University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Customisable and Reconfigurable Platform for Optimising Floating Point Computations

Chun Hok Ho

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, September, 2009

Abstract

This research proposes a platform for developing reconfigurable architectures dedicated to floating point computations. The platform involves customisable and reconfigurable architectures with the associated tools and methods for modelling, designing and using the proposed architectures to optimise floating point computations.

Customisability refers to modifying devices to target a specific application domain *before fabrication*. Reconfigurability refers to programming devices to implement different application circuitries in such domain *after fabrication*. A customisable and reconfigurable platform has been delivered, with the following contributions.

(1) Modelling: To model the proposed devices and compare with existing FPGA devices, this thesis proposes a methodology by using existing vendor tools to estimate the area, delay and power consumption of the devices.

(2) Synthesisable Datapath: The proposed architectures capture common patterns appearing in floating point datapaths such as bus-based logic and routing. By exploiting shared configuration bits, we propose a datapath-style coarse-grained reconfigurable fabric. In addition, we adopt synthesisable design flow allowing user customisation.

(3) Floating point FPGA: We propose an FPGA device which consists of island-style fine-grained fabric for general purpose computations and datapath-style coarse-grained fabric for floating point computations. The coarse-grained fabric contains dedicated circuitries for floating point operation and is customisable according to domain-specific requirements.

(4) Application design flow: A high level design flow is proposed which can translate a high level description of an application into a reconfigurable implementation. The key component in the high level design flow is a technology mapper which can map a given dataflow graph of an application into reconfigurable devices with different architectural parameters.

Floating point applications have been implemented on an instance of the proposed reconfigurable architecture and promising results have been reported. Application domains include digital signal processing, scientific applications, and financial applications. Area can be reduced by 25 times and delay can be shortened by 4 times on average, while dynamic energy consumption is reduced by 14 times when compared with a traditional FPGA implementation with comparable technology.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor Professor Wayne Luk, for his support, encouragement and trust throughout the years. His insightful comments have improved my papers, presentations, and this thesis tremendously, and his advice has always helped me to make the right decision.

This thesis would not have been possible without the guidance from my secondary thesis advisor Professor Philip Leong. He enlightened me the fascinating reconfigurable computing research and supervised me in both the bachelor and master degree studies. I deeply appreciated his encouragement and suggestions in these years.

I am indebted to Professor Steven Wilton, for assisting me in designing the architecture of the floating point FPGA. He proposed the synthesisable datapath FPGA architecture (Figure 4.1) described in Chapter 4. His knowledge in System-on-chip development as well as his research attitude inspired me the way to conduct serious scientific research.

Special thanks are due to Brad Quinton for his collaboration in our synthesisable datapath FPGA architecture study for generating the layout of the datapath FPGA fabric (Figure 4.6).

I would like to express my gratitude to Chi Wai Yu, for his contribution in preparing benchmark circuits used in Chapter 5.

It is a real privilege for me to study in the Custom Computing Group, Department of Computing at Imperial College London. It is one of the largest groups in the world dedicated to reconfigurable computing research. I enjoyed the research experience here and I learnt a lot from my sincere colleagues. Thanks are especially given to Ray Cheung, David Thomas, Kubilay Atasu, Haohuan Fu, Yuet Ming Lam, Kuen Hung Tsoi, Gary Chow, Julien Lamoureux and Markus Koester, for their invaluable supports and countless hours of discussions which help me in preparing this thesis.

During my graduate study, I had the good fortune to receive an internship opportunity and I enjoyed wonderful spring and summer working in the Xilinx Research Labs at San Jose, California. The experience was fruitful and the project gave me the idea of developing high level synthesis tools for floating point FPGA. I would like to express my most sincere thanks to my mentors Prasanna Sundararajan, Jeff Mason and Dave Bennett, for sharing their insights and expertise, which have greatly broadened my knowledge and vision.

My special thanks go to Lucia Lau, Edith Ngai, Dongping Zhang and Olivier Pernet, for their supportive comments and being companionable flatmates who enrich my overseas research experience in the United Kingdom.

The financial support from Overseas Research Students Award Scheme and UK Engineering and Physical Sciences Research Council (grant number EP/D060567/1) is gratefully acknowledged.

Dedication

To my parents,

for raising me to be the person I am today;

and to Angela,

for her continued patience, care, support and love.

Table of Contents

List of Tables

List of Figures

List of Publications

- **C.H. Ho**, C.W. Yu, P.H.W. Leong, W. Luk and S.J.E. Wilton, "Floating Point FPGA: Architecture and Modelling", to appear in *IEEE Transactions on Very Large Scale Integration Systems.*

- S.J.E. Wilton, **C.H. Ho**, B. Quinton, P.H.W. Leong and W. Luk, "A Synthesizable Datapath-Oriented Embedded FPGA Fabric for Silicon Debug Applications", in *ACM Transaction of Reconfigurable Technology and Systems*, 1(1):7:1–7:25, March 2008.

- **C.H. Ho**, P.H.W. Leong, W. Luk and S.J.E. Wilton, "Rapid Estimation of Power Consumption for Hybrid FPGAs", in *Proceedings of Field Programmable Logic*, pp. 227–232, 2008. **Stamatis Vassiliadis outstanding paper award.**

- **C.H. Ho**, C.W. Yu, P.H.W. Leong, W. Luk and S.J.E. Wilton, "Domain-Specific FPGA: Architecture and Floating Point Applications", in *Proceedings of Field Programmable Logic*, pp. 196–201, 2007. **Stamatis Vassiliadis outstanding paper award.**

- S.J.E. Wilton, **C.H. Ho**, P.H.W. Leong, W. Luk and B. Quinton, "A Synthesizable Datapath-Oriented Embedded FPGA Fabric", in *Proceedings of Fifteenth ACM/SIGDA International Symposium on FPGAs*, pp. 33–41, 2007.

- **C.H. Ho**, P.H.W. Leong, W. Luk, S.J.E. Wilton, S. Lopez-Buedo, "Virtual Embedded Blocks: A Methodology for Evaluating Embedded Elements in FPGAs", in *Proceedings of Field-Programmable Custom Computing Machines*, pp. 35–44, 2006.

Chapter 1

Introduction

Improvements in floating point performance have led to major advances in applications as diverse as weather forecasting, problem modelling, financial engineering, molecular dynamics and drug discovery. Although supercomputers based on microprocessor clusters are commonly used for these applications, it appears that their efficiency in terms of sustained performance and power consumption can be significantly improved through increased fined-grained parallelism and better memory utilisation. A good example of a processor optimised for power consumption and performance is the one used in IBM Roadrunner machine [Craw 08], which is currently one of the fastest supercomputers [Dong 08]. Yet we note that its floating point unit (FPU), which contributes more than 95% computation time in benchmark programs like LINPACK [Dong 03] as well as many other floating point applications like Monte Carlo simulation and N-body problem, constitutes only 10% of the total roadrunner computation chip (Cell Broadband Engine) area [Flac 05]. The other 90% serves to provide the FPUs with data and perform tasks such as caching, instruction fetching, memory decoding, speculative execution, register files, etc. In addition, the high power consumption of general-purpose processors prohibits the use of floating point arithmetic in low cost embedded systems. We believe that using spatially-parallel hardware oriented techniques with a cluster of FPUs often has advantages in terms of performance, power consumption and area over the traditional general-purpose processor approach.

It is possible to build a dedicated circuit for a specific floating point application using application specific integrated circuit (ASIC) technology, which offers the potential to achieve the highest performance with the least power consumption and area. However, the associated fabrication cost and design time preclude their use in low to medium volume applications, and ASIC designs are not flexible since the circuits cannot be changed once they are fabricated. Another way of implementing floating point applications is to use field programmable gate array (FPGA) technology. An FPGA contains an array of logic gates and storage elements, in which the functionality and interconnection can be configured by downloading a bitstream into its configuration memory. Given a flexible FPGA architecture, a tailor-made datapath and an FPU cluster, a floating point application is likely to have faster execution speed and lower power consumption than general-purpose processors. FPGA technology has been successfully applied to accelerate a large number of diverse applications including signal processing, communications, networking and robotics. The application of FPGA technology to computational problems is also known as reconfigurable computing.

In recent years there has been a significant increase in the size of FPGAs. Current FPGA technology allows arbitrary precision floating point arithmetic while retaining hardware speed. Recent work on dot product, matrix-vector and matrix multiplication [Unde 04] indicates that FPGAs will soon be able to significantly outperform modern microprocessors because of advantages in memory bandwidth and in floating point performance. Another study [Dou 05] shows that an FPGA-based FPU implementation can achieve 15.6 GFLOPS (billion floating point operations per second) with 1.6 MB local memory and a 400 MB/s external memory bandwidth. Our previous research [Ho 02a] indicated that using different arbitrary size of floating point operation in a single design can reduce the circuit area while the accuracy can remain the same. In addition, another work [Zhan 05] shows that an FPGA-based implementation of BGM financial model running at 50MHz is over 25 times faster than software computations on a 1.5 GHz Intel Pentium 4 machine. However, as the current FPGA architecture only embeds blocks for fixed point operations such as fast carry-chains and block multipliers, it is expected that the computation speed can be made even faster and the power consumption can be lower if more primitive blocks optimised for floating point operations are embedded in FPGAs.

Having better floating point performance in terms of speed, area and power consumption is beneficial to several domains of applications. For example, in Monte Carlo simulation models, increasing performance of floating point operations will allow more paths to be simulated, and therefore the result will be more accurate and faster to converge. Financial applications which require real-time response can meet stringent timing requirement with less hardware, and therefore reduce the associated cost. Graphics applications can process more transformations to produce more realistic effects. Reduction in power consumption of floating point operations allows longer battery life for embedded systems, significantly improving their effectiveness.

The research proposes a *platform* for developing and using novel reconfigurable architectures which improve the execution speed of floating point computations while retaining their programmability. In particular, we propose a standard island-style FPGA architecture coupled with novel reconfigurable heterogeneous fabric to facilitate the *reconfigurable* aspect of this platform. By adopting FPGA architectures, one can implement different applications and datapaths on FPGA devices *after fabrication*.

The platform consists of tools allowing *customisation* of reconfigurable devices *before fabrication*. Using standard macrocell design flow, one can customise their own reconfigurable device by specifying architecture parameters of the heterogeneous fabric. Such customisation enables further optimisations on domain-specific applications. Moreover, the platform consists of high level synthesis tools allowing users to implement floating point applications into the novel FPGA device using higher abstraction of description. The high level synthesis tools also assist the design and development of heterogeneous fabric.

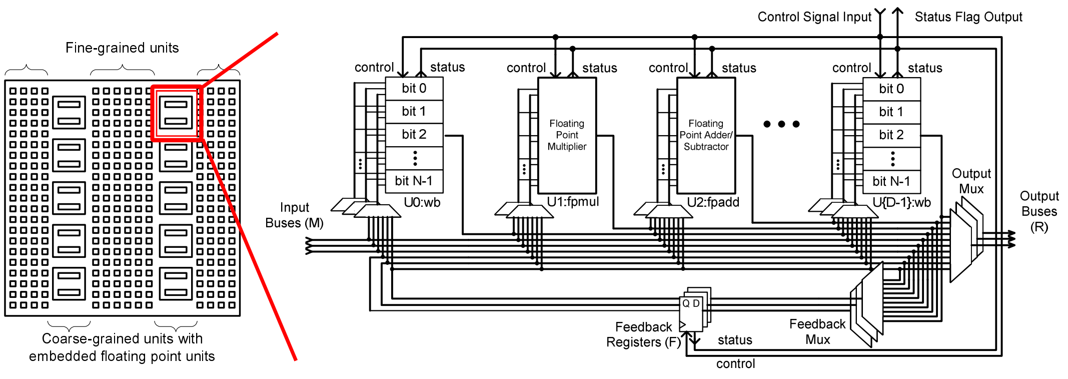


Figure 1.1: Floating Point FPGA.

We propose a hypothesis that certain specific reconfigurable architecture can accelerate floating point computations with much less silicon area and energy. To verify this hypothesis, we evaluate instances of reconfigurable architecture based on a set of benchmark circuits. Figure 1.1 shows an instance of a typical customisation of such reconfigurable architecture. More details are discussed in Chapter 5.

To summarise, a *customisable* and *reconfigurable platform* for optimising floating point computations has been developed, with the following contributions.

(A) Modelling: To model the proposed devices and compare with existing FPGA devices, this thesis proposes a modelling methodology by using existing vendor tool to estimate the area, delay and power consumption of the devices.

(B) Synthesisable Datapath: The proposed architectures capture the common pattern appearing in floating point datapaths such as bus-based logic and routing. By exploiting shared configuration bits, we propose a datapath-style coarse-grained reconfigurable fabric. In addition, we adopt synthesisable design flow allowing user customisation.

(C) Floating point FPGA: We propose an FPGA device which consists of island-style fine-grained fabric for general purpose computations and datapath-style coarse-grained fabric for floating point computations. The coarse-grained fabric contains dedicated circuitries for floating point operation and is customisable according to domain-specific requirements.

(D) Application design flow: A high level design flow is proposed which can translate a high level description of an application into a reconfigurable implementation. The key component in the high level design flow is a technology mapper which can map a given dataflow graph of an application into reconfigurable devices with different architectural parameters.

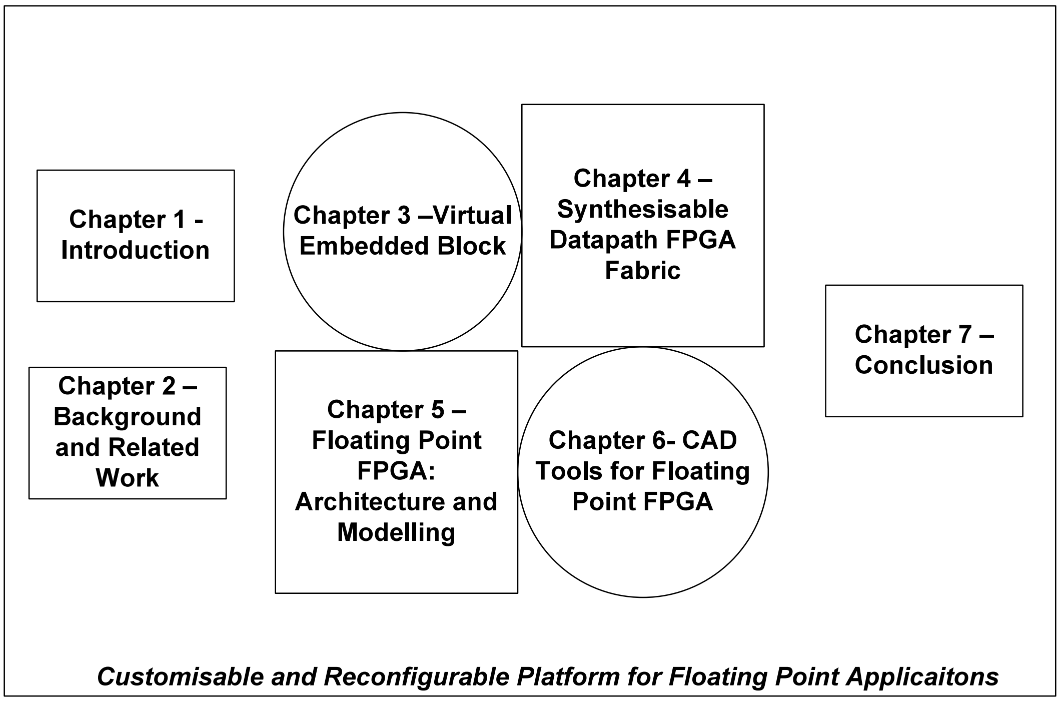

Chapter 1 - Introduction
Chapter 3 –Virtual Embedded Block
Chapter 4 – Synthesisable Datapath FPGA Fabric
Chapter 7 – Conclusion
Chapter 2 – Background and Related Work
Chapter 5 – Floating Point FPGA: Architecture and Modelling
Chapter 6- CAD Tools for Floating Point FPGA
Customisable and Reconfigurable Platform for Floating Point Applicaitons

Figure 1.2: Relationship diagram of each chapter.

Figure 1.2 provides a relationship diagram of each chapter in the thesis. Each block corresponds to a chapter in the thesis. Circular blocks indicate contributions related to software aspects while square blocks indicate contributions related to hardware aspects. Rectangular blocks refer to auxiliary materials.

Chapter 2 offers a comprehensive literature review of related work. The chapter introduces island-style FPGA architecture, the associated CAD tools to model and to program FPGA devices, the floating point number system, the FPGA-based implementations of floating point operators and benchmarks circuits employed to evaluate floating point FPGA architectures. The circuits can be as small as a simple dot-vector product or as large as interest rate model derivatives.

Chapter 3 demonstrates a methodology to model a commercial FPGA with arbitrary embedded blocks. This chapter corresponds to Contribution A and it addresses one of the important challenges in this thesis – how to conduct a meaningful comparison with real FPGA.

Chapter 4 proposes a customisable and reconfigurable heterogeneous architecture and a synthesisable design flow which can model this architecture. The parameterised architecture allows us to search for a near-optimum floating point FPGA design. This chapter corresponds to Contribution B and it is an initial attempt to approach new architecture for floating point computation.

Chapter 5 presents a novel floating point FPGA architecture by extending the heterogeneous architecture discussed in Chapter 4 and employing the FPGA modelling methodology used in Chapter 3. This chapter corresponds to Contribution C and we evaluate the performance of proposed FPGA architecture in area, speed, and dynamic power consumption.

Chapter 6 describes a high level synthesis flow to support the proposed reconfigurable architecture. The chapter demonstrates an architecture-aware technology mapper which can be integrated into existing hardware compiler to produce circuits implemented on the proposed reconfigurable device. This chapter corresponds to Contribution D.

Conclusion of the thesis and suggestions for future work are made in **Chapter 7**.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter presents an introduction to the concepts and terminology relevant to the thesis. It covers both *development* and *usage* of reconfigurable devices. The chapter begins with a brief overview of standard FPGA architecture. The concepts of fine-grained units and heterogeneous blocks are explained with published work as examples. The development of current floating point units on commercial FPGA devices is described and the operations involved in floating point circuitry are illustrated. This chapter reports a set of FPGA-based floating point operators and they are used as baseline to evaluate our work. We also show that these operators are compliant with the IEEE 754 standard. It is crucial to maintain the integrity of the thesis such that a justified comparison is allowed. Examples which demand intensive floating point operations are presented to demonstrate the importance of high speed floating point arithmetic. It is followed by an introduction to a set of benchmark circuits which are used iteratively in the thesis. The benchmark circuits include small kernels which capture common operations used in digital signal processing and linear algebra computations. The selected kernels require intensive floating point operation. The chapter concludes number of design flows to model reconfigurable architecture. Finally, a list of termi-

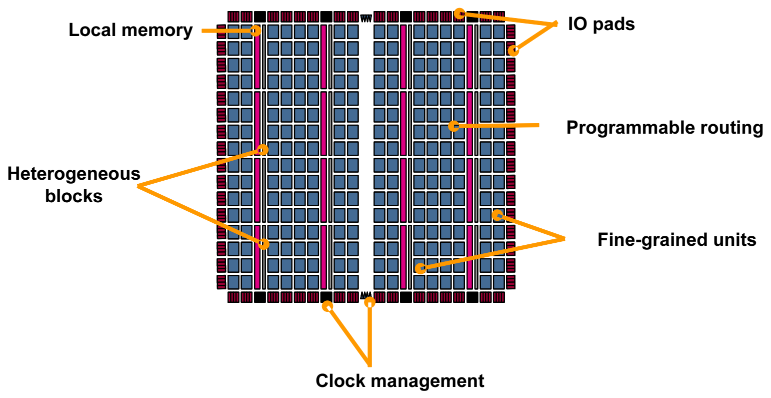


Figure 2.1: Standard island-style FPGA architecture.

nology is given to summarise some acronyms and technical terms uses iteratively in the thesis.

2.2 FPGA Architecture

Figure 2.1 shows the block diagram of a standard island-style FPGA structure. An FPGA is made up of reconfigurable fabric. The fabric itself consists of arrays of fine-grained units and heterogeneous blocks. A fine-grained unit usually implements a single function and has a single bit output. The most common fine-grained unit is a K-input lookup table (LUT), where K typically ranges from 4 to 6. The LUT can implement any Boolean equation of K-inputs. This type of fabric is called a LUT-based fabric. Several LUT-based cells can be joined in a hardwired manner to make a cluster. This results in little loss in flexibility but can reduce area and routing resources within the fabric [Ahme 04].

Fine-grained units can also be implemented using a product-term block consisting of an AND plane and an OR plane. The area of a product-term block is usually larger than that of a LUT-based fabric, as it usually has larger fan-in along with small amounts of routing resources to connect the planes. We consider the product-term unit to be a fine-grained unit, because it usually has a small number of output bits. Product-term blocks appear in system-on-chip [Yan 06] as well as commercial CPLD devices.

While a fine-grained unit is flexible and can usually implement any Boolean function, the area, delay and power overhead of an array of fine-grained units that implement a given function are often significantly larger than an appropriate heterogeneous block. Commercial FPGAs, which employ fine-grained fabric as the major component, include special features in the fabric dedicated to operations which are common in digital design. A notable example is the dedicated carry-chain on both Xilinx and Altera devices. The reason for adding such feature is obvious - integer addition and subtraction are common operations for all digital circuits. Multiplexers are another example, as they are inferred frequently in a digital design.

A heterogeneous block is usually less flexible and is typically much larger than a fine-grained one, but is often more efficient for implementing specific functions. The heterogeneous block is usually programmable to some degree, combining several functions such as those in an arithmetic logic unit (ALU). Outputs are often bus-based. They can be parameterised in terms of features such as bus-width and functionality. As an example, the ADRES architecture [Mei 03] assumes that the wordlength and the functionality of a heterogeneous block is the same as the targeted processor.

Heterogeneous functional blocks are found on commercial FPGA devices. For example, a Virtex II device has embedded fixed-function 18-bit multipliers and a Xilinx Virtex 4 device has embedded DSP units with 18-bit multipliers and 48-bit accumulators. The flexibility of these blocks is limited and it is less common to build a digital system solely using these blocks. When the blocks are not used, they consume die area and may contribute to increased delay without adding to functionality.

Numerous research projects on FPGA architecture to support domain-specific applications have been conducted. Leijten-Nowak and van Meerbergen [Leij 03] proposed mixed-level granularity logic blocks and compared their benefits with a standard island-style FPGA using the Versatile Place and Route tool (VPR) [Betz 99]. Ye, Rose and Lewis [Ye 03] studied the effects of coarse grained logic cells and routing resources for datapath circuits, also using VPR.

Kuon [Kuon 07] has reported the effectiveness of embedded elements in current FPGA devices by comparing with the equivalent ASIC circuit under 90*nm* technology process. Akan'Ova et al. [Aken 05] has demonstrated a standard-cell-based FPGA with improving performance using a structural design and layout approach. Compton and Hauck [Comp 04] have suggested a flexibility measurement on domain-specific reconfigurable architecture. Beck revised VPR to explore the effects of introducing hard macros [Beck 04]. A more recent VPR tool [Luu 09] supports single-driver routing configuration, and heterogeneous blocks, and provides optimised electrical models.

Several previous studies have considered datapath-oriented FPGAs [Cher 96, Hauc 04, Leij 03, Ye 06, Ye 03]. In these architectures, configuration bits are shared among multiple lookup-tables and multiple routing switches.

Coarse-grained architectures, in which lookup-tables are replaced by ALUs, have also been described in [Cron 99, Gold 00, Mars 99, Sing 00, Ebel 96]. Of these, the RaPiD architecture [Ebel 96] is specifically designed for use in an SoC. RaPiD contains a linear array of dedicated functional units connected using dedicated buses. Control logic is implemented using a separate module that provides control signals to the functional units.

RaPiD is intended to support fairly large applications such as image and signal processing, and may be best implemented as a hard programmable logic core. It would be possible to "scale down" RaPiD and use it as a synthesisable core. However, like the datapath FPGAs described in the previous section, the unconfigured RaPiD fabric contains combinational loops. Our architecture eliminates these using a directional routing network.

While many studies can satisfy certain domain-specific applications, they fail to recognise the applications which demand intensive floating point computations. Our project aims at inventing methodology and architecture to produce an FPGA optimised for floating point computations.

There are a few research projects dedicated to FPGA architecture for floating point computations. Beauchamp et al. augmented VPR to assess the impact of embedding floating-point units in FPGAs [Beau 08]. The study of embedded heterogeneous blocks for the acceleration of floating point computations has been reported by Roseler and Nelson [Roes 02]. Both studies conclude that employing heterogeneous blocks in designing FPU on FPGAs achieves area saving and increased clock rate over a fine-grained approach.

However, the work in [Roes 02] does not take account of the architectural modification of the FPGA device and solely adopts existing heterogeneous blocks in FPGA device to design floating point units. Our research considers any potential embedded elements, including embedded floating point unit, or embedded floating point operators in the design of fabric. It is described in Chapter 5.

While [Beau 08] evaluate the results by employing a modified VPR flow, where floating point unit model is added to the VPR design flow, their work inherits the limitation given by VPR. For instance, as direct comparison to commercial FPGA device cannot be made, the results may not reveal the actual situation. In addition, since their work does not consider any routing resource optimisation as well as bus-based logic optimisation, their reported results may tend to be too conservative. This project proposes a model which is comparable to existing FPGA device and that could produce more realistic results. This methodology is discussed in Chapter 3.

2.3 FPGA Design Tools

Different strategies have been proposed to model FPGA architectures. The VPR computer aided design (CAD) tool [Betz 97, Luu 09], originally developed by Betz and Rose, supports parameterised island-style FPGA architectures. It can place and route designs and can be used to estimate performance. However, the model of the reconfigurable fabric is obsolete and most commonly available features such as carry-chains cannot be modelled without proper modification to the tool, in which the modification requires significant understanding of the software. In addition, there is no commercial quality synthesis tool to support the VPR tool. This prohibits the use of VPR as it is difficult to implement relatively large circuit defined by users.

Yan and Wilton [Yan 06] employ a synthesisable flow to model reconfigurable fabric. They describe the architectures of the fabric using hardware description language (HDL) and synthesis it with standard cell library design flow. The area and timing information can be obtained directly from the synthesis tool. The model also facilitates rapid evaluation because of the mature ASIC standard cell library design flow. However, it is usually not the most optimum ASIC implementation because of the limitation of the standard cell library design flow. A full-custom ASIC design flow can usually implement the same model with less area and shorter delay. More information regarding to the FPGA modelling is presented in Chapter 3

In terms of high level synthesis on an FPGA device, several schemes such as ASC [Menc 06], Handel-C [Agil 07], Trident [Trip 07], the fly compiler [Ho 02b] are proposed. ASC, also known as a stream compiler, provides a software-like programming interface to hardware design while at the same time keeping the performance of manually-design circuits. It allows existing C/C++ code be seamlessly transformed to ASC code to increase productivity and generate a large selection of implementations. The user can choose the most suitable design from them.

Handel-C is a language that is similar to ANSI-C but dedicated to hardware design. It allows parallel execution constructs and offers a software-influenced hardware design methodology. It can produce a register transfer level netlist based on a code written in C language.

Fly compiler adopts similar semantic to Handel-C. However, the core is simple and lightweight in which new constructs can be easily integrated into the compiler. This facilitates high level synthesis research and this project employs the fly compiler to produce different experiments efficiently. Furthermore, it is possible to extend the fly compiler such that it can support the proposed FPGA architecture and this is illustrated in Chapter 6.

Studies have been made in optimising floating point operations on existing FPGA architecture. Langhammer [Lang 08] proposes a datapath circuit generator which optimises single precision floating point operations on traditional FPGA devices. The improvements in area and speed are achieved by exploiting fused operations, in which consecutive floating point operations are analysed, redundant normalisations in the operations are removed and a new type of operation is created to replace the original operations.

2.4 Floating Point Number System

2.4.1 Overview

Every real number can be approximated by a floating point number in the IEEE 754 standard [ANSI 85] 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 normalised 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 754 floating point number F can be expressed as follows:

$$F = (-1)^s \times 1.f \times 2^{e-b}, \text{where} \tag{2.1}$$

$$b = 2^{e_{size}-1} - 1 \tag{2.2}$$

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 an *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 2.2. 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 normalised. Subnormal numbers are indicated by the exponent being 0. In this case, $F = 0.f \times 2^{-126}$ is represented.

2.4.2 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 $F_{minus} = F_1 - F_2$ can be rewritten as $F_{minus} = F_1 + (-F_2)$. Therefore, this section discusses 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 754 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 size of fraction field is 2, $1.11 \cdot 2^{10} + 1.00 \cdot 2^{-2}$ 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 6 times the size of fraction. When the difference of exponent is larger, the size of intermediate result is larger too.

Without using infinite precision for storing 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 known as guard bit and round bit respectively. The guard bit can guarantee the relative rounding error in the result is less then 2ϵ, where ϵ is referred to as machine epsilon, the smallest value that can be represented under the given exponent. The round bit can guarantee the rounding to zero mode is always correct [Gold 91]. In general, the sum of F_1 and F_2 is evaluated as shown in Listing 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 assumes that it uses 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.

Listing 1: Calculate $F_1 + F_2$ with floating point arithmetic

```
Data: $F_1 = (s_1, e_1, f_1), F_2 = (s_2, e_2, f_2)$
Result: $F_{ans} = (s_{ans}, e_{ans}, f_{ans}) = F_1 + F_2$
$e_{diff} \Leftarrow e_1 - e_2$
if $e_{diff} \geq 0$ then
    $f_a \Leftarrow f_1$
    $f_b \Leftarrow f_2$
    $e_s \Leftarrow e_{diff}$
else
    $f_a \Leftarrow f_2$
    $f_b \Leftarrow f_1$
    $e_s \Leftarrow$ 2's complement of $e_{diff}$
end
if $s_a = 1$ then
    $rm_a \Leftarrow$ 2's complement of $f_a$
end
if $s_b = 1$ then
    $rm_b \Leftarrow$ 2's complement of $f_b$
end
$f_a \Leftarrow (``001''\#\#f_a)$
$f_b \Leftarrow (``001''\#\#f_b)$
$f_b \Leftarrow$ shift $f_b$ right by $e_{diff}$ bits
$f_{tmp} \Leftarrow rm_a + rm_b$
if $f_{tmp}$ is negative then
    $f_{tmp} \Leftarrow$ 2's complement of $f_{tmp}$
    $s_{ans} \Leftarrow 1$
else
    $s_{ans} \Leftarrow 0$
end
find the leading one of $f_{tmp}$
shift $f_{tmp}$ left until $f_{tmp}(msb) = 1$,
$e_{ans} \Leftarrow e_a$ - number of bits shift to left. $msb$ is the location of most significant bit
remove the integer bit, $f_{ans} = f_{tmp}(msb - 1...0)$
return $s_{ans}$, $e_{ans}$ and $f_{ans}$ as sign bit, biased exponent field and fraction field
    respectively
```

2.4.3 Multiplication

Multiplication is simpler than addition assuming that a fixed point multiplier is provided. The product of F_1 and F_2, where both F_1 and F_2 are normalised floating point numbers, is evaluated as in Listing 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.

Listing 2: Calculate $F_1 \times F_2$ with floating point arithmetic

```
Data: $F_1 = (s_1, e_1, f_1), F_2 = (s_2, e_2, f_2)$
Result: $F_{ans} = (s_{ans}, e_{ans}, f_{ans}) = F_1 \times F_2$
$s_{ans} \Leftarrow s_1 \oplus s_2$
append 1 bit "1" to $f_1$ and $f_2$ at left as the hidden integer
field
$v_1 \Leftarrow$ "1"##$f_1$
$v_2 \Leftarrow$ "1"##$f_2$
do fixed point unsigned multiplication $mc \Leftarrow v1 \times v2$
$r_{e1} \Leftarrow e_1 + e_2 - b$
shift $mc$ to left until msb of $mc$ is 1
$e_s \Leftarrow$ number of bit shifted to left
$e_{ans} \Leftarrow r_{e1} - e_s$
$f_{ans} \Leftarrow mc(44...22)$
return $s_{ans}$, $e_{ans}$ and $f_{ans}$ as sign bit, biased exponent field and fraction field respectively
```

2.5 FPGA-based Floating Point Units

Table 2.1 presents a list of published information of FPGA-based implementation. Jaenicke and Luk [Jaen 01] have implemented parameterised floating point adder and multiplier on FPGAs. The design is based on Handel-C language and the data format is variance of IEEE 754 standard. It is reported that the floating point adder can perform 28 MFLOPS (million floating operations per second) 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. Belanovìc et al [Bela 02] implemented a parameterised floating

	Jaenicke [Jaen 01]	Belanovic [Bela 02]	Dido [Dido 02]	OpenFPU [Rudo 05]	Xilinx Coregen [Xili 05]
Target device	Xilinx XCV1000	Xilinx XCV1000	Xilinx VirtexE	Xilinx XC2V1000	Xilinx XC2V1000
Adder speed	28MHz	–	140MHz	137MHz	158MHz
Multiplier speed	28MHz	–	140MHz	142MHz	176MHz
Data format	32-bit single precision	32-bit single precision	16-bit (6-bit exponent, 9-bit fraction)	32-bit single precision	32-bit single precision
Latency	5	3	5	5	5
Parameterised bitwidth	yes	yes	no	no	yes
Rounding mode	nearest	zero nearest	nearest	zero nearest positive ∞ negative ∞	nearest
Subnormal number	yes	yes	no	yes	no
Open source	no	yes	no	yes	no
Year	2001	2002	2002	2005	2005

Table 2.1: FPGA-based floating point operators implementations.

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 is used to develop a hybrid implementation of the K-means clustering algorithm applied to multi-spectral images.

Dido et al. [Dido 02] proposed a flexible floating point format which is optimised for video signal processing application. The format employs moderate bitwidth but it can maintain sufficient output accuracy. This can deliver better performance and consume less area with acceptable trade-off on accuracy.

