Project 3 Discrete Fourier Transform

Name: Arunkumar Ravichandran and Debosmit Majumder
PID : A53244124, A53242244
Email: arravich@eng.ucsd.edu , debosmit.majumdar30@gmail.com

Question 1.

What changes would this code require if you were to use a custom CORDIC similar to what you designed for Project 2? 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.

Answer:

The implemented cordic function has to be declared in the header file. It has to be called every time “cos” and “sin” functions are calculated.

We refer to the “**dft256_baseline**” here.

The accuracy of the cordic core could be changed by increasing or decreasing the number of iterations in the cordic algorithm. Decreasing the number of iterations for the loop in the cordic core will decrease the accuracy but will reduce the hardware resource usage. Also, due to reduced number of loop iterations, the latency will decrease. So, we can perform a trade off between the accuracy and the resource usage as well as throughput.

Decreasing the accuracy by reducing the number of loop iterations will also improve the performance.

There may be another case: Assuming the HLS tool does not optimize the number of bits used for each input angle and uses float data types, the data types in our implemented cordic core can be changed to fixed point types. According to the input angles, the “ap_fixed” should have specific number of bits. This will also decrease the hardware resource usage.

2nd case: Maybe we can use inline functions to call the cordic coe function whenever we have to compute “cos” and “sin” functions. That may reduce the calling time of the functions by directly expanding the function code.

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 area? What happens to the table lookup when you change the size of your DFT?

Answer:

We refer to “**dft256_optimized1**” here.

Removing the math functions and using lookup tables improves the performance significantly. The throughput increases by a good margin. Also, the resource usage decreases (Refer to Table 2.1).

Assuming the HLS tool uses its own version of the Cordic Algorithm whenever “cos” and “sin” functions are called, it takes up several loop iterations and in each loop iteration, corresponding multiplication and division operations are done. If these functions are removed and lookup tables are used, then cordic algorithm is no longer needed and values can be fetched directly from memory.

Note: However, using lookup tables increases the number of BRAMS significantly.

As we increase or decrease the size of the DFT, the sizes of lookup tables also increase and decrease respectively. The size of the lookup tables is equal to the corresponding size of the DFT.

Table 2.1. Resource usage and throughput comparison across LUT implementation and sin/cos math function implementation.

	BRAM	DSP48	FF	LUT	Interval	Time	Throughput
dft256_baseline	20	190	9008	16762	3215619	8.41	36.7 Hz
dft256_optimized1	260	50	3714	3524	1115139	8.15	110 Hz

Question 3.

Modify the DFT function interface so that the input and outputs are stored in separate arrays. How does this affect the optimizations that you can perform? How does it change the performance? What about the area results? Modify your testbench to accommodate this change to DFT interface. **You should use this modified interface for the remaining questions.**

Answer:

We refer to "**dft256_optimized2**" here.

Separating the interfaces for input and output arrays enables us to modify and optimize the design in a few ways.

For separate interfaces, the temporary variable arrays can be done away with. Hence, that decreases the resource usage.

It decreases the BlockRAMs used because temporary variable arrays are not used anymore.

It also removes the 2nd loop. Thus, we see that the number of LUTs and FFs decrease (Refer Table 3.1.).

However, the latency increases and throughput decreases because now interfaces have increased in the loop. Accessing the input interface takes more time now.

Potential Case: We can use separate bit-widths/data-types for each of the input and the output interfaces now. But applying them did not change the resource usage or the throughput much.

Table 3.1. Resource usage, throughput comparison across 3 architectures

	BRAM	DSP48	FF	LUT	Interval	Time	Throughput
dft256_baseline	20	190	9008	16762	3215619	8.41	36.7 Hz
dft256_optimized1	260	50	3714	3524	1115139	8.15	110 Hz
dft256_optimized2	256	50	3654	3430	1180162	8.68	97.62 Hz

Question 4.

