Automatic Synthesis and Optimization of Floating Point Hardware

Ho Chun Hok

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Philosophy

in

Department of Computer Science & Engineering

July, 2003

Automatic Synthesis and Optimization of Floating Point Hardware

submitted by

Ho Chun Hok

for the degree of Master of Philosophy

at the Chinese University of Hong Kong

Abstract

This thesis presents a methodology for designing floating point and fixed point systems on FPGA platforms by means of a programming language. A compiler, *fly*, floating point library, *float* and arbitrary function module generator, were developed for the rapid system prototyping research. *fly* takes a Perl-like program as input and produces a synthesizable VHDL description of a one-hot state machine and the associated datapath elements as output. Furthermore, it is tightly integrated with the hardware design environment and implementation platform, and is able to hide issues associated with these tools from user. The *float* library consists of a floating point class for the simulation of quantization effects associated with high precision floating point operators, an optimizer which can automatically determine the minimal number of exponent and fraction bits required for a specified degree of accuracy, and a parameterized floating point library which can generate floating point operators with arbitrary precision. The function generator can generate any one-operand function and is compatible with the *fly* compiler. The systems was used to prototype an FPGA based greatest common divisor (GCD) coprocessor, digital sine-cosine generator, a dedicated circuit for solving ordinary differential equation (ODE), and a simulation model for the N-Body problem. By combining these design tools, the time and knowledge required for a designer to implement a floating point algorithm in hardware can be greatly reduced.

浮點運算硬件的自動生成及優化

何循學

香港中文大學

計算機科學與工程學課程

哲學碩士論文

2003年7月

摘要

本論文展示了一個以編程語言爲手段去設計現場可編程門陣列平台上的浮點和定點系統。爲了研究快速系統原型化，我們開發了編譯器(*fly*)，浮點程式庫(*float*)，及任意函數模塊產生器。*fly* 接受類似 Perl 的編程語言作爲輸入，並產生一個以 VHDL 作爲描述的 one-hot 狀態機和相應的數據通道。再者，它與硬件設計環境和實施平台緊密結合，讓使用者能略過硬件設計工具的使用問題。*Float* 程式庫包括了一個可以模擬高精度浮點運算的量化效果的浮點數類及一個可以自動判定在特定準確度要求下的指數和分數部份的最少位元的優化器，和一個可以生成任意精確度浮點運算子的參數化的浮點數程式庫。函數產生器可以產生任何一元操作符的函數及可以與 *fly* 編繹器兼容。此系統可用來製作基於現場可編程門陣列的最大公約數協處理器的原型，數字式正弦-餘弦產生器，解決一般微分方程的專門電路和一個 N 體問題模擬模型。通過結合這些設計工具，在硬件上實施浮點演算法時，可大大減少對設計者的時間和知識要求。

Acknowledgment

Many people have contributed to my education through their guidance and support in my graduate school years. I especially wish to thank my final year project and Master degree supervisor, Dr. Philip Leong for his suggestion and ideas on research. He also reviewed my manuscript carefully. This dissertation cannot be done without his help and support.

I would like to acknowledge Dr. P. Zipf, Mr. R. Ludewig and Mr. A. G. Ortiz of Institute of Microelectronic Systems, Darmstadt University of Technolgy for the development of the *fly* compiler project. They provided the embryo of *fly* compiler so that I can extend from their work.

Thanks must be given to Mr. K. H. Tsoi, who assisted me in debugging various host interface used in this thesis. He also reviewed my chinese abstract thoroughly

I would like to thank my colleagues. In particular, Mr. Y. H. Cheung, Mr. C. W. Sham, Mr. Y. M. Lam, Mr. C. L. Yuen, Mr. K. Y. Tong and F. Wu for their assistance and support.

Finally, I would like to thank my parents for their love, warmth and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Traditional development method for FPGA is complex

In the standard field programmable gate array (FPGA) based prototyping methodology, algorithms are first developed in programming languages such as C on a personal computer or workstation using floating point arithmetic. When the system is later implemented in hardware, a fixed point version of the algorithm is derived from the floating point version and then translated into a hardware design in a hardware description language such as VHDL. Finally, the design is synthesized for a field programmable gate array (FPGA) based prototyping environment where it can be tested.

However, it is found that using a HDL based design methodology results in low productivity compared with software development with programming language because of the following issues:

- Hardware designs are parallel in nature while most of the people think in sequential patterns
- The standard technique of decomposing a hardware design into datapath and controls adds complexity to the task

- Designers must develop a hardware interface for the FPGA board as well as a software/hardware interface between a host system and the FPGA
- Elementary functions are not supported and designer needs to build operations like reciprocal, log and sin from primitive operations before the design can actually begin

The above issues significantly increase the design complexity, with associated increase in design time and debugging, especially in developing the interface between a host system and the FPGA. Furthermore, the time spent in the above process restricts the amount of time which can be spent on dealing with higher level issues such as evaluating different algorithms and architectures for the system.

Floating point arithmetic can take advantages on FPGA

Today, FPGA systems have almost solely used fixed point arithmetic. Although several groups have implemented floating point adders and multipliers using FPGA devices [SWA95, LMM$^+$98, JL01], very few systems employing floating point arithmetic have been reported. It is envisaged that FPGA density has improved to a point where area concerns are becoming less significant, and aided by Moore's Law, silicon density will continue to improve at an exponential rate. It is believed that hardware systems employing floating point computations will become increasingly popular as the density of hardware improves, particularly in applications where variables have a very large dynamic range, or the designer wishes to avoid the complexity of translating the implementation to fixed point.

In this work, an efficient way to implement floating point arithmetic on FPGA using flexible architectures will be presented.

1.2 Aims

The objective of this research was to provide a design environment such that any algorithm designer, even if not an expert in hardware development, can implement their floating point algorithm on the FPGA by using Perl-like language to describe their algorithm. The detail research aims are:

- The designer need not be familiar with hardware description language yet can implement the algorithm on the FPGA.
- The interface between the host and the FPGA board is encapsulated such that it hides the details of the host interface from the designer.
- The designer need not have expertise in the implementation of floating point arithmetic.
- The designer can focus on the algorithm and the implementation is done by the system.
- Any differentiable function can be automatically generated and used in the language
- Design time is greatly reduced since the simulation is done at a very high level and the resulting hardware implementation is correct by construction.

1.3 Contributions

To address the design time issue, a compiler called *fly* for the translation of software descriptions into hardware is developed. The input of *fly* is a Perl-like description and it generates synthesizable VHDL for adaption to different FPGA and ASIC design tools. In addition, a VHDL Floating Point library was designed in which includes an optimizer for determining the minimum floating point precision for each variable to reach some user-specified tradeoff between quantization error and circuit size. To enhance its flexibility, arbitrary functions for fixed point arithmetic is supported through table lookup approach. To the best of the author's knowledge, the integration of a hardware compiler, floating point library, optimizer and table lookup generator, resulting in a dedicated development environment is novel.

Several applications, using fixed point and floating point arithmetic, have been developed using the tools. These include the following:

- Greatest Common Divisor Processor
- Digital Sine Cosine Generator
- Ordinary Differential Equation Solver
- N-Body Problem Simulator

Compared with previous design systems, the design time required for these application is greatly reduced while the error is eliminated by automatic hardware construction.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes previous work and implementations. Chapter 3 introduces floating point arithmetic. In chapter 4, the *fly* compiler is described. Chapter 5 will discuss the optimization of floating point operations and the related library will be presented. The implementation of the table lookup approach and the algorithm will be described in chapter 6. Results from experiments using the system will be reported in chapter 7. Conclusions will be drawn and further work suggested in chapter 8

Chapter 2

Background and Literature Review

2.1 Introduction

This chapter provides some background informations about the thesis. It includes an introduction to Field Programmable Gate Array (FPGA) technology and one of its development languages - VHDL. Then the chapter reviews previous hardware compilation techniques, construction of floating point arithmetic and implementation of functions using the look up table approach. Hardware compilation refers to translation of an algorithm specified in a source file into a hardware design. The aim of program translation is to build a working environment such that implementation of FPGA application is just like software programming, avoiding traditional hardware level descriptions completely [Pag96].

2.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGA) are an integrated circuits where the functionality can be modified in the field after the fabrication. Therefore, FPGA can be customized for different application as long as the device itself is complex enough to store the logic.

A regular FPGA chip consists of an array of logic blocks and routing channels. I/O pads are attached at the sides of the chip. Both logic blocks and routing channels can be reconfigured to handle arbitrarys function and connections respectively. Different FPGA chips have different internal structure of logic blocks.

In this research, the Xilinx Virtex XCV1000E FPGA [Xil01] will be used unless otherwise specified. The XCV1000E contains 6,144 configurable logic blocks (CLB). Each logic block contains 4 logic cells and organized in two similar slices. The slice can be referred as the primitive component in XCV1000E. Each slices consists of two 4-input look up tables (LUTs) and two flip-flops. XCV1000E also provides 96 blocks of on-chip dual-read/write port synchronous RAM with 4096 memory cells in each block. The storage element can use for data transferring between the host machine and FPGA board and act as temporal storage inside the FPGA. The routing channel is implemented using routing matrix which can connect I/O pads, clock signal and general purpose logic together.

2.3 Traditional design flow and VHDL

Several steps are necessary for implementing customized functions on FPGA chips. It is first required to simulate the algorithm in software, construct the datapath in hardware, design the control signals for the datapath, simulate the datapath and control signals for verification and implement a protocol for interfacing between the host and FPGA board.

Though simulating the algorithm on software is easy for a software designer, the remaining stages require extra hardware knowledge to realize the design. To construct the datapath, a schematic approach can be used for simple design but it may not be practical to implement some real life applications which often involve thousands of logic gates. Therefore, it is necessary to use a hardware description language such as VHDL [IEE02] when implementing complex logic on the hardware or FPGA. Even though the programming language and hardware description language share some properties like variables versus signal, the nature of hardware description language is totally different from programming language. Hardware description language, as the name suggests, is used to describe the hardware functionality. Unlike normal programming languages, hardware description languages may run several operations in parallel and explicit specification of the timing is required to make the design work.

In software designs, the execution sequence of the code is sequential. To achieve the same effect in the hardware, control signals and state machines can be described using VHDL. To complete the logic design, both datapath and state machines must be implemented. Mostly it involves rewriting the algorithm in VHDL.

In order to program the FPGA, a bitstream generated by the design tool is required. The VHDL code will be synthesized into a netlist. The netlist will contain the representation of the hardware such as the function of each basic blocks and the connection between the blocks. The design tool will extract the information in the netlist and map the logical blocks and connection to specific lookup table and routing matrix respectively. It finally produces a bitstream can customize the functionality of the FPGA by writing this information onto the chip.

2.4 Single Description for Hardware-Software Systems

I. Page [Pag96] demonstrated the translation of basic programming constructs, including assignment statement, parallel composition, sequential composition, conditional composition and repetitive composition, into hardware. I. Page used this architecture to implement a real-time video processing application. It is reported that the fully operational, high-bandwidth hardware system was constructed by an undergraduate programmer without knowledge of hardware as a summer course project.

M. Ward et al [WA02] proposed a hardware implementation of the Ada language that allows accurate timing analysis. It supports standard programming statements such as assignment, branching and loop and include non-recursive sub-program calls. Two standard parameter-function passing techniques, namely pass-by-value and pass-by-reference can be used in this language depends on the type of variable. The timing of the produced circuit is analyzed accurately and the main application is the real-time systems.

MATCH (MATlab Compiler for Heterogeneous computing systems) [BSC+99] is a compiler project developed at Northwestern University. MATCH takes MATLAB descriptions of various embedded systems applications, and automatically maps them on to a configurable computing environment consisting of FPGAs, embedded processors and digital signal processors. Among the supported function are matrix addition, matrix multiplication and one dimensional FFT, FIR and IIR filters. The code generation of FPGA is a conversion to VHDL so branching and assignment is straight forward. A finite state machine was developed to control loop statement. A MPEG decoder was developed using heterogeneous set of resources as a MATCH example.

2.5 Parameterized Floating Point Arithmetic Implementation

FPGA technology is desirable for parameterized floating point arithmetic implementation. A. Jaenicke and W.Luk [JL01] have implemented parameterized floating point adder and multiplier on FPGAs. The design is based on Handel-C language and the data format is variance of IEEE standard. It's reported that the floating point adder can perform 28 MFLOPS for arbitrary sizes of fraction and exponent. A 2D Fast Hartley Transform (FHT) processor has been developed by using this FPU as basic building blocks and it can perform a 1K-point transform in 10 μs.

P. Belanoviċ et al [BL02] implemented a parameterized floating point library for use with reconfigurable hardware. It is based on the IEEE 754 floating point format standard. The library includes addition, subtraction, multiplication and conversion between fixed point and floating point numbers. All of these modules are specified in VHDL and implemented on the Wildstar reconfigurable computing engine. They are fully-pipelined and cascadable to form pipelines of floating point operations. This library was used to develop a hybrid implementation of the K-means clustering algorithm applied to multispectral images.

2.6 Function Approximations by Table Lookup and Addition

Elementary function approximations are important in scientific computing. Lookup table approach is the most common technique for implementing these functions since the storage size is increased rapidly in FPGA device recently. J. E. Stine and M.J. Schulte [SS99a] have developed a method for computing elementary functions using parallel table lookups and multi-input adder. The method is suitable for any differentiable function and the input range can be varied according to specific needs. The latency of the design is low because of applying parallelism.

2.7 Summary

In this chapter, different aspects of FPGA design, including applying single description for both hardware and software system, floating point arithmetic and elementary functions implementation have been reviewed. This thesis will apply these techniques to form rapid system prototyping of floating point systems.

Chapter 3

Floating Point Arithmetic

3.1 Introduction

This chapter is an introduction to floating point number arithmetic. Floating point algorithms are used frequently in modern applications such as speech recognition, image processing and financial engineering because of its ability to represent a good approximation to the real numbers.

The IEEE 754 floating point standard [ANS85] has been widely accepted for representing floating point numbers. With this standard, the result and the error of each floating point operation can be retained the same even if the platform of the computation is changed.

