FPGA Implementation of a Configurable FFT Accelerator

Hugo Zuniga Calvo

June 13, 2018

SPECIALIZATION PROJECT

Department of Computer Science

Norwegian University of Science and Technology

Supervisor 1: Dr. Waqar Hussain

Supervisor 2: Dr. Magnus Själander

Summary and Conclusions

The Discrete Fourier Transform (DFT) is a fundamental operation in many digital signal processing applications. It allows the transformation of signals from the discrete time domain to the discrete frequency domain. By using discrete time signals, it allows computers to be employed for digital signal processing. Nevertheless, in practice it suffers from high operation complexity which limits its applicability.

In order to overcome this issue, a clever algorithm that uses the periodicity in the coefficients and the discrete frequency function resulting from the DFT was proposed. This algorithm known as the Fast Fourier Transform(FFT), reduces the amount of operations needed to transform a signal from discrete time to the discrete frequency domain. This reduction in operations allowed to widen the application domain of the DFT. Therefore, FFT can be applied to a variety of areas. One of this, is the production of dedicated FFT processors, which allows for digital signal processing applications to be more efficient.

In the field of standalone FFT processors, many architectures have been proposed along the years. The evolution of these systems has been dependent on the technology available and the incremental improvement of the algorithms. Currently, one of the main focus of studies is the application of FFT as coprocessors or accelerators implemented in FPGAs.

In this report, we present an architecture for a reconfigurable FFT accelerator. Our architecture uses an iterative approach to the FFT implementation, in which only one butterfly is employed and the calculation is carried out iteratively stage by stage. In order to determine the correctness of the implementation, a test framework was implemented utilizing the Vivado design suite and the Octave platform. Based on the results we could determine that the implementation was correct and it accomplished the proposed objectives.

Contents

Chapter 1

Introduction

In this chapter, we will present the state-of-the-art on the design of such systems and define the objectives that motivated the development of the proposed platform.

1.1 Background

The development of accelerators to compute the fourier transform in discrete time was enabled by the discovery of the FFT algorithm. In this section, we will provide background on the FFT algorithms and the previous architectures that have been used in order to implement it in hardware.

1.1.1 Fast Fourier Transform

The Discrete Fourier Transform (DFT) is an operation that transforms discrete periodic signals from the time domain to the frequency domain. The DFT is applied in many areas of digital signal processing, for example signal modulation and demodulation, signal convolution and many others.

Equation 1.1, depicts the formula for calculating the DFT. The main problem with this formula is the complexity of the operations. For each of the N samples, N multiplications and N-1 additions must be performed. The total operation complexity for the DFT is O(N^2), which during most time of the twentieth century was a limiting issue since there were not many machines capable of performing such amount of calculations.

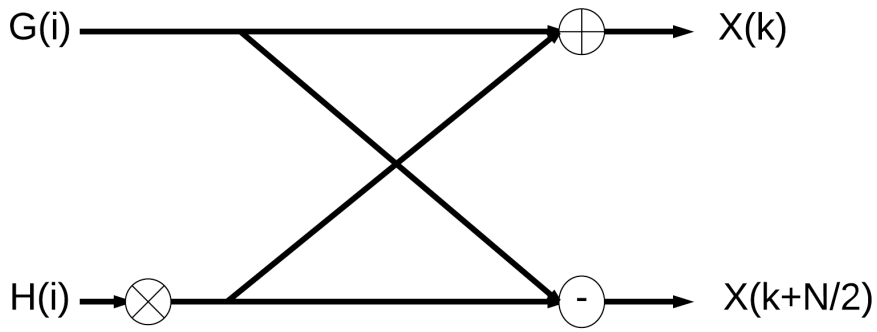

Figure 1.1: Radix 2 butterfly flow graph [2]

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi kn}{N}} \tag{1.1}$$

Motivated by the need of a less computationally demanding algorithm, the first FFT algorithm was proposed by Cooley and Tukey[1]. The algorithm proposed was the radix-2 algorithm, in which the linearity and periodicity of the DFT formulation were exploited in order to execute the algorithm using a divide and conquer approach. To do so, Cooley proposed grouping the samples depending on the position they occupied, therefore he separated them on even and odd samples. After the grouping was perform, the two resulting groups would represent a DFT formula on itself with half the samples of the original input. This algorithm can be applied recursively until a terminal case is found.

1.1.2 FFT Algorithms

There are many proposed algorithms for calculating the FFT. Each of them offer different grouping paradigms, as well as execution methodology. In this subsection, we will give a short review of the algorithms that are relevant to this project.

1.1.2.1 Radix-2 FFT

The radix-2 FFT is the simplest algorithm for FFT computation. It consists of dividing the samples into even and odd indexes. Equation 1.2 shows the radix-2 formula.

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) \cdot W_N^{nk} \\ &= \sum_{r=0}^{N/2-1} x(2r) \cdot W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1) \cdot W_N^{(2r+1)k} \\ &= \sum_{r=0}^{N/2-1} x(2r) \cdot W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1) \cdot W_{N/2}^{rk} \\ &= G(k) + W_N^k H(k) \end{aligned} \tag{1.2}$$

It can be observed from this formula that both G(k) and H(k) are also DFT formulas but for inputs half the size of the initial one. Based on this, the equivalences in equation 1.3 is satisfied.

$$\begin{aligned} G(k) &= G(k + N/2) \\ H(k) &= H(k + N/2) \end{aligned} \tag{1.3}$$

Using the results of 1.3 on 1.2 the formulas in equation 1.4 is obtained.

$$\begin{aligned} X(k) &= G(k) + W_N^k H(k) \\ X(k + N/2) &= G(k) - W_N^k H(k) \end{aligned} \tag{1.4}$$

Equation 1.4 is the formula for calculating the radix-2 FFT. It gives the fundamental arithmetic structure for any hardware implementation of this algorithm. The butterfly for the radix 2 is shown in Figure 1.1.

1.1.2.2 Radix 4

The derivation of the radix-4 algorithm is similar to the radix-2 algorithm. The most noticeable change is the size of the groupings, in this case the original DFT operation is divided in 4 groups. Each group is then applied for the algorithm in a recursive manner. By doing so, the final result is the group of formulas shown in equation 1.5.

