Implementation of a 2-D Fast Fourier Transform on an FPGA-based Computing Platform

by

Nabeel Shirazi

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

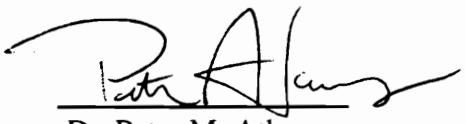
in partial fulfillment of the requirements for the degree of

Master of Science

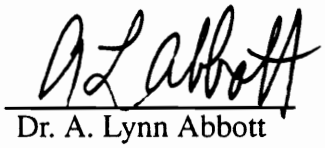
in

Electrical Engineering

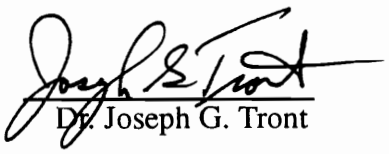
APPROVED:



Dr. Peter M. Athanas



Dr. A. Lynn Abbott



Dr. Joseph G. Tront

May, 1995
Blacksburg, Virginia

Implementation of a 2-D Fast Fourier Transform on an FPGA-based Computing Platform

by

Nabeel Shirazi

Committee Chairman: Dr. Peter Athanas

Electrical Engineering

(ABSTRACT)

The two dimensional fast Fourier transform (FFT) is a useful operation in many digital signal processing applications, but it is often avoided due to its large computational requirements. This thesis presents the implementation and performance figures for the fast Fourier transform on an FPGA-based custom computer. The computation of a 2-D FFT requires $O(N^2 \log_2 N)$ complex floating point arithmetic operations for an NxN image. By implementing the FFT algorithm on a custom computing machine (CCM) called Splash-2, a computation speed of at least 180 Mflops and a speed-up of 23 times over a SPARC-10 workstation is achieved.

Acknowledgments

I would like to thank Dr. Athanas for his technical advice and direction, and his constant support during the past two years. I wish to express my gratitude to Dr. Abbott and Dr. Tront for serving on my committee and for their guidance in this effort. I would also like to thank Dr. DeGroat of The Ohio State University for her technical advice in the preliminary stages of my thesis work.

Brad Fross, Al Walters, and Jim Peterson have been great friends and have always encouraged me to work hard on this research. In addition, I appreciate all the work Brad Fross has put into developing the hardware interface to Splash. I would like to thank the Virginia Tech Crew team for providing a healthy break from constantly working in the lab. I would most of all like to express my gratitude to my parents and my brother for their constant support and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Solutions to computational problems requiring very high performance often come in the form of Application Specific Integrated Circuits (ASICs). Special purpose machines built with ASICs provide better performance than general purpose machines of the same cost. Unfortunately, such solutions lack the flexibility to adapt to changing requirements, something that general purpose systems do well. Therefore, there exists a trade-off between cost, performance, and reuse for software programmable, general purpose computer solutions and special purpose hardware for certain classes of problems.

A relatively new technology that has shown great promise in filling this gap is the Field Programmable Gate Array (FPGA). The FPGA is essentially a large programmable integrated circuit composed of independent configurable logic modules that are interconnected with programmable routing channels. Although such devices have actually existed for almost ten years, recent advances in design tools as well as in the devices themselves have made their use in real systems practical. FPGAs have found their

way into many products from supercomputers to personal computers [1].

At the Supercomputing Research Center (SRC) in Bowie, Maryland, researchers are experimenting with the application of these programmable hardware devices in a system capable of high performance on a range of problems. Splash-2 is the name of the experimental machine which has been designed and is being built at SRC to demonstrate such a capability, using Xilinx 4000 series FPGAs as the programmable processing elements. Splash-2 is a multi-board system hosted on a Sun Sparc-2 workstation, which provides the control and data interface. Splash-2 is a follow-on to Splash-1 [2], the first experimental machine of this kind at SRC. Several applications were successfully demonstrated on Splash-1; however, hardware constraints, including I/O and internal interconnect bandwidth, and programming difficulties resulting from immature tools, limited its effectiveness.

1.1 Motivation

Image and digital signal processing (DSP) applications typically require high calculation throughput [3, 4]. The two-dimensional (2-D) fast Fourier transform (FFT) application presented here was implemented for near real-time filtering of video images on the Splash-2 FPGA-based custom computing machine (CCM). This application requires the ability to do floating point arithmetic. The use of floating point allows a large dynamic range of real numbers to be represented, and helps to alleviate the underflow and

overflow problems often seen in fixed point formats. An advantage of using a CCM for floating point implementation is the ability to customize the format and algorithm data flow to suit the application's needs.

The motivation behind implementing an FFT on Splash-2 is to show that although Splash-2 is a general purpose attached processor and was not designed to perform image processing, it can improve the performance of an FFT to the point where it is competitive with a DSP processor. An additional reason for implementing an FFT on Splash-2 is to show that floating-point arithmetic can be effectively performed on an FPGA-based processor.

The contributions of my research are:

- The implementation of floating point arithmetic on an FPGAs
- Showing an FPGA-based system can produce performance results similar to a typical DSP processor
- Demonstrating that the Splash-2 architecture is well-suited to implement an FFT.

1.2 Thesis Organization

An overview of the FFT algorithm and the method used to filter video images is given in Chapter 2. In addition, a discussion of other uses of the FFT and special purpose hardware that has been developed in order to calculate the FFT is included in Chapter 2. In Chapter 3, background information is provided on the Splash-2 architecture in order to

show the reasoning behind the method of partitioning the FFT between processing elements. A description of the floating point format and floating point addition and multiplication units used in the application is given in Chapter 4. The implementation of the 2-D FFT and the recirculation method used to feed data into a butterfly computational unit is shown in Chapter 5. In Chapter 6, error analysis is presented to show that the chosen floating point format used is adequate for this application. The performance of this implementation of an FFT was compared to a wide range of architectures in Chapter 7. Conclusions are given in Chapter 8.

Chapter 2

Background

This chapter presents background information regarding the FFT. Section 2.1 describes two uses of the FFT in image processing. Section 2.2 describes the discrete Fourier transform as the basis for the FFT. Section 2.3 discusses other hardware implementations for the FFT.

2.1 Applications of the Fast Fourier Transform in Image Processing

Two dimensional convolution using template-based window operations is a common way of filtering an image in the spatial domain if the template being used is relatively small (e.g., 8x8 pixels). Convolution using larger templates can be done much faster by converting an image in the spatial domain to the frequency domain and then applying a filter by doing point-by-point multiplication [5]. The filtered image in the frequency domain is then converted back to the spatial domain by performing an inverse DFT (IDFT).

An example of the discrete Fourier transform applied to images is shown in Figure 2.1. This illustrates the result of applying an exponential highpass filter. This filter is used to attenuate the low frequency components in order to perform edge detection. The black pixels of the filter in Figure 2.1 correspond to zero values and the white pixels correspond to values of one with the remaining gray pixels ranging between 0.0 and 1.0. The four corners of the images in the frequency domain are the locations of the low frequency components. The high-frequency components are distributed in the center of the image.

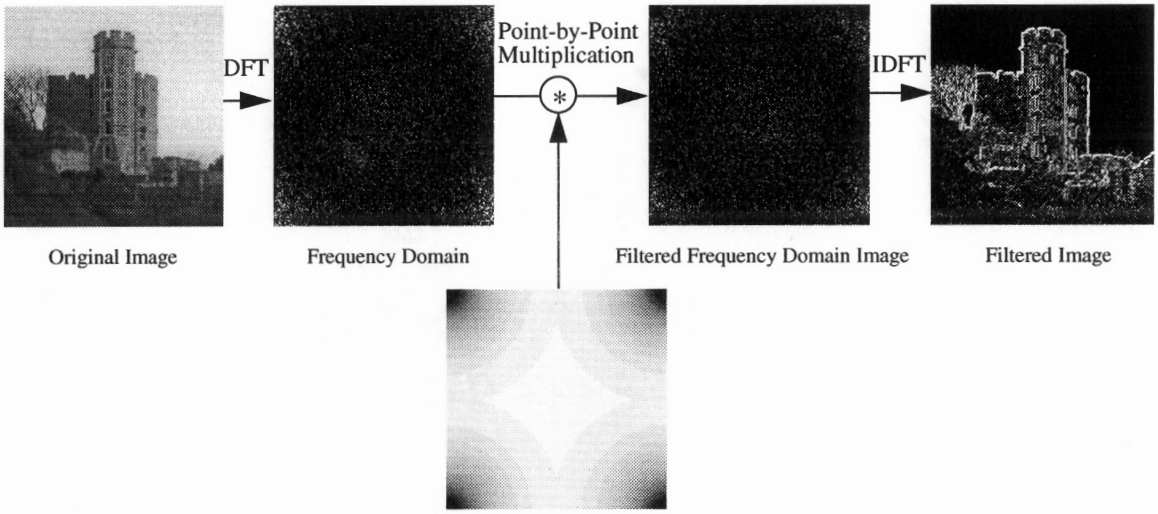


Figure 2.1: Fourier transform filtering method.

The Fourier transform can also be used to focus a video camera. If an image is in focus, its frequency domain representation will have a larger number of high-frequency components (higher intensity values are located more towards the center of the frequency domain image). In the frequency domain representation, a high intensity value is represented by a white pixel. The frequency representation of an image that is out of focus has

fewer sharp edges, and thus, lower high frequency spectral components. Most of its high intensity values are located near the corners of the image. For example, in Figure 2.2, two images were captured, one in focus, image (a), and one out of focus, image (c). The corresponding Fourier transform of each image was calculated and is also shown in Figure 2.2. The frequency domain representation of the focused image, image (b), has a larger number of pixels with high intensity values located more towards the center of the image than image (d). Using a feedback system, the focus of the camera can be changed until a maximum energy is achieved in the center of the frequency domain image. It will be shown in Chapter 7 that the calculation of a 128x128 FFT can be done in real time, i.e., in less that 1/30 of a second. By calculating an FFT on a 128x128 section of a video image, a video camera can be focused in a fraction of a second using this method.

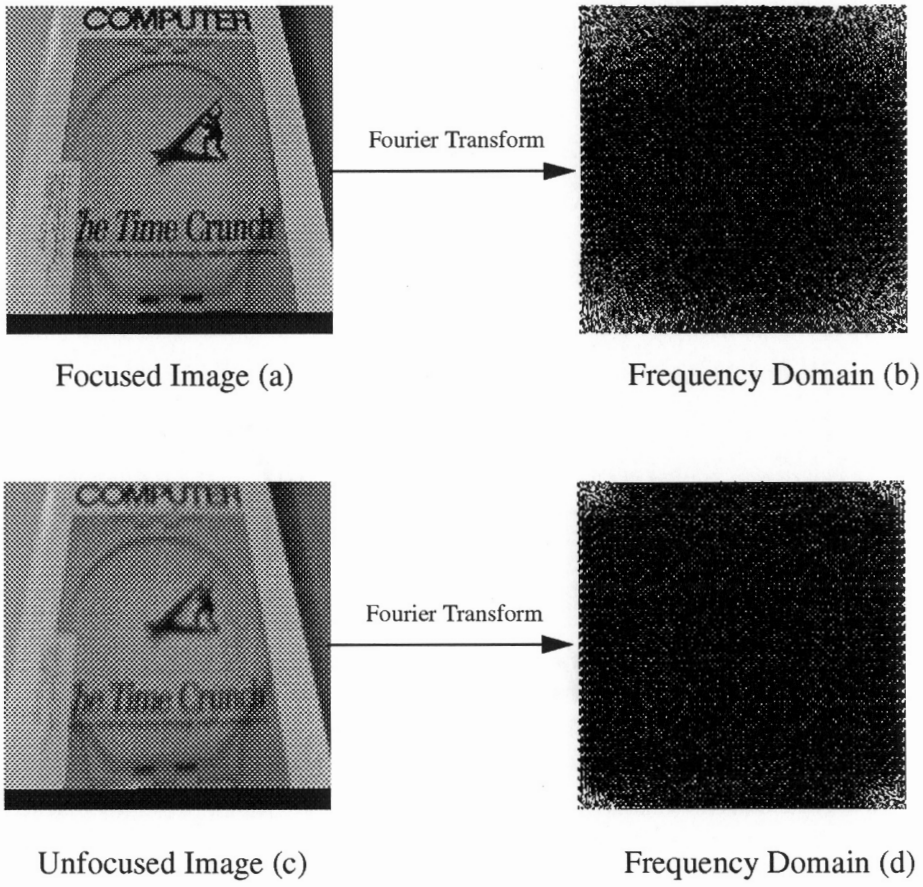


Figure 2.2: Fourier transform focusing example.

2.2 The Fourier Transform

This section describes the discrete Fourier transform (DFT) as a basis to describe the fast Fourier transform in Section 2.2.2.

2.2.1 Discrete Fourier Transform

The N-point DFT is defined as:

$$F(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)\, e^{-j2\Pi ux/N}$$

$$\text{for } u \in [0, N-1]$$

and is examined in detail in [5].