The floating point arithmetic, including addition, subtraction and multiplication is covered in this chapter. The rounding error imposed by using floating point arithmetic will be discussed. The concepts of quantization error between IEEE standard and the variant used in this thesis will be introduced.

3.2 Floating Point Number Representation

Every real number can be approximated by a floating point number in the IEEE 754 standard as long as that number is within specific range. The floating point number format is based on scientific notation with limited size for each field. For a normalized floating point number in the IEEE 754 single precision standard where the integer part is always equals to 1, the sign bit is 1 bit in size. The integer part is omitted as it is always equals to 1. The size of fraction part is 23 bit and the size of exponent is 8 bit. The base is always equal to 2 and the total size of a single precision floating point number is 32 bits. In general, an IEEE floating point number F can be expressed as follow:

$$F = (-1)^s \cdot 1.f \cdot 2^{e-b} \tag{3.1}$$

$$b = 2^{e_{size}-1} - 1 \tag{3.2}$$

Where s stands for the *sign* bit, f stands for the *fraction* and e stands for the *biased exponent*. In order to express a negative exponent, there is a *exponent bias* b associated with the exponent field. The actual exponent is the value of the exponent field minus the bias. The value of bias depends on the size of exponent e_{size} as in equation 3.2. The term *significand* represents $1.f$ in which integer field and fraction field are packed together.

For single precision floating point system, the bias is 127 since e_{size} is 8. If the exponent field e is 128, the actual exponent is 128 - 127 = 1. The integer field for most numbers is equal to 1 since they are normalized. Denormalized numbers are indicated by the exponent being 0. In this case, $F = 0.f \times 2^{-126}$ is represented. The above floating point format without denormalized numbers is used throughout this thesis to represent floating point values with arbitrary exponent and fraction sizes.

3.3 Rounding Error

There are four rounding modes in the IEEE floating point standard, namely, round to nearest, round towards $+\infty$, round towards $-\infty$ and round towards zero. The algorithm described above below uses round to zero mode. Under this mode, the result shall be the value closest and no greater in magnitude than the infinitely precise result. Assuming that the length of precision, including the integer field, is p bit, for each of the floating point operations, there will be an absolute error less than 2^{e-p}, where e is the exponent after the normalization of the resulting value. For example, let $p = 3$, the result of the following floating point addition

$$\begin{aligned} & 1.01 \times 2^0 + 1.00 \times 2^{-3} \\ = & \ 1.011 \times 2^0 \\ \approx & \ 1.01 \times 2^0 \end{aligned}$$

will contribute the absolute error of $2^{e-p} = 2^{0-3} = 2^{-3}$

As the answer, after normalization, must greater than 2^e, the relative error corresponding to the answer will be smaller than

$$\epsilon = \frac{2^{e-p}}{2^e} \tag{3.3}$$

$$= 2^{-p} \tag{3.4}$$

When analyzing the rounding error caused by various formulas, relative error is better than absolute error, especially if we need to compare the error of certain equation using different value, it can be estimated the relative error since it is independent to the given value itself. The relative error is always bounded by ϵ, which is referred to as *machine epsilon.*

3.4 Floating Point Number Arithmetic

In this section, the arithmetic of the floating point number is outlined. It focuses on the hardware aspect of the floating point operation using a register transfer language (RTL). The descriptions further assumed that it use IEEE rounding to zero mode when handle inexact number condition.

3.4.1 Addition and Subtraction

Let F_1 and F_2 represent the two single precision floating point numbers, F_{sum} is the sum of these two numbers and F_{minus} is $F_1 - F_2$. As floating point format uses a signed-magnitude representation, the equation

$$F_{minus} = F_1 - F_2 \tag{3.5}$$

can be rewritten as

$$F_{minus} = F_1 + (-F_2) \tag{3.6}$$

So this section will deal with the addition algorithm only. Subtraction is a variation of addition in which the sign bit of F_2 is inverted.

Let F_i be denoted as $(-1)^{s_i} \cdot (1 + 0.f_i) \cdot 2^{e_i - b}$ where s_i, f_i and e_i are the sign field, fraction field and the exponent field in floating point representation respectively and b is the exponent bias.

The IEEE standard requires that the arithmetic operations, including addition and multiplication should be computed as if first produced an intermediate result correct to infinite precision with unbounded range, and then coerced this to fit in the destination's format. However, it is very expensive in terms of the intermediate storage size, if the operands differ greatly in size. Assuming that $p = 3$, $1.11 \cdot 2^{10} + 1.00 \cdot 2^{-2}$ would be calculated as

$$\begin{aligned} x &= 1.110000000000 \cdot 2^{10} \\ y &= 0.000000000001 \cdot 2^{10} \\ x+y &= 1.110000000001 \cdot 2^{10} \end{aligned}$$

which is then rounded to $1.11 \cdot 2^{10}$. It uses 13 bits to store the result which is 4 times the numbers of bits. When the difference of exponent is larger, the size of intermediate result is larger too.

Without using infinite precision for the intermediate result, lengthening the intermediate result by 2 bits at the right is adequate for obtaining properly rounded to zero result. These 2 bits are called guard bit and round bit. The guard bit can guarantee the relative rounding error in the result is less then 2ϵ. The round bit can guarantee the rounding to zero mode is always correct [Gol91]. In general, the sum of F_1 and F_2 is evaluated as shown in algorithm 1, where the symbol ## denotes concatenation of two registers, s_i, e_i and f_i denote the sign field, exponent field and fraction field of the floating point number F_1 respectively. The algorithm further assumed that it used single precision format for F_1 and F_2. However, with some minor modifications, it can be used for arbitrary precision floating point formats. For simplicity, the algorithm does not check any special cases such as negative zero, illegal number and so on. These cases are handled in the hardware implementation of floating point addition.

Algorithm 1 Calculate $F_1 + F_2$ with floating point arithmetic

```
Require: $F_1 = (s_1, e_1, f_1), F_2 = (s_2, e_2, f_2)$
Ensure: $F_{ans} = (s_{ans}, e_{ans}, f_{ans}) = F_1 + F_2$
1: $e_{diff} \leftarrow e_1 - e_2$
2: if $e_{diff} \geq 0$ then
3:     $f_a \leftarrow f_1$, $f_b \leftarrow f_2$, $e_s \leftarrow e_{diff}$
4: else
5:     $f_a \leftarrow f_2$,
6:     $f_b \leftarrow f_1$, $e_s \leftarrow$ 2's complement of $e_{diff}$
7: end if
8: $f_a \leftarrow$ ("001"##$f_a$), $f_b \leftarrow$ "001"##$f_b$
9: $f_b \leftarrow$ shift $f_b$ right with $e_{diff}$ bits
10: if $s_a = 1$ then
11:     $rm_a \leftarrow$ 2's complement of $f_a$
12: end if
13: if $s_b = 1$ then
14:     $rm_b \leftarrow$ 2's complement of $f_b$
15: end if
16: $f_{tmp} \leftarrow rm_a + rm_b$
17: if $f_{tmp}$ is negative then
18:     $f_{tmp} \leftarrow$ 2's complement of $f_{tmp}$, $s_{ans} \leftarrow 1$
19: else
20:     $s_{ans} \leftarrow 0$
21: end if
22: find the leading one of $f_{tmp}$, shift $f_{tmp}$ left until $f_{tmp}(msb) = 1$,
23: $e_{ans} \leftarrow e_a$ - number of bits shift to left. $msb$ is the location of most
    significant bit
24: omit the integer part, $f_{ans} = f_{tmp}(msb - 1...0)$
```

3.4.2 Multiplication

Multiplication is simpler than addition assuming that the fixed point multiplier is provided. The product of F_1 and F_2, where both F_1 and F_2 are normalized floating point numbers, is evaluated as in algorithm 2. For simplicity, the algorithm does not check any special cases such as negative zero, illegal number and so on. These cases are handled in the hardware implementation of floating point multiplication.

Algorithm 2 Calculate $F_1 \times F_2$ with floating point arithmetic

```
Require: $F_1 = (s_1, e_1, f_1), F_2 = (s_2, e_2, f_2)$
Ensure: $F_{ans} = (s_{ans}, e_{ans}, f_{ans}) = F_1 \times F_2$
 1: $s_{ans} \leftarrow s_1 \oplus s_2$
 2: append 1 bit "1" to $f_1$ and $f_2$ at left as the hidden integer field
 3: $v_1 \leftarrow$ "1"##$f_1$
 4: $v_2 \leftarrow$ "1"##$f_2$
 5: do fixed point unsigned multiplication $mc \leftarrow v1 \times v2$
 6: $r_{e1} \leftarrow e_1 + e_2 - b$
 7: shift $mc$ to left until msb of $mc$ is 1
 8: $e_s \leftarrow$ number of bit shifted to left
 9: $e_{ans} \leftarrow r_{e1} - e_s$
10: $f_{ans} \leftarrow mc(44...22)$
```

3.5 Summary

This chapter described the fundamental concepts of the floating point numbers. It introduced various number formats and operations including addition, subtraction and multiplication. It further discussed the effect of rounding errors for floating point operation.

Chapter 4

FLY - Hardware Compiler

4.1 Introduction

This chapter describes the implementation details of *fly* compiler. *Fly* compiler translates a Perl-like algorithm description into synthesizable VHDL code. *Fly* supports most elementary constructs such as conditional branching and looping. This chapter begins with the syntax of *fly* programming language. For each constructs, the implementation will be described using a greatest common divisor as an example. Summary is given at the end of the chapter.

4.2 The *Fly* Programming Language

The syntax of the *fly* programming language is modeled on Perl, with extensions for parallel statements and the host/FPGA interface. Table 4.1 shows the main elements of the *fly* language with simple examples. The formal grammar definition is in Appendix A.

Using Perl-like description has its advantages. It facilitates the compatibility between software simulation and hardware implementation. Any algorithm that can be described in *fly* without using parallel constructs, would be able to simulate on Perl by executing the script without any modification. In addition, it is easier for designers to learn the *fly* other than HDL based languages. It also minimizes the error due to the translation of software simulation version to hardware datapath description.

Constructs	**Elements**	**Example**
assignment	var = expr;	$\$var1 = \$tempvar;$
parallel statement	[{ ... } { ... } ...]	$[\ \{\$a = \$b;\}\ \{\ \$b = \$a * \$c;\}\ ]$
expression	val op expr; valid ops: $*, /, +, -, .*, .-, .+$	$\$a = \$b. * \$c;$
loop	while (rel) { ... }	while $(\$x < \$y)$ { $\$a = \$a + \$b; \$y = \$y + 1;\}$

if-else	if (cond) { ... } else { ... }
	
	if (cond) { ... }
		if $(\$i <= \$j)$ { $\$a = \$b;$}
		else $\{a = c;\}$
		if $(\$i > \$j)$ $\{\$i = \$i + 1;\}$

cond	expr rel expr valid rels: $>, <, <=, >=, ==, !=$	$\$i >= \$c$
built-in function	&read_host(..)	$\$i = \&read\_host(255)$
comment	# comment	*#this line is comment*

Table 4.1: Main elements of the *fly* language.

The *fly* program for a greatest common divisor (GCD) CO-processor, which will be used as an example in the rest of this chapter is given in listing 4.1:

The program uses most elements of the *fly* language and system including the host interface, while loops, if-else branches, integer arithmetic, parallel statements and register assignment. This example will be used in the rest of this chapter to illustrate the translation process.

4.3 Implementation details

4.3.1 Compilation Technique

Programs in the *fly* language are automatically mapped to hardware by using the technique described by Page [Pag96]. The compiler generates synthesizable VHDL code instead of a netlist, simplifying code generation and making the

Listing 4.1: Greatest Common Divisor

```
{
    $s = $din[1]; $l = $din[2];
    while ($s != $l) {
        $a = $l - $s;
        if ($a > 0) {
            $l = $a;
        }
        else {
            [ {$s = $l;} {$l = $s;} ]
        }
    }
    $dout[1] = $l;
}
```

output portable to many different FPGA and ASIC design tools. Furthermore, as an intermediate language, VHDL enables the logical optimization of the synthesis tool to be included in the design flow.

In order to facilitate the support of control structures, each statement has a `start` and `end` signal that specifies temporally when the execution of one statement begins and ends. By connecting the `start` and `end` signals of adjacent statements together, a one-hot state machine is constructed that serves as the control flow of the *fly* program.

Fly is written in the Perl programming language [WCO00]. Perl is a language with very good portability, string handling facilities and libraries. The *fly* system's source code in Appendiex B is made simpler and concise as a result of using Perl. Development of the *fly* compiler was also facilitated using a parser generator called `Parse::RecDescent` [Con01] which generates a Perl based recursive descent parser from a description of the grammar of the target language.

4.3.2 Statement

A program is a sequence of statements, each statement being either an assignment, sequences of statements to be executed in parallel, if-else, or a while loop. Each statement has an associated start and end signal, and a sequence of statements is constructed by connecting the individual statement's start and end signals together. A statement is said to be enabled if its start signal is high during the rising edge of the (global) clock.

The start signal of the entire program is generated by the host interface. For example, the first statement of the GCD program that is enabled is the assignment `$s = $din[1];`. The end signal of this statement is connected to the start signal of the next statement, namely `$l = $din[2];`. In this case, the end signal is generated from the start signal by delaying it one clock cycle using a D-type flip flop.

Eventually, the last statement of the program `$dout[1] = $l;` will be enabled, and after it has been executed (i.e. its end signal is asserted), the execution of the program is completed.

4.3.3 Assignment

Assignments are implemented simply by asserting the destination register's enable signal when its associated statement is enabled. If a variable is the target of an assignment from more than one statement, a multiplexer and encoder is used to select the according source value.

For example, if a program has two assignments to the same variable i.e. `$l = $a` and `$l = $s`, and if the associated start and end signals are `$start1`, `$end1` and `$start2`, `$end2` respectively, the circuit in Figure 4.1 is generated.

