Multithreaded Processor Core Optimized for Parallel Thread Execution

BY

Ayman Ali Mohammad Hroub

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

May 2011

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Ayman Ali Mohammad Hroub** under the supervision of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** in **COMPUTER ENGINEERING**

Thesis Committee

Dr. Muhamed Mudawar
(Chairman)

Dr. Aiman El-Maleh
(Member)

Dr. Abdelhafid Bouhraoua
(Member)

Dr. Basem AL-Madani
Department Chairman

Dr. Salam Zummo
Dean of Graduate Studies

21/6/11
Date

Dedicated to

the memory of my mother

and

the memory of my brother Abdul-Lateef

ACKNOWLEDGMENT

In the name of God the Merciful

All praises and glory be to Allah (SWT) who gave me everything and I pray that He continues giving me the guidance, the grace and the health during the rest of my life.

I would like to thank the great university; King Fahd University of Petroleum and Minerals for supporting this research and for giving me the opportunity to pursue my graduate studies.

I wish to express my appreciation to my advisor, **Dr. Muhamed Mudawar** for his guidance, support, help, cooperation and constructive feedback. Dr. Mudawar dedicated a lot of his valuable time for me and he always provides me with brilliant ideas. I am also very grateful to my thesis committee members: **Dr. Abdelhafid Bouhraoua and Dr. Aiman El-Maleh** for their help, cooperation, support, guidance and constructive feedback.

I would like to thank my family for their support and encouragement. I am so grateful to my father, my brothers and my sisters.

Table of Contents

LIST OF TABLES viii

LIST OF FIGURES ix

THESIS ABSTRACT xi

THESIS ABSTRACT (ARABIC) xiii

CHAPTER 1 INTRODUCTION 1

1.1 Overview 1

1.2 Multithreaded Processors 2

1.2.1 The Motivation Behind Having Multithreaded Processors 3

1.3 PAR ISA 4

1.3.1 Par Instruction 6

1.3.2 Stop Bit 6

1.3.3 Control Instructions 8

1.4 Thesis Motivation 11

1.5 Thesis Organization 11

CHAPTER 2 LITERATURE REVIEW 12

2.1 Multithreaded Processors 12

2.2 Vector Processors 16

2.3 Processors Frontend 20

2.4 Block-Structured ISA 22
2.5 Simulation 24
2.6 Discussion 26
CHAPTER 3 PAR CORE HARDWARE MODEL 29
3.1 Overview 29
3.2 Assumptions 31
3.3 The Multithreading Approach 32
3.4 Why is PAR Core Optimized? 33
3.5 Frontend Structure 34
3.6 How Does the Frontend Work? 39
3.7 Backend Structure 44
3.7.1 Register File 46
3.7.2 Predicate Registers 47
3.7.3 Reorder Buffer (ROB) 48
3.7.4 Functional Units 49
3.7.5 Instruction Waiting Queue 50
3.7.6 Decode and Dispatch Logic 51
3.7.7 Issue Logic 52
3.7.8 Forwarding Network 52
3.7.9 Write Back Logic 55

3.8 Hazards 56

CHAPTER 4 PARSIM SIMULATOR 58

4.1 Overview 58

4.2 ParSim Structure 59

4.3 Simulation Methodology 59

4.4 Performance Metrics 66

4.5 The Source Code File 66

4.6 The Configuration File 69

4.7 The Performance Statistics Report File 70

4.8 Conclusion 71

CHAPTER 5 BENCHMARKS 73

5.1 Overview 73

5.2 Dense Matrix-Matrix Multiplication (DMMM) 74

5.3 Jacobi Iterative Method (JIM) 74

5.4 Gauss-Seidel (GS): Red-Black Gauss-Seidel on a 2D Grid [32] 75

5.5 RGB to YIQ Conversion (RGB-YIQ) [33] 76

5.6 RGB to CMYK Conversion (RGB-CMYK) [33] 77

5.7 High Pass Grey-Scale Filter (HPF) [33] 77

5.8 Scaled Vector Addition (SVA) 79

CHAPTER 6 EXPERIMENTAL RESULTS AND DISCUSSION 80

6.1 Overview 80

6.2 Analyzing Single Lane Performance 81

6.3 Analyzing Multiple Lanes Performance 85

CHAPTER 7 CONCLUSION AND FUTURE WORK 88

Appendix A PAR ISA 90

Appendix B Benchmarks C++ Source Code 110

Appendix C Benchmarks PAR Assembly Source Code 121

Appendix D Experimental Results 143

D.1 Results for Single Lane 143

D.2 Results for Multiple Lanes 146

Bibliography 150

Vitae 155

LIST OF TABLES

Table 1: The Values of the Fixed Parameters 80

Table 2: DMMM Performance Results for Single Lane 143

Table 3: JIM Performance Results for Single Lane 143

Table 4: GS Performance Results for Single Lane 144

Table 5: RGB-YIQ Performance Results for Single Lane 144

Table 6: RGB-CMYK Performance Results for Single Lane 145

Table 7: HPF Performance Results for Single Lane 145

Table 8: SVA Performance Results for Single Lane 146

Table 9: DMMM Performance Results for Multiple Lanes 146

Table 10: JIM Performance Results for Multiple Lanes 147

Table 11: GS Performance Results for Multiple Lanes 147

Table 12: RGB-YIQ Performance Results for Multiple Lanes 148

Table 13: RGB-CMYK Performance Results for Multiple Lanes 148

Table 14: HPF Performance Results for Multiple Lanes 149

Table 15: SVA Performance Results for Multiple Lanes 149

LIST OF FIGURES

Figure 1: An Instruction Block is Terminating with a Stop Bit 5

Figure 2: VIRAM Prototype Processor [17] 18

Figure 3: Vector Lane Organization: Centralized (a) Clustered (b) [17] 19

Figure 4: The 32-Bit Basic Block Descriptor Format in BLISS [3] 23

Figure 5: High Level Block Diagram of PAR Core 30

Figure 6: Block Diagram of PAR Core's Frontend 36

Figure 7: Block Diagram of PAR Core's Backend 45

Figure 8: PAR Core's Register File 46

Figure 9: Forwarding Network for ALU 54

Figure 10: High Level Flowchart of ParSim 60

Figure 11: ParSim's Simulation Process 62

Figure 12: Functional Unit Monitoring Flowchart 65

Figure 13: A Sample Source Code Input File 68

Figure 14: A Snapshot of the Configuration File for ParSim 70

Figure 15: A Snapshot from the Performance Statistics Report Generated by ParSim 71

Figure 16: High Pass Grey-Scale Filter Coefficients [33] 78

Figure 17: High Pass Grey-Scale Filter Equations [33] 78

Figure 18: IPC for a Single Lane 82

Figure 19: FPU Throughput for a Single Lane 83

Figure 20: ALU Utilization 84

Figure 21: Load/Store Unit Utilization 85

Figure 22: IPC in Multiple Lanes 86
Figure 23: Speedup across Multiple Lanes 87
Figure 24: Instruction Formats 90
Figure 25: ALU Instruction Formats 98
Figure 26: Format of the SET and SLI instructions 99
Figure 27: Integer Multiply and Divide Instruction Formats 102
Figure 28: R-type and I-type Compare Instruction Formats 108
Figure 29: DMMM C++ Source Code 110
Figure 30: JIM C++ Source Code 111
Figure 31: GS C++ Source Code 113
Figure 32: RGB-YIQ C++ Source Code 115
Figure 33: RGB-CMYK C++ Source Code 117
Figure 34: HPF C++ Source Code 119
Figure 35: SVA C++ Source Code 120
Figure 36: DMMM Source Code 122
Figure 37: JIM Source Code 124
Figure 38: GS Source Code 127
Figure 39: RGB-YIQ Source Code 130
Figure 40: RGB-CMYK Source Code 134
Figure 41: HPF Source Code 137
Figure 42: SVA Source Code 141

THESIS ABSTRACT

Name: Ayman Ali Mohammad Hroub

Title: Multithreaded Processor Core Optimized for Parallel Thread Execution

Major Field: Computer Engineering

Date of Degree: May 2011

The accelerating improvements in VLSI technology allow adding more and more transistors on a single chip. This has been exploited by computer architects to develop more complex and more efficient processors like superscalar which exploits the instruction level parallelism (ILP) in the applications to handle multiple instructions simultaneously. Some applications like graphics processing and scientific computing are throughput applications and they have a lot of data level parallelism. The complex features such as aggressive branch prediction, multiple instructions issue and out of order execution that exist in many processors like superscalar are not needed for these computing areas. Special purpose processors should be designed for these applications. These processors should be single instruction multiple threads (SIMT) processors such that when an instruction is issued, it is executed for multiple independent threads sequentially.

In this research, I am proposing a processor core called PAR core which is based on the PAR instruction set architecture (PAR ISA) proposed by Dr. Mudawar. PAR core is an SIMT core that receives the workload from the master process in a format called

PAR packet which orders the PAR core to execute the same sequence of instructions for a given number of threads specified by the PAR packet.

The simulation results showed that the PAR core has high throughput and high utilization of the hardware resources. The maximum hardware utilization is 100% and the maximum IPC gained is 2.75 instructions/ cycle for a 4-way multithreaded PAR core. Besides that, the simulation results showed that this architecture is completely scalable which means replicating the processing lanes will replicate the throughput. PAR core has been scaled up to 64 processing lanes and the speedup is linear and the maximum IPC is 174.26 instructions/ cycle.

THESIS ABSTRACT (ARABIC)

ملخّص الرسالة

الاسم: أيمن علي محمد حروب

عنوان الرسالة: نواة معالج متعددة النياسب أمثلية للتنفيذ المتوازي للنياسب

التخصص: هندسة الحاسب الآلي

تاريخ التخرّج: أيّار ٢٠١١

لقد استغل معماريّو الحاسب الآلي التطوّر المتسارع في تقنية الدوائر المتكاملة عالية الكثافة (VLSI) و المتمثل في زيادة عدد الترانزستورات على الرقاقة الإلكترونية الواحدة في تطوير معالجات أكثر تعقيداً و أكثر فعاليّةً مثل معالج (Superscalar) الذي يعتمد على وجود تعليمات متوازية (ILP) في البرامج لكي يتسنّى له معالجة أكثر من تعليمة في آن واحد. بعض التطبيقات كمعالجة الرسوم و الحسابات العلمية هي تطبيقات كثيرة الإنتاجيّة حيث تحتوي على كمية هائلة من التوازي. إنّ المزايا المعقدة كالتنبأ بالتفرّع و إصدار أكثر من أمر معاً و التنفيذ غير المرتّب و الموجودة في العديد من المعالجات مثل (Superscalar) غيرضرورية لهذه التطبيقات. لذا فإن هذه التطبيقات تحتاج إلى معالجات خاصة تصمّم خصصيصا لها. هذه المعالجات الخاصة يجب أن تكون معالجات من نوع تعليمة واحدة و عدة نياسب (SIMT) و الذي يقتضي إصدار تعليمة واحدة و تنفيذها لعدة نياسب بالتعاقب.

إنني أقترح في هذا البحث نواة معالج تسمّى (PAR) و المبنية على مجموعة التعليمات (PAR ISA) التي اقترحها الدكتور مدوّر. تعتبر نواة المعالج (PAR) من نوع (SIMT) و تقوم باستلام العبء من الإجرائية الرئيسية على شكل رزمة تسمّى (PAR Packet) حيث تقوم النّواة (PAR) بتنفيذ البرنامج المرتبط بهذه الرزمة لعدد معيّن من النياسب يتم تحديده في الرزمة (PAR Packet).

لقد أظهرت نتائج المحاكاة أن النواة (PAR) نواة عالية الإنتاجية و عالية الاستفادة من مصادر العتاد المادي الموجودة. لقد كانت نسبة الاستفادة من العتاديات المادية ١٠٠% في أحسن الأحوال و كان الحد الأقصى لعدد التعليمات التي يتم تنفيذها في الدورة الزمنية الواحدة ٢,٧٥ تعليمة/الدورة عندما كانت النواة تحتوي على أربعة نياسب. بالإضافة إلى ذلك، أظهرت نتائج المحاكاة أن هذا النوع من العمارة تتضاعف إنتاجيته بتضاعف مصادر العتاد المادي حيث تم زيادة حجم النواة إلى ٦٤ قناة معالجة فكان التسريع خطياً و كان عدد التعليمات التي يتم تنفيذها في الدورة الواحدة ١٧٤,٢٦ تعليمة.

CHAPTER 1 INTRODUCTION

1.1 Overview

The accelerating improvements in VLSI technology which allow adding more and more transistors on a single chip have been exploited by the computer architects to develop more efficient processors. Adding more transistors means adding more functional units and thus increasing the processor's throughput if these units are utilized properly. The processor throughput is defined as the number of instructions executed per the unit of time. Increasing the processor throughput requires that the processor itself has multiple functional units as well as the architecture should take maximizing the utilization of these resources into consideration.

Moreover, there is an accelerating application demand on more powerful processors because the computer applications like simulation, scientific computing, multimedia processing etc. become more and more complex. The uniprocessor computer cannot fulfill the requirements of these applications because the execution time will be too long. Increasing the efficiency of a uniprocessor will make it more complex and it will be hit with the heat problem. The solution for that is having parallel computers.

The parallel computer is defined as "a collection of processing elements that cooperate and communicate to solve large problems fast" [1]. Parallel computing implies

replicating the processing unit to handle multiple processes concurrently. These parallel processing units may be arranged to communicate together via shared memory or message passing. So there are multiple architectures of parallel computers such that the workload is partitioned into smaller pieces that are assigned to different processing units.

In addition to the ability of parallelizing the execution of an application on multiple processing units, there is an ability to run multiple threads on a single processing pipeline using some multithreading technique. In this thesis, I am proposing an SIMT multithreaded processor core called PAR core to handle multiple independent threads concurrently.

1.2 Multithreaded Processors

There are two approaches for multithreading architecture [2]. (1) The single chip multiprocessor approach which integrates two or more independent processors on a single chip. (2) The multithreaded processor which is able to pursue two or more threads of control in parallel within a single processor pipeline.

The multithreaded processor can be defined as a processor that is able to handle multiple instructions of multiple threads concurrently by multiplexing the functional units in the execution pipeline among these threads. The multithreaded processor pipeline can be single issue or multiple issue. Ungerer et al. [2] indicated that there are three principle multithreading techniques. (1) Interleaved multithreading technique in which one instruction from another thread is fetched and fed to the execution pipeline each clock

cycle so this technique is a fine grained cycle-by-cycle multithreading technique. (2) Blocked multithreading technique, it is coarser grained than the first one such that in this technique a thread continues execution until a blocking event occurs like remote memory request, then another ready thread is activated. This technique of multithreading requires implementing some efficient context-switch mechanism. (3) Simultaneous multithreading in which the wide superscalar instruction issue is combined with multithreading such that instructions are simultaneously issued from multiple threads to the execution pipeline.

1.2.1 The Motivation Behind Having Multithreaded Processors

There were some reasons that pushed the architects to propose this kind of architecture. One of the strongest reasons is to tolerate the memory latency. When a process makes a remote memory request which takes a long time, then this process will be blocked until this request is satisfied. Blocking the process means killing the throughput and the hardware resources utilization. Multithreading solves this problem by switching to another thread and therefore the processor stays busy and productive.

Another reason for having multithreaded processors is the data dependence among the instructions of a single thread. While a long latency instruction is being executed, the subsequent instructions that depend on its result cannot be issued and then some functional units will stay idle which reduces the hardware utilization and the processor's throughput. Multithreading techniques can solve this problem by executing instructions

from other threads on these functional units which increases the hardware utilization and the processor's throughput.

It is clear that the motivation behind having multithreaded processors is increasing the functional units' utilization and thus increasing the processor's throughput through filling the unused slots caused by data dependence and remote memory requests with useful work from other threads.

1.3 PAR ISA

The PAR ISA is a block-structured ISA proposed by Dr. Mudawar. The program in this ISA consists of blocks of instructions. The instruction block is defined as a sequence of instructions starting at the target address of a control instruction and ending with an instruction whose stop bit is set. There is no need to store additional information to determine the instruction block end because the stop bit which is a part of the instruction binary format does that. Figure 1 shows an instruction block terminating with a stop bit.

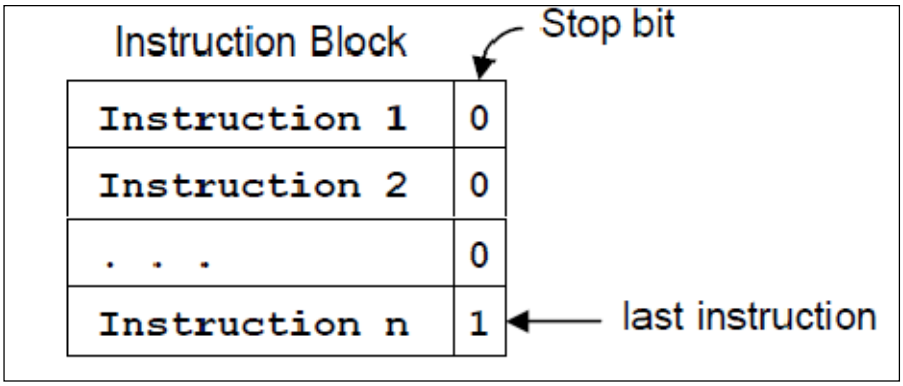

Figure 1: An Instruction Block is Terminating with a Stop Bit

PAR ISA contains all kinds of standard instructions like integer and floating point (FP) arithmetic, logic, memory, compare and control instructions. Appendix A shows the syntax and the format of these instructions. In addition to that, PAR ISA contains the par instruction to run independent threads in parallel. All instructions have a fixed length of 32 bits. These 32 bits are divided into several fields according to the instruction format. Every instruction has a stop bit and a qualifying predicate register (qp). This means that all instructions are predicated such that the instruction will be dropped from the execution pipeline and doesn't change the system status when its qualifying predicate register value is zero. In the subsequent subsections, some of the PAR ISA features such as the par instruction, the stop bit, the control instructions and the control stack will be described.

1.3.1 Par Instruction

Par instruction is used to spawn a group of independent threads called worker threads that can work in parallel. This instruction appears only within a master thread which runs on a master core. This instruction has the following syntax:

(qp) par i , target

When this instruction is encountered and its qp is true, then a set of PAR packets is generated. The PAR packet contains the number of threads to be executed, the values of the inherited registers from the master thread and the starting address of the thread program which is specified by the label *target*. The number **i** refers to the inherited registers from 0 to **i**. After the PAR packets are generated, the thread scheduler distributes them on the PAR cores according to the scheduling policy.

1.3.2 Stop Bit

As mentioned above, the stop bit is a part of the instruction format to mark the end of an instruction block. This feature is visible to the programmer. The programmer can indicate that a certain instruction is the last instruction in the instruction block by typing the optional hash symbol (#) at the end of the instruction line. When the assembler detects the hash symbol, it knows that this instruction is the last instruction in the block and therefore it sets the stop bit of that instruction and when no hash symbol is detected this

means that this instruction is not the last instruction in the block and therefore the stop bit of this instruction is cleared.

The stop bit is beneficial for the following reasons:

1. It simplifies the hardware and the software because it eliminates the need to store additional information to track the instruction blocks and no complex action is needed from the compiler to detect the ends of the blocks and to perform further code optimization due the descriptors as in [3].
2. It removes the need of having a return instruction to return to the caller after the function execution finishes, so it saves one instruction per function call.
3. It offers simpler and more accurate mechanism in pre-fetching instructions from the I-cache to the execution pipeline because the fetch unit can detect the instruction block end.
4. The stop bit with the control stack that will be described later helps the fetch unit to detect the thread termination. The thread terminates when the last instruction is reached and the control stack is empty.
5. It helps in designing an instruction pre-fetcher to pre-fetch instructions from the main memory to the on chip I-cache since it tells the pre-fetcher where the end of the instruction block is.

1.3.3 Control Instructions

PAR ISA contains five control instructions that control the program's execution flow. These instructions are useful in building an effective and more accurate fetch unit that can feed the execution engine with instructions in a high rate. These instructions include expand, indirect expand, counter-controlled loop, pure conditional loop and break instructions.

1. The Expand Instruction

It is used to expand instruction blocks and it is equivalent to the function call instruction but without having a corresponding return instruction because the stop bit has eliminated this need. The expand instruction syntax is as follows:

(qp) xp Label

As any other instruction in PAR ISA, the expand instruction is predicated. If qp is true, then the execution flow should be transferred to the target address specified by the label and when qp is false then the expand instruction is skipped. In addition to its role in the program flow control, the expand instruction can be exploited in implementing instruction pre-fetchers to transfer the instruction blocks from the main memory to the I-Cache ahead of time. This can be done by checking the opcode of the instruction while the instruction block is being fetched and when a control instruction is detected like expand, indirect expand or loop instruction then the next required block will be known ahead of time and it can be pre-fetched. The expand instruction and the other control instructions also guide the fetch unit to update the program counter.

2. The Indirect Expand Instruction

This instruction is used when the target address is known at runtime. The value of the target address is specified indirectly in a general purpose register. The syntax of this instruction is:

(qp) xp r

It can be used to address shared libraries that are dynamically linked at runtime, to expand methods indirectly in object-oriented programming languages etc.

3. The Counter-Controlled Loop Instruction

This instruction is used to implement the counter-controlled loops. It has the following syntax:

(qp) loop r, L

In this instruction, the loop block at the target address specified by the label is executed at most n times. The number n is specified in a general purpose register **r**. As long as the value of qp is true, the loop instruction block is expanded.

If the loop instruction is not the last instruction of the instruction block, its effect will be loop and continue i.e. after the loop instruction finishes, the execution flow resumes at the next instruction after the loop instruction. However if the loop instruction is the last instruction of the instruction block, then its effect will be loop and return i.e. after the loop finishes, the execution flow resumes at the return address.

4. The Pure Conditional Loop Instruction

It is equivalent to the ***while*** loop in C++. It has the following syntax:

(qp) loop L

When this instruction is encountered, the instruction block at the target address specified by the label is expanded as long as the qp is true.

5. The Break Instruction

The effect of this instruction is to terminate the current instruction block prematurely by resuming execution at the return address that is popped off the control stack. The general syntax of this instruction is:

(qp) brk n

If n is not specified, the effect of this instruction will be skipping the rest of instructions in the current instruction block and resumes execution from the return address when this instruction is encountered in a non-loop instruction block. However, if it is encountered in a loop instruction block, the effect will be skipping the rest of instructions in the current loop iteration and resuming execution from the next iteration and this is equivalent to the ***continue*** statement in C++.

If n equals 1 and the break instruction is encountered within a non-loop instruction block, the effect will be skipping the rest of instructions in the current instruction block and its parent block. If it is encountered in a loop instruction block, the effect will be skipping the remaining instructions in the current iteration and the remaining iterations and this is equivalent to the ***break*** statement in C++.

1.4 Thesis Motivation

The existence of data parallel applications that have completely independent fine grained threads motivate to have a multithreaded architecture that issues one instruction and executes it for multiple threads. This kind of architecture increases the processor's throughput.

Besides that, the features of the PAR ISA like the PAR instruction, the PAR packet and the ability to have a simple processor frontend have motivated me to propose the PAR core.

1.5 Thesis Organization

This Thesis has been organized in seven chapters. Chapter 2 contains a literature review about the related work to this research. In the literature review, some existing vector processors, multithreaded architectures and fetch mechanisms have been discussed. In chapter 3, the microarchitecture of the PAR core has been described. Chapter 4 describes ParSim simulator which is a cycle accurate multithreaded simulator that has been developed to assess the performance of the PAR core. In chapter 5, the data parallel benchmarks that have been used to measure the PAR core’s performance have been discussed. In chapter 6, the experimental results have been displayed and analyzed. Finally chapter 7 describes the conclusion drawn from this research work and it contains the future work activities that extend this research.

CHAPTER 2 LITERATURE REVIEW

2.1 Multithreaded Processors

A multithreaded processor is a processor that can handle more than one instruction from different threads simultaneously. The first multithreaded processors appeared in 1970s and 1980s to solve the problem of remote memory access. From those days until now, many multithreading architectures have been proposed either for general or special purpose computing.

