Computation Enhancement using Reconfigurable Computing

By

Ayman Hassan Salim Gumaa

17878

Dissertation submitted in partial fulfillment of

the requirements for the

Bachelor of Engineering (Honors)

(Electrical)

FYP II May 2015

Universiti Teknologi PETRONAS

Bandar Seri Iskandar

31750 Tronoh

Perak Darul Ridzuan

ABSTRACT

In light of the industry's constant need for better computer performance, this project aims to choose and evaluate an approach for facing this issue. The targeted category of computers is single board computers (e.g. Raspberry Pi). The approach utilized for enhancing performance is the use of reconfigurable computing as to execute computationally expensive calculations on a runtime custom-tailored hardware. The objective of this project is the test of the potential this approach has for increasing computers performance through comparing a software implementation of an algorithm with an FPGA assisted implementation of the same algorithm.

The platforms chosen for this project are the Rapsberry Pi and the Parallella P1602 board with its Zynq SoC for the software implementation and the FPGA assisted implementation in that order. The chosen algorithm is Fourier Fast Transform due to its part in many DSP applications and its suitability for the project objective. While the software solution worked successfully resulting in an asymptotic cost of O(N log N); the reconfigurable computing solution couldn't be completed due to time constraints and lack of experience of the student. Future work should complete the experiment and add a multicore implementation of the same algorithm to add yet another class to the comparison.

ACKNOWLEDGEMENTS

My sincerest thanks to my Supervisor Dr. Zuki Mohd who accepted my selfish project proposal and supported the project throughout the FYP period and to Dr. Shawn Tan who gave a lot of his time and guided this project in a more feasible direction. Without your efforts this project would be nothing.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1
INTRODUCTION

1.1.BACKGROUND

The impact computers have on humans life and technology is arguably unrivaled by any other invention in modern history. Their applications encompass everything from nuclear reactors and space stations to smart watches and glasses. As the applications evolve, they become greedier in terms of required hardware performance, that drives the computer industry to innovate and produce superior computers which in turn inspires greedier software and the race continues. One of the most relatable examples for the public would be video games; the quality of the games improves rapidly, forcing the gamers to upgrade their hardware every few years.

However, the industry have started running into hardware limitations such as transistors' sizes and the power wall; these limitations point that advancement of the hardware will rely on improving computer architectures and organization so that better designs emerge using the same hardware resources. Performance itself though depends on many parameters such as computation speed, memory access seed, and I/O speed, this project focuses on improving the performance in terms of computation speed. A plethora of methods to improve the performance were invented and implemented in the last few decades; the literature review chapter will provide a brief introduction to some of them and highlight the idea which the author think has the greatest potential in enhancing computer performance.

Having chosen a worthy approach, this project attempts to test the appeal of this approach. Due to time limitations However, this project will only attempt to proof that the chosen concept for performance improvement (reconfigurable computing; which, simply put, is adding an FPGA to the microprocessor to increase its computational power) has the potential to lead to the development of higher performing computers. Development of a computer architecture based on the recommendations of this project is left for future work in the area.

1.2.PROBLEM STATEMENT

The hardware limitations are hindering the development of processors computation capabilities. This project suggests re-configurable computing as a technologically-feasible possible approach to enhance the computations speed of the processors and shows supporting results.

1.3.OBJECTIVE AND SCOPE OF STUDY

The objectives of this project are:

1- Evaluating the performance of a typical computer.
2- Evaluating the performance using re-configurable computing.

The project time is four months; therefore, the scope of the project will be limited and will not attempt to actually implement a standalone microarchitecture. The proof of concept will be achieved through comparing the performance of pure software implementation of algorithms compared to FPGA assisted implementation.

It is important to stress that this project is not attempting to provide a solution that fulfills industry qualifications; it is simply attempting to find a promising approach and provide a simple proof of concept to show that further research in this direction could potentially help the industry. This conservative aim is due to the project being a one man job in a very limited time frame with no prior experience.

CHAPTER 2
LITERATURE REVIEW

Since the aim of the project is improving computer performance; this chapter starts with defining performance and its different measures. After that a general overviews provided for several concepts used to maximize the computer performance and they are evaluated to pick an approach whose full potential has not been reached and which class of computers it might suit. Following these sections is an overview of the Fast Fourier Transform (FFT); the algorithm used in this project to compare the performance of re-configurable computing to the performance of a traditional computer.

2.1. PERFORMANCE

As stated in the previous chapters, this project aims to improve computer performance. Therefore the first step would be defining performance. The problem with defining performance is that it depends on the application as Hennessy and Patterson mentioned in [**2**]. For example when running a sequential program, normally the computer that finished the program faster is considered the highest performing computer. On the other hand, in a datacenter the server that finishes the largest number of tasks is labeled the highest performing server. Those two cases introduce two popular performance measures that are usually called Execution Time or Response Time, and Throughput or Bandwidth. For a general purpose computer it will run tasks where Throughput is more important than execution time and vice versa, therefore selecting the right measure of performance is necessary for evaluating and enhancing the computer's performance. Personal computers usually emphasize response time over throughput.

When evaluating execution time, a good approach would be breaking it into several factors as shown in [**2**]. Suitable factors are: the number of instructions in the program, the average number of clock cycles required for an instruction (CPI, i.e. clock cycles per instruction), and the number of clock cycles per second. Multiplying these three factors yields the CPU execution time of the program. It should be noted

that, due to the operating system division of resources, this time could differ from the actual time it will take for the program to be executed; nevertheless it still provides a great insight into the performance [**2**].

$$\frac{instruction\ count}{Program} * \frac{clock\ cycles}{instruction} * \frac{seconds}{clock\ cycle} = \frac{Seconds}{Program}$$

Until recent years, computers' performances would raise continuously just by raising the clock rate of the processor. This made it easier for the programmers and the designers because using the same architecture the performance will improve rapidly even if they don't modify anything else. The results were great and the performance doubled every 18 months thanks to rapid improvements in transistors speed and numbers [**3**]. However, raising the clock rate means raising the dissipated power which means the clock rate is limited by the amount of heat that could be dissipated successfully. The researchers managed to keep increasing the clocking speed by lowering the chip voltage the square of which is inversely proportional to the heat [**2**]. In the last decade, this approach reached a dead-end; according to the international roadmap for semiconductors in 2005, by 2010 a rate of over 10 GHz should have been achieved [**3**]. Failure in complying with these projections clearly shows that this approach reached its limit and raising the clock rates anymore is not feasible. Other approaches, which have been proposed and developed throughout the age of computer industry, to maximize performance are presented in the next sections. The key concept that is currently utilized is parallelization with its different levels. Due to the significance of this approach, the next section has been devoted to exploring few of its concepts.

