Project 3: Discrete Fourier Transform

Name: Will Zegers
PID: A09291983
Email: wzegers@eng.ucsd.edu

Question 1.

As hinted at in question 2, the `cos()` and `sin()` are computationally expensive and resource demanding operations in relation to the rest of the DFT design, so its likely that replacing the HLS math function with a CORIDC would give significant benefits. Moreover, by structuring the CORDIC and designing its interface with the application in mind, very little would need to change in the DFT code to switch from HLS math to custom CORDIC functions.

Using a custom CORDIC, rather than the one provided in the Xilinx HLS tools, would allow for finer-grained control and a more specific implementation. In other words, whereas the Xilinx CORDIC is designed as a general use component for a wide breath of domain, a custom CORDIC would only provide what is necessary to the design in which its being used and, could therefore, be optimized to remove any unnecessary functionality. This could mean both increasing performance and reducing resource usage since it's designed for a very specific application; of course, this comes with the tradeoff of having to actually write and implement this custom CORDIC if one is not already available.

Question 2.

Table 1 shows a comparison between the baseline implementation, using `cos()` and `sin()` function calls, and using a table of pre-computed values (optimization 1). By removing the expensive sine and cosine calculations, this not only *significantly* boosted throughput but also cut down in the number of resources used, the majority of which were likely used in these calculations.

Table 1: Comparison and percent change between baseline and optimization1

	Baseline	Optimization 1	Change
Throughput	1.777 to 6.525 Hz	106.580 Hz	+1530 to 5897 %
BRAM_18K	15	4	-73.33 %
DSP48E	44	17	-61.4 %
FF	7437	1372	-81.6 %
LUT	23168	2415	-89.6 %

The effect of changing the value of N (and consequently the size of the array of lookup values) depends on what specific sizes N are being compared, as can be seen in table 2. Both 32 and 256-sized DFTs use 4 BRAMs: four `DTYPE` arrays for coefficient and temp values, each of size 1024 and 8192 bytes respectively, both of which can fit into a single 18K BRAM. Therefore, it makes sense that DFTs with $N = 8$ and $N = 256$ use the same number of BRAMs.

Table 2: BRAM utilization for DFTs of size N

N	8	32	256	1024
BRAM_18K	0	4	4	8
FF	1495	1327	1372	1398

Each array in the $N = 1024$ implementation, however, requires 1024×32 bytes $= 32768$ bytes, meaning two 18K BRAMS are needed to store a single array. Thus, the number of BRAMs to store the lookup table in the 1024 design is double that of the 32 and 256 designs.

Finally, the implementation with $N = 8$ uses no BRAMs and is, instead, likely storing these values using individual registers made of flip-flops; hence, the higher FF utilization in the $N = 8$ design.

Question 3.

The biggest change present in table 3 is the reduction in BRAM utilization, from 4 to 2, thanks to removing the intermediate temporary arrays previously used in the DFT function. Apart from this, area and performance are only very marginally improved.

However, the biggest advantage of splitting the input and output arrays allows for greater potential to pipeline and stream data from input to output. In the previous architecture, the matrix-multiply operation had to complete entirely before the data was copied from the temporary arrays back into the input/output array. By making a separate output array, data could potentially be made available (element-by-element) on each complete iteration of the outer for loop. If the DFT was feeding its output to another module in a larger design, the HLS tool could stream the output into the next module as it's made available by the DFT, rather than waiting for the entire function to complete and deliver a massive chunk of data all at once.

Table 3: Comparing optimization with inputs and outputs stored in the same (optimzation 1) and separate (optimization 2) arrays.

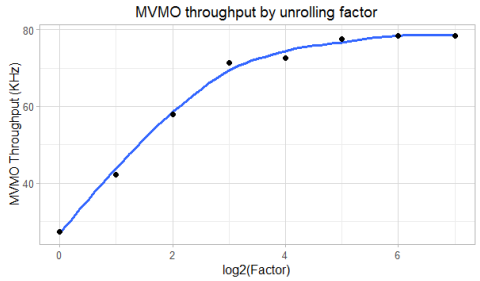
	Optimization 1	Optimization 2	Change
Throughput	106.580 Hz	106.678 Hz	+0.0919%
BRAM18	4	2	-50.000%
DSP48E	17	17	0 %
FF	1372	1325	-3.426%
LUT	2415	2360	-2.277%

Question 4.