[0]Image obtained from [3]

$$
\begin{aligned}
X(k) &= A(k) + W_N^k B(k) + W_N^{2k} C(k) + W_N^{3k} D(k) \\
X(k+N/4) &= A(k) - iW_N^k B(k) - W_N^{2k} C(k) + iW_N^{3k} D(k) \\
X(k+N/2) &= A(k) - W_N^k B(k) + W_N^{2k} C(k) - W_N^{3k} D(k) \\
X(k+3N/4) &= A(k) + iW_N^k B(k) - W_N^{2k} C(k) - iW_N^{3k} D(k)
\end{aligned}
\tag{1.5}
$$

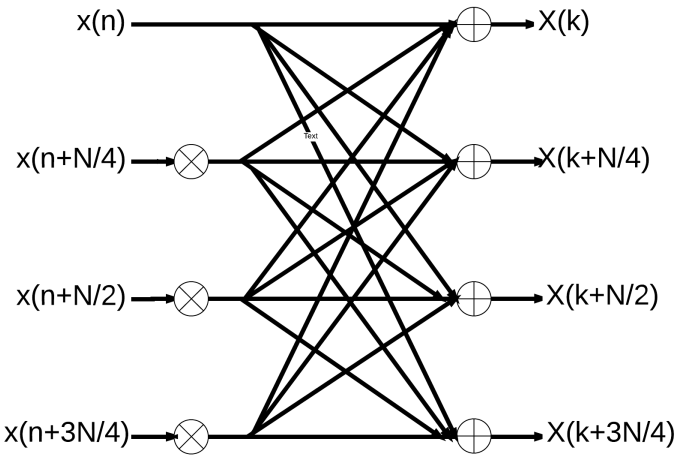

Figure 1.2: Radix 4 butterfly [4]

The advantage of radix-4 over radix-2 lies in the fact that the amount of complex multiplications for the complete algorithm is reduced. Therefore, the operational complexity is further reduced in the whole computation. The disadvantage, specially for standalone FFT processors, is that the amount of operations in one given recursion is more than doubled. This implies increasing the amount of hardware available. Nevertheless, radix-4 provides a faster approach towards calculation; this is because the number of stages to process FFT reduces by half in comparison to radix-2 FFT algorithm. Figure 1.2 shows the data flow graph for the radix-4 butterfly.

1.1.2.3 Split Radix

Split radix uses a mixed approach. It tries to fuse the multiplication complexity of radix-2 with the speed of the radix-4. In order to do so the grouping phase is performed so that at the end three groups will be created. The first group contains all even samples, while the uneven sam-

ples are split into two more groups. The result of this grouping are the formulas shown in Equation 1.6.

$$\begin{aligned} X(k) &= A(k) + (W_N^k B(k) + W_N^{3k} D(k)) \\ X(k+N/2) &= A(k) - (W_N^k B(k) + W_N^{3k} D(k)) \\ X(k+N/4) &= A(k) - i \cdot (W_N^k B(k) - W_N^{3k} D(k)) \\ X(k+3N/4) &= A(k) + i \cdot (W_N^k B(k) - W_N^{3k} D(k)) \end{aligned} \tag{1.6}$$

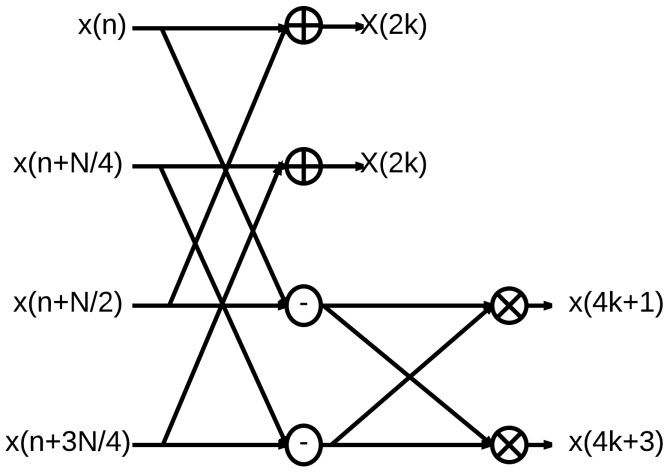

Figure 1.3: Split radix data flow graph [2]

By performing this alternative for the grouping stage, the resulting algorithm has a reduced operation count when compared to radix-2 and an increase in performance as well. The main drawback of this implementation is the lack of regularity, which makes it more complex to implement in hardware platforms.

1.1.3 FFT Architectures

As mentioned previously, there have been many proposals for FFT Processor architectures. The two most common ones are the Single-path Delay Feedback [5] proposed by Wold and Despain and Multi-path Delay Feedback [6] proposed by Bi and Jones. Most of the designs that have been

introduced since there are variations of this two architectures [7] [8]. In this subsection we will introduce the basic architectures and the cache architecture.

1.1.3.1 Multipath Delay Commutator Architecture

Figure 1.4 shows the basic architecture of the radix-2 Multipath Delay Commutator architecture. Each of the recursions of the algorithm is developed into one stage of delay elements, commutator and one butterfly. The main idea behind this architecture is using the delay elements to buffer the input until the corresponding sample comes and the algorithm can start execution.

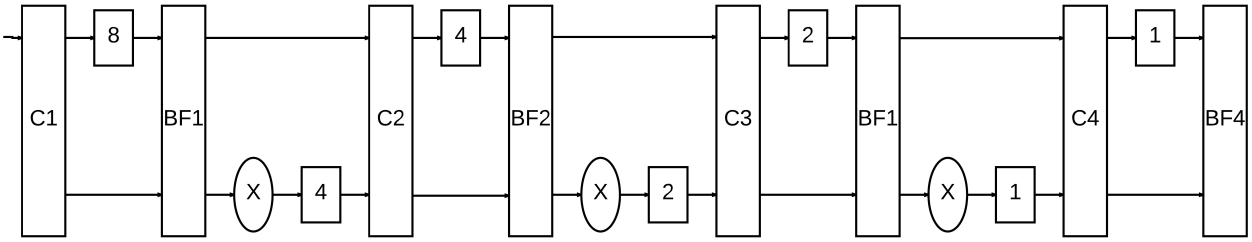

Figure 1.4: Multi-path delay commutator architecture [9]

In the case of radix-2, the first N/2 samples are stored. Once the N/2 + 1 sample arrives, the first buffered sample and the current one are moved to the pipeline where the result is computed and passed to the next pipeline stage in which the new samples are buffered in order to wait for their corresponding pair in the given stage.

