Examensarbete

Adaptation of The ePUMA DSP Platform for Coarse Grain Configurability

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

Sepehr Pishgah

LiTH-ISY-EX--11/4540--SE

Linköping 2011
Adaptation of The ePUMA DSP Platform for Coarse Grain Configurability

Examensarbete utfört i Datorteknik vid Tekniska högskolan i Linköping av

Sepehr Pishgah

LiTH-ISY-EX--11/4540--SE

Handledare: **Andreas Ehliar**

ISY, Linköpings universitet

Examinator: **Olle Seger**

ISY, Linköpings universitet

Linköping, 19 December, 2011
Svensk titel
Adaptation of The ePUMA DSP Platform for Coarse Grain Configurability

Författare
Sepehr Pishgah
Author

Sammanfattning
Abstract

Configurable devices have become more and more popular nowadays. This is because they can improve the system performance in many ways. In this thesis work it is studied how introduction of coarse grain configurability can improve the ePUMA, the low power high speed DSP platform, in terms of performance and power consumption. This study takes two DSP algorithms, Fast Fourier Transform (FFT) and FIR filtering as benchmarks to study the effect of this new feature. Architectures are presented for calculation of FFT and FIR filters and it is shown how they can contribute to the system performance. Finally it is suggested to consider coarse grain configurability as an option for improvement of the system.

Nyckelord
Keywords
Coarse Grain Reconfigurable Hardware, CRA, ePUMA
Abstract

Configurable devices have become more and more popular nowadays. This is because they can improve the system performance in many ways. In this thesis work it is studied how introduction of coarse grain configurability can improve the ePUMA, the low power high speed DSP platform, in terms of performance and power consumption. This study takes two DSP algorithms, Fast Fourier Transform (FFT) and FIR filtering as benchmarks to study the effect of this new feature. Architectures are presented for calculation of FFT and FIR filters and it is shown how they can contribute to the system performance. Finally it is suggested to consider coarse grain configurability as an option for improvement of the system.
Acknowledgments

I would like to thank Dr. Andreas Ehliar and Dr. Olle Seger for giving me such an opportunity to improve my vision and skills in design of the digital systems and studying the system from different view points.

I also would like to thank my supervisor, Dr. Andreas Ehliar for helping me through my thesis work providing me with hints and solutions as well as giving me the chance to have a more detailed look to CAD tools used in system design and analysis.

I would also like to thank PhD students in Computer Engineering Division, Andreas Karlsson, Jian Wang and Joar Sohl providing me with help, support and consultant during my thesis work.

I would also like to thank Dr. Olle Seger for his time being my examiner and his support for me as well as his help with my thesis.

I would also like to thank my colleagues in my office having enjoyable work environment and being helpful.

I would also like to thank Mr. Emile Farmer for his help to my English language issues.

I would also like to thank anyone who helped and supported me during my thesis work.

Sepehr, 2011
## Contents

1 Introduction 3
   1.1 Fine Grain and Coarse Grain ........................................ 4
   1.2 The Goal of This Project ........................................... 5
   1.3 Thesis Outline ...................................................... 5

2 ePUMA DSP platform outline 7
   2.1 The Master Unit .................................................... 8
   2.2 SIMD Unit ............................................................ 9
   2.3 Memory Subsystem ................................................... 9

3 Coarse Grain Reconfigurable Hardware 13

4 FFT Algorithms and Background 17
   4.1 Radix-2 FFT Algorithm ............................................. 18
   4.1.1 Radix 2 FFT properties ......................................... 20
   4.2 Radix-4 FFT Algorithm .............................................. 21
   4.3 Pipelined Radix-4 FFT Algorithm ................................... 22
   4.3.1 Pipeline FFT Architectures ..................................... 22

5 FIR Digital Filters Algorithm and Background 27
   5.1 Fast FIR Algorithms ................................................ 27
   5.1.1 Introduction ....................................................... 27
   5.1.2 (2-by-2) Fast FIR Algorithm ................................... 28

6 Proposed FFT Architecture for ePUMA 29
   6.1 The Proposed FFT Architecture for ePUMA ......................... 29
   6.1.1 Methodology ...................................................... 29
   6.1.2 Drawbacks .......................................................... 30
   6.1.3 Benchmark ......................................................... 31
   6.1.4 Verification ...................................................... 36

7 Proposed FIR Digital Filters Architecture for ePUMA 37
   7.1 The Proposed FIR Digital Filters Architecture for ePUMA ....... 37
   7.1.1 Methodology ...................................................... 37
   7.1.2 Drawbacks .......................................................... 41
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Field programmable Gate Array</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ePUMA</td>
<td>Embedded Parallel DSP platform with Unique Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>LVM</td>
<td>Local Vector Memory</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The demands for higher performance and lower power consumption in digital systems are rapidly increasing. As new complex algorithms are designed and developed, they obviously introduce new needs of hardware resources and higher performance of the hardware system. One issue with high performance hardware systems in addition to design complexities, is their power consumption. Power consumption in a digital system should be as low as possible. Assuming digital systems to be generally divided into two main groups, mobile devices and stationary devices, in mobile devices low power consumption leads to longer battery lifetime and for stationary devices low power consumption leads to more energy efficient system. Since the consuming power is transformed into heat, energy efficiency can vary from the energy efficiency of the system itself to even cooling of very large digital systems e.g. data centers, which is not in the scope of this thesis work.

General purpose processors provide good flexibility implementing different algorithms, but they will be left behind in terms of performance once they are compared with Application Specific Integrated Circuits (ASICs) for certain applications. ASICs are also missing flexibility for implementing diverse algorithms. The loss of flexibility at the price of performance creates a gap between general purpose processors and ASICs. Development of reconfigurable hardware has attempted to cover this gap both in terms of performance and flexibility. Reconfigurable devices such as FPGAs are flexible for implementing complex algorithms, having bit level accessibility. Implementation of bit level accessibility leaves the large area overhead for interconnections and high power consumption as the major drawbacks. Flexibility of general purpose processors and robustness of ASICs gathered in reconfigurable hardware have made them very favorable and interesting.

As mentioned earlier in this chapter, reconfigurable devices such as FPGAs are developed with the objective to deliver ASIC like performance. This makes the comparison of ASICs and FPGAs interesting. Characteristics of an FPGA based digital system firstly include configurability. Assume the customer needs a bug to be fixed or the system to be upgraded. In case of configurable devices being used this can be done simply by uploading the bug-fixed or upgraded version of the
design to the FPGA chip. Secondly, the configurable device may be reconfigured
dynamically to match the needs of variety of applications at run time. Finally,
availability of configurable devices and their ease of use have made them a popular
and cheap solution for digital systems. On the other hand ASICs provide higher
performance as well as lower power consumption. High performance of an ASIC
originates from the fact that the design is optimized for a specific application.
Power consumption of an ASIC is lower due to removal of configuration logic as
well as design of the circuit with low power objective. Although low power design
approaches are taken in FPGA design, this is not comparable to ASICs because of
limitations in design of FPGAs and designs using FPGAs. It should also be noted
that one privilege of FPGAs over ASICs is fast prototyping of designs based on
FPGAs rather than ASICs. Designs with ASICs benefit more flexibility comparing
to designs using configurable devices. This is due to the fact that the design is
limited by the resources on an FPGA chip once configurable devices are used,
unlike ASICs where instead of that the designer can implement as much hardware
as desired [1]. Comparison of FPGAs and ASICs from the design perspective is
beyond the scope of this thesis work. Moreover, according to a study in [2] ASICs
and FPGAs can be compared in terms of area, power consumption and delay. For
such a comparison many benchmark designs have been implemented both in an
FPGA and also in standard-cell ASIC. Results provided by [2] indicate that the
area needed for different designs in FPGA is in range of 18 to 35 times larger
than it is for ASICs. The area calculation in [2] was done taking only the core
design area, ignoring the area demands for I/O. This rather large range of ratio
is due to variety of designs implemented, as an example designs using available
blocks such as memory or DSP blocks within an FPGA are more area efficient
than those which are not utilizing these blocks. This is not far from imagination
since these blocks are ASICs by themselves. Comparing the speed, FPGAs fall
behind ASICs being 3.4 to 4.6 times slower. To compensate for the performance,
an ideal parallelization of the design for FPGA implementation can be considered.
The design parallelization translates in larger area of 119 times, and higher power
consumption, while the dynamic power consumption is 14 times higher for FPGAs
than ASICs. These results encourage designers to bring the FPGAs closer to ASICs
even more.

1.1 Fine Grain and Coarse Grain

Reconfigurable devices can be divided into two main groups: fine grain reconfig-
urable hardware (e.g. FPGAs) and coarse grain reconfigurable hardware. The
former gives flexibility for most of applications while the latter is used when there
are certain algorithms to be run on the hardware. In coarse grain reconfigurable
hardware, unnecessary interconnections and configuration logic are removed and
bit level access (fine granularity) is not available any longer. The blocks also may
be optimized for specific target applications resulting in an area efficient and less
power consuming device. Once the bit level access is replaced with word level
access unnecessary interconnection and interconnecting infrastructures (e.g. mul-
tiplexers) can be removed. Furthermore the size of configuration memory used to store the configuration data for the design to be implemented can be reduced. Such improvements result in a more area efficient hardware, better power consumption and improved delay. Utilization of optimized blocks and application specific blocks is also another step toward ASIC like behavior and characteristics for coarse grain reconfigurable hardware.

1.2 The Goal of This Project

At the Computer Engineering Department of Linköping University a new parallel (master-multiSIMD) DSP processor, named ePUMA, is being designed [3]. In this work we are studying how introducing reconfigurability to ePUMA platform can improve the system performance and efficiency. It is known that there are certain tasks with certain behavior to be run on the DSP system, consequently, it is convincing to optimize the system architecture for certain tasks to achieve better performance. Having different tasks to optimize the system architecture for, makes the reconfigurable hardware structure a reasonable option to choose.

1.3 Thesis Outline

First of all an overview of the ePUMA DSP platform is given in chapter 2. Chapter 3 introduces coarse grain configurable devices and in chapter 4 and 5 FFT and FIR filters algorithm and background is presented respectively. Chapter 6 and 7 present proposed architectures for calculation of FFT and FIR filtering and finally some implementation and area estimation results followed by the conclusion and future work are presented in chapters 8 and 9 respectively.
Chapter 2

ePUMA DSP platform outline

The ePUMA\(^1\) DSP platform is a low power, high performance DSP platform targeting future multimedia and communication applications. The ePUMA architecture consists of one master processor and 8 SIMD units [4]. An overview of the ePUMA architecture is presented in Figure 2.1. The master’s processing power is delivered by a 16 bit RISC processor while each SIMD unit is an 8-way, 16-bit SIMD processor with its local memory resources. Communication between SIMD units and the master processor is provided through the Network on Chip, (NoC) [5]. Table 2.1 summarizes some specifications of ePUMA.

![Figure 2.1. ePUMA Architecture. a)Overview of the system. b)Detailed view of one SIMD unit. Figure inspired from [4]](image)

With ever increasing demand for fast and real time signal processing, parallel processing has emerged to be one solution. Parallel solutions mostly include two

\(^1\)ePUMA, embedded Parallel computing architecture with Unique Memory Access, is a project at Linköping University and supported by the Swedish Foundation for Strategic Research (SSF).
categories, general multi-core platforms and application specific hardware architectures. The former not being cost and energy efficient for embedded systems and the latter can only be accepted for certain solutions. On the other hand existing architectures with on-chip scratch pad memory and/or very large register file are also known to be power hungry. Master-multi-SIMD architectures like CELL [5] by STI deliver a good performance for a wide range of applications, however the power consumption might not be pleasing for certain applications. The ePUMA platform is a master-multi-SIMD architecture with optimization done based on predictable computing resulting in efficient power consumption and improved performance. Predictable computing here refers to the fact that most of the computations in high performance embedded systems are based on regular and repetitive data access pattern.