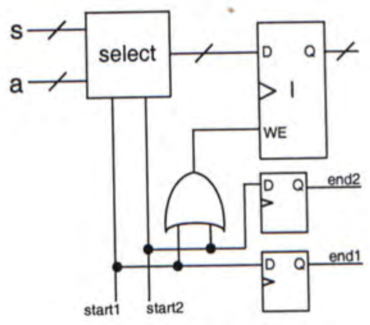

Figure 4.1: Circuitry used to handle multiple assignments to the same variable. This is the circuit which results from a program with two assignments `$l=$a` and `$l=$s`.

4.3.4 Conditional Branch

If-else statements have both a condition and two statements. The start signal of the if-else statement is routed to the appropriate block of statements depending on the condition. Figure 4.2 shows the resulting circuit for the statement `if ($a > 0) ... else ...` . The end signals of both blocks are or'd together to produce the end signal of the if-else statement.

4.3.5 While

The end signal of a while statement must be conditionally fed back to the start signal for the statement block. The circuit corresponding to the while loop in the GCD algorithm is shown in Figure 4.3.

4.3.6 Parallel Statement

In the GCD example, a parallel statement is used to swap the `$s` and `$l` variables. As shown in Figure 4.4, each sequential block enclosed by parallel brackets [ ] will start execution at the same time. The parallel block will end when all sequential blocks give an end signal. A statement will only have an active end signal for a single cycle, so flip-flops (labelled "FF" in the figure)

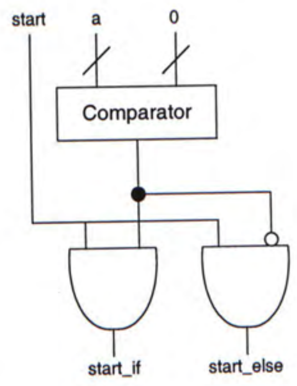

Figure 4.2: Circuitry for if-else statements. This is the circuit which results from the statement `if ($a > 0) ... else ... .`

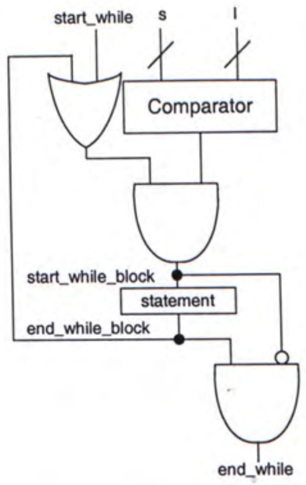

Figure 4.3: Circuitry for while statements. This is the circuit which results from the statement `while ($s != $l)`.

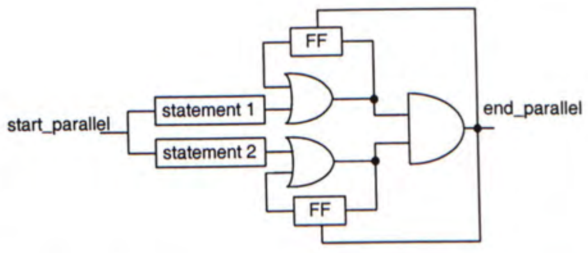

Figure 4.4: Circuitry for parallel statements.

are added to determine when all statements have finished. If all the flip flops are set, it indicates the end of the parallel statement and they will be cleared at next clock cycle.

4.4 Development Environment

4.4.1 From Fly to Bitstream

Although the interface is easily adaptable to any reconfigurable computing card, the *fly* system currently only supports the Pilchard reconfigurable computing platform [LLC+01]. Pilchard uses a DIMM memory bus interface instead of a conventional PCI bus. The advantage of the memory bus is that it acheives much improved latency and bandwidth over the standard PCI bus.

The translated output of a *fly* program is interfaced with a generic Pilchard core written in VHDL. A shell script, automatically invoked by the *fly* system, includes the libraries and invokes the programs which are required to compile the VHDL representation of the user's program to a bitstream. The bitstream is also automatically downloaded to the FPGA and the host interface program automatically invoked. Thus the entire compilation and execution process are hidden from the user.

4.4.2 Host Interface

To enhance the flexibility of host/FPGA interface, two interfaces were developed namely register and BlockRAM approach. Each approach suits for certain application.

Registers can be used to transfer data between the FPGA and host. The architecture of host interface is shown in Figure 4.5 In normal operation, the host processor would initialize values in `$din[1]` to `$din[x]`, and then start execution of the FPGA based coprocessor by performing a write cycle to the

`$din[0]` register. The write cycle causes the start signal of the first statement in the FPGA to be asserted. The software then polls the least significant bit of `$din[0]` which is connected to the end signal of the last statement. When execution on the FPGA finishes, the least significant bit of `$din[0]` is set and the program can read values returned by the hardware by reading the appropriate registers.

By using the register interface, the fly core can be adopted to different FPGA and ASIC products. The data can be fetched immediately without address decoding cycles inside the FPGA. However, the register approach cannot support streaming data which is common in DSP design. The number of argument passing to the fly core is limited since register will use the resource of FPGA cells.

Another approach to the host/FPGA interface is using the BlockRAM [Xil01] feature which is available on Xilinx Virtex devices. BlockRAM is dual port configured and one side of port is connected to the host bus while the other side is connected to the fly core as shown in Figure 4.6. Two built-in functions `read_host()` and `write_host()` are introduced to access the data in the BlockRAM. The handshaking is similar to the register approach. The address 0 in the BlockRAM is used for handshaking and will trigger the start of FPGA coprocessor during a write cycle is issued on address 0. When the FPGA finishes the execution, it will return 1 once the host performs a read cycle on address 0.

Since the BlockRAM does not consume the logic resources in the FPGA, it has advantages in area and performance over a large number of registers. In addition, the interface clock and the core clock can be of different frequencies. This can enhance the flexibility to reach specific design constraints. It is possible that the core clcok can run faster then the interface clock when two clocks are provided. It also supports data streaming such that the processor can provide data to the FPGA and the FPGA can return the result at the

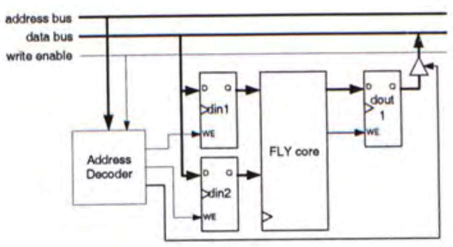

Figure 4.5: Circuitry for the host to FPGA interface using register

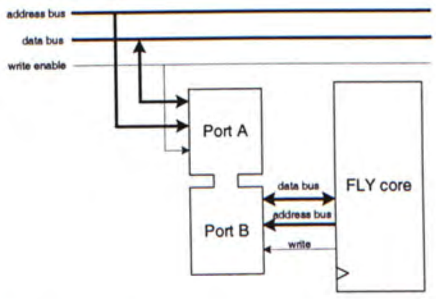

Figure 4.6: Circuitry for the host to FPGA interface using dual-port BlockRAM

same time since BlockRAM is dual portted

4.5 Summary

In this chapter, the Perl programming language was used to develop a powerful yet simple hardware compiler for FPGA design. Unlike previous compilers, *fly* was designed to be easily modifiable to facilitate research in hardware languages and code generation. Since *fly* is tightly integrated with the hardware design tools and implementation platform, designers can operate with a higher level of abstraction than they might be accustomed to if they used VHDL. An example of a GCD coprocessor was given. Development time was significantly reduced since deubgging can be done through the simulation of the program.

Chapter 5

Float - Floating Point Design Environment

5.1 Introduction

With the increasing size of FPGA devices, implementing floating point arithmetic on FPGAs are now possible. However, as the size of the FPGA is still limited, a carefully designed floating point implementation is essential. In custom hardware designs, there are always trade-offs between conflicting requirements of performance, area and quantization error to be addressed. For example, area can usually be reduced if a larger quantization error is allowed for a hand-held application. It would be desirable to allow a program to automatically determine the minimum exponent and fraction sizes required for each signal to reach some user-specified quantization error. A floating point library called *float* is presented to enable users to optimize the design. In addition, a library which can generate arbitrary sized floating point adders and multipliers was developed to facilitate the FPGA-based floating point applications.

The first section will discuss the software aspect of this system. An example using floating point tools to develop and optimize a digital sine-cosine compiler is presented. To generate a arbitrary sized of floating point operator, a Perl program has been developed as a VHDL generation module and will be introduced in Section 5.4.

5.2 Floating Point Tools

Float consists of the following modules:

- A Perl class called *float* for the representation of floating point numbers. Simulation of the effect of low precision floating point operations is performed using this class.
- An optimizer which minimizes a cost function by adjusting the floating point format of the *float* variables in an algorithm function.
- A VHDL generation module which produces synthesizable VHDL code.
- *float* is compatible with *fly* compiler described in the previous chapter.

Figure 5.1 illustrates the *float* design flow. A designer begins by writing a Perl function, hereafter referred to as the algorithm function, to represent the algorithm to be implemented. All variables used in the algorithm are *float* objects, where *float* is a Perl class that is capable of representing a floating point value under arbitrary precision. The function takes a number of *float* variables as input and produces a number of *float* variable as the output.

By varying the precision of the *float* objects, the optimizer minimizes a cost function which is a weighted sum of the quantization error of the outputs of the algorithm function and the circuit size of the resulting implementation. In order to determine the outputs, a set of test input vectors are required. The algorithm function is executed with the test vectors as inputs, *float* operators being used to perform computation. The class computes the result using both IEEE double precision and the user-specified precision. These two results are then used to compute the quantization error, with an underlying assumption that the IEEE double precision result is without quantization error, and the

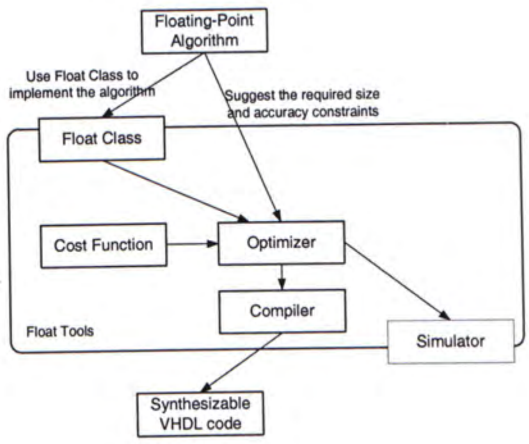

Figure 5.1: Floating point algorithm design flow.

float precision is less than double precision. Given the precision of a floating point operator, the cost function also includes a term which is an estimation of the circuit size.

Once the optimizer has determined a suitable precision for each variable in an algorithm function, the same function will pass to *fly* compiler which can output synthesizable VHDL code for implementing the algorithm on the FPGA. The precision of variables are provided by the optimizer, the *fly* simply instantiates components with the required precision from a floating point operator module generator library.

5.2.1 Float Class

To describe hardware that utilizes variable precision floating point computations, a class called *float*, which facilitates the simulation of arbitrary precision floating point arithmetic was developed. Perl is a modern high level programming language which offers improved productivity over traditional languages such as C. The following features of Perl were important to the design of the *float* system:

- Perl supports objects which are used to abstract the details of variable wordlength operators.

- Perl supports operator overloading so that if x and y are *float* objects, one can write x + y instead of x.add(y).
- Perl has strong memory management and string manipulation facilities making it easy to construct VHDL module generators.
- Perl is very portable so the *float* design environment can run on many platforms including Unix, Linux and Windows.
- There are many open source software libraries available for Perl.

The *float* object provides several methods for interrogation of its parameters and computation. The main ones are:

- `add()`, `multiply()`:

  The `add()` and `multiply()` methods will add/multiply two *float* objects together at their specified precision, creating a new *float* object. If the two floating point numbers have a different number of exponent bits, the output will have an exponent being the larger one of the two. Similarly, if the two numbers have different fraction sizes, the output will have fraction bit length equal to the larger one of the two input bit lengths. Overloading is used so that the $+$ and $*$ operators will invoke the `add()` and `multiply()` methods respectively.

  Apart from the arbitrary precision result, another IEEE 754 double precision floating point calculation is also computed. This value is used as a reference value for computing quantization error. Furthermore, the maximum and minimum range of this reference value is stored in the object for computation of the minimum exponent value which is required.

- `setExponentSize()`, `setFractionSize()`:

  The `setExponentSize()`, `setFractionSize()` methods will set the precision of a *float* object. For `setFractionSize()`, the value of the object will be truncated if the fraction size will be smaller than original.

- `setValue()`, `getValue()`:

  These two methods are used to retrieve and write the value represented by the *float* object. Two values are stored, the IEEE double precision reference value, and the arbitrary precision value.

- `getQERR()`:

  Both the arbitrary size floating point number and reference double precision floating point value are stored in the *float* object. `getQERR()` returns their difference.

5.2.2 Optimization

Although any measure of accuracy could be used, average quantization error, QERR, in decibels is used in this dissertation. QERR is computed as follows:

$$\mathrm{QERR} = \frac{1}{N}\sum_{i=1}^{N} 20\log\left|\frac{\mathrm{out}_i - \mathrm{ref}_i}{\mathrm{ref}_i}\right| \tag{5.1}$$

where out_i are the outputs and ref_i are the corresponding double precision reference outputs.

The total circuit area is determined by summing the area estimated for each operator. Operator area is estimated from the precision of the *float* class, assuming a Xilinx Virtex-E series FPGA [Xil01]. Although the area estimation is based on a specific reconfigurable computing platform, optimization using these measures should lead to reasonable area estimates on other platforms.

The area in Virtex slices [Xil01] occupied by floating point adder is estimated based on the fraction size and exponent size. Nonlinear regression has been applied to model the relation between area and precision using adaptive nonlinear least-squares algorithm purposed by J.E. Dennis et al [JGW81]. The architecture of floating point adder, as discussed in section 5.4, has linear relationship of exponent size and fraction size. The initial relationship is modeled as follows:

$$\text{add\_area} = a \times \text{ebits} + b \times \text{fbits} + c \tag{5.2}$$

where ebits is the number of exponent bits in the *float* representation and fbits is the number of fraction bits.

To determine the parameters a, b and c, different precision of floating point adders were implemented on FPGA and the slices used was collected as shown in chapter 7 which acts as sample data point in the nonlinear regression algorithm. The result was further fine-tuned and the best approximation was found that $a = 6$, $b = 12$ and $c = 0$.

