ABSTRACT

VECTOR SUPPORT FOR MULTICORE PROCESSORS WITH MAJOR EMPHASIS ON CONFIGURABLE MULTIPROCESSORS

by
Hongyan Yang

It recently became increasingly difficult to build higher speed uniprocessor chips because of performance degradation and high power consumption. The quadratically increasing circuit complexity forbade the exploration of more instruction-level parallelism (ILP). To continue raising the performance, processor designers then focused on thread-level parallelism (TLP) to realize a new architecture design paradigm. Multicore processor design is the result of this trend. It has proven quite capable in performance increase and provides new opportunities in power management and system scalability. But current multicore processors do not provide powerful vector architecture support which could yield significant speedups for array operations while maintaining area/power efficiency.

This dissertation proposes and presents the realization of an FPGA-based prototype of a multicore architecture with a shared vector unit (MCwSV). FPGA stands for Filed-Programmable Gate Array. The idea is that rather than improving only scalar or TLP performance, some hardware budget could be used to realize a vector unit to greatly speedup applications abundant in data-level parallelism (DLP). To be realistic, limited by the parallelism in the application itself and by the compiler's vectorizing abilities, most of the general-purpose programs can only be partially vectorized. Thus, for efficient resource usage, one vector unit should be shared by several scalar processors. This approach could also keep the overall budget within acceptable limits. We suggest that this type of vector-unit sharing be established in future multicore chips.

The design, implementation and evaluation of an MCwSV system with two scalar processors and a shared vector unit are presented for FPGA prototyping. The MicroBlaze processor, which is a commercial IP (Intellectual Property) core from Xilinx, is used as the

scalar processor; in the experiments the vector unit is connected to a pair of MicroBlaze processors through standard bus interfaces. The overall system is organized in a decoupled and multi-banked structure. This organization provides substantial system scalability and better vector performance. For a given area budget, benchmarks from several areas show that the MCwSV system can provide significant performance increase as compared to a multicore system without a vector unit.

However, a MCwSV system with two MicroBlazes and a shared vector unit is not always an optimized system configuration for various applications with different percentages of vectorization. On the other hand, the MCwSV framework was designed for easy scalability to potentially incorporate various numbers of scalar/vector units and various function units. Also, the flexibility inherent to FPGAs can aid the task of matching target applications. These benefits can be taken into account to create optimized MCwSV systems for various applications. So the work eventually focused on building an architecture design framework incorporating performance and resource management for application-specific MCwSV (AS-MCwSV) systems. For embedded system design, resource usage, power consumption and execution latency are three metrics to be used in design tradeoffs. The product of these metrics is used here to choose the MCwSV system with the smallest value.

VECTOR SUPPORT FOR MULTICORE PROCESSORS WITH MAJOR EMPHASIS ON CONFIGURABLE MULTIPROCESSORS

by
Hongyan Yang

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Electrical Engineering

Department of Electrical and Computer Engineering

May 2008

APPROVAL PAGE

VECTOR SUPPORT FOR MULTICORE PROCESSORS WITH MAJOR EMPHASIS ON CONFIGURABLE MULTIPROCESSORS

Hongyan Yang

Dr. Sotirios G. Ziavras, Dissertation Advisor Date
Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Jie Hu, Committee Member Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Roberto Rojas-Cessa, Committee Member Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Alexandros V. Gerbessiotis, Committee Member Date
Associate Professor of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author: Hongyan Yang

Degree: Doctor of Philosophy

Date: May 2008

Undergraduate and Graduate Education:

- Doctor of Philosophy in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2008
- Master of Science in Electrical Engineering,
  Beijing University of Posts and Telecommunications, Beijing, China, 2001
- Bachelor of Science in Electrical Engineering,
  Beijing University of Posts and Telecommunications, Beijing, China, 1998

Major: Electrical Engineering

Publications:

H. Yang, S. G. Ziavras, and J. Hu, "Reconfiguration support for vector operations," *International Journal of High Performance Systems Architecture*, vol. 1, no. 2, pp. 89–97, 2007.

S. Wang, H. Yang, J. Hu, S. G. Ziavras, "Asymmetrically banked value-aware register files for low energy and high performance," *Microprocessors and Microsystems*, 2007.

H. Yang, S. G. Ziavras, and J. Hu, "FPGA-based vector processing for matrix operations," in *International Conference on Information Technology: New Generations*, Las Vegas, USA, April 2007, pp. 989–994.

H. Yang and S. G. Ziavras, "FPGA-based vector processor for algebraic equation solvers," in *IEEE International Systems-On-Chip Conference*, Washington, DC, USA, 2005, pp. 115–116.

H. Yang, S. Wang, S. G. Ziavras, and J. Hu, "Vector processing support for FPGA-oriented high performance applications," in *IEEE Computer Society Annual Symposium on VLSI*, Porto Alegre, Brazil, May 2007, pp. 447–448.

S.Wang, H. Yang, J. Hu, and S. G. Ziavras, “Asymmetrically banked value-aware register files,” in *IEEE Computer Society Annual Symposium on VLSI*, Porto Alegre, Brazil, May 2007, pp. 363–368.

To my parents, Jianyun Zhang and Caifeng Yang

To my husband, Shuangquan Wang

To my daughter, Andrea Wang

ACKNOWLEDGMENT

First of all, I would like to express my deep and sincere gratitude to my advisor: Dr. Sotirios G. Ziavras. I could not have finished this dissertation without his patience, understanding and encouragement during the last five years. His knowledge and expertise in computer architecture has constantly helped and inspired me to go further in my research. I want to thank him for his help in many aspects: his stimulating and constructive suggestions on my research; his patience to review and improve my writings hundreds of times; his kindness for providing financial support during my study and research and much more. I also learned a lot from his professional work style. Those will be of great value in my future career.

I warmly thank Dr. Edwin Hou, Dr. Hu Jie, Dr. Roberto Rojas-Cessa and Dr. Alexandros Gerbessiotis for being my committee members, spending time on reviewing my dissertation and giving me valuable suggestions.

I also thank my lab colleagues in CAPPL for our interesting discussions on various research topics and the fun of being together.

I owe my deepest gratitude to my husband, Dr. Shuangquan Wang, for his understanding and help.

Finally, I am deeply indebted to my father, Jianyun Zhang, my mother, Caifeng Yang, for their unconditional love and continuous support.

TABLE OF CONTENTS

Chapter Page

1 INTRODUCTION 1

1.1 Vector Processing 1

1.1.1 Vector Supercomputers and Their Applications 2

1.1.2 Vector Processing in Embedded Systems 3

1.2 Multicore Processors 5

1.2.1 Motivation for Their Development 5

1.2.2 State of the Art and Future Multicore Processors 5

1.3 Configurable Computing 8

1.3.1 Overview of Field-Programable Gate Arrays (FPGAs) 8

1.3.2 Recent Advances in FPGAs 9

1.3.3 Design Flow 11

1.4 Motivations and Objectives 12

2 VECTOR PROCESSING FOR EMBEDDED APPLICATIONS 15

2.1 Architecture of the Vector Processor 16

2.1.1 Instruction Set Architecture 17

2.1.2 Scalar Unit 18

2.1.3 Vector Register File 19

2.1.4 Vector Memory Interface (VMI) 22

2.2 Performance Results 22

2.2.1 Preliminaries on Vector Processing 23

2.2.2 Sparse Linear Equation Solver 26

2.2.3 Dense Matrix-Matrix Multiplication 30

2.2.4 Sparse Matrix-Vector Multiplication 31

3 MULTICORE PROCESSOR SYSTEM WITH A SHARED VECTOR UNIT . 33

TABLE OF CONTENTS
(Continued)

Chapter Page

3.1 Scalar Processor and FPGA Resources 34

3.1.1 MicroBlaze Soft Processor 34

3.1.2 Virtex-5 Resources 35

3.2 System Design and Organization 36

3.2.1 Vector Issue Logic 36

3.2.2 Vector Register File 38

3.2.3 Function Units 41

3.2.4 Memory Controller 42

3.3 System Implementation on the FPGA Chip 43

3.3.1 Digital Signal Processing (DSP) Blocks 43

3.3.2 Digital Clock Manager (DCM) Primitive 44

3.3.3 On-Chip Block Memories 44

3.3.4 Resource Usage for Various Configurations 45

4 PERFORMANCE RESULTS AND ANALYSIS 47

4.1 RGB to YIQ Conversion (RGB2YIQ) 47

4.2 Demodulation Algorithm in Communication Systems 49

4.3 Discrete Cosine Transform 54

5 POWER CONSUMPTION CHARACTERIZATION AND RESOURCE MANAGEMENT FOR MCWSV SYSTEMS 58

5.1 Resource/Power Modeling for the MCwSV System 59

5.1.1 Dynamic Power Consumption of Floating-Point Arithmetic Units . 60

5.1.2 Dynamic Power Consumption of Dual-Port Block Memories . . . 62

5.1.3 Power Consumption of MicroBlaze w/wo a Floating-Point Unit . . 64

5.1.4 Power Consumption of the Vector Unit for Different Configurations 64

5.2 Power/Resource-Efficient MCwSV Systems for Various Applications . . . 67

5.2.1 Scalar/Vector Execution Time Estimation 71

TABLE OF CONTENTS
(Continued)

Chapter Page

5.2.2 Application-Specific MCwSV (AS-MCwSV) System for Power/Resource Efficiency 74

6 CONCLUSIONS AND FUTURE WORK 81

6.1 Conclusions 81

6.2 Future Work 83

REFERENCES 84

LIST OF TABLES

Table Page

2.1 Non-Zero Element (NNZ) Changes in W-Matrix Partition Generation 28

2.2 Input Sparse Matrices from the Harwell-Boeing Collection in Our Experiments 31

3.1 Resource Consumption of Single-Precision Implementations 43

3.2 Configuration and Usage of Block Memories 45

3.3 Resource Usage of the Vector Unit and the Memory Controller 46

4.1 Execution Times (us) for RGB2YIQ Conversion 48

4.2 Execution Times (ms) of the Distance Calculation Algorithm when Run on One MicroBlaze with the Vector Unit for 1000 Input Data. 51

4.3 Execution Times (ms) of the Demodulation Algorithm (MB: MicroBlaze, DIS:Distance, VU:Vector Unit.) 53

4.4 Vector and Scalar Execution Times and Their Ratio for the DCT Algorithm . 55

5.1 Resource Usage and Dynamic Power Consumption of IEEE Single-Precision FPUs on the Virtex-5 FPGA 62

5.2 Resource Usage and Power Consumption of a MicroBlaze w/wo FPU Support 64

5.3 Resource Usage and Power Consumption of the Vector Unit Without Enable Control of Block Memories 65

5.4 Dynamic Power Consumption of the BRAMs and the VRF 66

5.5 Resource Usage and Power Consumption of the Vector Unit with Enable Control of the Block Memories 67

5.6 Estimated Execution Time for the Demodulation Algorithm 78

5.7 Execution Time and Resource/Power Consumption for the Demodulation Algorithm 79

LIST OF FIGURES

Figure Page

1.1 FPGA design flow. 11

2.1 Overview of the system. 16

2.2 Block diagram of the vector processor (VLR: Vector Length Register, VMR: Vector Mask Register). 17

2.3 Instruction formats. 18

2.4 Scalar processor architecture. 18

2.5 Resource and power consumption for single-block implementation of a vector register file containing 8 vector registers of 32 32-bit elements. 20

2.6 Vector register file organization and connections with other components. . . . 20

2.7 Recurrence problem. 24

2.8 Performance of sparse linear equation solver. 29

2.9 Performance of dense matrix-matrix multiplication. 30

2.10 Performance of sparse matrix-vector multiplication. 32

3.1 MicroBlaze core block diagram [1]. 34

3.2 MCwSV multicore processor system with a shared vector unit (The scalar registers are not shown in the vector unit). 37

3.3 Vector register renaming table (R: Reserved; V: Valid). 38

3.4 The format of incoming and outgoing instructions in VIL. "MB ID" indicates which MicroBlaze issued the operation; "FU ID" indicates which function unit should this instruction go to. (OP: Opcode, R: Reserved.) 38

3.5 Multiport vector register implementation. 40

3.6 Vector chaining. 41

3.7 Vector chaining in a multi-lane structure (LSU: load/store unit). 42

4.1 RGB2YIQ speedup of using the vector unit as compared to one MicroBlaze processor without this unit. 49

4.2 RGB2YIQ normalized speedup (using resource usage) of the vector unit as compared to one MicroBlaze processor. 50

LIST OF FIGURES
(Continued)

Figure Page

4.3 A typical digital communication system. 50

4.4 The speedup for distance calculation of the vector unit over one MicroBlaze. . 52

4.5 Speedup normalized by the slice usage ratio for distance calculation. 52

4.6 Speedup of including a vector unit in a MicroBlaze processor for the DCT algorithm. 56

4.7 Speedup of two MicroBlazes versus one MicroBlaze, both systems with a vector unit, as a function of the ratio of vector and scalar execution times. . . . 57

5.1 Dynamic power consumption of one memory block VS the enable rate. . . . 63

5.2 Dynamic power consumption of the dual-port memory VS the toggle rate. . . 63

5.3 Dynamic power consumption of a MicroBlaze w/wo an FPU at different toggle rates. 65

5.4 Percentage of power consumption and slice usage for the vector unit with the 1-bank configuration. 67

5.5 Overview of the procedure for architecture template creation. 69

5.6 Estimation model for the vector execution time. 72

5.7 Instruction profiling for the RGB2YIQ algorithm on the vector unit. 73

5.8 Error percentage of the vector time estimation model for the RGB2YIQ algorithm. 73

5.9 Execution time and power estimation for the MCwSV system for various data sizes and different bank configurations for the RGB2YIQ algorithm. 76

5.10 Normalized design metric (M) for various data sizes and different bank configurations. 76

5.11 Normalized design metrics. 80

CHAPTER 1

INTRODUCTION

1.1 Vector Processing

During the past thirty years processor designers have mainly concentrated on exploring instruction-level parallelism (ILP) to improve performance; only recently have they applied thread-level parallelism (TLP) approaches in chip design. The former approach has lead to the development of superscalar and VLIW (Very Long Instruction Word) architectures [2]. Superscalar architectures are flexible, can exploit ILP dynamically by hardware and have dominated CPU design for a long time. This design paradigm, however, requires a complex control unit with complexity growing exponentially; this eventually diminishes the return of issuing more instructions [3]. Also, the majority of applications have a limited degree of ILP. The pure VLIW paradigm relies on the compiler to exploit ILP and issues each time a fixed number of instructions fetched as one large instruction. It has simpler hardware as compared to superscalars, but requires large instruction cache space, increased instruction fetch bandwidth and very sophisticated compilers.

Vector processing avoids the above problems of superscalar and VLIW architectures for applications rich in data parallelism [2]. It was very common for supercomputers built in the 1980s and 1990s; then it began to fade away but some relevant features were realized in advanced microprocessors for operations on streams of data. The release of the Earth Simulator in 2002, the fastest supercomputer in the world for three consecutive years [4], led to a resurgence of interest in the vector architecture [2]. Vector processors have made a comeback in several domains, including scientific computing [5, 6], multimedia processing [7–9], telecommunications and signal processing [10, 11]. Abundant data-level parallelism (DLP) in these domains allows vector processors to provide higher peak and sustained performance than superscalar, VLIW and chip multiprocessor (CMP) designs [9, 12]. IBM, Toshiba and Sony announced the Cell processor in 2000 which consists in part of many vector processors and is optimized for compute-intensive workloads, broadband data transmission and multimedia processing [13]. Today almost all commodity CPU designs include some vector processing instructions, typically known as of the Single Instruction, Multiple Data (SIMD) type. Modern video game consoles and consumer computer-graphics hardware in particular rely heavily on vector processors.

The properties of the vector architecture, which render it the most efficient and lowest-complexity approach for array operations are [2]:

- A single vector instruction implies lots of essential work; it is equivalent to an entire loop implemented in sequential processing. So the instruction fetch and decode bandwidths needed to feed multiple, deeply pipelined functional units are dramatically reduced.

- Each element in the resulting vector is independent of the current computation of other results in the same vector. Thus, multiple operations can be executed in parallel on vector elements and the effect of data hazards is minimized. A multi-lane memory-access structure which explores data parallelism is described in Subsection 1.1.2; this structure is power efficient, simple to design, and easy to scale [14].

- Vector instructions access the memory with known patterns. Thus, the cost of the latency to access the main memory is amortized because a single access is initiated for a large part or the whole of a vector.

1.1.1 Vector Supercomputers and Their Applications

Vector processing is particularly useful for large science and engineering applications [2]. Actually, the early supercomputing arena was dominated by vector supercomputers. In 1993, 340 out of the top 500 supercomputers were based on the vector structure [4]. Cray, NEC, Fujitsu, Hitachi, Convex CDC, EDT and IBM all developed high performance vector supercomputers [2]. Then, the fast development of commodity clusters began to take a leading role in the supercomputing field, benefitting from the performance improvement of microprocessors [15]; vector supercomputers appeared to fade. For a decade most of the computer architects believed that the future belongs to Massively Parallel Processing (MPP) systems made from off-the-shelf commodity chips. Some persons even predicted that the vector structure will be of diminished value in the supercomputing area and will be totally replaced by systems built form large numbers of superscalar microprocessors [2]. But in 2002, NEC Inc. announced the world's fastest supercomputer, the Earth Simulator based on 5120 vector microprocessors. The impact of the Earth Simulator, together with the release of a new generation of vector machines from Cray, led to a renaissance of interest in the vector processing structure [2]. Vector processors still remain the most effective way to exploit data-parallel applications [9]. Due to the high communication latency of clusters, vector supercomputers have been widely used and will continue to play an essential role in many application areas in government, commercial and academic use [15]. These areas include stockpile stewardship, signal intelligence defense, climate prediction, plasma physics and etc. [15].

1.1.2 Vector Processing in Embedded Systems

Embedded systems are ubiquitous and can be found in most of the electronic products related to consumer electronics, office automation, home appliances and automobiles. In recent years, embedded computing systems have grown tremendously not only in their popularity, but also in their complexity [16]. Embedded systems are typically single-functioned and have tight constraints on implementation. In particular, they must often react to the external events in real time, and meet tight the requirements of size, weight, budget power and cooling consumption [17].

The vector architecture is especially suitable for embedded systems targeting array-based applications because of simplicity, good performance and small code size. The simple architecture implies small area, low power consumption, low cost and short design time. It can outperform the two architectures widely used in desktop domain, superscalar and VLIW by a factor of two to ten on array-intensive applications despite its much lower clock frequency [9].

Two previous vector systems were built at U.C. Berkeley for embedded applications: one is T0, a vector microprocessor [14]; the other is a scalable vector media-processor [18]. T0 divides the central vector register file into multiple lanes where each lane contains a slice of vector registers. Each lane also contains a datapath to a function unit, so data parallelism can be easily employed because there is no data dependency between element calculations [14]. The scalable vector media-processor not only uses the multi-lane structure, but also a cluster organization which divides the set of vector registers into different clusters, where each cluster contains part of the set. A communication network is used to transfer the data in vector registers between different clusters, when needed. Different function units are connected with the clusters to achieve ILP. One problem in conventional vector processor design is that the complexity and size of the centralized multi-port vector register file limits the number of function units [19]. Both of these architectures can solve the problem and the latter can also hide long memory latencies [14, 18] .

Multimedia and telecommunications benchmarks from the Embedded Microprocessors Benchmark Consortium (EEMBC) [20] were run on the latter and results showed that the vector processor is two times faster than a 4-way superscalar processor running at five times higher clock frequency; it is also ten times faster than a 5-way to 8-way VLIW design. It also showed that the vector microprocessor has smaller code size and less power consumption as compared to RISC, VLIW and superscalar architectures [9].

1.2 Multicore Processors

1.2.1 Motivation for Their Development

In the 1990s, computer performance was increasing according to Moore's law, driven largely by increasing transistor density, accelerating operating frequency and exploring more ILP. But then it couldn't keep up. The performance increase was 60 percent per year in the 1990s but slowed to 40 percent from 2000 to 2004; then, it reached only 20 percent after 2004 [21]. Today, it costs twice the chip area to gain a 20 percent speed increase with a uniprocessor. The reason lies in two facts. One is that heat management is becoming a very serious problem with transistor size decreases. Leakage power constrains frequency acceleration [22]. The other is that both the superscalar and VLIW, two widely used architectures in modern computers, approached performance limits of exploring ILP because of quadratic increases in power, area, hardware control or compiler complexity [3]. The diminishing returns of building more complex uniprocessors forced researchers to change their design philosophy toward improving performance and managing power consumption.

As a result, chip processor makers have turned into multicore processor designs. It can potentially explore TLP to provide increased single-chip computational capability without requiring a complex microarchitecture. Multiple cores having the same microarchitecture are integrated. Generally these cores have a shared L2 cache and each core is equipped with its own cooling support. They do not necessarily run as fast as single-core models, but they can improve the overall performance by handling more work in parallel [21]. Thus simple multicore processors can provide better performance per watt for a given area than more complex single-core competitors [23]. The simplicity of their architecture also offers new opportunities for scalability and thermal control.

1.2.2 State of the Art and Future Multicore Processors

Processor makers like Intel, AMD, IBM and Sun have introduced multicore chips for servers, desktops and laptops [21]. Intel has shipped Core Duo, Core 2 Duo, and Xeon (x1xx series) microprocessors with dual-core technology; it also provided in 2006 quad-core versions of the Core 2 chip, called the Core 2 Quad and Core 2 Extreme [24]. Also, it has developed an 80-core processor prototype that can perform a trillion floating-point operations per second at 3.16GHz and this processor will be rolled out within the next five years. Moreover, Intel has more than 15 multicore processor projects underway, and the architecture is expected to hold dozens or even hundreds of processor cores in a single die [22]. AMD also has released its dual-core Opteron server/workstation processors, the Athlon 64 X2 family desktop and the Turion 64 X2 for laptops in 2005, and a quad-core Opteron processor in 2007 [25]. IBM has announced its dual-core chips POWER4 and POWER5. Sun Microsystems introduced its dual-core chip UltraSPARC IV early in 2003 and unveiled the UltraSPARC T1 in 2005, Sun's first microprocessor that is both multicore and multithreaded. The UltraSPARC T1 is available with four, six or eight CPU cores, and each core is able to handle four threads concurrently. Thus, the processor is capable of processing up to 32 threads concurrently [26]. It is widely believed by computer architects that future microprocessors will perhaps include hundreds or even thousands of cores in a single die [27].

Inspired by the fast development of multicore designs, accelerators prototyped on FPGA boards for multicore processors are under development by groups of researchers in universities and companies. They aim to build research platforms for multcore/multiple processor designs with up to thousands of cores in hardware/software IPs, in order to provide highly accurate performance prediction, parameterized reconfiguration and extensive monitoring. Such accelerators are supposed to provide multiple orders of magnitude speedup over software-based simulators of multicore processor designs [27].

While most of the current multicore processors only include homogeneous general processing cores, it could be promising to include heterogenous cores in the multicore design [22, 23, 28]. Frameworks for heterogenous multicore architectures were studied in [28, 29]. They tried to provide platforms for integrating different cores flexibly, while offering the necessary software support for application scheduling and a simulation environment for multicore designs. A performance-asymmetric multicore design, where individual cores have different compute capabilities, was also described in [23]. While a few high-performance complex cores can provide good serial performance required by single threads, many low-performance simple cores could provide higher performance. It was demonstrated that some degree of asymmetric performance is beneficial to particular applications. Multicore designs mixed with multithreading and hyperthreading participants, such as the UltraSPARC announced by Sun, could increase computer performance even further [26]. Higher performance can also be expected from integrating special-purpose processor cores in multimedia processing, speech recognition, networking applications, etc. [22].

On the other hand, substantial effort has been made to improve software support for parallel programming on multicore processor platforms. Intel provides a number of compilers, software primitives and tools, aiming at enabling parallel programming on its processors. AMD, on the other hand, has announced plans to extend the X86 instruction set to ease the job of programming multicore processors.

Multicore processors also provide new opportunities in thermal management and scalability because of multithreading/multitasking. Many new innovations designed to optimize power, performance, inter connectivity and scalability have been implemented for multicore processors [23, 30, 31].

