A VECTOR FLOATING POINT PROCESSING UNIT DESIGN

SHI CHEN

A Vector Floating Point Processing Unit Design

by

© Shi Chen

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Engineering.

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

May 14, 2008

St. John's Newfoundland

Abstract

The main contribution of this thesis is the successful development of a vector floating point processing unit for high accuracy science computing. For these numerically-intensive applications, vector processing offers simple and straightforward parallelism by executing mathematical operations on multiple data elements simultaneously. The simple control and datapath structures of vector processing enable the embedded computing system to attain high performance at low power.

This vector floating point processing unit includes: a vector register file, vector floating point arithmetic units, and vector memory units. The central module, a vector register file, is divided into twelve lanes. One lane contains 16 vector registers, each including 32×32-bit elements, and is connected to a floating point adder and a floating point multiplier. By modeling the multi-port register file using configurable block RAM on Field Programmable Gate Arrays (FPGA) target, vector register files can efficiently obtain data from external memory and feed data to different arithmetic units simultaneously. Utilizing the quick carry out path and embedded multiplier macro unit, the vector floating point arithmetic units can run at over 200 MHz. A flag register is used to indicate the calculation sequence for the specific computing model. Moreover, the embedded Power PC processor not only can easily control the calculation flow, but also can support an embedded operating system to extend a broad range of applications. The prototype is implemented on Xilinx Virtex II

Pro devices, and a peak performance of 4.530 GFLOPS at 188.768 MHz has been achieved.

First, we present a brief introduction to the floating point arithmetic operations, including addition, multiplication, and multiplier-adder-fused. Second, the architecture of the vector processing unit and a detailed description of vector function units are introduced. Moreover, for a specific computing application, the appropriate overlap execution scheme is discussed. In the end, the performance of each component is analyzed, and the time and area analysis of whole system is provided.

Acknowledgements

I first thank my supervisors, Dr. Ramachandran Venkatesan and Dr. Paul Gillard, for their strict academic training and for their advice and encouragement while I pursued my interest in digital design. Their invaluable expertise greatly helps me to improve my work. I also want to thank Dr. Phillip Bording for research support.

The Computer Engineering Research Laboratory (CERL) in Memorial University of Newfoundland provided a great environment for much of this work based on FPGA platform. Every graduate student in our laboratory always likes to selflessly help each other. Special thanks to Tianqi Wang, Ling Wu, Pu Wang, Liang Zhang, Shenqiu Zhang, Tao Bian, Rui He, and Guan Wang for their precious friendship and generous support. I would like to thank all of you who have given your time, assistance and patience so generously. Extra special thanks to Reza Shahidi, the former laboratory administrator, for his large responsibility for the outstanding computer facilities we have in CERL.

I am also indebted to my parents and my wife, Li Liu. Thank you for your unending support, willingness to accept and eagerness to love.

Finally, thanks to Canadian Microelectronics Corporation (CMC) for providing Xilinx FPGA develop kits. Thanks to Natural Sciences and Engineering Research Council of Canada (NSERC) for research funding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

From the early 1960s, the vector processing model was used to cope with data-intensive application in the scientific computing area. The basic concept of vector processing is fairly simple and straightforward. A large number of arithmetic units (or co-processors) are used to execute mathematical operations on multiple data elements simultaneously. Although many supercomputers initially utilized vector processor and continually broke the performance record through the 1980s and into the 1990s, the scalar microprocessor-based systems swiftly replaced vector machines in the early 1990s, because they approached or even exceeded the performance of vector supercomputers at much lower costs. However, the vector architecture recently reemerges in some today's commodity CPU designs, such as IBM's Cell processor, because the data parallelism is always an efficient scheme for the numerically-intensive applications.

As the number of transistors on integrated circuits has increased rapidly, an efficient and cheap vector processor can be implemented using the advanced silicon CMOS fabrication technology. Field Programmable Gate Arrays (FPGAs) is one of these mature technologies. The modern FPGAs not only can easily implement com-

plex logic functions, but also provide rich macro function units, such as digital signal processing modules, on-chip block memory, input/output (I/O) controllers, or even a complete scalar processor. Moreover, most or all of the functions of a complete digital electronic system can be implemented on this powerful FPGA chip. Utilizing mature CMOS technology, the specific algorithms are vectorized, optimized, and implemented on FPGA platform. For general-purpose applications, the vector-thread (VT) architectural paradigm [1] has become important for all-purpose embedded computing. VT architectures unify the vector and multithreaded execution models. A large amount of structured parallelism can be implemented on VT architectures. The simple control and datapath structures of vector processing enable the embedded computing system to attain high performance at low power. In this thesis, a vector floating point processing unit is implemented on Xilinx FPGAs.

This chapter will introduce the general concepts, and discuss a suitable parallelism. The appropriate implementation method will be also discussed in the end of this chapter.

Section 1.1 reviews floating point number representation and shows the main advantages of IEEE 754 standard for floating point operations.

Section 1.2 compares various forms of microprocessor parallelism methods, including instruction-level parallelism (Pipeline), thread-level parallelism (Out-of-Order), and vector data parallelism.

Section 1.3 analyzes different implementation methods, including Application Specific Integrated Circuit (ASIC) and FPGA. Although the ASIC implementation will be faster and more energy-efficient, the FPGA platform leads to a more flexible solution and can be easily built as an embedded system for a much wider range of applications.

Section 1.4 summarizes the main challenges in our research work, and presents

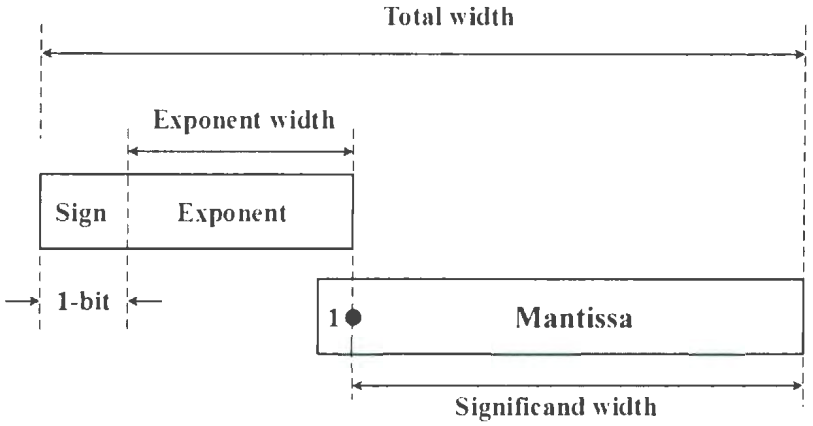

Figure 1.1: Floating point number structure

the organization of this thesis.

1.1 Basic Concepts of Floating Point Number

Floating point operations not only are the basis for scientific and engineering computation, but also are ubiquitous in 3D applications. The most common floating point number representation is the exponent-mantissa method. In this way, a wider range of numbers can be represented compared with the fixed point representation. In this chapter, the IEEE 754 floating point number standard [2] will be introduced and the main advantages of this representation format will be discussed.

Following the IEEE 754 standard, floating point numbers represent a subset of real numbers using three parts: a sign bit, an exponent part, and a mantissa part. Figure 1.1 shows the structure. For different application requirements, IEEE 754 standard defines four closely-related formats: single precision, double precision, single-extended precision, and double-extended precision. For four formats, the main difference is the width of the exponent part and the fraction part, and the hidden bit is also different.

Table 1.1: IEEE 754 Floating Point Number Formats

Formats	Total bits	Sign	Exponent	Significand	Hidden bit
Single	32	1	8	23	1
Double	64	1	11	52	1
Single extended	> 42	1	-	-	none
Double extended	80	1	15	64	none

The actual value of the floating point number is obtained by multiplying the sign, the mantissa, and the exponent part. A floating point number can be described as

$$F = s \times m \times b^e, \tag{1.1}$$

where s denotes the floating point number sign, e stands for the exponent value, b is the base of exponent part, and m stands for the mantissa part. Note that mantissa part includes the hidden bit (only if $b = 2$).

For instance,

3E2BDA28 (hex, IEEE 754 single precision)

= 0 01111100 01010111101101000101000 (binary)

$= (1.01010111101101000101000)_2 \times 2^{(01111100\ \ 01111111)_2}$

= 0.167824 (dec).

When the exponent is non zero, the hidden bit equals one, and the mantissa value is in the normalized format; otherwise, this bit equals zero, and the mantissa value is in the denormalized format. Following the IEEE 754 Standard, a single precision number is consisted of sign bit, exponent, and significand. The normalization format means the hide bit of significand is equal to 1, and the value of this number is represented by $(1.xxx....x \times 2^e)$. For the partial result of floating point operations, the value may

be formed as $(1.xxx...x \times 2^e)$ or $(0.xxx...x \times 2^e)$. In this case, we have to apply the post shift (1 bit right shift or several bits left shift) and appropriate exponent adjustment to adjust the result and generate the normalization format according to $(1.xxx...x \times 2^e)$.

In some special cases, the result is very small number and the value of exponent part is less than zero. To represent these small values in a certain range, we can use the de-normalization format. In this way, we will set the exponent part as zero, and right shift the significand part to form a de-normalization format as $(0.xxx...x \times 2^0)$.

To sum up, for both normalization and de-normalization, we should check the most significand bit of the significand part first, and then take appropriate shifting operation on the significand part, and adjust the exponent part.

The first bit is the sign bit to indicate the sign of floating point numbers. The exponent part is represented in a bias format. For the single precision representation, the exponent is biased by 127 ($2^{e-1}-1$, e is the number of the exponent bits), and is used to represent both tiny and huge values. The use of a biased exponent format makes comparison easy, because we need not use an extra sign bit for the exponent part. If the exponent uses the usual representation for signed values, like the two's complement format, the time complexity for comparison would be similar to the carry lookahead adder. Floating point numbers are equal if and only if their every corresponding bit is identical. Leaving out the exceptional values, comparisons on the bit patterns can directly determine the relative magnitudes of floating point numbers.

To tolerate error, the IEEE 754 standard also defines a set of exceptions. The exception formation can be summarized as in Table 1.2. During the floating point number calculation, some exceptional events may occur:

- Overflow, which arises because the calculated result is too large and exceeds

Table 1.2: IEEE 754 Exception Definitions

Type	Exponent	Mantissa
Zeroes	0	0
Denormalized numbers	0	non zero
Normalized numbers	1 to $2^e - 2$	any
Infinities	$2^e - 1$	0
NaNs	$2^e - 1$	non zero

the range of the IEEE representation. The signed infinity value will be output, and an appropriate exception signal will be produced.

- Underflow, which arises because the calculation result is too small and under the range of the IEEE representation. The denormalized value will be output, and an appropriate exception signal will be produced.

- Zerodivide, which arises whenever a divisor is zero. The signed infinity value will be output, and an appropriate exception signal will be produced.

- Operand error, which arises whenever any operand to an operation is a Not a Number (NaN), or the result is an imaginary, such a $sqrt(-1.0)$ or $log(-2.0)$. The signed NaN value will be output, and an appropriate exception signal will be produced.

With the floating point number representation, a floating point arithmetic operation has to complete more tasks than the corresponding fixed-point arithmetic operation, such as sign bit determination, exponent calculation and adjustment, and mantissa part calculation and adjustment. Each part requires separate combinational

logic units or fixed-point operation components. Therefore, the implementation of the floating point arithmetic operation is much more complicated than corresponding fixed-point arithmetic operation. To speed up the floating point computing process, we not only should design independent floating point execution units, but also should utilize different parallelism methods to optimize the computing system architecture.

1.2 Parallelism in Microprocessors

For complicated scientific applications, such as floating point number operations introduced above, the fundamental ways in which advances in technology improve performance are parallelism and locality [3]. This section will introduce the common methods employed in microprocessors to implement parallelism and locality improvement methods.

1.2.1 Instruction-level Parallelism

The basic stages in data processing are: loading from memory to register, executing in arithmetic units, writing back to register, and storing to memory. Each stage is generally performed in a single clock cycle. If these stages use different function units without hazard, or if the system can provide enough function resource to execute these tasks, these stages can be pipelined in a straightforward manner, so that the final result can be acquired almost every clock cycle after the initial latency. This overlap execution mode is a classical mode of instruction-level parallelism.

Moreover, the complicated floating point arithmetic operation can be easily divided to multiple sub-function stages, because each floating point operation needs to cope with individual sign, exponent, and mantissa parts. These sub-execution stages can also be pipelined and will speed up the whole execution process.

The main advantage of the pipelined structure is reducing the total latency of a long data queue from input to output. However, the pipelined structure can not reduce the latency for a single task. The pipelined structure improves the throughput of the entire workload, and make each function unit work more efficiently.

1.2.2 Thread-level Parallelism

The pipelined structure makes the different function units to operate simultaneously. However, a structural hazard or data dependency will cause the stall of pipeline. Meanwhile, the increased throughput aggravates the speed gap between processor and memory. The function units in the processor still have to wait to be fed data. Increasing the number of execution units per processor can directly map multiple tasks to the multiple execution units. This trend has resulted in modern microprocessor design to integrate multiple cores on a chip, which can exploit thread-level parallelism by having sufficient execution resource [4].

Another method is the out-of-order execution, which provides an efficient control scheme for multiple threads within a process. In this way, instructions are scheduled dynamically and allowed to complete out of order to keep function units busy. In general, out-of-order execution is an extension of instruction level parallelism, because the crucial property is the improvement of function unit efficiency. On the other hand, out-of-order execution requires sophisticated branch prediction techniques and sophisticated caches. These associate units occupy considerable area in a modern processor and consume extra energy.

1.2.3 Vector Data Parallelism

Instruction-level parallelism and thread-level parallelism require a more complex control system to avoid structural hazard and data dependency. A more straightforward parallel mode is data parallelism, which is where the same operation is performed simultaneously on a set of data elements [5]. A vector processor includes multiple homogeneous function units to operate on multiple data. Comparing with a scalar processor, the number of instructions will be reduced and the instruction decoding time will be also reduced. Therefore, vector processing can significantly improve the execution efficiency.

However, vector processing requires a certain amount of time to load data sets, before it fills the pipe. To reduce the loading time, an appropriate vector register file is designed for fast access between sequence operations. A batch of vector operations will be pipelined, and this technique is called vector chaining. In this way, the data sets can be held in register files, and the overall performance will be dramatically improved. Independent parallel datapaths can be applied to most scientific computing models, and vector processing have very good cost/performance on data parallel codes.

Figure 1.2 shows the difference between the three types of parallelism. Within each diagram, each box represents one instruction, and each shape within a box represents one operation. Boxes are grouped by instruction stream. For Instruction-level parallelism, the possible interactions between concurrent instruction grows quadratically with the number of parallel instructions. Thread-level parallelism incurs the expense of duplicating instruction management logic for each instruction stream, and also suffers overheads for inter-thread synchronization and communication [5]. Therefore, we utilize a pipelined structure to speed up the floating point arithmetic unit, and use the vector data parallelism to improve the efficiency of the entire computing

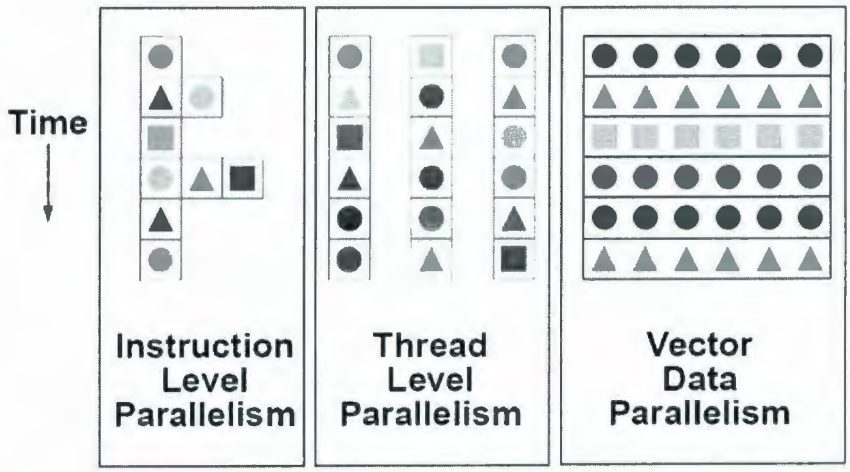

Figure 1.2: Different forms of machine parallelism. [5]

process.

1.3 Hardware Platforms

In the previous section, a brief overview of data parallelism has been provided. Another important issue in the development of the vector operation unit is the proper choice of hardware platforms. Currently, most popular hardware platforms include FPGAs and ASICs. This section will provide a brief discussion of the features and applications of different hardware platforms.

1.3.1 Application Specific Integrated Circuits (ASICs)

Application Specific Integrated Circuits (ASICs) are employed in the fully customized hardware approach which heavily focuses on particular applications. A functional description of digital ASICs will be completed first using a hardware description language (HDL), such as Verilog or VHDL, and this process is usually called the

Register Transfer Level (RTL) design. The logic synthesis tools can compile the RTL design and generate a gate-level netlist, which is next used to place the cells using a placement tool. During this design process, several optimization methods will be used to improve the performance, and will be subjected to specified constraints. The flexible routing and placing technology reduces the interconnect cost between different function units.

ASICs are generally used to achieve low power consumption and high speed, because of reduced area and increased speed. However, non-recurring engineering costs and the complexity of design tools not only increase manufacturing and design time and cost, but also require the designer to possess higher design and optimization skills.

1.3.2 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are employed in semi-customized hardware approach which consists of programmable logic components and programmable interconnects. In most modern FPGAs, the logic components are implemented by RAM-based look-up table (LUT) and can be programmed to perform various logic functions. These RAM-based logic components need to load the function configuration during boot-up process, that usually takes mantissas of a second. A hierarchy of programmable interconnects connect logic components to implement more complex logic functions. As the gate density on integrated circuits is increased rapidly, modern FPGAs not only provide embedded memory module to build fast register or on-chip memory, but also implement embedded processor blocks within the FPGA's logic fabric, which only occupy a small die area and consume low power.

The RTL design and synthesis process for FPGA is similar to the design for ASIC.

Based on technology-mapped netlists, the FPGA company's place-and-route software will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. After the design and validation process, the binary configuration file is generated and can be downloaded to the FPGA chip via a Joint Test Action Group (JTAG) debug port.

The FPGA implementation usually runs slower than an ASIC realization, because the programmable interconnect module cannot always provide an efficient interconnect solution as in the ASIC platform. However, the FPGA design could be an iterative process, and can be easily re-programmed after debugging. These advantages not only reduce development time, but also provide a great flexibility for various applications. The flexibility has evolved to exploit particular forms of parallelism common to certain classes of embedded applications.

For high performance computing applications, FPGA architectures can offer inherent parallelism of the logic resources. A fast register file can be built using the embedded memory blocks, and a lot of single precision floating point units can be implemented on the FPGA. Considering these advantages of FPGA, the FPGA is well suited for implementing the vector processing paradigm.

1.3.3 Embedded Systems

With embedded processors, multipliers, and block memory, an FPGA chip may contain over 400 million transistors. Harnessing all this raw computing power requires that the developer's attention moves beyond logic function design into parallel computer system architecture. Moreover, these hardware components and appropriate software modules compose an embedded system, which can be optimized for one or a few dedicated applications.

The embedded system design process includes three steps:

- Architecture configuration, which contains target chip selection, base system building, local bus design, memory on chip configuration, and I/O interface design. This step will determine the system architecture and define the interface between different components.

- Hardware component design, which contains control unit design and function unit design. These hardware components can be implemented and tested on the FPGA platform independently, and then imported to the embedded system. For each component, a wrapper module will be used to customize the interface and connect to the local bus.

- Software application design, which contains embedded operating system configuration, board support package (BSP) development, and application design. The software application can be loaded in two ways: initializing Block RAMs and running under an embedded operating system. The first approach is faster, because the binary code of the software application will be loaded to a memory space with the hardware configuration. The second method needs operating system support and will be more flexible for debugging.

1.4 Motivation and Organization of the Thesis

Many science and engineering applications require high computational accuracy and flexibility. For instance, the Earth Simulator contributes to prediction of environmental changes by analyzing the vast volume of the observed data. A fast floating point computing system can provide strong support for these applications. Moreover, the vector-thread (VT) architectural paradigm [1] has become one of the new ways

to implement all-purpose embedded computing. VT architectures unify the vector and multithreaded execution models. A large amounts of structured parallelism can be implemented on VT architectures. The simple control and datapath structures of vector processing enable the embedded computing system to attain high performance at low power.

The main purpose of this thesis is to study a vector computing scheme for floating point operations. The study of vector data parallelism for floating point number computing model is very challenging. The challenging objectives are: 1) high throughput of the floating point arithmetic operations, 2) efficiency of local storage access, and 3) chaining between all function units and registers.

In particular, the floating point arithmetic operations are time consuming for any processor. To minimize the critical path delay, quick carry chain and embedded multiplier are utilized in the FPGA implementation.

In order to easily access data on chip, a vector register file is desirable. In this thesis, we also proposed a generic multi-ported register file structure, which can efficiently load data from the local bus and provides different data elements to multiple operation units.

To further improve the performance of the proposed vector floating point processing unit, we propose several design improvements for FPGA implementation. The synthesis results show that the performance is satisfactory in many numerically-intensive applications, such as the Earth Surface Simulation Model.

The rest of this thesis is organized as follows: Chapter 2 presents the basic floating point arithmetic units, including Adder, Multiplier, and Multiply-Add-Fused modules. In this chapter, we not only present the structure and algorithm for these floating point arithmetic units, but also discuss several optimization methods and compare these different architectures for these fast designs. A detailed discussion of the vector

data parallelism scheme will be given in Chapter 3. The vector architecture will be introduced first, and the heart module, vector register file, will be discussed in detail. For the common floating point operations with two operands, we also present the two loader scheme to improve the access performance. Implementation and optimization on FPGA devices will be introduced in Chapter 4, and two FPGA platforms of the leader manufactures, Altera and Xilinx, will be compared. Performance analysis and further discussion will be given in Chapter 5. For each floating point arithmetic units, the timing analysis will be presented. The extensibility of whole vector floating point unit will be discussed, and the bandwidth requirement for our design will be also presented. Chapter 6 concludes the thesis and proposes future research directions.

Chapter 2

Floating Point Arithmetic

In this chapter, we will introduce three important floating point number arithmetic units: the adder, the multiplier, and the adder-multiplier-fused unit. In the last section the extension to a complex floating point operation will also be discussed.

