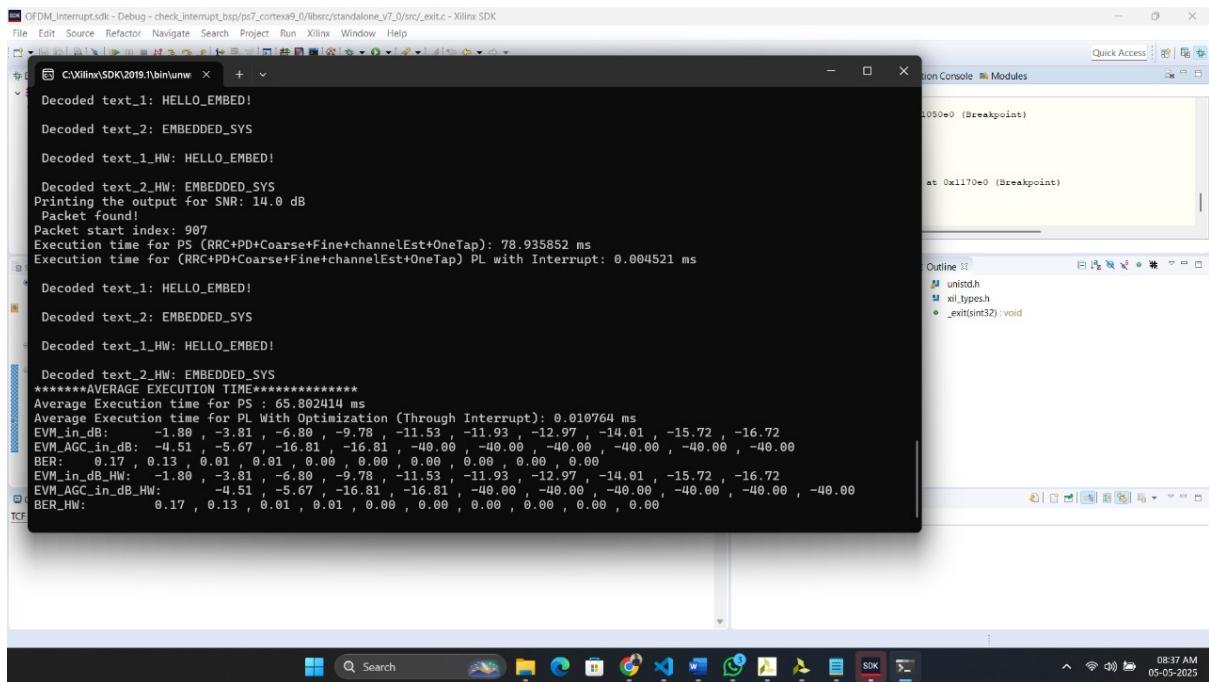
Using **interrupts** instead of **polling** provides multiple performance and efficiency benefits. Firstly, interrupts improve **CPU efficiency** by allowing the processor to perform other operations and only respond when required. In contrast, polling continuously consumes CPU cycles, even when no event has occurred.

Secondly, interrupts contribute to **lower power consumption**, as systems can enter low-power or sleep states and wake only when necessary. Polling systems must remain active, consuming more energy overall.

Thirdly, interrupts offer **faster response times**, reacting immediately to events and supporting real-time system requirements. Polling may delay event detection, depending on the polling interval.

Additionally, interrupts enable better **multitasking**, as the CPU can manage multiple asynchronous events without being blocked by constant status checks. This allows for more scalable and responsive system design.

Lastly, interrupt-driven code is often more **modular and maintainable**, since it follows an event-driven approach rather than relying on loops and state machines typical in polling-based designs.



## 10.5 Word Length Optimization

We select the minimum number of bits required to represent the data without significantly reducing the accuracy or performance.

Smaller word lengths consume fewer hardware resources, viz critical in FPGAs. It also leads to energy efficiency. Sometimes performance too.

But, the issues of overflow or underflow, precision loss should be handled with care.

```

EVM_in_dB: -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 , -12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB: -4.51 , -5.67 , -16.81 , -16.81 , -40.00 , -40.00 , -40.00 , -40.00 , -40.00 , -40.00
BER: 0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00
EVM_in_dB_HW: 15.95 , 9.83 , 14.45 , 15.70 , 13.02 , 14.10 , 13.32 , 14.37 , 15.11 , 12.82
EVM_AGC_in_dB_HW: 1.85 , 1.42 , 1.58 , 1.87 , 2.43 , 1.65 , 2.01 , 2.07 , 2.15 , 1.35
BER_HW: 0.47 , 0.45 , 0.45 , 0.47 , 0.44 , 0.46 , 0.44 , 0.47 , 0.45 , 0.44

```

Figure 5: BER Increase Due to Word Length Optimization

We verified every IPs with the tolerance level of around 0.9 (as directed). And, hence the above Bit Error Rate, which is high compared to other methods here. Hence, we could see the discrepancy in the decoding the text.



