**Project: Discrete Fourier Transform (DFT)**
**K. Maddox**

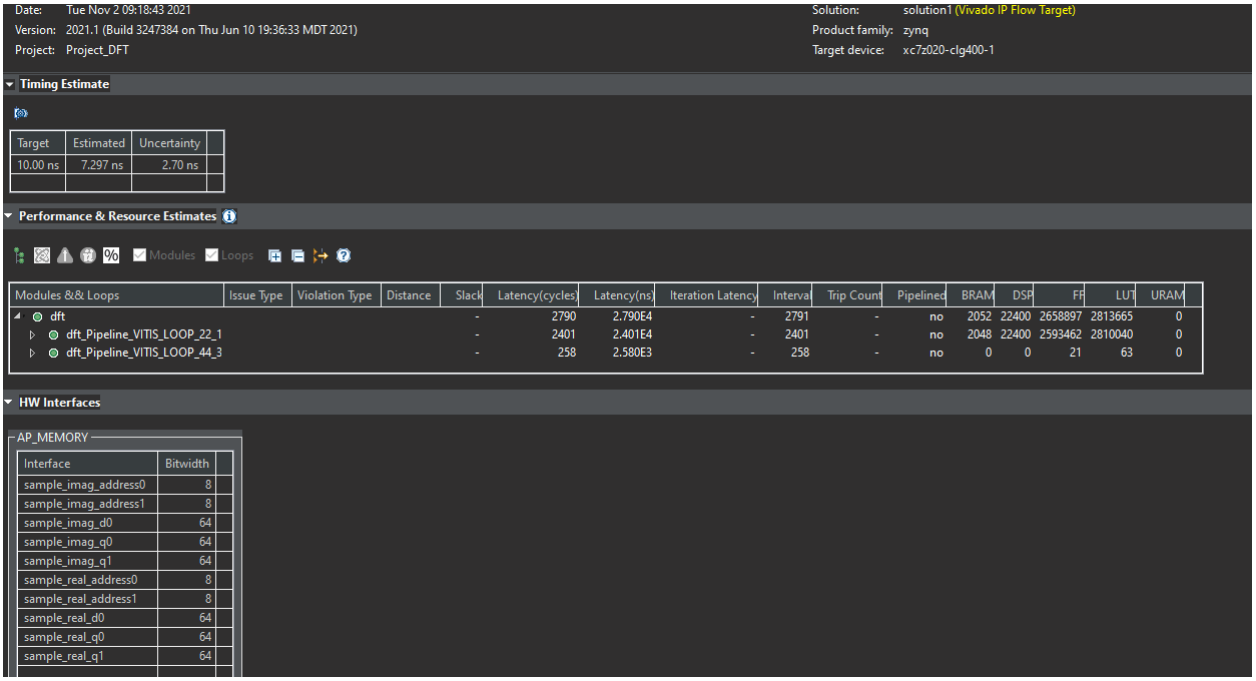


*Figure 1 Baseline*

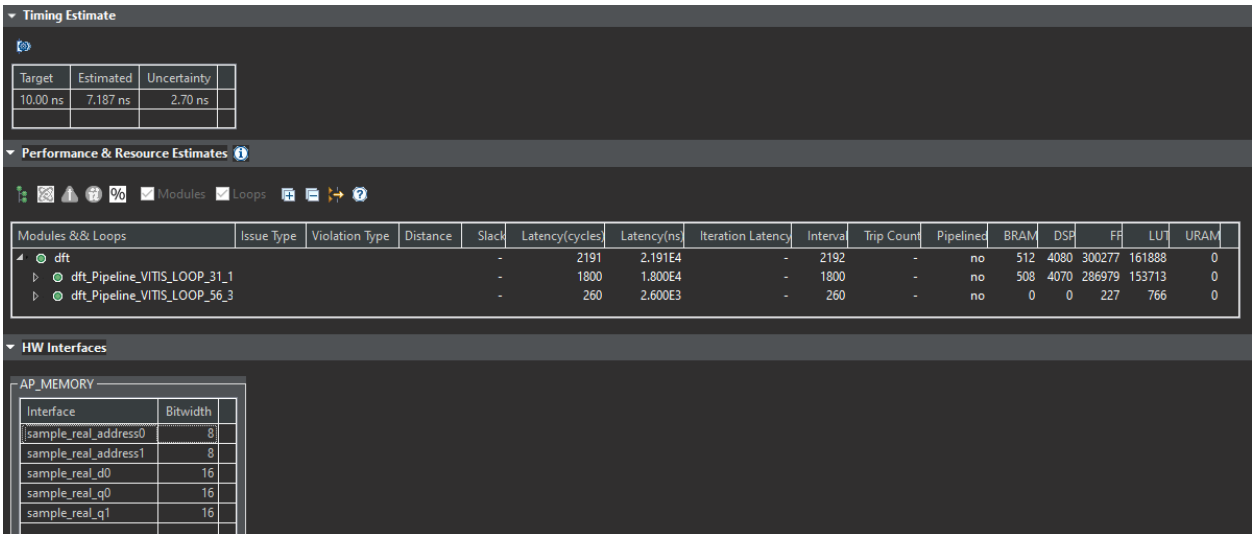
## Questions

***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.*

ANS: Using a CORDIC DFT can now calculate by being arranged in a matrix. When arranging sine cosine values in two different matrix one as a real part and another as a imaginary part, then multiplying by sampled data. One signicant change that is required is that we must be able to handle complex numbers. For a custom CORDIC there is a need for scaling and require signicant amount of resources – memory allocation and data types.

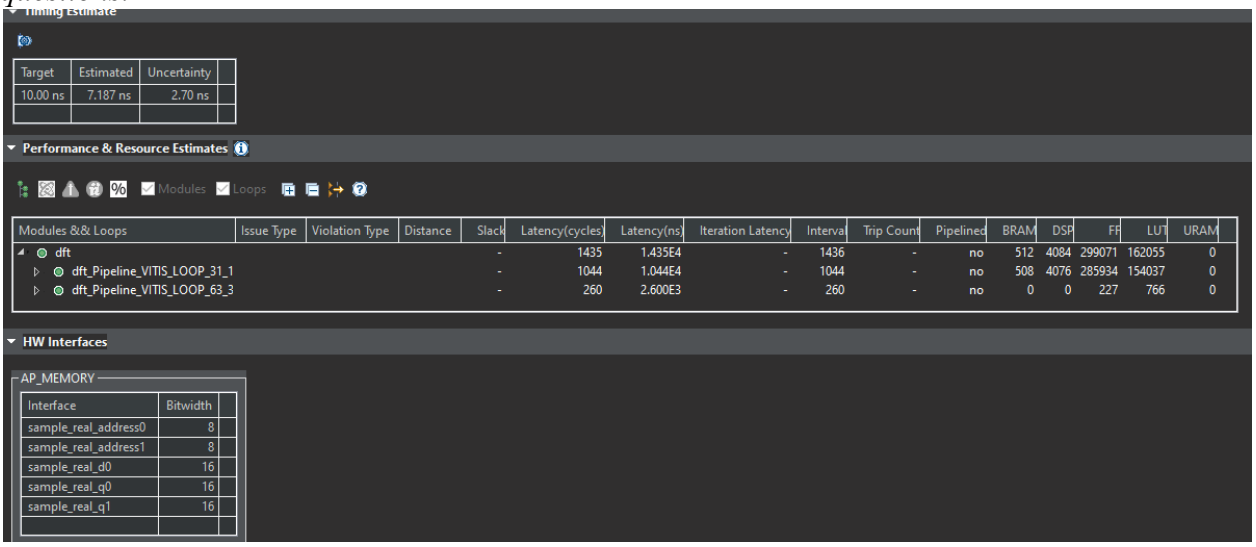
You basically do the inverse of calculating magnitude/phase by adding/subtracting phases so as to "accumulate" a rotation equal to the given phase. The accuracy of the result converges with each iteration: the more iterations you do, the more accurate it becomes, but will use more resources. When optimized for latency can also have minimal round-off errors, albeit with greater resource usage.

***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?*




ANS: With pipelining, the affect of these high-latency operations is less critical, since multiple executions of the loop can execute concurrently. The uses of the 32-bit  float type or the 16-bit half types rather than double possible solution is to reduce the precision of the computation. This is always a valuable technique when it can be applied, since it reduces the resources required for each operation, the memory required to store any values, and often reduces the latency of operations as well.