2.2. PARALLELISM

The first model that comes to mind when imagining a computer running a program is a sequence of instructions stored in the memory; the processor fetches an instruction, executes it, and then fetches the next instruction. Actual models of sequential computing are more sophisticated and efficient than that, thanks to concepts such as pipelining, prefetching, multiple instruction issue, out of order execution, and other similar techniques [**3**]. Such techniques are often hidden from the programmer and are handled by the architecture; they can be referred to as Instruction Level Parallelism (ILP) [**2**]. Other types of parallelism were developed also to maximize

performance such as Data Level Parallelism (e.g. the use of vectors), Thread Level Parallelism, and Multiprocessors.

2.2.1 Instruction Level Parallelism

This type of parallelism aims to exploit parallelism on the level of instructions; therefore, its significance to the design of a novel instruction set is clear. There are two approaches to maximizing the gains of ILP: utilizing the stages required for executing an instruction (fetching it, decoding it, using ALU, memory access… etc.) and insuring that at any given moment there is an instruction in each step of the process, this is called pipelining; the other approach is duplicating the components of the processor to run multiple instructions in the same pipeline level at the same time, which are called multiple issue processors [**2**].

The concept of pipelining is utilized in our everyday activities; for example, when a student in a dorm wakes up in the morning his process could be: taking a shower, brushing his teeth, preparing breakfast, ironing his clothes, and finally going to class. Now if this student has a roommate, this roommate can use the exact steps but, assuming the existence of only one shower and one heater for breakfast, he cannot do them at the same time with his roommate. However, instead of waiting for him to finish the whole process before starting; he will start it one stage later, that is, when the roommate exits the shower he will enter, then when he exits the shower and brush his teeth his roommate will be done with the breakfast and he can use the heater freely. This example shows that in case of limited resources, there is no need for the second student to wait for the first to be fully done; instead, he can begin executing as soon as the other student advances in the pipeline. Assuming that an instruction progresses through several stages from fetching until the result is written in a register, the longer the pipeline is, the more instructions that will be in it simultaneously which is referred to as increasing the pipeline depth [**2**].

As can be noticed from the previous explanation, pipelining does not speed up the execution of individual instructions; instead, it eliminates any unnecessary gap between them so that the total number of instructions will be executed faster. In technical terms, pipelining increases the instructions throughput. There are few hazards that may arise from an improper pipelining. They are generally divided to Structural hazards (e.g. an instruction writing to the memory while another one is

reading from the same memory), Data hazards (e.g. instruction needs a result that hasn't been obtained yet), and Control hazards (e.g. which instruction to load in the pipeline just after fetching a conditional instruction). A good overview for those hazards and methods to counter them was provided in [**2**]. For this project, pipelining will be implemented.

Capitalizing on our previous analogy, if we want to utilize this concept we will build a second shower room so that no time is wasted waiting for it to be empty. Multiple Issue processors can be divided into dynamic multiple issue processors and static multiple issue processors depending on the time at which decisions are made; dynamically during runtime, or statically during compiling time; it can also be said that the distinction depends on the task division between the compiler and the processor. The decisions deal with questions pertaining to how many and which instructions should be executed simultaneously (packaging instructions into issue slots), and how to handle data and control hazards. Usually, each of the two methods embraces the other to achieve better results [**2**]. Two commonly used names for these methods are *Very Long Instruction Word (VLIW)* for static multiple issue, and *Superscalars* for dynamic issue processors [**2**].

An important concept in ILP is Speculation, which is guessing the outcome of an instruction to avoid waiting for it to be calculated. An example would be guessing the outcome of a conditional branch and loading the instruction that is likely to be next. Speculation obviously could be wrong, thus it requires the inclusion of a recovery mechanism if the speculated result was wrong [**2**]. The microarchitecture developed in this project should support multiple issuing of instructions; it has not been decided whether it will be dynamic or static, further research is required.

While ILP has been very significant, it is a fact that manufacturers have reduced its utilization and reduced their pipelines lengths and relied more on multi-cored architectures [**2**] which will be described later in this chapter. There are several reasons for this, one of the most prominent being power efficiency [**2**]. For cutting edge technology, minimizing power consumption has been a growing concern. When also factoring the extensive research done by the industry in this field it seems unlikely that it still has much to offer. Therefore, this approach was not chosen for this project.

2.2.2. SIMD

Parallel hardware can be categorized in terms of instructions streams and data streams; an interesting category would be *Single Instruction Multiple Data* (SIMD) organizations which utilizes multiple data streams with a single instruction [**2**] [**4**]. SIMD requires what is called *data-level parallelism* (similarly structured data. e.g. *for* loops in arrays) in order to be fully utilized. One of the applications of SIMD organizations is their use in multimedia extensions instructions, such as MMX instructions in x86 instruction set. As stated in the previous chapter, this is beyond the scope of this project. Another application is *vector* architectures. Vector architectures fetch the instructions and store them in special registers that act as a pipeline to the ALU; this approach causes in a dramatic reduction in the number of instructions required for constructing a loop because a single instruction loads all the data into the registers to be processed by the ALU. It can be easily shown that vector instructions can run faster than *scalar* (conventional) instructions [**2**]. As mentioned before, these instructions have been a part of the x86 architecture for quite some time. This has two implications, this approach has been and is still being studied and researched extensively in technology leading facilities, and it is likely that most of what it can offer has been achieved. For these two reasons this approach will not be used for this project.