## 10.6 Result Graphs

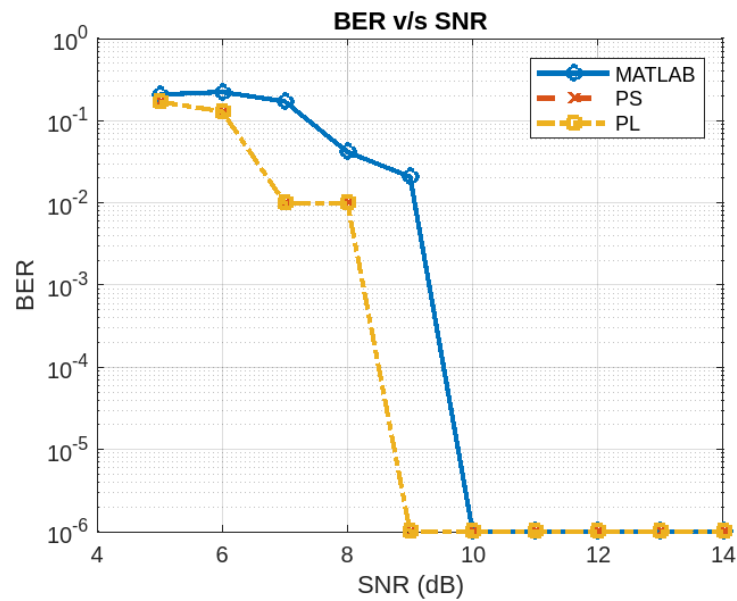


Figure 6: BER vs SNR

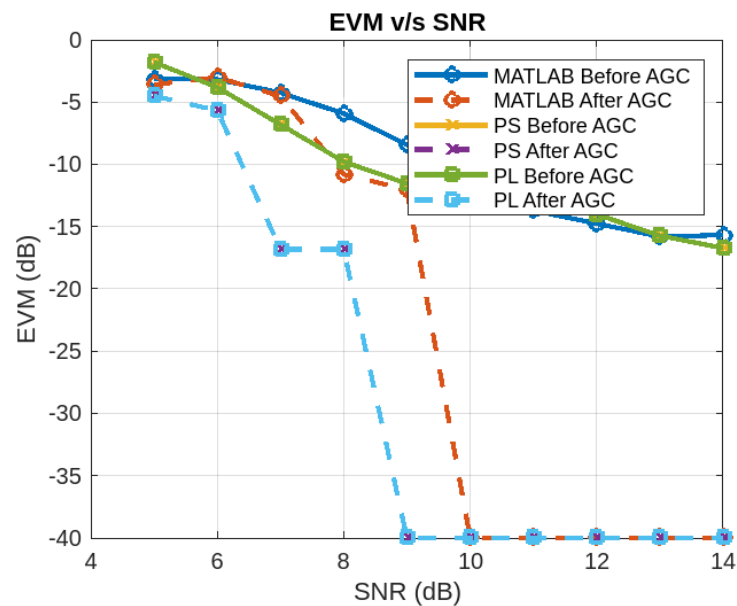


Figure 7: EVM vs SNR

We can observe that PL exactly matches with the PS. When compared with the MATLAB outputs we could see better results. The proper reception has been observed at lesser SNR in PS and PL.

