CSSE4010 – 2022 semester 2 – Project

Custom Computing Hardware Architectures for a Digital Multi-Beamformer

Kuang Sheng

45752720

Lab Session attend: Friday 12-2pm

Submission date: 07/11/2022

1. Introduction

This research investigates the use of approximate algorithms and approximate hardware in designing low complexity accelerators with an application in antenna array based digital beamforming. However, the research focused on developing FPGA based hardware design for the proposed 8-point spatial Discrete Fourier Transform (DFT) only, rather than putting too much effort into the specific application.

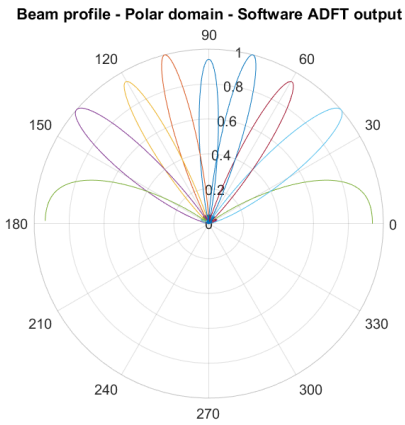
Nowadays as the dimension of CMOS being reduced, it is difficult to improve the circuit performance. In the area of digital system design, the implementation of adders and multipliers have always played an important role in determining the circuit performance and power consumption. The multiplier complexity of ideal N-point DFT is N^2 and that of FFT is Nlog(N). If the complexity of both multiplier and adder could be reduced further, the circuit would save more resources and have a stronger performance in timing.

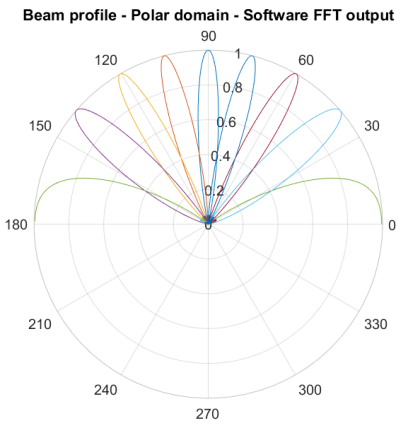
Therefore, approximate algorithm is introduced as a new way to save power and resources, improving performance with a trade-off of acceptable accuracy. The implied computations are performed without multiplications so that the multiplier complexity would be reduced to 0. Though the presence the computational error, it is acceptable in many applications such as antenna array based digital beamforming. Furthermore, the approximate adders were being implemented to replace the full adder to reduce the adder complexity of the system.

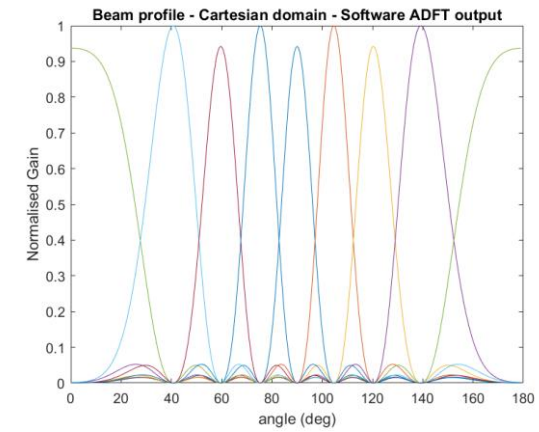
2. ADFT Algorithms in Software

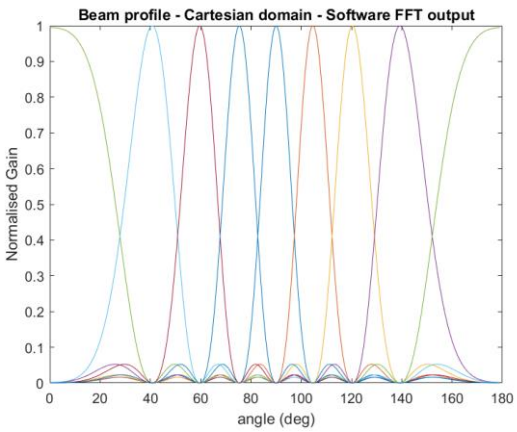
Before constructing the hardware architecture for ADFT, a software version was implemented to examine the functionality of the ADFT algorithm by comparing with the Matlab FFT function. The ideal FFT operation in Matlab was replaced with an ADFT algorithm proposed in [1].

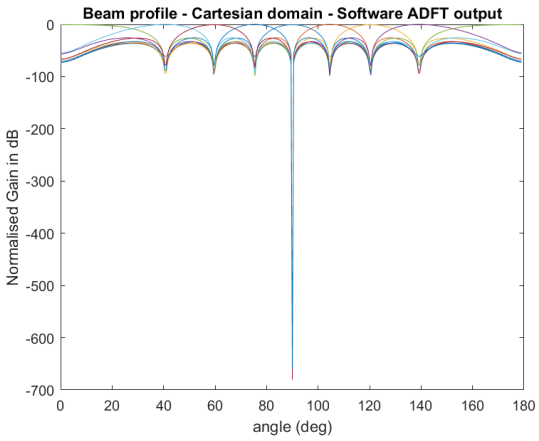
By visually comparing the beam patterns from the ADFT algorithm in software with the equivalent from the FFT algorithm, the operation of the ADFT algorithm can be verified.