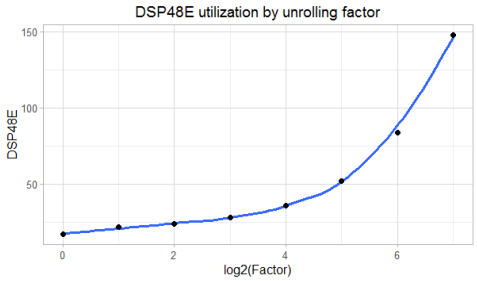
Figure 1 shows semi-log plots of unrolling the inner loop alone, sans any array partitioning. Absent from this figure is the BRAM_18K utilization, which was constant (2) for all cases (evidence that loop unrolling had no effect on memory structure).

Figure 1a shows that, up until an unrolling factor of about 8, matrix-multiplications throughput increases by about 14.7 KHz for every doubling of the unrolling factor. After this, higher loop unrolling factors offer diminishing returns and only see a total gain of 1.79 KHz going from unrolling factors 8 to 128.

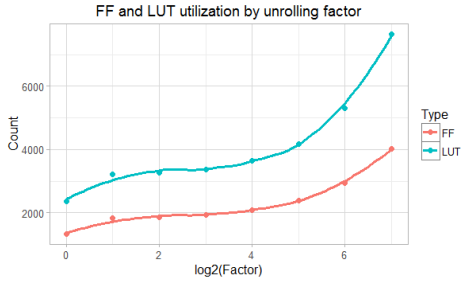
Similarly, figures 1b and 1c illustrate that resource utilization is approximately log-linear up until an unrolling factor of 16, after which utilization becomes exponential with each doubling of the unrolling factor.



(a) Matrix-multiplication throughput



(b) DSP48E utilization



(c) FF and LUT utilization

Figure 1: Effects of loop unrolling on DFT throughput (in matrix vector multiply operations, MVMO) and area

Though I have had success in the past with combining loop unrolling and array partitioning to yield even better optimization, in this case after playing with multiple partitioning factors and parameters (e.g. cyclic vs block, etc.), array partitioning seemed only to be detrimental to performance, area, or both. Table 4 shows typical result I was getting for loop unrolling in this particular case using a factor of 8, and array partitioning of the same. (Note this table is meant to be illustrative of the results I was getting. I tried tweaking *many* more partitioning parameters and factors, with no success).

Table 4: Comparing optimization with with just loop unrolling (optimization 2), loop unrolling and partitioning the input array (optimization 3), and loop unrolling and partitioning both the input and coefficient arrays.

	Optimization 2	Optimization 3	Optimization 4
MM Throughput	71.29 KHz	63.89 KHz	63.89 KHz
BRAM18	2	2	16
DSP48E	28	44	44
FF	1925	3067	3521
LUT	3298	5586	6796

Question 5.

In order to facilitate the implementation of the dataflow pragma, I restructured the code to make it more modular and adhere to Xilinx's prescribed "coding styles" for the pragma to work. The newly structured design appears in optimization 5, and is mainly structured around modularity and the single-producer-consumer model. Table 5 shows some comparison between optimization 2 (before restructuring), after restructuring (*without* dataflow), and optimization 5 (restructuring with dataflow).

Table 5: Results of restructuring the code and adding the dataflow pragma.

	Pre-restructuring	Post-restructuring	Post restructuring with dataflow
MM Throughput	27.31 KHz	20.17 to 28.99 KHz	40.40 KHz
BRAM18	2	8	14
DSP48E	17	17	21
FF	1325	1579	1966
LUT	2360	2405	3123

Overall, restructuring the design with an added dataflow pragma saw a 47% increase in throughput, but at the cost of 23%, 48%, and 32% greater DSP48E, FF, and LUT utilization, respectively. Most striking is the demand for 7x more BRAM, mainly used to implement the arrays that facilitate dataflow between modules (as for loops, in this case). Figure 2 shows the architecture of the restructured code with the implemented dataflow pragma. Note that I actually applied the pragma to the top-level *for* loop, rather than in the top-level `dft()` function, since nearly all of the computation and data handling is performed with this loop, and the dataflow pragma is not recursive to modules (or for loops) within its modules.