2.2.3. Hardware Multithreading

Hardware multithreading is achieved when multiple threads utilize the same hardware; in other words, when several independent tasks run on the same processor, in contrast to different threads running on different cores (processors) which will be covered in the next section. Hardware multithreading exploits *thread level parallelism.* Typically, multithreading is used to hide the latency in executing a thread that has been *stuck* by switching to another thread. This is particularly useful with lengthy latencies such as cache misses. Approaches to thread switching are normally categorized as *fine-grained* and *coarse-grained.* Fine grained multithreading switches threads every clock, thus it can hide even the shortest latencies since the thread has a few instructions gap to prepare the next instruction. The drawback however is that the individual threads slow down since it is delayed even if no latency exists. Coarse grained multithreading only switches on lengthy *stalls;* which insures that individual threads are not slowed down. Nevertheless, it

does not hide short stalls which may cause the overall execution time to increase [**2**]. Multithreading is particularly useful for *Graphical Processing Units* (GPUs) which may run thousands of threads; the sheer number of threads helps hide the longest latencies. A type of multithreading is *Simultaneous Multithreading (*SM) in which a multiple issue processor runs several threads concurrently. Experiments have shown that SM models perform better than Superscalars and fine-grained multithreading models, which is expected since they combine the multiple issue characteristic of the former with the latency hiding of the latter [**5**]. Out of the discussed variants, SM offers the best performance. While this approach is intriguing and might deserve more research, its complexity is considerably beyond the level of a final year project.

2.2.4 Multiprocessors.

The need for multiprocessor architecture arose with the need to solve computationally costly problems; they appeared in the form of *clusters.* However, with the increasing demand for stronger personal computers, multiprocessors became very popular. These CPUs are called multicore multiprocessors whereas processors are called *cores.* There are two models for multiprocessor architecture depending on the memory construction, namely clusters and shared memory models. Clusters use separate memories for each processor and uses message passing to communicate and synchronize. However, clusters are considered beside the scope of this project. Shared memory structures use the same physical address for all processors which emphasizes the need for synchronizing by implementing means, such as *locking* the data, so that no two processors try modifying the same data or use an outdated value of a variable. [**2**]Another interesting approach to multiprocessing is *General Purpose GPU* (GPGPU) which utilizes the high level of parallelism in GPU to execute highly parallelized algorithms. NVIDIA's *Compute Unified Device Architecture* (CUDA) is a popular example of this approach. [**2**]. Out of these approaches, the one that fits the project scope the best is the shared memory model.

Parallel Processing faces many challenges. Fundamentally, in order to harness its full potential, the tasks executed by the processors should be distributed perfectly evenly. The bottleneck in performance is always the longest sequential code [**2**]. The main hurdle currently faced by parallel computing seems to be the software According to [**3**]. Three reasons are provided for that: the difficulty for programmers to migrate from the sequential mindset, understand the parallel models of programming, and

implement the rather unfamiliar or difficult concepts of parallel programming (e.g. deadlocks, load balancing, scheduling … etc.); the slow rate of development for operating systems and compilers compared to the rapid innovations required to achieve the desired results; and finding a subjective measure for programming languages improvement which has depended on the researchers personal opinions so far [**3**]. The research shown in [**6**] shows that the difference in performance between *naïvely-written* C/C++ code and well optimized code averages 24X (and reaches 53X) for a 6-core Intel® Core™ i7 X980 Westmere CPU. These numbers show that the software awareness of the parallel programming is currently crucial for performance; especially when considering that with utilizing few algorithms and advancements, the gap drops to 1.3X [**6**].

Multicore technology is well researched and implemented, the optimization of the code also affects the result a few tens of times. Due to these reasons, this option is not chosen for this project.

2.3. RECONFIGURABLE COMPUTING

Reconfigurable computing refers to the use of FPGAs or other reconfigurable devices for computation purposes. It has been reported to achieve a performance improvement of 500 times and a power saving of 70% compared to microprocessor architectures [**7**]. However, in the context of this report, this term is used to refer to *run-time reconfigurable computing;* in which the processor employs a reconfigurable unit that is configured on run-time depending on the instructions it should execute. This approach attempts to merge the concept of microprocessor architectures where the same hardware is used for any application and the *Application-specific Integrated Circuit* (ASIC) approach which designs and manufacture an IC specifically tailored to deal with the particular task. Microprocessors are simply insufficient for many applications, and ASICs are notoriously expensive which is why a configurable cheap processor is a very good solution. Reconfigurable computers consist of a processor that is used for light tasks and a fabric that is configured to efficiently execute the application which makes it more capable of achieving high parallelization for example compared to microprocessors. [**7**]. This section will be more detailed because this project will design an architecture based on reconfigurable computing.

Looking into researches in the area it seems that reconfigurable computing architectures typically use a simple, usually RISC, microprocessor and a reconfigurable fabric. Looking into the different architectures that utilize reconfigurable computing, a classification based on the relation between the standard core and the reconfigurable unit; this classification is shown in Figure 1. Comparing the five classes; the fifth case of a processor being embedded in the logic fabric provides the highest bandwidth among the alternative. It also adds the possibility of using a soft core processor which is a processor that is built entirely in an FPGA [**7**]. Considering the resources available for implementation for an undergraduate project, the choice of a soft core processor is likely the most feasible option, it also adds to the simplicity of the whole processing unit; this way any user with a suitable FPGA