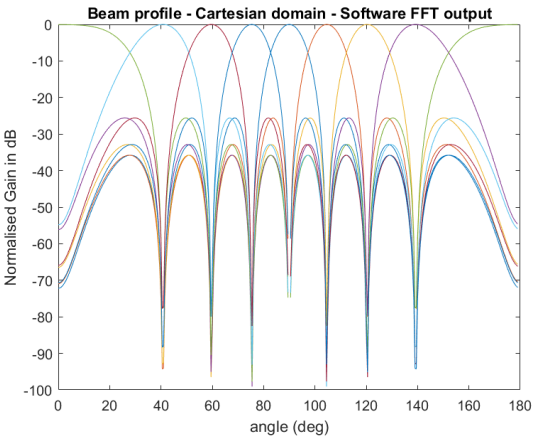








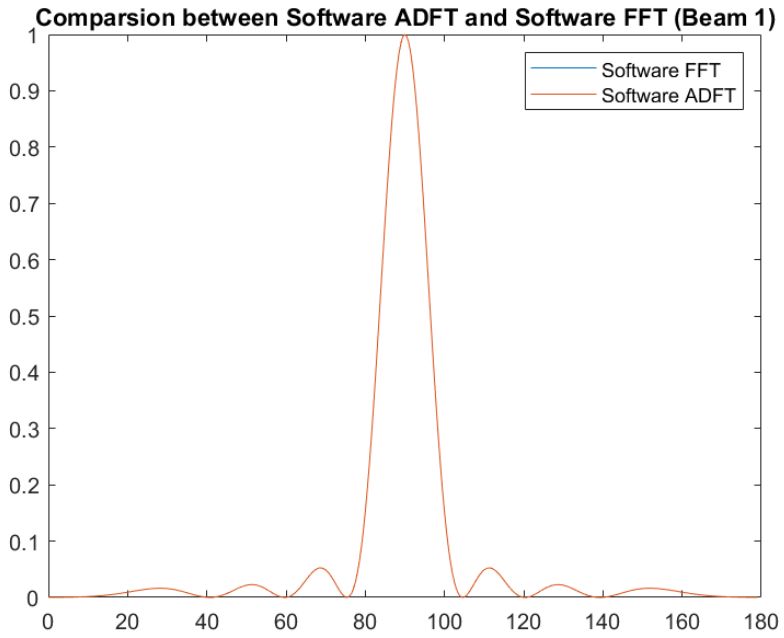




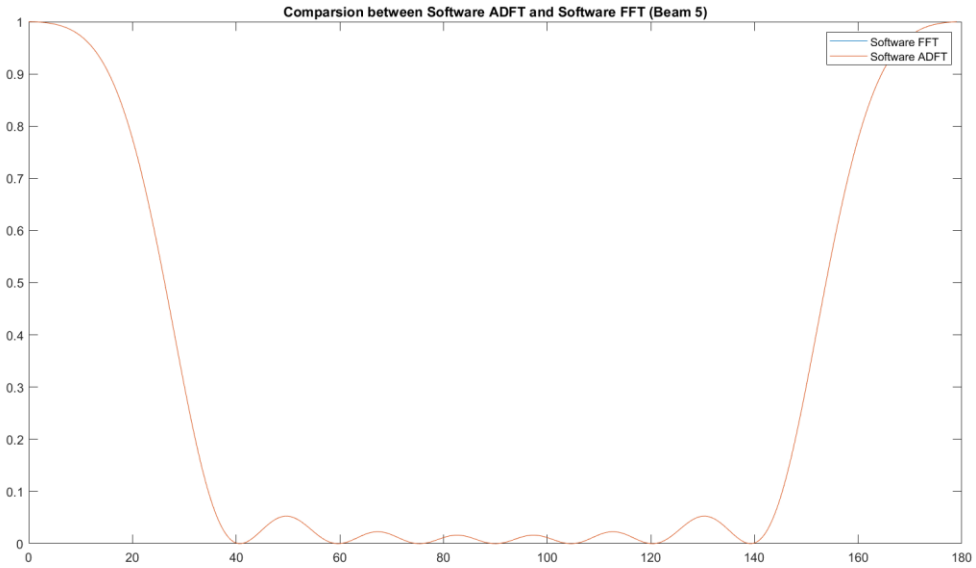
The beam patterns were plotted in both polar domain and cartesian domain, by comparing the plots of the ADFT output (left) and the desired FFT output (right), we can see that though some beams provided by ADFT algorithm has less gain, it is tolerable because the beam directions are correct.

By plotting the ADFT and FFT output for each beam k, where k = 1, 2…, 8 one by one, after comparing we see that all the beams are reasonably accurate (i.e., the direction is not too much deviated). For example, the plots for beam 1 and beam 5 are shown below.

For Beam 1,



For Beam 5,



After checking the outputs for all the 8 beams, the ADFT output and FFT output matches well which means that the ADFT algorithm performs correctly.

3. Block Diagram of ADFT Hardware

The un-optimized hardware prototype architecture of the proposed 8-beam multi-beamformer is shown below. This architecture has no multipliers; thus the multiplier complexity is 0.

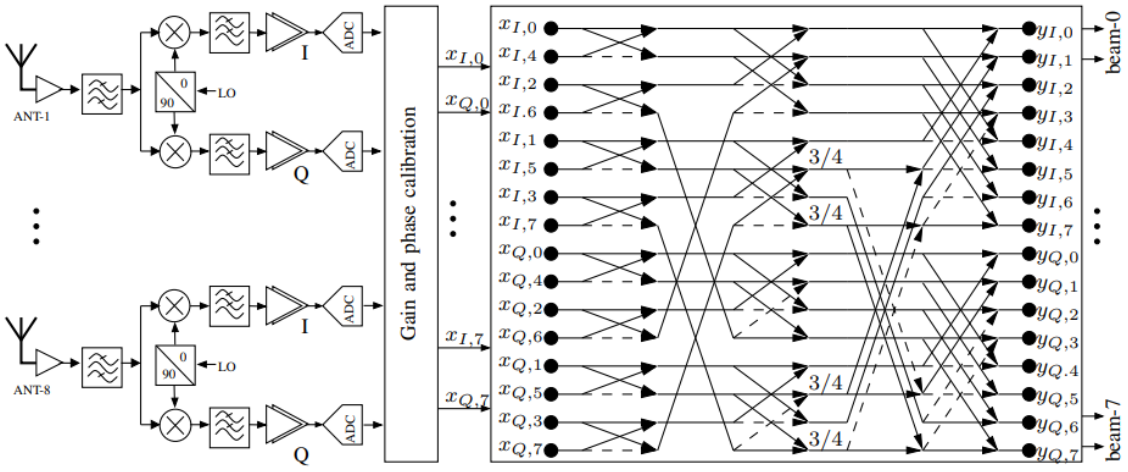
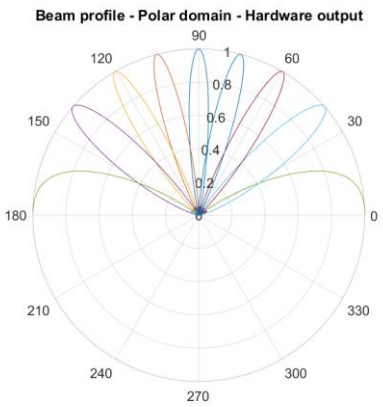
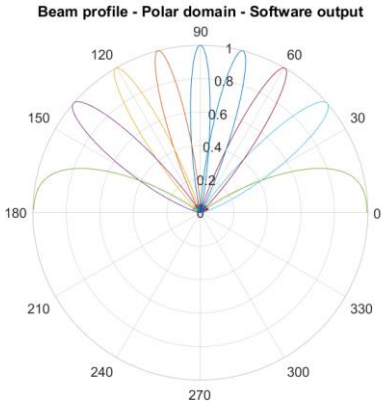


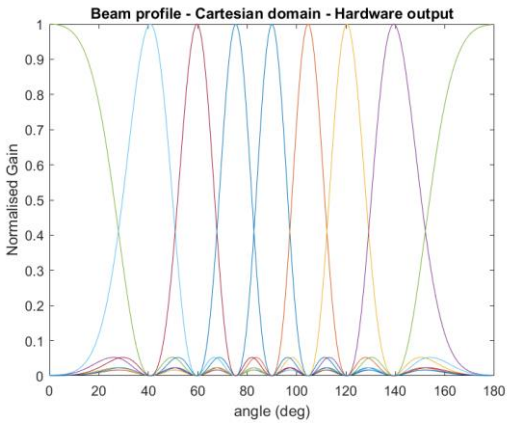
Figure 3.1: Prototype of the ADFT hardware architecture [1]

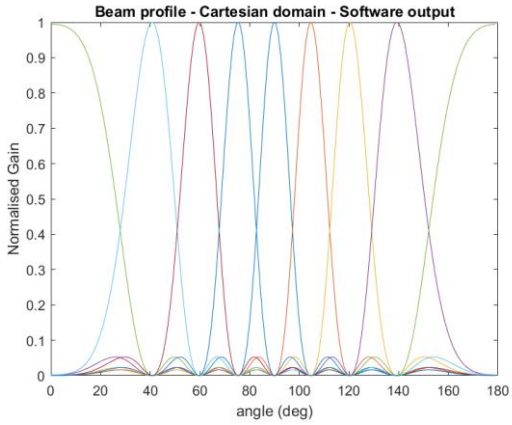
4.1 Functional Verification of un-optimized design

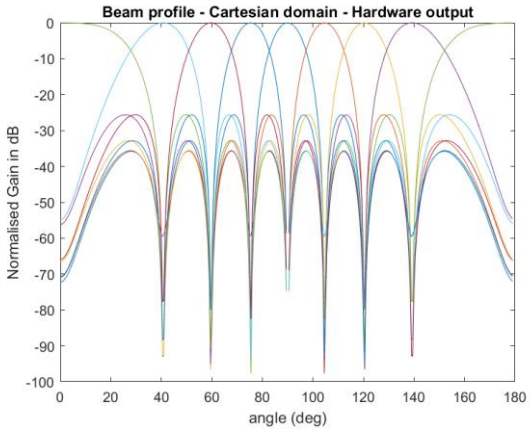
The initial un-optimized design has no pipelining and arbitrarily large word length. After visually comparing the beam patterns obtained from the hardware ADFT design and the software FFT algorithm, the functionality of the proposed hardware architecture can be verified.