Answer:

We refer to designs “**dft256_optimized3, dft256_optimized4, dft256_optimized5”.**

Array partitioning was implemented using the #pragma HLS array_partition variable=cos_coefficients_table block factor=8. By changing the array partitioning method(block, cyclic), we didn’t observe any change in the performance. Changes in the array partitioning factor will change the resource usage. Increasing the array partitioning factor reduces the BRAM usage significantly but LUT/ FF usage increases as shown in the below graph, Fig. 4.1. This because now a single array is partitioned into separate arrays.

Fig: 4.1 Array Partitioning was resource usage especially BRAM usage

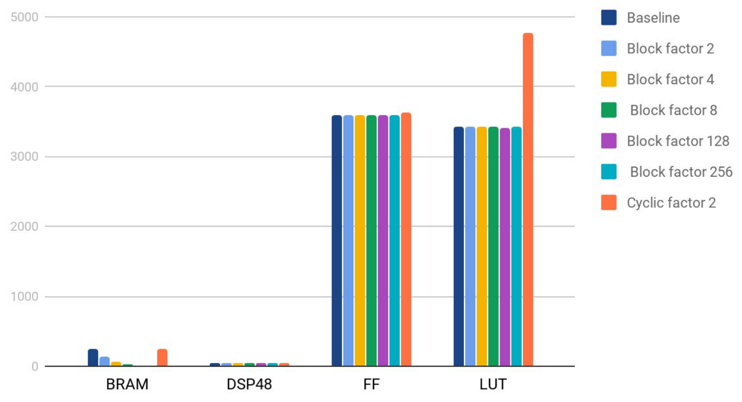
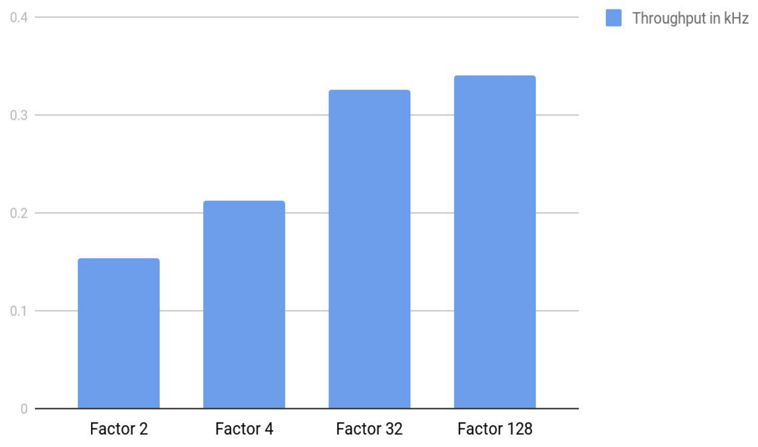