In 1998, El-Kharashi et al. predicted that the multithreaded processors will be the upcoming generation for multimedia chips because the multimedia applications suffer from long latencies as a result of networks contention, frequent memory references and limited communication bandwidth. Having multithreaded processors will tolerate these latencies by switching to another thread whenever some thread faces a long latency operation. El-Kharashi et al. mentioned the motivations of having multithreaded processors such as: hiding latencies, having dynamic task scheduling, it is a further step towards concurrency, to improve multichip behavior, to alleviate the operating system overhead. They also mentioned that the multithreaded processor can be fine-grained and coarse-grained and each class has its pros and cons and it has its own hardware and software requirements. They listed the general hardware requirements for a multithreaded processor and these requirements include: handling multiple contexts, efficient register manipulation, hardware thread scheduler, state replication, additional control circuitry,

handling pipelining, sharing resources, advanced memory management, scalable memory protection, efficient communication and built-in synchronization.

Many ideas have been proposed to increase the performance of multithreaded processors by proposing new multithreaded architectures to make better utilization of the resources, to decrease the hardware complexity or to target a given class of applications like multimedia or scientific computing. In 1999, Zahran and Franklin [4] proposed a speculative multithreaded architecture with dynamic thread resizing at runtime. In this architecture, the threads are extracted from a sequential program by the compiler or by hardware and they are speculatively executed in parallel. In the first step, tasks are generated statically at compile time and later they are resized dynamically at runtime according to the program behavior. Their main contributions are: Hierarchical technique for building threads. A non-sequential scheme to assign threads to the processing elements (PEs) since the sequential scheme has some limitations. A selection scheme to squash threads in case of misprediction. They showed that the dynamic thread resizing approach has 11.6% greater performance than the conventional speculative multithreaded processors.

In 2002, Ungerer et al. surveyed and classified the various multithreading techniques in research and in commercial microprocessors [2]. As I mentioned in chapter 1, they classified the multithreading techniques into Interleaved multithreading technique, blocked multithreading technique and simultaneous multithreading.

Also Ungerer et al. talked about two types of multithreading architectures. (1) Implicit multithreading in which several threads are extracted from a single sequential

program with or without the help of the compiler and these threads are executed concurrently. A thread in this architecture refers to any contiguous region of the static or dynamic instruction sequence. In 2003, Park et al. [5] proposed an implicitly-multithreaded processor (IMT) which executes compiler speculative threads from a sequential program on a wide issue SMT pipeline. They showed that IMT outperforms on aggressive superscalar and the two prior proposals TME [6] and DMT [7]. (2) Explicit multithreading in which the processor interleaves the execution of instructions of different threads of control in the same pipeline.

In 2001, IBM introduced Power4-based systems in which two processor cores have been integrated on a single chip. In 2004, they introduced the Power5 processor [8] as a next generation after Power4. Power5 is a dual-core multithreaded processor; its processor core supports both enhanced SMT and single-threaded (ST) operation modes. Power5 provides higher performance in the ST mode than Power4 at equivalent frequencies. Power5 has some enhancements over Power4 processor like dynamic resource balancing, software-controlled thread prioritization and dynamic power management. The multithreading approach in Power5 is two-way SMT on each of the chip's two processor cores.

In 2005, Sun Microsystems developed the Niagara processor [9] which is a multithreaded processor designed to provide high performance for commercial server applications. The Niagara processor is an entirely new implementation of the SPARC V9. This processor supports 32 threads of execution such that these threads are organized in groups of four so the processor has eight thread groups. The threads in each thread group share one processing pipeline and there is a fair thread selection policy such that the least

recently used thread is selected. This kind of architecture helps in hiding the latency of the memory access such that when one thread is stalled because it made a memory request, then another thread is selected. The context switch in Niagara processor has a penalty of zero cycles.

Lindholm et al. [10] described the Tesla architecture that was introduced in 2006 in the GeForce 8800 GPU. Tesla architecture is based on a scalable processor array. GeForce 8800 GPU consists of 128 streaming-processor (SP) cores organized as 16 streaming multiprocessors (SMs). The 16 multithreaded processors are also organized in eight independent processing units called texture/processor clusters (TPCs). Tesla architecture is scalable and it achieves high throughput for the throughput applications which have extensive data parallelism, intensive floating- point arithmetic, modest task parallelism and modest inter-thread synchronization.

The SIMT architecture has been introduced in SM which creates, manages and executes threads in groups of 32 parallel threads called wraps. The threads in the same wrap are of the same type and they start from the same address but during the execution they are free to branch independently.

Tesla SM manages a pool of 24 wraps which have a total number of 768 threads. At each time an instruction is issued, the SIMT multithreaded instruction unit selects a ready wrap and broadcasts the SIMT instruction to the active parallel threads of that wrap. Moreover, Tesla introduced the cooperative thread array (CTA) which is an array of threads that execute the same thread program and can cooperate to compute a result.

In 2008, Latorre et al. [11] studied the synergies and trade-offs between the clustering in SMT processors. They proposed a novel resource assignment scheme for the clustered approaches and this scheme improved the performance by 17.6% compared with the Icount and it improved the fairness by 24%.

In 2010, Li et al. [12] adopted the hardware context switch driven by external events in multithreaded processors that are used in IP-Packet processing. The proposed processor has only one hardware context that can support multiple program counters which belong to different threads and there are tags to distinguish among the instructions of different threads in the pipeline. They showed that their proposal improved the overall performance by almost 3.8 times greater than the baseline structure while the area has increased by only 7%.

2.2 Vector Processors

Single instruction multiple data (SIMD) is the key aspect of vector processors [13], [14], [15]. In this type of processors, the operands are vectors so there is a need to have vector register file; the same instruction has to be executed on multiple elements of the vector simultaneously.

Kozyrakis et al. [16] proposed a novel micro-architecture called CODE (Clustered Organization for Decoupled Execution) to overcome the following limitations of the conventional vector processors, namely:

1. Complexity of multi-ported centralized vector register file VRF: since it stores a large number of vector elements. It supports high bandwidth communication of operands among the vector functional units so it should have large number of ports. To overcome this problem, they proposed the clustered vector register file CLVRS such that each cluster has a partition of the vector register file storing the operands of the local vector functional unit. The number of ports of the CLVRF is independent of the number of clusters and it is five; 2 ports of operands read and one for result write, 2 ports are used for intercluster communication one for input and the other for output.
2. The difficulty of implementing precise exceptions for vector instructions.
3. The high cost of on-chip vector memory systems.

They claimed that CODE is scalable up to eight functional units and it can hide the latency of off-chip memory access. In the cluster organization they proposed, each cluster consists of a single vector functional unit and a small number of vector registers.

In [17], Kozyrakis et al. talked about scalable vector processors for embedded systems. To demonstrate that vector architectures meet the requirements of embedded media processing, they evaluated the Vector IRAM (VIRAM) architecture using benchmarks from the Embedded Multiprocessor Benchmark Consortium (EEMBC). VIRAM architecture is a complete load-store architecture defined as a coprocessor extension to the MIPS architecture. Vector load and store instructions support the three common access patterns: unit stride, strided, and indexed (scatter/gather).

The elements in the VIRAM vector can be 64, 32 or 16 bits wide. VIRAM uses flag registers to support the predicated execution of instructions. Also it implements speculative vectorization of loops with data-dependent exit points. Figure 2 shows VIRAM prototype processor.

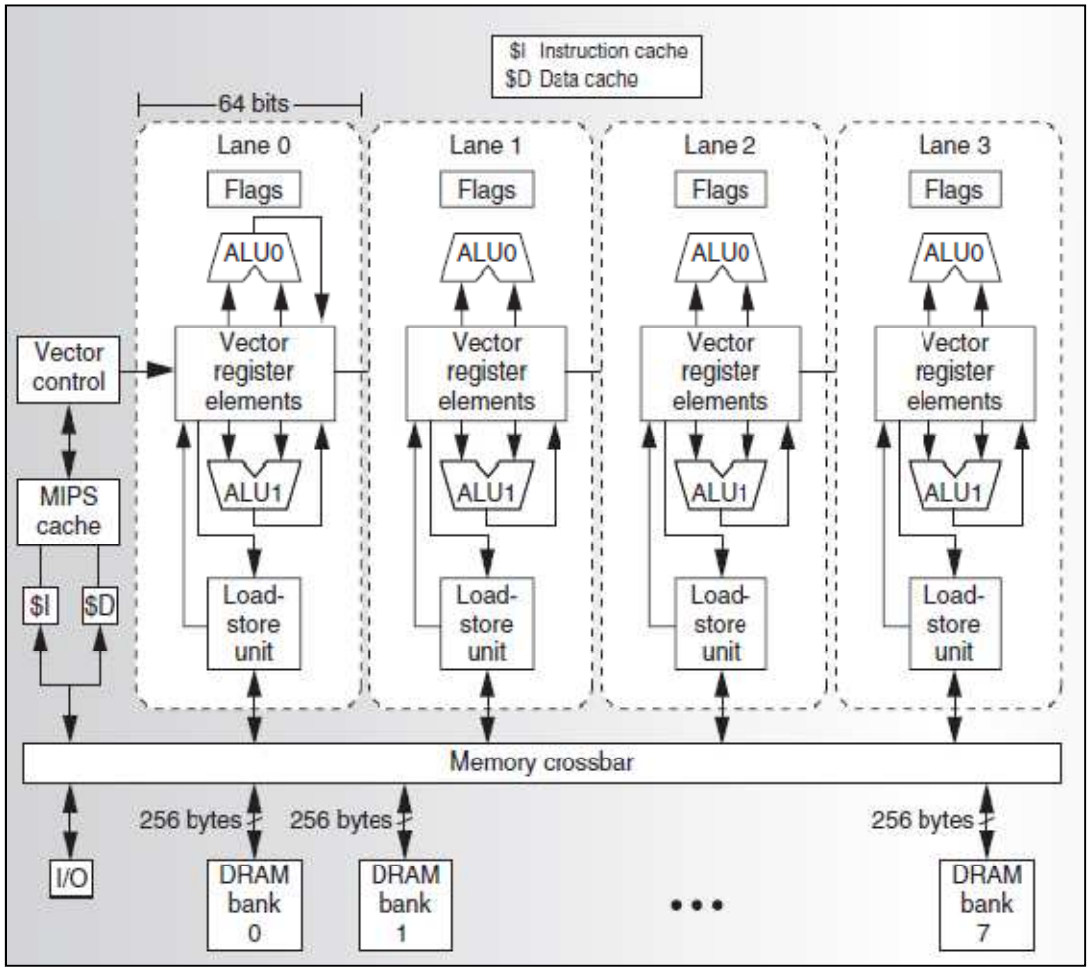

Figure 2: VIRAM Prototype Processor [17]

From figure 2, we notice that the processor consists of four lanes. Lanes make the processor scalable in performance, power dissipation and complexity. The drawback of

this architecture is the complexity of VRF partition within each lane, since the number of ports increases as the number of functional units increases. They used clustering to reduce the VRF complexity and improve the performance. Each cluster contains a data path for a single vector functional unit and a few vector registers as shown in figure 3 but they still need an inter-cluster network to communicate among the clusters.

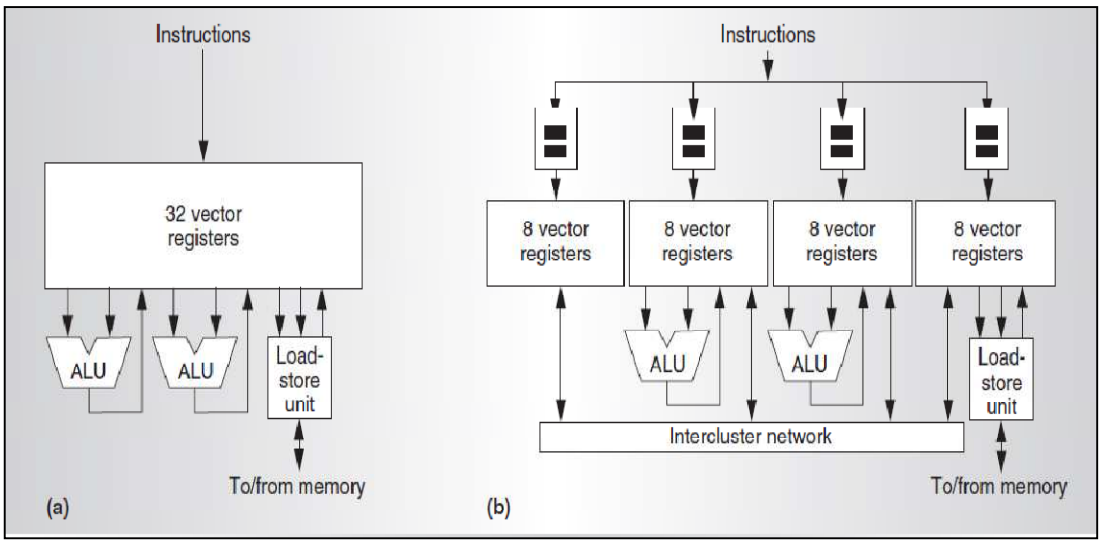

Figure 3: Vector Lane Organization: Centralized (a) Clustered (b) [17]

Krashinsky et al. [18] introduced the thread-vector VT architecture to unify the vector and multithreaded compute models, so the VT architecture is a hybrid of these two models. The VT abstraction provides the programmer with a control processor and a vector of virtual processors VPs. The control processor can broadcast the instructions to the VPs using the vector-fetch or the VP can use thread-fetching to direct its control flow. In this architecture, there are two interacting instruction set architectures: one for the control processor and the other is for the VP.

Each VP has a set of registers and ALUs. VP instructions are grouped in atomic instruction blocks (AIBs). The block must be requested explicitly by the VP or the control processor because there is no automatic program counter or implicit instructions fetch mechanism. They focused on an instantiation of the VT architecture called SCALE. It was designed for high performance and low power consumption for embedded systems.

2.3 Processors Frontend

Each processor consists of two parts: the frontend and the backend or the execution engine. The frontend job is to feed the backend of instructions to be executed. The appearance of superscalar processors that exploits ILP has placed more pressure on the frontend to provide instructions in a higher rate so as to exploit the hardware resources in the backend. For this reason, many complex frontend architectures have been proposed. These frontends implement pre-fetching mechanisms for higher instruction fetch rates and also they implement branch-prediction mechanisms. Many instruction pre-fetching techniques like [19], [20], [21], [22] have been proposed to increase the instruction fetch rate.

Also some people proposed instruction streaming as a way of providing the execution engine with instructions in a smooth way and some people exploited the block-structured ISA for this purpose. For example, in 2002, Ramirez et al. [23] proposed novel fetch architecture called next stream architecture. This architecture is based on the execution of long streams of sequential instructions exploiting the code layout

optimizations. In this architecture, they focused on simplifying the frontend design while getting a performance that is close to the state-of-art.

The instruction stream is defined as a sequence of instructions from the target of a taken branch to the next taken branch which can contain multiple basic blocks. The instruction stream is identified by the starting instruction address and the stream length. In this architecture, the instruction stream is the unit of fetching and it directly maps to the structures of the high-level programming constructs. These things make the design complexity under control.

For wide issue processors, the next stream predictor performance was 10% higher than the EV8 fetch architecture and 4% higher than the FTB fetch architecture. It was 1.5% slower than the trace cache architecture but with less design complexity.

In 2007, Santana et al. [24] proposed to enlarge these instruction streams to get significant performance in a new mechanism called multiple-stream predictor that combines single frequently executed streams into long virtual streams regardless the type of these single streams. This predictor provides predictions that contain on average 20 instructions. It doesn't need hardware overriding mechanism to hide the branch prediction table access latency.

In 2004, HE et al. [25] proposed an IPC-Based Fetch Policy (IPCBFP) for SMT processors; IPC refers to the number of instructions executed per clock cycle. This policy fetches instructions for any running thread depending on the instantaneous value of the IPC and the number of instructions in the instruction queue for that thread. So this policy should be able to count the number of instructions for each thread in the instruction queue

and to approximate the current value of the IPC for each thread. This policy selects the two threads with the least instructions in the instruction queue and feeds as many as needed number of instructions to every selected thread up to eight in total.

2.4 Block-Structured ISA

In the block-structured ISA, the program consists of basic blocks. This kind of architectures appeared in 1990s to increase the instructions fetch rate in the wide issue processors to utilize the ILP and the hardware resources. Hao et al. [26] defined an instance of a block-structured ISA for a wide issue dynamically scheduled processors. They constructed a compiler to generate a block-structured code. They used an optimization technique called block enlargement in which multiple basic blocks are combined together in one larger basic block and they forced the atomic execution of the block to reduce the hardware complexity.

Hao et al. showed that for SPECint95 benchmarks that the block-structured ISA processor executing enlarged atomic blocks outperforms a conventional ISA processor by 12% with less hardware complexity.

In 2006, Zmily et al. [3] proposed a block-aware instruction set architecture called BLISS to address the basic challenges of frontend for wide issue high frequency superscalar processors. BLISS defines basic block descriptors in addition and separately from the actual instructions of each program, so the code segment of each program consists of two sections; the basic block descriptors and the actual instructions of the

program. The basic block BB is defined as a sequence of instructions starting at the target or fall-through of a control flow instruction and ending with the next control flow instruction or the next potential branch target.

The descriptor consists of several fields to store sufficient information about the basic block of instructions. Figure 4 shows a 32-bit BB descriptor:

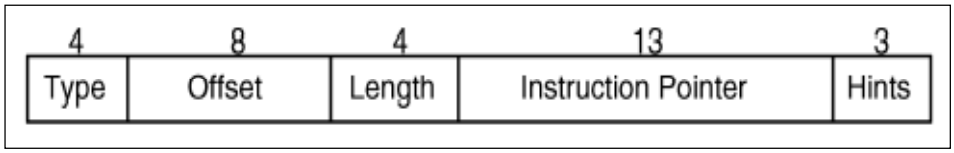

Figure 4: The 32-Bit Basic Block Descriptor Format in BLISS [3]

The descriptor appearing in figure 4 contains the following fields:

Type: The basic block type i.e. type of the terminating branch.

Offset: Displacement of program counter (PC)-relative branches and jumps.

Length: Number of instructions in the BB

Instruction Pointer: The address of the first instruction in the BB

Hints: Hints generated by the compiler to help the processor takes some decisions like control-flow predictions and instruction fetch at runtime in a more efficient way. Hints aim at balancing the load overhead between the hardware and the software to simplify the hardware and reduce the power consumption.

The execution of the BB is atomic to simplify the hardware and the software. The naive generated is larger than the assembly code because of adding descriptors. But this code can be highly optimized in multiple ways like removing the jump instructions because they provide no information with the existing of the descriptors. Also the repeated instruction sequences can be removed.

Based on this architecture, Zmily et al. proposed a simple decoupled frontend for the superscalar processor by replacing the branch target buffer with a BB-cache that caches the block descriptors in programs. They demonstrated that BLISS has achieved 20% performance improvement and 14% total energy savings over conventional superscalar design. It also achieved 13% performance improvement and 7% total energy savings over aggressive frontend that dynamically builds fetch blocks in hardware.

2.5 Simulation

Simulation is very important for computer architects because it is more flexible and it has low cost. It helps the architect to explore the design space and to find the optimal design. Although the simulators are very helpful, the architects may suffer because in some cases the simulators are very slow and they have poor accuracy. In 2006, Yi et al. [27] surveyed the existing methodologies and techniques for cycle-accurate simulation. They stated that the accuracy is affected by four factors. (1) Simulator's accuracy. (2) The soundness of the simulation methodology. (3) The representativeness of the benchmarks. (4) The simulation technique that is used.

Yi et al. classified the simulators that they surveyed into five classes. (1) Single-processor performance simulators such as SimpleScalar simulator. (2) Full system simulators like Simics simulator. (3) Single-processor power consumption simulator like Wattch simulator. (4) Multiprocessor performance simulators like Rice Simulator ILP Multiprocessors (RSIM). (5) Modular simulators like Liberty Simulation Environment (LSE).

Yi et al. defined the simulation process as the sequence of steps that the architect must perform to run and analyze the simulation. They divided the simulation process into six steps. (1) Simulator validation and accuracy. In this step, the simulator must be validated before the results can be trusted. (2) Processor enhancement, implementation and verification. (3) Selecting processor and memory parameter values. (4) Selecting benchmarks and input sets. (5) Simulation; in which the benchmarks are run on the configured simulator. (6) Performance analysis. In this final step, the obtained results are analyzed to see the effect of the enhancement.

For some benchmarks, the input data sets are very large and therefore the simulation time is very long. To address this problem, there are several techniques like reducing, truncating and sampling the input sets.

In 2008, Cho et al. [28] proposed a simulation framework called Two-Phase Trace-driven Simulation (TPTS). The motivation was increasing the simulation speed through splitting the detailed timing simulation into two phases. (1) Trace generation phase. (2) Trace simulation phase. In the trace generation phase, they use a filtering

technique in order to avoid the need for simulating uninteresting architectural events in the repeated simulation phase.

In 2010, Ubal et al. [29] proposed a simulation framework called Multi2Sim to evaluate multi-core multithreaded processors. They claimed that this simulation framework was intended to cover the limitations of the existing simulators and it models the major components of the incoming systems.

Since developing a cycle-accurate simulator is a complex and timing-consuming task, there are techniques like Architecture Description Languages (ADLs) that are used to provide an abstraction layer for describing the computer architecture and generating simulators for these architectures automatically. In [30], they presented an XML-based ADL that receives the functional description of the architecture in XML format and generates the corresponding multithreaded simulator.

2.6 Discussion

In this chapter, I have explored the related work to my research. One of the related architectures to the PAR core is vector architecture. In vector processors, once an instruction is issued, it is executed on a vector of data in parallel; so many functional units are needed, whereas in PAR core the single instruction is executed for multiple threads sequentially on the same functional unit. Also it is possible to replicate the processing lane in PAR core and have higher throughput.

Vector processors need to define vector instructions and vector register file. However, PAR core uses scalar instructions to execute instructions for multiple threads. In addition to that, the frontend of PAR core is more dynamic and more flexible such that the threads can expand and terminate loops independently. Moreover, the vector processor is a standalone processor while PAR core is a part of a larger system which contains master cores to run the master processes. So the PAR core is integrated within a multi-core system and it executes the parallel threads assigned from the master core.

In the recent multithreaded processors, NVIDIA TESLA architecture is the closest one to PAR core. The main similarity between them is that both of them are SIMT processors. Moreover, the idea of having multiple multithreaded lanes such that each lane manages a group of threads is similar to thread grouping proposed in Niagara processor. One of the main differences between the two architectures is that the single processing pipeline in Niagara processor supports multiple threads with different instruction streams, whereas the PAR core's lane executes the same sequence of instructions for a number of times equals the number of thread group per lane.

Regarding the frontend complexity, PAR core's frontend is simpler than the previous proposals; it doesn't contain branch prediction and instruction pre-fetchers. There are two reasons that make PAR core's frontend simple. (1) The features of the PAR ISA like the stop bit and the control stack. (2) The nature of the SIMT architecture which doesn't place a pressure on the fetch unit, because once an instruction is issued it does a lot of work so there is no need for a high fetch rate.

Regarding the block-structured ISA, they have been proposed to increase the instructions fetch rate. PAR ISA allows having a high instruction fetch rate but with simpler fetch unit because the need for branch prediction can be eliminated. Also PAR ISA doesn't need to store information about the instruction blocks as in BLISS ISA [3] because of the existence of the stop bit which marks the end of the instruction blocks.

Regarding simulators, ParSim is a cycle-accurate execution-driven simulator. The goal of ParSim was to show the throughput and the scalability of the proposed architecture. My experiments showed that the number of threads specified in a PAR packet doesn't affect the performance results like IPC and speedup if the number of threads exceeds a certain limit. This limit was not large and then if the benchmark has a huge number of threads, then these threads can be truncated and the simulation time will be in minutes or even in seconds.

CHAPTER 3 PAR CORE HARDWARE MODEL

3.1 Overview

PAR core is a multi-lane multithreaded processor core i.e. it consists of multiple lanes such that each lane is multithreaded. This core is intended to receive a PAR packet generated by a master process running on a separate core called the master core and executes the threads specified by this packet simultaneously. The PAR packet's threads are completely independent, so the communication overhead among them is zero and thus the ideal speedup is expected. This kind of threads can be found in data-parallel applications in which the elements of an array or a matrix are processed independently; these applications can be found in multimedia processing and scientific computing areas.