2.1 Floating Point Adder

Floating point addition is a fundamental operation in many scientific and engineering applications. The addition process for two floating point numbers is shown in the following expression:

$$\begin{aligned} F_1 \pm F_2 &= (s_1 \times m_1 \times b^{e_1}) \pm (s_2 \times m_2 \times b^{e_2}) \\ &= (s_1 \times m_1 \pm s_2 \times \frac{m_2}{b^{e_1 - e_2}}) \times b^{e_1}, \end{aligned} \tag{2.1}$$

A floating point adder (FADD) consists of a fixed-point subtracter for exponents, a fixed-point adder for aligned significands, a barrel shifter for potential pre-shifting (alignment) and post-shifting (normalization), a rounding module, and support circuitry for sign detection and exponent adjustment. Compared with fixed point addition, floating point addition is more complex because differences in the exponent

part require alignment shifting before the mantissa addition, and the subtraction of close numbers leads to potential left shifting for the result normalization. In detail, floating point addition is composed of the following six steps:

Step 1. Exponent subtraction

$$e_{diff} = |e_1 - e_2| \tag{2.2}$$

The difference between the exponents will be used to align the mantissa part of the smaller operand. A fixed-point subtracter is used to generate the subtraction result, and the result should be positive value.

$$e_s = max(e_1, e_2) \tag{2.3}$$

When we complete the subtraction, we can easily determine which exponent value is the larger one as shown in Eq. 2.3, and this exponent value will be set as the partial exponent result.

Step 2. Mantissa alignment

The barrel shifter[6] is used to quickly align the mantissa part. The barrel shifter module consists of a multiplexer array and can provide the fast shift operation for the preshift and postshift modules. The compact barrel shifter, with the encoded control, reduces the critical path because it contains few multiplexers. For instance, an 8-bit left barrel shifter is shown in the figure 2.1, which includes 24 2-to-1 multiplexers. The delay for the 8-bit left shifting is three level delay of the 2-to-1 multiplexer.

Step 3. Fixed-point number addition and leading zero anticipation.

The carry lookahead adder (CLA) is used for fixed point number addition, and the Leading Zero Anticipator (LZA)[7] is used for detecting the postshift bits in the parallel mode. To reduce the critical path delay, the key module of the floating point adder is the leading zero anticipator, which can predict the left-shifting bits in parallel

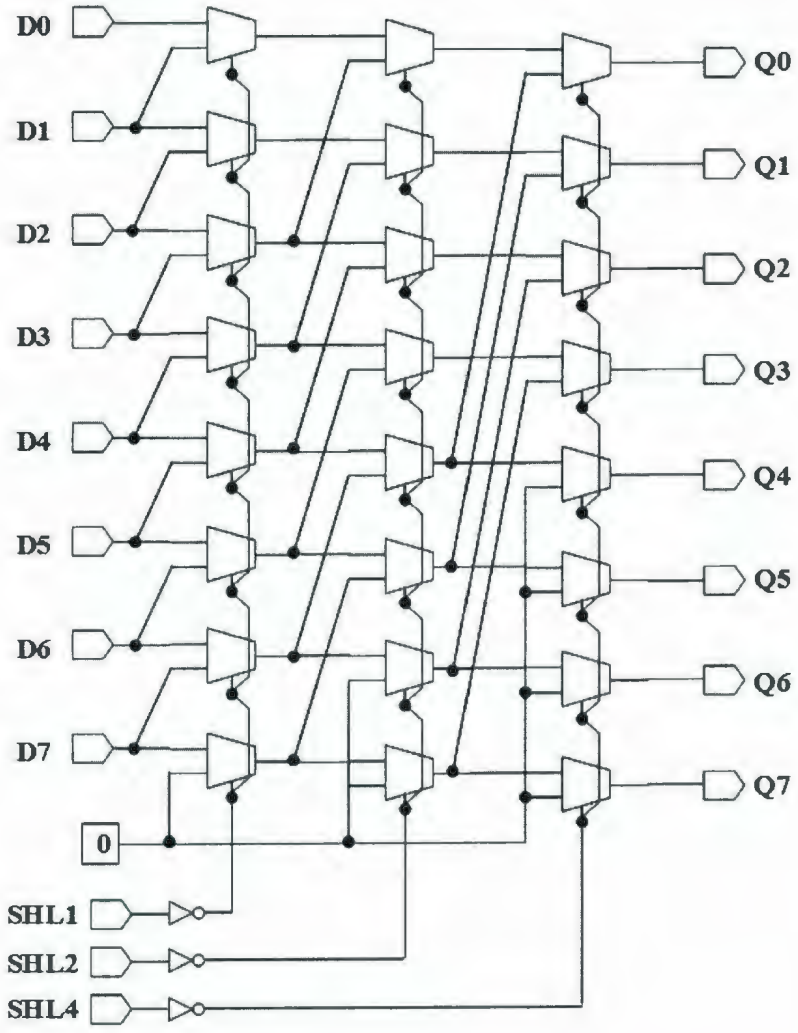

Figure 2.1: Multiplexer-based 8-bit Barrel Shift Circuit

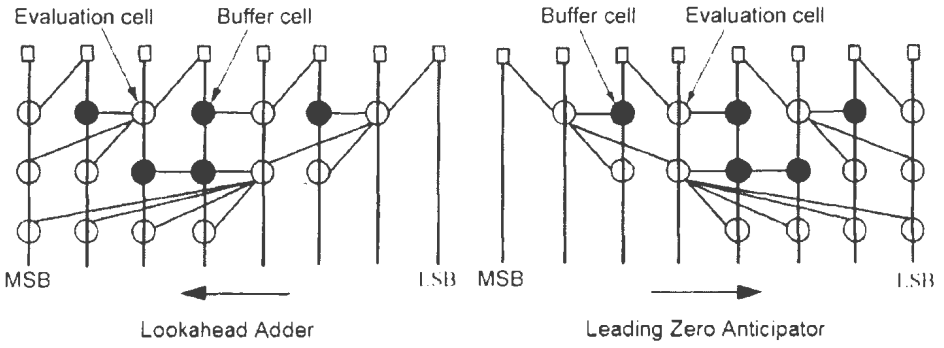

Figure 2.2: Implementation sequence of CLA and LZA [7]

mode. The LZA operates in the opposite direction from the carry lookahead adder, and its scheme is similar to the parallel prefix computation [8]. Figure 2.2 illustrates the implementation sequence of CLA and LZA.

The LZA can directly determine the number of leading zeros or ones from the addition results of two input operands. This process is building a 1-string followed by a 0-string, and an encoder or priority encoder yields the index of the leading 1. The P, G, and Z signals describe the bit to bit relation of two input operands, A and B. The definition is following:

$$P = A \oplus B,$$

$$G = A \bullet B,$$

$$Z = \overline{A + B},$$

The following four cases provide the leading one/zero result of mantissa addition.

Case1: $A > 0, B > 0, A + B > 0$

A 0...00...010001...

B 0...00...000110...

$Z...ZZ...ZPZP...$ Carry $= 0$

A 0...00...0100100...

B 0...00...0111010...

$Z...ZZ...ZGPP...$ Carry = 1

Case2: $A < 0, B < 0, A + B < 0$

A 1...11...100000010...

B 1...11...110000101...

$G...GG...GPZZ...$ Carry = 0

A 1...11...101100...

B 1...11...111100...

$G...GG...GPGG...$ Carry = 1

Case3: $A > 0, B < 0, A + B > 0$

A 0...00...01000001010...

B 1...11...01000100101...

$P...PP...PGZZ...ZP...$ Carry = 0

Case4: $A > 0, B < 0, A + B < 0$

A 0...00...0111111000010...

B 1...11...0111011000100...

$P...PP...PZGG...GP...$ Carry = 1

Figure 2.3 shows the finite-state diagram. The state transition represents all possible bit to bit relations described above.

Following this finite state diagram, a component is constructed to generate the characteristic string, and an encoder is used to count the number of leading ones or zeros.

Step 4. Normalize significand and adjust exponent.

The normalization for the addition result has two possibilities: one is a 1 bit right shift for the carry out mode; the other is a multiple bit left shift for close number subtraction. For each shift operation, the appropriate adjustment will also be done

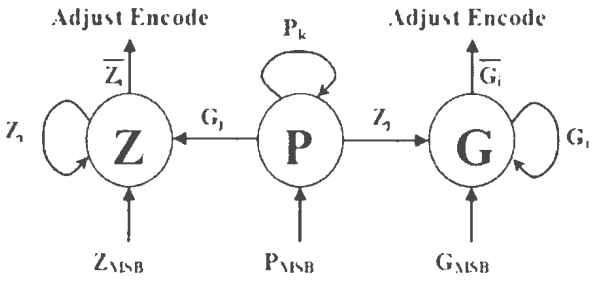

Figure 2.3: Finite state diagram of LZA (modified from a picture in [7])

Table 2.1: The Scheme of Rounding to The Nearest Even Number

Origin	...	Z_{l+1}	Z_l	G	R	S
1bit right shift	...	Z_{-l+2}	Z_{-l+1}	Z_l	G	$R \vee S$
1bit left shift	...	Z_{-l}	G	R	S	0
After Normalization	...	Z_{-l+1}	Z_l	Z_{-l-1}	Z_{-l-2}	Z_{l-3}

on the partial exponent result.

Step 5. Rounding, normalize, and adjust exponent.

The rounding module converts intermediate addition results to lower-precision formats for storage or output. After the first normalization and potential left shifting, the rounding scheme is used to adjust the last bit of the mantissa and can cause adjustment of exponent. Three additional digits are kept and can potentially affect the rounding result. We have the following format output of the mantissa adder:

G: Guard bit

R: Round bit

S: Sticky bit

Table 2.1 shows the nearest even rounding scheme [9]. The extra 3 bits at the right are adequate for determining properly rounded results.

The algorithm for nearest even rounding is as follows [9]:

if $Z_{-l-1} = 0$ *or* $Z_{-l} = Z_{-l-2} = Z_{-l-3} = 0$ *then*

do nothing;

else

add 1 to mantissa;

endif;

If the potential add one operation causes a carry out, the 1 bit right shift and appropriate exponent adjustment will be done.

Step 6. Sign detection and set flag for exception.

The last step is sign detection and exception detection. In the worst case, we need to compare both the exponent part and the mantissa part to determine the sign bit.

For error tolerance, an extra logic module will be used to generate the appropriate exception signals: overflow, underflow, or NaN occasions. The generic scheme of the floating point adder is depicted in Figure 2.4.

If it is not necessary to economize on hardware, the dual path design can be used to reduce one shift step from the generic design, because the preshift and postshift always occur in different occasions. Figure 2.5 shows the difference between the single path design and dual path design. The shaded module indicates the slow part in the critical path. The single path design includes three slow parts in the critical path, but the dual path includes only two slow parts.

2.2 Floating Point Multiplier

In contrast to the floating point adder, the algorithm of a floating point multiplier (FMUL) is quite simple. The multiplication process is described by the following

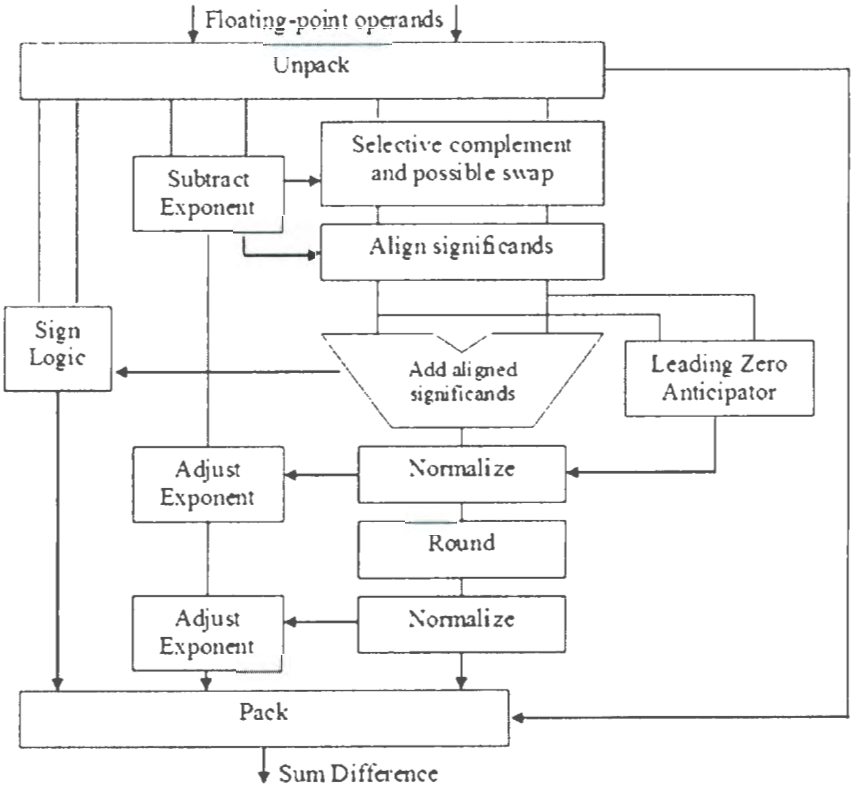

Figure 2.4: Block diagram of a floating point adder/subtracter (modified from [9])

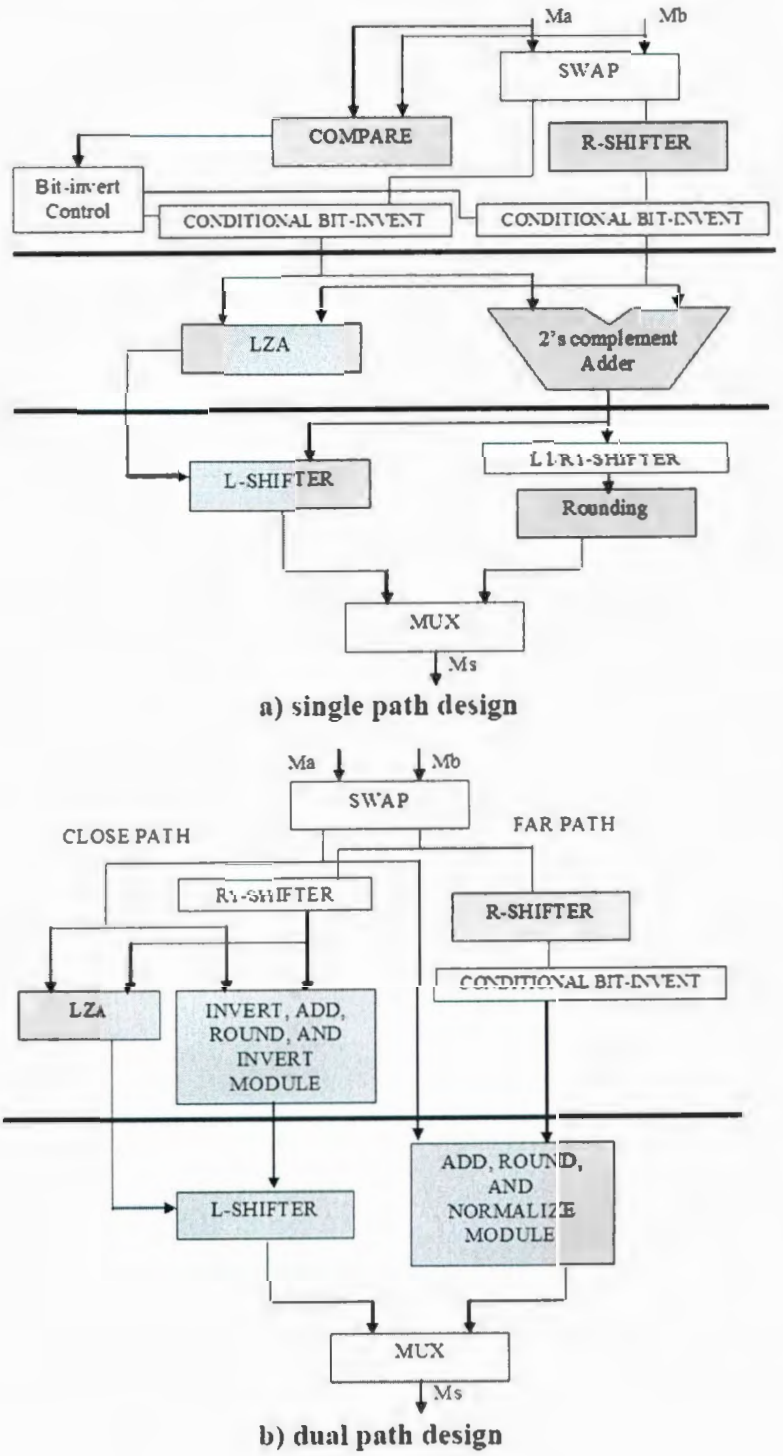

Figure 2.5: Block diagram of the single path and the dual path designs (modified from a picture in [9])

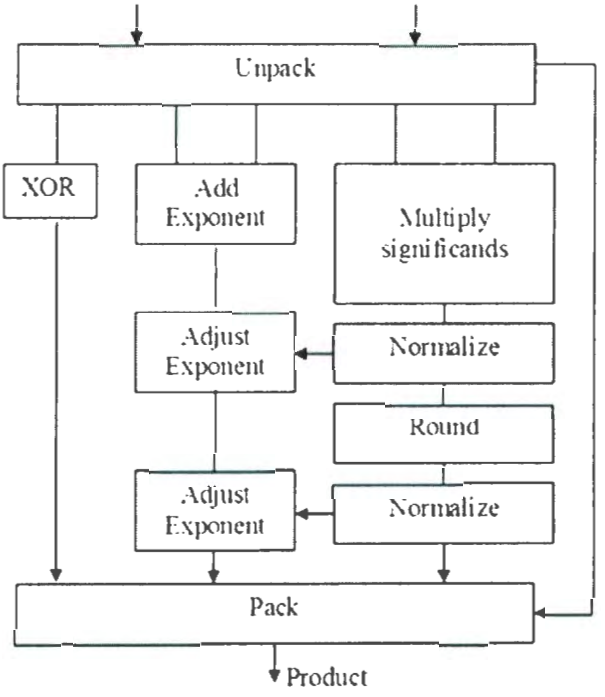

Figure 2.6: Block diagram of a floating point multiplier (modified from a picture in [9])

expression:

$$\begin{aligned} F_1 \times F_2 &= (s_1 \times m_1 \times b^{e_1}) \times (s_2 \times m_2 \times b^{e_2}) \\ &= (s_1 \times s_2) \times (m_1 \times m_2) \times b^{e_1+e_2}, \end{aligned} \tag{2.4}$$

A floating point multiplier includes five modules: exponent adder, fixed point multiplier, rounding, normalization, and a support module to decide the sign bit and adjust the exponent. The arithmetic block diagram is illustrated in Figure 2.6.

The fixed point multiplier occupies a lot of die area and causes a long delay in the critical path, because fixed point multiplication is equivalent to multi-operand addition. The carry save adder (CSA) consists of several 1-bit full adders, each of which computes a single sum and carry bit based solely on the corresponding bits.

In this way, the three input numbers (two operands and carry in number) can be reduced to two numbers, and the carry adder tree can reduce the multi-operands to two operands. The Wallace tree [10] is one particular structure of the CSA trees. This structure reduces the number of operands in the first stages. A carry lookahead adder will be used in the last stage to add the last two operands and generate the final multiplication result. For the single precision floating point number, the fixed-point multiplier part is a 24-bit multiplier. Using the Wallace Tree structure, the compression process includes 7 steps to reduce 24 operands to 2 operands, shown in the following process:

$$24 \rightarrow 16 \rightarrow 12 \rightarrow 8 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2$$

In a fixed point multiplier, the digits of operands determine the number of operation cycles. A higher radix number representation leads to fewer digits, and the reduction in number of cycles can simplify the CSA Wallace tree. The radix-4 Booth code[11] is an efficiently method to reduce the height of the Wallace tree and shorten the critical path delay.

When multiplication is done in radix-4, the two bits $(x_{i+1}x_i)_2$ are used to decide the multiples: $0\times$, $1\times$, $2\times$, or $3\times$. However, computing $3\times$ operand needs an extra addition operation. A possible solution is adding $-1\times$ and send a carry of 1 into next radix-4 digit of the multiplier. In this way, the radix-4 Booth encoding method is shown in Table 2.2.

Using the radix-4 Booth encoding, the initial number of addition operations is reduced from 24 to 12, but some addition operations require the carry-in bit. Therefore, an extra partial product is used to collect these carry-in bits. The total initial number of addition operations is 13 and the compression process are:

$$13 \rightarrow 9 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2$$

There are only 5 steps in the array, and Figure 2.7 shows the 5 stage Wallace Tree.

Table 2.2: Radix-4 Booth Encoding

B_{i+1}	B_i	B_{i-1}	*Encode*	C_{in}
0	0	0	0	0
0	0	1	A	0
0	1	0	A	0
0	1	1	2A	0
1	0	0	-2A	1
0	1	1	2A	0
1	0	1	-A	1
1	1	0	-A	1
1	1	1	0	0

For the n-bit traditional multiplier, the time complexity is $nlog(n)$, where $log(n)$ is the time complexity of the n-bit lookahead adder. Utilized the radix-4 Booth coded Wallace tree structure, the time complexity is $log(2n) + c$, where $log(2n)$ is the time complexity of the final $2n$-bit lookahead adder, and c is the extra delay caused by CSA tree. Therefore, the time complexity is reduced from $nlog(n)$ to $log(n)$.

