AN APPROACH TO LOW-POWER, HIGH-PERFORMANCE, FAST FOURIER TRANSFORM PROCESSOR DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Bevan M. Baas
February 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

G. Leonard Tyler
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Oyekunle A. Olukotun

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Teresa H. Meng

Approved for the University Committee on Graduate Studies:

Abstract

The Fast Fourier Transform (FFT) is one of the most widely used digital signal processing algorithms. While advances in semiconductor processing technology have enabled the performance and integration of FFT processors to increase steadily, these advances have also caused the power consumed by processors to increase as well. This power increase has resulted in a situation where the number of potential FFT applications limited by maximum power budgets—not performance—is significant and growing.

We present the *cached-FFT* algorithm which explicitly caches data from main memory using a much smaller and faster memory. This approach facilitates increased performance and, by reducing communication energy, increased energy-efficiency.

Spiffee is a 1024-point, single-chip, 460,000-transistor, 40-bit complex FFT processor designed to operate at very low supply voltages. It employs the cached-FFT algorithm which enables the design of a well-balanced, nine-stage pipeline. The processor calculates a complex radix-2 butterfly every cycle and contains unique *hierarchical-bitline* SRAM and ROM memories which operate well in both standard and low supply voltage, low threshold-voltage environments. The processor's substrate and well nodes are connected to chip pads, accessible for biasing to adjust transistor thresholds.

Spiffee has been fabricated in a standard 0.7 μm ($L_{poly} = 0.6\,\mu$m) CMOS process and is fully functional on its first fabrication. At a supply voltage of 1.1 V, Spiffee calculates a 1024-point complex FFT in 330 μsec, while dissipating 9.5 mW—resulting in an adjusted energy-efficiency more than 16 times greater than that of the previously most efficient FFT processor. At a supply voltage of 3.3 V, it operates at 173 MHz—a clock rate 2.6 times faster than the previously fastest.

Acknowledgments

As with most large projects, this research would not have been possible without considerable guidance and support. I would like to acknowledge those who have enabled me to complete this work and my years of graduate study.

To Professor Len Tyler, my principal adviser, I am deeply indebted. I have learned a great deal from him in technical matters, in improving my written communication skills, and in a wide variety of areas through his consistent mentoring as my Ph.D. adviser.

I thank Professor Kunle Olukotun for serving as my associate adviser, for encouragement and advice, and for many enlightening discussions.

To Professor Teresa Meng, who also served as my adviser during my M.S., I express my sincere gratitude for the valuable guidance she has provided.

During my first several years of graduate school, I worked under the supervision of the late Professor Allen Peterson. He introduced me to the study of high-performance digital signal processors and developed my interest in them. He is sorely missed.

I especially thank Professors Tyler, Olukotun, and Meng for serving as readers of this dissertation. I also thank Professors Tyler, Olukotun, Don Cox, and Thomas Cover for serving on my Oral Examination Committee.

I am deeply indebted to Jim Burr who first introduced me to the study of FFT processors, taught me much about low-power computation, and mentored me on a wide variety of research issues. Masataka Matsui answered countless circuit and layout questions, and I thank him for passing along some of his knowledge. Other colleagues in the ULP group have helped me tremendously at various times. I thank Yen-Wen Lu for cheerfully helping when asked, Vjekoslav Svilan for frequent assistance and for being a great golf partner, and Gerard Yeh for many valuable discussions. Other members of STARLab and the EE Department from whose interactions I have benefitted significantly include: Mark Horowitz, Ivan Linscott, Ely Tsern, Jim Burnham, Kong Kritayakirana, Jawad Nasrullah,

Snežana Maslaković, Weber Hoen, Karen Huyser, Edgar Holmann, Mitch Oslick, Dan Weinlade, Birdy Amrutur, and Gu-Yeon Wei. I am grateful for the administrative assistance given by Doris Reed, Marli Williams, and Sieglinde Barsch.

In addition, I would like to acknowledge those who have supported my work financially. My graduate education was largely supported by: a National Science Foundation Fellowship, a NASA Graduate Student Research Program Fellowship, and a GE Foundation Scholarship through the American Indian Science and Engineering Society. Additional support was provided by a Stanford University School of Engineering Fellowship.

I thank MOSIS for providing the fabrication of the *Spiffee1* chip, and Carl Lemonds of Texas Instruments for arranging the fabrication of the three low-V_t chips described in Ch. 6. I am also indebted to Michael Godfrey for allowing me the generous use of his test equipment, James Pak for the high-magnification die microphotographs of Ch. 5, and Sabina Chen and Karletta Chief for editing assistance.

On a personal level, my deepest thanks go to my parents for their unwavering love and encouragement. *Ahéhee' shi yaa hłiniil'a dóó shíka' ayi' nołchíí'. Ntsaago nich'į' baa ahéédh nisin dóó ayóó' anóshní.* I also thank my sister Janet, brother-in-law Ivan, brother Vern, and sister-in-law Karen for their prodding questions and encouragement that helped keep me motivated through my many years of graduate school.

I'm thankful too for special people I've been blessed to have in my life; among them are: Bonnie Arnwine, Jennifer Thenhaus, Laurie Wong, Kim Greenwaldt, John Chan, Jon Stocker, Eddy Chee, Eugene Chi, Jason Swider, Crystie Prince, Debbie Tucker, Amy Duncan, and Anne Duncan.

Above all, I thank my God and Creator—who gave me the opportunity to undertake this work, and who, long ago, designed an ordered universe in which FFTs could be studied.

Contents

List of Tables

List of Figures

List of Symbols

A	butterfly input, arbitrary integer.
a	activity of a node.
B	butterfly input, arbitrary integer.
b_k	k^{th} bit within the *butterfly* counter.
C	size of a cache memory (words), butterfly input, arbitrary integer, capacitance (F).
C_{load}	load capacitance (F).
c	stages in an RRI-FFT, $\log_r C$.
D	butterfly input, arbitrary integer.
E	number of epochs in a cached-FFT, energy (J).
$F(\cdot)$	the continuous Fourier transform of the function $f(\cdot)$, if it exists.
f	clock frequency (Hz).
$f(\cdot)$	an arbitrary, complex function.
Gnd	ground reference potential of a circuit.
g_k	k^{th} bit within the *group* counter.
$H(\cdot)$	Heaviside step function.
I	current (A).
I_{ds}	current flowing from the drain to the source of a MOS transistor (A).
$I_{leakage}$	I_{ds} for a MOSFET with $V_{gs} = 0$ (A).
I_{off}	I_{ds} for a MOSFET with $V_{gs} = 0$ (A).
I_{on}	I_{ds} for an NMOS device with $V_{gs} = V_{dd}$ or a PMOS device with $V_{gs} = -V_{dd}$ (A).
i	$\sqrt{-1}$.
$\Im\{\cdot\}$	imaginary component of its argument.
k	integer index, arbitrary integer.
k_1, k_2	integer indexes.
k_e	processor energy coefficient (μJ/V^2).

k_f	processor clock frequency coefficient (MHz/V).
L	inductance (H), words in a memory.
L_{poly}	minimum gate length or polysilicon width (μm).
l	arbitrary integer.
M	$\log_r N$, data sequence length.
m	arbitrary integer.
N	length of the input and output sequences of a DFT or FFT.
N'	a particular value of N.
N_1, N_2	factors of N.
n	integer index.
n_1, n_2	factors of N, integer indexes.
n_{even}	even-valued index.
n_{odd}	odd-valued index.
$\mathcal{O}(\cdot)$	"on the order of."
P	power (W), number of passes in a cached-FFT.
p	small prime number, arbitrary real number.
Q	electrical charge (coulombs).
R	resistance (ohms).
r	radix of an FFT decomposition, arbitrary real number.
$\Re\{\cdot\}$	real component of its argument.
S	segments in a hierarchical-bitline memory.
s	continuous, real, independent variable of $F(\cdot)$; stage of an FFT.
T	time (sec).
t	time (sec), thickness of a material (m).
V	voltage (V), butterfly output.
V_{bs}	body voltage of a MOS transistor with respect to its source (V).
V_{dd}	supply voltage of a circuit (V).
V_{ds}	drain voltage of a MOS transistor with respect to its source (V).
V_{gs}	gate voltage of a MOS transistor with respect to its source (V).
V_{in}	input voltage of a circuit (V).
$V_{n\text{-}well}$	n-well voltage (V).
V_{out}	output voltage of a circuit (V).
$V_{p\text{-}substrate}$	p-substrate voltage (V).

V_{pulse}	pulsed voltage source voltage (V).
$V_{p\text{-}well}$	p-well voltage (V).
V_{sb}	source voltage of a MOS transistor with respect to its body (V).
V_{swing}	voltage difference between logic "0" and logic "1" values (V).
V_t	threshold-voltage of a MOS transistor (V).
$V_{t\text{-}nmos}$	V_t of an NMOS transistor (V).
$V_{t\text{-}pmos}$	V_t of a PMOS transistor (V).
W	butterfly output.
W_N	DFT kernel, $e^{-i2\pi/N}$; also called the "N^{th} root of unity."
X	butterfly output.
$X(k)$	output sequence of a forward DFT, defined for $k = 0, 1, \ldots, N-1$.
x	continuous, real, independent variable of $f(\cdot)$; arbitrary real number.
$x(n)$	input sequence of a forward DFT, defined for $n = 0, 1, \ldots, N-1$.
$x_{\text{even}}(\cdot)$	sequence consisting of the even members of $x(\cdot)$.
$x_{\text{odd}}(\cdot)$	sequence consisting of the odd members of $x(\cdot)$.
x^*	complex conjugate of x.
$\hat{X}, \hat{x}$	intermediate values in the calculation of an FFT or IFFT.
Y	butterfly output.
y	arbitrary real number.
Z	intermediate butterfly value.
z	arbitrary real number.
α	fabrication technology scaling factor, arbitrary integer.
β	arbitrary integer.
γ	arbitrary integer.
δ	arbitrary integer.
$\delta(\cdot)$	Dirac delta "function."
ϵ	permittivity (F/cm).
ϵ_0	permittivity of vacuum, 8.854×10^{-14} F/cm.
ϵ_r	relative permittivity.
θ	arbitrary angle.
λ	scalable unit of measure commonly used in chip designs (μm).
$\phi, \overline{\phi}$	clock signals.

Chapter 1

Introduction

The Discrete Fourier Transform (DFT) is one of the most widely used digital signal processing (DSP) algorithms. DFTs are almost never computed directly, but instead are calculated using the Fast Fourier Transform (FFT), which comprises a collection of algorithms that efficiently calculate the DFT of a sequence. The number of applications for FFTs continues to grow and includes such diverse areas as: communications, signal processing, instrumentation, biomedical engineering, sonics and acoustics, numerical methods, and applied mechanics.

As semiconductor technologies move toward finer geometries, both the available performance and the functionality per die increase. Unfortunately, the power consumption of processors fabricated in advancing technologies also continues to grow. This power increase has resulted in the current situation, in which potential FFT applications, formerly limited by available performance, are now frequently limited by available power budgets. The recent dramatic increase in the number of portable and embedded applications has contributed significantly to this growing number of power-limited opportunities.

1.1 Research Goals

The goal of this research is to develop the algorithm, architectures, and circuits necessary for high-performance and energy-efficient FFT processors—with an emphasis on a VLSI implementation. A suitable solution achieves the following sub-goals:

High Energy-Efficiency

Power by itself is an inadequate design goal since the power of a processor can easily be decreased by reducing its workload. Energy consumption is a measure of the cost per work performed, and therefore provides a more useful "low-power" design goal.

High Throughput

While the calculation of results with low *latency* is desirable for both general-purpose and DSP processors, processing data with high *throughput* is much more important for nearly all DSP processor applications. Therefore, this research will emphasize high data throughput at the expense of latency, when necessary.

VLSI Suitability

The signal processing literature contains a large number of FFT algorithms. Nearly all authors evaluate algorithms based on simple measures, such as the number of multiplications, or the number of multiplications plus additions. This dissertation argues that for the VLSI implementation of an FFT processor, other measures, such as complexity and regularity, are at least as important as the number of arithmetic operations.

Simple designs permit straightforward implementations, require less design effort, and typically contain fewer errors, compared to complex designs. In a commercial setting, simple designs reduce time-to-market. Design time no longer required to manage complexity can be used to further optimize other measures such as performance, power, and area.

Regular designs have few constituent components and are largely built-up by replicating several basic building blocks. Regular implementations gain many of the same benefits enjoyed by simple designs.

1.2 Synopsis

Contributions

The principal contributions of this research are:

1. Development of a new FFT algorithm, called the *cached-FFT* algorithm. This algorithm makes use of a cache memory and enables a processor to operate with high energy-efficiency and high speed.

2. Design of a wide variety of circuits which operate robustly in a low supply-voltage environment. The circuits were developed to function using transistors with either standard or low thresholds.

3. Design, fabrication, and characterization of a single-chip, cached-FFT processor. The processor has high energy-efficiency with good performance at low supply voltages, and high performance with good energy-efficiency at typical supply voltages.

These contributions are reviewed in the final chapter of this work.

Overview

Chapter 2 begins with an introduction of the continuous Fourier transform and the discrete Fourier transform. A derivation of the FFT is given; the remainder of the chapter presents an overview of the major FFT algorithms. Chapter 3 introduces the four primary power consumption mechanisms of CMOS circuits and reviews a number of popular power reduction techniques. Chapter 4 presents the cached-FFT algorithm. This algorithm is well-suited to high-performance, energy-efficient, hardware applications, as well as exhibiting potential on programmable processors. Chapter 5 describes *Spiffee*, a single-chip FFT processor design which uses the cached-FFT algorithm. Details of the design are given at the algorithmic, architectural, and physical levels, including circuits and layout. Chapter 6 presents performance data from *Spiffee1,* the first Spiffee processor. Spiffee1 was fabricated in a 0.7 μm CMOS process, and is fully functional. Finally, Chapter 7 summarizes the work presented and suggests areas for future work.

Chapter 2

The Fourier Transform

This chapter begins with an overview of the Fourier transform in its two most common forms: the continuous Fourier transform and the Discrete Fourier Transform (DFT). Many excellent texts (Bracewell, 1986; Oppenheim and Schafer, 1989; Roberts and Mullis, 1987; Strum and Kirk, 1989; DeFatta *et al.*, 1988; Jackson, 1986) exist on the topic of the Fourier transform and its properties, and may be consulted for additional details. The remainder of the chapter focuses on the introduction of a collection of algorithms used to efficiently compute the DFT; these algorithms are known as Fast Fourier Transform (FFT) algorithms.

2.1 The Continuous Fourier Transform

The continuous Fourier transform is defined in Eq. 2.1[1]. It operates on the function $f(x)$ and produces $F(s)$, which is referred to as the Fourier transform of $f(x)$.

$$F(s) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi xs} dx \tag{2.1}$$

The constant i represents the imaginary quantity $\sqrt{-1}$. The independent variables x and s are real-valued and are defined from $-\infty$ to ∞. Since the independent variable of f is often used to represent time, it is frequently called the "time" variable and denoted t. This title can be a misnomer since f can be a function of a variable with arbitrary units and is, in fact, commonly used with units of distance. Analogously, the independent variable of F

[1] Alternate, but functionally equivalent forms can be found in Bracewell (1986).

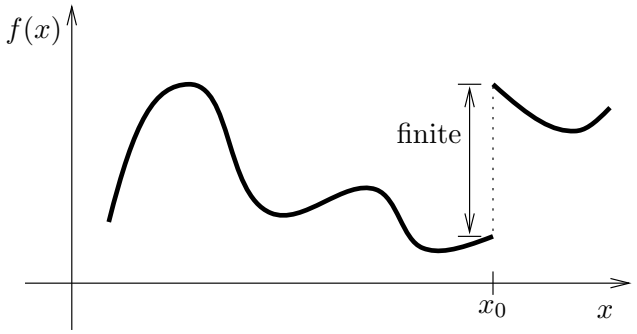


Figure 2.1: Example function with a finite discontinuity

is often denoted f and called the "frequency" variable; its units are the inverse of x's units. In general, the functions $f(x)$ and $F(s)$ are complex-valued.

Because of the way the Fourier integral is defined, not every function $f(x)$ has a transform $F(s)$. While necessary and sufficient conditions for convergence are not known, two conditions which are sufficient for convergence (Bracewell, 1986) are:

Condition 1

The integral of $|f(x)|$ from $-\infty$ to ∞ exists. That is,

$$\int_{-\infty}^{\infty} |f(x)|dx < \infty. \tag{2.2}$$

Condition 2

Any discontinuities in $f(x)$ are bounded. For example, the function shown in Fig. 2.1 is discontinuous at $x = x_0$, but only over a finite distance, so this function meets the second condition.

Because these conditions are only sufficient, many functions which do not meet these conditions nevertheless have Fourier transforms. In fact, such useful functions as $\sin(x)$, the Heaviside step function $\mathrm{H}(x)$, $\mathrm{sinc}(x)$, and the "generalized" Dirac delta "function" $\delta(x)$, fall into this category.

The counterpart to Eq. 2.1 is the *inverse Fourier transform*, which transforms $F(s)$ back into $f(x)$ and is defined as,

$$f(x) = \int_{-\infty}^{\infty} F(s)e^{i2\pi xs}ds. \tag{2.3}$$

While the generality of the continuous Fourier transform is elegant and works well for the study of Fourier transform theory, it has limited direct practical use. One reason for the limited practical use stems from the fact that any signal $f(x)$ that exists over a finite interval has a spectrum $F(s)$ which extends to infinity, and any spectrum that has finite bandwidth has a corresponding infinite-duration $f(x)$. Also, the functions $f(x)$ and $F(s)$ are continuous (at least stepwise) or analog functions, meaning that they are, in general, defined over all values of their independent variables (except at bounded discontinuities as described in Condition 2). Since digital computers have finite memory, they can neither store nor process the infinite number of data points that would be needed to describe an arbitrary, infinite function. For practical applications, a special case of the Fourier transform known as the *discrete Fourier transform* is used. This transform is defined for finite-length $f(x)$ and $F(s)$ described by a finite number of samples, and is discussed in the next section.

2.2 The Discrete Fourier Transform (DFT)

The discrete Fourier transform operates on an N-point sequence of numbers, referred to as $x(n)$. This sequence can (usually) be thought of as a uniformly sampled version of a finite period of the continuous function $f(x)$. The DFT of $x(n)$ is also an N-point sequence, written as $X(k)$, and is defined in Eq. 2.4. The functions $x(n)$ and $X(k)$ are, in general, complex. The indices n and k are real integers.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-i2\pi nk/N}, \quad k = 0, 1, \ldots, N-1 \tag{2.4}$$

Using a more compact notation, we can also write,

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \quad k = 0, 1, \ldots, N-1 \tag{2.5}$$

where we have introduced the term W_N,

$$W_N = e^{-i2\pi/N} \tag{2.6}$$

$$= \cos\left(\frac{2\pi}{N}\right) - i \sin\left(\frac{2\pi}{N}\right). \tag{2.7}$$

The variable W_N is often called an "N^{th} root of unity" since $(W_N)^N = e^{-i2\pi} = 1$. Another very special quality of W_N is that it is periodic; that is, $W_N^n = W_N^{n+mN}$ for any integer m. The periodicity can be expressed through the relationship $W_N^{n+mN} = W_N^n W_N^{mN}$ because,

$$\begin{aligned} W_N^{mN} &= \left(e^{-i2\pi/N}\right)^{mN}, \quad m = -\infty, \ldots, -1, 0, 1, \ldots, \infty && (2.8)\\ &= e^{-i2\pi m} && (2.9)\\ &= 1. && (2.10) \end{aligned}$$

In a manner similar to the inverse continuous Fourier transform, the *Inverse DFT (IDFT)*, which transforms the sequence $X(k)$ back into $x(n)$, is,

$$\begin{aligned} x(n) &= \frac{1}{N}\sum_{k=0}^{N-1} X(k)e^{i2\pi nk/N}, \quad n = 0, 1, \ldots, N-1 && (2.11)\\ &= \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{-nk}, \quad n = 0, 1, \ldots, N-1. && (2.12) \end{aligned}$$

From Eqs. 2.4 and 2.11, $x(n)$ and $X(k)$ are explicitly defined over only the finite interval from 0 to $N-1$. However, since $x(n)$ and $X(k)$ are periodic in N, (viz., $x(n) = x(n+mN)$ and $X(k) = X(k+mN)$ for any integer m), they also exist for all n and k respectively.

An important characteristic of the DFT is the number of operations required to compute the DFT of a sequence. Equation 2.5 shows that each of the N outputs of the DFT is the sum of N terms consisting of $x(n)W_N^{nk}$ products. When the term W_N^{nk} is considered a pre-computed constant, calculation of the DFT requires $N(N-1)$ complex additions and N^2 complex multiplications. Therefore, roughly $2N^2$ or $\mathcal{O}(N^2)$ operations[2] are required to calculate the DFT of a length-N sequence.

For this analysis, the IDFT is considered to require the same amount of computation as its forward counterpart, since it differs only by a multiplication of the constant $1/N$ and by a minus sign in the exponent of e. The negative exponent can be handled without any additional computation by modifying the pre-computed W_N term.

Another important characteristic of DFT algorithms is the size of the memory required for their computation. Using Eq. 2.5, each term of the input sequence must be preserved

[2]The symbol $\mathcal{O}$ means "on the order of"; therefore, $\mathcal{O}(P)$ means *on the order of P*.

until the last output term has been computed. Therefore, at a minimum, $2N$ memory locations are necessary for the direct calculation of the DFT.

2.3 The Fast Fourier Transform (FFT)

The fast Fourier transform is a class of efficient algorithms for computing the DFT. It always gives the same results (with the possible exception of a different round-off error) as the calculation of the direct form of the DFT using Eq. 2.4.

The term "fast Fourier transform" was originally used to describe the fast DFT algorithm popularized by Cooley and Tukey's landmark paper (1965). Immediately prior to the publication of that paper, nearly every DFT was calculated using an $\mathcal{O}(N^2)$ algorithm. After the paper's publication, the popularity of the DFT grew dramatically because of this new efficient class of algorithms.

2.3.1 History

Despite the fact Cooley and Tukey are widely credited with the discovery of the FFT, in reality, they only "re-discovered" it. Cooley, Lewis, and Welch (1967) report some of the earlier known discoverers. They cite a paper by Danielson and Lanczos (1942) describing a type of FFT algorithm and its application to X-ray scattering experiments. In their paper, Danielson and Lanczos refer to two papers written by Runge (1903; 1905). Those papers and lecture notes by Runge and König (1924), describe two methods to reduce the number of operations required to calculate a DFT: one exploits the *symmetry* and a second exploits the *periodicity* of the DFT kernel $e^{i\theta}$. In this context, symmetry refers to the property $\Re\{e^{i\theta}\} = \Re\{e^{-i\theta}\}$ and $\Im\{e^{i\theta}\} = -\Im\{e^{-i\theta}\}$; or simply $e^{i\theta} = (e^{-i\theta})^*$, where x^* is the complex conjugate of x. Exploiting symmetry allows the DFT to be computed more efficiently than a direct process, but only by a constant factor; the algorithm is still $\mathcal{O}(N^2)$. Runge and König briefly discuss a method to reduce computational requirements still further by exploiting the periodicity of $e^{i\theta}$. The periodicity of $e^{i\theta}$ is seen by noting that $e^{i\theta} = e^{i\theta+2\pi m}$ where m is any integer. By taking advantage of the periodicity of the DFT kernel, much greater gains in efficiency are possible, such that the complexity can be brought below $\mathcal{O}(N^2)$, as will be shown in Sec. 2.3.2. The length of transforms Runge and König worked with were relatively short (since all computation was done by hand); consequently, both methods brought comparable reductions in computational complexity

and Runge and König actually emphasized the method exploiting symmetry.

Fifteen years after Cooley and Tukey's paper, Heideman *et al.* (1984), published a paper providing even more insight into the history of the FFT including work going back to Gauss (1866). Gauss' work is believed to date from October or November of 1805 and, amazingly, predates Fourier's seminal work by two years.

2.3.2 Simple Derivation

This section introduces the FFT by deriving one of its simplest forms. First, the DFT, Eq. 2.5, and the definition of W_N, Eq. 2.7, are repeated below for convenience.

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \quad k = 0, 1, \ldots, N-1 \tag{2.13}$$

$$W_N = e^{-i2\pi/N} \tag{2.14}$$

For this illustration, N is chosen to be a power of 2, that is, $N = 2^m$, where m is a positive integer. The length N is therefore an even number, and $x(n)$ can be separated into two sequences of length $N/2$, where one consists of the even members of x and a second consists of the odd members. Splitting Eq. 2.13 into even-indexed and odd-indexed summations gives,

$$X(k) = \sum_{n_{\text{even}}=0}^{N-2} x(n) W_N^{nk} + \sum_{n_{\text{odd}}=1}^{N-1} x(n) W_N^{nk}. \tag{2.15}$$

If $2m$ is substituted for n in the even-indexed summation and $2m+1$ is substituted for n in the odd-indexed summation (with $m = 0, 1, \ldots, N/2-1$), the result is,

$$X(k) = \sum_{m=0}^{N/2-1} x(2m) W_N^{2mk} + \sum_{m=0}^{N/2-1} x(2m+1) W_N^{(2m+1)k} \tag{2.16}$$

$$= \sum_{m=0}^{N/2-1} x(2m) \left(W_N^2\right)^{mk} + \sum_{m=0}^{N/2-1} x(2m+1) \left(W_N^2\right)^{mk} W_N^k. \tag{2.17}$$

But W_N^2 can be simplified. Beginning with Eq. 2.14:

$$\begin{aligned} W_N^2 &= \left(e^{-i2\pi/N}\right)^2 && (2.18)\\ &= e^{-i2\pi 2/N} \\ &= e^{-i2\pi/(N/2)} \\ &= W_{N/2}. && (2.19) \end{aligned}$$

Then, Eq. 2.17 can be written,

$$\begin{aligned} X(k) &= \sum_{m=0}^{N/2-1} x(2m)W_{N/2}^{mk} + W_N^k \sum_{m=0}^{N/2-1} x(2m+1)W_{N/2}^{mk} && (2.20)\\ &= \sum_{m=0}^{N/2-1} x_{\text{even}}(m)W_{N/2}^{mk} + W_N^k \sum_{m=0}^{N/2-1} x_{\text{odd}}(m)W_{N/2}^{mk}, && (2.21)\\ & \quad k = 0, 1, \ldots, N-1 \end{aligned}$$

where $x_{\text{even}}(m)$ is a sequence consisting of the even-indexed members of $x(n)$, and $x_{\text{odd}}(m)$ is a sequence consisting of the odd-indexed members of $x(n)$. The terms on the right are now recognized as the $(N/2)$-point DFTs of $x_{\text{even}}(m)$ and $x_{\text{odd}}(m)$.

$$X(k) = \text{DFT}_{N/2}\left\{x_{\text{even}}(m), k\right\} \; + \; W_N^k \cdot \text{DFT}_{N/2}\left\{x_{\text{odd}}(m), k\right\} \tag{2.22}$$

At this point, using only Eqs. 2.22 and 2.13, no real savings in computation have been realized. The calculation of each $X(k)$ term still requires $2 \cdot \mathcal{O}(N/2) = \mathcal{O}(N)$ operations; which means all N terms still require $\mathcal{O}\left(N^2\right)$ operations. As noted in Sec. 2.2, however, the DFT of a sequence is periodic in its length ($N/2$ in this case), which means that the $(N/2)$-point DFTs of $x_{\text{even}}(m)$ and $x_{\text{odd}}(m)$ need to be calculated for only $N/2$ of the N values of k. To re-state this key point in another way, the $(N/2)$-point DFTs are calculated for $k = 0, 1, \ldots, N/2-1$ and then "re-used" for $k = N/2, N/2+1, \ldots, N-1$. The N terms of $X(k)$ can then be calculated with $\mathcal{O}\Big((N/2)^2\Big) + \mathcal{O}\Big((N/2)^2\Big) = \mathcal{O}\big(N^2/2\big)$ operations plus $\mathcal{O}(N)$ operations for the multiplication by the W_N^k terms, called "twiddle factors." The mathematical origin of twiddle factors is presented in Sec. 2.4.1. For large N, this $\mathcal{O}\big(N^2/2 + N\big)$ algorithm represents a nearly 50% savings in the number of operations required, compared to the direct evaluation of the DFT using Eq. 2.13. Figure 2.2 shows

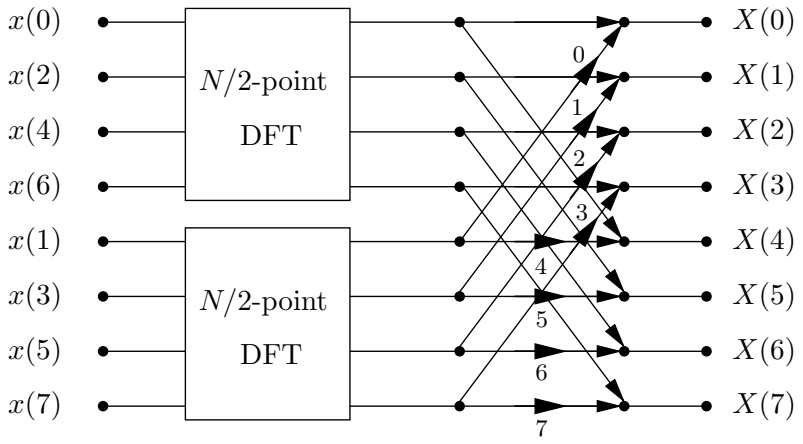


Figure 2.2: Flow graph of an 8-point DFT calculated using two $N/2$-point DFTs. Integers adjacent to large arrow heads signify a multiplication of the corresponding signal by W_8^k, where k is the given integer. Signals following arrows leading into a dot are summed into that node.

the dataflow of this algorithm for $N = 8$ in a graphical format. The vertical axis represents memory locations. There are N memory locations for the N-element sequences $x(n)$ and $X(k)$. The horizontal axis represents stages of computation. Processing begins with the input sequence $x(n)$ on the left side and progresses from left to right until the output $X(k)$ is realized on the right.

It is possible to reduce the N multiplications by $W_N^k, k = 0, 1, \ldots, N-1$ by exploiting the following relationship:

$$\begin{aligned} W_N^{x+N/2} &= W_N^x W_N^{N/2} && (2.23)\\ &= W_N^x \left(e^{-i2\pi/N}\right)^{N/2} && (2.24)\\ &= W_N^x e^{-i2\pi N/2N} && (2.25)\\ &= W_N^x e^{-i\pi} && (2.26)\\ &= -W_N^x. && (2.27) \end{aligned}$$

In the context of the example shown in Fig. 2.2 where $N = 8$, Eq. 2.27 reduces to $W_8^{x+4} = -W_8^x$ and allows the transformation of the dataflow diagram of Fig. 2.2 into the one shown in Fig. 2.3. Note that the calculations after the W_N multiplications are now 2-point DFTs.

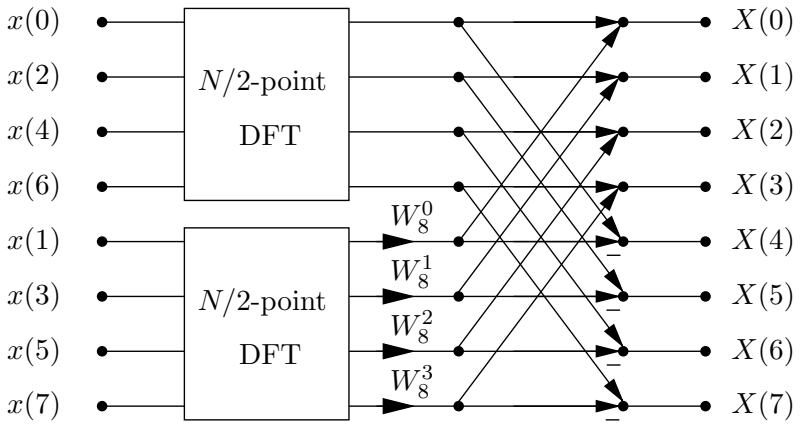


Figure 2.3: Flow graph of an 8-point DFT calculated using two $N/2$-point DFTs, with merged W_N coefficients. A minus sign $(-)$ adjacent to an arrow signifies that the signal is subtracted from, rather than added into, the node.

Since N was chosen to be a power of 2, if $N > 2$, both $x_{\text{even}}(m)$ and $x_{\text{odd}}(m)$ also have even numbers of members. Therefore, they too can be separated into sequences made up of their even and odd members, and computed from $N/2/2 = N/4$-point DFTs. This procedure can be applied recursively until an even and odd separation results in sequences that have two members. No further separation is then necessary since the DFT of a 2-point sequence is trivial to compute, as will be shown shortly. Figure 2.4 shows the dataflow diagram for the example with $N = 8$. Note that the recursive interleaving of the even and odd inputs to each DFT has scrambled the order of the inputs.

This separation procedure can be applied $\log_2(N) - 1$ times, producing $\log_2 N$ stages. The resulting m^{th} stage (for $m = 0, 1, \ldots, \log_2(N) - 1$) has $N/\left(2^{m+1}\right) \cdot 2^m = N/2$ complex multiplications by some power of W. The final stage is reduced to 2-point DFTs. These are very easy to calculate because, from Eq. 2.13, the DFT of the 2-point sequence $x\,(n) = \{x\,(0)\,, x\,(1)\}$ requires no multiplications and is calculated by,

$$X(0) = x(0) + x(1) \tag{2.28}$$

$$X(1) = x(0) - x(1). \tag{2.29}$$

Each stage is calculated with roughly $2.5N$ or $\mathcal{O}(N)$ complex operations since each

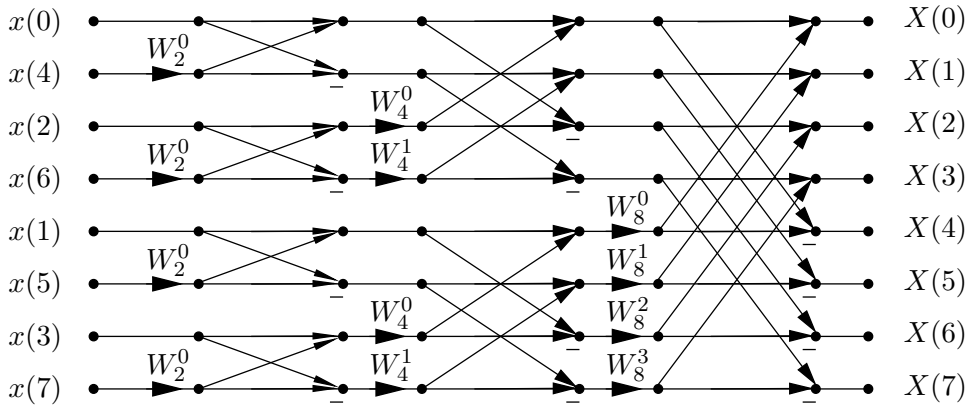


Figure 2.4: Flow graph of an 8-point radix-2 DIT FFT

of the $N/2$ 2-point DFTs requires one addition and one subtraction, and there are N/2 W_N multiplications per stage. Summing things up, the calculation of this N-point FFT requires $\mathcal{O}(N)$ operations for each of its $\log_2 N$ stages, so the total computation required is $\mathcal{O}\left(N \log_2 N\right)$ operations.

To reduce the total number of W coefficients needed, all W coefficients are normally converted into equivalent W_N values. For the diagram of Fig. 2.4, $W_4^0 = W_2^0 = W_8^0$ and $W_4^1 = W_8^2$; resulting in the common dataflow diagram shown in Fig. 2.5.

A few new terms

Before moving on to other types of FFTs, it is worthwhile to look back at the assumptions used in this example and introduce some terms which describe this particular formulation of the FFT.

Since each stage broke the DFT into two smaller DFTs, this FFT belongs to a class of FFTs called *radix-2* FFTs. Because the input (or time) samples were recursively decimated into even and odd sequences, this FFT is also called a *decimation in time (DIT)* FFT. It is also a *power-of-2* FFT for obvious reasons and a *constant-radix* FFT because the decimation performed at each stage was of the same radix.

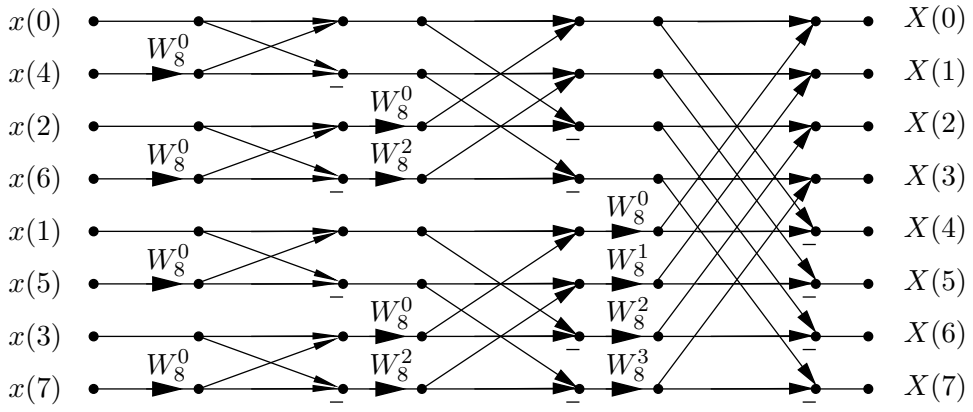


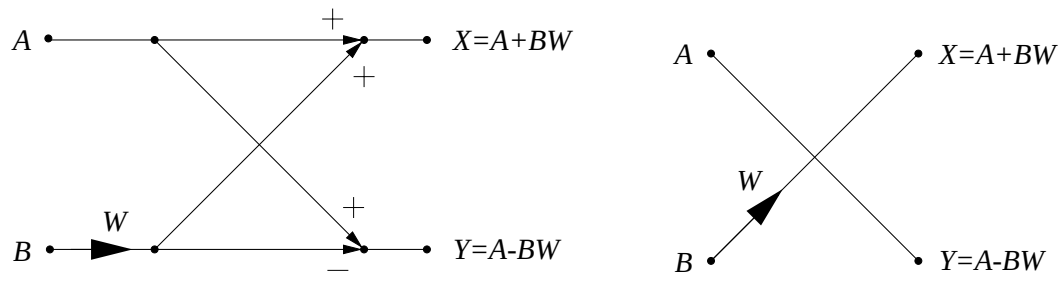
Figure 2.5: Flow graph of an 8-point radix-2 DIT FFT using only W_8 coefficients

Transform length (N)	DFT operations	FFT operations	DFT ops ÷ FFT ops
16	256	64	4
128	16,400	896	18
1024	1.05×10^6	10,240	102
32,768	1.07×10^9	4.92×10^5	2185
1,048,576	1.10×10^{12}	2.10×10^7	52,429

Table 2.1: Comparison of DFT and FFT efficiencies

2.3.3 Relative Efficiencies

As was mentioned in Sec. 2.2, the calculation of the direct form of the DFT requires $\mathcal{O}\left(N^2\right)$ operations. From the previous section, however, the FFT was shown to require only $\mathcal{O}(N \log N)$ operations. For small values of N, this difference is not very significant. But as Table 2.1 shows, for large N, the FFT is orders of magnitude more efficient than the direct calculation of the DFT. Though the table shows the number of operations for a radix-2 FFT, the gains in efficiency are similar for all FFT algorithms, and are $\mathcal{O}(N/\log N)$.



(a) Signal flow representation (b) Simplified representation

Figure 2.6: Radix-2 DIT FFT butterfly diagrams

2.3.4 Notation

A *butterfly* is a convenient computational building block with which FFTs are calculated. Using butterflies to draw flow graphs simplifies the diagrams and makes them much easier to read. Figure 2.6(a) shows a standard signal flow representation of a radix-2 DIT butterfly. Large arrows signify multiplication of signals and smaller arrows show the direction of signal flow. Figure 2.6(b) shows an alternate butterfly representation that normally is used throughout this dissertation because of its simplicity.

Figure 2.7 shows the same FFT as Fig. 2.5 except that it uses the simplified butterfly. For larger FFTs or when only the dataflow is of importance, the W twiddle factors will be omitted.

The butterflies shown in Fig. 2.6 are radix-2 DIT butterflies. Section 2.4 presents other butterfly types with varying structures and different numbers of inputs and outputs.