***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 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.*



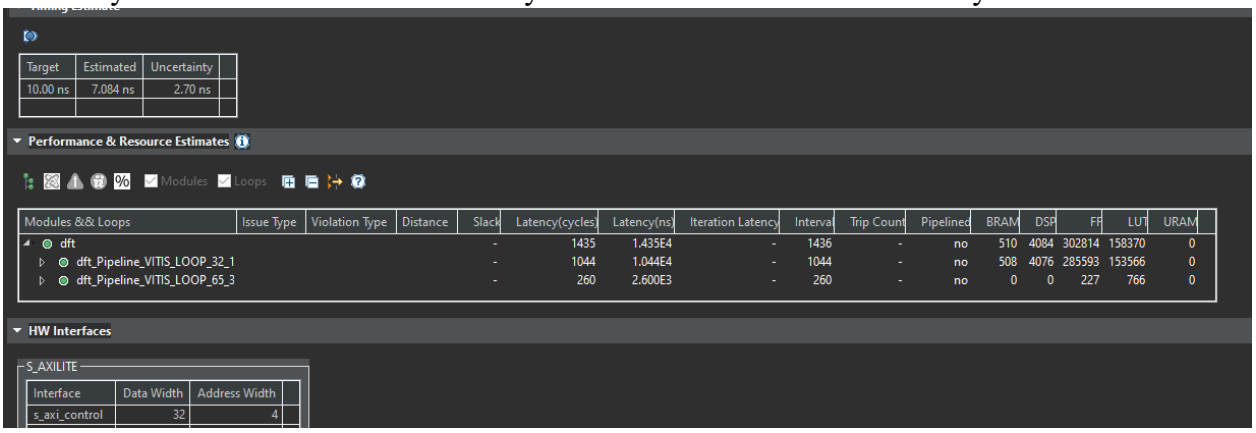

We can see the change in the latency while resources stay constant. Manual unrolling and removing bottlenecks allow the pragma optimization to perform better. It also helps performance when arrays are local in the for.

***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, BRAMs). What is the general trend in both cases? Which design would you select? Why?*

I didn't see much improvement overall using array partitioning

*#pragma HLS ARRAY_PARTITION dim=2 type=complete variable=sample_real*
*#pragma HLS ARRAY_PARTITION variable=sample_imag type=complete*

Over the manual unrolling and flattening #pragma and pipelining with a latency #pragma. Therefore, the fundamental trade off boils down to the required bandwidth versus the capacity. If throughput is the number one concern, all of the data would be stored in FFs. This would allow any element to be accessed as many times as it is needed each clock cycle.



| Target | Estimated | Uncertainty |
|---|---|---|
| 10.00 ns | 7.084 ns | 2.70 ns |

Performance & Resource Estimates

Modules Loops

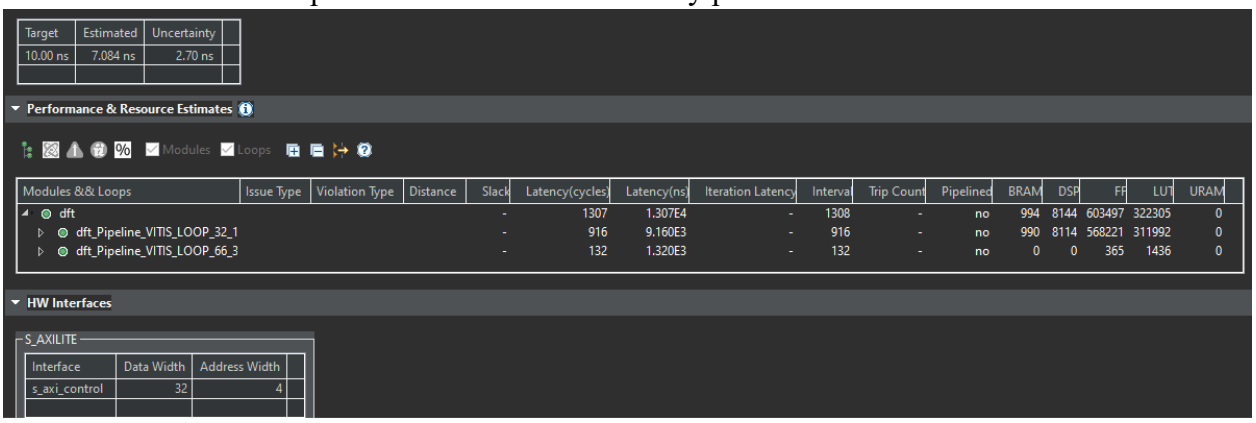
| Modules && Loops | Issue Type | Violation Type | Distance | Slack | Latency(cycles) | Latency(ns) | Iteration Latency | Interval | Trip Count | Pipelined | BRAM | DSP | FF | LUT | URAM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| dft | | | | - | 1435 | 1.435E4 | - | 1436 | - | no | 510 | 4084 | 302814 | 158370 | 0 |
| dft_Pipeline_VITIS_LOOP_32_1 | | | | - | 1044 | 1.044E4 | - | 1044 | - | no | 508 | 4076 | 285593 | 153566 | 0 |
| dft_Pipeline_VITIS_LOOP_65_3 | | | | - | 260 | 2.600E3 | - | 260 | - | no | 0 | 0 | 227 | 766 | 0 |

HW Interfaces

S_AXILITE

| Interface | Data Width | Address Width |
|---|---|---|
| s_axi_control | 32 | 4 |

Once I added unrolling to each loop we got a lower latency but our resources about doubled. I used #pragma HLS latency min=<> max=<> and receive a bit lower latency cycles and time. All times and resources performed better without array partition used.



| Target | Estimated | Uncertainty |
|---|---|---|
| 10.00 ns | 7.084 ns | 2.70 ns |

