CUSTOMISABLE FPGA PLATFORM FOR ACCELERATING FLOATING POINT COMPUTATIONS

by

CHUN HOK HO

(cho@doc.ic.ac.uk)

A report submitted in fulfillment of requirements for the MPhil to PhD transfer examination.

Custom Computing Group

Department of Computing

Imperial College London

January, 2007

Abstract

Floating point computations have been one of the demanding calculations among diverse applications. While FPGA technology has been widely adopted to speed up computationally intensive applications, the capability of running floating point application on an FPGA is limited due to the size of an FPGA and the complexity of a floating point unit.

This research is focused on a reconfigurable device, which is optimised for floating point computation. This reconfigurable device employs significantly amount of heterogeneous blocks on fine-grained island-style FPGA fabric. The architecture of heterogeneous block employs parameterised design flow to allow architectural exploration. The fine-grained island-style FPGA fabric employs existing commercial FPGA architecture.

The issues and challenges associated with the architecture, modelling and design flow have been discussed in this report. The initial model of such an FPGA has been developed and preliminary results support that an FPGA specialised for floating point computations can yield better speed and density than a classic commercial FPGA device. In addition, this report presents the future plan and an outline of the thesis.

List of Publications

The following publications have been written during the course of this work:

- C.H. Ho, C.Y. Yu, P.H.W. Leong, W. Luk, S.J.E. Wilton, "Domain-Specific Hybrid FPGA: Architecture and Floating Point Applications", submitted to *Proceedings of Field-Programmable Custom Computing Machines*, 2007.

- C.H. Ho, P.H.W. Leong, W. Luk, S.J.E. Wilton, S. Lopez-Buedo, "Evaluating Embedded Elements in Reconfigurable Devices Using Virtual Embedded Blocks", to be submitted to *IEEE Transactions on Very Large Scale Integration Systems*.

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

- G.L.Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C. Cheung, D. Lee and W. Luk, "Reconfigurable Monte Carlo Simulation for Financial Modelling", submitted to *IEEE Transactions on Computers*.

- C.H. Ho, K.F.C. Yiu, J. Huo and W. Luk, "Reconfigurable acceleration of robust frequency-domain echo cancellation", in *Proceedings of Engineering of Reconfigurable Systems and Algorithms*, pp. 184–190, 2006.

- 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.

Contents

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 Blue Gene/L (BG/L) machine [1], which is currently the fastest supercomputer [2]. Yet I note that its floating point unit (FPU), which contributes more than 95% computation time in benchmark programs like Linpack as well as many other floating point applications like Monte Carlo simulation and N-body problem, constitutes only 10% of the total BG/L compute chip area. 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. I 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 the 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 [3] 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 [4] 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. My previous research [7] 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, my recent research [8] 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 and power consumption is beneficial to several area 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. 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 focuses on developing methods and tools for new FPGA architectures which improve the execution speed of floating point operations while retaining their programmability. The research will study current FPGA architectures, demanding floating point applications, and propose novel FPGA architectures that optimise floating point arithmetic. The *customisable* FPGA platform contains libraries and tool to allow user-defined customisations in FPGAs which have special facilities to support the implementation of floating point operations. Different floating point applications will be used to measure the performance of the *customisable* FPGA platform. The three main objectives of the research are:

1. To investigate architectural innovations to support floating point operations in FPGA, for instance by studying the function, the granularity and connectivity of possible hardware primitives for floating point arithmetic.

2. To explore advanced design tools for the *customisable* FPGA platform, to facilitate experiments with different customisations.

3. To investigate applications that can benefit from the proposed research and to quantify such benefit based on current and future technology.

To approach the objectives, we require (1) a generic FPGA architecture which represent different family of FPGA with arbitrary reconfigurable block, (2) a generic FPGA model which can produce accurate timing and area results according to (i) FPGA architectural

parameters and (ii) user-defined circuits implemented on this architecture, (3) an automatic synthesis tool which can produce a bitstream from high level description of a circuit and (4) representative floating point applications to evaluate an FPGA architecture.

Several work has been done to address the above-mentioned requirements. In particular, A comprehensive literature review and related work are presented in Chapter 2. This chapter also describes some of the benchmarks we used to evaluate FPGA architectures, which include floating point unit and some floating point circuits. The circuit can be as small as a simple dot-vector product or as large as an interest rate model derivatives. Chapter 3 demonstrates a methodology to model a commercial FPGA with arbitrary embedded blocks. This addresses the requirement (2) in some degree and it can provide a rapid evaluation on existing FPGA architecture. Chapter 4 propose a parametrised coarse-grained FPGA architecture and a synthesisable design flow which can model this architecture. The parametrised architecture allows us to search for a near-optimum floating point FPGA design in the future. By combining the methodology described in both chapter, an initial design of FPGA which is optimised for floating point computations is illustrated in Chapter 5. The future work, thesis outline, as well as the conclusion are made in Chapter 6.

Chapter 2

Related Work

A survey of several related topics has been conducted. Published work which includes floating point units, FPGA fabric architecture, FPGA design tools has been reviewed and summarised as below. We also summarised the benchmark circuits used in subsequent chapters.

2.1 FPGA architecture

An FPGA is made up of a reconfigurable fabric. The fabric itself consists of arrays of fine-grained or coarse-grained units. 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 [57].

Fine-grained units can also be implemented using a product-term block consisting of an AND and 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 [55] 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 coarse-grained unit. 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 coarse-grained unit is usually less flexible and typically much larger than a fine-grained one, but is often more efficient for implementing specific functions. The coarse-grained unit is usually programmable to some degree, combining several functions such as those in an arithmetic logic unit (ALU). Outputs are often multibit. They can be parameterised in terms of features such as bus-width and functionality. As an example, the ADRES architecture [58] assumes that the wordlength and the functionality of a coarse-grained unit 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 are 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 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 [14] proposed mixed-level granularity logic blocks and compared their benefits with a standard island-style FPGA using the Versatile Place and Route tool (VPR) [15]. Ye, Rose and Lewis [67] studied the effects of coarse grained logic cells and routing resources for datapath circuits, also using VPR.

Kuan [69] has reported the effectiveness of embedded elements in current FPGA devices by comparing with the equivalent ASIC circuit under $90nm$ technology process. Akan'Ova et. al.[70] has demonstrated a standard-cell-based eFPGA with improving performance using a structural design and layout approach. Compton and Hauck [71] have suggested a flexibility measurement on domain-specific reconfigurable architecture. Beck revised VPR to explore the effects of introducing hard macros [68].

Several previous studies have considered datapath-oriented FPGAs [33, 36, 39, 47, 48]. 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 [34, 35, 40, 44]. Of these, the RaPiD architecture [44] was 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 unprogrammed 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 a customisable FPGA optimised for floating point computations.

Yet, 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 [53]. The study of embedded heterogeneous blocks for the acceleration of floating point computations has been reported by Roseler and Nelson [52]. 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 [52] does not take into the account 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, in the design of fabric. It is described in Chapter 5.

While [53] evaluate the results by employing a modified VPR flow, where floating point unit model is added to the VPR framework, 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 propose a model which is comparable to existing FPGA device and that could produce more realistic results. This methodology is discussed in Chapter 3.

2.2 FPGA-based floating point units

A. Jaenicke and W.Luk [5] 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 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.

P. Belanovic̀ et al [6] implemented a parameterised floating point library for use with reconfigurable hardware. It is based on the IEEE 754 floating point format standard. The library includes addition, subtraction, multiplication and conversion between fixed point and floating point numbers. All of these modules are specified in VHDL and implemented on the Wildstar reconfigurable computing engine. They are fully-pipelined and cascadable to form pipelines of floating point operations. This library was used to develop a hybrid implementation of the K-means clustering algorithm applied to multi-spectral images.

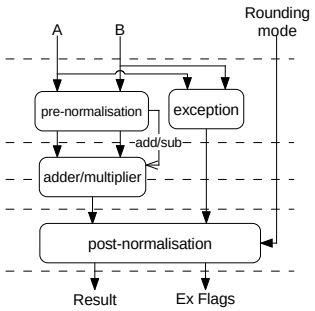


Figure 2.1: Simplified datapath of floating point adder/multiplier

Dido et.al. [76] 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.

A set of parameterisable floating point operators and floating point benchmark circuits have been developed in HDL model. The floating point operators are fully-compliant with IEEE754 [25] 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 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 [24]. It consists of several blocks, namely, a pre-normalisation block, an addition block, a post-normalisation blocks and an exception handling block. A simplified architecture of the floating point adder and multiplier is shown in figure 2.1. In the pre-normalisation stage, the inputs are registered and the exponents are compared. The inputs are swapped if it is necessary. The fractions are shifted to right accordingly and operation mode 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 is 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. In modelling the addition block, instead of using + operator in HDL code, a dedicated carry chain is explicitly specified. Therefore the VPR tools can associated the adder with dedicated carry chain instead of a traditional full adder block.

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 – 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 blocks 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 comfort to the IEEE 754 standard, an open-source program called *TestFloat-2a* [26] 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 does not rely on machine-specific floating point implementation. 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 "level 1" in the initial settings of *TestFloat-2a*. Table 2.1 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 error are found.

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

Table 2.1: 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 2.2: FPGA implementation results for Floating Point operators

Implementation – In order to compare with the floating point core using current commercial FPGA with the one using *customisable* FPGA, the floating point operators circuits have been implemented on FPGA device. The reference FPGA platform is XC2V6000 and the speed grade is -5. All the design is synthesis using "Synplify Pro 8.0". The designs are placed and routed and the area and the timing are obtained by vendor CAD packages ISE 7.1. Table 2.2 presents the area and frequency of the double precision floating point adder and multiplier.

2.3 Floating point applications

Many floating point systems have been implemented on FPGA devices. In [73], 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 [74], three of the basic linear algebra subroutine (BLAS) functions are estimated and it suggests that FPGAs platform 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 [77] 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.

O. Callanan et. al [78] demonstrate a 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 two times better than purely software implementation.

Zhuo and Prasanna [79] 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 achieve 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 [80] 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 uni processor implementation.

2.4 FPGA design tools

Different strategies have been proposed to model FPGA architectures. The VPR computer aided design (CAD) tool [54], 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, embedded memories, embedded multipliers cannot

be modelled. In addition, there is no commercial quality synthesis tool to support the VPR tool. This prohibit the use of VPR as it is difficult to implement relatively large circuit.

Yan and Wilton employ a synthesisable flow to model reconfigurable fabric [55]. 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.

In terms of high level synthesis on an FPGA device, several schemes such as ASC [63], Handel-C [81] and fly [11] 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 fly compiler to produce different experiments efficiently. Furthermore, it is possible to modify the fly compiler such that it can support the proposed FPGA architecture and this is briefly illustrated in Chapter 5.

2.5 Benchmark Circuits

As there are no existing standard benchmark circuits for floating point applications, a set of benchmark circuit is implemented using *fly* [64] compiler or using HDL. Significant amount

of time in developing the benchmark circuits are reduced as *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, *fly* compiler can speed up the implementation time of the benchmark applications. Four application circuits have been generated using *fly* compiler framework, 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 framework*(bgm)* [22] . 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.

2.5.1 Digital Sine-Cosine Generator *(dscg)*

Digital sine-cosine generator [20] 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:

$$\begin{bmatrix} s1_{n+1} \\ s2_{n+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \cos(\theta)+1 \\ \cos(\theta)-1 & \cos(\theta) \end{bmatrix} \begin{bmatrix} s1_n \\ s2_n \end{bmatrix} \tag{2.1}$$

2.5.2 Ordinary Differential Equation *(ode)*

Many scientific problems involve the solution of ordinary differential equations (ODEs). An ODE solver (ode) is implemented as part of the floating point benchmarks. The benchmark circuit solves the ODE [21]:

$$\frac{dy}{dt} = \frac{(t-y)}{2} \text{ over } t \in [0,3] \text{ with } y(0) = 1 \tag{2.2}$$

Euler method was used and y was approximated by

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

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.

2.5.3 Matrix Multiplication *(mm3)*

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

2.5.4 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:

$$y_4 = \sum_{j=0}^{4} k_j x_{4-j} \tag{2.3}$$

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 2.2.