To allow comparison between traditional FPGA device and proposed reconfigurable devices, floating point operators which support arbitrary precision and floating point benchmark circuits have been developed in HDL model and implemented on commercial FPGA devices. The floating point operators are fully compliant with IEEE 754 [ANSI 85] standard and support 4 rounding modes, subnormal numbers and exceptions. In addition, they are fully-pipelined and arbitrary size of exponent and fraction are allowed by modifying the model slightly. To support parameterised floating point operators, a HDL generator is developed using the Perl language which can generate the associated logic for a specific size of exponent and significant on-the-fly.

Floating Point Adder – The floating point adder is based on a heavily modified open-source floating point unit [Rudo 05]. It consists of several blocks, namely, a pre-normalisation block, an addition block, a post-normalisation blocks and an exception handling block. The datapath of the floating point adder is shown in Figure 2.2. The figure captures essential steps in computing floating point addition as shown in the Listing 1. In the pre-normalisation stage, the inputs are registered and the exponents are compared. The inputs are swapped if the exponent of first operand is smaller than the second operand. The fractions are shifted to right accordingly and operation mode

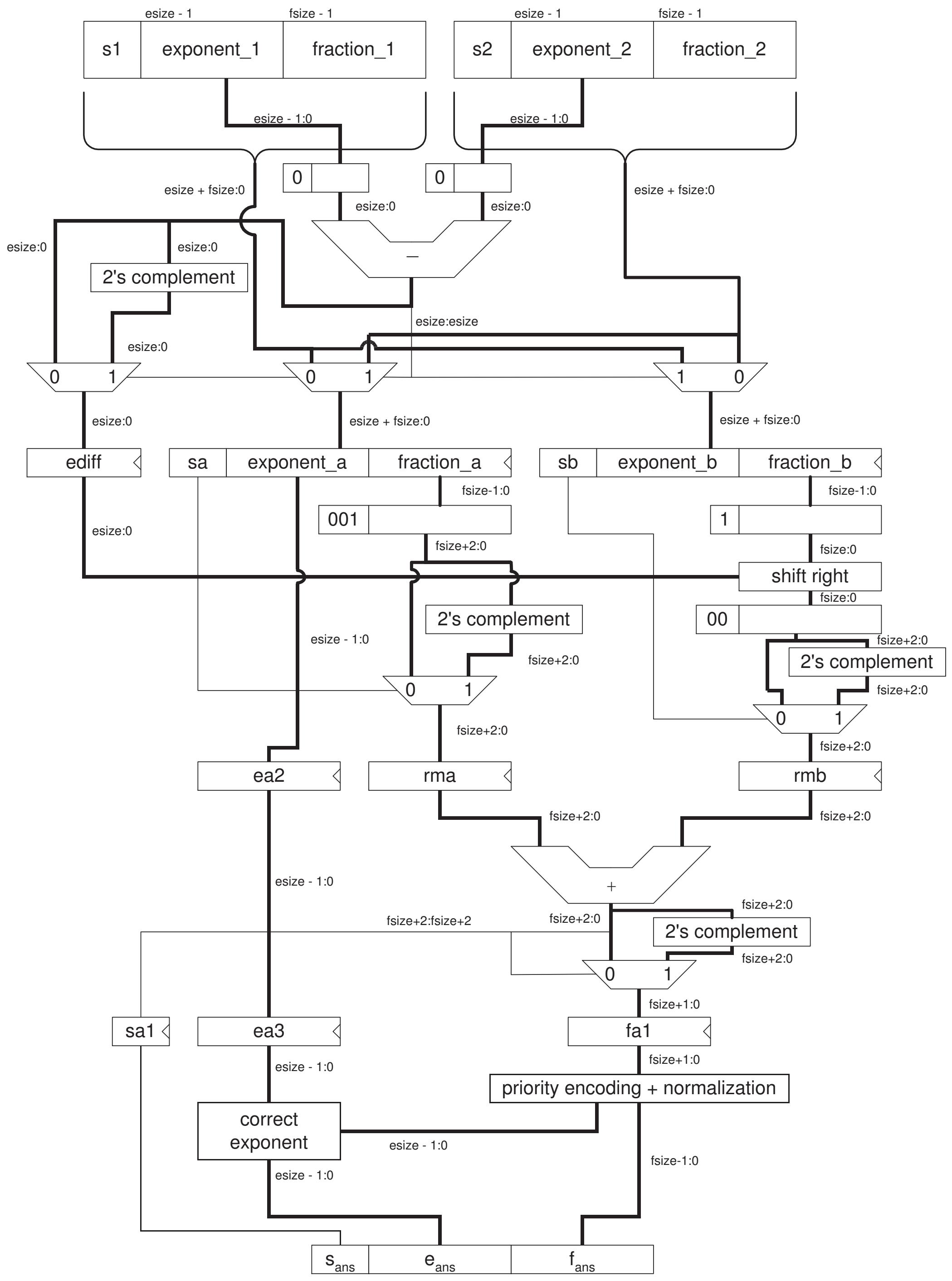


Figure 2.2: Datapath of floating point adder.

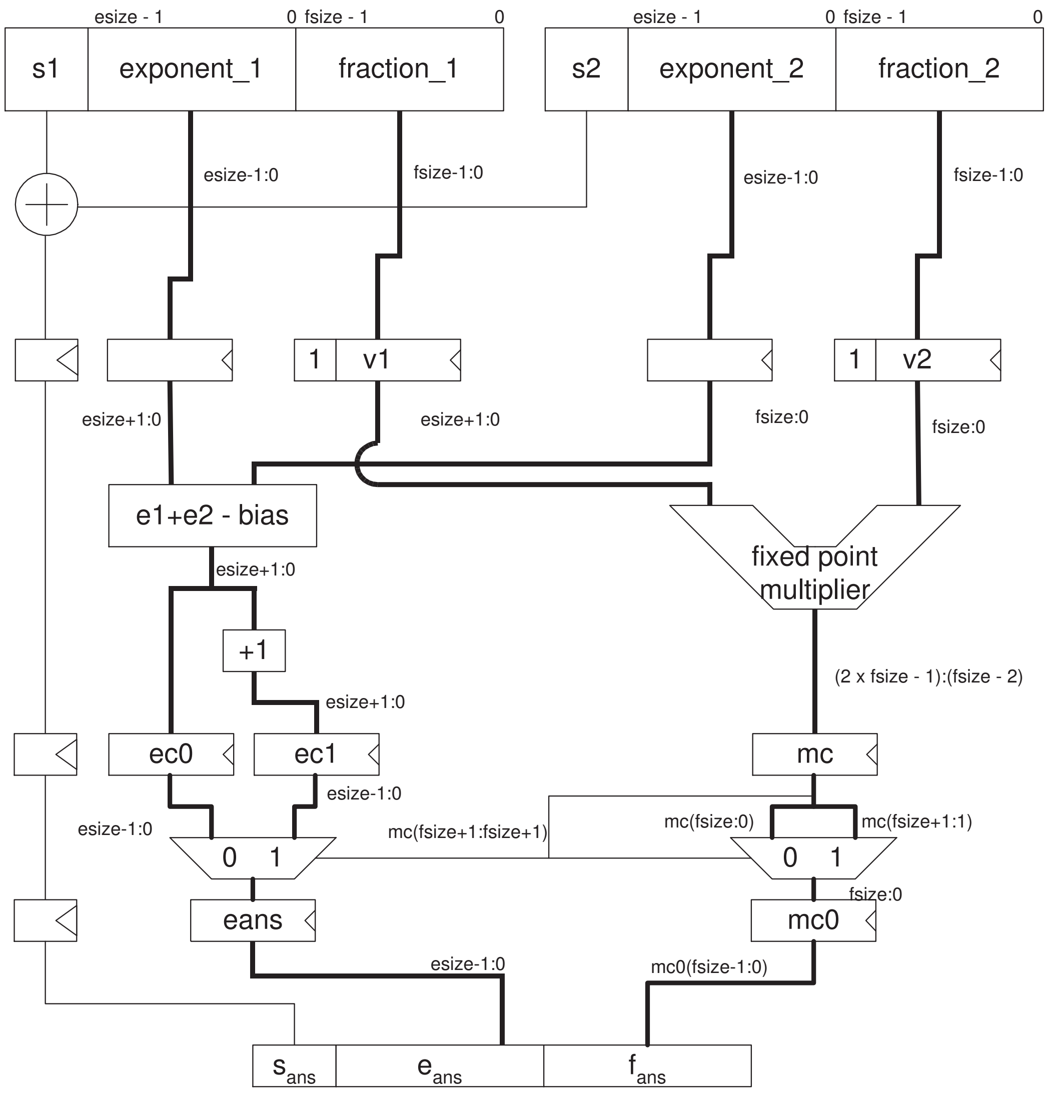


Figure 2.3: Datapath of floating point multiplier.

which indicates if the effective operation (either addition or subtraction) is evaluated. The most expensive circuit for the pre-normalisation stage is the barrel shifter.

Special number from the input such as subnormal number, infinity and not a number (NaN) are handled in the exception handling block. Corresponding flags, such as subnormal flag, zero flag, infinity flag, NaN flag are set according to the combination of the input. This circuit is simple and only comparators are required. The addition block takes the output from the pre-normalisation block, in which the data has been properly aligned and the operation mode is well defined. The addition block adds or subtracts the numbers according to the operation mode.

The post-normalisation block is the most complicated circuit in the floating point adder. After the intermediate result is generated by addition block, a priority encoder inside the post-normalisation block takes the result as an input and counts the number of leading zero of the result. The exponent is then adjusted and the fraction is shifted to left depending on the number of leading zero. Different rounding scheme is enforced according to the input to produce final result. Exception flags like inexact number, overflow, underflow is generated based on the final result. All the outputs are registered so the result and the corresponding exception flags are given in next clock cycle. This block contains two expensive circuits, namely barrel shifters and a priority encoder.

Floating Point Multiplier – Figure 2.3 illustrates the datapath of floating point multiplier. Same as the floating point adder, the floating point multiplier is based on the same open-source floating point unit, and it has been heavily modified. It consists of several blocks, namely, a pre-normalisation block, a multiplication block, a post-normalisation block and an exception handling block. In the pre-normalisation stage, the intermediate exponent is determined by adding exponents from the inputs. Hidden bits of the fractions are recovered based on the exponent values. This block does not have expensive circuits and most of them are comparators and adders.

The exception block of floating point multiplier is the same as the one in floating point adder. It detects any special input values. The multiplication block takes the output from the pre-normalisation block, in which the hidden bits in fraction has been recovered properly and an integer multiplication is computed by a multiplier. The results are then populated to post-normalisation block. The multiplier circuit consumes significant amount of resource in this block.

The post-normalisation takes the intermediate product from the multiplication block as input. A priority encoder in the post-normalisation block counts the number leading zero in the intermediate product. Similar to the post-normalisation block in the floating point adder, the exponent is then adjusted and the fraction is shift to the left based on the number of leading zero. Different rounding mode is enforced according to the input to produce final result. Exception flags like inexact number, overflow, underflow is generated based on the final result. And all the outputs are registered and the final result and the associated flags are given in next clock cycle. This block contains two expensive blocks, namely barrel shifters and a priority encoder.

Verification – To verify the correctness of the floating point operators so that they are compliant with the IEEE 754 standard, an open-source program called *TestFloat-2a* [Haus 98] is employed. By slightly modifying the output options in *TestFloat-2a* program, it can create a large number of test cases, which make up of simple pattern tests intermixed with weighted random inputs for the floating point operators. The “level 1” test in *TestFloat-2a* covers all 4 rounding modes, and all boundary cases of given arithmetic, including underflows, overflows, invalid operations, subnormal inputs, zeros (positive and negative), infinities (positive and negative), and NaNs. Each test case contains an operation, a rounding mode, floating point numbers to be evaluated, an expected result and expected exception flags. The expected results and the expected exception flags are computed purely in software and do not rely on machine-specific floating point implementations. A corresponding testbench written in Verilog is created which reads the test cases generated by *TestFloat-2a*, invokes the corresponding floating point operator to compute the result, and compares the result and the exception flags with the expected output. The test case is created with switch

Floating Point Operation	Number of tests
Floating Point Addition/Subtraction	371712
Floating Point Multiplication	371712

Table 2.2: Number of tests generated by *TestFloat-2a*.

Operator	Slices	Embedded Multiplier	Latency	Frequency (MHz)
fpadd2	1777	0	5	134
fpmul2	2150	9	5	76

Table 2.3: FPGA implementation results for double precision floating point operators.

"level 1" in the initial settings of *TestFloat-2a*. Table 2.2 shows the number of test vectors has been created for specific floating point operation. All test cases assume double precision floating point format. The testbench is run in ModelSim 5.7d and no errors are found.

Implementation – In order to compare with the floating point core using current commercial FPGA with the one using *customisable* FPGA, the floating point operator circuits have been implemented on FPGA device. The reference FPGA device is XC2V6000 and the speed grade is -5. All the design is synthesised using Synplify Premier 9.0. The designs are placed and routed and the area and the timing are obtained by vendor CAD suites ISE 9.2i. Table 2.3 presents the area and frequency of the double precision floating point adder and multiplier.

2.6 Floating Point Applications

Many floating point systems have been implemented on FPGA devices. In [Ho 03], an N-body solver is developed using Virtex-E device. The computations are based on parameterised floating point library and can achieve a peak speed of 3GFLOPS. In [Unde 04], three of the basic linear algebra subroutine (BLAS) functions are estimated and it suggests that FPGA-based implementations outperform modern general-purpose processor on double precision floating point operations. It also mentions that unlike CPUs, FPGAs are usually limited by peak FLOPS rather than by memory bandwidth so improving the floating point computation performance of an FPGA can obtain similar gain on overall systems.

Zhang et al. [Zhan 05] employ floating point arithmetic to compute the Brace, Gątarek and Musiela interest rate model for pricing derivatives. While running at relatively low frequency (50MHz), the performance is 25 times faster than software running on a 1.5GHz Intel Pentium 4 machine.

Callanan et al. [Call 06] demonstrate an FPGA based lattice QCD processors using IEEE double precision floating point format and compared with corresponding ASIC based solutions and PC cluster-based solutions. The FPGAs version, which is implemented on a Virtex II FPGA device, can achieve 1.2GFLOPS when performing Dirac operation and deliver 0.94GFLOPS on conjugate gradient solver. The performance of Dirac operation is two times better than purely software implementation.

Zhuo and Prasanna [Zhuo 04] propose an FPGA-based architecture for floating point matrix multiplication. It employs a linear array architecture and effectively utilises the hardware resources on the entire FPGA device while reduces the routing complexity. Their work achieves comparable floating point computation performance and can deliver up to 26.6GFLOPS and 12.3GFLOPS for single precision and double precision floating point format respectively.

Morris and Prasanna [Morr 05] report an FPGA-based floating point Jacobi iterative solver. The design employs a deeply pipelined, highly parallelised IEEE double precision floating point operator. The solver is implemented on a Virtex II Pro device running at 77MHz. Depending on the nature of input data, it can achieve up to 36.8 times speedup when compared with a single processor implementation.

2.7 Benchmark Circuits

As there are no existing standard benchmark circuits for floating point applications, a set of benchmark circuit is implemented using the fly compiler [Ho 02b] or using HDL. Significant amount of time in developing the benchmark circuits are reduced as the fly compiler can generate a circuit which contains a datapath and associated control signals from a software description.

In addition, all the floating point application benchmark circuits assume double precision floating point arithmetic and employ round-to-nearest-even rounding mode, while the exception signals from the floating point operator are ignored in the circuits. By describing the application using Perl-like description and simulate it in Perl environment, the fly compiler can speed up the implementation time of the benchmark applications. Four application circuits have been generated using the fly compiler, which include a digital sine cosine generator *(dscg)*, an ordinary differential equation solver *(ode)*, a 3-by-3 matrix multiplication *(mm3)*, a four-tap finite impulse response filter (fir4), a butterfly circuit for fast Fourier transform *(bfly)* and a financial derivatives modelling circuit using Brace, Gątarek and Musiela model *(bgm)* [Zhan 05]. These benchmark applications contain different number of floating point operators the inter-connection between those floating point operators are different. The benchmark applications further assume the input data comes from an off-chip memory.

2.7.1 Digital Sine-Cosine Generator *(dscg)*

Digital sine-cosine generator [Mitr 98] has a number of applications, such as the computation of discrete Fourier transform and in certain digital communication systems, such as in future Hiperlan systems 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:

$$\begin{bmatrix} s1_{n+1} \\ s2_{n+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \cos(\theta)+1 \\ \cos(\theta)-1 & \cos(\theta) \end{bmatrix} \begin{bmatrix} s1_n \\ s2_n \end{bmatrix} \tag{2.3}$$

2.7.2 Ordinary Differential Equation *(ode)*

Many scientific problems involve the solution of ordinary differential equations. An ODE solver (ode) is implemented as part of the floating point benchmarks. The benchmark circuit solves the ODE [Math 99]:

$$\frac{dy}{dt} = \frac{(t-y)}{2} \text{ over } t \in [0,3] \text{ with } y(0) = 1 \tag{2.4}$$

Euler method is used and y is approximated by

$$y_{k+1} = y_k + h\frac{(t_k - y_k)}{2} \text{ and } t_{k+1} = t_k + h \tag{2.5}$$

where h is the step size, the smaller value of h, the more accurate of the result.

The ordinary equation solver can take the step size h as the parameter and return the value of y.

2.7.3 Matrix Multiplication *(mm3)*

Matrix multiplication is used frequently in different domains. Hence a 3x3 matrix multiplication application benchmark circuit is developed. The core of the circuit implements the operation required to evaluate an element of the resulting matrix, which is a vector dot-product. Extra logic is added to control the dataflow of the circuit.

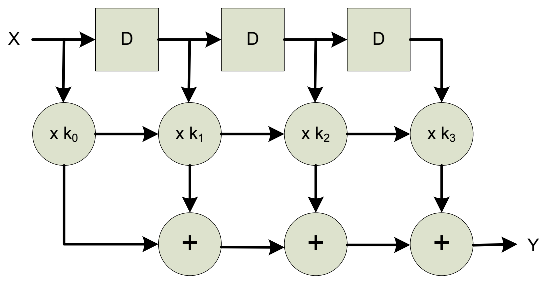


Figure 2.4: Four-tap FIR filter.

2.7.4 FIR Filter *(fir4)*

Digital filter is one of the most common applications which requires floating point arithmetic for high accuracy and precision, we have implemented a 4-tap finite impulse response filter, which is characterised by the following equation:

$$y_4 = \sum_{j=0}^{4} k_j x_{4-j} \tag{2.6}$$

where x_i is the input of the filter, k_i is the filter window and y_i is the output. The datapath of the filter is shown in Figure 2.4.

2.7.5 Butterfly Circuit *(bfly)*

The fast Fourier transform (FFT) is another important signal processing primitive. The FFT is composed from butterfly operations which compute $z = y + x \times w$, where x and y are the inputs from previous stage and w is a twiddle factor. All values are complex numbers; therefore each multiplication involves 4 multipliers and 2 adders (bfly). A state machine is implemented to control the dataflow of the circuits. Figure 2.5 illustrates the datapath of a single butterfly which is used as the benchmark circuit.

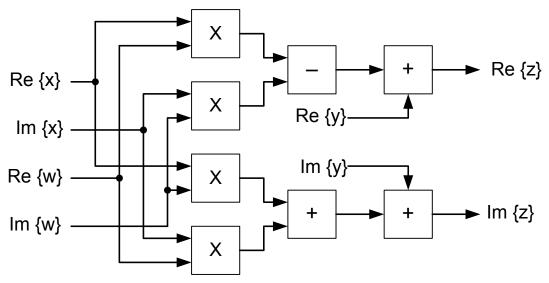


Figure 2.5: One butterfly stage in FFT.

2.7.6 Brace, Gątarek and Musiela *(bgm)*

The datapath of a design to compute Monte Carlo simulations of interest rate model derivatives priced under the Brace, Gątarek and Musiela (BGM) model is used as the final test circuit *(bgm)* [Zhan 05]. Denote $F(t, t_n, t_{n+1})$ as the forward interest rate observed at time t for a period starting at t_n and ending at t_{n+1}. Suppose the time line is segmented by the reset dates $(T_1, T_2, ..., T_N)$ (called the standard reset dates) of actively trading caps on which the BGM model is calibrated. In the model, the forward rates $\{F(t, T_n, T_{n+1})\}$ are assumed to evolve according to a log-normal distribution. Writing $F_n(t)$ as the shorthand for $F(t, T_n, T_{n+1})$, the evolution follows the stochastic differential equation (SDE) with d stochastic factors:

$$\frac{dF_n(t)}{F_n(t)} = \vec{\mu_n}(t)dt + \vec{\sigma}_n(t) \cdot d\vec{W}(t) \qquad n=1 \ldots N. \tag{2.7}$$

In this equation, dF_n is the change in the forward rate, F_n, in the time interval dt. The drift coefficient, $\vec{\mu_n}$, is given by

$$\vec{\mu_n(t)} = \vec{\sigma}_n(t) \cdot \sum_{i=m(t)}^{n} \frac{\tau_i F_i(t) \vec{\sigma}_i(t)}{1 + \tau_i F_i(t)} \tag{2.8}$$

where $m(t)$ is the index for the next reset date at time t and $t \leq t_{m(t)}$, $\tau_i = T_{i+1} - T_i$ and σ_n is the d-dimensional volatility vector. In the stochastic term (the second term on the right hand side of Equation 2.7), $d\vec{W}$ is the differential of a d-dimensional uncorrelated Brownian motion $\vec{W}$, and each component can be written as $dW_k(t) = \epsilon_k\sqrt{dt}$ where ϵ_k is a Gaussian random number drawn from a standardised normal distribution, i.e. $\epsilon \sim \phi(0, 1.0)$.

2.8 Terminology

Here is a list of special terminologies and the corresponding abbreviations used throughout the thesis.

- Field programmable gate array (FPGA).
- Application specific integrated circuit (ASIC).
- Floating point unit (FPU).
- Very high speed integrated circuits hardware description language (VHDL).
- Logic cell (LC) – It describes the smallest logic unit in the FPGA. This consists a 4-input LUT, register and dedicated carry logic.
- Logic block (LB) – It refers to an array of LCs which are interconnected via the local routing resources in the FPGA.
- Embedded block (EB) – It refers to heterogeneous element in the island-style FPGA. An EB usually contains specific function in an FPGA such as embedded multiplier and block memory.
- Virtual embedded block (VEB) – It refers to virtual heterogeneous element in the island-style FPGA. VEB can model arbitrary embedded block even if such block does not exist in real FPGA devices. More detail on VEB is given in Chapter 3.
- Fine-grained unit (FGU) – It refers to homogeneous element in the island-style FPGA. It is usually the same as LC unless otherwise specified. We use the term embedded block.

- Coarse-grained unit (CGU) – In later context of the thesis, we use the term CGU to describe a family of heterogeneous architecture proposed in this thesis. It can be considered as a subset of EB.
- Standard FPGA – It refers to commodity FPGA devices which are usually commercially available.
- Register transfer language (RTL).

2.9 Summary

This chapter introduces the work related to the project. Section 2.2 describes the common island-style fine-grained fabric and the application-specific heterogeneous fabric. Section 2.3 discusses the CAD tools for modelling an FPGA and the high level synthesis design tools for implementing user circuits on an FPGA. Floating point number system, including the number representation and primitive operations is covered in Section 2.4. Some published information of FPGA-based floating point units and the architecture of our FPGA-based floating point operators are summarised in Section 2.5. Section 2.6 introduces some floating point applications implemented on FPGA devices and their performance. Section 2.7 compiles a set of floating point benchmark circuits which are used iteratively in the thesis. Finally, Section 2.8 lists terminologies used in the thesis.

Chapter 3

Virtual Embedded Block

3.1 Introduction

Similar to designing ASIC circuitry, when designing FPGA architecture, one major concern is to identify a justified model which can estimate the performance in terms of area, delay and power consumption based on user-supplied applications.

Modelling ASIC circuitry is relatively simple once the netlist of the circuit is defined. Given a set of process parameters, the area of a circuit can be estimated by counting the number of transistors used with some assumptions on the routing. The timing or critical delay can be estimated by enumerating the delay of each combinational path.

Modelling reconfigurable architecture is a non-trivial process. The major difference between ASIC and reconfigurable devices is the latter one can change their datapath substantially after fabrication. Because of this fundamental difference, definitions of "area" and "delay" are not the same as those in ASIC. Area is not simply the transistor count of the device and delay is not the critical path of the device. Rather, we wish to adopt a definition of area and delay to reflect characteristics of applications implemented on the device.

An application implemented on an FPGA device does not consume all the reconfigurable resources on the FPGA. Only part of reconfigurable resources are used to perform required operations. Therefore, area is tied to the transistor count of the configured region in the device while delay is the critical path of the configured region in the device. When enumerating the delay of each path, the model has the ability to recognise the delay of each fine-grained, coarse-grained components and interconnect along the paths. It also adds complexity in modelling FPGA device.

FPGA vendors offer design tools for users to model and implement applications on their FPGAs. The tools often involve a tool which translates user applications described in HDL to an architecture-specific netlist. This process usually refers to synthesis. The netlist is then implemented in a reconfigurable device. The implementation process defines the configured region in the device. Area and timing can be retrieved according to the implementation results. The tools released by vendors support their own devices. In other words, the tools support different applications but are limited to specific devices.

In this research, we begin with modifying existing FPGA architecture by embedding user-defined heterogeneous blocks to reconfigurable device to accelerate floating point computations. We show that this approach allows us to reuse existing design tools to model new FPGA architecture and also allows us to compare the new FPGA architecture with the existing device.

To capture the performance of new FPGA architecture, we employ floating point applications as benchmarks. It is because we need to deal with more advanced architectures with different heterogeneous blocks which have multiple granularities. In addition, our application domains usually involve floating point arithmetic so we can select representative kernels or applications to further optimise the architectures. In order to establish a model for reconfigurable architecture for floating point computation, the following issues have to be considered:

1. The model can capture the performance by providing the area, delay and power data associated with the benchmark circuit.

2. The data obtained in 1 are comparable to existing FPGA devices.
3. The model can allow the reuse of existing CAD tools, such as synthesis tools, place and route tools, and mapping tools as much as possible.
4. The approach can provide sophisticated fine-grained model which is similar to the one used in commercial FPGA and still allows flexibility in creating customised coarse-grained model.
5. The model can allow rapid assessment such that "parameter sweep" architecture exploration approach is possible.

VPR [Betz 97, Luu 09] modelling flow has been widely employed to perform FPGA architecture modelling and the associated CAD research. VPR allows users to model arbitrary fine-grained architecture and routing of island-style FPGA. However, VPR does not consider some of the issues such as 2, 3 and 4. It results in unjustified comparison with existing FPGA. To address these issues, we propose a device and vendor independent methodology for rapid assessment of the effects of adding embedded elements to an existing FPGA architecture. The key element of our methodology is to adopt Virtual Embedded Blocks (VEBs), created from the FPGA's logic resources, to model the placement, delay and power consumption of the embedded block to be included in the FPGA fabric. Using this method, the benefits of incorporating embedded elements in improving application performance and reducing area and power usage can be quickly evaluated, even if an actual implementation of the element is not available. In addition, most commercial quality CAD tools are allowed. Therefore, we can achieve commercial quality timing and area results. For example, some optimisations such as retiming are possible during the synthesis stage. A summary of how the proposed methodology addresses those issues is provided in Section 3.6.

To measure the accuracy of this approach, block multipliers are modelled using VEBs and compared with FPGAs having this feature. A study of the benefits of double precision floating point embedded blocks is also made as an initial attempt to explore how embedded FPUs affect the performance of FPGA. Using this approach, the speedup of an application as a function of the speed of the embedded block can be easily quantified, and these studies are made for some of the benchmarks.

While this approach is adopted to model the reconfigurable architecture for floating point computation, the idea of VEB can generally be applied to model arbitrary embedded block. This chapter aims at provide a general overview of the VEB flow which is not limited to embed FPU or multiplier. We provide specific examples on how to use VEB flow to emulate an FPGA which has FPU embedded blocks and compare with the existing counterpart without FPU.

In this chapter, Section 3.2 describes how the idea of VEB can be implemented on a general manner. It is followed by vendor specific procedure on how to create VEBs in Section 3.3 for both Xilinx and Altera devices. A verification scheme on how to measure the accuracy of the model and the results is illustrated in Section 3.4. The results of embedded various embedded multiplier and FPUs are described in the same section. The discussion and comparison to another mainstream FPGA architecture exploration tool such as VPR is presented in Section 3.5.

3.2 Overview

3.2.1 VEB Model

The basic strategy is to use the logic resources of an existing FPGA to match the expected position, area, and delay of an ASIC implementation of an embedded block (EB). We address the existing FPGA as host FPGA since it offers a template to model arbitrary embedded block on top of it. The model of an ASIC EB, including area and critical path parameters, can be obtained from implementing the circuit or from published ASIC EB information. The matching could be achieved using appropriate vendors' tools such as Xilinx ISE or Altera Quartus II. In order to estimate its perfor-

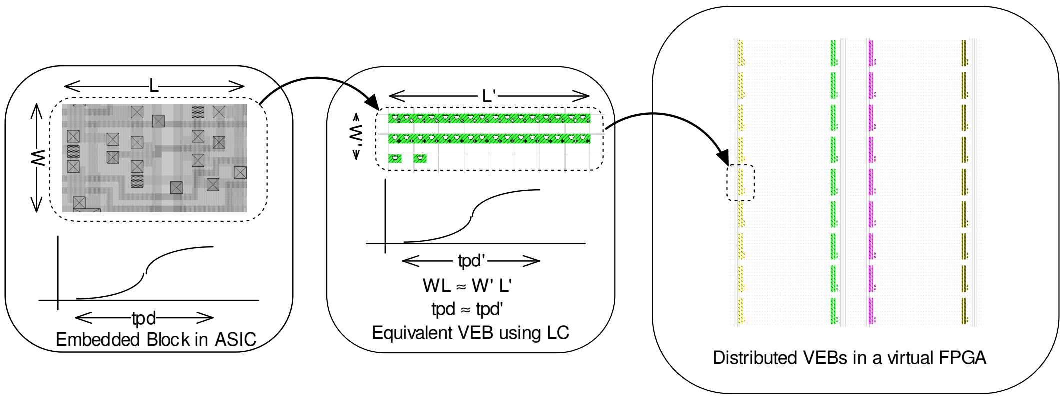


Figure 3.1: Modelling embedded elements in FPGAs using Virtual Embedded Blocks.

mance, the EB is modelled using logic cell resources in VEBs. Our model of an FPGA with VEBs is called a virtual FPGA as illustrated in Figure 3.1.

Area

To employ this methodology, area and delay models for the EB are required. The area model is translated into equivalent logic cell resources in the virtual FPGA. In order to make this translation, an estimate of the area of a logic cell in the FPGA is required, this value including the associated routing resources and configuration bits. All area measures are normalised by dividing the actual area by the square of the feature size, making them independent of feature size. VEB utilisation can then be computed as the normalised area of the EB divided by the normalised area of a logic cell. This value is in units of equivalent logic cells and the mapping encourages thinking about EBs in terms of FPGA resources. In addition, special consideration is given to the interface between the LCs and the VEB to ensure that the corresponding VEBs have sufficient I/O pins to connect to the routing resources. This can be verified by keeping track of the number of inputs and outputs which connect to the global routing resources in a LC. For example, if a logic cell only has 2 output, it is not possible to have a VEB with an area of 4 LCs that requires 9 outputs. For such a case, the area is increased to 5 LCs.

Delay

In order to accurately model the delay, both the logic and the wiring delay of the virtual FPGA must match that of the FPGA. The logic delay can be matched by introducing delays in the VEB which are similar to those of the EB. In the case of very small VEBs, it may not be possible to accurately match the number of I/O pins, area or logic delay and it may result in some inaccuracies. A complex EB might have many paths, each with different delays. In this case, we assume that all delays are equal to the longest one (i.e. the critical path) as it is the most important characteristic of an EB in terms of timing.

In our implementation, area matching is done by creating a dedicated scan-chain using shift registers. A longer scan-chain consumes more LC and therefore the VEB is larger. As the growth of area is linear with the scan-chain, an efficient search algorithm can be used to automatically generate a VEB as described in Section 3.2.2.

There are many options available to match the timing of a VEB. We utilize the fast carry-chains in most FPGAs to generate delays that emulate the critical path in a VEB. This choice has the added advantage that relocation of LCs on the FPGA does not affect the timing of this circuit. Similar to a scan-chain, the growth of delay is linear with the carry-chain, so the same search can be applied to determine the required length automatically.

It should also be noted that the use of the carry-chain and scan-chain allows delay and area to be varied relatively independently. However, modelling wiring delays is more challenging, since the placement of the virtual FPGA must be similar to that of an FPGA with EBs to ensure that their routing is similar. This requires that:

1. The absolute location of VEBs matches the intended locations of real embedded blocks in the FPGA with EBs.
2. The design tools are able to assign instantiations of VEBs in the netlist to physical VEBs while minimising routing delays.

Requirement (1) is addressed by locating VEBs at predefined absolute locations that matches the floorplan of the FPGA with coarse-grained units. To address (2), the assignment of physical VEBs is currently made by two-phase placement strategy which consists of unconstrained placement followed by manual placement. We first assume that the VEB can be placed anywhere on the virtual FPGA so the place and route tools can suggest the most suitable location for each VEB. Once the suggested VEB locations are known, a manual placement is applied to ensure that the placement of each VEB is aligned on dedicated columns while maintaining nearest displacement to the suggested location. We believe this strategy can provide a reasonable placement as the location of each VEB is derived from the suggested placement.

Power

Let P_{fgu} be the power dissipation of fine-grained unit and let P_{veb} be the power dissipation of VEB. The dynamic power dissipation (P_{all}) of a virtual FPGA can be represented by the following equation:

$$P_{all} = P_{fgu} + P_{veb} + P_r \tag{3.1}$$

In the proposed flow, P_{fgu} is estimated using a spreadsheet approach, and P_{veb} is determined using an ASIC power estimation tool. However, neither P_{veb} nor P_{fgu} account for the power dissipation in routing resources between VEB and fine-grained units. We introduce P_r which represents this power which can be obtained by modelling the output loading of the VEB.

