2005-06-16

The Hybrid Architecture Parallel Fast Fourier Transform (HAPFFT)

Joseph M. Palmer
Brigham Young University - Provo

THE HYBRID ARCHITECTURE PARALLEL FAST FOURIER TRANSFORM (HAPFFT)

by

Joseph McRae Palmer

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

August 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Joseph McRae Palmer

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Brent E. Nelson, Chair
Date	Michael J. Wirthlin
Date	Clark N. Taylor

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Joseph McRae Palmer in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Brent E. Nelson
Chair, Graduate Committee

Accepted for the Department

Michael A. Jensen
Graduate Coordinator

Accepted for the College

Douglas M. Chabries
Dean, Ira A. Fulton College
of Engineering and Technology

ABSTRACT

THE HYBRID ARCHITECTURE PARALLEL FAST FOURIER TRANSFORM (HAPFFT)

Joseph McRae Palmer

Department of Electrical and Computer Engineering

Master of Science

The FFT is an efficient algorithm for computing the DFT. It drastically reduces the cost of implementing the DFT on digital computing systems. Nevertheless, the FFT is still computationally intensive, and continued technological advances of computers demand larger and faster implementations of this algorithm.

Past attempts at producing high-performance, and small FFT implementations, have focused on custom hardware (ASICs and FPGAs). Ultimately, the most efficient have been single-chipped, streaming I/O, pipelined FFT architectures. These architectures increase computational concurrency through the use of hardware pipelining.

Streaming I/O, pipelined FFT architectures are capable of accepting a single data sample every clock cycle. In principle, the maximum clock frequency of such a circuit is limited only by its critical delay path. The delay of the critical path may be decreased by the addition of pipeline registers. Nevertheless this solution gives

diminishing returns. Thus, the streaming I/O, pipelined FFT is ultimately limited in the maximum performance it can provide.

Attempts have been made to map the Parallel FFT algorithm to custom hardware. Yet, the Parallel FFT was formulated and optimized to execute on a machine with multiple, *identical*, processing elements. When executed on such a machine, the FFT requires a large expense on communications. Therefore, a direct mapping of the Parallel FFT to custom hardware results in a circuit with complex control and global data movement.

This thesis proposes the Hybrid Architecture Parallel FFT (HAPFFT) as an alternative. The HAPFFT is an improved formulation for building Parallel FFT custom hardware modules. It provides improved performance, efficient resource utilization, and reduced design time.

The HAPFFT is modular in nature. It includes a custom front-end parallel processing unit which produces intermediate results. The intermediate results are sent to multiple, independent FFT modules. These independent modules form the back-end of the HAPFFT, and are generic, meaning that any prexisting FFT architecture may be used. With P back-end modules a speedup of P will be achieved, in comparison to an FFT module composed solely of a single module. Furthermore, the HAPFFT defines the front-end processing unit as a function of P. It hides the high communication costs typically seen in Parallel FFTs. Reductions in control complexity, memory demands, and logical resources, are achieved.

An extraordinary result of the HAPFFT formulation is a *sublinear area-time growth*. This phenomenon is often also called *superlinear speedup*. Sublinear area-time growth and superlinear speedup are equivalent terms. This thesis will subsequently use the term superlinear speedup to refer to the HAPFFT's outstanding speedup behavior.

A further benefit resulting from the HAPFFT formulation is reduced design time. Because the HAPFFT defines only the front-end module, and because the back-end parallel modules may be composed of any preexisting FFT modules, total design time for a HAPFFT is greatly reduced.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Brent Nelson for his advice and help throughout the course of this project. Especially for his patience in teaching a new graduate student how to correctly perform research.

Thank you also to Sandia National Laboratories for providing the funding for most of this work.

Finally, this thesis would not have been possible without the support of my wife, Betty. I'm grateful for her patience, and for her desire to see me succeed. I'm also grateful for the inspiration that she and our three children have given me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The discrete Fourier transform (DFT) is a fundamental mathematical operations used in digital signal processing. It allows the user to analyze, modify, and synthesize signals in a digital environment. Because of this, it has found a wide range of uses in engineering and scientific applications.

The DFT is performed on a discrete numerical sequence. This is in contrast to the analog Fourier transform, which operates on continuous signals. A discrete sequence is typically a sampling in time of an analog signal, but this is not always the case. For instance, the two-dimensional DFT plays a valuable role in frequency-domain image processing. It operates on discrete data representing image pixels, sampled spatially, rather than temporally.

The DFT produces a spectral profile of the frequency components found within a sequence. In other words, it transforms the sequence from a sequence domain (for example, the time domain, or the spatial domain) to the frequency domain. The resulting transformed signal can then be analyzed, or manipulated in ways that are not possible in the sequence domain, or in a manner that would be difficult or time consuming. For example, a common application of the DFT is in digital filtering. If a noisy input is known to contain a useful signal within a known bandwidth, the DFT can be used to first produce a spectral profile of the signal. Next, one can nullify all signal components outside the target bandwidth. When the now modified frequency profile is subsequently transformed back from the frequency domain to its original domain, the undesired noise will be greatly reduced. Though this same operation can be performed outside the frequency domain, it must be done using time-domain

convolution. Convolution becomes prohibitively expensive for anything but small sequences.

Prior to the introduction of the fast Fourier transform (FFT), signal processing had been mostly limited to analog methods; the DFT was seen as an academic curiosity, with few practical uses. This is because in terms of computational *time complexity*, the DFT algorithm exhibits a $O(N^2)$ execution time.[1] Because it was such an expensive operation, the primitive digital computers of the time could not produce results in a manner that was timely enough for practical applications.

As an example of the computational challenges related to the DFT, in 1964 (the eve of the introduction of the FFT) the CDC 6600 was the premier supercomputer in the world, capable of sustaining 1 million floating-point operations per second (FLOPS). An important signal processing application in that era was radar range discrimination. One of the tasks of a surveillance radar is to determine the distance of a target. This is typically accomplished through some type of signal filtering. The ability of a radar to resolve targets at various distances is known as its range discrimination. Consider a hypothetical radar that can discriminate targets separated by more than 500 m in range. Such a system, if implemented using DSP techniques, would require a digital sampling rate of approximately 1 MHz. Ignoring a large number of details, if the system must detect targets up to 150 Km in distance, it might need to compile a 1024-point DFT every 1 milliseconds. Yet, in 1964, the most powerful supercomputer in the world, the CDC 6600, would have needed at least 8 seconds to complete a 1024-point DFT! Considering that this is an example of a relatively tame radar system, digital filtering techniques were obviously not a practical solution for radar engineers in 1964.

The FFT is an efficient algorithm for computing the DFT. Though variations of the FFT were invented prior to 1965, it was not until that year that the seminal paper by Cooley and Tukey [6] presented the first widely used FFT algorithm. Because the

[1]This terminology is adopted from the field of computational theory. The notation $O(Z(N))$ is defined as "on the order of Z", where Z is some function of N, and N represents the problem size. Thus, the DFT exhibits an execution time "on the order of N^2". It becomes prohibitively expensive for anything but the smallest input sequences.

Cooley-Tukey FFT allowed the DFT to be efficiently computed on digital computers, it had a tremendous impact on a wide-range of fields. Using the previous radar example, if the FFT were used to compute the 1024-point DFT, then the CDC 6600 would now only require about 50 milliseconds. Though still too slow for the example system, a DSP solution is now not so far out of reach. Thus, with the adoption of the FFT, a large number of signal processing algorithms became of more than just academic interest.

Despite the tremendous advancements made in digital computers during recent decades, the impact of the FFT continues to be felt. Many technologies enjoyed by the common public would as yet not be possible without the Cooley-Tukey FFT and its derivatives. Synthetic aperture radar (SAR), a type of imaging radar, operates at sampling rates of hundreds of Mega-Hertz, or even Giga-Hertz. A 4096-point DFT might need to be computed every 800 micro-seconds. This is a tremendous computational load, even for modern digital computers. A typical general-purpose computer would be hard-pressed to sustain such a load in real-time. If implemented using the DFT, then the task would be impossible.

Though the FFT offers performance advantages over the DFT, it is nevertheless an expensive operation. This is compounded by the fact that technologies continue to appear which demand ever higher data throughput, executed on larger and larger data sets. For example, some real-time radar systems require a 4096-point DFT to be computed with a data sample rate exceeding 500 million samples per second. Such a *single* module must execute at the equivalent rate of about 40 GFLOPS, and maintain a data throughput of 32 Gbps.

This example shows that some applications of the FFT are beyond any general purpose microprocessor, and even some of the latest multiprocessing systems. Considering that some DSP algorithms require multiple DFT calculations to be executed concurrently, and on a platform that is both small and low-power, it is clear that the demand for high-performance FFT implementations has only increased with time, and will continue for the foreseeable future.

There are a number of performance metrics that can be used to evaluate a given implementation of the FFT. The four most useful are data throughput, transform size, resource requirements, and power requirements. This thesis develops a high-performance, parallel FFT architecture, called the Hybrid Architecture Parallel FFT (HAPFFT). The HAPFFT is targeted for single-chip, high-performance, custom hardware applications. Transform size and data throughput were the primary design criteria, with resource requirements of secondary concern. Power was never considered, and thus will not be discussed further.

The first performance metric, data throughput, is the principal means of measuring FFT performance. The FFT is often incorporated into a signal processing pipeline. Data proceeds down this pipeline, and is processed in various ways at different stages, eventually exiting the pipeline fully processed. The rate at which the pipeline can process data is limited by its slowest component. Thus, an FFT stage must be able to provide a minimum level of throughput so that it does not become a processing bottleneck. Using DSP terminology, this minimum pipeline throughput is referred to as the data sample rate, and is measured in terms of *samples per second* (sps). For example, a DSP pipeline running at 330 Ksps must be able to process 330 thousand samples every second.

The second FFT performance metric is transform size. There are a number of reasons for demanding a large transform size. First, typically the FFT is computed on an entire block of discrete data. But, if the block is too large, it may not be possible to efficiently compute an FFT for the whole block. This could be a result of either memory or computational resources. In such a case other techniques exist for approximating a frequency profile for the data block, but the results will be inferior. Second, the *frequency resolution* of the FFT output is proportional to the size of the transform. For example, a 1024-point FFT, though computationally more expensive than a 256-point FFT, will nevertheless have four times the resolution. For applications demanding a high level of precision, it is desirable to use the largest possible FFT transform size. In fact, in some, the input sequence is zero-padded in order to produce a larger input sequence, and thus a finer output resolution.

The third FFT performance metric is resource requirements. No matter how high the throughput of a given implementation, it is of little use if its hardware requirements are unrealistic. There exist FFT architectures that though slow, require very little hardware. Likewise, extremely high throughputs can be achieved by the use of massive amounts of hardware. A useful architecture must find a good balance that meets throughput requirements within the resource constraints.

Throughput, and resource requirements are related to the transform size. As discussed earlier in this section, the FFT has a time complexity of $O(N \log N)$. Also, its *memory complexity* is $O(N)$. What this signifies is that, for a constant level of throughput, the computational resources grow by $O(N \log N)$, and the memory resources by $O(N)$, in proportion to the transform size, N.

One means of measuring how efficient a given FFT implementation uses its resources is to quantify its hardware *utilization*. Utilization is a metric for evaluating hardware efficiency. It is the percentage of time that a given hardware resource is doing useful work.

The best way to decrease the hardware requirements of an FFT implementation, and yet maintain throughput, is to increase the hardware utilization. General purpose processors are inefficient because a large fraction of their composition is made up of functional units that are rarely used. Because of their "jack of all trades" approach, they must be able to handle not only the common case, but also any exceptional cases, no matter how rare. Thus, a significant portion of their hardware is idle at any given instant.

In contrast, custom hardware implementations of the FFT are constrained to a single, or narrow range of uses. Therefore, they can achieve much higher hardware utilization in comparison to a general-purpose processor. This will be directly translated into either lower resource requirements, or higher throughput.

Two common custom hardware FFT paradigms are in use. The first is the streaming I/O pipeline. It consists of a pipeline capable of processing a single stream of data at a constant rate of throughput. A single data sample can be accepted every

clock cycle. The other is the *bursty* pipeline. It will accept a burst of data for a short time, after which the stream must stall until the data is processed.

The streaming I/O pipelines give the best throughput, since the data stream is never stalled. Nevertheless, they are only able to process a single data point every clock cycle. Thus, the maximum performance will be limited by the maximum achievable clock frequency.

The conventional pipelined FFTs achieve high throughput by increasing the computational concurrency. This concurrency is found by pipelining the computations. But, because the clock frequency ceiling imposes limits on the maximum achievable throughput, additional concurrency must be found using other methods.

The parallel FFT has long been used in the supercomputing community [17, 2]. The parallel FFT increases concurrency by executing kernels of the FFT simultaneously in parallel. This approach is orthogonal to hardware pipelining, and thus the two approaches can be easily combined. This translates into an FFT composed of multiple, parallel pipelines. Because of the multiple pipelines, it can now accept multiple samples each clock cycle.

Many recent research efforts[13, 5, 31, 20, 30, 18, 29, 8, 11] have investigated techniques (see Section 1.2 for more details) that allow the hardware FFTs to process more than a single sample each clock cycle. Most have attempted to map the parallel FFT algorithm to hardware. While achieving their performance objectives, such a direct mapping is not efficient. The parallel FFT algorithm assumes execution is on a parallel computing machine with multiple, *identical* processors. Because of the homogeneous nature of the computing environment, data movement is global, and control is complex. A direct mapping of this algorithm to hardware does not take advantage of the flexibility of custom hardware in overcoming these performance and design obstacles.

This thesis proposes an alternative high-performance FFT architecture: the Hybrid Architecture Parallel FFT (HAPFFT). Rather than mapping the parallel FFT to hardware, the HAPFFT instead traces its roots from the custom hardware single-pipeline FFT architectures already in use. It is modular in nature, and includes a

custom front-end parallel processing unit which produces intermediate results. The intermediate results are then sent to multiple, independent FFT modules. The formulation hides the Parallel FFTs communication details within the front-end processing unit. No global communication is necessary between the independent, back-end modules.

The HAPFFT's resulting control requirements are therefore simple, and the architecture is straight-forward to implement. Also, the back-end FFT modules can be implemented using the designer's architecture of choice. The HAPFFT's purpose is to enable the designer to incorporate already existing FFT modules into a parallel environment. It formulates the hardware and computations necessary for achieving this integration. Additionally, my analysis and experimental results have shown that the HAPFFT exhibits sublinear area-time growth, or alternatively, *superlinear speedup*[2]. The HAPFFT makes efficient use of hardware resources while achieving its performance goals.

1.1 Thesis Organization

The thesis is organized as follows: Chapter 2 will cover the Fast Fourier Transform, with a focus on deriving the abstract algorithms for computing it. An understanding of these algorithms forms the basis for deriving the HAPFFT. Chapter 3 is a survey of architectural techniques for creating high-performance implementations of the FFT. It will cover pipelined FFT architectures, two common parallel FFT algorithms for parallel processing environments, and survey recent attempts to produce hardware parallel FFTs. Chapter 4 derives the HAPFFT. It gives a general formulation of the architecture, discusses some example implementations of it, and then presents and analyzes the results of my implementation experiments. Finally, Chapter 5 concludes the thesis. It will discuss future research possibilities using the HAPFFT.

[2]Superlinear speedup is a phenomenon in which a new custom hardware implementation of some application achieves an M-times speedup (over previous implementations) with less then an M-times increase in hardware.

1.2 Related Work

The fast Fourier transform has been one of the most thoroughly studied computing algorithms in the last four decades. This is both because of its importance in so many scientific and engineering fields, and because it is computationally expensive. Literally hundreds of papers have been published alone on the topic of custom hardware FFT architectures. This doesn't include the countless others which investigate its implementation in software environments, it proper usage, or algorithmic variations (two-point, Singleton three-point, Winograd 5-point, PTL 9-point, mixed-radix, convolution approach, prime-factor, etc.).

Despite the large body of research on FFT architectures, only a select few have focused on parallel FFT architectures for single-chip implementations. All have been published within the last ten years, with the papers from the last two years being the most closely related to the HAPFFT.

The first custom hardware parallel FFTs were implemented in multi-chip environments. Up until the last few years, integrated circuit technology did not provide the transistor densities necessary for implementing a useful sized parallel FFT on a single chip [31, 20, 16, 30, 18, 19, 22]. As an example of the computational complexity of the FFT, as recent as 1984, a 4096-point streaming I/O *single*-pipelined FFT required eleven printed circuit boards, and 1,380 discrete chips![24]

The HAPFFT is intended for single-chip implementations (though the formulation could be easily adapted for a multi-chip environment). The published work on multi-chip, parallel FFTs, is not closely related. Most of the implementations take a multi-processor, software implementation of the FFT, and replace the processors with ASICs. The more noteworthy are [29, 8, 5, 13, 11]. The most recent, and most interesting is COBRA[5]. It is based on a single, 64-point FFT chip. The chip is designed such that multiple chips can be configured in arrays, thereby both permitting larger transform sizes, and increasing potential concurrency.

Recent years have seen several proposals for single-chip parallel FFTs, as well as two commercial offerings. Both Pentek[19] and SiWorks[22] have released parallel FFT IP cores in the last two years. Pentek has recently published high-level details of

their implementation. Their architecture implements a commutator-based streaming I/O pipeline variation based on the Radix-4 Multi-Delay Commutator (R4MDC), which will be examined in more detail in Section 3.1.1. The implementation is simple, but not very imaginative. The result is higher resource requirements, and lower clock frequencies than the HAPFFT. SiWorks has not published any details on their architecture, though they have implementation results. There implementation also requires more resources, and lower clock frequencies, in comparison to the HAPFFT.

In [18] a single-chip 4096-point FFT is developed which uses eight processing elements. But, the architecture does not take advantage of hardware pipelining, only parallel execution. Thus, the performance results are disappointing. In addition, control is very complicated, and any implementation would be difficult.

In [30], a single-chip parallel FFT is presented which makes use of the CORDIC algorithm for computing the twiddle factor multiplications. Nevertheless, the implementation is targeted for area-constrained, low-power applications, and a small transform size of 128-points. It is therefore difficult to draw a comparison with the HAPFFT.

A multi-pipelined FFT synthesis tool is presented in [20]. The authors' intent is to develop an automated FFT synthesizer to be operated in a manner similar to DISC[26]. The resulting modules obtain parallelism through the use of arrays of processing elements. The work is not complete, and the results they do post are both slow and large. Nevertheless, this may be more a result of the inadequacies of their automated synthesizer than the chosen architecture.

Except for the commercial parallel FFT offerings, the results in [31] come closest to that of the HAPFFT. This work is architecturally similar to the multi-chip FFT presented in [13], except that it is targeted for a single chip. The resulting 4096-point module implements eight parallel pipelines and exhibits good performance. But, control is complicated, and the resource requirements are excessive, requiring 1.5-4 times that of a similar sized HAPFFT.

In all the reviewed works, no architecture was found that can compete with the HAPFFT in terms of resource requirements versus throughput, or simplicity of control and communication. Additionally, the HAPFFT offers a degree of flexibility far beyond these other results. This is because the number of pipelines in the HAPFFT can be easily varied, and the parallel pipelines themselves are architecturally independent of the HAPFFT's formulation.

Chapter 2

The Fast Fourier Transform

In order to more fully understand the operation of high-throughput FFT architectures, one must first study the FFT algorithm. There exist numerous algorithmic variations of it, and this chapter will derive and explain the most common. In addition, insight into these particular algorithms is required to fully understand the HAPFFT.

First, I will motivate the existence of the FFT by using time complexity analysis to compare it to the DFT. Next, I will derive three different FFT algorithms that are commonly used. Finally, I will briefly cover the mixed-radix FFT.

2.1 Motivation for the FFT

For the discrete sequence $x[n] = x(0), x(1), ..., x(N-1)$ of length N, the DFT, $X[m]$, is defined as

$$X[m] = \sum_{n=0}^{N-1} x[n] W_N^{mn}, \qquad 0 \leq m < N, \tag{2.1}$$

where $W_N = e^{-j2\pi/N}$. W_N is known as the $(1/N)-th$ root *twiddle factor*. Note that the index term m is unit-less. This is the primary difference between the DFT and the discrete-time Fourier transform (DTFT). The DTFT is a special case of the DFT, in which the input sequence is assumed to be defined in the time domain. The DTFT will always use the input index term t, for time.

The DFT compiles a sequence $X[m]$ of length N. Each element of $X[m]$ denotes the relative magnitude of a frequency component of the original sequence,

$x[n]$. The frequency is in terms of the sampling frequency, i.e. the inverse of the spacing between samples. A *frequency bin* of $X[m]$ is given in terms of m as

$$f = mf_s/N, \tag{2.2}$$

where f_s is the sample frequency, and N is the length of the sequence, with f being in units of Hertz. For example, for a sequence of length $N = 256$, sampled at $f_s = 1$ KHz, the element $m = 10$ of the DFT would correspond to the frequency $f = (10)(1000Hz)/(256) = 39.06$ Hz.

Analyzing Equation 2.1, we can see that each element of $X[m]$ requires N complex multiplications and $N - 1$ complex additions. Thus, the time complexity of computing the DFT for a sequence of length N is $O(N^2)$. Though not intractable, it is nevertheless very expensive. On the other hand, the FFT produces a result identical to the DFT [1], but has a time complexity of only $O(N \log N)$. To put this in perspective, for a sequence of 1024 elements (a common length encountered in real-world applications), the DFT is $\frac{O(1024^2)}{O(1024 \log(1024))} = 102.4$ times more complex than the FFT. For a sequence of 16,384 elements (again, a typical size), the DFT is 1,170.3 times more complex than the FFT of the same sequence.

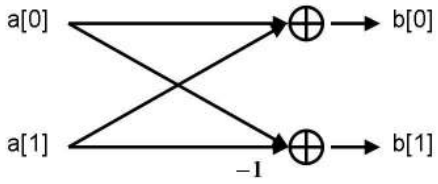
2.1.1 Frequency Aliasing in the DFT

Before plunging into the derivation of the FFT algorithms, I will briefly discuss an issue that effects how the DFT is used.

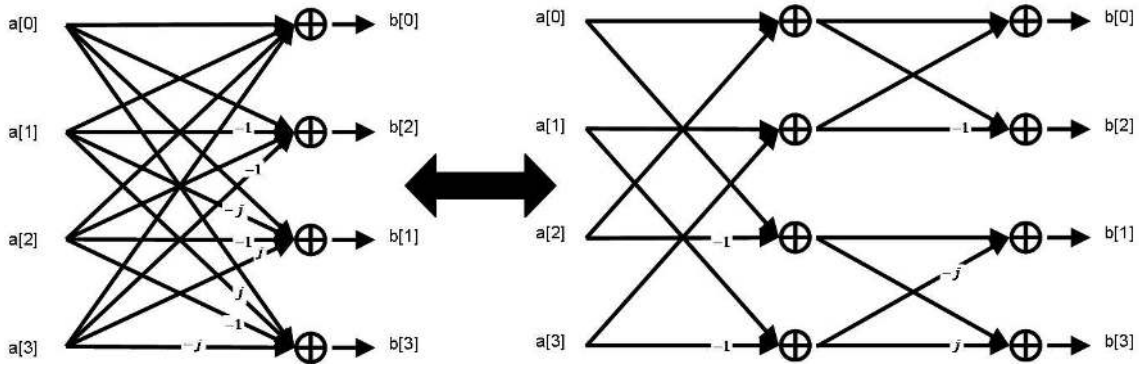
Because the DFT operates on sampled, discrete data, a phenomenon known as *frequency aliasing* can occur. The Nyquist sampling theorem states that a sequence, $x[n]$, is uniquely determined if the sampling frequency of its elements is at least twice the bandwidth of the sequence. Thus, the maximum detectable frequency of the DFT of $x[n]$, $X[m]$, is $f_s/2$. If frequency components exist above this limit, then they will still appear in the DFT output. But, they will be mislabeled as lower frequencies. In other words, the high frequency components will be aliased.