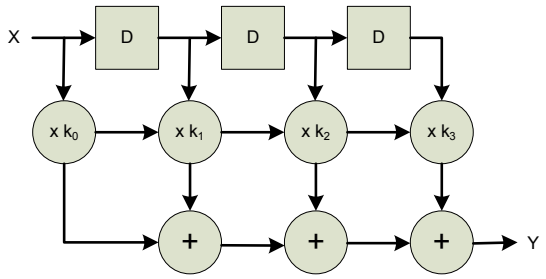


Figure 2.2: Four-tap FIR filter

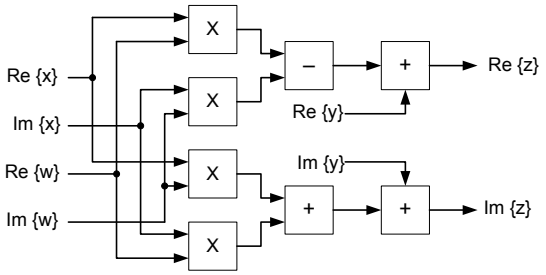


Figure 2.3: One butterfly stage in FFT

2.5.5 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 2.3 illustrates the datapath of a single butterfly which is used as the benchmark circuit.

2.5.6 Brace, Gąatarek and Musiela *(bgm)*

The datapath of a design to compute Monte Carlo simulations of interest rate model derivatives priced under the Brace, Gąatarek and Musiela (BGM) framework is used as the final test circuit *(bgm)* [77]. 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 BGM framework, 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:

$$\frac{dF_n(t)}{F_n(t)} = \vec{\mu_n}(t)dt + \vec{\sigma}_n(t) \cdot d\vec{W}(t) \qquad n=1 \ldots N. \tag{2.4}$$

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

$$\vec{\mu_n}(t) = \vec{\sigma}_n(t) \cdot \sum_{i=m(t)}^{n} \frac{\tau_i F_i(t) \vec{\sigma}_i(t)}{1 + \tau_i F_i(t)} \tag{2.5}$$

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 2.4), $d\vec{W}$ 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. $\epsilon \sim \phi(0, 1.0)$.

2.6 Summary

This chapter introduces the work related to the project. It first describes the common island-style fine-grained fabric and the application-specific coarse-grained fabric. The CAD tools for modelling an FPGA and the high level synthesis design tools for implementing user circuits on an FPGA are presented. A set of floating point units that use in many experiments are then illustrated. Some of the ideas in modelling reconfigurable fabric inspire me to develop platform independent synthesisable models. The end of chapter suggests some of the benchmark circuits to evaluate the customisable floating point FPGAs.

Chapter 3

Virtual Embedded Block

One major requirement of designing an FPGA architecture is to find a justified model which can produce high quality results based on the user-defined circuit. This chapter describes a methodology to we present 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 and delay 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 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.

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. 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. Power consumption is not considered in this study.

3.1 Methodology: Generic Aspects

In this section, the methodology is first described as a generic approach which can be applied to any FPGA and the associated design tools. The next section will cover the actual vendor-specific design flow used in this study.

We shall first provide an overview of our methodology that supports rapid generation of various benchmark applications to target reconfigurable architectures with VEB models. A modifiable compiler, called *fly* [64], is used so that different wordlength and back-end operator instances can easily be produced from a single algorithmic description. This allows both fixed- and floating-point implementations to be generated from the same description. We apply this methodology to a set of benchmark circuits generated in this fashion.

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. 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. Power consumption is not considered in this study.

In the descriptions that follow, we use the term logic cell (LC) for the smallest logic unit in the FPGA (usually a lookup table plus a register) and configurable logic block (CLB) for an array of LCs that are interconnected via the connection and switch blocks in the FPGA.

The basic strategy employed is to use the logic resources of a real FPGA to match the expected position, area and delay of an application specific integrated circuit (ASIC) implementation of an embedded block (EB). This could be achieved using appropriate vendor's tools or generic tools such as VPR [15]. In order to estimate its performance, the EB is modelled using logic cell resources in VEBs. Our model of an FPGA with EBs is called a virtual FPGA as illustrated in figure 3.1.

To employ this methodology, an area and delay model for the EB is required. The model should provide a high level estimate of the area and delay of the block, extracted from simulations of an existing design or come from measurements of an actual ASIC. The area

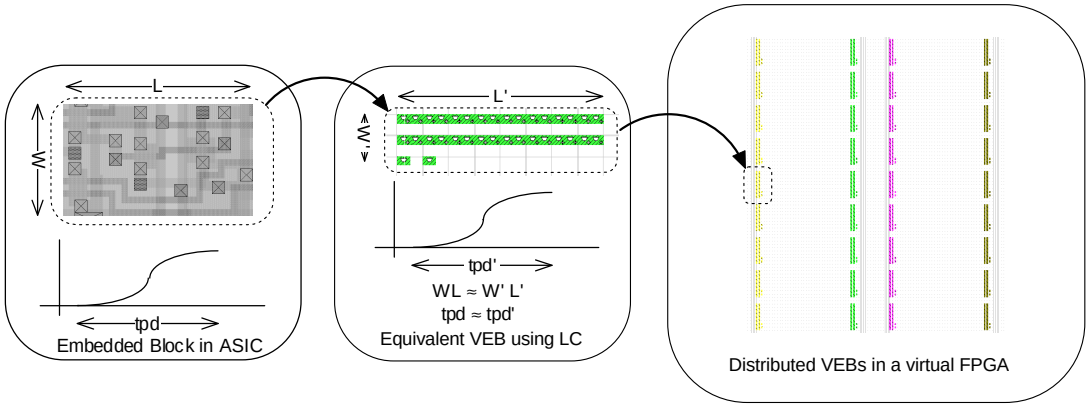


Figure 3.1: Modelling embedded elements in FPGAs using Virtual Embedded Blocks.

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. All area measures are normalised by dividing the actual area by the square of the feature size, making the area estimates independent of feature size. The 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. Table 3.1 shows a number of logic cell area estimates. The area estimate of the embedded blocks studied are given in section 3.3. We assume that there are sufficient ports to allow interconnection of the EB to the routing fabric. This may not be the case in some designs, particularly those with small EBs.

In order to accurately model delay, both the logic and 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 EB/VEBs, it may not be possible to accurately match the number of ports, area or logic delay and some inaccuracies will result. A complex EB might have many paths, each with different delays. It is possible to either assume that all delays are equal to the longest one (i.e. the critical path), or generate different delays for important paths. In the latter case, shorter delays can be obtained by taking intermediate points along the longest delay path.

Device	LCs/CLB L	Area/CLB A (μm^2)	Feature Size f (μm)	Normalised LC area ($N = A/Lf^2$)
Apex 20K400E [31]	10	63161	0.18	195,000
Virtex E [31]	4	35462	0.18	267,000
Virtex II 3000 [17]	8	$71,429 \times 0.7$	0.12	434,000
Virtex II 1000 [18]	8	$72,782 \times 0.7$	0.12	442,000

Table 3.1: Estimates of logic cell area including configuration bit, buffer and interconnect overheads. The Virtex II value of A is based on the estimate that 70% of the total die area is used for logic cells, the other area being for pads, block memories, multipliers etc.

Modelling wiring delays is more problematic, since the placement of the virtual FPGA must be similar to that of an FPGA with EBs so that their routing is similar. This requires that:

- The absolute location of VEBs match the intended locations of real embedded blocks (REBs) in the FPGA with EBs.
- The design tools be able to assign instantiations of VEBs in the netlist to physical VEBs while minimising routing delays.

The first requirement is addressed by locating VEBs at predefined absolute locations that match the floorplan of the FPGA with EBs. The assignment of physical VEBs is currently made by manually specifying its placement. Automated methods will be the subject of a later study.

3.2 Methodology: Vendor Specific Aspects

This section illustrates how a VEB can be used to model a real embedded multiplier block in Virtex II device as a case study. All of the results described in this work are obtained

using the Synplicity Synplify Pro 8.0 synthesis tool, the Xilinx ISE 7.1i design tools, and the Xilinx Virtex II XC2V6000-6-FF1152 FPGA device.

3.2.1 VEB Delay and Area model

While the ports for the VEB must be the same as those of the real embedded block, the VEB logic delay is emulated using a dummy circuit in the VEB implementation. Although many methods are possible, in this study, delays are inserted using adder carry chains for the following reasons:

- Adder carry chains are common to most FPGA platforms, enhancing the portability of the proposed methodology.
- The adder carry chain can be specified as a behavioural description, hence a platform independent delay block can be constructed.
- It is relatively easy to adjust an adder's carry chain delay by changing its length. This feature is used to model different embedded blocks.

The combinatorial logic delay of an adder carry chain 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 combinatorial 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 3.2.

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 trail-and-error is required to achieve a given delay.

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	4.66
T_{multck}	Registered embedded multiplier	3.000
T_{dyck}	Register setup and hold time	0.293

Table 3.2: Delay parameters for Virtex II-6 devices.

For the area model, the normalised LC area for the Virtex-II 1000 in table 3.1 is used in this study.

3.2.2 Integration of VEB into toolchain

In order to produce a VEB, it is first synthesised from a hardware description language (HDL) description. Features in the synthesis tool for regular design flows such as automatic I/O block insertion, pipelining and retiming are disabled. The resulting netlist is placed and routed using the vendor's toolchain. Area constraints must be specified to force the placement of the VEB in a rectangular block. The "trim unconnected logic" option is disabled to ensure that the VEB is not optimised away. After place and route, another constraint file which contains the actual placement information for each LC in the VEB is generated. The placement information and the netlist of the VEB is compiled to create a relationally placed macro (RPM).

To employ the VEB in an application, its 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, the VEBs are placed in a 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. The delays introduced in the VEB model the logic delay and its placement means that realistic routing is required. The vendor's tools are used to obtain resource utilisation delay information about the circuit.

3.3 Results

3.3.1 Verification of the VEB Approach

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

To estimate the normalised area of an EM in Virtex II, we assume that they occupy a total of 2% of the die area which, in turn, is reported to be 93 mm^2 [18]. This translates to a normalised LC area of approximately 2,751,000, which is 6 LCs. The timing information is extracted from the data sheet of the device; the relevant parameters are shown in table 3.2.

The benchmark circuits are implemented both using the EMs and the VEB multiplier. Table 3.3 summarises the resource utilisation and critical path delay for both implementations. Let us first compare the critical path delay, which is usually the parameter of most interest to a designer since it determines the maximum clock frequency at which the circuit can be operated. As one can see from the table, the difference between the two approaches is at most 11%. For most of the circuits, the critical path would involve 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.

For the bgm benchmark, table 3.3 shows that a speedup of 1.2 is gained by retiming. In designs where the stages are not as well balanced, as is often the case when a VEB is

introduced, more dramatic speedups are often observed. The retiming feature is absent from most VPR based design flows [15].

Table 3.4 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 3.3, 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. 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.

For the bgm circuit with retiming enabled, there is no corresponding path between the EM and VEB implementation because the registers are moved during this optimisation. The critical path of the VEB implementation is shown in this case and the difference column left blank.

3.3.2 Faster Embedded Multipliers

The VEB approach can be used to (a) obtain a single performance estimate for introducing embedded blocks, (b) analyse performance/area trade-offs, and (c) determine the EM speed required to meet a given system performance. To illustrate this point, we measure the bgm performance over a range of VEB delays. 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 multicycle designs.

The results are shown in figure 3.2. 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 [27]. 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 [28]. 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%.

3.3.3 Embedded Floating-Point Unit

An FPGA implementation of a double-precision FPU is made by synthesising the floating-point library in Section 2.5 targeting Virtex II technology. The size and performance of the adder and multiplier in this FPU are shown in table 3.5.

The area and delay model of a VEB floating-point unit (FPU) is made based on area and speed estimates of the Blue Gene ASIC [29, 30]. This is a state-of-the art 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 [29] which translates to 570 LCs. The area estimate is very conservative, since this FPU is much more sophisticated than the one used for the FPGA implementation.

Since 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 a one cycle latency. This essentially trades off clock frequency for reduced latency.

The performance of the Virtex II FPGA is compared to a virtual FPGA with embedded FPUs using the floating-point benchmarks. A summary of the results is given in table 3.6. As one can see, augmenting the FPGA with embedded FPUs leads to a mean improvement

in area and delay by factors of 3.7 and 4.4 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 55.0% and average increase of 40.7% in clock rate over existing architectures [16]. We attribute the differences to: different benchmarks being used; CAD tools; FPU delay and latency; FPGA model; and our use of retiming optimisations during synthesis.

Note that, for instance in the case of the ode benchmark, one can potentially support 3.8 times more dedicated FPUs in the same area as FPUs from programmable resources, meaning that more instances of the design can operate in parallel. Hence in the limit, the system throughput can be improved by up to 40 times if we include both improvement in speed and in parallelism due to area reduction.

Dedicated FPUs are wasted resources for designs that do not make use of them; however, each FPU occupies approximately the same area as 72 CLBs, which translates to 0.9% of the chip area of an XC2V6000 device.

3.3.4 Impact of Embedded Block Performance

Experiments are conducted, similar to those in section 3.3.2, 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 3.3). 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.