In order to employ the proposed power estimation flow, the following assumptions are made and justified as follows:

1. The process technology used in building the VEB should be similar to that of the fine-grained unit. With similar transistor size, the area and delay of the units can be directly compared. Moreover, transistors with similar size have similar capacitance which is a critical factor when estimating power consumption.

2. Constant activity rates are assumed on all the nets in a design. This allows us to

rapidly estimate the power consumption without estimating activity rate using post-place and route simulation.

3. Apart from logic cells, registers and embedded blocks, there are several components which may affect the dynamic power consumption in an FPGA. Such components include I/O cells and clock management units. In this study, we assume all hybrid FPGAs share the same architecture so the dynamic power consumption of these components is the same. Only the dynamic power dissipated in computation cores are considered in the estimation.

Given the area of fine-grained units and the operation frequency, the power consumption of the fine-grained units can be determined by a commercial FPGA power estimation tool as the fine-grained unit has the same architecture as the commercial one. A high level power estimation, which assumes a constant toggle rate of all nets, is employed to obtain the dynamic power dissipation of fine-grained units. Similar to estimating the power consumption of the fine-grained unit, the power consumption of VEB can be obtained from an ASIC power estimation tool, assuming a fixed operation frequency and toggle rate for all the nets.

The dynamic power consumption of the routing resources between the VEB and fine-grained unit can be modelled by setting the appropriate output loading of the VEB. A calibration scheme is proposed to determine the output loading. The dynamic power consumption of an existing embedded block in a commercial FPGA is first measured. An equivalent embedded block is then implemented with standard cell design flow and the dynamic power consumption is assessed assuming no loading at the outputs. Then the output loading of the embedded block is increased until the dynamic power consumption matches the commercial FPGA. This value represents the average loading of the routing resources and is applied to the output of the VEB. The dynamic power consumption of the routing resources can be obtained from the ASIC power estimation tool. The high level estimation of the dynamic power consumption of the hybrid FPGA can be obtained by combining both numbers.

The same approach can be applied to energy consumption or the power-delay product which are often better metrics for FPGA architectures than power consumption.

The major issue concerning the proposed approach is accuracy. There are several contributing factors: (1) assumption of constant switching activity rate and (2) the uncertainty of the power dissipation in routing resources. To address (1), we can estimate the power consumption under different switching activity rates to measure the lower and upper bounds for the power consumption. To address (2), we propose a calibration scheme which adjusts the output loading of the VEB.

The use of VEB to measure power consumption of proposed FPGA architecture is described in chapter 5 in greater detail.

3.2.2 VEB generation tool

A VEB generation tool is developed to automate generation of VEBs. This tool can support different vendors and device families through a device dependent interface. It accepts entity configuration, timing, aspect ratio and area of a desired embedded block as input arguments and produces a synthesisable VEB design that meets these parameters. This is done using an iterative algorithm that matches the required parameters to within a desired error.

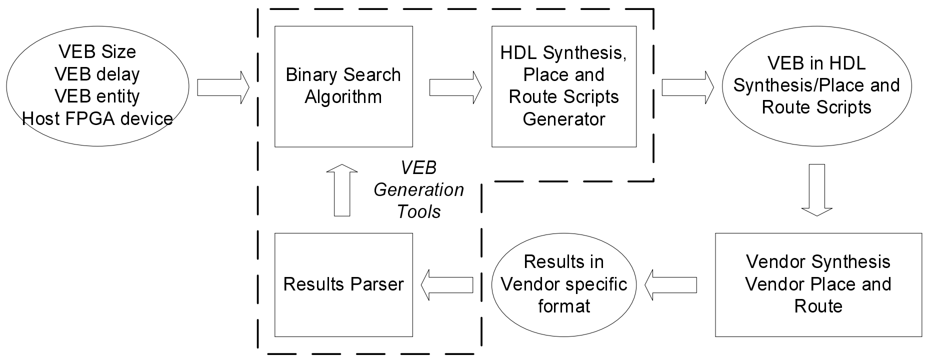

VEB Size
VEB delay
VEB entity
Host FPGA device
Binary Search Algorithm
HDL Synthesis, Place and Route Scripts Generator
VEB in HDL Synthesis/Place and Route Scripts
VEB Generation Tools
Results Parser
Results in Vendor specific format
Vendor Synthesis Vendor Place and Route

Figure 3.2: The block diagram of the VEB design flow

A block diagram of the generation tool is shown in Figure 3.2. The modules include a HDL implementation script generator, search and implementation result parser. Users can specify parameters for the VEB design flow including the expected area and delay, acceptable error, aspect ratio as well as the host FPGA family. Based on these parameters, the script generator produces the corresponding HDL and implementation script for a specific vendor design suite. The implementation script is then executed, serving to run synthesis, place and route and timing analysis. Upon completion, the result parser obtains the VEB area and speed from the tool's result files and passes them to the search algorithm. The search algorithm iteratively increases or decreases the size of the carry-chains and scan-chains until the desired area and delay are obtained. In order to support different vendor's design flows, the script generator and results parser are customised for different vendors.

The search algorithm is based on binary search where the upper values of the size of the chains are required. The algorithm first determines an upper value for the chain size by doubling its size until the implementation exceeds the desired delay or area. The search algorithm uses an initial carry-chain and scan-chain of 10 and 1 respectively. A binary search algorithm is then applied to obtain the chain size that best matches the desired delay. As the size of carry-chain affects the area of the VEB and the size of scan-chain does not affect the delay, both carry-chain and scan-chain can meet the user requirement by matching the carry-chain first followed by scan-chain.

There may be some cases that both area and delay cannot be matched using logic cells. For examples, if the required delay is too small that even combinatorial delay of a LUT is larger than the required one, such VEB cannot be made using this methodology. This case seldom occurs since an embedded block usually has more complex function than a LUT and therefore the delay should be larger. If the area is too small while the required number of I/O pins is too large, VEB methodology may not match the area since each logic cell has limited number of I/O only. We can apply post-adjustment to obtain the effective area consumption as illustrated in Section 3.3 in this case.

3.3 Vendor Specific Design Flow

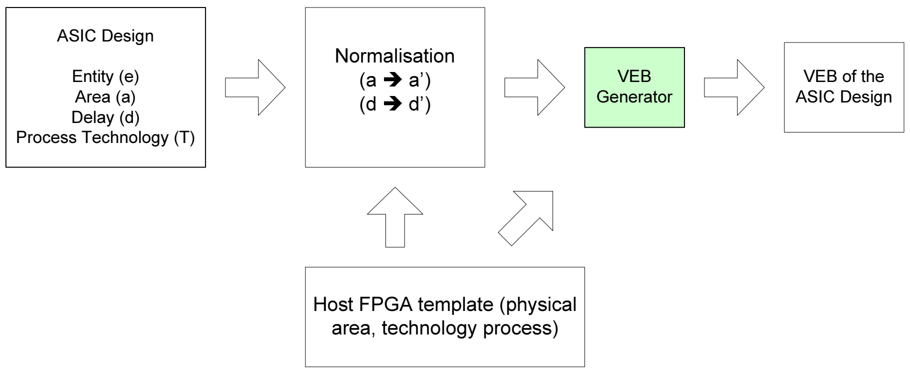


Figure 3.3: Phase 1 – VEB creation flow.

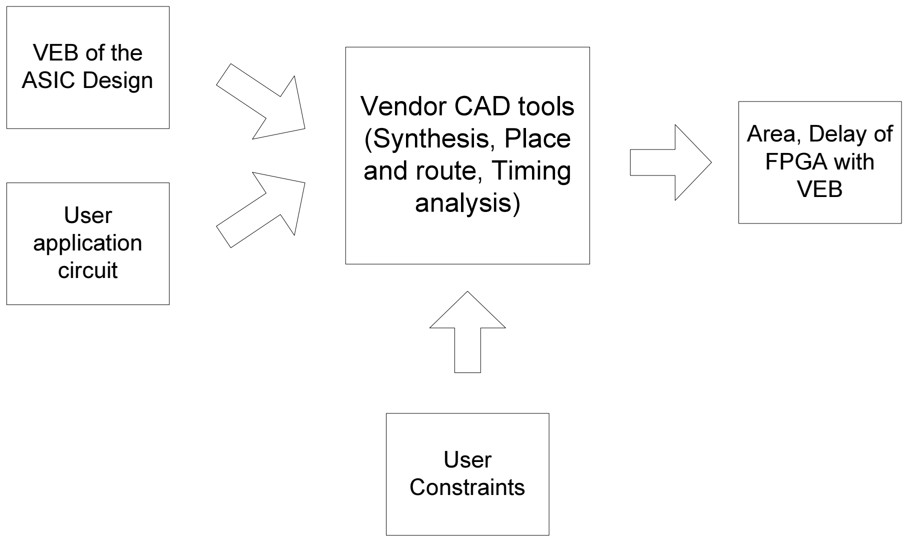

VEB of the ASIC Design
User application circuit
Vendor CAD tools (Synthesis, Place and route, Timing analysis)
Area, Delay of FPGA with VEB
User Constraints

Figure 3.4: Phase 2 – VEB integration flow.

There are two phases in the VEB design flow. The first phase is to create a VEB block based on the ASIC design. The VEB generation tool discussed in 3.2.2 has discussed how a VEB blcok is generated. Figure 3.3 captures the design flow of this phase. The second phase, as illustrated in Figure 3.4, is to apply VEB to model an FPGA with arbitrary embedded blocks. While we have VEB generation tools to produce VEB automatically, users are required to manually integrate VEB into host FPGA since the existing vendor tools are not supposed to support it.

As an example to demonstrate the use of VEB design flow, this section illustrates how a VEB can be used to model a real embedded multiplier block, using Xilinx

Virtex II and Altera Stratix devices as case studies. All of the results described in this work are obtained using the Synplicity Synplify Premier 9.0 synthesis tool and the Xilinx ISE 9.2 or Altera Quartus 6.0 design tools. The Synplicity synthesis tool is used to demonstrate our methodology is compatible with current FPGA development practices, and allows for advanced optimisation such as retiming and pipelining to be applied.

3.3.1 VEB Parameters Estimation

This section covers the first phase of VEB flow. We explain how to obtain the parameters required to create a VEB. We begin with logic cell area estimation for both devices. The physical die area of the Virtex device is reported in [Saab 05, Yui 02] and in this study, the value reported in [Yui 02] is used. We assume 70% of the total die area is used for logic cells and their associated routing while the other area is I/O pads, block memories, block multipliers, etc. As the area of a Stratix device is undocumented, we assume the area of a LC in the Stratix device is the same as the Virtex II one. Table 3.3 shows a number of logic cell area estimates. The area estimates for the embedded blocks studied are given in Section 3.4.

To model the area of the embedded multipliers on each device, we assume that they occupy a total of 2% of the die area which, in turn, is reported to be 93 mm^2 [Yui 02]. This translates to a normalised EM area of approximately 2,751,000, which is 6 LCs. Assuming each LC has 4 input and 2 outputs, one embedded multiplier is equivalent to 18 LCs such that all the connection from embedded multiplier can be connected. We also assume the multiplier in Stratix device has the same size as the one in Virtex II device.

The combinatorial logic delay of an adder carry-chain in Virtex II device can be modelled by $t_{pd} = T_{opcy} + \frac{N-4}{2} \times T_{byp} + T_{ciny}$, where N is the length of the adder carry-chain, T_{opcy} is the combinatorial delay from the input to the COUT output, T_{byp} is the com-

Delay name	Description	delay (ns)
T_{opcy}	F to COUT	0.665
T_{byp}	CIN to COUT	0.084
T_{ciny}	CIN to Y via XOR	0.940
T_{mult}	Embedded multiplier	4.660
T_{multck}	Registered embedded multiplier	3.000
T_{dyck}	Register setup and hold time	0.293

Table 3.1: Delay parameters for Virtex II-6 devices.

binatorial delay from CIN to COUT, and T_{ciny} is the combinatorial delay from CIN to the Y output via an XOR gate. If the output is latched, the setup and hold time of a register (T_{dyck}) should be added to this value. Typical values for these parameters in the Virtex II adder carry-chain and multiplier block are extracted from vendor's timing analysis tool and given in Table 3.1.

As an example, to model a registered multiplier block with delay of 3 ns, $N = 30$ gives a logic delay (including setup and hold time) of 2.99 ns. In the Xilinx device, the carry-chains run along the columns. One issue to note is that the carry-chain only runs in a single direction in the device and breaking the carry-chain introduces a long wiring delay. In our current approach, a certain amount of trial-and-error is required to achieve a given delay.

Although there is no official documentation suggesting the timing model of the carry-chain in Stratix device, the vendor has disclosed typical values for some common functions and they are listed in Table 3.2. One advantage of the VEB model is that even if no clear documentation available from the device vendor, it is still possible to accurately model the time and delay by adjusting the size of carry-chain and scan-chain.

An 18-by-18 multiplier has 36 output pins and each LC has 2 output pins. This means that we would need at least 18 LCs for our VEB model. Although the difference in area does not affect the logic delay in a VEB model, the routing delays are slightly longer when the wires are routed through more LCs. More LCs in the actual VEB model

Delay name	Description	delay (ns)
T_{lut}	LUT delay	0.469
T_{su}	Setup time	0.010
T_h	Hold time	0.100
T_{add16}	Registered 16 bit counter	2.369
T_{add64}	Registered 64 bit counter	3.441
T_{mult}	Registered Embedded Multiplier (18x18)	4.212

Table 3.2: Delay parameters for Stratix-6 devices.

Device	LCs/LB	Area/LB	Feature Size	Normalised LC area
	L	A (μm^2)	f (μm)	($N = A/Lf^2$)
Virtex II 3000 [Saab 05]	8	$71,429 \times 0.7$	0.15	277,779
Virtex II 1000 [Yui 02]	8	$72,782 \times 0.7$	0.15	283,045
Stratix	-	-	0.13	283,045

Table 3.3: Estimates of logic cell area including configuration bit, buffer and interconnect overheads. There is no public information about the die area of Stratix device so we assume that it has the same normalised LC area as the one in Virtex II 1000 device.

also means that the resulting area is different from the real embedded multiplier. We apply post-adjustment to compensate for this effect by using the larger area model for timing analysis and then refining the area estimate by deducting the difference between VEB area model and the actual area.

For example, if we implement a circuit which instantiates 4 virtual multipliers on the virtual FPGA and the reported area is 144 LC. It means the effective area of the circuit is 72 LC after post-adjustment. It is because each multiplier consumed 18 LCs only.

To obtain the timing of a real embedded multiplier, it is implemented and the register-to-register delays reported by the vendor's timing analysis tools extracted. The delays are 4.29 ns and 4.21 ns for Virtex II and Stratix devices respectively.

3.3.2 Integration into Xilinx Tools

This section covers the second phase of the VEB design flow, while focusing on the Xilinx tool chain. To integrate our design flow into the Xilinx tools, the VEB is implemented using relationally placed macros (RPMs) which allow placed sub-circuits to be defined [Xili 04]. During HDL generation of the VEB, a vendor-specified user constraint file (UCF) is created to define the placement of the VEBs, an example being shown below.

Xilinx user constraint file.

```
AREA_GROUP "AG_mul18" RANGE = SLICE_X10Y17:SLICE_X13Y4 ;
INST "/" AREA_GROUP = "AG_mul18" ;
```

Several options need to be specified during synthesis to avoid unwanted optimisation by the synthesis tools. In order to instruct the synthesis tools inferring registers in the logic cell rather than those in memory, the `syn_srlstyle` attribute must be specified as `register`. Automatic I/O insertion, pipelining and retiming must also be disabled and, during the place and route process, the `trim unconnected logic` option must also be disabled. The VEB generation tool encapsulates these all of these steps as discussed in Section 3.2.2.

After the VEB has been placed and routed, another constraint file containing the relative placement information for each LC in the VEB is generated. The placement information and the netlist for the VEB are compiled to create a RPM.

To employ the VEB in an application, the HDL description is modified to instantiate the corresponding VEB block. Since the VEB is considered as a black box during synthesis, timing information must also be specified to allow the synthesis tool to take timing of the block into account during optimisation. This makes optimisations such as retiming possible. During place and route of the benchmark circuit, the VEBs are placed in regular locations on the FPGA, modelling the expected locations of the EBs. This is achieved using placement constraints. The design is then placed and routed in the usual fashion and timing analysis applied to produce area and timing results.

3.3.3 Integration into Altera Tools

This section covers the second phase of the VEB design flow, while focusing on the Altera tool chain. We illustrate our approach using the Altera tools in a similar manner. The LogicLock feature is employed to group logic cells. For synthesis, we use Synplicity Synplify and the same options as for Xilinx are used except the target device.

During place and route of the VEB, all LogicLock settings are specified in a Tcl script. One key precaution is that once we place the LogicLock region in the design, we are not allowed to optimise the circuit as the placer may potentially modify the LogicLock region by trimming unnecessary logic. Therefore options such as register duplication, register retiming, packing register to I/O pads have to be disabled for placement.

The following settings are used:

Altera Tcl script for setting logic lock.

```
# logic lock assignment
initialize_logiclock
set_logiclock -region veb_fpu_lock -floating true -height 16 -width 7
set_logiclock_contents -region veb_fpu_lock -to *
# compile
execute_flow -compile
# export desgin
logiclock_back_annotate -region veb_fpu_lock -lock
logiclock_export -file veb_fpu_lock.qsf
```

Once the VEB is created as a LogicLock component, the VEB can be imported in a design and placed and routed in the normal fashion. Timing and area results can be obtained via the timing analyser.

3.4 Results

3.4.1 Verification of the VEB Design Flow

In order to verify the results obtained using our methodology, we develop VEBs for embedded 18×18 multiplier (EM) on the Virtex II and the Stratix devices. Since such embedded multipliers are found in those FPGA devices, it is possible to compare the routing and logic delays of benchmark circuits from the VEB design flow with those given by the actual EMs.

The benchmark circuits are implemented both using the EMs and the VEB multiplier. Table 3.4 and 3.5 summarises the resource utilisation and critical path delay for both implementations on Xilinx and Altera devices respectively. First considering the critical path delay, the difference between the real FPGA and the virtual one is less than 12%. For most of the circuits, the critical path passes through the multiplier. In those cases where it is not, the longest delay through the multiplier is very close to the critical path of the circuit.

The table also shows that retiming can be used in the VEB methodology. Retiming the bgm circuit can achieve additional speedup of 1.27 on the Virtex II device and 1.32 on the Stratix device. The bgm benchmark shows a lower error than the other circuits on both devices because the critical path does not involve the EMs but rather is in the delay of the accumulator plus rounding logic after the summation.

The relative error for the Stratix device is larger than Virtex II device and is largest (11.94%) when a large number of multipliers are instantiated. We believe this is because of an inaccurate estimate of the size of the embedded multiplier for Stratix. The inaccurate estimation in the physical area of the embedded block may result in longer or shorter routing delay. The difference in error between Virtex II device and Stratix device suggests that we have a better estimation of the size of embedded multiplier in Virtex II device than the Stratix one. As a result, when many multiplier blocks are instantiated, the routing delay wire may be different to the real embedded multiplier.

Benchmark	Size (logic cell)	# of EMs	EM delay (ns)	VEB delay (ns)	Difference (ns)	Difference (%)
bfly	1,240	4	4.97	5.25	0.28	5.63%
dscg	350	4	4.53	4.79	0.26	5.74%
fir4	384	4	4.53	4.79	0.26	5.74%
mm3	1,366	3	4.73	4.79	0.06	1.27%
ode	404	2	4.54	4.79	0.25	5.51%
mul34	352	4	8.77	8.38	0.39	4.45%
mul68	1,728	16	9.75	9.13	0.62	6.36%
mul136	6,512	64	11.98	11.67	0.31	2.59%
bgm	6,684	46	11.63	11.29	0.34	2.92%
bgm*	6,954	46	9.11	9.07	0.04	0.44%

Table 3.4: Summary of resource utilisation and critical path delay for embedded multiplier (MULT18X18S) and VEB implementations on an Xilinx XC2V6000-FF1152-6 device. An asterisk (*) indicates that retiming is enabled during synthesis.

Benchmark	Size (logic cell)	# of EMs	EM delay (ns)	VEB delay (ns)	Difference (ns)	Difference (%)
bfly	942	8	8.14	8.91	0.77	9.46%
dscg	312	8	4.21	4.51	0.30	7.13%
fir4	354	8	4.21	4.53	0.32	7.60%
mm3	644	6	8.76	9.45	0.69	7.88%
ode	303	4	5.05	5.44	0.39	7.72%
mul34	199	8	8.93	8.26	0.67	7.50%
mul68	1,074	16	9.40	10.28	0.88	9.36%
mul136	3,667	128	11.81	10.40	1.41	11.94%
bgm	4,070	92	10.96	11.12	0.16	1.46%
bgm*	5,822	92	8.29	8.51	0.22	2.65%

Table 3.5: Summary of resource utilisation and critical path delay for 18x18 embedded multiplier in DSP block and VEB implementations on an Altera EP1S80F1508C6 device. An asterisk (*) indicates that retiming is enabled during synthesis.

Table 3.6 shows the breakdown of the critical path into logic and routing delays for the EM implementation. The corresponding path in the VEB implementation is identified and shown in the same table. The sum of the logic and routing delay for the EM should be equal to the corresponding value in Table 3.4, but due to clock skew it is slightly different. The logic delays between the two implementations are very similar. The routing delays differ greatly because the EM and VEB implementations often have different placement, but since the nets are not on the critical path in the VEB implementation, they do not affect the maximum operating frequency of the circuit.

Benchmark	EM delay		Equivalent VEB path delay		Difference			
	logic (ns)	routing (ns)	logic (ns)	routing (ns)	logic (ns)	logic (%)	routing (ns)	routing (%)
dscg	3.449	1.150	3.445	1.536	0.004	0.116%	0.386	25%
fir4	3.449	1.167	3.445	0.815	0.004	0.116%	0.352	43%
ode	3.449	0.911	3.445	0.672	0.004	0.116%	0.239	36%
mm3	3.449	1.366	3.445	1.067	0.004	0.116%	0.299	28%
bfly	3.449	2.062	3.445	1.411	0.004	0.116%	0.651	46%
mul34	8.818	2.345	8.990	2.202	0.172	1.913%	0.143	6%
mul68	8.682	3.687	8.990	4.960	0.308	3.426%	1.273	26%
mul136	8.682	5.950	8.990	4.258	0.308	3.426%	1.692	40%
bgm	10.119	3.901	10.019	1.916	0.1	0.998%	1.985	104%
bgm*	8.439	3.155	7.631	3.971	n/a	n/a	n/a	n/a

Table 3.6: Breakdown of critical path delay for embedded multiplier and VEB implementations on a XC2V6000 device. bgm* indicates that retiming is enabled during synthesis.

It would be possible to also match the routing delays by locking placement of all of the LCs in the design rather than just the VEB, if closer matching of the routing delays is desired.

3.4.2 Embedded Floating Point Unit

This section demonstrates the modelling of an embedded floating point unit (FPU) on an FPGA. The area and delay model of a VEB FPU is made based on area and speed estimates of the Blue Gene ASIC [Brig 05, Wait 05]. This is a full-functional FPU fabricated in a similar technology ($0.13\mu m$) to the Xilinx Virtex II. It operates at a clock frequency of 700 MHz, with an area estimated to be 4.26 mm^2 [Brig 05] which translates to 905 LCs on both FPGA devices. The area estimate is very conservative, since we expect an FPU embedded on an FPGA would be less complex than the Blue Gene ASIC.

As the Blue Gene 700 MHz FPU design has a much smaller logic delay than the routing delay of the FPGA, a better implementation can be obtained by reducing both its latency and clock frequency by a factor of 5. Thus the VEB FPU considered has a clock frequency of 140 MHz with (7.14ns) a one cycle latency. This essentially trades

	FPGA				VEB				Improvement	
	EMs	through-put (# of cycle)	size (LC)	delay (ns)	FPUs	through-put (# of cycle)	size (LC)	delay (ns)	Area	Speedup
bfly	36	1	27,576	24.35	8	1	7,328 + 4,490	7.62	2.33	3.20
dscg	36	1	19,408	22.84	6	1	5,496 + 1,192	7.34	2.90	3.11
fir4	36	1	22,966	22.35	7	1	6,412 + 1,442	7.32	2.92	3.05
mm3	27	225	17,714	23.35	5	45	4,580 + 3,600	7.58	2.17	15.40
ode	18	20	15,886	22.19	5	4	4,580 + 1,224	7.62	2.74	14.60
								Geometric Mean:	2.59	5.84

Table 3.7: FPGA implementation results for floating point benchmark applications on a Xilinx XC2V6000-6-FF1152 device. The VEB size is given as the FPU area (in equivalent LC resources) plus the LC resources needed to implement the rest of the circuit. The second column (EMs) indicates number of MULT18X18S instantiated.

	FPGA				VEB				Improvement	
	EMs	through-put (# of cycle)	size (LC)	delay (ns)	FPUs	through-put (# of cycle)	size (LC)	delay (ns)	Area	Speedup
bfly	96	1	22,498	25.47	8	1	7,328 + 2,638	9.60	3.07	2.65
dscg	96	1	16,959	26.49	6	1	5,496 + 880	7.89	3.09	3.36
fir4	96	1	17,977	28.73	7	1	6,412 + 983	7.61	2.80	3.77
mm3	78	225	13,812	26.22	5	45	4,580 + 1,782	12.07	3.02	10.87
ode	52	20	11,902	23.18	5	4	4,580 + 794	7.97	2.60	14.54
								Geometric Mean:	2.91	5.56

Table 3.8: FPGA implementation results for floating point benchmark applications on an Altera EP1S80F1508C6 device. The VEB size is given as the FPU area (in equivalent LC resources) plus the LC resources needed to implement the rest of the circuit. The second column (EMs) indicates number of 9x9 multipliers instantiated.

off clock frequency for reduced latency.

The performance of the Virtex II FPGA and Stratix FPGA is compared to a corresponding virtual FPGA with embedded FPUs using the floating point benchmarks. A summary of the results is given in Table 3.7 and Table 3.8. Since VEB FPU has reduced latency, the speedup on certain benchmarks which requires data dependency is not only because of the increase in clock rate but also the improvement in throughput. For examples, the throughput of benchmark circuits *mm3* and *ode* has improved by 5 times when embedded FPUs are introduced so the overall speedup is 10.87 times and 14.54 times respectively. It is more than the relative difference of the delay.

The results show that augmenting the FPGA with embedded FPUs leads to an average improvement in area and speedup by factors of 2.59 and 5.84 respectively on the Virtex II device. Similar improvement is obtained in Stratix device where the area is reduced by 2.91 and speedup is 5.56 on average. The overall improvements in area and speedup are 2.75 and 5.7 respectively. In contrast, a recent investigation of embedding double-precision FPUs in FPGAs based on VPR with a different set of benchmarks results in estimates of average area savings of 2.21 times and 1.33 times in clock rate over existing architectures [Beau 08]. We attribute the differences to: different benchmarks being used, CAD tools, FPU delay and latency, FPGA model and the use of retiming optimisations during synthesis.

3.4.3 Exploration of Technology Trends

The VEB design flow can be used to (1) obtain a single performance estimate for introducing embedded blocks, (2) analyse performance/area trade-offs, and (3) determine the EM speed required to meet a given system performance.

In one experiment, we use VEB to implement virtual embedded multiplier with different delays and observe how the delay of multiplier affects the overall performance of the bgm circuit. We assume the virtual embedded multipliers have constant area to minimise the routing effect. We generate five virtual embedded multipliers. The baseline virtual embedded multiplier has the same delay as the real one (4.66ns). The delay of subsequence virtual embedded multipliers are reduced gradually until the limit of the host FPGA itself.

The virtual embedded multipliers are than used to implement the bgm circuit using the vendor design flow and results are collected. Retiming is used in such experiments since, for pipelined designs, improving the performance of one pipeline stage can create slack in another stage, moving the bottleneck to a different stage of the pipeline. A similar situation occurs in multi-cycle designs.

The results are shown in Figure 3.5. An EM performance of 1 is the same as the performance of the Xilinx EM, and a normalised system performance of 1 corresponds to the execution time of the bgm benchmark. From this figure, one can determine the maximum speedup that can be achieved in this application via faster EMs to be approximately 1.4, which can be obtained by speeding up the block multiplier in Virtex II devices by 2.2 times.

As an example of estimating system performance of a design fabricated in a different process technology, consider a 16×16 bit combinational multiplier operating at 1 GHz with an area of 0.474 mm^2 at 1.3 V in 90 nm technology [Hsu 06]. Assuming velocity saturated general scaling of transistor lengths from 90 nm to 0.13 μm ($1/S = 0.13/0.09$), the delay would scale by $1/S$, i.e. from 1 ns to 1.44 ns [Raba 02]. The scaled area of the implementation would be 132 LCs. Such an implementation is thus 1.44 times faster but uses 3.6 times more area than the Xilinx EM, and improves bgm performance by 15%.

Experiments are conducted to assess the impact of embedded block performance on system performance. Specifically, we study the speedup of the bfly benchmark as a function of the FPU performance (Figure 3.6). We generated 7 virtual embedded FPUs with various delay. And each virtual embedded FPU are used to implement the bfly circuit.

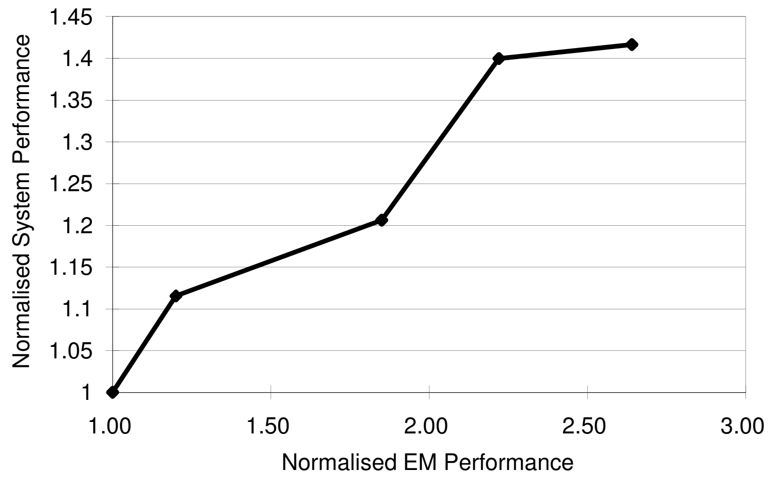


Figure 3.5: Performance of fixed-point bgm benchmark with different VEBs, with retiming.

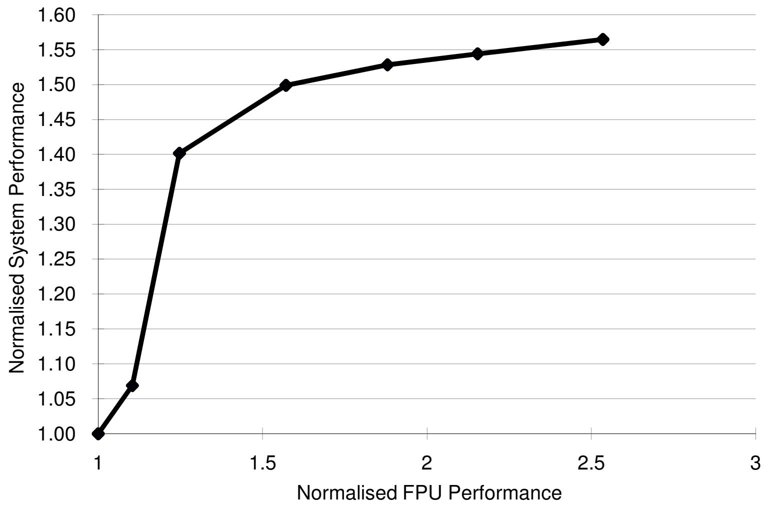


Figure 3.6: Performance of floating-point bfly benchmark with different FPU delays, with retiming.

We assume the bfly circuit always operates in fully-pipelined manner even if each virtual embedded FPU has different pipeline stages. It can be seen that a modest improvement in FPU speed can lead to a large improvement in the bfly benchmark: for instance improving the FPU performance by 30% improves bfly performance by 40%. Beyond a factor of 1.4, the speedup of the benchmark increases rather more slowly. This type of information can be used to determine the best option for ASIC implementations of EBs in which the synthesis tools offer a wide range of possible area/delay trade-offs.

3.5 Discussion

The methodology proposed in this work shares some similarities with the VPR, a popular CAD tool for architectural exploration of island-style FPGAs in academic research. The key differences among the methodologies, including the usage, the design entry, the architecture and the flexibility, are discussed below. Table 3.9 highlights some of their differences.

The purpose of VPR tool is to provide a flexible CAD infrastructure to evaluate the performance, in terms of area, power and delay, of different FPGA architectures. The tool allows different customisations of FPGA architectures. For instance, users can configure high-level parameters such as the size of a LUT, the number of LUT inputs, the number of LUTs in a cluster, the number of inputs to a cluster, the number of tracks in a routing channel, the switch box topology etc. Moreover, users can specify low-level parameters such as RC values of pass transistors, buffers, LUT and I/O pads. These low-level parameters are usually determined by HSPICE simulation or predictive model [Zhao 06] and the parameters are usually architecture and technology dependent.

The purpose of the VEB methodology is to model arbitrary embedded elements on a commercial FPGA. Therefore the flexibility of this model is limited by the nature of the host FPGA. Users cannot modify the architectural parameters in the same way as for VPR. However, users can create equivalent embedded elements which have similar area and delay, then place and route them using vendor CAD tools and assume that the embedded elements exist in the device. Thus a VEB model encapsulates most of the properties of the underlying architecture.

One of the main advantages of the VEB methodology is that it can provide a rapid evaluation for arbitrary embedded blocks. Thus it can be used to rapidly identify which embedded block is particularly useful. Further investigation can later be applied to obtain more accurate results if warranted.

To obtain meaningful results from VPR, accurate area and delay measurements of the target fine-grained architecture are necessary. Users are required to provide their own specific transistor-level circuits to generate their own area, delay, resistance and capacitance parameters relating to an architecture. This can be a difficult and time consuming process. A more recent version of VPR [Luu 09] provides architecture templates to alleviate the process. However, users who want to modify the fine-grained architecture are still required to supply their own physical level parameters.