This architecture is a straight forward approach to FFT implementation in hardware. Because of the regularity of the operations, a small control logic is needed. The control logic would be in charge of mainly synchronizing the buffering and calculation phases. The main drawback of this architecture, is that unless it is constantly performing computations of different data stream, the utilization of the resources is poor. This comes as a result of having one set of resources for each of the stages in the algorithm. Also, this architecture suffers low flexibility for computing different sample sizes because of its simple control logic.

1.1.3.2 Single-path Delay Feedback Architecture

The single-path delay feedback architecture is shown in Figure 1.5. As it can be observed, the delay elements are reduced when compared with the MDC architecture. This comes as a result of using a feedback shift register that will be used for fulfilling the functionality of delay loop for

two stages at the same time. By reducing the amount of memory needed by the architecture, it provides a more appealing design than the multi-path delay commutator.

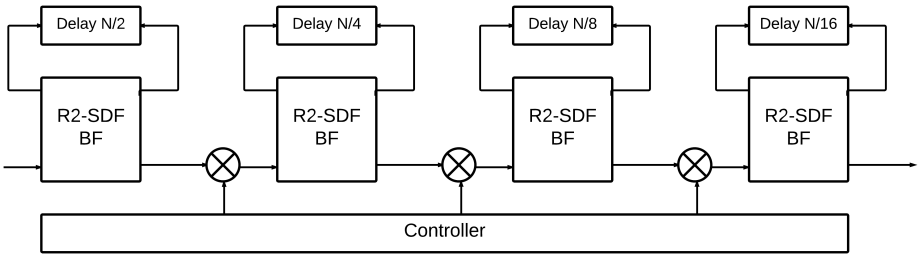

Figure 1.5: Single Delay Feedback architecture [9]

The drawbacks for this architecture are the same as the multi-path delay commutator. This is due to the similarity in the use of resources as well as the simplicity of control logic. Also, because of its low flexibility in the sample size and algorithm adaptability it is not a suitable option for an efficient reconfigurable accelerator.

1.1.3.3 Cache FFT

Cache FFT is a low power alternative to perform the FFT algorithm in hardware [10]. The main idea behind this architecture is to use a cache memory close to the processor, in order to fetch a part of the data and carry out as many operations as possible before storing it back to memory and taking another part.

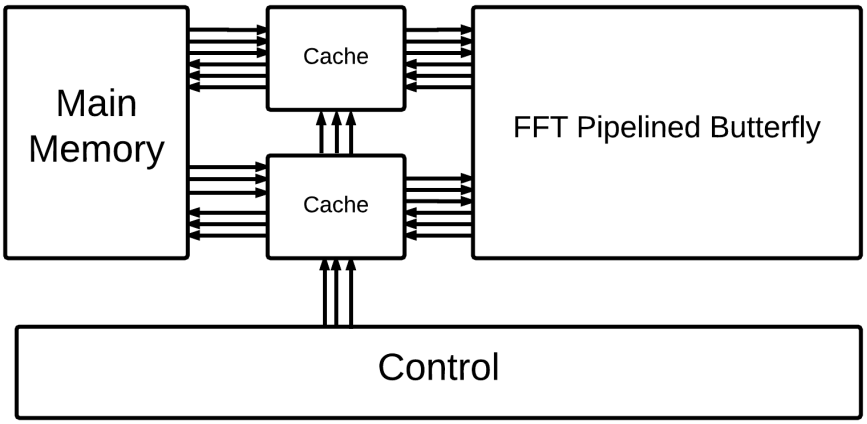

Figure 1.6: Cache FFT Architecture Diagram

The main purpose of the algorithm is to save resources by using a more complex control logic. This control logic is in charge of fetching the samples from the main memory as well as synchronizing the operations in the FFT unit itself. In order to simplify the design, the cache FFT assumes the inputs are organized in a bit reverse order from the beginning.

Figure 1.6 shows the architecture of the Cache FFT. The most noticeable fact about this diagram is the need for only one FFT butterfly unit, which is a significant reduction from the butterflies needed for the SDF and MDC. This allows this implementation to have a higher utilization of the resources as well as lower power consumption due to the savings on static power coming from idle units. Also the need for a dedicated commutator for sample ordering is eliminated because the reordering phase would be carried out by the controller once the samples are written back to the main memory.

Chapter 2

Architecture and Design

In this chapter we will present the architecture of the configurable FFT. First, we will describe the system level architecture and the different macro blocks and then we will focus on the individual designs of the different subcomponents that make up the macro blocks.

2.1 System Level Design

In this section we will present the objectives that drove the FFT accelerator architecture as well as the top level architecture of the platform.

2.1.1 Design Objectives

To properly describe the architecture of the designed FFT accelerator, it is necessary to first introduce the requirements that were considered while building the system in order to have a better understanding of the design choices and their impact on the system. For the system designed in this project, the main requirements that were considered are:

1. System must be able to perform radix-2 and radix-4 FFT algorithms.
2. Architecture must be capable of handling data widths of 64, 128, 256, 512 and 1024 samples.
3. Computational resources must have a high utilization.

4. Design must make use of FPGA resources.

The purpose of the first two objectives is to allow the resulting platform to be reconfigurable at runtime, the user shall have the ability to configure the device so that it can handle different sample sizes and perform different algorithm. This allows for a higher customizability at the application side, because the user would be able to set the adapt the platform to best fit the application and get an efficient result in terms of quality of calculations.

The third objective arises as a consequence of the result of previous research, in which the utilization of the resources is low [11] [6] [2]. This is the case specially for the computational units like the butterflies, this comes as the result of hardware replication in different stages of the design in order to provide a real-time performance. Since the aim of this project is to come up with an accelerator that would be used along with a processor, the main focus is to come up with a highly efficient architecture in term of component utilization.

Last, the fourth objective is used to avoid investing effort into designing elements that are already provided on the FPGA platform. This is the case of components like memories and multipliers, because the Zynq-7000 offers both block RAMS which can be inferred during synthesis as well as DSP hard-macros that are able to perform fast multiplication.

2.1.2 FFT Accelerator Architecture

The architecture chosen for the implementation of the FFT accelerator can be observed in figure 2.1. This architecture is a derivative of the cache FFT architecture [10]. This architecture was chosen over more traditional architectures like Single Delay Feedback (SDF) or Multipath Delay Feedback (MDF) because both SDF and MDF have a low utilization of resources and offer low flexibility to adapt to different sample sizes and algorithms [5].