2.4 Common FFT Algorithms

This section reviews a few of the more common FFT algorithms. All approaches begin by choosing a value for N that is highly composite, meaning a value with many factors. If a certain N' is chosen which has l factors, then the N'-element input sequence can be represented as an l-dimensional array. Under certain circumstances, the N'-point FFT can be calculated by performing DFTs in each of the l dimensions with the (possible)

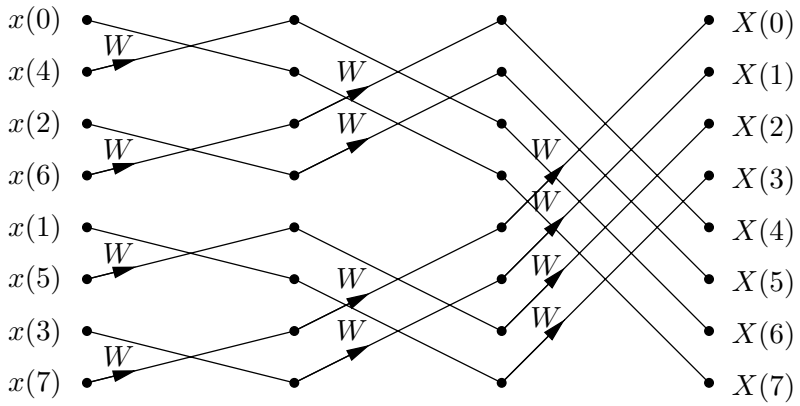
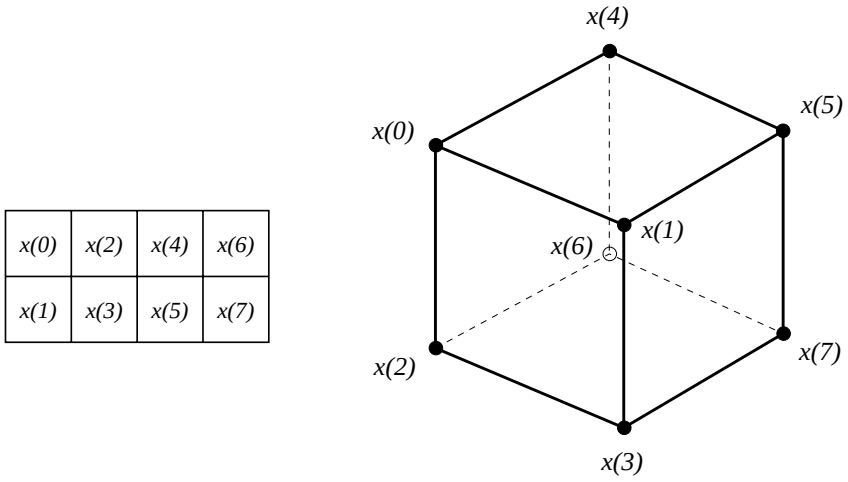


Figure 2.7: Flow graph of an 8-point radix-2 DIT FFT using simpler butterflies

multiplication of intermediate terms by appropriate twiddle factors in between sets of DFTs.

Because l-dimensional arrays are difficult to visualize for $l > 3$, this procedure is normally illustrated by reordering the 1-dimensional input sequence into a 2-dimensional array, and then recursively dividing the sub-sequences until the original sequence has been divided by all l factors of N'. The sample derivation of Sec. 2.3.2 used this recursive division approach. There, $N' = 8 = 2 \times 2 \times 2$, hence $l = 3$. Figure 2.8(a) shows how the separation of the eight members of $x(n)$ into even and odd sequences placed the members of $x(n)$ into a 2-dimensional array. For clarity, twiddle factors are not shown. The next step was the reordering of the rows of Fig. 2.8(a). The reordered rows can be shown by drawing a new 2×2 table for each of the two rows; or, the rows can be shown in a single diagram by adding a third dimension.

Figure 2.8(b) shows the elements of $x(n)$ placed into a 3-dimensional cube. It is now possible to see how the successive even/odd decimations scrambled the inputs of the flow graph of Fig. 2.5, since butterfly input pairs in a particular stage of computation occupy adjacent corners on the cube in a particular dimension. The four butterflies in the leftmost column of Fig. 2.5 have the following pairs of inputs: $\{x\,(0)\,, x\,(4)\}\,, \{x(2), x(6)\}, \{x(1), x(5)\}$, and $\{x(3), x(7)\}$. These input pairs are recognized as being adjacent on the cube of Fig. 2.8(b) in the "front-left" to "back-right" direction. In the second column of butterflies, without special notation, the terms of $x(n)$ can no longer be referred to directly, since the inputs



(a) 2-dimensional organization (b) 3-dimensional organization

Figure 2.8: $x(n)$ input mappings for an 8-point DIT FFT

and outputs of the butterflies are now values computed from various $x(n)$ inputs. However, the butterfly inputs can be referenced by the $x(n)$ values which occupy the same rows. Pairs of inputs to butterflies in the second column share rows with the following $x(n)$ pairs: $\{x(0), x(2)\}, \{x(4), x(6)\}, \{x(1), x(3)\}$, and $\{x(5), x(7)\}$, which are adjacent on the cube of Fig. 2.8(b), in the vertical direction. Thirdly, the $x(n)$ members corresponding to the inputs to the butterflies in the third column of Fig. 2.5, are: $\{x(0), x(1)\}, \{x(4), x(5)\}, \{x(2), x(3)\}$, and $\{x(6), x(7)\}$, which are adjacent on the cube of Fig. 2.8(b) in the "back-left" to "front-right" direction.

There are many possible ways to map the 1-dimensional input sequence x into a 2-dimensional array (Burrus, 1977; Van Loan, 1992). The sequence x has N elements and is written,

$$x(n) = [x(0), x(1), \ldots, x(N-1)]\,. \tag{2.30}$$

With N being composite, N can be factored into two factors N_1 and N_2,

$$N = N_1 N_2. \tag{2.31}$$

The inputs can be reorganized, using a one-to-one mapping, into an $N_1 \times N_2$ array which we call $\hat{x}$. Letting n_1 be the index in the dimension of length N_1 and n_2 the index in the dimension of length N_2, $\hat{x}$ can be written,

$$\hat{x}(n_1, n_2) = \begin{bmatrix} \hat{x}(0,0) & \hat{x}(0,1) & \dots & \hat{x}(0, N_2-1) \\ \hat{x}(1,0) & \hat{x}(1,1) & & \\ \vdots & & \ddots & \\ \hat{x}(N_1-1, 0) & & & \hat{x}(N_1-1, N_2-1) \end{bmatrix} \tag{2.32}$$

with $n_1 = 0, 1, \dots, N_1 - 1$, $n_2 = 0, 1, \dots, N_2 - 1$, and

$$x(n) = \hat{x}(n_1, n_2). \tag{2.33}$$

One of the more common systems of mapping between the 1-dimensional x and the 2-dimensional $\hat{x}$ is (Burrus, 1977),

$$n = An_1 + Bn_2 \bmod N \tag{2.34}$$

for the mapping of the inputs $x(n)$ into the array $\hat{x}(n_1, n_2)$, and

$$k = Ck_1 + Dk_2 \bmod N \tag{2.35}$$

for the mapping of the DFT outputs $X(k)$ into the array $\hat{X}(k_1, k_2)$, where $\hat{X}$ is the 2-dimensional map of $X(k)$. Substituting Eqs. 2.33–2.35 into Eq. 2.13 yields,

$$\hat{X}(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \hat{x}(n_1, n_2) W_N^{(An_1+Bn_2)(Ck_1+Dk_2)} \tag{2.36}$$

$$= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \hat{x}(n_1, n_2) W_N^{An_1Ck_1} W_N^{An_1Dk_2} W_N^{Bn_2Ck_1} W_N^{Bn_2Dk_2}. \tag{2.37}$$

The modulo terms of Eqs. 2.34 and 2.35 are dropped in Eqs. 2.36 and 2.37 since, from Eq. 2.10, the exponent of W_N is periodic in N.

Different choices for the $\hat{X}$ and $\hat{x}$ mappings and N produce different FFT algorithms. The remainder of this chapter introduces the most common types of FFTs and examines two of their most important features: their flow graphs and their butterfly structures.

2.4.1 Common-Factor Algorithms

The arguably most popular class of FFT algorithms are the so-called *common-factor* FFTs. They are also called *Cooley-Tukey* FFTs because they use mappings first popularized by Cooley and Tukey's 1965 paper. Their name comes from the fact that N_1 and N_2 of Eq. 2.31 have a common factor, meaning there exists an integer other than unity that evenly divides N_1 and N_2. By contrast, this does not hold true for *prime-factor* FFTs, which are discussed in Sec. 2.4.2.

For common-factor FFTs, A, B, C, and D of Eqs. 2.34 and 2.35 are set to $A = N_2, B = 1, C = 1$, and $D = N_1$. The equations can then be written as,

$$n = N_2 n_1 + n_2 \tag{2.38}$$

$$k = k_1 + N_1 k_2. \tag{2.39}$$

Eq. 2.37 then becomes,

$$\hat{X}(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \hat{x}(n_1, n_2) W_N^{N_2 n_1 k_1} W_N^{N_2 n_1 N_1 k_2} W_N^{n_2 k_1} W_N^{n_2 N_1 k_2}. \tag{2.40}$$

The term $W_N^{N_2 n_1 N_1 k_2} = W_N^{N n_1 k_2} = 1$ for any values of n_1 and k_2. From reasoning similar to that used in Eq. 2.19, $W_N^{N_2 n_1 k_1} = W_{N_1}^{n_1 k_1}$ and $W_N^{n_2 N_1 k_2} = W_{N_2}^{n_2 k_2}$. With these simplifications, Eq. 2.40 becomes,

$$\hat{X}(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \hat{x}(n_1, n_2) W_{N_1}^{n_1 k_1} W_N^{n_2 k_1} W_{N_2}^{n_2 k_2} \tag{2.41}$$

$$= \sum_{n_1=0}^{N_1-1} \left(\left[\sum_{n_2=0}^{N_2-1} \hat{x}(n_1, n_2) W_{N_2}^{n_2 k_2} \right] W_N^{n_2 k_1} \right) W_{N_1}^{n_1 k_1}. \tag{2.42}$$

Reformulation of the input sequence into a 2-dimensional array allows the DFT to be computed in a new way:

1. Calculate N_1 N_2-point DFTs of the terms in the rows of Eq. 2.32
2. Multiply the $N_1 \times N_2 = N$ intermediate values by appropriate $W_N^{n_2 k_1}$ twiddle factors
3. Calculate the N_2 N_1-point DFTs of the intermediate terms in the columns of Eq. 2.32.

The three components of this decomposition are indicated in Eq. 2.43.

$$\hat{X}(k_1,k_2) = \underbrace{\sum_{n_1=0}^{N_1-1}\left(\left[\underbrace{\sum_{n_2=0}^{N_2-1}\hat{x}(n_1,n_2)W_{N_2}^{n_2k_2}}_{N_2-pointDFT}\right]\underbrace{W_N^{n_2k_1}}_{TwiddleFactor}\right)W_{N_1}^{n_1k_1}}_{N_1-pointDFT} \tag{2.43}$$

Radix-r vs. Mixed-radix

Common-factor FFTs in which $N = r^k$, where k is a positive integer, and where the butterflies used in each stage are the same, are called *radix-r* algorithms. A radix-r FFT uses radix-r butterflies and has $\log_r N$ stages. Thinking in terms of re-mapping the sequence into 2-dimensional arrays, the N-point sequence is first mapped into a $r \times (N/r)$ array, and then followed by $k-2$ subsequent decimations. Alternatively, a multi-dimensional mapping places the N input terms into a $\log_r N$-dimensional $(r \times r \times \cdots \times r)$ array.

On the other hand, FFTs in which the radices of component butterflies are not all equal, are called *mixed-radix* FFTs. Generally, radix-r algorithms are favored over mixed-radix algorithms since the structure of butterflies in radix-r designs is identical over all stages, therefore simplifying the design. However, in some cases, such as when $N \neq r^k$, the choice of N determines that the FFT must be mixed-radix.

The next three subsections examine three types of common-factor FFTs which are likely the most widely used FFT algorithms. They are the: radix-2 decimation in time, radix-2 decimation in frequency, and radix-4 algorithms.

Radix-2 Decimation In Time (DIT)

If $N = 2^k$, $N_1 = N/2$, and $N_2 = 2$; Eqs. 2.38 and 2.39 then become,

$$n = 2n_1 + n_2 \tag{2.44}$$

$$k = k_1 + \frac{N}{2}k_2. \tag{2.45}$$

When applied to each of the $\log_2 N$ stages, the resulting algorithm is known as a *radix-2 decimation in time* FFT. A radix-2 DIT FFT was derived in Sec. 2.3.2, so Figs. 2.6 and 2.7

show the radix-2 DIT butterfly and a sample flow graph respectively.

Because the exact computation required for the butterfly is especially important for hardware implementations, we review it here in detail. As shown in Fig. 2.6, the inputs to the radix-2 DIT butterfly are A and B, and the outputs are X and Y. W is a complex constant that can be considered to be pre-computed. We have,

$$X = A + BW \tag{2.46}$$

$$Y = A - BW. \tag{2.47}$$

Because it would almost certainly never make sense to compute the $B \times W$ term twice, we introduce the variable $Z = BW$ and re-write the equations,

$$Z = BW \tag{2.48}$$

$$X = A + Z \tag{2.49}$$

$$Y = A - Z. \tag{2.50}$$

Thus, the radix-2 DIT butterfly requires one complex multiplication and two complex additions [3].

Radix-2 Decimation In Frequency (DIF)

The second popular radix-2 FFT algorithm is known as the *Decimation In Frequency (DIF)* FFT. This name comes from the fact that in one common derivation, the "frequency" values, $X(k)$, are decimated during each stage. The derivation by the decimation of $X(k)$ is the dual of the method used in Sec. 2.3.2. Another way to establish its form is to set $N_1 = 2$ and $N_2 = N/2$ in Eqs. 2.38 and 2.39.

$$n = N/2n_1 + n_2 \tag{2.51}$$

$$k = k_1 + 2k_2 \tag{2.52}$$

We will not examine the details of the derivation here—a good treatment can be found

[3] For applications where an FFT is executed by a processor, a subtraction is considered equivalent to an addition. In fact, the hardware required for a subtracter is nearly identical to that required for an adder.

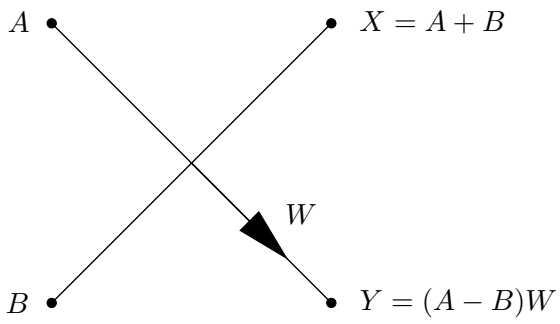


Figure 2.9: A radix-2 Decimation In Frequency (DIF) butterfly

in Oppenheim and Schafer (1989). Figure 2.9 shows a radix-2 DIF butterfly. A and B are inputs, X and Y are outputs, and W is a complex constant.

$$X = A + B \tag{2.53}$$

$$Y = (A - B)W \tag{2.54}$$

The radix-2 DIF butterfly also requires one complex multiplication and two complex additions, like the DIT butterfly.

Radix-4

When $N = 4^k$, we can employ a *radix-4* common-factor FFT algorithm by recursively reorganizing sequences into $N' \times N'/4$ arrays. The development of a radix-4 algorithm is similar to the development of a radix-2 FFT, and both DIT and DIF versions are possible. Rabiner and Gold (1975) provide more details on radix-4 algorithms.

Figure 2.10 shows a radix-4 decimation in time butterfly. As with the development of the radix-2 butterfly, the radix-4 butterfly is formed by merging a 4-point DFT with the associated twiddle factors that are normally between DFT stages. The four inputs A, B, C, and D are on the left side of the butterfly diagram and the latter three are multiplied by the complex coefficients W_b, W_c, and W_d respectively. These coefficients are all of the same form as the W_N of Eq. 2.14, but are shown with different subscripts here to differentiate the three since there are more than one in a single butterfly.

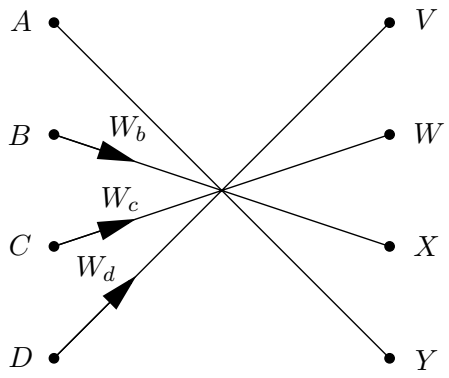


Figure 2.10: A radix-4 DIT butterfly

The four outputs V, W, X, and Y are calculated from,

$$V = A + BW_b + CW_c + DW_d \tag{2.55}$$
$$W = A - iBW_b - CW_c + iDW_d \tag{2.56}$$
$$X = A - BW_b + CW_c - DW_d \tag{2.57}$$
$$Y = A + iBW_b - CW_c - iDW_d. \tag{2.58}$$

The equations can be written more compactly by defining three new variables,

$$B' = BW_b \tag{2.59}$$
$$C' = CW_c \tag{2.60}$$
$$D' = DW_d, \tag{2.61}$$

leading to,

$$V = A + B' + C' + D' \tag{2.62}$$
$$W = A - iB' - C' + iD' \tag{2.63}$$
$$X = A - B' + C' - D' \tag{2.64}$$
$$Y = A + iB' - C' - iD'. \tag{2.65}$$

It is important to note that, in general, the radix-4 butterfly requires only three complex

FFT Radix	Number of Complex Multiplications Required
2	$0.5000\ MN - (N-1)$
4	$0.3750\ MN - (N-1)$
8	$0.3333\ MN - (N-1)$
16	$0.3281\ MN - (N-1)$

Table 2.2: Number of multiplications required for various radix algorithms

multiplies (Eqs. 2.59, 2.60, and 2.61). Multiplication by i is accomplished by a swapping of the real and imaginary components, and possibly a negation.

Radix-4 algorithms have a computational advantage over radix-2 algorithms because one radix-4 butterfly does the work of four radix-2 butterflies, and the radix-4 butterfly requires only three complex multiplies compared to four multiplies for four radix-2 butterflies. In terms of additions, the straightforward radix-4 butterfly requires 3 adds $\times$ 4 terms = 12 additions compared to 4 butterflies $\times$ 2 = 8 additions for the radix-2 approach. With a little cleverness and some added complexity, however, a radix-4 butterfly can also be calculated with 8 additions by re-using intermediate values such as $A + CW_c$, $A - CW_c$, $BW_b + DW_d$, and $iBW_b - iDW_d$. The end result is that a radix-4 algorithm will require roughly the same number of additions and about 75% as many multiplications as a radix-2 algorithm. On the negative side, radix-4 butterflies are significantly more complicated to implement than are radix-2 butterflies.

Higher radices

While radix-2 and radix-4 FFTs are certainly the most widely known common-factor algorithms, it is also possible to design FFTs with even higher radix butterflies. The reason they are not often used is because the control and dataflow of their butterflies are more complicated and the additional efficiency gained diminishes rapidly for radices greater than four. Although the number of multiplications required for an FFT algorithm by no means gives a complete picture of its complexity, it does give a reasonable first approximation. With $M = \log_2 N$, Table 2.2 shows how the number of complex multiplications decreases with higher radix algorithms (Singleton, 1969). It is interesting to note that the number of

N	FFT Radix	Real Multiplies Required	Real Additions Required
256	2	4096	6144
256	4	3072	5632
256	16	2560	5696
512	2	9216	13,824
512	8	6144	12,672
4096	2	98,304	147,456
4096	4	73,728	135,168
4096	8	65,536	135,168
4096	16	61,440	136,704

Table 2.3: Arithmetic required for various radices and transform lengths

multiplications is always $\mathcal{O}(MN)$, and only the constant factor changes with the radix.

To give a more complete picture of the complexity, the number of additions must also be considered. Assuming that a complex multiplication is implemented with four real multiplications and two real additions, Table 2.3, from Burrus and Parks (1985), gives the number of real multiplications and real additions required to calculate FFTs using various radices. Although the number of multiplications decreases monotonically with increasing radix, the number of additions reaches a minimum and then *increases*.

Further reductions in computational complexity are possible by simplifying trivial multiplications by ± 1 or $\pm i$. With questionable gains, multiplications by $\pi/4$, $3\pi/4$, $5\pi/4$, and $7\pi/4$ can also be modified to require fewer actual multiplications.

2.4.2 Prime-Factor Algorithms

In this section, we consider *prime-factor* FFTs which are characterized by N_1 and N_2 being relatively prime, meaning that they have no factors in common except unity, and—this being their great advantage—have no twiddle factors. The generation of a prime-factor FFT requires a careful choice of the mapping variables A, B, C, and D used in Eqs. 2.34 and 2.35.

For convenience, we repeat Eq. 2.37 which shows the DFT of $x(n)$ calculated by the 2-dimensional DFTs of $\hat{x}(n_1, n_2)$,

$$\hat{X}(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \hat{x}(n_1, n_2) W_N^{An_1Ck_1} W_N^{An_1Dk_2} W_N^{Bn_2Ck_1} W_N^{Bn_2Dk_2}. \tag{2.66}$$

For the twiddle factors to be eliminated, the variables A, B, C, and D must be chosen such that,

$$AC \bmod N = N_2 \tag{2.67}$$

$$BD \bmod N = N_1 \tag{2.68}$$

$$AD \bmod N = 0 \tag{2.69}$$

$$BC \bmod N = 0. \tag{2.70}$$

When Eqs. 2.67 and 2.68 are satisfied, they will reduce their respective W_N terms into the kernels of their respective 1-dimensional DFTs. Equations 2.69 and 2.70 force their respective W_N terms to equal unity, which removes the twiddle factors from the calculation.

The methods for finding suitable values are not straightforward and require the use of the *Chinese Remainder Theorem* (CRT). Burrus (1977) and Blahut (1985) provide some details on the use of the CRT to set up a prime-factor FFT mapping. Following Burrus (1977), one example of a proper mapping is to use,

$$A = \alpha N_2 \tag{2.71}$$

$$B = \beta N_1 \tag{2.72}$$

$$C = \gamma N_2 \tag{2.73}$$

$$D = \delta N_1 \tag{2.74}$$

with α, β, γ, and δ being integers. Equation 2.66 then becomes,

$$\begin{aligned}
\hat{X}(k_1,k_2) &= \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} \hat{x}(n_1,n_2) W_N^{\alpha N_2 n_1 \gamma N_2 k_1} W_N^{\alpha N_2 n_1 \delta N_1 k_2} \\
&\qquad\qquad W_N^{\beta N_1 n_2 \gamma N_2 k_1} W_N^{\beta N_1 n_2 \delta N_1 k_2} \qquad (2.75)\\
&= \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} \hat{x}(n_1,n_2) W_{N_1}^{\alpha N_2 n_1 \gamma k_1} \times 1 \times 1 \times W_{N_2}^{\beta N_1 n_2 \delta k_2} \qquad (2.76)\\
&= \sum_{n_1=0}^{N_1-1} \left[\sum_{n_2=0}^{N_2-1} \hat{x}(n_1,n_2) W_{N_2}^{\beta\delta N_1 n_2 k_2} \right] W_{N_1}^{\alpha\gamma N_2 n_1 k_1}. \qquad (2.77)
\end{aligned}$$

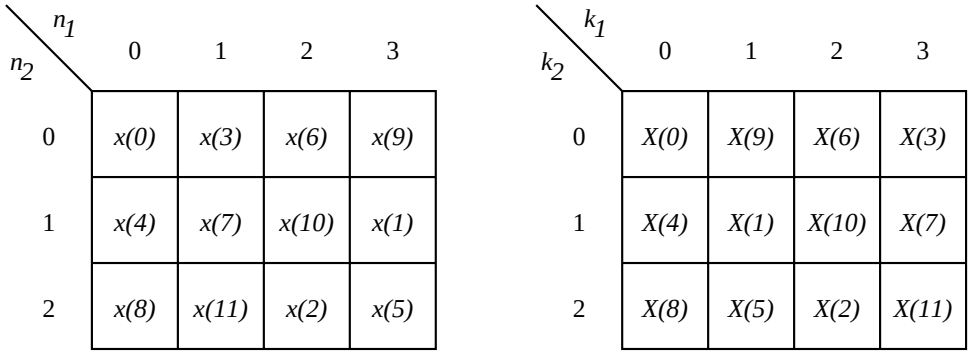
Equation 2.77 follows since $W_N^{N_2} = W_{N_1}$ and $W_N^{N_1} = W_{N_2}$, and also because $W_N^{\alpha N_2 \delta N_1 n_1 k_2} = W_N^{ADn_1k_2} = 1$ and $W_N^{\beta N_1 \gamma N_2 n_2 k_1} = W_N^{BCn_2k_1} = 1$, for any values of the integers n_1, n_2, k_1, and k_2.

By comparing Eq. 2.77 with Eq. 2.43, it should be clear that the prime-factor algorithm does not require twiddle factors. Equation 2.78 is diagrammed below showing the N_1-point and N_2-point DFTs.

$$\hat{X}(k_1,k_2) = \underbrace{\sum_{n_1=0}^{N_1-1} \left[\underbrace{\sum_{n_2=0}^{N_2-1} \hat{x}(n_1,n_2) W_{N_2}^{\beta\delta N_1 n_2 k_2}}_{N_2-point\ DFT} \right] W_{N_1}^{\alpha\gamma N_2 n_1 k_1}}_{N_1-point\ DFT} \qquad (2.78)$$

A considerable disadvantage of prime-factor FFTs that is not readily apparent from Eq. 2.78 is that the mappings for x to $\hat{x}$ and X to $\hat{X}$ are now quite complicated and involve either the use of length-N address mapping pairs, or significant computation including use of the *mod* function which is, in general, difficult to calculate. [4]

[4] A straightforward calculation of the *mod* function requires a division, a multiplication, and a subtraction. The calculation of a division requires multiple multiplications and additions.



(a) x to $\hat{x}$ mapping

(b) X to $\hat{X}$ mapping

Figure 2.11: Input and output mappings for a 12-point prime-factor FFT

Example 1 Prime-Factor FFT with $N = 12 = 3 \cdot 4$

To illustrate the steps required to develop a prime-factor FFT, consider an example with $N = 12 = 3 \cdot 4$. We note that, as required for prime-factor FFTs, the factors $N_1 = 3$ and $N_2 = 4$ have no common factors except unity. Following Oppenheim and Schafer (1989), we select $A = N_2 = 3, B = N_1 = 4, C = 9$, and $D = 4$ so that,

$$n = 3n_1 + 4n_2 \bmod N \tag{2.79}$$

$$k = 9k_1 + 4k_2 \bmod N. \tag{2.80}$$

Two mappings are required for a prime-factor FFT. One map is needed for the transform's inputs and a second map is used for the transform's outputs. The prime-factor FFT can then be calculated in the following four steps:

1. Organize the inputs of $x(n)$ into the 2-dimensional array $\hat{x}(n_1, n_2)$ according to the map shown in Fig. 2.11(a).

2. Calculate the N_1 N_2-point DFTs of the columns of $\hat{x}$.

3. Calculate the N_2 N_1-point DFTs of the rows of $\hat{x}$.

4. Unscramble the outputs $X(k)$ from the array $\hat{X}(k_1, k_2)$ using the map shown in

Fig. 2.11(b).

◁

Since the prime-factor FFT does not require multiplications by twiddle factors, it is generally considered to be the most efficient method for calculating the DFT of a sequence. This conclusion typically comes from the consideration of only the number of multiplications and additions, when comparing algorithms. For processors with slow multiplication and addition times, and a memory system with relatively fast access times (to access long programs and large look-up tables), this may be a reasonable approximation. However, for most modern programmable processors, and certainly for dedicated FFT processors, judging algorithms based only on the number of multiplications and additions is inadequate.

2.4.3 Other FFT Algorithms

This section briefly reviews three additional FFT algorithms commonly found in the literature. They have not been included in either the common-factor or the prime-factor sections since they do not fit cleanly into either category.

Winograd Fourier Transform Algorithm (WFTA) The WFTA is a type of prime-factor FFT where the building block DFTs are calculated using a very efficient convolution method. Blahut (1985) provides a thorough treatment of the development of the WFTA, which requires the use of advanced mathematical concepts. In terms of the number of required multiplications, the WFTA is remarkably efficient. It requires only $\mathcal{O}(N)$ multiplications for an N-point DFT. On the negative side, the WFTA requires more additions than previously-discussed FFTs and has one of the most complex and irregular structures of all FFT algorithms.

Split-radix For FFTs of length $N = p^k$, where p is generally a small prime number and k is a positive integer, the *split-radix* FFT provides a more efficient method than standard common-factor radix-p FFTs (Vetterli and Duhamel, 1989). Though similar to a common-factor radix-p FFT, it differs in that there are not $\log_p N$ distinct stages. The basic idea behind the split-radix FFT is to use one radix for one decimation-product of a sequence, and use other radices for other decimation-products of the sequence. For example, if $N = 2^k$, the split-radix algorithm uses a radix-2 mapping for the even-indexed members and a radix-4 mapping for the odd-indexed members of the sequence (Sorensen *et al.*, 1986). Figure 2.12

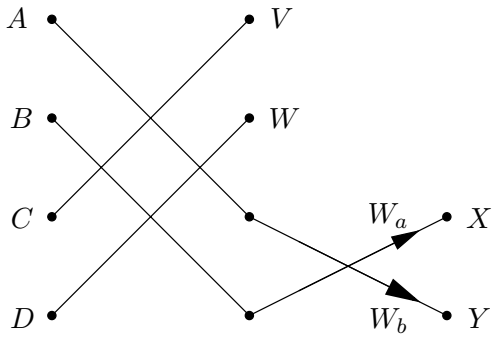


Figure 2.12: A split-radix butterfly

shows the unique structure resulting from this algorithm (Richards, 1988). For FFTs of length $N = 2^k$, the split-radix FFT is more efficient than radix-2 or radix-4 algorithms, but it has a more complex structure, as can be seen from the butterfly.

Goertzel DFT Though not normally considered a type of FFT algorithm because it does not reduce the computation required for the DFT below $\mathcal{O}(N^2)$, the *Goertzel* method of calculating the DFT is very efficient for certain applications. Its primary advantage is that it allows a subset of the DFT's N output terms to be efficiently calculated. While it is possible to use a regular FFT to more efficiently calculate a subset of the N output values; in general, the computational savings are not significant—less than a factor of two. Oppenheim and Schafer (1989) present a good overview of the Goertzel DFT algorithm.

2.5 Summary

This chapter presents an introduction to three of the most popular varieties of the Fourier transform, namely, the continuous Fourier transform, the Discrete Fourier Transform (DFT), and the Fast Fourier Transform (FFT). Since the continuous Fourier transform operates on (and produces) continuous functions, it cannot be directly used to transform measured data samples. The DFT, on the other hand, operates on a finite number of data samples and is therefore well suited to the processing of measured data. The FFT comprises a family of algorithms which efficiently calculate the DFT.

After introducing relevant notation, an overview of the common-factor and prime-factor

algorithm classes is given, in addition to other fast DFT algorithms including WFTA, split-radix, and Goertzel algorithms.

Chapter 3

Low-Power Processors

3.1 Introduction

Until the early to mid-90s, low-power electronics were, for the most part, considered useful only in a few niche applications largely comprising small personal battery-powered devices (*e.g.,* watches, calculators, etc.). The combination of two major developments made low-power design a key objective in addition to speed and silicon area. The first development was the advancement of submicron CMOS technologies which produced chips capable of much higher operating power levels, in spite of a dramatic drop in the energy dissipated per operation. The second development was the dramatic increase in market demand for, and the increased capabilities of, sophisticated portable electronics such as laptop computers and cellular phones. Since then, market pressure for low-power devices has come from both ends of the "performance spectrum." Portable electronics drive the need for lower power due to a limited energy budget set by a fixed maximum battery mass, and high-performance electronics also require lower power dissipation, but for a different reason: to keep packaging and cooling costs reasonable.

This chapter considers circuits fabricated only using Complementary Metal Oxide Semiconductor (CMOS) technologies because of its superior combination of speed, cost, availability, energy-efficiency, and density. Other available semiconductor technologies such as BiCMOS, GaAs, and SiGe generally have higher performance, but also have characteristics such as significant leakage or high minimum-V_{dd} requirements that make them far less suitable for low-power applications.

3.1.1 Power vs. Energy

Before beginning a discussion of low-power electronics, it is worthwhile to first review the two key measures related to power dissipation, namely, *energy* and *power*. Energy has units of joules and can be related to the amount of work done or electrical resources expended to perform a calculation. Power, on the other hand, is a measure of the *rate* at which energy is consumed per unit time, and is typically expressed in units of watts, or joules/sec.

In an effort to describe the dissipative efficiency of a processor, authors frequently cite power dissipation along with a description of the processor's workload. The power figure by itself only specifies the rate at which energy is consumed—without any information about the rate at which work is being done. Energy, however, can be used as a measure of exactly how efficiently a processor performs a particular calculation. The drawback of considering energy consumption alone is that it gives no information about the speed of a processor. In order to more fully compare two designs in a single measure, the product $energy \times time$ is often used, where *time* is the time required to perform a particular operation.

A more general approach considers merit functions of the form $energy^x \times time^y$ and $energy^x \times time^y \times area^z$ where x, y, and z vary depending on the relative importance of the parameters in a particular application (Baas, 1992).

3.2 Power Consumption in CMOS

Power consumed by digital CMOS circuits is often thought of as having three components. These are *short-circuit*, *leakage*, and *switching power* (Weste and Eshraghian, 1985; Chandrakasan and Brodersen, 1995). We also consider a fourth, *constant-current power*. Figure 3.1 shows a CMOS inverter with its load modeled by the capacitor C_{load}. The voltage of the input, V_{in}, is switched from high to low to high, causing three modes of current flow. Paths of the three types of current flow are indicated by the arrows. Details of the power dissipation mechanisms are given in the following three sections.

In most CMOS circuits, switching power is responsible for a majority of the power dissipated. Short-circuit power can be made small with simple circuit design guidelines. For standard static circuits, leakage power is determined primarily by V_t and the sub-threshold current slope of transistors. Constant-current power can often be eliminated from digital circuits through alternate circuit design. The total power consumed by a circuit is the sum

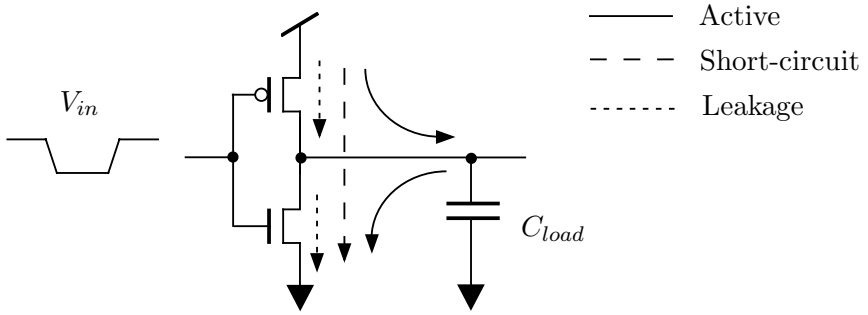


Figure 3.1: Three primary CMOS power consumption mechanisms demonstrated with an inverter

of all four components:

$$P_{Total} = P_{Short-Circuit} + P_{Leakage} + P_{Switching} + P_{Constant-Current}. \quad (3.1)$$

3.2.1 Short-Circuit Power

Many types of CMOS circuits dissipate short-circuit power while switching from one state to another. While most static circuits do not allow significant current to flow when they are in one of their two binary states, current typically flows directly from V_{dd} to Gnd during the brief period when the circuit is transitioning between states. For the case of a static inverter, as shown by the single dashed line in Fig. 3.1, current passes directly through the PMOS and NMOS transistors from V_{dd} to Gnd.

Short-circuit power is a strong-function of the rise and fall times of the input(s). A slowly switching input signal keeps the circuit in the intermediate state for a longer period and thereby causes more charge to be shorted to Gnd. The total charge lost also increases with wider-channel transistors, but the width is normally determined by the magnitude of the load being "driven," and thus is not a parameter that can be significantly or easily optimized to reduce short-circuit power.

3.2.2 Leakage Power

While most CMOS circuits are in a stable state, the gates and drains of transistors are at a potential of either V_{dd} or Gnd. Transistors with $V_{gs} = 0$ normally are considered to

have zero drain current. Although this approximation is often sufficient, it is not exact. Transistors with $V_{gs} = 0$ and $V_{ds} \neq 0$ have a non-zero "sub-threshold" drain current, I_{ds}. This current is also called the leakage current and is indicated with dotted lines in Fig. 3.1. The inverter shown has two normal states, one with $V_{in} = 0$ and $V_{out} = V_{dd}$, and a second with $V_{in} = V_{dd}$ and $V_{out} = 0$. In the first case, $V_{gs_{NMOS}} = 0$ and $V_{ds_{NMOS}} = V_{dd}$, so any I_{ds} current passing through the NMOS transistor is leakage current. Similarly, with $V_{in} = V_{dd}$, the leakage current of the PMOS transistor flows from V_{dd} to the drain of the NMOS transistor which is at Gnd.

Another source of leakage current is the reverse-biased parasitic diode caused by the p-substrate (or p-well) to n-well (or n-substrate) junction. This current is independent of the states of circuits and depends only on the process characteristics, diode junction area, temperature, and the potential difference between the p-substrate and n-well.

3.2.3 Switching Power

For most CMOS circuits, the majority of current flowing from the supply is used to charge the load to V_{dd}. In Fig. 3.1, this occurs when V_{in} is switched from high to low and charge flows through the PMOS transistor from V_{dd} to C_{load}. In the other half of the cycle, when the output transitions to 0 V, charge flows from C_{load} through the NMOS transistor to Gnd. Charging C_{load} to V_{dd} requires $Q = CV = C_{load}V_{dd}$ coulombs of charge. Hence, the supply expends $QV = C_{load}V_{dd}V_{dd} = C_{load}V_{dd}^2$ joules of energy charging C_{load} to V_{dd}. The energy stored on the capacitor is dissipated when the node is driven to Gnd. Therefore, the total energy dissipated during one cycle is $C_{load}V_{dd}^2$.

If a node is switching at f Hz, the switching power consumed by that node is,

$$P_{switching} = C_{load}\, V_{dd}^2 f \ \text{ W}. \tag{3.2}$$

Since most nodes do not switch every cycle, the *activity*, a, of a node is defined as the fraction of cycles that a node switches, and Eq. 3.2 becomes,

$$P_{switching_{node}} = a\, C_{load}\, V_{dd}^2 f \ \text{ W}. \tag{3.3}$$

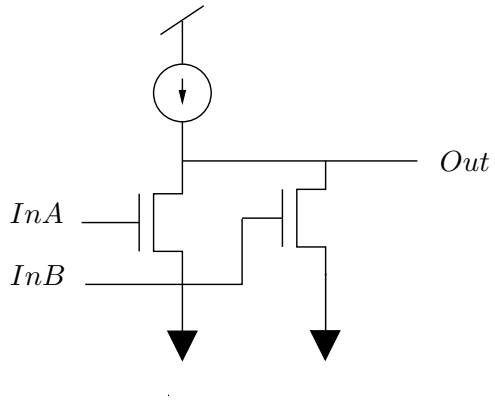


Figure 3.2: A constant-current NOR gate

The total power is the sum of the powers consumed by individual nodes. Assuming V_{dd} and f are constant over all nodes,

$$P_{switching_{total}} = \sum_{allnodes} a\, C_{load}\, V_{dd}^2\, f \ \text{W} \tag{3.4}$$

$$= V_{dd}^2\, f \sum_{allnodes} a\, C_{load} \ \text{W}. \tag{3.5}$$

3.2.4 Constant-Current Power

The fourth category of power consumption in digital circuits that can often be removed through alternate circuit design is constant-current power. Circuits that dissipate this type of power contain a current path that may conduct current even when the circuit is in a stable, non-transitory state. Frequently, the speed of the circuit is roughly proportional to the magnitude of current flow, so there is a strong incentive to design high-power circuits when using this methodology. An example of this style includes *pseudo-NMOS* circuits, which may conduct constant-current depending on the state of the circuit. Figure 3.2 shows a two-input NOR gate which dissipates power when either InA or InB is high, even while the gate is inactive. In pure NMOS technology, the current source would be an NMOS transistor with its gate tied to V_{dd}, and in pseudo-NMOS, it would be a PMOS transistor with its gate tied to Gnd. Because the current goes to zero when both inputs are low, the

Supply voltage (Volts)	Switching power (relative)	Power reduction
5.0	1.00	—
3.3	0.44	2.3×
2.5	0.25	4.0×
1.0	0.04	25×
0.4	0.006	156×

Table 3.1: Power reduction through lower supply voltages at a constant clock speed

load could also be modeled as a resistor. Another example circuit style which consumes constant current is an ECL-style circuit which uses differential pairs fed by current sources.

3.3 Common Power Reduction Techniques

This section presents a number of popular approaches which reduce the power consumption of circuits and processors. Although reducing the supply voltage is actually a specific power-reducing technique that could easily fit into several of the categories, it is given a separate subsection and placed first because it provides a straightforward way to tradeoff performance and energy-efficiency, and is often leveraged in other approaches.