[1] This is not entirely correct. As will be seen in Sections 2.2.1 - 2.2.3, either the output or input of the FFT is scrambled. Many applications require that it be reordered.

Because frequency aliasing can produce incorrect DFT output, one way to reduce its effect is to low-pass filter the input sequence before compiling the DFT. This will reduce the effect of unwanted, high-frequency signals. For a more detailed discussion of this topic, please refer to the relevant chapters in [15] or [12].



(a) BF2



(b) BF4

Figure 2.1: Radix-2 and Radix-4 Butterflies.

2.2 Three Common FFT Algorithms

The original Cooley-Tukey FFT has also come to be known as the radix-2 decimation in frequency FFT. Over the years many derivatives of it have been introduced. I will cover three of the most common here, the radix-2 decimation in time and decimation in frequency algorithms, and the radix-4 decimation in frequency algorithm.

One of the criteria that distinguishes different FFT algorithms is the FFT *radix*. The radix determines one of the atomic building block of the algorithm. I have already mentioned the radix-2 and radix-4 FFTs. These atomic units of computation are known as FFT *butterflies*. Figure 2.1 shows the radix-2 and radix-4 butterflies. They are called butterflies because of their distinctive shape. The radix-4 butterfly is also often referred to as the FFT *dragonfly*. The other FFT building block is the twiddle-factor complex multiplier.

The radix-2 butterfly is used to construct FFT algorithms for operating on sequences of a size that is a power-of-two. The radix-4 butterfly is the building block for power-of-four FFT algorithms. Though the radix-4 algorithms are more restrictive on available input sequences, they require fewer twiddle-factor multiplications. The general rule is that as the radix of the butterflies increase, fewer twiddle factor multiplications are required, but this is at the expense of less flexibility in available sizes.

What does a butterfly compute? It computes a DFT of size n, where n is the radix. So the radix-2 butterfly computes a 2-point DFT, and the radix-4 butterfly computes a 4-point DFT. There exist dozens of other FFT butterflies of varying radices, each an atomic unit that computes some n-point DFT. See Chapter 8 of [23] for more details. A knowledge of these will be useful in Section 2.3, when I discuss the mixed-radix FFT.

The following derivations will show how the radix-2 and radix-4 butterflies are incorporated into three different FFT algorithms.

2.2.1 The Decimation in Time Radix-2 FFT

The decimation in time (DIT) radix-2 FFT is the most intuitive FFT algorithm, and the simplest to derive, so it will be presented first. It is also the same algorithm presented in the original Cooley and Tukey paper[6] on the FFT.

The term *decimation in time* refers to the method of derivation. Given the DFT for a discrete data sequence in time,

$$X[\omega] = \sum_{t=0}^{N-1} x[t] W_N^{\omega n}, \tag{2.3}$$

where N is the length of $x[t]$, the DIT FFT follows by recursively splitting the DFT of $x[t]$ into multiple, smaller DFTs of subsequences of $x[t]$; in other words, to decimate $x[t]$ in time. For the radix-2 DIT FFT, $x[t]$ will be recursively decimated into two smaller sequences of length $N/2$.

Given the discrete sequence $x[n] = \{x[0], x[1], ..., x[N-1]\}$, where N is power-of-two, the DFT of $x[n]$, $X[m]$, is given by (2.1). The summation of (2.1) can be split into two summations of length $N/2$,

$$\begin{aligned} X[m] &= \sum_{n=0}^{N/2-1} x[2n] W_N^{m2n} + \sum_{n=0}^{N/2-1} x[2n+1] W_N^{m(2n+1)} \\ &= \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{mn} + W_N^m \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{mn}, \end{aligned} \tag{2.4}$$

where the identity $W_N^2 = W_{N/2}$ is used.

Now observe that the upper-half of $X[m]$ can be obtained from the bottom half, giving

$$\begin{aligned} X[m+N/2] &= \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{(m+N/2)n} \\ &+ W_N^{(m+N/2)} \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{(m+N/2)n} \\ &= \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{mn} - W_N^m \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{mn}. \end{aligned} \tag{2.5}$$

This holds because

$$W_{N/2}^{n(m+N/2)} = W_{N/2}^{nm} W_{N/2}^{nN/2} = W_{N/2}^{nm}, \tag{2.6}$$

and

$$W_N^{m+N/2} = W_N^m W_N^{N/2} = -W_N^m. \tag{2.7}$$

By comparing equations (2.4) and (2.5), it can be sees that $X[m]$ and $X[m + N/2]$, for $0 \leq m < N/2$, only differ by a sign. Therefore, the two halves of the DFT

result can be produced by using the same operands. These operands are the two summations in (2.4) and (2.5); they are two $N/2$-point DFTs.

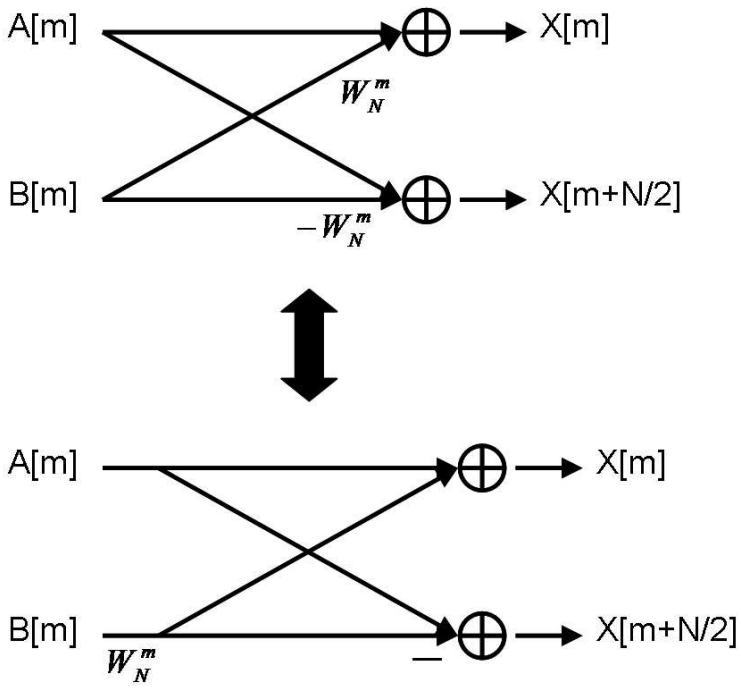


Figure 2.2: Simplification of radix-2 butterfly twiddle factor multiplications.

In addition to the sharing of the two $N/2$-point DFT outputs, there is a further simplification that can be made. Let us name the two $N/2$-point DFT outputs $A[m]$ and $B[m]$, respectively, of length $N/2$. Then (2.4) and (2.5) can be posed as

$$X[m] = A[m] + W_N^m B[m] \tag{2.8}$$

$$X[m + N/2] = A[m] - W_N^m B[m], \tag{2.9}$$

where $0 \leq m < N/2$. Observe that the m-th and $(m + N/2)$-th members of $X[m]$ can be generated by the circuit shown at the top of Figure 2.2.

The simplification results because the circuit transformation illustrated in Figure 2.2 can be performed. Note that the converted circuit consists of a single twiddle factor multiplication on $B[m]$, followed by a radix-2 butterfly. These circuits are

equivalent, yet the transformed circuit reduces the number of twiddle factor multiplications by half.

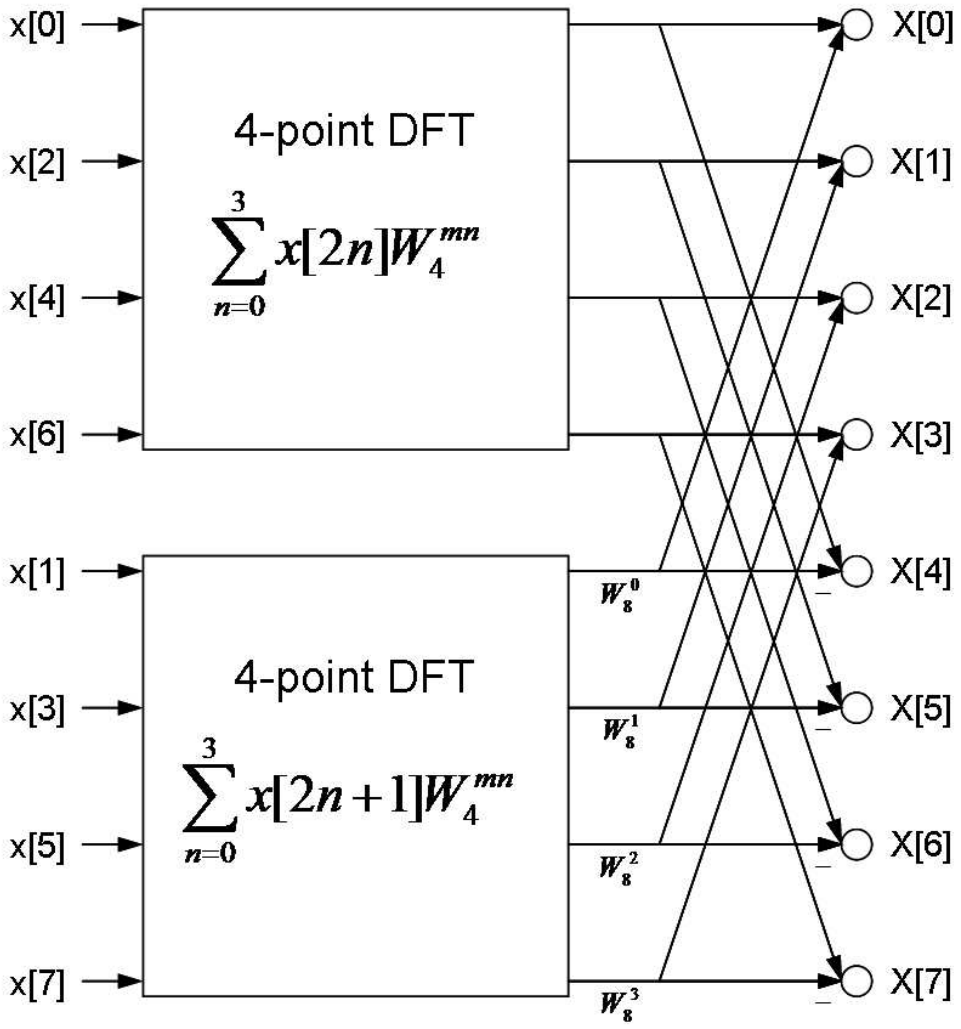


Figure 2.3: Data flow graph for 8-point DFT using 4-point DFTs.

If the simplifications just discussed are applied to an 8-point DFT, then Figure 2.3 shows its resulting data flow graph. Note the two 4-point DFT blocks, and that the outputs of the blocks are shared as operands for the bottom and top halves of the output. $X[m]$ is produced by executing a series of four radix-2 butterflies on the

outputs of two 4-point DFT blocks; the outputs of the DFT block corresponding to the $B[m]$ sequence are also modified by twiddle factor multiplications.

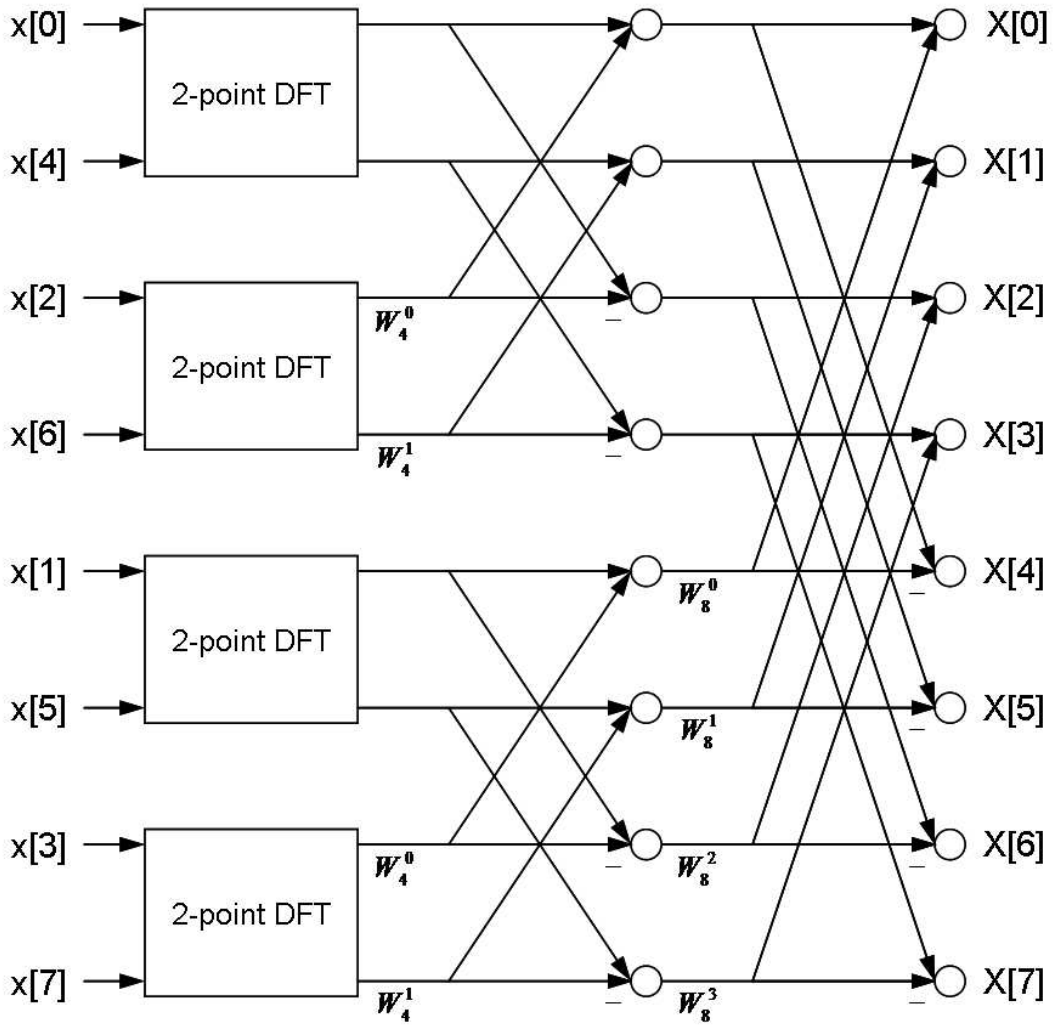


Figure 2.4: Data flow graph for 8-point DFT using 2-point DFTs.

The computation of our transformed $N-$point DFT can be further simplified. The simplifications presented in the previous discussion can be recursively applied to the two $N/2$-point DFTs, $A[m]$ and $B[m]$. For our 8-point DFT example in Figure 2.3, this will yield the data flow graph shown in Figure 2.4. The outputs of the two

4-point DFTs are now also computed with twiddle factor multiplications followed by radix-2 butterflies; the inputs to the multiplication/butterfly combo are generated from four 2-point DFTs.

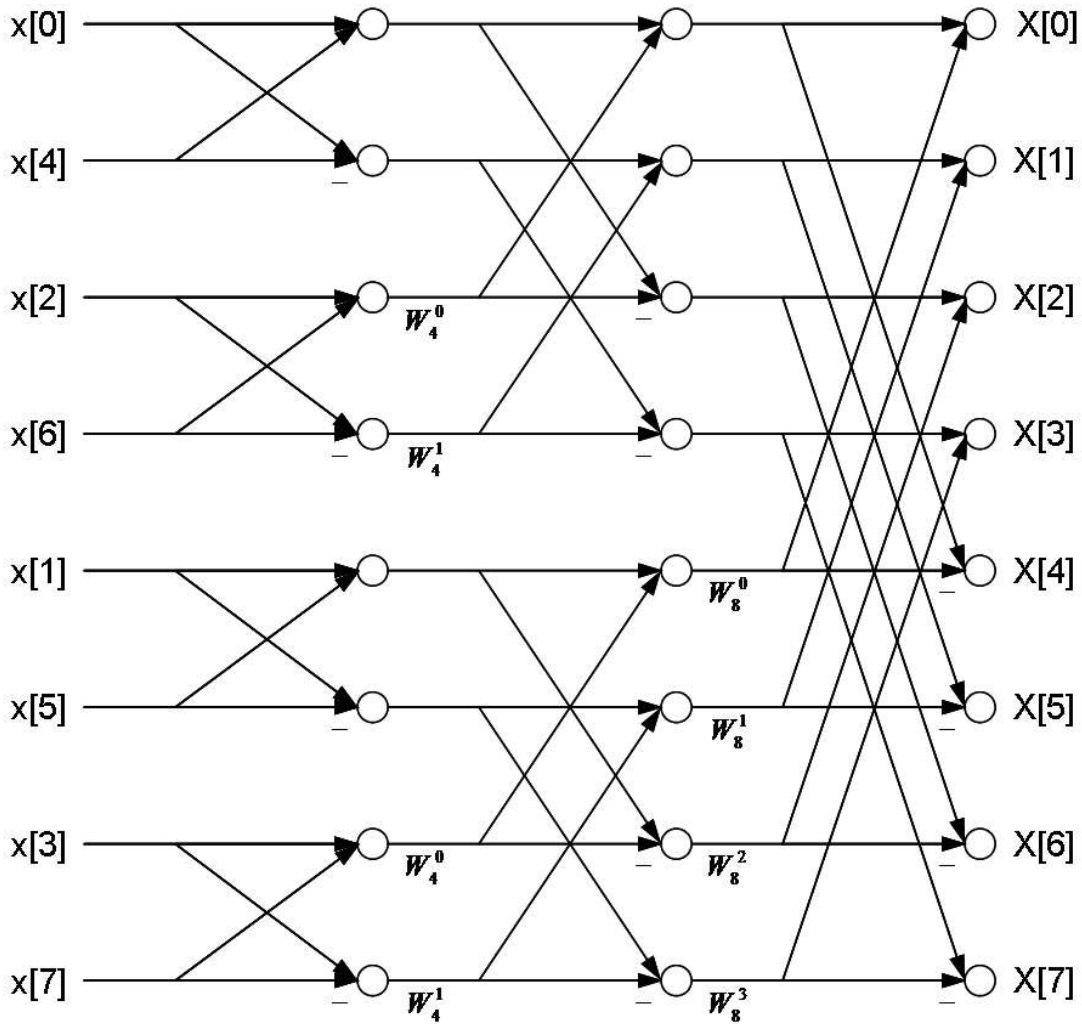


Figure 2.5: Data flow graph for 8-point DIT FFT using radix-2 butterflies.

The objective of the radix-2 DIT FFT algorithm is to reduce the DFT computation to a series of radix-2 butterfly operations and twiddle factor multiplies. Each radix-2 butterfly computes a 2-point DFT. The 2-point DFT blocks in Figure 2.4 can

therefore be replaced by radix-2 butterflies, finally giving Figure 2.5; this result is the complete data flow graph for an 8-point DIT FFT.

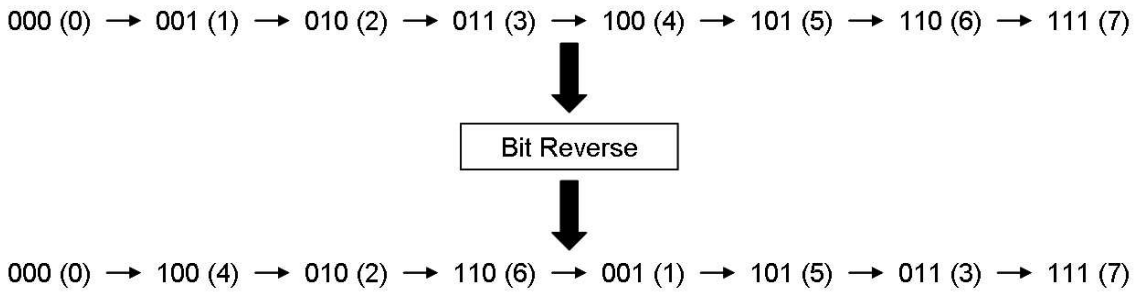


Figure 2.6: Bit-reverse operation on address sequence.

Because the input sequence, $x[n]$, has been recursively decimated in time, the input order has been scrambled. This can be seen in Figure 2.5. For FFTs of a power of two radix, the data can be reordered by a simple *bit-reverse-copy*. This consists of copying the input sequence into a new sequence where the elements have been assigned to bit-reversed addresses. Figure 2.6 shows how the bit-reverse operation is performed on the addresses of the input sequence; after the bit-reverse copy, the data can be presented to the radix-2 DIT FFT in the correct order.

The reordering of data is a side-effect of all FFT algorithms (depending on the algorithm it can be either the input or the output that is scrambled). The scrambling effect is why it is not precise to call the FFT an equivalent operator of the DFT. But, because it is usually simple to reorder the data, most people ignore this subtle difference.

Despite the need to reorder the data, the computational savings of the FFT are considerable, compared to the DFT. In our 8-point DFT example, the resulting FFT requires 24 complex additions and 8 complex multiplications. The equivalent

```
1.   ITERATIVE_DIT_FFT(x,X) {
2.     X = x; /* copy x */
3.     n = length(x);
4.     bit_reverse(X); /* reorder X */
5.     for(s = 1; s <= log(n); s++) { /* outer loop */
6.       m = 2^s;
7.       wm = cos(2*pi/m)-sqrt(-1)*sin(2*pi/m);
8.       for(k = 0; k < n; k += m) { /* inner loop */
9.         w = 1; /* twiddle factor */
10.        for(j = 0; j < m/2; j++) { /* execute butterflies */
11.          t = X[k+j];
12.          u = w*X[k+j+m/2];
13.          X[k+j] = t + u;
14.          X[k+j+m/2] = t - u;
15.          w = w*wm; /* compute next twiddle factor */
16.        }
17.      }
18.    }
19.  }
```

Figure 2.7: Pseudocode for the sequential, iterative radix-2 DIT FFT

DFT would require 56 complex additions and 64 complex multiplications. Except for the data shuffling, the FFT and DFT results are *identical*.

Figure 2.7 presents pseudocode describing the radix-2 DIT FFT. Though the DIT FFT was derived recursively, recursive algorithms are difficult to map to hardware. Therefore, the pseudocode describes an iterative algorithm. Note that the outer loop iterates $\log(N)$ times, and the inner loop iterates N times. Also, lines 11-14 perform the twiddle factor multiplication and the radix-2 butterfly. Line 4 completes a bit-reverse copy of the input data sequence.

2.2.2 The Decimation in Frequency Radix-2 FFT

In Section 2.2.1 the FFT was derived by recursively decimating the input sequence in time. An alternative approach is to instead decimate the output sequence

in frequency. This leads us to the radix-2 decimation in frequency (DIF) FFT. The DIF FFT produces a computation that accepts the input in order, and produces a shuffled output.

The derivation of the radix-2 DIF FFT is not as intuitive in comparison to that of the DIT FFT. But, in many hardware applications of the FFT the input data is presented serially. Because of this, a full data sequence must be buffered before executing the reorder. This holds for both the DIF and DIT FFT. Nevertheless, some FFT applications can use the FFT output without reordering. Such a case would allow fewer resources to be used for the DIF FFT, since the reorder buffering would be unnecessary. Because of this, it is therefore more widely used for hardware applications.

