Patrick Hanrahan A53204304
Frank Chang A08362337
Ben Hobbs A113171489
WES 237C
11/8/2018

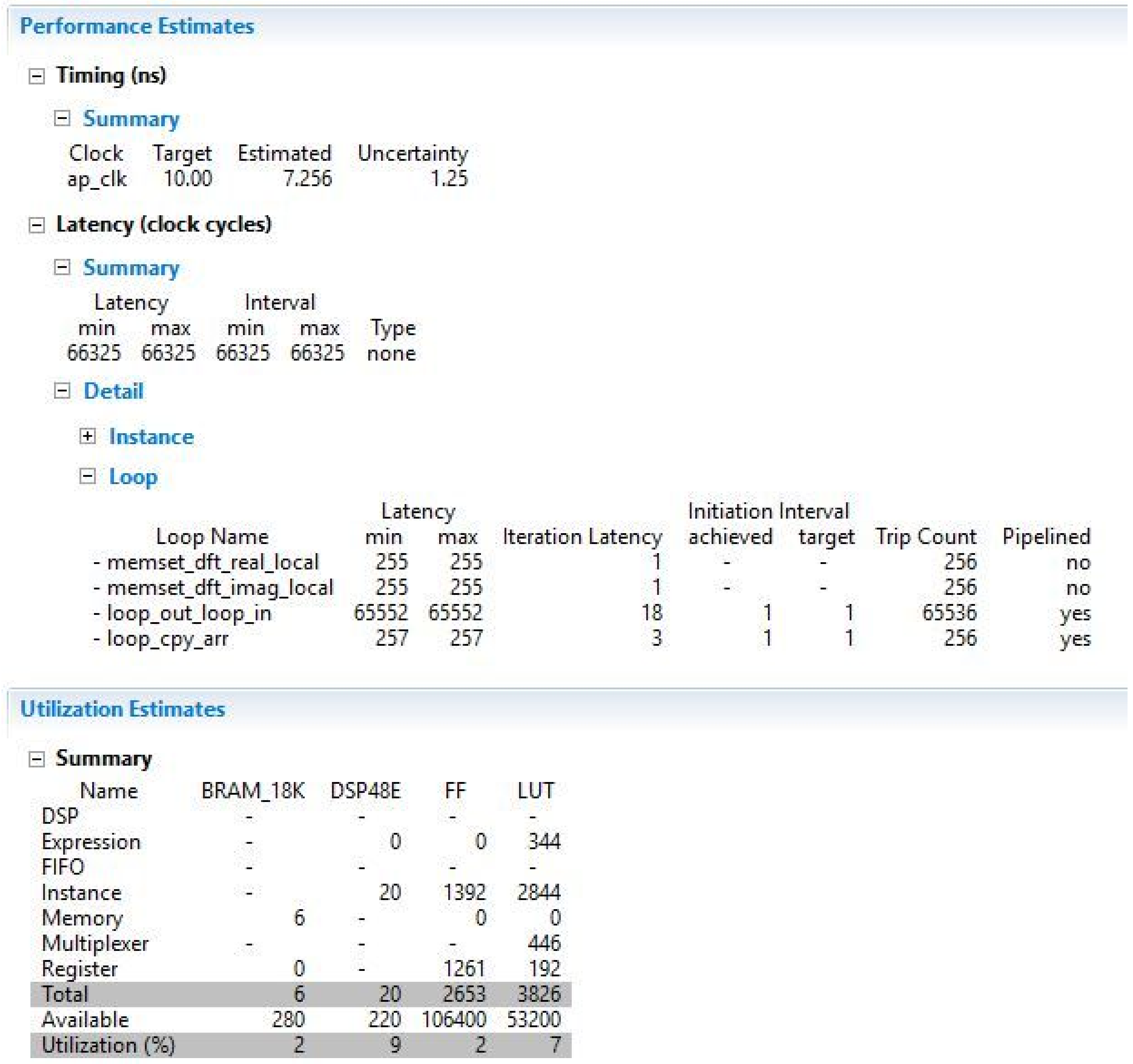
Project 3: Discrete Fourier Transform (DFT)

DEFINING THROUGHPUT AND DFT/SECOND

Moving forward, we will be working with several assumptions as we attempt to show certain optimizations are beneficial, certain optimizations does not help or to even characterize the performance.

We will be using an already optimized code as baseline to understand how changing certain parts to that code may benefit or hinder performance. Our optimized code can be found in QUESTION 6.

One optimization factor is throughput. The text mentions our optimization factor may not be a function of *latency* and *clock interval*. Let's look at an example below:


Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.256	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
66325	66325	66325	66325	none

⊟ Detail

⊞ Instance

⊟ Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- memset_dft_real_local	255	255	1	-	-	256	no
- memset_dft_imag_local	255	255	1	-	-	256	no
- loop_out_loop_in	65552	65552	18	1	1	65536	yes
- loop_cpy_arr	257	257	3	1	1	256	yes

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	344
FIFO	-	-	-	-
Instance	-	20	1392	2844
Memory	6	-	0	0
Multiplexer	-	-	-	446
Register	0	-	1261	192
Total	6	20	2653	3826
Available	280	220	106400	53200
Utilization (%)	2	9	2	7

Figure 1 - DFT256, optimized and synthesized version.

Figure 1 shows our optimized and synthesized version.

- *memset_dft_real_local* and *memset_dft_imag_local* creating two sets of function scoped arrays to store the results.
- *loop_out_loop_in* pair of loops shows a pipelined loop that does the actual DFT computation.
- *loop_cpy_arr* shows the copy of two locally scoped arrays to an output array.

All these components are considered part of the DFT operation and without any one of these three components we would not be able to successfully perform a (256 point) DFT. We will then justify calling our throughput as number of DFT operations per second:

$$THROUGHPUT = 1/(LATENCY * ESTIMATED\ CLK)$$

When we say throughput in this text we are referring to number of DFT operations per second. A throughput on a DFT256 case of 2.1KHz means the FPGA will perform 2,100 256-point DFT operations per second.

QUESTION 1: CORDIC COMPARISON

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

A. Code changes required to implement a cordic:
   a. A CORDIC taking phase/radius to x/y (effectively cos(phase) and sin(phase)) would need to be implemented.
   b. Translate the cos_coefficients_tabel and sin_coefficients_table index into actual angle to be supplied to the cordic phase. Radius is 1 (no gain).
   c. We would then replace the cos_coefficients_table and sin_coefficients_table with the cordic x and y value (respectively).
   d. We would also need to be cautious of fixed point arithmetic and determine the best bit length to represent the code based input/output to avoid overflow.
   e. The rest of the code would be unchanged.

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?

A. Yes, changing the accuracy of the CORDIC core would change hardware resource usage because you are using smaller bit length to represent your accuracy.

How would it affect the performance?