The 2-D DFT of a NxN image, *f(x,y)*, is defined by the expression,

$$F(u) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\, e^{-j2\Pi (xu+vy)/N}$$

$$\text{for } u, v \in [0, N-1]$$

The 2-D DFT expression can be decomposed into multiple 1-D Fourier transforms. The above equation can be expressed in the form:

$$F(u,v) = \frac{1}{N}\sum_{x=0}^{N-1} F(x,v)\, e^{-j2\Pi ux/N}$$

$$\text{for } u \in [0, N-1]$$

and,

$$F(x,v) = \frac{1}{N}\sum_{y=0}^{N-1} F(x,y)\, e^{-j2\Pi vy/N}$$

$$\text{for } x, v \in [0, N-1]$$

where $W^k_N = e^{-j2\pi x/N}$ or $e^{-j2\pi y/N}$ and is called the twiddle factor.

This shows that an NxN 2-D DFT can be computed by first performing N 1-D DFTs (one for each row), followed by another N 1-D DFTs (one for each column).

2.2.2 Fast Fourier Transform Algorithm

The FFT consists of a variety of tricks for reducing the computation time required to compute a DFT [4]. The number of complex multiplications and additions required to implement a 1-D DFT is proportional to N^2. The 1-D DFT computation can be decomposed so that the number of multiply and add operations is reduced to $O(N \log_2 N)$. The FFT algorithm achieves its computational efficiency through a divide-and-conquer strategy. The essential idea is a grouping of the time and frequency samples in such a way that the DFT summation over N values can be expressed as a combination of N/2 point DFTs. This process is repeated recursively until 2-point transforms remain. The two-point DFTs are called butterfly computation and involve computing one complex multiply and two complex additions. The notation used for a butterfly structure is shown in Figure 2.3.

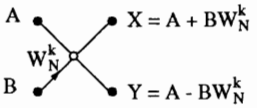


Figure 2.3: Radix-2 butterfly structure.

By using the FFT partitioning scheme, an eight point DFT can be computed as shown in Figure 2.4. Each stage of the N point FFT is composed of N/2 radix-2 butterflies. Since there are a total of $\log_2 N$ stages it follows that there are a total of $(N/2)\log_2 N$ butterfly structures per FFT. In addition, the input is in bit-reverse order and the output is in linear order. A NxN 2-D DFT can be decomposed into N^2 1-D FFTs in the same manner as a 2-D DFT.

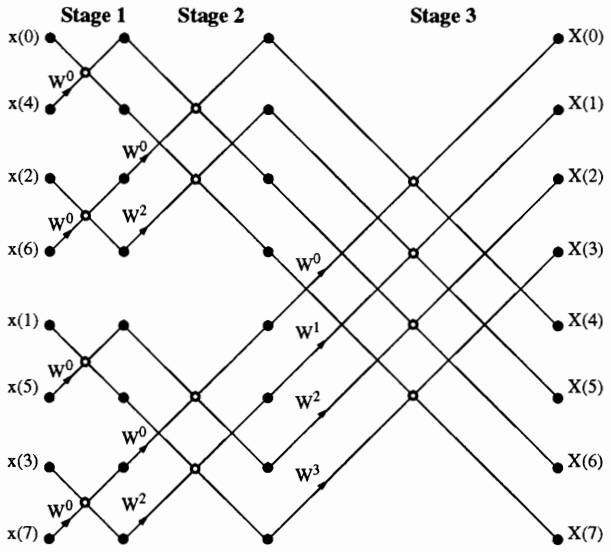


Figure 2.4: Eight point FFT.

2.3 Previous Hardware and Software Implementations of the Fourier Transform

Exploiting the inherent parallel processing abilities of dedicated hardware can lead to increased performance of the FFT algorithm. There are some general categories of parallelism that are suited to produce a more efficient implementation of the FFT. The first type of parallelism which can be used is time-overlapping of arithmetic and memory functions. The calculation of addresses of the data values and twiddle factors can be done in parallel with the butterfly operation. A second type of parallelism is to use distributed memory instead of a single global memory containing the real and imaginary components of the data values and twiddle factors. To perform a butterfly operation shown in Figure 2.3, six memory cycles are needed to fetch the real and imaginary components of the A and B input values and the twiddle factor. If distributed memory is used, these values could be fetched from their individual memories and the memory bottleneck would be avoided.

Higher radix structures could be used to increase the number of additions and multiplications done in a butterfly operation. For example, a radix-2 butterfly consists of a single complex multiplication and two complex additions, while a radix-4 butterfly consists of three complex multiplications and eight complex additions. Special purpose hardware could be designed to compute higher radix butterflies and would decrease the number of stages in the FFT flow graph to $\log_r N$ where r is the radix.

Pipelined FFTs is another method to increase performance. A pipelined structure

uses $\log_r N$ butterfly elements and can increase performance from 2 to 20 times[4]. For example, in the first stage of Figure 2.4, samples 0 and 4 followed by 2 and 6 could be processed and then the processing of stage two could be started without finishing the processing of stage one. Systolic implementations of a pipelined FFT processor have been implemented, however, they require larger amounts of hardware [6].

For high-speed computation of the FFT, hardware can be constructed with as many as $N/2 \log_2 N$ parallel butterfly units. For even a modest size N, this is a very large amount of hardware, however, it is sometimes needed in radar applications.

ASICs have been developed for the addressing circuitry needed to compute an FFT. A single IC developed by Advanced Micro Devices (AMD) can perform accesses to both data and twiddle factors[21]. With the addition of multiplier/accumulator chips, a processor array for two-dimensional Fourier transform has been constructed [22].

A procedure that involves fewer number of multiplications than the FFT was proposed and developed by S. Winograd and is known as the Winograd Fourier transform algorithm (WFTA) [23]. The DFT is decomposed into several short-length DFTs. With the WFTA approach the number of multiplications required for an N-point DFT is proportional to N rather than NlogN. Although this approach leads to algorithms that are optimal in terms of minimizing multiplications, the number of additions is significantly increased in comparison with the FFT. Therefore, the WFTA is most advantageous when multiplication is significantly slower than addition, as is often the case with fixed-point digital arithmetic. However, in processors where multiplication and accumulation are tied together, the Cooley-Tukey [4] algorithms are generally preferable. Although the WFTA

is extremely important as a benchmark for determining how efficient the DFT computation can be (in terms of multiplications), other factors often dominate in determining the speed and efficiency of a hardware or software implementation of the DFT computation.

Chapter 3

Splash-2 Architecture

The Splash-2 system is based on FPGA devices and has been used to improve the computational performance of a number of applications, including DNA pattern matching [7] and many image processing algorithms [8]. This chapter describes the Splash-2 architecture, the real-time video interface, and the method used to program Splash-2.

3.1 The Array Board

The Splash-2 custom computing machine (CCM) consists of up to 15 Splash-2 boards and an interface board contained in a standard 19" rack connected via cable to a Sparc-2 host [9]. Figure 3.1 shows the primary components of the architecture for a multi-board system. The Sparc-2 is responsible for loading the configurations into the Xilinx 4010 FPGAs [10], transmitting data to and from the Interface board and setting up the control registers on both the Splash-2 boards and the Interface boards. There are 16 Xilinx FPGAs on the board that are user programmable and where the majority of the

computations take place. These FPGAs are denoted by X1 through X16 in Figure 3.1. These Xilinx chips are interconnected in a linear array. The data path connecting all the Xilinx chips is 36 bits wide. This allows for 32-bit data plus a 4-bit tag to be transmitted from the host into the Splash-2 processor boards. Within the Splash-2 board, however, there is no restriction on the use of the 36 bits. The chips are also connected to a 16-way, 36-bit full crossbar switch, providing considerable communication flexibility. A 17th Xilinx chip, also user programmable, controls the configuration of the crossbar and provides broadcast capability. Each Xilinx chip is connected to a 256K by 16-bit static RAM across a 16-bit data bus. These RAMs can be accessed by the host (memory mapped) when the Splash-2 board is idle, or by the Xilinx chips at a maximum access rate of 40 MHz. The actual Splash-2 array board is shown in Figure 3.2.

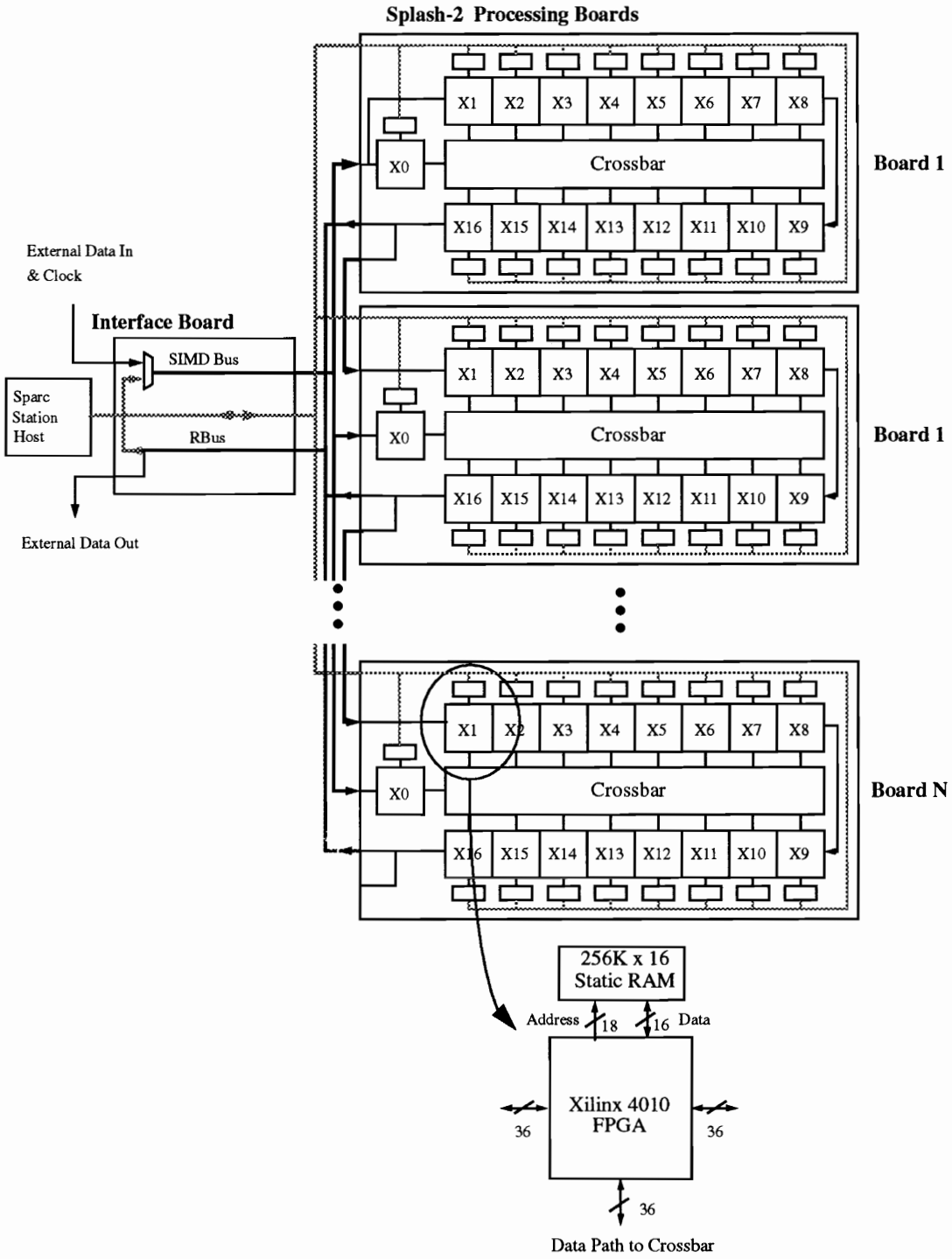


Figure 3.1: Splash-2 architecture.

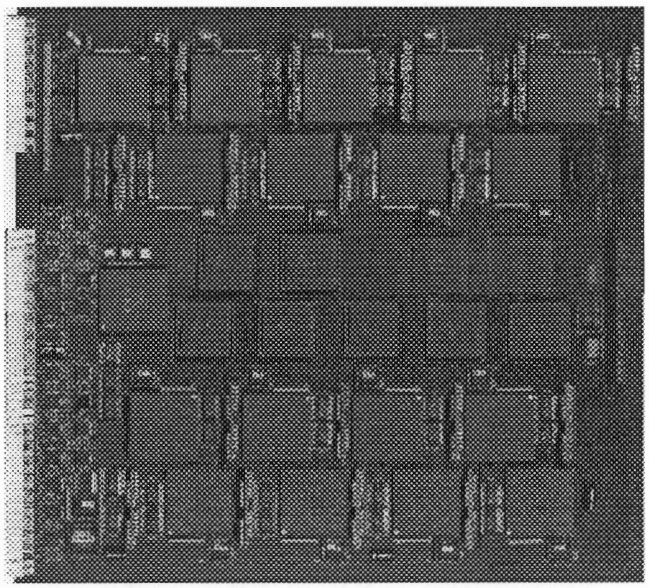


Figure 3.2: Splash-2 array board.
(Board Size: 16 1/2 x 17 1/4 inches)

3.2 Real-Time Video Interface

Figure 3.3 shows the image-processing system known as VTSplash that was developed at Virginia Polytechnic Institute and State University [8]. A VTSplash system is a modified version of the original Splash-2 system that incorporates real-time video processing hardware onto the interface board of the processing array.