### 2.1 The Master Unit

The master unit, using a 16-bit RISC processor controls the memory system and the SIMD units. The master processor is also responsible for executing sequential tasks while the SIMD units take part in execution of parallelizable segments in an application algorithm. An example of ratio between sequential and parallel segments in an application in this domain can be 1 to 9 [4, 5].

---

2For further reading about CELL please refer to [6]
2.2 SIMD Unit

Major processing power of the ePUMA is contributed by the SIMD units. Each SIMD unit comes with an 8-way 16-bit simple processor including 128-bit wide program memory (PM), 128-bit wide constant memory (CM), and finally three 128-bit wide local vector memory (LVM) [4, 5].

In this project two typical DSP algorithms Fast Fourier Transform (FFT) and Finite Impulse Response (FIR) Digital Filters are focused on. The basic operation for both of them is the Multiply and Accumulate (MAC) operation. The input data for these applications are complex values so complex operations will be needed. For the sake of simplicity, 1/4 SIMD unit data path is shown in Figure 2.2 showing that one complex MAC operation can be done in 1/4 SIMD unit, hence, 4 complex operation per each SIMD unit.

2.3 Memory Subsystem

The memory subsystem is designed with objective of reducing communication overhead and data access cost which is achievable once memory access patterns are regular and predictable. Illustrated in Figure 2.3, the memory hierarchy is composed of three levels from off-chip main memory down to local vector memory and finally the registers with the shortest access time. Each SIMD unit uses 3 eight-bank single port Local Vector Memory (LVM) modules each sizes 5KW\(^3\). An overview of ePUMA’s local memories is given in Table 2.2. Data transaction between SIMD core and the main memory is done in two steps as it can be observed in Figure 2.3. To hide the data transaction overhead the memory subsystem features ping-pong buffer implementation. In ping-pong buffering, one LVM is used for data communication and the other two are dedicated to the SIMD core once data communication is complete the LVMs will swap [5].

\(^3\) 1 Word = 128 bits
Figure 2.2. 1/4 SIMD Unit Data Path. Figure provided by the department[7].
Figure 2.3. ePUMA Memory Hierarchy. Figure inspired from [5]
Chapter 3

Coarse Grain Reconfigurable Hardware

Reconfigurable hardware structures may generally be divided into two main groups, fine grain reconfigurable hardware like FPGAs and coarse grain reconfigurable hardware. Coarse grain reconfigurable architectures have been very attractive recently. Unlike fine grain reconfigurable hardware architectures, the data path width is greater than 1 bit in coarse grain reconfigurable hardware removing the unnecessary routing area overhead. Word level data path width also reduces the configuration memory demand as well as configuration time along side the placement and routing overhead [8]. There are different architectures for coarse grain reconfigurable hardwares\(^1\) but a universal reconfigurable hardware is more or less impractical and presented architectures are more domain specific e.g. image processing, wireless communication or multimedia etc.

This thesis work attempts to look at ePUMA DSP platform as a coarse grain reconfigurable hardware to study how it can benefit from a coarse grain reconfigurable hardware perspective.

Fine granularity of reconfigurable hardware contributes to high flexibility of such architectures, easily implementing irregularities in different architectures. On the other hand, drawbacks of this type include large routing overhead, high power consumption, and large configuration memory demands. However having the fact that specific architectures are considered for implementation in advanced, it is not far to imagine that the designer would remove the none-required interconnection and improve the logic blocks for particular architectures. Such an approach preserves flexibility in the system and also avoids the drawbacks mentioned earlier above [9]. As an example, RaPiD architecture was designed for DSP domain applications, having dedicated 16-bit multipliers, ALU and RAM on a programmable and pipelined word-wise data bus [10]. Another example is MORA (Multimedia Oriented Reconfigurable Array) proving enhanced performance and area efficiency comparing with FPGAs [11]. In MORA architecture reconfigurable cells are ar-

\(^1\)a brief description of different coarse grain reconfigurable architecture can be found in [8]
ranged in an 8x8 matrix on a hierarchical reconfigurable network. Each cell comes with its dedicated memory. Performance in this architecture is decided by the memory access time and it is considered each operation is executed within one clock cycle, with memory being the slowest part in the data path, the performance is decided accordingly. The interconnecting network in MORA architecture is to be configured statically as in FPGAs [11], however dynamic configuration is also possible as in PipeRench [8]. Coarse grain reconfigurable architectures may generally be categorized into three different groups from interconnection perspective [8]:

- **Primarily Mesh-Based**, in which processing arrays are connected horizontally and vertically, supporting robust communication between arrays, elements may be connected diagonally in addition to horizontal and vertical neighbor connectivity for enhanced communication. Known examples of such an architecture are DP-FPGA and The KressArray. An illustration of primarily mesh based architecture is shown in Figure 3.1.

- **Architectures Based on Linear Arrays**, shares some basics with the previous architecture, the primarily mesh based, however it attempts to support pipelined structures. In this architecture extra routing infrastructure for bypassing processing parts or a complete array is considered to support pipelined structures. Such an architecture can be found in RaPiD or PipeRench. Figure 3.2 provides an overview of linear array based architecture.

- **Crossbar-Based Architectures**, having a complete crossbar switch, are the most handy to route architectures. Due to complexities in implementing the crossbar, examples of this architecture features a reduced crossbars as in PADDI-1/2. An outline of this architecture is presented in Figure 3.3.

Said earlier in this chapter, coarse grain reconfigurable data paths may be optimized for certain applications. This implies the correlation of application and the hardware bringing coarse grain reconfigurable hardwares closer to ASIC. As mentioned in section 1 reconfigurable hardwares are introduced to remove the gap between general purpose processors and ASIC [2], now the gap is being narrowed even more by bringing the coarse grain reconfigurable hardwares into the picture. Architectural optimization for coarse grain reconfigurable data paths include operation level parallelization, pipelined architectures and operation chaining. These optimization techniques may be taken individually or in form of combination of the concepts [12].

In this work it is assumed that a custom structure for interconnection will be implemented, making sure to meet the demand for better performance. In terms of architectural optimization, processing blocks are already optimized for the application and it is attempted to combine operation parallelization and operation pipelining for further optimization.
Figure 3.1. Outline of Primarily Mesh-Based. *Figure inspired from [8]*

Figure 3.2. Outline of Architectures Based on Linear Arrays. *Figure inspired from [13]*
Figure 3.3. Outline of Crossbar-Based Architectures. *Figure inspired from [14]*
Chapter 4

FFT Algorithms and Background

To study how configurability can contribute to the performance of the ePUMA DSP platform, two widely used algorithms, FFT and FIR filtering, are taken to study. A brief description of Fast Fourier Transform (FFT) is presented in this section. The Pipeline FFT algorithm taken in this study, is also presented later in this section.

Discrete Fourier Transform (DFT) is one of the most common algorithms in DSP applications. By definition, DFT of a finite length sequence of samples is as follows [15].

\[
X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \quad 0 \leq k \leq N - 1 \quad \text{Where} \quad W_N = e^{-j2\pi/N} \quad (4.1)
\]

And in a similarly the Inverse Discrete Fourier Transform (IDFT) is as follows.

\[
x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)W_N^{-kn} \quad 0 \leq n \leq N - 1 \quad \text{Where} \quad W_N = e^{-j2\pi/N} \quad (4.2)
\]

Having the same basic calculation for DFT and IDFT, an improvement for either DFT or IDFT will contribute to both. Looking into (4.1), calculation of each \( X(k) \) takes \( N \) complex multiplication and \( N - 1 \) complex addition, hence, \( N^2 \) complex multiplication and \( N^2 - N \) addition\(^1\).

\(^1\)Please note that 1 complex multiplication typically involves 4 real multiplications and 1 complex addition takes 2 real additions, however there are methods which take 3 multiplications and 5 addition like Gauss’s algorithm a description of which can be found in [16] for example.
4.1 Radix-2 FFT Algorithm

Calculation of DFT according to (4.1) can be simplified due to two properties of the phase factor, $W_N$. Phase factor is also known as twiddle factor.

$$W_{N}^{k+N/2} = -W_{N}^{k} \quad \text{Symmetry property} \quad (4.3)$$
$$W_{N}^{k+N} = W_{N}^{k} \quad \text{Periodicity property} \quad (4.4)$$

FFT technique attempts to break original sequence into smaller sequences in such a way that the combination of those results the same as the initial sequence DFT. Assuming we have an even number of samples for DFT, we can break them into two parts, even and odd. This way the number of multiplications are reduced to $(N/2)^2 \times 2 = N^2/2$:

$$X(k) = \sum_{n=0}^{N/2-1} x(2n)W_{N/2}^{2nk} + \sum_{n=0}^{N/2-1} x(2n+1)W_{N}^{(2n+1)k} \quad 0 \leq k \leq N - 1 \quad (4.5)$$

Hence, according to (4.3) and (4.4):

$$X(k + N/2) = \sum_{n=0}^{N/2-1} x(2n)W_{N/2}^{nk} - W_{N}^{k} \sum_{n=0}^{N/2-1} x(2n+1)W_{N/2}^{nk} \quad 0 \leq k \leq N/2 - 1 \quad (4.7)$$

Further iteration of the same decomposition can reduce the complexity of $N/2$-point DFT to $N/4$-point ($N/2$ is even). With the assumption that $N$ is a power of 2, meaning $N/2$ is possible at every decomposition stage, the decomposition can be continued until 2-point DFTs are achieved. The 2-point DFT is given by (4.8) and this approach is known as Radix 2 FFT [15].

$$F(0) = f(0) + f(1)W_{N}^{0}$$
$$F(1) = f(0) + f(1)W_{N}^{N/2} \quad (4.8)$$

$f(n)$ is a 2 point sequence to be transformed. According to Equation (4.2) $W_{N}^{0} = 1$, $W_{N}^{N/2} = -1$, hence, no multiplication is needed. Figure 4.2 illustrates an 8-point Radix 2 FFT. As it can be seen in Figure 4.2, the basic operation in FFT can be explained as in (4.9) which is called the butterfly. The butterfly operation flow graph is shown in Figure 4.1.

$$X = A + BW_{N}^{k}$$
$$Y = A - BW_{N}^{k} \quad (4.9)$$
4.1 Radix-2 FFT Algorithm

**Figure 4.1.** 2 point butterfly

**Figure 4.2.** 8 point radix 2 FFT. Figure borrowed from [15]. Each X shaped element in this figure is representing a butterfly operation presented in Figure 4.1.
4.1.1 Radix 2 FFT properties

Reviewing Figure 4.2, at each stage \( N/2 \) complex multiplications are required. Since there are \( \log_2 N \) stages totally, \((N/2)\log_2 N\) complex multiplications are to be done rather than \( N^2 \) complex multiplications as mentioned in Section 4.2. The algorithm presented in Section 4.1 is known as Decimation in Time (DIT) FFT, since at each step the input sequence is reordered for processing. However another FFT approach is known as Decimation in Frequency (DIF). In this approach, the input sequence is divided into 2 parts at first stage:

\[
X(k) = \sum_{n=0}^{N/2-1} x(n)W_N^{nk} + \sum_{n=N/2}^{N-1} x(n)W_N^{nk} \tag{4.10}
\]

\[
= \sum_{n=0}^{N/2-1} x(n)W_N^{nk} + \sum_{n=0}^{N/2-1} x(n+N/2)W_N^{(n+N/2)k} \tag{4.11}
\]

let \( W_N^{N/2} = e^{-j\pi k} \).

\[
X(k) = \sum_{n=0}^{N/2-1} [x(n) + e^{-j\pi k}x(n+N/2)]W_N^{nk} \tag{4.12}
\]

Equation (4.12) shows the complexity reduction to \( N/2 \) point FFT. Rewriting Equation (4.12) in form of odd and even terms we have:

\[
X(2k) = \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)](W_N^2)^{nk} \quad 0 \leq k \leq N/2 - 1
\]

\[
= \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)]W_N^{nk} \tag{4.13}
\]

\[
X(2k + 1) = \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)]W_N^{n(2k+1)} \quad 0 \leq k \leq N/2 - 1
\]

\[
= \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)]W_N^{n}W_N^{nk} \tag{4.14}
\]

Similarly this procedure can be repeated for \( N/2 \) point DFT until the sequence ends in a 2 point DFT sets. Finally, with same approach as 4.7 complete DFT can be calculated. This should be noted that the output of each stage will be reordered each time. This approach is known as Decimation in Frequency due to,

---

2Please note that \( W_N^0, W_N^{N/4}, W_N^{N/2}, W_N^{3N/4}, \ldots \) can also be translated in such a way that only complex additions and subtractions operations are involved and consequently reducing the number of multiplication even more, but once the number of input samples, \( N \), increases, \((N/2)\log_2 N\) is the well approximated number of nontrivial multiplications. The term nontrivial multiplication refers to multiplications that cannot be replaced by single additions or subtractions.
reordering of the output in frequency domain. Figure 4.3 illustrates an 8-point DIF FFT.