Performance & Resource Estimates

Modules Loops

| Modules && Loops | Issue Type | Violation Type | Distance | Slack | Latency(cycles) | Latency(ns) | Iteration Latency | Interval | Trip Count | Pipelined | BRAM | DSP | FF | LUT | URAM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| dft | | | | - | 1307 | 1.307E4 | - | 1308 | - | no | 994 | 8144 | 603497 | 322305 | 0 |
| dft_Pipeline_VITIS_LOOP_32_1 | | | | - | 916 | 9.160E3 | - | 916 | - | no | 990 | 8114 | 568221 | 311992 | 0 |
| dft_Pipeline_VITIS_LOOP_66_3 | | | | - | 132 | 1.320E3 | - | 132 | - | no | 0 | 0 | 365 | 1436 | 0 |

HW Interfaces

S_AXILITE

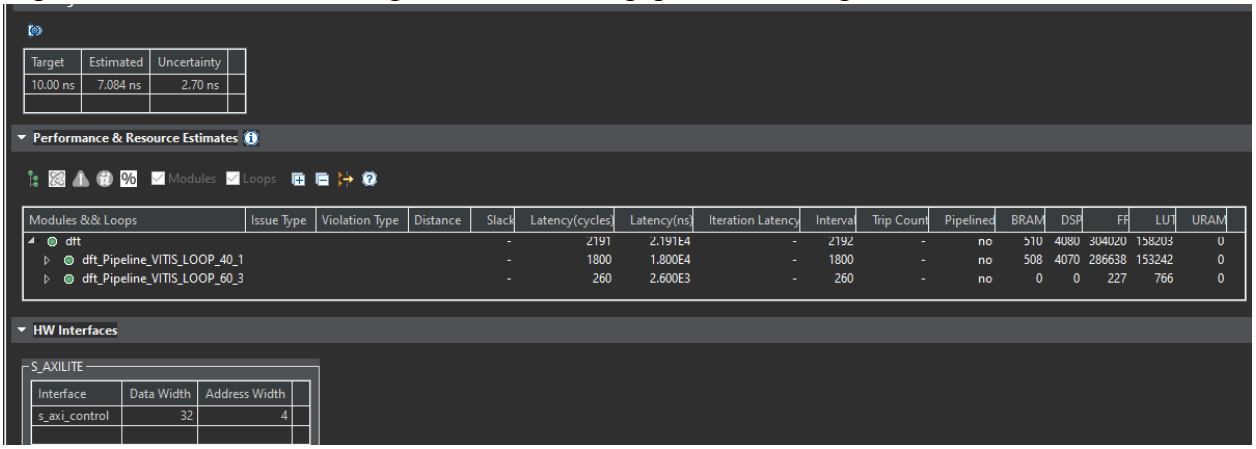
| Interface | Data Width | Address Width |
|---|---|---|
| s_axi_control | 32 | 4 |

Loops in the C/C++ functions are kept rolled by default. When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. It's seems best to a combination of loop pipeline with unrolling.

When the use of pragmas like these are used, they let more data be accessed in a single clock cycle. And while doing so they improve the throughput. The loop(s) can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle.

***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 HLS User Guide pages 145-154 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.*

ANS: Arrays involved in loop-based DATAFLOW optimizations are implemented as a RAM pingpong buffer channel. The DATAFLOW pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design.



| Target | Estimated | Uncertainty |
|---|---|---|
| 10.00 ns | 7.084 ns | 2.70 ns |

Performance & Resource Estimates

Modules Loops

| Modules && Loops | Issue Type | Violation Type | Distance | Slack | Latency(cycles) | Latency(ns) | Iteration Latency | Interval | Trip Count | Pipelined | BRAM | DSP | FF | LUT | URAM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| dft | | | | - | 2191 | 2.191E4 | - | 2192 | - | no | 510 | 4080 | 304020 | 158203 | 0 |
| dft_Pipeline_VITIS_LOOP_40_1 | | | | - | 1800 | 1.800E4 | - | 1800 | - | no | 508 | 4070 | 286638 | 153242 | 0 |
| dft_Pipeline_VITIS_LOOP_60_3 | | | | - | 260 | 2.600E3 | - | 260 | - | no | 0 | 0 | 227 | 766 | 0 |

HW Interfaces

S_AXILITE

| Interface | Data Width | Address Width |
|---|---|---|
| s_axi_control | 32 | 4 |

***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?*
ANS: After reviewing the output the code with the DATAFLOW is the best architecture. Dataflow allows you to pipeline multiple functions so that you can execute their instructions simultaneously on different sets of iterations. DATAFLOW optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function. Tradeoff is that this only effect the resources and the extra temp variables in the code.

***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?*