A video camera produces an RS-170 analog video stream which is processed by a custom-built frame-grabber card. The frame-grabber card uses the Analog Devices 9502

A/D chip [11] that produces vertical and horizontal sync signals along with a stream of 8-bit gray scale values. A parallel interface was constructed between the frame-grabber board and the Splash-2 system in order to provide the sync signals, and 8-bit video data stream to the Splash-2 system at 10 MHz. The data stream is put in raster scan by the first two PEs of the Splash-2 array board. The real-time video data is processed by Splash-2 and the resulting output data stream is first presented to another custom board to re-buffer and format the data stream. The formatted data is then presented to a Data Translation (DT) 2867-LC image-acquisition/display card [12], which presents the image to a color video monitor. The DT card can overlay a colormap and other information on the real-time video data stream.

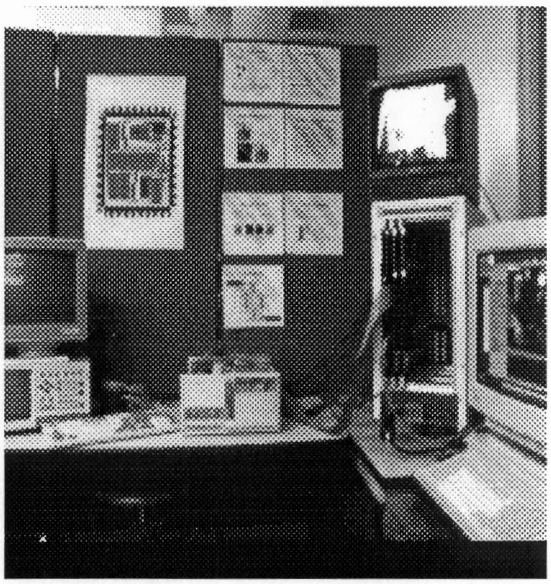


Figure 3.3: VTSplash system.

3.3 Programming Splash-2

Programming Splash-2 consists of specifying the configurations for the Xilinx chips on each Splash-2 board and for the two Xilinx chips on the interface board. Splash-2 programs are specified by using the hardware description language VHDL (VHSIC Hardware Description Language) [13]. A simulator for the complete Splash-2 system has been developed at SRC, allowing user-generated VHDL descriptions of the Xilinx chips to be simulated in the context of the full Splash-2 system. Digital logic synthesis tools are applied to these designs to generate gate-level descriptions, which are then placed and routed on the FPGAs using Xilinx-provided tools. A complete application also includes a front-end C program to configure and execute the hardware. A diagram of the programming process for Splash-2 is shown on Figure 3.4.

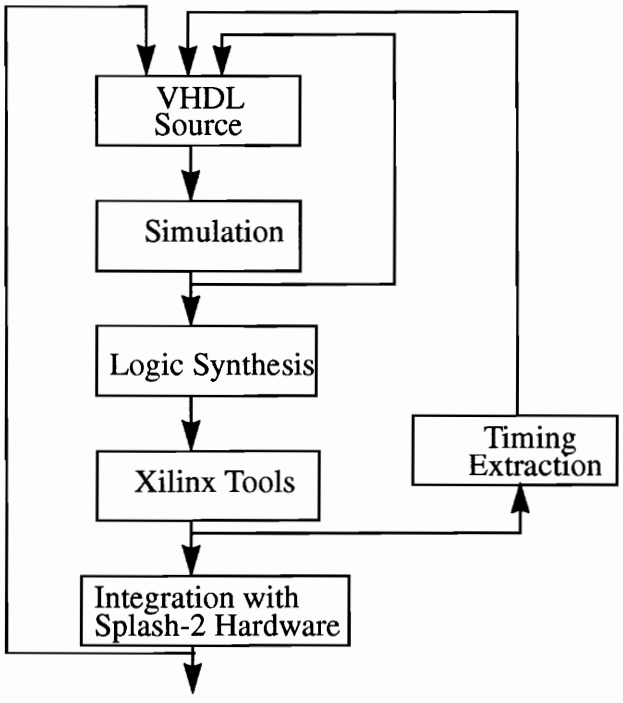


Figure 3.4: Splash-2 programming process.

Chapter 4

Floating Point Arithmetic on FPGAs

Until recently, any meaningful floating point arithmetic has been virtually impossible to implement on FPGA-based systems due to the limited density, routing resources and speed of older FPGAs. In addition, mapping difficulties occurred due to the inherent complexity of floating point arithmetic. With the introduction of high level languages such as VHDL, rapid prototyping of floating point units has become possible. Elaborate simulation and synthesis tools at a higher design level aid the designer in obtaining a more controllable and maintainable product. Although low level design specifications were alternately possible, the strategy used in the work presented here was to specify every aspect of the design in VHDL and rely on automated synthesis to generate the FPGA mapping.

Image and digital signal processing applications typically require high calculation throughput [3]. The fast Fourier transform and other signal processing techniques necessitate a large dynamic range of numbers [14]. The use of floating point helps to

alleviate the underflow and overflow problems often seen in fixed point formats. An advantage of using a CCM for floating point implementation is the ability to customize the format and algorithm data flow to suit the application's needs. In addition, multiple floating point units can be instantiated in order to exploit the parallelism of the CCM.

This chapter examines the implementations of various arithmetic operators, using a floating point format similar to the IEEE 754 standard [15]. Eighteen bit floating point adders/subtracters, multipliers, and division units have been synthesized for Xilinx 4010 FPGAs [10]. Sections 4.2, 4.3, and 4.4, present the algorithms, implementations, and optimizations used for the different operators. A summary, in terms of size and speed, of the different floating point units is given Section 4.5.

4.1 Floating Point Format Representation

The format which was used in the FFT application is similar to the IEEE 754 standard used to store floating point numbers. For comparison purposes, single precision floating point uses the 32-bit IEEE 754 format shown in Figure 4.1.

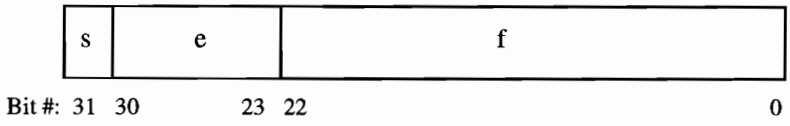


Figure 4.1: 32-bit floating point format.

The floating point value (v) is computed by:

$$v = (-1)^s \, 2^{(e - 127)}(1.f)$$

In Figure 1, the sign field, s, is bit 31 and is used to specify the sign of the number. Bits 30 down to 23 are the exponent field. This 8-bit quantity is a signed number represented by using a bias of 127. Bits 22 down to 0 are used to store the binary representation of the floating point number. The leading one in the mantissa, 1.f, does not appear in the representation, therefore the leading one is implicit. For example, -3.625 (dec) or -11.101 (binary) is stored in the following way:

$$v = (-1)^1 \, 2^{(128-127)} 1.1101$$

where: s = 1, e = 128 (dec) 80 (hex), and f = 680000 (hex).

Therefore -3.625 is stored as: C0680000 (hex).

The 18-bit floating point format was developed in the same manner for the 2-D fast Fourier transform application. The format was chosen to accommodate two specific requirements: (1) the dynamic range of the format needed to be quite large in order to represent very large and small, positive and negative real numbers accurately, and (2) the data path width into one of the Xilinx 4010 processors of Splash-2 is 36 bits wide and two operands were needed to be input on every clock cycle. Based on these requirements the format in Figure 4.2 was used.

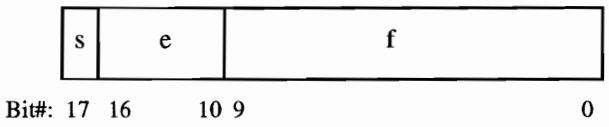


Figure 4.2: 18-bit floating point format.

The 18-bit floating point value (v) is computed by:

$$v = (-1)^s 2^{(e - 63)}(1.f)$$

The range of real numbers that this format can represent is +/- 3.6875 x 10^{19} to +/- 1.626 x 10^{-19}.

4.2 Floating Point Addition and Subtraction

The aim in developing a floating point adder/subtracter routine was to pipeline the unit in order to produce a result every clock cycle. By pipelining the adder the speed increased, however, the area increased as well. Different VHDL coding structures were tried in order to minimize size and increase speed.

4.2.1 Algorithm

The floating point addition and subtraction algorithm studied here is similar to what is done in most traditional processors, however, the computation is performed in three

stages. The notation s_i, e_i and f_i are used to represent the sign, exponent and mantissa fields of the floating point number, v_i, where i is operand 1 or 2. A block diagram of the three stage adder is shown in Figure 4.3.

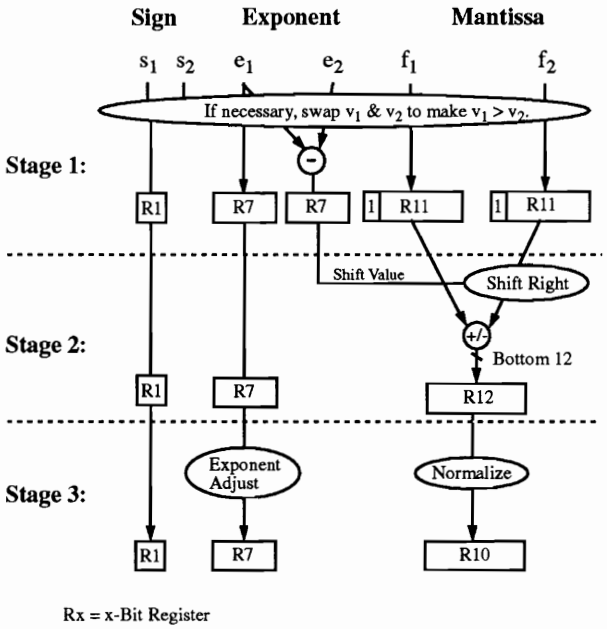


Figure 4.3: Three stage 18-bit floating point adder.

The computations required for each stage is as follows:

Stage 1:

- If the absolute value of v_1 is less than the absolute value of v_2 then swap v_1 and v_2. The absolute value is checked by comparing the exponents and mantissas of the two values.
- Subtract e_2 from e_1 in order to calculate the number of positions to shift f_2 to the right so that the decimal points are aligned before addition or subtraction in Stage 2.

Stage 2:

- Shift $1.f_2$ to the right $e_2 - e_1$ places calculated in the previous stage.
- Add $1.f_1$ to $1.f_2$ if s_1 equals s_2 else subtract $1.f_2$ from $1.f_1$.
- Set the sign and the exponent of the final result, v_3, to the sign and the exponent of the greater value v_1.

Stage 3:

- Normalize f_3 is by shifting it to the left until the high order bit is a one.
- Adjust the exponent of the result, e_3, by subtracting it from the number of positions that f_3 was shifted left.

4.2.2 Eighteen Bit Floating Point Addition Example

To demonstrate an 18-bit, three stage floating point adder , v_1 is added to v_2 to produce v_3 . Let $v_1 = 24.046875$ and $v_2 = -25.40625$; therefore v_3 should equal -1.359375.

	Decimal	Binary	18-Bit Format
v_1	24.046875	1.1000000011×2^4	0 1000011 1000000011
v_2	-25.40625	1.1001011010×2^4	1 1000011 1001011010

Therefore: $s_1 = 0\ e_1 = 1000011 \quad 1.f_1 = 1.1000000011$
$s_2 = 1\ e_2 = 1000011 \quad 1.f_2 = 1.1001011010$

Stage 1:

- Swap v_1 and v_2 since $e_1 = e_2$ and $f_2 > f_1$
  Now: $s_1 = 1\ e_1 = 1000011 \quad 1.f_1 = 1.1001011010$
  $s_2 = 0\ e_2 = 1000011 \quad 1.f_2 = 1.1000000011$
- $e_1 - e_2 = 0$ therefore $1.f_2$ does not need to be shifted in the next stage

Stage 2:

- $1.f_3 = 1.f_1 - 1.f_2$ since s_1 does not equal s_2.
- $s_3 = f_1$ and $e_3 = e_1$ since they are the sign and exponent of the greater value.
  After stage 2: $s_3 = 1\ e_3 = 1000011 \quad 1.f_3 = 0.0001010111$

Stage 3:

- Normalize f_3 by shifting it 5 places to the left.
- Adjust the Exponent, e_3, by subtracting 5 from it.
  After final stage: $s_3 = 1\ e_3 = 0111111 \quad 1.f_3 = 1.0101110000$

The result v_3 after addition:

	Decimal	Binary	18-Bit Format
v_3	-1.359375	1.010111×2^0	0 0111111 0101110000

4.2.3 Optimization

The circuits produced by contemporary VHDL synthesis tools are, unfortunately, highly sensitive as to how the original behavioral or structural description is expressed. When designing the floating point adder/subtracter it was found that using different VHDL constructs to describe the same behavior resulted in a faster and smaller design, explained below.

The parts of the adder which caused the bottleneck were the exponent subtracter, the mantissa adder/subtracter and the normalization unit. An 8-bit and a 16-bit Xilinx hard-macro adder/subtracter [10] was used in place of VHDL code written for the exponent and mantissa computation. This increased the overall speed of the design even though a smaller 12-bit adder/subtracter was replaced with a 16-bit adder/subtracter Xilinx hard macro [10]. The first cut at the normalization unit resulted in a very slow and large design. VHDL *for* loops were used for the shift left and for the code that finds the most significant bit during normalization. In order to decrease the size and increase the speed of the design, the *for* loops were unrolled and *if* statements used instead.