Assume the length N of the sequence $x[n]$ is a power of two. Its DFT, $X[m]$, can be split into two sequences of length $N/2$, where one sequence contains all the even elements, and the other the odd elements. The even elements of $X[m]$ can be computed using Equation (2.1), giving

$$\begin{aligned} X[2m] &= \sum_{n=0}^{N-1} x[n]W_N^{2mn} \\ &= \sum_{n=0}^{N/2-1} x[n]W_N^{2mn} + \sum_{n=N/2}^{N-1} x[n]W_N^{2mn} \\ &= \sum_{n=0}^{N/2-1} x[n]W_N^{2mn} + \sum_{n=0}^{N/2-1} x[n+N/2]W_N^{2m(n+N/2)}. \end{aligned} \tag{2.10}$$

Also, because W_N^{2mn} is periodic, the following is obtained

$$W_N^{2m(n+N/2)} = W_N^{2mn}W_N^{mN} = W_N^{2mn} = W_{N/2}^{mn}. \tag{2.11}$$

Using this observation, and combining the two summations of (2.10), results in

$$X[2m] = \sum_{n=0}^{N/2-1} (x[n] + x[n+N/2])W_{N/2}^{mn}. \tag{2.12}$$

Equation (2.12) is the $(N/2)$-point DFT of the sequence obtained by performing a vector summation of the first and second halves of $x[n]$.

The odd elements of $X[m]$ can be obtained as follows:

$$X[2m+1] = \sum_{n=0}^{N-1} x[n]W_N^{(2m+1)n}$$

$$= \sum_{n=0}^{N/2-1} x[n]W_N^{(2m+1)n} + \sum_{n=N/2}^{N-1} x[n]W_N^{(2m+1)n}. \quad (2.13)$$

The second summation of (2.13) can be rearranged as

$$\begin{aligned} \sum_{n=N/2}^{N-1} x[n]W_N^{(2m+1)n} &= \sum_{n=0}^{N/2-1} (x[n]+N/2)W_N^{(2m+1)(n+N/2)} \\ &= W_N^{(2m+1)N/2} \sum_{n=0}^{N/2-1} (x[n+N/2])W_N^{(2m+1)n}, \end{aligned} \quad (2.14)$$

and because $W_N^{(2m+1)N/2} = W_N^{mN}W_N^{N/2} = e^{-j2\pi m}e^{-j\pi} = -1$, Equation (2.14) becomes

$$\sum_{n=N/2}^{N-1} x[n]W_N^{(2m+1)n} = -\sum_{n=0}^{N/2-1} (x[n+N/2])W_N^{(2m+1)n}. \quad (2.15)$$

By substituting (2.15) into (2.13) and combining the summations, the odd elements of $X[m]$ can be expressed as

$$\begin{aligned} X[2m+1] &= \sum_{n=0}^{N/2-1} (x[n]-x[n+N/2])W_N^{(2m+1)n} \\ &= \sum_{n=0}^{N/2-1} (x[n]-x[n+N/2])W_{N/2}^{mn}W_N^n, \end{aligned} \quad (2.16)$$

since $W_N^2 = W_{N/2}$. Equation (2.16) is the $(N/2)$-point DFT of the sequence obtained by performing a vector subtraction of the second half of $x[n]$ from the first half, and multiplying the result by W_N^n.

Just as with the DIT FFT algorithm presented in Section 2.2.1, I have obtained a simplified N-point DFT, where $X[m]$ is formed from two $N/2$-point DFTs. The inputs to the DFTs are $N/2$-point sequences formed by vector operations on the first and second halves of $x[n]$, and also a twiddle factor multiplication.

The same simplifications described above can be applied recursively to $X[2m]$ and $X[2m+1]$. This is done until 2-point DFTs are being computed. At this point the blocks are replaced with the radix-2 butterfly. Because the sequence is a power of two length N, $\log_2 N$ recursions will be required.

Figure 2.8 shows the data flow diagram for a 16-point radix-2 DIF FFT. Note that there are $N = 16$ rows and $\log_2 N = 4$ columns of operations. Instead of the input being scrambled, it is now the output that must be reordered. This is from

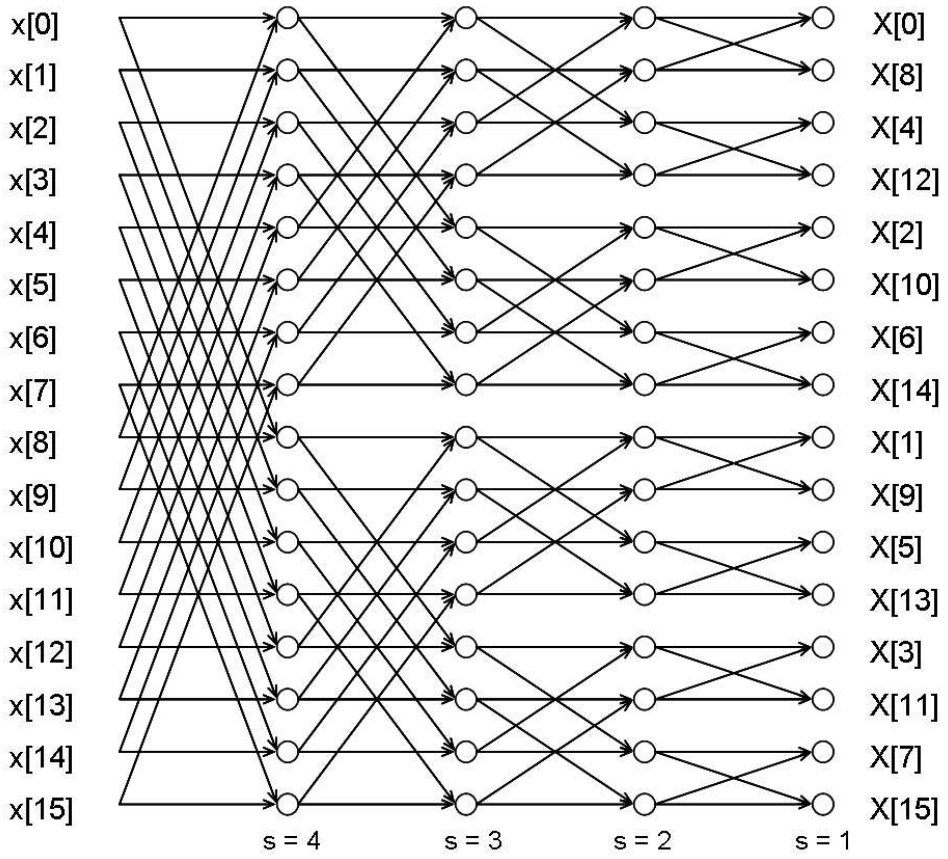


Figure 2.8: Data flow graph for a 16-point DIF radix-2 FFT.

decimating the output in frequency. Note that in Figure 2.8, there are implied twiddle factor multiplications between butterfly columns.

When implemented as a sequential program, the radix-2 FFT can be described in either a recursive or iterative algorithm. The recursive algorithm fits the preceding derivation better, but does not easily map to hardware. Figure 2.7 is the pseudocode for such an FFT algorithm. The outer loop loops $logN$ times, and the inner loop does so $N/2$ times. Lines 10-13 perform the radix-2 butterfly and the twiddle factor multiplication. The `bit_reverse(X)` procedure on line 18 is required for unscrambling the FFT result.

```
1.   ITERATIVE_DIF_FFT(x,X) {
2.     X = x; /* copy x */
3.     n = length(x);
4.     for(s = log(n); s >= 1; s--) { /* outer loop */
5.       m = 2^s;
6.       wm = cos(2*pi/m)-sqrt(-1)*sin(2*pi/m);
7.       for(k = 0; k < n; k += m) { /* inner loop */
8.         w = 1; /* twiddle factor */
9.         for(j = 0; j < m/2; j++) {
10.          t = X[k+j];
11.          u = X[k+j+m/2];
12.          X[k+j] = t + u;
13.          X[k+j+m/2] = w*(t - u);
14.          w = w*wm; /* compute next twiddle factor */
15.        }
16.      }
17.    }
18.    bit_reverse(X);
19.  }
```

Figure 2.9: Pseudocode for the sequential, iterative radix-2 DIF FFT.

2.2.3 The Decimation in Frequency Radix-4 FFT

In this section I will derive the radix-4 DIF FFT. I will not do so for the radix-4 DIT FFT, because the derivation is very similar to the radix-2 DIT FFT, and uses some of the same as for the radix-4 DIF FFT. It is therefore left as an exercise for the interested reader. The reason I choose to derive the radix-4 DIF FFT instead of the DIT is because the formulation of the HAPFFT is obtained in a similar manner as that for the radix-4 DIF FFT. Thus, an understanding of this section will aid in deriving the HAPFFT.

The radix-4 DIF algorithm is similar to the radix-2 DIF algorithm. The derivation uses the same approach, by decimating the DFT output in the frequency domain. They differ in that the atomic computational unit is a radix-4 butterfly, as introduced in Figure 2.1. The advantage of using the radix-4 butterfly is that it can be computed without any twiddle factor multiplications, while the total number of butterflies required is half that of the radix-2 algorithm. Thus, the total number of complex twiddle factor multiplications for a radix-4 FFT is half that of the radix-2 FFT. The disadvantage is that twice the number of complex additions are needed, and the size of the input data set is limited to a power of four length. Nevertheless, when using fixed-point computer arithmetic, because complex multiplications are very often more expensive than complex additions, a radix-4 FFT may be cheaper to implement. As an aside, in section 3.1 I will discuss the radix-2^2 FFT architecture. It emulates the radix-4 FFT, but can do so with fewer complex additions, resulting in a very efficient architecture.

The radix-4 DIF FFT algorithm is derived in the same manner as the DIF radix-2 algorithm. But instead of decimating the DFT output into odd and even halves, the radix-4 algorithm decimates it into quarters. For a power of four data set, $x[n]$, of length N, I decimate its DFT, $X[m]$, into the $X[4m]$, $X[4m+1]$, $X[4m+2]$, and $X[4m+3]$ output sequences. For the $X[4m]$-th quarter I get

$$X[4m] = \sum_{n=0}^{N-1} x[n] W_N^{4mn}$$

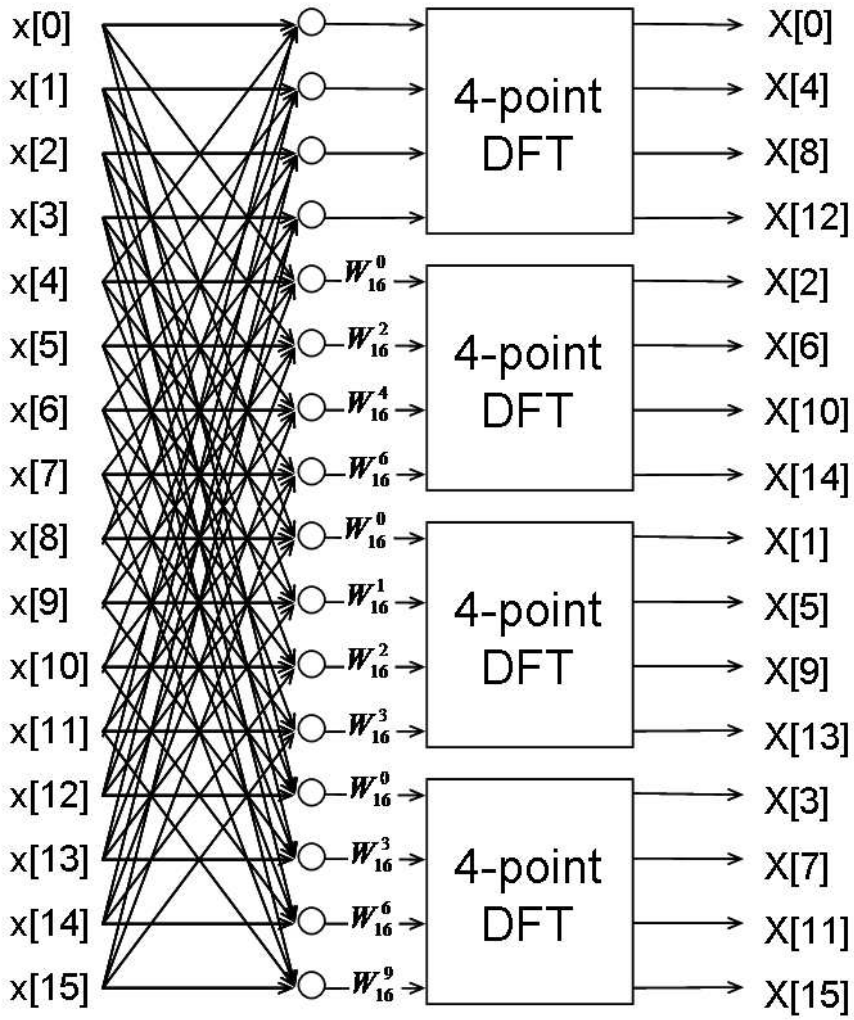


Figure 2.10: Data flow graph for 16-point DFT using 4-point DFTs.

$$
\begin{aligned}
&= \sum_{n=0}^{N/4-1} x[n]W_N^{4mn} + \sum_{n=N/4}^{N/2-1} x[n]W_N^{4mn} \\
&\quad + \sum_{n=N/2}^{3N/4-1} x[n]W_N^{4mn} + \sum_{n=3N/4}^{N-1} x[n]W_N^{4mn} \\
&= \sum_{n=0}^{N/4-1} x[n]W_N^{4mn} \\
&\quad + \sum_{n=0}^{N/4-1} x[n+N/4]W_N^{4m(n+N/4)}
\end{aligned}
$$

$$+ \sum_{n=0}^{N/4-1} x[n+N/2]W_N^{4m(n+N/2)} + \sum_{n=0}^{N/4-1} x[n+3N/4]W_N^{4m(n+3N/4)}. \quad (2.17)$$

$X[4m]$ can be further simplified by observing that

$$\begin{aligned} W_N^{4mn} &= W_{N/4}^{mn}, \\ W_N^{4m(n+N/4)} &= W_N^{4mn}W_N^{mN} = W_{N/4}^{mn}, \\ W_N^{4m(n+N/2)} &= W_N^{4mn}W_N^{2mN} = W_{N/4}^{mn}, \\ W_N^{4m(n+3N/4)} &= W_N^{4mn}W_N^{3mN} = W_{N/4}^{mn}. \end{aligned}$$

Using these twiddle factors, and combining the summations in (2.17), the result is

$$X[4m] = \sum_{n=0}^{N/4-1} (x[n] + x[n+N/4] + x[n+N/2] + x[n+3N/4])W_{N/4}^{mn}. \quad (2.18)$$

The $X[4m+1]$-th sequence is computed in a similar fashion, obtaining

$$\begin{aligned} X[4m+1] &= \sum_{n=0}^{N-1} x[n]W_N^{(4m+1)n} \\ &= \sum_{n=0}^{N/4-1} x[n]W_N^{(4m+1)n} + \sum_{n=N/4}^{N/2-1} x[n]W_N^{(4m+1)n} \\ &\quad + \sum_{n=N/2}^{3N/4-1} x[n]W_N^{(4m+1)n} + \sum_{n=3N/4}^{N-1} x[n]W_N^{(4m+1)n} \\ &= \sum_{n=0}^{N/4-1} x[n]W_N^{(4m+1)n} \\ &\quad + \sum_{n=0}^{N/4-1} x[n+N/4]W_N^{(4m+1)(n+N/4)} \\ &\quad + \sum_{n=0}^{N/4-1} x[n+N/2]W_N^{(4m+1)(n+N/2)} \\ &\quad + \sum_{n=0}^{N/4-1} x[n+3N/4]W_N^{(4m+1)(n+3N/4)}. \end{aligned} \quad (2.19)$$

$X[4m+1]$ can also be simplified by observing that

$$W_N^{(4m+1)n} = W_{N/4}^{mn}W_N^n,$$

$$
\begin{aligned}
W_N^{(4m+1)(n+N/4)} &= W_{N/4}^{mn} W_N^{mN} W_N^n W_N^{N/4} = -jW_{N/4}^{mn} W_N^n, \\
W_N^{(4m+1)(n+N/2)} &= W_{N/4}^{mn} W_N^{2mN} W_N^n W_N^{N/2} = -W_{N/4}^{mn} W_N^n, \\
W_N^{(4m+1)(n+3N/4)} &= W_{N/4}^{mn} W_N^{3mN} W_N^n W_N^{N/2} = jW_{N/4}^{mn} W_N^n.
\end{aligned}
$$

Thus, Equation (2.19), by combining the summations, can be rewritten as

$$
\begin{aligned}
X[4m+1] &= \sum_{n=0}^{N/4-1} (x[n] - jx[n+N/4] \\
&\quad -x[n+N/2] + jx[n+3N/4]) W_{N/4}^{mn} W_N^n. \qquad (2.20)
\end{aligned}
$$

The $X[4m+2]$-th and $X[4m+3]$-th sequences can in likewise manner be found, giving us

$$
\begin{aligned}
X[4m+2] &= \sum_{n=0}^{N/4-1} (x[n] - x[n+N/4] \\
&\quad +x[n+N/2] - x[n+3N/4]) W_{N/4}^{mn} W_N^{2n}, \qquad (2.21) \\
X[4m+3] &= \sum_{n=0}^{N/4-1} (x[n] + jx[n+N/4] \\
&\quad -x[n+N/2] - jx[n+3N/4]) W_{N/4}^{mn} W_N^{3n}. \qquad (2.22)
\end{aligned}
$$

Equations (2.18), (2.20), (2.21), and (2.22), are each $N/4$-point DFTs. The inputs for the DFTs are formed by computing $N/4$ radix-4 butterflies.

Figure 2.10 shows the dfg for a 16-point DFT after incorporating the simplifications derived above. The inputs to the DFTs are the four sequences of length $N/4 = 4$, computed according to Equations (2.18), (2.20), (2.21), and (2.22).

Each of the $N/4$-point DFTs found in (2.18), (2.20), (2.21), and (2.22) can be recursively simplified using the same methods. The recursion is executed until 4-point DFT blocks are generated. At this point the 4-point DFTs can be replaced by the equivalent radix-4 butterfly.

If these simplifications are applied to the example in in Figure 2.10, the result is found in Figure 2.11. Note that the atomic computational unit is the radix-4 butterfly. This is the complete radix-4 DIF FFT dfg for a 16-point FFT. Comparing it to the 16-point radix-2 DIF FFT in Figure 2.5, it can be sees that the total number of butterflies and twiddle factor multiplications is greatly reduced. In addition, notice

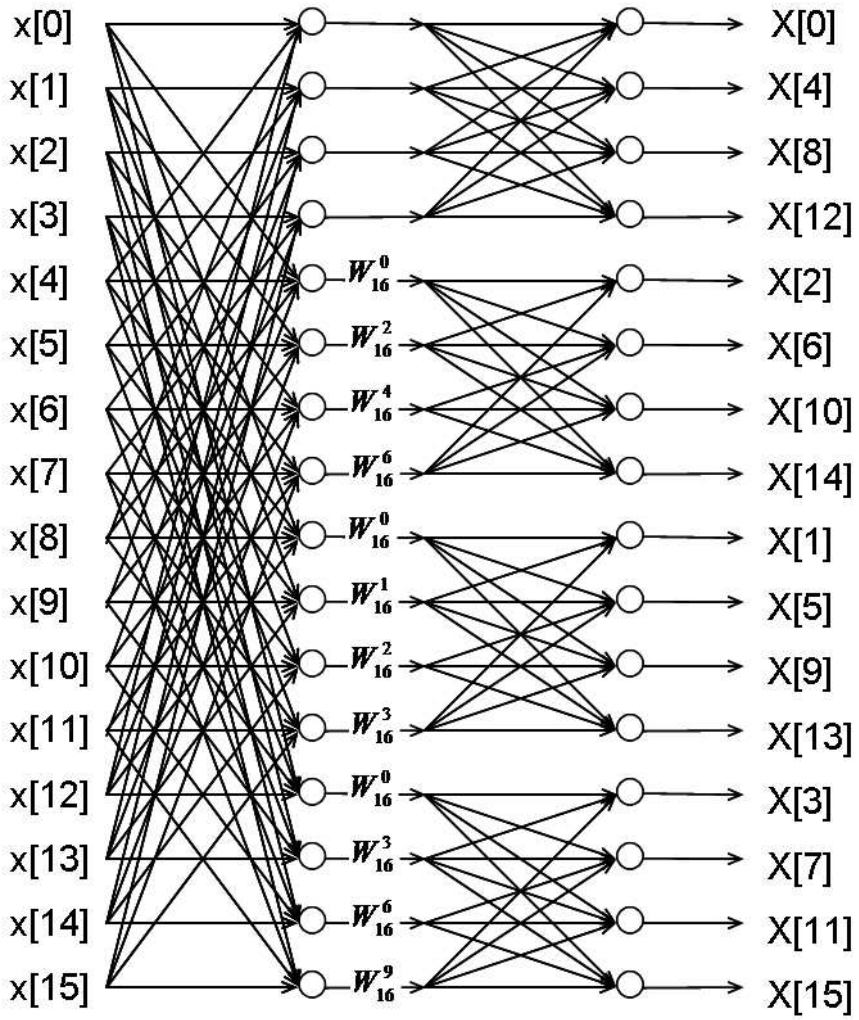


Figure 2.11: Data flow graph for 16-point DIF FFT using radix-4 butterflies.

that the output is again scrambled as a result of the frequency decimation. These can be reordered using the address bit-reverse method.

2.3 The Mixed-Radix FFT

All the FFT algorithms derived up to this point have been homogeneous, composed of a single type of butterfly. While this provides for simple design and algorithm derivation, it limits the size of sequences to which the FFT can be applied. They must be a power of a size, where a is the radix of the butterfly.

One may argue that this is not a problem. If a sequence is not of a proper length, then just zero-pad the sequence until it is of the proper power of a in length. While this is commonly done in practice, in some applications it is not desirable. The mixed-radix FFT algorithm allows an FFT of any non-prime size to be computed by factoring the FFT into a sequence of smaller FFTs.

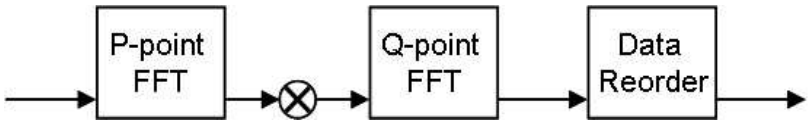


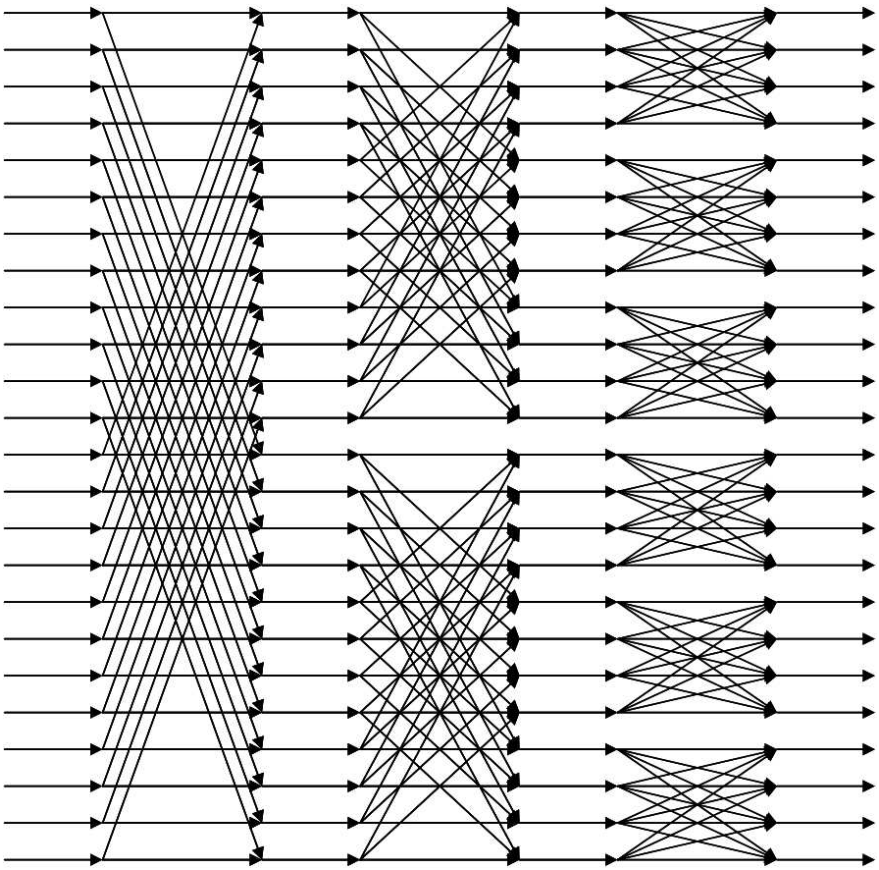
Figure 2.12: An $N = PQ$-point mixed-radix FFT.