3.3.1 Power Supply Reduction

A common and very effective method of reducing the energy consumption of a circuit is to reduce its supply voltage. Equation 3.5 shows the $switching\_power \propto V_{dd}^2$ relationship that makes this technique so effective. Table 3.1 illustrates the power reductions possible through voltage scaling at a constant clock frequency.

A major drawback of this approach is that, assuming no other changes are made, circuits operate more slowly as the supply voltage decreases (in fact, *much* more slowly as V_{dd} approaches V_t, the thresholds of the MOS transistors). Another major drawback is that some circuit styles can not function at low supply voltages.

3.3.2 Algorithmic and Architectural Design

Methods which reduce the power consumed by a processor through algorithmic or architectural modifications are among the most effective because of the enormous reductions in computational complexity that are sometimes possible. Several examples are given to illustrate this approach.

Operation count reduction

One of the most obvious, but difficult, approaches to reducing power at constant throughput is to reduce the number of operations necessary to complete a given workload. In an already well-designed system, however, this method provides little benefit since the number of operations should already have been minimized to maximize performance. However, if reducing the operation count is looked at as a power-saving method, dramatic savings are possible. If (i) the number of operations is reduced to a fraction r (*e.g.,* a 10% reduction gives $r = 0.9$), (ii) performance is proportional to the supply voltage, and (iii) energy consumption is proportional to V_{dd}^2; the resulting energy consumption will be proportional to r^3. This savings comes from a factor of r energy reduction due to fewer operations, and an energy reduction of r^2 due to a supply voltage reduction by r.

Disabling inactive circuits

For processors in which all functional units are not active continuously, power can be saved by switching off inactive circuits. Many circuits can be switched off by disabling the clock signal through the ANDing of the clock with an enable signal, as Fig. 3.3 shows. The primary drawback of this method, commonly called *clock gating*, is that the "clock-qualifying" circuits introduce clock skew into the timing budget that must be carefully managed. Also, the required control signals add additional, albeit manageable, complexity. Clearly, the technique works best in situations where large power-consuming functional units are unused a significant fraction of the time. Tsern (1996) reports a 2× reduction in power through clock gating in a video decoder chip.

Parallelizing and pipelining

Circuits normally can be made more energy-efficient if reduced performance is acceptable. If the workload can be parallelized onto multiple processors, and if the extra cost is acceptable,

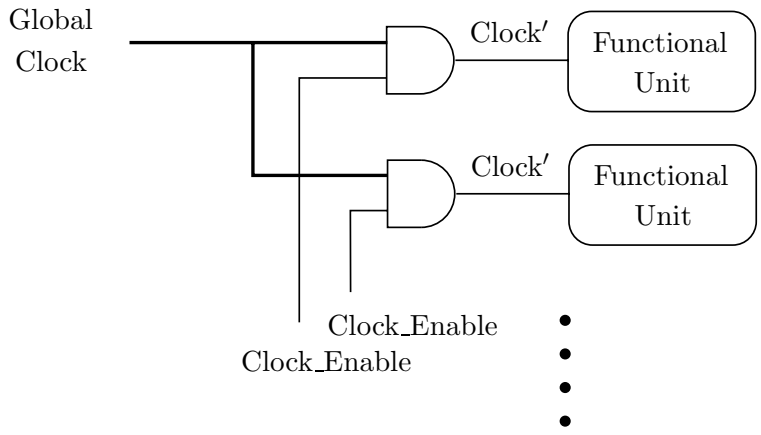


Figure 3.3: Functional units using gated clocks

there is an opportunity to reduce the total power at the same throughput by operating parallel processors. Parallelizing and pipelining are actually methods to increase *performance*, but are described here since they are often cited as techniques to reduce power. If the performance of a processor is increased by a factor p through parallelizing or pipelining, and a reduction in the supply voltage by p results in circuits that run $1/p$ as fast, then the overall energy efficiency is increased by a factor p with constant performance.

Pipelining can be thought of as a type of parallelism where instead of making duplicate copies of hardware units, the units are partitioned "in time," permitting multiple calculations to take place in a hardware unit at one time. Pipelining is accomplished by segmenting functional units into stages and inserting latches between the stages. Neglecting additional latency caused by the added latches, throughput can be multiplied by the number of smaller blocks a hardware unit is partitioned into. For example, if a block is divided into two pipeline stages, a new latch is inserted between the two halves, and the clock speed is doubled; two calculations are then performed during the time of an original clock cycle. The primary drawbacks with pipelining are: (i) difficulty managing higher-speed clocks, (ii) difficulty partitioning some circuit structures after a certain pipeline depth has been reached, and (iii) a near inability to partition some structures at all (*e.g.,* memories).

Chandrakasan *et al.* (1992) were early proponents of using pipelining to lower power dissipation, applying it to the design of a chipset for a portable multimedia terminal (Chandrakasan *et al.*, 1994).

Reduction of data word precision

For many applications, the required data precision is less than the 16 or 32 bits typically found in present day programmable processors. Therefore, a designer can choose the word width in a special-purpose processor such that no unnecessary precision is calculated. For most types of functional units, the power savings from reducing the number of bits in data words is linear in $word\_width$. For other structures, such as multipliers, the energy savings are proportional to $(word\_width)^2$, and a substantial reduction of power is possible.

As an example of this technique, a video subband decoder chip by Gordon and Meng (1995) performs YUV to RGB conversion using filter coefficients with only *two bits* of precision, without a noticeable degradation in visual quality.

Data locality

A processor spends some amount of its energy moving data and/or instructions between where they are stored and where they are used. If needed information can be stored closer to where it is used, then there is an opportunity to reduce communication energy. This approach works effectively only if the data exhibit some amount of *temporal locality*, meaning a subset of the global data is frequently referenced over a relatively short time interval.

General-purpose processors have long made use of locality for performance reasons and on many different levels (Hennessy and Patterson, 1996). Register files are typically fast, multiple-ported memories used to store frequently-used data. Data caches and instruction caches are larger memories which hold data and instructions respectively, and are frequently found in multiple "levels" (*e.g.,* level-1 cache, level-2 cache, etc.). Main memory functions as a buffer for information stored in mass storage such as disk or tape. Networks add more layers to the hierarchy with local-area networks being faster but less far-reaching than wide-area networks.

While exploiting locality often has a significant effect on performance, it can also decrease power dissipation because it reduces the energy required for communication.

3.3.3 Circuit Design

Techniques to achieve low-power dissipation through circuit design are difficult to summarize because methods do not generalize well across different circuit types. For example, it is likely that the best approaches to reducing power for a memory and a bus driver are

different. Nevertheless, this section reviews three common circuit design approaches that reduce energy dissipation across a wide variety of circuit types.

Transistor sizing

As illustrated in Fig. 3.1 on page 34, the load CMOS circuits drive is normally capacitive. This capacitance has three primary components: (i) gate capacitance, (ii) diffusion capacitance, and (iii) interconnection capacitance,

$$C_{total} = C_{gate} + C_{diffusion} + C_{interconnect}. \tag{3.6}$$

The speed of a CMOS gate can be approximated by,

$$\begin{aligned} Delay &\approx C_{total}/I && (3.7)\\ &\approx (C_{gate} + C_{diffusion} + C_{interconnect})/I && (3.8)\\ &\propto (C_{gate} + C_{diffusion} + C_{interconnect})/width && (3.9)\\ &\propto (C_{gate} + C_{diffusion})/width + C_{interconnect}/width. && (3.10) \end{aligned}$$

Typically, transistors in digital circuits have minimum-length channels, but channel *widths* are varied according to the needed drive current, since the drain current I is essentially proportional to the width. For wide transistors, both the gate and diffusion capacitances are roughly proportional to the widths of the transistor. For a circuit dominated by gate and diffusion capacitance, reducing transistor widths reduces energy, and from Eq. 3.10, does not have a large impact on delay. However, for circuits whose load is dominated by interconnection capacitance, narrowing of the transistors again causes energy to be reduced, but unfortunately, also causes the delay to increase proportionately.

Reduced voltage swing

Although Eq. 3.5 states that switching power is proportional to V_{dd}^2, it is actually proportional to $V_{dd}V_{swing}$, where V_{swing} is the difference between logic "0" and logic "1" voltages. For many CMOS circuits, the voltage swing is V_{dd}, and the approximation is accurate. If the nodes of a circuit swing through small voltage variations, the energy dissipation is reduced significantly (Matsui *et al.*, 1994; Matsui and Burr, 1995). The primary difficulty with this approach is that circuits to detect small voltage differences are typically complex and often

require special timing signals. Therefore, reduced-swing circuits are best used when driving a large capacitive load and the extra effort is justified—such as with buses or high-fan-in topologies. As an example, Yamauchi *et al.* (1995) present a differential bus structure which operates with low voltage swings. To further save power, the scheme re-uses charge in the bus to precharge other wires, rather than dumping charge to *Gnd* every cycle.

Elimination of constant current sources

As mentioned in Sec. 3.2.4, some circuits consume power even when they are in a stable state. CMOS circuits usually can be redesigned to eliminate this inefficient characteristic. However, since constant-current circuits can be very fast, redesigning the circuit may reduce its speed.

3.3.4 Fabrication Technology

The optimization of various aspects of CMOS fabrication technologies can result in dramatic improvements in energy-efficiency. Although this is probably the most difficult and expensive way to reduce power, it also gives one of the most significant gains.

Technology scaling

The advancement of fabrication technologies is nearly synonymous with the reduction of minimum feature sizes. Although scaling entails a nearly complete overhaul of a CMOS process, it gives substantial improvements to the most critical of integrated circuit parameters, namely, area, speed, and power. If the scale of a fabrication technology is reduced by a factor α (*e.g.,* 0.5 μm scaled to 0.25 μm implies $\alpha = 2$), then, to first order, using constant-field scaling where the supply voltage is reduced by $1/\alpha$; the area is reduced by $1/\alpha^2$, delay decreases by $1/\alpha$ and the energy consumption is decreased by $1/\alpha^3$ (Weste and Eshraghian, 1985).

Reduction of interconnection capacitance

As feature sizes shrink, the percentage of total load capacitance attributable to wiring capacitance grows. The flat-plate capacitance between an interconnect layer and an adjacent surface can be written as,

$$C = \epsilon_r \epsilon_0 \frac{area}{t}, \tag{3.11}$$

where ϵ_r is the relative permittivity of the dielectric, ϵ_0 is the permittivity of free space, $area$ is the bottom-plate area of the interconnect, and t is the thickness of the dielectric material. Since the capacitance is proportional to the permittivity, $\epsilon = \epsilon_r \cdot \epsilon_0$, of the dielectric between the interconnect and the substrate, a low ϵ material will give a lower load capacitance and result in lower power dissipation and higher circuit speed. More accurate models which take into account fringing capacitance and interconnection aspect ratios are given by Barke (1988).

SiO_2 is by far the most commonly used dielectric material in CMOS chips, largely because it is easy to grow or deposit in standard processes. For thermally-grown oxide, $\epsilon_r \approx 3.9$; values for chemical-vapor-deposited (CVD) oxide are in the range 4.2–5.0. The material SiOF holds some promise as a low-dielectric material; values of ϵ_r are in the range of 3.0–3.6. Ida *et al.* (1994) report a process for the deposition of SiOF in deep sub-micron CMOS processes that achieves a 16% reduction in wiring capacitance.

Ultra Low Power (ULP) CMOS

ULP CMOS is an approach to producing very low-power CMOS circuits by reducing the supply voltage to several hundred millivolts (Burr and Peterson, 1991b; Burr and Shott, 1994). To maintain good performance at low supply voltages, the threshold voltages of MOS transistors must also be reduced. Unfortunately, this requires a change in the CMOS fabrication process. Because variations in V_t, when operating in a low-V_{dd} and low-V_t environment, cause significant variations in performance, the ULP approach relies on the biasing of transistor bodies to adjust thresholds. Tuning through body bias can cancel threshold changes due to temperature shifts and inter-die process variations. Additionally, thresholds can be adjusted to optimize power dissipation and/or performance according to circuit activity. Figure 3.4 shows a simplified cross-section of an NMOS and a PMOS transistor with separate well biases.

A side effect of reducing V_t is a large increase in leakage current when transistors are cutoff, and a much lower $(I_{ds}@\,|V_{gs}| = V_{dd})/(I_{ds}@V_{gs} = 0)$, or I_{on}/I_{off} ratio. A key tradeoff made by ULP technology is the acceptance of increased leakage power in return for greatly reduced active power. For many applications, this results in a net power reduction. The presence of large leakage currents requires the modification of some circuits to maintain correct operation, however.

The primary drawbacks to ULP CMOS are that it requires (i) a change in the fabrication

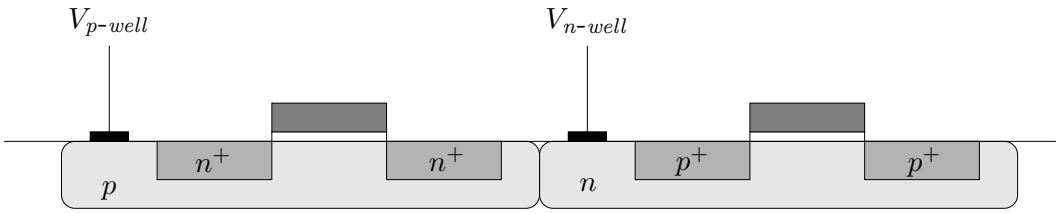


Figure 3.4: Cross-section of ULP NMOS and PMOS transistors

process, (ii) additional circuitry to adjust body potentials, and (iii) additional routing of separate $V_{p\text{-}well}$ and $V_{n\text{-}well}$ nodes.

Silicon On Insulator technology (SOI)

SOI technology is different from standard bulk processes in that for SOI, the active silicon sits on an insulating layer above the die's substrate, which insulates the source, drain, and channel regions from the substrate. One of the largest benefits of the technology is that diffusion capacitances are greatly reduced, as compared to devices fabricated in a standard bulk process where diffusion regions are uninsulated from the substrate. Since diffusion capacitance contributes a sizeable fraction to the total load capacitance, C_{load}, both C_{load} and the switching power are significantly reduced. SOI devices have roughly 25% less switched capacitance which results in a significant reduction in power. In addition, the smaller C_{load} makes circuits faster which can be traded-off for even lower power dissipation through reduced supply voltages. Using an SOI technology running at $V_{dd} = 1.5\,\mathrm{V}$ as a reference, a similar bulk process would have to run at $V_{dd} = 2.15\,\mathrm{V}$ to achieve the same performance—which would increase its overall power consumption to 2.7× greater than the SOI's (Antoniadis, 1997). Though not stated by Antoniadis, it appears that the improvement would be about 1.9× if the comparison were made relative to bulk at $V_{dd} = 1.5\,\mathrm{V}$.

The SOI device's structure allows new device styles to be implemented more easily. One possibility is a device whose gate is connected to the transistor's body. Called *DT-CMOS* or *VTMOS* (Assaderaghi *et al.*, 1994), the transistors' V_ts are lowered while the device is conducting and raised while the device is off. One problem with the approach is that the supply voltage is limited to $V_{dd} < 0.7\,\mathrm{V}$ since the source to gate-body is now a forward-biased diode. Antoniadis also points out a limitation on transistor width due to the high

body resistance, since body contacts must be made at the short end(s) of a wide transistor.

With significant added complexity, a second gate beneath the channel, or *back gate*, can be added to the device. This back gate can then be connected to the front gate to increase current drive, but it appears this approach does not give a larger current drive per input capacitance ratio, compared to standard SOI devices (Antoniadis, 1997).

A third possible enhancement to standard SOI is to use a back gate structure, but adjust the potential according to circuit activity, or expected activity, rather than the state of each individual device. Known as *SOIAS* (Vieri *et al.*, 1995; Yang *et al.*, 1997), this approach appears promising, though it has the added complexity of back-gate control.

The primary drawback of SOI is that it requires special wafers which have the insulating layer built in, as well as considerable changes to the fabrication process. The extra cost of the wafers has been a barrier to the use of SOI, but as the prices of wafers continue to drop, this technology will no doubt gain in popularity.

3.3.5 Reversible Circuits

Reversible circuits can be broadly defined as those circuits in which a computation can be performed producing a result, and then every step taken in the forward computation is "un-done" in reverse order. The reason for undoing the calculation is that under certain conditions, some of the energy consumed in the forward computation can be recovered during the reversed half of the cycle. Except for the simplest circuits, the technique requires the saving of intermediate values that were generated during the forward computation. These circuits are also called *adiabatic* because in the limit as the operating frequency goes to zero, they theoretically can operate with heat production (or energy dissipation) that approaches zero. They are also called *charge-recycling* circuits for obvious reasons. Because one approach uses multi-phase signals resembling clocks, some reversible circuits are also called *clock-powered* circuits.

To explain briefly how reversible circuits work, we first compare them to a common static CMOS inverter. In the inverter of Fig. 3.1 on page 34, the PMOS transistor acts as a switch and, when activated, connects the output to V_{dd} to drive the output high. Similarly, when activated, the NMOS transistor connects the output to *Gnd* to drive the output to a low voltage. In the reversible circuit case, MOS transistors are also used as switches between the output and "higher-voltage" and "lower-voltage" power sources. However, that is about all that is common to both circuits. For reversible circuits, the supply is not a fixed voltage

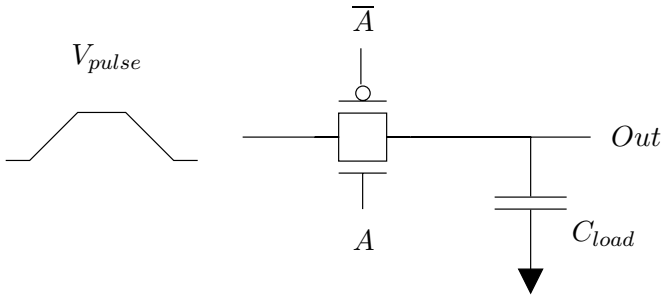


Figure 3.5: A reversible logic gate

but instead is at a voltage that ramps up and then ramps down. Rise and fall times of the supply are generally much longer than the rise and fall times of typical gate outputs. In Fig. 3.5, V_{pulse} is this supply node. When "pulling up" its output, the circuit uses a network of transistors (a transmission gate in the simple example shown) to conditionally connect its output to the ramping supply. The ramping times must be slow enough so that the voltage drop across the transistors is small. In the second half of the cycle, the supply ramps downward and the charge stored in the capacitive load is returned to the supply—which must then recover and reuse the "borrowed" energy.

The signals A and $\overline{A}$ are the inputs to the gate and must be stable throughout the rise, high, and fall periods of V_{pulse}. This requirement is necessary to keep the voltage drop between V_{pulse} and Out low. V_{pulse} is generated by a resonant LC circuit where C is the capacitance of on-chip circuits, and L is a large inductor, normally located off-chip.

Solomon (1994) summarizes three key requirements for efficient reversible circuits first described by Hall (1992):

- Gate voltages can be changed only while source-drain voltages are zero.
- Source-drain voltages can only be changed while transistors are cutoff.
- Voltages must be ramped slowly to minimize $P = VI$.

Solomon then gives the following reasons why MOSFETs are especially well suited for use in reversible circuits, compared to other types of devices: (i) current can flow in both directions through the source and drain terminals, (ii) the source to drain offset voltage is zero, and (iii) there is zero gate current.

Typically, a companion gate produces the complementary output $\overline{Out}$, which has the important added benefit of keeping the loading of V_{pulse} constant and independent of the state of the circuit. A constant $\sum_{allnodes} C_{load}$ greatly simplifies the design of the resonant LC power supply.

If R is the effective MOSFET channel resistance through which the capacitive load C_{load} is charged, and T is the switching time or length of time that current flows to and from C_{load}, then the power consumed in a reversible circuit is (Athas *et al.*, 1994),

$$P_{reversible} = \frac{RC_{load}}{T} C_{load} V_{dd}^2 f. \tag{3.12}$$

When compared with Eq. 3.2 on page 35, it is apparent that adiabatic circuits can have lower power than standard CMOS circuits if T is sufficiently long.

The major drawbacks of reversible circuits include: (i) low performance at efficient operating points, (ii) a V_{pulse} circuit that is difficult to design because it must recycle energy very efficiently, and (iii) a completely new circuit style that is not suited to all circuit types. For example, a reversible memory is only theoretically possible. Further, it appears reversible techniques are clearly advantageous only for circuits which slowly drive large-capacitance loads. A 0.5 μm 12,700-transistor processor core using mostly-reversible circuits has been reported (Athas *et al.*, 1997). The processor is a remarkable achievement but its power dissipation of 26 mW at 59 MHz is comparable to what could be expected from a standard low-V_{dd} CMOS design.

3.3.6 Asynchronous Systems

An asynchronous system is one in which there are no clocks. Instead of using clocks for synchronization, asynchronous designs use special handshaking signals between blocks to coordinate the passing of data among them. The primary advantages of these systems are:

- Circuits are only active when there are data for them to process.
- Their speed is not limited by a global clock frequency that was selected to operate under *worst-case* conditions. Instead, the system can operate at the maximum performance it is capable of, at whatever the *current* conditions are.

Asynchronous systems have not yet seen wide acceptance—largely because of their added complexity and their incompatibility with current design methods and tools.

3.3.7 Software Design

Although gains are typically modest, power can sometimes be saved through careful selection of instructions on a programmable processor. It is often possible to calculate a result in more than one way. Since different instructions can consume different amounts of power (particularly in a processor which shuts down unused functional units), it is likely that using equivalent but lower-power instructions can reduce the overall energy required to perform a job. Also, since the order in which instructions are executed affects node switching, the *ordering* of instructions can also affect energy consumption.

A good example of a use of this technique is a "power-aware" compiler which optimizes the code it produces considering power dissipation, as well as other standard factors such as execution time and code size. Tiwari *et al.* (1994) achieved a 40% reduction in energy by hand-tuning a block of code on a 486DX2 system. Their energy-efficiency accrued largely through the reduction of reads and writes to memory through better register allocation. It is important to note that the hand-tuning by Tiwari *et al.* also provided a 36% reduction in the execution time, so it appears, as they state, that a compiler using their technique does not produce code much different than a compiler tuned for maximum performance. Other possible methods to reduce energy consumption mentioned by Tiwari *et al.* include the reduction of pipeline stalls, cache misses, switching on address lines, and page misses in page-mode DRAMs. But these approaches seem to be either no different from a high-performance approach, or are likely to have little impact on total power. Another example cited is work by Su *et al.* (1994) in which they reduce energy used in control circuits by modifying instruction scheduling. Although Su claims a significant reduction in energy, Tiwari *et al.*'s measurements yielded only a 2% maximum reduction in overall energy consumption using Su's technique.

Mehta *et al.* (1997) report a 4.25% average reduction in energy consumption on a simulated DLX machine by reassigning registers to minimize toggling in the instruction register and register file decoder, and on the instruction bus.

3.4 Summary

This chapter presents details of the four primary CMOS power-consumption classes: switching power, short-circuit power, leakage power, and constant-current power.

The second half of the chapter reviews techniques commonly used to reduce power

dissipation. Because switching power is proportional to the square of the supply voltage for many circuits, reducing the supply voltage is one of the most often used and one of the most effective methods to achieve reduced power dissipation. A number of techniques used at the algorithmic, architectural, circuit, and technological levels are given.

Chapter 4

The Cached-FFT Algorithm

FFT algorithms typically are designed to minimize the number of multiplications and additions while maintaining a simple form. Few algorithms are designed to take advantage of hierarchical memory systems, which are ubiquitous in modern processors. This chapter presents a new algorithm, called the *cached-FFT,* which is designed explicitly to operate on a processor with a hierarchical memory system. By taking advantage of a small cache memory, the algorithm enables higher operating clock frequencies (for special-purpose processor applications) and reduced data communication energy.

FFT algorithms in the signal processing literature typically are described by butterfly equations, a dataflow diagram, and sometimes through a specific implementation described by pseudo-code. An algorithm which exploits a hierarchical memory system must also specify its memory access patterns, as the order of memory accesses strongly effects performance. These patterns could be given with a pseudo-code example, but this would only show one specific approach and does not contain the generality we desire. To describe the memory access patterns of the cached-FFT, we derive (i) the cache addresses and W_N exponents, for butterfly execution, and (ii) the memory and cache addresses, for cache loading and flushing. We present these addresses and exponent control signals independent of radix and in such a way that memory access patterns can be rearranged while maintaining correct operation and maximum reuse of data in the cache. Because the simple and regular structures of radix-r and in-place algorithms make their use attractive, particularly for hardware implementations, the cached-FFT we present is also radix-r and in-place.

Although there are many ways to derive the cached-FFT, it is presented most easily by beginning with a new and particularly regular FFT algorithm, which we call the *RRI-FFT.*

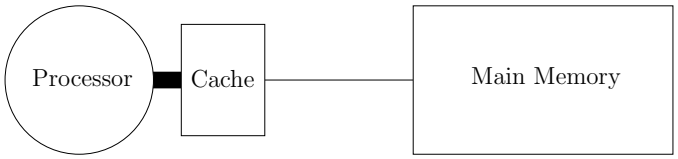


Figure 4.1: Cached-FFT processor block diagram

We define, show the existence, and derive a general description of the RRI-FFT. We then extend that description to derive a general description for the cached-FFT.

This chapter introduces new terms, characteristics, and implementation details of the cached-FFT. To aid in its presentation, we assume that the cached-FFT algorithm operates on a processor with a hierarchical memory structure.

4.1 Overview of the Cached-FFT

4.1.1 Basic Operation

The cached-FFT algorithm utilizes an architecture similar to the single-memory architecture shown in Fig. 5.1, but with a small cache memory positioned between the processor and main memory, as shown in Fig. 4.1. The C-word cache has significantly lower latency and higher throughput than the main memory since normally, $C \ll N$, where N is the length of the FFT transform, and the main memory has at least N words.

The FFT caching algorithm operates with the following procedure:

1. Input data are loaded into an N-word main memory.
2. C of the N words are loaded into the cache.
3. As many butterflies as possible are computed using the data in the cache.
4. Processed data in the cache are flushed to main memory.
5. Steps 2–4 are repeated until all N words have been processed once.
6. Steps 2–5 are repeated until the FFT has been completed.

4.1.2 Key Features

A distinguishing characteristic of the cached-FFT is that it isolates the main memory from the high-speed portion of the processor. The algorithm allows repeated accesses of data from a faster level of the memory hierarchy rather than a slower level. This characteristic offers several advantages over methods which do not exploit the use of data caches, such as,

- increased speed—since smaller memories are faster than larger ones
- increased energy-efficiency—since smaller memories require lower energy per access and can be positioned closer to the processor, than larger memories

The gains possible in performance and energy-efficiency through the use of the cached-FFT increase as the transform length increases.

The algorithm also presents several disadvantages, including:

- the addition of new functional units (caches), if they are unavailable
- added controller complexity

4.1.3 Relevant Cached-FFT Definitions

To aid in the introduction of the cached-FFT algorithm, we define several new terms:

Definition: Epoch An *epoch* is the portion of the cached-FFT algorithm where all N data words are loaded into a cache, processed, and written back to main memory *once*. □

Although the number of epochs, E, can equal one—that case is degenerate, and normally $E \geq 2$. Steps 2–5 in the list of Sec. 4.1.1 comprise an epoch.

Definition: Group A *group* is the portion of an epoch where a block of data is read from main memory into a cache, processed, and written back to main memory. □

Steps 2–4 in the list of Sec. 4.1.1 comprise a group.

Definition: Pass A *pass* is the portion of a group where each word in the cache is read, processed with a butterfly, and written back to the cache *once*. □

Definition: Balanced cached-FFT A cached-FFT is *balanced* if there are an equal number of passes in the groups from all epochs. □

Balanced cached-FFTs do not exist for all FFT lengths. The transform length of a balanced cached-FFT is constrained to values of r^{EP}, where r is the radix of the decomposition, E is the number of epochs, and P is the number of passes per group.

4.2 FFT Algorithms Similar to the Cached-FFT

Gentleman and Sande (1966)

Gentleman and Sande propose an FFT algorithm that can make use of a "hierarchical store." The method they propose selects a transform length N that has two factors: $N = AB$. The factor A is the "largest Fourier transform that can be done in the faster store." A *mixed-radix* decomposition is performed on the N-point DFT resulting in A-point transforms and B-point transforms. Their decomposition can also be viewed as a single radix-A decimation or a single radix-B decimation of the input sequence. This paper provides an example where a 256K data set is read from tape, written to disk, transformed with $A = 4096$-point FFTs in one pass, transformed with $B = 64$-point FFTs in a second pass, then written back to disk and tape.

Singleton (1967)

Singleton's approach is perhaps of limited use for modern processors because of its intended application which used "serial-organized memory files, such as magnetic tapes or serial disk files." Nearly all semiconductor memories are random access and the serial-access restriction is a significant, unnecessary limitation. The algorithm is also optimized to use a particularly small amount of fast memory, and requires a full $\log_2 N$ passes through the whole data set. However, it does read correct-order input and write correct-order output (as opposed to bit-reversed input or output).

Brenner (1969)

Brenner proposes two algorithms in his paper. One is better suited for cases where the length of the transform is not much longer than the size of the fast memory, and the second is better suited for cases where the transform length is much longer. Although he assumes

both memories are random-access, both algorithms require multiple transpositions of data in memory, which adds significantly to the execution time of a modern processor.

Rabiner and Gold (1975)

Rabiner and Gold discuss a method they call "FFT computation using fast scratch memory." They propose using a small fast memory of size $\sqrt{N}$ to reduce the number of "major memory" accesses. A single DIF dataflow diagram for $N = 16$ is given which uses an unusual not-in-place structure. Unfortunately, they give neither a derivation method nor references. Through private communication with both authors (Rabiner, 1997; Gold, 1997), however, it was learned that although neither author is certain of the origins of the algorithm, Rabiner believes it came from one of the many FFT conferences in the 1960s, and is not clearly attributable to any one individual.

Gannon (1987)

Gannon's method, which is similar to Gentleman and Sande's approach, reformulates the DFT equation, Eq. 2.5, using a mixed-radix decomposition with 2^{α}-point and $2^{n-\alpha}$-point transforms. His method can be viewed as a single radix-2^{α} or radix-$2^{n-\alpha}$ decimation of the input sequence. Gannon designed his algorithm to execute on an Alliant FX/8 multi-vector computer. The re-structured FFT allows the eight vector processors to more effectively use the available memory bandwidth through the use of a shared multi-ported cache.

Blocking through compiler optimizations

Blocking is a general optimization technique used by compilers to increase the temporal locality in a computer program (Hennessy and Patterson, 1996). A program which exhibits high temporal locality frequently references a small number of addresses, and therefore suffers few cache misses. Rather than accessing data in an array by rows or by columns, a "blocked" algorithm repeatedly accesses data in a submatrix or *block* (Lam *et al.*, 1991). Keeping data references within a block reduces the size of the working set, and thereby improves the cache's effectiveness.

Since a compiler generates nearly all code used by general-purpose processors, FFTs executing on those processors may perform better if they are blocked by an optimizing compiler. However, blocking compilers only attempt to minimize the expectation of the

cache miss rate. An FFT algorithm that has been optimized for a hierarchical memory, on the other hand, will likely have higher performance.

Although designers frequently use compilers to generate some code for programmable DSP processors, compiled code often performs poorly and critical sections of code are normally written by hand (Bier, 1997). For special-purpose FFT processors, hardware state machines or a small number of very-high-level instructions control the processor and thus, code generation is non-existent or trivial, and compilers are not used.

Bailey (1990)

Bailey proposes a "four-step FFT algorithm" for length-N transforms where $N = n_1 n_2$. Again similar to Gentleman and Sande's approach, the method uses a mixed-radix decomposition with n_1-point and n_2-point transforms. The method is equivalent to a single radix-n_1 or radix-n_2 decimation of the input sequence. The following four steps are performed on the $n_1 \times n_2$ input data array:

1. n_1 n_2-point FFTs are calculated.
2. The resulting data are multiplied by twiddle factors.
3. The matrix is transposed.
4. n_2 n_1-point FFTs are calculated.

Bailey uses the four-step algorithm to reduce the number of accesses to data in "external memory" on Cray-2, Cray X-MP, and Cray Y-MP supercomputers.

Carlson (1991)

The processors of Cray-2 supercomputers have access to 16,000-word *local* memories in addition to a 268-million-word common memory. Carlson proposes a method using local memories to hold FFT "subproblems," which increases performance by increasing the total memory bandwidth (*i.e.,* bandwidth to local and common memories). Subproblems consisting of only four or eight words realize the best performance because of features peculiar to the Cray-2 architecture.

Smit and Huisken (1995)

Smit and Huisken propose using 32 and 64-word memories in the design of a 2048-point FFT processor. Unfortunately, they give no details or references regarding the algorithm.

Stevens *et al.* (1998)

Stevens *et al.* (1998) and Hunt *et al.* (1998) propose a mixed-radix FFT decomposition using N_1-point and N_2-point transforms, where $N = N_1 N_2$. They estimate the energy dissipation of their proposed 1024-point asynchronous FFT processor to be 18 μJ per transform. In their comparison with other processors, they unfortunately cite Spiffee1's energy dissipation at a supply voltage of 3.3 V as 50 μJ, when it is actually 25 μJ. Although not noted in their papers, Spiffee1's measured energy dissipation at a supply voltage of 1.1 V is 3.1 μJ per transform.

4.3 General FFT Terms

Some of the terms discussed in this section were used loosely in Ch. 2. They are now described more completely.

Radix-r FFT algorithms. A few key characteristics of radix-r, FFTs are that they:

- Use only radix-r butterflies, which have r inputs and r outputs
- Have $\log_r N$ *stages* (see *stage* definition below)
- Contain N/r butterflies per stage

Definition: Stage A *stage* is the part of an FFT where all N memory locations are read, processed by a butterfly, and written back once. □

In an FFT dataflow diagram, a stage corresponds to a column of butterflies. Figure 4.2 is a representative dataflow diagram which shows the six stages (0–5) of a 64-point radix-2 FFT. Additionally, the 64 memory locations are indicated along the vertical axis. By convention, memory locations are numbered with 0 at the top and $N - 1$ at the bottom.

Definition: In-place A butterfly is *in-place* if its inputs and outputs use the same memory locations. An *in-place FFT* uses only in-place butterflies. □

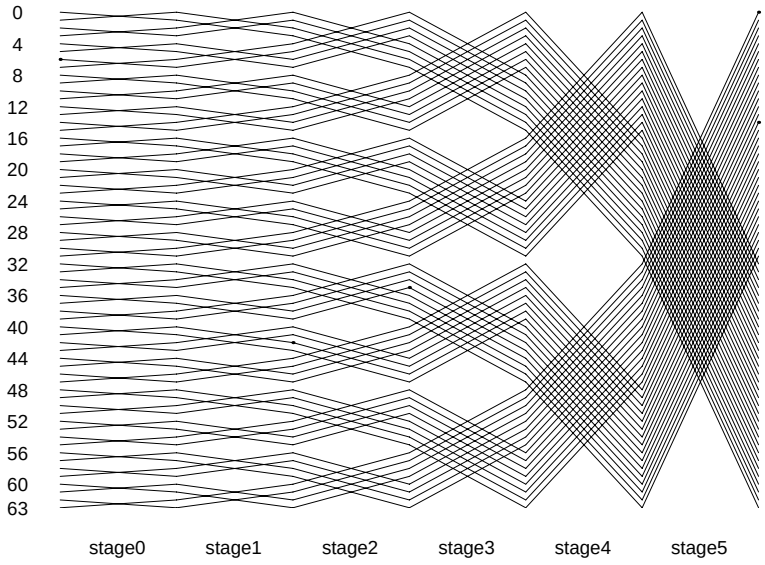


Figure 4.2: FFT dataflow diagram with labeled stages

Definition: Stride The *stride* of a butterfly is the distance (measured in memory locations) between adjacent "legs" or "spokes" of a butterfly. □

The stride is also the increment between memory addresses if memory words are accessed in increasing order. As an example, Fig. 4.3 shows a radix-2 butterfly with one leg reading memory location k and the adjacent leg (the only other one in this case) reading location $k+4$. The stride is then $(k+4)-k=4$. Similarly, Fig. 4.4 shows a radix-4 butterfly with a stride of one between its input and output legs.

Definition: Span The *span* of a butterfly is the maximum distance (measured in memory locations) between any two butterfly legs. □

Using the two previous examples, the span of the butterfly in Fig. 4.3 is four and the span of the butterfly in Fig. 4.4 is three. For a radix-r butterfly with constant stride between all adjacent legs pairs, the span can also be found by,

$$span = (r-1)stride. \tag{4.1}$$

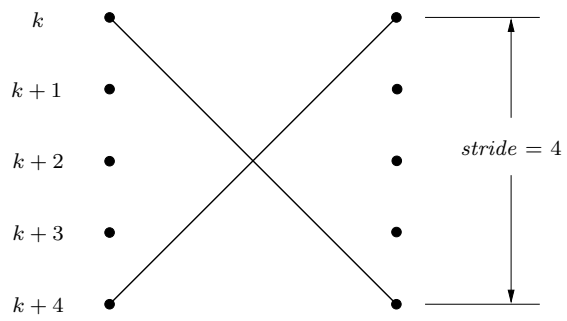


Figure 4.3: Radix-2 butterfly with $stride = 4$

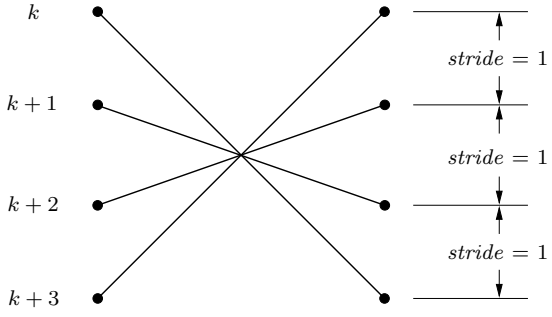


Figure 4.4: Radix-4 butterfly with $stride = 1$

4.4 The RRI-FFT Algorithm

Definition: RRI-FFT The *RRI-FFT* (Regular, Radix-r, In-place FFT) algorithm is a radix-r in-place FFT with the following additional characteristics:

- The stride of all legs of a butterfly (input and output) are equal.
- The stride of all butterflies in a stage are equal.
- The stride of butterflies monotonically increases or decreases by a factor of r as the stage number varies from 0 to $\log_r(N) - 1$.

□

4.4.1 Existence of the RRI-FFT

Theorem 1 *It is always possible to draw an RRI-FFT dataflow diagram where the stride of butterflies increases by a factor r through succeeding stages.*

Proof From the RRI-FFT definition, we assume a radix-r, in-place decomposition using butterflies with constant stride. Extending the FFT derivation given in Sec. 2.4, common-factor FFTs can be developed through the following procedure:

1. Decimate a 1-dimensional M-point data sequence by r, which results in a 2-dimensional $(M/r) \times r$ array
2. Perform DFTs of the columns (or rows)
3. Possibly multiply the intermediate data by twiddle factors
4. Perform DFTs of the rows (or columns)

The calculation of the DFT of a row (or column) longer than r members normally involves the recursive treatment of that sequence. Figure 4.5 shows a common way of performing the decimation by r for a sequence of length N. The sequence is decimated such that the index of $x(n)$ increases by one in one dimension (vertical in this case), and increases by r in the other dimension. Radix-r butterflies calculate the column-wise DFTs. The row-wise DFTs are, in general, longer than r and therefore require further decimations.

During the decimation of the sequences, a single sequence is folded into r sequences that are $1/r$ the length of the previous sequence. The final decimation ends with an $r \times r$ array where radix-r butterflies calculate DFTs of both columns and rows.

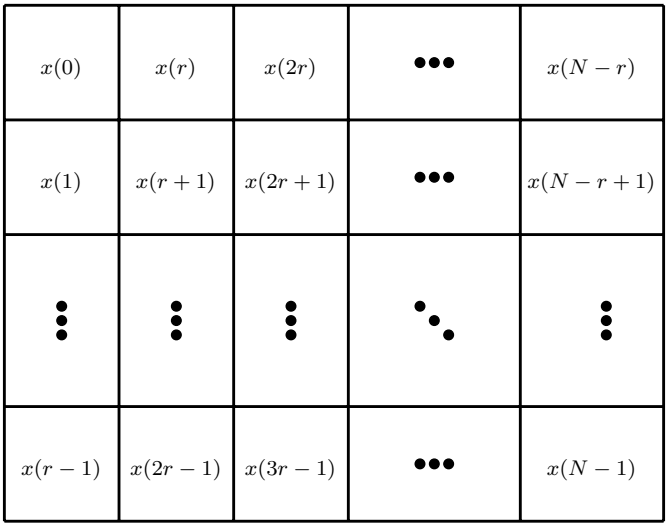


Figure 4.5: Decimation of an N-point sequence into an $(N/r) \times r$ array

In general then, the first decimation produces columns with index spacings of one, which are calculated by a radix-r butterfly with a *stride* of one. Assuming a large N, the second decimation is performed on each of the r rows and results in r separate $(N/r^2) \times r$ arrays where the columns now have strides of r. The third decimation produces r^2 arrays where the columns have strides of r^2. Therefore, the k^{th} decimation ($k = 1, 2, \ldots$) produces columns with strides of r^{k-1} and the final decimation (out of $\log_r(N) - 1$ total) has columns with strides of $r^{\log_r N - 1 - 1} = r^{\log_r N}/r^2 = N/r^2$ and rows with strides of N/r.