The limited access to physical level parameters of commercial FPGA devices implies that direct comparison to commercial FPGA is very unlikely unless the vendors disclose such proprietary technical data to the public. VEB model, however, can indirectly access such information through the tools offered by the vendor and thus meaningful comparison can be made.

In terms of the design flow, VPR tools only allow BLIF format input which is usually synthesised by SIS [Sent 92]. Odin [Jami 05] can convert Verilog description to BLIF format. However, the tool supports RTL descriptions only and circuits described as behavioural one has to be rewritten. Both SIS and Odin do not provide common optimisation features such as retiming and physical synthesis, which considers the physical placement of the logic cells and the routing resource used during synthesis. These features usually present in commercial CAD tools but cannot be found in the academic one.

More importantly, VPR and commercial FPGA CAD tools use different synthesis, place and route algorithms, making comparisons more difficult. The VEB methodology, on the other hand, is seamlessly integrated into the commercial FPGA design flow and entry. It allows schematic as well as HDL design entry and follows the same synthesis flow, placement and routing and timing analysis as the host FPGA.

While the VPR tool is sufficiently flexible to model island-style FPGA architectures, several issues prohibit it to model a commercial FPGA. Despite the timing model and the design flow as mentioned above, VPR models the Xilinx XC4000X [Xili 99] fine-grained fabric which is no longer appeared in commodity commercial FPGA device. Even though some modified versions of VPR support carry-chains, block multipliers and block memories, we are not aware of any published results that quantify the difference between a VPR model and a commercial FPGA.

However, there are some limitations in the VEB methodology when compared to VPR. Beside the architecture issues mentioned above, there are certain embedded elements that cannot be implemented as a VEB. For instance, it is not possible to accurately model small embedded elements with large delay due to the difficulty of introducing large delays with limited logic cells. Similarly, when the I/O to area ratio is too high, modelling using VEBs becomes inaccurate. In such cases, it is still possible to measure the timing by allowing a larger VEB. It should be noted that this timing may not be accurate as longer wires are needed to route through a VEB. Our experimental results show that this error is less than 12% for the circuits tested. Table 3.9 summarises the differences between VEB and VPR-based models.

3.6 Summary

We propose a methodology for estimating the effects of introducing embedded blocks to commercial FPGA devices. It is vendor independent and offers rapid evaluation of arbitrary embedded blocks. The proposed flow can address the issues raised in Section 3.1. Since the flow emphasises the reuse of existing CAD tools from FPGA vendor in modelling, it can capture the area, delay and power when a particular circuits implemented on a virtual FPGA with arbitrary embedding blocks (Issue 1, 3). In addition, by using vendor tools as media in modelling, we can assume the same fine-grained architecture as existing FPGA device and recover the associated physical parameters indirectly, allowing the results obtained from the VEB flow are comparable to existing FPGA device (Issue 2, 4). Using the VEB flow, we are able to perform parameter sweep experiments as demonstrated in Section 3.4.3 (Issue 5).

	VEB	VPR [Betz 97, Luu 09]
Purpose	Modelling existing island-style FPGA with arbitrary heterogeneous blocks	Modelling island-style FPGA with arbitrary fine-grained fabric, routing architecture and heterogeneous block
Design Entry	VHDL / Verilog	BLIF / Verilog
Fine-grained Architecture	Same as existing FPGA	User-defined or based on XC4000X FPGA device
Coarse-grained Architecture	User-defined	User-defined
Timing Model (fine-grained fabric)	Information provided by vendor tools	User-defined based on their own HSPICE model or Predictive Technology Model [Zhao 06]
Area Model (fine-grained fabric)	Derived from die area information released by vendors	User-defined based on their ASIC model or Predictive Technology Model
Timing Model (coarse-grained fabric)	User-defined based on their ASIC model or published IP information	User-defined based on their ASIC model or published IP information
Area Model (coarse-grained fabric)	User-defined based on their ASIC model or published IP information	User-defined based on their ASIC model or published IP information
Synthesis Tools	Vendor supplied synthesis tools	SIS [Sent 92], Odin [Jami 05] or manual synthesis
Comparison	Existing FPGA	Another FPGA model from VPR
Benchmarks	Domain specific applications / kernels	MCNC benchmarks [Yang 91]

Table 3.9: Comparison of VEB and VPR.

The methodology is evaluated by modelling block multipliers in Xilinx Virtex II devices and in Altera Stratix devices and we find that prediction of critical paths to approximately 12% accuracy can be achieved. The methodology is then applied to predict the impact of embedded floating point units, showing a possible reduction in area of 2.75 times and speedup of 5.7 times.

Chapter 4

Synthesisable Datapath FPGA Fabric

4.1 Introduction

In previous chapter, we have assessed the performance of an FPGA when a whole hardcore FPU is embedded into it. However, in most applications, it is very likely that only part of the FPU is involved in the computation. For instance, additions and multiplications contribute most of the computation in an application while division and square root constitute the minority of an application.

Decomposing the FPU to smaller floating point operators and selecting suitable operators to embed into an FPGA are one of the possible solutions when designing an FPGA device dedicated to floating point computation. However, several issues arise when we consider the interconnection among those components. Although the interconnection can be made using existing fine-grained routing resources in the FPGA, one may consider including dedicated bus routing architecture to connect the components together.

Apart from interconnection optimisation, certain operations other than floating point arithmetic can be optimised. Because the nature of floating point applications, bus-based logic contributes much more computation time than the random logic. For instance, most floating point applications involve the comparison operation and it is essentially a fixed point subtraction and thus is bus-based logic. In addition, multiplexing or moving floating point data is also bus-based logic.

Bus-based logic, while appears in floating point application, can also be found in debugging circuit in System-on-Chip (SoC) system. A key part of this embedded debug infrastructure is the programmable logic fabric. Although it would be possible to create a fabric based on commercial stand-alone FPGA architectures, this may not be desirable for three reasons. First, commercial architectures are optimised for large applications. Since the nature of debugging test circuits tends to be small, it is likely that such architectures provide far more routing resources than are required, leading to increased area overhead. Area overhead is especially important in our applications since the embedded debug fabric is pure overhead and it is not used when the device is finally operational.

Second, commercial FPGA architectures are optimised to work for a wide variety of circuits in different applications. Since much debugging is performed by monitoring buses, we would expect our applications to be primarily datapath-oriented. In addition, because the embedded fabric is a fixed part of an integrated circuit, the context in which it is used (the buses that are monitored, for instance) does not change over time. As we will show in Section 4.6, we can significantly reduce the area required by our architecture by taking advantage of this.

Third, commercial FPGA fabrics may not work well because they are not easily synthesisable by common commercial synthesis tools. Most System-on-Chip (SoC) designs are implemented by synthesising hardware description language (HDL) specifications into standard macrocells. The use of embedded cores are much more palatable to SoC designers if their integration can be made as seamless as possible [Wilt 05]. We discuss the concept of synthesisable fabrics in Section 4.2.

Other embedded fabrics have been reported. Both datapath fabrics [Cher 96, Hauc 04, Leij 03, Ye 03, Ye 06] and coarse-grained architectures [Mars 99, Cron 99, Gold 00, Sing 00] may provide better density than commercial FPGAs, but still suffer in that they are not easily synthesisable. Also like commercial FPGAs, these architectures have been optimised for large stand-alone applications. Synthesisable programmable logic fabrics have been described in [Wilt 05, Yan 06]. These architectures are not datapath-oriented, and hence suffer from a significant density overhead.

This chapter describes a novel reconfigurable architecture for an embedded FPGA fabric that has been optimised for both bus-based or datapath applications such as debug circuits and floating point circuits. Such an architecture can also be employed to implement other arithmetic-intensive circuits. The unique features of the proposed reconfigurable architecture include:

1. Bus-based routing and logic which clusters the fine-grained logic and reduce the chip area.
2. Parameterised design which allows designer to further optimise the architecture based on domain-specific applications.
3. Module design which allows designer to embed or remove individual heterogeneous blocks

We compare this architecture to both a fine-grained synthesisable architecture, and an ASIC implementation of the debug circuitry. Unlike the fine-grained architecture, our fabric contains support for word-level operations and routing, and contains embedded multipliers. Unlike the ASIC implementation, our fabric is flexible enough to implement a wide variety of embedded applications. We show that the new architecture (including embedded multipliers) has a density similar to that of a standard full-custom fine-grained FPGA (without embedded multipliers).

This chapter is organised as follows. Section 4.2 describes the environment in which our embedded core will be used, and describes the requirements of our architecture. The architecture itself is then described in Section 4.3. Section 4.4 then gives an example of how an application can be mapped to our architecture. Section 4.5 reports the efficiency of our architecture as a function of various architectural parameters.Section 4.6 compares our architecture to a previous synthesisable programmable logic core, as well as to an ASIC implementation. Area, delay are power consumptions are reported while a proof-of-concept layout is provided. Section 4.7 compares our approach to that taken in stand-alone datapath-oriented FPGAs and coarse-grained architectures. Finally, Section 4.8 presents concluding remarks.

4.2 Overview and Architectural Requirements

A programmable logic fabric can either be *hard* or *soft*. An ASIC designer using a hard fabric would obtain a transistor level layout and embed it directly into the integrated circuit. These hard fabrics could either be based on commercial FPGA designs, or generated automatically using a layout or architecture generator [Pada 03, Comp 07, Holl 07].

One challenge with this approach is that design tools that allow seamless integration of fixed and programmable logic are still not mature. Timing analysis, power distribution, and verification are difficult when the function to be implemented in the core is not known.

An alternative technique has been recently described which addresses this concern by shifting the burden from the ASIC designer to mature standard macrocell synthesis tools [Wilt 05, Yan 06]. In this technique, an ASIC designer would obtain a synthesisable version of their programmable logic fabric (a *soft* core) written in a hardware description language, and would synthesise it along with the rest of the ASIC. The primary advantage of this technique is that the task of integrating such cores is far easier than the task of integrating hard cores. The synthesis tools can be the same ones that are used to synthesise the fixed (ASIC) portions of the chip. No modifications to the tools are required, and the flow follows a standard integrated circuit design flow that designers are familiar with.

For fabric to be synthesisable in this way, it must not contain combinational loops. Standard synthesis tools, timing analysis tools, and power estimation tools are designed to optimise circuits without combinational loops more efficiently. Although circuits with such loops can be synthesised, this usually requires the designer to manually resolve the loops by identifying some false paths. This requires considerably more understanding about the internals of the core than a typical ASIC designer would have. Note that a standard unconfigured FPGA contains many combinational loops. A designer will rarely configure the FPGA to implement combinational loops, but before configuration, such loops exist. Thus, the first requirement of our architecture is that it does not contain any combinational loops.

The second requirement of our architecture is that it is as small as possible. The area devoted to on-chip debug is not used during the normal operation of the chip (of course, the fabric could be removed from production versions of a high-volume chip). Existing synthesisable fabrics suffer a 6.4 times area overhead, compared to a hard programmable logic core [Wilt 05]. As will be shown in the next section, we address this by taking advantage of the datapath nature of the anticipated debug circuits. In addition, we take advantage of the fact that the context in which the core will be used is known when the SoC is designed. As an example, if buses are connected to the core, the specific pins on which these buses are mapped, as well as the width of each bus, are known when the fabric is instantiated, and will not change over the lifetime of the chip.

The third requirement is that the fabric should be as fast as possible. Ideally, we would like to run our integrated circuit at the highest possible speed during debugging. The nature of programmable logic means we may not be able to achieve this, but we would like to be as close as possible to this goal. Power consumption is a secondary concern, since the fabric will likely only be used “in the lab” during debugging. However, if our fabric is to be used to implement other arithmetic-oriented applications in a

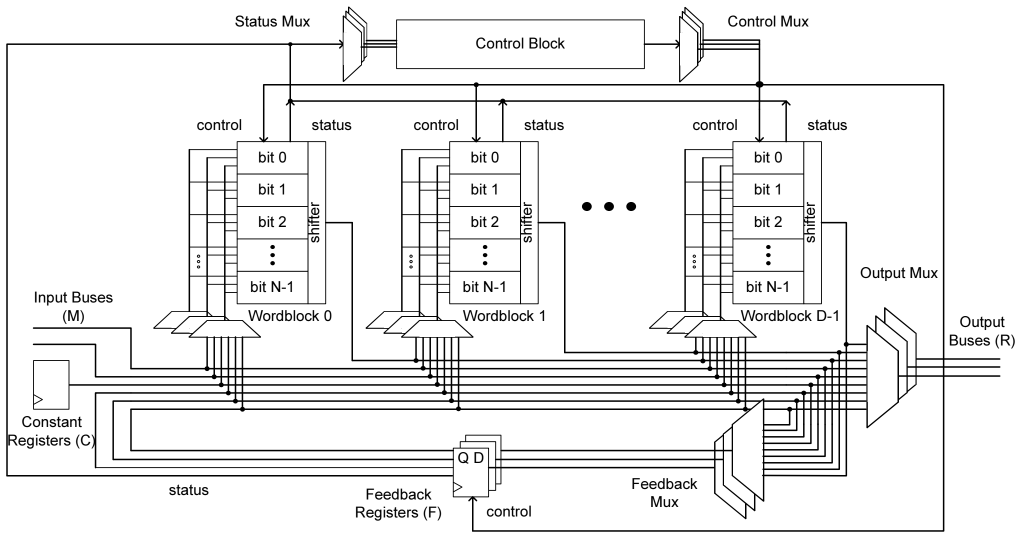


Figure 4.1: Fabric architecture (configuration elements not shown).

production version of an integrated circuit, then power consumption may become important.

Our methodology provides a unique opportunity for optimisation. When designing a hard layout for an FPGA, layout effort is reduced by dividing the design into tiles, where each tile is identical. In our case, the tiles are synthesised and laid out automatically by CAD tools; thus, it is no longer critical that each tile has to be identical.

4.3 Architecture

In this section, we describe a family of architectures for our embedded programmable logic core. Each member of the family is differentiated by various parameters. An SoC designer would select an architecture from this family based on the amount of programmable logic required, as well as the number and nature of the connections to the programmable logic.

Figure 4.1 shows our architecture. The fabric contains *D* identical *wordblocks*, each containing *N* identical *bitblocks*. Unlike a fine-grained FPGA, the bitblocks within a wordblock are all controlled by the same set of control bits. This means all bitblocks within a wordblock perform the same function. We will consider the impact of this feature on density in Section 4.5.

As shown in Figure 4.2, each bitblock contains two lookup-tables, several multiplexers, and a flip-flop. A single wordblock can implement an N bit adder/subtractor, an N-bit wide three-input multiplexer, any other three-input logic function, or some five-input functions. Two control inputs k_1 and k_2 (from the control block, to be described below) allow for efficient implementation of multiplexers and other datapath functions that require a control input. The same two control lines are driven to all bitblocks in a wordblock. The select lines of the multiplexers in Figure 4.2 as well as the function lines of the two lookup-tables are driven by configuration bits. In total, 35 configuration bits are required per bitblock; as described above, these bits are shared between all bitblocks in a wordblock. The wordblock also contains a programmable shifter, which can pass data through unchanged, or shift the word one bit to the right (signed or unsigned shift) or one bit to the left; the state of the shift block is controlled by two configuration bits.

Each wordblock receives up to three inputs from either the M primary bus inputs, the F feedback paths, the C constant registers, or any of the outputs of wordblocks to the left. The control lines for the input selection multiplexers are driven by configuration bits. Note that buses are switched as a unit; this improves density, since one set of configuration bits can be shared among all bits. However, it also reduces flexibility, since it is not possible to select part of one bus and part of another bus, although this functionality can be implemented within a wordblock by careful use of a "mask" in one of the C constant registers. The R output buses of the architecture can be selected from the same set of $M + F + C$ buses or from the output of any of the D wordblocks. The same signals (except the C constants) can be fed back, through a flip-flop, to all wordblocks; this provides a mechanism to connect wordblock outputs to the inputs of wordblocks to the left, and also supports an efficient way to delay signals by one clock cycle without using a wordblock.

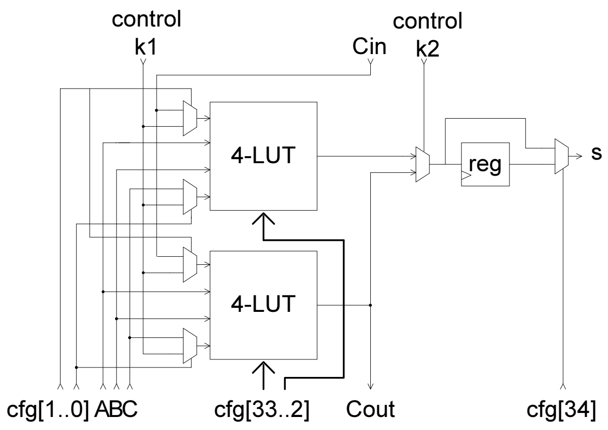


Figure 4.2: Bitblock (status flags not shown).

Multipliers are an important part of some target applications. Therefore, selected wordblocks in the fabric are replaced with embedded multipliers. Each embedded multiplier has two N-bit inputs which are selected from the $M + C + F + i$ (where i is the number of wordblocks to the left of the multiplier) buses using routing multiplexers. The multiplier produces two output buses, one for the high order result and one for the low order result. These outputs can be selected by all subsequent routing multiplexers including the output and feedback multiplexers. We denote the number of multipliers as A, and assume each multiplier displaces one wordblock (so, the number of wordblocks is $D - A$).

Although our architecture is aimed at datapath-oriented applications, a small amount of control logic is sometimes needed to control the datapath. Such logic can be implemented in the control block. This block contains fine-grained product-term based programmable logic resources, and is similar to the architecture described in [Yan 06]. The fabric contains P product-term blocks, each with 9 inputs, 10 product terms, and 3 outputs (this is shown to work well in [Yan 06]). The control block also contains registers to support state machines. Inputs to the control block are selected from a number of status signals generated throughout the datapath. Each wordblock generates a carry-out, an overflow, an MSB, an LSB, and a zero flag; each feedback path generates the same flags, with the exception of the carry-out. This large numbers of status bits are multiplexed into a small number of inputs using the status multiplexer, which is controlled by configuration bits. The exact number of these status bits that can be provided to the control block depends on the size of the control block. Similarly, the control block generates a number of outputs. These outputs can be provided to various control lines in the fabric using the control multiplexer; for each control line in the fabric, any of the control block outputs or the constants '0' or '1' can be selected.

The parameters used to describe the architecture are summarised in Table 4.1.

Symbol	Parameter descriptions
D	Number of wordblocks (including multipliers)
N	Bit Width
M	Number of input buses
R	Number of output buses
F	Number of feedback paths
C	Number of constant registers
A	Number of multipliers
P	Number of product-term blocks

Table 4.1: Architectural parameters.

A tool has been developed to generate synthesisable RTL description of the datapath FPGA fabric. By supplying parameters as described in Table 4.1, a Verilog description of the desired datapath FPGA fabric is generated. The description can be synthesised as ASIC and the area and delay data can be retrieved from the ASIC model. The generator allows us to produce many different datapath FPGA architectures rapidly. It greatly reduces the time to perform some "parameter sweep" experiments such as the one described in Section 4.5.

4.4 Example Mapping

To demonstrate how this architecture can be used to implement a circuit, we focus on a single example. The example is a common debugging operation; the circuit monitors two buses, and counts the number of times a certain mask (composed of 1's,

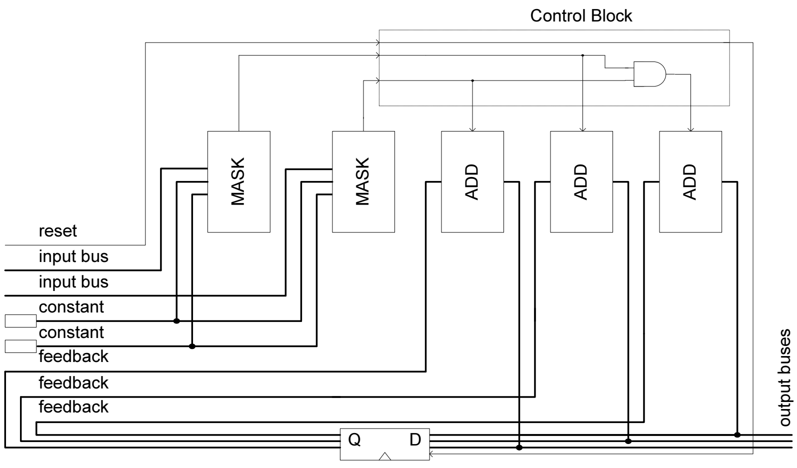


Figure 4.3: Example mapping.

Bit *i* from Constant value 1	Bit *i* from Constant Value 2	Meaning
0	0	Data bit *i* must be 0
0	1	Data bit *i* must be 1
1	0	Data bit *i* can be 0 or 1

Table 4.2: Meaning of mask bits in the example.

0's and "don't care" bits) matches each bus, as well as the number of times both buses match the mask at the same time.

Figure 4.3 illustrates how the application can be implemented. The mask value is represented by two constants. Each bit in the constant corresponds to one bit in the incoming data stream. As shown in Table 4.2, the two bits together determine whether the corresponding bit in the data stream must be a '1', '0', or is a "don't care" bit. The left-most wordblock in Figure 4.3 combines the incoming data word on the first input bus with the two constant values to determine whether the incoming data word is a match. To do this, each of the bitblocks within the wordblock performs the following function:

$$\overline{(a_i + b_i \oplus d_i)}$$

where a_i is bit i of the first constant, b_i is bit i of the second constant, and d_i is bit i of the incoming data word. If the result of this function is 0, a match in bit i has occurred. If *all* bits produce a result of 0, then a match has occurred. As described in

Section 4.3, each wordblock has a "zero" flag output that is asserted when the result from all wordblocks are 0; this flag is sent to the control block to indicate a match has occurred. The second wordblock in Figure 4.3 performs the same function, but uses the incoming data word on the second input bus.

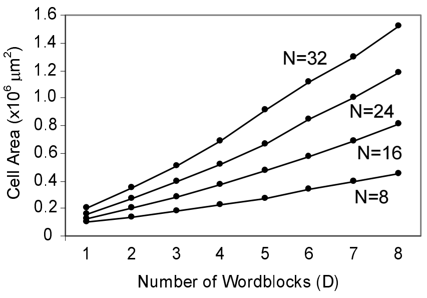
The control block then uses these two match flags to determine which counters to increment. If the first flag is set, the first counter is incremented (implemented using the third wordblock in Figure 4.3). The second counter is incremented when the second flag is set, and the final counter is incremented when both flags are set. Each of the three accumulated counts is stored in the feedback registers; these counts are fed back to the input signals of the adders. The reset control lines for the feedback registers are also controlled by the control block. Finally, the three adder outputs are connected to the outputs of the fabric.

4.5 Parameter Optimisation

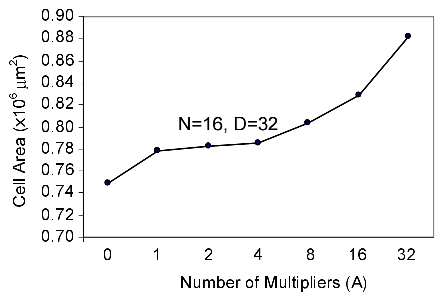
The datapath generator allows us to design fabric with different customisations. We want to observe how each parameters affect area in order to obtain and optimised customisation. In this section, we first determine the impact of the parameters in Table 4.1 on the area of the fabric. Delay and power will be considered in Section 4.6.5.

To obtain the area results, we can first use the datapath generator to create the fabric in Verilog by supplying designated parameters. The resulting Verilog files are synthesised using Synopsys Design Compiler V-2006.06. with UMC 0.13μm technology process. The area results are retrieved from the synthesis report.

Table 4.3 shows a breakdown of the area of a fabric with N=16, D=16, M=3, R=2, F=3, C=2, A=4, and P=4. The various components are synthesised using Synopsys Design Compiler, and the cell area predicted by the same tool is reported. All area values are given to three significant digits. Configuration circuits, clock circuits, and all other essential parts of the core are included in the synthesisable model. Although

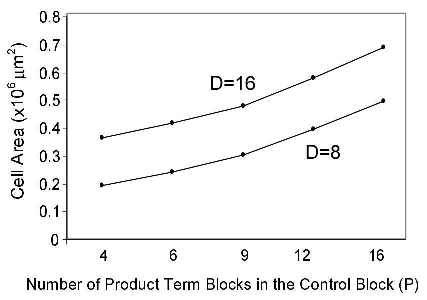


(a) Impact of D and N

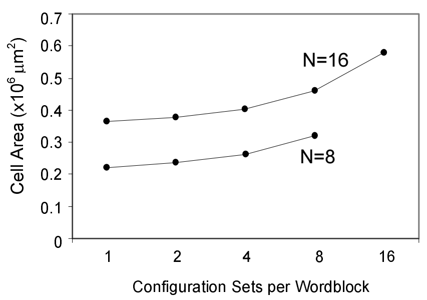


(b) Impact of A

Figure 4.4: Parameter sweeps, where M=3, R=2, F=3, C=2, A=4, P=4 unless otherwise specified.



(a) Impact of size of Control Block



(b) Impact of Wordblock granularity

Figure 4.5: Parameter sweeps, where M=3, R=2, F=3, C=2, A=4, P=4 unless otherwise specified.

it would be more accurate to perform place and route on the Synopsys-generated netlist and measure the chip area directly, [Wilt 05] have shown that the Synopsys area results have a good correlation to the final chip area results. A 130*nm* process is assumed.

As shown in the table, most of the area is used to implement the datapath portion of the fabric. Within the datapath, the largest component of the area is due to the routing multiplexers. The four multipliers and 12 wordblocks consume a significant amount of area. The configuration bits within the datapath consume 6.7% of the entire fabric.

Figure 4.4(a) shows the impact of N and D on area. In this experiment, M=3, R=2, F=3, C=2, A=4, and P=4. As the graph shows, the area is roughly proportional to both D and N; increasing D increases the number of wordblocks and corresponding

Module		Area in μm^2	Percentage
Datapath	wordblocks	86,300	23.8 %
	multipliers	45,200	12.5 %
	configuration bits	24,300	6.70 %
	feedback registers	2,320	0.600 %
	routing multiplexer	86,300	33.2 %
	total datapath	120,000	76.7 %
status multiplexer		18,500	5.10%
control multiplexer		14,600	4.00%
control block		51,400	14.2%
Total		363,000	100%

Table 4.3: Area breakdown.

routing multiplexers, while increasing N increases the sizes of these blocks.

The impact on area of the number of multipliers, A, is shown in Figure 4.4(b). All other parameters are as before, with N=16 and D=32. Intuitively, as A increases, the area goes up. This is the case despite the fact that the area of the 32-bit multiplier is roughly the same as the area of a 32-bit wordblock (including the associated routing multiplexers and configuration bits). The reason that the area goes up as A increases is that the multiplier produces two bus outputs (a wordblock produces one). This increases the size of the routing multiplexers in all downstream wordblocks, as well as the output multiplexers and feedback multiplexers. The graph shows that the increase from $A = 0$ to $A = 1$ is larger than the increase from $A = 1$ to $A = 2$. This is because if there is only one multiplier, it is placed in the left-most slot. This increases the size of all subsequent routing multiplexers. When a second multiplier is added, it is placed in the middle of the fabric, so only half of the routing multiplexers are increased (those to the right of the new multiplier).

Figure 4.5(a) shows the impact of P on the area of the fabric. As one can see, the number of product-term blocks in the control block has a significant effect on the size of the overall architecture.

We also measure the impact of M, R, C, and F. Each of these parameters has a linear effect on area. Increasing M from 1 to 8 increases the area by 15%, increasing R from

1 to 8 increases the area by 7.8%, increasing F from 0 to 6 increases the area by 25%, and increasing C from 0 to 8 increases the area by 17%. Parameter R (the number of output buses) has the smallest effect on area, since an increase in R does not imply an increase in the size of any of the routing multiplexers. For all other parameters, as the parameter is increased, additional buses are created; these buses are supplied to all routing multiplexers, making them larger. Parameter F has the largest impact since each feedback register is associated with three status bits and one control bit.

In our architecture, the same set of 35 configuration bits is shared among all bitblocks in a wordblock. To investigate the impact of this feature on density, we vary the number of configuration bit sets per wordblock from 1 (the baseline architecture) to N, in which every bitblock is controlled by a separate set of 35 configuration bits. The impact on area is shown in Figure 4.5(b) for two values of N, with all other parameters the same as before. As the graph shows, more flexible architectures with more configuration sets per wordblock require more area because of the extra configuration bits. For $N = 16$, an architecture in which each bitblock has its own configuration set is 60% larger than an architecture in which all bitblocks within a wordblock share a configuration set.

4.6 Results

In this section, we use benchmark circuits to compare our architecture to a fine-grained synthesisable programmable logic core [Yan 06] and to an ASIC implementation. We first describe our benchmark circuits. We then present mapping results in terms of area, first assuming that the architecture is tailored for each benchmark, and then assuming the more realistic case in which the fabric is not tuned for each benchmark. Individual path delays in the fabric are assessed. Delay and power consumptions are reported while using the same assumption as in Section 4.6.3. A proof-of-concept layout is illustrated to conclude the section.

4.6.1 Benchmark Circuits

To evaluate our architecture, we use a collection of datapath circuits. Although the primary motivation for our architecture is to implement debug circuits, the fabric can actually be used to implement datapath-oriented circuits. Thus, in order to fully exercise the fabric, we have created a suite of benchmark circuits representative of the types of circuits that would be implemented in our fabric. These circuits typically contain a single datapath controlled by a small controller. We focus on these single datapath circuits since circuits with multiple intersecting datapaths are likely too large to implemented using a synthesisable core.

We use ten benchmark circuits. Three of these are example debug applications, and the remainder are circuits that are similar in size and structure to the type of circuits that would be implemented in our core. The first debug circuit, *debug1* is the circuit described in Section 4.4. The second debug circuit, *seqchk* is a sequence number checking circuit. Many packet based inter-chip communication schemes (such as PCI Express) use sequence numbers to ensure that packets arrive in order and are not lost. The circuit monitors incoming data words, identifies the start of a packet (using a pre-determined mask), parses through the packet (using a counter) to find the sequence number, and compares it with the previous sequence number. Any out-of order sequence number, which would indicate a lost packet on a direct point-to-point link, increments a counter.

The third debug circuit, *fletcher*, can be used to detect checksum mismatches. In many communication applications, when a circuit detects a checksum mismatch, it will enter an error state and ask for re-transmission of the data. Determining that this is happening in a chip can often be an important step in the debugging process; it can explain low performance/throughput, it can alert the debugger that a different state of the circuit is being stimulated, and can potentially point to overall system problems. In its simplest form the checksum is calculated by simply adding the bytes in the data stream. However, this allows rearranged words or extra zero bytes to pass undetected.

The Fletcher algorithm contains an additional accumulator to help detect these error conditions [Flet 82, Naka 88]. The benchmark circuit monitors an incoming bus, and uses the Fletcher algorithm to compute the checksum of the incoming stream.

Of the remaining benchmarks, three, *bfly*, *dscg* and *fir4* are introduced in Chapter 2. The *bfly* benchmark performs the computation $z = y + x * w$ where the inputs and output are complex numbers; this is commonly used within a Fast Fourier Transform computation. The *dscg* circuit is the datapath of a digital sine-cosine generator. The *fir4* circuit is a 4-tap finite impulse response filter. The *dotv3*, *momul*, and *median* circuits are constructed for this work. The *dotv3* benchmark computes the dot product of two input vectors. The *egcd* circuit implements an extended binary greatest common divisor algorithm [Mene 96]. The *momul* benchmark is a Montgomery Multiplier [Mene 96]. Finally, the *median* circuit is a median filter that accepts streaming data and returns the median (actually second-largest) of the last four entries.

All benchmarks assume 8 bit operands, except *median*, *debug1*, *fletcher*, and *seqchk* which assume 16 bit operands. We have specifically chosen these circuits since they are small, and support the type of application we would expect to implement on a synthesisable programmable logic core. Large user circuits would be typically implemented using a hard programmable logic core.

4.6.2 Area Results - Optimised Parameters

We first compare our architecture to a previous synthesisable architecture [Yan 06] and to a non-programmable ASIC implementation of each circuit. This will give an upper-bound of the efficiency of our architecture if tuned properly.

To map each benchmark to our architecture, the benchmark is first split into datapath and control sections. The datapath portion of the circuit is mapped (by hand) to wordblocks, and appropriate values of D, N, M, R, D, A, F, and C are chosen. The control section is mapped to product-term blocks, using PLAmap [Chen 01]. Using

Benchmark	Fabric Parameters							
	D	N	M	R	C	F	A	P
debug1	5	16	2	3	2	3	0	1
seqchk	5	16	1	1	3	3	0	2
fletcher	8	16	1	2	2	3	0	2
bfly	8	8	6	1	0	5	4	0
dotv3	5	8	6	1	0	2	3	0
dscg	8	8	3	2	0	2	4	1
egcd	27	8	2	4	1	9	0	15
fir4	11	8	1	1	4	0	0	0
median	8	16	1	1	0	4	0	2
momul	13	8	7	2	0	6	1	8

Table 4.4: Parameters used for each benchmark circuit.

Benchmark	Datapath (ours) (μm^2)	Fined-Grain [Yan 06] (μm^2)	ASIC (μm^2)	Fine-Grain/ Datapath	Datapath/ ASIC
debug1	87,300	1,300,000	3,640	14.9	24.0
seqchk	92,500	1,200,000	3,600	13.0	25.7
fletcher	133,000	2,580,000	4,660	20.0	28.5
bfly	68,200	132,000,000	17,800	1,940	3.83
dotv3	34,100	65,500,000	8,350	1,920	4.08
dscg	72,200	116,000,000	11,600	1,610	6.22
egcd	1,230,000	22,800,000	9,880	18.5	124
fir4	76,200	131,000,000	12,100	1,720	6.30
median	142,000	10,700,000	5,270	75.4	26.9
momul	294,000	11,400,000	7,100	38.8	41.4

Table 4.5: Area results when the fabric is optimised for each benchmark circuit.

the number of product-term blocks required by PLAmap to implement the circuit, as well as the datapath parameters described above, a custom-built tool is used to generate an appropriately-sized fabric. The parameters use to construct the datapath for each benchmark circuit is shown in Table 4.4. Each fabric is then synthesised using Synopsys Design Compiler, and the cell area predicted by the same tool is reported. Again, a 130*nm* CMOS process is assumed. The results are shown in Column 2 of Table 4.5 (note that these results are slightly different than those from [Wilt 07] since here we assume all inputs and outputs are registered). All results are shown to three significant digits.