Similarly, the area occupied by a floating point multiplier is modeled by the equation 5.3, fraction size is contributed large portion of slices because larger value of fraction size means larger fixed point multiplier is used.

$$\text{mul\_area} = a \times \text{ebits} + \ b \times \text{fbits}^2 + c \tag{5.3}$$

After applying nonlinear regression algorithm and fine-tuning, the best approximation was $a = 8$, $b = 0.47$ and $c = 230$.

The cost function is computed from the QERR and circuit area is measured using the equation 5.4:

$$f_{cost} = a \times \left(\sum_i \text{add\_area}_i + \sum_j \text{mul\_area}_j \right) + b \times \text{QERR} \tag{5.4}$$

where a and b are non-negative weightings and i and j sum over all the add and multiply operators in the algorithm function respectively.

The optimizer uses the Nelder-Mead [NM65] method to minimize the cost function (without requiring the computation of derivatives) by adjusting the precisions of *float* variables in the algorithm function. The designer can adjust a and b in equation 5.4 to weigh the relative importance of area and QERR. For example, if the designer needs a very accurate result and circuit area is not critical, a large value of b can be used.

The optimization procedure is outlined as follows:

1. Change the precisions of *float* variables (using Nelder-Mead).
2. Simulate the algorithm function at the specified precision using user-supplied input data.
3. Compare the result with the reference result and compute the cost function.
4. Repeat until the optimization terminates.

5.3 Digital Sine-Cosine Generator

Digital sine-cosine generators [Mit98] have a number of applications, such as the computation of discrete Fourier transform and in certain digital communication systems, such as in future Hiperlan systems [ETS96] for high performance wireless indoor communication. Let $s1_n$ and $s2_n$ denote the two outputs of a digital sine-cosine generator, the outputs at the next sample can be computed using the following formula:

$$\begin{bmatrix} s1_{n+1} \\ s2_{n+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \cos(\theta)+1 \\ \cos(\theta)-1 & \cos(\theta) \end{bmatrix} \begin{bmatrix} s1_n \\ s2_n \end{bmatrix} \tag{5.5}$$

Equation 5.5 will be used as one of the example of *float* application in this chapter, with $\cos\theta = 0.9$. Its algorithm function can be described by the Perl code listing 5.1:

Listing 5.1: Digital sine cosine generator

```
$cos_theta = new Float(8, 23, 0.9);
$cos_theta_p1 = new Float(8, 23, 1.9);
$cos_theta_m1 = new Float(8, 23, -0.1);
$s1[0] = new Float(8, 23, 0);
$s2[0] = new Float(8, 23, 1);
for ($i = 0; $i < 50; $i++){
    $s1[$i+1] = $s1[$i] * $cos_theta + $s2[$i]
        * $cos_theta_p1;
    $s2[$i+1] = $s1[$i] * $cos_theta_m1 + $s2[$i]
        * $cos_theta;
}
```

This algorithm function first declares the variables used via *float* object instantiations, each object being specified to have an 8-bit exponent and a 23-bit fraction in this example. The initial value of the variable is also defined in the *float* constructor, with $s1$ and $s2$ being initialized to 0 and 1 respectively. The update values of $s1$ and $s2$ are derived using the floating point operators provided by the *float* class via overloading.

This algorithm function can be passed to different components for processing. Normally, a set of input vectors is specified for the algorithm function, but since this particular function is an oscillator with no inputs, the time domain response is computed via the loop in the algorithm function.

The simulator can be used to determine the result and the optimizer can determine a suitable precision format for each of the five *float* objects in the algorithm function, which minimizes the following optimization. The inner part of the algorithm function can be given to *fly* compiler to produce VHDL code. Finally, the VHDL output can be used for simulation and/or implementation on a reconfigurable computing platform.

5.4 VHDL Floating Point operator generator

The module library was implemented in Perl and currently supports two operators, namely multiplication and addition. Thus one can use the module library to generate operators with arbitrary precision. Operators are pipelined for high throughput.

5.4.1 Floating Point Multiplier Module

The Algorithm 2 in chapter 3 was implemented as a VHDL module and the corresponding datapath of the parameterized floating point multiplier is shown in Figure 5.2 using the mentioned algorithm. It has 4 stages with 8 clock cycles pipelining to evaluate the product of the given numbers.

In the first stage, the steps 1 and 2 are implemented by padding one to the fraction to produce the significand and calculating the sign bit using the XOR of the sign bits. This stage uses 1 clock cycle.

In the second stage, steps 3 - 5 are implemented. The significands $v1$ and $v2$ will be multiplied. The most significant bits of the product, ranged from $2 \times fsize - 1$ to $fsize - 1$, where $fsize$ is the size of fraction, is stored to the register mc. Since both $v1$ and $v2$ have leading 1 at most significant bit, the leading 1 of mc is at its first two most significant bits. This observation can simplify the normalization process as described below.

The intermediate exponent will be calculated by considering two cases. If the leading 1 of mc is located at the most significant bit, mc is a normalized number and the final exponent would be $e1 + e2 + 1 - bias$. This exponent is stored as $ec1$. If the leading 1's of mc is located at the next most significant bit, mc should be normalized by shifting 1 bit to left, and the exponent would be $e1 + e2 - bias$. This exponent is stored as $ec0$. Since at mc is not determined, both $ec0$ and $ec1$ are stored to save time. Since a fixed point multiplier is involved, the latency of this stage is 5 clock cycles.

The third stage does steps 6 - 9. As mc is evaluated, e_{ans} is determined by the most significant bit of mc. The mc will shift left appropriately so that the most significant bit of mc is 1. The result of normalization will be stored at $mc0$. This stage takes 1 clock cycle.

The forth stage implements steps 10. It omits the integer part of $mc0$ and stores the remaining fraction as f_{ans} and the product is returned. This stage uses 1 clock cycle. Extra logic is required to complete the floating point multiplier. These logics include zero checking and infinity handling. They are omitted in the Figure 5.2 for simplicity but implemented in the module generator.

5.4.2 Floating Point Adder Module

The datapath of a parameterized floating point adder/subtractor is shown in Figure 5.3 is the hardware implementation of algorithm 1. Similar to floating point-multiplier, it has 4 stages to evaluate the product of the given numbers. Each stage uses 1 clock cycle. A subtractor is implemented by flipping the sign bit of the second operand and is not shown in the figure.

The first stage implements steps 1 - 7. $ediff$, which is the difference of e_1 and e_2 is calculated and if $ediff$ is negative, f_1 and f_2 will be swapped. After swapping, F_a is the number with larger exponent and the other one is called F_b.

The second stage implements step 8 - 15. The correct significands are evaluated from the given fractions. $fraction\_b$ will be aligned such that both fraction share the same intermediate exponent, namely, $exponent\_a$. The significands are not in 2's complement format, so conversion is necessary if the corresponding sign bit is set. The intermediate exponent, $exponent\_a$, is propagated to $ea2$. The intermediate significands are stored in register rm_a and rm_b.

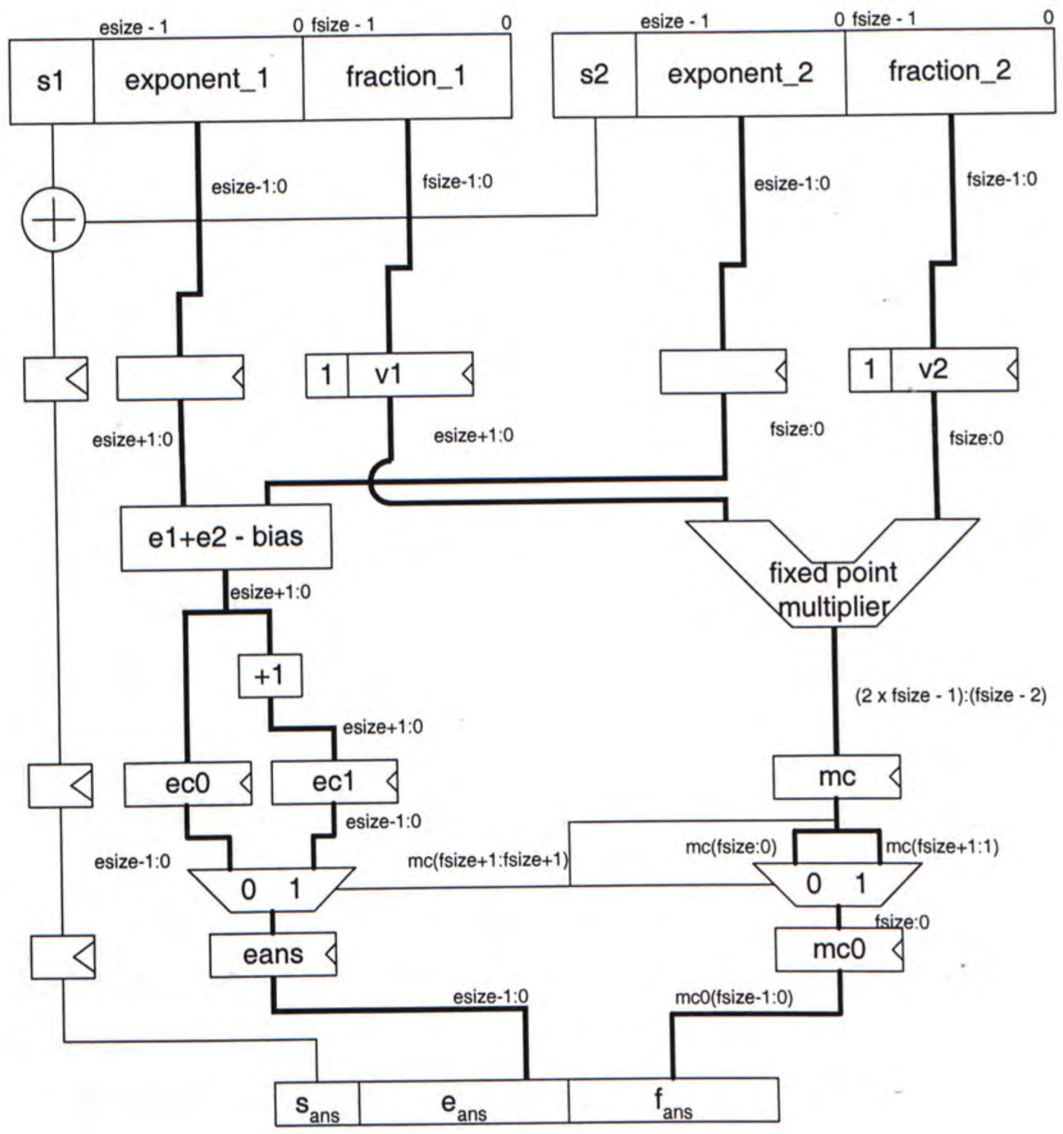

Figure 5.2: Parameterized Floating Point multiplier datapath

The third stage does the steps 16 - 21. The significands are added. The sum of rm_a and rm_b will be stored to register f_{tmp}. The value of f_{tmp} should be an unsigned number it is returned. So conversion is necessary if f_{tmp} is negative. The sign bit is retrieved from the adder and stored to register $sa1$. In addition, the intermediate exponent, which is $exponent\_a$, is propagated to $ea3$.

The last stage, steps 22 - 24, is normalization and rounding. A priority encoder is used to determine the location of leading 1 at register f_{tmp}. The final exponent, namely e_{ans}, is calculated by $e_{ans} = ea3 -$number of bits shift to left$+$ $ebias$. f_{ans} is obtained by shifting f_{tmp} to left such that the most significand bit of f_{tmp} is 1, and the leading one is omitted. s_{ans} is propagated from $sa1$. Rounding is a truncation in round to zero mode so it is done implicitly when the result is packed in the f_{ans} register.

Like the multiplier, extra logic is required to complete the floating point adder. These include zero checking and infinity handling. They are omitted in Figure 5.3 for simplicity, but implemented in the module generator.

5.5 Application to Solving Differential Equations

The floating point generation module and *fly* compiler were used to solve the ordinary differential equation

$\frac{dy}{dt} = \frac{(t-y)}{2}$ over $t \in [0,3]$ with $y(0) = 1$ [MF99].

The Euler method was used so the evolution of y is computed by $y_{k+1} = y_k + h\frac{(t_k - y_k)}{2}$ and $t_{k+1} = t_k + h$ where h is the step size.

The following *fly* program implements the scheme, where h is a parameter sent by the host, as shown in listing 5.2.

