A Many Processing Element Framework for the Discrete Fourier Transform

Andrew van der Byl [#1], Michael Inggs [*2], Richardt H. Wilkinson [#3]

[#] *Centre for Instrumentation Research,*
Department of Electrical Engineering,
Cape Peninsula University of Technology
PO Box 652, Cape Town, 8000, South Africa
[1] vanderbyla@cput.ac.za
[3] rwilknsn@ieee.org

[*] *Department of Electrical Engineering, University of Cape Town,*
Rondebosch, Cape Town, 7701, South Africa
[2] Michael.Inggs@uct.ac.za

***Abstract*—For the greater part of the last 30 years of computing history, computational processing has been primarily based on performance improvements of sequential processors. Research has shown that the future no longer lies in the direction of sequential processor performance improvements, but rather on the adoption of parallel processors and the development of suitable architectures. Computing in general has settled on generic suitable-for-all processing architectures, however recent research has since highlighted the importance of adopting many simpler processing elements, with a tailored focus on domain specific designs. In light of this, this work focuses on the development of a parallel framework for the execution of the Discrete Fourier Transform using a recursive algorithm using many simple processing elements. The system described in this work shows a 64 point Discrete Fourier Transform computation in parallel, which achieves a throughput of 2.19GSPS(39.49Gbit/s). The framework proposed promotes adjustable parallelism using the many simple processing elements, and allows simple adoption on existing systems.**

I. Introduction

For the greater part of the last 30 years of computing history, computational processing has been primarily based on performance improvements of sequential processors. Sequential processors have been the dominant hardware, and as a result, the primary body of software available today has been optimized for use on a sequential processor. Software optimization is based on the best use of the underlying hardware, however a better approach is to first determine the fundamental requirements of the software, and then develop hardware to suit the application. It is true that a system design using this approach could easily become application specific, however if fundamental computations relevant to many applications are identified, such as the Discrete Fourier Transform (DFT) for the signal processing class of applications, processing hardware can be developed and cast as a generic processor applicable to a wide range of applications. The DFT has played a fundamental role in the analysis of signal data applicable to many disciplines. The interpretation and meaning of the results may differ from discipline to discipline, however a common thread through all is the reliance on efficient processing techniques and the availability of hardware resources to perform the necessary computations. The architecture of the underlying hardware is as important as the techniques themselves for efficient use, and it is necessary to ensure that all hardware is utilised as efficiently as possible to promote high throughput.

Different application implementations may differ, however often the fundamental computations remain the same. It is essential to identify both computational and communication requirements of key shared algorithmic sections in a class of applications, and tailor hardware to suit the common sections to allow efficient overall execution. The fundamental computations of the DFT imposes limitations on the practical realisation of the transform, as it comes with a high computational price for a large value of N [1]. For practical implementation of the DFT, many approaches have been explored, and the Fast Fourier Transform (FFT) emerged as the more popular choice as an efficient technique which provided a significant improvement in computational cost [2]. Software which has been optimized for single-thread execution on a single-core processor no longer experiences a performance improvement as threading capabilities are improved in today's generation of multi-core processors. Often legacy code can be compiled for use on multi-threaded hardware, however it suffers diminishing returns as more threads are realised due to computational or communication bottlenecks [3]. This suggests that it may be time to re-evaluate the methods employed for some of the key techniques that have been fundamental in so many disciplines, and discover new methods that can exploit the parallel hardware available today. It is therefore the aim of this study to re-evaluate the requirements of the DFT, and implement efficient processing hardware on a naturally parallel platform such as a Field Programmable Gate Array (FPGA). This work discusses a many-processing element implementation of the DFT, aims for high throughput, and focuses on being expandable as more parallelism can be realised and accommodated in hardware. This paper discusses this work

978-1-4244-8983-1/10/$26.00 ©2010 IEEE

Authorized licensed use limited to: Cape Peninsula University of Technology. Downloaded on May 10,2021 at 07:13:25 UTC from IEEE Xplore. Restrictions apply.

in the following sections: Section II discusses the Recursive DFT and highlights previous work in the field. Section III provides an overview of the FPGA implementation, including the system overview and hardware architectures used. Section IV discusses results and Section V concludes the work.