DIT and DIF FFT are the same in terms of computation complexity however, in DIT the input is bit reversed but the output is in order, while in DIF the input is in order but the output in bit reversed format. Comparing the butterfly in DIT FFT and DIF FFT, in DIT multiplication takes place prior to the add-sub operation but in DIF after the add-sub operation.

![Figure 4.3. 8 point radix 2 DIF FFT. Figure borrowed from [15]](image)

### 4.2 Radix-4 FFT Algorithm

In an analogous fashion to radix-2, once the number of samples are power of 4 the input sequence can be broken into 4 parts recursively until it ends in a sequence of length 4 DFT. An illustration of this approach is depicted in Figure 4.4 and 4.5 [15, 17]. In this approach the basic operation in the butterfly will look like Equation (4.15) in matrix format [17]. Radix-4 FFT reduces the computation complexity of the transform to $N \log_4 N$, despite any $n^4$ point FFT can be calculated using radix-2 approach.

$$
\begin{bmatrix}
X_0 \\
X_1 \\
X_2 \\
X_3
\end{bmatrix} =
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -j & -1 & j \\
1 & -1 & 1 & -1 \\
1 & j & -1 & -j
\end{bmatrix}
\begin{bmatrix}
W_0^{Nx_0} \\
W_1^{x_1} \\
W_2^{x_2} \\
W_3^{x_3}
\end{bmatrix}
$$

(4.15)
4.3 Pipelined Radix-4 FFT Algorithm

As depicted in Figure 4.4, at each stage the radix-4 butterflies can be calculated separately, not requiring any particular order of calculations. Therefore, introducing parallelism to the algorithm, which is so beneficial as long as there is the need to run the FFT as fast as possible. On the other hand, calculation of one stage is not necessarily dependent to completion of its previous stage, once certain intermediate points are calculated, the butterfly element in the next stage can be initiated. Based on these properties, a new class of FFT parallel algorithm is defined as, The Pipelined Radix-4 FFT Algorithm\(^3\). The minimum parallelism introduced by this approach is estimated by \(\log_4 N\), meaning \(\log_4 N\) radix-4 butterflies are calculated simultaneously in a pipelined fashion.

4.3.1 Pipeline FFT Architectures

Here a brief overview of different pipeline FFT architectures are presented. For further information reader is referred to the references.

**R2MDC:** Radix-2 Multi-path Delay Commutator [15, 18] takes two parallel data streams as input. The inputs shall be with correct distance from each other for each pair of samples applied to the butterfly elements. The required distance is achieved using delay elements. In this approach the multipliers and the butterflies usage is 50%. \(\log_2 N - 2\) multipliers, \(\log_2 N\) radix-2 butterflies and \(3/2N - 2\) delay element are required in this approach.

\(^3\)pipelined FFT algorithm can be applied for any radix.
Figure 4.5. 64 point radix 4 DIF FFT. Figure borrowed from [15]
**R2SDF:** In Radix-2 Single-path Delay Feedback [19, 18] architecture, a single data stream is fed to the multiplier at every stage. In terms of number of multipliers and butterflies it is the same as **R2MDC**. This approach is more efficient in terms of memory requirements, $N - 1$ delay elements. An illustration of this architecture is shown in Figure 4.6.

**R4SDF:** Radix-4 Single-path Delay Feedback [20, 18], is the radix-4 version of **R2SDF**, using CORDIC\(^4\) iteration. The multipliers are used more often, 75%, but the butterfly usage is decreased to 25%. In this algorithm $\log_4 N - 1$ multipliers, $\log_4 N$ radix-4 butterflies and a storage of size $N - 1$ are required.

**R4MDC:** Radix-4 Multi-path Delay Commutator [15, 18] is similar to **R2MDC**, but for radix 4. This is the architecture used for initial VLSI implementation of pipeline FFT processor. It requires $3\log_4 N$ multipliers $\log_4 N$ radix-4 butterflies and $5/2N - 4$ delay elements. One disadvantage of this algorithm is the 25% utilization of all components. An illustration of this architecture is shown in Figure 4.7.

**R4SDC:** The Radix-4 Single-path Delay Commutator [21, 18] uses modified radix-4 algorithm with programmable $1/4$ radix-4 butterflies, giving 75% multipliers utilization. Memory requirements is reduced to $2N - 2$ compared to its predecessor. This algorithm is recently used in HDTV applications [22].

**R2\(^2\)SDF:** The Radix-2\(^2\) Single-path Delay Feedback [18] uses radix-2\(^2\) algorithm which maintains the same butterfly structure as radix-2, as well as radix-4 multiplicative complexity. This results in $\log_4 N - 1$ complex multipliers and $N - 1$ memory requirement.

Having a reviewed different architectures above, it is realized that architectures with delay feedback are more memory efficient rather than those with delay commutators. Comparison of different radix also indicates that, despite the simpler architecture of radix-2 butterfly, radix-4 butterfly have a higher multipliers usage [18]. In this work it is considered that enough multipliers are available and the major concern is how to use resources more efficiently.

---

\(^4\)CORDIC: COordinate Rotation Digital Computer, simple and efficient algorithm used when hardware multiplier is not available, the algorithm only requires addition, subtraction, bit shift and table lookup.
4.3 Pipelined Radix-4 FFT Algorithm

**Figure 4.6.** Radix 2, 32 point pipeline FFT (R2SDF). *Figure inspired from [18]*

**Figure 4.7.** Radix 4, 64 point pipeline FFT (R4MDC). *Figure borrowed from [15]*
Chapter 5

FIR Digital Filters
Algorithm and Background

As mentioned in previous section, to study how configurability can contribute to the performance of the ePUMA DSP platform, two widely used algorithm, FFT and FIR filtering, are taken to study. In this section an overview of FIR filters and their algorithms suitable for ePUMA platform are presented. It is attempted to choose parallel algorithms to benefit from ePUMA resources as much as possible.

There are two class of digital filters, Finite-length Impulse Response (FIR) and Infinite-length Impulse Response (IIR). FIR filters are featuring stability, linear phase response and short data word length requirement, but compared to IIR filters with the same specifications, they are usually of higher order [23]. Filtering of a signal by an N-tap FIR filter is done by convolving the impulse response\(^1\) of the filter and the input samples. The discrete convolution of \(h\) and \(x\) is given in Equation (5.1).

\[
y(n) = [h \ast x](n) = \sum_{k=0}^{N-1} h(k)x(n - k), \quad n = 0, 1, 2, \ldots, \infty
\]

There are different structures realizing FIR filters such as Direct Form, Transposed Direct Form, etc.. Such architectures mainly use adders, multipliers and delay elements [23]. However, there are other FIR digital filters structures introducing parallelism to the calculation flow. One such an approach is the Fast FIR Algorithm.

5.1 Fast FIR Algorithms

5.1.1 Introduction

Algorithm parallelism increases the performance of the system, so does here for calculation of FIR filters. As a drawback, the increase of hardware resources should

\(^1\)Impulse response is obtained once an impulse is applied to the filter as an input sequence [23].
be considered. In this work, since the hardware is already available in ePUMA platform, the demand for hardware resources is not of major concern however, using the resources more efficiently is the primary challenge. The Fast FIR Algorithm (FFA) claims reducing complexity of parallel filter structures. With this approach an $L$-parallel filter will require $2L - 1$ filtering operations of length $(N/L)$ rather than $L^2$ filtering operations of the same length $(N/L)$ [24].

Generally an (n-by-n) FFA consists of n filters, $H_0, H_1, \ldots and H_{n-1}$, of length $N/n$ each. These filters are the polyphase decomposition of the original filter $H$ of length $N$. The combination of these $n$ filters forms the original $H$ filter [24].

### 5.1.2 (2-by-2) Fast FIR Algorithm

Figure 5.1 shows the 2-parallel FIR filtering structure with four filtering operation, the reduced complexity 2-parallel FFA is depicted in Figure 5.2. For further reading please refer to [24].

**Figure 5.1.** Traditional 2-Parallel FIR Filter Implementation.inspired from [24]

**Figure 5.2.** Reduced-Complexity 2-Parallel Fast FIR Filter Implementation.inspired from [24]
Chapter 6

Proposed FFT Architecture for ePUMA

In this section the proposed architecture for FFT calculation on ePUMA platform is introduced and analyzed. The new approach is compared with the old approach using a benchmark problem. Finally verification of the new approach is presented.

6.1 The Proposed FFT Architecture for ePUMA

As mentioned in Section 2, the ePUMA DSP platform has eight 8-way SIMD units with its dedicated local vector memories. An inspection of the SIMD unit reveals that with regard to the number of multipliers (16 real multipliers) and the rest of the data path each SIMD unit can easily fit a radix-4 butterfly element, considering the input samples are in complex format. On the other hand, studying of radix-4 FFT algorithm in Section 4.2 suggests that the pipelined algorithms benefit from parallel nature of the radix-4 FFT algorithm. These are the bases of proposing pipeline FFT architecture for calculation of FFT on ePUMA platform.

6.1.1 Methodology

The proposed FFT architecture uses the pipelined FFT algorithm with an architecture similar to the R4MDC, which was presented earlier in Section 4.3.1. The modification of the algorithm includes replacement of the delay commutators in R4MDC with the SIMD unit’s vector memory.

In this thesis work the objective is to study the introduction of configurability feature to ePUMA DSP platform. It is assumed that changes in the data path are acceptable, as long as they are not very costly and also, that they can improve the system in terms of performance and power consumption. Mentioned earlier, each SIMD processor is capable of calculating one radix-4 butterfly. Looking into Figure 6.1, the multi-path delay commutators are still missing in the ePUMA architecture. The delay commutators are used to reorder the output of radix-4
butterfly in such a way that they are in the right order to be applied to the next butterfly element (please refer to Figure 4.5). Instead of using delay commutators the data can be written into the LVM (SIMD unit’s local vector memory), later to be read by the next SIMD unit. The reordering here shall be done implicitly by reading from the correct-distance address in the LVM. An overview of the new approach is depicted in Figure 6.2.

![Figure 6.1. Radix 4, 64 point pipeline FFT (R4MDC).](image)

**Figure 6.1.** Radix 4, 64 point pipeline FFT (R4MDC).

![Figure 6.2. Overview of the new FFT approach. For the sake of simplicity, only one stage of the pipeline is presented.](image)

**Figure 6.2.** Overview of the new FFT approach. For the sake of simplicity, only one stage of the pipeline is presented.