A. The performance would increase (throughput) at the cost of accuracy. Less bit length to represent our data means a faster computation on each arithmetic operation. Also, the CORDIC core would take less time to hone in on the X and Y values (less iterations because less accuracy).

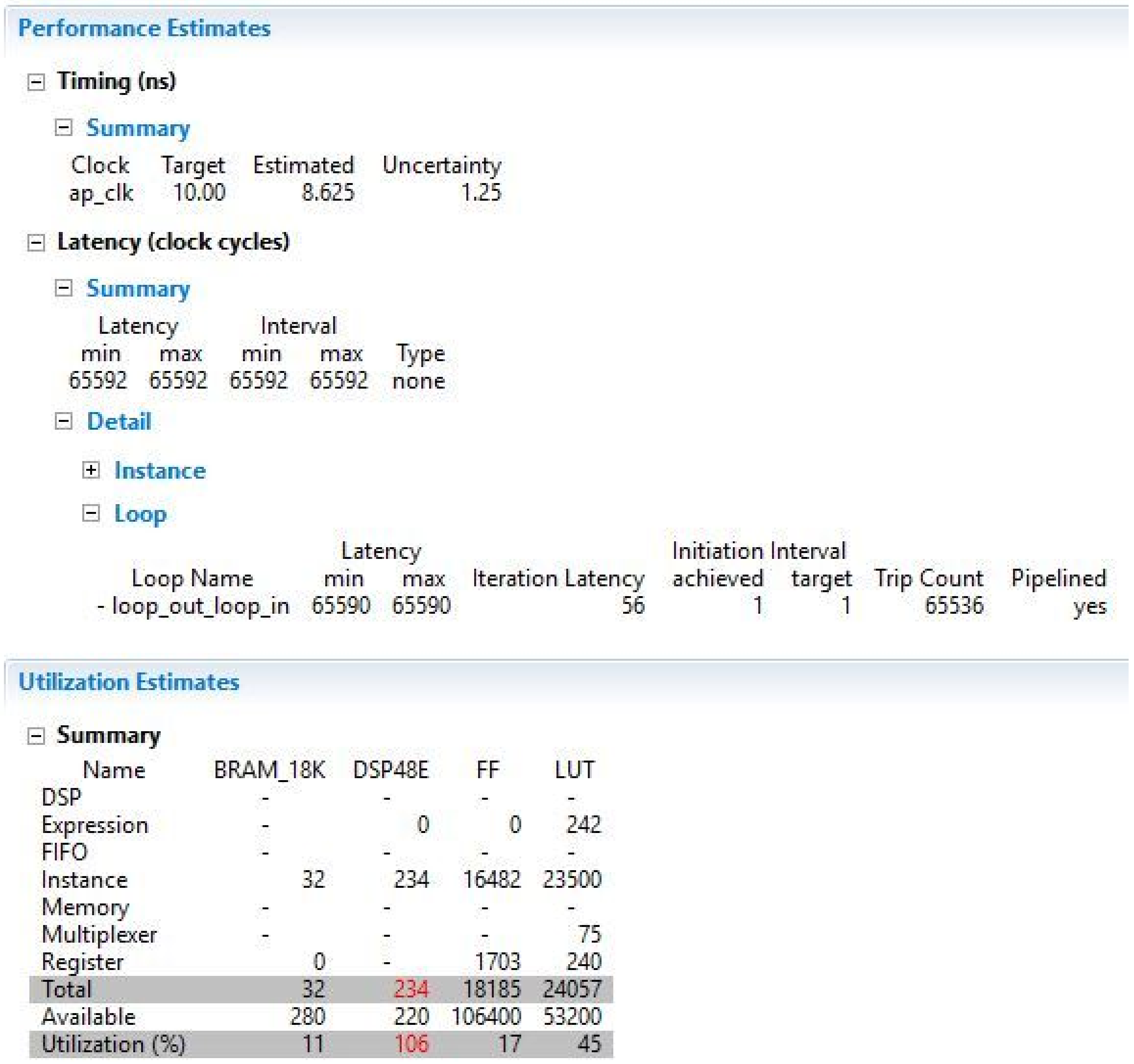
QUESTION 2: LOOK UP TABLE:

Moving forward with this question we make the following assumptions:

- We will use an optimized version of our code to compare the effects cos and sin has on our performance (defined by throughput and resource usage). We will modify the optimized code to see how introducing trigonometric math operations from the math.h library will deter the performance.
- Our optimized code includes:
    - Inner and outer loop swap.
    - Two input arrays for real and imag values and two output arrays for real and imag dft results.
    - Pipelining inner loop.
- We will be evaluating the DFT256 case, results for the DFT1024 can be extrapolated from the DFT256 case.

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?

A. Using the trigonometric functions, we have a throughput of 1.76KHz however we red-lined our DSP48E usage at 106%, more than what the board can deliver. By performing a loop-swap (swapping indices of inner and outer loop) the throughput drops to 0.14KHz. The resource usage has been resolved however we have a 17.98ns expected clock which is greater than our 10ns target clock. This is because the trigonometric function and loop-swap introduced a latency of 6 clock cycles for the mathematical operation rendering our pipeline II of 1 to have no effect on the loop.



Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.625	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
65592	65592	65592	65592	none