The Booth's coded Wallace CSA tree can efficiently shorten the critical path. However, the Wallace tree structure not only occupies large area on the target chip, but also introduces large interconnection delay. Modern FPGA chips provide embedded fixed-point multipliers to perform fixed word width multiplication. These embedded multipliers can also provide fast interconnection paths with other function units. Karatsuba Multiplication formulations [12] are used to extend these fixed word size multiplier to variable word width multipliers. For instance, the fractional part of the single precision floating point number is 24 bits, and the Xilinx embedded multi-

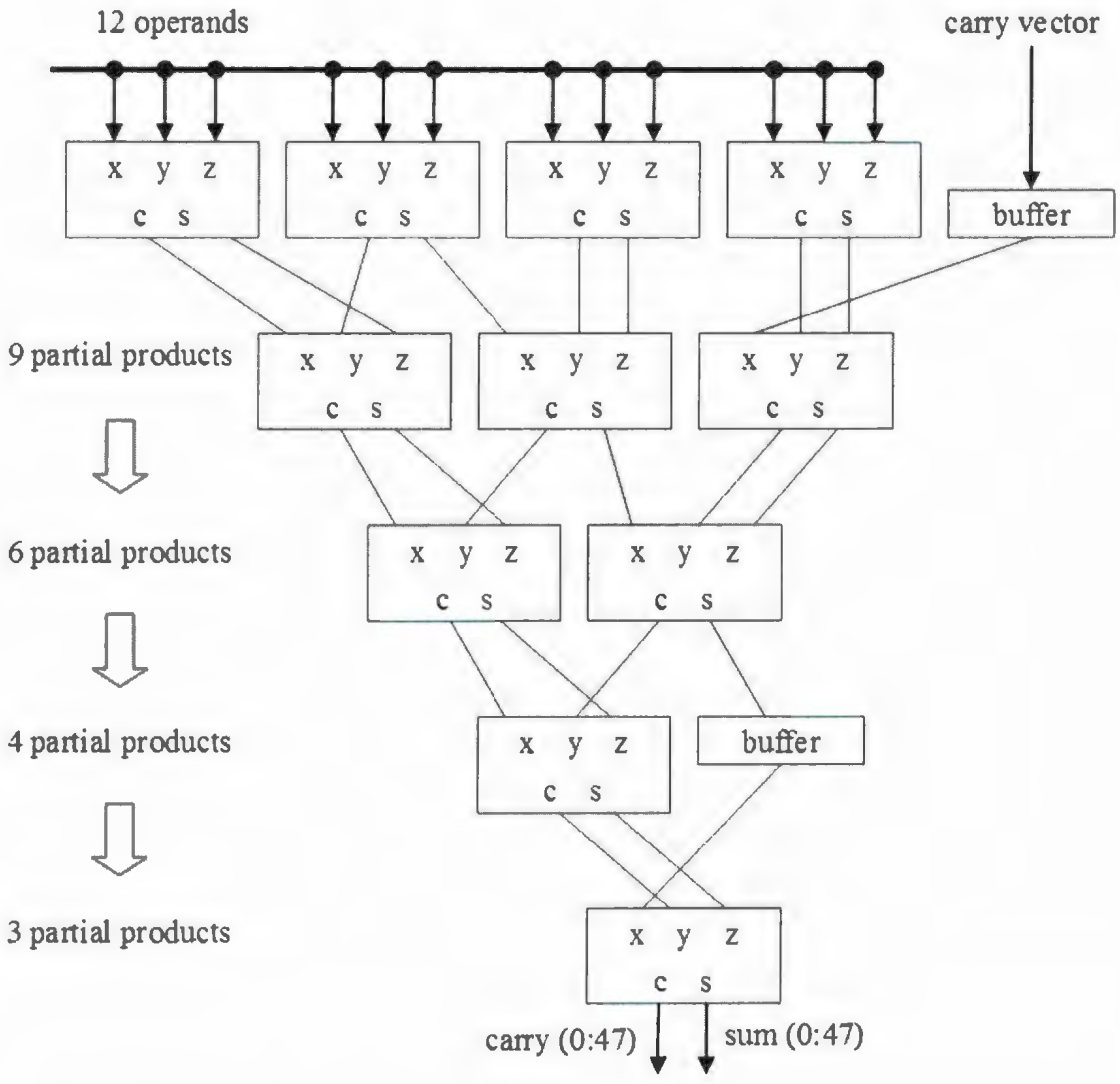

Figure 2.7: A radix-4 Booth coded Wallace tree (modified from a picture in [11])

plier is a 18-bit fixed-point multiplier. Therefore, the fractional part will be divided into a high part and a low part to use the embedded multiplier. This multiplication includes four multiplications, three additions, and a few shifters. If the fractional part is directly divided into a high part (6 bits) and a low part (18 bits). The following mathematical manipulation shows the process of the Karatsuba Multiplication:

$$A = A_{high} \times 2^{18} + A_{low}, \tag{2.5}$$

$$B = B_{high} \times 2^{18} + B_{low}, \tag{2.6}$$

$$\begin{aligned} A \times B &= A_{high} \times B_{high} \times 2^{36} + A_{high} \times B_{low} \times 2^{18} \\ &+ A_{low} \times B_{high} \times 2^{18} + A_{low} \times B_{low}. \end{aligned} \tag{2.7}$$

This multiplication includes four multiplications, three additions, and a few shifters. The multiplication result is 48 bits, but the useful part is the high order 27 bits: a 24 bits mantissa part and 3 extra bits for rounding. To eliminate the useless low part result, we can divide the mantissa part into the high part (14 bits) and the low part (10 bits). In this way, the partial multiplication result of two low parts will be shifted out from the final result. We can only check if the partial result is zero and keep one bit as the lost bit to determine the sticky bit for rounding. The multiplication process is shown in the following formulations:

$$A = A_{high} \times 2^{10} + A_{low}, \tag{2.8}$$

$$B = B_{high} \times 2^{10} + B_{low}, \tag{2.9}$$

$$\begin{aligned} A \times B &= A_{high} \times B_{high} \times 2^{20} + A_{high} \times B_{low} \times 2^{10} \\ &+ A_{low} \times B_{high} \times 2^{10}. \end{aligned} \tag{2.10}$$

In this way, this multiplication includes three multiplications, two additions, and a few shifters, and the critical delay will be shortened. Without leading zero check and potential left shift, normalization of the floating point multiplier is simpler than the floating point adder. Also, the exception detection module will be more compact than that from the adder.

In order to achieve high throughput for the floating point arithmetic operations, we use a deep pipeline structure for the floating point arithmetic units. The FADD is broken into 10 stages, and the FMUL is broken into 8 stages. Although the deep pipeline structure will cause longer latency, the throughput of floating point arithmetic units is significantly increased.

2.3 Floating Point Multiply-Add-Fused (MAF)

Many scientific and engineering applications will execute a coherent calculation process including differential operations. The floating point multiply-add-fused (MAF) [13] operation will complete multiplication and addition in one execution unit. In this design, addition is merged into the adder array which is used for multiplication, and the rounding step for the partial multiplication result is eliminated. In this way, the critical path delay will be shortened with respect to separate multiplication and addition. Moreover, a single instruction, the MAF, i.e., $A \times B + C$, is capable of handling both addition and multiplication operations, e.g., by defining $B = 1$ for addition and $C = 0$ for multiplication.

The main components of MAF include: CSA Wallace Tree for multiplication, pre-alignment module for operand C, CSA for merging, Fixed-point number Adder and Leading Zero Anticipator (LZA), normalization module, and rounding module. Figure 2.8 illustrates the structure diagram of a single precision floating point MAF.

Table 2.3: Single Direction Shifting for Alignment

Case 1	$3m+2$ bits $xx\cdots x.0000\cdots 00\cdots 0$	Operand C: 2 bits for Rounding, no shift
	$xx.xx\cdots xxx\cdots x$ $2m$ bits	Partial Result of $A \times B$
Case 2	$3m+2$ bits $xx\cdots xxx.xx\cdots x00\cdots 00\cdots 0$	Operand C: $shiftbits = (e_a + e_b - bias) - e_c + (m+3)$
	$xx.xx\cdots xxx\cdots x$ $2m$ bits	Partial Result of $A \times B$
Case 3	$3m+2$ bits $xx.xx\cdots x00\cdots 00\cdots 0$	Operand C: Maximum shift, $shiftbits = m+2$
	$xx.xx\cdots xxx\cdots x$ $2m$ bits	Partial Result of $A \times B$

The exponent logic module calculates the preshift bits, $e_a+e_b-e_c-100$, and generates the partial exponent result, $max(e_a + e_b - 100, e_c)$.

Most of the components of MAF are extracted from the adder or multiplier. One special module is the pre-alignment module. Comparing with the multiplication result $(A \times B)$, the third operand C may be shift left or right. To simplify the shift process and guarantee the accuracy, the third operand C will be extended to $3m+2$ bits by adding a zero string to its end. In this way, the potential bi-directional preshift becomes a uni-directional (right) shift.

Table 2.3 describes the details of the shift algorithm. Case 1 indicates the occasion that the operand C is much large than the multiplication result, and the operand C

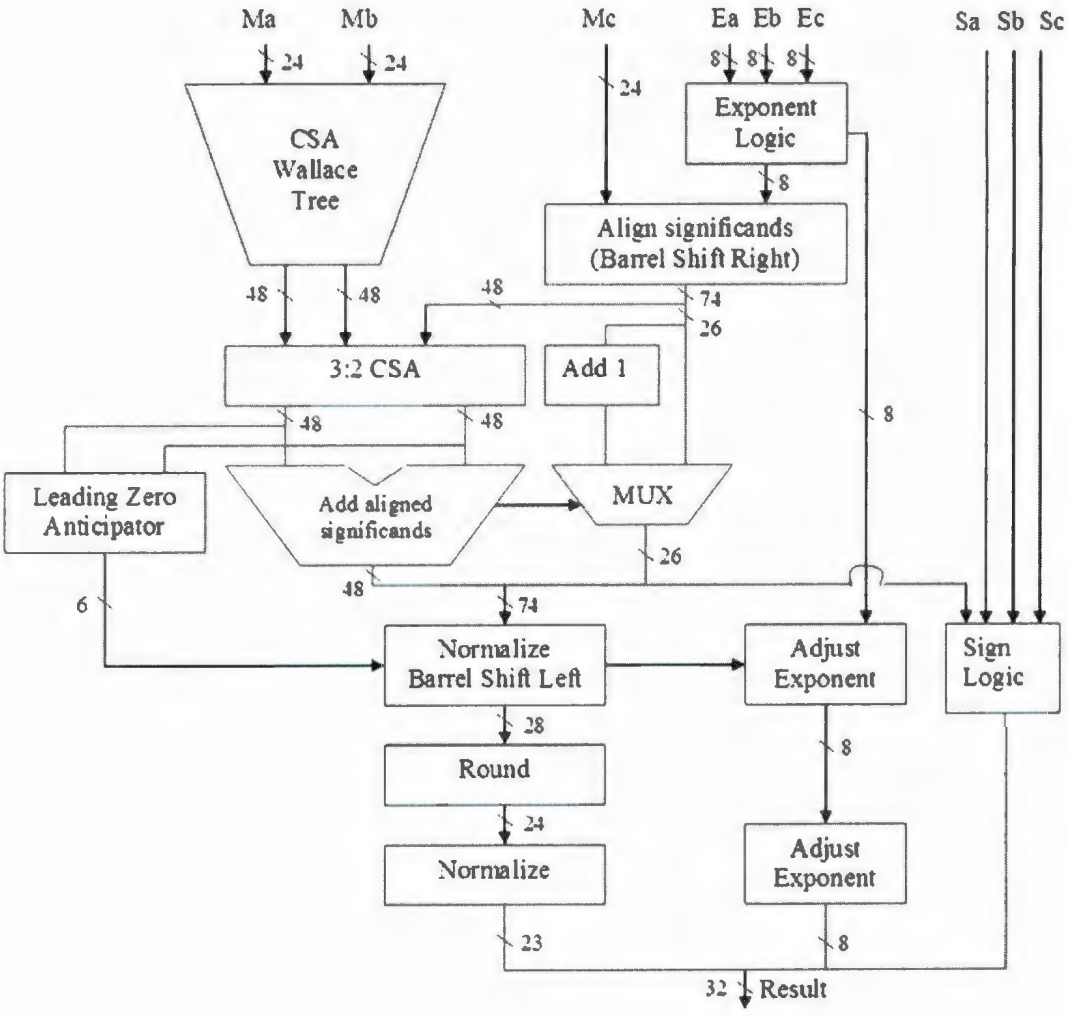

Figure 2.8: Block diagram of a floating point multiply-add-fused (modified from a picture in [11])

need not shift. Case 2 is a common case for preshift, and the shift bits are decided by the difference between the multiplication result and the operand C. Case 3 is an extreme case of case 2, and the maximum shift bits is $m + 2$. The shift bits are determined by the exponent part of operands, and the preshift operation can be performed in parallel mode with the CSA Wallace tree.

2.4 Other Extensions of Floating Point Operation

Floating point arithmetic not only involves addition and multiplication, but also includes division, logarithmic arithmetic, square-root, and many trigonometric functions. These arithmetic functions are generally regarded as slow and complex parts in most implementations. Fortunately, these functions are rarely used in most applications, and some possible solutions exist when these complex functions need to be implemented. The first option is to evaluate the series expansions of these complex functions by means of addition and multiplication. Another method is using table lookup and interpolation as an aid in arithmetic computations [9].

2.5 Summary

The floating point representation is presented in this chapter first, and the IEEE 754 standard is used for our designs of floating point arithmetic units. Floating point addition and multiplication are the primary floating point arithmetic operations. For the floating point adder, we main discuss the rounding scheme and leading zero anticipation algorithm. The Radix-4 Booth encoding Wallace tree and Karatsuba multiplication method are introduced to improve the performance of the floating point multiplier. These fast floating point arithmetic units can be used as efficient

execution units, and a high performance floating point processing unit can be built utilizing the vector architecture.

Chapter 3

Vector Floating Point Processing Unit

Vector processing is an efficient parallel computing mode for data-intensive applications. A traditional vector supercomputer includes complex logic chips, huge amounts of SRAM memory chips, and multiple CPUs. Such supercomputer not only features super performance in scientific and engineering applications, but also has super size and power consumption. Utilizing mature CMOS technology, Krste Asanovic built the first complete single-chip vector microprocessor in 1998 [5]. In todays high performance processors, vector processing units cooperate with scalar processors to provide large speedup on data parallel codes. Meanwhile, this compact implementation of a vector processing module is well suited to some special applications, such as earth simulation, which deals with a high volume of floating point operations. This chapter presents a detailed description of the architecture of the Vector Floating Point Processing Unit (VFPU), which use the IEEE 754 standard (single precision) for floating point arithmetic operations.

The basic component of the VFPU consists of a vector register file, vector memory

units, vector arithmetic units, and a control logic unit. Section 3.1 is a detailed description of the architecture. Section 3.2 gives the configuration of a vector register file. Section 3.3 describes the vector memory unit, which will comprise the load/align function and write back function. Section 3.4 discusses the floating point arithmetic units that are used as execution units. Section 3.5 discusses the most important vector chaining scheme [14]. Vector processing requires high data throughput and experiences long startup penalties. The chaining scheme allows a coherent execution on vector data in the vector register file. These long running vector instructions reduce the ratio of memory access time to the total execution time.

3.1 Architecture

As the standard vector machine needs to consider compatibility with scalar instructions, the control logic is very complicated and constrains the performance. VFPU mainly focuses on vectorizable computation, and the main challenges are improving efficiency for memory access and keeping smooth execution process.

Figure 3.1 shows an overall block diagram of the VFPU, which contains vector register file (VREG), vector memory unit (VMU), vector floating point adder (VFADD), vector floating point multiplier (VFMUL), control logic unit, and address generator. Three vector function units, VMU, VFADD, and VFMUL, are structured as eight parallel lanes, and communicate via the central VREG. The number of lanes is determined by the target chip size and local bus throughput.

Considering the logic usage of FPGA chip and bus bandwidth, we select 8-lane configuration as the major configuration in the prototype, and the extensibility of VFPU will be discussed in Chapter 5. For the 8-lane configuration, eight pipelined floating point multipliers and eight pipelined floating point adders are connected

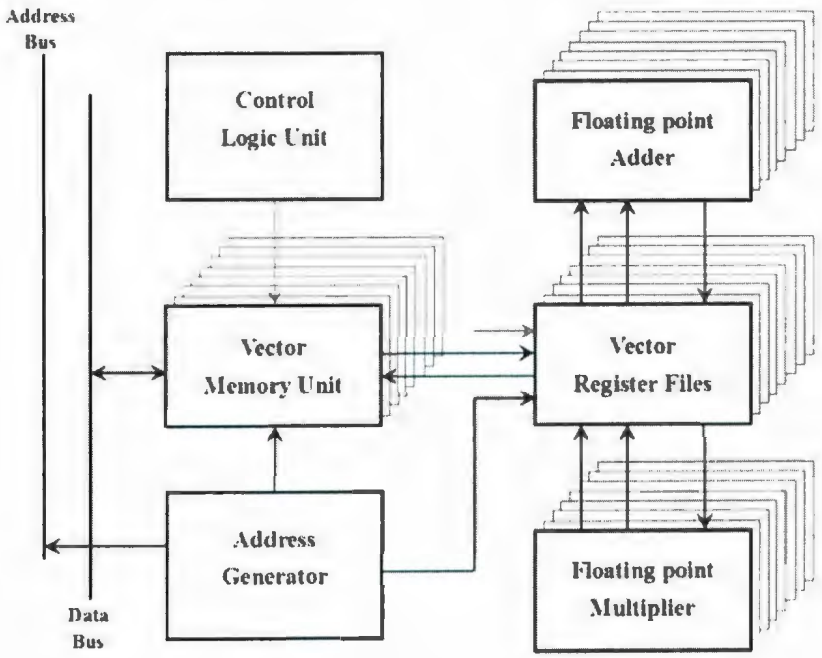

Figure 3.1: Block diagram of a vector floating point processing unit (modified from a picture in [5])

to the VREG. These execution units acquire data from the VREG and write the results back. Meanwhile, eight vector memory units work between the register file and internal data bus, and the control logic unit controls the access sequence to avoid hazards.

The VREG is the central module of the VFPU. Every vector function unit has an independent data access path to the VREG. Since data is acquired from the internal data bus, most of the communication between vector instructions occurs locally which greatly improves the data exchange efficiency.

Vector execution generates an increase used for bandwidth to access the external memory. To improve the external memory throughput, the traditional vector machine employs interleaving and provides adequate buffer to access the memory banks [15]. In VFPU, the external memory module is implemented by Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM), which can transfer data on the rising and falling edges of the clock signal. Considering the high data transfer rate of DDR SDRAM, VFPU applies a single memory access port and an align module to generate the 256-bit internal data bus. In this way, the memory control logic is markedly simplified.

The structure of the control logic unit is quite simple too, and can be customized for any specific applications. A flag register is used to indicate the calculation sequence. Following these status codings in flag register, the appropriate control signals are generated and sent to the vector function units. Moreover, the modern FPGA chip provides an embedded processor, which can be used as a flexible control unit and complete more dedicated control tasks.

3.2 Vector Register File

As locality is a key to high-performance vector processor, the VREG is the heart module of VFPU. Vector function units can access local data on VREG via independent access ports. Meanwhile, continual vector instructions can quickly exchange data through the VREG. The size and configuration of the VREG largely affects the performance of the vector processing unit. Therefore, VFPU appears a natural match to intelligent RAM (IRAM) [16] technology, that can directly convert high on-chip memory bandwidth into entire system speedup.

Increasing the number of vector register increases the spatial locality, which will reduce the requirement to access external memory. However, a long VREG will not only increase the startup overhead, but also will add complexity to specify the vector register. The VREG of VFPU is divided into eight parallel lanes. One lane contains 16 vector registers, each including 16 32-bit elements. Modern FPGA devices provide large numbers of on-chip memory, which can be easily configured in different arrangements. In Chapter 5, we will compare the performance of VFPU with the different VREG configurations: the number of lanes will be increased from 8 to 12, and the number of vector register per lane will be increased from 16 to 32.

The VREG enables multi-port data access. To avoid conflict in the VREG, one read and one write port are provided per vector memory unit, and two read and one write ports are provided for each floating point adder and floating point multiplier. Each lane has five read ports and three write ports (5R3W). The general block diagram of the VREG is shown in Figure 3.2. The eight lanes can share one set of address decoders to generate the word select signals for different ports. In this way, eight lanes operate in parallel mode and perform vector instructions on the VREG. Thus, all function units are able to work concurrently.

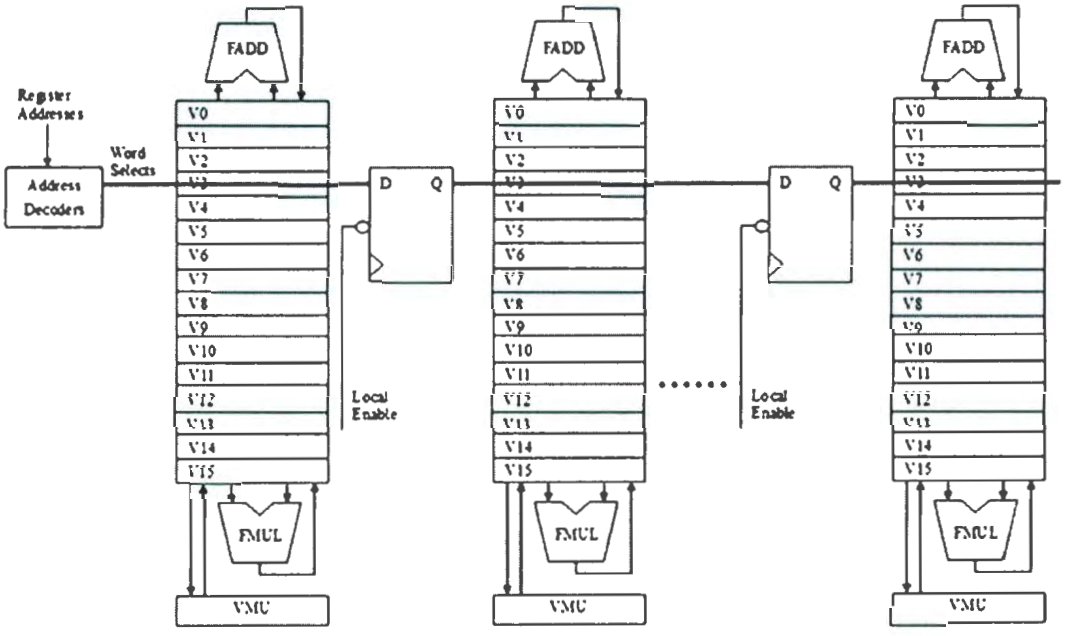

Figure 3.2: Block diagram of a vector register file (modified from a picture in [5])

3.3 Vector Memory Unit

The VREG can provide huge computational throughput to exploit the data parallelism. However, the limited memory throughput often saturates the VREG. Earlier vector supercomputers [17] use multi-port memory access interface to provide parallel data access service. The interleaved SRAM memory banks can provide high memory bandwidth with moderate latency [15]. To saturate the external memory bus, unit-stride loads and stores can transfer multiple words per cycle between memory and the vector register file and limited by the available ports into the vector register file. Using an unit stride load scheme, the multi-word can be loaded from different memory banks to VREG in one cycle. However, this unit-stride scheme requires an appropriate multi-bank memory system, and consequently often increases the cost and logic complexity.

For the FPGA device, the number of I/O ports limits the memory access interface. Therefore, VFPU only has one set of ports to access external memory. Single memory access interface requires two steps for data transfer between external memory and VREG: access/align operation between external memory and local data bus, and load/store operation between the local data bus and VREG. In this way, the load/store units can work as a cache for the VREG. Alignment will increase the data density and improve the flexibility, but extra registers are required and the operating frequency of whole system will be decreased. However, the access/align operation simplifies the memory interface and easily extend to a general data interface via fast interconnection technology, such as HyperTransport [18] and Gigabit ethernet. In this way, the data can be easily transferred between different source devices.

The separate load and store modules will transfer data between local data bus and VREG. A special First-In First-Out(FIFO) register with two different bit-width data ports will be used to implement the load and store module. The separate load/store module not only avoids the structure hazard, but also efficiently reduces the complexity of control logic.

3.4 Vector Arithmetic Units

As VFPU focuses on selected scientific applications, vector arithmetic units (VAUs) include two primary function units: vector floating point adder (VFADD) and vector floating point multiplier (VFMUL). Each VAU includes eight scalar function units, each with two dedicated read ports and one write port to access VREG. Together, VAUs can sustain sixteen single precision floating point operations per cycle.

Using a pipelined structure, the throughput of a floating point arithmetic operation can be improved. As described in Chapter 2, the time complexity of a FADD is

comparable to that of a FMUL. Therefore, the number of pipeline stages for floating point addition and multiplication can be similar. In this way, VAUs have the same startup overhead and a synchronous execution pace.

Fine grain division of pipelined implementation increases the work frequency, and introduces extra latency for internal data transfer. For the FPGA platform, the large disparity between logic delay and interconnection delay limits the pipeline division. In Chapter 5, we will compare the performance of pipelined floating point adders and multipliers with different pipeline divisions.