In each iteration of the program, the evolution of y is written to the block

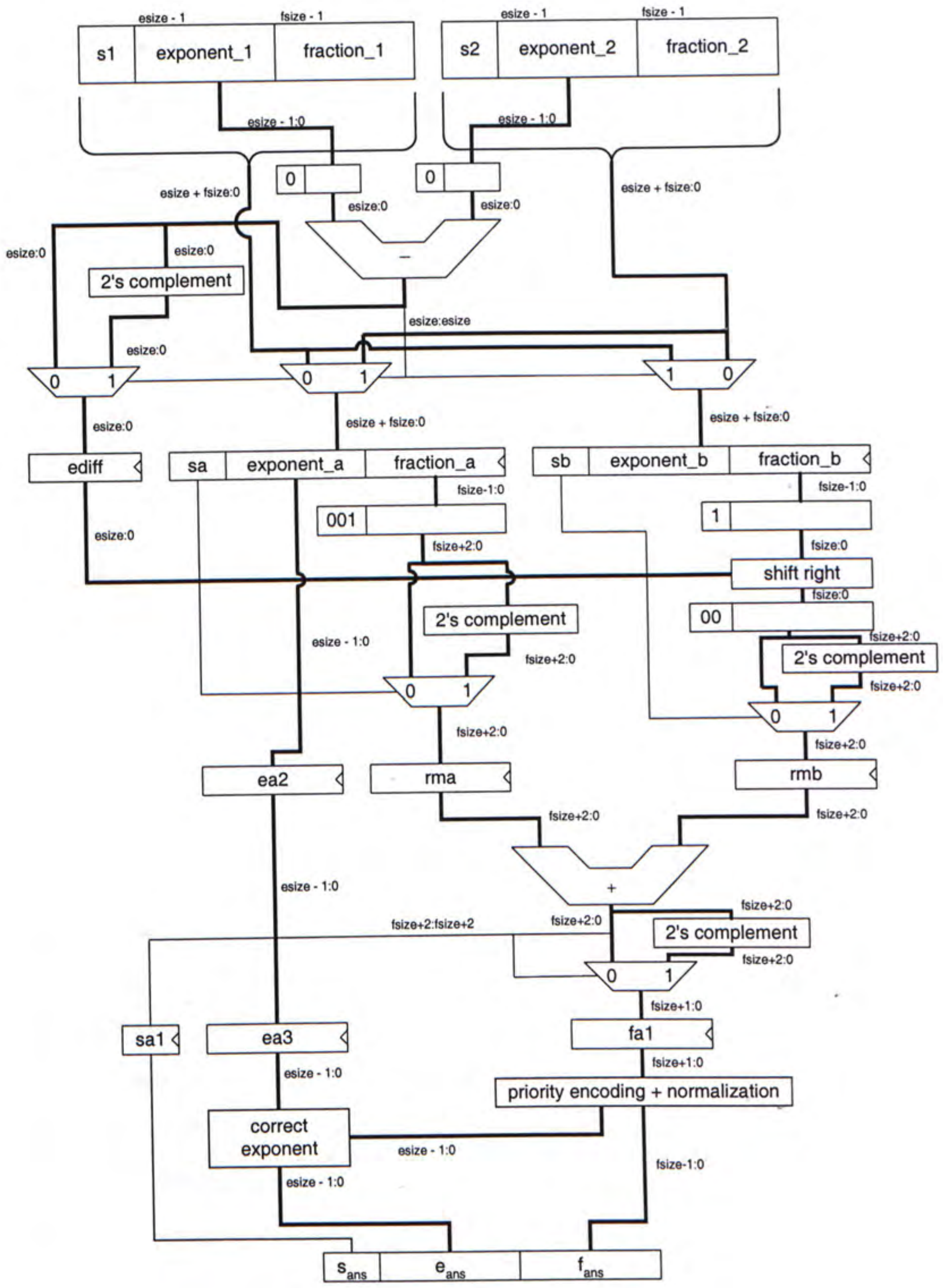

Figure 5.3: Parameterized Floating Point adder datapath

Listing 5.2: Ordinary Differentiable Equation Solver

```
{
    $h = &read_host(1); #fetch
    [
      {$t = 0.0;} {$y = 1.0;} {$dy = 0.0;}
      {$onehalf = 0.5;} {$index = 0;}
    ] # parallel assignment
    while ($t < 3.0) {
        [ {$t1 = $h .* $onehalf;} {$t2 = $t .- $y;} ]
        [ {$dy = $t1 .* $t2;} {$t = $t .+ $h;} ]
        [
          {$y = $y .+ $dy;}
          {$index = $index + 1;}
        ]
        $void = &write_host($y, $index);
        #write host
    }
}
```

RAM via a `write_host()` function call and a floating point format with 1 sign bit, 8-bit exponent and 23-bit fraction was used throughout. The floating point format can, of course, be easily changed. Parallel statements in the main loop achieve a 1.43 speedup over a straightforward serial description.

5.6 Summary

The *float* environment for the rapid prototyping of floating point digital system was described. These tools enable the designers to concentrate on higher level algorithmic issues thus increasing their productivity and being able to explore more of the design space in a give time. A digital sine-cosine generator and a differentiable equation solver were as an example of using *float*. The module geneartor is packaged in Perl so as to allow easy interface with the current development tools.

The *float* environment extends the capabiltiy of *fly* compiler in which floating point operator is now supported. With a single Perl description, the algorithm function can be optimized and implemented using the provided design environment with ease.

Chapter 6

Function Approximation using Lookup Table

This chapter discusses an efficient table lookup generation system for supplementing a hardware description language (HDL). In particular, an implementation of the Symmetric Table Addition Method (STAM) which acts as a module generator for any differentiable functions is described. This module generator was integrated with *fly* compiler to produce a very flexible design environment which allows the specification of arbitrary functions in a high level manner. The environment is used to develop a coprocessor for the computation of the N-body problem, and the designer productivity is much higher than a typical designer using VHDL.

6.1 Table Lookup Approximations

6.1.1 Taylor Expansion

The main idea behind the table lookup approximation algorithms is the Taylor Expansion. If a function $f(x)$ has continuous derivatives up to $(n+1)^{th}$ order, then it can be expanded as

$$
\begin{aligned}
f(x) &= f(a) + f'(a)(x-a) + \frac{f''(a)(x-a)^2}{2!} + \cdots + \frac{f^{(n)}(a)(x-a)^n}{n!} + R_n \\
&= \sum_{i=0}^{n} \frac{f^{(i)}(a)(x-a)^i}{i!} + R_n \qquad (6.1)
\end{aligned}
$$

where

$$
\begin{aligned}
R_n &= \int_a^x f^{(n+1)}(u)\frac{(x-u)^n}{n!}du \\
&= \frac{f^{(n+a)}(\xi)(x-a)^{n+1}}{(n+1)!} \quad \text{for} \quad a < \xi < x
\end{aligned}
$$

To reduce the required hardware resources and/or computating time, only the first few terms in the Taylor series are used to approximate the function in practice. The selection of a will affect the error introduced and a carefully selected a can be used to introduce symmetry in the lookup table as explained later.

6.1.2 Symmetric Bipartite Table Method (SBTM)

The SBTM uses the first two terms of the Taylor series to approximate a function $f(x)$ as $\widetilde{f(x)}$ [SS97]. In the SBTM, two lookup tables are constructed and the precision of the output is maximized.

Assume that the n-bit input, x, of the function to be approximated ranges in $[0, 1)$. It is first partitioned into 3 segments as shown in Fig 6.1 where $x = x_0 + x_1 + x_2$.

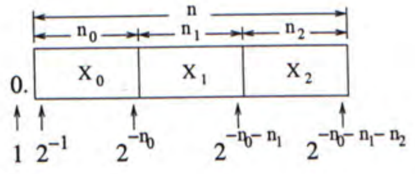

Figure 6.1: Input partition of SBTM.

The ranges of x_i are:

$$\begin{aligned} 0 &\le x_0 &&\le 1 - 2^{-n_0} \\ 0 &\le x_1 &&\le 2^{-n_0} - 2^{-n_0-n_1} \\ 0 &\le x_2 &&\le 2^{-n_0-n_1} - 2^{-n_0-n_1-n_2} \end{aligned}$$

Two lookup tables which return the value a_0 and a_1 are then constructed. The sum of these two values will be the approximated result of the function.

$$\begin{aligned} \widetilde{f(x)} &= a_0(x_0, x_1) + a_1(x_0, x_2) \\ &\approx f(x_0 + x_1 + x_2) \end{aligned} \tag{6.2}$$

We first select mid points in the ranges of x_1 and x_2:

$$\begin{aligned} \delta_1 &= (2^{-n_0} - 2^{-n_0-n_1})/2 \\ &= 2^{-n_0-1} - 2^{-n_0-n_1-1} \\ \delta_2 &= (2^{-n_0-n_1} - 2^{-n_0-n_1-n_2})/2 \\ &= 2^{-n_0-n_1-1} - 2^{-n_0-n_1-n_2-1} \end{aligned} \tag{6.3}$$

Let $a = x_0 + x_1 + \delta_2$ and use the first two terms of the Taylor Expansion:

$$\begin{aligned} f(x) &= f(x_0 + x_1 + x_2) \\ &\approx f(x_0 + x_1 + \delta_2) + f'(x_0 + \delta_1 + \delta_2)(x_2 - \delta_2) \\ &= a_0(x_0, x_1) + a_1(x_0, x_2) \\ &= \widetilde{f(x)} \end{aligned} \tag{6.4}$$

Not all bits from a_1 are required to be in the table as the carefully selected δ_2 results in a large number of leading 0s or 1s in the a_1 table. Since δ_2 is located in the center of x_2's range,

$$|x_2 - \delta_2| \leq \frac{x_{max}}{2} \quad \Rightarrow \quad |x_2 - \delta_2| < 2^{-n_0-n_1-1} \tag{6.5}$$

The upper bound of a_1 is

$$\begin{aligned} |a_1(x_0, x_2)| &= |f'(x_0 + \delta_1 + \delta_2)(x_2 - \delta_2)| \\ &< |f'(\xi_1)|2^{-n_0-n_1-1} \end{aligned} \tag{6.6}$$

where

$$\xi_i \in [0,1) | (\forall x \in [0,1) | f^{(i)}(\xi_i) > f^{(i)}(x))$$

6.1.3 Symmetric Table Addition Method (STAM)

The logic in SBTM is simple and two tables are required. The STAM algorithm uses more tables with smaller size to significantly reduce the overall memory required [SS99b].

As shown in Fig 6.2, the n-bit input is partitioned into m segments instead of 3 in SBTM. The input is now $x = \sum_{i=0}^{m-1} x_i$.

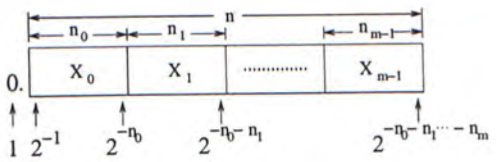

Figure 6.2: Input partition of STAM.

The ranges of x_i are shown here:

$$\begin{aligned} 0 &\leq x_0 &&\leq 1 - 2^{-n_0} \\ 0 &\leq x_i &&\leq 2^{-p_{i-1}} - 2^{-p_i} \end{aligned} \tag{6.7}$$

where $p_i = \sum_{k=0}^{i} n_k$ and δ_i is defined as following:

$$\delta_i = (2^{-p_{i-1}} - 2^{-p_i})/2 \tag{6.8}$$

To apply the Taylor approximation, let the $a = x_0 + x_1 + \sum_2^m \delta_i$. The approximation function is now:

$$\begin{aligned}
\widetilde{f(x)} &= f(x_0 + x_1 + \sum_2^m \delta_i) + f'(x_0 + \delta_1 + \sum_2^m \delta_i)(\sum_2^m x_i - \sum_2^m \delta_i) \\
&= a_0(x_0, x_1) + \sum_2^m a_{i-1}(x_0, x_i) \qquad (6.9)
\end{aligned}$$

where

$$a_{i-1}(x_0, x_i) = f'(x_0 + \delta_1 + \sum_2^m \delta_k)(x_i - \delta_i) \quad 2 \le i \le m$$

The error analysis of STAM is very similar to the SBTM algorithm. The constraints for the parameter configuration are:

$$2n_0 + n_1 \le p_f + log_2(|f''(\xi_2)|) \tag{6.10}$$

$$g \le 2 + log_2(m-1) \tag{6.11}$$

6.1.4 Input Range Scaling

The analysis above are all based on the input range $[0, 1)$. Both SBTM and STAM can be adapted to other input ranges. But this requires some transformations when generating the table contents. The transformation is done by dividing the input range evenly for all the possible input patterns.

For an n-bit input x, let $\widehat{x}$ be the integer value of the bit pattern assuming the decimal point is on the right of the LSB. If the input range is $[x_{min}, x_{max})$, then

$$\begin{aligned}
\frac{x - x_{min}}{x_{max} - x_{min}} &= \frac{\widehat{x}}{2^n} \\
\Rightarrow x &= \frac{\widehat{x}}{2}(x_{max} - x_{min}) + x_{min} \qquad (6.12)
\end{aligned}$$

Let this be the transform function $t(\widehat{x})$. The range of x_i in (6.7) is modified as in (6.13):

$$\begin{aligned} x_{min} &\leq x_0 &\leq t(1 - 2^{-n_0}) \\ x_{min} &\leq x_i &\leq t(2^{p_{i-1}} - 2^{p_i}) \end{aligned} \tag{6.13}$$

Let M_i be the maximum value of x_i. The x_i and δ_i are first transformed as in (6.14) before passing to a_i to generate the table contents.

$$\begin{aligned} x_i &= \frac{\widehat{x_i}}{2^{n_i}}(M_i - x_{min}) + x_{min} \\ \delta_i &= \frac{\widehat{\delta_i}}{2^{n_i}}(M_i - x_{min}) + x_{min} \end{aligned} \tag{6.14}$$

The transformation must also be applied when analyzing the errors in the approximations.

6.2 VHDL Extension

To allow for the easy implementation of the STAM algorithm in VHDL designs, simple extension is introduced by making use of the comment section inside VHDL code segments as many synthesis tools do for the synthesizing directories. A set of preprocessing tools are developed to generate VHDL codes using the STAM algorithm. The user includes the name and the body of the target function as well as some configuration parameters. The preprocessing tools will generate the corresponding VHDL codes of the function using STAM algorithm which can be used directly anywhere in the design. The listing 6.1 demonstrates the instantiation and usage of a *sin* function in the VHDL source.

In the example above, the *sin* function will accept input ranges of $[0, 1)$ and the input will be partitioned as described in the segments statement. Four

Listing 6.1: STAM instantiation

```
architecture ...
...
-- __STAM_BEGIN__
-- my_function(x) = Sin(x)
-- range_min = 0
-- range_max = 1
-- segments = 8 2 2 2 2
-- decimal_point = 16
-- __STAM_END__
component my_function is port (
clk: in std_logic;
x: in std_logic_vector(15 downto 0);
fx: out std_logic_vector(20 downto 0));
end component;
...
begin
...
f0: my_function
  port map (clk=>clk, x=>x, fx=>fx);
...
```

tables will be generated for the 16-bit input. The decimal_point statement indicates that decimal point is located at the left side of the most significant bit. The output will be ready the after next rising clock edge and will be valid as long as the input x is valid. The clock signal is required since synchronous RAMs are used to store the contents of the tables. Since the descriptions are only inside the comment section, the VHDL code can be processed by traditional synthesis tools without modification.

The VHDL codes are first passed to a preprocessor before going to the synthesis stage. A flow chart of the preprocessor is shown in Fig 6.3. First, the function extractor extracts the function body in the extended VHDL block and passes it to YACAS (YACAS is a public domain software which perform symbolic arithmetic operations [Pin03]). YACAS accepts the input function to find the symbolic first and second derivatives and passes the results to the table generation program. The table generation program uses a stack to transform the input strings to a sequence of arithmetic operations and generates the content of the lookup tables. These contents will be used in the VHDL generator to generate a complete VHDL code using Xilinx BlockRAM as the lookup tables.