⊟ Detail

⊞ Instance

⊟ Loop

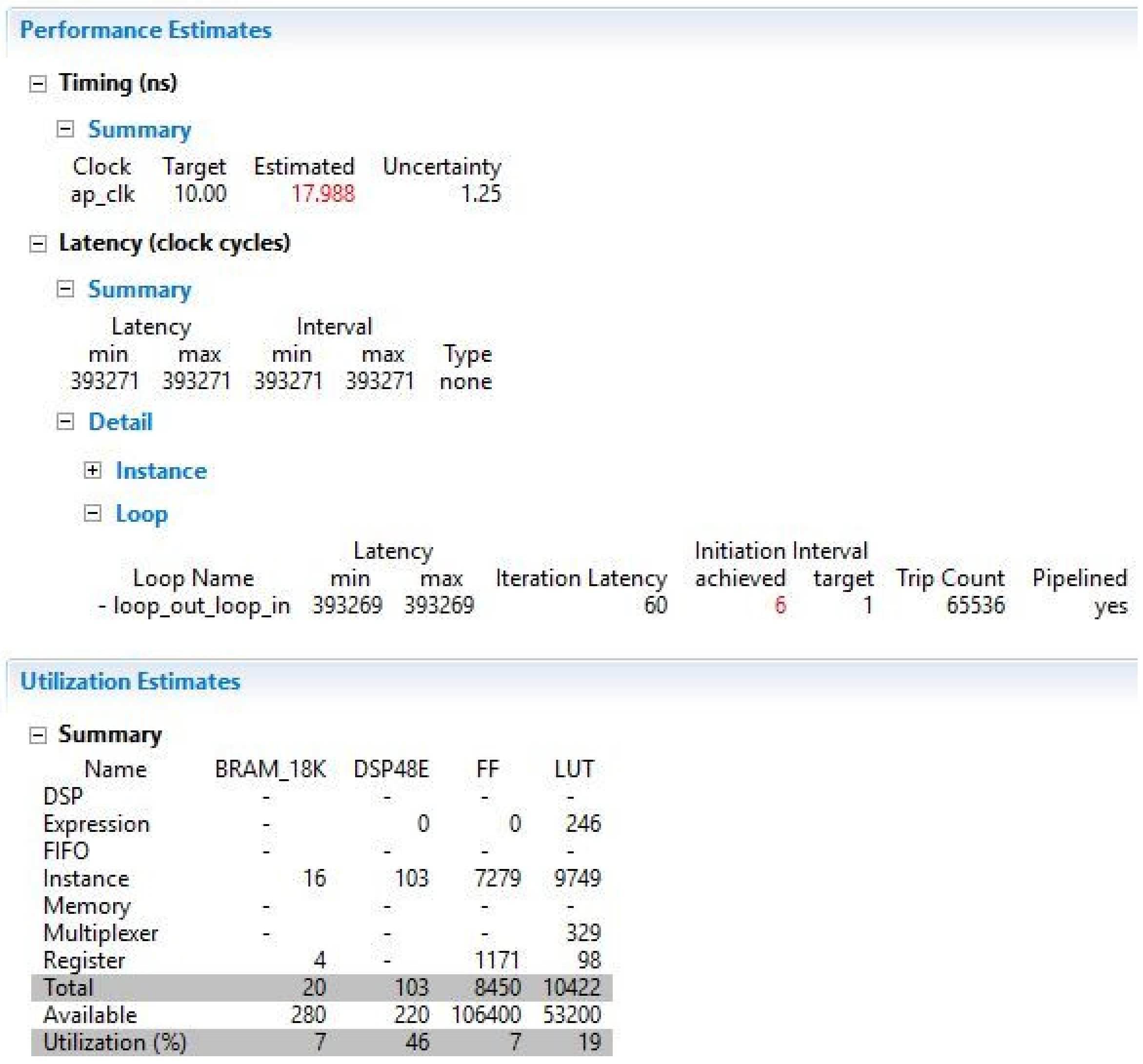
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop_out_loop_in	65590	65590	56	1	1	65536	yes

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	242
FIFO	-	-	-	-
Instance	32	234	16482	23500
Memory	-	-	-	-
Multiplexer	-	-	-	75
Register	0	-	1703	240
Total	32	234	18185	24057
Available	280	220	106400	53200
Utilization (%)	11	106	17	45

Figure 2: Performance deviating from optimized code using trigonometric function.



Performance Estimates

⊟ **Timing (ns)**

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	17.988	1.25

⊟ **Latency (clock cycles)**

⊟ Summary

Latency		Interval		
min	max	min	max	Type
393271	393271	393271	393271	none

⊟ Detail

⊞ Instance

⊟ Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop_out_loop_in	393269	393269	60	6	1	65536	yes

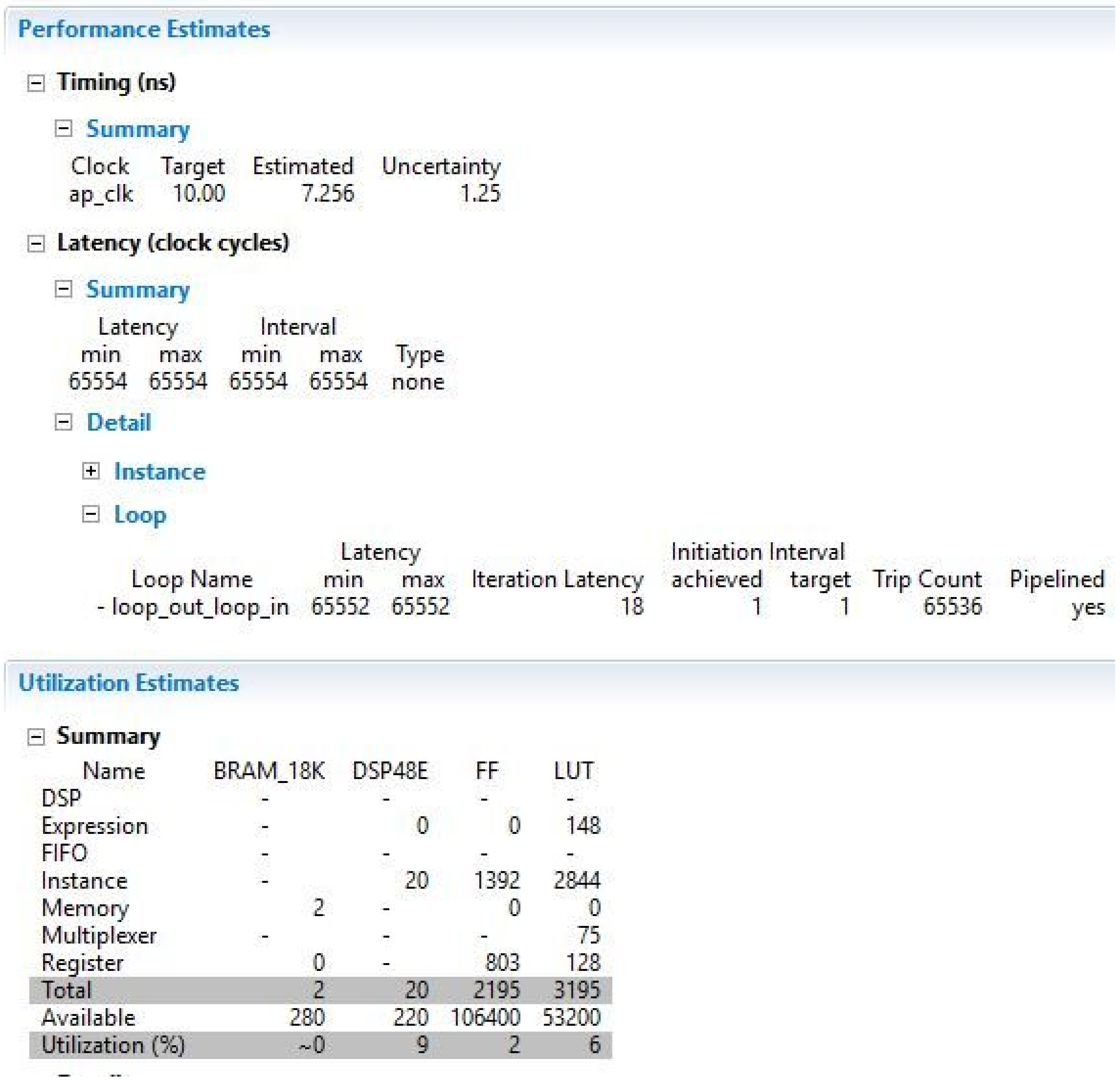
Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	246
FIFO	-	-	-	-
Instance	16	103	7279	9749
Memory	-	-	-	-
Multiplexer	-	-	-	329
Register	4	-	1171	98
Total	20	103	8450	10422
Available	280	220	106400	53200
Utilization (%)	7	46	7	19

Figure 3: Performance deviating from optimized code using trigonometric function and swapping inner and outer loop.