Therefore, since the stride of each column after the k^{th} decimation ($k-1^{th}$ stage) is r^{k-1} and the final decimation produces columns with a stride of N/r^2 and rows with a stride of N/r, a dataflow diagram can be drawn with the stride of its butterflies increasing by a factor r from stage to stage. QED

Table 4.1 summarizes the strides of butterflies at different stages, as described in the previous proof.

Stage	0	1	2	...	$\log_r(N)-2$	$\log_r(N)-1$
Stride	1	r	r^2	...	N/r^2	N/r

Table 4.1: Strides of butterflies across $\log_r N$ stages

Example 1 N**=16, radix-2 FFT**

We now illustrate Theorem 1's proof using $N = 16$ and a radix-2 decomposition. To simplify the illustration, twiddle factor multiplications are omitted.

Figure 4.6 shows the original data sequence with its 16 elements. First, the sequence

$x(0)$	$x(1)$	$x(2)$	$x(3)$	$x(4)$	$x(5)$	$x(6)$	$x(7)$	$x(8)$	$x(9)$	$x(10)$	$x(11)$	$x(12)$	$x(13)$	$x(14)$	$x(15)$

Figure 4.6: 16×1 $x(n)$ input sequence array

is re-organized (or decimated) into an 8×2 array as shown in Fig. 4.7. Since the data shown in this and subsequent diagrams are no longer elements of the input sequence x, but

$\hat{x}(0)$	$\hat{x}(2)$	$\hat{x}(4)$	$\hat{x}(6)$	$\hat{x}(8)$	$\hat{x}(10)$	$\hat{x}(12)$	$\hat{x}(14)$
$\hat{x}(1)$	$\hat{x}(3)$	$\hat{x}(5)$	$\hat{x}(7)$	$\hat{x}(9)$	$\hat{x}(11)$	$\hat{x}(13)$	$\hat{x}(15)$

Figure 4.7: 8×2 $\hat{x}(n)$ intermediate data array

rather are intermediate results, they are written as $\hat{x}$. Element numbers of $\hat{x}$ correspond to the element numbers of x which occupied corresponding memory locations. DFTs are performed on the columns using radix-2 butterflies with a stride of one. The input/output locations of butterflies are indicated in the figures with heavy lines. Next, the 8-element rows are decimated into 4×2 arrays. The results of only the first row of Fig. 4.7 are shown

$\hat{x}(0)$	$\hat{x}(4)$	$\hat{x}(8)$	$\hat{x}(12)$
$\hat{x}(2)$	$\hat{x}(6)$	$\hat{x}(10)$	$\hat{x}(14)$

Figure 4.8: 4×2 $\hat{x}(n)$ intermediate data array

in Fig. 4.8. Again, the DFTs of the columns are calculated with radix-2 butterflies, but this time with a stride of two. The third decimation results in 2×2 arrays. The results of the decimation of only the first row of the data in Fig. 4.8 are shown in Fig. 4.9. The FFT

$\hat{x}(0)$	$\hat{x}(8)$
$\hat{x}(4)$	$\hat{x}(12)$

Figure 4.9: 2×2 $\hat{x}(n)$ intermediate data array

can now be completed with radix-2 butterflies of the columns and rows. The stride of the columns is four and the stride of the rows is eight.

The strides of increasing butterfly stages throughout the FFT are thus: 1, 2, 4, 8; which increase by a factor $r = 2$. ◁

Example 2 $N = 27$**, radix-3 FFT**

A second example uses $N = 27 = 3 \times 3 \times 3$ and a radix-3 decomposition. Figure 4.10 shows the results of the first decimation of the 27-element input sequence x. The stride of the radix-3 column butterflies is one. The 9-element rows are then decimated by $r = 3$ as shown in Fig. 4.11. The stride of the resulting columns is three and the stride of the final rows is nine. In summary, the strides of subsequent butterfly stages throughout the FFT are: 1, 3, 9; which increase by a factor $r = 3$. ◁

$\hat{x}(0)$	$\hat{x}(3)$	$\hat{x}(6)$	$\hat{x}(9)$	$\hat{x}(12)$	$\hat{x}(15)$	$\hat{x}(18)$	$\hat{x}(21)$	$\hat{x}(24)$
$\hat{x}(1)$	$\hat{x}(4)$	$\hat{x}(7)$	$\hat{x}(10)$	$\hat{x}(13)$	$\hat{x}(16)$	$\hat{x}(19)$	$\hat{x}(22)$	$\hat{x}(25)$
$\hat{x}(2)$	$\hat{x}(5)$	$\hat{x}(8)$	$\hat{x}(11)$	$\hat{x}(14)$	$\hat{x}(17)$	$\hat{x}(20)$	$\hat{x}(23)$	$\hat{x}(26)$

Figure 4.10: 9×3 $\hat{x}(n)$ intermediate data array

$\hat{x}(0)$	$\hat{x}(9)$	$\hat{x}(18)$
$\hat{x}(3)$	$\hat{x}(12)$	$\hat{x}(21)$
$\hat{x}(6)$	$\hat{x}(15)$	$\hat{x}(24)$

Figure 4.11: 3×3 $\hat{x}(n)$ intermediate data array

Corollary *An RRI-FFT dataflow diagram can also be drawn with its stride decreasing by r through succeeding stages (given without proof).*

4.5 Existence of the Cached-FFT

Theorem 2 *All butterflies in an RRI-FFT (except those in the last stage), share at most one output with the inputs of any single butterfly in the subsequent stage.*

Proof If the stride of butterflies in stage s is $stride_s$, then, from Theorem 1, the stride of butterflies in stage $s+1$ is $stride_{s+1} = stride_s \cdot r$. From Eq. 4.1, butterflies in stage s have a span of $(r-1)stride_s$. Since the span of the s-stage butterflies is less than the stride of the $s+1$-stage butterflies, it is impossible for a butterfly in stage $s+1$ to share an input/output with more than one leg of a butterfly in stage s. QED

Figure 4.12 shows the positioning of these two butterflies with respect to each other.

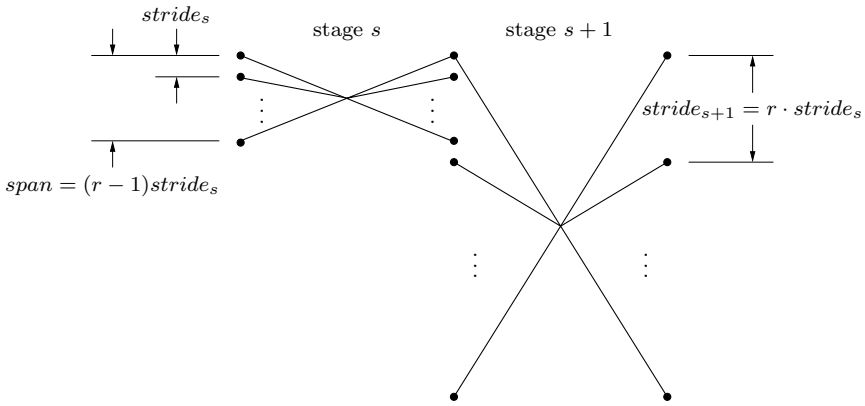


Figure 4.12: *Stride* and *span* of arbitrary-radix butterflies

Example 3 $N = 27$, **radix-3 FFT**

To illustrate Theorem 2, Fig. 4.13 shows two radix-3 butterflies in adjacent stages where a span of two in stage s and a stride of three in stage $s+1$ clearly prevent more than one output/input from coinciding. ◁

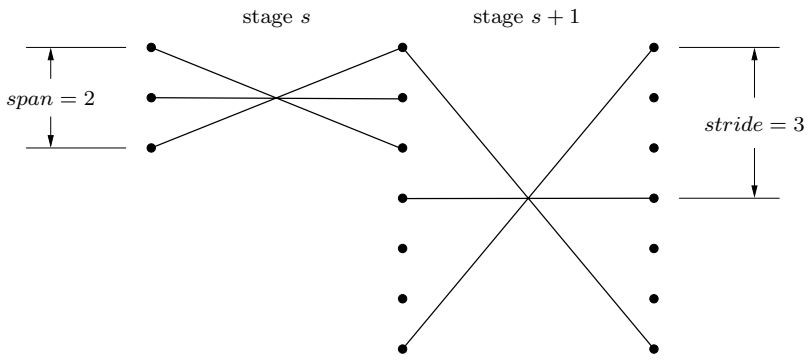


Figure 4.13: *Stride* and *span* of radix-3 butterflies

Corollary *All butterflies in an RRI-FFT (except those in the last stage), share each of their r outputs with one input of r different butterflies in the subsequent stage (given without proof).*

Theorem 3 *In c contiguous stages of an RRI-FFT, it is always possible to find r^{c-1} butterflies per stage ($c \cdot r^{c-1}$ butterflies total) that can be calculated using r^c memory locations.*

Proof Assuming s and $s+1$ are valid stage numbers of an RRI-FFT, from Theorem 2, each butterfly in stage $s+1$ shares an input with r separate butterflies in stage s. These r butterflies in stage s are located every $span_s + stride_s$ words as shown in Fig. 4.14. The k^{th} inputs or outputs of these butterflies are then also spaced every $span_s + stride_s$ words. Because $span_s + stride_s$ is equal to $stride_{s+1}$, a single butterfly in stage $s+1$ exists which shares its inputs with the k^{th} outputs of the r butterflies in stage s. Therefore, there exist r butterflies in stage $s+1$ whose $r \cdot r$ inputs match all $r \cdot r$ outputs of r butterflies in stage s.

It has been shown that over two stages, r butterflies per stage can be calculated using r^2 memory locations. Assuming $s+2$ is a valid stage number, the process can be repeated recursively for butterflies in stage $s+2$, which results in the calculation of r^2 butterflies using r^3 memory locations over three stages.

As each additional stage adds a factor of r butterflies and a factor of r memory locations, by induction, it is possible to calculate r^{c-1} butterflies using r^c memory locations over c stages. QED

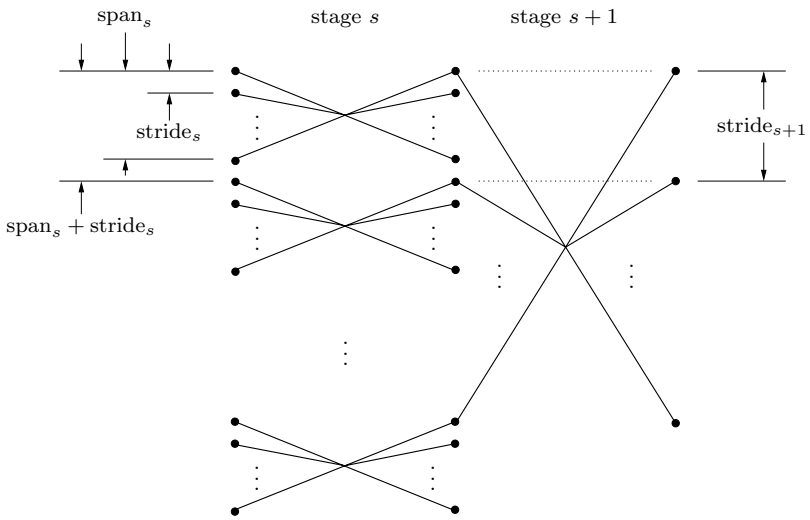


Figure 4.14: Extended diagram of the *stride* and *span* of arbitrary-radix butterflies

4.6 A General Description of the RRI-FFT

Descriptions of FFT algorithms in the literature typically consist of a dataflow diagram for a specific case, and a simple software program. This section provides a general description of the RRI-FFT which more readily provides insight into the algorithm and extensions to it.

We begin by closely examining the memory access pattern of the RRI-FFT dataflow diagram, shown in Fig. 4.2 on page 57, and recognizing that butterflies are clustered into "groups." For example, in stage 2, butterflies are clustered into eight "groups" of four butterflies each. Because of the unique clustering of butterflies into groups, a concise description of the memory accesses makes use of two counters:

- a *group* counter—which counts groups of butterflies
- a *butterfly* counter—which counts butterflies within a group

Table 4.2 contains a generalized listing of the number of groups per stage and the number of butterflies per group, as well as the number of digits required for each counter. The rightmost column contains the sum of the *group* counter bits and the *butterfly* counter bits and is constant across all stages. The constant sum is clearly necessary since both counters

Stage number	Groups per stage	Digits in *group* counter	Butterflies per group	Digits in *butterfly* counter	Total counter bits
0	N/r	$\log_r(N)-1$	1	0	$\log_r(N)-1$
1	N/r^2	$\log_r(N)-2$	r	1	$\log_r(N)-1$
2	N/r^3	$\log_r(N)-3$	r^2	2	$\log_r(N)-1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_r(N)-2$	r	1	N/r^2	$\log_r(N)-2$	$\log_r(N)-1$
$\log_r(N)-1$	1	0	N/r	$\log_r(N)-1$	$\log_r(N)-1$

Table 4.2: *Butterfly* counter and *group* counter information describing the memory access pattern for an N-point, radix-r, DIT RRI-FFT

together address a single butterfly within a stage, and the number of butterflies per stage is constant (N/r).

To fully describe the RRI-FFT, it is necessary to specify the memory access patterns and the values of the W_N butterfly coefficients. Table 4.3 shows the values of digits in the *group* and *butterfly* counters which are needed to generate memory addresses. The asterisks represent digit positions where different values in that position address the r inputs and r outputs of the butterflies. For example, in the radix-2 case, a "0" in place of the asterisk addresses the A input and X output of the butterfly, and a "1" in the asterisk's position addresses the B input and Y output. In the rightmost column of the table, b_k digits generate exponents for the W_N coefficients.

Note that the table does not specify a particular order in which to calculate the FFT. The subscripts of the counter digits g and b show only the relationship between address digits and W_N-generating digits. The digits can be incremented in any order or pattern—the only requirement is that all N/r butterflies are calculated once and only once. As shown in Sec. 4.7, the development of the cached-FFT relies on this property. Of course, the order of calculation is not entirely arbitrary, as a butterfly's inputs must be calculated before it can calculate its outputs.

Although the radix-2 approach is easiest to visualize, the description given by the tables applies to algorithms with other radices as well. For use with radices other than two, the

Stage number	Digits in *group* counter	Digits in *butterfly* counter	Butterfly addresses (x = one digit)	W_N exponents
0	$\log_r(N)-1$	0	$\underline{g_{\log_r(N)-2}}\ \underline{g_{\log_r(N)-3}}\ \cdots\ \underline{g_2}\ \underline{g_1}\ \underline{g_0}\ \underline{*}$	$000\cdots00$
1	$\log_r(N)-2$	1	$\underline{g_{\log_r(N)-3}}\ \underline{g_{\log_r(N)-4}}\ \cdots\ \underline{g_1}\ \underline{g_0}\ \underline{*}\ \underline{b_0}$	$b_000\cdots00$
2	$\log_r(N)-3$	2	$\underline{g_{\log_r(N)-4}}\ \underline{g_{\log_r(N)-5}}\ \cdots\ \underline{g_0}\ \underline{*}\ \underline{b_1}\ \underline{b_0}$	$b_1b_00\cdots00$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_r(N)-2$	1	$\log_r(N)-2$	$\underline{g_0}\ \underline{*}\ \underline{b_{\log_r(N)-3}}\ \cdots\ \underline{b_2}\ \underline{b_1}\ \underline{b_0}$	$b_{\log_r(N)-3}b_{\log_r(N)-4}\cdots b_00$
$\log_r(N)-1$	0	$\log_r(N)-1$	$\underline{*}\ \underline{b_{\log_r(N)-2}}\ \underline{b_{\log_r(N)-3}}\ \cdots\ \underline{b_2}\ \underline{b_1}\ \underline{b_0}$	$b_{\log_r(N)-2}b_{\log_r(N)-3}\cdots b_1b_0$

Table 4.3: Addresses and base W_N exponents for an N-point, radix-r, DIT RRI-FFT. The variables g_k and b_k represent the k^{th} digits of the *group* and *butterfly* counters respectively. Asterisks ($*$) represent digit positions where the r possible values address the r butterfly inputs and r outputs.

counter digits (*e.g.,* g_1, $*$, b_0,...) are interpreted as base-r digits instead of binary digits ($\in [0, 1]$) as in the radix-2 case. Higher-radix algorithms generally require more than one W_N coefficient, so although Table 4.3 gives only one W_N, it serves as a base factor where other coefficients are normally multiples of the given value.

Example 4 $N = 64$ **Radix-2 RRI-FFT**

Table 4.4 details the generation of butterfly addresses and W_N coefficients for a 64-point, radix-2, DIT RRI-FFT. Because $r = 2$, there are $\log_2 64 = 6$ stages. Since the example uses a radix-2 decomposition, all counter places are binary digits. ◁

Stage number	Butterfly address digits (x = one digit)	W_N butterfly coefficients
stage 0	g_4 g_3 g_2 g_1 g_0 $*$	W_{64}^{00000}
stage 1	g_3 g_2 g_1 g_0 $*$ b_0	$W_{64}^{b_0 0000}$
stage 2	g_2 g_1 g_0 $*$ b_1 b_0	$W_{64}^{b_1 b_0 000}$
stage 3	g_1 g_0 $*$ b_2 b_1 b_0	$W_{64}^{b_2 b_1 b_0 00}$
stage 4	g_0 $*$ b_3 b_2 b_1 b_0	$W_{64}^{b_3 b_2 b_1 b_0 0}$
stage 5	$*$ b_4 b_3 b_2 b_1 b_0	$W_{64}^{b_4 b_3 b_2 b_1 b_0}$

Table 4.4: Addresses and W_N coefficients for a 64-point, radix-2, DIT FFT

4.7 A General Description of the Cached-FFT

As previously mentioned, the derivation of the cached-FFT is greatly simplified by viewing the cached-FFT as a modified RRI-FFT. To further simplify the development, we initially consider only *balanced* cached-FFTs. Section 4.7.2 addresses the handling of unbalanced cached-FFTs. Further, we consider only DIT decompositions, and note that the development of DIF variations begins with a DIF form of the RRI-FFT and is essentially identical.

First, a review of two key points regarding the counter digits in Table 4.3: (i) the sum of the number of digits in the *group* and *butterfly* counters is constant, and (ii) the counter

digits can be incremented in any order, as long as every butterfly is executed once and only once.

The goal is to find a grouping of the memory accesses such that a portion of the full FFT can be calculated using fewer than N words of memory. Table 4.5 shows one re-labeling of the counter digits that achieves this goal. Fixing the number of bits in the *group* and *butterfly* counters, and keeping the positions of the *group* counter digits fixed across an epoch allows a subset of the FFT to be calculated in $r^{\log_r(C)} = C$ memory locations.

To increase readability, Table 4.6 repeats the information in Table 4.5 but removes redundant labeling of the *group* counter positions.

Another key point of departure of the cached-FFT from the RRI-FFT is the renaming of the RRI-FFT's stages, which are now identified by an *epoch* number and a *pass* number. The renaming reinforces the following features of the cached-FFT:

- Across any epoch, the positions and values of the *group* counter digits are constant.
- Within each epoch, the memory address pattern is identical for the $\log_r(C)-1$ address digits not connected to the *group* counter. These digits are the $\log_r(C) - 2$ *butterfly* digits plus the one "$*$" digit.

The W_N coefficients are generated using the same method that the RRI-FFT uses, except that the new *group* and *butterfly* mappings are used. Table 4.7 shows how W_N exponents are formed.

Example 5 $N = 64$**,** $E = 2$**, Radix-2 Cached-FFT**

To illustrate how the cached-FFT works, we now consider a cached-FFT implementation of the $N = 64$, radix-2, DIT FFT considered in Example 4. We choose two epochs ($E = 2$) for this cached-FFT example. Following the format of information for the RRI-FFT given in Table 4.4, Table 4.8 provides the address digit positions and W_N coefficients for a 64-point cached-FFT.

Two epochs of three passes each replace the six stages of the RRI-FFT. The three *group* counter digits (g_2, g_1, g_0) are fixed across both epochs. The *butterfly* counter and asterisk digit $(b_1, b_0, *)$ positions are the same in both epochs. These characteristics enable the algorithm to be used efficiently on a processor with a cache memory.

One step in adapting the algorithm to a hierarchical memory structure is the separation of the data transactions between main memory and the cache, and transactions between the

Epoch number	Pass number	Butterfly Addresses ($\underline{\texttt{x}}$ = one digit)			
0	0	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{b_0}\ \underline{*}$
	1	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{*}\ \underline{b_0}$
	$\vdots$	$\vdots$		$\vdots$	$\vdots$
	$\log_r(N)/E-1$	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$	$\underline{*} \cdots \underline{b_2}\ \underline{b_1}\ \underline{b_0}$
1	0	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{b_0}\ \underline{*}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$
	1	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{*}\ \underline{b_0}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$
	$\vdots$	$\vdots$		$\vdots$	$\vdots$
	$\log_r(N)/E-1$	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{*} \cdots \underline{b_2}\ \underline{b_1}\ \underline{b_0}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$E-1$	0	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{b_0}\ \underline{*}$	$\cdots$	$\underline{g_{2\cdot\log_r(C)-1}} \cdots \underline{g_{\log_r C}}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$
	1	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{*}\ \underline{b_0}$	$\cdots$	$\underline{g_{2\cdot\log_r(C)-1}} \cdots \underline{g_{\log_r C}}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$
	$\vdots$	$\vdots$		$\vdots$	$\vdots$
	$\log_r(N)/E-1$	$\underline{*} \cdots \underline{b_2}\ \underline{b_1}\ \underline{b_0}$	$\cdots$	$\underline{g_{2\cdot\log_r(C)-1}} \cdots \underline{g_{\log_r C}}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$

Table 4.5: Memory addresses for an N-point, balanced, radix-r, DIT cached-FFT. The variables g_k and b_k represent the k^{th} digits of the *group* and *butterfly* counters respectively. Asterisks ($*$) represent digit positions where the r possible values address the r butterfly inputs and r outputs.

Epoch number	Pass number	Butterfly Addresses ($\underline{\texttt{x}}$ = one digit)			
0	0	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{b_0}\ \underline{*}$
	1				$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{*}\ \underline{b_0}$
	$\vdots$				$\vdots$
	$\log_r(N)/E-1$				$\underline{*} \quad \ldots \quad \underline{b_2}\ \underline{b_1}\ \underline{b_0}$
1	0	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{b_0}\ \underline{*}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$
	1			$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{*}\ \underline{b_0}$	
	$\vdots$			$\vdots$	
	$\log_r(N)/E-1$			$\underline{*} \quad \ldots \quad \underline{b_2}\ \underline{b_1}\ \underline{b_0}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$E-1$	0	$\underline{b_{\log_r(C)-2}} \quad \cdots \quad \underline{b_1}\ \underline{b_0}\ \underline{*}$	$\cdots$	$\underline{g_{2\cdot\log_r(C)-1}} \cdots \underline{g_{\log_r C}}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$
	1	$\underline{b_{\log_r(C)-2}} \quad \cdots \quad \underline{b_1}\ \underline{*}\ \underline{b_0}$			
	$\vdots$	$\vdots$			
	$\log_r(N)/E-1$	$\underline{*} \quad \ldots \quad \underline{b_2}\ \underline{b_1}\ \underline{b_0}$			

Table 4.6: A simplified view of the memory addresses shown in Table 4.5 that is possible because the positions of digits in the *group* counter are constant across epochs.

Epoch number	Pass number	W_N exponents
0	0	$000\cdots000$
	1	$b_000\cdots000$
	2	$b_1b_00\cdots000$
	$\vdots$	$\vdots$
	$\log_r(N)/E-1$	$b_{\log_r(C)-2}\cdots b_1b_00\cdots00$
1	0	$g_{\log_r(C)-1}\cdots g_1g_00\cdots00$
	1	$b_0g_{\log_r(C)-1}\cdots g_1g_00\cdots00$
	2	$b_1b_0g_{\log_r(C)-1}\cdots g_1g_00\cdots00$
	$\vdots$	$\vdots$
	$\log_r(N)/E-1$	$b_{\log_r(C)-2}\cdots b_1b_0g_{\log_r(C)-1}\cdots g_1g_00\cdots00$
$\vdots$	$\vdots$	$\vdots$
$E-1$	0	$g_{\log_r(N/C)-1}\cdots g_1g_00\cdots00$
	1	$b_0g_{\log_r(N/C)-1}\cdots g_1g_00\cdots00$
	2	$b_1b_0g_{\log_r(N/C)-1}\cdots g_1g_00\cdots00$
	$\vdots$	$\vdots$
	$\log_r(N)/E-1$	$b_{\log_r(C)-2}\cdots b_1b_0g_{\log_r(N/C)-1}\cdots g_1g_0$

Table 4.7: Base W_N coefficients for an N-point, balanced, radix-r, DIT cached-FFT. The variables g_k and b_k represent the k^{th} digits of the *group* and *butterfly* counters respectively.

Epoch number	Pass number	Butterfly address digits (x = one digit)	W_N butterfly coefficients
0	0	$g_2\ g_1\ g_0\ b_1\ b_0\ *$	W_{64}^{00000}
	1	$g_2\ g_1\ g_0\ b_1\ *\ b_0$	$W_{64}^{b_0 0000}$
	2	$g_2\ g_1\ g_0\ *\ b_1\ b_0$	$W_{64}^{b_1 b_0 000}$
1	0	$b_1\ b_0\ *\ g_2\ g_1\ g_0$	$W_{64}^{g_2 g_1 g_0 00}$
	1	$b_1\ *\ b_0\ g_2\ g_1\ g_0$	$W_{64}^{b_0 g_2 g_1 g_0 0}$
	2	$*\ b_1\ b_0\ g_2\ g_1\ g_0$	$W_{64}^{b_1 b_0 g_2 g_1 g_0}$

Table 4.8: Addresses and W_N coefficients for a 64-point, radix-2, DIT, 2-epoch cached-FFT

Epoch number	Memory address digits (x = one digit)	Cache address digits (x = one digit)
0	$g_2\ g_1\ g_0\ *\ *\ *$	$*\ *\ *$
1	$*\ *\ *\ g_2\ g_1\ g_0$	$*\ *\ *$

Table 4.9: Main memory and cache addresses used to load and flush the cache—for a 64-point, radix-2, DIT, 2-epoch cached-FFT

cache and the processor. Table 4.9 shows how the *group* counter digits generate memory addresses for the $r^3 = 2^3 = 8$ words that are accessed each time the caches are loaded or flushed.

Another step is the generation of cache addresses used to access data for butterfly execution. Table 4.10 shows the cache addresses generated by the *butterfly* counter digits. Again, cache addresses are the same across both epochs. Table 4.10 also shows the counter digits used to generate W_N coefficients. The rightmost column of the table shows how W_N values are calculated using both *group* and *butterfly* digits.

Figure 4.15 shows the flow graph of the 64-point cached-FFT. Radix-2 butterflies are

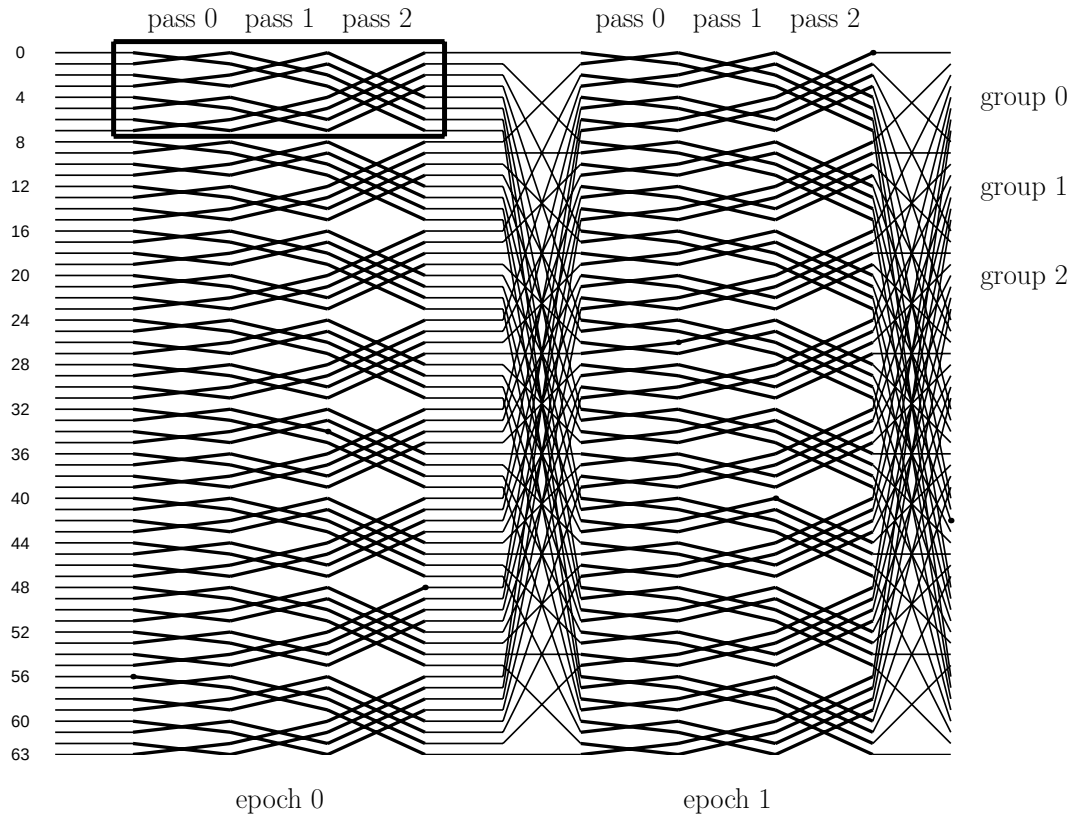


Figure 4.15: Cached-FFT dataflow diagram

Epoch number	Pass number	Cache address digits (x = one digit)	W_N butterfly coefficients
0	0	$\underline{b_1}\ \underline{b_0}\ \underline{*}$	W_{64}^{00000}
	1	$\underline{b_1}\ \underline{*}\ \underline{b_0}$	$W_{64}^{b_0 0000}$
	2	$\underline{*}\ \underline{b_1}\ \underline{b_0}$	$W_{64}^{b_1 b_0 000}$
1	0	$\underline{b_1}\ \underline{b_0}\ \underline{*}$	$W_{64}^{g_2 g_1 g_0 00}$
	1	$\underline{b_1}\ \underline{*}\ \underline{b_0}$	$W_{64}^{b_0 g_2 g_1 g_0 0}$
	2	$\underline{*}\ \underline{b_1}\ \underline{b_0}$	$W_{64}^{b_1 b_0 g_2 g_1 g_0}$

Table 4.10: Cache addresses and W_N coefficients for a 64-point, radix-2, DIT, 2-epoch cached-FFT

drawn with heavier lines, and transactions between main memory and the cache—which involve no computation—are drawn with lighter-weight lines. A box encloses an 8-word group to show which butterflies are calculated together from the cache. ◁

4.7.1 Implementing the Cached-FFT

As with any FFT transform, the length (N) and radix (r) must be specified. The cached-FFT also requires the selection of the number of epochs (E).

Although the computed variables presented below are derived from N, r, and E—and are therefore unnecessary—we introduce new variables to clarify the implementation of the algorithm.

Calculating the number of passes per group

For a balanced cached-FFT, the number of passes per group can be found by,

$$NumPassesPerGroup = \log_r(N)/E. \tag{4.2}$$

For unbalanced cached-FFTs, the number of passes per group varies across epochs, so there is no single global value. Though the sum of the number of passes per group over all epochs must still equal $\log_r N$, the passes are not uniformly allocated across epochs.

Calculating the cache size, C

From Eq. 4.2 and Theorem 3, the cache size, C, is,

$$C = r^{\left(\frac{\log_r N}{E}\right)} \tag{4.3}$$

$$C = \left[r^{(\log_r N)}\right]^{1/E} \tag{4.4}$$

$$C = N^{1/E} \tag{4.5}$$

$$C = \sqrt[E]{N}. \tag{4.6}$$

For an unbalanced cached-FFT, the cache memory must accommodate the largest number of passes in an epoch, which (excluding pathological cases) is $\lceil \log_r(N)/E \rceil$. Again using Theorem 3, the cache size, C, is calculated by,

$$C = r^{\left\lceil \frac{\log_r N}{E} \right\rceil}. \tag{4.7}$$

Other variables

For balanced cached-FFTs,

$$NumGroupsPerEpoch = N/C \tag{4.8}$$

$$NumButterfliesPerPass = C/r. \tag{4.9}$$

Cases with fixed cache sizes

In some cases, the cache size and the transform length are fixed, and the number of epochs must be determined. Since the cache size is not necessarily a power of r, a maximum of $\lfloor \log_r C \rfloor$ passes can be calculated from data in the cache. To attain maximum reusability of data in the cache, the minimum number of epochs is desired. As the number of epochs must be an integer, the expression for E is then,

$$E = \left\lceil \frac{\log_r N}{\lfloor \log_r C \rfloor} \right\rceil. \tag{4.10}$$

Pseudo-code algorithm flow

```
for e = 0 to E-1
  for g = 0 to NumGroupsPerEpoch-1
    Load_Cache(e,g);
    for p = 0 to NumPassesPerGroup-1
      for b = 0 to NumButterfliesPerPass-1
         [X,Y,...] = butterfly[A,B,...];
      end
    end
    Dump_Cache(e,g);
  end
end

Load_Cache(e,g)
for i = 0 to C-1
  CACHE[AddrCache] = MEM[AddrMem];
end

Dump_Cache(e,g)
for i = 0 to C-1
  MEM[AddrMem] = CACHE[AddrCache];
end
```

4.7.2 Unbalanced Cached-FFTs

Unbalanced cached-FFTs do not have a constant number of passes in the groups of each epoch. The quantity $\sqrt[E]{N}$ is also not an integer (excluding pathological cases).

We develop unbalanced cached-FFTs by first constructing a cached-FFT algorithm for the next longer transform length (N') where $\sqrt[E]{N'}$ is an integer. For radix-r cached-FFTs, $N' = N \cdot r^k$, where k is a small integer. After the length-N' transform has been designed, the appropriate k rows of the address-generation table are removed and the *epoch* and *pass* numbers are reassigned. Which k rows are removed depends on how the cached-FFT was formulated. However, in all cases, the k additional rows which must be removed to make an N-point transform from the N'-point transform are unique. The remaining passes can be placed into any epoch and the resulting transform is not unique.

The main disadvantage of unbalanced cached-FFTs is that they introduce additional controller complexity.

4.7.3 Reduction of Memory Traffic

By holding frequently-used data, the data cache reduces traffic to main memory several fold. Without a cache, data are read from and written to main memory $\log_r N$ times. With a cache, data are read and written to main memory only E times. Therefore, the reduction is memory traffic is,

$$\textit{Memory traffic reduction multiple} = \log_r(N)/E. \tag{4.11}$$

This reduction in memory traffic enables more processors to work from a unified main memory and/or the use of a slower lower-power main memory. In either case, power dissipated accessing data can be decreased since a smaller memory that may be located nearer to the datapath stores the data words.

4.7.4 Calculating Multiple Transform Lengths

It is sometimes desirable to calculate transforms of different lengths using a processor with a fixed cache size. The procedure is simplified by first formulating the cached-FFT algorithm for the longest desired transform length, and then shortening the length to realize shorter transforms. Since it is possible to calculate r transforms of length N/r by simply omitting one radix-r decimation from the formulation of the FFT, we can calculate multiple shorter-length transforms by removing a stage from an FFT computation. For a cached-FFT, removing a stage corresponds to the removal of a pass, which involves changing the number of passes per group and/or the number of epochs. Depending on the particular case and the number of passes that are removed, the resulting transform may be unbalanced.

Since the amount of time spent by the processor executing from the cache decreases as the number of passes per group decreases, the processor may stall due to main memory bandwidth limitations for short-length transforms. However, in all cases, throughput should never decrease.

4.7.5 Variations of the Cached-FFT

Although the description of the cached-FFT given in this section is sufficient to generate a wide variety of cached-FFT algorithms, additional variations are possible. While not described in detail, a brief overview of a few possible variations follows.

Many variations to the cached-FFT are possible by varying the placement of data words in main memory and the cache. Partitioning the main memory and/or cache into multiple banks will increase memory bandwidth and alter the memory address mappings.

Although FFT algorithms scramble either the input or output data into bit-reversed order, it is normally desirable to work only with normal-order data. The cached-FFT algorithm offers additional flexibility in sorting data into normal order compared to a non-cached algorithm. Depending on the particular design, it may be possible to overlap input and output operations in normal order with little or no additional buffer memories.

Another class of variations to the cached-FFT is used in applications which contain more than two levels of memory. For these cases, multiple levels of cached-FFTs are constructed where a *group* from one cached-FFT is calculated by a full cached-FFT in a higher-level of the memory hierarchy (*i.e.,* a smaller, faster memory). However, the whole problem is more clearly envisioned by constructing a multi-level cached-FFT where, in addition to normal groupings, memory address digits are simply formed into smaller groups which fit into higher memory hierarchies.

4.7.6 Comments on Cache Design

For systems specifically designed to calculate cached-FFTs, the full complexity of a general-purpose cache is unnecessary. The FFT is a fully deterministic algorithm and the flow of execution is data-independent. Therefore, cache tags and associated tag circuitry are unnecessary.

Furthermore, since memory addresses are known *a priori*, pre-fetching data from memory into the cache enables higher processor utilization.

4.8 Software Implementations

Although implemented in a dedicated FFT processor in this dissertation, the cached-FFT algorithm can also be implemented in software form on programmable processors.

The algorithm is more likely to be of value for processors which have a memory hierarchy where higher levels of memory are faster and/or lower power than lower levels, and where the highest-level memory is smaller than the whole data set.

Example 6 Programmable DSP processor

The Mitsubishi D30V DSP processor (Holmann, 1996) utilizes a Very Long Instruction Word (VLIW) architecture and is able to issue up to two instructions per cycle. Examples of single instructions include: multiply-accumulate, addition, subtraction, complex-word load, and complex-word store. Sixty-three registers can store data, pointers, counters, and other local variables.

The calculation of a complex radix-2 butterfly requires seven instructions, which are spent as follows:

1 cycle: Update memory address pointers

1 cycle: Load A and B into registers

4 cycles: Four $B_{real,imag} \cdot W_{real,imag}$ multiplications, two $A + BW$ additions, and two $A - BW$ subtractions

1 cycle: Store X and Y back to memory

The processor calculates 256-point IFFTs while performing MPEG-2 decoding. The core butterfly calculations of a 256-point FFT require $7 \cdot 256/2 \cdot \log_2 256 = 7168$ cycles.

If a cached-FFT algorithm is used with two epochs ($E = 2$), from Eq. 4.6, the size of the cache is then $C = \sqrt[E]{N} = \sqrt[2]{256} = 16$ complex words—which requires 32 registers. The register file caches data words from memory.

From Eq. 4.2, there are $\log_r(N)/E = \log_2(256)/2 = 4$ passes per group. The first pass of each group does not require stores, the middle two passes do not need loads or stores, and the final pass does not require loads. The first and last passes of each group require one instruction to update a memory pointer. Pointer-update instructions can be grouped among pairs of butterflies and considered to consume 0.5 cycles each. The total number of cycles required is then $(5.5 + 4 + 4 + 5.5) \cdot 256/2 \cdot 2 = 4864$ cycles, which is a $1 - 4864/7168 = 32\%$ reduction in computation time, or a $1.47\times$ speedup! ◁

4.9 Summary

This chapter introduces the cached-FFT algorithm and describes a procedure for its implementation.

The cached-FFT is designed to operate with a hierarchical memory system where a smaller, higher-level memory supports faster accesses than the larger, lower-level memory. New terms to describe the cached-FFT are given, including: *epoch*, *group*, and *pass*.

The development of the cached-FFT is based on a particularly regular FFT which we develop and call the RRI-FFT. Using the RRI-FFT as a starting point, it is shown that in c contiguous stages of an FFT, it is possible to calculate r^{c-1} butterflies per stage using only r^c memory locations. This relationship is the basis of the cached-FFT.

Chapter 5

An Energy-Efficient, Single-Chip FFT Processor

Spiffee[1] is a single-chip, 1024-point, fast Fourier transform processor designed for low-power and high-performance operation.

This chapter begins with a number of key characteristics and goals of the processor, to establish a framework in which to consider design decisions. The remainder and bulk of the chapter presents details of the algorithmic, architectural, and physical designs of Spiffee. Where helpful, we also present design alternatives considered.

5.1 Key Characteristics and Goals

1024-point FFT

The processor calculates a 1024-point fast Fourier transform.