PAR core can be described as a single instruction multiple threads (SIMT) processor since once an instruction is issued, it is executed for multiple threads. This kind of architecture reduces the pressure on the frontend and reduces the hardware cost because it eliminates the need for having a wide issue frontend, branch prediction policies and instructions' pre-fetching.

Figure 5 shows a high level block diagram of the PAR core. From this figure, it is clear that there is only one fetch unit for all lanes because all threads will execute the same instruction. Since the thread program is going to be executed multiple times, may be thousands or millions, the level one instruction cache should be large enough to store all instructions of the thread program. Usually the number of instructions per PAR packet's

thread program is not large, so an instruction cache with reasonable size will be able to store all of these instructions and therefore the instruction cache miss will be zero after brining all of these instructions from the main memory or the lower level cache memory to the first level instruction cache.

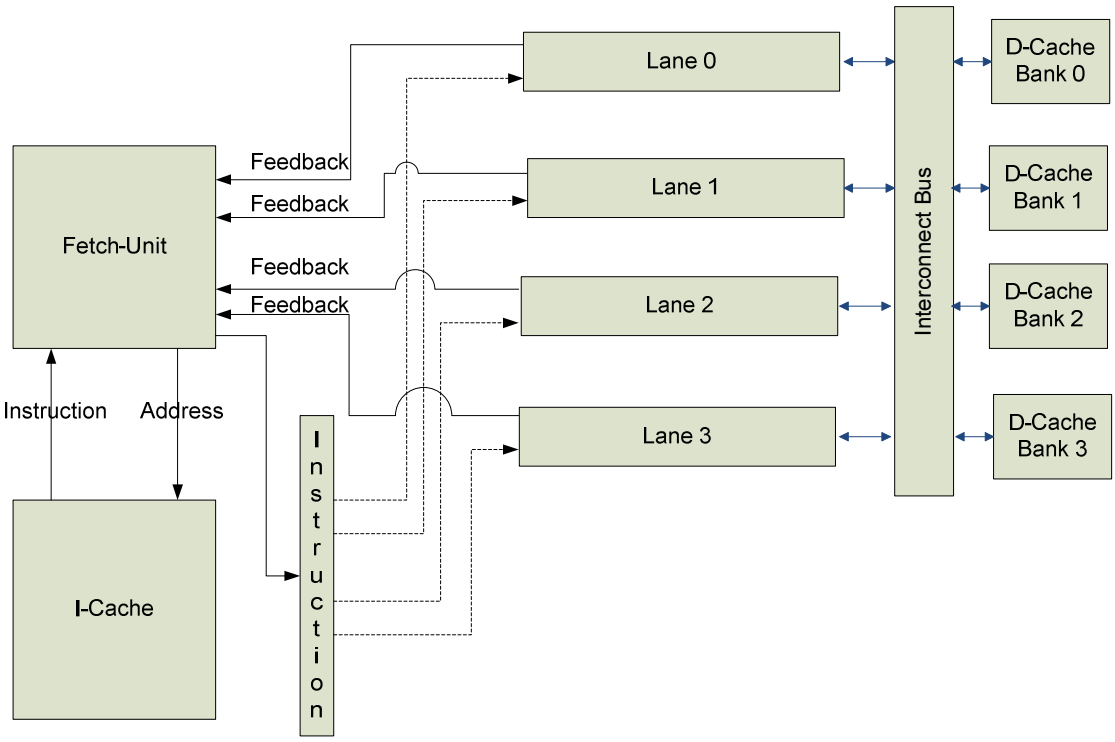

Figure 5: High Level Block Diagram of PAR Core

Besides that, PAR core has many characteristics such as it is a single issue, in-order issue and an in-order commitment processor. Despite the fact that different instructions may complete out of order because they have different latencies, their results are written back in order to guarantee the correctness of the program's results. Most of these characteristics make the PAR core simpler in terms of hardware complexity.

3.2 Assumptions

1. The level one instruction cache is assumed to be large enough to contain the whole thread program. This assumption is reasonable because the thread program is most probably small enough to fit in an instruction cache with an acceptable size. This assumption results in a zero cache miss rate except the cold start misses. It is necessary to realize this assumption because the thread program will be executed many times, so caching the whole thread program results in higher performance.

2. In PAR core, the data cache has been replaced by a memory module called the local memory. Currently, this local memory is treated as a black box and it will be added to the PAR core later. It has been assumed that the local memory miss rate is zero. This assumption can be realized up to certain point thanks to the global load and global store instructions that will be implemented later. Global load and global store instructions are intended to allow data exchange between local memories and the global memory in bulks and they will be implemented in a way to allow communication-computation overlapping.

3. For PAR packet's threads that contain counter-controlled loops, it has been assumed that all threads have the same loop counter. This assumption simplifies the hardware because only single counter has to be maintained per counter-controlled loop. This assumption will not prevent threads from having the freedom to exit loops independently.

3.3 The Multithreading Approach

PAR core implements two multithreading techniques. (1) The simultaneous multithreading approach in which different independent threads run simultaneously on multiple lanes; but this is different from the traditional SMT technique because in this technique only one instruction is fetched and executed for multiple threads. (2) The interleaving approach which has been implemented within the single lane. Inside each lane, the threads interleave with each other since the instruction is executed multiple times sequentially such that each time it is executed for another thread. This interleaving technique is different from the traditional cycle-by-cycle interleaving technique because in this technique the instruction is fetched, issued only once and it is executed for multiple threads.

Since all instructions including the control instructions are predicated, then it is natural that some threads within the same lane are going to expand and some of them are not, some of threads are going to break a loop and some of them are not etc. So it is important to give each lane this kind of flexibility. To implement that, each lane has a mask register. The width of this register equals the number of simultaneous threads supported by the lane. Each bit in the mask register corresponds to one thread. The mask register plays the role of the top controller of the lane. The thread is active if its corresponding bit in the mask register is true and it is inactive otherwise. For an instruction to write back its result and to affect the state of the system, its corresponding bits in the mask register and the qualifying predicate register must be true.

3.4 Why is PAR Core Optimized?

PAR core has many features that make it optimized in terms of performance and hardware complexity. These features are the following:

1. There is no context switch overhead. PAR core repeats the execution of the same instruction multiple times, so it doesn't need to switch among different threads of different instruction address spaces.
2. Since PAR core is an SIMT, then the functional units' utilization should be high because the issued instruction does a lot of work because it is executed for multiple threads.
3. Control instructions' overhead is reduced. The control instructions are predicated and the qualifying predicate register's value may not be ready at the fetching time, so the fetch unit has to stall until the value of the qualifying predicate register becomes ready. Since the instruction is fetched once for multiple threads, then the overhead associated with the control instructions and the fetch unit in general is reduced.
4. There is only one and a light weight fetch unit for the PAR core. The fetch unit is light because it has no branch prediction, no wide issue and no instruction pre-fetching policies.
5. The instructions are issued in-order. So the logic needed to maintain out of order execution has been eliminated.

6. Since the floating point unit (FPU) is pipelined, running the same FP instruction sequentially for multiple independent threads guarantees that the FPU is highly utilized and the FP instructions' latency is hidden.
7. If the load instruction is going to load for multiple threads from the same address, then this load instruction is executed only once and its result is replicated.
8. There is no instruction cache miss except the cold start misses, because the level one instruction cache must be large enough to store all instruction blocks of the thread program.

3.5 Frontend Structure

The frontend is responsible for providing the execution engine with instructions in a reasonable rate. The reasonable rate is the rate that makes the computing resources in the backend busy. Figure 6 shows a block diagram of the PAR core's frontend. PAR core's frontend is decoupled from the backend. The backend receives instructions from the frontend and provides it with information like the loop counter, the qualifying predicates registers' values, mask register and the stall signal. The feedback port shown in figure 6 abstracts all information sent from the backend to the frontend except the loop counter. In this section, the structure of the frontend will be described. From figure 6, it is noticed that the frontend consists of the following components:

1. **Instruction cache**: it caches the instructions of the thread program. The instruction cache must be large enough to store all of these instructions because they are going to be executed many times.
2. **Control Stack:** it is used to schedule the execution of the program. It can be split into two independent stacks: the counter control stack which saves the loops' counters and the remaining number of threads to be executed and the command control stack which stores the commands that are used to guide the fetch unit in how the PC should be updated. These commands include:

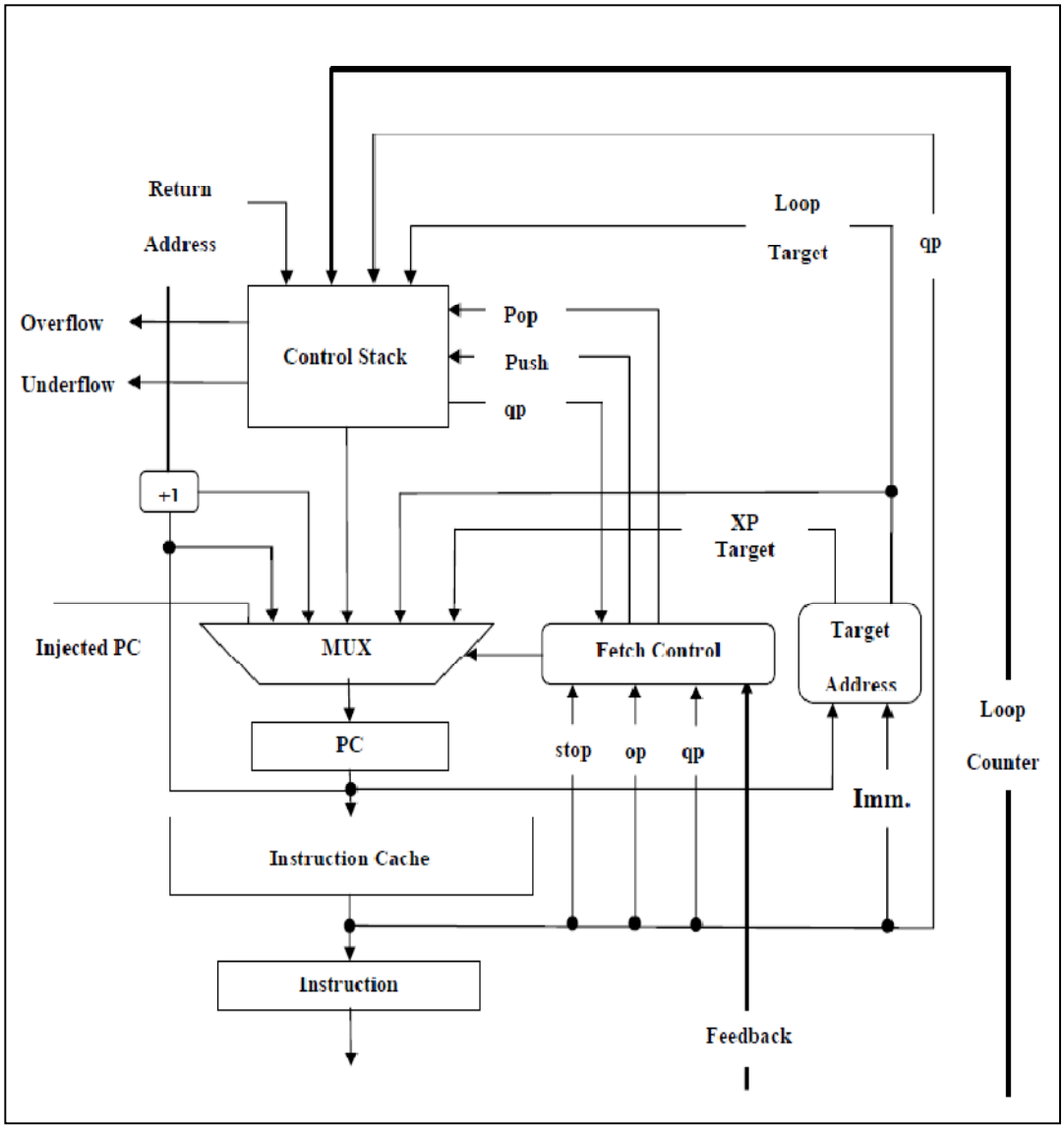

Figure 6: Block Diagram of PAR Core's Frontend

- **The PAR Command:** this command entry is pushed on the command control stack upon the reception of the PAR packet. In this entry, the starting address of the thread program is saved. Besides that, a counter entry is pushed on the counter control stack to indicate the number of threads to be executed. Each time the PAR command becomes the top entry of the command control stack, this means that the

end of the thread program has been reached and therefore the top entry of the counter control stack is checked. If the counter value is greater than the number of simultaneous threads supported by PAR core, then it is decremented by the number of simultaneous threads supported by PAR core, say 16 threads if the PAR core has four lanes and each lane is 4-way multithread, and the PC is set to the starting address of the thread program. If the number of the remaining threads is greater than zero and less than or equal the number of simultaneous threads supported by PAR core, then the counter is set to zero and the PC is set to the starting address of the thread program. When the PAR command becomes the top command on the command control stack and the corresponding counter is zero, then this indicates that the PAR packet's execution has finished and both stacks are freed.

- **The Return Command:** this command is used to save the return address and the mask registers. If an expand or a loop instruction is encountered and the stop bit of this instruction is cleared, then the return address and the mask registers are saved within the return command which is pushed on the command control stack.
- **The Loop Command:** this command is used to schedule the loop instruction execution. If a conditional counter-controlled loop is encountered, then the starting address of the loop block and the qualifying predicate register number are stored within the loop command which is pushed on the command control stack. Regarding the loop counter, it is saved on the counter control stack. If a pure conditional loop instruction is encountered, then it is treated in the same manner as the counter-controlled loop except that in this case there is no counter entry.

There are two fields common among the control commands. (1) The command type field which tells if the command is a PAR, counter-controlled loop, pure conditional loop or a return command. (2) The address field. In PAR command, the address field is the starting address of the thread program. In loop command, it is the starting address of the loop block. Finally, it is the return address in the return command.

Besides that, the return command contains an additional field which stores the mask registers of PAR core's lanes. Also the loop command has a field to store the qp register number of the loop instruction. The control command entry size should equal to the size of the longest command which is the return command.

3. **Fetch Control:** this component is the heart of the fetch unit, it controls the PC update and it detects the program termination. The PC can be updated in different methods according to the type of the fetched instruction, the stop bit value and the top entry of the command control stack. The following are the different cases of updating the PC:
    - The PC is injected from outside within the PAR packet. This PC value is the starting address of the thread program. This is done only once per PAR packet.
    - The PC is not updated and this happens when there is a stall.
    - The PC is simply incremented and this happens when the fetched instruction is a non-control instruction and its stop bit is cleared.

- The PC is set to the starting address of the thread program and this occurs when the end of the thread program is reached and the number of the remaining threads is non-zero.
- The PC is set to the return address popped off the command control stack. This happens when the execution of an instruction block terminates and the top entry on the command control stack is a return command entry.
- The PC is set to the starting address of the loop block for the first qualified loop iteration and each time the loop block finishes and there is still at least one loop iteration remaining.

3.6 How Does the Frontend Work?

The frontend contains the fetch unit that fetches instructions from the first level instruction cache and feeds them to the execution engine to be executed, but not all types of instructions are executed by the execution engine. Only the non-control instructions are fed to the execution pipeline and the control instructions are executed in the fetch unit. Before the fetched instruction is sent to the execution engine, it is checked by the fetch control to determine its type and to decide how to update the PC. The fetch control reads the opcode, the stop bit and the qualifying predicate register number of the fetched instruction on the flight while the fetched instruction is being transferred from the instruction cache to the instruction register. Checking the instruction's opcode determines the instruction's type and guided by the stop bit, the qualifying predicate register and the

control stack the fetch control decides whether to send the instruction to the execution pipeline or not and decides how to update the PC.

In this section, I will describe the two scenarios of the fetch control behavior. These two scenarios are:

1. If the fetched instruction is a non-control instruction, then it is broadcasted to all lanes and the PC is updated according to the value of the instruction's stop bit and the type of the topmost command on the command control stack. If the stop bit is cleared then the PC is simply incremented. If the stop bit is set, then the command and counter control stacks are checked and the PC is updated accordingly. The following are the possible cases in updating the PC when the fetched instruction is a non-control instruction and its stop bit is set:
    a. If the topmost command of the command control stack is a return command, then the PC is set to the return address.
    b. If the topmost command of the command control stack is a counter-controlled loop command, then the qualifying predicate register and the topmost counter on the counter control stack are checked. If the counter is non-zero and the qualifying predicate register value ANDed with the current mask registers is true for at least one thread, then the loop counter is decremented and the PC is set to the loop target address. Nevertheless, if the counter is zero or the qualifying predicate register value ANDed with the current mask registers is false for all threads, then the counter and the loop command entries are popped off the stacks and the PC is updated according to the new topmost command.

c. If the topmost command on the command control stack is a pure conditional loop command, then the qualifying predicate register is checked. If the qualifying predicate register value ANDed with the current mask registers is true for at least one thread, then the PC is set to the loop target address. Nevertheless, if the qualifying predicate register value ANDed with the current mask registers is false for all threads, then the loop is terminated and the loop command entry is popped off the stack and the PC is updated according to the new topmost command.

d. If the topmost command on the command control stack is a par command, then the topmost counter on the counter control stack is checked. If the counter is non-zero, then it is decremented by the number of simultaneous threads supported by PAR core and the PC is set to the starting address of the thread program. However, if the counter is zero, then this indicates the termination of the program and the PAR core is freed.

2. If the fetched instruction is a control instruction, then it is processed in two stages: the first stage includes fetching it from the instruction cache and checking the opcode, the stop bit and qualifying predicate register and this stage is common among all kinds of instructions. The second stage includes reading the qualifying predicate register associated with it. If the qualifying predicate register value ANDed with the current mask value is false for all threads, then the control instruction is skipped. However, if the qualifying predicate register value ANDed with the current mask value is true at least for one thread, then the control instruction will have an effect and it will change the execution flow of the program.

If the qualifying predicate register value is not ready because it is being produced by another instruction, then the fetch unit has to stall until the qualifying predicate register value becomes ready.

In the rest of this section, I will talk about how the control instructions are implemented in PAR core.

a. Direct Expand Instruction (xp)

For each lane, the new mask register's value is calculated by bitwise ANDing the current mask register with the qualifying predicate register of the xp instruction. If the new mask value is true for at least one thread, then the execution flow is transferred to the target address of xp. If the stop bit of this instruction is cleared, the return address and the old mask are saved on the command control stack. However, if the stop bit is set, then nothing is pushed on the command control stack.

The two stages of executing the xp instruction take place in two clock cycles. (1) The instruction is fetched and the PC is updated to the target address. (2) The qualifying predicate register is read and the first instruction of the target block is fetched. If the xp is qualified, then the flow of execution proceeds. If the xp instruction is disqualified, then the fetched instruction is ignored and the PC is updated to the return address popped off the control stack.

b. Counter-Controlled Loop Instruction

For PAR core, it has been assumed that the value of the loop counter is unified for all threads and it is specified in an inherited register. For each lane, the new mask register's value is calculated by bitwise ANDing the current mask

register's value with the qualifying predicate register of the loop instruction. This instruction is qualified if the mask register is true for at least one thread and it is skipped otherwise.

When the loop instruction is qualified, a loop command entry is pushed on the command control stack and a counter entry is pushed on the counter control stack. The command entry specifies that this command is a counter-controlled loop command; it contains the qualifying predicate register number associated with this loop instruction and the starting address of the loop block. The counter entry simply contains the number of the remaining loop iterations.

If the loop instruction's stop bit is cleared i.e. it is in the middle of an instruction block, then its effect is loop and continue. To implement that, a return command is pushed on the command stack before the loop command is pushed. This return command saves the return address and the current mask registers value.

At the end of each loop iteration, the mask register is updated to reflect any possible changes on the qualifying predicate register within the loop body. Besides that, the loop counter is decremented and the PC is set to the starting address of the loop block. The loop continues until the loop counter becomes zero or the mask register becomes false for all threads. When the loop terminates, the loop command and the counter entries are popped off the stacks and the execution resumes at the address specified by the next entry on the command control stack.

c. Pure Conditional Loop Instruction

This loop is a pure conditional loop i.e. it is not controlled by a counter. It continues as long as the qualifying predicate register ANDed with the current

mask register is true at least for one thread. This loop is implanted as the counter controlled loop except that it has no counter.

d. Break Instruction

The break instruction that has been implemented in PAR core is used to terminate the loop instruction prematurely if its qualifying predicate register is true for all threads. If not all bits of the qualifying predicate register are true, then some threads are going to break and some threads are not going to break. When all threads are going to break, then the loop entries are popped off the control stacks and the PC is updated according to the next command on the command control stack. If not all threads are going to break then the mask register is updated by ANDing it with the bitwise complement of the qualifying predicate register associated with the break instruction.

3.7 Backend Structure

PAR core pipeline consists of six stages: instruction fetch, decode, dispatch, issue, execute and write back. Once a non-control instruction is fetched, it is sent to the backend for execution. Each lane has the same computational resources which include decode and dispatch logic, issue logic, forwarding network, write back logic, functional units and their instructions' queues, functional units' input buffers, functional units' output buffers, general purpose registers, predicate registers and inherited registers. Figure 7 shows a

block diagram of the execution pipeline for a single PAR core's lane. In this section, the functionality of each component in the backend has been described.

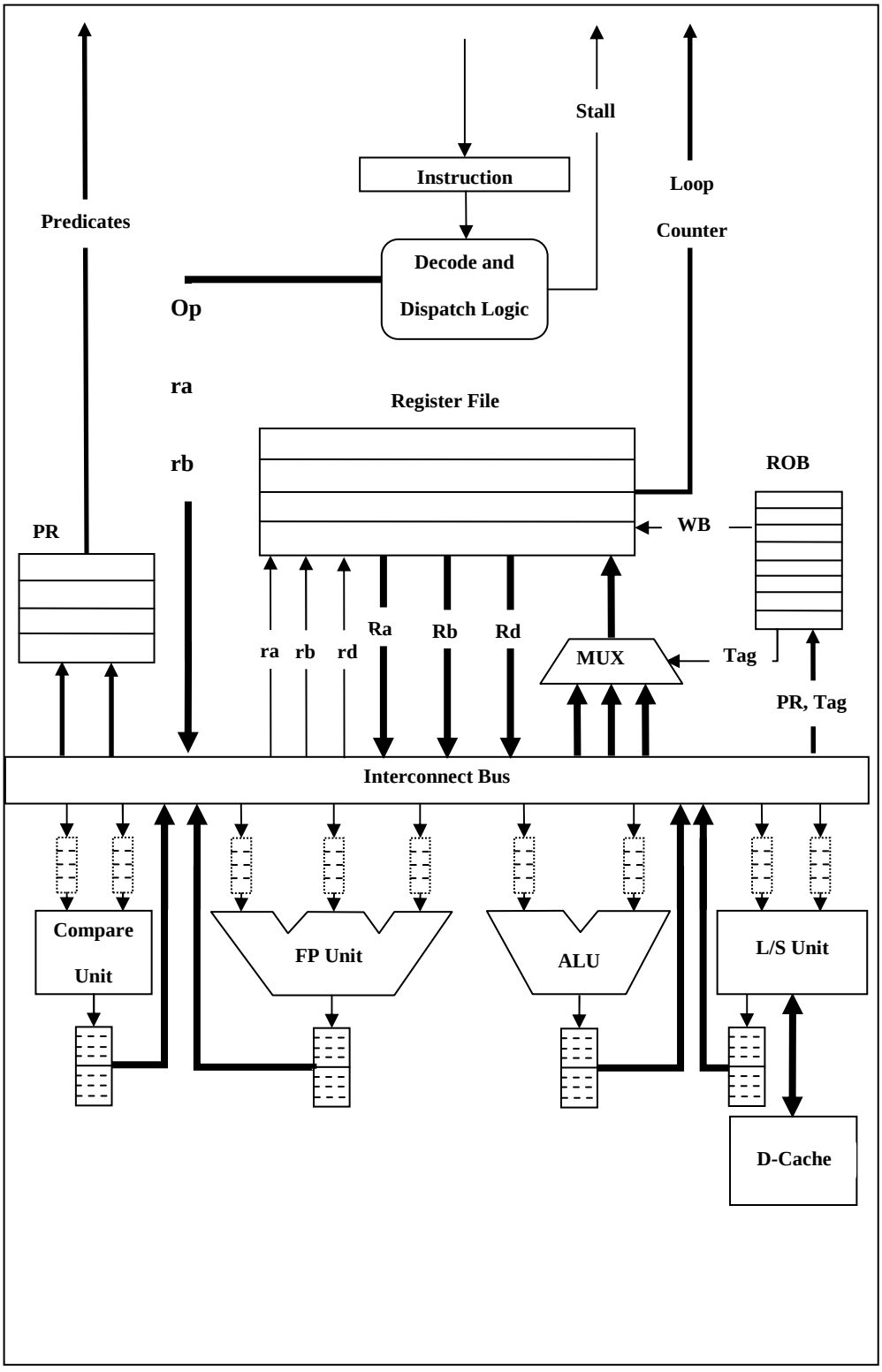

Figure 7: Block Diagram of PAR Core's Backend

3.7.1 Register File

Each lane has a wide general purpose register file such that the width of the register is proportional to the number of simultaneous threads supported by a single lane. Each thread has a register of 64 bits from that wide register as shown in figure 8.