Multicore processors have been used to manage web applications (Sun Microsystems/UltraSparc T1 [32]), game consoles (IBM, Sony, Toshiba/Cell [33]), data storage networks (Cavium Networks/Octeon), image processing (Rapport/KC256), wireless communications (PicoChip/PC2xx), etc.

1.3 Configurable Computing

Computation-intensive applications are a great challenge to conventional instruction set processors due to the latter's underlying sequential architecture. ASIC designs with abundant calculation units are very efficient for such tasks. But they are resilient to potential modifications required to fit new applications. This makes the ASIC approach prohibitively expensive for small productions and drives designers to search for flexible solutions. Configurable or reconfigurable computing provides a way to combine the flexibility of microprocessors and the implementation efficiency of ASICs. It has blurred the distinction between hardware and software, and benefits from the advantages of both [34]. Configurable computing has demonstrated its superiority in many application domains and many configurable computer systems are based on FPGAs. In this section, we will introduce the FPGA technology with recent developments that make it feasible to form computation elements.

1.3.1 Overview of Field-Programable Gate Arrays (FPGAs)

After their invention by the Xilinx Lab in 1984, the FPGA technology has developed rapidly in both capacity and performance. Take the example of the Virtex-5 series, a new 65nm FPGA announced by Xilinx Inc. It offers up to $51,840$ slices, with each slice containing four 6-input look-up tables (LUTs), four storage elements, wide-function multiplexers and carry logic, 1200 user I/Os, 10Mbits of 36Kbit block RAM, and 3.4Mbits of distributed RAM, plus an abundance of hardened intellectual property (IP) blocks. It provides 30% higher speed, 35% lower dynamic power and 45% less area than the previous generation Xilinx FPGA Virtex-4. Additionally, FPGA providers continue to decrease their price. The cost of logic cells has been reduced 30-fold from their introduction, to as little as less than 50 cents per 1000 logic cells. Although FPGAs have historically been used as "glue logic" to connect various other elements within a system or to replace an ASIC phototype, they are now widely used in several diverse domains because of their advances in capacity and performance provided mainly by the last couple of FPGA generations. The recently improved computational power of FPGAs has also attracted mainstream high-performance computing companies. Cray's XD1 supercomputer combines AMD Opteron processors with FPGAs to accelerate computations. This configurable supercomputer can yield orders-of-magnitude speedups. Silicon Graphics also includes FPGAs in its Altix platform [35].

As FPGAs have matured, researchers were quick to recognize their potential and they soon became the most common devices in configurable computing. Many reconfigurable systems are based on FPGA technology. They can be classified in fine grained and coarse grained architectures according to the complexity of the lowest level design; or closely coupled and loosely coupled architectures according to the proximity to the host machine [34]. These systems include Splash [36], BEE2 [37], ArMen [38], TM and TM-2 [39], BORG and BORG II, SPYDER, RENCO, etc. Performance increase of orders of magnitude has been reported in various application areas such as multimedia, wireless communications and signal processing [34].

1.3.2 Recent Advances in FPGAs

Most of the recent generation FPGAs are called platform FPGAs as compared to previous products which were based on fine-grain reconfigurable architectures. The term "platform FPGA" represents a coarse-grain architecture with the integration of a wide variety of hard and soft IP cores on a single device. The programmability of this coarse-grain architecture greatly reduces system development time and complexity as compared to fine-grain FPGA architectures. Platform FPGAs generally provide plenty of DSP blocks, large on-chip memories, and a lot of IP cores, such as high-speed transceiver links, sophisticated memory controllers and network interfaces. Take the example of the newest Xilinx FPGA Virtex-5 series: it provides a built-in PCI express endpoint block which supports a ubiquitous serial connectivity standard, several built-in Ethernet MAC blocks and 1.25Gbps selectIO in some models. It also provides up to 640 DSP48E slices with each of the enhanced slices containing a 25 × 18 multiplier and a 48-bit adder, and greatly improves the signal processing ability. These DSP blocks can enable single-precision floating-point math, which was an impossible task for earlier FPGAs, and wide filters.

Another important feature provided by platform FPGAs is the embedded microprocessor. The FPGA processor can be of the "hard" core or "soft" core type. Hard processors are microprocessors that have been diffused into the silicon of the FPGA, similar to discrete processors. However, in the FPGA the CPU is surrounded by programmable logic which can be configured to perform other functions and can be coupled tightly with the CPU [40]. The IBM PowerPC is a hard processor which can be found in some Xilinx devices such as the Virtex-II PRO, Spartan-3 and Virtex-4 series chips. AVR is another hard processor used in the Atmel FPSLIC device in combination with Atmel's programmable logic architecture.

Soft processors are microprocessors that are implemented by the user using the logic primitives of the FPGA. They provide necessary elements and leave control to the user to choose optional features, which may include caches, various arithmetic units, FPUs, debug logic and other function modules for particular application requirements. The MicroBlaze is a 32-bit soft processor and PicoBlaze is a simpler alternative with an 8-bit data bus made available from Xilinx. Nios and NiosII are 32-bit soft processors provided by Altera. Other soft cores include the open source LatticeMico32 and LatticeMico8, as well as third-party (either commercial or free) processor cores.

The key benefits of using a soft processor include configurability to trade off price and performance, faster time to market, easy integration with the FPGA fabric, and no processor obsolescence [1]. It is easy to create a custom processor with the exact mix of peripherals, memory interfaces, and hardware accelerators for a given application. The number of soft processors that can be instantiated in a device is only limited by the FPGA's logic resources. The number of hard processors, conversely, is fixed in a particular FPGA. While soft IP cores provide flexibility to the user, the optimized design techniques for hard IP blocks ensure low area/power consumption and substantially better performance as compared to typical un-optimized designs.

Their coarse-grained architectures and the embedded microprocessors have made FPGAs feasible choices in the high-performance computing area. FPGA-based SIMD/MIMD systems [41–44], vector microprocessors [45–48] and other reconfigurable systems [34] have reported impressive performance improvement for applications from various domains.

1.3.3 Design Flow

Fig. 1.1 shows the traditional FPGA design flow. Our design follows this conventional design flow and ModelSim is used for gate-level simulation, Synplify pro and Xilinx XST are used for synthesis, and Xilinx place-and-route tools are used to put the design on FPGA chips. Other development tools, such as Viva of Starbridge Systems [49] and Corefire of Annapolis Micro Systems[50] aim to provide alternative programming styles for FPGA designs targeting specific systems. They provide user friendly graphic interfaces and incorporate a bunch of high performance IP cores. The user can just use a "drag and drop approach." Their benefit is short design time and development that does not descend

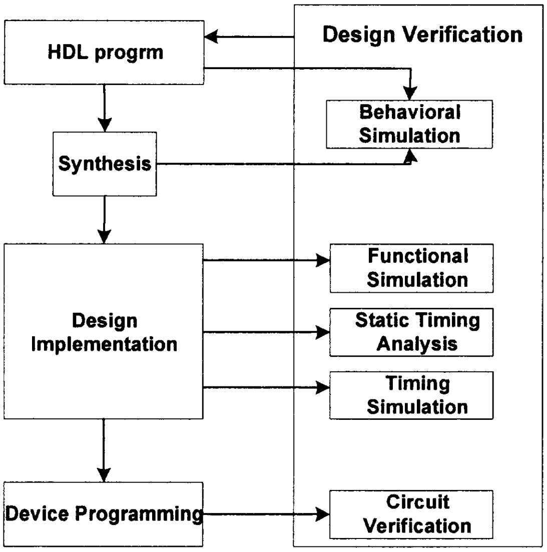

Figure 1.1: FPGA design flow.

into low-level hardware details. But on the other hand, they limit the designer's control and, therefore, the hardware implementation can be hardly fully optimized. Our experiments show that a processor written in VHDL and running at 100 MHz on the Annapolis WildStar-II board can only run at 60 MHz when designed by Corefire.

1.4 Motivations and Objectives

It is becoming increasingly difficult to build higher performance uniprocessor chips because of the complexities in manufacturing and power management. On the other hand, the quadratically increasing circuit complexity forbids the exploration of more ILP, which is anyway rather limited in the majority of programs. To continue raising the performance, silicon designers then focused on TLP to realize next generation processors. Multicore processor design is the result of this trend. It can potentially explore TLP to provide increased single-chip computational capability without requiring a complex microarchitecture. It has thus been deemed to be the most important trend in computer architecture and has been produced for servers, desktops and laptops by major processor chip makers like Intel, AMD, Sun and IBM.

On the other hand, in the embedded system market developers have been working with multicore devices for years. For example, the Apple iPOD features dual 90MHz ARM7TDMI processors; engineers at NEC have developed a processor for cellular telephones that includes three ARM processor cores in a single chip, aimed to improve the multimedia capabilities of cell phones. Texas Instruments Inc. (TI) will introduce two multicore DSPs for wireless infrastructure. ARM, a leader company in embedded system design, developed its multicore embedded processor ARM11 MPCore, which can be configured to contain one to four processors [51].

Although multicore processor designs can boost system performance by exploring both TLP and ILP, and can solve the power management problem, their ability to utilize DLP is still limited. Since vector processing is the most efficient way to explore DLP and vector operations are rather common in embedded applications, including a vector processing unit in the embedded multicore processor can increase the system performance in many cases. Although current general-purpose processors normally support DLP, the allocated hardware resources are quite limited [9, 52]. We propose that some hardware budget in multicore processor systems be used to realize a shared vector unit to greatly speedup computations for applications abundant in DLP. As the number of transistors doubles in about 18 months (Moore's law), the need for shared vector units in multicore designs will become even more important. Thus, our thesis is that rather than improving only scalar or TLP performance, we could spend some of the hardware budget to realize a vector unit which will greatly speedup computations for applications abundant in DLP. To be realistic, limited by the application itself and the compiler's vectorizing capabilities, most of the general-purpose programs can only be partially vectorized. For efficient resource usage, one vector unit should be shared by several scalar processors. The specific objectives of this dissertation are as follows:

- Design and implement a vector microprocessor which can provide significant speedups on vector-oriented, computation-intensive algorithms, with low energy consumption.
- Build an FPGA-based prototype for a multicore processor system with a shared vector unit, which should be able to deliver higher performance as compared to multiple scalar processors under the same area budget for some full or partially vactorizable applications.
- The designed system should support high-level languages and compilers.
- The system should support modular hardware implementation.
- Benchmark the prototype for embedded applications.

Thus, the target is a vector processor design framework suitable for embedded systems in multicore processor environments with a shared vector unit (MCwSV). The vector unit should support massive floating-point calculations, generally a time-consuming component in scientific computing. The design should be easily configured with respect to the kinds/number of function units in order to match various application requirements. MCwSV should be able to take advantage of mature programming languages and compiler technologies by using processor IPs, such as MicroBlaze, which is a commercial 32-bit RISC processor. The vector unit should be connected to the scalar processors with standard bus interfaces to provide ease of scalability and code portability.

Considering the inherent flexibility of FPGAs and due their (re)configuration capability, an MCwSV architecture template should be built based on tradeoffs involving resource consumption and performance.

CHAPTER 2

VECTOR PROCESSING FOR EMBEDDED APPLICATIONS

As stated before, a vector microprocessor has much simpler hardware architecture and lower energy consumption than VLIW and superscalar processors. And it is probably the most efficient way to deal with applications that have abundant data parallelism and little data reuse. Also, as shown in [18], the degree of vectorization can exceed 90% for most benchmarks in the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [20]. All of these make vector processing a very competitive choice for embedded systems.

In this chapter, a vector processing system with two identical vector processors is proposed and implemented on the Annapolis WildStar II board [53]. Each vector microprocessor has abundant parallel calculation units and supports floating-point calculations. Specialized hardware and respective user instructions for efficient sparse matrix operations were implemented as well. To test its performance, dense matrix-matrix multiplication, sparse matrix-vector multiplication and a sparse linear equation solver, with input matrices from various application domains, are run on the system [45–48]. Our comparison with a commercial PC demonstrates that our implementation is very efficient despite its low frequency. Our design permits general-purpose programmability and can be applied to other application areas as well. Such applications having similar types of data parallelism could benefit from reduced costs and smaller execution times on our vector processing system. With continued advances in FPGA technologies, the expected increased speeds and densities could yield much better performance in the near future for such computationally intensive problems on FPGA-based vector implementations.

For area/power efficiency, the vector processor is organized in eight banks; each bank has part of the vector register file and two arithmetic units. The memory is also divided into eight banks to provide higher bandwidth.

2.1 Architecture of the Vector Processor

Our vector processing system includes two identical programmable vector microprocessors embedded in two Xilinx Virtex-II XC2V6000 chips on the Annapolis WildStar II board [53]. The Annapolis board can be mounted on the host machine through a PCI socket [53]. Libraries in the host machine and drivers for the board are provided for data transmissions between the off-chip/on-chip memories and the host. Our two vector processors communicate with the host machine via an on-chip dual port memory. The host machine assigns work to them based on their requests and the load balance. Each vector processor runs at 70MHz and supports the IEEE 754 single-precision floating-point standard and efficient implementation for sparse matrices. The overview of the system is shown in Fig. 2.1.

The vector processor is composed of a vector core and a tightly coupled five-stage pipelined scalar unit, as shown in Fig. 2.2. It is organized as a Harvard architecture with

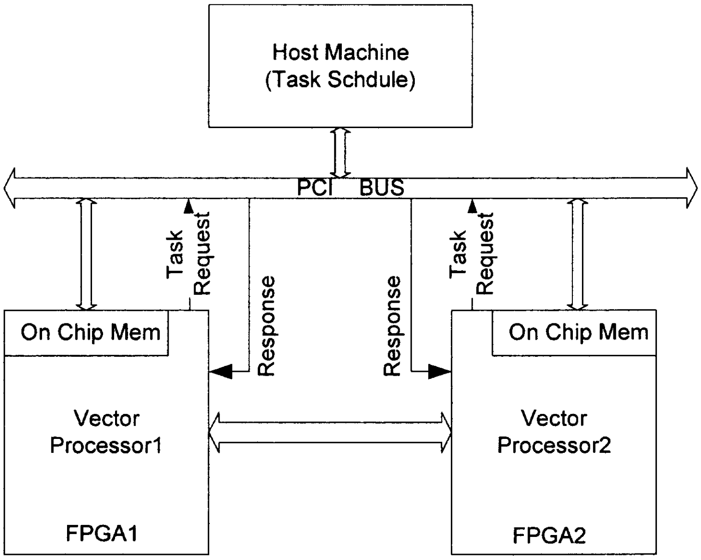

Figure 2.1: Overview of the system.

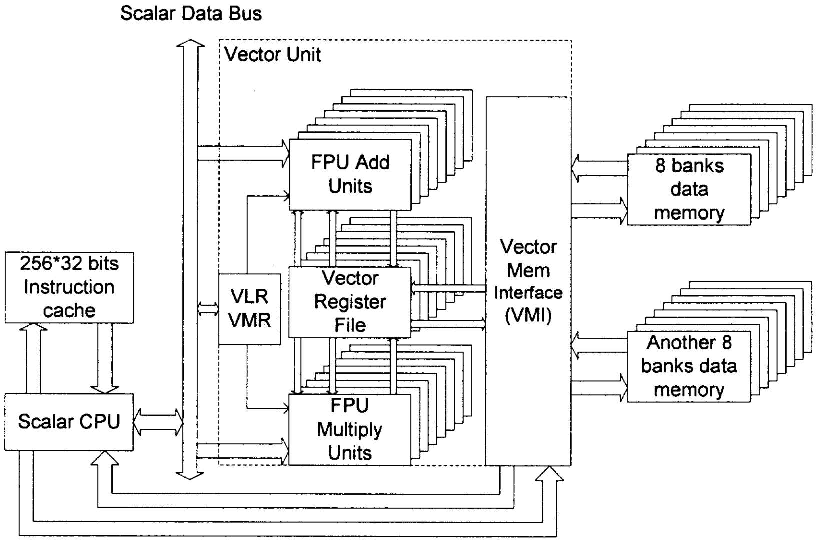

Figure 2.2: Block diagram of the vector processor (VLR: Vector Length Register, VMR: Vector Mask Register).

separate bus interface units for instruction and data access. The scalar processor fetches and decodes instructions. It does the actual work for scalar commands and forwards the vector instructions to the vector core. The vector core is structured as eight parallel lanes, where each lane contains a portion of the vector register file, a floating-point multiplier, a floating-point adder and connection to the eight-bank memory system. It can produce up to eight results and get a maximum of eight data items from the memory banks per clock cycle.

2.1.1 Instruction Set Architecture

Our vector processor has a RISC architecture supporting 24 instructions. There are 16 scalar instructions in the areas of data transfer, arithmetic operations and program control. The other eight instructions run in the vector mode for data transfers and arithmetic operations. The latter instructions are of Type A or Type B, as shown in Fig. 2.3. Type A instructions use up to two source registers and one destination register. Type B instructions use one destination register and a 16-bit immediate operand. Although we do not need eight bits to represent the opcode or registers, it is a good choice for a possible extension

Bit	32--24	23--16	15--8	7--0
Type A	opcode	Destination Register	Source Reg2 (optional)	Source Reg1 (optional)

Bit	32--24	23--16	15--0
Type B	opcode	Destination Register	Immediate Data

Figure 2.3: Instruction formats.

for the instruction set or register populations in the future. Two addressing modes are used in our design: direct and register indexed.

2.1.2 Scalar Unit

The scalar processor in our system supports 16 instructions for control, register and memory access, and arithmetic operations. There is a five-stage pipeline (fetch, decode, execute, memory access and write back) in it, as shown in Fig. 2.4. This scalar processor includes an arithmetic logic unit (ALU), a register file, a data hazard detection unit, and a data forwarding unit. For the sake of simplicity, Fig. 2.4 does not depict all the hardware. The shaded areas are unique to the vector system design; they are used to transfer useful information to the vector core. The ALU is able to deal with 16-bit integer addition/subtraction and mul-

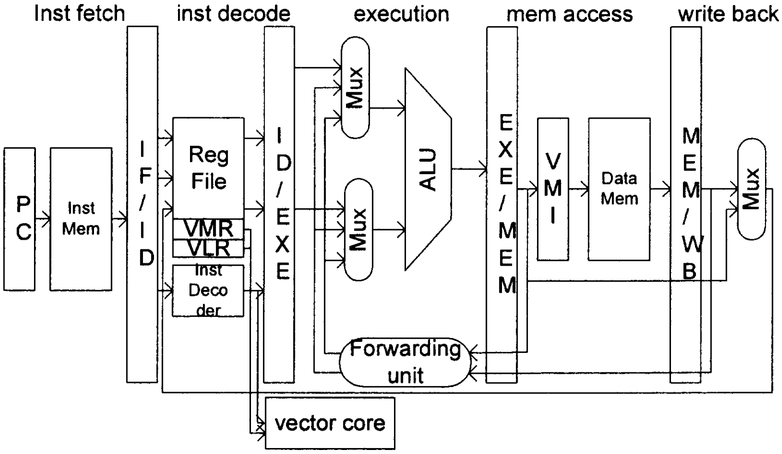

Figure 2.4: Scalar processor architecture.

tiplication. The register file includes 30 general-purpose registers and two special-purpose registers for vector processing. It supports two read ports and one write port.

Two specialized registers in the register file are used to control vector operations. They are: vector-length register (VLR) and vector-mask register (VMR). VLR is used to control the length of vector operations and VMR indicates that operations are to be applied only to the vector elements with corresponding entries equal to 1 in VMR.

To avoid EXE and MED data hazards due to pipelining, data hazard detection and forwarding units are implemented. We must emphasize that all scalar pipeline hazards can be avoided either with data forwarding or interlocking in hardware, so scalar instruction scheduling is not required for correctness; however, it may improve performance. This greatly eases code writing for our processor.

2.1.3 Vector Register File

The vector register file lies in the heart of the vector unit. It provides both temporary storage for intermediate values as well as the interconnect between the vector floating-point units (VFUs) [14]. A straightforward way to implement the vector register file is to use a single multi-ported memory. But this is a very expensive solution requiring many logic resources that increase the power consumption of the FPGA chip. Take the example of eight vector registers each having 32 32-bit elements; the left diagram in Fig. 2.5 shows the slice usage for a Xilinx XC2V6000 chip and the right one shows the power consumption assuming that it runs at 70MHz. It can be observed that the slices will be used up quickly and the power consumption increases greatly for an increased number of ports. We'd like to emphasis here that all the results presented in this chapter are after the place-and-route step for the XC2V6000 chip.

To reduce the cost, we could divide the vector register file into banks having smaller numbers of register elements and ports. A similar method has been used in a media processor [7] and a smart memory structure [54]. In our design, the vector register file is divided

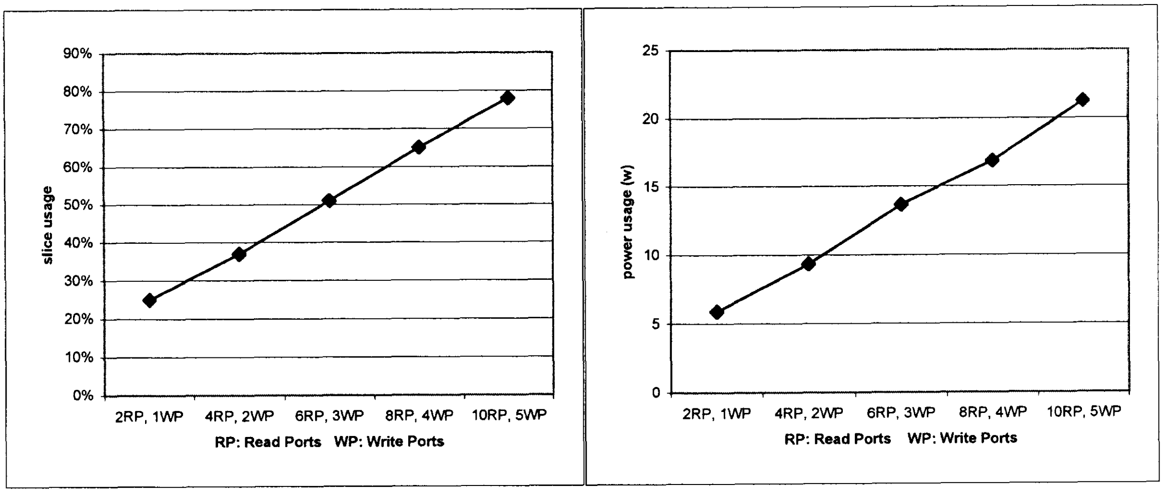

Figure 2.5: Resource and power consumption for single-block implementation of a vector register file containing 8 vector registers of 32 32-bit elements.

into eight banks, where each bank has two read ports and one write port. Take the example of 8 vector registers, each having 32 32-bit elements; the vector register file construction and its connections with other components are shown in Fig. 2.6.