3.5 Chaining

Vector chaining is a key feature of vector processing units. The data will be directly transferred between different vector function units without writing back to a register. In this way, the execution of earlier vector instructions can overlap with a subsequent vector instruction. In VFPU, the vector chaining has three types depending on the different execution sequence. Figure 3.3 illustrates these three chaining modes.

- Load and arithmetic operation chaining: the vector data is loaded from the vector load/align unit, and fed to a subsequent arithmetic unit.
- Arithmetic operation and arithmetic operation chaining: the result from arithmetic unit is fed to subsequent arithmetic units.
- Arithmetic operation and store chaining: the result from arithmetic unit is fed straight to the store unit.

For a pipelined implementation, vector function units, VMUs and VAUs, run at the same frequency, and the chaining data transfer does not need extra buffers. Figure

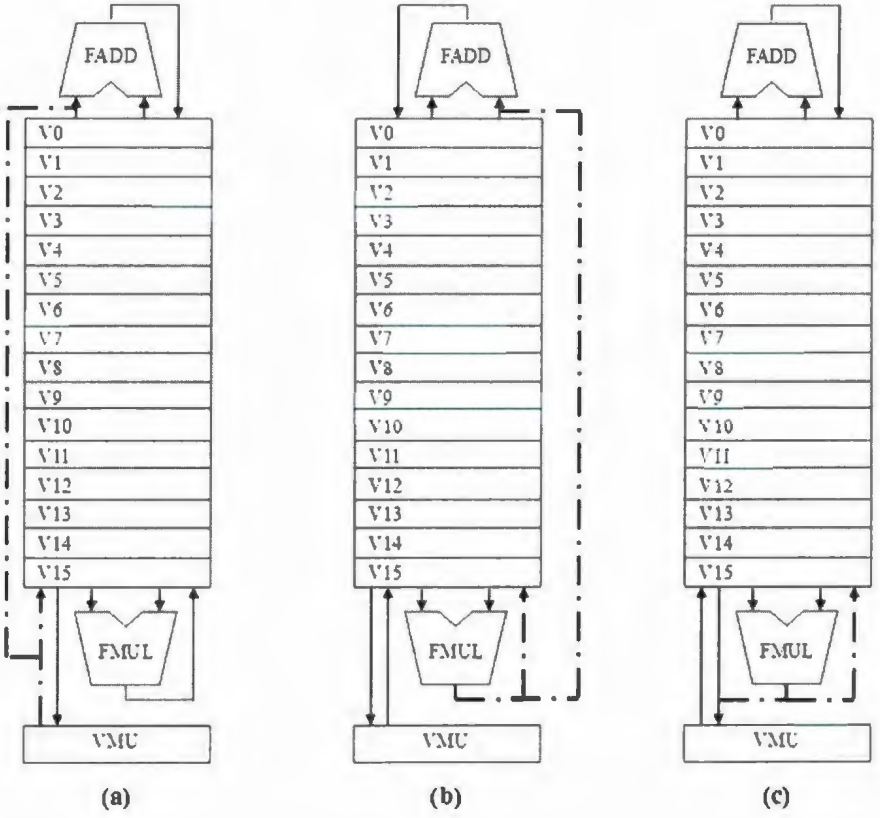

Figure 3.3: Three major chaining modes: (a) arithmetic operation after loading, (b) arithmetic operation after arithmetic operation, and (c) storing after arithmetic operation

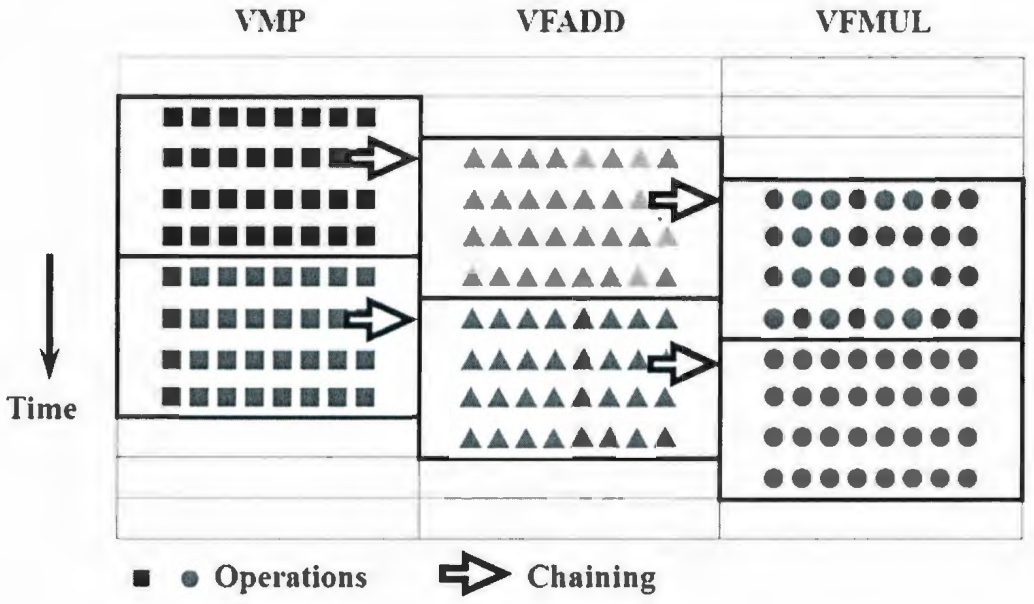

Figure 3.4: The overlapped execution process (modified from a picture in [5])

3.4 shows the overlapped execution process. When vector data is loaded to VMP on the first cycle, a vector add is ready in the subsequent cycle to VFADD. Similarly, a vector multiply is ready in the third cycle to VFMUL. This overlapped execution scheme increases the operation density and forms a long pipeline.

In the chaining mode, the data will remain in the fast storage, VREG, for a long period, and more arithmetic operations can be directly performed on the local data set. This extension of function execution process reduces the proportion of external memory access. Considering the large disparity of speed between execution units and external memory access, chaining is the important feature affecting the speed of a vector processing unit.

3.6 Summary

This chapter present the vector architecture of VFPU, which contains vector register file (VREG), vector memory unit (VMU), vector floating point adder (VFADD), vector floating point multiplier (VFMUL), control logic unit, and address generator. In the general prototype, three vector functional units, VMU, VFADD, and VFMUL, are structured as eight parallel lanes, and communicate via the central vector register file. The control logic unit will control the access sequence to avoid hazards. A flag register is used in the control logic unit to indicate the calculation sequence. The specific control pattern for the selected application is loaded into the flag register before running. Following these status codings of the specific control pattern, the appropriate control signals are generated and sent to the vector function units and the address generator. Additionally, the modern FPGA device provides an embedded processor, which can be set as a flexible control unit and complete more dedicated control tasks.

Chapter 4

FPGA Implementation

The FPGAs are the inexpensive and reconfigurable platform to prototype and verify the vector processing architecture in hardware. Modern FPGAs have tremendously increased both in terms of gate count and circuit speed. Large on-chip block memory provides an abundant local storage, and embedding dedicated arithmetic units and general purpose processor cores easily enable high performance computing. Moreover, the FPGAs design can incorporate additional hardware and software to monitor any logic transaction at run time. This observability enables easy verification of the correctness and the efficiency of the vector processing architecture. Considering the advantages of cost, power, speed, flexibility, observability, reproducibility, and credibility, FPGAs become an attractive platform for implementing the parallel processing structures [19].

This chapter mainly discuss the implementation platforms, appropriate hardware structures, and optimization methods. In Section 4.1, the basic methodology for FPGAs is introduced. Two main FPGA manufacturers, Altera and Xilinx, have different architectures for their FPGAs. Section 4.2 gives a detailed description for each component of VFPU. Section 4.3 describes an extension of the VFPU. An embedded

system is configured based on the VFPU. Section 4.4 proposes an application example and explains the overlap execution process for that specific calculation instance.

4.1 Design Methodology for FPGAs

The top-down design approach is a common design method for a digital system. The main steps of digital design are: behavior description, RTL coding, function simulation, synthesis, formal verification, static timing analysis, placement and routing, and configuration and verification.

- behavior description: The I/O interface is determined, and the function block diagram can help to define the relationships among different modules.
- RTL coding: The RTL coding mainly specifies the architecture of the design and detail of logic functions. The embedded dedicated arithmetic modules and on-chip block memory should be applied to the implementation for optimization.
- Simulation: Simulation is used to verify the functionality of the design. The corresponding test bench is designed, and the test data set is generated for different testing schemes.
- Synthesis: Synthesis is not only converting the RTL description into a low-level implementation consisting of primitive logic gates, but also estimates the resource usage and the time delay.
- Static timing analysis: Static timing analysis is used to compute the expected timing of the design. The path from the input to the output with the maximum delay is called the critical path. The difference between the required time and the arrival time is called the slack. Large negative slack implies that the path

is too slow; otherwise, the path is fine, if the slack is a small negative number or zero.

- Placement and routing: Placement is the process of assigning the design components to the chip's core area and can determine the total wire length, timing, and resource congestion. Routing is the next process and generates the wires to properly connect all the placed components. After the placement and routing, the verification should be done for testing.

- Configuration and verification: FPGA devices use the configuration memory to define the lookup table (LUT) equations, signal routing, IOB voltage standards, and all other aspects of the specific design. These configuration memory cells will be volatile and must be configured on power-up. For the specific FPGA chip, a corresponding configuration pattern will be generated to program configuration memory, instructions for the configuration control logic and data for the configuration memory. This binary configuration pattern is called bitstream, which can be delivered to the target chip through one of the Joint Test Action Group (JTAG), SelectMAP, or Serial configuration interfaces. After configuration, we can use a logic analyzer and appropriate software debugger to verify the logic correctness.

From the initial concept to specification, through block-level design using top down methodology to the implementation and verification, the FPGAs implementation will achieve the final design requirements. Altera and Xilinx are the two market leaders in the FPGA industry. We will briefly discuss their architectures and main features in the following parts.

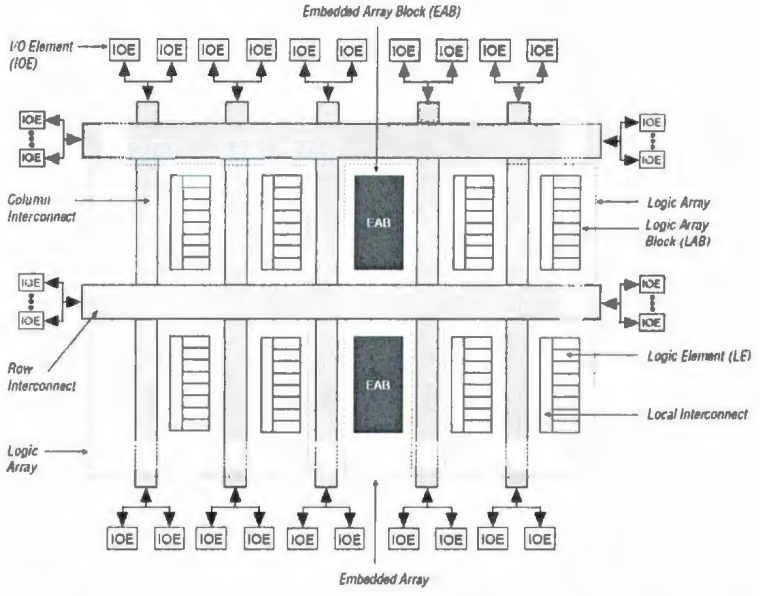

Figure 4.1: Block diagram of Altera FLEX 10K FPGAs [20]

4.1.1 Altera FPGA Family

Altera Corporation mainly provides programmable logic solutions. The Altera FPGA family includes FLEX series, Stratix series, Cyclone series, Arria series, and Hardcopy series. As FPGA density grows rapidly, the on-chip memory size has increased to over 1M Byte, and the number of dedicated fast adders and multipliers has increased also. For high performance applications, Altera introduced an embedded processor, Nios, on Stratix II, Cyclone, and Hardcopy FPGAs.

Figure 4.1 shows the block diagram of Altera FLEX 10K FPGAs [20]. The basic logic function unit is the logic element (LE). In each LE, a 4-input lookup table is used to encode any 4-input Boolean function. To implement more complex logic functions, a logic array block connects a set of LEs using the local interconnect bus. Recently, Altera created an adaptive logic module (ALM) in its recent FPGAs. Each ALM contains a variety of LUTs that can be divided between two adaptive LUTs

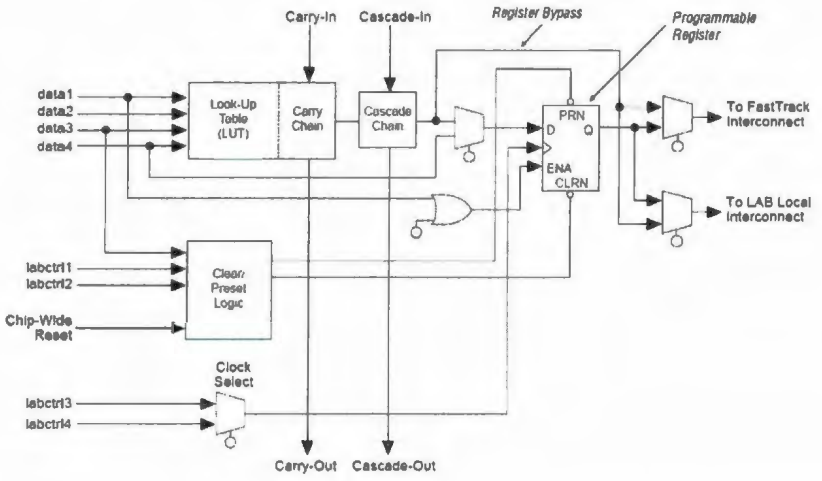

Figure 4.2: Logic element structure of Altera FLEX 10K FPGAs [20]

(ALUTs). With up to eight inputs to the two ALUTs, one ALM can implement various combinations of two functions. This adaptability not only provides advanced features with efficient logic utilization, but also reduces the power consumption [21]. To speedup the arithmetic operations, Altera FPGAs provide a quick carry chain for adders, and embed multipliers and digital signal processing (DSP) modules. These dedicated arithmetic units not only optimize the arithmetic functions, but also provide fast paths for data transfer. Additionally, Altera FPGAs include rich interconnection resources: local, row, column, carry chain, shared arithmetic chain, register chain, and direct link interconnects. These interconnection resources provide efficient hierarchical interconnect solutions.

The Altera Quartus II is a complete design environment and it easily combines with other simulation and timing-analysis software. This develop environment can complete the syntax analysis, synthesis, and timing analysis. The synthesis result can help the designer to choose the appropriate FPGA serial product and assign the I/O pins.

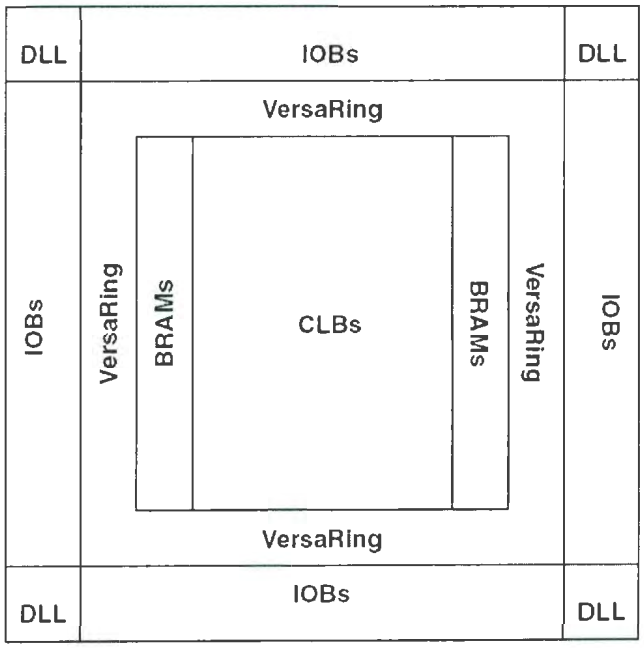

Figure 4.3: Block diagram of Xilinx Virtex FPGAs [22]

4.1.2 Xilinx FPGA Family

Xilinx Corporation is another leading FPGA manufacturer and provides comparable programmable logic solutions. The Xilinx FPGA family consists of the Virtex series, Spartan series, and EasyPath series. To improve the embedded processing performance, Xilinx also provides dedicated fast adders, embedded multipliers, and rich on-chip block memory. A 32-bit soft processor, MicroBlaze, can provide a flexible processor, and the PowerPC 32-bit hard processors can meet high performance requirement.

Figure 4.3 shows a block diagram for Xilinx Virtex FPGAs. For Xilinx FPGAs, the basic logic function unit is the slice, which includes two logic cell (LC), as shown in Figure 4.4. Each LC consists of a 4-input LUT, carry logic, and a storage element. The Configurable Logic Block (CLB) collects a set of slices to implement some specific

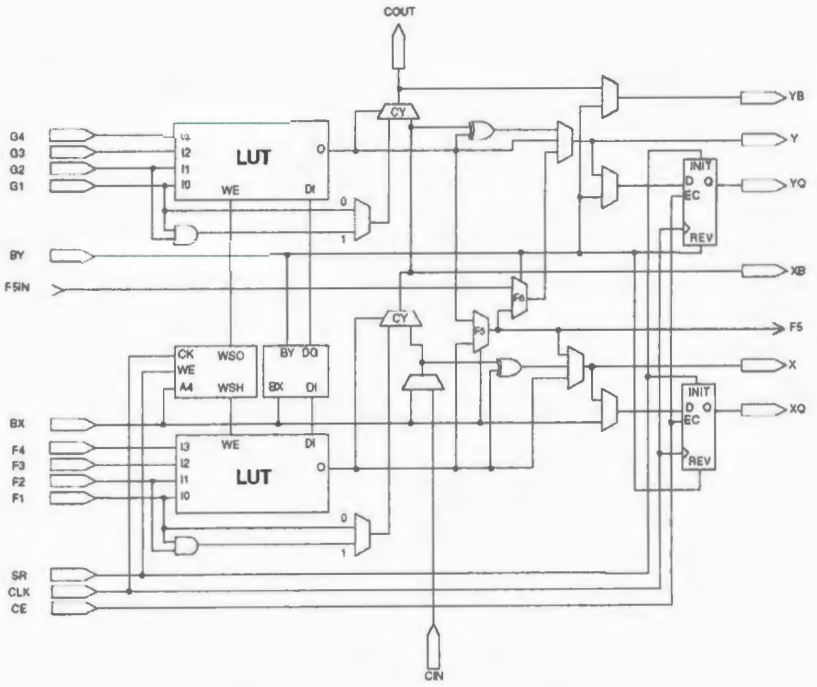

Figure 4.4: Slice structure of Xilinx Virtex FPGAs [22]

logic functions [22]. Therefore, the number of slices is often used to estimate the resource utilization. The two-LUT configuration for a slice not only improves the performance of basic logic units, but also reduces the interconnection cost by using fewer logic units for a specific design. In Chapter 5, we will compare the performance of FADD and FMUL with the different FPGA devices: the Altera Cyclone II and the Xilinx Virtex II Pro.

Xilinx developed a powerful automatic design software system for logic design, the ISE foundation, and an integrated development environment for embedded system design, Platform Studio and the EDK. ISE foundation mainly completes the syntax checking, synthesis, timing analysis, and generating the binary stream file for FPGA configuration. Platform Studio and the EDK combine the hardware configuration and software design. Using these development kits, the designer not only can easily

implement hardware design on FPGAs, but also can extend the design to an embedded system.

Comparing the performance of FADD and FMUL on Altera and Xilinx FPGAs, we choose the Xilinx Virtex II Pro (XC2VP100-6-FF1704) as the target device for VFPU. Using the $0.13\mu m$ CMOS nine-layer copper process, the Embedded IBM PowerPC 405 RISC processor blocks are integrated in Virtex-II Pro XC2VP100 to optimize high performance designs. The Block SelectRAM memory modules provide large 18 Kb storage elements of true Dual-Port RAM, and the total on-chip memory size is up to 7,992 Kbits [23].

4.2 VHDL Models on FPGAs

Figure 4.5 shows the RTL schematic diagram of the VFPU, which contains a VREG, a VMU, a VFADD, a VFMUL, a control logic unit, and an address generator. The data interface to the VFPU is the memory data port, MDATA[255 downto 0]. The corresponding address interface is the memory address port, MADD[21 downto 0]. For the VFPU, the external memory module is implemented by Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM), which can transfer data on the rising and falling edges of the clock signal. The external memory interface has a 256 bit data bus, and a 22 bit address bus, supporting up to 128 MB of industry standard DDR SDRAM. This 128 MB external memory is configured as two banks of 64 MB each.