### 6.1.2 Drawbacks

The new approach proposed in section 6.1.1, requires the LVMs to be shared between two SIMD processors. Access to an LVM should be supported in such a way that the previous SIMD processor writes into the LVM and the next SIMD processor reads from the same LVM. Thus, each SIMD processor reads from the previous stage LVM and writes into the next stage LVM. According to the current architecture of the ePUMA each SIMD units can access its own LVM and data communication between SIMD units is facilitated by the Network on Chip (NoC) through LVMs. However, this is not the best way it can be done, due to overhead for transferring the data between SIMD units which is not pleasant at all. The solution now is to reconfigure the data path in such a way that each SIMD unit can directly write into the corresponding LVM in the neighboring SIMD unit. Having the fact that the ring network in the ePUMA architecture has a 128 bit width, it is promising to reconfigure the data path in such a way that the output of each SIMD unit is routed to the input of the LVM in neighboring SIMD unit. This maybe done either through the network on chip infrastructure or by adding
extra path for this data communication. As mentioned in Section 6.1.1, the LVM is serving as the delay commutator of the \textbf{R4MDC} architecture for the proposed approach. Therefore, to meet the addressing required for this purpose, a simple address generation and control unit is needed, behavior of which is presented in Appendix A.2.

Another issue with this approach is that, the optimum efficiency of the pipelined FFT is achieved when all pipeline stages are run in parallel. The Pipelined algorithm requires 1 read and 1 write at the same time from and into the LVM. This means a dual port memory system is required for LVMs, which is not available in ePUMA current architecture, and also, it is expensive to implement. Although such a problem can be solved by defining read and write cycles, but compromising the performance of the pipeline algorithm is not desirable. Nonetheless the ePUMA memory system for SIMD units comes with three 8-bank local vector memories (see Section 2.3), it can be considered that two LVMs replace the LVM shared between two SIMD units, one for reading from and one for writing to. These two LVMs should swap their role every clock cycle, hence, making simultaneous read and write possible.

\subsection*{6.1.3 Benchmark}

Benchmarking of the proposed architecture was done comparing it with the same FFT points as it was taken for the previous traditional approach. Comparison metrics here include, the number of read/write, the number of calculation cycles and the number of involved SIMD units.

In this section, first, the old FFT approach is briefly explained and later followed by comparison results. The FFT size is 4K complex samples.

\textbf{The Old Approach} The old non-pipelined approach takes radix-4 FFT algorithm and utilizes one SIMD unit with the input samples stored in one LVM. The number of stages\textsuperscript{1} is 6 and there are 1024 radix-4 butterflies\textsuperscript{2} in each stage. This approach takes the input and performs radix-4 butterfly operation on every set of input samples and store the intermediate results in the other LVM (see Figure 4.5). Here there are 1024 sets, hence, 1024 butterfly operations. Once the computation of one stage is complete, the LVMs are swapped and the calculation of radix-4 butterflies of the second stage will start, writing the second round intermediate results in the very first LVM. The procedure of the calculation of radix-4 butterflies and switching back and forth between the LVMs is continued until last stage that the FFT results are available.

The new pipelined approach however, takes the pipeline algorithm with the data path configured to best fit the algorithm. The number of stages are obviously six resulting in utilization of six SIMD units. Comparison results of the new approach and the old approach is presented in Table 6.1.

As it can be seen in Table 6.1, the number of reads has been increased by 34\%. In every stages, there are certain number of butterflies with identical twiddle factors,

\begin{align*}
\text{\textsuperscript{1}} & log_4 4096 = 6 \\
\text{\textsuperscript{2}} & 4096/4(radix - 4)
\end{align*}
Table 6.1. The comparison of new approach and the old approach

<table>
<thead>
<tr>
<th>Approach</th>
<th>no. SIMD units</th>
<th>no. Twiddle Factors Read</th>
<th>no. Data Values Read</th>
<th>no. Writes</th>
<th>no. Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old approach</td>
<td>1</td>
<td>1365</td>
<td>6144</td>
<td>6144</td>
<td>7670</td>
</tr>
<tr>
<td>New approach</td>
<td>6</td>
<td>3840</td>
<td>6144</td>
<td>6144</td>
<td>1024</td>
</tr>
</tbody>
</table>

- Total no. Reads includes the no.Read for twiddle factor as well as the data read at every stage.
- The Total no.Read in the New approach is more than it is in the Old approach due to reuse of twiddle factors in the Old approach.
- The no.Cycles for the first FFT in the New approach is 2048 due pipeline loading overhead.

which the old approach benefits from this characteristic and reduces the number of reads from 11,264 to 7,509 (see Equation (6.1)). On the other hand, in each stage, 1/4 of twiddle factors are zero which reduces the number of memory accesses to 9,984 (see Equation (6.2)). The number of writes in both approaches are the same.

In the old approach intermediate data should be written into the memory. In the new approach, the write sequence of each stage does not match the read sequence of the next stage. This dictates that the result should be stored somewhere in such a way that it can be passed to next stage with the correct distance (see Section 6.1.1). One other issue in terms of memory access is that in the new approach, there will be no need for accessing the program memory for calculation of the FFT like in the old approach, which can translate in improved power consumption. A Finite State Machine (FSM) will be responsible for the flow of FFT calculation. The number of cycles for calculation of FFT has improved 6 times due to parallelization of the algorithm and usage of 6 SIMD units instead of 1 SIMD unit. Of course, due to the pipelined nature of this approach, calculation of one single 4K point FFT will take 2,053 cycles to complete. In the case more than on FFT should be calculated, 1,024 cycles are required to finish every FFT.

\[
6 \times 1024 \quad \text{Total no. of samples read.} \\
+ 5 \times 1024 \quad \text{Total no. of twiddle factors read The twiddle factors for the first stage are zero} \\
= 11,264 \quad (6.1)
\]

\[
6 \times 1024 \quad \text{Total no. of samples read.} \\
+ 5 \times 1024 \times 0.75 \quad \text{Total no. of twiddle factors read 25% are zeros as well as the first stage.} \\
= 9,984 \quad (6.2)
\]
The new approach using the pipeline FFT algorithm demonstrates its performance once the number of FFT sets\(^3\) increases, Figure 6.3 illustrates this. Number of SIMD units considered for the comparison of pipelined and non-pipelined approach are 6. This is imposed by the fact that pipeline approach utilizes 6 SIMD units for the 4K point FFT. Utilizing the full system capacity, 6 SIMD units can be dedicated for pipeline approach and the 2 remaining can be configured to calculate non-pipelined FFT. The cycle count vs. number of FFT sets is illustrated in Figure 6.5.

![Figure 6.3](image-url)

**Figure 6.3.** Cycle comparison of pipelined approach and non-pipelined approach. Please note that the graphs are ideal and in reality the graph may look slightly different. The difference originates from the delay overhead for memory access in the ePUMA memory hierarchy.

- In the pipelined approach for 4k point FFT 6 SIMD units are used so the number for SIMD units taken for comparing the non-pipelined approach are 6.
- The step like shape of the graph for non-pipelined approach is because SIMD units are calculating the FFTs in parallel e.g. 3 FFTs and 6 FFTs will require same number of cycles but different number of SIMD units are used, 3 for the former and 6 for the latter.

\(^3\)FFT set here refers to a 4K sample data set Fourier transform of which should be calculated e.g. a data packet of size 4K samples
Figure 6.4. Cycle comparison of pipelined approach and non-pipelined approach using 8 SIMD units. Please note that the graphs are ideal and in reality the graph may look slightly different. The difference originates from the delay overhead for memory access in the ePUMA memory hierarchy.
Figure 6.5. Cycle comparison of pipelined approach and non-pipelined approach using 8 SIMD units vs using 6 SIMD units and also when the system is running at its full capacity. Please note that the graphs are ideal and in reality the graph may look slightly different. The difference originates from the delay overhead for memory access in the ePUMA memory hierarchy.
Cycle Estimation

Cycle estimation presented here is done by formulating and plotting the graphs in Matlab which its corresponding code and formula can be found in Appendix A.6.

6.1.4 Verification

Verification of the proposed architecture is done in matlab. Firstly, the calculation of one single butterfly in one SIMD unit is implemented as a function performing the butterfly operation on 4 data samples and corresponding twiddle factors. The twiddle factors are calculated run time but in practices they are expected to be pre-calculated and stored in the memory.

In the main part of the code, the butterfly function is called in a pipelined fashion, calling the last stage first. This is done since the matlab programs are executed sequentially.

Memory access is simulated also with matlab functions, to make sure that read and write from and into the same memory do not occur at the same time. As mentioned earlier, for each stage two LVMs are considered, one to be read from and one to be written into at the same time. To verify this, the memory functions set a parameter once they access the memory, hence, making it possible to verify that the same memory is not accessed at the same time.

Finally the results from the pipelined FFT algorithm is compared against the matlab FFT function in order to verify the correctness of the results. Source codes of the implementations can be found in Appendix A.
Chapter 7

Proposed FIR Digital Filters Architecture for ePUMA

In this section the proposed architecture for FIR Digital Filters calculation on ePUMA platform is introduced and analyzed. The new approach is compared with the old approach for calculating FIR on ePUMA DSP platform using a benchmark problem. Finally verification of the new approach is presented.

7.1 The Proposed FIR Digital Filters Architecture for ePUMA

Inspection of the ePUMA Architecture and examination of an N-tap FIR filter in Equation (5.1) suggests that, with support of parallel calculation in ePUMA, use of parallel FIR filter algorithms improves the performance.

7.1.1 Methodology

The current method of calculation of the FIR filter is done using vmac operation, which is basically vector multiply and accumulate operation. FIR filters of any order can be calculated by iterating this operation as many as the filter order. The calculation of FIR filter may involve 1-8 SIMD unit(s) producing 4-32 results on every completion of vmac iterations. One problem with this approach is that the memory is not read efficiently. Looking at Equation (7.1), it is observed that the filter coefficients should be read for each iteration, besides, only one new data sample is needed while the other 3 could have been reused. vmac is done reading \(x_n, x_{n+1}, x_{n+2}, x_{n+3}\) and \(x_{n-1}, x_n, x_{n+1}, x_{n+2}\) and \(\ldots\). In the new approach proposed for FIR filter calculation, it is attempted to remove unnecessary memory access both for the filter coefficients and for the data samples.

In this approach the data path is changed in such a way that the data samples are shifted to the next register instead of being read again. It also maintains the
coefficients as much as possible. Furthermore, parallel FIR filter algorithm and FFA algorithm are also considered for a better improvement.

\[
\begin{align*}
    y_n &= h_1x_n + h_2x_{n-1} + h_3x_{n-2} + \ldots \\
    y_{n+1} &= h_1x_{n+1} + h_2x_{n} + h_3x_{n-1} + \ldots \\
    y_{n+2} &= h_1x_{n+2} + h_2x_{n+1} + h_3x_{n} + \ldots \\
    y_{n+3} &= h_1x_{n+3} + h_2x_{n+2} + h_3x_{n+1} + \ldots
\end{align*}
\] (7.1)

**The new approaches:** The memory band width of the SIMD units (LVM) can provide 4 data samples in complex format. On the other hand, each SIMD unit is capable of performing 4 complex multiplications at a time. Such an architecture suggests that, an FIR filter of 4th order can be calculated using 4 SIMD processors very efficiently. This is the basis of the proposed methods for FIR filter calculation. In the first method, *The Parallel Approach*, the initial FIR filter is decomposed into 4th order FIR filters using traditional parallel filter Algorithm. The filtering is done using SIMD processors in parallel. An illustration of this is depicted in Figure 7.1. Regarding to Figure 5.1, \(H_0\) on even samples and \(H_1\) on odd samples are performed concurrently in the first phase. In the second phase, \(H_1\) on even samples and \(H_0\) on odd samples. Figure 7.2 illustrates the input sample order required for the filtering as well as the proposed data path for supporting algorithm. Table 7.1 to 7.4 represents the contents of LVMs. The LVMs are read line after line and the data read is applied to SIMD units for filter calculation. Figure 7.2, represents the overview of the system for FIR filtering calculation. please note that the output of each multiplexer is registered in the input registers of SIMD unit, no new registers are added. Assume that all the multiplexers in the Figure 7.2 behave in the same

