WES 237C: Project 3 Discrete Fourier Transform (DFT)

Yi Song
yis057@ucsd.edu
PID# A53266440

Sebastian Nevarez
senevarez@ucsd.edu
PID# A59021322

---

1. Introduction

The goal of this project is for us to design different architectures that implement the Discrete Fourier Transform (DFT). This report details our design process and answers the questions posed in the project description.

2. Questions

Question 1: What changes would this code require if you were to use a custom CORDIC similar to what you designed for Project: CORDIC? Compared to a baseline code with HLS math functions for cos() and sin(), would changing the accuracy of your CORDIC core make the DFT hardware resource usage change? How would it affect the performance? Note that you do not need to implement the CORDIC in your code, we are just asking you to discuss potential tradeoffs that would be possible if you used a CORDIC that you designed instead of the one from Xilinx.

The CORDIC designed in the last project takes in x and y cartesian coordinates and outputs the length R and the theta angle with respect to +x axis counterclockwise. To replace the cos and sin calculations with that, we would need to pick cartesian coordinates on a unit circle based on the DFT size and then pass them into the CORDIC. From the output length R, we can calculate the cos and sin values using the inputs x, y and R. One suggestion would be to use CORDIC to compute a LUT for the DFT size and then use the LUT cos/sin table to compute DFT. For the implementation using math functions cos() and sin(), it has an accuracy of 1e-6 compared to CORDIC accuracy of 1e-4. From this we can see that using the math.h library is more accurate than the developed CORDIC. In terms of resource utilization, using math functions incurs a huge number of DSP and LUT exceeding the available resources. It also has a higher latency than the LUT method so the performance would be worse than the LUT method.

RMSE (R)	RMSE (I)
0.000007830698451	0.000004315968454

Table 1: Math.h cos() and sin() implementation RMSE

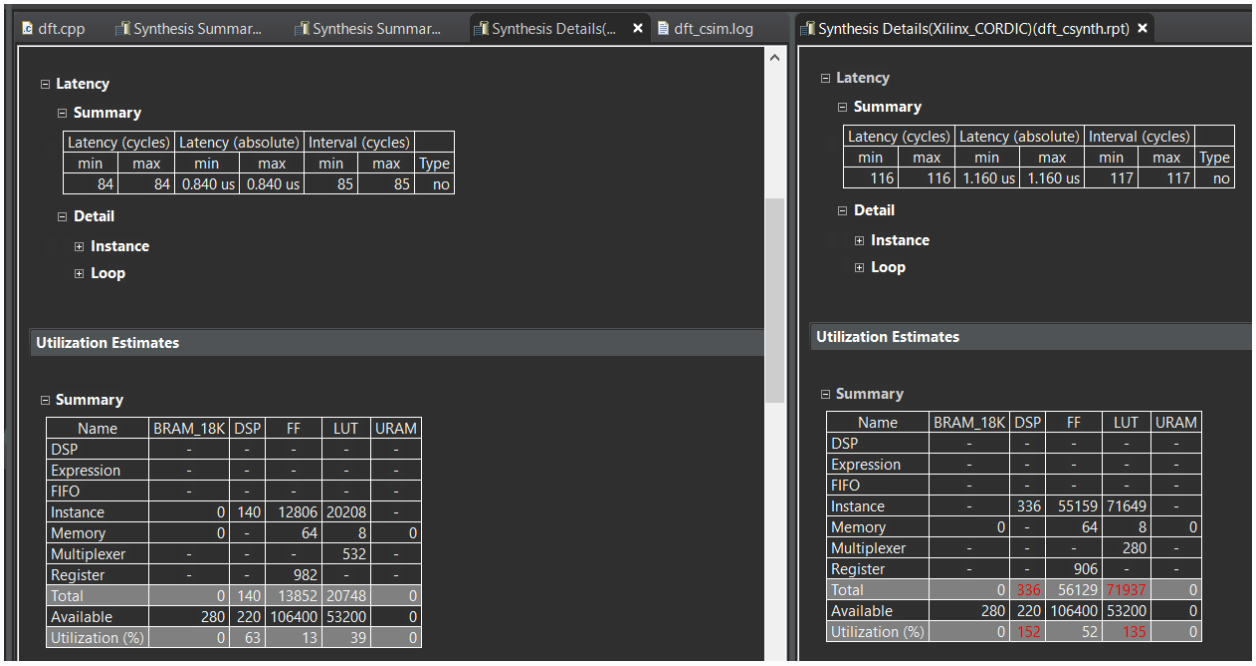


Figure 1: cos() and sin() LUT implementation (left) vs Xilinx CORDIC implementation (right)

Question 2: Rewrite the code to eliminate these math function calls (i.e. cos() and sin()) by utilizing a table lookup. How does this change the throughput and resource utilization? What happens to the table lookup when you change the size of your DFT?