For a data set of a non-prime number size, N, if it has two factors, P and Q, then the FFT of N can be computed by instead computing a P-point FFT, and then a Q-point FFT (or vice-versa). Figure 2.12 shows this process.

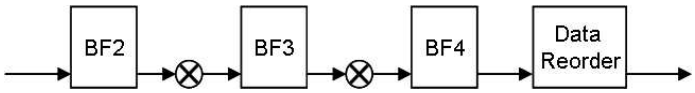
Each FFT block, if it is not a prime number size, can likewise be factored into smaller blocks. This can continue until all blocks are of some prime-number in size. The radix-2 FFT algorithm is a special case of the mixed radix algorithm; it is computed using $\log_2 N$ 2-point FFTs.

Figure 2.13 shows a mixed-radix FFT example. It shows a dfg for a 24-point FFT. The FFT is factored into three stages: a 2-point FFT, then a 3-point FFT, and finally a 4-point FFT. Note that twiddle-factor multiplies are required in between each stage. These can be computed using the same techniques found in the radix-2 and radix-4 DIF derivations.

When using non-power of two sized butterflies, the data reordering becomes more complex. It is no longer just a simple case of reversing the address bits. In practice, for the mixed-radix FFT, the easiest way to reorder the data is to reorder after every FFT stage, rather than all at once at the end.



(a) dfg



(b) diagram

Figure 2.13: Mixed-radix 24-point FFT.

The mixed-radix FFT will be of particular importance later. It is related to the HAPFFT presented in chapter 4.

For data sequences of a prime number size there also exist FFT algorithms. The most common are grouped into a category known as the convolution-based FFT. Though much more expensive than the standard FFT, they are normally faster than explicitly computing the DFT.

Chapter 3

High Performance FFT Computations

In Chapter 2, I demonstrated that the FFT is a more computationally efficient means of computing the DFT. Despite this, the FFT is still a relatively expensive and complex operation. This is caused by the need to operate on complex numbers, access and manipulate often large blocks of memory, and control complicated movements of data.

One of the most thoroughly studied areas of FFT research has to do with techniques for increasing FFT performance. When I indicate performance I am referring to the data throughput, the average number of samples-per-second that a particular FFT implementation can consume, denoted by $\sigma_{pipeline}$. Other criteria may also be of equal or greater importance, such as computational latency, resource requirements, or power.

This chapter will provide background on some of the techniques used for high-performance implementations of the FFT. Some of these are architectures intended for custom hardware (such as VLSI or FPGA blocks). Others are intended for parallel computing environments. Also, in Section 1.2, I reviewed the state-of-the-art in the field of hardware parallel FFTs.

Given a particular computational algorithm and problem size, there are two ways that it can be executed faster. Either complete each algorithmic step in less time, or execute the steps concurrently. The sections in this chapter will focus on the second technique: increasing the computational concurrency of the FFT. First, Section 3.1 will discuss hardware pipelining, and then Section 3.2 will introduce the parallel FFT, and review two common algorithms for its computation.

3.1 Hardware Pipelined FFT Architectures

Hardware pipelining is an important and effective technique used to increase computational concurrency. Pipelining is best illustrated by using an assembly line analogy. Using the example of an automotive assembly line, at each step in the line a given assembly step is performed. At one step the chassis is welded together, at another the engine is mounted, and a subsequent step will install the wheels. The assembly line could be split up into an arbitrary number of steps. If there are N steps involved in the assembly, and each assembly line step is always doing useful work, then it can be said that the assembly line can assemble N automobiles in parallel. This holds even though only one car exits the factory at a time.

Pipelining in custom hardware is based on a similar concept. For a given computational algorithm that requires N steps to complete, a hardware functional unit could be constructed for each step. Then, if the algorithm is suitably parallel, the pipeline can complete N times more work than an implementation that only performs a single computation at a time.

For pipelining to be effective, a few assumptions must be made: there is enough data to feed the pipeline a constant stream of data, the dependencies between data points is of a nature such that they won't interfere with the correct execution of any given pipeline stage, and there are enough hardware resources so that no stage need share functional units with another. If any of these don't hold, then the N pipeline stages may have to execute less than N computations at a time.

I will make the assumption that all pipelines discussed in this section have transitions which are *synchronous* to some clock. This means that each pipeline stage will consume and produce a datum at either the rising or falling edge of a common clock signal.

Digital signal processing tends to fit the pipelining paradigm well. Many signal processing algorithms consist of taking a block of data, and executing a number of steps on it. Often the data is a constant stream, and the execution steps are independent of each other. If hardware resources are not an issue, it is quite easy to construct a high throughput pipeline, as illustrated in Figure 3.1. For example, since

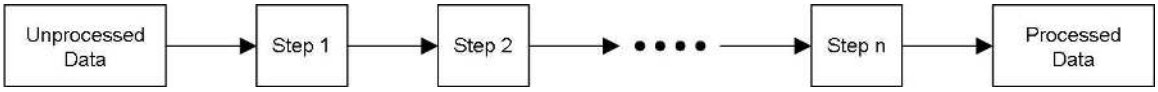


Figure 3.1: A typical DSP processing pipeline.

the DFT is a building block for many DSP algorithms, one or more FFTs may form blocks in such a pipeline.

The performance of a hardware pipeline can be quantified by using the pipeline throughput, $\sigma_{pipeline}$. An important parameter that determines the pipeline throughput is the *data introduction interval*, denoted as δ_0 [14]. Though having no effect on the throughput, another important parameter which places constraints on δ_0 is the *pipeline latency*. Pipeline latency is defined as the number of clock cycles that must occur after the start of a computation is begun, until a result appears. By definition, $\delta_0 < latency$ will always hold for a pipelined circuit. If $\delta_0 = latency$, then a new computation is initiated only after the previous one has completed, and therefore there is no pipelining. For a pipeline with $\delta_0 = 0$, the data introduction interval is non-existent, and thus a computation is being initiated at every clock cycle. Such a circuit is fully pipelined, and pipelining can no longer be used for increasing computational concurrency.

Referring back to Figure 3.1, depending on a given computational step in the DSP pipeline paradigm, the step itself may be able to be further subdivided into pipeline stages. This would enable an increase in total computational concurrency. For the FFT architectures discussed in this chapter, there exist a number of methods for their pipelining. Section 3.1.1 introduces some of the more common types.

3.1.1 A Taxonomy of FFT Architectures for Custom Hardware

Chapter 2 introduced a number of FFT algorithms, as well as pseudocode that can be used in a practical software implementation of these algorithms. However, the

FFT is often used in high-performance systems where the use of a software FFT implementation, running on a general-purpose microprocessor, is inadequate. At times this can be resolved by using a DSP processor. Alternatively, a parallel FFT algorithm (to be discussed in Section 3.2) can be implemented using a parallel computing environment.

In many applications even these approaches will not meet design requirements. Either they are too expensive in terms of power and size, or their performance may still be insufficient. In such cases a custom hardware FFT can often resolve the problem.

Hardware FFT architectures come in many flavors, depending on the criteria of the application. Some architectures provide unusually low power demands, others use almost trivial amounts of hardware, while some give exceptional data throughput. I will focus here on architectures which are targeted for high-throughput applications.

Hardware FFT modules differ in which FFT algorithm is used (radix-2 DIF, radix-4 DIT, etc.), and in how the algorithm is mapped to hardware. The architectures can be categorized into *bursty*, and *streaming* architectures.

Bursty architectures have a computational latency longer than the length of the input data set. This means that after a data set is input, a delay must be included before the next subsequent data set can be input. In other words, on average, $\delta_0 > 0$.

In contrast, the streaming architecture is capable of accepting one or more data points every clock cycle. Bursty architectures are used mostly for low-power and/or low-resource applications. Streaming architectures are found more often in high-throughput applications.

Bursty Architectures

Though I shall mostly address streaming architectures, it is useful to first briefly study a common bursty architecture, called the in-place FFT architecture, shown in Figure 3.2. The purpose is to compare and contrast the bursty computational paradigm with the streaming pipeline.

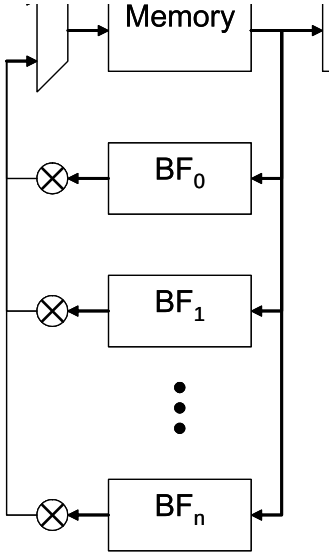


Figure 3.2: Diagram for a general in-place FFT architecture.

The in-place architecture uses the most obvious mapping of the algorithms shown in Figure 2.7 and 2.9. Comparing Figure 3.2 to the algorithm presented in Figure 2.9, it can be seen that the memory block is equivalent to the `X` array, with the input data stream corresponding to the `x` array. To implement the outer and inner loops the contents of the memory block are repeatedly sent to a set of butterfly and twiddle factor multiplier functional units. Upon termination of the computation, the data is reordered, and output.

It should be apparent that performance trade-offs can be easily made. The latency, power, throughput and resource requirements can be changed by adding or subtracting to the total number of functional units.

The control of the in-place FFT architecture tends to be complicated; there is a lot of resource sharing, and the data must be carefully directed to the correct functional units. Also, though the functional units will, in general, be identical, the inputs to the twiddle factor multipliers vary from unit to unit, and from iteration to iteration.

The in-place architecture can be modified to allow streaming behavior. This can be done by duplicating the memory and functional units. If enough functional units are provided, it is possible to have one core computing while the other is inputting, and vice-versa. For example, Xilinx, Inc., provides a streaming radix-4 FFT IP core based on such a scheme [28].

Streaming Architectures

While there are a number of different streaming FFT architectures [27, 7, 10, 21, 3, 4], most share a single characteristic that differentiates them from bursty architectures, namely: rather than continually reading and writing the data to the same location in memory, the data instead moves through a pipeline. Therefore, these architectures are sometimes called *pipelined FFT* architectures.

Pipelined FFTs typically have simpler control than their bursty counterparts. Nevertheless, they will also have higher resource requirements. Also, the mapping from the abstract FFT algorithms to the hardware is not quite so obvious.

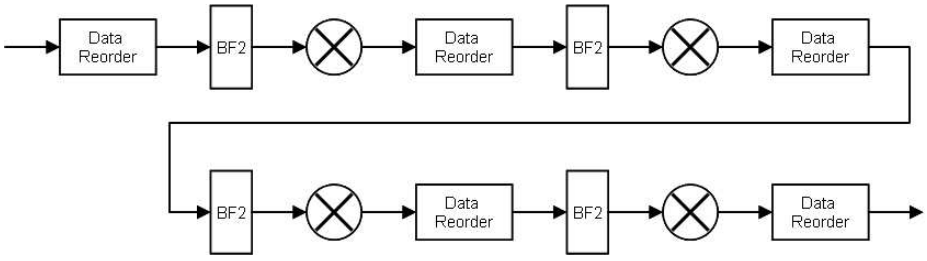


Figure 3.3: Pipelined FFT DFG for Figure 2.8.

I will discuss two families of pipelined FFT architectures: the *delay feedback* and the *delay commutator* architectures. The families differ in the way that they present inputs to the butterflies. Figure 2.8 can be used to understand this difference. The input data stream will typically provide a single data point every clock cycle. Assuming that the butterflies will be executed starting at the top of the left column

of Figure 2.8, executing each butterfly from top to bottom, and then proceed with the next column of butterflies, and so on. If $x[0]$ arrives in the first clock cycle, the first butterfly cannot be immediately executed; the other butterfly operand, $x[8]$, will not arrive for seven more clock cycles. Likewise, for each column of our dfg, the same problem occurs as the data proceeds down the pipeline. The data must be reordered before every butterfly.

Figure 3.3 shows how the 16-point DIF FFT example could be mapped to hardware in such a way that the data is presented correctly to the butterflies. Figure 3.3 is for a radix-2 DIF algorithm, though a similar block diagram applies to other radices and algorithms.

Delay Feedback Pipelined Architectures

The delay feedback architectures reorder the input by first accepting part of the data stream into the butterfly elements, but instead of computing on the block, it is redirected to a feedback delay line. By the time the data appears again at the input of the butterfly the other inputs of the butterfly will also be ready.

Figure 3.4 shows how a 16-point radix-2 DIF would be implemented using a single delay feedback for each butterfly. Looking again at Figure 2.8, each column of the dfg corresponds to one of the butterfly elements. The feedback delay, λ, for each butterfly is given as

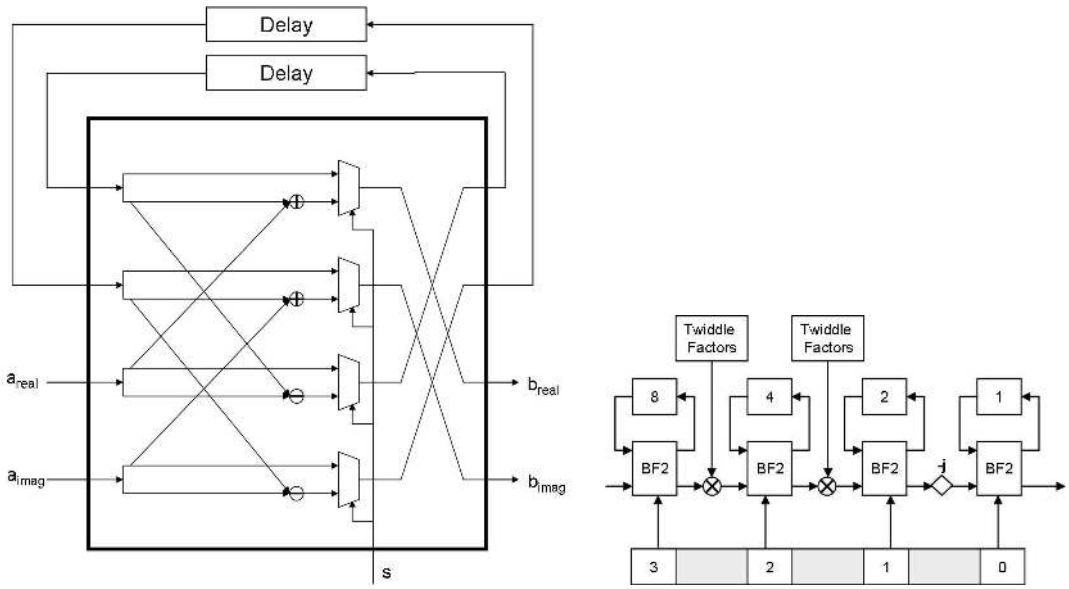
$$\lambda = 2^s/2 \tag{3.1}$$

where s corresponds to the column labeling in Figure 2.8.

Figure 3.4 is known as the radix-2 single delay feedback (R2SDF) architecture. There are a number of variations of this same theme. The most common are described in Figure 3.5, each computing a 64-point DIF FFT.

- **R2SDF [27]**

  An efficient implementation of the radix-2 FFT algorithm. It Requires $N-1$ memory elements for the delay lines, $2\log_2 N$ complex additions, and $\log_2 N - 2$ complex multipliers. Control is trivial, requiring a simple binary counter; each output bit of the counter corresponds to a butterfly element.



(a) Details of the BF2

(b) Diagram

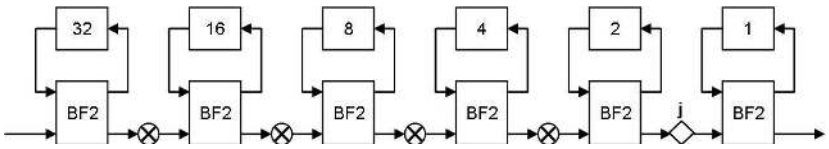
Figure 3.4: 16-point implementation of the radix-2 SDF.

- **R4SDF [7]**

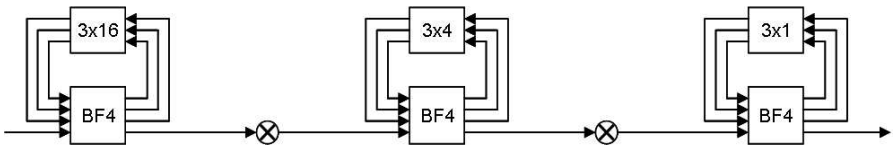
  By using a higher butterfly radix, a pipelined FFT can be built that needs fewer twiddle factor multiplications. The radix-4 single delay feedback (R4SDF) uses the same theme as the R2SDF, but with three delay lines per butterfly instead of one, and twice the number of complex adders. It requires $N-1$ memory elements, $4\log_2 N$ complex adders, and $.5\log_2 N - 1$ complex multipliers. Control is more complex, since each butterfly must now direct four data streams.

- **R2^{2}SDF [10]**

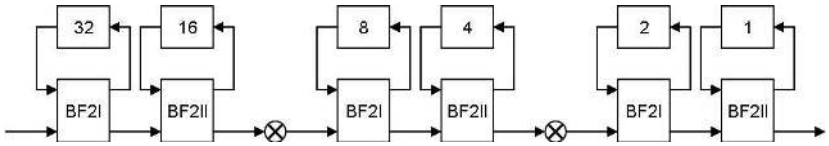
  This is a recent architecture presented in [10], known as radix-2^2 single delay feedback (R2^{2}SDF). The R2^{2}SDF architecture emulates radix-4 butterfly elements by using a pair of modified radix-2 butterflies. Referring to Figure 3.5(c), the **BF2I** element is a standard **BF2**, as found in the R2SDF pipeline. The **BF2II** element is slightly modified, allowing selected inputs to be multiplied by



(a) R2SDF



(b) R4SDF



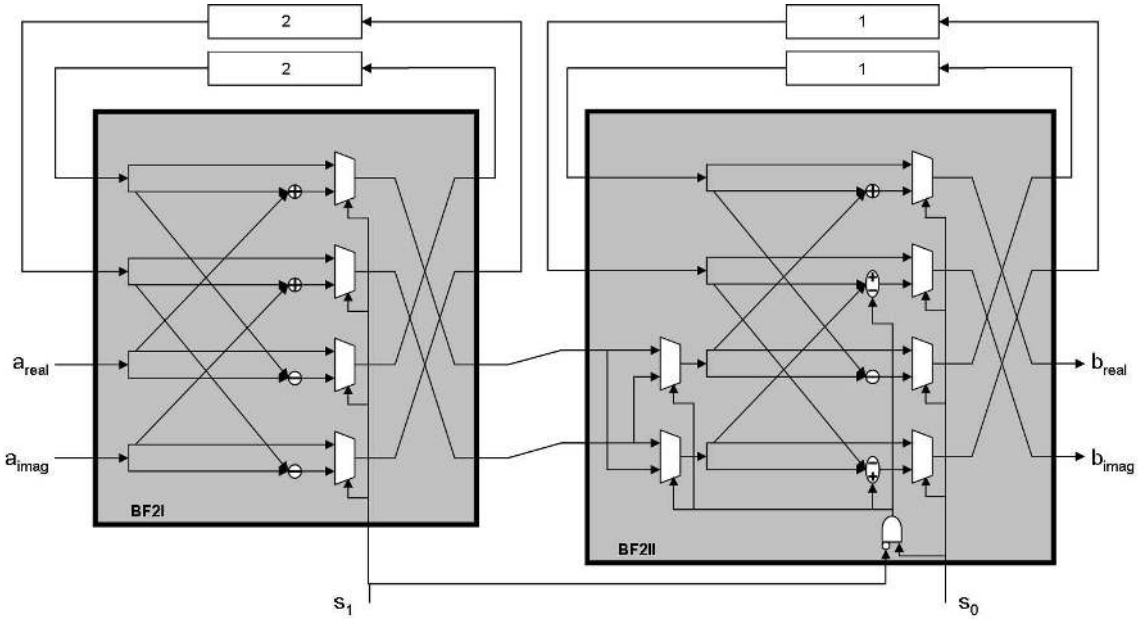
(c) R2^{2}SDF

Figure 3.5: Single Delay Feedback (SDF) Pipelined 64-point FFT Architectures

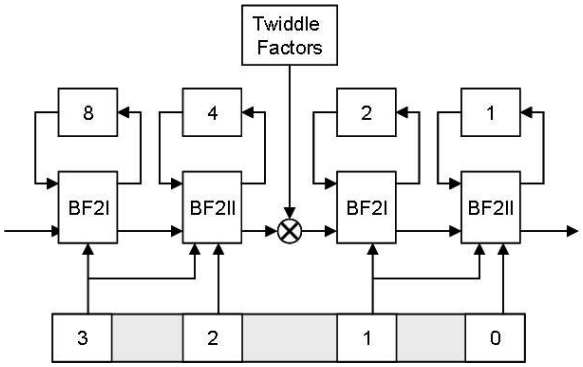
a $-\sqrt{-1}$. Figure 3.6 shows how a R2^{2}SDF pipeline is constructed, and includes details on the butterfly elements.The overall effect of the emulation is a radix-4 algorithm, but with the control and complex additions of a radix-2 algorithm. It requires $N-1$ memory elements, $2\log_2 N$ complex adders, and $.5\log_2 N - 1$ complex multipliers. The control is more complex than that of the R2SDF, but still very simple.

Delay Commutator Pipelined Architectures

The delay commutator pipelined FFT architectures take a different approach to reordering the data. Instead of streaming it through delay feedbacks, the data is delayed and commuted (passed through a switching element) prior to arriving at the butterfly elements. For example, Figure 3.7 shows the delay commutator element



(a) Details of the BF2I and BF2II



(b) Diagram

Figure 3.6: 16-point implementation of the radix-2^2 SDF.

used in the radix-4 multi-delay commutator (R4MDC) architecture. After the data passes through this element, it will be presented at the inputs of the radix-4 butterfly in the correct order.