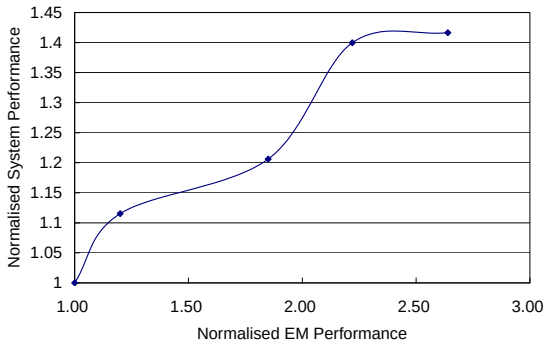


Figure 3.2: Performance of fixed-point bgm benchmark with different VEBs, with retiming.

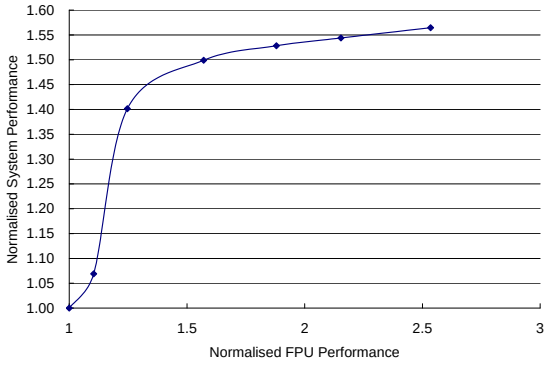


Figure 3.3: Performance of floating-point bfly benchmark with different FPU delays, with retiming. frequency.

3.4 Summary

This chapter propose a methodology for estimating the effects of introducing embedded blocks to existing FPGA devices. The methodology is evaluated by modelling block multipliers in Xilinx Virtex II devices, and we find that prediction of critical paths to approximately 10% 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 3.7 times and speedup of 4.4 times.

Benchmark	Size (slices)	# of EMs	EM delay (ns)	VEB delay (ns)	Difference (ns)	Difference (%)
dscg	177	4	4.599	4.981	0.382	8%
fir4	193	4	4.616	4.704	0.088	2%
ode	204	2	4.402	4.539	0.137	3%
mm3	469	3	4.859	4.815	0.044	1%
bfly	629	4	5.668	5.224	0.444	8%
mul34	141	4	11.191	11.287	0.096	1%
mul68	604	16	12.553	14.099	1.546	11%
mul136	2426	64	14.632	13.248	1.384	10%
bgm	2315	46	14.055	13.866	0.189	1%
bgm*	2205	46	11.594	11.602	0.008	0%

Table 3.3: Summary of resource utilisation and critical path delay for embedded multiplier (MULT18X18) and VEB implementations. A * indicates that retiming is enabled during synthesis.

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	1.15	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	8.99	2.202	0.172	1.913%	0.143	6%
mul68	8.682	3.687	8.99	4.96	0.308	3.426%	1.273	26%
mul136	8.682	5.95	8.99	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 3.4: Breakdown of critical path delay for embedded multiplier and VEB implementations. A * indicates that retiming is enabled during synthesis.

Operator	size (LCs)	# of EMs	Latency	Delay (ns)
FP Adder	3554	0	5	7.465
FP Multiplier	4300	9	5	13.197
FPU (VEB)	570	0	1	7.151

Table 3.5: FPGA implementation results for floating-point operators, where FPU(VEB) indicates the equivalent ASIC implementation of FPU using VEB approach. The FPU(VEB) is 6 times smaller than the floating-point adder and has only one clock cycle latency.

	FPGA				VEB				Reduction Factor	
	EMs	throughput (# of cycle)	size (LC)	delay (ns)	FPUs	throughput (# of cycle)	size (LC)	delay (ns)	Area	Delay
dscg	36	1	19006	22.711	6	1	3420 + 940	8.807	4.4	2.6
fir4	36	1	20590	23.545	7	1	3990 + 996	9.539	4.1	2.5
ode	18	20	13984	17.756	5	4	2850 + 870	8.525	3.8	10.4
mm3	27	225	17236	19.320	5	45	2850 + 2390	8.587	3.3	11.3
bfly	36	1	25640	20.245	8	1	4560 + 3424	8.821	3.2	2.3
								Geometric Mean:	3.7	4.4

Table 3.6: FPGA implementation results for floating-point benchmark applications. 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.

Chapter 4

Synthesisable Datapath FPGA Fabric

To allow design exploration for FPGA architecture, it is relatively difficult to employ full-custom design such as module generation which is common in SRAM design, where SRAM module generator usually have only one or two critical parameters such as bitwidth and address width. However, an FPGA fabric involves different parameters and it is less likely to introduce a module generator which is similar to those for SRAM. To remedy this, we propose a synthesisable methodology to model FPGA fabric.

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 a new type of FPGA can be created almost immediately by modifying the architectural parameters.

This chapter proposes this approach. Based on this methodology, we create an FPGA architecture which can be synthesised on an ASIC device. This architecture exploit characteristic of bus-based operation to produce bus-oriented coarse-grained architecture which optimised for performing computations such as those found in signal processing and arithmetic applications. Although such cores would be less flexible than their bit-level counterparts, this is less of a concern in an embedded FPGA core than in a stand-alone FPGA since the context in which the core will be used is known when the chip is designed. As

an example, a programmable logic core embedded into the datapath of a signal processing ASIC will certainly be used to implement multiply/accumulate-type functions, rather than a more general logic circuit. This allows us to create a core that is optimised for datapath operations without worrying about how well it can implement random logic functions. In addition, if buses are used to connect the programmable logic core and the fixed function circuitry (as would be expected in a datapath-oriented circuit), the specific pins on which these buses are mapped, as well as the width of the bus, are known at the time the fabric is instantiated, and will not change over the lifetime of the ASIC. This allows us to further optimise our fabric.

4.1 Architecture

In this section, we first outline the requirements of an architecture for a synthesisable FPGA core, and then describe our architecture in detail. We actually describe a *family* of architectures, where each member of the family is differentiated by various parameters. An FPGA 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.

4.1.1 Requirements of a synthesisable architecture

The proposed design methodology requires that the programmable logic fabric be *synthesisable*. By this, we mean the fabric can be synthesised and implemented using existing synthesis and ASIC design tools with no modifications to the tools or the CAD flow.

For a fabric to be synthesisable in this way, it must not contain combinational loops. Standard synthesis tools, timing analysis tools, and power estimation tools are optimised for circuits without combinational loops. Although circuits with such loops can be synthesised, this usually requires the designer to manually "break" the loops by identifying some false paths. This requires considerably more understanding about the internals of the core that 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.

On the other hand, 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 is identical.

One important aspect of our work is that we are focusing on *small* user circuits. Large circuits would typically be implemented using a hard-programmable logic core. An example circuit might be a small debug controller, as will be described later in this report.

4.1.2 Our architecture

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 implication of this feature on density in Section 4.3.

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 three 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

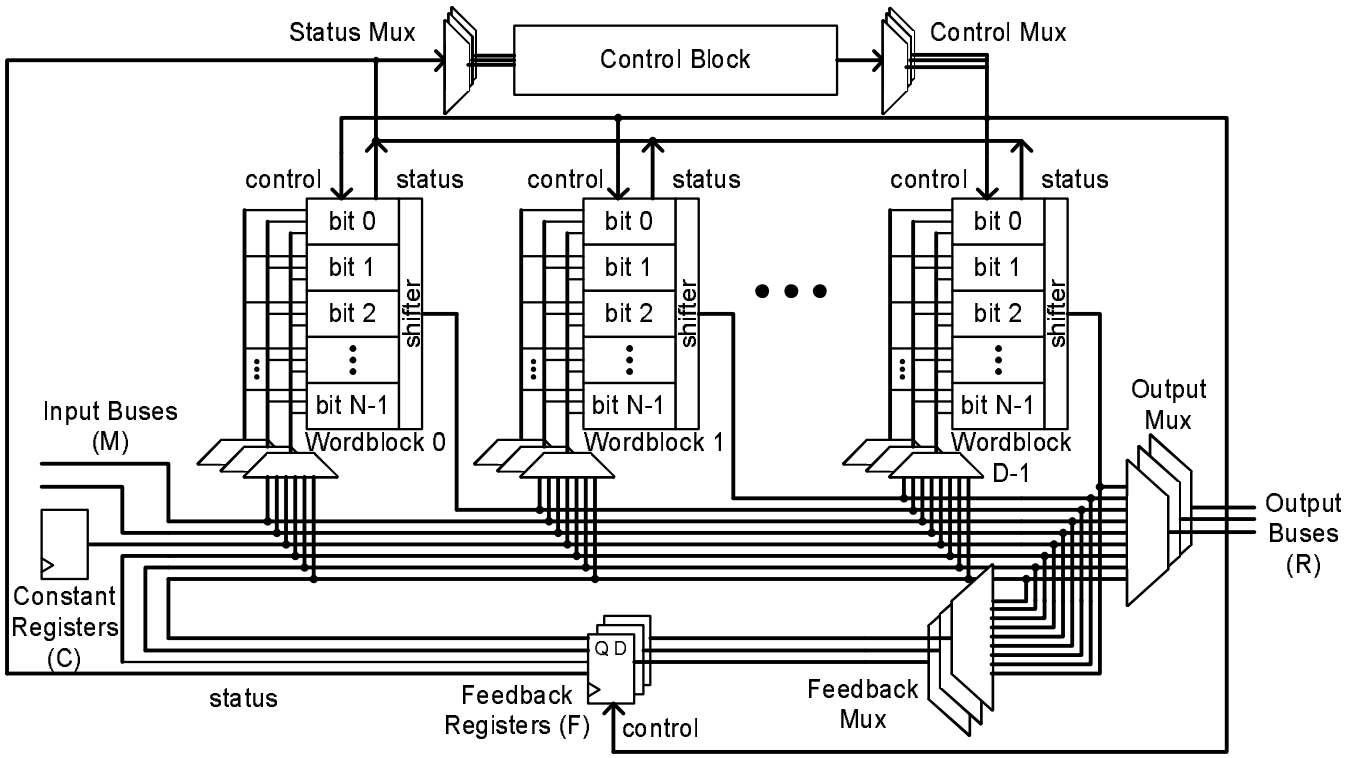


Figure 4.1: Fabric architecture (configuration elements not shown).

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 (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

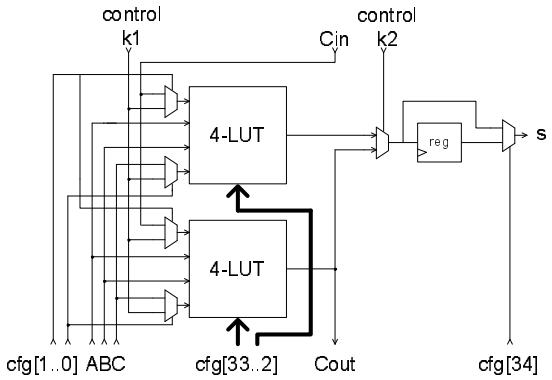


Figure 4.2: Bitblock (status flags not shown).

signals by one clock cycle without using a wordblock.

Wordblocks can efficiently implement combinational functions including adders and multiplexers, and can perform masking operations in conjunction with one or more of the constant registers. However, they cannot efficiently implement multipliers. Since multipliers are an important part of our target applications, 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 [46]. The fabric contains P product-term blocks, each with 9 inputs, 10 product terms, and 3 outputs (this was shown to work well in [46]). 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

D	Number of Wordblocks (incl. 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.

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 number 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.

4.2 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 [42]; the circuit monitors two buses, and counts the number of times a certain mask (composed of 1's, 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. Two constant registers

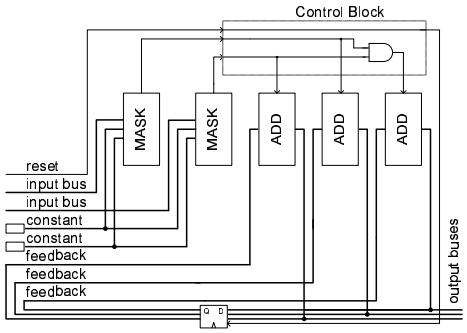


Figure 4.3: Example mapping.

are used to hold the mask value (two registers are required so that "don't care" bits can be specified). One wordblock combines these two mask values and the first input bus to produce a 0 if the bit matches (or is a "don't care") or a 1 otherwise. A second wordblock performs the same function on the second bus. Both wordblocks provide their zero flag (indicating a match has occurred) to the Control Block; the Control Block provides this signal to the carry-in signals of two adders (each implemented in a wordblock). The Control Block also provides the AND of the two zero flags to a third adder (implemented in another wordblock). Each of the three accumulated counts are 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.