### 10.6.1 Resource Utilization of Receiver Part

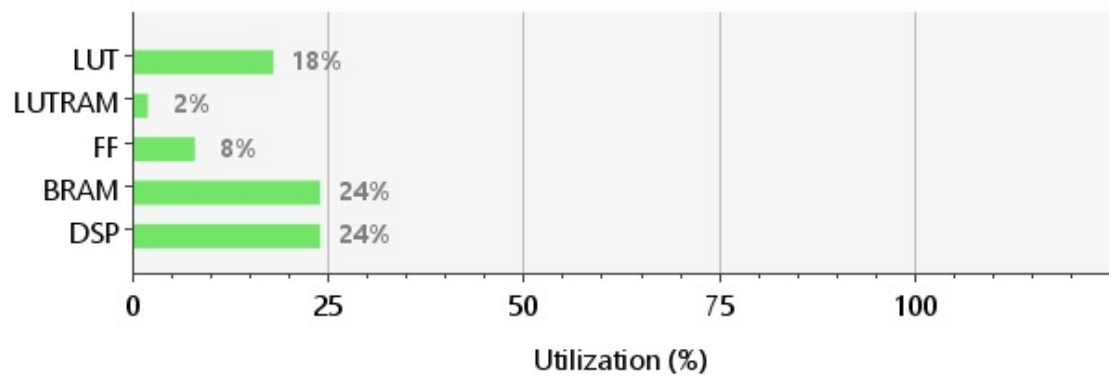


Figure 8: Before Optimization

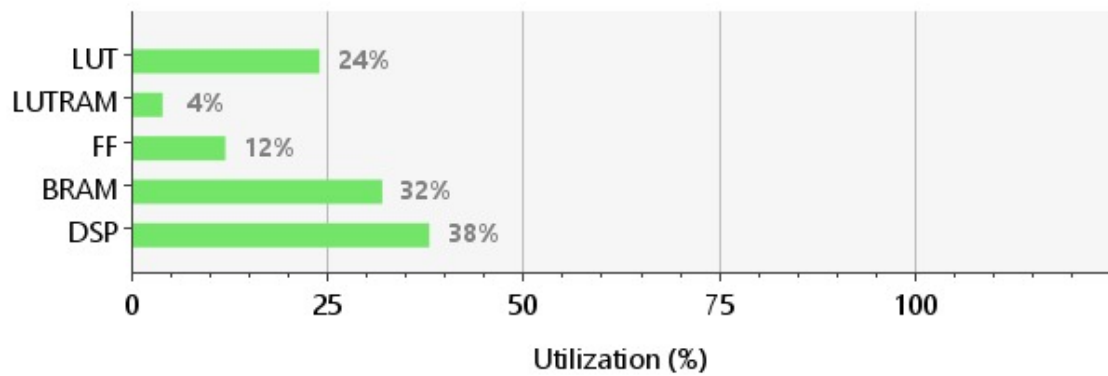


Figure 9: After Optimization

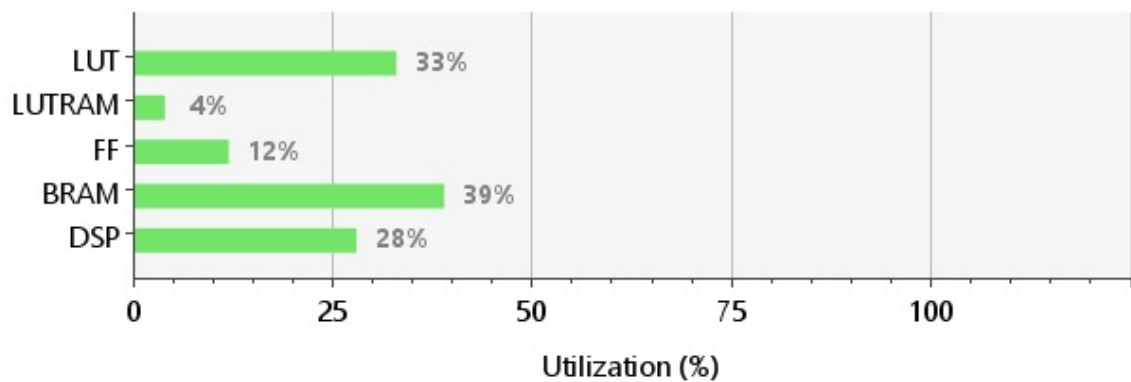


Figure 10: After Word Length Optimization

# APPENDIX A: JTAG Outputs

## Non Optimized

```
JTAG-based Hyperterminal.
Connected to JTAG-based Hyperterminal over TCP port : 49385
(using socket : sock620)
Help :
Terminal requirements :
  (i) Processor's STDOUT is redirected to the ARM DCC/MDM UART
  (ii) Processor's STDIN is redirected to the ARM DCC/MDM UART.
      Then, text input from this console will be sent to DCC/MDM's
      UART port.
NOTE: This is a line-buffered console and you have to press "Enter"
      to send a string of characters to DCC/MDM.

Starting...!
DMA Initialization successful.
Printing the output for SNR: 5.0 dB
Packet found!
Packet start index: 922
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    30316.808594 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 23326.468750 microseconds

Decoded text_1: H 00_EMBFx

Decoded text_2:  MBEwDuD_SY  ^

Decoded text_1_HW: H 00_EMBFx

Decoded text_2_HW:  MBEwDuD_SY  ^
Printing the output for SNR: 6.0 dB
Packet found!
Packet start index: 914
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    33232.132812 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 23326.937500 microseconds

Decoded text_1: 8ELL_E}BED-

Decoded text_2:  E    r E    D DSYS

Decoded text_1_HW: 8ELL_E}BED-

Decoded text_2_HW:  E    r E    D DSYS
Printing the output for SNR: 7.0 dB
Packet found!
Packet start index: 912
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    79529.765625 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 23327.105469 microseconds
```

```

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 8.0 dB
Packet found!
Packet start index: 910
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    78385.554688 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
    OPT: 23326.927734 microseconds

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 9.0 dB
Packet found!
Packet start index: 909
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    78015.132812 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
    OPT: 23326.970703 microseconds

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 10.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    60102.222656 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
    OPT: 23326.970703 microseconds

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 11.0 dB
Packet found!

```

```

Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    63958.343750 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
    OPT: 23326.916016 microseconds

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 12.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    62026.011719 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
    OPT: 23327.095703 microseconds

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 13.0 dB
Packet found!
Packet start index: 907
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    78225.054688 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
    OPT: 23327.033203 microseconds

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 14.0 dB
Packet found!
Packet start index: 907
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap):
    81014.101562 microseconds
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
    OPT: 23326.765625 microseconds

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

```

```
Decoded text_1_HW: HELLO_EMBED!
```

```
Decoded text_2_HW: EMBEDDED_SYS
```

```
EVM_in_dB:      -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,  
               -12.97 , -14.01 , -15.72 , -16.72
```

```
EVM_AGC_in_dB:  -4.51 , -5.67 , -16.81 , -16.81 , -40.00 , -40.00 ,  
               -40.00 , -40.00 , -40.00 , -40.00
```

```
BER:           0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00 ,  
               0.00
```

```
EVM_in_dB_HW:   -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,  
               -12.97 , -14.01 , -15.72 , -16.72
```

```
EVM_AGC_in_dB_HW: -4.51 , -5.67 , -16.81 , -16.81 , -40.00 ,  
               -40.00 , -40.00 , -40.00 , -40.00 , -40.00
```

```
BER_HW:         0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 ,  
               0.00 , 0.00
```