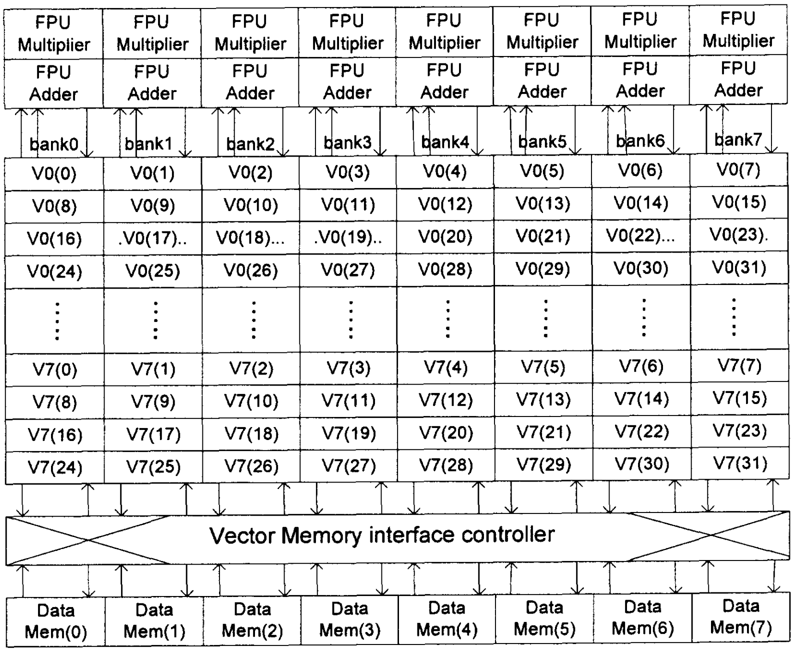
FPU Multiplier
FPU Adder
bank0
bank1
bank2
bank3
bank4
bank5
bank6
bank7
V0(0) V0(1) V0(2) V0(3) V0(4) V0(5) V0(6) V0(7)
V0(8) V0(9) V0(10) V0(11) V0(12) V0(13) V0(14) V0(15)
V0(16) .V0(17).. V0(18)... .V0(19).. V0(20) V0(21) V0(22)... V0(23).
V0(24) V0(25) V0(26) V0(27) V0(28) V0(29) V0(30) V0(31)
V7(0) V7(1) V7(2) V7(3) V7(4) V7(5) V7(6) V7(7)
V7(8) V7(9) V7(10) V7(11) V7(12) V7(13) V7(14) V7(15)
V7(16) V7(17) V7(18) V7(19) V7(20) V7(21) V7(22) V7(23)
V7(24) V7(25) V7(26) V7(27) V7(28) V7(29) V7(30) V7(31)
Vector Memory interface controller
Data Mem(0)
Data Mem(1)
Data Mem(2)
Data Mem(3)
Data Mem(4)
Data Mem(5)
Data Mem(6)
Data Mem(7)

Figure 2.6: Vector register file organization and connections with other components.

The vector memory interface (VMI), FPU adder and FPU multiplier share the read/write ports of the register file in a time-multiplexed way. The bandwidth of the vector register file in this configuration is 6.72 GBytes/sec when operating at 70 MHz. If the equivalent bandwidth is to be provided by a single register bank, a register file with 16 read ports and 8 write ports would be required. This is significantly less efficient in terms of area, speed and power consumption than the bank-based architecture since the latter only consumes 24% of the slices and 4.3W of power while the resource and power consumptions increase dramatically with an increase in the number of ports, as shown in Fig. 2.5. The additional cost of the bank structure corresponds to a circuit for data transfers between any pair of memory-register banks. This circuit uses quite a few FPGA resources and lies in the critical path; but comparing to the single register file implementation, this resource usage is much smaller and does not change the fact that the bank structure is much more efficient than the single block implementation.

Besides the structure of the vector register file, we also need to determine its size. Eight vector registers are chosen in our implementation. Although increasing the number of vector registers can reduce the memory bandwidth requirements by allowing more data reuse, most matrix-based applications have little data reuse. Thus, eight vector registers suffice and can demonstrate the effectiveness of our design. Each vector element has 32 bits, which is required for single-precision floating-point calculations. More vector elements in a vector register could amortize the startup time and speedup the overall execution; the time to fill up the pipeline is eight clock cycles for floating-point multiplication and eleven clock cycles for floating-point addition. So, we decided to implement as many elements as allowed by the available resources without increasing the circuit complexity tremendously. We experimented with 8, 16, 32 and 64 elements per vector in our design. With the increased circuit complexity and congestion of the on-chip routing resources for more elements, the system frequency of the design drops from 70 MHz for 8, 16 and 32 elements to 62.5 MHz for 64 elements. A more substantial frequency reduction should be expected for more elements.

2.1.4 Vector Memory Interface (VMI)

VMI controls all the data transfers to/from the data memory banks. It supports scalar loads/stores from/to any data memory bank, vector loads/stores starting with any data memory bank and for any length, and indexed loads/stores for sparse matrices. The execution time of vector load/store and indexed load/store is not deterministic. The starting point in memory and the length of the data affect the execution time of these operations. Besides the impact of the vector length, different data storage patterns in the eight data memory banks may result in different contention patterns for the indexed load/store, thus resulting in different execution times. A 8x8 32-bit crossbar is implemented with 16 eight-to-one 32-bit multiplexers to transfer data between any pair of memory-register banks. This crossbar has a constant switch delay of one clock cycle. Because of the complicated control logic and large number of multiplexers, it contributes to part of the critical path in our design.

2.2 Performance Results

To test the performance of our vector processing system, dense matrix-matrix multiplication, sparse matrix-vector multiplication and a sparse linear equation solver were run on it. Our system makes no assumption about the sparsity structure of the input matrices to guarantee the performance. Various matrices from different application areas with different sparse structures are given as input. Comparisons are made with the calculation time on a commercial Dell PC that contains a 1.2G Pentium-III processor, 512M bytes of memory and 512K bytes of L2 cache, and employs the Linux operating system. It is a difficult job to get the accurate execution time on the PC; we apply several criteria to make sure that the time measured on the PC is representative of its capabilities [55].

1. Since the time for our experiments is at the milliseconds level, the interrupt timer that samples every 10ms is imprecise. So we use assembly-language code to reach the IA32 clock cycle register and get the clock cycles used by the program.

2. In order to minimize data and instruction cache misses, the code is run once before it is run again to measure the time.

3. We run the program twenty times and take the smallest value as the calculation time.

For the execution time on our vector processor, the transmission time between the host machine and the FPGA board is not included. Because data transmissions are through the PCI socket and under the control of the interrupt handler, the delay time is large and unpredictable. [44] presents a way to hide such latencies. We emphasize the architecture of the vector processor and how to provide efficient vector processing. Therefore, the transmission time is omitted in our performance analysis. We can see from the following examples that the clock cycles on our vector processing system are hundreds of times smaller than those on the Dell PC. But the architecture of the FPGAs limits the maximum frequency, so our system can only run at 70MHz. Despite the tremendous frequency disparity compared to the 1.2GHz of the commercial PC, we will see that our vector processing system still outperforms the PC when the matrix size becomes larger or when two vector microprocessors are used.

2.2.1 Preliminaries on Vector Processing

Vector computers require advanced pipelining [56]. Longer vectors in the program will provide better performance because the penalty caused by the start-up time is amortized. Actually the throughput of the vector processor highly depends on the length of the vectors. The recurrence problem and sparse matrices are two general situations which forbid generating long vectors. Without appropriate preprocessing, the vector processor can hardly achieve significant improvement [57, 58]. So in implementations the data storage scheme

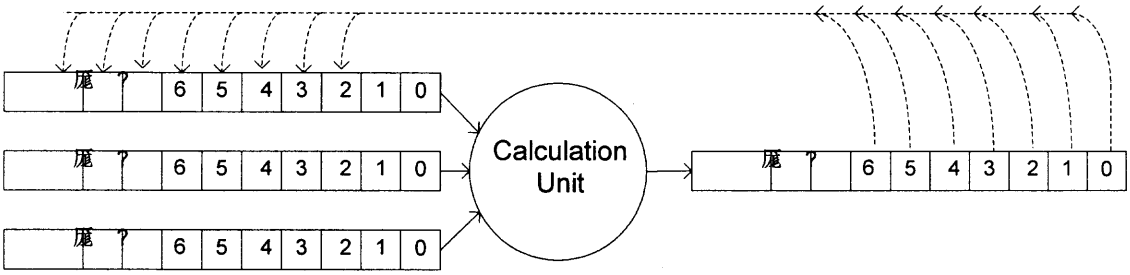

Figure 2.7: Recurrence problem.

and application algorithm are modified to get longer input vectors. We explain respectively how to avoid the effects of the recurrence problem and sparse matrices in the flowing two subsections.

2.2.1.1 The Recurrence Problem.

Optimizing a code for vector processing basically consists of arranging the algorithmic steps in such a form as to produce long and vectorizable DO loops [58]. One of the most common obstacles to vectorization is the "recurrence" problem; it is when a variable is obtained based on the value of that variable in an earlier iteration [2], as shown in Fig. 2.7. The delay caused by the recurrence problem can be large, especially for floating-point operations which require more clock cycles. Generally, this problem can be solved by loop unpacking and parallel calculations. Multiplication and accumulation (MAC) is highly used in matrix operations, and creates a recurrence problem. A common solution to MAC is to use more addition units and organize them in a binary tree. This way the delayed clock cycles can be reduced from $n \times (delay\ clocks\ for\ addition)$ to $\lceil \log(2, n) \rceil \times (delay\ clocks\ for\ addition)$. But for floating-point operations, where, for example, 11 clock cycles are needed for single-precision floating-point addition in our implementation, the binary tree structure still has quite a significant delay. Also, the binary tree needs more circuit area and more complex control logic, which may make the MAC unit in a microprocessor a critical part in the circuit design [59]. So in our implementation we replace the MAC operations by vector multiplication and addition as shown in the following pseudo codes. Take the example of matrix-vector multiplication; matrix-matrix multiplication can be seen as an extension to matrix-vector multiplication.

```
      loadV   V1  col 1 of result matrix c
loop1:loadV   V2  col j of matrix a
      load    R1  j element of vector b
      mulVS   V3  V2   R1
      addV    V1  V3   V1
      j--, if not zero, jump to loop1
      storeV V1   col 1 of result matrix c
```

2.2.1.2 Pseudo-Column Ordering for Sparse Matrix Operations. The efficiency of sparse matrix multiplication highly depends on how the non-zero entries are represented in the memory. A proper storage scheme can greatly enhance the performance of the vector processor by allowing much longer vectors even in a very sparse matrix. The pseudo-column method stores the non-zeros in a compact way by shifting all non-zeros in a row to the left. The following example shows that the non-zeros in matrix A are shifted to the left to produce matrix AP and their corresponding column number is stored in matrix CI.

$$A = \begin{bmatrix} & & 31 & & 51 & & \\ 12 & & & & & & 72 \\ & & & & & 63 & \\ & & & & & & \\ & & 34 & 44 & & & \\ & 26 & & & 56 & 66 & \\ 17 & 27 & & 47 & & & \end{bmatrix} \quad AP= \begin{bmatrix} 31 & 51 & \\ 12 & 72 & \\ 63 & & \\ & & \\ 34 & 44 & \\ 26 & 56 & 66 \\ 17 & 27 & 47 \end{bmatrix} \quad CI= \begin{bmatrix} 3 & 5 & \\ 1 & 7 & \\ 6 & & \\ & & \\ 3 & 4 & \\ 2 & 5 & 6 \\ 1 & 2 & 4 \end{bmatrix}$$

These two matrices are not actually built; they are just used for explanation. Values and column indices are stored in the memory based on the pseudo columns. Because the pseudo-column only contains data from different rows, the recurrence problem caused by

MAC can be avoided by vector multiplication and addition. Another advantage of this storage scheme is that elements in the pseudo-columns can be loaded contiguously and accessed in a unit stride fashion.

2.2.2 Sparse Linear Equation Solver

In a conventional way, a linear system of equations given in the form $Ax = b$ is solved by factorizing matrix A into triangular matrices and then carrying out data forward/backward substitutions. These substitutions are essentially sequential and hinder parallel computing. Getting the inverse matrix of A and solving the problem by matrix multiplication is an obvious solution, but the question is how to get the inverse matrix. There will also be a major problem with sparse matrices: the inverse matrix of A may turn to be dense even if A has only a few non-zeros. The W-matrix method, which was proposed in [60, 61], provides an easy way to get the inverse matrix by partitioning the sparse matrix into a product having partitions with no fill-ins or with user controlled fill-ins. Thus, in each partition calculations can be done in parallel. Several W-matrix solvers have been run on shared-memory computers, multiprocessors and vector supercomputers [57, 58, 62, 63]. We show in this paper that it can also run efficiently on our vector system.

2.2.2.1 The W-matrix Method. A linear equation given in the form

$$Ax = b \tag{2.1}$$

where A is a large, sparse and symmetric matrix can be solved in the following steps:

$$x = A^{-1}b = (LDU)^{-1}b = U^{-1}D^{-1}L^{-1}b \tag{2.2}$$

where L, D and U represent the decomposition of A into a lower triangular, diagonal and upper triangular matrix, respectively. With appropriate ordering [61], we can first reduce the factorization fill-ins and factorize A into the form LDL^T. After this ordering, assume that $W = L^{-1}$. Then, (2.2) can be rewritten as:

$$x = W^T D^{-1} W b. \tag{2.3}$$

It is obvious that (2.3) can be solved in three steps:

$$z = Wb; y = D^{-1}z; x = W^T y \tag{2.4}$$

that replace forward and backward substitutions with matrix-vector products. Within each step, all multiplications can be carried out concurrently, which is suitable for parallel programming and vector computing. W-matrix is associated with algorithms that partition the inverses of L and U into elementary matrices with no fill-ins or only user controlled fill-ins. Based on [61] and [60], we can write matrix L as

$$L = L_1 L_2 \cdots L_n \tag{2.5}$$

where L_i is an identity matrix except that its i-th column is actually the i-th column of matrix L. Then:

$$W = L^{-1} = L_n^{-1} L_{n-1}^{-1} \cdots L_1^{-1} = W_n W_{n-1} \cdots W_1 \tag{2.6}$$

where W_n is equal to L_n with the sign of its off-diagonal elements reversed. Plugging (2.6) into (2.3), we get the expression:

$$x = W_1^T W_2^T \cdots W_n^T D^{-1} W_n \cdots W_2 W_1 b. \tag{2.7}$$

To avoid fill-ins induced in (2.7), we need $2n + 1$ sequential steps of multiplication to get the final solution; it has no advantage over the common substitution method. But according to [60, 61], adjacent matrices W_i, for $1 \leqslant i \leqslant n$, can be combined in several ways to form various partitions:

$$x = W_1^T W_2^T \cdots W_p^T D^{-1} W_p \cdots W_2 W_1 b. \tag{2.8}$$

Now the triangular factors are partitioned into p parts, where we can have $p \ll n$ for a large n. According to (2.8), the solution x can be obtained after $2p + 1$ steps, where many operations can be executed concurrently in each matrix-vector product step. Different reordering and partitioning schemes based on the factorization path length tree [60, 61] show that the W partitions can be chosen without adding new fill-ins or with adding only user controlled fill-ins in efforts to minimize the number of arithmetic operations. Thus, the combined sparsity of the p factors can be the same as that of L.

2.2.2.2 W-matrix Method Implementation on the Vector Processor. At static time algorithms for approximate minimum degree ordering and LU factorization were applied to the input matrix, then the elimination tree of the matrix was generated and the W-matrix was finally transformed based on the path lengths in the elimination tree [60, 61]. Finding out how to partition the inverse triangular factors in order to get the shortest solution time on the vector processor is difficult since there are numerous ways to form W-matrix partitions. The partitioning method used in this paper is easy to implement and can also guarantee good performance. We generate W-matrix partitions in such a way that when an added column increases the non-zeros in the partition to more than two times the original non-zeros in those columns in the L/U matrix, a new partition will be created. Partition numbers for each input matrix and non-zero numbers after each step are shown in Table 2.1. It can

Table 2.1: Non-Zero Element (NNZ) Changes in W-Matrix Partition Generation

Matrix Size	Original NNZs	NNZs after LU	Partitions	NNZs in w-matrix
49 × 49	118 / 4.9%	160 / 6.7%	4	265 / 11%
118 × 118	358 / 2.6%	526 / 3.8%	4	792 / 5.7%
443 × 443	1180 / 0.6%	1936 / 1.0%	5	3543 / 1.8%
1454 × 1454	3840 / 0.18%	6878 / 0.33%	6	11434 / 0.54%
1723 × 1723	4782 / 0.16%	8984 / 0.30%	7	14307 / 0.48%

be seen that, after W-matrix factorization the sparsity of the matrix is still large. After the W-matrix partitions are formed, pseudo-columns are generated by the host computer, and data is downloaded into the FPGA board for actual computations. This experiment is suitable for those applications that require repetitive linear equation solutions without any change in the input matrices.

2.2.2.3 Performance Result. The linear equation solver uses input matrices from real power networks containing 49 to 1723 nodes [64]. It is run on one vector processor and the result is compared with that of a commercial PC. It is not distributed into two vector processors because there will be a lot of data communication and synchronization, which will increase the solution time significantly. Performance comparisons with the commercial PC are shown in Fig. 2.8.

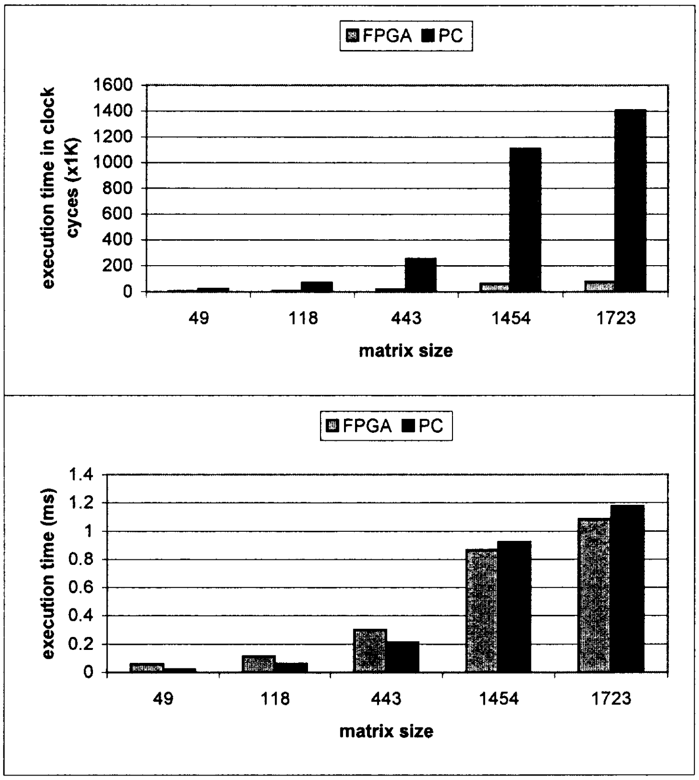

Figure 2.8: Performance of sparse linear equation solver.

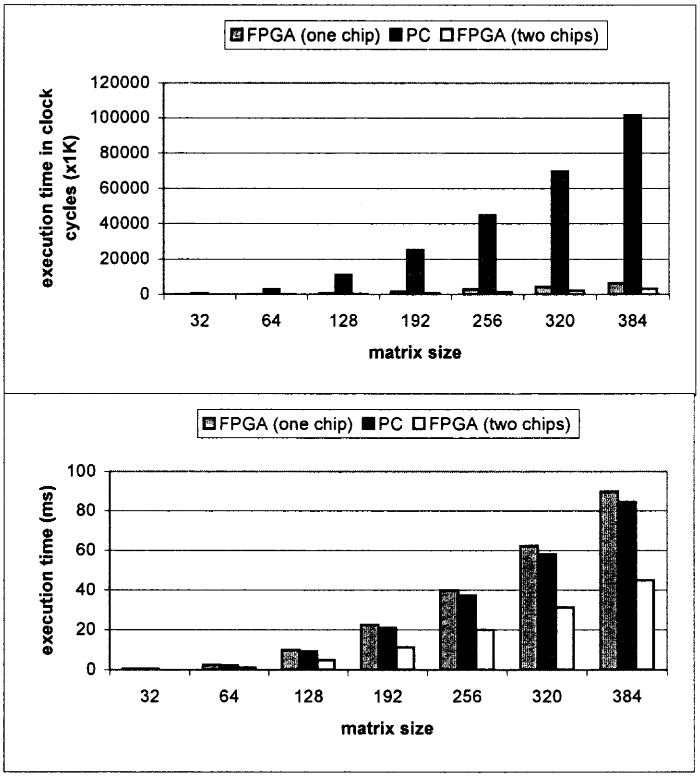

Figure 2.9: Performance of dense matrix-matrix multiplication.

2.2.3 Dense Matrix-Matrix Multiplication

The dense matrix is divided into 32x32 sub-blocks because of limitations in the capacity of the on-chip data memory. Matrices of different size were run and the performance is shown in Fig. 2.9. For simplicity in programming and without loss of generality, we assume test matrices with dimensionality which is a multiple of the sub-block dimensionality. Also, for a fair comparison with a commercial PC, we did not really use very large matrices on the PC but just used a loop for sub-block multiplication and addition on the same 32x32 sub-blocks; the input matrices were created in such a manner that all the sub-blocks contain the same data. This way, matrix data can all be stored in the cache and there will be no large time delays because of cache misses. Otherwise, the calculation time on the PC may increase cubically with increases in the matrix size, not conforming to the proportional increase shown in Fig. 2.9. If these matrix-matrix multiplications are divided to run on two vector processors residing on two FPGAs, then the calculation time is reduced approximately by half since equivalent numbers of calculations are assigned to the FPGAs. Fig. 2.9 shows the performance of matrix-matrix multiplication on one vector processor, the commercial PC and two vector processors. We can see that the two vector processors achieve a speedup of about 53% compared to the PC.

2.2.4 Sparse Matrix-Vector Multiplication

Sparse matrices from various engineering application areas, including power flow, structural engineering and aircraft design, were run on our vector processing system for evaluation. All these matrices are from the Harwell-Boeing Collection [64]. For simplicity and without loss of generality, all chosen matrices are square and symmetric. Table 2.2 gives the basic characteristics of the input matrices used in our experiments of sparse matrix-vector multiplication. The calculation time depends on the numbers of non-zero elements since they determine the amount of floating-point operations needed. Another factor affecting the calculation time is the maximum number of non-zeros in rows because this number

Table 2.2: Input Sparse Matrices from the Harwell-Boeing Collection in Our Experiments

Name	Application Area	MD[1]	NNZ[2]	% of NNZ	ANNZ[3]	MNNZ[4]
can 144	aircraft/structure	144	1296	6.25%	9	15
dwt 193	structure/knee prosthesis	193	3493	9.38%	18	30
dwt 992	structure/mirror	992	16744	1.70%	17	18
jagmech9	finite-element model	1349	9101	0.50%	6.7	7
lshp2614	finite-element model	2614	17980	0.26%	6.9	7
bcspwr10	power networks	5300	21842	0.078%	4.1	14

[1] MD: Matrix dimension.

[2] NNZ: Number of Non-Zeros.

[3] ANNZ: Average Number of Non-Zeros per row.

[4] MNNZ: Maximum Number of Non-Zeros in rows.

decides how many pseudo-columns will be generated, which in turn decides the number of loops to run. For larger matrices having more non-zeros, the calculation is divided into the two vector processors by dividing the matrix into sub-blocks. Since the distribution of non-zeros is not balanced in these sub-blocks and in the dense matrices, the calculation time on the two processors varies from 50% to 100% of the time on one processor. For matrices with a balanced distribution of the non-zeros, the calculation time can be reduced by half when run on two processors (e.g., *jagmech*9 and *lshp*2614). In a well unbalanced situation (e.g., *dwt*992), the time can only be reduced by about 30%.

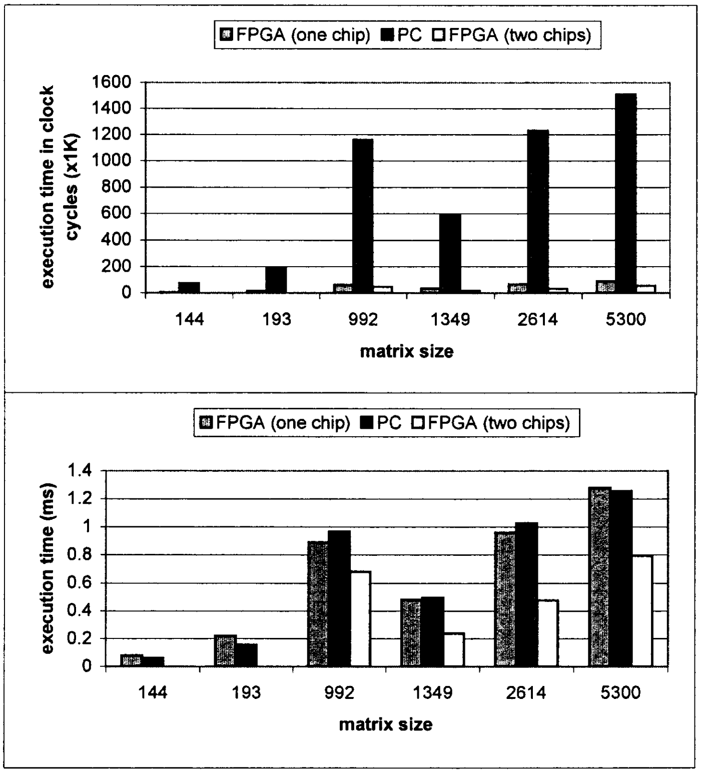

Figure 2.10: Performance of sparse matrix-vector multiplication.