When we implemented the LUT approach, we ran into an II violation issue. The for loop has an II of 6 but we are only setting it to 1. To solve this problem, we used the pipeline pragma to set the initial interval to be larger than 6.

Previously we were calculating throughput using

100 / (cycle time * latency)

but this assumes we are processing only 1 data element. In this case, we are doing an N-size DFT where N is the size of our data element. To account for this, we should include multiplying the size of input in our throughput calculation. This gives us

256 / (10ns * 393493) = 65.0583KHz

We were heavily limited by the latency so we needed to use pragmas to solve this. In terms of resource utilization, we are using much less than the cos/sin approach because we save a lot of runtime arithmetic by using LUT. When the size of our DFT increases, we will have a larger LUT with the same size.

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	2	5	1223	1371	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	173	-
Register	-	-	6	-	-
Total	4	5	1229	1544	0
Available	280	220	106400	53200	0
Utilization (%)	1	2	1	2	0

Figure 2: LUT based DFT256

Question 3: Modify the DFT function interface so that the input and outputs are stored in separate arrays. Modify the testbench to accommodate this change to the DFT interface. How does this affect the optimizations that you can perform? How does it change the performance? And how does the resource usage change? **You should use this modified interface for the remaining questions.**

We have smaller resource utilizations compared to sharing the same interface for inputs and outputs. We are also a little bit faster in terms of latency with a throughput increase to

2556 * 1000 / (10 * 333825) = 76.68KHz

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	25	-
FIFO	-	-	-	-	-
Instance	2	5	1005	1097	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	114	-
Register	-	-	151	-	-
Total	2	5	1156	1236	0
Available	280	220	106400	53200	0
Utilization (%)	~0	2	1	2	0

Figure 3: LUT based DFT256 with changes in the input/output interface

Question 4: Loop Optimizations: Examine the effects of loop unrolling and array partitioning on the performance and resource utilization. What is the relationship between array partitioning and loop unrolling? Does it help to perform one without the other? Plot the performance in terms of number of DFT operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for resources (showing LUTs, FFs, DSP blocks). What is the general trend in both cases? Which design would you select? Why?

Using loop unrolling and array partitioning in our inner loop, we keep them with the same factor - when unrolling the loop by a factor of N, we are using an N-size data array in the inner loop to calculate the temp real/imag output. Without appropriate array partitioning, unrolling the inner loop offers no increase in performance, as the number of concurrent read operations is limited by the number of ports to memory. In the plot, we can see that as we increase the unroll and partition factor, more resources are being used with no gain in throughput. This makes sense as we didn't utilize pipeline to reduce the II of each loop. When trying to pipeline our inner loop, we see a gradual increase in throughput and a reduction in resource utilization. Since a lower pipeline factor causes the synthesis to use more resources than the PYNQ board has, the best throughput we are able to achieve with the resources available is 1.53MHz when pipelined with II=60.

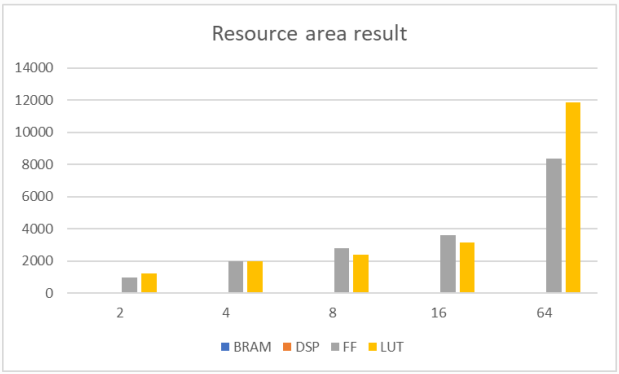
The critical path of our code

```
real_output[i] += (temp_real - temp_imag);
```

accumulates the results calculated from temp values to the real output. This is a read after write operation, so when we pipeline it, the parallelled operations have to run sequentially. So the more II is increased, the more latency and less resource utilization it will have.