The control signals include a reset signal and an enable signal, which will be mapped to the control logic unit and start or clear the calculation process. Using the global clock resource on FPGAs, the dedicated clock input is directly feed to the low-skew buffers inside the FPGA for routing clocks. The two flag signals, RW and

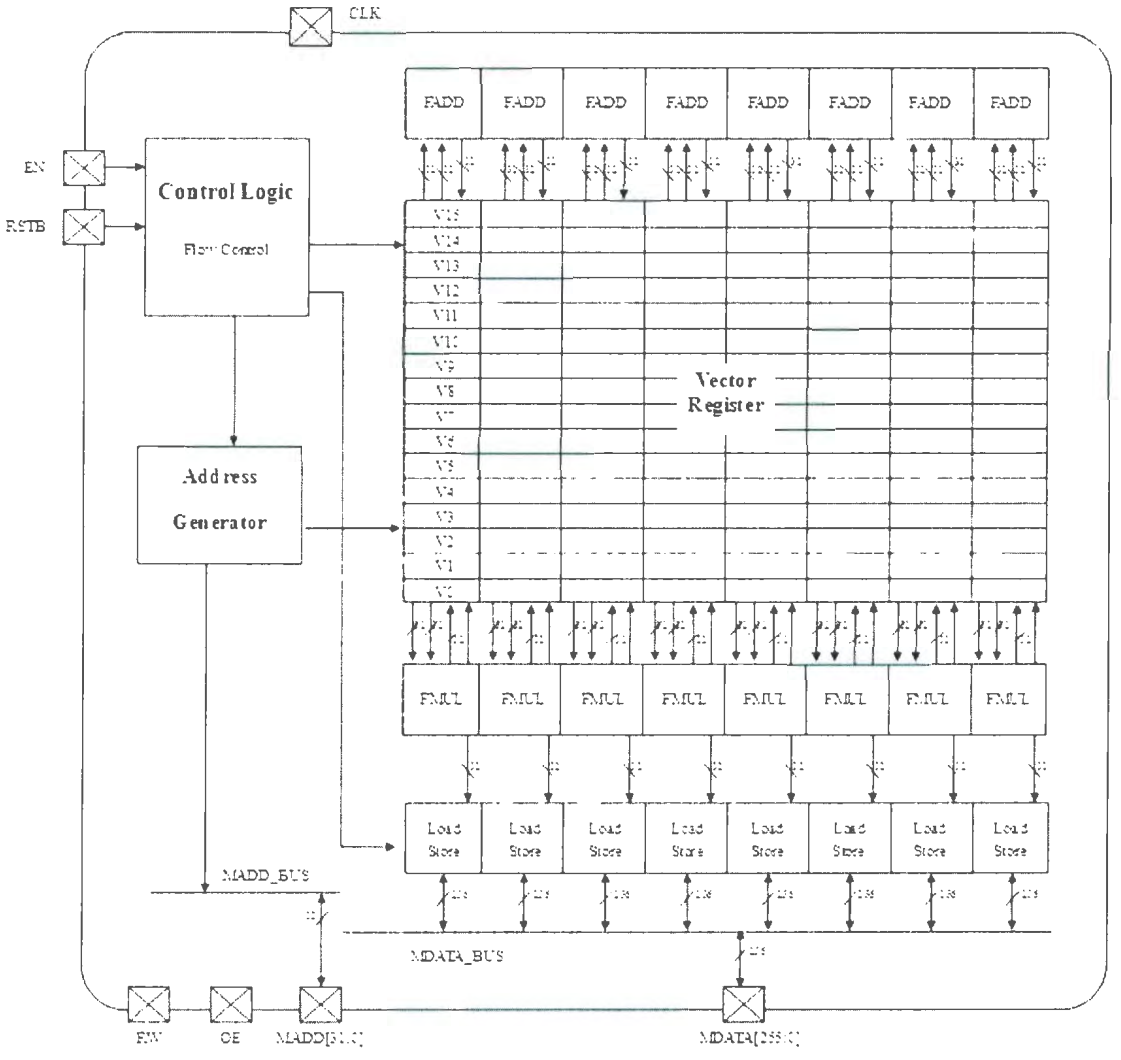

Figure 4.5: VFPU RTL schematic diagram (modified from a picture in [5])

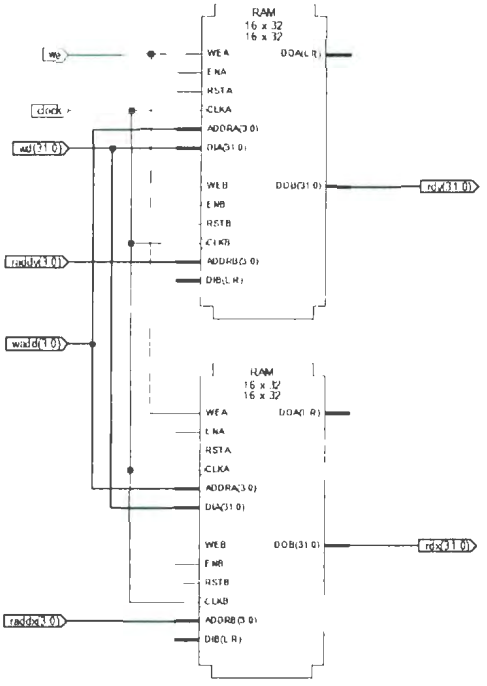

Figure 4.6: Three port data bank structure

OE, will indicate the status of the memory interface.

4.2.1 Register Files

The on-chip block memory of the Xilinx Virtex II Pro FPGA can be configured as a true dual port RAM module, which includes two sets of address inputs for two independent ports. As shown in Figure 4.6, the two dual port RAM modules are combined to form a three-port data bank, which includes 16×32-bit data elements. Based on this three-port data bank, we can implement the 5R3W register file lane with extra control logic, which will map the word selection signal to the different access ports.

Figure 4.7 shows the schematic diagram of the vector register file within one lane. In this design, sixteen three-port data banks, each with two read ports and one write

port, are used to support two VAUs and one VMU with a total requirement of 5 reads and 3 writes per cycle.

4.2.2 Arithmetic Modules

In VFPU, the arithmetic units are FADD and FMUL. The combinational logic designs of the floating point arithmetic units are implemented first. Based on the timing analysis of the combinational logic design, the number of stages for the pipelined design will be determined. For the combinational logic design, the speed depends on two characteristics: transition time and propagation time. The transition time is the amount of time that output of a logic unit takes to change from one state to another [6], and will be determined by device technology parameters. The propagation delay is the amount of time that it takes for a change in the input signal to produce a change in the output signal [6], and the longest propagation delay of a particular path through the overall circuit is called the critical path delay.

The basic arithmetic components, such as fixed point two's complement adder, barrel shifter, carry save adder Wallace tree, and rounding module, have been developed. Using these components, the appropriate hierarchical VHDL models of the floating point arithmetic units described above are implemented. These VHDL models have been verified via simulation and synthesized to FPGA devices. To reduce the critical path delay of the fixed point adder, the carry lookahead adder is always used for fast carry out calculation. However, the interconnect delay of the carry lookahead adder on FPGAs rapidly increases with the number of the logic components. Taking advantage of dedicated carry chains, the ripple carry adder is a little faster than the carry lookahead adder on FPGAs.

A common implementation of the fixed point multiplier is using the Booth's coded

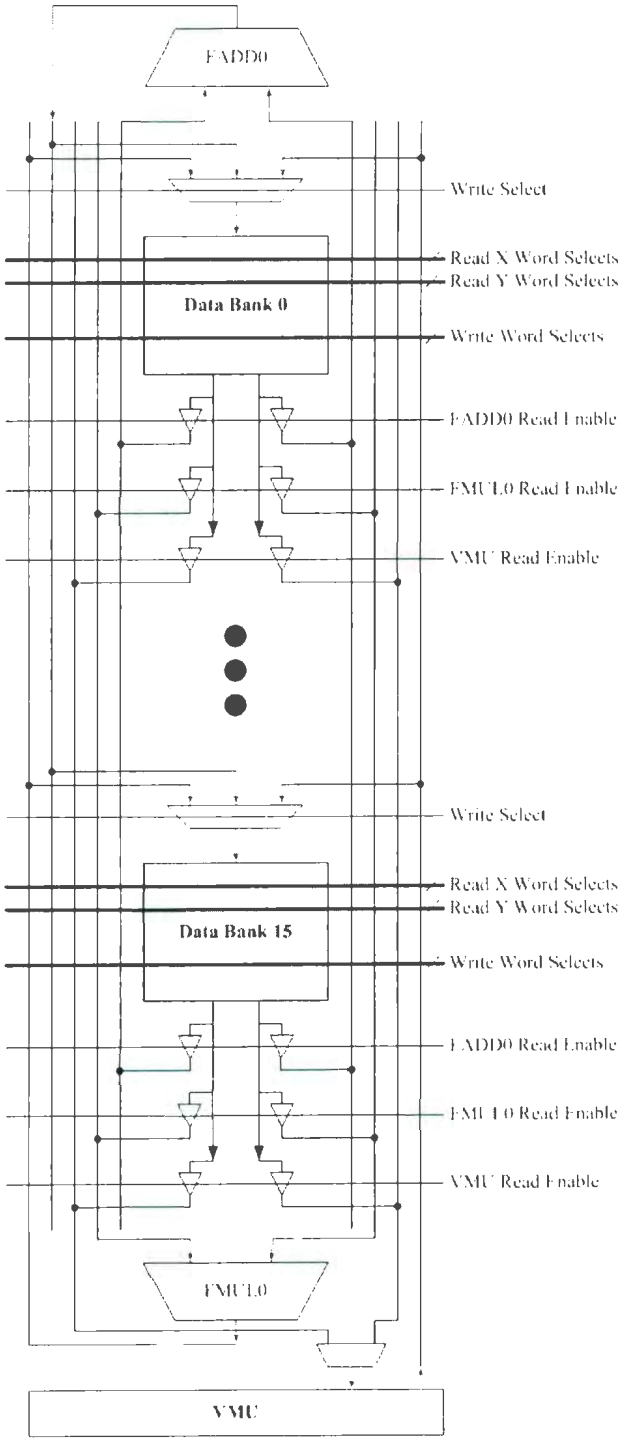

Figure 4.7: The schematic diagram of vector register file within one lane (modified from a picture in [5])

Wallace carry save adder tree [11] to reduce the critical path delay. However, the Wallace tree structure not only occupies large area on the FPGA chip, but also introduces significant interconnection delay. As fast embedded fixed-point multipliers are embedded on FPGAs, the various word width multipliers can be formed by Karatsuba Multiplication formulations [12], and the fast interconnection paths can guarantee an efficient data transfer rate between the function units.

In an effort to achieve high throughput of the floating point arithmetic operations, we use a deeply pipelined structure for the floating point arithmetic units. The FADD is broken into 13 stages, and the FMUL is broken into 8 stages. Although the deeply pipelined structure will cause long latency, the throughput of the floating point arithmetic units is significantly increased. In a pipelined design, a set of data sub-operation components is connected in series, and these components can be executed in parallel. The slowest part determines the pipelined design speed. An output register is used to synchronize data for every stage, because the critical path delay of different stages is different. To avoid metastability, the inputs are held constant for specified periods before and after the clock pulse.

To verify the functional correctness, a special testbench has been developed, and a Visual C++ program has been developed to generate extensive test patterns. This application can transform a random number from decimal format to IEEE 754 binary format, and calculate the correct result for different floating point operations. The test patterns contain random values and extremity values, such as zero, infinity, NaN, maximum value, and minimum value. The test pattern can be generated automatically for an arbitrary size of the data set and stored in IEEE 754 binary format.

As visual examples of the arithmetic operations, the simulation waveforms are produced by ModelSim.

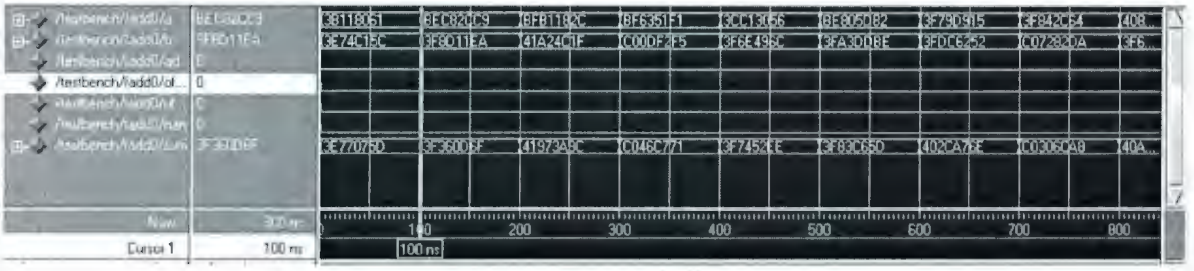

Figure 4.8: A simulation waveform for floating point adder

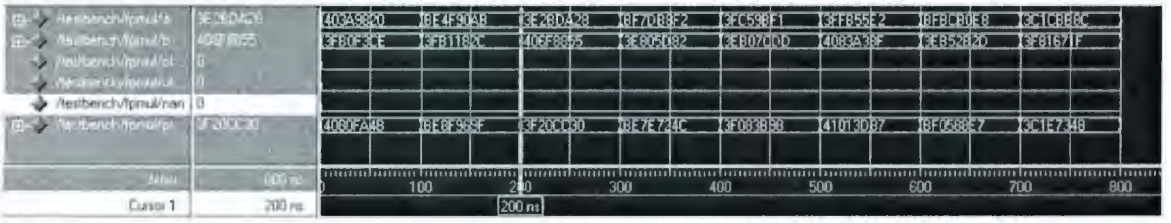

Figure 4.9: A simulation waveform for floating point multiplier

Figure 4.8 shows this situation with 'addsub' signal low, specifying the addition operation. Consider the following operands:

BEC82CC9 (hex, IEEE 754 single precision) = -0.390967 (dec),

3F8D11EA (hex, IEEE 754 single precision) = 1.102109 (dec).

The sum is: 3F360D6F (hex, IEEE 754 single precision) = 0.711142 (dec).

Figures 4.9 shows a simulation waveform of the multiplication operation. Consider the following operands:

3E2BDA28 (hex, IEEE 754 single precision) = 0.167824 (dec),

406F8855 (hex, IEEE 754 single precision) = 3.742696 (dec).

The product is: 3F20CC30 (hex, IEEE 754 single precision) = 0.628116 (dec).

Figures 4.10 shows a simulation waveform of the multiply-add-fused operation. Consider the following operands:

3E2BDA28 (hex, IEEE 754 single precision) = 0.167824 (dec),

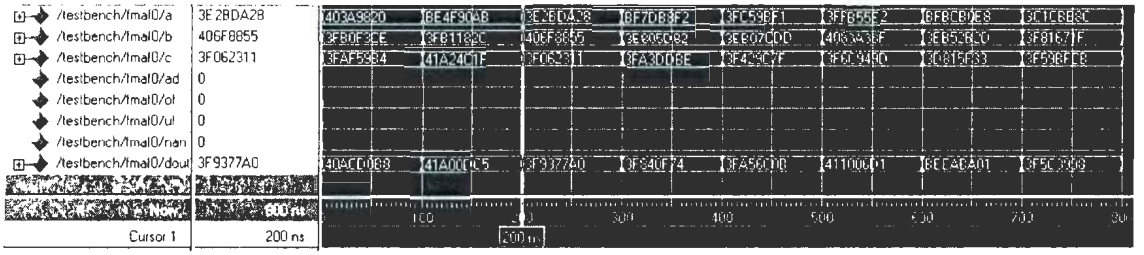

Figure 4.10: A simulation waveform for floating point multiplier

406F8855 (hex, IEEE 754 single precision) = 3.742696 (dec),

3F062311 (hex, IEEE 754 single precision) = 0.523973 (dec).

The result is: 3F9377A0 (hex, IEEE 754 single precision) = 1.152088 (dec).

4.2.3 Memory Access Units

The memory access units include the load module and the store module, which are implemented by a special FIFO register with two different bit-width data ports. Figure 4.11 shows the structure of the load module. The load FIFO contains sixteen dual port RAM modules, each including two 32 bit data elements.

The rotate read scheme is used to feed vector data into the vector register file. In the first cycle, the first 256 bit vector data is fed to the first load module, and then the load module starts to feed each 32 bit data element to a vector register in eight cycles. During these eight cycles, the sequential 256 bit vector data is rotationally fed to eight load modules. In this way, the eight lanes work concurrently and independently.

4.3 Embedded System Configuration

Based on the VFPU prototype, we can combine the on-chip PowerPC 405 processor to configure an embedded system, which can support an embedded Linux system

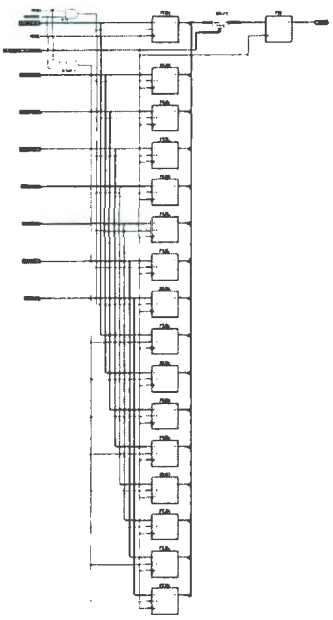

Figure 4.11: The load FIFO register with two different bit-width data ports

and provide more flexibility and applicability. Xilinx EDK software integrates the hardware configuration and software design. For the Amirix AP1000 PCI platform FPGA development board, we can configure the hardware specification: the embedded processor and the bus clock frequency, on-chip cache size, I/O devices, and debug interface.

Figure 4.12 shows a configuration of the embedded system. The embedded processor and high speed components are connected by the processor local bus (PLB), which is a high-performance 64-bit address bus and a 128-bit data bus [24]. The embedded processor will generate the flow control pattern and update the flag register of VFPU via the PLB. Meanwhile, the embedded processor will control the DDR controller to feed or update data from the data cache.

Based on the embedded Linux system, we can directly use the C language to describe the specific calculation model. In this way, the most numerically-intensive

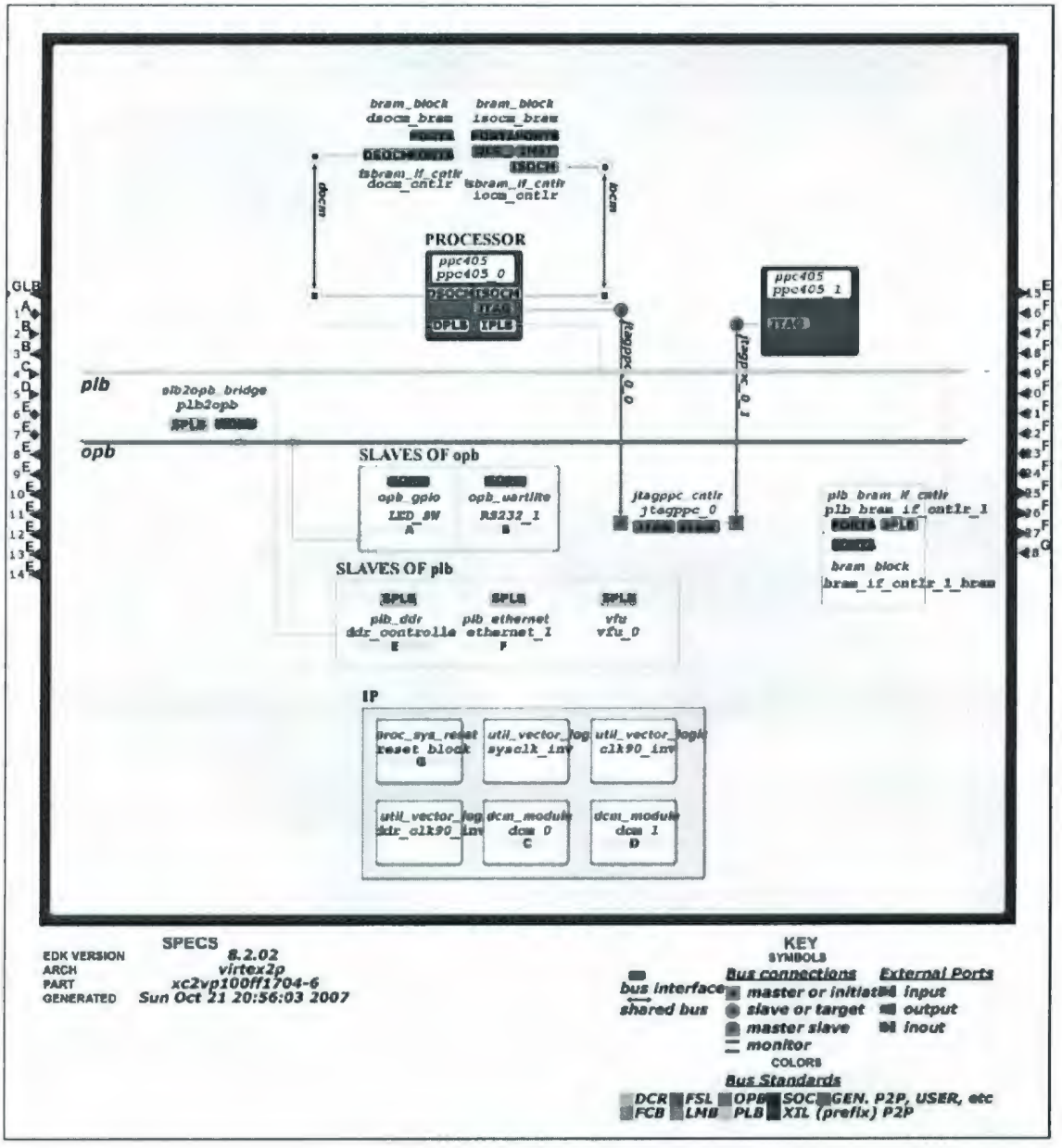

Figure 4.12: Block diagram of an embedded system configuration

applications can be easily vectorized to improve the performance by using the VFPU.

4.4 Application Example

The VFPU can be used to implement numerically-intensive problems on an FPGA prototyping board. For instance, the Earth Simulator contributes to predict environmental changes by analyzing the vast volume of observation data. A general operation for seismic applications is the distance calculation in 3D space. An illustration of this calculation is given below:

$$D^2 = (X_i - X_j)^2 + (Y_i - Y_j)^2 + (Z_i - Z_j)^2. \tag{4.1}$$

Here, (X_i, Y_i, Z_i) and (X_j, Y_j, Z_j) denote two sets of points. The square operation can be replaced by self-multiplication. The steps of distance calculation are:

1 : Load X_i,

2 : Load X_j,

3 : Subtract X_j from X_i,

4 : Load Y_i,

5 : Load Y_j,

6 : Subtract Y_j from Y_i,

7 : Load Z_i,

8 : Load Z_j,

9 : Subtract Z_j from Z_i,

10 : Multiplication of X coordinate value,

11 : Multiplication of Y coordinate value,

12 : Multiplication of Z coordinate value,

13 : Sum of X^2 and Y^2,

14 : Sum of Z^2 and $(X^2 + Y^2)$

15 : Store result

Assume the size of the point set is n, the latency of FADD is l_a, the latency of FMUL is l_m, and every operation uses one cycle.