![Figure 7.1](image-url). Overview of the new FIR approach The Parallel approach.
way, either shifting their preceding register value or passing the new value. In the first cycle, 16 data sample are applied to 4 SIMD units at the same time, 4 samples per each SIMD unit, as shown in Figure 7.2 at the output of splitters. In the first cycle, each SIMD unit is accessing its LVM, hence, the multiplexers should be set in not-shifting mode. In the next cycle however, the data path needs to be reconfigured dynamically, in such a way that 4 data samples are read from one LVM (1.LVM0 in Figure 7.2). This makes the multiplexers to be configured in the shift mode. As a result 4 samples read from one LVM are distributed among four SIMD units, while the samples registered at the input of each SIMD unit is shifted one place to the right. The data path now remains unchanged for the third, forth and fifth cycle until it is reconfigured back to its state in the first cycle mentioned earlier. This procedure is repeated until filtering of all samples are done. Writing the data back is assumed to be done every 4 cycles, meanwhile the output data will be stored at the SIMD’s output register. The write back is done simultaneous with memory read from two different LVMs.

| Table 7.1 | Memory Contents for LVMs |
| SIMD Unit 0 |
| Adr. | Contents |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 2 | 7 | 12 | 17 |
| 2 | 3 | 8 | 13 | 18 |
| 3 | 4 | 9 | 14 | 19 |
| 4 | 5 | 10 | 15 | 20 |
| 6 | 21 | 20 | 19 | 18 |
| 7 | 22 | 27 | 32 | 37 |

| Table 7.2 | Memory Contents for LVMs |
| SIMD Unit 1 |
| Adr. | Contents |
| 0 | 6 | 5 | 4 | 3 |
| 1 | 26 | 25 | 24 | 23 |

| Table 7.3 | Memory Contents for LVMs |
| SIMD Unit 2 |
| Adr. | Contents |
| 0 | 11 | 10 | 9 | 8 |
| 1 | 31 | 30 | 29 | 28 |

| Table 7.4 | Memory Contents for LVMs |
| SIMD Unit 3 |
| Adr. | Contents |
| 0 | 16 | 15 | 14 | 13 |
| 1 | 36 | 35 | 34 | 33 |

The other approach, The FFA Approach, takes the Fast FIR Algorithm performing the vmac operation as in the old approach (see Figure 5.2). The advantage here is breaking larger order filter into smaller one, hence, reducing the number of vmac operation required and consequently the total execution cycles. The data path here should also be modified in such a way that it will fit the architecture in Figure 5.2. Since filter order is assumed to be larger than 3, taking vmac operation, there remains at least 3 clock cycles in which 1/4 of the SIMD unit is idle. Modification in SIMD data path should be in such a way that the output of \( H_0, H_1, \) and \( H_0 + H_1 \) are fed into the first and second adder layers (see Figure 2.1) in the idling 1/4 SIMD unit.
Proposed FIR Digital Filters Architecture for ePUMA

Indices of complex data samples to be applied to SIMD units. Each sample is composed of 16 bits real and 16 bits imaginary parts.

<table>
<thead>
<tr>
<th>Cyc.</th>
<th>SEL</th>
<th>SIMD 0</th>
<th>SIMD 1</th>
<th>SIMD 2</th>
<th>SIMD 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1 0 0 0</td>
<td>6 5 4 3</td>
<td>11 10 9 8</td>
<td>16 15 14 13</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2 1 0 0</td>
<td>7 6 5 4</td>
<td>12 11 10 9</td>
<td>17 16 15 14</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3 2 1 0</td>
<td>8 7 6 5</td>
<td>13 12 11 10</td>
<td>18 17 16 15</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4 3 2 1</td>
<td>9 8 7 6</td>
<td>14 13 12 11</td>
<td>19 18 17 16</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5 4 3 2</td>
<td>10 9 8 7</td>
<td>15 14 13 12</td>
<td>20 19 18 17</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>21 20 19 18</td>
<td>26 25 24 23</td>
<td>31 30 29 28</td>
<td>36 35 34 33</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>22 21 20 19</td>
<td>27 26 25 24</td>
<td>32 31 30 29</td>
<td>37 36 35 34</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>23 22 21 20</td>
<td>28 27 26 25</td>
<td>33 32 31 30</td>
<td>38 37 36 35</td>
</tr>
</tbody>
</table>

-Signal SEL is common for all multiplexers. '0' passes the input data, '1' passes the preceding register value (shift to right)

Figure 7.2. Configuration Overview required for the FIR filtering, the parallel approach.
7.1.2 Drawbacks

In the parallel approach the data path needs to be reconfigured so that, initially, the data can be fed into 4 SIMD processors from each SIMD’s LVM\(^1\) in the second cycle however, the data path should be reconfigured in a way that the data samples are now fed from one LVM to 4 SIMD processors for the next 4 cycle, leaving 3 LVMs unmounted. This is the procedure to be repeated until the filtering of all samples are done. The input registers in each SIMD processor (see Figure 2.2) need to be chained to support data shifts according to Figure 7.2. Moreover, a 64-bit path is needed to deliver the output of even filters to be added to output of odd filters through the delay element.

The situation is even more troublesome once it comes to fitting the FFA approach. For the architecture depicted in Figure 5.2, 3/4 SIMD processor is occupied performing \(H_0, H_1\) and \(H_0 + H_1\) filtering by \(\text{vmac}\) operation. This leaves 1/4 of the SIMD processor unused. Having the fact that \(\text{vmac}\) operation is used, there are certain number of cycles\(^2\) before the output is ready. During this time, the idle 1/4 can be used to fit the adders shown in 5.2 before the final output is ready. On the other hand, the second approach will require one SIMD processor to work on, so taking all SIMD processors will be beneficial. More parallelism can be introduced to filter calculation by breaking the input sample sequence into smaller sequences and performing the filtering in parallel. The only drawback will be that depending to the filter order same data is needed in different LVMs which is negligible\(^3\). Finally, another disadvantage of the parallel approach is that the architecture is well suited for 4\(^{th}\) and 8\(^{th}\) order filters. This limitation is imposed by the architecture however, can be avoided using the second approach using FFA algorithm.

7.1.3 Benchmark

An FIR filter of order 8 with input sample size of 4K in complex format was taken as the benchmark problem to compare the old FIR filtering calculation and the two new approaches. Comparison is made in terms of number of memory access and number of cycles to complete. Table 7.5 presents the comparison of the old approach and the new parallel approach. It can be observed the number of memory accesses is reduced but, the number of the execution cycles remained unchanged. Table 7.6 represents the comparison results of the FFA approach with the old approach. The FFA approach however, reduces both the number of memory access and execution cycles. The number of memory accesses in the the parallel approach is most satisfactory. On the other hand, the FFA approach provides more robustness for different filter orders taking the \(\text{vmac}\) operation.

---

\(^1\)In total 8 SIMD processors will be used 4 for the even part and 4 for the odd part. Since the procedure for both parts are identical for the sake of simplicity the description here is given for one part only.

\(^2\)As many as filter order

\(^3\)At most the same size at filter order
### Table 7.5. Comparison of the parallel approach and the old approach for 8-tap FIR filter with 4K input complex samples

<table>
<thead>
<tr>
<th>Approach</th>
<th>no. SIMD units</th>
<th>no. Reads</th>
<th>no. Writes</th>
<th>Total memory acc.</th>
<th>no. Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old approach</td>
<td>8</td>
<td>8192</td>
<td>1024</td>
<td>9216</td>
<td>1024</td>
</tr>
<tr>
<td>The Parallel Approach</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phase 1</td>
<td>8</td>
<td>1638.4</td>
<td>512</td>
<td>2150.4</td>
<td>512</td>
</tr>
<tr>
<td>Phase 2</td>
<td>8</td>
<td>1638.4</td>
<td>512</td>
<td>2150.4</td>
<td>512</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>3276.6</td>
<td>1024</td>
<td>4300.8</td>
<td>1024</td>
</tr>
</tbody>
</table>

In the first phase 2K output samples are produced. Since 4 output samples are produced every cycle hence 512 cycles is needed and the number of writes will be 512 as well. According to figure reffig:firhw every 5 cycles we have 8 memory reads, 4 direct reads and 4 reads from one LVM to be fed to 4 SIMD units. This results in 819 memory reads and since both odd and even samples are read, there will be 1638 reads in total.

### Table 7.6. Comparison of the FFA approach and the old approach for 8-tap FIR filter with 4K input complex samples

<table>
<thead>
<tr>
<th>Approach</th>
<th>FIR taps</th>
<th>no. SIMD units</th>
<th>no. Reads</th>
<th>no. Writes</th>
<th>Total memory acc.</th>
<th>no. Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old approach</td>
<td>8</td>
<td>8</td>
<td>8192</td>
<td>1024</td>
<td>9216</td>
<td>1024</td>
</tr>
<tr>
<td>The FFA Approach</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method 1</td>
<td>4</td>
<td>1</td>
<td>4096</td>
<td>1024</td>
<td>5120</td>
<td>4096</td>
</tr>
<tr>
<td>Method 2</td>
<td>4</td>
<td>8</td>
<td>4096</td>
<td>1024</td>
<td>5120</td>
<td>512</td>
</tr>
</tbody>
</table>

For method 2: 4K sample FIR filter generates 4K output result. Writing 4 output samples per cycles there will be 1024 writes into the memory. Since the filter is a 4th order filter hence 4 iteration of vmac operations should be done, Assuming the input data is distributed between 8 SIMD units, each SIMD unit is responsible for 512 input samples. For 512 input samples 128 reads should take place (512/4). 128 reads multiplied by 4 iteration results in 512 execution cycles.
7.1.4 Verification

Verification of the proposed FIR architecture is done theoretically. Because of relatively simple addressing and simple calculation structure here, the verification is limited to prove the data path’s capability to conduct the filtering operation as showed earlier.
Chapter 8

Implementation and Area Estimation Results

In this section, an overview of the necessary hardware for supporting the proposed modification in ePUMA platform and an area estimation of the excess hardware are presented.

8.1 Pipeline FFT Architecture on ePUMA

The Hardware. Changes in SIMD data path required for pipeline FFT architecture are only limited to the memory system (LVMs) and address generation unit. As mentioned earlier in Section 6.1.1, each SIMD unit should be able to read its own LVM, as it is able to. The SIMD unit also needs to write into the next SIMD unit’s LVM, which can be done by changing the NoC\(^1\) in such a way that provides this possibility or by adding extra interconnection. Multiplexers will be needed for applying input to the LVMs, which area overhead for them comparing to the rest of the chip seems to be negligible.

The Address Generation. Mentioned in Section 6.1.1, pipeline FFT architecture demands simultaneous read and write from and into the same memory (LVM) which suggests use of dual port memory. Due to implementation costs, that is not tolerable. Finally the problem was remedied taking two vector memories, one for read and one for write These two will change their role every clock cycle to act as a one dual port memory. On the other hand, since the data applied to each butterfly element\(^2\) should be with correct distance to each other, a specific addressing mode is required to achieve this. The address generation should be facilitated by modifying the current address generation unit or by adding a new FSM for example. For any given stage a certain amount of memory is needed (please see Figure 6.1), it is attempted to implement this in the LVM using the minimum

\(^1\)the Network on Chip

\(^2\)the SIMD unit is performing the butterfly operation
amount of memory required and possibly switching off the extra memory. This is done to achieve more power satisfactory results however, switching of the memory partially is not feasible. Assuming two LVMs available, one for read and one for write, in this fashion the data is stored in one memory in a specific places and read from the other memory from a specific place then they switch their role. Since we are taking the memory size smaller than the FFT sample size it should be handled with caution that the new data overwrites only the old data (data already read) once the memory is full. The size of memory required in every stage changes and so does the behavior of addressing for every stage. A description of memory sizes for every stage is given in Table 8.1. As a result the address generation unit should be able to generate the addresses with correct distance as well as handling how the data is overwritten in the LVMs in such a way that no unread data is overwritten.