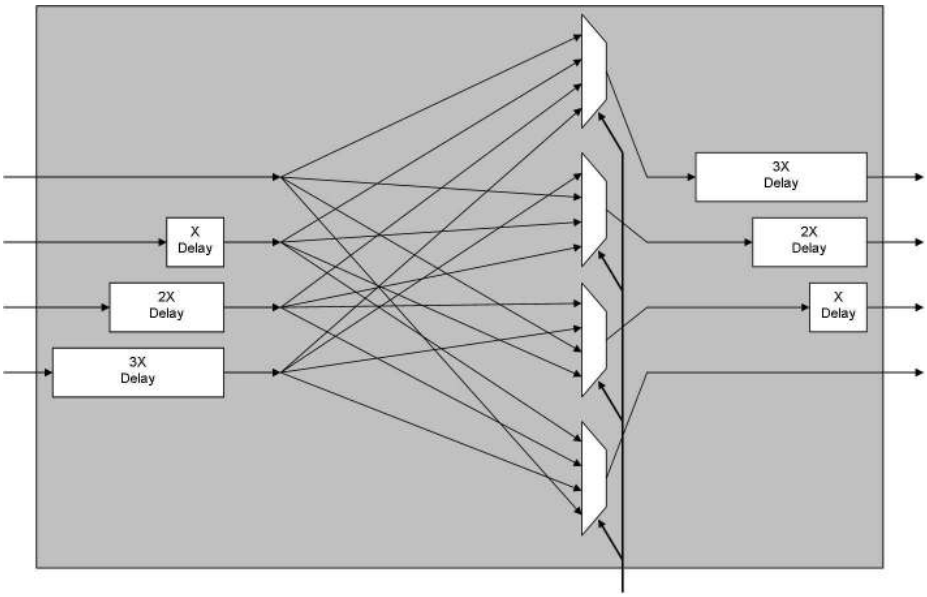


Figure 3.7: A multi-delay commutator for the R4MDC. From [25]

The delay commutator approach tends to be more complex and expensive than the delay feedback method[1]. Given an initial evaluation it would seem that the single-delay feedback architectures are a better choice. But an advantage of the delay commutator is that pipelining is less constrained. The delay feedback architectures are limited in the number of pipeline stages that can be added, because of the feedback delay lines. This can be understood by analyzing Figure 3.6.

Since the maximum clock frequency is limited by the critical path of the circuit, if the butterfly stages in Figure 3.6(b) contain a critical delay, this can be alleviated by the addition of pipeline registers. And for every pipeline register added, a clock

[1]One exception is for block floating-point implementations of the FFT. The R4SDC requires slightly less memory than the R2^{2}SDF when implemented using block floating point arithmetic. But, the R4SDC has very complex control

delay must be removed from the feedback delay loop. This is needed so that the butterfly operands continue to be presented in the correct order. But, the final butterfly feedback loop in the pipeline has only a single clock delay. Only a single pipeline register may be included within the butterfly stage. Thus, for a SDF pipeline architecture which has been maximally pipelined, the final butterfly stage will most likely contain the critical delay path of the entire circuit. This delay cannot be further reduced using pipeline registers.

In contrast, since delay commutators contain only feed-forward paths, then the level of pipelining is limited only by the granularity of the hardware substrate. Thus, delay commutator architectures are commonly found in applications which demand very high clock frequencies [4].

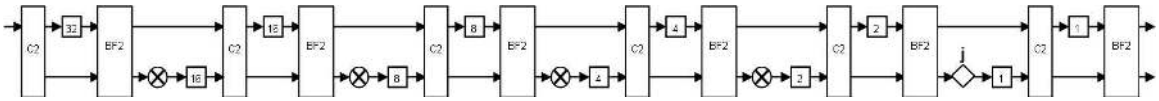
Figure 3.8 contains diagrams for four types of delay commutator FFT architectures. They all implement a 64-point DIF FFT. There are two varieties: the single-delay and multi-delay commutators. The single-delay commutators use fewer resources, but have higher control complexity.

- **R2MDC [21]**

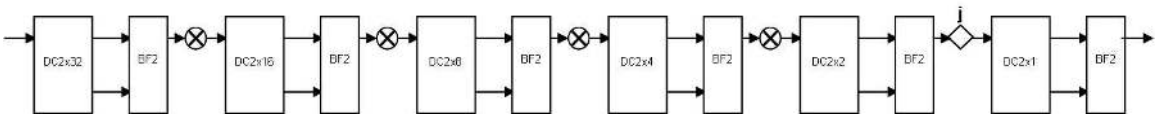
  The radix-2 multi-delay commutator (R2MDC) was an early pipeline FFT architecture. It is the most obvious way to map the radix-2 FFT to a pipeline. It consists of butterfly elements that are essentially identical to those found in the in-place FFT architectures. Delay lines and simple commutators are used for reordering the data for correct presentation to the butterflies. It requires $3N/2 - 2$ memory elements, $2\log_2 N$ complex adders, and $\log_2 N - 2$ complex multipliers. Control is simple, only requiring the data to be switched in the commutator.

- **R2SDC**

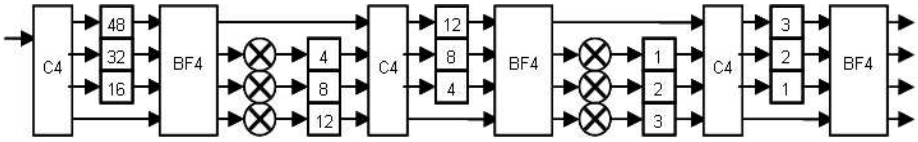
  The R2SDC differs from the R2MDC in that it limits the output from the butterflies to a single stream. The only advantage this provides is increased utilization of the multipliers. But, this is at the expense of greatly increased



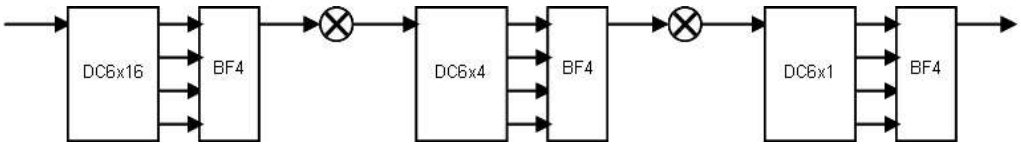
(a) R2MDC



(b) R2SDC



(c) R4MDC



(d) R4SDC

Figure 3.8: Single- and Multi-Delay Commutator 64-point FFTs

complexity and memory. It requires $2N-2$ memory elements, $2\log_2 N$ complex adders, and $\log_2 N - 2$ complex multipliers.

This implementation is only theoretical, and has never actually been implemented to my knowledge. This is because it is inferior to other radix-2 pipeline architectures. It is included here only for completeness.

- **R4MDC [21]**

  The R4MDC is an attempt, similar to the R2MDC, to map the radix-4 FFT

algorithm to a pipeline. But where the R2MDC provides an adequate implementation, the R4MDC is impractical. The purpose of any radix-4 FFT algorithm is to reduce the total number of twiddle factor multiplications from that required by the radix-2 algorithms. But, the R4MDC actually demands *more* multipliers, adders, and memory. And the hardware has a low utilization. Still, some of the first pipelined FFT implementations used this architecture [25]. It requires $5N/2-4$ memory elements, $4\log_2 N$ complex adders, and $1.5\log_2 N-3$ complex multipliers. Control is simple, for the same reasons as found in the R2MDC.

- **R4SDC [3]**

  The radix-4 single delay commutator (R4SDC) reduces the hardware costs of the R4MDC, and increases hardware utilization. This is at the expense of greatly increased control. The R4SDC has a particular advantage for block floating-point implementations, if memory is at a premium [4]. The fixed-point version requires $2N-2$ memory elements, $4\log_2 N$ complex adders, and $0.5\log_2 N-1$ complex multipliers.

Table 3.1: Comparison of Pipelined FFT Architectures

type	multipliers	adders	memory	control
R2SDF	$\log_2 N-2$	$2\log_2 N$	$N-1$	simple
R4SDF	$0.5\log_2 N-1$	$4\log_2 N$	$N-1$	medium
R2^{2}SDF	$0.5\log_2 N-1$	$2\log_2 N$	$N-1$	simple
R2MDC	$\log_2 N-2$	$2\log_2 N$	$3N/2-2$	simple
R2SDC	$\log_2 N-2$	$2\log_2 N$	$2N-2$	complex
R4MDC	$1.5\log_2 N-3$	$4\log_2 N$	$5N/2-1$	simple
R4SDC	$0.5\log_2 N-1$	$4\log_2 N$	$2N-1$	complex

Table 3.1 tabulates the resource requirements of each of the pipelined FFT architectures discussed. It includes information on complex multipliers and adders, and memory requirements. In addition, the control complexity is compared for each design.

Table 3.1 shows that the single-delay feedback architectures require less memory, and often fewer complex multipliers. Also, the single-delay commutators have complex control. Based solely on this information, the single-delay feedback architectures are better because of the reduced hardware requirements, with the R2^{2}SDF being the ideal choice.

Nevertheless, as previously explained, the delay commutator architectures can be more easily pipelined, and thus it will be less challenging to use them in applications where high clock frequencies are encountered. For example, during the development of the HAPFFT, I implemented a quad-pipelined HAPFFT module on an FPGA. Portions of the module used the R2^{2}SDF architecture. In order to improve the FPGA's maximum clock frequency (and thus the total throughput), I heavily pipelined the HAPFFT module. In the end, the maximum clock frequency achieved was 160 MHz, though the FPGA (a Xilinx Virtex II 6000, with a speed grade of -4) was still capable of frequencies beyond 200 MHz. This occurred because the feedback within the R2^{2}SDF pipeline ultimately limited the amount of pipeline registers that could be used. The use of a delay-commutator architecture may have permitted high performance, though with increased hardware.

3.2 Parallel FFT Algorithms for Software

For many years now the FFT has been an important algorithm used in parallel computing applications. Some applications that utilize both parallel computers and the FFT are digital signal processing, solutions for partial differential equations, and image processing, to name just a few.

Parallel computers refers to any computer system which utilizes a collection of identical, general-purpose processors. The memory space may be a single space, shared among the processors, or it may also be partitioned and distributed among them. Distributed memory machines are more difficult to program and use effectively, but also scale more easily. Some distributed memory machines have thousands of processors. The algorithms presented in this chapter are intended for distributed memory machines. Nevertheless, it is often the case that an algorithm optimized for

a distributed memory machine will run equally well, or sometimes even better, on a shared memory machine[2].

Parallel computers take the "brute-force" approach in obtaining high performance. This is at the expense of high cost, in terms of both hardware and power. Nevertheless, because they are programmable, and therefore may be used for a wide range of tasks, they are a very a popular platform for computing the FFT. This has resulted in the development of a wide variety of parallel FFT algorithms for multiprocessor machines.

Though these parallel FFT algorithms are optimized for a software environment, it is nevertheless useful to study them. Because the intent is to increase FFT performance by the addition of more hardware, many of the same problems are encountered in a parallel computing environment as are found in custom-hardware architectures. Two of the most important parallel computer algorithms, the *binary-exchange algorithm* and the *transpose algorithm*, will be discussed in this section.

3.2.1 The Binary-Exchange Algorithm

Figure 3.9 shows the data flow diagram for a 16-point FFT. The rows of the butterfly network have been divided up between 16 different processors, each processor being responsible for the execution of a single row of the network. That is, every processor will execute a single task, which is a sequence of complex additions and complex multiplies. The sequence consists of 4 steps in this example, or $\log N$ steps for an FFT of size N.

At the beginning of each step in the sequence, a processor requires data from two locations: the first from the current processor node, and the other from another

[2]Though a shared memory machine may share its address space, yet the individual processors have separate data and instruction caches. A distributed memory program will often get good performance on these machines because there is less conflict between processor caches trying to access memory values which are in close proximity to each other (known as false sharing). Additionally, larger shared memory machines have distributed physical memory, and the hardware or the operating system creates the illusion of a shared memory space. Such machines have what is termed as non-uniform memory access (NUMA). A processor will see different memory access latencies depending on the location of the target data. Thus, distributed memory programs will also see good performance in this case.

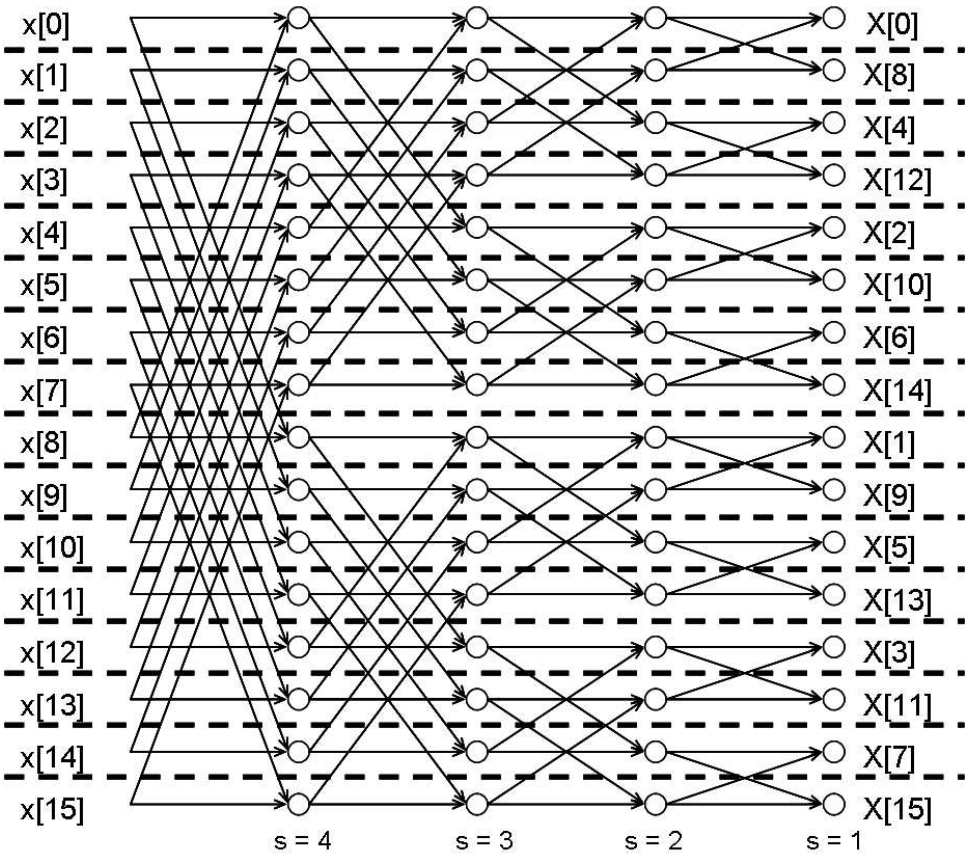


Figure 3.9: 16-point FFT data-flow-graph, mapped onto 16 processors.

process. Herein lies the primary problem with the parallel FFT: the FFT computational tasks can be easily subdivided, but the tasks are not entirely independent of one another. Data must be exchanged between the tasks at the beginning of every processing step. The binary-exchange algorithm and the transpose algorithm are differentiated by the manner in which they handle this problem.

The binary-exchange algorithm handles the data dependency problem by the use of messaging. After each execution step, a messaging step will be taken to exchange data between the tasks prior to the next execution step.

It is shown in [9] that a hypercube parallel processing network provides an optimal messaging environment for the bit-exchange algorithm. Figure 3.10 shows four different example hypercube networks. A complete hypercube network will consist of

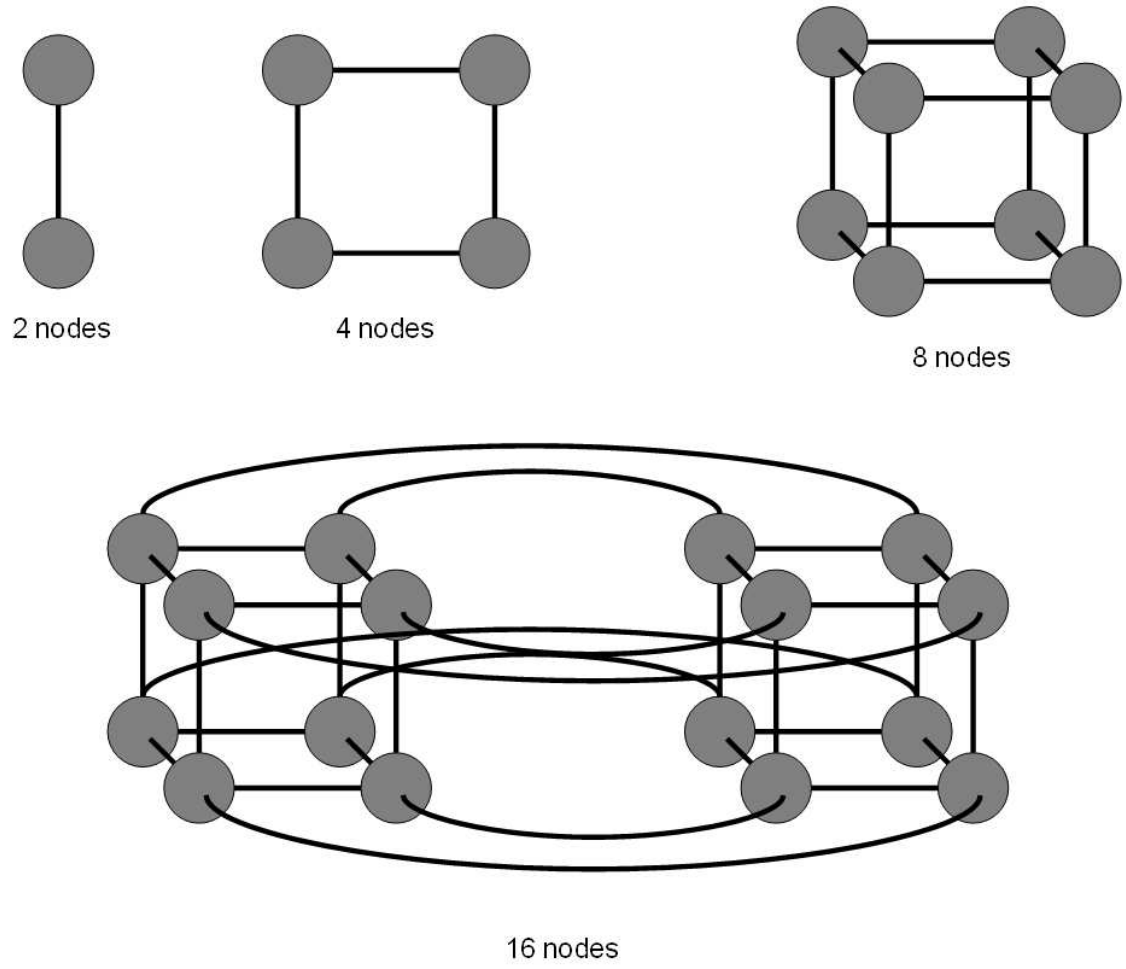


Figure 3.10: Hypercube networks consisting of 2, 4, 8 and 16 nodes.

a power-of-two number of nodes, and each node will have $\log_2 N$ links to other nodes, where N is the number of nodes in the network.

When the binary-exchange algorithm is implemented on a hypercube network, the data required by each task will always be found in an adjacent node; so the cost of messaging is minimized. It can also be implemented on other network topologies, though with an increased cost of communication.

In the previous example, each FFT row had its own processor. But multiple rows can also be assigned to single processors; in such a case a given processor would

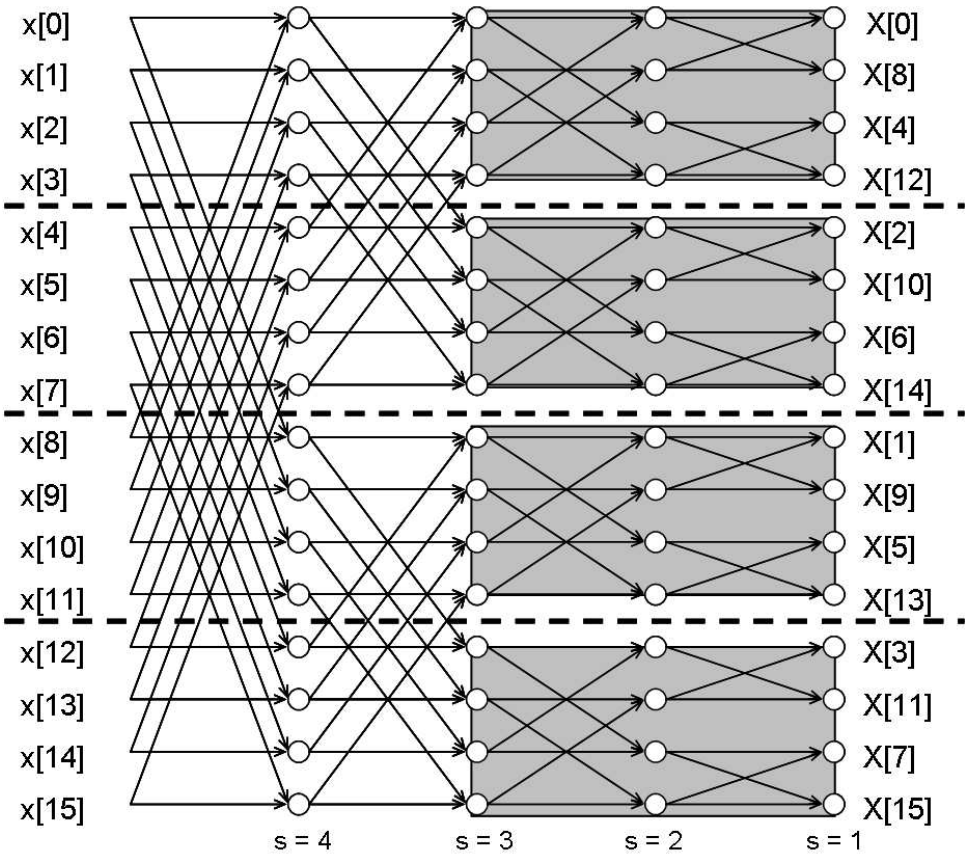


Figure 3.11: 16-point FFT data-flow-graph, mapped onto 4 processors.

execute each row for a given FFT step sequentially before proceeding to the next step. Figure 3.11 shows the same dfg from Figure 3.9, but divided among only four processors. Even when multiple tasks are assigned to each processor, the hypercube network is still the optimal processor network, as the data for all tasks assigned to each processor can be found in an adjacent processing node.

One item of interest to note in Figure 3.11 is that after the first two processing steps, all data dependencies become confined within all processors. The blocks in the dfg show the portions of the FFT computation that can be distributed among the processors without any need of interprocessor messaging. Because messaging is no longer required, multi-task per processor implementations of the bit-exchange

algorithm make more efficient use of the hardware, since less time is spent waiting for or transmitting data.

3.2.2 The Transpose Algorithm

Consider the iterative radix-2 FFT algorithm in Figure 2.7. The data is computed in-place. Assume that the data is of size N, $\sqrt{N}$ is a power of two, and the array is mapped to a 2-D memory space of size $\sqrt{N}$ by $\sqrt{N}$. Figure 3.12 shows the memory reads for each iteration of the outer loop of the iterative FFT algorithm. In a radix-2 FFT, two array elements are required for every butterfly operation. The 2-D memory space is accessed, as shown in Figure 3.12, in the first two outer loop iterations by columns, and then by rows for the final two iterations.

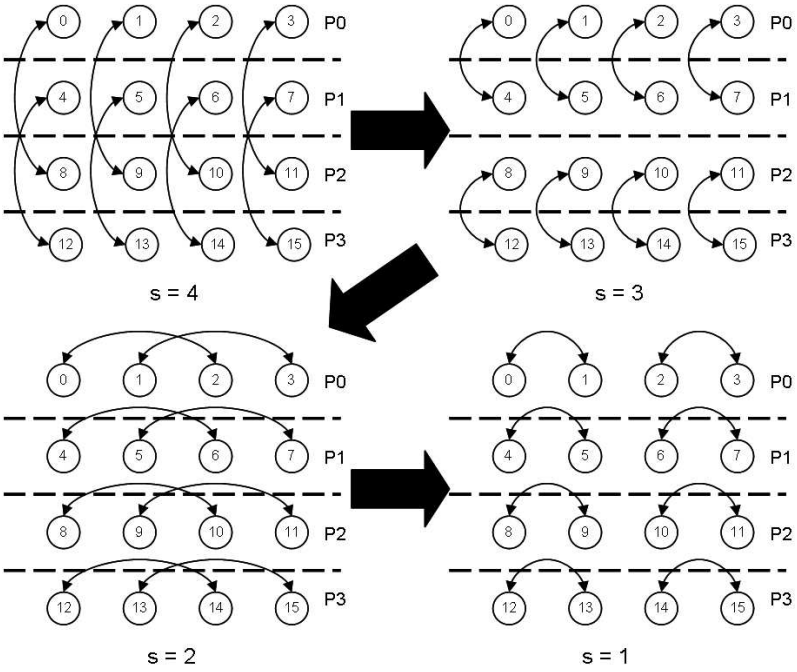


Figure 3.12: Memory plan for the iterative FFT (see Figure 2.7).