Different from SDF and MDC, this architecture provides a high utilization of resources, because it uses only one processing element which is the butterfly pipeline as shown in Figure 2.1. This pipeline butterfly can compute both radix-2 and radix-4 butterflies, and has a depth of 2 stages. The pipeline receives the samples from a crossbar unit which is in charge of serving as the communication channel between data memory and the pipeline, therefore its mission is to fetch the samples from memory and deliver them to the right port in the pipeline. It is also in

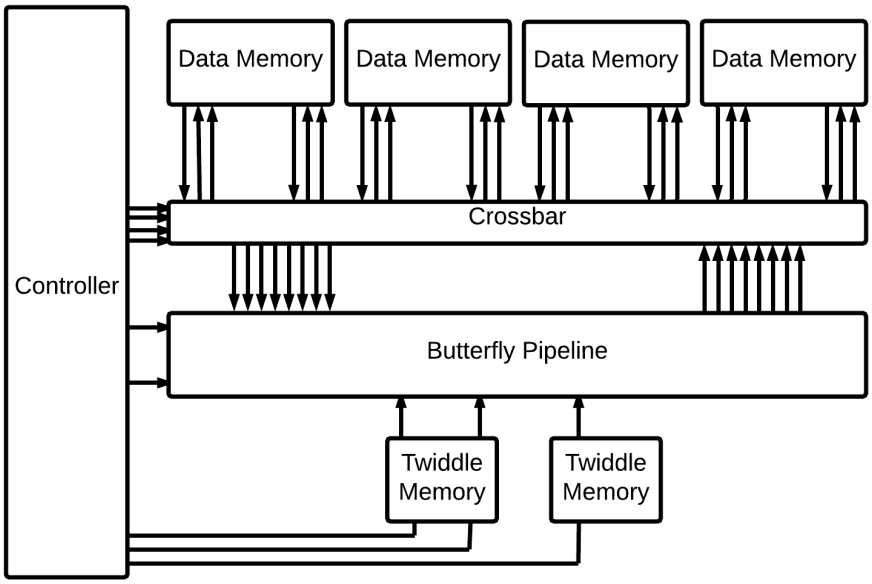

Figure 2.1: FFT Accelerator Architecture

charge of taking the results from the pipeline and route them to the corresponding memory so that they can be stored for the next iteration.

The control unit is in charge of the configuration of the different elements and the synchronization of the operands and twiddle factors depending on the stage of the algorithm. This unit is also in charge of interacting with the status and control registers which shall be written by the user in order to set the desired operation mode. Finally, the memory system consists of 1 single port memory and 5 dual port memories of which 4 are used to store the samples that will be processed, while the last one along with the single port memory stores the twiddle factors which will be fed directly to the butterfly pipeline.

The main idea behind this architecture is to have a ping-pong like behaviour. The memories will alternate between read and write cycles, the memories will switch tasks every time a FFT stage is finished. Therefore the data flow will be constantly switching from stage to stage. This allows for exploiting the throughput to the maximum.

2.2 Sub-module Architecture and Design

In this section, we will discuss the architecture of each of the sub-modules present in the architecture, as well as the design decisions that were taken during the implementation of such blocks.

2.2.1 Controller

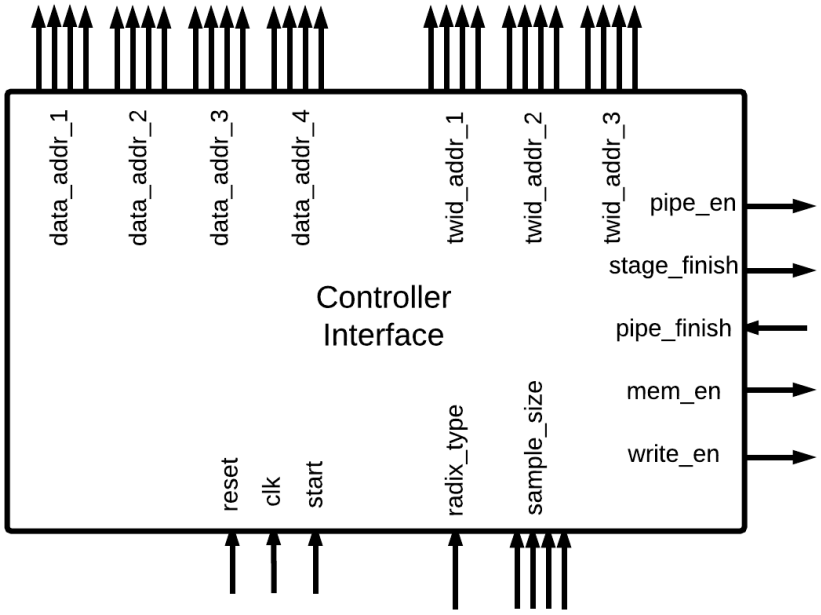

Figure 2.2: Controller interface

The controller is the element in charge of organizing and synchronizing the data processing and synchronization. In order to do so it performs the following tasks:

1. Generate the addresses for the data memory accesses.
2. Generate the addresses for the twiddle factors.
3. Keep track of the stage of the algorithm that the unit is currently processing.

4. Enable the operation of the different elements.

5. Configure the sub-modules according to the configuration registers.

Each of this tasks is carried out by a separate sub-module in the controller. Each of this sub-modules will be further described later in this section. The interface of controller module is shown in Figure 2.2.

The signals coming from the configuration registers are the radix type, the start flag and the sample_size signals. The values for the sample_size and radix type signals must be set up before setting the start flag and initiating computation, otherwise the accuracy of the results cannot be guaranteed.

2.2.1.1 Address Generator

The address generator is in charge of generating the addresses of the samples that will be fed to the pipeline. In order to do so, it takes advantage of the regularity in the access of the different stages of the radix-2 and radix-4 algorithms. In order to illustrate it, Table 2.1 shows the behaviour of the addresses of a few operations of a 16 sample FFT using radix-2 execution with its inputs organized in memory in bit reversed order.

Table 2.1: Address patterns for an extract of a 16 sample radix-2 execution.