## Optimized

```
JTAG-based Hyperterminal.
```

```
Connected to JTAG-based Hyperterminal over TCP port : 56818
```

```
(using socket : sock620)
```

```
Help :
```

```
Terminal requirements :
```

```
(i) Processor's STDOUT is redirected to the ARM DCC/MDM UART
```

```
(ii) Processor's STDIN is redirected to the ARM DCC/MDM UART.
```

```
Then, text input from this console will be sent to DCC/MDM's  
UART port.
```

```
NOTE: This is a line-buffered console and you have to press "Enter"  
to send a string of characters to DCC/MDM.
```

```
Starting...!
```

```
DMA Initialization successful.
```

```
Printing the output for SNR: 5.0 dB
```

```
Packet found!
```

```
Packet start index: 922
```

```
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 47.867588  
ms
```

```
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without  
OPT: 3.717481 ms
```

```
Decoded text_1: H 00_EMBFx
```

```
Decoded text_2:  MBEwDuD_SY ^
```

```
Decoded text_1_HW: H 00_EMBFx
```

```
Decoded text_2_HW:  MBEwDuD_SY ^
```

```
Printing the output for SNR: 6.0 dB
```

```
Packet found!
```

```
Packet start index: 914
```

```
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 35.276539  
ms
```

```
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without  
OPT: 3.717859 ms
```

```

Decoded text_1: 8ELL_E}BED-

Decoded text_2: E    r E    D DSYS

Decoded text_1_HW: 8ELL_E}BED-

Decoded text_2_HW: E    r E    D DSYS
Printing the output for SNR: 7.0 dB
Packet found!
Packet start index: 912
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 81.199570
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 3.717835 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EBEDDED_SYS
Printing the output for SNR: 8.0 dB
Packet found!
Packet start index: 910
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 65.453102
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 3.717820 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EBEDDED_SYS
Printing the output for SNR: 9.0 dB
Packet found!
Packet start index: 909
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 65.434952
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 3.717784 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 10.0 dB
Packet found!

```

```

Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 65.487434
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 3.717682 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 11.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 61.226780
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 3.717802 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 12.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 71.139900
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 3.717835 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 13.0 dB
Packet found!
Packet start index: 907
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 78.827263
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 3.717814 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

```



```

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 14.0 dB
Packet found!
Packet start index: 907
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 62.674328
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL without
OPT: 3.717724 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
*****AVERAGE EXECUTION TIME*****
Average Execution time for PS : 63.458740 ms
Average Execution time for PL With Optimization: 3.717763 ms
EVM_in_dB:      -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,
-12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB:  -4.51 , -5.67 , -16.81 , -16.81 , -40.00 , -40.00 ,
-40.00 , -40.00 , -40.00 , -40.00
BER:           0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00 ,
0.00
EVM_in_dB_HW:   -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,
-12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB_HW: -4.51 , -5.67 , -16.81 , -16.81 , -40.00 ,
-40.00 , -40.00 , -40.00 , -40.00 , -40.00
BER_HW:         0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00 ,
0.00 , 0.00

```

## Memory Mapped

```

JTAG-based Hyperterminal.
Connected to JTAG-based Hyperterminal over TCP port : 59333
(using socket : sock616)
Help :
Terminal requirements :
(i) Processor's STDOUT is redirected to the ARM DCC/MDM UART
(ii) Processor's STDIN is redirected to the ARM DCC/MDM UART.
    Then, text input from this console will be sent to DCC/MDM's
    UART port.
NOTE: This is a line-buffered console and you have to press "Enter"
      to send a string of characters to DCC/MDM.

Starting...!
RRC Memory Mapped IP Initialization successful.
One Tap Memory Mapped IP Initialization successful.
Printing the output for SNR: 5.0 dB
Packet found!
Packet start index: 922

```

```

Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 46.689434
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038712 ms

Decoded text_1: H 00_EMBFx

Decoded text_2:  MBEwDuD_SY  ^

Decoded text_1_HW: H 00_EMBFx

Decoded text_2_HW:  MBEwDuD_SY  ^
Printing the output for SNR: 6.0 dB
Packet found!
Packet start index: 914
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 47.725281
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038443 ms

Decoded text_1: 8ELL_E}BED-

Decoded text_2:  E    r E    D DSYS

Decoded text_1_HW: 8ELL_E}BED-

Decoded text_2_HW:  E    r E    D DSYS
Printing the output for SNR: 7.0 dB
Packet found!
Packet start index: 912
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 77.302216
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038177 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EBEDDED_SYS
Printing the output for SNR: 8.0 dB
Packet found!
Packet start index: 910
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 63.432014
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038662 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

```

```

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 9.0 dB
Packet found!
Packet start index: 909
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 63.078609
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038548 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 10.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 76.294624
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038261 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 11.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 79.003181
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038502 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 12.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 77.678391
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038324 ms