With this extension, an arbitrary function can be used in VHDL code without any knowledge of the detailed implementation. The default evaluation time is 1 clock cycle but this can be easily modified in the generated VHDL codes. The only limitation is that the function must be twice differentiable due to the Taylor Expansion. As a structural design, this preprocessing method can be easily modified to other HDL languages such as Verilog.

6.3 Floating Point Extension

In the original STAM algorithm, the input value is considered a fixed point number within a predefined range. It is possible to modify the logic such that

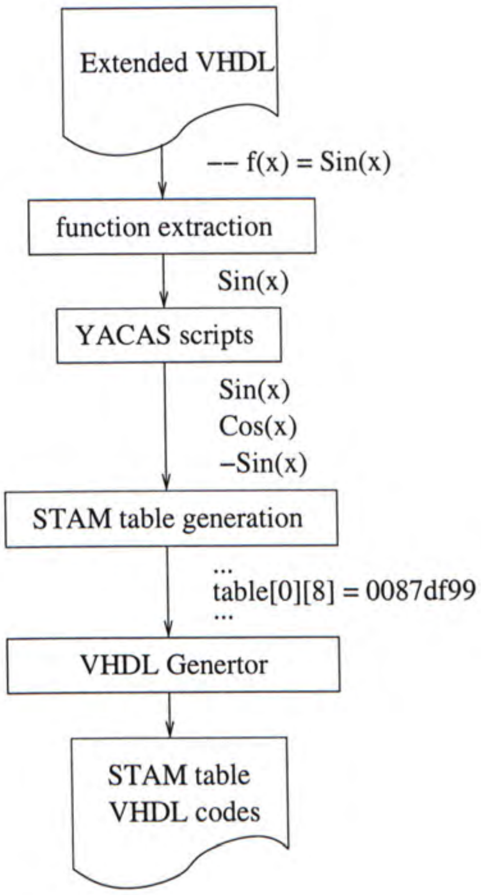

Figure 6.3: Extended VHDL Preprocessor.

it can handle specific functions for floating point arithmetic. This section will describe this process as used for the development of a floating point coprocessor for the N-body problem.

The extended *fly* compiler can use basic floating point operations, such as addition, subtraction and multiplication with different precision. Transcendental functions such as square root and exponential are frequently required to evaluate the force or acceleration in N-body problem. Such functions can be implemented using the modified STAM approach. In this research, $v^{-3/2}$ was implemented using this approach.

The STAM is configured to use 4 lookup tables.

Range reduction and result correction are necessary in the floating point implementation. Consider the IEEE 754 binary floating point number representation:

$$v = 1.f \times 2^{e} \tag{6.15}$$

Then,

$$v^{-3/2} = (1.f \times 2^{0})^{(-3/2)} \times (2^{(-3/2)e}) \tag{6.16}$$

When e is even, let $e = 2N$, equation 6.16 becomes:

$$v^{-3/2} = (1.f \times 2^{0})^{(-3/2)} \times (2^{-3N}) \tag{6.17}$$

Similarly, if e is odd, let $e = 2N + 1$, equation 6.16 becomes:

$$v^{-3/2} = (1.f \times 2^{0})^{(-3/2)} \times (2^{-3N}) \times (2^{-3/2}) \tag{6.18}$$

In both cases, the fraction part can be calculated using STAM with the input range $[1, 2)$, and the exponent part is shift and add operations. The only difference is that if e is odd, the final result should be obtained by multiplying a constant $2^{-3/2}$.

IEEE 754 requires normalization of the result from STAM. Since the output of STAM $0.354 < v^{-3/2} < 1$ for $1 \leq v < 2$, the location of the leading one must lie at either of the two most significant bits. The datapath of the calculation is shown in Fig 6.4. Since it supports parameterized size floating point numbers, it can generally fit in different FPGA devices.

To implement the circuit on FPGA, the *fly* [HLT+02] compiler was used to generate synthesizable VHDL code and the Pilchard board [LLC+01] was used as the reconfigurable platform. Pilchard uses a DIMM memory bus interface to provide high I/O performance compared to the PCI bus. *Fly* is used because of its efficiency to design a floating point algorithm by using Perl-like descriptions and its handy library which fully supports parameterized floating point arithmetic. In addition, the mechanism of the *fly* compiler allows for easy integration of a block such as STAM. The *fly* compiler was modified such that it can handle the `_power15()` function using the built-in function mechanism.

Due to the limitation of memory available on the FPGA chips, the STAM used 16-bit integers as input and the table size is (8, 2, 2, 2, 2). The STAM is used to process the function $f(x) = x^{-3/2}$ where $1 \leq x < 2$. After scaling, 0 at the input of the STAM stands for 1 according to equation 6.14. Additionally, to enhance the efficiency of the STAM and minimize the critical path in the STAM, the symmetric property in the lookup table is removed. As the output of BlockRAM is 32-bit, the memory usage is $2^{(8+2)} \times 4 \times 32 = 131072$ bits or 32 BlockRAMs.

6.4 The N-body Problem

N-body simulation finds application in various fields of science. A wide range of physical systems can be studied by modeling them as an N-Body problem. They include problems in various fields of science such as astrophysics and molecular biology. The basic idea of the N-Body problem is simple. Particles are modeled as points in space. The potential of the system can be expressed as a function of the properties and positions of all particles in the system. The force exerted on a particle is the first derivative of this potential with respect to the position of the particle. The N-body problem for different systems share the same basic structure but differ in the physical law that governs the force between particles. Therefore, the exact equation for calculating the potential and force depends on the application. By integrating the force acting on a particle, its position can be computed as a function of time.

There is no known analytic solution for the N-body problem for $N \geq 3$. Therefore, N-Body problems are solved numerically using simulation in practice. The simulation is performed in discrete time steps. In each time step, forces exerted on each particle are computed. The positions of the particles are updated at the end of the time step by integrating all forces acted on the particle.

In N-body simulation, most computation time is spent on force calculation. The number of interactions between particles grows as $O(n^2)$, where n is the number of particles. For large n, the calculation becomes very expensive and time consuming. In spite of algorithms that reduced the computation time of force calculation at the expense of accuracy, the force calculation remains an expensive step and pose a limit on the size of system that can be realistically studied.

Since the force calculation part consumes most of the CPU time, and at the same time has a rather simple algorithm, it is a good candidate for hardware acceleration. In fact, this has been done in many systems. Such systems usually have a heterogeneous architecture consisting of a general purpose host computer and a special purpose hardware. The special purpose hardware handles the force calculation while the host computer handles all other computations. Most notable of those is the GRAPE (Gravitational Pipeline) computer for the gravitational N-body problem [MT98].

The reason for using such architecture is as follows. In a system powered by general purposed processors, only a small fraction of the transistors in the processor are doing useful work at any moment. The key for GRAPE or other such systems to achieve performance orders of magnitude higher than a general-purpose system is to utilize almost all of the transistors on the chip at any moment. With filled pipelines of the processors for the force calculation, almost all of the transistors are performing useful computations at any given moment.

6.5 Implementation

In this work, a FPGA based co-processor for evaluating gravitational forces in N-body simulations was built using the module generator approach. The architecture of the co-processor is similar to the GRAPE-1 system. GRAPE-1 is the first in a series of specialized processors evaluating gravitational forces or acceleration in a gravitational N-body simulation. Equations 6.19, 6.20, 6.21 show the force evaluation in this system.

$$\mathbf{a}_i = \sum_{j=1}^{N}(\mathbf{a}_{ij}) \tag{6.19}$$

$$\mathbf{a}_{ij} = (\mathbf{x}_j - \mathbf{x}_i)(r_{ij}^2 + \epsilon^2)^{-3/2} \tag{6.20}$$

$$r_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 \tag{6.21}$$

The equations are the same as those implemented in the GRAPE-1 system. $\mathbf{a}_i$ is the gravitational acceleration at the position of particle i, $\mathbf{x}_i$ is the position of vector particle i, r_{ij} is the distance between particles i and j and ϵ is the artificial potential softening used to suppress the divergence of the force at $r_{ij} \to 0$.

A program written in *fly* language is used to implement the equation 6.19, which is used intensively during the whole calculation as shown in listing 6.2. The input of the program is $\mathbf{x}_i$, $\mathbf{x}_j$ and ϵ while the output is the acceleration $(\mathbf{a}_{ij})$ for a particular value of $\mathbf{x}_j$. Most of the constructs are parallel in nature so that the vector manipulation can be processed simultaneously. For example, $\mathbf{x}_j - \mathbf{x}_i$ can be done at the same time for each scalar in $\mathbf{x}_j$ and $\mathbf{x}_i$. The *fly* code can used for simulation and verification by directly executing it under the Perl environment, which saved time and reduced the error when compared with manually translating the algorithm description into VHDL.

The floating point module supplied by the *fly* compiler is readily parameterized, so the tradeoff between the accuracy and slice resources is adjustable. Different sizes

Listing 6.2: Implementation of N-Body Simulation

```
{
# initialization , fetch xi,yi,zi
while ($j < $n) {
    # fetch xj,yj,zj from memory
        $xj = &read_host($index);
        $index = $index + 1;
        $yj = &read_host($index);
        $index = $index + 1;
        $zj = &read_host($index);
        $index = $index + 2;
        [{$diffx = $xj .- $xi;}
        {$diffy = $yj .- $yi;}
        {$diffz = $zj .- $zi;}]
        [
        {$x = $diffx .* $diffx;}
        {$y = $diffy .* $diffy;}
        {$z = $diffz .* $diffz;}
        ]
        [
        {$r1 = $x .+ $y;}
        {$r2 = $z .+ $epsilon;}
        ]
    # caculate rij
        $rij = $r1 .+ $r2;

    # call built-in function power^{-1.5}
        $tmp2 = &_power15($rij);

        [{$tmpx = $tmp2 .* $diffx;}
        {$tmpy = $tmp2 .* $diffy;}
        {$tmpz = $tmp2 .* $diffz;}]
        [{$ax = $ax .+ $tmpx;}              # accumulate a
        {$ay = $ay .+ $tmpy;}
        {$az = $az .+ $tmpz;}]
        $j = $j + 1;
}
$void = &write_host($ax, 60);
$void = &write_host($ay, 61);
$void = &write_host($az, 62);    # write back to host
}
```

of fraction and magnitude can be implemented and the best performance rating can be achieved.

6.6 Summary

A flexible framework for implementing elementary function using lookup table on the FPGA has been introduced in this chapter. Using the STAM algorithm, it can be used to generate synthesizable VHDL modules from comments in the VHDL source code. This function generator was integrated into the *fly* environment to extend flexibility and efficiency. An N-body problem simulation was implemented on the FPGA to demonstrate the power of this framework. Without detailed knowledge of the STAM implementation, the N-body core was generated from 45 lines of fly source code. This example shows that this framework can be used to solve a real world problem with minimum design effort.

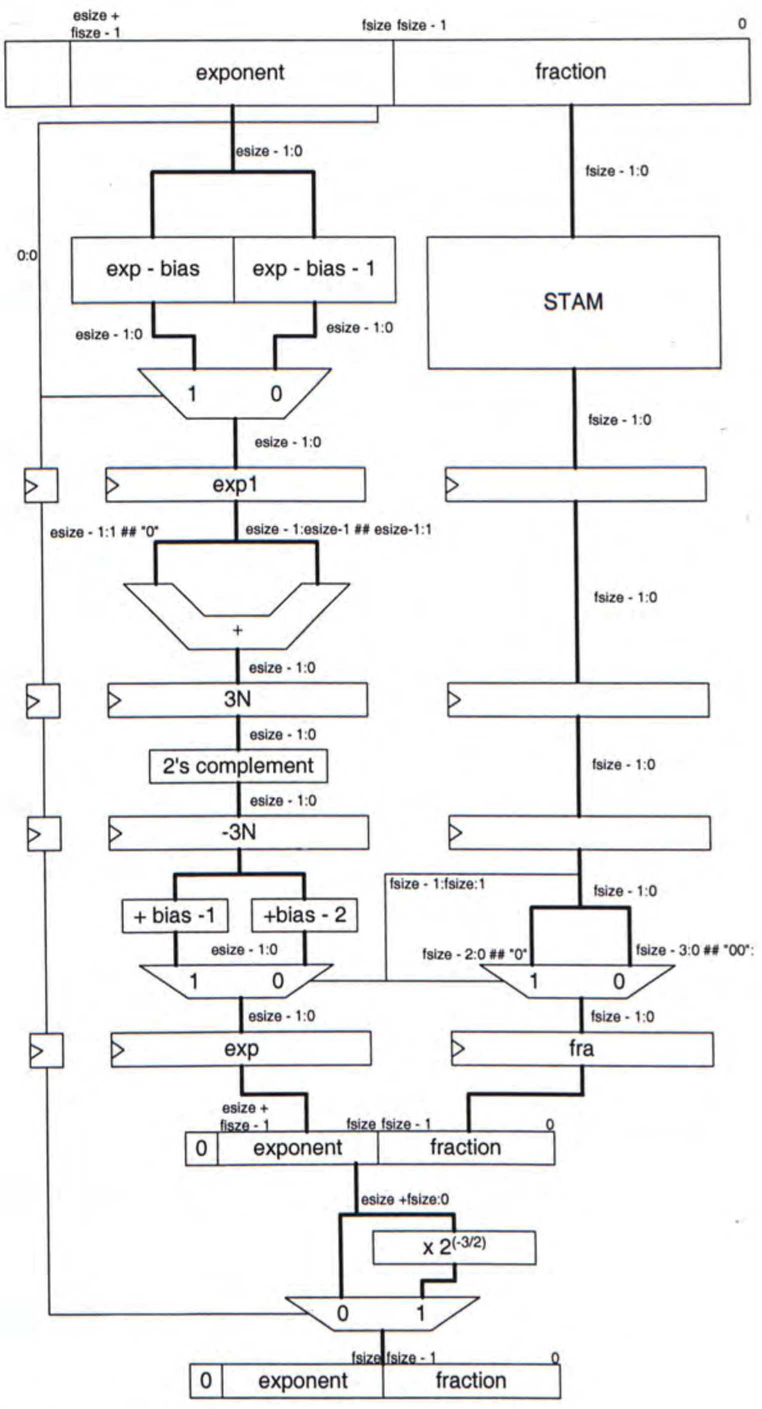

Figure 6.4: Datapath of $v^{-3/2}$ using STAM for floating point arithmetic

Chapter 7

Results

7.1 Introduction

In this chapter, results of all the experiments described in previous chapters are presented. All experiments, unless other specified, were tested on a Pilchard FPGA board [LLC$^+$01] and the FPGA chips used was Xilinx XCV1000E-6 which contains a total of 12,288 slices and 96 BlockRams. This chapter includes the following experiments:

1. GCD coprocessor
2. Floating point module generator
3. Digital sine-cosine generator (DSCG)
4. Ordinary differentiable equation solver (ODE)
5. N-body problem simulation

7.2 GCD coprocessor

The GCD coprocessor design was synthesized for a Xilinx XCV300E-8 and the design tools reported a maximum frequency of 126 MHz. The design, including interfacing circuitry, occupied 135 out of 3,072 slices. The design time for the GCD processor, including host interface was approximately one hour.