CHAPTER 3

MULTICORE PROCESSOR SYSTEM WITH A SHARED VECTOR UNIT

There has been a recent trend to implement single-chip multicore processors on because of limited ILP exploitation in real applications and uniprocessor power dissipation problems. A multicore design has the potential to provide higher throughput, easier scalability, and greater peformance/power ratio than its single-core comparables [30]. Incorporating a shared vector unit in a multicore system as an accelerator for computation-intensive tasks could greatly increase system performance through DLP at low area and power costs. Various applications can then be mapped onto such a system. Results to support this argument are shown in the next chapter.

Relevant system design and implementation issues are detailed in this chapter. The system is implemented on the latest Xilinx FPGA series: the Virtex-5. MicroBlaze, the 32-bit soft processor from Xilinx, is used as the scalar processor in the system. The benefit is that we can then use mature high-level languages and compilers to implement a substantial part of the applications. The vector unit designed by ourselves is closely coupled to the MicroBlazes through their fast data links. A decoupled structure, in which the instruction stream is split into several instruction queues for different function units and load/store units, can hide part of the latency of fetching data from the memory by the fact that the load/store unit can start fetching data ahead of the computation units and increase the total throughput of the system [65]. The benefit is even more prominent in the multithreaded environment [65]. In [66], large speedups are reported with very simple hardware and a small number of threads. Our shared vector unit in the MCwSV system, dealing with instructions from several scalar processors, works in the multithreaded way and the efficiency can be improved greatly by employing a decoupled technique. So along with the multibank structure implemented in Chapter 2, our vector unit is also organized in a decoupled way with two arithmetic units and one load/store unit. This architecture can further increase the scalability and flexibility for the system because of the modular design approach. We also designed the vector unit to support configurations with 1, 2, 4 or 8 banks, and different register/element sizes to add even more flexibility to the system. For a better understanding of the system, we will first present an introduction to MicroBlaze and the Virtex-5 FPGA chip. Then, the design diagram and implementation will be explained.

3.1 Scalar Processor and FPGA Resources

3.1.1 MicroBlaze Soft Processor

MicroBlaze is a 32-bit Harvard RISC architecture optimized for Xilinx FPGAs. We used version V5.00 with a five-stage pipeline. The stages are Fetch (IF), Decode (DF), Execute (EX), Access Memory (MEM), and Writeback (WB). The block diagram of MicroBlaze is shown in Fig. 3.1. Its basic architecture consists of 32 general-purpose registers, an Arithmetic Logic Unit (ALU), a shift unit, and two levels of interrupts. The shaded parts in the diagram are optional and can be configured to meet the exact needs of the target application

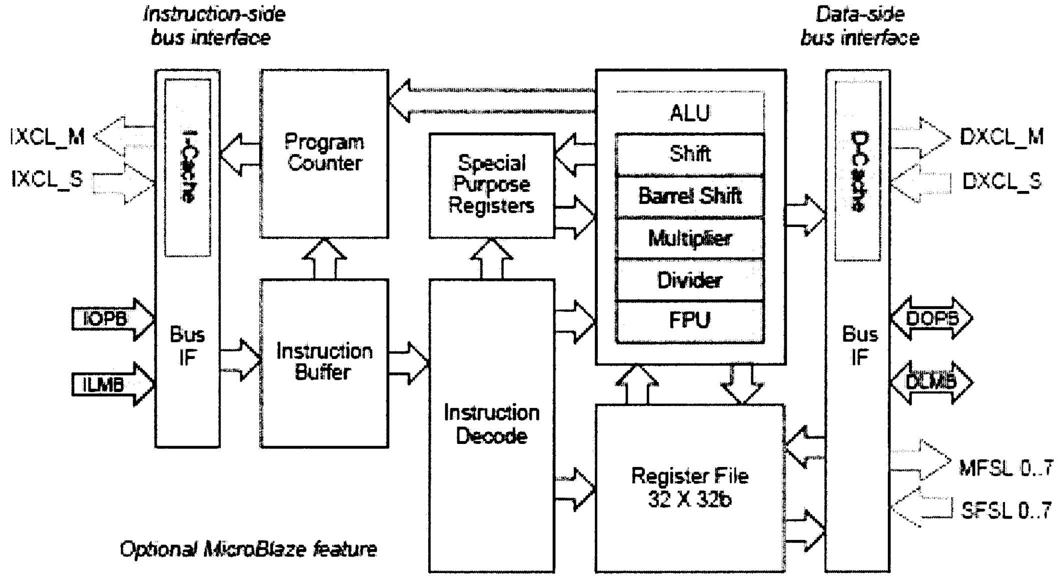

Figure 3.1: MicroBlaze core block diagram [1].

domain. They provide great flexibility but do not add any cost if they are not needed for a specific system. These configurable features include the barrel shifter, integer divider, integer multiplier, floating-point unit, instruction and data caches, on-chip peripheral bus (OPB), fast simplex link (FSL) interfaces, and hardware debug logic. Depending on the configurable options and target FPGA, MicroBlaze uses about 800 to 2600 LUTs.

MicroBlaze has three kinds of bus infrastructure: local memory bus (LMB), FSL to interface co-processors and OPB to interface peripherals. We focus on the FSL and OPB buses here. MicroBlaze provides up to eight FSLs to interface hardware co-processors. Each FSL channel provides a low latency interface (two clocks for write and one clock for read) to the processor pipeline and is ideal for extending the processor's execution unit with custom hardware accelerators. The length and depth of the FSL FIFO is configurable. In our system, MicroBlaze executes scalar operations and sends vector instructions to the shared vector unit through FSL. There are three pairs of FSLs in our system: a pair between each MicroBlaze and the vector unit is used to transfer vector instructions and handshake signals; another exists between the two MicroBlaze processors for communication signals. An OPB bus that involves an OPB arbiter can support up to 16 masters and any number of slaves. Masters send data requests on the OPB bus; and the OPB arbitrator decides which request can be satisfied and then uses hand shake signals. In our system, the memory controller for the shared memory banks is attached to the OPB bus as one of its slaves. The vector unit as well as the two scalar processors send requests to the OPB bus to access the shared memory; they are all connected to the OPB bus as masters. Each scalar processor fetches instructions from its local memory using its own LMB.

3.1.2 Virtex-5 Resources

Our design is implemented on Xilinx Virtex-5 XC5VLX50-1FFG676 FPGAs. Virtex-5 is based on a 65-nm copper CMOS process and triple-oxide technology. This new manufacturing technology makes Virtex-5 more area and power efficient than its predecessors.

Along with the larger 6-input LUTs, Virtex-5 can provide maximum routing capability and accommodate a large number of built-in hard-IPs. Particularly for the XC5VLX50-1FFG676 chip, there are 7200 slices, 28800 6-input LUTs, 480Kb distributed memory, 48 36Kb block memories/built-in FIFOs, 6 clock management tiles with each containing two Digital Clock Managements (DCMs) and one PLL, and 48 DSP48E slices. Each of the DSP slices has a 25 x 18 two's complement multiplier or a 48-bit adder/subtracter/accumulator. The 36Kb block memory can also be configured as two independent 18Kb blocks/18Kb built-in-FIFOs.

3.2 System Design and Organization

Fig. 3.2 shows the block diagram of the system with two scalar processors and a shared vector unit. Each MicroBlaze has its own instruction/data memory connected via the local memory bus (LMB). The two cores share with the vector unit a low-order interleaved data memory which is organized in multiple banks. The scalar processors fetch the vector operations and subsequently forward them to the vector unit through the fast simple links (FSLs) interconnect. This system is easily scalable to many processors. The vector unit is composed of four parts: the vector issue logic (VIL), several function units, scalar registers and a central vector register file (VRF). The vector register file is organized in several banks, and can be configured into groups of 1, 2, 4 or 8 banks. The function units are also duplicated to match the number of banks in the vector register file.

3.2.1 Vector Issue Logic

The scalar processors send the vector instructions to their FSLs; the vector unit takes instructions out of each FSL in a round-robin fashion potentially balancing the load of the function units and substantially hiding the latency of memory accesses. VIL is also in charge of register renaming and instruction reformatting. VIL has a renaming table where every register in a scalar processor has a corresponding entry in it as shown in Fig. 3.3.

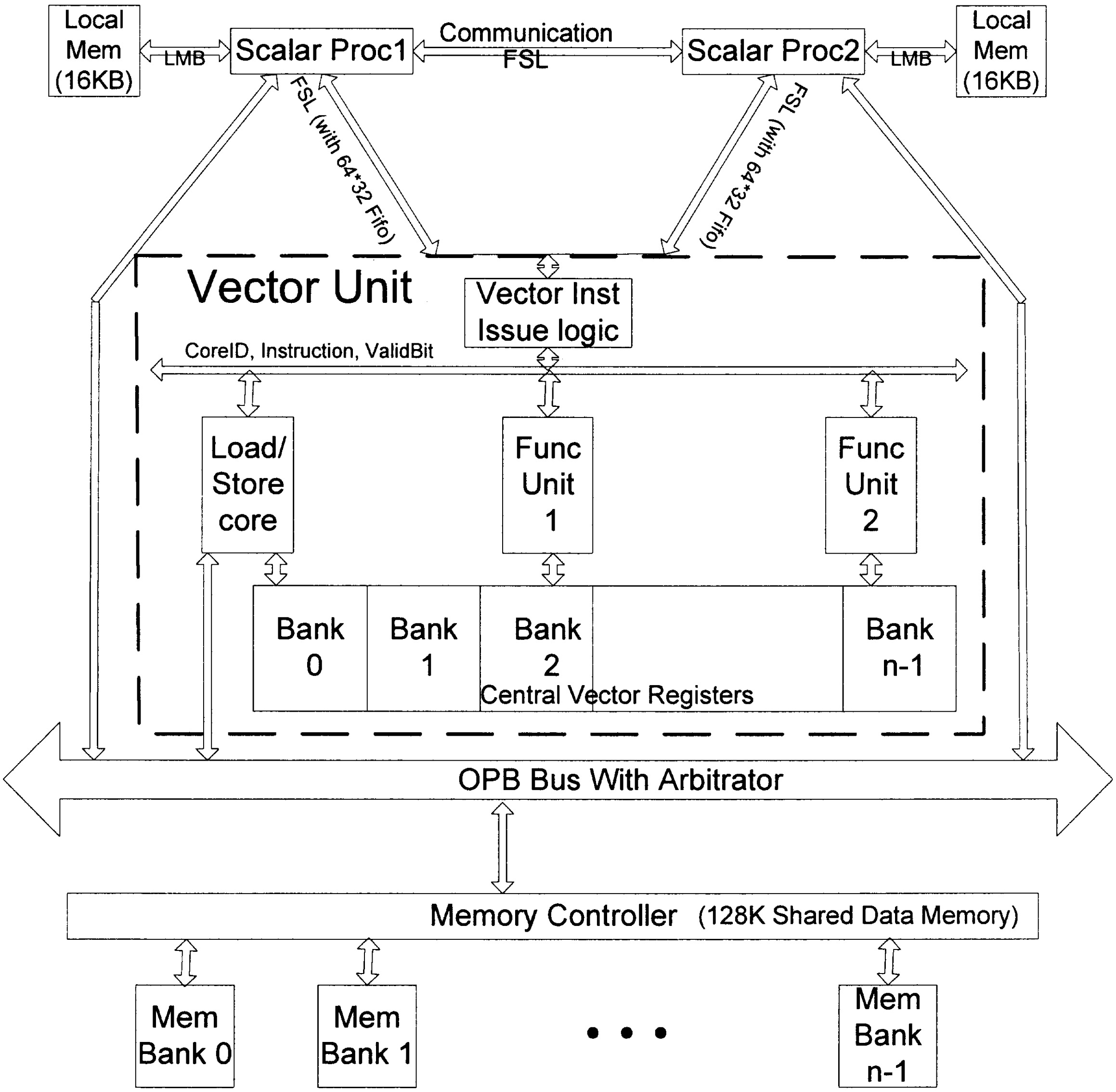

Figure 3.2: MCwSV multicore processor system with a shared vector unit (The scalar registers are not shown in the vector unit).

The renaming table maintains the current mapping between the vector registers in the instruction set of each scalar processor and the actual registers in the vector processing unit. VIL reads the instruction in, does vector register renaming, decides which function units this operation should go to, and then generates new, local instructions and broadcasts them on IBUS. Fig. 3.4 shows the instruction formats of incoming and outgoing instructions.

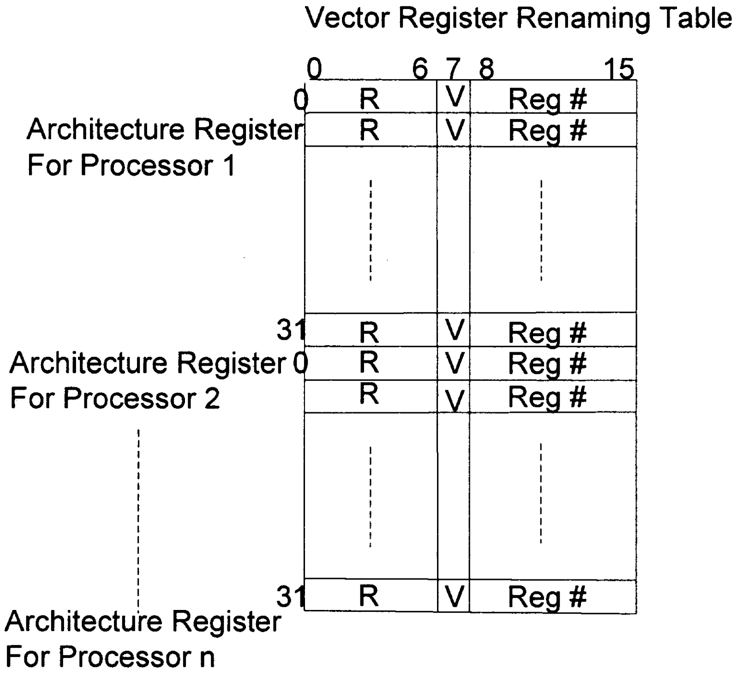

Figure 3.3: Vector register renaming table (R: Reserved; V: Valid).

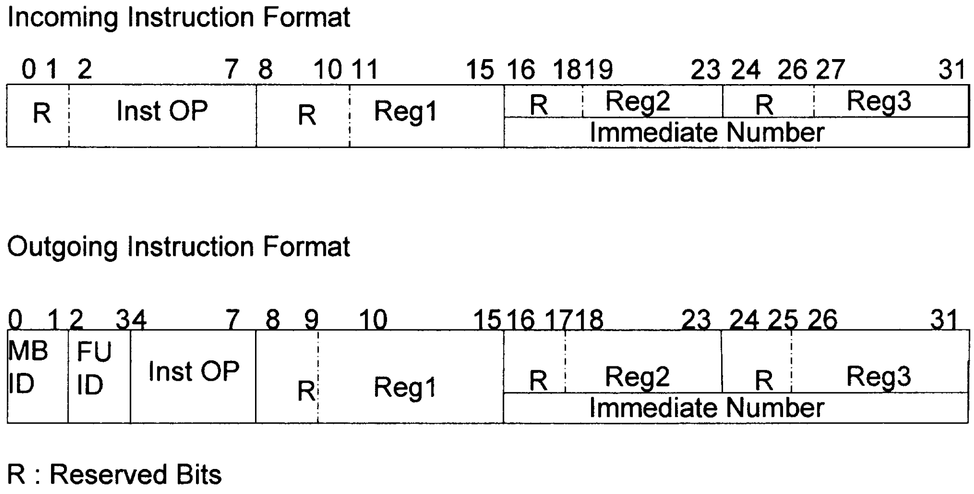

Figure 3.4: The format of incoming and outgoing instructions in VIL. "MB ID" indicates which MicroBlaze issued the operation; "FU ID" indicates which function unit should this instruction go to. (OP: Opcode, R: Reserved.)

3.2.2 Vector Register File

Designing high-performance low-power register files is very important to the continuation of current performance advances in wide-issue and deeply-pipelined superscalar microprocessors [67, 68]. Vector processors also have the same concern. And normally vector register file is very large, so the higher-performance and lower-power vector register file is of critical importance to the whole system design.

The vector register file can be organized in either a distributed or a centralized way; both organizations can employ the multibank structure with each bank containing several elements of the vector registers [3]. In the distributed structure [52], the vector unit is based on a clustered design. Each cluster has one function unit and part of the vector registers. Each distributed vector register file only needs to have two read ports and one write port to feed its corresponding function unit. A communication network is used to connect all the clusters. A data transfer is needed if the vector register is used by a remote cluster. A centralized register file serves all the function units; thus, we need to provide a number of ports which is three times the number of function units if all of them are to run simultaneously. But it eliminates the need for a crossbar to transmit data between different function units, which could add much more complexity to the control structure with additional power requirements as well. Also, highly frequent data transfers among the function units could further diminish the benefit of the distributed structure. Therefore, although the distributed register file has several advantages, we have chosen to use the centralized structure for our implementation.

Similar to the vector register structure in Chapter 2, the vector register file is divided into multiple banks for area and power efficiency [14]. The vector register file can be configured to have 1, 2, 4 or 8 banks, and the number-of-vector-register/vector-size pair can be configured to be 128/32, 64/64, 32/128, 16/256 and 8/512 where each word is four bytes long. Each vector register bank provides five read ports and three write ports for the three function units to be capable of running simultaneously. The vector register file construction and its connections with other components are similar to Fig. 2.6 in Chapter 2. The only difference is that the function units use two read ports and one write port in a time-multiplexed way in Chapter 2, but each function unit in our MCwSV system has its own read/write ports.

Since the vector register file in our implementation needs 16K bytes of storage space and eight ports, it could consume most of the slice resources in an FPGA chip if implemented by D registers or distributed memory blocks. Each Virtex-5 slices contains four 6-input LUTs and four flip-flops. Unlike ASIC technology which can be used to build a dedicated circuit for the large multiport register file, FPGA designs utilize pre-manufactured resources. Fortunately, current FPGAs provide abundant on-chip block memories which can be used for the register file. But the block memory only provides two read/write ports whereas our register file needs eight ports. So we have to use the block memory to form multiport vector registers, via time-multiplexing on its dual port. The block memory can be extended into 8-port vector registers as shown in Fig. 3.5. A DCM block is used to generate the synchronous clock with a frequency three times higher than that of the system clock. This implementation is very resource efficient, but it also constitutes a big part of the critical path; this reduces somewhat the operating frequency of the vector unit.

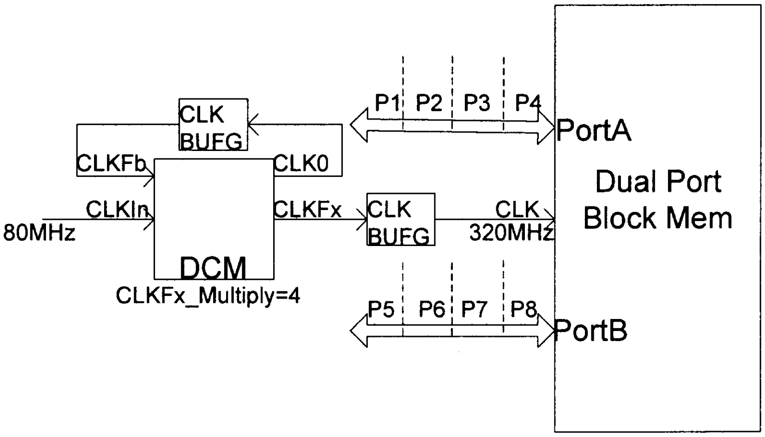

Figure 3.5: Multiport vector register implementation.

3.2.3 Function Units

A function unit gets an instruction from IBUS if the broadcast ID matches its own ID, and stores it in an execution FIFO. Then, it checks the renaming table to see if the source registers (either scalar or vector registers) are valid (have valid data); if they are, then it starts execution. This structure allows good scalability and easy implementation of instruction chaining. Chaining allows a vector operation to start as soon as the individual elements of its vector source operands become available; it acts like operand forwarding in scalar processor design (that is, the results from a function unit are sent to another function unit in a chain [2]). There are three function units in our system: load/store core, floating-point multiplier and floating-point adder/subtracter. If there is only one bank in the system, chaining acts as shown in the example of Fig. 3.6. If these function units are replicated for several banks and multiple vector elements can be fed into them per cycle, then multiple results can be produced simultaneously. With eight banks accessed in parallel, 24 operations can be processed simultaneously at peak rate with the support of chaining as shown in Fig. 3.7. In our implementation, the two function units are single-precision floating-pointer multiplier and adder/subtractor based on the IEEE 754 standard. They can be easily configured into other kinds of arithmetic units. The function units in our proof-of-concept system can be configured into 1, 2, 4 or 8 banks interfacing 1, 2, 4 or 8 memory banks, respectively.

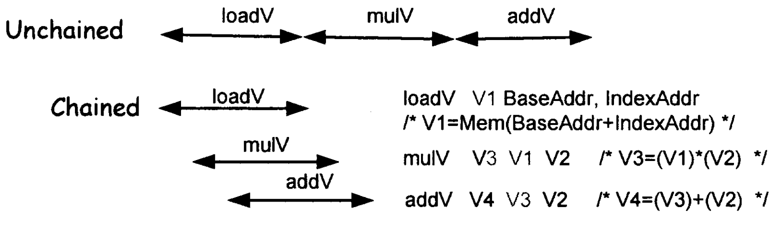
Unchained
loadV
mulV
addV
Chained
loadV
mulV
addV
loadV V1 BaseAddr, IndexAddr
/* V1=Mem(BaseAddr+IndexAddr) */
mulV V3 V1 V2 /* V3=(V1)*(V2) */
addV V4 V3 V2 /* V4=(V3)+(V2) */

Figure 3.6: Vector chaining.

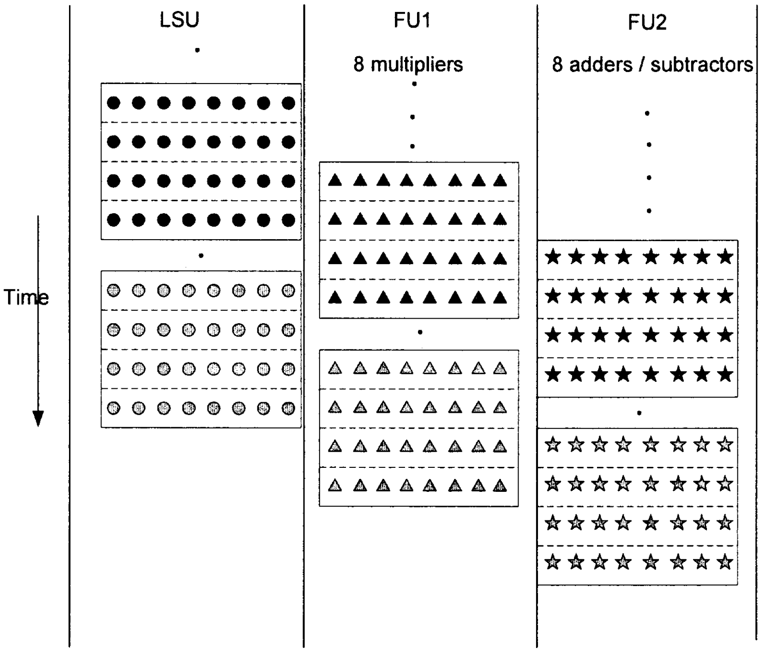

Figure 3.7: Vector chaining in a multi-lane structure (LSU: load/store unit).