4.3 Parameter optimisation

In this section, we first determine the impact of the parameters in Table 4.1 on the area and delay of the fabric.

Table 4.2 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 were synthesised using Synopsys Design Compiler, and the cell area predicted by Synopsys was reported. Configuration circuits, clock circuits, and all other essential parts of the core were included in the synthesisable

Module		Area in μm^2	Percentage
Datapath	Wordblocks	86, 251	23.8 %
	Multipliers	45, 236	12.5 %
	Config. Bits	24, 323	6.7 %
	Feedback Regs	2, 322	0.6 %
	Routing Muxes	86, 251	33.2 %
	Total Datapath	120, 460	76.7 %
Status Multiplexer		18, 520	5.1%
Control Multiplexer		14, 603	4.0%
Control Block		51, 418	14.2%
Total		363, 136	100.0%

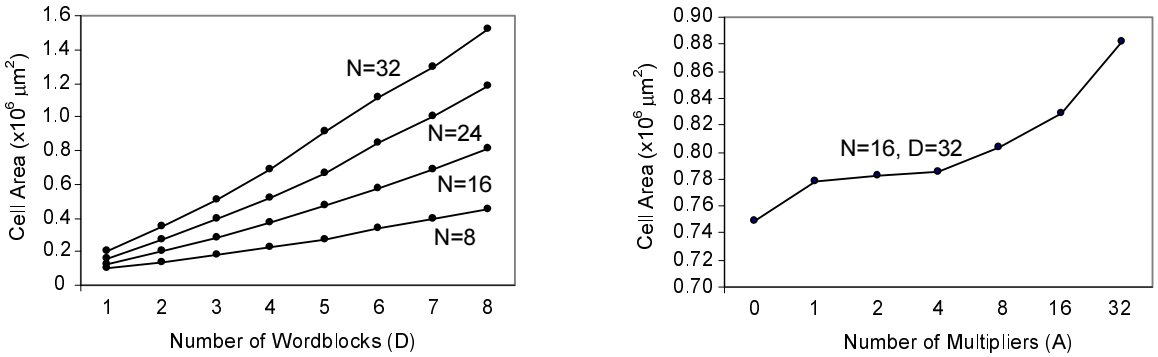
Table 4.2: Area breakdown.

model. Although it would be more accurate to perform place and route on the Synopsys-generated netlist and measure the chip area directly, previous results have shown that the Synopsys area results have a good correlation to the final chip area results [45]. A $130nm$ process was assumed.

As one can see, 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 also consume a significant amount of area. The configuration bits within the datapath consumes 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 routing multiplexers, while increasing N increases the sises 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 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



(a) Impact of D and N

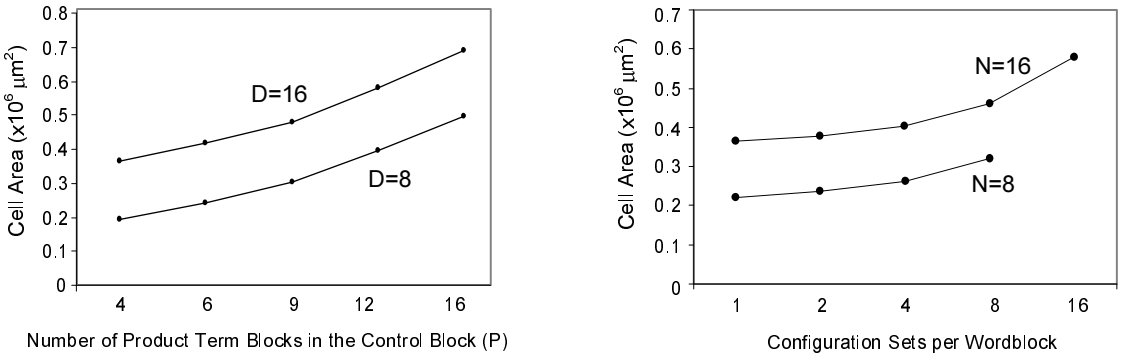
(b) Impact of A

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

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 measured the impact of M, R, C, and F. Each of these parameters had a linear effect on area. Increasing M from 1 to 8 increased the area by 15%, increasing R from 1 to 8 increased the area by 7.8%, increasing F from 0 to 6 increased the area by 25%, and increasing C from 0 to 8 increased 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



(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.

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 are shared among all bitblocks in a wordblock. To investigate the implication of this feature on density, we varied 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 (all other parameters are the same as before). As the graph shows, the more flexible architecture, the more area is required (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.

The maximum clock frequency at which the fabric can run depends on the configuration implemented in the fabric. Table 4.3 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,

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)	14.93ns
Delay through 24 wordblocks and 8 multipliers	178ns

Table 4.3: Delay estimates.

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). On the other hand, 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.

4.4 Mapping results

In this section, we use benchmark circuits to compare our architecture to a fine-grained synthesisable programmable logic core [46]. We first describe our benchmark circuits. We then present mapping results, 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.

4.4.1 Benchmark circuits

As described earlier, we are focusing on user circuits. An example is the debug controller described in Section 4.2. Such circuits typically contain a single datapath controlled by

Bench-mark	Fabric Parameters								Datapath (ours)	Fined-Grain [46]	ASIC	Fine-Grain/ Datapath	Datapath/ ASIC
	D	N	M	R	C	F	A	P					
bfly	8	8	6	1	0	5	4	0	68,190	132,339,335	9,300	1940	7.33
dotv3	5	8	6	1	0	2	3	0	34,119	65,534,780	6,575	1921	5.19
dscg	8	8	3	2	0	2	4	1	72,178	116,271,968	9,473	1611	7.62
fir4	11	8	1	1	4	0	0	0	76,213	130,971,120	9843	1718	7.74
egcd	27	8	2	4	1	9	0	27	1,225,231	22,776,474	10,420	18.6	117
momul	13	8	7	2	0	6	2	8	294,135	11,448,589	7,097	38.9	41
median	8	16	1	1	0	4	0	2	142,172	10,733,962	4,420	75.5	32
debug1	5	16	2	3	2	3	0	1	87,265	1,302,928	3,484	14.9	25

Table 4.4: Area results when the fabric is optimised for each benchmark circuit.

a small controller; circuits with multiple intersecting datapaths are likely too large to be implemented using a synthesisable core, and thus, we do not consider such circuits in this section.

We used eight benchmark circuits. Three of the benchmarks, *bfly*, *dscg* and *fir4* are described in Section 2.5. The other four circuits were constructed specifically for this work: The *dotv3* benchmark computes the dot vector product of two inputs. The *egcd* circuit implements an extended binary greatest common divisor algorithm [41]. The *momul* benchmark is a Montgomery Multiplier [41]. The *median* circuit is a median filter that accepts streaming data and returns the median (actually second-largest) of the last four entries. Finally, the *debug1* benchmark is the debugging circuit considered in Section 4.2. All benchmarks assume 8 bit operands, except *median* and *debug1* 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.

4.4.2 Optimised parameters

We first compare our architecture to the best previous synthesisable architecture [46] 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 was first split into datapath and control sections. The datapath portion of the circuit was mapped (by hand) to wordblocks, and appropriate values of D, N, M, R, D, A, F, and C were chosen. The control section was mapped to product-term blocks, using PLAmap [32]. Using 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 was used to generate an appropriately-sized fabric. This fabric was then synthesised using Synopsys Design Compiler, and the cell area predicted by Synopsys was reported. Again, a $130nm$ CMOS process was assumed. The results are shown in Column 10 of Table 4.4.

For comparison, we also show the area that would be required to implement the same circuit using the fine-grained synthesisable fabric from [46] in Column 11. These measurements were obtained using the architectures and tools described in [46]. We were unable to compare our architecture to the architecture described in [45], since that architecture only supports combinational circuits, and most of our benchmarks are sequential. However [46] shows that their architecture is significantly more dense than that in [45], even for combinational circuits. Column 12 shows the area required by the benchmark circuit if synthesised directly in standard cells (in which case there is no programmability).

Column 13 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 1610 and 1940. In other words, our architecture is 1610 times to 1940 times more area-efficient than the fine-grained fabric. The remaining circuits show more modest ratios between 14 and 75.

These results are dramatic. First consider those benchmarks with ratios between 14 and 75. 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 this has to do with the inefficiencies of the fine-grained

architecture when implementing very large circuits. The architecture in [46] contains many routing multiplexers; the size of these multiplexers *and* the number of these multiplexers both grow linearly with the size of the fabric. For the small circuits for which the previous architecture was designed, these multiplexers are small. However, when the fabric is scaled large enough to implement our benchmark circuits, these multiplexers become unwieldy, causing the area to grow significantly.

This does not explain 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). On the other hand, the fine-grained architecture does not contain these embedded blocks, meaning the multipliers must be implemented using the normal logic resources. This is aggravated by the fact that product-term based architectures, such as [46] are notoriously bad at implementing XOR functions, which are common in multipliers.

Column 14 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 5 and 8. For circuits without a significant number of embedded multipliers, this number is between 25 and 117. 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 [38]. 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.

4.4.3 Derived parameters

When gathering the results in Section 4.4.2 we chose all fabric parameters independently for each circuit. This unfairly biases the results in our favour. One of the drawbacks of

Benchmark	Fabric Parameters				Computed				Datapath	Fine-Grain	ASIC	Fine-Grain/	Datapath/
	D	N	M	R	C	F	A	P	(ours)	[46]		Datapath	ASIC
bfly	16	8	6	1	4	8	4	6	332,091	132,339,335	9,300	399	35.7
dotv3	9	8	6	1	3	5	3	3	225,518	65,534,780	6,575	291	34.3
dscg	16	8	3	2	4	8	4	6	325,029	116,271,968	9,473	358	34.3
fir4	16	8	1	1	4	8	4	6	307,154	130,971,120	9843	426	31.2
egcd	70	8	2	4	18	35	18	24	3,778,611	22,776,474	10,420	6.02	363
momul	22	8	7	2	6	11	6	8	486,316	11,448,589	7,097	23.5	68.5
median	9	16	1	1	3	5	3	3	194,654	10,733,963	4,420	55.1	44
debug1	6	16	2	3	2	3	2	2	119,286	1,302,928	3,484	10.9	34

Table 4.5: Area results when low-level parameters are computed.

partitioning the fabric between control and datapath is that different user circuits require different amounts of control and datapath; 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 repeated the experiments in Section 4.4.2. 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 is known. Unlike the previous experiments, however, we calculated 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. On the other hand, 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.

Table 4.5 shows the results, using the same columns as in Table 4.4. The size of the fine-grained fabric and the ASIC implementation are copied into Table 4.5 for convenience. 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 were 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 were conducted, and the optimum ratios found, they would tend to improve the results in this section.

As the results in Table 4.5 show, 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 [46] to the area required to implement the same benchmark in our fabric now ranges from 10.9 to 426, 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 31.2 to 363.

4.5 Proof-of-concept layout

As a proof-of-concept, we performed 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. The Verilog description of the fabric was synthesised with Synopsys Design Compiler, targeting the STMicroelectronics $90nm$, 7-layer metal process using the STMicroelectronics CORE90GPSVT standard cell library. The netlist was flattened into a single level of hierarchy before layout. The pre-layout netlist contained a total gate area of 300098 μm^2. The cell placement, cell sizing and repeater insertion was performed by Cadence SoC Encounter. Detailed wire routing was performed using Cadence NanoRoute and was completed with no violations. The total gate area after place and route was 336402 μm^2. The placement region set to approximately 625 μm × 625 μm, resulting in a gate density of 86.1%.

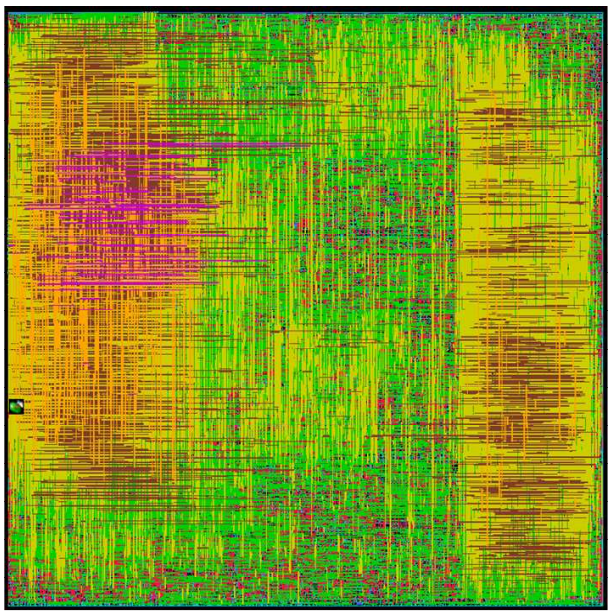


Figure 4.6: Proof-of-concept layout.