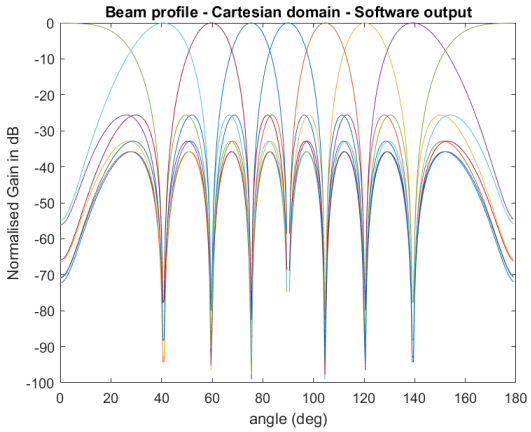








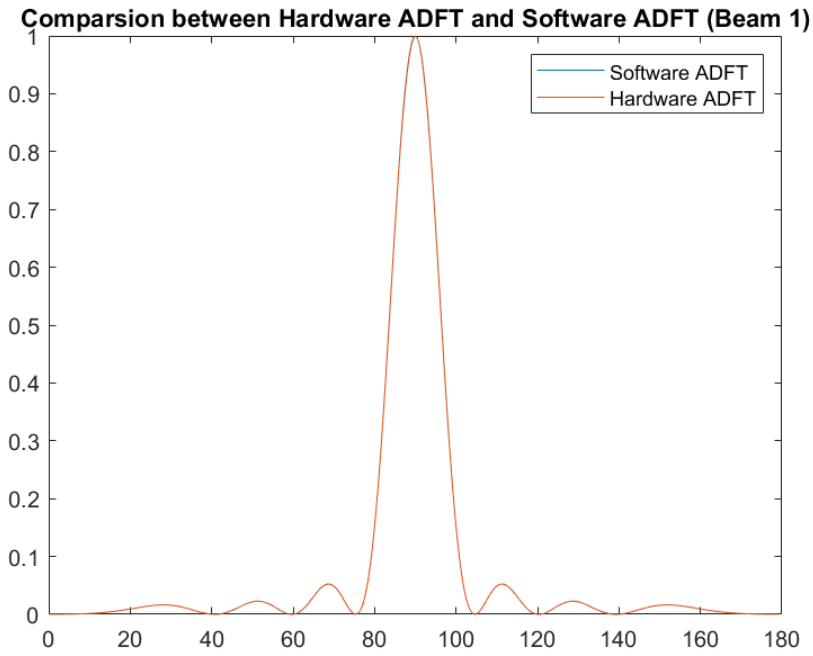




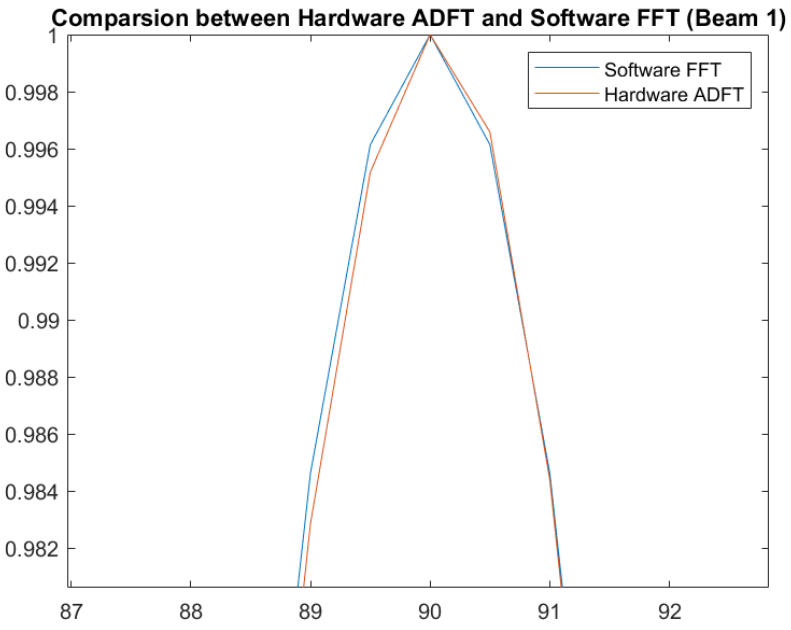
The beam patterns were plotted in both polar domain and cartesian domain, by comparing the plots of the hardware output (left) and the software output (right), it can be clearly observed that the proposed hardware design has the correct functions to form the 8 beams same as the software algorithm.

By plotting the hardware and software output for each beam k, where k = 1, 2…, 8 one by one, after comparing we see that all the beams are reasonably accurate (i.e., the direction is not too much deviated). For example, the plots for beam 1 and beam 5 are shown below.

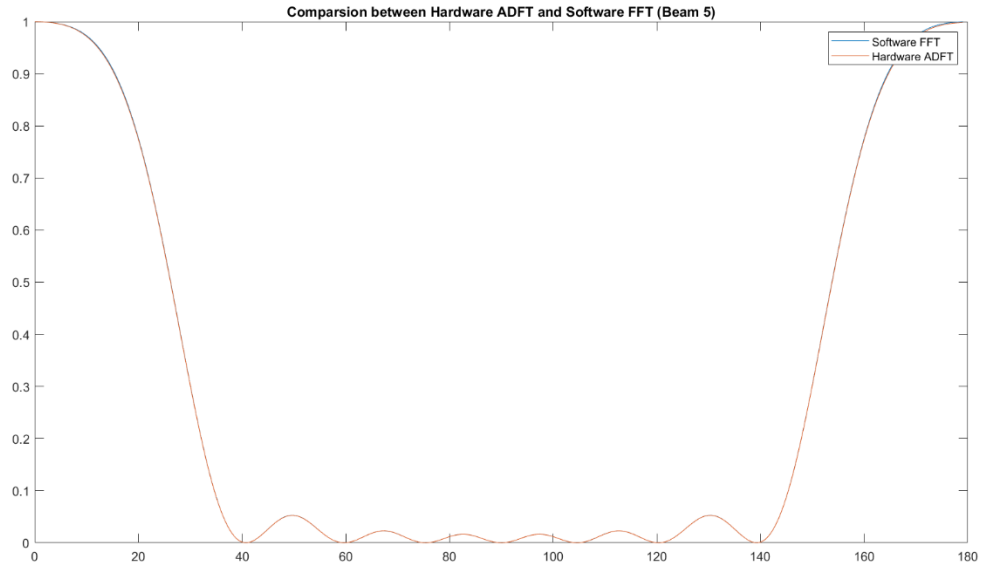
For Beam 1,



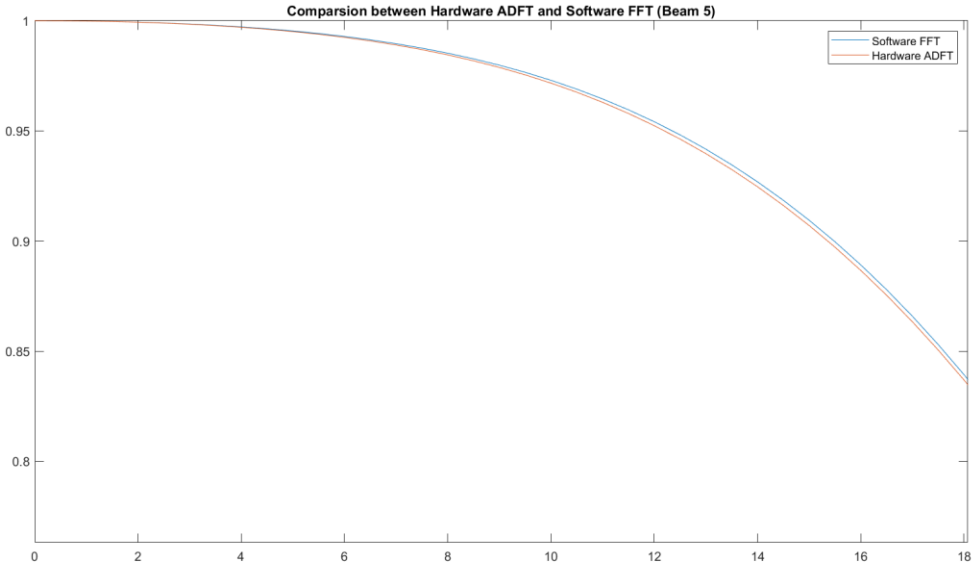
After zooming in,



For Beam 5,



After zooming in,



As we can see from the plots, after checking the outputs for all the 8 beams, the outputs obtained from the hardware and the software matches well except reasonable small errors (Zooming in). We can therefore verify that the initial construction of the hardware circuit performs well in spatial DFT. Also, the ADFT Hardware construction worked. Before pipelining, the critical path is shown below.