Stage 1		Stage 2		Stage 3		Stage 4	
000**0**	000**1**	00**0**0	00**1**0	**0**00**0**	**0**10**0**	**0**000	**1**000
001**0**	001**1**	00**0**1	00**1**1	**0**00**1**	**0**10**1**	**0**001	**1**001
010**0**	010**1**	01**0**0	01**1**0	**0**01**0**	**0**11**0**	**0**010	**1**010
011**0**	011**1**	01**0**1	01**1**1	**0**01**1**	**0**11**1**	**0**011	**1**011
100**0**	100**1**	10**0**0	10**1**0	**1**00**0**	**1**10**0**	**0**100	**1**100
101**0**	101**1**	10**0**1	10**1**1	**1**00**1**	**1**10**1**	**0**101	**1**101
110**0**	110**1**	11**0**0	11**1**0	**1**01**0**	**1**11**0**	**0**110	**1**110
111**0**	111**1**	11**0**1	11**1**1	**1**01**1**	**1**11**1**	**0**111	**1**111

As it can be observed, in each stage there is one bit position that doesn't change between two different operand addresses in the same column. After excluding this bit from the address it behaves as a counter, this behaviour was noticed by [10] and it applies to all the different sample widths and to the radix-4 algorithm, just that in the radix-4 version there are two bits that remain unchanged.

In Figure 2.3, the architecture of the address generator can be observed, it is a simple design consisting of a counter to keep track of the dynamic part of the address and an array of multiplexers which select if the corresponding bit of the address should be tied to a given value or assigned from the counter bits. These multiplexers are handled by a the stage_count signal. This signal is managed by the stage counter which is another submodule of the controller unit

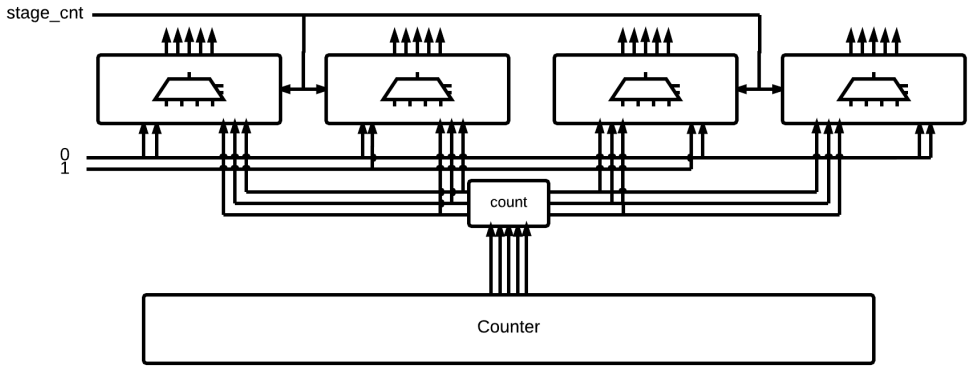

Figure 2.3: Address Generator Architecture

2.2.1.2 Twiddle Address Generator

This submodule is in charge of generating the address signals that are sent to the twiddle memories. The same as in the case of the address generator, the twiddle address algorithm depends on the current stage of the algorithm that is being performed.

$$X^F(k) = G^F(K) + W_N^k H^F(K) \tag{2.1}$$

$$X^F(k+N/2) = G^F(K) - W_N^k H^F(K) \tag{2.2}$$

In order to illustrate the twiddle address generation algorithm, it is necessary to understand the twiddle factor used in Equation 2.1 and Equation 2.2. It can be observed from the equations, the exponent of the twiddle factor depends on the index of the frequency element being calculated as well as on the size of the FFT in the current recursion stage. Table 2.2 presents an example of twiddle addresses for a 16 sample radix 2 FFT algorithm with bit reversed inputs.

Twiddle addresses show a regular behaviour along the whole FFT calculation. It can be described mathematically by using Equation 2.3 for radix-2, in this equation s represents the sam-

Table 2.2: Twiddle factor access patterns for a 16 sample radix-2 execution.

Stage 1	Stage 2	Stage 3	Stage 4
0	0	0	0
0	4	2	1
0	0	4	2
0	4	6	3
0	0	0	4
0	4	2	5
0	0	4	6
0	4	6	7

ple number, while r stands for the recursion stage. A similar equation can be determined for the twiddle factors for the radix-4 algorithm.

$$twid_addr(s,r) = \frac{s * NUM\_SAMPLES}{2^r} \bmod \frac{NUM\_SAMPLES}{2} \tag{2.3}$$

Figure 2.4 shows the architecture used for the twiddle address generator. The block consists of an adder loop which keeps updating the value of the base twiddle address every clock cycle by adding a value obtained from shifting the sample_size signal by a given amount of bits every clock cycle. The shifting amount is obtained from a small combinational logic which encodes the stage_count and translates it into a shift amount. In the case of the radix-4 twiddle factors more logic needs to be added in order to generate two more twiddle factors.

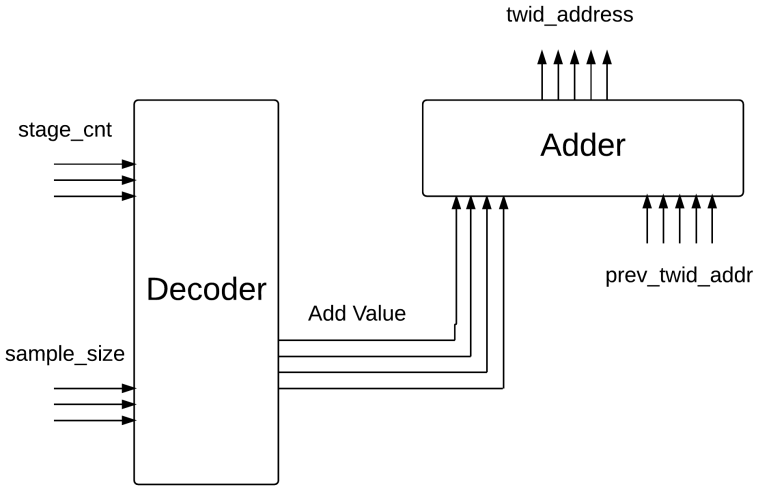

Figure 2.4: Twiddle address generator block diagram

2.2.1.3 Enable Generator

The enable generator is the block in charge of synchronizing the work of the controller. It is responsible for receiving the start command and initiating execution, as well as disabling address generation every time a stage finish to allow for the pipeline to flush out and write all the results to memory.