For comparison, we also show the area that would be required to implement the same circuit using the fine-grained synthesisable fabric from [Yan 06] in Column 3. These measurements are obtained using the architectures and tools described in [Yan 06]. We are unable to compare our architecture to the architecture described in [Wilt 05], since that architecture only supports combinational circuits, and most of our benchmarks are sequential. Column 4 shows the area required by the benchmark circuit if synthesised directly in macrocell, in which case there is no programmability.

Column 5 shows the ratio of the area required to implement each benchmark using the fine-grained fabric to the area required to implement the same benchmark in our architecture. As the table shows, there are two categories of circuits. Circuits *bfly*, *dotv3*, *dscg* and *fir4* all show ratios of between 1611 and 1940. In other words, our architecture is 1611 times to 1940 times more area-efficient than the fine-grained fabric. The remaining circuits show more modest ratios between 13.0 and 75.4.

These results are dramatic. First consider those benchmarks with ratios between 13.0 and 75.4. Given that, for each circuit, we are creating a fabric in which configuration bits are shared between either 8 or 16 bits, we would expect to see a ratio of no larger than 8 or 16. The reason our ratios are larger than these has to do with the inefficiencies of the fine-grained architecture when implementing very large circuits. The architecture in [Yan 06] is optimised for somewhat smaller circuits (between 10 and 300 equivalent 4-input lookup tables). As the fine-grained architecture is scaled to implement larger circuits, the size of the routing multiplexers grows. Each multiplexer has an input for every primary input and every output in the previous levels within the fabric. In [Yan 06], depopulating these multiplexers is not considered, since the circuits are small enough that the multiplexer area does not become unwieldy. In addition, the number of these multiplexers is proportional to the amount of logic in the fabric, since there is one multiplexer per product-term block input. This means that the overall size of the fabric grows quadratically with circuit size.

This quadratic increase in size suggests that the previous architecture is not efficient at implementing these sorts of large circuits. In addition, the architecture in [Yan 06] is optimised for control circuits rather than datapath circuits. Thus, the comparison to the fine-grained architecture must be made with caution. However, even if the fine-grained architecture is optimised for our benchmark circuits, we would still expect that our architecture would be significantly smaller than the fine-grained architecture.

The above explanation does not cover the four benchmarks that have ratios greater than 1600. These benchmarks all contain a significant number of multipliers. In our architecture, these multipliers are implemented as a hard embedded block (as in many commercial stand-alone FPGAs). However, the fine-grained architecture does not contain these embedded blocks, so the multipliers must be implemented using the normal logic resources. This is aggravated by the fact that product-term based architectures, such as [Yan 06] are notoriously bad at implementing XOR functions, which are common in multipliers.

Column 6 shows the ratio of the area required to implement each benchmark circuit in our fabric to the area required to implement the same benchmark circuit using fixed ASIC cells (with no programmability). This measure is the overhead resulting from configurability using our architecture. As the table shows, for the circuits with a significant number of embedded multipliers, this ratio is between 3.8 and 6.3. For circuits without a significant number of embedded multipliers, this number is between 24 and 124. It is interesting that these larger numbers are of the same order of magnitude as the ratio of an FPGA implementation to an ASIC implementation [Kuon 07]. In other words, the overhead due to configurability in our architecture is similar to the overhead inherent in a hand-designed stand-alone FPGA. This is a surprising result; it shows that synthesisable cores *can* provide the density that designers currently accept from non-synthesised programmable logic devices.

Benchmark	Fabric Parameters				Computed			
	D	N	M	R	C	F	A	P
debug1	7	16	2	3	2	4	2	3
seqchk	9	16	1	1	3	5	3	3
fletcher	11	16	1	1	3	6	3	4
bfly	16	8	6	1	4	8	4	6
dotv3	9	8	6	1	3	5	3	3
dscg	16	8	3	2	4	8	4	6
egcd	70	8	2	4	18	35	18	24
fir4	16	8	1	1	4	8	4	6
median	11	16	1	1	3	6	3	4
momul	24	8	7	2	6	12	6	8

Table 4.6: Parameters used for each benchmark circuit when low-level parameters are computed.

Benchmark	Datapath (ours) (μm^2)	Fine-Grain [Yan 06] (μm^2)	ASIC (μm^2)	Fine-Grain/ Datapath	Datapath/ ASIC
debug1	178,000	1,300,000	3,640	7.30	48.9
seqchk	220,000	1,200,000	3,600	5.45	61.1
fletcher	196,000	2,580,000	4,660	13.2	42.1
bfly	335,000	132,000,000	17,800	394	18.8
dotv3	226,000	65,500,000	8,350	290	27.1
dscg	325,000	116,000,000	11,600	357	28.0
egcd	3,190,000	22,800,000	9,880	7.15	323
fir4	307,000	131,000,000	12,100	427	25.4
median	272,000	10,700,000	5,270	39.3	51.6
momul	542,000	11,400,000	7,100	21.0	76.3

Table 4.7: Area results when low-level parameters are computed.

4.6.3 Area Results - Derived Parameters

When gathering the results in Section 4.6.2 we choose all fabric parameters independently for each circuit. This unfairly biases the results in our favour. One of the drawbacks of partitioning the fabric between controls and datapaths is that different user circuits require different amounts of controls and datapaths; since we do not know what will be implemented in the fabric when the ASIC is designed, choosing the amount of each type of fabric is difficult. If the partition is not chosen carefully, either control resources or datapath resources will be wasted. This is not a problem with fine-grained architectures, since the fine-grained fabric can be used to build either control or datapath structures. In this section, we address this issue by fixing this parameter (as well as other parameters) as a function of the fabric size.

We repeat the experiments in Section 4.6.2. We choose values of D, N, M, and R independently for each benchmark circuit. This is reasonable; when including a fabric in an ASIC, the bit-width, the number of input and output buses, and the desired fabric size are known. Unlike the previous experiments, however, we calculate the remaining parameters as a function of D. If the resulting architecture has more constant registers, feedback paths, multipliers, or product term blocks than are needed by the benchmark circuit, then the extra resources are wasted. If the fabric does not contain enough of any of these resources, the fabric size (D) is increased until the benchmark circuit can be implemented. The parameters used for each benchmark circuit are shown in Table 4.6. In all cases, we compute $C = \lceil \frac{D}{4} \rceil$, $F = \lceil \frac{D}{2} \rceil$, $A = \lceil \frac{D}{4} \rceil$, and $P = \lceil \frac{D}{3} \rceil$. Although these may not be the optimum ratios, we do not have enough benchmark circuits to determine optimum ratios for each parameter. These ratios are selected because they appear "reasonable" based on our experience (for example, since each product term block has three outputs, setting $P = \lceil \frac{D}{3} \rceil$ means that, on average, one select line per wordblock can be generated). If additional experiments are conducted, and the optimum ratios found, they would tend to improve the results in this section.

Table 4.7 shows the results, using the same columns as in Table 4.5. Again, all results are shown to three significant digits. The size of the fine-grained fabric and the ASIC implementation are copied into Table 4.7 for convenience. In general, the area required to implement each benchmark circuit on our fabric has increased, due to the benchmark circuits not exactly matching the generated architecture. The ratio of the area required to implement each circuit in the fine-grained architecture of [Yan 06] to the area required to implement the same benchmark in our fabric now ranges from 7.1 to 427, while the ratio of the area required to implement each circuit in our fabric to the area required to implement the same circuit in an ASIC ranges from 18.8 to

323.

4.6.4 Path Delay Results

Table 4.8 shows post-synthesis, pre-place and route delay estimates for various paths within the fabric. The delay through the wordblock is the delay from the output of the register in one wordblock to the input of the register in the next wordblock. This quantity is independent of N, and depends very slightly on M, C, and F, as well as the position of the wordblock in the array (since these parameters determine the size of the routing multiplexer used to select inputs for the second wordblock). The delay of the multiplier goes up as N increases. Measurements of the maximum carry-chain delay within one wordblock are also given in the table (from the carry-in of the least significant bit to the carry-out of the most significant bit). The last entry in the table shows the delay of a combinational path that passes through all wordblocks in a fabric with D=32 and A=8; clearly, most applications would not configure the fabric to have such a long critical path.

Delay through one wordblock	3.25ns
Delay through one multiplier (8 bits)	5.39ns
Delay through one multiplier (16 bits)	8.50ns
Delay through carry-chain (8 bits)	8.71ns
Delay through carry-chain (16 bits)	14.9ns
Delay through 24 wordblocks and 8 multipliers	178ns

Table 4.8: Delay estimates of paths within fabric.

4.6.5 Delay and Power Results - Derived Parameters

The delay and power dissipation of our architecture depend on the circuit implemented in the fabric. To estimate the delay and power overhead of our architecture, we map each of our benchmark circuits to a datapath constructed using the derived parameters from Table 4.6. For each mapping, we determine appropriate values for all configuration bits, and use Synopsys Design Compiler to estimate the critical path and dynamic power dissipated by the fabric with these configuration bits set properly. Again, a 130nm technology is assumed.

The results in this section are for the datapath portion of the fabric only. Measuring the delay paths through the control block is difficult. Determining the state of every programming bit in the datapath portion of the architecture is not difficult since there are only a small number of programming bits. However, since in the fine-grained control fabric, there are so many more programming bits, manually determining and setting the state of each bit would be infeasible.

Table 4.9 shows the results. Column 2 shows the critical path delay of each circuit implemented on our architecture, Column 3 shows the same quantity for each circuit implemented as an ASIC, and Column 4 shows the ratio between these two estimates. This ratio, which is the delay overhead imposed by reconfigurability, varies from 1.4 to 7.2. The larger ratios correspond to circuits that do not use the embedded multipliers. In our architecture, the embedded multipliers are implemented using ASIC circuitry, thus we would expect that circuits that make heavy use of the multipliers run closer to the speed of the corresponding ASIC implementation. As with the area results, these delay ratios are of the same order of magnitude as the ratio of the delay of a standard FPGA implementation to that of an ASIC implementation [Kuon 07]. This means that the delay overhead due to configurability in our architecture is similar to the delay overhead inherent in a hand-designed stand-alone FPGA. Unlike our results, however, [Kuon 07] found that the ratio does not depend strongly on the number of embedded multipliers used. This is likely because, in a standard FPGA, the delay of a net is primarily due to the routing connections between logic blocks, while in our fabric, the delay depends more on the gates and connections within each wordblock and multiplier.

The final three columns in Table 4.9 show power measurements for our architecture and an ASIC. The ratios vary from 2.5 to 65. In general, the circuits that do not use

Benchmark	Datapath (ns)	ASIC (ns)	Ratio	Datapath (mW)	ASIC (mW)	Ratio
debug1	14.6	2.02	7.23	2.7	0.13	21
seqchk	15.4	2.27	6.78	4.0	0.12	33
fletcher	16.5	8.37	1.97	5.8	0.23	25
bfly	11.1	2.81	3.95	3.0	1.19	2.5
dotv3	9.94	3.75	2.65	1.7	0.52	3.3
dscg	7.64	4.72	1.62	2.6	0.70	3.7
egcd	14.3	6.65	2.15	26	0.40	65
fir4	10.5	4.21	2.49	2.3	0.62	3.7
median	16.5	2.33	7.08	4.6	0.44	10
momul	7.53	5.34	1.41	4.2	0.41	10

Table 4.9: Datapath delay and power estimates for configured fabric.

embedded multipliers show a larger ratio, as expected. The ratio for *egcd* is significantly larger than the others. As shown in Table 4.4, this circuit requires more control logic than the other circuits. Because we are fixing the ratio of control resources (P) to wordblocks (D), a fabric large enough to implement the control part of the circuit has many more wordblocks than are needed. In our architecture, these unused wordblocks consume power (this suggests that we should "turn off" unused wordblocks, however we do not consider this in this work). In [Kuon 07], it is reported that a standard FPGA dissipated 14 times more power than an ASIC; once again, this is in-line with our results.

4.6.6 Proof-of-Concept Layout

As a proof-of-concept, we perform place and route on the datapath portion of our fabric with D=12, N=8, M=7, R=2, F=6, A=0, and C=0 and is shown in Figure 4.6. The Verilog description of the fabric is synthesised with Synopsys Design Compiler, targeting the STMicroelectronics 90nm, 7-layer metal process using the STMicroelectronics CORE90GPSVT macrocell library. The netlist is flattened into a single level of hierarchy before layout. The pre-layout netlist contains a total gate area of 300098 μm^2. The cell placement, cell sizing and repeater insertion is performed by Cadence SoC Encounter. Detailed wire routing is performed using Cadence

NanoRoute and is completed with no violations. The total gate area after place and route is 336402 μm^2. The placement region set to approximately 625 $\mu m \times 625\ \mu m$, resulting in a gate density of 86.1%.

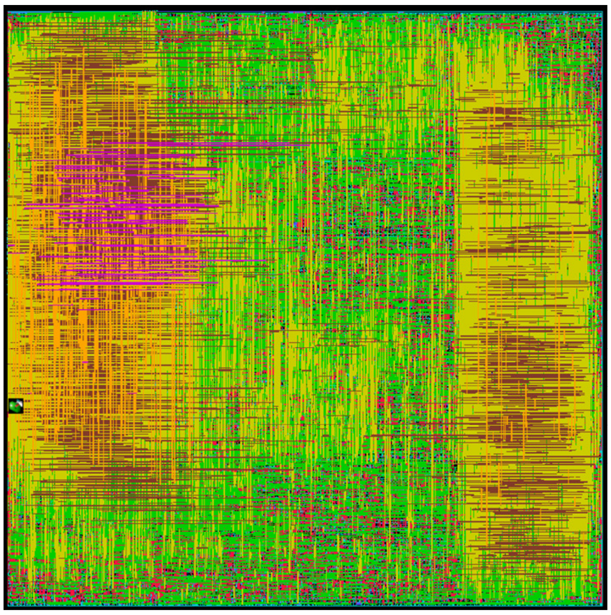


Figure 4.6: Proof-of-concept layout.

4.7 Comparison to Previous Work

Our architecture inherits ideas from previous work on fine-grained synthesisable fabric, datapath-oriented FPGAs and coarse-grained reconfigurable architectures, such as RaPiD [Cron 99]. This section compares our architecture to several previous studies, as well as to architectures that have been previously proposed for debugging.

4.7.1 Alternative Debugging Architectures

Several other embedded debugging architectures have been proposed. Abramovici, et al. have described their reconfigurable design-for-debug infrastructure for SoCs [Abra 06]. Like our proposal, this infrastructure is targeted at general-purpose digital logic in a SoC design. Their architecture, however, is based on a distributed heterogeneous reconfigurable fabric. Distributing debug circuitry across the chip has the advantage that the debug logic is likely to be positioned closer to the source of the monitored signals. However, distributed circuitry makes it more difficult to combine the debugging resources to implement larger debugging functions. A centralised scheme like ours can likely support more complex debugging operations, and perhaps can better amortise the cost of the debugging circuitry across different parts of the SoC. Another difference between our architecture and that in [Abra 06] is that ours does not require the identification of specific trigger signals when the debugging circuitry is instantiated and connected to the SoC.

Sarangi et al. have described a proposal for using programmable hardware to help patch design errors in processors [Sara 07]. As part of their patching process they make use of programmable logic to detect specific conditions in the processor. In some cases, after the detection of these specific problem conditions, they can make use of existing processor features, such as pipeline flushes, cache re-fills, or instruction editing to correct the error; in other cases they can cause an exception to be serviced by the operating system or hypervisor. The primary motivation of their proposal is the in-field correction of processor design errors, and not post-silicon debug, however it is clear that their proposal could also be used for post-silicon debug. Because their architecture has been designed with one application in mind, it may not be general enough for implementing debugging circuits useful in other types of integrated circuits.

A similar proposal is described in [Wagn 06]. The focus in that work is on providing a configurable state machine that matches error states in a processor and takes corrective action when these error states occur during system operation. Although this is not designed specifically for post-silicon debugging, it may be helpful in uncovering some types of design errors in a processor. Again, however, it is not as flexible as our architecture in which more general debug circuits can be implemented.

Previous work [Quin 05] describes another design-for-debug proposal. In that work, we employ a fine-grained programmable logic core based on a standard FPGA architecture. This previous work leads to perhaps the most general implementation of programmable debug circuitry, but it suffers from the overhead implicit in a fine-grained architecture. In addition, this previous core is not synthesisable, which is a key attribute of the architecture described in this work.

4.7.2 Fine-Grained Synthesisable Fabric

Although previous fine-grained synthesisable fabrics are not designed with debugging in mind, they could be used for this purpose. We have compared our architecture to a previous synthesisable architecture in Section 4.6.2 using a set of benchmark circuits. The architecture proposed in [Yan 06] is fine-grained and the reconfigurability is provided by programmable logic arrays (PLA). For the circuits which contain significant number of multipliers, our architecture is 1610 times to 1940 times more area-efficient than the fine-grained fabric. This is because the multiplier in our architecture is implemented as a hard embedded block while the fine-grained architecture does not contain these blocks. It means the multipliers must be implemented using normal logic resources which contribute to large area consumption.

For some other circuits which do not have a large number of multipliers, the area ratio is between 13 and 75. We observe that the architecture in [Yan 06] is not efficient when implementing large circuits. The architecture in [Yan 06] contains many routing multiplexers. Both the size of these multiplexers and the number of multiplexers grow linearly with the size of fabric. When the fabric is scaled sufficiently large to implement the given benchmark circuits, these multiplexers become unwieldy and cause the area to grow significantly.

4.7.3 Datapath-Oriented FPGAs

Several previous studies have considered datapath-oriented FPGAs [Cher 96, Hauc 04, Leij 03, Ye 06, Ye 03]. In these architectures, configuration bits are shared among multiple lookup-tables and multiple routing switches. Again, these could also be used for debugging.

In these previous works, it is assumed that the FPGA is to be laid out by hand or using a custom layout tool, and thus, no attempt is made to remove combinational loops in the unconfigured fabric. The absence of combinational loops is a key requirement of a synthesisable architecture. Although these architectures can be synthesised (as in [Leij 03]), the combinational loops will require designers to resolve these loops by declaring false paths; this increases the difficulty of including these fabrics in a large SoC.

A second difference between these datapath FPGAs and our architecture is that these previous architectures have been optimised assuming that the bus width of the target application and the pin assignments of the buses are not known when the fabric is designed. This limits the amount of optimisation possible; for example, in [Ye 06], it is found that the number of blocks sharing a set of configuration bits should be no more than four. In our context, the bus width and pin assignments are determined when the ASIC is designed, and will not change over the lifetime of the chip. This allows us to share a set of configuration bits across all datapath bits in a word.

4.7.4 Coarse-Grained Fabrics

Coarse-grained architectures, in which lookup-tables are replaced by ALUs, have also been described in [Cron 99, Gold 00, Mars 99, Sing 00]. Of these, the RaPiD architecture [Cron 99] is specifically designed for use in an SoC. RaPiD contains a linear array of dedicated functional units connected using dedicated buses. Control logic is implemented using a separate module that provides control signals to the functional units.

RaPiD is intended to support fairly large applications such as image and signal processing, and may be best implemented as a hard programmable logic core. It would be possible to “scale down” RaPiD and use it as a synthesisable core. However, like the datapath FPGAs described in the previous section, the unconfigured RaPiD fabric contains combinational loops. Our architecture eliminates these using a directional routing network.

Another difference between RaPiD and our architecture is that RaPiD (as well as many coarse-grained architectures) contains a heterogeneous mix of fixed-function datapath elements rather than configurable wordblocks. When creating a RaPiD fabric, one must choose the number of each type of functional unit to be included in the fabric. However, once that decision is made, the *location* of each functional unit does not matter, since buses can be routed from any functional unit to any other functional unit. In our architecture, however, the routing network requires less area but is less flexible, so it is less likely that a pre-positioned set of fixed functional units could be connected to implement a target application. Thus, we provide a general-purpose wordblock that can be used to implement many functions. The only exceptions to this rule are the embedded multiplier blocks; we distribute these evenly across the fabric to maximise the likelihood that applications can be mapped successfully.

A list of comparison between different type of datapath or coarse-grained reconfigurable fabric can be found in Table 4.10.

4.8 Summary

We have presented an architecture for a datapath-oriented synthesisable FPGA core which can be used to provide post-fabrication flexibility to an SoC. The primary application of such a core is to enable efficient on-chip debugging, but it can also be used

	Synthesisable FPGA	Cherepacha [Cher 96]	Leijten-Nowak [Leij 03]	RaPiD [Cron 99]	PipeRench [Gold 00]
Coarse-grained Logic	shared-bit LUT	shared-bit LUT	shared-bit LUT	functional units (registers, multiplier, shifter, adder, ALU)	ALU, LUT
Other Logic	carry-chain, multiplier	carry-chain, shifter	carry-chain	not available	carry-chain, shifter, configuration controller
Fine-grained Logic	programmable logic array	not available	a configuration of coarse-grained logic	not available	virtualisation
Routing Architecture	bus-based routing, unidirectional	bus-based routing, dedicated control signal routing	bus-based routing, partial bit-based routing	bus-based routing, dedicated control signal routing	strip-based routing, global routing for selected registers
Configuration Options	4-input bus-based logic, adder, multiplexer, FIFO	bus-based logic, adder, shifter, multiplexer	bus-based logic, adder, multiplier, multiplexer, random logic	not available – each functional unit has hardwired function	adder, multiplier, random logic
Modelling flow	Standard macrocell	Analytical	Standard macrocell	Custom layout, Physical measurement	Standard macrocell for datapath, custom layout for routing
Technology	0.13μm	1.2μm	0.13μm	0.5μm	0.25μm

Table 4.10: Comparison of coarse-grained reconfigurable fabric.

to implement small datapath circuits. The proposed architecture features sharing configuration bits, carry-chains, directional routing architecture and embedded multipliers. Compared to a previous synthesisable embedded programmable logic core, our architecture is between 7 times and 427 times more area efficient, depending on the number of embedded multipliers in the fabric. This opens the use of synthesisable embedded programmable logic cores to significantly larger applications, and provides a configuration overhead similar to that of standard hand-designed FPGAs. We have shown that the delay and power overhead of our architecture is also similar to that of standard FPGAs. A proof-of-concept layout of the core is described.

There are two important limitations of these comparisons. First, the fine-grained architecture is optimised for smaller control-type circuits, and thus is inefficient at implementing larger datapath circuits. If the fine-grained architecture is optimised for our benchmark circuits, the difference between the fine-grained and datapath architecture would be reduced significantly. Second, the fine-grained architecture does not contain embedded multipliers, while the datapath architecture does. If multipliers are added to the fine-grained architecture, the fine-grained architecture would perform better on those circuits that contain multiplication (four of our ten benchmark circuits).

Using standard macrocell design flow, we show that customisability of datapath FPGA fabric. The parameter sweep approach presented in the chapter allows us to identify how the architecture parameters affect the performance of the circuits. The same approach can be applied on other architectures or designs which involves parameterised design flow.

Chapter 5

Floating Point FPGA: Architecture and Modelling

5.1 Introduction

This chapter introduces the architecture of FPFPGA. The architecture is an extension of the one discussed in Chapter 4. Apart from bus-based logic fabric, the proposed FPFPGA architecture has both fine-grained and heterogeneous blocks, such usage of multiple granularity having advantages in speed, density and power over more conventional heterogeneous FPGAs. The heterogeneous block is used to implement the datapath, while lookup table (LUT) based fine-grained resources are used for implementing state machines and bit level operations. In our architecture, the heterogeneous blocks have flexible, parameterised architectures which are synthesised from a hardware description language. This allows tuning of the parameters in a quantitative manner to achieve a good balance between area, performance and flexibility.

One major issue when evaluating new architectures is determining how a fair comparison to existing FPGA architectures can be made. The Versatile Place and Route (VPR) tool [Betz 97] is widely used in FPGA architecture research. However, the CAD algorithms used within are different to those of modern FPGAs, as is its underlying island-style FPGA architecture. As examples, VPR does not support retiming, nor does it support carry-chains which are present in all major FPGA devices. To enable modelling of our FPFPGA and comparison with a standard island-style FPGA, we adopt the methodology presented in Chapter 3. Using this method, the impact of incorporating embedded elements on performance and area can be quickly evaluated, even if an actual implementation of the element is not available.

Since total power consumption of FPGAs increases with each reduction of integrated circuit feature size, power reduction has become one of the primary concerns in FPGA architecture. Power consumption can be divided into two parts, static and dynamic power consumption. Static power dissipation is due to leakage while dynamic power dissipation is due to switching activity of the circuits. It is reported that a commercial FPGA device with 90*nm* technology process consumes 62% of its total power as dynamic power [Tuan 06]. This chapter extends the VEB methodology to illustrate how dynamic power estimation can be achieved under this design flow.

As explained in later context of this chapter, the core of a FPFPGA is a set of heterogeneous blocks packed with the most frequently used logic to reduce area and delay. In the design phase of these heterogeneous blocks and FPFPGA architecture, one critical issue is that how to rapidly estimate the dynamic power consumption across a number of candidate designs. Traditional power measurement involving retrieval of switching activity based on post-place and route simulation may not be practical as for every new architecture, the application circuit has to be manually mapped and the configuration of the heterogeneous block may be different each time. In addition, simulation test vectors have to be adjusted according to the mapped design. Such estimation is too tedious and is usually not necessary for initial design exploration.

To address this issue, we propose a high level dynamic power consumption estimation technique for FPFPGAs. We assume constant activity rate on all the nets in a design. The basic idea is to measure the power consumption of the heterogeneous block with ASIC tools and the power consumption of the fine-grained unit with FPGA design tools under the same conditions. The total power estimate can be given by the sum of these two measurements. This approach is briefly described in Chapter 3 and this chapter provides an extensive review to this approach and demonstrate how to use this approach to estimate the dynamic power consumption of FPFPGA.

This chapter is organised as follows. Section 5.2 analyses the characteristics appearing in most floating point applications and suggests a list of requirement for the architecture of FPFPGA device. It is then followed by the proposed architectures. Section 5.3 demonstrates an example on using an FPFPGA device to implement application circuits. Matrix multiplication is employed as an example. Section 5.4 presents a modelling flow to estimate the performance delivered by the FPFPGA. It is an extension from Chapter 3 and we include area, delay and power estimation for an FPFPGA device. Section 5.5 reports the results compared with traditional FPGA devices. Section 5.6 compares with previous attempts in designing reconfigurable architecture for floating point computations and a summary is given in Section 5.7.

5.2 FPFPGA Architecture

5.2.1 Requirements

Before we introduce the FPFPGA architecture, common characteristics of what we consider a reasonably large class of floating point applications which might be suitable for signal processing, linear algebra and simulation are first described. Although the following analysis is qualitative, it is possible to develop the architecture in a quantitative fashion by profiling application circuits in a specific domain.

In general, FPGA-based floating point application circuits can be divided into control and datapath portions. The datapath typically contains floating point operators such as adders, subtractors, and multipliers, and occasionally square root and division operations. The datapath often occupies most of the area in an implementation of the application. Existing FPGA devices are not optimised for floating point computations and for this reason, floating point operators consume a significant amount of FPGA resources. For instance, if the embedded DSP48 blocks are not used, a double precision floating point adder requires 701 slices on a Xilinx Virtex 4 FPGA, while a double precision floating point multiplier requires 1238 slices on the same device [Xili 05].

The floating point precision is usually a constant within an application. The IEEE 754 single precision format (32-bit) or double precision format (64-bit) is commonly used.

The datapath can often be pipelined and connections within the datapath may be uni-directional in nature. Occasionally there is feedback in the datapath for some operations such as accumulation. The control circuit is usually much simpler than the datapath and therefore the area consumption is typically lower. Control is usually implemented as a finite state machine and most FPGA synthesis tools can produce an efficient mapping from the Boolean logic of the state machine into fine-grained FPGA resources.

Based on the above analysis, some basic requirements for FPFPGA architectures can be derived.

1. A number of coarse-grained floating point addition and multiplication blocks are necessary since most computations are based on these primitive operations. Floating point division and square root operators can be optional, depending on the domain-specific requirement.

2. Coarse-grained interconnection, fabric and bus-based operations are required to allow efficient implementation and interconnection between fixed-function operators.

3. Dedicated output registers for storing floating point values are required to support pipelining.

4. Fine-grained units and suitable interconnections are required to support implementation of state machines and bit-oriented operations. These fine-grained units should be accessible by the coarse-grained units and vice versa.

5.2.2 Architecture

In Chapter 4, the reconfigurable fabric of datapath FPGA consists of wordblocks and control blocks. The wordblock is designed for bus-based computation while the control block is designed for random logic. The following observations have been made regarding to such architecture:

1. The control block is implemented in programmable logic array style and thus is not effective to handle random logic when compared with LUT approach.
2. The functionality of the wordblock can be more domain-specific. For example, floating point circuit hardly benefits from the generic wordblock.
3. It is not necessary to bundle the control block and wordblock together. Some circuits may require more wordblock for datapath construction. Other may demand more control block such as state machine circuitry.

Considering both the observations of datapath FPGA and the requirement of FPFPGA, we invent a novel FPGA architecture which captures the essence of datapath FPGA and existing island-style FPGA as follows:

1. Island-style FPGA architecture is maintained. The routing topology and LUT-based fine-grained unit is the same as existing commercial FPGA device. However, certain amount of heterogeneous datapath FPGA fabric is embedded as floating point arithmetic extension.
2. The datapath FPGA fabric resembles the one presented Chapter 4. The control block is removed. Some generic wordblocks are replaced with floating point

specific circuit. This modified datapath FPGA fabric is referred to *coarse-grained unit* in the remaining of the thesis.

Figure 5.1 shows a top-level block diagram of our FPFPGA architecture. It employs an island-style fine-grained FPGA structure with dedicated columns for coarse-grained units. Both fine-grained and coarse-grained units are reconfigurable. The coarse-grained part contains embedded fixed-function floating point operators such as adders and multipliers while it is surrounded by fine-grained unit. The connection between coarse-grained units and fine-grained units is similar to the connection between embedded blocks (embedded multiplier, DSP block or block RAM) and fine-grained units in existing FPGA devices.

The coarse-grained logic architecture is optimised to implement the datapath portion of floating point applications. The architecture of each block is illustrated in Figure 5.2. Each block consists of a set of floating point multipliers, adder/subtractors, and general-purpose bitblocks connected using a uni-directional bus-based interconnect architecture. Each of these blocks will be discussed in this section. To keep our discussion general, we have parameterised the architecture as shown in Table 5.1. There are D subblocks in each coarse-grained block. P of these D subblocks are floating point multipliers, another P of them are floating point adders and the rest $(D-2P)$ are general-purpose wordblocks. Specific values of these parameters will be given in Section 5.4. Generic wordblock can configure as register, FIFO, fixed point addition, subtraction, multiplexor, floating point comparison and other bus-based 4-input logic.

We decide not to embed a whole FPU in an FPFPGA as we demonstrate in Chapter 3. Instead, we decompose an FPU to different floating point operators such as adders and multipliers. Those units then cluster together and connected using bus-based routing supported by bus-based generic wordblock. Each cluster, also known as coarse-grained unit, is surrounded by fine-grained logic. We believe such architecture can yield better performance since every single floating point operator can operate in parallel. In addition, decomposing the FPU can allow more flexible combination when construction a datapath to floating point applications.

Symbol	Parameter description
D	Number of blocks (including floating point blocks and wordblocks)
N	Bus width
M	Number of input buses
R	Number of output buses
F	Number of feedback paths
P	Number of floating point adders and multipliers

Table 5.1: Parameters for the coarse-grained unit.

The core of each coarse-grained block contains P multiplier and P adder/subtractor subblocks. Each of these blocks has a reconfigurable registered output, and associated control input and status output signals. The control signal is a write enable signal that controls the output register. The status signals report the subblock's status flags and include those defined in IEEE standard as well as a zero and sign flag. The fine-grained unit can monitor these flags via the routing paths between them.

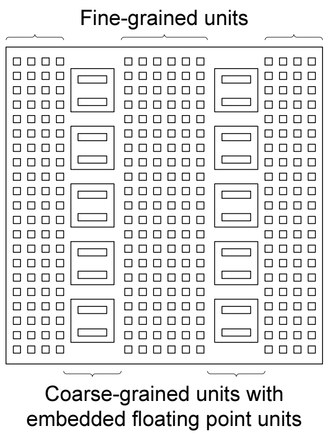


Figure 5.1: Architecture of the FPFPGA.

Each coarse-grained block also contains general-purpose wordblocks as mentioned in Chapter 4. Apart from the control and status signals, there are M input buses and R output buses connected to the fine-grained units. Each subblock can only accept inputs from the left, simplifying the routing. To allow more flexibility, F feedback registers have been employed so that a block can accept the output from the right block through the feedback registers. For example, the first block can only accept

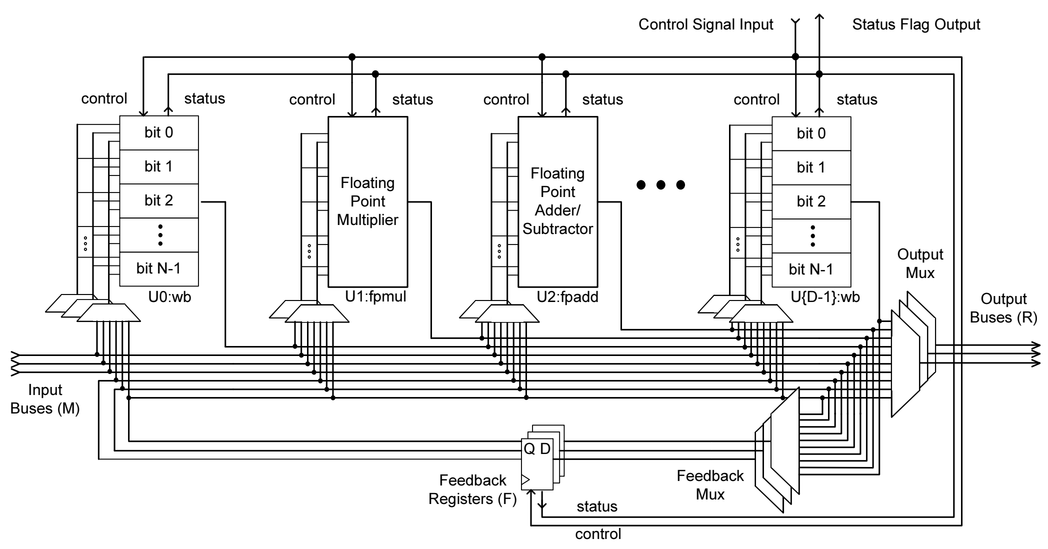


Figure 5.2: Architecture of the coarse-grained unit.

input from input buses and feedback registers, while the second block can accept input from input buses, the feedback registers and the output of the first block.