Complex data

In general, both input and output data have real and imaginary components.

[1] The name *Spiffee* is loosely derived from Stanford low-power, high-performance, FFT engine.

Simple data format

In order to simplify the design, data are represented in a fixed-point notation. Internal datapaths maintain precision with widths varying from 20 to 24 bits.

Emphasis on energy-efficiency

For systems which calculate a parallelizable algorithm, such as the FFT, an opportunity exists to operate with both high energy-efficiency and high performance through the use of parallel, energy-efficient, processors (see *Parallelizing and pipelining*, page 38). Thus, in this work, we place a larger emphasis on increasing the processor's energy-efficiency than on increasing its execution speed. To be more precise, our merit function is proportional to $energy^x \times time^y$ with $x > y$.

Low-V_t and high-V_t CMOS

Spiffee is designed to operate using either (i) low and tunable-threshold CMOS transistors (see *ULP CMOS*, page 43), or (ii) standard high-V_t transistors.

Robust operation with low supply voltages

When fabricated in ULP-CMOS, the processor is expected to operate at a supply voltage of 400 mV. Because the level of circuit noise under these conditions is not known, circuits and layout are designed to operate robustly in a noisy low-V_{dd} environment.

Single-chip

All components and testing circuitry fit on a single die.

Simple testing

Chip testing becomes much more difficult as chip I/O speeds increase beyond a few tens of MHz, due to board-design difficulties and chip tester costs which mushroom in the range of

50–125 MHz. Because a very low cost and easy to use tester[2] was readily available, Spiffee was designed so that no high-speed I/O signals are necessary to test the chip, even while running at full speed.

5.2 Algorithmic Design

5.2.1 Radix Selection

As stated in Sec. 1.1, simplicity and regularity are important factors in the design of a VLSI processor. Although higher-radix, prime-factor, and other FFT algorithms are shown in Sec. 2.4 to require fewer operations than radix-2, a radix-2 decomposition was nevertheless chosen for Spiffee with the expectation that the simpler form would result in a faster and more energy-efficient chip.

5.2.2 DIT vs. DIF

The two main types of radix-2 FFTs are the Decimation In Time (DIT) and Decimation In Frequency (DIF) varieties (Sec. 2.4.1). Both calculate two butterfly outputs, X and Y, from two butterfly inputs, A and B, and a complex coefficient W. The DIT approach calculates the outputs using the equations: $X = A + BW$ and $Y = A - BW$, while the DIF approach calculates its outputs using: $X = A + B$ and $Y = (A - B)W$. Because the DIT form is slightly more regular, it was chosen for Spiffee.

5.2.3 FFT Algorithm

Spiffee uses the cached-FFT algorithm presented in Ch. 4 because the algorithm supports both high-performance and low-power objectives, and it is well suited for VLSI implementations.

In order to simplify the design of the chip controller, only those numbers of epochs, E, which give *balanced* configurations with $N = 1024$ were considered. Since $\sqrt[3]{1024} \approx 10.1$

[2]The QDT (*Quick and Dirty Tester*) is controlled through the serial port of a computer and operates at a maximum speed of 500 vectors per second. It was originally designed by Dan Weinlader. Jim Burnham made improvements and laid out its PC board. An Irsim-compatible interface was written by the author.

and $\sqrt[4]{1024} \approx 5.7$ are not integers, and because $E \geq 5$ results in much smaller and less effective cache sizes, the processor was designed with two epochs. With $E = 2$, each word in main memory is read and written twice per transform. From Eq. 4.6, the cache size C is then,

$$\begin{aligned} C &= \sqrt[E]{N} && (5.1)\\ &= \sqrt[2]{1024} && (5.2)\\ &= 32 \text{ words.} && (5.3) \end{aligned}$$

Although this design easily supports multiple processors, Spiffee contains a single processor/cache pair and a single set of main memory. Using Eq. 4.11, the cached-FFT algorithm reduces traffic to main memory by several times:

$$\begin{aligned} \textit{Memory traffic reduction multiple} &= \log_r(N)/E && (5.4)\\ &= \log_2(1024)/2 && (5.5)\\ &= 5. && (5.6) \end{aligned}$$

The five-fold reduction in memory traffic permits the use of a much slower and lower-power main memory.

From Eqs. 4.2, 4.8, and 4.9 on page 76, Spiffee then has 32 groups per epoch, 5 passes per group, and 16 butterflies per pass.

5.2.4 Programmable vs. Dedicated Controller

There are two primary types of processor controllers: *programmable* controllers which execute instructions dynamically, and *dedicated* controllers whose behavior is inherent in the hardware design, and thus fixed. While the flexibility of a programmable processor is certainly desirable, its complexity is several times larger than that of a non-programmable processor. To keep Spiffee's complexity at a level that can be managed by one person, it uses a dedicated controller.

5.3 Architectural Design

5.3.1 Memory System Architecture

This section presents several memory system architectures used in FFT processors followed by a description of the one chosen for Spiffee. All example processors can calculate 1024-point complex FFTs.

Single-memory

The simplest memory system architecture is the *single-memory* architecture, in which a memory of at least N words connects to a processor by a bi-directional data bus. In general, data are read from and written back to the memory once for each of the $\log_r N$ stages of the FFT.

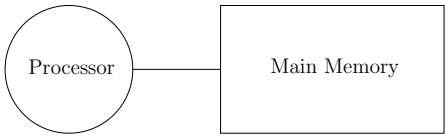


Figure 5.1: Single-memory architecture block diagram

Dual-memory

Processors using a *dual-memory* architecture connect to two memories of size N using separate busses. Input data begin in one memory and "ping-pong" through the processor from memory to memory $\log_r N$ times until the transform has been calculated. The Honeywell DASP processor (Magar *et al.*, 1988), and the Sharp LH9124 processor (Sharp Electronics Corporation, 1992) use the dual-memory architecture.

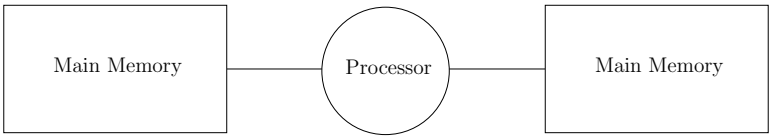


Figure 5.2: Dual-memory architecture block diagram

Pipeline

For processors using a *pipeline* architecture, a series of memories, which generally range in size from $\mathcal{O}(N)$ to a few words, replace N-word memories. Either physically or logically, there are $\log_r N$ stages. Figure 5.3 shows the flow of data through the pipeline structure and the interleaving of processors and buffer memories. Typically, an $\mathcal{O}(r)$-word memory is on one end of the pipeline, and memory sizes increase by r through subsequent stages, with the final memory of size $\mathcal{O}(N)$. The LSI L64280 FFT processor (Ruetz and Cai, 1990; LSI Logic Corporation, 1990a; LSI Logic Corporation, 1990b; LSI Logic Corporation, 1990c) and the FFT processor designed by He and Torkelson (1998) use pipeline architectures.

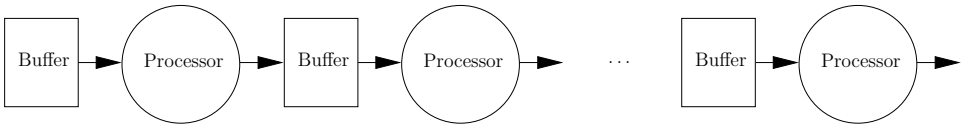


Figure 5.3: Pipeline architecture block diagram

Array

Processors using an *array* architecture comprise a number of independent processing elements with local buffers, interconnected through some type of network. The Cobra FFT processor (Sunada *et al.*, 1994) uses an array architecture and is composed of multiple chips which each contain one processor and one local buffer. The Plessey PDSP16510A FFT

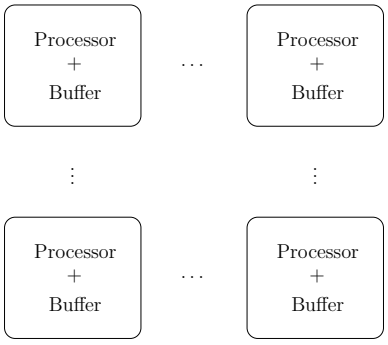


Figure 5.4: Array architecture block diagram

processor (GEC Plessey Semiconductors, 1993; O'Brien *et al.*, 1989) uses an array-style architecture with four datapaths and four memory banks on a single chip.

Cached-memory

The cached-memory architecture is similar to the single-memory architecture except that a small cache memory resides between the processor and main memory, as shown in Fig. 5.5. Spiffee uses the cached-memory architecture since a hierarchical memory system is necessary to realize the full benefits of the cached-FFT algorithm.

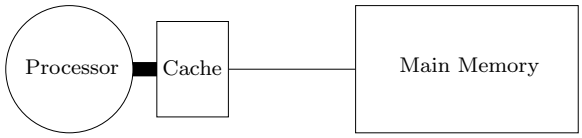


Figure 5.5: Cached-FFT processor block diagram

Performance of the memory system can be enhanced, as Fig. 5.6 illustrates, by adding a second cache *set*. In this configuration, the processor operates out of one cache set while the other set is being flushed and then loaded from memory. If the cache flush time plus load time is less than the time required to process data in the cache, which is easy to accomplish, then the processor need not wait for the cache between groups. The second cache set increases processor utilization and therefore overall performance, at the expense of some additional area and complexity.

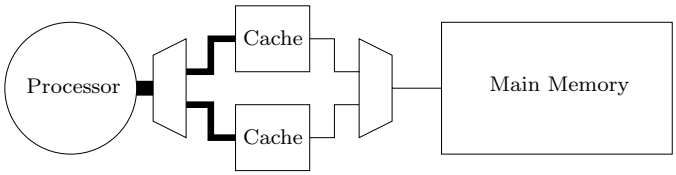


Figure 5.6: Block diagram of cached-memory architecture with two cache sets

Performance of the memory system shown in Fig. 5.6 can be further enhanced by partitioning each of the cache's two sets (0 and 1) into two *banks* (A and B), as shown in Fig. 5.7.

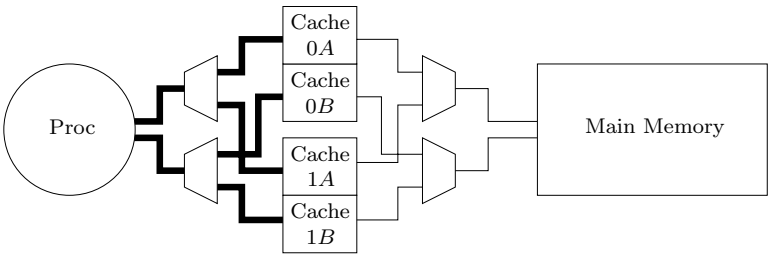


Figure 5.7: Block diagram of cached-memory architecture with two cache sets of two banks each

The double-banked arrangement increases throughput as it allows an increased number of cache accesses per cycle. Spiffee uses this double-set, double-bank architecture.

5.3.2 Pipeline Design

Because the state of an FFT processor is independent of datum values, a deeply-pipelined FFT processor is much less sensitive to pipeline hazards than is a deeply-pipelined general-purpose processor. Since clock speeds—and therefore throughput—can be dramatically increased with deeper pipelines that do not often stall, Spiffee has an aggressive cache→processor→cache pipeline. The cache→memory and memory→cache pipelines have somewhat-relaxed timings because the cached-FFT algorithm puts very light demands on the maximum cache flushing and loading latencies.

Datapath pipeline

Spiffee's nine-stage cache→processor→cache datapath pipeline is shown in Fig. 5.8. In the first pipeline stage, the input operands A and B are read from the appropriate cache set and W is read from memory. In pipeline stage two, operands are routed through two 2×2

Figure 5.8: Spiffee's nine-stage datapath pipeline diagram

crossbars to the correct functional units. Four $B_{\{real,imag\}} \times W_{\{real,imag\}}$ multiplications of the real and imaginary components of B and W are calculated in stages three through five. Stage six completes the complex multiplication by subtracting the real product $B_{imag} \times W_{imag}$ from $B_{real} \times W_{real}$ and adding the imaginary products $B_{real} \times W_{imag}$ and $B_{imag} \times W_{real}$. Stage seven performs the remaining additions or subtractions to calculate X and Y, and pipeline stages eight and nine complete the routing and write-back of the results to the cache.

Pipeline hazards

While deep pipelines offer high peak performance, any of several types of pipeline conflicts, or "hazards" (Hennessy and Patterson, 1996) normally limit their throughput. Spiffee's pipeline experiences a read-after-write data hazard once per *group*, which is once every 80 cycles. The hazard is handled by stalling the pipeline for one cycle to allow a previous write to complete before executing a read of the same word. This hazard also could have been handled by bypassing the cache and routing a copy of the result directly to pipeline stage two—negating the need to stall the pipeline—but this would necessitate the addition of another bus and another wide multiplexer into the datapath.

Cache⟷memory pipelines

As Eq. 5.4 shows, the cached-FFT algorithm significantly reduces the required movement of data to and from the main memory. The main memory arrays are accessed in two cycles in order to make the design of the main memory much easier and to reduce the power they consume. In the case of memory writes, only one cycle is required because it is unnecessary to precharge the bitlines or use the sense amplifiers.

Figure 5.9 shows the cache→memory pipeline diagram. A cache is read in the first stage, the data are driven onto the memory bus through a 2×2 crossbar in stage two, and data are written to main memory in stage three.

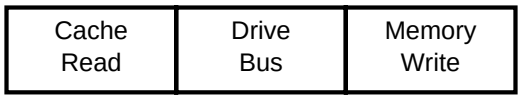


Figure 5.9: Cache→memory pipeline diagram

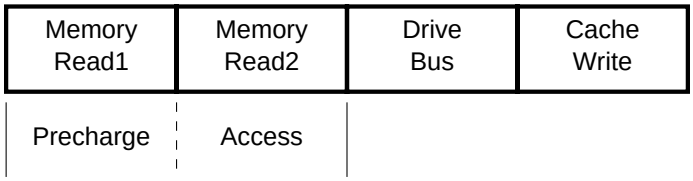


Figure 5.10: Memory→cache pipeline diagram

The memory→cache pipeline diagram is shown in Fig. 5.10. In the first stage, the selected memory array is activated, bitlines and sense amplifiers are precharged, and the array address is predecoded. In the second pipeline stage, the wordline is asserted; and the data are read, amplified, and latched. In stages three and four, data are driven onto the memory bus, go through a crossbar, and are written to the cache.

5.3.3 Datapath Design

In this section, "full" and "partial" blocks are discussed. By "full" we mean the block is "fully parallel," or able to calculate one result per clock cycle, even though the latency may be more than one cycle. By "partial" we mean the complement of full, implying that at least some part of the block is iterative, and thus, new results are not available every cycle.

A "block" can be a functional unit (*e.g.,* an adder or multiplier), or a larger computational unit such as a complete datapath (*e.g.,* an FFT butterfly). A *partial functional unit* contains iteration *within* the functional unit, and so data must flow through the same circuits several times before completion. A *partial datapath* contains iteration *at* the functional unit level, and so a single functional unit is used more than once for each calculation the computational unit performs.

A full non-iterative radix-2 datapath has approximately the right area and performance for a single-chip processor using 0.7 μm technology. Spiffee's datapath calculates one complex radix-2 DIT butterfly per cycle. This fully-parallel non-iterative butterfly processor has high hardware utilization—100% not including system-wide stalls.

Some alternatives considered for the datapath design include:

- A higher-radix "full" datapath—unfortunately, this is too large to fit onto a single die
- Higher-radix "partial" datapaths (*e.g.,* one multiplier, one adder,...)

- Higher-radix datapath with "partial" functional units (*e.g.,* radix-4 with multiple small iterative multipliers and adders)
- Multiple "partial" radix-2 butterfly datapaths—in this style, multiple self-contained units calculate butterflies without communicating with other butterfly units. Iteration can be performed either at the datapath level or at the functional unit level.

The primary reasons a "full" radix-2 datapath was chosen are because of its efficiency, ease of design, and because it does not require any local control, that is, control circuits other than the global controller.

5.3.4 Required Functional Units

The functional units required for the Spiffee FFT processor include:

Main memory — an N-word × 36-bit memory for data

Cache memories — 32-word × 40-bit caches

Multipliers — 20-bit × 20-bit multipliers

W_N **generation/storage** — coefficients generated or stored in memories

Adders/subtracters — 24-bit adders and subtracters

Controller — chip controller

Clock — clock generation and distribution circuits

5.3.5 Chip-Level Block Diagram

Once the required functional units are selected, they can be arranged into a block diagram showing the chip's floorplan, as shown in Fig. 5.11. Figure 6.1 on page 129 shows the corresponding die microphotograph of Spiffee1.

5.3.6 Fixed-Point Data Word Format

Spiffee uses a signed 2's-complement notation that varies in length from 18+18 bits to 24+24 bits. Table 5.1 gives more details of the format. Sign bits are indicated with an "`S`" and general data bits with an "`X.`"

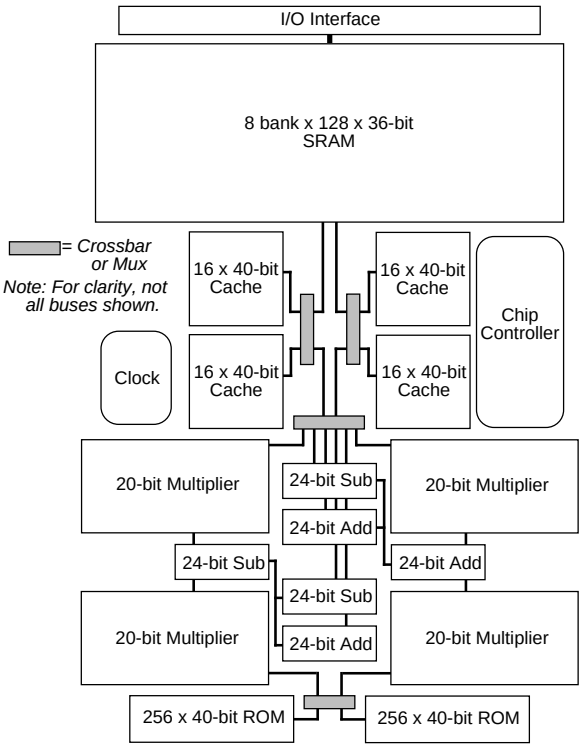


Figure 5.11: Chip block diagram

Format	Binary	Decimal
General format	`SXXXXXXXXXXXXXXXXXXX`	
Minimum value	`10000000000000000000`	-1.0
Maximum value	`01111111111111111111`	$+0.9999981$
Minimum step size	`00000000000000000001`	$+0.0000019$

Table 5.1: Spiffee's 20-bit, 2's-complement, fixed-point, binary format

5.4 Physical Design

This section provides some details of the main blocks listed in the previous section. Circuit schematics presented adhere to the following rules-of-thumb to increase readability: two wires which come together in a "⊤" are electrically connected, but wires which cross ("+") are unconnected, unless a "•" is drawn at the intersection. Also, several die microphotographs of Spiffee1 are shown.

5.4.1 Main Memory

This section discusses various design decisions and details of the 1024-word main memory, including an introduction of the *hierarchical bitline* memory architecture, which is especially well suited for low-V_{dd}, low-V_t operation.

SRAM vs. DRAM

The two primary types of RAM are static RAM (SRAM) and dynamic RAM (DRAM). DRAM is roughly four times denser than SRAM but requires occasional refreshing, due to charge leakage from cell storage capacitors. For a typical CMOS process, refresh times are on the order of a millisecond. This could have worked well for an FFT processor since data are processed on the order of a tenth of a millisecond—so periodic refreshing due to leakage would not have been necessary. However, DRAMs also need refreshing after data are read since memory contents normally are destroyed during a read operation. But here again, use in an FFT processor could have worked well because initial and intermediate data values are read and used only once by a butterfly and then written over by the butterfly's outputs.

So while the use of DRAM for the main memory of a low-power FFT processor initially appears attractive, DRAM was not used for Spiffee's main memory because in a low-V_t process, cell leakage (I_{off}) is orders of magnitude greater than in a standard CMOS process—shortening refresh times inverse-proportionately. Refresh times for low-V_t DRAM could easily be on the order of a fraction of a microsecond, making their use difficult and unreliable. The other significant reason is that use of DRAM would complicate testing and analysis of the chip, as well as making potential problems in the memory itself much more difficult to debug.

Circuit design

Although it is possible to build many different styles of SRAM cells, common six-transistor or *6T* cells (Weste and Eshraghian, 1985) operate robustly even with low-V_t transistors. Four-transistor cells require a non-standard fabrication process, and cells with more than six transistors were not considered because of their larger size.

Since memories contain so many transistors, it is difficult to implement a large low-power memory using a low-V_t process because of large leakage currents. Design becomes even more difficult if a "standby" mode is required—where activity halts and leakage currents must typically be reduced.

Although several lower-V_{dd}, lower-V_t SRAM designs have been published, nearly all make use of on-chip charge pumps to provide voltages larger than the supply voltage. These larger voltages are used to reduce leakage currents and decrease memory access times through a variety of circuit techniques. Also, these chips typically use a process with multiple V_t values. Yamauchi *et al.* (1996; 1997) present an SRAM design for operation at 100 MHz and a supply voltage of 0.5 V using a technology with 0.5 V and 0 V thresholds. The design requires two additional voltage supplies at 1.3 V and 0.8 V. The current drawn from these two additional supplies is low and so the voltages can be generated easily by on-chip charge pumps. Yamauchi *et al.* present a comparison of their design with two other low-power SRAM approaches. They make one comparison with an approach proposed by Mizuno *et al.* (1996), who propose using standard V_t transistors, $V_{dd} < 1.0$ V, and a negative source-line voltage in the vicinity of -0.5 V. Yamauchi *et al.* state the source-line potential must be -0.75 V with $V_{dd} = 0.5$ V, and note that the charge pump would be large and inefficient at those voltage and current levels. They make a second comparison with a design by Itoh *et al.* (1996). Itoh *et al.* propose an approach using two boosted supplies which realizes a $V_{dd} = 300$ mV, 50 MHz, SRAM implemented in a 0.25 μm, $V_{t\text{-}nmos} = 0.6$ V, $V_{t\text{-}pmos} = -0.3$ V technology. At a supply voltage of $V_{dd} = 500$ mV, these additional boosted voltages are in the range of 1.4 V and -0.9 V. Amrutur and Horowitz (1998) report a 0.35 μm, low-V_{dd} SRAM which operates at 150 MHz at $V_{dd} = 1.0$ V, 7.2 MHz at $V_{dd} = 0.5$ V, and 0.98 MHz at $V_{dd} = 0.4$ V.

To avoid the complexity and inefficiency of charge-pumps, Spiffee's memory is truly low-V_{dd} and operates from a single power supply. In retrospect, however, it appears likely that generating local higher supply voltages would have resulted in a more overall energy-efficient

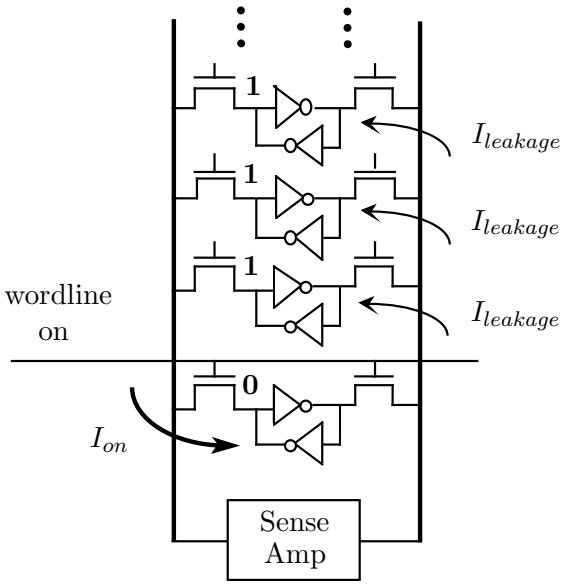


Figure 5.12: Circuit showing SRAM bitline fan-in leakage

design. In any case, constructing additional supplies involves a design-time/efficiency trade-off that clearly requires substantially more design effort.

Bitline leakage

Memory reads performed under low-V_{dd}, low-V_t conditions can fail because of current leakage on the high fan-in bitlines. Figure 5.12 illustrates the problem. The worst case for a memory with L rows occurs when all L cells in a column—except one—store one value, and one cell (the bottom one in this schematic) stores the opposite value. For 6T SRAMs, the value of a cell is sensed by the cell *sinking* current into the side storing a "0"; very little current flows out of the side storing a "1." To first order, if leakage through the $L-1$ access transistors on one bitline ($(L-1) \cdot I_{leakage}$) is larger than the I_{on} current of the accessed cell on the other bitline, a read failure will result.

Figure 5.13 shows a spice simulation of this scenario with a 128-word SRAM operating at a supply voltage of 300 mV. In this simulation, the access transistors of 127 cells are leaking, pulling *bitline_* toward *Gnd*. The cell being read is pulling *bitline* toward *Gnd*. The simulation begins with a precharge phase (6 ns $<$ *time* $<$ 13 ns) which boosts both *bitline* and *bitline_* toward V_{dd}. The leakage on *bitline_* causes it to immediately begin

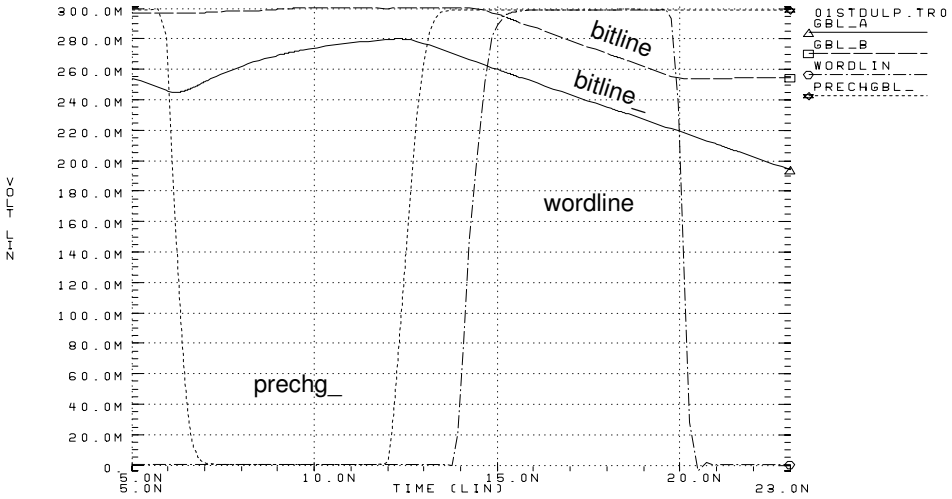


Figure 5.13: Spice simulation of a read failure with a 128-word, low-V_t, non-hierarchical-bitline SRAM operating at $V_{dd} = 300\,\text{mV}$

sloping downward as *prechg_* rises and turns off. As the wordline rises ($time \approx 14\,\text{ns}$), the selected RAM cell pulls *bitline* toward *Gnd*. *Bitline* holds a constant voltage after *wordline* drops ($time > 20\,\text{ns}$). Leakage continues to drop the voltage of *bitline_* regardless of the wordline's state. Since the "0" on *bitline* is never lower than the "1" on *bitline_*, an incorrect value is read.

The most common solution to this problem is to reduce the leakage onto the bitlines using one of two methods. The first approach is to overdrive the access transistor gates by driving the wordline below *Gnd* when the cell is unselected (for NMOS access transistors). Another method is to reduce the V_{ds} of the access transistors by allowing the voltage of the access transistors' sources to rise near the average bitline potential, and then pull the sources low when the row is accessed, in order to increase cell drive current.

Hierarchical bitlines

Our solution to the bitline-leakage problem is through the use of a *hierarchical-bitline* architecture, as shown in Fig. 5.14. In this scheme, a column of cells is partitioned into segments by cutting the bitlines at uniform spacings. These short bitlines are called *local bitlines* (*lbl* and *lbl_* in the figure). A second pair of bitlines called *global bitlines* (*gbl* and *gbl_* in the figure) is run over the entire column and connected to the local bitlines through connecting

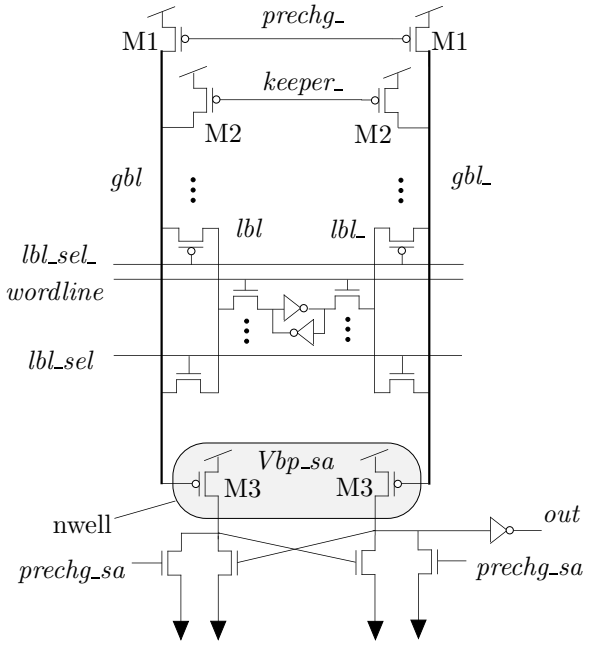


Figure 5.14: Schematic of a hierarchical-bitline SRAM

transistors. Accesses are performed by activating the correct wordline as in a standard approach, as well as connecting its corresponding local bitline to the global bitline.

If an L-word memory is partitioned into S segments, the worst-case bitline leakage will be on the order of, but less than, $(L/S-1)I_{leakage}$ from the leakage of cells in the connected local bitline, plus $2(S-1)I_{leakage}$ from leakage through the $S-1$ unconnected local bitlines. For non-degenerate cases (*e.g.,* $S = 1, S \approx L, L \approx 1$), this leakage is clearly less than when using an approach with a single pair of bitlines, which leaks $(L-1) \cdot I_{leakage}$. Values of S near $\sqrt{L}$ work best because a memory with $S = \sqrt{L}$ has S memory cells attached to each local bitline and S local bitlines connected to each global bitline. Under those circumstances, access transistors contribute the same amount of capacitance to both local and global bitlines, and the overall capacitance is minimized, to first order.

Figure 5.15 shows a spice simulation of the same memory whose simulation results are shown in Fig 5.13—but with hierarchical bitlines. The same *precharge_* and *wordline* circuits are used as in the previous case. Here, the 127 cells are leaking onto the *gbl_* bitline (most, however, are leaking onto disconnected local bitlines), and the accessed cell is pulling down

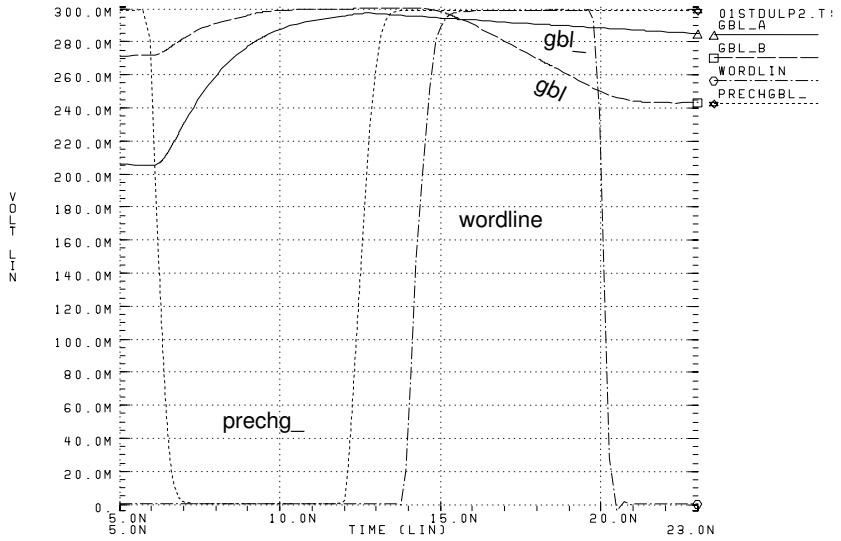


Figure 5.15: Spice simulation of a successful read with a 128-word, low-V_t, hierarchical-bitline SRAM operating at $V_{dd} = 300\,\text{mV}$

gbl. The downward slope of *gbl_* from leakage begins at the end of the precharge stage, but is clearly less than in the non-hierarchical case. For these two simulations, *gbl_* of the hierarchical-bitline memory leaks approximately 1 mV/ns while *bitline_* of the standard memory leaks approximately 8 mV/ns. When the wordline accesses the cell, *gbl* drops well below *gbl_* and the sense amplifier reads the correct value.

The primary drawbacks of the hierarchical-bitline approach are the extra complexity involved in decoding and controlling the *local bitline select* signals; and the area penalty associated with local-bitline-select circuits, bitline-connection circuits, and two pairs of bitlines. Using MOSIS design rules, however, the cell width is not limited by the *lbl*, *lbl_*, *gbl*, and *gbl_* pitch (which are laid out using the metal2 layer), so the use of hierarchical bitlines does not increase the cell size. The series resistance of bitline-connection transistors slows access times, although a reduction of overall bitline capacitance by nearly 50% provides a significant speedup in the wordline→bitline access time. Longer precharge times may degrade performance further, although it is easy to imagine schemes where the precharge time could be shortened considerably (such as by putting precharge transistors on each local bitline).

To control bitline leakage—which is especially important while the clock is run at very low frequencies, such as during testing—a programmable control signal enables weak *keeper*

transistors (labeled M2 in Fig. 5.14). Keepers are commonly used in dynamic circuits to retain voltage levels in the presence of noise, leakage, or other factors which may corrupt data. For Spiffee, two independently-controllable parallel transistors are used, with one weaker than the other.

Sense amplifiers

Figure 5.14 shows the design of the sense amplifiers. They are similar to ones proposed by Bakoglu (1990, page 153), but are fully static. They possess the robust characteristic of being able to correct their outputs if noise (or some other unexpected source) causes the sense amplifier to begin latching the wrong value. Popularly-used sense amplifiers with back-to-back inverters do not have this property. Of course, a substantial performance penalty is incurred if the clock is slowed to allow the sense amplifiers to correct their states—but the memory would at least function under those circumstances. Because the bitlines swing over a larger voltage range than typical sense amplifier input-offset voltages (Itoh *et al.*, 1995), the sense amplifiers operate correctly at low supply voltages.

The gates of a pair of PMOS transistors serve as inputs to the sense amplifiers (M3 in Fig. 5.14). Because the chip is targeted for n-well or triple-well processes, the bodies of the PMOS devices can be isolated in their own n-well. By biasing this n-well differently from other n-wells, robust operation is maintained while improving the bitline→sense-amplifier-out time by up to 45%. For this reason, the n-well for the M3 PMOS is biased separately from other n-wells.

Phases of operation

The operation of Spiffee's memory arrays can be divided into three phases. Some circuits are used only for reads, some only for writes, and some for both. Memory writes are simpler than reads since they do not use precharge or sense amplifier circuits.

Precharge phase — During the precharge phase, the appropriate local bitlines are connected to the global bitlines. Transistors M1 charge *gbl* and *gbl_* to V_{dd} when *prechg_* goes low. Wordline-selecting address bits are decoded and prepared to assert the correct wordline. When performing a read operation, internal nodes of the sense amplifiers are precharged to *Gnd* by the signal *prechg_sa* going high.

Access phase — In the access phase, a wordline's voltage is raised, which connects a row of memory cells to their local bitlines. For read operations, the side of the memory cell which contains a "0" reduces the voltage of the corresponding local bitline, and thus, the corresponding global bitline as well. For write operations, large drivers assert data onto the global bitlines. Write operations are primarily performed by the bitline which is driven to *Gnd*, since the value stored in a cell is changed by the side of the cell that is driven to *Gnd*. Write operations which do not change the stored value are much less interesting cases since the outcome is the same whether or not the operation completes successfully.

Sense phase — For read operations, when the selected cells have reduced the voltage of one global bitline in each global bitline pair to $V_{dd} - V_{t\text{-}pmos}$, the corresponding M3 transistor begins to conduct and raise the voltage of its drain. When that internal node rises $V_{t\text{-}nmos}$ above *Gnd*, the NMOS transistor whose gate is tied to that node clamps (holds) the internal node on the other side of the sense amplifier to *Gnd*. As the rising internal node crosses the switching threshold of the static inverter, the signal *out* toggles and the read is completed. The output data are then sent to the bus interface circuitry, which drives the data onto the memory bus.

Memory array specifics

To increase the efficiency of memory transactions, Spiffee's memory arrays have the same 36-bit width as memory data words. Previous memory designs commonly use *column decoders* which select several bits out of an accessed word, since the memory array stores words several times wider than the data-word width. Unselected bits are unused, which wastes energy.

When determining the number of words in a memory array, it is important to consider the area overhead of circuits which are constant per array—such as sense amplifiers and bus interface circuits. For example, whether a memory array contains 64, 128, or 256 words makes no difference in the absolute area of the sense amplifiers and bus interface circuits for the array. Spiffee's memory arrays contain 128 words because that size gives good area efficiency, and because arrays with 128-words fit well on the die.

Spiffee's main memory comprises eight 128-word by 36-bit SRAM arrays. Seven address bits are used to address the 128 words. Each column has eight segments or local bitlines

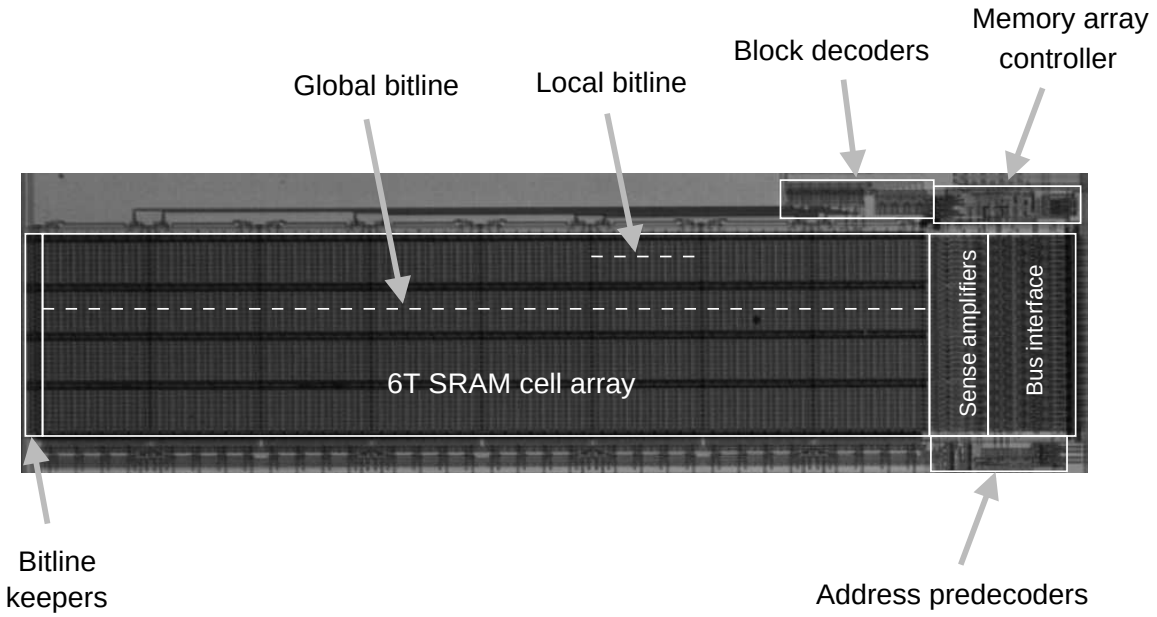


Figure 5.16: Microphotograph of a 128-word × 36-bit SRAM array

($S = 8$). Figure 5.16 shows a microphotograph of an SRAM array. The eight local bitlines are clearly seen and dashed lines indicate the lengths of local and global bitlines. *Block decoders* decode the upper three bits of the seven-bit address to select the correct local bitline. The *memory array controller* generates all timing and control signals for the array. *Bus interface* circuitry buffers data to and from the memory bus. *Address predecoders* partially decode the four least-significant address bits and drive this information along the bottom side of the array to the wordline drivers. Programmable *bitline keepers* control bitline leakage and are highlighted on the far left side of the array. For Spiffee, the area penalty of the hierarchical-bitline architecture is approximately 6%.

Figure 5.17 shows a closeup view of the region around one row of bitline-connection transistors. Both global and local bitlines are routed in the second layer of metal. A box around a pair of SRAM cells indicates the area they occupy.