The interface of the enable generator is shown in Figure 2.5, the inputs because the enable generator is the one in charge of synchronizing the operation of the different units in the accelerator, it receives the start signal as input from the user. Besides the start signal, it receives the stage_finish signal which indicates if the current recursion of the FFT is finished and the pipe_finish signal, which is received from the pipeline when all the results have been written into memory and the next recursion can be initiated.

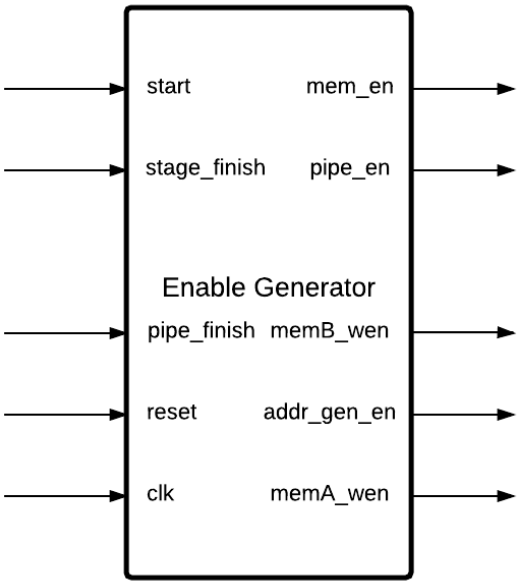

Figure 2.5: Enable generator interface

The design of the unit is a finite state machine that implements a group of timing transitions that can be represented through timing diagrams. First, a high reset signal sets all the outputs to low. This is so, that the system can start from a known state regardless of the previous state. This behaviour is represented in Figure 2.6.

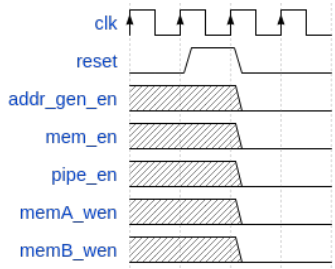

Figure 2.6: Timing diagram for reset behaviour

For the case of the pipe_en and mem_en signals, these signals shall remain at zero after reset and be activated to one only after the start signal is received. After the start signal is received, the enable generator shall set the pipe_en and mem_en signals to one and they should remain in the same state until the computation is completed. This behaviour is depicted in Figure 2.7. The start signal is read only once per FFT operation, only after receiving a reset high will the enable controller wait for the start signal to start the operation.

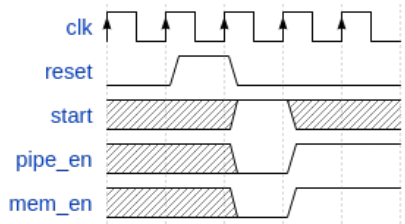

Figure 2.7: Timing diagram for pipe_en and mem_en behaviour

The behaviour of the address_gen_en signal is different because it controls the address generators; both the data address and the twiddle address. Because of this, when the controller finalizes the execution of one of the recursion stages, the address generation needs to halt until the pipeline finishes storing all the results of the current iteration in memory. Hence, the addr_gen_en goes up after the start signal is received, it stays in this state until the stage_finish signal is received then it goes down and once the pipeline signals that the results have been successfully stored to memory with the pipe_finish signal it goes high again. The timing diagram for this trace is shown in figure 2.8.

Finally, the memA_wen and memB_wen control which memory is to be read and written in the current recursion stage. This signal switch every time a pipe_finish event is received, if so then the memory which is currently being read will be employed for writing in the next iteration and the other way around for the other memory. Just like the other signals, after reset it waits for the start signal to set the memory to write and then it continues switching from reading to

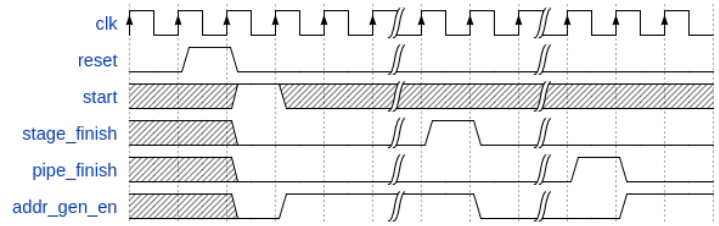

Figure 2.8: Timing diagram for addr_gen_en signal generation

writing state until the end of execution. The timing diagram describing this signal control is shown in Figure 2.9.

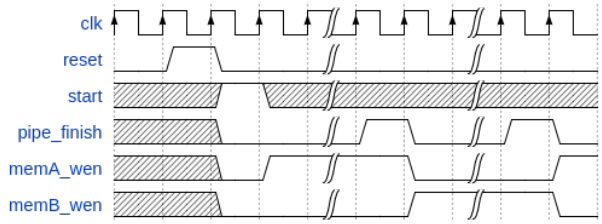

Figure 2.9: Timing diagram for memA_wen and memB_wen

2.2.1.4 Stage Counter

The stage counter is a simple counter which determines the recursion number that keeps track of the FFT algorithm's recursion stage is. This information is sent to all the other modules inside the controller because they need it to be able to adapt their operation accordingly. The signal that will trigger the count in the stage counter is the pipe_finish signal because after this the results have all been written to memory and the system is ready to begin with the next stage.

2.2.2 Butterfly Pipeline

The butterfly pipeline is the arithmetic unit of the FFT accelerator. It takes the operands along with the twiddle factors that were fetched from the memory and applies the FFT algorithm.

The pipeline has a depth of two stages. The first stage is dedicated to the multiplication units, in this case the system is tuned so that the multiplication is carried out by the specialized DSP modules provided inside the Zynq-7000 platform. We decided to use the DSP hard macros because that saves the overhead of designing a multiplier.

The second stage of the pipeline is dedicated to the add and substract operations to finish the complex multiplications, as well as the additions and substractions necessary for the result

calculation. The decision of not separating both sets of calculations was made in order to reduce the critical path of the design and therefore enhance the speed of the operations.

One more important fact about the architecture of the butterfly pipeline is that it uses fixed point arithmetic for all operations. Therefore, the twiddle factors must be stored in fixed point representation. As a design decision, in order to gain one more extra bit for the precision in the calculations the twiddle factor representation for the number 1, 1i, -1 and -1i have special representations which are resolved by the multiplier with a multiplexer circuit.