Each floating point multiplier is logically located to the left of a floating point adder so that no feedback register is required to support multiply-add operations, which appear in many applications such as DSP and scientific calculation. The coarse-grained units can support multiply-accumulate functions by utilising the feedback registers. The bus width of the coarse-grained units is 32-bit for the single precision FPFPGA and 64-bit for double precision one.

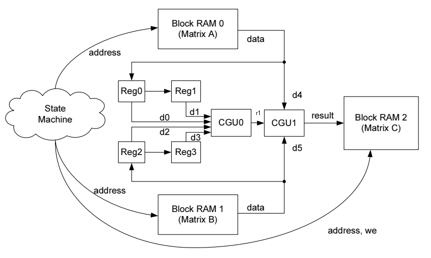
Switches in the coarse-grained unit are implemented using multiplexers and are bus-oriented. A single set of configuration bits is required to control each multiplexer, improving density compared to a fine-grained fabric.

The novel aspect of the proposed architecture lies on both customisability and reconfigurability. During the architecture design phase, (i.e. the stage before fabrication), FPGA designers can choose the suitable parameters according to the domain-specific requirement. In particular, the designer can change the functionality of each block. While we present generic wordblock, floating point adder and floating point multiplier in this work, each wordblock can essentially replace by any bus-based computation unit. For example, if the application-domain demands large amount of floating point division, the designer may want to embed couples of floating point divider or inversion unit instead of floating point adder into the coarse-grained fabric by trading off area.

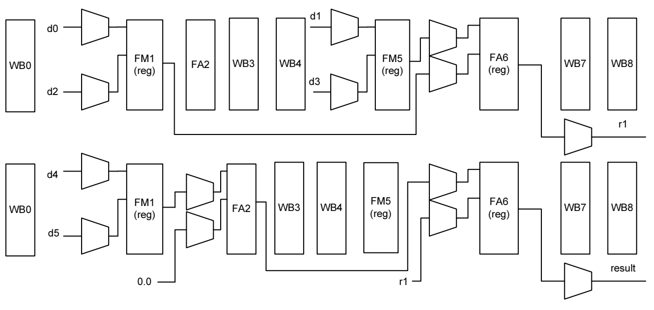
Another important parameter of the FPFPGA architecture is the number of input and output. Both parameters affect the number of routing channel inside the coarse-grained unit. One should notice that each coarse-grained block may have different customisation. Even though each customisation generates different layout, it can still be possible to embed into FPGA as long as it have the same width so that the coarse-grained units can be packed into the same column. It means that different combination of coarse-grained units can co-exist with each other in the same FPGA device. This further enhances the customisability of FPFPGA.

We modify the circuit generator in Chapter 4 to produce FPFPGA circuit and two more parameters are introduced. One can specify the number of floating point adders and multipliers to the generator to produce synthesisable RTL description of the corresponding coarse-grained unit.

5.3 Example Mapping



(a) Fine-grained unit mapping.



(b) Coarse-grained unit mapping.

Figure 5.3: Example mapping for matrix multiplication.

To illustrate how our architecture can be used to implement a datapath, we use the example of a floating point matrix multiply. Figure 5.3 illustrates the example datapath and the implementation of this datapath on our architecture. In this example, we assume an architecture in which the multiplication subblocks are located in the second and sixth subblock within the architecture and floating point adder/subtractor units are located in the third and the seventh subblock.

The datapath of this example application can be implemented using two coarse-grained blocks. The datapath produces the result of the equation $d0 \times d2 + d1 \times d3 + d4 \times d5$. The first coarse-grained unit performs two multiplications and one addition. The result ($r1$) is forwarded to the next coarse-grained unit. The second coarse-grained unit performs one multiplication and one addition. However, as all multiplications start in the same clock cycle, the last addition cannot start until $r1$ is ready. In order to synchronise the arrival time of $r1$ and $d4 \times d5$, another floating point adder (FA2) in the second coarse-grained block is instantiated as a FIFO with the same latency as FA6 in CGU0. This demonstrates an alternate use of a coarse-grained unit. Finally $r1$ and $d4 \times d5$ are added together and the state machine sends the result to the block RAM. All FPU subblocks have an enabled registered output to further pipeline the datapath.

5.4 Modelling

5.4.1 Overview

In this section, we describe how to employ the VEB methodology to assist the modelling of FPFPGA. The host FPGA supplies fine-grained unit model while the coarse-grained unit model can be described using register transfer language (RTL) and the corresponding characteristics can be obtained by following an ASIC macrocell library development approach. We also provide addition information on how to estimate the dynamic power consumption.

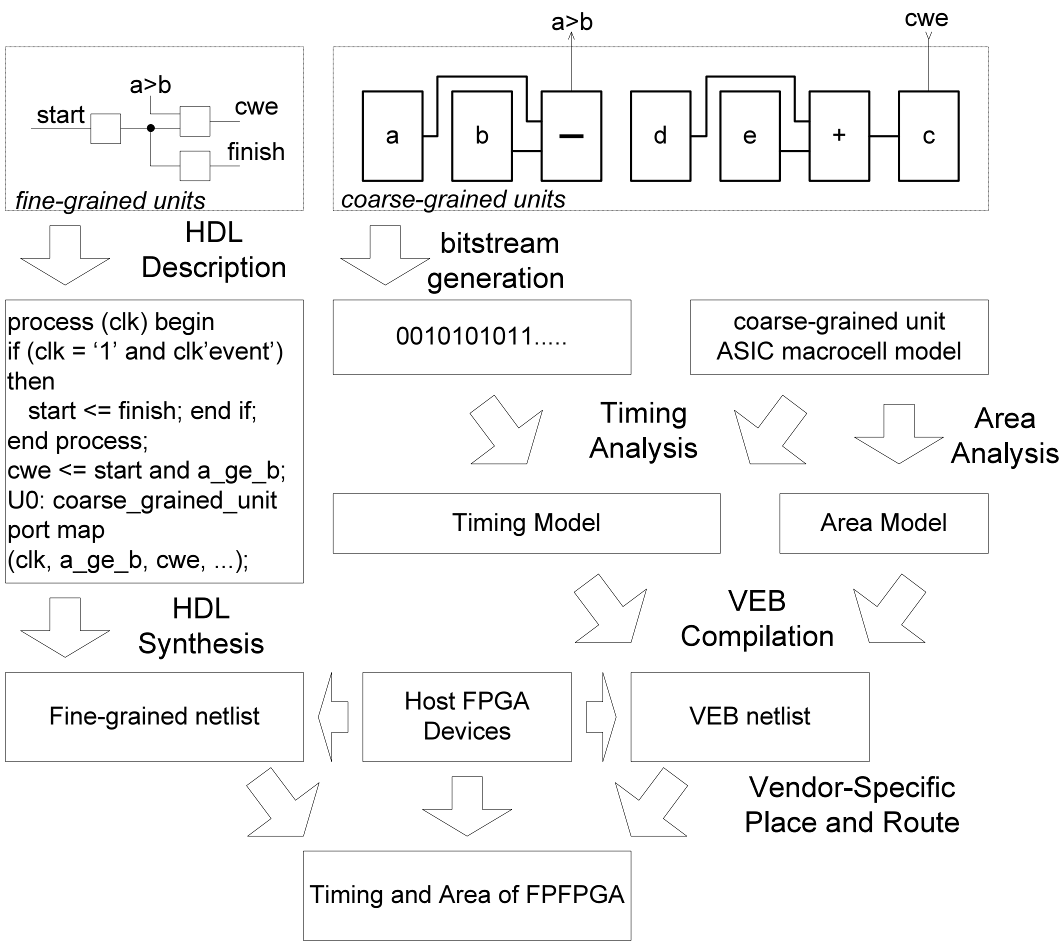


Figure 5.4: Modelling flow overview.

Figure 5.4 illustrates the modelling flow using the VEB methodology. The input is a high level application description and the output is an FPGA bitstream. We define the parameters used in constructing the coarse-grained unit of FPFPGA. The application is first broken into control logic and datapath portions. The datapath portion is then mapped to the pre-defined coarse-grained unit whenever possible. Fine-grained units representing those parts of the circuit that will be mapped to lookup-tables in the cases that no suitable coarse-grained unit is found or all have been used. This procedure is done manually as it is illustrated in Section 5.3.

While the area model of the VEB experiment presented in Chapter 3 is obtained from the physical layout of existing design, there is no such coarse-grained unit available as physical layout since it is still in design stage. The area model is therefore obtained from other means. For instance, we apply the standard ASIC macrocell design flow to obtain area model. The circuit generator discussed in Chapter 4 is extended to support generation of floating point operator in datapath FPGA fabric. As soon as the circuit generator produces a RTL description of the coarse-grained unit according to the input parameters, the description can then be fed into synthesis tool and the area model can be retrieved from the standard ASIC macrocell design flow.

It is important to note that timing information cannot be determined before programming the configuration bits. Otherwise, the synthesis tool reports the worst case scenario where the longest combinational path from the first wordblock to the last wordblock is considered as critical path and this is usually not the correct timing in most designs. To address this issue, the tool has to recognise the configuration of the coarse-grained unit before the timing analysis. Therefore, a set of configurations is generated during manual mapping, and the associated bitstream can be used in timing analysis. This bitstream can be imported to the timing analysis tool as case analysis, so the tool can identify false paths during timing analysis and produce correct timing for that particular configuration.

5.4.2 Power Modelling

This section illustrates a detail power estimation flow of FPFPGA architecture. While Chapter 3 mentions the basic idea of the power estimation flow, this section discusses how to apply the idea to obtain the power consumption of FPFPGA device. We apply the same assumption as discussed in Section 3.2.1 throughout this section.

A web-based power estimation tool [Xili 08b] is provided by the vendor which involves a spreadsheet. The spreadsheet requires users to specify the frequency, number of registers used, number of LUT used, number of embedded multiplier used, number of block memory used, amount of routing used and the average toggle rate of the design. Before we estimate the dynamic power consumption on the FPFPGA, we employs the standard VEB flow described in Section 5.4.1 once to obtain the area and the delay of the FPFPGA. The information is required during the dynamic power estimation. The dynamic power consumption of fine-grained units (P_{fgu}), coarse-grained units (P_{veb}) the associated output loading (P_r) are estimated separately.

Fine-grained units (P_{fgu})

When estimating the power consumption of the fine-grained unit using the web-tool, we choose a medium amount of routing resources because circuits implemented on fine-grained units are mostly control signals which are usually random logic. According to the vendor's suggestion [Xili 08b], medium routing should be selected for random logic. In addition, 12.5% toggle rate on all the nets is specified. There are other information can be specified in the power estimation tool such as the I/O cells and clock configurations. As the power estimation on I/O cells and clock management units are not included in the comparison, these information can be ignored in power estimation. Once all the required data is specified in the power estimation tool, the dynamic power consumption of the circuit can be obtained.

Coarse-grained units (P_{veb})

To comply with Assumption 1 listed in Section 3.2.1, a UMC 0.13μm technology process and its associated ASIC macrocell library is used when modelling the dynamic power consumption of the coarse-grained unit. We believe this technology process is similar to the one used in Xilinx Virtex II device.

It would appear at first that dynamic power consumption of a coarse-grained unit can be determined by setting a constant toggle rate on all the nets in that unit. However, this is usually not the case as there may be some unused wordblocks where the input is always a constant and therefore the activity rate of these blocks is zero. Instead, we assume constant toggle rates on all *used* wordblocks and floating point operators, and zero toggle rate on all unused wordblocks. As a result, all unused wordblocks have zero dynamic power consumption. In an analogous manner, we also assume unused logic cells have zero dynamic power consumption in the commercial FPGA. Unused wordblocks can be identified from the routing configuration bits in the bitstream.

Similar to the fine-grained fabric, a 12.5% toggle rate is applied to all the nets in used wordblocks and floating point operators.

Output loading (P_r)

The output loading of the coarse-grained unit has to be considered when estimating the dynamic power consumption. Based on the calibration experiment as explained in Section 5.5, the output loading of each pin is found to be 4.5pF. This allows the ASIC tool chain to consider the dynamic power consumption of the coarse-grained unit logic with the associated routing resources to the fine-grained unit.

The total dynamic power consumption of FPFPGA can be obtained by the sum of dynamic power consumption from coarse-grained and fine-grained unit.

5.5 Results

Circuit	# of Add/Sub	# of Mul	Domain	Nature
bfly	4	4	DSP	kernel
dscg	2	4	DSP	kernel
fir	3	4	DSP	kernel
mm3	2	3	Linear algebra	kernel
ode	3	2	Linear algebra	kernel
bgm	9	11	Finance	application
syn2	5	4	N/A	synthetic
syn7	25	25	N/A	synthetic

Table 5.2: Benchmark circuits.

Eight benchmark circuits are used in this study as shown in Table 5.2. Five are computational kernels, one is a Monte Carlo simulation datapath, and two are synthetic circuits. All benchmark circuits involve single precision floating point operations. We choose these circuits since they are representative of the applications we envision being used on an FPFPGA. The application domains we chosen involve DSP and linear algebra. These kind of applications usually demand plenty of floating point operation. We also select one financial computation application to show that our architecture can implement a more complex system which can solve some real-life problems.

We note that the strong representation of simple floating point kernels that map directly to the coarse-grained units favourably influences the overall density and performance metrics so our results can be considered an upper bound. Dependencies, mapping, control and interfacing are issues likely to degrade performance. Applications, which do not require floating point operations or bus-based operations, would probably cannot exploit the coarse-grained units and these may cannot be implemented efficiently on FPFPGA. Such applications include, but not limited to, bit manipulation algorithms such as symmetric encryption, multimedia encoding and data compression.

The *bfly* benchmark performs the computation $z = y + x * w$ where the inputs and output are complex numbers; this is commonly used within a Fast Fourier Transform computation. The *dscg* circuit is the datapath of a digital sine-cosine generator. The *fir* circuit is a 4-tap finite impulse response filter. The *mm3* circuit performs a 3-by-3 matrix multiplication. The *ode* circuit solves an ordinary differential equation. The *bgm* circuit computes Monte Carlo simulations of interest rate model derivatives priced under the Brace, Gątarek and Musiela (BGM) model [Zhan 05]. All the wordlengths of the above circuits are 32 bit.

In addition, a synthetic benchmark circuit generator based on [Kund 04] is used. The generator can produce floating point circuits from a characterisation file describing circuit and cluster statistics. Two synthetic benchmark circuits are produced. Circuit *syn2* contains five floating point adders and four floating point multipliers. Circuit *syn7* contains 25 floating point adder and 25 floating point multipliers. The *syn7* circuit is considerably larger than the other benchmarks.

In this section, we present an evaluation of our architecture. The flow described in the previous section is employed.

The best-fit architecture can be determined by varying the parameters to produce a

Fabric	Area (A) (μm^2)	Feature Size (L) (μm)	Normalised Area (A/L^2)	Area (LC)	Input Pin	Output Pin
Virtex II LC	5,456	0.15	242,489	1	4(4)	2(2)
SP-CGU	498,847	0.13	30,203,964	122	157 (488)	162(244)
DP-CGU	1,025,624	0.13	60,687,797	251	285 (1004)	258(502)

Table 5.3: Normalisation on the area of the coarse-grained units against a Virtex II LC. SP and DP stand for single precision and double precision respectively. For the values shown in the second column (Area), 15% overheads have already been applied on the coarse-grained units.

design with maximum density over the benchmark circuits. Additional wordblocks are included, allowing more flexibility for implementing circuits outside of the benchmark set. Manual mappings are performed for each benchmark. A more in-depth analysis on how those parameters affect the application performance is on-going work.

For the single precision FPFPGA device, a Xilinx XC2V3000-6-FF1152 FPGA is used as the host and we assume 16 coarse-grained units. We emphasise that the parameter settings chosen for the coarse-grained block is fixed over the entire set of benchmarks, each coarse-grained unit having nine subblocks ($D = 9$), four input buses ($M = 4$), three output buses ($R = 3$), three feedback registers ($F = 3$), two floating point adders and two floating point multipliers ($P = 2$). We assume that the two floating point multipliers in the coarse-grained unit are located at the second and the sixth subblock. The two floating point adders are located in the third and the seventh subblock.

The coarse-grained blocks constitute 7% of the total area of an XC2V3000 device. All FPGA results are obtained using Synplify Premier 9.0 for synthesis and Xilinx ISE 9.2i design tools for place and route. All ASIC results are obtained using Synopsys Design Compiler V-2006.06.

The physical die area and photomicrograph of a Virtex II device has been reported [Yui 02], and the normalisation of the area of coarse-grained unit is estimated in Table 5.3. From inspection of the die photo, we estimate that 60% of the total die area is used for logic cells.

This means that the area of a Virtex II LC is 5,456μm^2. This number is normalised against the feature size (0.15μm). A similar calculation is used for the coarse-grained units. The ASIC synthesis tool reports that the area of a single precision coarse-grained unit is 433,780μm^2. We further assume 15% overhead after place and route the design based on our experience [Wilt 07]. The area values are normalised against the feature size (0.13μm). The number of equivalent logic cell is obtained through the division of coarse-grained unit area by slice area. This shows that single precision coarse-grained unit is equivalent to 122 LCs. Assuming each LC has two outputs, the VEB allow maximum of 244 output pins while the coarse-grained unit consumes 162 output pins only. Therefore, we do not need to further adjust the area.

Single precision FPFPGA results are shown in Table 5.4a and Figure 5.5a and 5.5b. A comparison between the floorplan of the Virtex II device and the floorplan of the FPFPGA on *bgm* circuit is illustrated in Figure 5.6.

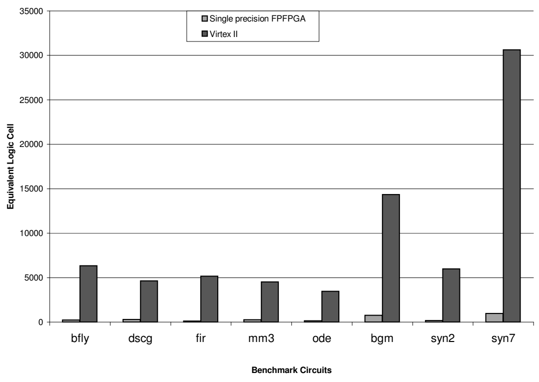
The FPU implementation on FPGA is based on the work in [Rudo 05]. This implementation supports denormalised floating point numbers which are required in the comparison with the FPFPGA. The FPU area for the XC2V3000 device (seventh column) is estimated from the distribution of LUTs, which is reported by the FPGA synthesis tool. The logic area (eighth column) is obtained by subtracting the FPU area from the total area reported by the place and route tool. As expected, FPU logic occupies most of the area, typically more than 90% of the user circuits. While the *syn7* circuit cannot fit in an XC2V3000 device, it can be tightly packed into a few coarse-grained blocks. The circuit *syn7* has 50 FPUs which consume 214% of the total FPGA area. They can fit into 16 coarse-grained units, which constitute just 6.8% of the total FPGA area.

Similar experiments for double precision floating point applications have been conducted and the results are reported in Table 5.4b, Figure 5.5c and Figure 5.5d. In double precision FPFPGA, we use the XC2V6000 FPGA as the host FPGA and the comparison is done on the same device.

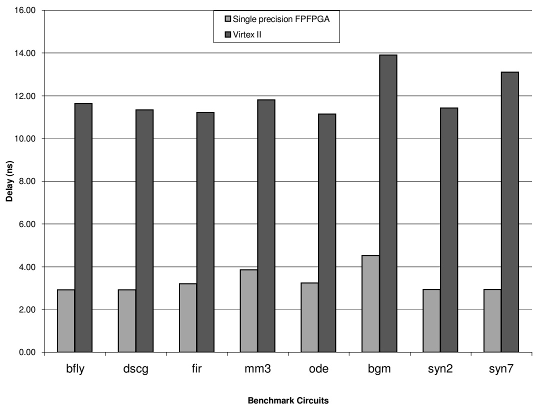
For both single and double precision benchmark circuits, the proposed architecture reduces the area by a factor of 25 on average, a significant reduction. The saving is achieved by (1) embedded floating point operators, (2) efficient directional routing and (3) sharing configuration bits. On larger circuits, or on circuits with a smaller ratio of floating point operations to random logic, the improvement will be less significant. However, the reported ratio gives an indication of the improvement possible if the architecture is well-matched to the target applications. In essence, our architecture stands between ASIC and FPGA implementation. The authors in [Kuon 07] suggest that the ratio of silicon area and delay required to implement circuits in FPGAs and ASICs is on average 35. Our proposed architecture can reduce the gap between FPGA and ASIC from 35 times to 1.4 times when floating point applications are implemented on such FPGAs.

The delay reduction is also significant. In our benchmark circuits, delay is reduced by 3.6 times on average for single precision applications and 4.3 times on average for double precision applications. We believe that double precision floating point implementation on commercial FPGA devices is not as effective as the single precision one. Therefore, the double precision FPFPGA offers better delay reduction than the single precision one. In our circuits, the critical path is always within the embedded floating point units, thus we would expect a ratio similar to that between normal FPGA and ASIC circuitry. Our results are consistent with [Kuon 07] which suggests the ratio is between 3 and 4. As the critical paths are in the FPU, improving the timing of the FPU through full-custom design would further increase the overall performance.

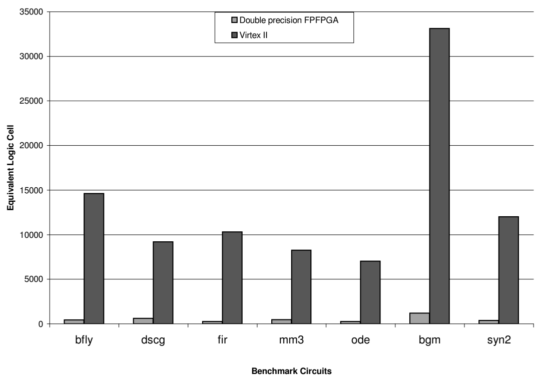
It is possible to allow more flexibility by replacing coarse-grained units with fine-grained ones. As an example, the *fir4* circuit requires 2 coarse-grained units and 4 slices. However, it can also be implemented using one coarse-grained unit and 1317 slices, while the delay is increased from 4.8ns to 9.1ns. By slightly increasing the number of slices by 100, this configuration allows us to implement the equation $\sqrt{x^2 + y^2 + z^2}$ which consists of 3 multipliers, 2 adders and a square root. Two additions and two multiplications can be implemented on a coarse-grained unit, while another multiplication and the square root operation can be implemented on fine-



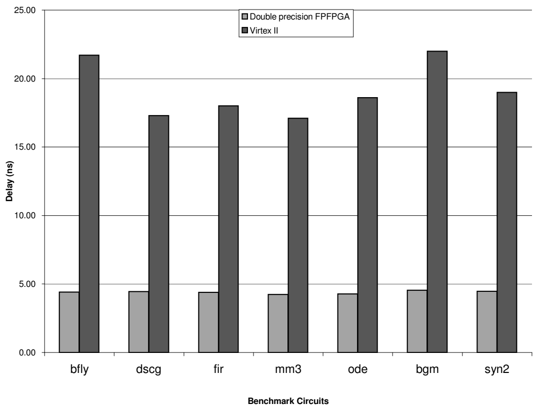
(a) Single precision – area.



(b) Single precision – delay.

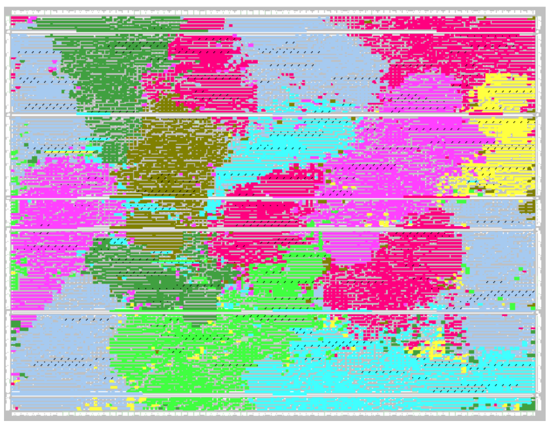


(c) Double precision – area.

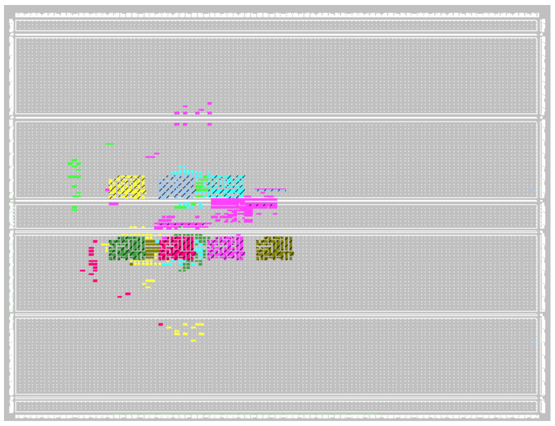


(d) Double precision – delay.

Figure 5.5: Comparisons of FPFPGA and Xilinx Virtex II FPGA device.



(a) Virtex II 3000. The circuit consumes 100% of chip area.



(b) FPFPGA. Coarse-grained units are identified by tightly packed logic cells in a rectangular region. The circuit consumes 5% of chip area.

Figure 5.6: Floorplan of the single precision *bgm* circuit on Virtex II FPGA and FPFPGA. Area is significantly reduced by introducing coarse-grained units.

grained units. The resulting circuit requires one coarse-grained unit and 1412 slices and the delay is 10.2ns. The alternative, with two coarse-grained units and 763 slices, has delay of 5.4ns.

In order to determine a suitable output loading of the coarse-grained unit, an embedded multiplier is implemented using ASIC macrocell flow which has similar architecture as the embedded multiplier used in Virtex II FPGA. The output loading of each pin on the embedded multiplier is adjusted to match the same dynamic power consumption as the one in Virtex II FPGA. We find that the output loading is 4.5pF. Therefore this value is used as the output loading of the coarse-grained unit.

Table 5.5 summarises the dynamic power consumption of single precision FPFPGA and Xilinx Virtex II FPGA. Our finding agrees with [Kuon 07] which suggests the dynamic power consumption ratio of FPGA to ASIC is around 12. The dynamic power consumption of the FPFPGA architecture stands between these ratios. Based on this observation, we are more confident in the proposed power estimation flow.

Since the FPFPGA can run higher frequency than the Virtex II FPGA, it is expected that for the same operation, the FPFPGA can complete faster. Their energy consumption is different as the elapsed time is different. Figure 5.7 illustrates the ratio in dynamic energy consumption between the FPFPGA and the Virtex II FPGA. The energy consumption for is determined by dynamic power consumption divided by the operating frequency. On average, floating point applications implemented on FPFPGA can reduce dynamic energy consumption by a factor of 14 compared to the Virtex II FPGA.

5.6 Comparison with Previous Work

Modelling of embedded FPUs based on island-style FPGAs has been reported in [Beau 08] and in Chapter 3. The improvements observed are summarised in Table 5.6. This work adopts a more generic model in which a number of FPUs and LUTs form a

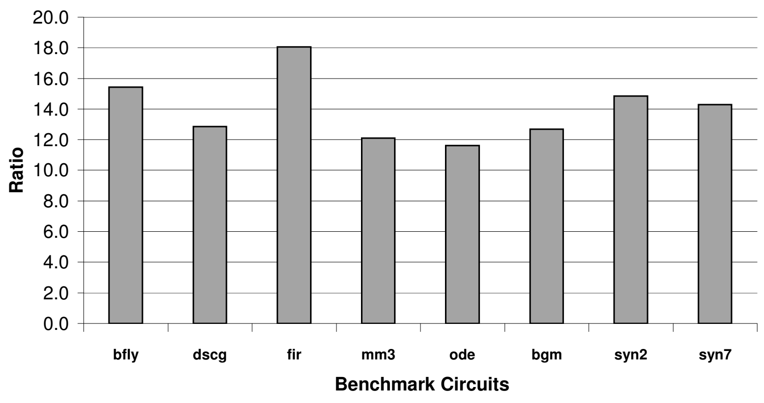


Figure 5.7: Dynamic energy consumption ratio of single precision FPFPGA.

coarse-grained unit to allow higher density and better speed. In addition, FPUs and LUTs are connected by reconfigurable bus-based routing to allow efficient datapath logic mapping. Area saving is given by (1) an improved coarse-grained unit, (2) efficient directional routing, (3) sharing configuration bits and (4) reduced functionality FPUs. The methodology, benchmarks, tools and assumed architecture of our approach compared with [Beau 08] are very different and a direct comparison cannot be made.

As similar circuits are used in Section 3.4 (embedded FPU design), a more detailed analysis is performed. Table 5.7 presents the area distribution and the reported delay between this work and the work in Chapter 3. One significant reduction is the size of FPU. In embedded FPU design, a full FPU which could perform not only addition and multiplication, but also division, square root and integer to floating point conversion is employed. In this version, the FPU can only perform addition or multiplication. Another significant reduction lies in the improved functionality of the coarse-grained unit which can implement bus-based logic, buffering and multiplexing operations. For embedded FPU design, such logic is implemented using fine-grained resources. The delay reported in this work is better than the embedded FPU design because of difference in pipeline stages. The embedded FPU design assumes one clock cycle latency while this work assumes five clock cycles. Therefore, some applications which require data dependency achieve better performance in the embedded FPU design.

5.7 Summary

We propose a customisable FPFPGA architecture which involves a combination of reconfigurable fine-grained and reconfigurable coarse-grained units optimised for floating point computations. A parameterisable description is presented which allows us to explore different configurations of this architecture. To provide a more accurate evaluation, we adopt a methodology for estimating the effects of introducing embedded blocks to commercial FPGA devices. The approach is vendor independent and offers a rapid evaluation of arbitrary embedded blocks in existing FPGA devices. Using this approach, we show that the proposed FPFPGA enjoys improved speed and density over a conventional FPGA for floating point intensive applications. The area can be reduced by 25 times, the frequency is increased by 4 times and dynamic energy consumption is reduced by 14 times on average when comparing the proposed architecture with an existing commercial FPGA device.

	Single precision FPFPGA					XC2V3000-6-FF1152				Reduction	
Circuit	number of CGU	CGU area (LC)	FGU area (LC)	Total Area (LC)	Delay (ns)	FPU area (LC)	Logic area (LC)	Total Area (LC)	Delay (ns)	Area (times)	Delay (times)
bfly	2	244 (0.9%)	212 (0.74%)	456 (1.6%)	2.92	11,678 (41%)	988 (3.4%)	12,666 (44%)	11.6	27.8	3.99
dscg	2	244 (0.9%)	352 (1.23%)	596 (2.1%)	2.92	8,838 (31%)	406 (1.4%)	9,244 (32%)	11.3	15.5	3.88
fir	2	244 (0.9%)	14 (0.05%)	258 (0.9%)	3.20	10,118 (35%)	218 (0.8%)	10,336 (36%)	11.2	40.1	3.51
mm3	2	244 (0.9%)	268 (0.93%)	512 (1.8%)	3.86	8,004 (28%)	1,010 (3.5%)	9,014 (31%)	11.8	17.6	3.06
ode	2	244 (0.9%)	38 (0.13%)	282 (1.0%)	3.24	6,658 (23%)	282 (1.0%)	6,942 (24%)	11.1	24.6	3.44
bgm	7	854 (3.0%)	646 (2.25%)	1,500 (5.2%)	4.52	27,856 (97%)	812 (2.8%)	28,668 (100%)	13.9	19.1	3.08
syn2	3	366 (1.3%)	0 (0.0%)	366 (1.3%)	2.93	11,966 (42%)	0 (0.0%)	11,966 (42%)	11.4	32.7	3.90
syn7*	16	1,952 (6.8%)	0 (0.0%)	1,952 (6.8%)	2.93	61,250 (214%)	0 (0.0%)	61,250 (214%)	13.1	31.4	4.47
									Geometric Mean:	24.9	3.64

(a) Single precision FPFPGA results. *Circuit *syn7* cannot be fitted in a XC2V3000-6 device. The area and the delay are obtained by implementing on a XC2V8000-5 device.

	Double precision FPFPGA					XC2V6000-6-FF1152				Reduction	
Circuit	number of CGU	CGU area (LC)	FGU area (LC)	Total Area (LC)	Delay (ns)	FPU area (LC)	Logic area (LC)	Total Area (LC)	Delay (ns)	Area (times)	Delay (times)
bfly	2	504 (0.7%)	402 (0.74%)	906 (1.3%)	4.42	27,306 (40%)	1,926 (2.9%)	29,232 (43%)	21.7	32.3	4.91
dscg	2	504 (0.7%)	726 (1.07%)	1,230 (1.8%)	4.45	17,968 (27%)	404 (0.6%)	18,372 (27%)	17.3	14.9	3.89
fir	2	504 (0.7%)	12 (0.02%)	516 (0.8%)	4.38	20,290 (30%)	330 (0.5%)	20,620 (31%)	18.0	40.0	4.11
mm3	2	504 (0.7%)	458 (0.68%)	962 (1.4%)	4.25	15,058 (22%)	1,454 (2.2%)	16,512 (24%)	17.1	17.2	4.03
ode	2	504 (0.7%)	44 (0.07%)	548 (0.8%)	4.27	13,588 (20%)	478 (0.7%)	14,066 (21%)	18.6	25.7	4.35
bgm	7	1,764 (2.6%)	642 (0.95%)	2406 (1.0%)	4.55	65,836 (97%)	398 (0.6%)	66,234 (98%)	22.0	27.5	4.84
syn2	3	756 (1.1%)	0 (0%)	756 (1.1%)	4.47	24,032 (36%)	0 (0%)	24,032 (36%)	19.0	31.8	4.26
									Geometric Mean:	25.7	4.33

(b) Double precision FPFPGA results. Circuit *syn7* is omitted since it cannot be fitted on any Virtex II FPGA device.