For a scalar pipeline processing unit, the instructions will be issued one by one as shown in Figure 4.13, and the operation can not be overlapped. The total number of cycles is:

$$\begin{aligned} C_{scalar} &= (n + n + (n + l_a) + (n + l_m)) \times 3 \\ &\quad + (n + l_a) + (n + l_m) + n \\ &= 15n + 4l_a + 4l_m. \end{aligned} \tag{4.2}$$

The vector processing unit can execute these steps in an overlap mode, and the chaining scheme can directly transfer data between different vector function units without writing back to registers. Moreover, the eight lanes in the vector processing work in parallel and can deal with eight data sets. Therefore, the vector processing unit can work almost seventeen times as fast as the scalar processing unit. Figure 4.14 illustrates the overlap execution flow in one lane of the vector processing unit. The total number of cycles used in one lane of the vector processing unit is:

$$\begin{aligned} C_{vector} &= n \times 3 + (n + l_a) \times 4 \\ &= 7n + 4l_a. \end{aligned} \tag{4.3}$$

Note that the two vector operands will be loaded in serial mode. If we add an extra load port to each lane, the two vector operands can be loaded in parallel, and the total number of cycles can be reduced to $5n + 5l_a$. Figure 4.15 shows the overlap

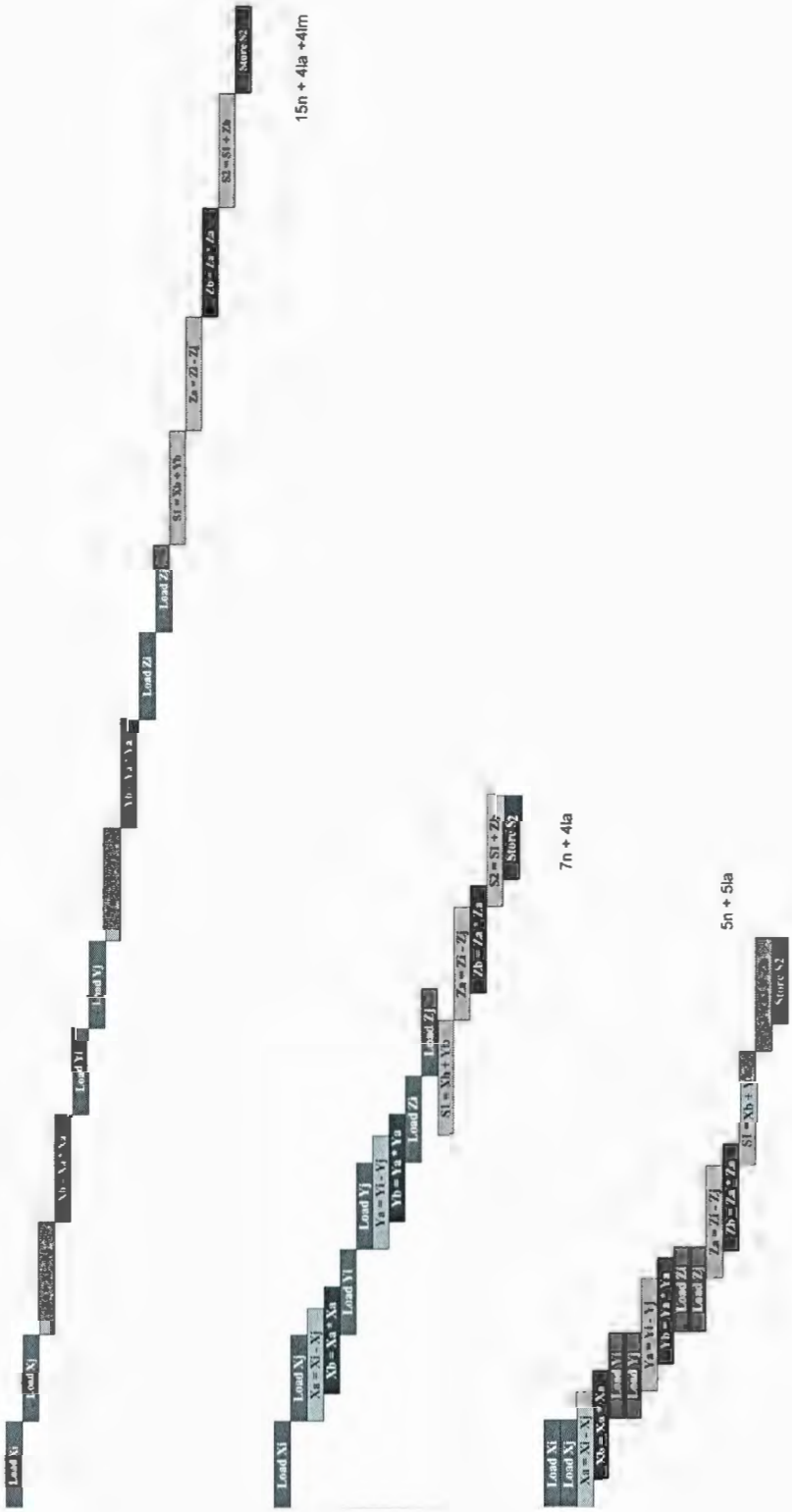

Figure 4.13: The comparison among three different execution flows

Figure 4.14: The sequential execution flow

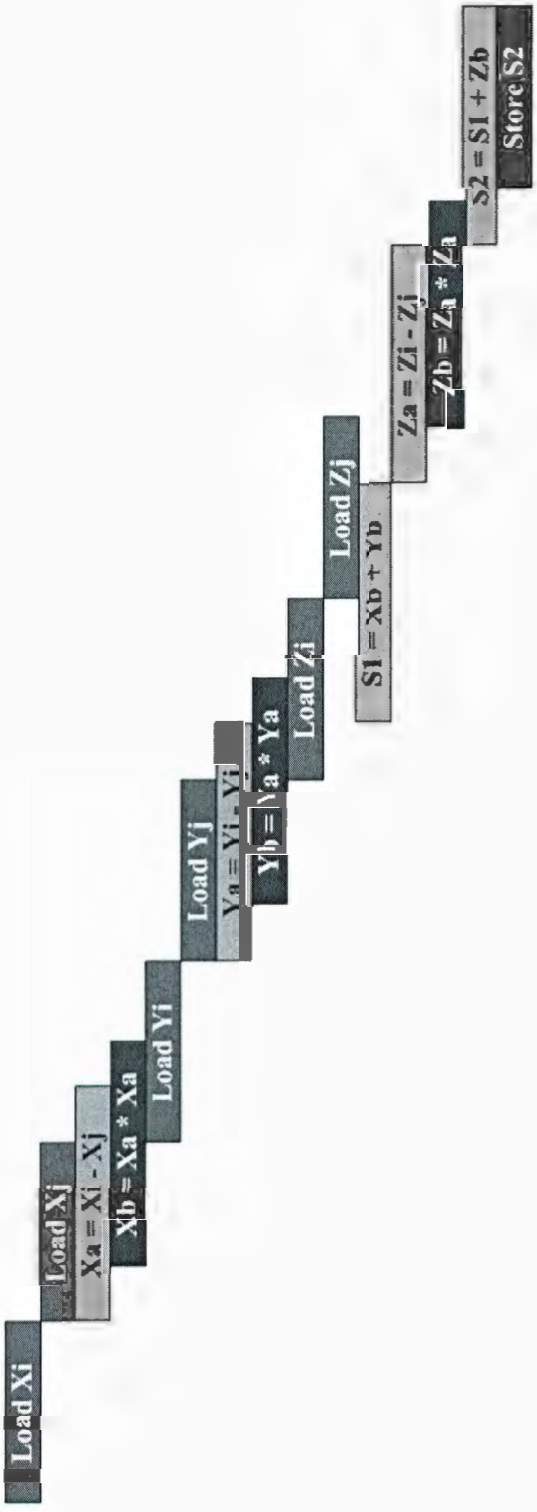

Figure 4.15: The chaining overlap execution flow

execution flow in one lane of the vector processing unit with two loaders. However, the extra load ports not only increase the number of I/O ports and control logic, but also increase the disparity of the throughput between local storage and external memory.

Comparing these three execution processes, we can conclude that the overlap significantly improves the efficiency and speedup the execution process. Using the chaining scheme, partial data can be directly passed to the next function unit without register access. Therefore, we not only reduce one cycle from the two overlapped steps, but also may save considerable space on the register file. Moreover, as the chaining scheme can keep the data in the vector register file for a long period, more arithmetic operations can be directly performed on the local data. The extension of function execution process reduces the proportion of external memory access. Considering the large disparity of speed between execution units and external memory access, chaining is the major feature affecting the speed of the vector processing unit.

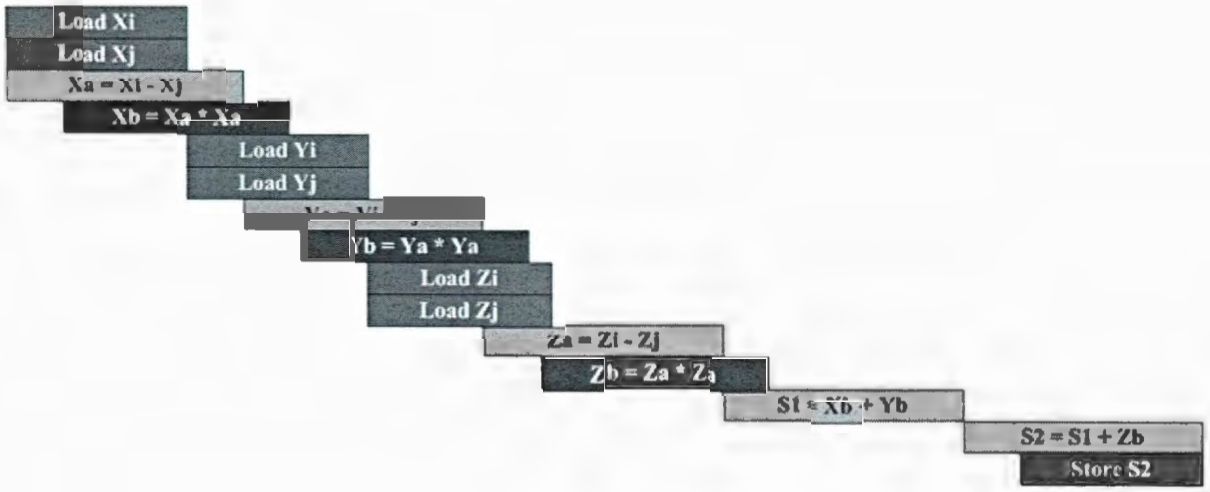

Figure 4.16: The chaining overlap execution flow with two loaders

Chapter 5

Performance Analysis

The main objective of the VFPU design is to minimize critical path delay with a reasonable resource utilization. The synthesis results of the FPGA implementations in terms of timing and resource utilization will be presented in this chapter, and then the analysis and timing optimization methods will be discussed.

First, we will focus on the timing performance of the combinational design for different arithmetic units. Second, we will compare the peak work frequency of pipelined designs with different numbers of stages. Third, we will analyze the performance for VFPU with different lane configurations. In the end of this chapter, we will compare the proposed design with other related work.

5.1 Performance Analysis for Combinational Implementations

This section presents the synthesis result of the combinational design for different function components. These combinational Implementations are synthesized for both Altera and Xilinx FPGA devices. The Altera's target FPGA device (EP2C20F484C7)

belongs to the Cyclone II Family, which provides 68,416 logic elements, up to 622 usable I/O pins, and up to 1,152 Kbits of embedded memory [25]. The synthesis result is generated by the Altera's integrated development environment software, Quartus II 7.2. The Xilinx target FPGA chip (xc2vp100-6ff1704) is one member of the Virtex-II Pro Family, which contains up to 99,216 logic cells and supports up to 1,164 user I/O pads and up to 7,992 Kbits of block RAM [23]. The synthesis process for Xilinx FPGA implementation is generated by the Xilinx's development kit, ISE 8.2i.

The fixed point adder modules will be analyzed first, because they are the basic components for every arithmetic unit. The timing performance of the fixed point adder has a great effect on the arithmetic units in VFPU. The fixed point multiplier with different structures will be discussed at the end of this section.

5.1.1 Ripple Carry Adder

For the fixed point adder, the propagation of carries is a major impediment to high-speed addition [9]. The ripple carry adder directly ripples down the carry lines of the 1-bit full adders to generate the carry out bit. Although the worst-case delay always grows linearly with the word width, a ripple carry structure is simple and suitable for compact applications.

Table 5.1 shows the timing performance and resource utilization of the ripple carry adder on Altera Cyclone II FPGA device. The pin-to-pin delay (t_{PD}) is the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin. The t_{PD} consists of the cell delay for combinational logic and the interconnect delay for the signal transmission. Table 5.1 indicates that the cell delay of the ripple carry adder linearly increases with the word width. Meanwhile, the interconnect delay also rapidly grows as the number of logic

Table 5.1: Performance of the Ripple Carry Adder on Altera Cyclone II

bits	*delay*	*logic*	*route*	*LEs*	*pins*
1	10.116 ns	4.177 ns (41.29 %)	5.939 ns (58.71 %)	4	5
4	11.856 ns	4.944 ns (41.70 %)	6.912 ns (58.30 %)	9	14
16	21.897 ns	8.063 ns (36.82 %)	13.834 ns (63.18 %)	33	50
28	28.851 ns	9.441 ns (32.72 %)	19.410 ns (67.28 %)	57	86
32	30.078 ns	10.729 ns (35.67 %)	19.349 ns (64.33 %)	65	98
48	44.592 ns	13.252 ns (29.72 %)	31.340 ns (70.28 %)	97	146

elements increases.

The performance of the ripple carry adder on Xilinx Virtex II Pro is shown in Table 5.2. And the Figure 5.1 illustrates the difference between Altera Cyclone II and Xilinx Virtex II Pro. In FPGA devices, various logic functions are implemented by 4-input lookup tables (LUTs). The combinational logic function of 1-bit full adder occupies four LUTs on both Altera and Xilinx FPGA devices, and their cell delays are very close. The logic cell delay on two FPGA devices is at same level, and the main difference of the critical path delay depends on the interconnect delay. To optimize the timing performance on an FPGA platform, we should not only optimize the architecture and logic functions, but also should consider how to reduce the extra interconnect cost.

5.1.2 Carry Lookahead Adder

To reduce the critical path delay of fixed point adders, a commonly used scheme is carry lookahead addition (CLA), which is a classical scheme featuring logarithmic

Table 5.2: Performance of the Ripple Carry Adder on Xilinx Virtex II Pro

bits	*delay*	*logic*	*route*	*Slices*	*LUT*	*IOB*
1	5.102ns	4.083ns (80.0%)	1.019ns (20.0%)	2	4	7
4	7.683ns	5.022ns (65.4%)	2.661ns (34.6%)	4	8	14
16	18.171ns	8.778ns (48.3%)	9.393ns (51.7%)	18	32	50
28	28.659ns	12.534ns (43.7%)	16.125ns (56.3%)	32	56	86
32	32.155ns	13.786ns (42.9%)	18.369ns (57.1%)	37	64	98
48	46.139ns	18.794ns (40.7%)	27.345ns (59.3%)	55	96	146

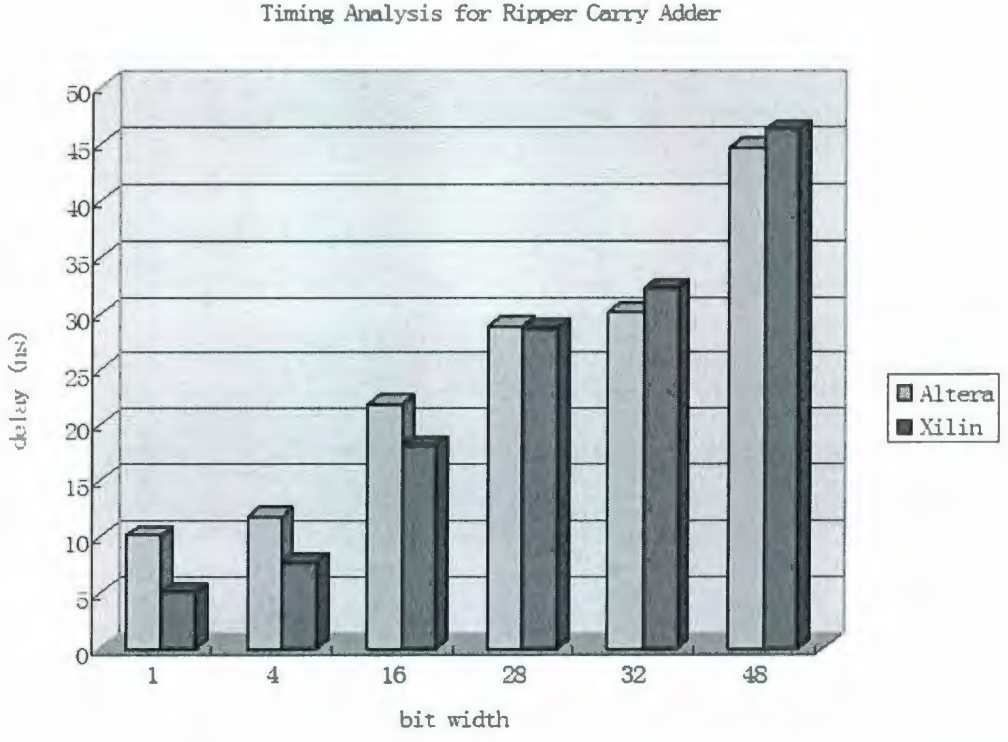

Figure 5.1: Comparison of critical path delay for Ripple carry adder on Altera and Xilinx FPGA devices

Table 5.3: Performance of the Carry Lookahead Adder on Altera FPGAs

bits	*delay*	*logic*	*route*	*LEs*	*pins*
1	10.116 ns	4.177 ns (41.29 %)	5.939 ns (58.71 %)	4	7
4	12.963 ns	5.296 ns (40.85 %)	7.667 ns (59.15 %)	14	16
16	19.778 ns	6.692 ns (33.84 %)	13.086 ns (66.16 %)	63	52
28	23.341 ns	6.605 ns (28.30 %)	16.736 ns (71.70 %)	105	86
32	24.447 ns	7.396 ns (30.25 %)	17.051 ns (69.75 %)	119	100
48a	25.787 ns	7.065 ns (27.40 %)	18.722 ns (72.60 %)	163	146
48b	25.775 ns	9.258 ns (35.92 %)	16.517 ns (64.08 %)	180	146

delay. However, the logic complexity of CLA rapidly increases with the word width, and the extra interconnect delay has to be considered.

Table 5.3 and Table 5.4 show the performance analysis on Altera and Xilinx platform, respectively, for the carry lookahead adder with different word widths. Similarly, the cell delays on two FPGAs are close, but the interconnect delay on the Altera FPGA device increases faster than the Xilinx FPGA device. As discussed in Chapter 4, the differences in the basic logic unit cause the disparity of interconnect delay. The Xilinx FPGA device provides two LUTs in one slice and greatly reduces the number of slices for a complex logic function. The extra basic logic units add extra interconnect cost to Altera FPGA implementations. Figure 5.2 and Figure 5.3 illustrate the proportion between cell delay and interconnect delay.

An interesting point of the synthesis result is the different delays between two 48-bit CLA with different structures. Figure 5.4 shows the block diagram of a 48-bit adder, CLA(b), constructed using three 16-bit CLAs, and Figure 5.5 illustrates the block diagram of a 48-bit adder, CLA(a), built using one 16-bit CLA and one 32-bit

Table 5.4: Performance of the Carry Lookahead Adder on Xilinx FPGAs

bits	*delay*	*logic*	*route*	*Slices*	*LUT*	*IOB*
1	5.102ns	4.083ns (80.0%)	1.019ns (20.0%)	2	4	7
4	6.746ns	4.709ns (69.8%)	2.037ns (30.2%)	8	16	16
16	10.803ns	6.607ns (61.2%)	4.196ns (38.8%)	33	61	52
28	22.854ns	10.962ns (48.0%)	11.892ns (52.0%)	47	84	86
32	26.319ns	12.214ns (46.4%)	14.105ns (53.6%)	54	97	100
48a	25.569ns	11.962ns (46.8%)	13.607ns (53.2%)	79	146	146
48b	20.028ns	10.057ns (50.2%)	9.971ns (49.8%)	95	173	146

CLA. Each 16-bit CLA includes 4 4-bit CLA and one lookahead carry generator. The 48-bit CLA(a) includes 12 4-bit adders and 5 lookahead carry generators, and the 48-bit CLA(b) includes 12 4-bit CLAs and 4 lookahead carry generators. Therefore, the critical path delay of 48-bit CLA(b) is slightly larger than 48-bit CLA(a). For a specific word width, the CLA has multiple possible configurations from the basic 4-bit CLA and lookahead carry generator, and the different structures have different effects on the logic complexity and route cost.

The logic level optimization is carried out in the Xilinx ISE platform during the synthesis process, and the optimization has different effects on different structures. Considering the optimization efficiency, the FPGA timing performance for CLAs not only depends on the word width, but also depends on the structure. Generally, the logic level optimization includes two phases: flattening and structuring. The flattening removes intermediate variables, simplifies boolean equations to a two-level logic mode, and reduces the logic delay. Moreover, the critical path can also be shortened by duplicating logic during flattening. Contrarily, structuring inserts intermediate

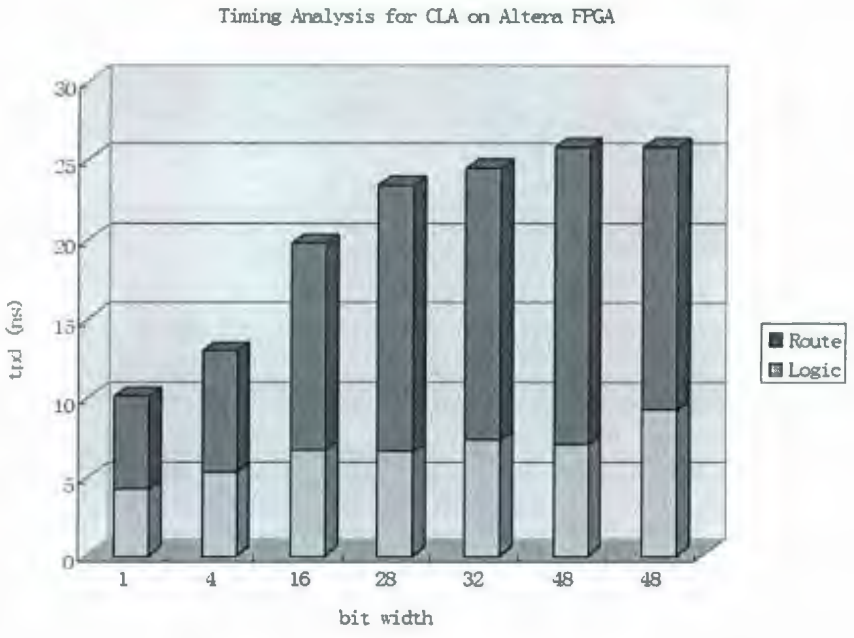

Figure 5.2: Delay proportion between logic and route for CLAs on Altera CycloneII

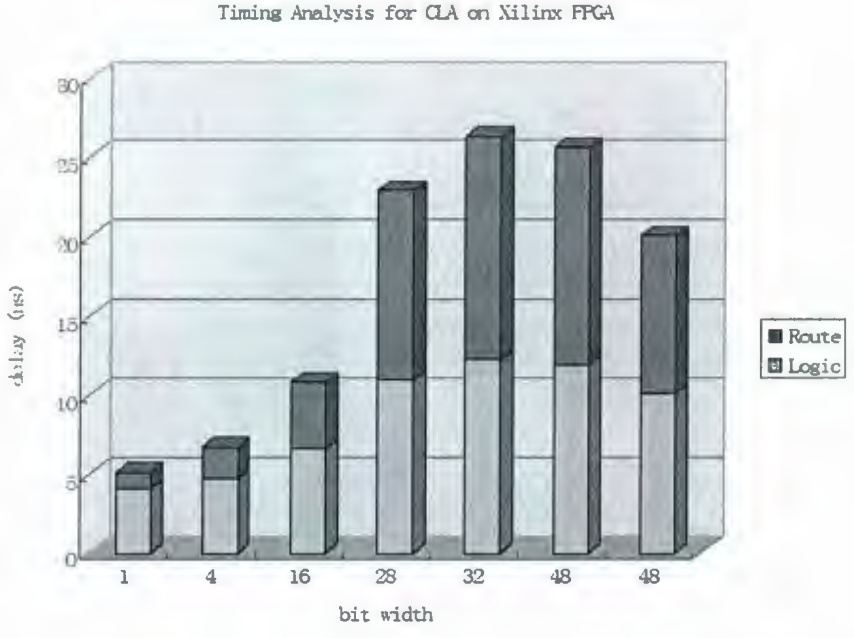

Figure 5.3: Delay proportion between logic and route for CLAs on Xilinx Virtex II Pro

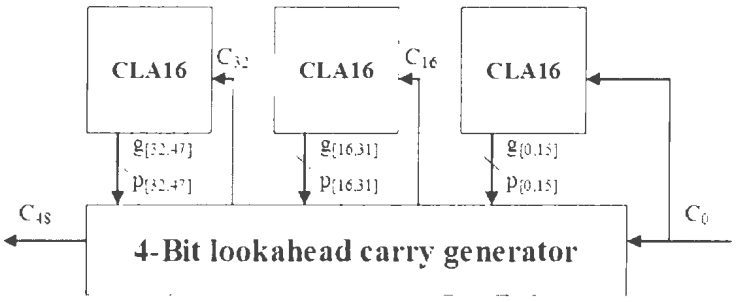

Figure 5.4: Building a 48-bit CLA from 12 4-bit CLAs and 4 lookahead carry generators

variables, generates multi-level logic modules, and saves logic resource. Through these optimization methods, the final logic level structure possibly achieves a balance point of timing and area with the specific optimization constraints. The aim of VFPU design is to generate a timing driven structuring implementation, and the Xilinx ISE optimization goal is set for speed. After the optimization for speed, the final design will fit better to FPGA architecture.