The optimized case of using the LUT yielded a 2.1Khz with no resource violation. Using the LUT is essential in implementing a fast DFT.



Performance Estimates

⊟ Timing (ns)

⊟ Summary

```
Clock   Target  Estimated  Uncertainty
ap_clk   10.00      7.256         1.25
```

⊟ Latency (clock cycles)

⊟ Summary

```
   Latency         Interval
 min    max     min    max    Type
65554  65554  65554  65554   none
```

⊟ Detail

⊞ Instance

⊟ Loop

```
                          Latency                        Initiation Interval
    Loop Name           min    max   Iteration Latency   achieved   target   Trip Count   Pipelined
 - loop_out_loop_in    65552  65552                 18          1        1        65536         yes
```

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	148
FIFO	-	-	-	-
Instance	-	20	1392	2844
Memory	2	-	0	0
Multiplexer	-	-	-	75
Register	0	-	803	128
Total	2	20	2195	3195
Available	280	220	106400	53200
Utilization (%)	~0	9	2	6

Figure 4: Performance using optimized implementation resolving any resource issues.

Figure #	Through-put	LUT	FF	DSP48E	BRAM
2	1.76Khz	45%	17%	106%	11%
3	0.14KHz	19%	7%	46%	7%
4	2.1KHz	6%	2%	9%	0%

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

A. The lookup table is in place to represent the cos/sin operations. Increasing the number of DFT point size means we have a finer resolution of frequency bins we are looking at. Take for example going from DFT256 to DFT1024 means we have increased our lookup table by a factor of 4. That also means the resources utilized to represent the LUT would have increase by a factor of 4 as well.

QUESTION 3: SEPARATE I/O

Moving forward we study the DFT256 case unless otherwise noted.

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?

A. The original function took in two arrays (real_sample and imag_sample) which were the complex input values. The DFT results were stored on dft_real and dft_imag which had a scope of the function. At the end of the function, dft_real and dft_imag were copied onto real_sample and imag_sample (respectively). The new function would take in four arrays real_sample, imag_sample, dft_real and dft_imag. The results would be written to the dft_real and dft_imag directly.

   Given the method we used on both implementation, it did not change our optimization method. We would have to allocate a few operations at the end of each DFT computation to copy results from one array to another (twice).

How does it change the performance?

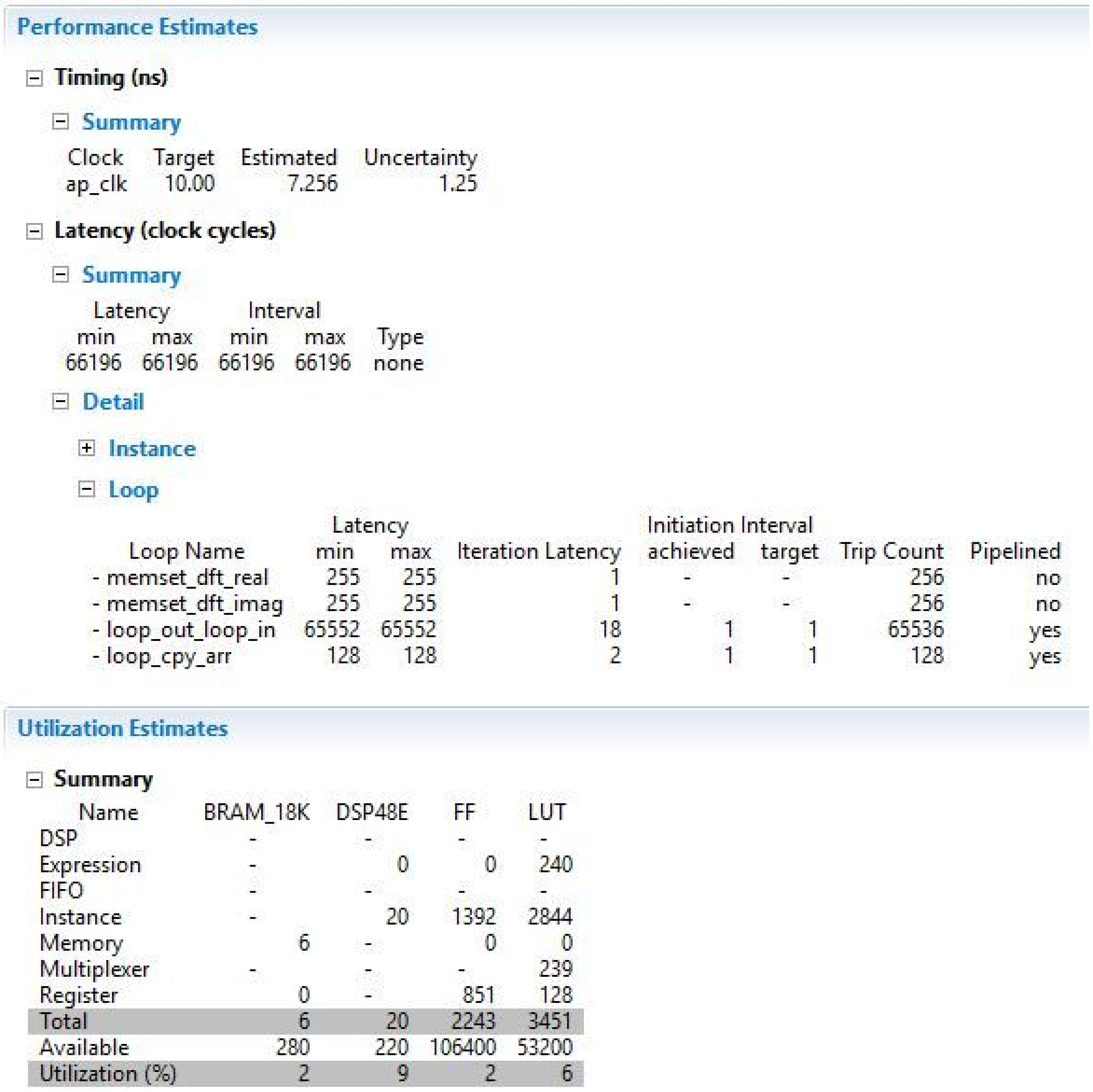
A. The performance of second implementation would be slightly better because at the end of each function on the first implementation we would copy the data from both dft_real and dft_imag array back to real_sample and imag_sample array.

The second implementation passed the values directly to the output array skipping the steps of copying.