Unroll & Array_Partition	Throughput	BRAM	DSP	FF	LUT
2	76.8KHz	4	5	996	1224
4	76.8KHz	2	7	2014	1974
8	76.8KHz	2	7	2795	2412
16	76.8KHz	2	7	3588	3170
64	74.8KHz	2	36	8351	11870
128	76.7KHz	0	36	21802	541653

Table 2



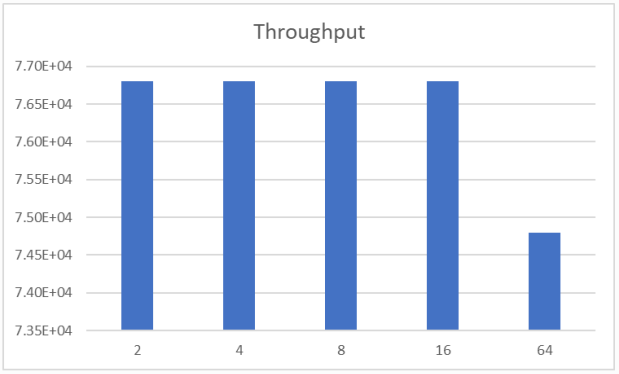


Table 2 & Figure 4: Resource area and throughput vs Loop unroll and array partition factor

pipeline	Throughput	BRAM	DSP	FF	LUT
20	3.92MHz	28	257	118686	86011
40	2.2MHz	16	130	101089	60272
60	1.53MHz	12	87	95468	52368
256	0.36MHz	2	20	73446	32627

Table 3

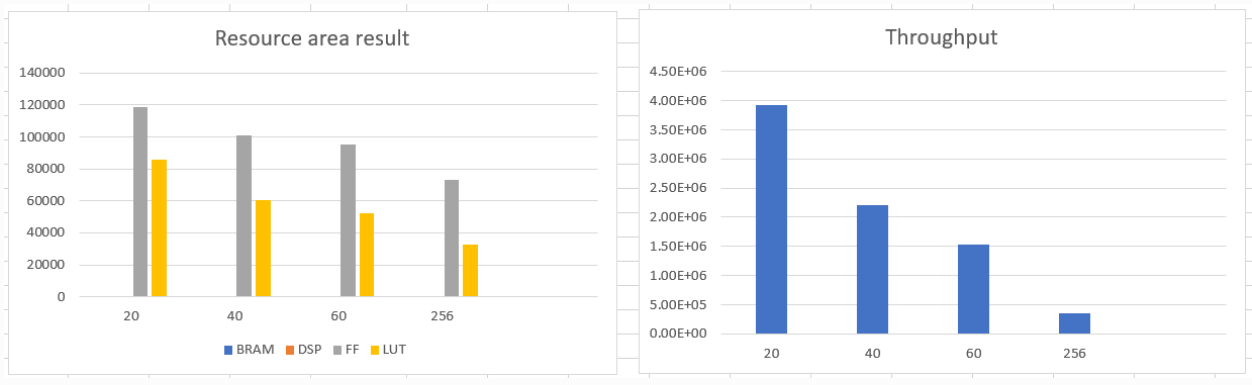


Table 3 & Figure 5: Resource area and throughput vs pipeline

Question 5: Dataflow: Apply dataflow pragma to your design to improve throughput. You may need to change your code and make submodules so that it aligns with the task-level or function-level modularity that dataflow can exploit; Xilinx provides some examples of dataflow code. The Vitis HLS User Guide and this summary provide more information. How much improvement does dataflow provide? How does dataflow affect resource usage? What about BRAM usage specifically? Did you modify the code to make it more amenable to dataflow? If so, how? Please describe your architecture(s) with figures on your report.

Dataflow optimization provided little improvement on resource usage at a cost of throughput. The optimization highlighted in green is the optimal one chosen by our team.