One of the steps in devising a parallel computing algorithm is partitioning the data among processors. Figure 3.12 shows how the input data array for the FFT

would be mapped for the binary-exchange algorithm example from Figure 3.11. The rows of data are each mapped to a processing node. If the data is accessed by rows, the most efficient utilization of hardware is achieved, since execution time is not wasted in inter-node communication. But when the memory is being accessed by columns, data must be passed between the nodes, with a resulting increase in execution time.

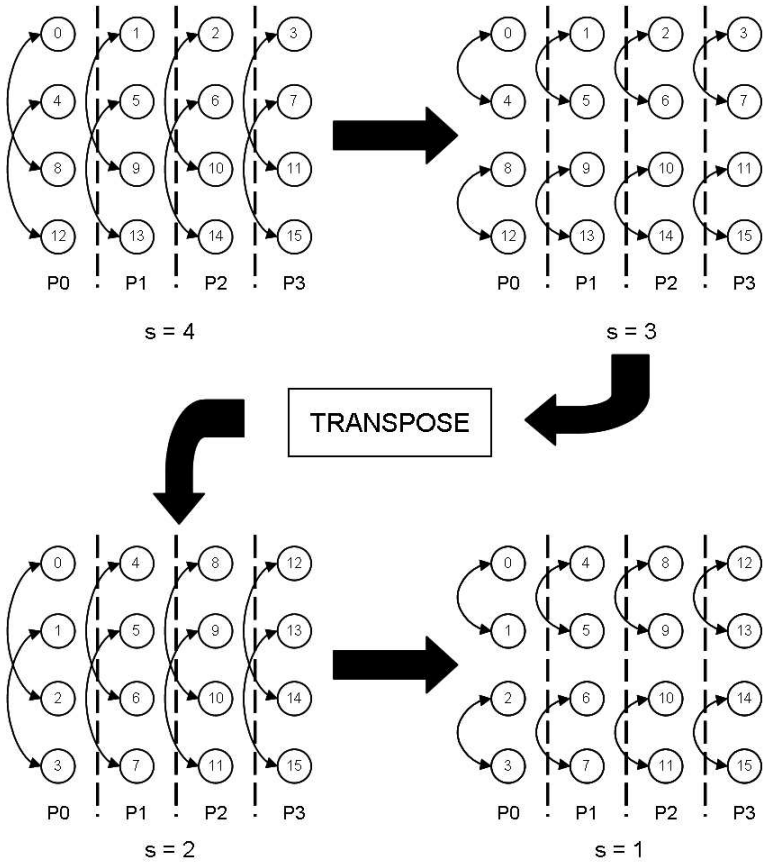


Figure 3.13: Memory plan for transpose parallel-FFT algorithm.

The transpose algorithm attempts to minimize the interchange of data between nodes. It does this by performing a matrix transpose of the 2-D array, allowing all the data required by a processing node to be found within its own local memory. Figure 3.13 illustrates how this is done for our 16-point FFT example. The 2-D

array is mapped to the four processors by columns. During the first two iterations of the outer loop all memory accesses are within local node memory. After the second iteration, a matrix transpose is performed on the array. When the subsequent iterations are performed, the data is again found to be located in local node memory. The consequence is that the only message passing required is during the transpose operation.

What the transpose algorithm is effectively doing is performing a $(\sqrt{N})$-point FFT on each column of the array, transposing it, and then repeating the previous step. It can thus be described in the following steps:

1. Compute a $(\sqrt{N})$-point FFT on each column of the 2-D data array.
2. Perform a matrix transpose on the 2-D data array.
3. Compute a $(\sqrt{N})$-point FFT on each column of the 2-D data array.

The binary-exchange algorithm is optimized for hypercubed network topologies. It is very inefficient on others. The transpose algorithm likewise is optimal on a hypercube, but unlike the binary-exchange, it can be efficiently implemented on others. Appendix B briefly discusses the efficiency of these two algorithms.

Chapter 4

The Hybrid Architecture Parallel FFT (HAPFFT)

Chapter 3 presented an overview of past implementations of the fast Fourier transform for high-performance applications. One of these methods is to use a parallel computing environment. While there will probably always be a supercomputer in existence that will be able to outperform any custom hardware implementation of the FFT, nevertheless supercomputers are very inefficient in terms of both hardware and power. The alternative, custom hardware, is much more attractive.

Until recently, most high-performance hardware implementations of the FFT attempted to achieve greater performance through the use of hardware pipelines. These architectures, though efficiently using hardware, are ultimately of limited use; they can accept a maximum of a single data point each clock cycle. Typical hardware implementations of these architectures operate at clock frequencies in the range of 100-300 MHz. Yet, some signal processing applications, such as radar processing and wireless communications, produce data at rates in excess of 500 mega-sample-per-second (Msps).

A solution to such demanding applications is to implement the FFT in a manner such that it can accept more than a single sample per clock cycle. This has resulted in a series of proposals for custom hardware versions of the parallel FFT. But, the majority of these algorithms are merely attempts to map existing parallel FFT algorithms to hardware. They do not make effective use of the flexibility provided by custom hardware platforms (such as VLSI circuits or FPGAs). This results in products with high memory requirements, complex control, and inefficient routing of data between processing elements.

This chapter proposes a new parallel FFT architecture for use in custom hardware applications: the Hybrid Architecture Parallel FFT (HAPFFT). The HAPFFT traces its origins not from parallel computing algorithms, but instead from the custom hardware FFT pipelines reviewed in Chapter 3. It is an integration of various features from different pipeline architectures, and produces an implementation that is simple to design, makes efficient use of hardware, and requires trivial control.

In the sections that follow, I will present a mathematical derivation of the HAPFFT. This derivation is similar in nature to the DIF FFT derivations discussed in Chapter 2. The HAPFFT derivation may be adapted by the reader to generate many different sizes and types of HAPFFTs. Subsequently, some examples of the HAPFFT are examined and discussed. Lastly, I will present experimental results obtained during the course of developing the HAPFFT.

4.1 Review of the Parallel FFT

Figure 4.1 shows the dfg for a 16-point DIF FFT. As a review of Chapter 2, the computation of an N-point FFT is split into log(N) stages, 4 in this case. Each stage consists of complex additions and multiplications.

The first stage is characterized by data communications between distant rows. For example, the first row computation requires $x[0]$ and $x[8]$. But observe that at each new stage the data dependencies move closer. For the second stage, the first row now requires data from $a[0]$ and $a[4]$, where $a[n]$ is the intermediate result produced by the first stage. The dfg branches into independent paths of computation.

After two stages the dfg has been divided into four independent computational branches, denoted by the horizontal dashed lines. This suggests the use of four independent processing elements to compute these branches in parallel. This is the same observation make in Section 3.2. The binary-exchange and transpose parallel FFT algorithms made use of this observation, and differed only in the manner in which they handled the interdependence of data during the first stages of of the dfg.

Parallel computer FFT algorithms are assumed to be operating on a number of *identical* processing elements. This constraint shapes the way that the algorithms

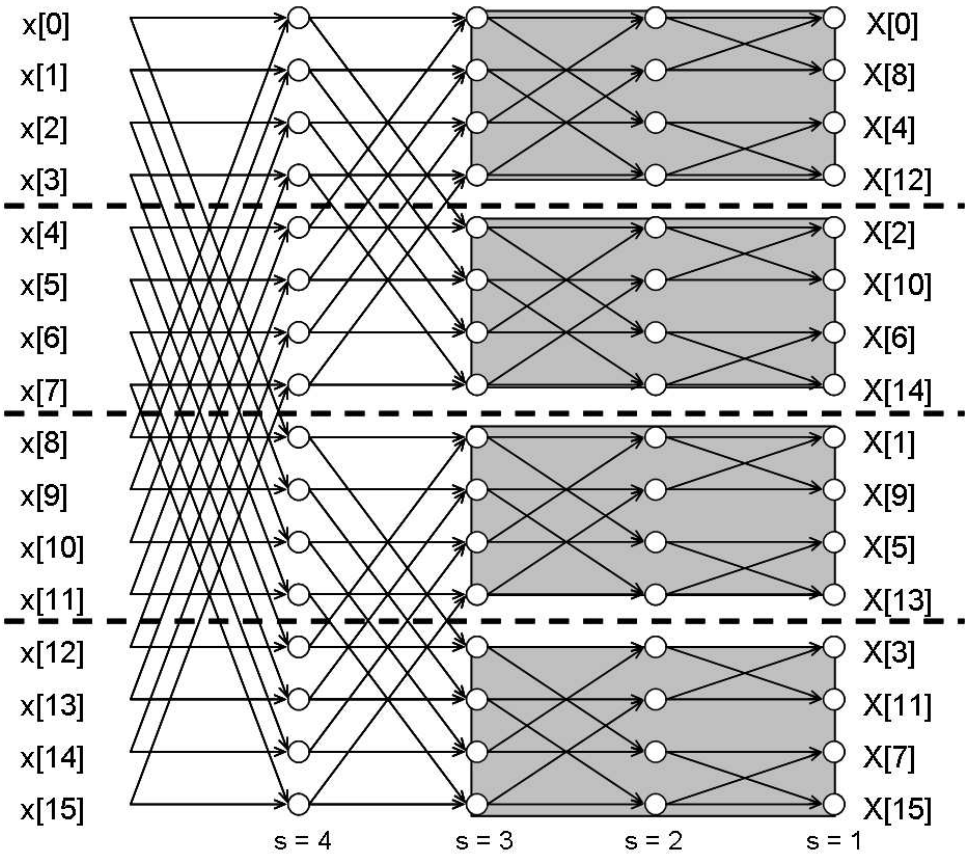


Figure 4.1: 16-point FFT data-flow-graph

are designed. Since general-purpose hardware attempts to make the common case fast, no specialized hardware is typically available for performing unusual operations. For the FFT, with its complex changes in data dependencies, this results in a decrease in parallel efficiency. This is because the input data array must be transformed one or more times over the course of the computation.

4.2 Mapping the Parallel FFT to Custom Hardware

For implementation with custom hardware, multiple, identical processing elements could be used. This is analogous to a parallel computing environment. Many existing custom hardware parallel FFT architectures indeed follow this paradigm.

Yet, because the HAPFFT is a custom hardware implementation, it need not be limited to using identical processing elements. Instead, my approach is to create a custom front-end processing element. This element handles the portion of the FFT in which data dependencies span large segments of the input sequence. The output of this processing element produces intermediate results. These results are then redirected to multiple, identical processing elements. The inputs to these processing elements are completely independent of one another, thus removing the need to share data between the elements. The following derivation will assume that there are four processing elements available. The derivation produces a formulation for the custom front-end module, and a formulation for each of the identical, parallel, back-end processing elements.

In review, the Discrete Fourier Transform (DFT) of an N-length sequence, $x[n]$, is defined as

$$X[m] = \sum_{n=0}^{N-1} x[n] W_N^{mn}. \tag{4.1}$$

There will be four processing elements producing the final result. Therefore, the output, $X[m]$, needs to be split into four sequences. This results in

$$X[4m] = \sum_{n=0}^{N-1} x[n] W_N^{(4m)n} \tag{4.2}$$

$$X[4m+1] = \sum_{n=0}^{N-1} x[n] W_N^{(4m+1)n} \tag{4.3}$$

$$X[4m+2] = \sum_{n=0}^{N-1} x[n] W_N^{(4m+2)n} \tag{4.4}$$

$$X[4m+3] = \sum_{n=0}^{N-1} x[n] W_N^{(4m+3)n}. \tag{4.5}$$

Because the module will be producing four results every clock cycle, in order to fully utilize the pipeline, four data point must also be input every clock cycle. Equations (4.2)-(4.5) must be in terms of four separate input sequences. Beginning with (4.2),

$$\begin{aligned} X[4m] &= \sum_{n=0}^{N-1} x[n] W_N^{(4m)n} \\ &= \sum_{n=0}^{N/4-1} x[n] W_N^{4m} + \sum_{n=N/4}^{N/2-1} x[n] W_N^{4m} + \end{aligned}$$

$$\sum_{n=N/2}^{3N/4-1} x[n]W_N^{4m} + \sum_{n=3N/4}^{N-1} x[n]W_N^{4m}.$$

Now, using variable substitution in the summations, it follows that

$$\begin{aligned} X[4m] &= \sum_{n=0}^{N/4-1} x[n]W_N^{4m} \\ &+ \sum_{n=0}^{N/4-1} x[n+N/4]W_N^{4mn}W_N^{mN} \\ &+ \sum_{n=0}^{N/4-1} x[n+N/2]W_N^{4mn}W_N^{2mN} \\ &+ \sum_{n=0}^{N/4-1} x[n+3N/4]W_N^{4mn}W_N^{3mN}, \end{aligned}$$

and because $W_N^{ZmN} = 1$ and $W_N^{Zmn} = W_{N/Z}^{mn}$, where Z is some integer, the final solution becomes

$$\begin{aligned} X[4m] &= \sum_{n=0}^{N/4-1} (x[n] + x[n+N/4] + \\ &\quad x[n+N/2] + x[n+3N/4])W_{N/4}^{mn}. \end{aligned}$$

The derivations for the other output blocks can be obtained in a similar fashion, resulting in

$$\begin{aligned} X[4m+1] &= \sum_{n=0}^{N/4-1} (x[n] - jx[n+N/4] + \\ &\quad x[n+N/2] + jx[n+3N/4])W_{N/4}^{mn}W_N^n, \\ X[4m+2] &= \sum_{n=0}^{N/4-1} (x[n] - x[n+N/4] + \\ &\quad x[n+N/2] - x[n+3N/4])W_{N/4}^{mn}W_N^{2n}, \\ X[4m+3] &= \sum_{n=0}^{N/4-1} (x[n] + jx[n+N/4] - \\ &\quad x[n+N/2] - jx[n+3N/4])W_{N/4}^{mn}W_N^{3n}. \end{aligned}$$

Next, the following variables will be created for the input sequences of the four output processing elements:

$$\begin{aligned} a_0[n] &= (x[n] + x[n+N/4] + x[n+N/2] + \end{aligned}$$

$$x[n+3N/4]), \tag{4.6}$$

$$\begin{aligned} a_1[n] &= (x[n] - jx[n+N/4] + x[n+N/2] + \\ & \quad jx[n+3n/4])W_N^n, \end{aligned} \tag{4.7}$$

$$\begin{aligned} a_2[n] &= (x[n] - x[n+N/4] + x[n+N/2] - \\ & \quad x[n+3n/4])W_N^{2n}, \end{aligned} \tag{4.8}$$

$$\begin{aligned} a_3[n] &= (x[n] + jx[n+N/4] - x[n+N/2] - \\ & \quad jx[n+3n/4])W_N^{3n}. \end{aligned} \tag{4.9}$$

Substituting these into the derived block DFT equations produces

$$\begin{aligned} X[4m] &= \sum_{n=0}^{N/4-1} a_0[n]W_{N/4}^{mn} \\ X[4m+1] &= \sum_{n=0}^{N/4-1} a_1[n]W_{N/4}^{mn} \\ X[4m+2] &= \sum_{n=0}^{N/4-1} a_2[n]W_{N/4}^{mn} \\ X[4m+3] &= \sum_{n=0}^{N/4-1} a_3[n]W_{N/4}^{mn}. \end{aligned}$$

Observe that these four results are each DFTs of length $N/4$, and that they each share the same twiddle-factors of $W_{N/4}^{mn}$. Each DFT can be computed by an independent FFT module of length $N/4$. The only additional hardware needed is to compute the set of input sequences $\{a_1[n], a_2[n], a_3[n], a_4[n]\}$. Figure 4.2 shows a circuit that accomplishes this. Comparing Figure 4.2 to Figure 2.1, it is seen that it is a radix-4 butterfly. A key observation of this thesis is that this set of four sequences can be computed using a conventional radix-4 butterfly, followed by the twiddle factor multiplications indicated in Equations 4.6 through 4.9. Though this result may seem obvious, recent architectures [31, 20, 30, 18, 19, 22] fail to use it. Instead, in order to handle data interdependencies, unnecessarily complex solutions are required. In contrast, the radix-4 butterfly, and the other butterfly elements of varying radices, are well understood structures, and are therefore simple to incorporate into the HAPFFT.

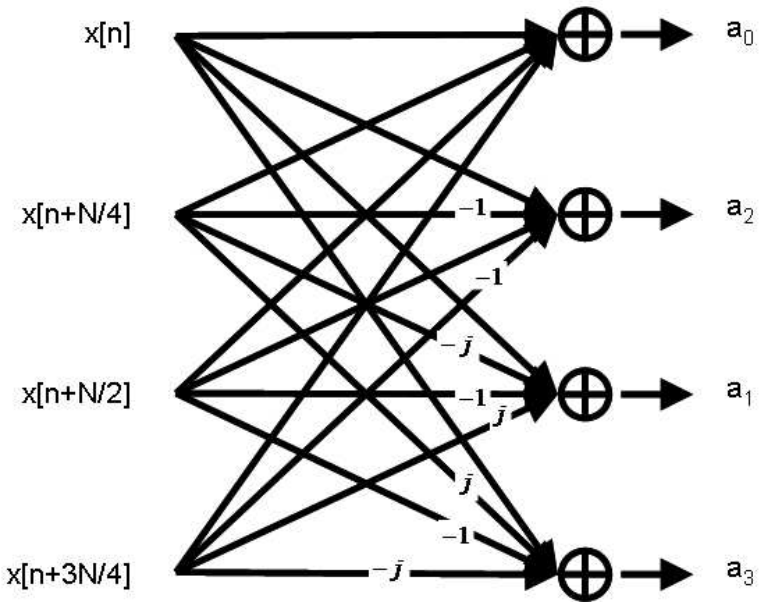


Figure 4.2: Module for computing the four DFT input sequences.

The radix-4 butterfly computes the input sequences for the four $N/4$ DFT blocks. The DFT blocks are not limited in the means by which they can be implemented. They could, for example, be radix-2 FFT pipelines, or they could each implement a different FFT algorithm. This result is equivalent to that used for computing mixed-radix FFTs [23], as discussed in Section 2.3.

The mixed-radix FFT is a solution that provides more flexibility in choosing the size of FFT modules. Another major contribution of this work is in recognizing that this same solution can be used for building efficient, simple, but also high-performance, parallel FFT modules for custom hardware. The resulting HAPFFT is so named because it is able to mix completely different FFT architectures to produce hybrid implementations. This is analogous to the way the mixed-radix FFT also mixes different FFT algorithms in order to compute non-standard-sized FFTs.

In addition to the greater design simplicity and flexibility that the HAPFFT gives a designer, it also has another, unexpected benefit. If the designer uses identical FFT modules for all the back-end processing elements, then the modules are able to

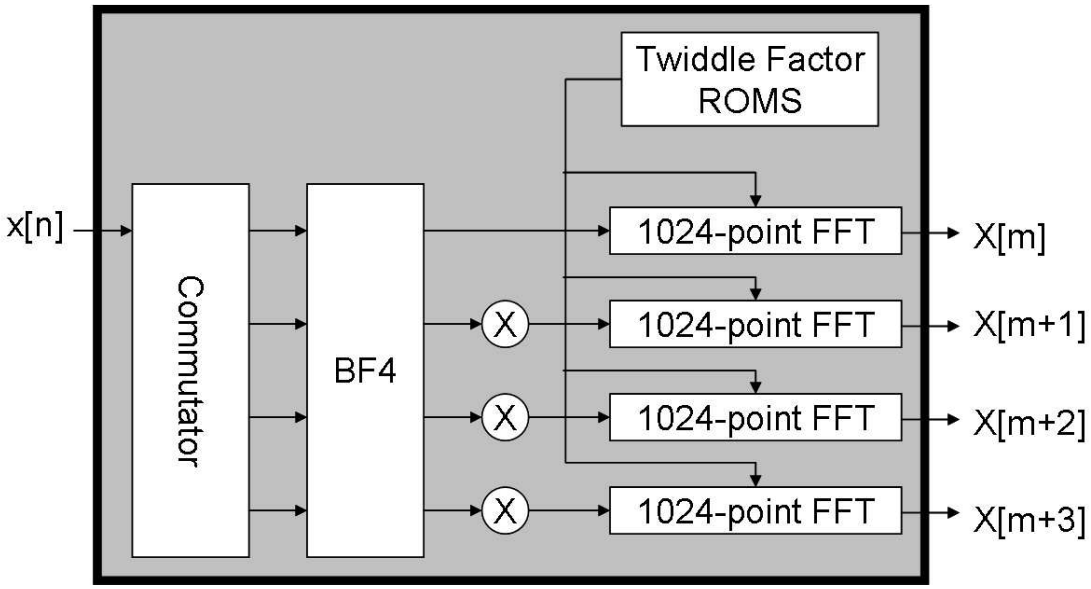


Figure 4.3: 4096-point Quad-pipeline HAPFFT

share many of their resources, such as twiddle factor ROMS, and control. Figure 4.3 shows an example implementation of the HAPFFT. A radix-4 butterfly produces intermediate results for four pipelined FFTs. The pipelines share twiddle factors from a common ROM. Also, The amount of memory required by the HAPFFT, in comparison to a standard FFT pipeline, does not increase. The aggregate of these effects is that a HAPFFT with N times the throughput of an otherwise similar FFT module will require *less* than N times the resources. Therefore, the HAPFFT exhibits superlinear speedup.