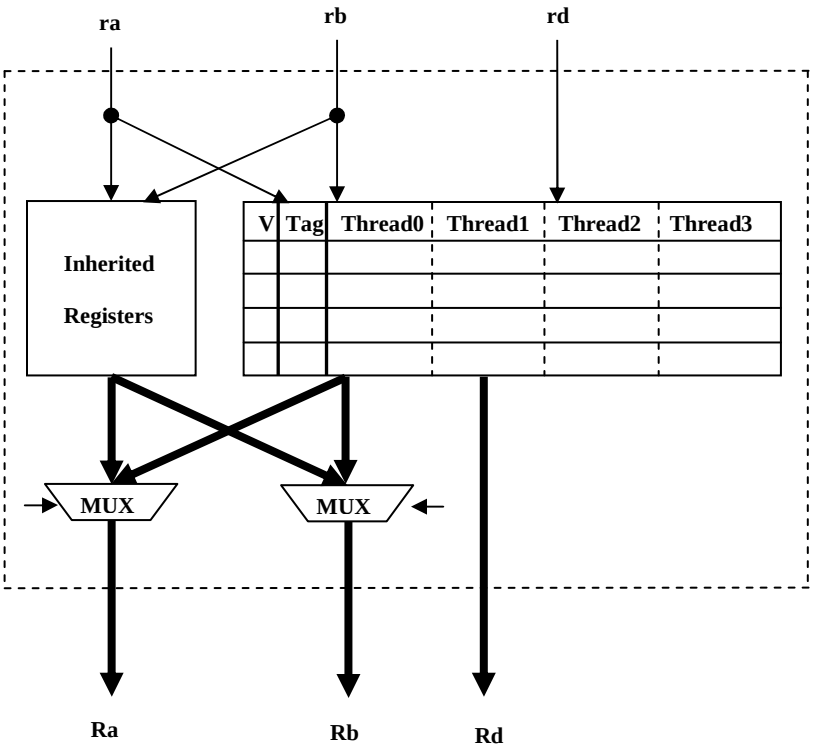

Figure 8: PAR Core's Register File

Besides the general purpose registers, there are inherited registers whose values are inherited from the master thread. The inherited registers are read only. They contain constant values which may represent base addresses, coefficients, loop counters, etc. The inherited register i0 contains the thread index and it is the only inherited register whose

value is not constant. This register is updated automatically each time the end of the thread program is reached

We notice from figure 8 that this lane is 4-way multithreaded. Besides the registers' contents, each register has two fields: the valid bit and the tag fields. The valid bit indicates whether this register's content is valid or not. If the register's content is invalid and it is being produced by another instruction, then the tag field is used to refer to the instruction that will write its result into this register. The tag field consists of two subfields. (1) The functional unit ID. (2) The instruction ID within that functional unit. The source operands' registers ra and rb can be either inherited or general purpose registers, so multiplexers are used to output from the right register file. The multiplexer is controlled via the most significant bit of the source registers numbers ra and rb. Regarding the destination register rd, it can be only a general purpose register since the inherited registers cannot be written.

3.7.2 Predicate Registers

Each lane in PAR core has eight predicate registers. Each predicate register has one bit corresponding to each thread within the lane. The predicate registers are used to qualify the instructions. In other words, if the qualifying predicate register associated with an instruction has a value of false then this instruction is dropped and its result is not committed. The predicate registers are updated via the compare instructions. There is a

valid bit associated with each predicate register to specify if its value is valid or it is being produced by a certain instruction.

3.7.3 Reorder Buffer (ROB)

It is a buffer used to reorder instructions' results writing back into the register file. The ROB's size should equal to the summation of the sizes of the instruction waiting queues in all functional units.

Each ROB's entry should contain the following fields:

1. The instruction's tag which consists of the functional unit ID that is executing this instruction and the instruction ID within that functional unit. The instruction ID actually is the index of the instruction within the instructions' waiting queue.
2. The destination register’s number.
3. The qualifying predicate register value is used to decide for which threads the destination register should be written and for which threads the destination register shouldn't be written. In other words, the qualifying predicate register bits ANDed with the mask register bits work as the write enable signals of the register file.
4. The tag of the qualifying predicate register, if the qualifying predicate register was invalid at the dispatch time because it was being produced by another instruction, then the tag of that instruction is stored within the ROB entry and once the predicate register is produced, then it is forwarded to the ROB.

5. A valid bit to indicate whether the qualifying predicate register's content is valid or not.

3.7.4 Functional Units

Each lane in PAR core has four different functional units that can work in parallel on different instructions. These functional units are:

- **ALU:** it executes integer addition and subtraction plus all different logic operations like and, or, xor, shift, etc.
- **FPU:** it executes integer and FP multiplication and division operations. Moreover, the FPU is capable of executing multiply/accumulate instructions for both integer and FP operands, so it has three inputs.
- **Load/Store Unit (L/S Unit):** this unit is responsible for executing memory instructions.
- **Compare Unit:** it is responsible for executing compare instructions like is equal? Is greater than? Etc. The output of this unit is written into the predicate register file.

There are common things among all of these four units; each unit has a waiting instruction queue, input buffers and output buffers. When an instruction is dispatched to its functional unit, it is queued in the waiting queue associated with that functional unit. Moreover, each functional unit has input buffers to buffer the operands of one instruction. For each operand there is a buffer whose size equals to the number of simultaneous

threads supported by a single lane. Each entry in this buffer has a valid bit to indicate whether this entry is valid or it is going to be produced later by some other instruction. For each operand, there is a tag that tells which instruction in which functional unit is going to produce this operand.

In addition to the input buffers, each functional unit has two output buffers to store the instructions' results temporarily. Storing the instructions' results is important for forwarding them among the functional units and also it enables writing back the instruction's result for all threads in one clock cycle. The reason for having two output buffers is if some instruction finishes execution and it cannot write back according to the ROB, then it keeps occupying the output buffer and the functional unit cannot overwrite its results and therefore it will stay idle. To solve this problem, the functional unit should have an additional output buffer and then another instruction can be issued.

3.7.5 Instruction Waiting Queue

There is a queue associated with each FU. This queue is intended to store the instructions that are waiting to be executed by this unit. This is important because it allows having multiple instructions in the backend simultaneously. The simulation results showed that the optimal size of this queue is two entries. Each entry contains the following fields:

1. Instruction's opcode.

2. Instruction's source registers numbers in order to read the contents of these registers at the issue time.
3. A valid bit corresponding to each source register to indicate if the register's content is valid and can be read from the register file or invalid and it should be read from the forwarding network.
4. A tag corresponding to each source register to specify which instruction is going to produce this register.
5. The immediate value if the instruction is I-Type.

3.7.6 Decode and Dispatch Logic

Once an instruction arrives from the fetch unit, it is received by the decode and dispatch logic. The different fields of the instruction are checked by the decode logic and the functional unit corresponding to this instruction is determined. If there is enough space in the instruction waiting queue and the ROB, then the instruction is dispatched. Dispatching an instruction means reserving two entries for it; one in the instruction waiting queue and the other is in the ROB. If either the ROB or the instruction waiting queue is full, then a stall signal is sent to the fetch unit to order it to stop fetching. The dispatch logic keeps monitoring the status of the hardware, once the resource that caused the stall is freed, then the instruction is dispatched and the fetch unit is informed to resume fetching.

3.7.7 Issue Logic

Issuing an instruction in this context means reading its operands either from the register file or from the forwarding network and storing them into the input buffers so as the functional unit can start executing the instruction. In general, the instruction can be issued if the following conditions have been satisfied:

1. The functional unit should be free: this means there is no active instruction in the functional unit and there is a free output buffer for this instruction.
2. The operands of the instruction must be available at least for one thread.

3.7.8 Forwarding Network

The forwarding network is used to forward results among functional units to solve the Read after Write (RAW) hazards. Since the instruction is executed for multiple threads sequentially then once it produces the result for the first thread, this result should be forwarded to all instructions that are waiting for it so that they can be issued and start execution and the results of the subsequent thread are forwarded upon their availability. At each clock cycle, each functional unit forwards its output buffers to all functional units including itself if those buffers have valid entries. On each input buffer, there is a multiplexer controlled by the tag that determines from which unit the operand value should be read. If the operand is not immediate, then its value may be read from the register file or from the forwarded output buffers. This depends on whether the source

register value was valid at the dispatch time or not. If it was valid, then it is read from the register file and otherwise it is forwarded. Figure 9 shows the forwarding network from the ALU side.

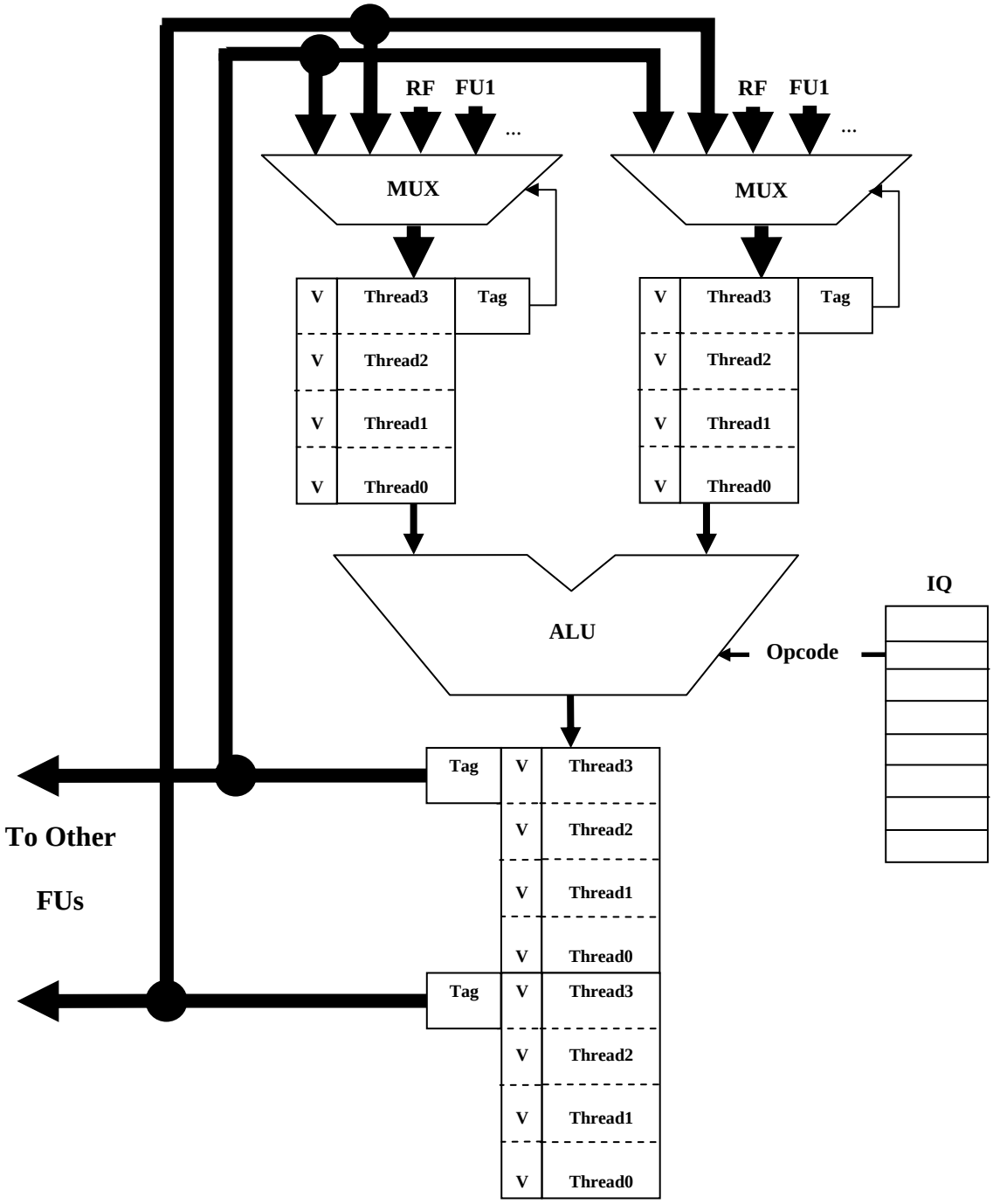

Figure 9: Forwarding Network for ALU

3.7.9 Write Back Logic

The instruction writes its results into the output buffer. After the instruction finishes execution, it writes back its results into the register file if the following conditions are met:

1. The instruction should be the first one in the ROB, this is checked by compares the instruction's tag with the tag of the first entry in the ROB. If there is a tag matching, then this instruction is the first instruction in the ROB. If it is not the first one in the ROB, then it cannot write back and this check should be performed again at the next clock cycle.
2. The qualifying predicate register should valid.
3. For each thread, the corresponding bits in the qp register and the mask register should be true in order to write back the result corresponding to this thread.

3.8 Hazards

In this section, I will discuss the different types of hazards and I will show how these hazards have been resolved in PAR core.

1. **Structural hazards:** this occurs when more than one instruction accesses the same hardware resource simultaneously. In PAR core, when a fetched instruction should be dispatched to a certain instruction queue and this queue is full or the ROB is full, then the fetch unit is stalled until the occupied resource becomes free.
2. **Control hazards:** this occurs when the value of the qualifying predicate register is being produced while it is needed by a subsequent instruction. To solve that, if the subsequent instruction is a control instruction, then the fetch unit stalls until the value of the qualifying predicate register becomes available. This stall doesn't affect the performance because there is no high pressure from the backend on the frontend. However, the non-control instructions are executed speculatively and the qualifying predicate register is checked at the write back stage.
3. **Data hazards:** there are three types of data hazards. (1) RAW or the true dependency hazard. In this case, the result of one instruction is consumed by a subsequent instruction. In PAR core, the RAW hazards have been resolved by forwarding the results from the producer to the consumer. The instruction cannot be issued until the operands are available at least for one thread. (2) Write after Read (WAR) hazard; occurs when a subsequent instruction is going to write on one of the source registers of a given instruction. (3) Write after Write (WAW)

hazard; occurs when two or more instructions are going to write on the same register. In WAW, there is a problem if these instructions are executed out of order. In PAR core, WAR and WAW hazards have been eliminated by register renaming. This has been implemented by dispatching instructions in order and marking the instruction's destination register with the instruction tag. Also if any of the instruction's source registers is unavailable at the dispatch time, then its tag is read and sent to the instruction waiting queue and the register is read from the forwarding network.

CHAPTER 4 PARSIM SIMULATOR

4.1 Overview

Since PAR core is based on a new ISA with unique features, the cycle-accurate multithreaded simulator, ParSim, has been developed from scratch to evaluate the performance of the PAR core. I used C++ to develop ParSim because C++ is efficient for simulation and it is an object oriented language which allows defining modules with their behaviors.

ParSim contains a model for the PAR core and simulates its behavior. So it supports multiple threads, multiple functional units and a frontend based on the PAR ISA features like the stop bit, the control instructions and the control stack.

ParSim receives a couple of files as input and generates another couple of files as output. The first input file is the source code file which contains the thread program and the directives that describe the PAR packet contents. The second input file is the configuration file which contains the PAR core configurable parameters' values. The two output files are the program's results file which contains the contents of the registers and the performance statistics' report file.

4.2 ParSim Structure

ParSim consists of two parts: the first one is the translator part which receives the input files and translates them into a certain format. The second part is the simulation process part which executes the thread program as many times as specified in the PAR packet and finally generates the output files.

The structure of ParSim reflects all architectural aspects of PAR core. So it has data structures to represent functional units, input buffers, output buffers, instruction cache, control stack, fetch unit, dispatch logic, issue logic, forwarding network and writing back logic. Besides the logic and the structures that have been implemented to simulate the functionality of the PAR core, a mechanism has been implemented to measure the performance of the PAR core accurately.

4.3 Simulation Methodology

Once the input files are fed to ParSim, the translator processes them and generates the intermediate representation of the instructions from the source code file and the configuration object from the configuration file. Figure 10 shows a high level flowchart that depicts the high level architecture of the ParSim. The intermediate representation of instructions is a list of instruction objects. The instruction object contains a set of attributes that store information about the instruction. Examples of these attributes are the instruction's opcode, the source register(s), the destination register, the stop bit value, the

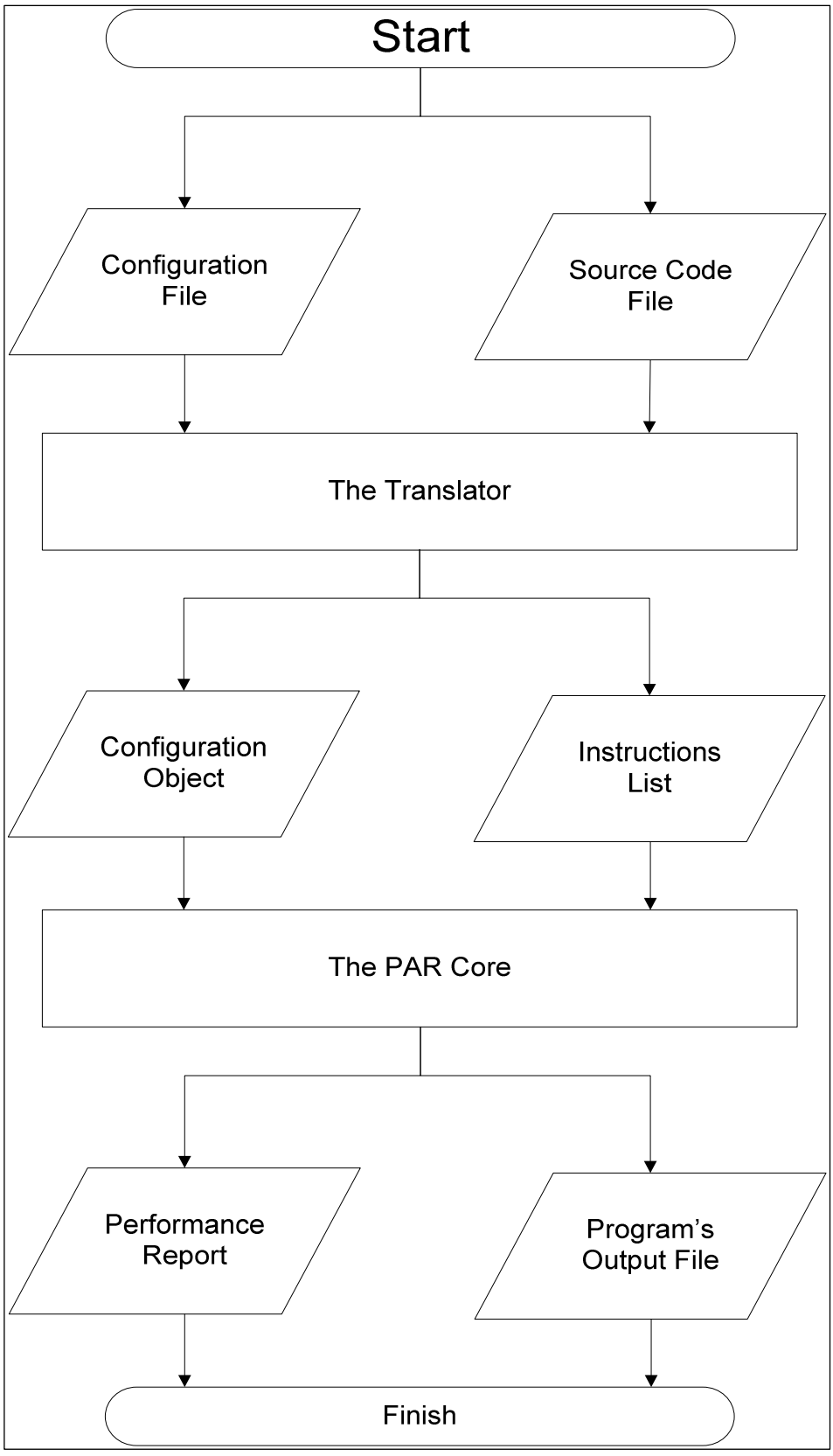

Figure 10: High Level Flowchart of ParSim

immediate value if any, the instruction address, etc. The configuration object has the attributes that store the values of the configuration parameters extracted from the configuration file.

After the translation phase finishes successfully, the output of this phase namely the list of instructions and the configuration object are fed to the PAR core object as inputs. The PAR core object configures itself by setting its configurable parameters to the values of these parameters in the configuration object and it starts the simulation process.

Figure 11 shows a flowchart that depicts the ParSim's simulation process. ParSim has a global clock that works as the heart beat of the simulator and it is used to synchronize the different operations. Since the PAR core's backend is pipelined, ParSim does the following actions at each clock cycle:

1. If there is no stall, ParSim updates the PC in the same manner the PAR core does.
2. It fetches the next instruction from the instruction cache according to the PC's value.
3. If the fetched instruction is a control instruction, the ParSim updates the PC and the control stack accordingly and in the same manner the PAR core does.
4. If the fetched instruction is a non-control instruction, it is dispatched to the corresponding functional unit if there are available buffers and otherwise a stall signal is generated.