The first method had a throughput of 2.08Khz while the second method (optimized method) had a throughput of 2.10Khz. The difference is found in the array copy loop at the end of the DFT loop where it uses 128 iterations (DFT256 case) to copy two 256 sized arrays. This loop has also been unrolled and pipelined for optimization.

What about the area results?

A. Surprisingly, the resulting area (resources) were less. The input arrays did not require additional resources to implement, at least not on the FPGA resources being monitored.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.256	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
66196	66196	66196	66196	none

Detail

Instance

Loop

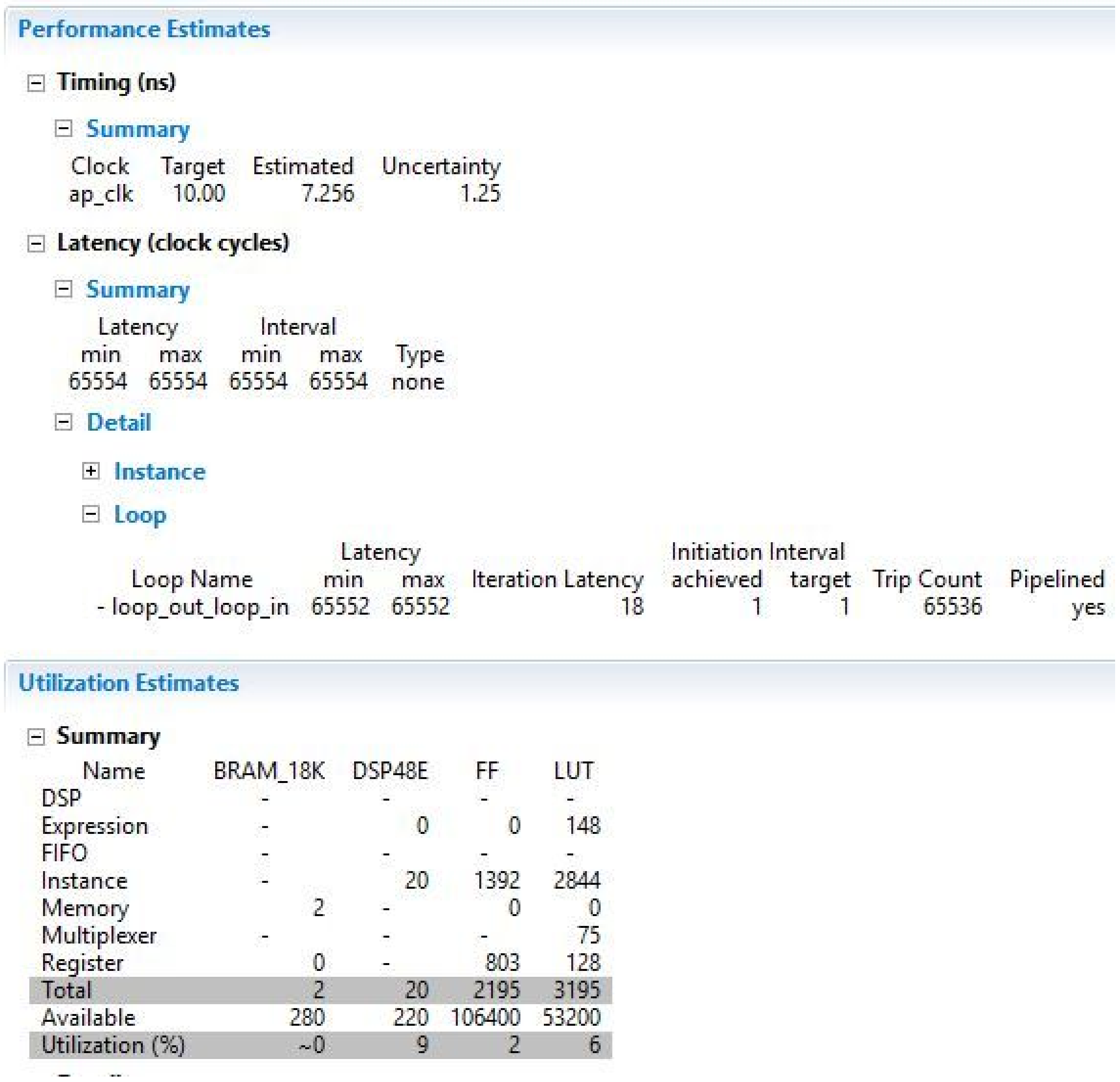
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- memset_dft_real	255	255	1	-	-	256	no
- memset_dft_imag	255	255	1	-	-	256	no
- loop_out_loop_in	65552	65552	18	1	1	65536	yes
- loop_cpy_arr	128	128	2	1	1	128	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	240
FIFO	-	-	-	-
Instance	-	20	1392	2844
Memory	6	-	0	0
Multiplexer	-	-	-	239
Register	0	-	851	128
Total	6	20	2243	3451
Available	280	220	106400	53200
Utilization (%)	2	9	2	6

Figure 5: Using two arrays (real_sample and imag_sample). Requires array copy operation at the end of DFT operation.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.256	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
65554	65554	65554	65554	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop_out_loop_in	65552	65552	18	1	1	65536	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	148
FIFO	-	-	-	-
Instance	-	20	1392	2844
Memory	2	-	0	0
Multiplexer	-	-	-	75
Register	0	-	803	128
Total	2	20	2195	3195
Available	280	220	106400	53200
Utilization (%)	~0	9	2	6

Figure 6: Using four arrays (real_sample, imag_sample, DFT_real, DFT_imag). Does not require array copy at the end of DFT operation.

QUESTION 4: LOOP UNROLLING/ARRAY PARTITIONING

Moving forward with this question we make the following assumptions:

- We will use an optimized version of our code to compare the effects array partitioning and unrolling has on our performance (defined by throughput and resource usage). We will modify the optimized code to see how introducing optimization will deter the performance.

- Our optimized code includes:
    - Inner and outer loop swap.
    - Two input arrays for real and imag values and two output arrays for real and imag dft results.
    - Pipelining inner loop.
- We will be evaluating the DFT256 case, results for the DFT1024 can be extrapolated from the DFT256 case.

Study the effects of loop unrolling and array partitioning on the performance and area. What is the relationship between array partitioning and loop unrolling?

A. Array partitioning will divide the array into several memory partition each with their own access ports. Depending on where the data lies on the array and how it is partitioned, there can be simultaneous access to the array if partitioned correctly (ie cyclic factor matching unrolling factor).

   To benefit from array partitioning, we would need to unroll the loop the same number of times we partition (cyclically) the array. Partitioning without unrolling does not benefit the architecture as the “choke” in the process is still limited by the availability of the element being accessed on the array.

Does it help to perform one without the other?

A. Greatest benefit is to perform both partitioning (cyclically) and unrolling at the same factor..

Plot the performance in terms of number of matrix vector multiply operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for area (showing LUTs, FFs, DSP blocks, BRAMs). What is the general trend in both cases? Which design would you select? Why?