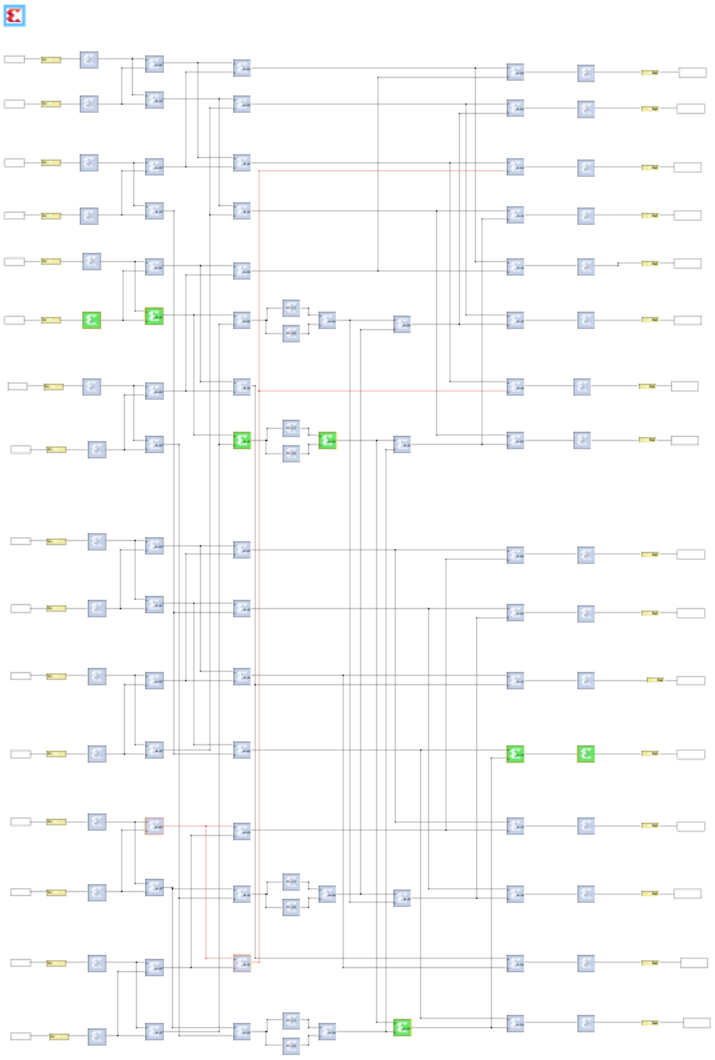


Figure 4.1: Critical path of the initial prototype design (Before pipelining)

4.2 Pipelining

Before pipelining the hardware design, the timing report is shown as below.

Timing Analyzer: ADFT

Post Synthesis Timing Paths: Clicking on a timing path highlights corresponding blocks in the model.

Violation type: setup — Status : PASSED

	Slack (ns)	Delay (ns)	Delay (ns)	Delay (ns)	vels of Log	Source	Destinatio	Source Clo	Destinatio	Path Constraints
1	1.232	18.728	13.421	5.307	73	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
2	1.232	18.728	13.421	5.307	73	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
3	1.232	18.728	13.421	5.307	73	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
4	1.232	18.728	13.421	5.307	73	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
5	1.238	18.722	13.421	5.301	73	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
6	1.238	18.722	13.421	5.301	73	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
7	1.238	18.722	13.421	5.301	73	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
8	1.238	18.722	13.421	5.301	73	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
9	9.931	10.029	7.051	2.978	33	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
10	9.931	10.029	7.051	2.978	33	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
11	9.931	10.029	7.051	2.978	33	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
12	9.931	10.029	7.051	2.978	33	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
13	9.931	10.029	7.051	2.978	33	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
14	9.931	10.029	7.051	2.978	33	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
15	9.937	10.023	7.051	2.972	33	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]
16	9.937	10.023	7.051	2.972	33	ADFT/D…	ADFT/D…	clk	clk	create_clock -name clk -period 20 [get_ports clk]

Figure 4.2.1: Timing report of the un-optimized design

The initial design has a Critical Path Delay (CPD) of 18.728ns as Figure 4.2.1 shows. The throughput performance is therefore,

$$throughput = f_{samp} = \frac{1}{18.728ns} = 53.4MHz$$

This design could be fully pipelined by adding delay blocks after adders which is equivalent to changing the latency of the adders. Initially, the latency of the adders at the three columns was changed to 1. The design could be further pipelined by adding more delays to the adders right after the shifters. Consequently, all the rows without shifters would be delayed as well to make sure the system has a linear phase. A screenshot of part of the pipelined hardware was demonstrated below.

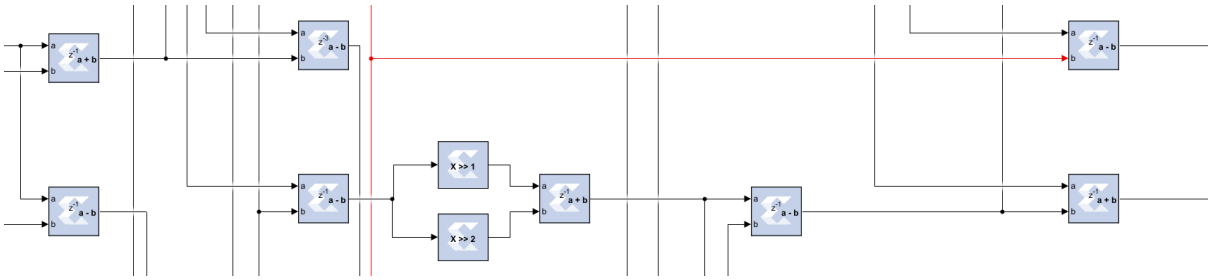


Figure 4.2.2: Screenshot of a part of the pipelined design

Thus, the system would work like a pipeline. The entire latency and CPD can be reduced.

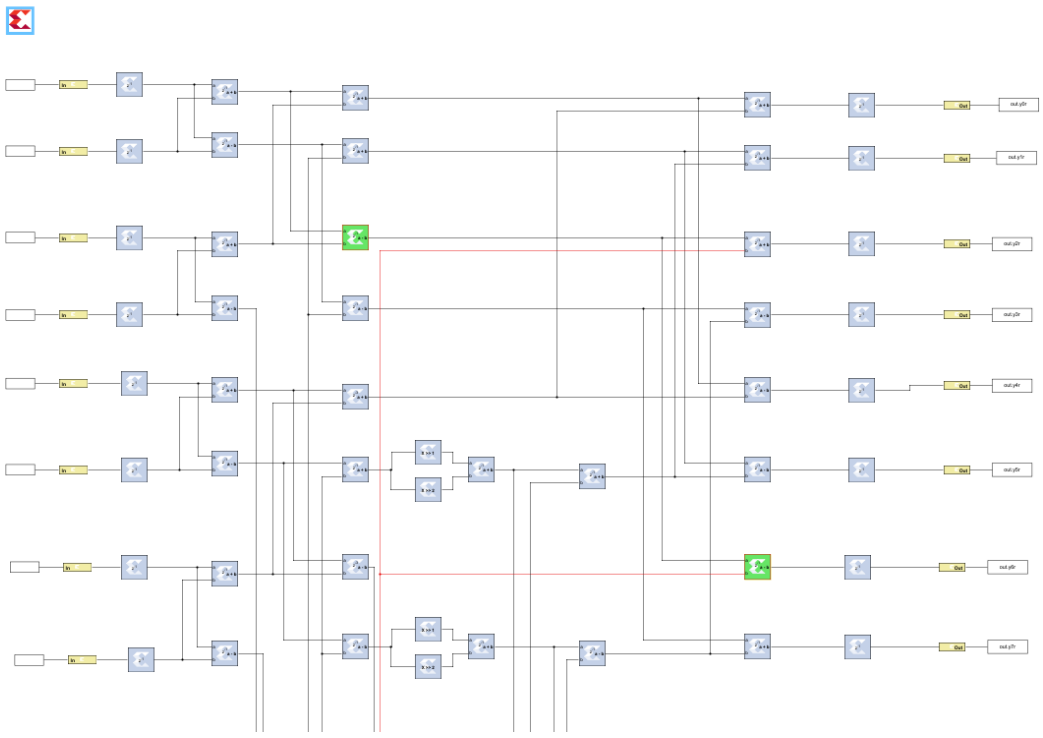
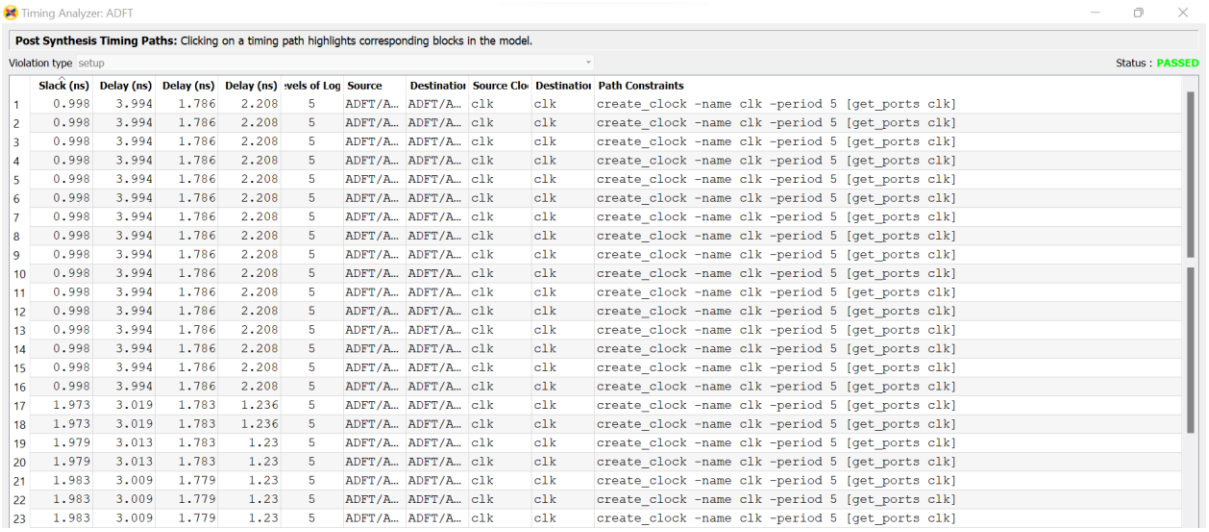


Figure 4.2.3: Critical path of the pipelined design

The critical path became only two blocks after pipelining. After fully pipelining, the timing report of the optimized design was shown as below.