Table 5.4: FPFPGA implementation results. Values in the brackets indicate the percentages of logic cell used in corresponding FPGA device. CGU stands for coarse-grained unit and FGU stands for fine-grained unit.

Circuit	FPFPGA			XC2V3000-6		Ratio
	# of CGU	Power (mW)	Frequency (MHz)	Power (mW)	Frequency (MHz)	
bfly	2	204	343	791	86	3.9
dscg	2	185	343	609	88	3.3
fir	2	130	310	674	89	5.2
mm3	2	137	259	544	85	4.0
ode	2	135	309	458	90	3.4
bgm	7	398	221	1,806	79	4.5
syn2	3	204	341	781	88	3.8
syn7∗	16	1,084	342	3,441	76	3.2
					Geometric Mean:	3.9

Table 5.5: Power estimations. ∗Circuit *syn7* cannot fit in a XC2V3000-6 FPGA so the power number of FPGA implementation is obtained from a XC2V8000-5 FPGA.

	Floating point FPGA	Embedded FPU (Chapter 3)	Beauchamp [Beau 08]
Area Reduction	25	2.75	2.21
Speedup	4.0	5.7	1.33
Data format	Single/double precision	Double precision	Double precision
Modelling flow	Standard macrocell, VEB	Estimation based on Blue gene, VEB	Estimation based on Pentium 4, VPR
Floating point operator	Multiplier, adder with reconfigurable interconnect and wordblocks	FPU based on Blue Gene	Multiply-and-add with reconfigurable interconnect

Table 5.6: Comparison of floating point reconfigurable fabric. Area reduction and speedup are compared to an FPGA device with embedded multiplier.

	This work			Embedded FPU		
	CGU area (LC)	FGU area (LC)	Delay (ns)	FPU area (LC)	FGU area (LC)	Delay (ns)
bfly	504	402	4.42	7,328	4,490	7.62
dscg	504	726	4.45	5,496	1,192	7.34
fir4	504	12	4.38	6,412	1,442	7.32
mm3	504	458	4.25	4,580	3,600	7.58
ode	504	44	4.27	4,580	1,224	7.62

Table 5.7: Comparison to previous embedded FPU model for double precision floating point benchmarks.

Chapter 6

CAD Tools for Floating Point FPGA

6.1 Introduction

While we have studied the performance of architecture of FPFPGA in Chapter 5, all the benchmark circuits are mapped to the FPFPGA manually. Although such mapping can usually produce high quality results on small circuits, the process is tedious and error-prone when applications become larger. The motivation of this chapter is to introduce various computer aid design (CAD) tools to address the issue of mapping applications into FPFPGA, especially for the coarse-grained units in FPFPGA.

A number of CAD tools have been employed to design and model FPFPGA devices. However, many of them can be categorised as electronic design automation (EDA) tools which offer physical level or low level automation. For instance, Chapter 3 employs standard FPGA design tools to model FPFPGA, and Chapter 4 and 5 employ ASIC design tools to develop datapath FPGA fabric and coarse-grained units.

Since FPFPGA architecture shares the same fine-grained units and routing architecture as standard FPGA devices, most EDA tools for standard FPGA devices, including synthesis tools, place and route tools, timing analysis tools and even power analysis tools can be reused on FPFPGA. However, the tools do not support user-defined coarse-grained units in the FPFPGA, and consequently, mapping circuits into coarse-grained units has to be done manually. This chapter focuses on high level aspects of designing applications on FPFPGA, in other words, how to “program” FPFPGA, especially for the coarse-grained units effectively.

A standard FPGA design flow usually begins with VHDL or Verilog description of a circuit. The description can either be a structural or behavioural one. A synthesis tool, after processing such descriptions, can produce a physical netlist of target FPGA and the netlist is ready for place and route. During the process, the tool parses the descriptions, translates them to Boolean logic, decomposes the logic into a set of Boolean equations, associates each Boolean equation into LUTs, and produces a netlist which contains information about the connection and configuration of each occupied LUT. The place and route tool, which takes the netlist as input, places the LUTs to suitable locations on the FPGA devices, and perform routing and timing analysis sequentially. After place and route, the bitstream generator produces a bitstream which contains location and configuration of every LUT, connection box and routing switch of the target FPGA. The bitstream represents the application circuit implemented on the FPGA device and can be downloaded to the FPGA to perform required functions.

This design flow, however, can usually be applied to FPGA with fine-grained units only. Although most standard FPGA devices consist of heterogeneous components such as multipliers or block memory, the design flow has limited support for them. For instance, block memory can be instantiated either explicitly specified in structural descriptions or a unique representation of behavioural descriptions. However, when heterogeneous components become more flexible and reconfigurable such as CGUs in FPFPGAs, users have to explicitly instantiate the components, connect the components to other fine-grained logic and configure the components, and traditional flow has no support for these.

High level synthesis employs another approach to translate applications into circuits implemented on FPGA devices. Instead of using HDL, the design entry is variation of high level language such as C or Perl. Handel-C [Agil 07] is one of the typical examples in this category. The advantage of this approach is that application designers do not require to fully understand the architecture of the FPGA in order to utilise the performance offered by FPGA effectively. In addition, this approach can capture certain coarse-grained operations for further optimisation and it is usually difficult to discover when the same applications are represented in structural HDL. For instance, it is easy to extract floating point multiplication from a C description by looking for a "$*$" operator. On the contrary, floating point multiplication represented by thousand of Boolean equations is obscured and difficult to extract.

A hardware compiler does not only reduce the effort of application designers, it is also beneficial to the development of the FPFPGA itself. When searching for suitable FPFPGA architecture by exploration, for example, one may want to explore the trade-off when more floating point multipliers are included in the coarse-grained unit, it is inevitable to write benchmark circuits for each set of architecture since benchmark circuits represented as HDL are architecture-specific. It is desirable to have an architecture-aware hardware compiler, which can produce a netlist for each benchmark application based on different architectural parameters of an FPFPGA. Architecture designers do not need to rewrite the benchmark when the FPFPGA architecture is changed. Rather, the designers just need to compile the benchmark applications written in high level descriptions with another set of architectural parameters to obtain benchmark circuits for evaluation.

In this work, we propose an approach to translate floating point applications into circuits implemented on FPFPGA device by adopting high level synthesis approach and employing high level language as the design entry. In particular, we demonstrate a technology mapper which translates a dataflow graph into a FPFPGA netlist in VHDL, in which the dataflow graph can be obtained by processing high level descriptions in other high level synthesis tool such as Trident [Trip 07] or the fly compiler [Ho 02b].

The chapter is organised as follows. Section 6.2 suggests the requirements of a desired compiler for FPFPGA devices. Section 6.3 illustrates the algorithm of the technology mapper. Section 6.4 discusses the integration into existing high level synthesis tools and results are presented in Section 6.5. Section 6.6 provides concluding remarks.

6.2 Requirements

The benchmarks in previous chapters are created by translating circuits from high level descriptions to structural descriptions manually. The mapping process, while it is tedious, helps us to identify the requirements of high level synthesis tool for FPFPGA as listed below.

1. The compiler should contain a set of pre-defined built-in functions which represent the functionality in the coarse-grained unit. For example, the compiler can provide floating point functions such as `fadd()`, `fmul()` (or even better, overloaded operators such as "+" or "*") which associate with the floating operators in the coarse-grained unit. This feature allows application designers to infer the coarse-grained units easily.
2. It should have the ability to differentiate the control logic and the datapath. This feature would allow the technology mapper to handle the control logic and the datapath separately. Since the control logic can be efficiently implemented using the fine-grained logic, a standard hardware compilation technique such as [Page 91] can be used. The datapath, which is usually much more complicated, can be mapped to coarse-grained units whenever it is possible.
3. The compiler should contain an architecture-aware technology mapper for the coarse-grained architecture. Since the FPFPGA architecture is customisable according to the applications domain, the corresponding technology mapper should map to devices with differing amounts of coarse-grained resources. For example, the technology mapper should be aware of the number of floating

point operator in a coarse-grained unit so it can fully utilise all the operators in a unit. This feature would allow FPGA designers to evaluate new architectures effectively by compiling benchmark circuits with modified architectural parameters.

4. The compiler should contain an intelligent resource allocation algorithm. It should be aware of the functionality of the coarse-grained unit and decide if the given operation is best implemented by coarse-grained units or fine-grained units. For example, if the compiler receives a “square root” instruction but there is no square root function in the coarse-grained units, the allocation algorithm can infer a square root operator using fine-grained unit instead.

5. Support is required for bitstream generation for coarse-grained units. Such a feature is necessary to determine the delay of a mapped coarse-grained unit.

It should be noted that some requirements, such as Requirement 1, have been studied in other contexts [Agil 08, Ho 02b], and Requirement 2 has been addressed in [Trip 07] in which the authors propose a compiler that can produce separate circuits for control logic and datapath for floating point applications. Requirement 3, 4 and 5 are new, and are specific to our architecture, and they are addressed in Section 6.3.

We decide not to develop a high level synthesis tool from scratch. Instead, we focus on components dedicated to FPFPGA such as the technology mapper and the bitstream generator. There are several reasons for this. First of all, because FPFPGA employs the same fine-grained units and fine-grained routing architecture as standard FPGA devices, most existing high level synthesis technology can still be applicable to FPFPGAs. For instance, standard components appearing in most high level synthesis tool such as front-end (parsing modules such as lexical analysis, abstract syntax tree construction), middle-end (optimisation modules such as static analysis, loop unrolling, pipelining, resource scheduling, dataflow graph generation) and back-end (code generation modules such as datapath construction, state machine construction, bitstream generation) can be reused and optimisations from those tools are still valid in FPFPGA in most cases. With minor modification, those tools can capture floating point operations provided that the corresponding high level language supports floating point operations natively.

Most high level design tools involve a datapath generation procedure. The procedure usually accepts a dataflow graph as an intermediate representation of the application and converts the dataflow graphs into corresponding components in FPGA. The dataflow graph is usually the kernel of the application such as inner loops. By manipulating the intermediate representation, we can integrate the technology mapper into existing high level synthesis tool.

6.3 Technology mapper

6.3.1 Overview

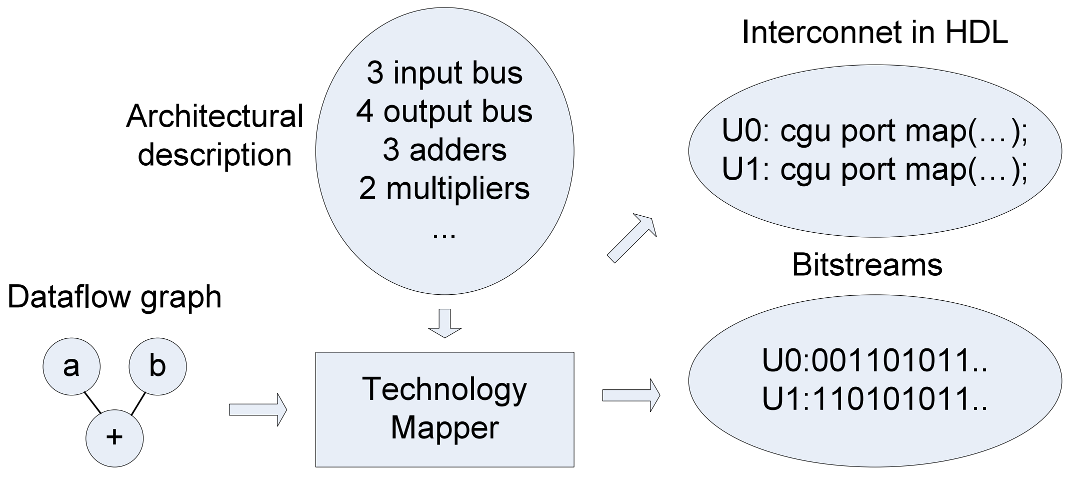


Figure 6.1: Logic flow of the technology mapper.

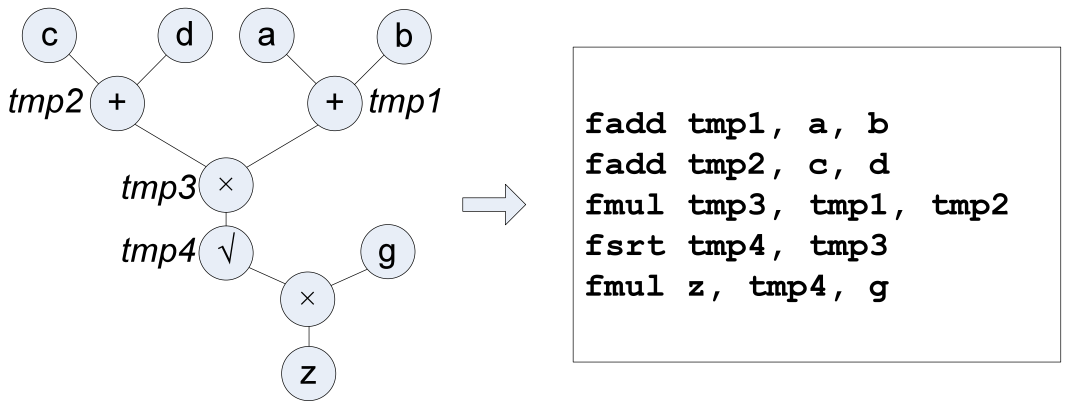
Figure 6.1 illustrates the logic flow. The mapper requires a specification of the coarse-grained unit to map floating point applications onto it. As coarse-grained units adopt parameterised design flow, the technology mapper has to consider the architectural description of the coarse-grained unit before mapping the application circuits. The architectural description includes type and number of floating point operators, number of input and output buses, number of feedback registers and the number of generic wordblocks. The descriptions also include the placement sequence of the generic wordblocks and the floating point operators as they may affect the routing configuration during mapping. Once the architecture of the coarse-grained unit is defined in the mapper, the number of the coarse-grained units in the FPFPGA has to be specified as this is a resource constraint.

Because of the customisable nature of FPFPGA, it should be noted that an FPFPGA allows different combination of coarse-grained unit. As a result, the mapper allows more than one type of coarse-grained unit. For example, the mapper allows an FPFPGA to have 3 type-A coarse-grained units, where each type-A coarse-grained unit has 4 floating point adders and 3 type-B coarse-grained units, and each type-B coarse-grained unit has 4 floating point multipliers in the mapper.

After the architecture of the FPFPGA is specified, the mapper can take a dataflow graph as an input and produce a mapped design on the FPFPGA. The mapped design consists of placed and routed configuration in every instantiated coarse-grained unit, configuration of the computation units implemented by fine-grained units, and the interconnection between them. The outputs of the mapper are a VHDL description which describes how the coarse-grained units and soft-cores (circuits implemented on fine-grained units) are connected and bitstreams for every coarse-grained unit and how they should be configured.

Each node on the dataflow graph represents a floating point operation in an application. The functionality of the node is not limited by the architecture of the coarse-grained unit. For example, a node can represent a square root operation but the coarse-grained units may not be able to support this. The dataflow graph for the mapper is represented as an assembly-language-like format. For example, the formula $z = \sqrt{a + b \times c + d} \times g$ can be expressed as a dataflow graph in Figure 6.2.

The mapping algorithm of the mapper is based on a greedy search. While this may



```
fadd tmp1, a, b
fadd tmp2, c, d
fmul tmp3, tmp1, tmp2
fsrt tmp4, tmp3
fmul z, tmp4, g
```

Figure 6.2: Sample dataflow graph and its representation.

not produce optimal solution, we find the quality of mapping results is acceptable. It is elaborated in detail in Section 6.5.

The mapper is configured for area-saving and optimises the density of nodes mapped to a coarse-grained unit. The mapper processes nodes sequentially from the text input. When we map the node to a coarse-grained unit, we consider the following conditions:

1. Location of input – the mapper tries to map the node to the same coarse-grained unit of the input block.

2. Functionality of the coarse-grained units – the mapper may instantiate soft-cores outside coarse-grained units if none of them support the operation.

3. Availability of the coarse-grained units – the mapper may instantiate soft-cores outside coarse-grained units if all the coarse-grained units are used up.

4. Number of input bus of the coarse-grained unit – if there is no input available for the coarse-grained unit, the mapper may instantiate new coarse-grained unit for new operation.

5. The location of the floating point operators – feedback registers in the coarse-grained unit may be inferred if the operation requires feedback path.

6. Number of output bus of the coarse-grained unit – if the output bus is used up, some nets cannot be routed to other coarse-grained unit. This may result in unroutable design and backtracking is required to resolve this issue.

The algorithm presented in Section 6.3.2 addresses all the conditions listed above.

6.3.2 Algorithm

The technology mapper employs object oriented design approach. Several objects are involved when designing the algorithms. In particular, `wordblock` represents a wordblock in a coarse-grained unit. Attributes of wordblock include a name, which represents the output net name of the wordblock, functionality (such as generic word-block, multiplier or adder), configurations (such as registered output), an occupied flag to identify if the block is occupied, and input net names. `feedback` represents a feedback register. It has three attributes including a name, an input net name and an occupied flag.

A `coarse-grained unit` is another object which represents the configuration of a coarse-grained unit. It contains a list of wordblocks, a list of feedback registers, a list of input net names and a list of output net names. One can define its own FPFPGA for technology mapping by creating instances of coarse-grained unit with different configuration of wordblocks. The technology mapper, which processes a list of coarse-grained unit instances and the dataflow graph, generates netlists which contain configured coarse-grained unit instances and the routing between them. The configured coarse-grained unit instances can then be used for HDL and bitstream generation. The following algorithms are used in the technology mapper.

- `searchnet` (Listing 3)

  – input: a net name, a list of coarse-grained units

  – output: index of a coarse-grained unit, index of a wordblock

– functionality: locates a wordblock in a specific coarse-grained unit which contains the given net name

- `cgumap` (Listing 4)

  – input: a node, a specific coarse-grained unit

  – output: a status code which indicates if the operation success

  – functionality: attempts to map the operation specified in the node to coarse-grained unit

- `cgugenericmap` (Listing 5)

  – input: a node, a list of coarse-grained unit

  – output: a status code which indicates if the operation success

  – functionality: attempts to map the operation specified in one of an available coarse-grained unit

- `addunary, addbinary, addternary` (Listing 6)

  – input: a node, a list of coarse-grained unit

  – output: a status code which indicates if the operation success

  – functionality: attempts to add the operation specific in the node to coarse-grained unit when locations of the operations are considered. `addunary`, `addbinary` and `addternary` assumes one, two and three input operands respectively.

- `techmap` (Listing 7)

  – input: direct cyclic graph, a list of coarse-grained units

  – output: a list of configured coarse-grained unit, and a direct cyclic graph

  – functionality: map the dataflow graph represented in input into both coarse-grained units and fine-grained units

The algorithms associated to the functions are shown in Listing 3-7. The function `searchnet` is a helper function which returns the location of a net specified in the input argument. It is done by scanning all the wordblocks in every coarse-grained unit.

Listing 3: searchnet

```
Data: net name ($net$), list of coarse grained units ($cgus$)
Result: ($d$, $b$) in which the net is located at coarse-grained unit $d$, wordblock $b$
foreach coarse-grained unit (c) in ($cgus$) do
    if $c$ contains $net$ then
        $d \Leftarrow$ index of $c$
        $b \Leftarrow$ index of corresponding wordblock contains $n$
        return ($d$,$b$)
    end
end
return (ERROR_NOT_FOUND, ERROR_NOT_FOUND)
```

Listing 4: cgumap

```
Data: $k$-input node ($n$), coarse grained unit ($c$)
Result: operation represented in node $n$ is mapped to $c$ and return status
$input\_list \Leftarrow$ list of input net name of $n$
foreach wordblock ($wb$) in ($c$) do
    if $wb$ is not occupied and $wb$ can implement $n$ then
        name of $wb \Leftarrow$ name of $n$
        configure $wb$ to implement $n$ with registered output
        flag $wb$ as occupied
        foreach net name ($net$) in ($input\_list$) do
            ($p$,$q$) $\Leftarrow$ searchnet($net$, $c$)
            if $b$ = ERROR_NOT_FOUND then
                if $c$ has no available input then
                    route all net in $input\_list$ to output
                    return ERROR_NOT_ENOUGH_INPUT
                else
                    set available input of $c$ to $net$
                end
            else if wordblock $q$ precedes $wb$ then
                if $c$ has no available feedback register then
                    route $net$ to output
                    return ERROR_NOT_ENOUGH_FEEDBACK
                end
            end
            configure wordblock $q$ as unregistered output
            rename wordblock $q$ to ($net$ "comb")
            find an unoccupied feedback register
            configure the feedback register with input $net$ "comb" and output
            $net$
        end
    end
    return SUCCESS
end
return ERROR_NO_SUITABLE_WORDBLOCK
```

Listing 5: cgugenericmap

```
Data: input node ($n$), a list of coarse-grained units ($cgus$)
Result: add $n$ to the first coarse-grained unit in $cgus$ that can implement $n$ or
        ERROR
foreach coarse-grained unit (c) in ($cgus$) do
    $status \Leftarrow$ cgumap($n$, $c$)
    if $status = SUCCESS$ then
        return SUCCESS
    end
end
return ERROR_NO_SUITABLE_WORDBLOCK
```

The function `cgumap` is to map a node to a specific coarse-grained unit. First the function tries to identify an unused wordblock in the coarse-grained unit which can implement the function specified in the node. If a suitable wordblock is found, it is configured as necessary function and corresponding inputs are routed into that wordblock. If an input comes from another wordblock in the same coarse-grained unit, it is routed internally. Because of the uni-directional nature of coarse-grained unit, if the input nets precede to the suitable wordblock, feedback registers are inferred such that the input signal can route back to the wordblock (Condition 6). If no suitable wordblock is found, the function assumes the input is from outside and annotates the input port of coarse-grained unit as the same name as the input name of the node.

If the mapper finds a suitable wordblock and maps the node without any error, it returns SUCCESS. If the mapper cannot find a suitable wordblock to map the node, it returns ERROR_NO_SUITABLE_WORDBLOCK, indicating the mapping is failed. There are two other special cases. If feedback registers need to be inferred but none of them are available, it results in error and returns ERROR_NOT_ENOUGH_FEEDBACK. When the mapper assumes inputs come from input port but no input port is available, the mapper returns ERROR_NOT_ENOUGH_INPUT. In both cases, the mapping for this coarse-grained unit is aborted and the mapper routes required internal signals (if any) to output ports.

The function `cgugenericmap` is to map a node to a list of coarse-grained unit. It

Listing 6: addbinary

```
Data: 2-input node ($n$), a list of coarse-grained units ($cgus$)
Result: configured $cgus$ with the operation represented in $n$, or return ERROR
$in0 \Leftarrow$ first input of $n$
$in1 \Leftarrow$ second input of $n$
$(cguid0, b) \Leftarrow$ searchnet($in0$, $cgus$)
$(cguid1, b) \Leftarrow$ searchnet($in1$, $cgus$)
add $in0$, $in1$ to global input list if they are not found in $cgus$
if both $in0$ and $in1$ are not found in $cgus$ then
    return cgugenericmap($n$, $cgus$)
end
if $in0$ and $in1$ are in the same coarse-grained unit then
    $c \Leftarrow$ cgus[$cguid0$]
    $status \Leftarrow$ cgumap($n$, $c$)
    if $status \neq SUCCESS$ then
        set first available output of $c$ to $in0$
        set second available output of $c$ to $in1$
        if no available output for $in0$ or $in1$ then
            return ERROR_NOT_ENOUGH_OUTPUT
        end
        return cgugenericmap($n$, $cgus$);
    end
end
if $in0$ are found in $cgus$ then
    $c0 = cgus[cguid0]$
    if cgumap($c0$, $n$) $= SUCCESS$ then
        return SUCCESS
    end
    if no available output for $in0$ then
        return ERROR_NOT_ENOUGH_OUTPUT
    end
    set first available output of $c0$ to $in0$
    return cgugenericmap($n$, $cgus$)
end
if $in1$ are found in $cgus$ then
    $c1 = cgus[cguid1]$
    if cgumap($c1$, $n$) $= SUCCESS$ then
        return SUCCESS
    end
    if no available output for $in1$ then
        return ERROR_NOT_ENOUGH_OUTPUT
    end
    set first available output of $c1$ to $in1$
    return cgugenericmap($n$, $cgus$)
end
```

Listing 7: techmap

```
Data: direct cyclic graph $G$, a list of coarse-grained unit $cgus$
Result: $cgus$ represents subset of coarse-grained mapping of $G$ and direct cyclic
        graph $g$ represents fine-grained mapping of $G$ or return ERROR
foreach node (n) in graph (G) do
    $k \Leftarrow$ number of input in $n$
    switch $k$ do
        case 1
            $status \Leftarrow$ addunary($n$, $cgus$)
        case 2
            $status \Leftarrow$ addbinary($n$, $cgus$)
        case 3
            $status \Leftarrow$ addternary($n$, $cgus$)
        otherwise
            return ERROR_INPUT
        end
    end
    if $status \neq$ SUCCESS then
        if $status \neq$ ERROR_NOT_ENOUGH_OUTPUT then
            add $n$ to graph $g$
        else
            return $status$
        end
    end
end
return SUCCESS;
```

invokes cgumap iteratively on each coarse-grained unit until a suitable coarse-grained unit is matched. The function is invoked when there is no better mapping available.

The functions addunary, addbinary, addternary are very similar to each other so only addbinary is shown in Listing 6 . This function is to map a node to a list of coarse-grained unit while the *position* is taken into account. In addbinary, it first searches for coarse-grained unit which contains the associated input net. Since the function assumes the input node has two inputs, one of the following four cases will happen. A two-input node $a_1 \Leftarrow a_2\ fop\ a_3$ is used as an example to explain the situation here.

1. Both inputs (a_2, a_3) are not found in all the coarse-grained unit – We assumed the inputs are all from fine-grained unit so the algorithm adds the input to the global input list. It then calls cgugenericmap to perform mapping.

2. One of the input nets (e.g. a_2) is found in a coarse-grained unit – The input net which is not in the coarse-grained units (a_3) is added to the global input list. The function then tries to map the node to the coarse-grained unit which contains another input net (a_2). If it fails, the function routes the net out of the coarse-grained unit and annotates the output of that coarse-grained unit as the same name as net a_2. Generic mapping cgugenericmap is then performed.

3. Both inputs are found in the same coarse-grained unit – The function first tries to map the node in that coarse-grained unit and internal routing is enforced. If it is not, both inputs are then routed to the output and generic mapping cgugenericmap is performed.

4. Input nets are found in different coarse-grained units. One net is routed out of a coarse-grained unit and mapping is performed on another coarse-grained unit. If it fails, both nets are routed to output and generic mapping cgugenericmap is performed.

The functions `addunary` and `addternary` employ the same logic flow as `addbinary` to perform position-aware mapping. A coarse-grained unit which contains more inputs net has higher priority to perform mapping.

The function `techmap` is the top level function. `techmap` calls `addunary` or similar functions on every node on the graph. In case the `addunary` or similar functions return ERROR, two scenarios may happen. ERROR_NO_SUITABLE_BLOCK is returned when the mapper cannot find suitable coarse-grained unit for the node. It can be solved by implementing the node in fine-grained unit. ERROR_NOT_ENOUGH_OUTPUT is returned when the mapper cannot route the signal from coarse-grained unit to output because all the output bus in that coarse-grained unit is occupied. This error is critical as it means the net cannot be used outside the coarse-grained unit. It can only be solved by backtracking. It can be applied to the unroutable net to trace how the signal is originally formulated and certain operations are duplicated on a new coarse-grained unit to reproduce the signal.

The coarse-grained unit designer can also consider increasing the number of output of coarse-grained units. Redesign coarse-grained units by adding more floating point operators or more output port can help to resolve both issues.

6.3.3 Bitstream Generator

The technology mapper produces two outputs. They are (1) interconnection of each instantiated coarse-grained units and (2) configuration of each instantiated coarse-grained units. Interconnection is represented in VHDL so that the description can work with VEB flow and other high level synthesis tools. Configuration, however, is described in Perl data structure to allow bitstream generator, which is written in Perl, interpreting the configuration and generating the bitstream. An example of configuration is given below:

Configuration of a CGU represented in Perl data structure.

```
$s[0] = {
        type => "wb",
        sh => "no shift",
        reg => "reg",
        lut0 => "0000000000000000",
        lut1 => "0000000000000000",
        mux0 => "select A",      #select A, select control
        mux1 => "select carry",  #select carry, select control
        input0 => "feedback_bus_0",
        input1 => "feedback_bus_1",
        input2 => "feedback_bus_2",
};
...
```

$s is an array of wordblocks. Each element in $s represents a configuration of each wordblock. The bitstream generator parses the elements in the array iteratively and produces a bitstream which corresponds to the ASIC implementation of the coarse-grained units. While a bitstream is a long run of '0' and '1', the bitstream generator produces Synopsys Tcl script which can readily be used in timing analysis of the coarse-grained units as mentioned in Section 5.4.1. An example of output given by the bitstream generator is shown below:

Partial bitstream of a CGU represented in Synopsys Tcl script.

```
...
# bit 32 : wb0: reg reg
set_case_analysis 1 u2000/config_reg[ 32 ]
# bit 33 : wb0: mux0 select A
set_case_analysis 0 u2000/config_reg[ 33 ]
# bit 34 : wb0: mux1 select control
set_case_analysis 1 u2000/config_reg[ 34 ]
...
```

6.3.4 Example

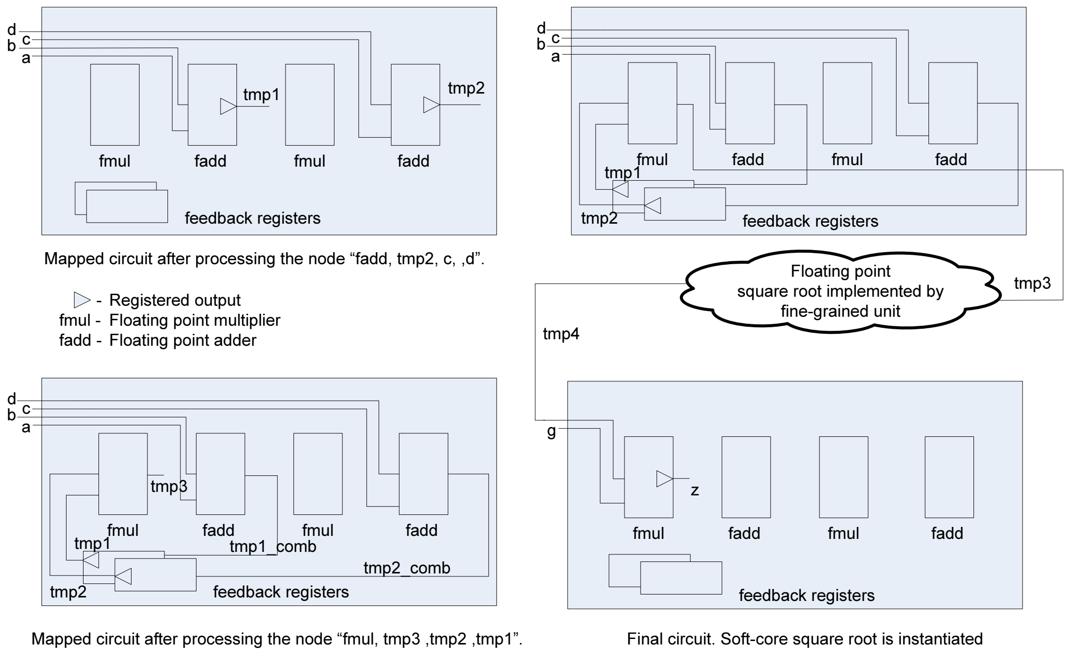

d
c
b
a
tmp1
tmp2
fmul
fadd
fmul
fadd
feedback registers
Mapped circuit after processing the node “fadd, tmp2, c, ,d”.
▷ - Registered output
fmul - Floating point multiplier
fadd - Floating point adder
tmp3
tmp1
tmp2
tmp1_comb
tmp2_comb
Mapped circuit after processing the node “fmul, tmp3 ,tmp2 ,tmp1”.
Floating point square root implemented by fine-grained unit
tmp4
g
z
Final circuit. Soft-core square root is instantiated

Figure 6.3: Mapping of equation $z = \sqrt{a + b \times c + d} \times g$.

As an example, Figure 6.3 presents how the formula $z = \sqrt{a + b \times c + d} \times g$ is mapped to the FPFPGA.

We assume there are two coarse-grained units in an FPFPGA and each coarse-grained unit has 2 floating point adders and 2 floating point multipliers. The floating point adders are located at second and forth column and floating point multipliers are located at first and third column. Each coarse-grained unit has 4 input bus, 3 output bus and 2 feedback registers.

Initially, the mapper receives the node “`fadd tmp1, a, b`”. It finds the first available coarse-grained unit which supports floating point addition and instantiate it. Then the mapper connects net “a” and net “b” from input to the first available floating point adder and annotates that adder as “tmp1”. The adder is configured as registered output. Similar case happens at the next node “`fadd tmp2, c, d`”. The mapper instantiates second floating point adder in the same coarse-grained unit and routes net “c” and “d” from inputs to that adder. The adder is annotated as “tmp2” and is output-registered.

The next node is "`fmul tmp3, tmp1, tmp2`". Although all input buses has been used in the first coarse-grained unit, the net "tmp1" and "tmp2" can both be found in the first coarse-grained unit and at the same time floating point multiplier is available. Condition 1 applies and the mapper instantiates the first available floating point multiplier in the same coarse-grained unit. The mapper finds that "tmp1" and "tmp2" are preceded to "tmp3" so feedback paths are required. Condition 5 applies and a feedback path is formulated. In this case, the floating point adders are configured as unregistered output and are renamed to "tmp1_comb" and "tmp2_comb" respectively. The outputs of the floating point adders are routed to available feedback registers which are then annotated as "tmp1" and "tmp2". After such modification, mapper can route "tmp1" and "tmp2" to "tmp3".