5.4.2 Caches

Single-ported vs. dual-ported

To sustain the throughput of the pipeline diagram shown in Fig. 5.8, cache reads of A and B and cache writes of X and Y must be performed every cycle. Assuming two independent

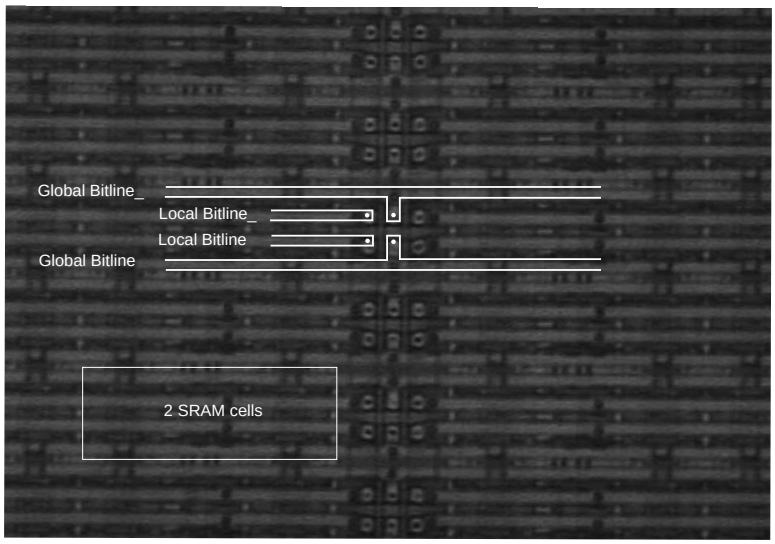


Figure 5.17: Microphotograph of an SRAM cell array

cache arrays are used, one read and one write must be made each cycle. Multiple memory transactions per cycle are normally accomplished in one of two ways. One approach is to perform one access (normally the write) during the first half of the cycle, and then the other access (the read) during the second half of the cycle. A second method is to construct the memory with two ports that can simultaneously access the memory.

The main advantage of the single-ported, split-cycle approach is that the memory is usually smaller since it has a single read/write port. Making two sequential accesses per cycle may not require much additional effort if the memory is not too large.

The dual-ported approach, on the other hand, results in larger cells that require both read and write capability—which implies dual wordlines per cell and essentially duplicate read and write circuits. The key advantage of the dual-ported approach is that twice as much time is available to complete the accesses. Fewer special timing signals are required, and circuits are generally simpler and more robust. For these reasons, Spiffee's caches use the dual-ported approach with one read-only port and one write-only port. A simultaneous read and write of the memory should normally be avoided. Depending on the circuits used, the read may return either the old value, the new value, or an indeterminate value. When used as an FFT cache, the simultaneous read/write situation is avoided by properly designing the cache access patterns.

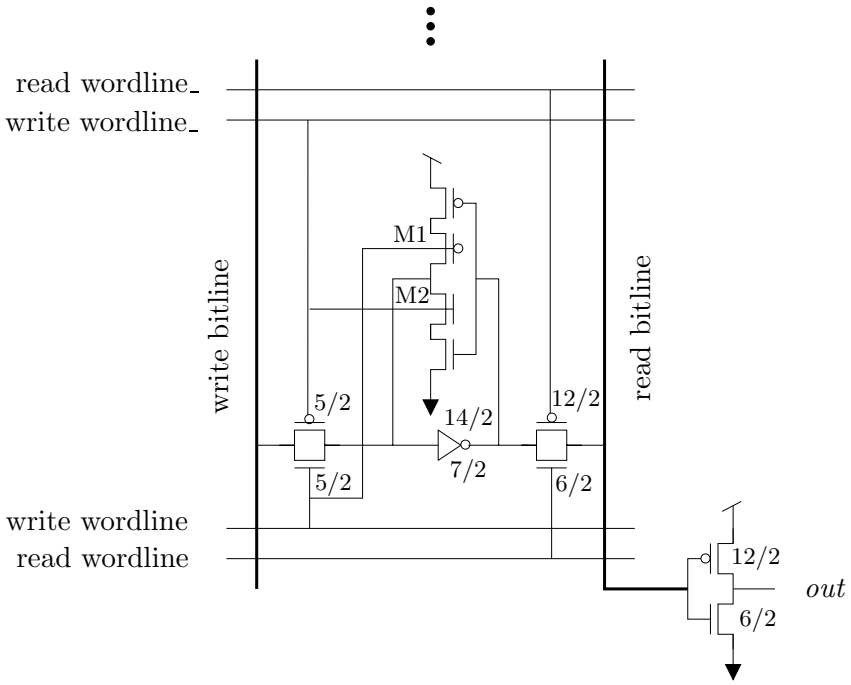


Figure 5.18: Simplified schematic of a dual-ported cache memory array using 10-transistor-cells. Some transistor dimensions are shown as: *width*(λ)/*length*(λ).

Circuits

Spiffee's caches use two independent, single-ended bitlines for read and write operations. Four wordlines consisting of a read pair and a write pair, control the cell. The memory is fully static and thus does not require a precharge cycle.

The memory's cells each contain ten transistors. Figure 5.18 shows a simplified schematic of a cell in an array. Transistor widths and lengths are indicated for some transistors in units of λ as the ratio: *width/length.*[3] Full transmission gates connect cells to bitlines to provide robust low-V_{dd} operation. Transistors M1 and M2 disable the feedback "inverter" during writes to provide reliable operation at low supply voltages. The read-path transmission gate and inverter are sized to quickly charge or discharge the *read bitline*. Both write and read bitlines swing fully from *Gnd* to V_{dd}, so only a simple inverter buffers the output before continuing on to the cache bus interface circuitry.

[3]The variable λ is a unit of measure commonly used in chip designs which allows designs to scale to different technologies.

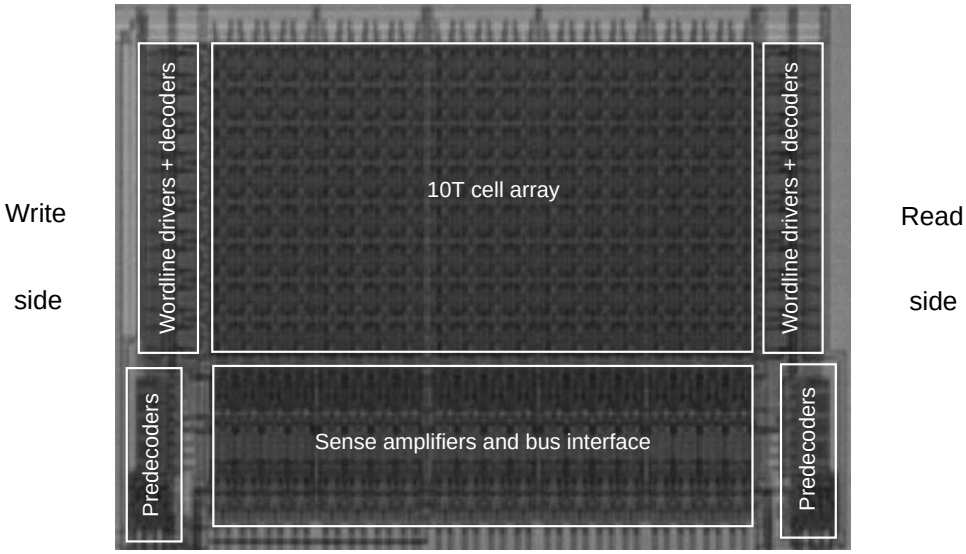


Figure 5.19: Microphotograph of a 16-word × 40-bit cache array

Each cache memory array contains 16-words of 40-bit data. Figure 5.19 shows a die microphotograph of one cache array. The left edge of the array contains the address predecoders, decoders, and wordline drivers necessary to perform writes to the cell array—which occupies the center of the memory. The right edge contains nearly identical circuitry to perform reads. The bottom center part of the memory contains circuits which read and write the data to/from the array, and interface to the cache data bus.

5.4.3 Multipliers

As mentioned previously, the butterfly datapath requires 20-bit by 20-bit 2's-complement signed[4] multipliers. To enhance performance, the multipliers are pipelined and use full, non-iterative arrays. This section begins with an overview of multiplier design and then describes the multipliers designed for Spiffee.

Multiplier basic principles

Figure 5.20 shows a block diagram of a generic hardware multiplier. The two inputs are the *multiplicand* and the *multiplier*, and the output is the *product*. The method for hardware

[4]Signed multipliers are "four-quadrant" arithmetic units whose operands and product can be either positive or negative.

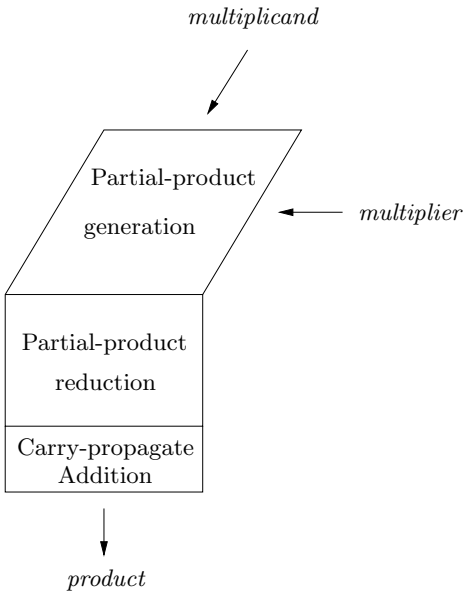


Figure 5.20: Generic multiplier block diagram

multiplication is similar to the one commonly used to perform multiplication using paper and pencil.

In the first stage, the multiplier generates a diagonal table of partial products. Using the simplest technique, the number of digits in each row is equal to the number of digits in the *multiplicand*, and the number of rows is equal to the number of digits in the *multiplier*.

The second and third stages add the partial products together until the final sum, or product, has been calculated. The second stage adds groups of partial product bits (often in parallel) using various styles of adders or *compressors*. At the end of this stage, the many rows of partial products are reduced to two rows. In the final stage, a *carry-propagate adder* adds the two rows, resulting in the final *product*. The carry-propagate addition is placed in a separate stage because an adder which propagates carries through an entire row is very different from an adder which does not.

Multiplier encoding

A distinguishing feature of multipliers is how they encode the *multiplier*. In the generic example discussed above, the *multiplier* is not encoded and the maximum number of partial products are generated. By using various forms of *Booth encoding*, the number of rows is

reduced at the expense of some added complexity in the *multiplier* encoding circuitry and in the partial product generation array. Because the size of the array has a strong effect on delay and power, a reduction in the array size is very desirable.

When using Booth encoding, partial products are generated by selecting from one of several quantities; typically: 0, $\pm$*multiplicand*, ± 2*multiplicand*, ... A *multiplexer* or *mux* commonly makes the selection, which is why the partial product array is also commonly called a *Booth mux array*.

A common classification of Booth algorithms uses the Booth-n notation, where n is an integer and is typically 2, 3, or 4. A Booth-n algorithm normally examines $n+1$ *multiplier* bits, encodes n *multiplier* bits, and achieves approximately an $n\times$ reduction in the number of partial product rows.

We will not review the details of Booth encoding. Bewick (1994) gives an overview of different Booth encoding algorithms and provides useful information for designing a multiplier. Waser and Flynn (1990) also provide a good, albeit brief, introduction to Booth encoding.

Spiffee's multipliers use Booth-2 encoding because this approach reduces the number of *multiplier* rows by 50%, and does not require the complex circuits used by higher-order Booth algorithms. Using Booth-2, the partial product array in the 20-bit by 20-bit multiplier contains ten rows instead of twenty.

Dot diagrams

Dot diagrams are useful for showing the design of a multiplier's array. The diagrams show how partial products are laid out and how they compress through successive adder stages. For the dot diagrams shown here, the following symbols are used with the indicated meanings:

```
. = input_bit, can be either a 0 or 1
, = NOT(.)
0 = always zero
1 = always one
S = the partial product sign bit
E = bit to clear out sign_extension bits
e = NOT(E)
- = carry_out bit from (4,2) or (3,2) adder in adjacent column to the right
x = throw this bit away
```

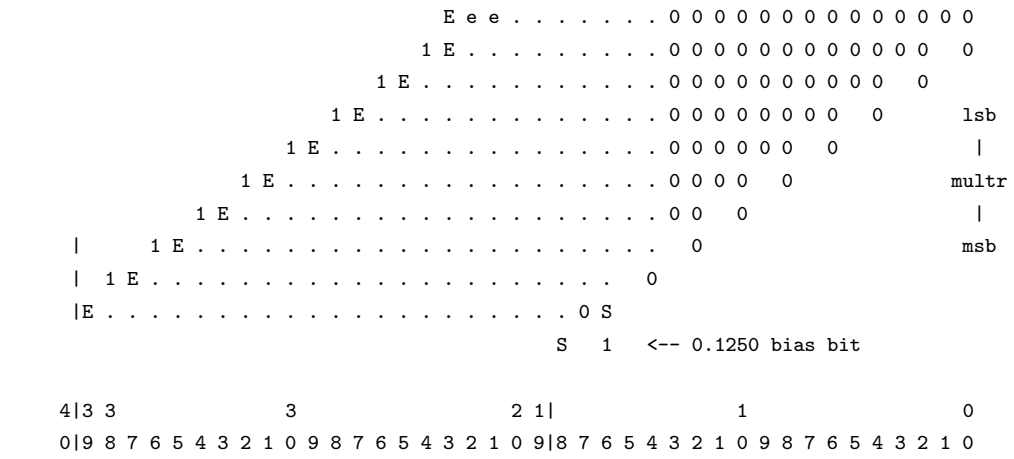
Bewick (1994) provides further details for some of the symbols.

Multiplier pipeline stage "0"

In the last part of the clock cycle prior to partial product generation, the bits of the *multiplier* are Booth-2 encoded and latched before they are driven across the Booth mux array in "stage 1."

Multiplier pipeline stage 1

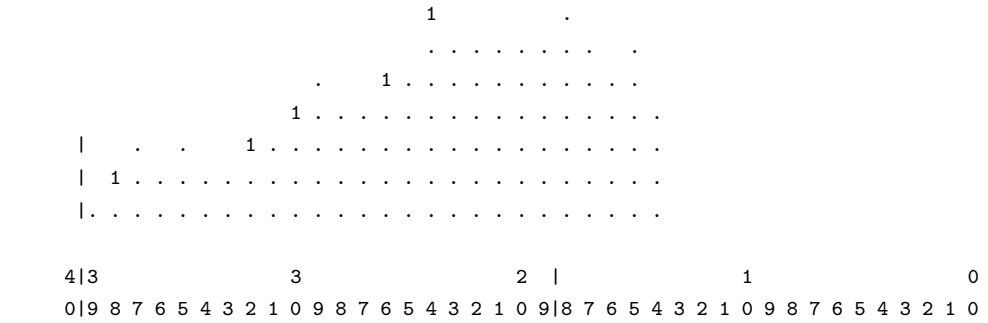
In the first of the three main pipeline stages, Booth muxes generate partial products in the partial product array. Since the fixed-point data format requires the truncation of the butterfly's outputs to 20 bits for both real and imaginary components, when writing back to the cache, it is unnecessary to calculate all 40 bits of the 20-bit by 20-bit multiplication. To save area and power, 63 bits in the tree are not calculated. These removed bits reduce the full tree by 27%, and are indicated by 0s in columns 0–13 of the first dot diagram:



A bias bit in column 16 sets the mean of the truncation error to zero. The 20 bits in columns 20–39 are the significant bits which are saved at the end of the butterfly calculation. Bits 14–19 are kept as guard bits and are discarded as the butterfly calculation proceeds.

To compress the rows of partial products before latching them at the end of the first pipeline stage, there are two rows of (3,2) or *full adders* (Sec. 5.4.5 further describes full adders), and one row of (4,2) adders. Santoro (1989) provides an excellent introduction

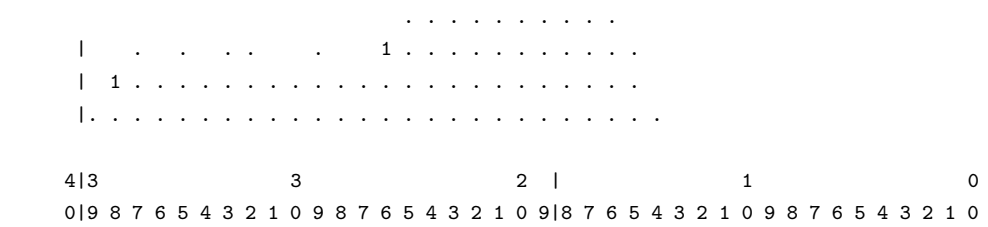
to the use of (3,2) and (4,2) adders in multiplier arrays. At the end of this stage, the dot diagram of the remaining bits is:



A dense row of flip-flops latches these remaining bits.

Multiplier pipeline stage 2

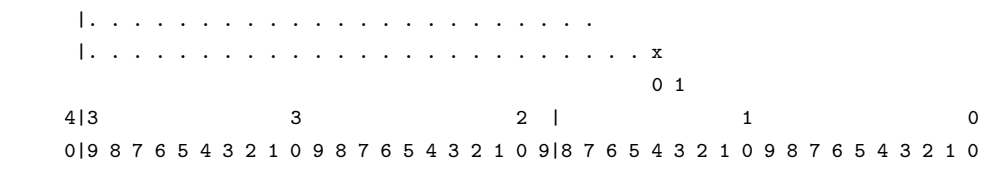
In the second multiplier pipeline stage, one row each of (3,2) and (4,2) adders sum the bits. The output of these rows is then:



The four rows of bits are then latched, which completes the second pipeline stage.

Multiplier pipeline stage 3

In the third pipeline stage, only one row of (4,2) adders is required to reduce the partial products to two rows. The output of those adders is:



Although output bits from the (4,2) adders could be sent to a carry-propagate adder to complete the multiplication, some operations related to the *complex* multiplication are

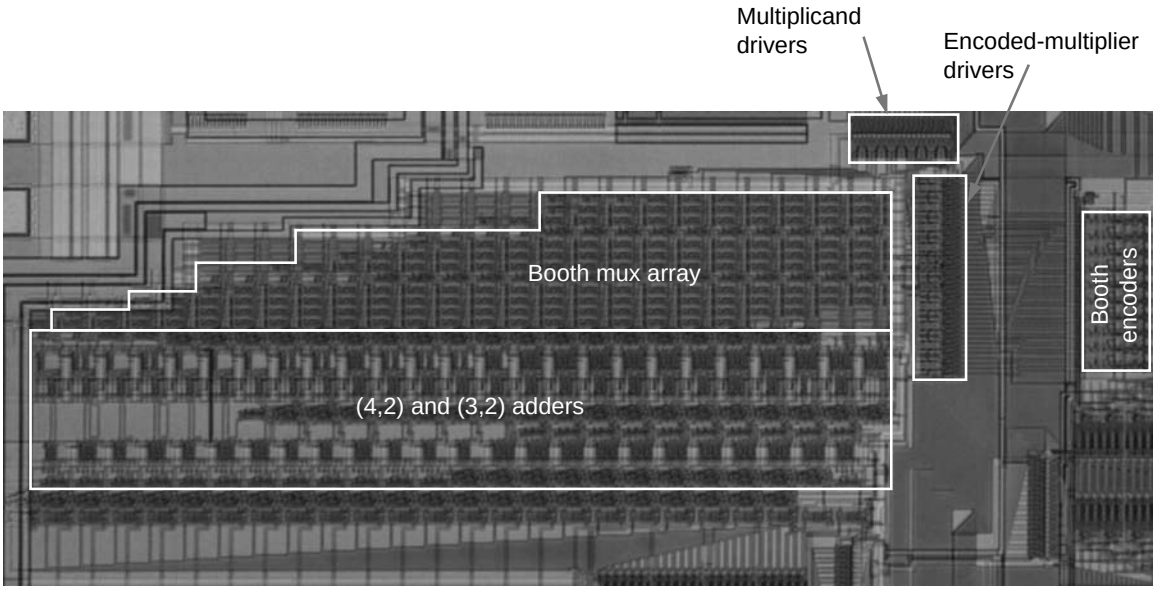


Figure 5.21: Microphotograph of a 20-bit multiplier

performed during the remainder of this cycle. After these additional operations, the two rows of bits are latched.

Carry-propagate addition

In the final pipeline stage of the multiplier, a carry-propagate adder reduces the product to a single row. The adder used is the same one used in other parts of the processor and is described in Sec. 5.4.5.

Multiplier photos

Figure 5.21 contains a microphotograph of the multiplier just described. The Booth encoders are placed away from the central multiplier because it results in a more compact layout overall. The Booth-encoded *multiplier* drivers and the *multiplicand* drivers reside in the upper-right corner of the multiplier. The Booth mux array has a slight parallelogram shape—like that shown in the first dot diagram.

Gaps in the partial product array are caused by the use of a near minimum number

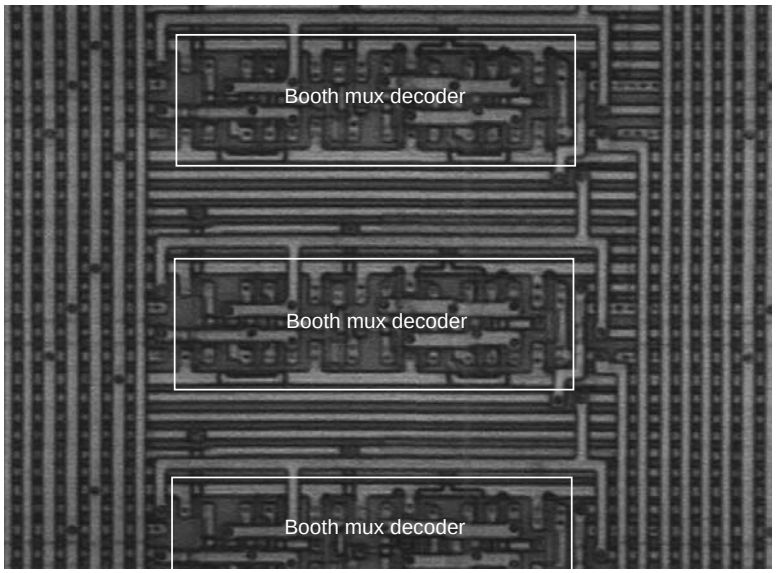


Figure 5.22: Microphotograph of Booth decoders in the partial product generation array

of (3,2) adders, (4,2) adders, and latches (for power reasons). A later version of the design (which has not yet been fabricated), places power supply bypass capacitors in these openings.

The Booth mux array is a structure filled with dense wiring channels. Figure 5.22 shows a high-magnification view of three Booth mux decoders in the array. The high-current nodes V_{dd} and *Gnd* run both vertically and horizontally, and $V_{n\text{-}well}$ and $V_{p\text{-}well}$ run vertically. Because data flow downward in the multiplier, the most important direction to route V_{dd} and *Gnd* wires is the vertical direction. In general, supply wires parallel to the direction of data flow are far more effective in maintaining V_{dd} and *Gnd* potentials than perpendicularly-routed supply wires. *Multiplicand* bits run diagonally across the Booth mux array, and encoded *multiplier* bits run horizontally.

Figure 5.23 is a picture of a (4,2) adder, and shows the four inputs, *in1–in4*, as well as the two outputs, *out-sum* and *out-carry*. The other input, C_{in}, and the other output, C_{out}, are not indicated because they are not easily visible. Wires for $V_{n\text{-}well}$ and $V_{p\text{-}well}$ are labeled and show the small additional area required by connecting and routing these nodes independently from V_{dd} and *Gnd*. In the figure, the higher-level metal2 layer is slightly out of focus and the lower metal1 layer is more sharply in focus.

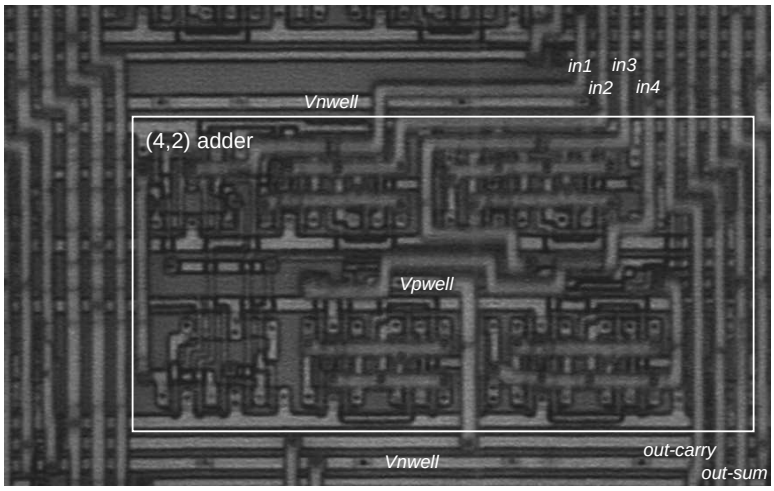


Figure 5.23: Microphotograph of a (4,2) adder

5.4.4 W_N Coefficient Storage

The calculation of an N-point radix-2 FFT requires $N/2$ complex W_N coefficients. Various techniques requiring additional computation can reduce this number. However, for the processor described here, where simplicity and regularity are a high priority, an on-chip Read Only Memory (ROM) stores all $1024/2 = 512$ coefficients. Because ROMs are among the densest of all VLSI circuits, a ROM of this size does not present a large area cost.

As a type of memory, ROMs experience the same bitline leakage problems when operating in a low-V_{dd}, low-V_t environment as the SRAM described in Sec. 5.4.1. To maintain functionality under these conditions, the ROMs also employ the *hierarchical-bitline* technique.

Figure 5.24 shows a schematic of Spiffee's ROM showing its hierarchical-bitline structure where individual memory cells connect to *local bitlines* which then connect to *global bitlines* through NMOS and PMOS connecting transistors. ROM memory cells are vastly simpler than 6T SRAM memory cells as they contain only one or zero transistors. If a transistor is present, its source connects to *Gnd*, to pull the bitline low when the cell is accessed. The opposite data value could be obtained by connecting the transistor's source to V_{dd}, but that would increase the cell area and would be only mildly effective because the transistor's body effect would drop the bitline's maximum voltage to $V_{dd} - V_t$. Instead, bitlines are precharged to V_{dd}, and omitting the cell transistor altogether generates the opposite data value. Pulsing the signal *prechg_* low precharges the global bitline. To control bitline leakage and to enable low-frequency operation, the signal *keeper_* activates the weak M2 PMOS keeper transistors.

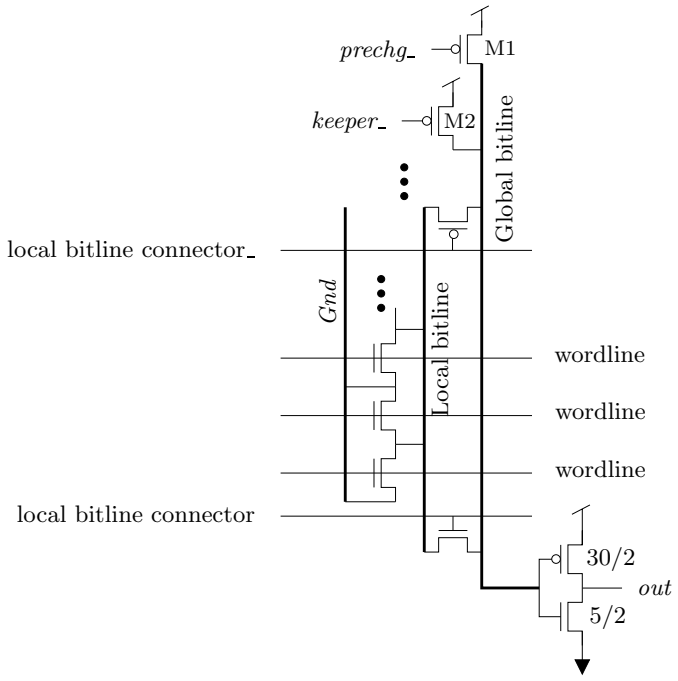


Figure 5.24: Schematic of a ROM with hierarchical bitlines

To obtain maximum robustness and simplicity, bitlines swing fully from V_{dd} to *Gnd*—at the expense of some energy-efficiency. The low impedance to *Gnd* of cell transistors allows fast access times with minimum-sized array transistors. Full-swing bitlines greatly simplify the sense amplifier design. A simple CMOS inverter with the PMOS device sized six times larger than the NMOS device performs well in terms of speed and noise-margin. This circuit also has the robust feature of being highly resistant to transient noise because the circuit always settles to the correct value if a noise spike causes it to begin sensing incorrectly.

ROM arrays

Spiffee's 512-word ROM consists of two 256-word by 40-bit arrays. Each array contains sixteen blocks, corresponding to a maximum of sixteen cells per local bitline (since only rarely are all sixteen bits in a block equal to the value requiring a transistor). Figure 5.25 shows a die microphotograph of a ROM array. Representative global and local bitlines are indicated, and the sixteen blocks are clearly visible. *Block select* predecoders and drivers use four bits from the ROM's address to select and activate the correct block. Address

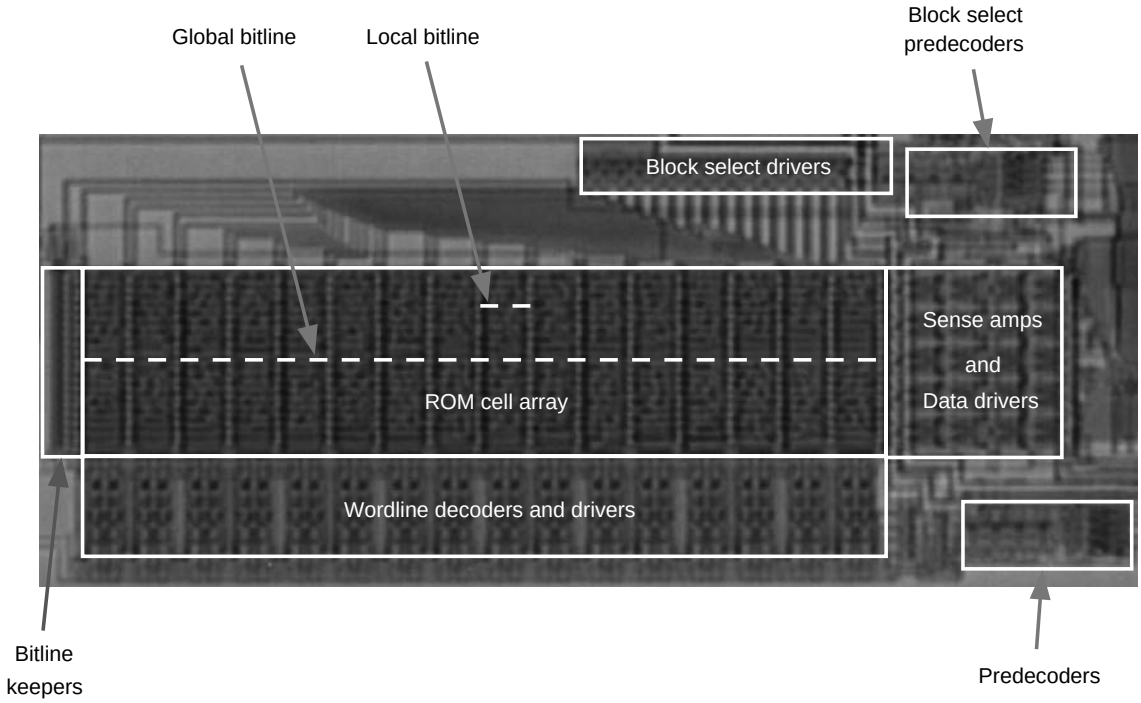


Figure 5.25: Microphotograph of a ROM array

predecoders and wordline decoding circuitry assert the selected wordline using the remaining four bits. *Bitline keepers* reside at one end of the cell array, with the sense amplifiers and bus drivers at the other end.

The ROM cell array uses the minimum metal pitch allowed by the technology's design rules. The dense cell pitch made the layout of the wordline decoders and drivers difficult as their pitch must match the pitch of the cell array. In retrospect, it would have been better to relax the horizontal pitch (in Fig. 5.25) of the cell array even though the total area would have been increased. Although the sense amplifier pitch was not as difficult to match as it was for the wordline drivers, the vertical pitch (in Fig. 5.25) of the cell array should probably have been relaxed somewhat as well.

5.4.5 Adders/Subtracters

The calculation of a complex radix-2 butterfly requires three additions and three subtractions. Since subtracters and adders have nearly identical hardware, this section will focus

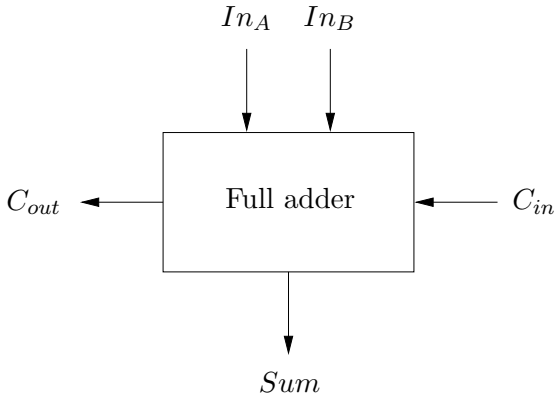


Figure 5.26: Full-adder block diagram

on adder design, as all principles apply directly to subtracters as well. To increase precision through the datapath, adders and subtracters are 24 bits wide.

Adders are often based on a simple building block called a *full adder*, which sums three inputs: In_A, In_B, and C_{in}; resulting in two output bits, Sum and C_{out}. Figure 5.26 is a block diagram of a full adder and Table 5.2 contains a truth table showing the output values for all eight possible input combinations. The three inputs as well as the output Sum all have equal weightings of "1." The output C_{out} has a weighting that is twice as significant, or "2."

A distinguishing feature of adders and subtracters is the method they use to propagate carry bits, or carries. As with most VLSI structures, there is a trade-off between complexity and speed. Waser and Flynn (1990) give a brief overview of different adder types. The simplest, albeit slowest, approach is the *ripple carry* technique, in which carries propagate sequentially through each bit of the adder. An n-bit ripple-carry adder can be built by placing n of the full adders shown in Fig. 5.26 next to each other, and connecting adjacent C_{out} and C_{in} signals.

A faster and more common type of adder is the *carry-lookahead,* or CLA adder. In this approach, circuits develop *propagate* and *generate* signals for blocks of adders. The number of levels of carry-lookahead depends on the number of adder blocks that feed into a CLA block. For a regular CLA implementation, if r adder blocks feed into a CLA block,

Inputs			Outputs	
C_{in}	In_A	In_B	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 5.2: Truth table for a full adder

the number of levels of carry-lookahead logic for an n-bit adder will be (Waser and Flynn, 1990),

$$levels = \lceil \log_r n \rceil . \tag{5.7}$$

Because Spiffee does not require the speed of a standard CLA adder, its adders use a combination of CLA and ripple-carry techniques. At the lowest level, carries ripple across three-bit blocks. The ripple-carry circuits reduce the complexity of the adder and thereby reduce energy consumption. Across higher levels of adder blocks, CLA accelerates carry propagation with $r = 2$ through three levels. The number of CLA levels is,

$$\begin{aligned} levels &= \log_r(n/ripple\_length) && (5.8)\\ &= \log_2(24/3) && (5.9)\\ &= 3. && (5.10) \end{aligned}$$

Figure 5.27 shows a block diagram of the adder.

Figure 5.28 is a microphotograph of a fabricated adder. The two 24-bit operand buses enter the top of the adder and the 25-bit sum exits the bottom. The dense row of 24 full adders can be seen in the top half of the structure and the more sparsely-packed carry-lookahead section occupies the lower half.

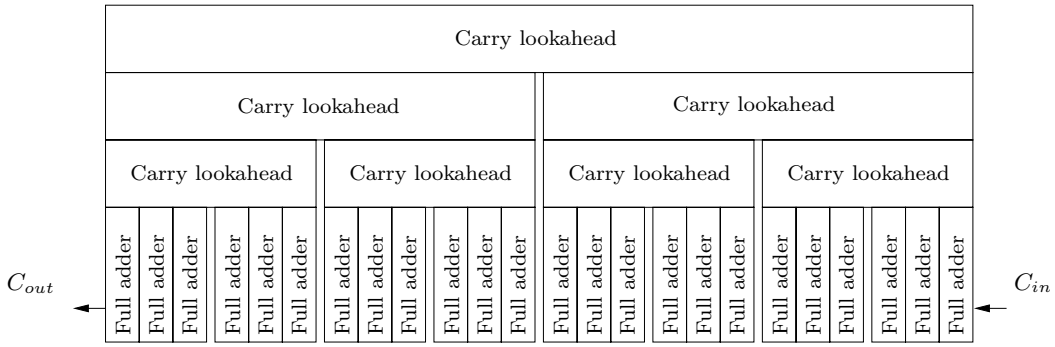


Figure 5.27: 24-bit adder block diagram

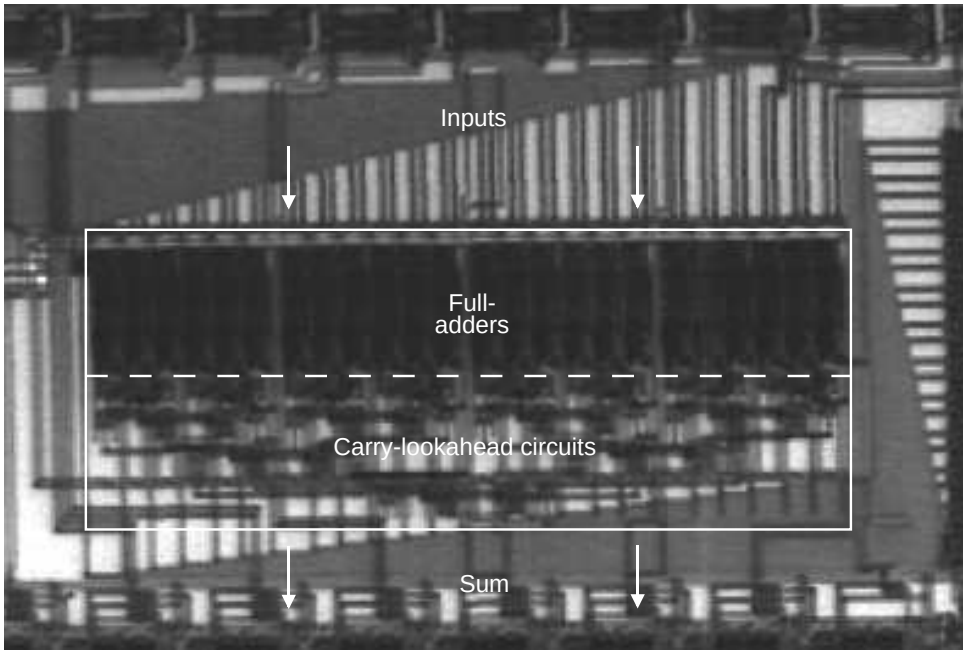


Figure 5.28: Microphotograph of a 24-bit adder

5.4.6 Controllers

Spiffee's controllers maintain the state of the processor and coordinate the synchronization of the functional units. Chip control is performed by two main sub-units: the memory controller and the central processor controller. The system is partitioned in this way because the operation of the main memory is largely independent from the rest of the system.

Memory controller

The memory controller operates in a "slave" mode since it accepts commands from the main processor controller. Output signals from the memory controller run to either the main memory or the cache set which is currently connected to the memory.

Processor controller

The processor controller maintains the system-level state of Spiffee including the current *epoch*, *group*, *pass*, and *butterfly*. There are two basic modes of operation:

1. *Single FFT* — A single FFT is calculated at high speed followed by a halting of the processor.
2. *Free run* — FFTs are run continuously.

The primary inputs to the processor controller are slow-speed command and test signals which include:

- *ResetChip*
- *Go* — begins high-speed execution
- *FreeRun* — asserts continuous running mode
- *Test_mode* — enables scan paths

Figure 5.29 shows a microphotograph of the two controllers. The four counters for the *epoch*, *group*, *pass*, and *butterfly* are indicated inside the processor controller. The non-synthesized, hand-laid-out style of the controller's circuits is evident in the figure.

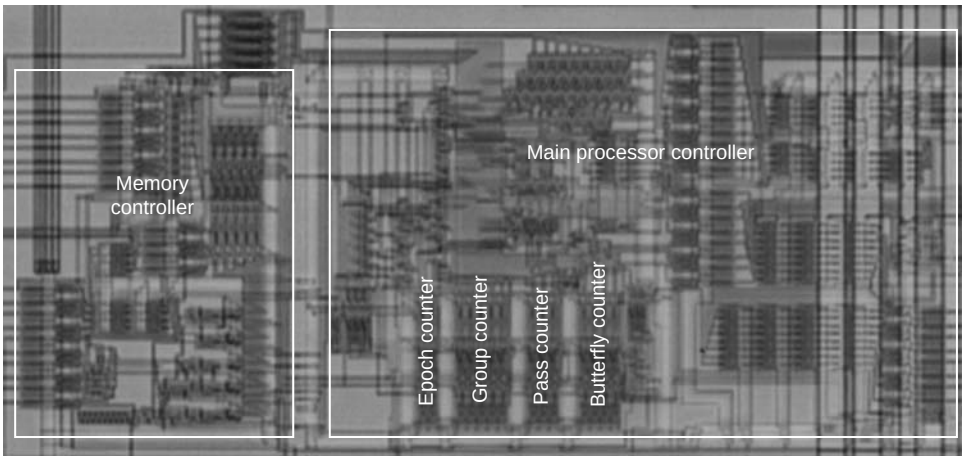


Figure 5.29: FFT processor controller

5.4.7 Clocking Methodology, Generation and Distribution

Spiffee's clocking scheme uses one single-phase global clock for the entire processor. There are no local clock buffers, as the global clock runs in a single chip-wide network. The functional units operate with a high fraction of utilization—meaning functional units are busy nearly every cycle. High utilization eliminates the need for clock gating—which is impossible with a single global clock. The advantages of a single non-buffered clock are that it is much simpler to design, and it has very low clock skew.

