ELD A+ PROJECT REPORT

Madhav Maheshwari
2023304

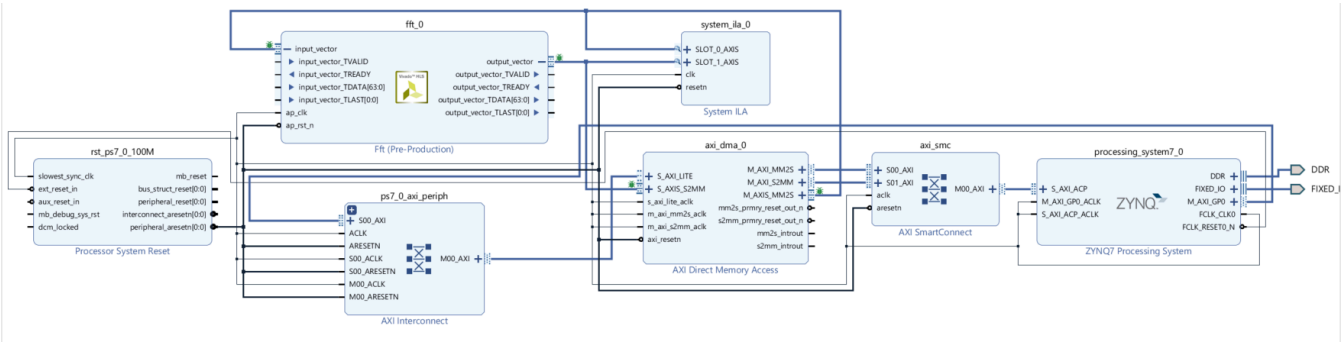
Introduction

This project centers on the design and implementation of an 8-point Fast Fourier Transform (FFT) Intellectual Property (IP) core using Vivado High-Level Synthesis (HLS). The primary objective is to develop a custom 8-point FFT IP, integrate it into a Vivado project, and conduct a detailed comparison between the results produced by the custom IP and the pre-built FFT IP provided by Vivado.

The project involves leveraging the capabilities of Vivado HLS to create a hardware-optimized FFT core, simulating its functionality, and synthesizing it into a design that can be implemented on FPGA hardware. A comprehensive performance evaluation is conducted to assess the accuracy, computational efficiency, and hardware resource utilization of the custom FFT IP compared to the standard Vivado FFT IP.

Block Diagram of Vivado Project

The block diagram of the Vivado project used for the implementation of the custom-made FFT IP is shown below:



The block diagram illustrates the integration of the ZYNQ7 Processing System IP, the AXI Direct Memory Access (DMA) IP, and the custom-designed 8-point FFT IP (referred to as "Fft" in the diagram). The ZYNQ7 Processing System IP utilizes its Accelerator Coherency Port (ACP) to transfer data between the DMA IP and memory efficiently. The ACP port was selected for this purpose because it provides direct access to cache-coherent data, which offers superior performance compared to the high-performance (HP) ports of the ZYNQ IP when handling data

transfers. Furthermore, the Master General-Purpose (GP) port of the ZYNQ7 IP is utilized to configure and manage the AXI DMA IP, ensuring seamless communication and operational control within the system.

The communication between the DMA and FFT is done through the AXI Stream Interface. Special care has been taken to implement the last signal in both input and output data ports of the FFT IP, ensuring that the data transfers between the DMA and FFT IPs occur successfully.

IP Information

1. Utilization Information of the Custom Made 8 Point FFT

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	400	-
FIFO	-	-	-	-	-
Instance	-	20	1392	2886	-
Memory	8	-	387	25	0
Multiplexer	-	-	-	1228	-
Register	-	-	929	-	-
Total	8	20	2708	4539	0
Available	280	220	106400	53200	0
Utilization (%)	2	9	2	8	0

Detail

2. Interface information:

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	fft	return value
ap_rst_n	in	1	ap_ctrl_none	fft	return value
input_vector_TDATA	in	64	axis	input_vector_V_data_V	pointer
input_vector_TVALID	in	1	axis	input_vector_V_last_V	pointer
input_vector_TREADY	out	1	axis	input_vector_V_last_V	pointer
input_vector_TLAST	in	1	axis	input_vector_V_last_V	pointer
output_vector_TDATA	out	64	axis	output_vector_V_data_V	pointer
output_vector_TREADY	in	1	axis	output_vector_V_data_V	pointer
output_vector_TVALID	out	1	axis	output_vector_V_last_V	pointer
output_vector_TLAST	out	1	axis	output_vector_V_last_V	pointer