The first method used for shifting the mantissa of the second operand, f_2, a variable number of places was originally coded in VHDL the following way:

```
--Signal and Variable Declarations:
    signal f2:              Bit_Vector(15 downto 0);
    signal f2_result:       Bit_Vector(15 downto 0);
    signal e_diff:          Bit_Vector(7 downto 0);
    variable f2_var:        Bit_Vector(10 downto 0);
    variable e_diff_var:    Bit_Vector(7 downto 0);

-- VHDL Code:
-- Shift f2 right ediff places
e_diff_var := e_diff;
f2_var := f2(10 downto 0);

for i in 1 to 11 loop
    if (e_diff_var > zero_8) then
    f2_var(9 downto 0) := f2_var(10 downto 1);
    f2_var(10) := '0';
    e_diff_var := e_diff_var - 1;
    end if;
end loop;

f2_result(10 downto 0) <= f2_var;
```

Method two used *if* statements to check each individual bit of the shift value and shift f_2 accordingly.

```
--Signal and Variable Declarations:
    signal f2:              Bit_Vector(15 downto 0);
    signal f2_result:       Bit_Vector(15 downto 0);
    signal e_diff:          Bit_Vector(7 downto 0);
    variable f2_var:        Bit_Vector(10 downto 0);
    variable e_diff_var:    Bit_Vector(7 downto 0);

-- VHDL Code:
-- Shift f2 right ediff places
if ((e_diff(7) = '1') or (e_diff(6) = '1') or  (e_diff(5) = '1') or (e_diff(4) = '1'))
   then
    e_diff_var(3 downto 0) := "1111";
else
    e_diff_var(3 downto 0) := e_diff(3 downto 0);
end if;

    -- Sequential Code for shifting f2_var
f2_var := f2(10 downto 0);
if (e_diff_var(0) = '1') then
    f2_var(9 downto 0) := f2_var(10 downto 1);
    f2_var(10) := '0';
end if;
if (e_diff_var(1) = '1') then
    f2_var(8 downto 0) := f2_var(10 downto 2);
    f2_var(10 downto 9) := "00";
end if;
if (e_diff_var(2) = '1') then
    f2_var(6 downto 0) := f2_var(10 downto 4);
    f2_var(10 downto 7) := "0000";
end if;
if (e_diff_var(3) = '1') then
    f2_var(2 downto 0) := f2_var(10 downto 8);
    f2_var(10 downto 3) := "00000000";
end if;

f2_result(10 downto 0) <= f2_var;
```

The result of using the second method is shown in Table 1. The variable 11-bit shifter became two times smaller and three times faster. The Xilinx timing tool, *xdelay*, was used to estimate the speed of the designs.

Table 4.1: Optimizations for variable 11-bit barrel shifter.

	Method 1	**Method 2**	**Advantage**
FG Function Generators (used/available)	85/800 10%	44/800 5%	2x smaller
Flip Flops	6%	6%	same
Speed	6.5 MHz	19.0 MHz	2.9x faster

The code used to normalize the result after addition or subtraction of f_1 and f_2 was also initially written using *for* loops in VHDL.

```
--Signal and Variable Declarations:
    signal f2:              Bit_Vector(15 downto 0);
    signal f2_result:       Bit_Vector(15 downto 0);
    signal e_diff:          Bit_Vector(7 downto 0);
    variable f2_var:        Bit_Vector(10 downto 0);
    variable e_diff_var:    Bit_Vector(7 downto 0);

-- VHDL Code:
-- Shift f_result left until msb = 1
msb := f(10);
f_result_var := f;
e_result_var := e;

for i in 1 to 11 loop
    if (msb = '0') then
        f_result_var(10 downto 1) := f_result_var(9 downto 0);
        f_result_var(0) := '0';
        e_result_var := e_result_var - 1;
        msb := f_result_var(10);
    end if;
end loop;

f_result <= f_result_var(9 downto 0);
e_result <= e_result_var;
```

The second method calculates the number of places to shift the mantissa left in order to position the most significant bit (msb) in the high order location. A series of *if* statements

are used to check all possible bit position locations for the msb in order to calculate the shift value. After the shift value is calculated, a procedure similar to the second method to perform a variable shift to the right is used to shift the un-normalized value the correct number of positions to the left in order to normalize it. Due to the size of the VHDL source code it is not listed here. Instead, a complete listing of the VHDL code used to implement a floating point adder, including the code for this second method of implementing the normalization unit, is given in Appendix A.

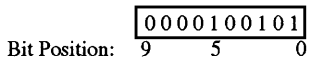
The individual bits of the shift value are calculated by checking each position of the un-normalized mantissa for a left most '1.' Pseudo code used to calculate each bit of the shift value is based on Table 4.2. The pseudo code is as follows:

Table 4.2: Truth table used to find the position of the MSB.

MSB Bit Pos.	9	8	7	6	5	4	3	2	1	0
# of Shift Lefts	1	2	3	4	5	6	7	8	9	10
Shift_Val (0)	1	0	1	0	1	0	1	0	1	0
Shift_Val (1)	0	1	1	0	0	1	1	0	0	1
Shift_Val (2)	0	0	0	1	1	1	1	0	0	0
Shift_Val (3)	0	0	0	0	0	0	0	1	1	1

Shift_Val(0) = (1) and not (2 through 9) or
(3) and not (4 through 9) or
(5) and not (6 through 9) or
(7) and not (8 or 9) or
(9)
Shift_Val(1) = (0) and not (1 through 9) or
(3 or 4) and not (5 through 9) or
(7 or 8) and not (9)
Shift_Val(2) = (3 or 4 or 5 or 6) and not (7 through 9)
Shift_Val(3) = (0 or 1 or 2) and not (3 through 9)

For example, if the un-normalized value is:



Shift_Val(0) = 1 since bit (5) is a one and (6 through 9) are zero. Shift_Val(1) = 0 because all cases for Shift_Val(1) fail. Shift_Val(2) = 1 since bit (5) is set and bits (7 through 9) are not. Shift_Val(3) = 0 since bits 3 through 9 are not all zero. Therefore the shift left value for the un-normalized number is 5.

By using this second method, the normalization unit became 2.9 times smaller and 2.6 times faster. A summary of the result of optimizing the normalization unit is shown in Table 4.3.

Table 4.3: Optimizations for normalization unit.

	Method 1	**Method 2**	**Advantage**
FG Function Generators	167/800 20%	58/800 7%	2.9x smaller
Flip Flops	6%	6%	same
Speed	5.1 MHz	13.4MHz	2.6x faster

The overall size and speed of the 18-bit floating point adder is given in Section 4.2.5.

4.3 Floating Point Multiplication

Floating point multiplication is much like integer multiplication. Because floating point numbers are stored in sign-magnitude form, the multiplier needs only to deal with unsigned integer numbers. Like the architecture of the floating point adder, the floating point multiplier unit is a three stage pipeline that produces a result on every clock cycle. The bottleneck of this design was due to its large propagation delay of the integer multiplier. Four different methods were used to optimize the integer multiplier in order to meet speed and size requirements.

4.3.1 Algorithm

The block diagram for the three stage 18-bit floating point multiplier is shown in Figure 4.4.

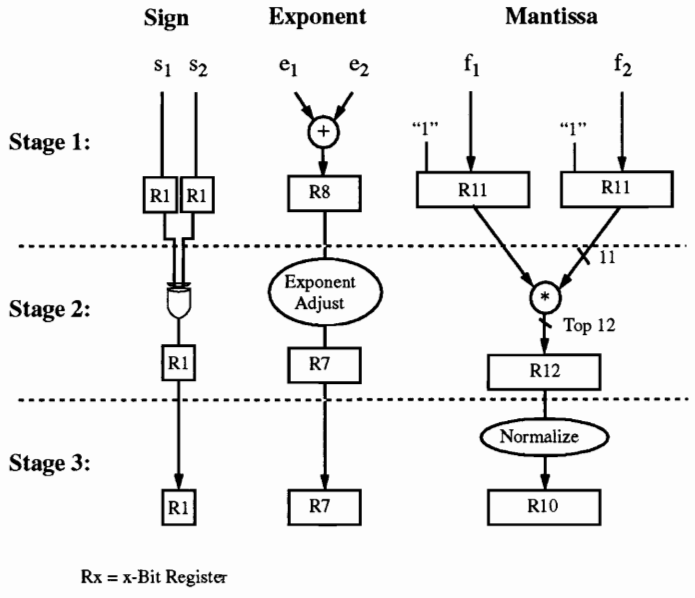


Figure 4.4: Three stage 18-bit floating point multiplier.

The algorithm for each stage is as follows:

Stage 1:

- The exponents, e_1 and e_2 are added and the result along with the carry bit is stored in an 8-bit register. If the addition of two negative exponents results in a value smaller than the minimum exponent that can be represented, i.e. -63, underflow occurs. In this case the floating point number is set to zero. If overflow occurs, the result is set to the maximum number the format can represent.
- If the floating point number is not zero, the implied one (1) is concatenated to the left side of the f_1 and f_2 terms.
- The sign bits are registered in this stage.

Stage 2:

- Integer multiplication of the two 11-bit quantities, $1.f_2$ and $1.f_1$, is performed. The top 12 bits of the 22-bit result is stored in a register.
- The exponent is adjusted depending on the high order bit of the multiplication result.
- The sign bits of the two operands are compared and if they are equal to each other the result is assigned a positive sign, if they differ the result is negative.

Stage 3:

- Normalization of the resulting mantissa is performed.
- The resulting sign, exponent and mantissa fields are placed into an 18-bit floating point word.

4.3.2 Optimization

Four different methods were considered in optimizing the 11-bit integer multiplier. The first method used the integer multiply available in the Synopsys 3.0a VHDL compiler. The second method was a simple array multiplier composed of ten 11-bit carry-save adders [16]. The last two methods involved pipelining the multiplier in order to increase the speed of the design. The multiplication of the two 11-bit quantities were broken up and multiplied in the following way:

```
          X6 X5
      *   Y6 Y5
          X5Y5
        X6Y5
        X5Y6
  +   X6Y6
  22-Bit Result
```

In Method 3, the first stage of the multiplier was the multiplication of the four terms X5Y5, X6Y5, X5Y6, and X6 Y6. The second stage involved adding the results of the four multiplications. In Method 4, two stages were used to sum the multiplication terms.

The results of the four methods are summarized in Table 4.4. The advantage in terms of the number of times faster and the number of times larger than Method 1 is shown.

Table 4.4: Results from optimizing an integer 11-bit multiplier.

	Method 1	Method 2	Method 3	Method 4
FG Function Generators	35%	31%	45%	47%
Stages	1	1	2	3
Speed	4.9 MHz	3.7 MHz	6.2 MHz	9.4 MHz
Area Advantage	1.0	0.90	1.29	1.34
Speed Advantage	1.0	0.75	1.24	1.92

Method 1 was used in the floating point multiplier unit since the size of the unit was too large using Methods 3 or 4 to allow an additional floating point unit in the same chip. The overall size and speed of the 16 and 18-bit floating point multipliers are given in Section 4.2.5. A complete listing of the VHDL code used to implement the floating point multiplier is given in Appendix B.

4.4 Floating Point Division

A floating point division technique is presented here which utilizes the pipelined multiplier discussed earlier. Division can be done by using the reciprocal of the divisor value so that the equation for division becomes a multiplication of (A x ($1/B$) = Q). Independent operations on the different floating point fields enable the design to be pipelined easily.

4.4.1 Algorithm

The reciprocal of a floating point value can be accomplished in two steps: (1) reciprocate the mantissa value by using a look-up table, and (2) negate the power of the base value. Since floating point representation already has the fields segregated, the task becomes trivial for a processing element which is complemented by a memory bank of at least 2^n x n bits, where n is the size of the mantissa's normalized binary representation. Local memories to the processing elements store the reciprocal of each bit combination of the mantissa.

In order to pipeline the design, three steps prior to the multiplication are necessary: (1) extract the mantissa from the input as the memory address and negate the exponent, (2) provide a delay until the memory data is valid, and (3) insert the new mantissa. The data word created during Stage 3 is passed to the multiplier. The second stage of the pipeline depends on the system being used which could result in longer delays before the data is made available from memory. In this case, a Splash-2 implementation was used which is shown in Figure 4.5. Memory reads require a single cycle after the address is presented before the data can be acquired from the memory data buffer.

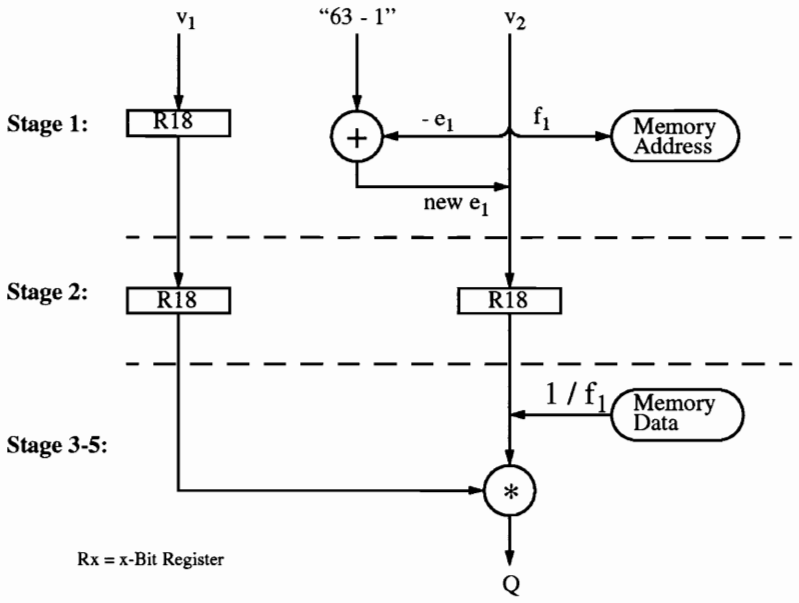


Figure 4.5: Five stage 18-bit floating point divider

The k_1 value negates the exponent, which still retains the embedded excess value. Note that since all the values reciprocated will be less than or equal to 1.0, normalization for the mantissa has to be done, though the special case for 1.0 has to be made. The normalization process is done automatically with k_1. Once the addition is done, the result becomes the new exponent passed onto Stage 2. The mantissa in Stage 1 directly goes to the memory address buffer to obtain the new mantissa, but the old mantissa continues into Stage 2 and is replaced in Stage 3. Stage 2 of the pipeline waits for the data to become available from the memory, which occurs at Stage 3. The new mantissa is inserted into the final operand to be passed to the multiplier. Although three pipeline stages are shown here, additional stages occur due to the pipelined multiplier to make a total of five stages.