Timing Analyzer: ADFT

Post Synthesis Timing Paths: Clicking on a timing path highlights corresponding blocks in the model.

Violation type: setup — Status : PASSED

	Slack (ns)	Delay (ns)	Delay (ns)	Delay (ns)	vels of Log	Source	Destinatio	Source Clo	Destinatio	Path Constraints
1	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
2	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
3	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
4	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
5	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
6	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
7	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
8	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
9	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
10	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
11	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
12	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
13	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
14	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
15	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
16	0.998	3.994	1.786	2.208	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
17	1.973	3.019	1.783	1.236	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
18	1.973	3.019	1.783	1.236	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
19	1.979	3.013	1.783	1.23	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
20	1.979	3.013	1.783	1.23	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
21	1.983	3.009	1.779	1.23	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
22	1.983	3.009	1.779	1.23	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
23	1.983	3.009	1.779	1.23	5	ADFT/A...	ADFT/A...	clk	clk	create_clock -name clk -period 5 [get_ports clk]

Figure 4.2.4: Timing report of the optimized design

The CPD of the optimized design is 3.994ns which is much faster than the un-optimized version. The throughput performance is

$$throughput = f_{samp} = \frac{1}{3.994ns} = 250.4MHz$$

4.3 Fine tuning word length

In FPGA, calculating float-point is resource-consuming and more complicated than fixed-points, hence fixed-point is implemented instead. Choosing appropriate fixed-point word length is important because a small length is likely to cause ineligible quantization error, while a big length will consume more FPGA resources. By comparing the software output with the hardware output to see how well they match, the most appropriate word length could be selected. However, there is always a trade-off between resource consumption and accuracy.

Therefore, after comparing the error between the hardware output and software output, the word length for the adders was selected as 8 with a fraction length of 4 (W=8, D=4). A word length less than this will produce unwanted noise while greater than this will consume more resources.

Furthermore, to verify the importance of fine tuning the word length, the following table is constructed based on the timing and resources reports of the two designs. Note that 'Before fine tuning' means using arbitrarily large word lengths to ensure the accuracy (i.e., using 18 bits and 14 binary points in this case).

	Before fine tunning		After fine tunning	
CPD for un-opt design	18.728ns		13.272ns	
CPD for opt design	5.316ns		3.994ns	
Resource for un-opt design	LUTs	Registers	LUTs	Registers
	1708	544	1116	384
Resource for opt design	LUTs	Registers	LUTs	Registers
	1508	1484	1212	1112

Table 4.3.1: CPD and resource before and after fine tuning

By comparing as we can see from the Table 4.3.1, selecting an appropriate word length can not only reduce the CPD but also save FPGA resources. Though an arbitrarily large word length has a good accuracy, it is worth to select a smaller word length as long as the error doesn't affect the result too much.

4.4 Functional Verification of the optimized design

After fully pipelining and selecting a most appropriate word length, a functional simulation was performed towards the optimized design. The 8 beams from the optimized ADFT hardware have been plotted in both Cartesian form and Polar form.

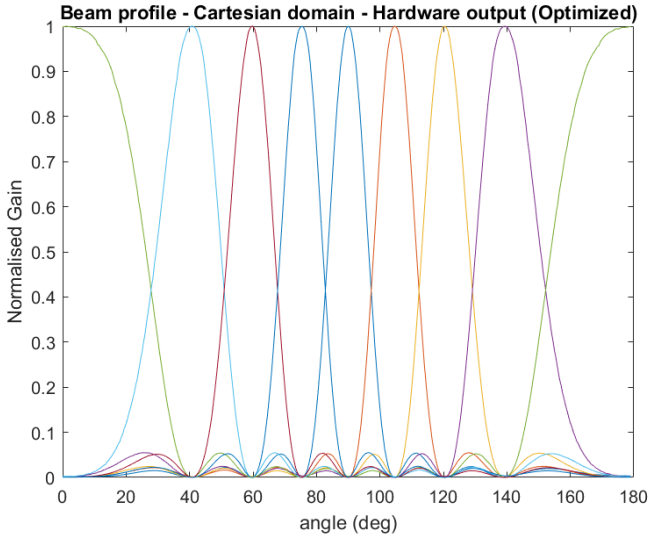


Figure 4.4.1: Beams from optimized design in Cartesian domain

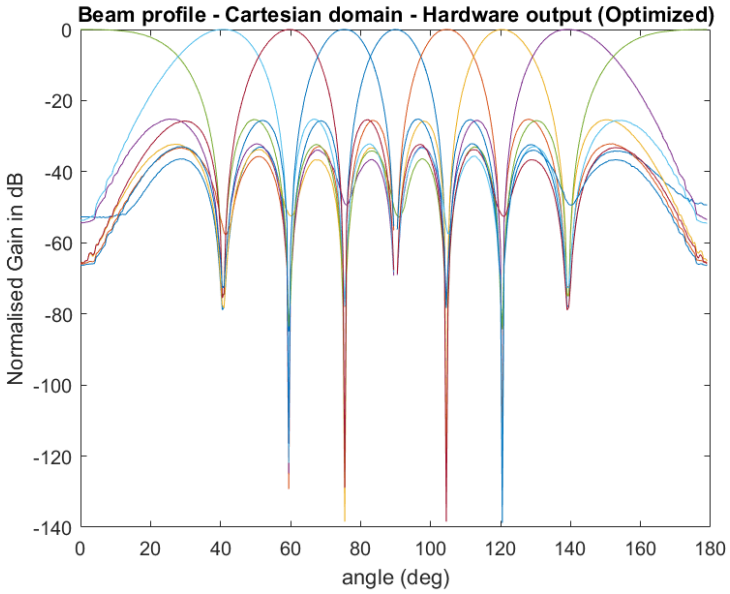


Figure 4.4.2: Beams from optimized design in Cartesian domain in Db

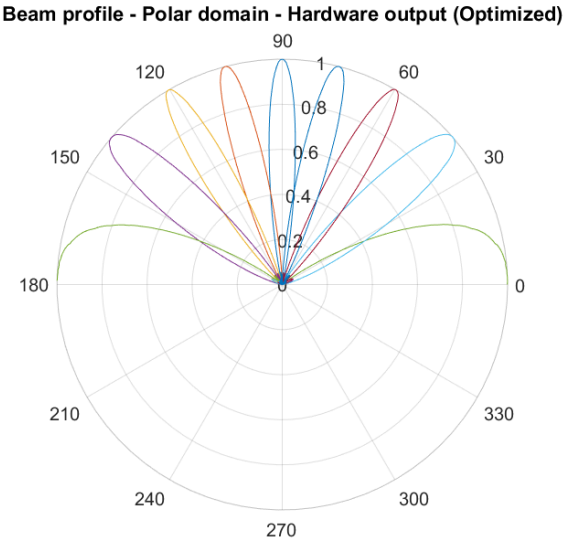


Figure 4.4.3: Beams from optimized design in Polar domain

As we can see from the plots above, the functionality of the output from the optimized ADFT hardware was verified by visually comparing with the previous plots. The error among software FFT, software ADFT and hardware ADFT was demonstrated below.