### 8.2 New FIR Architecture on ePUMA

The hardware modification for supporting the new FIR architecture is not as simple as it was for pipeline FFT. As it can be seen in Figure 7.1, the SIMD data path itself should be modified, unlike pipeline FFT that only extra input to the LVMs were required. Considering the parallel approach, the data should be fed from one LVM to four SIMD units and also the shift mechanism both at the input and output of the SIMD units should be implemented. In this approach according to Figure 7.1, four SIMD units are different from the remaining four having extra input to perform the addition at the output of total filter (see Figure 5.1). On the other hand, feeding the output of 3/4 SIMD unit to its idling 1/4 for the FFA approach and the shift mechanism at the output should also be considered.

For area estimation, SIMD’s unit description code [7] was modified in a such way that it meets mentioned requirement. The area estimation is done using *Synopsys Design Compiler tool*, the results are presented in Table 8.2. As it can be seen in Table 8.2, two types of SIMD units have been considered, TypeI which supports FFA approach and TypeII which supports parallel approach and FFA approach\(^3\), showing 13.9% area increase in the worst case. Finally total chip estimation is also presented in Table 8.3 showing 0.75% in over all area increase [5].

---

\(^3\)Please note that to support both parallel approach and FFA approach 4 SIMD unit of TypeI and 4 SIMD unit of TypeII is required as it is mentioned in Section 7.1.1 Figure 7.1
Table 8.2. Area report of one single SIMD unit excluding LVMs and Registers.

<table>
<thead>
<tr>
<th></th>
<th>Standard SIMD</th>
<th>SIMD typeI</th>
<th>SIMD typeII</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu m^2 )</td>
<td>222663.478648</td>
<td>234755.037853</td>
<td>253642.997765</td>
</tr>
<tr>
<td>( mm^2 )</td>
<td>0.22266348</td>
<td>0.23475504</td>
<td>0.25364300</td>
</tr>
<tr>
<td>Area Increase(%)</td>
<td>0</td>
<td>5.430</td>
<td>13.913</td>
</tr>
</tbody>
</table>

Table 8.3. Area estimation of new chip.

<table>
<thead>
<tr>
<th>Total Chip Area (mm²)</th>
<th>Chip Area Excluding Memory System (mm²)</th>
<th>New Chip Area (4xTypeI+4xTypeII)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>21.219</td>
<td>23.172</td>
</tr>
<tr>
<td>Area Increase(%)</td>
<td></td>
<td>0.7491</td>
</tr>
</tbody>
</table>

Improvement: To further improve the area and power consumption, pipeline registers were added to the modified data path to see if it affects the critical path or result in a more area efficient design. However, this resulted in larger design with more increase in non combinational area which is not desirable, considering that this is the area with highest power consumption.
Chapter 9

Conclusion and Future Work

9.1 Conclusion

With the fact that hardware reconfigurability improves the system performance, in this thesis work it is attempted to study the advantages of equipping ePUMA DSP platform for reconfigurability. These advantages could include contribution to the performance and/or improved power consumption of the system. Comparing performance figures with and without the new feature indicates improvement in performance and also reduction of memory access which translates to better power consumption and in some cases improved performance. It is shown that it can reduce the memory access by removing unnecessary memory access which results in more efficient power consumption or else improving the throughput of a given algorithm.

9.2 Future Work

In this work the introduction of reconfigurability to ePUMA is studied. One benchmark taken here is the 4K point FFT for which the results are presented in Section 6 however, calculation of non-power of 4 FFTs are of interest which can be considered to be studied further. One key factor of DSP applications is the predictable behavior of the procedure to be taken, in this work only two general applications FFT and FIR filters have been studied, other application must be considered for studying in future. This makes it possible to examine the trade off between the improvement achieved and the design and implementation costs. The implementation and verification of the proposed architectures can also be the subject of future works for this thesis work.

As a future work also, the power estimation of the design with new features can be studied. In particular power estimation including the memory system can be very interesting, to realize how memory access can affect power consumption. Since the data path may be reconfigured both statically and dynamically, design
of the state machines for configuration as well as control FSM of the whole system are also interesting future work subjects. It is necessary also to study weather a centralized or decentralized control FSM should be implemented.
Bibliography


[22] E. Bidet, D. Castelain, C. Joanblanq, and P. Senn, “A fast single-chip imple-
mentation of 8192 complex point fft,” Solid-State Circuits, IEEE Journal of,

Linköping University, 2007.

Appendix A

Source Code for Pipeline FFT

A.1 Pipeline FFT

Source code for the 6 stage pipeline FFT is showed below. At each stage data is read from the previous stage and processed for the next stage activity of different stages is controlled by a condition checking the \( pc \) value which in practices can be controlled by an FSM.

```matlab
function [FFT1 FFT2]=fft_pl_4k_e_d(data ,data_pr)
fft_size=length(data);
addr=digitreorder(1:4096 ,4);
FFT1 =zeros(4096 ,1);
FFT2 =zeros(4096 ,1);
twl_s0=[0 256 512 768];
cnt=0;
full_1=0;
run_s4=0;
run_s3=0;
run_s2=0;
run_s1=0;
run_s0=0;
global pc;
if(fft_size~=4096)
disp('ERROR:FFT\_size should be 4K');
else
for pc=1:3076

% %%%%%%%%%%%%%%%%%%%%%%%%%%%% STAGE % % % %
if (run_s0)
    base=mod((pc−769−193−49−13−4−1)*4,4096);
twiddle=twl_s0*(mod(fix(base/4),4));
    if (pc>2052)
        full_1=1;
    else
        full_1=0;
    end
end
end
```

55
if (full_1==0)
[D1 D2 D3 D4]=mem_sl_read_d((pc−769−193−49−13−4−1)) ;
[S0 Ra S0_Ia S0_Rb S0_Ib S0_Rc S0_Ic S0_Rd S0_Id]=simd_bf
([D1 D2 D3 D4], twidle, fft_size) ;
FFT1(addr(base+1))=S0_Ra+1j∗S0_Ia;
FFT1(addr(base+2))=S0_Rb+1j∗S0_Ib;
FFT1(addr(base+3))=S0_Rc+1j∗S0_Ic;
FFT1(addr(base+4))=S0_Rd+1j∗S0_Id;
else
[D1 D2 D3 D4]=mem_sl_read_d((pc−769−193−49−13−4−1)) ;
[S0 Ra S0_Ia S0_Rb S0_Ib S0_Rc S0_Ic S0_Rd S0_Id]=simd_bf
([D1 D2 D3 D4], twidle, fft_size) ;
FFT2(addr(base+1))=S0_Ra+1j∗S0_Ia;
FFT2(addr(base+2))=S0_Rb+1j∗S0_Ib;
FFT2(addr(base+3))=S0_Rc+1j∗S0_Ic;
FFT2(addr(base+4))=S0_Rd+1j∗S0_Id;
end
%tw0((pc−769−193−49−13−4,:)=twl_s0*(mod(fix((base−1)/4),4))
end

%%%%%%%%%%%%%%%%%%%%%%%%%%% STAGE
if (run_s1) && pc <= 769+193+49+13+2048
base=mod(mod((pc−770−193−49−13−1),4)+fix((pc−770−193−49−13−4)/16),4096)+1;
twidle=(twl_s0+mod(base−1,4)+64)*mod(fix((base−1)/16),4);
if (pc >= 769+193+49+13)
run_s0=1;
else
run_s0=0;
end

end
%%%%%%%%%%%%%%%%%%%%%%%%%%% STAGE
if (run_s2) && pc <= 769+193+49+2048
base=mod(mod((pc−770−193−49−13),4)+fix((pc−770−193−49−13)/16),4096)+1;
twidle=(twl_s0+mod(base−1,16)+64)*mod(fix((base−1)/16),4);
if (pc >= 769+193+49+13)
run_s1=1;
else
run_s1=0;
end

end
%%%%%%%%%%%%%%%%%%%%%%%%%%% STAGE
if (run_s3) && pc <= 769+193+2048
base=mod(mod((pc−770−193−49−13),64)+fix((pc−770−193−49−13)/64),4096)+1;
twidle=(twl_s0+mod(base−1,64)+16)*mod(fix((base−1)/64),4);
if (pc >= 769+193+49)
run_s2=1;
else
run_s2=0;
end

end

Source Code for Pipeline FFT
A.2 Memory Functions

In this section memory IO functions are presented. Each function controls how the data should be read/write from/to the memory correctly as mentioned in the Section 6 in such a way that unread data is not overwritten.

```matlab
function [D1 D2 D3 D4]=mem_s4_read_d(adr1)

% base+0
[D1 D2 D3 D4]=mem_s4_read_d(pc−770−193); % base+0
mem_s3_write_d(pc−770−193,S3_Ra+1j*S3_Ia,S3_Rb+1j*S3_Ib,S3_Rc+1j*S3_Ic,S3_Rd+1j*S3_Id);
%tw33(cnt:cnt+3)=(twl_s0+mod(base−1,64)+4)*mod(fix((base−1)/256)+4),
%cnt=cnt+4;
end

%%%%%%%%%%%%%%%%%%%%%%%%%% STAGE 4%%%%%%%%%%%%%%%%%%%%%%%
if (run_s4) && pc <=769+2048
base=mod(mod((pc−770),256)+fix((pc−770)/256)*1024),4096)+1;
twiddle=(twl_s0+mod(base−1,256))*mod(fix((base−1)/1024)+4),
if (pc>=769+193) run_s3=1;
else
run_s3=0;
end

%%%%%%%%%%%%%%%%%%%%%%%%%% STAGE 5%%%%%%%%%%%%%%%%%%%%%%%
if (pc>=769) run_s4=1;
else
run_s4=0;
end
base=mod(pc−1,1024)+1;
if (pc<=1024)
[S5_Ra S5_Ia S5_Rb S5_Ib S5_Rc S5_Ic S5_Rd S5_Id]=simd_bf([data(base+0) data(base+1024) data(base+2048) data(base+3072)],[0 0 0 0],fft_size);
mem_s5_write_d(pc−1,S5_Ra+1j*S5_Ia,S5_Rb+1j*S5_Ib,S5_Rc+1j*S5_Ic,S5_Rd+1j*S5_Id);
else% set 2
[S5_Ra S5_Ia S5_Rb S5_Ib S5_Rc S5_Ic S5_Rd S5_Id]=simd_bf([data_pr(base+0) data_pr(base+1024) data_pr(base+2048) data_pr(base+3072)],[0 0 0 0],fft_size);
mem_s5_write_d(pc−1,S5_Ra+1j*S5_Ia,S5_Rb+1j*S5_Ib,S5_Rc+1j*S5_Ic,S5_Rd+1j*S5_Id);
end
end

stop=1;
end

../ML_IMP/fft_pl_4k_e_d.m
```

A.2 Memory Functions

In this section memory IO functions are presented. Each function controls how the data should be read/write from/to the memory correctly as mentioned in the Section 6 in such a way that unread data is not overwritten.

```matlab
function [D1 D2 D3 D4]=mem_s1_read_d(adr1)
```
global vm_e_s1
global vm_o_s1
global mem_s1_RD

global pc;
global s1_mem_read_eo;
mem_s1_RD=mem_s1_RD+1;
if (mod(fix(adr1/4),2)==0)%%even
    t=vm_e_s1(:,mod(adr1,4)+1);
    s1_mem_read_eo(pc,:)==[1 0];
else
    t=vm_o_s1(:,mod(adr1,4)+1);
    s1_mem_read_eo(pc,:)==[0 1];
end
D1=t(1);
D2=t(2);
D3=t(3);
D4=t(4);
end