4.5 Summary of Floating Point Arithmetic

The aim in designing the floating point units was to pipeline each unit a sufficient number of times in order to maximize speed and to minimize area. It is important to note that once the pipeline is full, a result is output every clock cycle. A summary of the resulting size and speed of the 18-bit floating point unit is given in Table 4.5.

Table 4.5: Summary of 18-bit floating point units.

	Adder/ Subtracter	Multiplier	Divider
FG Function Generators	28%	44%	46%
Flip Flops	14%	14%	34%
Stages	3	3	5
Speed	8.6 MHz	4.9 MHz	4.9 MHz
Tested Speed	10 MHz	10 MHz	10 MHz

The Synopsys Version 3.0a VHDL compiler was used along with the Xilinx 5.0 tools to compile the VHDL description of the floating point arithmetic units. The Xilinx timing tool, *xdelay*, was used to estimate the speed of the designs.

Each of the floating point arithmetic units, with exception of the floating point divider, has been incorporated into the FFT application. All of the floating point units operate at 10 MHz and have been tested by taking the result of the arithmetic operation and storing it into the memory on the Splash-2 array board. The results were checked by doing the same operation on a Sparc workstation using the same 18-bit format.

Chapter 5

FFT Implementation

Implementing filtering methods discussed in Section 2.1 involved mapping a 2-D FFT, a filter, and a 2-D inverse fast Fourier transform (IFFT) to a two-board Splash-2 system. The filtering method was constructed in such a way that the FFT and the IFFT were computed in parallel and were continuously provided video images from a frame buffer. This section discusses the recirculation method used to implement the FFT, the butterfly operator used in the FFT, and the filtering process.

5.1 FFT Recirculation Method

To calculate a 2-D FFT, a method involving recirculating data through a butterfly operator was implemented. A block diagram of this method is shown in Figure 5.1. Two banks of memory are used to store the input and output data of each stage of the FFT. A bank of memory consists of three processing elements that store the real and imaginary

components of the two 18-bit floating point numbers into their local memories. Since the local memories are only 16 bits, the two 18-bit floating point values are divided between the three memories.

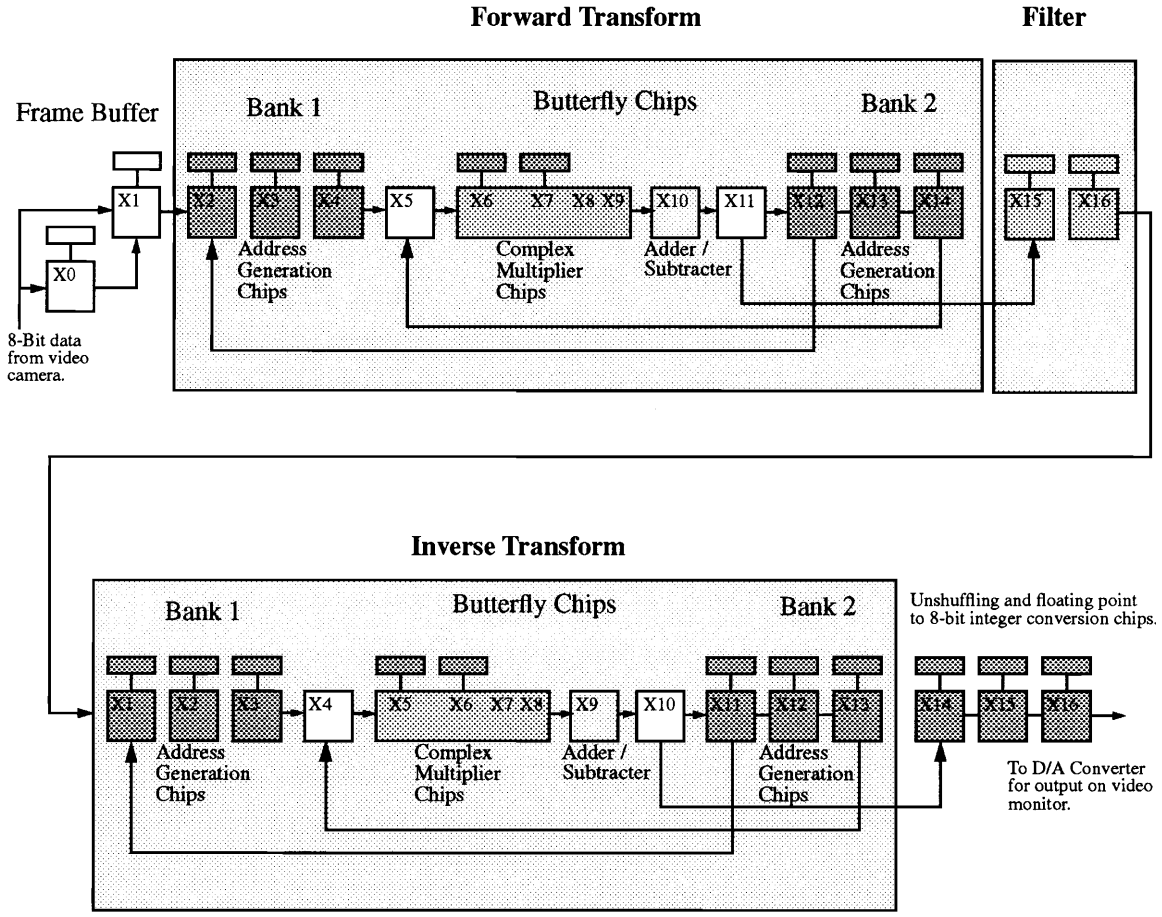


Figure 5.1: Splash-2 FFT image filtering method.

To compute an FFT on an input image, a frame of data is accepted from the frame buffer, converted from 8-bit integer values to 18-bit floating point values, and stored into Bank 1.

The 2-D FFT is computed by first computing a 1-D FFT of each row of the image and then a 1-D FFT on each column of the row transforms. The 1-D FFT is computed in the same manner as shown in Figure 2.3. The first stage of the FFT is computed by reading each row of data points in bit-reversed order from Bank 1 and passing it to the butterfly operation. The results of the butterfly operation are stored in the second bank of memory. Once each butterfly is computed in the first stage, the second stage is computed by reading the data out of the second bank of memory in linear order and into the butterfly operator. The results of this stage are stored in Bank 1. The recirculation method is continued by reading data out of one bank of memory while the other bank of memory is storing the results. The recirculation terminates when a 1-D FFT is calculated on each row of the image. The second set of 1-D FFTs are computed in the same manner except it is done on each column of the result of the first set of 1-D FFTs. Once the final stage of the last FFT is calculated, the data is passed over the crossbar from X11 to X15 where the data is filtered. The complete 2-D FFT process involves $2N^2 \log_2 N$ passes through the butterfly operator.

An IFFT is performed on the filtered data in the same manner. Each pixel of the new image is converted from floating point representation to integer values and is divided by N^2. The division by N^2 was performed by subtracting $2\log_2(N)$ from the exponent of the floating point value since N is a power of two. An in-place FFT algorithm [4] was not used since a read modify write procedure would needed to been used which would have decreased performance by a factor of two.

5.1.1 Addressing of Data Points and Twiddle Factors

Six counters were used to compute the address value of the A and B inputs of the butterfly operator. In addition to the counters, bit-reversing hardware and barrel shifters were used to concatenate the address correctly into a 18-bit value.

The 2-D input data is stored in raster scan order. The data is addressed by using *row* and *column* counters that are concatenated together to make the complete 18-bit address. A 2-D FFT is computed by doing N, 1-D FFTs of the rows of the input data followed by N, 1-D FFTs on the columns of the resulting row transformations. During the row transformations, the *row* counter is used to address the input data values for the FFT and the *column* counter is incremented after each N point 1-D FFT. During the column transformations, the *column* counter is used to address the input data values for the FFT and the *row* counter is incremented after each N point 1-D FFT.

During a 1-D transformation, the address of the A and B inputs of the butterfly operation is calculated using the following method. The inputs to the butterflies of the first stage of the FFT are in bit-reversed order. Three counters are used to calculate the addresses of the A and B inputs. The first counter, *stage*, is used to keep track of which stage of the FFT is being computed. In the *m*th stage of the FFT the butterfly inputs are separated by 2^{m-1}. The second counter, *stride*, is used to calculate the separation of the butterfly inputs. The third counter, *temp_cnt*, is a temporary counter used to determine

when a new value of stride should be calculated. The value of the counters are computed in the following way:

- *stage* = *stage* + 1 after N/2 butterfly computations.
- *stride* = *stride* * 2 when stage is incremented.
- *temp_cnt* = *temp_cnt* + 1, however, if the new value of *temp_cnt* = *stride* then *temp_cnt* = 1.
- if *temp_cnt* = *stride* then
  A_address = *A_address*+ *stride* + 1
  else
  A_address = *A_address* + 1.
- *B_address* = *A_address* + *stride*.

For example, Table 5.1 lists the address values for computing the A and B inputs of an 8 point FFT shown in Figure 5.2.

The twiddle factors are powers of $W_N^{(N/2^m)}$ in the *m*th stage of the FFT and are required in normal order if computation of butterflies begins at the top of the flow graph of Figure 5.2. The address of the twiddle factor is calculated in the Butterfly Chips in the following way:

- *W_address* = (*W_address* + *W_Index)* mod N/2.
- *W_Index* = N shifted to the left *stage* places.

Table 5.1: Addressing example.

Butterfly Number	stage	stride	temp_cnt	A_address	B_address	W_Index	W_address
1	1	1	1	0*	4*	4	0
2	1	1	1	2*	6*	4	0
3	1	1	1	1*	5*	4	0
4	1	1	1	3*	7*	4	0
5	2	2	1	0	2	2	0
6	2	2	2	1	3	2	2
7	2	2	1	4	6	2	0
8	2	2	2	5	7	2	2
9	3	4	1	0	4	1	0
10	3	4	2	1	5	1	1
11	3	4	3	2	6	1	2
12	3	4	4	3	7	1	3

* Bit Reversed Order

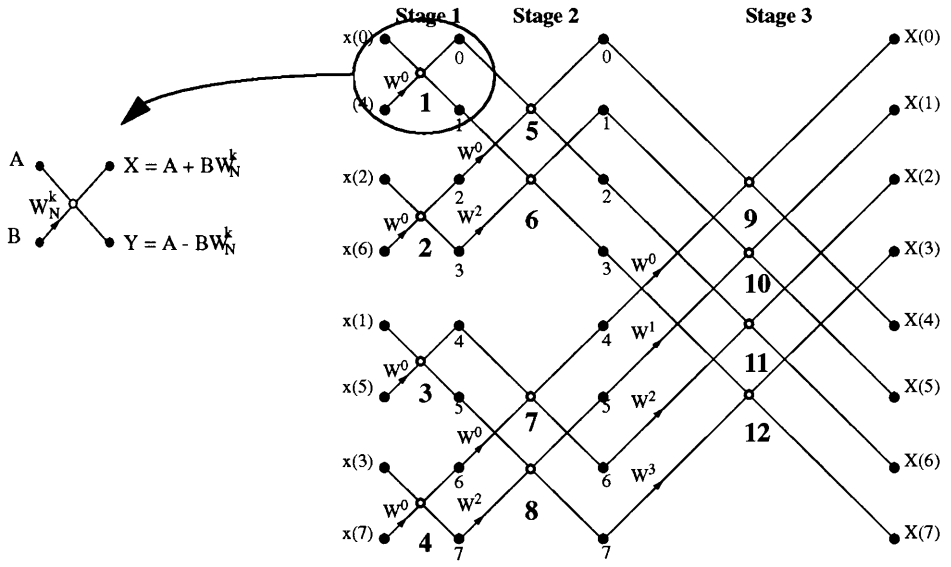


Figure 5.2: Annotated eight point FFT.

5.2 Butterfly Implementation

The butterfly operation is the heart of the FFT algorithm. It is pipelined in order to compute a real and complex result every clock cycle. The butterfly diagram shown in Figure 2.2 involves calculating a complex floating point multiplication and two floating point additions/subtractions. The complex multiply requires four multiplications and two additions/subtractions. In total, eight floating point operations are calculated every clock cycle at 10 MHz. The throughput of the butterfly operation is therefore 80 Mflops.