A. Based on the plots below, our best architecture is still the no partitioning and no unrolling even though we expected an equal number of partitioning and unrolling would be ideal. Any numbers below in the red are indicators that the clock is >10ns.

The partition and unrolling by a factor of 15 (which is partitioning/unrolling 16 times) outperforms any other factors based on throughput. However the heavy unrolling takes a penalty in resources.

Our optimized case based on this study would be the no partitioning and no unrolling for a light-weight (resource friendly) and fast DFT even though we expect better performance with equal factor partitioning and unrolling.

Throughput (also number of DFT/s)

Part.\Unroll	**0**	**1**	**3**	**7**	**15**
0	2.1kHz	2.1kHz	0.32kHz	0.61kHz	1.07Khz
1	2.1kHz	0.32kHz	0.322kHz	0.611kHz	1.06kHz
3	0.21kHz	0.21kHz	0.69kHz	0.28kHz	0.67kHz
7	0.20kHz	0.20kHz	0.20kHz	0.148kHz	0.20kHz
15	0.22kHz	.22kHz	0.61kHz	1.28kHz	2.7kHz

Red indicates we have a >10ns expected clock.

LUT

Part.\Unroll	0	1	3	7	15
0	6%	6%	3%	10%	12%
1	6%	3%	3%	10%	12%
3	3%	3%	6%	14%	19%
7	4%	4%	13%	18%	43%

15	5%	5%	13%	39%	37%

FFs

Part.\Unroll	0	1	3	7	15
0	2%	2%	1%	4%	5%
1	2%	1%	1%	4%	5%
3	1%	1%	3%	7%	8%
7	1%	1%	5%	8%	11%
15	2%	2%	7%	16%	18

DSP

Part.\Unroll	0	1	3	7	15
0	9%	9%	3%	10%	11%
1	9%	3%	3%	10%	11%
3	3%	3%	10%	15%	34%
7	3%	3%	8%	24%	30%
15	3%	3%	10%	34%	45%

BRAM

Part.\Unroll	0	1	3	7	15
0	0%	0%	0%	0%	0%

1	0%	0%	0%	0%	0%
3	2%	2%	2%	2%	2%
7	5%	5%	5%	5%	5%
15	0%	0%	0%	0%	0%

QUESTION 5: DATAFLOW

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?

In our case throughput was not as good when compared to our "best" implementation, however, our baseline was already somewhat optimized. When compared to a zero optimization implementation(no pragmas, and no LUT), we saw a dramatic increase in throughput, as seen in the table directly below. This confirms that using pragma dataflow increases performance by coarsely pipelining the implementation.

TYPE:	CLOCK	LATENCY	THROUGHPUT
DATAFLOW NO PRAGMAS	7.42	557314	0.2kHz
"BASELINE"	7.256	65554	2.1kHz
NO DATAFLOW NO PRAGMAS NO LUT	8.63	5177858	0.03kHz

How much does your design use resources?

TYPE:	BRAM	DSP	FF	LUT
DATAFLOW	2%	14%	2%	9%
"BASELINE"	11%	106%	17%	45%
"BEST"	0%	9%	2%	6%

What about BRAM usage?

When implementing BRAM, we saw an increase in BRAM usage from 0% in our “best” design to 2% in our “dataflow” design. This is still of course smaller than in our “baseline” design where we had no optimizations and used 11% BRAM. This makes sense for pragam dataflow to cause an increase in BRAM when compared to a pipelined design because we are more coarsely pipelining the code into larger chunks. This requires larger sections of BRAM to store these large chunks so that they can be computed at the same time for a latency equal to the “slowest” stage.

Please describe your architecture with figures on your report. (Make sure to add dataflow pragma on your top function.)

A. In our code, we separated our dft function into three stages, or three separate loops, then we called dataflow to string them together in such a way that improves performance. Rather than waiting for each stage to finish, we creating new variables for the output and input of each stage, allowing the DFT to complete each section at the same time to reduce total latency. This makes the latency dependent on the slowest computation/stage.

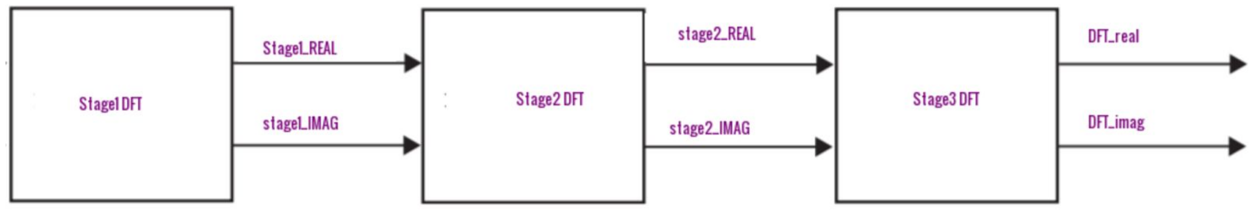
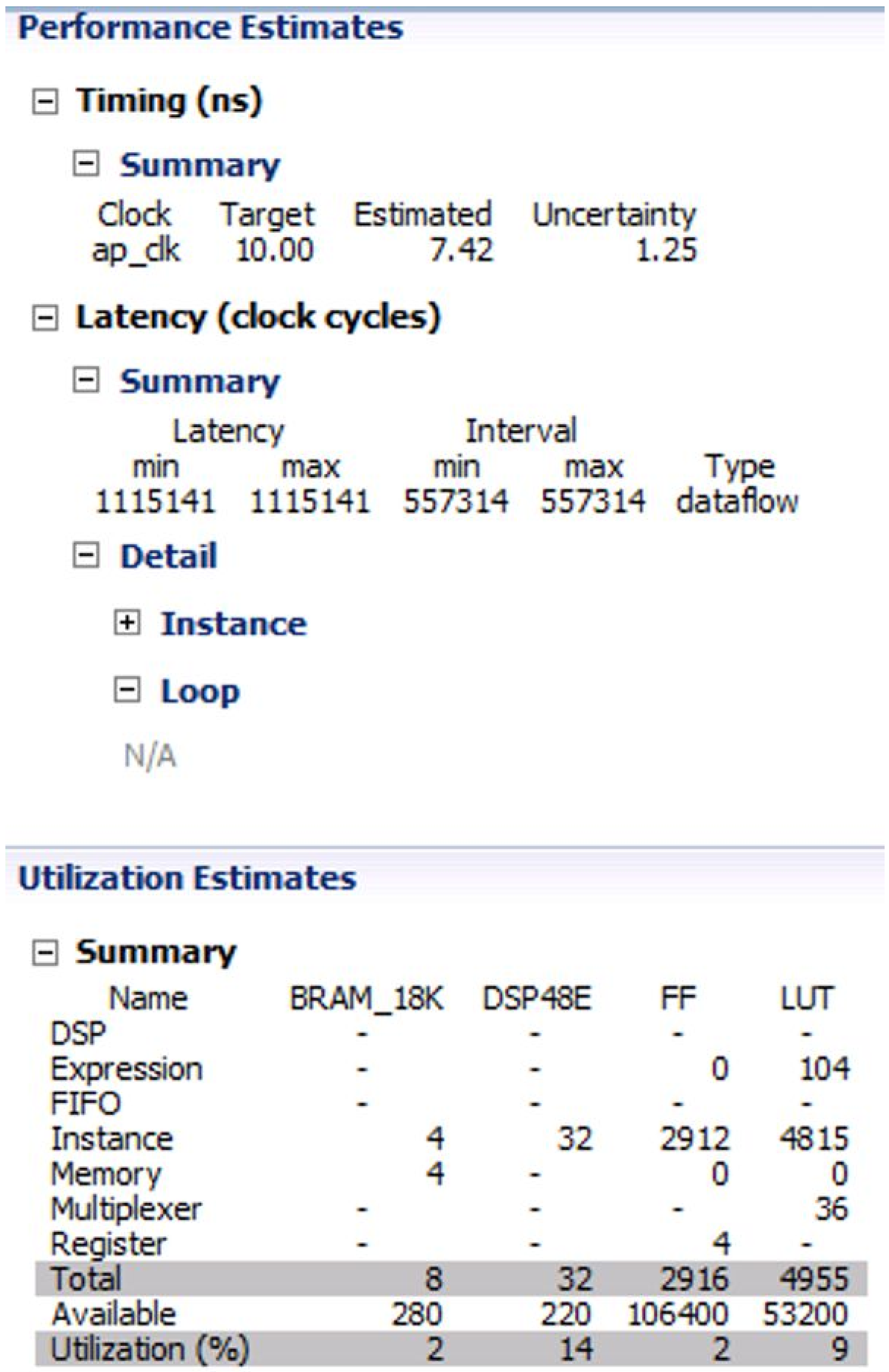