The HAPFFT requires a delay-commutator stage at the input of the front-end processing unit. This is needed so that the input array elements can be presented in the proper order to the processing unit. Figure 4.3 shows the commutator, radix-4 butterfly, and twiddle factor multipliers at the beginning of the pipeline. Comparing it to Figure 3.8(c), the front-end is very similar to the R4SDC architecture. They differ in that the R4SDC must only supply a single stream of data from the butterfly, where as the HAPFFT must supply a number equal to the number of back-end processing

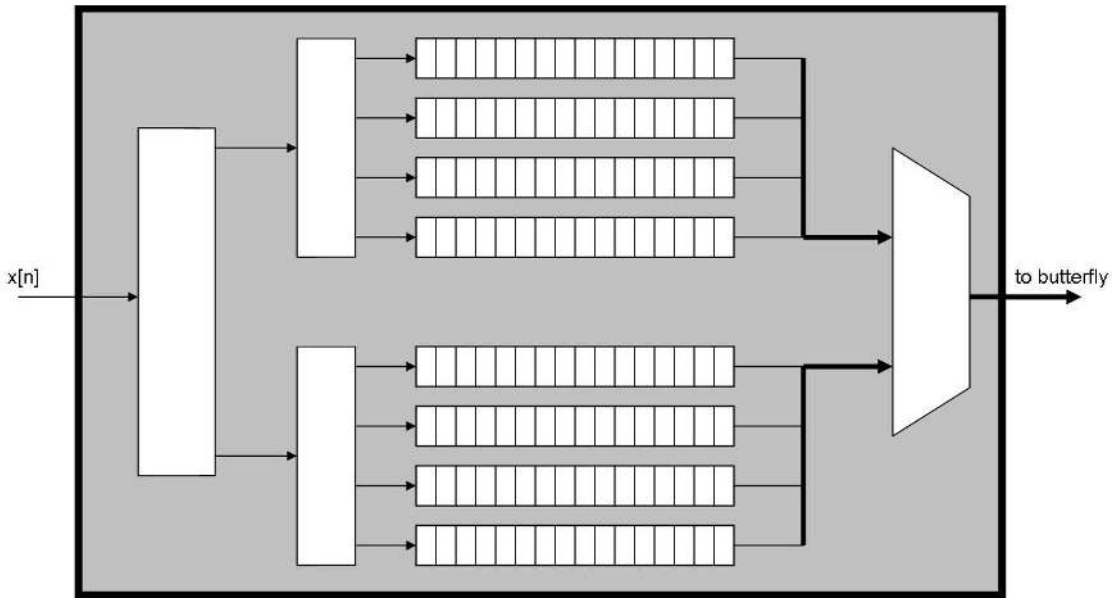
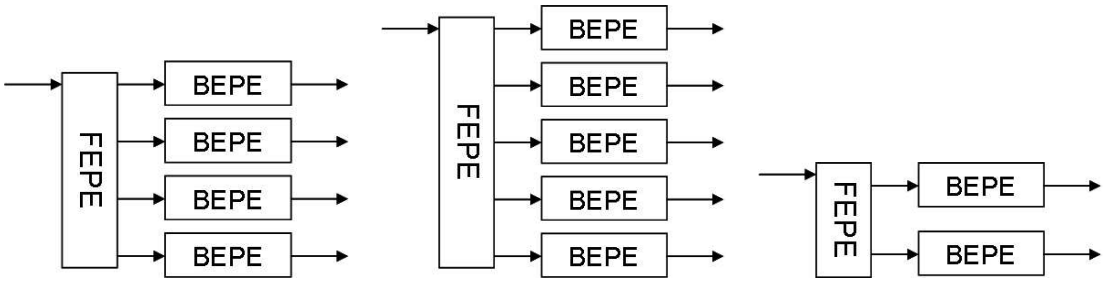


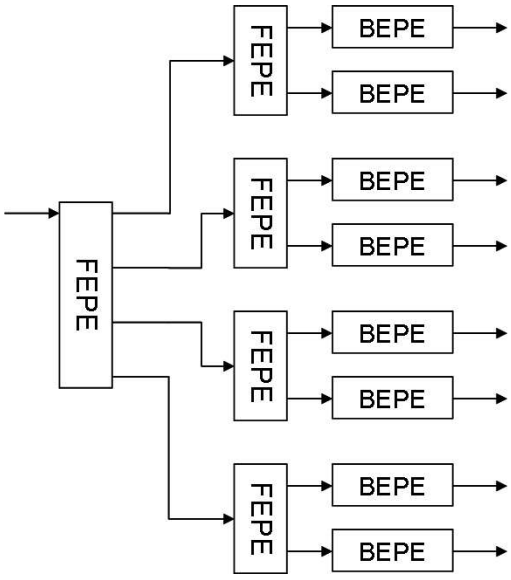
Figure 4.4: Delay commutator for a 64-point HAPFFT.

units. Additionally, the HAPFFT delay-commutator must operate at a much higher clock frequency, equal to the sampling frequency of the input. The delay-commutator of the HAPFFT is its most architecturally complex component. Figure 4.4 shows the schematic for a delay commutator from a 64-point quad-pipeline HAPFFT. In order to allow a constant input stream of data, $2 \times N = 128$ memory elements are required. In addition, control signals (not shown) are needed to manipulate the decoders and the multiplexer.

The HAPFFT examples used so far have been for a quad-pipelined parallel FFT. Yet there are other options available. Figure 4.5 show some variations on the HAPFFT. The **FEPE** blocks are front-end processing elements, incorporating the delay-commutator, butterfly, and twiddle factor multipliers. The **BEPE** blocks are the back-end processing elements, which implement the independent DFTs. These could be FFT pipelines, or other modules of the designers choice. Note the rather conventional dual- and penta-pipeline HAPFFT modules.



(a) Four Pipes (b) Five Pipes (c) Two Pipes



(d) Eight Pipes

Figure 4.5: Variations of the HAPFFT.

Figure 4.5(d) shows an interesting HAPFFT variation. Since the implementations of the back-end processing units are independent of the front-end, the back-ends can themselves be implemented using the HAPFFT.

4.3 The HAPFFT Exhibits Superlinear Speedup

As mentioned in Section 4.2, the HAPFFT exhibits superlinear speedup. The consequences of this are that FFT modules using the HAPFFT architecture will be able to achieve high-performance with a minimal increase in hardware resources.

Superlinear speedup is defined as a greater than m speedup associated with an m-times increase in resources. For example, consider a hypothetical algorithm that executes in time t_0 on a single-processor machine. Now take the same algorithm, but execute it on a multi-processor machine with m processors. If the new execution time is denoted by t_n, and $t_0/t_n > m$ holds, then a superlinear speedup of the algorithm has been achieved.

Superlinear speedup can alternatively be defined as a less than s-times increase in resources for a speedup of s. This type of speedup is also known as *sublinear area-time growth*. The type of superlinear speedup seen in the HAPFFT is of this type. In other words, the HAPFFT can give s-times the performance for less than s-times the hardware increase.

Superlinear speedup is a very desirable attribute of any algorithm or architecture, since it allows very efficient high-performance digital systems to be built. Nevertheless, it is not the rule, but rather the exception. Most solutions display a *sublinear speedup*, or a *superlinear area-time growth*. This fact is what makes the HAPFFT more interesting than it would otherwise be.

Table 4.1 tabulates the resource requirements for a typical HAPFFT implementation. The data is for a HAPFFT utilizing a R2SDF pipelined FFT for the BEPEs. Several incarnations of the HAPFFT are analyzed, each utilizing a different number of parallel pipelines. The *type* field denotes the name of the implementation. The R2SDF is used as a baseline example. The HAPFFT-P is for a given implementation of the HAPFFT, where P denotes the number of parallel pipelines. Each of

Table 4.1: HAPFFT resource requirements.

type	multipliers	adders	memory	throughput
R2SDF	$\log_2 N - 2$	$2\log_2 N$	$N - 1$	1
HAPFFT-2	$2\log_2 N/2 - 3$	$4\log_2 N/2 + 2$	$2N - 2$	2
HAPFFT-4	$4\log_2 N/4 - 5$	$8\log_2 N/4 + 8$	$2N - 4$	4
HAPFFT-8	$8\log_2 N/8 - 7$	$16\log_2 N/8 + 26$	$2N - 8$	8
HAPFFT-16	$16\log_2 N/16 - 7$	$32\log_2 N/16 + 74$	$2N - 16$	16
HAPFFT-P	$P\log_2 N/P - P - 1 + B_m$	$2P\log_2 N/P + B_a$	$2N - P$	P

the table fields shows data for the number of complex multipliers, complex adders, memory locations, and the throughput (in samples-per-clock-cycle). The HAPFFT-P row gives data for the general case of P parallel pipelines. B_m and B_a are the number of complex multipliers and complex adders, respectively, required by the FEPE. N denotes the size of the input data frame. Since all implementations in this example utilize the R2SDF architecture, the size must be a power-of-two in size.

Figure 4.6 plots the resource count versus throughput for the case $N = 4096$. A plot is included for complex multipliers, complex adders, and memory locations. For a throughput of 1 sample-per-clock-cycle, the R2SDF architecture is used, and all other data points correspond to the appropriate version of the HAPFFT example from Table 4.1. The dashed line shows the expected resource growth if linear speedup is assumed. The solid line shows the actual resource growth, derived from Table 4.1.

For multipliers, adders and memory locations, superlinear speedup is observed. The multipliers and adders only show a moderate effect, but it is dramatic for the number of memory locations. From a throughput of 1 sample-per-clock-cycle to 2 samples-per-clock-cycle there is a linear growth for memory locations. Yet, all throughputs above this maintain an approximately equal memory requirement. The reason for the initial linear growth is that the HAPFFT, in most applications, requires a commutator before the FEPE. This commutator is of size N. But, as the throughput increases above 2 samples-per-clock-cycle, the commutator size remains

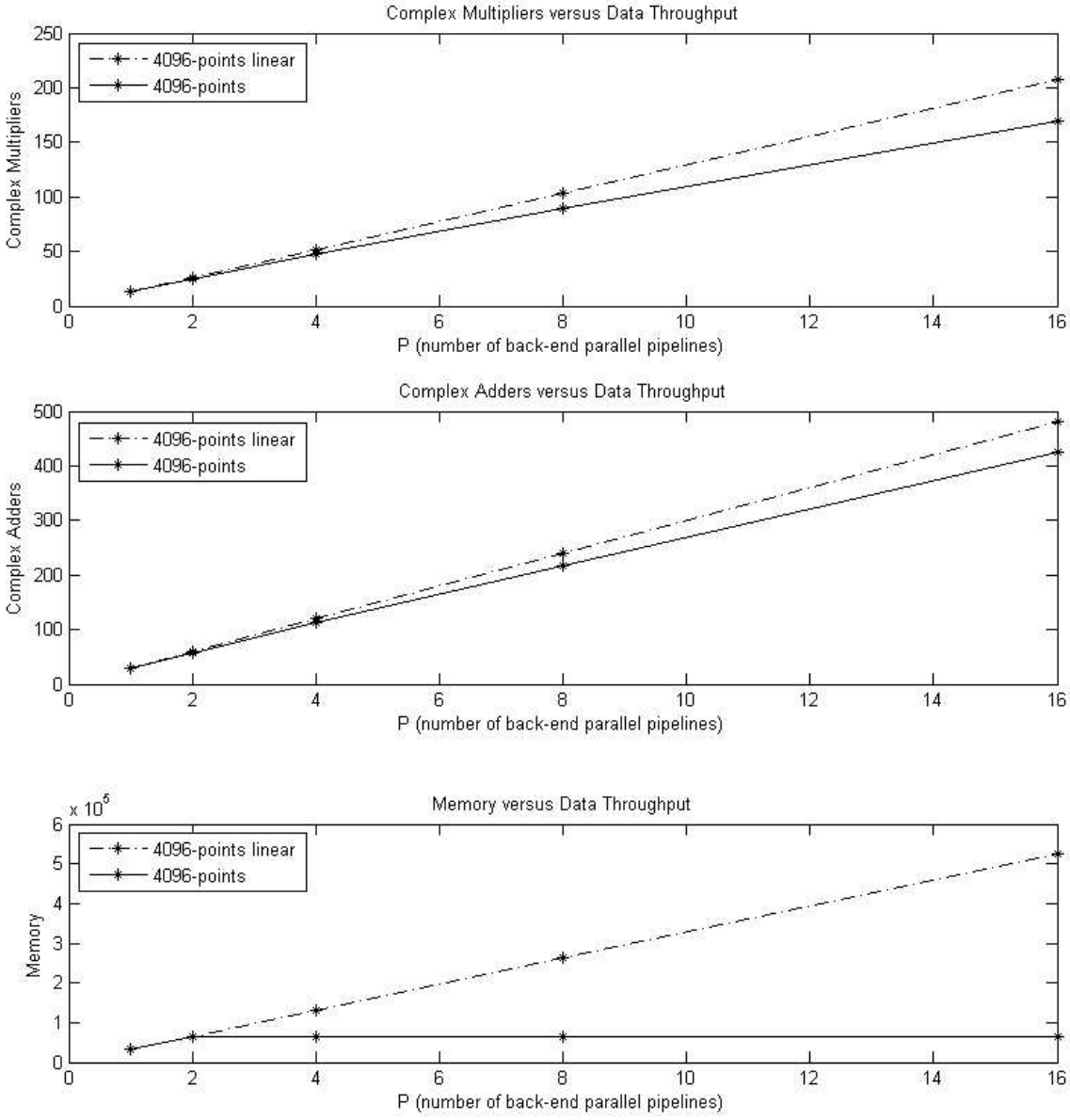


Figure 4.6: Resource requirements of the HAPFFT.

constant. Additionally, the collective memory requirements for all BEPEs remains approximately constant, as seen in Table 4.1 and Figure 4.6.

This data is for a specific example of the HAPFFT. Yet, the results, in general, hold for all implementations and sizes. Additionally, the analysis presented here gives theoretical limits on multiplier, adder and memory requirements. The actual results will be implementation and technology dependent. Nevertheless, as will be seen in Section 4.4, my own experiments have shown that these limits are achievable.

Why is superlinear speedup achievable? The biggest effect is attributed to memory growth. During the computation of the FFT, memory must be used for storing the intermediate results. Different architectures achieve this in different ways. Yet, the fact holds that an increase in FFT throughput does not affect the size of the input data array. Given that the data array has a static size, there should not be an exceptional increase in memory elements, despite a growth in total global resources. In general, any memory growth should only be required for additional pipeline registers.

The growth of multipliers and adders is only moderately superlinear. Nevertheless, it is thus so, and will not take away from the global superlinear speedup effect.

Another significant factor, not predicted in the analysis presented in this section, can contribute to an even more dramatic superlinear speedup. This can be achieved under certain conditions. First, the BEPEs must be architecturally identical, and second, they must work in lock-step (i.e. each BEPE must complete the same computational steps together). In such a case, the BEPEs may be able to share many of their resources. For example, for a typical large FFT module, twiddle factor storage can be significant. Since these twiddle factors are stored in ROMs, all BEPEs can share a single storage location. This can dramatically decrease the amount of memory. Also, since the BEPEs operate in lock-step, they can share control logic. For the R2SDF used in this example this is trivial, since the control is extremely simple. Nevertheless, in more complex modules this could be a source of further savings.

4.4 Experimental Results

The HAPFFT has been implemented using JHDL [1]. Both fixed-point and convergent block floating-point (CBFP) versions have been created. Dual-pipelined and quad-pipelined versions exist. The back-end processing elements are all implemented using the radix-2^2 pipelined FFT architecture.

All HAPFFT modules are parameterized for size and internal bit-width, and the I/O interfaces are single-precision floating-point. The data is converted internally to the desired format. Internal bit-widths of 9-32 bits have been tested. The module size are also parameterizable. Data array sizes from 32-points to 16,384-points have been tested. All results were obtained using the Xilinx Virtex 2, part XC2V6000-4.

A sample of the results is shown in Table 4.2. This data is for fixed-point arithmetic, with an 18-bit word size. The tables contain data for a single-pipeline implementation using the radix-2^2 architecture, as well as a quad-pipelined HAPFFT utilizing the radix-2^2 architecture for the BEPEs. Table 4.2(a) shows resource usage for the two architectures, implemented for three different DFT sizes. Table 4.2(b) gives a comparison of performance.

The tables show that the quad-pipelined HAPFFT uses only 2-3 times the amount of resources as a single-pipelined FFT of a similar size. Yet, it has 1/4-th the transform time (assuming an identical clock frequency). Table 4.2(c) plots the resource usage versus the number of pipelines for the two architectures. These experimental results show that the increase in resource usage is sublinear. The analysis in Section 4.3 predicts a sublinear growth in resource usage, and these results support this conclusion.

In conclusion, the superlinear speedup exhibited by the HAPFFT is unexpected. The HAPFFT is targeted at high-performance applications where data throughput demands are greater than those supplied by more conventional FFT implementations. Sublinear speedup would have been satisfactory, if the performance goals had been met. The fact that the performance goals were achieved, while also exhibiting superlinear speedup, is a very satisfying result.

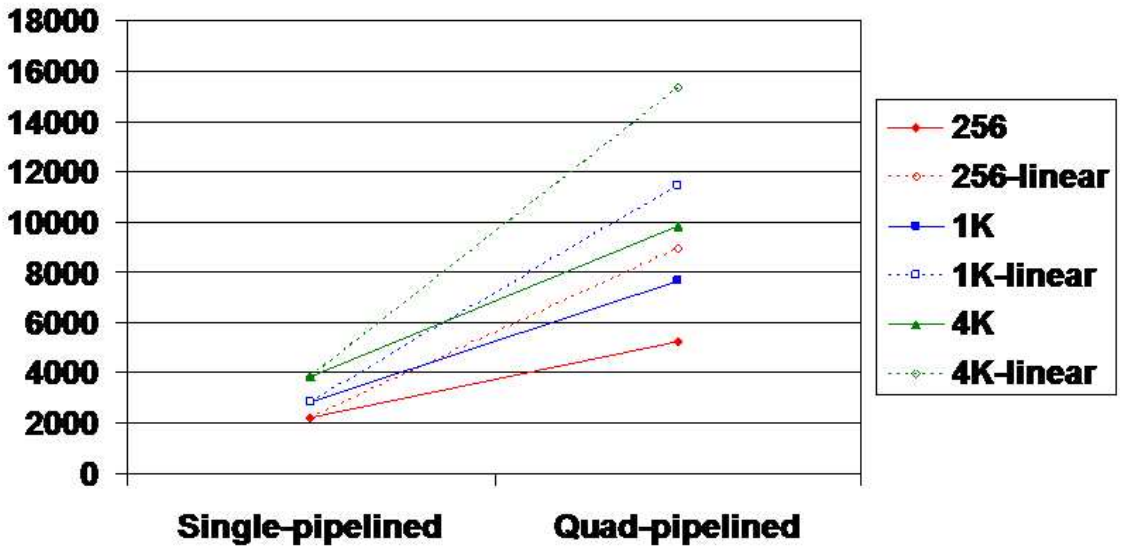
Table 4.2: Results for Fixed-point FFTs on the Xilinx XC2V6000-4

Input Size	Pipeline Style	Slices	Block RAMs	Block MULTs
256	Single	2,233	6	9
	Quad	5,228	11	33
1K	Single	2,870	15	12
	Quad	7,656	27	45
4K	Single	3,838	33	15
	Quad	9,846	63	57

(a) Resource Results

Input Size	Pipeline Style	Speed (MHz)	Latency (cycles)	Throughput Msps	Transform Time (μs)	Area $\times$ Time Product
256	Single	163	547	163	1.57	3,506
	Quad	151	161	604	0.42	2,196
1K	Single	164	2,092	164	6.24	17,909
	Quad	151	554	604	1.70	13,015
4K	Single	155	8,245	155	26.4	101,323
	Quad	150	2,099	600	6.83	67,248

(b) Performance Results



(c) Resources vs. Pipelines

Chapter 5

Conclusions

The FFT is an efficient algorithm for computing the DFT. It has had a profound impact in many engineering and scientific fields, and because of this it has been a widely studied research topic.

Nevertheless, because the FFT is an expensive operation, and because today's technology continues to demand ever more performance, new methods for implementing the FFT are needed. Parallel computers can give the desired performance, but they are large and expensive. Hardware pipelined FFTs are smaller and more efficient, yet the conventional approaches have limited performance. These architectures improve performance by increasing computational concurrency. But, this is done through hardware pipelining, and such an approach is ultimately limited by the maximum clock frequency of the hardware substrate.

Recent efforts have produced multipipelined FFT architectures capable of greater performance. The multiple pipelines allow concurrency to be increased not only through pipelining, but also through parallel streams of computation. This allows the clock frequency barrier to be ignored. But these architectures tend towards hardware mappings of the parallel FFT. The traditional parallel FFT is intended for parallel computing environments where all processing elements are assumed to be identical. Because of this, the hardware mappings are difficult to implement, and control can be complicated.

I have presented in this thesis the HAPFFT. It is a parallel FFT architecture that traces its roots from conventional hardware pipelined FFTs, rather than

from the software parallel processing algorithms. It can incorporate diverse FFT architectures into a single parallel pipelined architecture that is simple, and requires minimal control. It allows designers to adapt existing FFT architectures into a parallel FFT implementation with few changes. Additionally, experimental results have verified that the HAPFFT exhibits superlinear speedup. Therefore, the designer can achieve his or her performance goals without an exceptional increase in hardware requirements.

5.1 Future Research Involving the HAPFFT

The derivation of the HAPFFT results in a generalized formulation. It allows flexibility in the choice of processing elements used for the back-end processing of the FFT. Because of this, the HAPFFT can be easily adapted to a variety of computing models.

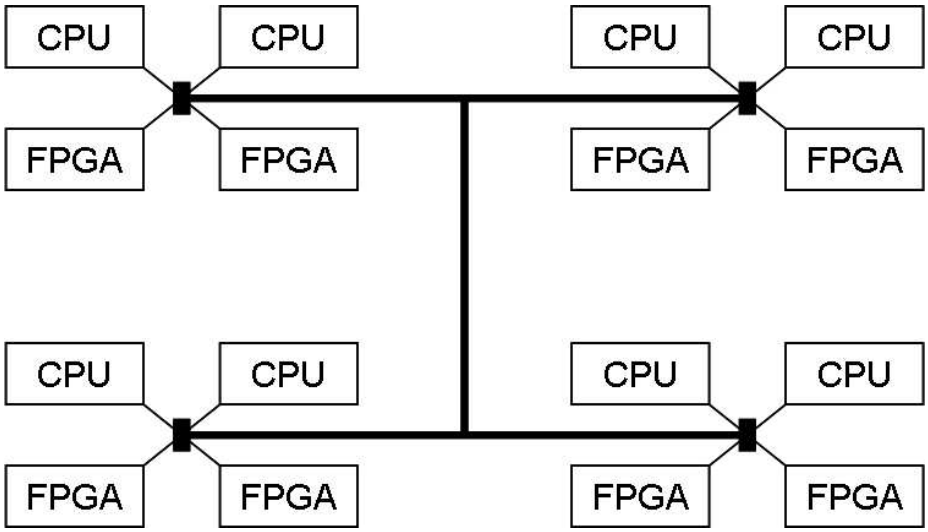


Figure 5.1: Hypothetical four-node distributed memory parallel computing system.

An interesting research project would be to implement a HAPFFT on a parallel computer. Researchers are investigating the incorporation of FPGA technology into

traditional supercomputers, as a way of increasing performance. This is done by including the FPGAs as a coprocessor to the microprocessor-based computing nodes. The FPGA can then be used to provide high-performance, custom functional units.

Consider the hypothetical parallel computer in Figure 5.1. It shows a four node distributed memory system. Each node consists of two CPUs, two FPGAs, and shared RAM (not shown). The HAPFFT topology shown in Figure 4.5(d) could be mapped to this system. The FEPEs can be implemented in the FPGAs, and the CPUs are then used to implement the BEPEs.

The result would be a high-performance FFT implementation that could be easily adapted to operate in harmony with other supercomputer applications. Additionally, it would scale much better than a purely hardware-based implementation. This is because the CPUs can handle different sized FFTs without having to reconfigure the FPGAs.

Other variations on the same theme could be imagined. The project would have to be modified to fit the topology of the particular parallel computing system, and the actual implementation would depend on how closely coupled the FPGAs are to the CPUs.

A second research direction would be to look at the Walsh-Hadamard transform as a candidate target for the HAPFFT. The WHT is an orthogonal transform very similar to the DFT. The DFT uses complex sinusoids as its basis functions. In contrast, the WHT uses one of the three Walsh function sequences. The WHT has applications in image processing, ultra-wideband communications systems, and pseudo-noise signal detection and measurement.

The WHT can be computed using the fast Walsh-Hadamard transform (FWHT), an algorithm very similar to the FFT. The dfg of the FWHT is almost identical to that of the FFT. The primary difference is that the FWHT does not require twiddle factor multiplications. Thus, it is computationally much cheaper than the FFT.

Since the FWHT is so similar to the FFT, the HAPFFT would be able to be adapted in a similar fashion to create a high-performance architecture for the FWHT. I have conducted initial experiments in which I have implemented and simulated a

FWHT pipeline similar to the R2SDF FFT pipeline. By adding a front-end processing element, and incorporating these FWHT pipelines, it would be possible to create a parallel FWHT architecture.

What makes an FWHT version of the HAPFFT so interesting is that that the WHT does not require twiddle factor multiplications. I've shown that, in the case of the HAPFFT, superlinear speedup was obtained for all major hardware elements, with the sole exception of the twiddle factor multipliers. These produced linear speedup. But, by implication, the FWHT would be able to achieve an even higher degree of superlinear speedup than the FFT version of the HAPFFT, since the linearly increasing multipliers would no longer skew the results.