Figure 5.3, shows a block diagram of how the butterfly operation was partitioned between five processing elements on Splash-2. The real and imaginary parts of the complex multiplication of BW^k_N is given respectively by the equations:

$$BW^k_N.re = B.re\ W^k_N.re + B.im\ W^k_N.im$$
$$BW^k_N.im = B.re\ W^k_N.im - B.im\ W^k_N.re$$

Both the A and B inputs of the butterfly operation shown in Figure 2.3 are denoted in Figure 5.3 as f(x). The A value is inserted into the pipeline followed by the B value on the next clock cycle. The A input is not multiplied by the twiddle factor. In order to pass the A value through the pipeline without changing its value, multiplexers are used to multiply it by one and zero is added to it.

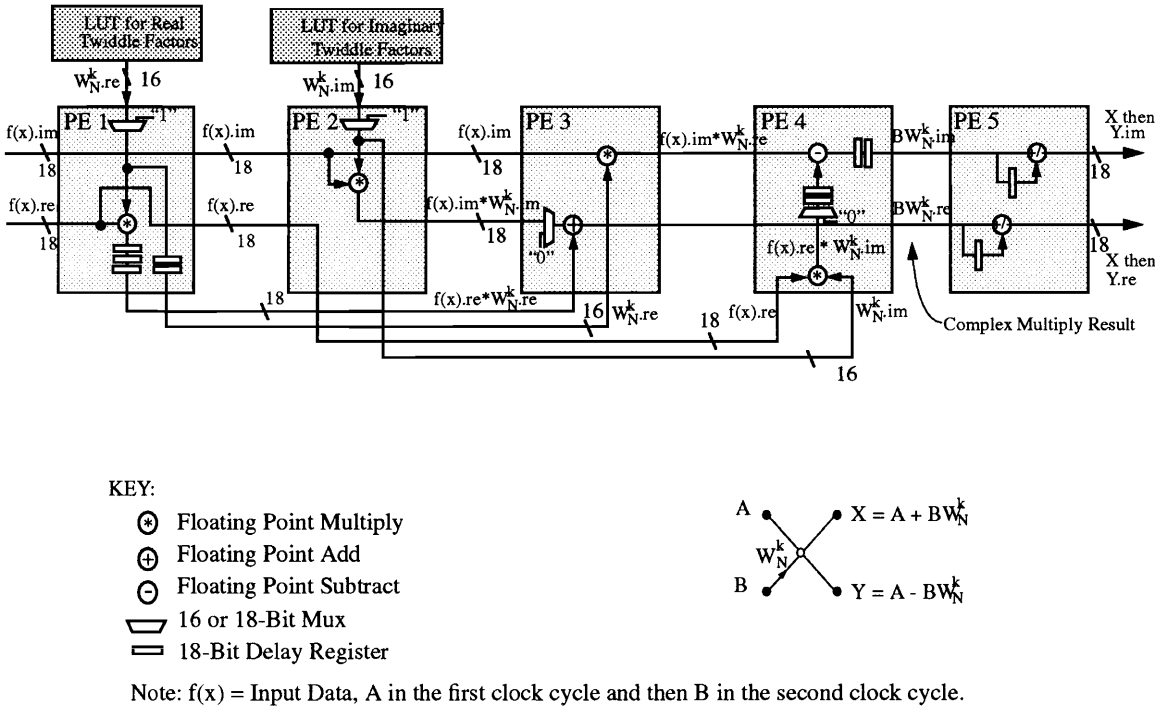


Figure 5.3: Block diagram of a five PE Splash-2 design for a butterfly operation.

When the real and imaginary values of B are inserted into the pipeline, these go through four processing elements in order to calculate the complex multiply of BW^k_N. The first processing element (PE 1) reads the real component of the appropriate twiddle factor and multiplies it by the real component of B. The result and the twiddle factor is passed via the crossbar to PE 3 and the real and imaginary components of B is passed to PE 2. The second PE multiplies the imaginary component of B with the appropriate twiddle factor and the result is passed to the third PE. The third PE reads the result from PE 1

(B.reW^k_N.re) off the crossbar and adds it to the result from PE 2 (B.imW^k_N.im) to produce the final result of the real component of the complex multiply (BW^k_N.re). The imaginary component of the complex multiply, BW^k_N.im, is computed in the same manner in PE 3 and PE 4. The butterfly operation is completed by adding A to BW^k_N in the first clock cycle to produce X and subtracting BW^k_N from A in the second clock cycle to produce Y.

The 18-bit format was not used to store the twiddle factors in the local memories of PE 1 and 2 since the memory data bus width is only 16 bits wide. A smaller 16-bit format was created by decreasing the exponent field of the 18-bit floating point number by 2 bits. Since twiddle factors can be expressed in terms of sine and cosine functions by using Euler's rule, the value of the floating point number will never have an exponent greater than 0 because the value will always be less than or equal to one. Because of this fact, the exponent field was changed to range from 0 to -31 instead of 63 to -63 in order to decrease the size of the exponent field from 7 bits to 5 bits. When the twiddle factor is read into the processing element, a conversion is done from the 16-bit format to the 18-bit format used in the arithmetic units.

5.3 Filtering

Once the input image is transformed to the frequency domain, point-by-point multiplication of the real matrix filter coefficients, H(u,v), with the transformed image can be computed in order to filter the image. The values of the elements of the matrix H(u,v), range between 0 and 1.0 and are stored in the local memories in the same manner as the twiddle factors of the butterfly operation. The filter coefficients are calculated before run-time and are stored in the local memories of PEs X15 and X16 as indicated in Figure 5.1. Filter chips consist of a floating point multiplier unit and addressing logic. X15 and X16 are used to filter, respectively, the real and imaginary components of the transformed image.

5.3.1 Filter Examples

Many different types of filters have been calculated such as ideal, Butterworth, exponential and trapezoidal filters [5]. These filters can be down-loaded on-the-fly from the Sparc-2 host to the local memories of the Splash-2 board in approximately 400ms.

Figure 5.4 shows the result of applying a Butterworth, exponential and ideal low-pass filters. The transfer functions for these lowpass filters are as follows:

Butterworth: $$H(u,v) = \frac{1}{1 + 0.414\,[D(u,v)/D_0]^{2n}}$$

Exponential: $$H(u,v) = e^{-0.347\,[D(u,v)/D_0]}$$

Ideal: $$H(u,v) = \begin{cases} 1 & \text{if } D(u,v) \le D_0 \\ 0 & \text{if } D(u,v) > D_0 \end{cases}$$

where: $D(u,v) = (u_2 + v_2)^{1/2}$, D_0 = the radius of the filter, and n is the order of the filter.

In Figure 5.4, a "ringing" effect can be noticed in the image filtered with the ideal filter [5]. This is due to the sharp transition between 1 and 0 in the filter. The images filtered with Butterworth and exponential filters do not exhibit this effect because of the gradual decrease in filter values.

Figure 5.5 shows the result of applying a Butterworth, exponential and ideal high-pass filters. The transfer functions for these highpass filters are as follows:

Butterworth: $$H(u,v) = \frac{1}{1 + 0.414\,[D_0/D(u,v)]^{2n}}$$

Exponential: $$H(u,v) = e^{-0.347\,[D_0/D(u,v)]}$$

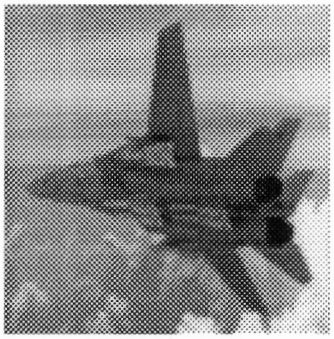
Ideal: $$H(u,v) = \begin{cases} 0 & \text{if } D(u,v) \le D_0 \\ 1 & \text{if } D(u,v) > D_0 \end{cases}$$

where: $D(u,v) = (u_2 + v_2)^{1/2}$, D_0 = the radius of the filter, and n is the order of the filter.

In addition to the previously discussed frequency domain filters, any NxN template used in 2-D convolution can be converted into a frequency domain filter by applying a Fourier transform on the template padded with zeros. An example of applying a 3x3 template in the frequency domain is shown in Figure 5.6.

Original Image

Filter Results:

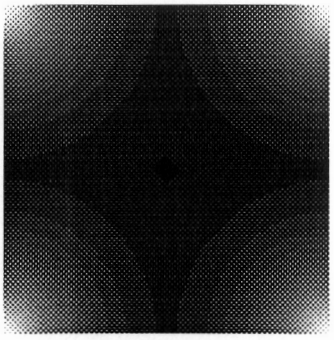


Butterworth

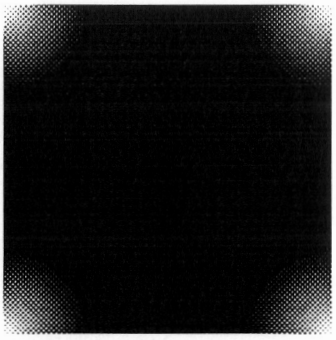
Exponential

Ideal

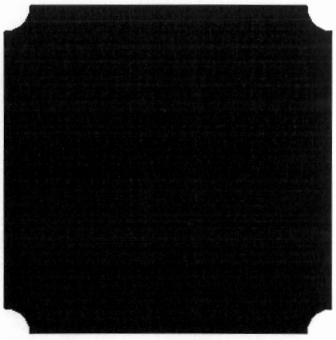
Filter Used:



Butterworth



Exponential



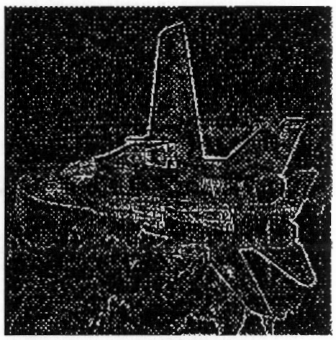
Ideal

Figure 5.4: Results from applying Butterworth, exponential, and ideal lowpass filters.

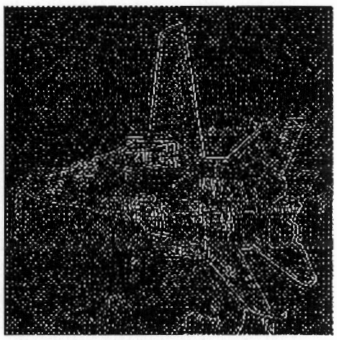
Original Image

Filter Results:

Butterworth

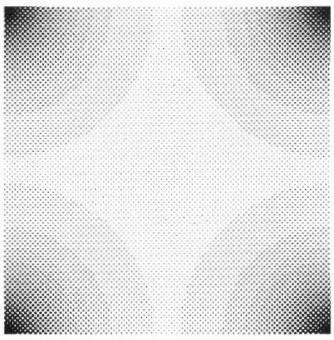


Exponential

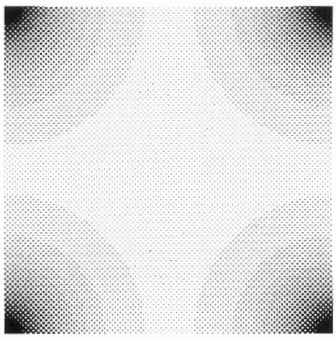


Ideal

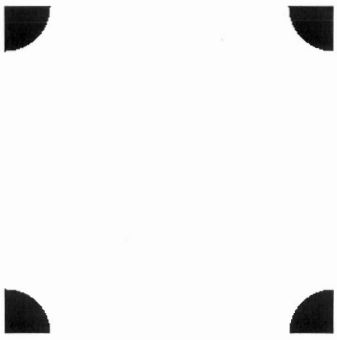
Filter Used:



Butterworth



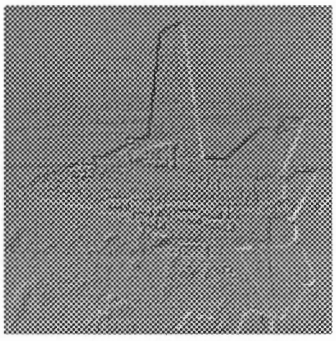
Exponential



Ideal

Figure 5.5: Results from applying Butterworth, exponential, and ideal highpass filters.

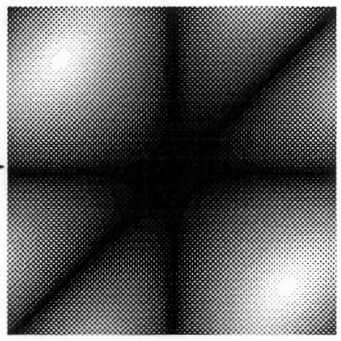
Original Image



Filtered Image

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

FFT



3x3 Template

Frequency Domain Filter

Figure 5.6: Results from applying a 3x3 template in the frequency domain.

5.4 User Interface

An X windows interface, shown in Figure 5.7, was developed for the FFT application. This interface enables different configurations and filters to be loaded on Splash-2 easily.

Figure 5.7: X windows interface.

Three different configurations are available. The first configuration computes an FFT on the input image, filters the image in the frequency domain, and displays the frequency domain on the video monitor. The second configuration does the same computation as the first configuration, and, in addition, an inverse FFT is computed on the filtered data. The resulting spatial domain image is displayed on the monitor. The third configuration is used to verify that the video interface is working properly. This configuration does no computation and simply passes the input video data through to the video monitor.

Four different size 2-D FFTs can be calculated. These sizes must be a power of 2 and are 64x64, 128x128, 256x256, and 512x512. Both 64x64 and 128x128 size FFTs can be computed in real-time (i.e., in less than 1/30 of second). Controls to start and stop execution and to reset all the flip-flops on the FPGAs on Splash-2 array boards are provided. The output colormap, zoom factor, and choice of a scale indicating the magnitude of an output pixel value can be selected.