```

```

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 13.0 dB
Packet found!
Packet start index: 907
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 64.164833
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038998 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 14.0 dB
Packet found!
Packet start index: 907
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 61.593399
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Memory Mapped IPs: 22.038214 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
*****AVERAGE EXECUTION TIME*****
Average Execution time for PS : 65.696198 ms
Average Execution time for PL With Memory Mapped IPs: 22.038485 ms
EVM_in_dB:      -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,
               -12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB:  -4.51 , -5.67 , -16.81 , -16.81 , -40.00 , -40.00 ,
               -40.00 , -40.00 , -40.00 , -40.00
BER:           0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00 ,
               0.00
EVM_in_dB_HW:   -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,
               -12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB_HW: -4.51 , -5.67 , -16.81 , -16.81 , -40.00 ,
               -40.00 , -40.00 , -40.00 , -40.00 , -40.00
BER_HW:         0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 ,
               0.00 , 0.00

```

## Interrupt

```

JTAG-based Hyperterminal.
Connected to JTAG-based Hyperterminal over TCP port : 51064
(using socket : sock612)
Help :
Terminal requirements :
  (i) Processor's STDOUT is redirected to the ARM DCC/MDM UART
  (ii) Processor's STDIN is redirected to the ARM DCC/MDM UART.
      Then, text input from this console will be sent to DCC/MDM's
      UART port.
NOTE: This is a line-buffered console and you have to press "Enter"
      to send a string of characters to DCC/MDM.

Starting...!
DMA Initialization successful.
Printing the output for SNR: 5.0 dB
Packet found!
Packet start index: 922
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 47.981743
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.067269 ms

Decoded text_1: H OO_EMBFx

Decoded text_2:  MBEwDuD_SY ^

Decoded text_1_HW: H OO_EMBFx

Decoded text_2_HW:  MBEwDuD_SY ^
Printing the output for SNR: 6.0 dB
Packet found!
Packet start index: 914
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 33.284069
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004497 ms

Decoded text_1: 8ELL_E}BED-

Decoded text_2:  E    r E    D DSYS

Decoded text_1_HW: 8ELL_E}BED-

Decoded text_2_HW:  E    r E    D DSYS
Printing the output for SNR: 7.0 dB
Packet found!
Packet start index: 912
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 80.254967
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004470 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

```

```

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 8.0 dB
Packet found!
Packet start index: 910
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 77.453964
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004482 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 9.0 dB
Packet found!
Packet start index: 909
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 78.379433
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004434 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 10.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 79.390427
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004458 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 11.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 63.404835
ms

```

```

Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004476 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 12.0 dB
Packet found!
Packet start index: 908
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 64.906166
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004467 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 13.0 dB
Packet found!
Packet start index: 907
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 63.699917
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004449 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS
Printing the output for SNR: 14.0 dB
Packet found!
Packet start index: 907
Execution time for PS (RRC+PD+Coarse+Fine+channelEst+OneTap): 64.886238
ms
Execution time for (RRC+PD+Coarse+Fine+channelEst+OneTap) PL with
Interrupt: 0.004425 ms

Decoded text_1: HELLO_EMBED!

Decoded text_2: EMBEDDED_SYS

Decoded text_1_HW: HELLO_EMBED!

Decoded text_2_HW: EMBEDDED_SYS

```

```

*****AVERAGE EXECUTION TIME*****
Average Execution time for PS : 65.364182 ms
Average Execution time for PL With Optimization (Through Interrupt):
0.010743 ms
EVM_in_dB:      -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,
                -12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB:  -4.51 , -5.67 , -16.81 , -16.81 , -40.00 , -40.00 ,
                -40.00 , -40.00 , -40.00 , -40.00
BER:           0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00 ,
                0.00
EVM_in_dB_HW:   -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,
                -12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB_HW: -4.51 , -5.67 , -16.81 , -16.81 , -40.00 ,
                -40.00 , -40.00 , -40.00 , -40.00 , -40.00
BER_HW:         0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 ,
                0.00 , 0.00