4.6 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. This section compares our architecture to several previous studies.

4.6.1 Fine-grained synthesisable fabric

We have compared our architecture to the best synthesisable architecture in Section 4.4.2 using a set of benchmark circuits. The architecture proposed in [46] is fine-grained and the configurability 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 are 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 contributes large area consumption.

For some other circuits which do not have large number of multipliers, the area ratio is between 14 and 75. We observe that the architecture in [46] is not efficient when implementing large circuits. The architecture in [46] 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 large enough to implement the given benchmark circuit, these multiplexers become unwieldy and it causes the area to grow significantly.

4.6.2 Datapath-oriented FPGAs

Several previous studies have considered datapath-oriented FPGAs [33, 36, 39, 47, 48]. In these architectures, configuration bits are shared among multiple lookup-tables and multiple routing switches.

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 unprogrammed fabric. This is a key requirement of a synthesisable architecture. Although these architectures can be synthesised (as in [39]), the combinational loops will require designers to "break" these loops by declaring false paths; this increases the difficulty of including these fabrics in a large FPGA.

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 [47], 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.

4.6.3 Coarse-grained fabrics

Coarse-grained architectures, in which lookup-tables are replaced by ALUs, have also been described in [34, 35, 40, 44]. Of these, the RaPiD architecture [44] was 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 unprogrammed 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 how many of each type of functional unit is 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.

4.7 Summary

We have presented an architecture for a datapath-oriented synthesisable FPGA core which can be used to provide a flexible coarse-grained block on existing island-style FPGA devices.

The proposed architecture features with sharing configuration bits, carry chains, directional routing architecture and embedded multipliers. Compared to the previous best synthesisable embedded programmable logic core, our architecture is between 6 times and 426 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.

Chapter 5

Hybrid Floating Point FPGA

5.1 Introduction

By employing the methodology presented in Chapter 3 and 4, we propose domain-specific coarse-grained architectures which can have advantages in speed, density and power over more conventional heterogeneous FPGAs. One key issue associated with such an approach lies in identifying the correct amount of coarse-grained logic necessary to enhance the performance of an application without adversely affecting area and flexibility. For example, an application that demands high performance floating point computation can potentially achieve better speed and density by introducing dedicated embedded floating point units (FPUs). However, for those applications which do not have any floating point computations, the FPU resources will be wasted. To address this issue, we advocate domain-specific FPGAs with flexible, parameterised architectures that can be generated to address application sets that are smaller than those targeted by conventional FPGAs, but possibly larger than that of ASICs.

We introduce a hybrid FPGA model in which both fine-grained and coarse-grained units are considered important. Given a domain-specific application requirement, a reconfigurable fabric consisting of both types of units is generated, the coarse-grained units being used for the datapath and fine-grained units for control and bit-oriented operations. A model is also

introduced that allows us to search for the best proportion of each type of fabric, and a method for rapidly evaluating the performance of the architecture is employed.

Initial experiments on the proposed generation framework shows promising results. A hybrid FPGA device which is optimised for floating point computations can achieve 2.4 times improvement in speed and 19 times reduction in area on average when compared with traditional FPGA devices on the benchmark circuits introduced in Section 2.5.

5.2 Generic domain-specific hybrid FPGA

In this section, a generic hybrid FPGA architecture is discussed with examples from different application domains. Issues and challenges associated with this architecture are mentioned, and a hybrid FPGA for floating point calculation will be introduced in Section 4. Compared with purely coarse-grained devices, having fine-grained units in the fabric serves to enhance flexibility.

A hybrid FPGA architecture for digital signal processing (DSP) applications may have the following types of reconfigurable blocks:

- Fixed-function blocks for Fast Fourier Transform (FFT) computation.
- Coarse-grained blocks for fused multiply-and-add operations.
- LUT-based fine-grained blocks to implement bit level operations and state machines.

The above example shows three levels of granularity. The interconnection of these blocks can be optimised based on the nature of DSP applications. For example, the routing between blocks can be directional to avoid tristate buffers; buses can be used; and a single configuration bit can be used to control multiple wires on the bus.

Hybrid FPGA based domain-specific architectures can also be developed for networking applications, with various coarse-grained units being devoted to packet/payload/header processing and fine-grained units used for implementing state machines. Routing between blocks could be either bus-based or packet-oriented.

Although a hybrid architecture may improve the overall performance of a domain-specific application, there are challenges and issues that have to be addressed in order to employ this architecture effectively. These include:

- The architecture and granularity of the fabric. Given domain-specific information, how to decide the architecture of a coarse-grained element, and how many levels of granularity should be supported?
- The proportion of each type of fabric. Assuming there are coarse-grained and fine-grained units, what proportion of these units is required so that the device is specialised while flexible enough to accommodate different applications in that domain?
- The interconnection between different fabric. The routing between fabric can be bit-oriented or bus-oriented. Should it be bidirectional or single direction? Which topology can best provide efficient communication between each fabric?
- The design flow. As block with different granularity are available, traditional hardware description language (HDL) based design flows may not be suitable for a hybrid FPGA. We need tools which can choose the best-fit fabric for a given computation, and partition the computation between fine-grained and coarse-grained structures.

In this work, we focus on hybrid FPGA architectures with multiple granularity, and use floating point computations as a case study. The requirements of such an FPGA is discussed in the next section.

5.3 Floating point hybrid FPGA architecture

5.3.1 Requirements

Before we describe the floating point hybrid FPGA 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 is first described. Although the

following analysis is qualitative, it is possible to develop the hybrid model 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 circuits. The datapath occupies most of the area in the form of FPUs. The required processing mainly consists of addition, subtraction and multiplication. Occasionally, square root and division may be required. Floating point adders and multipliers consume a lot of FPGA resources. For instance, a single precision floating point adder requires 297 slices on a Xilinx Virtex 4 device, while a single precision floating point multiplier requires 350 slices on the same device [60]. Although the number of slices can be reduced to 233 and 118 respectively by implementing part of the logic in DSP48s for Xilinx Virtex 4 devices, such an implementation is still expensive as the number of DSP48s is limited.

The floating point precision is usually a constant within an application. The IEEE 754 standard is an overwhelming first choice, especially the single precision format (32-bit) or double precision format (64-bit). The interconnection can be bus-oriented.

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

Based on the above analysis, the following presents some basic requirements for floating point hybrid FPGA architectures.

- 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.

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

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

- 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.

5.3.2 Architecture

Figure 5.1 shows a top-level block diagram of our hybrid FPGA architecture. It employs an island-style fine-grained FPGA structure with dedicated columns for coarse-grained units. The architecture consists of two types of reconfigurable units: fine-grained and coarse-grained, the coarse-grained part having embedded fixed-function floating point adders and multipliers.

The top-level architecture is inspired by existing commercial FPGAs. However, the proportion of coarse-grained blocks can be customised to meet design requirements. The island-style fine-grained block is used to generate different control signals for the coarse-grained units, and fine-grained interconnections such as connection boxes and switch matrices are employed as interconnections.

LUT-based fine-grained units, similar to Xilinx Virtex II slices, are employed. These can be configured to build state machines and to support bit-oriented operations. Since this is a domain-specific FPGA dedicated to floating point computations, it is assumed that the datapath for the floating point units is implemented on the coarse-grained logic.

The architecture of the coarse-grained units, inspired by previous work [51, 59], is shown in Figure 5.2. It is parameterised to support different proportions of fine and coarse-grained elements, the parameters being detailed in Table 5.1. There are D blocks in a unit, P of them are floating multipliers, another P of them are floating point adders and the rest $(D - 2P)$ are wordblocks.

The floating point multiplier block is a fixed-function block. The floating point adder block can be configured for either floating point addition or subtraction. This is achieved by

Symbol	Parameter Description
D	Number of blocks (Including FPUs, wordblocks)
N	Bit Width
M	Number of Input Buses
R	Number of Output Buses
F	Number of Feedback Paths
P	Number of Floating Points Adder and Multipliers

Table 5.1: Architectural parameters for the coarse-grained unit.

XORing the sign bit with the configuration bit. Each FPU has a reconfigurable registered output and associated control input and status output signals. The control signal is a write enable that controls the output register. The status signals report the FPU'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 as routing paths exist between them.

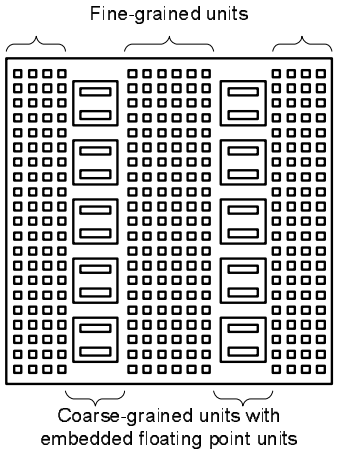


Figure 5.1: Top-level architecture of floating point hybrid FPGA.

A wordblock contains N identical bitblocks, and is similar to published designs [59]. A bitblock contains two 4-input LUTs and a reconfigurable output register. The value of N depends on the size of the FPU. Bitblocks within a wordblock are all controlled by the same

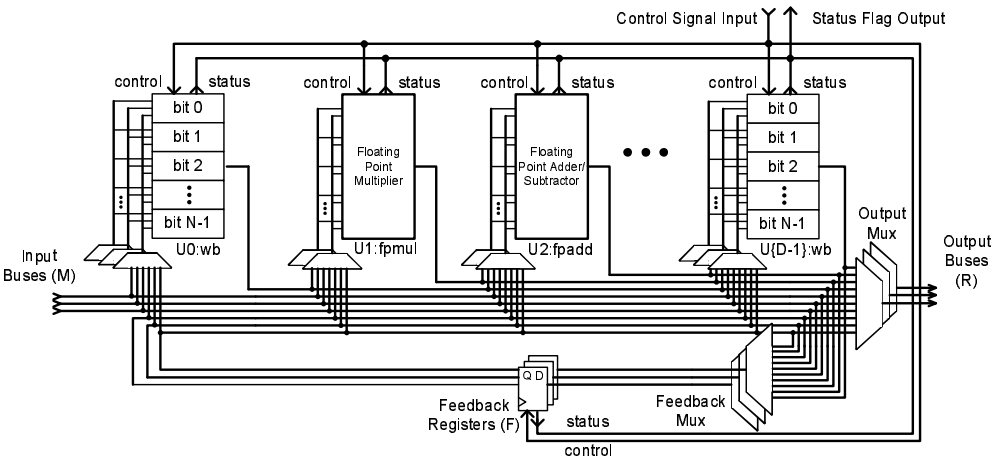


Figure 5.2: Architecture of the coarse-grained unit.

set of configuration bits, so all bitblocks within a wordblock perform the same function. A wordblock, which includes a register, can efficiently implement operations such as addition and multiplexing. Similar to FPUs, wordblocks generate status flags such as MSB, LSB, carry out, overflow and zero which are connected to the fine-grained blocks.

Apart from the control and status signals, there are M input buses and R output buses connected to the fine-grained units. The routing layout assumes that a block 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 input from input buses and feedback registers, while the second block can accept input from input buses, the feedback register and the first block. The feedback registers serve to latch the output a block and forward to another block. The location of a floating point multiplier is always logically located to the left of a floating point adder so that no feedback register is required to support multiply-and-add operations. The coarse-grained units can support multiply-accumulate functions by utilising the feedback registers.

Switches in the coarse-grained unit are implemented using multiplexers and are bus-oriented. A single set of configuration bits is required to control these multiplexers, improving

density. For the same reason, the FPUs are embedded in the coarse-grained units rather than distributed over the FPGA, such that an FPU can exploit the bus-oriented routing resources in the coarse-grained blocks. This can significantly reduce area.

5.3.3 Design flow

As mentioned earlier, a traditional HDL-based design flow is not suitable for the floating point hybrid FPGA, since a suitable partition between the different blocks must be made. A design can be partitioned either from the low-level description or from a high-level description. A methodology has been reported [61] which can extract coarse-grained logic from a netlist, where the netlist is targeted for a fine-grained fabric. Although this method can locate suitable logic to implement in a wordblock, it fails to recognise fixed-function logic such as floating point operations, since their low level representation is usually irregular.