Fig 4.2: Loop unrolling versus Throughput

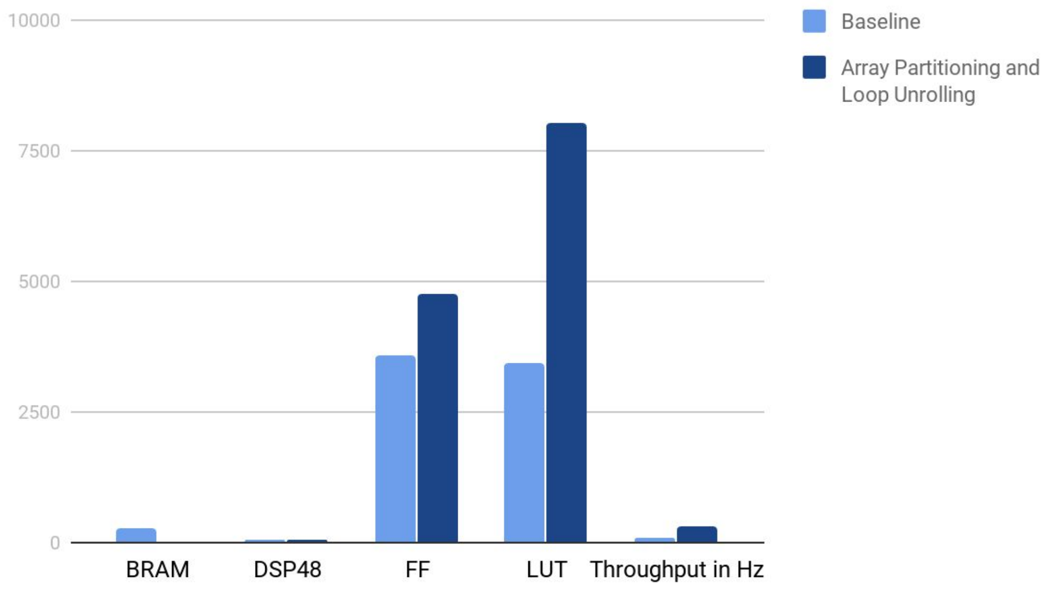


Loop unrolling was done and we observed improvement in the throughput performance. As we unrolled the loop by increased loop unrolling factors, the resource usage significantly increased but throughput also increased. This can be seen in the Figure 4.2.

The reasons for this are that by unrolling the loops, the tool performs multiple iterations of the loop simultaneously. Hence, now the tool requires separate resources (like adders, multipliers, etc) to perform multiple loop iterations. But we observe a decrease in the latency and an increase in the throughput because the number of loop iterations done simultaneously is proportional to the unroll factor.

Loop unrolling and Array partitioning should be done simultaneously to improve the throughput performance along with reduction of BRAM usage. Loop unrolling factor was set at 128 and array partitioning factor was set at 128 with block partitioning.
We have also made a comparison of Baseline DFT implementation with the Loop unrolled-array partitioned implementation in the figure 4.3 We will choose this design as a feature in our best architecture.

Fig 4.3: Baseline and Array Partitioning and Loop Unrolling



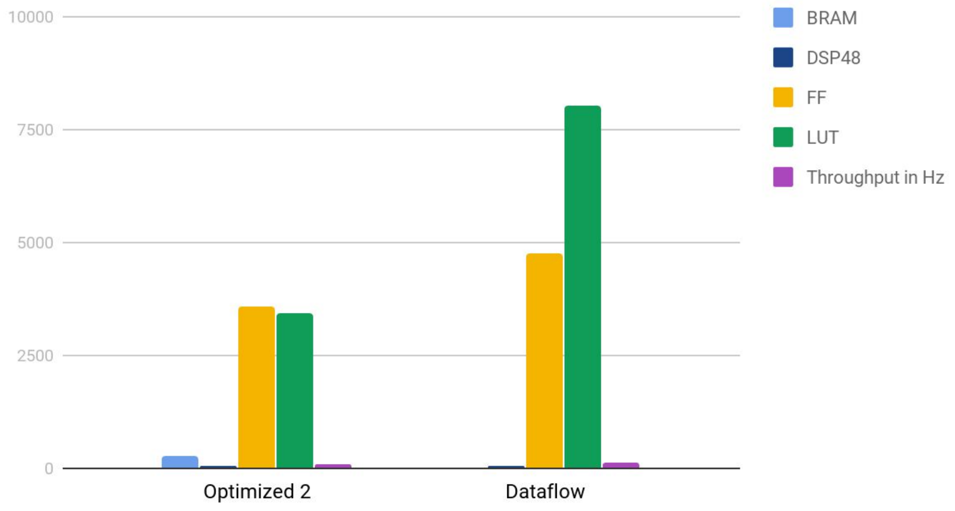
Question 5.

Please read dataflow section in the HLS user guide, and apply dataflow pragma to your design to improve throughput. You may need to change your code and make submodules. How much improvement can you make with it? How much does your design use resources? What about BRAM usage? Please describe your architecture with figures on your report. (Make sure to add dataflow pragma on your top function.)

Answer: We refer to “**dft256_dataflow” here.**
The dataflow pragma in HLS allows task level pipelining. We have made 2 functions: 1 to take the real and imaginary values as input and 2nd function to calculate the output real and imaginary values.