Pre-calculated filters can be loaded in less than a second during run-time. New filters can be computed on the Sparc-2 host and then downloaded to Splash-2. Three different types of filters can be computed. Exponential, Butterworth, and ideal filters [5] can be generated, and each type of filter can be a lowpass, bandpass, bandreject, or a highpass filter. The diameter of each filter, and the band width of a bandpass and band reject filter can be selected.

The input and output images can be captured and down loaded to the Sparc-2 host. The captured image is displayed in the large sub-window in the upper right-hand corner of the interface window shown in Figure 5.7. In addition to displaying captured images, the window can display the current filter being used. The smaller sub-window of the interface is used to display status information.

Chapter 6

Error Analysis

To test round-off error associated with the 18-bit floating point format, a forward FFT followed by an inverse FFT was calculated without doing any filtering. This process should ideally result in an image which is exactly the same as the original image, however, due to round off error the output image differed slightly.

Statistics such as RMS and absolute error were calculated to quantify the error. The equations used for calculating the RMS and absolute error are:

$$RMS\ Error = \sqrt{\frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\left(I(x,y) - I'(x,y)\right)^2}$$

$$Absolute\ Error = \frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\left|I(x,y) - I'(x,y)\right|$$

where I(x,y) is the original image, and I'(x,y) is the output image.

Multiple images were tested and the average RMS error was 0.4% and the average absolute error was 0.2%. As a point of reference, an error of less than 1.0% is generally acceptable in lossy image compression. Each pixel value can have a gray-scale value from 0 to 256. The output image had a maximum deviation of 2 gray-scale values from the corresponding pixel in the original image. In addition, no difference could be seen between the original and output images.

The calculated statistical values indicate that the smaller 18-bit floating point format is adequate for this application. By down-sizing the floating point format, more floating point operations can be done per Splash-2 board resulting in increased performance.

Chapter 7

Performance

In order to compare the performance of this application thoroughly, a wide range of architectures was selected. The architectures range from a general purpose workstation to special purpose DSP processors. Since the FFT is a commonly used DSP algorithm, it is used as a benchmark by many DSP chip manufacturers. Two DSP chips were selected; one which has approximately the typical performance of a DSP chip, and one which shows high end performance.

The test case used to evaluate the different architectures is a 2-D spatial filter of dimensions 512x512 pixels. This process involves performing a complex 512x512 2-D FFT, filtering the image by doing 512x512 point-by-point multiplications for each of the real and imaginary values of the image in the frequency domain, and then doing a complex 512x512 2-D IFFT to convert the image back to the spatial domain.

Table 7.1: Execution time for an NxN 2-D FFT of Splash-2.

Size	Execution Time (seconds)	Number of Frames per Second
64x64	.005	203.5
128x128	.023	43.6
256x256	.105	9.5
512x512	.472	2.1

The performance of the Splash-2 implementation of the FFT was calculated in the following way: The number of clock cycles required to compute the NxN 2-D FFT is 2 $N^2 \log_2 N$. The application was run at 10 MHz and therefore the execution time for doing a 512x512, 2-D FFT is 2 $(512)^2 \log_2(512)$ / $10x10^6$ = 0.47186 seconds or 2.1 frames per second. The execution times for a 64x64, 128x128, 256x256, and 512x512 2-D FFT are given in Table 7.1. Since the FFT and the IFFT are pipelined and are being computed concurrently, the time for the complete filtering process is essentially the time for calculating one 2-D FFT. The speed of this application was verified by using a logic analyzer to check the time between output frames. In addition, there are 18 floating point units distributed between the FFT, filter and IFFT designs. These units output a result every clock cycle at 10 MHz therefore, this application operates at 180 Mflops. This Mflops rating does not include the floating point load and store operations (which are tallied by other benchmarks). If these operations are taken into consideration, the total number of floating point operations per second would be 240 Mflops since there are 6 additional memory operations. These 6 memory operations can be broken down to two loads and two stores

for each complex data value and two loads for the complex twiddle factor value. The execution time, Mflops rating, and the speed-up factor of the Splash-2 implementation relative to a given architecture is shown in Table 7.1.

Table 7.2: Comparison of Splash-2 implementation with other architectures. (Note: the execution times are provided by the vendors)

	Execution Time (seconds)	Mflops	Speed-up Factor
Splash-2	**.47**	**180**	**1**
Sparc-2	18	32	38.3
Sparc-10	11	60	23.4
Intel i860	.35	200	.74
TI DSP TMS320C40	1.7	80	3.6
Sharp LH9124 DSP	.08	300	.17

For comparison purposes, the Cooley-Tukey FFT algorithm [5] was implemented in C and was compiled using the highest optimization level of the *gcc* compiler on a Sparc workstation.

The execution time for the Intel i860 based processing board, Texas Instruments, and Sharp DSP chips was estimated by doubling the time required to do a single 512x512 2-D complex FFT and then adding a relatively very small amount of time for the filtering process. The time for doing an FFT was doubled in order to account for the time to do an IFFT. The i860 processing board consisted of two 50 MHz i860 chips and 200 Mbytes of RAM. The Sharp DSP chip was chosen because it was the fastest DSP chip surveyed out of almost 60 DSP chips [17]. The Sharp DSP can do a complex multiply in one clock cycle at 40 MHz [18]. The Texas Instruments DSP chip was selected because its perfor-

mance was about average for the DSP chips in the survey. The algorithm used in the survey to benchmark the DSP chips was a 1024 point, one dimensional FFT.

It is essential to note that the other implementations used 32-bit single precision floating point arithmetic, and the Splash-2 design takes advantage of the ability to use a smaller 18-bit floating point format. However, this smaller format requires less computation per floating point arithmetic unit than the 32-bit implementation. To implement single precision floating point arithmetic units on the Splash-2 architecture, the size of the floating point arithmetic units would increase between 2 to 4 times over the 18-bit format. A multiply unit would require two Xilinx 4010 chips and an adder/subtracter unit could fit into a single Xilinx chip. The 24-bit multiplier needed in single precision floating point multiply can be broken up into four 12-bit multipliers, allocating two per chip [19]. After synthesizing a few different size multipliers, it was found that a 16x16 bit multiplier was the largest parallel integer multiplier that could fit into a Xilinx 4010 chip. When synthesized, this multiplier used 75% of the chip area. However, there was no need to emulate the 32-bit floating point arithmetic since the desired accuracy was achieved with an 18-bit format. This illustrates an important advantage of FPGA-based computers over traditional approaches.

In summary, the Splash-2 performance is more than an order of magnitude better than a general purpose workstation and is similar to an i860 processing board which is faster than many DSP processors. On the other hand, the Splash-2 implementation is less than six times slower than one of the fastest DSP processors on the market.

Chapter 8

Conclusions

Due to the flexibility of a CCM, customization of the floating point format can be performed to achieve adequate accuracy for a given task with the least number of bits. By taking advantage of the parallelism of the Splash-2 architecture, address calculation, butterfly operations and filtering could be done concurrently. By pipelining the butterfly operation, a real and complex result was obtained every clock cycle at 10 MHz. The performance of this application is much faster than a Sparc-10 workstation and is similar to typical a DSP processor.

The Splash-2 architecture has been used to improve the performance of a wide range of applications and can be considered as a general purpose custom computing platform. Applications include pattern matching, text searching and genome data base searching, and many different image processing algorithms [7, 20, 8]. The genome base search implementation has shown a speed-up of three orders of magnitude over the MasPar-1.

The Splash-2 implementation of a 2-D FFT has shown that the performance is similar to a DSP chip and has shown that floating point arithmetic can be done on CCMs effectively.

Bibliography

[1] S. Webber, "Programmable Uproar," *ASIC and EDA*, December, 1992, pp. 18-33.

[2] M. Gokhale et al., "SPLASH - Experience Building and Programming a Highly Parallel Programmable Logic Array," *IEEE Computer*, Vol. 24, No. 1, January, 1991, pp. 81-89.

[3] J.A. Eldon and C. Robertson, "A Floating Point Format for Signal Processing", *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1982, pp. 717-720.

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

[5] R. Gonzalez and P. Wintz, *Digital Image Processing*, Addison-Wesley Publishing Company, 1977.

[6] E. Swartzlander, *Systolic FFT Processors*, 1st International Workshop on Systolic Arrays, Oxford, July 1986.

[7] D. Hoang, "Searching Genetic Databases on Splash-2", *IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, pp. 185-191.

[8] P. Athanas and A L. Abbott, "Real-Time Image Processing on a Custom Computing Platform," *IEEE Computer*, Vol. 28, No. 2, February 1995, pp. 16-24.

[9] J, Arnold, D. Buell and E. Davis, "Splash-2," *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, June, 1992, pp. 316-322.

[10] Xilinx, Inc., *The Programmable Logic Data Book*, San Jose, California, 1995.

[11] Analog Devices Corp., Hybrid RS-170 Video Digitizer, Data Sheet for the AD9502 chip.

[12] Data Translation Inc., DT2867 and DT2867-LC Software User Manual, Second Edition, October 1992.

[13] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Boston, MA., 1989.

[14] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA-Based Custom Computing Machines," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[15] IEEE Task P754, "A Proposed Standard for Binary Floating Point Arithmetic," *IEEE Computer*, Vol. 14, No. 12, March 1981, pp. 51-62.

[16] K. Eshraghian and N.H.E. Weste, *Principles of CMOS VLSI Design, A Systems Perspective*, 2nd Edition, Addison-Wesley Publishing Company, 1993.

[17] *Computer Design*, "1995 Product Trends and Resource Guide," February 1995, pp. 44-47.

[18] Sharp Electronics Corp., "Fast Fourier Transform," Sharp Application Note for the LH9124 DSP Chip, November 1992.

[19] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," *IEEE Transactions on VLSI*, Vol. 2, No. 3, September 1994, pp. 365-367.

[20] D. Pryor, M. Thistle, and N. Shirazi, "Text Searching on Splash-2," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1993.

[21] D. Quong and R. Perlman, "One-chip sequencer shapes up addressing for large FFTs," *Electronic Design*, Vol. 32, No. 14, 1984.

[22] D. Korchev and J. Kanevsky, "Processor arrays for two-dimensional discrete Fourier transform," *IEEE Proceedings-E,* Vol. 140, No. 2, March 1993.