We propose a scheme to partition a given application from a high level description into a reconfigurable fabric with multiple granularity. The general approach is illustrated in Figure 5.3. Some key features of this scheme are:

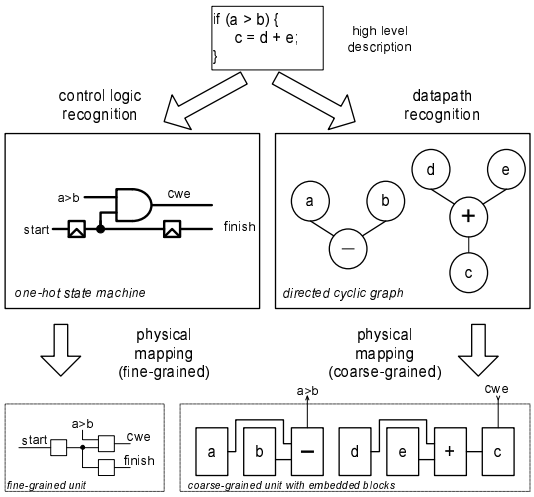


Figure 5.3: A general scheme of hybrid FPGA deign flow.

- Each fixed-function block represents a built-in function or operator in the high level

description. A fixed-function block is instantiated when the function or operator is called.

- Data dependence information can be extracted from the high level description to produce a directed cyclic graph. This graph represents a datapath and it can be mapped to coarse-grained units.

- Control statements such as "if", "while", "for" and control for the execution sequence can be translated to a one-hot state machine [62].

This scheme can be implemented on top of most available high level descriptions for digital design such as Handel-C and ASC [63]. We have modified a high level compilation tool called *fly* [64] to support the floating point hybrid FPGA architecture. The *fly* compiler has the following features:

- There are dedicated operators for floating point computations such as ".+", ".–" and ".*", which map to FPUs in the coarse-grained blocks.

- All variables starting with the letter "f" indicate a bus-oriented signal and will map to the wordblocks in the coarse-grained FPGA fabric.

- All control logic such as loop and conditional statements will be implemented as random logic, and the compiler maps them to the logic cells in the fine-grained FPGA fabric.

- It produces one-hot state machines which can be implemented efficiently on fine-grained units.

There are certain issues in the *fly* compiler that currently prohibit us from producing a bitstream from the high level description. The major issue is that it cannot partition directed cyclic graphs efficiently in the case of multiple coarse-grained blocks. In addition, the *fly* compiler is not aware of some architectural parameters such as the number of feedback registers. Hence designs for the hybrid FPGA are currently partitioned manually.

5.4 Modelling of a hybrid FPGA

A methodology, building on our earlier work [56, 59], is used to model floating point hybrid FPGAs with different architectural parameters and coarse-grained blocks, as described in Section 5.3.2. This methodology is general and can be used to model any FPGA provided that a floorplanner and a timing analysing tool are available for that device. In this methodology, an existing fine-grained commercial FPGA is used. Fine-grained blocks in our hybrid FPGA are directly mapped to the corresponding logic cells on the commercial FPGA.

The area and delay for the embedded coarse-grained units are first estimated by synthesising the design using a standard cell flow. They are then modelled in a commercial FPGA by employing blocks of logic cells with similar delay and area. The corresponding vendor's CAD tools are then used to estimate the delay and area of the hybrid FPGA.

5.4.1 Soft-core embedded floating point units

We employ a parameterised synthesisable IEEE 754 compliant floating point library in our experiments. The library supports four rounding modes and denormalised number. A floating point multiplier and floating point adder are generated and synthesised using a standard cell library design flow. The target process is 130nm and the Synopsys Design Compiler is used for synthesis. During synthesis, retiming optimisation is enabled to obtain better results. Table 5.2 shows the synthesis results for both single precision and double precision standards. It should be noted that there is potential improvement of the core in both the timing and density by adopting full-custom design.

5.4.2 Synthesisable coarse-grained units

To allow parameterised coarse-grained units, we employ a synthesisable flow which supports different granularities. To determine suitable parameters for generation of coarse-grained units, we first decide on an initial set of parameters and try to map a set of benchmark circuits to the units. Two parameters determine whether the architecture is best-fit. The

FP unit	Latency (clock cycle)	Area (μm^2)	Period (ns)	Frequency (MHz)
SP adder	5	45,147	2.23	448.4
SP multiplier	5	93,184	3.05	327.8
DP adder	5	104,606	2.61	383.1
DP multiplier	5	252,241	5.47	182.8

Table 5.2: Timing and area of embedded floating point units. SP stands for single precision and DP stands for double precision.

first is the number of coarse-grained units required to implement the circuit. The second is the percentage of blocks used in a unit.

The best-fit architecture can be determined by varying the parameters to produce a design with the least number of units with maximum density on the benchmark circuits. Surplus wordblocks are added to the design, allowing more flexibility for implementing other circuits outside of the benchmark set. Currently, the *fly* compiler is unable to produce very efficient physical mappings, so manual mappings are made for the benchmarks. Once the parameters are determined, a Verilog netlist is generated and synthesised together with soft-core FPUs using the Synopsys Design Compiler for the 130nm process. Area information can be obtained from the tool directly. Timing information, however, cannot be determined before programming the configuration bits.

During manual mapping, a set of configurations is generated and can be used in timing analysis. We use the case analysis feature provided in the Synopsys Design Compiler which takes configuration bits into account in the timing analysis.

The architectural parameters: 9 blocks ($D = 9$), 4 input buses ($M = 4$), 3 output buses ($R = 3$), 3 feedback registers ($F = 3$), 2 floating point adders and 2 floating point multipliers ($P = 2$) are determined empirically by trial-and-error as explained above. We generate both single and double precision coarse-grained fabrics with bitwidths (N) 32 and 64 respectively.

5.4.3 Integration with fine-grained units

LUT-based fine-grained units are mature in terms of architecture and design flow. They have been widely adopted in commercial FPGAs. We have employed a methodology called virtual embedded blocks (VEB) [56] to model fine-grained units in our architecture. The VEB flow allows the evaluation of embedded elements on FPGA devices by creating dummy logic cells.

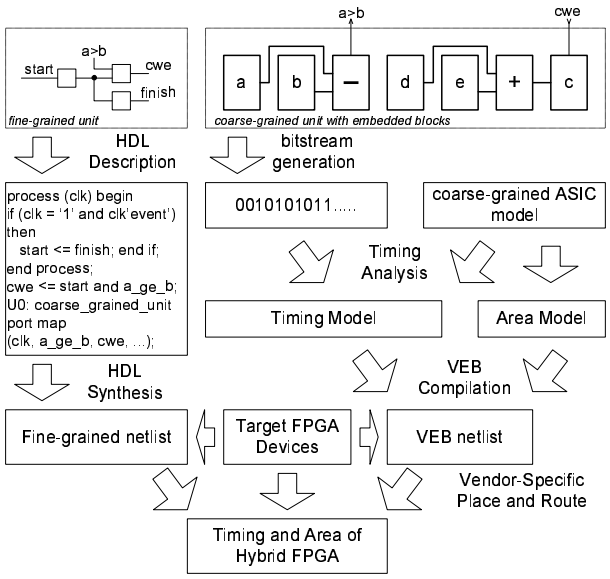


Figure 5.4: Integration of fine-grained and coarse-grained units.

Figure 5.4 illustrates the integration of synthesisable coarse-grained units and fine-grained units on a Virtex II device. This methodology is not limited to any particular device or vendor, and can be used for most commercial FPGA devices. We start the first step with fine-grained logic which is described in HDL. The HDL description also contains additional statements which instantiate the coarse-grained units explicitly, and signals between the fine-grained and coarse-grained units are mapped to appropriate routing resources. The design is then synthesised on the target device and a device specific netlist is generated. The synthesis tool considers the coarse-grained unit as a black box. The area utilisation is computed by determining the corresponding number of slices in Virtex II [65].

The second step is to obtain the timing and area model for each instantiated coarse-grained unit as described in Section 5.4.2. With this information, a VEB netlist can be compiled by generating dummy cells with appropriate area and delay. Special consideration is given to the interface between fine-grained units and coarse-grained units to make sure that the corresponding VEB model has sufficient I/O pins to connect to the fine-grained 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 slice. For example, it is not possible to have a VEB model which has area of 4 slices but demands 33 inputs and 9 outputs, as we assume one slice in Virtex II can only support 8 inputs and 2 outputs. Also, as we cannot route the configuration clock and configuration input pin to a coarse-grained unit, there are two programming pins connected to the I/O of the host FPGA which act as the configuration port for the coarse-grained unit.

After generating the VEB netlist for the targeted FPGA, a constraint file which forces the VEB located on the same column is specified. We then use the vendor's place and route tool to obtain the final area and timing results. This represents the characterisation of a circuit implemented on the hybrid floating point FPGA with fine-grained units and routing resources exactly the same as the targeted FPGA.

Using commercial FPGA fine-grained units in this manner has several advantages, since commercial quality synthesis and place and route tools can be in the modelling of the hybrid FPGA. It can produce a more realistic comparison to existing FPGA devices. Furthermore, optimisations such as retiming are available. We avoid redesigning the fine-grained unit, so adopting a rapid evaluation approach based on existing commercial fine-grained units.

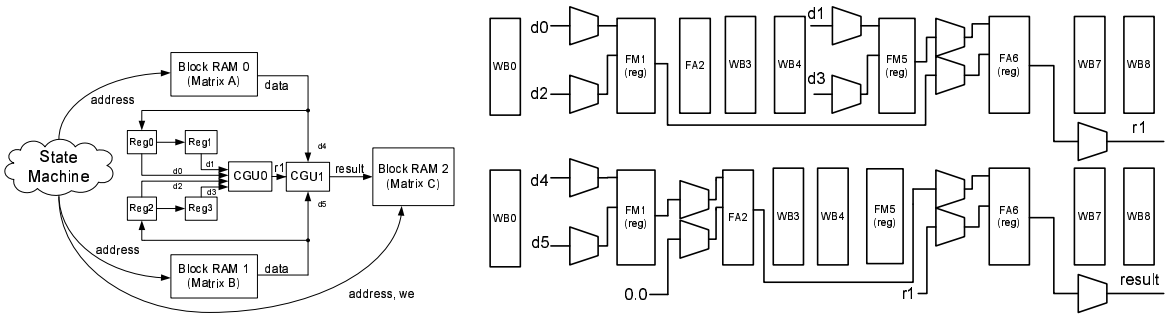
5.5 Results

A set of benchmark applications are mapped to the floating point hybrid FPGA, and the results are compared to a Virtex II device. This Section introduces the circuits and gives an example of mapping one of the circuits. Both single precision and double precision floating point hybrid FPGA are assessed. A floorplan of one of the example mappings on the hybrid

FPGA is given in Figure 5.6. All FPGA results are obtained using the Synplicity Synplify Premier 8.5 to synthesis and Xilinx ISE 8.1i design tools to place and route. All ASIC results are obtained using Synopsys Design Compiler V-2004.06.

Six benchmark circuits are used in this study as mentioned in Section 2.5 All the examples assume the two floating point multipliers are located at the second and the sixth block. The two floating point adders are located in the third and the seventh block. All other parameters are given in Section 5.4.2.

5.5.1 Example mapping



(a) Fine-grained unit mapping.

(b) Coarse-grained unit mapping.

Figure 5.5: Example mapping for matrix multiplication.

Figure 5.5 illustrates a manual mapping example for the matrix multiplication. The mapping process splits the circuit into a control unit and datapath, and they are respectively mapped to the fine-grained units and coarse-grained units. A state machine is implemented using the fine-grained units which generate address and write enable signals for the block RAM. The output of the block RAM connects to either registers or coarse-grained blocks, according to the timing requirement of the datapath. Registers are added to match the latency so that all multiplications take place in the same clock cycle.

Two coarse-grained units are instantiated. 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 exactly the same latency as the one in the first coarse-grained block. This demonstrates an alternative use of a coarse-grained unit. Finally $r1$ and $d4 \times d5$ are added together and the state machine fetches the result to the block RAM. All FPUs have enabled registered output to further pipeline the datapath.

5.5.2 Comparison with existing FPGA devices

In this paper, we choose the Virtex II device as the host FPGA for the floating point hybrid FPGA and comparisons are made between them. This is because Virtex II employs a comparable technology process (0.15μm/0.12μm) and its fine-grained units and routing resources can efficiently implement random logic.

The physical die area of a Virtex II device has been reported [65], and the normalisation of the area of coarse-grained unit is estimated in Table 5.3. We assume that 60% of the total die area is used for slices. Others are for I/O pads, block memory, multiplier etc. This means that the area of a Virtex II device is $10{,}912\mu\text{m}^2$. This number is normalised against the feature size (0.15μm). A similar calculation is used for the coarse-grained units. The area of a single precision coarse-grained unit is $567{,}146\mu\text{m}^2$ and the area of a double precision coarse-grained unit is $1{,}256{,}570\mu\text{m}^2$ are reported by the synthesis tool. We further assume 15% overhead after place and route the design based on our experience [59]. The area values are normalised against the feature size (0.13μm). The number of equivalent slices is obtained through the division of coarse-grained unit area by slice area. This shows that single precision coarse-grained unit is equivalent to 80 slices and our double precision coarse-grained unit is equivalent to 177 slices. On checking the input/output requirements, we find that the single precision coarse-grained unit cannot fit into 80 slices as it requires 162 outputs. Therefore, the number of slices of the fine-grained unit is increased to 81.