2.2.3 Crossbar

The main reason a crossbar was added to the FFT accelerator is that in order to execute the radix-4 in the same amount of clock cycles as the radix-2 four operands need to be fetched from memory every clock cycle. This means that the memories should be able to provide four operands in every access, but the BRAM blocks provided in the Zynq-7000 are dual port memories. In order to be able to fetch 4 operands in every clock cycle two block RAMS are used as individual banks and the crossbar contains the logic that decides the bank where each result should be written. In order to illustrate the logic used in the crossbar, we will use Table 2.3.

Table 2.3: Address patterns for an extract of a 16 sample radix-2 execution.

Stage 1				Stage 2			
0**000**	0**001**	0**010**	0**011**	0**000**	0**100**	1**000**	1**100**
0**100**	0**101**	0**110**	0**111**	0**001**	0**101**	1**001**	1**101**
1**000**	1**001**	1**010**	1**011**	0**010**	0**110**	1**010**	1**110**
1**100**	1**101**	1**110**	1**111**	0**011**	0**111**	1**011**	1**111**

As it can be observed in every stage, the four operands addresses have a group of bits that doesn't change the value, while there are two bits that change the value. The idea behind the crossbar is to exploit this characteristic by using the least significant bit that is changing in order to determine the memory bank, for the operand to be stored. This way, in the next stage the same procedure can be applied.

2.2.4 Memories

For the memories, we took the decision to employ the dual ported memories that are offered by the Zynq-7000 platform. Only one of the memories is a simple one port memory and that is the second twiddle memory because only one twiddle factor needs to be read from this memory on every clock cycle.

Chapter 3

Implementation and Results

In this chapter, the results from the FFT implementation will be discuss. The discussion will be divided into three sections. In the first section, the testing framework and results are described. The second section will discuss the synthesis results and FPGA utilization and the last section will do a comparison against previous implementations of FFT for FPGA.

3.1 Functional Testing Framework

In this section, we will introduce the testing framework used for the verification of the accelerator and discuss the results obtained.

3.1.1 Testing Framework

The testing of the VHDL model was performed through test bench simulation. It was divided in two main phases. The first phase was the platform refinement through impulse response analysis, the main objective of this phase was to find bugs on the different stages by isolating error propagation. In order to isolate the errors, the accelerator was excited with impulse functions as input and in every iteration a time displacement of one sample would be performed. By exercising the accelerator with an impulse report and shifting in time, the error sources were isolated and therefore the debugging process was simplified.

Once the incremental refinement of the platform was finished, a random input generator was written using Octave. The generator would create random complex samples and also a

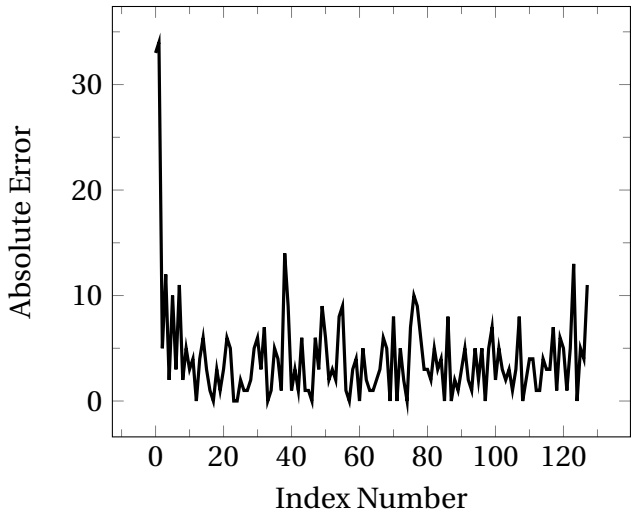

Figure 3.1: Absolute error for a 64 samples radix-2 FFT calculation

reference text file with the result of the DFT using its primitive FFT implementation. There were two purposes to this testing, the first is to finish verifying the platform by comparing the output results to a reference and also to determine the error behaviour in the implementation. Figure 3.1 shows the behaviour of the error for one execution of a 64 sample radix-2 FFT, also Table 3.1 shows the average of the error and the standard deviation on the random tests.

Table 3.1: Random Input Test Results

Number of samples	Radix-2		Radix-4	
	Average	STD DEV	Average	STD DEV
64	11.73	52.19	4.32	4.7
128	8.25	27.25		
256	9.37	17.82	8.14	10.1
512	11.99	17		
1024	16.11	22.4	15.46	21.2

The error is due to the fixed point calculations performed on the pipeline as well as the precision of the twiddle factors, if the width of the platform data word were to be increased then a reduction on the error would be appreciated. It is also interesting to note that the average error is higher for the radix 2 calculations than on the radix 4, this is mostly attributed to the number of iterations each of the algorithms has to perform in order to calculate the result. Given that a radix-2 calculation needs double the amount of iteration as its radix-4 counterpart, then the error accumulation is higher.

3.2 Synthesis Results

In order to confirm that the proposed platform could be implemented, it was synthesized using Vivado design suite. The results of the synthesis are shown on Table 3.2.

Table 3.2: Synthesis results FFT Accelerator

Component Type	Number of elements	Component Utilization(%)
LUT	3551	6.67
FF	512	0.48
BRAM	7	5
DSP	12	5.45

The synthesis results are in line with the expectations of the design. Each of the multiplications was mapped to a DSP module and the memories (twiddle and data) were mapped to BRAM blocks. Also the utilization of the platform is still small, so it opens the possibility of replicating the block in order to be able to carry out parallel FFT calculations.

In order to determine the efficiency of the platform, the maximum clock frequency and the power consumed were calculated using the synthesis model. The results of this simulations are displayed in Table 3.3.

Table 3.3: Maximum frequency and power from synthesis model

Parameter	Result
Max Frequency	125 MHz
Dynamic Power	177 mW

The results obtained are similar to the parameters obtained by other proposed FFT implementations for FPGA. For the power value, the current model doesn't have a CPU interface, therefore the output pins that are being chosen by the tool are not the definitive output pins. Hence, it is expected for the power consumption to decrease once the CPU interface is added and the real output pins are used for the power simulations.

3.3 Comparison Against Previous Work

In this section, a comparison with previous related works is made. Table 3.4 holds the information related to our work and to previous FPGA Fast Fourier Transform implementations.