../ML_IMP/mem_s1_read_d.m

function mem_s1_write_d(adr1,D1,D2,D3,D4)
global vm_e_s1
global vm_o_s1
global mem_s1_WR

global pc;
global s1_mem_write_eo;
mem_s1_WR=mem_s1_WR+1;
if (mod(fix(adr1/4),2)==0)%%even
    vm_e_s1(mod(adr1,4)+1,:)==[D1 D2 D3 D4];
    s1_mem_write_eo(pc,:)==[1 0];
else
    vm_o_s1(mod(adr1,4)+1,:)==[D1 D2 D3 D4];
    s1_mem_write_eo(pc,:)==[0 1];
end
end

../ML_IMP/mem_s1_write_d.m

function [D1 D2 D3 D4]=mem_s2_read_d(adr1)
%Read data in 0 4 8 12
%Memory size : 2x8x4

global vm_e_s2

global vm_o_s2

global mem_s2_RD
mem_s2_RD=mem_s2_RD+1;

global pc;
global s2_mem_read_eo;
if (mod(fix(adr1/2)==0)%%even
    if (mod(fix(adr1/16),2)==0)
        D1=vm_e_s2((mod(adr1,4)/2)+1,fix(mod(adr1,16)/4)+1);
        D2=vm_e_s2((mod(adr1,4)/2)+3,fix(mod(adr1,16)/4)+1);
        D3=vm_e_s2((mod(adr1,4)/2)+5,fix(mod(adr1,16)/4)+1);
        D4=vm_e_s2((mod(adr1,4)/2)+7,fix(mod(adr1,16)/4)+1);
    else
        D1=vm_e_s2((mod(adr1,16)/2)+1,1);
        D2=vm_e_s2((mod(adr1,16)/2)+1,2);
        D3=vm_e_s2((mod(adr1,16)/2)+1,3);
        D4=vm_e_s2((mod(adr1,16)/2)+1,4);
    end
    s2_mem_read_eo(pc,:)==[1 0];
else
    if (mod(fix(adr1/16),2)==0)
A.2 Memory Functions

D1=vm_o_s2(fix(mod(adr1,4)/2)+1,fix(mod(adr1,16)/4)+1);
D2=vm_o_s2(fix(mod(adr1,4)/2)+3,fix(mod(adr1,16)/4)+1);
D3=vm_o_s2(fix(mod(adr1,4)/2)+5,fix(mod(adr1,16)/4)+1);
D4=vm_o_s2(fix(mod(adr1,4)/2)+7,fix(mod(adr1,16)/4)+1);
else
D1=vm_o_s2(fix(mod(adr1,16)/2)+1,1);
D2=vm_o_s2(fix(mod(adr1,16)/2)+1,2);
D3=vm_o_s2(fix(mod(adr1,16)/2)+1,3);
D4=vm_o_s2(fix(mod(adr1,16)/2)+1,4);
end
s2_mem_read_eo(pc,:)=[0 1];
end
end

function mem_s2_write_d(adr1,D1,D2,D3,D4)
%Write data in 0 16 32 48
%Memory size : 2x8x4
global vm_e_s2
global vm_o_s2
global mem_s2_WR
mem_s2_WR=mem_s2_WR+1;

global pc;
global s2_mem_write_eo;
if(mod(adr1,2)==0)%even
if(mod(fix(adr1/16),2)==0)
vm_e_s2((mod(adr1,16)/2)+1,:)=D1 D2 D3 D4;
else
vm_e_s2((mod(adr1,4)/2)+1,fix(mod(adr1,16)/4)+1)=D1;
vm_e_s2((mod(adr1,4)/2)+3,fix(mod(adr1,16)/4)+1)=D2;
vm_e_s2((mod(adr1,4)/2)+5,fix(mod(adr1,16)/4)+1)=D3;
vm_e_s2((mod(adr1,4)/2)+7,fix(mod(adr1,16)/4)+1)=D4;
end
s2_mem_write_eo(pc,:)=[1 0];
else
if(mod(fix(adr1/16),2)==0)
v conced
vm_o_s2(fix(mod(adr1,16)/2)+1,:)=[D1 D2 D3 D4];
else
vm_o_s2(fix(mod(adr1,4)/2)+1,fix(mod(adr1,16)/4)+1)=D1;
v o_s2(fix(mod(adr1,4)/2)+3,fix(mod(adr1,16)/4)+1)=D2;
v o_s2(fix(mod(adr1,4)/2)+5,fix(mod(adr1,16)/4)+1)=D3;
v o_s2(fix(mod(adr1,4)/2)+7,fix(mod(adr1,16)/4)+1)=D4;
end
s2_mem_write_eo(pc,:)=[0 1];
end
end

function [D1 D2 D3 D4]=mem_s3_read_d(adr1)
%Read data in 0 16 32 48
%Memory size : 2x32x4
global vm_e_s3
global vm_o_s3
global mem_s3_RD
mem_s3_RD=mem_s3_RD+1;

global pc;
global s3_mem_read_eo;
if(mod(adr1,2)==0)%even
if(mod(fix(adr1/64),2)==0)
D1=vm_e_s3((mod(adr1,16)/2)+1,fix(mod(adr1,64)/16)+1);
D2=vm_e_s3((mod(adr1,16)/2)+9,fix(mod(adr1,64)/16)+1);
D3=vm_e_s3((mod(adr1,16)/2)+17,fix(mod(adr1,64)/16)+1);
else
D1=vm_o_s3((mod(adr1,16)/2)+1,1);
D2=vm_o_s3((mod(adr1,16)/2)+1,2);
D3=vm_o_s3((mod(adr1,16)/2)+1,3);
D4=vm_o_s3((mod(adr1,16)/2)+1,4);
end
s2_mem_read_eo(pc,:)=0 1;
end
end

function [D1 D2 D3 D4]=mem_s2_read_d(adr1)
%Write data in 0 16 32 48
%Memory size : 2x8x4
global vm_e_s2
global vm_o_s2

../ML_IMP/mem_s2_read_d.m

../ML_IMP/mem_s2_write_d.m

../ML_IMP/mem_s3_read_d.m

../ML_IMP/mem_s3_write_d.m
D4=vm_e_s3((mod(adr1,16)/2)+25,fix(mod(adr1,64)/16)+1);
else
  D1=vm_e_s3((mod(adr1,64)/2)+1,1);
  D2=vm_e_s3((mod(adr1,64)/2)+1,2);
  D3=vm_e_s3((mod(adr1,64)/2)+1,3);
  D4=vm_e_s3((mod(adr1,64)/2)+1,4);
end
s3_mem_read_eo(pc,:)=[1 0];
else
  if (mod(fix(adr1/64),2)==0)
    D1=vm_o_s3(fix(mod(adr1,16)/2)+1,fix(mod(adr1,64)/16)+1);
    D2=vm_o_s3(fix(mod(adr1,16)/2)+9,fix(mod(adr1,64)/16)+1);
    D3=vm_o_s3(fix(mod(adr1,16)/2)+17,fix(mod(adr1,64)/16)+1);
    D4=vm_o_s3(fix(mod(adr1,16)/2)+25,fix(mod(adr1,64)/16)+1);
  else
    D1=vm_o_s3(fix(mod(adr1,16)/2)+1,1);
    D2=vm_o_s3(fix(mod(adr1,16)/2)+1,2);
    D3=vm_o_s3(fix(mod(adr1,16)/2)+1,3);
    D4=vm_o_s3(fix(mod(adr1,16)/2)+1,4);
  end
end
s3_mem_read_eo(pc,:)=[0 1];
A.2 Memory Functions

```matlab
% A.2 Memory Functions

global mem_s4_RD
mem_s4_RD=mem_s4_RD+1;

global pc;
global s4_mem_read_eo;
if (mod(adr1,2)==0) even
    if (mod(fix(adr1/256),2)==0)
        D1=vm_e_s4((mod(adr1,128)/2)+1,fix(mod(adr1,256)/64)+1);
        D2=vm_e_s4((mod(adr1,128)/2)+33,fix(mod(adr1,256)/64)+1);
        D3=vm_e_s4((mod(adr1,128)/2)+65,fix(mod(adr1,256)/64)+1);
        D4=vm_e_s4((mod(adr1,128)/2)+97,fix(mod(adr1,256)/64)+1);
    else
        D1=vm_e_s4((mod(adr1,256)/2)+1,1);
        D2=vm_e_s4((mod(adr1,256)/2)+1,2);
        D3=vm_e_s4((mod(adr1,256)/2)+1,3);
        D4=vm_e_s4((mod(adr1,256)/2)+1,4);
    end
end
s4_mem_read_eo(pc,:) = [1 0];
else
    if (mod(fix(adr1/256),2)==0)
        D1=vm_o_s4(fix(mod(adr1,128)/2)+1,fix(mod(adr1,256)/64)+1);
        D2=vm_o_s4(fix(mod(adr1,128)/2)+33,fix(mod(adr1,256)/64)+1);
        D3=vm_o_s4(fix(mod(adr1,128)/2)+65,fix(mod(adr1,256)/64)+1);
        D4=vm_o_s4(fix(mod(adr1,128)/2)+97,fix(mod(adr1,256)/64)+1);
    else
        D1=vm_o_s4(fix(mod(adr1,256)/2)+1,1);
        D2=vm_o_s4(fix(mod(adr1,256)/2)+1,2);
        D3=vm_o_s4(fix(mod(adr1,256)/2)+1,3);
        D4=vm_o_s4(fix(mod(adr1,256)/2)+1,4);
    end
end
s4_mem_read_eo(pc,:) = [0 1];
end


% function mem_s4_write_d(adr1,D1,D2,D3,D4)
% Write data in 0 256 512 768
% Memory size : 2x128x4

global vm_e_s4
global vm_o_s4
global mem_s4_WR
mem_s4_WR=mem_s4_WR+1;

global pc;
global s4_mem_write_eo;
if (mod(adr1,2)==0) even
    if (mod(fix(adr1/256),2)==0)
        vm_e_s4((mod(adr1,128)/2)+1,:)=[D1 D2 D3 D4];
    else
        vm_e_s4((mod(adr1,128)/2)+1,:)=[D1 D2 D3 D4];
        vm_e_s4((mod(adr1,128)/2)+33,:)=[D1 D2 D3 D4];
        vm_e_s4((mod(adr1,128)/2)+65,:)=[D1 D2 D3 D4];
        vm_e_s4((mod(adr1,128)/2)+97,:)=[D1 D2 D3 D4];
    end
end
s4_mem_write_eo(pc,:) = [1 0];
else
    if (mod(fix(adr1/256),2)==0)
        vm_o_s4(fix(mod(adr1,128)/2)+1,:)=[D1 D2 D3 D4];
    else
        vm_o_s4(fix(mod(adr1,128)/2)+1,:)=[D1 D2 D3 D4];
        vm_o_s4(fix(mod(adr1,128)/2)+33,:)=[D1 D2 D3 D4];
        vm_o_s4(fix(mod(adr1,128)/2)+65,:)=[D1 D2 D3 D4];
        vm_o_s4(fix(mod(adr1,128)/2)+97,:)=[D1 D2 D3 D4];
    end
end
s4_mem_write_eo(pc,:) = [0 1];
end
```

```matlab
../ML_IMP/mem_s4_read_d.m
```
function D1 D2 D3 D4 = mem_s5_read_d(adr1)
% Read data in 0 256 512 768
% Memory size: 2x512x4
global vm_e_s5
global vm_o_s5
global mem_s5_RD
mem_s5_RD = mem_s5_RD + 1;

global pc;
global s5_mem_read_eo;
if (mod(adr1, 2) == 0) % even
    if (mod(fix(adr1/1024), 2) == 0)
        D1 = vm_e_s5((mod(adr1, 1024)/2)+1, fix(mod(adr1, 1024)/256)+1);
        D2 = vm_e_s5((mod(adr1, 1024)/2)+129, fix(mod(adr1, 1024)/256)+1);
        D3 = vm_e_s5((mod(adr1, 1024)/2)+257, fix(mod(adr1, 1024)/256)+1);
        D4 = vm_e_s5((mod(adr1, 1024)/2)+385, fix(mod(adr1, 1024)/256)+1);
    else
        D1 = vm_e_s5((mod(adr1, 1024)/2)+1, 1);
        D2 = vm_e_s5((mod(adr1, 1024)/2)+129, 1);
        D3 = vm_e_s5((mod(adr1, 1024)/2)+257, 1);
        D4 = vm_e_s5((mod(adr1, 1024)/2)+385, 1);
    end