The clocking system contains an on-chip digitally-programmable oscillator, control circuitry, and a global clock driver.

Flip-flop design

Due to their robust operation, static edge-triggered flip-flops are used instead of latches. Figure 5.30 shows the schematic of a flip-flop. Spice simulations with low-V_t transistors operating at low supply voltages show that placing a transmission gate in series with the two feedback inverters greatly improves the range over which the flip-flop operates. The use of full transmission gates instead of NMOS pass gates similarly improves supply-voltage operating range. The generation of ϕ and $\overline{\phi}$ locally in each flip-flop slightly increased the size of each cell but reduced the loading on the global clock network by a factor of four. Layout and transistor sizing of the flip-flop cells was designed and simulated to avoid race conditions.

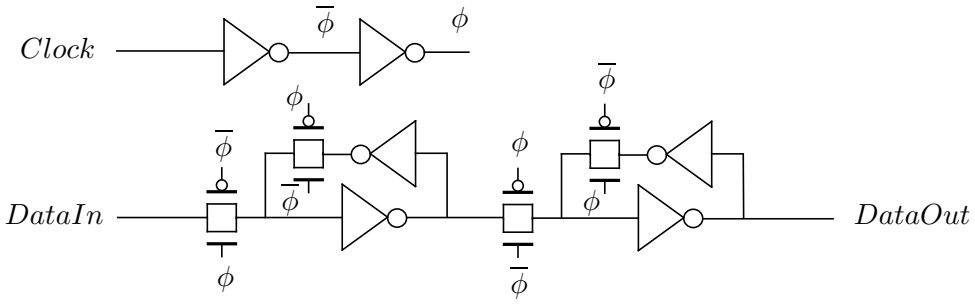


Figure 5.30: Schematic of a flip-flop with a local clock buffer

Oscillator

To achieve operation over a wide range of frequencies, the on-chip oscillator can be configured into 7-inverter, 11-inverter, or 19-inverter lengths. Typical measured frequencies in the fastest configuration are approximately eight times greater than frequencies in the slowest configuration.

The oscillator's stability is relatively insensitive to noise on the processor's power supply lines or in the substrate. Good noise resistance is achieved by making all control inputs (except two) digital and locally latched. In addition, large substrate guard rings and independent V_{dd} and *Gnd* nodes help isolate the oscillator from the rest of the chip.

Clock controller

The clock controller operates independently of the processor and memory controllers and has three primary modes of operation:

1. *Internal oscillator, free run.*

2. *Internal oscillator, burst* — The clock runs at high speed for 1 to 63 cycles. In conjunction with the scan paths, bursting the clock for a small number of cycles may enable the accurate measurement of individual functional unit latencies. The clock should be burst for the number of clock cycles equal to the number of pipeline stages in the functional unit, plus one.

3. *External clock input.*

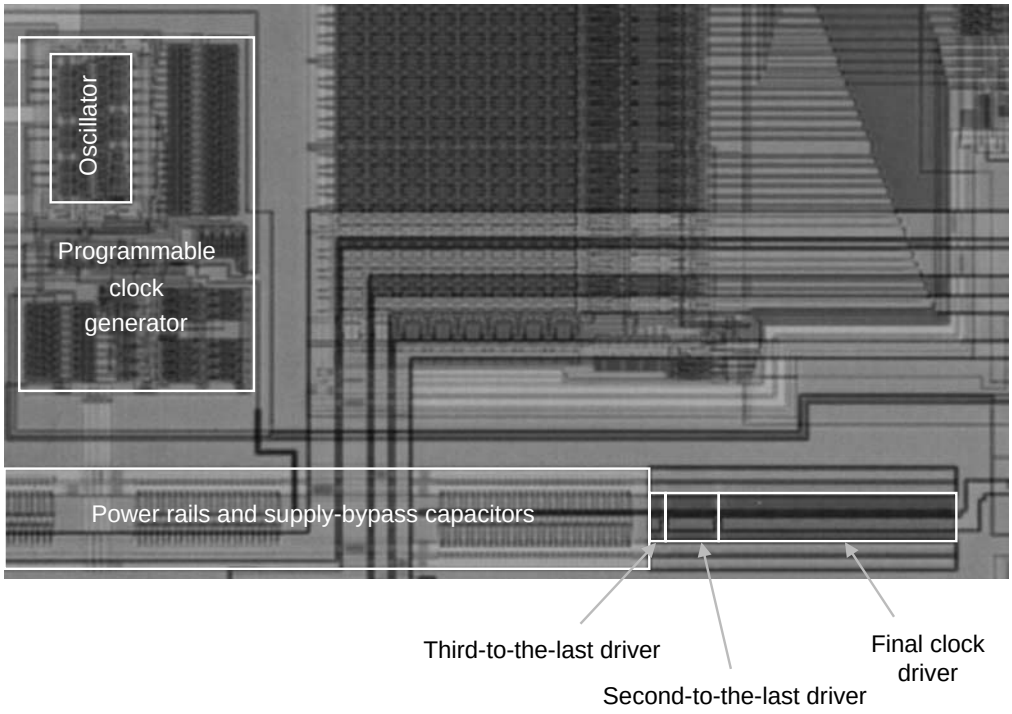


Figure 5.31: Microphotograph of clocking circuitry

Though relatively simple in design, the clock controller must operate at very high speeds—faster than the fastest functional unit on the chip.

Global clock buffer

The final three stages of the global clock buffer have a fanout of approximately 4.5 and are ratioed with the PMOS transistors 1.5× larger than corresponding NMOS transistors. Transistors of the three final stages have the following widths:

Third-to-the-last stage — 360λ PMOS × 240λ NMOS

Second-to-the-last stage — 1560λ PMOS × 1040λ NMOS

Final stage — 7680λ PMOS × 5120λ NMOS

where $\lambda = 0.35\ \mu$m. Figure 5.31 shows a microphotograph of the clock generation and clock buffer circuitry. Rows of control signal latches surround the oscillator. The final three

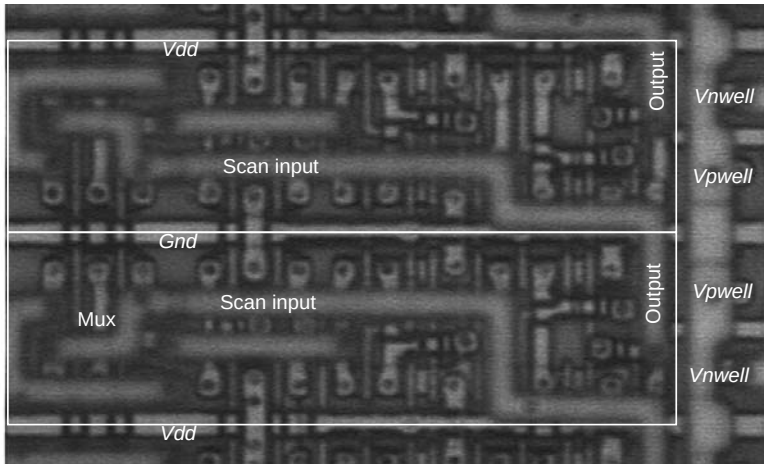


Figure 5.32: Microphotograph of scannable latches

stages of clock buffers are adjacent to a large array of bypass capacitors which filter the clock driver's V_{dd} and *Gnd* supply rails.

5.4.8 Testing and Debugging

To reduce the die size, the chip contains only nineteen I/O signals for data and control. To enable observation and control of data throughout the chip with a low I/O pad count, many flip-flops possess scan-path capability. While the global test signal *test_mode* is asserted, all scannable flip-flops are connected in a 650-element scan path. Data stored in flip-flops are shifted off-chip one bit at a time through a single I/O pad, and new data are simultaneously shifted into the shift chain through another I/O pad. In this way, the state of the scannable flip-flops is observed and then either restored or put into a new state.

The scan path is implemented by placing a multiplexer onto the input of each flip-flop. In normal mode, the multiplexer routes data into the flip-flop. In test mode, the output of an adjacent flip-flop is routed into the input. Figure 5.32 shows a high-magnification microphotograph of a column of flip-flops. The wire labeled *Scan input* is used during test mode to route the output of the lower flip-flop into the input of the upper flip-flop. Another interesting visible feature is the well/substrate connections on the far right side labeled *Vnwell* and *Vpwell*. For these flip-flops, it is sufficient to contact the well/substrate only on one end of the circuit, and considerable area is saved by not routing *Vnwell* and *Vpwell* rails parallel to the V_{dd} and *Gnd* power rails.

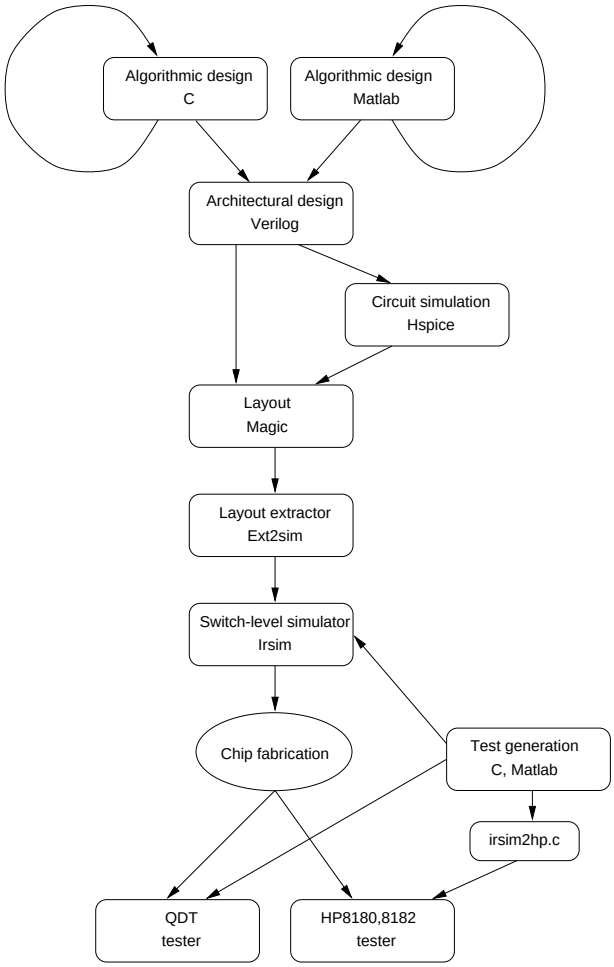


Figure 5.33: Design flow and CAD tools used

5.5 Design Approach and Tools Used

This section presents an overview of the design methodology and the CAD tools used to design the Spiffee processor. Figure 5.33 contains a flowchart depicting the primary steps taken in the design, fabrication, and test of the processor.

5.5.1 High-Level Design

The C and Matlab programming languages were used for algorithmic-level simulation and verification because of their high execution speed. In total, about ten simulations at various levels of abstraction were written.

Next, details of the architecture were fleshed out in more detail using the Verilog hardware description language and a Cadence Verilog-XL simulator. Approximately twenty total modules for the processor and its sub-blocks were written.

Circuit-level simulations were performed using Hspice. SRAM, cache, and ROM circuits required thorough Hspice analysis because of their more analog-like nature, and because their circuits are not as robust as static logic circuits. Other circuits were simulated using Hspice more for performance measurement reasons than to ensure functionality.

5.5.2 Layout

Because of the unusual circuit styles and layout required for the low-V_{dd}, low-V_t circuits, the entire chip was designed "full-custom"—meaning the design was done (almost) entirely by hand without synthesis or place-and-route tools. Layout was done using the CAD tool Magic. The only layout that was not done completely by hand was the programming of the ROMs. A C-language program written by the author places ROM cells in the correct locations and orientations in the ROM array and generates a Magic data file directly.

The extraction of devices and parasitic capacitances from the layout was done using Magic and Ext2sim. The switch-level simulator Irsim was used to run simulations on extracted layout databases. Test vectors for Irsim were generated using a combination of C and Matlab programs.

5.5.3 Verification

Chip testing was first attempted using an HP8180 Data Generator and an HP8182 Data Analyzer. Irsim test vectors were automatically converted to tester command files using the C program irsim2hp, which was written by the author. Because of limitations on vector length and the inability of the testers to handle bidirectional chip signals, testing using the HP8180 and HP8182 was eventually abandoned in favor of the QDT tester. A C program written by the author directly reads Irsim command files and controls the QDT tester. The QDT tester was successfully used to test and measure the Spiffee1 processor.

5.6 Summary

This chapter presents key features of the algorithmic, architectural, and physical-level design of the Spiffee processor. Spiffee is a single-chip, 1024-point complex FFT processor designed to operate robustly in a low-V_{dd}, low-V_t environment with high energy-efficiency.

The processor utilizes the cached-FFT algorithm detailed in Ch. 4 using a main memory of 1024 complex words and a cache of 32 complex words. To attain high performance, Spiffee has a well-balanced, nine-stage pipeline that operates with a short cycle time.

Chapter 6

Measured and Projected Spiffee Performance

This chapter reports measured results of *Spiffee1*, which is a version of the Spiffee processor fabricated using a high-V_t[1] process. Clock frequency, FFT execution time, energy dissipation, energy × time, and power data are presented and compared with other processors. A portion of the processor was manufactured using a 0.26 μm low-V_t[1] process; data from those circuits are used to predict the performance of a complete low-V_t version of Spiffee. Finally, results of simulations which estimate the performance of a hypothetical version of Spiffee fabricated in a 0.5 μm ULP process are presented.

6.1 Spiffee1

The Spiffee processor described in Ch. 5 was manufactured during July of 1995 using a standard, single-poly, triple-metal CMOS technology. Hewlett-Packard produced the device using their CMOS14 process. MOSIS handled foundry-interfacing details and funded the fabrication. MOSIS design rules corresponding to a 0.7 μm process ($\lambda = 0.35\ \mu$m) with $L_{poly} = 0.6\ \mu$m were used. The die contains 460,000 transistors and occupies 5.985 mm × 8.204 mm. Appendix A contains a summary of Spiffee1's key features. The processor is fully functional on its first fabrication.

[1] In this chapter, "high-V_t" refers to MOS devices or processes with transistor thresholds in the range of 0.7 V–0.9 V, and "low-V_t" refers to MOS devices or processes with thresholds less than approximately 0.3 V.

Well/substrate bias NMOS: V_{bs}, PMOS: V_{sb} (Volts)	NMOS V_t (Volts)	PMOS V_t (Volts)
−2.0 V	0.96 V	−1.14 V
0.0 V	0.68 V	−0.93 V
+0.5 V	0.48 V	−0.82 V

Table 6.1: Measured V_t values for Spiffee1

Although optimized to operate in a low-V_t CMOS process, Spiffee was manufactured first in a high-V_t process to verify its algorithm, architecture, and circuits. Figure 6.1 shows a die microphotograph of Spiffee1. Figure 5.11 on page 94 shows a corresponding block diagram.

6.1.1 Low-Power Operation

Since Spiffee1 was fabricated using a high-V_t process, tuning transistor thresholds through the biasing of its n-wells and p-substrate is unnecessary for normal operation. However, because the threshold voltages are so much higher than desired, lowering the thresholds improves low-V_{dd} performance. Thresholds are lowered by *forward* biasing the n-wells and p-substrate.

Forward biasing the wells and substrate is not a standard technique and entails some risk. Positively biasing the n-well/p+ and p-substrate/n+ diodes significantly increases the chances of latchup, and results in substantial diode currents as the bias voltages approach +0.7 V. Despite this, latchup was never observed throughout the testing of multiple chips at biases of up to +0.6 V. At supply voltages below approximately 0.9 V, the risk of latchup disappears as there is insufficient voltage to maintain a latchup condition.

Table 6.1 details the 480 mV and 320 mV V_t tuning range measured for NMOS and PMOS devices respectively. Because the absolute value of the PMOS thresholds is so much larger than the NMOS thresholds, the PMOS threshold voltage is the primary limiter of performance at low supply voltages.

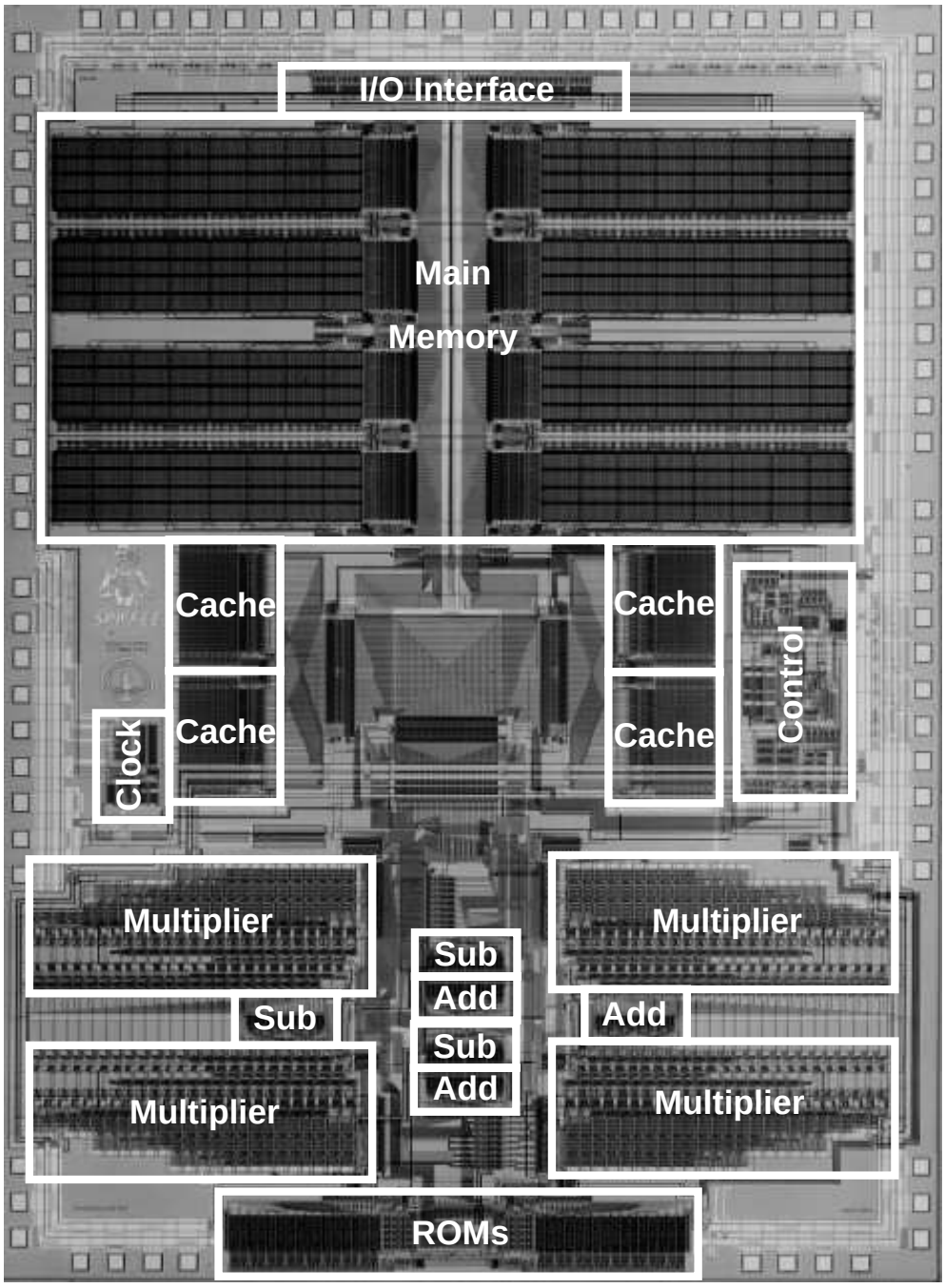


Figure 6.1: Microphotograph of the Spiffee1 processor

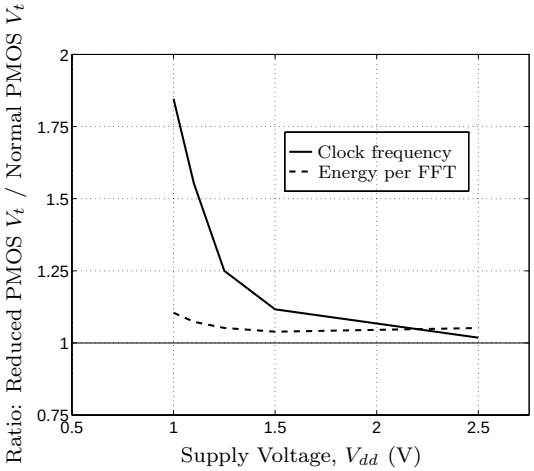


Figure 6.2: Measured change in performance and energy-consumption with an n-well bias of $V_{sb} = +0.5$ V applied, compared to operation without bias

The device operates at a minimum supply voltage slightly less than 1.0V. At $V_{dd} = 1.1$V, the chip runs at 16MHz and 9.5mW with the n-wells forward biased +0.5V ($V_{sb} = +0.5$V)—which is a 60% improvement over the 10MHz operation without bias. With +0.5V of n-well bias, 11 μA of current flows from the n-wells while the chip is active. Figure 6.2 shows the dramatic improvement in operating frequency and the slight increase in energy consumption per FFT caused by adjusting PMOS V_ts, for various values of V_{dd}.

6.1.2 High-Speed Operation

At a supply voltage of 3.3V, Spiffee is fully functional at 173MHz—calculating a 1024-point complex FFT in 30 μsec, while dissipating 845 mW. Though stressing the device beyond its specifications, the processor is functional at 201 MHz with a supply voltage of 4.0 V.

Despite having a favorable maximum clock rate, the chip's circuits are not optimized for high-speed operation—in fact, nearly all transistors in logic circuits are near minimum size. The processor owes its high speed primarily to its algorithm and architecture, which enable the implementation of a deep and well-balanced pipeline.

6.1.3 General Performance Figures

This section presents several performance measures for the Spiffee1 processor including: clock frequency, energy dissipation, energy × time, and power. Each plot shows data

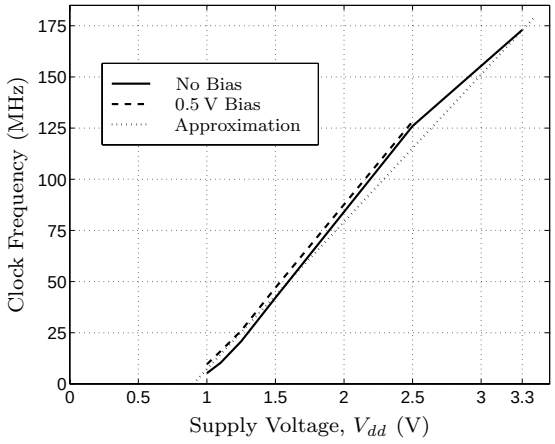


Figure 6.3: Maximum operating frequency vs. supply voltage

for the processor both with and without well and substrate biases. Solid lines indicate performance without bias ($V_{n\text{-}well} = V_{dd}$ and $V_{p\text{-}substrate} = Gnd$), and dashed lines indicate operation with an n-well forward bias of +0.5 V ($V_{dd} - V_{n\text{-}well} = +0.5$ V or $V_{sb} = +0.5$ V) and no substrate bias ($V_{p\text{-}substrate} = Gnd$). Measurements with bias applied were not made for supply voltages above 2.5 V because the performance with or without bias is virtually the same at higher supply voltages. Finally, an FFT sample input sequence is given, with FFT transforms calculated by both Matlab and Spiffee1.

Clock frequency

Figure 6.3 shows the maximum clock frequency at which Spiffee1 is functional, for various supply voltages. Although device and circuit theory predict a much more complex relationship between supply voltage and performance, the voltage vs. speed plot shown is approximated reasonably well by a constant slope for V_{dd} values greater than approximately $0.9\,\text{V} \approx V_t$ using the equation,

$$Max\ clock\ freq \approx k_f(V_{dd} - V_t), \qquad V_{dd} > V_t \tag{6.1}$$

where $k_f \approx 72\,\text{MHz/V}$. At higher supply voltages, the performance drops off slightly with a lower slope. This dropoff is likely caused by the velocity saturation of carriers.

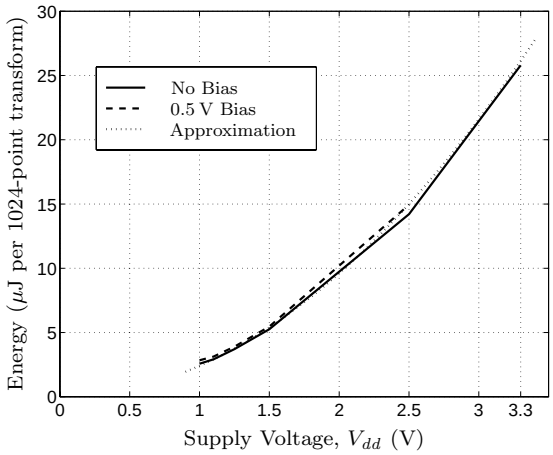


Figure 6.4: Energy consumption vs. supply voltage

Energy consumption

Figure 6.4 is a plot of the energy consumed per FFT by the processor over different supply voltages. As expected from considerations given in Sec. 3.2.3 on page 35 and by Eq. 3.5, the energy consumption is very nearly quadratic with a form closely approximated by,

$$\textit{Energy consumption} \approx k_e V_{dd}^2, \tag{6.2}$$

where $k_e \approx 2.4\,\mu\text{J/V}^2$.

Energy $\times$ time

The value of considering a merit function which incorporates energy-efficiency and performance is discussed in Sec. 3.1.1 on page 33. A popular metric which does this is *energy* $\times$ *time* (or $E \times T$), where time is the same as *delay* and is proportional to $1/$*frequency*. Figure 6.5 shows Spiffee1's measured $E \times T$ versus supply voltage.

For values of $V_{dd} \leq 1.5\,\text{V}$, the sharp increase in $E \times T$ is due to a dramatic increase in delay as V_{dd} approaches V_t. For values of $V_{dd} \geq 2.5\,\text{V}$, $E \times T$ increases due to an increase in energy consumption.

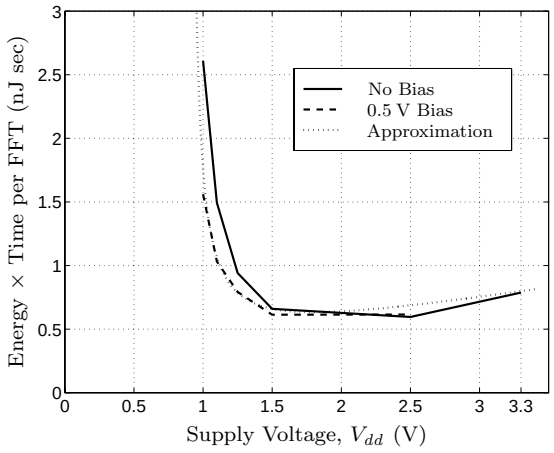


Figure 6.5: $E \times T$ per FFT vs. supply voltage

For Spiffee1, $E \times T$ per 1024-point complex FFT is,

$$E \times T = \textit{energy-per-FFT} \times \textit{time-per-FFT} \tag{6.3}$$

$$= \textit{energy-per-FFT} \times \frac{\textit{cycles-per-FFT}}{\textit{cycles-per-sec}} \tag{6.4}$$

$$= \textit{energy-per-FFT} \times \frac{5281}{\textit{frequency}}. \tag{6.5}$$

Using Eqs. 6.1 and 6.2, the $E \times T$ per FFT for $V_{dd} > V_t$ is,

$$E \times T \approx k_e V_{dd}^2 \times \frac{5281}{k_f (V_{dd} - V_t)} \tag{6.6}$$

$$\approx \frac{5281\, k_e}{k_f} \frac{V_{dd}^2}{(V_{dd} - V_t)}. \tag{6.7}$$

Equation 6.7 is plotted on Fig. 6.5 for comparison with the measured data.

Although the exact location of the $E \times T$ minimum is not discernable in Fig. 6.5 because of the spacing of the samples, it clearly falls between supply voltages of 1.4 V and 2.5 V. The magnitude of the $E \times T$ curve is expected to be fairly constant for supply voltages in the vicinity of $3V_t$ (Horowitz *et al.*, 1994). Analytically, the minimum value of $E \times T$ is the value of V_{dd} for which,

$$\frac{d}{dV_{dd}} E \times T = 0. \tag{6.8}$$

From Eq. 6.7, this occurs near where,

$$\frac{d}{dV_{dd}}\left[\frac{5281\,k_e}{k_f}\,\frac{V_{dd}^2}{(V_{dd}-V_t)}\right] = 0, \quad V_{dd} > V_t \tag{6.9}$$

$$\frac{d}{dV_{dd}}\,\frac{V_{dd}^2}{(V_{dd}-V_t)} = 0, \quad V_{dd} > V_t \tag{6.10}$$

$$\frac{2\,V_{dd}\,\cdot(V_{dd}-V_t)-V_{dd}^2}{(V_{dd}-V_t)^2} = 0 \tag{6.11}$$

$$2\,V_{dd}\,(V_{dd}-V_t)-V_{dd}^2 = 0 \tag{6.12}$$

$$2\,V_{dd}\,(V_{dd}-V_t) = V_{dd}^2 \tag{6.13}$$

$$2(V_{dd}-V_t) = V_{dd} \tag{6.14}$$

$$2V_{dd}-V_{dd} = 2V_t \tag{6.15}$$

$$V_{dd} = 2V_t \tag{6.16}$$

Standard long-channel quadratic transistor models predict the minimum $E \times T$ value at $V_{dd} = 3V_t$ (Burr and Peterson, 1991b). However, for devices that exhibit strong short-channel effects, the drain current increases at a less-than-quadratic rate (*i.e.,* $I_{ds} \propto (V_{gs} - V_t)^x, x < 2$), and the minimum $E \times T$ point is at a lower supply voltage than $3V_t$. The measured data given here are consistent with this expectation since a 0.6μm process exhibits some short-channel effects.

From Table 6.1, the larger V_t is the PMOS V_t which is $|-0.93\,\mathrm{V}| = 0.93\,\mathrm{V}$. The minimum value of $E \times T$ is then expected to be near the point where $V_{dd} = 2V_t = 1.86\,\mathrm{V}$, which is consistent with the $E \times T$ plot of Fig. 6.5.

Power dissipation

Figure 6.6 shows a plot of Spiffee1's power dissipation over various supply voltages. The operating frequency at each supply voltage is the maximum at which it would correctly operate.

Sample input/output waveform

As part of the verification procedure, various data sequences were processed by both Matlab and Spiffee1, and the results compared. The top subplot in Fig. 6.7 shows a plot of the

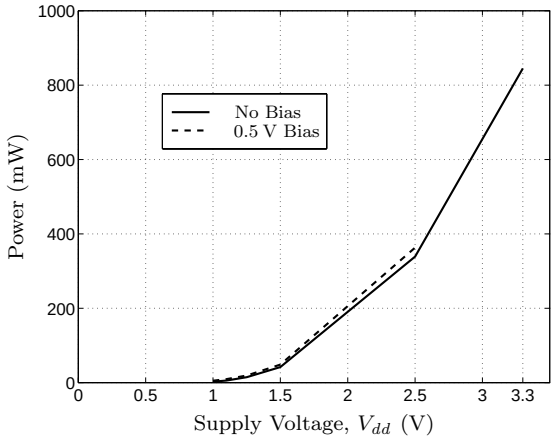


Figure 6.6: Power dissipation vs. supply voltage

input function,

$$\cos\left(\frac{2\pi \cdot 23}{N}\right) + \sin\left(\frac{2\pi \cdot 83}{N}\right) + \cos\left(\frac{2\pi \cdot 211}{N}\right) - j\sin\left(\frac{2\pi \cdot 211}{N}\right) \tag{6.17}$$

where $N = 1024$. The solid line represents the real component, and the dashed line represents the imaginary component of the sequences. The middle subplot shows the FFT of Eq. 6.17 calculated by Matlab, and the bottom subplot shows the FFT calculated by Spiffee1. Output from Spiffee1 differs from the Matlab output only by a scaling factor and the error introduced by Spiffee1's fixed-point arithmetic.

6.1.4 Analysis

Table 6.2 contains a summary of relevant characteristics of thirteen FFT processors calculating 1024-point complex FFTs. Seven of the processors are produced by companies, and six by researchers from universities. Information for processors without citation was gathered from company literature, WWW pages, and/or private communication with the designers. *CMOS Technology* is the minimum feature size of the CMOS process the chip was fabricated in. When two values are given, the first value is the base design rule dimension for the technology, and the second value is the minimum channel length. *Datapath width*, or *DPath*, is the width in number of bits, of the multipliers for the scalar datapaths. *Number of chips* values with +'s indicate additional memory chips beyond the number given are

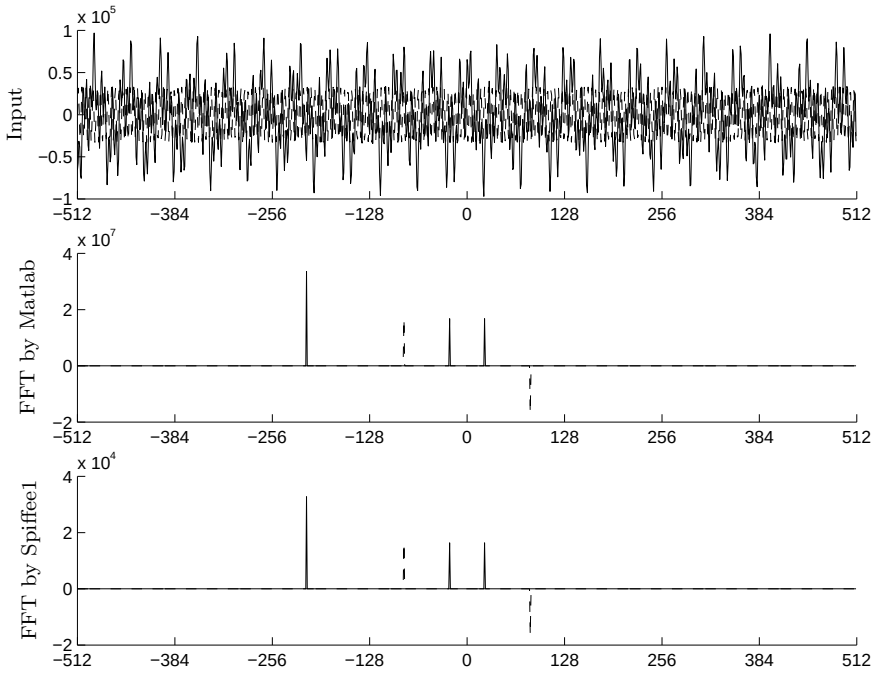


Figure 6.7: 1024-point complex input sequence with output FFTs calculated by both Matlab and Spiffee1

Processor	Year	CMOS Tech (μm)	Datapath width (bits)	1024-point Exec Time (μsec)	Power (mW)	Clock Freq (MHz)	Num of chips	Norm Area (mm^2)	FFTs per Energy
LSI, L64280 (Ruetz, 1990)	1990	1.5	20	26	20,000	40	20	233	2.9
Plessey, 16510A (O'Brien, 1989)	1989	1.4	16	98	3,000	40	1	22	3.6
Honeywell, DASP (Magar, 1988)	1988	1.2	16	102	$\sim$ 5,250	–	2+	–	1.7
Y. Zhu, U of Calgary	1993	1.2	16	155	–	33	–	–	–
Dassault Electronique	1990	1.0	12	10.2	15,000	25	6	240	3.4
Tex Mem Sys, TM-66	–	0.8	32	65	7,000	50	2+	–	3.4
Cobra, Col. State (Sunada, 1994)	1994	0.75	23	9.5	7,700	40	16+	1104+	12.4
Sicom, SNC960A	1996	0.6	16	20	2,500	65	1	–	9.0
CNET, E. Bidet (1995)[a]	1994	0.5	10	51	300	20	1	100	13.6
M. Wosnitza, ETH, Zurich (1998)[b]	1998	0.5	32	80	6000	66	1	167	2.4
Cordic FFT, R. Sarmiento (1998)[c]	1998	0.6 GaAs	8	7.5	12,500	700	1	–	2.0
Spiffee1, $V_{dd} = 3.3$ V	1995	0.7/0.6	20	30	845	173	1	25	27.6
Spiffee1, $V_{dd} = 2.5$ V	1995	0.7/0.6	20	41	363	128	1	25	47.0
Spiffee1, $V_{dd} = 1.1$ V	1995	0.7/0.6	20	330	9.5	16	1	25	**223**
Spiffee low-V_t[d], $V_{dd} = 0.4$ V	1995	0.8/0.26	20	93	9.7	57	1	25	**887**

Table 6.2: Comparison of processors calculating 1024-point complex FFTs

[a]The processor by Bidet *et al.* calculates FFTs up to 8192 points.
[b]The processor by Wosnitza *et al.* contains on-chip support for 2-dimensional convolution.
[c]The chip by Sarmiento *et al.* is fabricated using a GaAs technology.
[d]Spiffee low-V_t numbers are extrapolated from measurements of a portion of the chip fabricated in a low-V_t process.

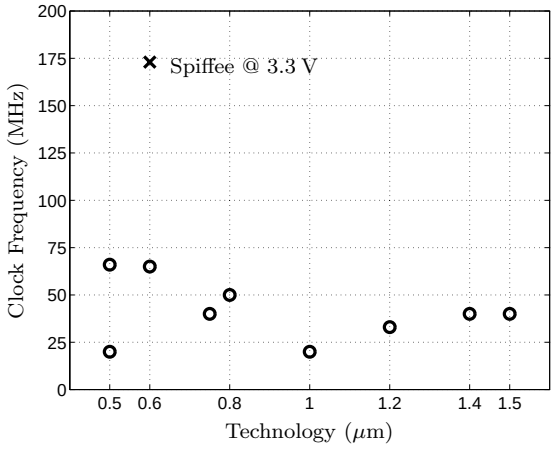


Figure 6.8: CMOS technology vs. clock frequency for processors in Table 6.2

required for data and/or coefficients. *Normalized Area* is the silicon area normalized to a 0.5 μm technology with the relationship,

$$Normalized\ Area = \frac{Area}{(Technology/0.5\ \mu\text{m})^2}. \tag{6.18}$$

The final column, *FFTs per Energy*, compares the number of 1024-point complex FFTs calculated per energy. An adjustment is made to the metric that, to first order, factors out technology and the datapath word width. The adjustment makes use of the observation that roughly 1/3 of the energy consumption of the 20-bit Spiffee processor scales as $DPath^2$ (*e.g.,* multipliers) and approximately 2/3 scales linearly with $DPath$. The value is calculated by,

$$FFTs\ per\ Energy = \frac{Technology \times \left(\frac{2}{3}\frac{DPath}{20} + \frac{1}{3}\left(\frac{DPath}{20}\right)^2\right)}{Power \times Exec\ Time \times 10^{-6}}. \tag{6.19}$$

While clock speed is not the only factor, it is certainly an important factor in determining the performance and area-efficiency of a design. Figure 6.8 compares the clock speed of Spiffee1 operating at $V_{dd} = 3.3$ V with other FFT processors, versus their CMOS technologies. Spiffee1 operates with a clock frequency that is 2.6× greater than the next fastest processor.

Figure 6.9 compares Spiffee's adjusted energy-efficiency with other processors. Operating

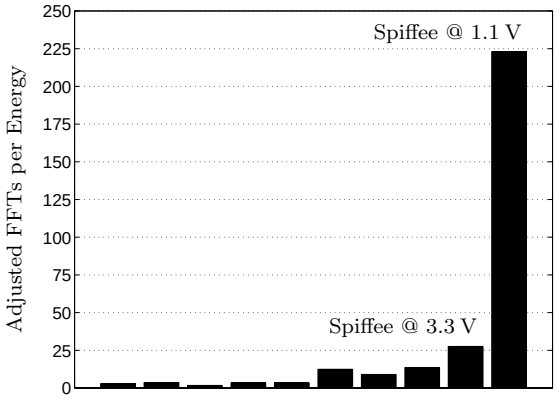


Figure 6.9: Adjusted energy-efficiency (*FFTs per Energy*, see Eq. 6.19) of various FFT processors

with a supply voltage of 1.1 V, Spiffee is sixteen times more energy-efficient than the previously most efficient known processor.