```



# APPENDIX B: IP Output Screenshots

## RRC Filter

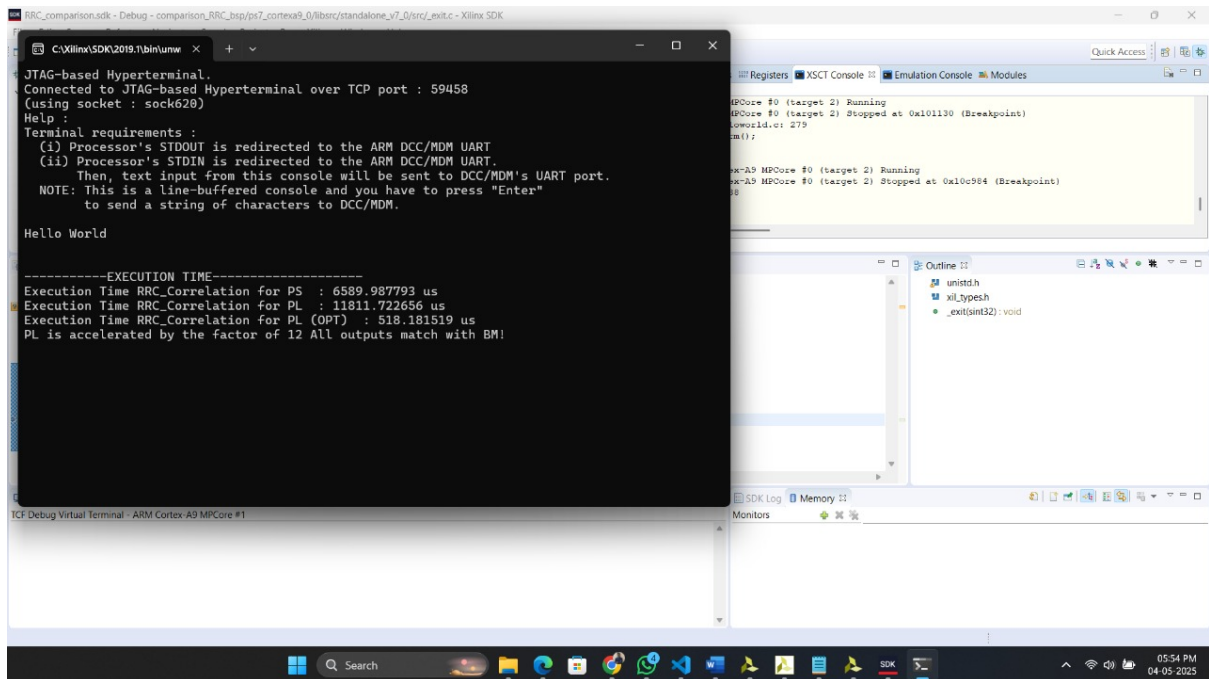


Figure 11: Optimization Verification for RRC

## Packet Detection

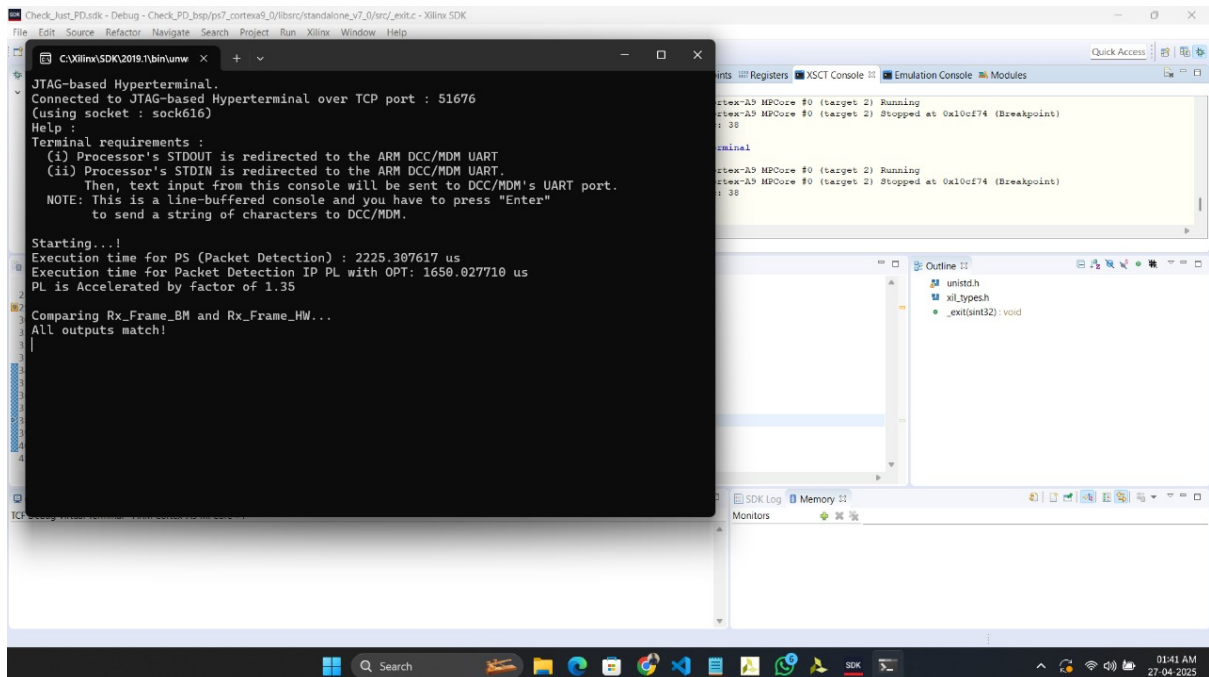


Figure 12: Optimization Verification for PD