When mapping the node "`fsrt tmp4, tmp3`", the mapper finds that none of the coarse-grained unit supports square root operation. So the mapper decides to instantiate soft-core to handle this. It first routes the net "tmp3" to the output of the first coarse-grained unit. It then instantiates a soft-core which can perform square root operation and connects the output of coarse-grained unit to the input of the soft-core. Finally, the output of the soft-core is annotated as "tmp4".

The last node is "`fmul z, tmp5, g`". Although the first coarse-grained unit still has an unused floating point multiplier, the input-bus is used up so condition 4 applies and second coarse-grained unit is instantiated. The mapper routes net "g" and "tmp3" to the coarse-grained unit and the first available floating point multiplier is instantiated and its output is annotated as "z".

6.4 Integration

This section studies various high level synthesis tools and discusses the opportunities to incorporate our FPFPGA technology mapper to them. Although there are plenty of high level synthesis tools, two hardware compilers are assessed. They are Trident [Trip 07] and the fly compiler [Ho 02b]. We choose them because of two reasons. First, all of them support floating point operators natively. The semantics of the languages supported by the tools allows variables can be implicitly or explicitly declared as floating point data. The authors of these tools have considered supporting floating point operations initially. Second, we have access to the source code of those compilers, allowing us to study the algorithm in depth.

It should be noted that the integration technique discussed in this section can generally be applicable to other high level synthesis tools who wish to support FPFPGA. The integration process involves two steps. The first step is to retrieve a dataflow graph from the compiler so the graph can be read by the technology mapper. This process occurs just before the datapath generation of the original flow. The second step is to integrate the datapath generated by the technology mapper into original datapath and control circuits.

6.4.1 Trident

Trident [Trip 07] is a compiler dedicated to designing floating point algorithms on FPGA. Using C language as design entry, Trident can exploit parallelism available in the input description. Trident also applies conventional compiler optimisation and scheduling techniques. Since the open source nature of Trident, we can study the algorithms and integrate the technology mapper with minor effort.

As we mention before, the first step is to discover the dataflow graph and map the graph to coarse-grained units. Trident has its own internal intermediate representation, yet the representation uses internally and does not store any intermediate representation to persistent media. In order to capture the dataflow graph in Trident, we modify the datapath generator routines in Trident.

Since Trident is written in Java language, the technology mapper is written in Java to achieve better integration. Although hundreds of class have defined in Trident, we are only interested in those classes resided in `fp.synthesis` package, in which the package contains methods dedicated to datapath generation. In particular, `DataPathCircuitGenerator` is a class which contains an instance method `generate()` to interpret dataflow graphs represented as a series of input arguments.

We modify the instance method `generate()` so we can gather the dataflow graph by intercepting the argument appearing in the method. We are interested in the following nodes that can be implemented in coarse-grained units.

1. `Load` – A node represents an input of the dataflow graph. The technology mapper adds the net name of the node to the global input list.
2. `Store` – A node represents an output of the dataflow graph. The technology mapper adds the net name of the node to the global output list.
3. `Unary` – An 1-input node represents a 1-operand generic operation, such as square root or reciprocal operations. `Unary` node can be mapped to a specialised block or fine-grained units.
4. `Binary` – A 2-input node represents a 2-operand generic operation, such as addition or multiplication. `Binary` node can be mapped to a specialised block or fine-grained units.
5. `Test` – A 2-input node represents a 2-operand comparison operation, such as "equal to" or "less than" operations. `Test` node can usually be implemented as a fixed point subtraction using a generic wordblock.

Once the dataflow graph is recovered, we can apply the algorithm described in Section 6.3.2 to implement datapath into coarse-grained units.

The second step is to modify the compiler which generates correct control signal to the coarse-grained units. Fortunately, Trident supports third-party floating point operators with different latency, area and speed. It is done by reading a template file describing the latency, area and speed of the floating point operators. By modifying this file, Trident can recognise the latency of each floating point operation in the coarse-grained unit and produces the correct control signals.

Once Trident generates VHDL descriptions for both datapath and control signal circuits, users are required to manually connect the entity from the technology mapper to the control circuit entity in Trident using `port map` keyword. This process has not been automated yet.

6.4.2 The fly compiler

The fly compiler [Ho 02b] is a compiler dedicated to rapid system prototyping research. It emphasises on modifiability by using relatively simple design – the whole source code can be fitted into 2 pages. The backbone of the compiler is a recursive descent parser which fires specific operations when certain patterns are matched. Those operations include datapath and control signal generations. The fly compiler accepts Perl-like description and produces VHDL as output.

Since the fly compiler is an one-pass compiler, there is no intermediate representation in the fly compiler. In order to integrate the technology mapper into the fly compiler, the first step is to modify the fly compiler to produce a dataflow graph instead of instantiating floating point operators when operations involving floating point data are encountered. The dataflow graph can then be used by the technology mapper to produce configured and mapped coarse-grained units.

The fly compiler is written in Perl and the "`Parse::RecDescent`" package is employed to implement the top-down recursive text parser which is the core of the fly compiler. The fly compiler has defined a set of grammar specification which consists of rules and the associated subroutines. The subroutines are executed when the rules are matched to generate datapath and control circuits.

The following list shows the rules that are relevant. It defines the floating point operations in the Perl description.

The Fly Grammar Related to Floating Point Operation.

```
fop: '.*' | '.+' | '.-'
var: /\$[a-z][\w\[\]]*/
fexpr: var fop var
asgn: var '=' fexpr ';'
```

Rule `fop` defines a set of floating point operators. Rule `var` defines the naming convention of a floating point variable. Rule `fexpr` defines the floating point expression and Rule `asgn` defines a statement in the fly compiler. For example, the parser will trigger the subroutines of `fexpr` and `asgn` sequentially when the statement `$d1 = $d2 .* $d3;` are parsed.

In the original fly compiler implementation, Rule `fexpr` instantiates floating point components and the associated control signals in VHDL. In this work, instead of instantiating floating point components, the operations are written in a file representing a dataflow graph as shown in Figure 6.2. Then we can use the `techmap` algorithm to generate mapped circuits. As an example, the modified compiler produces the following file when parsing the statement `$d1 = $d2 .* $d3;`

Sample Output After Modification.

```
fmul tmp1, d2, d3
fmov d1, tmp1
```

In the second step, we need to integrate the control signal into the coarse-grained units. The fly compiler uses one-hot encoding to construct state machine, in which all computation blocks are associated with a `start` and a `finish` signal. Users are required to design the same control signals for the datapath given by the technology mapper. It is essentially an 1-bit FIFO which has the same latency as the configured coarse-grained unit.

The technology mapper generates netlists in VHDL to explicitly instantiate required coarse-grained units. Since the fly compiler generates VHDL descriptions of control logic and other computation logic such as fixed point computation, the integration can be achieved by connecting these two VHDL descriptions using `port map` keyword manually.

6.5 Results

Circuit	number of adder	number of multiplier	Minimum CGU	Mapped CGU	Relative Difference
bfly	4	4	2	2	0%
dscg	2	4	2	2	0%
fir	3	4	2	2	0%
mm3	2	3	2	2	0%
ode	2	2	2	2	0%
bgm	9	11	6	7	16%
syn2	5	4	3	3	0%
syn7	25	25	13	16	23%

Table 6.1: Performance of the technology mapper. Most circuits require minimum number of coarse-grained unit (CGU).

There are several parameters to be specified in the construction of coarse-grained unit. In this experiment, the following properties of the coarse-grained unit are assumed: 3 input-buses, 4 output-buses, 4 feedback registers, 5 generic wordblocks, 2 floating point multipliers located in the second and fifth column, 2 floating point adders located in the third and sixth column. The bus widths are all 32-bit and the floating point operators are single precision. In addition Xilinx Virtex II 3000 FPGA is chosen to be host fine-grained fabric for the FPFPGA as the technology process is similar to the coarse-grained unit.

Eight benchmark circuits are used in this study. They are the same as those used in Chapter 5. The kernels of the benchmarks are described as dataflow graphs and the technology mapper is used to map the design. It shows that the mapper can map the applications effectively in terms of the number of coarse-grained unit instantiated. For most designs, the mapper can instantiate minimum amount of coarse-grained units. It is determined by the number of floating point operators required divided by the number of floating point operators in a coarse-grained unit. Then the result is rounded up to the nearest integer. Table 6.1 summarises the performance of the technology mapper. It shows that only circuit *syn7* and *bgm* require more coarse-grained units than the minimum requirements. This is because some outputs of coarse-grained units are used up so some of the signals cannot be routed. Consequently, the mapper has to replicate some of the operations which increases the number of coarse-grained unit required. It is expected that the utilisation rate of the coarse-grained unit can be improved by increasing the number of output buses.

6.6 Summary

This chapter discusses various CAD tools which can assist users to design application on FPFPGA devices. We have assessed different approaches to design applications on FPFPGA such as traditional HDL approach and high level synthesis approach. The traditional CAD tools appearing in standard FPGA devices can be reused on FPFPGA. We also explore opportunities to adopt high level synthesis design flow on FPFPGA devices. In particular, we propose architecture-aware technology mapper algorithms. It is capable of producing configured coarse-grained unit from a dataflow graph which represents the computation kernel of applications. We address the issue of integrating such technology mapper on various high level synthesis tools such as Trident and the fly compiler. Experiments show that the technology mapper is efficient and it can infer minimum number of coarse-grained units in 6 out of 8 benchmark kernels.

Chapter 7

Conclusion

7.1 Summary of Achievements

In this work, we have proposed a platform to design reconfigurable devices for optimising floating point applications. The platform consists of (1) a modelling methodology which allows rapid estimation of the performance when arbitrary heterogeneous blocks are embedded in existing standard FPGA, (2) a synthesisable FPGA fabric dedicated to datapath oriented operations, (3) a customisable coarse-grained unit embedded in FPGA for accelerating floating point applications, and (4) an architecture-aware technology mapper which can translate dataflow graphs into mapped circuits. The mapper can be integrated into high level synthesis tools for designing applications on the proposed reconfigurable platform.

It is challenging to compare the proposed architectures with existing architectures. Variation between design tools on different FPGA architectures may yield completely different results which are difficult to compare. Chapter 3 establishes a methodology to address this issue. We propose a novel approach to remedy this problem by introducing Virtual Embedded Block (VEB) design flow. By exploiting the vendor design tools, the flow allows comparison between existing FPGA and virtual FPGA with arbitrary heterogeneous blocks under the same conditions (including design entry, architecture of fine-grained units, routing resources, synthesis algorithms, place and route algorithms, timing analysis algorithms). This methodology helps us to produce reliable results for comparison.

Before identifying suitable reconfigurable architecture for floating point applications, we realise that most floating point applications are datapath oriented. By exploiting this property, Chapter 4 proposes a synthesisable datapath FPGA architecture which offers optimisation on bus-based logic. The architecture consists of general-purpose LUTs in which the LUTs contain shared configuration bits to reduce silicon area. Further area saving is given by bus-based routing and uni-directional channels. By exploiting strong correlation between output pin locations and their destinations in bus-based logic, few configuration bits are required to control the routing of a bus. In addition, we adopt standard macrocell design flow and parameterised design flow such that architecture exploration by parameter sweep is feasible.

Chapter 5 proposes a novel coarse-grained architecture which combines bus-based reconfigurable datapaths and routing with ASIC floating point operator. The coarse-grained units can be embedded in existing FPGA devices to form FPFPGA devices. FPFPGAs can be customised according to domain-specific requirement before fabrication and can be reconfigured to implement different applications. For instance, one can customise an FPFPGA by including a macrocell-based floating point divider and a square root operator into the coarse-grained unit when the operations are required in many times in a particular domain. The synthesisable design flow proposed in this work for FPFPGA design allows us to achieve the customisability.

The performance given by FPFPGA is promising in terms of area, speed and power consumption. In one particular implementation of FPFPGA, it is shown that the area can be reduced by 25 times, the clock frequency is increased by 4 times and dynamic energy consumption is reduced by 14 times on average when comparing the proposed architecture with an existing commercial FPGA device. We believe the performance can be further improved when custom layout design flow is employed in designing coarse-grained units.

The usability of FPFPGA has been taken into account; the FPFPGA is difficult to use if there are no CAD tools supporting it. If we consider the customisability aspect of FPFPGA, the CAD tools are more complicated as the tools should be able to support different customised versions of FPFPGA with little or no modification. Although the FPFPGA can adopt traditional FPGA design flow using HDL, Chapter 6 proposes a technology mapper to allow high level synthesis tools to produce circuits for FPFPGA devices. Algorithms associated with the technology mappers are architecture-aware in which the algorithms consider the architectural parameters of each coarse-grained units in the FPFPGA during the mapping process. The technology mapper can produce mapped circuits and bitstream which can be used in VEB modelling immediately. The mapped circuits, which are described in VHDL, instantiate all the required coarse-grained units and their interconnections, and can be used as benchmark while the bitstream can be used for power and timing analysis of coarse-grained units. This helps FPGA designer to explore different FPFPGA architectures without rewriting the benchmark circuit. We also demonstrate how to integrate the technology mapper into different high level synthesis tools. In particular, hardware compilers specially designed for floating point applications such as Trident and the fly compiler have been discussed.

The contributions described in Chapter 3 to 6 constitute an approach for designing customisable and reconfigurable devices dedicated to floating point computations. The devices can be customised in design phase (pre-fabrication) and can be reconfigured in usage phase (post-fabrication).

Before device fabrication, the platform provides a rapid modelling flow which allows designers to customise devices for specific application domains. After device fabrication, devices can be reconfigured for different floating point applications. In addition, the platform provides CAD tools for mapping applications represented in high level languages into datapath and control signal circuitry. Using the tools, application designers can efficiently develop reconfigurable circuits on FPFPGA and obtain improvements in speed, area, and power consumption.

FPGA device designers, while exploiting the VEB model, the standard macrocell design flow and the coarse-grained unit circuit generator, can produce different FPFPGA models rapidly with different set of architectural parameters. The benchmark circuits for performance evaluation are available by compiling their existing applications using the CAD tools with the same set of parameters. This design flow proposed in the thesis is generic and can also be used in developing other FPGA devices with different heterogeneous blocks.

7.2 Future Work

Although the thesis focuses on the development of FPFPGA devices, the methodology and architecture delivered in this work leads to other interesting research area. This section suggests some exciting future work which is derived from the contributions in the thesis.

VEB Experiments

The methodology proposed in Chapter 3 is a novel approach to FPGA modelling which can assess not only floating point operators but also arbitrary heterogeneous blocks. This approach is effective when studying heterogeneous components on existing island-style FPGA architectures. For instance, one can use the same approach to investigate the effects of embedding microprocessors into an FPGA and compare them with PowerPC 405 [Xili 08a], which has appeared in certain FPGA devices such as Virtex-II Pro and Virtex 4, soft-core (circuit implemented on fine-grained units) microprocessors OpenSparc [Leon 07] and MicroBlaze [Xili 07].

The area and timing information of OpenSparc, for example, can be obtained from synthesising the OpenSparc RTL description or from published information. We can study the trade-off of OpenSparc by customising some features in OpenSparc such as memory controller, floating point unit, memory management unit and so on. This allows us to study the optimal architecture of OpenSparc processor embedded in FPGA devices.

FPGA designers who are interested in networking applications can estimate the performance of an FPGA when dedicated networking components such as packet filters are embedded.

The design flow has currently been used to study other heterogeneous components embedded in FPGAs. Chong and Parameswaran [Chon 09] have applied the same approach to assess a multi-mode embedded floating point unit for FPGA. The essence of this approach is that potentially any embedded block can be modelled without actually *fabricating* the circuit. This greatly reduces both cost and time when conducting similar research.

Floating Point FPGA for High Performance Computing

This work introduces FPFPGA for general floating point applications. It can be further tailor-made to accelerate domain-specific floating point applications. For instance, high performance computing (HPC) applications can be good candidates to implement on customised FPFPGA devices. Many of them require a large amount of double precision floating point operations and demand fast local cache.

The research can first concentrate on commonly used routines for HPC. We can first study the HPC Challenge benchmarks suite [Lusz 06]. The suite provides well known computation kernels such as matrix-matrix multiply and Fast Fourier Transform (FFT). The suite also offers the LINPACK benchmark [Dong 03] which analyses and solves linear equations and linear least-squares problems. LINPACK is considered as a basic requirement when evaluating a HPC system. All the tests stress the floating point performance of a system hence they are good candidates to be representative applications for FPFPGA customised for HPC. These benchmarks can be mapped to FPFPGA by using high level synthesise tools containing our technology mapper. We can explore different architectural parameters of FPFPGA by adjusting the parameters, generating new FPFPGA coarse-grained units, compiling the benchmarks, evaluating the performance with VEB flow iteratively. Different combinations and configurations of coarse-grained units which involve floating point division, square root or other elementary function can also be considered.

FPFPGA and GPU Devices

The FPFPGA device shares similar building blocks with Graphic Processing Unit (GPU) in certain ways. Both of them possess large amount of floating point operators and specialise in floating point computations. However, their architecture is different from each other. For instance, there are no reconfigurable components in GPU so all the computations are done by software programming. On the contrary, GPU has better support in accessing local and off-chip memory and programming model [Nvid 09]. These advantages enable GPU to be one of the desired platforms for accelerating floating point applications.

However, by utilising reconfigurable architecture on GPU, it is exciting to combine the essence of FPFPGA and GPU to evolve a novel architecture for floating point computations. Some research has currently been conducted in this direction. For instance, Cope et al. [Cope 08] have explored opportunities to improve the performance of GPU memory accesses by introducing reconfigurable hardware on it. The future work can study the benefit and the trade-off when specific features implemented to the new device. For example, one may want to include reconfigurable bus-based interconnect into GPU in enhancing the flexibility or one may want to include dedicated memory interface to FPFPGA to increase the bandwidth. We believe that the architecture of GPU and FPFPGA can be improved by incorporating certain features of each other.

FPFPGA as IP blocks

Although we show that coarse-grained units in FPFPGA work well with existing fine-grained units in FPGA, the coarse-grained units can be IP blocks integrated into other devices. The idea is similar to the one discussed in Chapter 4, in which we have introduced a datapath-oriented fabric embedded into SoC to enhance the post-fabrication debugging ability. The following devices can be good candidates for embedding customised coarse-grained units.

SoC device: Since the advancement of technology process, the current trend of SoC development tends to include a dedicated floating point computation block to provide more computation power. Although the usual practice is to embed a floating point coprocessor, there are several potential advantages to adopt customised coarse-grained units over the traditional approach. For example, we can exploit the customisability of the coarse-grained unit to design a coarse-grained unit dedicated to a particular application domain. The coarse-grained unit may use fewer transistors while provides the same functionality as a floating point coprocessor.

Coarse-grained FPGA: Some implementations of coarse-grained FPGAs allow integration of different coarse-grained fabric and we can investigate the benefit of embedding coarse-grained units into those FPGA devices. Possible candidates include ADRES [Mei 03] and RaPiD [Ebel 96] architectures. In ADRES architecture, the coarse-grained units can be embedded as reconfigurable cells and the integration allows ADRES to perform floating point computation more efficiently. In RaPiD architecture, the coarse-grained unit can be embedded by attaching the reconfigurable coarse-grained units on bus connectors.

VLIW (Very Long Instruction Word) processor: One major advantage of VLIW processors is that their instruction set allows multiple operations to be encoded in a single instruction. Hence, a VLIW processor can control several computation cores in parallel. This can be a good match with coarse-grained units. A recent study has reported integration of reconfigurable hardware and VLIW processors [Hoar]. We can adopt similar techniques to embed the coarse-grained units into a VLIW processor. We believe such integration allows the VLIW processor to fully utilise the computation capability of the coarse-grained units.

Bibliography

[Abra 06] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In: *ACM IEEE Design Automation Conference*, pp. 7–12, June 2006.

[Agil 07] Agility Design Solution Inc. *Handel-C Language Reference Manual*. 2007.

[Agil 08] Agility Design Solution Inc. *Software Product Description for DK Design Suite Version 5.0*. April 2008.

[Ahme 04] E. Ahmed and J. Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 12, No. 3, pp. 288–298, March 2004.

[Aken 05] V. C. Aken'Ova, G. Lemieux, and R. Saleh. An improved "soft" eFPGA design and implementation strategy. *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, Vol. , No. , pp. 179–182, September 2005.

[ANSI 85] ANSI/IEEE, New York. IEEE Standard for Binary Floating-Point Arithmetic. Tech. Rep., The Insittution of Electrical and Electronics Engineerings, Inc, 1985. IEEE Std 754-1985.

[Beau 08] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert. Architectural Modifications to Enhance the Floating-Point Performance of FPGAs. *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 16, No. 2, pp. 177–187, 2008.

[Beck 04] l. Beck. A Place-and-Route Tool for Heterogeneous FPGAs. Tech. Rep., Cornell University, 2004.

[Bela 02] P. Belanovic and M. Leeser. A Library of Parameterized Floating-Point Modules and Their Use. In: *Proceedings of Field Programmable Logic (FPL)*, pp. 657–666, 2002.

[Betz 97] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In: *Proceedings of Field Programmable Logic (FPL)*, pp. 213–222, 1997.

[Betz 99] V. Betz, J. Rose, and A. Marquardt, Eds. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[Brig 05] A. Bright, R. Haring, M. Dombrowa, M. Ohmacht, D. Hoenicke, S. Singh, J. Marcella, R. Lembach, S. Douskey, M. Ellavsky, C. Zoellin, and A. Gara. Blue Gene/L compute chip: synthesis, timing, and physical design. *IBM Journal of Research and Development*, Vol. 49, No. 2/3, pp. 277–287, March/May 2005.

[Call 06] O. Callanan, D. Gregg, A. Nisbet, and M. Peardon. High Performance Scientific Computing Using FPGAs with IEEE Floating Point and Logarithmic Arithmetic for Lattice QCD. *Proceedings of Field Programmable Logic (FPL)*, Vol. , No. , pp. 1–6, August 2006.

[Chen 01] D. Chen, J. Cong, M. Ercegovac, and Z. Huang. Performance-Driven Mapping for CPLD Architectures. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 39–47, February 2001.

[Cher 96] D. Cherepacha and D. Lewis. DP-FPGA: An FPGA Architecture Optimized for Datapaths. *VLSI Design*, Vol. 4, No. 4, pp. 329–343, 1996.

[Chon 09] Y. J. Chong and S. Parameswaran. Flexible Multi-Mode Embedded Floating-Point Unit for Field Programmable Gate Arrays. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 171–180, ACM, New York, NY, USA, February 2009.

[Comp 04] K. Compton and S. Hauck. Flexibility measurement of domain-specific reconfigurable hardware. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 155–161, 2004.

[Comp 07] K. Compton and S. Hauck. Automatic Design of Area-Efficient Configurable ASIC Cores. *IEEE Transactions on Computers*, Vol. 56, No. 5, pp. 662–672, May 2007.

[Cope 08] B. Cope, P. Y. K. Cheung, and W. Luk. Using reconfigurable logic to optimise GPU memory accesses. In: *Proceedings of the conference on Design, automation and test in Europe*, pp. 44–49, ACM, New York, NY, USA, 2008.

[Craw 08] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In: *Proceedings of the 2008 conference on Computing frontiers*, pp. 3–12, 2008.

[Cron 99] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, Vol. , No. , pp. 23–40, March 1999.

[Dido 02] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 50–55, 2002.

[Dong 03] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, Vol. 15, No. 9, pp. 803–820, 2003.

[Dong 08] J. J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. 2008.

[Dou 05] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 86–95, 2005.

[Ebel 96] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In: *Proceedings of Field Programmable Logic (FPL)*, pp. 126–135, 1996.

[Flac 05] B. Flachs, S. Asano, S. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a CELL processor. In: *Digest of Technical Papers, Solid-State Circuits Conference*, pp. 134–135, 2005.

[Flet 82] J. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, Vol. COM-30, No. 1, pp. 247–252, January 1982.

[Gold 00] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. PipeRench: a reconfigurable architecture and compiler. *IEEE Transactions on Computers*, Vol. 33, No. 4, pp. 70–77, April 2000.

[Gold 91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, Vol. 23, No. 1, pp. 5–48, 1991.

[Hauc 04] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimera Reconfigurable functional unit. *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 12, No. 2, pp. 206–217, February 2004.

[Haus 98] J. Hauser. *TestFloat Release 2a General Documentation*. http://www.jhauser.us/arithmeic/testfloat.txt, 1998.

[Ho 02a] C. Ho, M. Leong, P. Leong, J. Becker, and M. Glesner. Rapid Prototyping of FPGA based Floating-point DSP Systems. In: *Proceedings of Rapid System Prototyping*, pp. 19–24, 2002.

[Ho 02b] C. Ho, P. Leong, K. H. Tsoi, R. Ludewig, P. Zipf, A. Ortiz, and M. Glesner. Fly - A Modifiable Hardware Compiler. In: *Proceedings of Field Programmable Logic (FPL)*, pp. 381–390, LNCS 2438, Springer, 2002.

[Ho 03] C. Ho, K. Tsoi, H. Yeung, Y. Lam, K. Lee, P. Leong, R. Ludewig, P. Zipf, A. Ortiz, and M. Glesner. Arbitrary function approximation in HDLs with application to the N-body problem. In: *Proceedings of Field Programmable Technology (FPT)*, pp. 84–91, 2003.

[Hoar] R. R. Hoare, A. K. Jones, D. Kusic, J. Fazekas, J. Foster, S. Tung, and M. McCloud. Rapid VLIW processor customization for signal processing applications using combinational hardware functions. *EURASIP J. Appl. Signal Process.*, Vol. 2006, pp. 1110–8657.

[Holl 07] M. Holland and S. Hauck. Automatic Creation of Domain-Specific Reconfigurable CPLD for SoC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, No. 2, pp. 291–295, February 2007.

[Hsu 06] S. Hsu, S. Mathew, M. Anders, B. Zeydel, V. Oklobdzija, R. Krishnamurthy, and S. Borkar. A 110 GOPS/W 16-bit Multiplier and Reconfigurable PLA Loop in 90-nm CMOS. *IEEE Journal of Solid State Circuits*, pp. 256–264, 2006.

[Jaen 01] A. Jaenicke and W. Luk. Parameterised floating-point arithmetic on FPGAs. In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 897–900, 2001.

[Jami 05] P. Jamieson and J. Rose. A Verilog RTL synthesis tool for heterogeneous FPGAs. In: *Proceedings of Field Programmable Logic (FPL)*, pp. 305–310, August 2005.

[Kund 04] P. D. Kundarewich. and J. Rose. Synthetic circuit generation using clustering and iteration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 23, No. 6, pp. 869–887, June 2004.

[Kuon 07] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, No. 2, pp. 203–215, Feb. 2007.

[Lang 08] M. Langhammer. Floating point datapath synthesis for FPGAs. In: *Proceedings of Field Programmable Logic (FPL)*, pp. 355–360, September 2008.

[Leij 03] K. Leijten-Nowak and J. L. van Meerbergen. An FPGA architecture with enhanced datapath functionality. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 195–204, 2003.

[Leon 07] A. S. Leon, K. W. Tam, J. L. Shin, D. Weisner, and F. Schumacher. A Power-Efficient High-Throughput 32-Thread SPARC Processor. *IEEE Journal of Solid State Circuits*, Vol. 42, No. 1, pp. 7–16, January 2007.

[Lusz 06] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) benchmark suite. In: *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 213, ACM, New York, NY, USA, 2006.

[Luu 09] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose. VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 133–142, ACM, New York, NY, USA, 2009.

[Mars 99] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 135–143, 1999.

[Math 99] J. Mathews and K. Fink. *Numerical Methods Using MATLAB*, pp. 433–441. Prentice Hall, 3rd Ed., 1999.

[Mei 03] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In: *Proceedings of Field Programmable Logic (FPL)*, pp. 61–70, 2003.

[Menc 06] O. Mencer. ASC: a stream compiler for computing with FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 25, No. 9, pp. 1603–1617, 2006.

[Mene 96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, pp. 602–606. CRC Press, 1996.

[Mitr 98] S. K. Mitra. *Digital Signal Processing A Computer-Based Approach International Editions 1998*, pp. 339–416. McGraw-Hill, 1998.

[Morr 05] G. Morris and V. Prasanna. An FPGA-based floating-point Jacobi iterative solver. *Proceedings of Parallel Architectures,Algorithms and Networks*, Vol. , No. , pp. 8–15, December 2005.

[Naka 88] T. Nakassis. Fletcher's Error Detection Algorithm: How to implement it efficiently and how to avoid the most common pitfalls. *ACM Computer Communication Review*, Vol. 18, No. 5, pp. 86–94, October 1988.

[Nvid 09] Nvidia Corp. *NVIDIA CUDA Architecture – Introduction and Overview*. April 2009.

[Pada 03] K. Padalia, R. Fung, M. Bourgeault, A. Egier, and J. Rose. Automatic Transistor and Physical Design of FPGA Tiles From an Architecture Specification. In: *ACM International Conference on FPGAs*, pp. 164–172, February 2003.

[Page 91] I. Page and W. Luk. Compiling Occam into FPGAs. In: *FPGAs*, pp. 271–283, Abingdon EE&CS Books, 1991.

[Quin 05] B. Quinton and S. Wilton. Post-Silicon Debug Using Programmable Logic Cores. In: *Proceedings of Field Programmable Technology (FPT)*, pp. 241–247, December 2005.

[Raba 02] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits A Design Perspective*. Prentice-Hall, 2002.

[Roes 02] E. Roesler and B. Nelson. Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture. In: *Proceedings of Field Programmable Logic (FPL)*, pp. 637–646, 2002.

[Rudo 05] Rudolf Usselmann. *Open Floating Point Unit: Overview*. 2005.

[Saab 05] Saab Ericsson Space AB European Space Agency Contract Report. *Application-like Radiation Test of XTMR and FTMR Mitigation Techniques for Xilinx Virtex-II FPGA*. 2005.

[Sara 07] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching Processor Design Errors with Programmable Hardware. *IEEE Micro*, Vol. 27, No. 1, pp. 12–25, Janary/February 2007.

[Sent 92] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Tech. Rep., University of California, Berkeley, 1992.

[Sing 00] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, Vol. 49, No. 5, pp. 465–481, 2000.

[Trip 07] J. Tripp, M. Gokhale, and K. Peterson. Trident: From High-Level Language to Hardware Circuitry. *IEEE Transactions on Computers*, Vol. 40, No. 3, pp. 28–37, March 2007.

[Tuan 06] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger. A 90nm low-power FPGA for battery-powered applications. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 3–11, 2006.

[Unde 04] K. Underwood and K. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In: *Proceedings of Field Custom Computing Machine (FCCM)*, pp. 219–228, 2004.

[Wagn 06] I. Wagner, V. Bertacco, and T. Austin. Shielding Against Design Flaws with Field Repairable Control Logic. In: *Design Automation Conference*, pp. 344–347, July 2006.

[Wait 05] C. Wait. IBM PowerPC 440 FPU with complex-arithmetic extensions. *IBM Journal of Research and Development*, Vol. 49, No. 2/3, pp. 249–254, March/May 2005.

[Wilt 05] S. Wilton, N. Kafafi, J. Wu, K. Bozman, V. Aken'Ova, and R. Saleh. Design Considerations for Soft Embedded Programmable Logic Cores. *IEEE Journal of Solid-State Circuits*, Vol. 40, No. 2, pp. 485–497, February 2005.

[Wilt 07] S. Wilton, C. Ho, P. H. Leong, W. Luk, and B. Quinton. A Synthesizable Datapath-Oriented Embedded FPGA Fabric. In: *Proceedings of Field Programmable Gate Array (FPGA)*, pp. 33–41, February 2007.

[Xili 04] Xilinx Inc. *Creating RPMs Using 6.2i Floorplanner*. Applications Note XAPP422, 2004.

[Xili 05] Xilinx Inc. *Floating-Point Operator v3.0*. Product Specification, 2005.

[Xili 07] Xilinx Inc. *MicroBlaze Processor Reference Guide*. 2007.

[Xili 08a] Xilinx Inc. *PowerPC 405 Processor Block Reference Guide*. 2008.

[Xili 08b] Xilinx Inc. *Web Power Tool Quick Start Guide*. 2008.

[Xili 99] Xilinx Inc. *XC4000XLA FPGAs Description*. 1999.

[Yan 06] A. Yan and S. Wilton. Product-Term Based Synthesizable Embedded Programmable Logic Core. *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 14, No. 5, pp. 474–488, 2006.

[Yang 91] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. 1991.

[Ye 03] A. Ye, J. Rose, and L. David. Architecture of datapath-oriented coarse-grain logic and routing for FPGAs. *Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003*, Vol. , No. , pp. 61–64, September 2003.

[Ye 06] A. Ye and J. Rose. Using Bus-Based Connections to Improve Field-Programmable Gate-Array Density for Implementing Datapath Circuits. *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 14, No. 5, pp. 462–473, 2006.

[Yui 02] C. Yui, G. Swift, and C. Carmichael. Single Event Upset Susceptibility Testing of the Xilinx Virtex II FPGA. In: *Military and Aerospace Applications of Programmable Logic Conference (MAPLD)*, 2002.

[Zhan 05] G. Zhang, P. Leong, C. Ho, K. Tsoi, C. Cheung, D.-U. Lee, R. Cheung, and W. Luk. Reconfigurable Acceleration for Monte Carlo Based Financial Simulation. In: *Proceedings of Field Programmable Technology (FPT)*, pp. 215–222, 2005.

[Zhao 06] W. Zhao and Y. Cao. New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration. *Electron Devices, IEEE Transactions on*, Vol. 53, No. 11, pp. 2816–2823, November 2006.

[Zhuo 04] L. Zhuo and V. K. Prasanna. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs. *Proceedings of Parallel and Distributed Processing*, Vol. 01, pp. 92–101, 2004.