There may be other orthogonal transforms to which the HAPFFT could be applied. The butterfly network encountered in the FFT is a common dfg form found in many algorithms. The HAPFFT could possibly be used for some of these.

Appendix

Appendix A

Implementation Details of the HAPFFT

A number of HAPFFT implementations were created using JHDL. The modules were simulated and verified. Also, the Xilinx ISE 6.1 tools were used to generate bitstreams, with the purpose of evaluating the resource usage of the implementations, and predicting maximum clock frequencies. We will cover the major implementation details of the the modules here.

The HAPFFT implementations all use the Radix-2^2[10] pipeline FFT for the back-end processing elements. It makes efficient use of chip multipliers and memory, and has extremely simple control. This algorithm will be examined in more detail subsequently.

Another important design consideration is the data representation for internal arithmetic. Fixed point is the simplest, permitting small, fast arithmetic units. But fixed point has a small dynamic range, and overflow can be a problem. Floating point is a good alternative for applications requiring high precision because of its large dynamic range. But floating point hardware is also expensive. A good middle-ground between fixed point and floating point is block floating point (BFP). This technique is a hybrid of fixed- and floating-point. It allows arithmetic to be implemented in a fixed point format, but has a larger dynamic range. The HAPFFT is implemented in both fixed-point and convergent block floating-point versions (CBFP) [4], a variation of BFP with a specific application to the FFT.

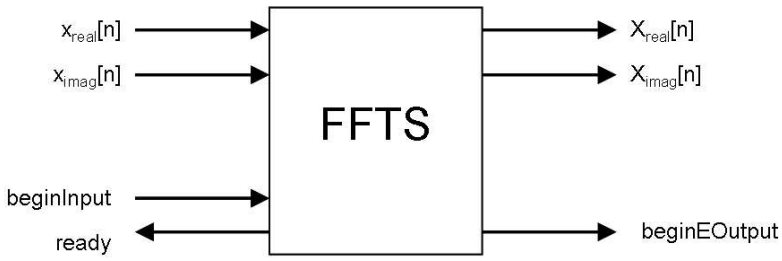


Figure A.1: Pinout for fixed-point Radix-2^2 FFT

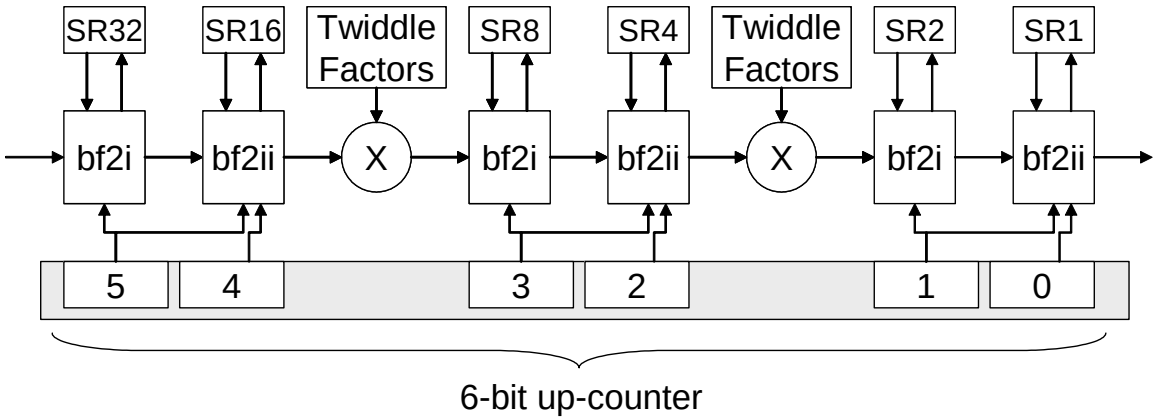


Figure A.2: 64-point fixed-point Radix-2^2 FFT

A.1 The Fixed Point FFT

Figure A.1 shows the pin-out for a single-pipeline fixed-point FFT based on the Radix-2^2 algorithm. The complex data is fed in one data-point per clock cycle. **ready** is asserted when the module is ready to accept a new input data stream. The **beginInput** signal is asserted the clock cycle previous to presenting the first data-point, and can be asserted the clock cycle following the assertion of **ready**. The **beginEOutput** signal will be asserted the clock cycle prior to the first output data-point appearing.

Figure A.2 is a block diagram of a 64-point Radix-2^2 fixed-point FFT example. The module consists of six radix-2 butterflies, shift registers associated with each

butterfly, two complex multipliers, two twiddle factor generators, and a simple 6-bit counter that provides the control signals. The shift registers vary in length from 1- to 32-bits, and are labeled accordingly.

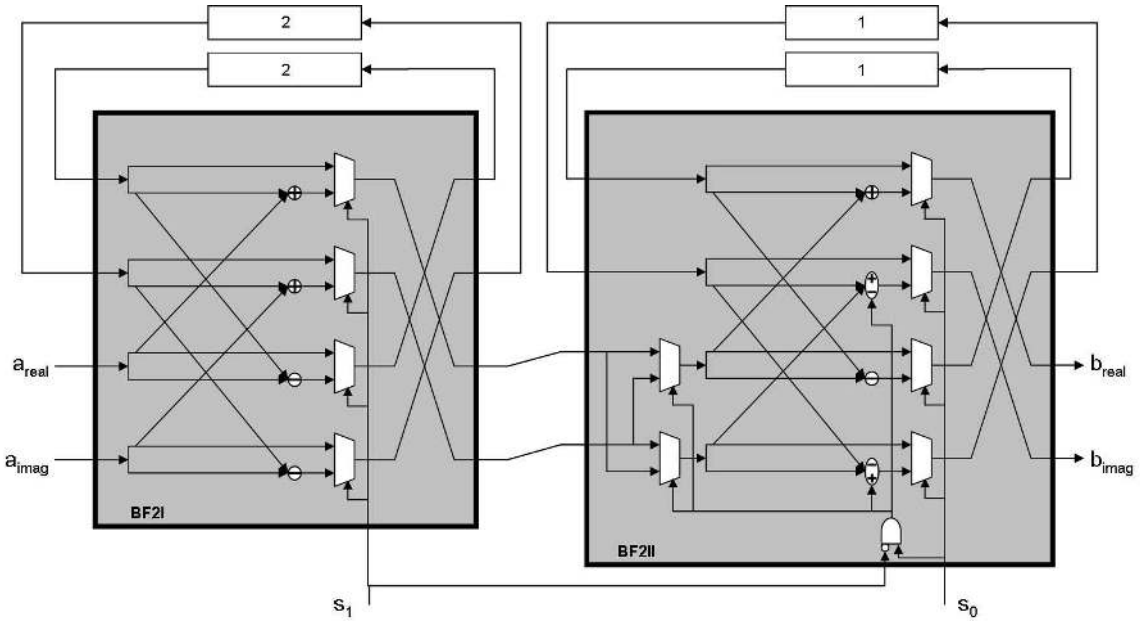


Figure A.3: bf2i and bf2ii details

A.1.1 Butterfly Operation

Each group of two butterflies, consisting of a **bf2i** and a **bf2ii**, together emulate a radix-4 butterfly. Figure A.3 shows the internals of each and how they are connected together.

These modules operate on a principal known as Single-path Delay Feedback (SDF) [27]. The FFT Radix-2 butterfly must have two inputs in order to produce the next FFT intermediate value, but the data in our scenario is available only in a serial mode. The SDF mechanism provides a solution where the first input is delayed until the second input is presented, after which the calculation can proceed. Both

the **bf2i** and **bf2ii** modules accomplish this by multiplexing the first input to a shift-register of sufficient length so that that data-point is present at the butterfly input when the second data-point appears. A counter provides the control signals for these multiplexers, which are internal to the butterfly modules.

The counter additionally provides signals to the **bf2ii** for switching the adder operations, and swapping the real and complex input wires. These mechanisms effect a multiplication of the input by $\sqrt{-1}$.

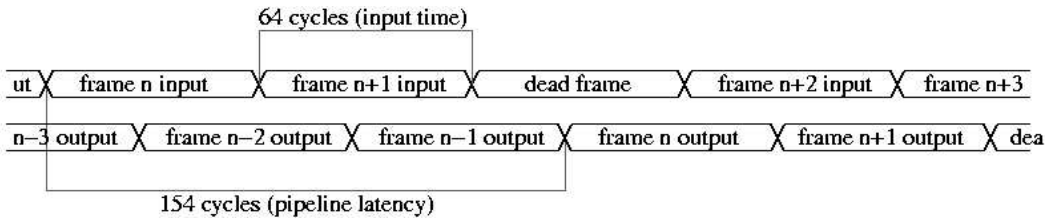


Figure A.4: 64-point FFT Pipeline Latency, 18-bit data

A.1.2 Timing Behavior

The latency of the fixed-point FFT pipeline, without additional pipeline registers, is equal to (N-1), where N is the frame size. In order to decrease the minimum clock period, and thereby increase throughput, pipeline registers are needed at strategic points. This usually occurs after components with large combinational logic delays, such as multipliers, or large multiplexers. With the addition of these pipeline registers the latency is increased slightly. Also, a required unscramble buffer at the output adds an additional N latency, so the actual latency for the single-pipeline fixed-point FFT is generally between 2N and 2.5N clock cycles.

Figure A.4 shows the timing for a 64-point FFT with 18-bit wires. The total latency between the time the first sample is input until the first result sample appears at the output is 154 clock cycles. Also, 64 clock cycles are required to input the 64

samples of a frame, and likewise 64 cycles for reading the output. For maximum throughput, the input frames must be adjacent, without any dead cycles between them. If this does occur, the FFT pipeline must be allowed time to clear and reset. This time is equal to the frame size, or 64 cycles in our example. Figure A.4 shows an example dead frame inserted between normal input frames.

A.1.3 Overflow Handling and Data-Scaling

In order to avoid overflow, the data set is scaled down as it propagates through the pipeline. The FFT operation consists of a long series of summations, and thus either the dynamic range of the numerical presentation must be large (floating-point of block floating-point), or the numerical data must be scaled down. Since the module is fixed point, the latter strategy is used. The scaling is implemented in the following manner:

The FFT is divided into segments each consisting of a **bf2i**, **bf2ii** and a complex multiplier. If the input wires of each segment are of width w, then they will be given two guard bits at the MSB in order to accommodate any overflow during the computations of the segment, making the internal data width for each segment $w+3$. After the computation, the segment will truncate two bits off the LSB of the data, and the remaining w LSBs will be sent to the next segment in the pipeline.

As indicated, the bits to be dropped from scaling are truncated. A rounding scheme can also be used, and this would prevent a drift in the DC offset of the output. But this would require an additional adder at the output of each stage, and has not been implemented.

A.2 Block Floating-Point FFT

The block floating-point architecture is a variation on the fixed-point architecture. The basic idea behind block floating-point (BFP) is to execute computations on blocks of data, each having an exponent assigned, where all data in the block is normalized to the exponent. This is similar to a typical floating-point scheme, in that a datum is represented by an exponent and a mantissa. But with BFP only one

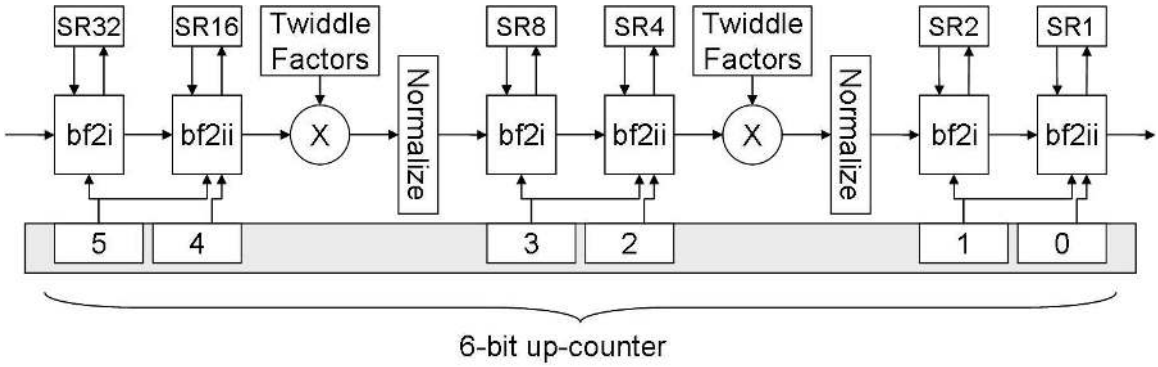


Figure A.5: 64-point Single-pipeline Block Floating-point Radix-2^2 FFT

exponent is stored for the whole block of data. Because all data in a block share the same exponent, operations can be done in fixed point. At the end of the stage the data will then be renormalized to the largest value in the block, and the exponent updated.

The advantage of BFP is the great savings in hardware from doing fixed-point arithmetic, without sacrificing the dynamic range advantages of floating-point. The disadvantages are the loss of precision from sharing one exponent among multiple data points, and the increased computing resources needed, though these are not as large as those needed by floating point.

The use of BFP in pipeline circuits introduces some unique problems, since the first result of a given block will proceed to the next pipeline element before the succeeding results have appeared. Renormalizing the data becomes difficult. It would be necessary to buffer the whole data block to memory before normalizing it, leading to large chip resource demands.

In response to the excessive memory needed by classical block floating point in pipelined circuits, convergent block floating point (CBFP) [4] has been proposed. CBFP is a design technique based on the observation that as the blocks of data proceed through the FFT pipeline, the data interdependencies are successively partitioned into smaller and smaller independent blocks. What this means for the designer

is that it is no longer necessary to wait for the whole block of data before renormalizing. In the case of a radix-4 algorithm, the output blocks of each dfg column can be split into four sub-blocks. Each sub-block then gets its own exponent. By the end of the pipeline, the block size has converged to unity, which is effectively floating point.

Not only does CBFP save memory, there is also an increased precision over classical block floating point because of the smaller blocks.

The radix-2^2 block floating-point architecture is similar to the fixed-point FFT, and a block diagram of it is found in Figure A.5. The FFT pipeline is segmented into units consisting of a **bf2i**, **bf2ii** and a multiplier. After each unit a buffer is included for normalizing the data. Because of the serial nature of the pipeline, it is necessary to buffer up all results for a given block before normalizing. Also, logic blocks are added for converting back-and-forth between IEEE 32-bit floating point.

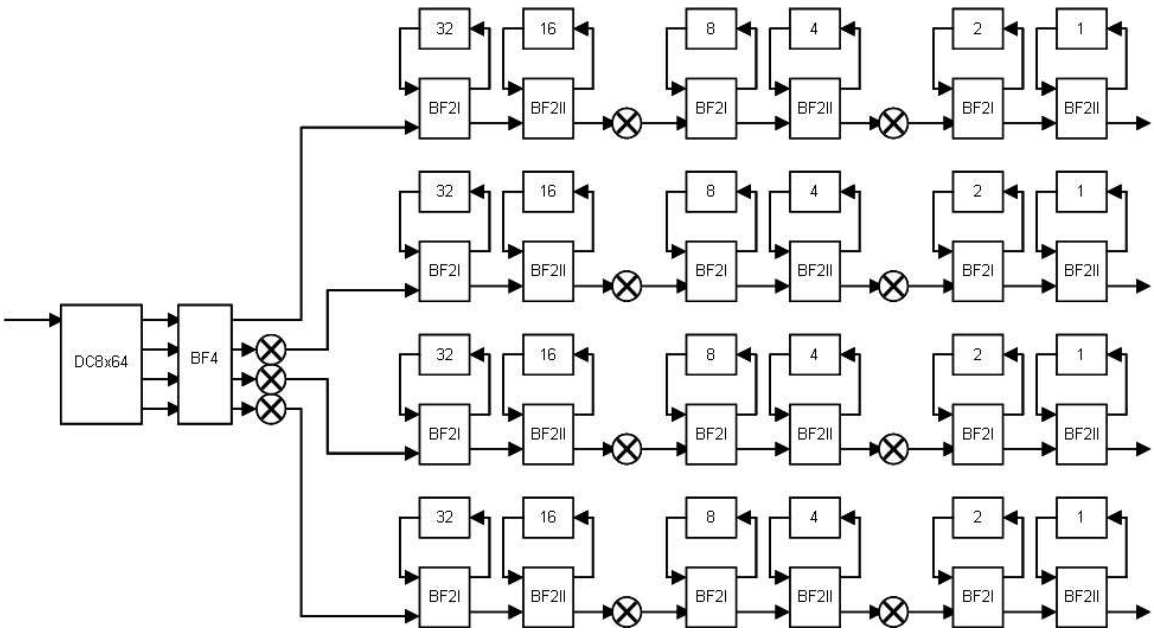


Figure A.6: 256-point Quad-pipeline fixed-point Radix-2^2 HAPFFT

A.3 The HAPFFT Implementation

Either the fixed-point or CBFP radix-2^2 pipelines can be incorporated into the HAPFFT. In the case of the CBFP version, additional resources may be needed for renormalizing data after the front-end processing unit, but it is not required.

Figure A.6 shows the block diagram for an implementation of a 256-point quad-pipelined fixed-point radix-2^2 HAPFFT. It consists of four independent pipelines, each fed by the radix-4 butterfly. The pipelines each use the same control signals and twiddle-factors. Also, latency is substantially reduced when compared to the single-pipeline modules, by about a factor of four.

Appendix B

Parallel Efficiency of the Binary-Exchange and Transpose Algorithms

An important criteria for evaluating parallel processing algorithms is what is known as *efficiency*. Efficiency is a measure of how well a processor is utilized; the percentage of program execution in which it is doing useful work. It is defined as the fraction of the speedup caused by using parallel processors, versus the number of processors. The *efficiency threshold* is the level of efficiency above which it is difficult improve.

By adding additional processors to a parallel system, parallel programs can achieve increased speedup. But for typical problems the efficiency will decrease if the *problem size* is kept constant. This is because the processing overhead for each processor increases, typically because of the need to pass data back-and-forth between more processors.

The problem size is a measure of the size and complexity of a given implementation of some parallel algorithm. It can be defined as the number of basic operations required to execute a program on a given data set. For example, for a data set of size n, the FFT would have a problem size of $O(n \log n)$. Most useful algorithms have a problem size that depends on the input data set.

Another evaluation criteria is the *isoefficiency function*. The isoefficiency function is a functional relationship between the number of processors and the problem

size. If additional processors are added to a parallel system, the isoefficiency function defines the amount the problem size must grow in order to maintain a constant efficiency.

For a number of processors p, arrayed as a hypercube network, the binary-exchange algorithm has an isoefficiency function of $O(p \log p)$. For the transpose algorithm, operating on the same system, the isoefficiency function is $O(p^2 \log p)$. But the transpose algorithm has a higher efficiency threshold than the binary-exchange algorithm, and will scale better for efficiency levels above the binary-exchange algorithm's threshold.

Another drawback of the binary-exchange algorithm is that it requires a hypercube network for good efficiency. Hypercube networks are relatively high-bandwidth, and the binary-exchange algorithm takes a big performance hit on other network architectures. For example, in a mesh network the binary-exchange algorithm's isoefficiency function is $O(2^{\sqrt{p}}\sqrt{p})$. In contrast, the transpose algorithm will scale well on other network architectures besides hypercubes.

Bibliography

[1] P. Bellows and B. L. Hutchings. JHDL - an HDL for reconfigurable systems. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 175–184, Napa, CA, Apr. 1998.

[2] G. D. Bergland and D. E. Wilson. A fast Fourier transform algorithm for a global, highly parallel processor. *IEEE Transactions on Audio and Electroacoustics*, AU-17:125–127, 1969.

[3] G. Bi and E. V. Jones. A Pipelined FFT Processor for Word-Sequential Data. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(12):1982–1985, dec 1989.

[4] E. Bidet, D. Castelain, C. Joanblanq, and P. Senn. A Fast Single-Chip Implementation of 8192 Complex Point FFT. *IEEE Journal of Solid-State Cicuits*, 30(3), mar 1995.

[5] T. Chen, G. Sunada, and J. Jin. COBRA: A 100-MOPS single-chip programmable and expandable FFT. *IEEE Transactions on VLSI Systems*, 7(2), jun 1999.

[6] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[7] A. Despain. Fourier Transform Computer Using CORDIC Iterations. *IEEE Transaction on Computers*, pages 993–1001, oct 1974.

[8] S. F. Gorman and J. M. Wills. Partial Column FFT Pipelines. *IEEE Transactions on Circuits and Systems*, 42(6):414–423, 1995.

[9] A. Gramam, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Pearson Education, second edition, 2003.

[10] S. He and M. Torkelson. A New Approach to Pipeline FFT Processor. In *The 10th International Parallel Processing Symposium(IPPS)*, pages 766–770, 1996.

[11] S. Horiguchi and X. X. Zhang. WSI Architecture of FFT. In *Proceedings of the 4th International Conference on Wafer Scale Integration*, pages 45–54, 1992.

[12] R. G. Lyons. *Understanding Digital Signal Processing*. Prentice-Hall PTR, 2001.

[13] Y. T. Ma. A VLSI-Oriented Parallel FFT Algorithm. *IEEE Transactions on Signal Processing*, 44(2), feb 1996.

[14] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[15] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete-Time Signal Processing, Second Edition*. Pearson Education, 1999.

[16] J. Palmer and B. Nelson. A Parallel FFT Architecture for FPGAs. In *FPL 2004*, pages 948–953, 2004.

[17] M. C. Pease. An adaptation of the fast fourier transform for parallel processing. *Journal of the ACM*, 15(2):252–264, 1968.

[18] Y. Peng. A Parallel Architecture for VLSI Implementation of FFT Processor. In *Proceedings of ASIC 2003*, volume 2, pages 748 – 751, 2003.

[19] Pentek. FPGA IP CORE, Model 4954-404: Ultra-High-Speed 4096-Point Fast Fourier Transform(FFT). Technical report, Pentek, Inc., 2003.

[20] A. A. Petrovsky and S. L. Shkredov. Multi-pipeline Implementation of Real-Time Vector DFT. In *Proceedings of the EUROMICRO Systems on Digital System Design*, pages 326–333, 2004.

[21] L. R. Rabiner and B. Gold. *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975.

[22] SiWorks. Product Brief: Parallel N-Point FFT/IFFT Core. Technical report, SiWorks, Inc., 2003.

[23] W. W. Smith and J. M. Smith. *Handbook of Real-Time Fast Fourier Transforms*. IEEE, 1995.

[24] E. E. Swartzlander. *VLSI Signal Processing Systems*. Kluwer Academic Publishers, 1986.

[25] E. E. Swartzlander, W. K. W. Young, and S. J. Joseph. A Radix-4 Delay Commutator for Fast Fourier Transform Processor Implementation. *IEEE J. Solid-State Circuits*, 19(5):702–709, 1984.

[26] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In P. Athanas and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 99–107, Napa, CA, Apr. 1995.

[27] E. Wold and A. Despain. Pipeline and parallel-pipeline FFT processors for VLSI implementation. *IEEE Transactions on Computers*, pages 414–426, may 1984.

[28] Xilinx. Xilinx LogiCORE Fast Fourier Transform v2.1. Technical report, Xilinx, Inc., 2003.

[29] C. H. Yeh and B. Parhami. A Class of Parallel Architectures for Fast Fourier Transform. In *IEEE 39th Symposium on Circuits and Systems*, volume 2, pages 856–859, 1997.

[30] G. Zhang and F. Chen. Parallel FFT with CORDIC for Ultra wide band. In *Proceedings of PIMRC 2004*, volume 2, pages 1173 – 1177, 2004.

[31] K. Zhong, H. H., and G. Zhu. An Ultra High-Speed FFT Processor. In *International Symposium on Signals, Circuits and Systems*, volume 1, pages 37–40, 2003.