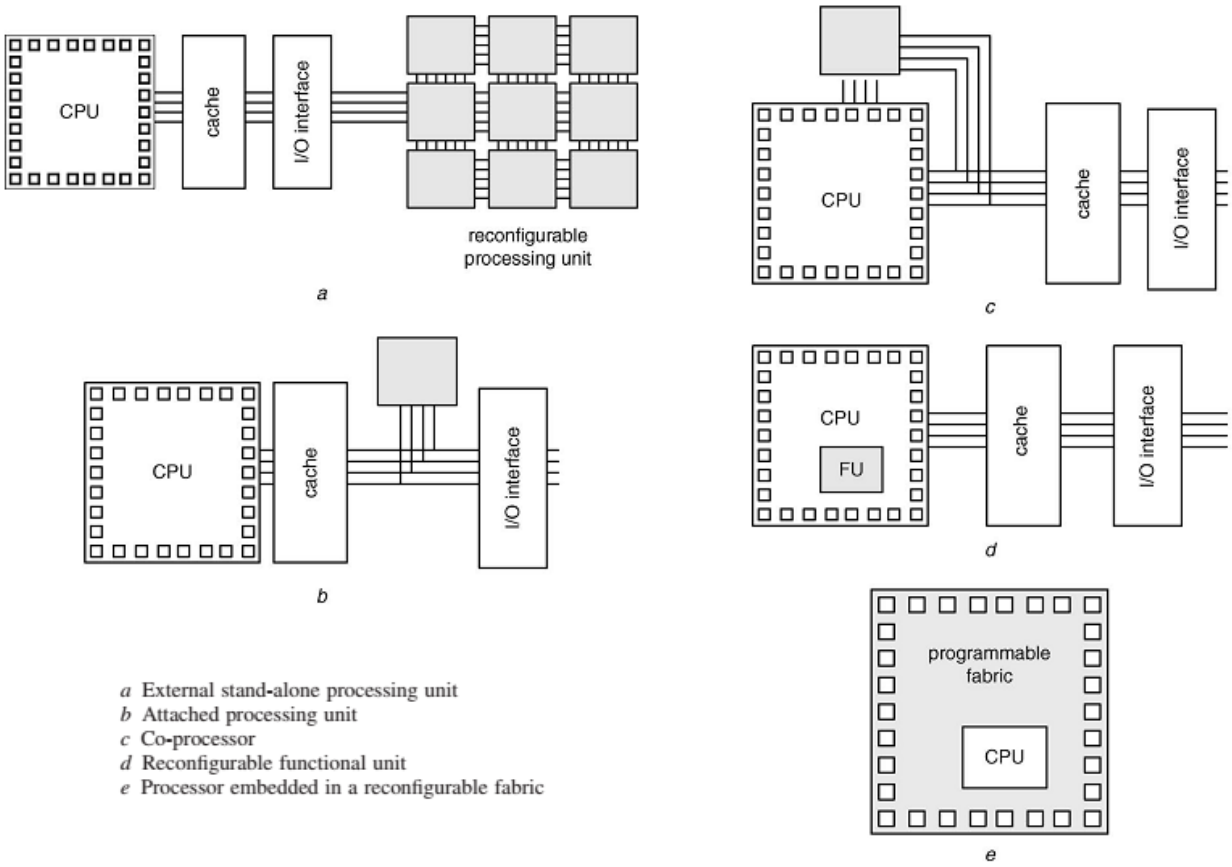

Figure 2. 1 Reconfigurable Systems classes. Adapted from [7]

will be able to download the whole reconfigurable processing unit.

In terms of granularity, a classification into fine-grained and coarse-grained architectures can be used. Fine-grained systems provide a small-sized logical unit that can be highly reconfigured to suit the desired applications; on the other hand,

coarse-grained systems utilize larger logic blocks [**8**]. Some architectures, such as the one proposed in [**9**], utilize several *Reconfigurable Functional Units (RFUs)* and VLIW instructions to make use of all of them at the same time. The architecture proposed there uses the compiler to determine the RFUs configuration most suitable to the 8 simple instructions it merges into a VLIW instruction. This current project will investigate the possibility of delegating this task to the hardware level; in other words, letting the processor itself decode the suitable configuration based on the concatenated instructions.

While intriguing, the reconfigurable computing approach seems very application specific which makes it unsuitable for improving general purpose computing; which might explain it not becoming popular. However, this point can be capitalized on by employing reconfigurable computing in platforms that are usually used for specific applications such as single-board computers; single board computers (e.g. Raspberry Pi) can be used for performing special functionality in different projects by utilizing common Operating Systems such as Linux and their ability to be stand-alone computers that do not require external computers to be programmed (which is the case for microcontrollers). Coupling these features with the vast computing potential of reconfigurable computing might prove to be the gate to propel reconfigurable computing to the public market. From this point of view, this approach is worthy of further research and seems to have good potential.

3.1 FFT:

Discrete Fourier Transform (DFT) is a very essential technique known to every person associated with Digital Signal Processing. Its applications range from filters to image processing to multimedia communication services. DFT basically transforms a discrete time signal into a discrete frequency signal following Equation (2.1).

$$Y[j] = \sum_{k=0}^{N_1-1} X[k] W_N^{jk} \quad ; where\ W_{Njk} = e^{-i2N\pi jk}$$

DFT however is very expensive computationally and has a complexity of $O(N^2)$; making it highly impractical for transforming long series.

It was noticed though that DFT can be broken down into a sum of smaller series to reduce the amount of multiplications. This algorithm is known as Fast Fourier Transform (FFT). The Decimation can be done in Time (DIT) or Decimation in Frequency (DIF) leading to vast performance improvements as shown in the following figure.

In case of N being a power of 2, the asymptotic cost drops to O(N*log(N)). Such major improvement made FFT the typical choice for implementing DFT.

There are many algorithms for implementing FFT, for the sake of simplicity only the most common algorithm, which is the Cooley-Tukey algorithm, is discussed in this project. This algorithm can be seen in the next pseudocode:

```
X_0,...,N-1 ← ditfft2(x, N, s):              DFT of (x_0, x_s, x_2s, ..., x_(N-1)s):
    if N = 1 then
        X_0 ← x_0                                          trivial size-1 DFT base case
    else
        X_0,...,N/2-1 ← ditfft2(x, N/2, 2s)                DFT of (x_0, x_2s, x_4s, ...)
        X_N/2,...,N-1 ← ditfft2(x+s, N/2, 2s)              DFT of (x_s, x_s+2s, x_s+4s, ...)
        for k = 0 to N/2-1                                 combine DFTs of two halves into full DFT:
            t ← X_k
            X_k ← t + exp(-2πi k/N) X_k+N/2
            X_k+N/2 ← t - exp(-2πi k/N) X_k+N/2
        endfor
    endif
```

CHAPTER 3

METHODOLOGY

This project examines the potential of reconfigurable processing in the single board computer class; therefore, the methodology pits this approach against one of the most popular single board computer, that is Raspberry Pi The methodology could be broken into two section, Raspberry Pi implementation of FFT (software implementation) and re-configurable processing implementation of the same function (FPGA implementation).