Beam number k	Main beam direction (deg)		
	FFT Software	ADFT Software	ADFT Hardware Optimized
1	179	179	180
2	139.5	139.5	139.5
3	120	120.5	120
4	104.5	104.5	105
5	90	90	90
6	75.5	75.5	75.5
7	59.5	59.5	59.5
8	40.5	40.5	40.5

Table 4.4.4: Comparison of various DFT Algorithms

As we can see from Table 4.4.4, the ADFT algorithm performs very well in the application of beamforming with negligible errors. After optimizing, the CPD of the hardware was being reduced while the FPGA resources were being increased as the implementation of delay blocks. Therefore, as reported in Table 4.3.1, we can see that 1212 LUTs and 1112 registers were being used and the CPD was 3.994ns for the optimized hardware, with a corresponding maximum frequency of 250.4MHz.

5. Approximate Adder Implementation

An approximate adder achieves tremendous improvements in power consumption and speed with a trade-off with reasonable accuracy. The complexity of the ADFT hardware architecture can be further reduced by replacing the full adder blocks with self-defined approximate adders.

5.1 Approximate Adder Structure

In this project, an 8-bit approximate adder has been constructed to replace the accurate adders using the selected word length from above (W=8, D=4). Since the carry in to the adder is always zero and there is no need to compute the carry out, the top level design of the 8-bit approximate adder only has two inputs and one output as the sum. The 8-bit adder was constructed using two 4-bit adders in ripple carry chain (i.e., the carry out from the first 4-bit adder is the carry in to the second 4-bit adder). However, each 4-bit adder was constructed using carry look-ahead because this structure has lower latency and reasonable resources consumption. The approximate one-bit adder is LUT-Based proposed in [2]. A truth table for the approximate adder is shown below.

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 5.1.1: Truth Table for one-bit approximate adder [2]

This architecture eliminated the carry computation circuit to save area, power, and FPGA resources. Furthermore, to balance the trade-off between the accuracy and the resource consumption, one of the 4-bit adders is approximate adder while the other is accurate. The accurate 4-bit adder is used for handling the higher four bits while the approximate 4-bit adder is used for handling the lower four bits. This structure increases the accuracy to a large extent. A block diagram of the proposed 8-bit adder at RTL level is demonstrated below.

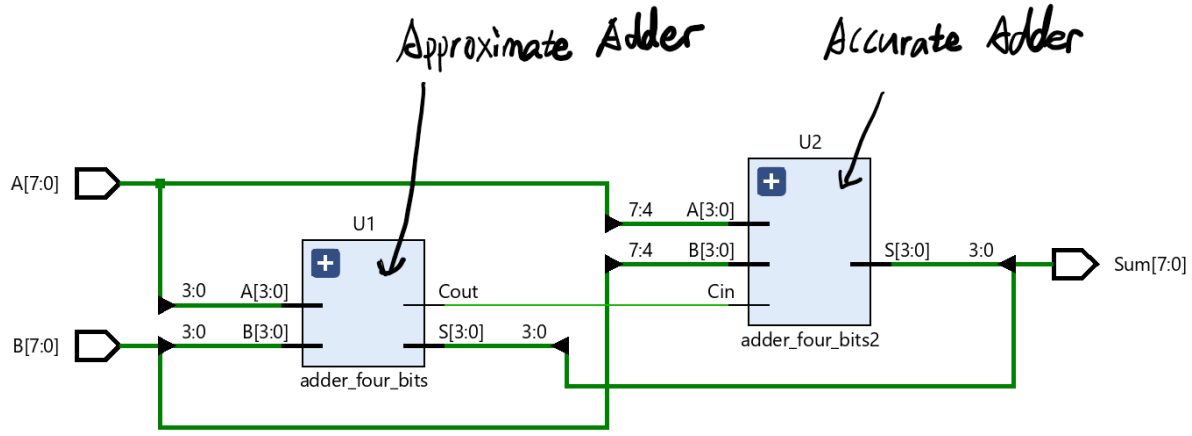


Figure 5.1.2: Block diagram of the 8-bit approximate adder at RTL level

Both 4-bit adders (U1 and U2) are carry look-ahead while they are connected in ripple carry chain

5.2 Functional Verification of the approximate adder

Once the approximate adder was designed in Vivado, a functional verification was also performed under the same environment by writing a suitable test bench.

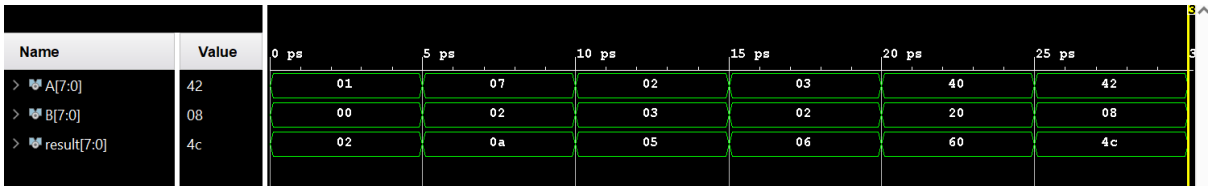


Figure 5.2.1: Vivado Simulation result for the 8-bit approximate adder

As we can see from Figure 5.2.1, the approximate adder performed well with reasonable errors. For example, at 10ps,

$$'00000010' + '00000011' \rightarrow '\ 00000101' (correct)$$

In contrast, swapping the inputs at 15ps,

$$'00000011' + '00000010' \rightarrow '\ 00000110' (error)$$

The error comes from the Truth Table (Table 5.1.1). However, error only happens in the lower four bits because of the architecture proposed in Figure 5.1.2. Hence, the function of the approximate adder is verified.

5.3 Functional Verification in Model Composer

Now the 8-bit approximate adder could be imported into the Model Composer as a black box after proper configuration.

By visually comparing the beam patterns obtained from the hardware ADFT design with approximate adders and the software FFT algorithm that plotted previously, the functionality of the proposed hardware architecture can be verified.

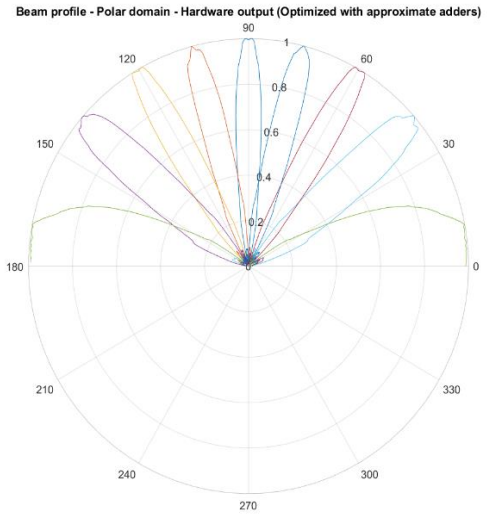


Figure 5.3.1: Beam profile in Polar domain (Optimized with approximate adders)

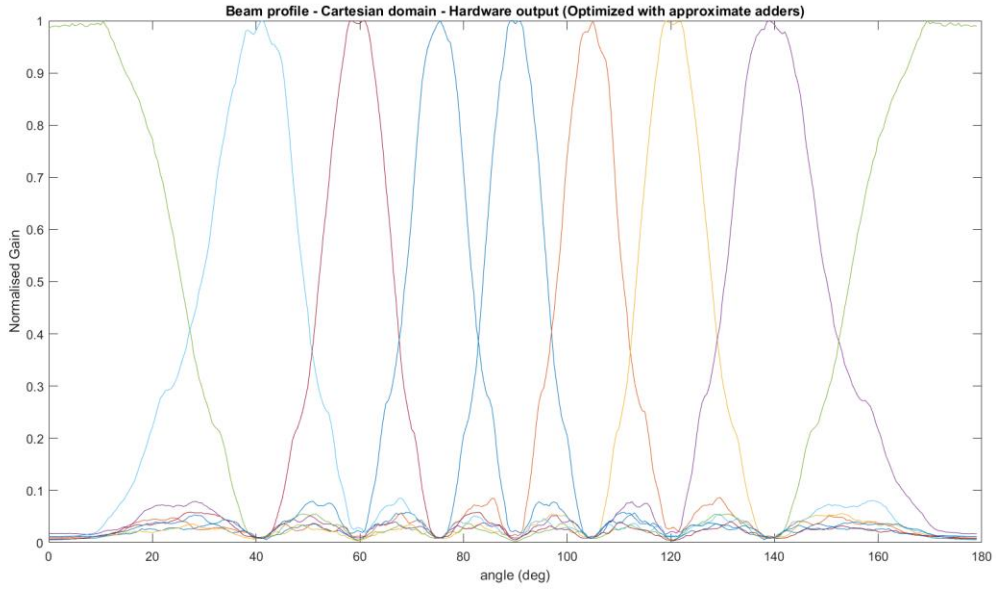


Figure 5.3.2: Beam profile in Cartesian domain (Optimized with approximate adders)