    s5_mem_read_eo(pc, :) = [1 0];
else
    if (mod(fix(adr1/1024), 2) == 0)
        D1 = vm_o_s5(fix(mod(adr1, 1024)/2)+1, fix(mod(adr1, 1024)/256)+1);
        D2 = vm_o_s5(fix(mod(adr1, 1024)/2)+129, fix(mod(adr1, 1024)/256)+1);
        D3 = vm_o_s5(fix(mod(adr1, 1024)/2)+257, fix(mod(adr1, 1024)/256)+1);
        D4 = vm_o_s5(fix(mod(adr1, 1024)/2)+385, fix(mod(adr1, 1024)/256)+1);
    else
        D1 = vm_o_s5(fix(mod(adr1, 1024)/2)+1, 1);
        D2 = vm_o_s5(fix(mod(adr1, 1024)/2)+129, 1);
        D3 = vm_o_s5(fix(mod(adr1, 1024)/2)+257, 1);
        D4 = vm_o_s5(fix(mod(adr1, 1024)/2)+385, 1);
    end

    s5_mem_read_eo(pc, :) = [0 1];
end
end

function mem_s5_write_d(adr1, D1, D2, D3, D4)
% Write data in 0 1024 2048 3072
% Memory size: 2x512x4
global vm_e_s5
global vm_o_s5
global mem_s5_WR
mem_s5_WR = mem_s5_WR + 1;

global pc;
global s5_mem_write_eo;
if (mod(adr1, 2) == 0) % even
    if (mod(fix(adr1/1024), 2) == 0)
        vm_e_s5((mod(adr1, 1024)/2)+1, :) = [D1 D2 D3 D4];
    else
        vm_e_s5((mod(adr1, 1024)/2)+1, :)=D1;
        vm_e_s5((mod(adr1, 1024)/2)+129, :)=D2;
        vm_e_s5((mod(adr1, 1024)/2)+257, :)=D3;
        vm_e_s5((mod(adr1, 1024)/2)+385, :)=D4;
    end

    s5_mem_write_eo(pc, :) = [1 0];
else

end
A.3 Butterfly Element in SIMD Unit

The radix4 butterfly operation performed in SIMD unit is represented in here.

```matlab
function [Ra, Ia, Rb, Ib, Rc, Ic, Rd, Id] = simd_bf(dat, coef, fft_size)
% take the input and perform radix4 butterfly and produces for outputs in
% complex form
x=[1 1 1 1
   1 -1i -1 1i
   1 -1 1 -1i
   1 1i -1 -1i];
y=[0 0 0 0];
z=x*y;
Ra = real(z(1));
Ia = imag(z(1));
Rb = real(z(2));
Ib = imag(z(2));
Rc = real(z(3));
Ic = imag(z(3));
Rd = real(z(4));
Id = imag(z(4));
end
../ML_IMP/simd_bf.m
```

A.4 Twiddle Factor Calculation

The Twiddle factors are calculated with respect to index of the data sample being processed.

```matlab
function [tw]=twiddle(index, fft_size, stage)
cons_coef_s2=[0 4 8 12];
cons_coef_s1=0:1:15;
total_stage=log2(fft_size)/2;%log4
```
step = power(4, total_stage - stage);
if (stage == 1)
    tw = mod(fix(index/16), 4) * cons_coef_s1(fix(mod(index, 16)/4)*4+1:fix(mod(index, 16)/4)*4+4);
else
    tw = mod(fix(index/4), 4) * cons_coef_s2;
end

../ML_IMP/twidle.m

A.5 Testbench

The testbench applies the randomly generated data samples to the pipeline FFT and compares the result applying the same data to matlabb built in FFT and presents the results. Testbench also checks if any memory conflict has happened by displaying memory access graphs.

```matlab
%clear all;
close all;

%%memory system
global mem_s2
global mem_s3
global mem_s4
global mem_s5
mem_s2 = zeros(4096, 1);
mem_s3 = zeros(4096, 1);
mem_s4 = zeros(4096, 1);
mem_s5 = zeros(4096, 1);

global vm_e_s1
global vm_o_s1
global vm_e_s2
global vm_o_s2
global vm_e_s3
global vm_o_s3
global vm_e_s4
global vm_o_s4
global vm_e_s5
global vm_o_s5

vm_e_s1 = zeros(4, 4);
vm_o_s1 = zeros(4, 4);
%%mem size=2*4*4=32 cpx

vm_e_s2 = zeros(8, 4);
vm_o_s2 = zeros(8, 4);
%%mem size=2*8*4=64 cpx

vm_e_s3 = zeros(32, 4);
vm_o_s3 = zeros(32, 4);
%%mem size=2*32*4=256 cpx

vm_e_s4 = zeros(128, 4);
vm_o_s4 = zeros(128, 4);
%%mem size=2*128*4=1024 cpx

vm_e_s5 = zeros(512, 4);
vm_o_s5 = zeros(512, 4);
%%mem size=2*512*4=4096 cpx
```
memory access

global mem_s1_RD
global mem_s2_RD
global mem_s3_RD
global mem_s4_RD
global mem_s5_RD
global mem_s1_WR
global mem_s2_WR
global mem_s3_WR
global mem_s4_WR
global mem_s5_WR

mem_s1_RD=0;
mem_s2_RD=0;
mem_s3_RD=0;
mem_s4_RD=0;
mem_s5_RD=0;
mem_s1_WR=0;
mem_s2_WR=0;
mem_s3_WR=0;
mem_s4_WR=0;
mem_s5_WR=0;

global pc;
global s1_mem_write_eo;
global s1_mem_read_eo;
global s2_mem_write_eo;
global s2_mem_read_eo;
global s3_mem_write_eo;
global s3_mem_read_eo;
global s4_mem_write_eo;
global s4_mem_read_eo;
global s5_mem_write_eo;
global s5_mem_read_eo;

generation of random complex samples
data_1=rand(1,4096)*10;
data_2=rand(1,4096)*10;
data_1=exp(1i*data_1);
data_2=exp(1i*data_2);

FFT calculation
[my_fft_1 my_fft_2]=fft_pl_4k_e_d(data_1 ,data_2) ;

Reference FFT calculation
re_fft_1=fft(data_1) ;
re_fft_2=fft(data_2) ;

Difference Calculation
dif_1=my_fft_1−reshape(re_fft_1 ,4096 ,1) ;
dif_2=my_fft_2−reshape(re_fft_2 ,4096 ,1) ;

figure();
subplot(2,2,1);
plot(1:4096,real(my_fft_1),'b−−',1:4096,real(re_fft_1),'rs−−')
title('Real part 1')

subplot(2,2,2);
plot(1:4096,imag(my_fft_1),'b−−',1:4096,imag(re_fft_1),'rs−−')
title('Image part 1')

subplot(2,2,3);
plot(1:4096,real(dif_1))
title('real difference')

subplot(2,2,4);
plot(1:4096,imag(dif_1))
title('image difference')
```matlab
figure();
subplot(2,2,1);
plot(1:4096,real(my_fft_2), 'b+-', 1:4096, real(re_fft_2), 'rs-');
title('Real part 2')

subplot(2,2,2);
plot(1:4096,imag(my_fft_2), 'b+-', 1:4096, imag(re_fft_2), 'rs-');
title('Image part 2')

subplot(2,2,3);
plot(1:4096,real(dif_2));
title('real difference')

subplot(2,2,4);
plot(1:4096,imag(dif_2));
title('image difference')

collission
figure();
subplot(10,1,1);
plot(1:3076,s1_mem_write_eo(:,1), 'b+-', 1:3076, s1_mem_read_eo(:,1), 'rs-');
title('s1 Memory even bank')

subplot(10,1,2);
plot(1:3076,s1_mem_write_eo(:,2), 'b+-', 1:3076, s1_mem_read_eo(:,2), 'rs-');
title('s1 Memory odd bank')

subplot(10,1,3);
plot(1:3076,s2_mem_write_eo(:,1), 'b+-', 1:3076, s2_mem_read_eo(:,1), 'rs-');
title('s2 Memory even bank')

subplot(10,1,4);
plot(1:3076,s2_mem_write_eo(:,2), 'b+-', 1:3076, s2_mem_read_eo(:,2), 'rs-');
title('s2 Memory odd bank')

subplot(10,1,5);
plot(1:3076,s3_mem_write_eo(:,1), 'b+-', 1:3076, s3_mem_read_eo(:,1), 'rs-');
title('s3 Memory even bank')

subplot(10,1,6);
plot(1:3076,s3_mem_write_eo(:,2), 'b+-', 1:3076, s3_mem_read_eo(:,2), 'rs-');
title('s3 Memory odd bank')

subplot(10,1,7);
plot(1:3076,s4_mem_write_eo(:,1), 'b+-', 1:3076, s4_mem_read_eo(:,1), 'rs-');
title('s4 Memory even bank')

subplot(10,1,8);
plot(1:3076,s4_mem_write_eo(:,2), 'b+-', 1:3076, s4_mem_read_eo(:,2), 'rs-');
title('s4 Memory odd bank')

subplot(10,1,9);
plot(1:3076,s5_mem_write_eo(:,1), 'b+-', 1:3076, s5_mem_read_eo(:,1), 'rs-');
title('s5 Memory even bank')

subplot(10,1,10);
plot(1:3076,s5_mem_write_eo(:,2), 'b+-', 1:3076, s5_mem_read_eo(:,2), 'rs-');
title('s5 Memory odd bank')

sprintf('No. read No. write total
\%d \%d \%d \%d \%d \%d \%d \%d'....)
```
A.6 Cycle Estimation

Cycle estimation according to the formula done in Matlab. As it can be seen the number of cycles for pipeline approach is linear with a coefficient equal 1024 and constant value of 1028. This is due to the fact that each completion of FFT takes 1024 cycles and it takes 1028 cycles for the pipeline to be fully loaded.

```matlab
clear all;
close all;
a=1:100;
cyc=fix(((a-1)/6)+1)*7670;
cyc=reshape(cyc,length(a),1);
cyc_pip=a*1024+1028;
cyc_pip=reshape(cyc_pip,length(a),1);
plot(a,cyc,'-',a,cyc_pip,'--')
ylabel('Number of Cycles')
xlabel('Number of FFT Sets')
legend('non-pipelined','pipelined')
ml=max(cyc)/max(cyc_pip)

figure()
cyc_8=fix(((a-1)/8)+1)*7670;
cyc_8=reshape(cyc_8,length(a),1);
cyc_2=fix(((a-1)/2)+1)*7670;
cyc_2=reshape(cyc_2,length(a),1);
cyc_pip=a*1024+1028;
cyc_pip=reshape(cyc_pip,length(a),1);
cyc_tot=cyc_2.*cyc_pip./(cyc_2+cyc_pip);
plot(a,cyc_8,'-',a,cyc_tot,'--')
ylabel('Number of Cycles Using 8 SIMD units')
xlabel('Number of FFT Sets')
legend('non-pipelined','pipelined')
m2=max(cyc_8)/max(cyc_tot)

figure()
plot(a,cyc_8,'-',a,cyc_tot,'--',a,cyc,'-',a,cyc_pip,'k')
ylabel('Number of Cycles Using 8 SIMD units vs. 6 SIMD units')
xlabel('Number of FFT Sets')
legend('non-pipelined 8 SIMD','pipelined 6 SIMD + 2 non-pipelined','non-pipelined 6 SIMD','pipelined 6 SIMD')

makfig.m
```