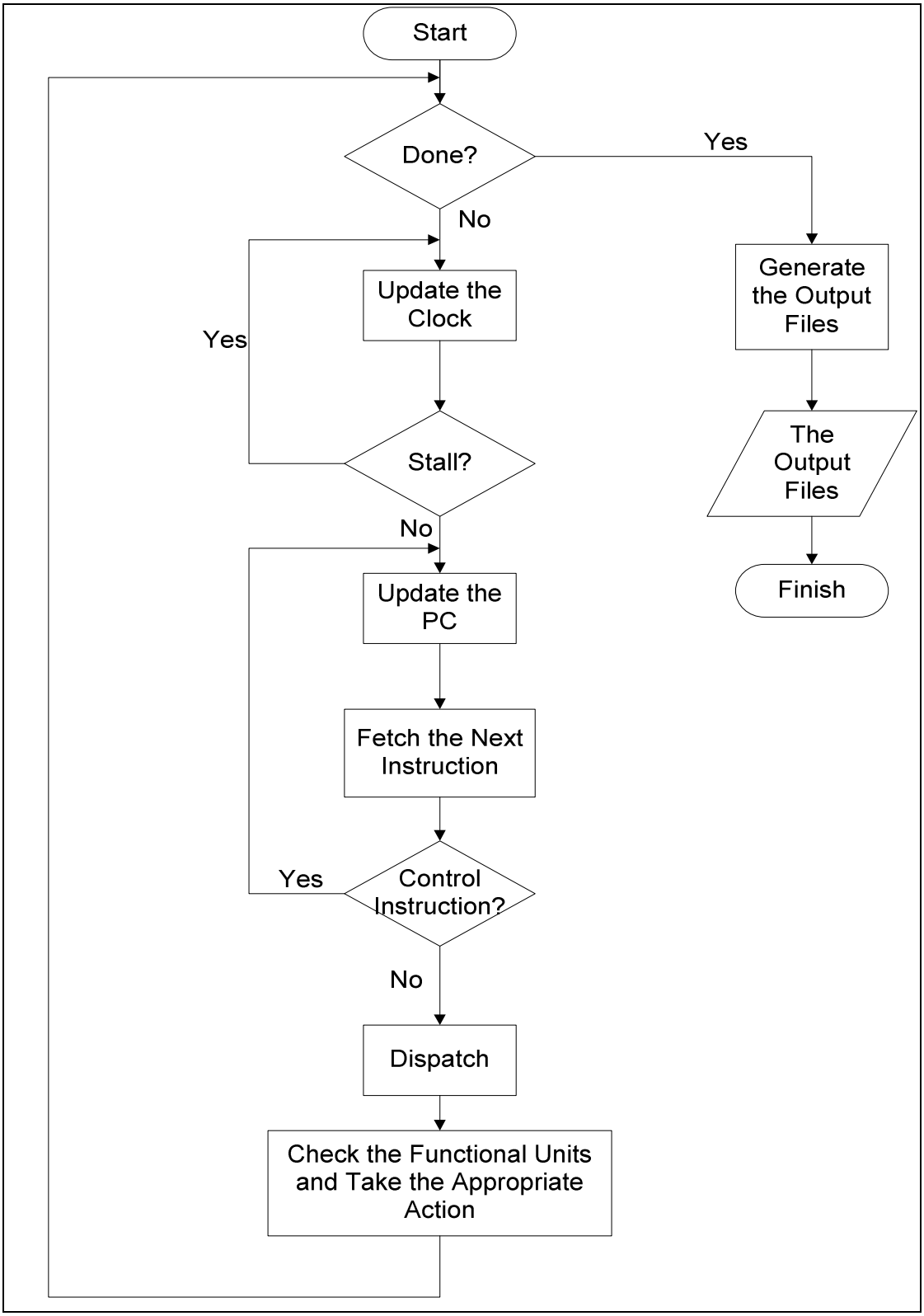

Figure 11: ParSim's Simulation Process

5. It monitors all functional units. Monitoring a functional unit includes the following checks:
    a. It checks if the functional unit has an active instruction or not. If it has an active instruction, this means that there is an instruction that is being executed by the functional unit. In this case, ParSim checks whether this instruction has finished execution or not. If it has finished execution, it checks whether this instruction is on the front of the ROB or not. If it is on the front of the ROB, the instruction is executed for all threads and the result is written back to the register file and the reserved resources for this instruction are freed. Besides that the statistical variables like the number of instructions executed are updated accordingly.
    b. If the functional unit has an active instruction and one or more of its operands are not available for all threads then ParSim reads them from the forwarding network upon their availability.
    c. If there is an instruction that has finished execution but its results have not been written back to the register file and they are still in the output buffer, ParSim checks if this instruction becomes the first one in the ROB or not and if it is the first one in the ROB then it writes its results back and free that output buffer.
    d. If the functional unit is free and it has a free output buffer and there is a waiting instruction whose operands are ready at least for the first thread, then this instruction is issued to start execution. Issuing an instruction includes marking the unit as busy and setting the expected time for the instruction to finish for all threads and the expected time for the instruction to finish for the

first thread because the result of the first thread might be required by other waiting instructions.

Figure 12 shows a flowchart that depicts how ParSim monitors a functional unit.

6. At each clock cycle, ParSim has to update the statistical variables that should be updated like the number of instructions executed, the busy time of the functional unit, the number of stall cycles, the number of ROB entries occupied etc.

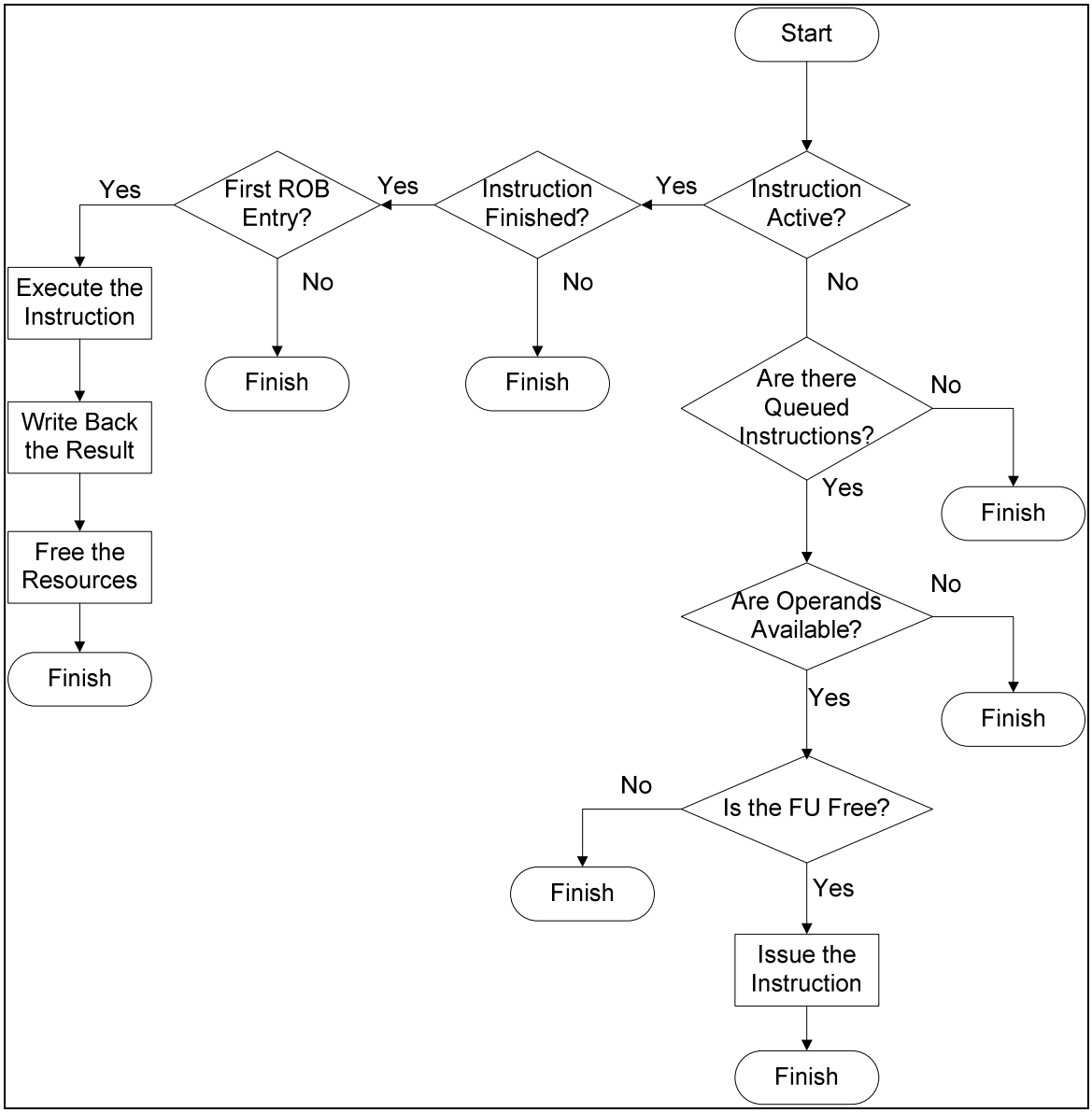

Figure 12: Functional Unit Monitoring Flowchart

4.4 Performance Metrics

Many performance metrics have been defined in ParSim to evaluate the proposed architecture and to measure how much the hardware resources are utilized. These performance metrics are defined as statistical variables in ParSim and they are updated during the simulation process.

These performance metrics include:

1. The program execution time in terms of clock cycles.
2. The number of instructions executed per functional unit.
3. The number of instructions executed per clock cycle (IPC).
4. FPU throughput.
5. ALU utilization.
6. Compare unit utilization.
7. L/S unit utilization.
8. ROB utilization.
9. Number of stall cycles.

4.5 The Source Code File

The source code file contains the contents of the PAR packet plus the thread program in PAR assembly. So it has two segments: the par segment and the code

segment. The PAR segment starts with the directive .PAR and it has the following directives:

1. .ADDRESS: to specify the starting address of the thread program.
2. .THREADS: to specify the number of threads to be executed for this par packet.
3. .In to specify the value of the nth inherited register.

The code segment starts with the directive .CODE and after that the instructions of the thread program are listed. Figure 13 shows a sample source code input file.

```
//Dense Matrix-Matrix Multiplication (DMMM)

//This is the PAR packet content

.PAR

.ADDRESS = 0

.THREADS = 1280

.i0 = 0

.i1 = 0    //&A

.i2 = 1000  //&B

.i3 = 2000  //&C

.i4 = 1000  //N


.CODE

DMMMThread: set r10 = 0

            rem  r2  =       i0,  i4  // r2 = the address of the first element of the current column

            sub  r1  =       i0,  r2  //r1 = the address of the first element of the current row

            loop i4,   dotProduct //multiply a row with a column

            st8  i3[i0]   =   r10#     //Store the value of the computed element.


dotProduct: ld8   r4     =   i1[r1] //Load A[i][k]

              ld8   r5   =   i2[r2]  //Load B[k][j]

              mac.d r10  =   r4,   r5 // r10 = r10 + A[i][k] * B[k][j]

              add   r1   =   r1,   1

              add   r2   =   r2,   i4#
```

Figure 13: A Sample Source Code Input File

4.6 The Configuration File

The configuration file is a plain text file which contains the values of the PAR core's configurable parameters. Figure 14 shows a snapshot of the configuration file. The configuration file contains the following configurable parameters of the PAR core:

1. The number of lanes in the core.
2. The instruction cache size.
3. The general purpose register file size.
4. The predicate register file's size.
5. The number of inherited registers.
6. ROB size.
7. The size of the instructions waiting queue per functional unit.
8. Multithreading depth which is the number of simultaneous threads supported by a single lane
9. The number of pipeline stages in the FPU unit which reflects the FP instruction's latency. That is, if the number of FP pipeline stages is four, then the FP instruction's latency is four clock cycles.
10. ALU latency.
11. Compare unit latency.
12. The data cache access latency if there is a cache hit.

```
NUMBER_OF_LANES = 1

INSTRUCTION_CACHE_SIZE = 1024 // instructions

GENERAL_PURPOSE_REGISTER_FILE_SIZE = 16

PREDICATE_REGISTER_FILE_SIZE = 8

NUMBER_OF_INHERITED_REGISTERS = 16

INSTRUCTION_WAITING_QUEUE_SIZE = 7

NUMBER_OF_OUTPUT_BUFFERS_PER_FUNCTIONAL_UNIT = 8

ROB_SIZE = 32

MULTITHREADING_DEPTH = 4

NUMBER_OF_PIPELINE_STAGES_IN_THE_FLOATING_POINT_UNIT = 5

ALU_LATENCY = 1
```

Figure 14: A Snapshot of the Configuration File for ParSim

4.7 The Performance Statistics Report File

The performance statistics report file is an output file that displays the performance statistics of the program that was executed by ParSim. These statistics include timing statistics like the execution time of the program and the IPC, hardware utilization statistics like the functional units and the buffers utilizations and other statistics like the number of stall cycles. Figure 15 shows a snapshot from this report.

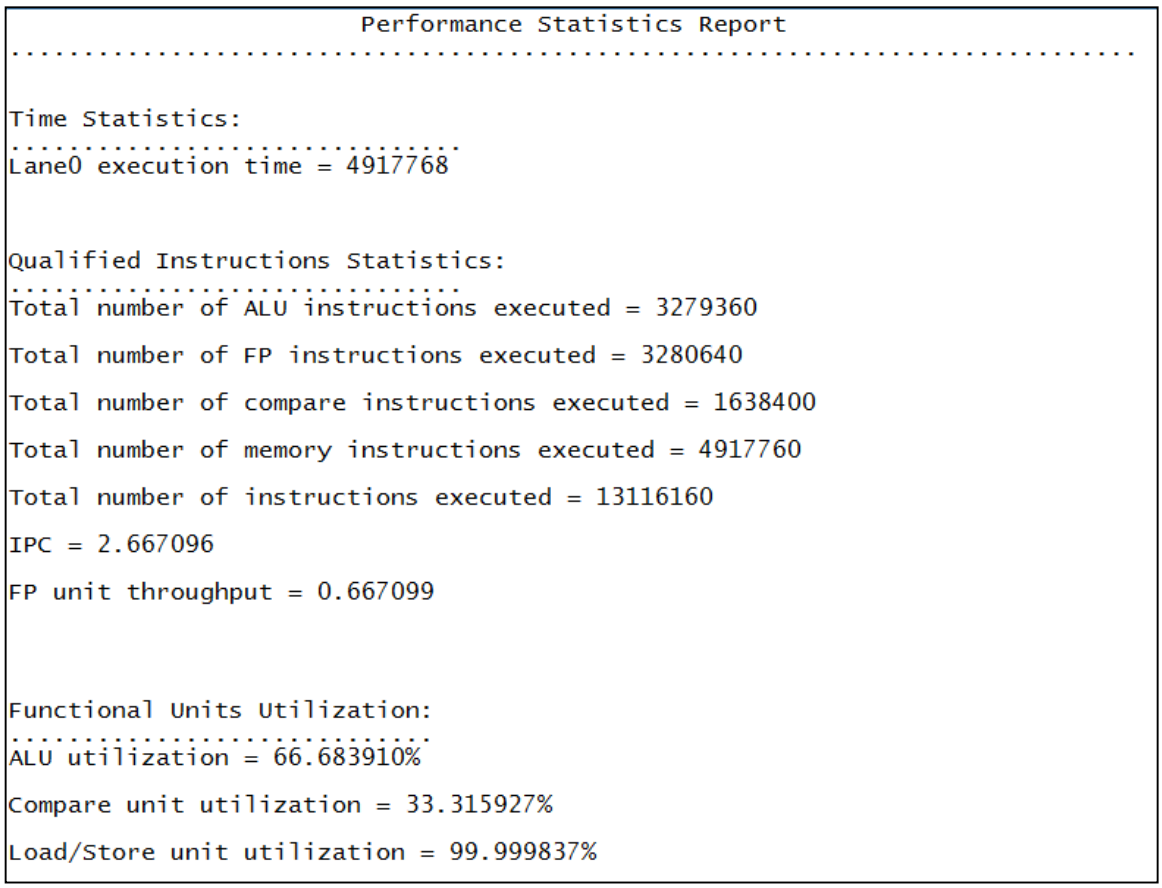

Figure 15: A Snapshot from the Performance Statistics Report Generated by ParSim

4.8 Conclusion

Intensive testing has been performed to ensure the correctness, accuracy and robustness. After adding any feature, I used to test the whole simulator by running many scenarios. In testing, I made sure that the PAR core is configured properly, the source code file is scanned, parsed and translated successfully, the instructions are executed correctly and the performance statistics are reported accurately.

After the development of ParSim completed, I wrote several data-parallel benchmarks, I ran them on ParSim and I performed extra quality assurance until ParSim becomes a mature simulator and its results can be trusted.

CHAPTER 5 BENCHMARKS

5.1 Overview

To evaluate the performance of the PAR core, I have coded some data-parallel benchmarks in PAR assembly. These benchmarks include the Jacobi iterative method, three benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC) and three benchmarks from the benchmarks that have been by Kumar et al. [31].

Since the PAR core is dedicated for executing the PAR packet which contains completely independent threads, I wrote the benchmark's kernel that represents a thread which is independent from other threads. For example, in matrix multiplication the thread’s kernel produces only one element in the destination matrix by multiplying a row from the first matrix by a column from the second matrix. Producing one element in the destination matrix is independent from producing other elements.

The granularity of the thread varies from one benchmark to another. For example, thousands of instructions may be executed per matrix multiplication thread while only several instructions are executed per scaled vector addition thread.

In the rest of this chapter, I will describe all of the benchmarks' kernels that I have coded in PAR assembly. Appendix B shows the source code of theses benchmarks’ kernels in C++ and appendix C shows their source code in PAR assembly.

5.2 Dense Matrix-Matrix Multiplication (DMMM)

This benchmark explores the target CPU's capability to perform two-dimensional array access and to perform a variety of arithmetic and memory operations. The matrix multiplication thread is responsible for producing one element in the target matrix by multiplying a row from the first matrix by a column from the second matrix.

5.3 Jacobi Iterative Method (JIM)

This iterative method is used for solving linear equations which is a common problem used in many science and engineering applications. It has been selected to evaluate PAR core because it can benchmark the access of two-dimensional arrays and it has a variety of instructions. The linear system of equations can be modeled as shown in equation 1.

$$AX = B \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$

Such that A is the two-dimensional coefficients' matrix, B is the constants' array and X is the unknowns' array. The solution of this system is by finding the values of those unknowns. The Jacobi iterative method solves this system of linear equations iteratively. In this method, the unknowns may be initialized to the values of constants in B and then

the algorithm iterates over these unknowns for a given number of iterations and updates them in each iteration according to equation 2.

$$Xi = \left(\frac{1}{Aii}\right)\left(Bi - \sum_{j \neq i}^{n} Aij \times Xj\right) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (2)$$

The JIM's thread is responsible for updating one unknown value in a single iteration. This thread covers most of features of the PAR core; it has ALU, FP, loop, compare and memory instructions.

5.4 Gauss-Seidel (GS): Red-Black Gauss-Seidel on a 2D Grid [32]

The GS method is a method of update in which the point in a two-dimensional grid is replaced by the weighted average of itself and its four neighbors. This update is used in solving a simple partial differential equation in an (N +2) (N+2) grid. This equation solver kernel proceeds over a number of sweeps such that in each sweep it operates on the whole N x N interior points of the grid and performs the GS update. These sweeps continue until some convergence occurs.

If each thread is defined to update only one point, then the data dependencies don't allow parallelizing these threads. The parallelizing approach of the GS comes from the knowledge of the problem behind the sequential program. Since the GS method is not

an exact solution but it just iterates until convergence, the points can be updated in different order as long as the updated values are used frequently enough.

This ordering is called red-black ordering in which the grid's points are separated into alternating red and black points. In this ordering, updating a red point doesn't need any updated red point and this is true also for the black points. So updating the red points can be done in parallel.

In this ordering, the grid is updated in two phases. (1) The red points are updated.(2) The black points are updated. The two phases should be separated by a global synchronization point and computations can be done in parallel within the same phase.

To execute the GS, the master core should generate PAR packets for the red points and wait for them to finish. After that, it generates PAR packets for the black points. After all black points are updated; the master core checks for convergence and decides whether to have another sweep or to exit the algorithm.

The GS's thread is responsible for updating only one point. This thread can be used to explore the target CPU's capability of performing accumulation, FP operations and calculating the absolute difference.

5.5 RGB to YIQ Conversion (RGB-YIQ) [33]

This is an EEMBC benchmark which benchmarks the performance for digital video processing and explores multiply/accumulate capability of the CPU. The RGB to

YIQ conversion implemented in this benchmark is used in many multimedia applications. The RGB to YIQ conversion thread is responsible for converting one pixel from the RGB format to YIQ format. This thread includes ALU, FP and memory instructions.

5.6 RGB to CMYK Conversion (RGB-CMYK) [33]

This is an EEMBC benchmark that is used to explore the target CPU's capability for basic arithmetic and minimum value detection. It is used to benchmark digital image processing performance in printers and other digital imaging products. It is used in color printers such that this benchmark receives the RGB inputs from the PC and converts them to CMYK color signals for printing. The RGB to CMYK thread is responsible for converting one pixel from the RGB format to the CMYK format.

5.7 High Pass Grey-Scale Filter (HPF) [33]

This is an EEMBC benchmark that benchmarks performance of image processing in digital cameras and other digital image products by exploring the target PC's capability to perform two dimensional data array access and multiply/accumulate calculation. This filter is used in many applications like DSCs (Digital Still Camera).

For each pixel in the image, the filter calculates the output results from the nine pixels which are the pixel under consideration plus its eight neighbors multiplied by the

filter coefficients shown in figure 16, then shift the accumulated value by eight bits. Figure 17 shows the equations for calculating the output of the pixel P(c) such that c is the center location of the filter window and w is the horizontal image width.

$$\begin{pmatrix} F11 & F21 & F31 \\ F12 & F22 & F32 \\ F13 & F23 & F33 \end{pmatrix} = \begin{pmatrix} -28 & -28 & -28 \\ -28 & 255 & -28 \\ -28 & -28 & -28 \end{pmatrix}$$

Figure 16: High Pass Grey-Scale Filter Coefficients [33]

```
PelValue = (Short)(  F11*P(c-w-1) +F21*P(c-w)  +F31*P(c-w+1)
                     +F12*P(c-1)    +F22*P(c)     +F32*P(c+1)
                     +F13*P(c+w-1)+F23*P(c+w)  +F33*P(c+w+1) )

Out = (Byte)(PelValue >>8);
```

Figure 17: High Pass Grey-Scale Filter Equations [33]

The HPF's thread is responsible for calculating the output of one pixel. This thread contains various ALU, FP and memory instructions.

5.8 Scaled Vector Addition (SVA)

It measures the capability of the target CPU in performing FP addition and multiplication operations. The SVA's thread is responsible for calculating one element in the target array.

CHAPTER 6 EXPERIMENTAL RESULTS AND DISCUSSION

6.1 Overview

Multiple data-parallel benchmarks have been run on ParSim simulator to measure the performance of the PAR core. In this chapter, I will analyze the performance of a single lane as well as the performance across multiple lanes. Regarding the PAR core settings for which these results have been taken, there are parameters that have been fixed to certain values and parameters that were variables to see how they affect the performance. Table 1 displays the fixed parameters with their values.

Table 1: The Values of the Fixed Parameters

Parameter Name	Parameter Value
Predicate register file size	8 Registers
Number of general purpose registers	16 Registers
Number of inherited registers	16 Registers
Number of input buffers per functional unit	1 Buffer
Number of output buffers per functional unit	2 Buffers
Dispatch queue size per functional unit	2 Entries
ROB size	8 Entries
The FPU pipeline depth	4 Stages
ALU latency	1 Clock Cycle
Compare unit latency	1 Clock Cycle
Level one data cache latency if hit	1 Clock Cycle

The number of simultaneous threads supported by a single lane was variable and it could be 1, 2, 4, or 8 threads. In measuring the performance of multiple lanes, the number of simultaneous threads supported by a single lane has been fixed to four threads and the number of lanes was variable and it could be either 1, 2, 4, 8, 16, 32 or 64 lanes. The number four has been selected as the number of simultaneous threads supported by a single lane because this number makes the single lane offers the maximum performance with the minimum hardware cost.

The performance metrics that have been used for single lane are: IPC, FPU throughput, ALU utilization and L/S unit utilization. For multiple lanes, besides the IPC, the speedup has been considered to measure the scalability of the PAR core.

6.2 Analyzing Single Lane Performance

Figure 18 shows how the IPC changes with respect to the number of simultaneous threads supported by a single lane. For all benchmarks, we notice that the IPC is increasing as long as the number of threads is increasing until the number of threads becomes four, then increasing the number of threads will not affect the IPC value. The reason standing behind that is that increasing the number of simultaneous threads will increase the work to be done by one instruction and then increasing the throughput of the functional units which means executing more instructions per the unit of time. Increasing

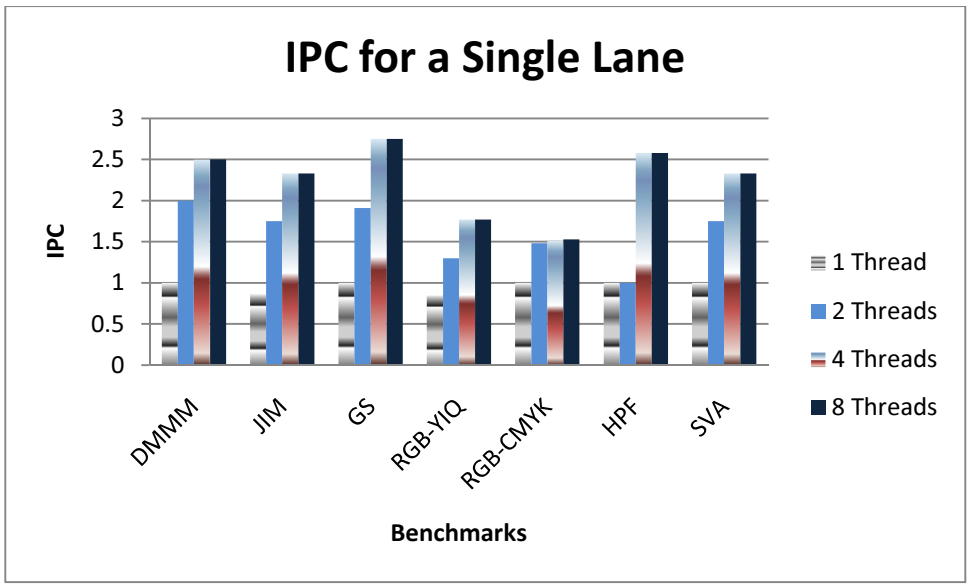

Figure 18: IPC for a Single Lane

the number of simultaneous threads beyond four will not improve the IPC because the instructions in the benchmarks are mixed such that eight threads offer the same performance gained from four threads and because the number of functional units per lane is fixed. One more note, in the RGB-CMYK benchmark is that the increases in the IPC are smaller than the increases in the other benchmarks; the reason is that the RGB-CMYK benchmark doesn't contain FP instructions. In PAR core, more FP instructions result in higher increases in the IPC when the number of threads supported by a single lane is increased because the FPU is pipelined.