3.1 SOFTWARE IMPLEMENTAION:

This implementation uses an arm sequential processor for executing the FFT algorithm. The chosen algorithm was radix-2 DIT (Decimation in Time) Cooley-Tukey FFT Algorithm. Before describing the code though, a quick overview of the hardware is appropriate.

Raspberry Pi is a single board computer that uses an arm microprocessor as its CPU. Its small size, computation ability, use of linux, and cheap price made it very appealing for engineers and hobbyists alike for numerous application. It runs a version of Debian linux called Raspbian and that offers it support for various high level languages like python or c++ which is supported through the very popular gcc tool chain. Figures 3.1 and 3.2 show the board and its linux.

Figure 3.1 Raspberry Pi

Figure 3.2 Raspbian

For this project, the code was developed on a windows laptop and copied to the raspberry Pi SD card later. Putty, a serial communication terminal, was used to connect the computer to the Pi through ssh to enable running the Pi headless.

The code itself comprises of three main phases; Time series generation, series elements re-ordering, and then FFT calculation. Each part will be discussed separately.

The series generation was done through the use of the standard library's "stdlib.h" rand() function. The current time was passed as a seed to the random function through srand(), this ensures that successive runs of the code will generate different series because the happen at different times thus using different values for the seed of the random number generator. While this approach has some shortcomings, it is good enough for the purposes of this project and it is very simple to write too. However, the timing taken for generating the string should be excluded from the execution time of the algorithm since in typical scenarios the numbers will be generated or received in another part of the program and then passed to the FFT function.

The recursive nature of Cooley-Tukey algorithm changes the order of the series elements when performing the transform. This happens because the function recursively divides the series into two sub-series of odd indexed elements and even indexed elements. To counter this rearrangement of elements, a step is performed in advance to reorder the elements so that they return to their proper places after the execution of the FFT algorithm. The algorithm used for reordering the elements is bit reversal algorithm which basically swaps the element number $(10000)_2$ with the element number $(00001)_2$ and so on. This simple algorithm leads to the final

frequency series being in-order when it is generated in the next step. The reversal algorithm takes a number in binary, divides it into two halves, and switches them. This process is repeated recursively for until all the bits of the number are swapped. Contrary to the previous step, the elements reordering is included in the total time consumed by the algorithm.

The last step is calculating the FFT itself. This step calls the recursive FFT function that divides and conquers the time series generating a frequency series in the end. The function, whenever called, basically divides the series into two subseries one comprising of even indexed numbers {X_i ; where i=0,2,4,6,..} and odd indexed numbers {X_i ; where i=1,3,5,7,..} and calls itself on each of these two subseries until the length of the subseries reaches 1; a point in which it simply copies the single element of the input subseries into the single element of the output subseries(Note that some implementations performs the transform in-place, for the sake of simplicity the applied algorithm in this project uses a series for the input and another for the output . After the two subseries are calculated, they are combined as typical in divide and conquer algorithms. The appropriate twiddle factors (values of $W_{Njk} = e^{-i2N\pi jk}$) are calculated and multiplied by the elements of the subseries when they are being combined. It should be noted that in future work, the twiddle factors should be calculated outside the FFT function so that their effect on the execution time of the function is eliminated. Especially since many FPGA implementations of FFT use tables to store the twiddle factors before even starting the process.

Finally, the execution time measurement was done through the high resolution clock available in the c++ chrono class. While this clock has considerably higher accuracy compared to using ctime::clock(); it is defined under the C++ 11 standard that is still considered experimental so the effects, if any, of that on the results are unknown to the author of this report; nevertheless, the results should be taken with a grain of salt.

3.2 RECONFIGURABLE SOLUTION:

This solution was intended to be built on a Zynq System on Chip. Zynq SoCs form a line of Xilinx chips each of which comprises of an arm cortex microprocessor embedded in an FPGA Fabric thus making it an example of a class e reconfigurable system as was shown in figure 2.1. Due to budget limitations, the only available Zynq Solutions were the parallella boards. Those boards have a multicore co-processor called epiphany that is connected to the zynq chip thus making it ideal for

testing various approaches such as sequential programming, parallel programming, or even reconfigurable computing for executing any algorithm. However, these boards have a much smaller community and support compared to alternatives like Zedboard or Xilinx Evaluation kits. The delays in receiving the board, lack of support from the parallella community, and relatively weak background of the author about FPGAs contributed to not managing to execute the task on the board. The methodology is mentioned nonetheless for future attempts.