II. Recursive Discrete Fourier Transform

The DFT is a fundamental algorithm in signal processing, and is used to represent data in a Fourier space. The DFT is an inherently parallel algorithm, however direct implementation (sequentially) of such an algorithm has been shown to be computationally intractable for larger values of *N* [1], and the more efficient Fast Fourier Transform (FFT) is typically used [2]. Both the full parallel approach and the FFT techniques require all input data to be present to compute the transform, and repeated for every change of input, even if only a single data point is added. In the case of an FFT, each successive butterfly stage is dependent on the output of the previous stage, whilst the DFT requires no inter-point communications. If the input sequence of size *N* changes, i.e. a new data sample to position *N+1* were added for example, the entire transform would need to be re-computed. An alternative to this approach is the recursive DFT. The recursive DFT is based on the principle of updating a current output *X[n]* as new data is added to the input sequence. The addition of new data points does not imply that the input sequence has to grow in size (from size *N* to *N+p*, where *p* is one for a single new input to the sequence), but rather imposes the constraint that a window function of size *N* is required, which shifts to include the new sample (*N+p*), and removes the outgoing sample (*N+p-N = p*). If the output is known a priori, an update can be computed by utilizing the Fourier shift theorem and computing a DFT on the new data that has been added, and removing the influence the outgoing sample had on the output. The computational cost for the recursive DFT technique is far lower than the FFT ($O(nlog_2 n)$)[1], [2], and is shown to improve by $\log_2 n$ [4]. The recursive technique is based on the work of [2], [4], [5], [6], [7] and computes an updated *X[n]* by removing the contribution of the outgoing sample (*f(n-N)* lets call this f_{out}) and adding in the contribution of the incoming sample (*f(N+1)* lets call this f_{in}). The new output can be computed using:

$$F_{new}(u) = [F_{current}(u) + \tfrac{f_{in} - f_{out}}{N}] \exp\left(\tfrac{j2\pi u}{N}\right) \quad (1)[4]$$

Using this formula, an updated output can be computed for each DFT point based on the difference between the incoming and outgoing samples. A further advantage of this technique is that each DFT point can compute an update independently (select frequencies of interest can be computed if desired), and only minimal data (new sample) needs to be transferred to the processing elements used for each DFT point. This is particularly advantageous as limited communications between processing elements results. Each update requires simple arithmetic and a single complex multiply for the updated result. The complex multiply relies on a complex exponential, however these value can be computed once-off initially, and re-used for future computations. For efficiency, only the first $\frac{N}{4}$ exponentials need to be computed, as the remaining values can be shown to be of the same magnitude, however only differ by sign and or location (i.e. real or imaginary values). This further reduces the initial setup-cost, and allows a dynamic system to be developed, whereby the number of processing elements can be generated at compile time, and after a small initial setup time, the complex values can be automatically allocated to the correct element computing the given DFT point.

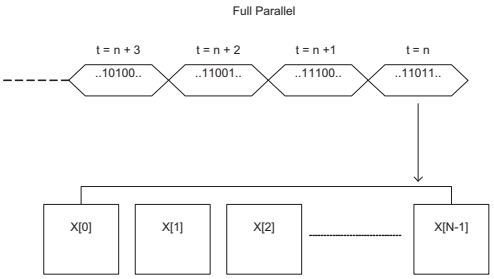

Fig. 1. *N*-wide Parallel Recursive DFT

Previous work [2], [4], [6] shows the initial output needs to be computed (typically using an FFT) before the recursive DFT can be used. It can be argued [2], that for a given data sequence *x[n]*, the output is already know at time *t=0* (*X[n] = 0* $\forall n$). Prior to any data entering the system, it can be assumed that the resulting output *X[n]* is zero, and therefore can be used as the initial state for the recursive DFT. The data sequence can be stepped in, and the resulting output updated as the samples are inserted. At the point where *t=N*, the resulting output matches the DFT output *X[n]* for the full time sequence *x[n]*. Further samples can now be added and the resulting updated output computed. It is important to consider the computational cost at this stage. If the recursive DFT were implemented sequentially, the cost would be in the order of $O(N^2)$. If concurrency were exploited by using many smaller processing elements, multiple DFT points can be computed concurrently, and the cost reduces to $O(N)$, for computing a DFT of length *N* if *N* processing elements are used. A trade-off exists for the initial cost, however the added advantage is no additional hardware is required to support an FFT for a once-off computation. Figure 1 illustrates using *N* computational elements to compute a recursive DFT in parallel.

III. FPGA Implementation

A. Hardware and Software

This work examines the recursive DFT and focuses on a many-processing element parallel implementation. A Xilinx Virtex 5 VSX50T was the reconfigurable fabric selected for this system, as it includes 288 DSP blocks within the fabric space. The inclusion of the DSP blocks allows the system to harness the computational power available within the FPGA. All gateware was jointly written and designed using a combination of VHDL, Matlab, Xilinx blockset and System Generator. The development environment used was Xilinx ISE 11.4, Xilinx AccellDSP and Mathworks Simulink. The design

Authorized licensed use limited to: Cape Peninsula University of Technology. Downloaded on May 10,2021 at 07:13:25 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SYSTEM SPECIFICATIONS

Component	*Specification*
Input	16-bit signed (Real)
DFT Outputs	64 points (Complex)
Resolution	18-bit
Memory	Internal Block RAM (BRAM)
Complex Multipliers	DSP Blocks (18-bit)
Complex Exponentials	Pre-Computed (BRAM)

was kept modular, allowing various aspects of the systems to be swapped when newer, more efficient blocks become available, or are designed.