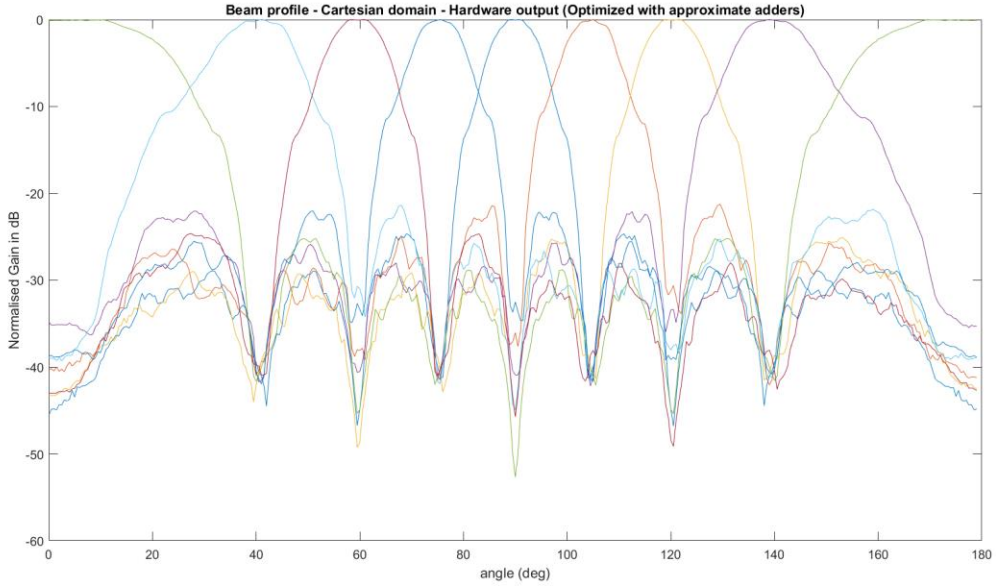
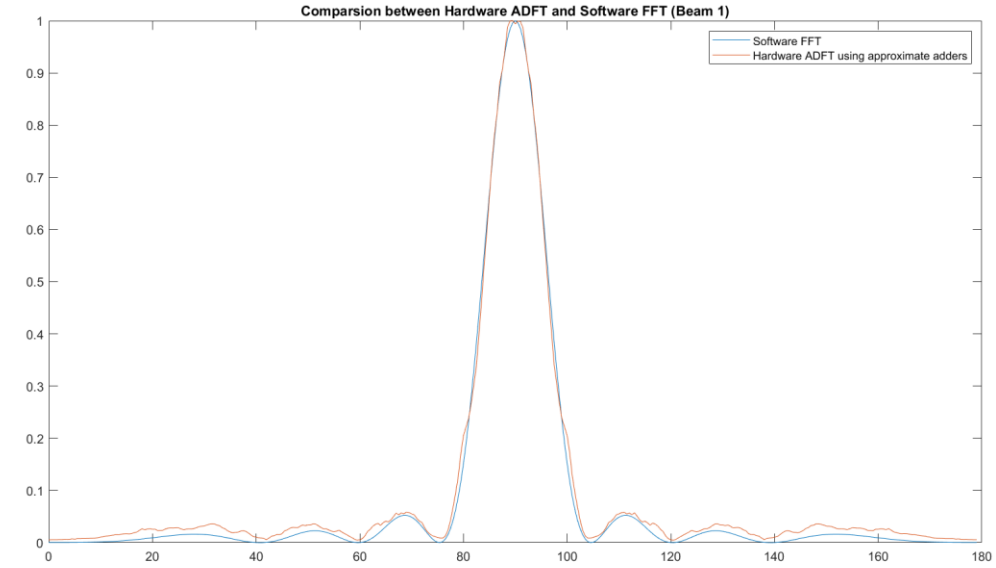


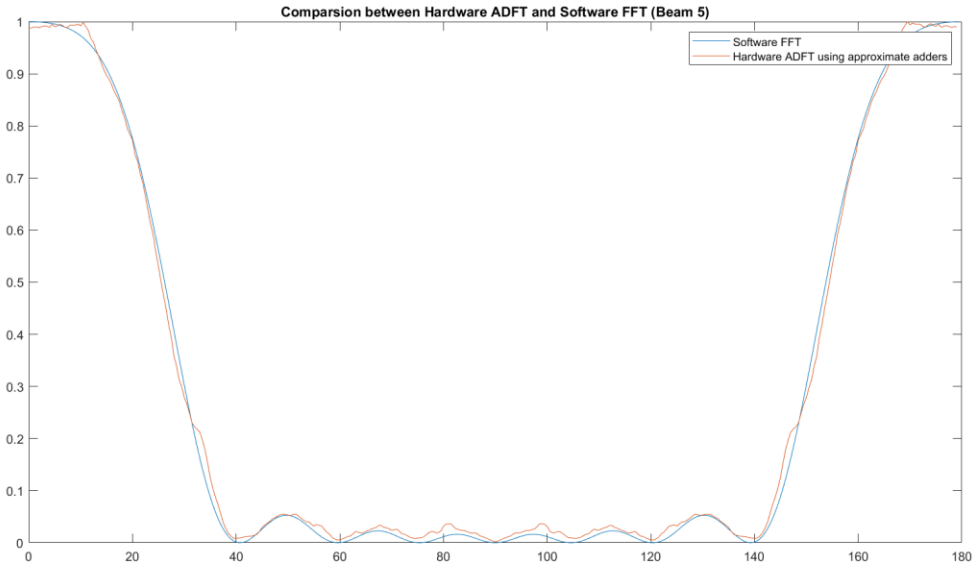
Figure 5.3.3: Beam profile in Cartesian domain in dB (Optimized with approximate adders)

Similar with previous verification, by plotting the hardware and software output for each beam k, where k = 1, 2…, 8 one by one, after comparing we see that all the beams are reasonably accurate (i.e., the direction is not too much deviated). For example, the plots for beam 1 and beam 5 are shown below.

For Beam 1,



For Beam 5,



After checking the outputs for all the 8 beams, the ADFT output with approximate adders and FFT output matches well which means that the proposed approximate architecture performs correctly. Therefore, though approximate adders may produce errors, it still can work. Since approximate adders were involved, Table 4.4.4 can be updated by adding another column.

Beam number k	Main beam direction (deg)			
	FFT Software	ADFT Software	ADFT Hardware Optimized	ADFT Hardware Optimized with Approximate Adders
1	179	179	180	178.5
2	139.5	139.5	139.5	139
3	120	120.5	120	120
4	104.5	104.5	105	105
5	90	90	90	90
6	75.5	75.5	75.5	75
7	59.5	59.5	59.5	59.5
8	40.5	40.5	40.5	41

Table 5.3.4: Comparison of various DFT Algorithms

By comparison, ADFT Algorithm performs well even using approximate adders. The critical path for the final design was demonstrated.