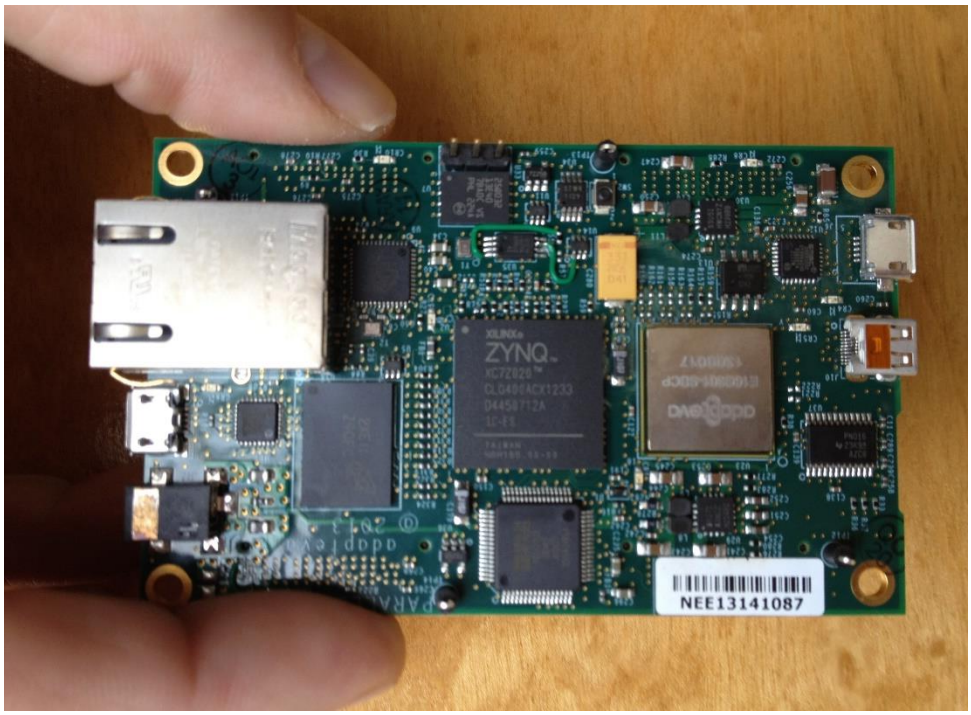

Figure 3.3 Parallella

The evaluated algorithm is the same Cooley-Tukey FFT algorithm that was mentioned in the previous section with its same three phases. The generation of random numbers was to be executed on the cortex processor, because it is not the part of the program that we want to evaluate, and the FFT itself is to be implemented in the FPGA. The transmission of the series from the core to the logical fabric should be very fast thanks to the tight coupling between them since the core is embedded in the fabric. The exact same functions as in the previous section are to be used here since both run on arm cortex cores.

After that, the FFT implementation was to be done in an FPGA. Lack of experience with fpga lead to the author being unable to write a working synthesizable code from

scratch. Some open core FFT processors were also tried with no result. Therefore this step was not completed successfully and that effectively prevented the confirmation or rejection of the null hypothesis of this research which is that reconfigurable computing will considerably outperform the software solution. Sadly, further effort is required to resolve this issue.

3.3 GANTT CHART:

The following Gantt chart shows the timeline for the project, the last deadline was not met leading to the project not being finished on time.

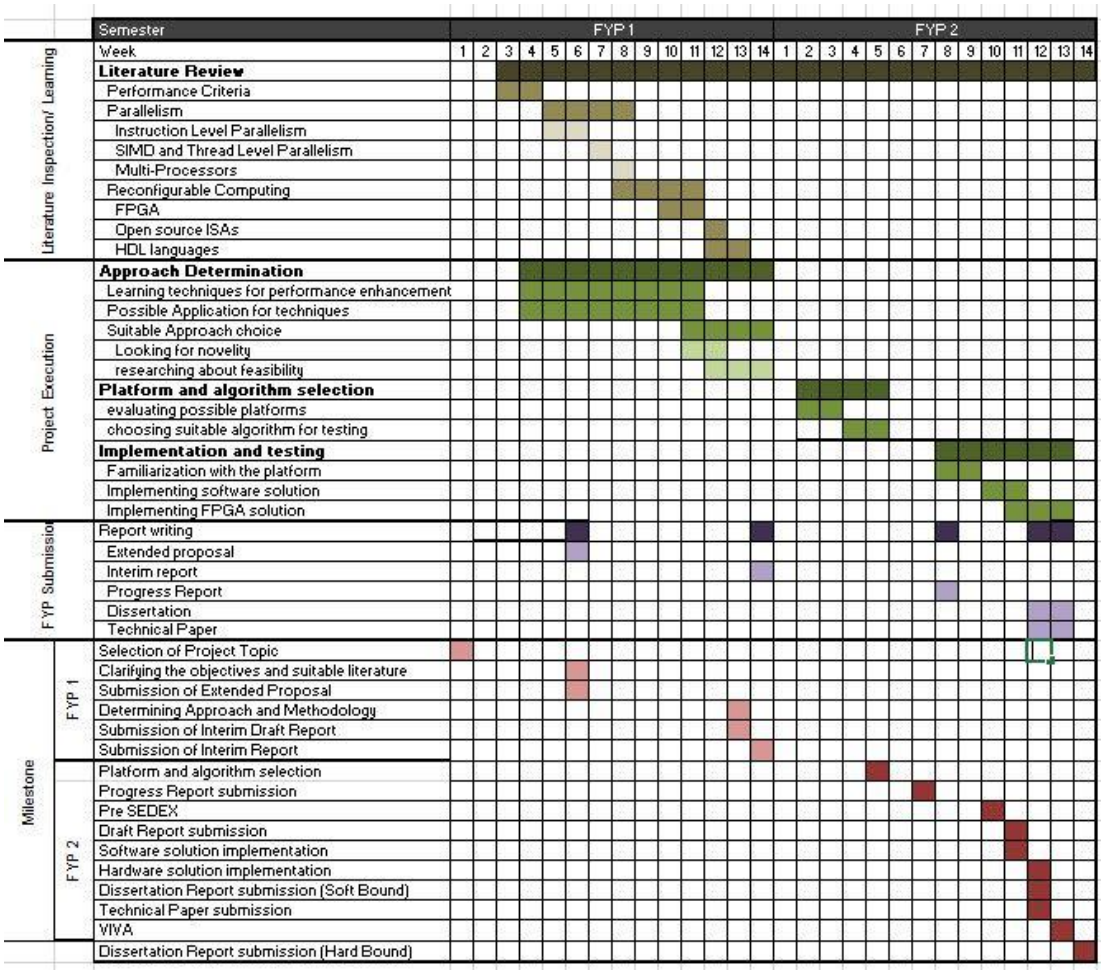

Figure 3.4 FYP Gantt chart

CHAPTER 4

RESULTS AND DISCUSSION

Since this project is composed of two parts, a software solution and a reconfigurable one, the results for both parts will be discussed separately.

4.1 SOFTWARE SOLUTION:

The methodology mentioned in the previous sections was executed for different series lengths. The following table shows the average number of clocks for each number. It should be noted that the times vary with each execution due to how the processor works, the differences are likely attributed to the effect of the operating system and the processor running background processes concurrently with the code.

Series Length (N)	Number of clocks required	Time (ms)
128	770	
256	1100	
512	1700	
1024	3400	
2048	7800	
4096	18000	
8192	43000	
16384	104800	
32768	243500	

In order to visualize the data easier, it has been plotted in Figure4.1 as can be seen in the next page. Figure 4.2 plots a O(N*log(N)) function, and the symmetry between the two plots indicates that this algorithm has that complexity, exactly as expected from the literature review.