So having four simultaneous threads per lane gives the highest IPC with the lowest hardware cost. For the benchmarks that have been run, the minimum IPC is 1.51 instructions /cycle and the maximum IPC is 2.75 instructions/ cycle for a 4-way multithreaded lane.

Regarding the FPU throughput, FP instructions are long-latency instructions; having a pipelined FPU with SIMT architecture will accelerate the execution of these instructions. Figure 19 shows the FPU throughput for the different benchmarks. The maximum FPU throughput is obtained when the number of threads is four because the number of pipeline stages in the FPU is four. For the benchmarks that contain FP instructions, it is noticed that the maximum FPU throughput is almost one which means that the FPU executes one instruction per clock cycle on average, while the minimum FPU throughput is 0.67 instructions/ cycle.

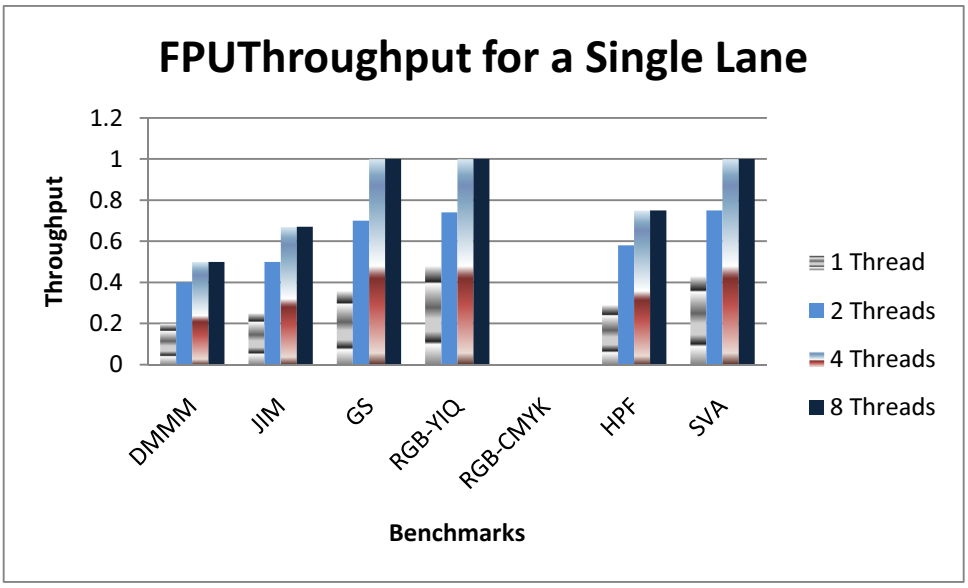

Figure 19: FPU Throughput for a Single Lane

The last performance metric used for the single lane is the functional units' utilization. Figures 20 and 21 show that increasing the number of threads supported by a single lane results in increasing the ALU and L/S units utilization respectively. Maximizing the utilization means keeping the functional units busy as long as possible

which means having more throughput. From these figures, we notice that the maximum utilization is gained when the number of threads is four and increasing the number of threads beyond four will not enhance the utilization. For four threads, the maximum ALU and L/S unit utilization obtained is almost 100% which means that these units are always busy for those benchmarks. Whereas the minimum ALU utilization obtained for four threads is 30.76% and the minimum L/S unit utilization is 46.14%.

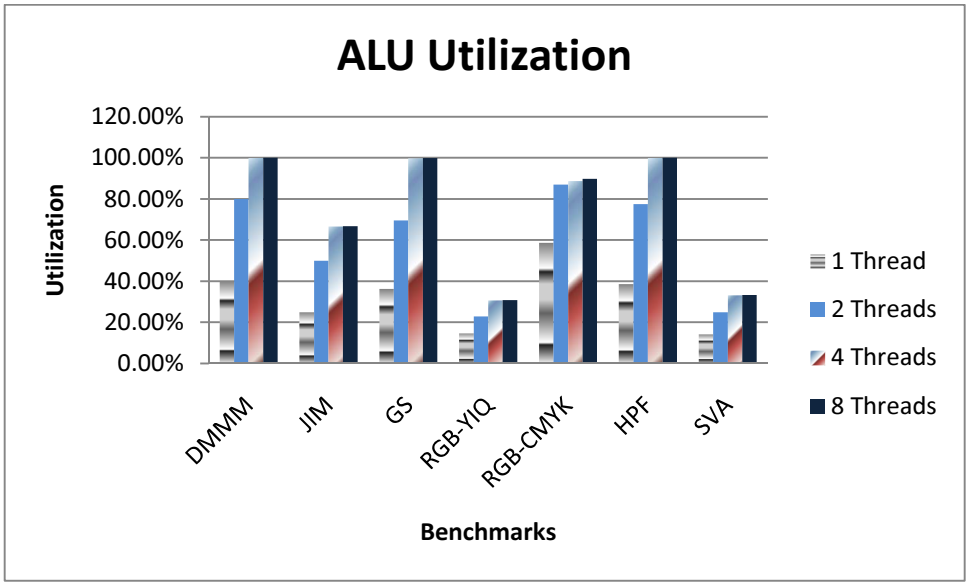

Figure 20: ALU Utilization

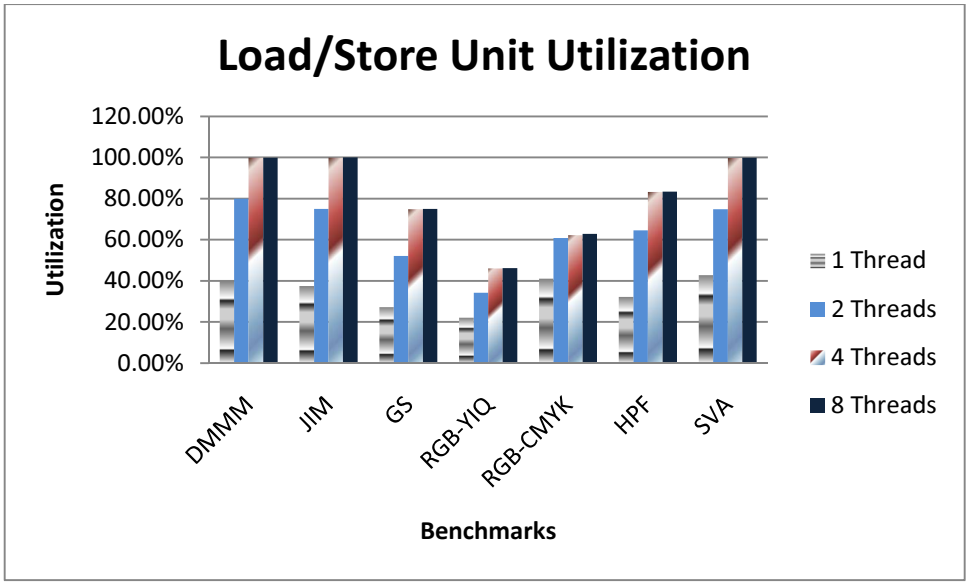

Figure 21: Load/Store Unit Utilization

6.3 Analyzing Multiple Lanes Performance

Adding more lanes to PAR core will replicate the throughput and reduce the execution time. Figure 22 shows the IPC for multiple lanes. It is obvious from this figure that the IPC increases as long as the number of lanes increases for all benchmarks. For 64 lanes, the minimum IPC is 96.28 instructions/ cycle while the maximum IPC is 174.26 instructions/ cycle. This increase in the IPC is on the account of the hardware cost. Adding more lanes will not affect the complexity of the frontend, but it increases the cost of the backend. Adding one more lane means adding four functional units with their buffers, dispatch and issue logic, registers and other components of the backend.

Despite the fact that I have reported performance results for up to 64 lanes in this section, the goal is to show that this architecture is scalable and it is not necessary that the hardware

implementation of the PAR core can contain this number of lanes. Adding additional lanes doesn't only increase the core's area, but it also increases the complexity of the interconnections between the frontend and the backend and the complexity of the interconnections with the memory. To avoid these complexities, these 64 lanes may be split into multiple cores such that each core contains 2, 4 or any reasonable number of lanes determined by the hardware designers.

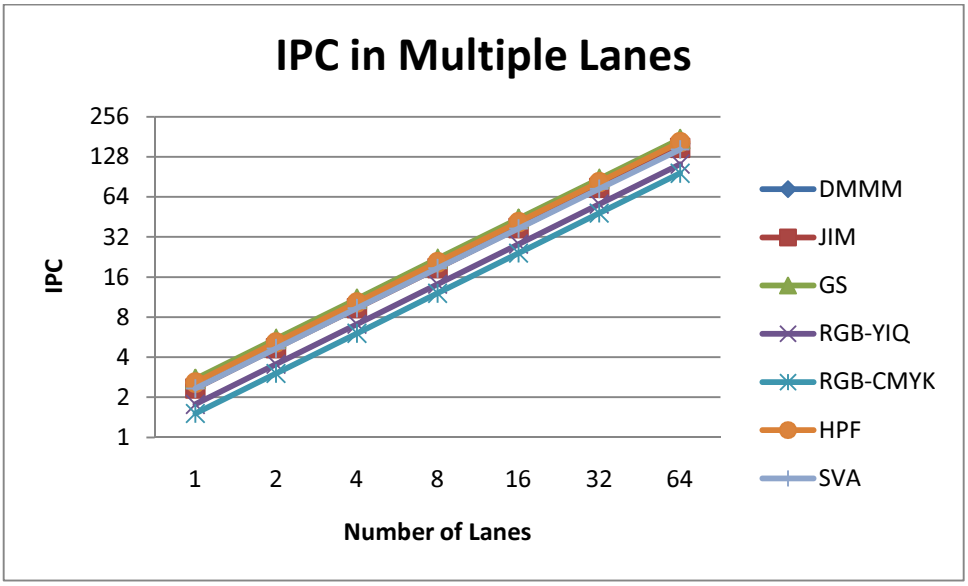

Figure 22: IPC in Multiple Lanes

The speedup is one of the most important performance metrics for parallel processors. It measures how much a parallel processor is scalable. The speedup is calculated by dividing the sequential execution time of a given program on the parallel execution time of that program. Since the threads in PAR core are completely independent, then the scheduling, the communication and the synchronization overhead is

zero which means adding more lanes results in performance enhancement proportional to the number of lanes added. Figure 23 shows that the PAR core's speedup is linear for all benchmarks.

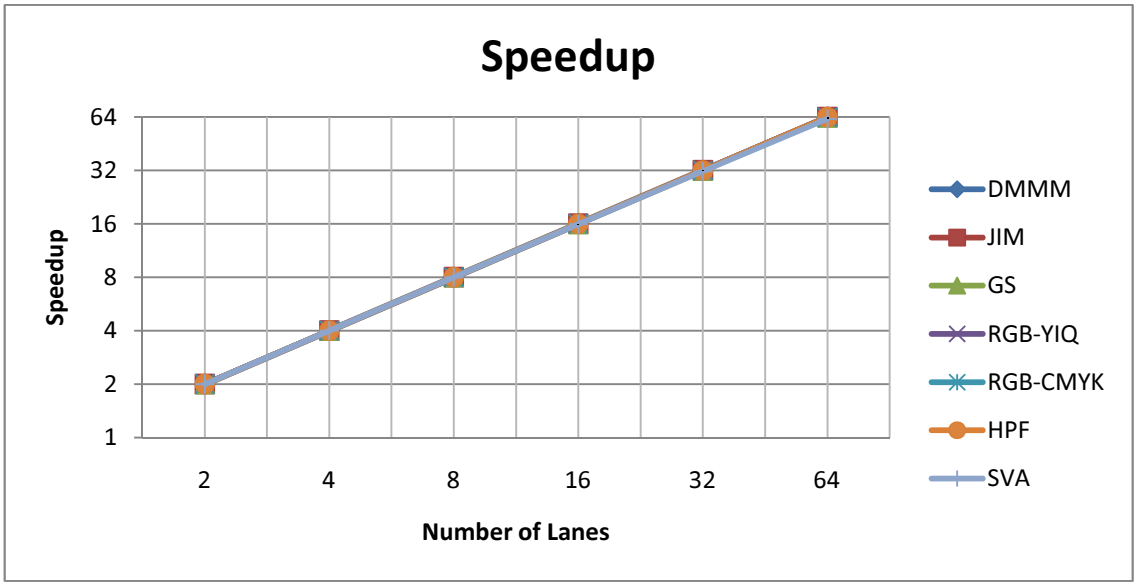

Figure 23: Speedup across Multiple Lanes

CHAPTER 7 CONCLUSION AND FUTURE WORK

The technology trends and the application demand motivate the architects to design more productive processors that handle instructions from different threads simultaneously. In this research work, I have proposed a scalable SIMT processor core called PAR core which may have multiple processing lanes but one simple frontend. I have described the micro-architecture of the PAR core which has been proposed for optimized parallel thread execution. Also I have developed ParSim Simulator which is a cycle-accurate multithreaded simulator that implements the PAR core features and evaluates its performance. The simulation results showed that the SIMT multithreading approach which has been implemented in the PAR core provided high performance such that the maximum IPC was 2.75 instructions /cycle and the maximum hardware utilization was almost full for 4-way multithreaded PAR core.

Besides that, the simulation results showed the scalability of this architecture because there is no inter-thread communication and synchronization. Adding more resources through replicating the processing pipeline results in performance enhancement proportional to the added resources i.e. the speedup was linear.

In the future, we are going to show the full picture by integrating the PAR core with the full system. PAR core is a part of a larger system which contains many cores with memory hierarchy, thread scheduler and interconnection network. Once the PAR core is integrated with the complete system, global load and global store instructions can

be implemented. These instructions are expected to achieve high computation communication overlapping and thus hide the memory latency.

Appendix A PAR ISA

1. Instruction Formats

Four instruction formats are defined as shown in figure 24. All instructions begin with a qualifying predicate **qp** and terminate with a stop bit **s**. The R-type format defines register-to-register arithmetic and logic instructions. The I-type format defines instructions that include a 9-bit constant, including load and store. The S-type format defines the SET and control instructions. The XP format defines the expand instruction.

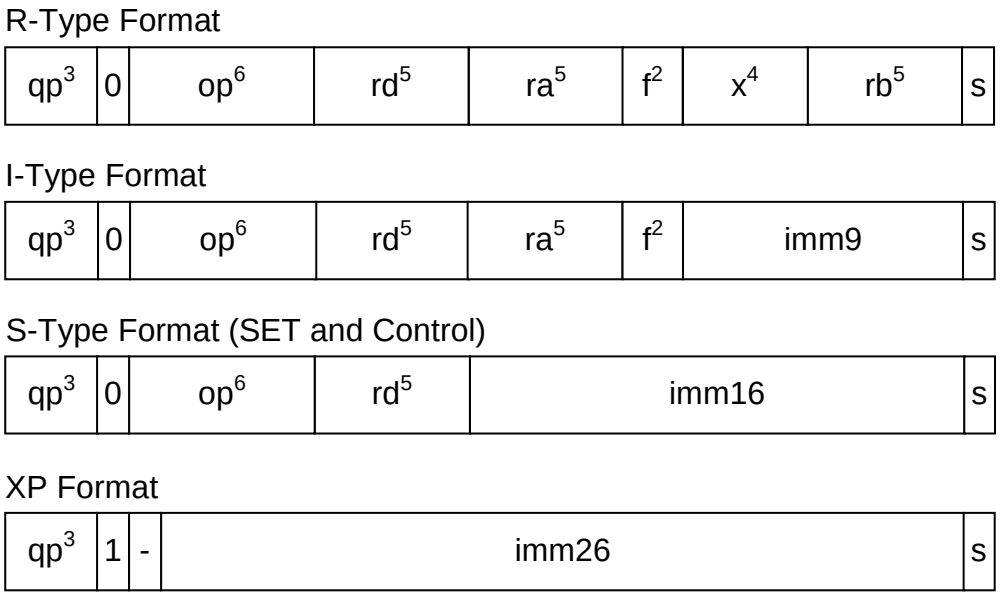

Figure 24: Instruction Formats

- **qp** 3-bit qualifying predicate, which can be **(p0)** to **(p7)**
- **op** 6-bit major opcode for most instructions, except for the XP format
- **rd** 5-bit destination or a data source register, which can be read and written
- **ra** 5-bit source register **a**, which is read only (except by load and store)
- **rb** 5-bit source register **b**, which is read only
- **f** 2-bit function code for R-type and I-type instructions
- **x** 4-bit opcode extension for R-type instructions only
- **s** stop bit that marks the end of an instruction block
- **imm9** 9-bit signed immediate constant for I-type instructions
- **imm16** 16-bit immediate constant for S-type instructions
- **imm26** 26-bit immediate constant used by XP (expand) instruction

The above formats represent the majority of instructions. However, there are minor variations used by few instructions, such as *compare*. These variations will be explained in later sections.

2. Integer Addition Instructions (R-type and I-type)

(qp) add rd = ra, rb // if (qp) rd ← (ra)+(rb), p7←OV

(qp) addu rd = ra, rb // if (qp) rd ← (ra)+(rb), p7←CA

(qp) add rd = ra, imm9 // if (qp) rd ← (ra)+imm9, p7←OV

(qp) addu rd = ra, imm9 // if (qp) rd ← (ra)+imm9, p7←CA

3. Integer Subtraction Instructions (R-type and I-type)

(qp) subf rd = ra, rb // if (qp) rd ← (rb)-(ra), p7←OV

(qp) subfu rd = ra, rb // if (qp) rd ← (rb)-(ra), p7←BO

(qp) subf rd = ra, imm9 // if (qp) rd ← imm9-(ra), p7←OV

(qp) subfu rd = ra, imm9 // if (qp) rd ← imm9-(ra), p7←BO

4. The SUB and NEG Pseudo Instructions

(qp) sub rd = ra, rb // Pseudo: (qp) subf rd = rb, ra

(qp) subu rd = ra, rb // Pseudo: (qp) subfu rd = rb, ra

(qp) neg rd = ra // Pseudo: (qp) subf rd = ra, 0

5. Bitwise Logic Instructions (R-type)

(qp) and rd = ra, rb // if (qp) rd ← (ra)&(rb)

(qp) or rd = ra, rb // if (qp) rd ← (ra)|(rb)

(qp) xor rd = ra, rb // if (qp) rd ← (ra)^(rb)

(qp) nor rd = ra, rb // if (qp) rd ← ~[(ra)|(rb)]

6. Bitwise Logic Instructions (I-type)

(qp) and rd = ra, imm9 // if (qp) rd ← (ra)&(imm9)

(qp) or rd = ra, imm9 // if (qp) rd ← (ra)|(imm9)

(qp) xor rd = ra, imm9 // if (qp) rd ← (ra)^(imm9)

(qp) nor rd = ra, imm9 // if (qp) rd ← ~[(ra)|(imm9)]

7. Additional Bitwise Logic Instructions (R-type only)

(qp) andc rd = ra, rb // if (qp) rd ← (ra)&~(rb)

(qp) orc rd = ra, rb // if (qp) rd ← (ra)|~(rb)

(qp) xnor rd = ra, rb // if (qp) rd ← ~[(ra)^(rb)]

(qp) nand rd = ra, rb // if (qp) rd ← ~[(ra)&(rb)]

8. The MOV and NOT Pseudo-Instructions

(qp) mov rd = ra // Pseudo: (qp) or rd = ra, 0

(qp) not rd = ra // Pseudo: (qp) nor rd = ra, 0

9. Shift and Rotate by a Variable Amount Instructions

(R-type)

(qp) sll rd = ra, rb // if (qp) rd ← (ra)‖0 << (rb)

(qp) srl rd = ra, rb // if (qp) rd ← 0‖(ra) >> (rb)

(qp) sra rd = ra, rb // if (qp) rd ← sign‖(ra) >> (rb)

(qp) ror rd = ra, rb // if (qp) rd ← (ra)‖(ra) >> (rb)

10. Shift and Rotate by a Constant Amount Instructions (I-type)

(qp) sll rd = ra, imm6 // if (qp) rd ← (ra)‖0 << imm6

(qp) srl rd = ra, imm6 // if (qp) rd ← 0‖(ra) >> imm6

(qp) sra rd = ra, imm6 // if (qp) rd ← sign‖(ra) >> imm6

(qp) ror rd = ra, imm6 // if (qp) rd ← (ra)‖(ra) >> imm6

11. The ROL Pseudo-Instruction

(qp) rol rd = ra, imm6 // Pseudo: (qp) ror rd = ra, 64-imm6

12. Shift Left and Add Instruction (R-type only)

(qp) sla rd = ra, rb, n // if (qp) rd←(ra)+(rb)<<n

13. Min and Max Instructions(R-type)

(qp) min rd = ra, rb // (qp) {rd←min (ra,rb), p7←(ra)$<_s$(rb)}

(qp) max rd = ra, rb // (qp) {rd←max (ra,rb), p7←(ra)$>_s$(rb)}

(qp) minu rd = ra, rb // (qp) {rd←minu(ra,rb), p7←(ra)$<_u$(rb)}

(qp) maxu rd = ra, rb // (qp) {rd←maxu(ra,rb), p7←(ra)$>_u$(rb)}

14. Min and Max Instructions (I-type)

(qp) min rd = ra, imm9 // (qp) {rd←min (ra,imm9), p7←($<_s$)}

(qp) max rd = ra, imm9 // (qp) {rd←max (ra,imm9), p7←($>_s$)}

(qp) minu rd = ra, imm9 // (qp) {rd←minu(ra,imm9), p7←($<_u$)}

(qp) maxu rd = ra, imm9 // (qp) {rd←maxu(ra,imm9), p7←($>_u$)}

15. Absolute Value Instruction

(qp) abs rd = ra // if (qp) rd ← (ra)≥0?(ra):-(ra)

16. Population Count and Counting Leading Zeros Instructions

(qp) popc rd = ra // if (qp) rd ← count_1(ra)

(qp) clz rd = ra // if (qp) rd ← count_leading_0(ra)

17. Constant Formation Instructions

(qp) set rd = imm16 // if (qp) rd ← ext(imm16,64)

(qp) sli rd = imm16 // if (qp) rd ← (rd)‖imm16 << 16

18. ALU Instruction Formats

The ALU immediate and register formats are shown in figure 25. Opcodes 16 to 19 define I-type ALU instructions, while opcode 31 defines the corresponding R-type format. The R-type and I-type formats use identical function codes. In addition, the replacement of the least-significant 4-bit of opcode 31 with the 4-bit extension field **x** matches the corresponding I-type opcode.

The 9-bit immediate of opcodes 16 to 19 is sign-extended to 64 bits. However, the I-type Shift, Rotate, and Extend instructions use the low 6-bit of the immediate constant. The **ext** and **extu** instructions are of the immediate type only.