3.2.4 Memory Controller

The memory also can be configured to have 1, 2, 4 or 8 banks in a low-order interleaved organization in order to match the structure of the vector register file. A memory controller was implemented to provide interleaved data to the vector processor as well as the scalar processors. Since the vector unit and the MicroBlaze processors are all connected to the OPB bus, the OPB arbitrator decides whose request should be served first. The memory controller is connected to the OPB bus as a slave. It receives requests from the OPB bus, loads or stores data from/to the data memory banks, and then sends back the data slave signals on the OPB bus. Our vector unit is connected to the OPB bus as a master. The OPB arbitrator could realize dynamic or fixed priority arbitration. We use the latter approach, giving the highest priority to the shared vector unit, then the first MicroBlaze and finally the second MicroBlaze. $2N$ $N - to - 1$ multiplexers are currently put between the multi-bank memory and the vector register file to act as a crossbar switch. They induce a delay of three clock cycles for each data read/write. These multiplexers are not needed for the 1-bank configuration. More advance interconnect organization could be developed for larger N.

3.3 System Implementation on the FPGA Chip

The platform Virtex-5 FPGA provides abundant dedicated function units such as block memories, embedded FIFOs and DSP blocks. These embedded devices are optimized for efficient usage of the silicon resources and outstanding performance. Targeting at high performance, our design tries to use such dedicated units whenever possible.

3.3.1 Digital Signal Processing (DSP) Blocks

DSP48E blocks serve as floating-point adders/subtractors and multipliers in the vector unit, and as floating-point operators, integer multipliers and integer dividers in the scalar processors. The two MicroBlaze processors in the system use 14 DSP48E tiles totally and 34 DSP blocks are left to the vector unit. Each MicroBlaze uses 7 DSP blocks for an integer adder/subtractor, an integer multiplier, an integer divider and floating-point calculations. The resource usage for the single-precision floating-point adder and multiplier is shown in Table 3.1. Even though the 3-DSP multiply implementation (the Xilinx library does not provide a similar implementation for add/sub) can increase the speed, it is not required for MCwSV because the floating-point arithmetic units do not create bottlenecks for the operating frequency. Thus, in our MCwSV system both the floating-point multiplier and adder/subtractor employ the 2-DSP implementation. If there are eight banks in the

Table 3.1: Resource Consumption of Single-Precision Implementations

Operation	Type	DSPs	LUTs	Flip-Flops
Multiply	DSP48E (3-DSP)	3	63	77
	DSP48E (2-DSP)	2	86	83
	logic only	0	664	142
Add/Subtract	DSP48E (2-DSP)	2	251	85
	Logic only	0	402	72

system, then the FPUs will need $(2DSPs(adder/subtractor) + 2DSPs(multiplier)) * 8(bank\ number) = 32$ DSP blocks.

3.3.2 Digital Clock Manager (DCM) Primitive

Three clock signals are needed in the MCwSV system; one is the system clock, which is 83.3MHz, 80MHz, 77MHz and 72.5MHz, respectively, for the 1-, 2-, 4- and 8-bank configurations; another is used for the 8-port vector registers whose frequency is four times the system frequency; the third one is used for the 6-port scalar registers and its frequency is three times the system frequency. All of the clocks must be synchronized. The FPGA provides the DCM primitive for implementing delay locked loop, digital frequency synthesizer, digital phase shifter, or digital spread spectrum. We use three DCMs in the MCwSV system to generate all the clocks: one is used to convert the 100MHz clock signal provided by the oscillator on the ML501 board to the system clock; another is used to generate the clock for vector registers from the system clock and the other one is used to generate the clock for scalar registers from the system clock.

3.3.3 On-Chip Block Memories

Our MCwSV system uses up almost all of the on-chip block memories and aims to provide large storage space with fast data access. As mentioned before, there are 48 36Kb block RAMs in our target FPGA chip XC5VLX50 and they can also be configured as 96 18Kb block RAMs. They are used as the instruction/data memory and FIFOs in our system, as shown in Table 3.2:

Table 3.2: Configuration and Usage of Block Memories

36Kb Configurations (40 blocks)		18Kb Configurations (16 blocks)	
Block #	Usage	Block #	Usage
32	128KB SDM[a]	8	Vector register file
		1	Scalar register file
4	16KB LIM/LDM[b] for MB0	3	256x32 FIFOs in two function units and load/store core
4	16KB LIM/LDM for MB1	3	256x32 FIFOs in FSLs between each MB and vector unit, and between two MBs
40 out 40 blocks		15 out of 16 used	

[a] SDM: Shared Data Memory.

[b] LIM: Local Instruction Memory. LDM: Local Data Memory.

3.3.4 Resource Usage for Various Configurations

The system is implemented on the Xilinx Virtex-5 series XC5VLX50 FPGA chip. Each MicroBlaze is attached to two pairs of FSL buses and an OPB bus, and contains an RS232 UART port, an integer multiplier, an integer divider and floating-point units. Each MicroBlaze uses 5% of the flip flops, 7% of the LUTs, 9% of the slices, and 7 of the 48 DSP blocks in this FPGA. The vector unit and the memory controller consume the resources shown in Table 3.3. The slice usage of the vector unit is less than that of one MicroBlaze for the 1- and 2-bank configurations, 1.3 times for the 4-bank configuration, and 2.4 times for the 8-bank configuration. The system frequency is 83.3MHz, 80MHz, 77MHz and 72.5MHz, respectively.

Table 3.3: Resource Usage of the Vector Unit and the Memory Controller

Banks #	1	2	4	8
DSPs	4	8	16	32
Flip-Flops	3%	4%	6%	11%
LUTs	6%	8%	11%	18%
Slices	7%	10%	14%	23%

CHAPTER 4

PERFORMANCE RESULTS AND ANALYSIS

The MCwSV system can find wide application in various scientific domains. To prove the viability of the shared vector unit concept, three testbenches were run on this system; the results are analyzed in this Chapter. The first application can be almost fully vectorized and is used to show the efficiency of a "standalone" vector unit. The second testbench provides performance evaluation when the vector unit is shared by two scalar processors. The third one demonstrates the use of stride load/store vector operations, and shows the relationship between speedup and the ratio of vector and scalar execution times.

4.1 RGB to YIQ Conversion (RGB2YIQ)

RGB(red, green and blue) to YIQ conversion is a benchmark in the EEMBC collection of embedded-computing benchmarks [20] that explores the CPU's capability to perform basic arithmetic and matrix math. The benchmark converts a digital image into the YIQ format that complies with the NTSC television standard. Each function takes a "point" in the RGB color space, and converts it into a luminance (Y) and two chrominance (I, Q) signals as follows:

$$\begin{aligned} Y &= (0.299 * R) + (0.587 * G) + (0.114 * B) \\ I &= (0.596 * R) - (0.275 * G) - (0.321 * B) \\ Q &= (0.212 * R) - (0.523 * G) - (0.311 * B) \qquad (4.1) \end{aligned}$$

The above equations can be fully vectorized for the vector unit in the MCwSV system; the scalar processor is only used for loop control, and preparing and transferring vector instructions. This is quite an ideal case for the vector unit because the distribution of tasks between its two function units (multiplier and adder) and the load/store unit is well balanced, and further parallelism is possible through instruction chaining.

As described in Chapter 3, the vector unit in the MCwSV system can be configured to have 1, 2, 4 or 8 banks and different number-of-vector-registers/elements-in-vector pairs. More elements in a vector register could amortize the startup time and enhance the overall execution time, so the largest number of elements for a vector register (its size) is chosen if possible. We assume 2^n points in the image for simplicity in programming, without loss of generality. The vector register size is chosen to match the number of pixels, when less than or equal to 256. This size cannot be increased further since there will not be enough vector registers to use. Table 4.1 shows the execution times on one MicroBlaze with and without the support of the vector unit under different configurations. Fig. 4.1 shows the speedup of using the vector unit as compared to a MicroBlaze without this unit for various vector configurations. In this comparison, the MicroBlaze runs at $100MHz$, whereas the vector unit runs at $83.3MHz$, $80MHz$, $77MHz$ and $72.5MHz$, respectively, for the 1-bank, 2-bank, 4-bank and 8-bank configurations.

The speedup mainly comes from the chaining of function units which can overlap executions and consecutive accesses to the main memory; this amortizes the cost of memory latency while greatly reducing instruction hazards. Although MicroBlaze is a 5-stage pipelined RISC processor, Xilinx does not pipeline its floating-point instructions. Floating-

Table 4.1: Execution Times (us) for RGB2YIQ Conversion

Data Size	32	64	128	256	512
One MicroBlaze	32.4	64.7	129.4	258.7	517.2
1 Bank	4.99	8.45	15.37	29.2	35.25
2 Banks	3.43	5.23	8.83	16.03	30.43
4 Banks	2.65	3.58	5.46	9.2	16.68
8 Banks	2.35	2.84	3.83	5.82	9.79

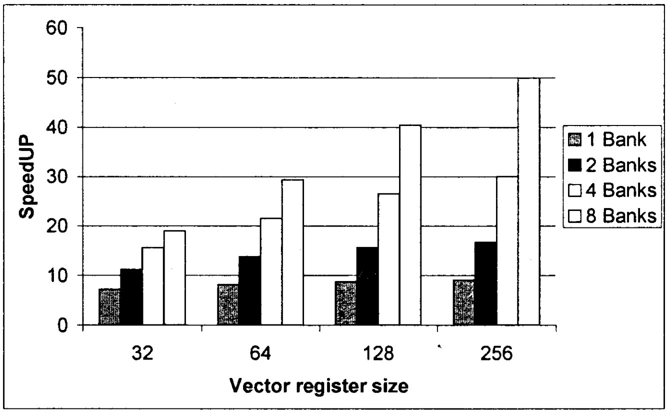

Figure 4.1: RGB2YIQ speedup of using the vector unit as compared to one MicroBlaze processor without this unit.

point multiplication or addition needs six clock cycles. Thus, for 256 additions and 256 multiplications, MicroBlaze needs $256 * 6 + 256 * 6$ clock cycles, while the vector unit under the 8-bank configuration only needs $(256 \div 8) + 2 * Latency$ clock cycles when considering chaining, where the latency is two in our implementation. This explains why our vector unit can yield a speedup higher than its number of banks, or even higher than 24, which is the number of operations that can be processed simultaneously at peak rate.

Increasing the number of banks can speedup the execution and the improvement becomes more significant when the vector register size increases. However, more banks means higher resource usage. If the speedup is "normalized" by dividing it with the slice usage ratio between the vector unit and one MicroBlaze, it will be as shown in Fig. 4.2. The 1-bank configuration provides good performance at low cost for small vector register size. The 4-bank configuration is better for larger vector registers. The 8-bank configuration is not preferable because of its significant slice usage increase from 14% for the 4-bank configuration to 23% as shown in Table 3.3.

4.2 Demodulation Algorithm in Communication Systems

This section deals with a testbench in the area of digital communication systems. A typical communication system is composed of a channel encoder, a modulator, a channel, a

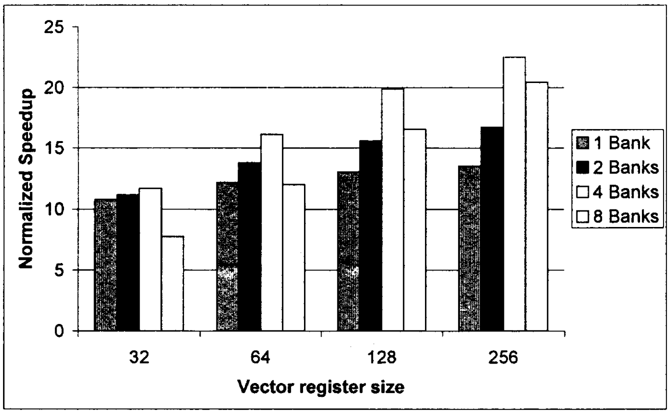

Figure 4.2: RGB2YIQ normalized speedup (using resource usage) of the vector unit as compared to one MicroBlaze processor.

demodulator and a channel decoder as shown in Fig. 4.3. The channel encoder adds data redundancy that can be used by the receiver for error correction. The binary sequences at the output of the encoder are passed to the modulator, which maps the binary bits to signal waveforms for transmission. The modulated bits are passed to the channel, which can be wireline or wireless. Whatever is the physical medium for information transmission, the transmitted signal can be corrupted by various types of noise. On the receiver side, the demodulator de-maps the potentially channel-corrupted transmitted waveform to a sequence of binary bits, which are passed to the channel decoder to reconstruct the original information bits.

Let us now demonstrate how our vector processor can be used to implement the demodulation part. On the receiver side, the received signal, y, is filtered by a finite impulse response (FIR) filter. The demodulator finds the symbol that has the shortest distance to

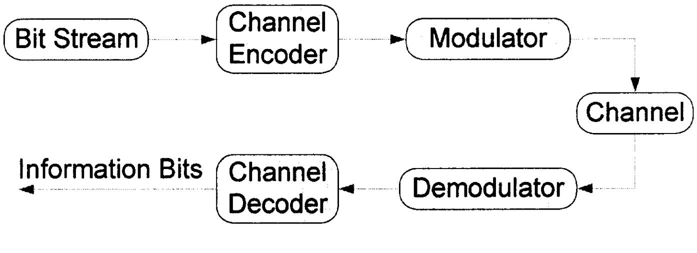

Figure 4.3: A typical digital communication system.

Table 4.2: Execution Times (ms) of the Distance Calculation Algorithm when Run on One MicroBlaze with the Vector Unit for 1000 Input Data.

Constellation points	32	64	128	256	512
1 Bank	1.873	2.26	3.457	6.531	12.677
2 Banks	1.75	1.95	2.35	3.6	6.8
4 Banks	1.741	1.818	2.026	2.442	3.74
8 Banks	1.761	1.821	1.931	2.152	2.593

the filtered signal, z, in the signal constellation:

$$\hat{s} = \arg\min_{s \in \mathcal{S}} |z - s|^2, \tag{4.2}$$

where $|\cdot|$ denotes amplitude. $|\mathcal{S}|$ is the number of points in the constellation. The algorithm can be divided into three sequential parts: FIR, distance calculation and distance comparison. Among them, FIR is an essential sequential part to be run on MicroBlaze. Distance calculation and distance comparison can be parallelized between MicroBlazes. Furthermore, distance calculation can be assigned to the vector unit. This approach results in substantial overlap of the scalar and vector execution times. Table 4.2 shows the distance calculation time for 1000 input data when one MicroBlaze is run with the support of the vector unit and Fig. 4.4 shows the speedup over one MicroBlaze under different bank configurations of the vector processor. For a fair comparison, the speedup can be normalized by dividing the slice usage ratio between the vector unit and one MicroBlaze. Fig. 4.5 shows that considering the area consumption; the vector unit can still yield a speedup from 2.5 to 35. As expected, the 1-bank configuration is the most efficient for a small data size and the benefit of more banks becomes more obvious when the data size increases.

Take the example of a 3-tap FIR and 512 points in the constellation; when there are 1000 input data, FIR consumes $0.58ms$, distance comparison takes $102.36ms$ and distance calculation takes $174.17ms$. The behavior of the 3-tap FIR is shown in Eq. (4.3), where b_0,

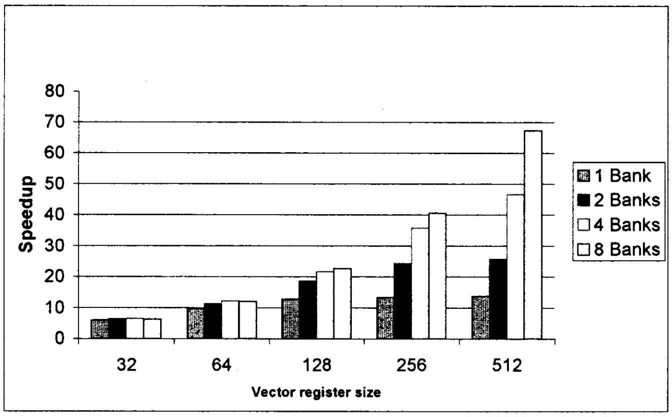

Figure 4.4: The speedup for distance calculation of the vector unit over one MicroBlaze.

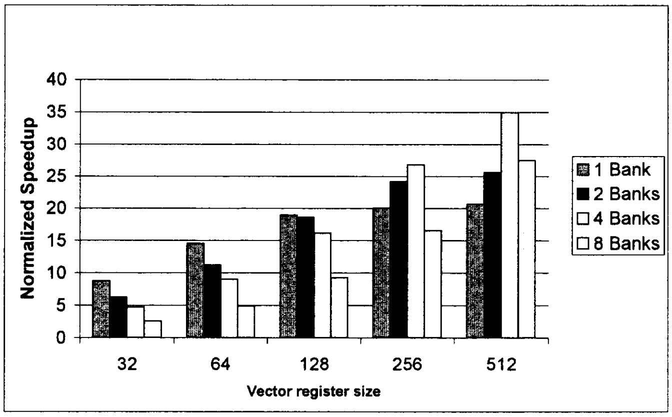

Figure 4.5: Speedup normalized by the slice usage ratio for distance calculation.

b_1 and b_2 are known coefficients.

$$z(n) = b_0 y(n) + b_1 y(n-1) + b_2 y(n-2) \tag{4.3}$$

Distance calculation contributes about 63% to the overall execution time. When this part is put on the vector unit, it is reduced to $12.68ms$, $6.8ms$, $3.74ms$ or $2.59ms$ based on the configuration. Since the time on the vector unit is much less than the scalar execution time and can be almost completely overlapped by the distance comparison time, we choose the 1-bank configuration for the vector unit with vector register size of 512 and one adder/multiplier. A higher degree bank configuration only induces more resource/power usage and cannot reduce the total execution time further.

Table 4.3: Execution Times (ms) of the Demodulation Algorithm (MB: MicroBlaze, DIS:Distance, VU:Vector Unit.)

	FIR	DIS calculation	DIS compare	Total
MB (100MHz)	0.58	174.17	102.36	277.11
One MB w/ VU (83.3MHz)	0.7	12.67	122.88	123.58
Two MBs w/ shared VU (83.3MHz)	0.7	12.7	62.4	63.1

Table 4.3 shows the whole execution time for demodulation with 1000 input data. The first line shows execution times on one MicroBlaze. The second line shows execution times when distance calculation is run on the vector unit while FIR and distance comparison are run on a MicroBlaze. With two scalar processors, the master-slave mode is used where one MicroBlaze controls communications with the other. Task division is carried out at static time and the programs are preloaded into each MicroBlaze. For demodulation, the programs for each MicroBlaze can run independently and need not communicate during execution. So the only overhead induced by the multiple scalar processors is the communication time to start and end the program, and probably delays caused by collisions on the shared data memory bus. But the memory access time in this algorithm is much smaller than the computation time and conflicts seldom happen; also, the communication overhead is negligible compared to the large calculation time. So the execution time for distance comparison is reduced almost by half for two MicroBlazes. The vector unit receives vector instructions from both MicroBlazes for distance calculation. The execution time is shown in the third line. More scalar processors could reduce further the execution time. For our proof-of-concept work, two scalar processors with a shared vector unit suffice.

If the same amount of slices with this MCwSV system (two MicroBlazes and a shared vector unit with the 1-bank configuration) are used to build a multicore scalar processor system, then 2.67 scalar "MicroBlazes" could fit. "Ideally", the whole execution time can then be reduced to $(174.17 + 102.36) \div (2.67) + 0.58 = 104.1ms$, which needs $(104.1 - 63.1)/63.1$ or 65% more time than the MCwSV system with the same slice usage.

4.3 Discrete Cosine Transform

The Discrete Cosine Transform (DCT) converts a spatial domain waveform into one in the frequency domain. It has been widely used in signal and image processing. We run one-dimensional DCT (1D DCT) on MCwSV for performance analysis. We show here how the ratio between vector and scalar execution times affects the MCwSV speedup. The 1D DCT conversion is given by:

$$S(u) = \sqrt{\frac{2}{N}} C_u \sum_{x=0}^{N-1} s(x) \cos \frac{(2x+1)u\pi}{2N} \tag{4.4}$$

where $C_u = \sqrt{1/2}$ for $u = 0$ and $C_u = 1$ otherwise; $u = 0, 1, 2..., or\ N - 1$, and N represents the size of the population for the input signal $s(\cdot)$. The basis triangular vector $\cos \frac{(2x+1)u\pi}{2N}$, which contains at most $2N$ values, can be stored in the on-chip memory for high efficiency.

Here we focus on accumulation and stride load, two kinds of new operations appearing in this application. Adder units organized in a binary tree can ensure fast accumulation. But this scheme requires additional hardware and new instructions. In our implementation, we use existing shift and addition instructions in the vector unit as well as the scalar processor to do accumulation. Thus, no more hardware resources or instructions are needed for this operation. The vector unit keeps shifting half of the elements in the vector register into another one, halving each time the vector length register (VLR) value and then applying vector addition until VLR contains eight. Then the scalar processor takes over to carry out the remaining accumulations sequentially. The number eight is chosen because of limitations in the shift instruction. For resource efficiency and implementation simplicity, the shared vector unit does not contain any switches between the vector register file and the function units; it does not include any switches between banks in the vector register file

Table 4.4: Vector and Scalar Execution Times and Their Ratio for the DCT Algorithm

Data Size		32	64	128	256	512
1 Bank (ms)	Vector	0.051	0.184	0.673	2.544	9.858
	Scalar time	0.079	0.169	0.361	0.768	1.629
	Ratio	0.644	1.091	1.864	3.312	6.053
2 Banks (ms)	Vector	0.067	0.24	0.902	3.488	13.568
	Scalar	0.082	0.176	0.376	0.8	1.696
	Ratio	0.82	1.364	2.4	4.36	8
4 Banks (ms)	Vector	0.054	0.188	0.595	2.56	9.854
	Scalar	0.085	0.183	0.391	0.831	1.762
	Ratio	0.634	1.027	1.523	3.08	5.592
8 Banks (ms)	Vector	0.044	0.143	0.500	1.833	6.871
	Scalar	0.090	0.194	0.415	0.883	1.871
	Ratio	0.483	0.736	1.204	2.076	3.672

either. So the shift instruction only supports shift sizes which are multiples of the number of banks. A stride load is needed to fetch the triangular vector from the memory into the vector register file. The stride is $2u$ and we will see later on how it affects the performance of the multi-bank MCwSV system.

Table 4.4 shows the vector and scalar execution times, and their ratio. The scalar time on the MicroBlaze, includes loop control, vector instruction preparation and transmission, eight floating-point additions, coefficient multiplication and result storage. The vector time is for the shared vector unit. We will see later on that, when the ratio is less than one, which means that the vector unit has some idle time when the scalar processor is busy, the DCT algorithm can yield higher speedup when run on two scalar processors with a shared vector unit. Since almost all of the work on the scalar processor and the vector unit can be run simultaneously, their execution times can be overlapped substantially.

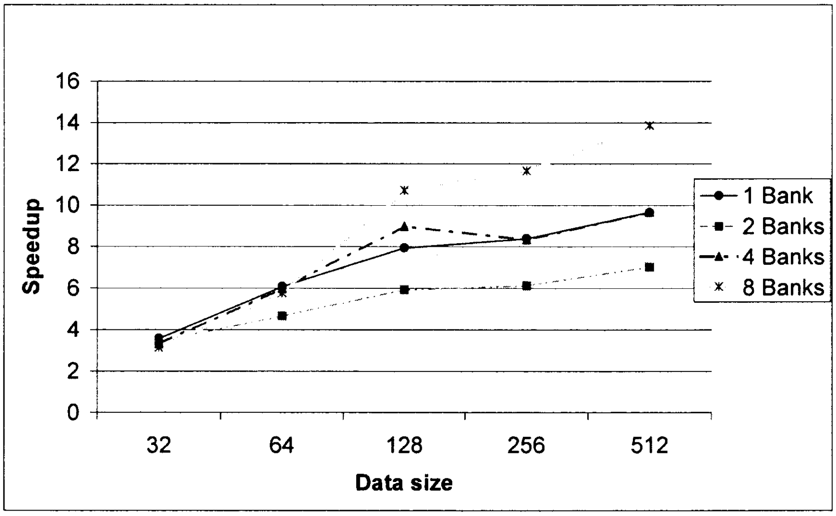

Figure 4.6: Speedup of including a vector unit in a MicroBlaze processor for the DCT algorithm.