B. System Specifications

It was noted earlier, that the system described here has not been developed for any particular application, and remains generic to any application requiring a DFT computation. This creates many interesting trade-offs, as a design is usually optimized for a given application, and hence fine system details such as bus widths, precision, as well as throughput become specific to the context of the application. Table I lists the specifications for this design.

The system input consists of a sequential inputs (Real)(16-bits fixed-point, with the fixed-point set at the 14^{th} bit position) which are streamed into the local Block RAM (BRAM). The bus width going to the individual processing elements is 18-bits wide, and originates from the choice of fixed-point resolution that was selected. Since this system is aimed to target a wide range of applications, it was decided to allow for up to 12-bits precision. The main factor which influenced the decision is the latency incurred when performing a complex multiply in a DSP block. The latency remains constant for bit widths up to 18 bits, and therefore, at no additional computational cost, a higher precision output can be obtained, limiting the cost to fabric resources. For this reason, the outputs of the adders and complex multiplier were allowed to grow up to 18-bits before saturating (accommodating the growth without incurring additional computational cycles - binary point remained at the 12^{th} point). The FPGA was capable of fitting 64 complex computational elements (66% of DSP Blocks used), inclusive of the additional supportive hardware (system controller, FIFO, BRAM etc.). The outputs are complex, giving a combined output of 128 real and imaginary values, each at 18-bit fixed-point resolution. A system overview with a further discussion on the specifications listed in Table I are discussed in Section C.

C. System Overview

A block diagram of the system framework is illustrated in Figure 2, and consists of a system controller(Finite State Machine - FSM), subtracter, Block RAM(BRAM) and the multiple processing elements. The system is designed to work independently of other processes, and processing elements, and can easily be attached to existing systems. The design approach for the system was focused exclusively on parallelism. As noted previously, both the DFT and the recursive DFT are inherently parallel algorithms, and exploiting this parallelism is a straight forward task (whilst being resource conscious). This system was feasible for a many-processing element approach, as the inter-element communication for the processing elements is minimal for this type of algorithm, and thus the upper-bound on the degree of performance is limited by FPGA resources and processing latency.

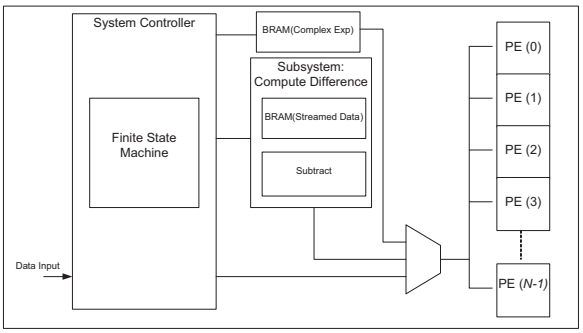

Fig. 2. System Block Diagram

The system was designed to allow expansion and therefore needed a framework which allowed adjustable parallelism. On power-up, the system auto-initialises and allocates all the necessary complex values for the respective processing elements. This introduces an initial penalty of 160 cycles (operating at 120MHz) for a 64 element DFT, but only needs to run once during power-up. The trigonometric values used in each element are pre-computed and stored in separate BRAM, and the FSM assists in the automatic allocation to the respective elements. Each element monitors the bus and stores only the data which corresponds to the respective element address. Following the initial setup, the system is ready to accept input data, and the streamed data which is buffered into BRAM is placed on the bus.

D. Processing Element Architecture

The architecture used for each processing element consists of a complex multiplier for the DFT update, logic and multiplexers for input data selection, and registers for complex value storage. The complex multiplier is a core-generated IP by Xilinx, and is pipelined for performance. The complex multipliers are implemented in the fabric using the DSP blocks so to allow the remaining logic resources for other hardware implementation. Reviewing the formula stated at (1), the update is computed by summing the current output value $F_{current}(u)$ and the difference between the new input f_{in}, and the outgoing value f_{out}. The difference value is not computed locally per element, but is handled by a separate external stage

Authorized licensed use limited to: Cape Peninsula University of Technology. Downloaded on May 10,2021 at 07:13:25 UTC from IEEE Xplore. Restrictions apply.

and is placed on the input bus to the processing elements. The resulting summation is multiplied with the associated complex value of the particular element to produce the output $F_{new}(u)$. This updated output is stored locally and is ready for external use.