Although a Virtex II slice employs smaller transistors (0.12μm) than those used for building the coarse-grained unit (0.13μm), we do not scale the timing of the coarse-grained unit and therefore conservative timing results are reported.

Fabric	Area (A) (μm^2)	Feature Size (L) (μm)	Normalised Area (A/L^2)	Area in Slices	Input Pin	Output Pin
Virtex II Slice	10,912	0.15	485,013	1	8(8)	2(2)
SP-CGU	567,146	0.13	38,592,775	80	157 (316)	162(160)
DP-CGU	1,256,570	0.13	85,506,242	176	285 (704)	258(352)

Table 5.3: Normalisation on the area of the coarse-grained units against a Virtex II slice. The values in the sixth and seventh columns represent the number of I/O required. The values in brackets indicate the maximum number of I/O allowed for the area in slices. SP and DP stand for single and double precision respectively. CGU stands for coarse-grained unit.

We use XC2V1000-6-FF896 as the host-FPGA for the floating point hybrid FPGA. There are 12 single precision coarse-grained blocks, embedded into this FPGA using VEB flow. The coarse-grained blocks constitute 19% of the total area in an XC2V1000 device. The VEB result is generated according to Section 5.4.2. Benchmark circuits are implemented on the same device and the results are shown in Table 5.4a. A sample floorplan of the *bgm* circuit implemented on a hybrid FPGA is illustrated in Figure 5.6.

The FPU values for the XC2V1000 device (seventh column) are estimated from the distribution of LUTs, which is reported by the 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. Although the *bfly* circuit cannot fit in a XC2V1000 device, it can be tightly packed into a few coarse-grained blocks. For example, the circuit *bfly* has 8 FPUs which consume 129% of the total FPGA area. They can fit into 2 coarse-grained units, which constitute just 3.2% of the total FPGA area. Delay is reduced by 2.4 times on average. As the critical

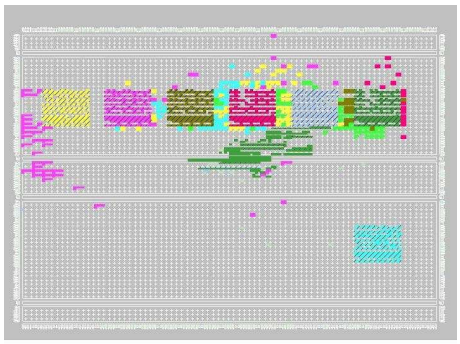


Figure 5.6: Floorplan of the *bgm* circuit on a hybrid FPGA. A coarse-grained unit is identified by tightly packed slices in a rectangular region.

paths are in the FPU, improving the timing of the FPU through full-custom design would further increase the overall performance.

Table 5.4b gives the implementation results for double precision floating point circuits. The hybrid FPGA is compared with an XC2V3000-6-FF1152 device. Similar reduction in area is found and delay reduction is slightly better than the single precision versions. The geometric means for reduction in area and in delay of all the circuits are 19 and 2.4 respectively.

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 1 coarse-grained unit and 1324 slices, while the delay is increased from 4.8ns to 10.2ns. 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 units, while another multiplication and the square root operation can be implemented on fine-grained units. The resulting circuit requires one coarse-grained unit and 1311 slices and the delay is 11.42ns. The alternative, with two coarse-grained units and 732 slices, has delay of 5.21ns.

	Single precision floating point hybrid FPGA				XC2V1000-6-FF896				Reduction	
Circuit	CGU area (slices)	FGU area (slices)	Total Area (slices)	Delay (ns)	FPU area (slices)	Logic area (slices)	Total Area (slices)	Delay (ns)	Area (times)	Delay (times)
bfly*	162 (3.2%)	102 (1.99%)	264 (5.2%)	5.54	6,615 (129%)	560 (11%)	7,175 (140%)	12.76	27.2	2.30
dscg	162 (3.2%)	147 (2.87%)	309 (6.0%)	4.72	4,548 (89%)	209 (4%)	4,757 (93%)	11.47	15.4	2.43
fir4	162 (3.2%)	4 (0.08%)	166 (3.2%)	4.80	4,736 (93%)	102 (2%)	4,838 (94%)	13.74	29.1	2.86
mm3	162 (3.2%)	131 (2.56%)	293 (5.7%)	5.09	4,017 (78%)	507 (10%)	4,524 (88%)	11.63	15.4	2.28
ode	162 (3.2%)	32 (0.63%)	194 (3.8%)	5.29	3,709 (72%)	159 (3%)	3,868 (76%)	10.73	19.9	2.03
bgm*	567 (11.1%)	368 (7.19%)	935 (18.3%)	7.40	13,801 (270%)	257(5%)	14,058(275%)	13.42	15.0	1.81
								Geometric Mean:	19.6	2.26

(a) Single Precision Floating Point hybrid FPGA Result. Values in brackets indicate the percentages of slices used in a XC2V1000 device. *Circuit *bfly* cannot be fitted in a XC2V1000 device. The area and the delay are obtained by implementing on a XC2V1500 device. Similar case applies to *bgm* circuit and the results are obtained by implementing on a XC2V3000 device.

	Double precision floating point hybrid FPGA				XC2V3000-6-FF1152				Reduction	
Circuit	CGU area (slices)	FGU area (slices)	Total Area (slices)	Delay (ns)	FPU area (slices)	Logic area (slices)	Total Area (slices)	Delay (ns)	Area (times)	Delay (times)
bfly	352 (2.5%)	213 (1.49%)	565 (3.9%)	9.02	12,813 (89%)	920 (6%)	13,733 (96%)	24.57	24.3	2.72
dscg	352 (2.5%)	309 (2.16%)	661 (4.6%)	10.11	9,287 (65%)	327 (2%)	9,614 (67%)	22.78	14.5	2.25
fir4	352 (2.5%)	19 (0.13%)	371 (2.6%)	9.06	11,143 (78%)	147 (1%)	11,290 (79%)	23.68	30.4	2.61
mm3	352 (2.5%)	290 (2.02%)	642 (4.5%)	8.9	8,071 (56%)	818 (6%)	8,889 (62%)	23.40	13.8	2.63
ode	352 (2.5%)	193 (1.35%)	545 (3.8%)	9.74	7,933 (55%)	305 (2%)	8,238 (57%)	21.93	15.1	2.25
bgm*	1232 (8.6%)	578 (4.03%)	1,810 (12.6%)	10.00	29,758 (208%)	539(4%)	30,207 (211%)	24.34	16.7	2.43
								Geometric Mean:	18.3	2.48

(b) Double precision floating point hybrid FPGA results. Values in the bracket indicate the percentages of slices used in a XC2V3000 device. *Circuit *bgm* cannot be fitted in a XC2V3000 device. The area and the delay are obtained by implementing on a XC2V6000 device.

Table 5.4: Floating point hybrid FPGA results. CGU stands for coarse-grained unit and FGU stands for fine-grained unit.

5.5.3 Comparison with previous work

Modelling of embedded FPUs based on island-style FPGAs has been reported [53, 56] and the improvements observed are summarised in Table 5.5. This work adopts a more generic model in which a number of FPUs and LUTs form a 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 [53] are very different and a direct comparison

Architecture	FPGA Model	FPU Model	Area Saving (times)	Delay Reduced (times)
[53]	VPR	full-custom (Pentium 4)	2.2	1.4
[56]	VEB	full-custom (Blue Gene)	3.7	4.4
This work	VEB	synthesisable	19	2.4

Table 5.5: Comparison to previous work. Area saving and delay reduced are compared to a FPGA device with embedded multiplier.

cannot be made.

As similar circuits are used in [56], a more detailed analysis is performed. Table 5.6 presents the area distribution and the reported delay between this work and [56]. One significant reduction is the size of FPU. In [56], 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 [56], such logic is implemented using fine-grained resources. The delay reported in [56] is better than this work because of the full-custom FPU. In addition, the latency in [56] is one clock cycle which can give better performance for some applications.

5.6 Summary

This chapter demonstrates the feasibility of developing automated tools for producing hybrid FPGAs that are tuned to specialised classes of problems. A generic architecture is given, together with a specific example involving an FPGA optimised for floating point computa-

	This work			Embedded FPU		
	CGU area (slices)	FGU area (slices)	Delay (ns)	FPU area (slices)	FGU area (slices)	Delay (ns)
bfly	352	213	9.02	2,280	1,712	8.82
dscg	352	309	10.11	1,710	470	8.81
fir4	352	19	9.06	1,995	498	9.54
mm3	352	290	8.90	1,425	1,195	8.59
ode	352	193	9.74	1,425	435	8.53

Table 5.6: Comparison to previous embedded FPU model [56] for double precision floating point benchmarks.

tions. We show that the proposed floating point hybrid FPGA enjoys improved speed and density over a conventional FPGA for a variety of applications.

Chapter 6

Conclusion and Future Work

This report discusses FPGA devices optimised for floating point computations. The objective and the requirements of the research is covered in Chapter 1. A comprehensive literature review is illustrated by Chapter 2, which covers different aspect of this project, including FPGA architecture, FPGA modelling, CAD tools, floating point computations. Different FPGA models are presented in Chapter 3 and 4. The models provide a potential automated framework to search for an optimised floating point FPGA architecture. Chapter 5 integrates previous work and proposes an initial floating point FPGA architecture which consists of coarse-grained units and fine-grained units. The architecture can deliver promising results on a set of selected floating point benchmark circuits.

Plenty of future work can be done to finalise the project. The goal is to automate the floating point FPGA design process by profiling user applications. The tasks include (1) automatic synthesis tools which translate high level description into bitstream, (2) more floating point benchmark circuits to evaluate the floating point FPGA design, as well as (3) a search algorithm which produces an FPGA architecture by providing examples of floating point applications. To achieve (1), we will extend the current fly compiler, as described in Section 5.3.3. More complex floating point computations such as LU decomposition, 64-point FFT and BLAS will be implemented on the floating point FPGA. Different search algorithms, including those in [50], will be considered to identify suitable architectural parameters.

The thesis outline is illustrated below. Italic text in the outline represents the current progress.

Chapter 1 Introduction

Chapter 2 Related Work

(Update content, estimated completion date: DEC-2007 – MAR-2008)

Section 2.1 FPGA architecture

Section 2.2 FPGA design tools

Section 2.3 Floating point application

Section 2.4 Floating point units

Section 2.5 Benchmark circuits

Section 2.6 Summary

Chapter 3 Virtual Embedded Block

(Most of the content completed, as illustrated in Chapter 3 of this report)

Section 3.1 Methodology: generic aspects

Section 3.2 Methodology: vendor specific aspects

Section 3.3 Results

Section 3.4 Summary

Chapter 4 Synthesisable Datapath FPGA Fabric

(Most of the content completed, as illustrated in Chapter 4 of this report)

Section 4.1 Architecture

Section 4.2 Example mapping

Section 4.3 Parameter optimisation

Section 4.4 Mapping results

Section 4.5 Proof-of-concept layout

Section 4.6 Comparison to previous work

Section 4.7 Summary

Chapter 5 Hybrid Floating Point FPGA

(Most of the content completed, as illustrated in Chapter 5 of this report)

Section 5.1 Generic domain-specific hybrid FPGA

Section 5.2 Floating point hybrid FPGA architecture

Section 5.3 Modelling a hybrid FPGA

Section 5.4 Results

Section 5.5 Summary

Chapter 6 Synthesis flow of hybrid FPGA

(Future work, estimated completion date: AUG-2007 – NOV-2007)

Section 6.1 High level description of floating point circuit

Section 6.2 Language support

Section 6.3 Synthesis algorithm

Section 6.4 Implementation

Section 6.5 Results

Section 6.6 Summary

Chapter 7 Automatic Floating Point FPGA Generation

(Future work, estimated completion date: APR-2008 – JUN-2008)

Section 7.1 Application profiling

Section 7.2 Search algorithm

Section 7.3 Implementation

Section 7.4 Results

Section 7.5 Summary

Chapter 8 Conclusion

Starting at early February, I plan to have an internship at Xilinx Inc for 6 months to learn the most state-of-the-art technology from the industry. This can definitely improve my current proposed FPGA architecture model. After the internship, my research plan in next 18 month is listed below:

- **AUG-2007 – NOV-2007:** Develop high level synthesis algorithm for the hybrid FPGA architecture. This task will contribute to the Chapter 6 of the thesis.
- **DEC-2007 – MAR-2008:** Implement more complex floating point applications, in-

cluding LU decomposition, principal components analysis and so on. This task will contribute to the Chapter 2 of the thesis.