Fig. 4.6 shows a large speedup between 3 and 14 when including a vector unit in MicroBlaze. The 2-bank configuration yields the smallest speedup whereas the 4-bank configuration cannot provide better performance than the 1-bank configuration for most data array sizes. This is because the interconnection structure between the memory and the register file induces a delay of three clock cycles for each step of indexed load/store for the multi-bank configuration. There is no such structure for the 1-bank configuration, and thus no such overhead. Also, the 2-bank configuration cannot load two elements in one step using a stride because the stride is always a multiple of two. So the load requires $3 * (array\ size)$ clock cycles, which contributes significantly to the execution time and neutralizes the benefit of multiple function units. The case for the 4-bank configuration is better, but still half of the stride load instructions can only retrieve one element at a time while the other half can only get two per cycle. So, it cannot provide better performance than the 1-bank configuration for most data array sizes. This situation is much better when there are eight banks in the vector unit since only 25% of the stride load instructions get one element per cycle; another 25% can retrieve two and the other 50% can retrieve four.

In this case, the benefit due to the multi-bank structure is more prominent, thus yielding higher speedup than the 1-bank configuration.

For those cases in Table 4.4 with a ratio less than one, implementing the DCT algorithm on two scalar processors with a shared vector unit will further improve the execution time. Fig. 4.7 shows the relevant speedups.

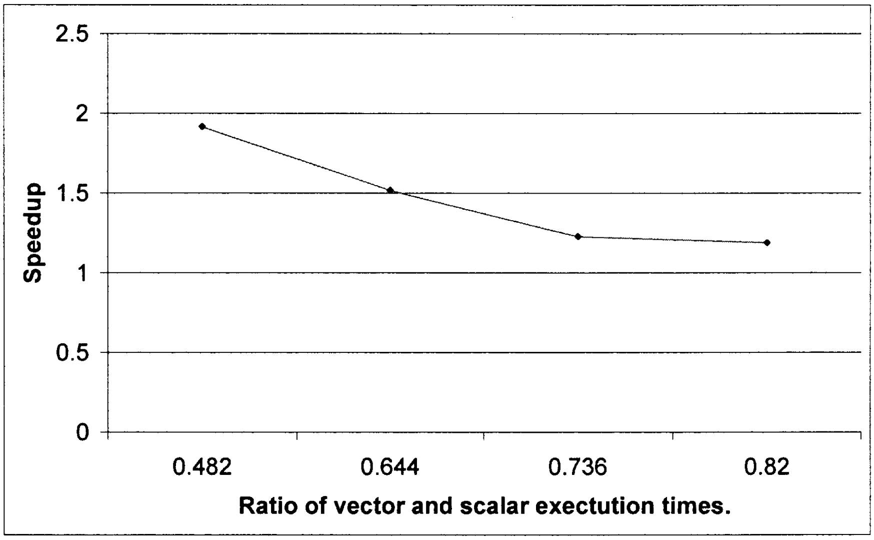

Figure 4.7: Speedup of two MicroBlazes versus one MicroBlaze, both systems with a vector unit, as a function of the ratio of vector and scalar execution times.

CHAPTER 5

POWER CONSUMPTION CHARACTERIZATION AND RESOURCE MANAGEMENT FOR MCWSV SYSTEMS

Dramatic increases in logic density, speed and abundant IP blocks have made FPGA platforms appealing alternatives to DSPs or ASICs for high-performance SOC designs. FPGAs have the inherent advantage of reconfiguration flexibility to fit various application requirements. But on the other hand, field programmability requires additional resources and increased power consumption [69]. Although recent FPGA chips, which incorporate millions of gates with abundant on-chip block memories and IP cores, can provide enough resources for most SOC designs, energy consumption still remains a critical issue. And as FPGA manufacturing reaches the nanometer scale and designs require higher operating frequency, the power dissipation problem becomes of higher concern.

In embedded system design, energy consumption is a key system evaluation metric in addition to system throughput, since embedded devices typically operate in power constrained environments. Current FPGAs require innovative methods to deal with the power consumption problem. There are two major power consumption types: static power and dynamic power; each one poses a unique challenge when reaching nanometer manufacturing processes. Static power consumption occurs as a result of current leakage in transistors. For FPGAs manufactured with a $0.35\mu m$ technology ten years ago, the static power consumption was negligible compared to the dynamic power. But as transistors get smaller and chip voltages get lower, the leakage current increases and, nowadays, static power is threatening to eclipse dynamic power [70]. Since the introduction of the Virtex-4 series, Xilinx has proposed a new manufacturing process called "triple oxide technology" to solve the static power problem. This method uses three gate-oxide thicknesses for transistors running at different speeds. As a result, Virtex-4 FPGAs consume 50% less static power than their predecessors, the 130nm Virtex-II Pro FPGAs. The Virtex-5 series also uses this method to control static power consumption. Other approaches to control the static power include power gating of unused FPGA logic blocks [71], dual-Vdd FPGA logic blocks [72, 73], and Vdd-programmable FPGA routing [74].

The well-known formula $P = CV^2f$ for dynamic power calculation shows that dynamic power benefits from a decrease in the transistor size, which means decreased capacity and lower core voltage with a smaller process. Thus, the rate of increase in dynamic power drops [70]. But still, the increase in frequency requires more dynamic power consumption. One major improvement for dynamic power control is to use hard IP cores. When embedded functions are implemented using dedicated hardware instead of configurable logic blocks and programmable interconnects, many fewer transistors are needed and thus much less power is consumed [70]. Actually, the resource and power usage of programmable interconnects play major roles in current FPGA designs at nanometer technologies [69]. In our MCwSV design, embedded function units are employed whenever possible for efficient power and resource usage. In Chapter 3, we listed the resource usage of the MCwSV system. In this Chapter, we build a resource/power usage model and present an architecture template to guide resource/power-efficient implementations for MCwSV systems targeting at given applications.

5.1 Resource/Power Modeling for the MCwSV System

In this section, we present characterization and modeling techniques for resource/power consumption of function units, block memories, scalar processors and the vector unit in the MCwSV system. This model will be used in the next section to build resource/power-efficient MCwSV systems for given applications. XPower, a low-level energy estimation tool provided by Xilinx, is used to analyze the power consumption. The typical estimation process is as follows: after the design is synthesized, it is placed and routed for the target FPGA chip and the native circuit description file (.ncd file) is generated. This .ncd file records various parameters of the circuit implementation, such as device utilization, routing structure, channel capacitance and embedded block usage. On the software side, the design is simulated at the transfer/gate level and the simulation record files (.vcd files) which record the switching activities of all the logic and wires on the FPGA device are generated. Then, XPower can be used to estimate the power consumption based on the information of circuit implementation (.ncd file) and signal activities (.vcd file). While this process can provide accurate power estimation, it is extremely time consuming and the simulation record files require huge storage space [75]. Also, the result is only accurate for the specific algorithm and there may be orders of magnitude difference for other applications. So in our implementation, instead of using the simulation record files, we give the calculated signal activities to XPower and generate power consumption estimates of the various components.

FPGA power consumption has three components related to different parts of the FPGA chip: core power, auxiliary power and input/output power. The later two are related to the I/O standards being implemented and are expected to remain about the same for different circuit implementations and softwares if they have the same input/output pin assignment and activities. So, here we put emphasis on core power, which is the power consumed by the embedded logic and signals. Different implementations will greatly affect the core dynamic power consumption [76, 77].

5.1.1 Dynamic Power Consumption of Floating-Point Arithmetic Units

The floating-point arithmetic units in the MCwSV system are taken from the Xilinx math library. It provides single/double-precision and other fraction/exponent combinations for floating-point adder/subtractor, multiplier, divider, comparator and square-root implementations. Among them, the multiplier and adder/subtractor can be implemented either by DSP embedded blocks and logic resources, or by logic resources only. Other arithmetic units are implemented by logic resources only. So here we just give the resource usage and power consumption for multipliers and adders/subtractors to compare various implementation methods.

The Xilinx math library provides four different ways to implement the floating-point multiplier by using logic resources only, one DSP with logic resources, two DSPs with logic resources, and three DSPs with logic resources. The implementation method with more DSP blocks does not imply less logic resource usage, but aims to provide higher maximum frequency which is achieved at largest latency. The logic only and the 2-DSP implementations provide user defined latency from 0 to 8 clock cycles. The 3-DSP implementation provides user defined latency from 0 to 9. The 1-DSP implementation, however, can only provide a fixed latency of 8 clock cycles. The maximum frequency provided by the 3-DSP, 2-DSP, 1-DSP and logic only implementations is $450MHz$, $406MHz$, $375MHz$ and $357MHz$, respectively, which are much higher than our targeted system frequency of $100MHz$. So instead of using the maximum latency cycles to get the highest frequency, we can use the configuration with smaller latency which can provide the result quicker and still satisfies the time requirement. Another advantage of the smaller latency cycles is less resource usage and less power consumption. Larger latency implies deeper pipeline and thus more resource usage. Thus in our implementation, we choose the smallest latency cycles which can satisfy the $100MHz$ requirement, which is two clock cycles. Further decrease in the latency cycles will cause a timing problem. Because the 1-DSP implementation can only provide a fixed number of latency cycles, we will not consider it. Thus in Tab.5.1, we only show the resource and power consumption for the logic only, 2-DSP and 3-DSP multiplier implementations with all of them having a latency of two clock cycles. For the floating-point adder/subtratror, the Xilinx math library only provides two kinds of implementations: logic resources only, and 2-DSP with logic resources. Similarly, the latency of two clock cycles is chosen; the resource and power consumption are also shown in Tab.5.1.

As shown in Tab.5.1, the multiplier with the 2-DSP implementation uses 44.5% and 30.9% less power compared to its logic only and 3-DSP implementation alternatives. So the

Table 5.1: Resource Usage and Dynamic Power Consumption of IEEE Single-Precision FPUs on the Virtex-5 FPGA

Operation	Type	DSPs	LUTs	FFs	Slices	Power (mw)
Multiply	DSP48E (3-DSP)	3	63	77	116	24.52
	DSP48E (2-DSP)	2	86	83	106	18.73
	logic	0	663	142	663	33.75
Add/Subtract	DSP48E (2-DSP)	2	251	85	279	27.06
	Logic	0	402	72	402	27.56

2-DSP implementation is chosen for the single-precision floating-point multipliers in our MCwSV system. For the single-precision floating-point adders, the two implementation methods consume almost the same power. However, we still choose the DSP configuration due to less slice usage in order to leave logic resources available to other logic and routing units.

Double-precision FPUs consume much more resource and power comparing to their single-precision versions. A double-precision multiplier, based on the latency configuration, can consume a dynamic power from $100mw$ to $300mw$.

5.1.2 Dynamic Power Consumption of Dual-Port Block Memories

There are 48 36-Kbit dual-port block memories in the target FPGA chip and they are used as local instruction/data memory, shared data memories and FIFOs. Each port has an enable bit to power on the block memory. When both ports are disabled, only clock power is consumed. So the block memory should be disabled for power efficiency when it is not in use. Fig. 5.1 shows the dynamic power consumption without including the clock power for one 36-Kbit embedded memory block at different enable rates. The dual-port line shows the power consumption when both ports are used assuming the same enable rates; the

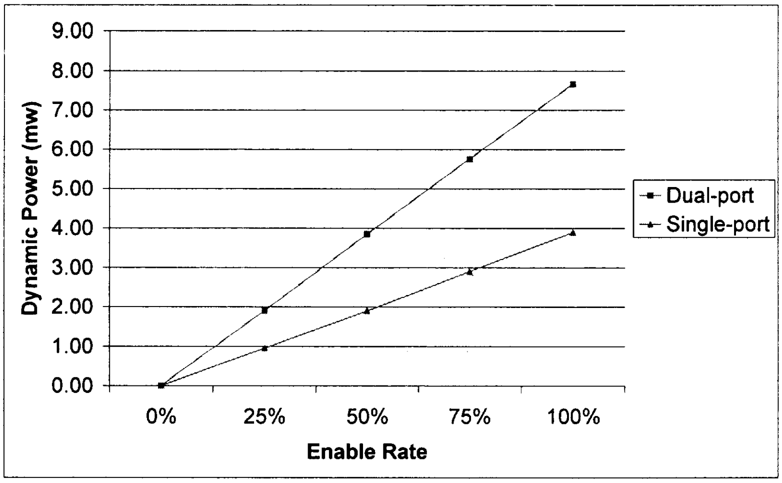

Figure 5.1: Dynamic power consumption of one memory block VS the enable rate.

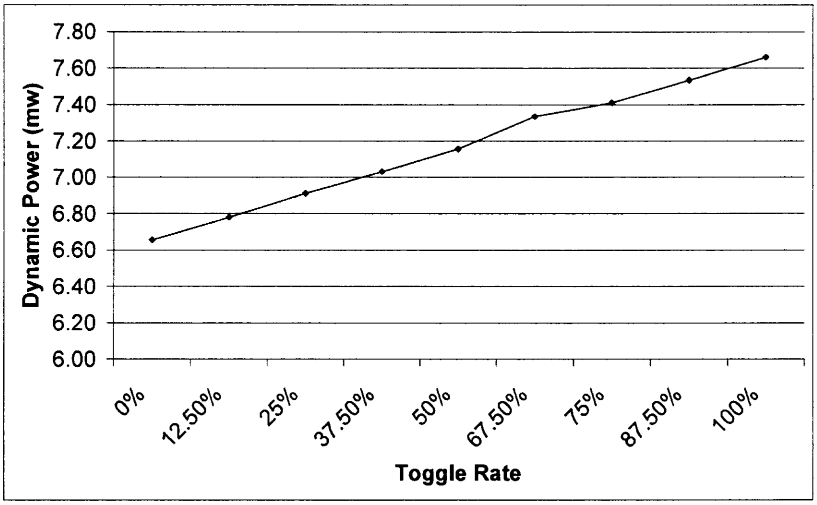

Figure 5.2: Dynamic power consumption of the dual-port memory VS the toggle rate.

single-port line shows the power consumption when only one port is used and the other one is always off. The toggle rate is assumed to be 100% in this figure.

Fig. 5.2 shows the power consumption for the dual-port block memory at different toggle rates assuming that the enable rate is 100%. One 36K-bit memory block can also be configured as two independent 18K-bit memory blocks. Each 18k-bit memory block consumes half the power of the 36K-bit memory with the same frequency, enable rate and toggle rate.

Table 5.2: Resource Usage and Power Consumption of a MicroBlaze w/wo FPU Support

	Resources					Power (mw)	
	DSPs	BRAMs	LUTs	FFs	Slices	Dynamic	Static
wo FPU	3	2	1085 (3%)	958 (3%)	1607 (5%)	93.96	11.37
w FPU	7	2	1885 (6%)	1440 (5%)	2508 (8%)	134	18.19

5.1.3 Power Consumption of MicroBlaze w/wo a Floating-Point Unit

MicroBlaze has some optional features and can be configured when needed. Among these features, the FPU is resource/power consuming. So here we study the two cases of resource/power consumption: MicroBlaze with or without FPU support. The configurable FPU for MicroBlaze includes a divider, a multiplier, an adder/subtractor, a comparator and a shifter. MicroBlaze can only be configured to include all or none of these function units. The resource usage and dynamic power with or without the FPU configuration are shown in Table 5.2. In this table, we use 12.5% as the default toggle rate for asynchronous signals. The default toggle rate of 12.5% is widely accepted in power analysis and is derived by analyzing over 100 proprietary benchmark designs [69] . From Table 5.2, it can be seen that a MicroBlaze with FPU support requires 56% more resources and 43% more power.

The dynamic power consumption of a MicroBlaze w/wo an FPU at other toggle rates is shown in Fig. 5.3.

5.1.4 Power Consumption of the Vector Unit for Different Configurations

Table 5.3 shows the resource usage and power consumption of the vector unit in the MCwSV system under different configurations without enable signals for block memories. Compared to the $134mw$ consumed by one MicroBlaze with FPU support, our vector unit for the 1-bank configuration needs quite more power even with similar resource usage. This is mainly because the multi-port block memories consume lots of power. As described in Chapter 3, our eight-port 16KB vector register is implemented by four 36Kb

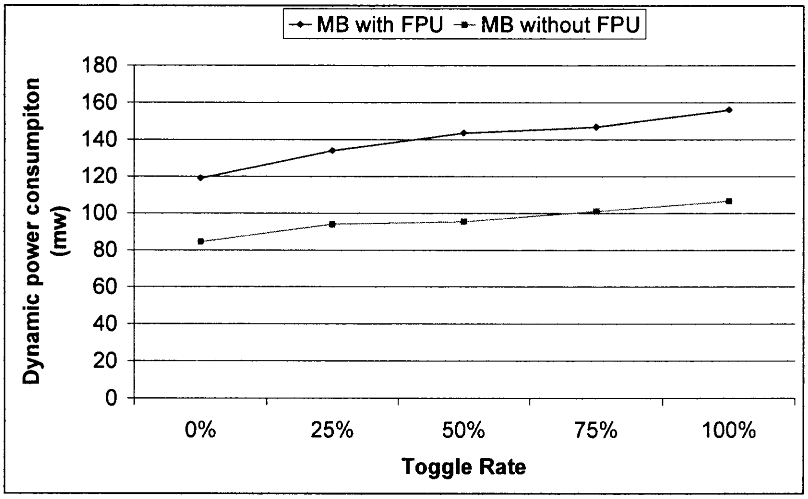

Figure 5.3: Dynamic power consumption of a MicroBlaze w/wo an FPU at different toggle rates.

Table 5.3: Resource Usage and Power Consumption of the Vector Unit Without Enable Control of Block Memories

		BRAMs		Resources	Power (mw)	
	DSPs	18KB	36KB	slices	Dynamic	Static
1 Bank	4	3	4	2278 (7%)	240.98	15.92
2 Banks	8	3	4	3071 (10%)	343.63	22.74
4 Banks	16	3	4	4275 (14%)	534.25	31.83
8 Banks	32	11	0	6873(23%)	886.14	52.29

block memories using time-multiplexing and its operating frequency is four times the system frequency. Similarly, the six-port scalar registers are implemented by an 18KB block memory with an operating frequency which is three times the system frequency. Take the example of the vector register file; if all the block memories are enabled at a 12.5% toggle rate, then the block memories will require $6.83 * 4 * 4 = 109.28mw$. 6.83 is the power consumption of the 36Kb block memory at 100MHz with a 12.5% toggle rate. There are four such blocks in the system and they are all running at $400MHz$, so the total power consump-

Table 5.4: Dynamic Power Consumption of the BRAMs and the VRF

	Enable Rate	Power(mw) BRAM	Power(mw) VRF
1 Bank	12.5%	13.66	94.66
2 Banks	25%	27.32	110.3
4 Banks	50%	54.64	192.86
8 Banks	100%	109.28	324.13

tion is calculated as above. This number is even larger than the power consumption of the whole system of a MicroBlaze without FPU support. So it is necessary to control the power consumption of the block RAMs for the power efficiency of the entire system. The enable signals of the BRAM should be set accordingly for less power consumption. Considering that half of the BRAM of the 36Kb configuration can be disabled if it is constructed by two 18Kb BRAM blocks, eight 18Kb BRAMs will be used for the vector register file instead of four 36Kb BRAMs for power efficiency. For the worst case, all three function units in the vector processing unit are run simultaneously and all the eight ports are occupied. Then the enable rate for the BRAMs and the dynamic power consumption of VRF for each bank configuration are shown in Table 5.4. A 12.5% toggle rate is assumed here. When the configuration in Table 5.4 is used in the vector unit, its power consumption can be reduced from $240.98mw$, $343.63mw$, $534.25mw$ to $145.36mw$, $261.67mw$ and $479.61mw$, respectively, for the 1-, 2- and 4-bank configuration. Table 5.5 shows the resource usage and power consumption of the vector unit with enable control for the block memories.

Fig. 5.4 shows the percentage of power consumption and the slice usage of each component in the vector unit. It can be seen that the multi-port vector register file and scalar register file contribute significantly to the power consumption and resource usage in the vector unit. So area and power efficient multi-port register files are critical in the vector processing unit design. But limited by the current FPGA resources, we cannot have a better implementation method.

Table 5.5: Resource Usage and Power Consumption of the Vector Unit with Enable Control of the Block Memories

		BRAMs	Resources			Power (mw)	
	DSPs	18KB	LUTs	Flip-Flops	slices	Dynamic	Static
1 Bank	4	11	1887 (6%)	929 (3%)	2278 (7%)	145.36	15.92
2 Banks	8	11	2482 (8%)	1259(4%)	3071 (10%)	251.67	22.74
4 Banks	16	11	3290 (11%)	1912(6%)	4275 (14%)	479.61	31.83
8 Banks	32	11	5199 (18%)	3238(11%)	6873(23%)	886.14	52.29

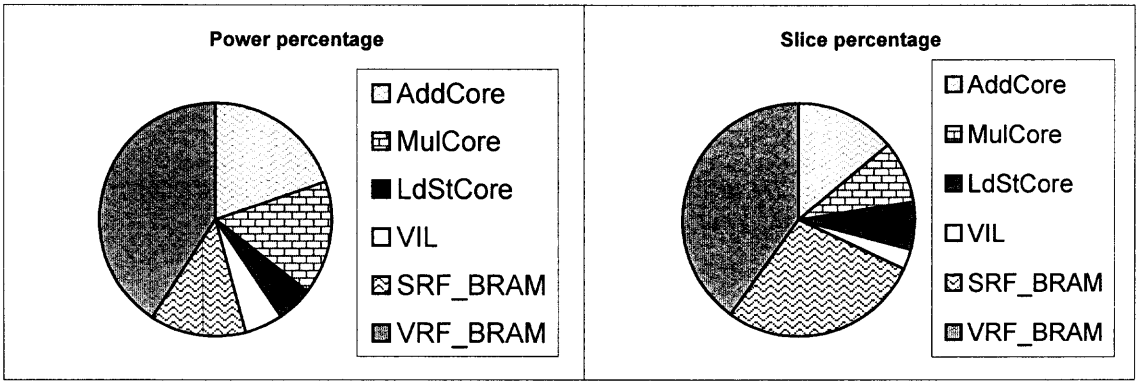

Figure 5.4: Percentage of power consumption and slice usage for the vector unit with the 1-bank configuration.

5.2 Power/Resource-Efficient MCwSV Systems for Various Applications

In Chapter 3, an MCwSV system with two MicroBlaze processors and a shared vector processing unit was designed and implemented on the target FPGA. Its efficiency was shown in Chapter 4 by running three testbenches on it. However, this organization was not optimized for various applications with different percentages of vectorization. But on the other hand, the MCwSV system was designed for easy scalability to potentially incorporate different numbers of scalar/vector units and various function units. Also, the flexibility inherent to the FPGA can aid the task of matching different applications. These benefits can be taken into account to create an optimized MCwSV system for each specific requirement. For embedded system design, the area, power and system throughput are three metrics for overall system evaluation; an optimized system should be based on a good tradeoff among them. In this section, we build an architecture framework for performance and resource management to guide configuration for the MCwSV system targeting at different applications. The target MCwSV system architecture has the following key features:

- A vector unit which can be configured to contain various types and numbers of function units, and different number-of-vector-registers/vector-size pairs.
- Multiple MicroBlaze scalar processors with or without FPU support.
- These multiple scalar processors are organized in the master-slave mode where one scalar processor controls each time communications with all the others.
- These multiple scalar processors can share the vector unit by sending instructions to the latter via standard bus interfaces.
- These multiple scalar processors and the vector unit share the data in the shared data memory via a memory access arbitrator.
- The MicroBlaze processors and the vector unit use different clock networks in the FPGA; thus, they can be disabled individually for power efficiency.