5.1.3 Quick Carry Chain

The ripple carry adder is slower than the carry lookahead adder, but its simplicity and greater modularity may compensate for this drawback. Most modern FPGA devices provide dedicated signal paths for carry chains that connect adjacent LEs without using local interconnect paths [20]. In this way, the ripple carry scheme can be directly used to implement the fixed point adder for arbitrary word width. Table 5.5 and Table 5.6 show the performance analysis for fast adders on Altera and Xilinx platforms. The quick carry chain not only reduces the interconnect delay, but also reduces the logic resource utilization.

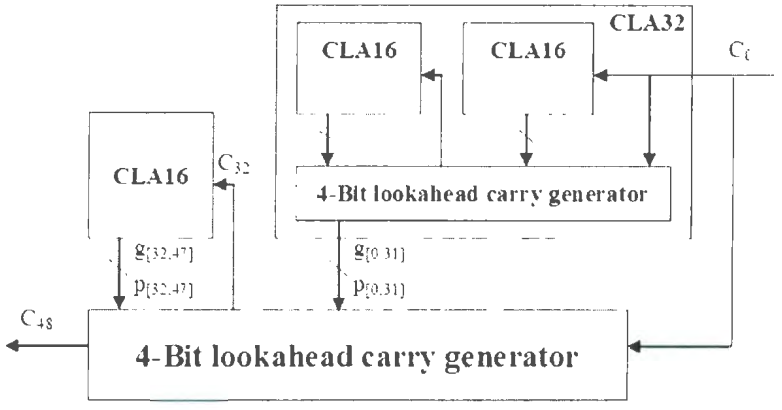

Figure 5.5: Building a 48-bit CLA from 12 4-bit CLAs and 5 lookahead carry generators

Table 5.5: Performance of the Fixed Point Adder with Quick Carry Chain on Altera FPGAs

bits	*delay*	*logic*	*route*	*LEs*	*pins*
1	9.735 ns	4.036 ns (41.46 %)	5.699 ns (58.54 %)	2	5
4	12.235 ns	5.897 ns (48.20 %)	6.338 ns (51.80 %)	9	14
16	15.296 ns	6.661 ns (43.55 %)	8.635 ns (56.45 %)	33	50
28	16.667 ns	7.135 ns (42.81 %)	9.532 ns (57.19 %)	57	86
32	17.774 ns	8.226 ns (46.28 %)	9.548 ns (53.72 %)	65	98
48	20.023 ns	9.709 ns (48.49 %)	10.314 ns (51.51 %)	97	146

Table 5.6: Performance of the Fixed Point Adder with Quick Carry Chain on Xilinx FPGAs

bits	*delay*	*logic*	*route*	*Slices*	*LUT*	*IOB*
1	5.061ns	4.083ns (80.7%)	0.978ns (20.0%)	1	2	7
4	7.570ns	5.022ns (66.3%)	2.548ns (19.3%)	6	11	14
16	8.978ns	7.467ns (83.2%)	1.511ns (16.8%)	17	33	50
28	9.476ns	7.965ns (84.1%)	1.511ns (15.9%)	29	57	86
32	9.642ns	8.131ns (84.3%)	1.511ns (15.7%)	33	65	98
48	10.306ns	8.795ns (85.3%)	1.511ns (14.7%)	49	97	146

Figure 5.6 illustrates the timing performance for the fast adder on Altera and Xilinx FPGA devices. As discussed above, the compact basic logic element helps Xilinx FPGAs to acquire better timing performance than Altera FPGAs.

5.1.4 Carry Save Adder

The carry save adder (CSA) is a row of 1-bit full adders as a mechanism to reduce three numbers to two numbers [9]. As every 1-bit full adder always works in parallel mode, and the critical path delay of the CSAs with different word width is almost close to the delay of the 1-bit full adder. Table 5.7 and Table 5.8 show the performance analysis for CSAs on Altera and Xilinx platforms.

5.1.5 Fixed Point Multiplier

As described in Chapter 2, the high performance CSA tree followed by a fast adder can speedup the multiplier design to achieve logarithmic time multiplication. Table

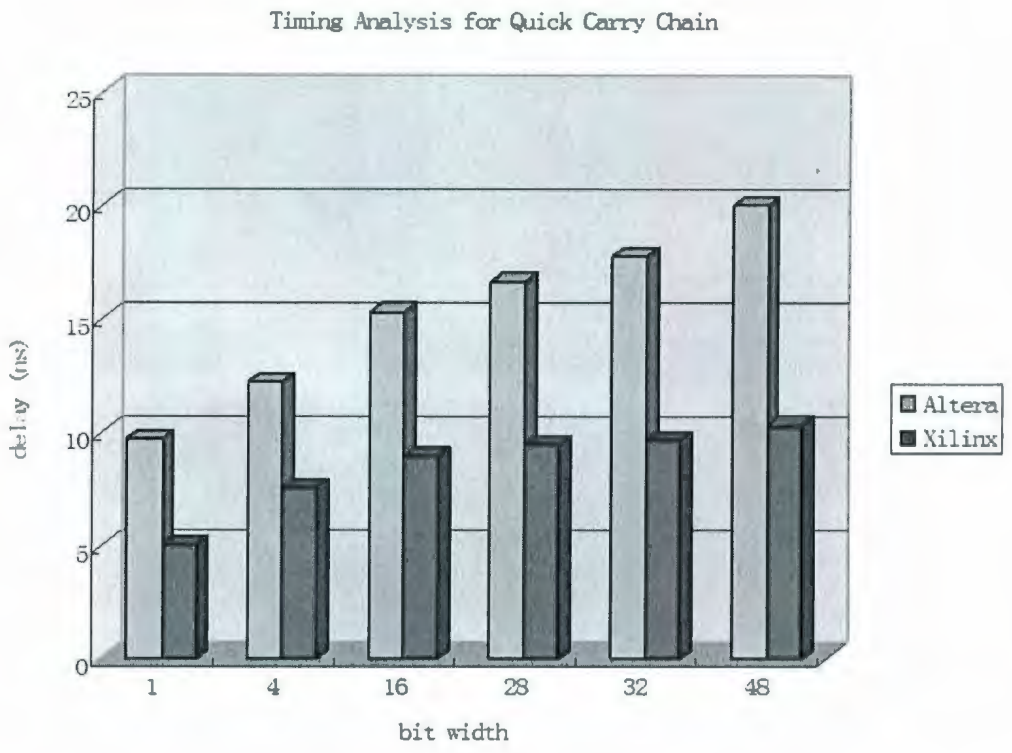

Figure 5.6: Comparison of timing performance for Fast Adder on Altera and Xilinx FPGA devices

Table 5.7: Performance of the Carry Save Adder on Altera FPGAs

bits	*delay*	*logic*	*route*	*LEs*	*pins*
8	10.716 ns	4.395 ns (41.01 %)	6.321 ns (58.99 %)	15	40
24	14.040 ns	4.393 ns (31.29 %)	9.647 ns (68.71 %)	48	121
28	12.537 ns	4.380 ns (34.94 %)	8.157 ns (65.06 %)	55	140
48	13.232 ns	4.249 ns (32.11 %)	8.983 ns (67.89 %)	95	240

Table 5.8: Performance of the Carry Save Adder on Xilinx FPGAs

bits	*delay*	*logic*	*route*	*Slices*	*LUT*	*IOB*
8	5.061ns	4.083ns (80.7%)	0.978ns (19.3%)	9	15	40
24	5.405ns	4.083ns (75.5%)	1.322ns (24.5%)	28	48	121
28	5.061ns	4.083ns (80.7%)	0.978ns (19.3%)	32	55	140
48	5.061ns	4.083ns (80.7%)	0.978ns (19.3%)	55	95	240

5.9 and Table 5.10 show the performance for the fixed point multiplier with different structures on Altera and Xilinx FPGA devices.

The first structure is a 32-bit full-tree multiplier, in which the CSA tree structure is an 1-bit adder array. Although the full-tree structure is not fast enough and occupies a vast logic resource, its partial products reduction tree is a combinational circuit that can be easily sliced into pipeline stages.

The second design not only utilizes the Wallace Tree structure to reduce logic utilization, but also uses Radix-4 Booth's recoding to handle two bits of the multiplier per cycle and reduces the critical path delay. As discussed in Section 2.2, we endeavor use the embedded macro units on FPGAs to reduce the interconnection cost.

The last two designs use the embedded fixed word width multiplier to build various word width multipliers. In the Karastura (b), two fast adders are used to generate the final sum of three partial products. In this way, the delay of the two-step addition is double the delay of the fast adder. To speedup this process, a CSA is introduced in the Karastura (a) to reduce the three partial products to two partial results first, and then a fast adder can generate the final result. The delay consists of one CSA delay and one fast adder delay, and the CSA is faster than any multi-bit fast adder. Therefore, the delay of Karastura (a) is slightly smaller than Karastura (b) in both

Table 5.9: Performance of the Fixed Point Multiplier on Altera FPGAs

	Full Tree	Wallace Tree	Karastura (a)	Karastura (b)
delay	46.150 ns	26.880 ns	23.266 ns	24.913 ns
logic	16.825 ns (36.46%)	10.451 ns (38.88%)	11.307 ns (48.60%)	12.504 ns (50.19%)
route	29.325 ns (63.54%)	16.429 ns (61.12%)	11.959 ns (51.40%)	12.409 ns (49.81%)
LEs	2033	1318	68	60
pins	96	96	96	96
$9 \times 9 Multiplier$	-	-	6	6

Table 5.10: Performance of the Fixed Point Multiplier on Xilinx FPGAs

	Full Tree	Wallace Tree	Karastura (a)	Karastura (b)
delay	28.710ns	13.600ns	11.654ns	12.751ns
logic	13.705ns (47.7%)	8.750ns (64.3%)	9.698ns (83.2%)	10.850ns (85.1%)
route	15.005ns (52.3%)	4.851ns (35.7%)	1.956ns (16.8%)	1.901ns (14.9%)
Slices	635	678	30	30
LUT	1105	1252	56	59
IOB	96	96	96	96
$MULT18 \times 18$	-	-	3	3

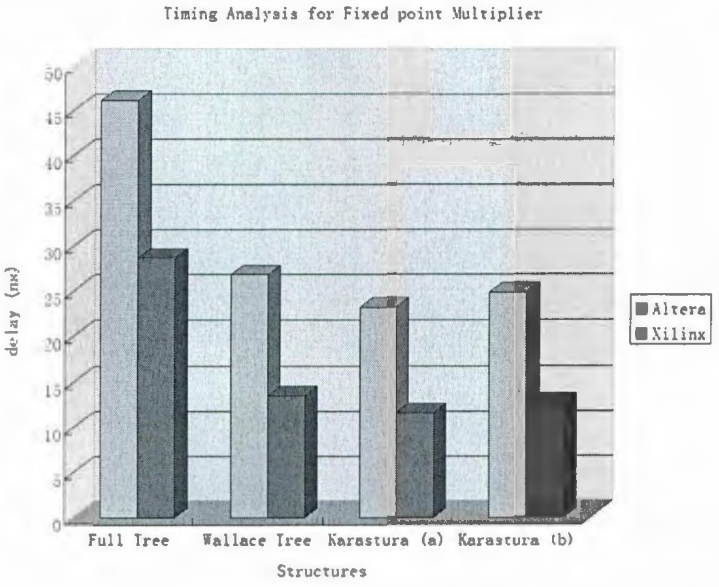

Figure 5.7: Comparison of timing performance for 32-bit multiplier on Altera and Xilinx FPGA devices

Table 5.9 and Table 5.10.

Another advantage of the embedded macro unit is the greatly improved logic utilization. The CSA tree occupies more than one thousand LUTs for implementing a 32-bit multiplier, but the Karastura design only uses less than one hundred LUTs. In this way, more floating point arithmetic units can be mapped onto a single FPGA to improve the calculation capability. Figure 5.7 compares the difference between two FPGA platforms for 32-bit multiplier. The delay of implementations on Xilinx FPGA device are almost half of Altera FPGA implementations. For the 32-bit fixed point multiplier, Xilinx FPGA devices still show timing superiority.

The nicer properties of Xilinx FPGA implementation for arithmetic units indicate that the two LUTs structure of Xilinx FPGA devices is more adapt to the logic intensive design, such as arithmetic units, because the few resource utilization significantly reduces the interconnection cost for the complex logic modules. Contrarily,

the simplified logic element of Altera's FPGA devices not only is more flexible for these digital designs without very complicated logic function, but also has the less power consumption. For instance, many telecommunication applications have very complex FSM modules, which require the high speed for status transfer and without very complex logic operation.

5.2 Performance Analysis for Pipelined Implementations

Based on the timing performance analysis, the combinational designs can be sliced into pipelined stages. Although the number of stages speeds up the pipelined design, the multiple-stage structure also introduces the extra interconnect and synchronization delays. As the delay of the slowest stage in the pipelined design decides the maximum work frequency, the proportional delay for every stage is significantly important for the timing performance of pipelined implementations.

In the pipelined implementations, the output registers are used to synchronize output signals in the same stage, and the inputs should be held constant for specified periods before and after the clock pulse to avoid metastability.

5.2.1 Pipelined Floating Point Adder

To speed up the pipelined floating point adder, the fast adder with the embedded quick carry chain is used for every fixed-point addition. Table 5.11 shows the timing characteristic for the pipelined floating point adders on Altera FPGAs. The maximum frequency of the 8-stage pipelined floating point adder achieves 158.33 MHz. In deeply pipelined implementations, such as the 8-stage floating point adder, some function

Table 5.11: Performance of the Pipelined Floating Point Adder on Altera FPGAs

Stages	Frequency	Delay	LEs	Registers	pins
1	37.95 MHz	26.352 ns	718	99	101
2	73.29 MHz	13.645 ns	745	113	101
4	127.52 MHz	7.842 ns	736	244	101
8	158.33 MHz	6.316 ns	756	461	101

Table 5.12: Performance of the Pipelined Floating Point Adder on Xilinx FPGAs

Stages	Delay	Frequency	Slices	Regs	4-LUTs	IOs	GCLKs
1	19.964ns	50.090MHz	479	107	921	101	1
2	10.309ns	97.004MHz	372	123	686	101	1
4	6.040ns	165.577MHz	402	268	746	101	1
8	5.398ns	185.271MHz	464	499	805	101	1

modules with long critical path delay are decomposed to reduce the delay in one stage. For instance, the 28-bit mantissa addition is divided to two stages with two 14-bit fixed point adders.

Figure 5.8 illustrates the speedup for pipelined implementations on Altera FPGAs. The working frequency rapidly increases from 1-stage design to 4-stage design, and the speedup is close to the ideal theoretical value. However, from 4-stage design to 8-stage design, the relative speedup is significantly decreased, because the extra interconnect delay restricts the timing performance of the deeply pipelined design.

Table 5.12 shows the performance of the pipelined floating point adder on Xilinx FPGAs. The maximum work frequency of the eight stages pipelined floating point

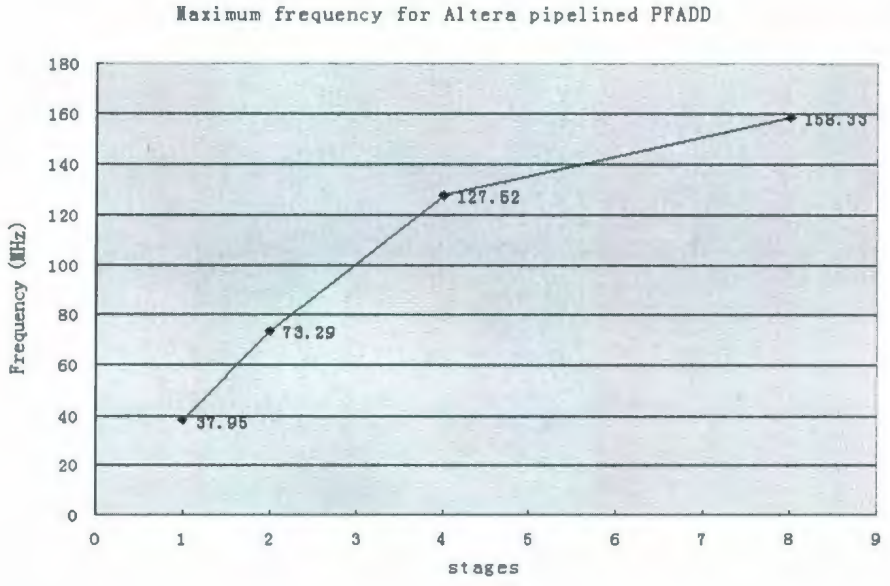

Figure 5.8: Maximum frequency of the Pipelined FADD on Altera FPGAs

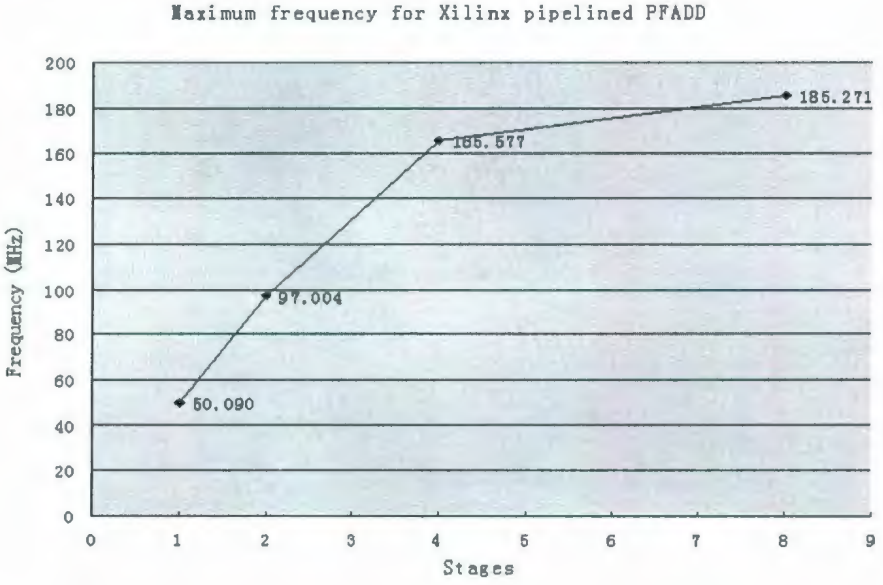

Figure 5.9: Maximum frequency of the Pipelined FADD on Xilinx FPGAs

adder achieves 185.271 MHz. As the Xilinx FPGAs have superior interconnect characteristic, the pipelined floating point adder on Xilinx FPGAs is faster than on Altera FPGAs, and the division scheme for eight stages pipelined design is slightly different between two FPGA platforms. Figure 5.9 illustrates the speedup for pipelined implementations on Xilinx FPGAs. Comparing with the Figure 5.8, the trends of the speedup curve on both Altera and Xilinx FPGAs are similar, but the speed of the pipelined floating point adder on Xilinx FPGAs is higher than on Altera FPGAs.

5.2.2 Pipelined Floating Point Multiplier

In contrast to the floating point adder, the floating point multiplier has a simple dataflow and can easily be sliced into multiple stages. The Wallace Tree structure is more easily sliced into multiple pipeline stages for the deeply pipelined design, but it also occupies vast logic resource and restricts the expansion in the vector architecture. Therefore, the embedded fixed point multipliers are used for mantissa multiplication in the pipelined design. Although the embedded fixed point multiplier can not be decomposed, but its advantage of speed is so distinct that the normalization module and rounding module become the comparatively slow parts.

Table 5.13 shows the performance for the pipelined floating point multiplier. The maximum working frequency of the eight stages pipelined floating point multiplier achieves 155.04 MHz. In the eight stages pipelined design, the 24-bit Karastura multiplier is divided to two stages: three partial products are generated by embedded 18 bit multiplier in the first stage, and a carry save adder and a fast adder are used to generate the final result. Figure 5.10 illustrates the speedup of the pipelined designs. Comparing with the pipelined floating point adder, the speedup curve of the pipelined floating point multiplier keeps an increasing trend.

Table 5.13: Performance of the Pipelined Floating Point Multiplier on Altera FPGAs

Stages	Frequency	Delay	LEs	Registers	MUL9	pins
1	44.98 MHz	22.231 ns	438	98	6	101
2	86.93 MHz	11.503 ns	448	112	6	101
4	120.73 MHz	8.283 ns	438	225	6	101
8	155.04 MHz	6.450 ns	463	389	6	101

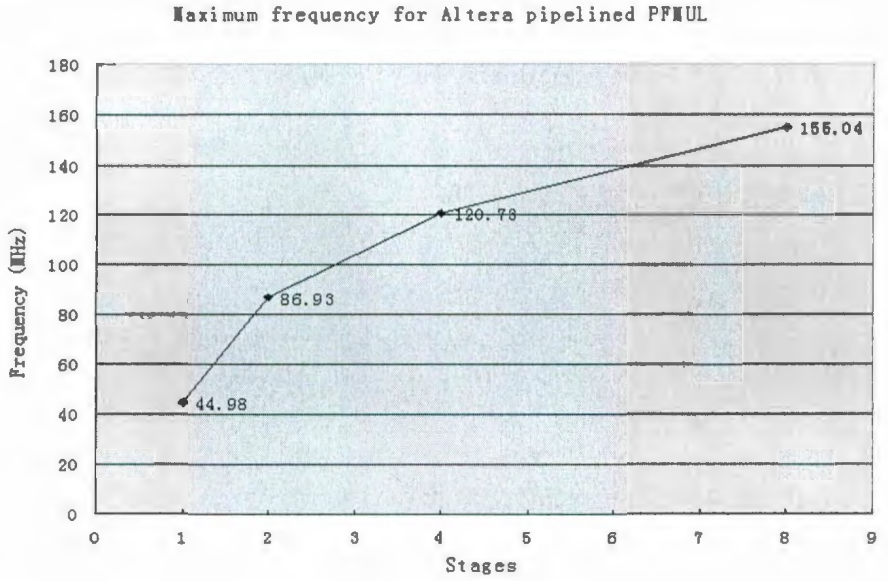

Figure 5.10: Maximum frequency of the Pipelined FMUL on Altera FPGAs

Table 5.14: Performance of the Pipelined Floating Point Multiplier on Xilinx FPGAs

Stages	Delay	Frequency	Slices	Regs	4-LUTs	MUL18	IOs	GCLKs
1	17.814ns	56.136MHz	290	98	557	3	100	1
2	11.332ns	88.249MHz	285	71	546	3	100	1
4	7.373ns	135.639MHz	265	184	492	3	100	1
8	4.883ns	204.813MHz	308	377	558	3	100	1

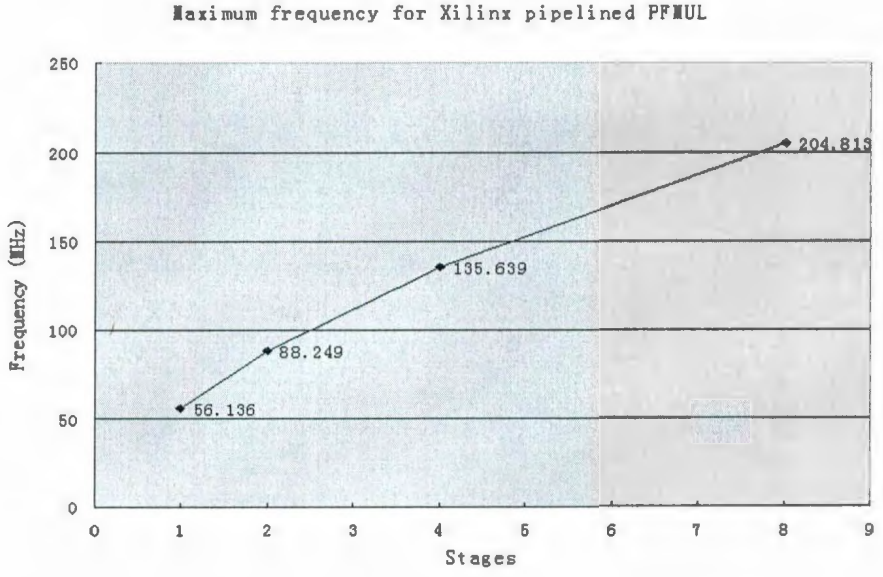

Figure 5.11: Maximum frequency of the Pipelined FMUL on Xilinx FPGAs

As the Xilinx FPGA devices retain the superiority for the interconnect delay, the timing performance of the pipelined floating point multiplier is much better than Altera FPGA devices as shown in Table 5.13. The maximum working frequency of the 8-stage pipelined floating point multiplier achieves 204.813MHz. Similarly, the speedup curve grows almost linearly as shown in Figure 5.11.