- **APR-2008 – JUN-2008:** Develop an automatic hybrid FPGA architecture generation algorithm. This task will contribute to the Chapter 7 of the thesis.
- **JUL-2008 – NOV-2008:** Collect all the results and thesis write up.

Bibliography

[1] A. Gara et. al, "Overview of the Blue Gene/L system architecture". In *IBM J. Res & Dev.*, Volume 49, No. 2/3, pp. 195–212, March/May, 2005.

[2] "TOP500 List of World's Fastest Supercomputers". In *http://www.top500.org/news/articles/article_68.php*, June, 2005.

[3] K.D. Underwood and K.S. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating point BLAS performance". In *Proc. FCCM*, pp. 219–228, 2004.

[4] Y. Dou, S. Vassiliadis, G.K. Kuzmanov, and G.N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication". In *Proc. FPGA*, pp. 86–95, 2005.

[5] A. Jaenicke and W. Luk, "Parameterised floating-point arithmetic on FPGAs". In *Proc. IEEE Int. Conf. on Acoust., Speech and Signal Process*, pp. 897–900, May 2001.

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

[7] C.H. Ho, M.P. Leong, P.H.W. 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.

[8] G.L. Zhang, P.H.W. Leong, C.H. Ho, et. al, "Reconfigurable Acceleration for Monte Carlo based Financial Simulation". In *Proc. FPT*, pp. 215–222, Dec 2005.

[9] C.H. Ho, K.H. Tsoi, H.C. Yeung, Y.M. Lam, K.H. Lee, P.H.W. Leong, R. Ludewig, P. Zipf, A.G. Ortiz, M. Glesner, "Arbitrary Function Approximation in HDLs with application to the N-Body Problem". In *Proc. of Field Programmable Technology*, pp. 84–91, 2003.

[10] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research". In *Proc. FPL*, pp. 213–222, 1997.

[11] C.H. Ho, P.H.W. Leong, K.H. Lee, K.H. Tsoi, R. Ludewig, P. Zipf, A.G. Ortiz and M. Glesner, "*fly* - A Modifiable Hardware Compiler". In *Proceedings of Field Programmable Logic and Applications*, pp. 381–390, 2002.

[12] K.H. Tsoi, C.H. Ho, H.C. Yeung and P.H.W. Leong, "An Arithmetic Library and its Application to the N-body Problem". In *Proceedings of Field-Programmable Custom Computing Machines*, pp. 68–78, 2004.

[13] J. Rose, "Hard vs. soft: The central question of pre-fabricated silicon," in *34th International Symposium on Multiple-Valued Logic (ISMVL'04)*, pp. 2–5, May 2004.

[14] K. Leijten-Nowak and J. L. van Meerbergen, "An FPGA architecture with enhanced datapath functionality," in *Proc. FPGA '03*, ACM Press, 2003, pp. 195–204.

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

[16] M. Beauchamp, S. Hauck, K. Underwood, and K. Hemmert., "Embedded floating point units in FPGAs," in *Proc. FPGA '06*, ACM Press, 2006.

[17] Saab Ericsson Space AB European Space Agency Contract Report, *Application-like Radiation Test of XTMR and FTMR Mitigation Techniques for Xilinx Virtex-II FPGA*. https://escies.org/public/radiation/esa/database/-ESA_QCA0415S_C.pdf, 2005.

[18] 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.

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

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

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

[22] J. Hull, *Options, futures and other derivatives*, 5th ed. Prentice-Hall, 2002.

[23] G. Zhang, P. Leong, C. H. Ho, K. H. Tsoi, C. Cheung, D.-U. Lee, R. Cheung, and W. Luk, "Reconfigurable acceleration for Monte Carlo based financial simulation," in *Proc. ICFPT*, pp. 215–222, 2005.

[24] R. Usselmann, *Floating Point Unit*. http://www.opencores.org/project.cgi/web/fpu/overview, 2005.

[25] N. Y. ANSI/IEEE, *IEEE Standard for Binary Floating-Point Arithmetic*, The Insittution of Electrical and Electronic Engineering, Inc, Tech. Rep., 1985, IEEE Std 754-1985.

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

[27] 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.

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

[29] A. Bright et. al., "Blue Gene/L compute chip: synthesis, timing, and physical design," *IBM J. Res & Dev.*, vol. 49, no. 2/3, pp. 277–287, March/May 2005.

[30] C. Wait, "IBM PowerPC 440 FPU with complex-arithmetic extensions," *IBM J. Res & Dev.*, vol. 49, no. 2/3, pp. 249–254, March/May 2005.

[31] K. Padalia, R. Fung, M. Bourgeault, A. Egier, and J. Rose, "Automatic transistor and physical design of FPGA tiles from an architectural specification," in *Proc. FPGA '03*, ACM Press, pp. 164–172, 2003.

[32] D. Chen, J. Cong, M. Ercegovac, and Z. Huang. Performance-driven mapping for CPLD architectures. In *ACM Int. Symp. on Field-Programmable Gate Arrays*, pages 39–47, Feb. 2001.

[33] D. Cherepacha and D. Lewis. DP-FPGA: An FPGA architecture optimized for datapaths. In *Int. Conf. on VLSI Design*, pages 329–343, 1996.

[34] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Twentieth Anniversary Conf. on Advanced Research in VLSI*, page 23, 1999.

[35] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[36] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimera Reconfigurable functional unit. *IEEE Trans. on VLSI*, 12(2):206–217, Feb. 2004.

[37] C. Ho, P. Leong, W. Luk, S. Wilton, and S. Lopez-Buedo. Virtual embedded blocks: A methodology for evaluating embedded elements in FPGAs. In *Int. Symp. on Field-Programmable Custom Computing Machines*, pages 35–44, Apr. 2006.

[38] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Int. Symp. on Field-Programmable Gate Arrays*, pages 21–30, Feb. 2006.

[39] K. Leijten-Nowak and J. L. van Meerbergen. An FPGA architecture with enhanced datapath functionality. In *Int. Symp. on Field-Programmable Gate Arrays*, pages 195–204, Feb. 2003.

[40] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *ACM Int. Symp. on Field-Programmable Gate Arrays*, pages 135–143, Feb. 1999.

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

[42] B. Quinton and S. Wilton. Post-silicon debug using programmable logic cores. In *Int. Conf. on Field-Programmable Technology*, pages 241–247, Dec. 2005.

[43] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, June 2006.

[44] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves. Morphosys: An integrated reconfigurable system for data-parallel and compute intensive applications. *IEEE Trans. on Computers*, 49(5):465–481, April 2000.

[45] 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*, 40(2):485–497, Feb. 2005.

[46] A. Yan and S. Wilton. Product-term based synthesizable embedded programmable logic cores. *IEEE Trans. on VLSI*, 14(5):474–488, May 2006.

[47] A. Ye and J. Rose. Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits. In *Int. Symp. on Field-Programmable Gate Arrays*, pages 3–13, Feb. 2005.

[48] A. Ye, J. Rose, and D. Lewis. Architecture of datapath-oriented coarse-grain logic and routing for FPGAs. In *IEEE Custom Integrated Circuits Conf.*, pages 61–64, Sept. 2003.

[49] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proc. FPGA*. New York, NY, USA: ACM Press, 2006, pp. 21–30.

[50] K. Compton and S. Hauck, "Totem: Custom Reconfigurable Array Generation," in *Proc. FCCM*, pp. 111–119, 2001.

[51] A. Ye and J. Rose, "Using Bus-Based Connections to Improve Field-Programmable Gate-Array Density for Implementing Datapath Circuits," *IEEE Trans. VLSI*, vol. 14, no. 5, pp. 462–473, 2006.

[52] E. Roesler and B. Nelson, "Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture," in *Proc. FPL*, 2002, pp. 637–646.

[53] M. Beauchamp, S. Hauck, K. Underwood, and K. Hemmert, "Embedded floating-point units in FPGAs," in *Proc. FPGA*, 2006, pp. 12–20.

[54] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *Proc. FPL*, 1997, pp. 213–222.

[55] A. Yan and S. Wilton, "Product-Term Based Synthesizable Embedded Programmable Logic Core," *IEEE Trans. VLSI*, vol. 14, no. 5, pp. 474–488, 2006.

[56] C. Ho, P. Leong, S. W. Luk, and S. Lopez-Buedo, "Virtual Embedded Blocks: A Methodology for Evaluating Embedded Elements in FPGAs," in *Proc. FCCM*, 2006, pp. 35–44.

[57] E. Ahmed and J. Rose, "The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density," *IEEE Trans. VLSI*, vol. 12, no. 3, pp. 288–298, March 2004.

[58] B. Mei and S. Vernalde and D. Verkest and H.D. Man and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *Proc. FPL*, 2003, pp. 61–70.

[59] S. Wilton, C. Ho, P. Leong, W. Luk, and B.Quinton, "A Synthesizable Datapath-Oriented Embedded FPGA Fabric," in *Proc. FPGA*, 2007 (in press).

[60] Xilinx Inc., *Floating-Point Operator v1.0*. Product Specification, 2005.

[61] A. Ye, J. Rose, and D. Lewis, "Synthesizing Datapath Circuits for FPGAs with Emphasis on Area Minimization," in *Proc. ICFPT*, 2002, pp. 219–226.

[62] I. Page, "Constructing hardware-software systems from a single description," *The Journal of VLSI Signal Processing*, vol. 12, no. 1, pp. 87–107, 1996.

[63] O. Mencer, "ASC: a stream compiler for computing with FPGAs," *IEEE Trans. CAD*, vol. 25, no. 9, pp. 1603–1617, 2006.

[64] C. Ho, P. Leong, K. H. Tsoi, R. Ludewig, P. Zipf, A. Ortiz, and M. Glesner, "Fly - a modifiable hardware compiler," in *Proc. FPL*. LNCS 2438, Springer, 2002, pp. 381–390.

[65] 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.

[66] J. Hull, *Options, futures and other derivatives*, 5th ed. Prentice-Hall, 2002.

[67] A. Ye, J. Rose, and D. Lewis, "Architecture of datapath-oriented coarse-grain logic and routing for FPGAs," in *CICC '03: Proceedings of the IEEE Custom Integrated Circuits Conference*, pp.61–64, 2003.

[68] L. Beck, "A Place-and-Route Tool for Heterogeneous FPGAs", in *relax Distributed Mentor Project Report*, Cornell University, 2004.

[69] I. Kuan and J. Rose, "Measuring the Gap between FPGAs and ASICs", in *Proc. FPGA '06*, ACM Press, pp. 21–30, 2006.

[70] V. Aken'Ova, G. Lemieux and R. Saleh, "An Improved "Soft" eFPGA Design and Implementation Strategy", in *Proc. of IEEE Custom Integrated Circuits Conference*, pp. 179–182, 2005.

[71] K. Compton and S. Hauck, "Flexibility Measurement of Domain-specific Reconfigurable Hardware", in *Proc. FPGA '04*, ACM Press, pp. 155–161, 2004.

[72] K.H. Tsoi, C.H. Ho, H.C. Yeung and P.H.W. Leong, "An arithmetic library and its application to the N-body problem", in *Proc. FCCM*, pp 68–78, 2004.

[73] C.H. Ho, K.H. Tsoi, H.C. Yeung, Y.M. Lam, K.H. Lee, P.H.W. Leong, R. Ludewig, P. Zipf, A.G. Ortiz and M. Glesner, "Arbitrary Function Approximation in HDLs wth Application to the N-body Problem"

[74] K.D. Underwood and K.S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance", in *Proc. FCCM*, pp 219–228, 2004.

[75] K.D. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance", in *Proc. FCCM*, pp 219–228, 2004.

[76] 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 *Proc. FPGA*, pp 50–55, 2002.

[77] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C.C.Cheung, D.-U. Lee, R.C.C.Cheung, W. Luk, "Reconfigurable acceleration for Monte Carlo based financial simulation", in *Proc. FPT*, pp 215–222, 2005.

[78] 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", in *Proc. FPL*, pp 29–34, 2006.

[79] L. Zhuo and V.K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on FPGAs", in *Proc. FCCM*, pp 26–30, 2004.

[80] G.R. Morris, V.K. Prasanna, "An FPGA-based floating-point Jacobi iterative solver", in *Proc. Parallel Architectures, Algorithms and Networks*, 2005.

[81] Celoxica Limited, "Handel-C Language Reference Manual", in *Product Documentation*, 2004.