Table 3.4: Maximum Frequency and Power from Synthesis Model

	Derafshi et al. (2010)	Abdullah et al. (2009)	Zhou et al. (2009)	Ingemarsson et al. (2017)	This Work
Device	xc4vlx25	xc4vfx140-12	xc4vsx25-10	xc4vsx25-10	xc7z020
Architecture Type	SDF	SDF	SDF	SDF	cache
Algorithm	radix-2	radix-2	radix-2	radix-2^2	radix-2/radix-4
Data word length	16	16	16	18	16
Coefficient word length	16	N/A	16	18	16
# slices	2472	13873	2256	642	3551
# BRAM	N/A	N/A	8	6	7
# DSP	10	0	16	16	12
fmax, MHz	100	198	236	294	125

It is important to note that all the architectures in the Table except by the one presented by us are based on the SDF Architecture. In terms of algorithms, in [12] and [13] decimation in frequency radix-2 was implemented. For [14] and [11] a radix-2^2 was the chosen algorithm. Based on this information and the data presented in the table, it is possible to perform a comparison between our approach and previous works.

First in terms of utilization of resources, it can be observed that our approach makes use of more slices than most of the previous platforms. This can be attributed to the retargetting capabilities of our platform. Because all the other platforms focus on one type of algorithm and one specific sample size, their control logic is much simpler. Meanwhile, our control logic must be capable of adapting to different sample sizes and switch between radix-4 and radix-2, therefore the resource overhead is higher. Nevertheless, if we compare the DSP and BRAM utilization, the results are very similar. This comes as a consequence of using a SDF approach, because each stage of the implementation requires a shift register and a multiplier in order to execute.

In terms of speed, the results show that our implementation is lacking. One of the reasons for this is the overhead for calculations due to the generic nature of our platform. Nevertheless, if we consider that when using radix-4 mode, the sample rate per stage is double than any of

radix-2 algorithms then we could argue that the speed difference is much less than shown in the table. Nevertheless, there are some optimizations that can be implemented like adding a preadder phase like in [11], or improving the balance of the pipeline from the memory accesses to the write process. Also, for the case of the radix-2 algorithm, because there is an excess of multipliers and adders in the pipeline because of the radix-4 support. The sample rate could be doubled by supporting parallel calculation of distinct sample pairs.

3.4 Summary and Conclusions

In this work, we have come up with an architecture for a reconfigurable FFT accelerator. The architecture is based on the cache FFT algorithm proposed by Baas [10]. This architecture is capable of performing radix-2 and radix-4 FFT algorithms for sample sizes of 64, 128, 256, 512 and 2048.

In order to determine the correctness of the funcitionality, a set of tests were performed on the designed platform. These tests were based on the simulation of the pre-synthesis model. After analyzing the resulting data, it was determined that the functionality had been correctly implemented and the error in the calculations was in line with the expected value.

A comparison with earlier implementations was also conducted. From the comparison it was determined that the platform offers an optimal use of resources than previous works. Nevertheless, in terms of speed of calculation it lacks compared to the best results of previous architectures. This is attributed mainly to the reconfigurability of the platform, due to the need of adding extra control logic to allow the platform to support different algorithms and sample sizes.

3.5 Recommendations for Further Work

In terms of recommendations for future works, the first task shall be to perform the implementation of the accelerator and interface it with a microprocessor. This will allow to evaluate the architecture in a real application. In order to interface it with the microprocessor it is necessary to create a bus interface that will allow the FFT accelerator to send and receive data to the CPU.

In terms of architecture performance, it was observed that it lacks compared to previous architectures. This could be solved by refining the pipelining of the butterfly unit. This was determined from the results of the timing reports, which showed that the critical paths lie inside this sub-unit. Therefore, an increase of the depth of the pipeline may be beneficial for the performance of the platform.

It is also necessary to add support for 2048 samples calculation using radix-2 algorithm. This is in order to satisfy the requirements demanded from this type of platforms in the industry.

Bibliography

[1] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[2] A. D. Das, A. Mankar, N. Prasad, K. K. Mahapatra, and A. K. Swain, "Efficient VLSI Architectures of Split-Radix FFT using New Distributed Arithmetic," 2013.

[3] "Decimation in time radix-2 fft." https://archive.cnx.org/contents/ce67266a-1851-47e4-8bfc-82eb447212b4@7/decimation-in-time-dit-radix-2-fft. Accessed: 2017-12-05.

[4] N. S, A. Suhas Sali, N. Velamala, S. S, and S. K, "Implementation of Radix-4 Butterfly Structure to Prevent Arithmetic Overflow," 11 2015.

[5] E. H. Wold and A. M. Despain, "Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementations," *IEEE Transactions on Computers*, vol. C-33, pp. 414–426, May 1984.

[6] G. Bi and E. V. Jones, "A Pipelined FFT Processor for Word-sequential Data," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, pp. 1982–1985, Dec 1989.

[7] L. Wenqi, W. Xuan, and S. Xiangran, "Design of fixed-point high-performance FFT processor," vol. 5, pp. V5–139–V5–143, June 2010.

[8] W.-C. Yeh and C.-W. Jen, "High-speed and low-power split-radix FFT," *IEEE Transactions on Signal Processing*, vol. 51, pp. 864–874, March 2003.

[9] N. Kirubanandasarathy and K. Karthikeyan, "VLSI Design of Pipelined R2MDC FFT for MIMO OFDM Transceivers," *Journal of Applied Sciences*, vol. 13, pp. 197–200, 2013.

[10] B. M. Baas, "A Generalized Cached-FFT Algorithm," *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, vol. 5, pp. v/89–v/92 Vol. 5, 2005.

[11] C. Ingemarsson, P. Källström, F. Qureshi, and O. Gustafsson, "Efficient FPGA Mapping of Pipeline SDF FFT Cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 2486–2497, Sept 2017.

[12] Z. H. Derafshi, J. Frounchi, and H. Taghipour, "A High Speed FPGA Implementation of a 1024-Point Complex FFT Processor," *2010 Second International Conference on Computer and Network Technology*, pp. 312–315, April 2010.

[13] S. S. Abdullah, H. Nam, M. McDermot, and J. A. Abraham, "A High Throughput FFT Processor with no Multipliers," *2009 IEEE International Conference on Computer Design*, vol. 2009, p. 9, 2009.

[14] Y. P. Bin Zhou and D. Hwang, "Pipeline FFT Architectures Optimized for FPGAs," *International Journal of Reconfigurable Computing*, pp. 485–490, Oct 2009.