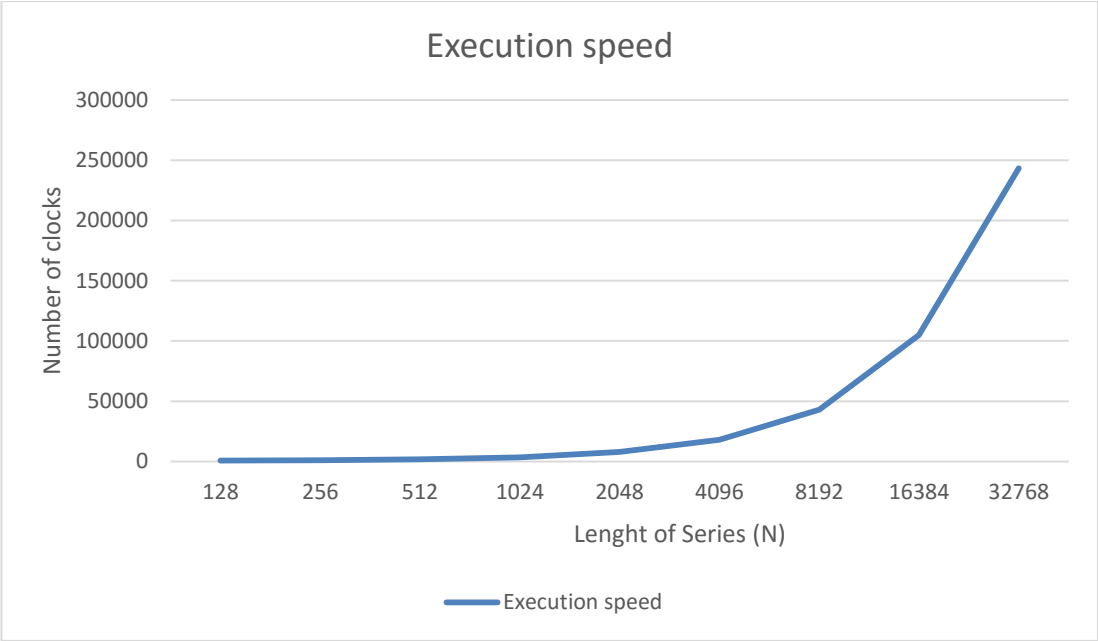

Figure 4.1

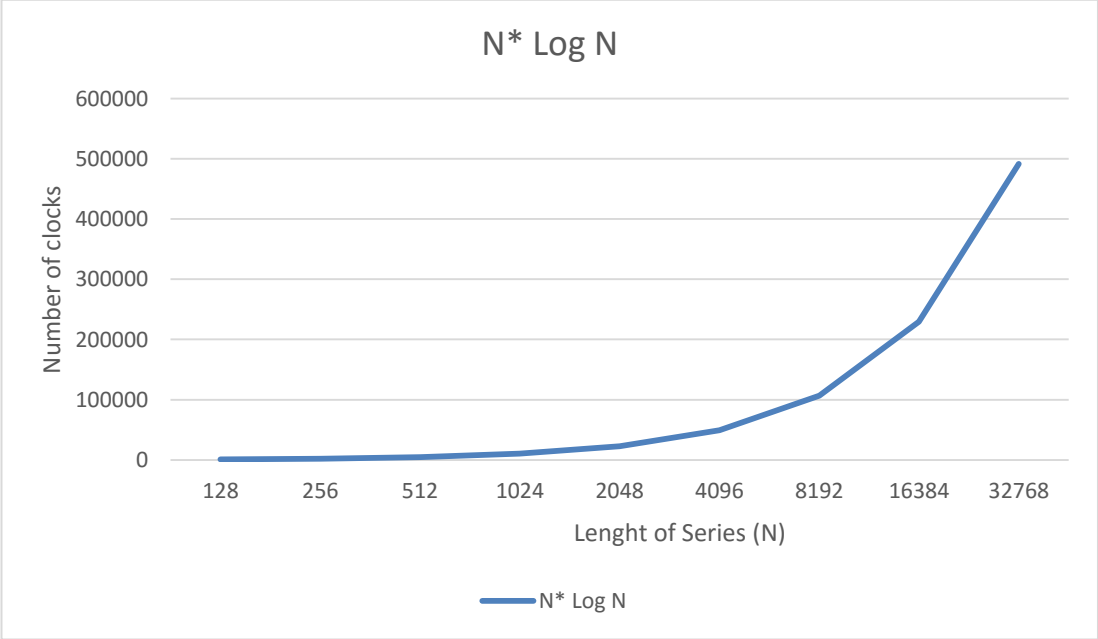

Figure 4.2

RECONFIGURABLE COMPUTING:

As mentioned in the methodology chapter, this part has not been finish until the writing of this dissertation. Therefore there are no results to discuss here.

Nevertheless, the results of the first part indicates huge potential improvement in this part. The software solution had a response of O(N*logN) despite running the code

sequentially. This level of complexity resulted from running the divide and conquer code sequentially but in the case of reconfigurable solution many of these events will run concurrently leading to enormous minimization of the required number of clocks. Considering the highly parallel nature of the algorithm, it is even possible to outperform any parallel processor. Future work could try comparing the performance of the reconfigurable system with that of the epiphany co-processor on Parallella board, an experiment that might show that reconfigurable computing can outperform the trendy multicore options.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

The industry is reaching the physical limits of silicon; further improvements in clock speed are not likely; therefore, alternatives approaches should be utilized in order to keep the computers evolving.

This project envisions a bright future for fine-grained reconfigurable single board computers. It used the very popular FFT algorithm to benchmark both a software solution running on the most popular single board computer (Raspberry Pi), and a reconfigurable computing solution that makes use of the Zynq SOC. The software solution ran the code successfully and produced results consistent with the potential of the algorithm as far as sequential processing is concerned, i.e. O(N log N). However, due to time limitations, lack of experience, and lack of online support, the reconfigurable solution was not materialized successfully.