Figure 7 - Dataflow architecture.



Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.42	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
1115141	1115141	557314	557314	dataflow

⊟ Detail

⊞ Instance

⊟ Loop

N/A

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	104
FIFO	-	-	-	-
Instance	4	32	2912	4815
Memory	4	-	0	0
Multiplexer	-	-	-	36
Register	-	-	4	-
Total	8	32	2916	4955
Available	280	220	106400	53200
Utilization (%)	2	14	2	9

Figure 8 - Dataflow performance/resource analysis.

QUESTION 6: BEST ARCHITECTURE

Moving forward we assume the DFT256 case.

Briefly describe your "best" architecture. In what way is it the best?

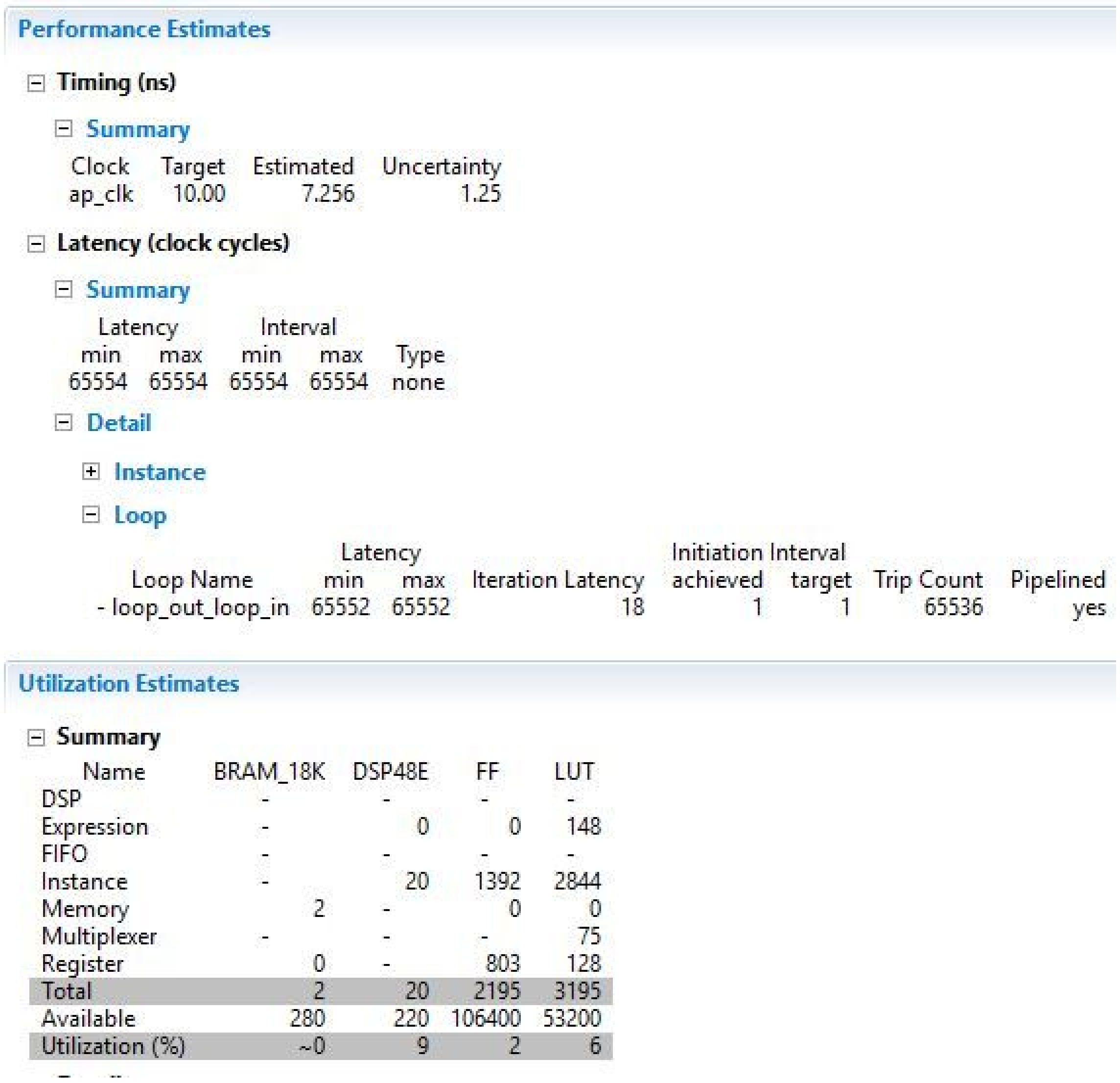
A. Our optimization attempted to maximize our throughput without significantly compromising our resource usage. Based on our results for question 4, no partitioning and unrolling was our best condition (2.1KHz). Unrolling and partitioning by a factor of 15 shows a higher throughput (2.7KHz) but at the cost of resource (>45% LUT).