As we can see from the below graph 5.1 , compared to the baseline architecture, the throughput has increased. This is because “dataflow” pragma is performing task-level pipelining to both the sub-functions. We are using single-producer and single-consumer. The data-flow is given by: input -> temp -> output. However, now the resource usage has increased; especially the BRAM usage has increased because we need to use temporary variables to store the input values and provide to the next function.

Fig. 5.1: Dataflow optimized



Architecture	BRAM	DSP48	FF	LUT	Throughput in Hz
Optimized 2	256	50	3590	3425	97
Dataflow	2	56	4771	8013	110

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 is trade-off you consider for the best architecture?

Answer: We refer to “**dft256_best” here.**
For our best architecture we use the loop unrolling factor of 128 and array partitioning factor of 128. With this we have slightly increased amount of resource usage but the performance is significantly improved from the baseline. Also with the interface separation, we saw a reduction in the resource usage. Hence interface separation was done even though it reduced the throughput by a small value. This is the trade-off that we have considered. Refer to Fig. 6.1 and 6.2.

Fig. 6.1: Throughput in Hz vs. Architectures

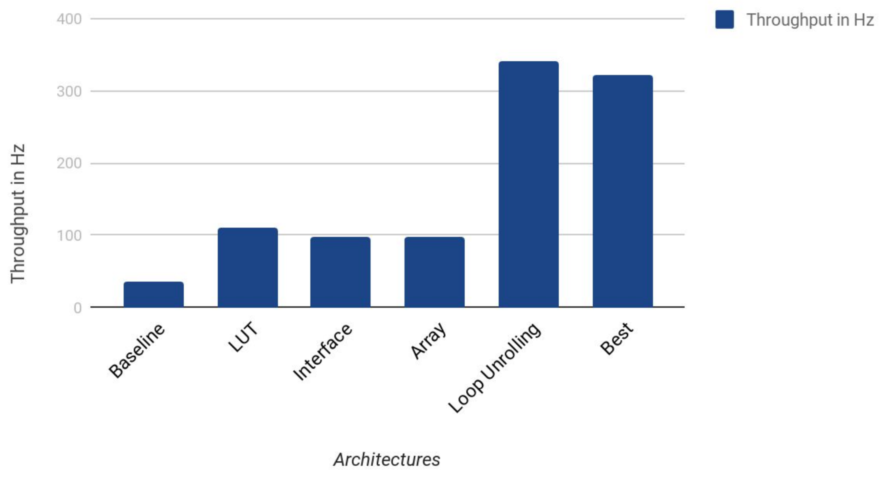
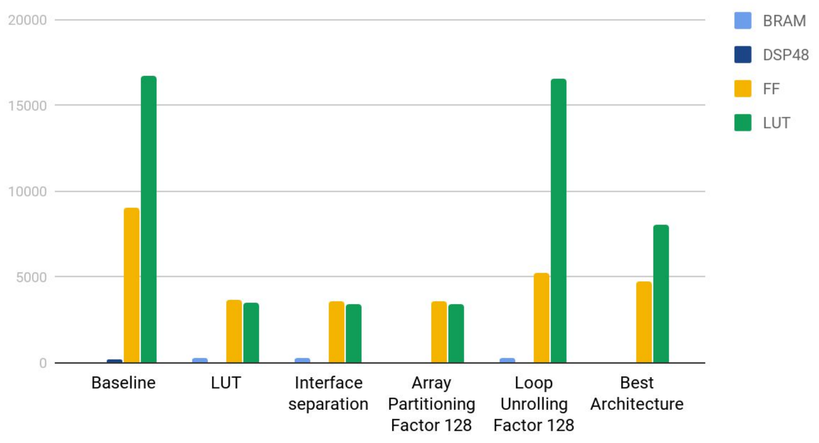


Fig 6.2: Resource usage across different architectures



Question 7 (Bonus)