## Coarse CFO

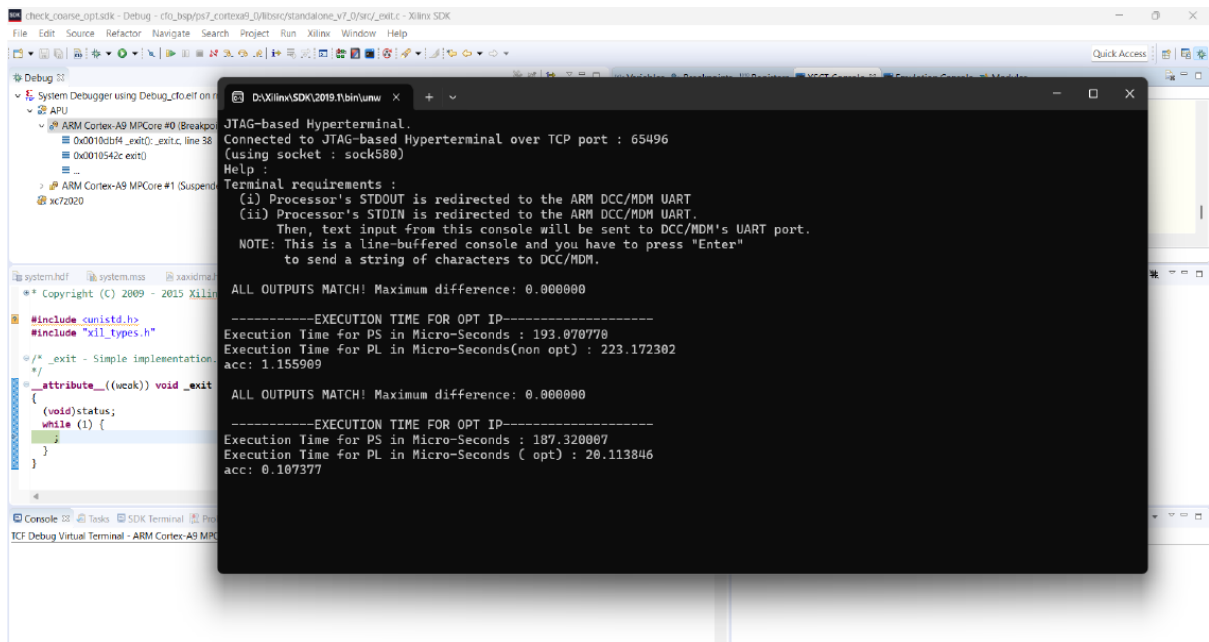


Figure 13: Optimization Verification for Coarse CFO

## Fine CFO

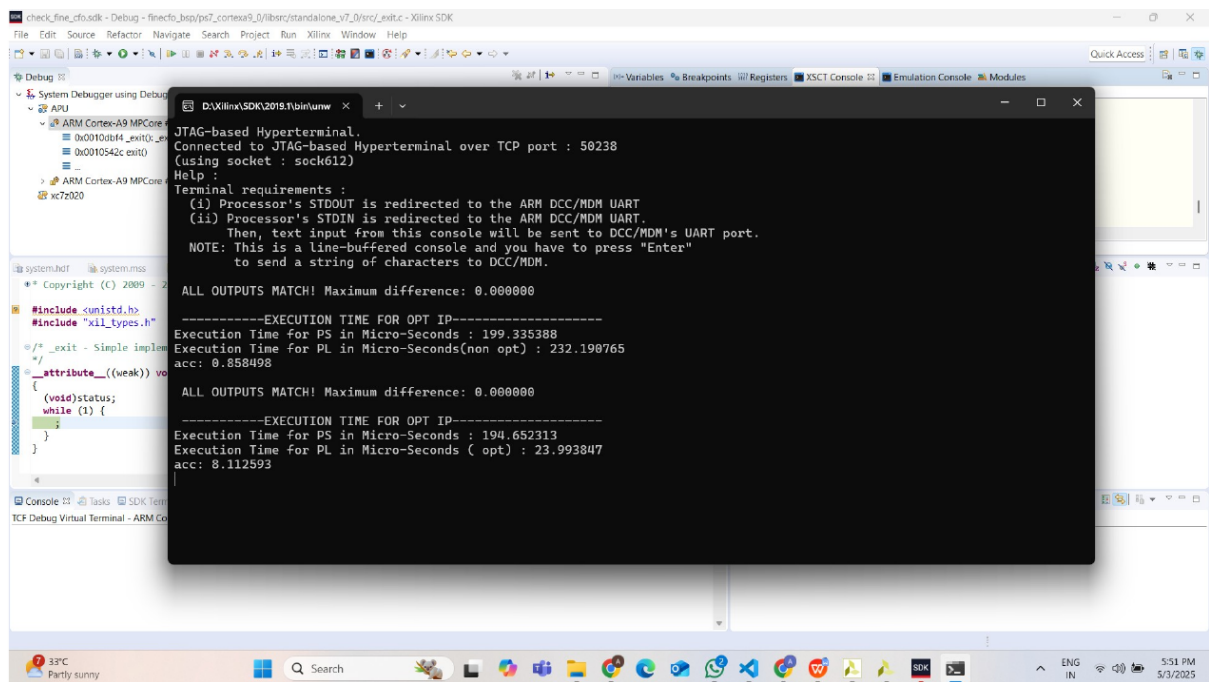


Figure 14: Optimization Verification for Fine CFO