To compare $E \times T$ values of the processors in Table 6.2, we define,

$$E \times T = \frac{Exec\ Time}{FFTs\ per\ Energy}. \tag{6.20}$$

Since the quantity *FFTs per Energy* is compensated, to first order, for different *Technology* and *DPath* values, the $E \times T$ product is also compensated. Figure 6.10 compares the $E \times T$ values for various FFT processors versus their silicon areas, normalized to 0.5 μm. The dashed line highlights a constant $E \times T \times Norm\ Area$ contour.

The most comprehensive metric we consider is the product $E \times T \times Norm\ Area$. The Spiffee1 processor running at a supply voltage of 2.5 V has a $E \times T \times Norm\ Area$ product that is seventeen times lower than the processor with the previously lowest value.

The cost of a device is a strong function of its silicon area. Therefore, processors with high performance and small area will be the most cost efficient. Figure 6.11 shows the first-order-normalized FFT calculation time (*Exec Time/Technology*) versus normalized silicon area for several FFT processors. The dashed line shows a constant $Time' \times Norm\ Area$ contour. The processor presented here compares favorably with other processors despite its lightly-compacted floorplan and its less area-efficient circuits—which were designed for low-voltage operation and V_t tunability.

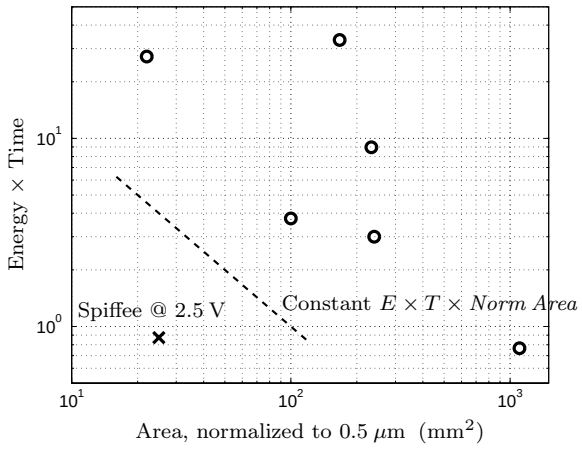


Figure 6.10: Silicon area vs. $E \times T$ (see Eq. 6.20) for several FFT processors

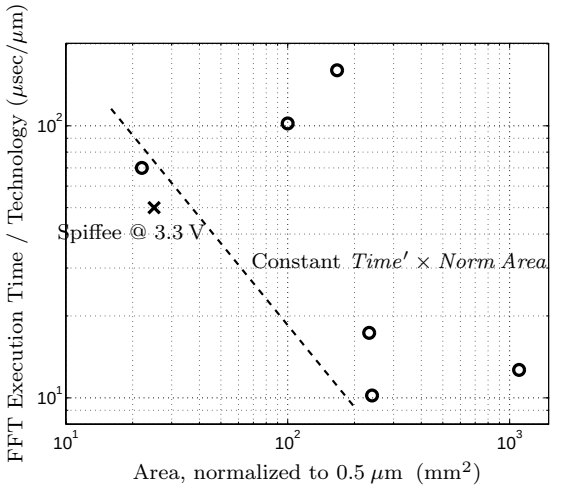


Figure 6.11: Silicon area vs. FFT execution time for CMOS processors in Table 6.2

6.2 Low-V_t Processors

6.2.1 Low-V_t 0.26 μm Spiffee

Although to date Spiffee has been fabricated only in a high-V_t process, portions of it have been fabricated in a low-V_t process. Fabrication for three test chips was provided by Texas Instruments in an experimental 0.25 μm process similar to one described by Nandakumar *et al.* (1995). The process provides two thresholds with the lower threshold being approximately 200 mV, and the drawn channel length $L_{poly} = 0.26\,\mu$m. The chips were fabricated using 0.8 μm design rules and with all transistors having the lower threshold.

All three test chips contain the identical oscillator, clock controller circuits, and layout described in Sec. 5.4.7 on page 120. The oscillator and controller contain approximately 1300 transistors and are supplied power by an independent pad. Unfortunately, the oscillator's power supply in the low-V_t versions also powers some extra circuits not included in the Spiffee1 version, which caused the estimates for a complete low-V_t Spiffee processor to be a little high.

The test circuits are functional at a supply voltage below 300mV. Power and performance measurements were made of the test chips. From measured low-V_t-chip data, measured Spiffee1 data, and measured data of Spiffee1's oscillator; the following estimates were made for a low-V_t version of Spiffee running at a supply voltage of 400 mV:

- 57 MHz clock frequency
- 1024-point complex FFT calculated in 93 μsec
- power dissipation less than 9.7 mW
- more than 65× greater energy-efficiency than the previously most efficient processor

Table 6.2 includes more information on the hypothetical low-V_t processor.

6.2.2 ULP 0.5 μm Spiffee

A version of Spiffee fabricated in a ULP process is expected to perform even better than the low-V_t version. Simulations in one typical 0.5 μm ULP process give the following estimates

while operating at a supply voltage of 400 mV:

- 85 MHz clock frequency
- 1024-point complex FFT calculated in 61 μsec
- power dissipation of 8 mW
- more than 75$\times$ more energy-efficient than the previously most efficient processor

Although the adjusted energy-efficiency of the ULP version is comparable to the low-V_t version, the performance is significantly better in the ULP case since this data comes from a processor with 0.5 μm transistors while the low-V_t processor has 0.26 μm transistors.

Chapter 7

Conclusion

7.1 Contributions

The key contributions of this research are:

1. The cached-FFT algorithm, which exploits a hierarchical memory structure to increase performance and reduce power dissipation, was developed. New terms describing the form of the cached-FFT (epoch, group, and pass), are introduced and defined. An implementation procedure is provided for transforms of variable lengths, radices, and cache sizes.

   The cached-FFT algorithm removes a processor's main memory from its critical path enabling: (i) higher operating clock frequencies, (ii) reduced power dissipation by reducing communication energy, and (iii) a clean partitioning of the system into high-activity and low-activity portions—which is important for implementations using low-V_{dd}, low-V_t CMOS technologies.

2. A wide variety of circuits were designed which operate robustly in a low-V_{dd}, low-V_t environment. The circuit types include: SRAM, multi-ported SRAM, ROM, multiplier, adder/subtracter, crossbar, control, clock generation, bus interface, and test circuitry. These circuits operate at a supply voltage under 1.0 V using a CMOS technology with PMOS thresholds of -0.93 V. A few of these circuits were fabricated in a technology with thresholds near 200 mV and were verified functional at supply voltages below 300 mV.

3. A single-chip, 1024-point complex FFT processor was designed, fabricated, and verified to be fully functional on its first fabrication. The processor utilizes the cached-FFT algorithm with two sets of two banks of sixteen-word cache memories. In order to increase the overall energy-efficiency, it operates with a low supply voltage that approaches the transistor threshold voltages.

   The device contains 460,000 transistors and was fabricated in a standard 0.7 μm / 0.6 μm CMOS process. At a supply voltage of 1.1 V, the processor calculates a 1024-point complex FFT in 330 μsec while dissipating 9.5 mW—which corresponds to an adjusted energy-efficiency over sixteen times greater than the previously highest. At a supply voltage of 3.3 V, it calculates an FFT in 30 μsec while consuming 845 mW at a clock frequency of 173 MHz—which is a clock speed 2.6 times higher than the previously fastest.

   A version of the processor fabricated in a 0.8μm/0.26μm low-V_t technology is expected to calculate a 1024-point transform in 93 μsec while dissipating 9.7 mW. A version fabricated in a 0.50 μm ULP process is projected to calculate FFT transforms in 61 μsec while consuming 8 mW. These low-V_t devices would operate 65× and 75× more efficiently than the previously most efficient processor, respectively.

7.2 Future Work

This section suggests enhancements to the precision and performance of any processors using the approach presented in this dissertation. The feasibility of adapting the Spiffee processor to more complex FFT systems is also discussed.

7.2.1 Higher-Precision Data Formats

Modern digital processors represent data using notations that generally can be classified as either fixed-point, floating-point, or block-floating-point. These three data notations vary in complexity, dynamic range, and resolution. The Spiffee processor uses a fixed-point notation for data words. We now briefly review the other two data notations and consider several issues related to implementing block-floating-point on a cached-FFT processor.

Background

Fixed-point In fixed-point notation, each datum is represented by a single signed or unsigned (non-negative) word. As discussed in Sec. 5.3.6, Spiffee uses a signed, 2's-complement, fixed-point data word format.

Floating-point In floating-point notation, each datum is represented by two components: a mantissa and an exponent in the following configuration: $mantissa \times base^{exponent}$. The *base* is fixed, and is typically two or four. Both the mantissa and exponent are generally signed numbers. Floating-point formats provide greatly increased dynamic range, but significantly complicate arithmetic units since normalization steps are required whenever data are modified.

Block-floating-point Block-floating-point notation is probably the most popular format for dedicated FFT processors and is similar to floating-point notation except that exponents are shared among "blocks" of data. For FFT applications, a block of data is typically N words. Using only one exponent per block dramatically reduces a processor's complexity since words are normalized uniformly across all words within a block. The complexity of a block-floating-point implementation is closer to that of a fixed-point implementation than that of a floating-point one. However, in the worst case, block-floating-point performs the same as fixed-point.

Applications to a cached-FFT processor

While the fixed-point format permits a simple and fast design, it also gives the least dynamic range for its word length. Floating-point and block-floating-point formats provide more dynamic range, but are more complex.

Unfortunately, block-floating-point is not as well suited to a cached-FFT implementation as it is to a non-cached implementation. This is because all N words are not accessed as often (and therefore give opportunity to normalize the data) as they are when using a non-cached approach. In a cached-FFT, N words are processed once per epoch (which typically occurs two or three times per FFT), compared to a non-cached approach where N words are processed each stage (which occurs $\log_r N$ times per FFT).

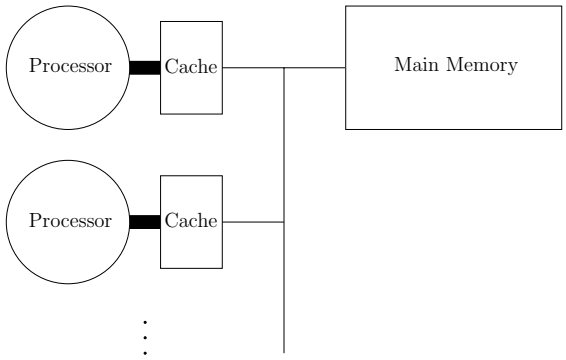


Figure 7.1: System with multiple processor-cache pairs

There are two principal approaches to applying block-floating-point to a cached-FFT:

1. One approach is to use as many exponents as there are groups in an epoch (N/C, from Eq. 4.8). The exponent for a particular group is updated on each pass. In general, multiple-exponent block-floating-point performs better than the standard single-exponent approach.

2. The second approach is to use one exponent for all N words, and only update the exponent at the beginning of the FFT, between epochs, and at the end of the FFT. In general, this approach performs worse than the standard single-exponent method.

7.2.2 Multiple Datapath-Cache Processors

Since the caching architecture greatly reduces traffic to main memory, it is possible to add additional datapath-cache pairs to reduce computation time. Figure 7.1 shows how a multiple datapath system is organized. Although bandwidth to the main memory eventually limits the scalability of this approach, processing speed can be increased several fold through this technique.

For applications which require extremely fast execution times and more processors than a single unified main memory allows, techniques such as: using separate read and write memory buses, sub-dividing the main memory into banks, or using multiple levels of caches, can allow even more processors to be integrated into a single FFT engine.

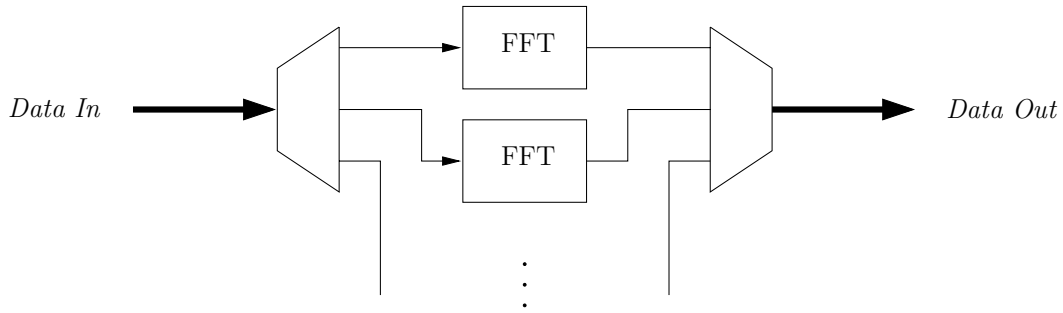


Figure 7.2: Multi-processor, high-throughput system block diagram

7.2.3 High-Throughput Systems

Most DSP applications are insensitive to modest amounts of *latency,* and their performance is typically measured in *throughput.* By contrast, the performance of general-purpose processors is strongly effected by the latency of operations (*e.g.,* memory, disk, network accesses). When calculating a parallelizable algorithm such as the FFT, it is far easier to increase throughput through the use of parallel processors than it is to decrease execution time.

For systems which calculate FFTs, require high-throughput, and are not latency sensitive (that is, latency on the order of the transform calculation time), the multi-processor topology shown in Fig. 7.2 may be used. Because FFT processors operate on blocks of N words at a time, the input data stream is partitioned into blocks of N words, and these blocks are routed to different processors. After processing, blocks of data are re-assembled into a high-speed data stream.

7.2.4 Multi-Dimensional Transforms

As FFT processor speeds continue to rise, they become increasingly attractive for use in multi-dimensional FFT applications. Wosnitza *et al.* (1998) present a chip containing an 80 μsec 1024-point FFT processor and the logic necessary to perform 1024×1024 multi-dimensional convolutions. Similarly, Spiffee could serve as the core for a multi-dimensional convolution FFT processor by adding a multiplier for "frequency-domain" multiplications, and additional control circuits.

7.2.5 Other-Length Transforms

The first version of Spiffee computes only 1024-point FFTs. As described in Sec. 4.7.4, a processor can be modified to calculate shorter-length transforms by reducing the number of passes per group. Modifying Spiffee to perform shorter, power-of-two FFTs is not difficult as it requires only a change to the chip controller.

Longer-length transforms, on the other hand, present a much more difficult modification, requiring more W_N coefficients and a larger main memory. Additional W_N coefficients can be generated using larger ROMs, a coefficient generator, or a combination of both methods. Since the main memory occupies approximately one-third of the total chip area, increasing N by a power-of-two factor significantly increases the die area.

Appendix A

Spiffee1 Data Sheet

General features	
FFT transforms	Forward and inverse, complex
Transform length	1024-point
Datapath width	20 bits + 20 bits
Dataword format	fixed-point
Number of transistors	460,000
Technology	0.7 μm CMOS
L_{poly}	0.6 μm
Size	5.985 mm $\times$ 8.204 mm
Area	49.1 mm^2
NMOS V_t	0.68 V
PMOS V_t	-0.93 V
Polysilicon layers	1
Metal layers	3
Fabrication date	July 1995
Performance at $V_{dd} = 1.1$ V	
1024-point complex FFT time	330 μsec
Power	9.5 mW
Clock frequency	16 MHz
Performance at $V_{dd} = 2.5$ V	
1024-point complex FFT time	41 μsec
Power	363 mW
Clock frequency	128 MHz
Performance at $V_{dd} = 3.3$ V	
1024-point complex FFT time	30 μsec
Power	845 mW
Clock frequency	173 MHz

Table A.1: Key measures of the Spiffee1 FFT processor

Glossary

(3,2) adder. A binary adder with three inputs and two outputs. Also called a "full adder."

(4,2) adder. A binary adder with four inputs, two outputs, a special input, and a special output.

6T cell. For *Six-transistor cell.* A common SRAM cell which contains six transistors.

activity. The fraction of cycles that a node switches.

architecture. The level of abstraction of a processor between the circuit and algorithm levels.

assert. To set the state of a node by sourcing current into or sinking current from it. *See* **drive a node.**

balanced cached-FFT. A cached-FFT in which there are an equal number of passes in the groups from all epochs.

(memory) bank. A partition of a memory.

BiCMOS. For *Bipolar CMOS.* A semiconductor processing technology that can produce both bipolar and CMOS devices.

Booth encoding. A class of methods to encode the *multiplier* bits of a multiplier.

butterfly. A convenient computational building block used to calculate FFTs.

cache. A high-speed memory usually placed between a processor and a larger memory.

CAD. For *Computer-Aided Design.* Software programs or tools used in a design process.

carry-lookahead adder. A type of carry-propagate adder.

carry-propagate adder. A class of adders which fully resolve *carry* signals along the entire word width.

charge pump. A circuit which can generate arbitrary voltages.

CLA. For *Carry-Lookahead Adder. See* **carry-lookahead adder.**

CMOS. For *Complementary Metal Oxide Semiconductor.* A silicon-based semiconductor processing technology.

CRT. For *Chinese Remainder Theorem.*

datapath. A collection of functional units which process data.

DFT. For *Discrete Fourier Transform.* A discretized version of the continuous Fourier transform.

DIF. For *Decimation In Frequency.* A class of FFT algorithms.

DIT. For *Decimation In Time.* A class of FFT algorithms.

dot diagram. A notation used to describe a multiplier's partial-product array.

DRAM. For *Dynamic Random Access Memory.* A type of memory whose contents persist only for a short period of time unless refreshed.

drive a node. To source current into or sink current from a node.

driver. A circuit which sources or sinks current.

DSP processor. For *Digital Signal Processing processor.* A special-purpose processor optimized to process signals digitally.

ECL. For *Emitter Coupled Logic.* A circuit style known for its high speed and high power dissipation.

epoch. The portion of the cached-FFT algorithm where all N data words are loaded into a cache, processed, and written back to main memory *once*.

Ext2sim. A VLSI layout extraction CAD tool.

fall time. The time required for the voltage of a node to drop from a high value (often 90% of maximum) to a low value (often 10% of maximum).

fan-in. The number of drivers connected to a common node.

fan-out. The number of receivers connected to a common node.

FFT. For *Fast Fourier Transform.* A class of algorithms which efficiently calculate the DFT.

fixed-point. A format for describing data in which each datum is represented by a single word.

flush (a cache). To write the entire cache contents to main memory.

full adder. *See* **(3,2) adder.**

functional unit. A loosely-defined term used to describe a block that performs a high-level function, such as an adder or a memory.

GaAs. For *Gallium Arsenide.* A semiconductor processing technology that uses Gallium and Arsenic as semiconducting materials.

general-purpose processor. A non-special-purpose processor (*e.g.,* PowerPC, Sparc, Intel x86).

group. The portion of an epoch where a block of data is read from main memory into a cache, processed, and written back to main memory.

high V_t. MOS transistor thresholds in the range of 0.7 V–0.9 V.

Hspice. A commercial spice simulator by Avant! Corporation. *See* **spice.**

IDFT. For *Inverse Discrete Fourier Transform.* The inverse counterpart to the forward DFT.

IFFT. For *Inverse Fast Fourier Transform.* The inverse counterpart to the forward FFT.

in-place. A butterfly is *in-place* if its inputs and outputs use the same memory locations. An *in-place FFT* uses only in-place butterflies.

Irsim. A switch-level simulation CAD tool.

keeper. A circuit which helps maintain the voltage level of a node.

layout. The physical description of all layers of a VLSI design.

load (a cache). To copy data from main memory to a cache memory.

low V_t. MOS transistor thresholds less than approximately 0.3 V.

Magic. A VLSI layout CAD tool.

metal1. The lowest or first level of metal on a chip.

metal2. The second level of metal on a chip.

metal3. The third level of metal on a chip.

MOS. For *Metal Oxide Semiconductor.* A type of semiconductor transistor.

MOSIS. A low-cost prototyping and small-volume production service for VLSI circuit development.

multiplicand. One of the inputs to a multiplying functional unit.

multiplier. One of the inputs to a multiplying functional unit.

NMOS. For *N-type Metal Oxide Semiconductor.* A type of MOS transistor with "n-type" diffusions. Also a circuit style or technology which uses only NMOS-type transistors.

n-well. The region of a chip's substrate with lightly-doped n-type implantation.

pad. A large area of metallization on the periphery of a chip used to connect the chip to the chip package.

pass. The portion of a group where each word in the cache is read, processed with a butterfly, and written back to the cache *once*.

pipeline stall. A halting of execution in a processor's datapath to allow the resolution of a conflict.

PMOS. For *P-type Metal Oxide Semiconductor.* A type of MOS transistor with "p-type" diffusions. Also a circuit style or technology which uses only PMOS-type transistors.

poly. *See* **polysilicon.**

polysilicon. A layer in a CMOS chip composed of polycrystalline silicon.

precharge. To set the voltage of a node in a dynamic circuit before it is evaluated.

predecode. To partially decode an address.

process. *See* **semiconductor processing technology.**

pseudo-code. Computer instructions written in an easy-to-understand format that are not necessarily from a particular computer language.

p-substrate. The lightly-doped p-type substrate of a chip.

p-well. The region of a chip's substrate with lightly-doped p-type implantation.

register file. The highest-speed memory in a processor, typically consisting of 16–32 words with multiple ports.

rise time. The time required for the voltage of a node to rise from a low value (often 10% of maximum) to a high value (often 90% of maximum).

ROM. For *Read Only Memory.* A type of memory whose contents are set during manufacture and can only be read.

RRI-FFT. For *Regular, Radix-r, In-place FFT.* A type of FFT algorithm.

scan path. A serially-interfaced testing methodology which enables observability and controllability of nodes inside a chip.

semiconductor processing technology. The collection of all necessary steps and parameters for the fabrication of a semiconductor integrated circuit. Sometimes referred to as simply "process" or "technology."

sense amplifier. A circuit employed in memories which amplifies small-swing read signals from the cell array.

(memory) set. A redundant copy of a memory.

SiGe. For *Silicon Germanium.* A semiconductor processing technology that uses Silicon Germanium as a semiconducting material.

SOI. For *Silicon On Insulator.* A semiconductor processing technology.

span. The maximum distance (measured in memory locations) between any two butterfly legs.

spice. A CAD circuit simulator.

Spiffee. A single-chip, 1024-point FFT processor design. The name is loosely derived from: Stanford Low-Power, High-Performance, FFT Engine.

Spiffee1. The first fabricated Spiffee processor.

SRAM. For *Static Random Access Memory.* A type of memory whose contents are preserved as long as the supply voltage is maintained.

stage. The part of a non-cached-FFT where all N memory locations are read, processed by a butterfly, and written back once.

stride. The distance (measured in memory locations) between adjacent "legs" or "spokes" of a butterfly.

technology. *See* **semiconductor processing technology.**

transmission gate. A circuit block consisting of an NMOS and a PMOS transistor, where the sources and drains of each transistor are connected to each other.

twiddle factor. A multiplicative constant used between stages of some FFTs.

ULP. For *Ultra Low Power.* A semiconductor processing technology.

Verilog. A hardware description language.

VLIW. For *Very Long Instruction Word.* A computer architecture utilizing very wide instructions.

VLSI. For *Very Large Scale Integration.* A loosely-defined term referring to integrated circuits with minimum feature sizes less than approximately 1.0 μm.

Bibliography

Amrutur, B. S. and M. A. Horowitz. "A Replica Technique for Wordline and Sense Control in Low-Power SRAM's." *IEEE Journal of Solid-State Circuits*, vol. 33, no. 8, pp. 1208–1219, August 1998.

Antoniadis, D. "SOI CMOS as a Mainstream Low-Power Technology: A Critical Assessment." In *International Symposium on Low Power Electronics and Design*, pp. 295–300, August 1997.

Assaderaghi, F., S. Parke, P. K. Ko, and C. Hu. "A Novel Silicon-On-Insulator (SOI) MOSFET for Ultra Low Voltage Operation." In *IEEE Symposium on Low Power Electronics*, volume 1, pp. 58–59, October 1994.

Athas, W., N. Tzartzanis, L. Svensson, L. Peterson, H. Li, X. Jiang, P. Wang, and W.-C. Liu. "AC-1: A Clock-powered Microprocessor." In *International Symposium on Low Power Electronics and Design*, pp. 328–333, August 1997.

Athas, W. C., L. J. Svensson, J. G. Koller, N. Tzartzanis, and E. Y.-C. Chou. "Low-Power Digital Systems Based on Adiabatic-Switching Principles." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 398–407, December 1994.

Baas, B. M. "A Pipelined Memory System For an Interleaved Processor." Technical Report NSF-GF-1992-1, STARLab, EE Department, Stanford University, June 1992.

Baas, B. M. "An Energy-Efficient FFT Processor Architecture." Technical Report NGT-70340-1994-1, STARLab, EE Department, Stanford University, January 1994.

Baas, B. M. "An Energy-Efficient Single-Chip FFT Processor." In *Symposium on VLSI Circuits*, pp. 164–165, June 1996.

Baas, B. M. "A 9.5 mW 330 μsec 1024-point FFT Processor." In *IEEE Custom Integrated Circuits Conference*, pp. 127–130, May 1998.

Baas, B. M. "A Low-Power, High-Performance, 1024-point FFT Processor." *IEEE Journal of Solid-State Circuits*, March 1999. In press.

Bailey, D. H. "FFTs in External or Hierarchical Memory." *The Journal of Supercomputing*, vol. 4, no. 1, pp. 23–35, March 1990.

Bakoglu, H. B. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, Reading, MA, 1990.

Barke, E. "Line-to-Ground Capacitance Calculation for VLSI: A Comparison." *IEEE Transactions on Computer Aided Design*, vol. 7, no. 2, pp. 295–298, February 1988.

Bewick, G. W. *Fast Multiplication: Algorithms and Implementation.* PhD thesis, Stanford University, Stanford, CA, February 1994.

Bidet, E., D. Castelain, C. Joanblanq, and P. Senn. "A Fast Single-Chip Implementation of 8192 Complex Point FFT." *IEEE Journal of Solid-State Circuits*, vol. 30, no. 3, pp. 300–305, March 1995.

Bier, J. "Processors for DSP—The Options Multiply." October 1997. Lecture given to EE380 course at Stanford University.

Blahut, R. E. *Fast Algorithms for Digital Signal Processing.* Addison-Wesley, Reading, MA, 1985.

Bracewell, R. N. *The Fourier Transform and Its Applications.* McGraw-Hill, New York, NY, second edition, 1986.

Brenner, N. M. "Fast Fourier Transform of Externally Stored Data." In *IEEE Transactions on Audio and Electroacoustics*, volume AU-17, pp. 128–132, June 1969.

Brigham, E. O. *The Fast Fourier Transform and Its Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

Burr, J. B. "Stanford Ultra Low Power CMOS." In *Symposium Record, Hot Chips V*, pp. 7.4.1–7.4.12, August 1993.

Burr, J. B., Z. Chen, and B. M. Baas. "Stanford Ultra-Low-Power CMOS Technology and Applications." In *Low-power HF Microelectronics, a Unified Approach*, chapter 3, pp. 85–138. The Institution of Electrical Engineers, London, UK, 1996.

Burr, J. B. and A. M. Peterson. "Energy considerations in multichip-module based multiprocessors." In *IEEE International Conference on Computer Design*, pp. 593–600, 1991.

Burr, J. B. and A. M. Peterson. "Ultra low power CMOS technology." In *NASA VLSI Design Symposium*, pp. 4.2.1–4.2.13, 1991.

Burr, J. B. and J. Shott. "A 200mV Self-Testing Encoder/Decoder using Stanford Ultra-Low-Power CMOS." In *IEEE International Solid-State Circuits Conference*, volume 37, pp. 84–85, 316, 1994.

Burrus, C. S. "Index Mappings for Multidimensional Formulation of the DFT and Convolution." In *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume ASSP-25, pp. 239–242, June 1977.

Burrus, C. S. and T. W. Parks. *DFT/FFT and Convolution Algorithms*. John Wiley & Sons, New York, NY, 1985.

Carlson, D. A. "Using Local Memory to Boost the Performance of FFT Algorithms on the Cray-2 Supercomputer." *The Journal of Supercomputing*, vol. 4, no. 4, pp. 345–356, January 1991.

Chandrakasan, A., A. Burstein, and R. Brodersen. "Low-Power chipset for a Portable Multimedia I/O Terminal." *IEEE Journal of Solid-State Circuits*, vol. 29, no. 12, pp. 1415–1428, December 1994.

Chandrakasan, A., S. Sheng, and R. Brodersen. "Low-Power CMOS Digital Design." *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–483, April 1992.

Chandrakasan, A. P. and R. W. Brodersen. "Minimizing Power Consumption in Digital CMOS Circuits." *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, April 1995.

Cooley, J. W., P. A. W. Lewis, and P. D. Welch. "Historical Notes on the Fast Fourier Tranform." In *IEEE Trans. on Audio and Electroacoustics*, volume AU-15, pp. 76–79, June 1967.

Cooley, J. W. and J. W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." In *Math. of Comput.*, volume 19, pp. 297–301, April 1965.

Danielson, G. C. and C. Lanczos. "Some Improvements in Practical Fourier Analysis and Their Application to X-ray Scattering From Liquids." In *J. Franklin Inst.*, volume 233, pp. 365–380,435–452, April 1942.

DeFatta, D. J., J. G. Lucas, and W. S. Hodgkiss. *Digital Signal Processing: A System Design Approach*. John Wiley & Sons, New York, NY, 1988.

Gannon, D. and W. Jalby. "The Influence of Memory Hierarchy on Algorithm Organization: Programming FFTs on a Vector Multiprocessor." In Jamieson, L., D. Gannon, and R. Douglass, editors, *The Characteristics of Parallel Algorithms*, chapter 11, pp. 277–301. MIT Press, Cambridge, MA, 1987.

Gauss, C. F. "Nachlass: Theoria Interpolationis Methodo Nova Tractata." In *Carl Friedrich Gauss, Werke, Band 3*, pp. 265–303, 1866.

GEC Plessey Semiconductors. *PDSP16510A MA Stand Alone FFT Processor*. Wiltshire, United Kingdom, March 1993.

Gentleman, W. M. and G. Sande. "Fast Fourier Transforms—For Fun and Profit." In *AFIPS Conference Proceedings*, volume 29, pp. 563–578, November 1966.

Gold, B. Private communication with author, 13 May 1997.

Gordon, B. M. and T. H. Meng. "A 1.2mW Video-Rate 2-D Color Subband Decoder." *IEEE Journal of Solid-State Circuits*, vol. 30, no. 12, pp. 1510–1516, December 1995.

Hall, J. S. "An Electroid Switching Model for Reversible Computer Architectures." In *Proceedings of ICCI '92, 4th International Conference on Computing and Information*, 1992.

He, S. and M. Torkelson. "Design and Implementation of a 1024-point Pipeline FFT Processor." In *IEEE Custom Integrated Circuits Conference*, pp. 131–134, May 1998.

Heideman, M. T., D. H. Johnson, and C. S. Burrus. "Gauss and the History of the Fast Fourier Transform." In *IEEE ASSP Magazine*, pp. 14–21, October 1984.

Hennessy, J. L. and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, second edition, 1996.

Holmann, E. "A VLIW Processor for Multimedia Applications." In *Hot Chips 8 Symposium*, pp. 193–202, August 1996.

Horowitz, M., T. Indermaur, and R. Gonzalez. "Low-Power Digital Design." In *IEEE Symposium on Low Power Electronics*, volume 1, pp. 8–11, October 1994.

Hunt, B. W., K. S. Stevens, B. W. Suter, and D. S. Gelosh. "A Single Chip Low Power Asynchronous Implementation of an FFT Algorithm for Space Applications." In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 216–223, April 1998.

Ida, J., M. Yoshimaru, T. Usami, A. Ohtomo, K. Shimokawa, A. Kita, and M. Ino. "Reduction of Wiring Capacitance with New Low Dielectric SiOF Interlayer Film and High Speed / Low Power Sub-Half Micron CMOS." In *Symposium on VLSI Technology*, June 1994.

Itoh, K., A. R. Fridi, A. Bellaouar, and M. I. Elmasry. "A Deep Sub-V, Single Power-Supply SRAM Cell with Multi-V_t, Boosted Storage Node and Dynamic Load." In *Symposium on VLSI Circuits*, June 1996.

Itoh, K., K. Sasaki, and Y. Nakagome. "Trends in Low-Power RAM Circuit Technologies." *Proceedings of the IEEE*, vol. 83, no. 4, pp. 524–543, April 1995.

Jackson, L. B. *Digital Filters and Signal Processing.* Kluwer Academic, Boston, MA, 1986.

Lam, M. S., E. E. Rothberg, and M. E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms." In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, April 1991.

LSI Logic Corporation. *Implementing Fast Fourier Transform Systems with the L64280/81 Chip Set.* Milpitas, CA, April 1990.

LSI Logic Corporation. *L64280 Complex FFT Processor (FFTP).* Milpitas, CA, April 1990.

LSI Logic Corporation. *L64281 FFT Video Shift Register (FFTSR).* Milpitas, CA, April 1990.

Magar, S., S. Shen, G. Luikuo, M. Fleming, and R. Aguilar. "An Application Specific DSP Chip Set for 100 MHz Data Rates." In *International Conference on Acoustics, Speech, and Signal Processing*, volume 4, pp. 1989–1992, April 1988.

Matsui, M. and J. B. Burr. "A Low-Voltage 32 × 32-Bit Multiplier in Dynamic Differential Logic." In *IEEE Symposium on Low Power Electronics*, pp. 34–35, October 1995.

Matsui, M., H. Hara, Y. Uetani, L.-S. Kim, T. Nagamatsu, Y. Watanabe, A. Chiba, K. Matsuda, and T. Sakurai. "A 200 MHz 13 mm^2 2-D DCT Macrocell Using Sense-Amplifying Pipeline Flip-Flop Scheme." *IEEE Journal of Solid-State Circuits*, vol. 29, no. 12, pp. 1482–1490, December 1994.

Mehta, H., R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. "Techniques for Low Energy Software." In *International Symposium on Low Power Electronics and Design*, pp. 72–75, August 1997.

Meng, T. H., B. M. Gordon, E. K. Tsern, and A. C. Hung. "Portable Video-on-Demand in Wireless Communication." *Proceedings of the IEEE*, vol. 83, no. 4, pp. 659–680, April 1995.

Mizuno, H. and T. Nagano. "Driving source-line cell architecture for sub-1-V high-speed low-power applications." *IEEE Journal of Solid-State Circuits*, vol. 31, no. 4, pp. 552–557, April 1996.

Nandakumar, M., A. Chatterjee, M. Rodder, and I.-C. Chen. "A Device Design Study of 0.25μm Gate Length CMOS for 1V Low Power Applications." In *IEEE Symposium on Low Power Electronics*, pp. 80–82, October 1995.

O'Brien, J., J. Mather, and B. Holland. "A 200 MIPS Single-Chip 1K FFT Processor." In *IEEE International Solid-State Circuits Conference*, volume 36, pp. 166–167, 327, 1989.

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

Rabiner, L. R. Private communication with author, 13 May 1997.

Rabiner, L. R. and B. Gold. *Theory and Application of Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

Richards, M. A. "On Hardware Implementation of the Split-Radix FFT." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 10, pp. 1575–1581, October 1988.

Roberts, R. A. and C. T. Mullis. *Digital Signal Processing*. Addison-Wesley, Reading, MA, 1987.

Ruetz, P. A. and M. M. Cai. "A Real Time FFT Chip Set: Architectural Issues." In *International Conference on Pattern Recognition*, volume 2, pp. 385–388, June 1990.

Runge, C. In *Zeit. für Math. und Physik*, volume 48, p. 443, 1903.

Runge, C. In *Zeit. für Math. und Physik*, volume 53, p. 117, 1905.

Runge, C. and H. König. "Die Grundlehren der Mathematischen Wissenschafter." In *Vorlesungen über Numerisches Rechnen*, volume 11, Berlin, 1924. Julius Springer.

Santoro, M. R. *Design and Clocking of VLSI Multipliers*. PhD thesis, Stanford University, Stanford, CA, October 1989.

Sarmiento, R., F. Tobajas, V. de Armas, R. Esper-Chaín, J. F. López, J. Montiel-Nelson, and A. Núñez. "A CORDIC Processor for FFT Computation and Its Implementation Using Gallium Arsenide Technology." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 1, pp. 18–30, March 1998.

Sharp Electronics Corporation. *LH9124 Digital Signal Processor User's Guide*. Camas, WA, 1992.

Singleton, R. C. "A Method for Computing the Fast Fourier Transform with Auxiliary Memory and Limited High-Speed Storage." In *IEEE Transactions on Audio and Electroacoustics*, volume AU-15, pp. 91–98, June 1967.

Singleton, R. C. "An Algorithm for Computing the Mixed Radix Fast Fourier Transform." In *IEEE Transactions on Audio and Electroacoustics*, volume AU-17, pp. 93–103, June 1969.

Smit, J. and J. A. Huisken. "On the energy complexity of the FFT." In Piquet, C. and W. Nebel, editors, *PATMOS '95*, pp. 119–132, Postfach 2541, 26015 Oldenburg, 1995. Bibliotheks- und Informationssystem der Universität Oldenburg.

Solomon, P. M. and D. J. Frank. "The Case for Reversible Computation." In *International Workshop on Low Power Design*, pp. 93–98, April 1994.

Sorensen, H. V., M. T. Heideman, and C. S. Burrus. "On Computing the Split-Radix FFT." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-34, no. 1, pp. 152–156, February 1986.

Stevens, K. S. and B. W. Suter. "A Mathematical Approach to a Low Power FFT Architecture." In *IEEE International Symposium on Circuits and Systems*, volume 2, pp. 21–24, June 1998.

Strum, R. D. and D. E. Kirk. *Discrete Systems and Digital Signal Processing*. Addison-Wesley, Reading, MA, 1989.

Su, C. L., C. Y. Tsui, and A. M. Despain. "Low power architecture design and compilation techniques for high-performance processors." In *IEEE COMPCON*, pp. 489–498, February 1994.

Sunada, G., J. Jin, M. Berzins, and T. Chen. "COBRA: An 1.2 Million Transistor Expandable Column FFT Chip." In *IEEE International Conference on Computer Design*, pp. 546–550, October 1994.

Tiwari, V., S. Malik, and A. Wolfe. "Compilation Techniques for Low Energy: An Overview." In *IEEE Symposium on Low Power Electronics*, volume 1, pp. 38–39, October 1994.

Tsern, E. K. and T. H. Meng. "A Low-Power Video-Rate Pyramid VQ Decoder." In *IEEE International Solid-State Circuits Conference*, volume 39, pp. 162–3, 436, 1996.

Tyler, G. L., B. M. Baas, F. Bauregger, S. Mitelman, I. Linscott, E. Post, O. Shana'a, and J. Twicken. "Radioscience Receiver Development for Low Power, Low Mass Uplink Missions." In *JPL's Planetary Instrumentation Definition and Development Program Workshop*, June 1997.

Van Loan, C. *Computational Frameworks for the Fast Fourier Transform.* Society for Industrial and Applied Mathematics, Philaelphia, PA, 1992.

Vetterli, M. and P. Duhamel. "Split-Radix Algorithms for Length-p^m DFT's." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 1, pp. 57–64, January 1989.

Vieri, C., I. Yang, A. Chandrakasan, and D. Antoniadis. "SOIAS: Dynamically Variable Threshold SOI with Active Substrate." In *IEEE Symposium on Low Power Electronics*, pp. 86–87, October 1995.

Waser, S. and M. Flynn. "Topics in Arithmetic for Digital Systems Designers." February 1990. Class notes for EE382 course at Stanford University.

Weste, N. and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, Reading, MA, 1985.

Wosnitza, M., M. Cavadini, M. Thaler, and G. Tröster. "A High Precision 1024-point FFT Processor for 2D Convolution." In *IEEE International Solid-State Circuits Conference*, volume 41, pp. 118–119, 424, 1998.

Yamauchi, H., H. Akamatsu, and T. Fujita. "An Asymptotically Zero Power Charge-Recycling Bus Architecture for Battery-Operated Ultrahigh Data Rate ULSI's." *IEEE Journal of Solid-State Circuits*, vol. 30, no. 4, pp. 423–431, April 1995.

Yamauchi, H., T. Iwata, H. Akamatsu, and A. Matsuzawa. "A 0.5V/100MHz Over-Vcc Grounded Data Storage (OVGS) SRAM Cell Architecture with Boosted Bit-line and Offset Source Over-Driving Scheme." In *International Symposium on Low Power Electronics and Design*, pp. 49–54, August 1996.

Yamauchi, H., T. Iwata, H. Akamatsu, and A. Matsuzawa. "A 0.5V Single Power Supply Operated High-Speed Boosted and Offset-Grounded Data Storage (BOGS) SRAM Cell Architecture." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 4, pp. 377–387, December 1997.

Yang, I. Y., C. Vieri, and D. Antoniadis. "Back-gated CMOS on SOIAS for dynamic threshold voltage control." *IEEE Transactions on Electron Devices*, vol. 44, no. 5, pp. 822–831, May 1997.

Index

Revision History

2004/09/29

- page xvii: Corrected W_N equation
- page 73: Removed extra b_k bits in table
- page 82: Fixed repeated-word typo