Listing 7.1: GCD Testing program

```
for (my $i = 0; $i < $cnt ; $i++) {
    $a = rand(0x7fff) & 0x7fff;
    $b = rand(0x7fff) & 0x7fff;

    &pilchard_write64(0, $a, 1);      # write a
    &pilchard_write64(0, $b, 2);      # write b
    &pilchard_write64(0, 0, 0);       # start coprocessor

    do {
        &pilchard_read64($data_hi, $data_lo, 0);
    } while ($data_lo == 0);          # poll for finish
    &pilchard_read64($data_hi, $data_lo, 1);

    print ("gcd $a, $b = $data_lo\n");
}
```

The Perl listing 7.1 tests the GCD coprocessor using randomly generated 15-bit inputs. The GCD coprocessor was successfully tested at 100 MHz by calling the FPGA-based GCD implementation with random numbers and checking the result against a software version. The resulting system could compute a GCD every 1.63 μs (including all interfacing overheads).

7.3 Floating Point Module Library

Different configurations of adders and multipliers were extracted from the module library, simulated and synthesized for the Virtex XCV1000E-6 FPGA. Table 7.1 is a summary of the resource requirements, maximum reported frequency and latency for a fixed exponent length of 8 bits and different fraction sizes. The adder is not yet fully optimized and the maximum frequency was 40 MHz with a 4 stage pipeline.

Table 7.1: Area and speed of the floating point library.

Fraction Size (bits)	Circuit Size (slices)	Frequency (MHz)	Latency (cycles)
Multiplication			
7	178	103	8
15	375	102	8
23	598	100	8
31	694	100	8
Addition			
7	120	58	4
15	225	46	4
23	336	41	4
31	455	40	4

7.4 Digital sine-cosine generator (DSCG)

The algorithm function of the sine-cosine generator was simulated by directly executing it in Perl. Figure 7.1 shows the resulting double precision reference output. The output will be used for evaluating the quantization error for different precision configurations.

Figure 7.2 shows the quantization error of the *Float* simulation for different fraction size, as a function of time. In the simulation, the exponent field was set to be large enough to avoid overflow. The maximum exponent value can be determined during the simulation of the algorithm. As expected, the error is reduced as the number of fractional bits (and hence precision) is increased.

Figure 7.3 shows the QERR as mentioned in Section 5.2.2 of digital sine-cosine generator with a varying number of fraction bits, assuming that the exponent field is large enough to avoid overflow. For fraction bits varying from 12 to 40 bits, the QERR ranged from -50 to -210 dB. Linear relationship is discovered between QERR and fraction size.

The single precision of digital sine-cosine generator is implemented. The reported frequency is 52.4 MHz and consumed 3,470 slices.

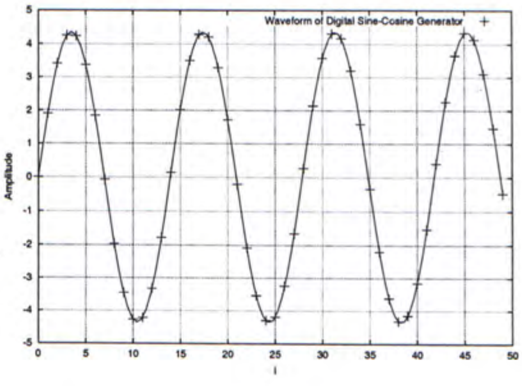

Figure 7.1: Digital sine-cosine generator reference output.

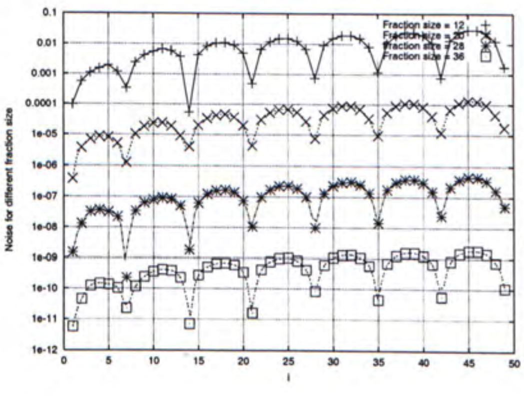

Figure 7.2: Quantization error of the sine-cosine generator for different fraction sizes.

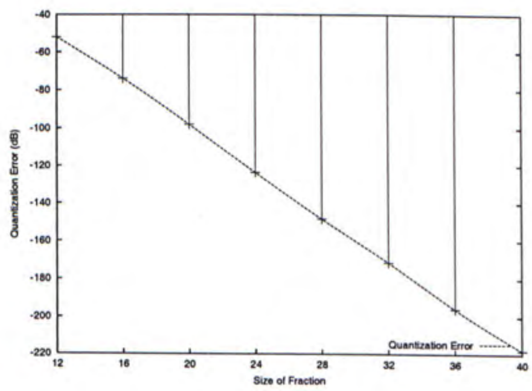

Figure 7.3: Quantization error for different fraction sizes.

Table 7.2: Optimization result using different QERR values where (x,y) are the (exponent size, fraction size) in bits.

QERR	s1	s2	$cos(\theta)$	$cos(\theta)+1$	$cos(\theta)-1$
-52	(5,10)	(5,12)	(5,11)	(5,11)	(5,12)
-73	(5,15)	(5,14)	(5,15)	(5,15)	(5,16)
-98	(5,19)	(5,18)	(5,19)	(5,19)	(5,20)
-123	(5,23)	(5,21)	(5,23)	(5,24)	(5,24)
-148	(5,26)	(5,28)	(5,27)	(5,27)	(5,28)
-171	(5,31)	(5,30)	(5,31)	(5,31)	(5,32)
-195	(5,35)	(5,33)	(5,35)	(5,36)	(5,36)
-218	(5,38)	(5,39)	(5,38)	(5,39)	(5,40)

7.5 Optimization

By varying the fraction size of the *Float* objects using the technique described in Section 5.2.2, the optimizer can minimize the cost function while maintaining a given maximum quantization error. This technique was used to determine the minimum area requirements for a given QERR. Table 7.2 shows the optimized number of fraction bits and exponent bits for different maximum QERR. As expected, the trend for all variables is an increase in wordlength as the QERR requirement is increased.

Figure 7.4 compares the optimized circuit size (which allows variables to have different numbers of fractional bits) to a scheme where all variables have the same number of fraction bits (i.e. the fixed fraction case). The "Fraction Size" curve was made by computing the area of the sine-cosine generator for the case that all variables have the fraction size on the x-axis. The "Optimized Circuit Size" curve was made by using the fraction size of the x-axis as the starting point for an optimization, with the maximum QERR specified to be that of the fixed fraction case. Thus it can be seen from the figure that for the same quantization error, a 2% to 5% reduction in area is achieved by the optimization process.

In the sine-cosine generator, all variables require similar precisions. In applications where variables have widely different precisions, one would expect the scheme

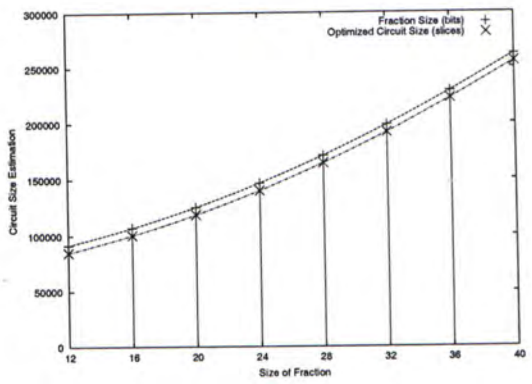

Figure 7.4: Area estimation of the fixed fraction and optimized circuits.

allowing different fractional sizes to offer a much larger improvement in area efficiency.

7.6 Ordinary Differential Equation (ODE)

The differential equation solver described in section 5.5 was synthesized for a Xilinx XCV300E-8 device and the design tools reported a maximum frequency of 53.9 MHz. The design, including interfacing circuitry, occupied 2,439 out of 3,072 slices. The outputs shown in Table 7.6 were obtained from the hardware implementation at 50 MHz using different h values. The resulting system ($h = \frac{1}{16}$) took 28.7 μs for an execution including all interfacing overheads.

7.7 N Body Problem Simulation (Nbody)

The VHDL code generated by *fly* compiler which implement n body problem simulation with $N = 10$ together with the STAM extensions was implemented using the design tools and the bitstream is generated. Table 7.7 shows the result of implementation using different floating point configurations. The number of BlockRAMs was always 32 and thus not included in the table. The data used was a NEMO N-body snapshot data set [Teu03]. For experimental purposes, $N = 10$ was used during the

t_k	$h=1$	$h=\frac{1}{2}$	$h=\frac{1}{4}$	$h=\frac{1}{8}$	$h=\frac{1}{16}$	$y(t_k)$ Exact
0	1.0	1.0	1.0	1.0	1.0	1.0
0.125	-	-	-	0.9375	0.940430	0.943239
0.25	-	-	0.875	0.886719	0.892215	0.897491
0.375	-	-	-	0.846924	0.854657	0.862087
0.50	-	0.75	0.796875	0.817429	0.827100	0.836402
0.75	-	-	0.759766	0.786802	0.799566	0.811868
1.00	0.5	0.6875	0.758545	0.790158	0.805131	0.819592
1.5	-	0.765625	0.846386	0.882855	0.900240	0.917100
2.00	0.75	0.949219	1.030827	1.068222	1.086166	1.103638
2.50	-	1.211914	1.289227	1.325176	1.342538	1.359514
3.00	1.375	1.533936	1.604252	1.637429	1.653556	1.669390

Table 7.3: Results generated by the differential equation solver for different values of h.

Table 7.4: The frequency and slices used reported by design tools for N-body problem

Floating Point Configuration (exponent size, fraction size)	Area (slices)	Frequency (MHz)	QERR (dB)
(5, 15)	3,523	47.34	-82
(5, 23)	5,267	44.07	-102
(8, 15)	3,837	48.92	-82
(8, 23)	5,475	44.79	-102

evaluation of quantization error.

7.8 Summary

This chapter presents the area and performance results for all the designs previously described. The tradeoff between area and precision are discussed for each experiment. The results are summarized in Table 7.8. The QERR of the GCD example is omitted as it does not involve approximations to fractional numbers. The exponent size of all the floating point numbers was fixed to 8-bit.

To implement each design, designer needs to do 2 steps:

1. Use a Perl-like language (*fly*) to describe the algorithm.

Problem Name	Fraction Size (bits)	Frequency (MHz)	Area (Slices)	QERR (dB)
GCD	16	126.0	135	-
DSCG N = 50	15	78.16	2,300	-81
DSCG N = 50	23	52.38	3,470	-127
ODE $h = \frac{1}{16}$	15	75.74	1,715	-84.8
ODE $h = \frac{1}{16}$	23	64.50	2,495	-134
Nbody N = 10	15	48.92	3,837	-82
Nbody N = 10	23	44.79	5,475	-102

Table 7.5: All Experiments Result

2. Suggest the precision of the floating/fixed point numbers to be used.

The *fly* description for all examples was short and easily understandable and it can be easily seen that the descriptions are much easier to write and understand than corresponding VHDL description. The design environment can generate the bitstream suitable for on-board testing. When compared with the traditional design flow, a significant amount of time is saved and thus the productivity of the designer is increased. To further customize the design, the precision of the floating point number can be varied as specified by the designer. The optimizer can help the designer to balance the tradeoff between accuracy and area of the hardware via the given cost function.

Chapter 8

Conclusion

This research purposed a mixture of hardware compilation, module generators, floating point arithmetic and automatic interface generation to improve the the efficiency, productivity and flexibility when implementing the floating point design on the FPGA. The framework allows designers to use a programming language to implement a design, automatically generating floating point circuits and elementary arithmetic. For the same design, this framework allows the tradeoff of precision and area used from a single description. Several applications, such as digital sine cosine generator, greatest common divisor coprocessor, ordinary differentiable equation solver and N-body problem simulator have been developed using this approach. The key issues of this research are highlighted as below.

Integration of programming language and FPGA

Using the same programming language for describing the algorithm and implementing on FPGA design can benefit designers in several ways. The algorithm can be verified and simulated by executing the code under a software environment. The translation and optimization process are done by the tools and the designer can concentrate on the higher level in details. This methodology greatly reduces design time and achieves rapid system prototyping. Design errors can be reduced compare with the traditional design flows since the translation is done automatically instead of manually porting the algorithm into datapath and control components.

Software programming and hardware designs being treated as distinct entities remain an obstacle to developing a FPGA based system. The design goal of *fly* and *float* is the bridge between these two entities in a way that a software program can be translated into a hardware implementation. Using these tools, the designer can reuse the software code, optimize the hardware resources used and perform on-board testing without additional effort. The time required to implement floating point algorithms on FPGA can be significantly reduced. With ever increasing device densities, this design methodology should become even more attractive in the future.