The architectural design is based on the analysis of scalar/vector execution times and resource/power consumptions of various components. This process is driven by analyzing the target algorithms. At the end, the derived architecture template will give the configuration of the MCwSV system with the type and number for FPUs in the shared vector unit, and the number of scalar processors and their configuration. The primary goal is to find a near-optimal configuration based on cost-performance tradeoffs for each given application. Fig. 5.5 provides an overview of the procedure for architecture template creation. In this template, $MBNum$ indicates the number of MicroBlaze processors, which are used as scalar processors in the system. $VBankNum$ indicates the number of banks in the vector

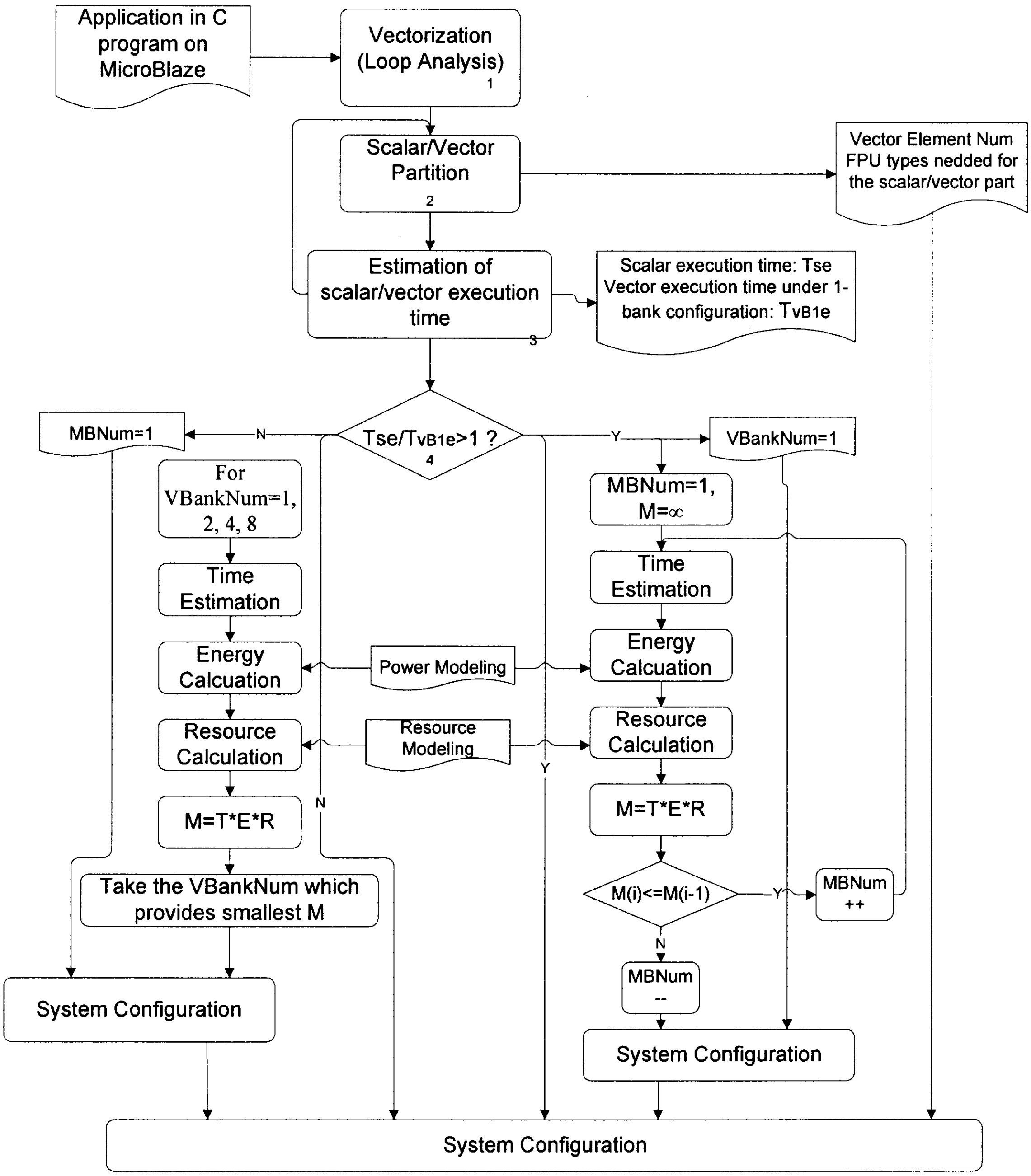

Figure 5.5: Overview of the procedure for architecture template creation.

unit. If the product $M = T * E * R$ of execution time, energy consumption and resources is used as the design metric here, then the architecture which can provide the smallest M is considered to be the optimized design.

For computation-intensive algorithms with floating-point calculations, which are the primary target of our MCwSV systems, generally a few blocks of code consume most of the overall execution time [78]. These blocks are usually represented by loops. The vectorization process for the architecture template is actually based on loop analysis of these blocks, which is the first task in the process. Vectorization are performed on the loops which has no flow dependence and cross-iteration dependence. Temporary storage space and loop splitting are used when necessary to remove dependence in some loops and they make partial vectorization possible. Then scalar and vector partitions are processed and their execution times on the scalar processor and the vector unit for the 1-bank configuration are estimated. We assume that the scalar and vector codes run exclusively on the scalar and vector units and the scalar calculation can be divided into several parts and run independently on several MicroBlaze. Examples are shown in Section 5.2.2. Then if the scalar execution time is larger than the vector execution time, the number of vector banks is set to one and the number of MicroBlazes is increased by one at each step. The design metric M is then calculated for each configuration based on the estimated scalar/vector execution time and the resource/power modeling built in the previous section. The scalar/vector time estimation will be explained in the following subsection. This process stops when increasing the number of MicroBlazes results in an increase of M. The vector bank number is set to one because reducing the vector execution time will not reduce the total execution time in this situation. Then more banks in the vector unit only means more resource used and higher power consumption. Increasing the number of scalar processors can reduce the scalar execution time proportionally for some applications as shown in Section 5.2.2 and thus reduce the total execution time. But more MicroBlazes definitely mean more resources to be used, so a tradeoff should be made between resources used and execution time, which in our implementation is interpreted as finding the smallest M.

In the other situation where the scalar execution time is less than the vector execution time, the number of MicroBlazes is set to one and M is calculated for various bank configurations in the vector unit. The number of banks that produces the smallest M is chosen for the vector unit.

5.2.1 Scalar/Vector Execution Time Estimation

Here we give a simplified estimation model for the scalar/vector execution times. For scalar execution time estimation, we need to emphasize here that although MicroBlaze is a 5-stage pipelined RISC processor, it does not pipeline the floating-point instructions. Floating-point multiplication or addition needs six clock cycles, floating-point comparison needs three clock cycles and floating-point division needs 30 clock cycles. To simplify the estimation model, we use the clock latency recorded in the MicroBlaze reference manual [79] for the scalar instructions, without considering hazards. This method, though not very accurate, is enough for our proof-of-concept work. The experimental data shows that when comparing with the real execution time, this estimation module has an error percentage from -20% to 0. The symbol "$-$" indicates that the estimation module always underestimates the execution time, which matches out assumption of not considering the hazards. Introducing the average stall cycles in the estimation module will give more accurate results and can be studied in future work.

For loops which can be vectorized, the optimized program for the vector unit will be written by the programmer manually and the instructions will be read into a simplified model for vector execution time estimation. Considering that the vector unit in the MCwSV system supports chaining and the code is optimized by hand, most execution times on different FPUs can be overlapped. So we only take the largest execution time of one FPU. But the index and stride loads/stores are exceptions because their behavior cannot be decided till run time and they do not support chaining either. So their execution times are calculated separately. Take the example of three function units in the system: floating-point multiplier, floating-point adder and load/store unit; the vector execution time is estimated as shown in Fig. 5.6. In this figure, two is the latency of the floating-point multiplier and adder, four is the execution time of a scalar instruction executed on an FPU control part, and three is the latency of each loadVI/storeVI and loadVS/storeVS induced by the crossbar between the multi-bank memory blocks and the register file blocks. $AVGNum$ is the average number

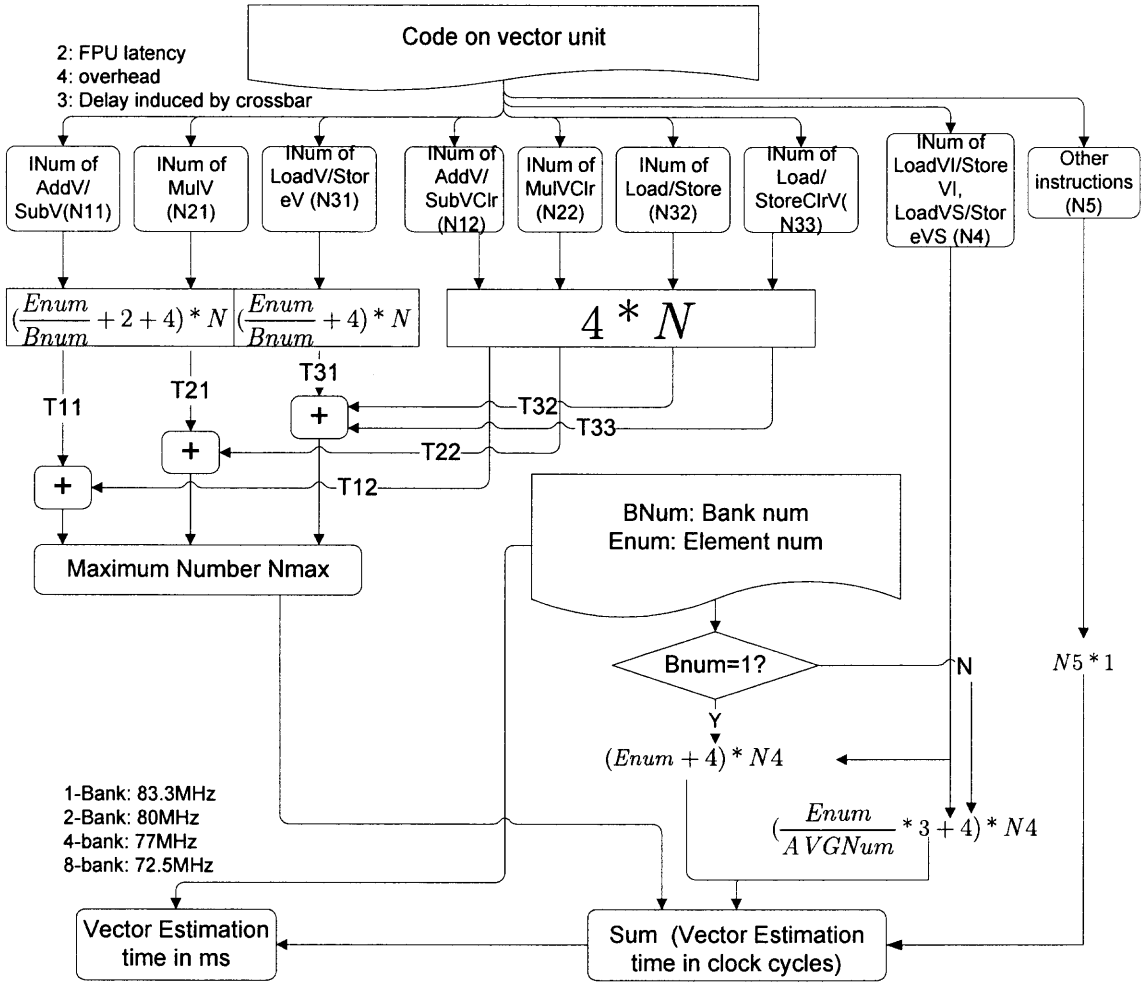
Code on vector unit
2: FPU latency
4: overhead
3: Delay induced by crossbar
INum of AddV/ SubV(N11)
INum of MulV (N21)
INum of LoadV/StoreV (N31)
INum of AddV/ SubVClr (N12)
INum of MulVClr (N22)
INum of Load/Store (N32)
INum of Load/ StoreClrV(N33)
INum of LoadVI/Store VI, LoadVS/StoreVS (N4)
Other instructions (N5)
$(\frac{Enum}{Bnum} + 2 + 4) * N$
$(\frac{Enum}{Bnum} + 4) * N$
$4 * N$
T11
T21
T31
T32
T33
T22
T12
+
Maximum Number Nmax
BNum: Bank num
Enum: Element num
Bnum=1?
Y
N
$(Enum + 4) * N4$
$(\frac{Enum}{AVGNum} * 3 + 4) * N4$
$N5 * 1$
1-Bank: 83.3MHz
2-Bank: 80MHz
4-bank: 77MHz
8-bank: 72.5MHz
Vector Estimation time in ms
Sum (Vector Estimation time in clock cycles)

Figure 5.6: Estimation model for the vector execution time.

of loads/stores at one clock cycle for the loadVI/storeVI and loadVS/StoreVS instructions. $AVGNum$ is calculated assuming the uniform distribution for the number of loads/stores throughout the bank. If $BNum$ is the number of banks in the vector unit, $AVGNum$ can be calculated by the following equation.

$$AVGNum = \frac{1 + 2 + \cdots + BNum}{BNum} = \frac{BNum * (BNum + 1)}{2 * BNum} = \frac{BNum + 1}{2} \tag{5.1}$$

Similar to the scalar time estimator, when compared with the real execution time the vector time estimation model always underestimates the time because it does not consider hazards either. Take the example of the RGB2YIQ algorithm; its instruction profiling for the vector unit is shown in Fig. 5.7. The percentage of error between the real and estimated vector execution times is shown in Fig. 5.8. It can be seen that the error percentage for the vector estimation time increases with the number of banks and decreases with data size increases. The reason is that larger data sizes and smaller numbers of banks mean a higher percentage of FPU vector execution time in the overall execution time, which is estimated with higher accuracy because there are no hazards for executions in any given FPU and the hazards between different FPUs are negligible.

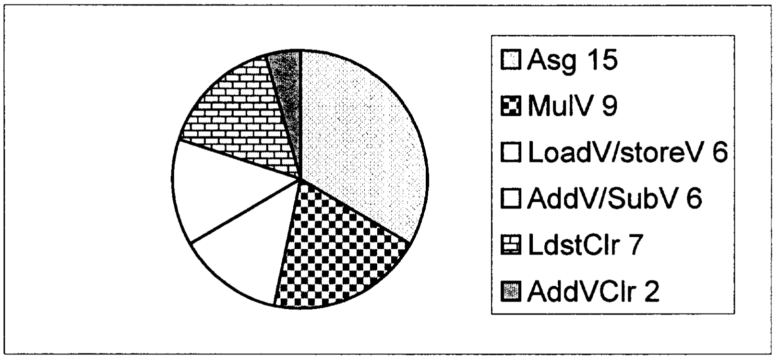

Figure 5.7: Instruction profiling for the RGB2YIQ algorithm on the vector unit.

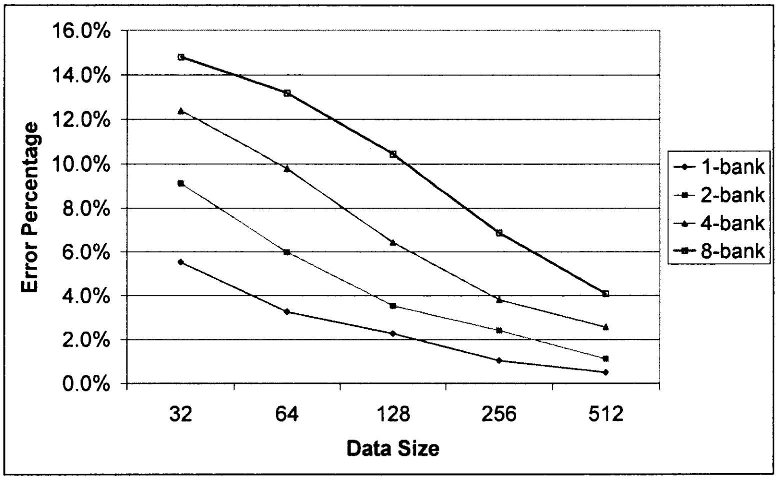

Figure 5.8: Error percentage of the vector time estimation model for the RGB2YIQ algorithm.

5.2.2 Application-Specific MCwSV (AS-MCwSV) System for Power/Resource Efficiency

In this section, we produce optimized configurations for two applications using the aforementioned architecture template process. They are the RGB2YIQ algorithm and the demodulation algorithm, which were used as testbenches in Chapter 4. These two algorithms represent two situations for architecture template creation, as shown in Fig. 5.5. RGB2YIQ conversion represents the case where the vector execution time is larger than the scalar execution time. In this case, the number of MicroBlazes is set to one and we need to find the optimized bank configuration for the vector unit. The demodulation algorithm represents the other case where the scalar execution time is larger than the vector execution time. In this situation, the number of banks in the vector unit is set to one and we need to find out how many scalar processors should be included in the MCwSV system.

5.2.2.1 AS-MCwSV System for RGB2YIQ Conversions. The main part in the RGB2-YIQ algorithm is a FOR loop without any flow dependencies and cross-iterations. It can be fully vectorized for the vector unit. The scalar processor is only used for loop control, preparing and transferring vector instructions. Its execution time is much less than the vector execution time. So for this algorithm, the AS-MCwSV system only needs one MicroBlaze without FPU support. The vector unit should have floating-point multipliers and adders/subtractors for the RGB2YIQ algorithm. We need to determine the bank configuration for the vector unit.

In our design, each MicroBlaze processor and the vector unit use different clock networks that can be turned off if the corresponding scalar processor(s) or vector unit are in the idle status. So the energy consumption is composed of the following four parts:

- Active power of the vector unit.
- Standby power of the vector unit.
- Active power of each MicroBlaze.

- Standby power of each MicroBlaze.

While the active power is composed of the dynamic power and the leakage power, the standby power is only the leakage power.

Since the estimation of the execution time was explained in the previous section, here we will proceed to the other steps shown in Fig. 5.5. For this algorithm, the execution time of MicroBlaze is small and can be assumed as totally overlapping the vector execution time. So the power consumption for this application is composed of the active/idle power of MicroBlaze and the active power of the vector unit. Take the example of 32 data items processed by the MCwSV system: the vector register size is set as 32 also to match the data size. For the 1-bank configuration, the estimated vector and scalar execution times are $3.93\mu s$ and $1.14\mu s$, respectively. The total energy consumption of $0.79\mu J$ is calculated as shown below using the power consumption model we built in the previous section.

$$\begin{aligned} E^{MB}_{active} &= E^{MB}_{dynamic} + E^{MB}_{leakage} = (P^{MB}_{dynamic} + P^{MB}_{leakage}) \times (MBExecutionTime) \\ &= (93.96 + 11.37) \times 1.14 = 120nJ = 0.12\mu J \end{aligned}$$

$$\begin{aligned} E^{MB}_{standby} &= P^{MB}_{leakage} \times (MBIdleTime) = (11.37) \times (3.93 - 1.14) \\ &= 31nJ = 0.03\mu J \end{aligned}$$

$$\begin{aligned} E^{VU}_{active} &= E^{VU}_{dynamic} + E^{VU}_{leakage} = (P^{VU}_{dynamic} + P^{VU}_{leakage}) \times (VUExecutionTime) \\ &= (145.36 + 15.92) \times 3.93 = 633nJ = 0.63\mu J \end{aligned}$$

$$E^{VU}_{standby} = E^{VU}_{leakage} = (P^{VU}_{leakage}) \times (VUIdleTime) = (15.92) \times 0 = 0\mu J$$

$$E_{Total} = E^{MB}_{active} + E^{MB}_{standby} + E^{VU}_{active} + E^{VU}_{standby} = 0.12 + 0.03 + 0.63 + 0 = 0.79\mu J$$

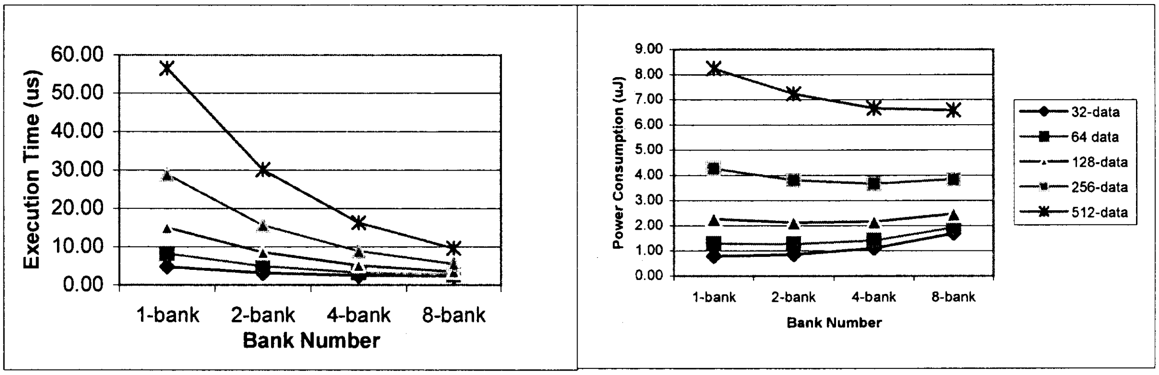

Figure 5.9: Execution time and power estimation for the MCwSV system for various data sizes and different bank configurations for the RGB2YIQ algorithm.

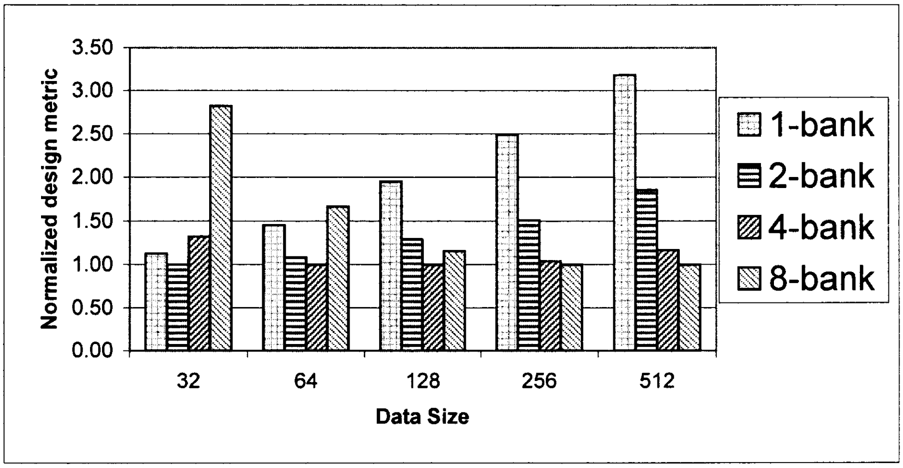

Figure 5.10: Normalized design metric (M) for various data sizes and different bank configurations.

If the same calculation is performed for other bank configurations and data sizes, Fig. 5.9 shows the estimated energy consumption based on power modeling and the estimated execution time according to the above equations.

It shows that as the data size increases, the benefit of large numbers of banks becomes more significant. The normalized design metric $M = Resource * Energy * Time$ is shown in Fig. 5.10. It can be seen from this figure that one MicroBlaze with the 8-bank configuration is the best choice for large data sizes greater than 256; for small data sizes, like 32, the 2-bank configuration is the most efficient solution. For other data sizes, the

4-bank configuration is the best choice. So for RGB2YIQ conversions, the AS-MSwSV system is one MicroBlaze processor without FPU and a vector unit with the 2-, 4-, or 8-bank configuration for various data sizes.

5.2.2.2 AS-MSwSV System for the Demodulation Algorithm. As described in Chapter 4, the algorithm is composed of three parts: FIR filtering, distance calculation and distance comparison. Among them, FIR is an essentially sequential part and has to be run on one MicroBlaze. The distance calculation and distance comparison can be divided into several parts to be run simultaneously on multiple processors. And the distance calculation part can be further vectorized. So in the MSwSV system, distance comparison will be run on multiple scalar processors and distance calculation will be run on the shared vector unit. Most of their execution times can be overlapped.

Assume that the system has a 3-tap FIR and 32 points in the signal constellation; then for 1000 input data, the scalar execution time for FIR and distance comparison is $0.58ms$ and $6.16ms$, respectively. The estimated vector execution time for distance calculation is $1.15ms$, $0.80ms$, $0.62ms$ and $0.55ms$ for the 1-, 2-, 4- and 8-bank configurations. In this case, the scalar execution time is much larger than the vector execution time and the number of banks in the vector unit is set to one as explained before. We need to find out how many scalar processors should be included in the AS-MCwSV system for the demodulation algorithm. Because all of the scalar processors need to deal with floating-point calculations, each MicroBlaze is configured to include FPU hardware support.