Integer Addition and Subtraction (I-Type and R-Type)

									Function Code
qp^3	0	$op^6 = 16$	rd^5	ra^5	f^2	imm9		s	f = 0 →**add**
qp^3	0	$op^6 = 31$	rd^5	ra^5	f^2	$x^4 = 0$	rb^5	s	f = 1 →**addu**

Bitwise Logic (I-Type and R-Type)

qp^3	0	$op^6 = 17$	rd^5	ra^5	f^2	imm9		s	f = 0 →**and**
qp^3	0	$op^6 = 31$	rd^5	ra^5	f^2	$x^4 = 1$	rb^5	s	f = 1 →**or**

Shift and Rotate (I-Type and R-Type)

qp^3	0	$op^6 = 18$	rd^5	ra^5	f^2	0	-	imm6	s	f = 0 →**sll**
qp^3	0	$op^6 = 31$	rd^5	ra^5	f^2	$x^4 = 2$		rb^5	s	f = 1 →**srl**

Extend (I-Type Only)

qp^3	0	$op^6 = 18$	rd^5	ra^5	f^2	1	-	imm6	s	f = 0 →**ext**

Minimum and Maximum (I-Type and R-type)

qp^3	0	$op^6 = 19$	rd^5	ra^5	f^2	imm9		s	f = 0 →**min**
qp^3	0	$op^6 = 31$	rd^5	ra^5	f^2	$x^4 = 3$	rb^5	s	f = 1 →**minu**

Figure 25: ALU Instruction Formats

19. SET Instruction Format

SET and SLI (S-Type Format)

qp^3	0	SET = 48	rd^5	imm16	s
qp^3	0	SLI = 49	rd^5	imm16	s

Figure 26: Format of the SET and SLI instructions

20. Integer Multiplication Instructions (R-type)

(qp) mul rd = ra, rb // (qp) rd ← lo[(ra)× (rb)]

(qp) mulh rd = ra, rb // (qp) rd ← hi[(ra)$\times_s$(rb)]

(qp) mulhu rd = ra, rb // (qp) rd ← hi[(ra)$\times_u$(rb)]

(qp) mulu rd = ra, rb // Pseudo: (qp) mul rd = ra, rb

21. Integer Multiplication Instructions (I-type)

(qp) mul rd = ra, imm9 // (qp) rd ← lo[(ra)× (imm9)]

(qp) mulh rd = ra, imm9 // (qp) rd ← hi[(ra)$\times_s$(imm9)]

(qp) mulhu rd = ra, imm9 // (qp) rd ← hi[(ra)$\times_u$(imm9)]

(qp) mulu rd = ra, imm9 // Pseudo: (qp) mul rd = ra, imm9

22. Integer Multiply-Accumulate Instructions (R-type)

(qp) mac rd = ra, rb // (qp) rd←(rd)+lo(ra)×$_s$ lo(rb), p7←OV

(qp) macu rd = ra, rb // (qp) rd←(rd)+lo(ra)×$_u$ lo(rb), p7←CA

23. Integer Multiply-Accumulate Instructions (I-type)

(qp) mac rd = ra, imm9 // (qp) rd←(rd)+lo(ra)×$_s$(imm9), p7←OV

(qp) macu rd = ra, imm9 // (qp) rd←(rd)+lo(ra)×$_u$(imm9), p7←CA

mul r6 = r2, r3 // r6 ← lo[(r2)× (r3)]

mulh r7 = r2, r3 // r7 ← hi[(r2)×$_s$(r3)]

addu r4 = r4, r6 // r4 ← (r4)+(r6), p7←CA

addx r5 = r5, r7 // r7 ← (r5)+(r7)+(p7), p7←CA

24. Integer Division Instructions (R-type)

(qp) div rd = ra, rb // if (qp) rd ← $(ra)/_s(rb)$

(qp) divu rd = ra, rb // if (qp) rd ← $(ra)/_u(rb)$

(qp) rem rd = ra, rb // if (qp) rd ← $(ra)\%_s(rb)$

(qp) remu rd = ra, rb // if (qp) rd ← $(ra)\%_u(rb)$

25. Integer Division Instructions (I-type)

(qp) div rd = ra, imm9 // if (qp) rd ← $(ra)/_s$ imm9

(qp) divu rd = ra, imm9 // if (qp) rd ← $(ra)/_u$ imm9

(qp) rem rd = ra, imm9 // if (qp) rd ← $(ra)\%_s$ imm9

(qp) remu rd = ra, imm9 // if (qp) rd ← $(ra)\%_u$ imm9

26. Multiply and Divide Instruction Formats

Integer Multiply (I-Type and R-Type)

									Function Code
qp^3	0	$op^6 = 32$	rd^5	ra^5	f^2	imm9		s	f = 0 →**mul**
qp^3	0	$op^6 = 35$	rd^5	ra^5	f^2	$x^4 = 0$	rb^5	s	f = 2 →**mulh**

Integer Multiply-Accumulate (I-Type and R-Type)

qp^3	0	$op^6 = 33$	rd^5	ra^5	f^2	imm9		s	f = 0 →**mac**
qp^3	0	$op^6 = 35$	rd^5	ra^5	f^2	$x^4 = 1$	rb^5	s	f = 1 →**macu**

Integer Divide and Remainder (I-Type and R-Type)

qp^3	0	$op^6 = 34$	rd^5	ra^5	f^2	imm9		s	f = 0 →**div**
qp^3	0	$op^6 = 35$	rd^5	ra^5	f^2	$x^4 = 2$	rb^5	s	f = 1 →**divu**

Figure 27: Integer Multiply and Divide Instruction Formats

27. Integer Compare Instructions (R-Type)

(qp) eq ptf = ra, rb // if (qp) {pt←(ra)= (rb), pf←(≠)}

(qp) lt ptf = ra, rb // if (qp) {pt←(ra)$<_s$(rb), pf←($\geq_s$)}

(qp) ltu ptf = ra, rb // if (qp) {pt←(ra)$<_u$(rb), pf←($\geq_u$)}

28. Integer Compare Instructions (I-Type)

(qp) eq ptf = ra, imm9 // if (qp) {pt←(ra)= (imm9), pf←(≠)}

(qp) lt ptf = ra, imm9 // if (qp) {pt←(ra)$<_s$(imm9), pf←($\geq_s$)}

(qp) ltu ptf = ra, imm9 // if (qp) {pt←(ra)$<_u$(imm9), pf←($\geq_u$)}

29. Pseudo Integer Compare Instructions (R-Type)

(qp) ne ptf = ra, rb // Pseudo: (qp) eq pft = ra, rb

(qp) le ptf = ra, rb // Pseudo: (qp) lt pft = rb, ra

(qp) leu ptf = ra, rb // Pseudo: (qp) ltu pft = rb, ra

(qp) gt ptf = ra, rb // Pseudo: (qp) lt ptf = rb, ra

(qp) gtu ptf = ra, rb // Pseudo: (qp) ltu ptf = rb, ra

(qp) ge ptf = ra, rb // Pseudo: (qp) lt pft = ra, rb

(qp) geu ptf = ra, rb // Pseudo: (qp) ltu pft = ra, rb

30. Pseudo Integer Compare Instructions (I-Type)

(qp) ne ptf = ra, imm9 // Pseudo: (qp) eq pft = ra, imm9

(qp) le ptf = ra, imm9 // Pseudo: (qp) lt ptf = ra, imm9+1

(qp) leu ptf = ra, imm9 // Pseudo: (qp) ltu ptf = ra, imm9+1

(qp) gt ptf = ra, imm9 // Pseudo: (qp) lt pft = ra, imm9+1

(qp) gtu ptf = ra, imm9 // Pseudo: (qp) ltu pft = ra, imm9+1

(qp) ge ptf = ra, imm9 // Pseudo: (qp) lt pft = ra, imm9

(qp) geu ptf = ra, imm9 // Pseudo: (qp) ltu pft = ra, imm9

(qp) eq pt = ra, rb // Pseudo: (qp) eq pt0 = ra, rb

(qp) lt pt = ra, rb // Pseudo: (qp) lt pt0 = ra, rb

(qp) ltu pt = ra, rb // Pseudo: (qp) ltu pt0 = ra, rb

(qp) gt pt = ra, rb // Pseudo: (qp) lt pt0 = rb, ra

(qp) gtu pt = ra, rb // Pseudo: (qp) ltu pt0 = rb, ra

(qp) ne pf = ra, rb // Pseudo: (qp) eq p0f = ra, rb

```
(qp)    le      pf = ra, rb     // Pseudo: (qp) lt      p0f = rb, ra

(qp)    leu     pf = ra, rb     // Pseudo: (qp) ltu     p0f = rb, ra

(qp)    ge      pf = ra, rb     // Pseudo: (qp) lt      p0f = ra, rb

(qp)    geu     pf = ra, rb     // Pseudo: (qp) ltu     p0f = ra, rb


(qp)    eq      pt  = ra, imm9  // Pseudo: (qp) eq      pt0 = ra, imm9

(qp)    lt      pt = ra, imm9   // Pseudo: (qp) lt      pt0 = ra, imm9

(qp)    ltu     pt = ra, imm9   // Pseudo: (qp) ltu     pt0 = ra, imm9

(qp)    le      pt = ra, imm9   // Pseudo: (qp) lt      pt0 = ra, imm9+1

(qp)    leu     pt = ra, imm9   // Pseudo: (qp) ltu     pt0 = ra, imm9+1

(qp)    ne      pf = ra, imm9   // Pseudo: (qp) eq      p0f = ra, imm9

(qp)    gt      pf = ra, imm9   // Pseudo: (qp) lt      p0f = ra, imm9+1

(qp)    gtu     pf = ra, imm9   // Pseudo: (qp) ltu     p0f = ra, imm9+1

(qp)    ge      pf = ra, imm9   // Pseudo: (qp) lt      p0f = ra, imm9

(qp)    geu     pf = ra, imm9   // Pseudo: (qp) ltu     p0f = ra, imm9
```

31. Predicate Logic Instructions

(qp) and ptf = pa, pb // (qp) {pt←(pa)&(pb), pf←(nand)}

(qp) or ptf = pa, pb // (qp) {pt←(pa)|(pb), pf←(nor)}

(qp) xor ptf = pa, pb // (qp) {pt←(pa)^(pb), pf←(xnor)}

(qp) andc ptf = pa, pb // (qp) {pt←(pa)&~(pb), pf←(orc)}

(qp) nand ptf = pa, pb // Pseudo: (qp) and pft = pa, pb

(qp) nor ptf = pa, pb // Pseudo: (qp) or pft = pa, pb

(qp) xnor ptf = pa, pb // Pseudo: (qp) xor pft = pa, pb

(qp) orc ptf = pa, pb // Pseudo: (qp) andc pft = pb, pa

(qp) mov ptf = pa // Pseudo: (qp) and ptf = pa, pa

(qp) not ptf = pa // Pseudo: (qp) and pft = pa, pa

32. Double Precision Floating-Point Compare Instructions (R-Type Only)

(qp) eq.d ptf = ra, rb // if (qp) {pt←(ra)$=_d$(rb), pf←($\neq_d$)}

(qp) lt.d ptf = ra, rb // if (qp) {pt←(ra)$<_d$(rb), pf←($\geq_d$)}

(qp) ne.d ptf = ra, rb // Pseudo: (qp) eq.d pft = ra, rb

(qp) le.d ptf = ra, rb // Pseudo: (qp) lt.d pft = rb, ra

(qp) gt.d ptf = ra, rb // Pseudo: (qp) lt.d ptf = rb, ra

(qp) ge.d ptf = ra, rb // Pseudo: (qp) lt.d pft = ra, rb

33. Compare Instructions Formats

The R-type and I-type compare instruction formats are shown in figure 28. The 5-bit destination register field **rd** is now replaced by two 3-bit predicate fields: **pt** and **pf**. The major opcode field is reduced from 6 to 5 bits. Opcodes 0 to 2 are reserved for the I-type compare format, while opcode 3 is used for the R-type. The same function field **f** is used for the R-type and I-type formats. The other fields appear in their exact same position as in all R-type and I-type instructions. For predicate logic, the source predicate registers **pa** and **pb** replace register fields **ra** and **rb**.

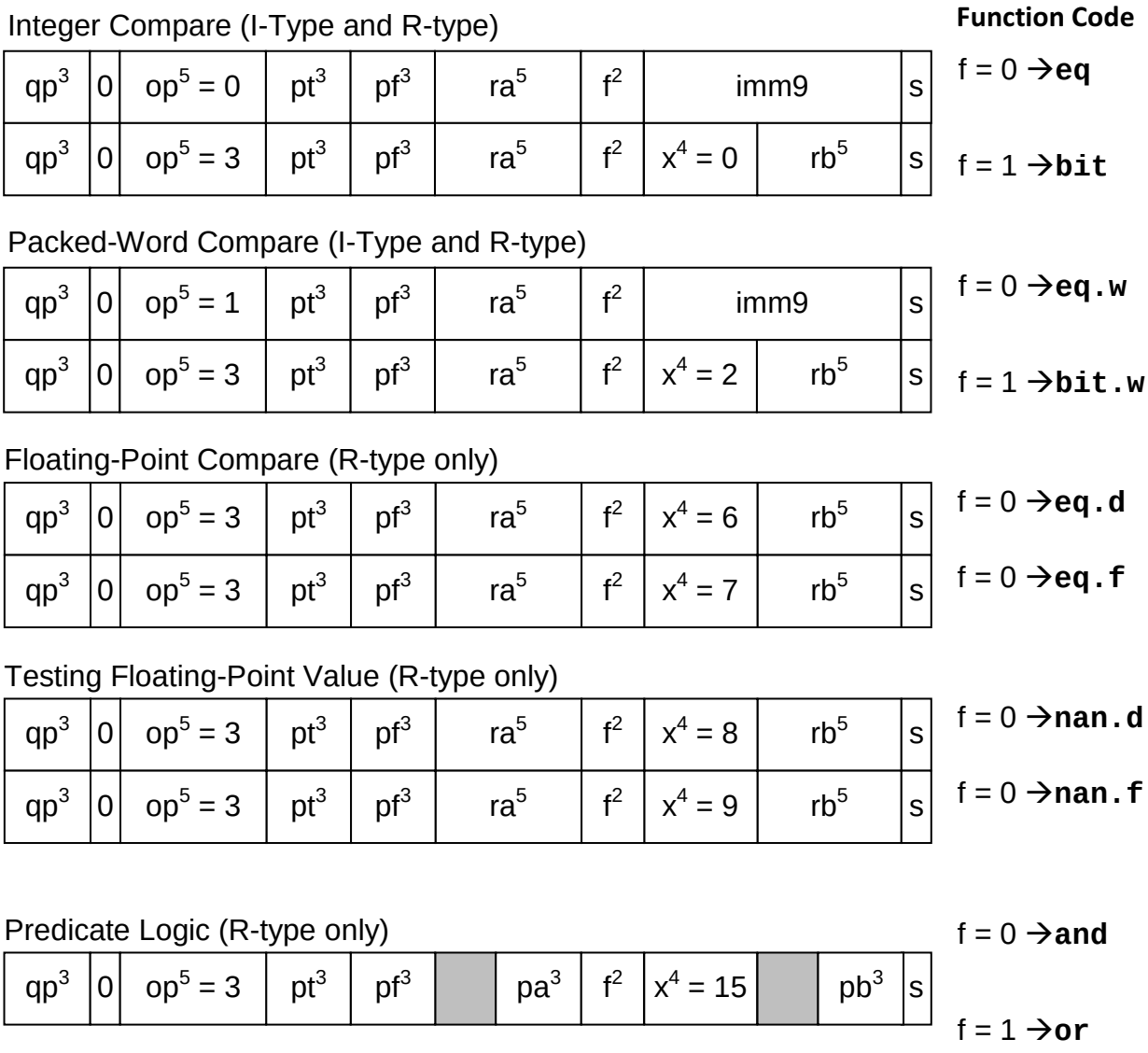

Figure 28: R-type and I-type Compare Instruction Formats

34. NOP Pseudo-Instruction

null // pseudo: eq p00 = r0, 0 (NULL instruction)

35. Load and Store Instructions

(qp) ld8 rd = ra[rb] // (qp) {rd $\leftarrow_{d8}$ MEM[ra + rb]}

(qp) ld4 rd = ra[rb] // (qp) {rd $\leftarrow_{d4}$ MEM[ra + rb] }

(qp) ld2 rd = ra[rb] // (qp) {rd $\leftarrow_{d2}$ MEM[ra + rb] }

(qp) ld1 rd = ra[rb] // (qp) {rd $\leftarrow_{d1}$ MEM[ra + rb] }

(qp) st8 ra[rb] = rd // (qp) {MEM[ra+ rb] $\leftarrow_{8}$ (rd)}

(qp) st4 ra[rb] = rd // (qp) {MEM[ra+ rb] $\leftarrow_{4}$ (rd) }

(qp) st2 ra[rb] = rd // (qp) {MEM[ra+ rb] $\leftarrow_{2}$ (rd) }

(qp) st1 ra[rb] = rd // (qp) {MEM[ra+ rb] $\leftarrow_{1}$ (rd) }

Appendix B Benchmarks C++ Source Code

1. Dense Matrix-Matrix Multiplication (DMMM)

```
const int N  = 1000;

double A[N][N]; // Define N x N matrix
double B[N][N]; // Define N x N matrix
double C[N][N]; // Define N x N matrix


for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
        for(int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Figure 29: DMMM C++ Source Code

2. Jacobi Iterative Method (JIM)

```
const int N = 1000; // Number of unknowns

const int LIMIT = 100; // Number of iterations

int A[N][N]; // Coefficients matrix

int B[N]; // Constants array

int X[N]; // Unkowns array
int new_X[N]; // Updated unkowns array

float sum;

//Initialize the unknowns array
for(int i = 0; i < N; i++)
{
    X[i] = B[i];
}
```

Figure 30: JIM C++ Source Code

```
for (int iteration = 0; iteration < LIMIT; iteration++)
{
    for (int i = 0; i < N; i++)
    {
        sum = 0;

        //PAR core's thread
        for (int j = 0; j < N; j++)
        {
            if (i != j)
            {
                sum = sum + A[i][j] * X[j];
                new_X[i] = (B[i] - sum) / A[i][i]; // Update unknown
            }
        }
    }
}

// Copy the updated unknowns into the unknowns' array
for (int i = 0; i < N; i++)
{
    X[i] = new_X[i];
}
```

Figure 30: JIM C++ Source Code (*Continued*) 1

3. Gauss-Seidel (GS): Red-Black Gauss-Seidel on a 2D Grid

```
const int N = 1000;
double const TOL = 0.6;  // Tolerance declaration
float A[N+2][N+2]; // Matrix "A" declaration
float diff; // To store the absolute difference
float temp; // To backup the current value of the point
bool done = false;  // To tell if convergence occured or not
int evenOrOdd;

while(!done)
{
    diff = 0;

    //Red sweep
    for(int i=1; i<=N; i++)
    {
        evenOrOdd = (i+1)%2;

        for(int j=evenOrOdd; j<=N; j+=2)
        {
            temp = A[i][j];
            A[i][j] = 0.2 *(A[i][j] + A[i][j-1] + A[i-1][j] + A[i][j+1] + A[i+1][j]);
            diff += fabs(A[i][j] - temp);
        }
    }
```

Figure 31: GS C++ Source Code

```
    //Black sweep
    for(int i=1; i<=N; i++)
    {
        evenOrOdd = i%2;
        for(int j=evenOrOdd; j<=N; j+=2)
        {
            //The body of this loop is one thread in PAR core
            temp = A[i][j];
            A[i][j] = 0.2 *(A[i][j] + A[i][j-1] + A[i-1][j] + A[i][j+1] + A[i+1][j]);
            diff += fabs(A[i][j] - temp);
        }
    }

    if(diff/(N*N) < TOL)
    {
        done = true;
    }
}
```

Figure 31: GS C++ Source Code (*Continued*) 1

4. RGB to YIQ Conversion (RGB-YIQ)

```
const int IMAGE_WIDTH = 320; // In pixels
const int IMAGE_HEIGHT = 240; // In pixels

float R [IMAGE_WIDTH][IMAGE_HEIGHT]; // Image declaration
float B [IMAGE_WIDTH][IMAGE_HEIGHT];
float G [IMAGE_WIDTH][IMAGE_HEIGHT];


float Y [IMAGE_WIDTH][IMAGE_HEIGHT];
float I [IMAGE_WIDTH][IMAGE_HEIGHT];
float Q [IMAGE_WIDTH][IMAGE_HEIGHT];


//Conversion coefficients
const float C0 = 0.299;
const float C1 = 0.587;
const float C2 = 0.114;
const float C3 = 0.596;
const float C4 = 0.275;
const float C5 = 0.321;
const float C6 = 0.212;
const float C7 = 0.523;
const float C8 = 0.311;
```

Figure 32: RGB-YIQ C++ Source Code

```
//RGB-YIC Conversion
for(int i = 0; i < IMAGE_HEIGHT; i++)
{
    for(int j = 0; j < IMAGE_WIDTH; j++)
    {
        Y[i][j] = C0 * R[i][j] + C1 * G[i][j] + C2 * B[i][j];

        I[i][j] = C3 * R[i][j] - C4 * G[i][j] - C5 * B[i][j];

        Q[i][j] = C6 * R[i][j] - C7 * G[i][j] + C8 * B[i][j];
    }
}
```

Figure 32: RGB-YIQ C++ Source Code (*Continued*) 1

5. RGB to CMYK Conversion (RGB-CMYK)

```
const int IMAGE_WIDTH = 320; // In pixels
const int IMAGE_HEIGHT = 240; // In pixels

short R [IMAGE_WIDTH][IMAGE_HEIGHT]; // Image declaration
short B [IMAGE_WIDTH][IMAGE_HEIGHT];
short G [IMAGE_WIDTH][IMAGE_HEIGHT];

short C [IMAGE_WIDTH][IMAGE_HEIGHT];
short M [IMAGE_WIDTH][IMAGE_HEIGHT];
short Y [IMAGE_WIDTH][IMAGE_HEIGHT];
short K [IMAGE_WIDTH][IMAGE_HEIGHT];
```

Figure 33: RGB-CMYK C++ Source Code

```
//RGB-CMYK Conversion
for(int i = 0; i < IMAGE_HEIGHT; i++)
{
    for(int j = 0; j < IMAGE_WIDTH; j++)
    {
        //Calculate complementary colors
        C[i][j] = 255 - R[i][j];
        M[i][j] = 255 - G[i][j];
        Y[i][j] = 255 - B[i][j];

        //Find the black level
        K[i][j] = Minimum (C[i][j] , M[i][j] , Y[i][j]);

        //Correct complementary color lever based on k
        C[i][j] = C[i][j] - K[i][j];
        M[i][j] = M[i][j] - K[i][j];
        Y[i][j] = Y[i][j] - K[i][j];
    }
}
```

Figure 33: RGB-CMYK C++ Source Code (*Continued*) 1

6. High Pass Grey-Scale Filter (HPF)

```
const int IMAGE_WIDTH = 320; // In pixels
const int IMAGE_HEIGHT = 240; // In pixels
short Image [IMAGE_WIDTH][IMAGE_HEIGHT]; // Image declaration
short output;
short pelValue;

//Coefficients
const short C = -28;
const short F22 = 255;

//Process the image
for(int i = 0; i < IMAGE_HEIGHT; i++)
{
    for(int j = 0; j < IMAGE_WIDTH; j++)
    {
        //Process one pixel
        pelValue = C * Image[i-1][j-1] + C * Image[i][j-1] + C * Image[i+1][j-1]
                 + C * Image[i-1][j] + F22 * Image[i][j] + C * Image[i+1][j]
                 + C * Image[i-1][j+1] + C * Image[i][j+1] + C * Image[i+1][j+1];

        output = pelValue >>8;

    }
}
```

Figure 34: HPF C++ Source Code

7. Scaled Vector Addition (SVA)

```
const int N  = 1000;

const double a = 0.5;
const double b = 0.7;


double V1[N]; // Define a vector with N elements
double V2[N]; // Define a vector with N elements
double V3[N]; // Define a vector with N elements

for(int i = 0; i < N; i++)
{
    //SVA's thread
    V3[i] = a * V1[i] + b * V2[i];
}
```

Figure 35: SVA C++ Source Code

Appendix C Benchmarks PAR Assembly Source Code

This appendix contains the PAR assembly source code of the data parallel benchmarks that have been used in this research. There is a part related to the main thread which contains the par instruction, this part has been added in the source code for clarification. To compile these benchmarks on the current version of ParSim, the block of the main thread should be removed.

1. Dense Matrix-Matrix Multiplication (DMMM)

```
//Dense Matrix-Matrix Multiplication (DMMM)

//main DMMM thread

mainDMMM:

.
.
.

par  r4, DMMMThread#



//This is the PAR packet content

.PAR

.ADDRESS = 0

.THREADS = 1280

.i0 = 0

.i1 = 0     //&A

.i2 = 1000  //&B

.i3 = 2000  //&C

.i4 = 1000  //N
```

Figure 36: DMMM Source Code

```
.CODE

DMMMThread: set r10 = 0

        rem  r2 =   i0,  i4  // r2 = the address of the first element of the current column

        sub  r1 =   i0,  r2  //r1 = the address of the first element of the current row

        loop i4,  dotProduct //multiply a row with a column

        st8  i3[i0]  =   r10#        //Store the value of the computed element.