Floating point/Elementary arithmetic on FPGAs

This dissertation discussed the possibility of connecting a floating point algorithm description to a hardware. When the floating point algorithm on the reconfigurable computing platform, using arbitrary length of operator is now possible such that the tradeoff between circuit size and the accuracy can be varied. Thus the designer can choose the best performance rating by providing a suitable cost function and the optimizer can return the best configuration for each of the floating point operator.

Elementary functions can be automatically generated using lookup tables. These act like a flexible mathematical library in software. It enhances the flexibility since the designer does not need to implement every elementary function from scratch. The automatic function generator saves the design time and extra hardware knowledge is not required to build any elementary function.

By combining all of these module generators, the implementation of floating point design in reconfigurable computing platform is made simpler. It allows wide range of applications, such as scientific simulation, equation solver, DSP design, to be implemented as a FPGA based coprocessor. It also benefits the HDL design flow because floating point arithmetic is available as a synthesizable VHDL module. Any HDL design can easily interface with the floating point operator.

Adapt to different architectures

Fly generates VHDL code since it is generally available to different reconfigurable computing platform. Therefore, even though the design environment is now targeted for the Pilchard board, it can be ported to different reconfigurable computing platforms such as other FPGA products or even ASICs with only slight modification. In addition, HDL output enables further optimization on different FPGA platform using the corresponding design tools.

Fly is a modifiable compiler which can be able to produce code for different HDLs, program proving tools, and programming languages. Having an easily understandable and easily modifiable compiler allows for the easy integration of the *fly* language to many other tools. The integration of *fly* language was introduced. For example, new host interface mechanism, floating point arithmetic and arbitrary function generation is extended from the basic *fly* environment.

8.1 Future Work

There are several possibilities for improvements to the system. The compiler produces only one-hot state machines which may be inefficient in certain cases. The state machine can be different and not limited to a certain implementation. The resulting datapath is not fully utilized, and the operators are idle most of the time. It would be desirable if the coding strategy let the datapath share hardware resources for some operation. This coding strategy thus can save area if it is critical for certain application. The parallelism must now be implemented by the user. It would be better if the compiler itself can detect the dependency to reorganize the datapath in which the parallelism can be achieved automatically. However, it is believed that the benefits in productivity and flexibility that could be gained from this approach outweighs the cons.

The compiler in Appendix B generates a bit parallel implementation but, for example, if a digit serial operator library were available, it could be easily modified to use digit serial arithmetic. Similarly, both fixed point and floating point implementations of the same algorithm could be generated from the same *fly* description. In the future, we will experiment with different code generation strategies. Many designs could be developed from the same program, and different *fly* based code generators could serve to decouple the algorithmic descriptions from the back-end implementation. In the case of using a digit serial library, users could select the digit size, or produce a number of implementations and choose the one which best meets their area/time requirements.

Finally, the elementary function generator is a fixed point one and floating point functions was implemented by the designer. This process could conceivably be further automated to produce an automatic floating point elementary function generator.

Appendix A

Fly Formal Grammar

program = statement_list

statement_list = statement | "{" statement(s) "}"

parallel_statement = "[" statement (s) "]"

statement = comment | assignment | ifelse | if | while | parallel_statement | function_call

assignment = variable "=" expression "; "

expression = value operator expression | value

operator = "*" | "/" | "+" | "-" | ".+" | ".-" | ".*"

value = INTEGER | variable

variable = "$" LETTER | "$" LETTER DIGIT

while = "while" "(" condition ")" statment_list

ifelse = "if" "(" condition ")" statment_list "else" statment_list

if = "if" "(" condition ")" statment_list

condition = expression relation expression

relation = ">" | "<" | "<=" | ">=" | "! =" | "=="

function_call = variable "=" function_name "(" variable_list ")" ";"

function_name = "read_host" | "write_host" | "_power15"

variable_list = value "," variable_list | value

comment = "#" ANYTHING

Appendix B

Original Fly Source Code

```
package main;
use Parse::RecDescent;

my $grammar = q {
{ my ($seq, $comb, $aux, $paux, $s, %sigs) =
        ("", "", 0, 0, "signal"); }

prog: stmtlist /^$/  {
  print <<EOF
   library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith.all;
   package hc_pack is ;
     subtype word is integer;
     type   words is array(integer
   range <>) of word;
   end hc_pack;

   library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith.all;
   use work.hc_pack.all;


   entity arith_core is
   port(
       clk: in std_logic;
       rst: in std_logic;
       start: in std_logic;
       din : in words( $sigs{din} ;
       downto 1);
       finish: out std_logic;
       dout: out words( $sigs{dout} ;
       downto 1));
   end arith_core;
   architecture rtl of arith_core is;
   EOF
   ;

 foreach my $k (keys %sigs) {
   if ($sigs{$k}) {
     print "$s  $k :\t words($sigs{$k}  " .
       "downto 0);\n"
     if !($k eq "din")
       and !($k eq "dout") ;
   }
   else {
     print "$s  $k :\t word; \n";
   }
 }
 for (my $i=1; $i<$aux; $i++) {
   print "$s s$i, f$i :\t boolean; ";
   print "--std_logic;\n";
 };
 for (my $i=1; $i<=$paux; $i++) {
   print "$s p$i, q$i :\t boolean; ";
   print "--std_logic;\n";
 };

 print "$s s$item[1], f$item[1] :\t boolean; ";
 print "--std_logic;\nbegin --architecture\n";
 print " s$item[1] <= TRUE when start='1' ";
 print "else FALSE ;--start;\n finish <= '1' ";
 print "when f$item[1] else '0'; --f$item[1];\n";
 print "process(clk)\nbegin\n";
 print "if rising_edge(clk) then\n";
```

```
  print $seq;
  print "end if;\nend process;\n";
  print "--combinational part\n$comb";
  print "end rtl;\n";
}

stmtlist: stmt | '{' stmt(s) '}' {
  my $fst_in = shift(@{$item[2]});
  my $int_in = $fst_in;
  $aux += 1 ;
  $comb .= "s$int_in <= s$aux; \n";
  foreach $int_in (@{$item[2]}) {
    $comb .= "s$int_in <= f$fst_in;\n";
    $fst_in = $int_in;
  }
  $comb .= "f$aux <= f$fst_in;\n";
  $aux;
}

stmt: asgn | ifelse | if  | while
    | pstmtlist | <error>

pstmtlist:  '[' stmtlist(s) ']' {
  $aux += 1;
  my $int_in;
  my @plist = ();
  foreach $int_in (@{$item[2]}) {
    $comb .= sprintf("s%d <= s%d;\n",
                    $int_in, $aux);
    $paux += 1;
    push (@plist, $paux);

    $seq .= "if f$aux then --pstmtlist\n\t";
    $seq .= "q$paux <= false;\n";
    $seq .= "else\n\t";
    $seq .= "q$paux <= p$paux; \n";
    $seq .= "end if; \n";

    $comb .= "p$paux <= f$int_in or q$paux;\n";
  }
  my $pend = "f$aux <= p" .
    join(" and p", @plist)
    . "; --pstmt end\n";
  $comb .= $pend;
  $aux;
}

asgn: var '=' expr ';' {
  $aux = $aux + 1;
  $seq .= "if s$aux then\n\t";
  $seq .= "$item[1] <= $item[3];\n";
  $seq .= "end if;\n";
  $seq .= "f$aux <= s$aux;\n\n";
  $aux;
}

expr: val op expr { "$item[1] $item[2] $item[3]" } |  val

op: '*' | '/' | '+' | '-'

val: /\d+/ | var

var: /\$[a-z][\w\[\]]*/  {
  $item[1] =~ s/^\$//;
  my $sig = $item[1];
  $sig =~ s/\[(\d+)\]//;
  $sigs{"$sig"} = ($sigs{"$sig"} && ($sigs{"$sig"} > $1) )
    ? $sigs{"$sig"} : $1;
  $item[1] =~ tr/\[\]/\(\)/;
  $item[1];
}

while: 'while' '(' cond ')' stmtlist {
  $aux += 1;
  $comb .= "s$item[5] <= ($item[3]) and " .
           "(s$aux or f$item[5]);\n";
  $comb .= "f$aux <= (not ($item[3])) and " .
           "(s$aux or f$item[5]);\n";
  $aux;
}

ifelse: 'if' '(' cond ')' stmtlist 'else' stmtlist {
  $aux += 1;
  $comb .= "s$item[5] <= ($item[3]) and s$aux;\n";
  $comb .= "s$item[7] <= (not ($item[3])) and s$aux;\n";
  $comb .= "f$aux <= f$item[5] or f$item[7];\n";
  $aux;
}
```

```
if: 'if' '(' cond ')' stmtlist {
  $aux += 1;
  $comb .= "s$item[5] <= ($item[3]) and s$aux;\n";
  $comb .= "f$aux <= (not ($item[3]) and s$aux) or f$item[5];\n";
  $aux;
}

cond: expr rel expr  { "$item[1] $item[2] $item[3]" }

rel: '>' | '<' | '<=' | '>=' | '!=' { "/=" } | '==' { "=" }

varlist: var ',' varlist  { "$item[1] $item[3]" } | var
};

$::RD_HINT = 0;
$::RD_AUTOACTION = q { $item[1] };
my $parser = Parse::RecDescent->new($grammar)
  or die "Bad grammar";

local $/;
my $script = <>;
my $tree = $parser->prog($script) or die "Bad script";
```

Bibliography

[ANS85] New York ANSI/IEEE. IEEE Standard for Binary Floating-Point Arithmetic. Technical report, The Insittution of Electrical and Electronics Engineerings, Inc, 1985. IEEE Std 754-1985.

[BL02] Pavlé Belanovic and Miriam Lesser. A Library of Parameterized Floating-point Modules and Their Use. In *Field Programmable Logic and Application. Reconfigurable Computing Is Going Mainstream*, pages 657–666. Springer-Verlag Heidelberg, Sept 2002.

[BSC$^+$99] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden. MATCH: a MATLAB compiler for configurable computing systems. Technical report, Center for Parallel and distributed Computing, Northwestern University, Aug 1999. Technical Report CPDCTR -9908-013.

[Con01] D. Conway. Parse::RecDescent Perl module. In *http://www.cpan.org/modules/by-module/Parse/DCONWAY/Parse-RecDescent-1.80.tar.gz*, 2001.

[ETS96] ETSI. *Radio Equipment and Systems (RES); HIgh PErformance Radio Local Area Network (HIPERLAN) Type 1; Functional specification.* The European Telecommunications Standards Institute, 1st edition, 1996.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[HLT$^+$02] C.H. Ho, P.H.W. Leong, K.H. Tsoi, R. Ludewig, P.Zipf, A.G. Ortiz, and M. Glesner. Fly - a modifiable hardware compiler. In *Proceedings of the twelfth International Workshop on Field-Programmable Logic & Applications*, 2002.

[IEE02] IEEE Computer Society. 1076 IEEE Standard VHDL Language Reference Manual. Technical report, The Insittution of Electrical and Electronics Engineerings, Inc, 2002. IEEE Std 1076-2002.

[JGW81] John E. Dennis Jr, David M. Gay, and Roy E. Welsch. An adaptive nonlinear least-squares algorithm. *ACM Transactions on Mathematical Software*, 7(3):348–368, Sept 1981.

[JL01] A. Jaenicke and W. Luk. Parameterised floating-point arithmetic on FPGAs. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 897–900, 2001.

[LLC$^+$01] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, M.Y. Wong, and K.H. Lee. Pilchard - a reconfigurable computing platform with memory slot interface. In *Proceedings of the IEEE Symposium on FCCM*, 2001.

[LMM$^+$98] W. B. Ligon, S. McMillan, G. Moon, K. Schoonover, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206–215. IEEE Computer Society Press, 1998.

[MF99] J. Mathews and K. Fink. *Numerical Methods Using MATLAB*, pages 433–441. Prentice Hall, 3rd edition, 1999.

[Mit98] Sanjit K. Mitra. *Digital Signal Processing A Computer-Based Approach International Editions 1998*, pages 339–416. McGraw-Hill, 1998.

[MT98] Junichiro Makino and Makoto Taiji. *Scientific Simulation with Special-Purpose Computers - the GRAPE systems*, pages 41–48. John Wiley & Sons Ltd, 1998.

[NM65] J. Nelder and R. Mead. A simplex method for function minimization. In *Computer Journal*, pages 308–313, 1965.

[Pag96] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[Pin03] Ayal Pinkus. Yet another computer algebra system (YACAS), 2003.

[SS97] Michael J. Schulte and James Stine. Symmetric bipartite tables for accurate function approximation. In Tomas Lang, Jean-Michel Muller, and Naofumi Takagi, editors, *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 175–183, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[SS99a] James E. Stine and Michael J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21:167–177, 1999.

[SS99b] James E. Stine and Michael J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21:167–177, 1999.

[SWA95] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *Proc. FCCM*, pages 155–162, 1995.

[Teu03] Peter Teuben. NEMO - A Stellar Dynamics Toolbox, 2003.

[WA02] M. Ward and N.C. Audsley. Hardware Implementation of Programming Languages for Real-Time. In *Proceedings of the Eigth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 276–285, Sept 2002.

[WCO00] L. Wall, T. Christianson, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

[Xil01] Xilinx Inc. *Architectural Description*, pages 6–57. Xilinx Inc, 2001.

Publications

Full Length Conference Papers

- C.H. Ho, M.P. Leong, P.H.W. Leong, J. Becker, M.Glesner, "Rapid Prototyping of FPGA based Floating Point DSP Systems", in *Proceedings of IEEE International Workshop on Rapid System Prototyping*, July 2002.
- C.H. Ho, P.H.W. Leong, K.H. Tsoi, R. Ludewig, P. Zipf, A.G. Ortiz, M.Glesner, "Fly - A Modifiable Hardware Compiler", in *Proceedings of International Conference on Field Programmable Logic and Applications*, September 2002.
- C.H. Ho, K.H. Tsoi, H.C. Yeung, Y.M. Lam, K.H. Lee, P.H.W. Leong, R. Ludewig, P. Zipf, A.G. Ortiz, M. Glesner, "Arbitrary Function Approximation in HDLs", submitted to *Proceedings of IEEE International Conference on Field-Programmable Technology*, December 2003.