Dataflow optimization + pipeline & unroll + array_partition	BRAM	DSP	FF	LUT	Throughput
dataflow+pipeline	2	5	1021	1184	19.5KHz
dataflow+pipeline +2+2	4	5	1398	1336	28.9KHz
dataflow+pipeline +4+4	8	5	2031	1793	42.1KHz
dataflow+pipeline +8+8	16	5	2520	4414	19.3KHz

Table 4

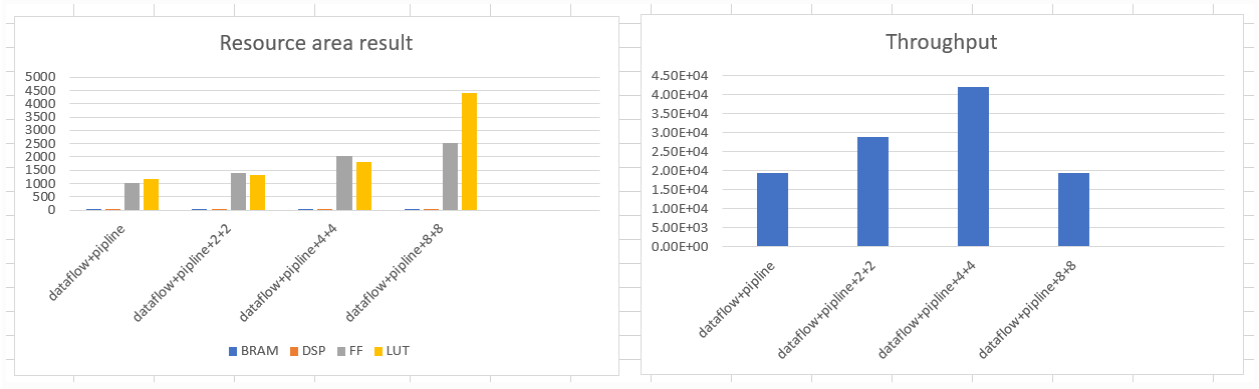


Table 4 & Figure 6: Resource area and throughput vs Dataflow + pipeline & unroll + array_partition optimizations

We found the improvement is little compared to the previous implementation. Below we have included a code snippet we believe explains why this is. The code in the red and green boxes are dependent on each other to calculate the final real and imaginary outputs. As they are data dependent, changing them into individual functions will not help with parallelizing the workload. The code in the green box needs to start before the red box finishes.

```
for (i = 0; i < SIZE; i += 1) {

    #pragma HLS pipeline II=64
    for (j = 0; j < SIZE; j += 1) {

        #pragma HLS unroll factor=64
        k = (i*j)%SIZE;
        c = cos_coefficients_table[k];
        s = sin_coefficients_table[k];
        temp_real = real_sample[j] * c;
        temp_imag = imag_sample[j] * s;
        real_output[i] += (temp_real - temp_imag);
        temp_real = real_sample[j] * s;
        temp_imag = imag_sample[j] * c;
        imag_output[i] += (temp_real + temp_imag);
```

Figure 7: DFT inner loop

Question 6: Best architecture: Briefly describe your “best” architecture. In what way is it the best? What optimizations did you use to obtain this result? What are the tradeoffs that you considered in order to obtain this architecture?

Table 5 below describes two good architectures comparing their different resource utilization and throughput. The first architecture had better throughput while the second architecture had better resource utilization. So the tradeoff here is between high throughput and low resource utilization.

Optimizations	BRAM	DSP	FF	LUT	Throughput
II=60 between the outer loop and inner loop	12	87	95468	52368	1.53MHz
dataflow+II=256+ unroll by 4+ array_partition by 4	8	5	2031	1793	42.1KHz

Table 5: Best architectures

Question 7: Streaming Interface Synthesis: Modify your design to allow for streaming inputs and outputs using hls::stream. You must write your own testbench to account for the function interface change from DTYPE to hls::stream. NOTE: your design must pass Co-Simulation (not just C-Simulation). You can learn about hls::stream from the HLS Stream Library. An example of code with both hls::stream and dataflow is available (along with its testbench) here, and another example showing hls::stream between functions. Describe the major changes that you made to your code to implement the streaming interface. What benefits does the streaming interface provide? What are the drawbacks?