5.3 Performance Analysis for VFPU

The Xilinx FPGA chip, Virtex II Pro, is chosen as the target platform for the VFPU implementation to take advantage of the high speed for floating point arithmetic units. This section describes the performance and extensibility for the VFPU implementation with different lane configuration.

5.3.1 Performance Analysis

After synthesis for Xilinx FPGA, Virtex II Pro (xc2vp100-6ff1704), we can obtain the resource utilization and timing performance for the VFPU with different lane configurations. Table 5.15 shows the detailed information. The one lane VFPU only takes 1,316 slices, and the 12-lane VFPU takes 25,994 slices. When the VFPU only includes a few lanes, fewer than or equal to four, the vector register file is directly built on 4-input LUTs. These VFPUs use a part of slices as RAMs, and the maximum working frequency can achieve to 217.014 MHz. The 8-lane and 12-lane VFPUs include a large local storage space as the vector register file, which is implemented by the on-chip block RAMs. In contrast with the 4-input LUT as RAM, the block RAMs are connected to logic slices by interconnection resource, which is much slower than the cascade chain between adjacent slices. Therefore, these VFPUs are slightly slower and can be clocked as fast as 188.768 MHz. Therefore, some more advanced

FPGA chips can further improve the timing performance by directly using the LUT as the vector register file.

Considering each lane of the VFPU can execute two arithmetic operations simultaneously, the 8-lane VFPU can achieve a peak performance rate of 3.020 GFLOPS, and 12-lane VFPU can achieve a peak performance rate of 4.530 GFLOPS.

5.3.2 Extensibility Analysis

While the peak performance rate of the VFPU grows following the number of lanes, the external data bandwidth will also increase rapidly. Table 5.16 shows the bandwidth information for VFPUs with different lane configurations, and lists the possible external RAM system and its peak transfer rate. One of the modern commercial RAM modules, DDR3-1600, can provide up to 12,800 MB/s peak transfer rate at 200 MHz, which can support enough bandwidth for the 16-lane VFPU. Moreover, as we scale up the number of lanes, we not only should consider the usage of IO pins, but also should design the dedicated load/align module to sustain peak throughput.

As discussed in Section 4.4, most vector arithmetic operations are executed on two vector operands, and two loaders can improve the efficiency of the overlap execution process. However, the two vector operands are stored as two sequential arrays in the external memory. In this way, the two vector load instructions should be sequentially issued to load two vectors into the vector register file, and the two loaders have to work simultaneously with two address decoders. The stride and indexed access scheme can share one address decoder for two loaders. In fact, a simple solution is to store two vector operands as one vector of the pair of operands. Figure 5.12 compares two load schemes for vector data. We can easily configure the word width to 64 bits and use the high part as one operand and the low part as another operand. When we generate

Table 5.15: Performance of VFPUs on Xilinx Virtex II Pro

Lanes	1	2	4	8	12	**Available**
Delay (ns)	4.608	4.608	4.608	5.298	5.298	-
Frequency (MHz)	217.014	217.014	217.014	188.768	188.768	-
Slices	1316 (2%)	2816 (6%)	6282 (14%)	14000 (31%)	25994 (58%)	**44096**
Slice Flip Flops	1178 (1%)	3043 (3%)	7384 (8%)	18317 (20%)	35616 (40%)	**88192**
4-LUTs	1792 (2%)	3888 (4%)	8920 (10%)	19198 (21%)	36489 (41%)	**88192**
used as logic	1348	3000	7144	18718	35769	
as Shift registers	60	120	240	480	720	
as RAMs	384	768	1536	-	-	
BRAMs	0 (0%)	0 (0%)	0 (0%)	48 (10%)	72 (16%)	**444**
MULT18X18s	3 (0%)	6 (1%)	12 (3%)	24 (5%)	36 (8%)	**444**
bonded IOBs	100 (9%)	164 (15%)	292 (28%)	548 (52%)	804 (77%)	**1040**
GCLKs	1 (6%)	1 (6%)	1 (6%)	1 (6%)	1 (6%)	**16**

Table 5.16: Bandwidth Analysis for VFPUs

Lanes	1	2	4	8	12
Word-width (bit)	32	64	128	256	384
Frequency (MHz)	217.014	217.014	217.014	188.768	188.768
Bandwidth (MB/s)	868	1,736	3,472	6,040	9,060
Possible external RAM	DDR-200	DDR-266	DDR2-533	DDR2-800	DDR3-1333
Work Frequency (MHz)	100[26]	133[26]	133[27]	200[27]	166[28]
Transfer rate (MB/s)	1,600[26]	2,100[26]	4,266[27]	6,400[27]	10,667[28]

it, we can easily organize the raw data in pair form and determine the data pattern in memory as shown in the low part of Figure 5.12.

5.3.3 Comparisons to Related Work

A prototype of the VFPU was successfully implemented on FPGA platform, and the performance is respectable. A peak performance of 4.530 GFLOPS at 188.768 MHz for the 12-lane VFPU is achieved on the Xilinx Virtex II Pro XC2VP100. If newer and faster FPGAs, such as Xilinx Virtex 5 or Altera Stratix III, as well as memory modules, such as DDR3 or XDR [29], are used, we should be albe to obtain much higher bandwidths.

In large commercial computing systems, advanced ASIC technology is used to achieve high speed and low power consumption. For example, TOSHIBA and SONY implemented a 2.44 GFLOPS at 300MHz floating-point vector processing unit for 3D graphics computing using a 0.18μm 4-metal layer technology [30]. For specific appli-

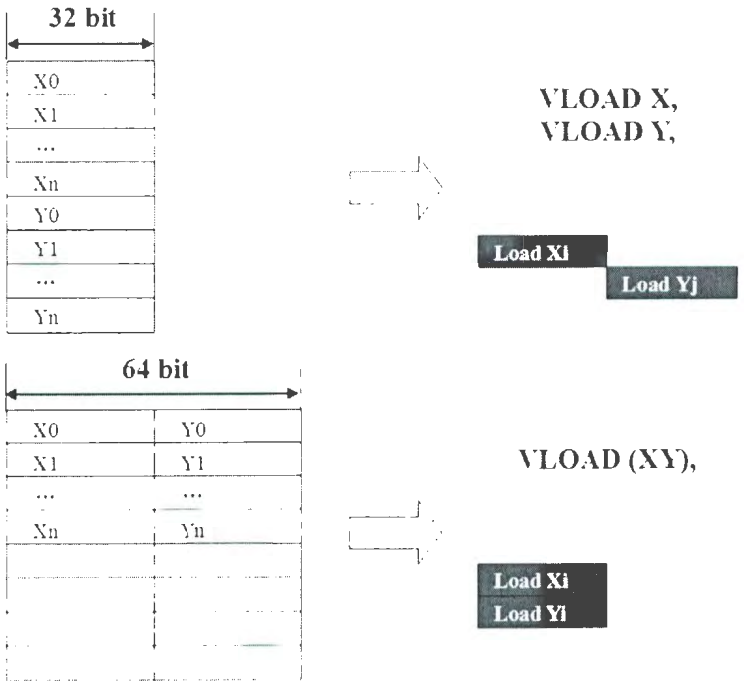

Figure 5.12: Comparison of two load schemes for vector data

cations, such as dynamic ray tracing in image processing, the ASIC implementation has estimated peak performance of 361.6 GFLOPS at 400 MHz [31]. Although the floating point performance is not the main goal for the general purpose processors, they also utilize hyper-thread technology or multi-core architecture to optimize the system performance in the 3D graphics applications. For instance, the Intel Pentium 4 can achieve up to 10.6 GFLOPS for the OpenRT software [31]. The IBM Cell processor includes eight synergistic processor elements for vector computing and have a theoretical peak performance of 256 GFLOPS.

Although the performance of our design is not at the level of ASIC implementations, the recurring design process and fast reconfiguration of FPGAs provide a great flexibility for various applications. Therefore, advances in FPGA capacity and speed can accelerate research in multiprocessor architecture and easily emulate high performance computing units, such as the vector processor, at very low cost [19]. Newer designs are also focused on the vectorization and optimization for specific algorithms. For example, a scalable Sparse Matrix-Vector Multiply implementation on Xilinx Virtex II 6000-4 can run at 1.5 GFLOPS 140 MHz [32].

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The main contributions presented in this thesis are the design, implementation, and performance evaluation of floating point arithmetic units and vector floating point units. The FPGA implementation of VFPUs demonstrates that the FPGA platforms are well suited to the implementation and evaluation for the vector processor architecture.

- Vector Architecture Compared with traditional vector supercomputers, vector processors can significantly reduce the latencies between different function units. The vector register files are implemented on LUTs in the slices or on-chip block RAMs. Every function unit can not only quickly access local storage, but also efficiently exchange data via the chaining scheme.

  Another contribution of this thesis is the good design choice of a vector memory unit that supports two loading ports to access memory data in pair mode. This structure is area-efficient and improves the efficiency of the overlap execution.

- FPGA Implementation Most multi-core microprocessors for high performance computing utilize advanced ASIC technology to design and manufacture the high speed and low power consumption products. However, recent extremely dense FPGAs create an inexpensive, reconfigurable, and highly parallel platform for the extensive codevelopment of hardware and software. The recurring design process not only accelerates prototyping a new architecture in hardware, but also can help to improve the immediate next generation of products by considering the feedback from software engineers [19].

  In additional, many floating point arithmetic units can be mapped onto a single FPGA, which means such a system is less expensive and consumes less power than the general purpose multi-core microprocessors. For example, 24 floating point arithmetic units are mapped onto a single Xilinx Virtex II Pro FPGA device in the design explained in this thesis.

- Performance Analysis The detailed analysis of timing performance for combinational logic design and pipelined design is presented in this thesis. The results demonstrate that an appropriate pipeline division can achieve significant speedups on the floating point arithmetic operations.

  The performance analysis of resource utilization and timing for VFPUs is presented, and the extensibility of VFPUs on FPGA platform is also discussed in this thesis. The measurements show that the Xilinx Virtex II Pro XC2VP100 can support up to a 12-lane VFPU. Future FPGA chips not only can support the VFPU with more lanes, but also can directly build the vector register files on LUTs in the slices to achieve a higher speed. The bandwidth analysis shows that an external memory module with significantly higher transfer rate is also required for the extension of the lanes.

6.2 Future Directions

Several research directions can arise from this work: a primary cache for vector units, an expanded embedded system for specific applications, and a massively parallel computing model.

- Embedded System Design

  The Xilinx embedded processor, PowerPC PPC405, has powerful fixed point performance, and the VFPU can easily work with the PowerPC core to improve the floating point performance. Therefore, a heterogeneous parallel processor architecture can be implemented on the FPGA platform. Figure 6.1 shows the block diagram of an embedded system on the Xilinx Virtex II Pro, XC2VP100. This system includes a custom IP core (VFPU), an embedded processor (PowerPC PPC405), a DDR2 RAM controller, and a primary cache module built on block RAMs. The PLB bus is used to connect these high speed components. The PowerPC core is the master device on this bus, and the VFPU and the DDR2 RAM controller are the slave devices. In this way, the PowerPC core can efficiently issue instructions and control the execution procedure. The Xilinx PowerPC processor environment can execute embedded operating system, such as the embedded Linux, or Real-Time Operating Systems, which will provide a flexible programming environment for different applications. Based on this embedded system, more vectorized applications can take advantage of the VFPU, and many computation-intensive applications can be considered for vectorization to improve the performance.

  In additional, a primary cache can help to exploit temporal locality and reduce memory bandwidth demands, and the dedicated primary cache design can

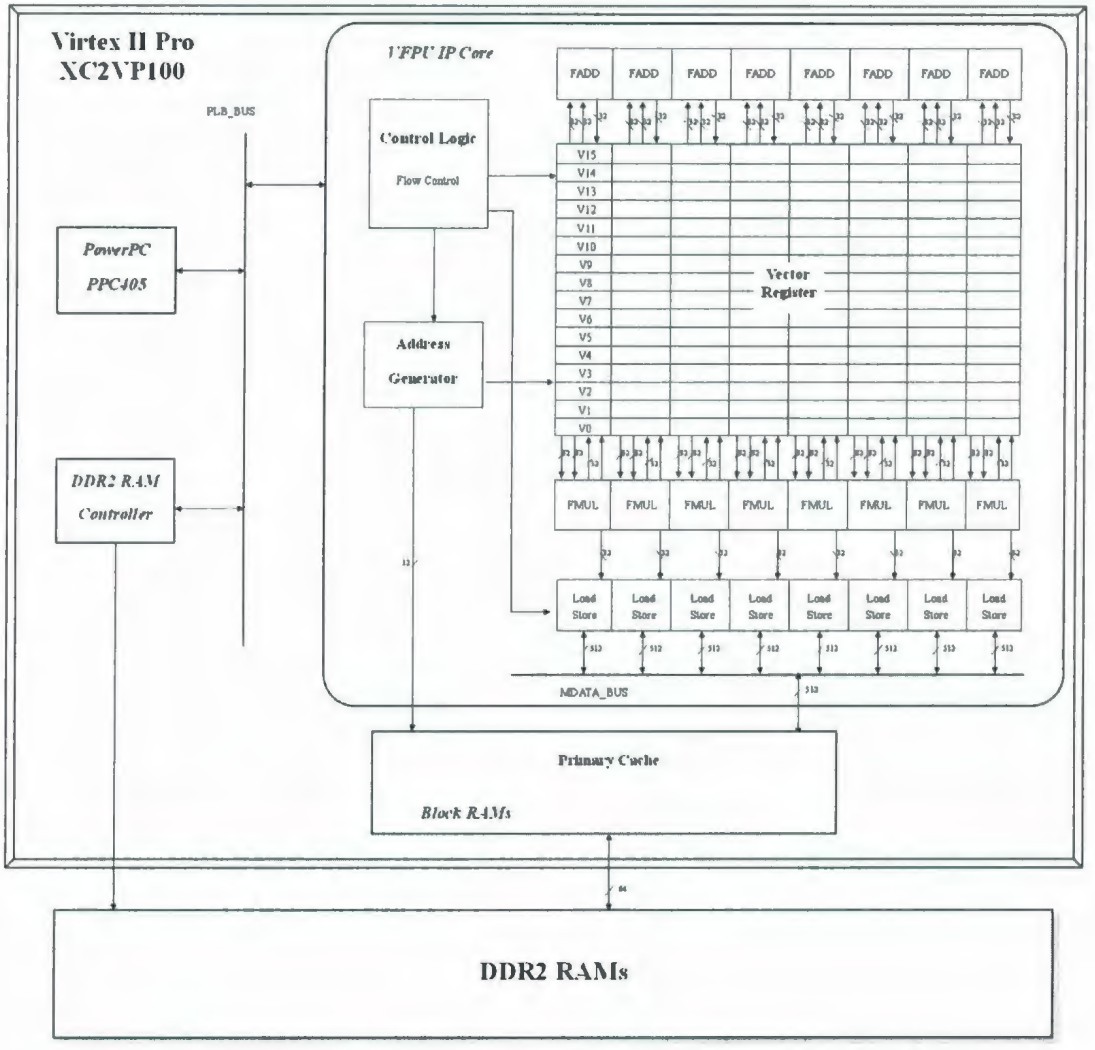

Figure 6.1: Block diagram of an embedded system with VFPU and PowerPC

significantly improve the performance of the embedded system. In particular, the cache refill/access decoupling scheme can eliminate many of the miss states required in traditional vector architectures, and has the potential to achieve better performance with fewer resources than traditional decoupling methods [33].

- Massively Parallel System Massively parallel systems are used to solve large-scale computation problems for many science and engineering applications, such as earth science simulations, weapons research, radio astronomy, protein folding, climate research, cosmology, and drug development. In a massively parallel system, many individual nodes are connected by high-performance interconnect networks and communicate by passing messages. Utilizing the quick interconnection technology, such as HyperTransport [18], multiple FPGA chips with VFPU can be integrated on one board to form a powerful node for the massively parallel system. These boards can then be equipped with Gigabit Ethernet interfaces and exchange data via a Gigabit Ethernet switch.

References

[1] R. Krashinsky, "The vector-thread architecture," *IEEE Micro*, vol. 24, no. 6, pp. 84 90, November/December 2004.

[2] I. S. Board, "IEEE standard for binary floating-point arithmetic," New York, Tech. Rep., 1985.

[3] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Handware/Software Approach*, 2nd ed. United States of America, San Francisco: Morgan Kaufmann, 1999.

[4] D. Geer, "Chip makers turn to multicore processors," *Computer*, pp. 11 13, May 2005.

[5] K. Asanovic, "Vector microprocessors," Ph.D. dissertation, University of California, Berkeley, 1998.

[6] J. F. Wakerly, *Digital Design: Principles and Practices*, 3rd ed. United States of America, New Jersey : Prentice Hall, 2005.

[7] E. Hokenek and R. Montoye, "Leading-Zero Anticipator (LZA) in the ibm risc system floating point execution unit," *IBM Journal of Research and Development*, vol. 33, pp. 71 77, Jan. 1990.

[8] R. Ladner and M. Fischer, "Parallel prefix computation," *Association for Computing Machinery*, vol. 27, pp. 831–838, Oct. 1980.

[9] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 1st ed. New York: Oxford University Press, 2000.

[10] C. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Computers*, vol. 1, pp. 14–17, Feb. 1964.

[11] H. Oh, S. M. Mueller, and C. Jacobi, "A fully-pipelined single-precision floating point unit in the synergistic processor element of a cell processor," *IEEE Journal of Solid-State Circuits*, vol. 41, pp. 759–771, 2006.

[12] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akad. Nauk SSSR*, vol. 14, no. 145, pp. 293–294, 1962.

[13] T. Lang and J. Bruguera, "Floating-point fused multiply-add with reduced latency," *Int. Rep. Univ. Santiago de Compostela (Spain)*, 2002.

[14] CRAY RESEARCH INC., *CRAY-1 Computer System Hardware Reference Manual*, 2240004 ed., 1977.

[15] G. Sohi, "High-bandwidth interleaved memories for vector processors - a simulation study," *IEEE Transactions on Computers*, vol. 42, no. 1, pp. 34–44, Jan. 1993.

[16] C. Kozyrakis, "Scalable processors in the billion-transistor era: IRAM," *IEEE Transactions on Computer*, vol. 30, no. 9, pp. 75–78, Sept. 1997.

[17] R. Russel, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

[18] HyperTransport Consortium Technology Corporation, "HyperTransport. I/O link specification," Tech. Rep., 2005.

[19] J. Wawrzynek, "Ramp: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.

[20] Altera Corporation, *FLEX 10K Embedded Programmable Logic Device Family Data Sheet*, 2003. [Online]. Available: *http://www.altera.com/literature/hb/flex/flex10k.pdf*

[21] Altera Corporation, *Stratix II Device Handbook*, 2007. [Online]. Available: *http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf*

[22] Xilinx Corporation, *Virtex 2.5 V Field Programmable Gate Arrays*, 2001. [Online]. Available: *http://www.xilinx.com/support/documentation/data_sheets/ds0031.pdf*

[23] Xilinx Corporation, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2007. [Online]. Available: *http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf*

[24] IBM Corporation, *128-Bit Processor Local Bus Architecture Specifications*, version 4.7 ed., 2007. [Online]. Available: *http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/file/PlbBusas_01_pub.pdf*

[25] Altera Corporation, *Cyclone II Device Handbook*, 2007. [Online]. Available: *http://www.altera.com/literature/hb/cyclone2/cyclone2_handbook.pdf*

[26] JEDEC Standard, "Double Data Rate (DDR) SDRAM specification," Tech. Rep., 2005. [Online]. Available: *http://www.jedec.org/download/search/JESD79E.pdf*

[27] JEDEC Standard, "DDR2 SDRAM specification," Tech. Rep., 2005. [Online]. Available: *http://www.jedec.org/download/search/JESD792B.pdf*

[28] JEDEC Standard, "DDR3 SDRAM specification," Tech. Rep., 2007. [Online]. Available: *http://www.jedec.org/download/search/JESD793A.pdf*

[29] Elpida Corporation, *512M bits XDR DRAM*, e1033e30 ed., 2007. [Online]. Available: *http://www.elpida.com/pdfs/E1033E30.pdf*

[30] N. Ide, "2.44-GFLOPS GFLOPS floating-point vector-processing unit for high-performance 3D graphics computing," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 7, pp. 1025 1033, Jul. 2000.

[31] S. Woop and E. Brunvand, "Estimating performance of a ray-tracing ASIC design," *Proceedings of IEEE Symposium on Interactive Ray Tracing*, pp. 7 14, Sep. 2006.

[32] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pp. 75 85, 2005.

[33] C. Batten and R. Krashinsky, "Cache refill/access decoupling for vector machines," *Proceedings of the 37th International Symposium on Microarchitecture*, pp. 331 342, Sep. 2004.