[23] A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

Appendix A

VHDL Code for a Floating Point Adder

```
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use SPLASH2.HMACROS.all;

----------------------------------------
--  Floating Point Adder (18-bit format)
----------------------------------------

-- STATS:
--   FG Function Generators:                  Used=229  Max=800  Util=29%
--   Maximum Clock Speed:                     8.6 MHz
--   Number of clock cycles to compute result:   3

entity FP_Add is
    port( a, b : in Bit_Vector(17 downto 0); Clk: in Bit; c : out Bit_Vector(17 downto 0) );
end FP_Add;


architecture behavior of FP_Add is
    signal s1:                      Bit;
    signal e1:                      Unsigned(7 downto 0);
    signal f1:                      Unsigned(15 downto 0);
    signal f1_dly:                  Unsigned(15 downto 0);
    signal s2:                      Bit;
    signal s2_dly:                  Bit;
    signal e2:                      Unsigned(7 downto 0);
    signal e2_dly:                  Unsigned(6 downto 0);
    signal f2:                      Unsigned(15 downto 0);
    signal f2_dly:                  Unsigned(15 downto 0);
    signal s_result:                Bit;
    signal s_result_dly:            Bit;
    signal e_result:                Bit_Vector(6 downto 0);
    signal e_result_dly:            Bit_Vector(6 downto 0);
    signal f_result:                Unsigned(9 downto 0);
    signal temp_e:                  Unsigned(8 downto 0);
    signal temp_f:                  Unsigned(16 downto 0);
    signal e_add_sub:               Bit;
    signal f_add_sub:               Bit;
    signal e_diff:                  Unsigned(7 downto 0);
    signal ofl:                     Bit;

begin

    -- Take the difference of e1 and e2
    e_add_sub <= '0';

    process

        variable e1_var:            Unsigned(6 downto 0);
```

```
    variable f1_var:              Unsigned(10 downto 0);
    variable e2_var:              Unsigned(6 downto 0);
    variable f2_var:              Unsigned(10 downto 0);
    variable f2_dly_var:          Unsigned(10 downto 0);
    variable temp_f_var:          Unsigned(11 downto 0);
    variable e_result_var:        Unsigned(6 downto 0);
    variable e_diff_var:          Unsigned(7 downto 0);
    variable shift_val_var:       Unsigned(3 downto 0);
    variable dec_val_var:         Unsigned(6 downto 0);

    variable carry:               Bit;
    variable msb:                 Bit;

    constant e_zero:              Bit_Vector(6 downto 0) := "0000000";
    constant f_zero:              Bit_Vector(9 downto 0) := "0000000000";
    constant zero_7:              Bit_Vector(6 downto 0) := "0000000";
    constant zero_8:              Bit_Vector(7 downto 0) := "00000000";
    constant zero_12:             Bit_Vector(11 downto 0) := "000000000000";
    constant zero_17:             Bit_Vector(16 downto 0) := "00000000000000000";

begin
    wait until Clk'Event and Clk = '1';

    -- Change 0.f to 1.f for a and b if not equal to zero
    if not(a(16 downto 0) = zero_17) then
        f1_var(10) := '1';
    else
        f1_var(10) := '0';
    end if;
    f1_var(9 downto 0) := a(9 downto 0);

    if not(b(16 downto 0) = zero_17) then
        f2_var(10) := '1';
    else
        f2_var(10) := '0';
    end if;
    f2_var(9 downto 0) := b(9 downto 0);

    -- Exponent
    e1_var := a(16 downto 10);
    e2_var := b(16 downto 10);


    -- Check if b > a, if true then swap a and b to make a > b
    if ((e2_var > e1_var) or ((e1_var = e2_var) and (f2_var > f1_var)) ) then
        -- Swap a and b and separate fields

        -- Sign
        s1 <= b(17);
        s2 <= a(17);

        -- Exponent
        e1(6 downto 0) <= b(16 downto 10);
        e1(7) <= '0';
        e2(6 downto 0) <= a(16 downto 10);
        e2(7) <= '0';

        -- F field
        f1(10 downto 0) <= f2_var;
        f2(10 downto 0) <= f1_var;

    else
        -- Break a and b up into separate fields

        -- Sign
        s1 <= a(17);
        s2 <= b(17);

        -- Exponent
        e1(6 downto 0) <= a(16 downto 10);
        e1(7) <= '0';
        e2(6 downto 0) <= b(16 downto 10);
        e2(7) <= '0';

        -- F field
        f1(10 downto 0) <= f1_var;
```

```
    f2(10 downto 0) <= f2_var;
end if;

-- Pad f1 and f2 in order to use 16 bit hard macro
f1(15 downto 11) <= "00000";
f2(15 downto 11) <= "00000";


-- Check sign to see if f_result = f1 + f2 or f1 - f2
if (s1 = s2) then
    f_add_sub <= '1';  -- Add
else
    f_add_sub <= '0';  -- Subtract
end if;


-------------------------------
-- Shift f2 right ediff places
-------------------------------
if ((e_diff(7) = '1') or (e_diff(6) = '1') or  (e_diff(5) = '1') or (e_diff(4) = '1')) then
    e_diff_var(3 downto 0) := "1111";
else
    e_diff_var(3 downto 0) := e_diff(3 downto 0);
end if;
f2_dly_var := f2(10 downto 0);

if (e_diff_var(0) = '1') then
    f2_dly_var(9 downto 0) := f2_dly_var(10 downto 1);
    f2_dly_var(10) := '0';
end if;
if (e_diff_var(1) = '1') then
    f2_dly_var(8 downto 0) := f2_dly_var(10 downto 2);
    f2_dly_var(10 downto 9) := "00";
end if;
if (e_diff_var(2) = '1') then
    f2_dly_var(6 downto 0) := f2_dly_var(10 downto 4);
    f2_dly_var(10 downto 7) := "0000";
end if;
if (e_diff_var(3) = '1') then
    f2_dly_var(2 downto 0) := f2_dly_var(10 downto 8);
    f2_dly_var(10 downto 3) := "00000000";
end if;

f2_dly(10 downto 0) <= f2_dly_var;
f1_dly <= f1;

s_result_dly <= s1;
s2_dly <= s2;
s_result <= s_result_dly;


e_result_dly <= e1(6 downto 0);
e2_dly <= e2(6 downto 0);
e_result_var := e_result_dly;
temp_f_var := temp_f(11 downto 0);



-------------------
-- Normalize Result (temp_f)
-------------------

carry := temp_f(11);
msb := temp_f(10);

if (carry = '1') then
    f_result <= temp_f(10 downto 1);
    e_result <= e_result_var + 1;
else

    -- Find position of msb
    if (msb = '1') then
        shift_val_var := "0000";
    else
        shift_val_var(3) := ( (temp_f_var(0) or temp_f_var(1) or temp_f_var(2)) and
              not(temp_f_var(3) or temp_f_var(4) or temp_f_var(5) or temp_f_var(6) or
```

```
                temp_f_var(7) or temp_f_var(8) or temp_f_var(9))  );
            shift_val_var(2) := ( (temp_f_var(3) or temp_f_var(4) or temp_f_var(5) or
                 temp_f_var(6)) and
                 not(temp_f_var(7) or temp_f_var(8) or temp_f_var(9))  );
            shift_val_var(1) := ( (temp_f_var(0) and not(temp_f_var(1) or temp_f_var(2) or
               temp_f_var(3) or
               temp_f_var(4) or temp_f_var(5) or temp_f_var(6) or
               temp_f_var(7) or temp_f_var(8) or temp_f_var(9)) )  or
                   ((temp_f_var(3) or temp_f_var(4)) and not(temp_f_var(5) or temp_f_var(6) or
               temp_f_var(7) or temp_f_var(8) or temp_f_var(9)) )  or
                   ((temp_f_var(7) or temp_f_var(8)) and not(temp_f_var(9))  )  );
            shift_val_var(0) := ( (temp_f_var(1) and not(temp_f_var(2) or temp_f_var(3) or
                temp_f_var(4) or
               temp_f_var(5) or temp_f_var(6) or temp_f_var(7) or
               temp_f_var(8) or temp_f_var(9)) ) or
                  (temp_f_var(3) and not(temp_f_var(4) or temp_f_var(5) or temp_f_var(6) or
               temp_f_var(7) or temp_f_var(8) or temp_f_var(9)) ) or
                  (temp_f_var(5) and not(temp_f_var(6) or temp_f_var(7) or temp_f_var(8) or
               temp_f_var(9)) ) or
                  (temp_f_var(7) and not(temp_f_var(8) or temp_f_var(9)) ) or
                  temp_f_var(9)  );
        end if;


        -- Shift f2 left shift_val_var places
        if (shift_val_var(0) = '1') then
            temp_f_var(10 downto 1) := temp_f_var(9 downto 0);
            temp_f_var(0) := '0';
        end if;
        if (shift_val_var(1) = '1') then
            temp_f_var(10 downto 2) := temp_f_var(8 downto 0);
            temp_f_var(1 downto 0) := "00";
        end if;
        if (shift_val_var(2) = '1') then
            temp_f_var(10 downto 4) := temp_f_var(6 downto 0);
            temp_f_var(3 downto 0) := "0000";
        end if;
        if (shift_val_var(3) = '1') then
            temp_f_var(10 downto 8) := temp_f_var(2 downto 0);
            temp_f_var(7 downto 0) := "00000000";
        end if;

        -- Subtract shift_val from e_result
        dec_val_var(3 downto 0) := shift_val_var;
        dec_val_var(6 downto 4) := "000";

        f_result <= temp_f_var(9 downto 0);
        -- If adding a number plus it's negative the answer should be zero
        if ( (temp_f_var = zero_12) and (e_result_dly = e2_dly) and (s_result_dly = not(s2_dly)) )
           then
            e_result <= zero_7;
        else
            e_result <= e_result_var - dec_val_var;
        end if;
    end if;

    -- Pack c with s_result, e_result, and f_result
    c(17) <= s_result;
    c(16 downto 10) <= e_result;
    c(9 downto 0) <= f_result;

end process;
U0: adsu8h
    port map(e1, e2, e_add_sub, e_diff, ofl);

U1: adsu16h
    port map(f1_dly, f2_dly, f_add_sub, temp_f(15 downto 0), temp_f(16));

end behavior;
```

Appendix B

VHDL Code for a Floating Point Multiplier

```
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;

------------------------------
--  Floating Point Multiplier  (18 bit IEEE 754 format)
------------------------------

-- STATS:
--   FG Function Generators:                    Used=352  Max=800  Util=44%
--   Maximum Clock Speed:               202.5ns or 4.9 MHz
--   Number of clock cycles to compute result:  3


entity FP_Mult is
    port( a, b : in Bit_Vector(17 downto 0); Clk: in Bit;c : out Bit_Vector(17 downto 0) );
end FP_Mult;


architecture behavior of FP_Mult is
    signal s1:                          Bit;
    signal e1:                          Unsigned(6 downto 0);
    signal e1_dly:                      Unsigned(6 downto 0);
    signal f1:                          Unsigned(10 downto 0);
    signal s2:                          Bit;
    signal e2:                          Unsigned(6 downto 0);
    signal e2_dly:                      Unsigned(6 downto 0);
    signal f2:                          Unsigned(10 downto 0);
    signal s_result:                    Bit;
    signal e_result:                    Unsigned(7 downto 0);
    signal f_result:                    Unsigned(9 downto 0);
    signal s_result_dly:                Bit;
    signal e_result_dly:                Unsigned(6 downto 0);
    signal f_result_dly:                Unsigned(9 downto 0);
    signal temp_f:                      Unsigned(11 downto 0);
    signal e_sum:                       Unsigned(7 downto 0);
    signal e_sum_dly:                   Unsigned(7 downto 0);


    constant zero_1:                    Bit := '0';
    constant e_zero:                    Unsigned(6 downto 0) := "0000000";
    constant f_zero:                    Unsigned(9 downto 0) := "0000000000";
    constant sixty_two:                 Unsigned(7 downto 0) := "00111110";
    constant sixty_three:               Unsigned(7 downto 0) := "00111111";

    component Mult11
        port ( x : in Unsigned(10 downto 0);
               y : in Unsigned(10 downto 0);
               Clk : in Bit;
               z : out Unsigned(11 downto 0) );
    end component;
```

```
begin
    process

    begin
        wait until Clk'Event and Clk = '1';

        -- Break down a and b into s, e, and f components

        -- Sign
        --------
        -- 1st Stage
        s1 <= a(17);
        s2 <= b(17);

        -- 2nd Stage (Calculate Sign)
        if (s1 = s2) then
            s_result_dly <= '0';
        else
            s_result_dly <= '1';
        end if;

        -- 3rd Stage
        s_result <= s_result_dly;



        -- Exponent
        ------------

        -- 1st Stage
        e1 <= a(16 downto 10);
        e2 <= b(16 downto 10);
        -- Note: Max e_sum = 188 else Overflow
        --       Min e_sum = 63 else Underflow
        e_sum <= (zero_1 & a(16 downto 10)) + (zero_1 & b(16 downto 10));
        -- 2nd Stage
        e_sum_dly <= e_sum;


        -- Change 0.f to 1.f for a and b
        f1(10) <= '1';
        f1(9 downto 0) <= a(9 downto 0);
        f2(10) <= '1';
        f2(9 downto 0) <= b(9 downto 0);


        -- Check for a = 0 or b = 0 or Underflow
        if (((e1 = e_zero) and (f1(9 downto 0) = f_zero)) or
            ((e2 = e_zero) and (f2(9 downto 0) = f_zero)) or
            (e_sum < sixty_three) ) then
            e_result_dly <= e_zero;
            f_result_dly <= f_zero;
        else
            -- Put a non zero value in the pipe
            e_result_dly(0) <= '1';
            f_result_dly(0) <= '1';
        end if;


        if (not((e_result_dly = e_zero) and (f_result_dly = f_zero)) ) then
            -- Check for carry
            if (temp_f(11) = '1') then
                 -- Calculate Exponent and Add 1 since carry
                e_result <= e_sum_dly - sixty_two;
                f_result <= temp_f(10 downto 1);
            else
                -- Calculate Exponent Normally
                e_result <= e_sum_dly - sixty_three;
                f_result <= temp_f(9 downto 0);
            end if;
        else
            e_result(6 downto 0) <= e_result_dly;
            f_result <= f_result_dly;
        end if;

        -- Pack c with s_result, e_result, and f_result
```

```
        c(17) <= s_result;
        c(16 downto 10) <= e_result(6 downto 0);
        c(9 downto 0) <= f_result;

    end process;

    U0:  Mult11
        port map(f1, f2, Clk, temp_f(11 downto 0));

end behavior;

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;

---------------------
--  11 bit Multiplier
---------------------


entity Mult11 is
    port( x, y : in Unsigned(10 downto 0); Clk: in Bit;z : out Unsigned(11 downto 0) );
end Mult11;


-- New Multiplier


architecture structural of Mult11 is

begin
   process
        variable tmp : Unsigned(21 downto 0);
     begin
        wait until Clk'Event and Clk = '1';
        tmp := x * y;
        z <= tmp(21 downto 10);
    end process;

end structural;
```

Vita

Nabeel Shirazi was born in Columbus, Ohio on August 17, 1970. He attended Beavercreek High School in Beavercreek, Ohio. After high school, he received a summer internship with the Supercomputing Research Center (SRC), in Bowie, Maryland. During his first summer at SRC he was able to work on applications for Splash-1 and became extremely interested and fascinated with the concept of using FPGAs for computing instead of just glue logic. He earned his Bachelor's degree in Electrical Engineering from The Ohio State University and graduated in June 1993. During his undergraduate degree, he continued interning each summer at SRC and contributed to the development of the Splash-2. In the Fall 1993, he started studying for his Master's degree at Virginia Polytechnic Institute and State University, Blacksburg, Virginia, where he also continued working in the field of FPGA computing. He received his Master's degree in May 1995.

While at Virginia Tech, he joined the Crew team and found out it is actually possible to wake up at 5:30 a.m. He really enjoyed rowing for a couple hours first thing in the morning on Claytor Lake, an absolutely beautiful lake thirty minutes south of Blacksburg in the Blue Ridge mountains. He also likes to travel and is looking forward to spending the next three months in Edinburgh, Scotland working for Xilinx Development Corporation. He plans to continue his education next year by pursuing a Ph.D. degree in Electrical Engineering.