3. Cosimulation Report

Cosimulation Report for 'fft'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	269	269	269	NA	NA	NA

Export the report(.html) using the Export Wizard

4. Performance Estimates

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.400	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
263	270	263	270	none

FFT IP CODES:

1. Source code:

```
#include "information.h"

#include <hls_stream.h>

#include <complex>

#include <stdint.h>



#include <math.h>




// Struct definition for axis_data
```

```
void fft(hls::stream<axis_data> &input_vector, hls::stream<axis_data>
&output_vector)

{

#pragma HLS INTERFACE ap_ctrl_none port=return

#pragma HLS INTERFACE axis register both port=output_vector

#pragma HLS INTERFACE axis register both port=input_vector



    // the input array in which we will place the extracted input for the
fft

    // operation

    std::complex<float> input_array[N];

    std::complex<float> reversed_array[N];

    std::complex<float> FFT_output[N];



    // this loop is used for storing the input data into an array which is

    // then used to perform the fft operation

    for (int i = 0; i < 8; i++)

    {



        axis_data input_data = input_vector.read();

        uint64_t raw_data = input_data.data.to_uint64();



        uint32_t real_part = static_cast<uint32_t>(raw_data & 0xFFFFFFFF);
// Lower 32 bits
```

```
        uint32_t imag_part = static_cast<uint32_t>((raw_data >> 32) &
0xFFFFFFFF); // Upper 32 bits


        float real = *reinterpret_cast<float *>(&real_part);

        float imag = *reinterpret_cast<float *>(&imag_part);


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

        if (input_data.last && i < 7)

        {

            // Break early if the last signal arrives before all 8
elements are read

            break;

        }

    }


    // Array of twiddle factors

    const std::complex<float> W[4] = {

        std::complex<float>(1, 0),

        std::complex<float>(std::cos(-2 * M_PI / 8), std::sin(-2 * M_PI /
8)),

        std::complex<float>(std::cos(-4 * M_PI / 8), std::sin(-4 * M_PI /
8)),

        std::complex<float>(std::cos(-6 * M_PI / 8), std::sin(-6 * M_PI /
8))

    };
```

```
    const int rev8[N] = {0, 4, 2, 6, 1, 5, 3, 7}; // Permutation for
bit-reversed order


    for (int i = 0; i < N; i++) {


        reversed_array[i] = input_array[rev8[i]]; // Rearrange data based
on the reverse bit order

    }


    std::complex<float> temp1[N];

    std::complex<float> temp2[N];


    // Stage 1: Butterfly computation with stride 2

    for (int i = 0; i < N; i += 2) {


        temp1[i] = reversed_array[i] + reversed_array[i + 1];      // Sum

        temp1[i + 1] = reversed_array[i] - reversed_array[i + 1];  //
Difference

    }


    // Stage 2: Butterfly computation with stride 4

    for (int i = 0; i < N; i += 4) {
```

```
        for (int j = 0; j < 2; ++j) {

            temp2[i + j] = temp1[i + j] + W[2 * j] * temp1[i + j + 2];
// Sum with twiddle factor

            temp2[i + 2 + j] = temp1[i + j] - W[2 * j] * temp1[i + j + 2];
// Difference with twiddle factor

        }

    }



    // Stage 3: Final butterfly computation with stride 8

    for (int i = 0; i < N / 2; i++) {

        FFT_output[i] = temp2[i] + W[i] * temp2[i + 4];      // Sum with
twiddle factor

        FFT_output[i + 4] = temp2[i] - W[i] * temp2[i + 4];  // Difference
with twiddle factor

    }



    // Writing FFT output to the output_vector stream

    for (int i = 0; i < N; i++) {

        float real_part = FFT_output[i].real();

        float imag_part = FFT_output[i].imag();



        uint32_t real_part_int = *reinterpret_cast<uint32_t
*>(&real_part);

        uint32_t imag_part_int = *reinterpret_cast<uint32_t
*>(&imag_part);
```

```
        uint64_t packed_data = (static_cast<uint64_t>(imag_part_int) <<
32) | real_part_int;

        axis_data output_data;

        output_data.data = (ap_uint<64>)packed_data;

        output_data.last = (ap_uint<1>)((i == N - 1) ? 1 : 0); // Set last
signal for the last element



        // Write to the output stream

        output_vector.write(output_data);

       }

}
```

2. information.h file code:

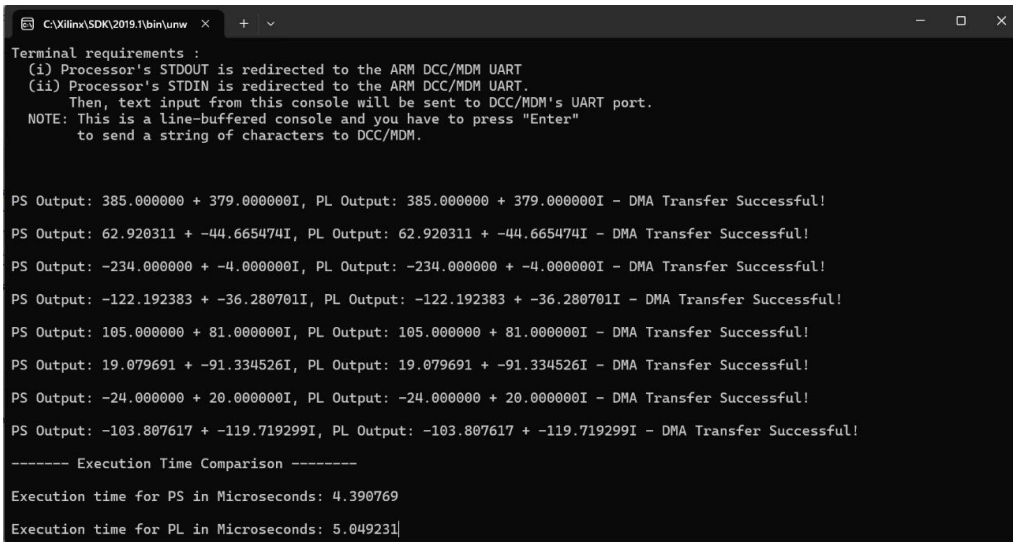
```
#define N 8

#include <hls_stream.h>

#include "ap_int.h"




struct axis_data

{

    ap_uint<64> data;

    ap_uint<1> last;


};
```

Results And Their Comparison:

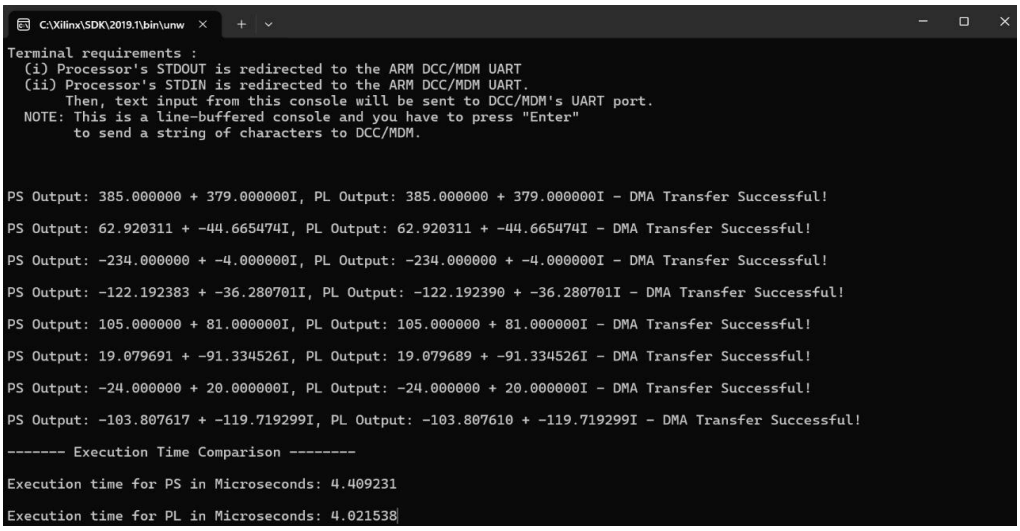
In this section, we will illustrate the comparison of the custom-made 8-point FFT IP and the built-in FFT IP available in Vivado. We will do this by running the same project for both of them, with the only difference being the IPs.

The project consists of executing an 8-point Fast Fourier Transform on a given input using PS and PL and comparing their execution times.

1. Results for Custom-made IP:



2. Results for Built-in IP:



We can observe that although both the IPs have the same output data, the built-in IP is faster than the custom-made one due to better optimization.

Conclusion

This project successfully developed a custom 8-point FFT IP using Vivado HLS, integrated it into an FPGA-based system, and validated its performance through testing. The comparison with Vivado's pre-built FFT IP highlighted a key trade-off in speed. This hands-on experience reinforced my knowledge of digital signal processing, FPGA system design, and high-level synthesis, providing a strong foundation for further exploration of hardware optimization techniques**.**

References:

1. AELD Lab Videos on Algorithms to Architecture YouTube channel, link:IIITD AELD Lab7_P1: HLS IP with AXI Stream Interface #zynq #vivado #zedboard #vivado #hls