The major changes made were hls::stream<transPkt> and union fp_int introduced in the cpp and header files. “transPkt” is used as an interface for input and output streaming with the typedef of typedef ap_axis<32,2,5,6>. The 32 here means 32 bits, so it is compatible with the floating point data type that we have previously used. After we read the streaming data in transPkt variables, we need to assign them to an union type variable with int and floating point data types. The integer type is used for indexing and the floating point type is used for calculations. After calculating the output data, we need to convert the floating point back to an integer for interface purpose and set the “tlast” bit to 1 if we reach the end of the matrix.

```
for (int i=0; i<SIZE; i++)
{
    real = real_sample.read();
    imag = imag_sample.read();

    real_data[i].i = real.data;
    imag_data[i].i = imag.data;

    real_out[i].fp = 0;
    imag_out[i].fp = 0;
}
```

Figure 8: Code snippet for reading-in data

Some of the benefits of using streaming interface are

- It allows us to use DMA (direct memory access) so the host CPU is not burdened with memory transfer and hence is available to perform other more important tasks
- Parallelism and pipelining helps in optimizing the performance of by utilizing available resources efficiently

Some of the drawbacks are

- Using streaming can result in complex hardware structures and so debugging and understanding the generated hardware can be challenging
- Resource utilization - some HLS tools may not always generate the most efficient hardware in terms of resource utilization so we need to analyze the generated hardware to make sure it meets performance and resource constraints

3. PYNQ Demo

Below we have included screenshots of our output from Jupyter notebook. Our errors are within 1e-5 magnitude and we are able to achieve numpy accuracy doing 1024 size DFT. So our DFT implementation is a success. 😀

In [11]:
```
plt.figure(figsize=(10, 5))
plt.subplot(1,2,1)
plt.bar(ind,real_error)
plt.title("Real Part Error")
plt.xlabel("Index")
plt.ylabel("Error")
#plt.xticks(ind)
plt.tight_layout()

plt.subplot(1,2,2)
plt.bar(ind,imag_error)
plt.title("Imaginary Part Error")
plt.xlabel("Index")
plt.ylabel("Error")
#plt.xticks(ind)
plt.tight_layout()
```

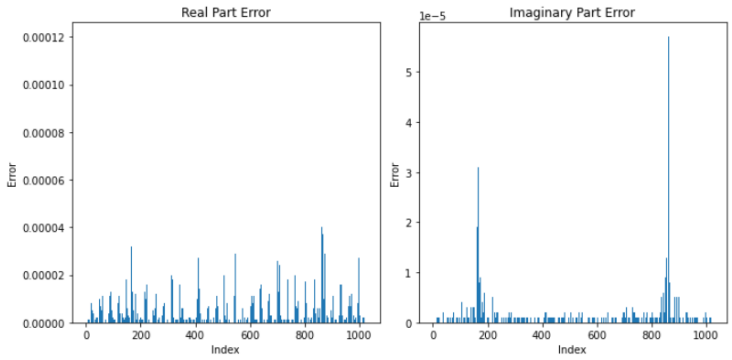


Figure 9: Plots of real and imaginary part error

In [12]:
```
freq=np.fft.fftfreq(1024)

plt.figure(figsize=(7, 4))
plt.subplot(1,2,1)
plt.plot(freq,out_r,label='real')
plt.plot(freq,out_i,label='imag')
plt.title("1024-DFT")
plt.xlabel("Frequency")
plt.ylabel("DFT real and imaginary data")
plt.legend()
plt.tight_layout()
plt.subplot(1,2,2)
plt.plot(freq,golden_op.real,label='real')
plt.plot(freq,golden_op.imag,label='imag')
plt.title("1024-FFT -Numpy")
plt.xlabel("Frequency")
plt.ylabel("FFT real and imaginary data")
plt.legend()
plt.tight_layout()
plt.show()
```

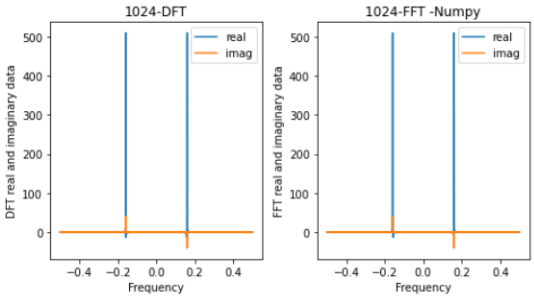


Figure 10: Implemented vs numpy