IV. Results

A. Error Performance

To assess the computational error introduced into the results by the implementation of fixed point arithmetic, the results imported from both simulation and FPGA hardware co-simulation execution models were compared to the fixed-point results obtained from a Matlab simulation. The accuracy of the system is represented by a square error computation between the simulation and hardware co-simulation results and the Matlab results (FFT). The results between the Simulink simulation and the FPGA execution were accurate, however when compared to the fixed-point FFT performed in Matlab, the square error varied in a cyclic fashion and increased incrementally after each iteration as the data was streamed, and highlighted the accumulated error that results from fixed-point recursive arithmetic[6]. Errors in the order of 10^{-1} were recorded after a few hundred input samples. To alleviate these errors, it has been suggested that the DFT should be recomputed periodically, or rather to implement floating-point arithmetic. Alternative error correction techniques are currently under investigation.

B. Computation Performance Comparison

It is useful at this point to compare the performance of this system to a well established system such an a pipelined 64-pt core generated FFT by Xilinx. The core generated FFT can be divided into two separate sections: The number of cycles required at startup (setup and loading of data) and the number of cycles required at roll-out (as the data is streamed out). It is equally important to consider the proposed synthesis speed that the design can meet. The maximum recommended operating frequency for a 64-pt pipelined FFT is 458MHz, whilst requiring 211 cycles at startup, and 64 cycles during data roll-out. By comparison, the system controller and computational elements are able to operate at 120MHz requiring 160 cycles at startup. The output data is not streamed out sequentially (1 point at a time as in the case of the FFT), but is available simultaneously in lock-step. A more significant feature of the applied algorithm, and subsequent framework is the update rate. For every new sample entering the system, the FFT would need a total of 275 cycles (600 ns) before all output data points (18-bits) are available (rate 3.84Gbit/s). The recursive DFT implementation requires 7 cycles (58 ns) after startup. This yields a data throughput of 2.19GSPS(39.49Gbit/s), however the throughput can be further improved as more parallelism, and subsequently more DFT points are added in parallel, as each point executes in parallel. An additional point of comparison is resource ultilisation of both designs (Table II). The recursive DFT has a higher resource cost in terms of DSP blocks and LUTs, however has the advantage of a higher throughput. The DSP blocks are primarily used by the complex multipliers, however the blocks are not operated at full potential. Future work will focus on the optimization of this stage.

TABLE II
Recursive DFT vs FFT: Resource Utilization

$Resource$	$RecursiveDFT$	FFT
Number of Block RAM/FIFO	3%	7%
Number of Slice Registers	47%	20%
Number of Slice LUTs	43%	19%
DSP48Es	66%	25%

V. Conclusion

The state of computing has recently reached an inflection point, where the future no longer lies in the direction of sequential processor performance improvements, but rather on the adoption of parallel processors, the development of suitable architectures, and the efficient execution of software on multiple computing cores. Computing in general has settled on generic suitable-for-all processing architectures, however recent research has since highlighted the importance of adopting many simpler processing elements, with a tailored focus on domain specific designs. The Fourier transform falls within the spectral domain, and is considered a fundamental computation within this domain. This work highlights the suitability of the recursive DFT when considering a many processing element approach to transforming data into Fourier space. The algorithm, and subsequent architecture requires no inter-element communications, and allows high-throughput data in the order of 2.19GSPS(39.49Gbit/s). The overall system has been designed and implemented into a framework which allows for expansion, and can be integrated into existing processing frameworks to enable efficient processing into Fourier space.

Acknowledgment

The authors would like to thank the Centre for High Performance Computing, in Rosebank, Cape Town, South Africa, for the use of the computing resources, and facilities.

References

[1] A. V. Oppenheim and R. W.Schafer, *Digital Signal Processing*. Prentice Hall, January 2002.
[2] E. Jacobsen and R. Lyons, "The sliding DFT," *Signal Processing Magazine, IEEE*, vol. 20, no. 2, pp. 74–80, Mar 2003.
[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.
[4] B. G. Sherlock and D. M. Monro, "Moving Discrete Fourier Transform," *IEE Proceedings*, vol. 139, no. 4, pp. 279–282, August 1992.
[5] E. Jacobsen and R. Lyons, "An update to the sliding DFT," *Signal Processing Magazine, IEEE*, vol. 21, no. 1, pp. 110–111, Jan 2004.
[6] A. Brown, "Running Fourier Transforms," *Electronics World*, p. 959, November 1998. [Online]. Available: http://techdoc.kvindesland.no/radio/ymse1/20061216153052112.pdf
[7] H. Kamei, T. Harada, and H. Kawarada, "A fast algorithm of running Fourier Transform," *Electronics and Communications in Japan (Part I: Communications)*, vol. 71, no. 8, pp. 1–10, 1988. [Online]. Available: http://dx.doi.org/10.1002/ecja.4410710801

Authorized licensed use limited to: Cape Peninsula University of Technology. Downloaded on May 10,2021 at 07:13:25 UTC from IEEE Xplore. Restrictions apply.