What optimizations did you use to obtain this result?

A. Our best architecture is composed of:
    a. Swap the inner and outer to enable efficient pipelining.
    b. Pipeline the inner loop only.
    c. Use cos and sin coefficient LUT.

What is trade off you consider for the best architecture?

A. Our best architecture did not utilize array partitioning or loop unrolling even though performing these two correctly would benefit our design. As of now, we have designed a light-weight (resource friendly) and high throughput design (2.1KHz) by loop swapping and pipelining the inner loop.



Performance Estimates

⊟ **Timing (ns)**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.256	1.25

⊟ **Latency (clock cycles)**

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
65554	65554	65554	65554	none

⊟ **Detail**

⊞ **Instance**

⊟ **Loop**

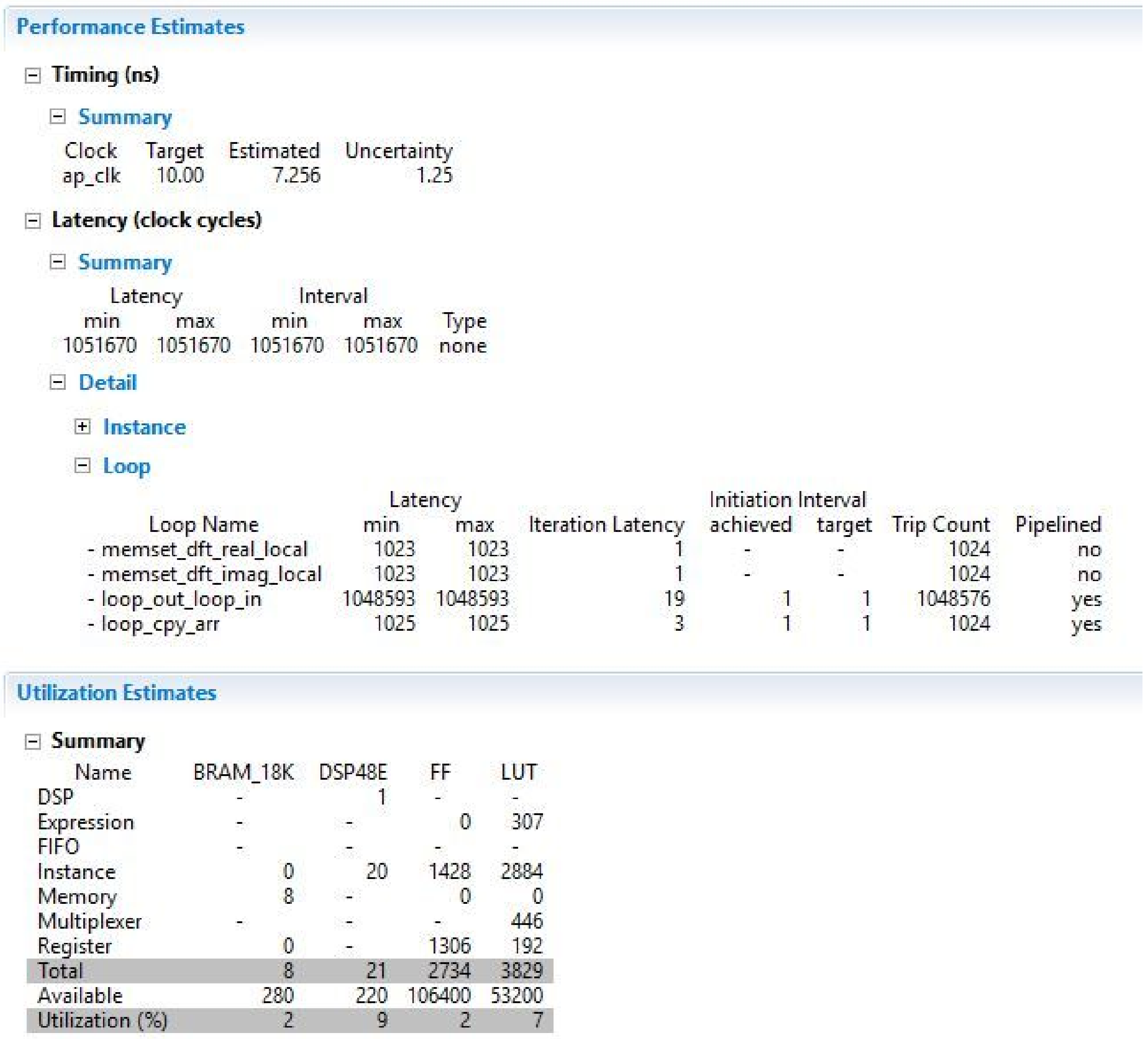
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop_out_loop_in	65552	65552	18	1	1	65536	yes

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	148
FIFO	-	-	-	-
Instance	-	20	1392	2844
Memory	2	-	0	0
Multiplexer	-	-	-	75
Register	0	-	803	128
Total	2	20	2195	3195
Available	280	220	106400	53200
Utilization (%)	~0	9	2	6

Figure 9 - Optimized for DFT256 simulation, throughput of 2.1KHz.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.256	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1051670	1051670	1051670	1051670	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- memset_dft_real_local	1023	1023	1	-	-	1024	no
- memset_dft_imag_local	1023	1023	1	-	-	1024	no
- loop_out_loop_in	1048593	1048593	19	1	1	1048576	yes
- loop_cpy_arr	1025	1025	3	1	1	1024	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	307
FIFO	-	-	-	-
Instance	0	20	1428	2884
Memory	8	-	0	0
Multiplexer	-	-	-	446
Register	0	-	1306	192
Total	8	21	2734	3829
Available	280	220	106400	53200
Utilization (%)	2	9	2	7

Figure 10- Optimized for DFT1024 simulation, throughput of 0.13KHz.

PYNQ DEMO

No demo available. Using the synthesized version of dft256_best, we were unable to receive results back from the DFT IP. Some techniques we have tried:

- Leveraged Lab2's pynq code.
- Ensure the last bit were set.
- Buffered our input and output DMA arrays. Stored out input array from the DMA on a temporary array before utilizing it for the DFT. Stored our output results on a temporary array before pushing to the DMA array.

ARCHITECTURE FOLDER

1. dft256_baseline
    - Baseline code for comparison.
2. dft256_optmized1
    - Trigonometric math implementation.
3. dft256_optmized2
    - Trigonometric math implementation with loop swap.
4. dft256_optmized3
    - Input/output arrays separated.
5. dft256_optmized4
    - Input/output arrays not separated
6. dft256_optmized5
    - Array partitioning.
7. dft256_optmized6
    - Best architecture before synthesis.
8. dft256_best
    - Best architecture.
9. dFT256_dataflow
    - Dataflow architecture.
10. Report.pdf
    - Report writeup.
11. Demo
    - PYNQ code, not functional.