dotProduct: ld8   r4 =  i1[r1] //Load A[i][k]

          ld8   r5   =  i2[r2]  //Load B[k][j]

          mac.d  r10  =  r4,  r5 // r10 = r10 + A[i][k] * B[k][j]

          add   r1   =  r1,  1

          add   r2   =  r2,  i4#
```

Figure 36: DMMM Source Code (*Continued*) 1

2. Jacobi Iterative Method (JIM)

```
//Jacobi Iterative Method (JIM)

//main JIM thread

mainJIM:

.
.
.

par  r7, JIMThread#


//The PAR packet content

.PAR

.ADDRESS = 0

.THREADS = 1280  // The number of threads

.i0 = 0   //The thread index

.i1 = 0   //The base address of matrix A

.i2 = 1638400  //The base address of array X

.i3 = 1639680  //The base address of array new_X

.i4 = 1640960  //The base address of array B

.i5 = 1280     // The number of unkowns
```

Figure 37: JIM Source Code

```
//This is the Jacobi iterative method thread which is responsible for updating
//one unknown

.CODE


JIMThread:

set r5 = 0 //r5 stores the sum, initialize it to zero

mul r1 = i0, i5 //r1 is the index within matrix A

mov r2 = i2     //r2 is the index within array X

loop i5, updateSum

ld8 r9 = i4[i0]  // r9 = B[i]

sub.d  r10 = r9, r5 // r10 = new_X[i]

div.d  r10 = r10,  r8  //r10 = new_X[i]/A[i][i]

st8 i3[i0] = r10#      // save the new_X[i]
```

Figure 37: JIM Source Code (*Continued*) 1

```
updateSum: div r11 = r1 , i5 // r11 contains the index within the ith row in the matrix A

eq p12 = r11, r2 // if(i != j)

(p1) ld8  r8 = il[r1] // r8 = A[i][i]

(p2) ld8 r6 =  il[r1] //r6 = A[i][j]

(p2) ld8 r7 =  i2[r2] //r7 = X[i]

(p2) mac.d r5 = r6, r7  // sum = A[i][j] * X[i]

add r1 = r1, 1  //increment the index

add r2 = r2, 1# //increment the index
```

Figure 37: JIM Source Code (*Continued*) 2

3. Gauss-Seidel (GS): Red-Black Gauss-Seidel on a 2D Grid

```
//Red-Black Gauss-Seidel on a 2D Grid (GS)

//main GS thread

mainGS:

.
.
.

par  r2, GSThread#



//The PAR packet content

.PAR

.ADDRESS = 0

.THREADS = 1280 // The number of threads

.i0 = 1  //The index starts at one becuase the actuall matriz size is (N + 2)(N + 2)

         //and the algorithm works on the interior N x N points


.i1 = 0    //i1 = &A

.i2 = 1000 //i2 = N
```

Figure 38: GS Source Code

```
//The Gauess Seidel thread aims at updating one black point and the difference variable

.CODE

GSThread:

mov r1 = i0 // r1 is the index of the matrix A

mul r1 = r1, 2  // To update the black points only (To reflect the alternating nature of the red-black)

ld8 r2 = i1[r1] // r2 = A[i][j]

mov r3 = r2 // r3 backups the value of r2


add r7 = i2, 2 // r7 = N+2


//Add the value of A[i][j+1]
add r5 = r1 , 1 //Compute the address of A[i][j+1]

ld8 r6 = i1[r5] // load the value of A[i][j+1]

add.d r2 = r2, r6 // Add the value of A[i][j+1] to r2


//Add the value of A[i][j-1]
sub r5 = r1 , 1 //Compute the address of A[i][j-1]

ld8 r6 = i1[r5] // load the value of A[i][j-1]

add.d r2 = r2, r6 // Add the value of A[i][j-1] to r2
```

Figure 38: GS Source Code (*Continued*) 1

```
//Add the value of A[i-1][j]
sub r5 = r1 , r7 //Compute the address of A[i-1][j]

ld8 r6 = il[r5] // load the value of A[i-1][j]

add.d r2 = r2, r6 // Add the value if A[i-1][j] to r2


//Add the value of A[i+1][j]
add r5 = r1 , r7 //Compute the address of A[i+1][j]

ld8 r6 = il[r5] // load the value of A[i+1][j]

add.d r2 = r2, r6 // Add the value if A[i+1][j] to r2


div.d r2 = r2 , 5 // r2 = r2 x 0.2 to find the weighted average

st8 il[r1] = r2 // sTore the new value of A[i][j]


sub.d r2 = r2, r3 // find the diff

abs.d r2 = r2 // find the absolute value of the difference

add.d r4 = r4, r2# // Accumulate the absolute difference to r4
```

Figure 38: GS Source Code (*Continued*) 2

4. RGB to YIQ Conversion (RGB-YIQ)

```
//RGB to YIQ Conversion


//main YIQ thread

mainYIQ:

.
.
.

par  r15, YIQThread#


//The PAR packet contents

.PAR

.ADDRESS = 0

.THREADS = 1280 //The number of threads

.i0 = 0     // the thread index

.i1 = 0     //&R

.i2 = 76800  //&G

.i3 = 153600 //&B

.i4 = 230400 //&Y
```

Figure 39: RGB-YIQ Source Code

```
.i5 = 307200 //&I

.i6 = 384000 //&Q

.i7 = 0.299  //Conversion coefficient

.i8 = 0.587  //Conversion coefficient

.i9 = 0.114  //Conversion coefficient

.i10 = 0.596 //Conversion coefficient

.i11 = 0.275 //Conversion coefficient

.i12 = 0.321 //Conversion coefficient

.i13 = 0.212 //Conversion coefficient

.i14 = 0.523 //Conversion coefficient

.i15 = 0.311 //Conversion coefficient
```

Figure 39: RGB-YIQ Source Code (*Continued*) 1

```
//The RGB to YIQ conversion kernel code in PAR assembly
//This thread computes the ith elements of the arrays Y, I and Q

.CODE

YIQThread:

set r2 = 255 // Store the constant 255 in r2

ld8 r3 = i1[i0] //Load the ith element of R

ld8 r4 = i2[i0] //Load the ith element of G

ld8 r5 = i3[i0] //Load the ith element of B


//Compute the ith element of Y

mul r6 = i7, r3

mov r7 = r6

mul r6 = i8, r4

add.d r7 = r7, r6

mac.d r7 = i9, r5 //Now r7 contains the value of the ith element of Y

st8 i4[i0] = r7   //Store the ith element of Y
```

Figure 39: RGB-YIQ Source Code (*Continued*) 2

```
//Compute the ith element of I

mul r8 = il0, r3

mov r9 = r8

mul r8 = il1, r4

sub.d r9 = r9, r8

mul r8 = il2, r5

sub.d r9 = r9, r8 // Now r9 contains the value of the ith element of I

st8 i5[i0] = r9 //Store the ith element of I



//Compute the ith element of Q

mul r10 = il3, r3

mov r11 = r10

mul r10 = il4, r4

sub.d r11 = r11, r10

mac.d r11 = il5, r5 //Now r11 contains the value of the ith element of Q

st8 i6[i0] = r11#   //Store the ith element of Q
```

Figure 39: RGB-YIQ Source Code (*Continued*) 3

5. RGB to CMYK Conversion (RGB-CMYK)

```
//RGB to CMYK Conversion
//main CMYK thread

mainCMYK:

.
.
.

par  r7, CMYKThread#


//This is the PAR packet content

.PAR

.ADDRESS = 0

.THREADS = 1280  //The number of threads

.i0 = 0          //The thread index

.i1 = 0         //&R

.i2 = 76800     //&G

.i3 = 153600    //&B

.i4 = 230400    //&C

.i5 = 307200    //&M
```

Figure 40: RGB-CMYK Source Code

```
.i6 = 384000     //&Y

.i7 = 460800     //&K


//This is the worker thread of that converts one pixel from RGB to CMYK

.CODE

CMYKThread:

mov r1 = i0 // r1 is the index within the seven arrays R,G,B,C,M,Y and K

set r2 = 255 // Store the constant 255 in r2


ld8 r3 = i1[r1] // Load the ith value of R

ld8 r4 = i2[r1] // Load the ith value of G

ld8 r5 = i3[r1] // Load the ith value of B


//Subtract the RGB values from 255 which is stored in r2

sub r3 = r2, r3 //now r3 contains the c_value

sub r4 = r2, r4 //now r4 contains the m_value

sub r5 = r2, r5 //now r5 contains the y_value
```

Figure 40: RGB-CMYK Source Code (*Continued*) 1

```
min  r6 = r3, r4

min r6 = r6, r5 // r6 = K[i] and it contains the minimum value of r3, r4 and r5



sub r3 = r3, r6 // now r3 = the ith element of C

sub r4 = r4, r6 // now r4 = the ith element of M

sub r5 = r4, r6 // the ith element of Y

//Store the CMYK values

st8 i4[r1] = r3

st8 i5[r1] = r4

st8 i6[r1] = r5

st8 i7[r1] = r6#
```

Figure 40: RGB-CMYK Source Code (*Continued*) 2

6. High Pass Grey-Scale Filter (HPF)

```
//High Pass Grey-Scale Filter (HPF)
//main HPF thread

mainHPF:

.
.
.

par r4, HPFThread#


//The PAR packet content

.PAR

.ADDRESS = 0

.THREADS = 1280 // The number of threads

.i0 = 1        // The thread index

.i1 = 0        //The base address of the image matrix

.i2 = 320      //The width of the image

.i3 = 255      //F22 filter Coefficient value

.i4 = -28      //The value of the other filter coefficients
```

Figure 41: HPF Source Code

```
//This is the code of the HPF's thread

.CODE

HPFThread:

ld8 r2 = i1[i0] // Load the value of P(c) into r2

mul r2 = r2, i3 //Compute the value of F22*P(c)

mov r1 = i0     // r1 contains the address of the pixel value to be loaded


add r1 = r1, 1 //Compute the address of P(c+1)

ld8 r3 = i1[r1] //Load the value of P(c+1) into r3

mac r2 = r3, i4 // Accumulate the computed value F32*P(c+1) into r2


sub r1 = r1, 2 //Compute the address of P(c-1)

ld8 r3 = i1[r1] //Load the value of P(c-1) into r3

mac r2 = r3, i4 // Accumulate the computed value F12*P(c-1) into r2
```

Figure 41: HPF Source Code (*Continued*) 1

```
//Compute the address of P(c+w)

mov r1 = i0

add r1 = r1, i2

ld8 r3 = i1[r1] //Load the value of P(c+w) into r3

mac r2 = r3, i4 // Accumulate the computed value F23*P(c+w) into r2


add r1 = r1, 1 //Compute the address of P(c+w+1)

ld8 r3 = i1[r1] //Load the value of P(c+w+1) into r3

mac r2 = r3, i4 // Accumulate the computed value F33*P(c+w+1) into r2


sub r1 = r1, 2 //Compute the address of P(c+w-1)

ld8 r3 = i1[r1] //Load the value of P(c+w-1) into r3

mac r2 = r3, i4 // Accumulate the computed value F13*P(c+w-1) into r2
```

Figure 41: HPF Source Code (*Continued*) 2

```
//Compute the address of P(c-w)

mov r1 = i0

sub r1 = r1, i2

ld8 r3 = il[r1] //Load the value of P(c-w) into r3

mac r2 = r3, i4 // Accumulate the computed value F21*P(c-w) into r2


add r1 = r1, 1 //Compute the address of P(c-w+1)

ld8 r3 = il[r1] //Load the value of P(c-w+1) into r3

mac r2 = r3, i4 // Accumulate the computed value F31*P(c-w+1) into r2


sub r1 = r1, 2 //Compute the address of P(c-w-1)

ld8 r3 = il[r1] //Load the value of P(c-w-1) into r3

mac r2 = r3, i4 // Accumulate the computed value F11*P(c-w-1) into r2


srl r2 = r2, 8 //Shift the result right by 8 bits

st8 il[i0] = r2# //Store the output
```

Figure 41: HPF Source Code (*Continued*) 3

7. Scaled Vector Addition (SVA)

```
//Scaled Vector Addition (SVA)
//main SVA thread

mainSVA:

.
.
.

par  r5, SVAThread#

//The PAR packet content

.PAR

.ADDRESS = 0

.THREADS = 1280 //The number of threads

.i0 = 0    // The thread index

.i1 = 0    //&V1

.i2 = 1280 //&V2

.i3 = 2560 //&V3

.i4 = 5    //The value of the constant a

.i5 = 7    //The value of the constant b
```

Figure 42: SVA Source Code

```
//The scaled vector addition thread code computes V3[i] = a*V1[i] + b*V2[i]

.CODE

SVAThread:

mov r1 = i0 // r1 is the index within the three arrays A,B and C


ld8 r2 = i1[r1] // Load V1[i]

ld8 r3 = i2[r1] //Load V2[i]

muld r2 = r2, i4 //Compute a*V1[i]

muld r3 = r3, i5 //Compute b*V2[i]

add.d r4 = r2, r3 //Compute a*V1[i] + b*V2[i]

st8 i3[r1] = r4# // Store the value of r4 which contains V3[i]
```

Figure 42: SVA Source Code (*Continued*) 1

Appendix D Experimental Results

D.1 Results for Single Lane

Table 2: DMMM Performance Results for Single Lane

Number of Simultaneous Threads	Total Execution Time	IPC	FPU Throughput	ALU Utilization	L/S Unit Utilization
1	6406403	1	0.20	40%	39.98%
2	3203204	2	0.40	80%	79.96%
4	2562562	2.50	0.50	100%	99.95%
8	2562562	2.50	0.50	100%	99.95%

Table 3: JIM Performance Results for Single Lane

Number of Simultaneous Threads	Total Execution Time	IPC	FPU Throughput	ALU Utilization	Compare Unit Utilization	L/S Unit Utilization
1	3279365	0.87	0.25	25%	12.49%	37.49%
2	1639686	1.75	0.50	50%	24.98%	74.98%
4	1229448	2.33	0.67	66.68%	33.32%	100%
8	1229448	2.33	0.67	66.68%	33.32%	100%

Table 4: GS Performance Results for Single Lane

Number of Simultaneous Threads	Total Execution Time	IPC	FPU Throughput	ALU Utilization	L/S Unit Utilization
1	28165	1	0.36	36.36%	27.27%
2	14727	1.91	0.70	69.53%	52.15%
4	10248	2.75	1	99.92%	74.94%
8	10248	2.75	1	99.92%	74.94%

Table 5: RGB-YIQ Performance Results for Single Lane

Number of Simultaneous Threads	Total Execution Time	IPC	FPU Throughput	ALU Utilization	L/S Unit Utilization
1	34568	0.85	0.48	14.81%	22.22%
2	22407	1.3	0.74	22.85%	34.28%
4	16644	1.77	1	30.76%	46.14%
8	16644	1.77	1	30.76%	46.14%

Table 6: RGB-CMYK Performance Results for Single Lane

Number of Simultaneous Threads	Total Execution Time	IPC	FPU Throughput	ALU Utilization	L/S Unit Utilization
1	21762	1	0.00%	58.82%	41.17%
2	14724	1.48	0.00%	86.93%	60.85%
4	14405	1.51	0.00%	88.86%	62.20%
8	14249	1.53	0.00%	89.83%	62.88%

Table 7: HPF Performance Results for Single Lane

Number of Simultaneous Threads	Total Execution Time	IPC	FPU Throughput	ALU Utilization	L/S Unit Utilization
1	39684	1	0.29	38.71%	32.25%
2	19845	2	0.58	77.40%	64.50%
4	15364	2.58	0.75	99.97%	83.31%
8	15364	2.58	0.75	99.97%	83.31%

Table 8: SVA Performance Results for Single Lane

Number of Simultaneous Threads	Total Execution Time	IPC	FPU Throughput	ALU Utilization	L/S Unit Utilization
1	8967	1	0.43	14.27%	42.82%
2	5128	1.75	0.75	24.96%	74.88%
4	3848	2.33	1	33.26%	99.79%
8	3847	2.33	1	33.27%	99.82%

D.2 Results for Multiple Lanes

Table 9: DMMM Performance Results for Multiple Lanes

Number of Lanes	Execution Time	IPC	Speedup
1	2562562	2.50	
2	1281282	5	2
4	640642	10	4
8	320322	20	8
16	160162	39.99	16
32	80082	79.98	32
64	40042	159.96	64

Table 10: JIM Performance Results for Multiple Lanes

Number of Lanes	Execution Time	IPC	Speedup
1	4917768	2.33	
2	2458888	4.67	2
4	1229448	9.34	4
8	614728	18.67	8
16	307368	37.34	16
32	153688	74.68	32
64	76848	149.36	63.99

Table 11: GS Performance Results for Multiple Lanes

Number of Lanes	Execution Time	IPC	Speedup
1	51208	2.75	
2	25608	5.50	2
4	12808	10.99	4
8	6408	21.97	7.99
16	3208	43.89	15.96
32	1608	87.56	31.85
64	808	174.26	63.38

Table 12: RGB-YIQ Performance Results for Multiple Lanes

Number of Lanes	Execution Time	IPC	Speedup
1	83204	1.77	
2	41604	3.54	2
4	20804	7.08	4
8	10404	14.15	8
16	5204	28.29	16
32	2604	56.53	31.95
64	1304	112.88	63.80

Table 13: RGB-CMYK Performance Results for Multiple Lanes

Number of Lanes	Execution Time	IPC	Speedup
1	72005	1.51	
2	36005	3.02	2
4	18005	6.04	4
8	9005	12.08	8
16	4505	24.15	15.98
32	2255	48.25	31.93
64	1130	96.28	63.72

Table 14: HPF Performance Results for Multiple Lanes

Number of Lanes	Execution Time	IPC	Speedup
1	76804	2.58	
2	38404	5.17	2
4	19204	10.33	4
8	9604	20.66	8
16	4804	41.30	15.99
32	2404	82.53	31.95
64	1204	164.78	63.79

Table 15: SVA Performance Results for Multiple Lanes

Number of Lanes	Execution Time	IPC	Speedup
1	19208	2.33	
2	9608	4.66	2
4	4808	9.32	4
8	2408	18.60	7.98
16	1208	37.09	15.90
32	608	73.68	31.59
64	308	145.45	62.36

Bibliography

[1] D. Culler, J. P. Singh, and A. Gupta, "Introduction," in *Parallel Computer Architecture A Hardware / Software Approach*., 1997, ch. 1, p. 20.

[2] T. Ungerer, B. Robič, and J. Šilc, "Multithreaded Processors," *The Computer Journal*, pp. 320-348, 2002.

[3] A. ZMILY and C. KOZYRAKIS, "Block-Aware Instruction Set Architecture," *ACM Transactions on Architecture and Code Optimization*, pp. 327–357, 2006.

[4] M. Zahran and M. Franklin, "Dynamic thread resizing for speculative multithreaded processors," in *21st International Conference on Computer Design*, 2003, pp. 313-318.

[5] Il Park, B. Falsafi, and T.N. Vijaykumar, "Implicitly-multithreaded processors," in *30th Annual International Symposium on Computer Architecture*, 2003, pp. 39- 50.

[6] S. Wallace, B. Calder, and D.M. Tullsen, "Threaded multiple path execution," in *The 25th Annual International Symposium on Computer Architecture*, 1998, pp. 238-249.

[7] H. Akkary and M.A. Driscoll, "A dynamic multithreading processor," in *31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998, pp. 226-236.

[8] R. Kalla, Balaram Sinharoy, and J.M. Tendler, "IBM Power5 chip: a dual-core multithreaded processor," *IEEE Computer Society*, pp. 40- 47, 2004.

[9] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded

Sparc processor," *IEEE computer Society*, pp. 21- 29, 2005.

[10] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Computer Society*, pp. 39-55, 2008.

[11] F. Latorre, J. González, and A. González, "Efficient resources assignment schemes for clustered multithreaded processors," in *IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1-12.

[12] K. Li, H. Zhang, J. Li, and Y. Xie, "A multithreaded processor core with low overhead context switch for IP-packet processing," in *10th IEEE International Conference on Solid-State and Integrated Circuit Technology*, 2010, pp. 272-274.

[13] S. Torii, K. Kojimu, Y. Kana, A. Sakata, and S. Yoshizumi, "Accelerating non-numerical processing by an extended vector processor," in *Fourth International Conference on Data Engineering*, 1988, pp. 194-201.

[14] N. Ishiura, M. Ito, and S. Yajima, "Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 868-875, 1990.

[15] J.D Gee and A.J. Smith, "The performance impact of vector processor cashes," in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 1992, pp. 437-448.

[16] C. Kozyrakis and D. Patterson, "Overcoming the limitations of conventional vector processors," in *30th Annual International Symposium on Computer Architecture*, 2003, pp. 399-409.

[17] C.E. Kozyrakis and D.A. Patterson, "SCALABLE VECTOR PROCESSORS FOR EMBEDDED SYSTEMS," *IEEE Computer Society*, pp. 36- 45, 2003.

[18] R. Krashinsky et al., "The vector-thread architecture," in *31st Annual International Symposium on Computer Architecture*, 2004, pp. 52- 63.

[19] L. Spracklen, Yuan Chou, and S.G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *1th International Symposium on High-Performance Computer Architecture*, 2005, pp. 225- 236.

[20] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *32nd Annual International Symposium on Microarchitecture*, 1999, pp. 16-27.

[21] T.M. Aamodt, P. Chow, P. Hammarlund, Hong Wang, and J.P. Shen, "Hardware Support for Prescient Instruction Prefetch," in *10th International Symposium on High Performance Computer Architecture*, 2004, p. 84.

[22] M. Ferdman, T.F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 1-10.

[23] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero, "Fetching instruction streams," in *35th Annual IEEE/ACM International Symposium on Microarchitecture*,

2002, pp. 371- 382.

[24] O. J. Santana, A. Ramirez, and M. Valero, "Enlarging Instruction Streams," *IEEE Computer Society*, pp. 1342-1357, OCTOBER 2007.

[25] Liqiang He and Zhiyong Liu, "An effective instruction fetch policy for simultaneous multithreaded processors," in *Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region*, 2004, pp. 162- 168.

[26] E. Hao, P. Chang, M. Evers, and Y. N. Patt, "Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, 1996, pp. 191-200.

[27] J.J. Yi and D.J. Lilja, "Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations," *IEEE Transactions on Computers*, pp. 268-280, 2006.

[28] Sangyeun Cho et al., "TPTS: A Novel Framework for Very Fast Manycore Processor Architecture Simulation," in *37th International Conference on Parallel Processing*, 2008, pp. 446-453.

[29] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez, "Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors," in *19th International Symposium on Computer Architecture and High Performance Computing*, 2007, pp. 62-68.

[30] C. Barnes, P. Vaidya, and J.J. Lee, "An XML-Based ADL Framework for Automatic Generation of Multithreaded Computer Architecture Simulators," in *Computer Architecture Letters*, 2009, pp. 13-16.

[31] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *International Symposium on Computer Architecture - ISCA*, 2007, pp. 162-173.

[32] D. Culler, J. P. Singh, and A. Gupta, "Parallel Programs," in *Parallel Computer Architecture A Hardware / Software Approach*., 1997, ch. 2, pp. 104-110.

[33] The Embedded Microprocessor Benchmark Consortium. [Online]. http://www.eembc.org/home.php

[34] M. Watheq El-Kharashi, F. ElGuibaly, and K.F. Li, "Multithreaded processors: the upcoming generation for multimedia chips," in *IEEE Symposium on Advances in Digital Filtering and Signal Processing*, 1998, pp. 111-115.

Vitae

Ayman Ali Hroub was born on September 19th, 1984 in Nablus, Palestine. His nationality is Jordanian, but he lives in Palestine. He obtained his BSc degree from Birzeit University in Palestine in 2008. In September 2009, he has gotten a scholarship as a Research Assistant in Computer Engineering Department at King Fahd University of Petroleum and Minerals to work towards his MS degree in Computer Engineering. His research interests are focusing on Computer Architecture and Parallel Processing Systems.

Contact Information:

Present and Permanent Address: Kharas, Hebron, Palestine.

E-mail Address: ahroub@gmail.com.

Mobile Number: +966-59-7470756.