When multiple scalar processors are employed in the system, they are organized in the master-slave mode with one MicroBlaze controls each time communications with all the other processors. Task division is made at static time and the programs are preloaded onto each MicroBlaze before execution. For the demodulation algorithm, the distance comparison part can be divided to run on several MicroBlazes independently and these MicroBlazes need not communicate during execution. As explained in Chapter 4, the overhead induced

Table 5.6: Estimated Execution Time for the Demodulation Algorithm

MicroBlazes	1	2	3	4	5	6
FIR (ms)	0.58	0.58	0.58	0.58	0.58	0.58
Distance Comparison (ms)	6.16	3.09	2.07	1.56	1.27	1.03
Distance Calculation (ms)	1.15	1.15	1.15	1.15	1.15	1.15
Total (ms)	6.74	3.67	2.65	2.14	1.85	1.73

by several MicroBlazes is negligible. So the execution time for distance comparison is reduced almost proportionally with increases in the number of scalar processors. And the execution time for FIR and distance calculation remains the same.

The estimated execution time is shown in Table 5.6 based on the time estimation module. Because the distance comparison time can be overlapped with the distance calculation time, the total execution time is the sum of the larger of these two numbers and the FIR calculation time.

The power consumption is then calculated based on the estimated execution time and power modeling. The total power consumption is the sum of the power of each MicroBlaze in the system and the power of the shared vector unit. Since all MicroBlazes have the same configuration, we use $P^{MB}_{dynamic}$ and $P^{MB}_{leakage}$ to represent their dynamic power and leakage power. Take the example where there are two MicroBlazes in the system:

$$E^{MB_1}_{active} = (E^{MB_1}_{dynamic} + E^{MB_1}_{leakage}) = (P^{MB}_{dynamic} + P^{MB}_{leakage}) \times (MB_1 ExecutionTime)$$
$$= (134 + 18.19) \times (0.58 + 3.09) = 558\mu J = 0.56mJ$$

$$E^{MB_1}_{standy} = E^{MB_1}_{leakage} = P^{MB}_{leakage} \times (MB_1 IdleTime) = (18.19) \times (3.67 - 0.58 - 3.09)$$
$$= (18.19) \times (0) = 0mJ$$

$$E^{MB_2}_{active} = (E^{MB_2}_{dynamic} + E^{MB_2}_{leakage}) = (P^{MB}_{dynamic} + P^{MB}_{leakage}) \times (MB_2ExecutionTime)$$
$$= (134 + 18.19) \times (3.09) = 470\mu J = 0.47mJ$$

$$E^{MB_2}_{standy} = E^{MB_2}_{leakage} = P^{MB}_{leakage} \times (MB_2IdleTime) = (18.19) \times (3.67 - 3.09)$$
$$= (18.19) \times (0.58) = 10\mu J = 0.01mJ$$

$$E^{VU}_{active} = E^{VU}_{dynamic} + E^{VU}_{leakage} = (P^{VU}_{dynamic} + P^{VU}_{leakage}) \times (VUExecutionTime)$$
$$= (145.36 + 15.92) \times 1.15 = 185\mu J = 0.19mJ$$

$$E^{VU}_{standby} = E^{VU}_{leakage} = (P^{VU}_{leakage}) \times (VUIdleTime) = (15.92) \times (3.67 - 1.15)$$
$$= 40\mu J = 0.04mJ$$

$$E_{Total} = \sum_{n=1}^{n=N} (E^{MB_n}_{active} + E^{MB_n}_{standby}) + (E^{VU}_{active} + E^{VU}_{standby})$$
$$= (0.56 + 0 + 0.47 + 0.01) + (0.19 + 0.05) = 1.26mJ$$

We perform the same calculation for all the other situations and Table 5.7 shows the total execution time, power and area consumption. Fig. 5.11 shows the normalized values of the design metrics, the normalized execution time, the normalized $Resource * Energy$ and the normalized $Time*Energy*Resource$. When $M = Time*Energy*Resource$ is

Table 5.7: Execution Time and Resource/Power Consumption for the Demodulation Algorithm

MicroBlazes	1	2	3	4	5	6
Execution times(ms)	6.74	3.66	2.63	2.12	1.85	1.73
Percentage of slices	15%	23%	31%	39%	47%	55%
Power consumption (mJ)	1.30	1.26	1.26	1.27	1.29	1.28

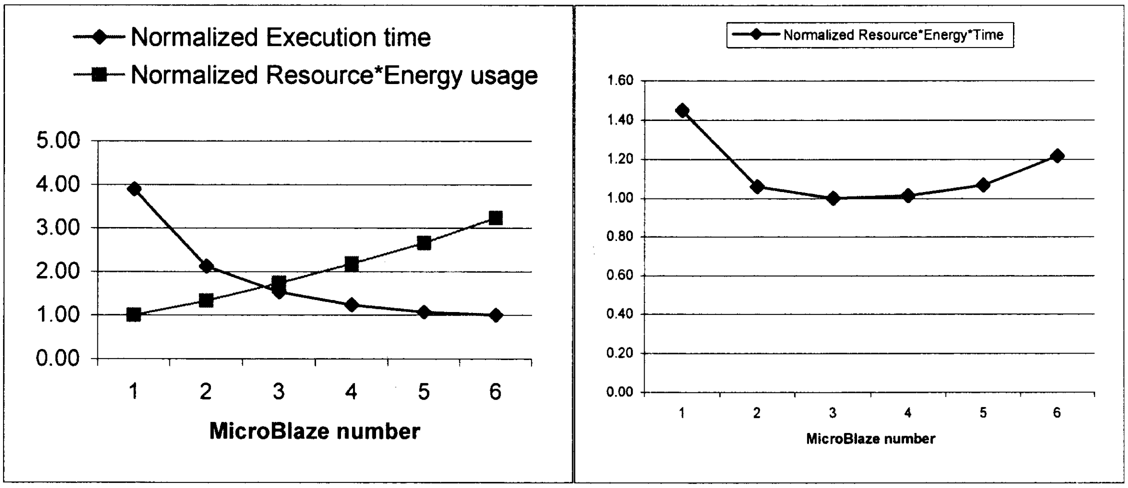

Figure 5.11: Normalized design metrics.

taken as the design metric for the demodulation algorithm, the right part in Fig. 5.11 shows that the AS-MSwSV system for demodulation algorithm should have three MicroBlazes with a shared vector unit. The vector unit should be configured with 1-bank having a multiplier and an adder. All the MicroBlazes should have FPU support.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The work in this dissertation stems from the observation that vector processing can easily outperform Superscalar and VLIW approaches for array-intensive algorithms while remaining area and power efficient. The fact that vector processing exploits DLP also makes rather simple the software/hardware implementation of vector operations. All of these facts make the vector structure a suitable approach for embedded applications with abundant DLP. Also, new FPGA generations demonstrate significant improvements in density, speed and structure, which make them feasible for SOC designs. So we first designed and implemented a programmable vector processor on a single-chip targeting embedded systems. It supports general-purpose instructions and is equipped with high-performance FPUs. A multibank structure is employed in the vector processor for area and power efficiency. Matrix multiplication and linear equation solvers, two kinds of representative computation-intensive benchmark algorithms, were run on the vector processor. The results are better than those of commercial PCs for large matrices despite much lower frequency due to the FPGA structure. With the fast development of FPGA technologies, more performance gains should be expected in the near future.

The focus in the second part of this dissertation, the MCwSV system, stems from the fact that recent research on GPP processor design has shifted from exploiting more ILP to multicore designs exploiting TLP, primarily because of the increasing difficulty in getting more ILP from applications. However, no current multicore processor contains a really powerful vector unit, which is the most efficient way to exploit DLP. Considering a vector unit requires many hardware resources and most general-purpose applications can only be partially vectorized. In the MCwSV system, the vector processing unit serves as a shared coprocessor to the resident scalar processors. The overall structure can then efficiently exploit all three kinds of parallelism: ILP, TLP and DLP. The implemented MCwSV system in this dissertation contains two scalar processors and one shared vector unit. It is prototyped on a Xilinx Virtex-5 FPGA platform. The MicroBlaze soft processor core from Xilinx is used as the scalar processor and the vector processing unit is developed to communicate with the MicroBlazes through standard interfaces and shared data memories. Besides the multibank structure employed in the first part of our work, the shared vector unit in this part is also organized in a decoupled way to facilitate ease of chaining and also to provide further performance increases. The incoming instruction stream is split into three different substreams for the three function units, which can be duplicated into several banks. The vector processor can be configured to have various kinds and different numbers of function units, and different number-of-vector-registers/vector-size pairs based on the requirements of applications and resource/power usage. The standard interfaces to the scalar processors, as well as the decoupled organization, provide good scalability in the MCwSV system. Three testbenches were run on the system. It was shown that our MCwSV system can yield significant performance increases while maintaining area efficiency.

However, the MCwSV system with two MicroBlazes and a shared vector unit is not always the optimized system configuration for various applications with different percentages of vectorization. So, finally, the third part of our work focused on building an architecture template for performance and resource management and guiding configuration decisions of the MCwSV system for each given application. The decision is based on the analysis of scalar/vector execution times and resource/power consumptions of various components. At the end, the architecture template gives the configuration of the MCwSV system with the type and number for FPUs in the shared vector processor, and the number of scalar processors and their configuration. The primary goal is to find a near-optimal configuration based on cost-performance tradeoffs for each given application. In out implementation, the product $M = Time * Energy * Resource$ of execution time, power and area is used to guide configurations. The MCwSV system which can provide the smallest M is taken as the optimized one. Two examples were given to show how to build an optimized MCwSV system for each specific application. For the demodulation algorithm, the optimized MCwSV system involves three MicroBlaze processors with FPU support and a 1-bank shared vector unit; for the RGB2YIQ algorithm, the optimized system needs one MicroBlaze without FPU support and a shared vector unit which has 2-, 4-, or 8-bank configuration for various data sizes.

Besides its use as a standalone system for embedded applications, the MCwSV system can also be used as a prototype for research in incorporating a vector coprocessor into current multicore processors. Our work demonstrates the effectiveness of a shared vector unit in the multicore processor environment. The same concept could be applied to the sharing of other units as well.

6.2 Future Work

More research could be done on the dynamic reconfiguration of the MCwSV system in terms of changing the FPU type/number dynamically for the vector processor and/or importing more scalar processors. The reconfiguration time should be calculated and compared with the estimated computation time, if possible, to decide if the reconfiguration should happen or not.

Different task scheduling policies, other than round-robin, could be studied. For example, an alternative scheduling method to allow a thread to run until it blocks due to a data or resource hazard may be a better choice for some tasks.

In addition, the performance of the MCwSV system is highly dependent on task scheduling between the scalar processor and the vector unit as well as on program vectorization. The current approach is neither portable nor general. It is necessary to explore languages and compilers to support program vectorization and task division, so the inherent features of the MCwSV system could be fully utilized.

REFERENCES

[1] MicroBlaze Processor. Date accessed Dec. 12, 2007 from the World Wide Web: http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm.

[2] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 4th ed. San Mateo: Morgan Kaufmann, 2006.

[3] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan, "Spert-II: A vector microprocessor system," *IEEE Computer*, vol. 29, no. 3, pp. 79–86, 1996.

[4] The 23rd TOP500 List. Date accessed Dec. 12, 2007 from the World Wide Web: http://www.top500.org/lists/2004/06.

[5] T. H. Dunigan, M. R. Fahey, J. B. White, and P. H. Worley, "Early evaluation of the Cray X1," in *SC'03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003, p. 18.

[6] S. Habata, M. Yokokawa, and S. Kitawaki, "The Earth simulator system," *NEC Research and Development*, vol. 44, pp. 21–26, 2003.

[7] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *International Symposium on Microarchitecture*, 1998, pp. 3–13.

[8] J. Gebis, S. William, C. Kozyrakis, and D. Patterson., "VIRAM-1: A media-oriented vector processor with embedded DRAM," in *Student Design Contest, the 41st Design Automation Conference*, 2004.

[9] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *The 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, 2002, pp. 283 – 293.

[10] L. Han, J. Chen, C. Zhou, Y. Li, X. Zhang, Z. Liu, X. Wei, and B. Li, "An embedded reconfigurable SIMD DSP with capability of dimension-controllable vector processing," in *ICCD'04: Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 2004, pp. 446 – 451.

[11] J. Chen, C. Zhou, and etc., "A reconfigurable architecture of high performance embedded DSP core with vector processing ability," in *Proceedings of the 5th International Conference on ASIC.*, Oct. 2003, pp. 377–380.

[12] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis, "Vector lane threading," in *ICPP'06: Proceedings of the 2006 International Conference on Parallel Processing*. Washington, DC: IEEE Computer Society, 2006, pp. 55–64.

[13] The Cell Project at IBM Research. Date accessed Dec. 12, 2007 from the World Wide Web: http://www.research.ibm.com/cell/.

[14] K. Asanović, “Vector microprocessors,” Ph.D. dissertation, University of California, Berkeley, 1998.

[15] S. L. Graham, M. Snir, and C. A. Patterson, *Getting up to Speed : the Future of Supercomputing.* Washington, DC: National Academies Press, 2005.

[16] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction.* New York, NY: John Wiley & Sons, Inc., 2002.

[17] P. Koopman, “Embedded system design issues (the rest of the story),” in *ICCD’96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, Washington, DC, 1996, p. 310.

[18] C. Kozyrakis, “Scalable vector media-processors for embedded systems,” Ph.D. dissertation, University of California, Berkeley, 2002.

[19] C. Kozyrakis and D. Patterson, “Overcoming the limitations of conventional vector processors,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 399–409.

[20] The Embedded Microprocessor Benchmark Consortium. Date accessed Dec. 12, 2007 from the World Wide Web: http://www.eembc.org/.

[21] D. Geer, “Industry trends: Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[22] P. Gepner and M. F. Kowalik, “Multi-core processors: New way to achieve high system performance,” in *PARELEC’06: Proceedings of the International Symposium on Parallel Computing in Electrical Engineering.* Washington, DC: IEEE Computer Society, 2006, pp. 9–13.

[23] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, “The impact of performance asymmetry in emerging multicore architectures,” in *ISCA’05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, Washington, DC, 2005, pp. 506–517.

[24] Intel Multi-Core Technology. Date accessed Dec. 12, 2007 from the World Wide Web: http://www.intel.com/multi-core/index.htm.

[25] AMD Multi-Core Processors. Date accessed Dec. 12, 2007 from the World Wide Web: http://multicore.amd.com/en/.

[26] Sun UltraSPARC T1 Processor. Date accessed Dec. 12, 2007 from the World Wide Web: www.sun.com/processors/UltraSPARC-T1/.

[27] Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek, “Ramp: Research accelerator for multiple processors - a community vision for a shared experimental parallel HW/SW platform,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-05-1412, 2005.

[28] T.-F. Chen, C.-M. Hsu, and S.-R. Wu, "Flexible heterogeneous multicore architectures for versatile media processing via customized long instruction words." *IEEE Trans. Circuits Syst. Video Techn.*, vol. 15, no. 5, pp. 659–672, 2005.

[29] G. Nicolescu, S. Yoo, A. Bouchhima, and A. A. Jerraya, "Validation in a component-based design flow for multicore SoCs," in *ISSS'02: Proceedings of the 15th International Symposium on System Synthesis.* New York, NY: ACM Press, 2002, pp. 162–167.

[30] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling," in *ISCA'05: Proceedings of the 32nd Annual International Symposium on Computer Architecture.* Washington, DC: IEEE Computer Society, 2005, pp. 408–419.

[31] G. Xu, "Thermal modeling of multi-core processors," in *ITHERM'06: The 10th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronics Systems.*, 2006, pp. 96–100.

[32] J. C. Hu, "Creating a framework for developing high-performance web servers over ATM." Date accessed Dec. 10, 2006 from the World Wide Web: citeseer.ist.psu.edu/63093.html.

[33] M. Gschwind, "Chip multiprocessing and the cell broadband engine," in *CF'06: Proceedings of the 3rd Conference on Computing Frontiers*, 2006, pp. 1–8.

[34] B. Radunovic, "An overview of advances in reconfigurable computing systems," in *Hawaii International Conference on System Sciences*, 1999.

[35] P. Lysaght and P. A. Subrahmanyam, "Guest editors' introduction: Advances in configurable computing," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 85–89, Mar.-Apr. 2005.

[36] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, Eds., *Splash2 : FPGAs in a Custom Computing Machine.* Los Alamitos, CA: IEEE Computer Society Press, 1996.

[37] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: A high-end reconfigurable computing system," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.

[38] C. Beaumont, P. Boronat, C. Champeau, J.-M. Filloque, and B. Pottier, "Reconfigurable technology: An innovative solution for parallel discrete event simulation support," in *PADS'94: Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, 1994.

[39] D. M. Lewis, M. van Ierssel, J. Rose, and P. Chow, "The Transmogrifier-2: A 1 million gate rapid-prototyping system," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 6, no. 2, pp. 188–198, 1998.

[40] B. Garrett, "FPGA-based design delivers customized embedded solutions," in *Embedded Systems Conference*, 2005.

[41] X. Wang and S. G. Ziavras, "Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 4, pp. 391–343, Apr. 2004.

[42] ——, "Parallel solution of Newton's power flow equations on configurable chips," *International Journal of Electrical Power and Energy Systems*, vol. 29, no. 5, pp. 422–431, June 2007.

[43] X. Xu and S. G. Ziavras, "A hierarchically-controlled SIMD machine for 2D DCT on FPGAs," in *IEEE International Systems-On-Chip Conference*, Sept. 2005.

[44] ——, "A coarse-grain hierarchical technical for 2-dimensional FFT on configurable parallel computers," *IEICE Trans. on Information and Systems, Special Issue on Parallel/Distributed Computing and Networking*, vol. E89-D, no. 2, Feb. 2006.

[45] H. Yang and S. G. Ziavras, "FPGA-based vector processor for algebraic equation solvers," in *IEEE International Systems-On-Chip Conference*, Washington, DC, Sept. 2005, pp. 115–116.

[46] H. Yang, S. Wang, S. G. Ziavras, and J. Hu, "Vector processing support for FPGA-oriented high performance applications," in *IEEE Computer Society Annual Symposium on VLSI*, Porto Alegre, Brazil, May 2007, pp. 447–448.

[47] H. Yang, S. G. Ziavras, and J. Hu, "FPGA-based vector processing for matrix operations," in *International Conference on Information Technology: New Generations*, Las Vegas, NV, Apr. 2007, pp. 989–994.

[48] ——, "Reconfiguration support for vector operations," *International Journal of High Performance Systems Architecture*, vol. 1, no. 2, pp. 89–97, 2007.

[49] Starbridge System. Date accessed Dec. 10, 2007 from the World Wide Web: http://www.starbridgesystems.com.

[50] CoreFire FPGA Design Suite. Date accessed Dec. 10, 2007 from the World Wide Web: http://www.annapmicro.com/corefire.html.

[51] ARM11 MPCore. Date accessed Dec. 12, 2007 from the World Wide Web: http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[52] C. E. Kozyrakis and D. A. Patterson, "Scalable vector processors for embedded systems," *IEEE Micro*, vol. 23, no. 6, pp. 36 – 45, Nov.-Dec. 2003.

[53] WILDSTAR II for PCI. Date accessed Dec. 10, 2007 from the World Wide Web: http://www.annapmicro.com/wsiipci.html.

[54] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 161–171.

[55] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective.* Prentice Hall, 2003.

[56] H. S. Huang and C. N. Lu, "Efficient storage scheme and algorithms for W-matrix vector multiplication on vector computers," *IEEE Trans. Power Syst.*, vol. 9, no. 2, pp. 1083–1091, May 1994.

[57] A. Gómez and R. Betancourt, "Implementation of the fast decoupled load flow on a vector computer," *IEEE Trans. Power Syst.*, vol. 5, no. 3, pp. 977–983, Aug. 1990.

[58] G. P. Granelli, M. Montagna, G. L. Pasini, and P. Marannino, "A W-matrix based fast decoupled load flow for contingency studies on vector computers," *IEEE Trans. Power Syst.*, vol. 8, no. 3, pp. 946–953, Aug. 1993.

[59] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proceedings of the ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays, FPGA 2005*, Monterey, CA, Feb. 2005, pp. 86–95.

[60] M. K. Enns, W. F. Tinney, and F. L. Alvarado, "Sparse matrix inverse factors," *IEEE Trans. Power Syst.*, vol. 5, no. 2, pp. 466–473, May 1990.

[61] F. L. Alvarado, D. C. Yu, and R. Betancourt, "Partitioned sparse A^{-1} methods," *IEEE Trans. Power Syst.*, vol. 5, no. 2, pp. 452–459, May 1990.

[62] J. Q. Wu and A. Bose, "A new successive relaxation scheme for the W-matrix solution method on a shared memory parallel computer," *IEEE Trans. Power Syst.*, vol. 11, no. 1, pp. 233–238, Feb. 1996.

[63] A. Padilha and A. L. F. Morelato, "A W-matrix methodology for solving sparse network equations on multiprocessor computers," *IEEE Trans. Power Syst.*, vol. 7, no. 3, pp. 1023–1030, Aug. 1992.

[64] Matrix Market. Date accessed Dec. 10, 2007 from the World Wide Web: http://math.nist.gov/MatrixMarket/.

[65] R. Espasa and M. Valero, "Decoupled vector architectures," in *Proceedings of Second International Symposium on High-Performance Computer Architecture*, Feb. 1996, pp. 281–290.

[66] ——, "Multithreaded vector architectures," in *Proceedings of Third International Symposium on High-Performance Computer Architecture*, Feb. 1997, pp. 237–248.

[67] S. Wang, H. Yang, J. Hu, and S. G. Ziavras, "Asymmetrically banked value-aware register files," in *IEEE Computer Society Annual Symposium on VLSI*, Porto Alegre, Brazil, May 2007, pp. 363–368.

[68] S. Wang, H. Yang, S. G. Ziavras, and J. Hu., "Asymmetrically banked value-aware register files for low energy and high performance," *Microprocessors and Microsystems.*

[69] F. Li, Y. Lin, L. He, D. Chen, and J. Cong, "Power modeling and characteristics of field programmable gate arrays," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1712–1724, 2005.

[70] A. Telikepalli, "Power vs. performance: The 90 nm inflection point," Xilinx white paper for Virtex-4.

[71] A. Gayasen, Y. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and T. Tuan, "Reducing leakage energy in FPGAs using region-constrained placement," in *FPGA'04: Proceedings of the 12th ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2004, pp. 51–58.

[72] F. Li, Y. Lin, L. He, and J. Cong, "Low-power FPGA using pre-defined dual-vdd/dual-vt fabrics," in *FPGA'04: Proceedings of the 12th ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2004, pp. 42–50.

[73] F. Li, Y. Lin, and L. He, "FPGA power reduction using configurable dual-vdd," in *DAC'04: Proceedings of the 41st Annual Conference on Design Automation*, 2004, pp. 735–740.

[74] J. H. Anderson and F. N. Najm, "Low-power programmable routing circuitry for FPGAs," in *ICCAD'04: Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, 2004, pp. 602–609.

[75] J. Ou and V. K. Prasanna, "Rapid energy estimation of computations on FPGA based soft processors," in *IEEE International Systems-On-Chip Conference*, 2004, pp. 285–288.

[76] L. Wang, M. French, A. Davoodi, and D. Agarwal, "FPGA dynamic power minimization through placement and routing constraints," *EURASIP J. Embedded Syst.*, vol. 2006, no. 1, 2006.

[77] J.-W. Jang, S. B. Choi, and V. K. Prasanna, "Energy- and time-efficient matrix multiplication on FPGAs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 11, pp. 1305–1319, 2005.

[78] X. Wang and S. G. Ziavras, "Adaptive scheduling of array-intensive applications on mixed-mode reconfigurable multiprocessors," in *39th IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, 2005.

[79] MicroBlaze Processor Reference Guide. Date accessed Dec. 10, 2007 from the World Wide Web: http://www.xilinx.com/support/documentation/sw_manuals/edk63i_mb_ref_guide.pdf.