## FFT

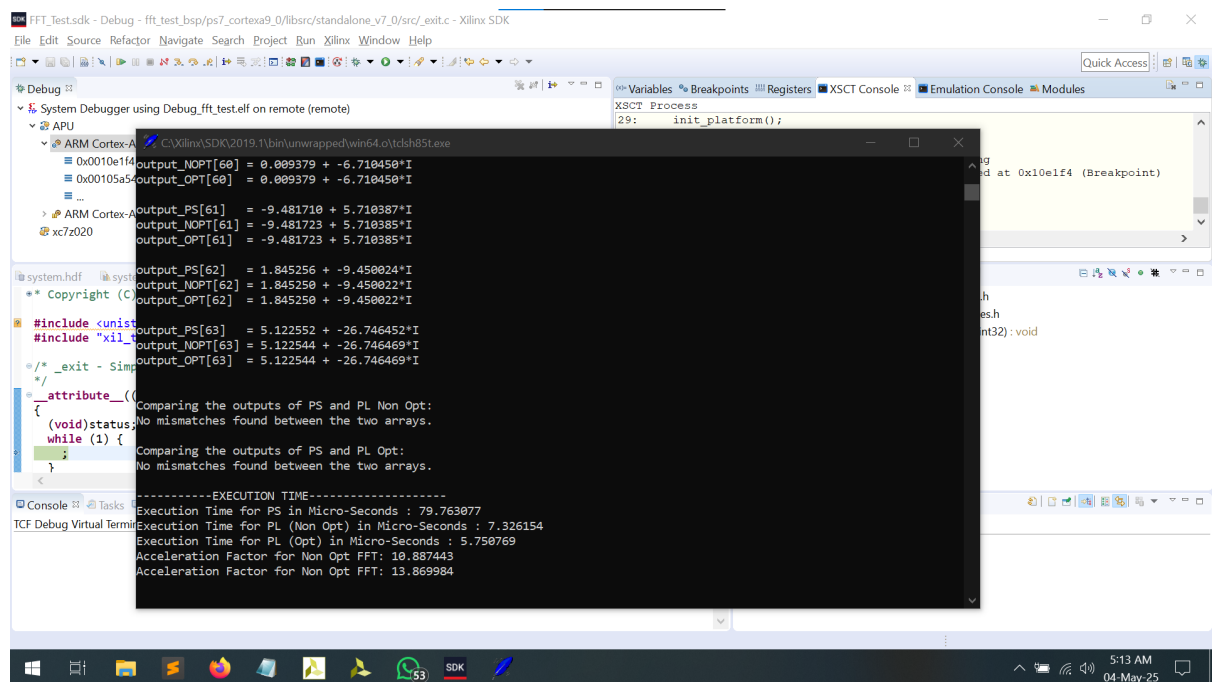


Figure 15: Optimization Verification for FFT

## Channel Estimation and One Tap Equalizer

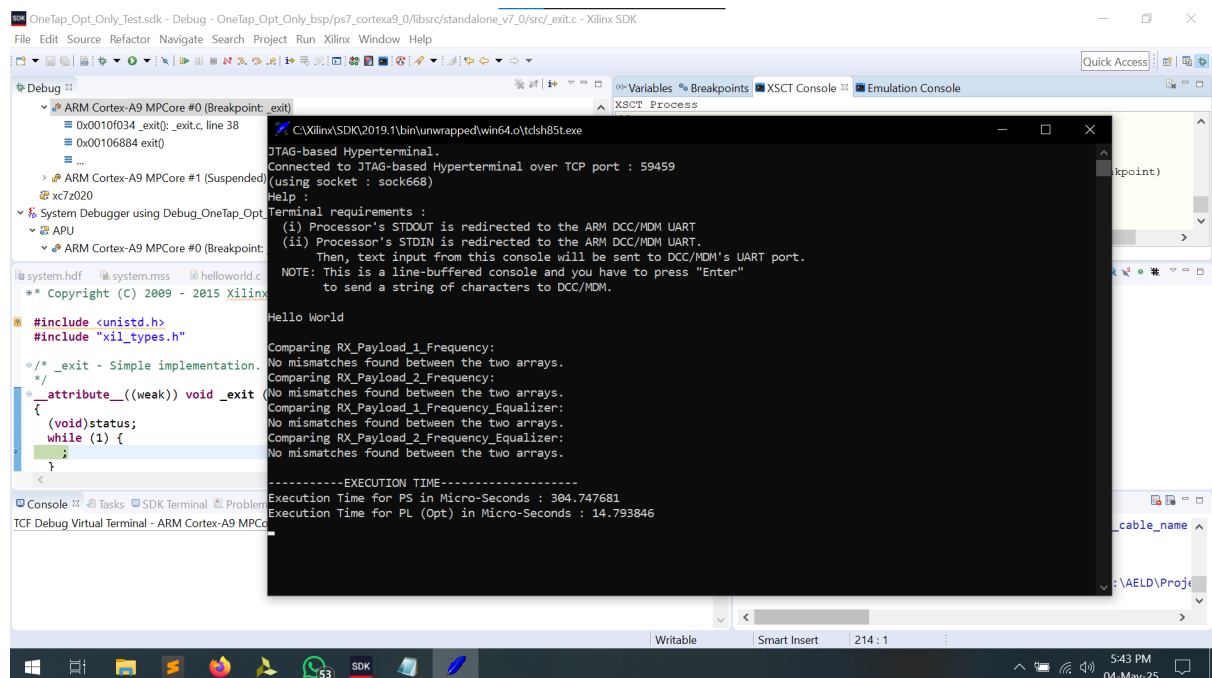


Figure 16: Optimization Verification for Channel Estimation and One Tap Equalizer