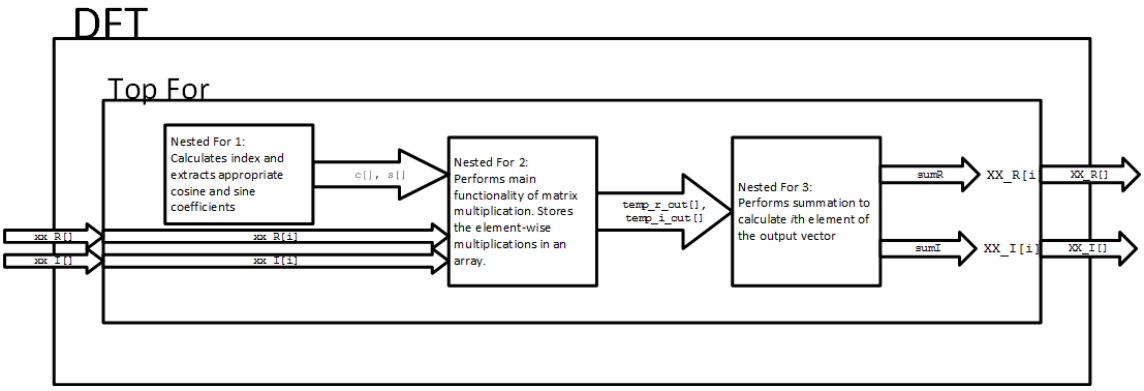
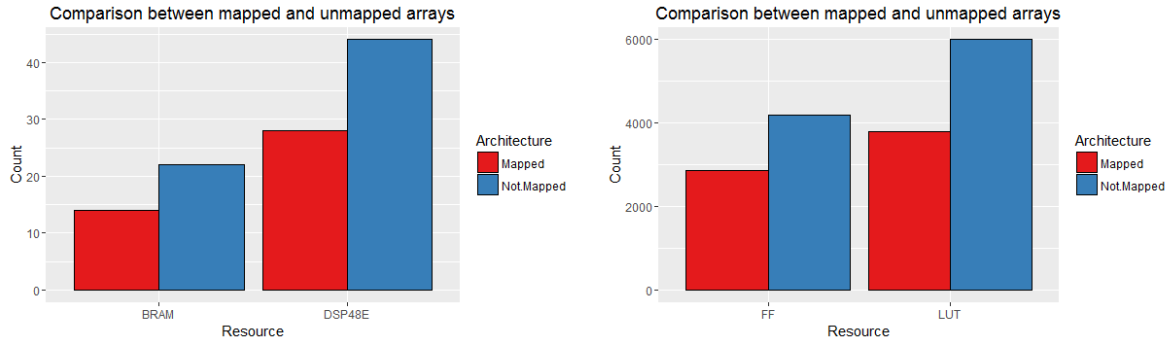


Figure 2: Architecture of the dataflow-structured code

Question 6.

For selecting a "best" architecture, I aimed to use a strategy similar to my analysis in question 4 for selecting an ideal loop unrolling factor. Essentially, I tried to hit a throughput right before the semi-log curve leveled off, while at the same time had a resource utilization that was right at the "elbow" of the curve; that is, right before resource usage became exponential with each subsequent optimization. Based on results from the previous questions, I opted to use a dataflow pragma on the outer loop, with a loop unrolling factor of 8 on the inner loops (based on the results from question 4).

I also introduced two additional optimizations — array mapping and loop pipelining — to either squeeze some extra performance from the design or reduce resource utilization. Figures 3 and 4 show a comparison between the base "best" architecture and architectures using these additional optimizations. Worth noting is that the pipeling optimization fails (i.e., it gives negative slack) without the use of array mapping.



(a) DSP48E utilization

(b) FF and LUT utilization

Figure 3: Effects of array mapping on DFT design by resource utilization

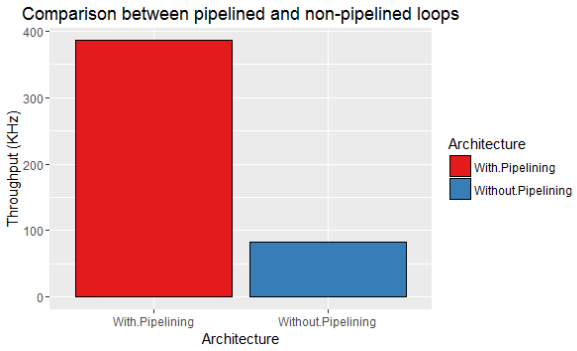


Figure 4: Effect on throughput of inner loop pipelining

Finally, in migrating the code and optimizations over to a DFT with input size 1024, all optimizations used in the 256 design needed no change in the 1024 design with one exception: the loop unrolling factor. I used the same analysis in question 4 to find the optimal points on the throughput and resource curves that would give an ideal unrolling factor for a DFT using a 1024 input vector. Incidentally, this once again turned out to be a factor of 8, so no change was actually necessary (though it at least gave me the data to rest assured that was the case).