Future work is needed to complete this experiment. Another recommendation is testing reconfigurable computing against parallel computing, possibly by running the same algorithm on the epiphany multicore coprocessor of the Parallella board.

REFERNCES:

[1] J., Geraci, A., & Katki, F Radatz, "IEEE Standard Glossary of Software Engineering Terminology," 610121990(121990), 3., 1990.

[2] John L. Hennessy David A. Patterson, *Computer Organization and Design : the Hardware/Software Interface*, 4th ed.: Morgan Kaufmann, 2011.

[3] Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, Katherine Yelick Krste Asanovic, "A View of the Parallel Computing Landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56-67, October 2009.

[4] J., Sheaffer, J. W., & Skadron, K Meng, "Robust SIMD: Dynamically adapted SIMD width and multi-threading depth.," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012, pp. 107-118.

[5] D. M., Eggers, S. J., & Levy, H. M. Tullsen, "Simultaneous multithreading: Maximizing on-chip parallelism," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 392-403, July 1995.

[6] Nadathur, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey Satish, "Can traditional programming bridge the ninja performance gap for parallel computing applications?," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 440-451, June 2012.

[7] Timothy J., George A. Constantinides, Steven JE Wilton, Oscar Mencer, Wayne Luk, and Peter YK Cheung. Todman, "Reconfigurable computing: architectures and design methods," *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 2, pp. 193-207, March 2005.

[8] M. A., & Awan, U. S. Iqbal, "An efficient configuration unit design for VLIW based reconfigurable processors. ," in *Multitopic Conference, 2008. INMIC 2008. IEEE International* , 2008, pp. 47-52.

[9] M. A., & Awan, U. S. Iqbal, "Run-time reconfigurable instruction set processor design: Rt-risp.," in *In Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on*, February 2009, pp. 1-6.

[10] H., & Platzner, M. Giefers, "An FPGA-Based Reconfigurable Mesh Many-Core," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 2919-2932, Dec 2014.

[11] Yunsup Lee, David Patterson, Krste Asanovi´ Andrew Waterman, *The RISC-V Instruction Set Manua. lVolume 1: User-Level ISA, Version 2.0.*, 2nd ed.: RISC-V, 2014.

[12] RISC-V. RISC-V. [Online]. http://riscv.org/

[13] SHAKTI Processor Project. [Online]. http://rise.cse.iitm.ac.in/shakti.html

[14] R., & Smith, M. D. Razdan, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 172-180.

APPENDICES

APPENDIX A : C CODE:

```
 #include<stdio.h>

#include <stdlib.h>

#include <math.h>

#include<time.h>

#include<iostream>

#include <chrono>

using namespace std;

using namespace std::chrono;

#define FFT_SIZE 4096

#define M_PI            3.14159265358979323846

//#define FREQ 10

void swap (double &a, double &b)

{

    double t=a;

    a=b;

    b=t;

}

unsigned char reverse(unsigned char b) {

   b = (b & 0xF0) >> 4 | (b & 0x0F) << 4;

   b = (b & 0xCC) >> 2 | (b & 0x33) << 2;

   b = (b & 0xAA) >> 1 | (b & 0x55) << 1;
```

```
   return b;
}
void four1(double* data, unsigned long nn)
{
    unsigned long n, mmax, m, j, istep, i;
    double wtemp, wr, wpr, wpi, wi, theta;
    double tempr, tempi;

    // reverse-binary reindexing
    n = nn<<1;
    j=1;
    for (i=1; i<nn; i+=2) {
/*        if (j>i) {
            swap(data[j-1], data[i-1]);
            swap(data[j], data[i]);
        }
        m = nn;
        while (m>=2 && j>m) {
            j -= m;
            m >>= 1;
        }
        j += m;*/
```

```
j=reverse(i);
swap(data[j-1], data[i-1]);
swap(data[j], data[i]);

}

// here begins the Danielson-Lanczos section
mmax=2;
while (n>mmax) {
   istep = mmax<<1;
   theta = -(2*M_PI/mmax);
   wtemp = sin(0.5*theta);
   wpr = -2.0*wtemp*wtemp;
   wpi = sin(theta);
   wr = 1.0;
   wi = 0.0;
   for (m=1; m < mmax; m += 2) {
      for (i=m; i <= n; i += istep) {
         j=i+mmax;

         tempr = wr*data[j-1] - wi*data[j];
         tempi = wr * data[j] + wi*data[j-1];
```

```
            data[j-1] = data[i-1] - tempr;
            data[j] = data[i] - tempi;
            data[i-1] += tempr;
            data[i] += tempi;
        }
        wtemp=wr;
        wr += wr*wpr - wi*wpi;
        wi += wi*wpr + wtemp*wpi;
      }
      mmax=istep;
   }
}



int main()
{
   double data[FFT_SIZE*2];
   double factor = sqrt(3.0 / 2.0);

   for(int i=0;i<FFT_SIZE*2;i++)
   {
      data[i]=(rand() * 2.0 * factor / RAND_MAX - factor);
   }
```

```
    for(int i=0;i<FFT_SIZE*2;i+=2)
    {
   //     printf("%i:   %f+i%f\n",i/2,data[i],data[i+1]);
    }
    //    clock_t tStart = clock();

   high_resolution_clock::time_point t1 = high_resolution_clock::now();

    four1(data,FFT_SIZE);
    high_resolution_clock::time_point t2 = high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::microseconds>( t2 - t1 ).count();
cout<<duration;
    //printf("\n%i\n",tStart);
/*
for (int i=0;i<324353;i++){
    int j,k;
    j=i;
    k=j;
}*/
//    clock_t tend = clock()-tStart;
```

```
  //   printf("\n%i: %i \n",tStart,tend);
 /*   for(int i=0;i<FFT_SIZE*2;i+=2)
    {
//        printf("%i:   %f+i%f\n",i/2,data[i],data[i+1]);
    }*/


     //      printf("\n%d: %d \n",tStart,tend);
//printf("\nTime elapsed: %.2f\n",1.0*(tend-tStart)/CLOCKS_PER_SEC);
//printf("hhhhhhhhhhh");


    return 0;
}
```