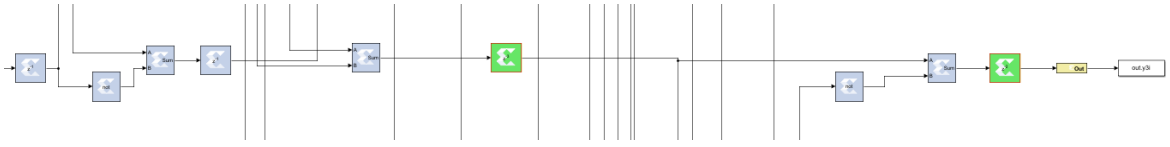
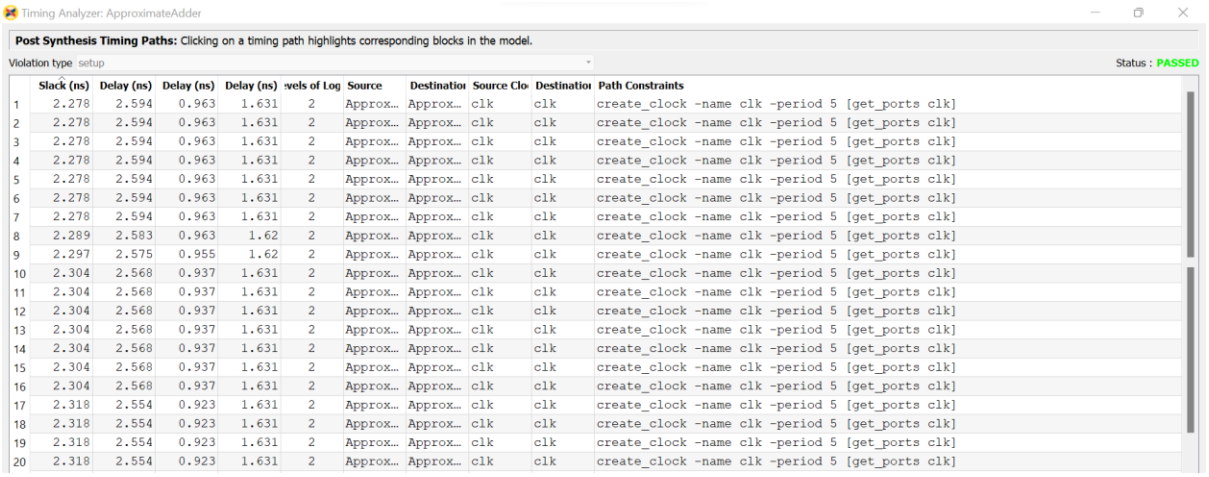


Figure 5.3.5: Critical Path of the final design

To examine whether it is worth to implement approximate adders, timing and resources reports were generated.



Timing Analyzer: ApproximateAdder

Post Synthesis Timing Paths: Clicking on a timing path highlights corresponding blocks in the model.

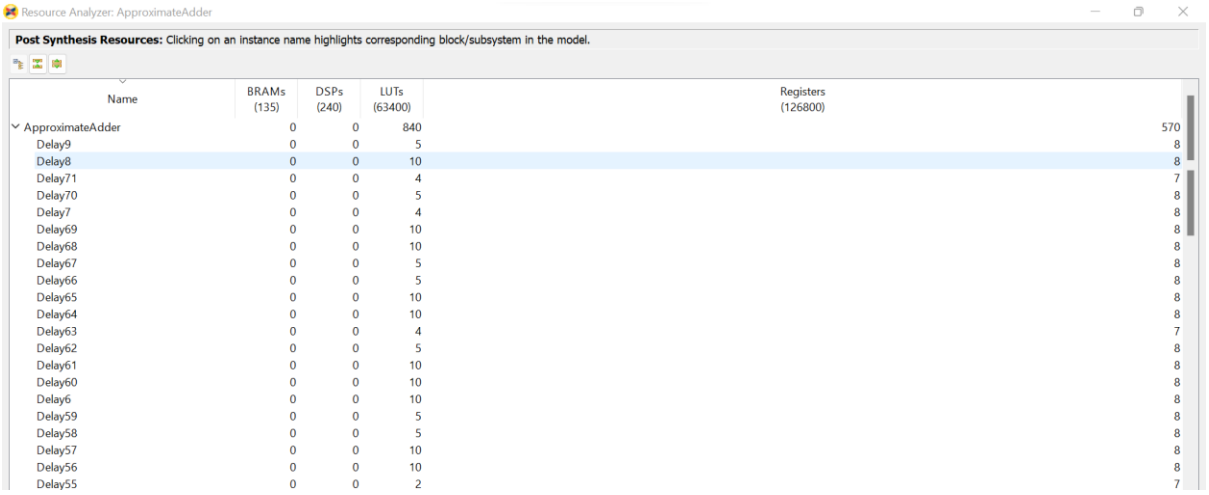
Violation type: setup — Status : PASSED

	Slack (ns)	Delay (ns)	Delay (ns)	Delay (ns)	vels of Log	Source	Destinatio	Source Clo	Destinatio	Path Constraints
1	2.278	2.594	0.963	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
2	2.278	2.594	0.963	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
3	2.278	2.594	0.963	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
4	2.278	2.594	0.963	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
5	2.278	2.594	0.963	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
6	2.278	2.594	0.963	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
7	2.278	2.594	0.963	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
8	2.289	2.583	0.963	1.62	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
9	2.297	2.575	0.955	1.62	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
10	2.304	2.568	0.937	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
11	2.304	2.568	0.937	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
12	2.304	2.568	0.937	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
13	2.304	2.568	0.937	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
14	2.304	2.568	0.937	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
15	2.304	2.568	0.937	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
16	2.304	2.568	0.937	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
17	2.318	2.554	0.923	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
18	2.318	2.554	0.923	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
19	2.318	2.554	0.923	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]
20	2.318	2.554	0.923	1.631	2	Approx...	Approx...	clk	clk	create_clock -name clk -period 5 [get_ports clk]

Figure 5.3.6: Timing report of the final design

As we can see from the timing report, the CPD reduced to only 2.594ns after replacing the accurate adders with approximate adders, which means the system performs faster. The throughput performance is

$$throughput = f_{samp} = \frac{1}{2.594ns} = 385.5MHz$$



Resource Analyzer: ApproximateAdder

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model.

Name	BRAMs (135)	DSPs (240)	LUTs (63400)	Registers (126800)
ApproximateAdder	0	0	840	570
Delay9	0	0	5	8
Delay8	0	0	10	8
Delay71	0	0	4	7
Delay70	0	0	5	8
Delay7	0	0	4	8
Delay69	0	0	10	8
Delay68	0	0	10	8
Delay67	0	0	5	8
Delay66	0	0	5	8
Delay65	0	0	10	8
Delay64	0	0	10	8
Delay63	0	0	4	7
Delay62	0	0	5	8
Delay61	0	0	10	8
Delay60	0	0	10	8
Delay6	0	0	10	8
Delay59	0	0	5	8
Delay58	0	0	5	8
Delay57	0	0	10	8
Delay56	0	0	10	8
Delay55	0	0	2	7

Figure 5.3.7: Resource report of the final design

As we can see from Figure 5.3.7, 840 LUTs and 570 Registers have been used. We can compare with the optimized design with accurate adders in respective of timing and resources.

	ADFT with Accurate Adders		ADFT with Approximate Adders	
CPD	3.994ns		2.594ns	
Throughput Performance	$250.4MHz$		$385.5MHz$	
Resources Consumption	LUTs	Registers	LUTs	Registers
	1212	1112	840	570

Table 5.3.8: CPD and resource before and after using approximate adders

Therefore, compared to accurate adders, the design with approximate adders performs 154% faster but only use 60% FPGA resources. It is a significant optimizing method in both latency and resources consumption.

Conclusion and Discussion

From this project, the effectiveness of the approximate algorithms has been examined in the application of antenna-based beamforming.

The Discrete Fourier Transform (DFT) has many applications while people often use Fast Fourier Transform (FFT) to compute the equivalent result with much lower complexity. However, the complexity of FFT could be further reduced by implementing Approximate DFT (ADFT) in both software and hardware at the cost of accuracy loss, so that the circuit would be multiplierless. It is useful as many applications do not rely on much high accuracy such as beamforming. By implementing ADFT hardware architecture, the output beams have similar properties as the equivalent beams output from the accurate FFT algorithm. Resources could be saved from unnecessary computation and the performance could be improved.

Moreover, a system could even be optimized further in complexity by implementing approximate adders. Though the adders would produce errors, it is fine because accuracy is not the priority under some circumstances. The approximate adder architecture can achieve significant gain in latency, area, and performance.

From this project, we can see that Model Composer/System Generator is strong and efficient in designing and constructing DSP system, while other designs can be constructed using HDL model. However, I still met some troubles in designing the system and gained some experience eventually. Since the timing report generating process and simulation process took a while (often 30min), I could read some relevant paper and design the hardware architecture using paper and pencil meanwhile. I realized how useful approximate algorithms are not only in the application of beamforming but also in many other applications such as image recognizing. I really hope I can gain enough FPGA-based DSP knowledge and skills in the future.

References

[1] V. A. D. F. G. C. V. A. Coutinho, "An 8-Beam 2.4 GHz Digital Array Receiver Based on a Fast Multiplierless Spatial DFT Approximation," *IEEE/MTT-S International Microwave Symposium,* 2018.

[2] S. R. M. A. H. Bharath Srinivas Prabakaran, "DeMAS: An Efficient Design Methodology for Building Approximate Adders for FPGA-Based Systems," *IEEE,* 2018.

[3] Open-source Library: https://sourceforge.net/projects/ approxfpgas/